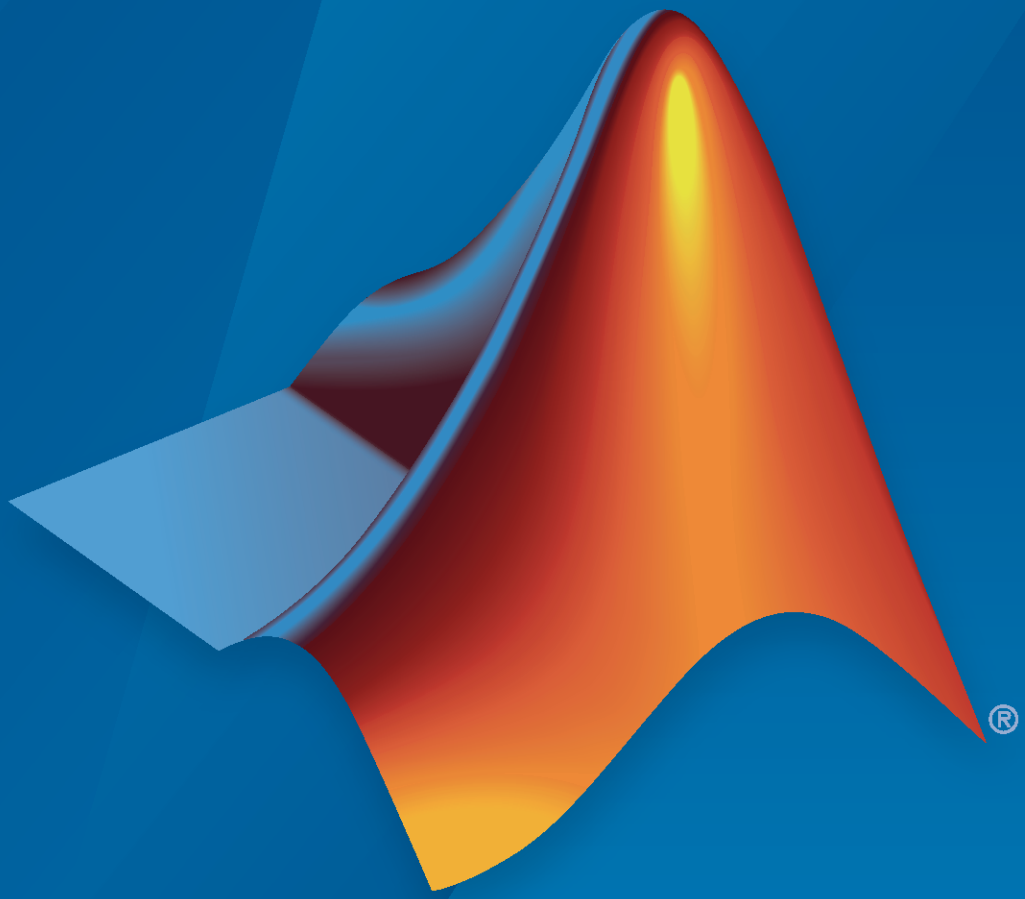
Simulink® Control Design™

User's Guide



MATLAB® & SIMULINK®

R2021b

MathWorks®

How to Contact MathWorks

Latest news: `www.mathworks.com`

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

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

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

Phone: 508-647-7000

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

Simulink® Control Design™ User's Guide

© COPYRIGHT 2004–2021 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)
March 2014	Online only	Revised for Version 4.0 (Release 2014a)
October 2014	Online only	Revised for Version 4.1 (Release 2014b)
March 2015	Online only	Revised for Version 4.2 (Release 2015a)
September 2015	Online only	Revised for Version 4.2.1 (Release 2015b)
March 2016	Online only	Revised for Version 4.3 (Release 2016a)
September 2016	Online only	Revised for Version 4.4 (Release 2016b)
March 2017	Online only	Revised for Version 4.5 (Release 2017a)
September 2017	Online only	Revised for Version 5.0 (Release 2017b)
March 2018	Online only	Revised for Version 5.1 (Release 2018a)
September 2018	Online only	Revised for Version 5.2 (Release 2018b)
March 2019	Online only	Revised for Version 5.3 (Release 2019a)
September 2019	Online only	Revised for Version 5.4 (Release 2019b)
March 2020	Online only	Revised for Version 5.5 (Release 2020a)
September 2020	Online only	Revised for Version 5.6 (Release 2020b)
March 2021	Online only	Revised for Version 5.7 (Release 2021a)
September 2021	Online only	Revised for Version 6.0 (Release 2021b)

Contents

1 Steady-State Operating Points

About Operating Points **1-2**
What Is an Operating Point? **1-2**
What Is a Steady-State Operating Point? **1-2**
Simulink Model States Included in Operating Point Object **1-3**

Compute Steady-State Operating Points **1-5**
Steady-State Operating Point Search (Trimming) **1-5**
Steady-State Operating Point from Simulation Snapshot **1-5**
Which Model States Must Be at Steady State? **1-6**
Choose Operating Point Search Tools **1-6**

View and Modify Operating Points **1-9**
View and Modify Operating Point in Steady State Manager **1-9**
View and Modify Operating Point in Model Linearizer **1-9**
View and Modify Operating Point at the Command Line **1-11**

Compute Steady-State Operating Points from Specifications **1-13**

Compute Operating Points from Specifications at the Command Line . . **1-15**

Compute Operating Points from Specifications Using Steady State Manager **1-20**
Open Steady State Manager **1-20**
Define Operating Point Specifications **1-21**
Trim Model **1-24**
Validate Operating Point **1-26**
Trim Model for Different Specifications **1-28**
Extract Operating Point from Report **1-29**
Export Operating Point Data **1-29**

Compute Operating Points from Specifications Using Model Linearizer **1-31**
Open Model Linearizer **1-31**
Define Operating Point Specifications **1-32**
Trim Model **1-34**
Constrain State Derivatives **1-35**

Validate Operating Point Against Specifications **1-38**
Validate Operating Point in Steady State Manager **1-38**
Validate Operating Point in Model Linearizer **1-42**
Validate Operating Point at the Command Line **1-42**

Initialize Steady-State Operating Point Search Using Simulation Snapshot **1-45**
Initialize Operating Point Search Using Steady State Manager **1-45**
Initialize Operating Point Search Using Model Linearizer **1-47**
Initialize Operating Point Search at the Command Line **1-49**

Change Operating Point Search Optimization Settings **1-51**
Interactively Change Optimization Settings **1-51**
Programmatically Change Optimization Settings **1-52**

Import and Export Specifications for Operating Point Search **1-53**
Import and Export Specification Using Steady State Manager **1-53**
Import and Export Specification Using Model Linearizer **1-55**

Compute Operating Points Using Custom Constraints and Objective Functions **1-57**

Batch Compute Steady-State Operating Points for Multiple Specifications **1-68**

Batch Compute Steady-State Operating Points for Parameter Variation **1-72**
Which Parameters Can Be Sampled? **1-72**
Vary Single Parameter **1-72**
Multidimensional Parameter Grids **1-73**
Vary Multiple Parameters **1-74**
Batch Trim Model for Parameter Variations **1-75**
Batch Trim Model at Known States Derived from Parameter Values **1-77**

Batch Compute Steady-State Operating Points Reusing Generated MATLAB Code **1-80**

Find Operating Points at Simulation Snapshots **1-83**
Compute Operating Points at Simulation Snapshots Using Steady State Manager **1-83**
Compute Operating Points at Simulation Snapshots Using Model Linearizer **1-84**
Find Operating Points at Simulation Snapshots at Command Line **1-86**

Compute Operating Point Snapshots at Triggered Events **1-88**

Simulate Simulink Model at Specific Operating Point **1-93**
Set Model Operating Point Using Steady State Manager **1-93**
Set Model Operating Point Using Model Linearizer **1-94**

Handle Blocks with Internal State Representation **1-96**
Operating Point Object Excludes Blocks with Internal States **1-96**
Configure Blocks with Internal States for Steady-State Operating Point Search **1-97**

Synchronize Simulink Model Changes with Operating Point Specifications **1-98**
Synchronize Model Changes Using Steady State Manager **1-98**
Synchronize Model Changes Using Model Linearizer **1-100**
Synchronize Model Changes at the Command Line **1-101**

Find Steady-State Operating Points for Simscape Models **1-104**
Projection-Based Trim Optimizers **1-104**

Steady-State Simulation with Projection-Based Trim Optimizer **1-106**

Generate MATLAB Code for Operating Point Configuration **1-110**
Generate MATLAB Code from Steady State Manager **1-110**
Generate MATLAB Code from Model Linearizer **1-111**

2 Linearization

Linearize Nonlinear Models **2-3**
What Is Linearization? **2-3**
Applications of Linearization **2-4**
Linearization in Simulink Control Design **2-4**
Model Requirements for Exact Linearization **2-5**
Operating Point Impact on Linearization **2-5**

Choose Linearization Tools **2-7**
Choosing Simulink Control Design Linearization Tools **2-7**
Choosing Exact Linearization Versus Frequency Response Estimation **2-7**
Linearization Using Simulink Control Design Versus Simulink **2-8**

Specify Portion of Model to Linearize **2-10**
Analysis Points **2-10**
Opening Feedback Loops **2-14**
Ways to Specify Portion of Model to Linearize **2-15**

Specify Portion of Model to Linearize in Simulink Model **2-17**
Specify Analysis Points **2-17**
Select Bus Elements as Analysis Points **2-18**

Specify Portion of Model to Linearize in Model Linearizer **2-22**
Specify Analysis Points **2-22**
Edit Analysis Points **2-25**
Edit Simulink Model Analysis Points **2-27**

Specify Portion of Model to Linearize at Command Line **2-29**
Specify Analysis Points **2-29**
Save Analysis Points in Simulink Model **2-30**
Obtain Analysis Points from Simulink Model **2-30**

How the Software Treats Loop Openings **2-31**

Linearize Plant **2-33**
Linearize Plant Using Model Linearizer **2-33**
Linearize Plant at Command Line **2-35**

Mark Signals of Interest for Control System Analysis and Design **2-38**
Analysis Points **2-38**
Specify Analysis Points for MATLAB Models **2-39**

Specify Analysis Points for Simulink Models 2-39
Refer to Analysis Points for Analysis and Tuning 2-42

Compute Open-Loop Response 2-46
Compute Open-Loop Response Using Model Linearizer 2-48
Compute Open-Loop Response at the Command Line 2-51

Linearize Simulink Model at Model Operating Point 2-54
Linearize Simulink Model Using Model Linearizer 2-54
Linearize Simulink Model at Command Line 2-58

Visualize Bode Response of Simulink Model During Simulation 2-61

Linearize at Trimmed Operating Point 2-67

Linearize at Simulation Snapshot 2-72

Linearize at Triggered Simulation Events 2-75

Linearize Models with Delays 2-78

Linearize Models with Model References 2-82

Visualize Linear System at Multiple Simulation Snapshots 2-85

Visualize Linear System of a Continuous-Time Model Discretized During Simulation 2-91

Plot Linear System Characteristics of a Chemical Reactor 2-95

Order States in Linearized Model 2-102
Control State Order of Linearized Model using Model Linearizer 2-102
Control State Order of Linearized Model using MATLAB Code 2-105

Validate Linearization In Time Domain 2-107
Validate Linearization in Time Domain 2-107
Choosing Time-Domain Validation Input Signal 2-107

Validate Linearization In Frequency Domain Using Model Linearizer 2-108
Linearize Model 2-108
Estimate Frequency Response of Model 2-108
Examine estimation results. 2-109

View Linearized Model Equations Using Model Linearizer 2-111

Analyze Results Using Model Linearizer Response Plots 2-113
View System Characteristics on Response Plots 2-113
Generate Additional Response Plots of Linearized System 2-114
Add Linear System to Existing Response Plot 2-116
Customize Characteristics of Plot in Model Linearizer 2-118
Print Plot to MATLAB Figure in Model Linearizer 2-118

Generate MATLAB Code for Linearization from Model Linearizer 2-120

When to Specify Individual Block Linearization **2-122**

Specify Linear System for Block Linearization Using MATLAB Expression **2-123**

Specify D-Matrix System for Block Linearization Using Function **2-124**

Specify Custom Linearizations for Simulink Blocks **2-128**

Augment Block Linearization **2-133**

Models with Time Delays **2-137**
Choose Approximate Versus Exact Time Delays **2-137**
Specify Exact Representation of Time Delays **2-137**

Linearize Multirate Models **2-139**
Change Sample Time of Linear Model **2-139**
Change Linearization Rate Conversion Method **2-139**
Multirate Linearization Algorithm **2-140**

Linearize Models Using Different Rate Conversion Methods **2-145**

Change Perturbation Level of Blocks Perturbed During Linearization **2-148**

Linearize Blocks with Non-Floating-Point Signals or States **2-150**
Override Data Type Using Data Type Conversion Block **2-150**
Overriding Data Types Using Fixed Point Tool **2-150**

Linearize Event-Based Subsystems (Externally Scheduled Subsystems) **2-152**
Linearizing Event-Based Subsystems **2-152**
Approaches for Linearizing Event-Based Subsystems **2-152**
Approximate Event-Based Subsystems Using Curve Fitting (Lump-Average Model) **2-152**
Approximate Event-Based Dynamics Using Periodic Function Call Subsystem **2-154**

Configure Models with Pulse Width Modulation Signals **2-158**

Linearize Simscape Networks **2-160**
Find Steady-State Operating Point **2-160**
Specify Analysis Points **2-160**
Linearize Model **2-160**
Troubleshoot Simscape Network Linearizations **2-161**

Linearize Sparse Models **2-164**
Linearize Sparse Models at the Command Line **2-165**
Linearize Sparse Models Using Model Linearizer **2-166**
Limitations **2-167**

Specify Linearization for Model Components Using System Identification **2-168**

Exact Linearization Algorithm **2-175**
Continuous-Time Models **2-175**

Multirate Models 2-176
Perturbation of Individual Blocks 2-179
User-Defined Blocks 2-180
Look Up Tables 2-180

Trim and Linearize an Airframe 2-181

Linearize Pneumatic System at Simulation Snapshots 2-186

Linearize Pulp Paper Process Model 2-190

3 Batch Linearization

What Is Batch Linearization? 3-2

Choose Batch Linearization Methods 3-4
Choose Batch Linearization Tool 3-5

Batch Linearization Efficiency When You Vary Parameter Values 3-7
Tunable and Nontunable Parameters 3-7
Controlling Model Recompilation 3-7

Mark Signals of Interest for Batch Linearization 3-9
Analysis Points 3-9
Specify Analysis Points 3-10
Refer to Analysis Points 3-12

Batch Linearize Model for Parameter Variations at Single Operating Point 3-13

Batch Linearize Model at Multiple Operating Points Derived from Parameter Variations 3-16

Batch Linearize Model at Multiple Operating Points Using linearize Command 3-19

Vary Parameter Values and Obtain Multiple Transfer Functions 3-21

Vary Operating Points and Obtain Multiple Transfer Functions Using slLinearizer Interface 3-28

Analyze Command-Line Batch Linearization Results Using Response Plots 3-33

Analyze Batch Linearization Results in Model Linearizer 3-39

Specify Parameter Samples for Batch Linearization 3-43
About Parameter Samples 3-43
Which Parameters Can Be Sampled? 3-43
Vary Single Parameter at the Command Line 3-43

Vary Single Parameter in Graphical Tools 3-44
Multi-Dimension Parameter Grids 3-47
Vary Multiple Parameters at the Command Line 3-48
Vary Multiple Parameters in Graphical Tools 3-50

Batch Linearize Model for Parameter Value Variations Using Model Linearizer 3-53

More Efficient Batch Linearization Varying Parameters 3-65

Validate Batch Linearization Results 3-69

Approximate Nonlinear Behavior Using Array of LTI Systems 3-70

LPV Approximation of Boost Converter Model 3-83

Linearize Engine Speed Model 3-93

Improve Linear Analysis Performance 3-97

4 Troubleshooting Linearization Results

Linearization Troubleshooting Overview 4-2
Troubleshooting Workflow 4-2
Troubleshoot Linearizations of Models with Special Characteristics 4-3

Check Operating Point 4-4

Check Analysis Point Placement 4-5
Check Linearization I/O Points Placement 4-5
Check Loop Opening Placement 4-5

Identify and Fix Common Linearization Issues 4-6
Enable Linearization Advisor 4-6
Blocks That Are Potentially Problematic for Linearization 4-9
Find Specific Blocks in Linearization Results 4-11
Linearization Path 4-11
Troubleshoot Batch Linearizations 4-13

Troubleshoot Linearization Results in Model Linearizer 4-16

Troubleshoot Linearization Results at Command Line 4-28

Find Blocks in Linearization Results Matching Specific Criteria 4-37
Run Built-In Queries 4-37
Create and Run Queries 4-38

Block Linearization Troubleshooting 4-42
Diagnostic Messages 4-43
Linearization Summary 4-44

Block Linearization 4-45
Block Operating Point 4-45
Common Problematic Blocks 4-45

Speed Up Linearization of Complex Models 4-48
Factors That Impact Linearization Performance 4-48
Blocks with Complex Initialization Functions 4-48
Disabling the Linearization Advisor in the Model Linearizer 4-48
Batch Linearization of Large Simulink Models 4-48

5 Frequency Response Estimation

Frequency Response Estimation Basics 5-2
Frequency Response Models 5-2
Offline and Online Estimation 5-3
Basic Estimation Workflow 5-3
Model Requirements 5-4

Estimate Frequency Response Using Model Linearizer 5-6

Estimate Frequency Response with Linearization-Based Input Using Model Linearizer 5-10

Estimate Frequency Response at the Command Line 5-14

Analyze Estimated Frequency Response 5-18
View Simulation Results 5-18
Interpret Frequency Response Estimation Results 5-19
Analyze Simulated Output and FFT at Specific Frequencies 5-21
Annotate Frequency Response Estimation Plots 5-21
Displaying Estimation Results for Multiple-Input Multiple-Output (MIMO) Systems 5-22
Result Thinning 5-23

Estimation Input Signals 5-24
Offline Estimation 5-24
Online Estimation 5-25
Sinestream Signals 5-25
Chirp Signals 5-25
PRBS Signals 5-25
Random Signals 5-25
Step Signals 5-26
Arbitrary Signals 5-26
Superposition Signals 5-27

Sinestream Input Signals 5-29
Create Sinestream Signals Using Model Linearizer 5-31
Create Sinestream Signals Using MATLAB Code 5-31
Sinestream Signals for Online Estimation 5-31

Chirp Input Signals **5-33**
Create Chirp Signals Using Model Linearizer **5-33**
Create Chirp Signals Using MATLAB Code **5-35**

PRBS Input Signals **5-36**
Create PRBS Signals Using Model Linearizer **5-37**
Create PRBS Signals Using MATLAB Code **5-38**
Improve Frequency Response Result at Low Frequencies **5-39**

Modify Estimation Input Signals **5-40**
Modify Sinestream Signal Using Model Linearizer **5-40**
Modify Sinestream Signal Using MATLAB Code **5-42**

Troubleshooting Frequency Response Estimation **5-43**
When to Troubleshoot **5-43**
Time Response Not at Steady State **5-43**
FFT Contains Large Harmonics at Frequencies Other than the Input Signal Frequency **5-45**
Time Response Grows Without Bound **5-46**
Time Response Is Discontinuous or Zero **5-47**
Time Response Is Noisy **5-49**
Time Response Shows Harmonics That Do Not Change Smoothly **5-51**

Effects of Time-Varying Source Blocks on Frequency Response Estimation **5-53**
Setting Time-Varying Sources to Constant for Estimation Using Model Linearizer **5-53**
Setting Time-Varying Sources to Constant for Estimation (MATLAB Code) **5-58**
Limitations **5-59**

Disable Noise Sources During Frequency Response Estimation **5-60**

Estimate Frequency Response Models with Noise Using Signal Processing Toolbox **5-63**

Estimate Frequency Response Models with Noise Using System Identification Toolbox **5-65**

Generate MATLAB Code for Repeated or Batch Frequency Response Estimation **5-67**

Managing Estimation Speed and Memory **5-68**
Ways to Speed up Frequency Response Estimation **5-68**
Speeding Up Estimation Using Parallel Computing **5-69**
Managing Memory During Frequency Response Estimation **5-71**

Frequency Response Estimation Using Simulation-Based Techniques . . **5-74**

Validate Linearization in Frequency Domain at Command Line **5-80**

Describing Function Analysis of Nonlinear Simulink Models **5-84**

Speed Up Frequency Response Estimation Using Parallel Computing . . **5-89**

Frequency Response Estimation for Power Electronics Model Using Pseudorandom Binary Signal **5-94**

Frequency Response Estimation in Model Linearizer Using Pseudorandom Binary Sequence **5-101**

Frequency Response Estimation for Permanent Magnet Synchronous Motor Model **5-113**

Frequency Response Estimation to Measure Input Admittance of Boost Converter **5-124**

6 Online Frequency Response Estimation

Online Frequency Response Estimation Basics **6-2**
When Not to Use Online Frequency-Response Estimation **6-2**
System Configurations for Online Frequency Response Estimation **6-2**
Estimation Workflow **6-3**

Online Estimation Using Plant Modeled in Simulink **6-5**
Workflow for Online Estimation in Simulink **6-5**
Step 1. Incorporate Frequency Response Estimator into Model **6-5**
Step 2. Configure Start/Stop Signal **6-6**
Step 3. Set Experiment Parameters **6-7**
Step 4. Run Model and Examine Estimated Frequency Response **6-7**

Deploy Frequency Response Estimation Algorithm for Real-Time Use . . . **6-9**
Workflow **6-9**
Step 1. Create Deployable Simulink Model with Frequency Response Estimator Block **6-9**
Step 2. Configure Start/Stop Signal **6-12**
Step 3. Set Experiment Parameters **6-12**
Step 4. Run Experiment **6-13**
Access Experiment Parameters After Deployment **6-13**

Online Frequency Response Estimation During Simulation **6-15**

Collect Frequency Response Experiment Data for Offline Estimation . . **6-18**

Online Estimation of Frequency Responses of a Nonlinear Plant **6-22**

7 PID Controller Tuning

Introduction to Model-Based PID Tuning in Simulink **7-2**
What Plant Does PID Tuner See? **7-2**
PID Tuning Algorithm **7-3**

Open PID Tuner **7-5**
Prerequisites for PID Tuning **7-5**
Opening PID Tuner **7-5**

Analyze Design in PID Tuner **7-8**
Plot System Responses **7-8**
View Numeric Values of System Characteristics **7-11**
Export Plant or Controller to MATLAB Workspace **7-11**
Refine the Design **7-12**

Verify the PID Design in Your Simulink Model **7-13**

Tune at a Different Operating Point **7-14**
Known State Values Yield the Desired Operating Conditions **7-14**
Model Reaches Desired Operating Conditions at a Finite Time **7-14**
You Computed an Operating Point in Model Linearizer **7-15**

Tune PID Controller to Favor Reference Tracking or Disturbance Rejection **7-17**

Design Two-Degree-of-Freedom PID Controllers **7-26**
About Two-Degree-of-Freedom PID Controllers **7-26**
Tuning Two-Degree-of-Freedom PID Controllers **7-26**
Fixed-Weight Controller Types **7-27**

Tune PID Controller Within Model Reference **7-30**
Models with Multiple Instances of the Referenced Model **7-31**
Referenced Model in Accelerated or Other Simulation Modes **7-32**

Specify PI-D and I-PD Controllers **7-33**
Specify PI-D and I-PD Controllers Using PID Controller (2DOF) Block . . . **7-34**
Automatic Tuning of PI-D and I-PD Controllers **7-34**

Design PID Controller from Plant Frequency-Response Data **7-36**
Use Frequency Response Based PID Tuner **7-36**
Use frestimate or Model Linearizer **7-36**

Frequency-Response Based Tuning **7-37**
How Frequency Response Based PID Tuner Works **7-37**
Open Frequency Response Based PID Tuner **7-37**
Configure Experiment Settings **7-39**
Configure Design Goals **7-41**
Tune and Validate Controller Gains **7-41**

Design PID Controller Using Plant Frequency Response Near Bandwidth **7-43**

Import Measured Response Data for Plant Estimation **7-51**

Interactively Estimate Plant from Measured or Simulated Response Data **7-55**

System Identification for PID Control **7-61**
Plant Identification **7-61**
Linear Approximation of Nonlinear Systems for PID Control **7-61**

Linear Process Models 7-62
Advanced System Identification Tasks 7-62

Preprocess Data 7-64
Ways to Preprocess Data 7-64
Remove Offset 7-64
Scale Data 7-65
Extract Data 7-65
Filter Data 7-65
Resample Data 7-65
Replace Data 7-66

Input/Output Data for Identification 7-67
Data Preparation 7-67
Data Preprocessing 7-67

Choosing Identified Plant Structure 7-68
Process Models 7-68
State-Space Models 7-71
Existing Plant Models 7-72
Switching Between Model Structures 7-73
Estimating Parameter Values 7-74
Handling Initial Conditions 7-74

Design Controller for Power Electronics Model Using Frequency Response Data 7-76

Design Controller for Power Electronics Model Using Simulated I/O Data 7-94
Boost Converter Model 7-94
Find Model Operating Point 7-95
Specify Controller Structure 7-97
Identify Plant Model 7-98
Tune Controller 7-102
Validate Controller 7-105

Design PID Controller Using Simulated I/O Data 7-108

Design PID Controller Using Estimated Frequency Response 7-124

Design Family of PID Controllers for Multiple Operating Points 7-132

Implement Gain-Scheduled PID Controllers 7-139

Design Controller for Vehicle Platooning 7-145

Plant Cannot Be Linearized or Linearizes to Zero 7-153
How to Fix It 7-153

Cannot Find a Good Design in PID Tuner 7-154
How to Fix It 7-154

Simulated Response Does Not Match PID Tuner Response 7-155

Cannot Find Acceptable PID Design in Simulated Model **7-157**
How to Fix It **7-157**

Controller Performance Deteriorates When Switching Time Domains **7-158**
How to Fix It **7-158**

When Tuning the PID Controller, the D Gain Has a Different Sign from the I Gain **7-159**

Tune Field-Oriented Controllers Using SYSTUNE **7-160**

Frequency Response Based PID Tuner **7-175**
Experiment Settings **7-175**
Design Specifications **7-177**
Automatically Update Block **7-178**
Tune and Cancel **7-178**
Tuning Results **7-178**

8 PID Autotuning

When to Use PID Autotuning **8-2**
PID Autotuning for a Physical Plant **8-2**
PID Autotuning for a Plant Model in Simulink **8-2**
Closed-Loop vs. Open-Loop PID Autotuning **8-2**
When Not to Use PID Autotuning **8-3**

How PID Autotuning Works **8-5**
Autotuning Process **8-5**
Workflow for PID Autotuning **8-6**

PID Autotuning for a Plant Modeled in Simulink **8-7**
Workflow for Autotuning in Simulink **8-7**
Step 1. Incorporate Autotuner into Model **8-7**
Step 2. Configure Start/Stop Signal **8-9**
Step 3. Specify Controller Parameters and Tuning Goals **8-9**
Step 4. Set Experiment Parameters **8-10**
Step 5. Run Model and Initiate Tuning Experiment **8-11**
Step 6. Stop Experiment and Examine Tuned Gains **8-11**
Step 7. Update PID Controller with Tuned Gains **8-11**

PID Autotuning in Real Time **8-13**
Workflow **8-13**
Step 1. Create Deployable Simulink Model with PID Autotuner Block . . . **8-13**
Step 2. Configure Start/Stop Signal **8-16**
Step 3. Set PID Tuning Parameters **8-16**
Step 4. Set Experiment Parameters **8-17**
Step 5. Tune and Validate **8-18**
Access Autotuning Parameters After Deployment **8-18**

Control Real-Time PID Autotuning in Simulink **8-20**
Simulink Model for External-Mode Tuning **8-20**

Run the Model and Tune the Controller Gains 8-21
Reduce Memory Footprint When Using External Mode 8-22

Tune PID Controller in Real Time Using Open-Loop PID Autotuner Block 8-23

Tune PID Controller in Real Time Using Closed-Loop PID Autotuner Block 8-29

BLDC Motor Speed Control with Cascade PI Controllers 8-36

Tune Field-Oriented Controllers Using Closed-Loop PID Autotuner Block 8-47

Tune Field-Oriented Controllers for an Asynchronous Machine Using Closed-Loop PID Autotuner Block 8-54

Tune Field-Oriented Controllers for a PMSM Using Closed-Loop PID Autotuner Block 8-62

Design PID Controllers for Three-Phase Rectifier using Closed-Loop PID Autotuner Block 8-69

PID Autotuning for UAV Quadcopter 8-75

Tune Gain-Scheduled Controller Using Closed-Loop PID Autotuner Block 8-88

9 Classical Control Design

Choose a Control Design Approach 9-2
Design in Simulink 9-2
Real-Time PID Autotuning 9-3

Control System Designer Tuning Methods 9-4
Graphical Tuning Methods 9-4
Automated Tuning Methods 9-4
Effective Plant for Tuning 9-5
Tuning Compensators In Simulink 9-6
Select a Tuning Method 9-6

What Blocks Are Tunable? 9-8

Designing Compensators for Plants with Time Delays 9-9

Design Compensator Using Automated PID Tuning and Graphical Bode Design 9-11

Analyze Designs Using Response Plots 9-27
Analysis Plots 9-27

Editor Plots 9-29
Plot Characteristics 9-30
Plot Tools 9-32
Design Requirements 9-33

Compare Performance of Multiple Designs **9-36**

Update Simulink Model and Validate Design **9-40**

Single Loop Feedback/Prefilter Compensator Design **9-41**

Cascaded Multiloop Feedback Design **9-49**

Tune Custom Masked Subsystems **9-59**

Tune Simulink Blocks Using Compensator Editor **9-67**

Reference Tracking of DC Motor with Parameter Variations **9-74**

Regulate Pressure in Drum Boiler **9-79**

Model Computational Delay and Sampling Effects **9-86**

10 Control System Tuning

Automated Tuning Overview **10-3**

Choosing an Automated Tuning Approach **10-4**

Automated Tuning Workflow **10-6**

Specify Control Architecture in Control System Tuner **10-7**
About Control Architecture **10-7**
Predefined Feedback Architecture **10-7**
Arbitrary Feedback Control Architecture **10-8**
Control System Architecture in Simulink **10-9**

Open Control System Tuner for Tuning Simulink Model **10-10**
Command-Line Equivalents **10-10**

Specify Operating Points for Tuning in Control System Tuner **10-11**
About Operating Points in Control System Tuner **10-11**
Linearize at Simulation Snapshot Times **10-11**
Compute Operating Points at Simulation Snapshot Times **10-13**
Compute Steady-State Operating Points **10-15**

Specify Blocks to Tune in Control System Tuner **10-17**

View and Change Block Parameterization in Control System Tuner . . . **10-19**
View Block Parameterization **10-19**

Fix Parameter Values or Limit Tuning Range 10-21
Custom Parameterization 10-22
Block Rate Conversion 10-23

Setup for Tuning Control System Modeled in MATLAB 10-26

How Tuned Simulink Blocks Are Parameterized 10-27
Blocks With Predefined Parameterization 10-27
Blocks Without Predefined Parameterization 10-28
View and Change Block Parameterization 10-28

Specify Goals for Interactive Tuning 10-29

Quick Loop Tuning of Feedback Loops in Control System Tuner 10-34

Quick Loop Tuning 10-42
Purpose 10-42
Description 10-42
Feedback Loop Selection 10-42
Desired Goals 10-43
Options 10-44
Algorithms 10-44

Step Tracking Goal 10-45
Purpose 10-45
Description 10-45
Step Response Selection 10-45
Desired Response 10-46
Options 10-47
Algorithms 10-48

Step Rejection Goal 10-50
Purpose 10-50
Description 10-50
Step Disturbance Response Selection 10-51
Desired Response to Step Disturbance 10-51
Options 10-52
Algorithms 10-52

Transient Goal 10-54
Purpose 10-54
Description 10-54
Response Selection 10-54
Initial Signal Selection 10-55
Desired Transient Response 10-56
Options 10-56
Tips 10-57
Algorithms 10-57

LQR/LQG Goal 10-59
Purpose 10-59
Description 10-59
Signal Selection 10-59
LQG Objective 10-60
Options 10-61

Tips 10-61
Algorithms 10-61

Gain Goal 10-63
Purpose 10-63
Description 10-63
I/O Transfer Selection 10-64
Options 10-64
Algorithms 10-65

Variance Goal 10-67
Purpose 10-67
Description 10-67
I/O Transfer Selection 10-67
Options 10-68
Tips 10-69
Algorithms 10-69

Reference Tracking Goal 10-71
Purpose 10-71
Description 10-71
Response Selection 10-72
Tracking Performance 10-72
Options 10-73
Algorithms 10-74

Overshoot Goal 10-76
Purpose 10-76
Description 10-76
Response Selection 10-77
Options 10-77
Algorithms 10-78

Disturbance Rejection Goal 10-80
Purpose 10-80
Description 10-80
Disturbance Scenario 10-81
Rejection Performance 10-82
Options 10-82
Algorithms 10-83

Sensitivity Goal 10-85
Purpose 10-85
Description 10-85
Sensitivity Evaluation 10-86
Sensitivity Bound 10-86
Options 10-86
Algorithms 10-87

Weighted Gain Goal 10-89
Purpose 10-89
Description 10-89
I/O Transfer Selection 10-89
Weights 10-90
Options 10-90

Algorithms 10-91

Weighted Variance Goal 10-92
Purpose 10-92
Description 10-92
I/O Transfer Selection 10-92
Weights 10-93
Options 10-93
Tips 10-94
Algorithms 10-94

Minimum Loop Gain Goal 10-96
Purpose 10-96
Description 10-96
Open-Loop Response Selection 10-97
Desired Loop Gain 10-97
Options 10-98
Algorithms 10-99

Maximum Loop Gain Goal 10-101
Purpose 10-101
Description 10-101
Open-Loop Response Selection 10-102
Desired Loop Gain 10-102
Options 10-103
Algorithms 10-104

Loop Shape Goal 10-106
Purpose 10-106
Description 10-106
Open-Loop Response Selection 10-107
Desired Loop Shape 10-108
Options 10-108
Algorithms 10-109

Margins Goal 10-111
Purpose 10-111
Description 10-111
Feedback Loop Selection 10-112
Desired Margins 10-112
Options 10-113
Algorithms 10-114

Passivity Goal 10-115
Purpose 10-115
Description 10-115
I/O Transfer Selection 10-116
Options 10-116
Algorithms 10-117

Conic Sector Goal 10-119
Purpose 10-119
Description 10-119
I/O Transfer Selection 10-120
Options 10-120

Tips 10-121
Algorithms 10-122

Weighted Passivity Goal 10-124
Purpose 10-124
Description 10-124
I/O Transfer Selection 10-125
Weights 10-125
Options 10-126
Algorithms 10-126

Poles Goal 10-128
Purpose 10-128
Description 10-128
Feedback Configuration 10-129
Pole Location 10-129
Options 10-130
Algorithms 10-130

Controller Poles Goal 10-132
Purpose 10-132
Description 10-132
Constrain Dynamics of Tuned Block 10-133
Keep Poles Inside the Following Region 10-133
Algorithms 10-133

Manage Tuning Goals 10-135

Generate MATLAB Code from Control System Tuner for Command-Line Tuning 10-136

Interpret Numeric Tuning Results 10-139
Tuning-Goal Scalar Values 10-139
Tuning Results at the Command Line 10-139
Tuning Results in Control System Tuner 10-140
Improve Tuning Results 10-141

Visualize Tuning Goals 10-143
Tuning-Goal Plots 10-143
Difference Between Dashed Line and Shaded Region 10-144
Improve Tuning Results 10-148

Create Response Plots in Control System Tuner 10-149

Examine Tuned Controller Parameters in Control System Tuner 10-155

Compare Performance of Multiple Tuned Controllers 10-157

Create and Configure slTuner Interface to Simulink Model 10-161

Stability Margins in Control System Tuning 10-165
Gain and Phase Margins 10-165
Interpret Gain and Phase Margin Plots 10-165
Simultaneous Gain and Phase Variations 10-166
Algorithm 10-167

Tune Control System at the Command Line **10-170**

Speed Up Tuning with Parallel Computing Toolbox Software **10-171**

Validate Tuned Control System **10-172**
Extract and Plot System Responses **10-172**
Validate Design in Simulink Model **10-173**

Extract Responses from Tuned MATLAB Model at the Command Line **10-176**

11 Gain-Scheduled Controllers

Gain Scheduling Basics **11-2**
Gain Scheduling in Simulink **11-2**
Tune Gain Schedules **11-2**

Model Gain-Scheduled Control Systems in Simulink **11-4**
Model Scheduled Gains **11-4**
Gain-Scheduled Equivalents for Commonly Used Control Elements **11-6**
Custom Gain-Scheduled Control Structures **11-9**
Tunability of Gain Schedules **11-10**

Tune Gain Schedules in Simulink **11-12**
Workflow for Tuning Gain Schedules **11-12**

Plant Models for Gain-Scheduled Controller Tuning **11-14**
Obtaining the Family of Linear Models **11-15**
Set Up for Gain Scheduling by Linearizing at Design Points **11-15**
Sample System at Simulation Snapshots **11-18**
Sample System at Varying Parameter Values **11-18**
Eliminate Samples at Unneeded Design Points **11-19**
LPV Plants in MATLAB **11-19**

Multiple Design Points in slTuner Interface **11-20**
Block Substitution for Plant **11-20**
Multiple Block Substitutions **11-20**
Substituting Blocks that Depend on the Scheduling Variables **11-21**
Resolving Mismatches Between a Block and its Substitution **11-22**
Block Substitution for LPV Blocks **11-23**

Parameterize Gain Schedules **11-24**
Basis Function Parameterization **11-24**
Tunable Gain Surfaces **11-26**
Tunable Gain with Two Independent Scheduling Variables **11-27**
Tunable Surfaces in Simulink **11-29**
Tunable Surfaces in MATLAB **11-31**

Change Requirements with Operating Condition **11-33**
Define Variable Tuning Goal **11-33**
Enforce Tuning Goal at Subset of Design Points **11-35**

Exclude Design Points from systune Run 11-35

Validate Gain-Scheduled Control Systems 11-36
Examine Tuned Gain Surfaces 11-36
Visualize Tuning Goals 11-36
Check Linear Performance 11-39
Validate Gain Schedules in Nonlinear System 11-39

Gain-Scheduled Control of a Chemical Reactor 11-41

Tuning of Gain-Scheduled Three-Loop Autopilot 11-55

Trimming and Linearization of the HL-20 Airframe 11-68

Angular Rate Control in the HL-20 Autopilot 11-75

Attitude Control in the HL-20 Autopilot - SISO Design 11-81

Attitude Control in the HL-20 Autopilot - MIMO Design 11-91

MATLAB Workflow for Tuning the HL-20 Autopilot 11-99

12 Loop-Shaping Design

Structure of Control System for Tuning With looptune 12-2

Set Up Your Control System for Tuning with looptune 12-3
Set Up Your Control System for looptunein MATLAB 12-3
Set Up Your Control System for looptune in Simulink 12-3

Tune MIMO Control System for Specified Bandwidth 12-4

Decoupling Controller for a Distillation Column 12-10

Tuning of a Digital Motion Control System 12-21

13 Control System Tuning Examples

Tuning Multiloop Control Systems 13-2

PID Tuning for Setpoint Tracking vs. Disturbance Rejection 13-11

Time-Domain Specifications 13-20

Frequency-Domain Specifications 13-26

Loop Shape and Stability Margin Specifications **13-34**

System Dynamics Specifications **13-40**

Configuring Design Requirements **13-42**

Validating Results **13-43**

Tune Control Systems in Simulink **13-51**

Tune a Control System Using Control System Tuner **13-59**

Using Parallel Computing to Accelerate Tuning **13-75**

Control of a Linear Electric Actuator **13-79**

Control of a Linear Electric Actuator Using Control System Tuner . . . **13-89**

Multi-Loop PI Control of a Robotic Arm **13-114**

Control of an Inverted Pendulum on a Cart **13-129**

Digital Control of Power Stage Voltage **13-136**

MIMO Control of Diesel Engine **13-145**

Tuning of a Two-Loop Autopilot **13-158**

Multiloop Control of a Helicopter **13-173**

Fixed-Structure Autopilot for a Passenger Jet **13-180**

Fault-Tolerant Control of a Passenger Jet **13-191**

Passive Control of Water Tank Level **13-200**

Tuning for Multiple Values of Plant Parameters **13-213**

14 Control System Tuning Applications

UAV Inflight Failure Recovery **14-2**

15 Adaptive Control

Extremum Seeking Control **15-2**

Time Domain **15-2**

Static Optimization 15-3
Dynamic System Optimization 15-6
ESC Design Guidelines 15-7

Extremum Seeking Control for Reference Model Tracking of Uncertain Systems **15-8**

Anti-Lock Braking Using Extremum Seeking Control **15-15**

Adaptive Cruise Control Using Extremum Seeking Control **15-21**

Model Reference Adaptive Control **15-28**
Nominal and Reference Models 15-29
Control Structure 15-29
Uncertainty and Error Dynamics 15-30
Parameter Updates 15-30
Learning Modification 15-31
Disturbance and Uncertainty Model Features 15-31

Model Reference Adaptive Control of Satellite Spin **15-33**

Model Reference Adaptive Control of Aircraft Undergoing Wing Rock **15-41**

16 Constraint Enforcement

Constraint Enforcement for Control Design **16-2**
Constraint Enforcement Block 16-2
Constraint Function Coefficients 16-2

Enforce Constraints for PID Controllers **16-4**

Learn and Apply Constraints for PID Controllers **16-10**

Train Reinforcement Learning Agent with Constraint Enforcement . . **16-18**

Train RL Agent for Adaptive Cruise Control with Constraint Enforcement **16-29**

Train RL Agent for Lane Keeping Assist with Constraint Enforcement **16-39**

17 Model Verification

Monitor Linear System Characteristics in Simulink Models **17-2**

Define Linear System for Model Verification Blocks **17-3**

Verifiable Linear System Characteristics **17-4**

Verify Model at Default Simulation Snapshot Time **17-5**

Verify Model at Multiple Simulation Snapshots **17-13**

Verify Model Using Simulink Control Design and Simulink Verification Blocks **17-20**

Verify Frequency-Domain Characteristics of an Aircraft **17-27**

18 Functions

19 Blocks

20 Objects

21 Model Advisor Checks

Simulink Control Design Checks **21-2**

Identify time-varying source blocks interfering with frequency response estimation **21-2**

22 Apps

Steady-State Operating Points

- “About Operating Points” on page 1-2
- “Compute Steady-State Operating Points” on page 1-5
- “View and Modify Operating Points” on page 1-9
- “Compute Steady-State Operating Points from Specifications” on page 1-13
- “Compute Operating Points from Specifications at the Command Line” on page 1-15
- “Compute Operating Points from Specifications Using Steady State Manager” on page 1-20
- “Compute Operating Points from Specifications Using Model Linearizer” on page 1-31
- “Validate Operating Point Against Specifications” on page 1-38
- “Initialize Steady-State Operating Point Search Using Simulation Snapshot” on page 1-45
- “Change Operating Point Search Optimization Settings” on page 1-51
- “Import and Export Specifications for Operating Point Search” on page 1-53
- “Compute Operating Points Using Custom Constraints and Objective Functions” on page 1-57
- “Batch Compute Steady-State Operating Points for Multiple Specifications” on page 1-68
- “Batch Compute Steady-State Operating Points for Parameter Variation” on page 1-72
- “Batch Compute Steady-State Operating Points Reusing Generated MATLAB Code” on page 1-80
- “Find Operating Points at Simulation Snapshots” on page 1-83
- “Compute Operating Point Snapshots at Triggered Events” on page 1-88
- “Simulate Simulink Model at Specific Operating Point” on page 1-93
- “Handle Blocks with Internal State Representation” on page 1-96
- “Synchronize Simulink Model Changes with Operating Point Specifications” on page 1-98
- “Find Steady-State Operating Points for Simscape Models” on page 1-104
- “Steady-State Simulation with Projection-Based Trim Optimizer” on page 1-106
- “Generate MATLAB Code for Operating Point Configuration” on page 1-110

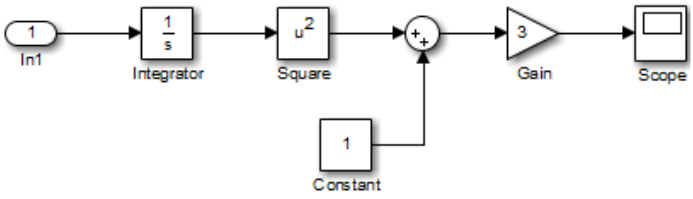
About Operating Points

What Is an Operating Point?

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

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

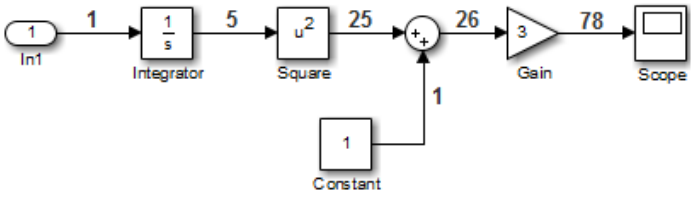
- A root-level input signal set to 1
- An Integrator block state set to 5



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

Block	Block Input	Block Operation	Block Output
Integrator	1	Integrate input	5, set by the initial condition`x0 = 5`
Square	5, set by the initial condition of the Integrator block	Square input	25
Sum	25 from Square block, 1 from Constant block	Sum inputs	26
Gain	26	Multiply input by 3	78

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



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

What Is a Steady-State Operating Point?

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

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

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

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

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

Simulink Model States Included in Operating Point Object

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

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

State Type	Included in Operating Point?
Double-precision real-valued states	Yes
States whose value is not of type `double`. For example, complex-valued states, `single`-type states, `int8`-type states.	No
States from root-level inport blocks with double-precision real-valued inputs	Yes
Internal state representations that impact block output, such as states in Backlash, Memory, or Stateflow blocks.	No (see “Handle Blocks with Internal State Representation” on page 1-96)
States that belong to a Unit Delay block whose input is a bus signal	No

See Also

`operpoint`

More About

- “Compute Steady-State Operating Points” on page 1-5
- “Handle Blocks with Internal State Representation” on page 1-96
- “Compute Steady-State Operating Points” on page 1-5
- “Find Operating Points at Simulation Snapshots” on page 1-83

Compute Steady-State Operating Points

An *operating point* of a dynamic system specifies the initial states and root-level input signals of the model at a particular time. For more information on operating points, see “About Operating Points” on page 1-2.

To find steady-state operating points you can use optimization-based searching or simulation snapshots.

Steady-State Operating Point Search (Trimming)

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

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

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

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

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

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

Steady-State Operating Point from Simulation Snapshot

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

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

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

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

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

state and use the resulting simulation snapshot to define the initial conditions for an optimization-based search.

Note If your Simulink model has internal states, do not linearize the model at an operating point you compute from a simulation snapshot. Instead, try linearizing the model using a simulation snapshot or at an operating point from optimization-based search. For more information, see “Handle Blocks with Internal State Representation” on page 1-96.

Which Model States Must Be at Steady State?

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

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

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

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

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

Choose Operating Point Search Tools

Simulink Control Design lets you search for operating points of your Simulink model both programmatically at the command line and interactively using one of two apps.

Search Tool	When to Use
`findop`	• Programmatically compute operating points • Compute operating points from specifications • Find operating points at simulation snapshots • Batch compute operating points for multiple specifications • Batch compute operating points for parameter variations

Search Tool	When to Use
Steady State Manager	• Interactively compute operating points • Compute operating points from specifications • Validate operating point search results against specifications • Find operating points at simulation snapshots • Generate MATLAB® code for computing operating points. This code can be reused for batch computation.
Model Linearizer	• Interactively find operating points within a linearization context • Compute operating points from specifications • Find operating points at simulation snapshots

Trimming Using Simulink Control Design Versus Simulink

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

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

See Also

`findop` | `trim`

More About

- “About Operating Points” on page 1-2
- “Handle Blocks with Internal State Representation” on page 1-96
- “Compute Steady-State Operating Points” on page 1-5
- “Find Operating Points at Simulation Snapshots” on page 1-83

- “Initialize Steady-State Operating Point Search Using Simulation Snapshot” on page 1-45

View and Modify Operating Points

You can view and modify operating point values programmatically at the command line, or interactively using the **Steady State Manager** or **Model Linearizer**.

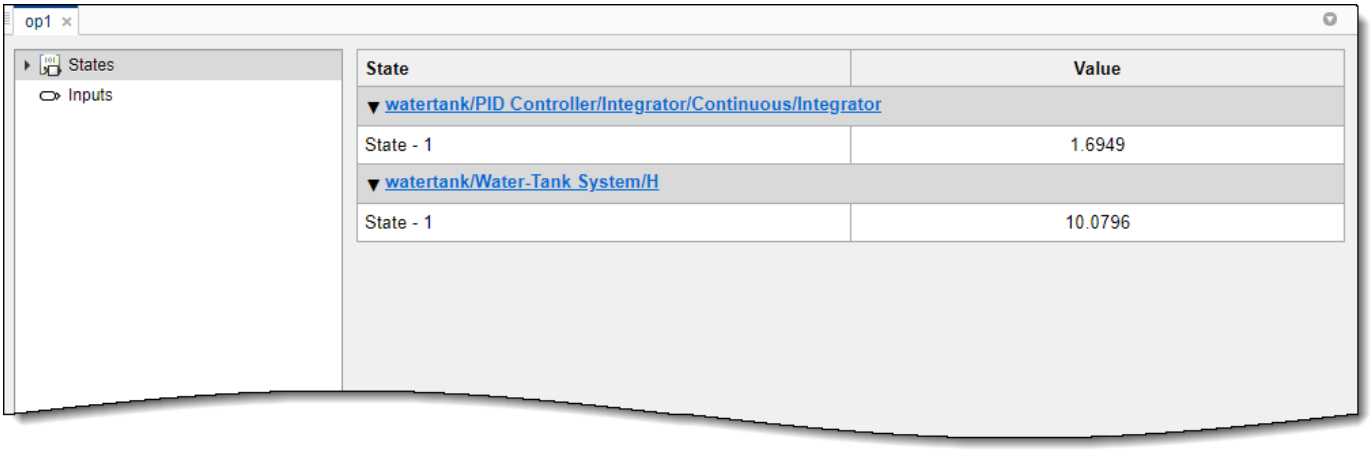
You can simulate the model at the modified operating point. For more information, see “Simulate Simulink Model at Specific Operating Point” on page 1-93.

View and Modify Operating Point in Steady State Manager

To view an operating point in the **Steady State Manager**, in the **Data Browser**, in the **Operating Points** section do one of the following:

- Double-click the operating point you want to view.
- Right-click the operating point you want to view, and select **Open Selection**.

In the operating point document that opens, you can view the input and state values of the operating point.

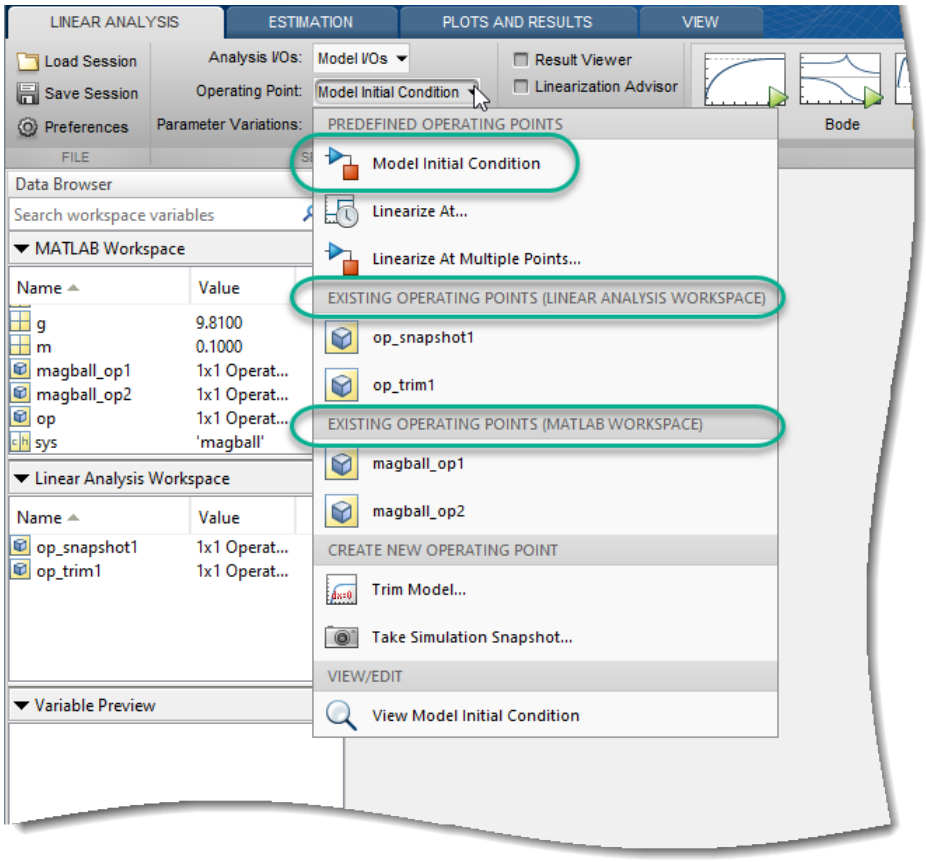


To modify a state or input value in an operating point, in the **Value** column, click the value you want to change, and enter the new value. If your operating point was at a steady state, changing any values in the **Steady State Manager** can place the operating point into a non-steady-state condition.

View and Modify Operating Point in Model Linearizer

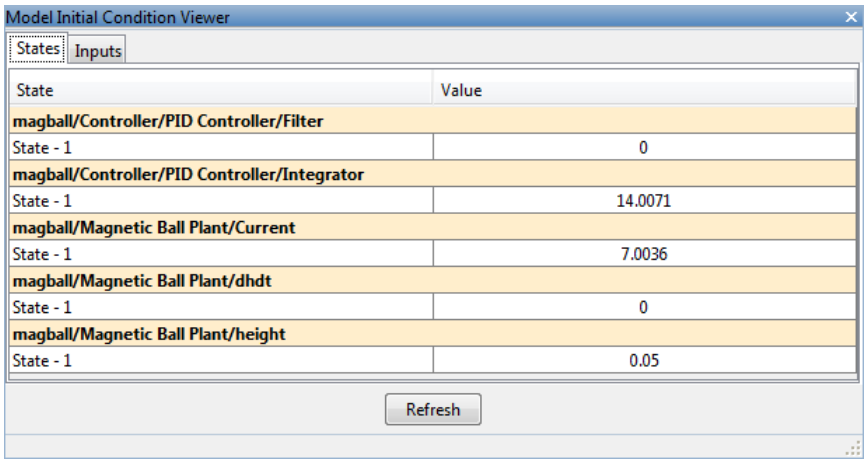
To view an operating point in the **Model Linearizer**, on the **Linear Analysis** tab, in the **Operating Points** drop-down list, select one of the following:

- `Model Initial Condition` — The current states and inputs in the model
- An operating point listed under **Existing Operating Points (Linear Analysis Workspace)** — These operating points are listed in the **Data Browser** in the **Linear Analysis Workspace** section. When you find an operating point trimming or a simulation snapshot, the software adds it to this list of operating points.
- An operating point listed under **Existing Operating Points (MATLAB Workspace)** — These operating points are listed in the **Data Browser** in the **MATLAB Workspace** section.



Then, in the **Operating Points** drop-down list, under **View/Edit**, click the view or edit option listed for the operating point.

The dialog box that opens shows the values of the operating point. For the model initial conditions and operating points found using simulation snapshots, you can view the input and state values. For operating points found using trimming, you can also view the model outputs that correspond to the operating point.



Within the **Model Linearizer**, you cannot edit the values of the model initial condition operating point or the values of an operating point that you found using trimming.

To edit an operating point that you found using a simulation snapshot, in the Edit dialog box, in the **Value** column, select the state or input you want to edit, and enter the new value. If your simulation snapshot was at a steady state, changing any values in the **Model Linearizer** can place the operating point into a non-steady-state condition.

View and Modify Operating Point at the Command Line

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

Create an operating point object from the Simulink Model.

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

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

States:
----------
    x
    _

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

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

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

Set the value of the first state.

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

View the updated operating point state values.

```
op.States
```

```
ans =
     x
    ____

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

You can also modify other operating points in the MATLAB workspace, including operating points found using trimming or simulation snapshots. If your operating point was at a steady state, changing any values can place the operating point into a non-steady-state condition.

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

See Also

`operspec` | `update`

More About

- “Simulate Simulink Model at Specific Operating Point” on page 1-93

Compute Steady-State Operating Points from Specifications

You can compute a steady-state operating point of a Simulink model by specifying constraints on the model states, outputs, and inputs, and finding a model operating condition that satisfies these constraints. You can trim your model to meet any combination of state, input, or output specifications. Computing an operating point in this way is called *trimming*. For more information on steady-state operating points, see “About Operating Points” on page 1-2.

You can trim your Simulink model:

- In the **Steady State Manager**. For an example, see “Compute Operating Points from Specifications Using Steady State Manager” on page 1-20.
- At the command line. For more information, see “Compute Operating Points from Specifications at the Command Line” on page 1-15.
- In the **Model Linearizer**. For more information, see “Compute Operating Points from Specifications Using Model Linearizer” on page 1-31.

For more information on selecting a trimming tool, see “Compute Steady-State Operating Points” on page 1-5.

For state specifications, you can constrain the values of model states to known values or ranges. You can also define bounds for the derivatives of states that are not at steady state. Using such constraints, you can trim derivatives to known nonzero values or specify derivative tolerances for states that cannot reach steady state. For an example that trims a model for state specifications, see “Compute Operating Points from Specifications Using Model Linearizer” on page 1-31.

You can constrain the values of any root-level input or output ports to known values or ranges. You can also add output specifications to signals in your Simulink model. For an example that adds an output specification in this way, see “Compute Operating Points from Specifications Using Steady State Manager” on page 1-20.

If your trimming is unsuccessful; that is, if the optimization search was unable to meet all of your specifications, determine the specifications that could not be met by validating your trimmed operating point against the original specifications. For more information, see “Validate Operating Point Against Specifications” on page 1-38.

After trimming your model, you can:

- Linearize your model at the resulting operating point. For more information, see “Linearize at Trimmed Operating Point” on page 2-67.
- Simulate your model at the resulting operating point. For more information, see “Simulate Simulink Model at Specific Operating Point” on page 1-93.

See Also

Functions
`findop` | `findopOptions`

Apps
Steady State Manager | **Model Linearizer**

More About

- “Compute Operating Points from Specifications at the Command Line” on page 1-15
- “Compute Operating Points from Specifications Using Steady State Manager” on page 1-20
- “Compute Operating Points from Specifications Using Model Linearizer” on page 1-31

Compute Operating Points from Specifications at the Command Line

You can compute a steady-state operating point of a Simulink® model by specifying constraints on the model states, outputs, and inputs, and by finding a model operating condition that satisfies these constraints. For more information on steady-state operating points, see “About Operating Points” on page 1-2 and “Compute Steady-State Operating Points” on page 1-5.

To find an operating point for your Simulink model, you can programmatically trim your model using the `findop`, as shown in this example.

Alternatively, you can trim your model in the:

- **Steady State Manager**. For more information, see “Compute Operating Points from Specifications Using Steady State Manager” on page 1-20.
- **Model Linearizer**. For more information, see “Compute Operating Points from Specifications Using Model Linearizer” on page 1-31.

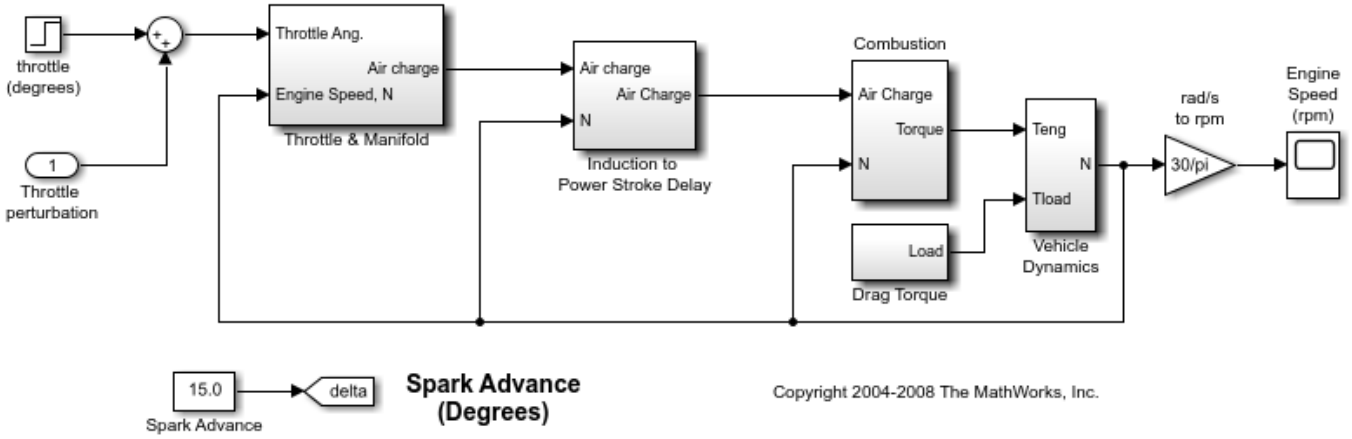
In this example, you compute an operating point to meet output specifications. Using a similar approach, you can define state or input specifications. Also, you can define a combination of state, output, and input specifications; that is, you do not have to use, for example, only state specifications.

For more information on trimming your model to meet specifications, see “Compute Steady-State Operating Points from Specifications” on page 1-13.

Open Simulink Model

Open the Simulink model.

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



Define Operating Point Specifications

Create a default operating point specification for the model.

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

```
opspec =


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

States:
----------
      <strong>x</strong>       <strong>Known</strong>    <strong>SteadyState</strong>    <strong>
    <strong>______</strong>    <strong>_____</strong>    <strong>___________</strong>    <strong>

(1.) scdspeed/Throttle & Manifold/Intake Manifold/p0 = 0.543 bar
     0.543    false       true        -Inf    Inf    -Inf      Inf
(2.) scdspeed/Vehicle Dynamics/w = T//J w0 = 209 rad//s
    209.48    false       true        -Inf    Inf    -Inf      Inf

Inputs:
----------
    <strong>u</strong>    <strong>Known</strong>    <strong>Min</strong>     <strong>Max</strong>
    <strong>_</strong>    <strong>_____</strong>    <strong>____</strong>    <strong>___</strong>

(1.) scdspeed/Throttle  perturbation
    0    false    -Inf    Inf

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

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

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

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

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

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

View the updated operating point specification.

```
opspec
```

```
opspec =


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

States:
----------
      <strong>x</strong>       <strong>Known</strong>    <strong>SteadyState</strong>    <strong>
    <strong>______</strong>    <strong>_____</strong>    <strong>___________</strong>    <strong>

(1.) scdspeed/Throttle & Manifold/Intake Manifold/p0 = 0.543 bar
```

```
     0.543    false       true        -Inf    Inf    -Inf      Inf
(2.) scdspeed/Vehicle Dynamics/w = T//J w0 = 209 rad//s
    209.48    false       true        -Inf    Inf    -Inf      Inf

Inputs:
----------
    <strong>u</strong>    <strong>Known</strong>    <strong>Min</strong>     <strong>Max</strong>
    <strong>_</strong>    <strong>_____</strong>    <strong>____</strong>    <strong>___</strong>

(1.) scdspeed/Throttle  perturbation
    0    false    -Inf    Inf

Outputs:
----------
     <strong>y</strong>      <strong>Known</strong>    <strong>Min</strong>     <strong>Max</strstro
    <strong>____</strong>    <strong>_____</strong>    <strong>____</strong>    <strong>___</strstro

(1.) scdspeed/rad//s to rpm
    2000    true     -Inf    Inf
```

Trim Model

Find an operating point that meets these specifications.

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


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

opreport =


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

Operating point specifications were successfully met.
States:
----------
    <strong>Min</strong>        <strong>x</strong>       <strong>Max</strong>    <strong>dxMin</s
    <strong>____</strong>    <strong>_______</strong>    <strong>___</strong>    <strong>_____</s

(1.) scdspeed/Throttle & Manifold/Intake Manifold/p0 = 0.543 bar
    -Inf    0.54363    Inf      0       2.6649e-13      0
(2.) scdspeed/Vehicle Dynamics/w = T//J w0 = 209 rad//s
    -Inf     209.44    Inf      0      -8.4758e-12      0

Inputs:
----------
    <strong>Min</strong>         <strong>u</strong>        <strong>Max</strong>
    <strong>____</strong>    <strong>_________</strong>    <strong>___</strong>

(1.) scdspeed/Throttle  perturbation
    -Inf    0.0038183    Inf

Outputs:
----------
    <strong>Min</strong>      <strong>y</strong>      <strong>Max</strong>
```

```
    <strong>____</strong>    <strong>____</strong>    <strong>____</strong>

(1.) scdspeed/rad//s to rpm
    2000    2000    2000
```

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

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

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

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

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

Find an operating point that meets these specifications.

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


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

opreport =


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

Operating point specifications were successfully met.
States:
----------
    <strong>Min</strong>       <strong>x</strong>       <strong>Max</strong>    <strong>dxMin</s
    <strong>____</strong>    <strong>______</strong>    <strong>___</strong>    <strong>_____</s

(1.) scdspeed/Throttle & Manifold/Intake Manifold/p0 = 0.543 bar
    -Inf    0.5436    Inf      0       2.9879e-13      0
(2.) scdspeed/Vehicle Dynamics/w = T//J w0 = 209 rad//s
    -Inf    209.48    Inf      0      -9.8968e-13      0

Inputs:
----------
    <strong>Min</strong>         <strong>u</strong>        <strong>Max</strong>
    <strong>____</strong>    <strong>_________</strong>    <strong>___</strong>

(1.) scdspeed/Throttle  perturbation
    -Inf    0.0050021    Inf

Outputs:
----------
    <strong>Min</strong>       <strong>y</strong>       <strong>Max</strong>
    <strong>____</strong>    <strong>______</strong>    <strong>____</strong>
```

```
(1.) scdspeed/rad//s to rpm
    1900    2000.4    2100
```

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

After trimming your model, you can:

- Linearize your model at the resulting operating point. For more information, see “Linearize at Trimmed Operating Point” on page 2-67.
- Simulate your model at the resulting operating point. For more information, see “Simulate Simulink Model at Specific Operating Point” on page 1-93.

See Also

`findop` | `addoutputspec` | `operspec`

More About

- “Compute Steady-State Operating Points from Specifications” on page 1-13
- “Compute Operating Points from Specifications Using Steady State Manager” on page 1-20
- “Compute Operating Points from Specifications Using Model Linearizer” on page 1-31

Compute Operating Points from Specifications Using Steady State Manager

You can compute a steady-state operating point of a Simulink model by specifying constraints on the model states, outputs, and inputs, and by finding a model operating condition that satisfies these constraints. For more information on steady-state operating points, see “About Operating Points” on page 1-2 and “Compute Steady-State Operating Points” on page 1-5.

To find an operating point for your Simulink model, you can interactively trim your model using the **Steady State Manager**, as shown in this example.

Alternatively, you can trim your model:

- At the command line. For more information, see “Compute Operating Points from Specifications at the Command Line” on page 1-15.
- In the **Model Linearizer**. For more information, see “Compute Operating Points from Specifications Using Model Linearizer” on page 1-31.

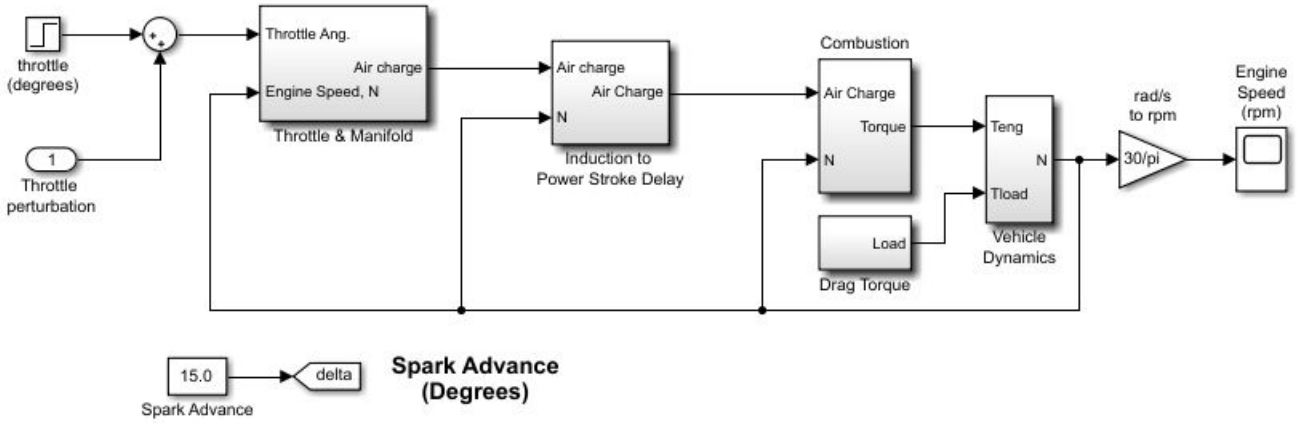
In this example, you compute an operating point to meet output specifications. Using a similar approach, you can define state or input specifications. Also, you can define a combination of state, output, and input specifications; that is, you do not have to use, for example, only state specifications.

For more information on trimming your model to meet specifications, see “Compute Steady-State Operating Points from Specifications” on page 1-13.

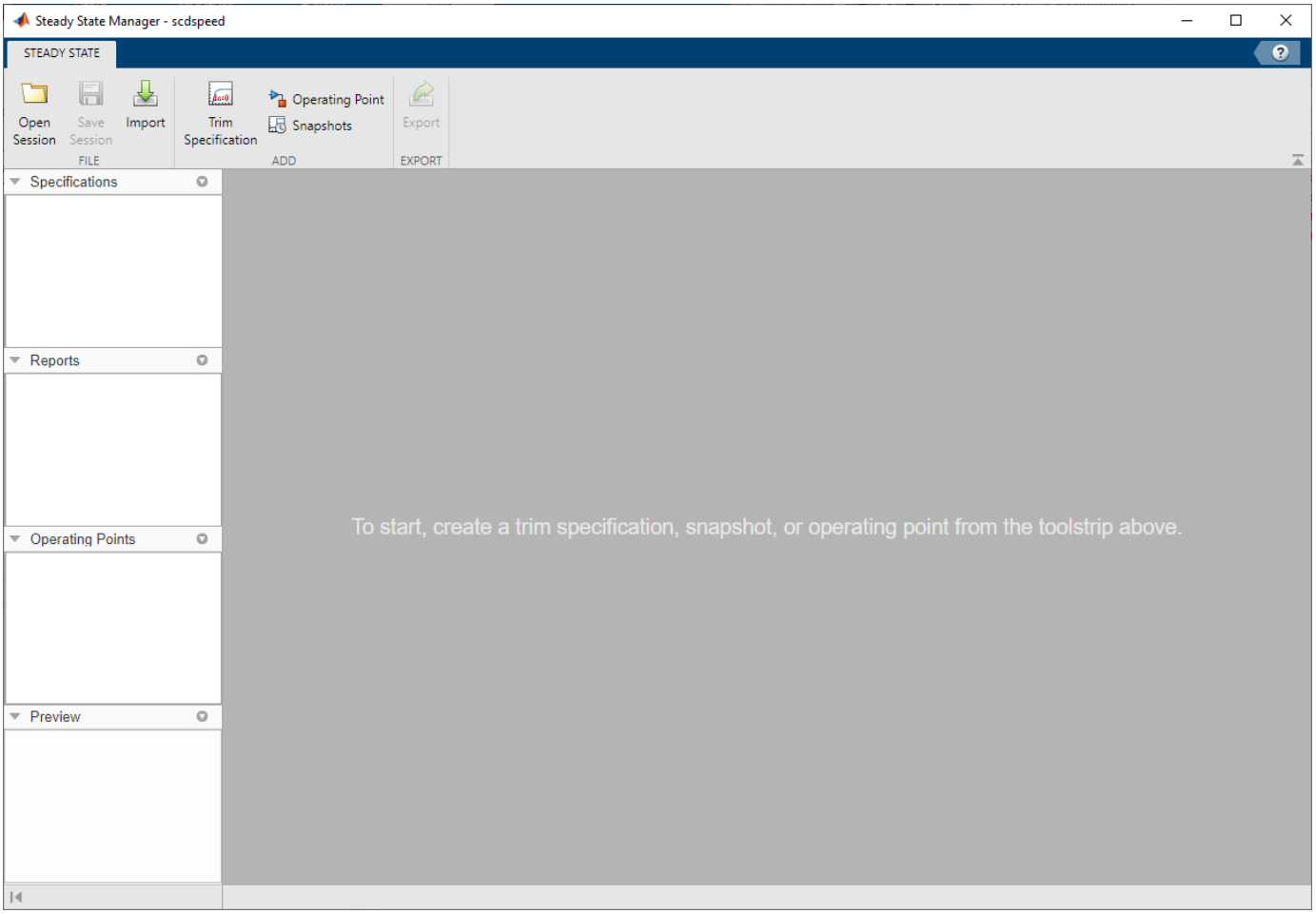
Open Steady State Manager

Open the Simulink model.

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

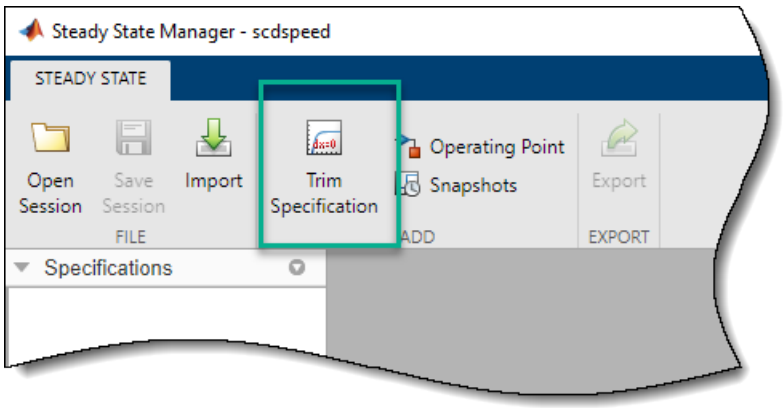


To open the **Steady State Manager**, in the Simulink model window, in the **Apps** gallery, click **Steady State Manager**.

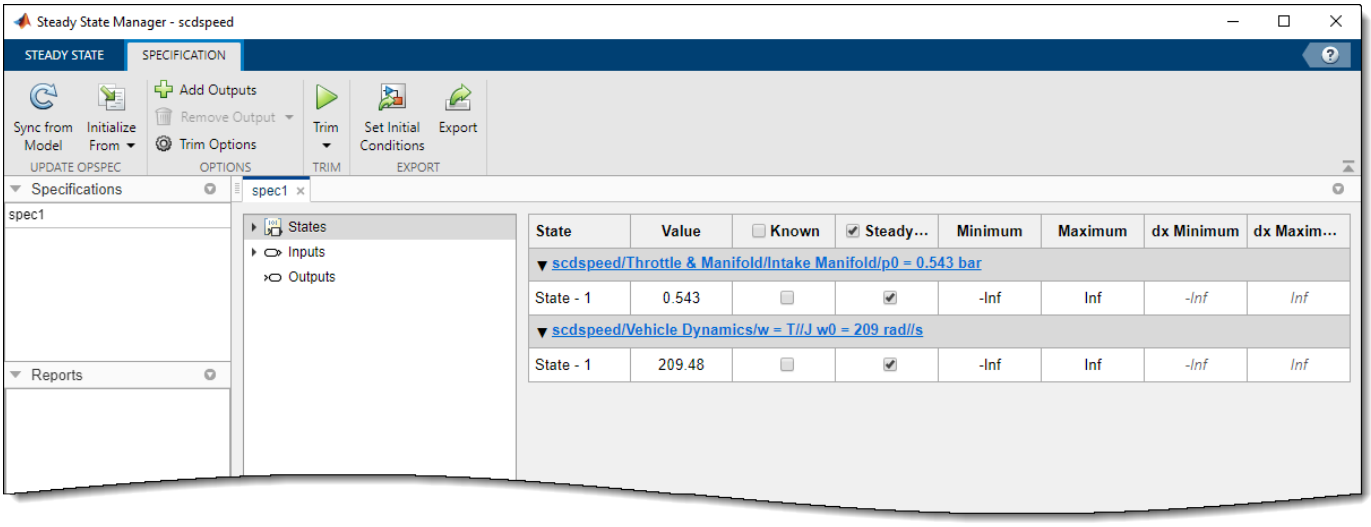


Define Operating Point Specifications

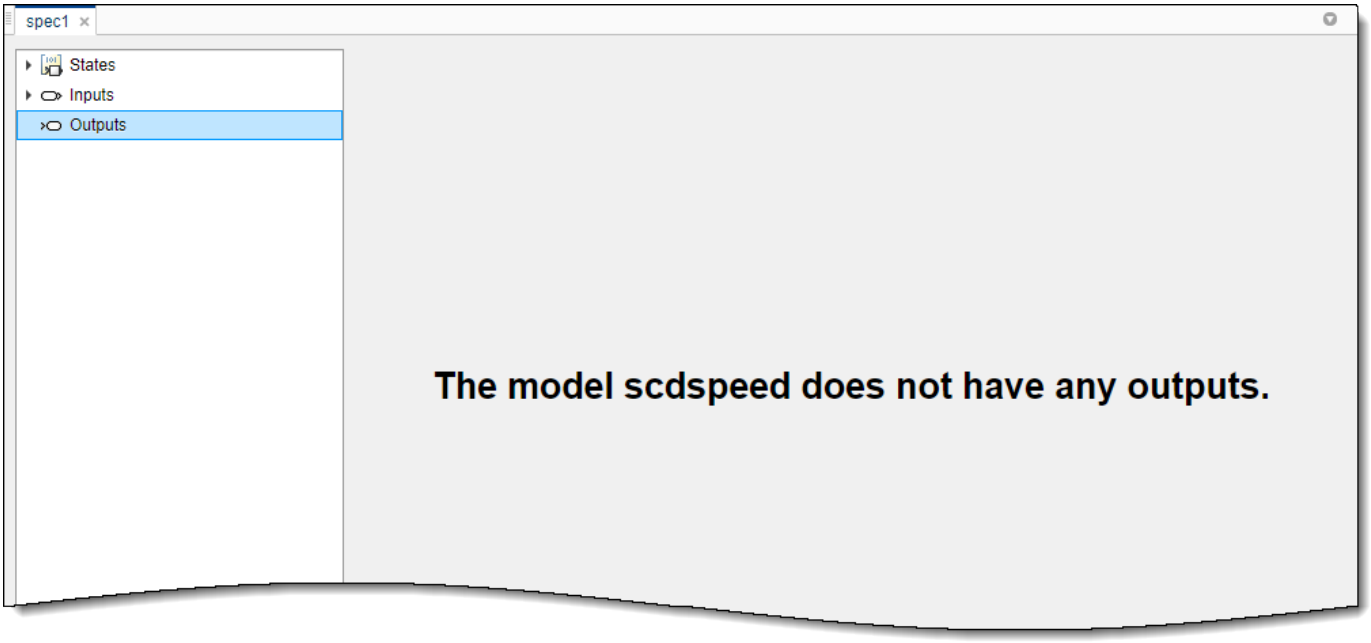
To create an operating point specification, in **Steady State Manager**, on the **Steady State** tab, click **Trim Specification**.



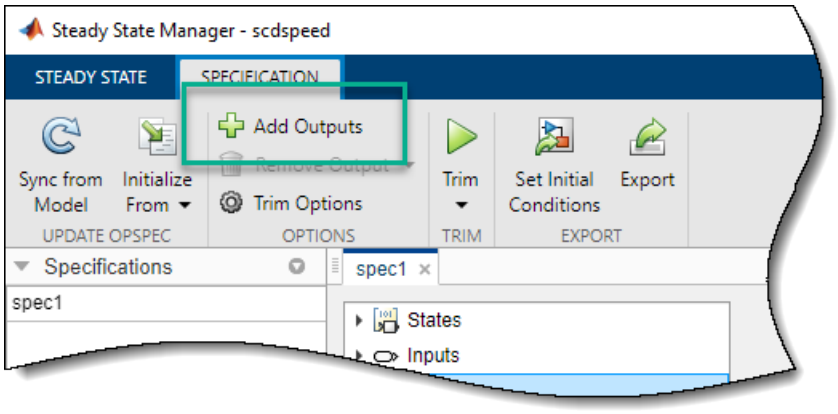
In the **Data Browser**, in the **Specifications** section, the software adds a default operating point specification, `spec1`. Also, the **Specification** tab opens along with a corresponding **spec1** document.



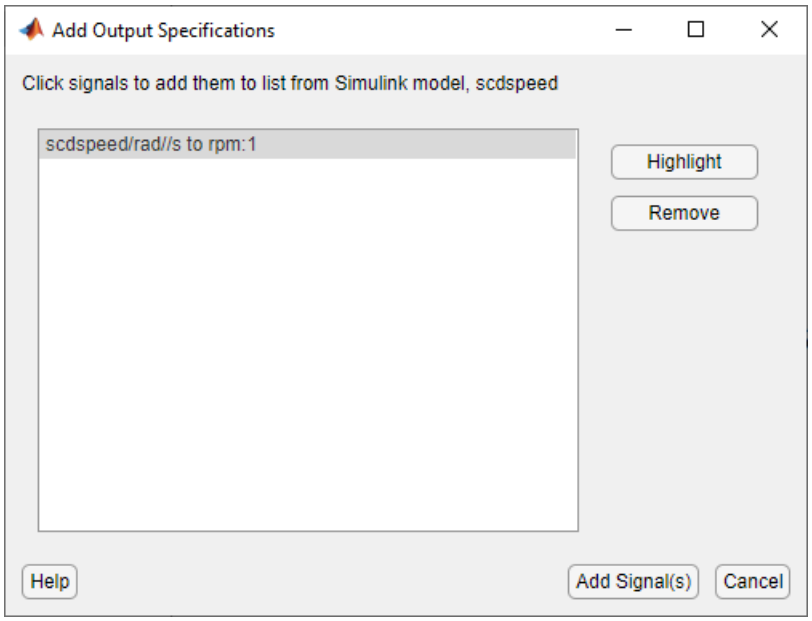
In the **spec1** document, in the navigation tree, select the type of specification that you want to add. For this example, you want to find a steady-state operating point at which the engine speed is fixed at 2000 rpm using an output specification. Therefore, click **Outputs**.



Since the model does not have any root-level output ports or defined trim outputs constraints, the operating point specification does not have any outputs. To add an output to the operating point specification, on the **Specification** tab, click **Add Outputs**.

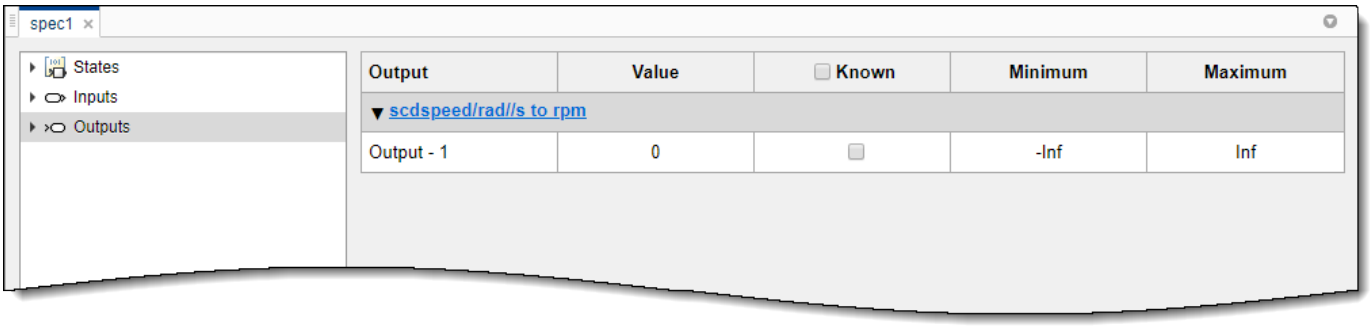


In the Add Output Specifications dialog box, specify the signals to which you want to add an output specification. To add a signal to the list, in the Simulink model window, click the output signal of the rad/s to rpm block. Doing so adds the signal to the Add Output Specifications dialog box.



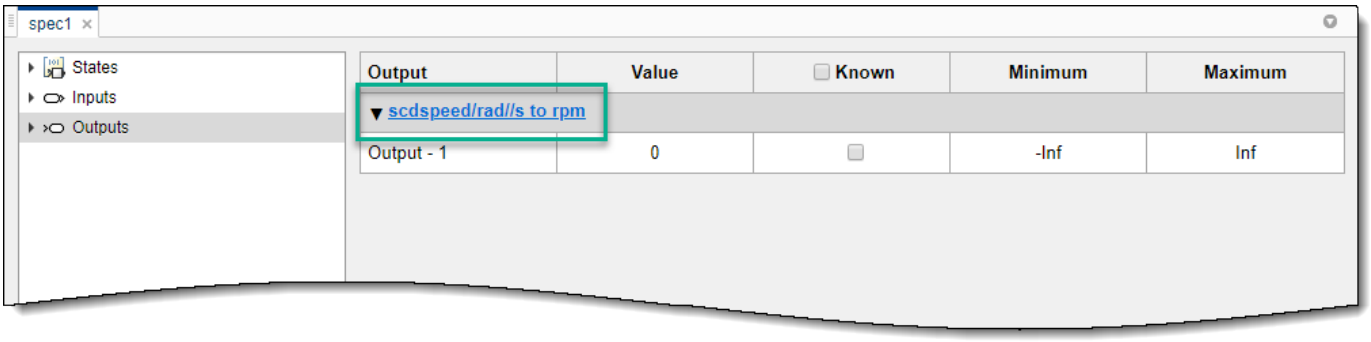
Click **Add Signal(s)**.

The software adds this signal to `spec1` as an output specification. To view and edit the specification, in the **spec1** document, click **Outputs**.

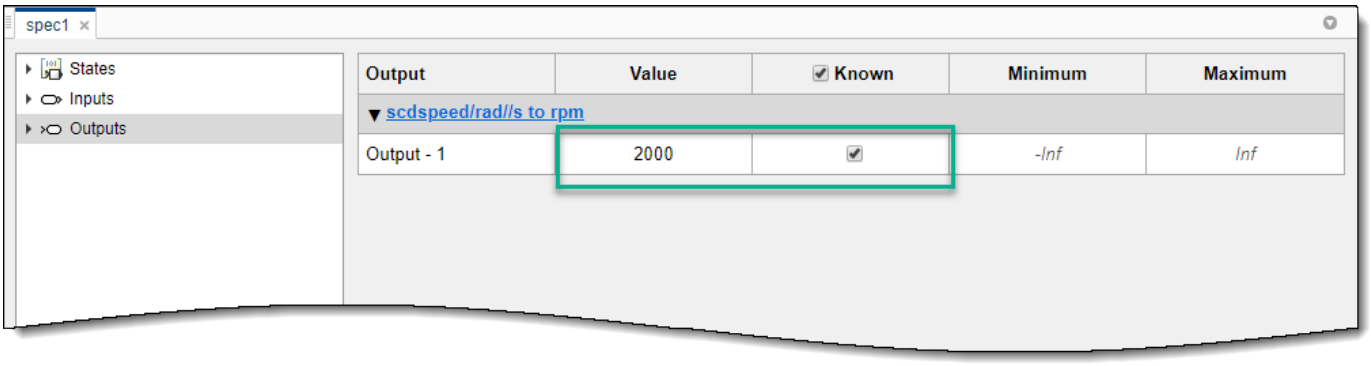


The selected signal is listed in the output specification table under the name of its source block.

Tip To go to the block in your model that is associated with a given state, input, or output specification, in the specification table, click the block name.

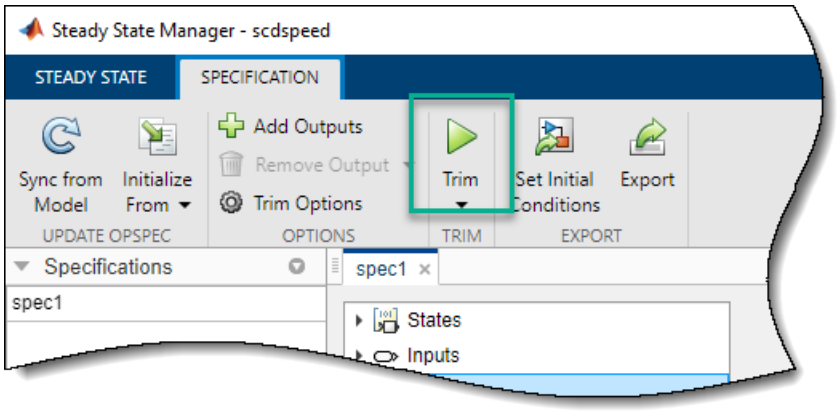


Specify a known speed value. In the **spec1** document, in the **Known** column, select the corresponding row, and in the **Value** column set the known value to `2000`.

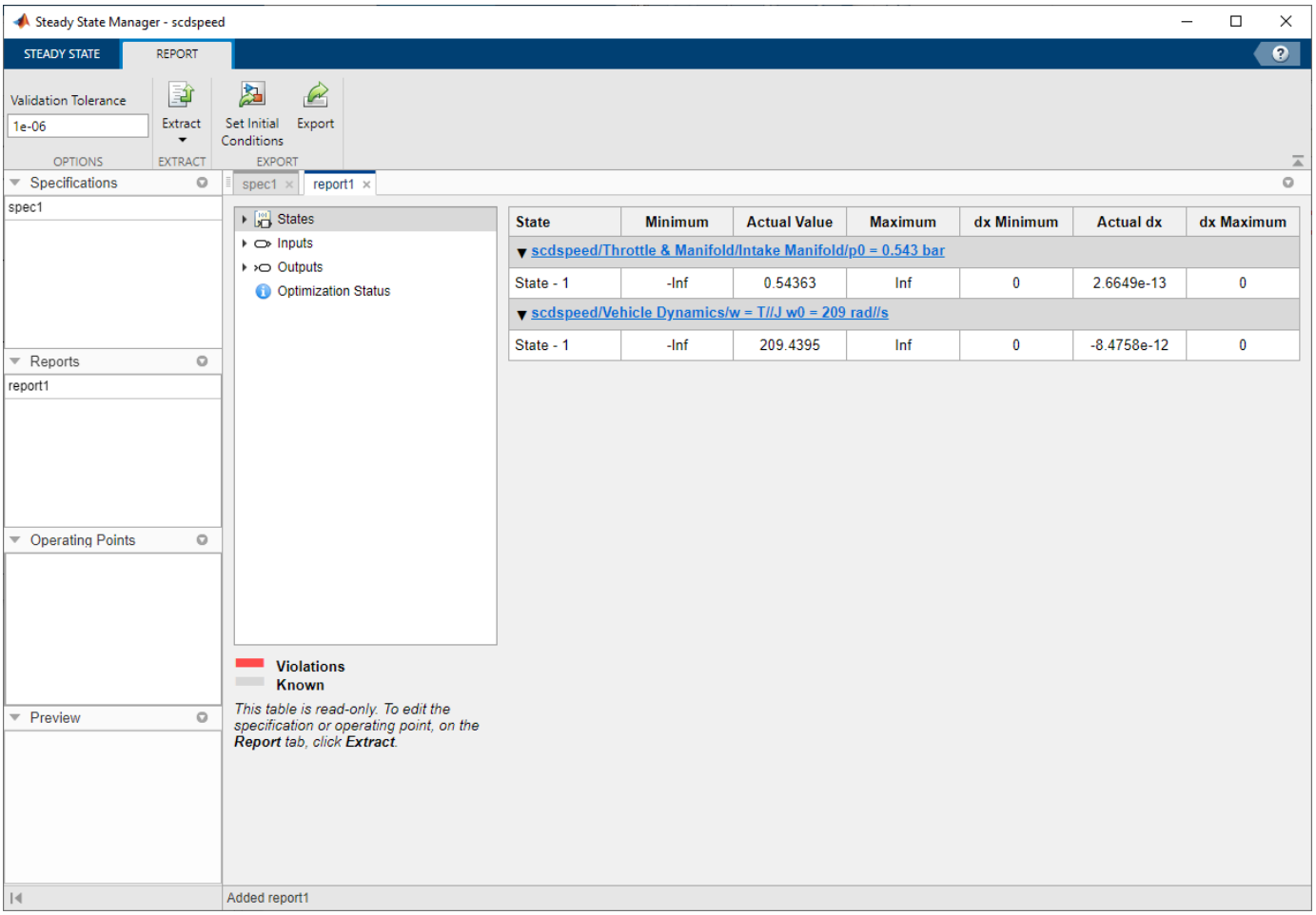


Trim Model

To compute the operating point that meets this output specification, on the **Specification** tab, click **Trim** .

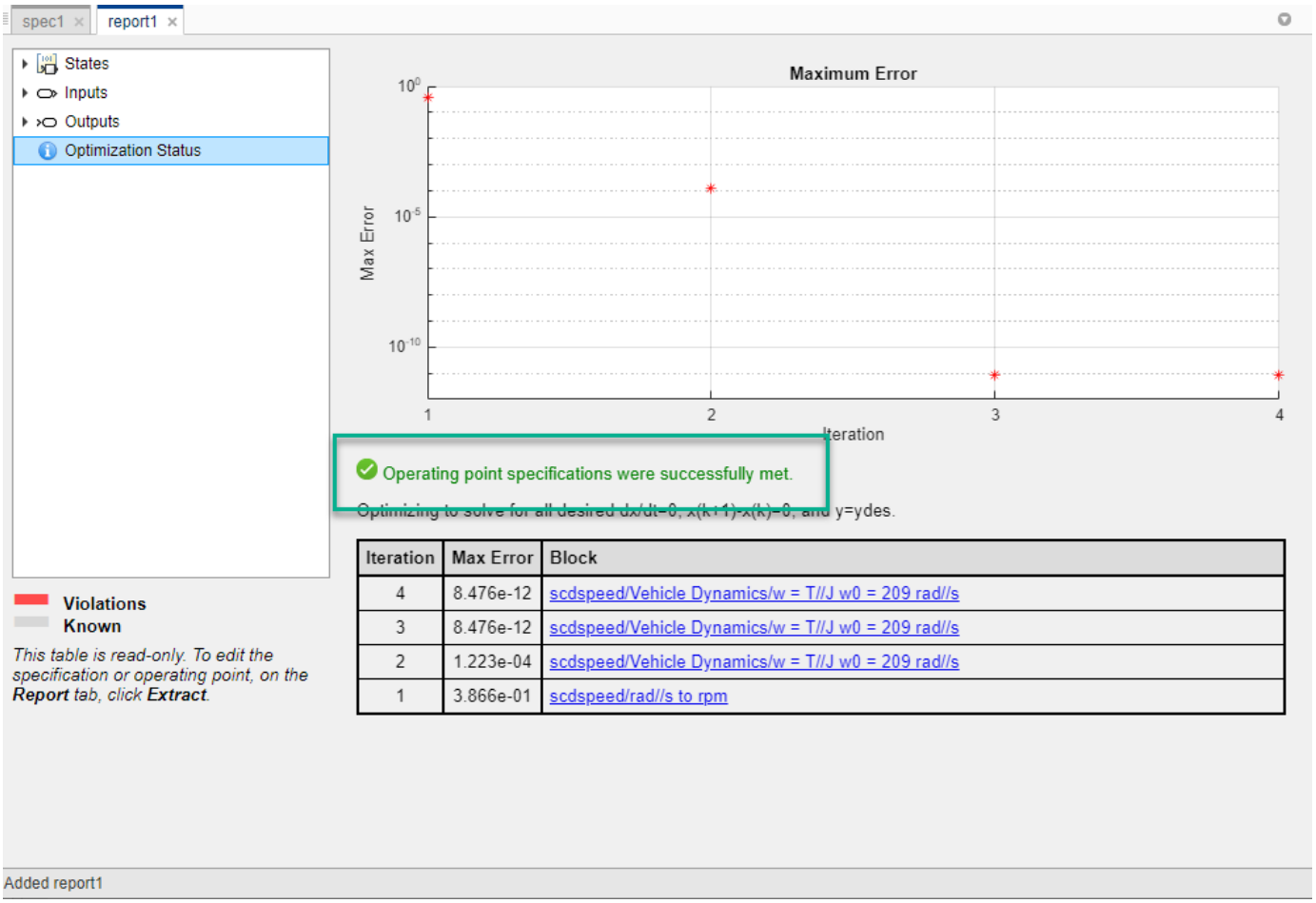


The software trims the model and generates an operating point search report. The report, `report1`, is added to the **Data Browser**, in the **Reports** section. Also, the **Report** tab opens along with a corresponding **report1** document.



For this example, you use default trimming options. To specify different options, such as the optimization method or a custom cost function, on the **Specification** tab, click **Trim Options**.

To check whether the optimization search converged to a solution that meets the specifications, in the **report1** document, click **Optimization Status**.

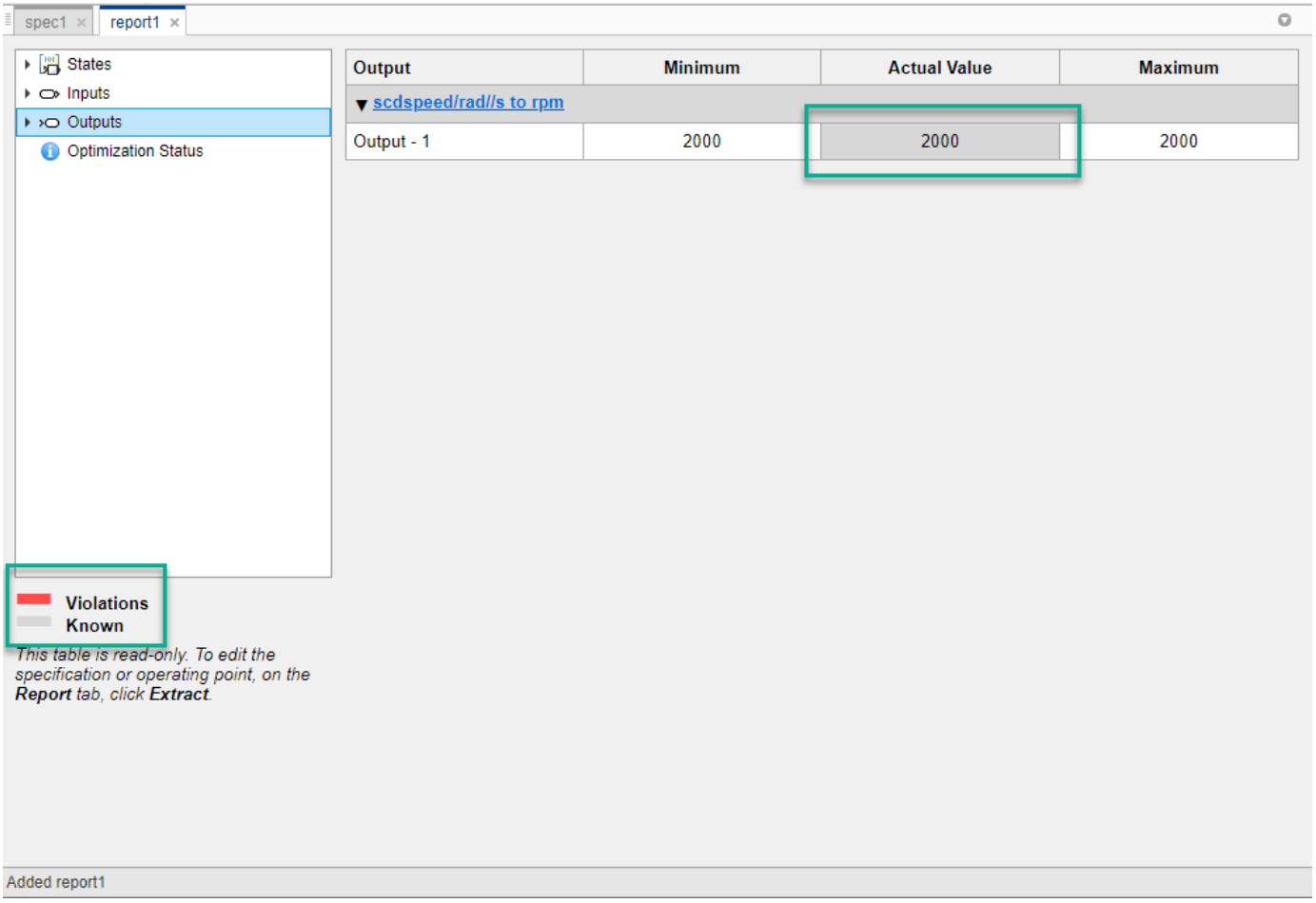


The optimization status shows that the optimization algorithm terminated successfully, finding an operating point that meets the specifications.

The **Maximum Error** plot and the **Max Error** column show the maximum constraint violation for each iteration. The **Block** column shows the block to which the maximum constraint violation applies.

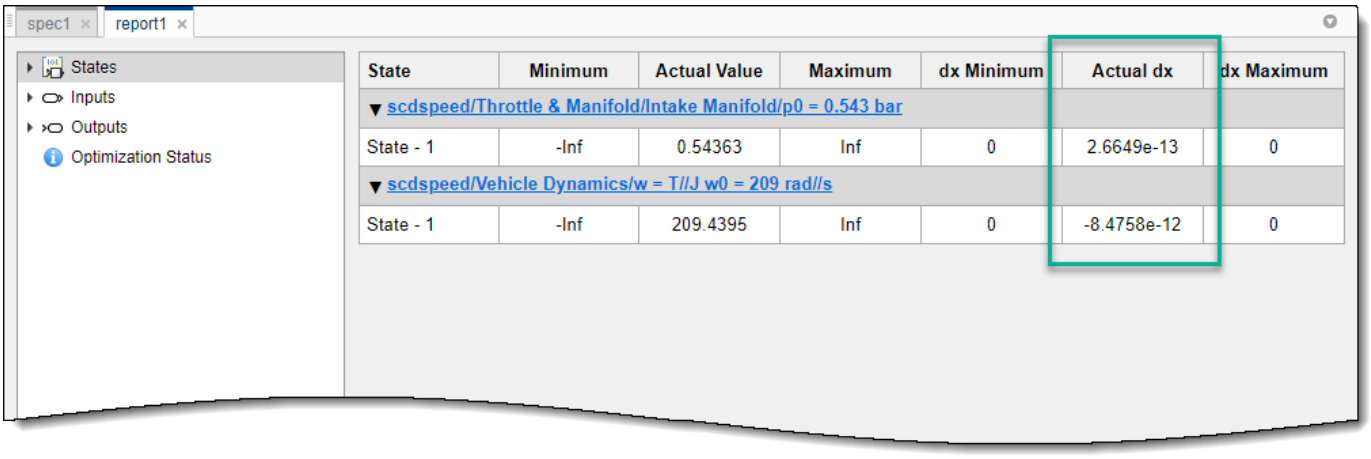
Validate Operating Point

For this example, the optimization search converged to an operating point that met the specification. When the operating point search report indicates that the search was unsuccessful, you can validate your operating point against the specifications. To do so, in the **report1** document, in the navigation tree, select the specifications that you want to check. For this example, click **Outputs**.



In the specification table, known values are highlighted in gray, and constraint violations are highlighted in red. For this example, there are no constraint violations.

You can also verify whether the operating point is at steady state. For example, in the **report1** document, click **States**.



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

For more information on validating operating points, see “Validate Operating Point Against Specifications” on page 1-38.

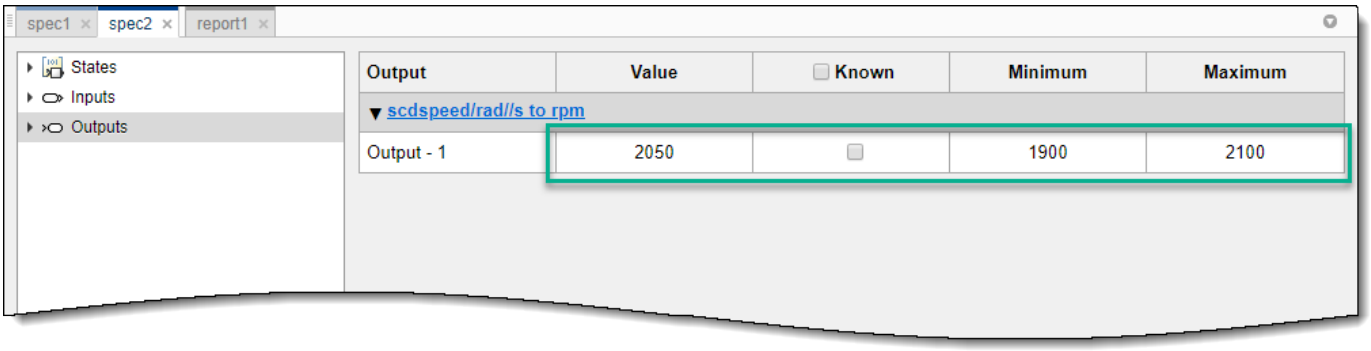
Trim Model for Different Specifications

You can also specify bounds for your specification rather than known values. For example, suppose that you know that there is a steady-state condition in the range from 1900 to 2100 rpm. To find this operating point, first create another specification by copying and editing previous specification. In the **Data Browser**, right-click `spec1`, and select **Copy**.

The software adds `spec2` to the **Data Browser**. To open the specification document for editing, double-click this new specification.

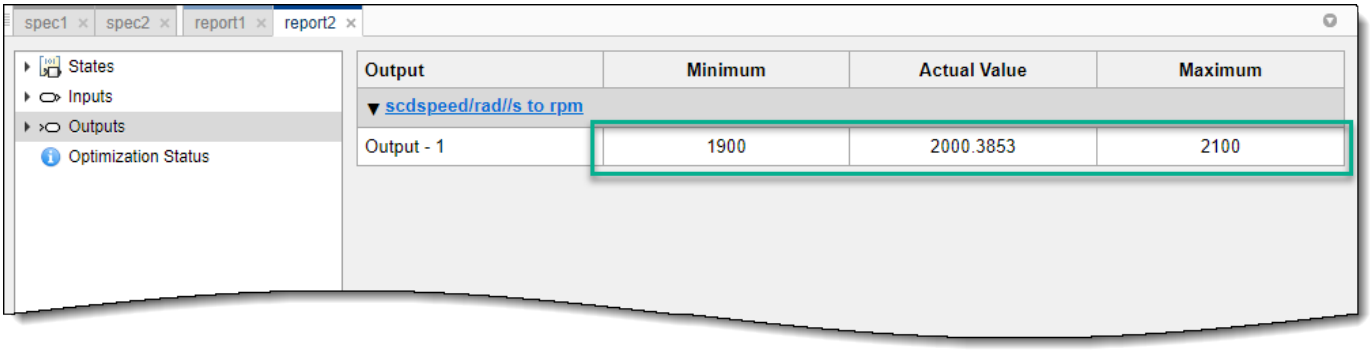
In the **spec2** document, click **Outputs**. Then, in the specification table:

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



On the **Specification** tab, click **Trim** . The software trims the model and opens the operating point search report in the **report2** document.

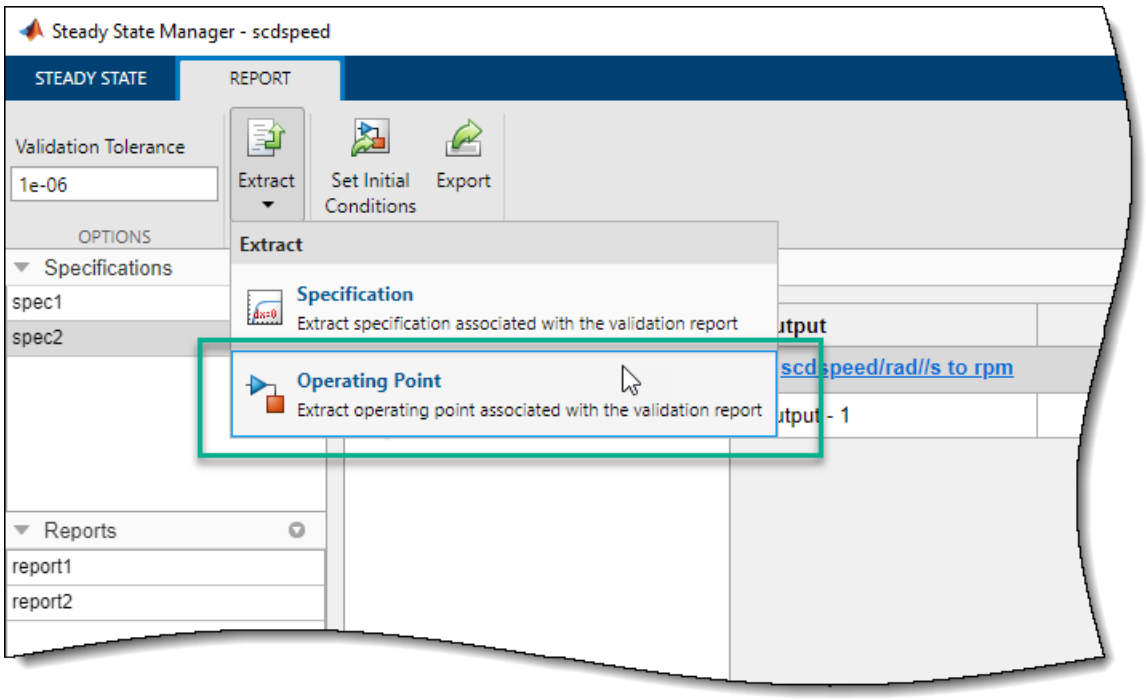
Click **Outputs**.



As shown in the **Actual Value** column, the trimmed output value is within the specified bounds.

Extract Operating Point from Report

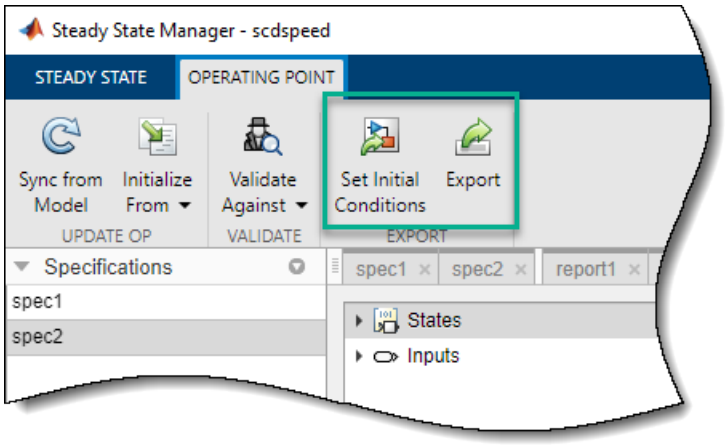
The operating point search report is read-only and contains information about both the specifications and the trimmed operating point. You can extract either a specification or operating point object from a search report. For example, on the **Report** tab for `report2`, click **Extract** > **Operating Point**.



The software extracts the trimmed operating point, `op1`, from the report, adding it to the **Data Browser**, in the **Operating Points** section.

Export Operating Point Data

Once you have computed an operating point that meets your specifications, you can export the model to the MATLAB workspace and set the initial conditions of your model to the values in the operating point. To do so, on the **Operating Point** tab, click **Export** or **Set Initial Conditions**, respectively.



For more information on setting your model initial conditions and simulating your model at a specific operating point, see “Simulate Simulink Model at Specific Operating Point” on page 1-93.

See Also

Steady State Manager

More About

- “Compute Steady-State Operating Points from Specifications” on page 1-13
- “Compute Operating Points from Specifications at the Command Line” on page 1-15
- “Compute Operating Points from Specifications Using Model Linearizer” on page 1-31

Compute Operating Points from Specifications Using Model Linearizer

You can compute a steady-state operating point of a Simulink model by specifying constraints on the model states, outputs, and inputs, and finding a model operating condition that satisfies these constraints. For more information on steady-state operating points, see “About Operating Points” on page 1-2 and “Compute Steady-State Operating Points” on page 1-5.

To find an operating point for your Simulink model, you can interactively trim your model using the **Model Linearizer**, as shown in this example.

Alternatively, you can trim your model:

- In the **Steady State Manager**. For more information, see “Compute Operating Points from Specifications Using Steady State Manager” on page 1-20.
- At the command line. For more information, see “Compute Operating Points from Specifications at the Command Line” on page 1-15.

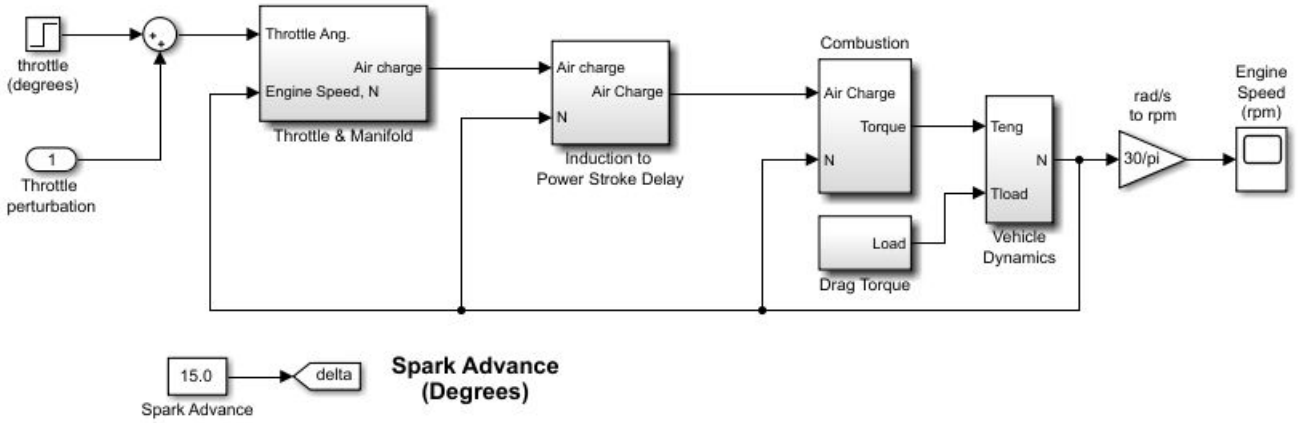
In this example, you compute an operating point to meet state specifications. Using a similar approach, you can define output or input specifications. Also, you can define a combination of state, output, and input specifications; that is, you do not have to use, for example, only state specifications.

For more information on trimming your model to meet specifications, see “Compute Steady-State Operating Points from Specifications” on page 1-13.

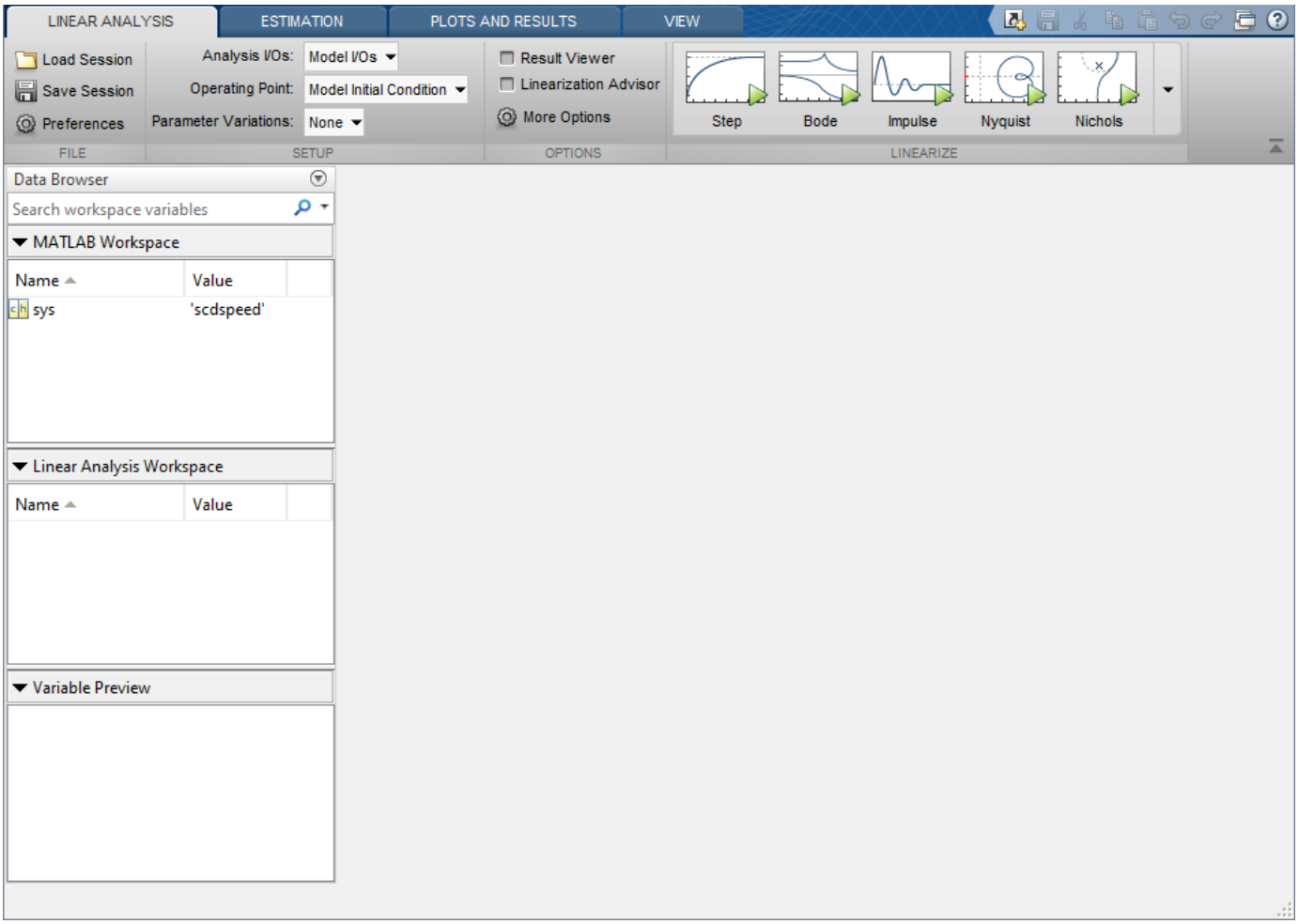
Open Model Linearizer

Open the Simulink model.

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



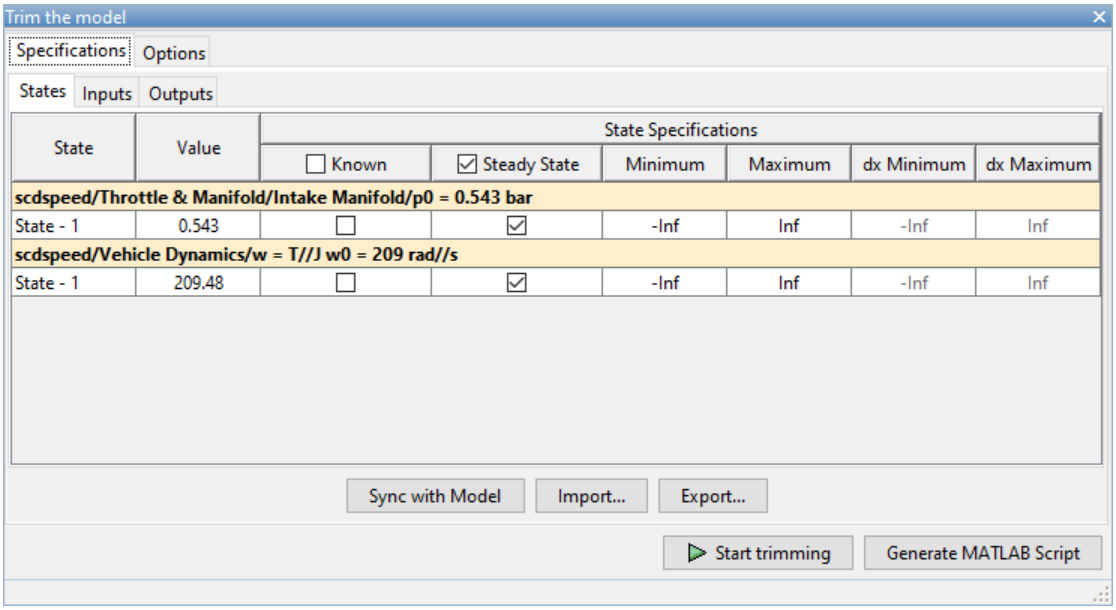
To open the **Model Linearizer**, in the Simulink model window, in the **Apps** gallery, click **Model Linearizer**.



Define Operating Point Specifications

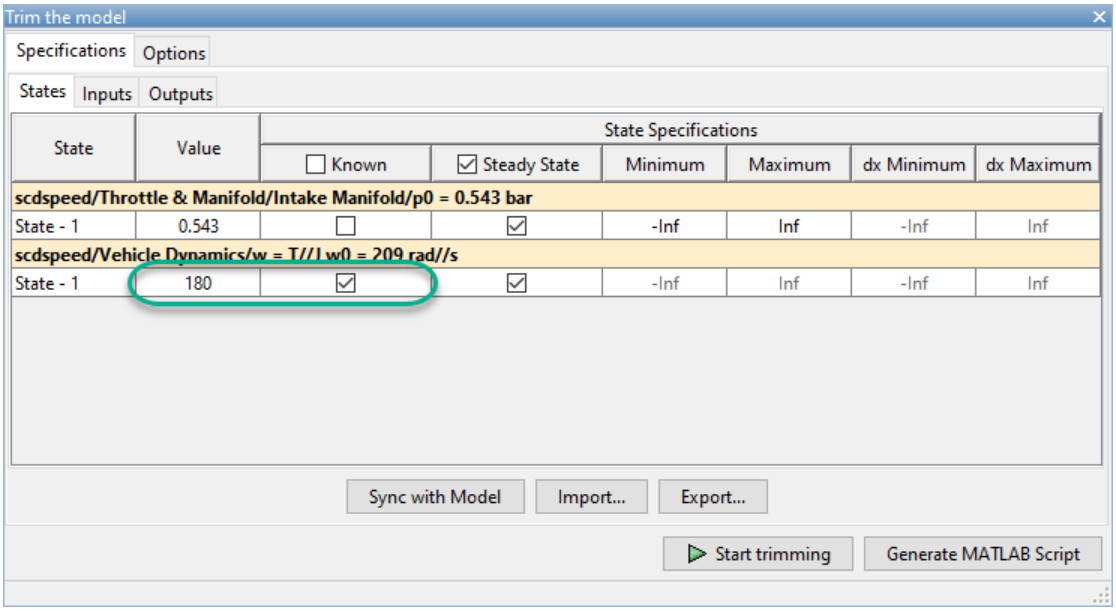
In the **Model Linearizer**, on the **Linear Analysis** tab, in the **Operating Point** drop-down list, select `Trim Model`.

In the Trim the model dialog box, on the **Specifications** tab, you can define specifications for model states, inputs, and outputs. For this example, click the **States** tab.

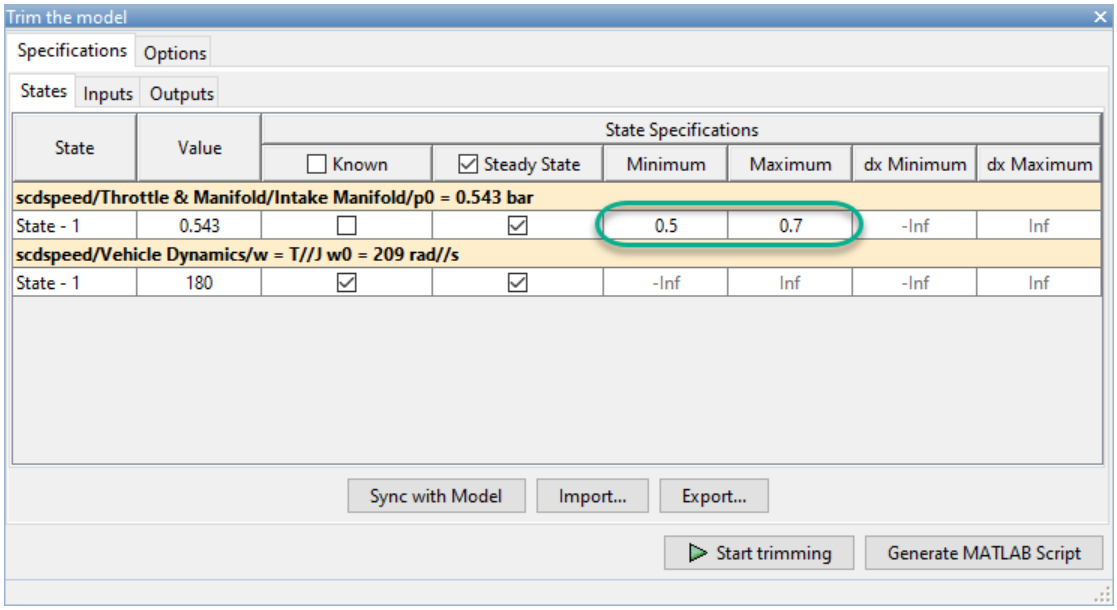


By default, on the **States** tab, the software specifies both model states to be at equilibrium, as shown by the check marks in the **Steady State** column. Both states are also specified as unknown values; that is, their steady-state values are calculated during trimming, with an initial guess specified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

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



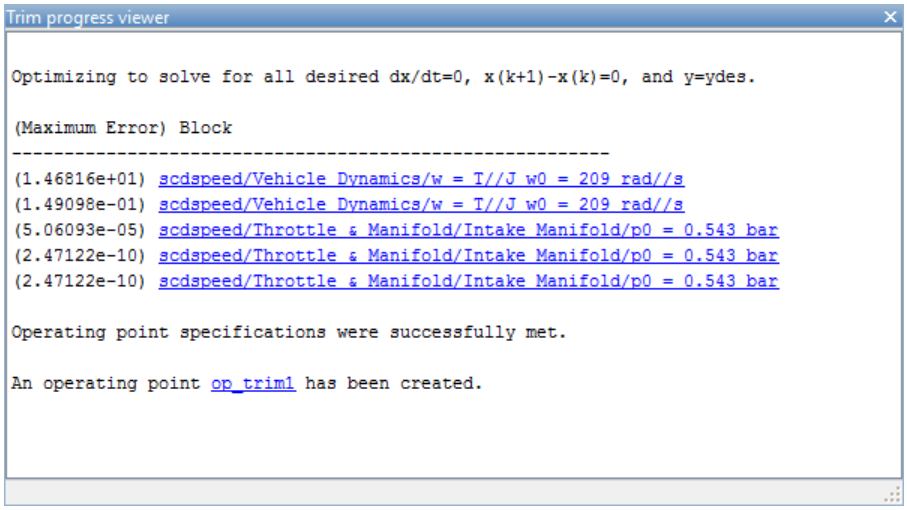
You can also specify bounds for model states during trimming. For this example, constrain the first state to be between `0.5` and `0.7`. To do so, enter these values in the **Minimum** and **Maximum** columns, respectively.



Trim Model

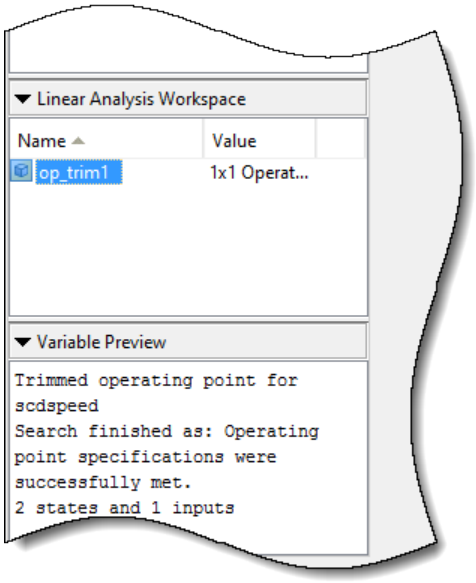
To compute the operating point that meets these specifications, click **Start trimming**.

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



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

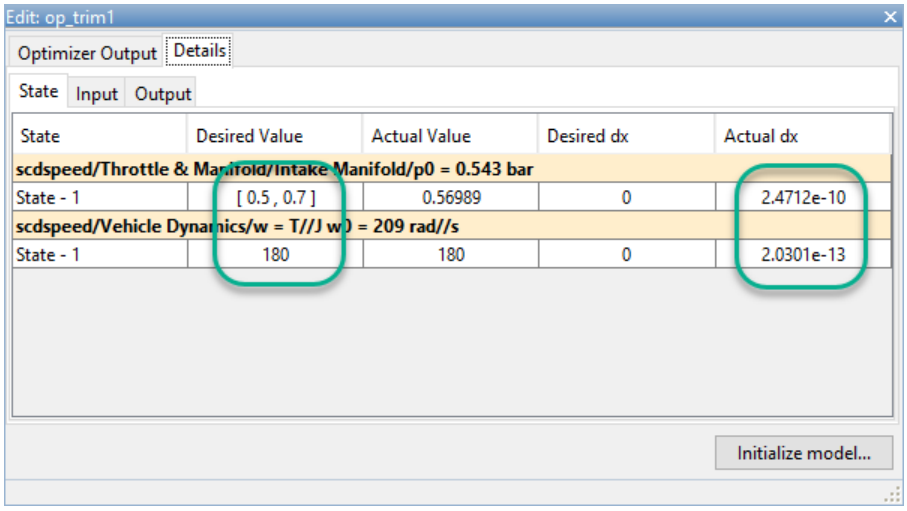
The trimmed operating point, `op_trim1`, appears in the **Linear Analysis Workspace**.



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

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

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

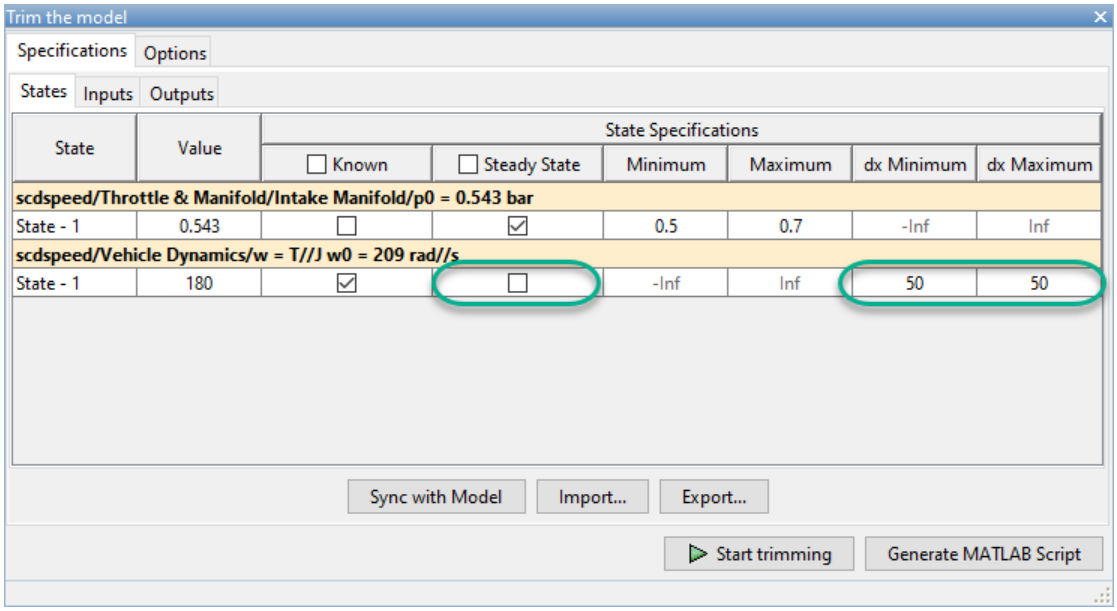


Constrain State Derivatives

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

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

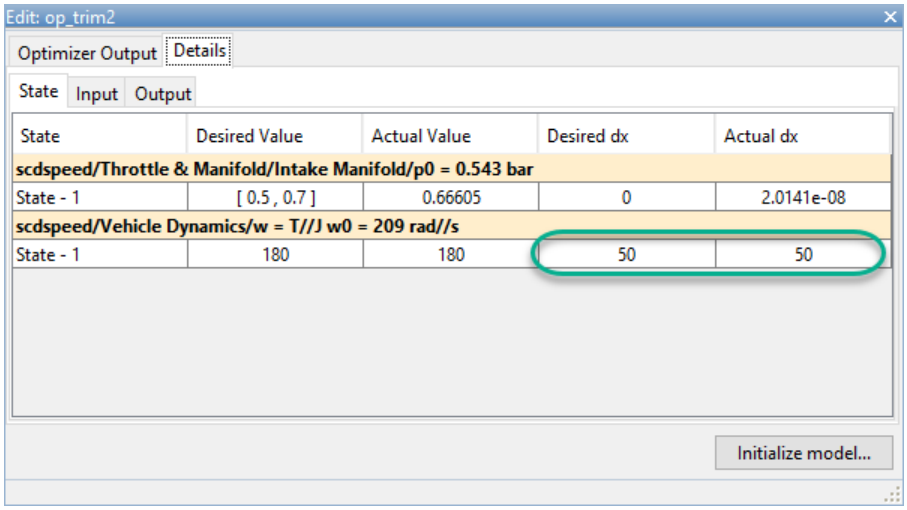
In the **Steady State** column, clear the selection in the corresponding row. Then, in the **dx Minimum** and **dx Maximum** columns, set both state derivative bounds to `50`.



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

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

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



After trimming your model, you can:

- Linearize your model at the resulting operating point. For more information, see “Linearize at Trimmed Operating Point” on page 2-67.
- Simulate your model at the resulting operating point. For more information, see “Simulate Simulink Model at Specific Operating Point” on page 1-93.

See Also

Model Linearizer

More About

- “Compute Steady-State Operating Points from Specifications” on page 1-13
- “Compute Operating Points from Specifications at the Command Line” on page 1-15
- “Compute Operating Points from Specifications Using Steady State Manager” on page 1-20

Validate Operating Point Against Specifications

When you compute an operating point based on input, output, or state specifications, the Simulink Control Design software indicates whether the specifications were successfully met during the trimming process. If the trimming was unsuccessful, to determine the specifications that could not be met, you must validate your trimmed operating point against the original specifications.

Validate Operating Point in Steady State Manager

When you compute an operating point using **Steady State Manager**, the software creates an operating point report object and highlights any operating point values that violate the constraints in the specification.

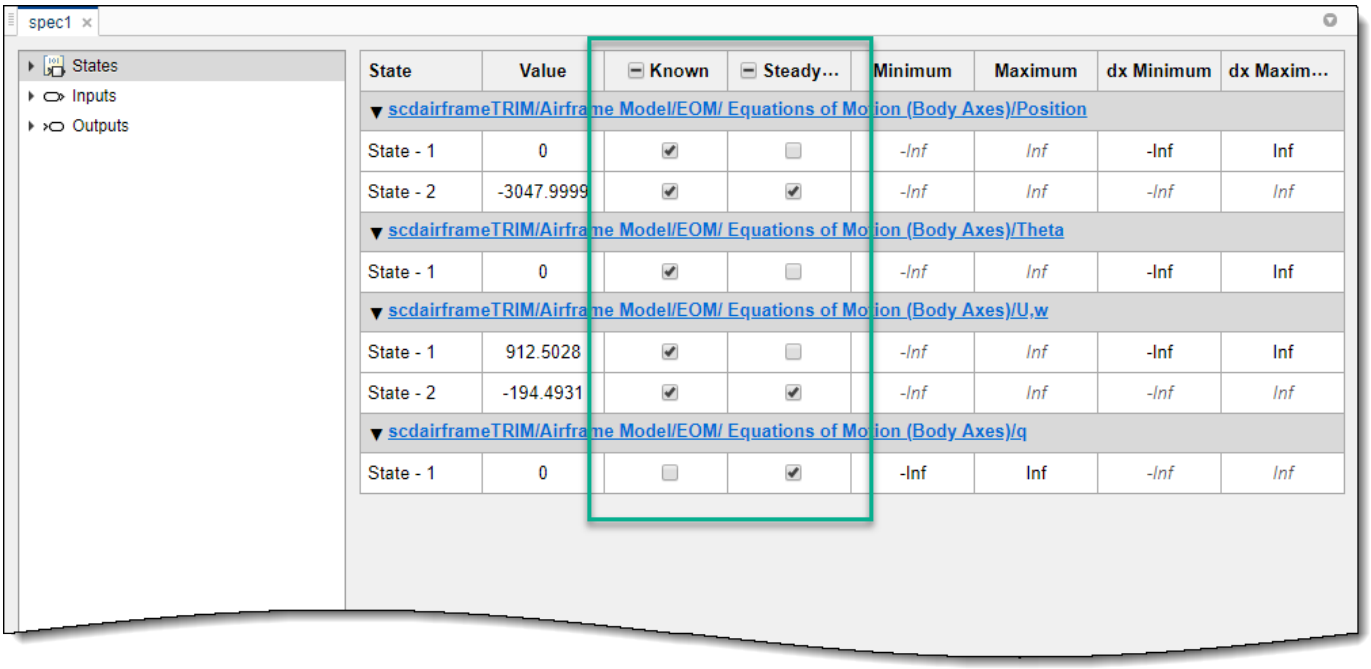
For example, consider the `scdairframeTRIM` model. Open the model and set the speed and incidence angle parameters.

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

alpha_ini = -0.21;
v_ini = 933;
```

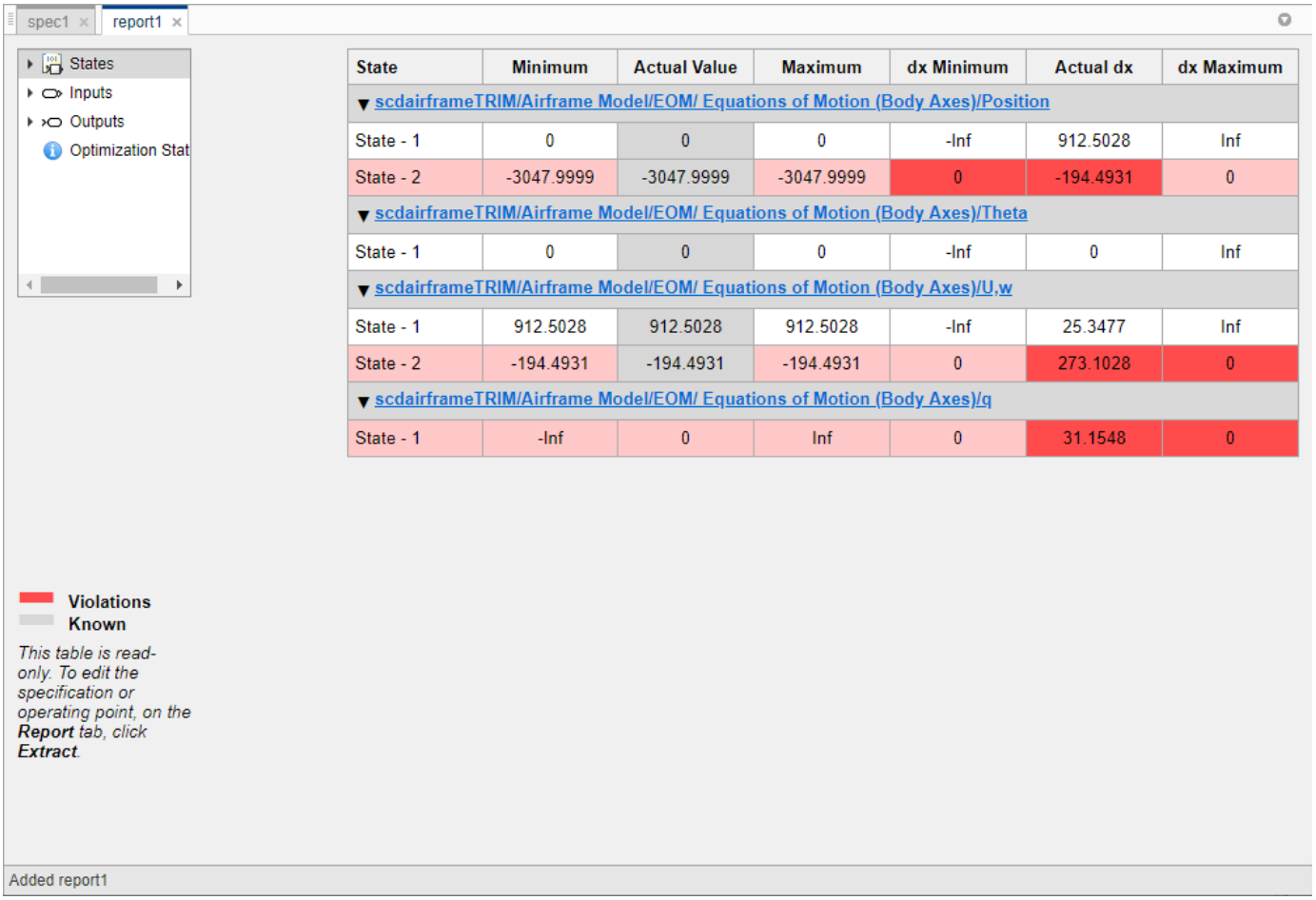
To open the **Steady State Manager**, in the Simulink model window, in the **Apps** gallery, click **Steady State Manager**.

Create a trim specification for the model. On the **Steady State** tab, click **Trim Specification**.

In the **spec1** document, specify which states are known and which are at steady state.

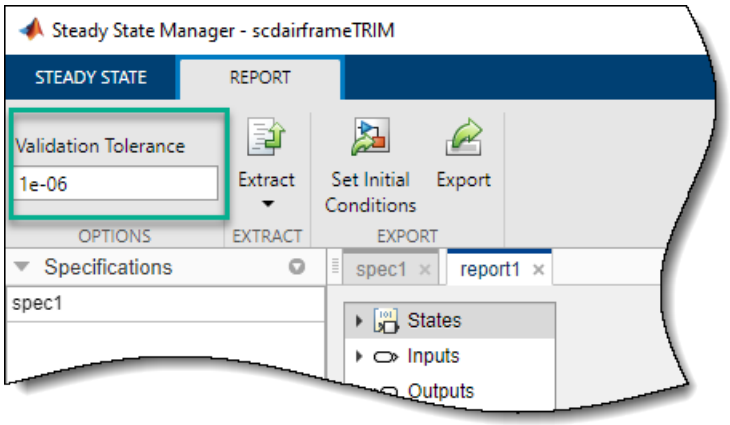


To trim the model, on the **Specification** tab, click **Trim** . The software generates an operating point report and, in the corresponding **report1** document, highlights any constraint violations in red



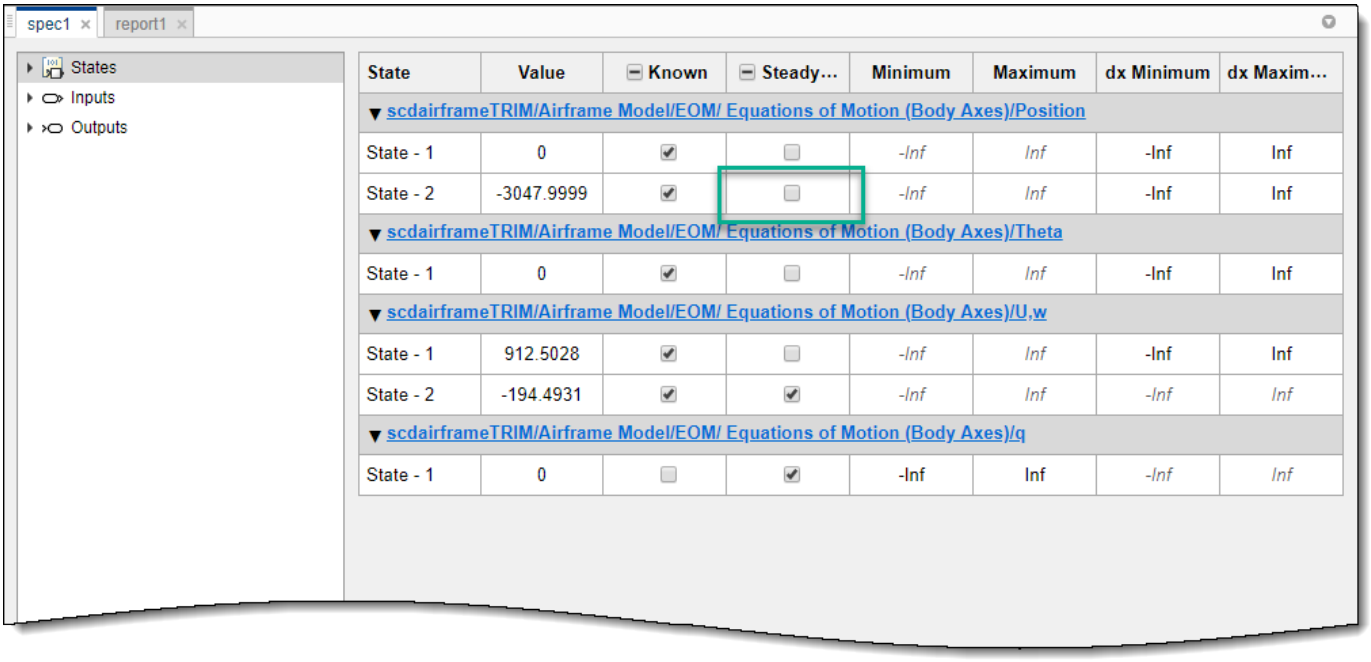
State	Minimum	Actual Value	Maximum	dx Minimum	Actual dx	dx Maximum
▼ scdairframeTRIM/Airframe Model/EOM/ Equations of Motion (Body Axes)/Position						
State - 1	0	0	0	-Inf	912.5028	Inf
State - 2	-3047.9999	-3047.9999	-3047.9999	0	-194.4931	0
▼ scdairframeTRIM/Airframe Model/EOM/ Equations of Motion (Body Axes)/Theta						
State - 1	0	0	0	-Inf	0	Inf
▼ scdairframeTRIM/Airframe Model/EOM/ Equations of Motion (Body Axes)/U,w						
State - 1	912.5028	912.5028	912.5028	-Inf	25.3477	Inf
State - 2	-194.4931	-194.4931	-194.4931	0	273.1028	0
▼ scdairframeTRIM/Airframe Model/EOM/ Equations of Motion (Body Axes)/q						
State - 1	-Inf	0	Inf	0	31.1548	0

The optimization search could not find an operating point that satisfies the specifications. As highlighted in **Steady State Manager**, the three states specified to be at steady state are not. The highlighted state values violate the specified constraints by more than the tolerance value specified on the **Report** tab, in the **Validation Tolerance** field. For steady-state conditions, the **dx Minimum** and **dx Maximum** constraints are both zero; that is, the rate of change for each state is zero. In the trimmed operating point, the **Actual dx** values violate these constraints.



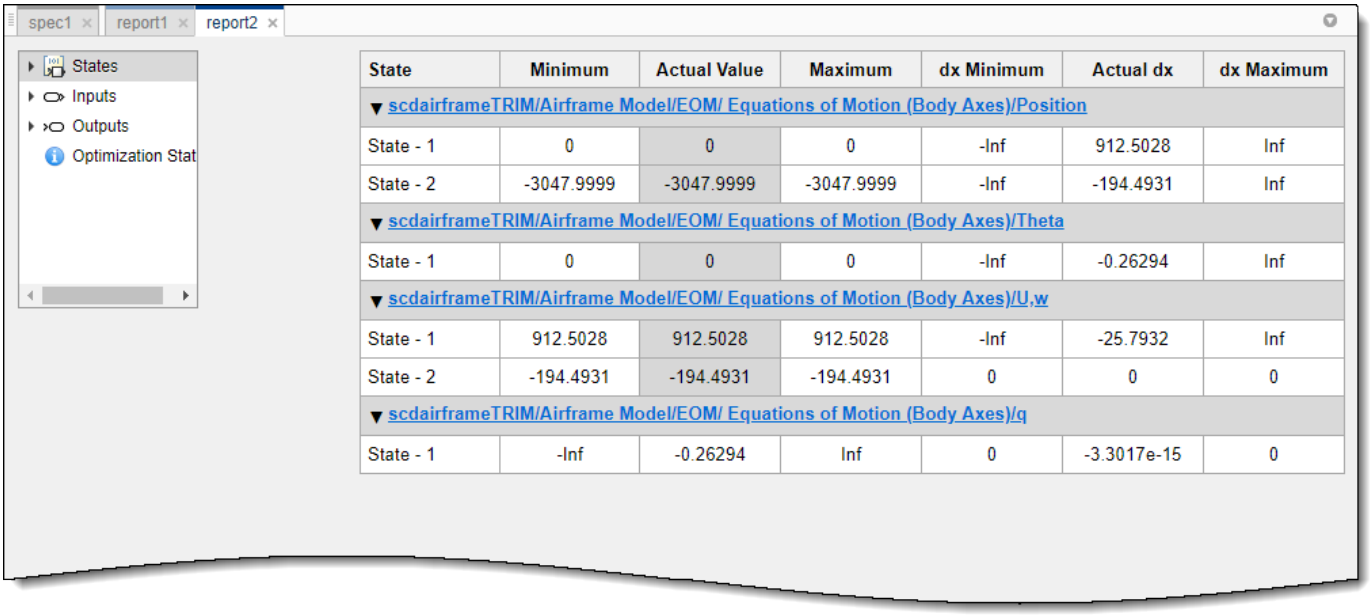
For this model, specifying the second position state to be at steady state overconstrains the system, making a steady-state solution impossible.

To remove this steady-state constraint, update the specification. In the **spec1** document, in the **Steady State** column, clear the corresponding row.



State	Value	Known	Steady...	Minimum	Maximum	dx Minimum	dx Maxim...
▼ scdairframeTRIM/Airframe Model/EOM/ Equations of Motion (Body Axes)/Position							
State - 1	0	☑	☐	-Inf	Inf	-Inf	Inf
State - 2	-3047.9999	☑	☐	-Inf	Inf	-Inf	Inf
▼ scdairframeTRIM/Airframe Model/EOM/ Equations of Motion (Body Axes)/Theta							
State - 1	0	☑	☐	-Inf	Inf	-Inf	Inf
▼ scdairframeTRIM/Airframe Model/EOM/ Equations of Motion (Body Axes)/U,w							
State - 1	912.5028	☑	☐	-Inf	Inf	-Inf	Inf
State - 2	-194.4931	☑	☑	-Inf	Inf	-Inf	Inf
▼ scdairframeTRIM/Airframe Model/EOM/ Equations of Motion (Body Axes)/q							
State - 1	0	☐	☑	-Inf	Inf	-Inf	Inf

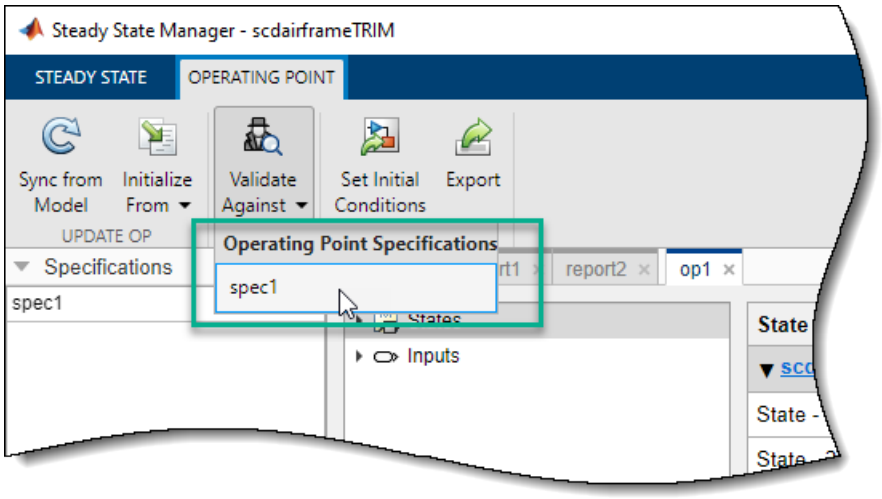
On the **Specification** tab, click **Trim** . The software trims the model and opens a corresponding **Report** tab. The resulting report shows that there are no constraint violations.



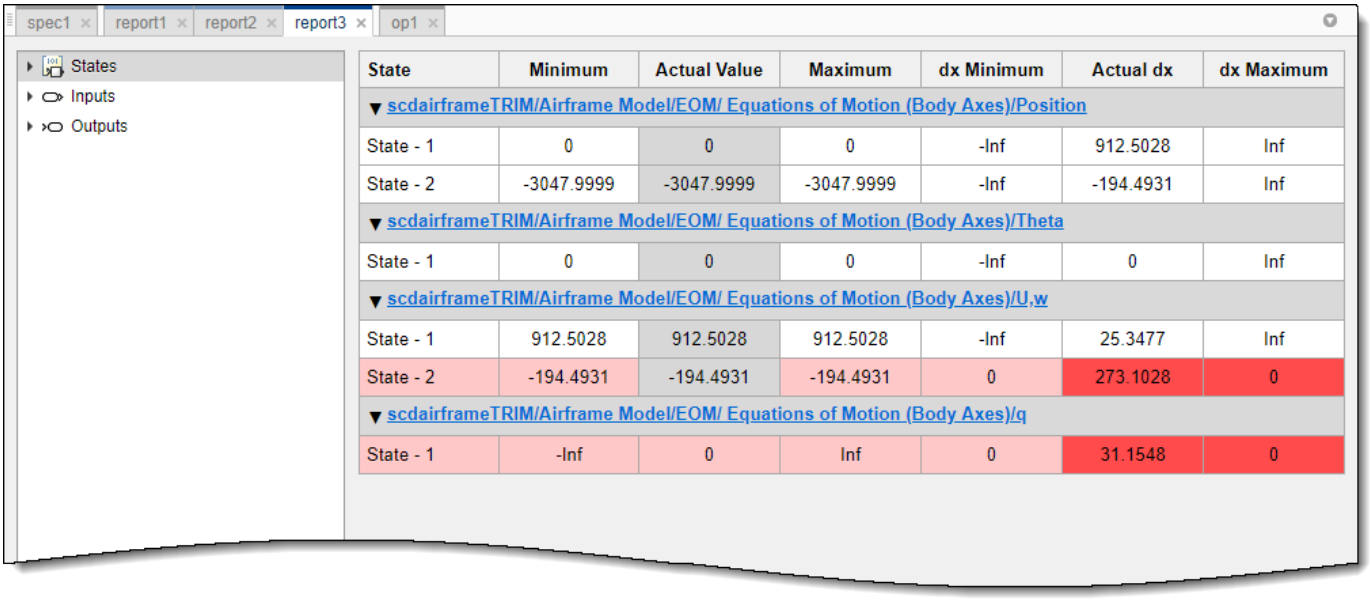
State	Minimum	Actual Value	Maximum	dx Minimum	Actual dx	dx Maximum
▼ scdairframeTRIM/Airframe Model/EOM/ Equations of Motion (Body Axes)/Position						
State - 1	0	0	0	-Inf	912.5028	Inf
State - 2	-3047.9999	-3047.9999	-3047.9999	-Inf	-194.4931	Inf
▼ scdairframeTRIM/Airframe Model/EOM/ Equations of Motion (Body Axes)/Theta						
State - 1	0	0	0	-Inf	-0.26294	Inf
▼ scdairframeTRIM/Airframe Model/EOM/ Equations of Motion (Body Axes)/U,w						
State - 1	912.5028	912.5028	912.5028	-Inf	-25.7932	Inf
State - 2	-194.4931	-194.4931	-194.4931	0	0	0
▼ scdairframeTRIM/Airframe Model/EOM/ Equations of Motion (Body Axes)/q						
State - 1	-Inf	-0.26294	Inf	0	-3.3017e-15	0

You can also validate an existing operating point against a set of specifications. For example, to check if the model initial conditions satisfy the requirements in `spec1`, first create an operating point based on the model initial conditions. On the **Steady State** tab, click **Operating Point**. The software creates an operating point and opens a corresponding **op1** document.

To validate this operating point against the specifications in `spec1`, on the **Operating Point** tab, under **Validate Against**, select `spec1`.



The software creates an operating point report and opens a corresponding **report3** document.



State	Minimum	Actual Value	Maximum	dx Minimum	Actual dx	dx Maximum
▼ scdairframeTRIM/Airframe Model/EOM/ Equations of Motion (Body Axes)/Position						
State - 1	0	0	0	-Inf	912.5028	Inf
State - 2	-3047.9999	-3047.9999	-3047.9999	-Inf	-194.4931	Inf
▼ scdairframeTRIM/Airframe Model/EOM/ Equations of Motion (Body Axes)/Theta						
State - 1	0	0	0	-Inf	0	Inf
▼ scdairframeTRIM/Airframe Model/EOM/ Equations of Motion (Body Axes)/U,w						
State - 1	912.5028	912.5028	912.5028	-Inf	25.3477	Inf
State - 2	-194.4931	-194.4931	-194.4931	0	273.1028	0
▼ scdairframeTRIM/Airframe Model/EOM/ Equations of Motion (Body Axes)/q						
State - 1	-Inf	0	Inf	0	31.1548	0

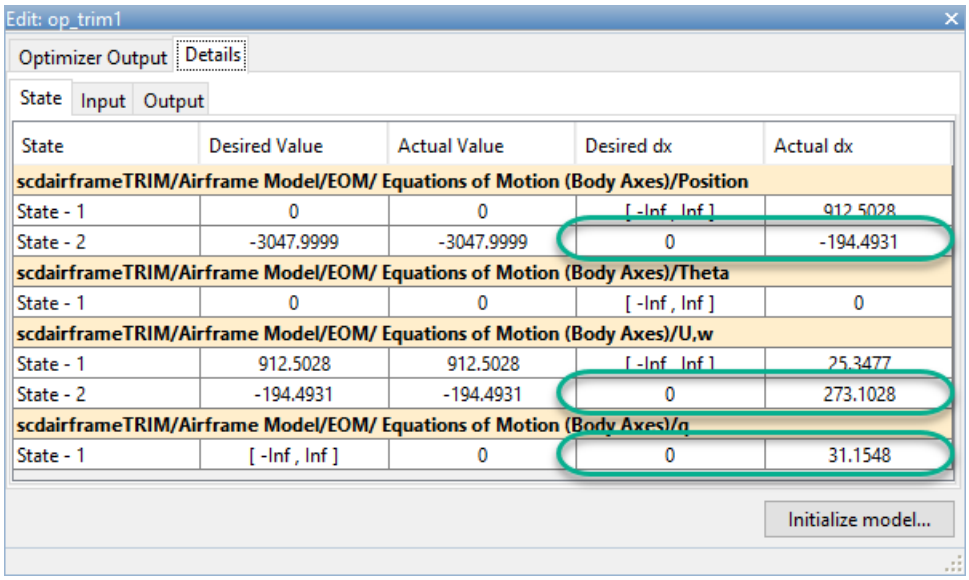
The model initial conditions do not satisfy the operating point specifications, as shown by the highlighted constraint violations.

Validate Operating Point in Model Linearizer

When you compute an operating point using **Model Linearizer**, the software does not highlight constraint violations. Instead, you must inspect the operating point report information for any violations.

If you trim the model from the preceding **Steady State Manager** example using the same specifications in the **Model Linearizer**, the software creates an operating point in the **Data Browser**, in the **Linear Analysis Workspace**.

To check whether the operating point satisfies the specified constraints, in the **Data Browser**, in the **Linear Analysis Workspace**, double-click the operating point.



In the Edit dialog box, for the three steady-state specifications, the trimmed state values in the **Actual dx** column violate the zero **Desired dx** values.

Validate Operating Point at the Command Line

When you compute an operating point at the command line, the `findop` function outputs an operating point report to the Command Window by default. You can also return the operating point report as an output argument. For more information, see `findop`. To validate your operating point against the specifications, you must check whether the operating point values satisfy the constraints.

For example, open the `scdairframeTRIM` model and set the model parameters.

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

alpha_ini = -0.21;
v_ini = 933;
```

Create an operating point specification object, and specify which states are known and which are at steady state.

```
opspec = operspec(sys);
opspec.States(1).Known = [1;1];
```

```
opspec.States(1).SteadyState = [0;1];
opspec.States(3).Known = [1;1];
opspec.States(3).SteadyState = [0;1];
opspec.States(2).Known = 1;
opspec.States(2).SteadyState = 0;
opspec.States(4).Known = 0;
opspec.States(4).SteadyState = 1;
```

Trim the model.

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

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

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

Could not find a solution that satisfies all constraints.  Relax the constraints to find a feasib
States:
----------
      Min         x         Max      dxMin      dx       dxMax
    _______    _______    _______    _____    _______    _____

(1.) scdairframeTRIM/Airframe Model/EOM/ Equations of Motion (Body Axes)/Position
         0          0          0    -Inf       912.5     Inf
      -3048      -3048      -3048       0     -194.49       0
(2.) scdairframeTRIM/Airframe Model/EOM/ Equations of Motion (Body Axes)/Theta
         0          0          0    -Inf           0     Inf
(3.) scdairframeTRIM/Airframe Model/EOM/ Equations of Motion (Body Axes)/U,w
      912.5      912.5      912.5    -Inf       25.348    Inf
    -194.49    -194.49    -194.49       0       273.1       0
(4.) scdairframeTRIM/Airframe Model/EOM/ Equations of Motion (Body Axes)/q
       -Inf          0        Inf       0      31.155       0

Inputs:
----------
    Min     u    Max
    ____    _    ___

(1.) scdairframeTRIM/delta
    -Inf    0    Inf

Outputs:
----------
    Min       y       Max
    ____    ______    ___

(1.) scdairframeTRIM/alpha
    -Inf     -0.21    Inf
(2.) scdairframeTRIM/V
    -Inf       933    Inf
(3.) scdairframeTRIM/q
    -Inf         0    Inf
(4.) scdairframeTRIM/az
    -Inf    263.29    Inf
(5.) scdairframeTRIM/gamma
    -Inf      0.21    Inf
```

In the operating point search report, the `dx` values for the specified steady states have zero constraints, as indicated by the `0` value in parentheses. The optimization search did not find a steady-state operating point, since all three of these states violate the constraints.

See Also

Functions
`findop` | `findopOptions`

Apps
Steady State Manager | **Model Linearizer**

More About

- “Compute Operating Points from Specifications at the Command Line” on page 1-15
- “Compute Operating Points from Specifications Using Steady State Manager” on page 1-20
- “Compute Operating Points from Specifications Using Model Linearizer” on page 1-31

Initialize Steady-State Operating Point Search Using Simulation Snapshot

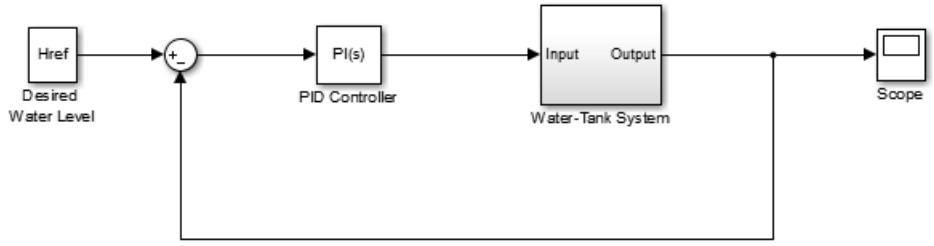
If you know the approximate time when the model reaches the neighborhood of a steady-state operating point, you can use simulation to get state values to use as the initial conditions for numerical optimization.

Initialize Operating Point Search Using Steady State Manager

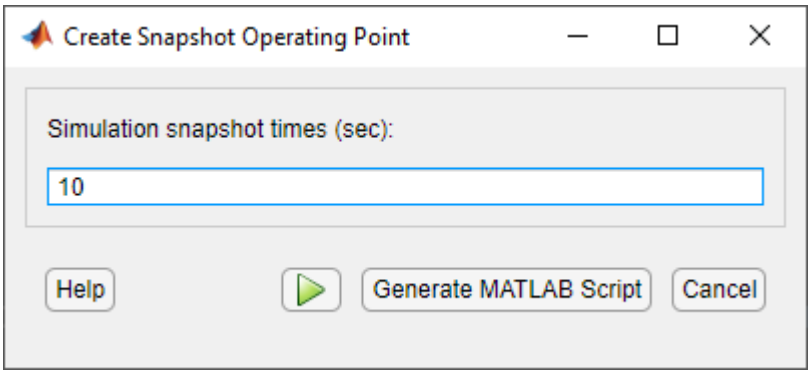
You can initialize operating point searches with a simulation snapshot when computing operating points using the **Steady State Manager**.

1. Open the Simulink model.

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



2. To open the **Steady State Manager**, in the Simulink model window, in the **Apps** gallery, click **Steady State Manager**.
3. In the **Steady State Manager**, on the **Steady State** tab, click **Snapshots**.
4. In the Create Snapshot Operating Point dialog box, enter `10` in the **Simulation snapshot times** field to extract the operating point at this simulation time.

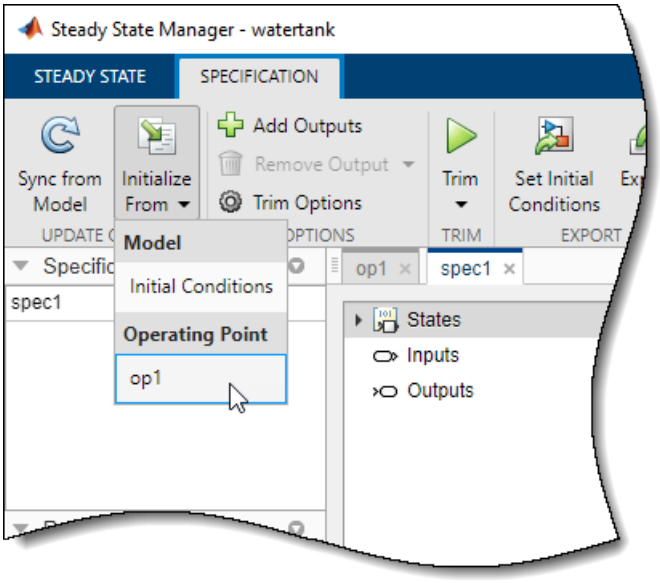


5. To take a snapshot of the system at the specified time, click .

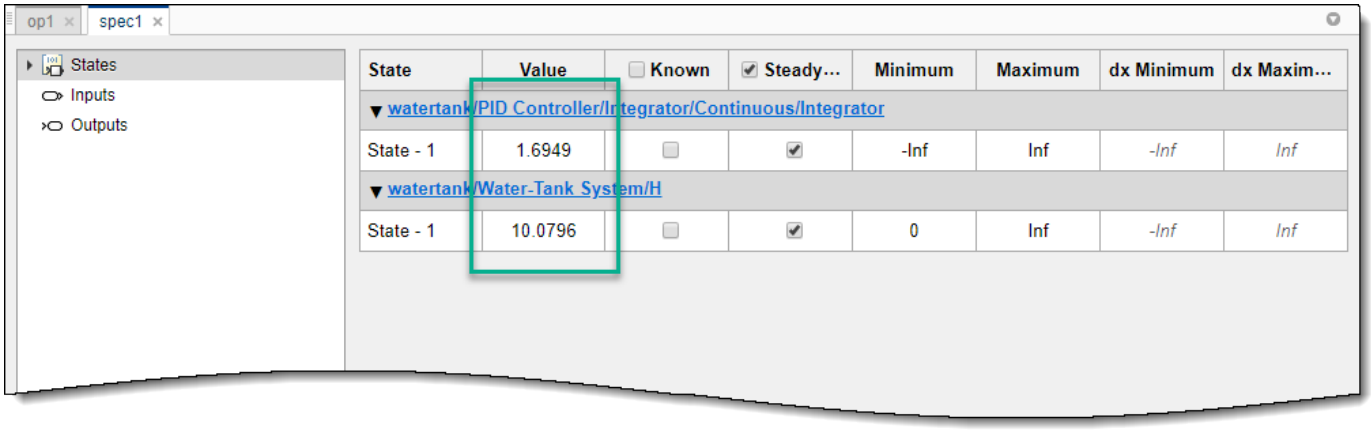
   The snapshot, `op1`, appears in the **Data Browser**, in the **Operating Points** section and contains all of the system state values at the specified time.

6 On the **Steady State** tab, click **Trim Specification**.

7 To Initialize the operating point states with the simulation snapshot values, on the **Specification** tab, click **Initialize From**, and select `op1`.

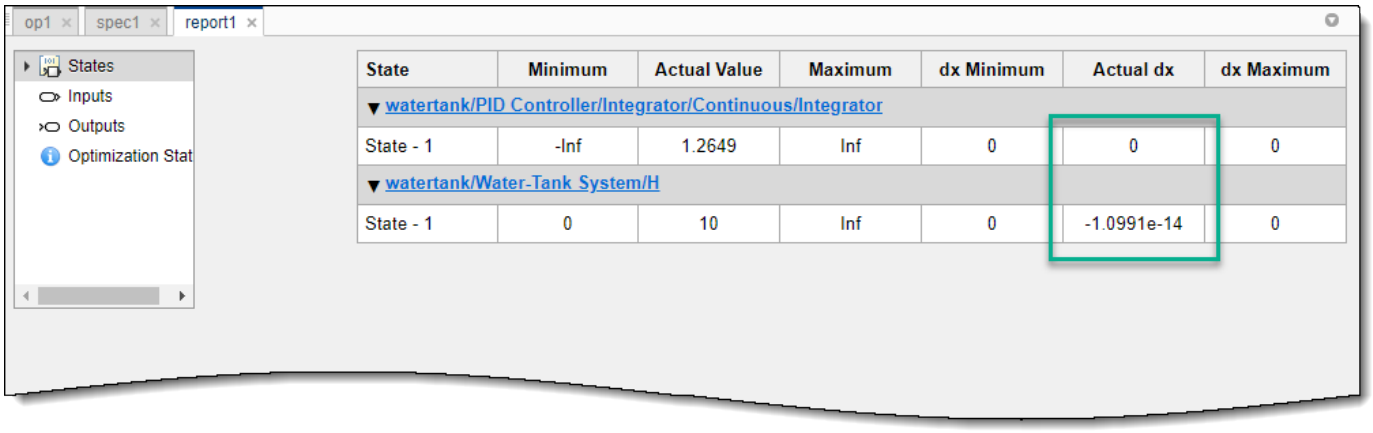


In the **spec1** document, the displayed state values update to reflect the imported values.



8 To find the optimized operating point using the states at `t = 10` as the initial values, on the **Specification** tab, click **Trim**. The software trims the model and generates an operating point search report.

9 In the **report1** document, the **Actual dx** values are at or near zero, showing that the operating point is at a steady state.

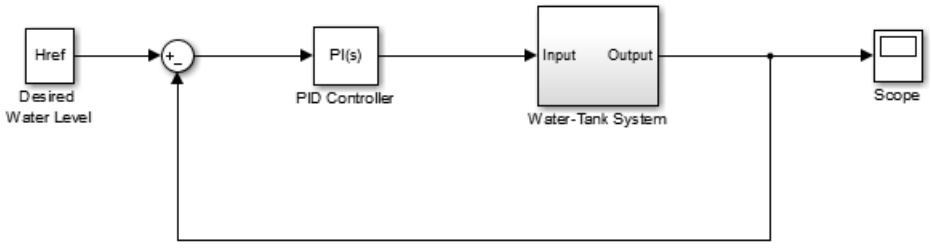


Initialize Operating Point Search Using Model Linearizer

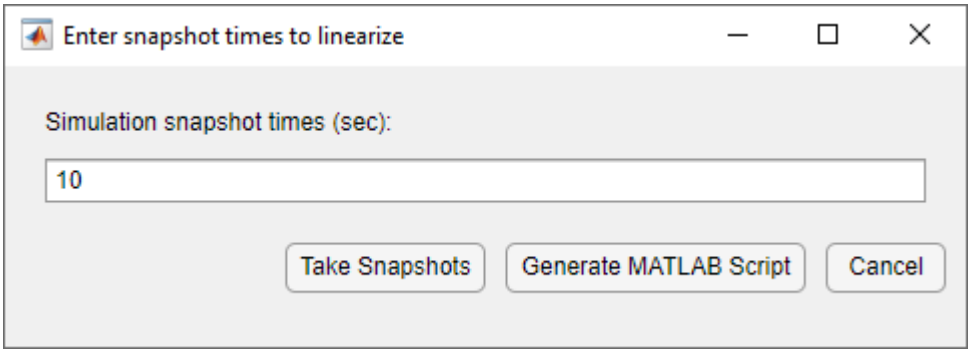
You can initialize operating point searches with a simulation snapshot when computing operating points using the **Model Linearizer**.

1. Open the Simulink model.

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



2. In the Simulink model window, in the **Apps** gallery, click **Model Linearizer**.
3. In the **Model Linearizer**, in the **Operating Point** drop-down list, click `Take Simulation Snapshot`.
4. In the Enter snapshot times to linearize dialog box, enter `10` in the **Simulation snapshot times** field to extract the operating point at this simulation time.



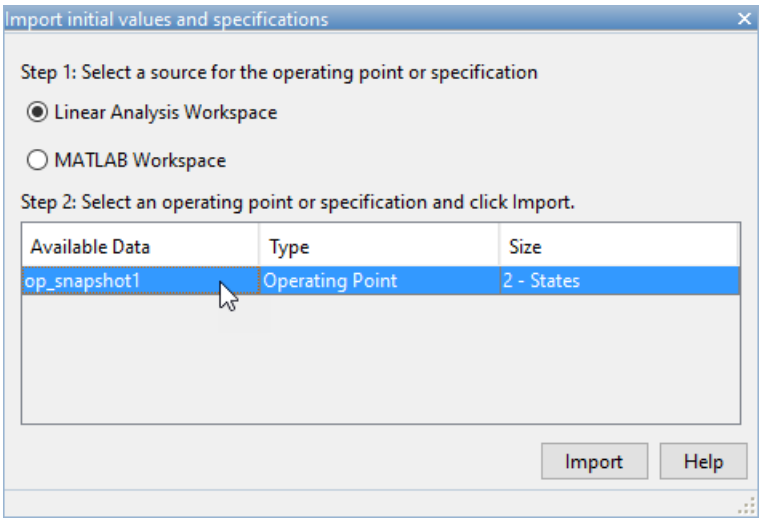
5 To take a snapshot of the system at the specified time, click **Take Snapshots**.

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

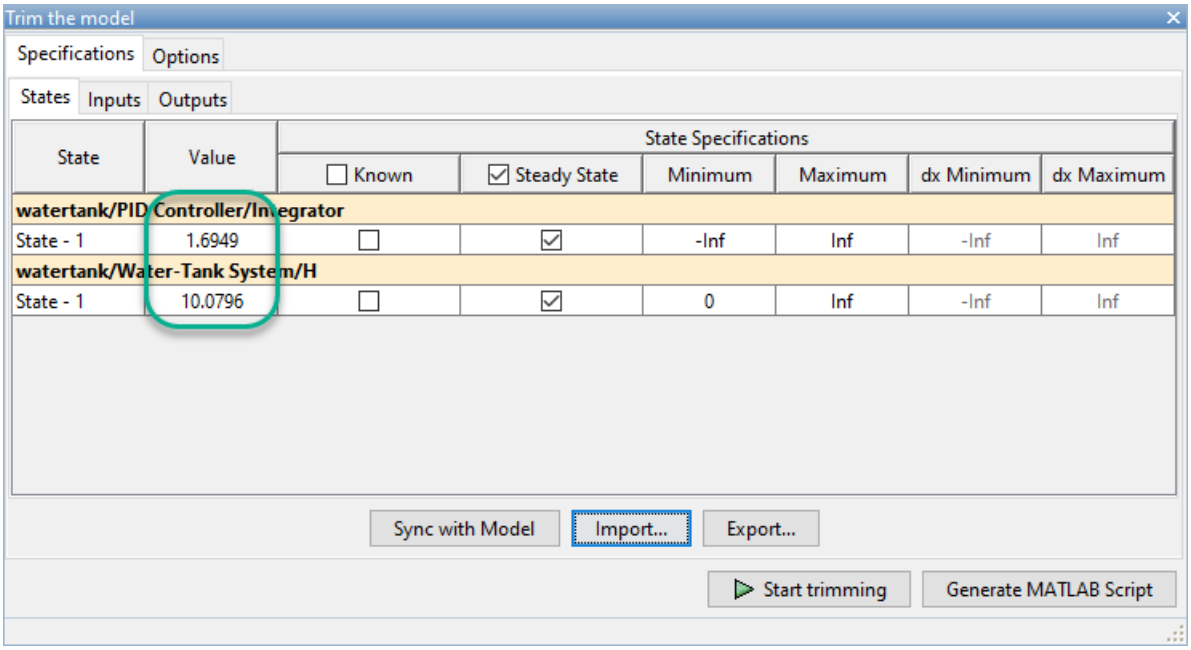
6 On the **Linear Analysis** tab, in the **Operating Point** drop-down list, click `Trim Model`.

7 To Initialize the operating point states with the simulation snapshot values, in the Trim the model dialog box, click **Import**.

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

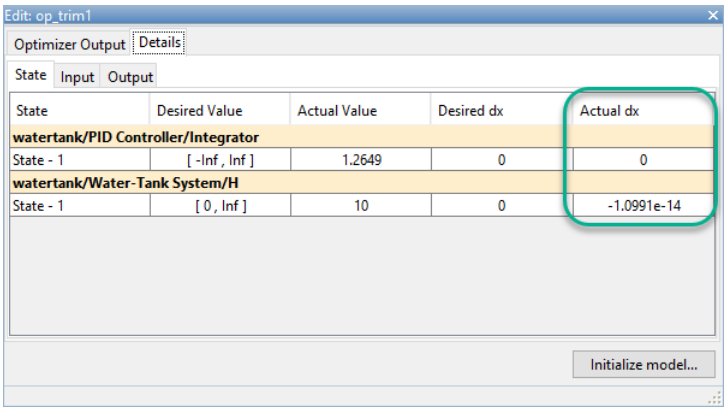


In the Trim the model dialog box, the displayed state values update to reflect the imported values.



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

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



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

Initialize Operating Point Search at the Command Line

You can initialize operating point searches with a simulation snapshot when computing operating points using the `findop` function.

Open the Simulink model.

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

Simulate the model until it reaches a steady state, and extract an operating point snapshot. For this example, use ten time units.

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

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

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

Configure initial values for operating point search using the snapshot data.

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

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

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

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

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

```
Operating point specifications were successfully met.
States:
----------
    Min       x       Max    dxMin        dx         dxMax
    ____    ______    ___    _____    ___________    _____

(1.) watertank/PID Controller/Integrator/Continuous/Integrator
    -Inf    1.2649    Inf      0                0      0
(2.) watertank/Water-Tank System/H
       0        10    Inf      0      -1.0991e-14      0

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

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

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

See Also

`initopspec`

More About

- “Compute Steady-State Operating Points” on page 1-5
- “Change Operating Point Search Optimization Settings” on page 1-51
- “Compute Steady-State Operating Points” on page 1-5

Change Operating Point Search Optimization Settings

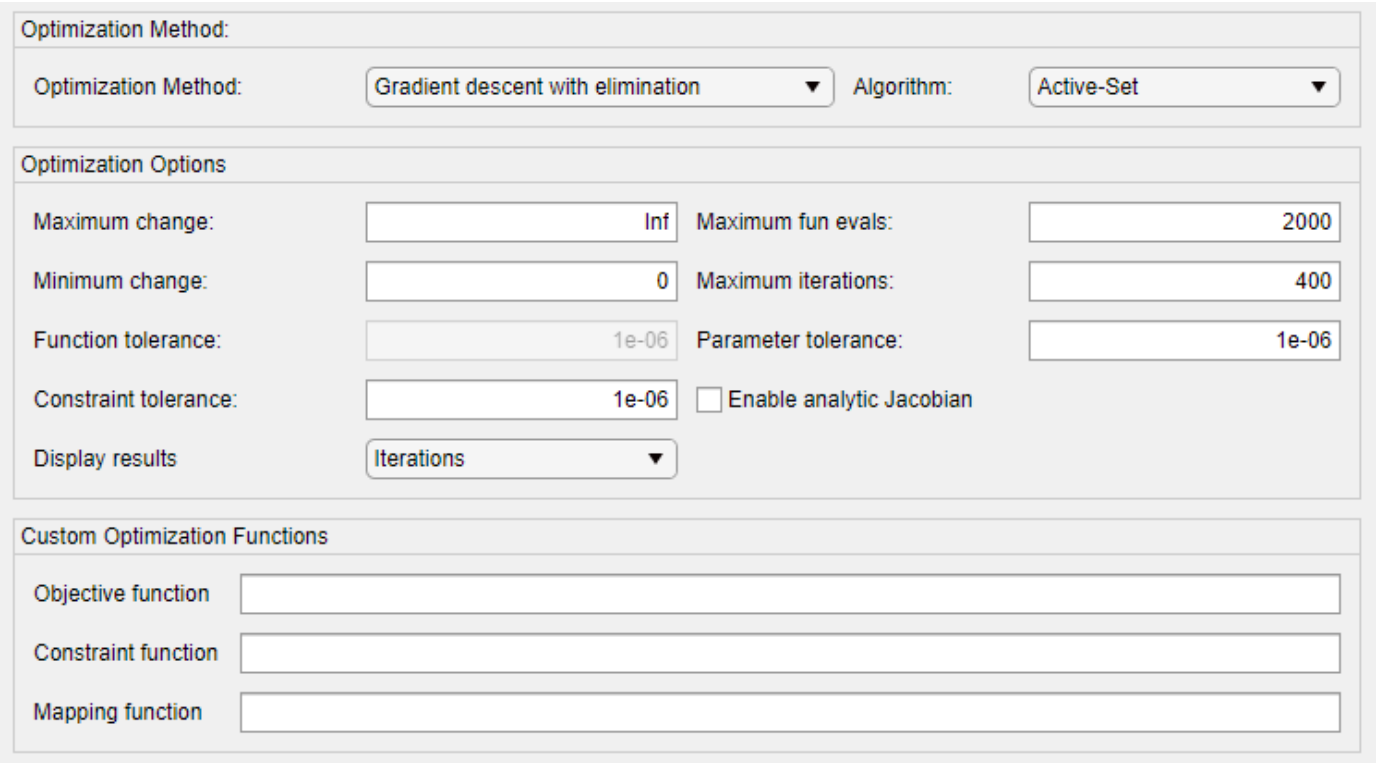
This example shows how to control the accuracy of your operating point search by configuring the optimization algorithm. Typically, you adjust the optimization settings based on the operating point search report, which is automatically created after each search.

You can change your optimization settings when computing operating points interactively using the **Steady State Manager** or **Model Linearizer**, or programmatically using the `findop` function.

Interactively Change Optimization Settings

You can configure the optimization settings for interactively computing operating points using the **Steady State Manager** or **Model Linearizer** using the same trimming options dialog box interface.

- In **Steady State Manager**, on the **Specification** tab, click **Trim Options**. Then, in the **Trim Options** dialog box, specify your optimization settings.
- In **Model Linearizer**, on the **Linear Analysis** tab, in the **Operating Point** drop-down list, click **Trim Model**. Then, in the Trim the model dialog box, on the **Options** tab, specify your optimization settings.



You can specify the **Optimization Method** and corresponding optimization options such as the options shown in the following table.

Optimization Status	Option to Change	Comment
Optimization ends before completing (too few iterations)	**Maximum iterations**	Increase the number of iterations.

Optimization Status	Option to Change	Comment
State derivative or error in output constraint is too large	**Function tolerance** or **Constraint tolerance** (depending on selected algorithm)	Decrease the tolerance value.

You can also specify custom cost and constraint functions for optimization using the **Custom Optimization Functions** parameters. For more information, see “Compute Operating Points Using Custom Constraints and Objective Functions” on page 1-57.

Programmatically Change Optimization Settings

To configure the optimization settings for computing operating points using the `findop` function, create a `findopOptions` option set. For example, create an option set and specify a nonlinear least-squares optimization method.

```
options = findopOptions('OptimizerType','lsqnonlin');
```

To specify options for each optimization method, set the `OptimizationOptions` parameter of the options set to a corresponding structure created using the `optimset` function.

To specify custom cost and constraint functions for optimization, create an `operspec` object and specify the `CustomObjFcn`, `CustomConstrFcn`, and `CustomMappingFcn` properties. For more information, see “Compute Operating Points Using Custom Constraints and Objective Functions” on page 1-57.

See Also

Functions
`findop` | `findopOptions` | `operspec`

Apps
Steady State Manager | **Model Linearizer**

More About

- “Compute Steady-State Operating Points” on page 1-5
- “Compute Steady-State Operating Points” on page 1-5
- “Batch Compute Steady-State Operating Points Reusing Generated MATLAB Code” on page 1-80

Import and Export Specifications for Operating Point Search

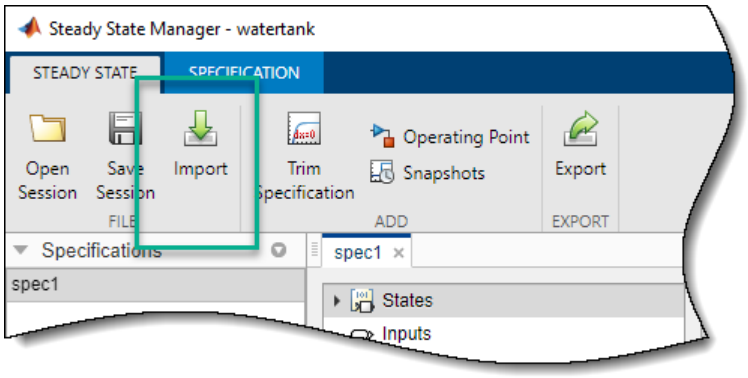
When you modify an operating point specification in the **Steady State Manager** or **Model Linearizer**, you can export the specification to the MATLAB workspace. Exported specifications are saved as operating point specification objects (see `operspec`). Exporting specifications can be useful when you expect to perform multiple trimming operations using the same or a similar set of specifications. Also, you can export interactively edited operating point specifications when you want to use the `findop` command to perform multiple trimming operations with a single compilation of the model. (See “Batch Compute Steady-State Operating Points for Multiple Specifications” on page 1-68.)

You can also import saved operating point specifications to the **Steady State Manager** or **Model Linearizer** 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 and interactively change it. You can then export the modified specification or compute a trimmed operating point from it.

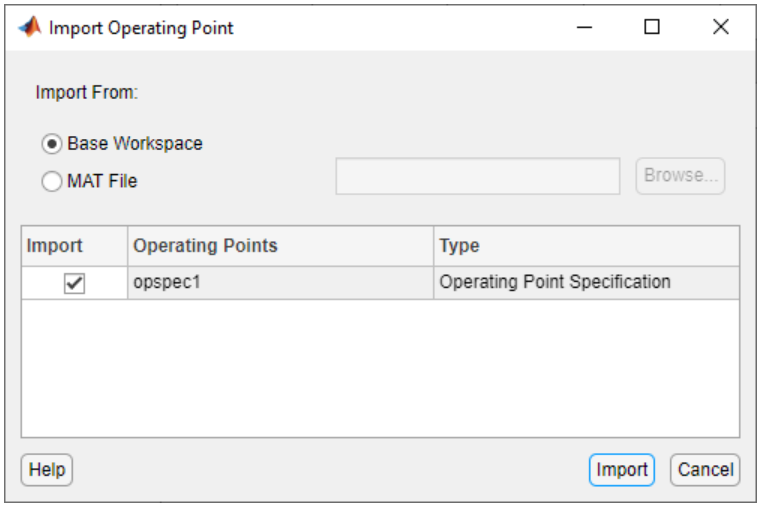
For more information about operating point specifications, see the `operspec` and `findop` reference pages.

Import and Export Specification Using Steady State Manager

To import an operating point specification into the **Steady State Manager**, on the **Steady State** tab, click **Import**.

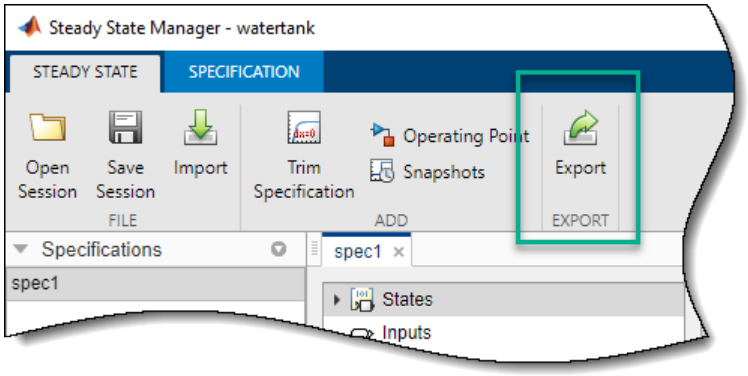


In the Import Operating Point dialog box, select whether you want to import the specification from the MATLAB workspace or from a MAT-file. Then, in the table, in the **Import** column, select the specification that you want to import.



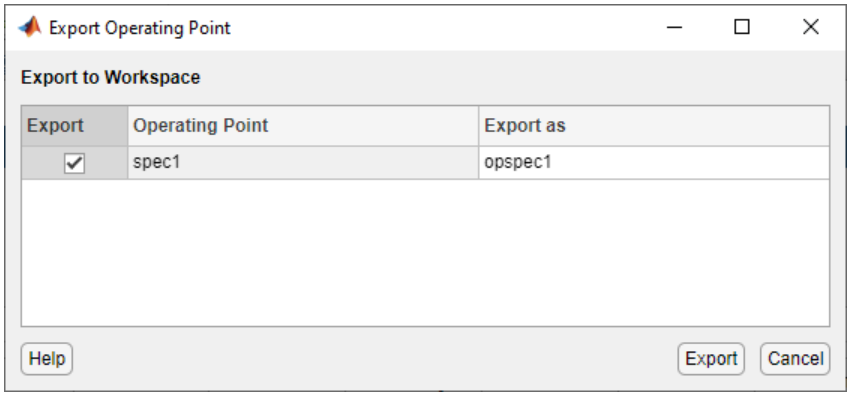
Click **Import**.

To export an operating point specification from the **Steady State Manager**, on either the **Steady State** tab or the **Specification** tab, click **Export**.



In the Export Operating Point dialog box, in the **Export** column, select the specification that you want to export. When you click **Export** from the **Specification** tab, the corresponding specification is preselected in the dialog box.

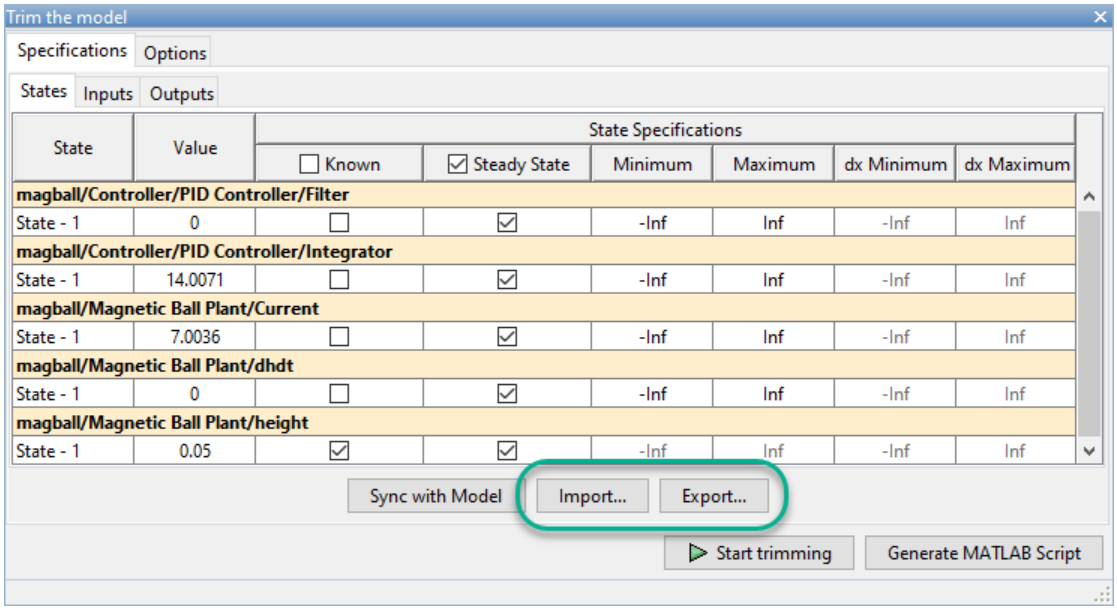
Then, in the **Export as** column, specify the name of the workspace variable to which you want to save the specification.



Click **Export**.

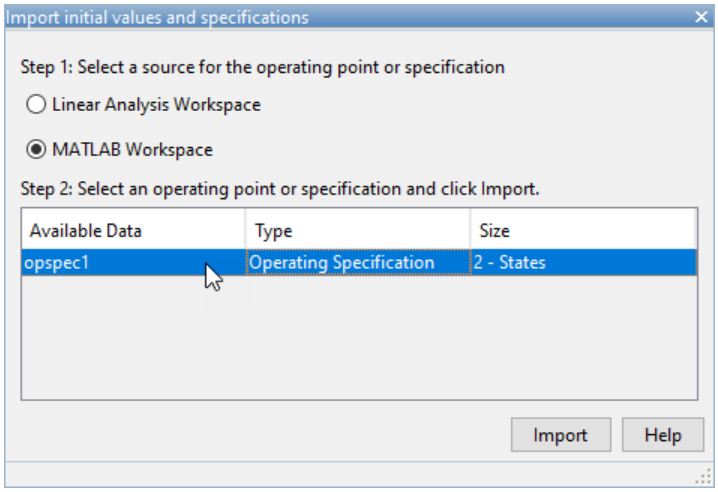
Import and Export Specification Using Model Linearizer

To import or export an operating point specification using the **Model Linearizer**, on the **Linear Analysis** tab, in the **Operating Point** drop-down list, select `Trim Model`.



To import a specification, in the Trim the model dialog box, click **Import**.

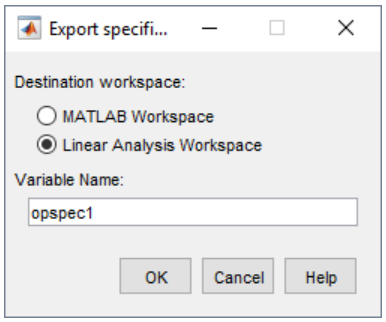
Then, in the Import initial values and specification dialog box, select whether you want to import the specification from the MATLAB workspace or the Linear Analysis Workspace. Then, in the table, click the specification that you want to import.



Click **Import**.

To export a specification, in the Trim the model dialog box, click **Export**.

Then, in the Export Specification dialog box, select whether you want to export the specification to the MATLAB workspace or the Linear Analysis Workspace. Then, in the **Variable Name** field, specify the name of the workspace variable to which you want to save the specification.



Click **Import**.

See Also

Functions

`findop` | `operspec`

Apps

Steady State Manager | **Model Linearizer**

More About

- “View and Modify Operating Points” on page 1-9
- “Batch Compute Steady-State Operating Points for Multiple Specifications” on page 1-68

Compute Operating Points Using Custom Constraints and Objective Functions

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

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

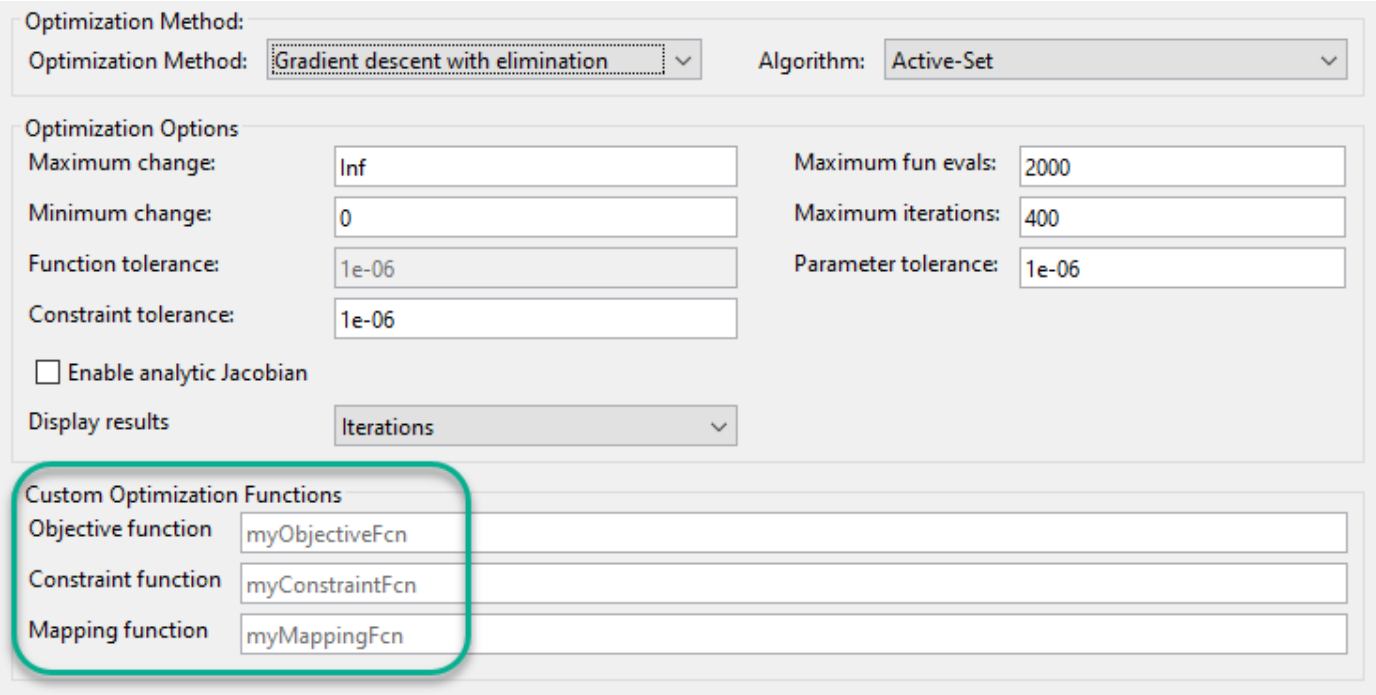
You can specify custom equality and inequality constraints as algebraic combinations of model states, inputs, and outputs. These constraints let you limit the operating point search space by specifying known relationships between inputs, outputs, and states. For example, you can specify that one model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

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

For complex models, you can specify a custom mapping function that selects a subset of the model inputs, outputs, and states to pass to the custom cost and constraint functions.

You can specify custom optimization functions when trimming your model:

- At the command line: Create an operating point specification using `operspec`, and specify the custom functions using the `CustomConstrFcn`, `CustomCostFcn`, and `CustomMappingFcn` properties of the specification.
- Using the **Steady State Manager**: On the **Specification** tab, click **Trim Options**. In the Trim Options dialog box, in the **Custom Optimization Functions** section, specify the function names.
- Using the **Model Linearizer**: On the **Linear Analysis** tab, in the **Operating Point** drop-down list, click **Trim Model**. In the Trim the model dialog box, on the **Options** tab, in the **Custom Optimization Functions** section, specify the function names.

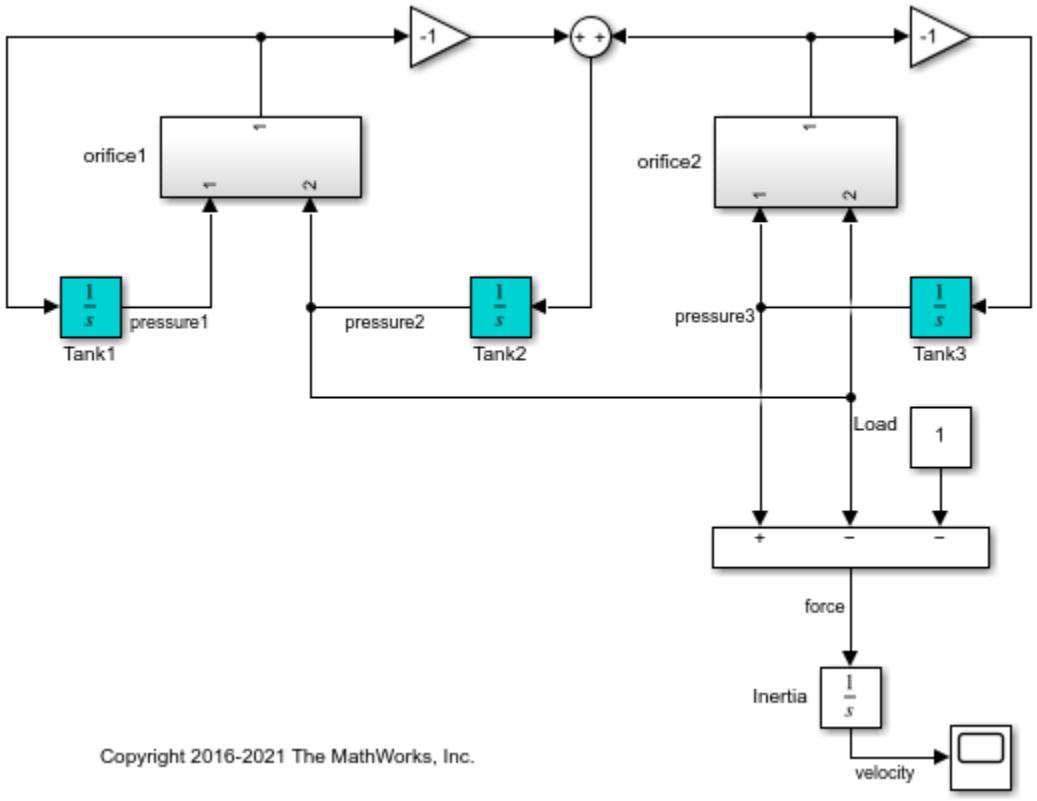


The following example shows how to create custom optimization functions and how to trim a model at the command line using these custom functions.

Simulink Model

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

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



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

At the expected steady state of this system:

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

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

Trim Model Without Customizations

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

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

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

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

Trim the model, and view the trimmed tank pressures.

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


ans =

     <strong>x</strong>
    <strong>____</strong>

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

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

Add Custom Constraints

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

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

and output arguments:

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

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

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

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

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

Add the custom constraint function to the operating point specification.

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

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

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

```
ans =

      <strong>x</strong>
    <strong>______</strong>

(1.) scdTanks/Inertia
         0
(2.) scdTanks/Tank1
    9.3333
(3.) scdTanks/Tank2
    9.3333
(4.) scdTanks/Tank3
    10.333
```

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

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

```
rpt1.CustomEqualityConstr
```

```
ans =

   1.0e-06 *

   -0.0001
   -0.1540
```

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

Add Custom Objective Function

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

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

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

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

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

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

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

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

```
ans =

    <strong>x</strong>
    <strong>__</strong>

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

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

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

```
rpt2.CustomObj
```

```
ans =

     0
```

Add Custom Mapping

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

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

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

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

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

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

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

Add the custom mapping to the operating point specification.

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

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

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



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

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

Add the updated custom functions to the operating point specification.

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

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

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

```
ans =

    <strong>x</strong>
    <strong>__</strong>

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

Add Analytic Gradients to Custom Functions

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

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

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

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

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

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

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

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

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

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

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

```
            0 -1];
end
```

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

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

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

    if x(3) >= 20
        G = [0 0 1]';
    elseif x(3) <= 16
        G = [0 0 -1]';
    else
        G = [0 0 0]';
    end
end
```

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

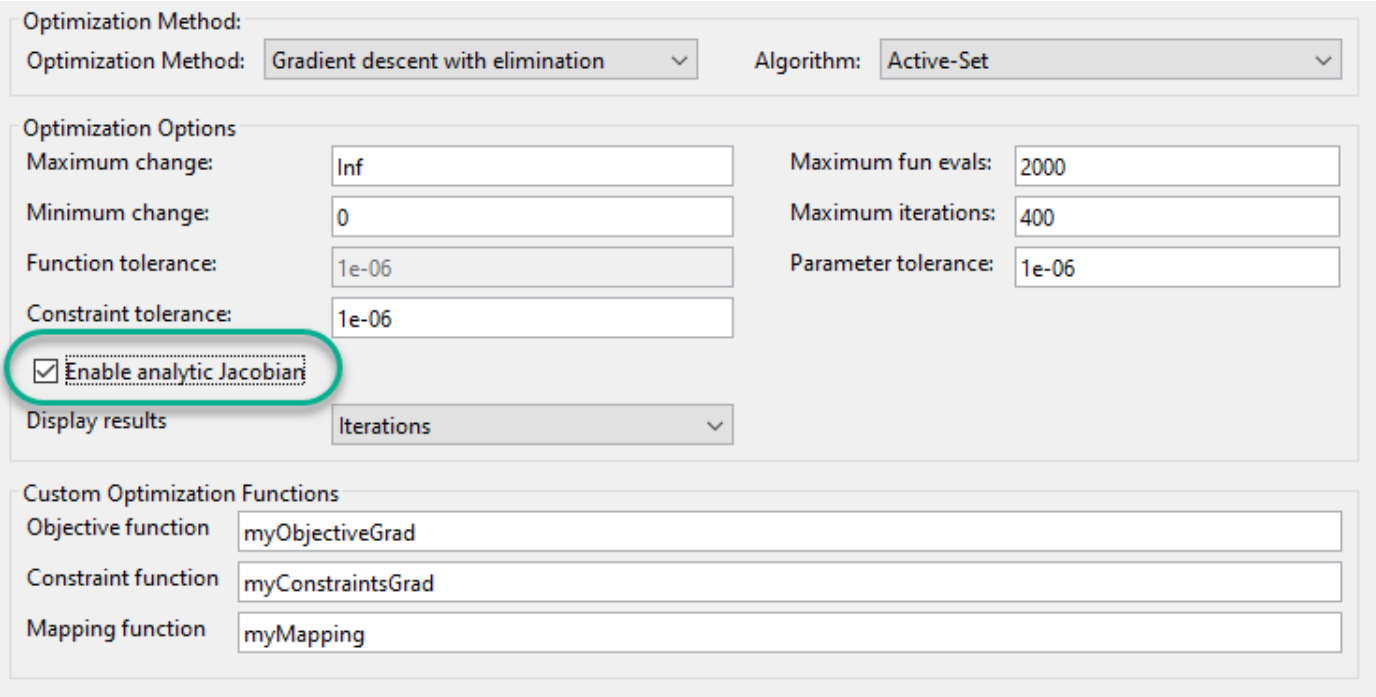
Add the updated custom functions to the operating point specification.

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

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

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

To use analytic Jacobians when trimming models using **Steady State Manager** or the **Model Linearizer**, select the **Enable analytic Jacobian** trimming option.



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

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

```
ans =

    <strong>x</strong>
    <strong>__</strong>

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

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

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

- Without gradients:

```
rpt3.OptimizationOutput.funcCount
```

```
ans =
```

```
    25
```

- With gradients:

```
rpt4.OptimizationOutput.funcCount


ans =

     5
```

For this example, adding the analytical gradients decreases the number of function calls during optimization.

See Also

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

More About

- “Compute Steady-State Operating Points” on page 1-5

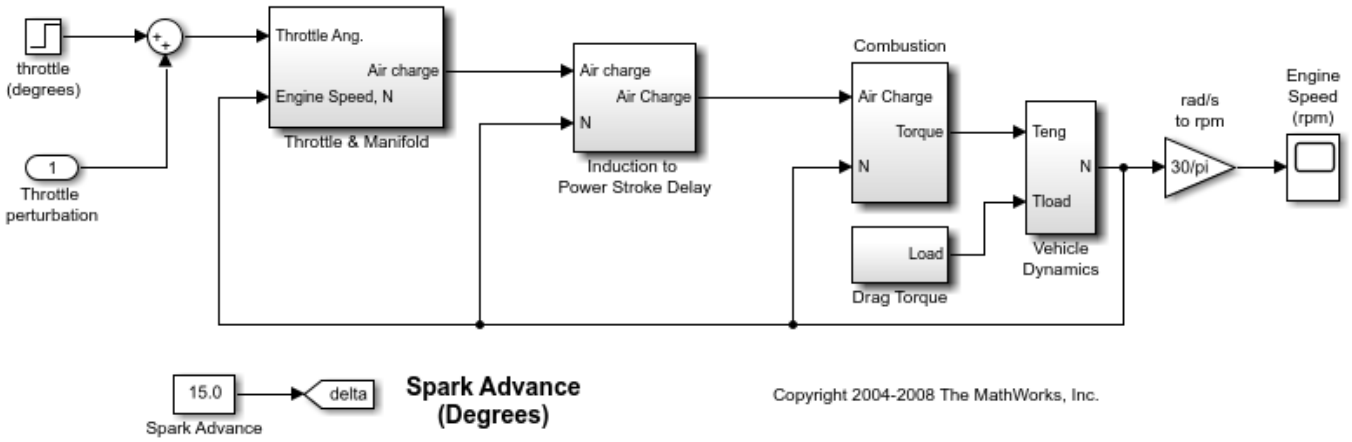
Batch Compute Steady-State Operating Points for Multiple Specifications

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

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

Open the Simulink model.

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



Create an array of default operating point specification objects.

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

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

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

Specify different known output values for each operating point specification.

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

Alternatively, you can configure operating point specifications using the **Model Linearizer** and export the specifications to the MATLAB workspace. For more information, see “Import and Export Specifications for Operating Point Search” on page 1-53.

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

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

```
 Operating point search report 1:
---------------------------------

opreport =


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

Operating point specifications were successfully met.
States:
----------
    <strong>Min</strong>        <strong>x</strong>       <strong>Max</strong>    <strong>dxMin</s
    <strong>____</strong>    <strong>_______</strong>    <strong>___</strong>    <strong>_____</s

(1.) scdspeed/Throttle & Manifold/Intake Manifold/p0 = 0.543 bar
    -Inf    0.59562    Inf      0      3.4112e-09      0
(2.) scdspeed/Vehicle Dynamics/w = T//J w0 = 209 rad//s
    -Inf     157.08    Inf      0      -5.572e-07      0

Inputs:
----------
    <strong>Min</strong>        <strong>u</strong>       <strong>Max</strong>
    <strong>____</strong>    <strong>_______</strong>    <strong>___</strong>

(1.) scdspeed/Throttle  perturbation
    -Inf    -1.6086    Inf

Outputs:
----------
    <strong>Min</strong>      <strong>y</strong>      <strong>Max</strong>
    <strong>____</strong>    <strong>____</strong>    <strong>____</strong>

(1.) scdspeed/rad//s to rpm
    1500    1500    1500

 Operating point search report 2:
---------------------------------

opreport =


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

Operating point specifications were successfully met.
States:
----------
    <strong>Min</strong>        <strong>x</strong>       <strong>Max</strong>    <strong>dxMin</s
    <strong>____</strong>    <strong>_______</strong>    <strong>___</strong>    <strong>_____</s

(1.) scdspeed/Throttle & Manifold/Intake Manifold/p0 = 0.543 bar
```

```
    -Inf    0.54363    Inf      0       2.6649e-13      0
(2.) scdspeed/Vehicle Dynamics/w = T//J w0 = 209 rad//s
    -Inf     209.44    Inf      0      -8.4758e-12      0

Inputs:
----------
    <strong>Min</strong>         <strong>u</strong>        <strong>Max</strong>
    <strong>____</strong>    <strong>_________</strong>    <strong>___</strong>

(1.) scdspeed/Throttle  perturbation
    -Inf    0.0038183    Inf

Outputs:
----------
    <strong>Min</strong>      <strong>y</strong>      <strong>Max</strong>
    <strong>____</strong>    <strong>____</strong>    <strong>____</strong>

(1.) scdspeed/rad//s to rpm
    2000    2000    2000

 Operating point search report 3:
---------------------------------

opreport =


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

Operating point specifications were successfully met.
States:
----------
    <strong>Min</strong>        <strong>x</strong>       <strong>Max</strong>    <strong>dxMin</
    <strong>____</strong>    <strong>_______</strong>    <strong>___</strong>    <strong>_____</

(1.) scdspeed/Throttle & Manifold/Intake Manifold/p0 = 0.543 bar
    -Inf    0.51066    Inf      0       1.3297e-08      0
(2.) scdspeed/Vehicle Dynamics/w = T//J w0 = 209 rad//s
    -Inf      261.8    Inf      0      -7.8334e-08      0

Inputs:
----------
    <strong>Min</strong>       <strong>u</strong>       <strong>Max</strong>
    <strong>____</strong>    <strong>______</strong>    <strong>___</strong>

(1.) scdspeed/Throttle  perturbation
    -Inf    1.4971    Inf

Outputs:
----------
    <strong>Min</strong>      <strong>y</strong>      <strong>Max</strong>
    <strong>____</strong>    <strong>____</strong>    <strong>____</strong>

(1.) scdspeed/rad//s to rpm
    2500    2500    2500
```

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

See Also

`findop` | `operspec`

More About

- “Batch Compute Steady-State Operating Points Reusing Generated MATLAB Code” on page 1-80
- “Batch Compute Steady-State Operating Points for Parameter Variation” on page 1-72
- “Batch Linearize Model at Multiple Operating Points Using linearize Command” on page 3-19
- “Vary Operating Points and Obtain Multiple Transfer Functions Using slLinearizer Interface” on page 3-28

Batch Compute Steady-State Operating Points for Parameter Variation

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

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

Which Parameters Can Be Sampled?

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

Vary Single Parameter

To vary the value of a single parameter for batch trimming with `findop`, specify the parameter grid as a structure having two fields. The `Name` field contains the name of the workspace variable that specifies the parameter. The `Value` field contains a vector of values for that parameter to take during trimming.

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

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

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

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

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

If the variable used by the model is not a scalar variable, specify the parameter name as an expression that resolves to a numeric scalar value. For example, suppose that `Kpid` is a vector of PID

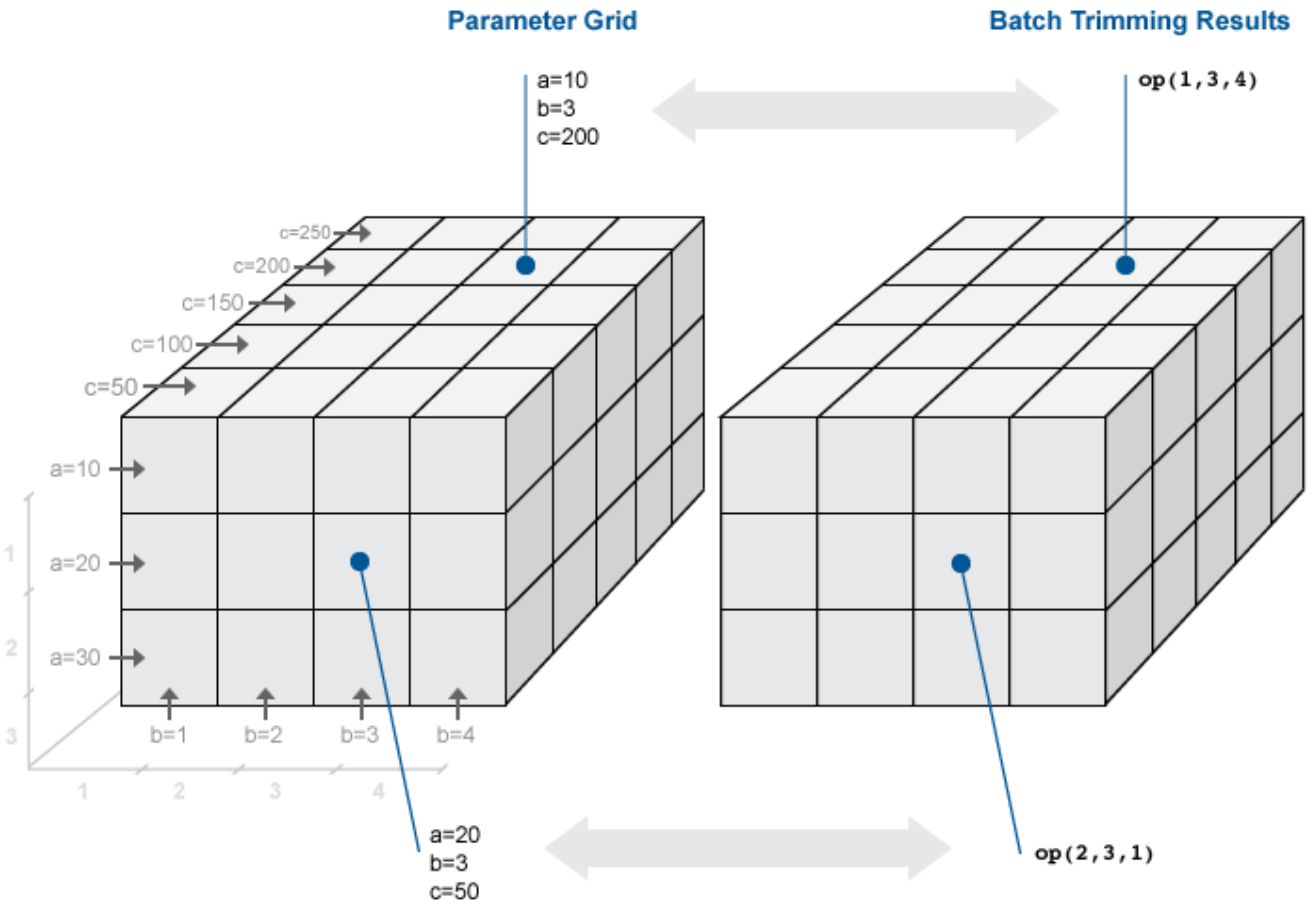
gains. The first entry in that vector, `Kpid`, is used as a gain value in a block in your model. Use the following commands to vary that gain using the values given in a vector `Kpvals`:

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

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

Multidimensional Parameter Grids

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



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

Vary Multiple Parameters

To vary the value of multiple parameters for batch trimming with `findop`, specify parameter samples as a structure array. The structure has an entry for each parameter whose value you vary. The structure for each parameter is the same as described in “Vary Single Parameter” on page 1-72. You can specify the `Value` field for a parameter as an array of any dimension. However, the size of the `Value` field must match for all parameters. Corresponding array entries for all the parameters, also referred to as a parameter grid points, must map to a specified parameter combination. When the software trims the model, it computes an operating point for each grid point.

Specify Full Grid

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

$$a = \{a1, a2\}$$
$$b = \{b1, b2\}$$

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

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

Create a rectangular parameter grid using `ndgrid`.

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

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

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

```
>> A

A =

     1     1
     2     2

>> B

B =

     3     4
     3     4
```

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

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

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

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

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

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

Create a 1 x N structure array.

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

Specify Subset of Full Grid

If your model is complex or you vary the value of many parameters, trimming the model for the full grid can become expensive. In this case, you can specify a subset of the full grid using a table-like approach. Using the example in “Specify Full Grid” on page 1-74, suppose that you want to trim the model for the following combinations of a and b:

$$\{(a_1, b_1), (a_1, b_2)\}$$

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

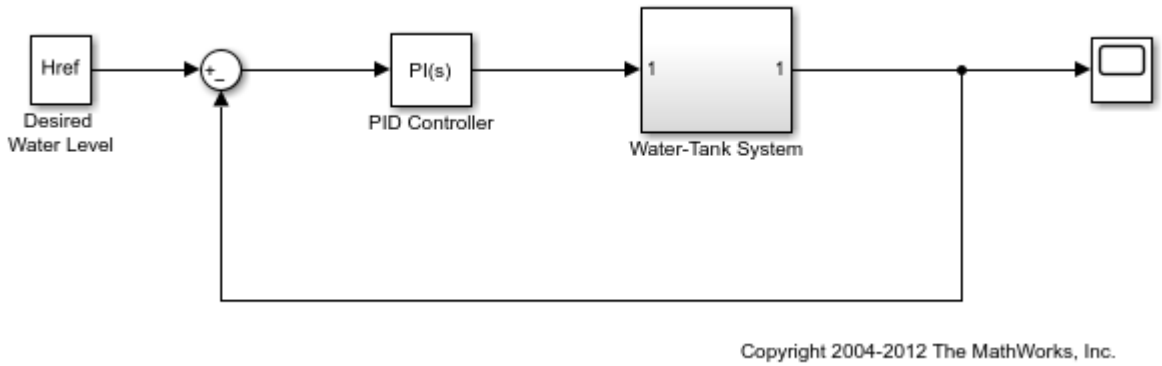
```
A = [a1 a1];
params(1).Name = 'a';
params(1).Value = A;

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

Batch Trim Model for Parameter Variations

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

Open the Simulink model.

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



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

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

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

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

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

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

```
opspec =


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

States:
----------
    <strong>x</strong>    <strong>Known</strong>    <strong>SteadyState</strong>    <strong>Min</
    <strong>_</strong>    <strong>_____</strong>    <strong>___________</strong>    <strong>____<

(1.) watertank/PID Controller/Integrator/Continuous/Integrator
    0    false       true        -Inf    Inf    -Inf      Inf
(2.) watertank/Water-Tank System/H
    1    false       true           0    Inf    -Inf      Inf

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

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

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

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

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

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

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

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

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

```
ans =


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

States:
----------
      <strong>x</strong>
    <strong>______</strong>

(1.) watertank/PID Controller/Integrator/Continuous/Integrator
    1.4055
(2.) watertank/Water-Tank System/H
        10

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

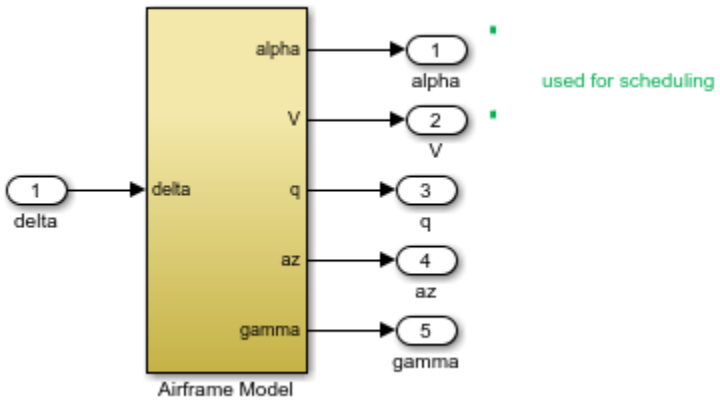
Batch Trim Model at Known States Derived from Parameter Values

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

In the “Batch Trim Model for Parameter Variations” on page 1-75 example, the model is trimmed to meet a single operating point specification that contains unknown states. In other cases, the model states are known for trimming, but depend on the values of the varying parameters. In this case, you cannot batch trim the model using a single operating point specification. You must create a separate specification for each parameter value grid point.

Open the Simulink model.

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



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

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

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

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

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

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

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

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

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

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

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

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

Trim the model using the specified parameter grid and operating point specifications. When you specify an array of operating point specifications and varying parameter values, the dimensions of the specification array must match the parameter grid dimensions.

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

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

See Also

`findop` | `operspec` | `linearize`

More About

- “Batch Compute Steady-State Operating Points for Multiple Specifications” on page 1-68

- “Batch Compute Steady-State Operating Points Reusing Generated MATLAB Code” on page 1-80
- “Batch Linearize Model at Multiple Operating Points Using linearize Command” on page 3-19
- “Vary Operating Points and Obtain Multiple Transfer Functions Using slLinearizer Interface” on page 3-28

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 either simulate or linearize your model at these operating points and study the change in model behavior.

If you are new to writing scripts, interactively configure your operating points search using the **Steady State Manager** or **Model Linearizer**.

Before generating code for batch trimming, first compute an operating point to meet an instance of your specifications. For more information on computing operating points in:

- **Steady State Manager**, see “Compute Operating Points from Specifications Using Steady State Manager” on page 1-20.
- **Model Linearizer**, see “Compute Operating Points from Specifications Using Model Linearizer” on page 1-31.

After computing an operating point, generate a MATLAB script. To do so in the:

- In **Steady State Manager**, on the **Specification** tab, click **Trim**, and select `Script`.
- In **Linear Analysis**, in the Trim the model dialog box, click **Generate MATLAB Script**.

For more information on generating scripts, see “Generate MATLAB Code for Operating Point Configuration” on page 1-110.

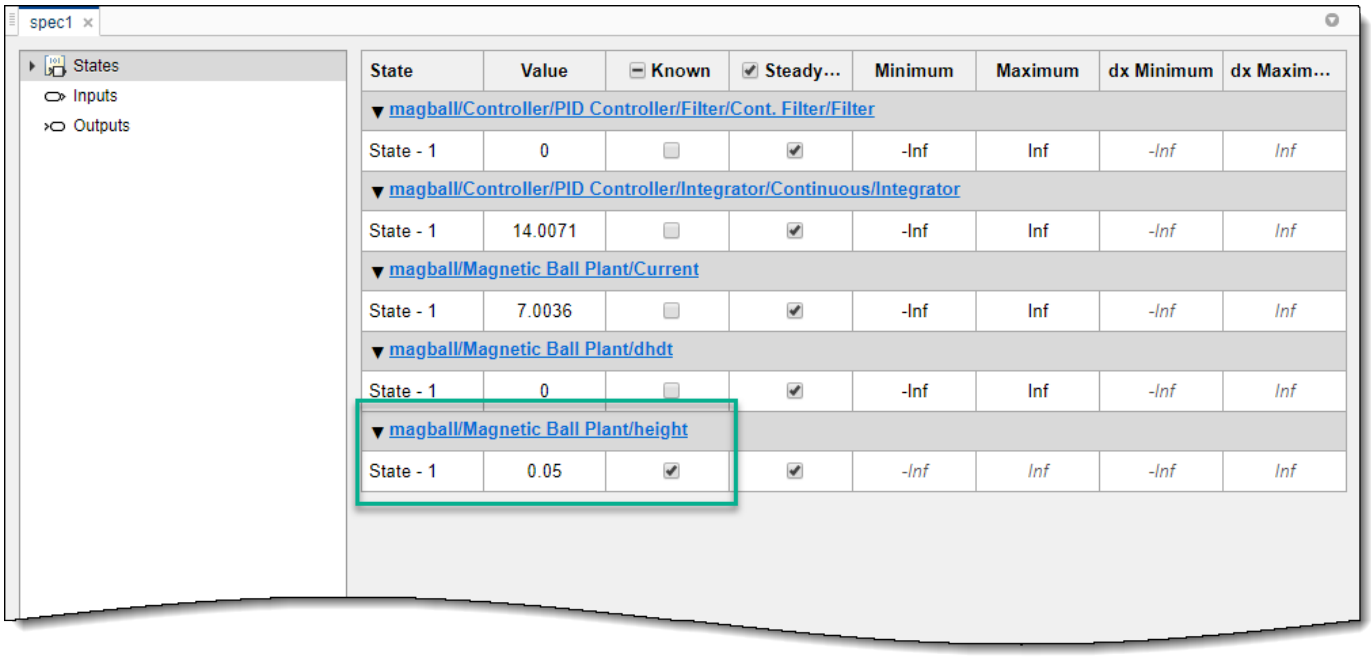
The generated script opens in the MATLAB Editor window. You can then modify the script to trim the model at multiple operating points.

This example demonstrates batch trimming using the `magball` Simulink model.

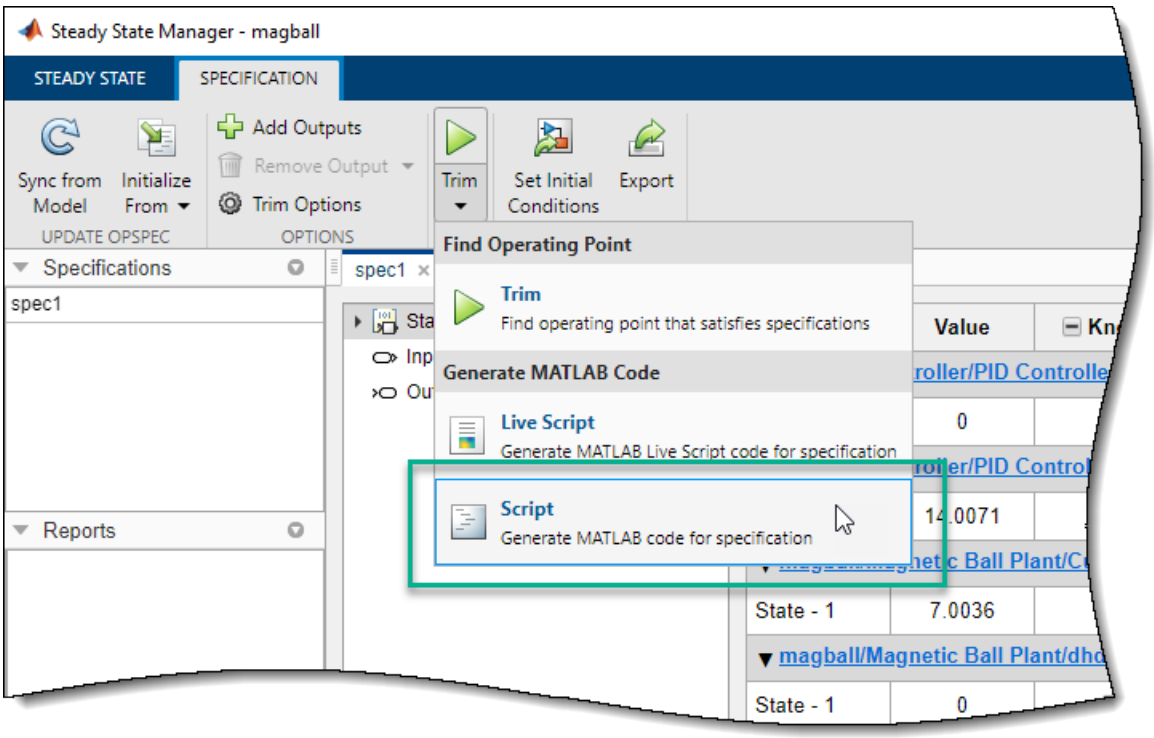
1. Open the model.

   ```
   open_system('magball')
   ```

2. To open the **Steady State Manager**, in the Simulink model window, in the **Apps** gallery, click **Steady State Manager**.
3. On the **Steady State** tab, click **Trim Specification**.
4. In the **spec1** document, in the **Known** column, select the **magball/Magnetic Ball Plant/height** state.



5 Generate the trimming MATLAB code. On the **Specification** tab, click **Trim**, and select `Script`.



6 In the MATLAB Editor window, modify the script to trim the model at multiple operating points.

- **a** Remove unneeded comments from the generated script.
- **b** Define the height variable, `height`, with values at which to compute operating points.

c Add a `for` loop around the operating point search code to compute a steady-state operating point for each `height` value. Within the loop, before calling `findop`, update the reference ball height, specified by the Desired Height block.

Your script should look similar to the following code.

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

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

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

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

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

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

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

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

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

See Also

Apps

Steady State Manager | **Model Linearizer**

Functions

`findop`

More About

- “Generate MATLAB Code for Operating Point Configuration” on page 1-110
- “Batch Linearize Model at Multiple Operating Points Using linearize Command” on page 3-19
- “Vary Operating Points and Obtain Multiple Transfer Functions Using slLinearizer Interface” on page 3-28
- “Batch Compute Steady-State Operating Points for Multiple Specifications” on page 1-68
- “Batch Compute Steady-State Operating Points for Parameter Variation” on page 1-72

Find Operating Points at Simulation Snapshots

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

To use simulation-based operating point computation, first configure your model initial conditions such that the model converges to an equilibrium point. You can then simulate your model and create operating points interactively using **Steady State Manager** or **Model Linearizer**. You can also find snapshots programmatically at the MATLAB command line using the `findop` function.

To find operating points using snapshots, the software simulates the model and creates an operating point at each simulation snapshot time. Each operating point contains the input and states values of the model at the corresponding snapshot time.

To verify that the operating point is at steady state, initialize your model with the operating point values, simulate the model, and check if key signals and states are at equilibrium. For more information on initializing your model with an operating point, see “Simulate Simulink Model at Specific Operating Point” on page 1-93.

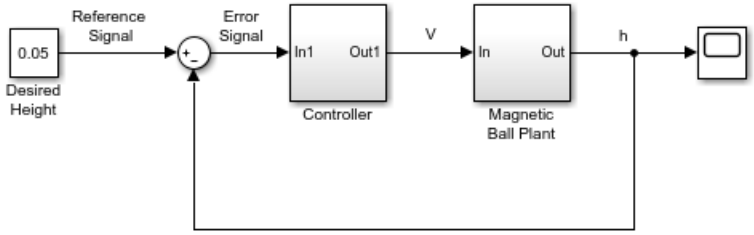
Note If your Simulink model has internal states, do not linearize the model at an operating point you compute from a simulation snapshot. Instead, try linearizing the model using a simulation snapshot or at an operating point from optimization-based search. For more information, see “Handle Blocks with Internal State Representation” on page 1-96.

Compute Operating Points at Simulation Snapshots Using Steady State Manager

You can find an operating point at specified simulation snapshot times using the **Steady State Manager**.

Open the Simulink model.

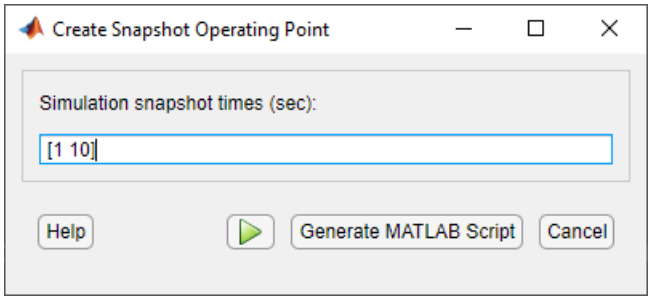
```
sys = 'magball';
open_system(sys)
```



To open the **Steady State Manager**, in the Simulink model window, in the **Apps** gallery, click **Steady State Manager**.

To specify the simulation snapshot time, in the **Steady State Manager**, on the **Steady State** tab, click **Snapshots**.

Specify simulation times at which to take snapshots. For this example, take snapshots at `1` and `10` time units. In the Create Snapshot Operating Point dialog box, in the **Simulation snapshot times** field, enter `[1 10]`.

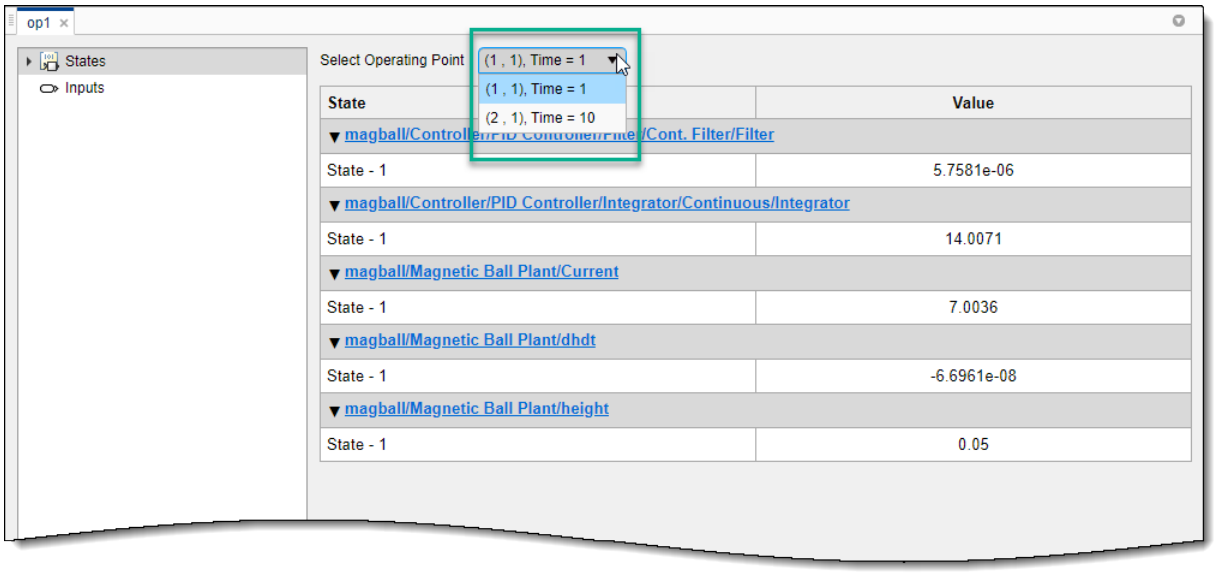


To take the snapshots, click .

An array of operating points, `op1`, appears in the **Data Browser**, in the **Operating Points** section. This array contains two operating points, one for each specified snapshot time.

The software also opens a corresponding **op1** document where you can view the operating points.

To select which operating point to view, use the **Select Operating Point** drop-down list.

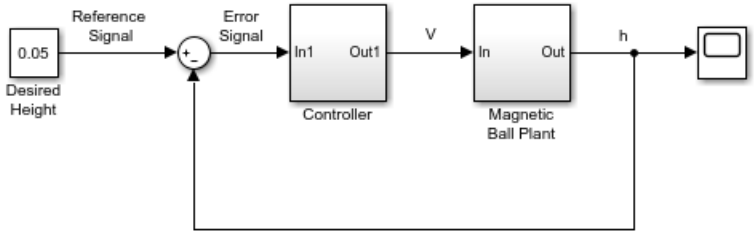


Compute Operating Points at Simulation Snapshots Using Model Linearizer

You can find an operating point at specified simulation snapshot times using the **Model Linearizer**.

Open the Simulink model.

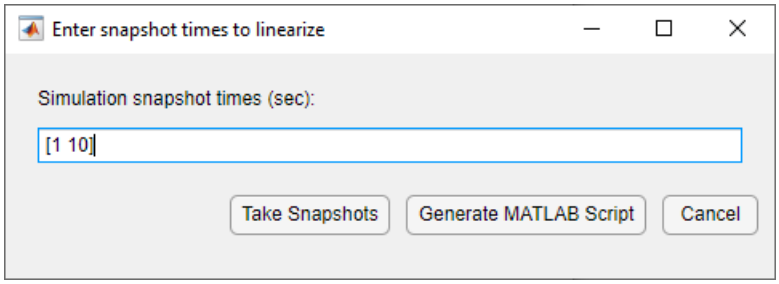
```
sys = 'magball';
open_system(sys)
```



To open the **Model Linearizer**, in the Simulink model window, in the **Apps** gallery, click **Model Linearizer**.

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

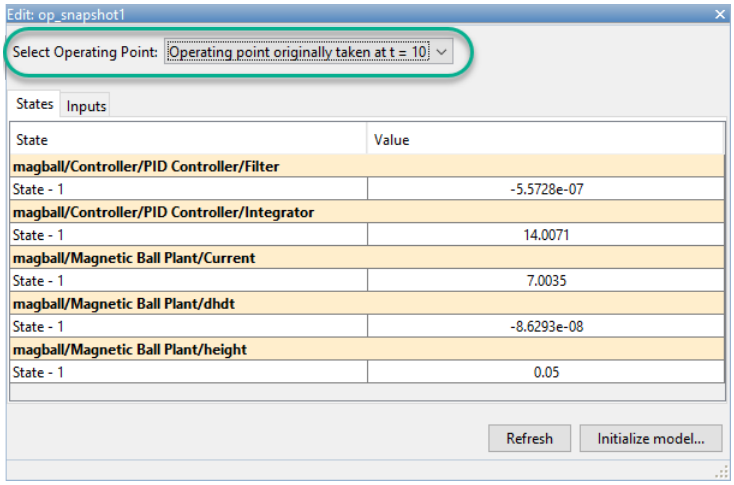
Take simulation snapshots at `1` and `10` time units. In the Enter snapshot times to linearize dialog box, in the **Simulation snapshot times** field, enter `[1 10]`.



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

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

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

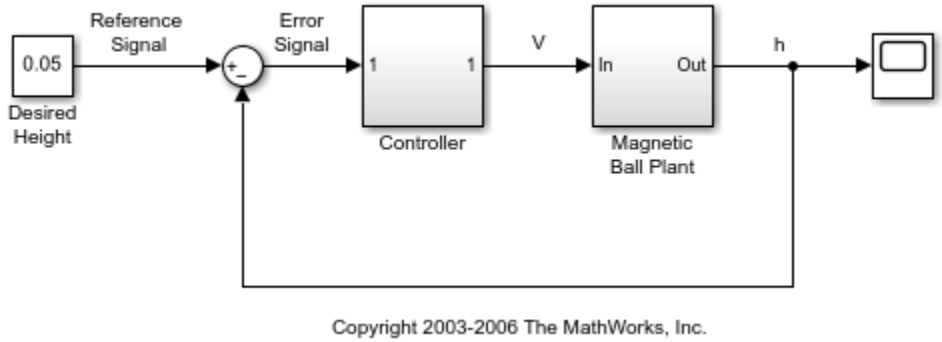


Find Operating Points at Simulation Snapshots at Command Line

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

Open the Simulink model.

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



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

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

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

Display the first operating point.

```
op(1)
```

```
ans =


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

States:
----------
         <strong>x</strong>
    <strong>___________</strong>

(1.) magball/Controller/PID Controller/Filter/Cont. Filter/Filter
     5.7581e-06
(2.) magball/Controller/PID Controller/Integrator/Continuous/Integrator
          14.007
(3.) magball/Magnetic Ball Plant/Current
          7.0036
(4.) magball/Magnetic Ball Plant/dhdt
    -6.6961e-08
(5.) magball/Magnetic Ball Plant/height
           0.05
```

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

See Also

Apps
Model Linearizer

Functions
`findop`

More About

- “About Operating Points” on page 1-2
- “Simulate Simulink Model at Specific Operating Point” on page 1-93
- “Initialize Steady-State Operating Point Search Using Simulation Snapshot” on page 1-45

Compute Operating Point Snapshots at Triggered Events

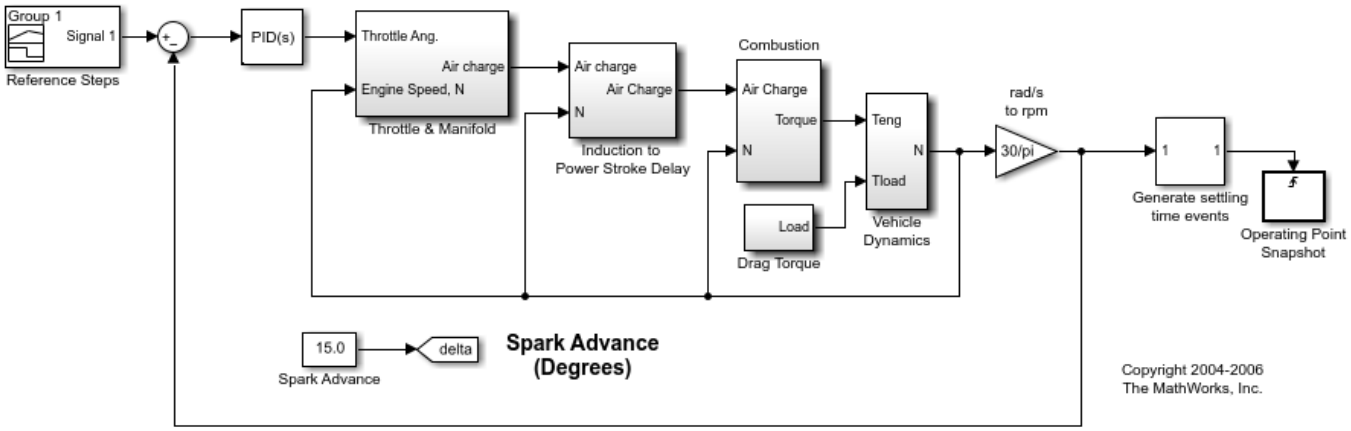
This example shows how to generate operating points using triggered simulation snapshots.

Open Model

The model for this example is a speed control system.

Open the model.

```
mdl = "scdspeedtrigger";
open_system(mdl)
```



The Reference Steps block generates a reference signal that steps through three steady-state speed conditions: 2500, 3000, and 3500 rpm. In this example, you find operating points at each of these conditions by taking operating point snapshots.

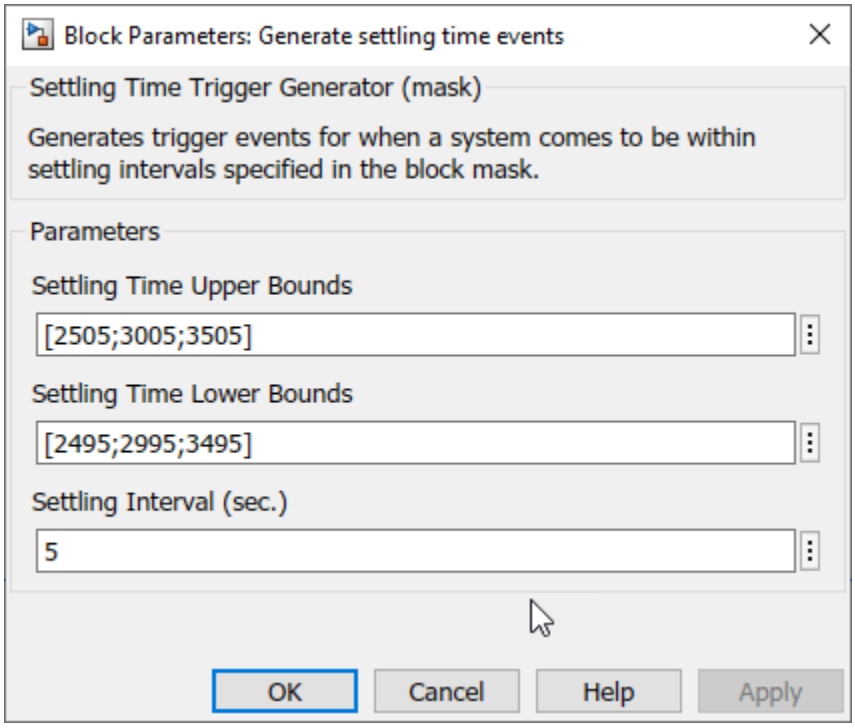
Configure Settling Time Events

Since the exact time that a system reaches a steady-state condition is not always known, you can configure your model to detect when a steady-state condition occurs and generate corresponding trigger events.

For this example, the Generate settling time events subsystem detects when the speed signal near a steady-state settling point. The block generates a trigger event when the input signal is within a specified region near the settling point for a minimum amount of time.

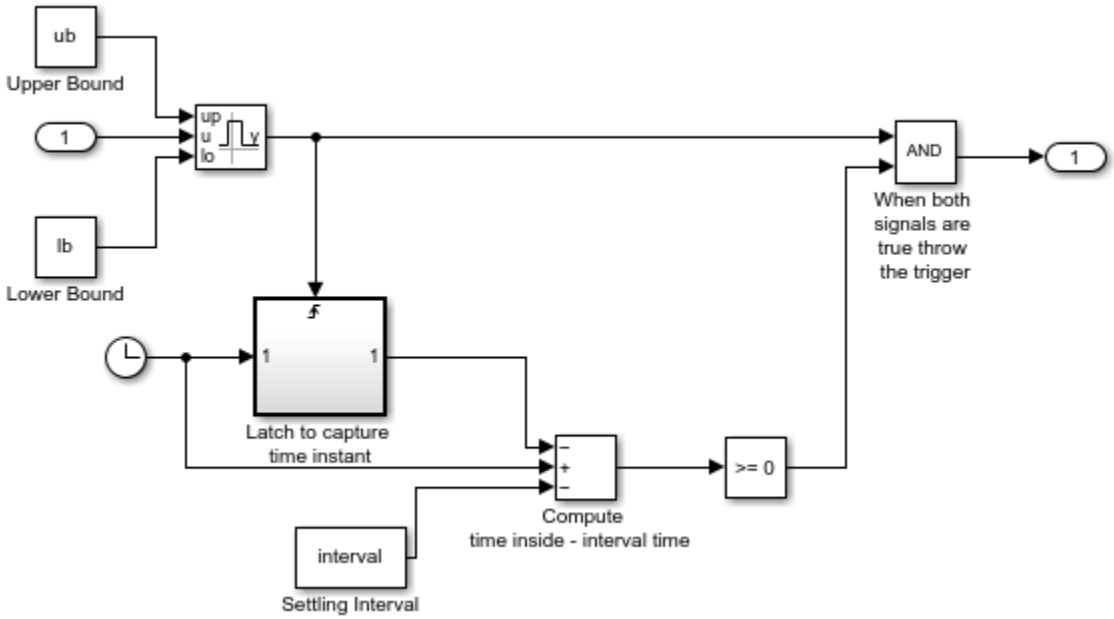
For this example, you define regions near the three steady-state speed values. Open the block and specify the upper and lower bounds for these ranges to be 5 rpm above and below the steady-state speed values. To do so, set the **Settling Time Upper Bounds** and **Settling Time Lower Bounds** parameters.

Also, specify a minimum settling interval of 5 seconds using the **Settling Interval** parameter.



Within the Generate settling time events subsystem:

- When the input signal is within the specified upper and lower bounds, the Interval Test Dynamic block outputs a `true` signal.
- The Interval Test Dynamic block output changing from `false` to `true` triggers a latching mechanism to track how long the signal is `true`.
- When the signal is `true` for a specified interval time, the latching mechanism outputs a `true` signal.
- When the outputs of the Interval Test Dynamic block and the latching mechanism are both `true`, the output trigger signal is set to `true`.



The trigger signal from the Generate settling time events subsystem connects to a Trigger-Based Operating Point Snapshot block. You can configure this block to take operating point snapshots on the rising or falling edge of a trigger signal. For this example, the block uses the rising edge of the trigger signal.

Find Operating Points

To compute the operating points, use the `findop` function to simulate the model for 60 seconds. This function returns a vector of four operating points: one for each triggered snapshot time and one at a simulation time of 60 seconds.

```
op = findop(mdl,60);
```

The first operating point is near the 2500 rpm (261.8 rad/s) settling condition.

```
op(1)
```

```
ans =


 Operating point for the Model scdspeedtrigger.
 (Time-Varying Components Evaluated at time t=10.63)

States:
----------
       <strong>x</strong>
    <strong>_______</strong>

(1.) scdspeedtrigger/PID Controller/Filter/Cont. Filter/Filter
          0
(2.) scdspeedtrigger/PID Controller/Integrator/Continuous/Integrator
      10.47
```

```
(3.) scdspeedtrigger/Throttle & Manifold/Intake Manifold/p0 = 0.543 bar
    0.51066
(4.) scdspeedtrigger/Vehicle Dynamics/w = T//J w0 = 209 rad//s
      261.8

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

The second operating point is near the 3000 rpm (314.16 rad/s) settling condition.

```
op(2)
```

```
ans =


 Operating point for the Model scdspeedtrigger.
 (Time-Varying Components Evaluated at time t=28.3703)

States:
----------
       <strong>x</strong>
    <strong>_______</strong>

(1.) scdspeedtrigger/PID Controller/Filter/Cont. Filter/Filter
          0
(2.) scdspeedtrigger/PID Controller/Integrator/Continuous/Integrator
     11.915
(3.) scdspeedtrigger/Throttle & Manifold/Intake Manifold/p0 = 0.543 bar
    0.49012
(4.) scdspeedtrigger/Vehicle Dynamics/w = T//J w0 = 209 rad//s
     314.16

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

The third operating point is near the 3500 rpm (366.52 rad/s) settling condition.

```
op(3)
```

```
ans =


 Operating point for the Model scdspeedtrigger.
 (Time-Varying Components Evaluated at time t=48.2688)

States:
----------
       <strong>x</strong>
    <strong>_______</strong>

(1.) scdspeedtrigger/PID Controller/Filter/Cont. Filter/Filter
          0
(2.) scdspeedtrigger/PID Controller/Integrator/Continuous/Integrator
     13.349
(3.) scdspeedtrigger/Throttle & Manifold/Intake Manifold/p0 = 0.543 bar
    0.47835
(4.) scdspeedtrigger/Vehicle Dynamics/w = T//J w0 = 209 rad//s
```

```
     366.52

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

For an example that linearizes the speed control model at these operating points, see “Linearize at Triggered Simulation Events” on page 2-75.

```
bdclose(mdl)
```

See Also

Functions
`findop`

Blocks
Trigger-Based Operating Point Snapshot

More About

- “Linearize at Simulation Snapshot” on page 2-72
- “Find Operating Points at Simulation Snapshots” on page 1-83
- “Initialize Steady-State Operating Point Search Using Simulation Snapshot” on page 1-45

Simulate Simulink Model at Specific Operating Point

This example shows how to initialize a model at a specific operating point for simulation. For more information on computing operating points, see “Compute Steady-State Operating Points” on page 1-5 and “Find Operating Points at Simulation Snapshots” on page 1-83

To simulate your model at your computed operating point, you must set the model initial conditions to match the states and inputs in the operating point.

If you already have an operating point, `op`, in your MATLAB or model workspace, you can set the initial conditions in the Configuration Parameters dialog box, in the **Data Import/Export** pane. To do so:

- Set the **Input** parameter to `getinputstruct(op)`.
- Set the **Initial state** parameter to `getstatestruct(op)`.

You can also set the model initial conditions programmatically. For more information, see `getstatestruct` and `getinputstruct`.

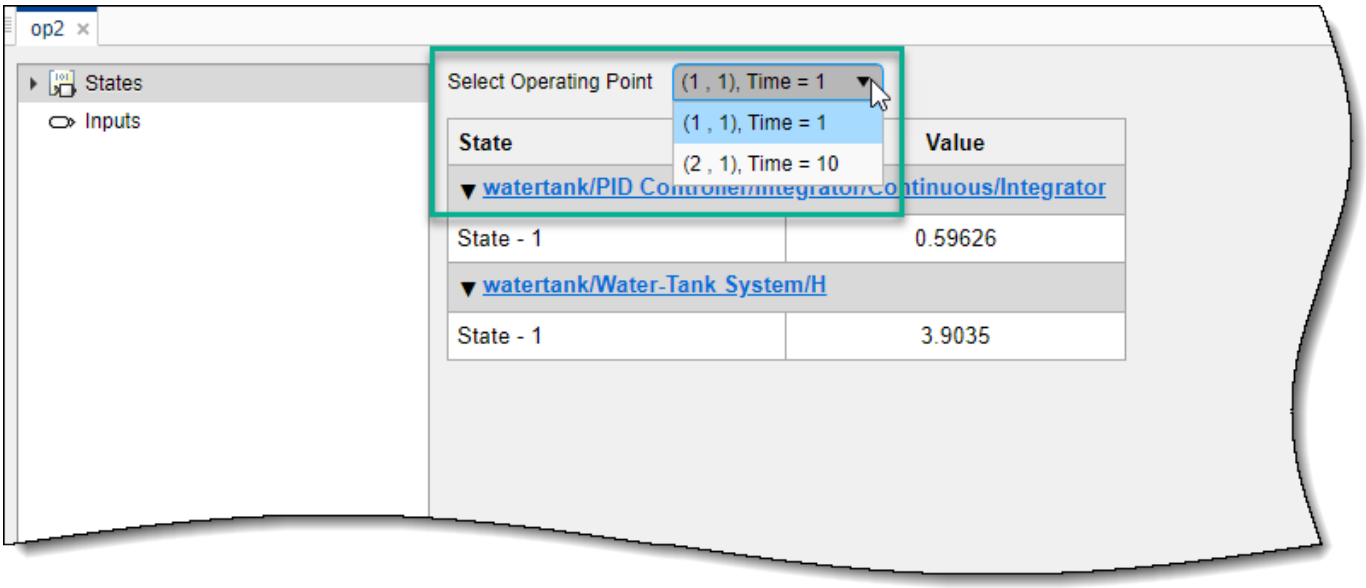
Alternatively, if you computed your operating point using the **Steady State Manager** or **Model Linearizer**, you can interactively set the model initial conditions form within these tools.

Once you have set your model initial condition, simulate your model at the specified operating point.

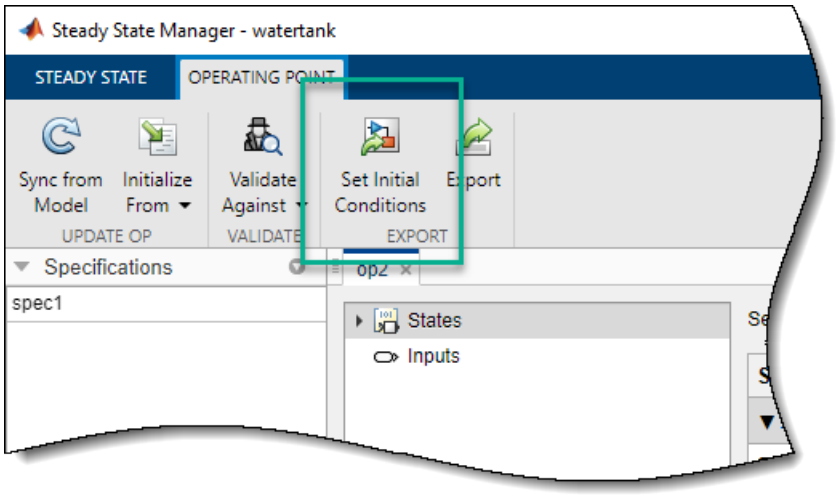
Set Model Operating Point Using Steady State Manager

In the **Steady State Manager**, in the **Data Browser**, in the **Operating Point** section, right-click the operating point at which you want to simulate the model, and select **Open Selection**.

If you computed multiple operating points using a simulation snapshot, in the operating point document, select an operating point from the **Select Operating Point** drop-down list.



On the **Operating Point** tab, click **Set Initial Conditions**.

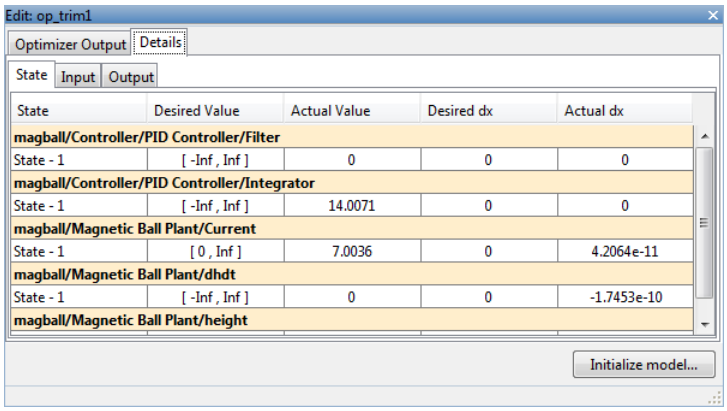


The software sets the initial conditions of the model to match the inputs and states in the selected operating point.

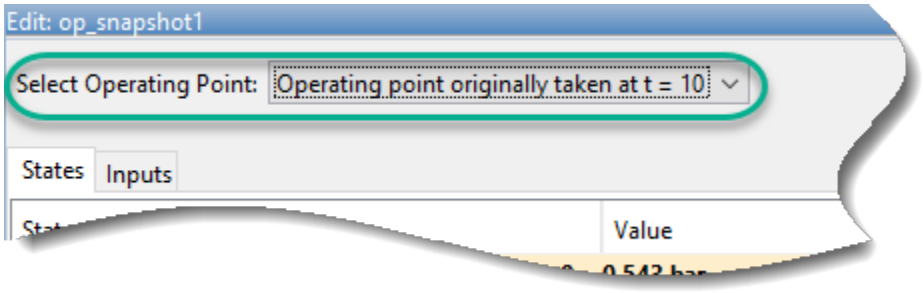
Similarly, using the **Steady State Manager**, you can also set the model initial conditions based on an operating point specification or an operating point search report.

Set Model Operating Point Using Model Linearizer

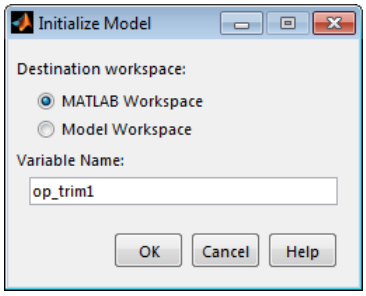
In the **Model Linearizer**, in the **Data Browser**, in the **Linear Analysis Workspace**, double-click the computed operating point or simulation snapshot.



If you computed multiple operating points using a simulation snapshot, select an operating point from the **Select Operating Point** drop-down list.



In the Edit dialog box, click **Initialize model**.



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

To export the operating point to the MATLAB workspace and set the model initial condition to this operating point, click **OK**.

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

See Also

Related Examples

- “Compute Steady-State Operating Points” on page 1-5
- “Find Operating Points at Simulation Snapshots” on page 1-83

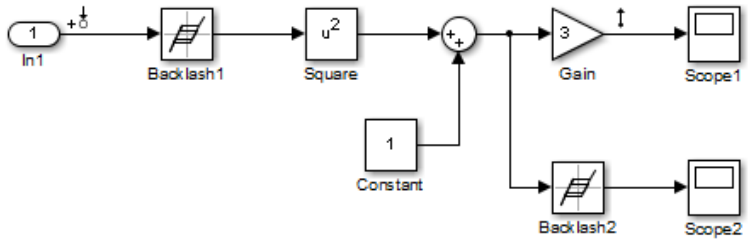
Handle Blocks with Internal State Representation

Operating Point Object Excludes Blocks with Internal States

The operating point object used for linearization and control design does not include Simulink blocks with internal state representation, such as the following:

- Memory blocks
- Transport Delay and Variable Transport Delay blocks
- Disabled If Action Subsystem and Switch Case 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 the following Simulink model, the resulting operating point object does not include the Backlash block states because these states have an internal representation. If you use this operating point object to initialize a Simulink model, the initial conditions of the Backlash blocks might be incompatible with the operating point.



As an example, you can compute an operating point for model `myModel` from a simulation snapshot at 10 seconds and then linearize the model at this operating point. In this case, the `linearize` function initializes the model state with the operating point before linearizing the model.

```
op = findop('myModel',10);
linsys = linearize('myModel',io,op);
```

If `myModel` contains a one or more blocks with an internal state representation, `op` does not contain the internal states. Therefore, `linsys` might not be an accurate linear representation of the model.

Instead of finding an operating point at the simulation snapshot, you can simulate the model to the snapshot time and linearize the model at the snapshot itself.

```
linsys = linearize('myModel',io,10);
```

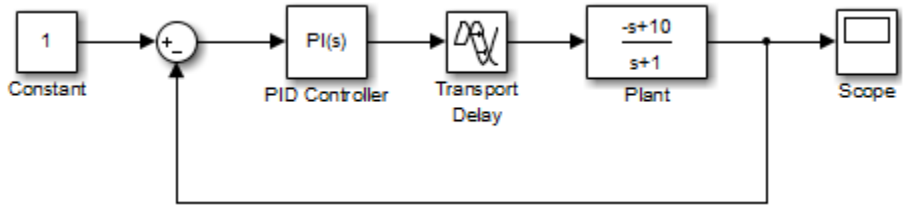
This approach avoids initializing the model with an operating point that is missing state information.

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

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

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

For example, the following 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. Since the reference signal is 1, the input to the Plant block must be nonzero to get the plant block to have an output of 1 and be at steady state.



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

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

Block	Command to Specify Direct Feedthrough
Memory	`set_param(blockname,'LinearizeMemory','on')`
Transport Delay or Variable Transport Delay	`set_param(blockname,'TransDelayFeedthrough','on')`

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

See Also

More About

- “About Operating Points” on page 1-2
- “Compute Steady-State Operating Points” on page 1-5

Synchronize Simulink Model Changes with Operating Point Specifications

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

Synchronize Model Changes Using Steady State Manager

If you change your Simulink model while the **Steady State Manager** is open, you must synchronize the operating point specifications in the **Steady State Manager** to reflect the changes in the model.

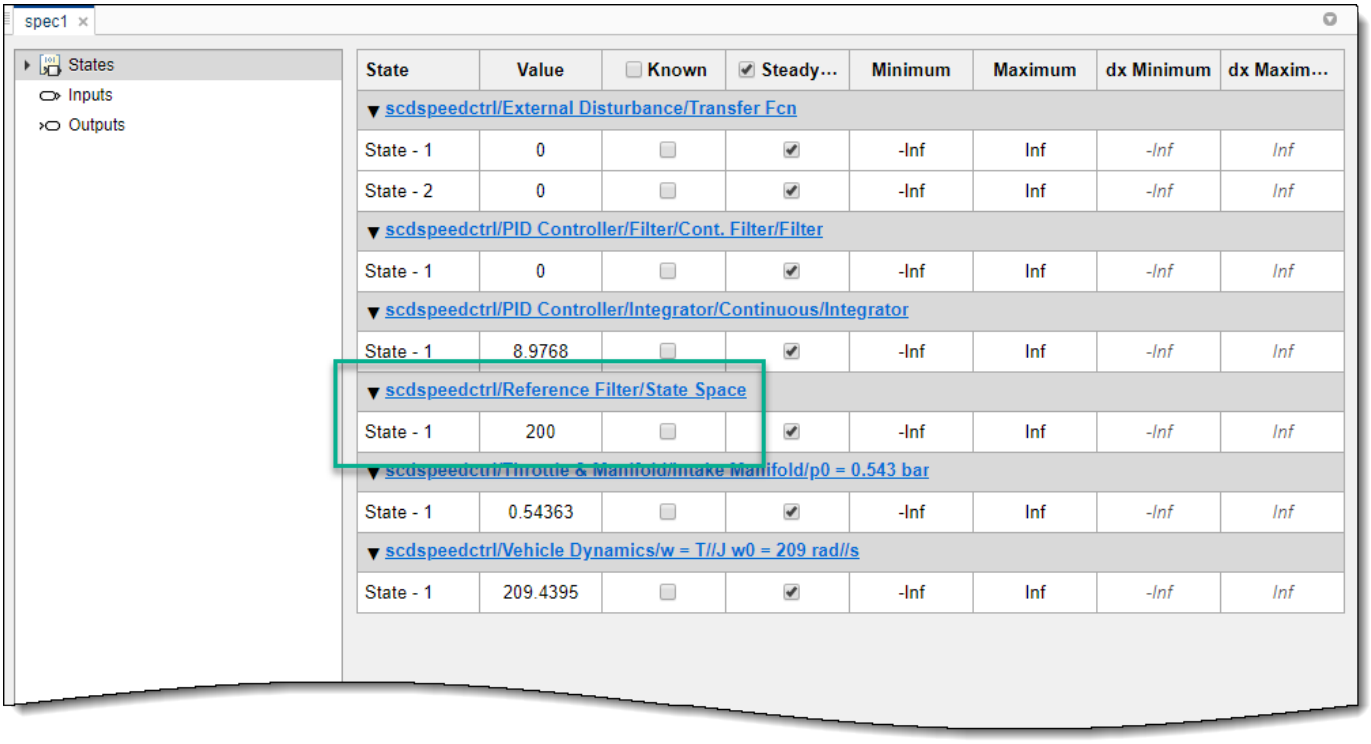
Open the Simulink model.

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

To open the **Steady State Manager**, in the Simulink model window, in the **Apps** gallery, click **Steady State Manager**.

To create an operating specification based on the current model configuration, in the **Steady State Manager**, on the **Steady State** tab, click **Trim Specification**.

In the **spec1** document, the Reference Filter block has one state.

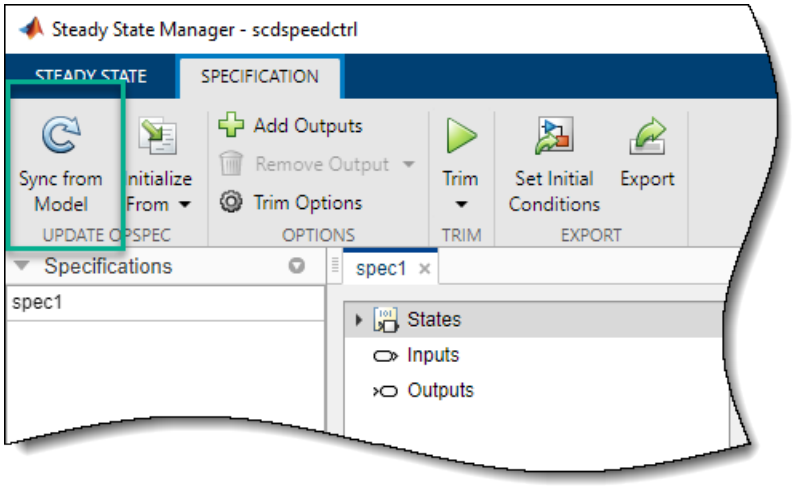


In the Simulink model window, double-click the Reference Filter block. Change the **Numerator** of the transfer function to `100`, and change the **Denominator** to `[1 20 100]`.

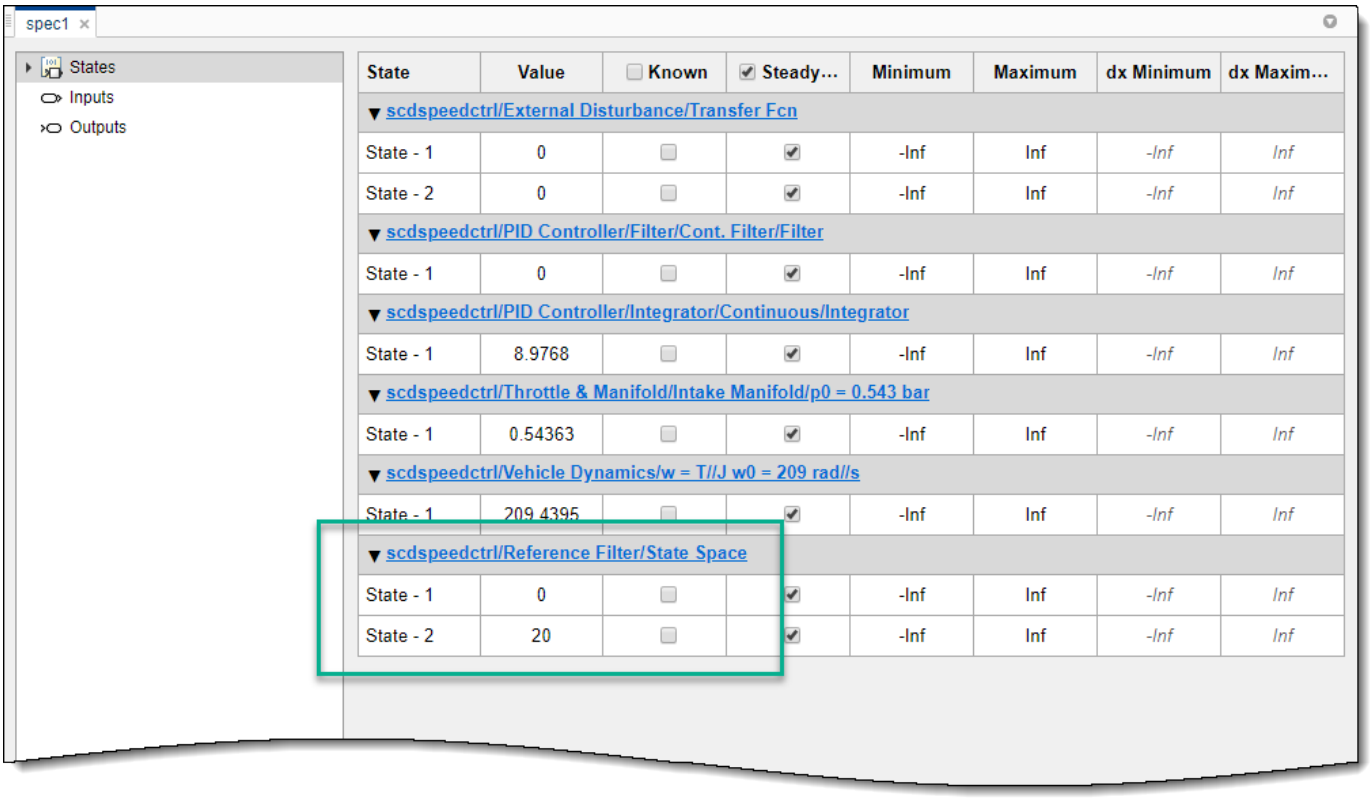
Click **OK**.

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

To update the operating point specifications to reflect the model changes, in the **Steady State Manager**, on the **Specification** tab, click **Sync from Model**.



The software updates the specifications. The Reference Filter block now has two states.



To find the operating point that meets these specifications, on the **Specification** tab, click **Trim** .

Synchronize Model Changes Using Model Linearizer

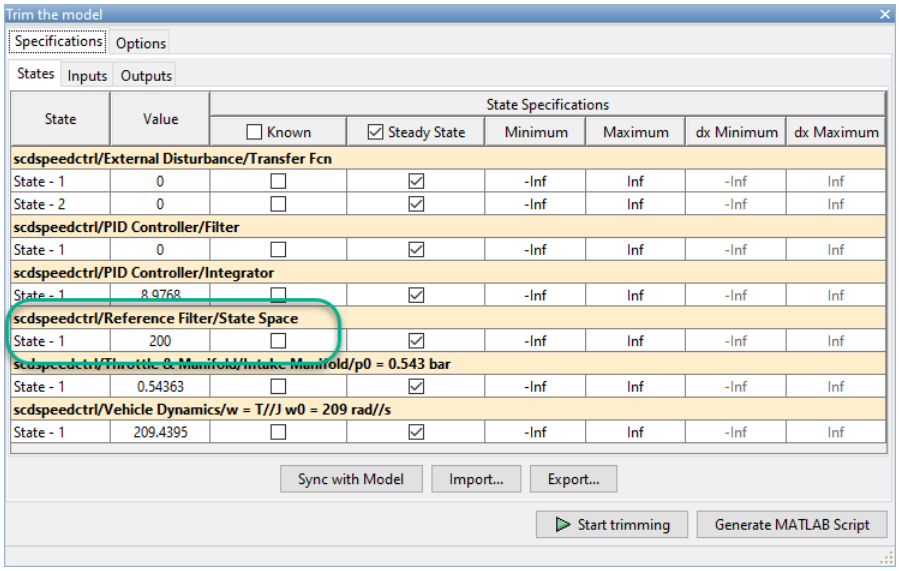
If you change your Simulink model while the **Model Linearizer** is open, you must synchronize the operating point specifications in the **Model Linearizer** to reflect the changes in the model.

Open the Simulink model.

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

To open the **Model Linearizer**, in the Simulink model window, in the **Apps** gallery, click **Model Linearizer**.

In the **Model Linearizer**, in the **Operating Points** drop-down list, select `Trim Model`.



In the Trim the model dialog box, the Reference Filter block contains one state.

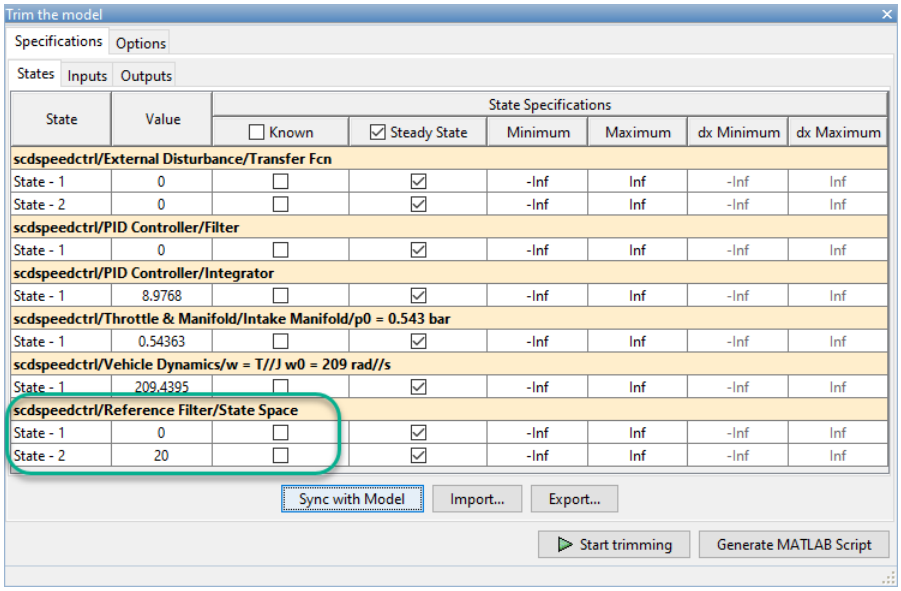
In the Simulink model window, double-click the Reference Filter block. Change the **Numerator** of the transfer function to `100`, and change the **Denominator** to `[1 20 100]`.

Click **OK**.

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

To update the operating point specifications to reflect the model changes, in the Trim the model dialog box, click **Sync with Model**.

The software updates the specifications. The Reference Filter block now has two states.



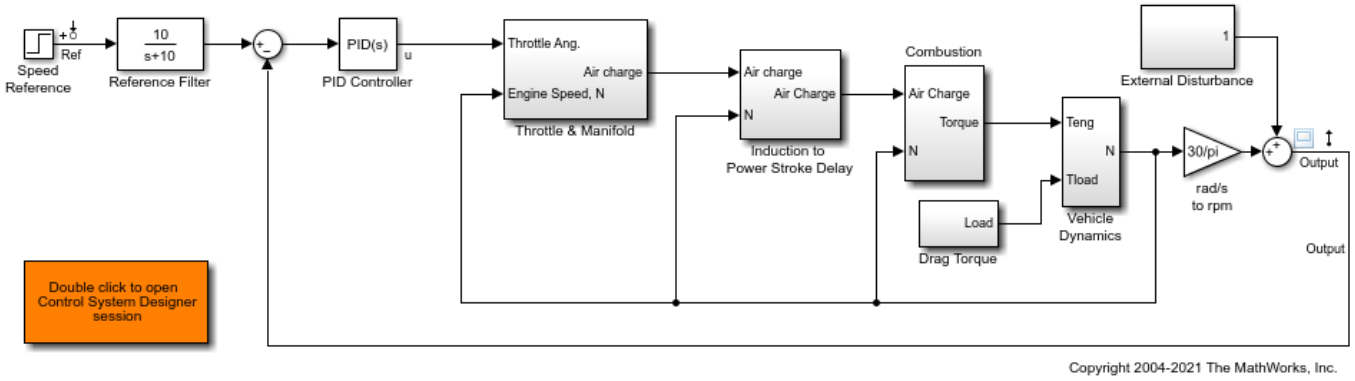
To find the operating point that meets these specifications, click **Start trimming**

Synchronize Model Changes at the Command Line

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

Open the model.

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



Create an operating point specification object based on the current model configuration.

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

```
opspec =
```

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

States:
----------
       <strong>x</strong>       <strong>Known</strong>    <strong>SteadyState</strong>    <strong
    <strong>_______</strong>    <strong>_____</strong>    <strong>___________</strong>    <strong

(1.) scdspeedctrl/External Disturbance/Transfer Fcn
          0    false       true        -Inf    Inf    -Inf      Inf
          0    false       true        -Inf    Inf    -Inf      Inf
(2.) scdspeedctrl/PID Controller/Filter/Cont. Filter/Filter
          0    false       true        -Inf    Inf    -Inf      Inf
(3.) scdspeedctrl/PID Controller/Integrator/Continuous/Integrator
     8.9768    false       true        -Inf    Inf    -Inf      Inf
(4.) scdspeedctrl/Reference Filter/State Space
        200    false       true        -Inf    Inf    -Inf      Inf
(5.) scdspeedctrl/Throttle & Manifold/Intake Manifold/p0 = 0.543 bar
    0.54363    false       true        -Inf    Inf    -Inf      Inf
(6.) scdspeedctrl/Vehicle Dynamics/w = T//J w0 = 209 rad//s
     209.44    false       true        -Inf    Inf    -Inf      Inf

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

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

Change the transfer function of the Reference Filter block. Set the **Numerator** parameter to `100` and the **Denominator** parameter to `[1 20 100]`.

```
set_param('scdspeedctrl/Reference Filter','N',"100");
set_param('scdspeedctrl/Reference Filter','D',"[1 20 100]");
```

Since the model parameters have changed, trying to find an operating point that meets the specifications in `opspec` using the following command generates an error.

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

Update the operating point specification object to reflect the changes in the model.

```
opspec = update(opspec);
```

Find an operating point that meets the updated specifications.

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


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

opreport =


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

Operating point specifications were successfully met.
```

```
States:
----------
    <strong>Min</strong>        <strong>x</strong>       <strong>Max</strong>    <strong>dxMin</s
    <strong>____</strong>    <strong>_______</strong>    <strong>___</strong>    <strong>_____</s

(1.) scdspeedctrl/External Disturbance/Transfer Fcn
    -Inf          0    Inf      0               0      0
    -Inf          0    Inf      0               0      0
(2.) scdspeedctrl/PID Controller/Filter/Cont. Filter/Filter
    -Inf          0    Inf      0               0      0
(3.) scdspeedctrl/PID Controller/Integrator/Continuous/Integrator
    -Inf     8.9768    Inf      0      -4.5077e-14      0
(4.) scdspeedctrl/Throttle & Manifold/Intake Manifold/p0 = 0.543 bar
    -Inf    0.54363    Inf      0       2.9365e-15      0
(5.) scdspeedctrl/Vehicle Dynamics/w = T//J w0 = 209 rad//s
    -Inf     209.44    Inf      0      -1.5226e-13      0
(6.) scdspeedctrl/Reference Filter/State Space
    -Inf          0    Inf      0               0      0
    -Inf         20    Inf      0               0      0

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

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

After you update the operating point specification object, the optimization algorithm successfully finds the operating point.

See Also

`update`

More About

- “Simulate Simulink Model at Specific Operating Point” on page 1-93

Find Steady-State Operating Points for Simscape Models

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

- **Optimization-based trimming** — Specify constraints on model inputs, outputs, or states, and compute a steady-state operating point that satisfies these constraints. For more information, “Compute Steady-State Operating Points” on page 1-5.

  By default, you can define operating point specifications for any Simulink and Simscape states in your model, and any root-level input and output ports of your model. You can also define additional output specifications on Simulink signals. To apply output specifications to a Simscape physical signal, first convert the signal using a PS-Simulink Converter block.
- **Simulation snapshot** — Specify model initial conditions near an expected equilibrium point, and simulate the model until it reaches steady state. You can then create an operating point based on the steady-state signals and states in the model. For more information, see “Find Operating Points at Simulation Snapshots” on page 1-83.

Projection-Based Trim Optimizers

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

You can use these projection-based optimizers when trimming models from the command line and in the **Model Linearizer**.

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

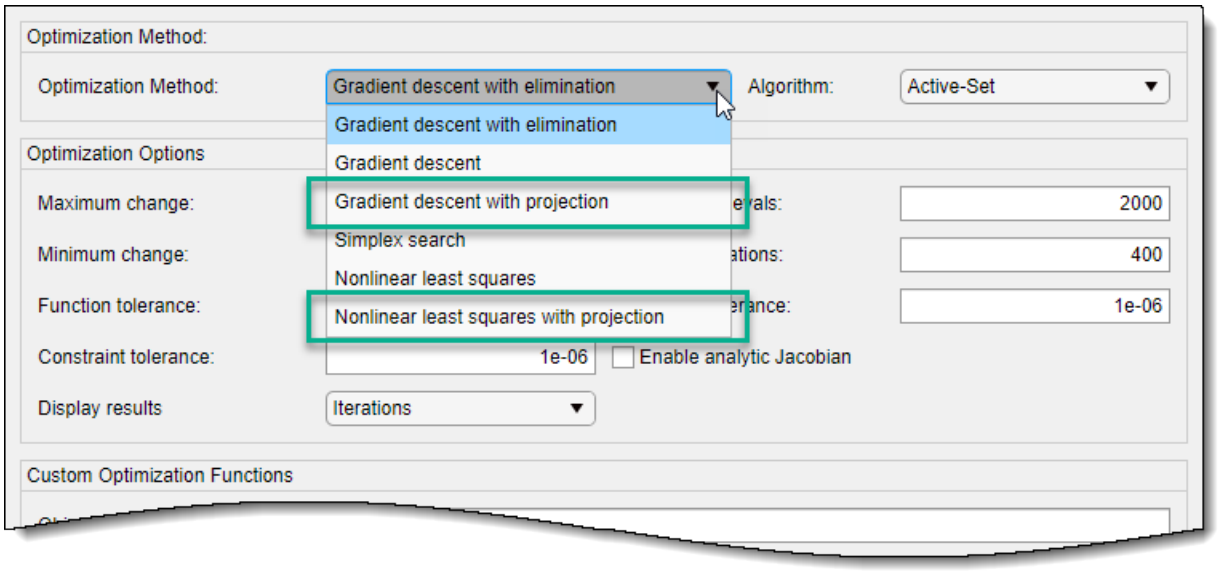
- `'lsqnonlin-proj'` — Nonlinear least squares with projection
- `'graddescent-proj'` — Gradient descent with projection

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

To specify the optimizer type in the:

- **Steady State Manager**, open the Trim Options dialog box. On the **Specification** tab, click **Trim Options**.
- **Model Linearizer**, first open the Trim the model dialog box. On the **Linear Analysis** tab, in the **Operating Point** drop-down list, select `Trim Model`. Then, select the **Options** tab.

In the **Optimization Method** drop-down list, select an optimizer.



When you use gradient descent with projection in **Steady State Manager** or **Model Linearizer**, the algorithm enforces the model initial conditions using hard constraints.

For an example that uses projection-based trim optimization, see “Steady-State Simulation with Projection-Based Trim Optimizer” on page 1-106.

See Also

Apps
Steady State Manager | **Model Linearizer**

Functions
`findop` | `findopOptions` | `operspec`

Blocks
PS-Simulink Converter

Steady-State Simulation with Projection-Based Trim Optimizer

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

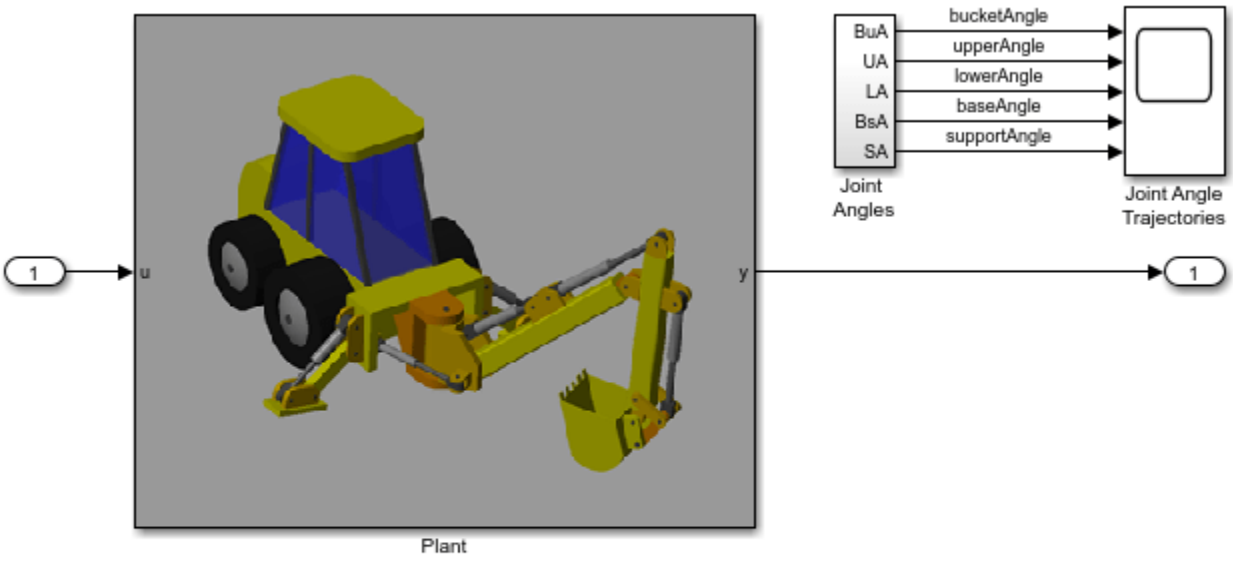
Projection-based optimizers enforce the consistency of the model initial conditions at each evaluation of the objective function or nonlinear constraint function, which can improve trimming results for Simscape models. Using projection-based trim optimizers requires Optimization Toolbox™ software.

Open Model

The model for this example is a backhoe system modeled in Simscape Multibody.

Open the Simulink model.

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



Copyright 2017-2021 The MathWorks, Inc

Define Operating Point Specifications

To define operating point specifications, first create a specification object. The input, output, and state values in `ops` match the model initial conditions.

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

Specify that the model outputs are known values for trimming.

```
opspec.Outputs(1).Known = true(10,1);
```

Specify known values for the angles in the backhoe system.

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

For the corresponding angular velocities, the known values are zero, which match the model initial conditions in `opspec`.

Trim Model

Create an option set for trimming and specify the optimizer type using the `OptimizerType` option. For this example use the projection-based gradient-descent solver. To view an iterative update of the trimming progress in the Command Window, set the `DisplayReport` option to `'iter'`.

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

Specify the maximum number of function evaluations for optimization.

```
opt.OptimizationOptions.MaxFunEvals = 20000;
```

Find the steady-state operating point that meets the specifications in `opspec`. This operation takes several minutes.

```
op = findop(mdl,opspec,opt);
```

```
Optimizing to solve for all desired dx/dt=0, x(k+1)-x(k)=0, and y=ydes.

(Maximum Error)  Block
 ---------------------------------------------------------
(4.50000e+01) scdbackhoeTRIM/Out1
(3.54436e+00) scdbackhoeTRIM/Out1
(2.29690e-01) scdbackhoeTRIM/Out1
(3.84952e-02) scdbackhoeTRIM/Plant/Mounting Assembly/Mounting Base and Support Arms/Support  Arm
(9.31981e-03) scdbackhoeTRIM/Plant/Mounting Assembly/Mounting Base and Support Arms/Support  Arm
(7.25866e-04) scdbackhoeTRIM/Plant/Mounting Assembly/Mounting Base and Support Arms/Support  Arm
(6.61701e-04) scdbackhoeTRIM/Plant/Mounting Assembly/Mounting Base and Support Arms/Support  Arm
(8.93133e-05) scdbackhoeTRIM/Plant/Mounting Assembly/Mounting Base and Support Arms/Support  Arm
(1.41437e-05) scdbackhoeTRIM/Plant/Mounting Assembly/Mounting Base and Support Arms/Support  Arm
(9.73295e-06) scdbackhoeTRIM/Plant/Cylinder Base to Mounting Plate
(1.02224e-06) scdbackhoeTRIM/Plant/Mounting Assembly/Mounting Base and Support Arms/Support  Arm
(1.04790e-06) scdbackhoeTRIM/Plant/Cylinder Base to Mounting Plate
(1.04790e-06) scdbackhoeTRIM/Plant/Cylinder Base to Mounting Plate

Operating point specifications were successfully met.
```

Simulate Model

Configure the model to use the computed operating point `op` as the model initial condition.

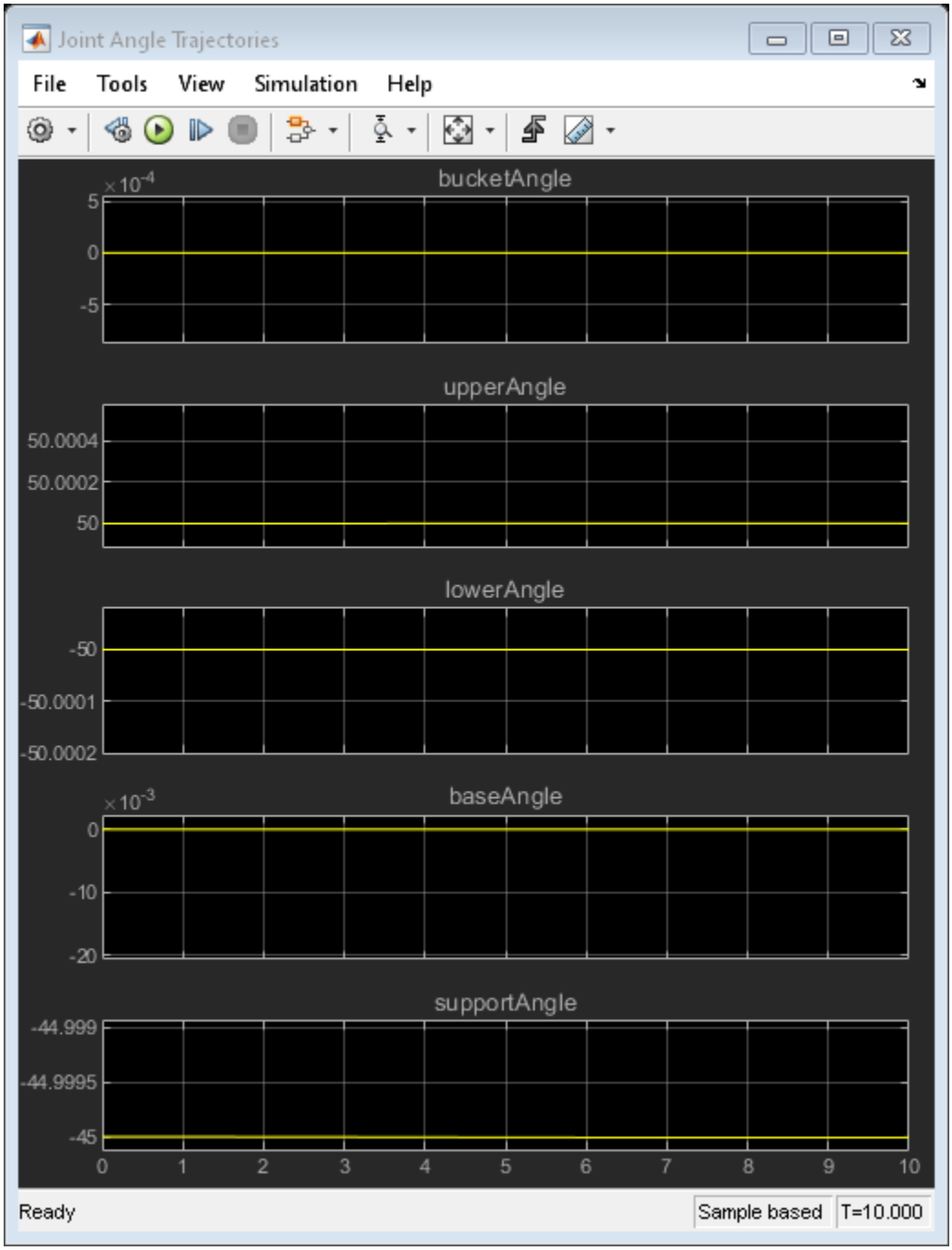
```
set_param(mdl,'LoadExternalInput','on')
set_param(mdl,'ExternalInput','getinputstruct(op)')
set_param(mdl,'LoadInitialState','on')
set_param(mdl,'InitialState','getstatestruct(op)')
```

Simulate the model.

```
sim(mdl);
```

View the joint angle trajectories.

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



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

See Also

Functions
`findop` | `findopOptions` | `operspec`

More About

- “About Operating Points” on page 1-2

Generate MATLAB Code for Operating Point Configuration

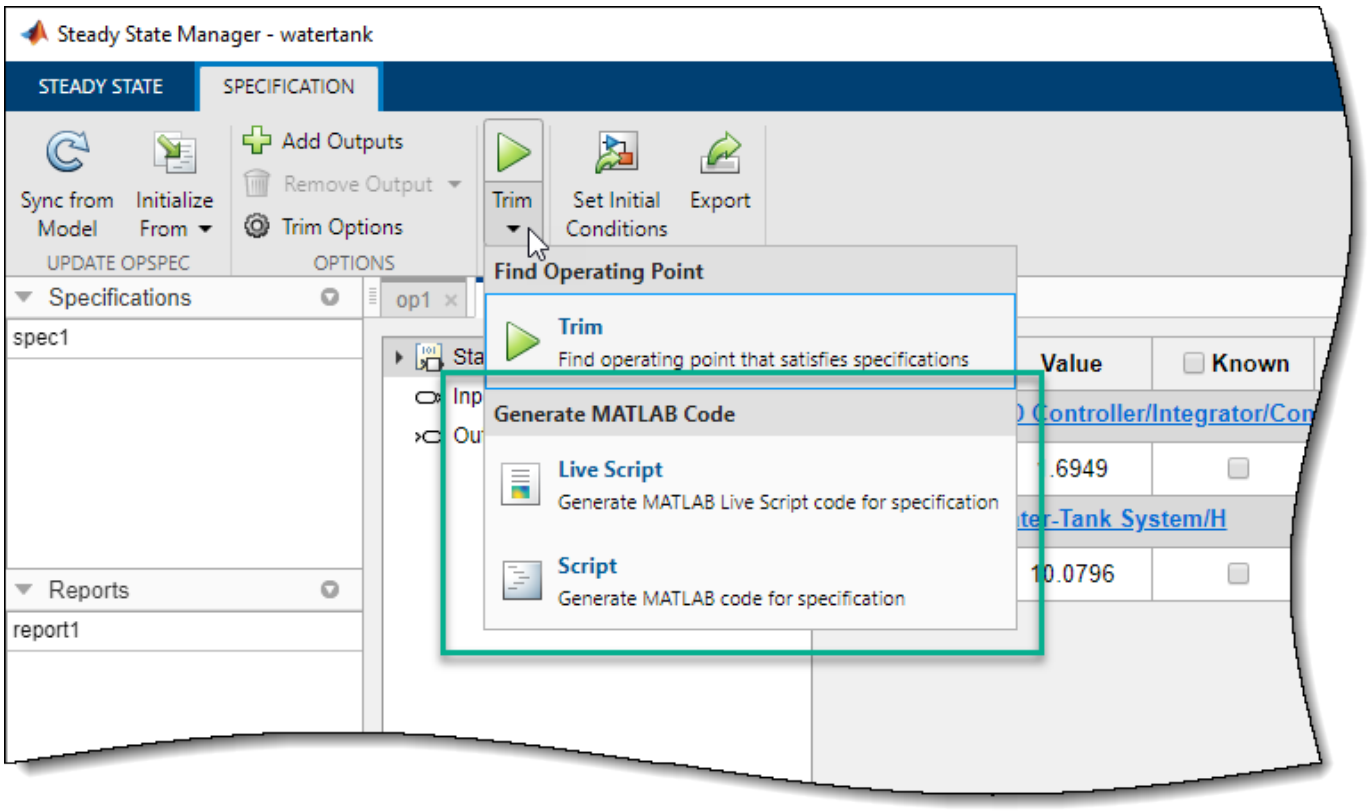
This topic shows how to generate MATLAB code for operating point configuration using the **Steady State Manager** or **Model Linearizer**. You can generate MATLAB code to programmatically reproduce a result that you obtained interactively.

You can also modify the script to compute multiple operating points with systematic variations in operating point specifications (batch computing). For more information, see “Batch Compute Steady-State Operating Points Reusing Generated MATLAB Code” on page 1-80.

Generate MATLAB Code from Steady State Manager

When computing operating points using the **Steady State Manager**, you can generate a MATLAB script or a live script for configuring operating point specifications and computing an operating point. To do so:

1. To create a specification, in the **Steady State Manager**, on the **Steady State** tab, click **Trim Specification**.
2. In the corresponding specification document, configure the operating point state, input, and output specifications. For an example, see “Compute Operating Points from Specifications Using Steady State Manager” on page 1-20.
3. To specify optimization search settings, on the **Specification** tab, click **Trim Options**. For more information, see “Change Operating Point Search Optimization Settings” on page 1-51.
4. To generate code that creates an operating point using your specifications and search options, click **Trim**, and select a code generation option.



You can generate one of the following scripts:

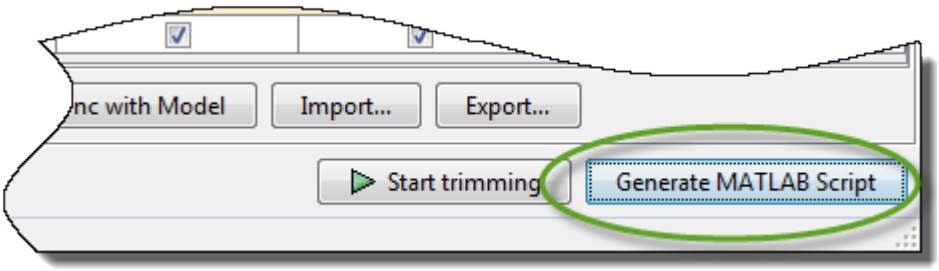
- Live script — Click `Live Script`.
- MATLAB script — Click `Script`.

The generated script opens in the MATLAB Editor.

Generate MATLAB Code from Model Linearizer

When computing operating points using the **Model Linearizer**, you can generate a MATLAB script for configuring operating point specifications and computing an operating point. To do so:

1. In the **Model Linearizer**, on the **Linear Analysis** tab, in the **Operating Points** drop-down list, click `Trim Model`.
2. In the Trim the model dialog box, on the **Specifications** tab, configure the operating point state, input, and output specifications. For an example, see “Compute Operating Points from Specifications Using Model Linearizer” on page 1-31.
3. In the **Options** tab, specify search optimization settings. For more information, see “Change Operating Point Search Optimization Settings” on page 1-51.
4. To generate code that creates an operating point using your specifications and search options, click **Generate MATLAB Script**.



The generated code opens in the MATLAB Editor.

See Also

Functions
`findop` | `operspec`

Apps
Steady State Manager | **Model Linearizer**

More About

- “Compute Steady-State Operating Points” on page 1-5
- “Compute Steady-State Operating Points” on page 1-5
- “Batch Compute Steady-State Operating Points Reusing Generated MATLAB Code” on page 1-80

Linearization

- “Linearize Nonlinear Models” on page 2-3
- “Choose Linearization Tools” on page 2-7
- “Specify Portion of Model to Linearize” on page 2-10
- “Specify Portion of Model to Linearize in Simulink Model” on page 2-17
- “Specify Portion of Model to Linearize in Model Linearizer” on page 2-22
- “Specify Portion of Model to Linearize at Command Line” on page 2-29
- “How the Software Treats Loop Openings” on page 2-31
- “Linearize Plant” on page 2-33
- “Mark Signals of Interest for Control System Analysis and Design” on page 2-38
- “Compute Open-Loop Response” on page 2-46
- “Linearize Simulink Model at Model Operating Point” on page 2-54
- “Visualize Bode Response of Simulink Model During Simulation” on page 2-61
- “Linearize at Trimmed Operating Point” on page 2-67
- “Linearize at Simulation Snapshot” on page 2-72
- “Linearize at Triggered Simulation Events” on page 2-75
- “Linearize Models with Delays” on page 2-78
- “Linearize Models with Model References” on page 2-82
- “Visualize Linear System at Multiple Simulation Snapshots” on page 2-85
- “Visualize Linear System of a Continuous-Time Model Discretized During Simulation” on page 2-91
- “Plot Linear System Characteristics of a Chemical Reactor” on page 2-95
- “Order States in Linearized Model” on page 2-102
- “Validate Linearization In Time Domain” on page 2-107
- “Validate Linearization In Frequency Domain Using Model Linearizer” on page 2-108
- “View Linearized Model Equations Using Model Linearizer” on page 2-111
- “Analyze Results Using Model Linearizer Response Plots” on page 2-113
- “Generate MATLAB Code for Linearization from Model Linearizer” on page 2-120
- “When to Specify Individual Block Linearization” on page 2-122
- “Specify Linear System for Block Linearization Using MATLAB Expression” on page 2-123
- “Specify D-Matrix System for Block Linearization Using Function” on page 2-124
- “Specify Custom Linearizations for Simulink Blocks” on page 2-128
- “Augment Block Linearization” on page 2-133
- “Models with Time Delays” on page 2-137
- “Linearize Multirate Models” on page 2-139
- “Linearize Models Using Different Rate Conversion Methods” on page 2-145

- “Change Perturbation Level of Blocks Perturbed During Linearization” on page 2-148
- “Linearize Blocks with Non-Floating-Point Signals or States” on page 2-150
- “Linearize Event-Based Subsystems (Externally Scheduled Subsystems)” on page 2-152
- “Configure Models with Pulse Width Modulation Signals” on page 2-158
- “Linearize Simscape Networks” on page 2-160
- “Linearize Sparse Models” on page 2-164
- “Specify Linearization for Model Components Using System Identification” on page 2-168
- “Exact Linearization Algorithm” on page 2-175
- “Trim and Linearize an Airframe” on page 2-181
- “Linearize Pneumatic System at Simulation Snapshots” on page 2-186
- “Linearize Pulp Paper Process Model” on page 2-190

Linearize Nonlinear Models

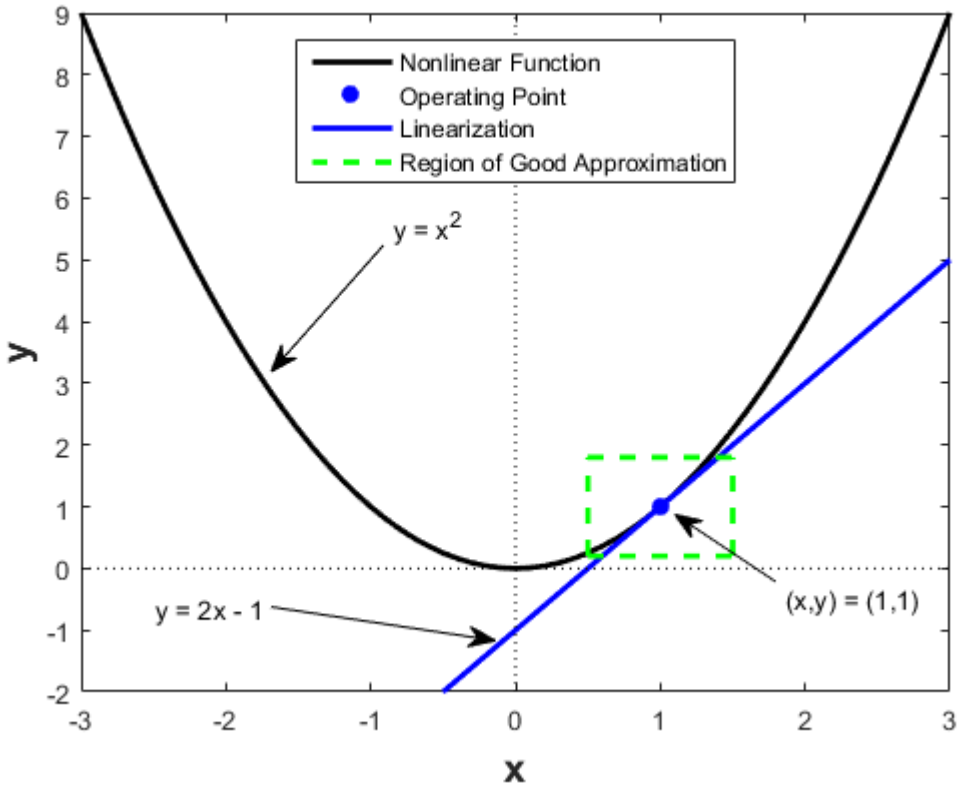
What Is Linearization?

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

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

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

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



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

$$\dot{x}(t) = f(x(t), u(t), t)$$
$$y(t) = g(x(t), u(t), t)\,.$$

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

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

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

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

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

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

Applications of Linearization

Linearization is useful in model analysis and control design applications.

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

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

  Classical control system analysis and design methodologies require linear, time-invariant models. Simulink Control Design automatically linearizes the plant when you tune your compensator. See “Choose a Control Design Approach” on page 9-2.
- Analyze closed-loop stability.
- Measure the size of resonances in frequency response by computing closed-loop linear model for control system.
- Generate controllers with reduced sensitivity to parameter variations and modeling errors.

Linearization in Simulink Control Design

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

By default, Simulink Control Design linearizes models using a *block-by-block* approach. 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.

You can also linearize your system using full-model numerical perturbation, where the software computes the linearization of the full model by perturbing the values of the root-level inputs and states. For each input and state, the software perturbs the model by a small amount and computes a linear model based on the model response to these perturbations. You can perturb the model using either forward differences or central differences.

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

- Most Simulink blocks have a preprogrammed linearization that provides an exact linearization of the block.
- You can use linear analysis points to specify a portion of the model to linearize.
- You can configure blocks to use custom linearizations without affecting your model simulation.
- Structurally nonminimal states are automatically removed.
- You can specify linearizations that include uncertainty (requires Robust Control Toolbox™ software).
- You can obtain detailed diagnostic information.
- When linearizing multirate models, you can use different rate conversion methods. Full-model numerical perturbation can only use zero-order-hold rate conversion.

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. Models that include event-based or discontinuous behavior require special handling by Simulink Control Design software. Such event-based or discontinuous behavior can come from blocks such as:

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

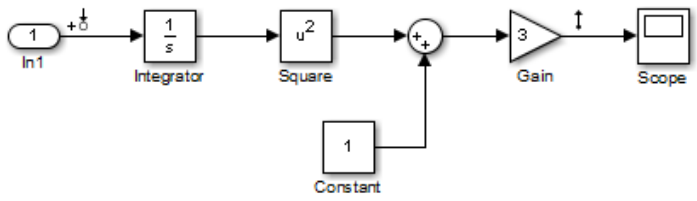
For most applications, the states in your Simulink model should be at steady state. Otherwise, your linear model is only valid over a small time interval.

Operating Point Impact on Linearization

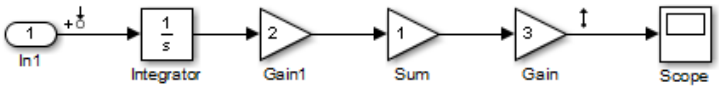
Choosing the right operating point for linearization is critical for obtaining an accurate linear model. The linear model is an approximation of the nonlinear model that is valid only near the operating point at which you linearize the model.

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

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



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



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

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

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

- Trimmed operating point — “Linearize at Trimmed Operating Point” on page 2-67
- Simulation snapshot — “Linearize at Simulation Snapshot” on page 2-72
- Triggered simulation event — “Linearize at Triggered Simulation Events” on page 2-75

See Also

More About

- “Exact Linearization Algorithm” on page 2-175
- “Linearize Plant” on page 2-33
- “Linearize Simulink Model at Model Operating Point” on page 2-54
- “Compute Open-Loop Response” on page 2-46
- “Visualize Bode Response of Simulink Model During Simulation” on page 2-61
- Trimming and Linearization Part 1: What is Linearization?
- Trimming and Linearization Part 2: The Practical Side of Linearization

Choose Linearization Tools

Choosing Simulink Control Design Linearization Tools

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

Linearization Tool	When to Use
Model Linearizer	• Interactively explore Simulink model linearization under different operating conditions. • Diagnose linearization problems. • Batch linearize for varying model parameter values. • Automatically generate MATLAB code for batch linearization.
`linearize`	• Linearize a Simulink model for command-line analysis of poles and zeros, plot responses, and control design. • Batch linearize for varying model parameter values and operating points.
`slLinearizer`	Batch linearize for varying model parameter values, operating points, and I/O sets.
Linear Analysis Plots blocks on page 2-61	• Visualize linear characteristics of your Simulink model during simulation. • View bounds on linear characteristics of your Simulink model on plots. • Optionally, check that the linear characteristics of your Simulink model satisfy specified bounds. **Note** Linear Analysis Plots blocks do not support code generation. You can only use these blocks in `Normal` simulation mode.

Choosing Exact Linearization Versus Frequency Response Estimation

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

Exact linearization:

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

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

Use frequency response estimation:

- To validate exact linearization accuracy. For more information, see “Validate Linearization In Frequency Domain Using Model Linearizer” on page 2-108.
- When your Simulink model contains discontinuities or non-periodic event-based dynamics.
- To study the impact of amplitude size on frequency response. For more information, see “Describing Function Analysis of Nonlinear Simulink Models” on page 5-84.

Linearization Using Simulink Control Design Versus Simulink

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

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

	Simulink Control Design Linearization	**Simulink Linearization**
Graphical-user interface	Yes. See “Linearize Simulink Model at Model Operating Point” on page 2-54.	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 2-67.	No. Only root-level linearization I/O points, which is equivalent to linearizing the entire model. Requires that you add and configure additional Linearization Point blocks.
Open-loop analysis	Yes. Lets you open feedback loops without deleting feedback signals in the model. See “Compute Open-Loop Response” on page 2-46.	Yes, but requires that you delete feedback signals in your model to open the loop
Control linear model state ordering	Yes. See “Order States in Linearized Model” on page 2-102.	No
Control linearization of individual blocks	Yes. Lets you specify custom linearization behavior for both blocks and subsystems. See “When to Specify Individual Block Linearization” on page 2-122.	No
Linearization diagnostics	Yes. Identifies problematic blocks and lets you examine the linearization value of each block. See “Linearization Troubleshooting Overview” on page 4-2.	No
Block detection and reduction	Yes. Block reduction detects blocks that do not contribute to the overall linearization yielding a minimal realization.	No

	Simulink Control Design Linearization	Simulink Linearization
Control of rate conversion algorithm for multirate models	Yes	No

See Also

More About

- “Linearize Nonlinear Models” on page 2-3

Specify Portion of Model to Linearize

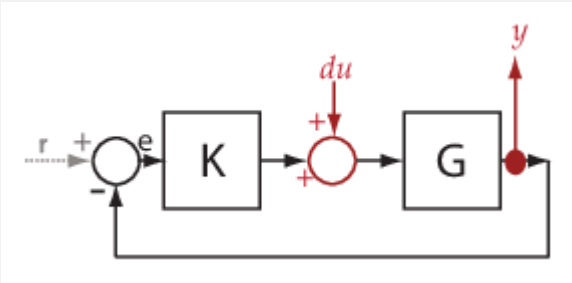
To linearize a subsystem, loop, or block in your model, you use *analysis points*. Each analysis point that you define in the model can serve one or more of the following purposes:

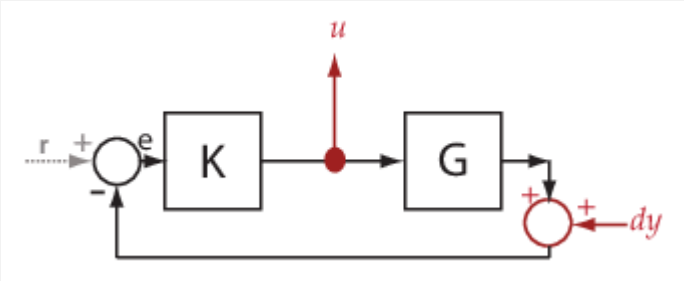
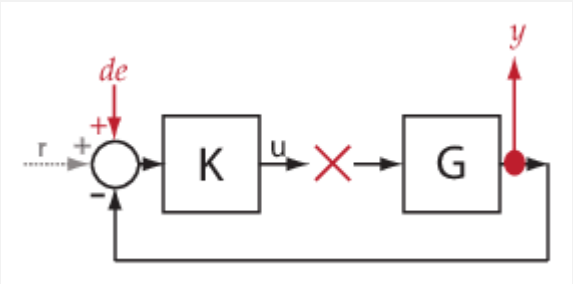
- **Input** — The software injects an additive input signal at an analysis point, for example, to model a disturbance at the plant input.
- **Output** — The software measures the signal value at a point, for example, to study the impact of a disturbance on the plant output.
- **Loop Opening** — The software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

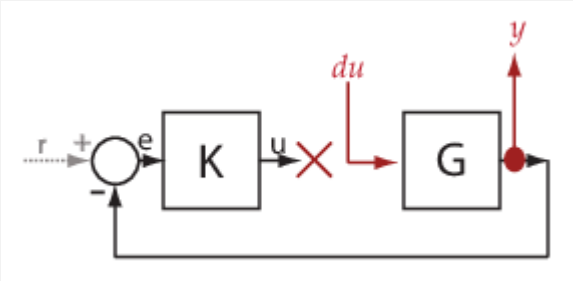
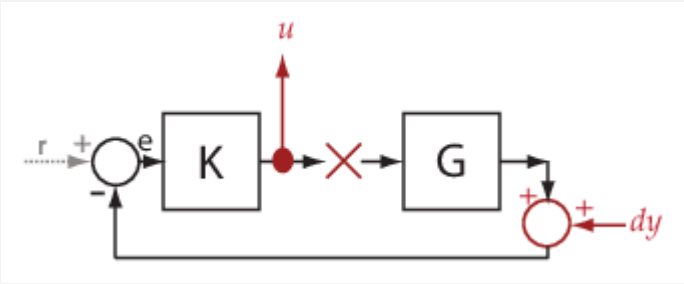
To compute a linear model for a portion of your system, specify a linearization input point and output point on the input and output signal to the portion of the model you want to linearize. To compute an open-loop response, specify loop openings to break the signal flow. You can also compute MIMO linear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

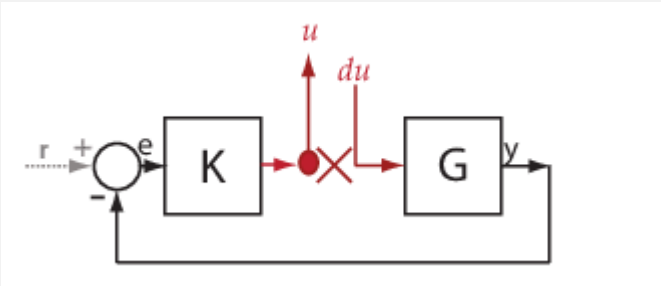
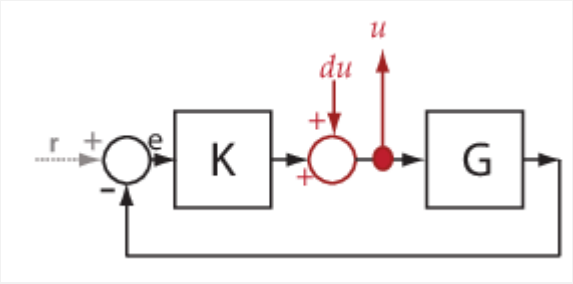
Analysis Points

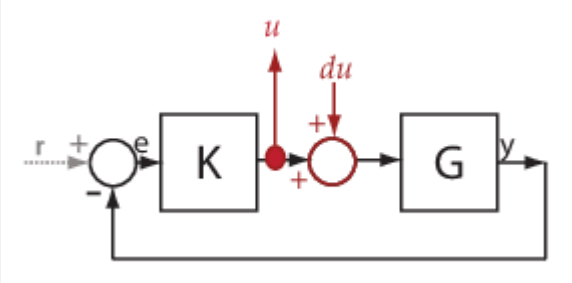
You can specify the following types of linear analysis points using Simulink Control Design software. These analysis points are pure annotations and do not impact model simulation.

Analysis Point	Description
Input perturbation	Specifies an additive input to a signal. To define a transfer function for a linearized system, you can use an input perturbation with an output measurement or open-loop output. For example, to compute the response $G/(1+GK)$ in the example system, specify an input perturbation du and an output measurement y as shown. 

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

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

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

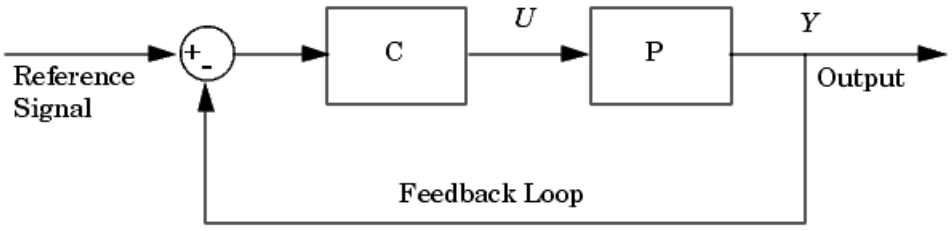
Analysis Point	Description
Complementary sensitivity function	Specifies an output measurement followed by an input perturbation. The complementary sensitivity function at a point is the transfer function from an additive disturbance at the point to a measurement at the same point. In contrast to the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

Opening Feedback Loops

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

To remove the effects of a feedback loop, using analysis points lets you insert a loop opening without manually breaking the signal line. Manually removing the feedback signal from a nonlinear model changes the model operating point and produces a different linearized model. For more information, see “How the Software Treats Loop Openings” on page 2-31.

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

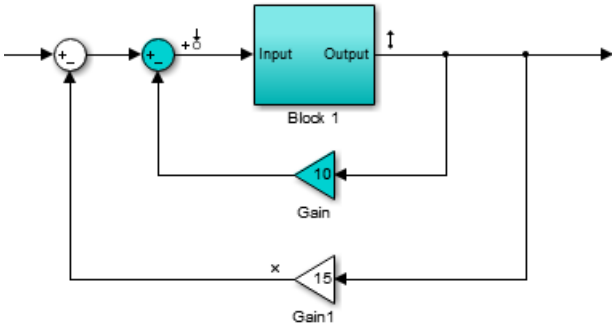


Suppose that you want to linearize the plant *P* about an equilibrium operating point of the model.

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

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

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



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

Ways to Specify Portion of Model to Linearize

There are several ways to define the portion of the model you want to linearize using linear analysis points. Each method has its own advantages and depends on which linearization tool you use. For more information on choosing linearization tools, see “Choose Linearization Tools” on page 2-7.

Specify portion of model...	Use this method if...	For more Information, see...
In Simulink model	You want to save the analysis points directly in the model or graphically display the analysis points within the model.	“Specify Portion of Model to Linearize in Simulink Model” on page 2-17
Using **Model Linearizer**	You want to linearize your model interactively using the **Model Linearizer** without changing the Simulink model. Using this method you can specify multiple open-loop or closed-loop transfer functions for your model.	“Specify Portion of Model to Linearize in Model Linearizer” on page 2-22
At command line using `linio` command	You want to linearize your model using the `linearize` command. Using `linio` does not change the Simulink model.	“Specify Portion of Model to Linearize at Command Line” on page 2-29

Specify portion of model...	Use this method if...	For more Information, see...
Using `slLinearizer` interface	You want to obtain multiple open-loop or closed-loop transfer functions from the linearized system without recompiling the model. Using this method does not change the Simulink model.	“Mark Signals of Interest for Batch Linearization” on page 3-9
Using `slTuner` interface	You want to obtain multiple open-loop or closed-loop transfer functions from a tuned control system without recompiling the model. Using this method does not change the Simulink model.	“Mark Signals of Interest for Control System Analysis and Design” on page 2-38
As a specific block or subsystem	You want to linearize a specific block or subsystem without defining analysis points for all the block inputs and outputs. Using this method does not change the Simulink model.	“Linearize Plant” on page 2-33

See Also

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

More About

- “Choose Linearization Tools” on page 2-7
- “Linearize Simulink Model at Model Operating Point” on page 2-54
- “Compute Open-Loop Response” on page 2-46

Specify Portion of Model to Linearize in Simulink Model

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

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

- In the **Model Linearizer**. For more information, see “Specify Portion of Model to Linearize in Model Linearizer” on page 2-22.
- At the command line. For more information, see “Specify Portion of Model to Linearize at Command Line” on page 2-29.

Specify Analysis Points

To specify analysis points directly in your Simulink model, first open the **Linearization** tab. To do so, in the **Apps** gallery, click **Linearization Manager**.

To specify an analysis point:

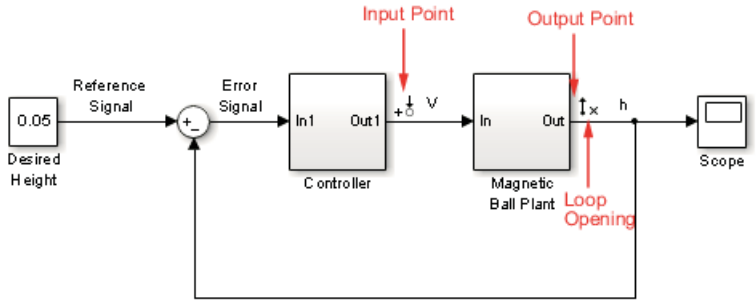
1. In the model, click the signal you want to define as an analysis point.
2. On the **Linearization** tab, in the **Insert Analysis Points** gallery, select the type of analysis point you want to define.

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

   For more information on the different types of analysis points, see “Specify Portion of Model to Linearize” on page 2-10.

   When you specify analysis points, the software adds annotations to your model indicating the linear analysis point type.
3. Repeat steps 1 and 2 for all signals you want to define as analysis points.

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

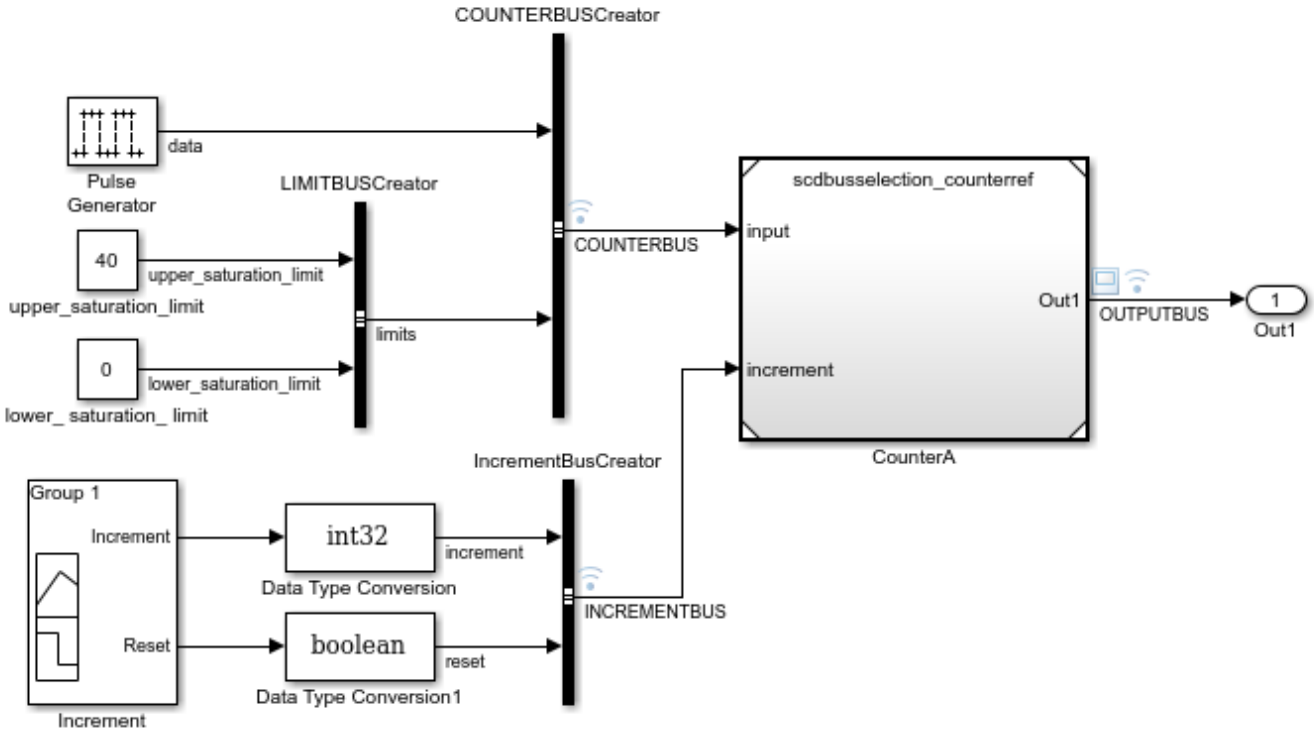


Select Bus Elements as Analysis Points

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

1 Open Simulink model.

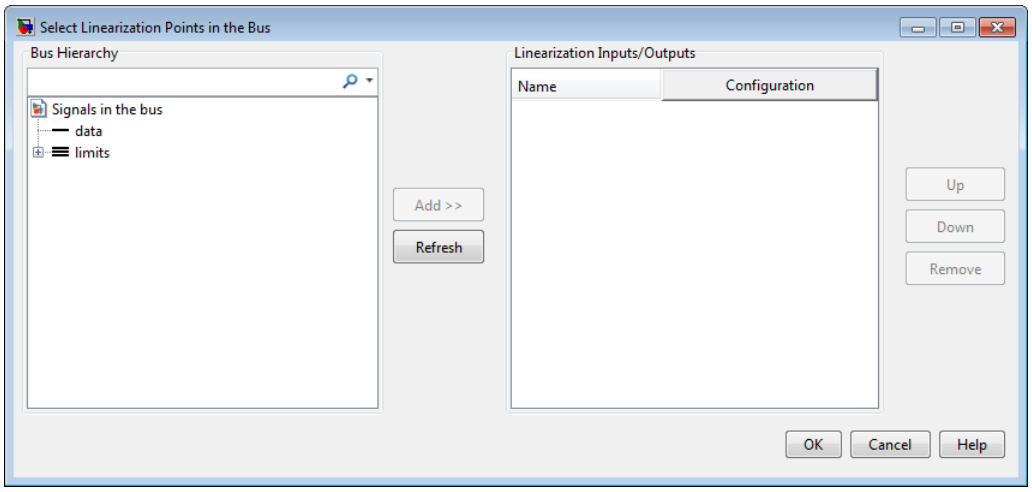
```
sys = 'scdbusselection';
open_system(sys)
```



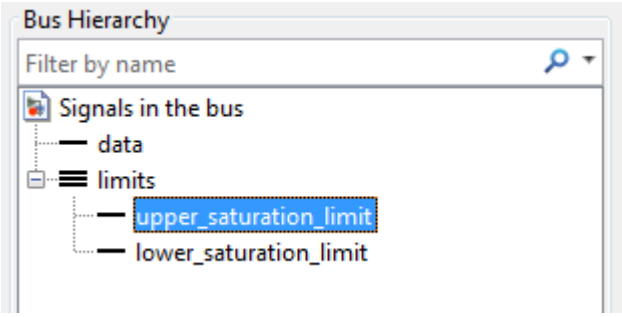
2 Specify a bus signal as a linear analysis point.

First, open the **Linearization** tab. In the Simulink model window, in the **Apps** gallery, click **Linearization Manager**.

In the model, click a bus signal, such as the `OUTPUTBUS` signal. On the **Linearization** tab, click **Select Bus Element**.



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



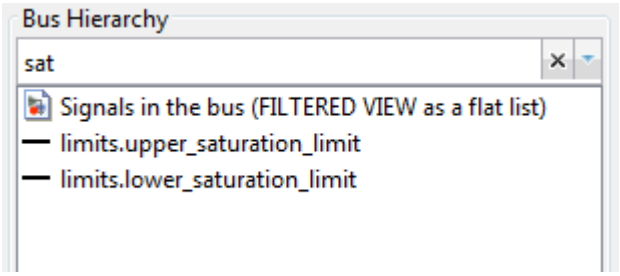
Tip To filter bus elements by name within a large bus, you can enter search text in the **Filter by name** box. The name match is case-sensitive. Also, you can enter a MATLAB regular expression.

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

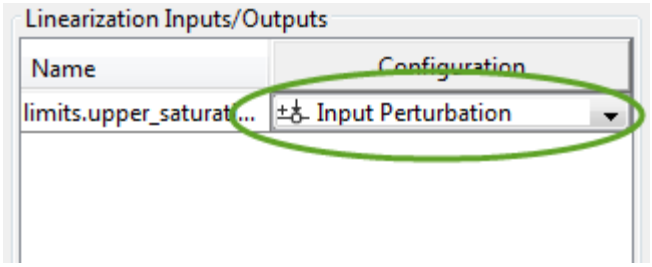
Filtering Options

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

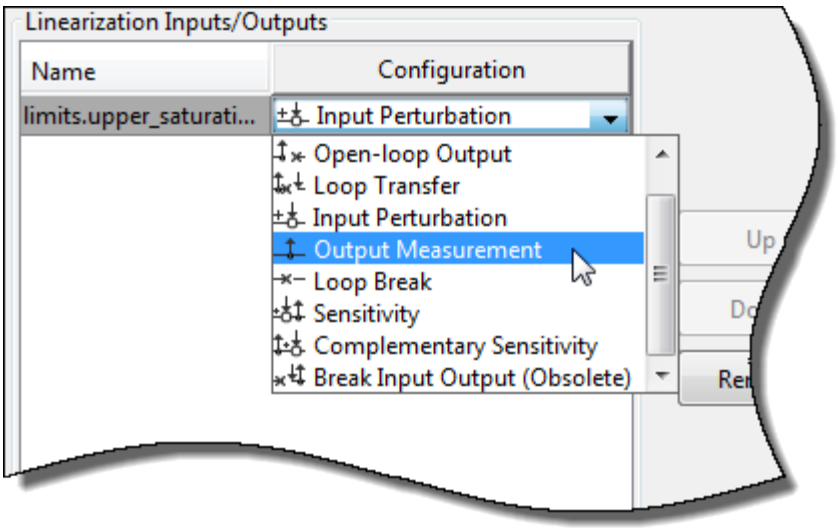
- **Regular expression** — Use MATLAB regular expressions for filtering signal names. For example, entering `t$` displays all signals whose names end with a lowercase `t` (and their immediate parents).
- **Show filtered results as a flat list** — Display the filtered signals in a flat list. By default,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



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



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

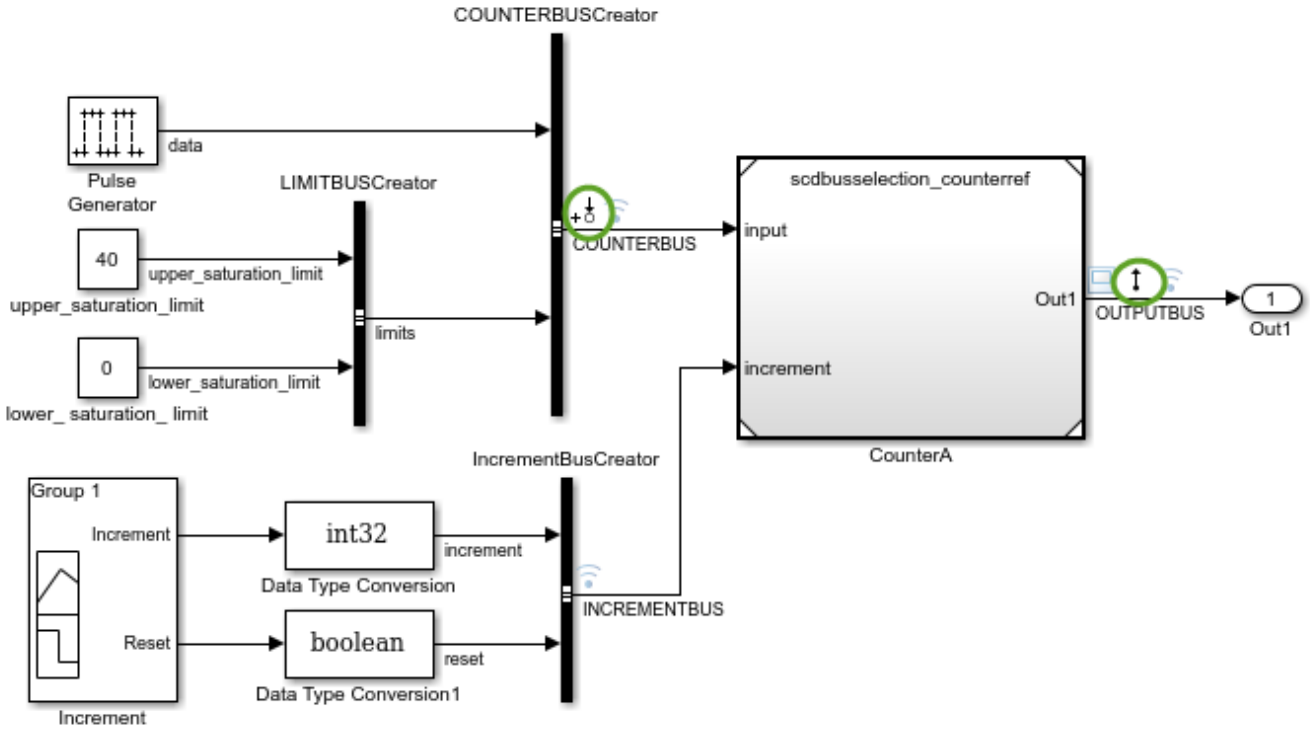


3. To add additional analysis points from within the same bus signal, repeat step 2.

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

   Once you have defined all of the required analysis points for that bus, click **OK**.
4. To specify analysis points for another bus signal, repeat steps 2 and 3.
5. To view linear analysis point indicators in the Simulink model, on the **Linearization** tab, in the **Insert Analysis Points** gallery, select **Linearization Indicators**.

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



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

See Also

More About

- “Specify Portion of Model to Linearize” on page 2-10
- “Linearize Simulink Model at Model Operating Point” on page 2-54
- “Compute Open-Loop Response” on page 2-46

Specify Portion of Model to Linearize in Model Linearizer

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

Alternatively, you can define analysis points:

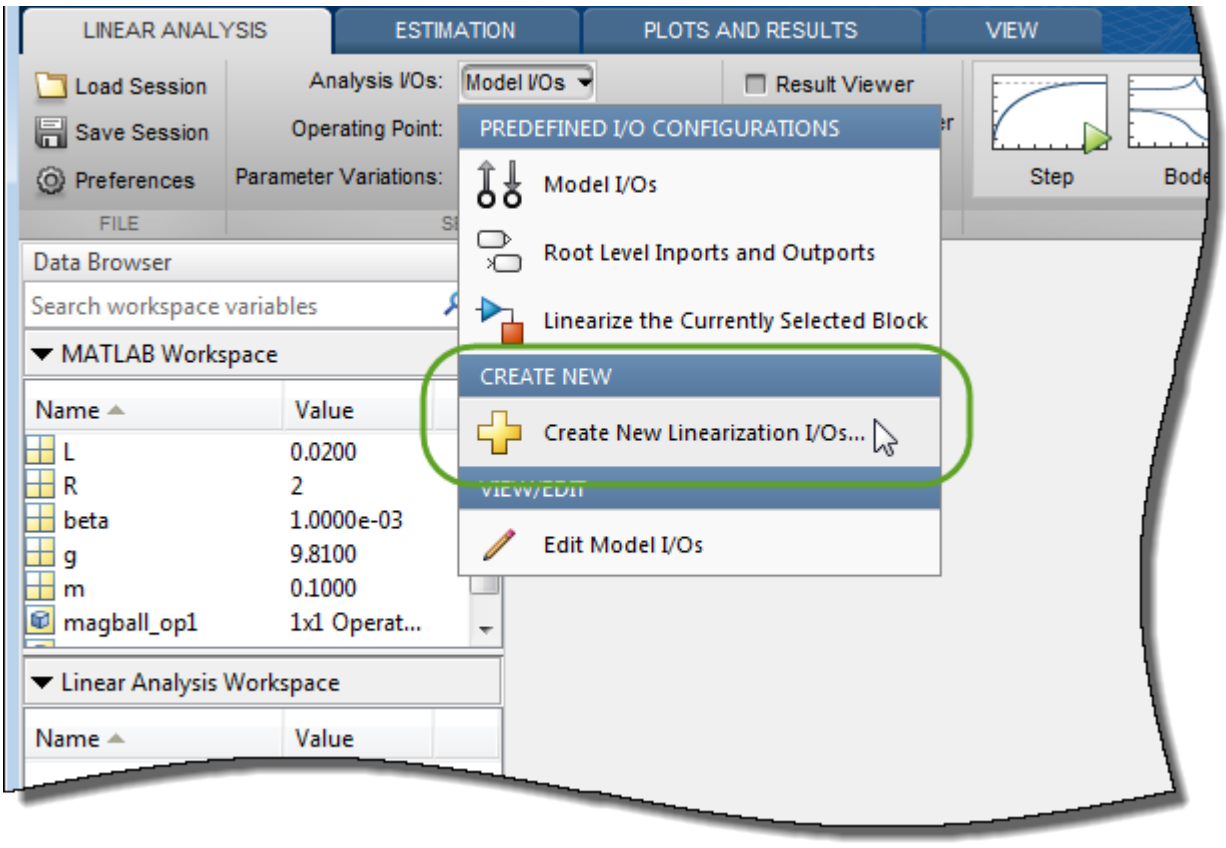
- Programmatically at the command line. For more information, see “Specify Portion of Model to Linearize at Command Line” on page 2-29.
- Directly in your Simulink model. Use this method to save your analysis points in the model. For more information, see “Specify Portion of Model to Linearize in Simulink Model” on page 2-17.

Specify Analysis Points

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

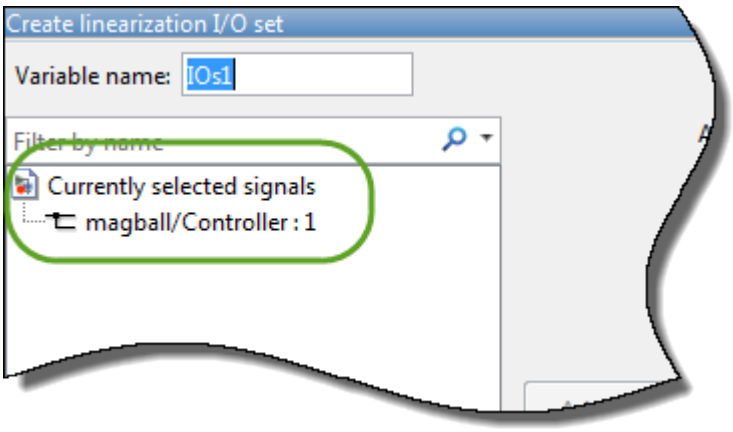
To create a linearization I/O set:

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



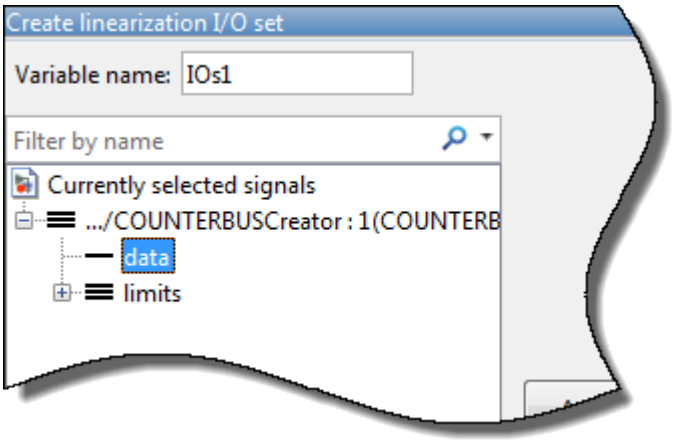
1 In your Simulink model, select one or more signals that you want to define as analysis points.

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

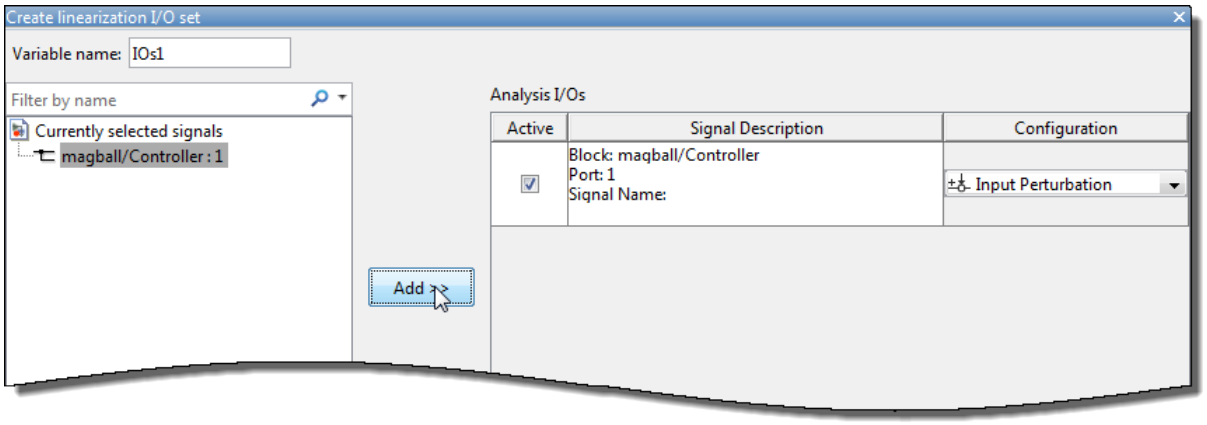


2 Under **Currently selected signals**, click the signal you want to add. To select multiple signals, hold **Ctrl** and click each signal you want to add.

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



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



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

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

For more information on the different types of analysis points, see “Specify Portion of Model to Linearize” on page 2-10.

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

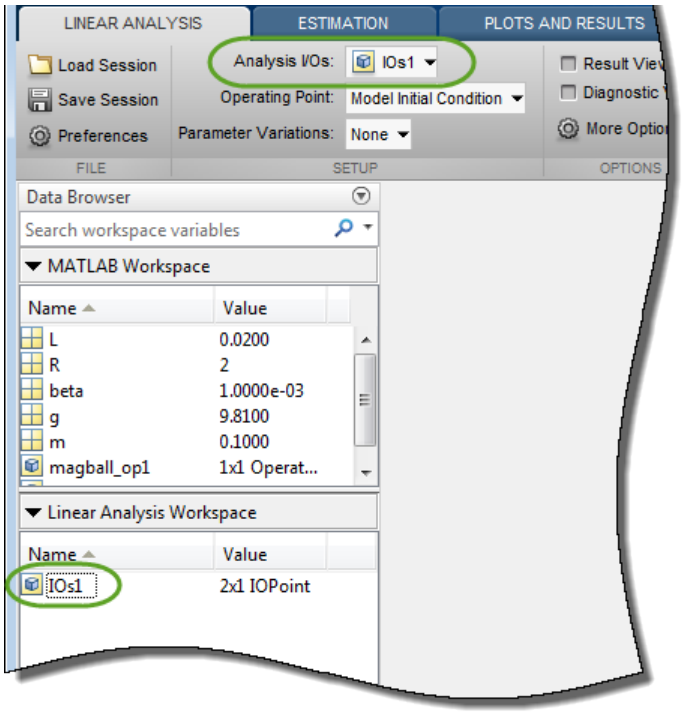
Tip To highlight the source block of an analysis point in the Simulink model, in the **Analysis I/Os** list, select the analysis point, and click **Highlight**.

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

7 Click **OK**.

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

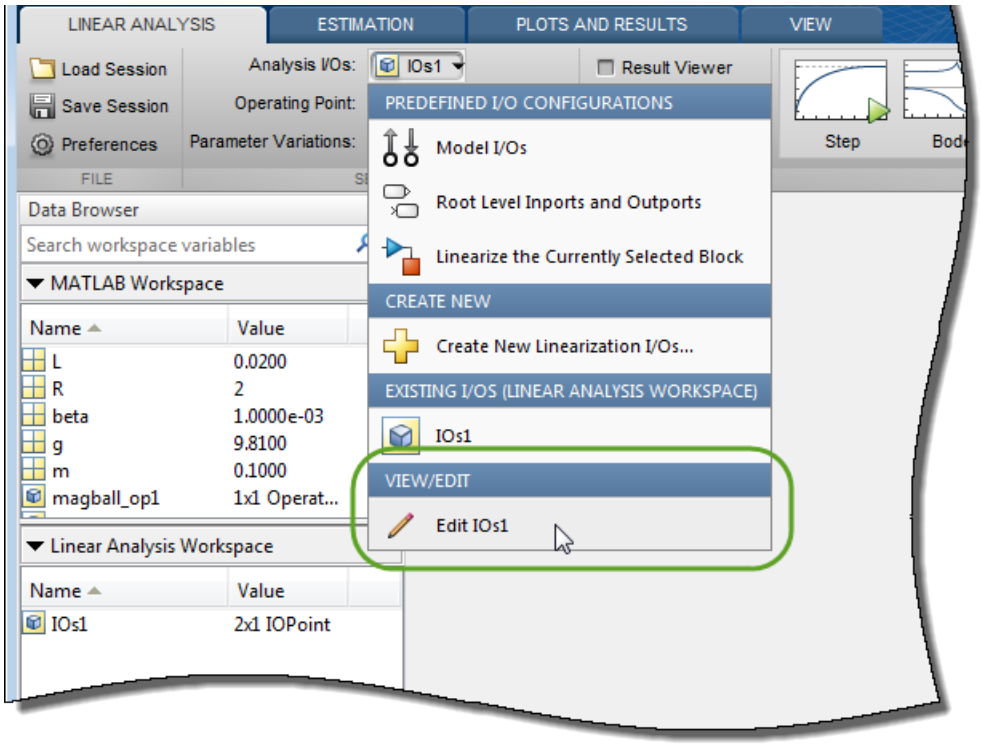
The software also adds the linearization I/O set to the **Analysis I/Os** drop-down list and automatically selects it.



Edit Analysis Points

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

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



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

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

Add Analysis Point to I/O Set

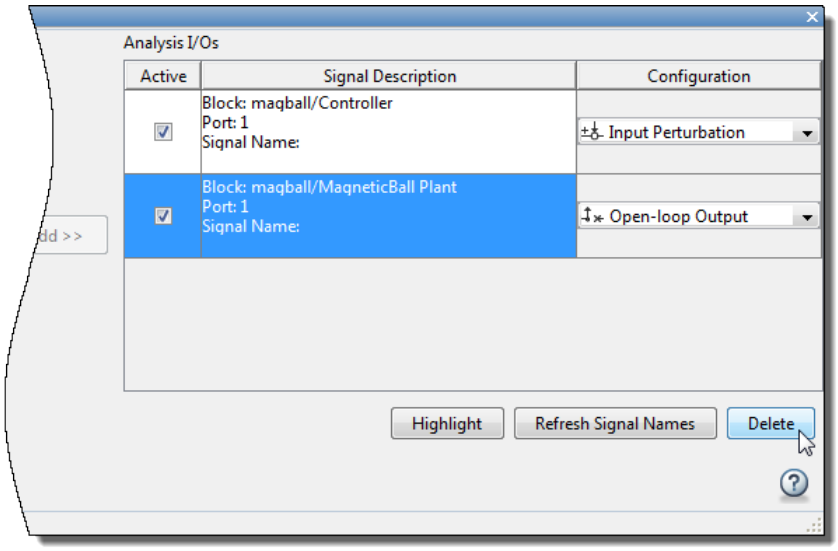
To add an analysis point to the linearization I/O set:

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

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

Remove Analysis Point from I/O Set

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



Change Analysis Point Type

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

Enable or Disable Analysis Points

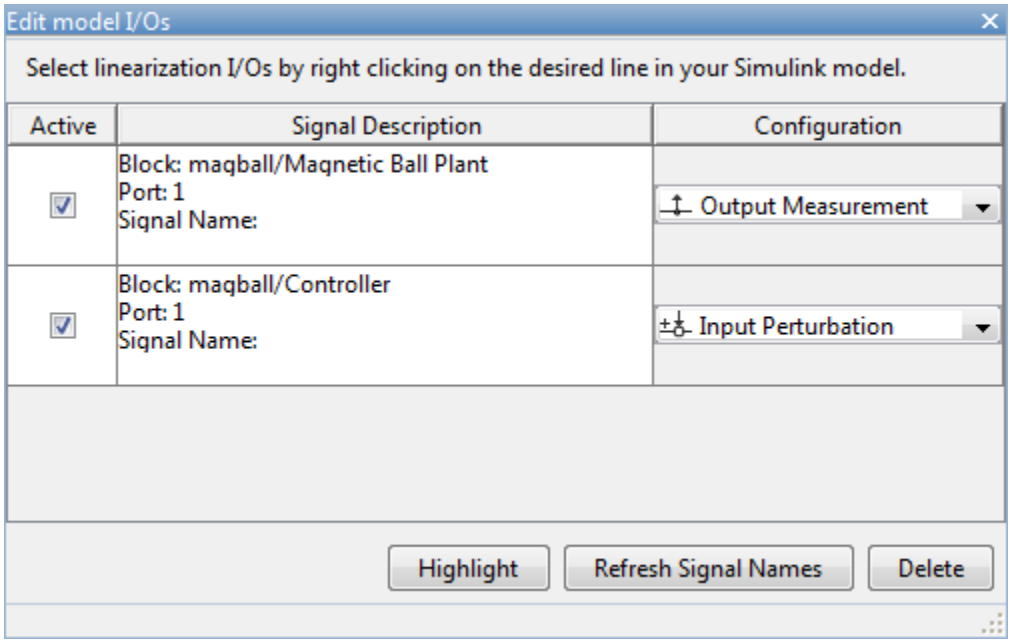
To modify an existing linearization I/O set without removing analysis points, you can disable one or more analysis points. To do so, in the **Analysis I/Os** section, under **Active**, clear the corresponding check box.

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

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

Edit Simulink Model Analysis Points

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



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

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

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

For information on adding analysis points to the model, see “Specify Portion of Model to Linearize in Simulink Model” on page 2-17.

See Also

Model Linearizer

More About

- “Specify Portion of Model to Linearize” on page 2-10
- “Linearize Simulink Model at Model Operating Point” on page 2-54
- “Compute Open-Loop Response” on page 2-46

Specify Portion of Model to Linearize at Command Line

To specify the portion of your Simulink model to linearize, you can define linear analysis points at the command line using the `linio`, `setlinio`, and `getlinio` functions. Analysis points represent linearization inputs, outputs, and loop openings for your model. Using this method, you can specify multiple sets of analysis points without changing your model.

Alternatively, you can define analysis points:

- In the **Model Linearizer**. For more information, see “Specify Portion of Model to Linearize in Model Linearizer” on page 2-22.
- Directly in your Simulink model. Use this method to save your analysis points in the model. For more information, see “Specify Portion of Model to Linearize in Simulink Model” on page 2-17.

Specify Analysis Points

To specify analysis points at the command line, create linearization I/O objects using the `linio` function. To create an analysis point at the output port of a block in your model, use the following syntax:

```
io = linio(block,port,type);
```

where

- `block` is the full block path of the block, specified as a character vector.
- `port` is the output port number.
- `type` is the analysis point type, specified as one of the following:
  - `'input'` — Input perturbation
  - `'output'` — Output measurement
  - `'loopbreak'` — Loop break
  - `'openinput'` — Open-loop input
  - `'openoutput'` — Open-loop output
  - `'looptransfer'` — Loop transfer
  - `'sensitivity'` — Sensitivity
  - `'compsensitivity'` — Complementary sensitivity

For more information on analysis point types, see “Specify Portion of Model to Linearize” on page 2-10.

After creating an analysis point, you can change its type using dot notation. For example, to change an analysis point to be an open-loop output, use:

```
io.Type = 'openoutput';
```

You can also specify analysis points on bus elements in your model. For an example, see `linio`.

To specify multiple analysis points, create a vector of linearization I/O objects. For example, create a set of analysis points that includes an input perturbation, an output measurement, and a loop opening.

```
io(1) = linio(block1,port1,'input');
io(2) = linio(block2,port2,'output');
io(3) = linio(block3,port3,'loopbreak');
```

To linearize your model using the specified analysis points, use the `linearize` function.

Save Analysis Points in Simulink Model

You can save your specified analysis points in your Simulink model using the `setlinio` function.

```
setlinio(mdl,io);
```

Here, `mdl` is a character vector specifying the name of a model in the current working folder or on the MATLAB path, and `io` is a vector of linearization I/O objects.

The analysis points in `io` overwrite any existing analysis points saved in the model.

Alternatively, you can specify analysis points directly in your model. For more information, see “Specify Portion of Model to Linearize in Simulink Model” on page 2-17.

Obtain Analysis Points from Simulink Model

To linearize your model with the `linearize` function using the analysis points saved in the model, you must first extract the analysis points using the `getlinio` function.

```
io = getlinio(mdl);
```

Here, `mdl` is a character vector specifying the name of a model in the current working folder or on the MATLAB path.

See Also

`linearize` | `linio` | `getlinio` | `setlinio`

More About

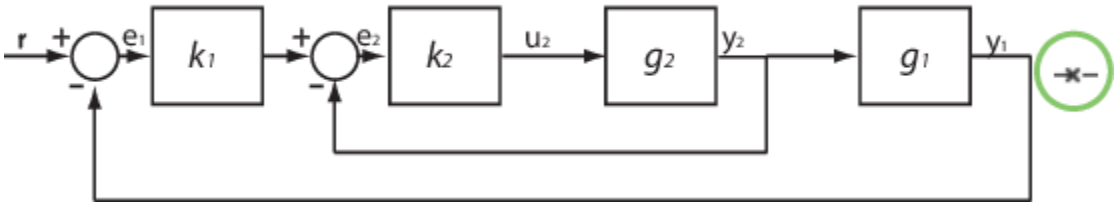
- “Specify Portion of Model to Linearize” on page 2-10
- “Linearize Simulink Model at Model Operating Point” on page 2-54
- “Compute Open-Loop Response” on page 2-46

How the Software Treats Loop Openings

Simulink Control Design software linearizes models using a block-by-block approach. The software individually linearizes each block in your Simulink model and produces the linearization of the overall system by combining the individual block linearizations. For more information, see “Exact Linearization Algorithm” on page 2-175.

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

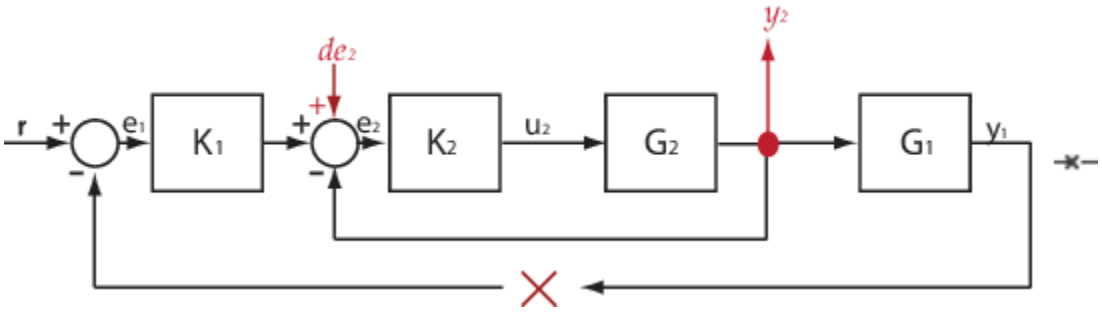
For example, in the following model, to compute the response from e_2 to y_2 without the effects of the outer loop, you open the outer loop by placing a loop opening analysis point at y_1.



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

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

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



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

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

See Also

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

More About

- “Specify Portion of Model to Linearize” on page 2-10
- “Mark Signals of Interest for Batch Linearization” on page 3-9
- “Mark Signals of Interest for Control System Analysis and Design” on page 2-38
- “Compute Open-Loop Response” on page 2-46

Linearize Plant

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

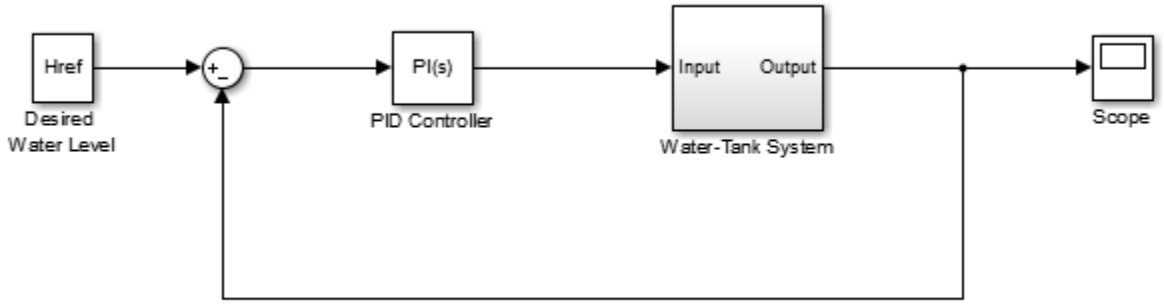
Linearizing a block in this way is equivalent to specifying open-loop input and open-loop output analysis points at the block inputs and outputs, respectively. For more information on specifying analysis points in your model, see “Specify Portion of Model to Linearize” on page 2-10.

Linearize Plant Using Model Linearizer

This example shows how to linearize a plant subsystem in a Simulink model using the **Model Linearizer**.

Open Simulink model.

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

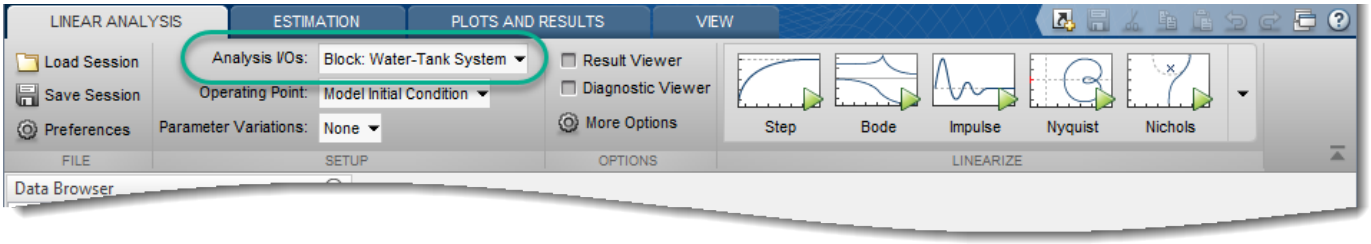


For this model, the Water-Tank System block contains all the nonlinear dynamics. To linearize the block, use the **Model Linearizer**.

To open the **Model Linearizer** with the inputs and outputs of the block selected as the linearization I/O set, first open the **Linearization** tab. To do so, in the Simulink model window, in the **Apps** gallery, click **Linearization Manager**.

In the model, click the Water-Tank System block. Then, on the **Linearization** tab, click **Linearize Block**.

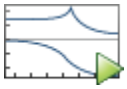
In the **Model Linearizer**, on the **Linear Analysis** tab, in the **Analysis I/Os** drop-down list, the software sets the I/O set for linearization to `Block: Water-Tank System`.



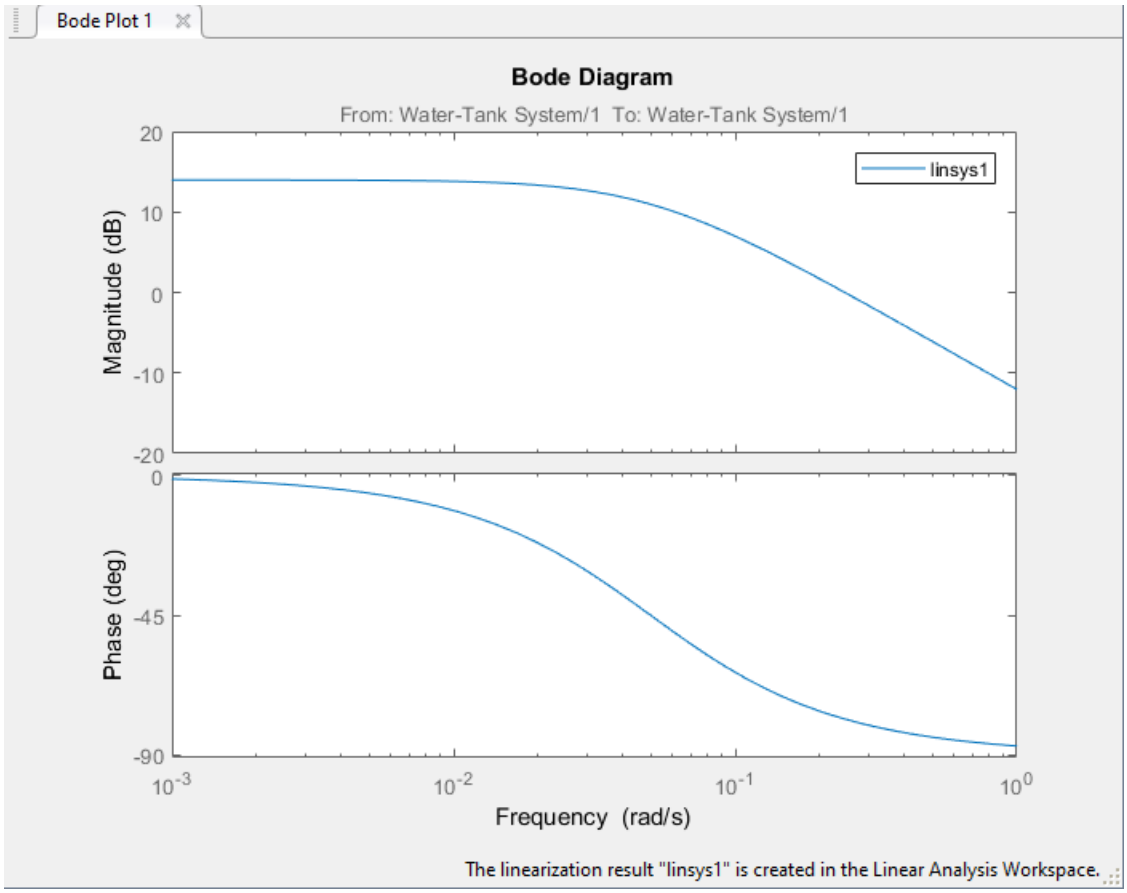
Alternatively, if the **Model Linearizer** is already open for your system, in the Simulink model window, click the Water-Tank System block. Then, in the **Model Linearizer**, in the **Analysis I/Os** drop-down list, select `Linearize the Currently Selected Block`.

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

For this example, use the model operating point for linearization. The model operating point consists of the initial state values and input signals stored in the model. In the **Model Linearizer**, on the **Linear Analysis** tab, in the **Operating Point** drop-down list, leave `Model Initial Condition` selected. For information on linearizing models at different operating points, see “Linearize at Trimmed Operating Point” on page 2-67 and “Linearize at Simulation Snapshot” on page 2-72.

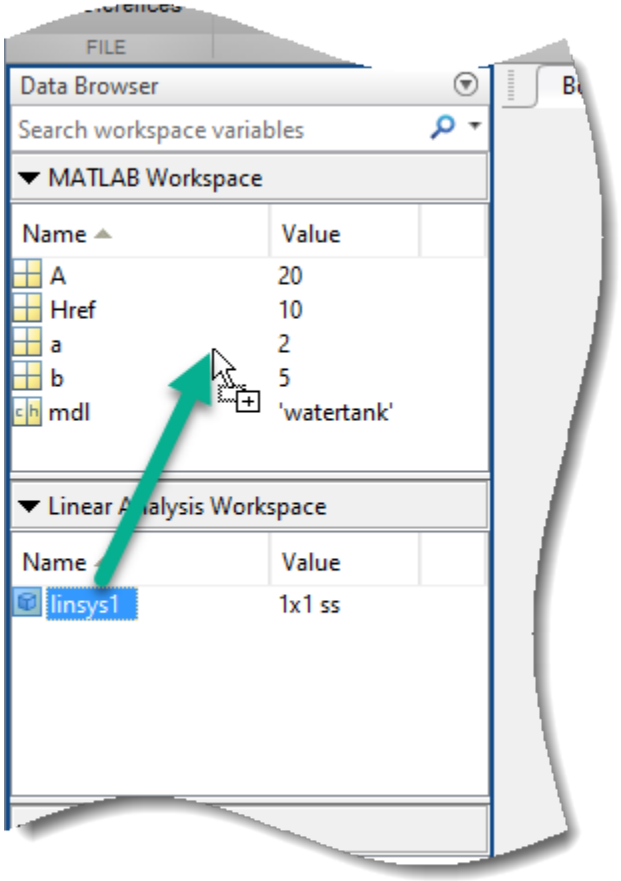
To linearize the specified block and generate a Bode plot for the resulting linear model, click  **Bode**.

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



For more information on analyzing linear models, see “Analyze Results Using Model Linearizer Response Plots” on page 2-113.

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

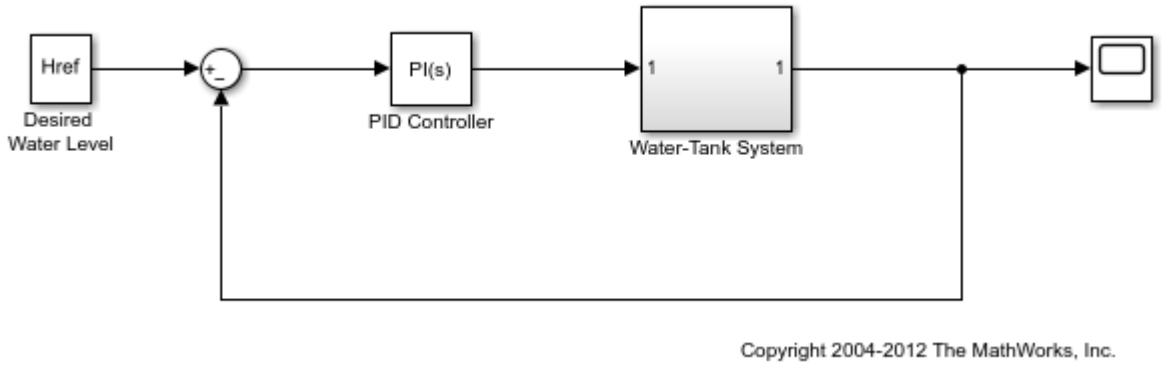


Linearize Plant at Command Line

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

Open Simulink model.

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



Copyright 2004-2012 The MathWorks, Inc.

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

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

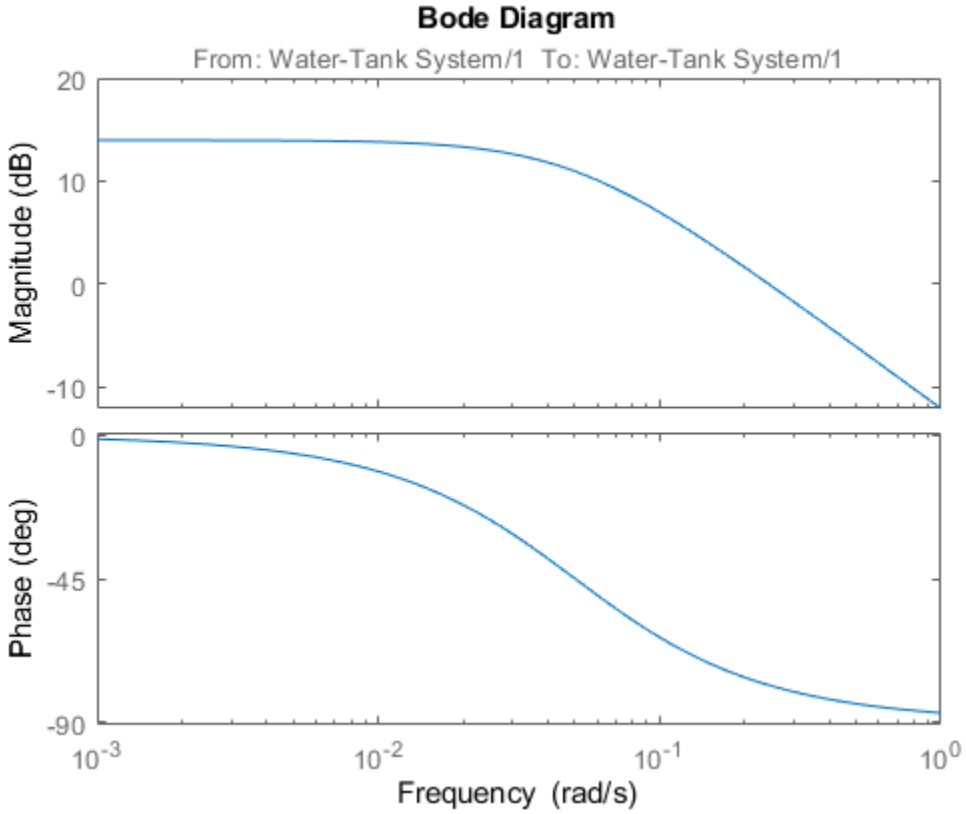
Then, linearize the plant subsystem at the model operating point.

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

The model operating point consists of the initial state values and input signals stored in the model. For information on linearizing models at different operating points, see “Linearize at Trimmed Operating Point” on page 2-67 and “Linearize at Simulation Snapshot” on page 2-72.

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

```
bode(linsys1)
```



For more information on analyzing linear models, see “Linear Analysis”.

See Also

Model Linearizer | `linearize`

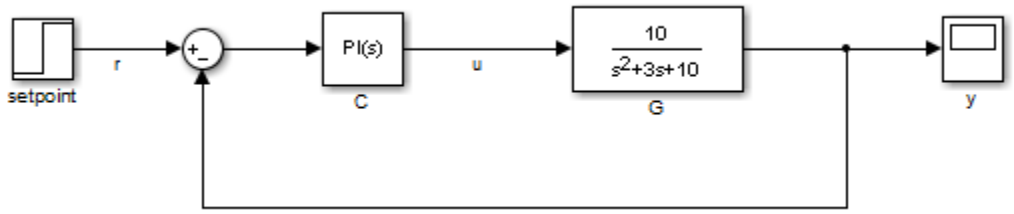
More About

- “Linearize Simulink Model at Model Operating Point” on page 2-54
- “Compute Open-Loop Response” on page 2-46
- “Visualize Bode Response of Simulink Model During Simulation” on page 2-61

Mark Signals of Interest for Control System Analysis and Design

Analysis Points

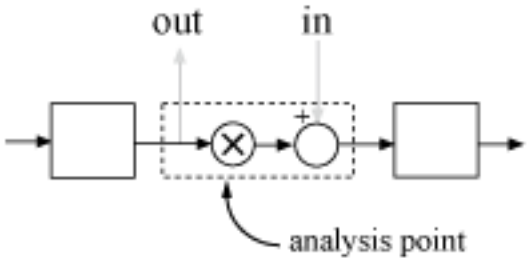
Whether you model your control system in MATLAB or Simulink, use analysis points to mark points of interest in the model. Analysis points allow you to access internal signals, perform open-loop analysis, or specify requirements for controller tuning. In the block diagram representation, an analysis point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



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

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

You can apply these purposes concurrently. For example, to compute the open-loop response from `u` to `y`, you can treat `u` as both a loop opening and an input. When you use an analysis point for more than one purpose, the software applies the purposes in this sequence: output measurement, then loop opening, then input.



Using analysis points, you can extract open-loop and closed-loop responses from a control system model. For example, suppose `T` represents the closed-loop system in the model above, and `u` and `y` are marked as analysis points. `T` can be either a generalized state-space model or an `slLinearizer`

or `slTuner` interface to a Simulink model. You can plot the closed-loop response to a step disturbance at the plant input with the following commands:

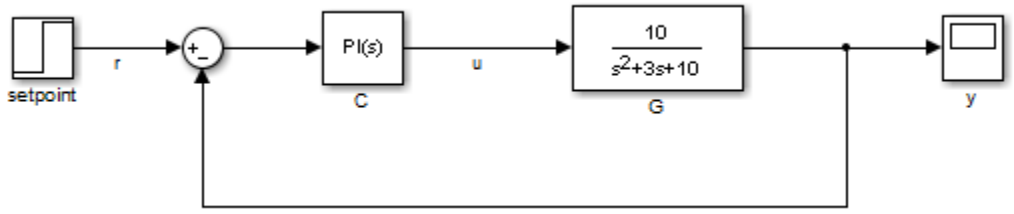
```
Tuy = getIOTransfer(T,'u','y');
stepplot(Tuy)
```

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

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

Specify Analysis Points for MATLAB Models

Consider an LTI model of the following block diagram.




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

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

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

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

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

Specify Analysis Points for Simulink Models

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

To specify analysis points directly in your Simulink model, first open the **Linearization** tab. To do so, in the **Apps** gallery, click **Linearization Manager**.

To specify an analysis point:

1. In the model, click the signal you want to define as an analysis point.
2. On the **Linearization** tab, in the **Insert Analysis Points** gallery, select the type of analysis point you want to define.

   When you specify analysis points, the software adds annotations to your model indicating the linear analysis point type.
3. Repeat steps 1 and 2 for all signals you want to define as analysis points.

You can select any of the following closed-loop analysis point types, which are equivalent within an `slLinearizer` or `slTuner` interface; that is, they are treated the same way by analysis functions, such as `getIOTransfer`, and tuning goals, such as `TuningGoal.StepTracking`.

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

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

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

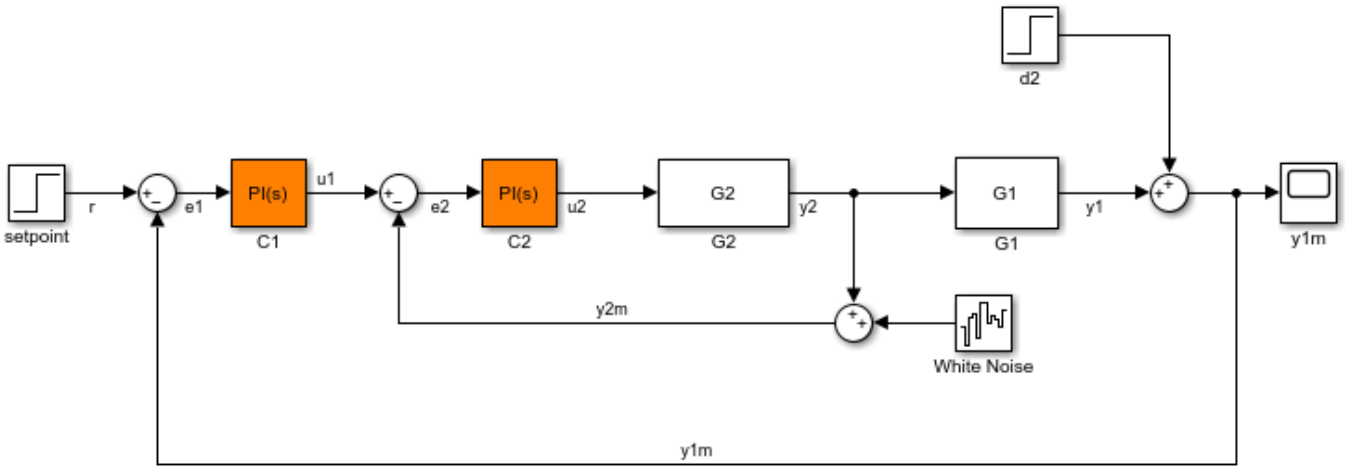
When you define a signal as an open-loop point, analysis functions such as `getIOTransfer` always enforce a loop break at that signal during linearization. All open-loop analysis point types are equivalent within an `slLinearizer` or `slTuner` interface. For more information on how the software treats loop openings during linearization, see “How the Software Treats Loop Openings” on page 2-31.

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

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

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

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



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

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

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

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

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

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

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

- Name of the signal.

  ```
  addPoint(ST,'y2');
  ```
- Combined source block path and port number.

  ```
  addPoint(ST,'scdcascade/C1/1')
  ```
- End of the full source block path when unambiguous.

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

You can also add permanent openings to an `slLinearizer` or `slTuner` interface using the `addOpening` command, and specifying signals in the same way as for `addPoint`. For more information on how the software treats loop openings during linearization, see “How the Software Treats Loop Openings” on page 2-31.

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

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

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

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

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

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

Refer to Analysis Points for Analysis and Tuning

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

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

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

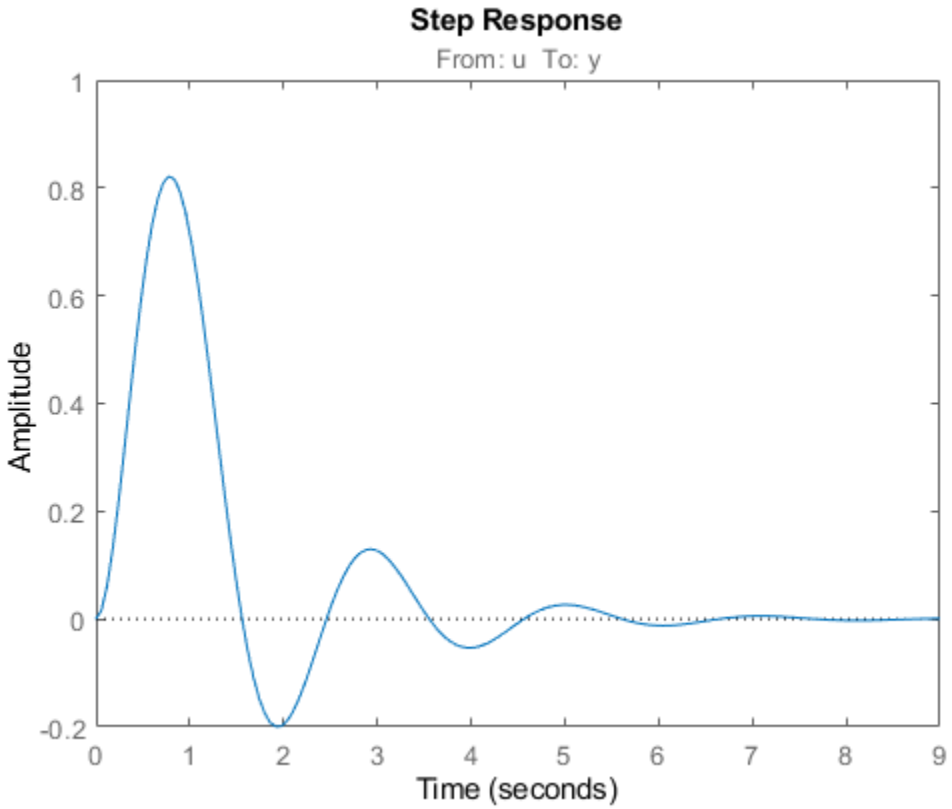
To view the available analysis points, use the `getPoints` function. You can view the analysis for models created:

- At the command line:
- In Simulink:

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

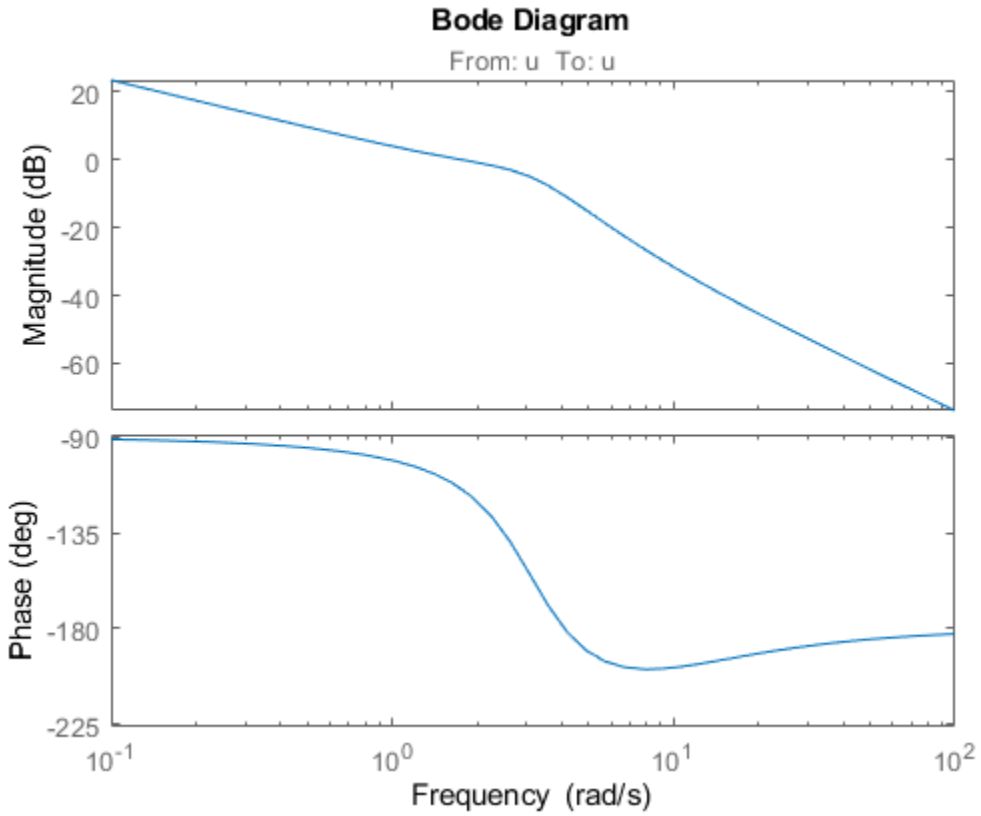
- Computing a closed-loop response.

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



- Computing an open-loop response.

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



- Creating tuning goals for `systune`.

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

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

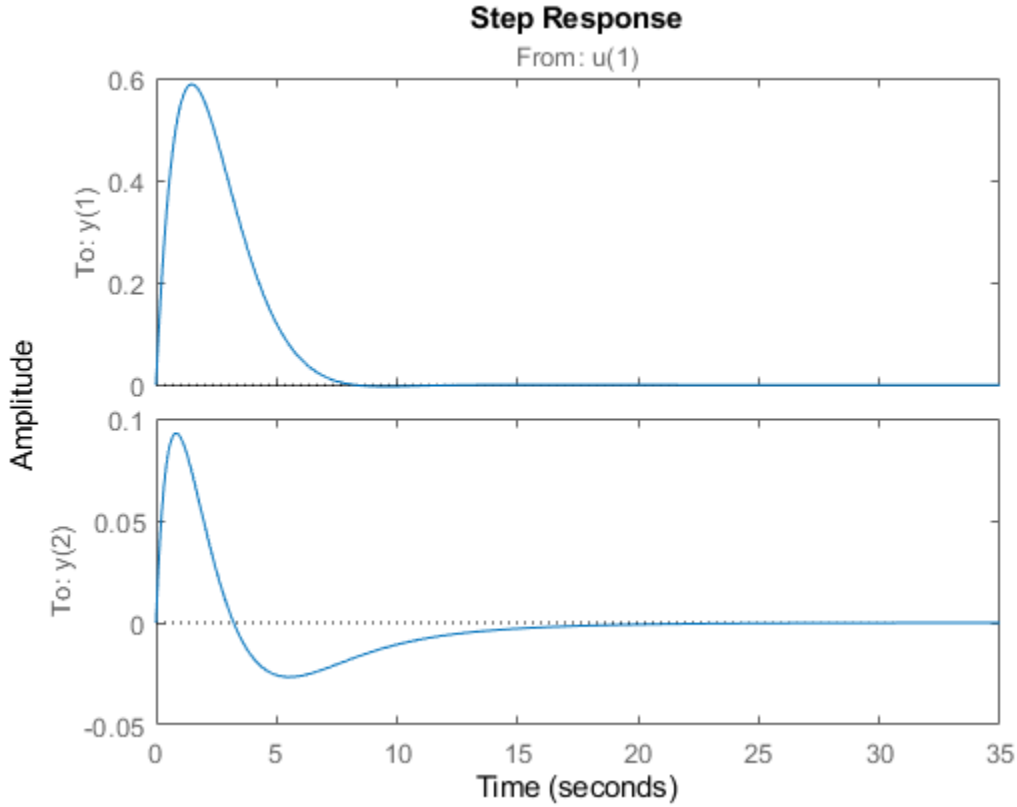
```
ioSys = getIOTransfer(ST,'u1','y1');
sensG2 = getSensitivity(ST,'G2');
R = TuningGoal.Margins('u1',10,60);
```

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

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

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

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



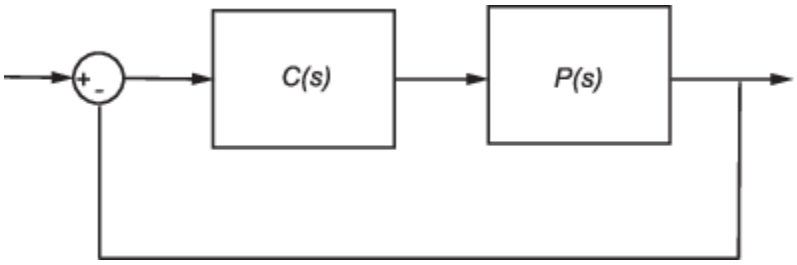
When you create tuning goals in **Control System Tuner**, the software creates analysis points as needed.

See Also

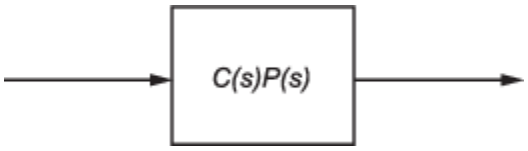
`addPoint` | `getPoints` | `slLinearizer` | `getIOTransfer`

Compute Open-Loop Response

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

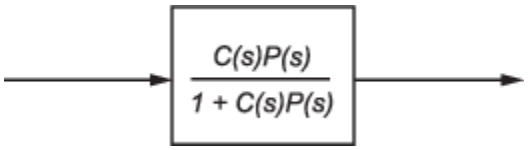


If the controller, $C(s)$, and plant, $P(s)$, are linear, the corresponding open-loop transfer function is $C(s)P(s)$.

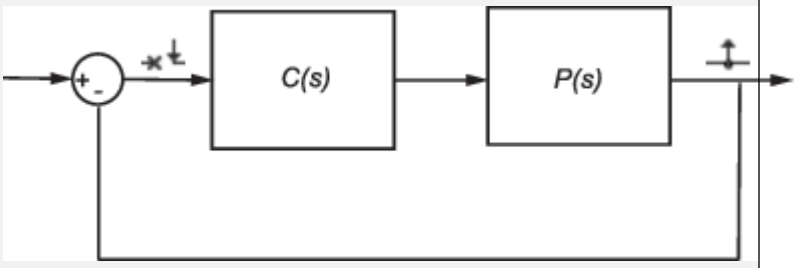
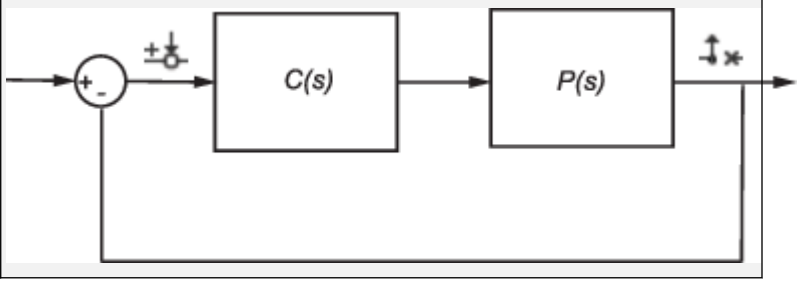


To remove the effects of the feedback loop, insert a loop opening analysis point without manually breaking the signal line. Manually removing the feedback signal from a nonlinear model changes the model operating point and produces a different linearized model. For more information, see “How the Software Treats Loop Openings” on page 2-31.

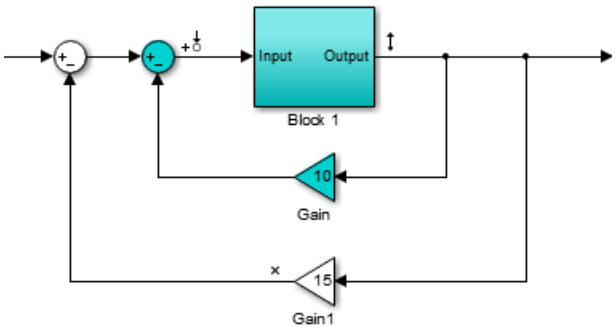
If you do not insert a loop opening, the resulting linear model includes the effects of the feedback loop.



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

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

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



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

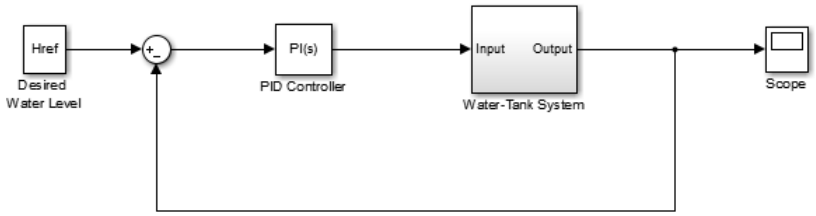
You can specify analysis points directly in your Simulink model, in the **Model Linearizer**, or at the command line. For more information, about the different types of analysis points and how to define them, see “Specify Portion of Model to Linearize” on page 2-10.

Compute Open-Loop Response Using Model Linearizer

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

Open Simulink model.

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

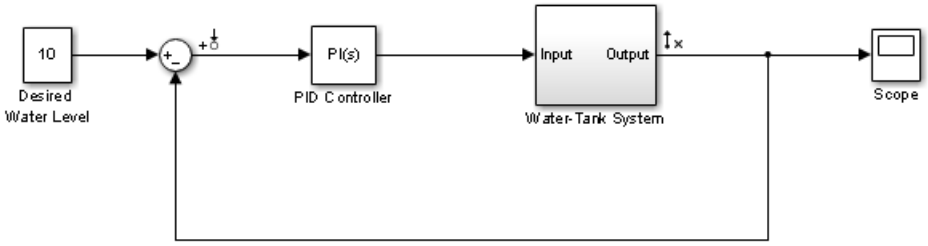


The Water-Tank System block represents the plant in this control system and contains all of the system nonlinearities.

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

1. Open the **Linearization** tab. To do so, in the **Apps** gallery, click **Linearization Manager**.
2. To specify an analysis point for a signal, click the signal in the model. Then, on the **Linearization** tab, in the **Insert Analysis Points** gallery, select the type of analysis point.
   - Configure the input signal of the PID Controller block as an **Input Perturbation**.
   - Configure the output signal of the Water-Tank System block as an **Open-loop Output**.

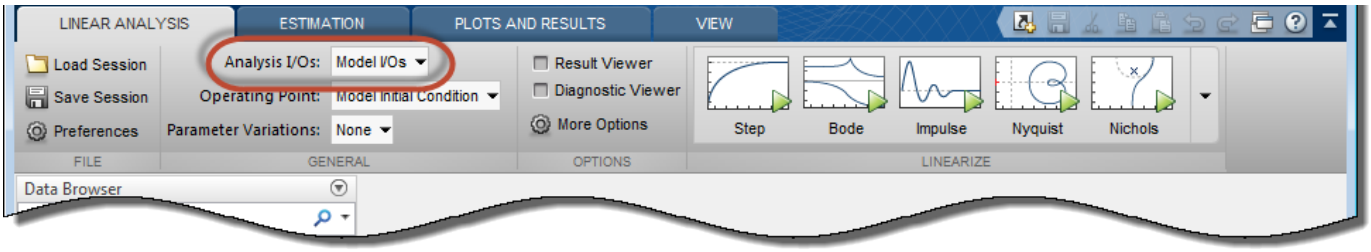
Annotations appear in the model indicating which signals are designated as analysis points.

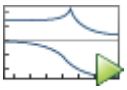


Tip If you do not want to introduce changes to the Simulink model, you can specify the analysis points in the **Model Linearizer**. For more information, see “Specify Portion of Model to Linearize in Model Linearizer” on page 2-22.

Open the **Model Linearizer** for the model. In the Simulink model window, in the **Apps** gallery, click **Model Linearizer**.

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

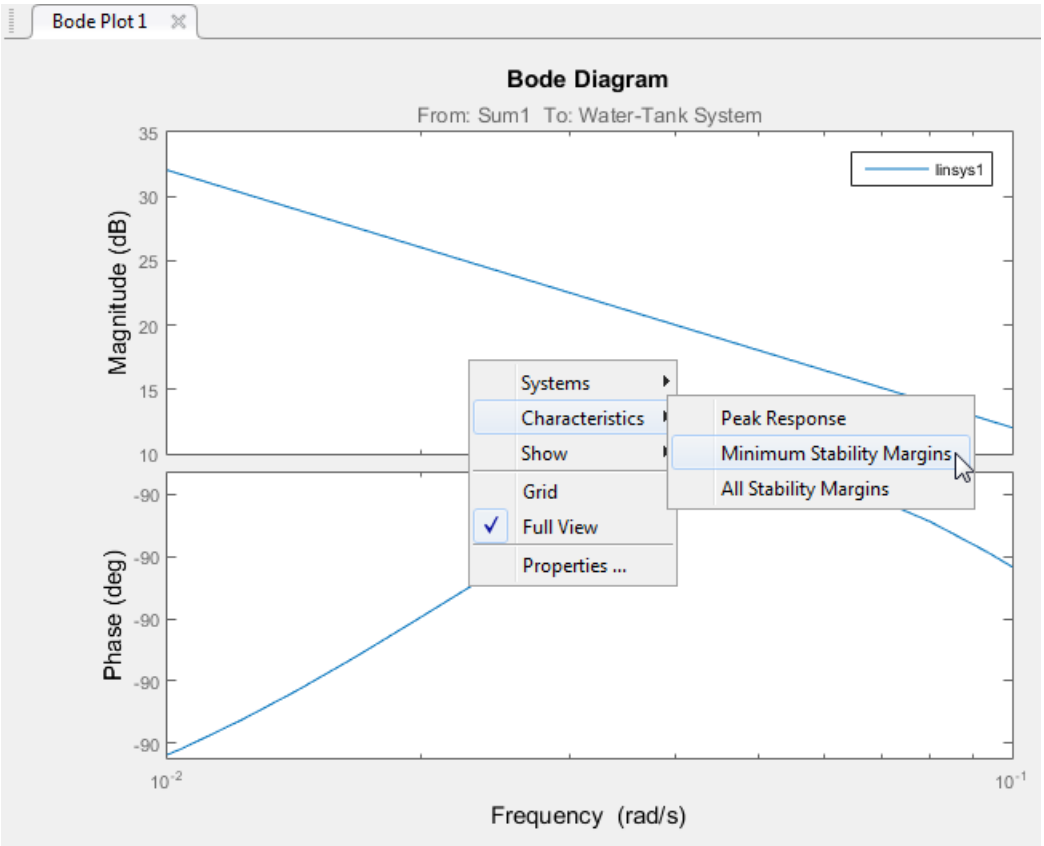


To linearize the model using the specified analysis points and generate a Bode plot of the linearized model, click  **Bode**.

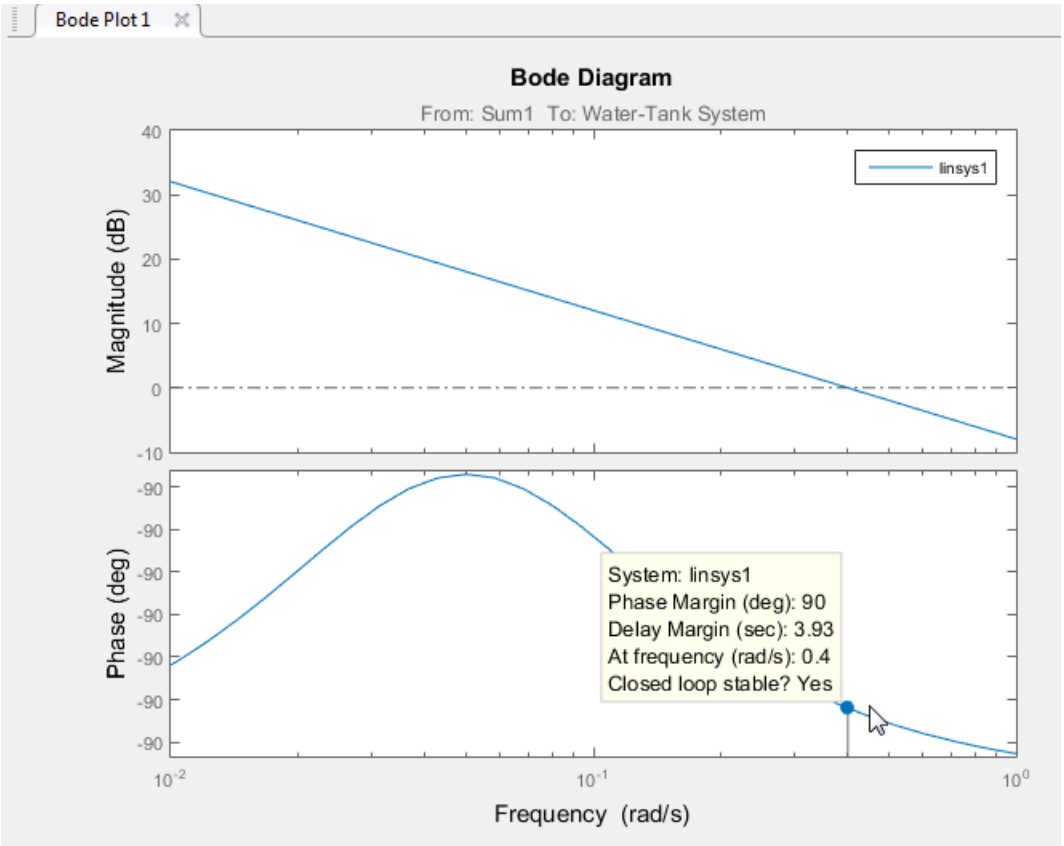
By default, the **Model Linearizer** linearizes the model at the model initial conditions, as shown in the **Operating Point** drop-down list. For examples of linearizing a model at a different operating point, see “Linearize at Trimmed Operating Point” on page 2-67 and “Linearize at Simulation Snapshot” on page 2-72.

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

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



The Bode plot displays the phase margin marker. To show a data tip that contains the phase margin value, click the marker.



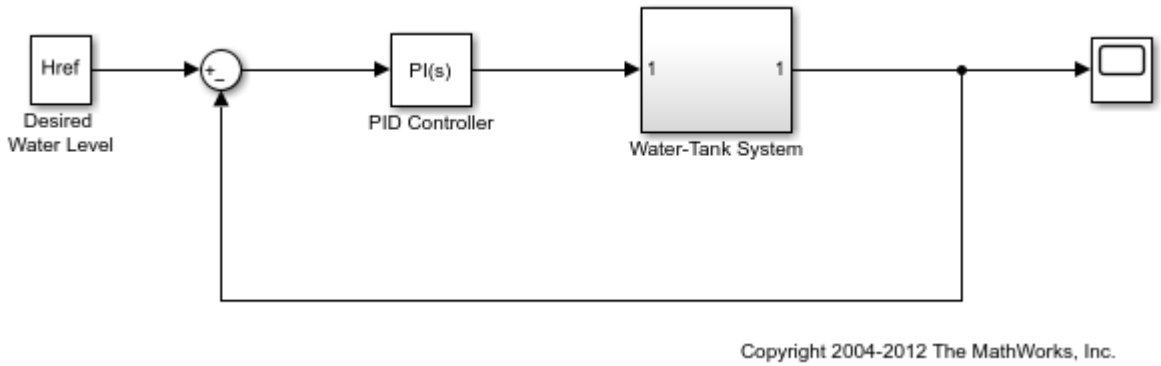
For this system, the phase margin is 90 degrees at a crossover frequency of 0.4 rad/s.

Compute Open-Loop Response at the Command Line

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

Open Simulink model.

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



Copyright 2004-2012 The MathWorks, Inc.

Specify the portion of the model to linearize by creating an array of analysis points using the `linio` command:

- Open-loop input point at the input of the PID Controller block. This signal originates at the output of the Sum1 block.
- Output measurement at the output of the Water-Tank System block.

```
io(1) = linio('watertank/Sum1',1,'openinput');
io(2) = linio('watertank/Water-Tank System',1,'output');
```

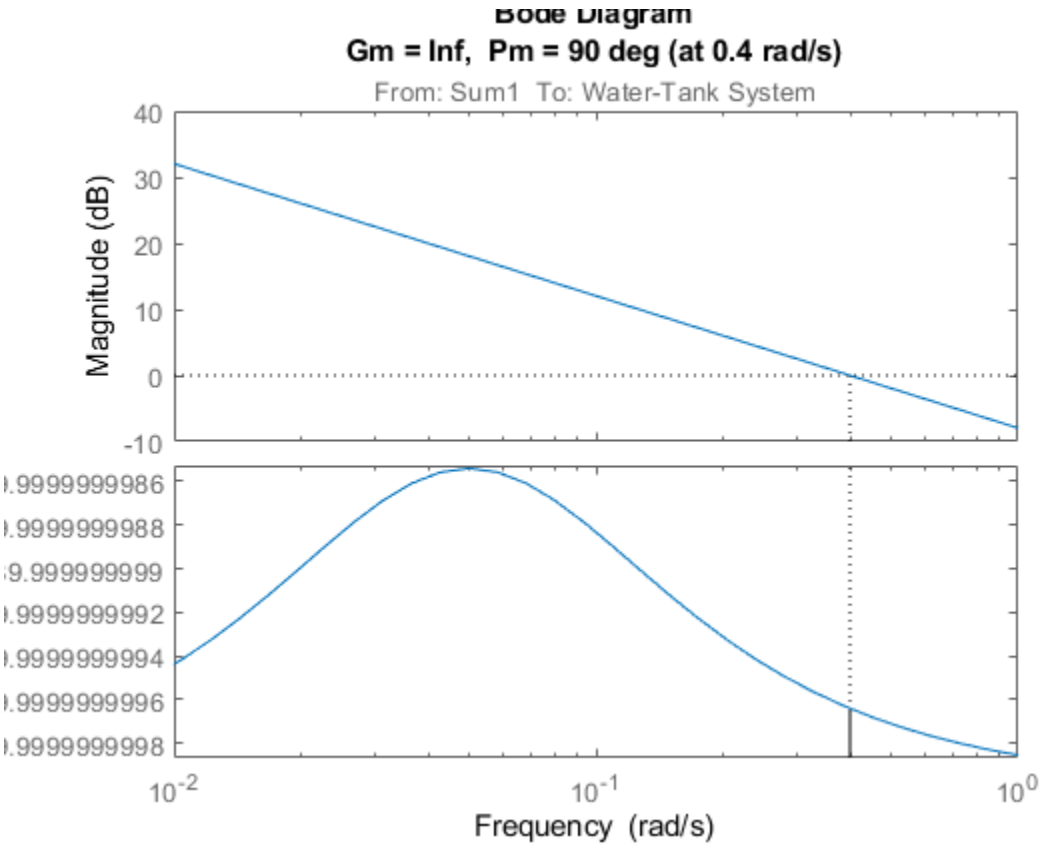
The open-loop input analysis point includes a loop opening, which breaks the signal flow and removes the effects of the feedback loop.

Linearize the model at the default model operating point using the `linearize` command.

```
linsys = linearize(sys,io);
```

`linsys` is the linearized open-loop transfer function of the system. You can now analyze the response by, for example, plotting its frequency response and viewing the gain and phase margins.

```
margin(linsys)
```



For this system, the gain margin is infinite, and the phase margin is 90 degrees at a crossover frequency of 0.4 rad/s.

See Also

Model Linearizer | `linearize`

More About

- “Specify Portion of Model to Linearize” on page 2-10
- “How the Software Treats Loop Openings” on page 2-31
- “Linearize Simulink Model at Model Operating Point” on page 2-54
- “Linearize Plant” on page 2-33
- “Visualize Bode Response of Simulink Model During Simulation” on page 2-61

Linearize Simulink Model at Model Operating Point

If you do not specify an operating point when linearizing a Simulink model, the software uses the operating point specified in the model by default. The model operating point consists of the initial state and input signal values stored in the model.

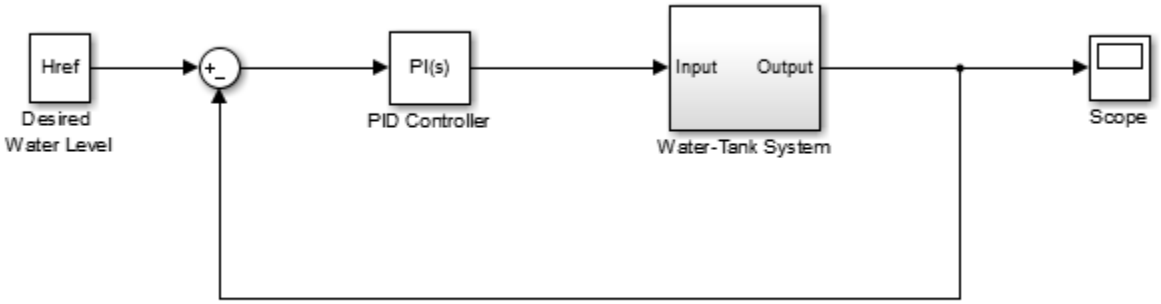
For information on linearizing models at different operating points, see “Linearize at Trimmed Operating Point” on page 2-67 and “Linearize at Simulation Snapshot” on page 2-72.

Linearize Simulink Model Using Model Linearizer

This example shows how to linearize a Simulink model at the operating point specified in the model using the **Model Linearizer**.

Open Simulink model.

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



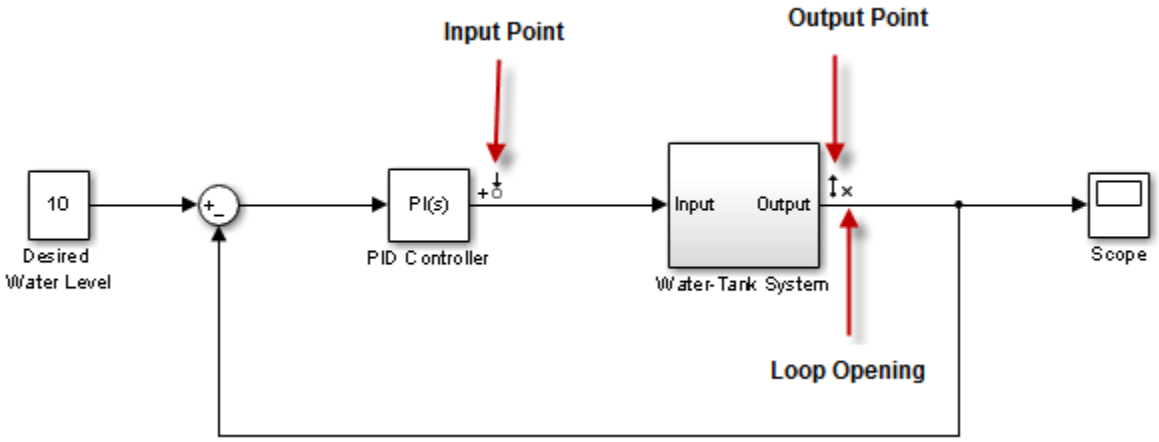
The Water-Tank System block represents the plant in this control system and includes all of the system nonlinearities.

To specify the portion of the model to linearize, first open the **Linearization** tab. To do so, in the Simulink window, in the **Apps** gallery, click **Linearization Manager**.

To specify an analysis point for a signal, click the signal in the model. Then, on the **Linearization** tab, in the **Insert Analysis Points** gallery, select the type of analysis point.

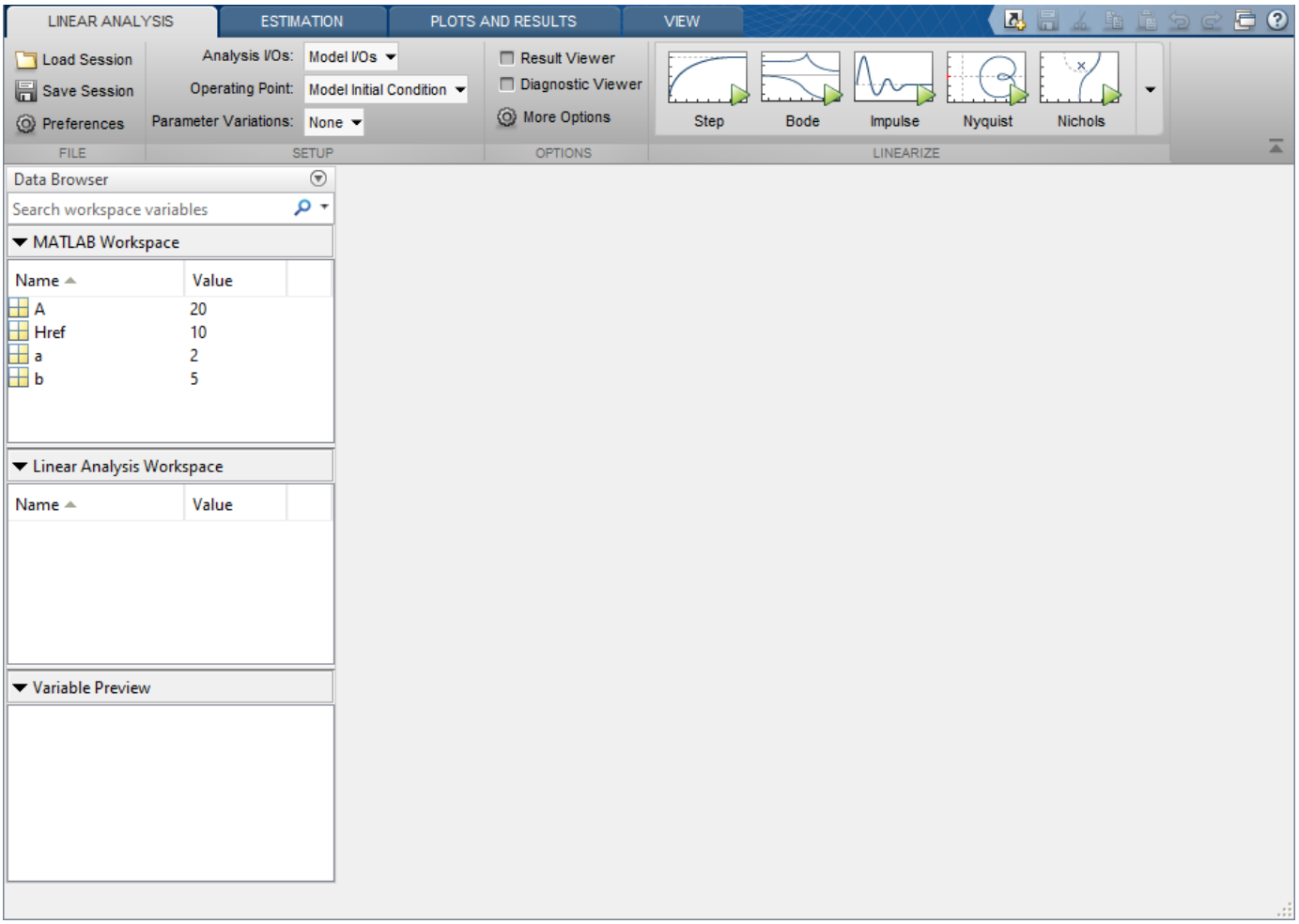
- Configure the output signal of the PID Controller block as an **Input Perturbation**.
- Configure the output signal of the Water-Tank System block as an **Open-loop Output**. An open-loop output point is an output measurement followed by a loop opening, which removes the effects of the feedback signal on the linearization without changing the model operating point.

When you add linear analysis points, the software adds markers at their respective locations in the model. For more information on the different types of analysis points, see “Specify Portion of Model to Linearize” on page 2-10.



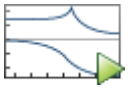
For more information on defining analysis points in a Simulink model, see “Specify Portion of Model to Linearize in Simulink Model” on page 2-17. Alternatively, if you do not want to introduce changes to the Simulink model, you can define analysis points using the **Model Linearizer**. For more information, see “Specify Portion of Model to Linearize in Model Linearizer” on page 2-22.

To open the **Model Linearizer** for the model, in the Simulink model window, in the **Apps** gallery, click **Model Linearizer**.

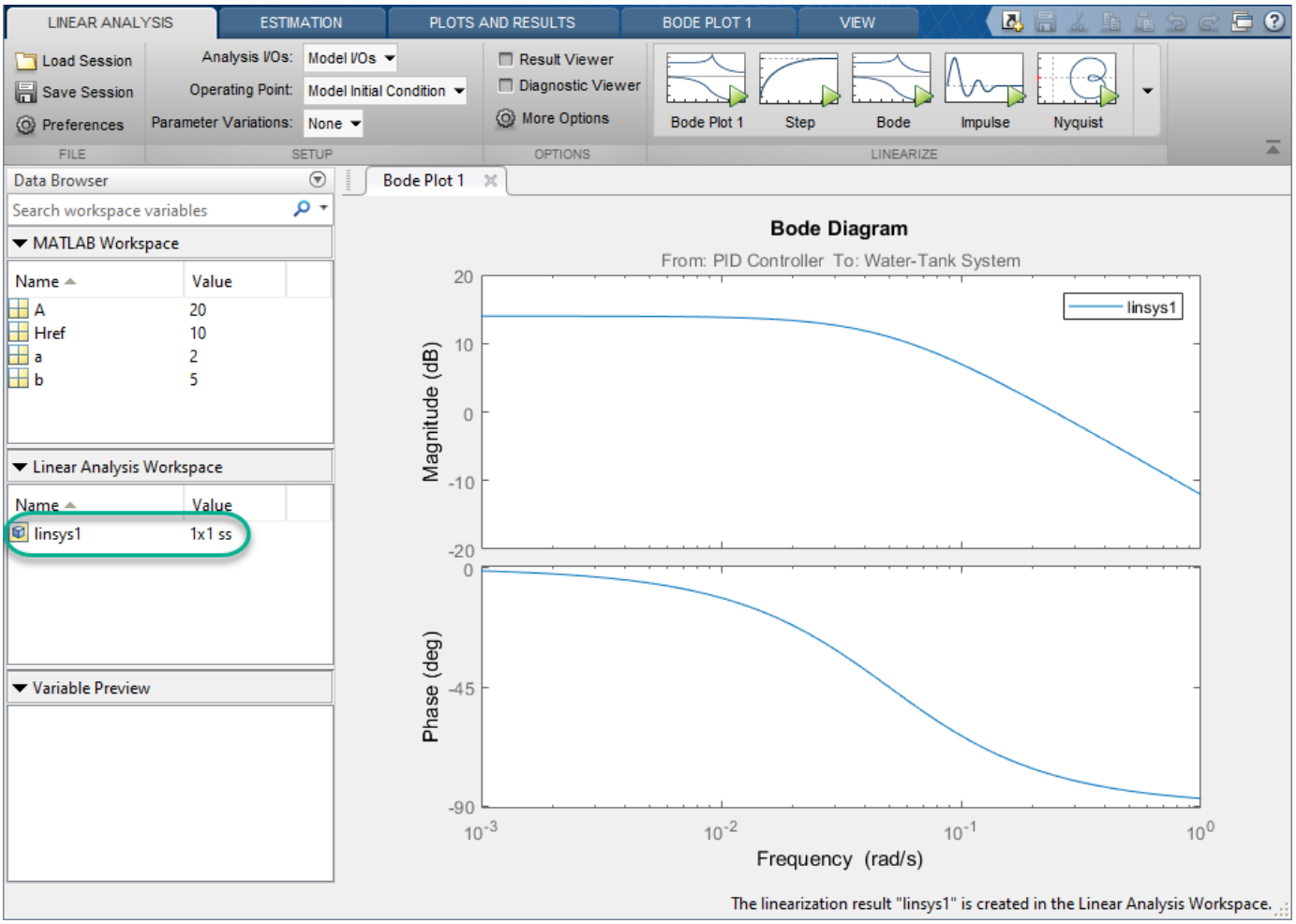


To use the analysis points you defined in the Simulink model as linearization I/Os, on the **Linear Analysis** tab, in the **Analysis I/Os** drop-down list, leave `Model I/Os` selected.

For this example, use the model operating point for linearization. In the **Operating Point** drop-down list, leave `Model Initial Condition` selected.

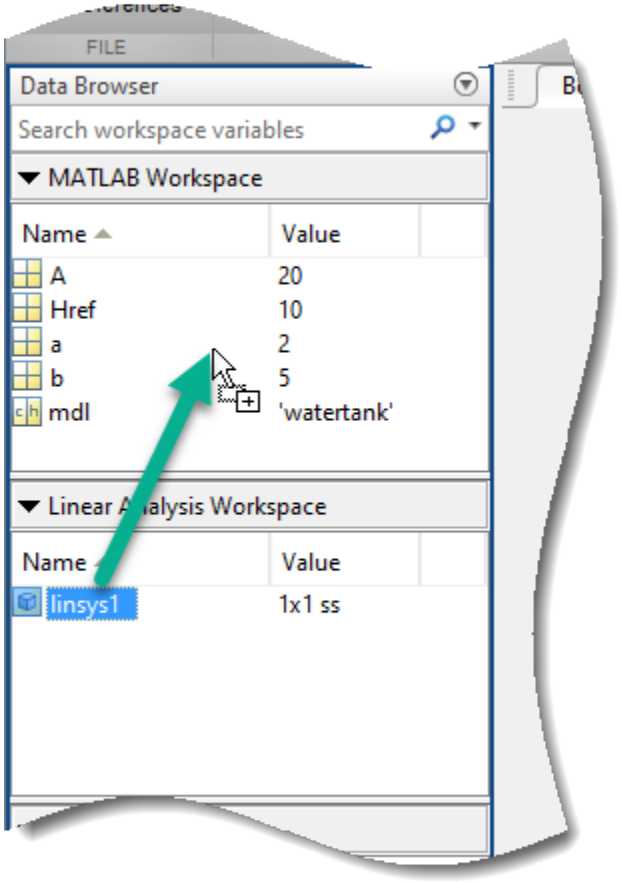
To linearize the system and generate a response plot for analysis, in the **Linearize** section, click a response. For this example, to generate a Bode plot for the resulting linear model, click  **Bode**.

The software adds the linearized model, `linsys1`, to the **Linear Analysis Workspace** and generates a Bode plot for the model. `linsys1` is the linear model from the specified input to the specified output, computed at the default model operating point.



For more information on analyzing linear models, see “Analyze Results Using Model Linearizer Response Plots” on page 2-113.

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

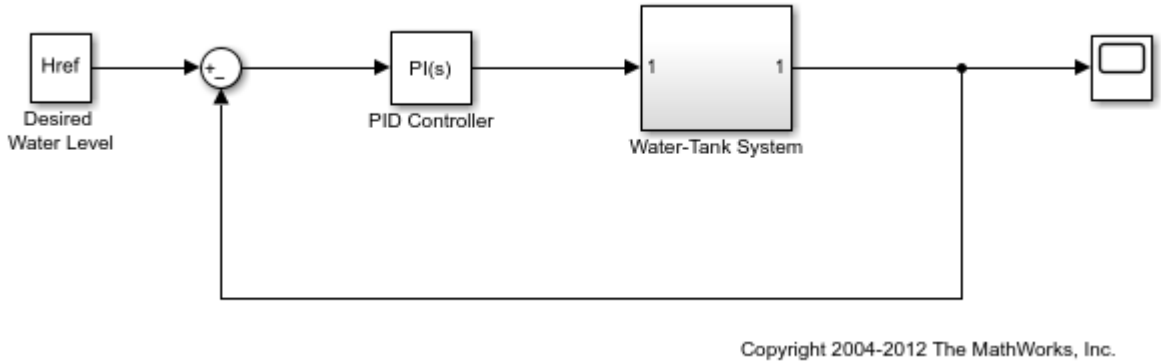


Linearize Simulink Model at Command Line

This example shows how to linearize a Simulink® model at the model operating point using the `linearize` command.

Open Simulink model.

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



For this system, the Water-Tank System block contains all the nonlinear dynamics. To specify the portion of the model to linearize, create an array of linearization I/O objects using the `linio` command.

Create an input perturbation analysis point at the output of the PID Controller block.

```
io(1) = linio('watertank/PID Controller',1,'input');
```

Create an open-loop output analysis point at the output of the Water-Tank System block. An open-loop output point is an output measurement followed by a loop opening, which removes the effects of the feedback signal on the linearization without changing the model operating point.

```
io(2) = linio('watertank/Water-Tank System',1,'openoutput');
```

For information on the different types of analysis points, see “Specify Portion of Model to Linearize” on page 2-10.

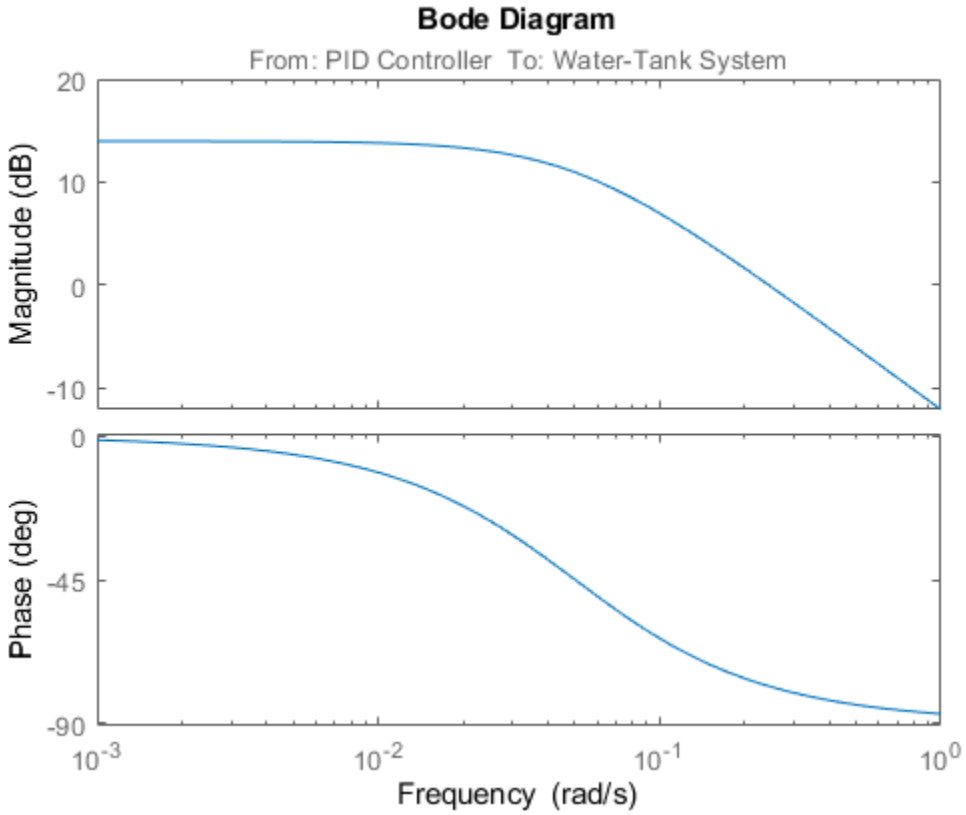
Linearize the model at the model operating point using the specified analysis points.

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

`linsys1` is the linear model from the specified input to the specified output, computed at the default model operating point.

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

```
bode(linsys1)
```



For more information on analyzing linear models, see “Linear Analysis”.

See Also

Model Linearizer | `linearize`

More About

- “Linearize at Trimmed Operating Point” on page 2-67
- “Linearize at Simulation Snapshot” on page 2-72
- “Linearize at Triggered Simulation Events” on page 2-75
- “Linearize Plant” on page 2-33
- “Compute Open-Loop Response” on page 2-46
- “Visualize Bode Response of Simulink Model During Simulation” on page 2-61

Visualize Bode Response of Simulink Model During Simulation

This example shows how to visualize linear system characteristics of a nonlinear Simulink model during simulation, computed at the model operating point (simulation snapshot time of 0).

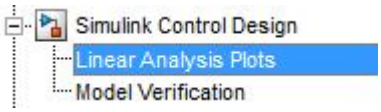
1. Open Simulink model.

   For example:

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

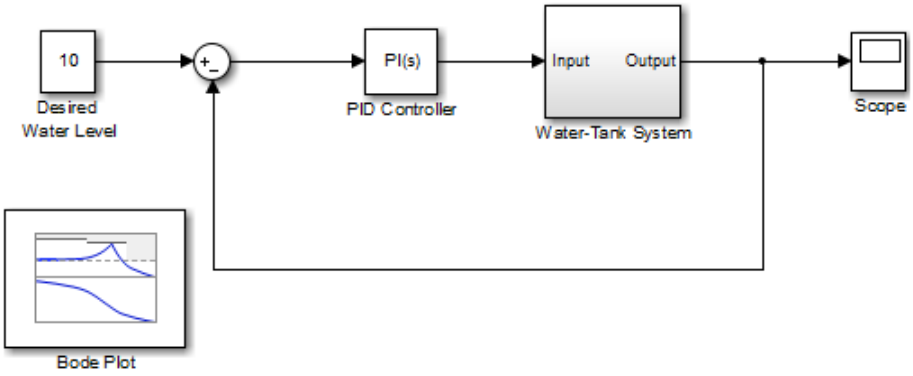
2. Open the Simulink Library Browser. In the Simulink Editor, on the **Simulation** tab, click **Library Browser**.
3. Add a plot block to the Simulink model.

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

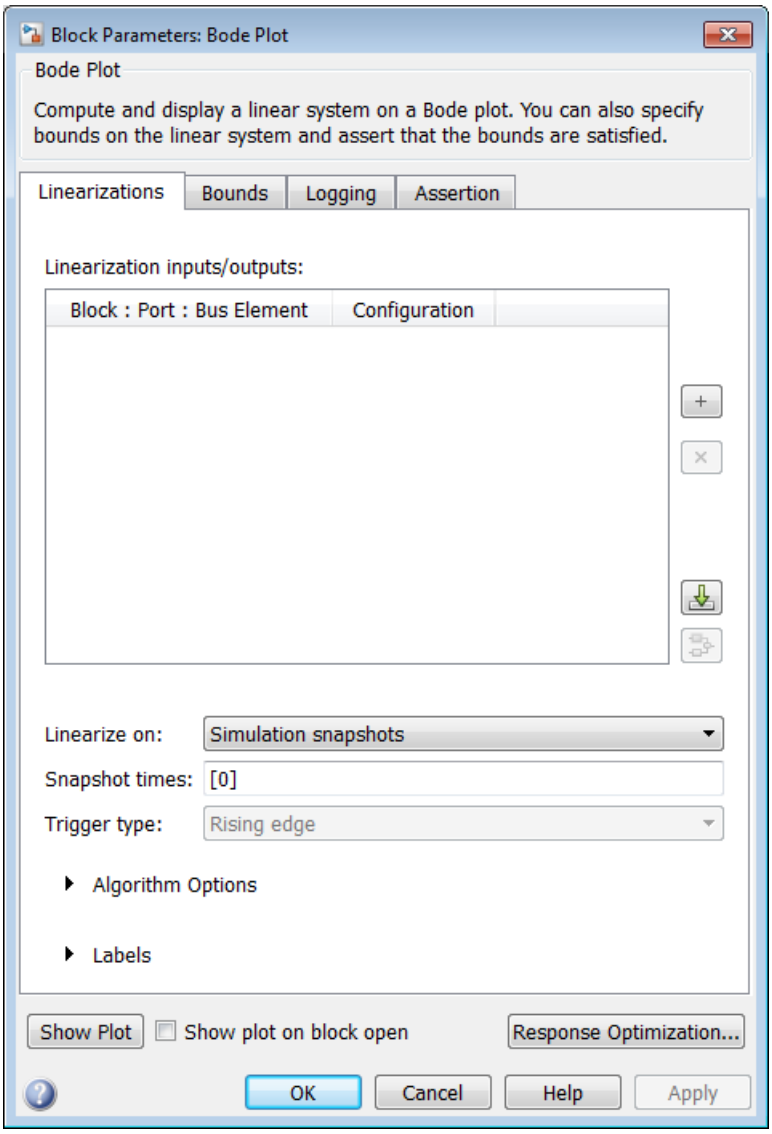
   

   b. Drag and drop a block, such as the Bode Plot block, into the model window.

   The model now resembles the following figure.



4. Double-click the block to open the Block Parameters dialog box.



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

5 Specify the linearization I/O points.

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

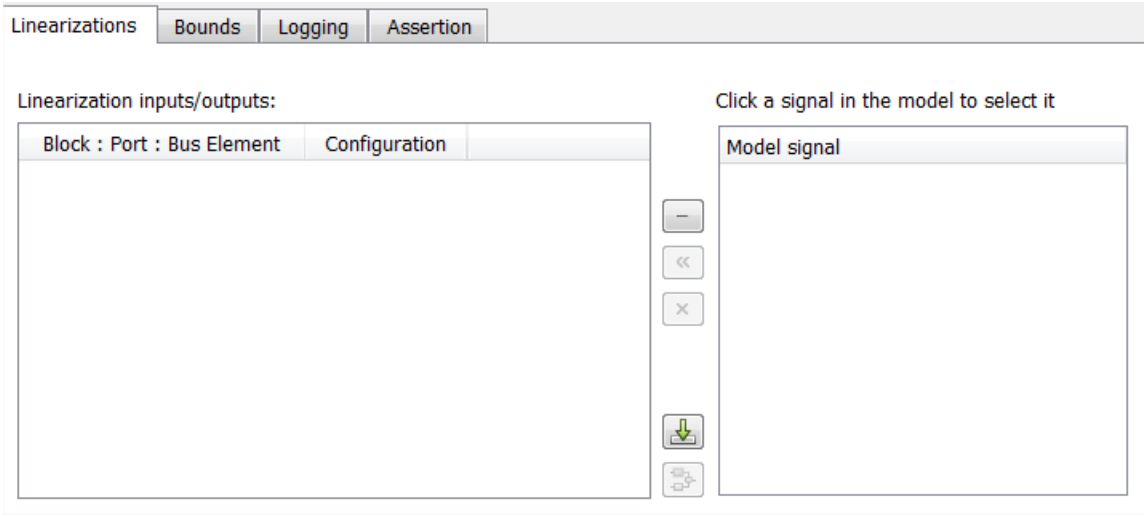
Tip If your model already contains I/O points, the block automatically detects these points and displays them. Click at any time to update the **Linearization inputs/outputs** table with I/Os from the model.

---

a To specify an input:

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

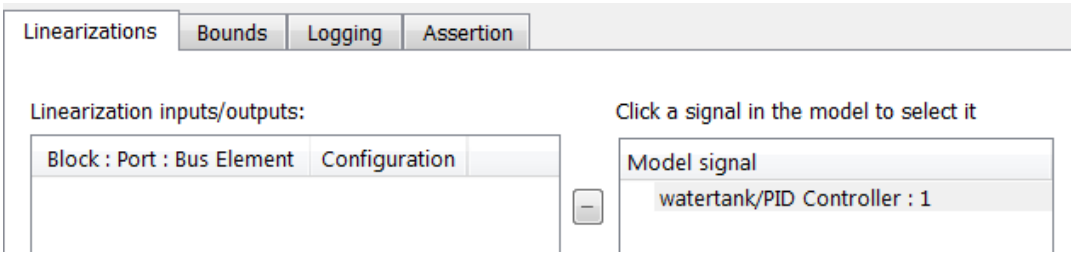
The Block Parameters dialog expands to display a **Click a signal in the model to select it** area.



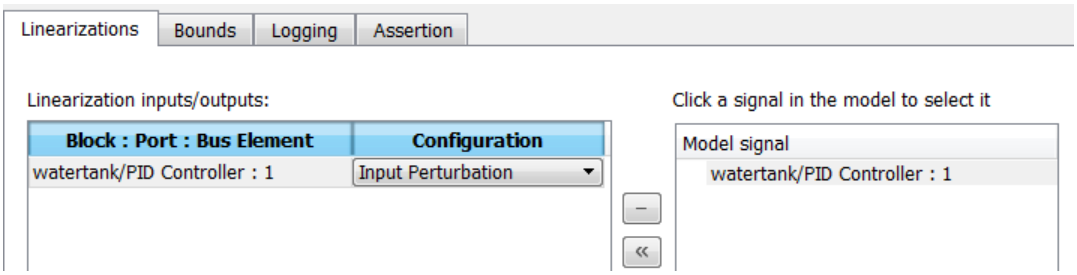
Tip You can select multiple signals at once in the Simulink model. All selected signals appear in the **Click a signal in the model to select it** area.

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

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



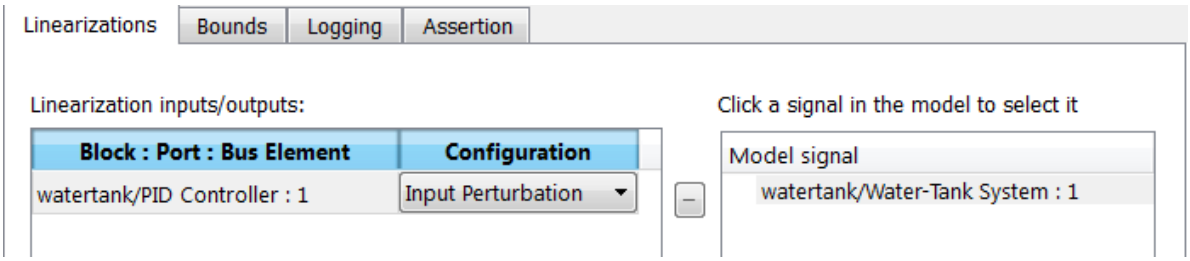
iii Click « to add the signal to the **Linearization inputs/outputs** table.



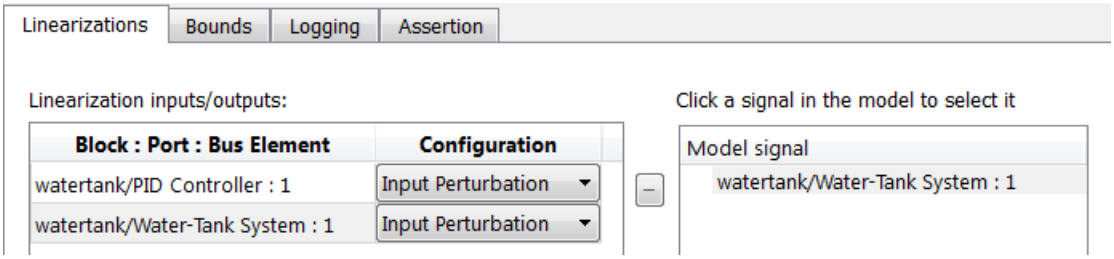
b To specify an output:

i In the Simulink model, click the output signal of the `Water-Tank System` block to select it.

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

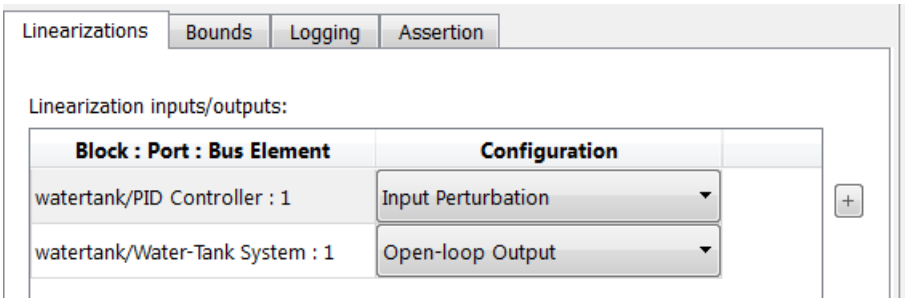


ii Click « to add the signal to the **Linearization inputs/outputs** table.



iii In the **Configuration** drop-down list of the **Linearization inputs/outputs** table, select `Open-loop Output` for **watertank/Water-Tank System : 1**.

The **Linearization inputs/outputs** table now resembles the following figure.



c Click − to collapse the **Click a signal in the model to select it** area.

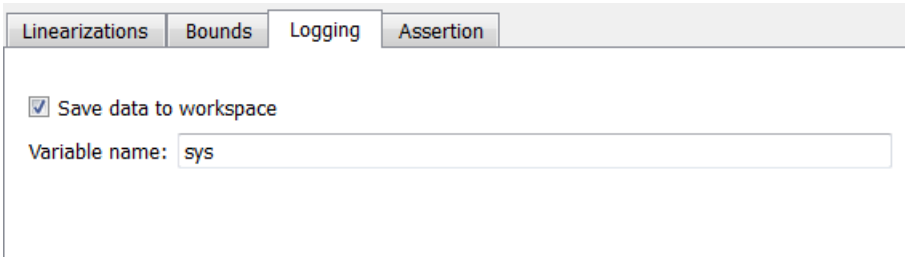
Tip Alternatively, before you add the Linear Analysis Plots block, right-click the signals in the Simulink model and select **Linear Analysis Points > Input Perturbation** and **Linear Analysis Points > Open-loop Output**. Linearization I/O annotations appear in the model and the selected signals appear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

6 Save the linear system.

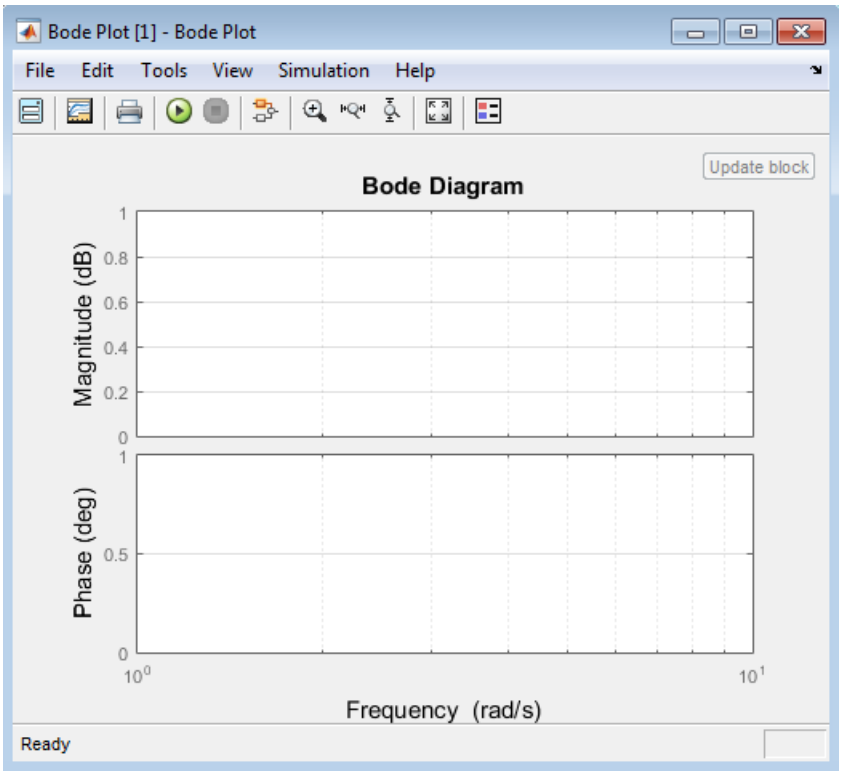
a Select the **Logging** tab.

b Select the **Save data to workspace** option, and specify a variable name in the **Variable name** field.

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



7 Click **Show Plot** to open an empty plot.

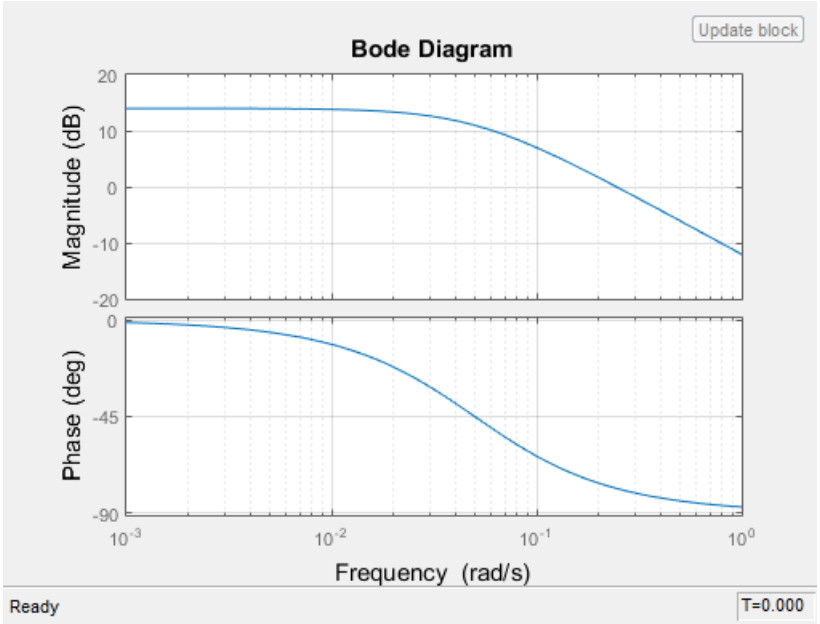


8 Plot the linear system characteristics by clicking in the plot window.

Alternatively, you can simulate the model from the model window.

The software linearizes the portion of the model between the linearization input and output at the default simulation time of 0, specified in **Snapshot times** parameter in the Block Parameters dialog box, and plots the Bode magnitude and phase.

After the simulation completes, the plot window resembles the following figure.



The computed linear system is saved as `sys` in the MATLAB workspace. `sys` is a structure with `time` and `values` fields. To view the structure, type:

```
sys
```

This command returns the following results:

```
sys =

         time: 0
       values: [1x1 ss]
    blockName: 'watertank/Bode Plot'
```

- The `time` field contains the default simulation time at which the linear system is computed.
- The `values` field is a state-space object which stores the linear system computed at simulation time of 0. To learn more about the properties of state-space objects, see `ss`.

(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.)

See Also

Bode Plot

More About

- “Visualize Linear System at Multiple Simulation Snapshots” on page 2-85
- “Plot Linear System Characteristics of a Chemical Reactor” on page 2-95
- “Visualize Linear System of a Continuous-Time Model Discretized During Simulation” on page 2-91

Linearize at Trimmed Operating Point

This example shows how to linearize a model at a trimmed steady-state operating point (equilibrium operating point) using the **Model Linearizer**.

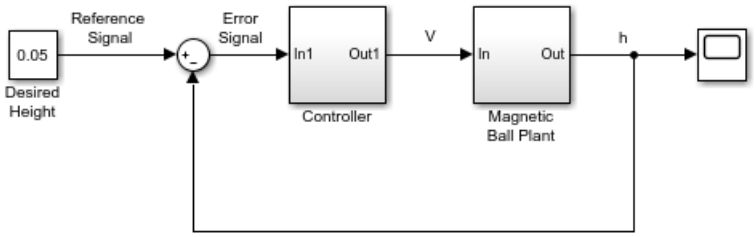
The operating point is *trimmed* by specifying constraints on the operating point values, and performing an optimization search that meets these state and input value specifications.

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 **Model Linearizer** for the model.

   In the Simulink model window, in the **Apps** gallery, click **Model Linearizer**.

3. To specify linearization input and output points, open the **Linearization** tab. To do so, in the **Apps** gallery, click **Linearization Manager**.

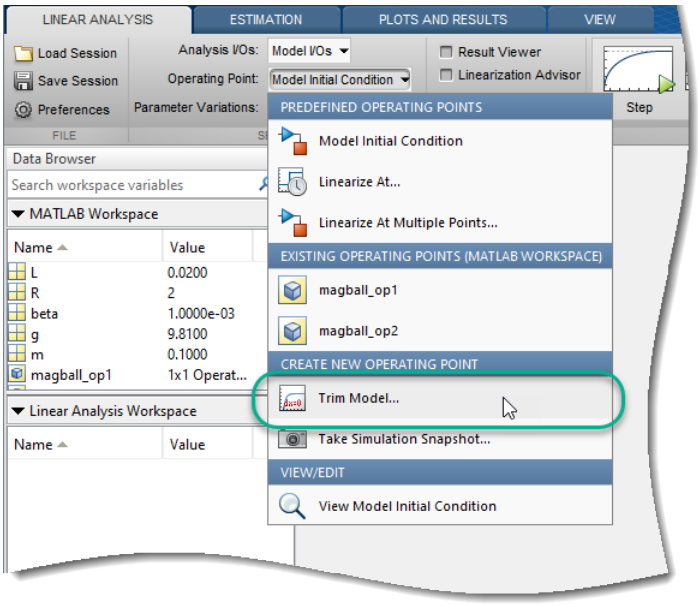
4. To specify an analysis point for a signal, click the signal in the model. Then, on the **Linearization** tab, in the **Insert Analysis Points** gallery, select the type of analysis point.

   - Configure the output signal of the Controller block as an **Input Perturbation**.
   - Configure the output signal of the Magnetic Ball Plant block as an **Open-loop Output**.

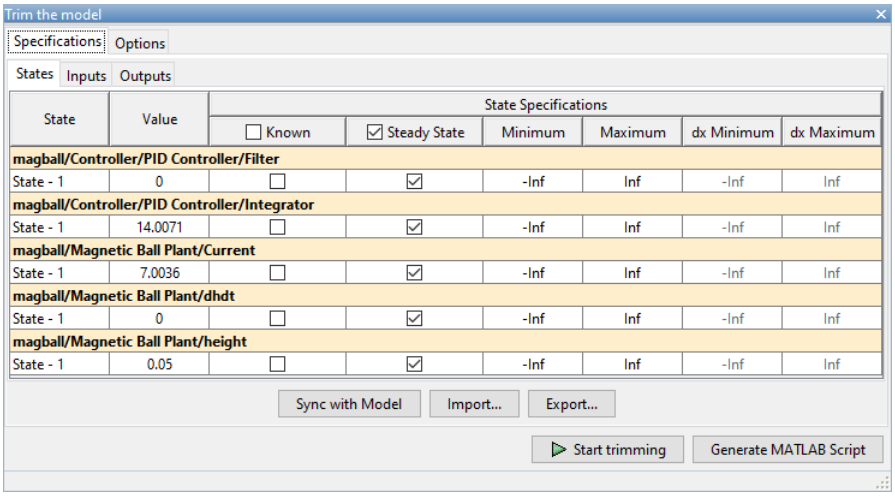
   Annotations appear in the model indicating which signals are designated as analysis points.

   Tip Alternatively, if you do not want to introduce changes to the Simulink model, you can specify the analysis points in the **Model Linearizer**. For more information, see “Specify Portion of Model to Linearize in Model Linearizer” on page 2-22.

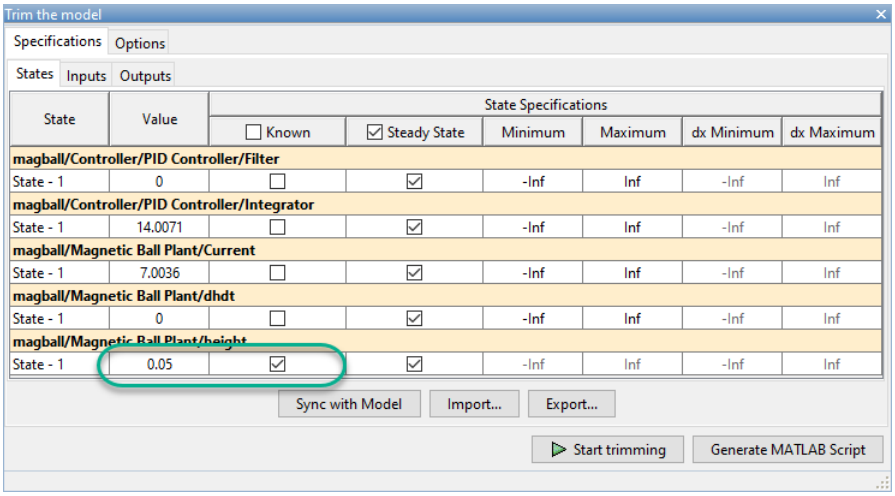
5. Create a new steady-state operating point at which to linearize the model. In the **Model Linearizer**, in the **Operating Point** drop-down list, select **Trim model**.



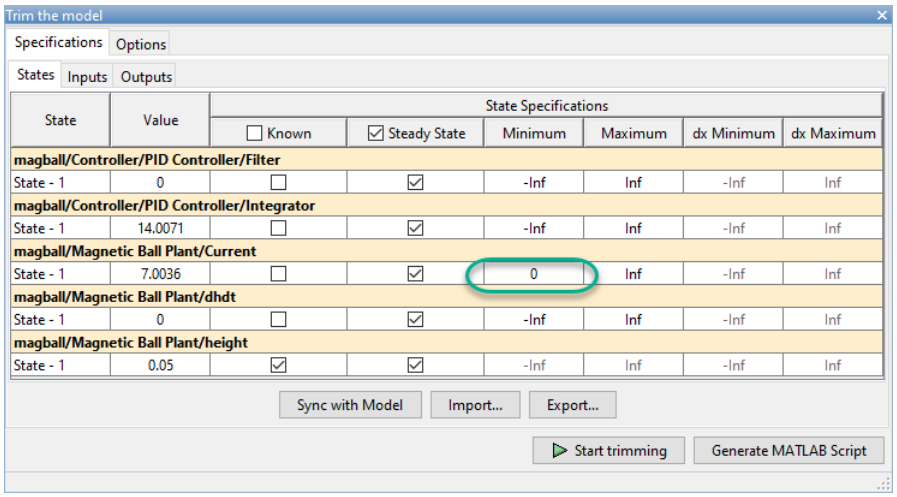
In the Trim the model dialog box, the **Specifications** tab shows the default specifications for model trimming. By default, all model states are specified to be at equilibrium, indicated by the check marks in the **Steady State** column.



6 Specify a steady-state operating point at which the magnetic ball height remains fixed at the reference signal value, 0.05. In the **States** tab, select **Known** for the **height** state. This selection tells **Model Linearizer** to find an operating point at which this state value is fixed.



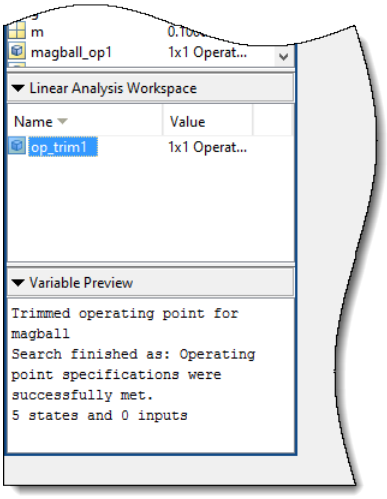
7 Since the ball height is greater than zero, the current must also be greater than zero. Enter `0` for the minimum bound of the **Current** block state.



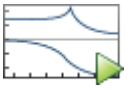
8 Compute the operating point.

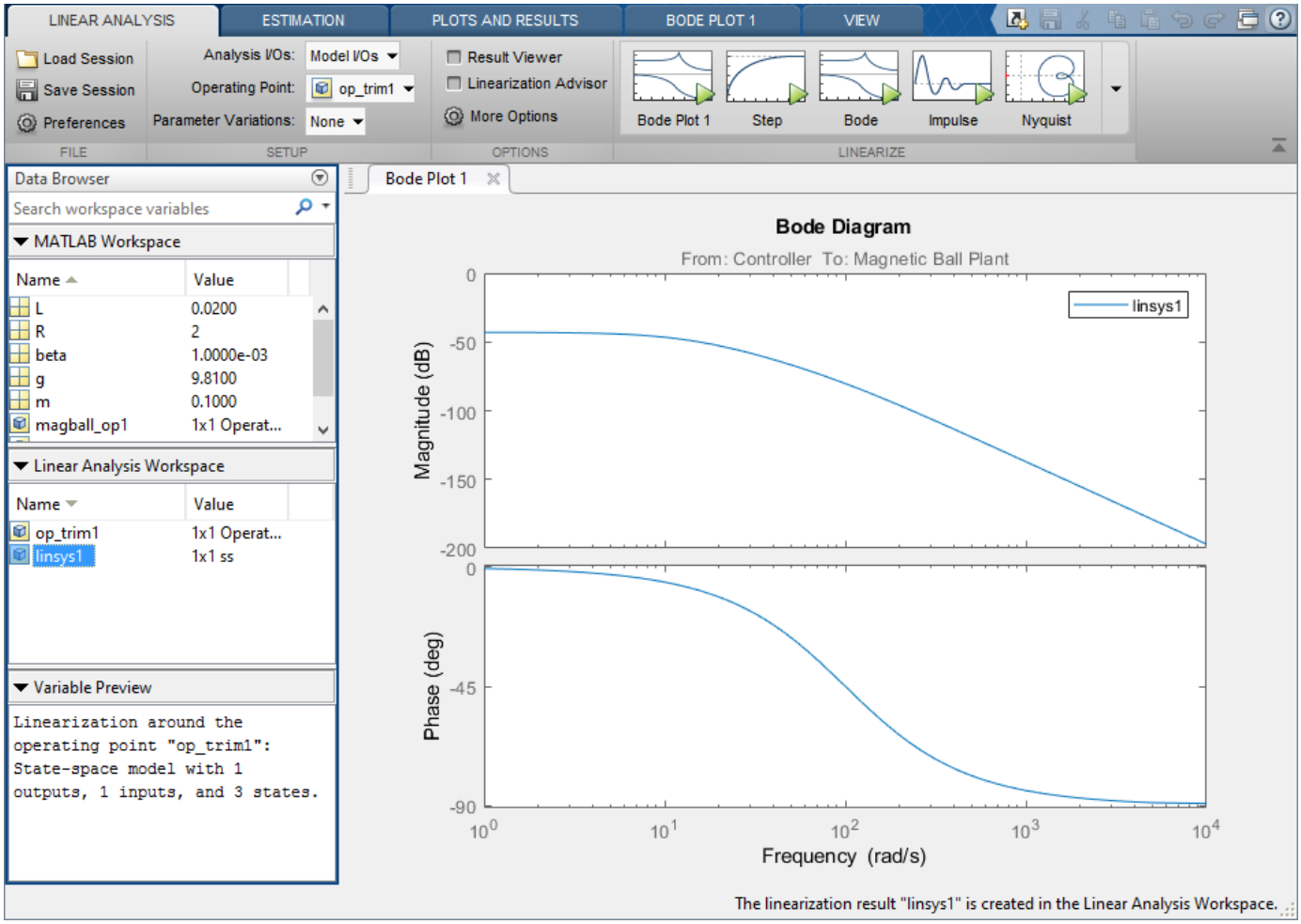
Click **Start trimming**.

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



In the **Operating Point** drop-down list, this operating point is now selected as the operating point to be used for linearization.

9 Linearize the model at the specified operating point and generate a bode plot of the result. Click  **Bode**. The Bode plot of the linearized plant appears, and the linearized plant `linsys1` appears in the Linear Analysis Workspace.



Tip Instead of a Bode plot, generate other response types by clicking the corresponding button in the plot gallery.

Right-click on the plot and select information from the **Characteristics** menu to examine characteristics of the linearized response.

See Also

More About

- “Compute Steady-State Operating Points” on page 1-5

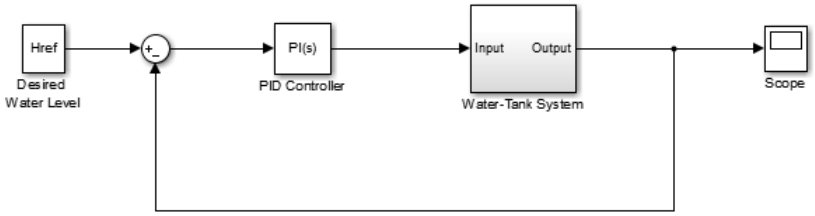
Linearize at Simulation Snapshot

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

To linearize your model at the command line, use the `linearize` function.

1. Open the Simulink model.

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



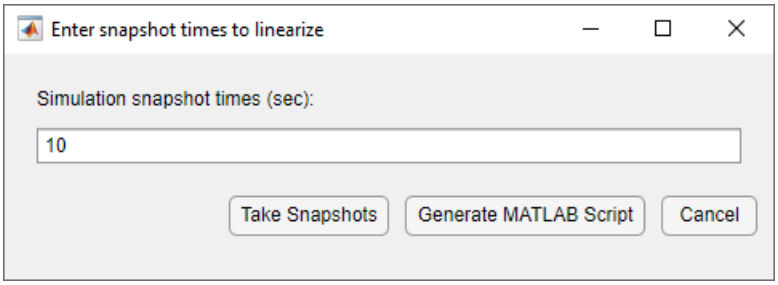
2. Open the **Model Linearizer** for the model.

   In the Simulink model window, in the **Apps** gallery, click **Model Linearizer**.

3. To specify linearization input and output points, open the **Linearization** tab. To do so, in the **Apps** gallery, click **Linearization Manager**.
4. To specify an analysis point for a signal, click the signal in the model. Then, on the **Linearization** tab, in the **Insert Analysis Points** gallery, select the type of analysis point.

   - Configure the input signal of the PID Controller block as an **Input Perturbation**.
   - Configure the output signal of the Water-Tank System block as an **Open-loop Output**.

5. Create a new simulation-snapshot operating point at which to linearize the model. In the **Model Linearizer**, in the **Operating Point** drop-down list, select **Take simulation snapshot**.
6. In the Enter snapshot times to linearize dialog box, in the **Simulation Snapshot Times** field, enter one or more snapshot times at which to linearize. For this example, enter `10` to extract the operating point at this simulation time.

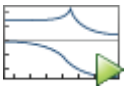


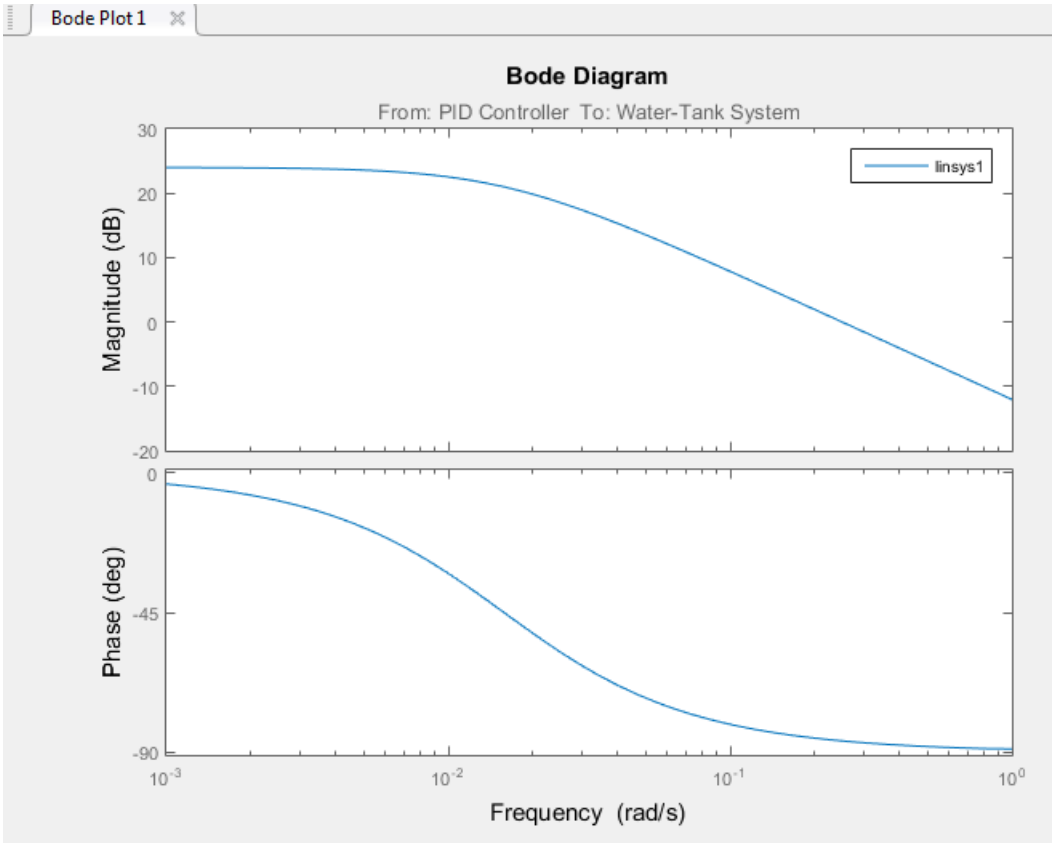
Tip To linearize the model at several operating points, specify a vector of simulation times in the **Simulation Snapshot Times** field. For example, entering `[1 10]` results in an array of two linear models, one linearized at `t = 1` and the other at `t = 10`.

7 Generate the simulation-snapshot operating point. Click **Take Snapshots**.

The operating point `op_snapshot1` appears in the Linear Analysis Workspace. In the **Operating Point** drop-down list, this operating point is now selected as the operating point to be used for linearization.

8 Linearize the model at the specified operating point and generate a bode plot of the result.

Click  **Bode**. The Bode plot of the linearized plant appears, and the linearized plant `linsys1` appears in the Linear Analysis Workspace.



9 Double click `linsys1` in the **Linear Analysis Workspace** to see the state space representation of the linear model. Right-click on the plot and select information from the **Characteristics** menu to examine characteristics of the linearized response.

10 Close Simulink model.

```
bdclose(sys);
```

See Also

More About

- “Linearize at Triggered Simulation Events” on page 2-75

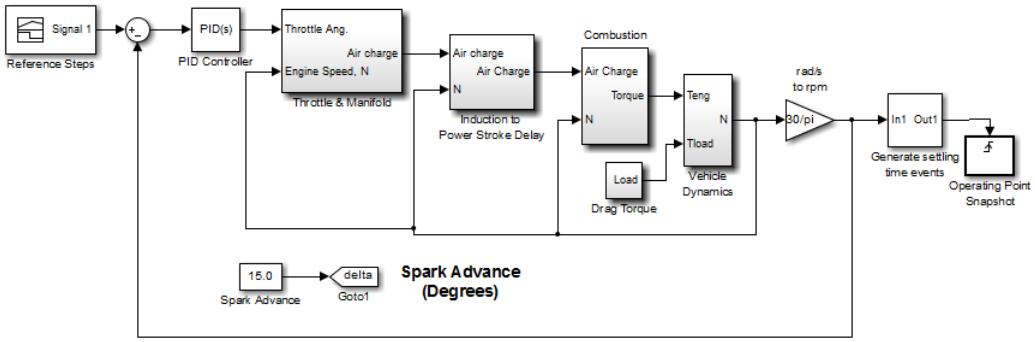
- “Visualize Linear System at Multiple Simulation Snapshots” on page 2-85
- “Visualize Linear System of a Continuous-Time Model Discretized During Simulation” on page 2-91
- “Plot Linear System Characteristics of a Chemical Reactor” on page 2-95

Linearize at Triggered Simulation Events

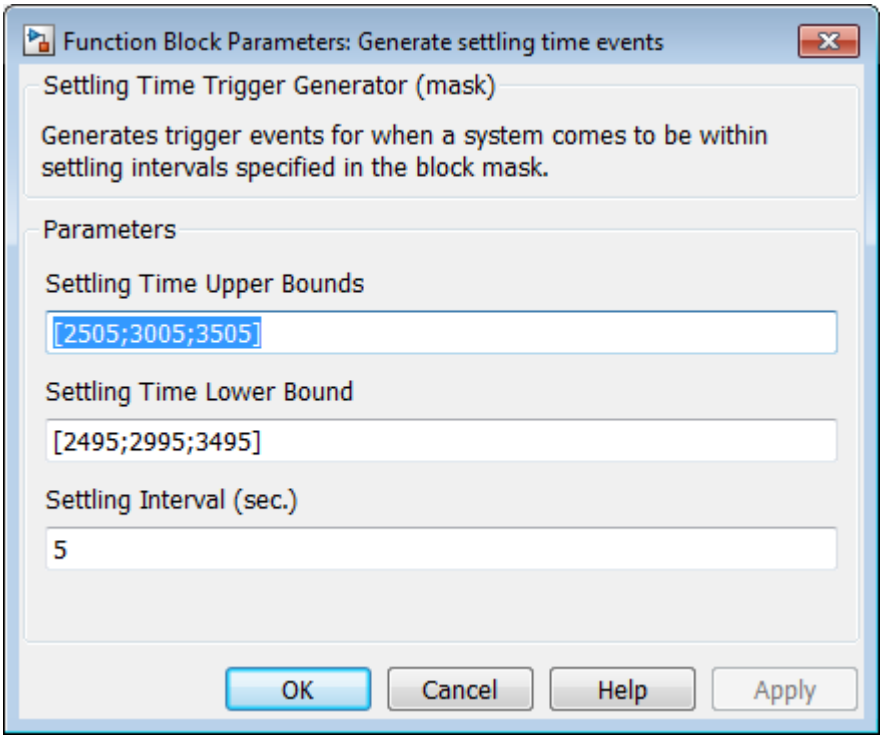
This example shows how to use linearize a Simulink model at specific events in time. Linearization events can be trigger-based events or function-call events. Specifically, the model is linearized at the steady-state operating points 2500, 3000, and 3500 rpm.

1 Open Simulink model.

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



To help identify when the system is at steady state, the Generate settling time events block generates settling events. This block sends rising edge trigger signals to the Operating Point Snapshot block when the engine speed settles near 2500, 3000, and 3500 rpm for a minimum of five seconds.



The model includes a Trigger-Based Operating Point Snapshot block. 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(mdl,60);
```

This function 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.

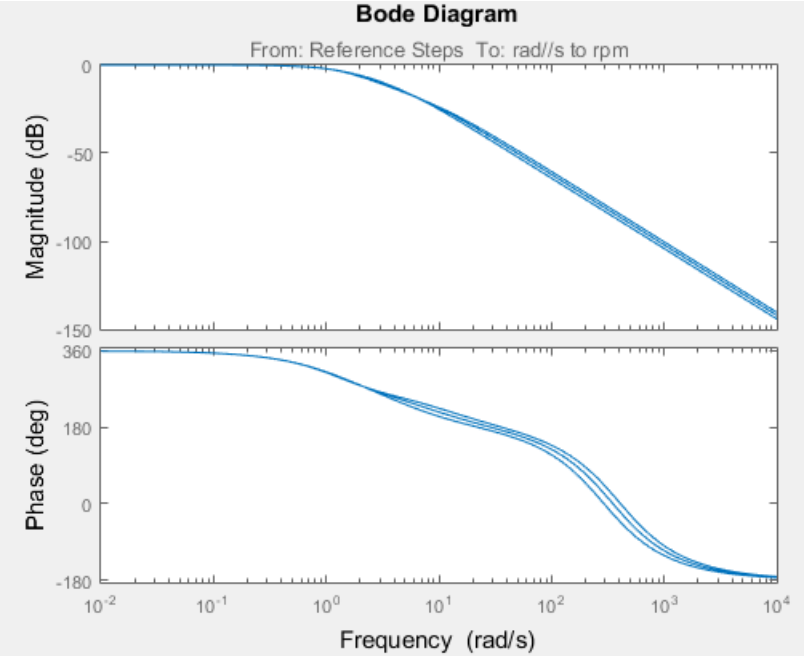
```
io(1) = linio('scdspeedtrigger/Reference Steps',1,'input');
io(2) = linio('scdspeedtrigger/rad//s to rpm',1,'output');
```

4 Linearize the model.

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

5 Compare linearized models at 2500, 3000, and 3500 rpm using Bode plots of the closed-loop transfer functions.

```
bode(linsys)
```



See Also

Functions
`findop`

Blocks
Trigger-Based Operating Point Snapshot

More About

- “Linearize at Simulation Snapshot” on page 2-72
- “Visualize Linear System at Multiple Simulation Snapshots” on page 2-85
- “Visualize Linear System of a Continuous-Time Model Discretized During Simulation” on page 2-91
- “Plot Linear System Characteristics of a Chemical Reactor” on page 2-95

Linearize Models with Delays

This example shows how to linearize a Simulink® model that contains delays.

For more information on manipulating linearized models with delays, see “Specifying Time Delays” and “Analyzing Control Systems with Delays”.

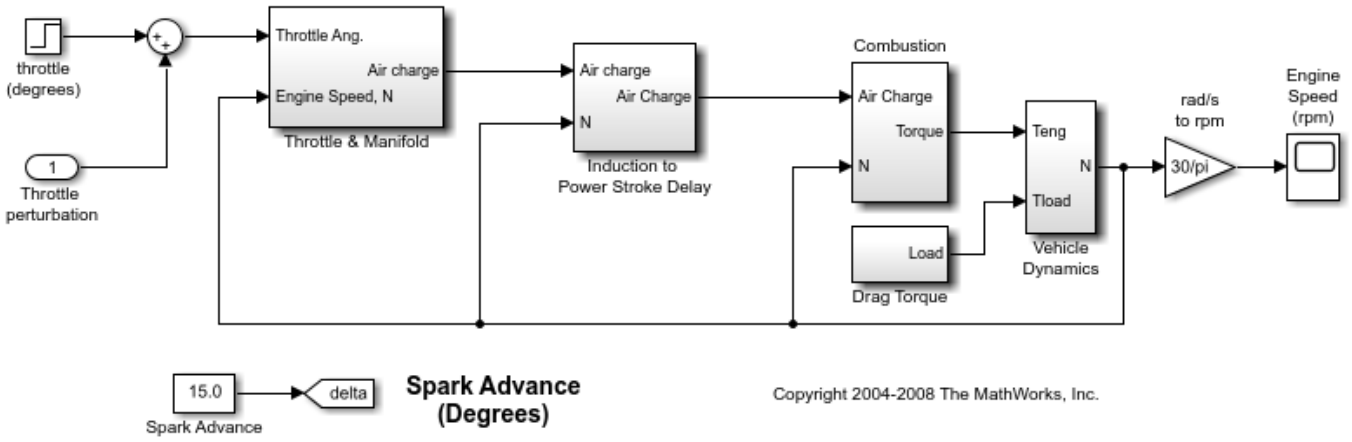
Linearize Model with Continuous Delays

You can linearize a Simulink model with continuous-time delay blocks such as the Transport Delay, Variable Transport Delay, and Variable Time Delay using one of the following options.

- Use Padé approximations of the delays to get a rational linear system through linearization. This option is the default method used by Simulink Control Design™ software.
- Compute a linearization where the delay is exactly represented. Use this option when you need accurate simulation and frequency responses from a linearized model and when assessing the accuracy of Padé approximation.

Open the engine speed model used in this example.

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



The Induction to Power Stroke Delay subsystem contains a Variable Transport Delay block named dM/dt. Specify the path to this block.

```
DelayBlock = 'scdspeed/Induction to  Power Stroke Delay/dM//dt delay';
```

To compute a linearization using a first-order approximation, set the order of the Padé approximation to 1. For the Variable Transport Delay block, **Pade Order** property to 1.

Alternatively, at the command line, enter the following code.

```
set_param(DelayBlock,'PadeOrder','1');
```

Specify the throttle angle as the linearization input and engine speed as the linearization output.

```
io(1) = linio('scdspeed/throttle (degrees)',1,'input');
io(2) = linio('scdspeed/rad//s to rpm',1,'output');
```

Linearize the model.

```
sysOrder1 = linearize(model,io);
```

To linearize the model using a second-order approximation, set the Padé order to 2.

```
set_param(DelayBlock,'PadeOrder','2');
sysOrder2 = linearize(model,io);
```

To compute a linear model with the exact delay representation, create a linearization options object and enable the `UseExactDelayModel` option.

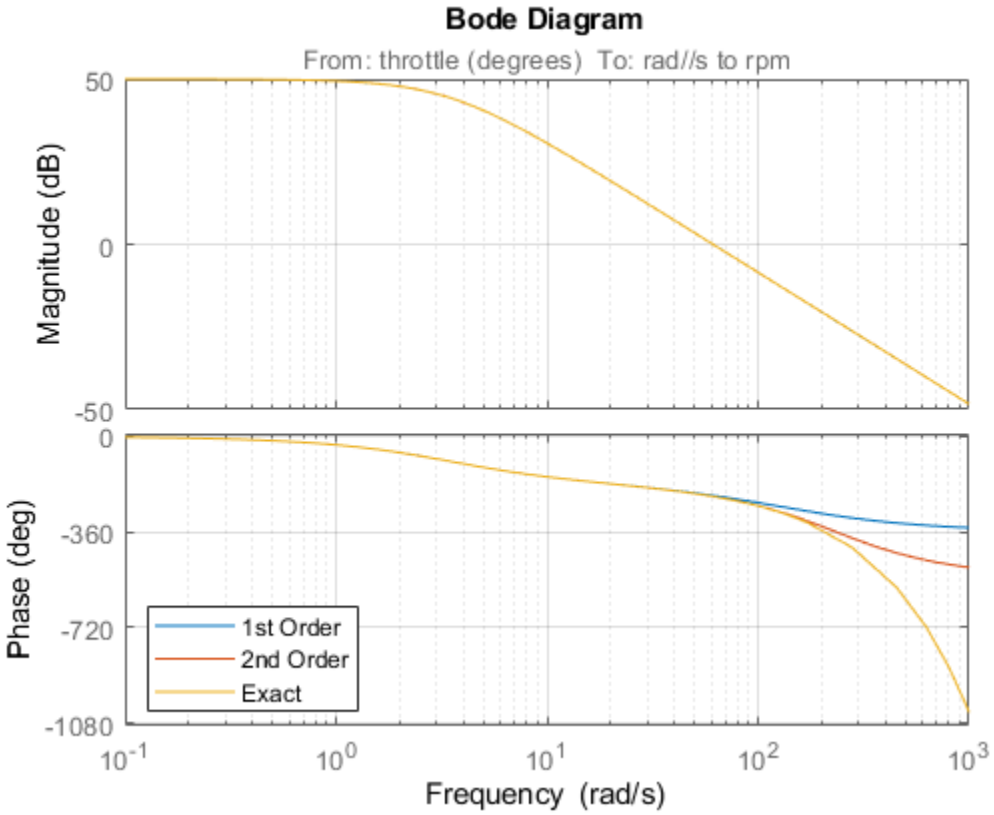
```
opt = linearizeOptions;
opt.UseExactDelayModel = 'on';
```

Linearize the model using the specified linearization options.

```
sysExactDiscrete = linearize(model,io,opt);
```

Compare the Bode response of the Padé approximation models and the exact linearization model.

```
p = bodeoptions('cstprefs');
p.Grid = 'on';
p.PhaseMatching = 'on';
p.XLimMode = {'Manual'};
p.XLim = {[0.1 1000]};
bode(sysOrder1,sysOrder2,sysExactDiscrete,p);
legend('1st Order','2nd Order','Exact','Location','SouthWest')
```



In the case of a first order approximation, the phase begins to diverge around 50 rad/s and diverges around 100 rad/s.

Close the Simulink model.

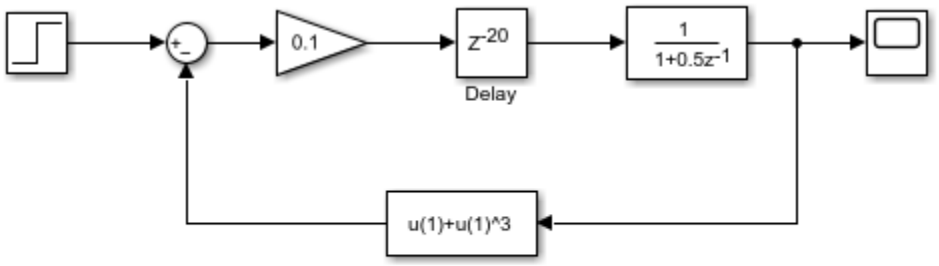
```
bdclose(model)
```

Linearize Model with Discrete Delays

When linearizing a model with discrete delay blocks, such as (Integer) Delay and Unit Delay blocks, use the exact delay option to account for the delays without adding states to the model dynamics. Explicitly accounting for these delays improves simulation performance for systems with many discrete delays because there are fewer states in your model.

Open a Simulink model of a discrete system that contains a Delay block with 20 delay states.

```
model = 'scdintegerdelay';
open_system(model)
```



By default the linearization includes all of the states folded into the linear model. Set the linearization input and output signals and linearize the model.

```
io(1) = linio('scdintegerdelay/Step',1,'input');
io(2) = linio('scdintegerdelay/Discrete Filter',1,'output');
sysDefault = linearize(model,io);
```

View the model size. It has 21 states (1 - Discrete Filter, 20 - Integer Delay).

```
size(sysDefault)
```

```
State-space model with 1 outputs, 1 inputs, and 21 states.
```

Linearize the model using exact delay representation.

```
opt = linearizeOptions;
opt.UseExactDelayModel = 'on';
sysExactDiscrete = linearize(model,io,opt);
```

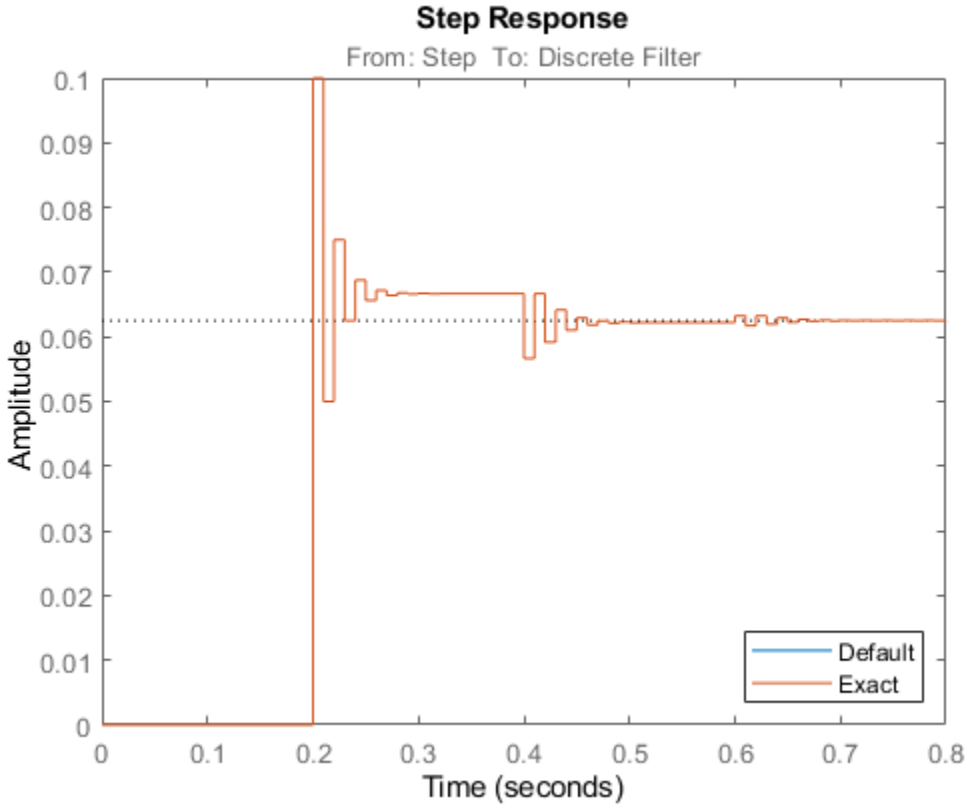
View the resulting model size. It has 1 state. The delays are accounted for internally in the linearized model.

```
size(sysExactDiscrete)
```

```
State-space model with 1 outputs, 1 inputs, and 1 states.
```

Compare the performance of the models using a step response. The models produce the same response.

```
step(sysDefault,sysExactDiscrete)
legend('Default','Exact','Location','SouthEast')
```



Close the Simulink model.

```
bdclose(model)
```

See Also

`linearize` | `linearizeOptions`

More About

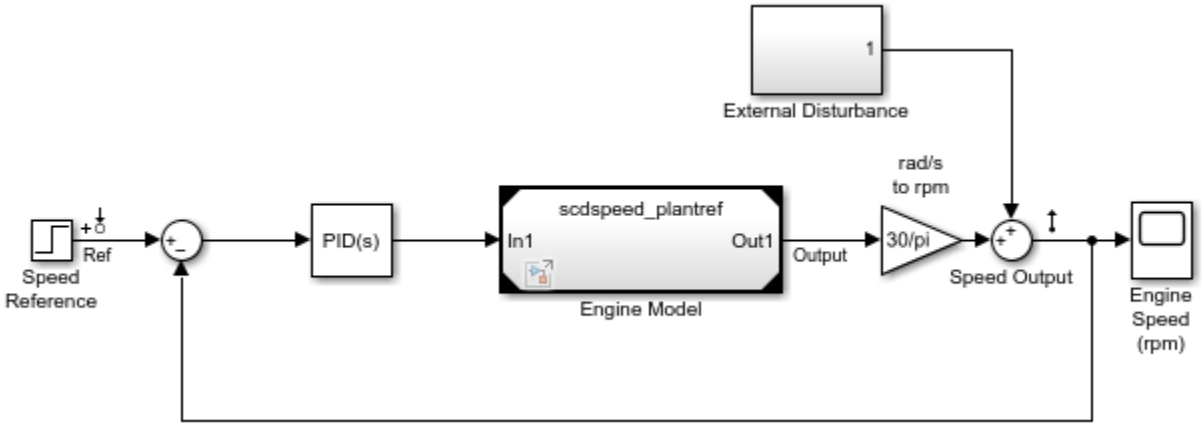
- “Models with Time Delays” on page 2-137

Linearize Models with Model References

This example shows how to linearize models that include references to other models using a Model block.

The `scdspeed_ctrlloop` model is a modified version of the `scdspeedctrl` model.

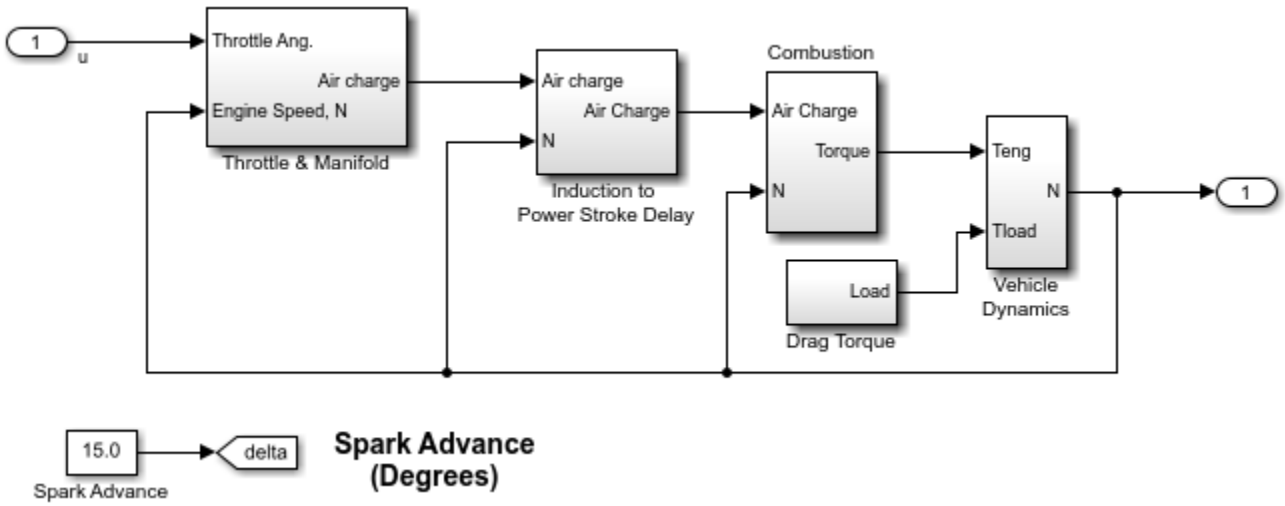
```
topmdl = 'scdspeed_ctrlloop';
open_system(topmdl)
```



For this example, the engine speed system is implemented in the `scdspeed_plantref` model. This model is referenced in the `scdspeed_ctrlloop` using a Model block.

View the referenced engine speed model.

```
open_system('scdspeed_plantref')
```



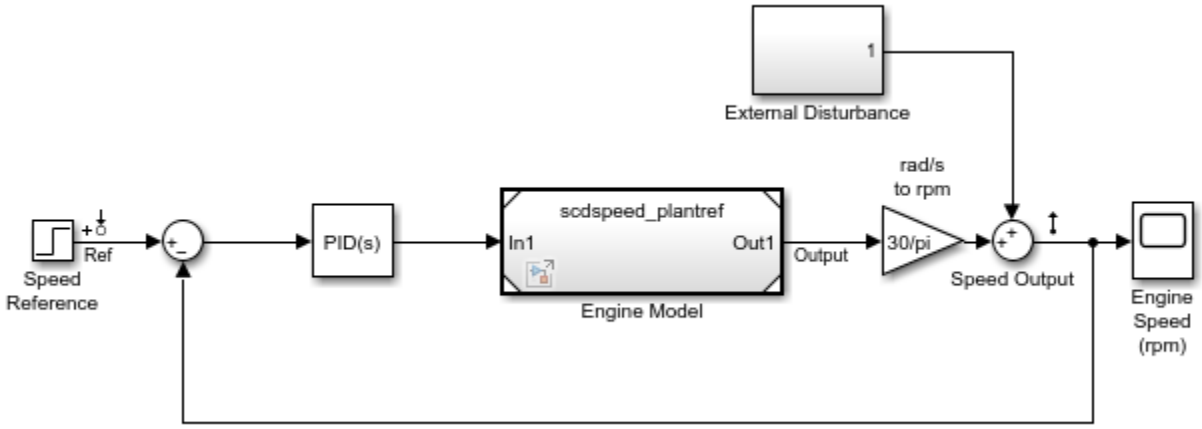
By default, the Engine Model block is set to accelerator simulation mode, as indicated by the black triangles at the corners of the Model block. Linearizing the model with this block in accelerator mode

numerically perturbs the entire Engine Model block. The accuracy of this linearization is sensitive to the blocks within the Engine model. In particular, the variable transport delay block is problematic.

To achieve an accurate linearization, set the Model block to normal simulation mode, which enables the block-by-block linearization of the referenced model.

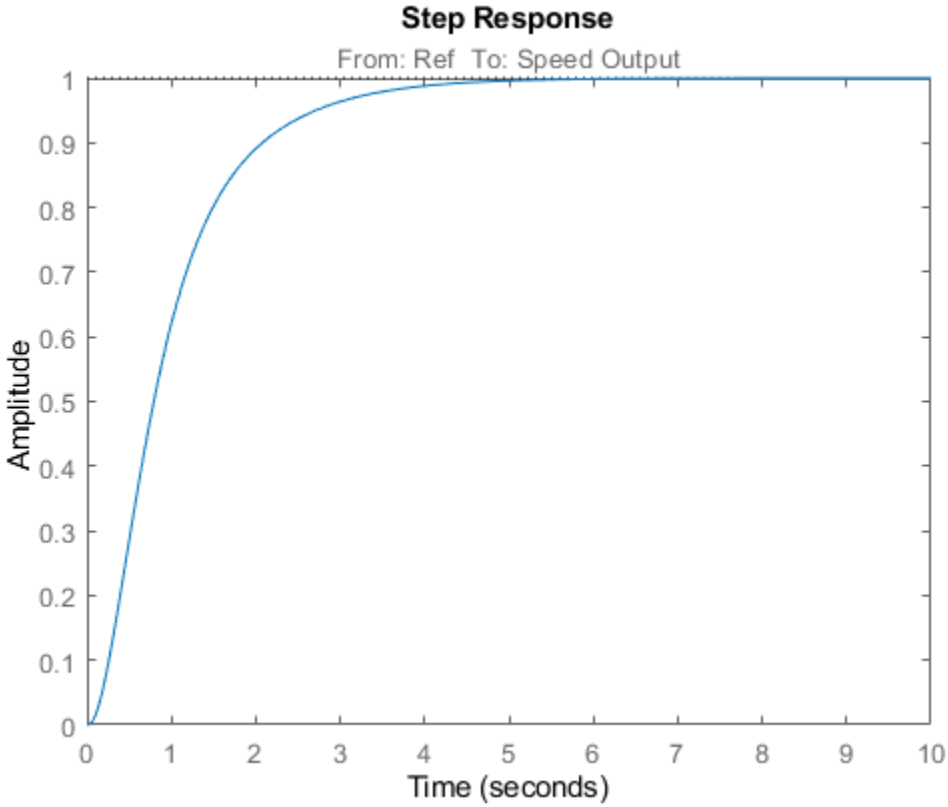
```
set_param('scdspeed_ctrlloop/Engine Model','SimulationMode','Normal')
```

The corners of the Engine Model block are now white triangles, indicating that its simulation mode is set to normal.



Linearize the model between the speed reference signal and the speed output, and plot the resulting step response.

```
io(1) = linio('scdspeed_ctrlloop/Speed Reference',1,'input');
io(2) = linio('scdspeed_ctrlloop/Speed Output',1,'output');
sys_normal = linearize(topmdl,io);
step(sys_normal)
```



Another benefit of switching the model reference to normal mode simulation is that you can take advantage of exact delay representations. For more information on linearizing models with delays, see “Linearize Models with Delays” on page 2-78.

Close the Simulink® model.

```
bdclose('scdspeed_ctrlloop')
```

See Also

`linearize`

More About

- “Linearize Plant” on page 2-33
- “Linearize Models with Delays” on page 2-78

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.

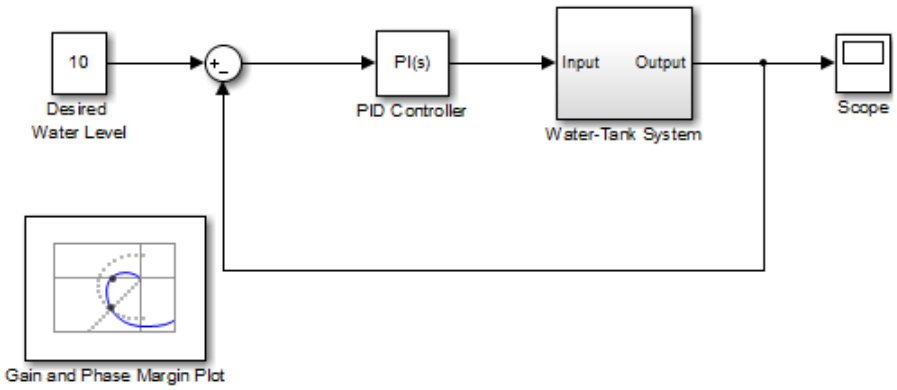
1. Open Simulink model.

   For example:

   ```
   watertank
   ```

2. Open the Simulink Library Browser. In the Simulink Editor, on the **Simulation** tab, click **Library Browser**.
3. Add a plot block to the Simulink model.
   a. In the **Simulink Control Design** library, select **Linear Analysis Plots**.
   b. Drag and drop a block, such as the Gain and Phase Margin Plot block, into the Simulink model window.

      The model now resembles the following figure.



4. Double-click the block to open the Block Parameters dialog box.

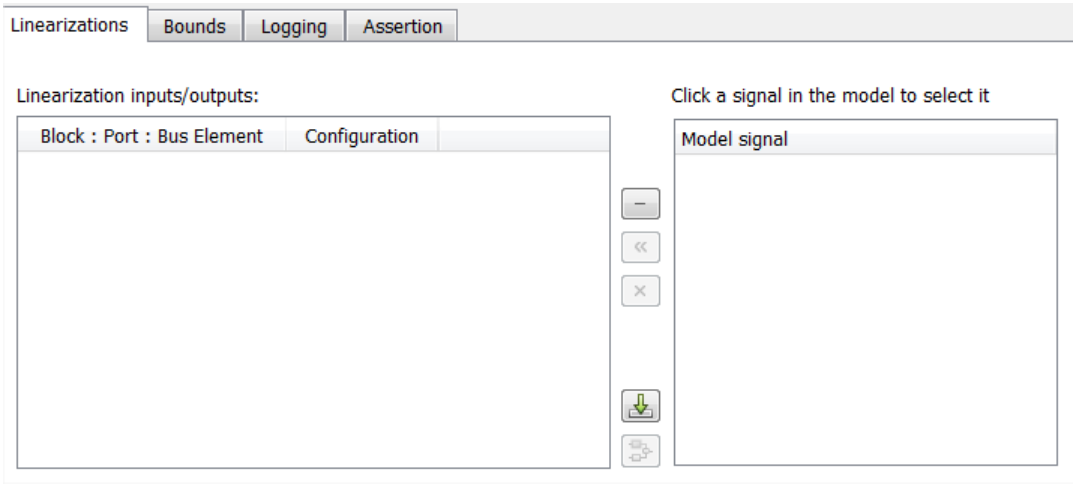
   To learn more about the block parameters, see the block reference pages.
5. Specify the linearization I/O points.

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

   Tip If your model already contains I/O points, the block automatically detects these points and displays them. Click at any time to update the **Linearization inputs/outputs** table with I/Os from the model.

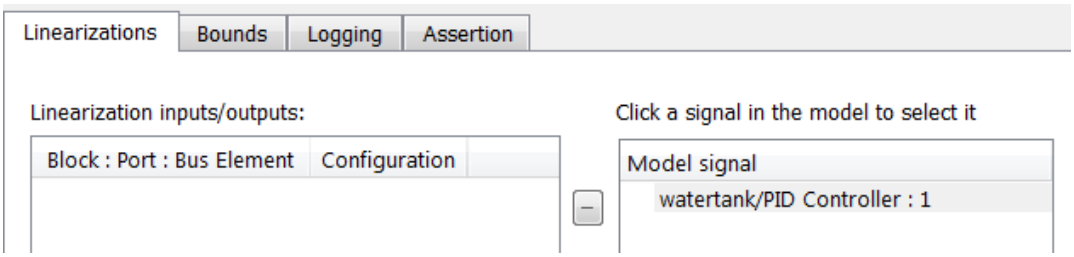
   a. To specify an input:
      i. Click adjacent to the **Linearization inputs/outputs** table.

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



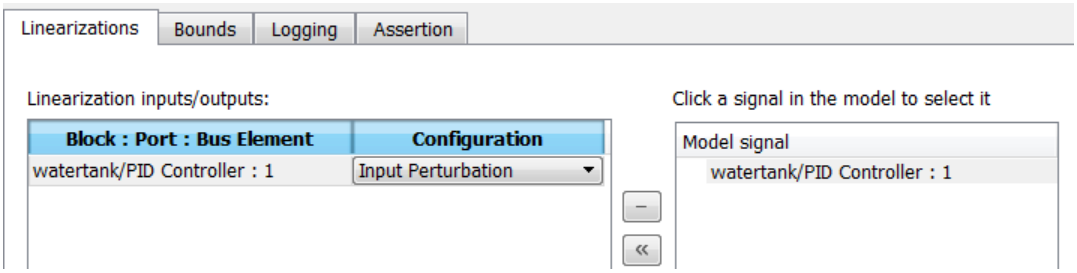
ii In the Simulink model, click the output signal of the `PID Controller` block to select it.

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



Tip You can select multiple signals at once in the Simulink model. All selected signals appear in the **Click a signal in the model to select it** area.

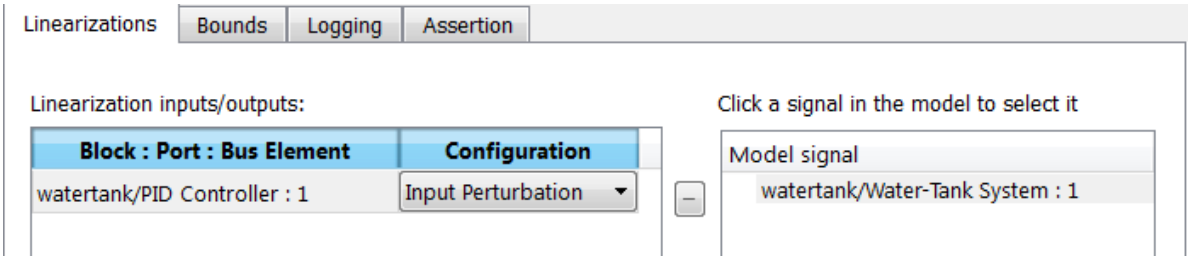
iii Click « to add the signal to the **Linearization inputs/outputs** table.



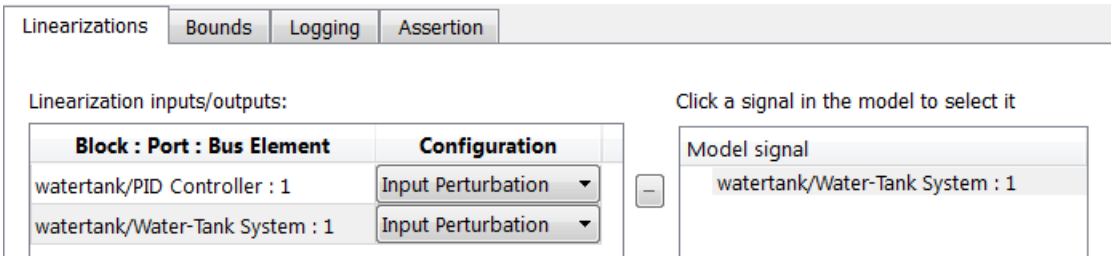
b To specify an output:

i In the Simulink model, click the output signal of the `Water-Tank System` block to select it.

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

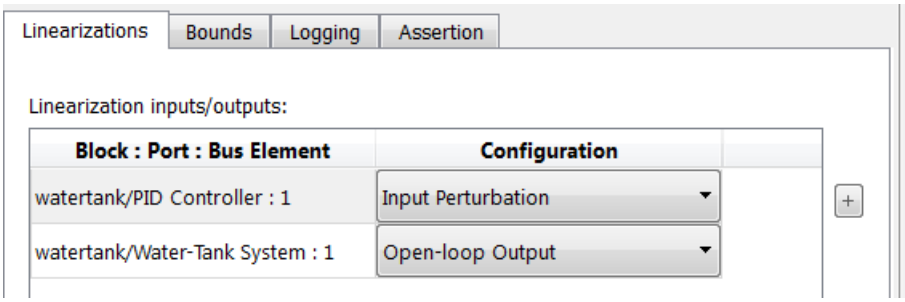


ii Click « to add the signal to the **Linearization inputs/outputs** table.



iii In the **Configuration** drop-down list of the **Linearization inputs/outputs** table, select `Open-loop Output` for **watertank/Water-Tank System : 1**.

The **Linearization inputs/outputs** table now resembles the following figure.



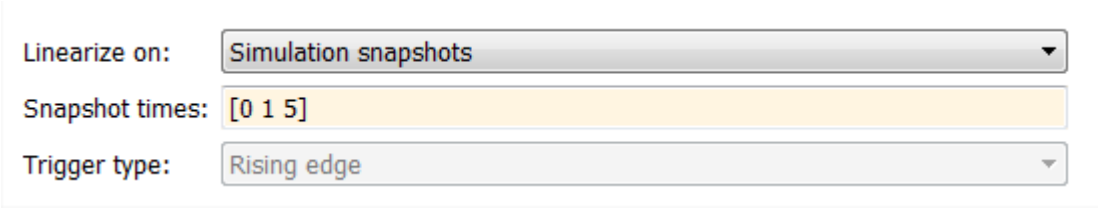
c Click − to collapse the **Click a signal in the model to select it** area.

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

6 Specify simulation snapshot times.

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

b In the **Snapshot times** field, type `[0 1 5]`.



7 Specify a plot type to plot the gain and phase margins. The plot type is `Bode` by default.

   a Select `Nichols` in **Plot type**

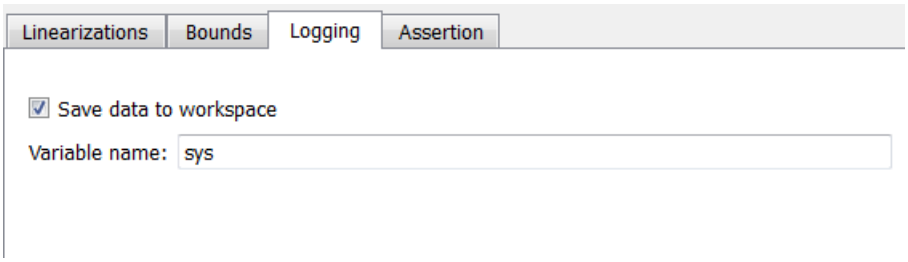
   b Click **Show Plot** to open an empty Nichols plot.

8 Save the linear system.

   a Select the **Logging** tab.

   b Select the **Save data to workspace** option and specify a variable name in the **Variable name** field.

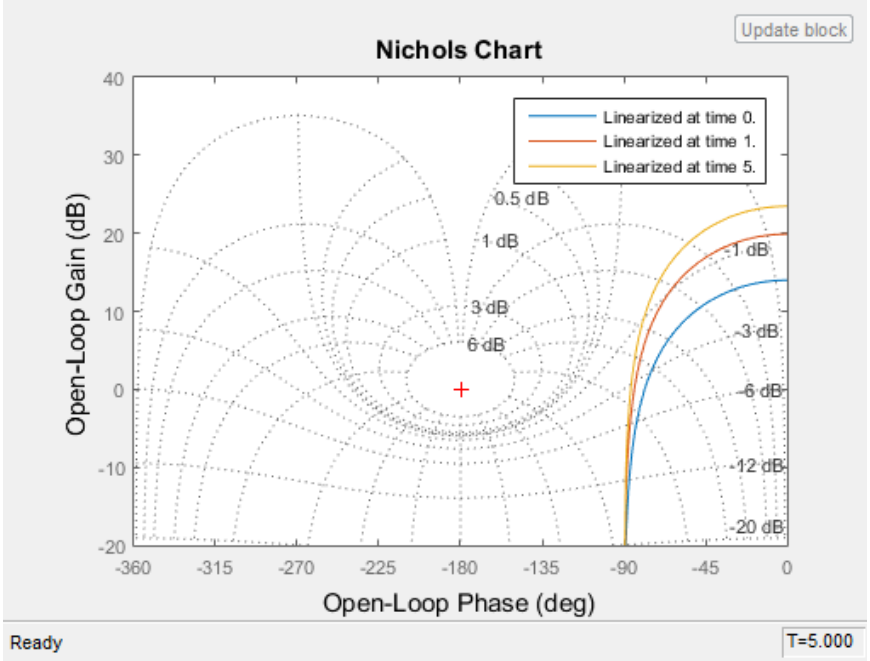
   The **Logging** tab now resembles the following figure.



9 Plot the gain and phase margins by clicking in the plot window.

The software linearizes the portion of the model between the linearization input and output at the simulation times of 0, 1 and 5 and plots gain and phase margins.

After the simulation completes, the plot window resembles the following figure.



Tip Click to view the legend.

The computed linear system is saved as `sys` in the MATLAB workspace. `sys` is a structure with `time` and `values` fields. To view the structure, type:

```
sys
```

This command returns the following results:

```
sys =

         time: [3x1 double]
       values: [4-D ss]
    blockName: 'watertank/Gain and Phase Margin Plot'
```

- The `time` field contains the simulation times at which the model is linearized.
- The `values` field is an array of state-space objects 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.)

See Also

Bode Plot | Gain and Phase Margin Plot | Linear Step Response Plot | Nichols Plot | Pole-Zero Plot | Singular Value Plot

More About

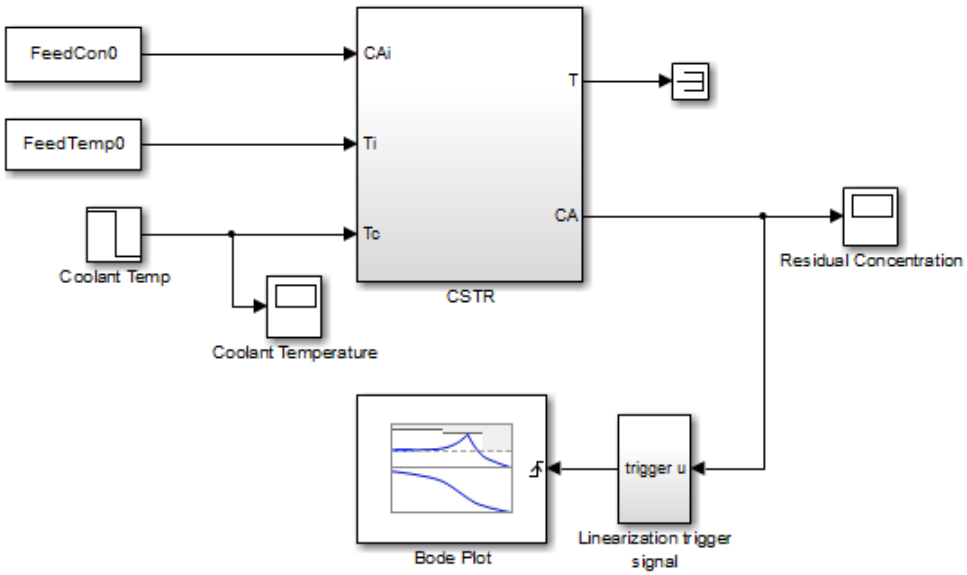
- “Visualize Bode Response of Simulink Model During Simulation” on page 2-61
- “Plot Linear System Characteristics of a Chemical Reactor” on page 2-95
- “Visualize Linear System of a Continuous-Time Model Discretized During Simulation” on page 2-91
- “Linearize at Simulation Snapshot” on page 2-72
- “Linearize at Triggered Simulation Events” on page 2-75

Visualize Linear System of a Continuous-Time Model Discretized During Simulation

This example shows how to discretize a continuous-time model during simulation and plot the model's discretized linear behavior.

1 Open the Simulink model:

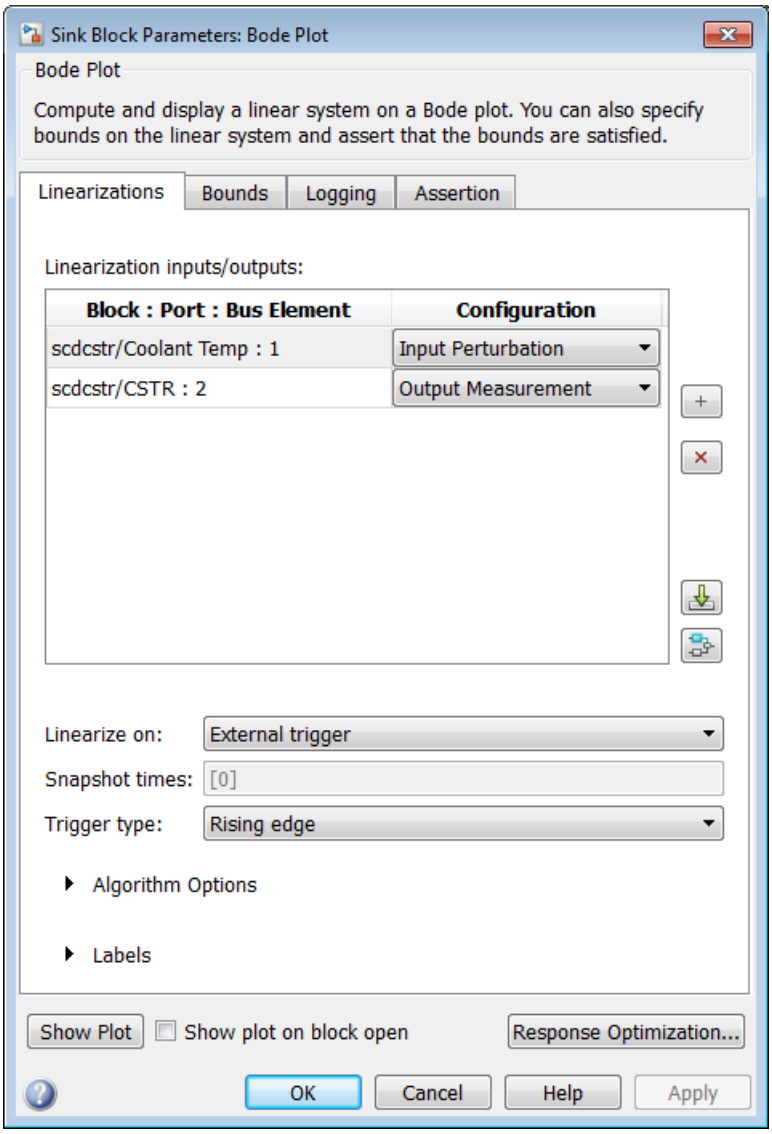
```
scdcstr
```



In this model, the Bode Plot block has already been configured with:

- Input point at the coolant temperature input `Coolant Temp`
- Output point at the residual concentration output `CA`
- Settings to linearize the model on a rising edge of an external trigger. The trigger signal is modeled in the `Linearization trigger signal` block in the model.
- Saving the computed linear system in the MATLAB workspace as `LinearReactor`.

To view these configurations, double-click the block.

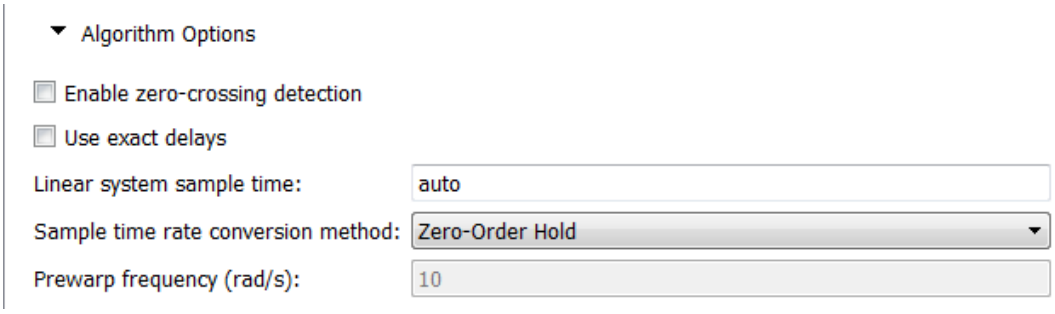


To learn more about the block parameters, see the block reference pages.

2 Specify the sample time to compute the discrete-time linear system.

a Click ▸ adjacent to **Algorithm Options**.

The option expands to display the linearization algorithm options.



b Specify a sample time of 2 in the **Linear system sample time** field.

To learn more about this option, see the block reference page.

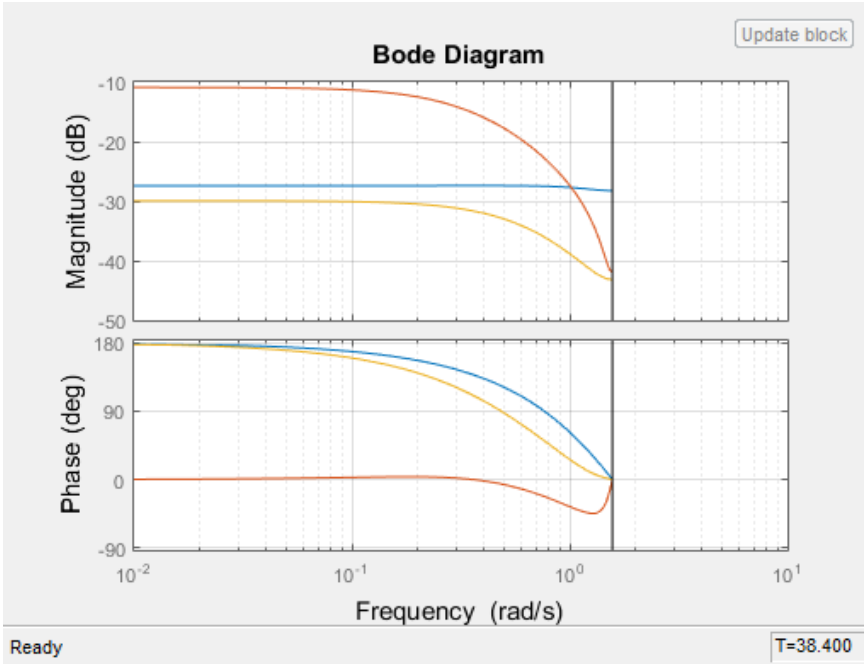
3 Click **Show Plot** to open an empty Bode plot window.

4 Plot the Bode magnitude and phase by clicking in the plot window.

During simulation, the software:

- Linearizes the model on encountering a rising edge.
- Converts the continuous-time model into a discrete-time linear model with a sample time of 2. This conversion uses the default `Zero-Order Hold` method to perform the sample time conversion.

  The software plots the discrete-time linear behavior in the Bode plot window. After the simulation completes, the plot window resembles the following figure.



The plot shows the Bode magnitude and phase up to the Nyquist frequency, which is computed using the specified sample time. The vertical line on the plot represents the Nyquist frequency.

See Also

Bode Plot | Gain and Phase Margin Plot | Linear Step Response Plot | Nichols Plot | Pole-Zero Plot | Singular Value Plot

More About

- “Visualize Bode Response of Simulink Model During Simulation” on page 2-61
- “Visualize Linear System at Multiple Simulation Snapshots” on page 2-85
- “Plot Linear System Characteristics of a Chemical Reactor” on page 2-95
- “Linearize at Simulation Snapshot” on page 2-72
- “Linearize at Triggered Simulation Events” on page 2-75

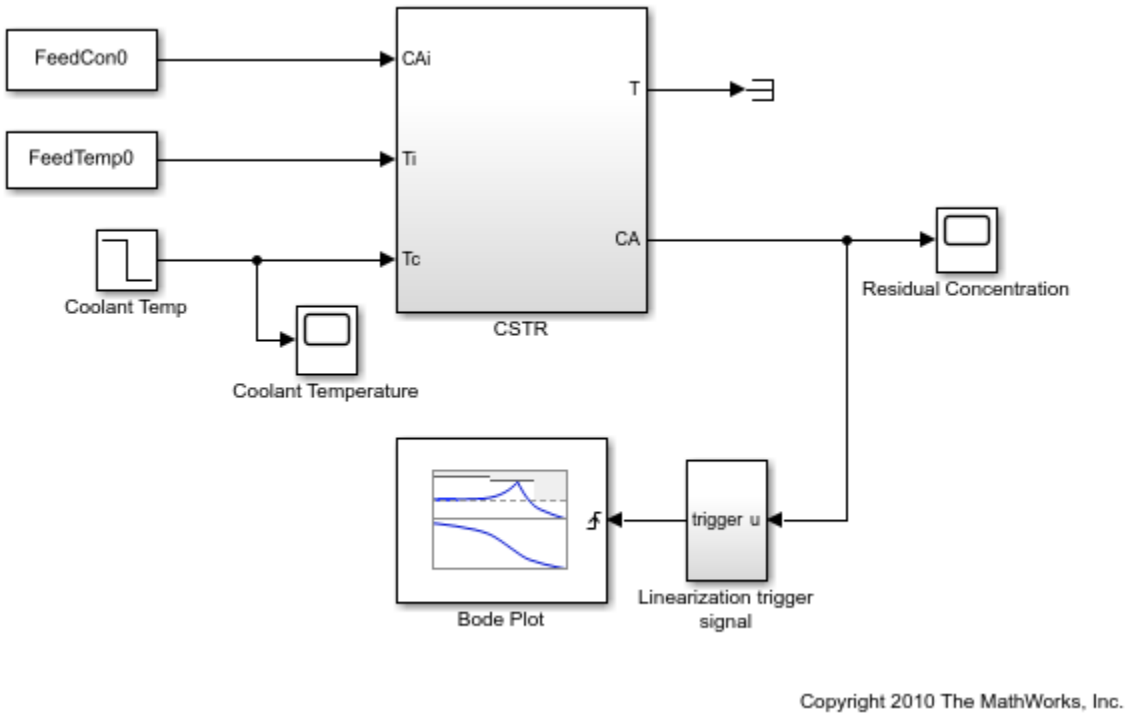
Plot Linear System Characteristics of a Chemical Reactor

This example shows how to plot the linearization of a Simulink® model at particular conditions during simulation. Simulink Control Design™ software provides blocks that you can add to Simulink models to compute and plot linear systems during simulation. In this example, you compute and plot a linearized model for a continuously stirred chemical reactor. Specifically, you linearize the reactor model as it transitions through different operating points.

Chemical Reactor Model

Open the Simulink model of the chemical reactor.

```
open_system('scdcstr')
```



The reactor has three inputs and two outputs.

- The `FeedCon0`, `FeedTemp0`, and `Coolant Temp` blocks model the feed concentration, feed temperature, and coolant temperature inputs, respectively.
- The `T` and `CA` ports of the `CSTR` block model the reactor temperature and residual concentration outputs, respectively.

This example focuses on the response from coolant temperature, `Coolant Temp`, to residual concentration, `CA`, when the feed concentration and feed temperature are constant.

For more information on modeling reactors, see [1].

Plot the Reactor Linear Response

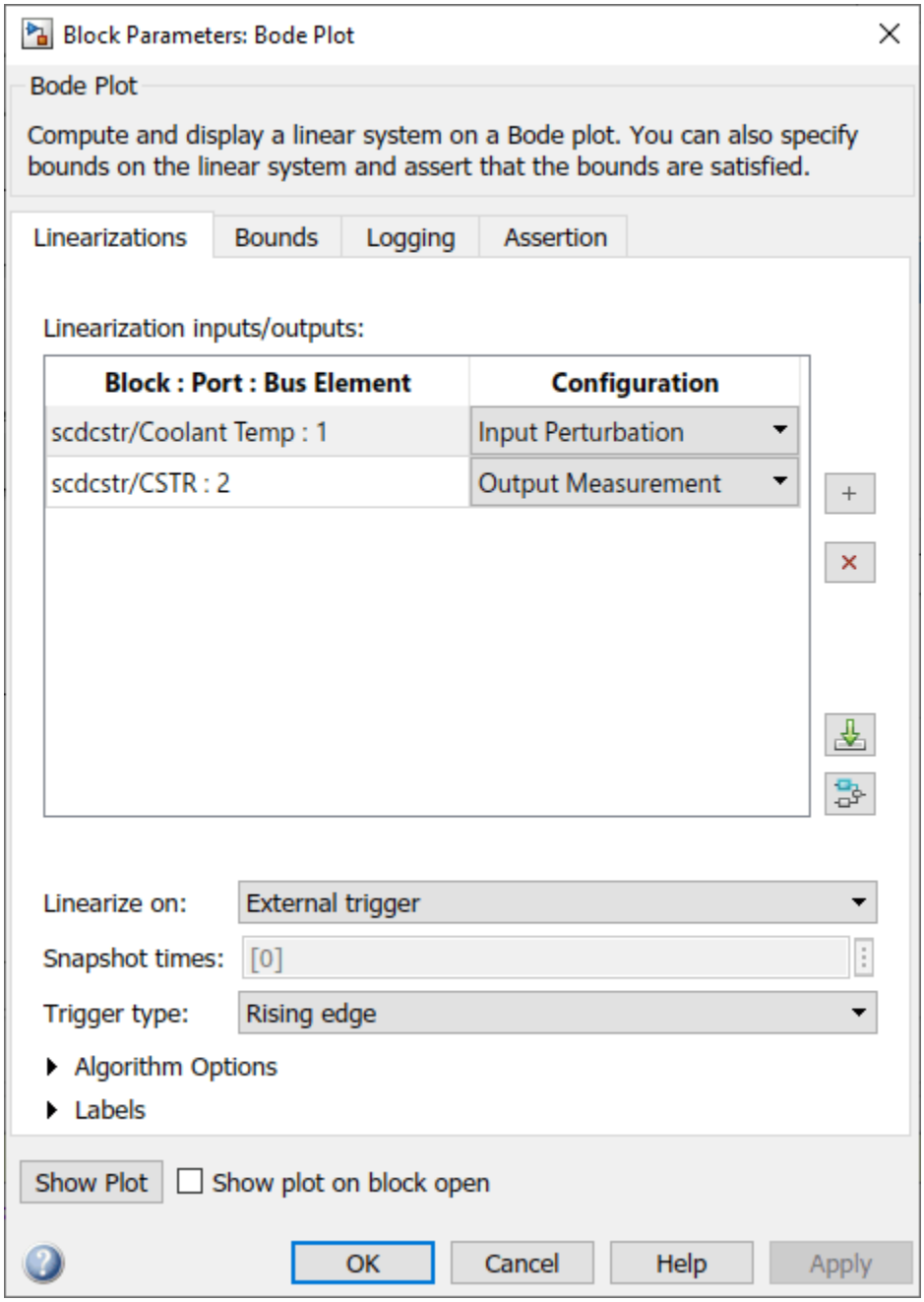
The reactor model contains a Bode Plot block from the Simulink Control Design Linear Analysis Plots library. The block is configured with:

- A linearization input at the coolant temperature `Coolant Temp`
- A linearization output at the residual concentration `CA`

The Bode Plot block is configured to perform linearizations on the rising edges of an external trigger signal. The Linearization trigger signal block computes the trigger signal and produces a rising edge when the residual concentration satisfies any one of the following conditions.

- At a steady state value of 2
- In a narrow range around 5
- At a steady state value of 9

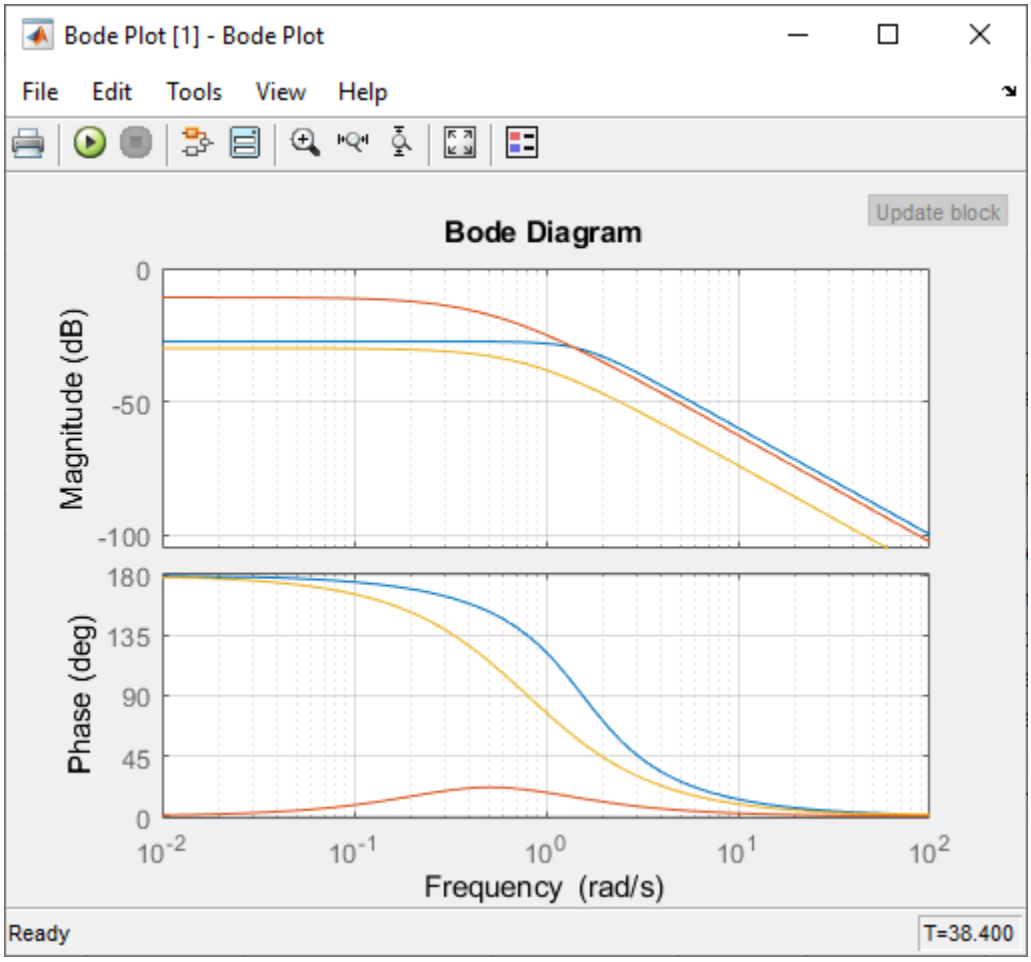
To view the Bode Plot block configuration, double-click the block.



To view the Bode plot for the block during the simulation, you must open the plot window before simulating the model. To do so, click **Show Plot**.

Simulate the model.

```
sim('scdcstr')
```



The Bode plot shows the linearized reactor at three operating points corresponding to the trigger signals generated by the Linearization trigger signal block.

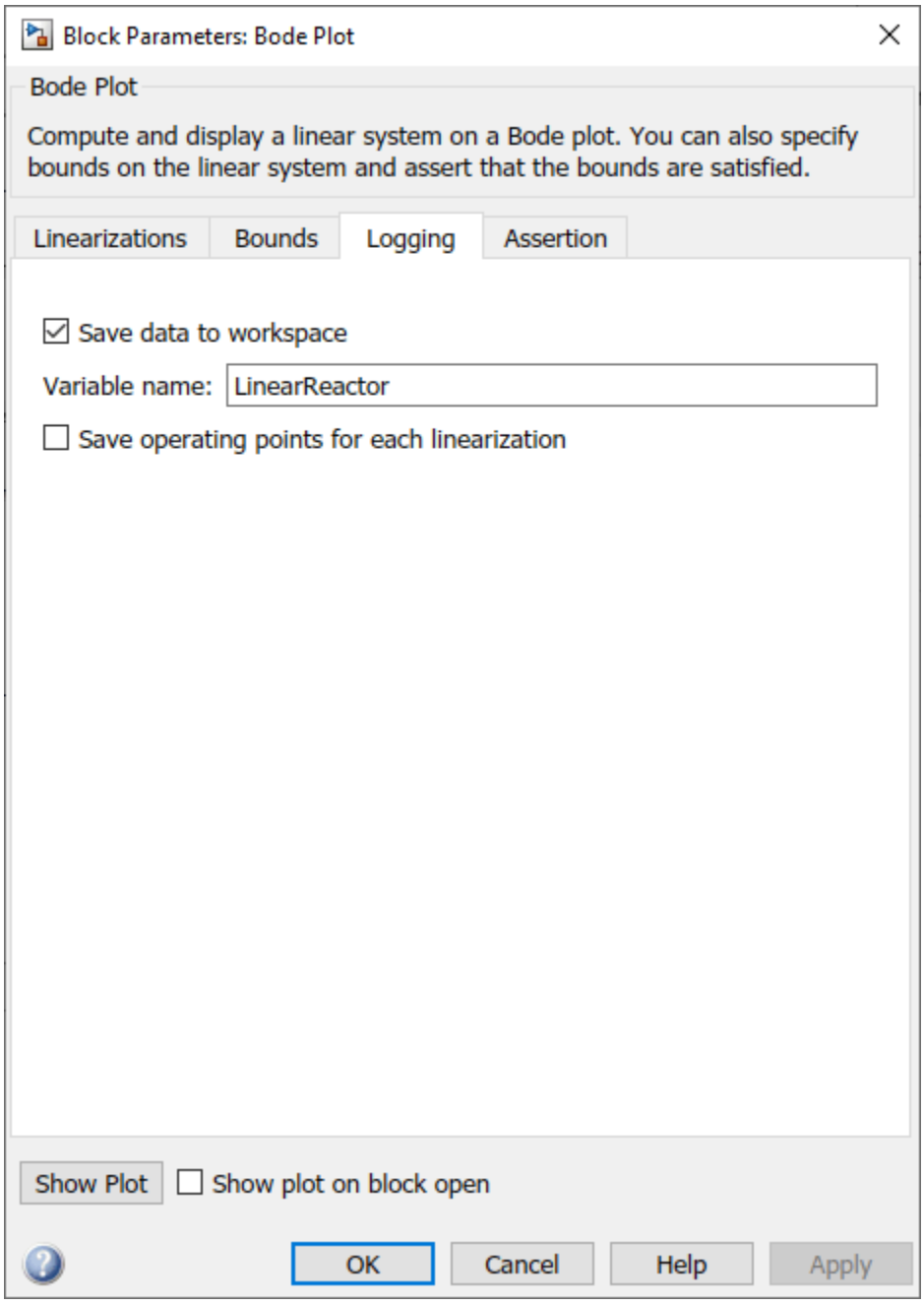
- At 5 sec, the linearization is for a low residual concentration.
- At 38 sec, the linearization is for a high residual concentration.
- At 27 sec, the linearization is as the reactor transitions from a low to high residual concentration.

The linearizations at low and high residual concentrations are similar, but the linearization during the transition has a significantly different DC gain and phase characteristics. At low frequencies, the phase differs by 180 degrees, indicating the presence of either an unstable pole or zero.

Log the Reactor Linear Response

You can save the computed linear systems to the model workspace. To do so, use the parameters on the **Logging** tab of the block.

For this example, the Bode Plot block is configured to save the computed responses in the MATLAB® workspace as a `LinearReactor` structure.




```
LinearReactor
```

```
LinearReactor =

  struct with fields:

         time: [3x1 double]
       values: [1x1x3x1 ss]
    blockName: 'scdcstr/Bode Plot'
```

In this structure, the `values` field stores the linear systems as an array of LTI state-space models. For more information, see “Model Arrays”. The corresponding simulation times for the linearizations are logged in the `time` field.

Obtain the computed linear models.

```
P1 = LinearReactor.values(:,:,1);
P2 = LinearReactor.values(:,:,2);
P3 = LinearReactor.values(:,:,3);
```

The Bode plot of the linear system at a time of 27 seconds, when the reactor transitions from low to high residual concentration, indicates that the system could be unstable. To confirm this result, displaying the linear systems in zero-pole-gain format.

```
zpk(P1)
zpk(P2)
zpk(P3)
```

```
ans =

  From input "Coolant Temp" to output "CSTR/2":
         -0.1028
  ----------------------
  (s^2 + 2.215s + 2.415)

Continuous-time zero/pole/gain model.


ans =

  From input "Coolant Temp" to output "CSTR/2":
        -0.07514
  ---------------------
  (s+0.7567) (s-0.3484)

Continuous-time zero/pole/gain model.


ans =

  From input "Coolant Temp" to output "CSTR/2":
        -0.020462
  --------------------
  (s+0.8542) (s+0.7528)

Continuous-time zero/pole/gain model.
```

Close the Simulink model.

```
bdclose('scdcstr')
```

Reference

[1] Seborg, Dale, Thomas Edgar, and Duncan Mellichamp. *Process Dynamics and Control, Second Edition*. John Wiley & Sons, Ltd, 2006.

See Also

Bode Plot | Gain and Phase Margin Plot | Linear Step Response Plot | Nichols Plot | Pole-Zero Plot | Singular Value Plot

More About

- “Visualize Bode Response of Simulink Model During Simulation” on page 2-61
- “Visualize Linear System at Multiple Simulation Snapshots” on page 2-85
- “Visualize Linear System of a Continuous-Time Model Discretized During Simulation” on page 2-91
- “Plot Linear System Characteristics of a Chemical Reactor” on page 2-95

Order States in Linearized Model

Control State Order of Linearized Model using Model Linearizer

This example shows how to control the order of the states in your linearized model. This state order appears in linearization results.

1. Open and configure the model for linearization by specifying linearization I/Os and an operating point for linearization. You can perform this step as shown, for example, in “Linearize at Trimmed Operating Point” on page 2-67. To preconfigure the model at the command line, use the following commands.

   ```
   sys = 'magball';
   open_system(sys)
   sys_io(1) = linio('magball/Controller',1,'input');
   sys_io(2) = linio('magball/Magnetic Ball Plant',1,'openoutput');
   setlinio(sys,sys_io)
   opspec = operspec(sys);
   op = findop(sys,opspec);
   ```

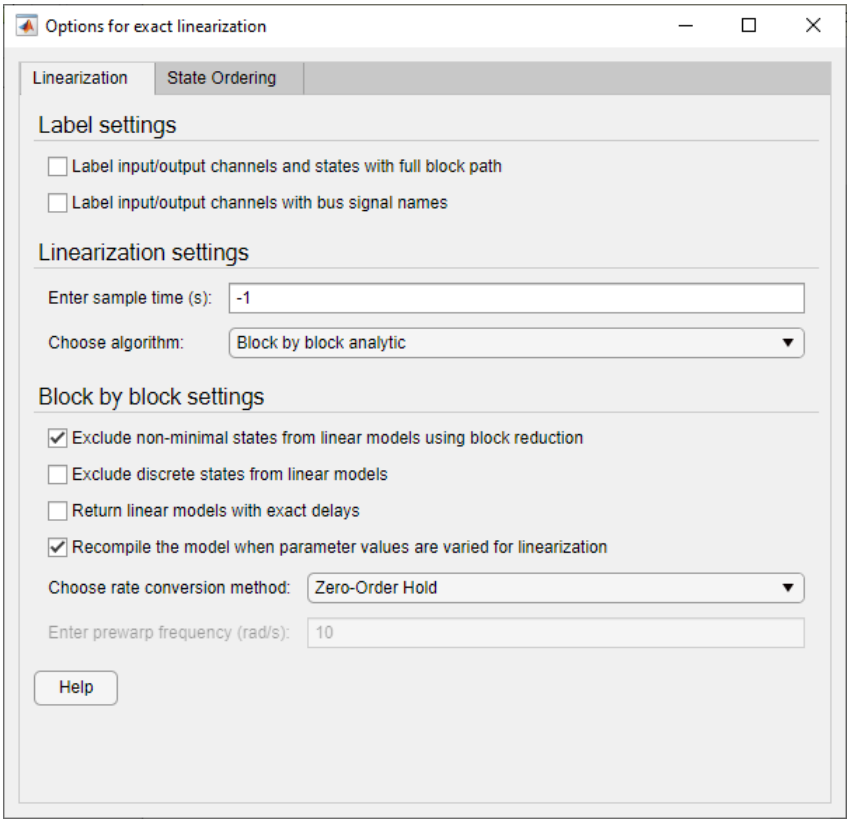
   These commands specify linear analysis points at the input and output of the plant and compute the steady-state operating point.

2. Open the **Model Linearizer** for the model.

   In the Simulink model window, in the **Apps** gallery, click **Model Linearizer**.

3. Open the Options for exact linearization dialog box.

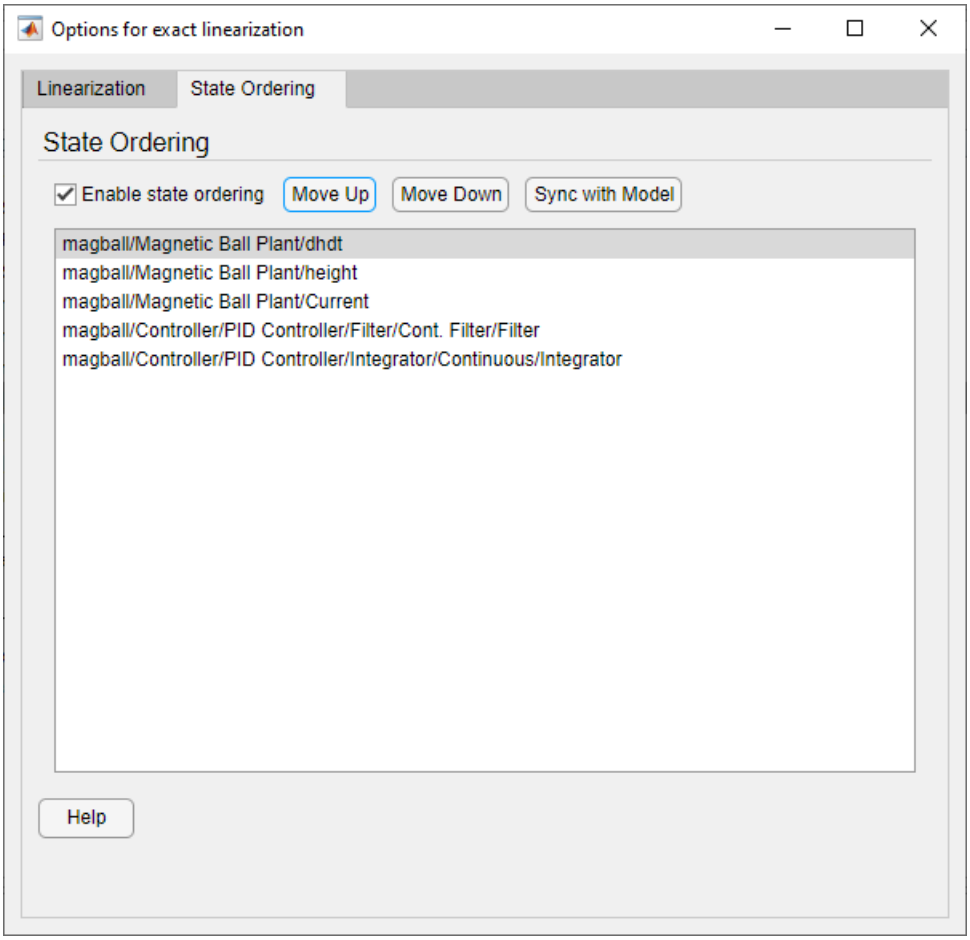
   On the **Linear Analysis** tab, click **More Options**.



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

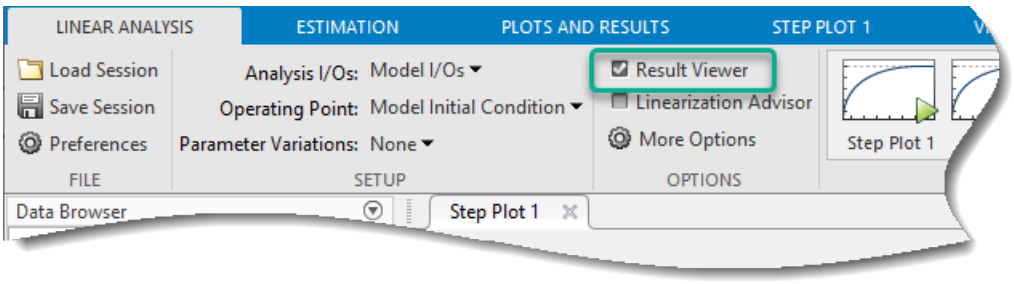
5 Specify the desired state order using the **Move Up** and **Move Down** buttons.

Tip If you change the model while **Model Linearizer** is open, click **Sync with Model** to update the list of states.



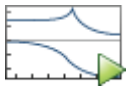
Close the dialog box.

6 Enable the linearization result viewer. On the **Linear Analysis** tab, select **Result Viewer**.

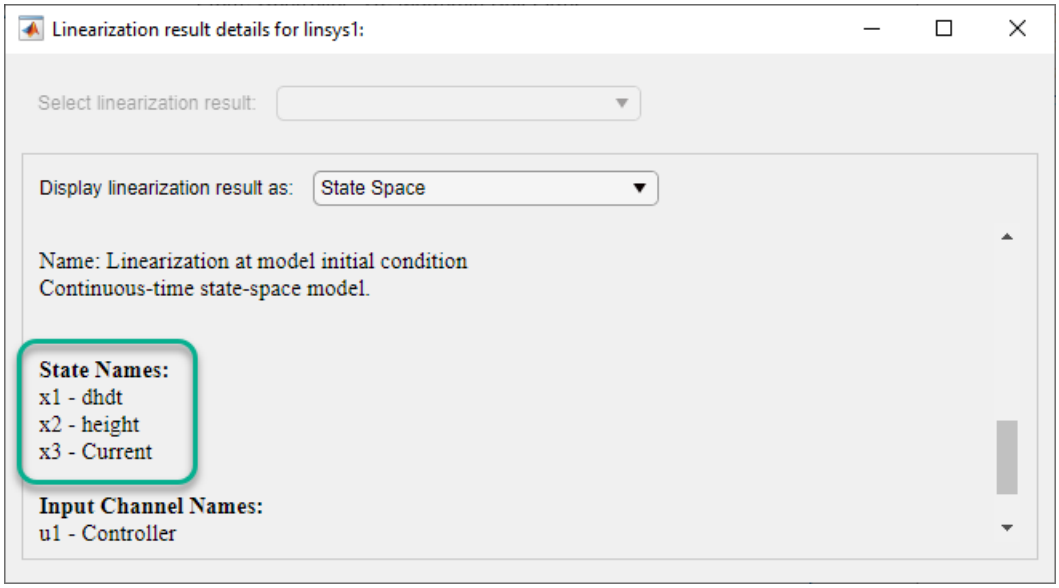


When this option is selected, the result viewer appears when you linearize the model, enabling you to view and confirm the state ordering.

Tip If you do not check **Result Viewer**, or if you close the result viewer, you can open the result viewer for a previously linearized model. To do so, in the **Plots and Results** tab, select the linear model in the **Linear Analysis Workspace**, and click **Result Viewer**.

7 Linearize the model. For example, click  **Bode**.

A new linearized model, `linsys1`, appears in the **Linear Analysis Workspace**. The linearization result viewer opens, displaying information about that model.



The linear model states appear in the specified order.

Control State Order of Linearized Model using MATLAB Code

This example shows how to control the order of the states in your linearized model. This state order appears in linearization results.

1 Load and configure the model for linearization.

```
sys = 'magball';
load_system(sys);
sys_io(1)=linio('magball/Controller',1,'input');
sys_io(2)=linio('magball/Magnetic Ball Plant',1,'openoutput');
opspec = operspec(sys);
op = findop(sys,opspec);
```

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

2 Linearize the model, and show the linear model states.

```
linsys = linearize(sys,sys_io);
linsys.StateName
```

The linear model states are in default order. The linear model includes only the states in the linearized blocks, and not the states of the full model.

```
ans =
    'height'
    'Current'
    'dhdt'
```

3 Define a different state order.

```
stateorder = {'magball/Magnetic Ball Plant/height';...
              'magball/Magnetic Ball Plant/dhdt';...
              'magball/Magnetic Ball Plant/Current'};
```

4 Linearize the model again and show the linear model states.

```
linsys = linearize(sys,sys_io,'StateOrder',stateorder);
linsys.StateName
```

The linear model states are now in the specified order.

```
ans =
    'height'
    'dhdt'
    'Current'
```

Validate Linearization In Time Domain

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:

   If you linearized your model in the **Model Linearizer**, you must export the linear model to the MATLAB workspace.
2. Create input signal for validation. For example, a step input signal:
3. Simulate the Simulink model using the input signal.

   `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.

   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`.
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.

   The step response of linear system you obtained using exact linearization does not match the step response of the estimated frequency response with large input signal amplitude. The linear model obtained using exact linearization does not match the full nonlinear model at amplitudes large enough to deviate from the specified operating point.

Choosing Time-Domain Validation Input Signal

For time-domain validation of linearization, use `frest.createStep` to create a step signal. Use the step signal as an input to `frest.simCompare`, which compares the simulated output of the nonlinear model and the linearized model.

The step input helps you assess whether the linear model accurately captures the dominant time constants as it goes through the step transients.

The step input also shows whether you correctly captured the DC gain of the Simulink model by comparing the final value of the exact linearization simulation with the frequency response estimation.

Validate Linearization In Frequency Domain Using Model Linearizer

This example shows how to validate linearization results using an estimated linear model.

In this example, you linearize a Simulink model using the I/Os specified in the model. You then estimate the frequency response of the model using the same operating point (model initial condition). Finally, you compare the estimated response to the exact linearization result.

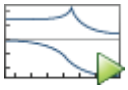
Linearize Model

Open the model.

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

Open the **Model Linearizer** app for the model.

In the Simulink model window, in the **Apps** gallery, click **Model Linearizer**.

Linearize the model at the default operating point and analysis I/Os, and generate a bode plot of the result. To do so, click  **Bode**.

The Bode plot of the linearized plant appears, and the linearized plant `linsys1` appears in the **Linear Analysis Workspace**.

Estimate Frequency Response of Model

For frequency-domain validation of linearization, use a sinestream input 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`

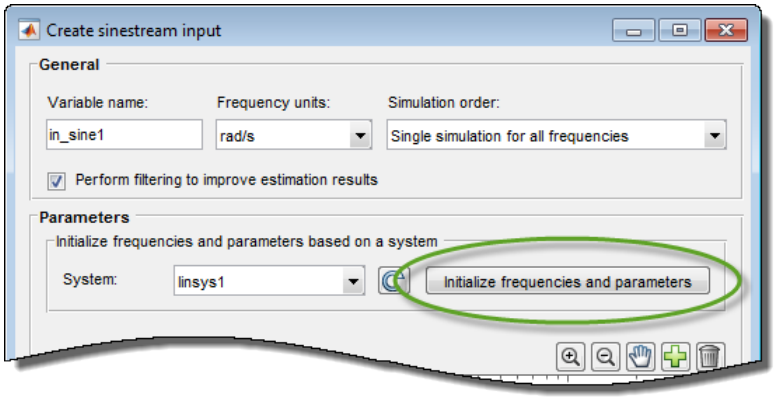
You can create a sinestream signal based on your linearized model. The software uses the linearized model characteristics to accurately predict the number of sinusoid cycles at each frequency to reach steady state.

When diagnosing the frequency response estimation, you can use the sinestream signal to determine whether the time series at each frequency reaches steady state.

Create a sinestream input signal for computing an approximation of the model by frequency response estimation. On the **Estimation** tab, in the **Input Signal** drop-down list, select `Sinestream`.

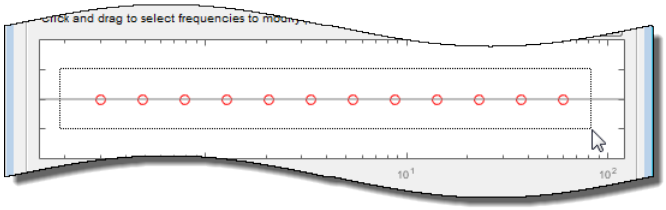
Initialize the input signal frequencies and parameters based on the linearized model.

Click **Initialize frequencies and parameters**.

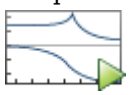


The frequency display in the dialog box is populated with frequency points. The software chooses the frequencies and input signal parameters automatically based on the dynamics of `linsys1`.

Set the amplitude of the input signal at all frequency points to 1. In the frequency display, select all the frequency points.

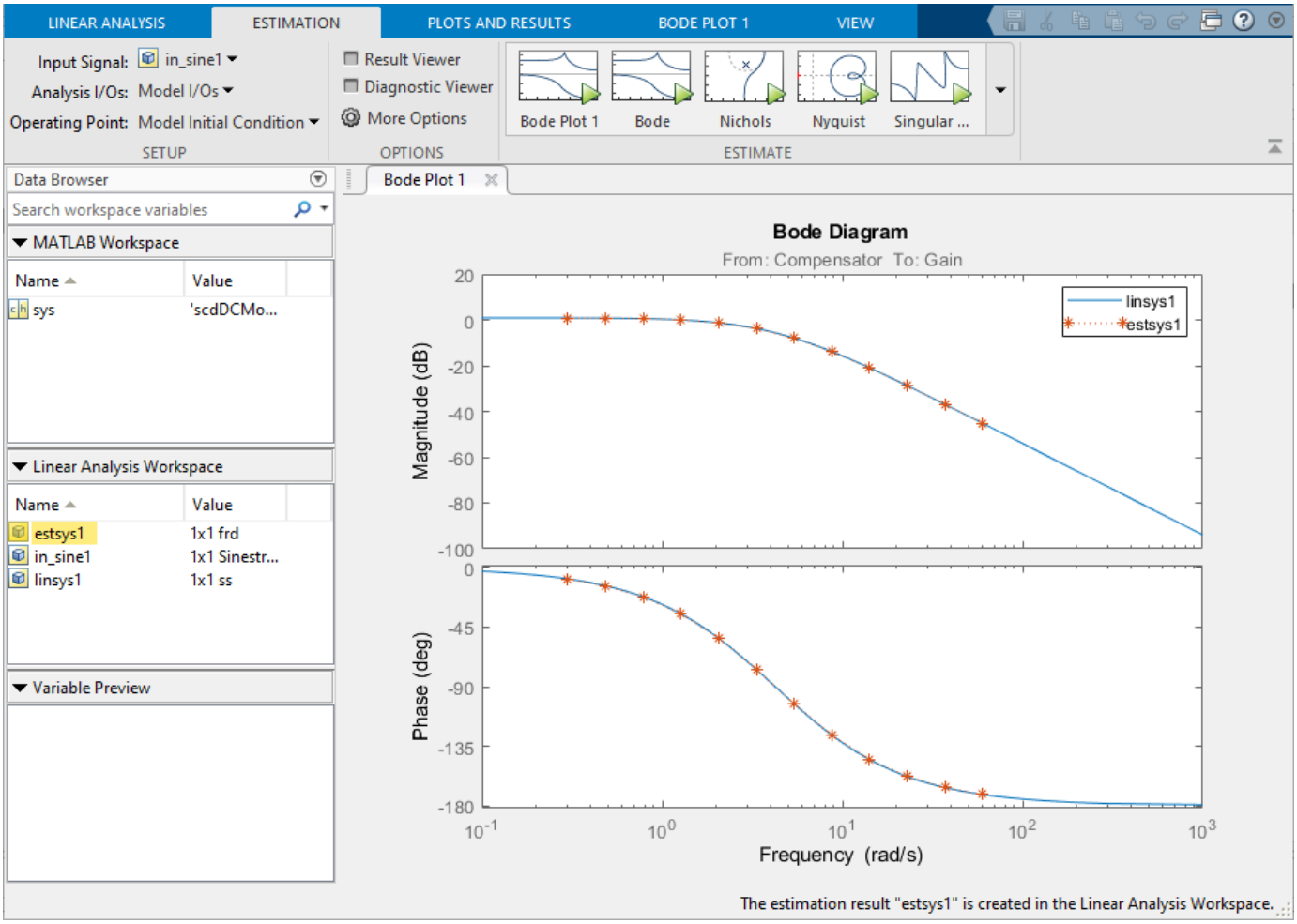


Enter 1 in the **Amplitude** field, and click **OK**. The new input signal `in_sine1` appears in the **Linear Analysis Workspace**.

Estimate the frequency response and plot its frequency response on the existing Bode plot of the linearized system response. Click  **Bode Plot 1**.

Examine estimation results.

Bode Plot 1 now shows the Bode responses for the estimated model and the linearized model.



The frequency response for the estimated model matches that of the linearized model.

For more information about frequency response estimation, see “Frequency Response Estimation Basics” on page 5-2.

See Also

Model Linearizer

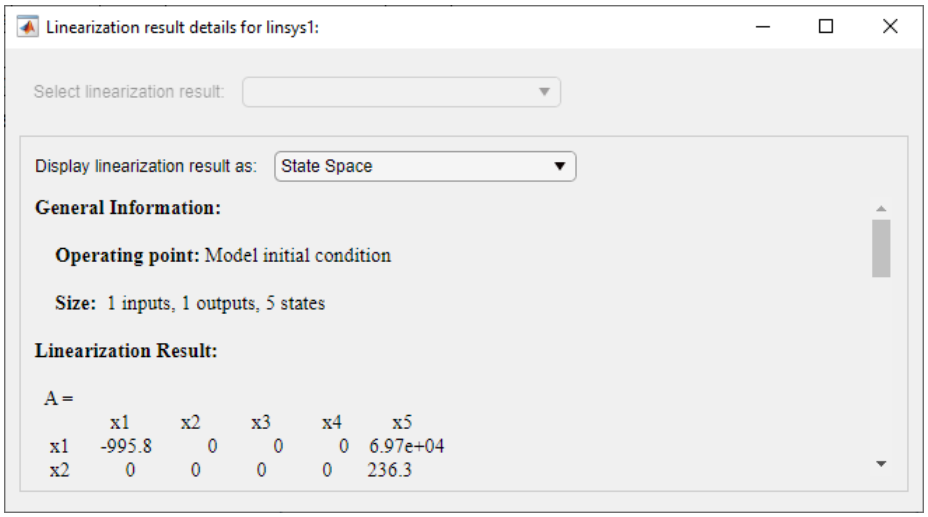
Related Examples

- “Estimation Input Signals” on page 5-24

View Linearized Model Equations Using Model Linearizer

When you linearize a Simulink model using the **Model Linearizer**, the software generates state-space equations for the resulting linear model. To view the linearized model equations:

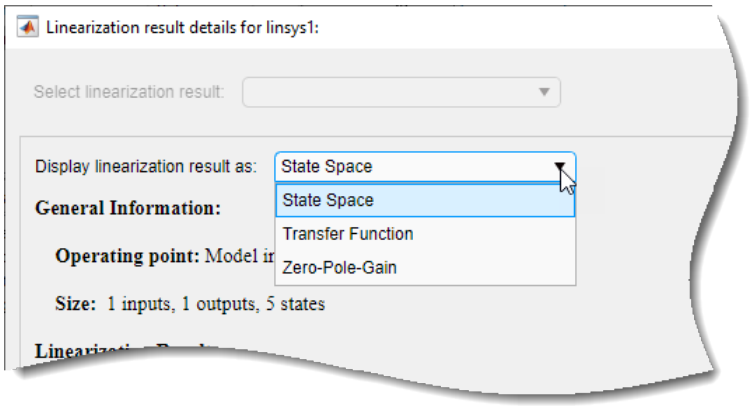
1. In the **Data Browser**, in the **Linear Analysis Workspace**, select the linear model you want to view.
2. On the **Plots and Results** tab, click **Result Viewer**.



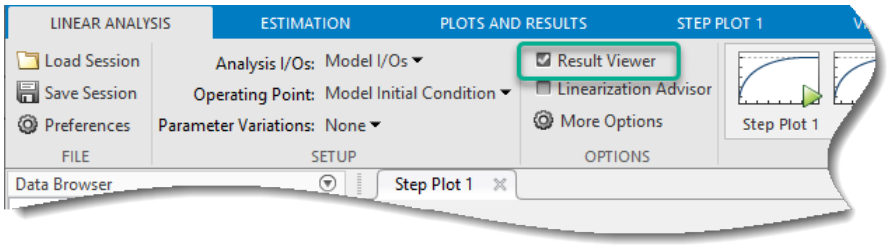
In the Linearization result details dialog box, the software displays:

- General information about the linearization, including the operating point and the number of inputs, outputs, and states.
- State-space matrices for the linearized model.
- Lists of the state, input, and output names. To highlight a state, input, or output in the Simulink model, click the corresponding name.

To display the system using either zero-pole-gain or transfer function equations, in the **Display linearization result as** drop-down list, select a format.



You can automatically open the Linearization result details dialog box when you linearize your model. To do so, on the **Linear Analysis** tab, select **Result Viewer** before you linearize the model.



See Also

Apps
Model Linearizer

More About

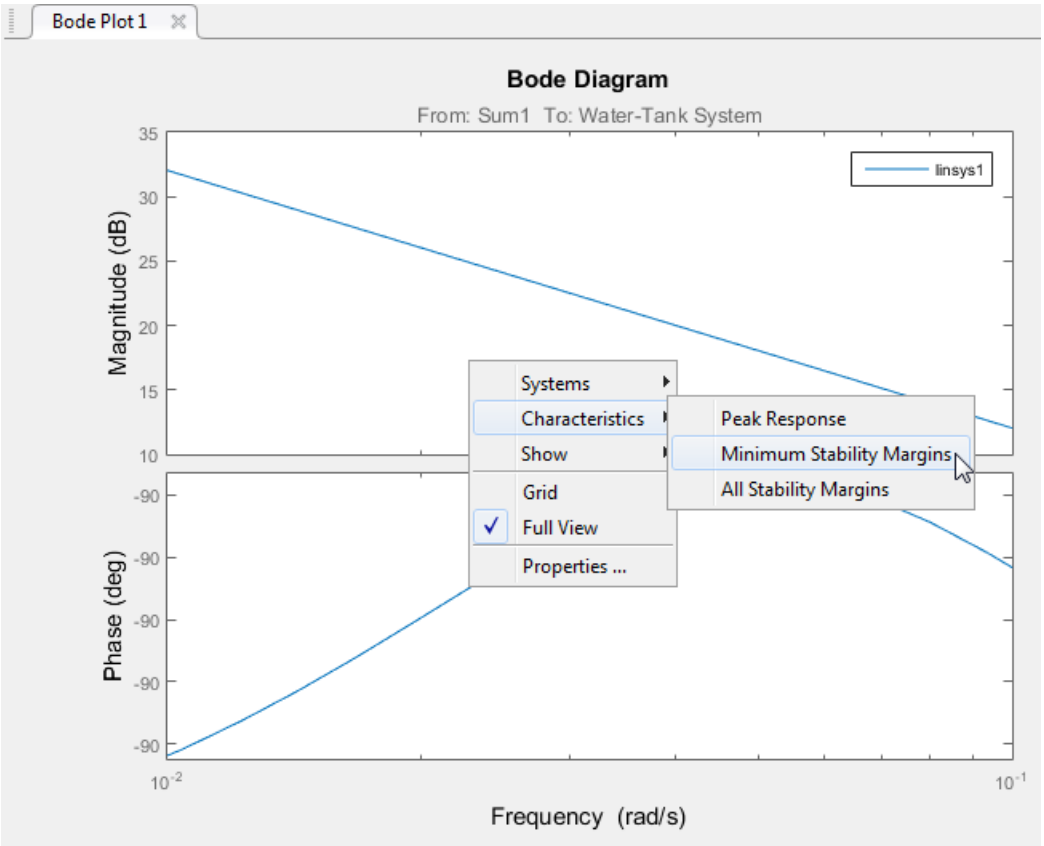
- “Analyze Results Using Model Linearizer Response Plots” on page 2-113

Analyze Results Using Model Linearizer Response Plots

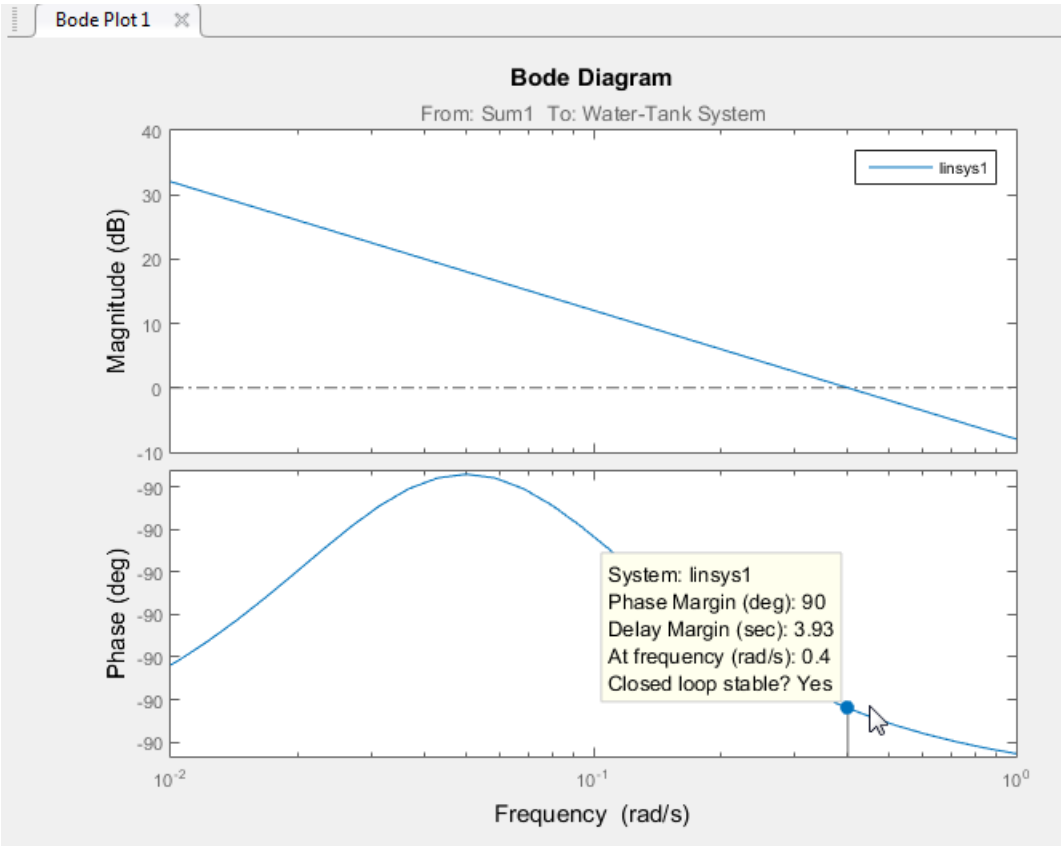
Using **Model Linearizer**, you can analyze time-domain and frequency-domain responses of linearized models. You can compare the responses of multiple models and view system characteristics such as stability margins and settling time.

View System Characteristics on Response Plots

To view system characteristics such as stability margins, overshoot, or settling time on a **Model Linearizer** response plot, right-click the plot and select **Characteristics**. Then select the system characteristic you want to view.



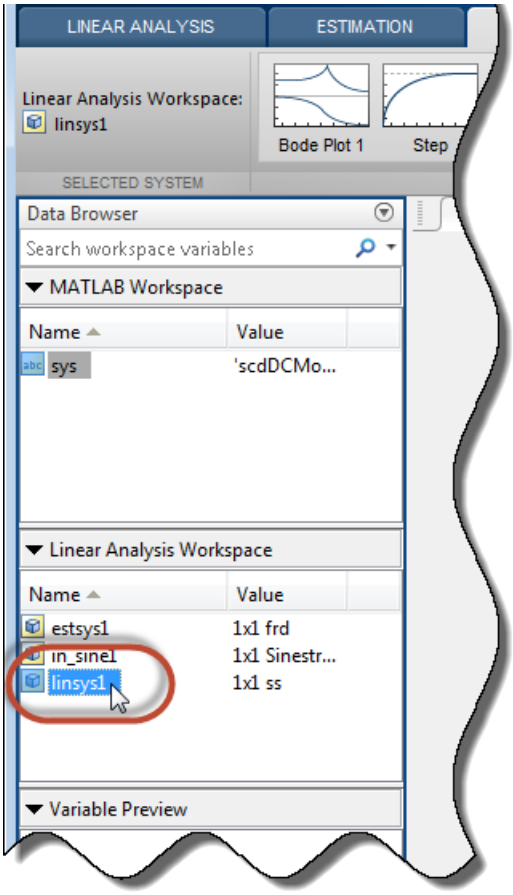
For most characteristics, a data marker appears on the plot. To show a data tip that contains information about the system characteristic, click the marker.



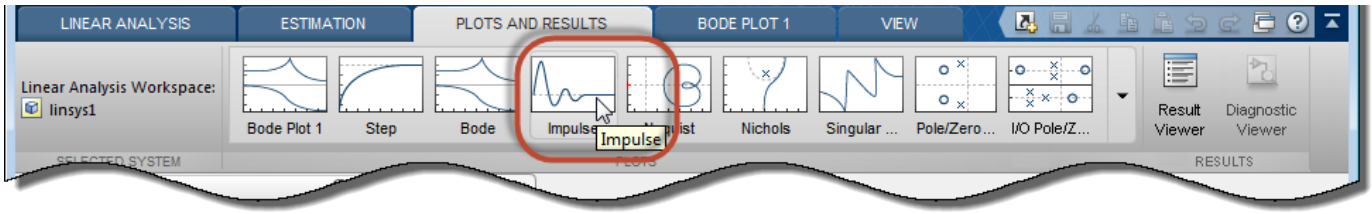
Generate Additional Response Plots of Linearized System

In **Model Linearizer**, when you have linearized or estimated a system, you can generate additional response plots of the system as follows:

1 In the **Model Linearizer**, click the **Plots and Results** tab. In the **Linear Analysis Workspace** or the **MATLAB Workspace**, select the system you want to plot.

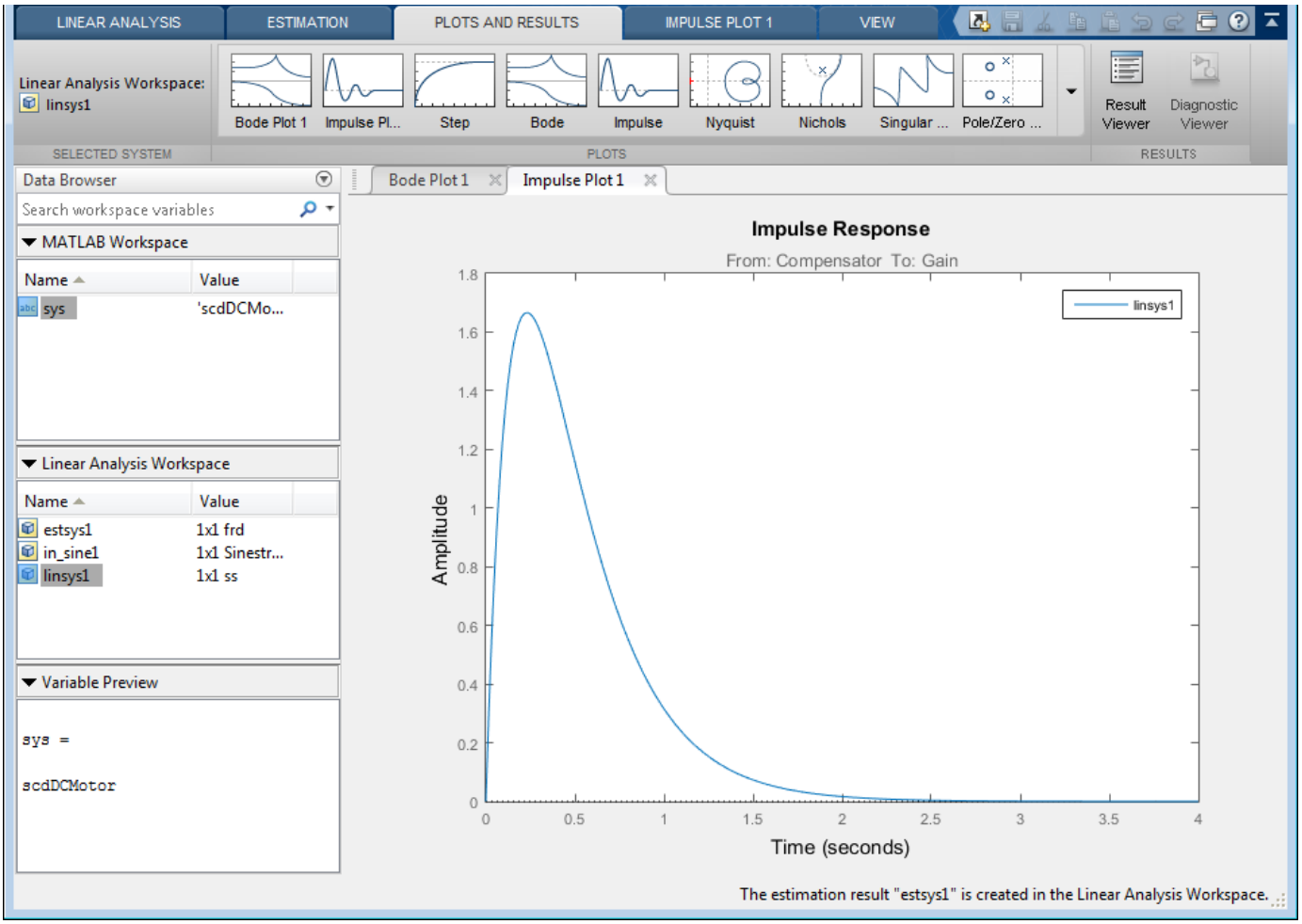


2 In the **Plots** section of the tab, click the type of plot you want to generate.



Tip To expand the gallery view, click ▾.

Model Linearizer generates a new plot of type you select.

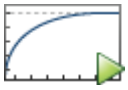


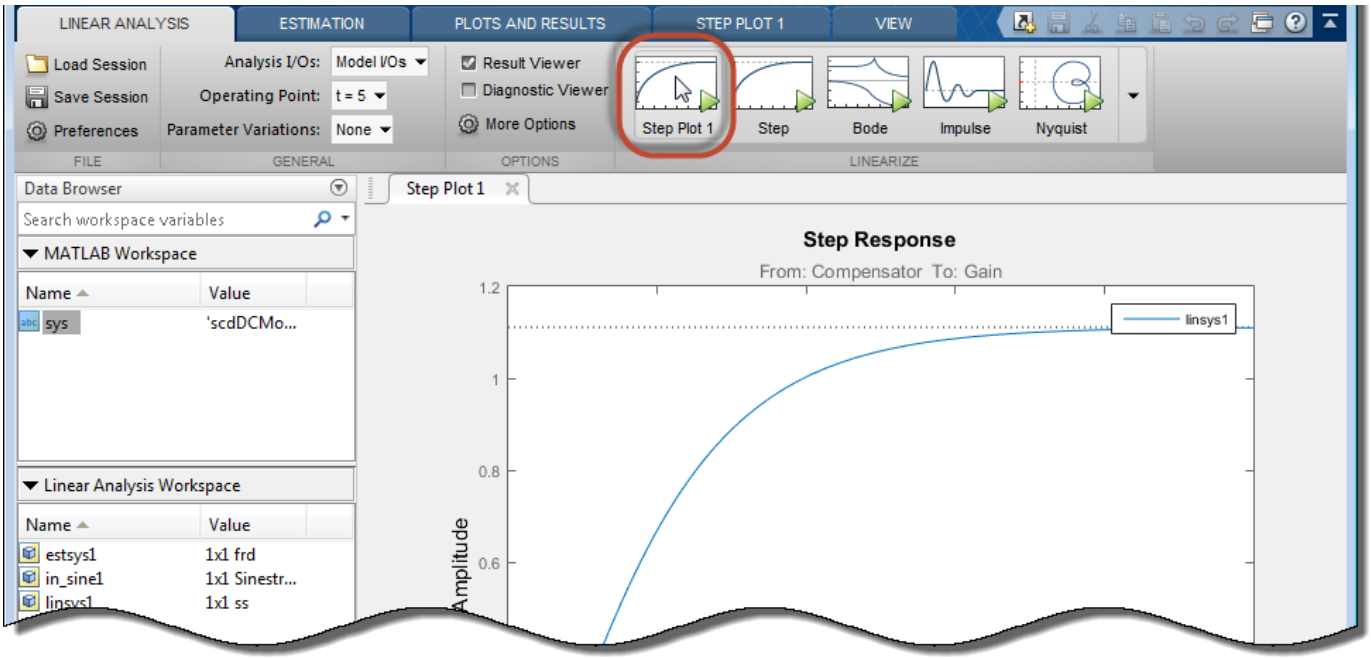
Tip To view multiple plots at the same time, select a layout in the **View** tab.

Add Linear System to Existing Response Plot

New Linear System

When you compute a new linearization or frequency response estimation, on the **Linear Analysis** tab, click the button corresponding to an existing plot to add the new linear system to that plot.

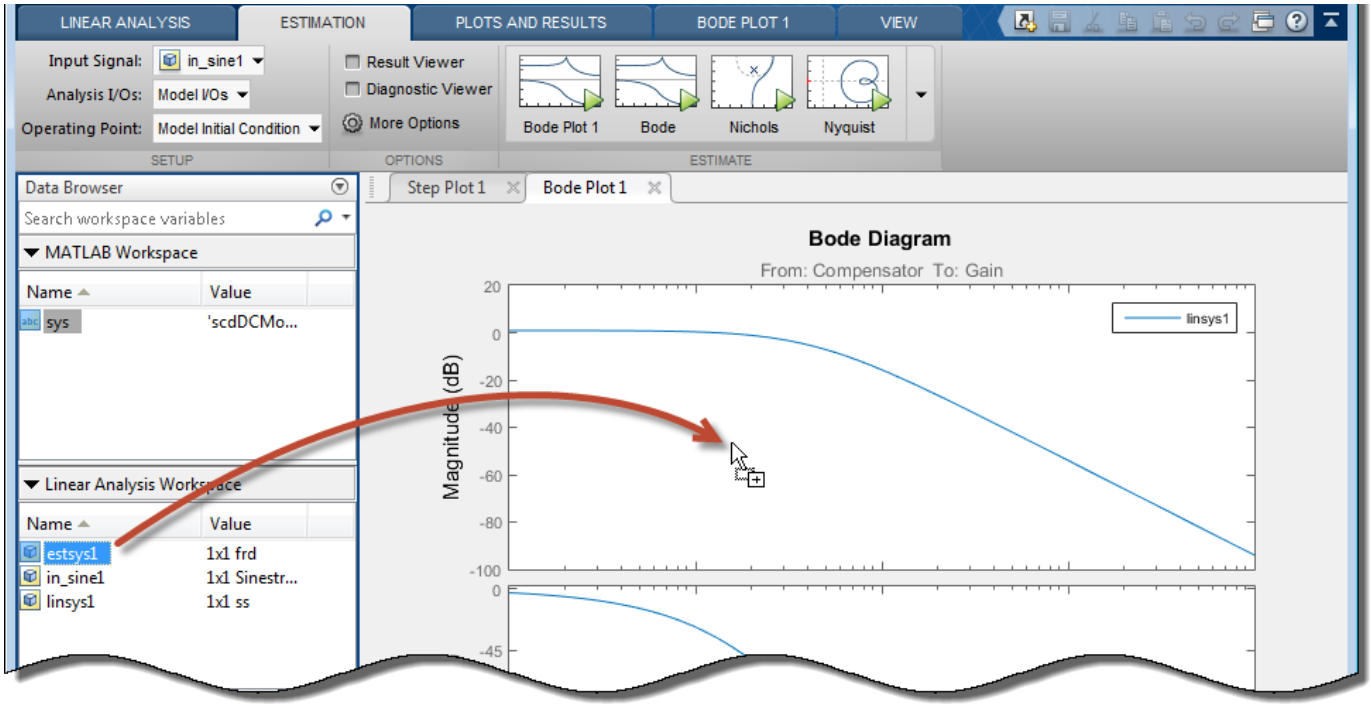
For example, suppose that you have linearized a model at the default operating point for the model, and have a step plot of the result, `Step Plot 1`. Suppose further that you have specified a new operating point at a linearization snapshot time. To linearize at the new operating point and add the result to `Step Plot 1`, click  **Step Plot 1**. **Model Linearizer** computes the new linearization and adds the step response of the new system, `linsys2`, to the existing step response plot.



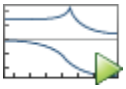
Linear System in Workspace

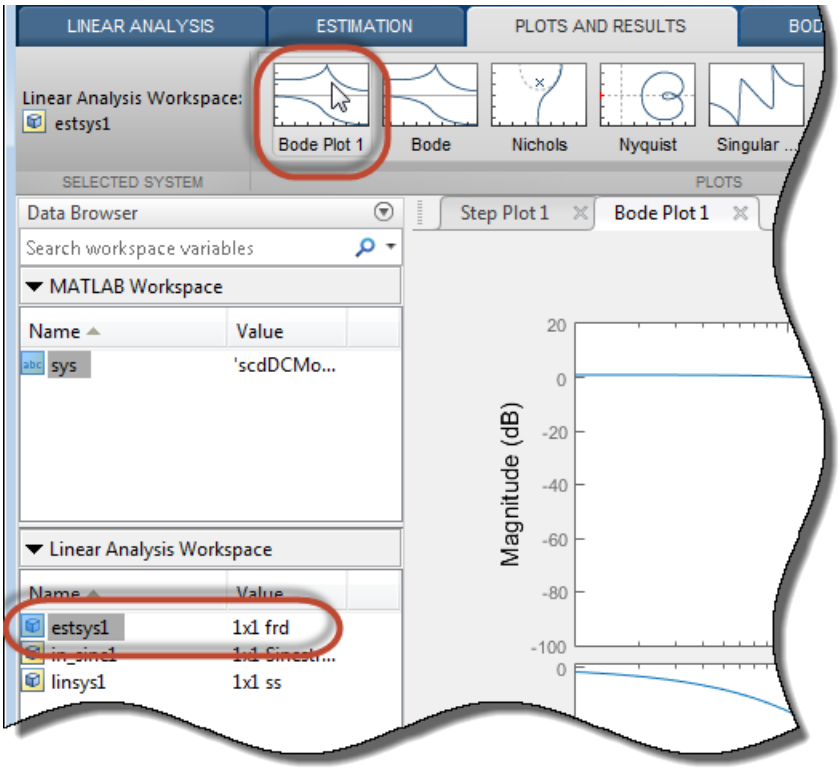
There are two ways to add a linear system from the **MATLAB Workspace** or the **Linear Analysis Workspace** to an existing plot in the **Model Linearizer**.

- Drag the linear system onto the plot from the **MATLAB Workspace** or the **Linear Analysis Workspace**.



- On the **Plots and Results** tab, in the **Linear Analysis Workspace**, select the system you want to add to an existing plot. Then, in the **Plots** section of the tab, select the button corresponding to the existing plot you want to update.

  For example, suppose that you have a Bode plot of the response of a linear system, `Bode Plot 1`. Suppose further that you have an estimated response in the **Linear Analysis Workspace**, `estsys1`. To add the response of `estsys1` to the existing Bode plot, select `estsys1` and click  **Bode Plot 1**.



Tip To expand the gallery view, click ▾.

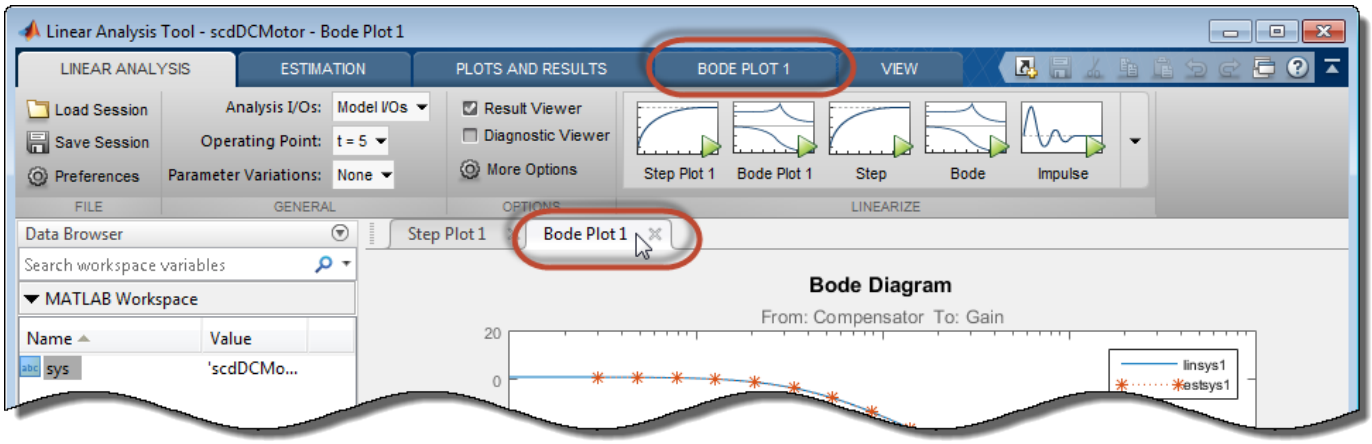
Customize Characteristics of Plot in Model Linearizer

To change the characteristics of an existing plot, such as the title, axis labels, or text styles, double-click the plot to open the properties editor. Edit plot properties as desired. Plots are updated as you make changes. When you are finished, click **Close**.

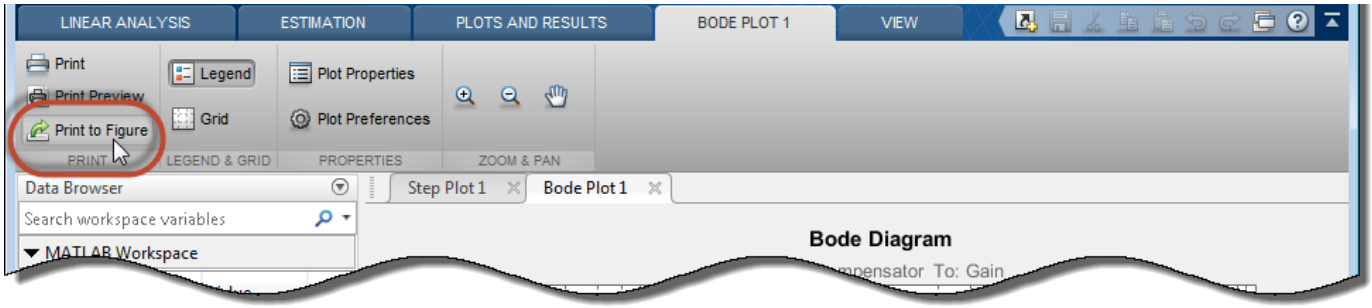
Print Plot to MATLAB Figure in Model Linearizer

To export a plot from the **Model Linearizer** to a MATLAB figure window:

1. Select the plot you want to export. A tab appears with the same name as the plot.



2 Click the new tab. In the **Print** section, click **Print to Figure**.



A MATLAB figure window opens containing the plot.

See Also

Model Linearizer

More About

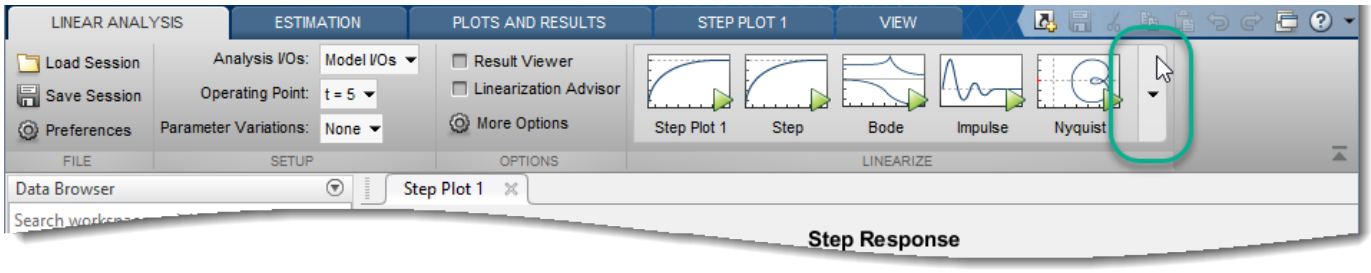
- “View Linearized Model Equations Using Model Linearizer” on page 2-111
- “Linearize at Trimmed Operating Point” on page 2-67
- “Linearize at Simulation Snapshot” on page 2-72

Generate MATLAB Code for Linearization from Model Linearizer

This topic shows how to generate MATLAB code for linearization from the **Model Linearizer**. You can generate either a MATLAB script or a MATLAB function. To programmatically reproduce a linearization result that you obtained interactively, you can use a generated MATLAB script. To perform multiple linearizations with systematic variations in your linearization configuration, you can use a generated MATLAB function.

To generate MATLAB code for linearization:

1. In the **Model Linearizer**, on the **Linear Analysis** tab, interactively configure the analysis points, operating points, and parameter variations for linearization.
2. In the **Linearize** section, expand the gallery by clicking .



3. In the gallery, depending on the type of code you want to create, click:

   - `Script` — Generate a MATLAB script that uses your configured analysis points, operating points, and parameter variations. Select this option when you want to repeat the same linearization at the MATLAB command line.
   - `Function` — Generate a MATLAB function that takes analysis points, operating points, and parameter variations as input arguments. Select this option when you want to perform multiple linearizations using different configurations (batch linearization). For more information on varying operating points and parameter when using the `linearize` function, see:

     - “Batch Linearize Model for Parameter Variations at Single Operating Point” on page 3-13
     - “Batch Linearize Model at Multiple Operating Points Using linearize Command” on page 3-19

   The software creates a MATLAB file that contains the generated code and opens the file in the MATLAB Editor.
4. In the MATLAB Editor, you can edit and save the file.

See Also

Functions
`linearize`

Apps
Model Linearizer

More About

- “What Is Batch Linearization?” on page 3-2

When to Specify Individual Block Linearization

Some Simulink blocks, including those with sharp discontinuities, can produce poor linearization results. For example, when your model operates in a region away from the point of discontinuity, the linearization of the block is zero. Typically, you must specify custom linearizations for such blocks. You can specify the block linearization as:

- A linear model in the form of a D-matrix.
- A Control System Toolbox model object.
- An uncertain state-space object or an uncertain real object (requires Robust Control Toolbox software).

See Also

More About

- “Specify Linear System for Block Linearization Using MATLAB Expression” on page 2-123
- “Specify D-Matrix System for Block Linearization Using Function” on page 2-124
- “Augment Block Linearization” on page 2-133
- “Change Perturbation Level of Blocks Perturbed During Linearization” on page 2-148
- “Block Linearization Troubleshooting” on page 4-42

Specify Linear System for Block Linearization Using MATLAB Expression

This example shows how to specify the linearization of any block, subsystem, or model reference without having to replace this block in your Simulink model.

1. Right-click the block in the model, and select **Linear Analysis > Specify Selected Block Linearization**.

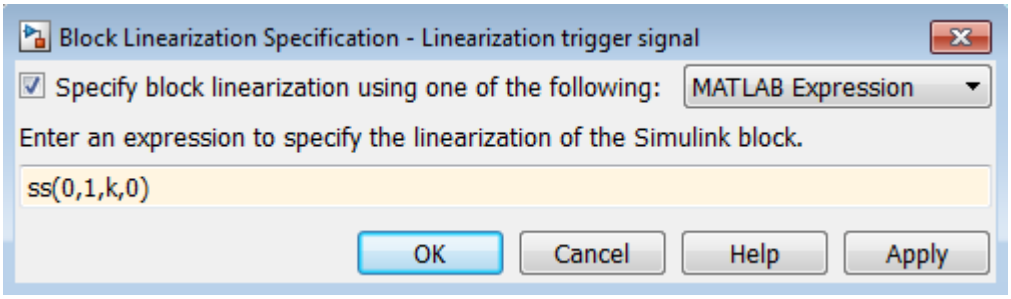
   The Block Linearization Specification dialog box opens.

2. In the **Specify block linearization using one of the following** list, select **MATLAB Expression**.

3. In the text field, enter an expression that specifies the linearization.

   For example, specify the linearization as an integrator with a gain of k, $G(s) = k/s$.

   In state-space form, this transfer function corresponds to `ss(0,1,k,0)`.



   Click **OK**.

4. Linearize the model.

See Also

Related Examples

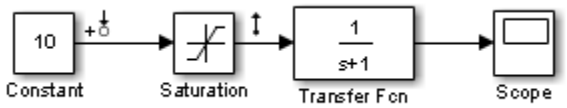
- “Specify D-Matrix System for Block Linearization Using Function” on page 2-124
- “Block Linearization Troubleshooting” on page 4-42

Specify D-Matrix System for Block Linearization Using Function

This example shows how to specify custom linearization for a saturation block using a function.

Open the Simulink model.

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



In this model, the limits of the saturation block are –`satlimit` and `satlimit`.

Define the saturation limit, which is a parameter required by the linearization function of the Saturation block.

```
satlimit = 10;
```

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(mdl);
linsys = linearize(mdl,io)
```

```
linsys =

  D =
                Constant
   Saturation       0.5

Static gain.
```

At the model operating point, the input to the saturation block is 10. This value is on the saturation boundary. At this value, the saturation block linearizes to 1.

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. For this example, use the configuration function in `mySaturationLinearizationFcn.m`.

```
function blocklin = mySaturationLinearizationFcn(BlockData)
% This function customizes the linearization of a saturation block. The
% linearization of the block is as follows
% BLOCKLIN = 0 when |U| > saturation limit
% BLOCKLIN = 1 when |U| < saturation limit
% BLOCKLIN = 0.5 when U = saturation limit

% Get the saturation limit specified in the parameters of the block.
satlimit = BlockData.Parameters.Value;

% Compute the linearization based on the input signal level to the block.
if abs(BlockData.Inputs.Values) > satlimit
    blocklin = 0;
elseif abs(BlockData.Inputs.Values) < satlimit
```

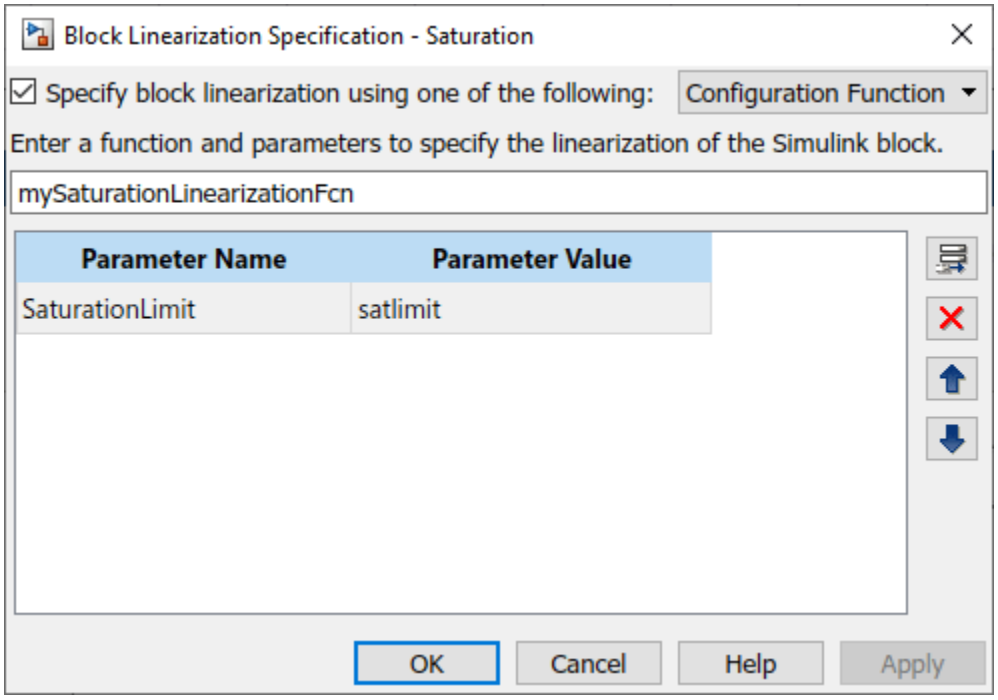
```
    blocklin = 1;
else
    blocklin = 0.5;
end
```

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. For 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.

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. If your configuration function requires additional parameters, you can configure the block to set those parameters in the `BlockData.Parameters` field.

Specify `mySaturationLinearizationFcn` as the linearization for the Controller block.

1. In the Simulink model, right-click the Saturation block, and select **Linear Analysis > Specify Selected Block Linearization**.
2. In the Block Linearization Specification dialog box, select **Specify block linearization using one of the following**.
3. In the drop-down list, select `Configuration Function`.
4. In the text box, enter the function name `mySaturationLinearizationFcn`.
5. Specify the saturation limit as a parameter for the function. In the **Parameter Value** column, enter the variable name `satlimit`.
6. In the **Parameter Name** column, enter the corresponding descriptive name `SaturationLimit`.
7. Click **OK**.



Alternatively, you can define the configuration function programmatically.

```
satblk = 'configSatBlockFcn/Saturation';
set_param(satblk,'SCDEnableBlockLinearizationSpecification','on')
rep = struct('Specification','mySaturationLinearizationFcn',...
             'Type','Function',...
             'ParameterNames','SaturationLimit',...
             'ParameterValues','satlimit');
set_param(satblk,'SCDBlockLinearizationSpecification',rep)
```

Linearize the model again. Now, the linearization uses the custom linearization of the saturation block.

```
linsys_cust = linearize(mdl,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 configuration function.

See Also

More About

- “Specify Linear System for Block Linearization Using MATLAB Expression” on page 2-123
- “Block Linearization Troubleshooting” on page 4-42

Specify Custom Linearizations for Simulink Blocks

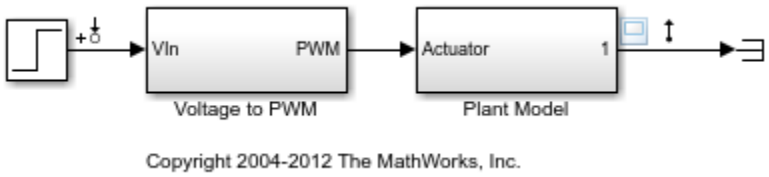
This example shows how to specify the linearization of a Simulink® block or subsystem.

In Simulink Control Design™ software, linearization provides a linear approximation of a Simulink model using an exact linearization approach. This linearization is valid in a small region around a given operating point. This approach works well for most Simulink models. However, in some cases, you must modify the exact linearization approach to take into account the effects of discontinuities or approximate the dynamics of derivative or delay action. Many built-in Simulink blocks, such as Saturation or Dead Zone blocks, provide a **Treat as gain when linearizing** parameter to control this behavior. For blocks or subsystems that cannot be linearized, it can be necessary to specify a linearization. In this example, you specify a custom linearization for a subsystem to approximate the linearization of a PWM signal.

Linearize a Model with a PWM Generation Subsystem

The `scdpwm` model contains a Voltage to PWM subsystem that models a PWM signal, which then enters a plant model.

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



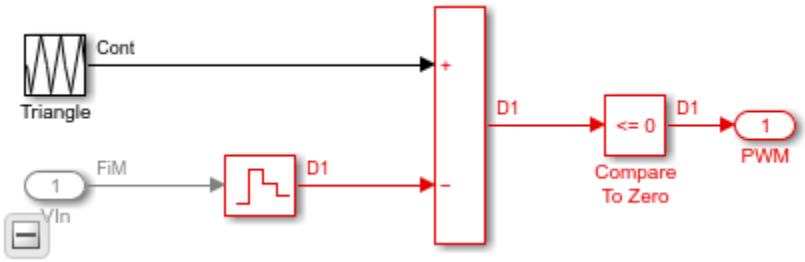
When you linearize this model using the standard configuration, the resulting linear model has a gain of zero.

```
io = getlinio(mdl);
sys = linearize(mdl,io)
```

```
sys =

  D =
                 Step
   Plant Model      0

Static gain.
```

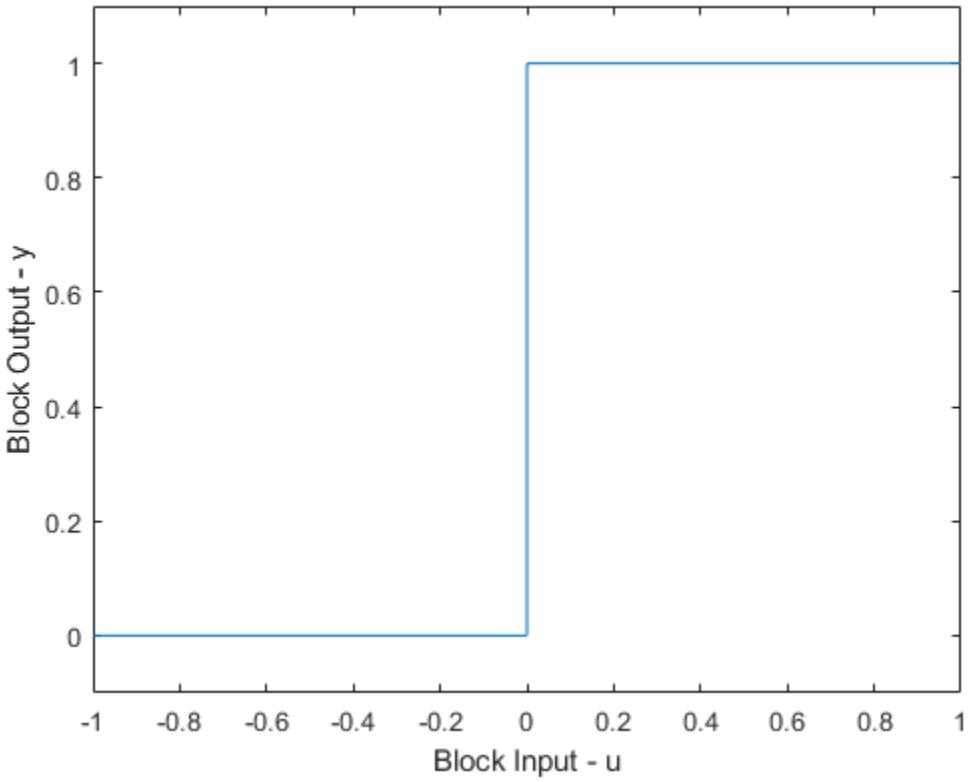
The Voltage to PWM/Compare To Zero block causes this linearization to be zero.

```
pwmblk = 'scdpwm/Voltage to PWM';
open_system(pwmblk)
```



The linearization is zero because the block represents a pure discontinuous nonlinearity.

```
u = [-1:0.1:0,0:0.1:1];
y = [zeros(11,1);ones(11,1)];
plot(u,y)
xlabel('Block Input - u')
ylabel('Block Output - y')
ylim([-0.1 1.1])
```



Specify a Custom Linearization for the PWM Subsystem

With Simulink Control Design software, you can control the linearization of the blocks in a Simulink model. You can specify the linearization of a block using:

- Matrices
- Linear time invariant models, such as transfer functions or state-space models

- Uncertain parameters or state-space models (requires Robust Control toolbox™ software)

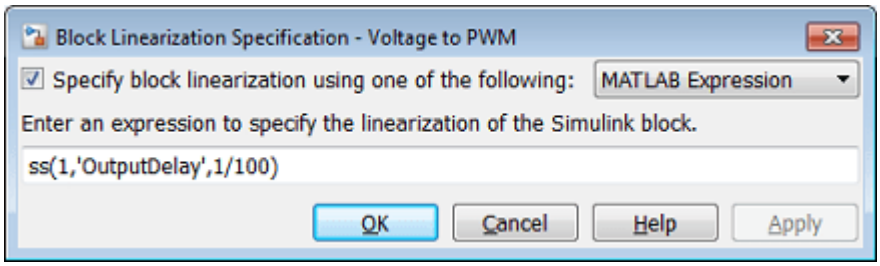
In this example, the PWM subsystem contains a time delay

$$PWM(s) = e^{-sT_s}$$

This time delay accounts for the duty cycle frequency of the PWM signal which is 100 Hz. To specify the delay for the Voltage to PWM subsystem, first select the block. Then, on the **Apps** tab, click **Linearization Manager**.

On the **Linearization** tab, click **Specify Block Linearization**. Then, in the Block Linearization Specification dialog box, perform the following steps.

1. Select the **Specify block linearization using one of the following** parameter.
2. In the drop-down box, select `MATLAB Expression`.
3. Specify the linearization using the expression `ss(1,'OutputDelay',1/100)`.



The following code is equivalent to entering the delay into the Block Linearization Specification dialog box.

```
set_param(pwmblk,'SCDEnableBlockLinearizationSpecification','on');
rep = struct('Specification','ss(1,''OutputDelay'',1/100)',...
             'Type','Expression',...
             'ParameterNames','',...
             'ParameterValues','');
set_param(pwmblk,'SCDBlockLinearizationSpecification',rep);
```

Create a continuous-time linear model of the system.

```
opt = linearizeOptions('SampleTime',0);
sys = linearize(mdl,io,opt);
```

The linearized model, which includes the specified subsystem linearization, now gives the expected result.

```
sys = zpk(sys)


sys =

  From input "Step" to output "Plant Model":
                        1
  exp(-0.01*s) * -------------
                 (s^2 + s + 1)

Continuous-time zero/pole/gain model.
```

Compare the Linearization and Simulation

You can compare the linearization of the model to an estimated frequency response of the model using a `frest.createStep` input signal.

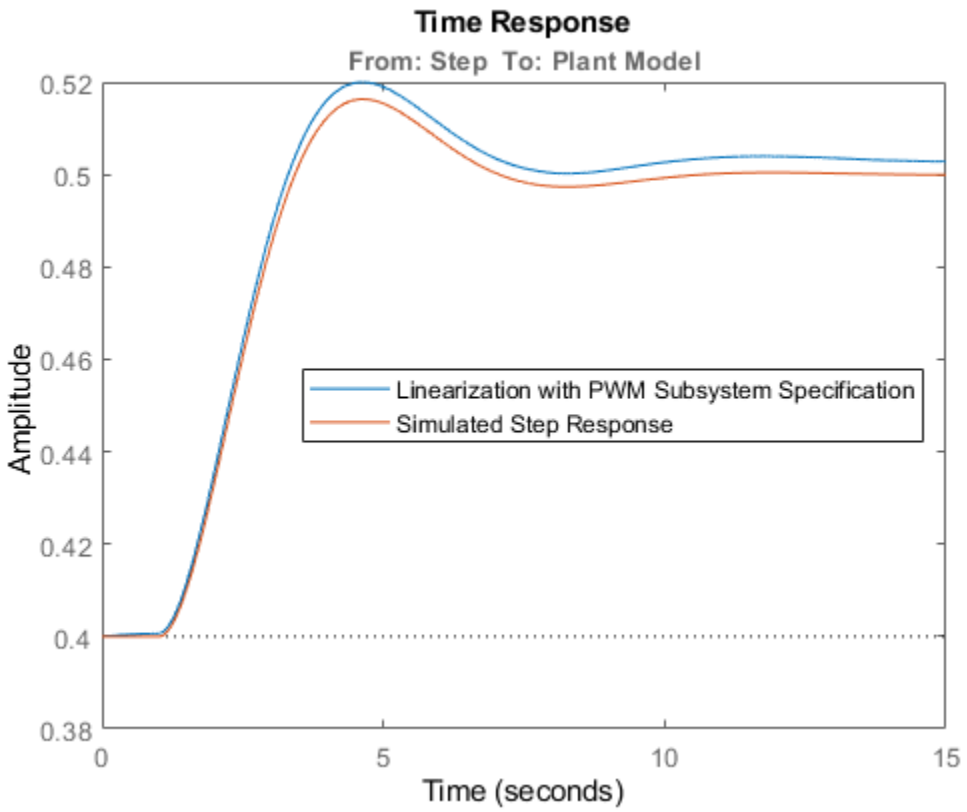
The linearization you specified for the Voltage to PWM subsystem affects only linearization and not model simulation. Therefore, you do not need to remove the linearization specification before you estimate the frequency response.

Create an input signal and estimate the frequency response of the model.

```
instep = frest.createStep('Ts',1/10000,'StepTime',1,...
                          'StepSize',1e-1,'FinalTime',15);
[sysf,simoutstep] = frestimate(mdl,io,instep);
```

Compare the linearized model with the simulation results from the estimated model.

```
frest.simCompare(simoutstep,sys,instep)
legend('Linearization with PWM Subsystem Specification',...
            'Simulated Step Response','Location','East')
```



The linearization accurately represents the dynamics of the estimated system.

Other Applications for Custom Linearizations

Block linearization specification is not limited to linear time-invariant models. If you have Robust Control Toolbox™ software, you can specify uncertain parameters and uncertain state-space (USS)

models for blocks in a model. The resulting linearization is then an uncertain model. The example “Linearization of Simulink Models with Uncertainty” (Robust Control Toolbox) shows how to compute a linearization with uncertainty.

You can also perform analysis of models with discrete controllers and continuous plant dynamics in the continuous domain. For more details, see “Model Computational Delay and Sampling Effects” on page 9-86.

Close the Simulink model.

```
bdclose(mdl)
```

See Also

`linearize`

Related Examples

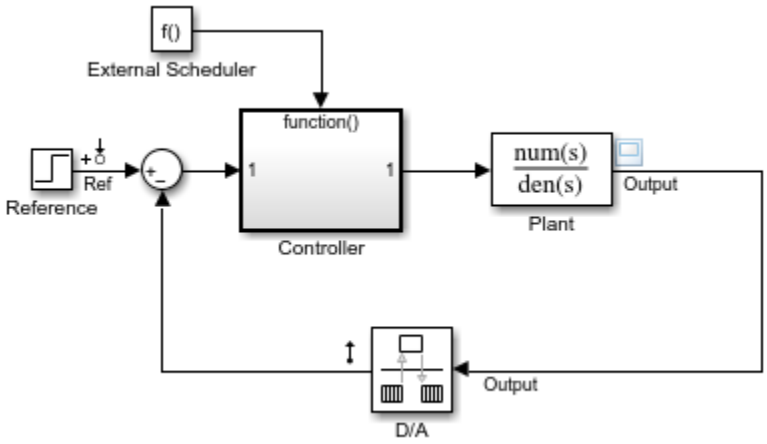
- “When to Specify Individual Block Linearization” on page 2-122
- “Specify Linearization for Model Components Using System Identification” on page 2-168

Augment Block Linearization

This example shows how to augment the linearization of a block with additional time delay dynamics using a block linearization specification function.

Open the Simulink model.

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



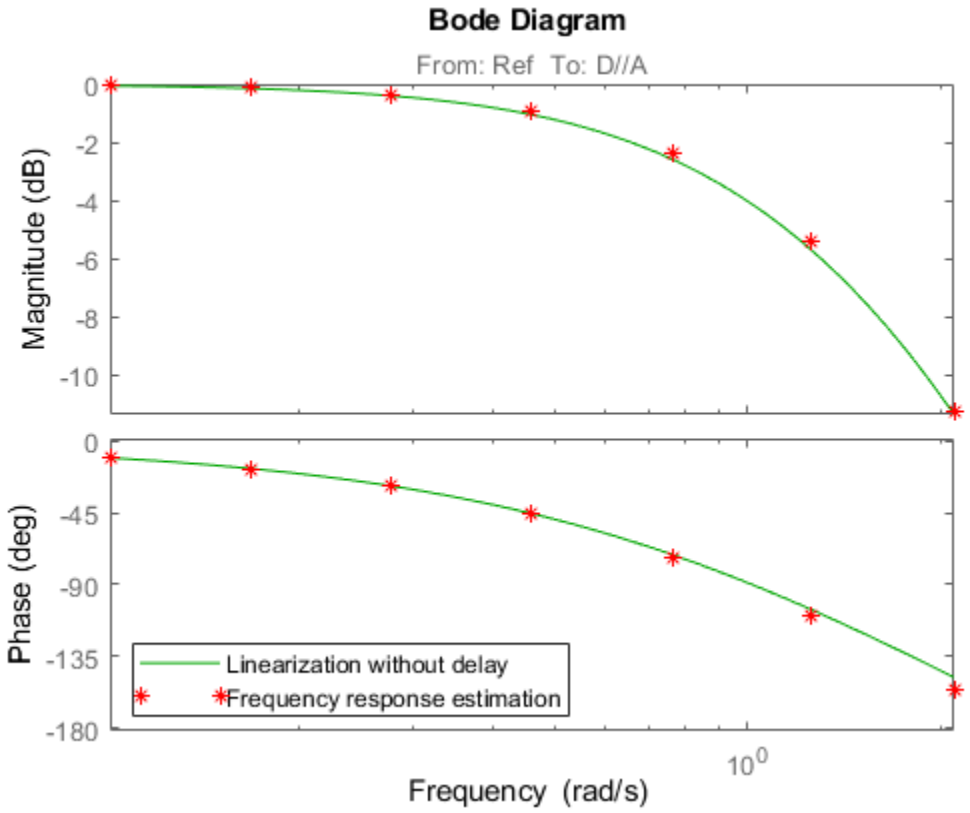
This model includes a continuous-time plant and a discrete-time controller. The D/A block discretizes the plant output with a sample time of 0.1 s. The External Scheduler block triggers the controller to execute with the same period. However, the trigger has an offset of 0.05 s relative to the discretized 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.

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);
```

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. There is a discrepancy in the results between the linearized model and the estimated model, especially at higher frequencies.

Create a function to specify the linearization of the Controller block that includes the time delay. For this example use the linearization specified in `scdAddDelayFcn.m`.

```
function sys = scdAddDelayFcn(BlockData)
    sys = BlockData.BlockLinearization*thiran(0.05,0.1);
end
```

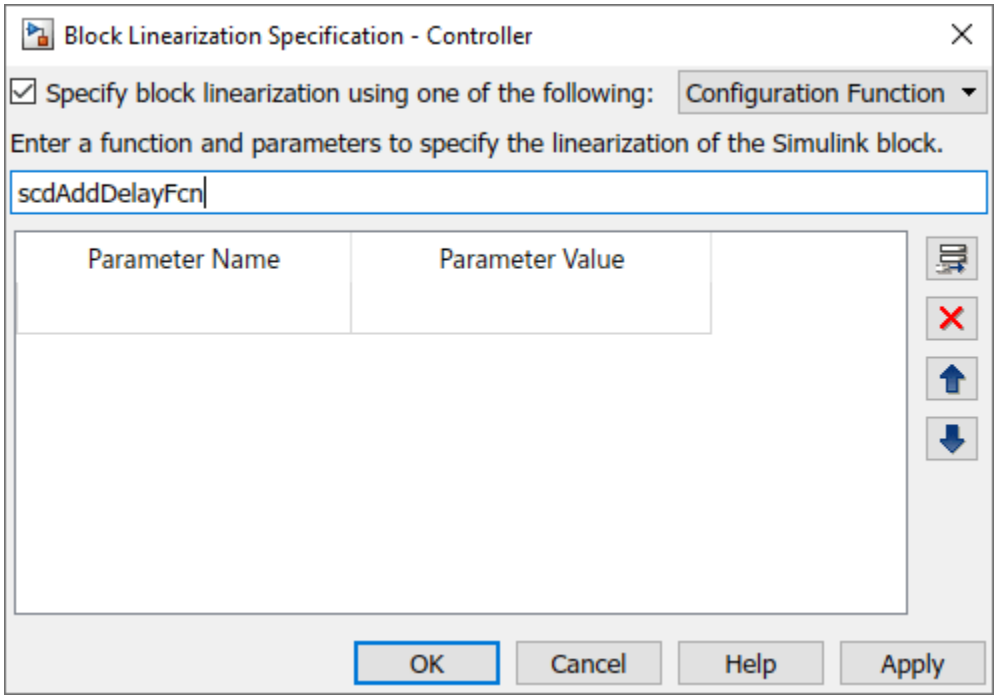
The input to the function, `BlockData`, is a structure that the software creates automatically each time it linearizes the block. When you specify a block linearization configuration function, the software automatically passes `BlockData` to the function. The field `BlockLinearization` contains the current 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 sample time of 0.1 s).

Specify `scdAddDelayFcn` as the linearization for the Controller block.

1. Right-click the Controller block, and select **Linear Analysis > Specify Selected Block Linearization**.
2. In the Block Linearization Specification dialog box, select **Specify block linearization using one of the following**.

3. In the drop-down list, select `Configuration Function`.
4. In the text box, enter the function name `scdAddDelayFcn`. This function has no additional parameters, so leave the parameter table blank.
5. Click **OK**.



Alternatively, you can specify the configuration function programmatically using the following code.

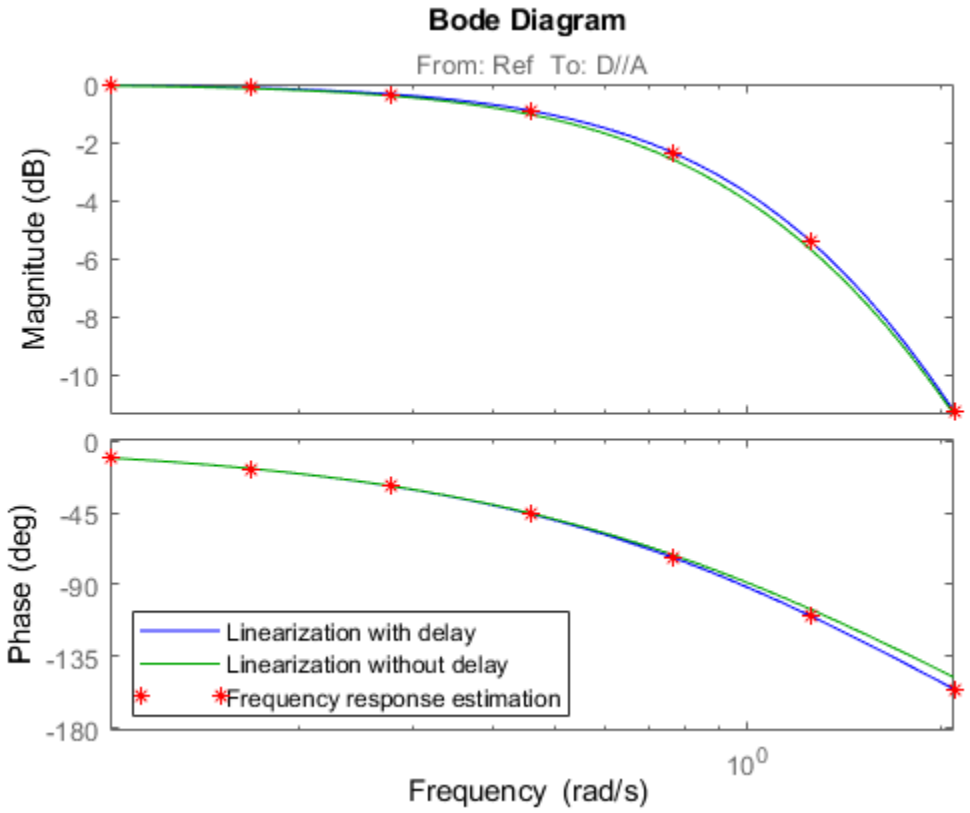
```
block = 'scdFcnCall/Controller';

set_param(block,'SCDEnableBlockLinearizationSpecification','on')
rep = struct('Specification','scdAddDelayFcn',...
             'Type','Function',...
             'ParameterNames','',...
             'ParameterValues','');
set_param(block,'SCDBlockLinearizationSpecification',rep)
```

Linearize the model using the augmented block linearization.

```
sys_d = linearize(mdl,io);
```

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 using the augmented block linearization now accounts for the time delay. This linear model more closely matches the real frequency response of the model.

See Also

`getlinio` | `linearize`

Models with Time Delays

Choose Approximate Versus Exact Time Delays

Simulink Control Design lets you choose whether to linearize models using exact representation or Pade approximation of continuous time delays. How you treat time delays during linearization depends on your nonlinear model.

Simulink blocks that model time delays are:

- Transport Delay block
- Variable Time Delay block
- Variable Transport Delay block
- Delay block
- Unit Delay block

By default, linearization uses Pade approximation for representing time delays in your linear model.

Use Pade approximation to represent time delays when:

- Applying more advanced control design techniques to your linear plant, such as LQR or H-infinity control design.
- Minimizing the time to compute a linear model.

Specify to linearize with exact time delays for:

- Minimizing errors that result from approximating time delays
- PID tuning or loop-shaping control design methods in Simulink Control Design
- Discrete-time models (to avoid introducing additional states to the model)

  The software treats discrete-time delays as internal delays in the linearized model. Such delays do not appear as additional states in the linearized model.

Specify Exact Representation of Time Delays

Before linearizing your model:

- In the **Model Linearizer**:

  1. On the **Linear Analysis** tab, click **More Options**.
  2. In the Options for exact linearization dialog box, in the **Linearization** tab, check **Return linear model with exact delay(s)**.
- At the command line, create a `linearizeOptions` option set, setting the `UseExactDelayModel` to `'on'`.

See Also

`linearizeOptions`

More About

- “Time Delays in Linear Systems”
- “Time-Delay Approximation”
- “Linearize Models with Delays” on page 2-78

Linearize Multirate Models

You can linearize a Simulink model that contains blocks with different sample times using Simulink Control Design software. By default, the linearization tools:

- Convert sample times using a zero-order hold conversion method.
- Create a linearized model with a sample time equal to the largest sample time of the blocks on the linearization path.

You can change either of these behaviors by specifying linearization options, which affects the linearization result.

Change Sample Time of Linear Model

By default, the software sets the sample time to the least common multiple of the nonzero sample times in the model. At this rate, down-sampling is exact for all of the rates in the model. If the default sample time is not appropriate for your application, you can specify a different sample time.

To specify the sample time of the linear model in the **Model Linearizer**:

1. On the **Linear Analysis** tab, click **More Options**.
2. In the Options for exact linearization dialog box, on the **Linearization** tab, in the **Enter sample time (sec)** field, specify the sample time. You can specify any of the following values.
   - `-1` — Set the sample time to the least common multiple of the nonzero sample times in the model.
   - `0` — Create a continuous-time model.
   - Positive scalar — Use the specified value for the sample time.

To specify the sample time of the linear model at the command line, create a `linearizeOptions` option set, and set the `SampleTime` option.

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

You can then use this option set with `linearize` or `slLinearizer`.

Change Linearization Rate Conversion Method

When you linearize models with multiple sample times, such as a discrete controller with a continuous plant, the software uses a rate conversion algorithm to create a single-rate linear model. The default rate conversion method is zero-order hold.

To specify the rate conversion method in the **Model Linearizer**:

1. On the **Linear Analysis** tab, click **More Options**.
2. In the Options for exact linearization dialog box, on the **Linearization** tab, in the **Choose rate conversion method** drop-down list, select one of the following rate conversion methods.

Rate Conversion Method	When to Use
`Zero-Order Hold`	You need exact discretization of continuous dynamics in the time domain for staircase inputs.
`Tustin`	You need good frequency-domain matching between a continuous-time system and the corresponding discretized system, or between an original system and a resampled system.
`Tustin with Prewarping`	You need good frequency-domain matching at a particular frequency between a continuous-time system and the corresponding discretized system, or between an original system and the resampled system.

`Upsampling when possible, Zero-Order Hold otherwise`
`Upsampling when possible, Tustin otherwise`
`Upsampling when possible, Tustin with Prewarping otherwise`
	Upsample discrete states when possible to ensure gain and phase matching of upsampled dynamics. You can only upsample when the new sample time is an integer multiple of the sample time of the original system. Otherwise, the software uses the alternate rate conversion method.

3 If you select either of the following rate conversion methods:

- `Tustin with Prewarping`
- `Upsampling when possible, Tustin with Prewarping otherwise`

then, in the **Enter prewarp frequency** field, specify the prewarp frequency.

To specify the rate conversion method at the command line, create a `linearizeOptions` object, and set the `RateConversionMethod` and `PreWarpFreq` options.

```
opt = linearizeOptions;
opt.RateConversionMethod = 'prewarp';
opt.PreWarpFreq = 100;
```

You can then use this options object with `linearize` or `slLinearizer`.

Note If you use a rate conversion method other than zero-order hold, the converted states no longer have the same physical meaning as the original states. As a result, the state names in the resulting LTI system are `'?'`.

Multirate Linearization Algorithm

This example demonstrates the algorithm that Simulink® Control Design™ software uses to linearize a multirate nonlinear Simulink model.

To illustrate the concepts, the example shows the linearization process using Control System Toolbox™ functions. Then, the same process is repeated using the `linearize` function.

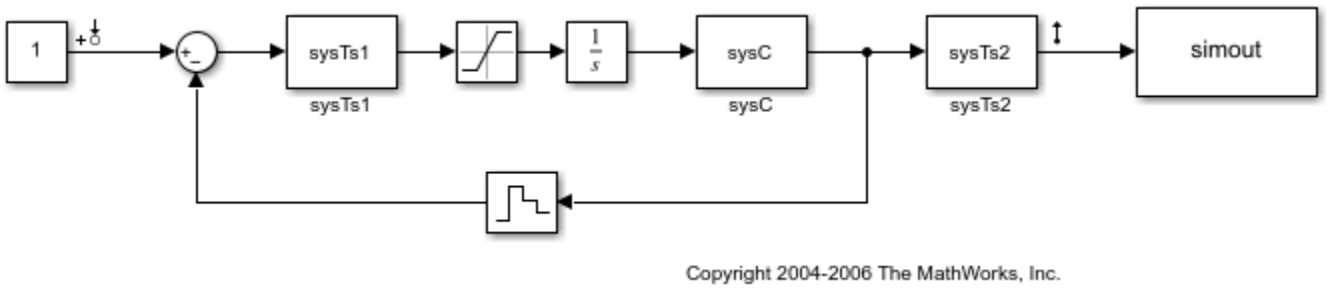
The `scdmrate` Simulink model contains five blocks with various sample times. All linear systems in this model are in zero-pole-gain format.

- `sysC` - Continuous-time linear time-invariant (LTI) system
- `Integrator` - Continuous-time integrator
- `sysTs1` - Discrete-time LTI system with a sample time of 0.01 seconds
- `sysTs2` - Discrete-time LTI system with a sample time of 0.025 seconds
- `Zero-Order Hold` - Block that samples the incoming signal at 0.01 seconds

```
sysC = zpk(-2,-10,0.1);
Integrator = zpk([],0,1);
sysTs1 = zpk(-0.7463,[0.4251 0.9735],0.2212,0.01);
sysTs2 = zpk([],0.7788,0.2212,0.025);
```

View the `scdmrate` model.

```
open_system('scdmrate')
```



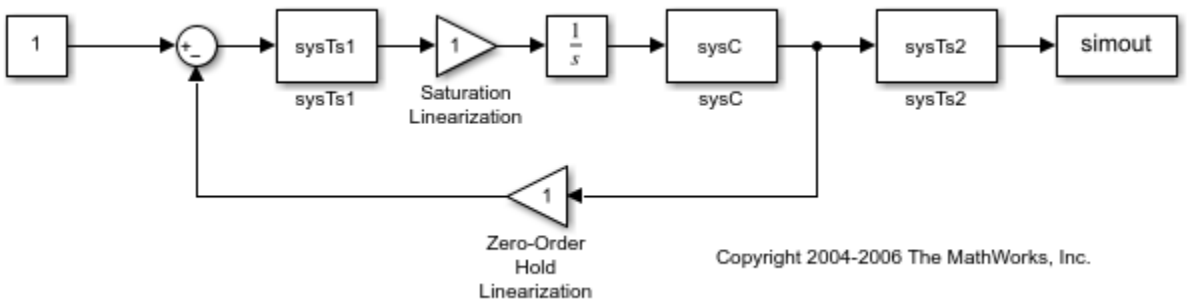
In this example, you linearize the model between the output of the Constant block and the output of the `sysTs2` block.

Linearize Individual Blocks

The first step of the linearization process is to linearize each block in the model. The linearization for the Saturation and Zero-Order Hold blocks is a gain of `1`. Since the LIT blocks are already linear, they are unchanged.

View the updated model with the linearized blocks.

```
open_system('scdmratestep1')
```



Perform Rate Conversion

Since the blocks in the model use different sample times, to create a single-rate linearized model for the system you must first convert the various sample rates to a representative single rate.

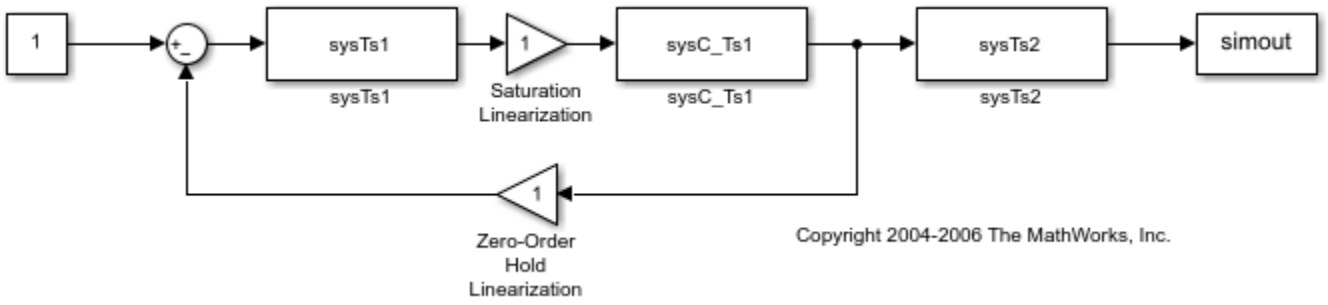
The `linearize` function uses an iterative rate conversion method. The iterations begin with the least common multiple of the sample times in the model. In this example, the sample times are 0, 0.01, and 0.025 seconds, which yields a least common multiple of 0.05.

The first rate-conversion iteration resamples the combination of blocks with the fastest sample rate at the next fastest rate. In this example, the first iteration converts the combination of the linearized continuous-time blocks, `sysC` and `Integrator`, to a sample time of 0.01 using a zero-order hold continuous-to-discrete conversion.

```
sysC_Ts1 = c2d(sysC*Integrator,0.01);
```

The blocks `sysC` and `Integrator` are now replaced by `sysC_Ts1`.

```
open_system('scdmratestep2')
```



The next iteration converts all the blocks with a sample time of 0.01 to a sample time of 0.025. In this example, all the blocks with a sample rate of 0.01 form a closed-loop system. Therefore, before converting their sample rate, the linearization algorithm computes the response of the closed-loop system.

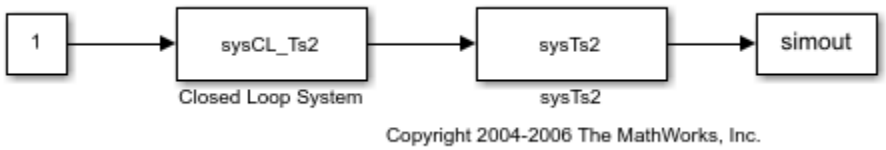
```
sysCL = feedback(sysTs1*sysC_Ts1,1);
```

Next, a zero-order hold method converts the closed-loop system from a sample time of 0.01 seconds to 0.025 seconds.

```
sysCL_Ts2 = d2d(sysCL,0.025);
```

The system `sysCL_Ts2` then replaces the feedback loop in the model.

```
open_system('scdmratestep3')
```



The final iteration resamples the combination of the closed-loop system and the `sysTs2` block from a sample time of 0.025 seconds to 0.05 seconds.

```
sys1 = d2d(sysCL_Ts2*sysTs2,0.05)
```

```
sys1 =

   0.0001057 (z+22.76) (z+0.912) (z-0.9048) (z+0.06495)
  ------------------------------------------------------
  (z-0.01373) (z-0.6065) (z-0.6386) (z-0.8588) (z-0.9754)

Sample time: 0.05 seconds
Discrete-time zero/pole/gain model.
```

Linearize Model Using Simulink Control Design Functions

The `linearize` function implements this iterative process for linearizing multirate models.

To linearize the model, first specify the linearization input and output points.

```
io(1) = linio('scdmrate/Constant',1,'input');
io(2) = linio('scdmrate/sysTs2',1,'openoutput');
```

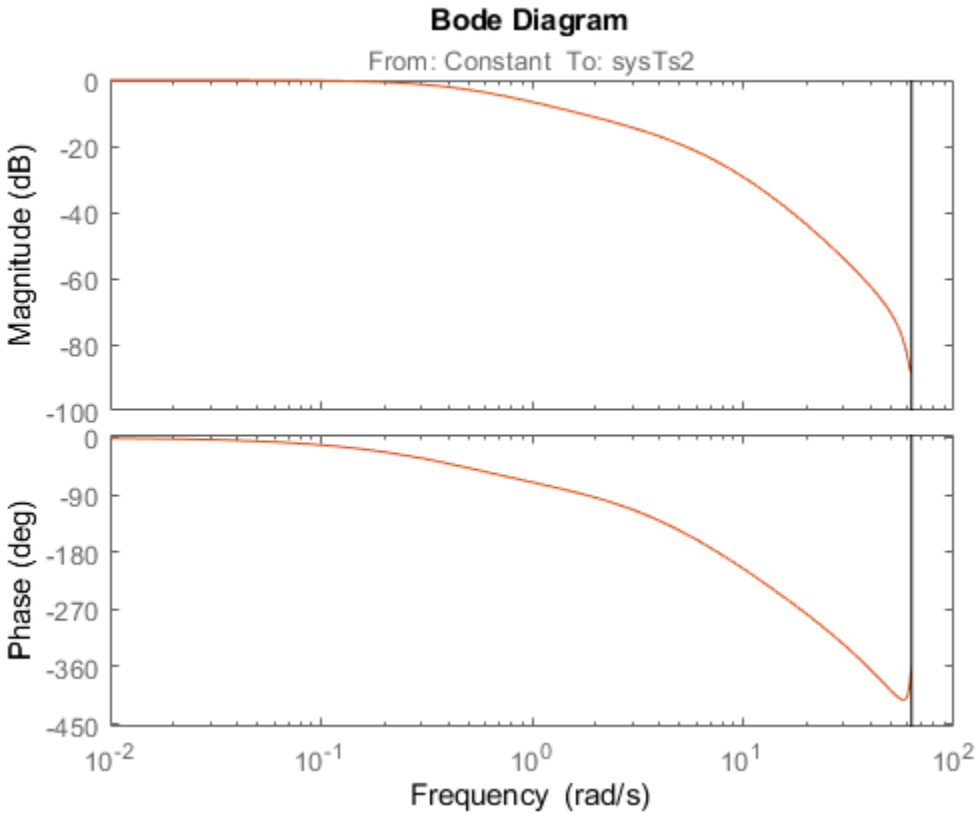
Linearize the model and convert the resulting state-space model to zero-pole-gain format.

```
sys2 = zpk(linearize('scdmrate',io))
```

```
sys2 =

  From input "Constant" to output "sysTs2":
   0.0001057 (z+22.76) (z+0.912) (z-0.9048) (z+0.06495)
  ------------------------------------------------------
  (z-0.6065) (z-0.6386) (z-0.8588) (z-0.9754) (z-0.01373)

Sample time: 0.05 seconds
Discrete-time zero/pole/gain model.
```

This model matches the model computed manually.

```
bode(sys1,sys2)
```



See Also

Apps
Model Linearizer

Functions
`linearize` | `linearizeOptions`

More About

- “Continuous-Discrete Conversion Methods”
- “Linearize Models Using Different Rate Conversion Methods” on page 2-145

Linearize Models Using Different Rate Conversion Methods

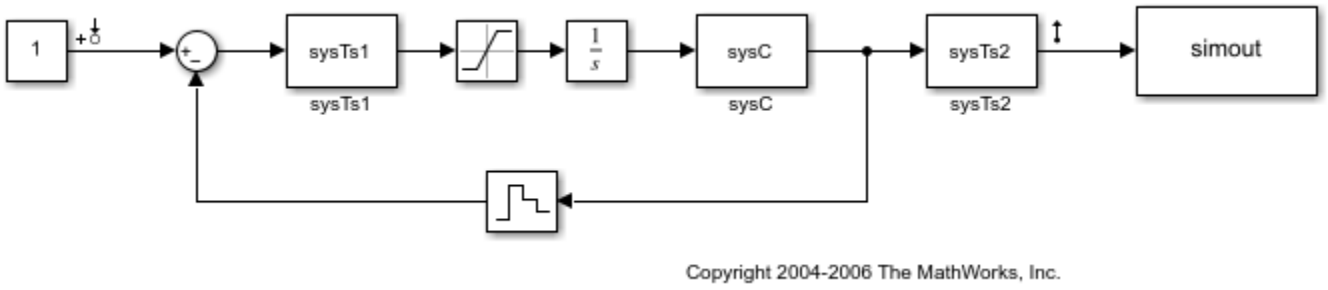
This example shows how to specify the rate conversion method when linearizing a multirate Simulink® model. The choice of rate conversion methodology can affect the resulting linearized model. This example illustrates the extraction of a discrete linear time-invariant (LTI) model using two different rate conversion methods.

The `scdmrate` Simulink model contains five blocks with various sample times. All linear systems in this model are in zero-pole-gain format.

- `sysC` - Continuous-time linear time-invariant (LTI) system
- `Integrator` - Continuous-time integrator
- `sysTs1` - Discrete-time LTI system with a sample time of 0.01 seconds
- `sysTs2` - Discrete-time LTI system with a sample time of 0.025 seconds
- `Zero-Order Hold` - Block that samples the incoming signal at 0.01 seconds

Open the Simulink model.

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

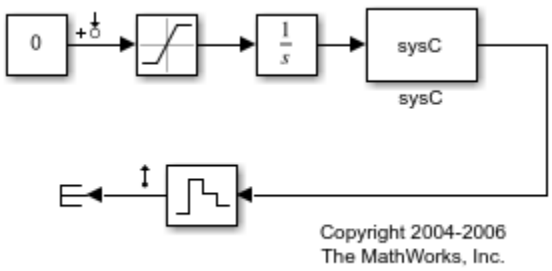


In this example, you linearize the model between the output of the `sysTs1` block and the output of the `Zero-Order Hold` block. To compute an open-loop response, you add a loop opening at the output of the `Zero-Order Hold` block.

```
io(1) = linio('scdmrate/sysTs1',1,'input');
io(2) = linio('scdmrate/Zero-Order Hold',1,'openoutput');
```

Using these linearization points, the linearization effectively results in the linearization of the model `scdmrate_ol`.

```
open_system('scdmrate_ol')
```



When linearizing a model that contains both continuous and discrete signals, the software first converts the continuous signals to discrete signals, using a rate conversion method. To specify the rate conversion method, create a `linearizeOptions` object and set the `RateConversionMethod` property. The default rate conversion method is zero-order hold (`'zoh'`).

```
opt = linearizeOptions;
opt.RateConversionMethod
```

```
ans =

    'zoh'
```

Linearize the model using the default zero-order hold method. Since the linearization includes the `Zero-Order Hold` block, the sample time of the linearization is 0.01 seconds.

```
syszoh = linearize(mdl,io,opt);
```

Change the rate conversion method to the Tustin (Bilinear transformation) method and linearize the model using this method. The sample time of the resulting model is also 0.01 seconds.

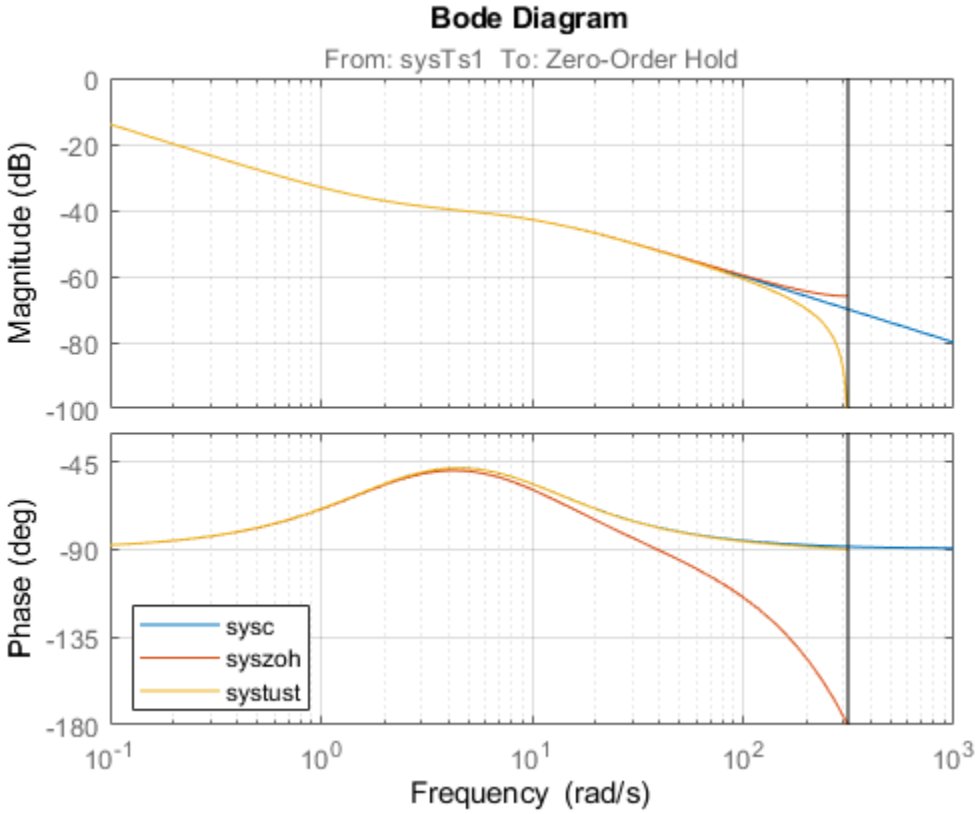
```
opt.RateConversionMethod = 'tustin';
systust = linearize(mdl,io,opt);
```

You can also create a continuous-time linearized model by specifying the sample time as `0` in the `linearizeOptions` object. The rate conversion method creates a single discrete-time linearized model and then converts the discrete-time model to a continuous-time model.

```
opt.SampleTime = 0;
sysc = linearize(mdl,io,opt);
```

The Bode plots for the three linearizations show the effects of the two rate conversion methods. In this example, the Tustin rate conversion method gives the most accurate representation of the phase response of the continuous-time system and the zero-order hold gives the best match to the magnitude response.

```
p = bodeoptions('cstprefs');
p.YLimMode = {'manual'};
p.YLim = {[-100 0];[-180 -30]};
p.Grid = 'on';
bodeplot(sysc,syszoh,systust,p);
legend('sysc','syszoh','systust','Location','SouthWest');
```



Close the models.

```
bdclose('scdmrate')
bdclose('scdmrate_ol')
```

See Also

`linearize` | `linearizeOptions`

Related Examples

- “Linearize Multirate Models” on page 2-139

Change Perturbation Level of Blocks Perturbed During Linearization

Blocks that do not have preprogrammed analytic Jacobians linearize using numerical perturbation. You can modify the size of the state and input signal perturbation levels for your application.

Change Block Perturbation Level

This example shows how to change the perturbation level to the Magnetic Ball Plant block in the magball model. Changing the perturbation level changes the linearization results.

For this model, the state and input signal values are double-precision values. The default perturbation size for the state and input signals in this model is $10^{-5}(1 + |x|)$, where x is the operating point value of the perturbed state or input signal.

Open the model before changing the perturbation level.

```
open_system('magball')
```

Change the perturbation level of the states to $10^{-7}(1 + |x|)$, where x is the state value.

```
blockname = 'magball/Magnetic Ball Plant';
set_param(blockname,'StatePerturbationForJacobian','1e-7');
```

To change the perturbation level of the input signal for this block to $10^{-3}(1 + |x|)$, where x is the input signal value, first obtain the block port handles and get the handle to the input port value.

```
ph = get_param(blockname,'PortHandles');
p_in = ph.Inport(1);
```

Then, set the input port perturbation level.

```
set_param(p_in,'PerturbationForJacobian','1e-3');
```

To obtain the current perturbation level for block states, use the following code.

```
statePerturb = get_param(blockname,'StatePerturbationForJacobian');
```

To obtain the current perturbation level for block input signals, use the following code.

```
inputPerturb = get_param(p_in,'PerturbationForJacobian');
```

When the corresponding state or input signal perturbation level is at its default value, both `statePerturb` and `inputPerturb` are `'auto'`.

Default Perturbation Levels

The default perturbation size for double-precision states and input signals is $10^{-5}(1 + |x|)$, where x is the operating point value of the perturbed state or input signal. For single-precision states and input signals, the default perturbation size is $0.005(1 + |x|)$.

To restore the default perturbation level for block states, use the following code.

```
set_param(blockname,'StatePerturbationForJacobian','auto');
```

To restore the default perturbation level for block input signals, use the following code.

```
set_param(p_in,'PerturbationForJacobian','auto');
```

Perturbation Levels of Integer-Valued Blocks

A custom block that requires integer input ports for indexing might have linearization issues when the block does not support small perturbations in the input value. 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.

See Also

`linearize`

More About

- “Exact Linearization Algorithm” on page 2-175
- “Block Linearization Troubleshooting” on page 4-42

Linearize Blocks with Non-Floating-Point Signals or States

You can linearize blocks that have non-floating-point signals or states 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 non-floating-point signals requires converting all signals to either double-precision or single-precision. This approach only works when your model can run correctly in full double or single precision.

When you have only a few blocks impacted by the non-floating-point data types, you can 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.

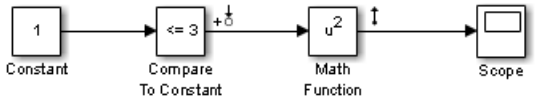
Override Data Type Using Data Type Conversion Block

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.

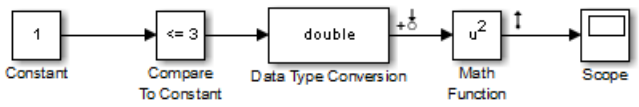
After linearizing the model, remove the Data Type Conversion block from your model.

Note Overriding non-floating-point data types is not appropriate when the model relies on these data types, such as relying on integer data types to perform truncation from floating-point values.

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 with a non-floating-point data type results in a linearization of zero.



In this case, inserting a Data Type Conversion block converts the input signal to the Square block to double precision. Doing so



Overriding Data Types Using Fixed Point Tool

When you linearize a model that contains non-floating-point data types but still runs correctly in full double or single precision, you can override all data types to be either double-precision or single-precision using the **Fixed Point Tool**. Use this approach when you have many non-floating-point precision signals.

After linearizing the model, restore your original settings.

Note Overriding non-floating-point 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. Open the **Fixed Point Tool**. In the Simulink model window, on the **Apps** tab, in the **Apps** gallery, under **Code Generation**, click **Fixed Point Tool**.
2. In the **Data type override** menu, select either **Double** or **Single**
3. Restore settings when linearization is complete.

See Also

`linearize`

More About

- “Change Perturbation Level of Blocks Perturbed During Linearization” on page 2-148
- “Block Linearization Troubleshooting” on page 4-42

Linearize Event-Based Subsystems (Externally Scheduled Subsystems)

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 periodic 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 a periodic Function-Call Subsystem.

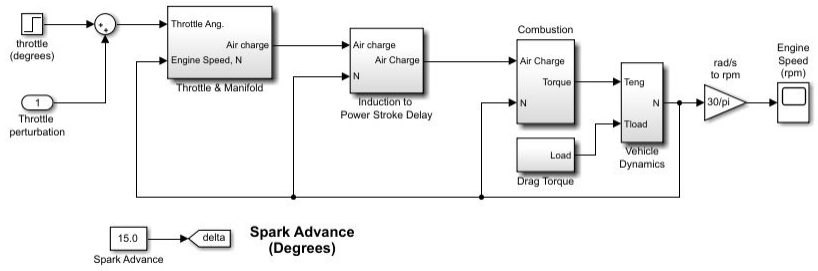
If recasting event-based dynamics does not produce good linearization results, try frequency response estimation. See “Estimate Frequency Response Using Model Linearizer” on page 5-6.

Note If a triggered subsystem is disabled at the current operating condition and has at least one direct passthrough I/O pair, then the subsystem will break the linearization path during linearization. In such a case, specify a block substitution or ensure that the subsystem does not have a passthrough I/O pair.

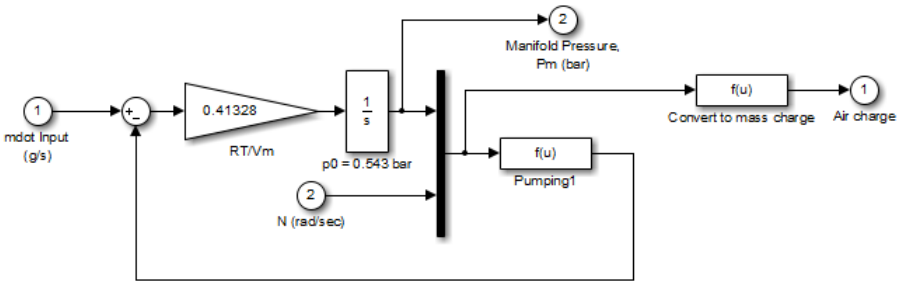
Approximate Event-Based Subsystems Using Curve Fitting (Lump-Average Model)

This example shows how to use curve fitting to approximate event-based dynamics of an engine.

The `scdspeed` model linearizes to zero because the `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.



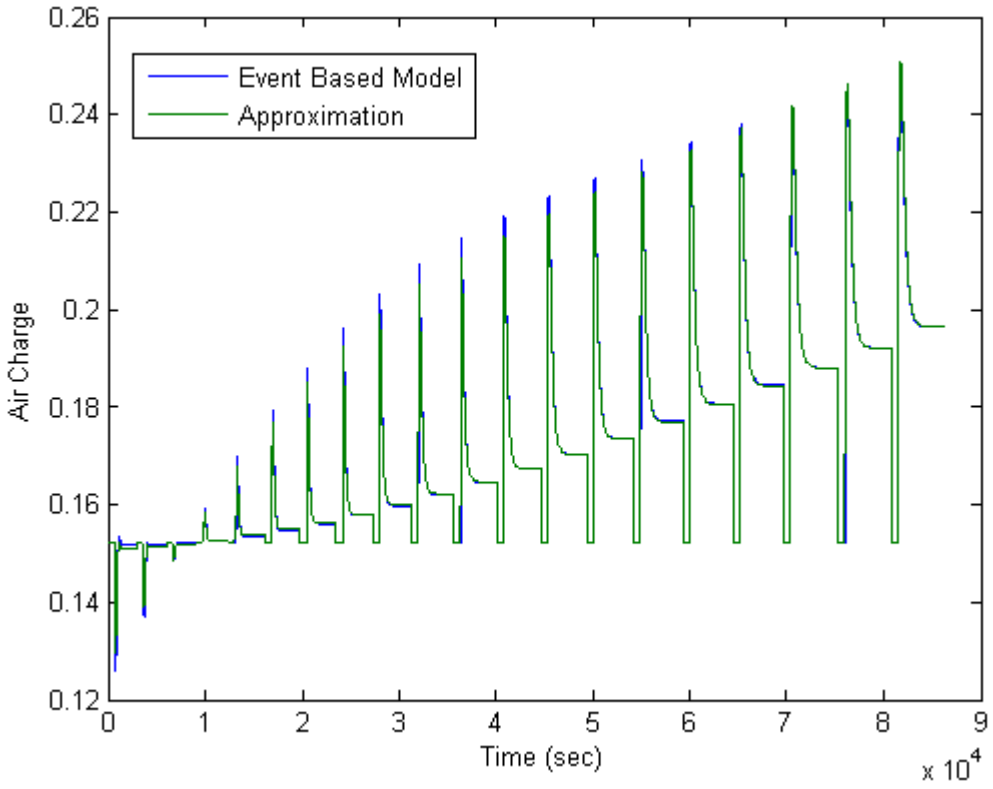
The Convert to mass charge block approximates the relationship between Air Charge, Manifold Pressure, and Engine Speed as a quadratic polynomial.

$$AirCharge = p_1 \times EngineSpeed + p_2 \times ManifoldPressure + p_3 \times (^{M}$$
$$+ p_4 \times ManifoldPressure \times EngineSpeed + p_5$$

If measured data for internal signals is not available, use simulation data from the original model to compute the unknown parameters p_1, p_2, p_3, p_4, and p_5 using a least squares fitting technique.

When you have measured data for internal signals, you can use the Simulink Design Optimization™ software to compute the unknown parameters. See “Engine Speed Model Parameter Estimation” (Simulink Design Optimization) to learn more about computing model parameters, linearizing this approximated model, and designing a feedback controlled for the linear model.

The next figure compares the simulations of the original event-based model and the approximated model. Each of the pulses corresponds to a step change in the engine speed. The size of the step change is between 1500 and 5500. Thus, you can use the approximated model to accurately simulate and linearize the engine between 1500 RPM and 5500 RPM.

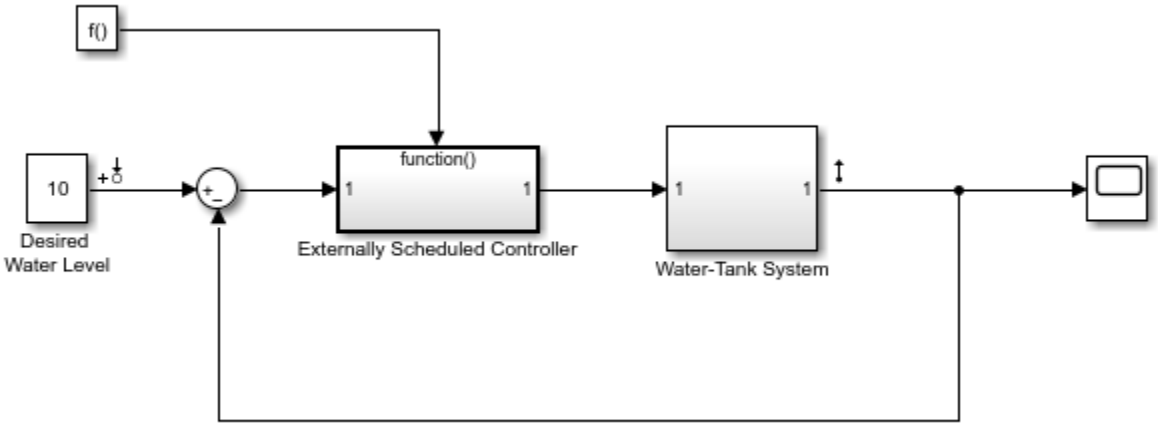


Approximate Event-Based Dynamics Using Periodic Function Call Subsystem

This example shows how to use periodic function call subsystems to approximate event-based dynamics for linearization.

Open the Simulink model.

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



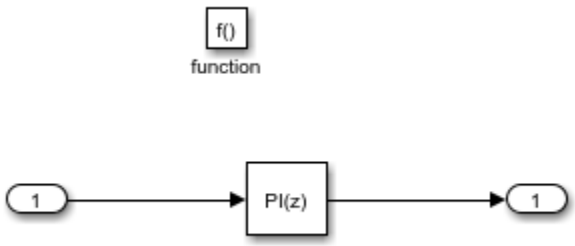
Linearize the model at the model operating point.

```
io = getlinio(mdl);
linsys = linearize(mdl,io)
```

```
linsys =

  D =
                 Desired  Wat
   Water-Tank S             0

Static gain.
```

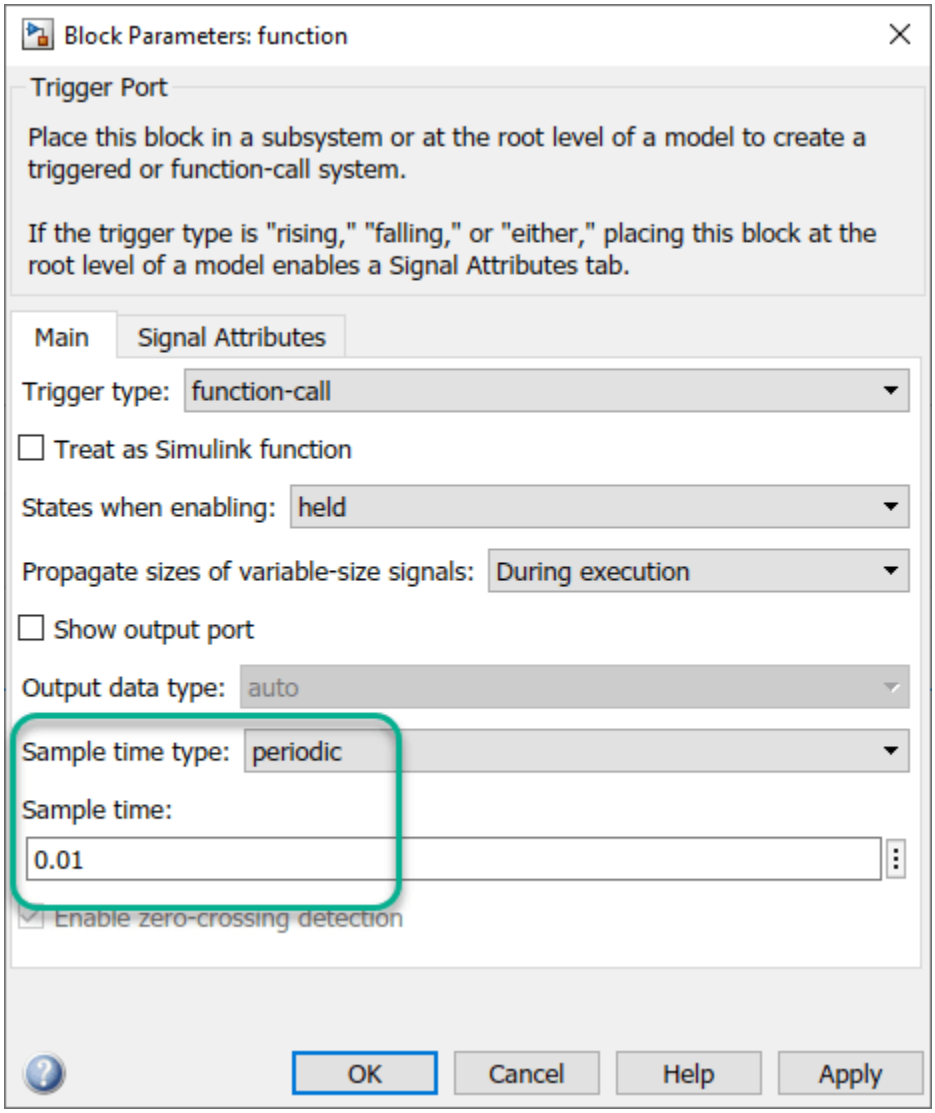
The linearization is zero because the subsystem is not a periodic function call.

Open the Externally Scheduled Controller block, which is a Function-Call Subsystem block.



Open the function block and configure it.

- Set the **Sample time type** parameter to `periodic`.
- Set the **Sample time** parameter to `0.01`, which is the sample time of the controller.

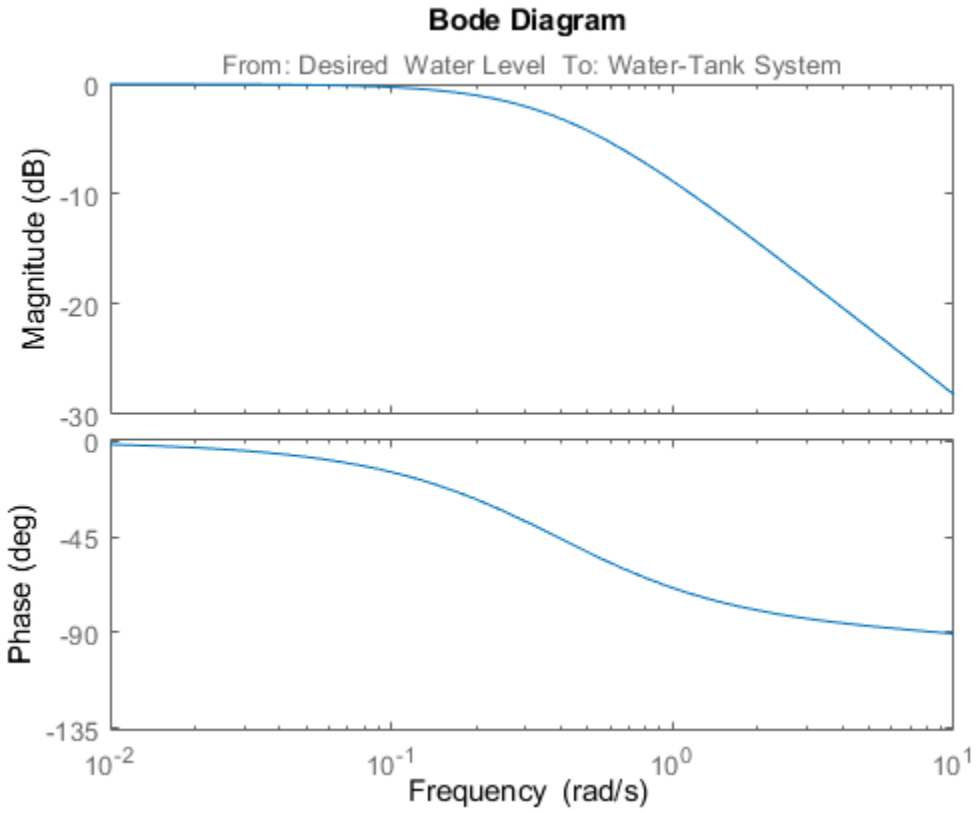


Alternatively, you can configure the function block programmatically using the following code.

```
block = 'scdPeriodicFcnCall/Externally Scheduled Controller/function';
set_param(block,'SampleTimeType','periodic')
set_param(block,'SampleTime','0.01')
```

Linearize the model.

```
linsys2 = linearize(mdl,io);
bode(linsys2)
```



The linearization is no longer zero.

See Also

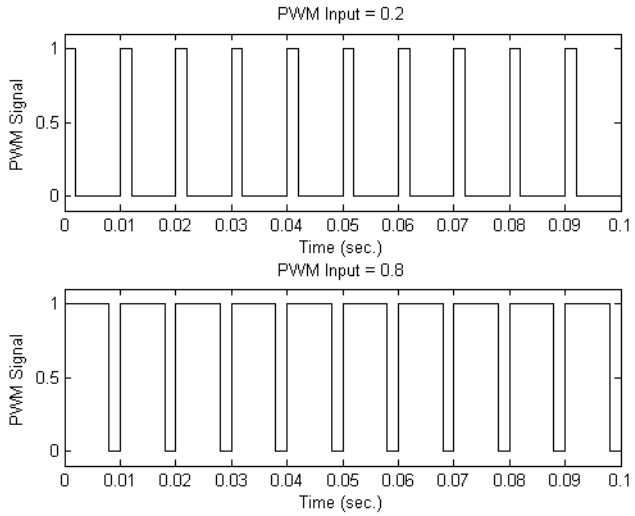
More About

- “Linearize Models with Model References” on page 2-82

Configure Models with Pulse Width Modulation Signals

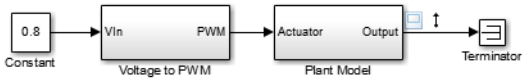
Many industrial applications use pulse width modulation (PWM) signals because such signals are robust in the presence of noise. When using Simulink Control Design software, subsystems that contain PWM signals do not linearize well due to discontinuities in the signal.

The following figure shows two PWM signals. The top plot shows a PWM signal with a 20% duty cycle, which represents a 0.2 V DC signal. The signal is 1 V for 20% of each cycle and 0 V for the remaining 80% of the cycle. The average signal value is 0.2 V. The bottom plot shows a PWM signal with an 80% duty cycle, which represents a 0.8 V DC signal.



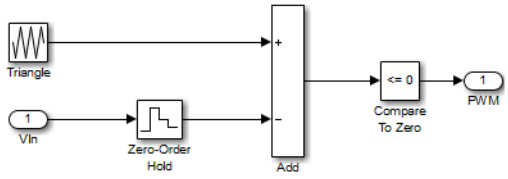
For an example of such a PWM system, open the `scdpwm` model. In this model, a constant signal is converted to a PWM signal using the Voltage to PWM subsystem.

```
open_system('scdpwm')
```



In this model, a constant signal is converted to a PWM signal using the Voltage to PWM subsystem.

```
open_system('scdpwm/Voltage to PWM')
```



When linearizing a model that contains PWM signals there are two effects that produce poor linearization results:

- 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 `scdpwm` model, 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 `Voltage to PWM` subsystem uses the Compare To Zero block. Such comparator blocks do not linearize well due to their discontinuities.

To linearize a model that contains PWM signals, you must replace the linearization of the blocks or subsystems that produce the PWM signals. To do so, use one of the following methods:

- Specify the linearization of the PWM blocks using known values. For example, in “Specify Custom Linearizations for Simulink Blocks” on page 2-128, the linearization of the `Voltage to PWM` subsystem is set to a DC gain of 1.
- Specify the linearization of the PWM subsystem using System Identification Toolbox software. For an example, see “Specify Linearization for Model Components Using System Identification” on page 2-168.
- Specify the linearization of the PWM subsystem using frequency response estimation. For more information on frequency response estimation, see “Estimate Frequency Response Using Model Linearizer” on page 5-6 and “Estimate Frequency Response at the Command Line” on page 5-14.

See Also

Apps
Model Linearizer

Functions
`linearize`

Linearize Simscape Networks

You can linearize models with Simscape components using Simulink Control Design software. Typically, some states in a Simscape network have dependencies on other states through constraints.

Find Steady-State Operating Point

To find a steady-state operating point at which to linearize a Simscape model, you can use:

- Optimization-based trimming — Specify constraints on model inputs, outputs, or states, and compute a steady-state operating point that satisfies these constraints.

  To produce better trimming results for Simscape models, you can use projection-based trim optimizers. These optimizers enforce the consistency of the model initial condition at each evaluation of the objective function or nonlinear constraint function.
- Simulation snapshots — Specify model initial conditions near an expected equilibrium point, and simulate the model until it reaches steady state.

For more information, see “Find Steady-State Operating Points for Simscape Models” on page 1-104.

Specify Analysis Points

To linearize your model, you must specify the portion of the model you want to linearize using linear analysis points; that is, linearization inputs and outputs, and loop openings. You can only add analysis points to Simulink signals.

To add a linearization input or loop opening to the input of a Simscape component, first convert the Simulink signal using a Simulink-PS Converter block.

To add a linearization output or loop opening to the output of a Simscape component, first convert the Simscape signal using a PS-Simulink Converter block.

For more information on adding linear analysis points, see “Specify Portion of Model to Linearize” on page 2-10.

Linearize Model

After you specify a steady-state operating point and linear analysis points, you can linearize your Simscape model using:

- The **Model Linearizer**.
- The `linearize` function.
- An `slLinearizer` interface.

For general linearization examples, see “Linearize Simulink Model at Model Operating Point” on page 2-54 and “Linearize at Trimmed Operating Point” on page 2-67.

Troubleshoot Simscape Network Linearizations

Simscape networks can commonly linearize to zero when a set of the system equation Jacobians are zero at a given operating condition. Usually, poor initial conditions of the network states cause these zero linearizations.

Zero Linearization Example

Consider a system where the mass flow rate from a variable orifice is controlling the position of a piston. The mass flow rate equation of the variable orifice is:

$$q = C_d A \sqrt{\frac{2}{\mu}} \left(\frac{p}{p^2 + p_{cr}^2} \right)^{0.25}$$

Where:

- q is the mass flow rate.
- C_d is the discharge coefficient.
- A is the area of the variable orifice opening.
- μ is the fluid density.
- p is the pressure drop across the orifice, $p = p_a - p_b$.
- p_{cr} is the critical pressure, which is a function of p_a and p_b.

The control variable for this system is the orifice area, A, which controls the mass flow rate. The Jacobian of the mass flow rate with respect to the control variable is:

$$\frac{\partial q}{\partial A} = C_d \sqrt{\frac{2}{\mu}} \left(\frac{p}{p^2 + p_{cr}^2} \right)^{0.25}$$

The linearized mass flow rate equation is:

$$\bar{q} = C_d\sqrt{\frac{2}{\mu}}\left(\frac{p}{p^2 + p_{cr}^2}\right)^{0.25}\bar{A} + \frac{\partial q}{\partial \mu}\bar{\mu}$$

$$+\left(\frac{\partial q}{\partial p_{cr}}\frac{\partial p_{cr}}{\partial p_a} + \frac{\partial q}{\partial p}\right)\bar{p}_a + \left(\frac{\partial q}{\partial p_{cr}}\frac{\partial p_{cr}}{\partial p_b} - \frac{\partial q}{\partial p}\right)\bar{p}_b$$

where $\bar{\cdot}$ represents a deviation from the nominal variable.

In the linearized equation, if the nominal pressure drop p across the orifice is zero, then $\bar{A}$ has no influence on $\bar{q}$. That is, if the instantaneous pressure drop across the orifice is zero, the orifice area has no influence on the mass flow rate. Therefore, you cannot control the piston position using the orifice area control variable.

To avoid this condition, linearize the model about an operating point where the pressure drop over the orifice is nonzero ($p_a \neq p_b$).

Troubleshooting Tips

To fix linearization problems caused by poor initial conditions of network states, you can:

1. Linearize the system at a snapshot operating point or trimmed operating point. When possible, this approach is recommended.
2. Find and modify the problematic states of the operating point. This option can be difficult for models with many states.

Using the first approach, you can ensure that the model states are consistent via the Simulink and Simscape simulation engine. Simscape initial conditions are not necessarily in a consistent state. The Simscape engine places them in a consistent state during simulation and for trimming using the Simscape trim solvers.

A common workflow is to simulate your model, observe at what time the model satisfies the operating condition at which you want to linearize, then take a simulation snapshot. Alternatively, you can trim the model about the condition you are interested in. In either case, the network states are in a consistent condition, which solves most poor linearization issues.

Using the second approach, you search through the physical network states to find conditions that can create a zero Jacobian. This approach can require some intuition about the dynamics of the physical components in your model. As a starting point, search for states that are zero and that interact directly with nonlinear physical elements, such as the variable orifice in the preceding example.

To search the physical states, you can use the Linearization Advisor, which collects diagnostic information during linearization. The Linearization Advisor does not provide diagnostic information on a component-level basis for Simscape networks. Instead, it groups diagnostic information for multiple Simscape components together.

1. Linearize your model with the Linearization Advisor enabled, and extract the `LinearizationAdvisor` object.

```
opt = linearizeOptions('StoreAdvisor',true);
[linsys,linop,info] = linearize(mdl,io,op,opt);
advisor = info.Advisor;
```

2. Create a custom query object, and search the diagnostic information for Simscape blocks.

```
qSS = linqueryIsBlockType('simscape');
advSS = find(advisor,qSS);
```

3. To find problematic state values, check the block operating point in each `BlockDiagnostic` object.

```
advSS.BlockDiagnostics(i).OperatingPoint.States
```

Once you find a problematic state, you can change the value of the state in the model operating point, or create an operating point using `operpoint`.

You can also search the Linearization Advisor in the **Model Linearizer**. For more information, see “Find Blocks in Linearization Results Matching Specific Criteria” on page 4-37.

See Also

Apps
Model Linearizer

Functions
`linearize` | `slLinearizer`

More About

- “Specify Portion of Model to Linearize” on page 2-10
- “Find Steady-State Operating Points for Simscape Models” on page 1-104
- “Linearize Simulink Model at Model Operating Point” on page 2-54

Linearize Sparse Models

You can obtain a sparse linear model from a Simulink model that contains a Sparse Second Order or Descriptor State-Space block. Linearizing such models produces one of the following sparse state-space objects.

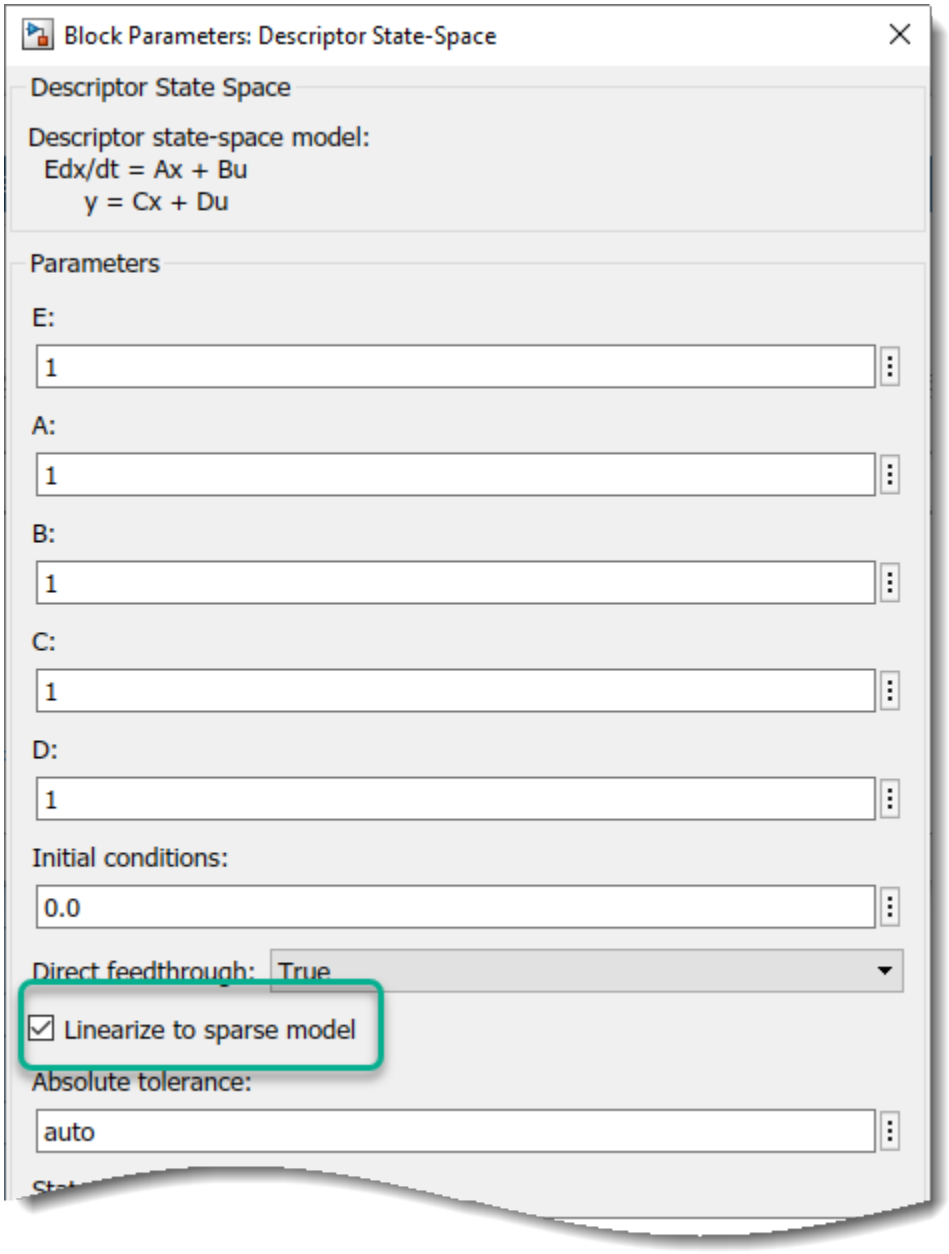
- `mechss` model when you use a Sparse Second Order block
- `sparss` model when you use a Descriptor State-Space block and configure the block to linearize to a sparse model

For more information on sparse models, see “Sparse Model Basics”.

For a command-line linearization example, see “Linearize Simulink Model to a Sparse Second-Order Model Object”.

The Sparse Second Order block always linearizes to a `mechss` model. As a result, the overall linearized model is a second-order sparse model when this block is present.

To linearize a Descriptor State-Space block to a sparse model, select the **Linearize to sparse model** block parameter.



You can also select this parameter programmatically. Here, `blockpath` is the path to the Descriptor State-Space block.

```
set_param(blockpath,'LinearizeToSparse','on')
```

The **Linearize to sparse model** parameter of the Descriptor State-Space block is ignored when you specify a custom linearization for the block.

Linearize Sparse Models at the Command Line

You can linearize a model that contains a sparse block at the command line using either the `linearize` function or an `slLinearizer` interface.

To analyze the resulting linearized model in the:

- Time domain, you must specify a time vector or a final simulation time. For example, plot the step response of linearized sparse model `linsys` for 10 seconds using 100 sample points.

  ```
  t = linspace(0,10,1000);
  step(sys,t)
  ```

- Frequency domain, you must specify a frequency vector. For example, plot the Bode response of linearized sparse model `linsys` from 10^1 to 10^5 rad/sec using 1000 logarithmically spaced frequency points.

  ```
  w = logspace(1,5,1000);
  bode(linsys,w)
  ```

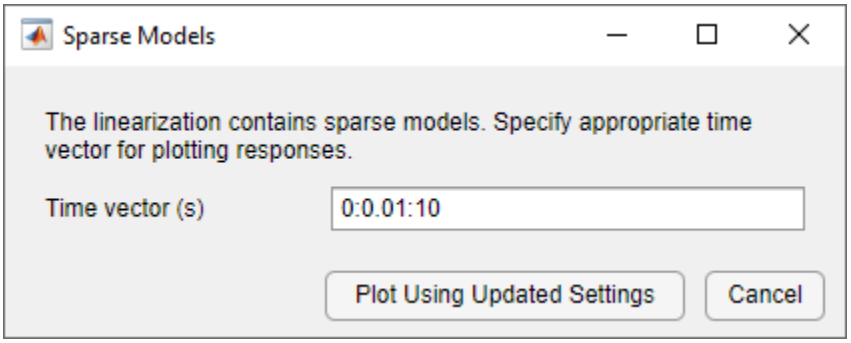
For more information on analyzing sparse models at the command line, see “Sparse Model Basics”.

Linearize Sparse Models Using Model Linearizer

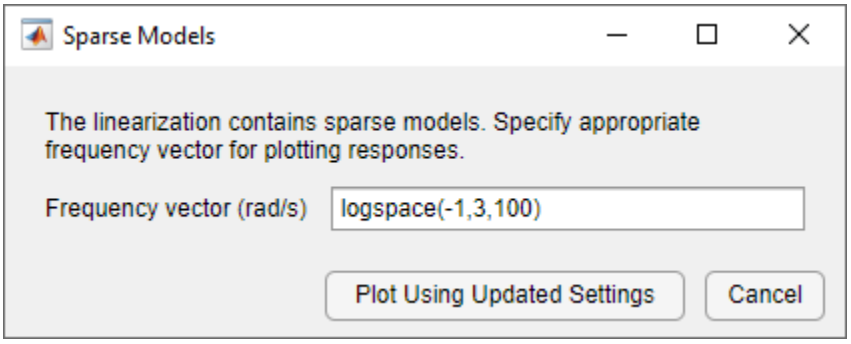
You can interactively linearize a model that contains a sparse block using the **Model Linearizer** app.

Each time you plot a time-domain or frequency-domain response of the resulting linearized model, **Model Linearizer** prompts you to enter a time or frequency vector for plotting. For example:

- Time domain — Plot response for 10 seconds in 0.01 second increments.



- Frequency domain — Plot response from 10^{-1} to 10^3 rad/sec using 100 logarithmically spaced frequency points.



You cannot change the time or frequency vector for an existing sparse model response plot. Instead, create a new plot for the same sparse model using an updated time or frequency vector.

To view the structure of the linearized sparse model, export the model to the MATLAB workspace and use the `spy` and `showStateInfo` functions.

Limitations

Sparse linearization has the following limitations.

- If you disable block reduction when linearizing a sparse model, the resulting linear model is a dense `ss` model object.
- Due to simulation limitations, snapshot linearization might not work when your model contains a Descriptor State-Space or Sparse Second Order block.
- Sparse linearization is incompatible with block substitutions involving tunable or uncertain models, such as `genss` or `uss`, respectively.
- When analyzing linearized sparse systems, **Pole-Zero Map** and **I/O Pole-Zero Map** plots are not supported.

See Also

Blocks
Sparse Second Order | Descriptor State-Space

Functions
`sparss` | `mechss` | `linearize` | `slLinearizer`

Apps
Model Linearizer

Related Examples

- “Sparse Model Basics”
- “Linearize Simulink Model to a Sparse Second-Order Model Object”

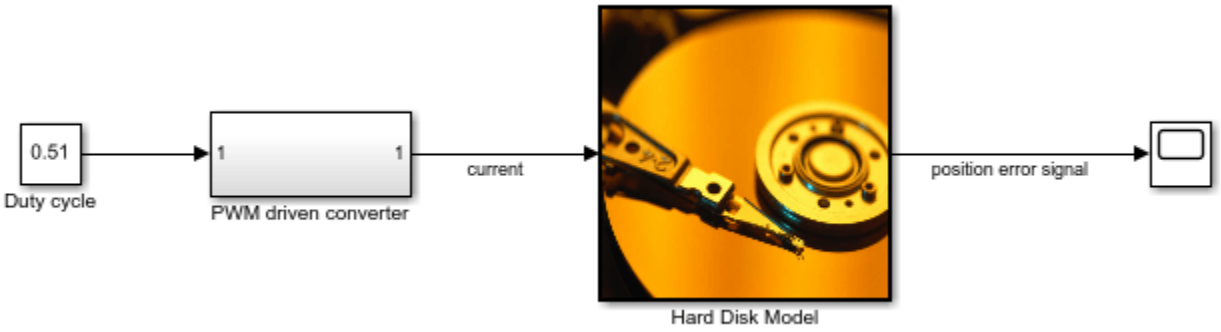
Specify Linearization for Model Components Using System Identification

This example shows how to specify the linearization for a model component that does not linearize well using a linear model identified using the System Identification Toolbox™. This example requires Simscape™ Electrical™ software.

Linearize 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 “Digital Servo Control of a Hard-Disk Drive”.

PWM-driven circuits usually have high frequency switching components, such as the MOSFET transistor in this model, whose average behavior is not well defined. Therefore, finding an exact linearization of this type of circuit is problematic. When you linearize the model from the duty cycle input to the position error, the result is zero.

The Simscape solver in this model is configured to run in local solver mode. When linearizing the model, first turn off the local solver.

```
SimscapeSolver = [model '/PWM driven converter/Solver Configuration'];
set_param(SimscapeSolver,'UseLocalSolver','off')
```

Linearize the model.

```
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.
```

As expected, the PWM components cause the system to linearize to zero.

Turn the local solver back on for simulation.

```
set_param(SimscapeSolver,'UseLocalSolver','on')
```

Find Linear Model for PWM Component

You can estimate the frequency response of the PWM-driven current source and use the result to identify a linear model.

The current signal has a discrete sample time of `1e-7`. Therefore, you need to use a sinestream signal with a fixed sample time as your estimation input signal. Create a signal that has frequencies between 2,000 and 200,000 rad/s.

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

Define the input and output points for the PWM-driven circuit, and run the frequency response estimation using the sinestream input signal.

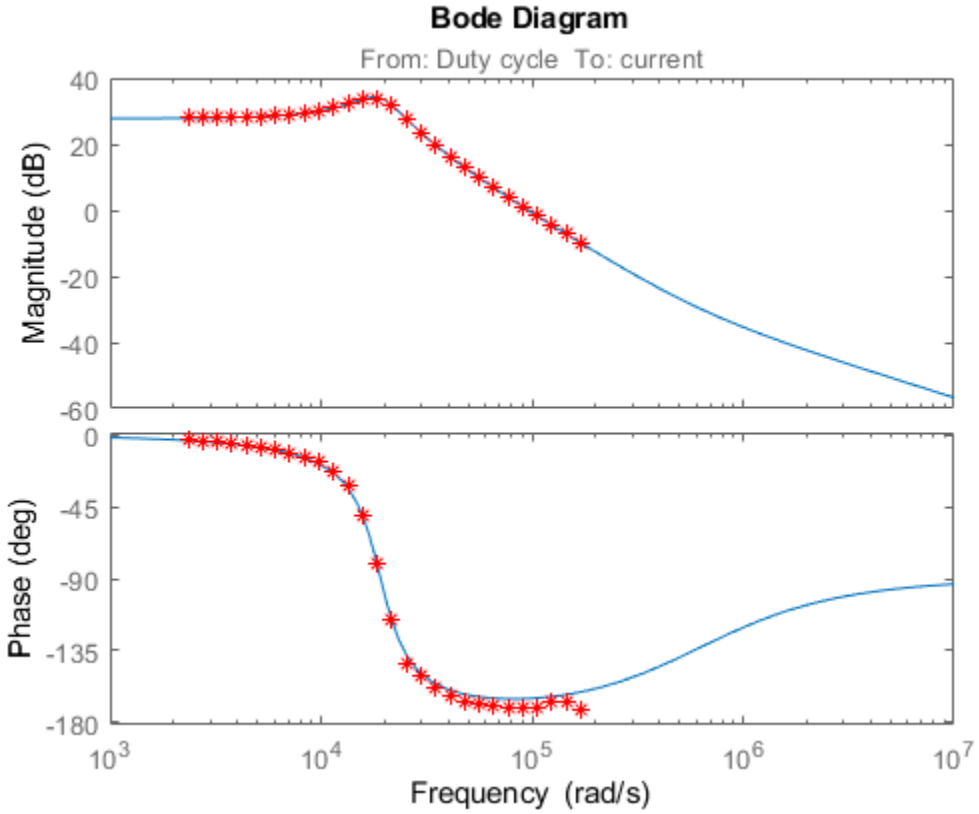
```
pwm_io(1) = linio('scdpwmharddrive/Duty cycle',1,'input');
pwm_io(2) = linio('scdpwmharddrive/PWM driven converter',1,'openoutput');
sysfrd = frestimate(model,pwm_io,idinput);
```

To identify a second-order model using the frequency response data, use the System Identification Toolbox function `tfest`.

```
sysid = ss(tfest(idfrd(sysfrd),2));
```

Compare the identified model to the original frequency response data.

```
bode(sysid,sysfrd,'r*')
```



To estimate the frequency response, you used frequencies between 2,000 and 200,000 rad/s. The identified model has a flat magnitude response for frequencies smaller than 2,000 rad/s. However, the previous estimation did not include those frequencies.

To verify whether the frequency response is flat for lower frequencies, estimate the response using a sinestream input signal with frequencies of 20 and 200 rad/s.

```
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
```

The combination of the fast sample time of `1e-7` seconds (10 MHz sampling frequency) and the lower frequencies creates high `SamplesPerPeriod` values for the input signal. In this case, considering

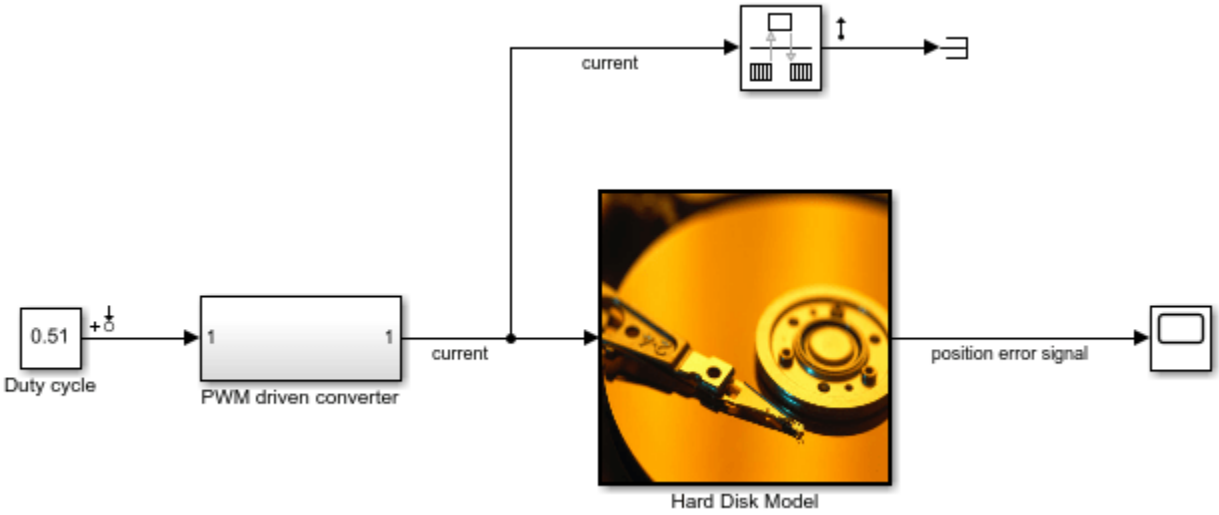
that each frequency has four periods, frequency response estimation would log output data with around 14 million samples.

Since such a high sampling rate is not necessary for analyzing 20 and 200 rad/s frequencies, you can avoid memory issues by increasing the sample time to `1e-4`.

```
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 sample time, resample the output data point using a rate transition block as in the modified model.

```
modellow = 'scdpwmharddrive_lowfreq';
open_system(modellow)
```



You can estimate a frequency response for the low frequencies using the following command.

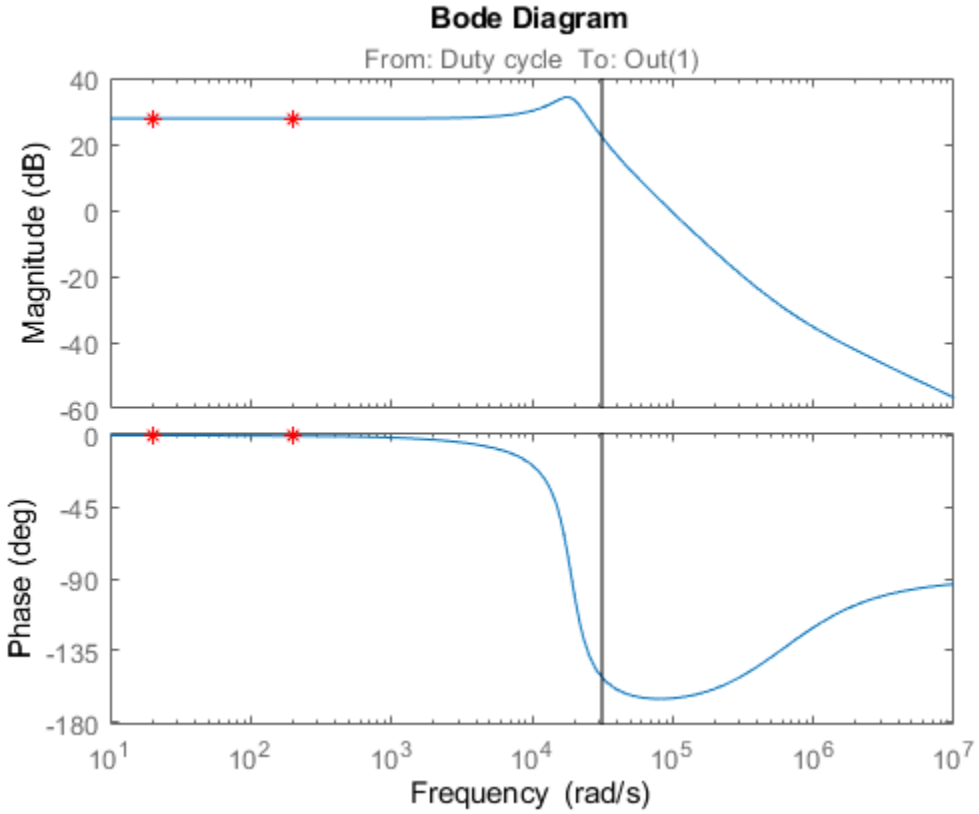
```
sysfrdlow = frestimate(modellow,getlinio(modellow),inputlow);
```

The frequency response estimation can take several minutes. For this example, load the estimation results.

```
load scdpwmharddrive_lowfreqresults
```

To validate the low-frequency response of the identified model, compare the estimated result with the identified model.

```
bode(sysid,sysfrdlow,'r*')
```



Close the low-frequency model.

```
bdclose(modellow)
```

Specify Linearization for PWM Component

To specify the linearization of the PWM-driven component using the validated identified model, specify the linearization of the PWM-driven converter subsystem.

To do so, first enable block linearization specification for the converter block.

```
pwmblock = 'scdpwmharddrive/PWM driven converter';
set_param(pwmblock,'SCDEnableBlockLinearizationSpecification','on');
```

Specify `sysid` as the block linearization using a MATLAB® expression.

```
rep = struct('Specification','sysid',...
             'Type','Expression',...
             'ParameterNames','',...
             'ParameterValues','');
set_param(pwmblock,'SCDBlockLinearizationSpecification',rep);
```

Set the sample time of the duty cycle reference signal to that of the plant.

```
set_param('scdpwmharddrive/Duty cycle','SampleTime','Ts_plant');
```

Linearize the model.

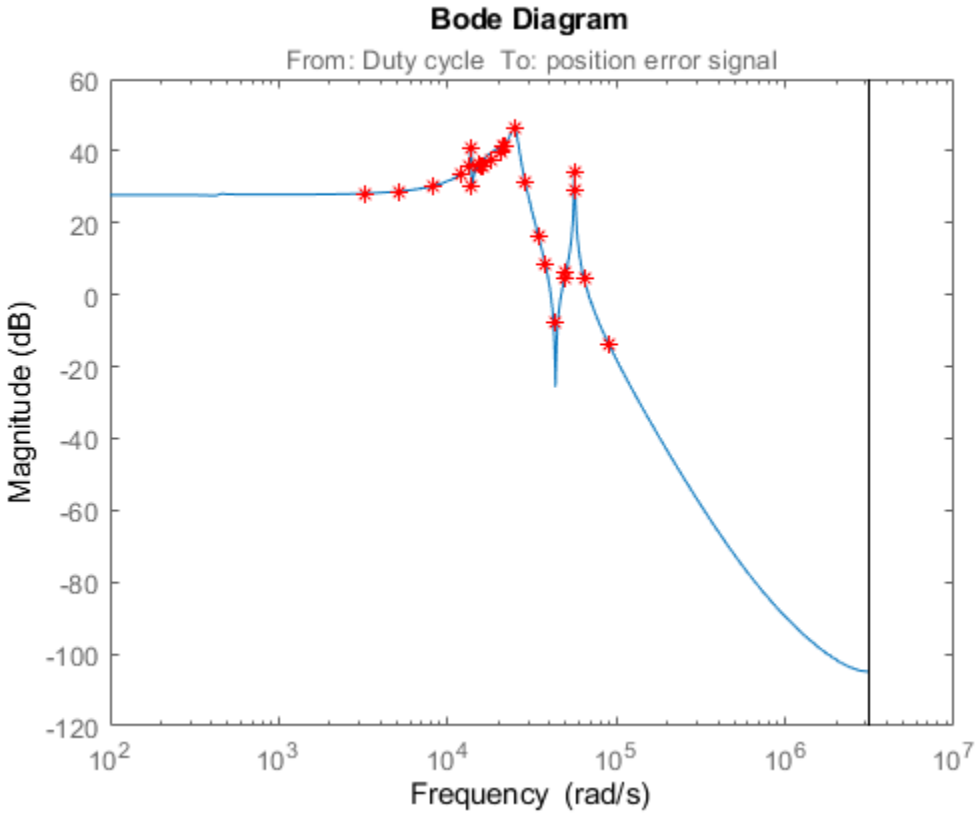
```
set_param(SimscapeSolver,'UseLocalSolver','off')
sys = linearize(model,io);
set_param(SimscapeSolver,'UseLocalSolver','on')
```

You can validate the overall linearization result by estimating the frequency response using the following commands.

```
valinput = frest.Sinestream(sys);
valinput = fselect(valinput,3e3,1e5);
valinput.Amplitude = 0.1;
sysval = frestimate(model,io,valinput);
```

The frequency response estimation can take several minutes. For this example, load the estimation results.

```
load scdpwmharddrive_valfreqresults
bodemag(sys,sysval,'r*')
```



The linearization result is accurate and all the resonances exist in the actual dynamics of the model.

Close the model.

```
bdclose('scdpwmharddrive')
```

See Also

`frestimate` | `tfest`

More About

- “Configure Models with Pulse Width Modulation Signals” on page 2-158

Exact Linearization Algorithm

Simulink Control Design software linearizes models using a block-by-block approach. The software individually linearizes each block in your Simulink model and produces the linearization of the overall system by combining the individual block linearizations.

The software determines the input and state levels for each block from the operating point, and requests the Jacobian for these levels from each block.

For some blocks, the software cannot compute an analytical linearization. For example:

- Some nonlinearities do not have a defined Jacobian.
- Some discrete blocks, such as state charts and triggered subsystems, tend to linearize to zero.
- Some blocks do not implement a Jacobian.
- Custom blocks, such as S-Function blocks and MATLAB Function blocks, do not have analytical Jacobians.

You can specify a custom linearization for any such blocks for which you know the expected linearization. If you do not specify a custom linearization, the software finds the linearization by perturbing the block inputs and states and measuring the response to these perturbations. For more information, see “Perturbation of Individual Blocks” on page 2-179.

Continuous-Time Models

Simulink Control Design software lets you linearize continuous-time nonlinear systems. The resulting linearized model is in state-space form.

In continuous time, the state space equations of a nonlinear system are:

$$\dot{x}(t) = f(x(t), u(t), t)$$
$$y(t) = g(x(t), u(t), t)$$

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:

$$A = \frac{\partial f}{\partial x}\Big|_{t_0, x_0, u_0} \qquad B = \frac{\partial f}{\partial u}\Big|_{t_0, x_0, u_0}$$

$$C = \frac{\partial g}{\partial x}\Big|_{t_0, x_0, u_0} \qquad D = \frac{\partial g}{\partial u}\Big|_{t_0, x_0, u_0}$$

This linear time-invariant approximation to the nonlinear system is valid in a region around the operating point at $t=t_0$, $x(t_0)=x_0$, and $u(t_0)=u_0$. In other words, if the values of the system states, $x(t)$, and inputs, $u(t)$, are close enough to the operating point, the system behaves approximately linearly.

The transfer function of the linearized model is the ratio of the Laplace transform of $\delta y(t)$ and the Laplace transform of $\delta u(t)$:

$$P_{lin}(s) = \frac{\delta Y(s)}{\delta U(s)}$$

Multirate Models

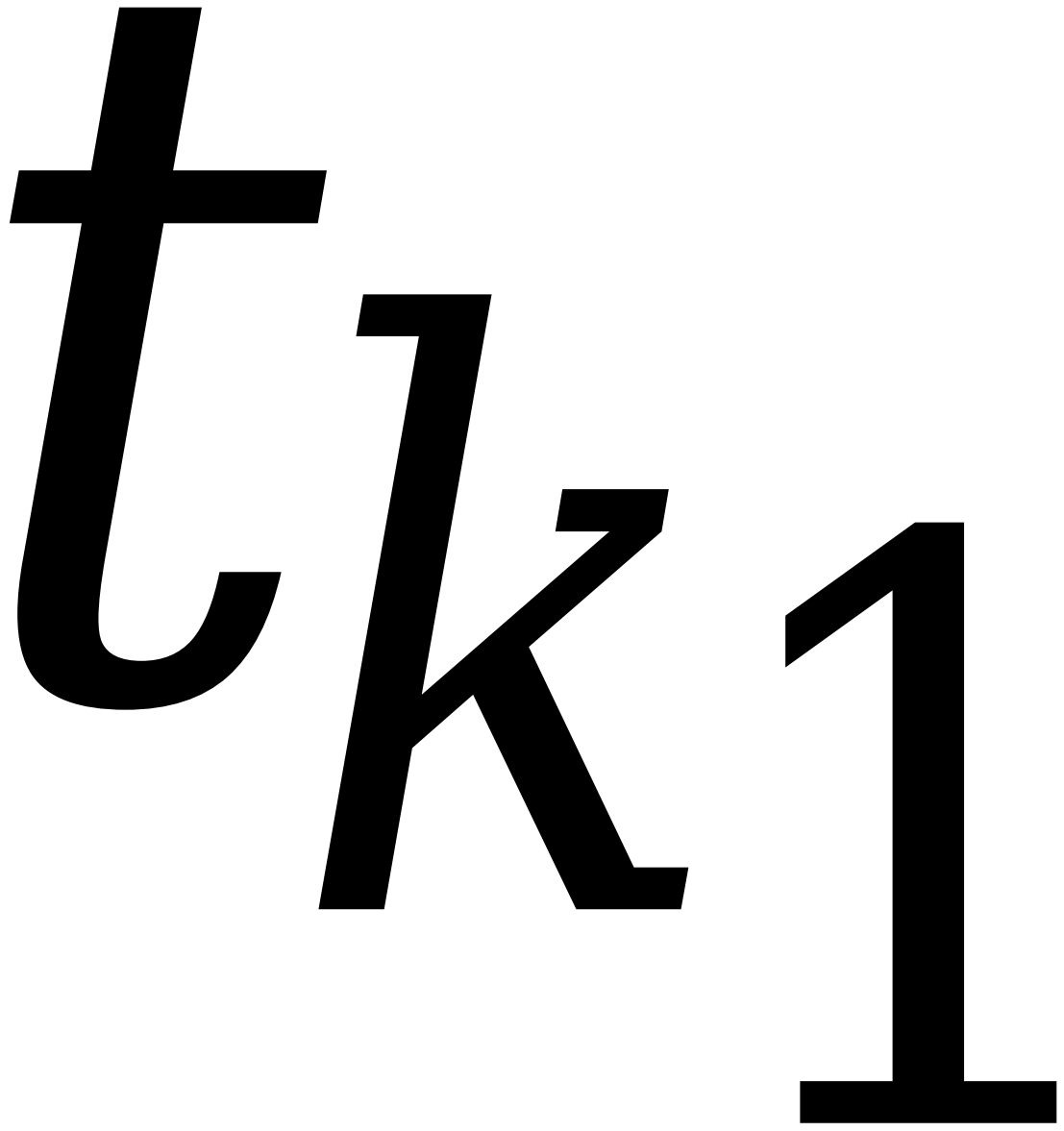
Simulink Control Design software lets you linearize multirate nonlinear systems. The resulting linearized model is in state-space form.

Multirate models include states with different sampling rates. In multirate models, the state variables change values at different times and with different frequencies. Some of the variables might change continuously.

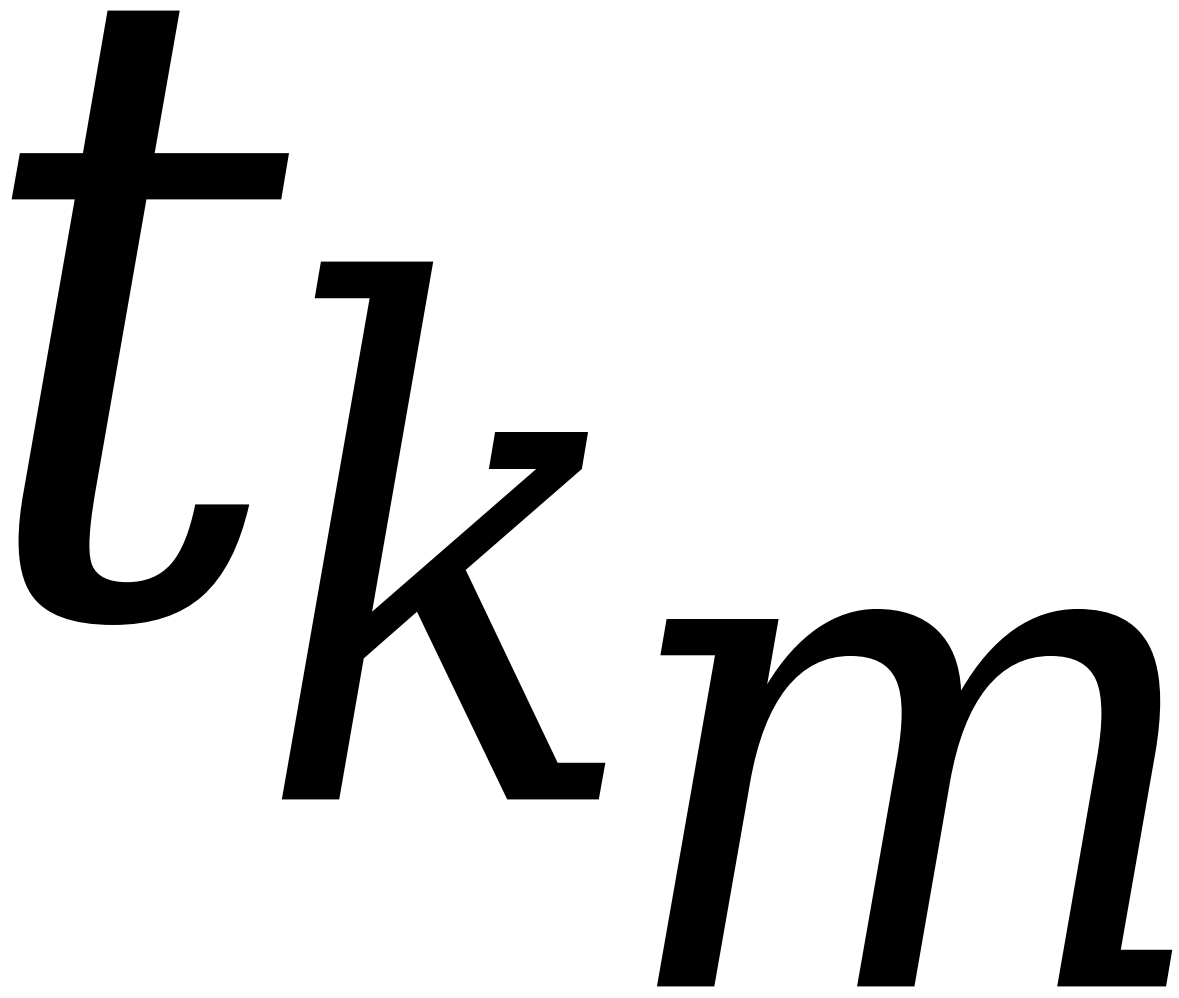
The general state-space equations of a nonlinear, multirate system are:

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

where $k_1,\ldots, k_m$ are integer values and



,...,



are discrete times.

The linearized equations that approximate this nonlinear system as a single-rate discrete model are:

$$\delta x_{k+1} \approx A\delta x_k + B\delta u_k$$

$$\delta y_k \approx C\delta x_k + D\delta u_k$$

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 “Linearize Multirate Models” on page 2-139.

Perturbation of Individual Blocks

Simulink Control Design software linearizes blocks that do not have a preprogrammed linearization using numerical perturbation. The software computes the block linearization by numerically perturbing the states and inputs of the block about the operating point of the block.

The block perturbation algorithm introduces a small *perturbation* to the nonlinear block and measures the response to this perturbation. The default difference between the perturbed value and the operating point value is $10^{-5}(1 + |x|)$, where x is the operating point value. The software uses this perturbation and the resulting response to compute the linear state-space of this block. For information on how to change perturbation levels for individual blocks, see “Change Perturbation Level of Blocks Perturbed During Linearization” on page 2-148.

In general, a continuous-time nonlinear Simulink block in state-space form is given by:

$$\dot{x}(t) = f(x(t), u(t), t)$$
$$y(t) = g(x(t), u(t), t)\,.$$

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 non-floating-point precision data types. For more information, see “Linearize Blocks with Non-Floating-Point Signals or States” on page 2-150.

Look Up Tables

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

Trim and Linearize an Airframe

This example shows how to trim and linearize an airframe using Simulink® Control Design™ software.

The goal is to find the elevator deflection and the resulting trimmed body rate that generate a given angle of incidence when the airframe is traveling at a set speed.

Once you find the trim condition, you can compute a linear model for the dynamics of the states around the trim condition.

Fixed parameters:

- Angle of incidence (`Theta`)
- Body attitude (`U`)
- Position

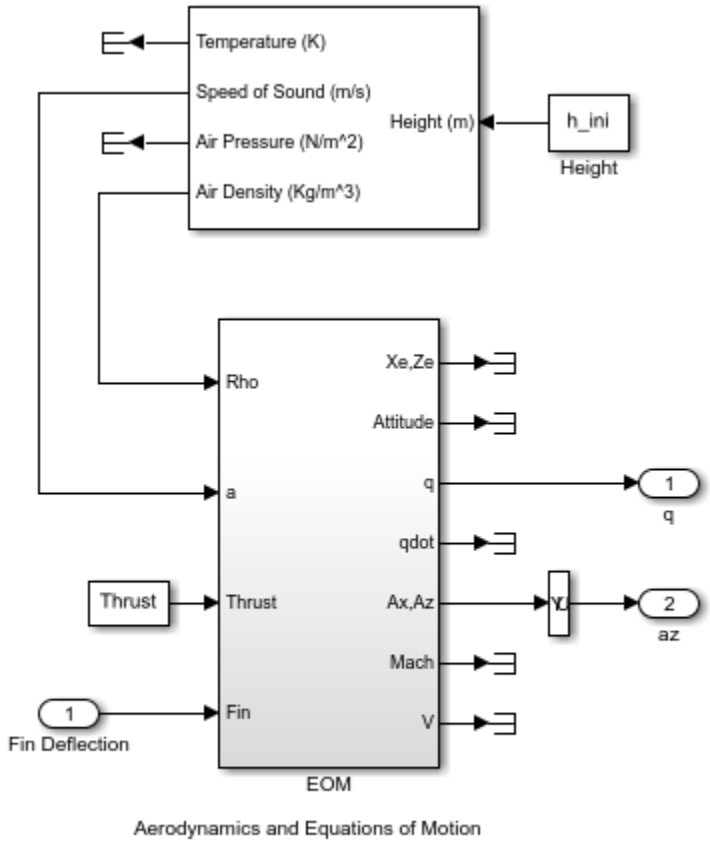
Trimmed steady-state parameters:

- Elevator deflection (`w`)
- Body rate (`q`)

Compute Operating Points

Open the model.

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



Create an operating point specification object for the model using the model initial conditions.

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

```
opspec =


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

States:
----------
      <strong>x</strong>      <strong>Known</strong>    <strong>SteadyState</strong>    <strong>M
    <strong>_____</strong>    <strong>_____</strong>    <strong>___________</strong>    <strong>_

(1.) scdairframe/EOM/ Equations of Motion (Body Axes)/Position
        0    false       true        -Inf    Inf    -Inf      Inf
    -3048    false       true        -Inf    Inf    -Inf      Inf
(2.) scdairframe/EOM/ Equations of Motion (Body Axes)/Theta
        0    false       true        -Inf    Inf    -Inf      Inf
(3.) scdairframe/EOM/ Equations of Motion (Body Axes)/U,w
      984    false       true        -Inf    Inf    -Inf      Inf
        0    false       true        -Inf    Inf    -Inf      Inf
```

```
(4.) scdairframe/EOM/ Equations of Motion (Body Axes)/q
        0    false       true        -Inf    Inf    -Inf      Inf

Inputs:
----------
    <strong>u</strong>    <strong>Known</strong>    <strong>Min</strong>     <strong>Max</strong>
    <strong>_</strong>    <strong>_____</strong>    <strong>____</strong>    <strong>___</strong>

(1.) scdairframe/Fin Deflection
    0    false    -Inf    Inf

Outputs:
----------
    <strong>y</strong>    <strong>Known</strong>    <strong>Min</strong>     <strong>Max</strong>
    <strong>_</strong>    <strong>_____</strong>    <strong>____</strong>    <strong>___</strong>

(1.) scdairframe/q
    0    false    -Inf    Inf
(2.) scdairframe/az
    0    false    -Inf    Inf
```

Specify which states in the model are:

- Known at the operating point
- At steady state at the operating point

Specify that the `Position` states are known and not at steady state. For the state values, specified in `opspec.States(1).x`, use the default values from the model initial condition.

```
opspec.States(1).Known = [1;1];
opspec.States(1).SteadyState = [0;0];
```

Specify that the second state, which corresponds to the angle of incidence `Theta`, is known but not at steady state. As with the position states, use the default state value from the model initial condition.

```
opspec.States(2).Known = 1;
opspec.States(2).SteadyState = 0;
```

The third state specification includes the body axis angular rates `U` and `w`. Specify that both states are known at the operating point and that `w` is at steady state.

```
opspec.States(3).Known = [1 1];
opspec.States(3).SteadyState = [0 1];
```

Search for the operating point that meets these specifications.

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


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

opreport =


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

```
Operating point specifications were successfully met.
States:
----------
     <strong>Min</strong>         <strong>x</strong>          <strong>Max</strong>     <strong>dx
    <strong>_____</strong>    <strong>__________</strong>    <strong>_____</strong>    <strong>__

(1.) scdairframe/EOM/ Equations of Motion (Body Axes)/Position
        0             0        0    -Inf             984     Inf
    -3048         -3048    -3048    -Inf               0     Inf
(2.) scdairframe/EOM/ Equations of Motion (Body Axes)/Theta
        0             0        0    -Inf      -0.0097235     Inf
(3.) scdairframe/EOM/ Equations of Motion (Body Axes)/U,w
      984           984      984    -Inf           22.69     Inf
        0             0        0       0     -1.4367e-11       0
(4.) scdairframe/EOM/ Equations of Motion (Body Axes)/q
     -Inf    -0.0097235      Inf       0      1.7215e-16       0

Inputs:
----------
    <strong>Min</strong>         <strong>u</strong>        <strong>Max</strong>
    <strong>____</strong>    <strong>_________</strong>    <strong>___</strong>

(1.) scdairframe/Fin Deflection
    -Inf    0.0014161    Inf

Outputs:
----------
    <strong>Min</strong>         <strong>y</strong>         <strong>Max</strong>
    <strong>____</strong>    <strong>__________</strong>    <strong>___</strong>

(1.) scdairframe/q
    -Inf    -0.0097235    Inf
(2.) scdairframe/az
    -Inf      -0.24207    Inf
```

Linearize Model

To linearize the model at the computed operating point, first specify the linearization input and output points.

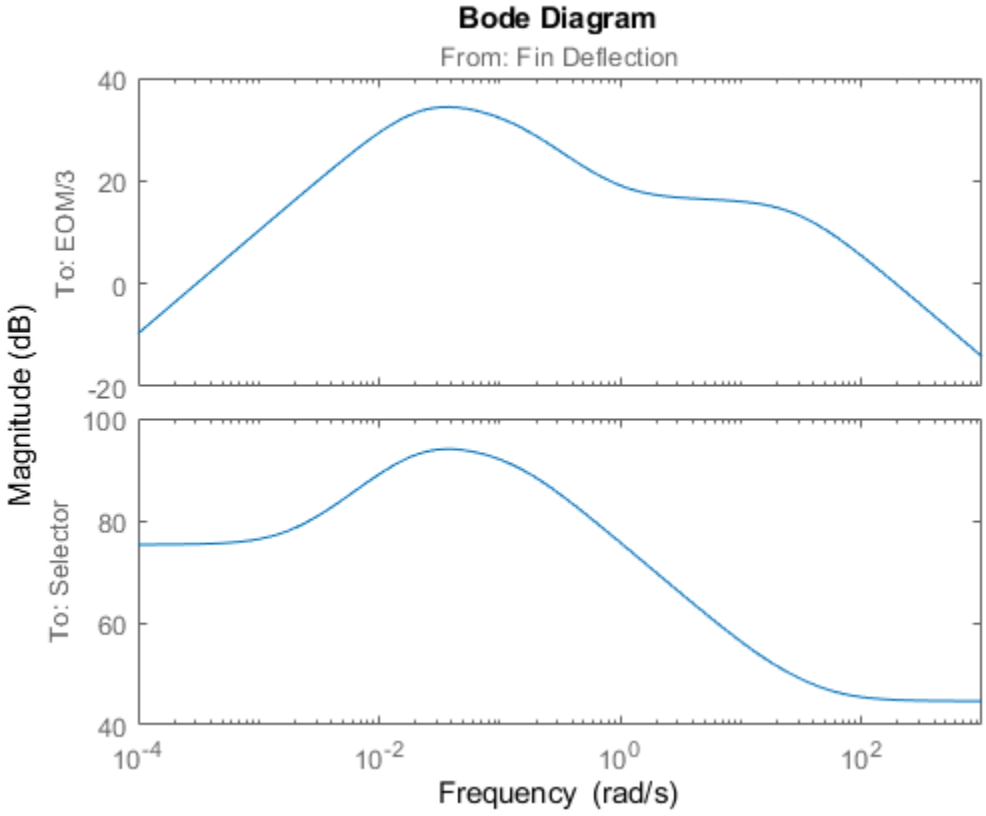
```
io(1) = linio('scdairframe/Fin Deflection',1,'input');
io(2) = linio('scdairframe/EOM',3,'output');
io(3) = linio('scdairframe/Selector',1,'output');
```

Linearize the model at the operating point.

```
sys = linearize(mdl,op,io);
```

Plot the Bode magnitude response for the linear model.

```
bodemag(sys)

bdclose('scdairframe')
```



See Also

`operspec` | `findop` | `linio` | `linearize`

Related Examples

- “Compute Steady-State Operating Points from Specifications” on page 1-13
- “Specify Portion of Model to Linearize” on page 2-10
- “Linearize at Trimmed Operating Point” on page 2-67

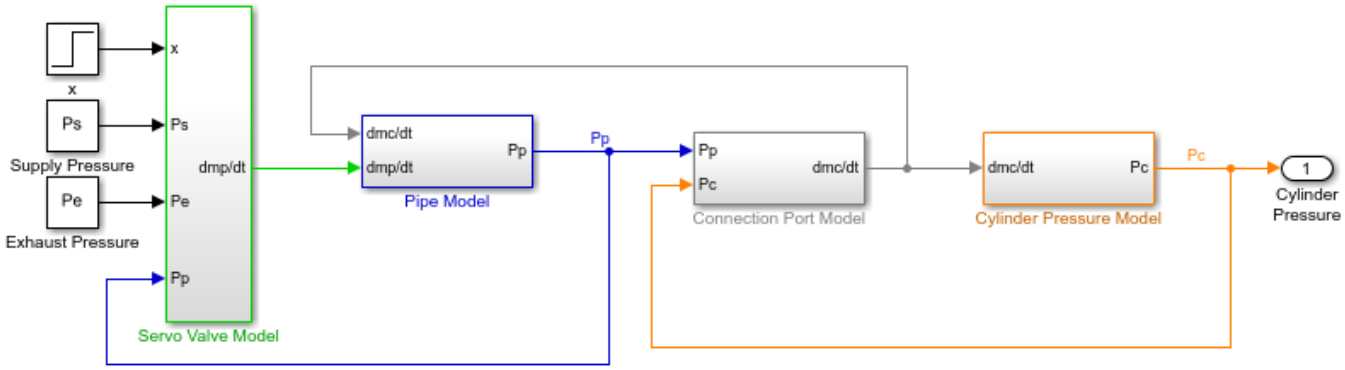
Linearize Pneumatic System at Simulation Snapshots

This example shows how to linearize a Simulink® model at time-based operating point snapshots. The example uses a model of the dynamics of filling a cylinder with compressed air.

Pneumatic System Model

Open the Simulink model.

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

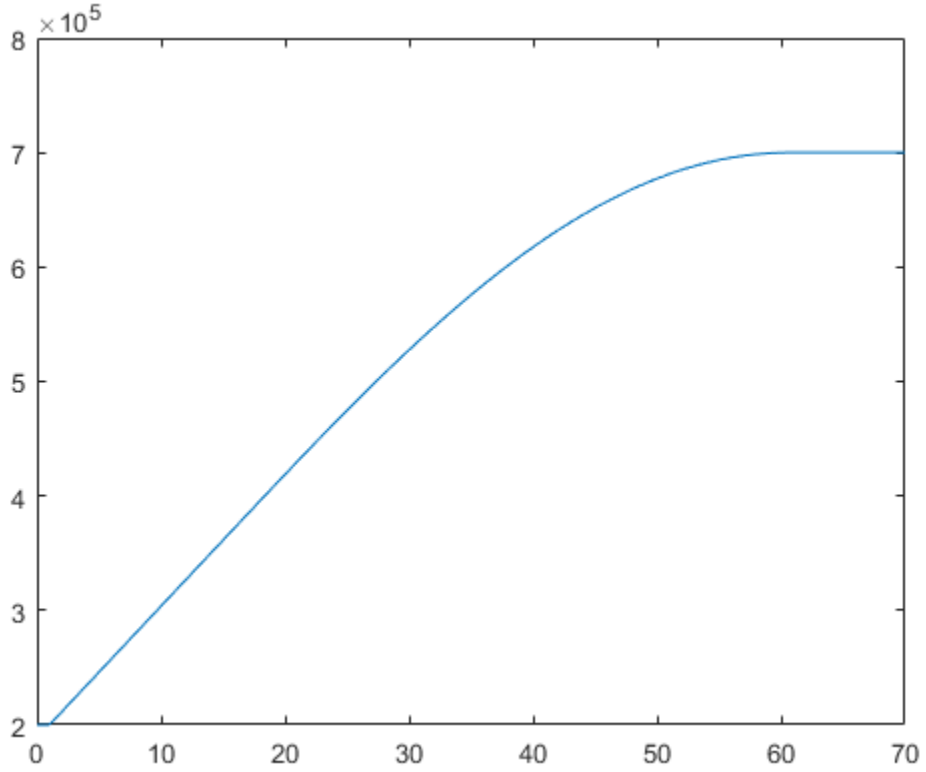


Simulate the model.

```
[t,x,y] = sim(mdl);
```

In this example, the supply pressure is closed and the system has an initial pressure of 0.2 MPa. The supply pressure is at 0.7 MPa. In the simulation, the servo valve is opened to 0.5e-4 m. During the simulation, the pressure increases from the initial pressure of 0.2 MPa and eventually settles at the supply pressure.

```
plot(t,y)
```



Take Simulation Snapshots

Compute operating points at multiple simulation times from 0 to 60 seconds in 10-second intervals. The `findop` function simulates the model, takes a snapshot of the model conditions at each simulation time, and computes an operating point for each snapshot.

```
op = findop(mdl,[0 10 20 30 40 50 60]);
```

View the operating point for the second snapshot time.

```
op(2)
```

```
ans =


 Operating point for the Model scdpneumaticlin.
 (Time-Varying Components Evaluated at time t=10.7245)

States:
----------
        <strong>x</strong>
    <strong>__________</strong>

(1.) scdpneumaticlin/Cylinder Pressure Model/dPc//dt
    3.1205e+05
(2.) scdpneumaticlin/Pipe Model/dPp//dt
    3.1251e+05
```

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

Linearize Model

To linearize the model, first specify the linearization input and output points. For this example, linearize the model from servo valve opening `x` to the output pressure.

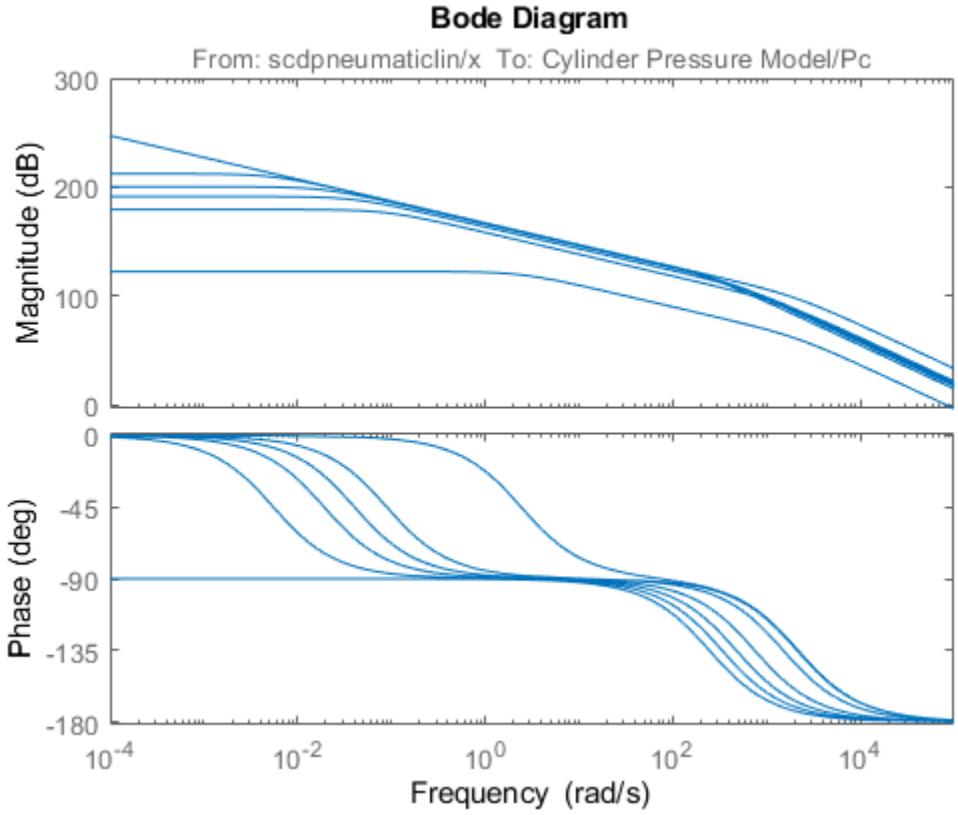
```
io(1) = linio('scdpneumaticlin/x',1,'input');
io(2) = linio('scdpneumaticlin/Cylinder Pressure Model',1,'output');
```

Linearize the model for all of the computed snapshots. `sys` is an array of state-space models.

```
sys = linearize(mdl,op,io);
```

To see the variability in the linearizations, plot the frequency responses of the resulting linear systems.

```
bode(sys)
```



Close the model.

```
bdclose(mdl)
```

See Also

`operspec` | `findop` | `linio` | `linearize`

Related Examples

- “Initialize Steady-State Operating Point Search Using Simulation Snapshot” on page 1-45
- “Specify Portion of Model to Linearize” on page 2-10
- “Linearize at Trimmed Operating Point” on page 2-67

Linearize Pulp Paper Process Model

This example shows how to linearize a process model at a steady-state operating point.

Thermo-mechanical pulping (TMP) is a process for producing mechanical pulp for newsprint. In this example, you use a typical process arrangement for a two-stage TMP operation:

- In the first stage, the primary refiner produces a course pulp from a feed of wood chips and water.
- In the second stage, the secondary refiner further develops the pulp bonding properties so that it is suitable for paper making.

Each refiner consists of two disks with overlaid grooved surfaces. When in operation, either the two disks rotate in opposite directions or one disk rotates while the other disk remains stationary.

The disk surfaces physically impact on a three-phase flow of wood fibers, steam, and water that passes from the center of the refiner disks to their periphery. The physical impact of the disk surfaces on the wood fibers:

1. Breaks rigid chemical and physical bonds between the fibers
2. Microscopically roughens the surface of individual fibers enabling them to mesh together on the paper sheet.

The primary objective of controlling the TMP plant is to apply sufficient energy to derive pulp with good physical properties without incurring excess energy costs or fiber damage due to the imposition of overly high stresses as fibers pass through the refiners. For practical purposes, this objective amounts to controlling the ratio of the total electrical energy applied by the two refiners to the dry mass flow rate of wood fibers, that is, controlling the estimated specific energy applied to the pulp.

A secondary control objective is to control the pulp consistency, that is the ratio of dry mass flow rate (fibers) to overall mass flow rate (water & fibers), to a value which optimizes a trade-off between energy consumption and pulp quality.

The TMP process has the following inputs.

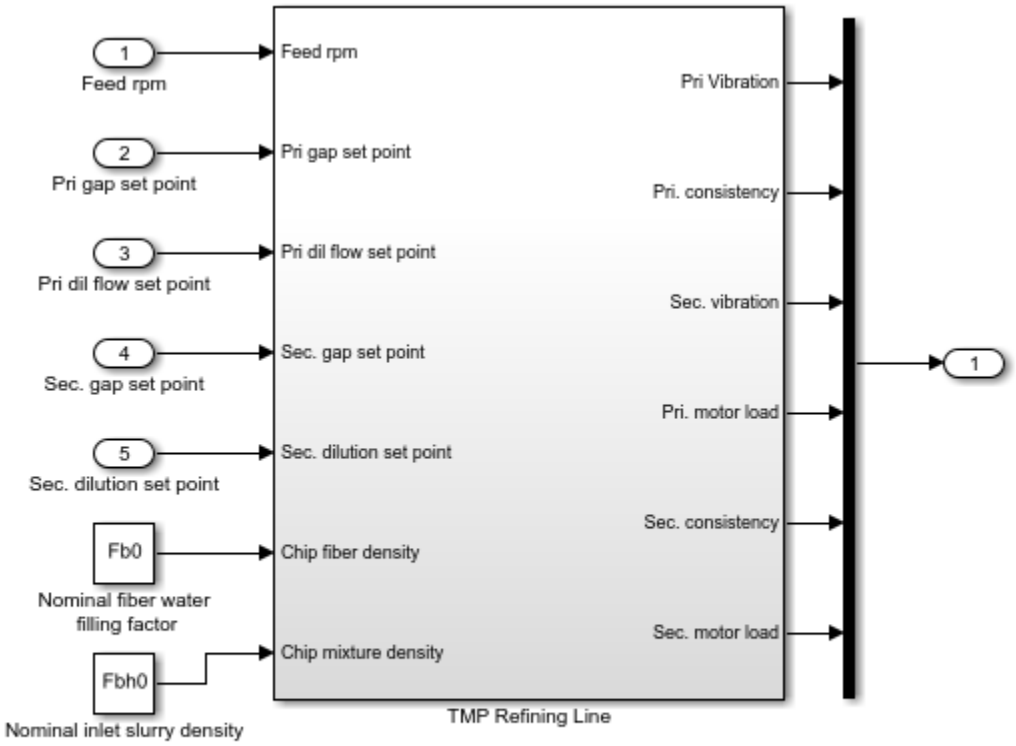
- Feed rate of chips (Feed rpm)
- Dilution water flow to each of the refiners (Primary and secondary dilution setpoints)
- Setpoints to two regulatory controllers which control the gap between the rotating disks in each set of refiners.

The TMP process has the following outputs.

- Primary and secondary refiner consistencies
- Primary and secondary refiner motor loads
- Vibration monitor measurements for the two refiners

Open the `scdtmp` model, which implements the thermo-mechanical pulping process.

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



In this example, your goal is to find a linear model of this system at a steady-state operating condition for the following input setpoint conditions.

- Feed Rate = 30
- Primary Gap = 0.8
- Primary Dilution = 170
- Secondary Gap = 0.5
- Secondary Dilution = 120

Find Operating Points

To find the operating point, first create an operating point specification object using the `operspec` function.

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

```
opspec =

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

States:
```

```
----------
      <strong>x</strong>       <strong>Known</strong>    <strong>SteadyState</strong>    <strong>
    <strong>______</strong>    <strong>_____</strong>    <strong>___________</strong>    <strong>

(1.) scdtmp/TMP Refining Line/Fiber fill dynamics/Internal
    3.5556    false       true        -Inf    Inf    -Inf      Inf
(2.) scdtmp/TMP Refining Line/Fiber water fill dynamics/Internal
    6.8283    false       true        -Inf    Inf    -Inf      Inf
(3.) scdtmp/TMP Refining Line/Primary dilution/Internal
       170    false       true        -Inf    Inf    -Inf      Inf
(4.) scdtmp/TMP Refining Line/Primary plate gap/Internal
       0.8    false       true        -Inf    Inf    -Inf      Inf
(5.) scdtmp/TMP Refining Line/Primary refiner motor/LTI System/Internal
       8.5    false       true        -Inf    Inf    -Inf      Inf
(6.) scdtmp/TMP Refining Line/Primary screw feeder/Internal
        30    false       true        -Inf    Inf    -Inf      Inf
(7.) scdtmp/TMP Refining Line/Sec refiner motor/LTI System/Internal
       6.7    false       true        -Inf    Inf    -Inf      Inf
(8.) scdtmp/TMP Refining Line/Secondary dilution/Internal
       0.5    false       true        -Inf    Inf    -Inf      Inf
(9.) scdtmp/TMP Refining Line/Secondary plate gap/Internal
       0.5    false       true        -Inf    Inf    -Inf      Inf

Inputs:
----------
    <strong>u</strong>    <strong>Known</strong>    <strong>Min</strong>     <strong>Max</strong>
    <strong>_</strong>    <strong>_____</strong>    <strong>____</strong>    <strong>___</strong>

(1.) scdtmp/Feed rpm
    0    false    -Inf    Inf
(2.) scdtmp/Pri gap set point
    0    false    -Inf    Inf
(3.) scdtmp/Pri dil flow set point
    0    false    -Inf    Inf
(4.) scdtmp/Sec. gap set point
    0    false    -Inf    Inf
(5.) scdtmp/Sec. dilution set point
    0    false    -Inf    Inf

Outputs:
----------
    <strong>y</strong>    <strong>Known</strong>    <strong>Min</strong>     <strong>Max</strong>
    <strong>_</strong>    <strong>_____</strong>    <strong>____</strong>    <strong>___</strong>

(1.) scdtmp/Out1
    0    false    -Inf    Inf
    0    false    -Inf    Inf
    0    false    -Inf    Inf
    0    false    -Inf    Inf
    0    false    -Inf    Inf
    0    false    -Inf    Inf
```

Specify the feed rate input value and indicate that this value is known.

```
opspec.Inputs(1).u = 30;
opspec.Inputs(1).Known = 1;
```

Similarly, specify the known primary gap setpoint.

```
opspec.Inputs(2).u = 0.8;
opspec.Inputs(2).Known = 1;
```

Specify the known primary dilution setpoint.

```
opspec.Inputs(3).u = 170;
opspec.Inputs(3).Known = 1;
```

Specify the known secondary gap setpoint.

```
opspec.Inputs(4).u = 0.5;
opspec.Inputs(4).Known = 1;
```

Specify the known secondary dilution setpoint.

```
opspec.Inputs(5).u = 120;
opspec.Inputs(5).Known = 1;
```

Trim the model for the given operating point specifications using the `findop` function.

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

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

opreport =


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

Operating point specifications were successfully met.
States:
----------
    <strong>Min</strong>       <strong>x</strong>       <strong>Max</strong>    <strong>dxMin</s
    <strong>____</strong>    <strong>______</strong>    <strong>___</strong>    <strong>_____</s

(1.) scdtmp/TMP Refining Line/Fiber fill dynamics/Internal
    -Inf    3.5556    Inf      0               0      0
(2.) scdtmp/TMP Refining Line/Fiber water fill dynamics/Internal
    -Inf    6.8283    Inf      0               0      0
(3.) scdtmp/TMP Refining Line/Primary dilution/Internal
    -Inf       170    Inf      0               0      0
(4.) scdtmp/TMP Refining Line/Primary plate gap/Internal
    -Inf       0.8    Inf      0               0      0
(5.) scdtmp/TMP Refining Line/Primary refiner motor/LTI System/Internal
    -Inf    8.4952    Inf      0               0      0
(6.) scdtmp/TMP Refining Line/Primary screw feeder/Internal
    -Inf        30    Inf      0               0      0
(7.) scdtmp/TMP Refining Line/Sec refiner motor/LTI System/Internal
    -Inf    6.6385    Inf      0      1.7355e-12      0
(8.) scdtmp/TMP Refining Line/Secondary dilution/Internal
    -Inf       120    Inf      0               0      0
(9.) scdtmp/TMP Refining Line/Secondary plate gap/Internal
    -Inf       0.5    Inf      0               0      0

Inputs:
----------
```

```
    <strong>Min</strong>     <strong>u</strong>     <strong>Max</strong>
    <strong>___</strong>    <strong>___</strong>    <strong>___</strong>

(1.) scdtmp/Feed rpm
     30     30     30
(2.) scdtmp/Pri gap set point
    0.8    0.8    0.8
(3.) scdtmp/Pri dil flow set point
    170    170    170
(4.) scdtmp/Sec. gap set point
    0.5    0.5    0.5
(5.) scdtmp/Sec. dilution set point
    120    120    120

Outputs:
----------
    <strong>Min</strong>        <strong>y</strong>        <strong>Max</strong>
    <strong>____</strong>    <strong>________</strong>    <strong>___</strong>

(1.) scdtmp/Out1
    -Inf    0.026027    Inf
    -Inf     0.39991    Inf
    -Inf     0.56757    Inf
    -Inf      8.4952    Inf
    -Inf     0.34914    Inf
    -Inf      6.6385    Inf
```

Linearize Model

To linearize the model, first define the linearization input and output points.

For this example, use the following input points.

- Feed rate
- Primary gap
- Primary dilution
- Secondary gap
- Secondary dilution

```
io(1) = linio('scdtmp/Feed rpm',1,'input');
io(2) = linio('scdtmp/Pri gap set point',1,'input');
io(3) = linio('scdtmp/Pri dil flow set point',1,'input');
io(4) = linio('scdtmp/Sec. gap set point',1,'input');
io(5) = linio('scdtmp/Sec. dilution set point',1,'input');
```

The output of the Mux block contains the six model outputs in the following order.

- Primary vibration
- Primary consistency
- Secondary vibration
- Primary motor load
- Secondary consistency
- Secondary motor load

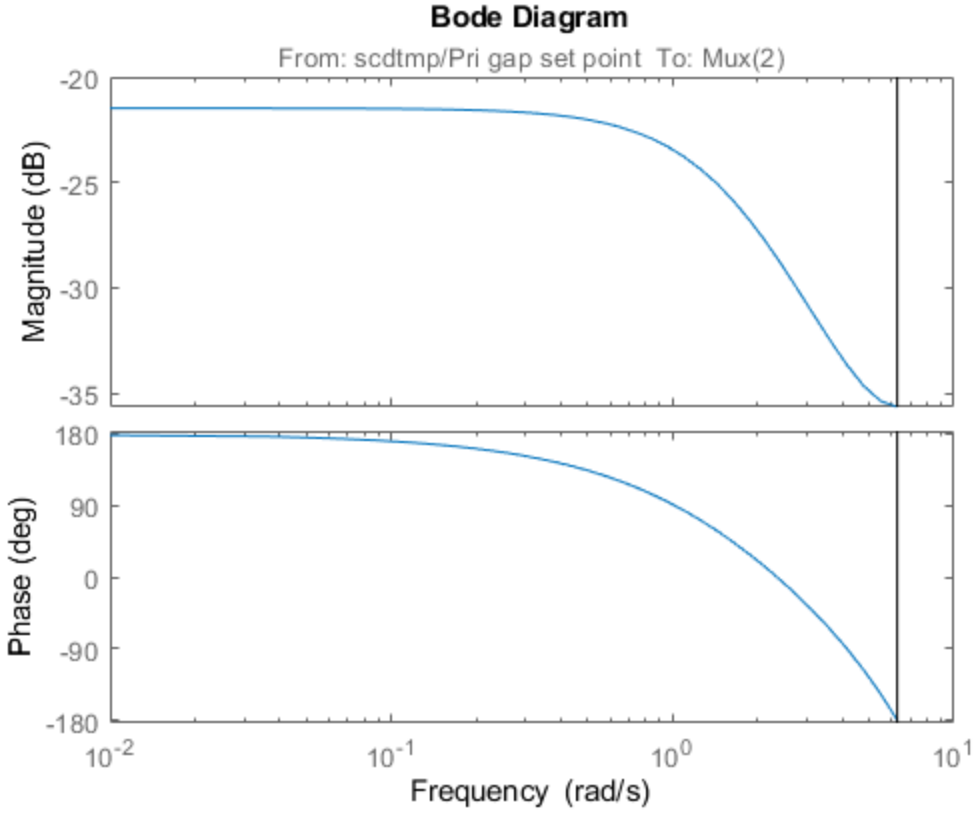
```
io(6) = linio('scdtmp/Mux',1,'output');
```

Linearize the model at the computed steady-state operating point.

```
sys = linearize(mdl,op,io);
```

You can view the response for the resulting linear system from any input to any output. For example, plot the Bode response between the primary gap setpoint and the primary consistency.

```
bode(sys(2,2))
```



See Also

`operspec` | `findop` | `linio` | `linearize`

Related Examples

- “Compute Steady-State Operating Points from Specifications” on page 1-13
- “Specify Portion of Model to Linearize” on page 2-10
- “Linearize at Trimmed Operating Point” on page 2-67

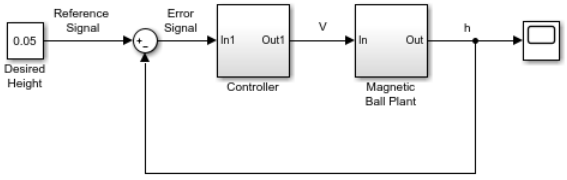
Batch Linearization

- “What Is Batch Linearization?” on page 3-2
- “Choose Batch Linearization Methods” on page 3-4
- “Batch Linearization Efficiency When You Vary Parameter Values” on page 3-7
- “Mark Signals of Interest for Batch Linearization” on page 3-9
- “Batch Linearize Model for Parameter Variations at Single Operating Point” on page 3-13
- “Batch Linearize Model at Multiple Operating Points Derived from Parameter Variations” on page 3-16
- “Batch Linearize Model at Multiple Operating Points Using linearize Command” on page 3-19
- “Vary Parameter Values and Obtain Multiple Transfer Functions” on page 3-21
- “Vary Operating Points and Obtain Multiple Transfer Functions Using slLinearizer Interface” on page 3-28
- “Analyze Command-Line Batch Linearization Results Using Response Plots” on page 3-33
- “Analyze Batch Linearization Results in Model Linearizer” on page 3-39
- “Specify Parameter Samples for Batch Linearization” on page 3-43
- “Batch Linearize Model for Parameter Value Variations Using Model Linearizer” on page 3-53
- “More Efficient Batch Linearization Varying Parameters” on page 3-65
- “Validate Batch Linearization Results” on page 3-69
- “Approximate Nonlinear Behavior Using Array of LTI Systems” on page 3-70
- “LPV Approximation of Boost Converter Model” on page 3-83
- “Linearize Engine Speed Model” on page 3-93
- “Improve Linear Analysis Performance” on page 3-97

What Is Batch Linearization?

Batch linearization refers to extracting multiple linearizations from a model for various combinations of I/Os, operating points, and parameter values. Batch linearization lets you analyze the time-domain, frequency-domain, and stability characteristics of your Simulink model, or portions of your model, under varying operating conditions and parameter ranges. You can use the results of batch linearization to design controllers that are robust against parameter variations, or to design gain-scheduled controllers for different operating conditions. You can also use batch linearization results to implement linear parameter varying (LPV) approximations of nonlinear systems using the LPV System block of Control System Toolbox.

To understand different types of batch linearization, consider the magnetic ball levitation model, `magball`. For more information about this model, see “magball Simulink Model”.



You can batch linearize this model by varying any combination of the following:

- I/O sets — Linearize a model using different I/Os to obtain any closed-loop or open-loop transfer function.

  For the `magball` model, some of the transfer functions that you can extract by specifying different I/O sets include:

  - Magnetic ball plant model, controller model
  - Closed-loop transfer function from the `Reference Signal` to the plant output, `h`
  - Open-loop transfer function for the controller and magnetic ball plant combined; that is, the transfer function from the `Error Signal` to `h` with the feedback loop opened
  - Output disturbance rejection model or sensitivity transfer function, obtained at the outport of `Magnetic Ball Plant` block
- Operating points — In nonlinear models, the model dynamics vary depending on the operating conditions. You can linearize a nonlinear model at different operating points to study how model dynamics vary or to design controllers for different operating conditions.

  For an example of model dynamics that vary depending on the operating point, consider a simple unforced hanging pendulum with angular position and velocity as states. This model has two equilibrium points, one when the pendulum hangs downward, which is stable, and another when the pendulum points upward, which is unstable. Linearizing close to the stable operating point produces a stable model, whereas linearizing this model close to the unstable operating point produces an unstable model.

  For the `magball` model, which uses the ball height as a state, you can obtain multiple linearizations for varying initial ball heights.
- Parameters — Parameters configure a Simulink model in several ways. For example, you can use parameters to specify model coefficients or controller sample times. You can also use a discrete

parameter, such as the control input to a Multiport Switch block, to control the data path within a model. Therefore, varying a parameter can serve a range of purposes, depending on how the parameter contributes to the model.

For the `magball` model, you can vary the parameters of the PID Controller block, `Controller/PID Controller`. The linearizations obtained by varying these parameters show how the controller affects the control-system dynamics. Alternatively, you can vary the magnetic ball plant parameter values to determine the controller robustness to variations in the plant model. You can also vary the parameters of the input block, `Desired Height`, and study the effects of varying input levels on the model response.

If the parameters affect the model operating point, you can batch trim the model using the parameter samples and then batch linearize the model at the resulting operating points.

See Also

LPV System

More About

- “Choose Batch Linearization Methods” on page 3-4
- “Batch Linearization Efficiency When You Vary Parameter Values” on page 3-7
- “Batch Linearize Model for Parameter Value Variations Using Model Linearizer” on page 3-53
- “Batch Linearize Model for Parameter Variations at Single Operating Point” on page 3-13
- “Vary Operating Points and Obtain Multiple Transfer Functions Using slLinearizer Interface” on page 3-28
- “LPV Approximation of Boost Converter Model” on page 3-83

Choose Batch Linearization Methods

Simulink Control Design software provides multiple tools and methods for batch linearization. Which tool and method you choose depends on your application requirements and software preferences. The following table describes the batch linearization workflows supported by Simulink Control Design software.

Application Description	Operating Point Computation Options	Linearization Workflow
Your model has more than one operating condition that does not depend on any varying model parameters. Use this approach when the model operating conditions depend only on the model states and inputs.	• Batch trim your model for multiple operating point specifications, using a single model compilation when possible. Batch trimming is not supported in the **Model Linearizer**. • Trim the model separately for each operating point specification, which requires multiple model compilations. Use this option with the **Model Linearizer**. • Compute operating points at multiple simulation snapshot times.	**1** Compute operating points. **2** Batch linearize the model at all operating points. For an example, see: • "Batch Linearize Model at Multiple Operating Points Using linearize Command" on page 3-19
Your model has a single operating condition, and you want to linearize the model at this operating point for varying model parameters. Examples of such an application include: • Studying the effect of component tolerances on model dynamics. • Examining controller robustness to variations in plant parameters.	• Trim the model for a single operating point specification. • Compute an operating point at a simulation snapshot time.	**1** Compute operating point. **2** Define parameter values for linearization. **3** Batch linearize the model at the computed operating point for the specified parameter variations. For an example, see: • "Batch Linearize Model for Parameter Value Variations Using Model Linearizer" on page 3-53 • "Batch Linearize Model for Parameter Variations at Single Operating Point" on page 3-13

Application Description	Operating Point Computation Options	Linearization Workflow
Your model has multiple operating conditions that depend on the values of varying model parameters. Use this approach when creating linear time-varying (LTV) models.	• Batch trim your model for the varying parameter values, using a single model compilation when possible. Batch trimming is not supported in the **Model Linearizer**. • Trim the model separately for each parameter value combination, which requires multiple model compilations. Use this option with the **Model Linearizer**. • Compute an operating point at a simulation snapshot for each parameter value combination.	**1** Define parameter values for trimming. **2** Compute operating points for the specified parameter value variations. **3** Batch linearize the model at the computed operating points using the corresponding parameter value combinations. For an example, see: • "Batch Linearize Model at Multiple Operating Points Derived from Parameter Variations" on page 3-16 • "LPV Approximation of Boost Converter Model" on page 3-83

In addition to varying operating points and model parameters, you can obtain multiple transfer functions from your system by varying the linearization I/O configuration using an `slLinearizer` interface. You can do so for a model with a single operating point and no parameter variation, and also for any of the batch linearization options in the preceding table. For more information, see "Vary Operating Points and Obtain Multiple Transfer Functions Using slLinearizer Interface" on page 3-28 and "Vary Parameter Values and Obtain Multiple Transfer Functions" on page 3-21.

Choose Batch Linearization Tool

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

Reasons to Use Model Linearizer	Reasons to Use `linearize`	Reasons to Use `slLinearizer`
• You are new to Simulink Control Design software. • You have experience with the **Model Linearizer**. • You do not want to batch trim your model, which is not supported in the **Model Linearizer**.	• You are new to Simulink Control Design or have experience with **Model Linearizer**, and you prefer to work at the command line or in a repeatable script. The workflow for using `linearize` closely mirrors the workflow for linearizing models using the **Model Linearizer**. When you generate MATLAB code from the **Model Linearizer** to reproduce your session programmatically, this code uses `linearize`. You can easily modify this code to batch linearize a model. • You are extracting linearizations for a single transfer function; that is, only one I/O set.	• You want to obtain multiple open-loop and closed-loop transfer functions without modifying the model or creating a linearization I/O set (using `linio`) for each transfer function. • You want to obtain multiple open-loop and closed-loop transfer functions without recompiling the model for each transfer function. You can also obtain multiple open-loop and closed-loop transfer functions using `linearize` or the **Model Linearizer**. However, the software recompiles the model each time you change the I/O set.

See Also

More About

- “What Is Batch Linearization?” on page 3-2
- “Batch Linearization Efficiency When You Vary Parameter Values” on page 3-7
- “Batch Linearize Model for Parameter Value Variations Using Model Linearizer” on page 3-53
- “Batch Linearize Model for Parameter Variations at Single Operating Point” on page 3-13
- “Batch Linearize Model at Multiple Operating Points Using linearize Command” on page 3-19
- “Vary Parameter Values and Obtain Multiple Transfer Functions” on page 3-21
- “Vary Operating Points and Obtain Multiple Transfer Functions Using slLinearizer Interface” on page 3-28

Batch Linearization Efficiency When You Vary Parameter Values

You can use the Simulink Control Design linearization tools to efficiently batch linearize a model at varying model parameter values. If all the model parameters you vary are tunable, the linearization tools use a single model compilation to compute linearizations for all parameter grid points.

Tunable and Nontunable Parameters

The term tunable parameters refers to parameters whose values you can change during model simulation without recompiling the model. In general, only parameters that represent mathematical variables are tunable. Common tunable parameters include the Gain parameter of the Gain block, PID gains of the PID Controller block, and Numerator and Denominator coefficients of the Transfer Fcn block.

In contrast, when you vary the value of nontunable parameters, the linearization tools compile the model for each parameter grid point. This repeated compilation makes batch linearization slower. Parameters that specify the appearance or structure of a block, such as the number of inputs of a Sum block, are not tunable. Parameters that specify when a block is evaluated, such as a block's sample time or priority, are also not tunable.

Controlling Model Recompilation

By default, the linearization tools compute all linearizations with a single compilation whenever it is possible to do so, i.e., whenever all parameters are tunable. If the software detects nontunable parameters specified for variation, it issues a warning and recompiles the model for each parameter-grid point. You can change this default behavior at the command line using the `AreParamsTunable` option of `linearizeOptions`. In the **Model Linearizer**, click **More Options** and use the **Recompile the model when parameter values are varied for linearization** option. The following table describes how these options affect the recompilation behavior.

	All varying parameters are tunable	**Some varying parameters are not tunable**
• Command line: `AreParamsTunable = true` (default) • **Model Linearizer**: **Recompile the model when parameter values are varied for linearization** is unchecked (default)	Linearizations are computed for all parameter-grid points with a single compilation.	Model is recompiled for each parameter-grid point. Software issues a warning.
• Command line: `AreParamsTunable = false` • **Model Linearizer**: **Recompile the model when parameter values are varied for linearization** is checked	Model is recompiled for each parameter-grid point.	Model is recompiled for each parameter-grid point. Warning is suppressed.

Suppose that you are performing batch linearization by varying the values of tunable parameters and notice that the software is recompiling the model more than necessary. To ensure that linearizations are computed with a single compilation whenever possible, make sure that:

- At the command line, the `AreParamsTunable` option is set to `true`.

- In **Model Linearizer**, **Recompile the model when parameter values are varied for linearization** is unchecked.

See Also

`slLinearizer` | `linearize` | `linearizeOptions`

More About

- “Set Block Parameter Values”
- “Batch Linearize Model for Parameter Value Variations Using Model Linearizer” on page 3-53
- “Batch Linearize Model for Parameter Variations at Single Operating Point” on page 3-13
- “Vary Parameter Values and Obtain Multiple Transfer Functions” on page 3-21
- “Specify Parameter Samples for Batch Linearization” on page 3-43

Mark Signals of Interest for Batch Linearization

When batch linearizing a model using an `slLinearizer` interface, you can mark signals of interest using analysis points. You can then analyze the response of your system at any of these points using functions such as `getIOTransfer` and `getLoopTransfer`.

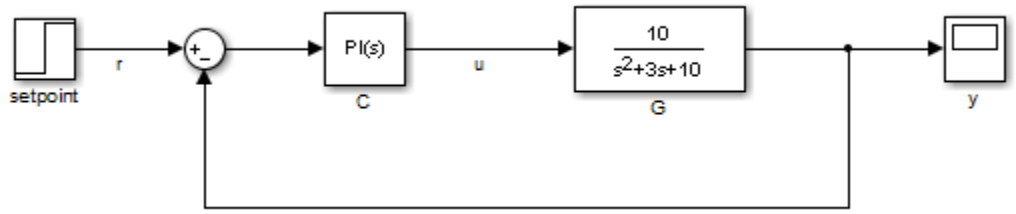
Alternatively, if you are batch linearizing your model using the:

- **Model Linearizer**, specify analysis points as shown in “Specify Portion of Model to Linearize in Model Linearizer” on page 2-22.
- `linearize` command, specify analysis points using `linio`.

For more information on selecting a batch linearization tool, see “Choose Batch Linearization Methods” on page 3-4.

Analysis Points

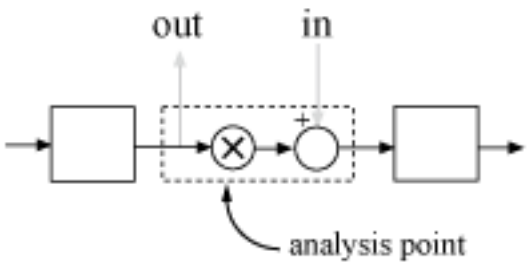
Analysis points identify locations within a Simulink model that are relevant for linear analysis. Each analysis point is associated with a signal that originates from the outport of a Simulink block. For example, in the following model, the reference signal `r` and the control signal `u` are analysis points that originate from the outputs of the setpoint and C blocks respectively.



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

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

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



Using analysis points, you can extract open-loop and closed-loop responses from a Simulink model. You can also specify requirements for control system tuning using analysis points. For more information, see “Mark Signals of Interest for Control System Analysis and Design” on page 2-38.

Specify Analysis Points

You can mark analysis points either explicitly in the Simulink model, or programmatically using the `addPoint` command for an `slLinearizer` interface.

Mark Analysis Points in Simulink Model

To specify analysis points directly in your Simulink model, first open the **Linearization** tab. To do so, in the **Apps** gallery, click **Linearization Manager**.

To specify an analysis point:

1. In the model, click the signal you want to define as an analysis point.
2. On the **Linearization** tab, in the **Insert Analysis Points** gallery, select the type of analysis point you want to define.

   When you specify analysis points, the software adds annotations to your model indicating the linear analysis point type.
3. Repeat steps 1 and 2 for all signals you want to define as analysis points.

You can select any of the following closed-loop analysis point types, which are equivalent within an `slLinearizer` interface.

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

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

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

When you define a signal as an open-loop point, analysis functions such as `getIOTransfer` always enforce a loop break at that signal during linearization. All open-loop analysis point types are equivalent within an `slLinearizer` interface. For more information on how the software treats loop openings during linearization, see “How the Software Treats Loop Openings” on page 2-31.

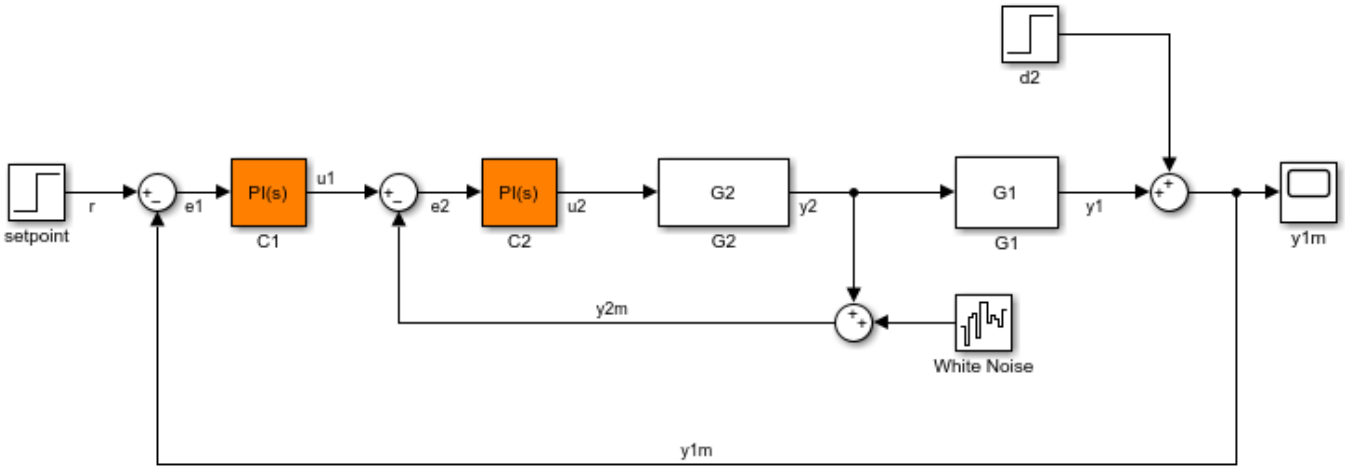
When you create an `slLinearizer` interface for a model, any analysis points defined in the model are automatically added to the interface. If you defined an analysis point using:

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

Mark Analysis Points Programmatically

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

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



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

```
sllin = slLinearizer('scdcascade');
```

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

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

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

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

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

- Name of the signal.

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

- Combined source block path and port number.

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

- End of the full source block path when unambiguous.

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

You can also add permanent openings to an `slLinearizer` interface using the `addOpening` command, and specifying signals in the same way as for `addPoint`. For more information on how the software treats loop openings during linearization, see “How the Software Treats Loop Openings” on page 2-31.

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

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

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

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

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

Refer to Analysis Points

Once you have marked analysis points in an `slLinearizer` interface, you can analyze the response at any of these points using the following analysis functions:

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

To view the available analysis points in an `slLinearizer` interface, use the `getPoints` command.

```
getPoints(sllin)
```

```
ans =

  3x1 cell array

    {'scdcascade/C1/1[u1]'}
    {'scdcascade/G2/1[y2]'}
    {'scdcascade/G1/1[y1]'}
```

To use an analysis point with an analysis function, you can specify an unambiguous abbreviation of the analysis point name returned by `getPoints`. For example, compute the transfer function from `u1` to `y1`, and find the sensitivity to a disturbance at the output of block G2.

```
ioSys = getIOTransfer(sllin,'u1','y1');
sensG2 = getSensitivity(sllin,'G2');
```

See Also

`addPoint` | `getPoints` | `slLinearizer` | `addOpening`

More About

- “Mark Signals of Interest for Control System Analysis and Design” on page 2-38

Batch Linearize Model for Parameter Variations at Single Operating Point

In this example, you vary model parameters and linearize a model at its nominal operating conditions using the `linearize` command.

You can batch linearize a model for parameter variations at a single operating point to study:

- Plant dynamics for varying component tolerances.
- Controller robustness to variations in plant parameters.
- Transient responses for varying controller gains.

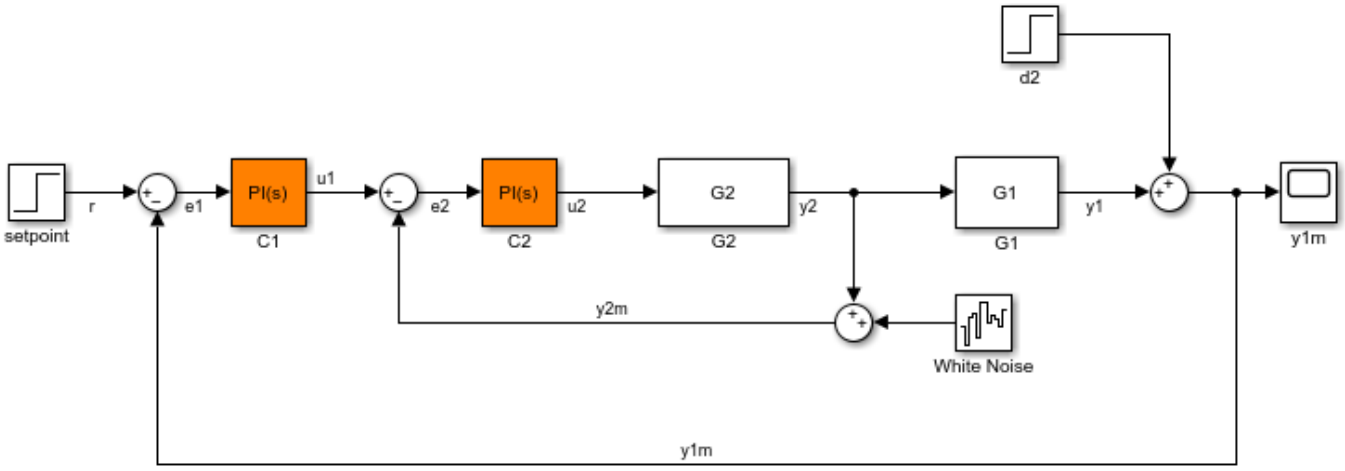
The `scdcascade` model contains two cascaded feedback control loops. Each loop includes a PI controller. The plant models, `G1` and `G2`, are LTI models.

For this model, the model operating point represents the nominal operating conditions of the system. Therefore, you do not have to trim the model before linearization. If your application includes parameter variations that affect the operating point of the model, you must first batch trim the model for the parameter variations. Then, you can linearize the model at the trimmed operating points. For more information, see “Batch Linearize Model at Multiple Operating Points Derived from Parameter Variations” on page 3-16.

To examine the effects of varying the outer-loop controller gains, linearize the model at the nominal operating point for each combination of gain values.

Open the model.

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



Define linearization input and output points for computing the closed-loop input/output response of the system.

```
io(1) = linio('scdcascade/setpoint',1,'input');
io(2) = linio('scdcascade/Sum',1,'output');
```

`io(1)`, the signal originating at the outport of the `setpoint` block, is the reference input. `io(2)`, the signal originating at the outport of the `Sum` block, is the system output.

To extract multiple open-loop and closed-loop transfer functions from the same model, batch linearize the system using an `slLinearizer` interface. For more information, see “Vary Parameter Values and Obtain Multiple Transfer Functions” on page 3-21.

Vary the outer-loop controller gains, `Kp1` and `Ki1`, within 20% of their nominal values.

```
Kp1_range = linspace(Kp1*0.8,Kp1*1.2,6);
Ki1_range = linspace(Ki1*0.8,Ki1*1.2,4);
[Kp1_grid,Ki1_grid] = ndgrid(Kp1_range,Ki1_range);
```

Create a parameter structure with fields `Name` and `Value`. `Name` indicates which the variable to vary in the model workspace, the MATLAB® workspace, or a data dictionary.

```
params(1).Name = 'Kp1';
params(1).Value = Kp1_grid;
params(2).Name = 'Ki1';
params(2).Value = Ki1_grid;
```

`params` is a 6-by-4 parameter value grid, where each grid point corresponds to a unique combination of `Kp1` and `Ki1` values.

Obtain the closed-loop transfer function from the reference input to the plant output for the specified parameter values. If you do not specify an operating point, `linearize` uses the current model operating point.

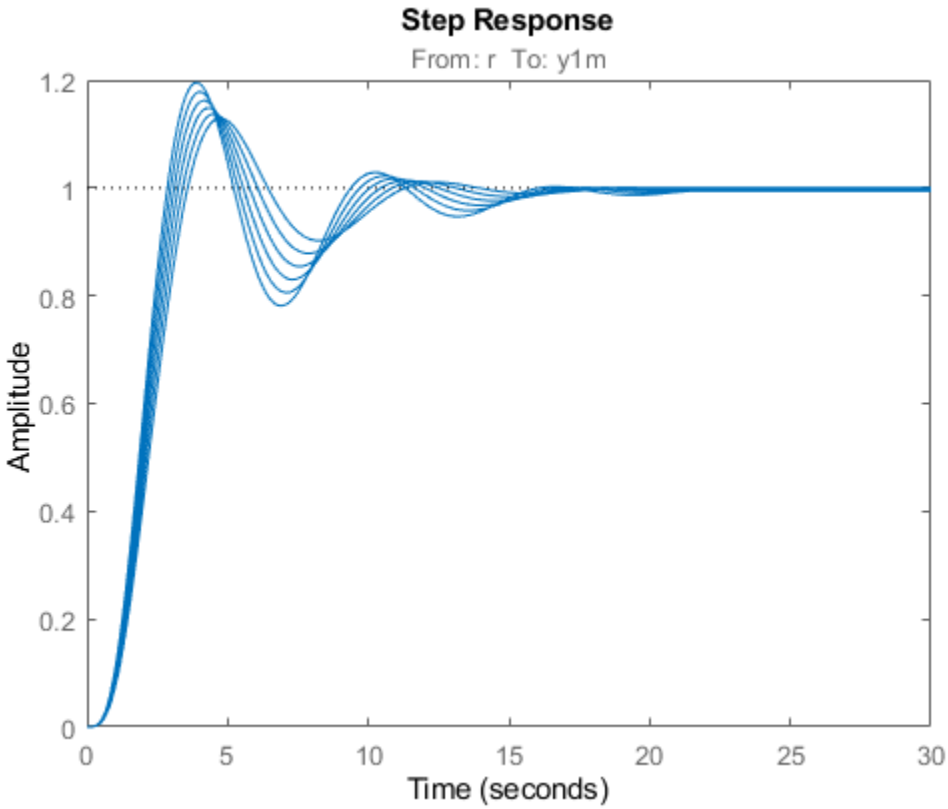
```
G = linearize(sys,io,params);
```

`G` is a 6-by-4 array of linearized models. Each entry in the array contains a linearization for the corresponding parameter combination in `params`. For example, `G(:,:,2,3)` corresponds to the linearization obtained by setting the values of the `Kp1` and `Ki1` parameters to `Kp1_grid(2,3)` and `Ki1_grid(2,3)`, respectively. The set of parameter values corresponding to each entry in the model array `G` is stored in the `SamplingGrid` property of `G`. For example, examine the corresponding parameter values for linearization `G(:,:,2,3)`:

```
G(:,:,2,3).SamplingGrid


ans =

  struct with fields:

    Kp1: 0.1386
    Ki1: 0.0448
```

To study the effects of the varying gain values, analyze the linearized models in `G`. For example, examine the step responses for all `Kp2` values and the third `Ki1` value.

```
stepplot(G(:,:,:,3))
```



See Also

`linearize` | `linio` | `ndgrid`

More About

- “watertank Simulink Model”
- “Batch Linearization Efficiency When You Vary Parameter Values” on page 3-7
- “Specify Parameter Samples for Batch Linearization” on page 3-43
- “Analyze Command-Line Batch Linearization Results Using Response Plots” on page 3-33
- “Batch Linearize Model at Multiple Operating Points Using linearize Command” on page 3-19
- “Batch Linearize Model for Parameter Value Variations Using Model Linearizer” on page 3-53
- “LPV Approximation of Boost Converter Model” on page 3-83

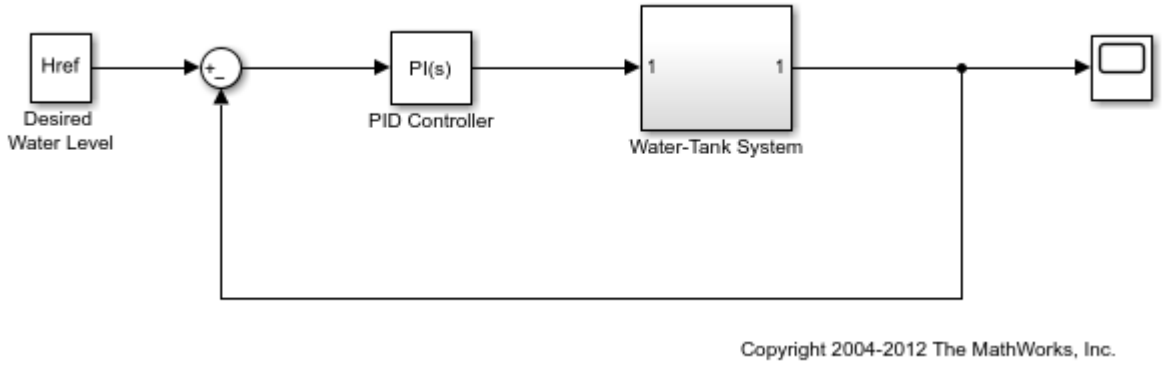
Batch Linearize Model at Multiple Operating Points Derived from Parameter Variations

If your application includes parameter variations that affect the operating point of the model, you must batch trim the model for the parameter variations before linearization. Use this batch linearization approach when computing linear models for linear parameter-varying systems.

For more information on batch trimming models for parameter variations, see “Batch Compute Steady-State Operating Points for Parameter Variation” on page 1-72.

Open the Simulink model.

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



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

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

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

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

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

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

Trim the model using the specified operating point specification, parameter grid, and option set. Suppress the display of the operating point search report.

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

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

To compute the closed-loop input/output transfer function for the model, define the linearization input and output points as the reference input and model output, respectively.

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

To extract multiple open-loop and closed-loop transfer functions from the same model, batch linearize the system using an `slLinearizer` interface. For more information, see “Vary Parameter Values and Obtain Multiple Transfer Functions” on page 3-21.

Batch linearize the model at the trimmed operating points using the specified I/O points and parameter variations.

```
G = linearize(sys,op,io,params);
```

`G` is a 3-by-4 array of linearized models. Each entry in the array contains a linearization for the corresponding parameter combination in `params`. For example, `G(:,:,2,3)` corresponds to the linearization obtained by setting the values of the `A` and `b` parameters to `A_grid(2,3)` and `b_grid(2,3)`, respectively. The set of parameter values corresponding to each entry in the model array `G` is stored in the `SamplingGrid` property of `G`. For example, examine the corresponding parameter values for linearization `G(:,:,2,3)`:

```
G(:,:,2,3).SamplingGrid
```

```
ans =

  struct with fields:

    A: 20
    b: 5.1667
```

When batch linearizing for parameter variations, you can obtain the linearization offsets that correspond to the linearization operating points. To do so, set the `StoreOffsets` linearization option.

```
opt = linearizeOptions('StoreOffsets',true);
```

Linearize the model using the specified parameter grid, and return the linearization offsets in the `info` structure.

```
[G,~,info] = linearize('watertank',io,params,opt);
```

You can then use the offsets to configure an LPV System block. To do so, you must first convert the offsets to the required format. For an example, see “LPV Approximation of Boost Converter Model” on page 3-83.

```
offsets = getOffsetsForLPV(info);
```

See Also

`linearize` | `linio` | `findop` | `ndgrid`

More About

- “Batch Linearization Efficiency When You Vary Parameter Values” on page 3-7
- “Specify Parameter Samples for Batch Linearization” on page 3-43
- “Analyze Command-Line Batch Linearization Results Using Response Plots” on page 3-33
- “Batch Linearize Model for Parameter Value Variations Using Model Linearizer” on page 3-53
- “LPV Approximation of Boost Converter Model” on page 3-83

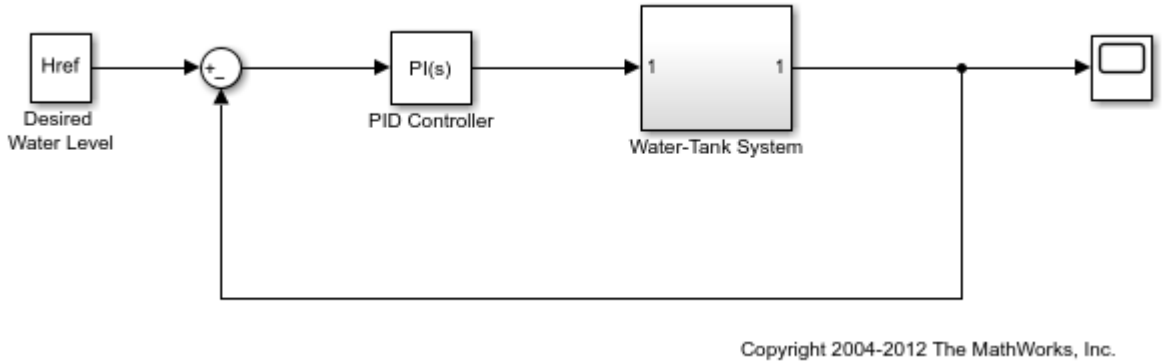
Batch Linearize Model at Multiple Operating Points Using linearize Command

This example shows how to use the `linearize` command to batch linearize a model at varying operating points.

Obtain the plant transfer function, modeled by the Water-Tank System block, for the `watertank` model. You can analyze the batch linearization results to study the operating point effects on the model behavior.

Open the model.

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



Specify the linearization I/Os.

```
ios(1) = linio('watertank/PID Controller',1,'input');
ios(2) = linio('watertank/Water-Tank System',1,'openoutput');
```

`ios(2)` specifies an open-loop output point; the loop opening eliminates the effects of feedback.

You can linearize the model using trimmed operating points, the model initial condition, or simulation snapshot times. For this example, linearize the model at specified simulation snapshot times.

```
ops_tsnapshot = [1,20];
```

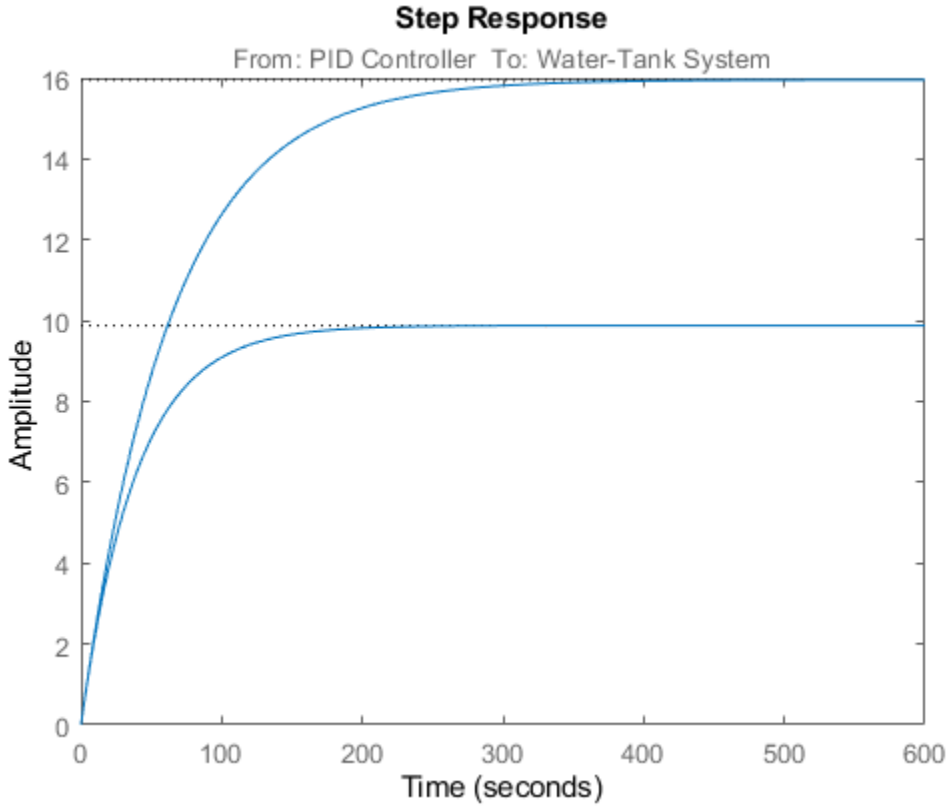
Obtain the transfer function for the Water-Tank System block, linearizing the model at the specified operating points.

```
T = linearize('watertank',ios,ops_tsnapshot);
```

`T` is a 2 x 1 array of linearized continuous-time state-space models. The software computes the `T(:,:,1)` model by linearizing `watertank` at `ops_tsnapshot(1)`, and `T(:,:,2)` by linearizing `watertank` at `ops_tsnapshot(2)`.

Use Control System Toolbox analysis commands to examine the properties of the linearized models in `T`. For example, examine the step response of the plant at both snapshot times.

```
stepplot(T)
```



See Also

`linearize` | `findop` | `linio` | `stepplot`

More About

- “watertank Simulink Model”
- “Batch Compute Steady-State Operating Points for Multiple Specifications” on page 1-68
- “Batch Linearize Model for Parameter Variations at Single Operating Point” on page 3-13
- “Analyze Command-Line Batch Linearization Results Using Response Plots” on page 3-33

Vary Parameter Values and Obtain Multiple Transfer Functions

This example shows how to use the `slLinearizer` interface to batch linearize a Simulink® model. You vary model parameter values and obtain multiple open-loop and closed-loop transfer functions from the model.

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

Since the parameter variations in this example do not affect the operating point of the model, you batch linearize the model at a single operating point. If your application uses parameter variations that affect the model operating point, first trim the model for each parameter value combination. For an example that uses the `linearize` function, see “Batch Linearize Model at Multiple Operating Points Derived from Parameter Variations” on page 3-16.

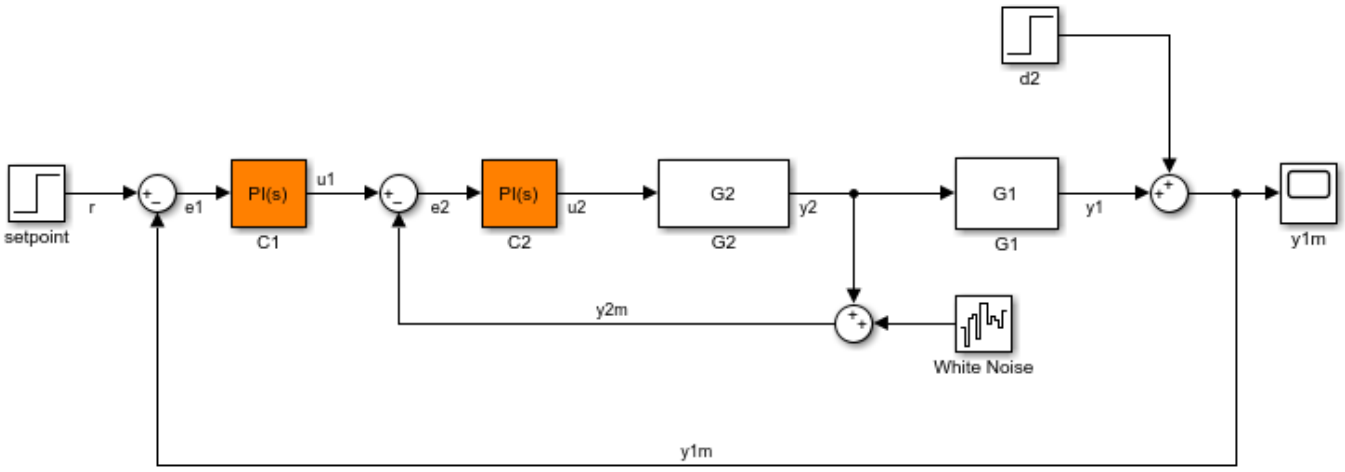
Create `slLinearizer` Interface for Model

The `scdcascade` model used for this example contains a pair of cascaded feedback control loops. Each loop includes a PI controller. The plant models, `G1` (outer loop) and `G2` (inner loop), are LTI models.

Use the `slLinearizer` interface to analyze the inner-loop and outer-loop dynamics.

Open the model.

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



Use the `slLinearizer` function to create the interface.

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

```
slLinearizer linearization interface for "scdcascade":

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

```
Properties with dot notation get/set access:
      Parameters         : []
      OperatingPoints    : [] (model initial condition will be used.)
      BlockSubstitutions : []
      Options            : [1x1 linearize.LinearizeOptions]
```

The Command Window display shows information about the `slLinearizer` interface. In this interface, no parameters to vary are yet specified, so the `Paramaeters` property is empty.

Vary Inner-Loop Controller Gains

For inner-loop analysis, vary the gains of the inner-loop PI controller block, C2. Vary the proportional gain (`Kp2`) and integral gain (`Ki2`) in the 15% range.

```
Kp2_range = linspace(Kp2*0.85,Kp2*1.15,6);
Ki2_range = linspace(Ki2*0.85,Ki2*1.15,4);
[Kp2_grid, Ki2_grid] = ndgrid(Kp2_range,Ki2_range);

params(1).Name = 'Kp2';
params(1).Value = Kp2_grid;
params(2).Name = 'Ki2';
params(2).Value = Ki2_grid;

sllin.Parameters = params;
```

`Kp2_range` and `Ki2_range` specify the sample values for `Kp2` and `Ki2`. To obtain a transfer function for each combination of `Kp2` and `Ki2`, use `ndgrid` and create a 6 x 4 parameter grid with grid arrays `Kp2_grid` and `Ki2_grid`. Configure the `Parameters` property of `sllin` with the structure `params`. This structure specifies the parameters to be varied and their grid arrays.

Analyze Closed-Loop Transfer Function for the Inner Loop

The overall closed-loop transfer function for the inner loop is equal to the transfer function from `u1` to `y2`. To eliminate the effects of the outer loop, you can break the loop at `e1`, `y1m`, or `y1`. For this example, break the loop at `e1`.

Add `u1` and `y2` as analysis points, and `e1` as a permanent opening of `sllin`.

```
addPoint(sllin,{'y2','u1'});
addOpening(sllin,'e1');
```

Obtain the transfer function from `u1` to `y2`.

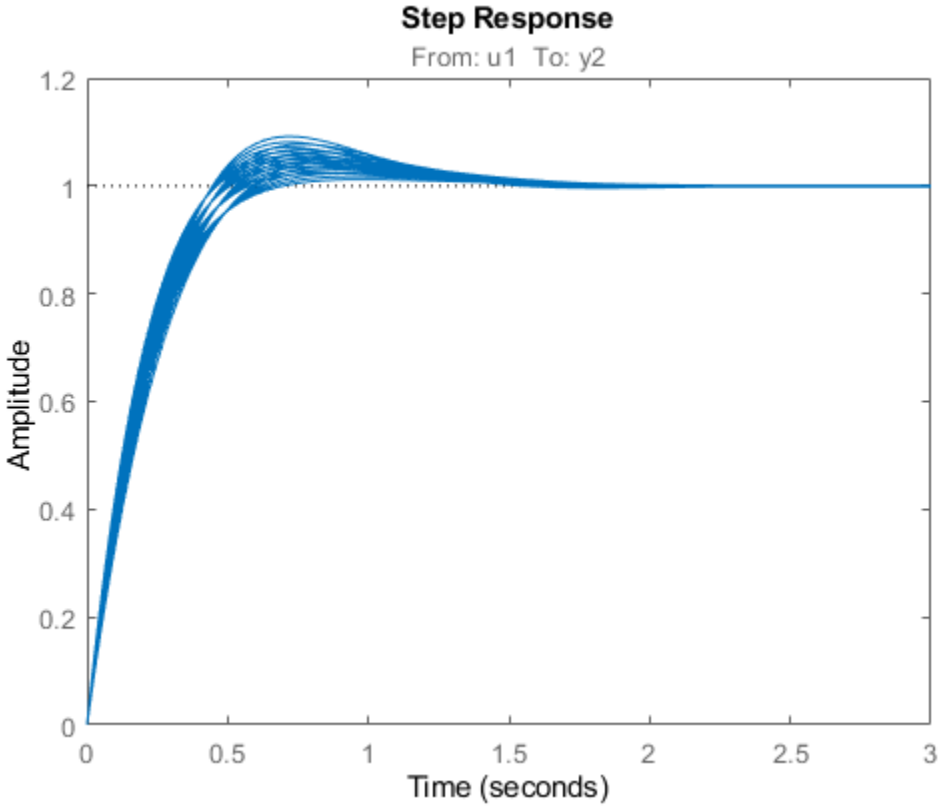
```
r2yi = getIOTransfer(sllin,'u1','y2');
```

`r2yi`, a 6 x 4 state-space model array, contains the transfer function for each specified parameter combination. The software uses the model initial conditions as the linearization operating point.

Because `e1` is a permanent opening of `sllin`, `r2yi` does not include the effects of the outer loop.

Plot the step response for `r2yi`.

```
stepplot(r2yi);
```



The step response for all models varies in the 10% range and the settling time is less than 1.5 seconds.

Analyze Inner-Loop Transfer Function at the Plant Output

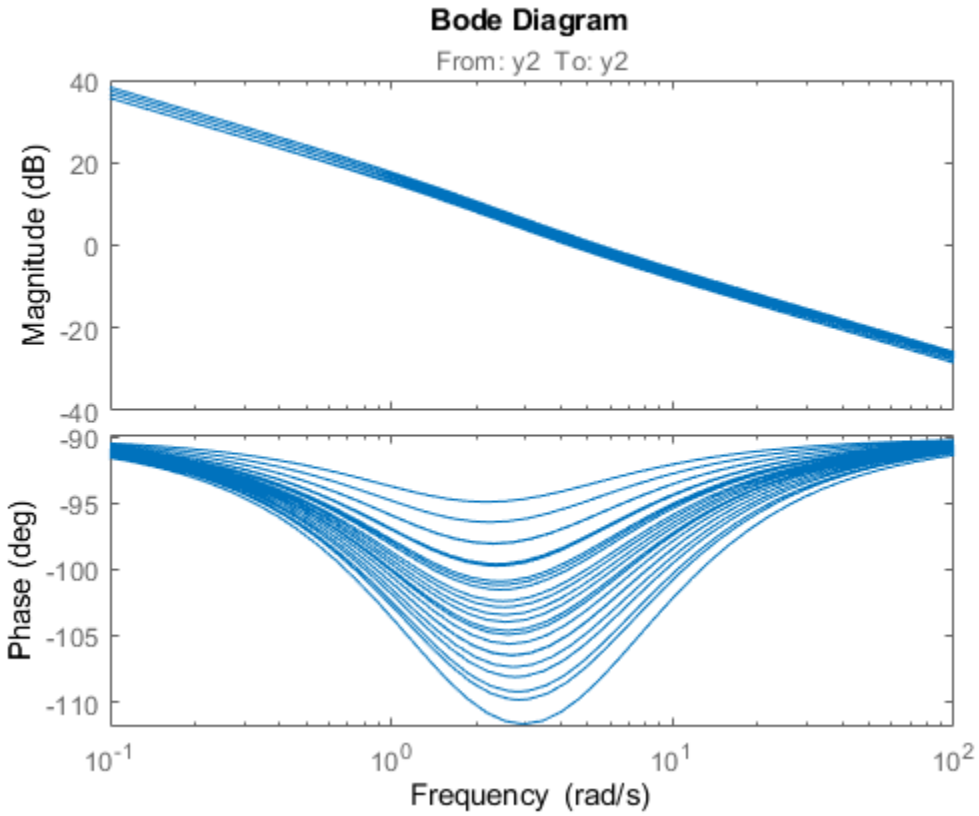
Obtain the inner-loop transfer function at `y2`, with the outer loop open at `e1`.

```
Li = getLoopTransfer(sllin,'y2',-1);
```

Because the software assumes positive feedback by default and `scdcascade` uses negative feedback, specify the feedback sign using the third input argument. Now, $L_i = -G_2C_2$. The `getLoopTransfer` command returns an array of state-space (`ss`) models, one for each entry in the parameter grid. The `SamplingGrid` property of `Li` matches the parameter values with the corresponding `ss` model.

Plot the bode response for L_i.

```
bodeplot(Li)
```



The magnitude plot for all the models varies in the 3-dB range. The phase plot shows the most variation, approximately 20°, in the `[1 10]` rad/s interval.

Vary Outer-Loop Controller Gains

For outer-loop analysis, vary the gains of the outer-loop PI controller block, C1. Vary the proportional gain (`Kp1`) and integral gain (`Ki1`) in the 20% range.

```
Kp1_range = linspace(Kp1*0.8,Kp1*1.2,6);
Ki1_range = linspace(Ki1*0.8,Ki1*1.2,4);
[Kp1_grid, Ki1_grid] = ndgrid(Kp1_range,Ki1_range);

params(1).Name = 'Kp1';
params(1).Value = Kp1_grid;
params(2).Name = 'Ki1';
params(2).Value = Ki1_grid;

sllin.Parameters = params;
```

Similar to the workflow for configuring the parameter grid for inner-loop analysis, create the structure, `params`, that specifies a 6 x 4 parameter grid. Reconfigure `sllin.Parameters` to use the new parameter grid. `sllin` now uses the default values for `Kp2` and `Ki2`.

Analyze Closed-Loop Transfer Function from Reference to Plant Output

Remove `e1` from the list of permanent openings for `sllin` before proceeding with outer-loop analysis.

```
removeOpening(sllin,'e1');
```

To obtain the closed-loop transfer function from the reference signal, `r`, to the plant output, `y1m`, add `r` and `y1m` as analysis points to `sllin`.

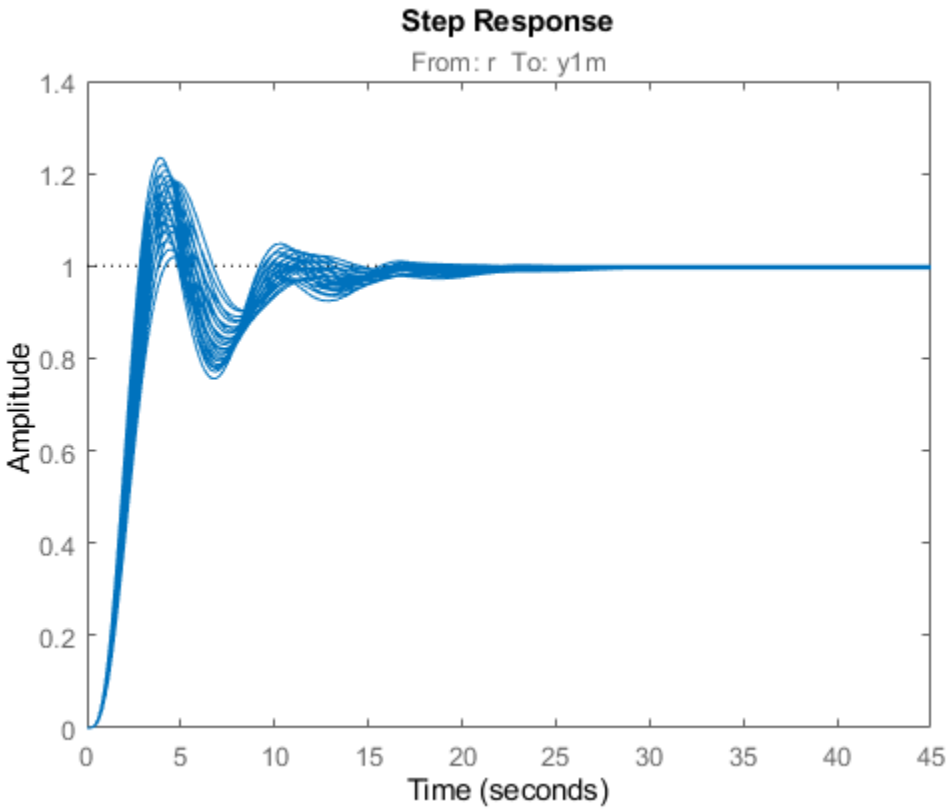
```
addPoint(sllin,{'r','y1m'});
```

Obtain the transfer function from `r` to `y1m`.

```
r2yo = getIOTransfer(sllin,'r','y1m');
```

Plot the step response for `r2yo`.

```
stepplot(r2yo)
```



The step response is underdamped for all the models.

Analyze Outer-Loop Sensitivity at Plant Output

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

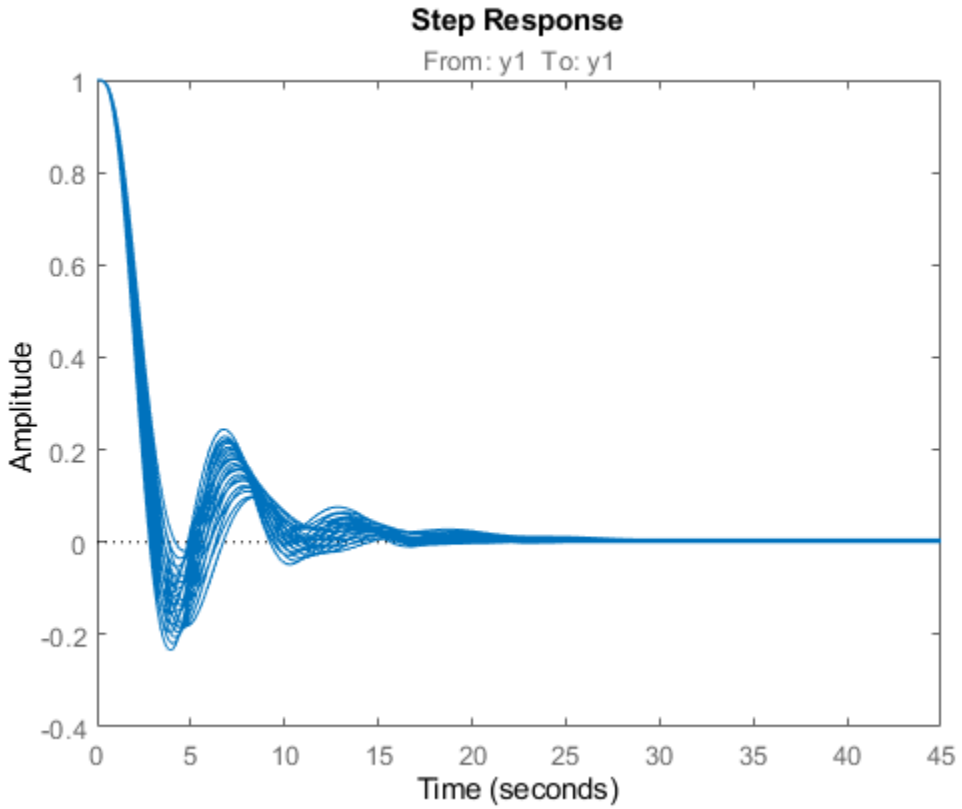
```
addPoint(sllin,'y1');
```

Obtain the outer-loop sensitivity at `y1`.

```
So = getSensitivity(sllin,'y1');
```

Plot the step response of `So`.

```
stepplot(So)
```



The plot indicates that it takes approximately 15 seconds to reject a step disturbance at the plant output, `y1`.

Obtain Linearization Offsets

When batch linearizing for parameter variations, you can obtain the linearization offsets that correspond to the linearization operating points. To do so, set the `StoreOffsets` linearization option in the `slLinearizer` interface.

```
sllin.Options.StoreOffsets = true;
```

When you call a linearization function using `sllin`, you can return linearization offsets in the `info` structure.

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

You can then use the offsets to configure an LPV System block. To do so, you must first convert the offsets to the required format. For an example that uses the `linearize` command, see “LPV Approximation of Boost Converter Model” on page 3-83.

```
offsets = getOffsetsForLPV(info);
```

Close the model.

```
bdclose(mdl)
```

See Also

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

More About

- “Batch Linearization Efficiency When You Vary Parameter Values” on page 3-7
- “Specify Parameter Samples for Batch Linearization” on page 3-43
- “Analyze Command-Line Batch Linearization Results Using Response Plots” on page 3-33
- “Vary Operating Points and Obtain Multiple Transfer Functions Using slLinearizer Interface” on page 3-28
- “Batch Linearize Model for Parameter Value Variations Using Model Linearizer” on page 3-53

Vary Operating Points and Obtain Multiple Transfer Functions Using slLinearizer Interface

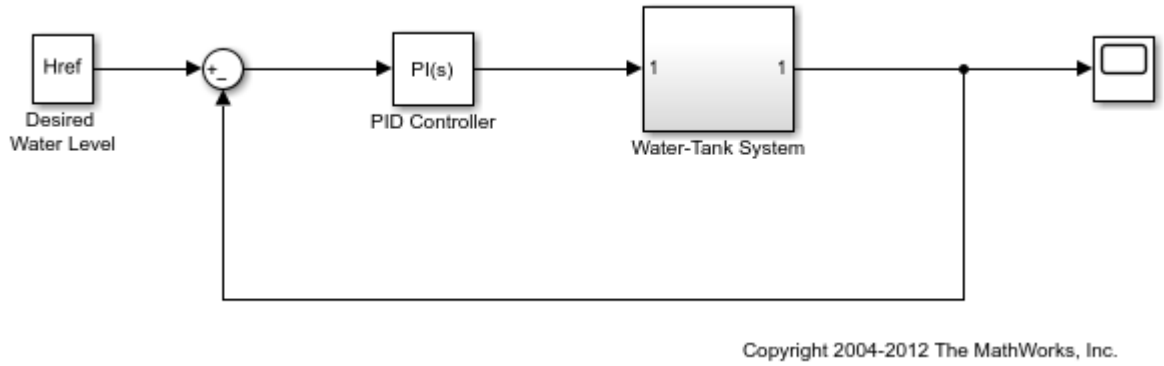
This example shows how to use the `slLinearizer` interface to batch linearize a Simulink® model. You linearize a model at multiple operating points and obtain multiple open-loop and closed-loop transfer functions from the model.

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

Create slLinearizer Interface for Model

Open the model.

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



Use the `slLinearizer` command to create the interface.

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

```
slLinearizer linearization interface for "watertank":

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

The command-window display shows information about the `slLinearizer` interface. In this interface, the `OperatingPoints` property display shows that no operating point is specified.

Specify Multiple Operating Points for Linearization

You can linearize the model using trimmed operating points, the model initial condition, or simulation snapshot times. For this example, use trim points that you obtain for varying water-level reference heights.

```
opspec = operspec(mdl);
opspec.States(2).Known = 1;
opts = findopOptions('DisplayReport','off');

h = [10 15 20];

for ct = 1:numel(h)
    opspec.States(2).x = h(ct);
    Href = h(ct);
    ops(ct) = findop(mdl,opspec,opts);
end

sllin.OperatingPoints = ops;
```

Here, `h` specifies the different water-levels. `ops` is a 1 x 3 array of operating point objects. Each entry of `ops` is the model operating point at the corresponding water level. Configure the `OperatingPoints` property of `sllin` with `ops`. Now, when you obtain transfer functions from `sllin` using the `getIOTransfer`, `getLoopTransfer`, `getSensitivity`, and `getCompSensitivity` functions, the software returns a linearization for each specified operating point.

Each trim point is only valid for the corresponding reference height, represented by the `Href` parameter of the Desired Water Level block. So, configure `sllin` to vary this parameter accordingly.

```
param.Name = 'Href';
param.Value = h;

sllin.Parameters = param;
```

Analyze Plant Transfer Function

In the `watertank` model, the Water-Tank System block represents the plant. To obtain the plant transfer function, add the input and output signals of the Water-Tank System block as analysis points of `sllin`.

```
addPoint(sllin,{'watertank/PID Controller','watertank/Water-Tank System'})
sllin


slLinearizer linearization interface for "watertank":

2 Analysis points:
--------------------------
Point 1:
- Block: watertank/PID Controller
- Port: 1
Point 2:
- Block: watertank/Water-Tank System
- Port: 1

No permanent openings. Use the addOpening command to add new permanent openings.
Properties with dot notation get/set access:
      Parameters         : [1x1 struct], 1 parameters with sampling grid of size 1x3
            "Href", varying between 10 and 20.
      OperatingPoints    : [1x3 opcond.OperatingPoint]
      BlockSubstitutions : []
      Options            : [1x1 linearize.LinearizeOptions]
```

The first analysis point, which originates at the outport of the PID Controller block, is the input to the Water-Tank System block. The second analysis point is the output of the Water-Tank System block.

Obtain the plant transfer function from the input of the Water-Tank System block to the block output. To eliminate the effects of the feedback loop, specify the block output as a temporary loop opening.

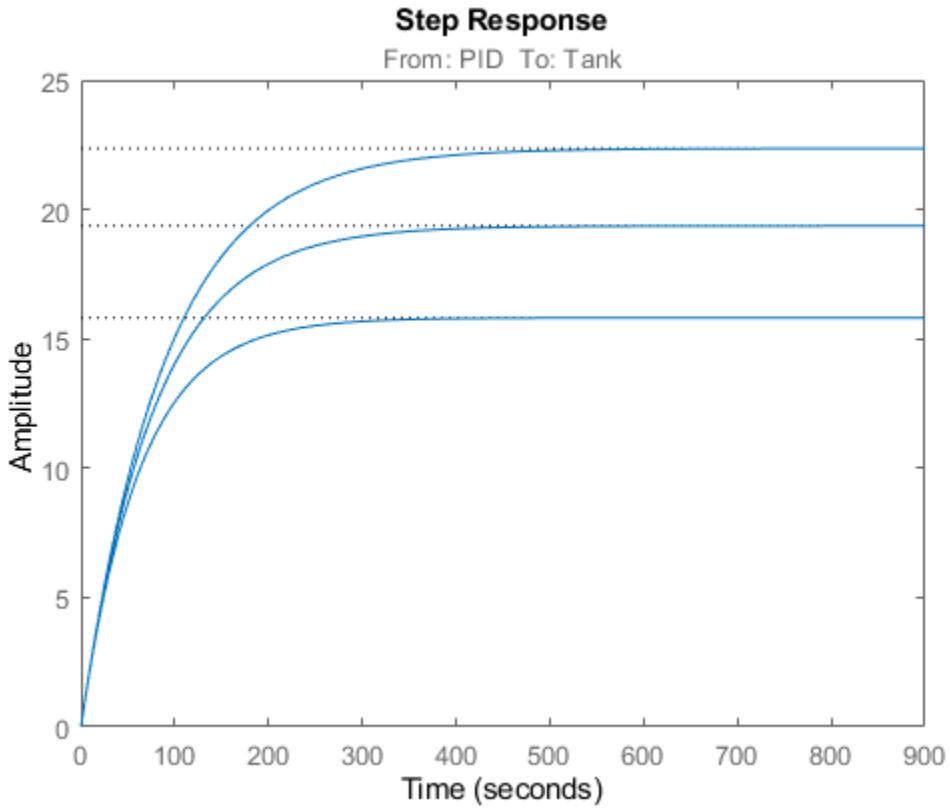
```
G = getIOTransfer(sllin,'PID','Tank','Tank');
```

In the call to `getIOTransfer`, `'PID'`, a portion of the block name `'watertank/PID Controller'`, specifies the first analysis point as the transfer function input. Similarly, `'Tank'`, a portion of the block name `'watertank/Water-Tank System'`, refers to the second analysis point. This analysis point is specified as the transfer function output (third input argument) and a temporary loop opening (fourth input argument).

The output, `G`, is a 1 x 3 array of continuous-time state-space models.

Plot the step response for `G`.

```
stepplot(G);
```



The step response of the plant models varies significantly at the different operating points.

Analyze Closed-Loop Transfer Function

The closed-loop transfer function is equal to the transfer function from the reference input, originating at the Desired Water Level block, to the plant output.

Add the reference input signal as an analysis point of `sllin`.

```
addPoint(sllin,'watertank/Desired  Water Level');
```

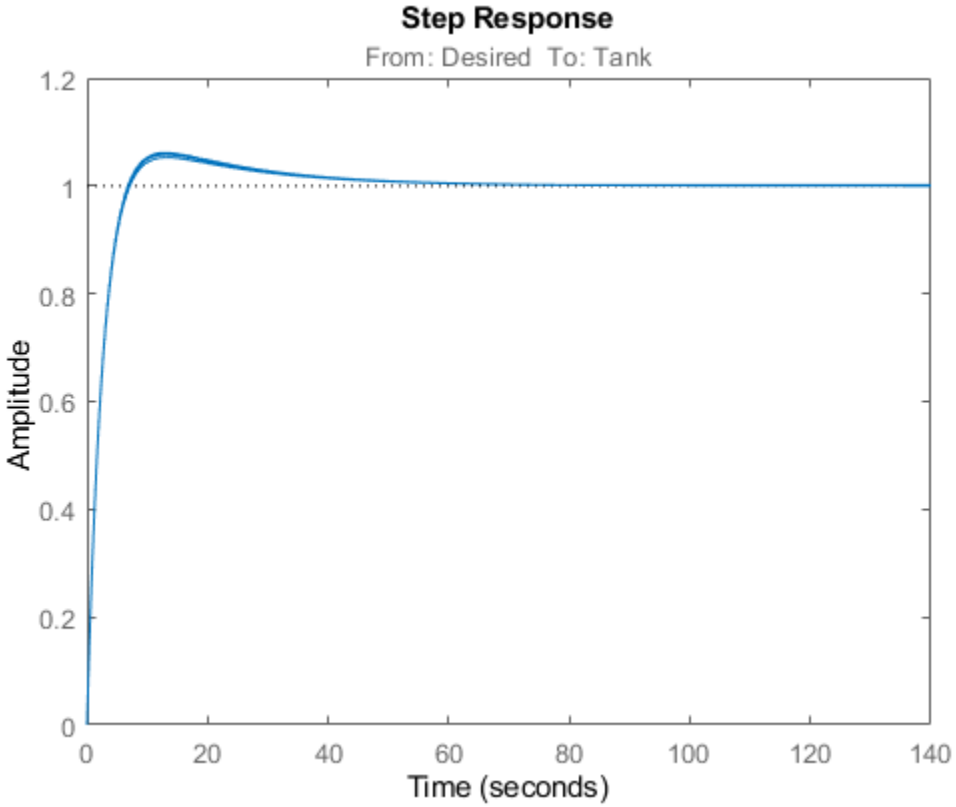
Obtain the closed-loop transfer function.

```
T = getIOTransfer(sllin,'Desired','Tank');
```

The output, `T`, is a 1 x 3 array of continuous-time state-space models.

Plot the step response for `T`.

```
stepplot(T);
```



Although the step response of the plant transfer function varies significantly at the three trimmed operating points, the controller brings the closed-loop responses much closer together at all three operating points.

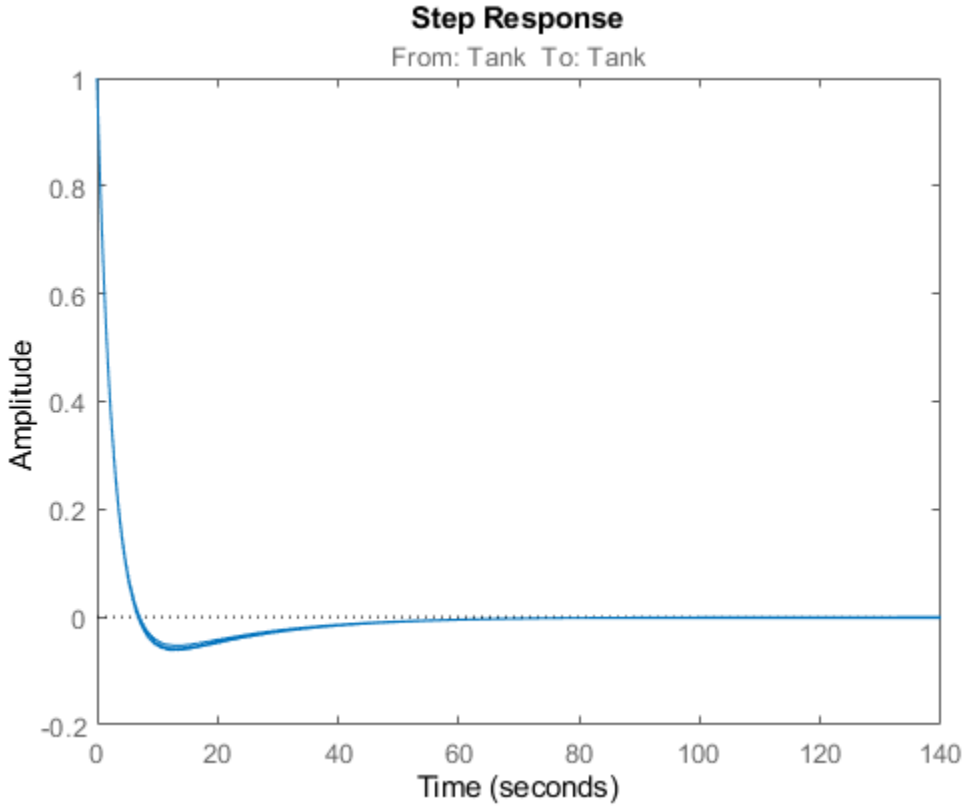
Analyze Sensitivity at Plant Output

```
S = getSensitivity(sllin,'Tank');
```

The software injects a disturbance signal and measures the output at the plant output. `S` is a 1 x 3 array of continuous-time state-space models.

Plot the step response for `S`.

```
stepplot(S);
```



The plot indicates that both models can reject a step disturbance at the plant output within 40 seconds.

See Also

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

More About

- “watertank Simulink Model”
- “Batch Compute Steady-State Operating Points for Multiple Specifications” on page 1-68
- “Vary Parameter Values and Obtain Multiple Transfer Functions” on page 3-21
- “Analyze Command-Line Batch Linearization Results Using Response Plots” on page 3-33

Analyze Command-Line Batch Linearization Results Using Response Plots

This example shows how to plot and analyze the step response for batch linearization results obtained at the command line. The term batch linearization results refers to the `ss` model array returned by the `slLinearizer` interface or `linearize` function. This array contains linearizations for varying parameter values, operating points, or both, such as illustrated in “Batch Linearize Model for Parameter Variations at Single Operating Point” on page 3-13 and “Vary Operating Points and Obtain Multiple Transfer Functions Using slLinearizer Interface” on page 3-28. You can use the techniques illustrated in this example to analyze the frequency response, stability, or sensitivity for batch linearization results.

Obtain Batch Linearization Results

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

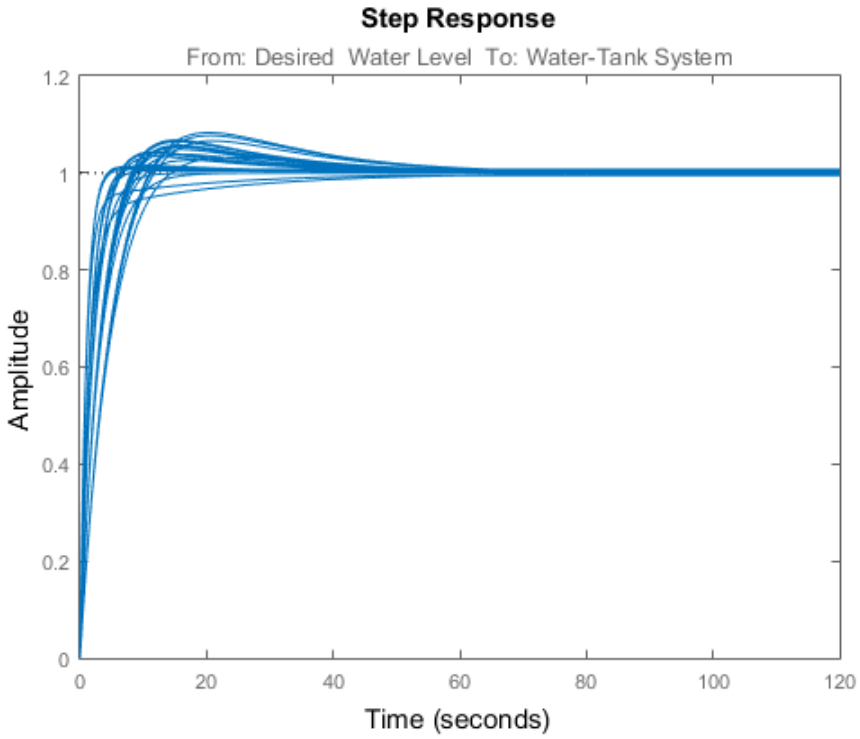
The following code obtains linearizations of the `watertank` model for four simulation snapshot times, `t = [0 1 2 3]`. At each snapshot time, the model parameters, `A` and `b`, are varied. The sample values for `A` are `[10 20 30]`, and the sample values for `b` are `[4 6]`. The `slLinearizer` interface includes analysis points at the reference signal and plant output.

```
open_system('watertank')
sllin = slLinearizer('watertank',{'watertank/Desired  Water Level',...
                                  'watertank/Water-Tank System'})

[A_grid,b_grid] = ndgrid([10,20,30],[4 6]);
params(1).Name = 'A';
params(1).Value = A_grid;
params(2).Name = 'b';
params(2).Value = b_grid;

sllin.Parameters = params;
sllin.OperatingPoints = [0,1,2,3];

linsys = getIOTransfer(sllin,'Desired  Water Level','Water-Tank System');
```

`linsys`, a 4-by-3-by-2 `ss` model array, contains the closed-loop transfer function of the linearized `watertank` model from the reference input to the plant output. The operating point varies along the first array dimension of `linsys`, and the parameters `A` and `b` vary along the second and third dimensions, respectively.

Plot Step Responses of the Linearized Models

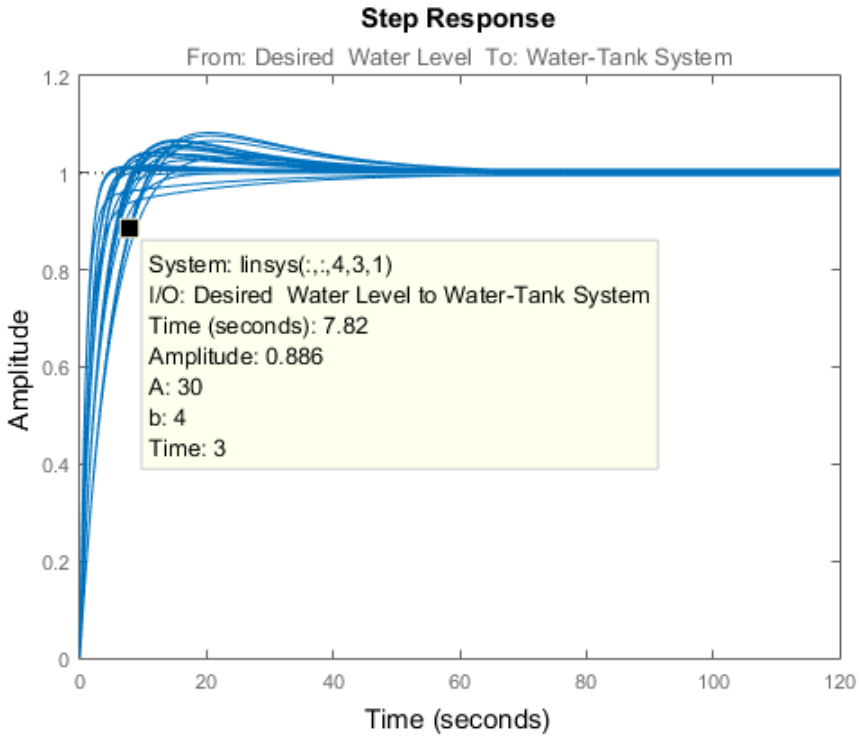
```
stepplot(linsys)
```



The step plot shows the responses of every model in the array. This plot shows the range of step responses of the system in the operating ranges covered by the parameter grid and snapshot times.

View Parameters and Snapshot Time of a Response

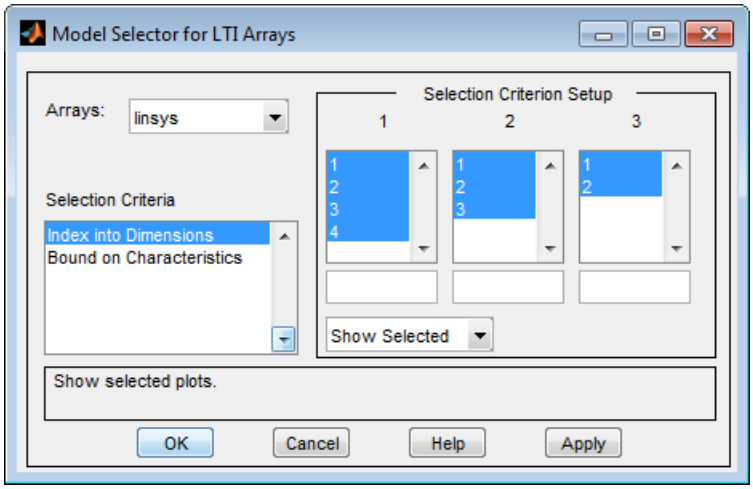
To view the parameters associated with a particular response, click the response on the plot.



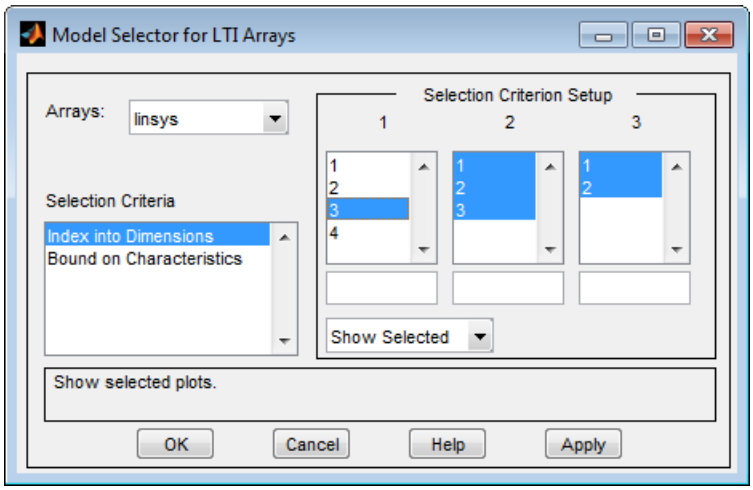
A data tip appears on the plot, providing information about the selected response and the related model. The last lines of the data tip show the parameter combination and simulation snapshot time that yielded this response. For example, in this previous plot, the selected response corresponds to the model obtained by setting `A` to `30` and `b` to `4`. The software linearized the model after simulating the model for three time units.

View Step Response of Subset of Results

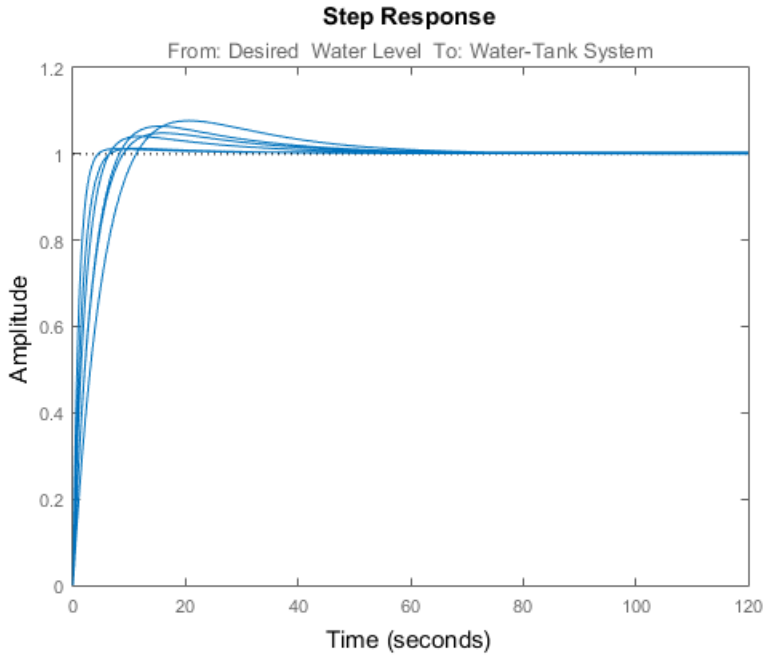
Suppose you want to view the responses for models linearized at a specific simulation snapshot time, such as two time units. Right-click the plot and select **Array Selector**. The Model Selector for LTI Arrays dialog box opens.



The **Selection Criterion Setup** panel contains three columns, one for each model array dimension of `linsys`. The first column corresponds to the simulation snapshot time. The third entry of this column corresponds to the simulation snapshot time of two time units, because the snapshot time array was `[0,1,2,3]`. Select only this entry in the first column.



Click **OK**. The plot displays the responses for only the models linearized at two time units.



Plot Step Response for Specific Parameter Combination and Snapshot Time

Suppose you want to examine only the step response for the model obtained by linearizing the `watertank` model at `t = 3`, for `A = 10` and `b = 4`. To do so, you can use the `SamplingGrid` property of `linsys`, which is specified as a structure. When you perform batch linearization, the software populates `SamplingGrid` with information regarding the variable values used to obtain the model. The variable values include each parameter that you vary and the simulation snapshot times at which you linearize the model. For example:

```
linsys(:,:,1).SamplingGrid

ans =

       A: 10
       b: 4
    Time: 0
```

Here `linsys(:,:,1)` refers to the first model in `linsys`. This model was obtained at simulation time `t = 0`, for `A = 10` and `b = 4`.

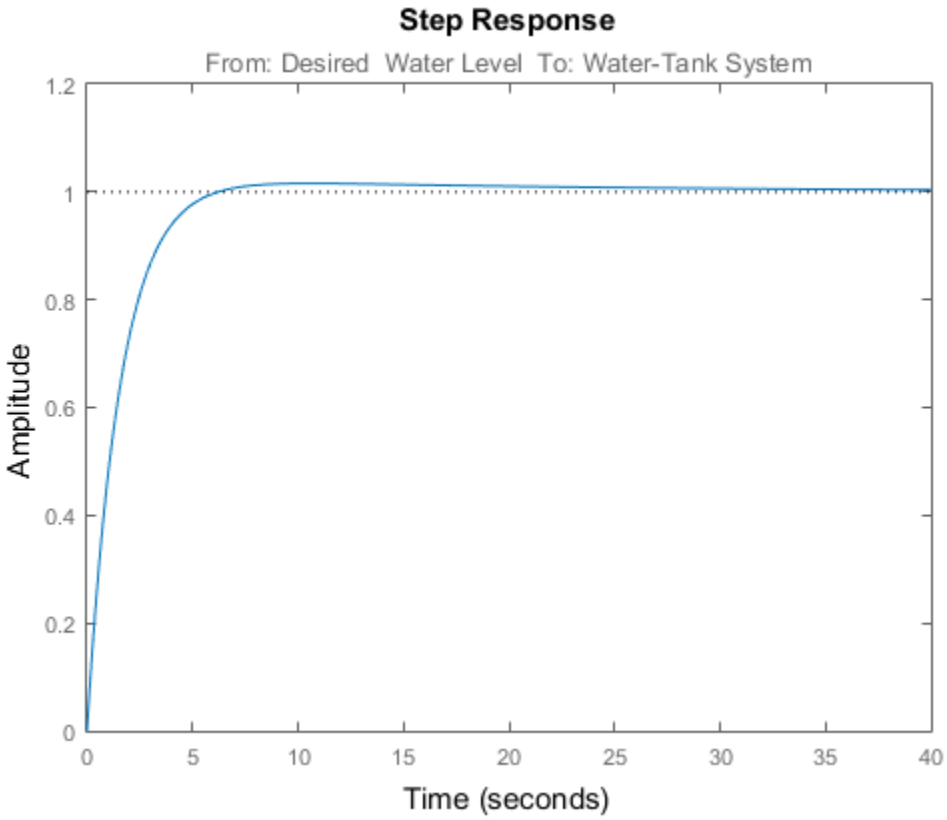
Use array indexing to extract from `linsys`the model obtained by linearizing the `watertank` model at `t = 3`, for `A = 10` and `b = 4`.

```
sg = linsys.SamplingGrid;
sys = linsys(:,:,sg.A == 10 & sg.B == 4 & sg.Time == 3);
```

The structure, `sg`, contains the sampling grid for all the models in `linsys`. The expression `sg.A == 10 & sg.B == 4 & sg.Time == 3` returns a logical array. Each entry of this array contains the logical evaluation of the expression for corresponding entries in `sg.A`, `sg.B`, and `sg.Time`. `sys`, a model array, contains all the `linsys` models that satisfy the expression.

View the step response for `sys`.

```
stepplot(sys)
```



See Also

Related Examples

- “Batch Linearize Model for Parameter Variations at Single Operating Point” on page 3-13
- “Vary Operating Points and Obtain Multiple Transfer Functions Using slLinearizer Interface” on page 3-28
- “Analyze Batch Linearization Results in Model Linearizer” on page 3-39
- “Validate Batch Linearization Results” on page 3-69

Analyze Batch Linearization Results in Model Linearizer

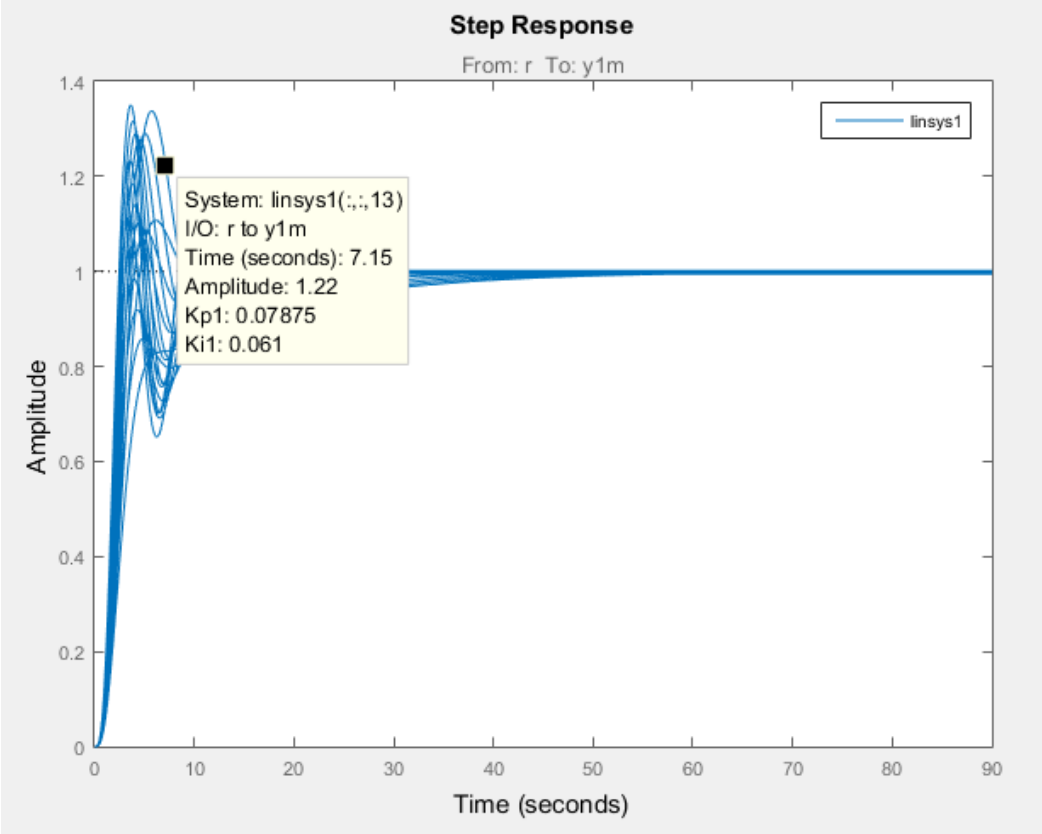
This example shows how to use response plots to analyze batch linearization results in **Model Linearizer**. The term batch linearization results refers to linearizations for varying parameter values, such as illustrated in “Batch Linearize Model for Parameter Value Variations Using Model Linearizer” on page 3-53. You can use the techniques illustrated in this example to analyze the frequency response, stability, and other system characteristics for batch linearization results.

View Parameters of a Response

For this example, suppose that you have batch linearized a model as described in “Batch Linearize Model for Parameter Value Variations Using Model Linearizer” on page 3-53. You have generated a step response plot of an array of linear models computed for a 2-D parameter grid, with variations of outer-loop controller gains `Ki1` and `Kp1`.

When you perform batch linearization, **Model Linearizer** generates a plot showing the responses of all linear models resulting from the linearization. You choose the response plot type, such as Step, Bode, or Nyquist, when you linearize. You can create additional plots at any time as described in “Analyze Results Using Model Linearizer Response Plots” on page 2-113.

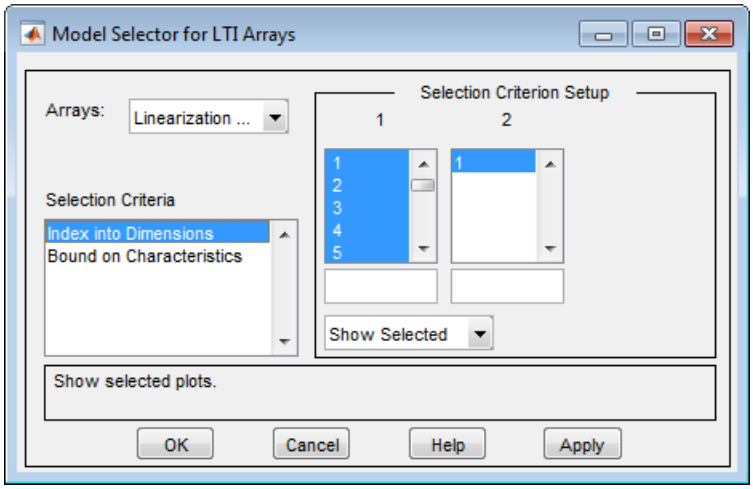
To view the parameters associated with a particular response, click the response on the plot.



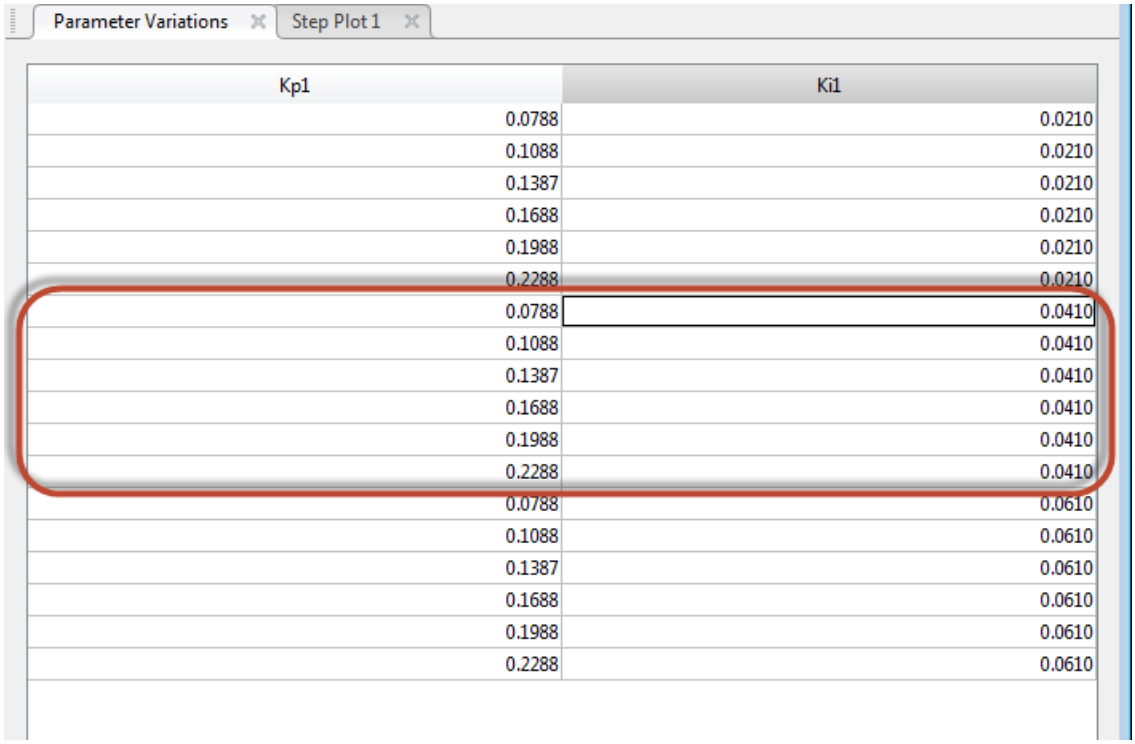
A data tip appears on the plot, providing information about the selected response and the related model. The last lines of the data tip show the parameter combination that yielded this response. For example, in this plot, the selected response corresponds to the model obtained by setting `Kp1` to `0.07875` and `Ki1` to `0.061`.

View Step Response of Subset of Results

Suppose you want to view the responses for only the models linearized at a specific `Ki1` value, the middle value `Ki1 = 0.0410`. Right-click the plot and select **Array Selector**. The Model Selector for LTI Arrays dialog box opens.



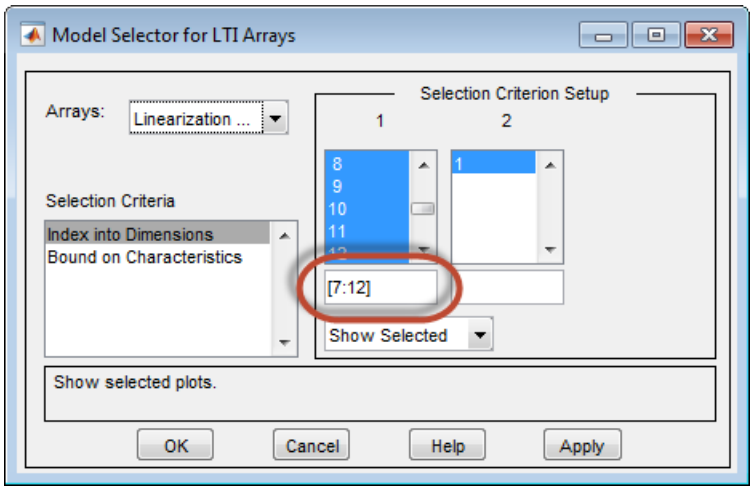
The **Selection Criterion Setup** panel contains two columns, one for each model array dimension of `linsys1`. **Model Linearizer** flattens the 2-D parameter grid into a one-dimensional array, so that variations in both `Kp1` and `Ki1` are represented along the indices shown in column **1**. To determine which entries in this array correspond to `Ki1 = 0.0410`, examine the **Parameter Variations** table.



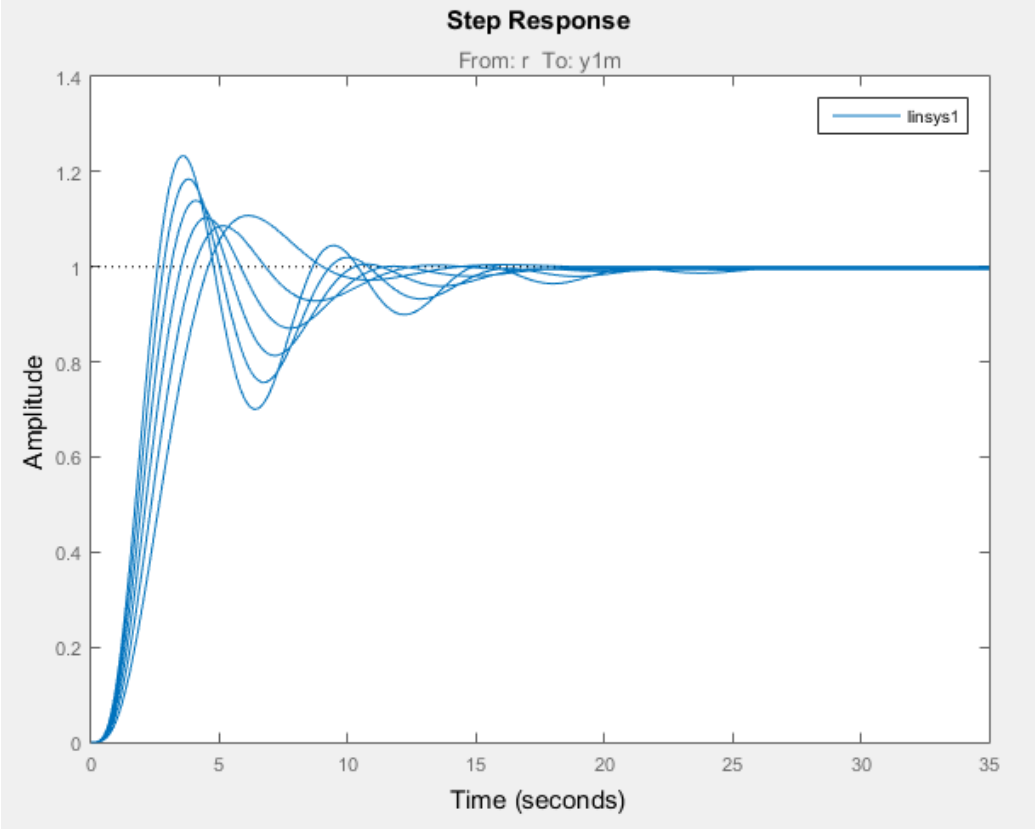
Kp1	Ki1
0.0788	0.0210
0.1088	0.0210
0.1387	0.0210
0.1688	0.0210
0.1988	0.0210
0.2288	0.0210
0.0788	0.0410
0.1088	0.0410
0.1387	0.0410
0.1688	0.0410
0.1988	0.0410
0.2288	0.0410
0.0788	0.0610
0.1088	0.0610
0.1387	0.0610
0.1688	0.0610
0.1988	0.0610
0.2288	0.0610

The `Ki1 = 0.0410` values are the seventh to twelfth entries in this table. Therefore, you want to select array indices 7–12.

In the Model Selector for LTI Arrays dialog box, enter `[7:12]` in the field below column **1**. The selection in the column changes to reflect this subset of the array.

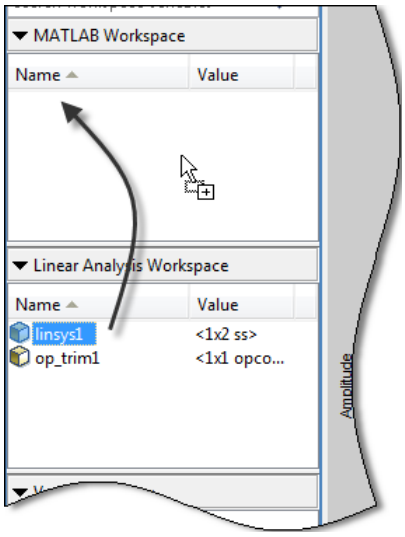


Click **OK**. The step plot displays responses only for the models with `Ki1 = 0.0410`.



Export Array to MATLAB Workspace

You can export the model array to the MATLAB workspace to perform further analysis or control design. To do so, in the **Model Linearizer**, in the **Data Browser**, drag the array from Linear Analysis Workspace to the MATLAB workspace.



You can then use Control System Toolbox control design tools, such as the Linear System Analyzer app, to analyze linearization results. Or, use Control System Toolbox control design tools, such as `pidtune` or **Control System Designer**, to design controllers for the linearized systems.

See Also

More About

- “What Is Batch Linearization?” on page 3-2
- “Batch Linearize Model for Parameter Value Variations Using Model Linearizer” on page 3-53

Specify Parameter Samples for Batch Linearization

About Parameter Samples

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

When using any of the Simulink Control Design linearization tools (or tuning with `slTuner` or Control System Tuner) you can specify a set of block parameter values at which to linearize the model. The full set of values is called a parameter grid or parameter samples. The tools batch-linearize the model, computing a linearization for each value in the parameter grid. You can vary multiple parameters, thus extending the parameter grid dimension. When using the command-line linearization tools, the `linearize` command or the `slLinearizer` or `slTuner` interfaces, you specify the parameter samples using a structure with fields `Name` and `Value`. In the **Model Linearizer** or **Control System Tuner**, you use the graphical interface to specify parameter samples.

Which Parameters Can Be Sampled?

You can vary any model parameter whose value is given by a variable in the model workspace, the MATLAB workspace, or a data dictionary. In cases where the varying parameters are all tunable, the linearization tools require only one model compilation to compute transfer functions for varying parameter values. This efficiency is especially advantageous for models that are expensive to compile repeatedly.

For more information, see “Batch Linearization Efficiency When You Vary Parameter Values” on page 3-7.

Vary Single Parameter at the Command Line

To vary the value of a single parameter for batch linearization with `linearize`, `slLinearizer`, or `slTuner`, specify the parameter grid as a structure having two fields. The `Name` field contains the name of the workspace variable that specifies the parameter. The `Value` field contains a vector of values for that parameter to take during linearization.

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

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

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

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

Parameter Sample-Space Type	How to Specify the Parameter Samples
Linearly varying	`param.Value = linspace(A_min,A_max,num_samples)`

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

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

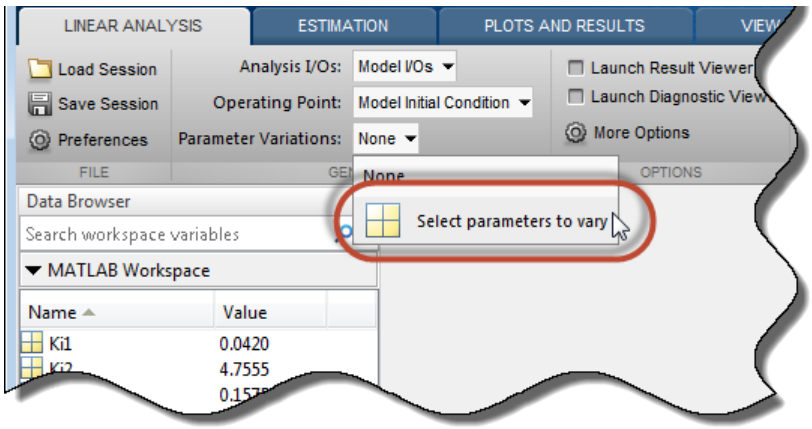
```
param.Name = 'Kpid(1)';
param.Value = Kpvals;
```

After you create the structure `param`:

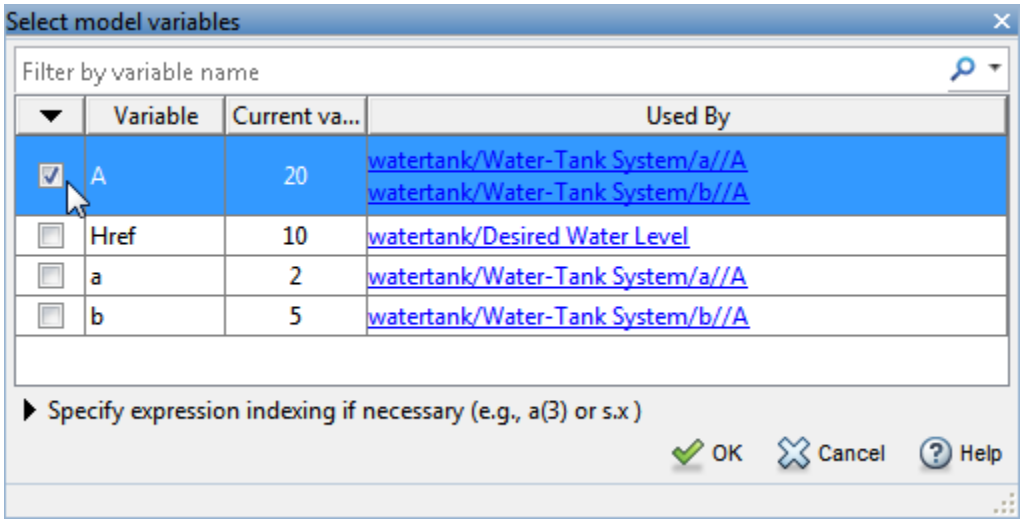
- Pass it to `linearize` as the `param` input argument.
- Pass it to `slLinearizer` as the `param` input argument, when creating an `slLinearizer` interface.
- Set the `Parameters` property of an existing `slLinearizer` interface to `param`.

Vary Single Parameter in Graphical Tools

To specify variations of a single parameter for batch linearization in **Model Linearizer**, on the **Linear Analysis** tab, in the **Parameter Variations** drop-down list, click `Select parameters to vary`. (In **Control System Tuner**, the **Parameter Variations** drop-down list is on the **Control System** tab.)



Click **Manage Parameters**. In the Select model variables dialog box, check the parameter to vary. The table lists all variables in the MATLAB workspace and the model workspace that are used in the model, whether tunable or not.

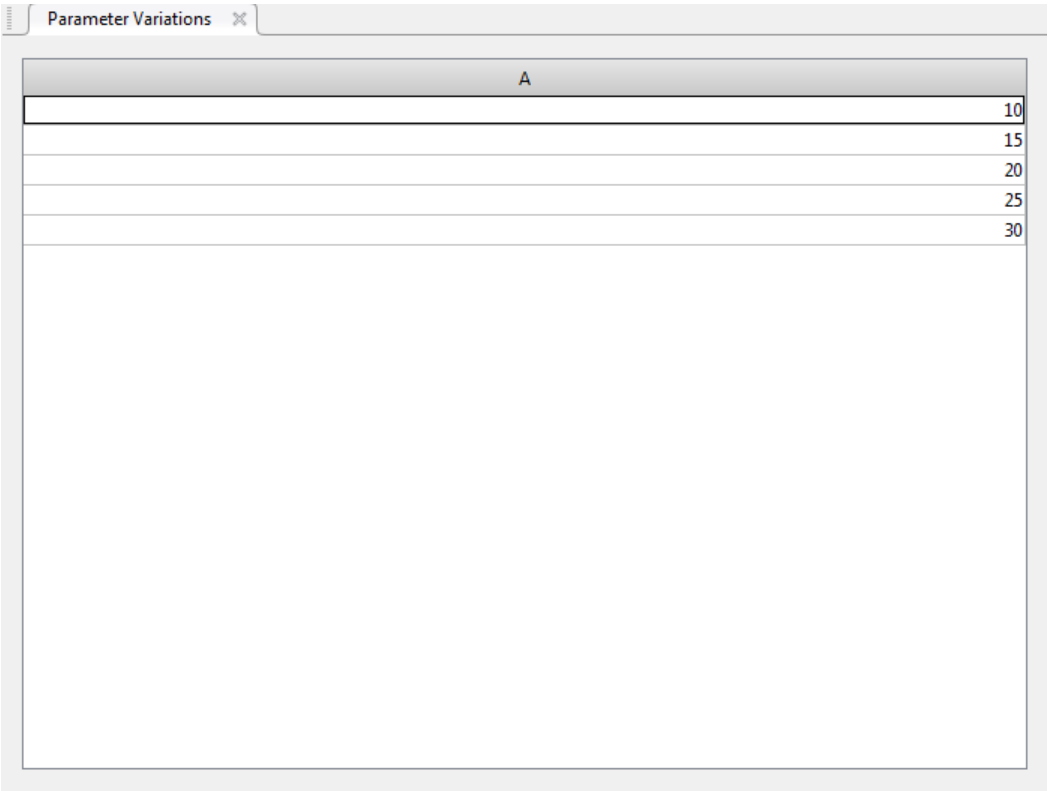


Note If the parameter is not a scalar variable, click **Specify expression indexing if necessary** and enter an expression that resolves to a numeric scalar value. For example, if `A` is a vector, enter `A(3)` to specify the third entry in `A`. If `A` is a structure and the scalar parameter you want to vary is the `Value` field of that structure, enter `A.Value`. The indexed variable appears in the variable list.

Click **OK**. The selected variable appears in the **Parameter Variations** table. Use the table to specify parameter values manually, or generate values automatically.

Manually Specify Parameter Values

To specify the values manually, add rows to the table by clicking **Insert Row** and selecting either `Insert Row Above` or `Insert Row Below`. Then, edit the values in the table as needed.

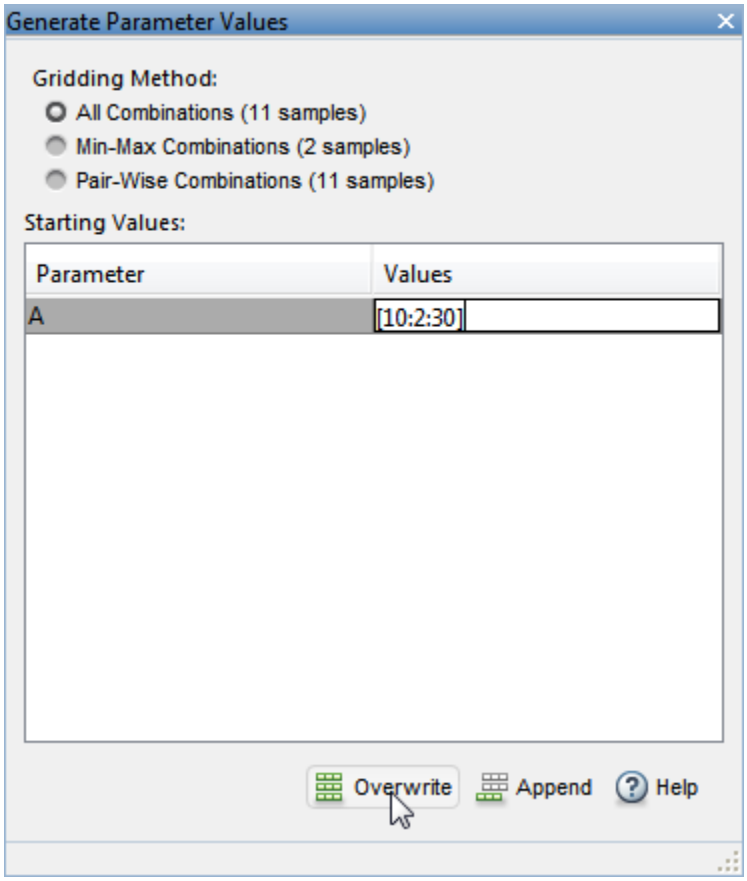


When you return to the **Linear Analysis** tab and linearize the model, **Model Linearizer** linearizes at all parameter values listed in the **Parameter Variations** table.

Note In Control System Tuner, when you are finished specifying your parameter variations, you must apply the changes before continuing with tuning. To do so, in the **Parameter Variations** tab, click **Apply**. Control System Tuner applies the specified parameter variations, relinearizes your model, and updates all existing plots.

Automatically Generate Parameter Values

To generate values automatically, click **Generate Values**. In the Generate Parameter Values dialog box, in the **Values** column, enter an expression for the parameter values you want for the variable. For example, enter an expression such as `linspace(A_min,A_max,num_samples)`, or `[10:2:30]`.



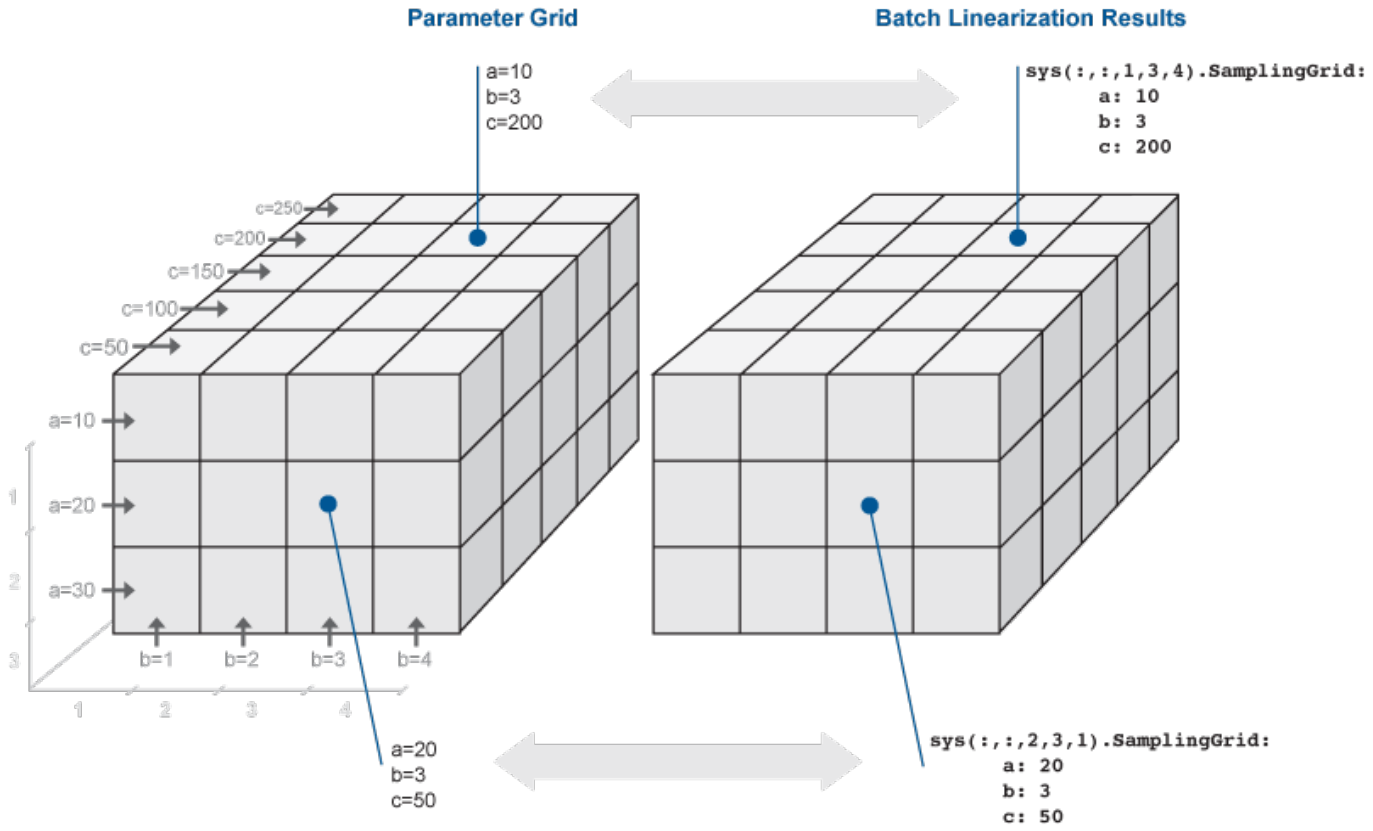
Click **Overwrite** to replace the values in the **Parameter Variations** table with the generated values.

When you return to the **Linear Analysis** tab and linearize the model, **Model Linearizer** computes a linearization for each of these parameter values.

Note In Control System Tuner, when you are finished specifying your parameter variations, you must apply the changes before continuing with tuning. To do so, in the **Parameter Variations** tab, click **Apply**. Control System Tuner applies the specified parameter variations, relinearizes your model, and updates all existing plots.

Multi-Dimension Parameter Grids

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



Here, you vary the values of three parameters, *a*, *b*, and *c*. The samples form a 3-by-4-by-5 grid. When batch linearizing your model, the `ss` model array, `sys`, is the batch result. Similarly, when batch trimming your model, you get an array of operating point objects.

Vary Multiple Parameters at the Command Line

To vary the value of multiple parameters for batch linearization with `linearize`, `slLinearizer`, or `slTuner`, specify parameter samples as a structure array. The structure has an entry for each parameter whose value you vary. The structure for each parameter is the same as described in “Vary Single Parameter at the Command Line” on page 3-43. You can specify the `Value` field for a parameter to be an array of any dimension. However, the size of the `Value` field must match for all parameters. Corresponding array entries for all the parameters, also referred to as a parameter grid point, must map to a desired parameter combination. When the software linearizes the model, it computes a linearization — an `ss` model — for each grid point. The software populates the `SamplingGrid` property of each linearized model with information about the parameter grid point that the model corresponds to.

Specify Full Grid

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

$a = \{a1, a2\}$
$b = \{b1, b2\}$

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

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

Create a rectangular parameter grid using `ndgrid`.

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

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

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

```
>> A

A =

     1     1
     2     2

>> B

B =

     3     4
     3     4
```

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

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

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

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

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

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

Create a 1 x N structure array.

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

Specify Subset of Full Grid

If your model is complex or you vary the value of many parameters, linearizing the model for the full grid can become expensive. In this case, you can specify a subset of the full grid using a table-like approach. Using the example in “Specify Full Grid” on page 3-48, suppose you want to linearize the model for the following combinations of a and b:

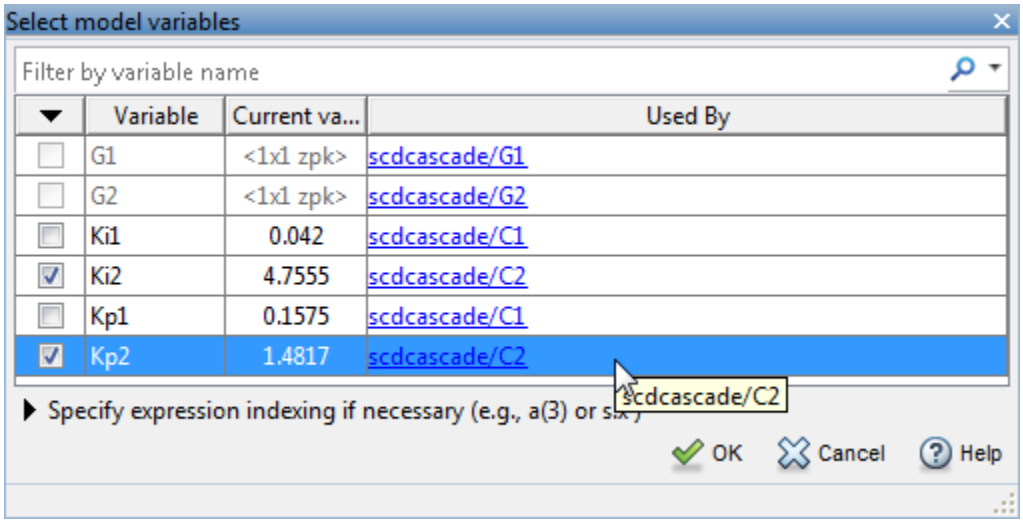
$$\{(a_1, b_1), (a_1, b_2)\}$$

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

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

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

Vary Multiple Parameters in Graphical Tools

To vary the value of multiple parameters for batch linearization in **Model Linearizer** or **Control System Tuner**, open the Select model variables dialog box, as described in "Vary Single Parameter in Graphical Tools" on page 3-44. In the dialog box, check all variables you want to vary.



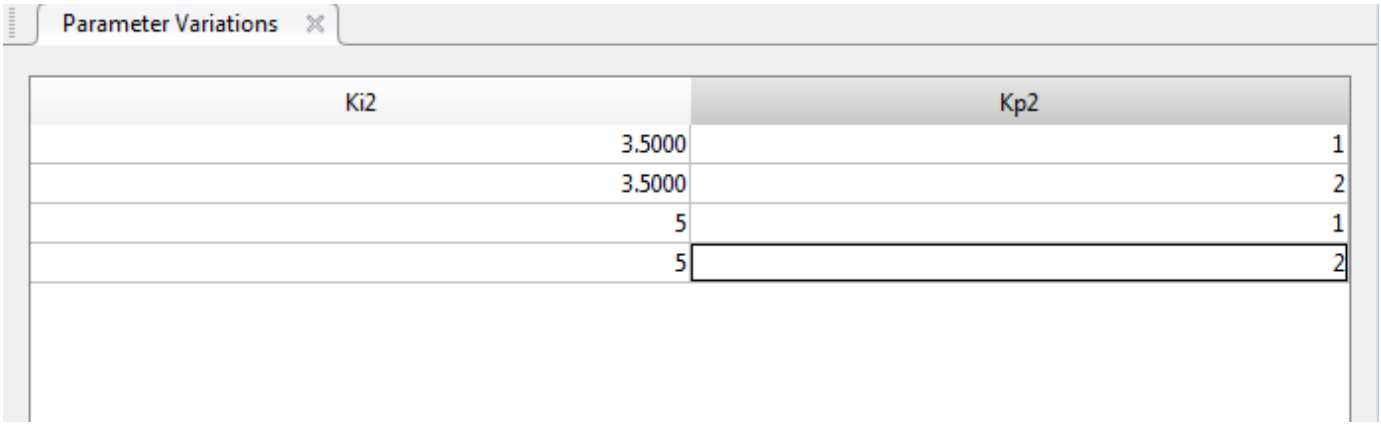
Note If a parameter you want to vary is not a scalar variable, click **Specify expression indexing if necessary** and enter an expression that resolves to a scalar value. For example, if `A` is a vector, enter `A(3)` to specify the third entry in `A`. If `A` is a structure and the scalar parameter you want to vary is the `Value` field of that structure, enter `A.Value`. The indexed variable appears in the variable list.

Click **OK**. The selected variables appear in the **Parameter Variations** table. Each column in the table corresponds to one selected variable. Each row in the table represents one full set of parameter values at which to linearize the model. When you linearize, **Model Linearizer** computes as many linear models as there are rows in the table. Use the table to specify combinations of parameter values manually, or generate value combinations automatically.

Manually Specify Parameter Values

To specify the values manually, add rows to the table by clicking **Insert Row** and selecting either `Insert Row Above` or `Insert Row Below`. Then, edit the values in the table as needed. For

example, the following table specifies linearization at four parameter-value pairs: `(Ki2,Kp2) =` (3.5,1), (3.5,2), (5,1), and (5,2).

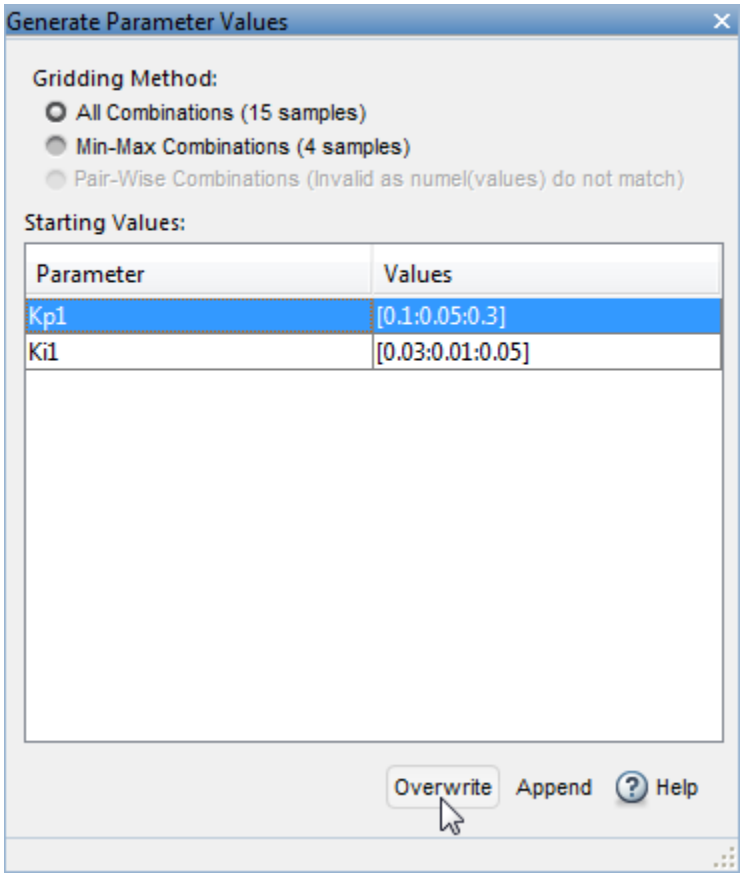


When you return to the **Linear Analysis** tab and linearize the model, **Model Linearizer** computes a linearization for each of these parameter-value pairs.

Note In Control System Tuner, when you are finished specifying your parameter variations, you must apply the changes before continuing with tuning. To do so, in the **Parameter Variations** tab, click **Apply**. Control System Tuner applies the specified parameter variations, relinearizes your model, and updates all existing plots.

Automatically Generate Parameter Values

To generate values automatically, click **Generate Values**. In the Generate Parameter Values dialog box, in the Values column, enter an expression for the parameter values you want for each variable, such as `linspace(A_min,A_max,num_samples)`, or `[10:2:30]`. For example, the following entry generates parameter-value pairs for all possible combinations of `Kp1 = [0.1,0.15,0.2,0.25,0.3]` and `Kp2 = [0.03,0.04,0.05]`.



Click **Overwrite** to replace the values in the **Parameter Variations** table with the generated values.

When you return to the **Linear Analysis** tab and linearize the model, **Model Linearizer** computes a linearization for each of these parameter-value pairs.

Note In Control System Tuner, when you are finished specifying your parameter variations, you must apply the changes before continuing with tuning. To do so, in the **Parameter Variations** tab, click **Apply**. Control System Tuner applies the specified parameter variations, relinearizes your model, and updates all existing plots.

See Also

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

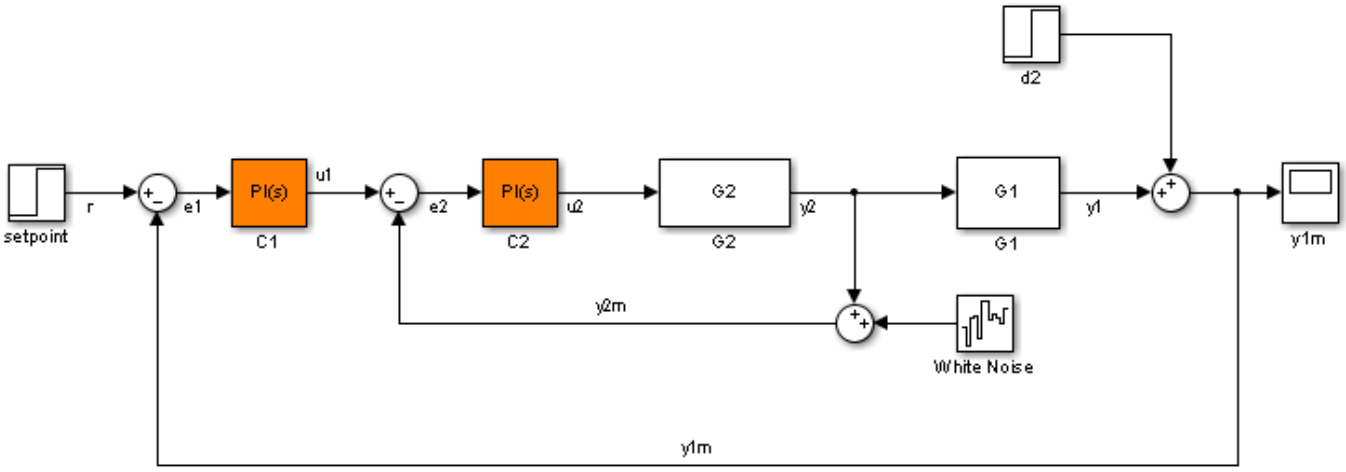
More About

- “Batch Linearization Efficiency When You Vary Parameter Values” on page 3-7
- “Batch Linearize Model for Parameter Value Variations Using Model Linearizer” on page 3-53
- “Batch Linearize Model for Parameter Variations at Single Operating Point” on page 3-13
- “Vary Parameter Values and Obtain Multiple Transfer Functions” on page 3-21

Batch Linearize Model for Parameter Value Variations Using Model Linearizer

This example shows how to use the **Model Linearizer** to batch linearize a Simulink model. You vary model parameter values and obtain multiple open-loop and closed-loop transfer functions from the model.

The `scdcascade` model used for this example contains a pair of cascaded feedback control loops. Each loop includes a PI controller. The plant models, G1 (outer loop) and G2 (inner loop), are LTI models. In this example, you use **Model Linearizer** to vary the PI controller parameters and analyze the inner-loop and outer-loop dynamics.

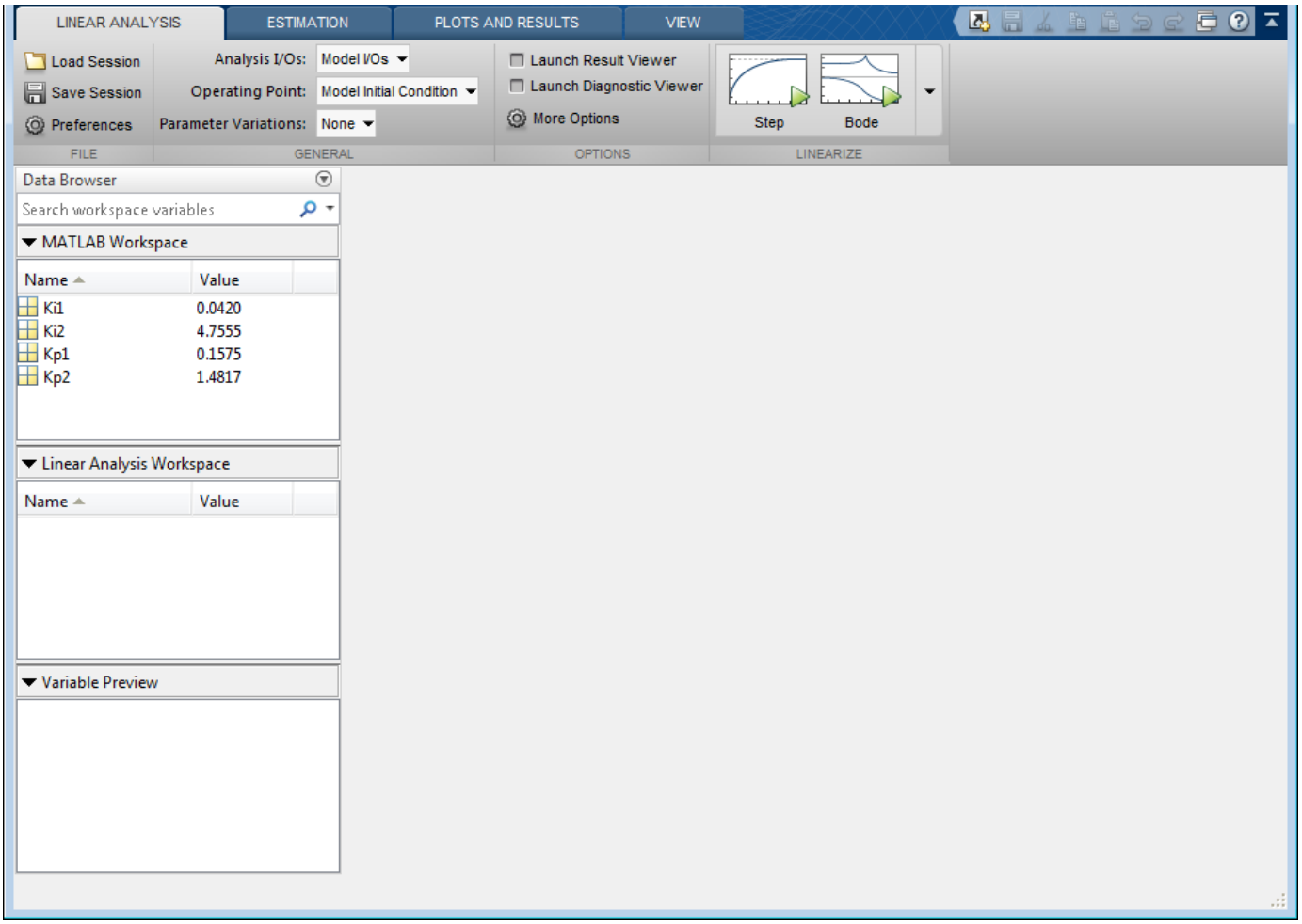


Open Model Linearizer for the Model

At the MATLAB command line, open the Simulink model.

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

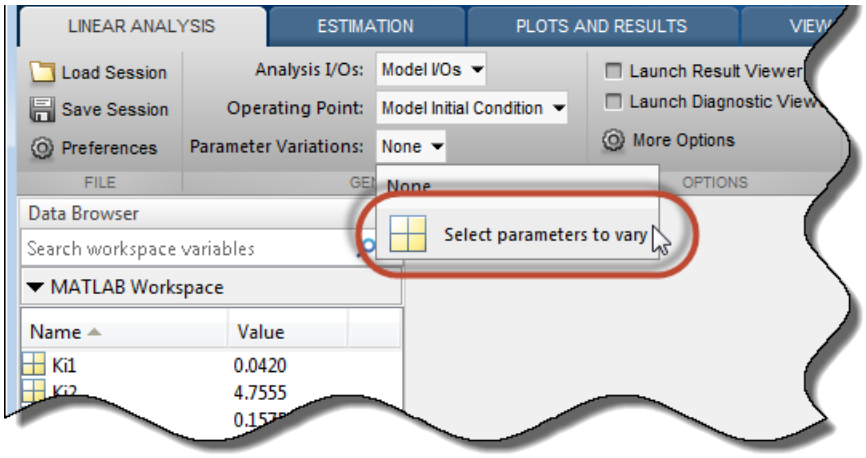
To open the **Model Linearizer**, in the Simulink model window, in the **Apps** gallery, click **Model Linearizer**.



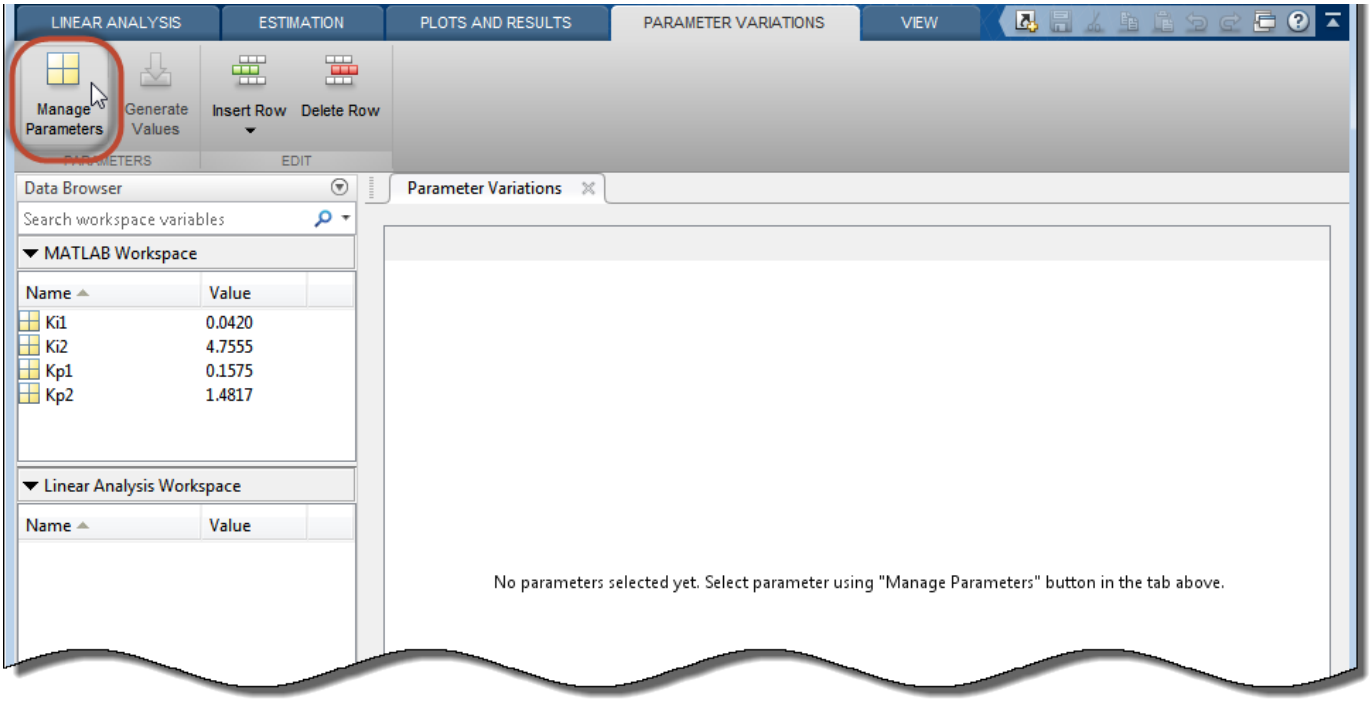
Vary the Inner-Loop Controller Gains

To analyze the behavior of the inner loop, very the gains of the inner-loop PI controller, `C2`. As you can see by inspecting the controller block, the proportional gain is the variable `Kp2`, and the integral gain is `Ki2`. Examine the performance of the inner loop for two different values of each of these gains.

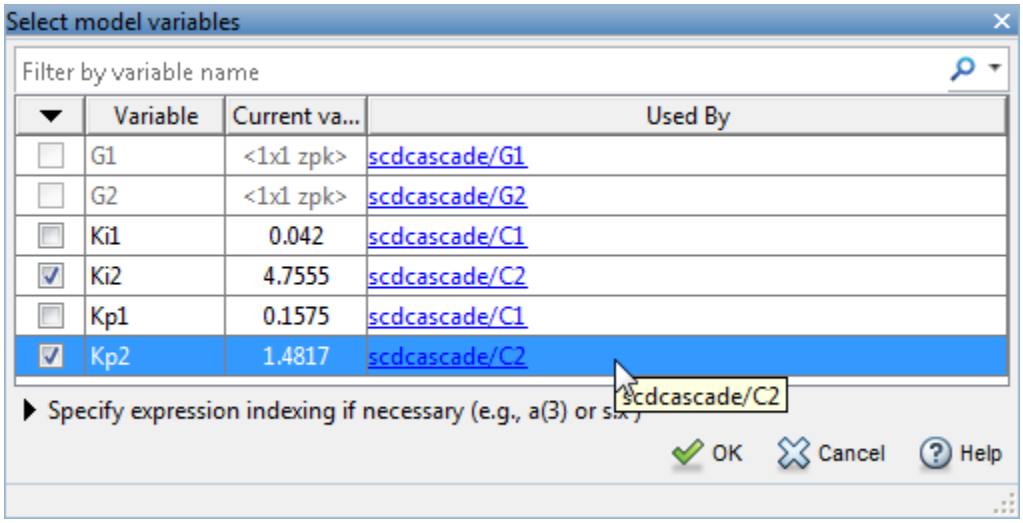
In the **Parameter Variations** drop-down list, click `Select parameters to vary`.



The **Parameter Variations** tab opens. click **Manage Parameters**.

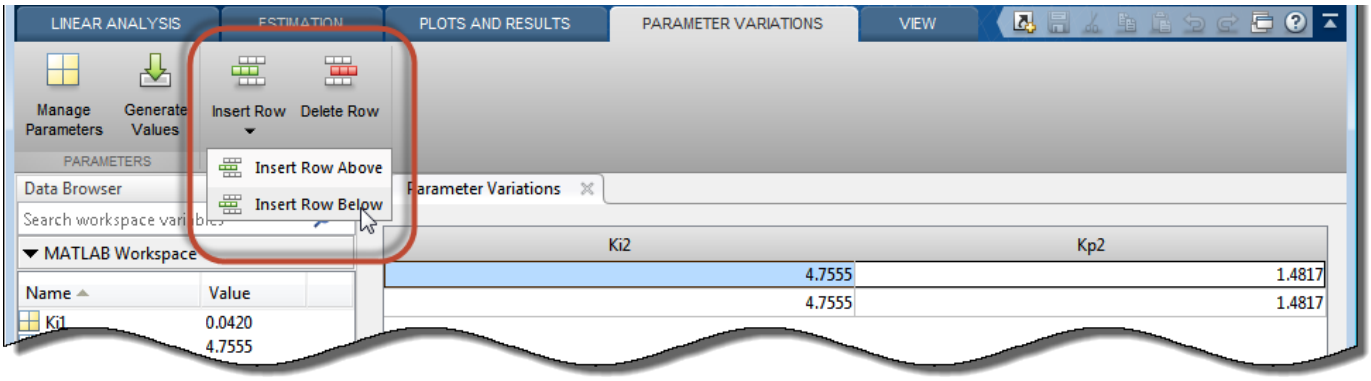


In the Select model variables dialog box, check the parameters to vary, `Ki2` and `Kp2`.

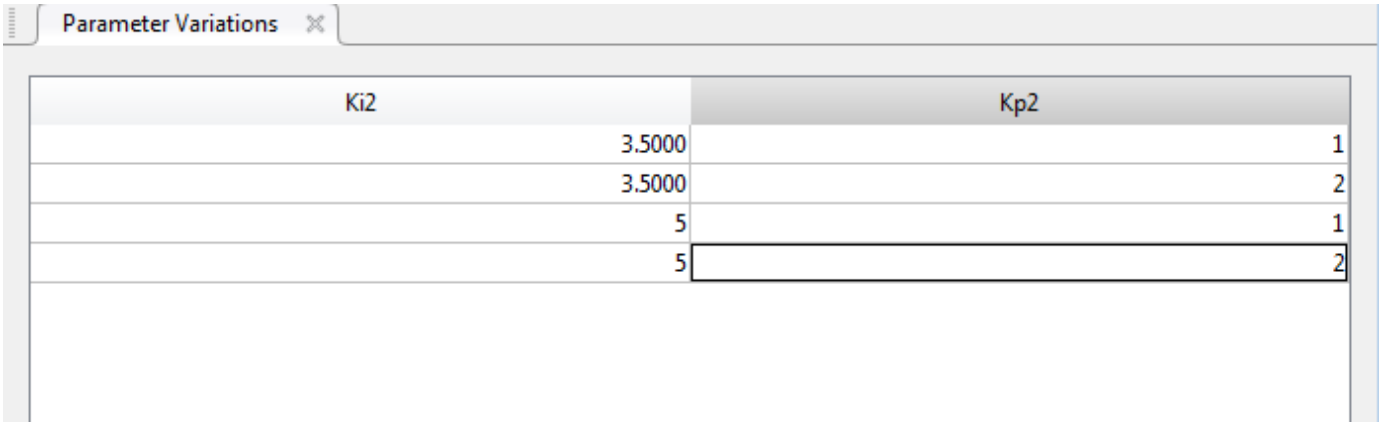


The selected variables appear in the **Parameter Variations** table. Each column in the table corresponds to one of the selected variables. Each row in the table represents one `(Ki2,Kp2)` pair at which to linearize. These parameter-value combinations are called parameter samples. When you linearize, **Model Linearizer** computes as many linear models as there are parameter samples, or rows in the table.

Specify the parameter samples at which to linearize the model. For this example, specify four `(Ki2,Kp2)` pairs, `(Ki2,Kp2)` = (3.5,1), (3.5,2), (5,1), and (5,2). Enter these values in the table manually. To do so, select a row in the table. Then, select **Insert Row** > **Insert Row Below** twice.



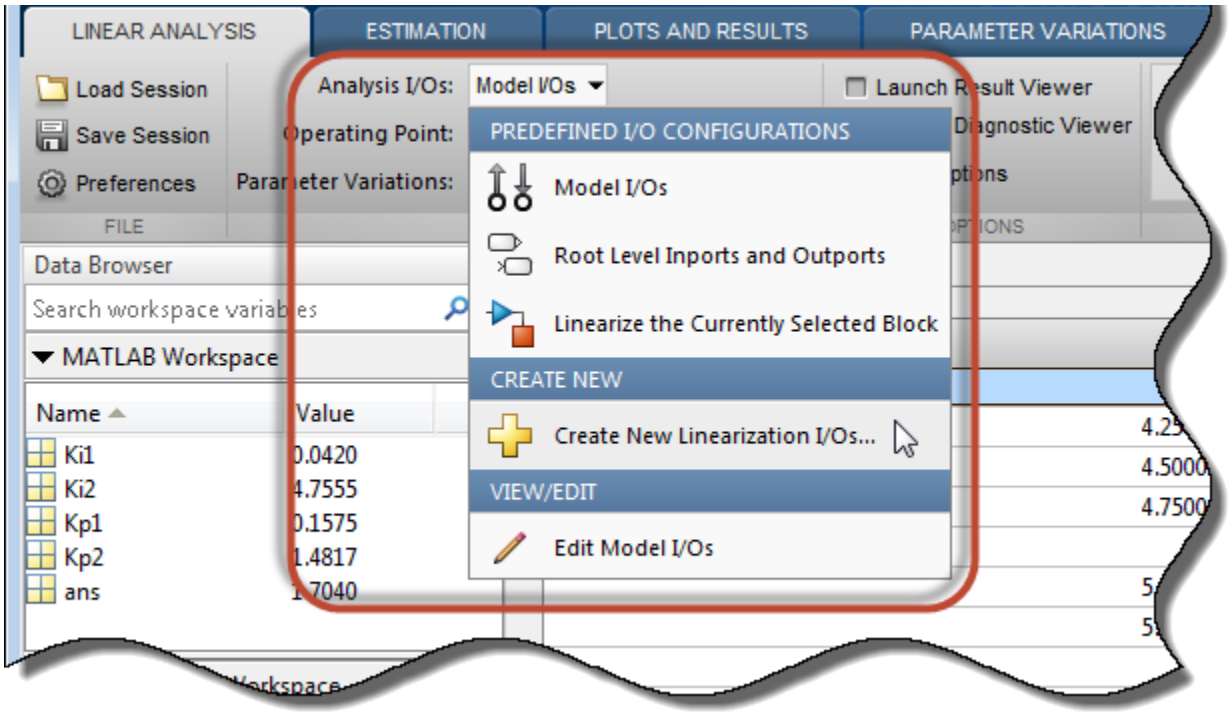
Edit the values in the table as shown to specify the four `(Ki2,Kp2)` pairs.



Tip For more details about specifying parameter values, see “Specify Parameter Samples for Batch Linearization” on page 3-43

Analyze the Inner Loop Closed-Loop Response

To analyze the inner-loop performance, extract a transfer function from the inner-loop input `u1` to the inner-plant output `y2`, computed with the outer loop open. To specify this I/O for linearization, in the **Linear Analysis** tab, in the **Analysis I/Os** drop-down list, select `Create New Linearization I/Os`.

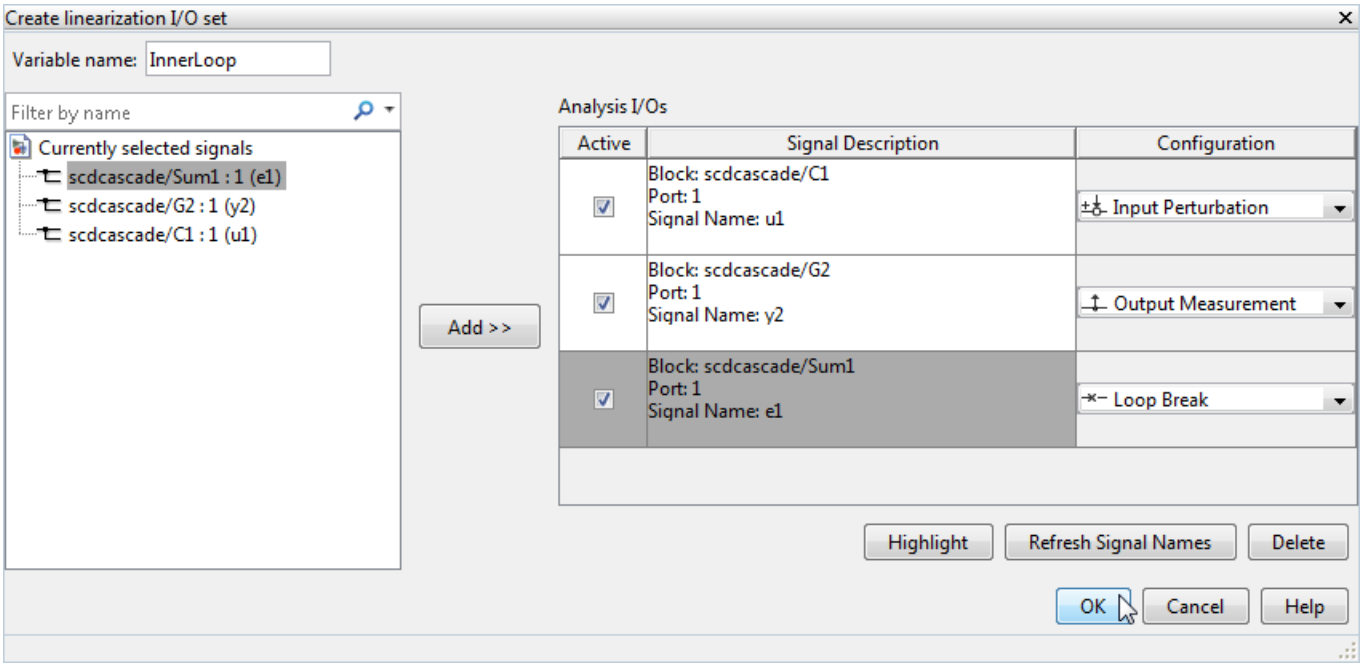


Specify the I/O set by creating:

- An input perturbation point at `u1`

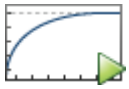
- An output measurement point at `y2`
- A loop break at `e1`

Name the I/O set by typing `InnerLoop` in the **Variable name** field of the Create linearization I/O set dialog box. The configuration of the dialog box is as shown.

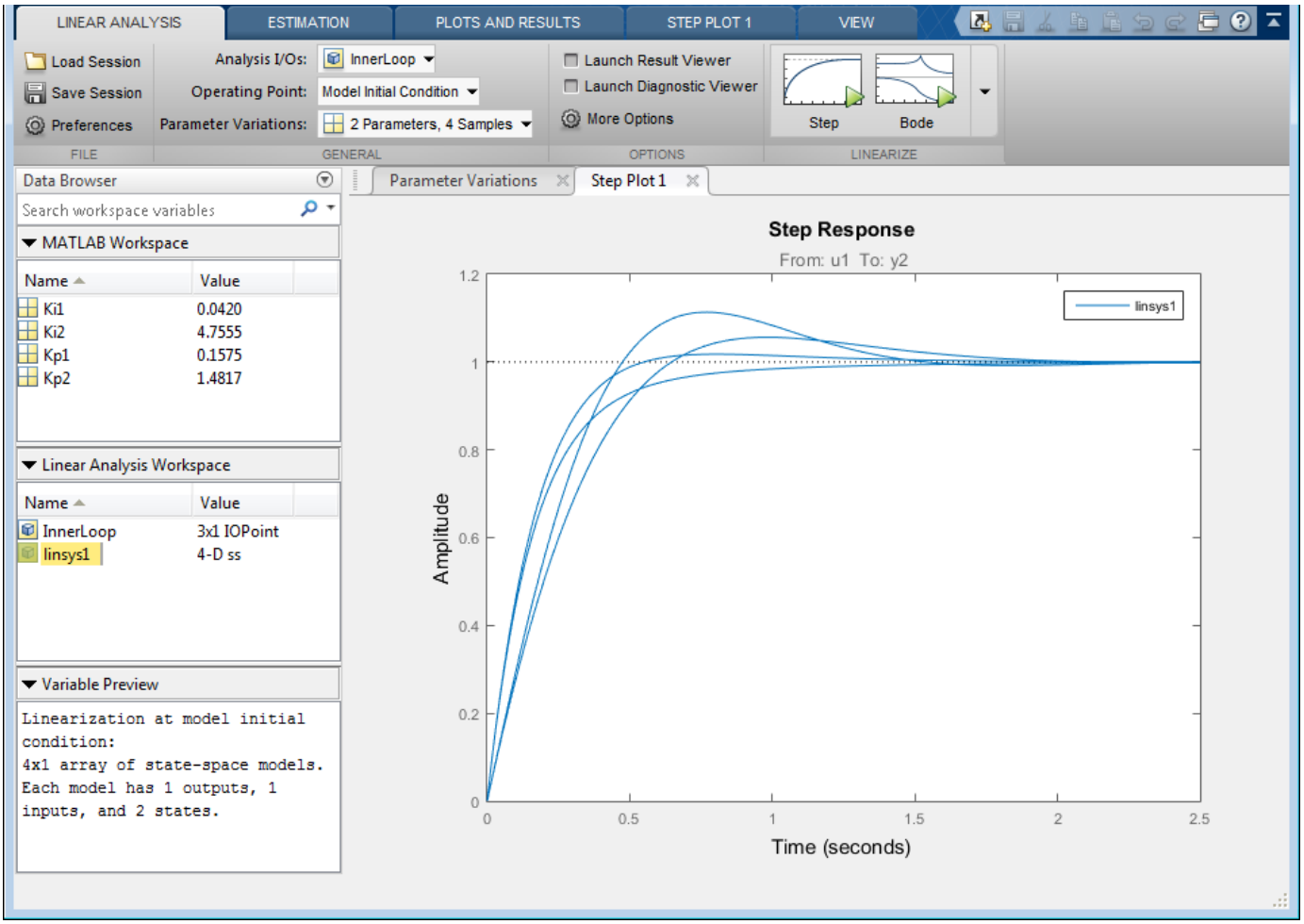


Tip For more information about specifying linearization I/Os, see “Specify Portion of Model to Linearize” on page 2-10.

Click **OK**.

Now that you have specified the parameter variations and the analysis I/O set for the inner loop, linearize the model and examine a step response plot. Click  **Step**.

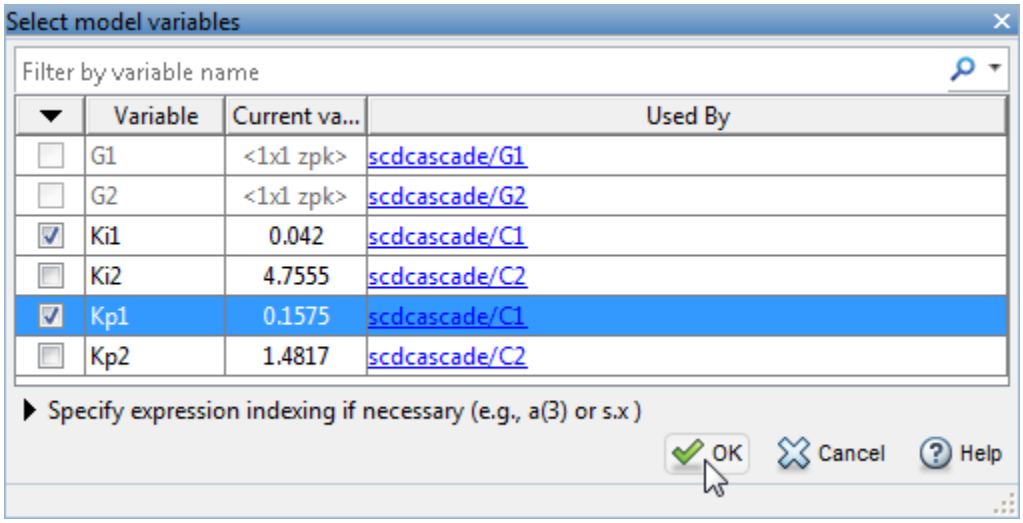
Model Linearizer linearizes the model at each of the parameter samples you specified in the Parameter Variations table. A new variable, `linsys1`, appears in the Linear Analysis Workspace section of the Data Browser. This variable is an array of state-space (`ss`) models, one for each `(Ki2,Kp2)` pair. The plot shows the step responses of all the entries in `linsys1`. This plot gives you a sense of the range of step responses of the system in the operating ranges covered by the parameter grid.



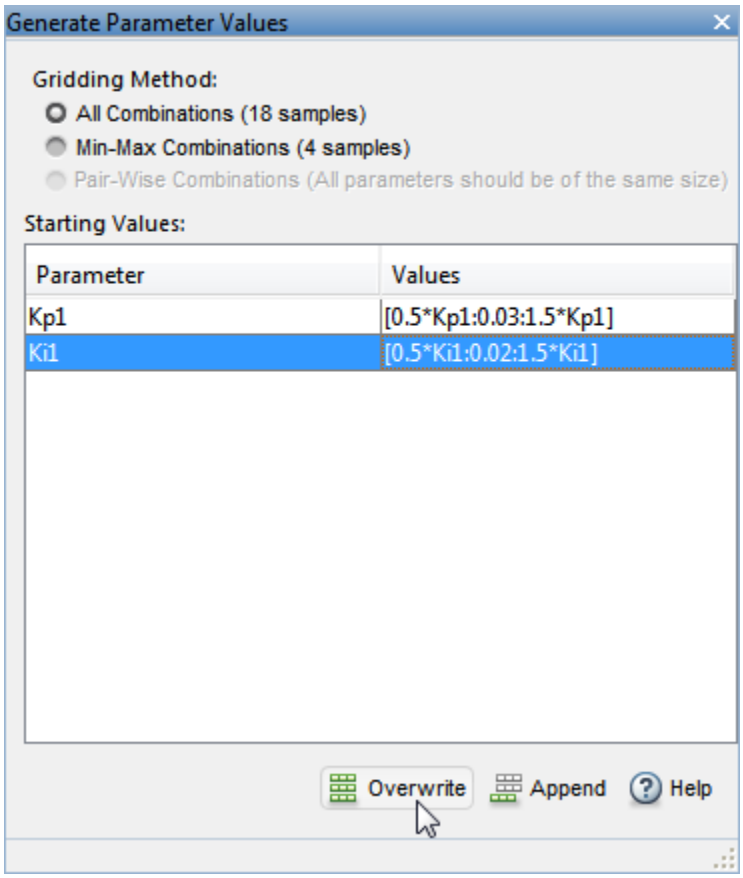
Vary the Outer-Loop Controller Gains

Examine the overall performance of the cascaded control system for varying values of the outer-loop controller, `C1`. To do so, vary the coefficients `Ki1` and `Kp1`, while keeping `Ki2` and `Kp2` fixed at the values specified in the model.

In the **Parameter Variations** tab, click **Manage Parameters**. Clear the `Ki2` and `Kp2` checkboxes, and check `Ki1` and `Kp1`. Click **OK**.



Use **Model Linearizer** to generate parameter values automatically. Click **Generate Values**. In the **Values** column of the Generate Parameter Values table, enter an expression specifying the possible values for each parameter. For example, vary `Kp1` and `Ki1` by ± 50% of their nominal values, by entering expressions as shown.

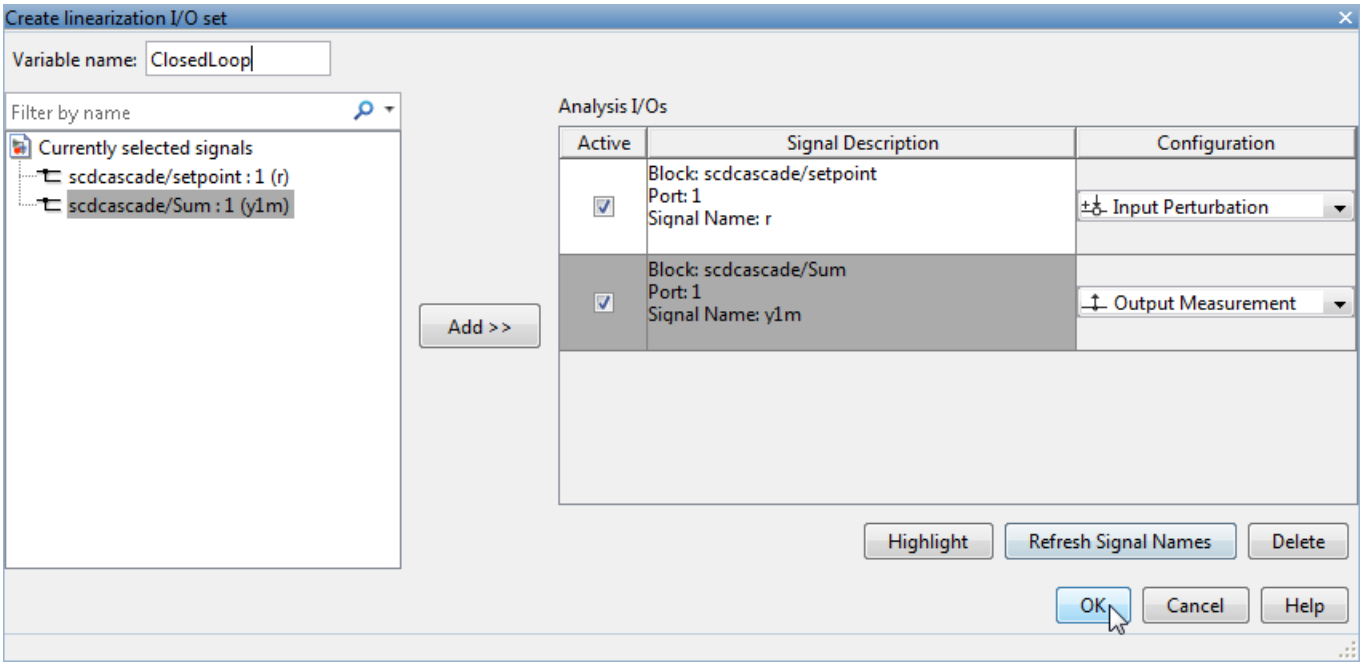


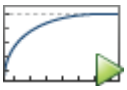
The **All Combinations** gridding method generates a complete parameter grid of `(Kp1,Ki1)` pairs, to compute a linearization at all possible combinations of the specified values. Click **Overwrite** to replace all values in the Parameter Variations table with the generated values.

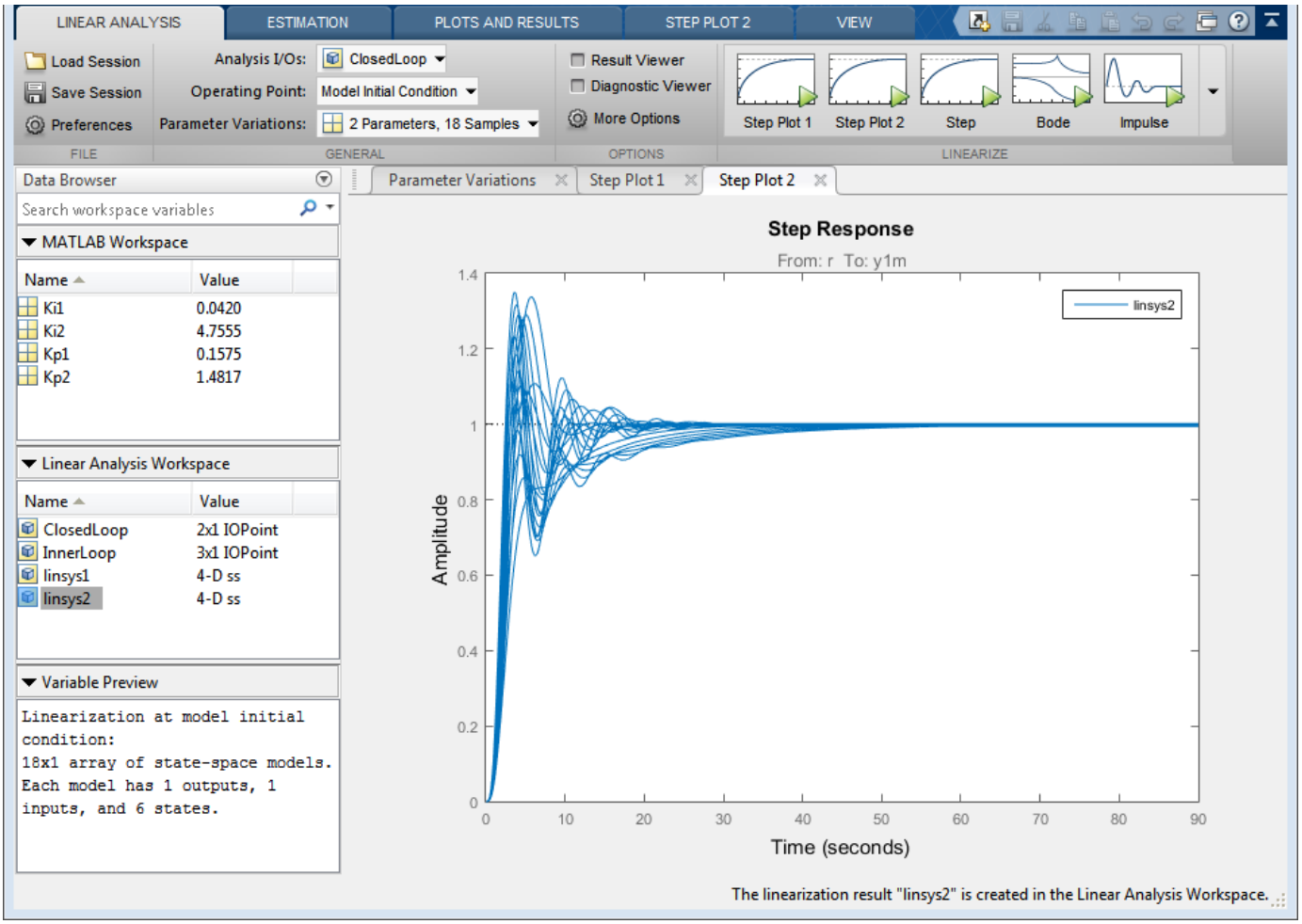
Parameter Variations | Step Plot 1

Kp1	Ki1
0.0788	0.0210
0.1088	0.0210
0.1387	0.0210
0.1688	0.0210
0.1988	0.0210
0.2288	0.0210
0.0788	0.0410
0.1088	0.0410
0.1387	0.0410
0.1688	0.0410
0.1988	0.0410
0.2288	0.0410
0.0788	0.0610
0.1088	0.0610
0.1387	0.0610
0.1688	0.0610
0.1988	0.0610
0.2288	0.0610

Because you want to examine the overall closed-loop transfer function of the system, create a new linearization I/O set. In the **Linear Analysis** tab, in the **Analysis I/Os** drop-down list, select `Create New Linearization I/Os`. Configure `r` as an input perturbation point, and the system output `y1m` as an output measurement. Click **OK**.



Linearize the model with the parameter variations and examine the step response of the resulting models. Click  **Step** to linearize and generate a new plot for the new model array, `linsys2`.



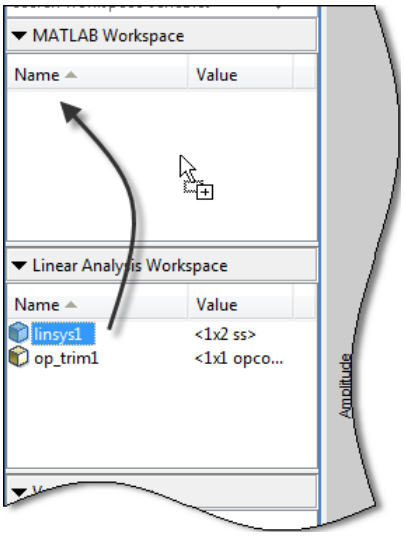
The step plot shows the responses of every model in the array. This plot gives you a sense of the range of step responses of the system in the operating ranges covered by the parameter grid.

Note Although the new plot reflects the new set of parameter variations, `Step Plot 1` and `linsys1` are unchanged. That plot and array still reflect the linearizations obtained with the inner-loop parameter variations.

Further Analysis of Batch Linearization Results

The results of both batch linearizations, `linsys1` and `linsys2`, are arrays of state-space (`ss`) models. Use these arrays for further analysis in any of several ways:

- Create additional analysis plots, such as Bode plots or impulse response plots, as described in “Analyze Results Using Model Linearizer Response Plots” on page 2-113.
- Examine individual responses in analysis plots as described in “Analyze Batch Linearization Results in Model Linearizer” on page 3-39.
- Drag the array from Linear Analysis Workspace to the MATLAB workspace.



You can then use Control System Toolbox control design tools, such as the Linear System Analyzer app, to analyze linearization results. Or, use Control System Toolbox control design tools, such as `pidtune` or **Control System Designer**, to design controllers for the linearized systems.

Also see “Validate Batch Linearization Results” on page 3-69 for information about validating linearization results in the MATLAB workspace.

See Also

More About

- “Validate Batch Linearization Results” on page 3-69
- “Batch Linearization Efficiency When You Vary Parameter Values” on page 3-7
- “Specify Parameter Samples for Batch Linearization” on page 3-43
- “Analyze Batch Linearization Results in Model Linearizer” on page 3-39
- “Batch Linearize Model for Parameter Variations at Single Operating Point” on page 3-13

More Efficient Batch Linearization Varying Parameters

This example shows how to speed up the batch linearization of a model when a set of model parameters are varied.

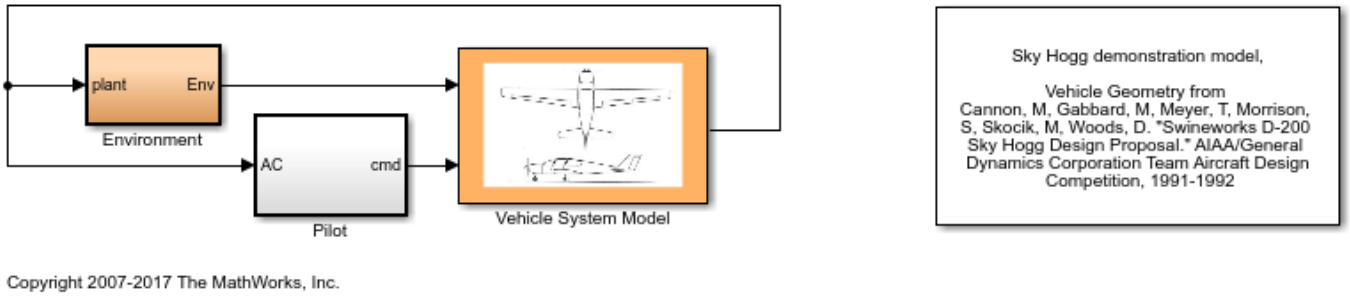
To decrease the linearization time for such a model, you can pass the varying parameter values to the `linearize` function. `linearize` avoids recompiling the model when the varied parameters are tunable parameters. The best improvements in the overall linearization time are for models with large model update times.

Plant Model

In this example, you linearize a lightweight airplane model. For more information on this model, see “Lightweight Airplane Design” (Aerospace Blockset). Using this model requires Aerospace Blockset™ software.

Open the model.

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



Extract the linearization inputs and outputs from the model.

```
io = getlinio(mdl);
```

Extract the initial operating point of the model.

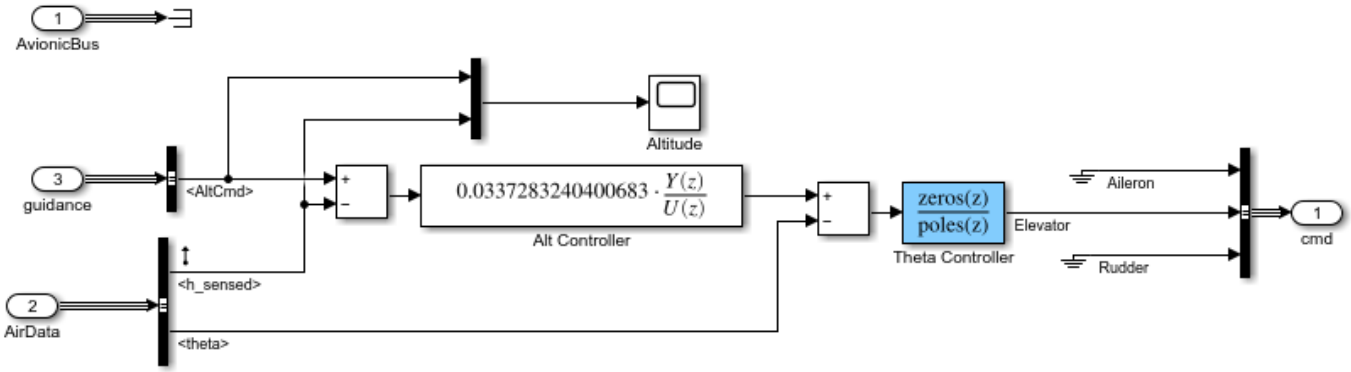
```
op = operpoint(mdl);
```

Linearize Model By Calling `linearize` Multiple Times

For this example, you vary the gains of the altitude and pitch controllers in the autopilot system by +/- 10%.

Open the auto pilot subsystem.

```
open_system('scdskyhogg/Vehicle System Model/Avionics/Autopilot')
```



Initialize the gains of the controllers to vary with MATLAB® workspace variables `k1` and `k2`.

```
blks = {'scdskyhogg/Vehicle System Model/Avionics/Autopilot/Alt Controller';...
        'scdskyhogg/Vehicle System Model/Avionics/Autopilot/Theta Controller'};
set_param(blks{1},'Gain','0.0337283240400683*k1')
set_param(blks{2},'Gain','-261.8699347622*k2')
```

Specify the varying values for `k1` and `k2`.

```
ct = 1:20;
k1val = 1+(ct-10)/100;
k2val = 1+(ct-10)/100;
```

Linearize the model for the specified values of `k1` and `k2` by calling the `linearize` function 20 times.

```
t = cputime;
for i=1:20
    k1 = k1val(i);
    k2 = k2val(i);
    sys_forloop(:,:,i) = linearize(mdl,op,io);
end
```

View the total time in seconds to compute the 20 linearizations.

```
dt_for = cputime - t
```

```
dt_for =

   57.0938
```

A factor that impacts this time is the total time it takes to compile and evaluate block masks and resolve workspace parameters. To identify bottlenecks in your model compilation, use the MATLAB Profiler.

Linearize Model By Passing Parameter Values to `linearize`

Instead of calling `linearize` separately for each combination of parameter values, you can pass the parameter values to `linearize` in a single function call.

Define the parameter values in a structure by specifying the name of the MATLAB workspace variables and the value arrays.

```
params(1).Name = 'k1';
params(1).Value = k1val;
params(2).Name = 'k2';
params(2).Value = k2val;
```

Linearize the model.

```
t = cputime;
sys_params = linearize(mdl,op,io,params);
```

View the total time to compute the 20 linearizations with one call to the `linearize` function. In this case, the software compiles the model once when varying the specified parameters.

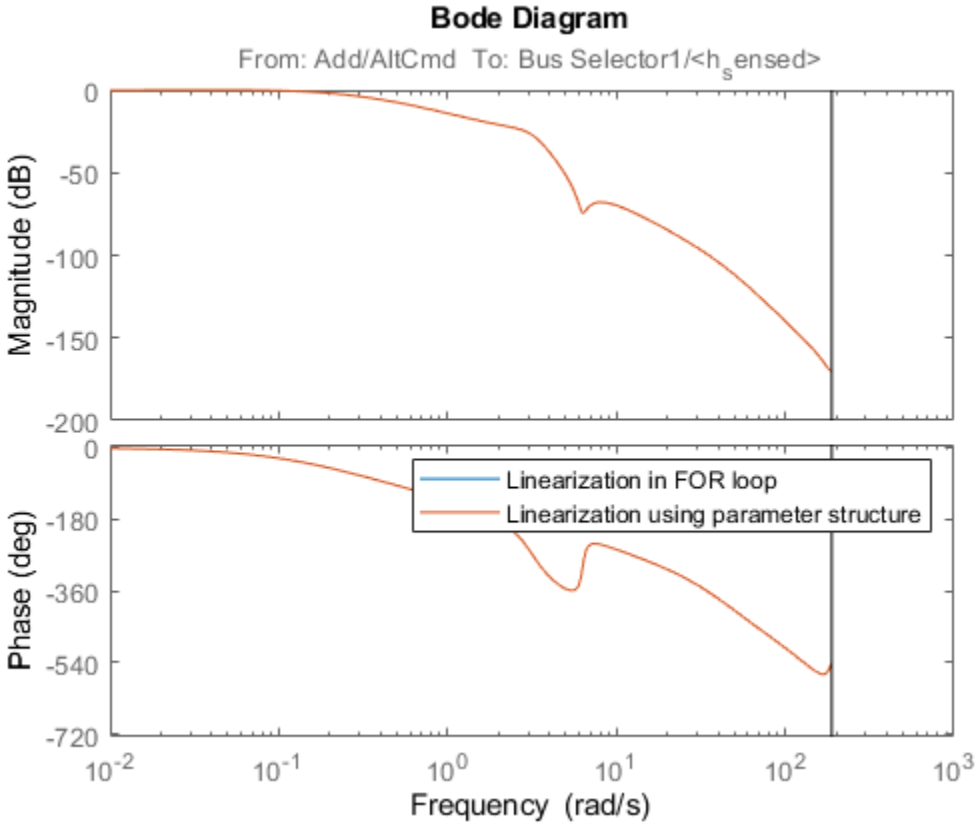
```
dt_params = cputime - t
```

```
dt_params =

   12.0469
```

Compare Results

In this example, the varying parameters do not impact the operating point of the Simulink model. The linearizations using both approaches are equivalent.

```
bode(sys_forloop(:,:,1),sys_params(:,:,1))
legend('Linearization in FOR loop','Linearization using parameter structure')
```



Calculate the resulting time improvement ratio.

```
ratio = dt_for/dt_params
```

```
ratio =

    4.7393
```

Close the model.

```
bdclose(mdl)
```

See Also

`linearize` | `slLinearizer` | `fastRestartForLinearAnalysis`

Related Examples

- “What Is Batch Linearization?” on page 3-2
- “Choose Batch Linearization Methods” on page 3-4
- “Improve Linear Analysis Performance” on page 3-97

Validate Batch Linearization Results

When you batch linearize a model, the software returns a model array containing the linearized models. There are two ways to validate a linearized model, but both methods have some computational overhead. This overhead can make validating each model in the batch linearization results infeasible. Therefore, it can be cost effective to validate either a single model or a subset of the batch linearization results. You can use linear analysis plots and commands to determine the validation candidates. For information regarding the tools that you can use for such analysis, see "Linear Analysis".

You can validate a linearization using the following approaches:

- Obtain a frequency response estimation of the nonlinear model, and compare its response to that of the linearized model. For an example, see "Validate Linearization In Frequency Domain Using Model Linearizer" on page 2-108.
- Simulate the nonlinear model and compare its time-domain response to that of the linearized model. For an example, see "Validate Linearization In Time Domain" on page 2-107.

See Also

`linearize` | `slLinearizer`

Related Examples

- "Analyze Batch Linearization Results in Model Linearizer" on page 3-39
- "Analyze Command-Line Batch Linearization Results Using Response Plots" on page 3-33

Approximate Nonlinear Behavior Using Array of LTI Systems

This example shows how to approximate the nonlinear behavior of a system as an array of interconnected LTI models.

The example describes linear approximation of pitch axis dynamics of an airframe over a range of operating conditions. The resulting array of linear systems is used to create a linear parameter varying (LPV) representation of the dynamics. The LPV model serves as an approximation of the nonlinear pitch dynamics.

Linear Parameter Varying Models

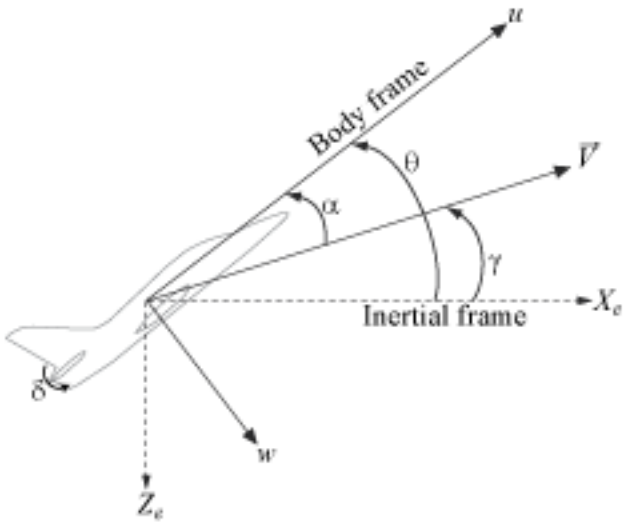
In many situations you must approximate the nonlinear dynamics of a system using simpler linear systems. A single linear system provides a reasonable model for behavior limited to a small neighborhood around an operating point of the nonlinear system. When you must approximate the nonlinear behavior needs over a range of operating conditions, you can use an array of linear models that are interconnected by suitable interpolation rules. Such a model is called an LPV model.

To generate an LPV model, the nonlinear model is trimmed and linearized over a grid of operating points. For this purpose, the operating space is parameterized by a small number of *scheduling parameters*. These parameters are often a subset of the inputs, states, and output variables of the nonlinear system. Important considerations in the creation of LPV models are the identification of a scheduling parameter set and the selection of a range of parameter values at which to linearize the model.

To illustrate this approach, this example approximates the pitch dynamics of an airframe.

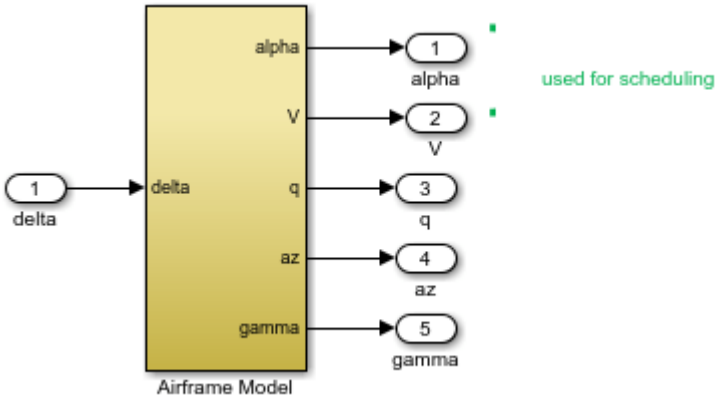
Airframe Pitch Dynamics

Consider a three-degree-of-freedom model of the pitch axis dynamics of an airframe. The states are the Earth coordinates (X_e, Z_e), the body coordinates (u, w), the pitch angle θ, and the pitch rate $q = \dot{\theta}$. The following figure summarizes the relationship between the inertial and body frames, the flight path angle γ, the incidence angle α, and the pitch angle θ.



The airframe dynamics are nonlinear and the aerodynamic forces and moments depend on speed V and incidence α. The `scdairframeTRIM` model `scdairframeTRIM` describes these dynamics.

```
open_system('scdairframeTRIM')
```



Batch Linearization Across the Flight Envelope

Use the speed V and the incidence angle α as scheduling parameters; that is, trim the airframe model over a grid of α and V values. These values are two of the five outputs of the `scdairframeTRIM` model.

Assume that the incidence α varies between -20 and 20 degrees and that the speed V varies between 700 and 1400 m/s. Use a 15-by-12 grid of linearly spaced (α, V) pairs for scheduling.

```
nA = 15;   % number of alpha values
nV = 12;   % number of V values
alphaRange = linspace(-20,20,nA)*pi/180;
```

```
VRange = linspace(700,1400,nV);
[alpha,V] = ndgrid(alphaRange,VRange);
```

For each flight condition (α, V), linearize the airframe dynamics at trim (zero normal acceleration and pitching moment). Doing so requires computing the elevator deflection δ and pitch rate q that result in steady w and q.

For each flight condition, you:

- Specify the trim condition using the `operspec` function.
- Compute the trim values of δ and q using the `findop` function.
- Linearize the airframe dynamics for the resulting operating point using the `linearize` function.

The body coordinates, (u, w), are known states for trimming. Therefore, you must provide appropriate values for them, which you can specify explicitly. However, in this example, let the model derive these known values based on each (α, V) pair. For each flight condition (α, V), update the values in the model and create an operating point specification. Repeat these steps for all 180 flight conditions.

```
clear op report
for ct = 1:nA*nV
   alpha_ini = alpha(ct);      % Incidence [rad]
   v_ini = V(ct);              % Speed [m/s]

   % Specify trim condition
   opspec(ct) = operspec('scdairframeTRIM');

   % Xe,Ze: known, not steady.
   opspec(ct).States(1).Known = [1;1];
   opspec(ct).States(1).SteadyState = [0;0];

   % u,w: known, w steady
   opspec(ct).States(3).Known = [1 1];
   opspec(ct).States(3).SteadyState = [0 1];

   % theta: known, not steady
   opspec(ct).States(2).Known = 1;
   opspec(ct).States(2).SteadyState = 0;

   % q: unknown, steady
   opspec(ct).States(4).Known = 0;
   opspec(ct).States(4).SteadyState = 1;

end
opspec = reshape(opspec,[nA nV]);
```

Trim the model for all of the operating point specifications.

```
opt = findopOptions('DisplayReport','off', 'OptimizerType','lsqnonlin');
opt.OptimizationOptions.Algorithm = 'trust-region-reflective';
[op,report] = findop('scdairframeTRIM',opspec,opt);
```

The `op` array contains the operating points found by `findop` that will be used for linearization. The `report` array contains a record of input, output, and state values at each point.

To linearize the model, first specify linearization input and output points.

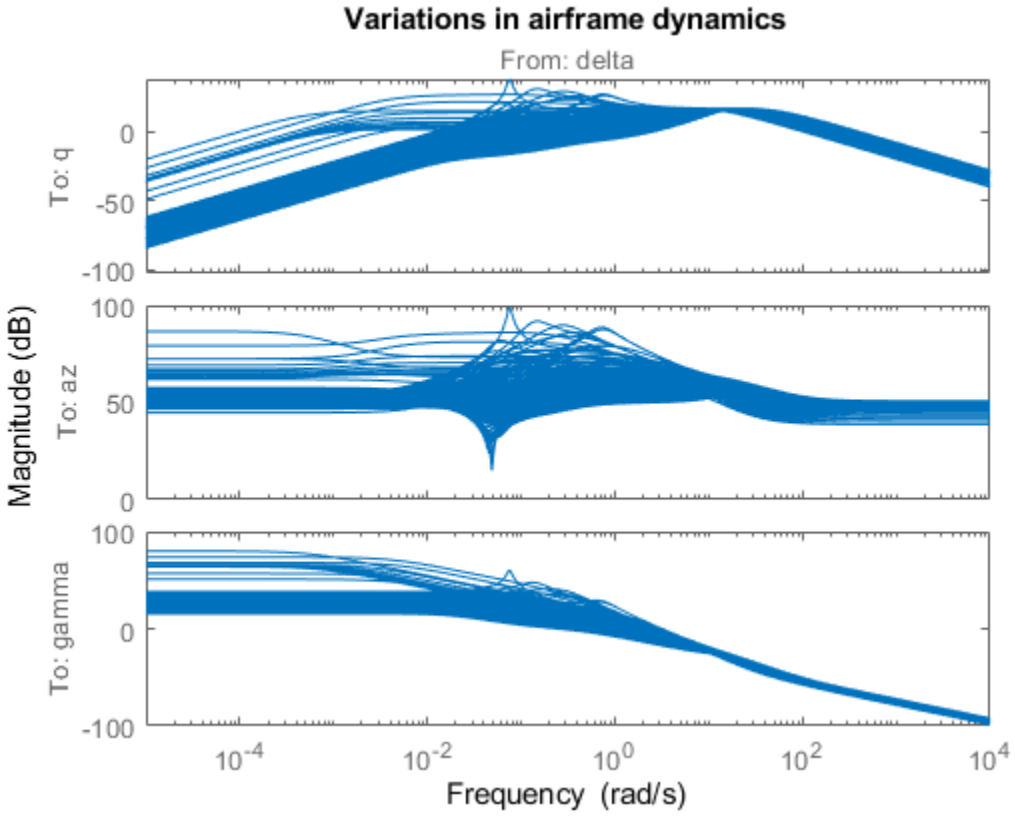
```
io = [linio('scdairframeTRIM/delta',1,'in');            % delta
      linio('scdairframeTRIM/Airframe Model',1,'out');  % alpha
      linio('scdairframeTRIM/Airframe Model',2,'out');  % V
      linio('scdairframeTRIM/Airframe Model',3,'out');  % q
      linio('scdairframeTRIM/Airframe Model',4,'out');  % az
      linio('scdairframeTRIM/Airframe Model',5,'out')]; % gamma
```

Linearize the model for each of the trim conditions. Store linearization offset information in the `info` structure.

```
linOpt = linearizeOptions('StoreOffsets',true);
[G,~,info] = linearize('scdairframeTRIM',op,io,linOpt);
G = reshape(G,[nA nV]);
G.u = 'delta';
G.y = {'alpha','V','q','az','gamma'};
G.SamplingGrid = struct('alpha',alpha,'V',V);
```

`G` is a 15-by-12 array of linearized plant models at the 180 flight conditions (α, V). The plant dynamics vary substantially across the flight envelope, including scheduling locations where the local dynamics are unstable.

```
bodemag(G(3:5,:,:,:))
title('Variations in airframe dynamics')
```



LPV System Block

The LPV System block facilitates simulation of linear parameter varying systems. The block requires the state-space system array `G` that you generated using batch linearization. You also augment this information with the input, output, state, and state derivative offsets from the `info` structure.

Extract the offset information.

```
offsets = getOffsetsForLPV(info);
xOffset = offsets.x;
yOffset = offsets.y;
uOffset = offsets.u;
dxOffset = offsets.dx;
```

LPV Model Simulation

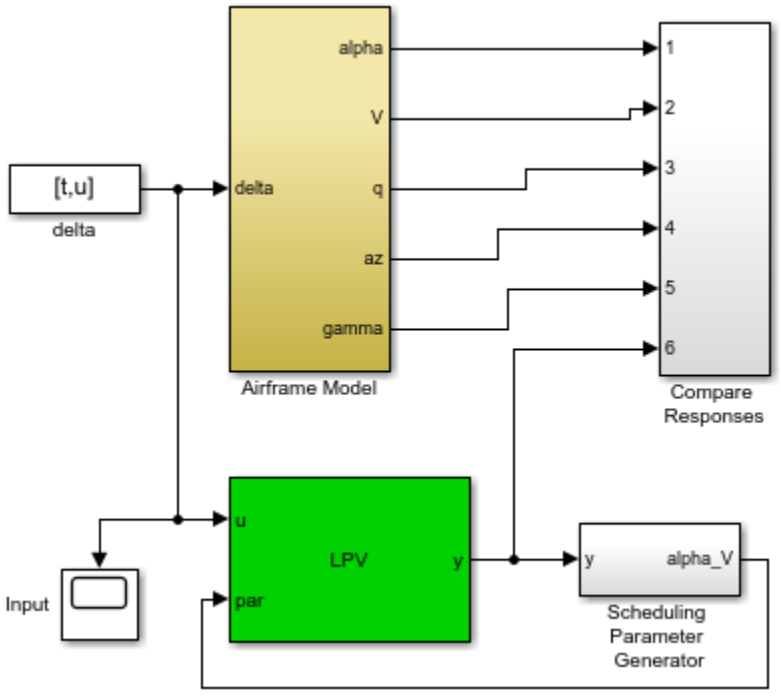
The `scdairframeLPV` model, which contains an LPV System block that uses the linear system array `G` and the corresponding offsets.

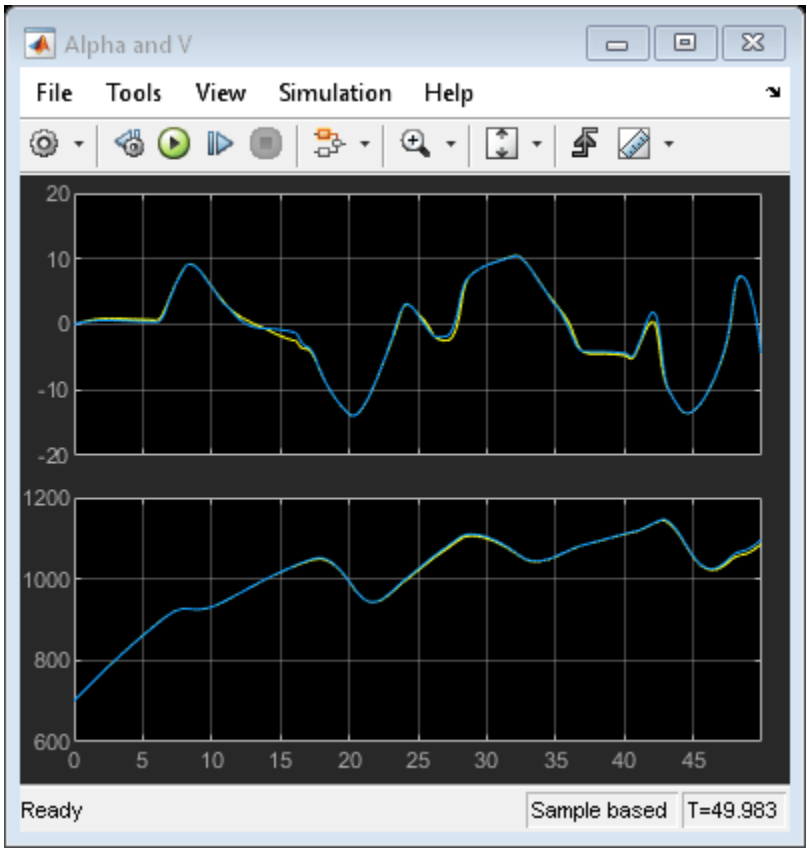
This model uses an input signal based on a desired trajectory of the airframe. This signal `u` and corresponding time vector `t` are saved in the `scdairframeLPVsimdata.mat` file, which is loaded by the model. Specify the initial conditions for simulation.

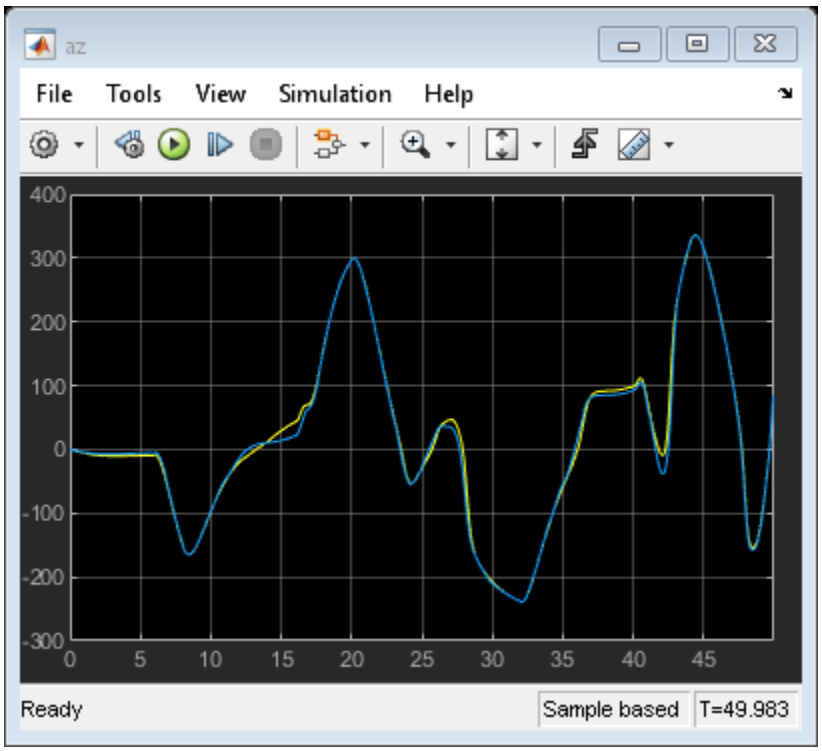
```
alpha_ini = 0;
v_ini = 700;
x0 = [0; 700; 0; 0];
```

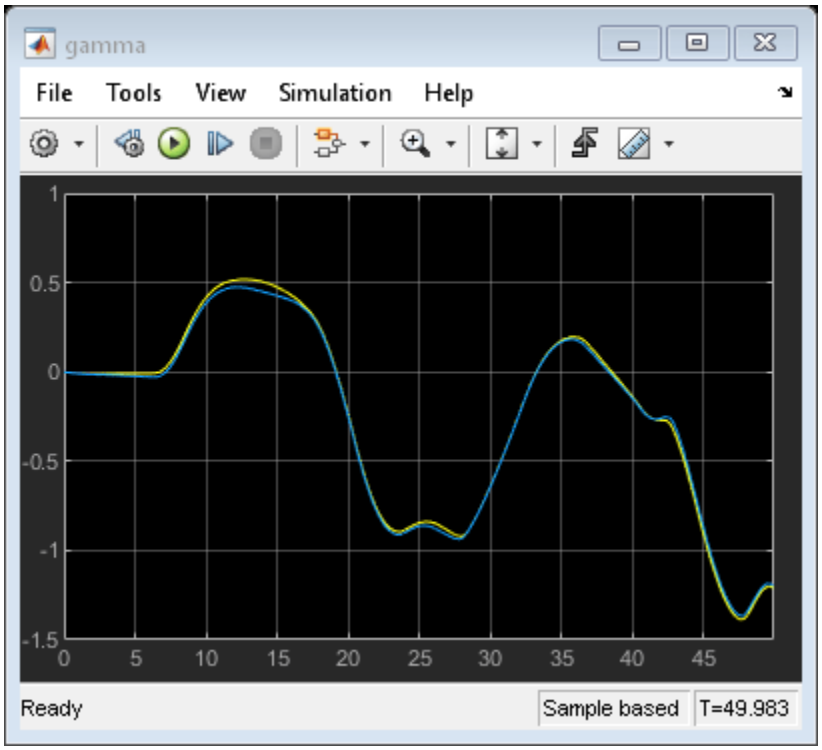
Open and simulate the model.

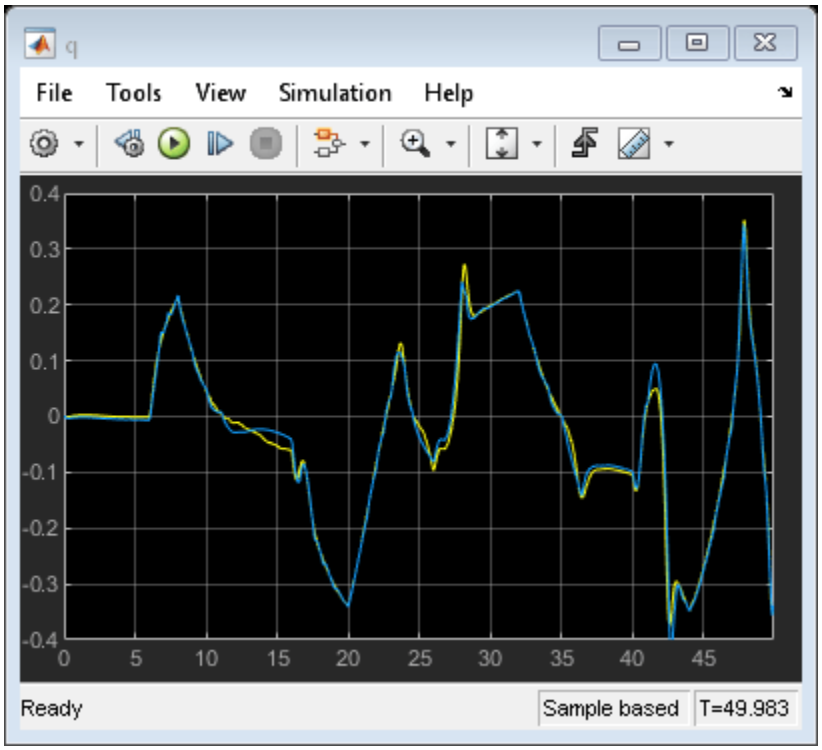
```
open_system('scdairframeLPV')
sim('scdairframeLPV')
```

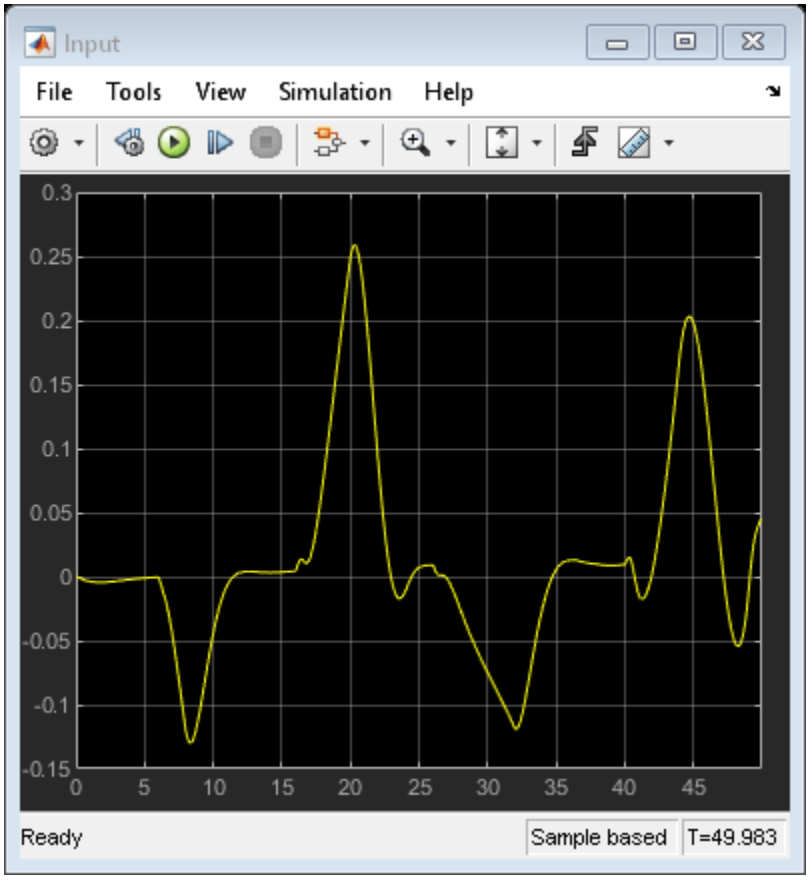



Alpha and V
File Tools View Simulation Help
20
10
0
-10
-20
1200
1000
800
600
0 5 10 15 20 25 30 35 40 45
Ready
Sample based
T=49.983


az
File Tools View Simulation Help
400
300
200
100
0
-100
-200
-300
0 5 10 15 20 25 30 35 40 45
Ready
Sample based
T=49.983


gamma
File Tools View Simulation Help
1
0.5
0
-0.5
-1
-1.5
0 5 10 15 20 25 30 35 40 45
Ready
Sample based
T=49.983


q
File Tools View Simulation Help
0.4
0.3
0.2
0.1
0
-0.1
-0.2
-0.3
-0.4
0 5 10 15 20 25 30 35 40 45
Ready
Sample based
T=49.983



The simulation shows good emulation of the airframe response by the LPV system. For this simulation you specified a fine gridding of the scheduling space leading to a large number (180) of linear models. Large array sizes can increase implementation costs. However, the advantage of LPV representations is that you can adjust the scheduling grid, and hence the number of linear systems in the array, based on:

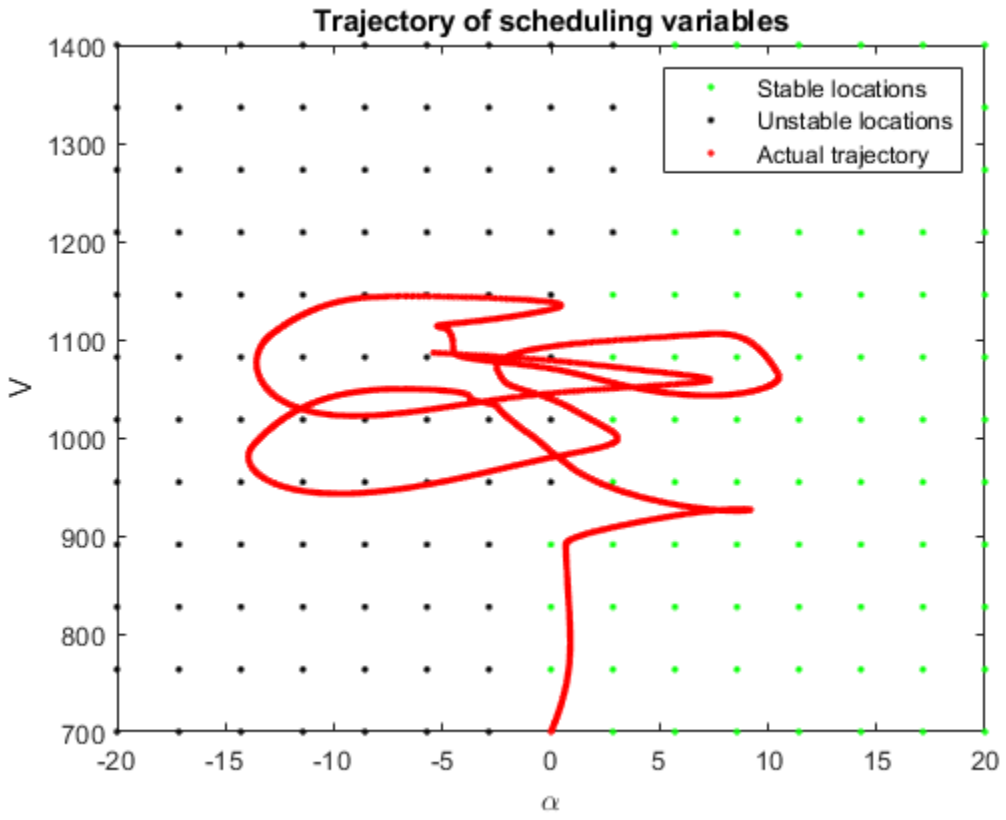
- The scheduling subspace spanned by the anticipated trajectory
- The level of accuracy desired in an application

The former information helps reduce the range for the scheduling variables. The latter helps pick an optimal resolution (spacing) of samples in the scheduling space.

Plot the actual trajectory of scheduling variables in the previous simulation against the backdrop of the gridded scheduling space. The (α, V) outputs were logged using their corresponding scopes (contained inside the Compare Responses block of `scdairframeLPV`).

```
Stable = false(nA,nV);
for ct = 1:nA*nV
   Stable(ct ) = isstable(G(:,:,ct));
end
alpha_trajectory = Alpha_V_Data.signals(1).values(:,1);
V_trajectory = Alpha_V_Data.signals(2).values(:,1);

plot(alpha(Stable)*180/pi,V(Stable),'g.',...
   alpha(~Stable)*180/pi,V(~Stable),'k.',...
```

```
   alpha_trajectory,V_trajectory,'r.')
title('Trajectory of scheduling variables')
xlabel('\alpha')
ylabel('V')
legend('Stable locations','Unstable locations','Actual trajectory')
```



The trajectory traced during simulation is shown in red. It traverses both the stable and unstable regions of the scheduling space.

Suppose that you want to implement this model on target hardware for input profiles similar to the one used for the previous simulation, while using the least amount of memory. The simulation suggests that the trajectory stays mainly in the 890 to 1200 m/s range of velocities and -15 to 12 degree range of incidence angle. Find the indices in the scheduling space that correspond to this operating region.

```
I1 = find(alphaRange>=-15*pi/180 & alphaRange<=12*pi/180);
I2 = find(VRange>=890 & VRange<=1200);
```

To reduce the number of flight conditions, you can also increase the spacing between the sampling points. For example, extract the indices for every third sample along the V dimension and every second sample along the α dimension.

```
I1 = I1(1:2:end);
I2 = I2(1:3:end);
```

Extract the subset of the LTI system array.

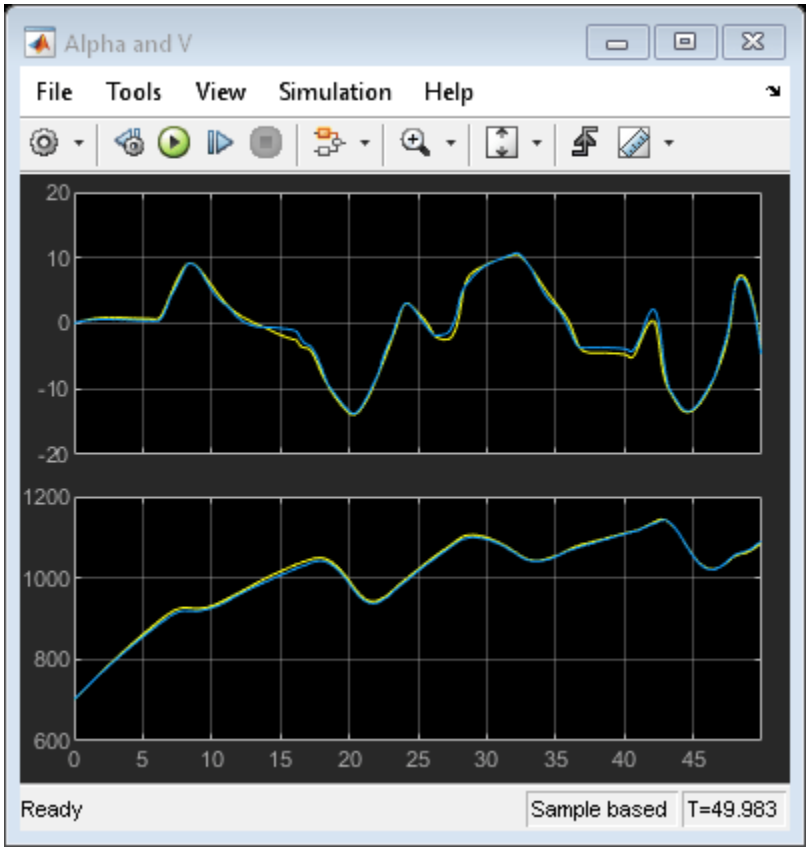
```
Gr = G(:,:,I1,I2);
size(Gr)
```

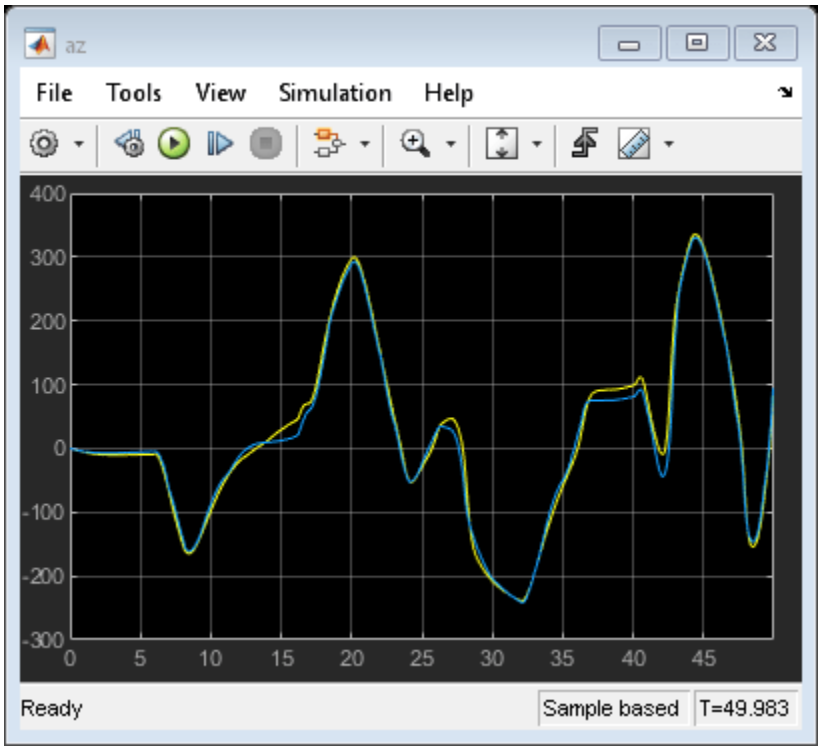
```
5x2 array of state-space models.
Each model has 5 outputs, 1 inputs, and 4 states.
```

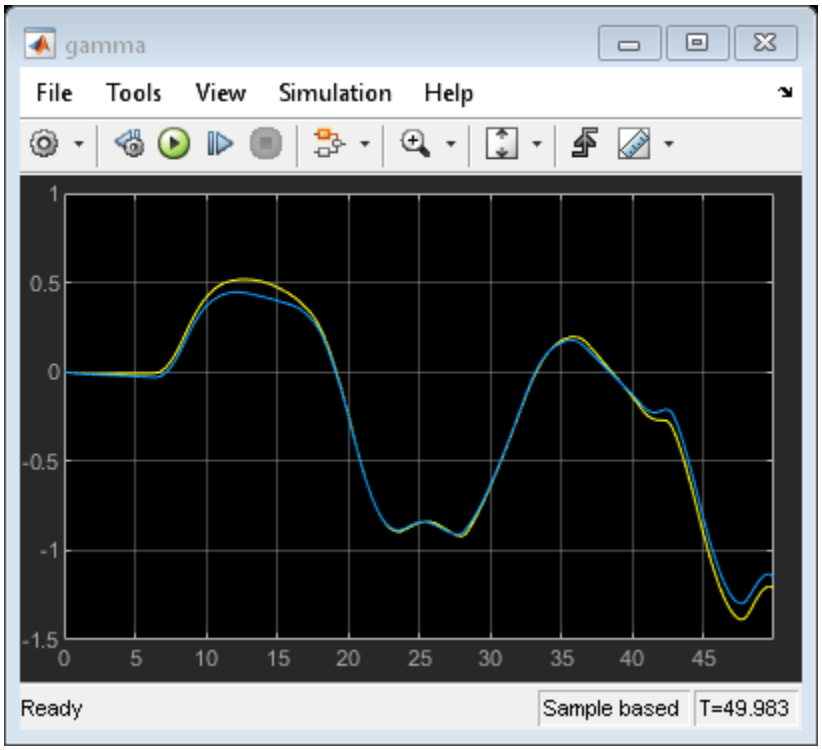
The new sampling grid, `Gr`, has a more economical size of 5-by-2. Simulate the reduced model and check its fidelity in reproducing the original behavior.

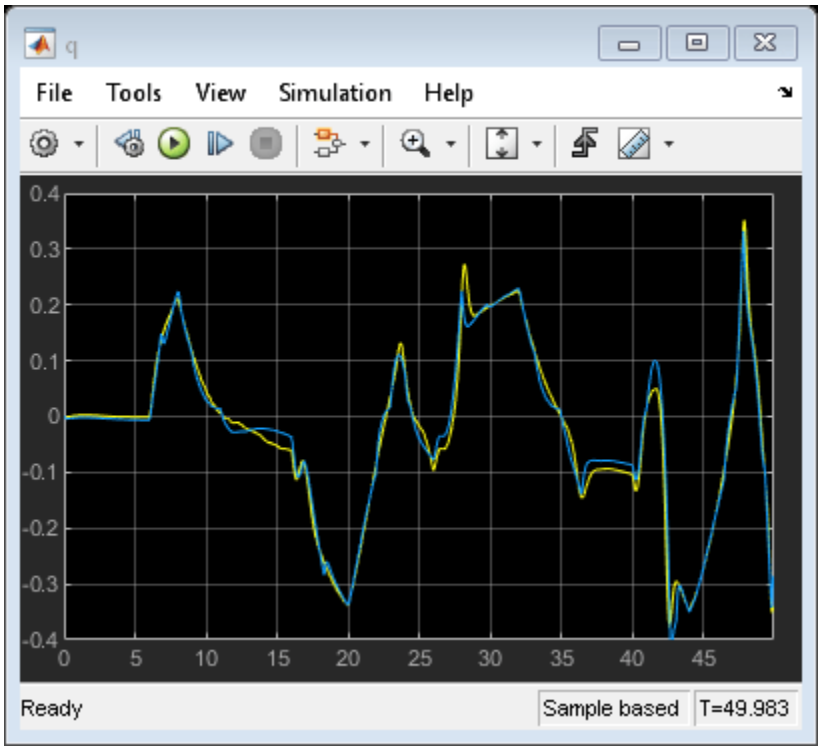
Configure the LPV System block to use the reduced model and corresponding offsets.

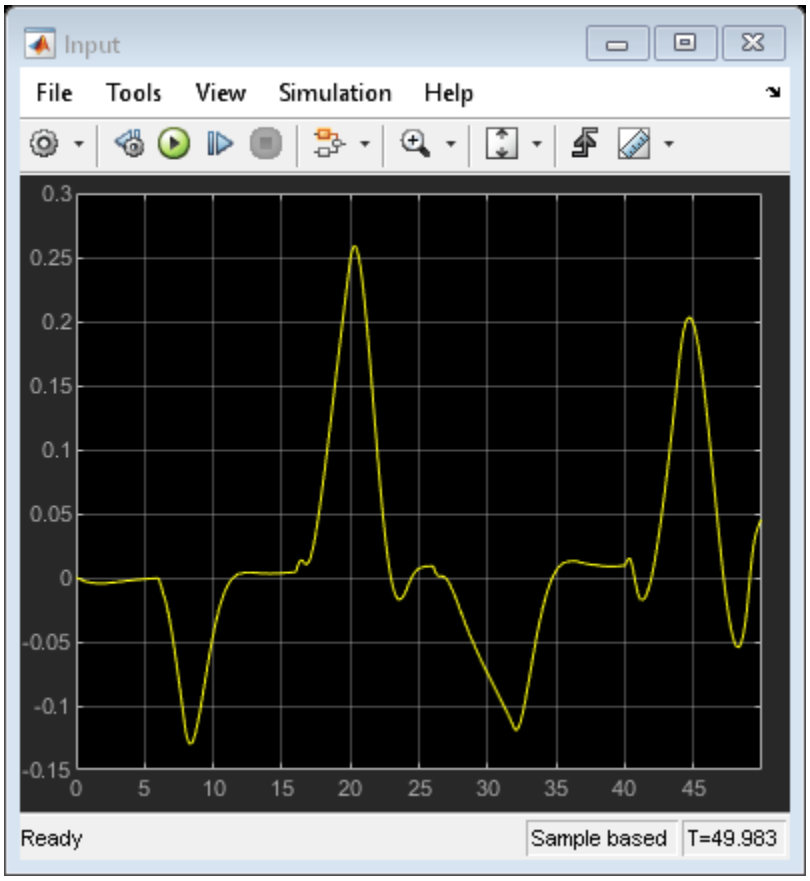
```
lpvblk = 'scdairframeLPV/LPV System';
set_param(lpvblk,...
   'sys','Gr',...
   'uOffset','uOffset(:,:,I1,I2)',...
   'yOffset','yOffset(:,:,I1,I2)',...
   'xOffset','xOffset(:,:,I1,I2)',...
   'dxOffset','dxOffset(:,:,I1,I2)')
sim('scdairframeLPV')
```




az
File Tools View Simulation Help
400
300
200
100
0
-100
-200
-300
0 5 10 15 20 25 30 35 40 45
Ready
Sample based
T=49.983


gamma
File Tools View Simulation Help
1
0.5
0
-0.5
-1
-1.5
0 5 10 15 20 25 30 35 40 45
Ready
Sample based
T=49.983


q
File Tools View Simulation Help
0.4
0.3
0.2
0.1
0
-0.1
-0.2
-0.3
-0.4
0 5 10 15 20 25 30 35 40 45
Ready
Sample based
T=49.983


Input
File Tools View Simulation Help
0.3
0.25
0.2
0.15
0.1
0.05
0
-0.05
-0.1
-0.15
0 5 10 15 20 25 30 35 40 45
Ready
Sample based
T=49.983

There is no significant reduction in overlap between the response of the original model and its LPV proxy.

The LPV model can serve as a proxy for the original system in situations where faster simulations are required. The linear systems used by the LPV model can also be obtained using system identification techniques (with additional care required to maintain state consistency across the array). The LPV model can provide a good surrogate for initializing Simulink® Design Optimization™ problems and performing fast hardware-in-loop simulations.

```
bdclose('scdairframeLPV')
bdclose('scdairframeTRIM')
```

See Also

LPV System | `linearize`

Related Examples

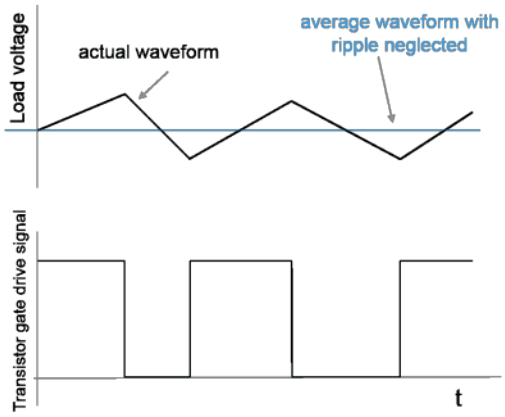
- “Batch Linearize Model at Multiple Operating Points Using linearize Command” on page 3-19
- “LPV Approximation of Boost Converter Model” on page 3-83

LPV Approximation of Boost Converter Model

This example shows how to obtain a linear parameter varying (LPV) approximation of a Simscape™ Electrical™ model of a boost converter. The LPV representation allows quick analysis of average behavior at various operating conditions.

Boost Converter Model

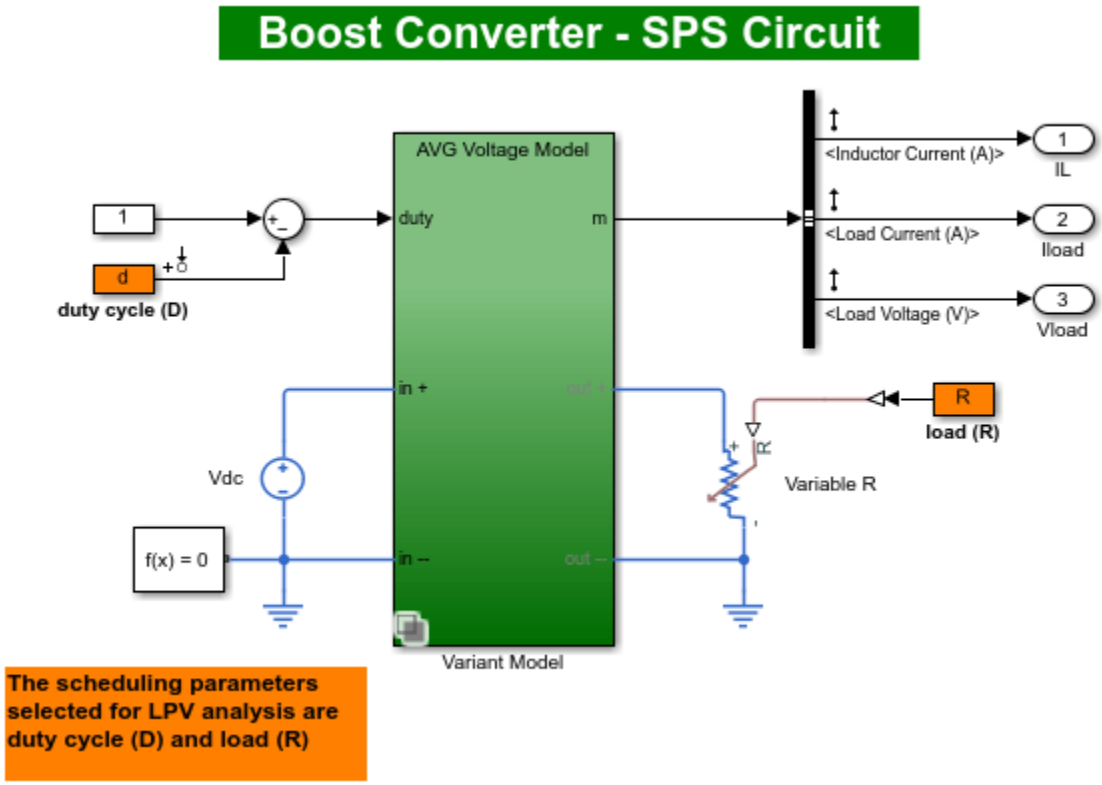
A Boost Converter circuit converts a DC voltage to another DC voltage by controlled chopping or switching of the source voltage. The request for a certain load voltage is translated into a corresponding requirement for the transistor duty cycle. The duty cycle modulation is typically several orders of magnitude slower than the switching frequency, which produces an average voltage with relatively small ripples, as shown in the following figure.



In practice, there are also disturbances in the source voltage V_{dc} and the resistive load R affecting the actual load voltage V_{load}.

Open the Simulink® model.

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



The circuit in the model is characterized by high-frequency switching. The model uses a sample time of 25 ns. The Boost Converter block used in the model is a variant subsystem that implements two different versions of the converter dynamics. The model takes the duty cycle value as its only input and produces three outputs: inductor current, load current, and load voltage.

Due to the high-frequency switching and short sample time, the model simulates slowly.

Batch Trimming and Linearization

In many applications, the average voltage delivered in response to a certain duty cycle profile is of interest. Such behavior is studied at time scales several decades larger than the fundamental sample time of the circuit. These *average models* for the circuit are derived by analytical considerations based on averaging of power dynamics over certain time periods. The `BoostConverterExampleModel` model implements such an average model of the circuit as its first variant, called AVG Voltage Model. This variant typically executes faster than the Low Level Model variant.

The average model is not a linear system. It shows nonlinear dependence on the duty cycle and the load variations. To produce faster simulation and to help with voltage stabilizing controller design, you can linearize the model at various duty cycle and load values.

For this example, use the snapshot-based trimming and linearization. The scheduling parameters are the duty cycle `d` and resistive load `R`. You trim and linearize the model for several values of the scheduling parameters.

For this example, select a span of 10-60% for the duty cycle variation and a span of 4-15 ohms for the load variation. Select five values in these ranges for each scheduling variable and linearization obtained at all possible combinations of their values.

```
nD = 5;
nR = 5;
dspace = linspace(0.1,0.6,nD); % Values of d in 10%-60% range
Rspace = linspace(4,15,nR);    % Values of Rin 4-15 Ohms range
[dgrid,Rgrid] = ndgrid(dspace,Rspace);  % All combinations of d and R values
```

Create a parameter structure array for the scheduling parameters.

```
params(1).Name = 'd';
params(1).Value = dgrid;
params(2).Name = 'R';
params(2).Value = Rgrid;
```

Specify the number of model inputs, outputs, and states.

```
ny = 3;
nu = 1;
nx = 2;
ArraySize = size(dgrid);
```

A simulation of the model under various conditions shows that the model outputs settle down to their steady-state values before 0.01 s. Therefore, use t = 0.01s as the snapshot time.

Compute equilibrium operating points at the snapshot time using the `findop` function. This operation takes several minutes to finish.

```
op = findop(mdl,0.01,params);
```

To linearize the model, first obtain the linearization input and output points from the model.

```
io = getlinio(mdl);
```

Configure the linearization options to store linearization offsets.

```
opt = linearizeOptions('StoreOffsets', true);
```

Linearize the model at the operating points in array `op`.

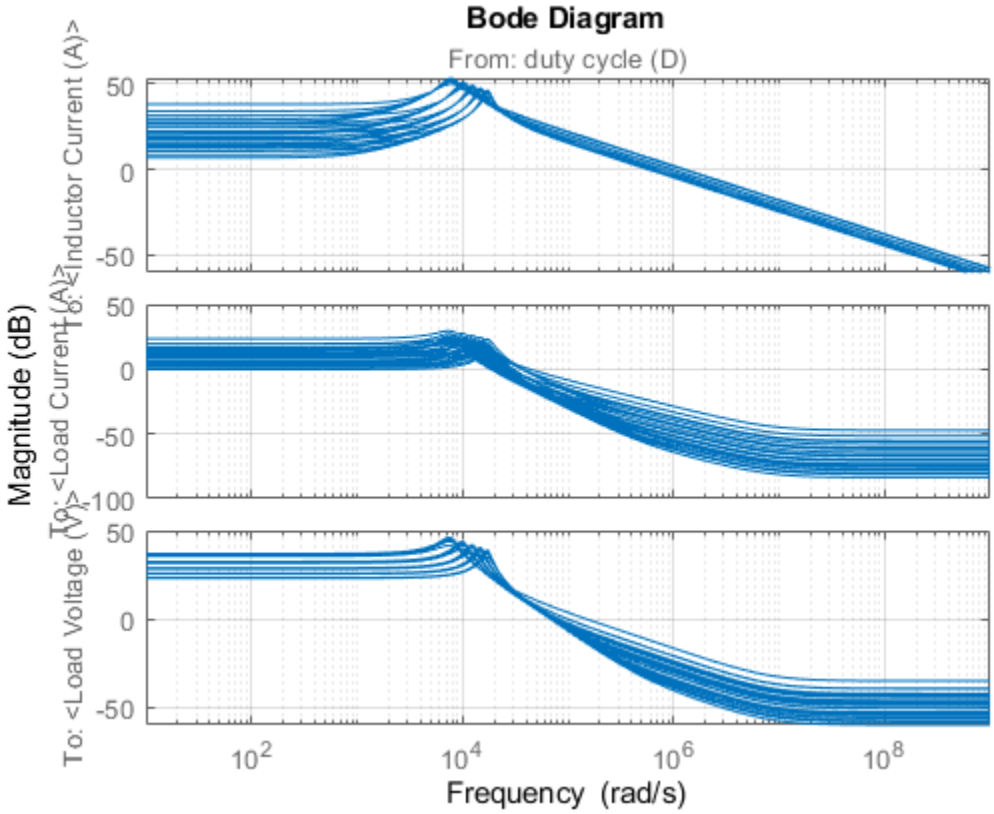
```
[linsys,~,info] = linearize(mdl,op,io,params,opt);
```

Extract offsets from the linearization results.

```
offsets = getOffsetsForLPV(info);
yoff = offsets.y;
xoff = offsets.x;
uoff = offsets.u;
```

Plot the linear system array.

```
bodemag(linsys)
grid on
```

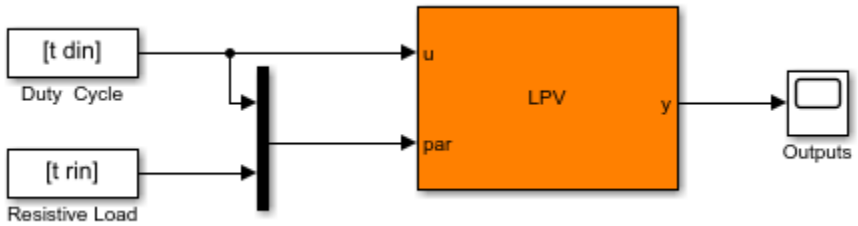


LPV Simulation

`linsys` is an array of 25 linear state-space models, each with 1 input, 3 outputs, and 2 states. The models are discrete-time with a sample time of 25 ns. The bode plot shows significant variation in dynamics over the grid of scheduling parameters.

You can configure an LPV System block using the linear system array and the accompanying offset data (`uoff`, `yoff`, and `xoff`). The resulting LPV model serves as a linear system array approximation of the average dynamics. The `BoostConverterLPVModel` model uses such an LPV approximation.

```
lpvmdl = 'BoostConverterLPVModel';
open_system(lpvmdl)
```



For simulating the model, use an input profile for the duty cycle that roughly covers its scheduling range. Also, vary the resistive load to simulate load disturbances.

Generate the duty cycle profile `din`.

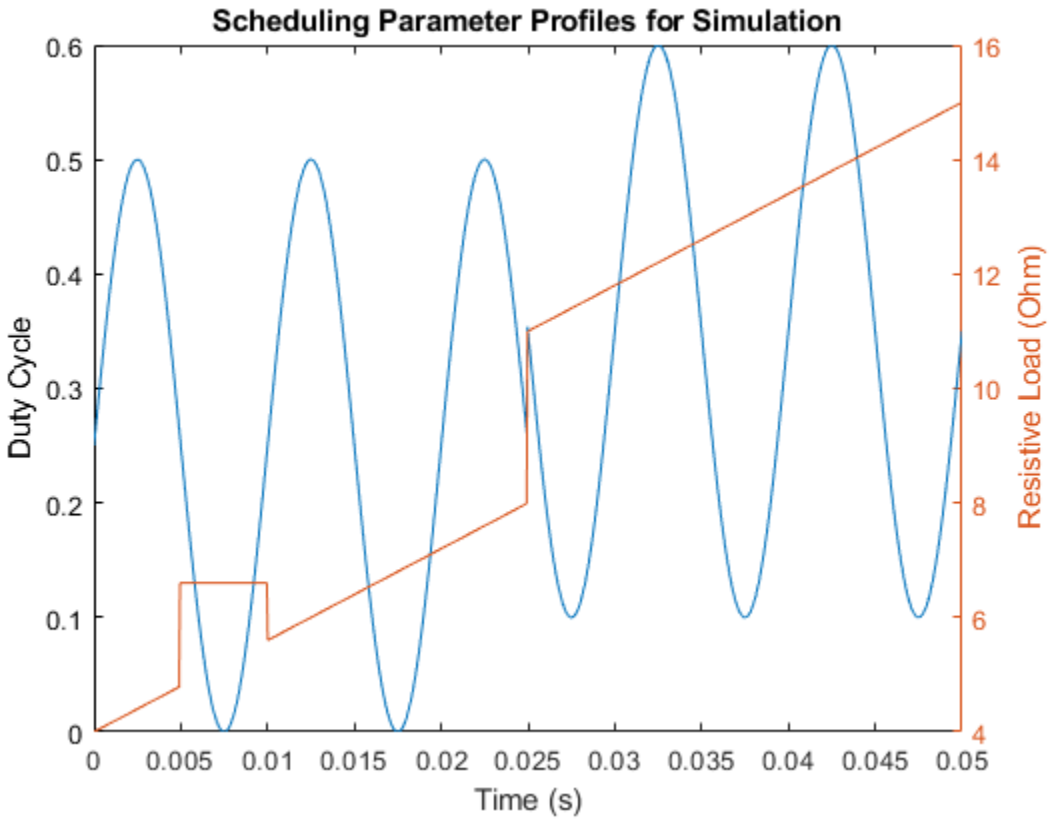
```
t = linspace(0,.05,1e3)';
din = 0.25*sin(2*pi*t*100)+0.25;
din(500:end) = din(500:end)+.1;
```

Generate the resistive load profile `rin`.

```
rin = linspace(4,12,length(t))';
rin(500:end) = rin(500:end)+3;
rin(100:200) = 6.6;
```

Plot the scheduling parameter profiles.

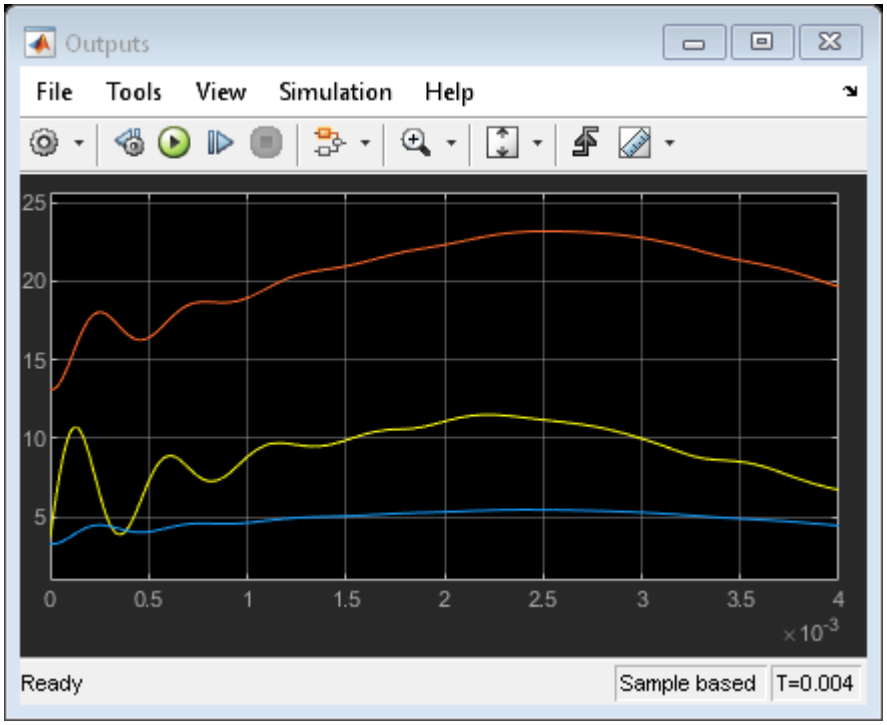
```
yyaxis left
plot(t,din)
xlabel('Time (s)')
ylabel('Duty Cycle')
yyaxis right
plot(t,rin)
ylabel('Resistive Load (Ohm)')
title('Scheduling Parameter Profiles for Simulation')
```



The code for generating the above signals has been added to the model `PreLoadFcn` callback for independent loading and execution. To override these settings and try your own, overwrite this data in the MATLAB® workspace.

Simulate the LPV model and view the resulting output.

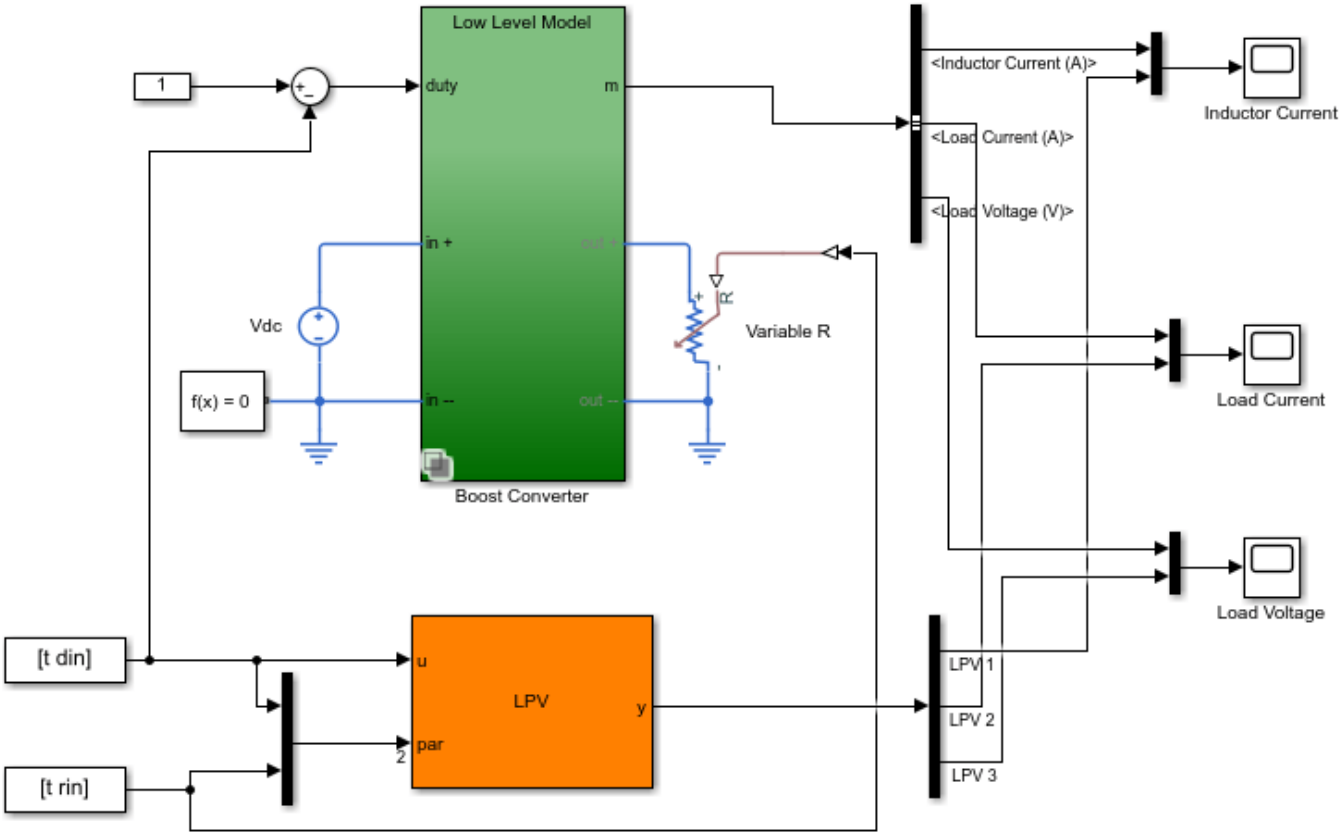
```
sim(lpvmdl,'StopTime','0.004');
open_system('BoostConverterLPVModel/Outputs')
```



The LPV model simulates significantly faster than the original `BoostConverterExampleModel` model.

To compare these simulation results with the original boost converter model simulation, use the `BoostConverterResponseComparison` model. This model uses the Boost Converter block configured to use the high-fidelity Low Level Model variant. It also contains the LPV System block. You can view the responses for both systems in the model scopes.

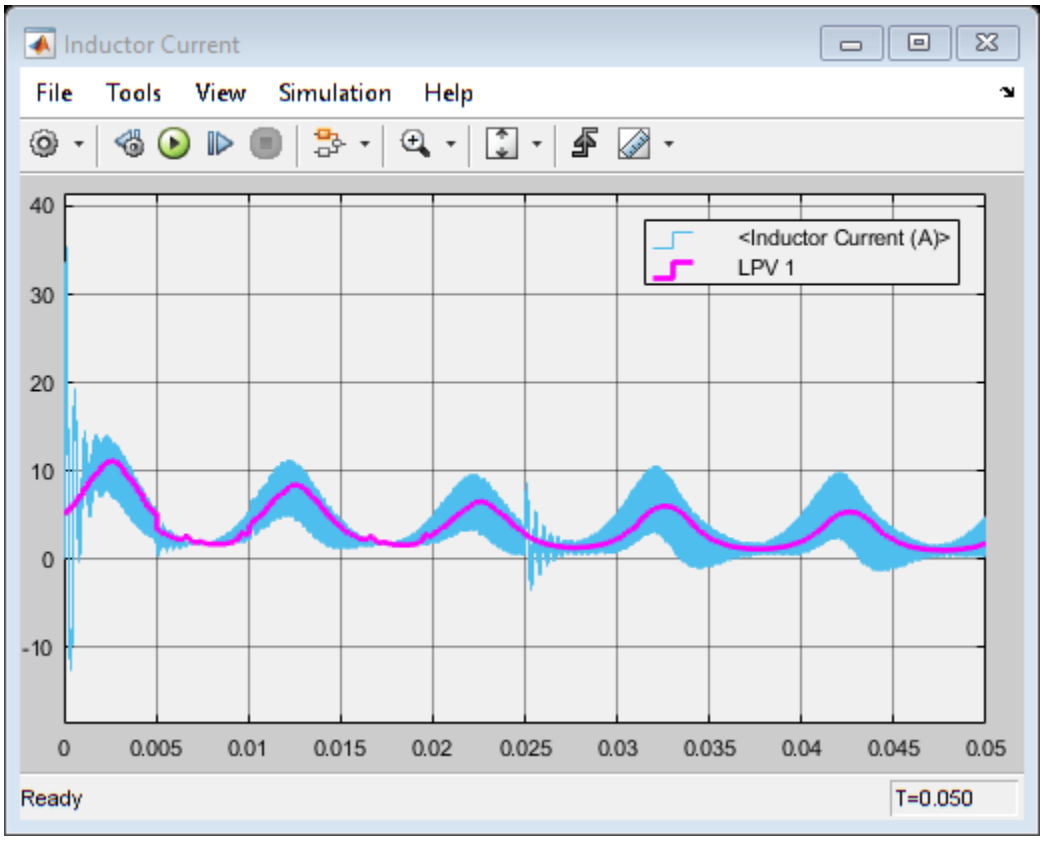
```
linsysd = c2d(linsys,Ts*1e4);
mdl = 'BoostConverterResponseComparison';
open_system(mdl)
```

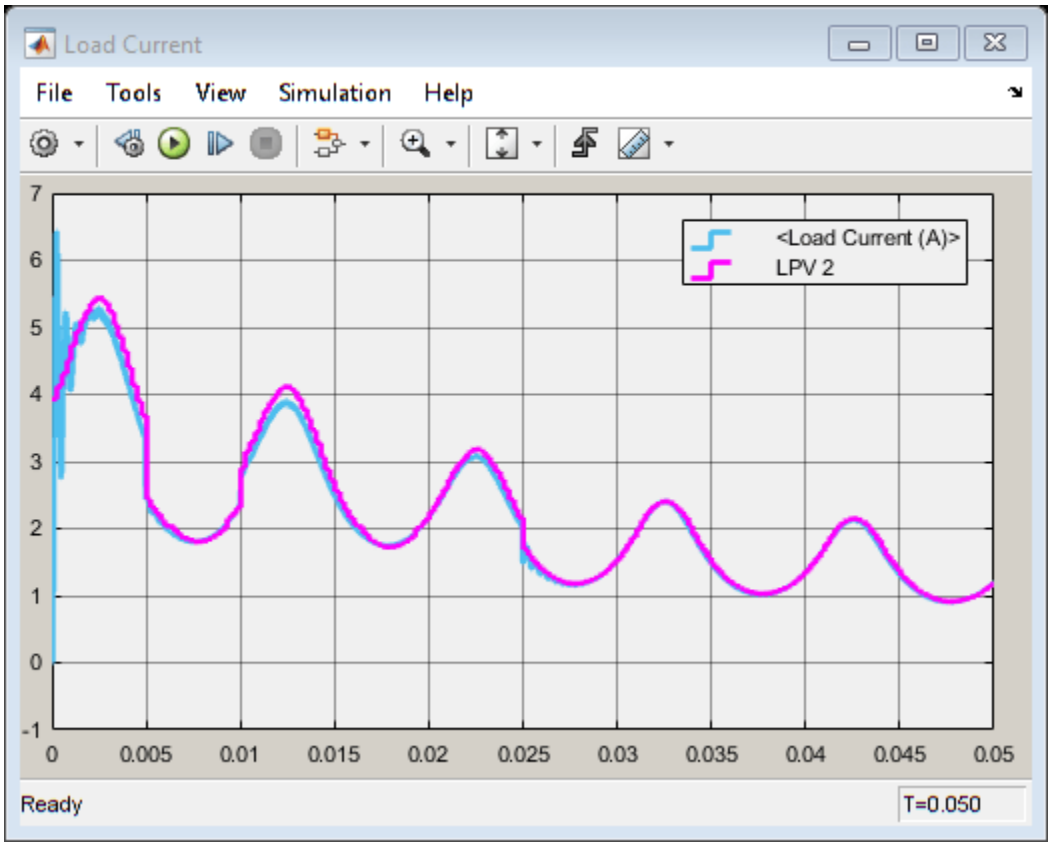


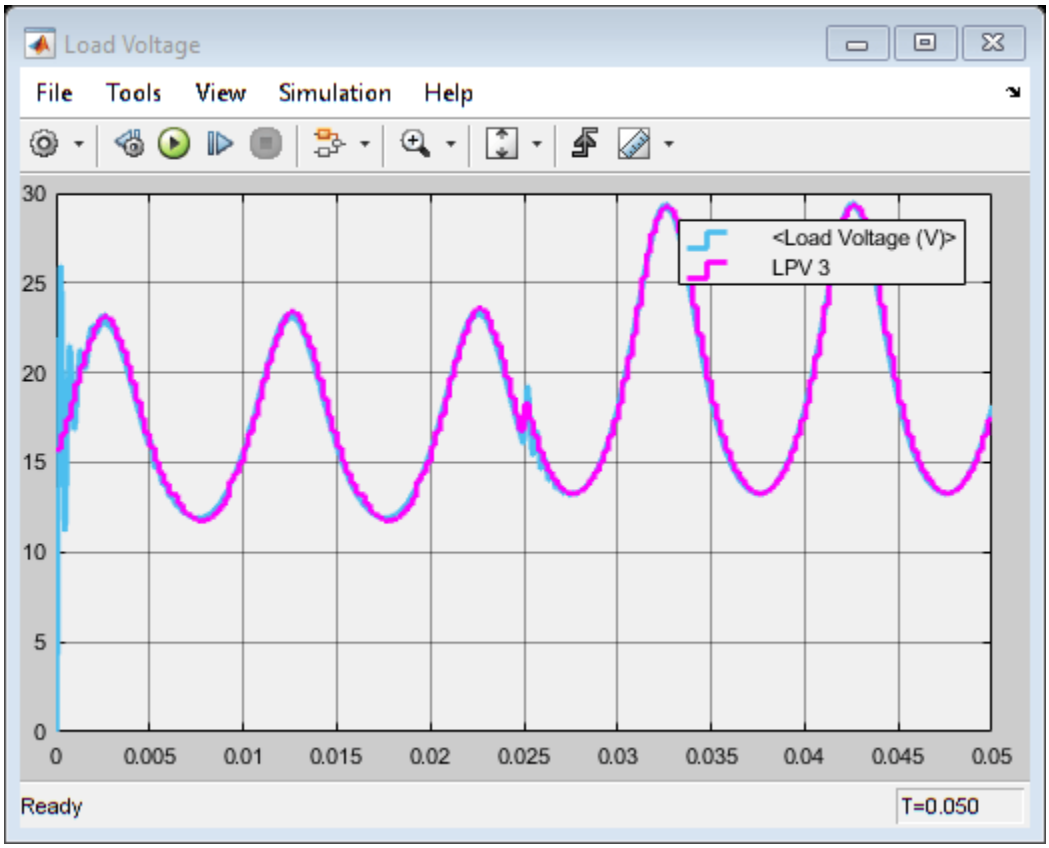
Simulate the model. The simulation runs quite slowly due to the fast switching dynamics in high-fidelity model. Uncomment the following code to simulate the model.

```
% sim(mdl);
```

The following figures show example simulation results for the inductor current, load current, and load voltage.


Inductor Current
File
Tools
View
Simulation
Help
<Inductor Current (A)>
LPV 1
40
30
20
10
0
-10
0
0.005
0.01
0.015
0.02
0.025
0.03
0.035
0.04
0.045
0.05
Ready
T=0.050


Load Current
File
Tools
View
Simulation
Help
<Load Current (A)>
LPV 2
7
6
5
4
3
2
1
0
-1
0
0.005
0.01
0.015
0.02
0.025
0.03
0.035
0.04
0.045
0.05
Ready
T=0.050



While the LPV model consumes less memory and simulates significantly faster than the high-fidelity model, it is able to emulate the average behavior of the boost converter.

See Also

LPV System | `linearize`

Related Examples

- “Batch Linearize Model at Multiple Operating Points Using linearize Command” on page 3-19
- “Approximate Nonlinear Behavior Using Array of LTI Systems” on page 3-70

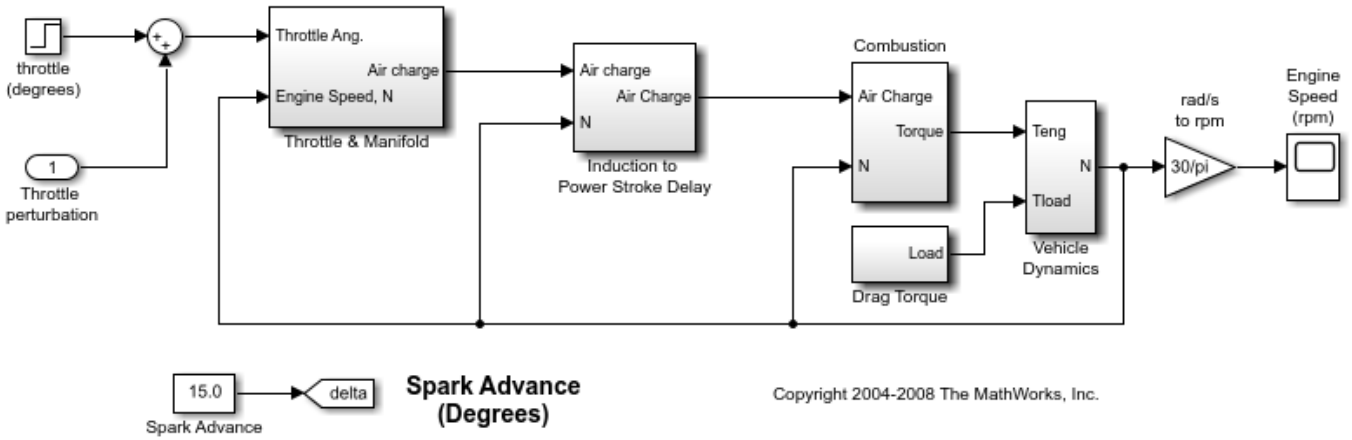
Linearize Engine Speed Model

This example shows how to linearize an engine speed model for multiple output conditions.

Engine Speed Model

Open the model.

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



For this example, you find linear models from the spark advance and throttle angle inputs to the output engine speed. You do so for three speed conditions: 2000, 3000, and 4000 rpm.

Find Operating Points

Create an array of three operating point specifications.

```
opspec = operspec(mdl,[3 1]);
```

Since the Simulink® model does not have any root-level inports, `opspec` does not contain any output specifications. You can add output specifications for a given signal in your model using the `addoutputspec` function.

Add an output specification to the output of the rad/s to rpm block.

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

For each specification, indicate that the output value is known and specify the output value. Set the known speed values to 2000, 3000, and 4000 rpm.

```
opspec(1).Outputs.Known = 1;
opspec(1).Outputs.y = 2000;
opspec(2).Outputs.Known = 1;
opspec(2).Outputs.y = 3000;
opspec(3).Outputs.Known = 1;
opspec(3).Outputs.y = 4000;
```

View the specifications object for the third operating condition.

```
opspec(3)
```

```
ans =


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

States:
----------
      <strong>x</strong>       <strong>Known</strong>    <strong>SteadyState</strong>    <strong>
    <strong>______</strong>    <strong>_____</strong>    <strong>___________</strong>    <strong>

(1.) scdspeed/Throttle & Manifold/Intake Manifold/p0 = 0.543 bar
     0.543    false       true        -Inf    Inf    -Inf      Inf
(2.) scdspeed/Vehicle Dynamics/w = T//J w0 = 209 rad//s
    209.48    false       true        -Inf    Inf    -Inf      Inf

Inputs:
----------
    <strong>u</strong>    <strong>Known</strong>    <strong>Min</strong>     <strong>Max</strong>
    <strong>_</strong>    <strong>_____</strong>    <strong>____</strong>    <strong>___</strong>

(1.) scdspeed/Throttle  perturbation
    0    false    -Inf    Inf

Outputs:
----------
     <strong>y</strong>      <strong>Known</strong>    <strong>Min</strong>     <strong>Max</str
    <strong>____</strong>    <strong>_____</strong>    <strong>____</strong>    <strong>___</str

(1.) scdspeed/rad//s to rpm
    4000    true     -Inf    Inf
```

Search for operating points that meet these specifications using the `findop` function.

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

View the resulting operating point for the third operating condition.

```
op(3)
```

```
ans =


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

States:
----------
      <strong>x</strong>
    <strong>______</strong>

(1.) scdspeed/Throttle & Manifold/Intake Manifold/p0 = 0.543 bar
    0.4731
(2.) scdspeed/Vehicle Dynamics/w = T//J w0 = 209 rad//s
```

```
    418.88

Inputs:
----------
      <strong>u</strong>
    <strong>______</strong>

(1.) scdspeed/Throttle  perturbation
    5.8292
```

Linearize Model

To linearize the model, first specify the linearization input points at the outputs of the throttle and Spark Advance blocks.

```
io(1) = linio('scdspeed/throttle (degrees)',1,'input');
io(2) = linio('scdspeed/Spark Advance',1,'input');
```

Next, specify the linearization output point at the output of the rad/s to rpm block.

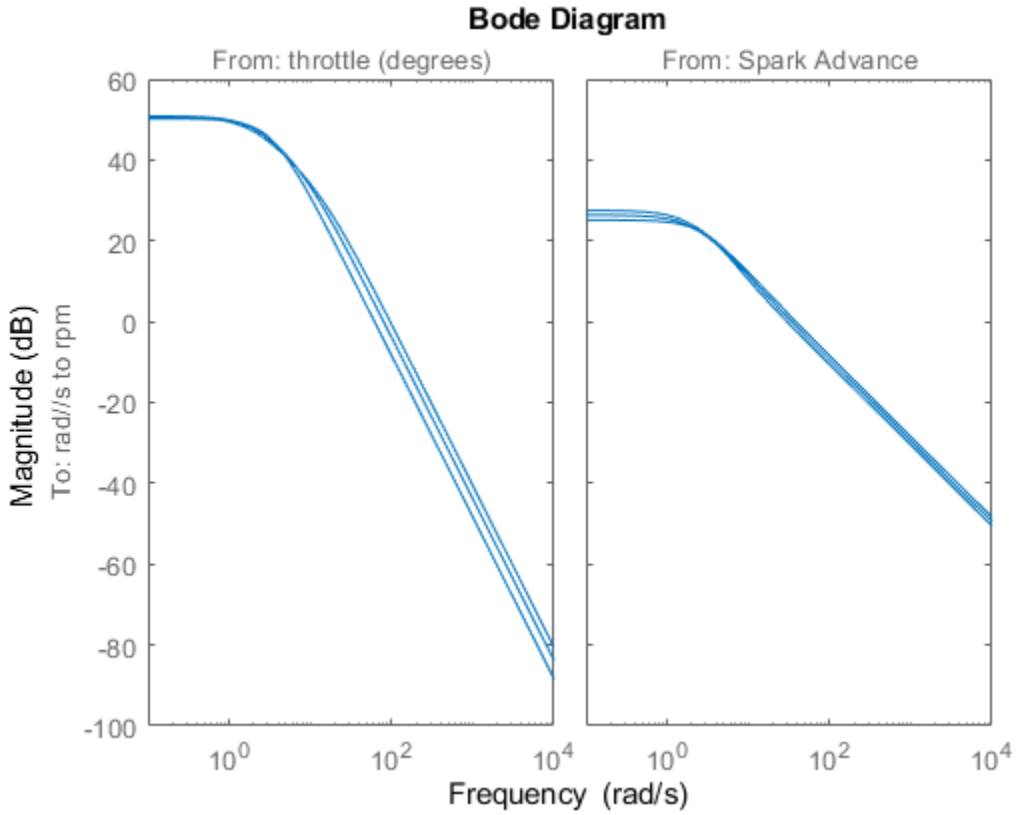
```
io(3) = linio('scdspeed/rad//s to rpm',1,'output');
```

Linearize the model for each of the operating conditions.

```
sys = linearize(mdl,op,io);
```

Plot the Bode magnitude response for the resulting linear models.

```
bodemag(sys)
```



Close the model.

```
bdclose(mdl)
```

See Also

`operspec` | `findop` | `linio` | `linearize`

Related Examples

- “Compute Steady-State Operating Points from Specifications” on page 1-13
- “Specify Portion of Model to Linearize” on page 2-10
- “Linearize at Trimmed Operating Point” on page 2-67
- “What Is Batch Linearization?” on page 3-2

Improve Linear Analysis Performance

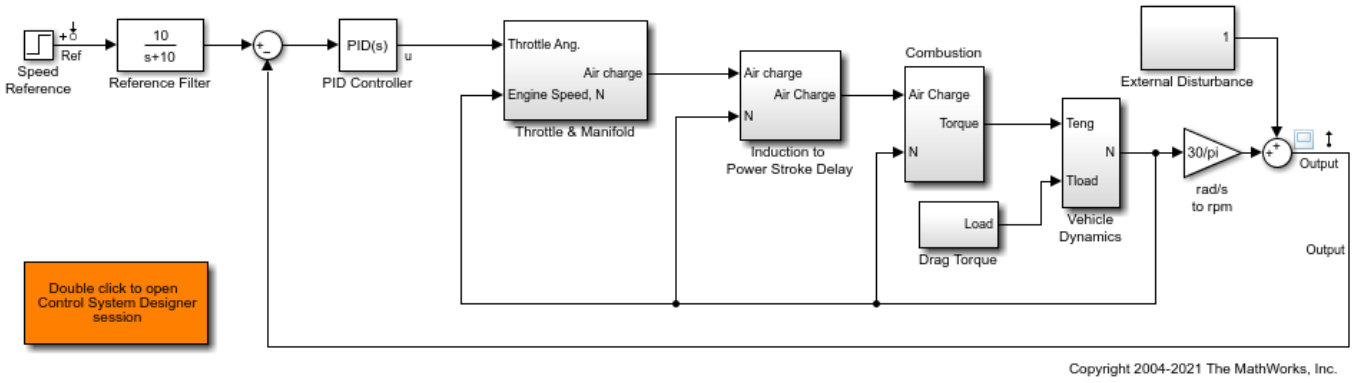
This example shows how to use the `fastRestartForLinearAnalysis` function to speed up multiple calls to compiling functions in Simulink® Control Design™ software such as `findop` and `linearize`.

Run Linear Analysis Commands in a Loop

In this example, you trim and linearize a closed-loop engine speed control model. You vary the PI control parameters and observe how the closed-loop behavior changes at steady state. Since `linearize` and `findop` are called in a loop, the model compiles 2*N + 1 times including the first call to `operspec`.

Open the engine speed control model and obtain the linear analysis points from the model.

```
mdl = 'scdspeedctrl';
open_system(mdl)
io = getlinio(mdl);
fopt = findopOptions('DisplayReport','off');
```



Configure the PI controller to use the base workspace variables `kp` and `ki`.

```
blk = [mdl,'/PID Controller'];
set_param(blk,'P','kp');
set_param(blk,'I','ki');
```

Create a grid of parameter values to vary.

```
vp = 0.0005:0.0005:0.003;
vi = 0.0025:0.0005:0.005;
[KP,KI] = ndgrid(vp,vi);
N = numel(KP);
sz = size(KP);
```

Initialize the base workspace variables.

```
kp = KP(1);
ki = KI(1);
```

Run the loop and record execution time.

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

```
for i = N:-1:1
    kp = KP(i);
    ki = KI(i);
    % Trim the model.
    op = findop(mdl,ops,fopt);
    [j,k] = ind2sub(sz,i);
    % Linearize the model.
    sysLoop(:,:,j,k) = linearize(mdl,io,op);
end
```

Calculate the elapsed time.

```
timeElapsedLoop = cputime - t;
```

Run Linear Analysis Commands in Batch

Rather than loop over the parameters, `findop` and `linearize` can accept a batch parameter variation structure directly to reduce the number of times the model compiles. In this case, the model compiles three times with calls to `operspec`, `findop`, and `linearize`.

Run and record execution time.

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

Create the batch parameter structure.

```
params(1).Name  = 'kp';
params(1).Value =  KP ;
params(2).Name  = 'ki';
params(2).Value =  KI ;
```

Trim the model across the parameter set.

```
op = findop(mdl,ops,params,fopt);
```

Linearize the model across the parameter and operating point set.

```
sysBatch = linearize(mdl,io,op,params);
```

Calculate the elapsed time.

```
timeElapsedBatch = cputime - t;
```

Run Linear Analysis Functions in Loop with Fast Restart

The `fastRestartForLinearAnalysis` function configures the model to minimize compilations even when compiling functions are run inside a loop. The model compiles with calls to `operspec`, `findop`, and `linearize` in a loop.

Run the loop and record execution time with fast restart for linear analysis enabled.

```
t = cputime;
```

Enable fast restart for linear analysis. Provide linear analysis points to minimize compilations between calls to `findop` and `linearize`.

```
fastRestartForLinearAnalysis(mdl,'on','AnalysisPoints',io);
```

Trim and linearize the model in a loop.

```
ops = operspec(mdl);
for i = N:-1:1
    kp = KP(i);
    ki = KI(i);
    % Trim the model.
    op = findop(mdl,ops,fopt);
    [j,k] = ind2sub(sz,i);
    % Linearize the model.
    sysFastRestartLoop(:,:,j,k) = linearize(mdl,io,op);
end
```

Turn off fast restart for linear analysis, which uncompiles the model.

```
fastRestartForLinearAnalysis(mdl,'off');
```

Calculate the elapsed time.

```
timeElapsedFastRestartLoop = cputime - t;
```

Run Linear Analysis Functions in Batch with Fast Restart

You can further improve performance by enabling fast restart for linear analysis and executing the batch `linearize` and `findop` functions. In this case, the model compiles once with calls to `operspec`, `findop`, and `linearize`.

Run and record execution time with fast restart for linear analysis on.

```
t = cputime;
```

Enable fast restart for linear analysis. Provide linear analysis points to minimize compilations between calls to `findop` and `linearize`.

```
fastRestartForLinearAnalysis(mdl,'on','AnalysisPoints',io)
```

Create the batch parameter structure.

```
params(1).Name  = 'kp';
params(1).Value =  KP ;
params(2).Name  = 'ki';
params(2).Value =  KI ;
```

Trim the model across the parameter set.

```
ops = operspec(mdl);
op = findop(mdl,ops,params,fopt);
```

Linearize the model across the parameter and operating point set.

```
sysFastRestartBatch = linearize(mdl,io,op,params);
```

Disable fast restart for linear analysis, which uncompiles the model.

```
fastRestartForLinearAnalysis(mdl,'off');
```

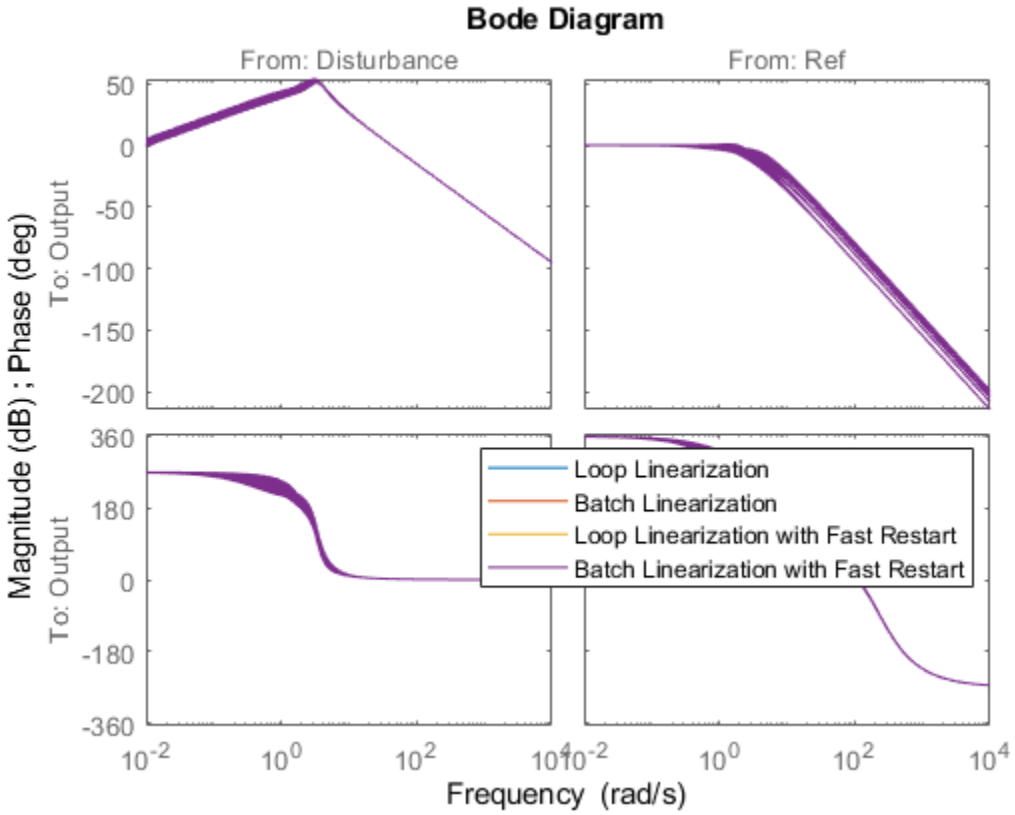
Calculate the elapsed time.

```
timeElapsedFastRestartBatch = cputime - t;
```

Compare the Results

Compare the linearization results of the four methods. The bode plot below shows that each method returns the same results.

```
compareIdx = 1:N;
bode(...
    sysLoop(:,:,compareIdx), ...
    sysBatch(:,:,compareIdx), ...
    sysFastRestartLoop(:,:,compareIdx), ...
    sysFastRestartBatch(:,:,compareIdx));
legend(...
    'Loop Linearization', ...
    'Batch Linearization', ...
    'Loop Linearization with Fast Restart', ...
    'Batch Linearization with Fast Restart')
```



Compile the elapsed time and speed-up ratio for each method in a table. You can obtain significant performance gains using batch trimming and linearization as well as `fastRestartForLinearAnalysis`.

```
Method = ["Loop","Batch","Fast Restart Loop","Fast Restart Batch"]';
TimeElapsed = [timeElapsedLoop timeElapsedBatch ...
    timeElapsedFastRestartLoop timeElapsedFastRestartBatch]';
SpeedUpFactor = TimeElapsed(1)./TimeElapsed;
TimeElapsedTable = table(Method,TimeElapsed,SpeedUpFactor)
```

```
TimeElapsedTable =

  4x3 table

          Method           TimeElapsed    SpeedUpFactor
    ____________________    ___________    _____________

    "Loop"                    71.484              1
    "Batch"                   15.844         4.5118
    "Fast Restart Loop"       20.766         3.4424
    "Fast Restart Batch"      14.609          4.893
```

Close the Simulink model.

```
bdclose(mdl)
```

See Also

`linearize` | `findop` | `getlinio` | `operspec`

Troubleshooting Linearization Results

- “Linearization Troubleshooting Overview” on page 4-2
- “Check Operating Point” on page 4-4
- “Check Analysis Point Placement” on page 4-5
- “Identify and Fix Common Linearization Issues” on page 4-6
- “Troubleshoot Linearization Results in Model Linearizer” on page 4-16
- “Troubleshoot Linearization Results at Command Line” on page 4-28
- “Find Blocks in Linearization Results Matching Specific Criteria” on page 4-37
- “Block Linearization Troubleshooting” on page 4-42
- “Speed Up Linearization of Complex Models” on page 4-48

Linearization Troubleshooting Overview

If you do not get expected results when you linearize your Simulink model, you can diagnose and fix potential linearization issues using Simulink Control Design troubleshooting tools. The definition of an *expected* linearization result depends on your specific application.

Troubleshooting Workflow

To determine whether a linearization is successful and find potential linearization issues, first check the equations and response plots of the linearized model.

Result to Check	Signs of Successful Linearization	Signs of Unsuccessful Linearization	More Information
Linear analysis plots	Time-domain and frequency-domain response plot characteristics, such as rise time and bandwidth respectively, capture the expected dynamics of your system.	Response plot characteristics do not capture the dynamics of your system. For example: • Bode plot gain is too large or too small. • Pole-zero plot contains unexpected poles or zeros.	"Analyze Results Using Model Linearizer Response Plots" on page 2-113.
Linear model equations	• State-space matrices have expected number of states, inputs, and outputs. The linearized model can have fewer states than your Simulink model because, often, the path between linearization input and output points does not reach all the model states. • Poles and zeros are in correct locations.	• Zero linearization (`D = 0`) • Infinite linearization (`D = Inf`)	"View Linearized Model Equations Using Model Linearizer" on page 2-111

If the response plots or model equations of the linearized system do not capture the expected dynamics of your system, check the:

- Operating point at which you linearized the model. For more information, see "Check Operating Point" on page 4-4.
- Analysis point placement in your model. For more information, see "Check Analysis Point Placement" on page 4-5.

Once you verify that the model operating point and analysis points are correct, if your model still does not linearize as expected, you can troubleshoot the linearization results using the Linearization Advisor. The Linearization Advisor is a troubleshooting tool that allows you to identify blocks in your model that are potentially problematic for linearization. For more information, see "Identify and Fix Common Linearization Issues" on page 4-6.

Once you have identified potentially problematic blocks, you can then troubleshoot the linearizations of the individual blocks using the Linearization Advisor. For more information, see "Block Linearization Troubleshooting" on page 4-42.

Troubleshoot Linearizations of Models with Special Characteristics

Some Simulink models and blocks do not linearize well or require special considerations during linearization.

Model Characteristic	Linearization Considerations	More Information
Large models	For some large complex models, you can systematically linearize specific model components. You can then check if these components linearize as expected.	"Specify Portion of Model to Linearize" on page 2-10
Models with delays	The method with which you represent time delays in your model can affect linearization results. For example, if a Bode plot shows insufficient lag in phase, the cause can be the Padé approximation of the model time delays.	• "Models with Time Delays" on page 2-137 • "Linearize Models with Delays" on page 2-78
Multirate models	Incorrect sample time and rate conversion methods can cause poor linearization results in multirate models.	"Linearize Multirate Models" on page 2-139
Models with PWM signals	Models with pulse width modulation signals do not linearize well due to their discontinuities and high-frequency switching components. Consider specifying a custom linearization for such blocks.	"Configure Models with Pulse Width Modulation Signals" on page 2-158
Models with Model Reference blocks	Linearization is not fully compatible with model reference blocks running in accelerator simulation mode. Configure these subsystems to run in normal mode during linearization.	"Linearize Models with Model References" on page 2-82
Simscape networks	Simscape networks commonly linearize to zero when a set of the system equation Jacobians are zero at a given operating condition.	"Linearize Simscape Networks" on page 2-160

See Also

Apps
Model Linearizer

More About

- "Identify and Fix Common Linearization Issues" on page 4-6
- "Block Linearization Troubleshooting" on page 4-42

Check Operating Point

To diagnose whether you used the correct operating point for linearization, simulate the model at the operating point you used for linearization.

The linearization operating point is incorrect when the critical signals in the model:

- Have unexpected values.
- Are not at steady state.

To fix the problem, compute a steady-state operating point, and repeat the linearization at this operating point. For more information, see “Compute Steady-State Operating Points” on page 1-5 and “Simulate Simulink Model at Specific Operating Point” on page 1-93.

See Also

More About

- “About Operating Points” on page 1-2
- “View and Modify Operating Points” on page 1-9

Check Analysis Point Placement

Incorrect placement of analysis points, including linearization I/Os and loop openings, can result in blocks being inappropriately included in or excluded from the linearization result linearization.

Check Linearization I/O Points Placement

After linearizing the model, check the block linearization values to determine which blocks are included in the linearization.

Blocks can be missing from the linearization path for different reasons.

Incorrect placement linearization I/O points can result in inappropriately excluded blocks from linearization. To fix the problem, specify correct linearization I/O points and repeat the linearization. For more information, see “Specify Portion of Model to Linearize” on page 2-10.

Blocks that linearize to zero (and other blocks on the same path) are excluded from linearization. To fix this problem, troubleshoot linearization of individual blocks, as described in “Block Linearization Troubleshooting” on page 4-42.

Check Loop Opening Placement

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

To fix the problem, check the individual block linearization values to identify which blocks are included in the linearization. If undesired blocks are included, place the loop opening on a different signal and repeat the linearization.

See Also

More About

- “Block Linearization Troubleshooting” on page 4-42
- “Opening Feedback Loops” on page 2-14
- “How the Software Treats Loop Openings” on page 2-31

Identify and Fix Common Linearization Issues

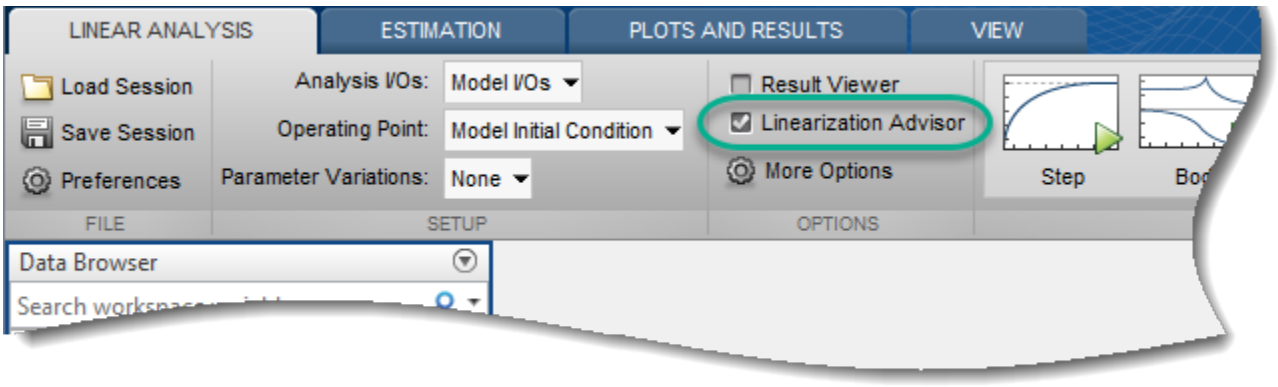
If your linearization results are not as expected, you can identify common linearization issues using the Linearization Advisor. The Linearization Advisor collects diagnostic information regarding individual block linearizations. Using this information, you can:

- View linearization details and operating points for each linearized block in your model.
- Identify potentially problematic blocks that cause common linearization issues.
- Determine which blocks are on and off the linearization path and which blocks contribute to the model linearization result.
- Search linearization results for blocks that meet specified criteria.

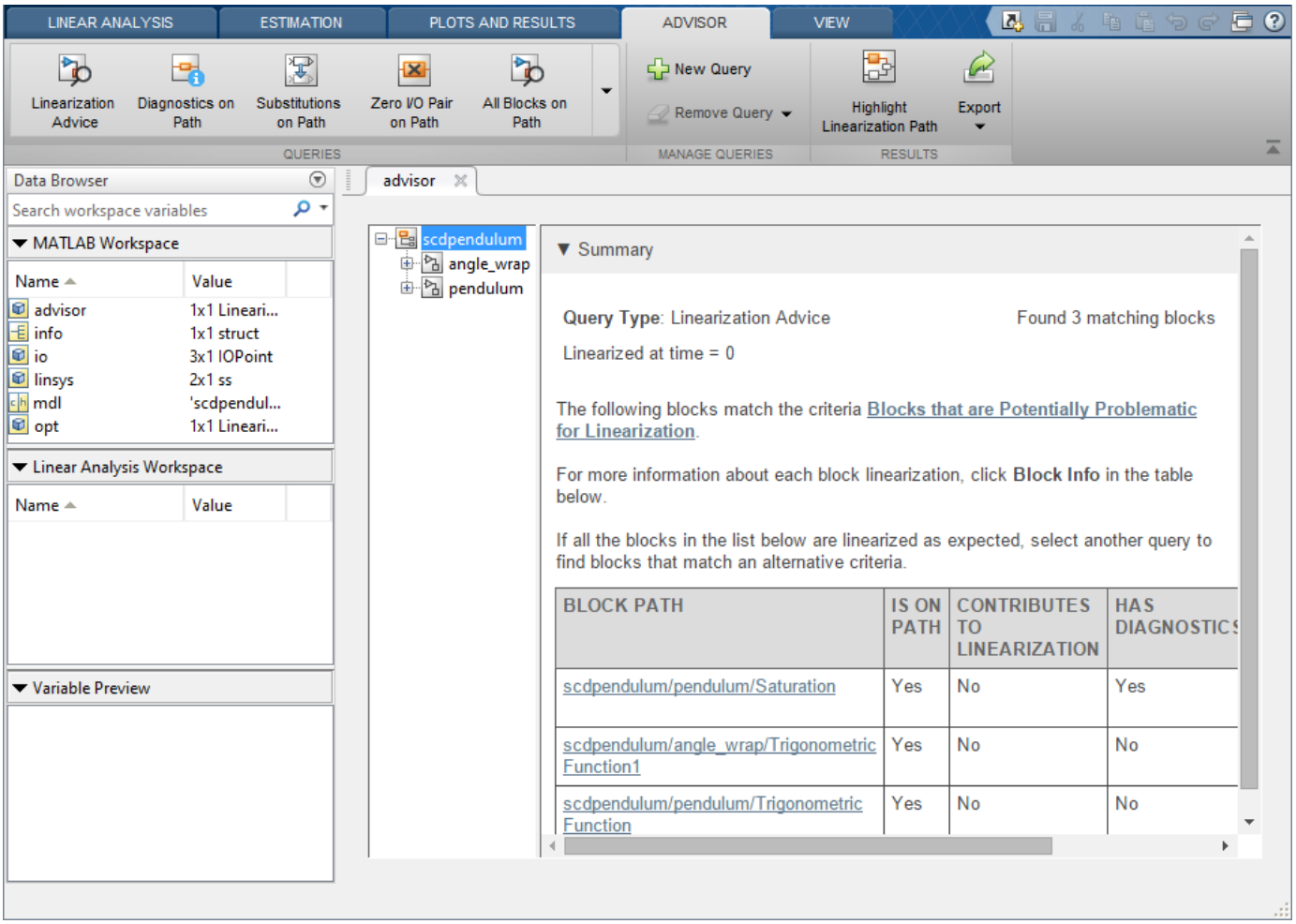
Enable Linearization Advisor

Since collecting diagnostic information adds linearization overhead, the Linearization Advisor is disabled by default. To collect diagnostic information, you must enable the Linearization Advisor before you linearize your model.

To enable the Linearization Advisor, in the **Model Linearizer**, on the **Linear Analysis** tab, select **Linearization Advisor**.



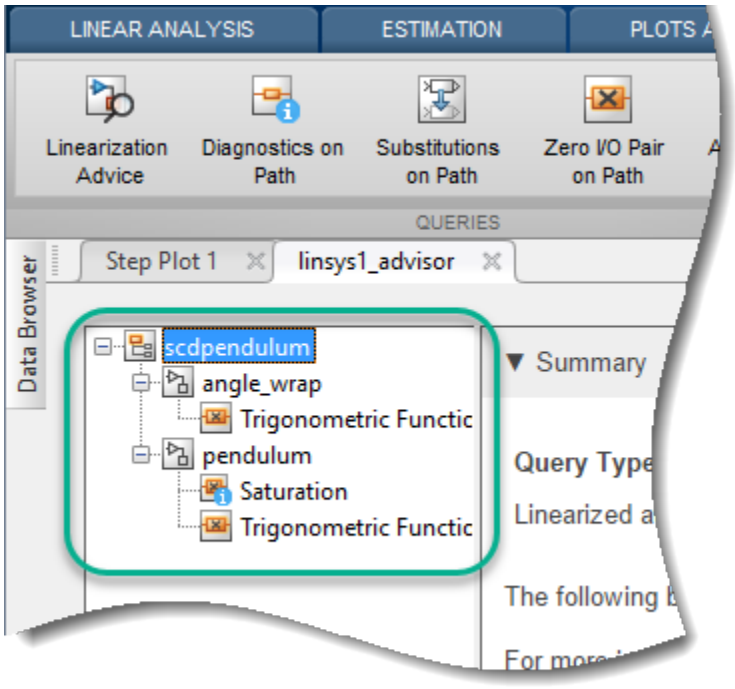
When you select this option and linearize your model, the software opens an **Advisor** tab for troubleshooting your linearization results.



Tip To make viewing the diagnostic information easier, you can minimize the **Data Browser**.

On the **Advisor** tab, you can gain insight into your model linearization by querying the diagnostic information. To do so, use the built-in queries in the **Queries** section, or create custom queries in the **Manage Queries** section.

When you run a query, the navigation tree lists the linearized blocks in your model that match the query search criteria. The tree structure reflects the model hierarchy.



To view a table of all blocks that match the search criteria, in the navigation tree, click the top-level model name. You can also view all blocks in a subsystem that satisfy the query by clicking the subsystem name. Each entry in the table summarizes the linearization diagnostics for a single block.

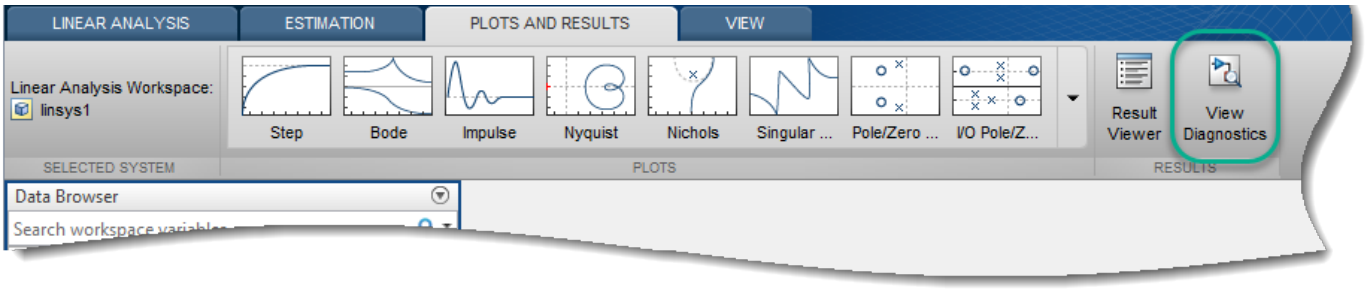
BLOCK PATH	IS ON PATH	CONTRIBUTES TO LINEARIZATION	HAS DIAGNOSTICS	LINEARIZATION METHOD	
scdpendulum/pendulum/Saturation	Yes	No	Yes	Exact	Block Info
scdpendulum/angle_wrap/Trigonometric Function1	Yes	No	No	Perturbation	Block Info
scdpendulum/pendulum/Trigonometric Function	Yes	No	No	Perturbation	Block Info

To view detailed diagnostic information for a block in a table, in the corresponding row, click **Block Info**. You can troubleshoot the block linearization using the detailed diagnostic information. For more information, see “Block Linearization Troubleshooting” on page 4-42.

For an example of interactive troubleshooting using the Linearization Advisor, see “Troubleshoot Linearization Results in Model Linearizer” on page 4-16.

Tip If you close the **Advisor** tab for a given linearization, you can reopen it from the **Plots and Results** tab.

In the **Linear Analysis Workspace**, select the linearized model you want to troubleshoot. Then, click **View Diagnostics**. This option is only available if you enabled the Linearization Advisor before linearizing the model.



You can also create a `LinearizationAdvisor` object when you linearize models at the command line. You can then troubleshoot the linearization results using the `advise` and `find` functions. For an example, see “Troubleshoot Linearization Results at Command Line” on page 4-28.

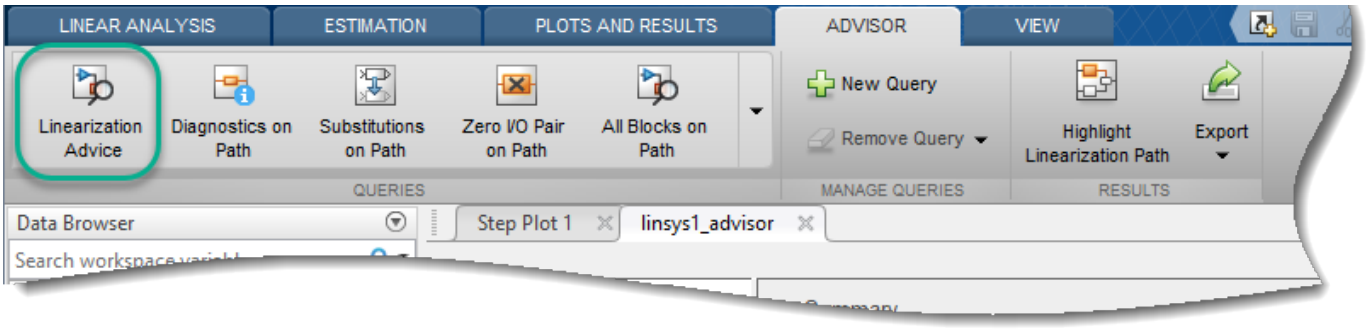
Blocks That Are Potentially Problematic for Linearization

As a starting point for troubleshooting, the Linearization Advisor searches the linearization diagnostic information for blocks that can cause common linearization issues. These potentially problematic blocks are on the linearization path and satisfy at least one of the following criteria.

Criteria	Description
Blocks with linearization diagnostic messages	Diagnostic messages indicate blocks with configurations or linearizations that correspond to common linearization problems.
Blocks that linearize to zero	Blocks with zero linearizations do not contribute to the linearization result and can remove other blocks from the linearization result.
Blocks with substituted linearizations	Errors in defining substitute linearizations can be difficult to diagnose.

For more information on the linearization path, see “Linearization Path” on page 4-11.

In the **Model Linearizer**, the diagnostic information for these blocks is listed on the **Advisor** tab when the tab first opens. Also, to access this diagnostic information at any time, in the **Queries** section, click **Linearization Advice**.



You can troubleshoot the linearizations of these blocks using the Linearization Advisor. For more information on troubleshooting block linearizations using diagnostic information, see “Block Linearization Troubleshooting” on page 4-42.

At the command line, the `advise` function returns diagnostic information for these blocks.

Blocks with Linearization Diagnostic Messages

Linearization diagnostic messages indicate blocks with properties or linearizations that correspond to common linearization problems. Fixing linearization issues identified in diagnostic messages is a good first step when troubleshooting your linearization.

Some block configurations that can generate diagnostic messages include:

- Blocks with non-floating-point input or output signals and no predefined exact linearization. Such blocks linearize to zero and generate diagnostic messages.
- Discontinuous blocks linearized at an operating point near a discontinuity. If such blocks are not treated as a gain during linearization, the software can generate diagnostic messages regarding their linearization.
- Blocks with least one input/output pair that linearizes to zero which causes a zero input/output pair in the overall model linearization.
- Blocks that do not support linearization because they do not have a predefined exact linearization and do not support numerical perturbation.

Some diagnostic messages propose solutions to their corresponding linearization issues. For example, when an input signal is outside the saturation limits of a Saturation block, the diagnostic message proposes treating the block as a gain during linearization.

Blocks That Linearize to Zero

A common cause of linearization issues is a block that unexpectedly linearizes to a gain of zero. To diagnose the cause of a zero block linearization, you can consider:

- Any corresponding diagnostic messages. These messages can highlight common causes of zero linearizations and propose potential solutions.
- The block operating point; that is the values of the block states and inputs at the model operating point used for linearization. For example, if the input to a saturation block is outside the block saturation limits, and the block is not configured to linearize as a gain, the block linearizes to zero.
- The block parameters. For example, if a block is configured to use non-floating-point inputs or states and is linearized using numerical perturbation, it linearizes to zero.

A zero block linearization does not necessarily indicate a linearization problem; that is, you may expect a block to linearize to zero under the expected operating conditions of the model. For example, if a Trigonometric Fcn block is configured as a `sin` function and the input value is $\pi/2$ at the model operating point, then the block linearizes to zero.

Blocks with Substituted Linearizations

Errors in defining a custom block linearization can be difficult to diagnose. After fixing issues related to diagnostic messages and zero linearizations, if your model still does not linearize as expected, verify that any substituted block linearizations in your model are correct.

For more information on specifying substitute block linearizations, see “When to Specify Individual Block Linearization” on page 2-122.

Find Specific Blocks in Linearization Results

If your model still does not linearize as you expect after fixing linearization issues related to potentially problematic blocks, you can query the Linearization Advisor for additional block diagnostic information. You can gain insight into your model linearization using this information. For example, you can investigate:

- Blocks that are linearized using numerical perturbation.
- Sampling rates of block linearizations in multirate models by finding blocks with a specified sample time.
- Blocks that have delays that can cause linearization issues.
- Blocks that are not on the linearization path.

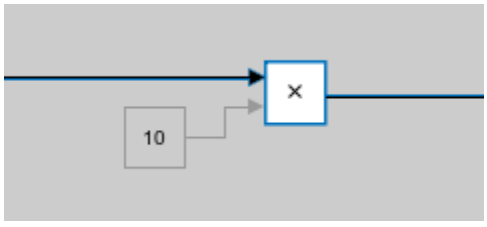
For more information, see “Find Blocks in Linearization Results Matching Specific Criteria” on page 4-37.

Linearization Path

The linearization path is the graphical connection in the Simulink model from the linearization inputs to the linearization outputs. A block is on the linearization path if at least one linearization input is connected to at least one linearization output through that block. For more information on specifying linearization inputs and outputs, see “Specify Portion of Model to Linearize” on page 2-10.

When a block is on the linearization path, its linearization can contribute to the overall model linearization. Blocks that linearize to zero do not contribute to the model linearization and can prevent branches of the linearization path from contributing to the model linearization.

Blocks that are not on the linearization path can still affect the linearization of other blocks, and therefore the model linearization, by modifying the operating points or parameters of the other blocks. For example, consider the following Product block that is on the linearization path (highlighted in blue):

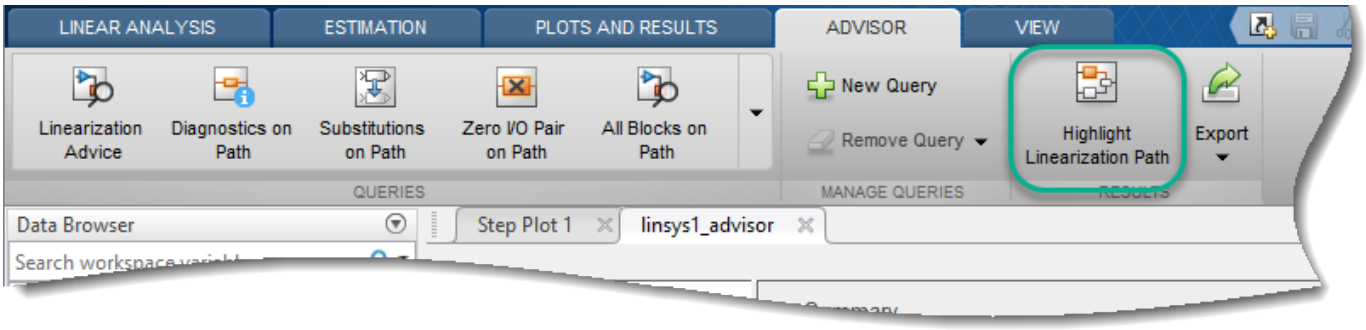


The constant block is not on the linearization path. However, the value of the constant affects the operating point of the Product block, which in turn affects the linearization from the first input of the Product block to the output.

Highlight Linearization Path

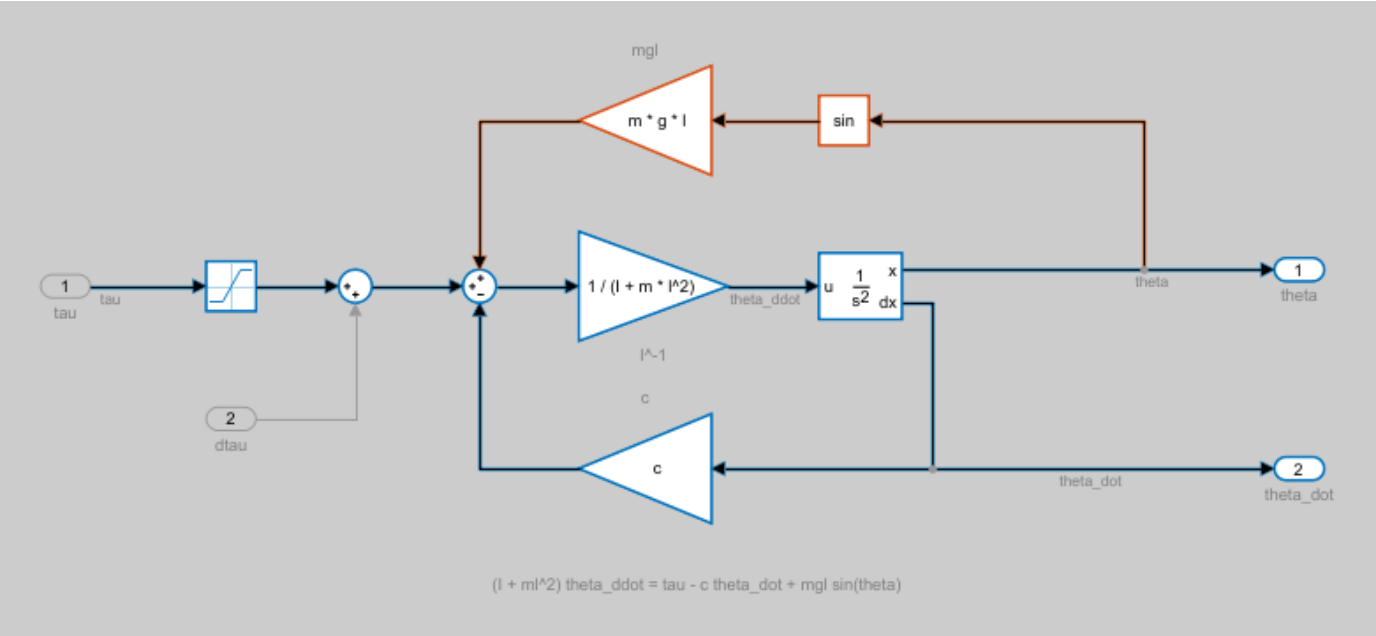
To visualize the linearization path and view blocks that contribute to the model linearization, you can highlight the linearization path in the Simulink model using the Linearization Advisor. A block is on the linearization path if there is a signal path from at least one linearization input to at least one linearization output that passes through the block.

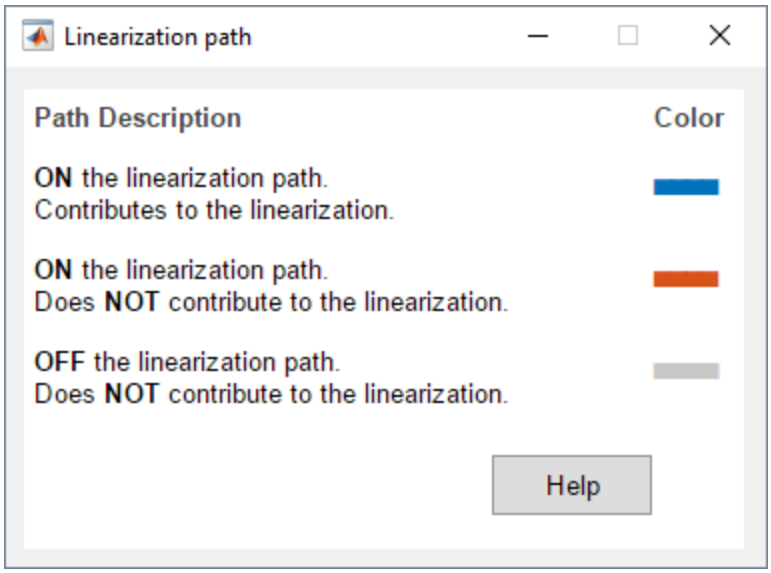
After you linearize your model with the Linearization Advisor enabled, to highlight the linearization path, in the **Model Linearizer**, on the **Advisor** tab, click **Highlight Linearization Path**.



The software highlights the linearization path in the model, showing which blocks are on the path and which blocks contribute to the model linearization. Blocks highlighted in:

- Blue are on the linearization path and numerically influence the model linearization.
- Red are on the linearization path, but have no influence on the model linearization due to at least one block on the linearization path that is linearized to zero.
- Gray are not on the linearization path and do not contribute to the model linearization.





To turn off the highlighting, close the Linearization path dialog box.

You can also highlight the linearization path from the command line using the `highlight` function.

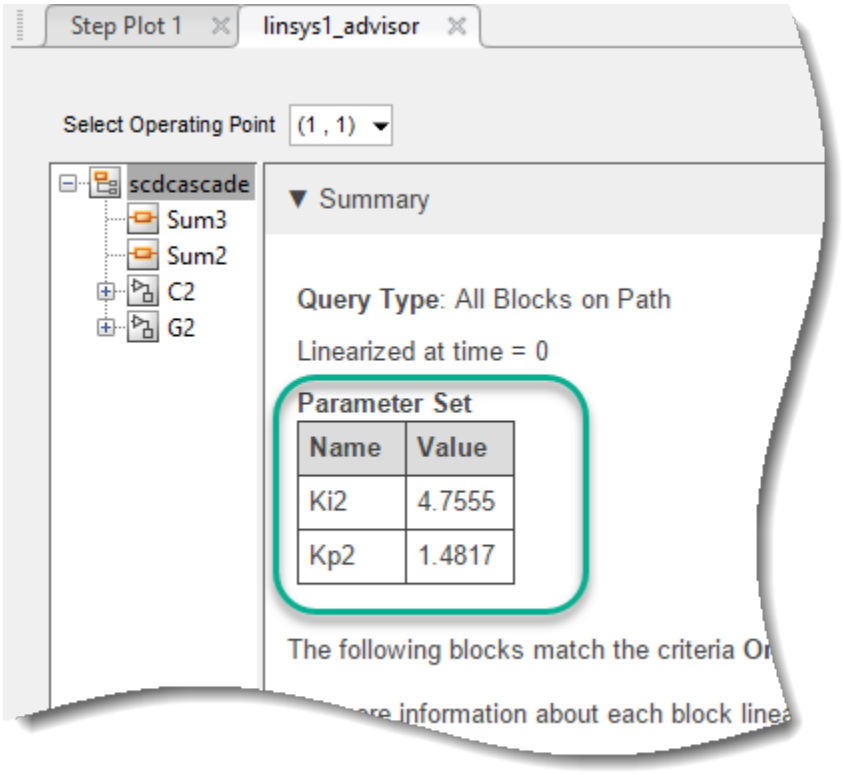
Troubleshoot Batch Linearizations

If you linearize your model at multiple operating points, you can troubleshoot each resulting linear model using Linearization Advisor.

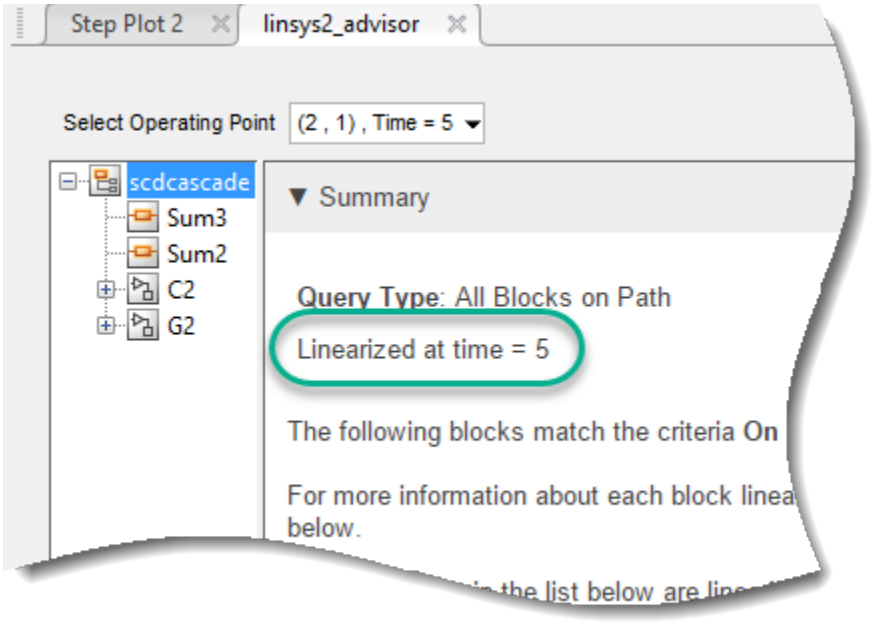
After batch linearizing the model, on the **Advisor** tab, in the **Select Operating Point** drop-down list, select the operating point for which you want to troubleshoot the linearization.

If you batch linearized your model using:

- Parameter variation, the linearization summary shows the parameter values that correspond to the selected operating point.



- Multiple simulation snapshot times, the linearization summary shows the time at which the model was linearized.



- Multiple trimmed operating points, the linearization summary does not show additional information about the operating point. To view details about the operating points, on the **Linear Analysis** tab, in the **Operating Point** drop-down list, select the operating point array used for linearization. In the same drop-down list, select `Edit`.

  Then, in the Edit dialog box, in the **Select Operating Point** drop-down list, select an operating point. The location of the operating point in this drop-down list corresponds to the location in the drop-down list on the **Advisor** tab.

See Also

Apps
Model Linearizer

Functions
`advise`

More About

- “Find Blocks in Linearization Results Matching Specific Criteria” on page 4-37
- “Troubleshoot Linearization Results in Model Linearizer” on page 4-16
- “Troubleshoot Linearization Results at Command Line” on page 4-28

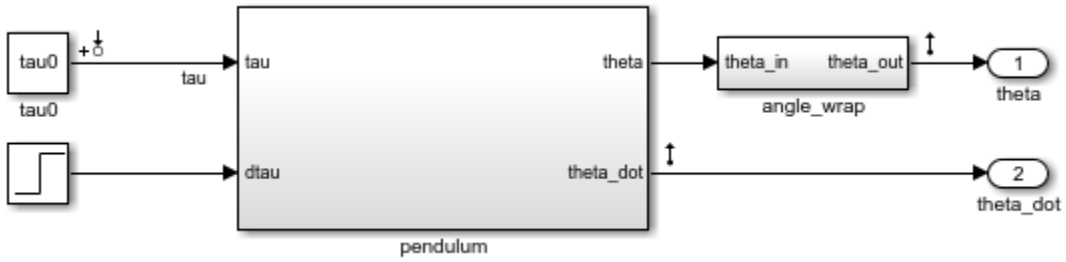
Troubleshoot Linearization Results in Model Linearizer

This example shows how to use the Linearization Advisor to debug the linearization of a pendulum model in the **Model Linearizer**.

Setup Model

Open the Simulink® model.

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



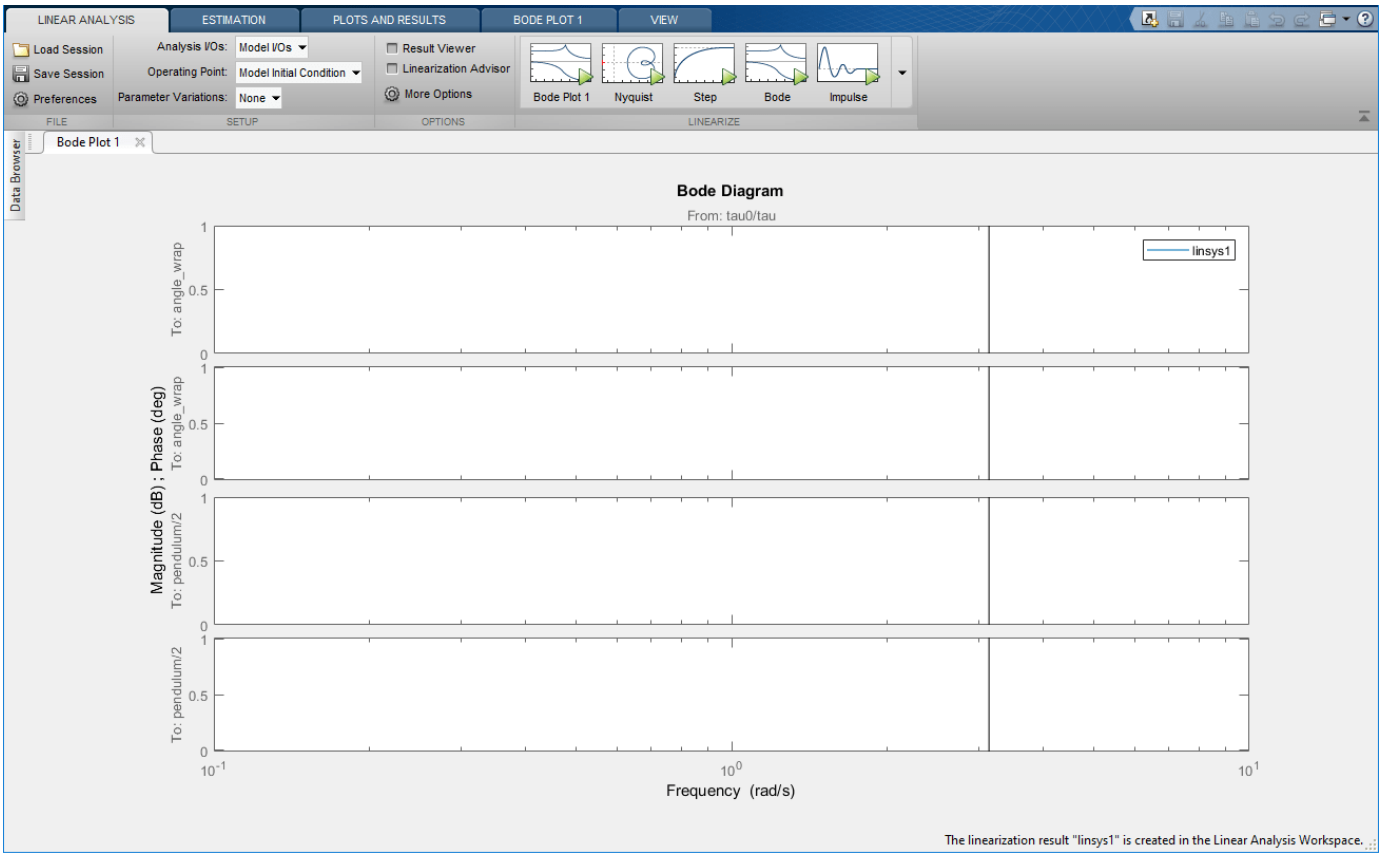
The initial condition for the pendulum angle is 90 degrees counterclockwise from the upright unstable equilibrium of 0 degrees. The initial condition for the pendulum angular velocity is 0 deg/s. The nominal torque to maintain this state is -49.05 N m. This configuration is saved as the model initial condition.

Open Model Linearizer and Linearize Model

To open **Model Linearizer**, in the Simulink model window, on the **Apps** tab, click **Model Linearizer**.

To linearize the model at the model initial condition, in **Model Linearizer**, on the **Linear Analysis** tab, click **Bode**.

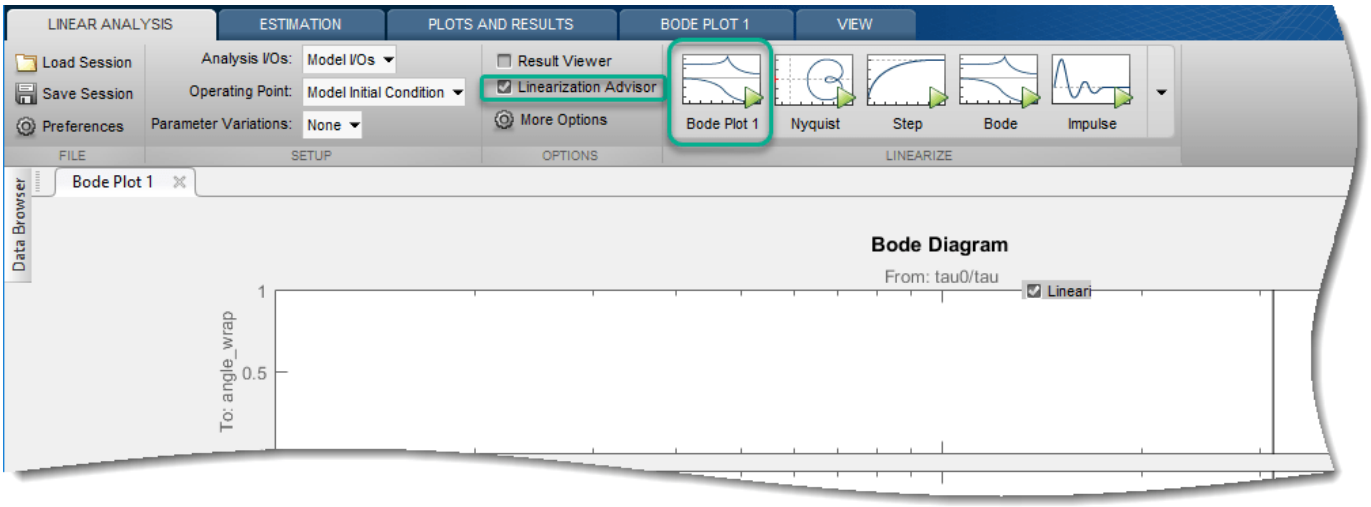
The software linearizes the model and plots its frequency response.



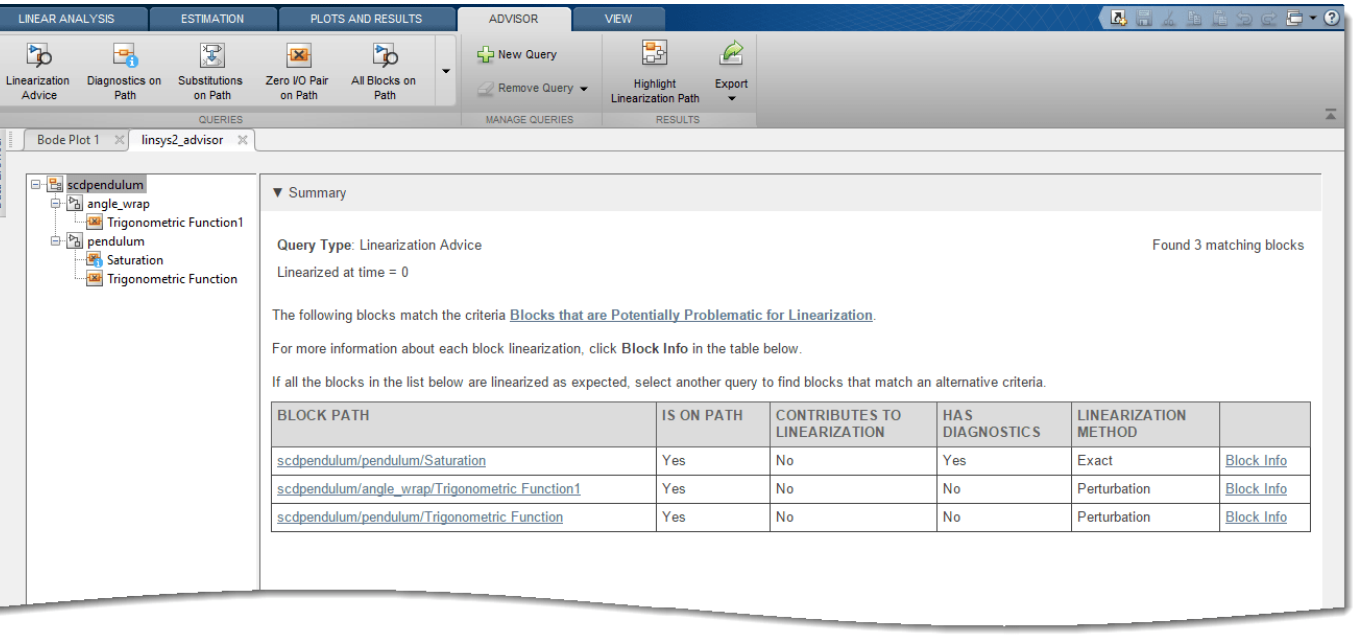
As can be seen from the Bode plot, the system linearizes to zero such that the torque has no effect on the angle or angular velocity. You can explore why this is the case using the Linearization Advisor.

Linearize Model with Advisor Enabled

To relinearize the model and generate an advisor, select **Linearization Advisor**, and click **Bode Plot 1**.



The software linearizes the model, creates the **linsys2_advisor** document, and opens the **Advisor** tab.

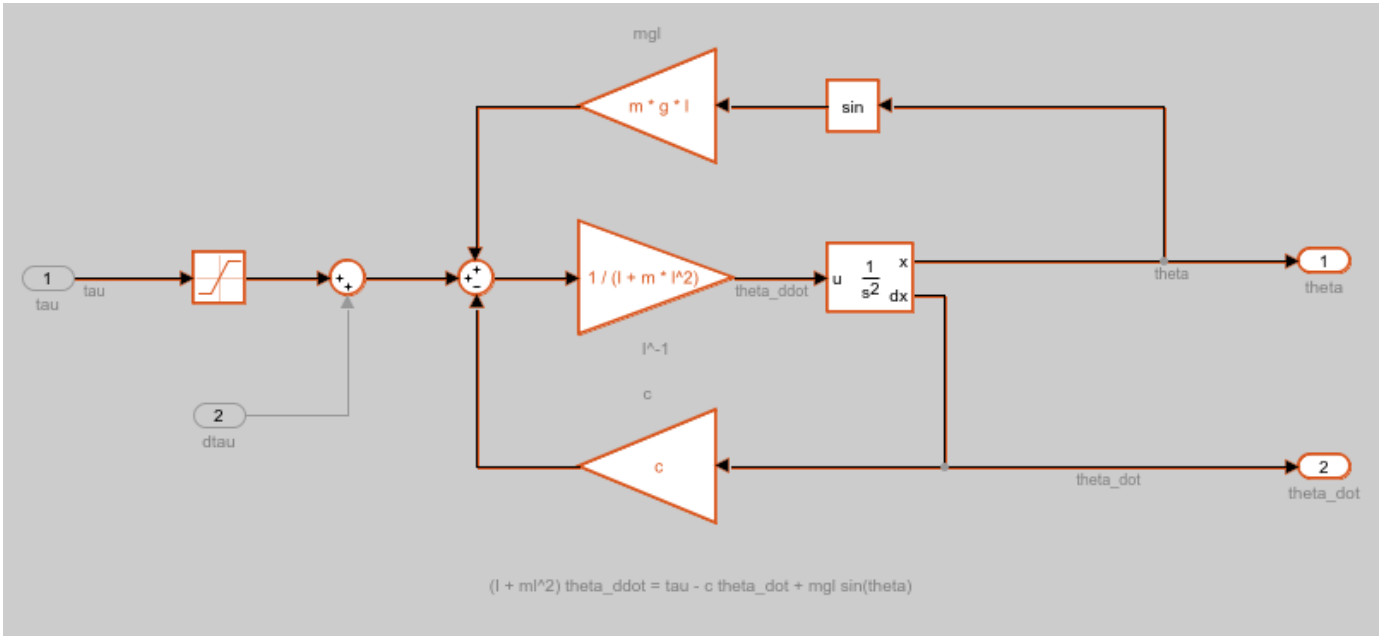


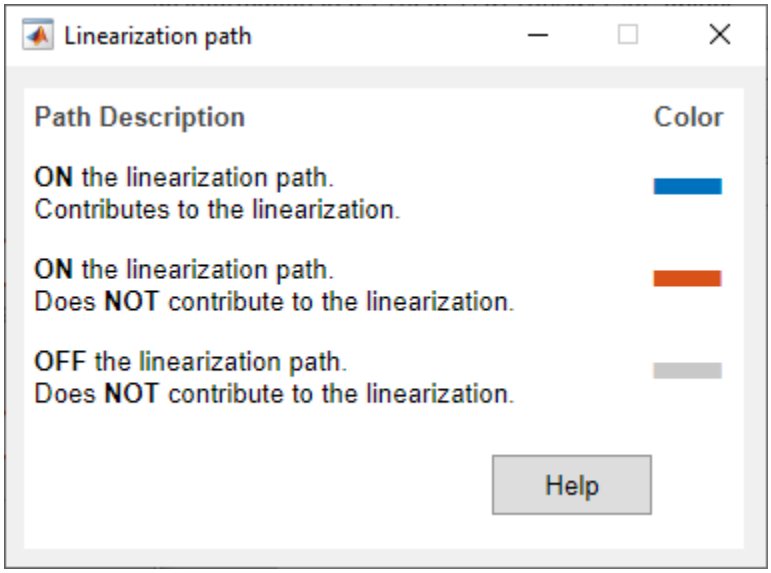
Highlight Linearization Path

To show the linearization path for the current linearization, on the **Advisor** tab, click **Highlight Linearization Path**. In the Simulink model, the blocks highlighted in:

- Blue numerically influence the model linearization.
- Red are on the linearization path but do not influence the model linearization for the current operating point and block parameters.

For convenience, only the blocks underneath the pendulum subsystem are shown.



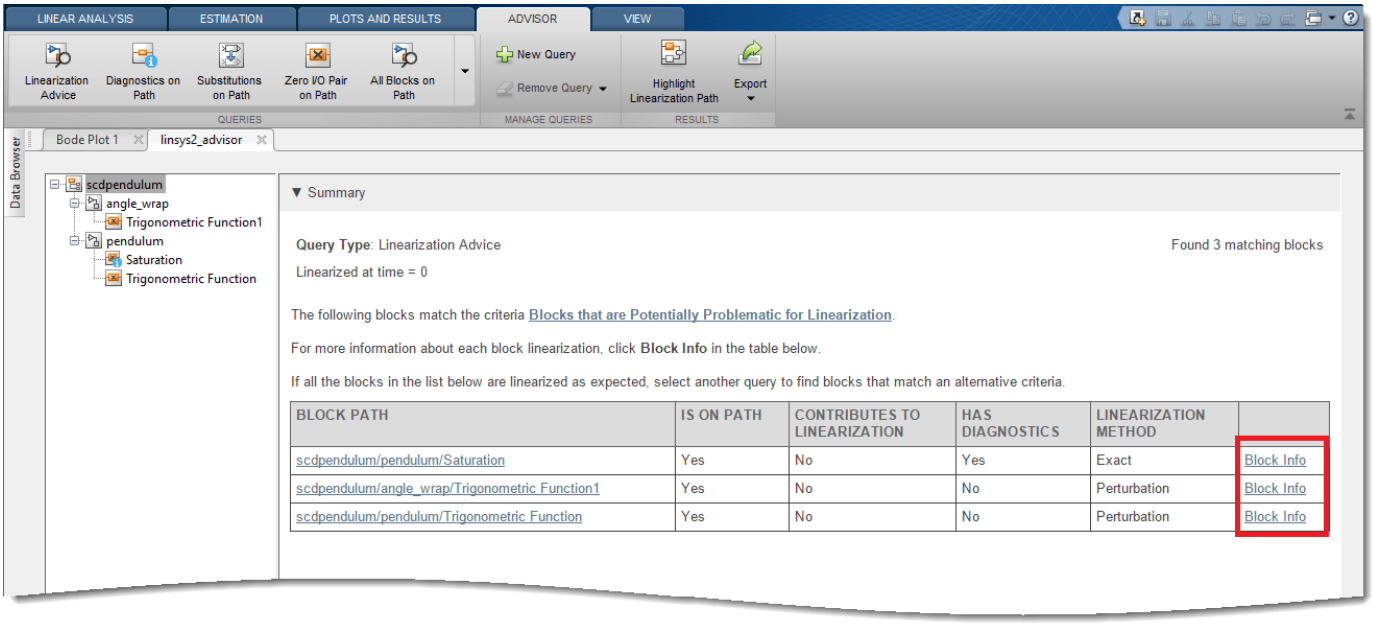


In this case, since the model linearized to zero, there are no blocks that contribute to the linearization.

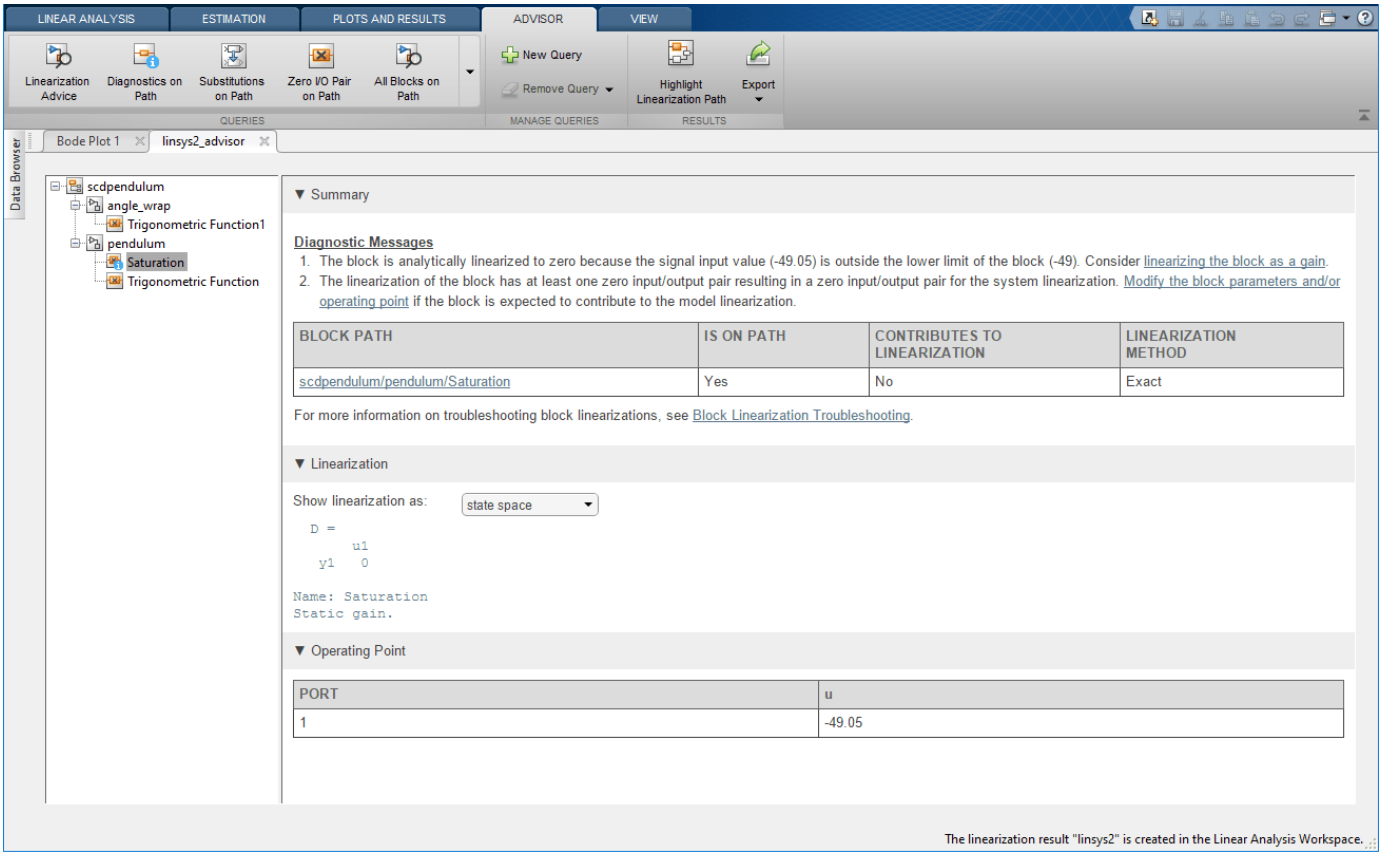
Investigate Potentially Problematic Blocks Using Advisor

The **linsys2_advisor** document shows a table listing blocks that may be problematic for the linearization.

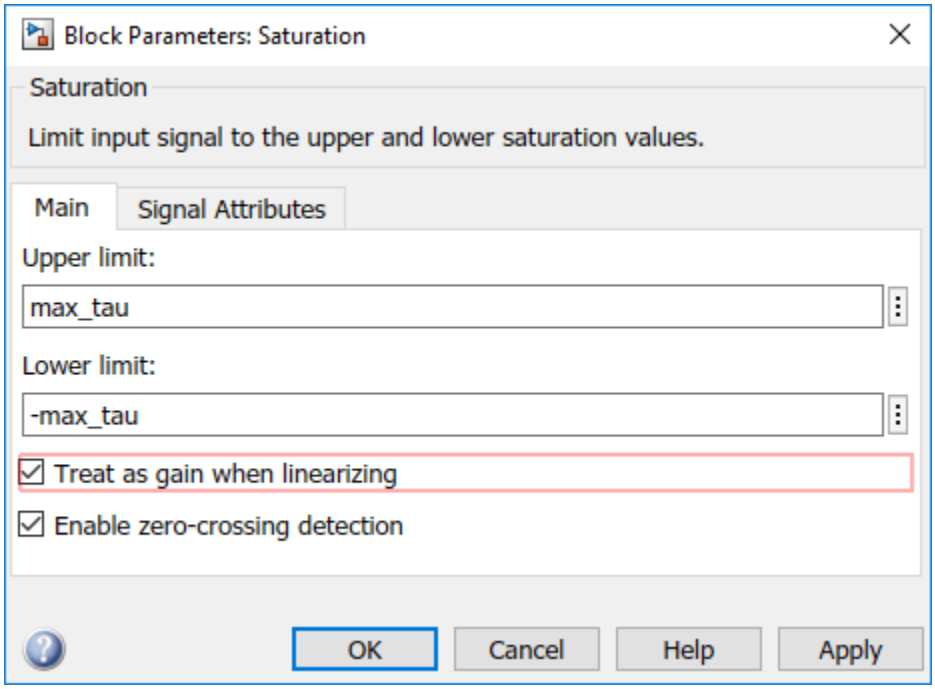
To view more information about a specific block linearization, in the corresponding row of the table, click **Block Info**.



In this case, three blocks are reported by the advisor, a Saturation block and two Trigonometric Function blocks. Investigate the Saturation block first since it has diagnostics. To do so, in the first row of the table, click **Block Info**.



There are two diagnostic messages for the Saturation block. The first message indicates that the block is linearized outside of its lower saturation limit of `-49`, since the input operating point is `-49.05`. The message also states the block can be linearized as a gain, which will linearize the block as `1` regardless of the input operating point. To do so, first click **linearizing the block as a gain**, which highlights the corresponding parameter in the block dialog box. Then, select the **Treat as gain when linearizing** parameter.

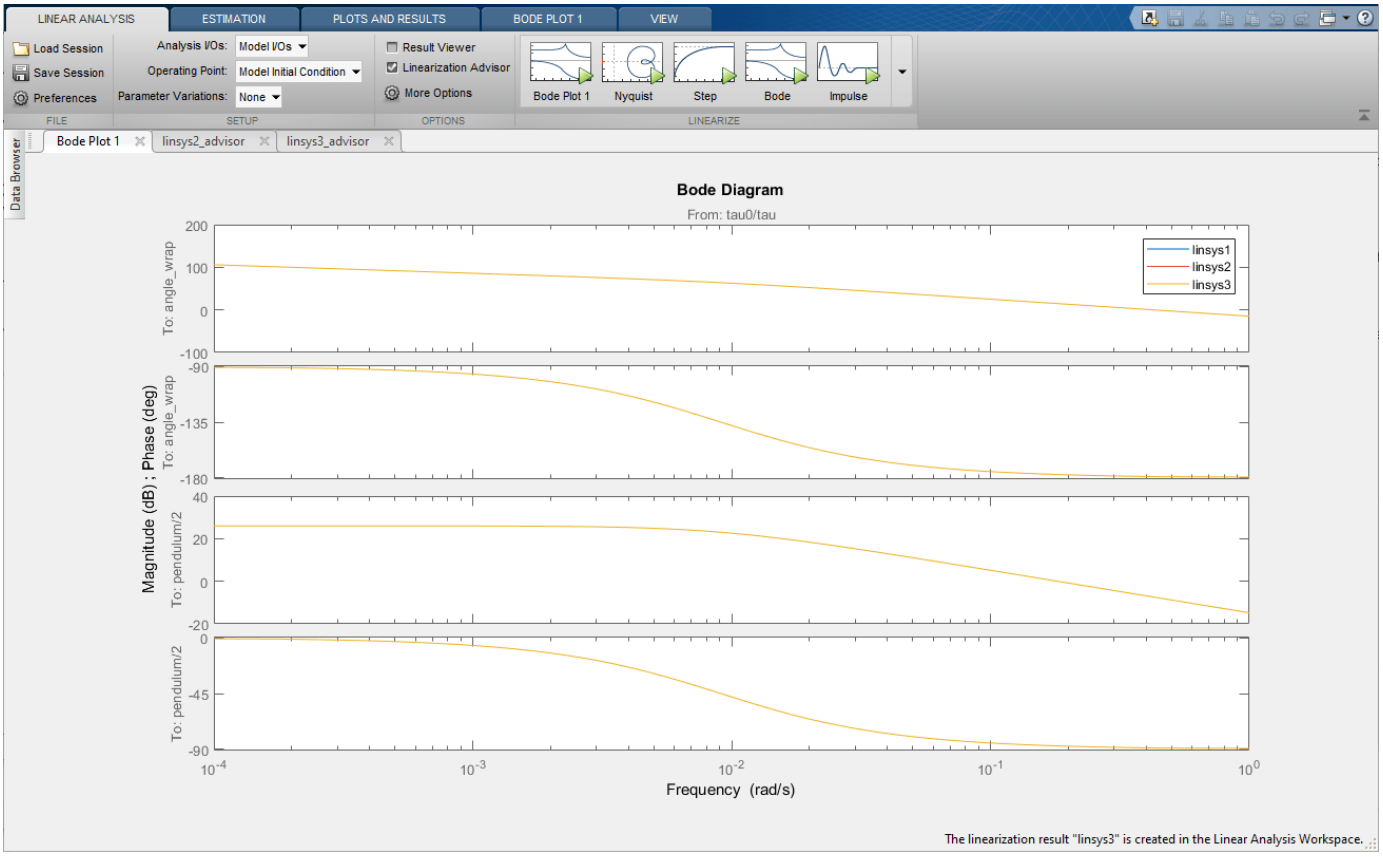


The second message states that the linearization of this block causes the model to linearize to zero. As shown in the **Linearization** section, the block is linearized to zero. Therefore, modifying the block linearization is a good first step toward obtaining a nonzero model linearization.

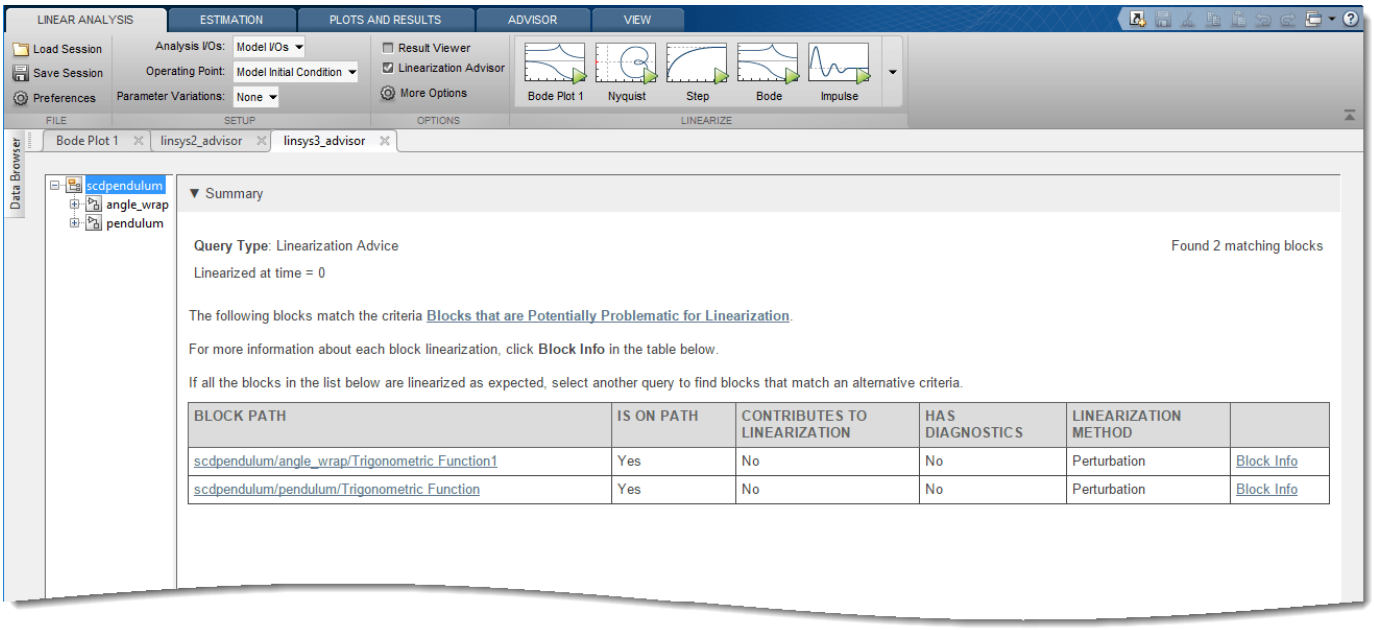
Relinearize Model

After setting the Saturation block to be treated as a gain, relinearize the model. For now, ignore the diagnostics for the two Trigonometric Function blocks.

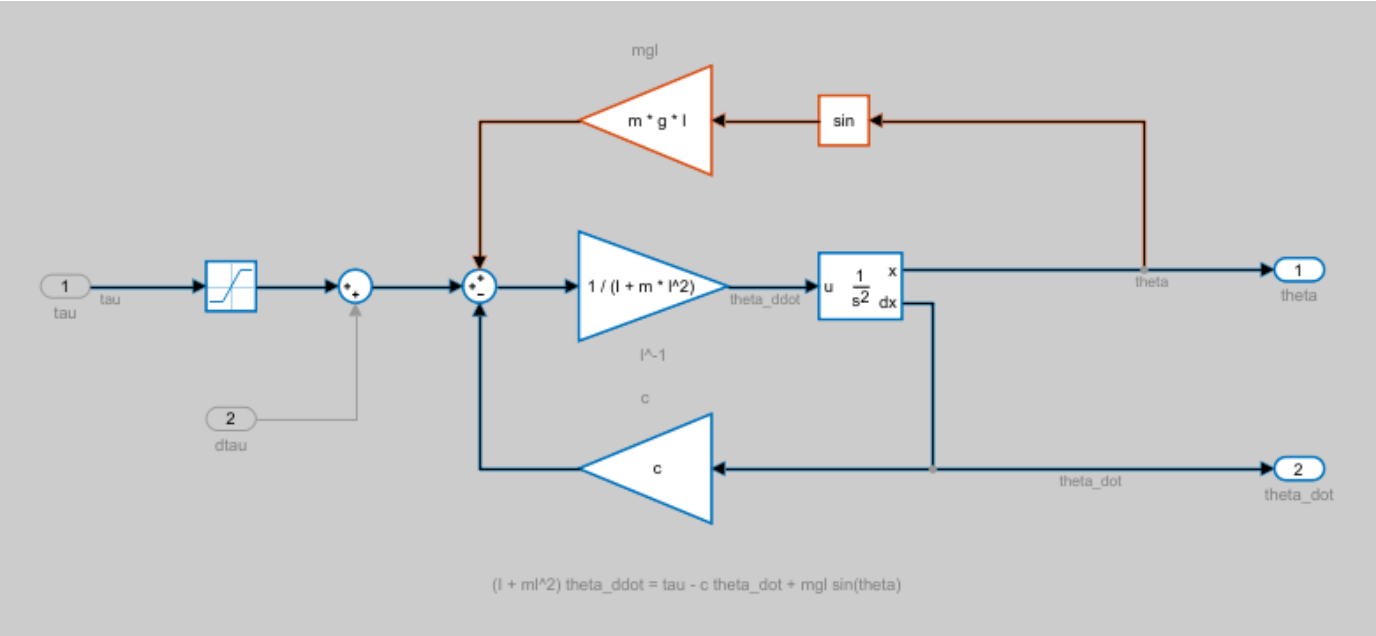
To relinearize the model, on the **Linear Analysis** tab, click **Bode Plot 1**. The **Bode Plot 1** document updates, showing the nonzero response of `linsys3`.



In the corresponding **linsys_advisor3** document, the Saturation block is no longer listed. However, the two Trigonometric Function blocks are still shown.

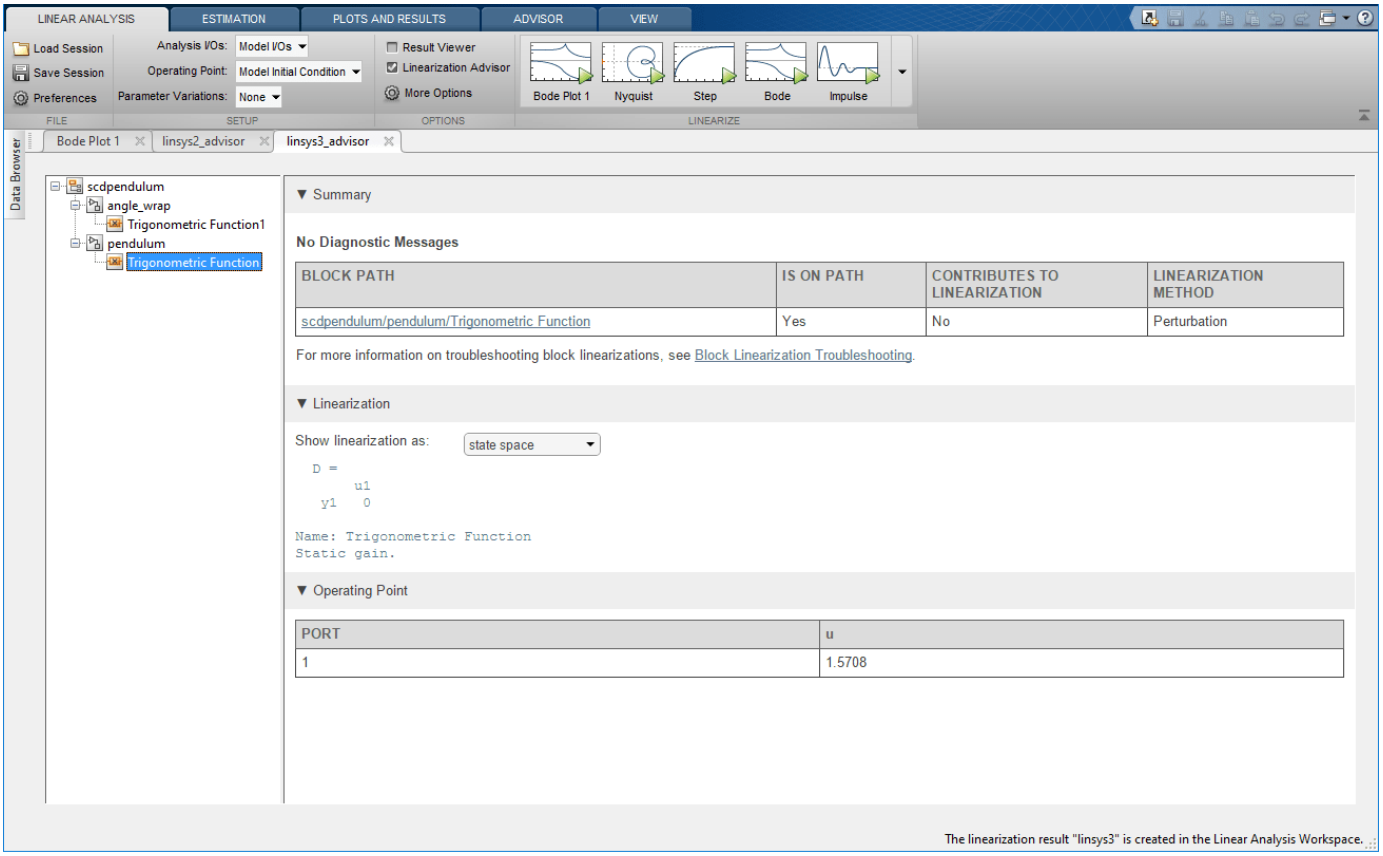


Highlight the linearization path.



Most of the blocks are now contributing to the model linearization, except for the paths going through the listed Trigonometric Function blocks.

To understand why these blocks are not contributing to the linearization, navigate to the blocks from the **linsys3_advisor** document. For example, click **Block Info** in the second row of the table.



For this Trigonometric Function block, the linearization is zero and the input operating point is $\pi/2 = 1.5708$.

You can find the linearization of the block analytically by taking the first derivative of the `sin` function with respect to the inputs:

$$\frac{\partial}{\partial u}\sin(u) = \cos(u)$$

Therefore, when evaluated at $u = \pi/2$ the linearization of the block is zero. The source of the input is the first output of the second-order integrator, which is dependent upon the state **theta**. Therefore, this block will linearize to zero if $\theta = \pi/2 + k\pi$, where k is an integer. The same condition applies for the other Trigonometric Function in the angle_wrap subsystem.

If these blocks are not expected to linearize to zero, you can modify the operating point state **theta**, and relinearize the model.

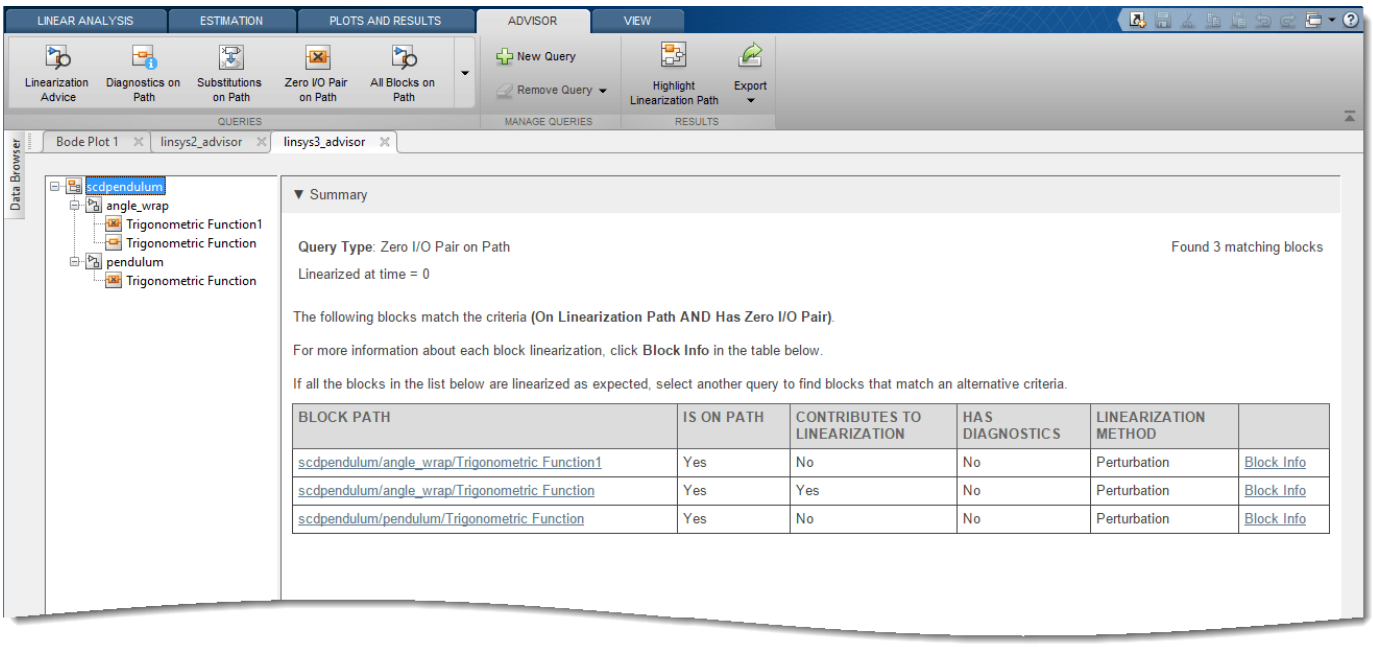
Run Prebuilt Advisor Queries

The Linearization Advisor provides a set of prebuilt queries for filtering block diagnostics. For example, the **Linearization Advice** query is the default query run when the advisor is first created and includes blocks on the path that:

- Have diagnostic messages regarding the block linearization.
- Linearized to zero.

- Have substituted linearizations.

To run a different prebuilt query, on the **Advisor** tab, in the **Queries** gallery, click the query. For example, click **Zero I/O Pair on Path**.



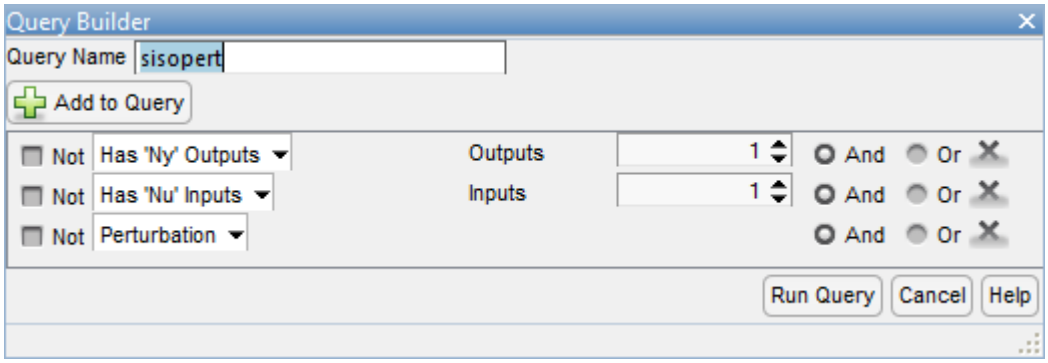
This query returns blocks with linearizations that have output channels that cannot be reached by any input channel, or input channels that have no influence on any output channels. For example, the second block in the table is a Trigonometric Function block configured as `atan2`. The first input of this block cannot reach the only output.

Create and Run Custom Queries

The Linearization Advisor also provides a Query Builder for creating custom queries. You can use these queries to find blocks in your model that match specific criteria. For example, to find all SISO blocks that are numerically perturbed, first open the Query Builder. To do so, on the **Advisor** tab, click **New Query**.

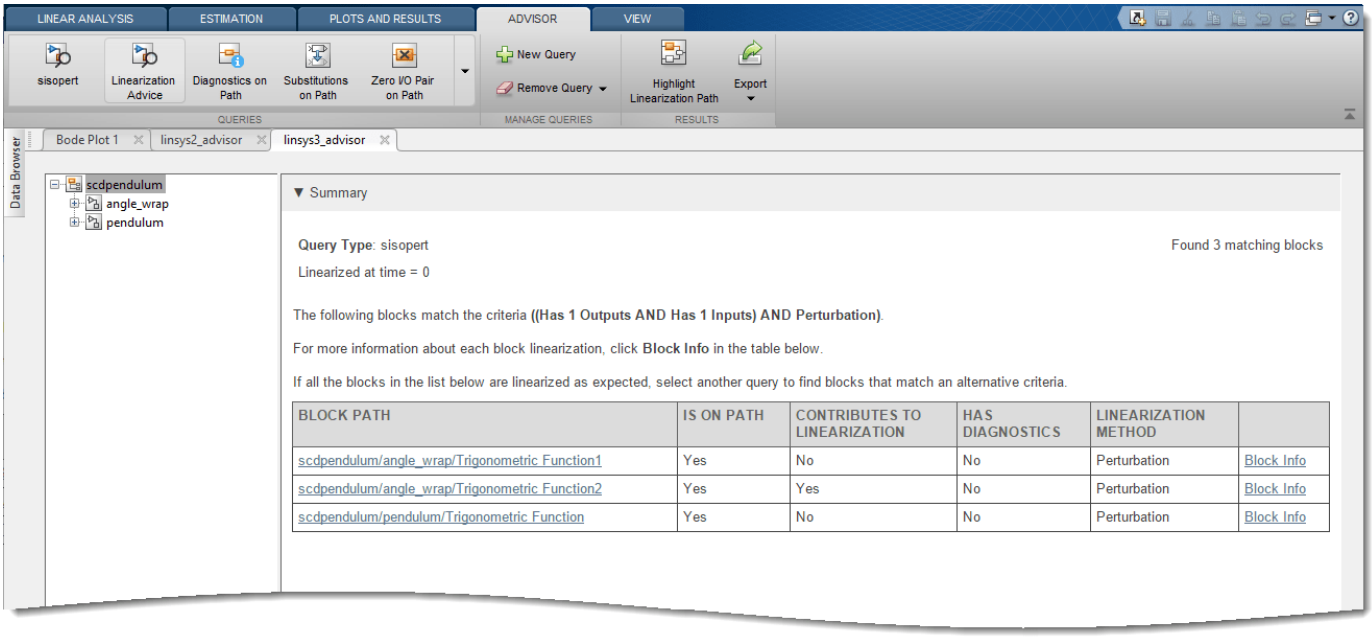
In the Query Builder dialog box:

1. Specify the **Query Name** as `sisopert`.
2. In the drop-down list, select `Has 'Ny' Outputs'`, and specify `1` in the **Outputs** box.
3. To add another component to the query, click **Add to Query**.
4. In the second drop-down list, select `Has 'Nu' Inputs'`, and specify `1` in the **Inputs** box.
5. Click **Add to Query**.
6. In the third drop-down list, select `Perturbation`.

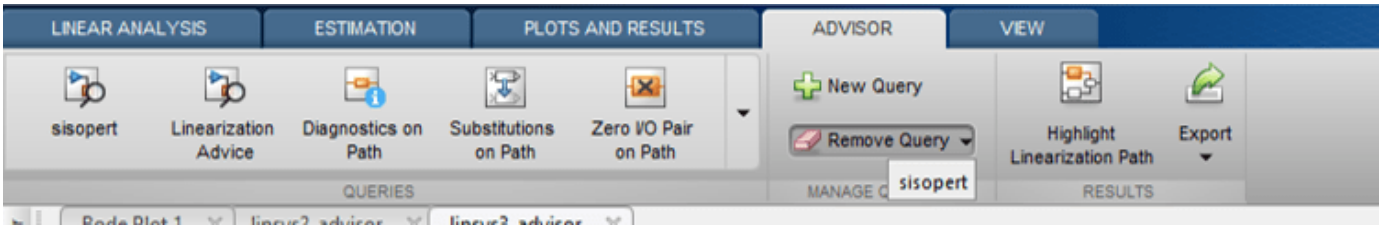


Click **Run Query**.

The **linsys3_advisor** document shows the blocks that match the specified query criteria, and the **sisopert** query is added to the **Queries** gallery.

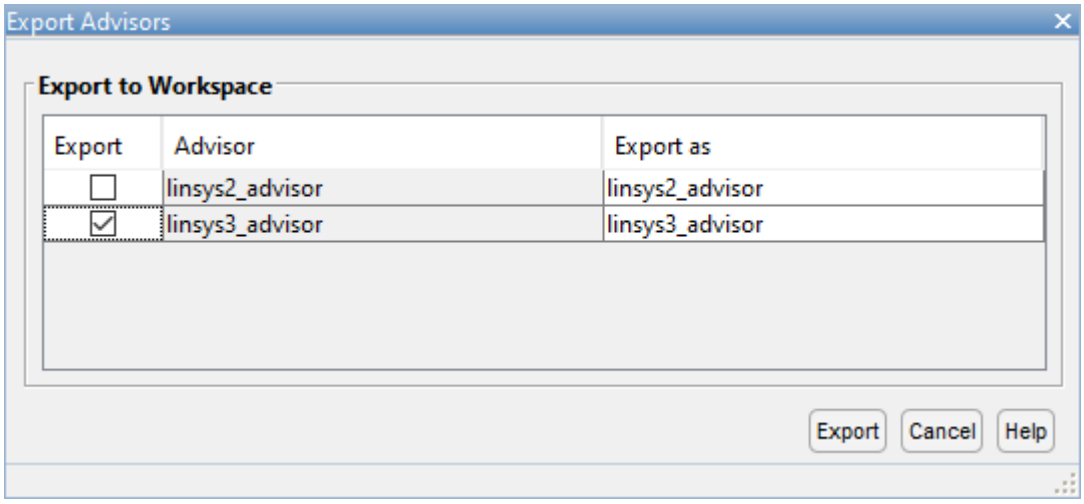


To remove the **sisopert** query, on the Advisor tab, click **Remove Query**, and select **sisopert**.



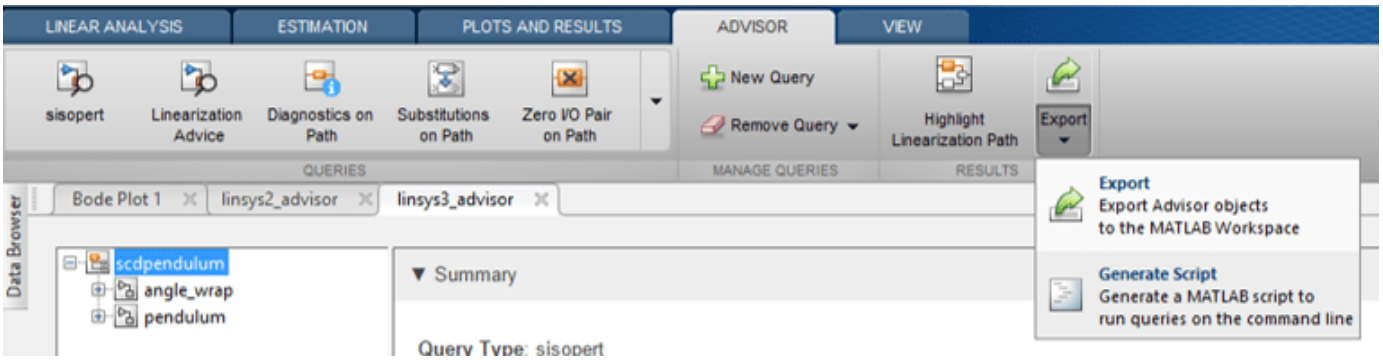
Export Advisor and Generate MATLAB Script

You can also debug model linearizations using the Linearization Advisor command-line functions. To export the advisor object to the MATLAB® workspace, click **Export**. Then, in the Export Advisors dialog box, select one or more advisors to export. For example, select **linsys3_advisor**.



Click **Export**.

Alternatively, you can generate a MATLAB script that automates the linearization, extraction of the advisor, generation of custom queries, and running of queries. To generate this script, click the **Export** split button, then select **Generate Script**.




```
bdclose(mdl)
```

See Also

Apps
Model Linearizer

More About

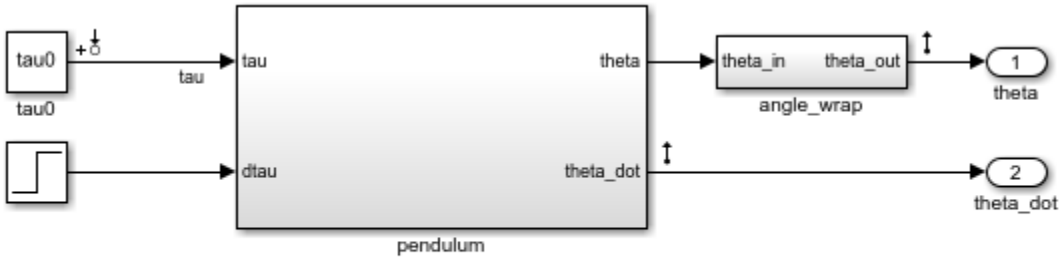
- “Identify and Fix Common Linearization Issues” on page 4-6
- “Troubleshoot Linearization Results at Command Line” on page 4-28
- “Find Blocks in Linearization Results Matching Specific Criteria” on page 4-37

Troubleshoot Linearization Results at Command Line

This example shows how to debug the linearization of a Simulink® model at the command line using a `LinearizationAdvisor` object. You can also troubleshoot linearization results interactively. For more information, see “Troubleshoot Linearization Results in Model Linearizer” on page 4-16.

Open the model.

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



Copyright 2017 The MathWorks, Inc.

The initial condition for the pendulum angle is `90` degrees counterclockwise from the upright unstable equilibrium of `0` degrees. The initial condition for the pendulum angular velocity is `0` deg/s. The nominal torque to maintain this state is `-49.05` N m. This configuration is saved as the model initial condition.

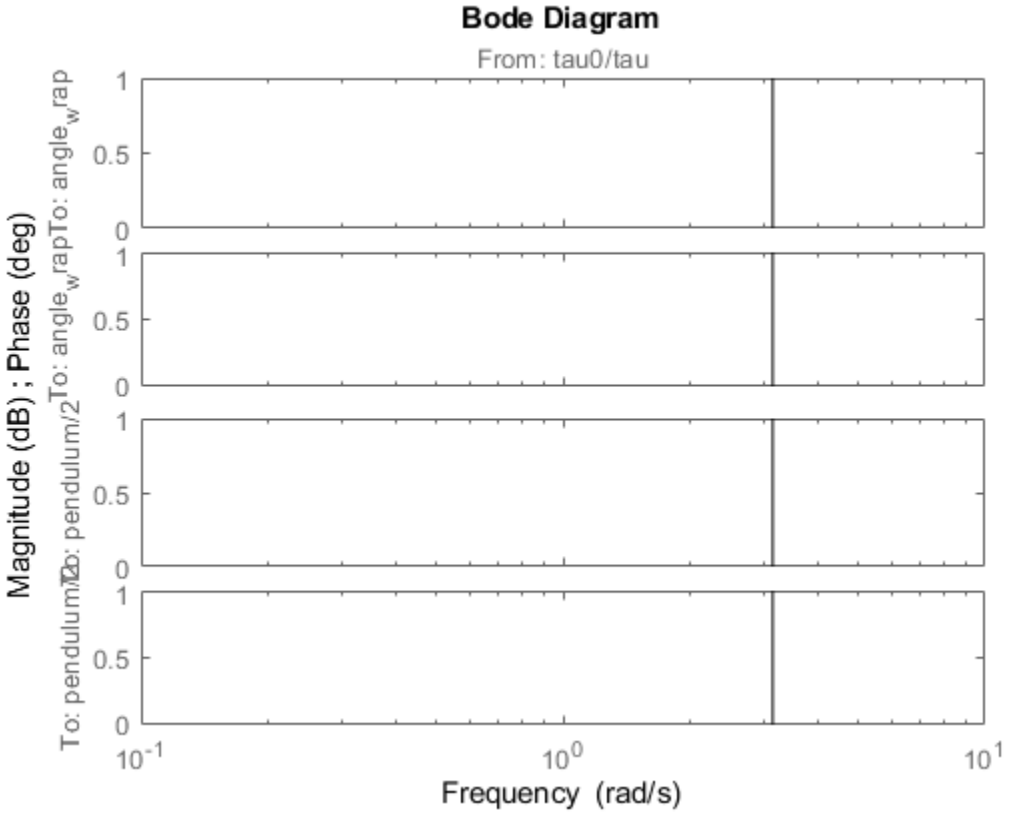
Linearize the Model

Linearize the model using the analysis points defined in the model and the model operating point.

```
io = getlinio(mdl);
linsys = linearize(mdl,io);
```

To check the linearization result, plot its Bode response.

```
bode(linsys)
```



The model linearized to zero such that the torque, `tau`, has no effect on the angle or angular velocity. To find the source of the zero linearization, you can use a `LinearizationAdvisor` object.

Linearize Model with Advisor Enabled

To collect diagnostic information during linearization and create an advisor for troubleshooting, first create a `linearizeOptions` option set, specifying the `StoreAdvisor` option as `true`.

```
opt = linearizeOptions('StoreAdvisor',true);
```

Linearize the Simulink model using this option set. Return the `info` output argument, which contains linearization diagnostic information in a `LinearizationAdvisor` object.

```
[linsys1,~,info] = linearize(mdl,io,opt);
```

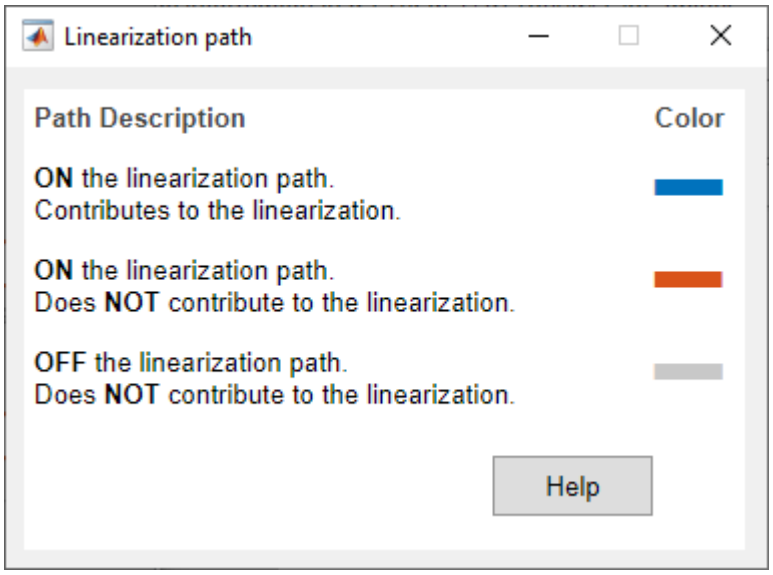
Extract the `LinearizationAdvisor` object.

```
advisor = info.Advisor;
```

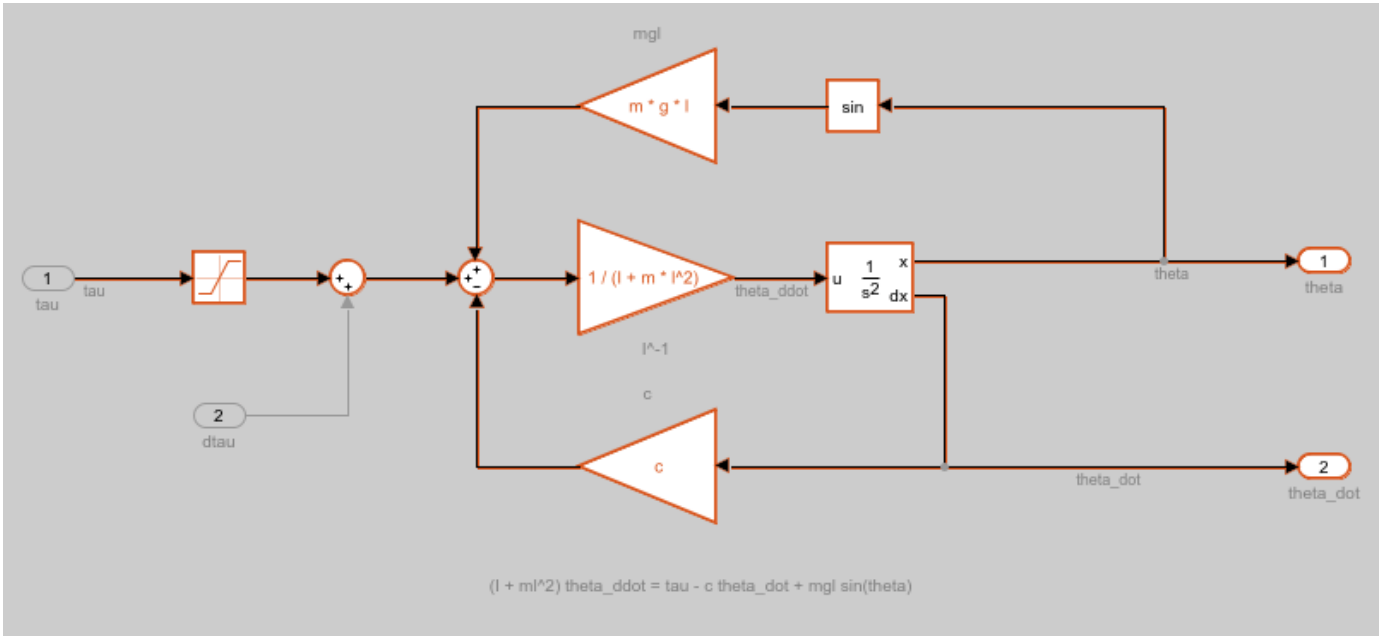
Highlight Linearization Path

To show the linearization path for the current linearization, use `highlight`.

```
highlight(advisor)
```



View the pendulum subsystem.



As shown in the Linearization path dialog box, the blocks highlighted in:

- Blue numerically influence the model linearization.
- Red are on the linearization path but do not influence the model linearization for the current operating point and block parameters.

Since the model linearized to zero, there are no blocks that contribute to the linearization.

Investigate Potentially Problematic Blocks

To obtain diagnostic information for blocks that may be problematic for linearization, use `advise`. This function returns a new `LinearizationAdvisor` object that contains information on blocks on the linearization path that satisfy at least one of the following criteria:

- Have diagnostic messages regarding their linearization
- Linearize to zero
- Have substituted linearizations

```
adv1 = advise(advisor);
```

View a summary of the diagnostic information for these blocks, use `getBlockInfo`.

```
getBlockInfo(adv1)
```

```
ans =

Linearization Diagnostics for the Blocks:

Block Info:
-----------
Index   BlockPath                                          Is On Path   Contributes To Lineariza
1.      scdpendulum/pendulum/Saturation                    Yes          No
2.      scdpendulum/angle_wrap/Trigonometric Function1     Yes          No
3.      scdpendulum/pendulum/Trigonometric Function        Yes          No
```

In this case, the advisor reports three potentially problematic blocks, a Saturation block and two Trigonometric Function blocks. When you run this example in MATLAB, the block paths display as hyperlinks. To go to one of these blocks in the model, click the corresponding block path hyperlink.

To view more information about a specific block linearization, use `getBlockInfo`. For information on the available diagnostics, see `BlockDiagnostic`.

For example, obtain the diagnostic information for the Saturation block.

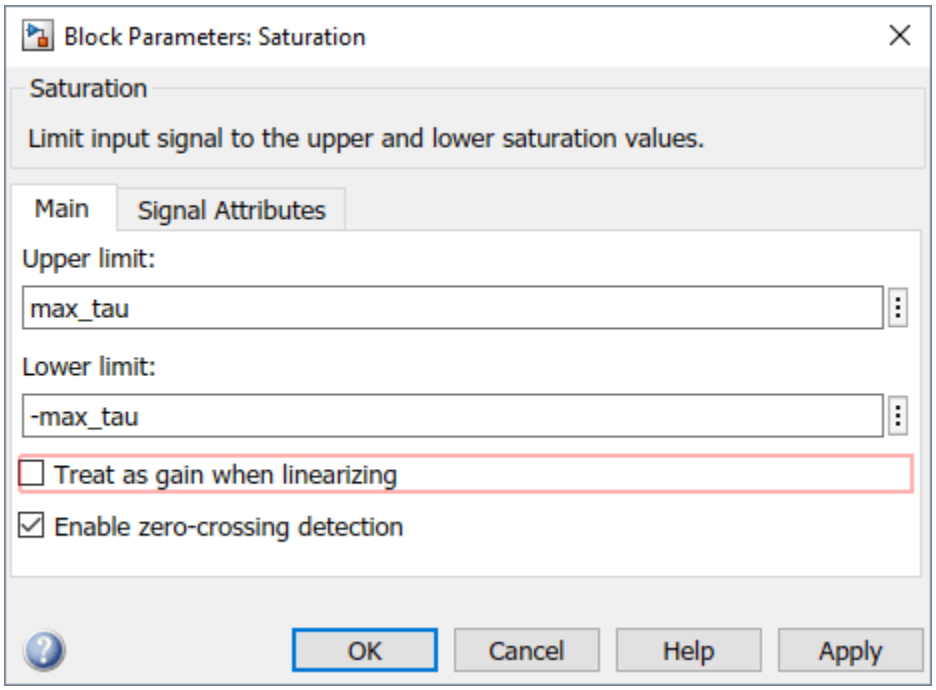
```
diagInfo = getBlockInfo(adv1,1)
```

```
diagInfo =

Linearization Diagnostics for scdpendulum/pendulum/Saturation with properties:

                     IsOnPath: 'Yes'
    ContributesToLinearization: 'No'
           LinearizationMethod: 'Exact'
                 Linearization: [1x1 ss]
              OperatingPoint: [1x1 linearize.advisor.BlockOperatingPoint]
```

This block has two diagnostic messages regarding its linearization result. The first message indicates that the block is linearized outside of its lower saturation limit of `-49`, since the input operating point is `-49.05`.

The message also indicates that the block can be linearized as a gain, which linearizes the block as `1` regardless of the input operating point.

When you run this example in MATLAB, the text **linearizing the block as a gain** displays as a hyperlink. To open the Block Parameters dialog box for the Saturation block, and highlight the option for linearizing the block as a gain, click this hyperlink.



Select **Treat as gain when linearizing**, and click **OK**.

Alternatively, you can set this parameter from the command line.

```
set_param('scdpendulum/pendulum/Saturation','LinearizeAsGain','on')
```

The second diagnostic message states that the linearization of this block causes the overall model to linearize to zero. View the linearization of this block.

```
diagInfo.Linearization
```

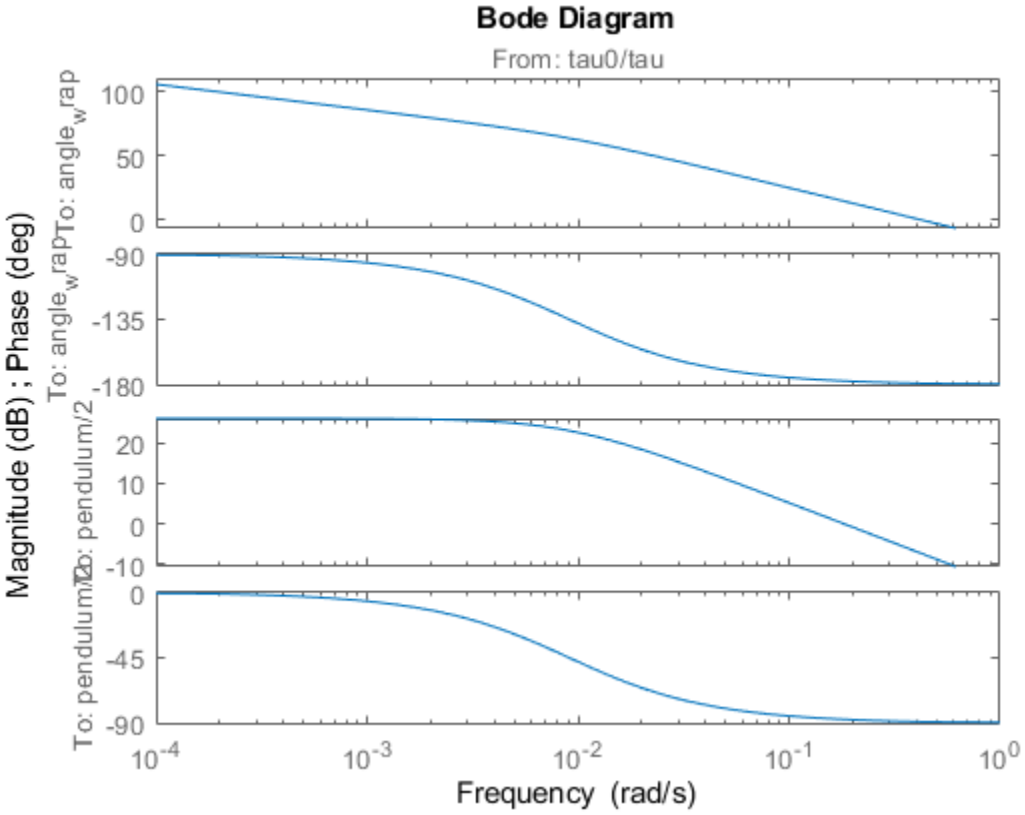
```
ans =

  D =
       u1
   y1   0

Name: Saturation
Static gain.
```

Since this block linearized to zero, modifying the block linearization by treating it as a gain is a good first step toward obtaining a nonzero model linearization.

Relinearize Model

To see the effect of treating the Saturation block as a gain, relinearize the model, and plot its Bode response.

```
[linsys2,~,info] = linearize(mdl,io,opt);
bode(linsys2)
```



The model linearization is now nonzero.

To check if any blocks are still potentially problematic for linearization, extract the advisor object, and use the `advise` function.

```
advisor2 = info.Advisor;
adv2 = advise(advisor2);
```

View the block diagnostic information.

```
getBlockInfo(adv2)
```

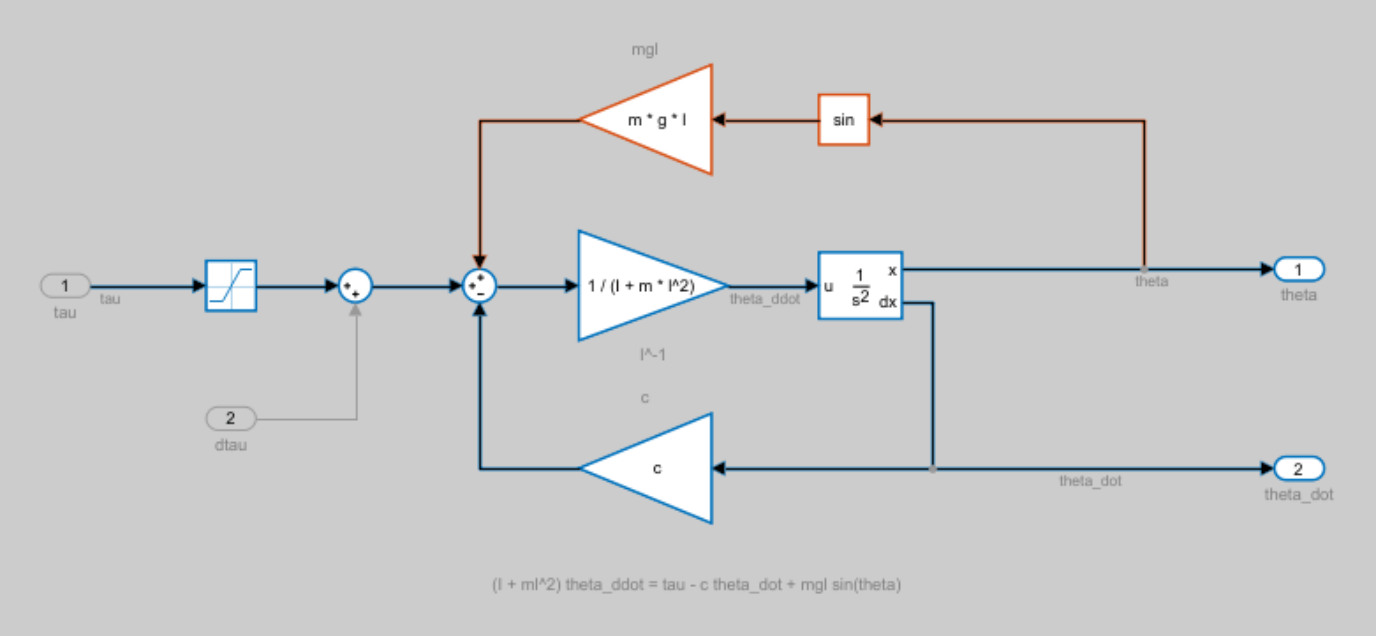
```
ans =

Linearization Diagnostics for the Blocks:

Block Info:
-----------
Index   BlockPath                                              Is On Path   Contributes To Linearizat
1.      scdpendulum/angle_wrap/Trigonometric Function1         Yes          No
2.      scdpendulum/pendulum/Trigonometric Function            Yes          No
```

The two Trigonometric Function blocks are still listed.

Highlight the linearization path for the updated linearization.

```
highlight(advisor2)
```

View the pendulum subsystem.



To understand why these blocks are not contributing to the linearization, view their corresponding block diagnostic information. For example, obtain the diagnostic information for the second Trigonometric Function block.

```
diagInfo = getBlockInfo(adv2,2)
```

```
diagInfo =

Linearization Diagnostics for scdpendulum/pendulum/Trigonometric Function with properties:

                     IsOnPath: 'Yes'
    ContributesToLinearization: 'No'
           LinearizationMethod: 'Perturbation'
                 Linearization: [1x1 ss]
               OperatingPoint: [1x1 linearize.advisor.BlockOperatingPoint]
```

View the linearization of this block.

```
diagInfo.Linearization
```

```
ans =

  D =
       u1
   y1   0
```

```
Name: Trigonometric Function
Static gain.
```

The block linearized to zero. To see if this result is expected for the current operating condition of the block, check its operating point.

```
diagInfo.OperatingPoint
```

```
ans =

Block Operating Point for scdpendulum/pendulum/Trigonometric Function


Inputs:
-------
Port   u
1      1.5708
```

The input operating point of the block is $\pi/2 = 1.5708$.

You can find the linearization of the block analytically by taking the first derivative of the sin function with respect to the input.

$$\frac{\partial}{\partial u}\sin(u) = \cos(u)$$

Therefore, when evaluated at $u = \pi/2$ the linearization of the block is zero. The source of the input is the first output of the second-order integrator, which is dependent upon the state `theta`. Therefore, this block linearizes to zero if $\theta = \pi/2 + k\pi$, where k is an integer. The same condition applies for the other Trigonometric Function block in the angle_wrap subsystem. If these blocks are not expected to linearize to zero, you can modify the operating point state `theta`, and relinearize the model.

Create and Run Custom Queries

The Linearization Advisor also provides objects and functions for creating custom queries. Using these queries, you can find blocks in your model that match specific criteria. For example, to find all SISO blocks that are linearized using numerical perturbation, first create query objects for each search criterion:

- Has one input
- Has one output
- Is numerically perturbed

```
qIn = linqueryHasInputs(1);
qOut = linqueryHasOutputs(1);
qPerturb = linqueryIsNumericallyPerturbed;
```

Create a `CompoundQuery` object by combining these query objects using logical operators.

```
sisopert = qIn & qOut & qPerturb;
```

Search the block diagnostics in `advisor2` for blocks matching these criteria.

```
sisopertBlocks = find(advisor2,sisopert)
```

```
sisopertBlocks =

  LinearizationAdvisor with properties:

              Model: 'scdpendulum'
     OperatingPoint: [1x1 opcond.OperatingPoint]
    BlockDiagnostics: [1x3 linearize.advisor.BlockDiagnostic]
           QueryType: '((Has 1 Inputs & Has 1 Outputs) & Perturbation)'
```

There are three SISO blocks in the model that are linearized using numerical perturbation.

For more information on using custom queries, see “Find Blocks in Linearization Results Matching Specific Criteria” on page 4-37.

```
bdclose(mdl)
```

See Also

Functions
`advise` | `find`

More About

- “Identify and Fix Common Linearization Issues” on page 4-6
- “Troubleshoot Linearization Results in Model Linearizer” on page 4-16
- “Find Blocks in Linearization Results Matching Specific Criteria” on page 4-37

Find Blocks in Linearization Results Matching Specific Criteria

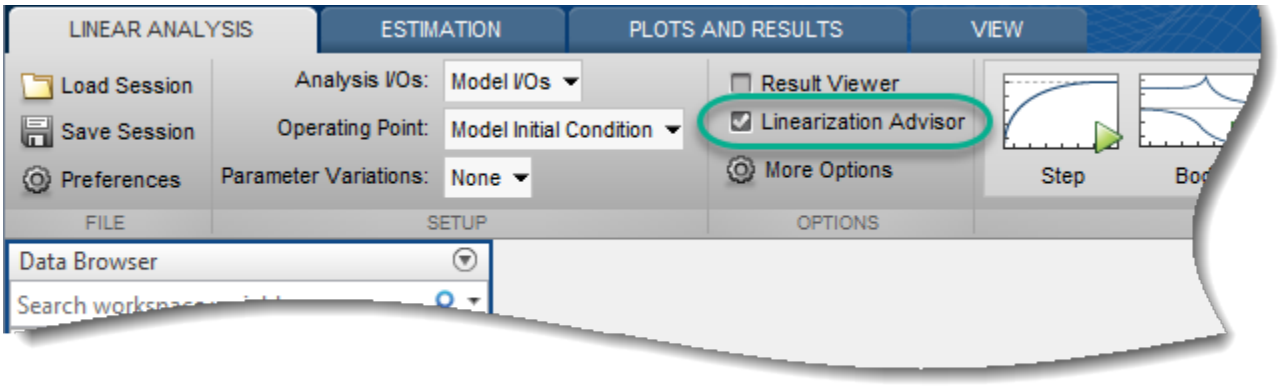
When you linearize a Simulink model, you can find blocks in your linearization result that match specific criteria using the Linearization Advisor. You can specify search criteria to find blocks that can:

- Potentially cause linearization issues in your model, if your model does not linearize as expected. For more information on identifying and fixing linearization issues using the Linearization Advisor, see “Identify and Fix Common Linearization Issues” on page 4-6.
- Help you gain insight into your model linearization, even if the model has linearized as expected.

You can also query the Linearization Advisor at the command line using the `find` function. For an example, see “Troubleshoot Linearization Results at Command Line” on page 4-28.

Searching the linearization results requires linearization diagnostic information. To collect this information, you must enable the Linearization Advisor before linearizing your model.

To enable the Linearization Advisor, in the **Model Linearizer**, on the **Linear Analysis** tab, select **Linearization Advisor**.



When you select this option and linearize your model, the software opens an **Advisor** tab for troubleshooting your linearization results. You can then find blocks of interest in the linearization results by running queries with the Linearization Advisor.

After finding blocks of interest, you can examine the individual block linearizations using the linearization diagnostic information. For more information, see “Block Linearization Troubleshooting” on page 4-42.

Run Built-In Queries

The Linearization Advisor provides a set of built-in queries for searching your linearization results. These queries are useful for finding blocks that are potentially causing linearization issues. To run one of these queries, on the **Advisor** tab, in the **Queries** section, click the query.

Built-In Query	Find Blocks That...
Linearization Advice	Are potentially problematic for linearization. This query is performed by default when the **Advisor** tab opens.

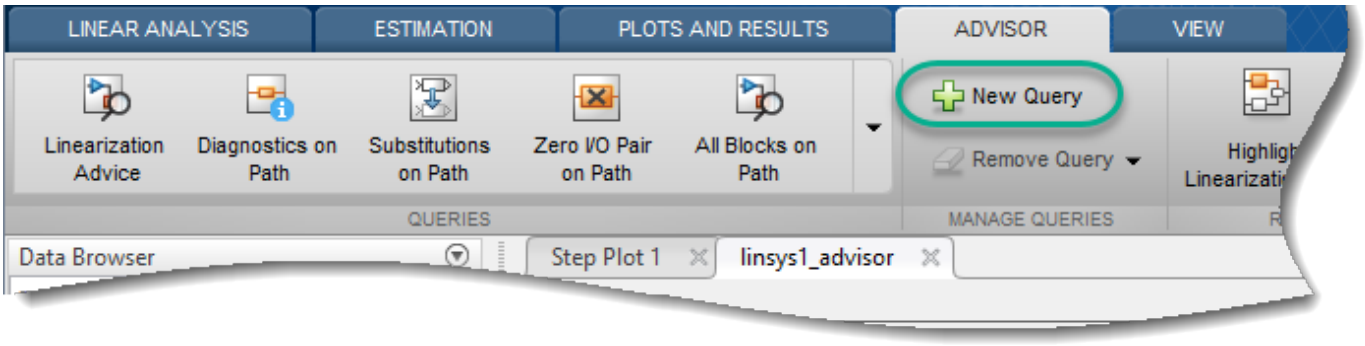
Built-In Query	Find Blocks That...
Diagnostics on Path	Are on the linearization path and that have diagnostic messages regarding their linearization. This query is a subset of the `Linearization Advice` query.
Substitutions on Path	Are on the linearization path and have a custom block linearization specified. This query is a subset of the `Linearization Advice` query.
Zero I/O Pair on Path	Are on the linearization path and have at least one input/output pair that linearizes to zero.
All Blocks on Path	Are on the linearization path; that is, blocks where at least one linearization input is connected to at least one linearization output through the block.

Create and Run Queries

The linearization advisor also provides a set of simple queries for searching your model. You can run these queries on their own or use them to create compound queries.

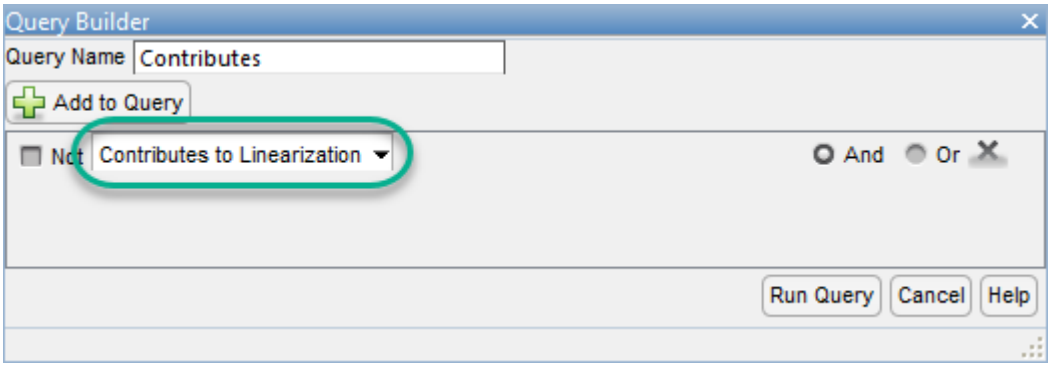
Simple Query	Find Blocks That...
`All Blocks`	Are in the linearized model.
`Linearized to Zero`	Linearize to zero.
`Block Substituted`	Have a custom block linearization specified.
`On Linearization Path`	Are on the linearization path.
`Contributes to Linearization`	Numerically contribute to the model linearization result.
`Exact`	Are linearized using their defined exact linearization.
`Perturbation`	Are linearized using numerical perturbation.
`Has Diagnostics`	Have diagnostic messages regarding their linearization.
`'BlockType' Blocks`	Are of a specified type.
`Has 'Nu' Inputs`	Have a specified number of inputs.
`Has 'Nx' States`	Have a specified number of states.
`Has 'Ny' Outputs`	Have a specified number of outputs.
`Has 'Ts' Sample Time`	Have a specified sample time.
`Has Zero I/O Pair`	Have at least one input/output pair that linearizes to zero.

To run a simple query, in the **Model Linearizer**, on the **Advisor** tab, click **New Query**.



In the Query Builder dialog box, configure the query. For example, create a query for finding all blocks that numerically contribute to the linearization result.

1. In the **Query Name** field, specify the name for the query as `Contributes`.
2. In the drop-down list, select `Contributes to Linearization`.

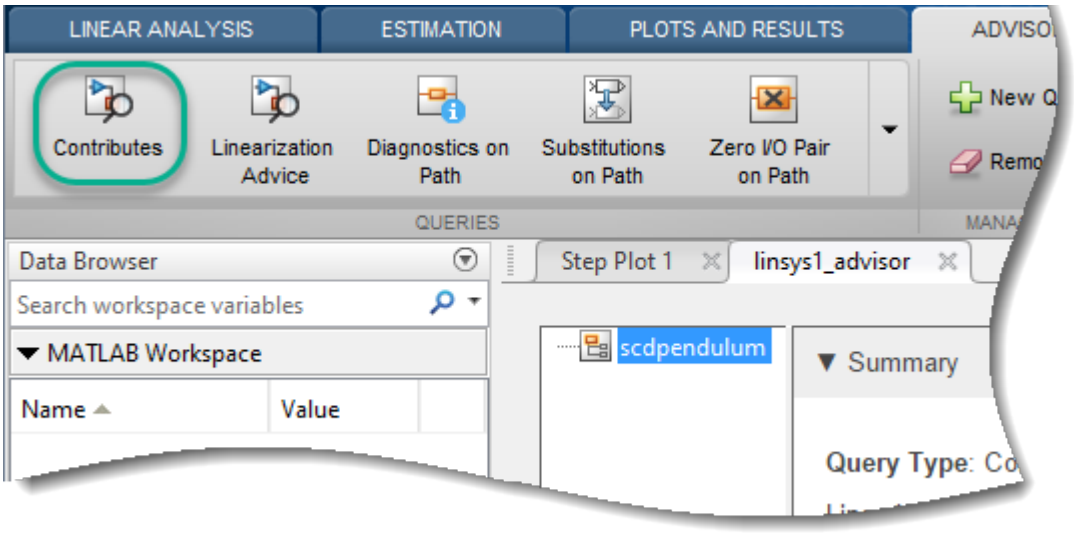


3. If you select any of the following queries, specify the corresponding search parameter.

Query	Search Parameter
`'BlockType' Blocks`	**Block Type** — This parameter corresponds to the `blocktype` property of the block. For more information, see `linqueryIsBlockType`.
`Has 'Nu' Inputs`	**Inputs** — Specify a positive integer.
`Has 'Nx' States`	**States** — Specify a positive integer.
`Has 'Ny' Outputs`	**Outputs** — Specify a positive integer.
`Has 'Ts' Sample Time`	**Sample Time** — Specify a nonzero scalar. To find continuous-time blocks, specify `0`.

4. To create and run the query, click **Run Query**. The software runs the query and, on the **Advisor** tab, displays the list of blocks that contribute to the model linearization.

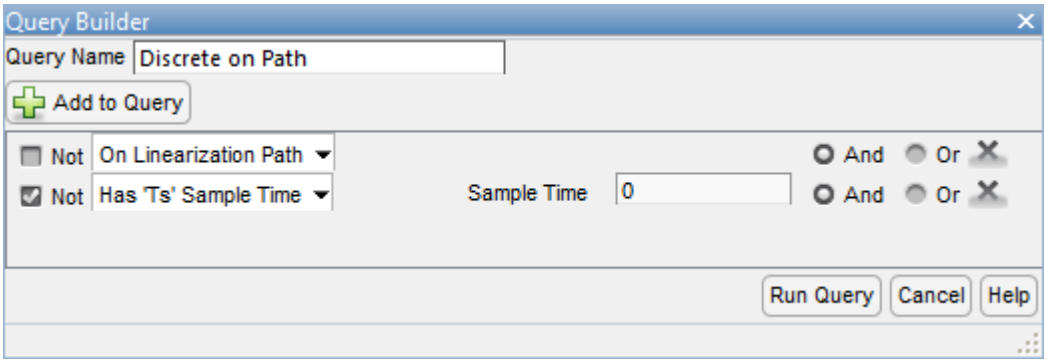
   The query is added to the **Queries** section.



You can also create compound queries by logically combining existing queries using **And**, **Or**, and **Not** logical operations. You can create a compound query using simple queries, built-in queries, or other compound queries.

To create a compound query, in the Query Builder dialog box, configure the query using multiple search criteria. For example, create a query to find all discrete-time blocks that are on the linearization path.

1. In the **Query Name** field, specify the name for the query as `Discrete on Path`.
2. To find blocks on the linearization path, in the drop-down list, select `On Linearization Path`.
3. To add another search criteria, click **Add to Query**. The software adds a second row to the search criteria. By default, the search criteria are combined using an **And** operation.
4. To find discrete-time blocks, first add a search criteria to find continuous-time blocks. In the second row, in the drop-down list, select `Has 'Ts' Sample Time`. Keep the default **Sample Time** of `0`.
5. To find discrete-time blocks, in the second row, select **Not**.



6. Click **Run Query**.

Each time you create a custom query, the software adds it to the drop-down list of search criteria in the Query Builder dialog box. You can then use your custom queries to create more complex queries. For example, to find discrete-time blocks on the linearization path that are linearized using numerical

perturbation, create a query that combines the `Discrete on Path` custom query with the `Perturbed` simple query using an **And** operation.

See Also

Apps
Model Linearizer

Functions
`find`

More About

- “Identify and Fix Common Linearization Issues” on page 4-6
- “Troubleshoot Linearization Results in Model Linearizer” on page 4-16
- “Troubleshoot Linearization Results at Command Line” on page 4-28

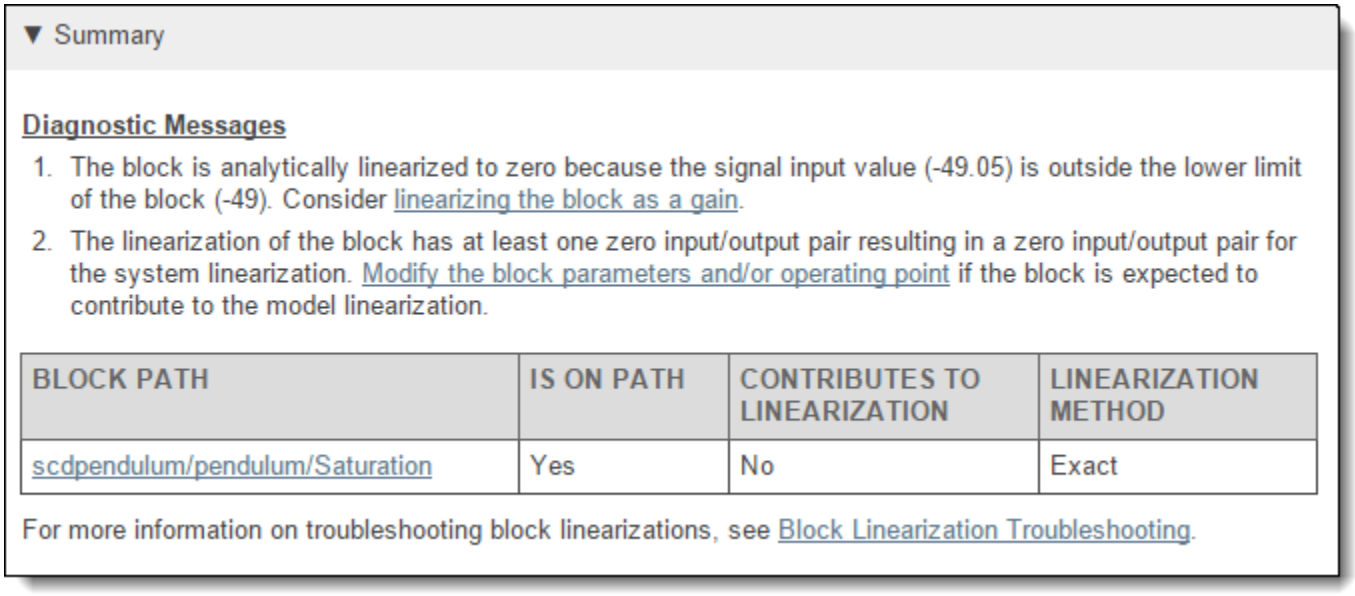
Block Linearization Troubleshooting

Once you identify blocks of interest in the linearization results for your Simulink model by querying the Linearization Advisor, you can troubleshoot the individual block linearizations. For more information on querying the Linearization Advisor and viewing block diagnostic information, see “Identify and Fix Common Linearization Issues” on page 4-6.

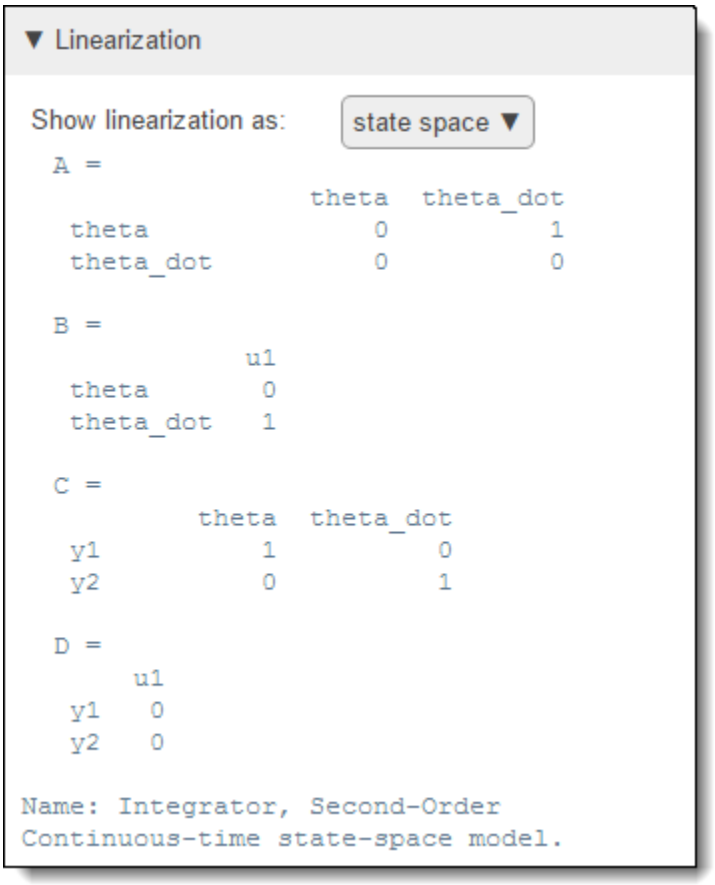
You can also troubleshoot individual block linearizations at the command line using a `BlockDiagnostic` object. For an example, see “Troubleshoot Linearization Results at Command Line” on page 4-28.

In the **Model Linearizer**, on the **Advisor** tab, the detailed diagnostic information for a block linearization shows:

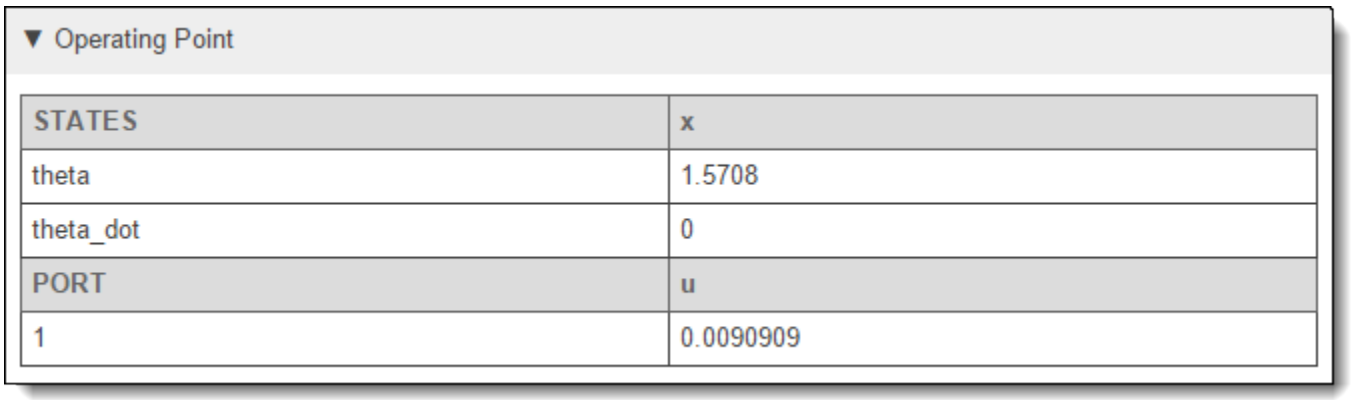
- A diagnostic summary, showing any corresponding diagnostic messages, and a linearization summary table.



- The block linearization value.



- The block operating point; the state and input values for which the block is linearized.



You can diagnose potential linearization issues using this information.

Diagnostic Messages

Linearization diagnostic messages indicate blocks with properties or linearizations that correspond to common linearization problems. Fixing linearization issues identified in diagnostic messages is a good first step when troubleshooting your linearization.

Some block configurations that can generate diagnostic messages include:

- Blocks with no predefined exact linearization and with non-floating-point signals or states. Such blocks linearize to zero and generate diagnostic messages.
- Discontinuous blocks linearized at an operating point near a discontinuity. If such blocks are not treated as a gain during linearization, the software generates diagnostic messages regarding their linearization.
- Blocks with least one input/output pair that linearizes to zero and that causes a zero input/output pair in the overall model linearization. A linearization has a zero input/output pair when a change in an input signal value does not produce a corresponding change in an output value.
- Blocks that do not support linearization because they do not have a predefined exact linearization and do not support numerical perturbation.

Some diagnostic messages propose solutions to their corresponding linearization issues. For example, when an input signal is outside the saturation limits of a Saturation block, the diagnostic message proposes treating the block as a gain during linearization.

Linearization Summary

The linearization summary table displays the following properties of the block linearization:

- **Block Path** — Location of the block in the Simulink model. To highlight the block in the model, click the block path.
- **Is On Path** — Flag indicating whether the block is on the linearization path, that is, at least one linearization input is connected to at least one linearization output through the block. If you expect a block to be on the linearization path and it is not on the path, check the analysis point configuration in your model. Incorrectly placed linearization I/Os or loop openings can exclude blocks from the linearization path. Similarly, placing incorrect analysis points can unexpectedly add blocks to the linearization path.
- **Contributes to Linearization** — Flag indicating whether the block numerically contributes to the overall model linearization. If a block unexpectedly does not contribute to the linearization result, investigate the linearization of the block and other blocks in the same branch of the linearization path. For example, if an adjacent block on the linearization path linearizes to zero, an otherwise correctly linearized block can be excluded from the linearization result.
- **Linearization method** — The method used to linearize the model, specified as one of the following:
  - **Exact** — The block linearization is computed using the defined analytic Jacobian of the block.
  - **Perturbation** — The block does not have an analytic Jacobian. Instead, the block is linearized using numerical perturbation of its inputs and states. Some numerically perturbed blocks, such as those with discontinuities or non-floating-point input signals can linearize to zero.
  - **Block Substituted** — The block linearization is specified using a custom block linearization. Consider checking that the specified block linearization is correct for your application. For more information, see “Specify Linear System for Block Linearization Using MATLAB Expression” on page 2-123 and “Specify D-Matrix System for Block Linearization Using Function” on page 2-124.
  - **Simscape Network** — The block diagnostics correspond to a Simscape network in your model. For more information on linearizing and troubleshooting Simscape networks, see “Linearize Simscape Networks” on page 2-160.

- **Not Supported** — The block does not have an analytic Jacobian and does not support numerical perturbation. Specify the linearization for this block using a custom linearization. For more information, see “Specify Linear System for Block Linearization Using MATLAB Expression” on page 2-123 and “Specify D-Matrix System for Block Linearization Using Function” on page 2-124.

Block Linearization

To verify whether a block linearized as expected, check the block linearization equations. By default the software displays the linearization in state-space format. In the **Show linearization as** drop-down list, you can select a different display format.

To diagnose the cause of an unexpected block linearization, such as a block that linearizes to zero, consider:

- Any corresponding diagnostic messages. These messages can highlight common causes of incorrect linearizations and propose potential solutions.
- The block operating point. For example, if the input to a saturation block is outside the saturation limits of the block, the block linearizes to zero.
- The block parameters. For example, if a block is configured to use non-floating-point inputs or states and has no predefined exact linearization, it linearizes to zero.

Block Operating Point

If the block does not linearize as expected, check the operating point. The operating point at which the block is linearized consists of input and state values. If the operating point for the block is incorrect, check whether the overall model operating point is correct. For more information, see “Check Operating Point” on page 4-4.

If an input signal value in the block operating point is incorrect, investigate the linearization of upstream blocks from that signal. For example, consider a Product block with two inputs. The operating point of this block consists of the two input signal values. If either input value is zero, the path from the other input to the output linearizes to zero.

If you expect the Product block to contribute to the linearization result for the operating point at which you linearized the model, check the linearization for the block that generates the zero input signal. For complex models, the cause of the incorrect input signal can be more than one block upstream.

Common Problematic Blocks

Some Simulink blocks have properties that cause them to linearize poorly. Often, such blocks either linearize to zero or have linearization diagnostic messages associated with them. Therefore, the Linearization Advisor identifies them as potentially problematic blocks when the **Advisor** tab first opens.

The following table shows some blocks that commonly cause linearization issues and proposes potential fixes for each block. All these blocks have corresponding diagnostic messages.

Block Type	Linearization Issue	Possible Fix
Blocks that do not support linearization	Some blocks are implemented without defined analytic Jacobians and do not support numerical perturbation.	Specify a custom block linearization. For examples, see "Specify Linear System for Block Linearization Using MATLAB Expression" on page 2-123 and "Specify D-Matrix System for Block Linearization Using Function" on page 2-124.
Blocks with discontinuities	Blocks with discontinuities typically have poor linearization results when the operating point is near the discontinuity.	• Treat the block as a gain of 1 during linearization. To do so, select the **Treat as gain when linearizing** block parameter. • Specify a custom block linearization. For examples, see "Specify Linear System for Block Linearization Using MATLAB Expression" on page 2-123 and "Specify D-Matrix System for Block Linearization Using Function" on page 2-124.
Event-Based Subsystems (triggered subsystems)	Blocks within event-based subsystems linearize to zero because such subsystems do not trigger during linearization.	When possible, specify a custom event-based subsystem linearization as a lumped average model or periodic function call subsystem. For more information, see "Linearize Event-Based Subsystems (Externally Scheduled Subsystems)" on page 2-152.
Blocks with non-floating-point signals	Blocks that have non-floating-point input signals or states and do not have defined analytic Jacobians linearize to zero.	Convert the non-floating-point data types to either double precision or single precision.. For more information, see "Linearize Blocks with Non-Floating-Point Signals or States" on page 2-150.

Block Type	Linearization Issue	Possible Fix
Blocks that linearize using numerical perturbation rather than defined analytic Jacobians	Blocks that are located near discontinuous regions, such as S-Functions, MATLAB function blocks, or lookup tables, are sensitive to numerical perturbation levels. If the perturbation level is too small, the block linearizes to zero.	Change the numerical perturbation level of the block. For more information, see “Change Perturbation Level of Blocks Perturbed During Linearization” on page 2-148.

See Also

More About

- “Linearization Troubleshooting Overview” on page 4-2
- “Identify and Fix Common Linearization Issues” on page 4-6

Speed Up Linearization of Complex Models

Factors That Impact Linearization Performance

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

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

Blocks with Complex Initialization Functions

Use the MATLAB Profiler to identify complex bottlenecks in block initialization functions.

In the MATLAB Profiler, run the command:

```
set_param(modelname,'SimulationCommand','update')
```

Disabling the Linearization Advisor in the Model Linearizer

You can speed up the linearization of large models by disabling the Linearization Advisor in the **Model Linearizer** app.

The Linearization Advisor stores diagnostic information, including linearization values of individual blocks, which can impact linearization performance.

To disable the Linearization Advisor, in **Model Linearizer**, on the **Linear Analysis** tab, clear **Linearization Advisor**.

Tip Alternatively, you can disable the Linearization Advisor by default when the **Model Linearizer** app opens. To do so, in the MATLAB preferences dialog box, click **Simulink Control Design**. Then, clear the **Launch Linearization Advisor for exact linearizations in the Model Linearizer** option. This global setting persists from session to session until you change this option.

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.

For more information, see “More Efficient Batch Linearization Varying Parameters” on page 3-65.

See Also

Frequency Response Estimation

- “Frequency Response Estimation Basics” on page 5-2
- “Estimate Frequency Response Using Model Linearizer” on page 5-6
- “Estimate Frequency Response with Linearization-Based Input Using Model Linearizer” on page 5-10
- “Estimate Frequency Response at the Command Line” on page 5-14
- “Analyze Estimated Frequency Response” on page 5-18
- “Estimation Input Signals” on page 5-24
- “Sinestream Input Signals” on page 5-29
- “Chirp Input Signals” on page 5-33
- “PRBS Input Signals” on page 5-36
- “Modify Estimation Input Signals” on page 5-40
- “Troubleshooting Frequency Response Estimation” on page 5-43
- “Effects of Time-Varying Source Blocks on Frequency Response Estimation” on page 5-53
- “Disable Noise Sources During Frequency Response Estimation” on page 5-60
- “Estimate Frequency Response Models with Noise Using Signal Processing Toolbox” on page 5-63
- “Estimate Frequency Response Models with Noise Using System Identification Toolbox” on page 5-65
- “Generate MATLAB Code for Repeated or Batch Frequency Response Estimation” on page 5-67
- “Managing Estimation Speed and Memory” on page 5-68
- “Frequency Response Estimation Using Simulation-Based Techniques” on page 5-74
- “Validate Linearization in Frequency Domain at Command Line” on page 5-80
- “Describing Function Analysis of Nonlinear Simulink Models” on page 5-84
- “Speed Up Frequency Response Estimation Using Parallel Computing” on page 5-89
- “Frequency Response Estimation for Power Electronics Model Using Pseudorandom Binary Signal” on page 5-94
- “Frequency Response Estimation in Model Linearizer Using Pseudorandom Binary Sequence” on page 5-101
- “Frequency Response Estimation for Permanent Magnet Synchronous Motor Model” on page 5-113
- “Frequency Response Estimation to Measure Input Admittance of Boost Converter” on page 5-124

Frequency Response Estimation Basics

Frequency response describes the steady-state response of a system to sinusoidal inputs. Simulink Control Design lets you estimate the frequency response of a model or perform online estimation of a physical plant. The result is a frequency response model, stored as an `frd` model object. Applications of frequency response models include:

- Validate 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.
- Analyze linear model dynamics or design a controller for the plant represented by the estimated frequency response.
- Estimate a parametric model using System Identification Toolbox software.

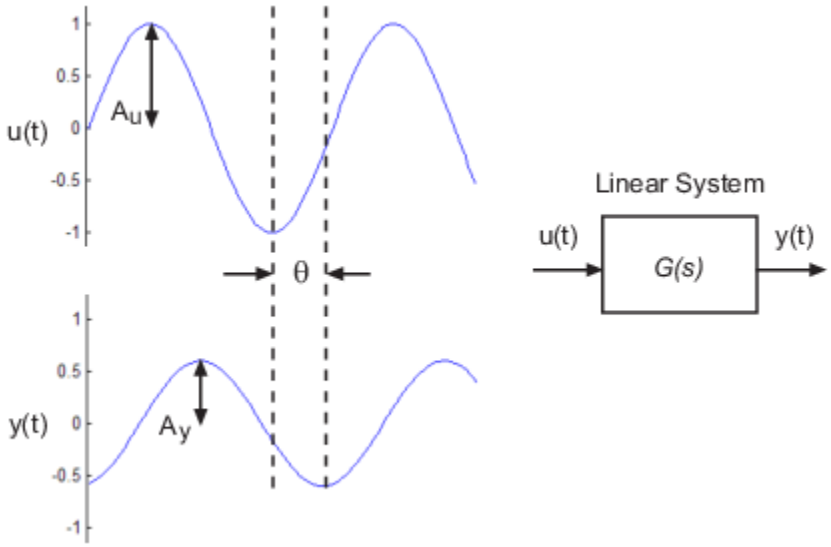
Frequency Response Models

Consider applying a sinusoidal input of frequency ω to a linear system:

$$u(t) = A_u \sin \omega t \, .$$

The result is an output that is also a sinusoid with the same frequency, but with a different amplitude and phase θ:

$$y(t) = A_y \sin(\omega t + \theta) \, .$$



The frequency response for a stable system describes the amplitude change and phase shift as a function of frequency. If $Y(s)$ and $U(s)$ are the Laplace transforms of $y(t)$ and $u(t)$, respectively, then $G(s)$ is:

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

where

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

$$\theta = \angle\frac{Y(j\omega)}{X(j\omega)} = \tan^{-1}\left(\frac{\mathrm{Im}[G(j\omega)]}{\mathrm{Re}[G(j\omega)]}\right).$$

The `frd` model that you get from frequency response estimation contains $G(s)$ evaluated at particular frequencies. Although your Simulink is usually nonlinear, you typically perform estimation at a steady-state operating point. If the applied perturbation is small, the resulting `frd` model is an approximation of the linearized response at that nominal operating point.

Offline and Online Estimation

Simulink Control Design lets you:

- Estimate the frequency response of a system modeled in Simulink, without modifying the model. This approach is sometimes called offline frequency response estimation.
- Estimate the frequency response of a physical plant during real-time operation. This approach is called online frequency response estimation.

The following table summarizes some of the differences between offline and online estimation and the tools you use to perform them.

Goal	Tool	More Information
Estimate frequency response of a system modeled in Simulink without modifying the model	• Interactive workflow — **Model Linearizer** • Command-line workflow — `frestimate` command	• "Estimate Frequency Response Using Model Linearizer" on page 5-6 • "Estimate Frequency Response at the Command Line" on page 5-14
Deploy frequency response estimation algorithm for real-time estimation of a physical plant	Frequency Response Estimator block	"Deploy Frequency Response Estimation Algorithm for Real-Time Use" on page 6-9
Perform online estimation of a plant modeled in Simulink, such as to validate estimation parameters before deployment	Frequency Response Estimator block	"Online Estimation Using Plant Modeled in Simulink" on page 6-5

Basic Estimation Workflow

For offline estimation, the basic frequency response estimation workflow includes the following steps:

1. Specify the portion of the model you want to estimate. You do so by configuring linearization analysis points that specify the inputs and outputs for estimation.
2. Specify an operating point for estimation. Generally, you perform estimation at a steady-state operating point. You can find such an operating point by trimming the model.
3. Create an input signal for estimation. The software injects this signal at the input you specify and measures the response at the output.

4 Perform the estimation and examine the results.

For examples illustrating this workflow, see:

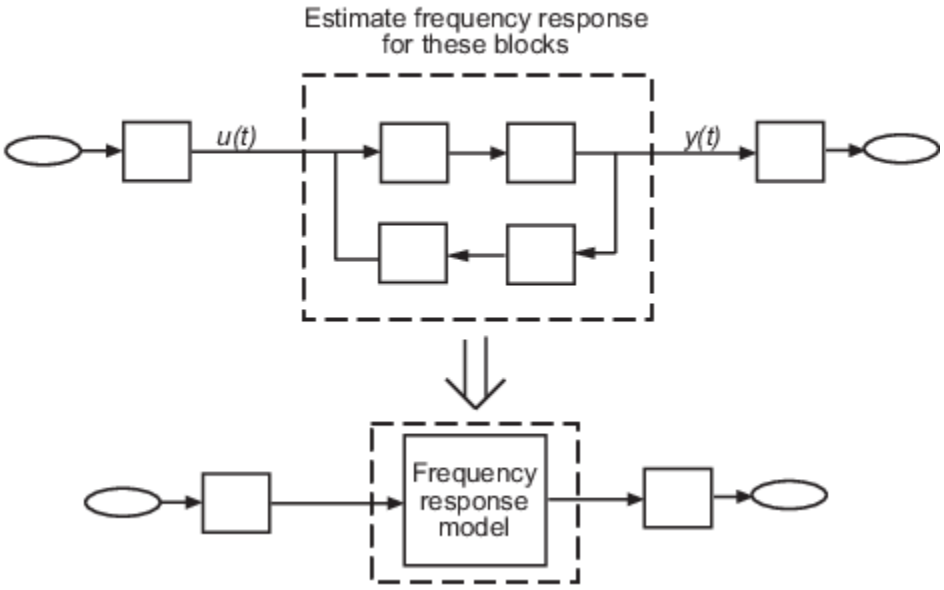
- “Estimate Frequency Response Using Model Linearizer” on page 5-6
- “Estimate Frequency Response at the Command Line” on page 5-14

For information about the online estimation workflow, see “Online Frequency Response Estimation Basics” on page 6-2.

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 and triggered subsystems.



Disable the following types of blocks before estimation:

- Blocks that simulate random disturbances (noise). For alternatives ways to model systems with noise, see “Estimate Frequency Response Models with Noise Using Signal Processing Toolbox” on page 5-63.
- Source blocks that generate time-varying outputs that interfere with the estimation. See “Effects of Time-Varying Source Blocks on Frequency Response Estimation” on page 5-53.

See Also

`frestimate` | **Model Linearizer** | Frequency Response Estimator

More About

- “Estimate Frequency Response Using Model Linearizer” on page 5-6

- “Estimate Frequency Response at the Command Line” on page 5-14
- “Estimation Input Signals” on page 5-24
- “Validate Linearization In Frequency Domain Using Model Linearizer” on page 2-108
- “Estimate Frequency Response Models with Noise Using System Identification Toolbox” on page 5-65
- “Online Frequency Response Estimation Basics” on page 6-2

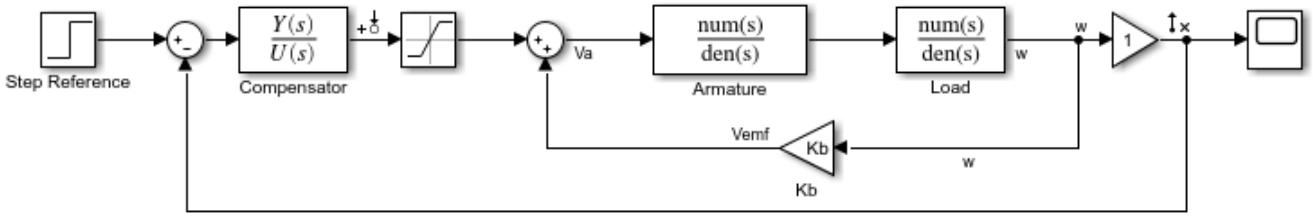
Estimate Frequency Response Using Model Linearizer

This example shows how to estimate the frequency response of a portion of a Simulink model using the **Model Linearizer**. To estimate the frequency response, you specify the portion of the model you want to estimate, the operating point for estimation, and the input signal to use for estimation.

Open Simulink Model and Model Linearizer

Open the Simulink model.

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



To open the **Model Linearizer**, in the Simulink model window, in the **Apps** gallery, click **Model Linearizer**.

Specify Portion of Model to Estimate

By default, **Model Linearizer** uses the linearization analysis points defined in the model (the model I/Os) to determine where to inject the test signal and where to measure the frequency response. The model `scdDCMotor` contains predefined linear analysis points: an input point at the compensator output, and an open-loop output after the unit gain block. For this example, use these predefined model I/Os to obtain the frequency response of the inner loop of the model with the outer loop open.

If you want to obtain the frequency response of a different portion of the model, on the **Estimation** tab of **Model Linearizer**, use the **Analysis I/Os** drop-down list. Analysis points for estimation work the same way as analysis points for linearization. For more information about linear analysis points, see “Specify Portion of Model to Linearize” on page 2-10.

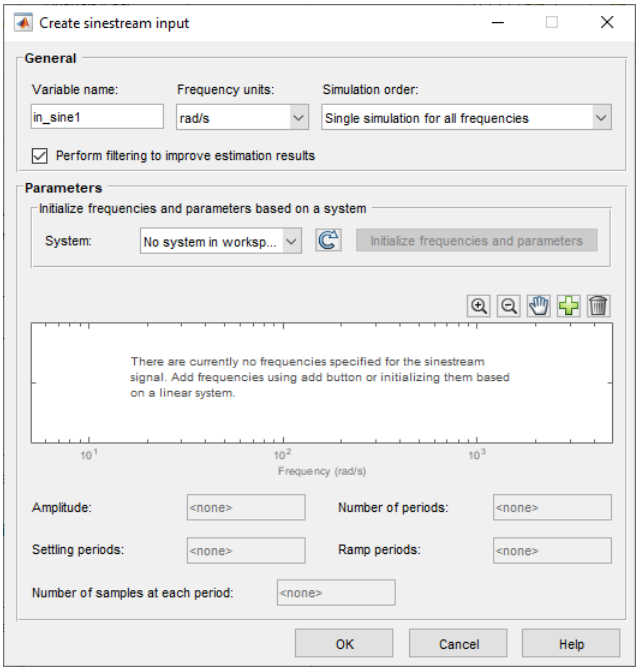
Specify Operating Point for Estimation

You perform frequency response estimation at a steady-state operating point of the model. You can compute or specify an operating point in **Model Linearizer** using the **Operating Point** drop-down list. By default, **Model Linearizer** uses the operating point defined by the model initial conditions. For this example, use that operating point. For more information about operating points, see “About Operating Points” on page 1-2.

Create Input Signal for Estimation

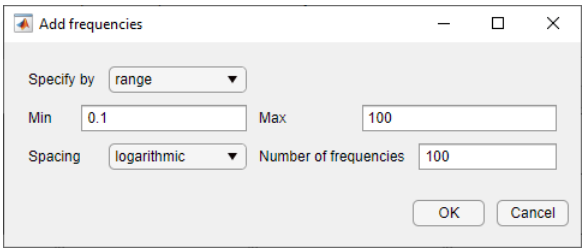
Frequency response estimation injects an input signal into the input analysis point you specify for estimation. For this example, configure a sinestream signal, which is a series of sinusoidal perturbations at frequencies you specify. (For more information about input signals, see “Estimation Input Signals” on page 5-24.)

1 On the **Estimation** tab, in the **Input Signal** drop-down list, select `Sinestream`. The Create sinestream input dialog box opens.

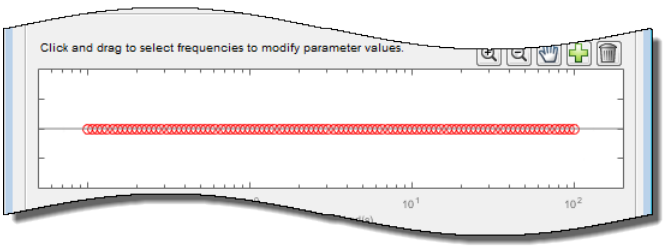


2 To specify frequency points for the input signal, click . In the Add frequencies dialog box, specify the frequency range and number of points for the input signal. The frequency points you specify are the frequencies at which **Model Linearizer** computes the estimated response.

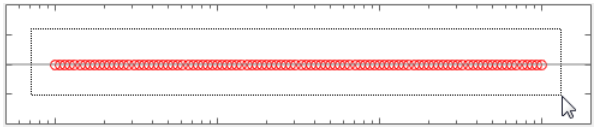
For this example, specify a range from 0.1 to 100 rad/s. Also, specify 100 logarithmically spaced frequencies.



Click **OK**. The added points are visible in the frequency content viewer of the Create sinestream input dialog box.



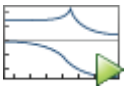
Select all these frequency points for the estimation input signal.

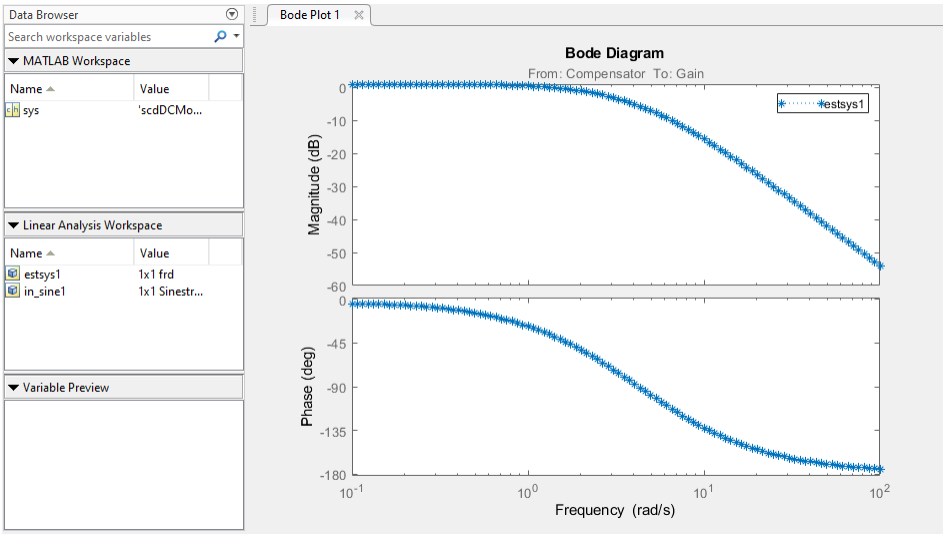


3 Specify the amplitude of the input signal. Enter `1` in the **Amplitude** field. When you specify a scalar value, **Model Linearizer** uses the same amplitude for all frequencies.

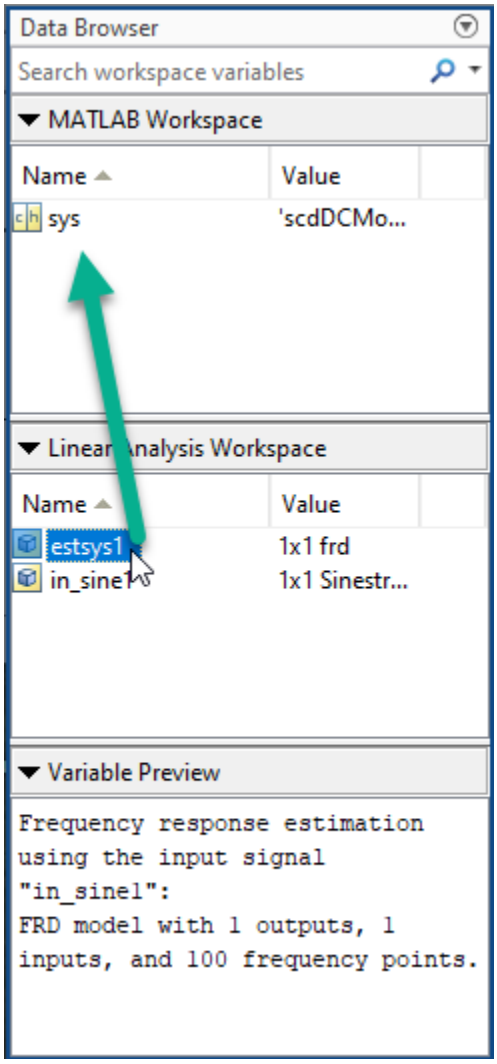
4 Click **OK** to create the sinestream input signal. The new input signal, `in_sine1`, appears in the **Linear Analysis Workspace**.

Estimate Frequency Response

You can now estimate the frequency response and generate a frequency-domain plot of the result. To do so, click  **Bode**. The estimated frequency response, appears in the **Linear Analysis Workspace** as the `frd` model `estsys1`.



To export the estimated frequency response model to the MATLAB workspace for further analysis, drag it from the **Linear Analysis Workspace** section of the Data Browser to the **MATLAB Workspace** section.



Analyze Estimated Frequency Response

The simulation results viewer in **Model Linearizer** lets you examine further details of the frequency response estimation. For more information, see “Analyze Estimated Frequency Response” on page 5-18.

See Also

More About

- “Frequency Response Estimation Basics” on page 5-2
- “Estimation Input Signals” on page 5-24
- “Analyze Estimated Frequency Response” on page 5-18
- “Estimate Frequency Response at the Command Line” on page 5-14

Estimate Frequency Response with Linearization-Based Input Using Model Linearizer

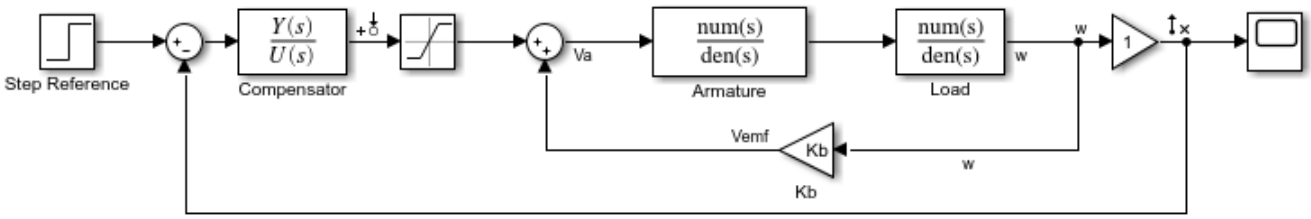
When you have a linear model representing the portion of your Simulink model that you want to estimate, you can use that model to generate the input signal. This alternative to manually specifying the estimation signal (as shown in “Estimate Frequency Response Using Model Linearizer” on page 5-6) can be useful when you are using frequency response estimation to validate a model obtained through linearization. This example shows how to perform frequency response estimation in **Model Linearizer** using an input signal that is based on the dynamics of an exact linearization of the model.

Linearize Simulink Model

In this example, linearize a Simulink model to obtain the linear model you use to generate the estimation input signal.

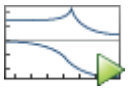
1. Open the Simulink model.

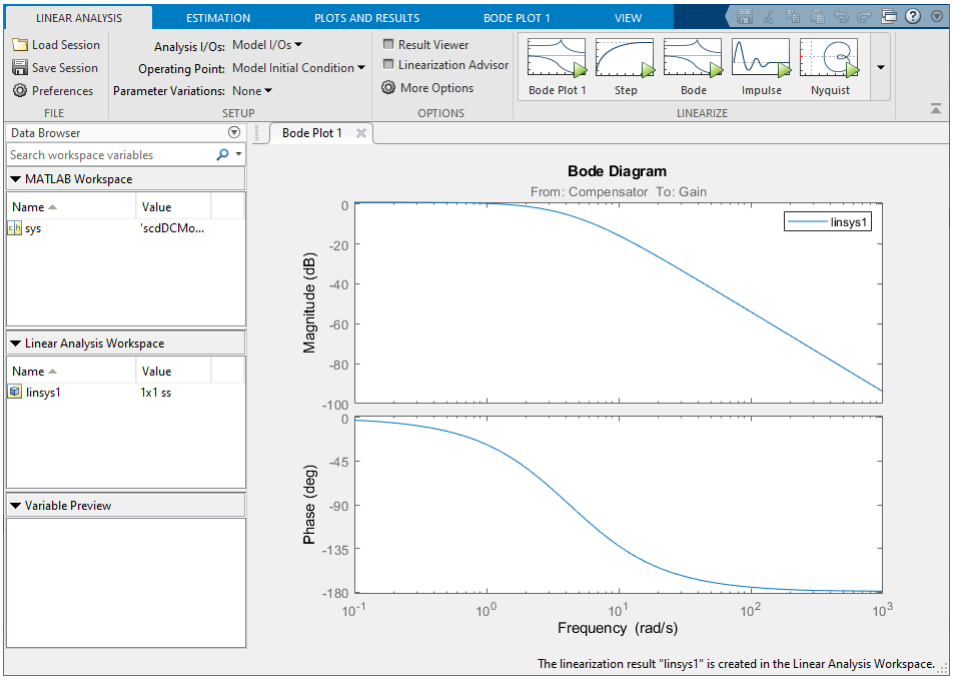
```
sys = 'scdDCMotor';
open_system(sys)
```



This model contains predefined linear analysis points. There is an input point at the compensator output and an open-loop output after the unit gain block. The response from the defined input to the defined output is the response of the inner loop of the model, with the outer loop open.

2. In the Simulink model window, in the **Apps** gallery, click **Model Linearizer**.
3. Linearize the model using the predefined analysis points and using the model initial conditions as the operating point.

   On the **Linear Analysis** tab, click  **Bode**.

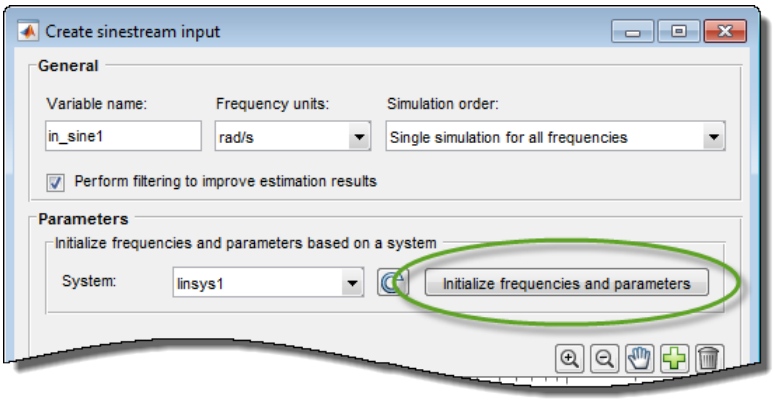


A new linearized model, `linsys1`, appears in the **Linear Analysis Workspace**.

Create Sinestream Input Signal

1. On the **Estimation** tab, in the **Input Signal** drop-down list, select `Sinestream`.
2. Initialize the input signal frequencies and parameters based on `linsys1`.

   In the Create sinestream input dialog box, click **Initialize frequencies and parameters**.

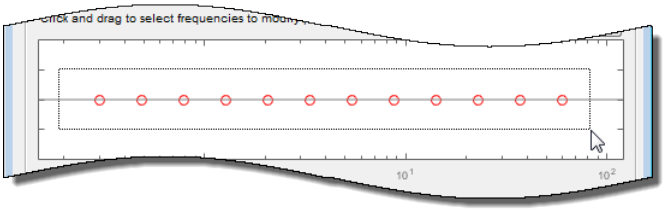


The frequency content viewer is populated with frequency points. The software chooses the frequencies and input signal parameters automatically based on the dynamics of `linsys1`. The software also automatically initializes other parameters of the sinestream signal, including:

- Amplitude
- Number of periods
- Settling periods

- Ramp periods
- Number of samples at each period

3 Select all the frequency points.



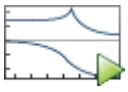
4 Specify the amplitude of the input signal.

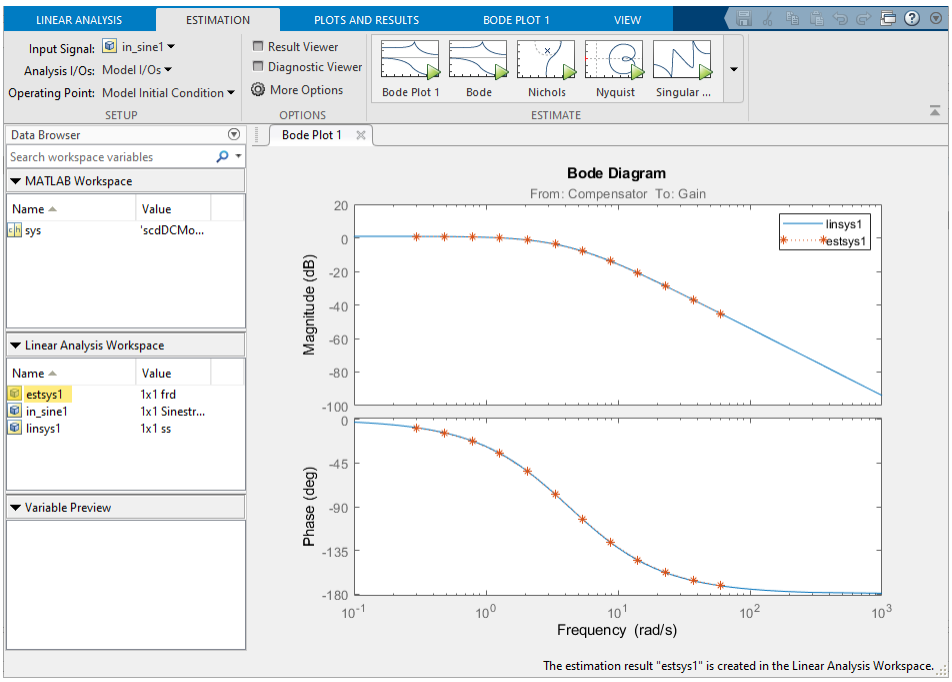
Enter 1 in the **Amplitude** box.

5 Create the input sinestream signal.

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

Estimate Frequency Response

Click  **Bode Plot 1** to estimate the frequency response.



The estimated system, `estsys1`, appears in the **Linear Analysis Workspace** and the its frequency response is added to **Bode Plot 1**.

The frequency response for the estimated model matches that of the linearized model. You can use this approach to validate an exact linearization.

See Also

More About

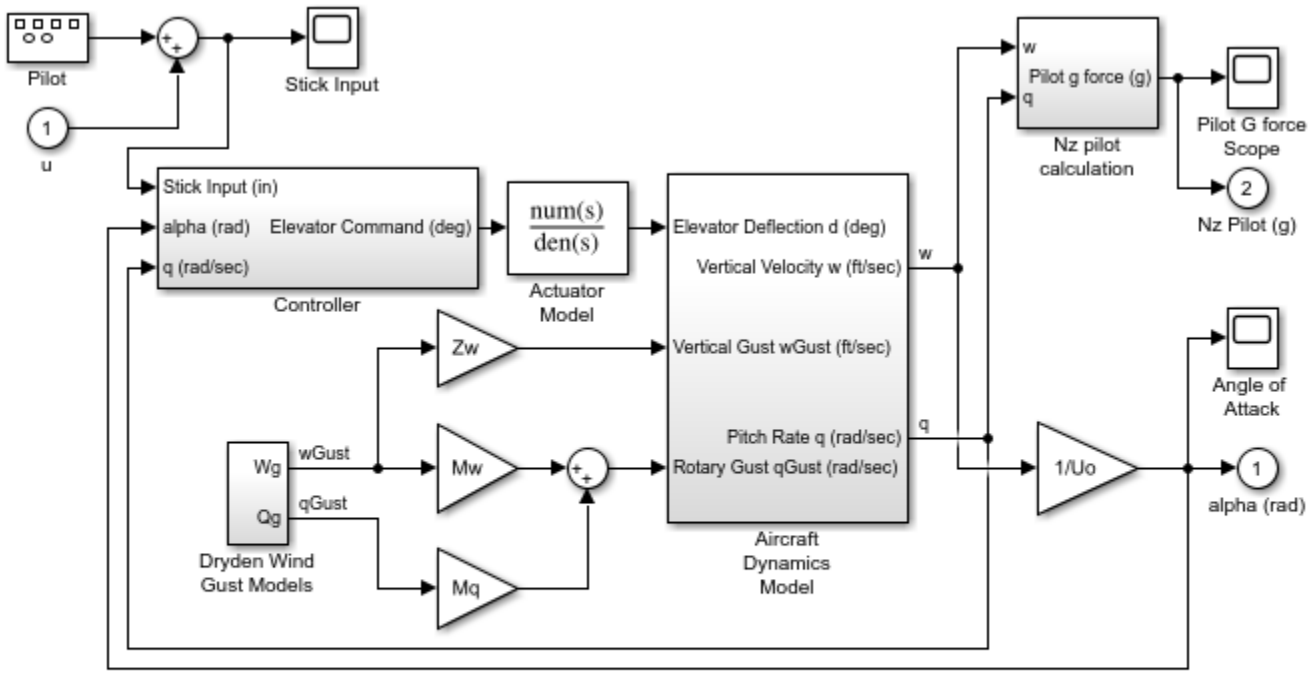
- “Estimation Input Signals” on page 5-24
- “Estimate Frequency Response Using Model Linearizer” on page 5-6

Estimate Frequency Response at the Command Line

This example shows how to estimate the frequency response of a Simulink® model at the MATLAB® command line.

Open the Simulink model.

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



Copyright 1990-2012 The MathWorks, Inc.

For more information on the general model requirements for frequency response estimation, see “Model Requirements” on page 5-4.

Specify input and output points for frequency response estimation using analysis points. Avoid placing analysis points on bus signals.

```
io(1) = linio('scdplane/Sum1',1);
io(2) = linio('scdplane/Gain5',1,'output');
```

For more information about linear analysis points, see “Specify Portion of Model to Linearize” on page 2-10 and `linio`.

Linearize the model and create a sinestream signal based on the dynamics of the resulting linear system. For more information, see “Estimation Input Signals” on page 5-24 and `frest.Sinestream`.

```
sys = linearize('scdplane',io);
input = frest.Sinestream(sys);
```

If your model has not reached steady state, initialize the model using a steady-state operating point before estimating the frequency response. You can check whether your model is at steady state by simulating the model. For more information on finding steady-state operating points, see “Compute Steady-State Operating Points” on page 1-5.

Find all source blocks in the signal paths of the linearization outputs that generate time-varying signals. Such time-varying signals can interfere with the signal at the linearization output points and produce inaccurate estimation results.

```
srcblks = frest.findSources('scdplane',io);
```

To disable the time-varying source blocks, create an `frestimateOptions` option set and specify the `BlocksToHoldConstant` option.

```
opts = frestimateOptions;
opts.BlocksToHoldConstant = srcblks;
```

Estimate the frequency response.

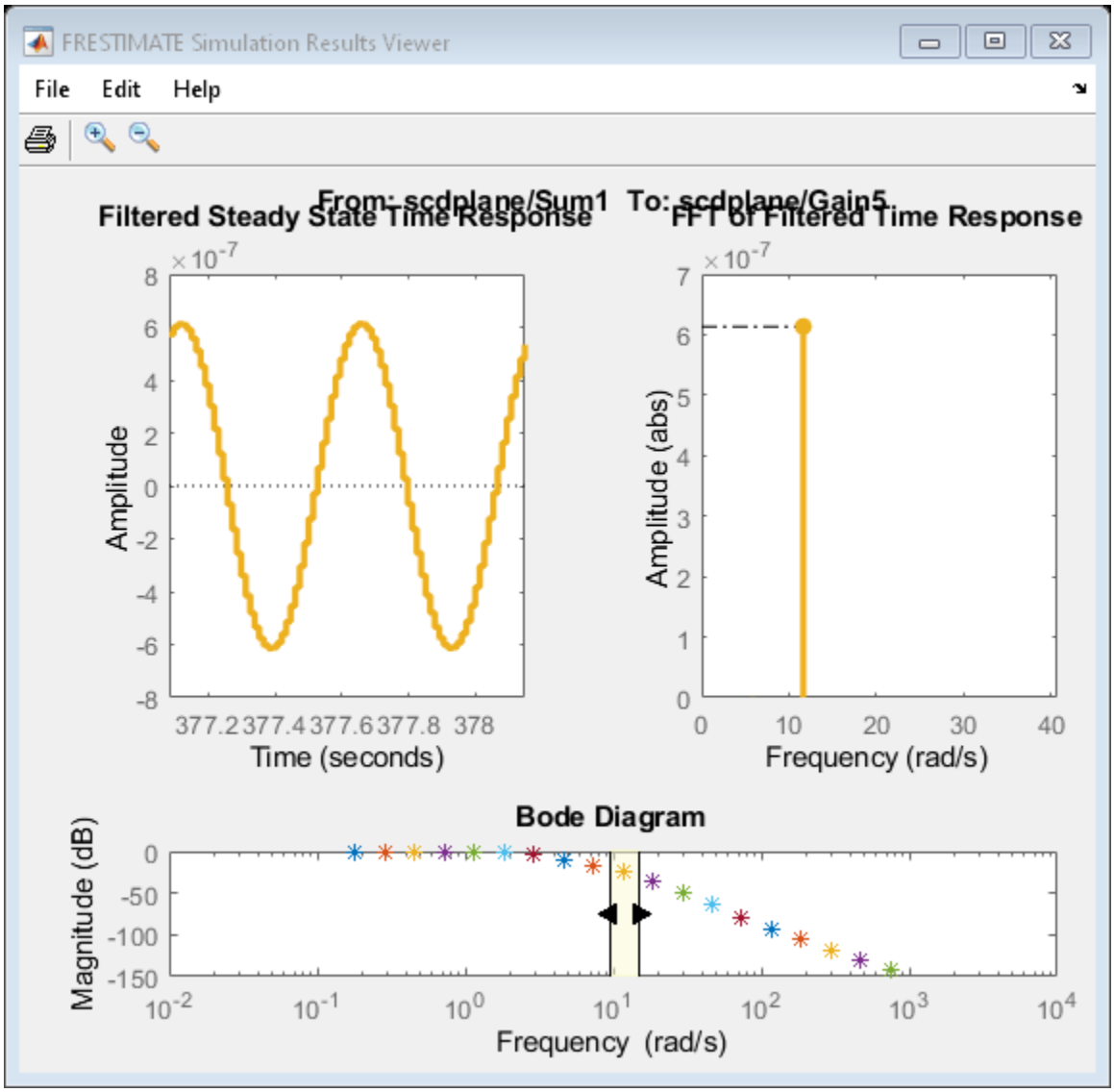
```
[sysest,simout] = frestimate('scdplane',io,input,opts);
```

`sysest` is the estimated frequency response. `simout` is a `Simulink.Timeseries` object representing the simulated output.

To speed up your estimation or decrease its memory requirements, see “Managing Estimation Speed and Memory” on page 5-68.

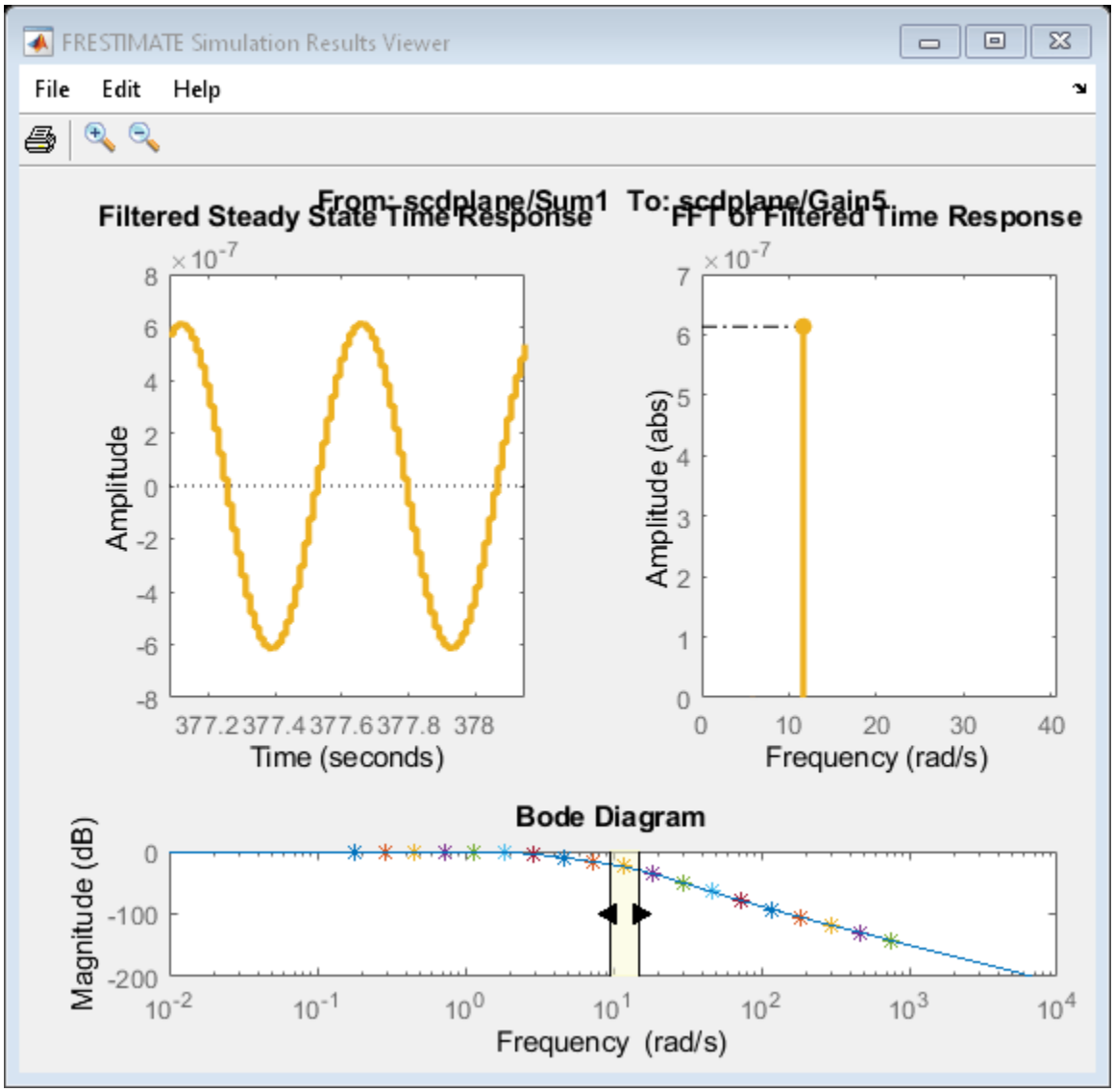
Open the Simulation Results Viewer to analyze the estimated frequency response.

```
frest.simView(simout,input,sysest)
```



You can also compare the estimated frequency response, `sysest`, to an exact linearization of your system, `sys`.

```
frest.simView(simout,input,sysest,sys)
```



The **Bode Diagram** plot shows the response `sys` as a blue line.

See Also

`linio` | `operspec` | `findop` | `frest.findSources` | `frestimateOptions` | `frestimate`

More About

- “Estimation Input Signals” on page 5-24
- “Analyze Estimated Frequency Response” on page 5-18

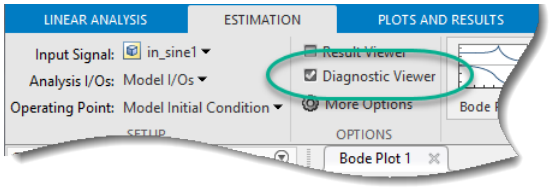
Analyze Estimated Frequency Response

When you perform frequency response estimation, you can analyze the result by examining the raw simulated response and the FFT used to convert it to an estimated frequency response. To do so, use the Diagnostic viewer (in **Model Linearizer**) or the Simulation Results Viewer (at the MATLAB command line).

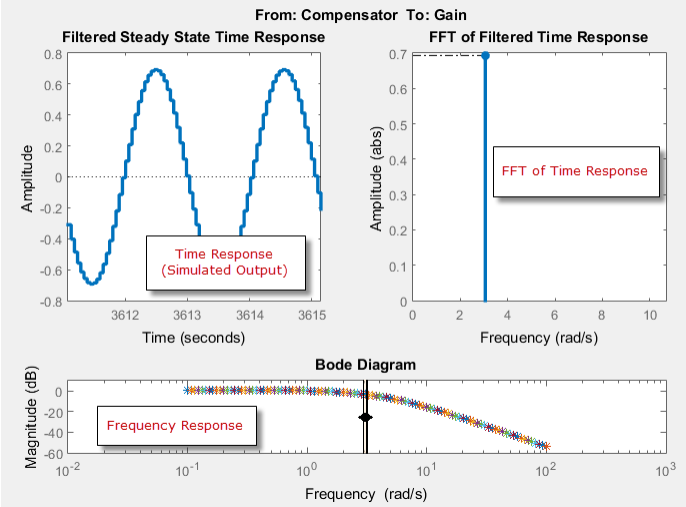
View Simulation Results

View Simulation Results Using Model Linearizer

Use the Diagnostic Viewer to analyze the results of your frequency response estimation, obtained by performing the steps in “Estimate Frequency Response Using Model Linearizer” on page 5-6, with the extra step of activating the Diagnostic Viewer before performing estimation. To do so, in the **Estimation** tab, select **Diagnostic Viewer**.



Then, perform the estimation. The Diagnostic Viewer appears in the plot pane.



To open the Diagnostic Viewer to view a previously estimated model in the **Model Linearizer**:

1. In the **Linear Analysis Workspace**, select the estimated model.
2. In the **Plots and Results** tab, click **Diagnostic Viewer**.

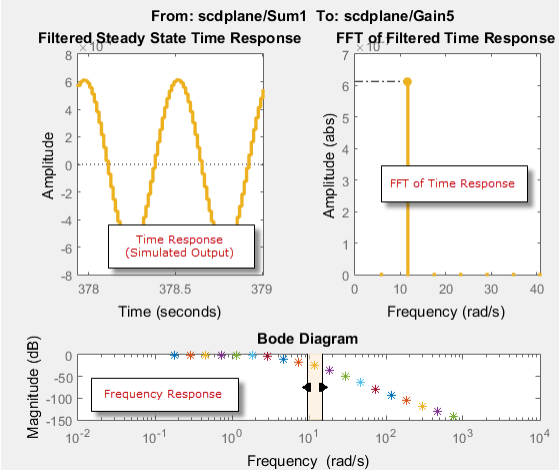
> **Note** This option is only available for models that have been previously estimated with the **Diagnostic Viewer** check box selected.

View Simulation Results (MATLAB Code)

Use the Simulation Results Viewer to analyze the results of your frequency response estimation, obtained by performing the steps in “Estimate Frequency Response at the Command Line” on page 5-14. Make sure you keep the `simout` output argument of `frestimate`.

To open the Simulation Results Viewer using the `frest.simView` command using the simulated output `simout`, the input signal `input` is that you used for estimation, and the estimated frequency response you obtained, `sysest`.

```
frest.simView(simout,input,sysest)
```



Interpret Frequency Response Estimation Results

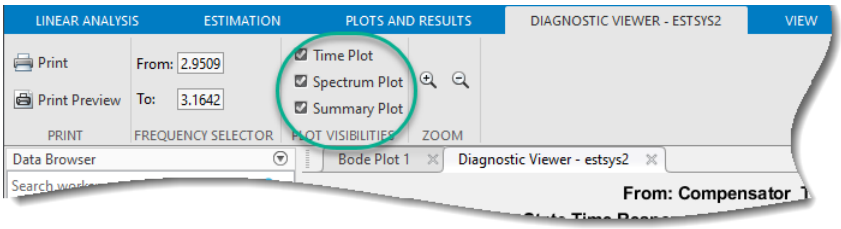
Select Plots Displayed in Diagnostic Viewer

By default, the Diagnostic Viewer shows these plots:

- Frequency Response
- Time Response (Simulated Output)
- FFT of Time Response

To select the plots displayed in the Diagnostic Viewer using the **Model Linearizer**:

1. If the **Diagnostic Viewer** tab is not visible, in the **Plots and Results** tab, select the **Diagnostic Viewer** plot.
2. In the **Diagnostic Viewer** tab, in the **Plot Visibilities** section, select the plots that you want to view.



To modify plot settings, such as axis frequency units, right-click on a plot, and select the corresponding option.

Select Plots Displayed in Simulation Results Viewer

By default, the Simulation Results Viewer shows these plots:

- Frequency Response
- Time Response (Simulated Output)
- FFT of Time Response

To select the plots displayed in the Simulation Results Viewer, choose the corresponding plot from the **Edit > Plots** menu. To modify plot settings, such as axis frequency units, right-click a plot, and select the corresponding option.

Frequency Response

Use the Bode plot to analyze the frequency response. If the frequency response does not match the dynamics of your system, see “Troubleshooting Frequency Response Estimation” on page 5-43 for information about possible causes and solutions. While troubleshooting, you can use the Bode plot controls to view the time response at the problematic frequencies on page 5-21.

You can usually improve estimation results by either modifying your input signal on page 5-40 or disabling the model blocks that drive your system away from the operating point, and repeating the estimation.

Time Response (Simulated Output)

Use this plot to check whether the simulated output is at steady state at specific frequencies. If the response has not reached steady state, see “Time Response Not at Steady State” on page 5-43 for possible causes and solutions.

If you used the sinestream input for estimation, check both the filtered and the unfiltered time response. You can toggle the display of filtered and unfiltered output by right-clicking the plot and selecting **Show filtered steady state output only**. If both the filtered and unfiltered response appear at steady state, then your model must be at steady state. You can explore other possible causes in “Troubleshooting Frequency Response Estimation” on page 5-43.

Note If you used the sinestream input for estimation, toggling the filtered and unfiltered display only updates the Time Response and FFT plots. This selection does not change the estimation results. For more information about filtering during estimation, see the Algorithms section of `frestimate`.

FFT of Time Response

Use this plot to analyze the spectrum of the simulated output.

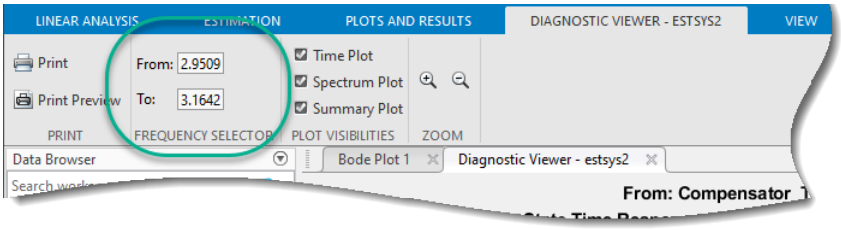
For example, you can use the spectrum to identify strong nonlinearities. When the FFT plot shows large amplitudes at frequencies other than the input signal, your model is operating outside of linear range. If you are interested in analyzing the linear response of your system for small perturbations, explore possible solutions in “FFT Contains Large Harmonics at Frequencies Other than the Input Signal Frequency” on page 5-45.

Analyze Simulated Output and FFT at Specific Frequencies

Using the Diagnostic Viewer in Model Linearizer

Use the controls in the **Diagnostic Viewer** tab of the **Model Linearizer** to analyze the estimation results at specific frequencies.

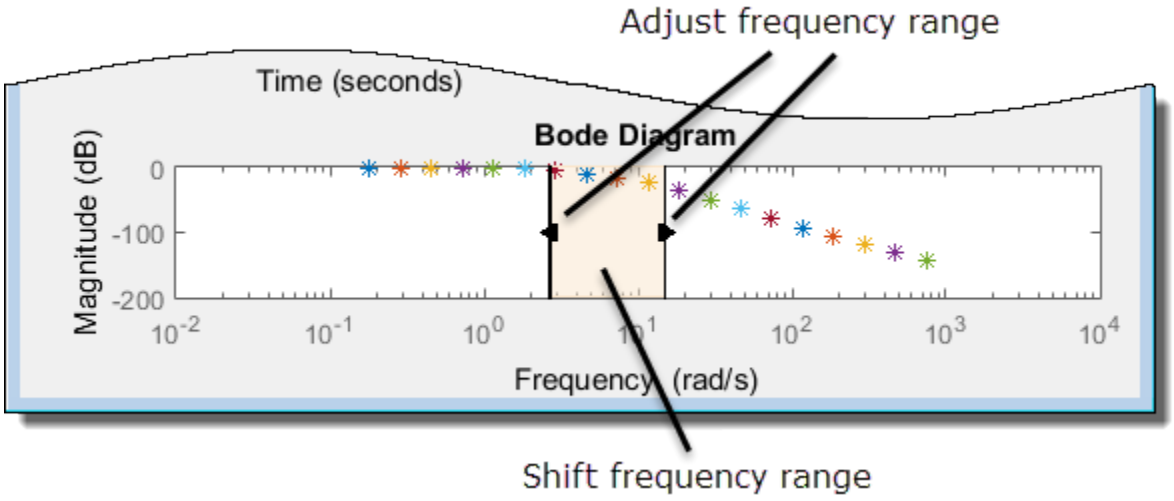
1. If the **Diagnostic Viewer** tab is not visible, in the **Plots and Results** tab, select the **Diagnostic Viewer** plot.
2. In the **Diagnostic Viewer** tab, in the **Frequency Selector** section, specify the frequency range that you want to inspect. Use the frequency units used in the Bode plot in the Diagnostic Viewer.



Using the Simulation Results Viewer

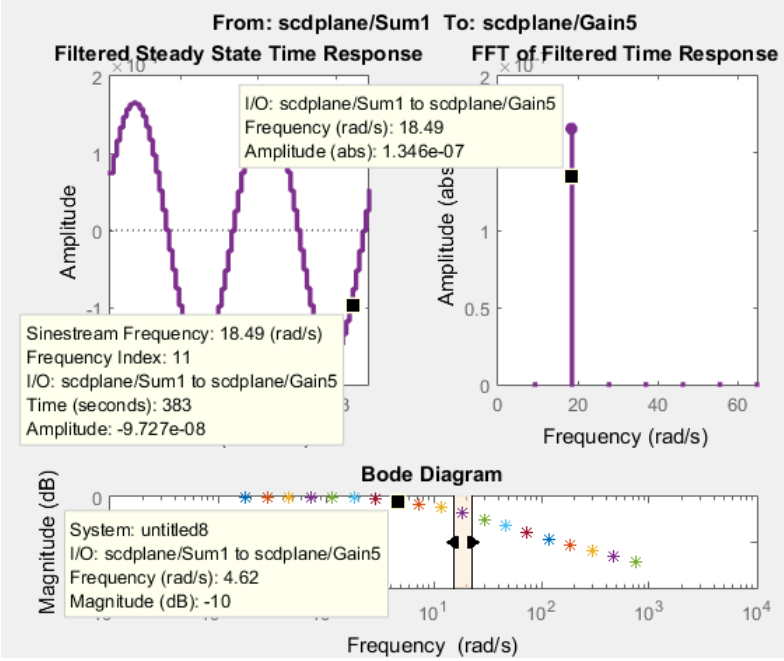
In the Simulation Results Viewer, use the Bode controls to display the simulated output and its spectrum at specific frequencies.

- Drag the arrows individually to display the time response and FFT at specific frequencies.
- Drag the shaded region to shift the time response and FFT to a different frequency range.



Annotate Frequency Response Estimation Plots

You can display a data tip on the Time Response, FFT, and Bode plots in the Simulation Results Viewer by clicking the corresponding curve. Dragging the data tip updates the information.



Data tips are useful for correcting poor estimation results at a specific sinestream frequency, which requires you to modify the input at a specific frequency. You can use the data tip to identify the frequency index where the response does not match your system.

In the previous figure, the Time Response data tip shows that the frequency index is 11. You can use this frequency index to modify the corresponding portion of the input signal. For example, to modify the `NumPeriods` and `SettlingPeriods` properties of the sinestream signal, using MATLAB code:

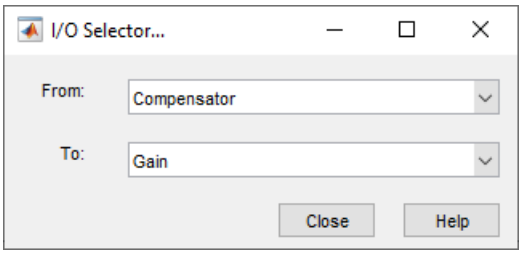
```
input.NumPeriods(11) = 80;
input.SettlingPeriods(11) = 75;
```

To modify the sinestream in the **Model Linearizer**, see “Modify Sinestream Signal Using Model Linearizer” on page 5-40

Displaying Estimation Results for Multiple-Input Multiple-Output (MIMO) Systems

For MIMO systems, view frequency response information for specific input and output channels:

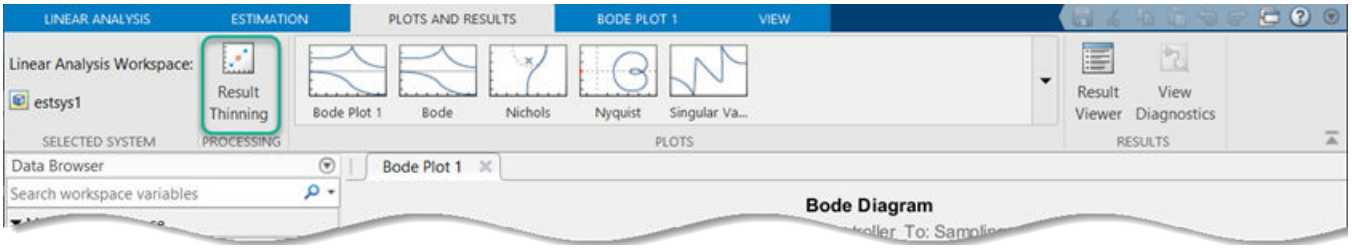
1. In both the Diagnostic Viewer and Simulation Results Viewer, right-click any plot, and select **I/O Selector**.
2. Choose the input channel in the **From** list and the output channel in the **To** list.



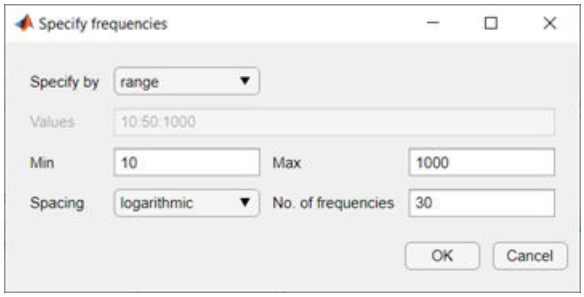
Result Thinning

When you have an estimated frequency response result with a large number of frequency points and you are interested in data across a certain frequency range at a specific resolution, you can use **Result Thinning** in **Model Linearizer** to extract interpolated frequency response data from an estimated frequency response `frd` model across a specified frequency range and number of frequency points.

To apply thinning to an estimated frequency response result, select the estimated model in the **Linear Analysis Workspace** or **MATLAB Workspace** pane. Then, on the **Plots and Results** tab, click **Result Thinning**.



In the Specify frequencies dialog box, specify the frequencies by either `range` or `values`. By default, the dialog lets you specify logarithmically-spaced or linearly-spaced frequencies by range. To specify frequencies by values, in the **Specify by** list, select `values`, and then specify a vector of frequency values using the **Values** parameter. The frequency values must lie between the smallest and largest frequency points in the model you want to thin.



Click **OK**. The software performs linear interpolation and returns an `frd` model containing the interpolated frequency response data at the specified frequencies.

For an example of thinning a response estimated with PRBS input signal, see “Frequency Response Estimation in Model Linearizer Using Pseudorandom Binary Sequence” on page 5-101.

See Also

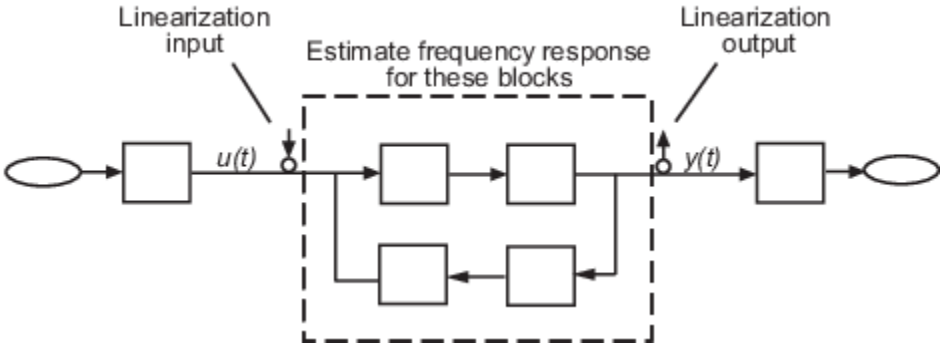
`frest.simView` | `frest.simCompare` | `frestimate`

More About

- “Estimate Frequency Response Using Model Linearizer” on page 5-6
- “Estimate Frequency Response at the Command Line” on page 5-14
- “Frequency Response Estimation Basics” on page 5-2

Estimation Input Signals

Frequency response estimation requires an input signal to excite the model at frequencies of interest. The software then measures the response at the specified output, using the input signal and measured response to estimate the frequency response.



When you perform frequency response estimation, you specify what type of input signal to use and what its properties are.

Offline Estimation

The following table summarizes the types of input signals you can use for offline estimation in **Model Linearizer** or at the MATLAB command line for use with `frestimate`.

Signal	Description
Sinestream on page 5-29	A series of sinusoidal perturbations applied one after another. Sinestream signals are recommended for most situations. They are especially useful when your system contains strong nonlinearities or you require highly accurate frequency response models.
Chirp on page 5-33	A swept-frequency signal that excites your system at a range of frequencies, such that the input frequency changes instantaneously. Chirp signals are useful when your system is nearly linear in the simulation range. They are also useful when you want to obtain a response quickly for a lot of frequency points.
PRBS on page 5-36	A deterministic pseudorandom binary sequence that shifts between two values and has white-noise-like properties. PRBS signals reduce total estimation time compared to using sinestream input signals, while producing comparable estimation results. PRBS signals are useful for estimating frequency responses for communications and power electronics systems.
Random on page 5-25	A random input signal. Random signals are useful because they can excite the system uniformly at all frequencies up to the Nyquist frequency.
Step on page 5-26	A step input signal. Step inputs are quick to simulate and can be useful as a first try when you do not have much knowledge about the system you are trying to estimate.

Signal	Description
Arbitrary on page 5-26	A MATLAB timeseries with which you can specify any time-varying signal as input.

In general, the estimated frequency response is related to the input and output signals as:

$$Resp = \frac{FFT(y_{est}(t))}{FFT(u_{est}(t))}.$$

Here, $u_{est}(t)$ is the injected input signal and y_{est} (t) is the corresponding simulated output signal. For more details, see the Algorithms section of `frestimate`.

Online Estimation

For online estimation with the Frequency Response Estimator block, you can use two types of input signals:

- Sinestream on page 5-29 — A series of sinusoidal perturbations applied one after another
- Superposition on page 5-27 — A set of sinusoidal perturbations applied simultaneously

For online estimation, using a sinestream signal can be more accurate and can accommodate a wider range of frequencies than a superposition signal. The sinestream mode can also be less intrusive. However, due to the sequential nature of the sinestream perturbation, each frequency point you add increases the experiment time. Thus the estimation experiment is typically much faster with a superposition signal with satisfactory results.

To specify which type of input signal to use for online estimation, use the **Experiment mode** parameter of the Frequency Response Estimator block.

Sinestream Signals

For details about the structure of sinestream signals and how to create them, see “Sinestream Input Signals” on page 5-29.

Chirp Signals

For details about the structure of chirp signals and how to create them, see “Chirp Input Signals” on page 5-33.

PRBS Signals

For details about the structure of PRBS signals and how to create them, see “PRBS Input Signals” on page 5-36.

Random Signals

Random signals are useful because they can excite the system uniformly at all frequencies up to the Nyquist frequency. To create a random input signal for estimation:

- In the **Model Linearizer**, on the **Estimation** tab, select **Input Signal** > **Random**.

- At the command line, use `frest.Random` to create the random signal and use it as an input argument to `frestimate`.

The random signal comprises uniformly distributed random numbers in the interval `[0 Amplitude]` or `[Amplitude 0]` for positive and negative amplitudes, respectively. You can specify the amplitude, sample time, and number of samples directly when you create the input signal. Alternatively, if you have a relevant linear time-invariant (LTI) model such as a state-space (`ss`) model, you can use it to initialize the random signal parameters. For instance, if you have an exact linearization of your system, you can use it to initialize the parameters.

When you use a random input signal for estimation, the frequencies returned in the estimated `frd` model depend on the length and sampling time of the signal. They are the frequencies obtained in the fast Fourier transform of the input signal (see the Algorithm section of `frestimate`).

Step Signals

Step inputs are quick to simulate. Like a random signal, a step signal can excite the system at all frequencies up to the Nyquist frequency. For those reasons, a step input can be useful as a first try when you do not have much knowledge about the system you are trying to estimate. However, the amplitude of the excitation decreases rapidly with increasing frequency. Therefore, step signals are best used to identify low-order plants where the slowest poles are dominant. Step inputs are not recommended for estimation across a wide range of frequencies.

To create a step input signal for estimation, use `frest.createStep`. This function creates a MATLAB `timeseries` that represents a step input having the sample time, step time, step size, and total signal length that you specify when you call `frest.createStep`.

To use the step input signal you created in the MATLAB workspace:

- In the **Model Linearizer**, on the **Estimation** tab, select it from the **Existing Input Signals** section of the **Input Signal** drop-down list.
- At the command line, use it as an input argument to `frestimate`.

When you use a step input signal for estimation, the frequencies returned in the estimated `frd` model depend on the length and sampling time of the signal. They are the frequencies obtained in the fast Fourier transform of the input signal (see the Algorithm section of `frestimate`).

Arbitrary Signals

If you want to use a signal other than a sinestream, chirp, step, or random signal, you can provide your own MATLAB `timeseries` object. For instance, you can create a `timeseries` representing a ramp, sawtooth, or square wave input.

To use a `timeseries` object as the input signal for estimation, first create the `timeseries` in the MATLAB workspace. Then:

- In the **Model Linearizer**, on the **Estimation** tab, select it from the **Existing Input Signals** section of the **Input Signal** drop-down list.
- At the command line, use it as an input argument to `frestimate`.

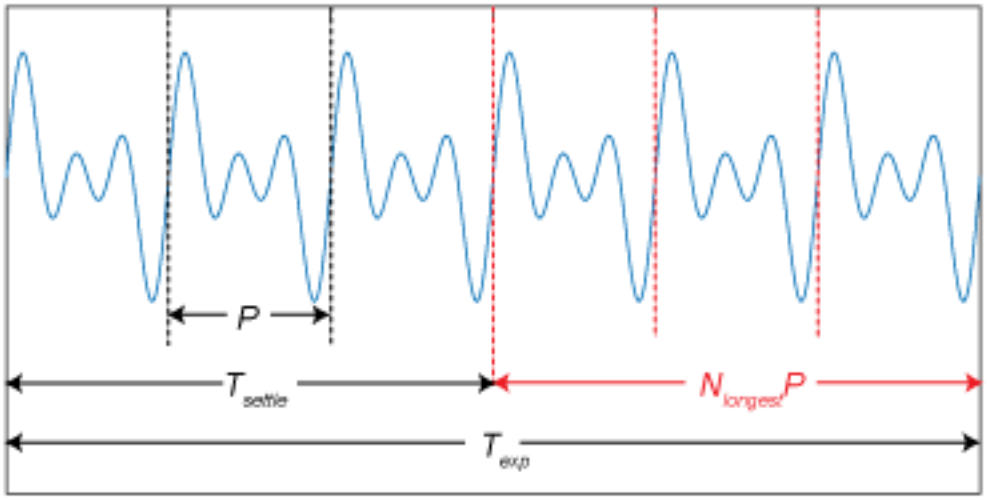
When you use an arbitrary input signal for estimation, the frequencies returned in the estimated `frd` model depend on the length and sampling time of the signal. They are the frequencies obtained in the fast Fourier transform of the input signal (see the Algorithm section of `frestimate`).

Superposition Signals

Superposition signals are available only for online estimation with the Frequency Response Estimator block. For frequency response estimation at a vector of frequencies $\omega = [\omega_1, \ldots, \omega_N]$ at amplitudes $A = [A_1, \ldots, A_N]$, the superposition signal is given by:

$$\Delta u = \sum_i A_i \sin(\omega_i t) .$$

The block supplies the perturbation Δu for the duration of the experiment (while the **start/stop** signal is positive). The block determines how long to wait for system transients to die away and how many cycles to use for estimation as shown in the following illustration.



T_{exp} is the experiment duration that you specify with your configuration of the start/stop signal (See the **start/stop** port description on the block reference page for more information). For the estimation computation, the block uses only the data collected in a window of $N_{longest}P$. Here, P is the period of the slowest frequency in the frequency vector ω, and $N_{longest}$ is the value of the **Number of periods of the lowest frequency used for estimation** block parameter. Any cycles before this window are discarded. Thus, the settling time $T_{settle} = T_{exp} - N_{longest}P$. If you know that your system settles quickly, you can shorten T_{exp} without changing $N_{longest}$ to effectively shorten T_{settle}. If your system is noisy, you can increase $N_{longest}$ to get more averaging in the data-collection window. Either way, always choose T_{exp} long enough for sufficient settling and sufficient data-collection. The recommended $T_{exp} = 2N_{longest}P$.

To use a superposition signal for estimation, in the Frequency Response Estimator block, set the **Experiment mode** parameter to **Superposition**. For details, see Frequency Response Estimator.

See Also

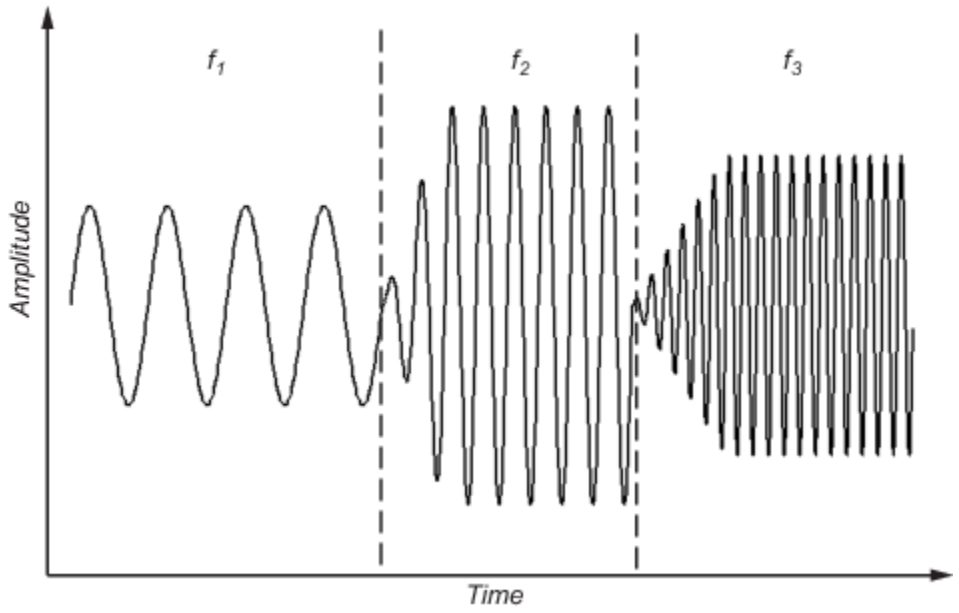
`frestimate` | `frest.createStep` | `frest.Random` | `frest.Sinestream` | `frest.Chirp` | `frest.PRBS`

More About

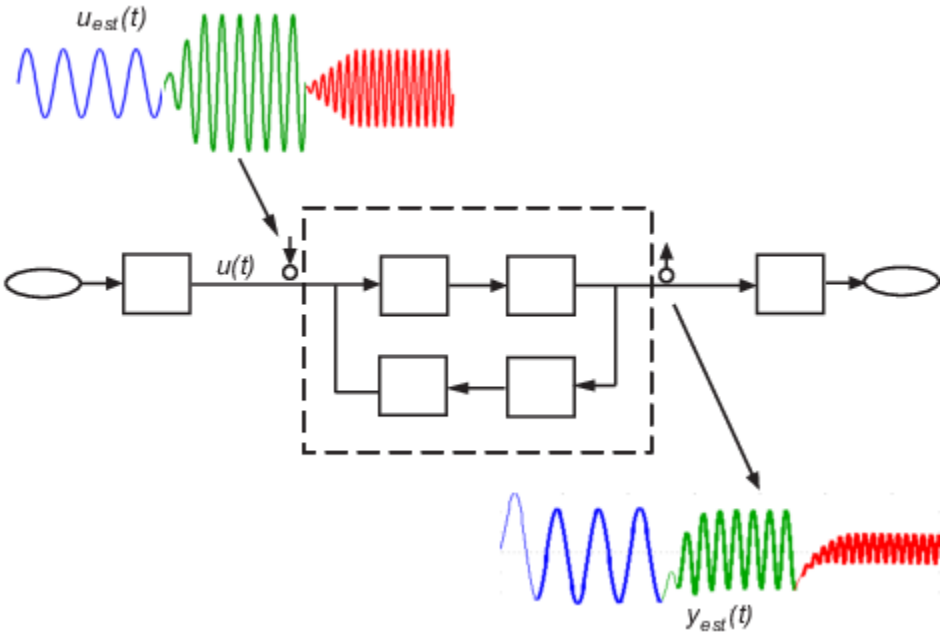
- “Sinestream Input Signals” on page 5-29
- “Chirp Input Signals” on page 5-33
- “Estimate Frequency Response Using Model Linearizer” on page 5-6
- “Estimate Frequency Response at the Command Line” on page 5-14

Sinestream Input Signals

In frequency response estimation, a *sinestream* signal consists of sine waves of varying frequencies applied one after another. Each frequency excites the system for a period of time.



You can use a sinestream input signal for estimation at the command line, in **Model Linearizer**, or with the Frequency Response Estimator block. The estimation algorithm injects the sinestream signal at the input point you specify for estimation, and measures the response at the output point.



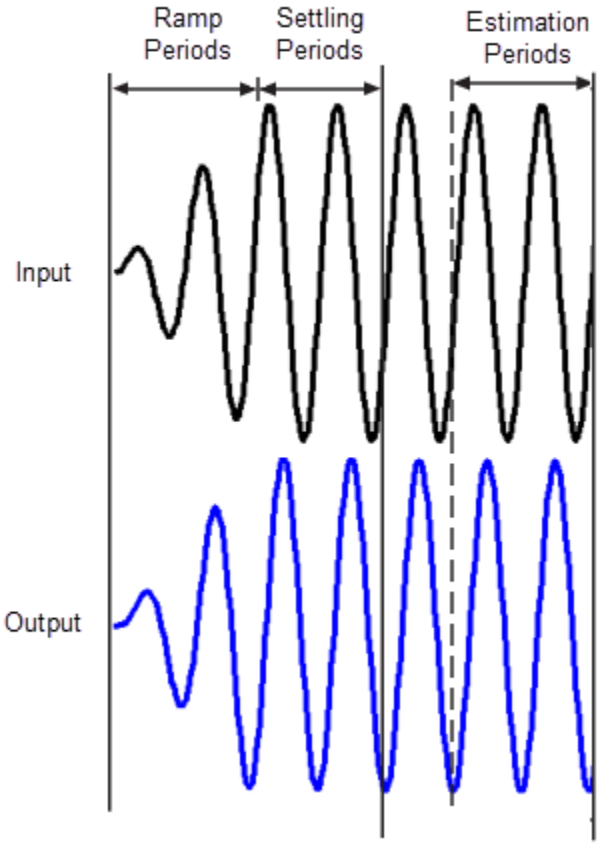
Sinestream signals are recommended for most situations. They are especially useful when your system contains strong nonlinearities or you require highly accurate frequency response models. The

frequency-response model that results when you use a sinestream input contains all the frequencies in the sinestream signal.

When you create your sinestream input signal, you specify the following parameters for each frequency:

- Number of periods for ramping up the signal to its maximum value
- Number of settling periods
- Total number of periods.

The number of estimation periods is the total number of periods minus the number of settling periods. The estimation algorithms discard response data collected during the ramp periods and settling periods time frames. Doing so allows any transient responses to die out. The algorithm uses data collected during the estimation periods to compute the estimated frequency response.



(In offline estimation, if FIR filtering is on, the software also discards the first estimation period as shown in the illustration. For details about the offline and online estimation algorithms, see the Algorithms sections of `frestimate` and Frequency Response Estimator, respectively.)

Create Sinestream Signals Using Model Linearizer

In the **Model Linearizer**, to use a sinestream input signal for estimation, on the **Estimation** tab, select:

- **Input Signal > Sinestream** when the sample time of the I/Os is continuous.
- **Input Signal > Fixed Sample Time Sinestream** when the sample time of the I/Os is discrete.

You can specify the frequencies to use in the sinestream in one of two ways:

- Manually, as shown in “Estimate Frequency Response Using Model Linearizer” on page 5-6
- Based on the dynamics of a linear model, such as a linearization of your system, as shown in “Estimate Frequency Response with Linearization-Based Input Using Model Linearizer” on page 5-10

Other parameters you can specify for a sinestream signal in **Model Linearizer** include:

- **Amplitude** — Amplitude of injected sine waves
- **Number of periods** — Total number of periods at each frequency
- **Settling periods** — Number of periods to discard for the estimation computation
- **Ramp periods** — Number of periods for ramping up the amplitude of each sine wave to its maximum value
- **Perform filtering to improve estimation results** — Filter the response data before estimating frequency response (see the Algorithms section of `frestimate`)

Create Sinestream Signals Using MATLAB Code

To create a sinestream signal for estimation at the command line with `frestimate`, use:

- `frest.Sinestream` — Use when signal at input linearization point is continuous.
- `frest.createFixedTsSinestream` — Use when signal at input linearization point is discrete.

Sinestream Signals for Online Estimation

You can use a sinestream signal for online estimation with the Frequency Response Estimator block. To do so, set the **Experiment mode** parameter to **Sinestream**. Other relevant block parameters include:

- **Frequencies** — Vector of frequencies for the sinestream signal.
- **Amplitudes** — Signal amplitudes. You can specify a single amplitude for all frequencies, or separate amplitudes for each frequency.
- **Number of settling periods** — Number of periods to discard for the estimation computation.
- **Number of estimation periods** — Number of periods to use in the estimation computation.

For details, see the Frequency Response Estimator block reference page.

See Also

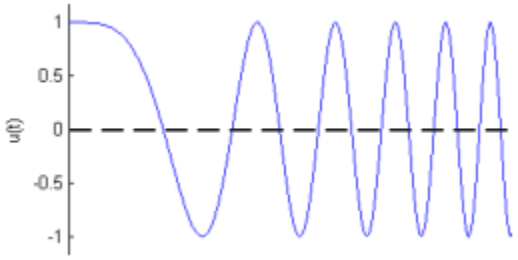
`frest.Sinestream` | `frest.createFixedTsSinestream`

More About

- “Estimation Input Signals” on page 5-24
- “Chirp Input Signals” on page 5-33
- “Modify Estimation Input Signals” on page 5-40
- “Frequency Response Estimation Using Simulation-Based Techniques” on page 5-74

Chirp Input Signals

A swept-frequency cosine input signal, or chirp signal, excites your system at a range of frequencies, such that the input frequency changes instantaneously.



You can use a chirp input signal for frequency-response estimation at the command line or in **Model Linearizer**. The estimation algorithm injects the chirp signal at the input point you specify for estimation, and measures the response at the output point. Chirp signals are useful when your system is nearly linear in the simulation range. They are also useful when you want to obtain a response quickly for a lot of frequency points. The frequency-response model that results when you use a chirp input contains only frequencies that fall within the range of the chirp.

Create Chirp Signals Using Model Linearizer

In the **Model Linearizer**, to use a chirp input signal for estimation, in the **Estimation** tab, select **Input Signal > Chirp**. You can specify the frequency range and other properties of the chirp in one of two ways:

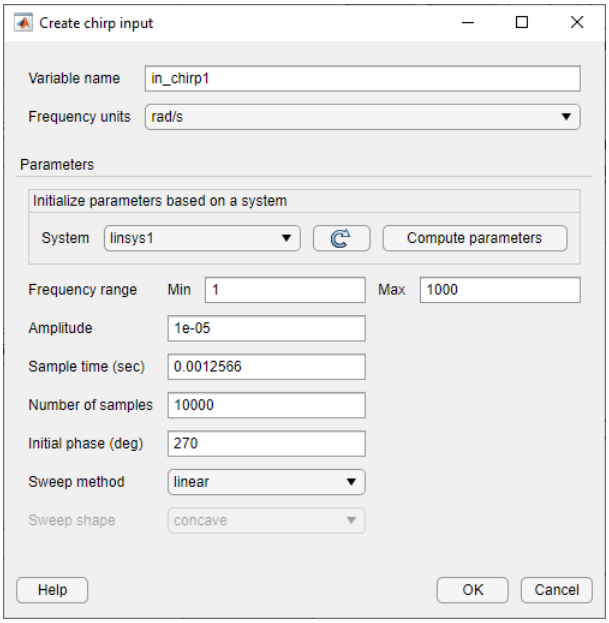
- Enter the values manually in the Create chirp input dialog box.
- Initialize the frequencies based on the dynamics of a linear model, such as a linearization of your system.

To create a chirp signal based on a linear model:

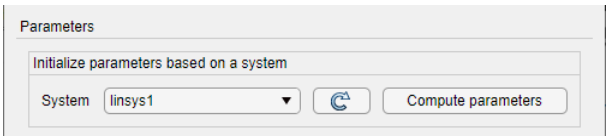
1. Obtain a linearized model, `linsys1`.

   For example, see “Linearize Simulink Model at Model Operating Point” on page 2-54, which shows how to linearize a model.
2. In the **Model Linearizer**, in the **Estimation** tab, select **Input Signal > Chirp**.

   The Create chirp input dialog box opens.



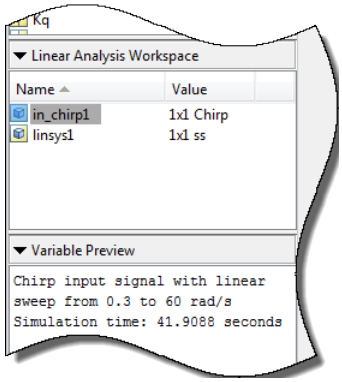
3 In the **System** list, select `linsys1`. Click **Compute parameters**.



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

- **Frequency range** — Range of frequencies for the chirp, which the software chooses based on the frequencies at which the linear system has interesting dynamics.
- **Amplitude** — Amplitude of applied perturbation.
- **Sample time** — Sample time of signal. To avoid aliasing, the software chooses the sample time such that 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**.



You can now select this signal in the **Input Signal** drop-down list for estimation.

The mapping between the parameters of the Create chirp input dialog box in the **Model Linearizer** and the properties of `frest.Chirp` is as follows:

Create chirp input dialog box	**`frest.Chirp` property**
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'`

Create Chirp Signals Using MATLAB Code

To create a chirp signal for estimation at the command line with `frestimate`, use `frest.Chirp`. See that page for examples and more information about chirp signal properties.

See Also

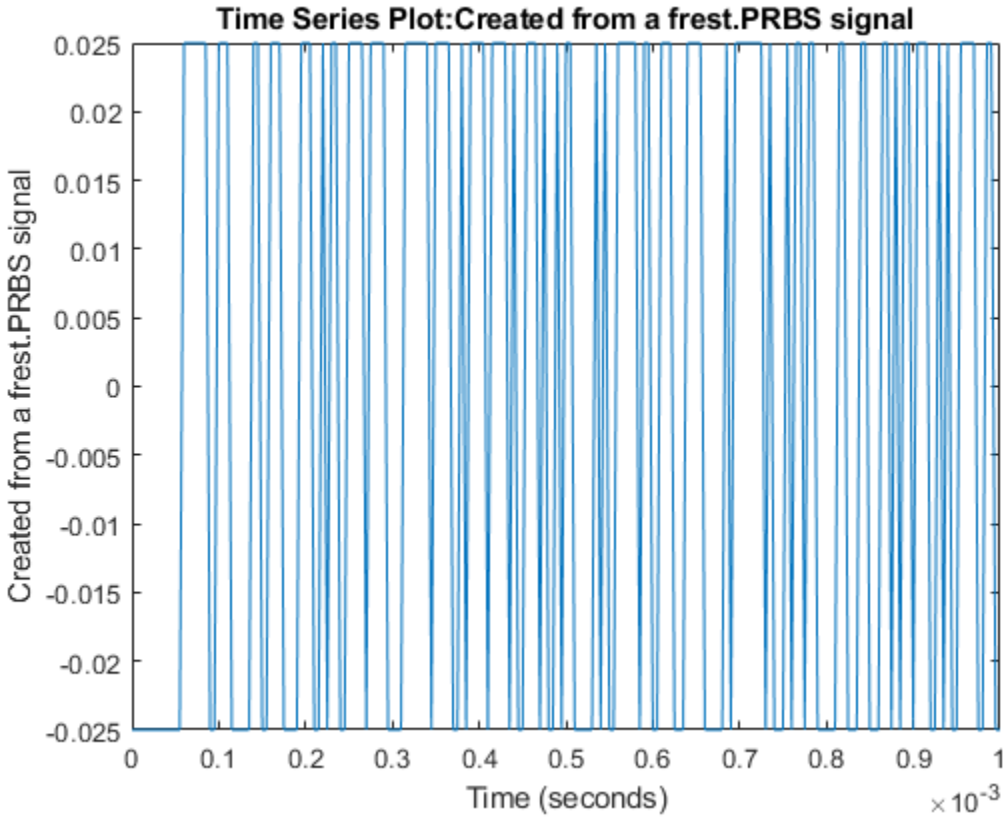
`frest.Chirp`

More About

- “Estimation Input Signals” on page 5-24
- “Sinestream Input Signals” on page 5-29
- “Frequency Response Estimation Using Simulation-Based Techniques” on page 5-74

PRBS Input Signals

A pseudorandom binary sequence (PRBS) is a periodic, deterministic signal with white-noise-like properties that shifts between two values. A PRBS signal is inherently periodic with a maximum period length of 2^n–1, where n is the PRBS order.



You can use a PRBS input signal for frequency-response estimation at the command line or in **Model Linearizer**. The estimation algorithm injects the PRBS signal at the input analysis point you specify for estimation, and measures the response at the output analysis point. PRBS signals are useful for estimating frequency responses for communications and power electronics applications.

Using PRBS input signals, you can:

- Reduce total estimation time compared to using sinestream input signals, while producing comparable estimation results.
- Obtain faster frequency response estimation with a higher frequency resolution than using chirp input signals.

When you create your PRBS input signal, specify the following parameters.

- Signal amplitude — The peak-to-peak range of the signal.
- Sample time — Set the sample time to match the sample time at the signals that correspond to the input and output linear analysis points.

- Signal order — The maximum length of the PRBS signal is 2^n–1, where n is the signal order.
- Number of periods — Number of periods N_p in the PRBS signal.

When specifying your PRBS signal parameters, consider the following:

- Set the amplitude such that the system is properly excited for your application. If the input amplitude is too large, the signal can deviate too far from the model operating point. If the input amplitude is too small, the PRBS signal is indistinguishable from noise and ripples in your model.
- For a given sample time, to increase the resolution over the low-frequency range, increase the order of the PRBS signal.
- For most frequency response estimation applications, use a single period. Doing so produces a flat frequency response across the frequency range of the signal.
- The frequency range of the generated PRBS signal is $[F_{min},F_{max}]$, where $F_{\min} = (F_N/N_p) \cdot (2/2^n - 1)$ and $F_{\max} = F_N$. F_N is the Nyquist frequency of the signal.

You can also create a PRBS signal with parameters based on the dynamics of a linear system, `sys`. For instance, if you have an exact linearization of your system, you can use it to initialize the parameters.

When you set the PRBS parameters using a linear system, the amplitude of the signal is `0.05` and the number of periods is `1`. To set the sample time and order of the signal, the software first selects a signal frequency range, $[F_{min},F_{max}]$, based on the dynamics of `sys`.

If `sys` is a discrete-time system, then:

- The sample time of the PRBS is equal to the sample time of `sys`.
- The order of the PRBS is as follows, where $\lceil . \rceil$ is the ceiling operator.

$$\text{Order} = \left\lceil \frac{\log\left(\frac{2\pi}{\text{Ts}\cdot\text{Fmin}}\right)}{\log(2)} \right\rceil$$

If `sys` is a continuous-time system, then:

- The sample time of the PRBS is

$$\text{Ts} = \frac{2\pi}{5 \cdot \text{Fmax}}$$

- The order of the PRBS is as follows, where $\lfloor . \rfloor$ is the floor operator.

$$\text{Order} = \left\lfloor \frac{\log\left(\frac{2\pi}{\text{Ts}\cdot\text{Fmin}}\right)}{\log(2)} \right\rfloor$$

Create PRBS Signals Using Model Linearizer

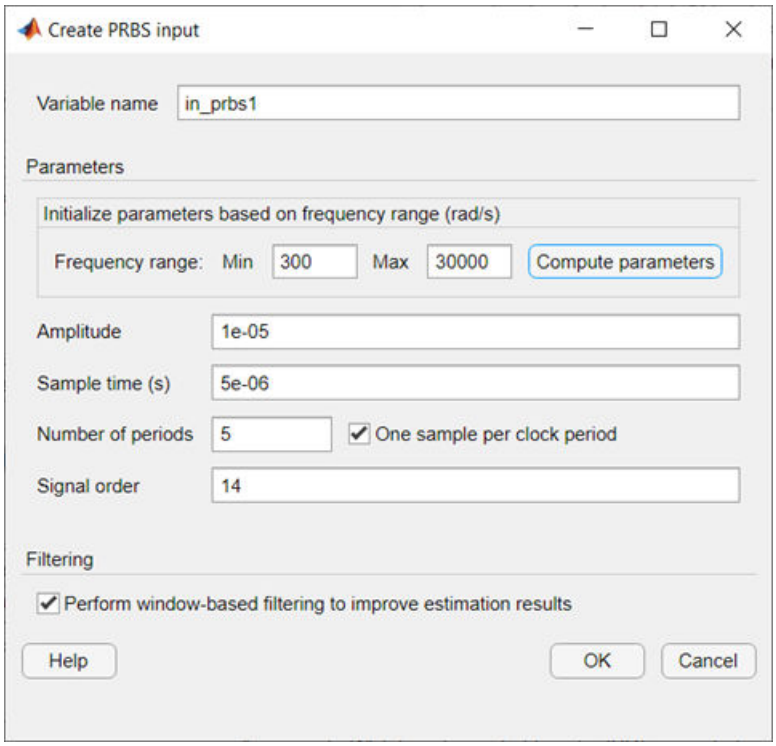
In the **Model Linearizer**, to use a PRBS input signal for estimation, on the **Estimation** tab, select **Input Signal** > **PRBS Pseudorandom Binary Sequence**.

In the Create PRBS input dialog box, specify the name of the PRBS signal object in **Variable Name**. You can then specify the parameters of your PRBS input signal using the following fields.

- **Amplitude** — Signal amplitude
- **Sample time** — Sample time
- **Number of periods** — Number of periods
- **Signal order** — Signal order

You can also automatically determine the parameters **Number of periods** and **Signal order** based on a frequency range of interest. Automatic parameter determination helps create an input signal that leads to an accurate frequency response over a specified frequency range.

To determine the parameters automatically, first set the **Sample time** parameter to match the sample time at the point of signal injection. Next, specify the frequency range of interest in rad/s using the **Min** and **Max** parameters, and then click **Compute parameters**.



Additionally, you can:

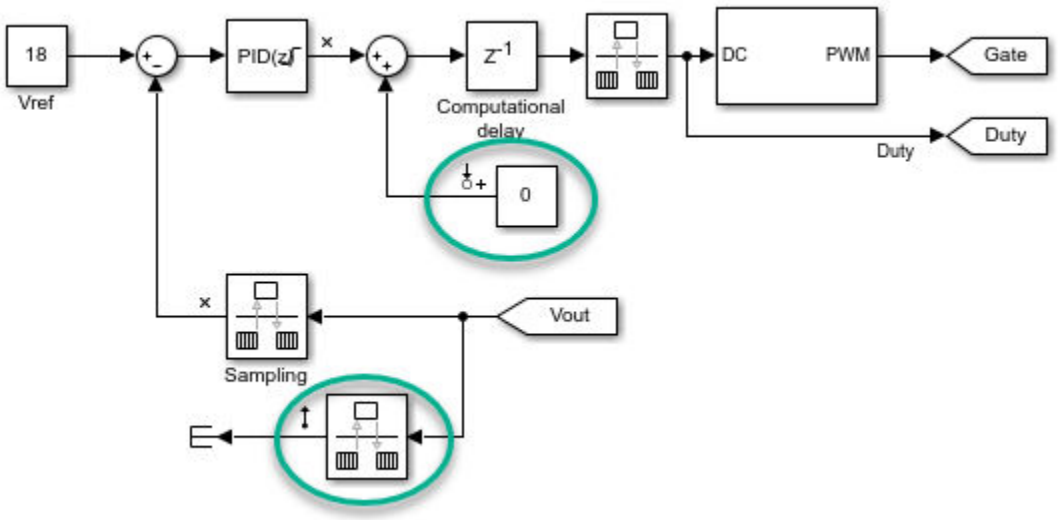
- Use the **One sample per clock period** parameter to specify whether the signal remains constant for one sample per clock period or multiple samples per clock period. Use this parameter if you have **Number of periods** > 1. By default, this option is enabled and the generated signal is constant over one sample. When you disable this option, the generated signal is constant for the specified number of samples.
- Use the **Perform window-based filtering to improve estimation results** parameter to apply Hann window-based filtering, which produces a smoother frequency response estimation result.

Create PRBS Signals Using MATLAB Code

To create a PRBS signal for estimation at the command line with `frestimate`, use a `frest.PRBS` object.

Improve Frequency Response Result at Low Frequencies

To improve the frequency response estimation result at low frequencies, you can use a different sample time other than the sample time in the original model. To do so, modify your model to use a Constant block at the input analysis point and a Rate Transition block at the output analysis point.



For both the Constant block and the Rate Transition block, specify a new sample time for your PRBS signal that is larger than the original sample time of the model.

The ability to change the sample time of the PRBS input signal provides an additional degree of freedom in the frequency response estimation process. By using a larger sample time than in the original model, you can obtain a higher resolution frequency response estimation result over the low-frequency range. Additionally, running estimation at lower sampling rate reduces processing requirements when deploying to hardware.

See Also

`frest.PRBS` | `frestimate`

More About

- “Estimation Input Signals” on page 5-24
- “Modify Estimation Input Signals” on page 5-40
- “Frequency Response Estimation Using Simulation-Based Techniques” on page 5-74
- “Frequency Response Estimation for Power Electronics Model Using Pseudorandom Binary Signal” on page 5-94

Modify Estimation Input Signals

When the frequency response estimation produces unexpected results, you can try modifying the input signal properties in the ways described in “Troubleshooting Frequency Response Estimation” on page 5-43.

Modify Sinestream Signal Using Model Linearizer

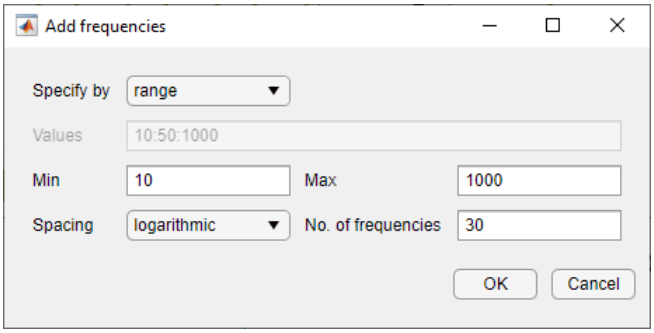
Add Frequency Points to Sinestream Input Signal

This example shows how to add frequency points to an existing sinestream input signal using the **Model Linearizer**.

1. Create a sinestream input signal, `in_sine1`, as shown in “Create Sinestream Signals Using Model Linearizer” on page 5-31.
2. Double-click `in_sine1` in the **Linear Analysis Workspace** area of the **Model Linearizer**.

   The Edit sinestream dialog box opens.
3. In the frequency content viewer, click in the Frequency content toolbar.

   The Add frequencies dialog box opens.



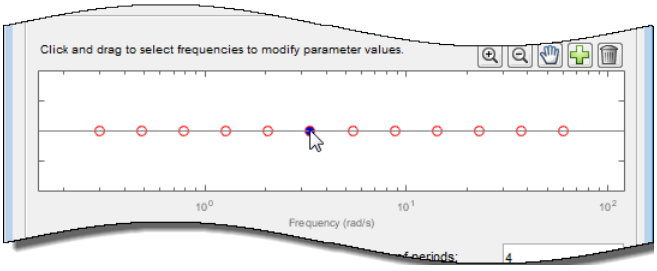
4. Enter the frequency range of the points to be added.
5. Click **OK** to add the specified frequency points to `in_sine1`.

Delete Frequency Point from Sinestream Input Signal

This example shows how to delete frequency points from an existing sinestream input signal using the **Model Linearizer**.

1. Create a sinestream input signal, `in_sine1`, as shown in “Create Sinestream Signals Using Model Linearizer” on page 5-31.
2. Double-click `in_sine1` in the **Linear Analysis Workspace** area of the **Model Linearizer**.

   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 points from the frequency content viewer.

5 Click **OK** to save the modified input signal.

Modify Parameters for a Frequency Point in Sinestream Input Signal

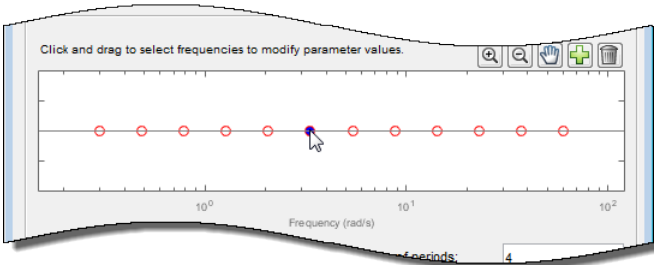
This example shows how to modify signal parameters of an existing sinestream input signal using the **Model Linearizer**.

1 Create a sinestream input signal, `in_sine1`, as shown in “Create Sinestream Signals Using Model Linearizer” on page 5-31.

2 Double-click `in_sine1` in the Linear Analysis Workspace area of the **Model Linearizer**.

The Edit sinestream dialog box opens.

3 In the frequency content viewer, select the frequency point(s) to delete.



The selected point(s) appears blue.

Tip To select multiple frequency points, click and drag across the frequency points of interest.

4 Enter the new values for the signal parameters.

If the parameter value is `<mixedvalue>`, the parameter has different values for some of the frequency points selected.

5 Click **OK** to save the modified input signal.

Modify Sinestream Signal Using MATLAB Code

For example, suppose that you used a sinestream input signal, and the output at a specific frequency did not reach steady state. In this case, you can modify the characteristics of the sinestream input at the corresponding frequency.

```
input.NumPeriods(index) = NewNumPeriods;
input.SettlingPeriods(index) = NewSettlingPeriods;
```

where `index` is the frequency value index of the sine wave you want to modify. `NewNumPeriods` and `NewSettlingPeriods` are the new values of `NumPeriods` and `SettlingPeriods`, respectively.

To modify several signal properties at a time, you can use the `set` command. For example:

```
input = set(input,'NumPeriods',NewNumPeriods,...
                  'SettlingPeriods',NewSettlingPeriods)
```

After modifying the input signal, repeat the estimation.

Troubleshooting Frequency Response Estimation

When to Troubleshoot

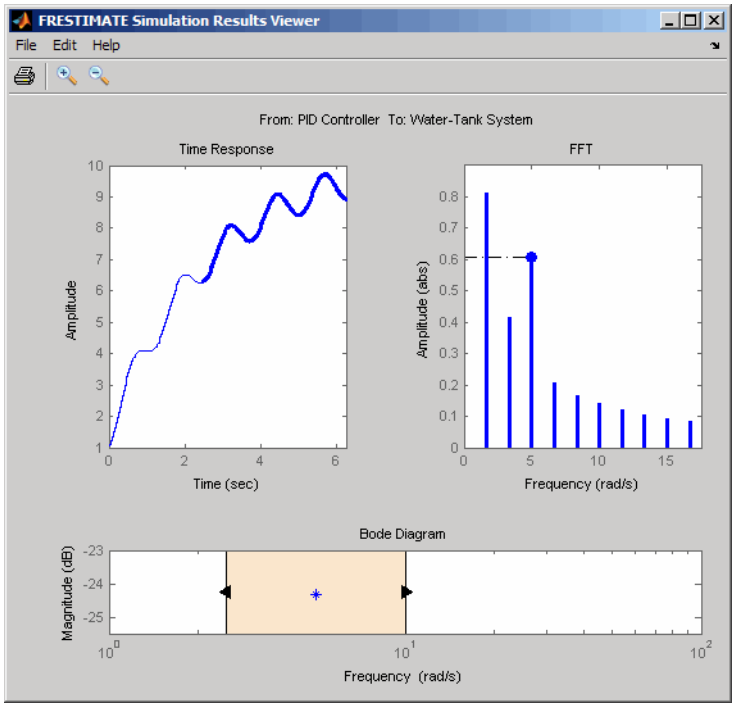
If, after analyzing your frequency response estimation, the frequency response plot does not match the expected behavior of your system, you can use the time response and FFT plots to help you improve the results.

If your estimation is slow or you run out of memory during estimation, see “Managing Estimation Speed and Memory” on page 5-68.

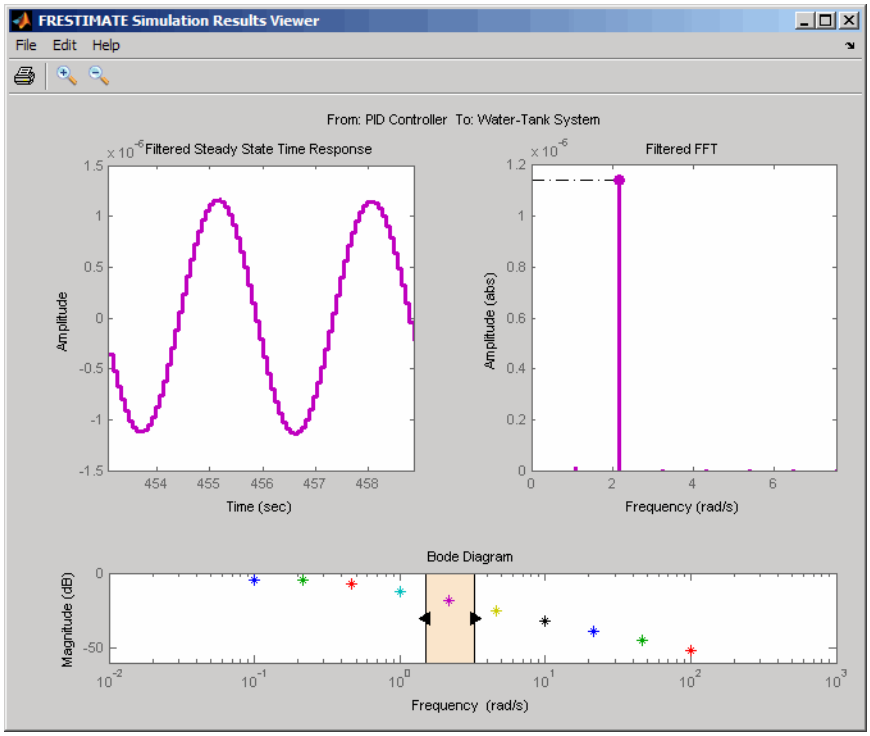
Time Response Not at Steady State

What Does This Mean?

This time response has not reached steady state.



This plot shows a steady-state time response.



Because frequency response estimation requires steady-state input and output signals, transients produce inaccurate estimation results.

For sinestream input signals, transients sometimes interfere with the estimation either directly or indirectly through spectral leakage. For chirp input signals, transients interfere with estimation.

How Do I Fix It?

Possible Cause	Action
Model cannot initialize to steady state.	• Increase the number of periods for frequencies that do not reach steady state by changing the `NumPeriods` and `SettlingPeriods` properties. See “Modify Estimation Input Signals” on page 5-40. • Disable all time-varying source blocks in your model and repeat the estimation. See “Effects of Time-Varying Source Blocks on Frequency Response Estimation” on page 5-53.
(Sinestream input) Not enough periods for the output to reach steady state.	• Increase the number of periods for frequencies that do not reach steady state by changing the `NumPeriods` and `SettlingPeriods`. See “Modify Estimation Input Signals” on page 5-40. • 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.

Possible Cause	Action
(Chirp input) Signal sweeps through the frequency range too quickly.	Increase the simulation time by increasing `NumSamples`. See “Modify Estimation Input Signals” on page 5-40.

After you try the suggested actions, recompute the estimation either:

- At all frequencies
- In a particular frequency range (only for sinestream input signals)

To recompute the estimation in a particular frequency range:

1. Determine the frequencies for which you want to recompute the estimation results. Then, extract a portion of the sinestream input signal at these frequencies using `fselect`.

   For example, these commands extract a sinestream input signal between 10 and 20 rad/s from the input signal `input`:

   ```
   input2 = fselect(input,10,20);
   ```

2. Modify the properties of the extracted sinestream input signal `input2`, as described in “Modify Estimation Input Signals” on page 5-40.
3. Estimate the frequency response `sysest2` with the modified input signal using `frestimate`.
4. Merge the original estimated frequency response `sysest` and the recomputed estimated frequency response `sysest2`:

   a. Remove data from `sysest` at the frequencies in `sysest2` using `fdel`.

   ```
   sysest = fdel(sysest,input2.Frequency)
   ```

   b. Concatenate the original and recomputed responses using `fcat`.

   ```
   sys_combined = fcat(sysest2,sysest)
   ```

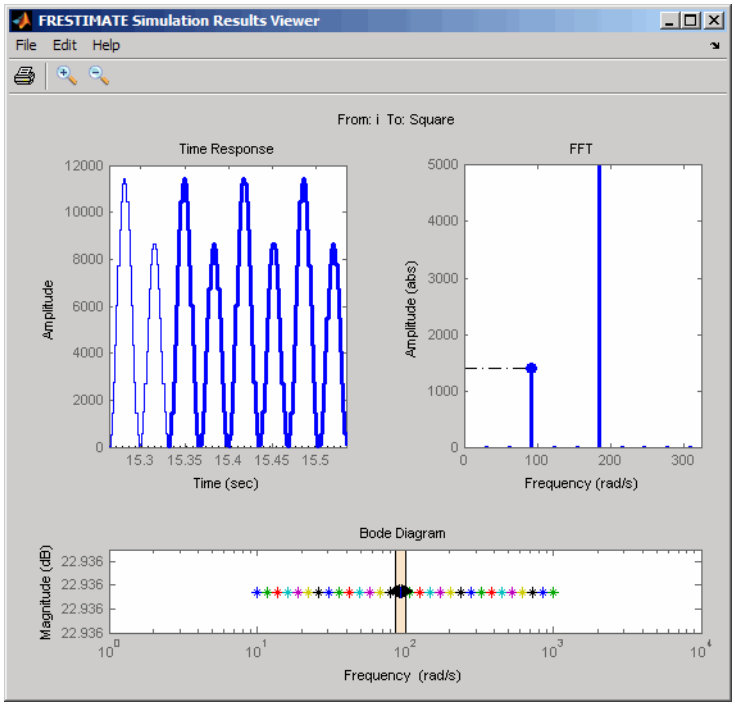
Analyze the recomputed frequency response, as described in “Analyze Estimated Frequency Response” on page 5-18.

For an example of frequency response estimation with time-varying source blocks, see “Effects of Time-Varying Source Blocks on Frequency Response Estimation” on page 5-53

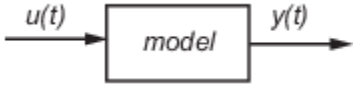
FFT Contains Large Harmonics at Frequencies Other than the Input Signal Frequency

What Does This Mean?

When the FFT plot shows large amplitudes at frequencies other than the input signal, your model is operating outside the linear range. This condition can cause problems when you want to analyze the response of your linear system to small perturbations.



For models operating in the linear range, the input amplitude A_1 in *y(t)* must be larger than the amplitudes of other harmonics, A_2 and A_3.



$$u(t) = A_1\sin(\omega_1 + \phi_1)$$
$$y(t) = A_1\sin(\omega_1 + \phi_1) + A_2\sin(\omega_2 + \phi_2) + A_3\sin(\omega_3 + \phi_3) + \ldots$$

How Do I Fix It?

Adjust the amplitude of your input signal to decrease the impact of other harmonics, and repeat the estimation. Typically, you should decrease the input amplitude level to keep the model operating in the linear range.

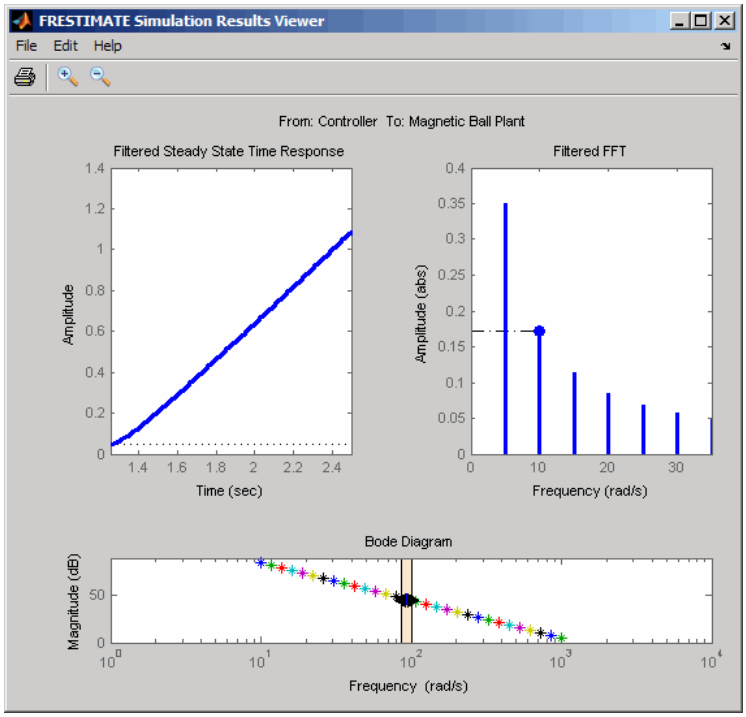
For more information about modifying signal amplitudes, see one of the following:

- `frest.Sinestream`
- `frest.Chirp`
- `frest.PRBS`
- “Modify Estimation Input Signals” on page 5-40

Time Response Grows Without Bound

What Does This Mean?

When the time response grows without bound, frequency response estimation results are inaccurate. Frequency response estimation is only accurate close to the operating point.



How Do I Fix It?

Try the suggested actions listed the table and repeat the estimation.

Possible Cause	Action
Model is unstable.	You cannot estimate the frequency response using `frestimate`. Instead, use exact linearization to get a linear representation of your model. See “Linearize Simulink Model at Model Operating Point” on page 2-54 or the `linearize` reference page.
Stable model is not at steady state.	Disable all source blocks in your model, and repeat the estimation using a steady-state operating point. See “Compute Steady-State Operating Points” on page 1-5.
Stable model captures a growing transient.	If the model captures a growing transient, increase the number of periods in the input signal by changing `NumPeriods`. Repeat the estimation using a steady-state operating point.

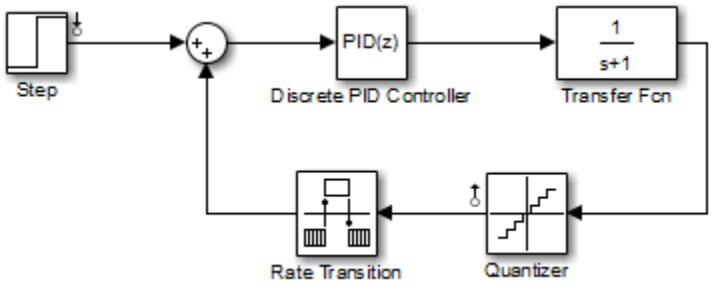
Time Response Is Discontinuous or Zero

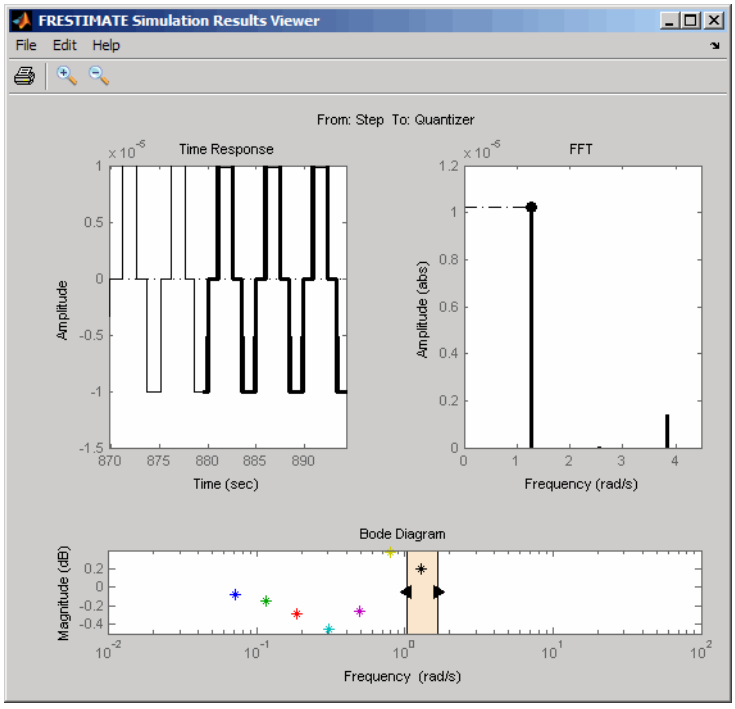
What Does This Mean?

Discontinuities or noise in the time response indicate that the amplitude of your input signal is too small to overcome the effects of the discontinuous blocks in your model. Examples of discontinuous blocks include Quantizer, Backlash, and Dead Zones.

If you used a sinestream input signal and estimated with filtering, turn filtering off in the Simulation Results Viewer to see the unfiltered time response.

The following model with a Quantizer block shows an example of the impact of an input signal that is too small. When you estimate this model, the unfiltered simulation output includes discontinuities.

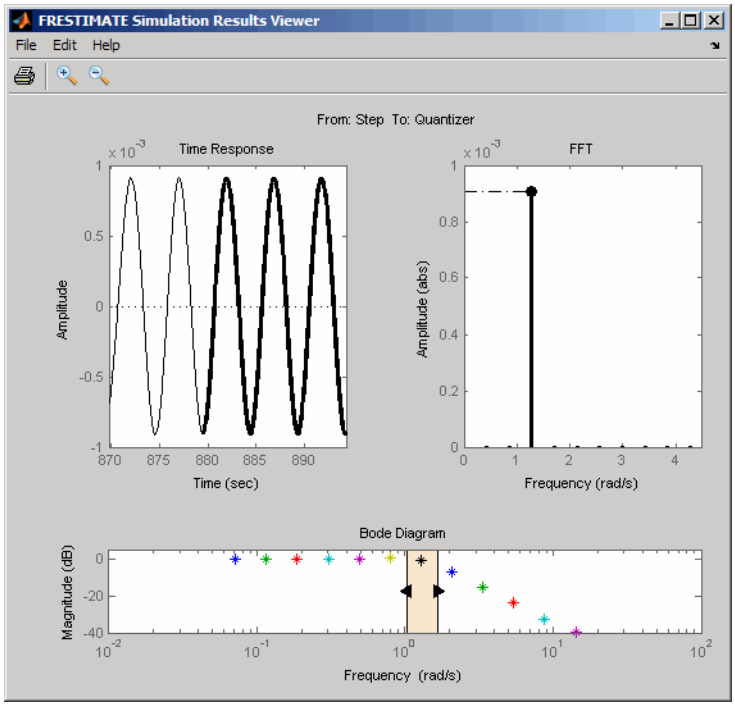




How Do I Fix It?

Increase the amplitude of your input signal, and repeat the estimation.

With a larger amplitude, the unfiltered simulated output of the model with a Quantizer block is smooth.



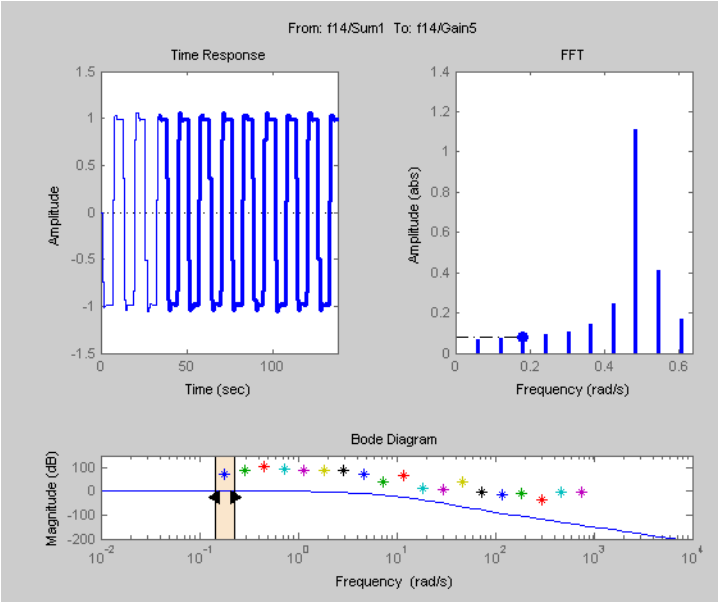
For more information about modifying signal amplitudes, see one of the following:

- `frest.Sinestream`
- `frest.Chirp`
- “Modify Estimation Input Signals” on page 5-40

Time Response Is Noisy

What Does This Mean?

When the time response is noisy, frequency response estimation results may be biased.



How Do I Fix It?

`frestimate` does not support estimating frequency response estimation of Simulink models with blocks that model noise. Locate such blocks with `frest.findSources` and disable them using the `BlocksToHoldConstant` option of `frestimate`.

If you need to estimate a model with noise, use `frestimate` to simulate an output signal from your Simulink model for estimation—without modifying your model. Then, use the Signal Processing Toolbox™ or System Identification Toolbox software to estimate a model.

To simulate the output of your model in response to a specified input signal:

1. Create a random input signal. For example:

   ```
   in = frest.Random('Ts',0.001,'NumSamples',1e4);
   ```

   You can also specify your own custom signal as a `timeseries` object. For example:

   ```
   t = 0:0.001:10;
   y = sin(2*pi*t);
   in_ts = timeseries(y,t);
   ```

2. Simulate the model to obtain the output signal. For example:

   ```
   [sysest,simout] = frestimate(model,op,io,in_ts)
   ```

   The second output argument of `frestimate`, `simout`, is a `Simulink.Timeseries` object that stores the simulated output. `in_ts` is the corresponding input data.

3. Generate `timeseries` objects before using with other MathWorks® products:

   ```
   input = generateTimeseries(in_ts);
   output = simout{1}.Data;
   ```

   You can use data from `timeseries` objects directly in Signal Processing Toolbox software, or convert these objects to System Identification Toolbox data format. For examples, see “Estimate

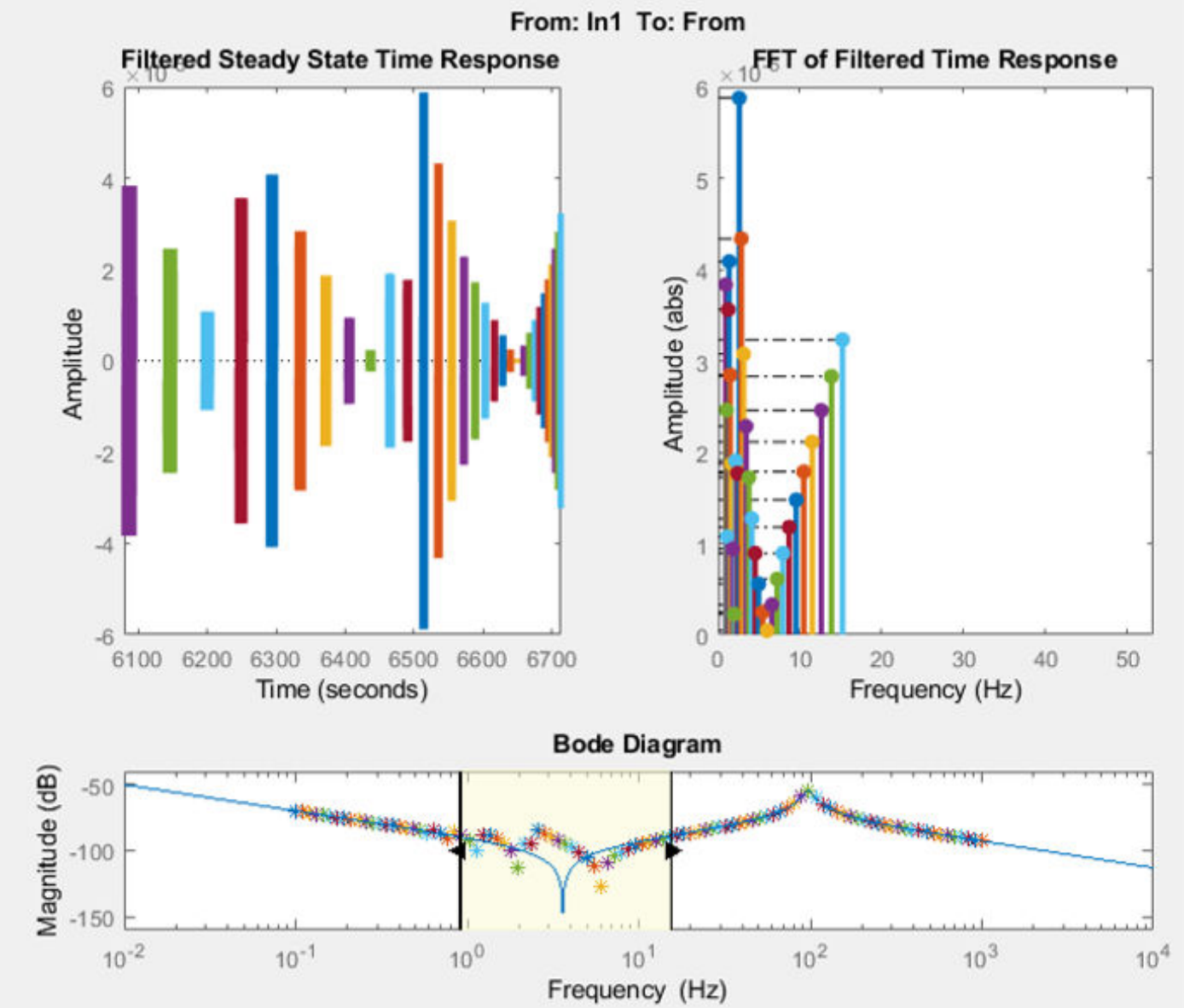
Frequency Response Models with Noise Using Signal Processing Toolbox” on page 5-63 and “Estimate Frequency Response Models with Noise Using System Identification Toolbox” on page 5-65.

For a related example, see “Disable Noise Sources During Frequency Response Estimation” on page 5-60.

Time Response Shows Harmonics That Do Not Change Smoothly

What Does This Mean?

The estimated frequency response result does not match the linear system bode plot, possibly only over a certain frequency range. When the time responses show magnitudes that do not change smoothly, additional frequency components are affecting the response. These additional frequency components come from the defined input signal.



When the input signal is created using `frest.Sinestream`, the default value of `SamplesPerPeriod` is `40`. This default setting produces a coarse input signal, which causes the mismatch in the bode plot.

How Do I Fix It?

To create a smoother input signal, increase the value of the `SamplesPerPeriod` setting. For more information about setting `SamplesPerPeriod`, see the following:

- `frest.Sinestream`
- “Modify Estimation Input Signals” on page 5-40

Effects of Time-Varying Source Blocks on Frequency Response Estimation

Setting Time-Varying Sources to Constant for Estimation Using Model Linearizer

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)
   ```

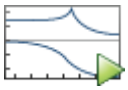
2. Linearize the model.

   a. Set the `Engine Model` block to normal mode for accurate linearization.

   ```
   set_param('scdspeed_ctrlloop/Engine Model','SimulationMode','Normal')
   ```

   b. Open the **Model Linearizer** for the model.

   In the Simulink model window, in the **Apps** gallery, click **Model Linearizer**.

   c. Click  **Bode** to linearize the model and generate a Bode plot of the result.

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

3. Create an input sinestream signal for the estimation.

   a. Open the Create sinestream input dialog box.

   On the **Estimation** tab, in the **Input Signal** drop-down list, select **Sinestream**.

   b. Open the Add frequencies dialog box.

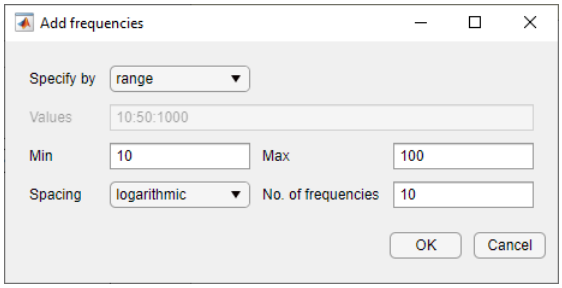
   Click .

   c. Specify the input sinestream frequency range and number of frequency points.

   Enter `10` in the **Min** box.

   Enter `100` in the **Max** box.

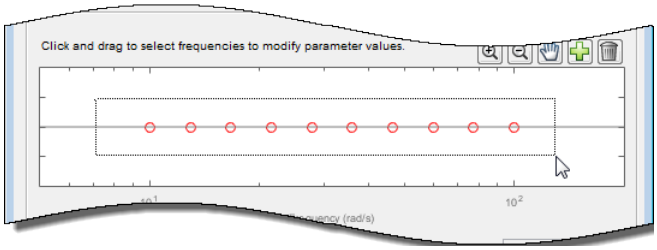
   Enter `10` in the box for the number of frequency points.



Click **OK**.

The added points are visible in the frequency content viewer of the Create sinestream input dialog box.

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



e Specify input sinestream parameters.

Change the **Number of periods** and **Settling periods** to ensure that the model reaches steady-state for each frequency point in the input sinestream.

Enter `30` in the **Number of periods** box.

Enter `25` in the **Settling periods** box.

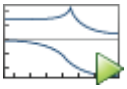
f Create the input sinestream.

Click **OK**. The new input signal, `in_sine1`, appears in the **Linear Analysis Workspace**.

4 Set the Diagnostic Viewer to open when estimation is performed.

On the **Estimation** tab, select **Diagnostic Viewer**.

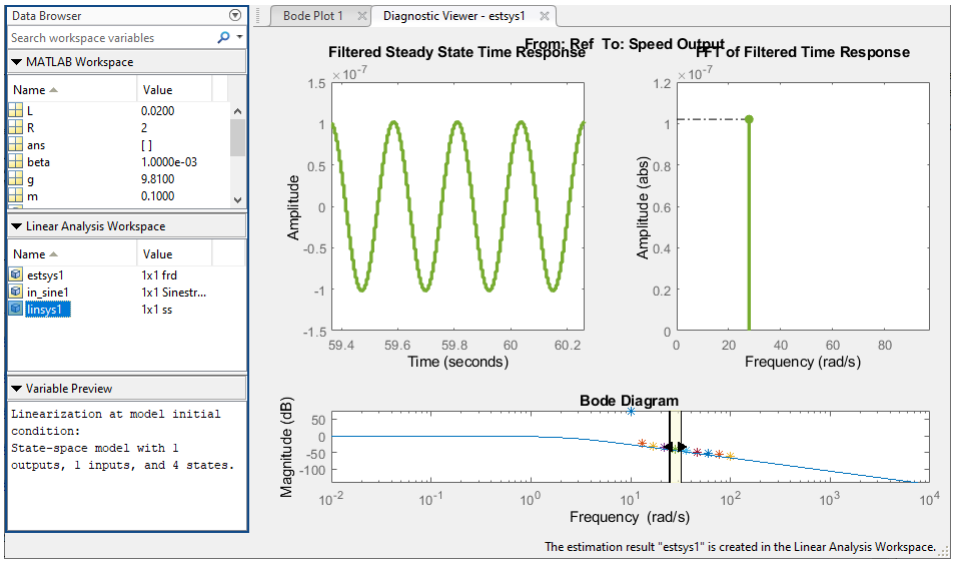
5 Estimate the frequency response for the model.

Click  **Bode Plot 1** to estimate the frequency response. The Diagnostic Viewer appears in the document area and the estimated system `estsys1` appears in the **Linear Analysis Workspace**.

6 Compare the estimated model and the linearized model.

a Click on the **Diagnostic Viewer - estsys1** tab in the plot area of the **Model Linearizer**.

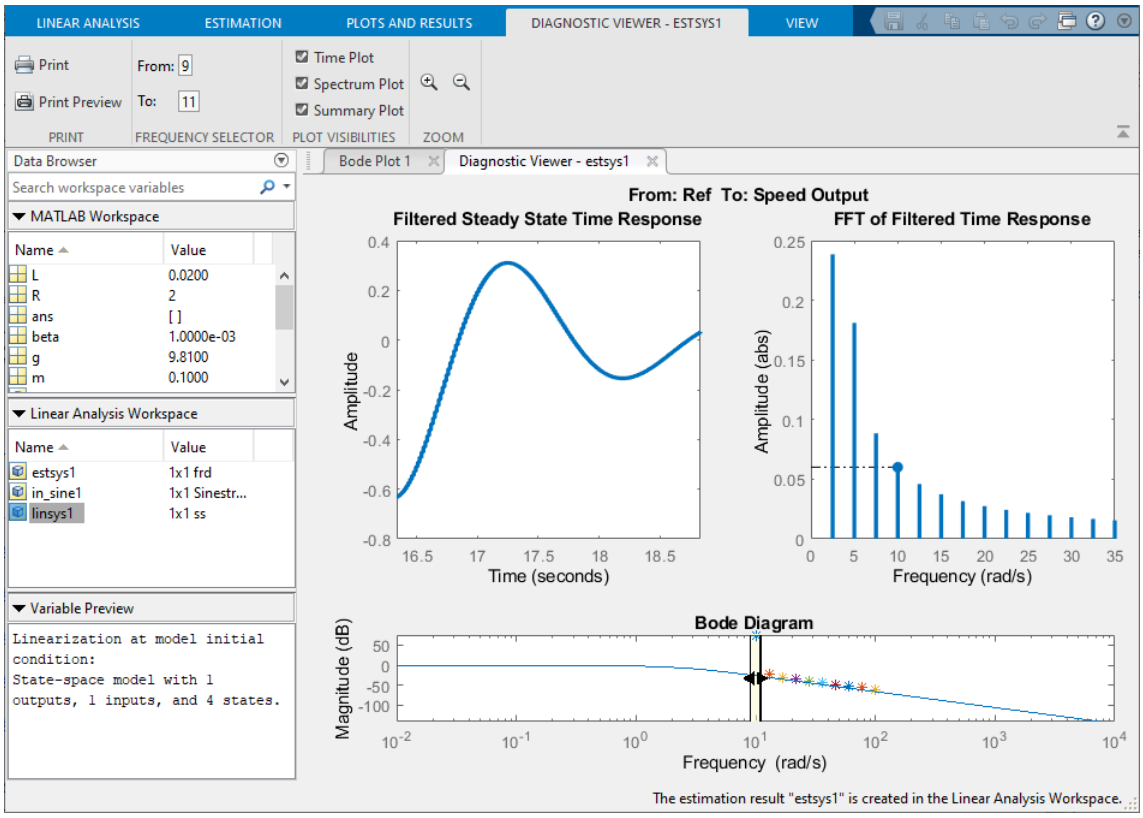
b Drag `linsys1` from the **Linear Analysis Workspace** onto the Diagnostic Viewer to add `linsys1` to the **Bode Diagram**.



c Configure the Diagnostic Viewer to show only the frequency point where the estimation and linearization results do not match.

Click the **Diagnostic Viewer** tab.

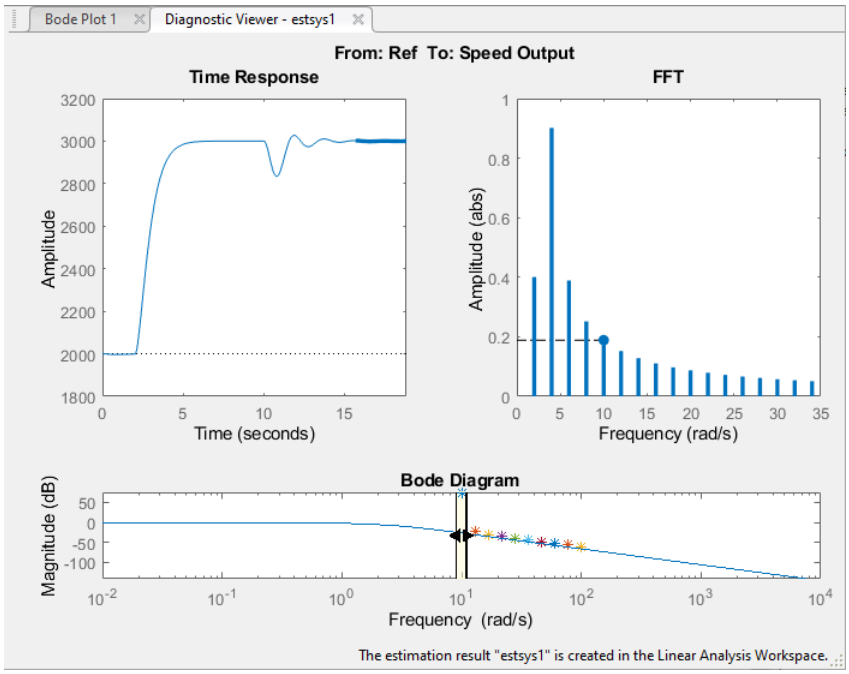
In the **Frequency Selector** section, enter 9 in the **From** box and 11 in the **To** box to set the frequency range that is analyzed in the Diagnostic Viewer.



The **Filtered Steady State Time Response** plot shows a signal that is not sinusoidal.

d View the unfiltered time response.

Right-click the **Filtered Steady State Time Response** plot and clear the **Show filtered steady state output only** option.

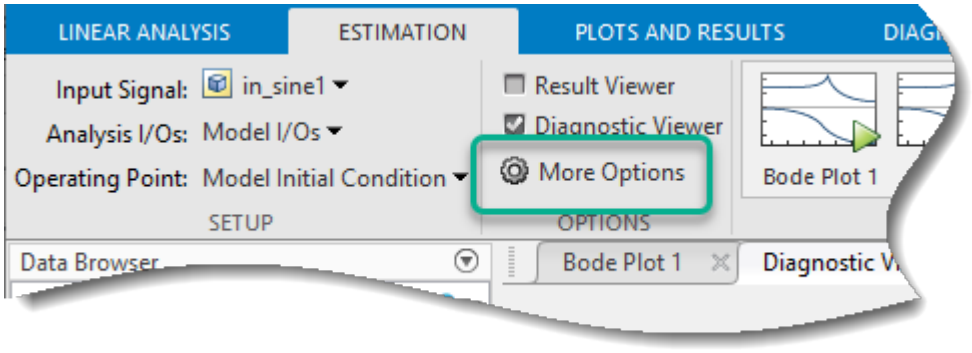


The step input and external disturbances drive the model away from the operating point used to linearize the model, which prevents the response from reaching steady-state. To correct this problem, find and disable the time-varying source blocks that interfere with the estimation. Then estimate the frequency response of the model again.

7 Find and disable the time-varying sources within the model.

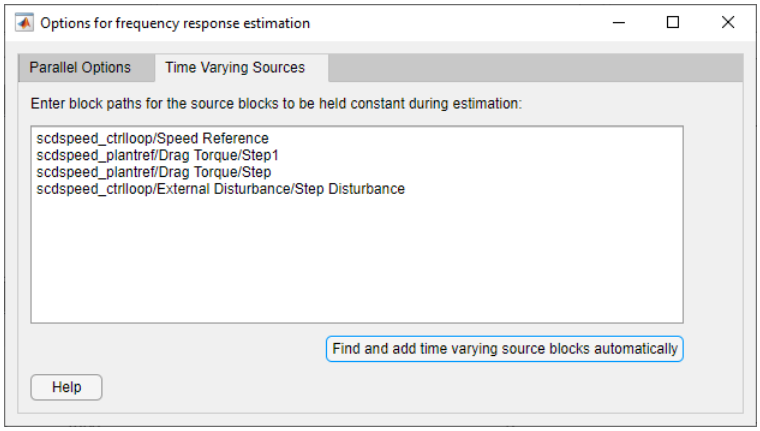
a Open the Options for frequency response estimation dialog box.

On the **Estimation** tab, in the **Options** section, click **More Options**.



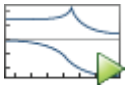
b In the Options for frequency response estimation dialog box, on the **Time Varying Sources** tab, click **Find and add time varying source blocks automatically**.

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.



Close the dialog box.

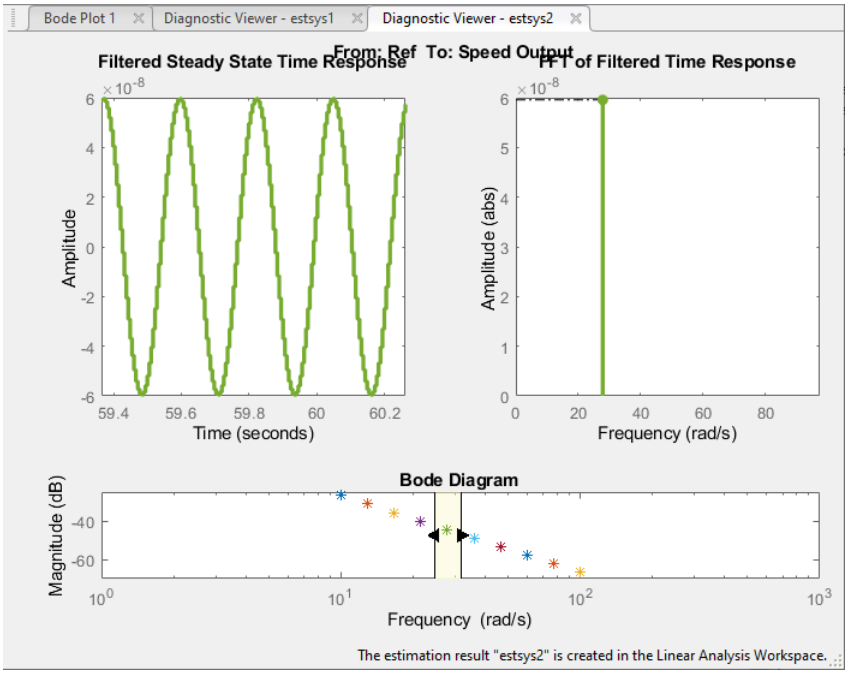
8 Estimate the frequency response for the model.

On the estimation tab, click  **Bode Plot 1** to estimate the frequency response. The estimated system `estsys2`, appears in the **Linear Analysis Workspace**.

9 Compare the newly estimated model and the linearized model.

Click on the **Diagnostic Viewer - estsys2** tab in the plot area of the **Model Linearizer**.

Click and drag `linsys1` onto the Diagnostic Viewer.



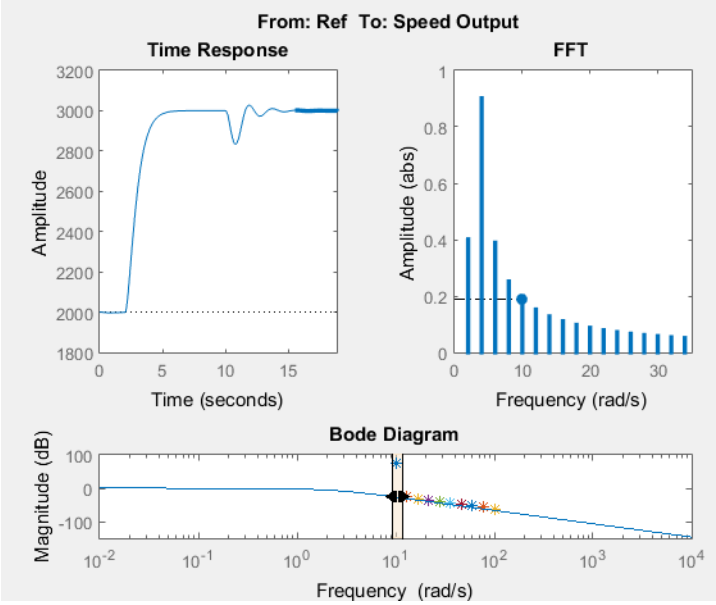
The frequency response obtained by holding the time-varying sources constant matches the exact linearization results.

Setting Time-Varying Sources to Constant for Estimation (MATLAB Code)

Compare the linear model obtained using exact linearization techniques with the estimated frequency response:

```
% Open the model
mdl = 'scdspeed_ctrlloop';
open_system(mdl)
io = getlinio(mdl);

% Set the model reference to normal mode for accurate linearization
set_param('scdspeed_ctrlloop/Engine Model','SimulationMode','Normal')

% Linearize the model
sys = linearize(mdl,io);

% Estimate the frequency response between 10 and 100 rad/s
in = frest.Sinestream('Frequency',logspace(1,2,10),'NumPeriods',30,'SettlingPeriods',25);
[sysest,simout] = frestimate(mdl,io,in);

% Compare the results
frest.simView(simout,in,sysest,sys)
```

The linearization results do not match the estimated frequency response for the first two frequencies. To view the unfiltered time response, right-click the time response plot, and clear **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);
```

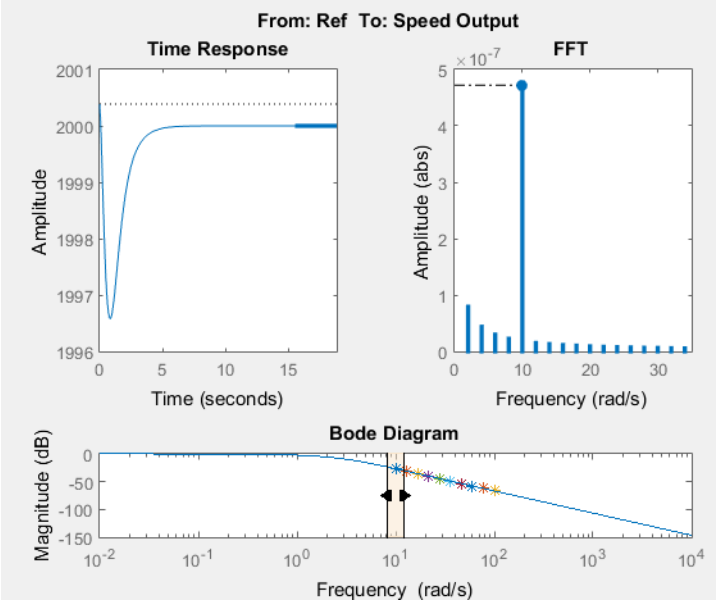
Create a `frestimate` options set to disable the blocks.

```
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 clear **Show filtered steady state output only**.



Limitations

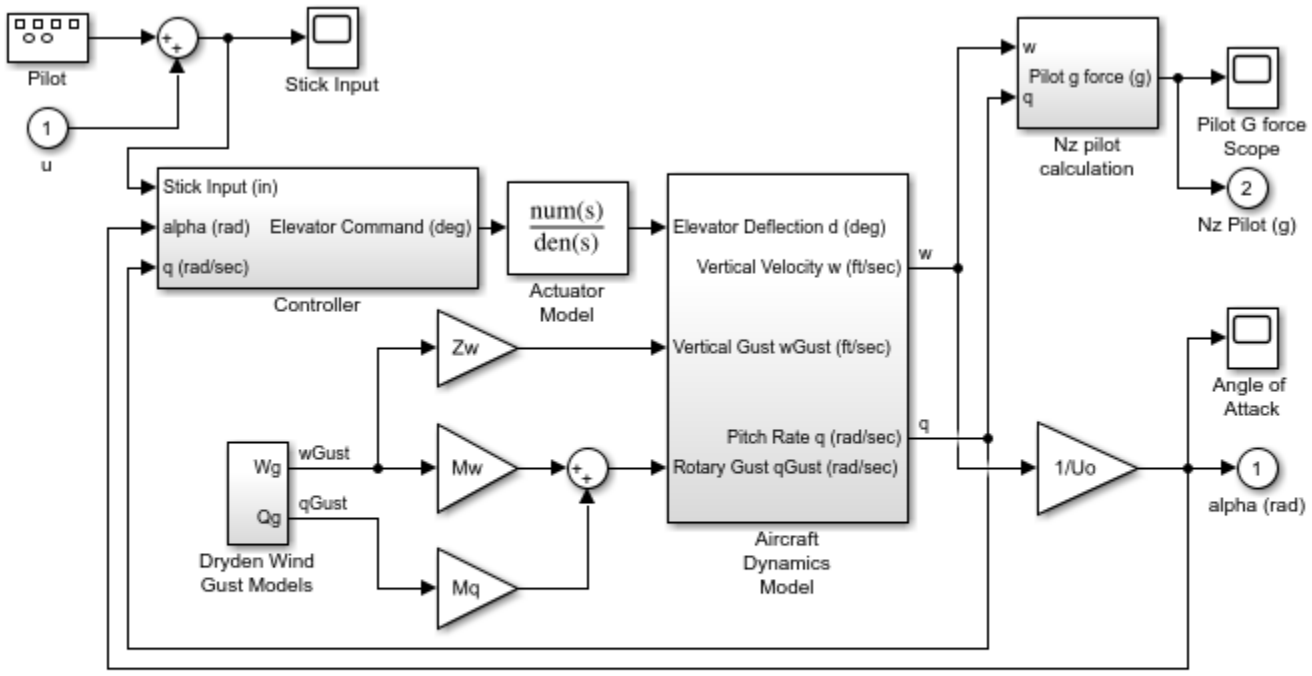
Some time-varying source blocks might not be found by the algorithm. If the internal signal path of a block does not contain a block with no input port, that block is not reported by the `frest.findSources` function or in the app. To bring the model to steady state, replace the source block with a Constant block, or a different source block.

Disable Noise Sources During Frequency Response Estimation

This example shows how to disable noise sources in your Simulink® model during frequency response estimation. Such noise sources can interfere with the signal at the linearization output points and produce inaccurate estimation results.

Open the model.

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



Copyright 1990-2012 The MathWorks, Inc.

Specify linearization input and output points.

```
io(1) = linio('scdplane/Sum1',1);
io(2) = linio('scdplane/Gain5',1,'output');
```

Linearize the model and create a sinestream estimation input signal based on the dynamics of the resulting linear system.

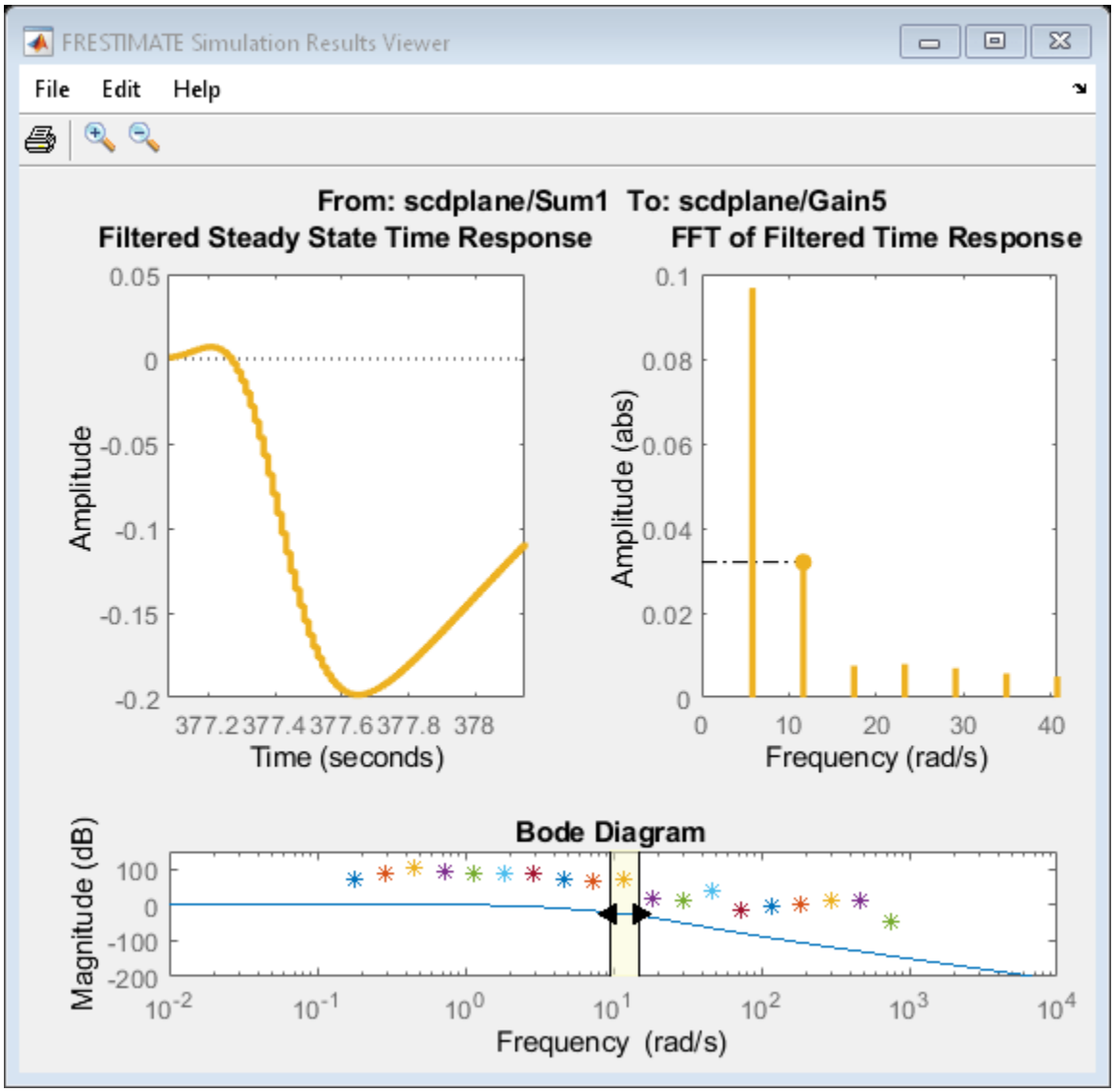
```
sys = linearize(mdl,io);
in = frest.Sinestream(sys);
```

Estimate frequency response.

```
[sysest,simout] = frestimate(mdl,io,in);
```

Compare the estimated frequency response to the exact linearization result.

```
frest.simView(simout,in,sysest,sys)
```



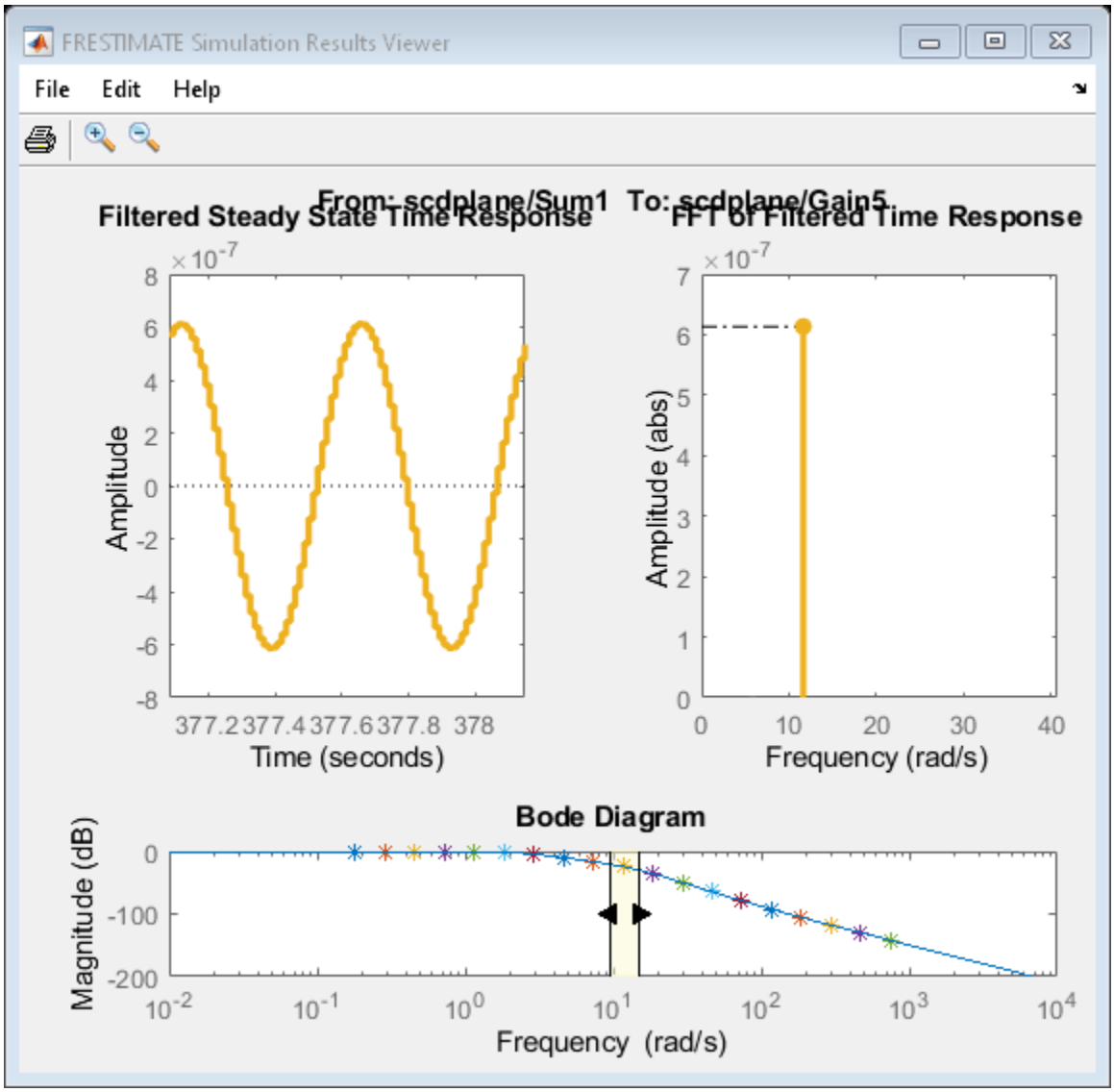
In the **Bode Diagram**, the estimated frequency response does not match the response of the exact linearization. This result is due to the effects of the Pilot and Wind Gust Disturbance blocks in the model. To view the effects of the noise on the time response at a given frequency, right-click the time response plot and make sure **Show filtered steady state output only** is selected.

Locate the source blocks in the model.

```
srcblks = frest.findSources(mdl,io);
```

Repeat the frequency response estimation with the source blocks disabled.

```
opts = frestimateOptions('BlocksToHoldConstant',srcblks);
[sysest,simout] = frestimate(mdl,io,in,opts);
frest.simView(simout,in,sysest,sys)
```



The resulting frequency response matches the exact linearization results.

See Also

`frestimate` | `frestimateOptions` | `frest.findSources` | `frest.simView`

More About

- “Effects of Time-Varying Source Blocks on Frequency Response Estimation” on page 5-53

Estimate Frequency Response Models with Noise Using Signal Processing Toolbox

Open the Simulink model, and specify which portion of the model to linearize:

```
load_system('magball')
io(1) = linio('magball/Desired Height',1);
io(2) = linio('magball/Magnetic Ball Plant',1,'output');
```

Create a random input signal for simulation:

```
in = frest.Random('Ts',0.001,'NumSamples',1e4);
```

Linearize the model at a steady-state operating point:

```
op = findop('magball',operspec('magball'),...
            findopOptions('DisplayReport','off'));
sys = linearize('magball',io,op);
```

Simulate the model to obtain the output at the linearization output point:

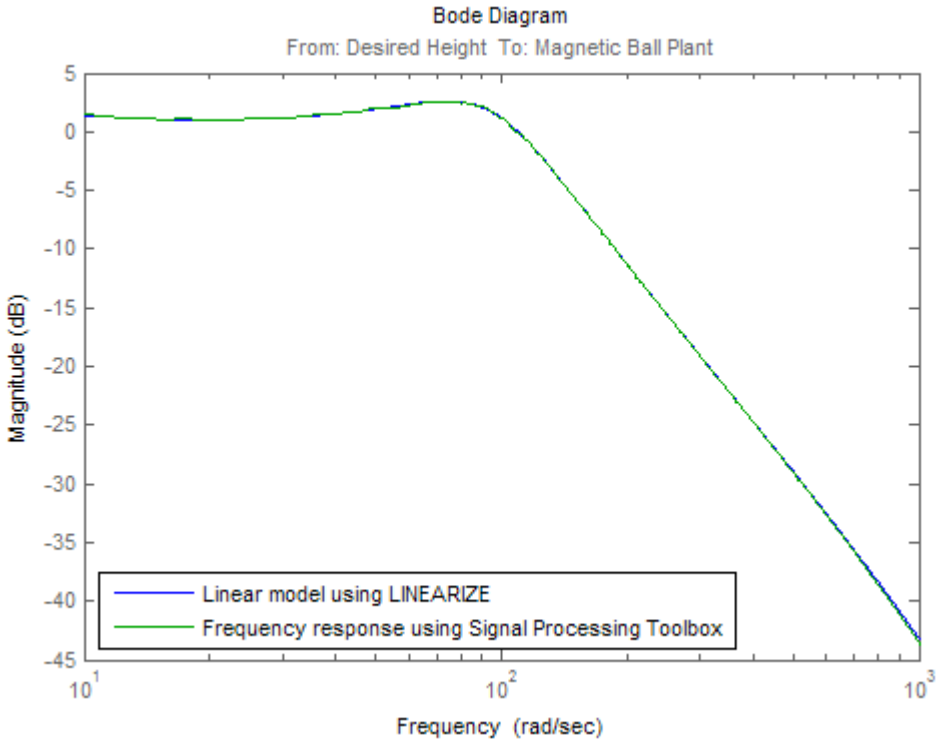
```
[sysest,simout] = frestimate('magball',io,in,op);
```

Estimate a frequency response model using Signal Processing Toolbox software, which includes windowing and averaging:

```
input = generateTimeseries(in);
output = detrend(simout{1}.Data,'constant');
[Txy,F] = tfestimate(input.Data(:),...
          output,hanning(4000),[],4000,1/in.Ts);
systfest = frd(Txy,2*pi*F);
```

Compare the results of analytical linearization and `tfestimate`:

```
ax = axes;
h = bodeplot(ax,sys,'b',systfest,'g',systfest.Frequency);
setoptions(h,'Xlim',[10,1000],'PhaseVisible','off')
legend(ax,'Linear model using LINEARIZE','Frequency response using Signal Processing Toolbox',...
           'Location','SouthWest')
```



In this case, the Signal Processing Toolbox command `tfestimate` gives a more accurate estimation than `frestimate` due to windowing and averaging.

Estimate Frequency Response Models with Noise Using System Identification Toolbox

Open the Simulink model, and specify which portion of the model to linearize:

```
load_system('magball');
io(1) = linio('magball/Desired Height',1);
io(2) = linio('magball/Magnetic Ball Plant',1,'output');
```

Compute the steady-state operating point, and linearize the model:

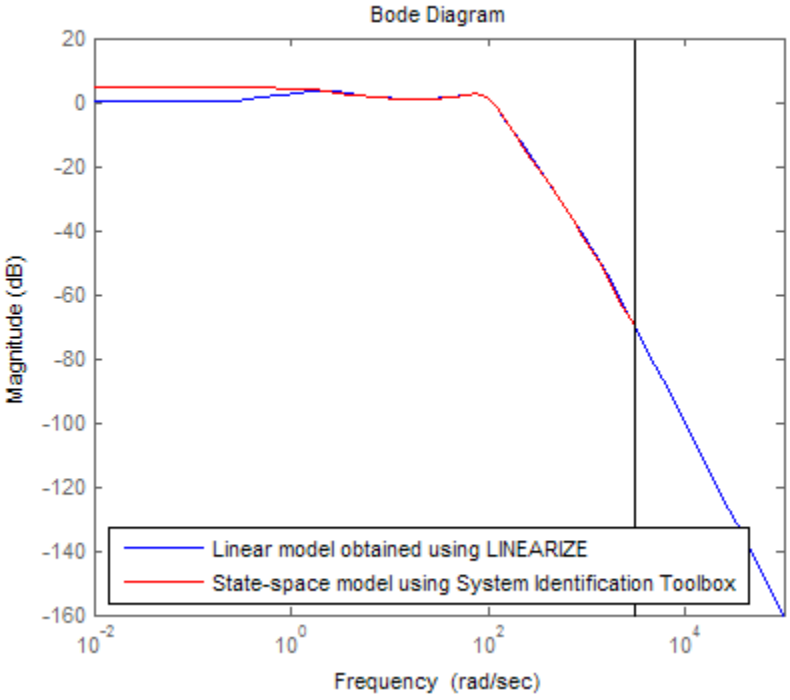
```
op = findop('magball',operspec('magball'),...
                      findopOptions('DisplayReport','off'));
sys = linearize('magball',io,op);
```

Create a chirp signal, and use it to estimate the frequency response:

```
in = frest.Chirp('FreqRange',[1 1000],...
                             'Ts',0.001,...
                             'NumSamples',1e4);
[~,simout] = frestimate('magball',io,op,in);
```

Use System Identification Toolbox software to estimate a fifth-order, state-space model. Compare the results of analytical linearization and the state-space model:

```
input = generateTimeseries(in);
output = simout{1}.Data;
data = iddata(output,input.Data(:),in.Ts);
sys_id = n4sid(detrend(data),5,'cov','none');
bodemag(sys,ss(sys_id('measured')),'r')
legend('Linear model obtained using LINEARIZE',...
       'State-space model using System Identification Toolbox',...
       'Location','SouthWest')
```

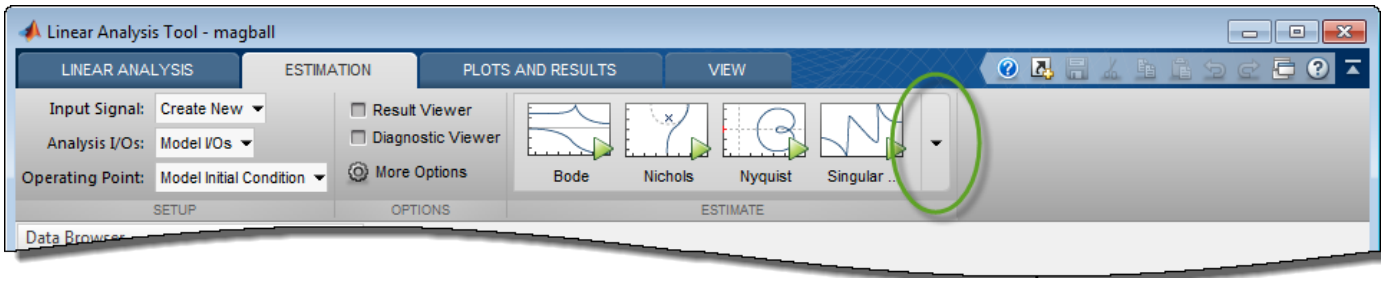

Bode Diagram
Magnitude (dB)
20
0
-20
-40
-60
-80
-100
-120
-140
-160
10^-2
10^0
10^2
10^4
Frequency (rad/sec)
Linear model obtained using LINEARIZE
State-space model using System Identification Toolbox

Generate MATLAB Code for Repeated or Batch Frequency Response Estimation

This topic shows how to generate MATLAB code for frequency response estimation from the **Model Linearizer**. 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 **Model Linearizer**, in the **Estimation** tab, interactively configure the input signal, analysis I/Os, operating point, and other parameters for frequency response estimation.
2. Click ▾ to expand the gallery.



3. Select the type of code you want to generate:
   - `Script` — Generate a MATLAB script that uses your configured parameter values. Select this option when you want to repeat the same frequency response estimation at the MATLAB command line.
   - `Function` — Generate a MATLAB function that takes analysis I/Os, operating points, and input signals as input arguments. Select this option when you want to perform multiple frequency response estimations using different parameter values (batch estimation).

To use a generated MATLAB function for batch estimation, you can create a MATLAB script with a `for` loop that cycles through values of the parameter you want to vary. Call the generated MATLAB function in each iteration of the loop.

Managing Estimation Speed and Memory

Ways to Speed up Frequency Response Estimation

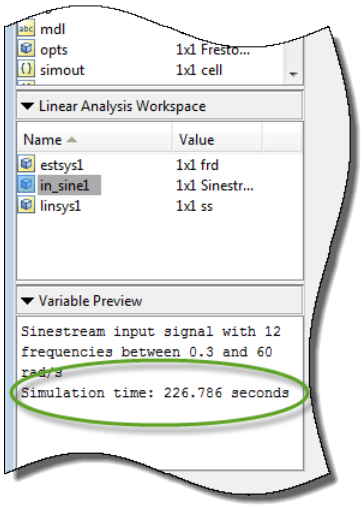
The most time consuming operation during frequency response estimation is the simulation of your Simulink model. You can try to speed up the estimation using any of the following ways:

- Reducing simulation stop time
- Specifying accelerator mode
- Using parallel computing

Reducing Simulation Stop Time

The time it takes to perform frequency response estimation depends on the simulation stop time.

To obtain the simulation stop time, in the **Model Linearizer**, in the **Linear Analysis Workspace**, select the input signal. The simulation time will be displayed in the **Variable Preview**.



To obtain the simulation stop time from the input signal using MATLAB Code:

```
tfinal = getSimulationTime(input)
```

where `input` is the input signal. The simulation stop time, `tfinal`, serves as an indicator of the frequency response estimation duration.

You can reduce the simulation time by modifying your signal properties.

Input Signal	Action	Caution
Sinestream	Decrease the number of periods per frequency, `NumPeriods`, especially at lower frequencies.	You model must be at steady state to achieve accurate frequency response estimation. Reducing the number of periods might not excite your model long enough to reach steady state.

Input Signal	Action	Caution
Chirp	Decrease the signal sample time, `Ts`, or the number of samples, `NumSamples`.	The frequency resolution of the estimated response depends on the number of samples `NumSamples`. Decreasing the number of samples decreases the frequency resolution of the estimated frequency response.

For information about modifying input signals, see “Modify Estimation Input Signals” on page 5-40.

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?”.

Using Parallel Computing

You can try to speed up frequency response estimation using parallel computing in the following situations:

- Your model has multiple inputs.
- Your single-input model uses a sinestream input signal, where the sinestream `SimulationOrder` property has the value `'OneAtATime'`.

  For information on setting this option, see the `frest.Sinestream` reference page.

In these situations, frequency response estimation performs multiple simulations. If you have installed the Parallel Computing Toolbox™ software, you can run these multiple simulations in parallel on multiple MATLAB sessions (pool of MATLAB workers).

For more information about using parallel computing, see “Speeding Up Estimation Using Parallel Computing” on page 5-69.

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 **Model Linearizer** and `frestimate`. When you perform frequency response estimation using parallel computing, the software uses the available parallel pool. If no parallel pool is available and **Automatically create a parallel pool** is selected in your Parallel Computing Toolbox preferences, then the software starts a parallel pool using the settings in those preferences.

You can configure the software to automatically detect model dependencies and temporarily add them to the parallel pool workers. However, to ensure that workers are able to access the undetected file and path dependencies, create a cluster profile that specifies the same. The parallel pool used to optimize the model must be associated with this cluster profile. For information on creating a cluster profile, see “Add and Modify Cluster Profiles” (Parallel Computing Toolbox).

To manually open a parallel pool that uses a specific cluster profile, use:

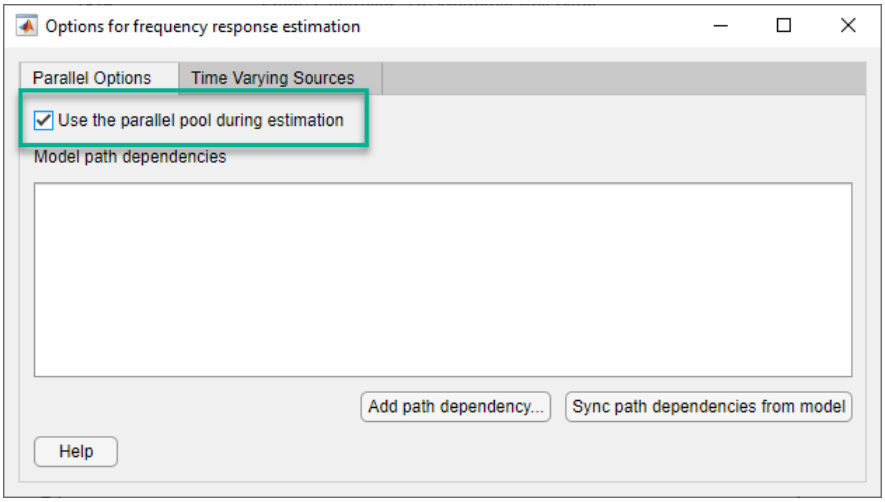
```
parpool(MyProfile)
```

`MyProfile` is the name of a cluster profile.

Estimating Frequency Response Using Parallel Computing Using Model Linearizer

After you configure your parallel computing settings, as described in “Configuring MATLAB for Parallel Computing” on page 5-69, you can estimate the frequency response of a Simulink model using the **Model Linearizer** app.

1. In **Model Linearizer**, on the **Estimation** tab, click **More Options**.
2. In the Options for frequency response estimation dialog box, on the **Parallel Options** tab, select **Use the parallel pool during estimation**.



3. (Optional) Click **Add path dependency**.

   In the Browse For Folder dialog box, navigate to and select the directory to add to the model path dependencies.

   Click **OK**.

   Tip Alternatively, manually specify the paths in the Model path dependencies list. You can specify the paths separated with a new line.

4. (Optional) Click **Sync path dependencies from model**.

   This action finds the model path dependencies in your Simulink model and adds them to the **Model path dependencies** list box.

Estimating Frequency Response Using Parallel Computing (MATLAB Code)

After you configure your parallel computing settings, as described in “Configuring MATLAB for Parallel Computing” on page 5-69, you can estimate the frequency response of a Simulink model.

1. Find the paths to files that your Simulink model requires to run, called *path dependencies*.

   ```
   dirs = frest.findDepend(model)
   ```

`dirs` is a cell array of character vectors containing path dependencies, such as referenced models, data files, and S-functions.

For more information about this command, see `frest.findDepend`.

To learn more about model dependencies, see “Analyze Model Dependencies” and “Dependency Analyzer Scope and Limitations”.

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 “Speed Up Frequency Response Estimation Using Parallel Computing” on page 5-89.

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”.

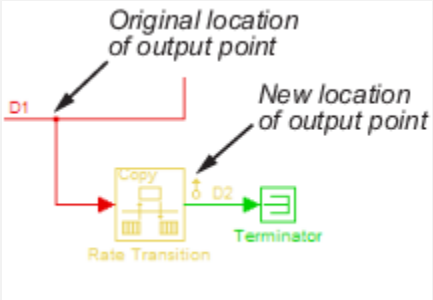
2 Try one or more of the actions listed in the following sections:

- “Model-Specific Ways to Avoid Memory Issues” on page 5-72
- “Input-Signal-Specific Ways to Avoid Memory Issues” on page 5-72

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 5-68.
Models with multiple input and output points (MIMO models)	• Estimate the response for all input/output combinations separately. Then, combine the results into one MIMO model using the data format described in “Create Frequency-Response Model from Data”. • Use parallel computing to run the independent simulations in parallel on different computers. See “Speeding Up Estimation Using Parallel Computing” on page 5-69.

Input-Signal-Specific Ways to Avoid Memory Issues

To avoid memory issues, try one or more of the actions listed in the following table, as appropriate for your input signal type.

Input Signal Type	Action
Sinestream	• Remove low frequencies from your input signal for which you do not need the frequency response. • Modify the sinestream signal to estimate each frequency separately by setting the `SimulationOrder` option to `OneAtATime`. Then estimate using a `frestimate` syntax that does not request the simulated time-response output data, for example `sysest = frestimate(model,io,input)`. • Use parallel computing to run independent simulations in parallel on different computers. See “Speeding Up Estimation Using Parallel Computing” on page 5-69. • Divide the input signal into multiple signals using `fselect`. Estimate the frequency response for each signal separately using `frestimate`. Then, combine results using `fcat`.
Chirp	Create separate input signals that divide up the swept frequency range of the original signal into smaller sections using `frest.Chirp`. Estimate the frequency response for each signal separately using `frestimate`. Then, combine results using `fcat`.
Random	Decrease the number of samples in the random input signal by changing `NumSamples` before estimating. See “Time Response Is Noisy” on page 5-49.

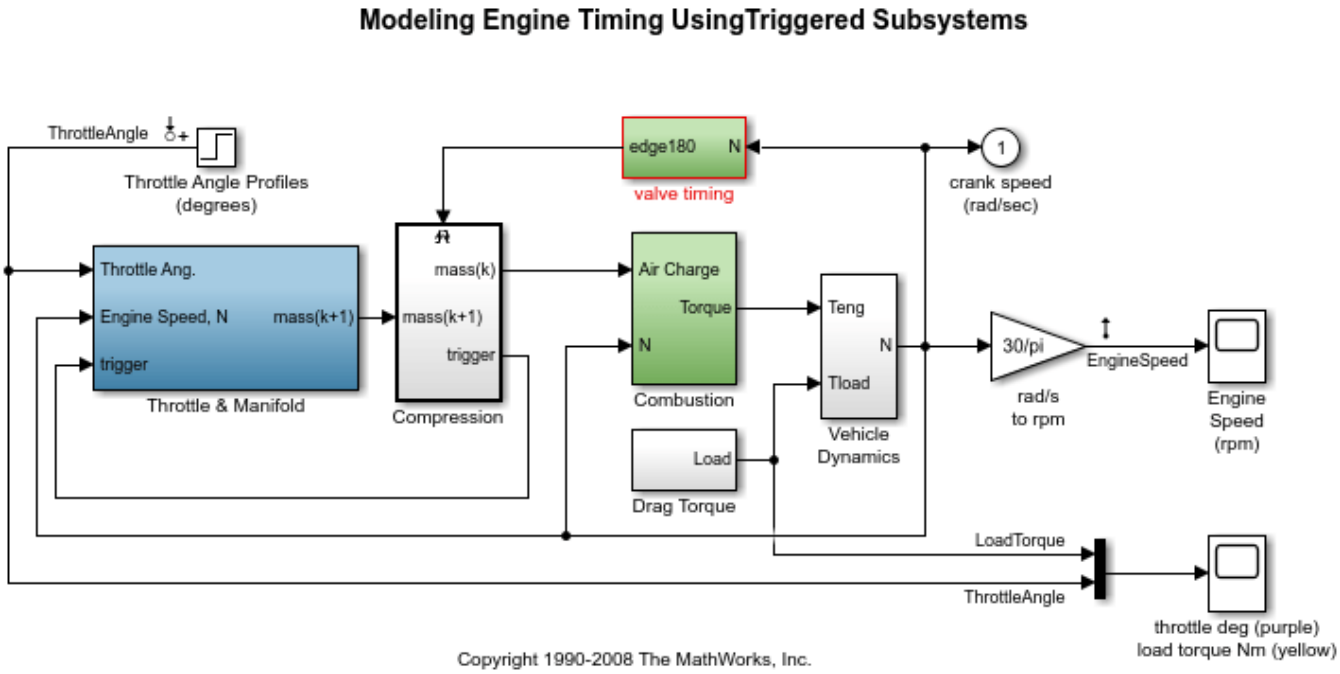
Frequency Response Estimation Using Simulation-Based Techniques

This example shows how to obtain the frequency response of Simulink® models when analytical block-by-block linearization does not provide an accurate answer due to event-based dynamics in the linearization path. Examples of such systems are models with triggered subsystems or models using pulse width modulation (PWM) input signals.

Open the Model

Open the Simulink model for engine timing.

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



Analytical block-by-block linearization of this model from the throttle angle to engine speed gives a zero linearization result due to the triggered Compression subsystem in the linearization path.

```
io = getlinio(mdl);
linsys = linearize(mdl,io)
```

```
linsys =

  D =
                ThrottleAngl
   EngineSpeed             0

Static gain.
```

Estimate Frequency Response Using Sinestream Input Signal

Sinestream input signals are the most reliable input signals for estimating an accurate frequency response of a Simulink model using the `frestimate` function. A sinestream signal is composed of individual sinusoidal signals that are appended to each other. The `frestimate` function simulates the model for each frequency in the sinestream input signal, as specified using the `Frequency` parameter, for the corresponding amount of periods, as specified using the `NumPeriods` parameter. After the simulation, `frestimate` uses the output signal to compute the response at each frequency. `frestimate` uses only the periods after the system reaches a steady state for that input frequency, that is, after a number of settling periods as specified using the `SettlingPeriods` parameter.

Create a sinestream signal with 50 logarithmically spaced distinct frequencies between 0.1 and 10 rad/s.

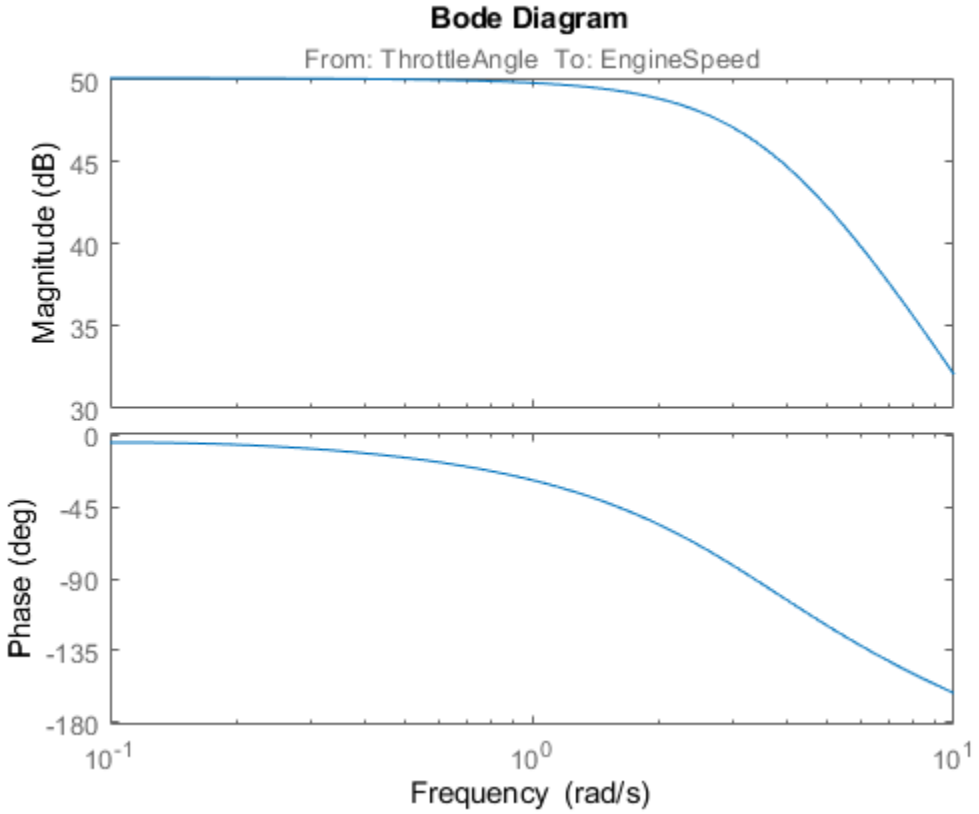
```
in = frest.Sinestream('Frequency',logspace(-1,1,50),'Amplitude',1e-3)
```

```
The sinestream input signal:

      Frequency           : [0.1 0.10985 0.12068 0.13257 ...] (rad/s)
      Amplitude           : 0.001
      SamplesPerPeriod    : 40
      NumPeriods          : 4
      RampPeriods         : 0
      FreqUnits (rad/s,Hz): rad/s
      SettlingPeriods     : 1
      ApplyFilteringInFRESTIMATE (on/off)    : on
      SimulationOrder (Sequential/OneAtATime): Sequential
```

By default, each frequency in a sinestream input signal is simulated for four periods, that is, `NumPeriods` is `4` for all frequencies. Also, the end of the first period is specified to be where the system reaches steady state, that is, `SettlingPeriods` is `1` for all frequencies. Therefore, `frestimate` uses the last three periods of the output signals.

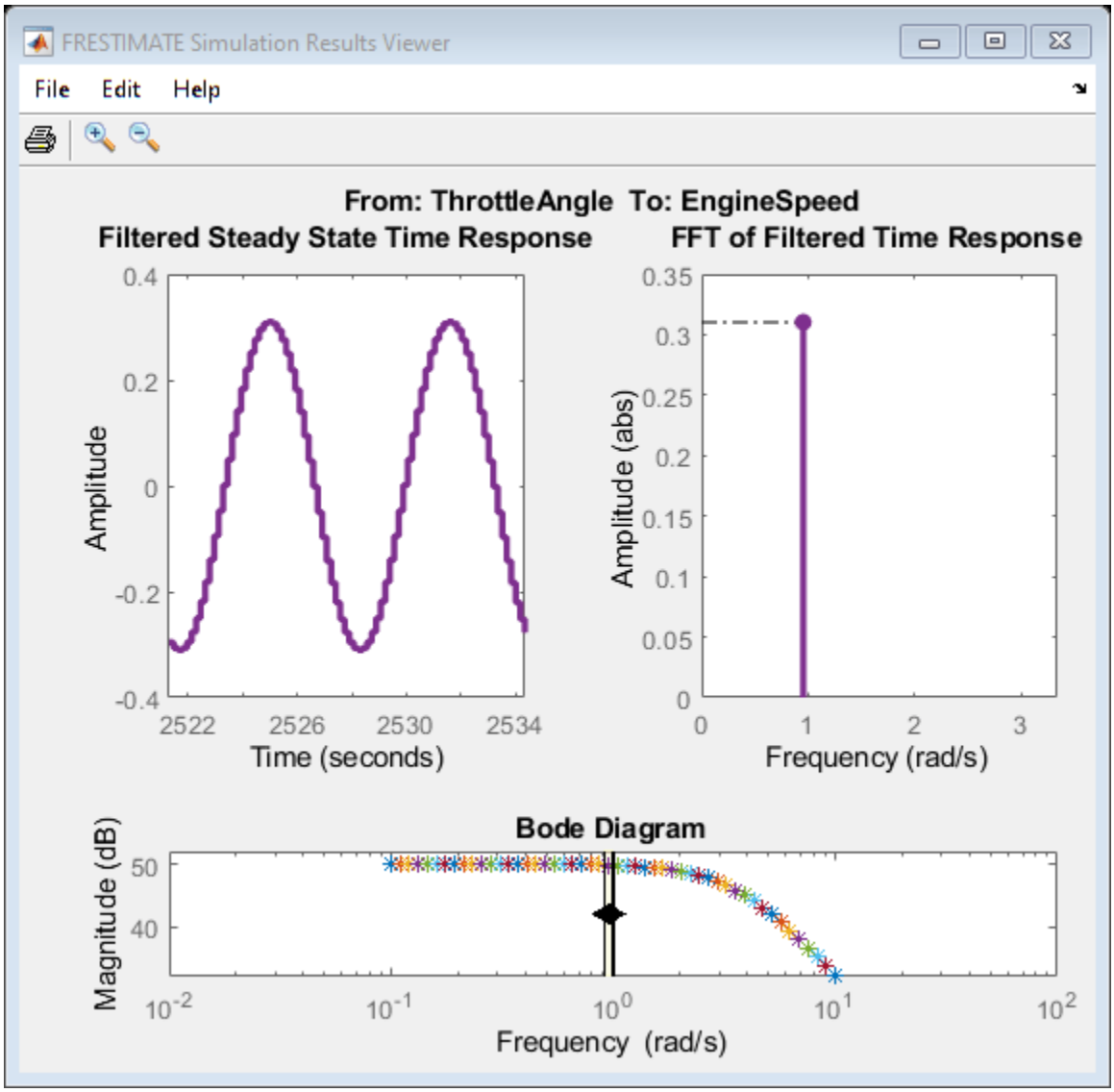
Using this sinestream input signal, perform the frequency response estimation using the `frestimate` function, and plot the resulting frequency response data.

```
[sys,simout] = frestimate(mdl,io,in);
bode(sys)
```



You can inspect the estimation results using the Simulation Results Viewer. The viewer shows time-domain and frequency-domain simulation results for the selected frequencies and a summary bode plot where you can interactively switch between frequencies.

```
frest.simView(simout,in,sys);
```



You can use the viewer as a tool to diagnose issues that impact the accuracy of the frequency response estimation, such as:

- Not reaching steady state
- Excitation of nonlinearities
- Running into instabilities

Estimate Frequency Response Using Chirp Input Signal

Another input signal you can use when estimating frequency response data from a Simulink model is a frequency chirp. A frequency chirp differs from a sinestream in that the frequency is instantaneously varied.

You can use chirp input signals to obtain faster frequency response estimation. However, the resulting frequency response can be less reliable than that obtained using sinestream inputs, since each frequency is not simulated long enough to drive the system to steady state at that frequency.

Create a chirp signal that sweeps between the frequencies 0.1 and 10 rad/s logarithmically.

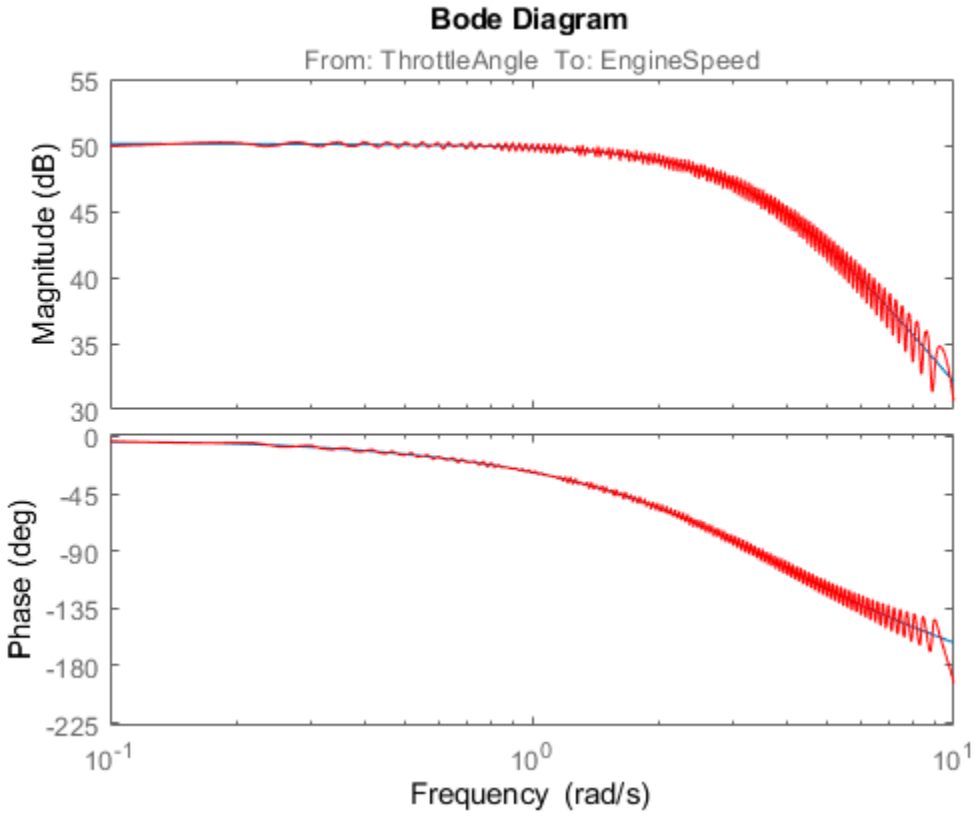
```
in_chirp = frest.Chirp('FreqRange',[0.1 10],'Amplitude',1e-3,...
    'SweepMethod','logarithmic','NumSamples',3000);
```

Perform the frequency response estimation using the chirp signal.

```
sys_chirp = frestimate(mdl,io,in_chirp);
```

Plot the results obtained from both the sinestream and chirp input signals together.

```
bode(sys,sys_chirp,'r')
```



Estimate Frequency Response Using PRBS Input Signal

Another input signal you can use when estimating frequency response data from a Simulink model is a pseudorandom binary sequence (PRBS). A PRBS is a periodic, deterministic signal with white-noise-like properties that shifts between two values.

You can use PRBS input signals to obtain faster frequency response estimation with a higher frequency resolution than the chirp signal.

Create a PRBS signal with an order of 12 and one period in the signal. Using a single period produces a nonperiodic PRBS. The length of the generated PRBS is 4095 (2^12 - 1). To obtain an accurate frequency response estimation, the length of the PRBS must be sufficiently large. To ensure that the system is properly excited, specify the perturbation amplitude of the PRBS as `0.01`.

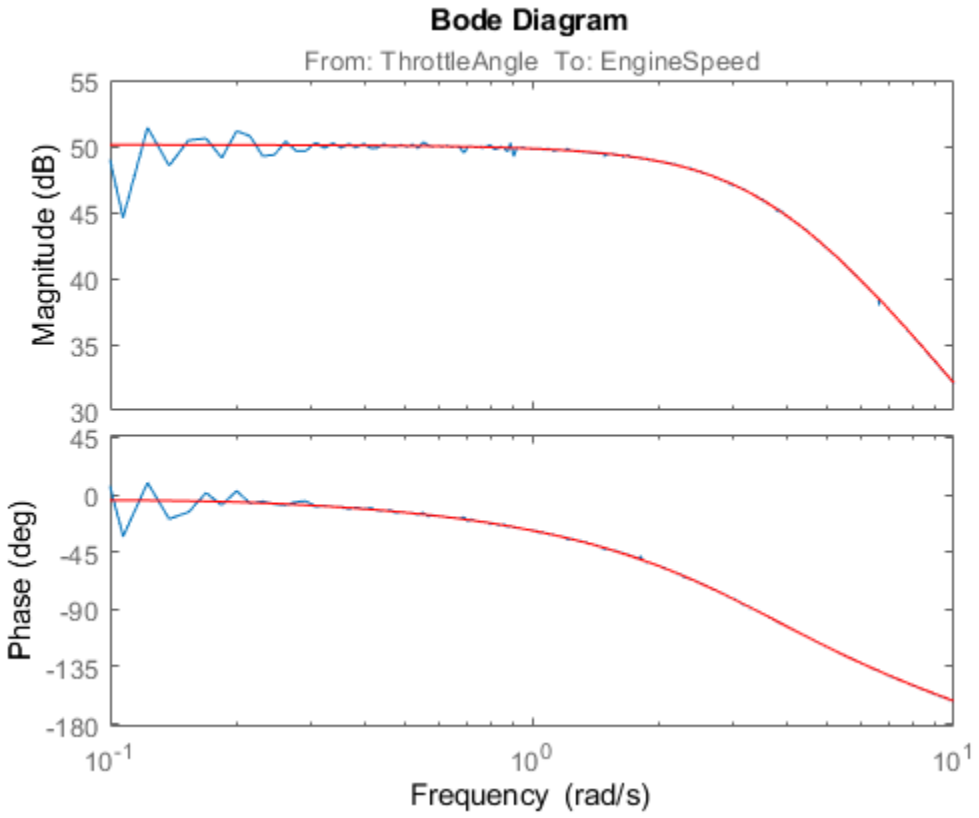
```
in_PRBS = frest.PRBS('Order',12,'NumPeriods',1,'Amplitude',1e-2,'Ts',0.1);
```

Perform the frequency response estimation using the PRBS signal.

```
sys_PRBS = frestimate(mdl,io,in_PRBS);
```

Plot the results obtained with both PRBS and sinestream input signals together.

```
bode(sys_PRBS,sys,'r',{0.1 10})
```



Close the model.

```
bdclose('scdengine')
```

See Also

`frestimate`

More About

- “Frequency Response Estimation Basics” on page 5-2

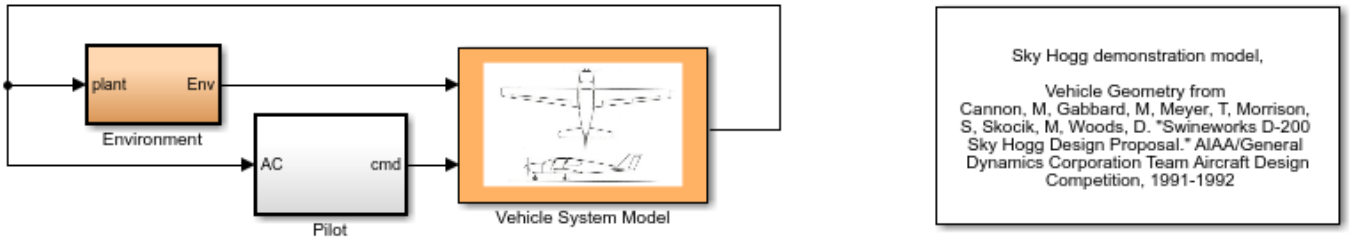
Validate Linearization in Frequency Domain at Command Line

This example shows how to validate a block-by-block analytical linearization result using frequency response estimation. To run this example, you need Aerospace Blockset™ software.

Linearize Model

Open the Simulink® model for the lightweight airplane. For more information on this model, see "Lightweight Airplane Design" (Aerospace Blockset).

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



Copyright 2007-2017 The MathWorks, Inc.

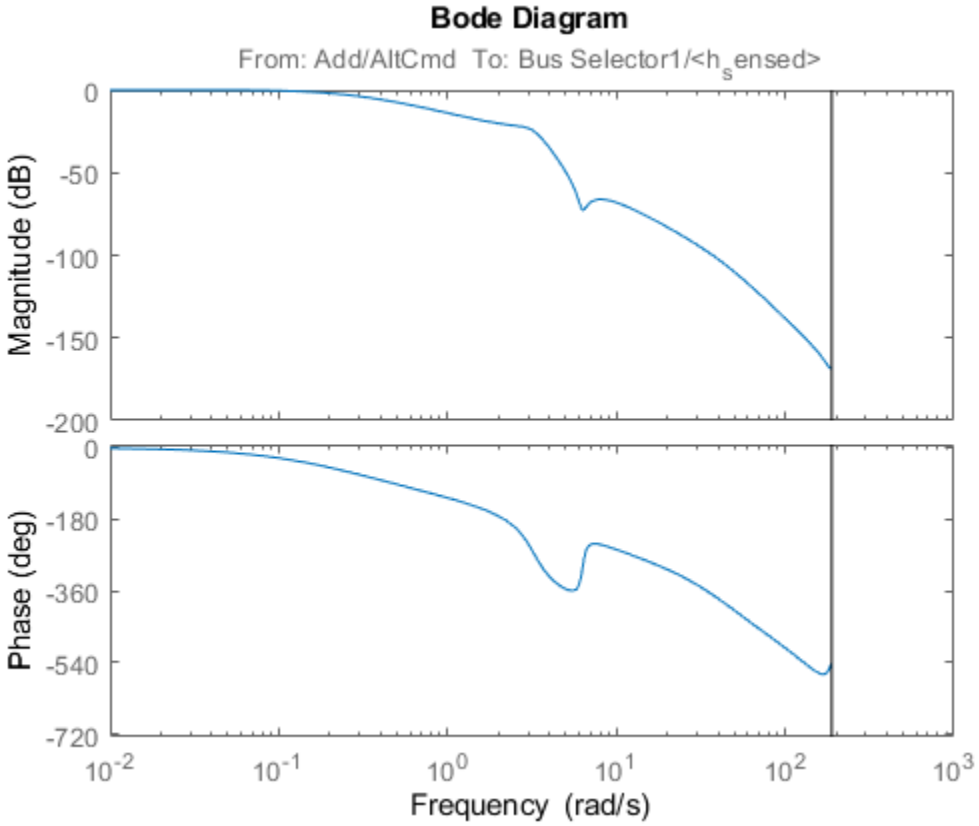
You can linearize the lightweight airplane model from the altitude command signal, `AltCmd`, to the sensed height, `h_sensed`. These linear analysis points are already specified in the model.

```
io = getlinio(mdl)
```

```
2x1 vector of Linearization IOs:
--------------------------
1. Linearization input perturbation located at the following signal:
- Block: scdskyhogg/Pilot/Add
- Port: 1
- Signal Name: AltCmd
2. Linearization output measurement located at the following signal:
- Block: scdskyhogg/Vehicle System Model/Avionics/Autopilot/Bus Selector1
- Port: 1
- Signal Name: <h_sensed>
```

Linearize the model using the `linearize` function. The model is preconfigured to use an operating point obtained using a simulation snapshot at t = 75.

```
sys = linearize(mdl,io);
bode(sys)
```



Estimate Frequency Response

To determine whether the linearization results properly capture characteristics of the nonlinear model, such as the anti-resonance around 6.28 rad/s, you can validate the linearization result using `frestimate`.

Create a sinestream input signal. Use the linearization result as an input argument to automatically set various parameters of the sinestream input signal, such as the set of frequencies and the number of periods for each frequency, based on the linear system.

```
in = frest.Sinestream(sys);
in.Amplitude = 0.5
```

```
The sinestream input signal:

      Frequency           : [0.0034142;0.0054345;0.0086502;0.013768 ...] (rad/s)
      Amplitude           : 0.5
      SamplesPerPeriod    : [110417;69370;43582;27381 ...]
      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
```

The software selects 25 frequencies at which to compute the response. These frequencies vary between 0.0034 rad/s and 14.5 rad/s. The frequencies that are automatically selected focus on where interesting dynamics occur (such as the anti-resonance at 6.28 rad/s). The number of periods that it takes for the system to reach steady state is estimated for each of these frequencies and varies between 1 period (for 0.0034 rad/s) and 188 periods (for 14.5 rad/s).

Estimate the frequency response using this input signal. `frestimate` simulates the model with the input signal, which can take a long time in a normal simulation model. To speed up the simulation, configure the model to use rapid accelerator mode.

```
set_param(mdl,'SimulationMode','rapid');
```

Using rapid accelerator mode can significantly increase the speed of the simulation. The actual speed improvement depends on your computer configuration.

To run the frequency response estimation use the following command.

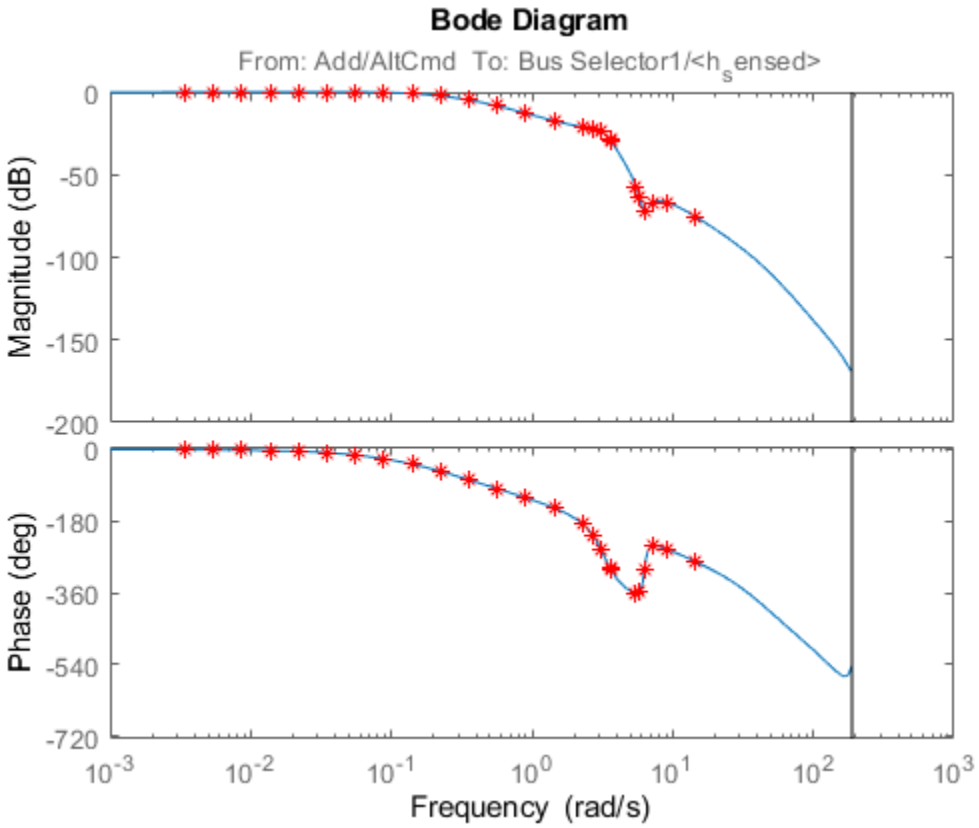
```
sysest = frestimate(mdl,in,io);
```

For this example, you can load the estimation result from a MAT-file.

```
load scdskyhogg_frestresults.mat;
```

Compare the analytical linearization result against the frequency response data from `frestimate`. The frequency response data and analytical linearization result match well, validating that anti-resonance between the frequencies 1 and 10 rad/s does exist in the actual nonlinear airplane model.

```
bode(sys,sysest,'r*')
```



Close the model.

```
bdclose('scdskyhogg')
```

See Also

`frestimate`

More About

- “Validate Linearization In Frequency Domain Using Model Linearizer” on page 2-108
- “Validate Linearization In Time Domain” on page 2-107

Describing Function Analysis of Nonlinear Simulink Models

This example shows how to use the frequency response estimation to perform a sinusoidal-input describing function analysis for a model with a saturation nonlinearity.

Describing function analysis is a technique for studying the frequency response of nonlinear systems. It is an extension of linear frequency response analysis. In linear systems, transfer functions depend only on the frequency of the input signal. In nonlinear systems, when a specific class of input signal, such as a sinusoid, is applied to a nonlinear element, you can represent the nonlinear element using a describing function. A describing function depends not only on the input frequency but also on the input amplitude. Describing function analysis has a wide area of applications from frequency response analysis to the prediction of limit cycles.

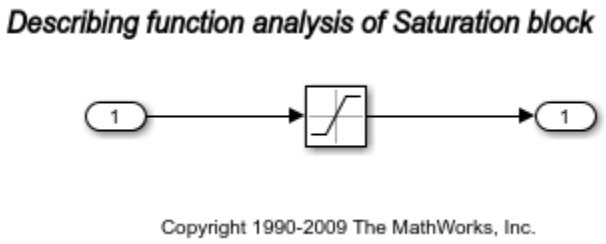
To use sinusoidal-input describing function analysis, which is the most common type of describing function analysis, your model should satisfy the following conditions.

1. Nonlinearity is time-invariant.
2. Nonlinearity does not generate any subharmonics in response to the input sinusoid.
3. The system filters out any superharmonics generated by the nonlinearity (this condition is often referred to as the filtering hypothesis).

In this example, you perform describing function analysis on a model with a saturation nonlinearity that satisfies these conditions.

Open the Simulink® model.

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



The saturation nonlinearity has the following sinusoidal input describing function.

$$N_A(\gamma) = -1, \gamma \le -1$$

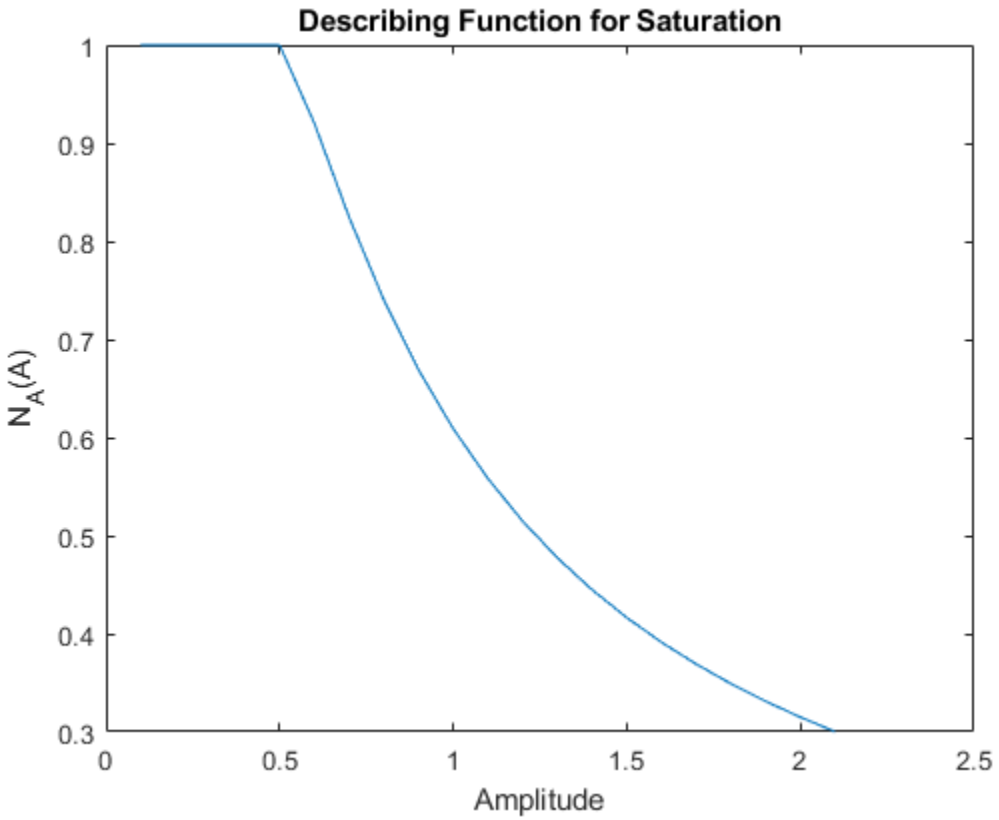
$$N_A(\gamma) = \frac{2}{\pi}(sin^{-}1(\gamma) + (\gamma\sqrt{(1-\gamma^2)})), -1 < \gamma < 1$$

$$N_A(\gamma) = 1, \gamma \ge 1$$

Here, $\gamma = 0.5/A$ for a saturation with upper and lower limits of 0.5 and -0.5, respectively, and A is the amplitude of the sinusoidal input signal.

Compute and plot the describing function $N_A(A)$ for amplitudes varying between 0.1 and 2.1. The describing function is implemented in `saturationDF.m`.

```
A = linspace(0.1,2.1,21);
N_A = saturationDF(0.5./A);
plot(A, N_A)
xlabel('Amplitude')
ylabel('N_A(A)')
title('Describing Function for Saturation')
```



You can compute the describing function for the saturation nonlinearity using `frestimate` over the same set of amplitudes for a fixed frequency of 5 rad/s. Since the describing function for a saturation does not depend on frequency, it is sufficient to run the analysis at a single frequency.

Define the input and output linear analysis points.

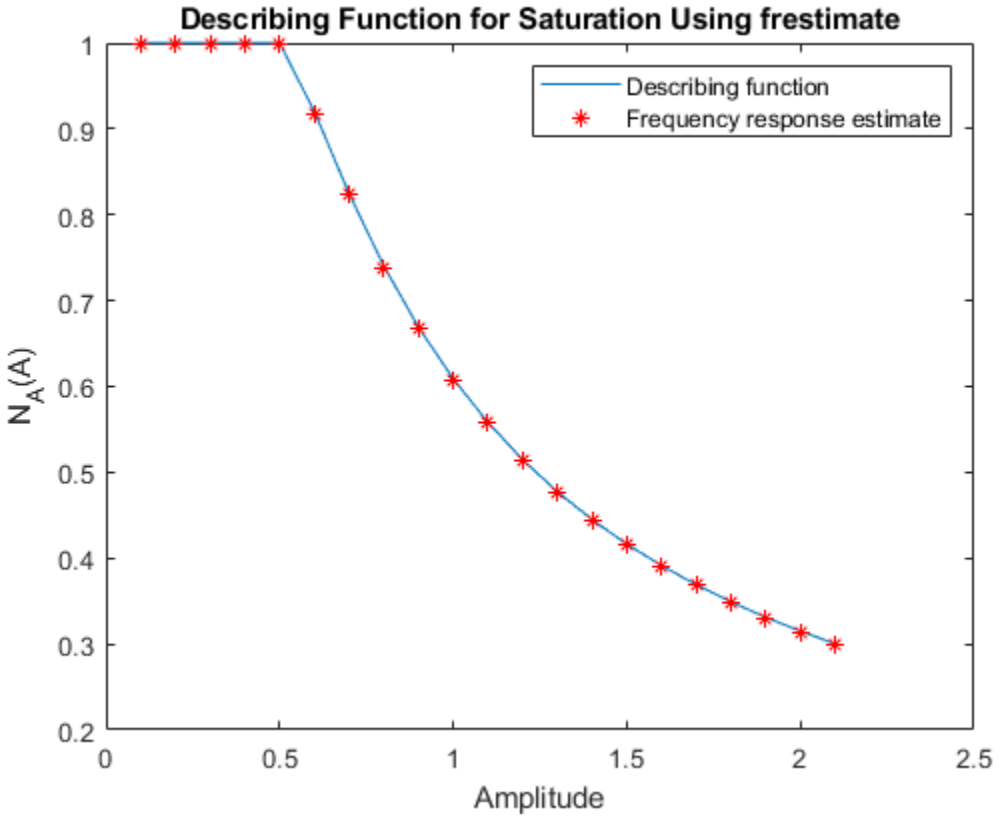
```
io(1) = linio('scdsaturationDF/In1',1,'input');
io(2) = linio('scdsaturationDF/Saturation',1,'output');
```

For each amplitude, create a sinestream input with the fixed 5 rad/s frequency and given amplitude. Then, run `frestimate` using this input signal.

```
w = 5;
N_A_withfrest = zeros(size(N_A));
for ct = 1:numel(A)
    in = frest.Sinestream('Frequency',w,'Amplitude',A(ct));
    sysest = frestimate(mdl,in,io);
    N_A_withfrest(ct) = real(sysest.resp);
end
```

Plot the frequency response estimation result along with the describing function.

```
plot(A,N_A,A,N_A_withfrest,'r*')
xlabel('Amplitude')
ylabel('N_A(A)')
title('Describing Function for Saturation Using frestimate')
legend('Describing function','Frequency response estimate')
```

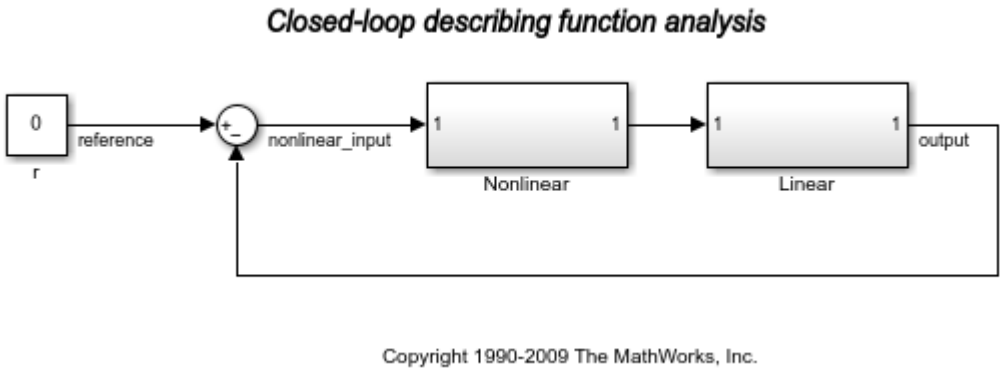


Close the model.

```
bdclose(mdl)
```

You can also run a closed-loop describing function analysis over a frequency range. Open the model for this analysis.

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



To begin, analytically compute the frequency response from the reference to the output using describing functions. To do so, first compute the amplitude of the input signal to the nonlinearity, given the reference amplitude and frequency. The input amplitude for the nonlinearity is not necessarily equal to the reference amplitude.

```
L = zpk([],[0 -1 -10],1);
w = logspace(-2,2,50);
A_DF = zeros(numel(A),numel(w));
for ct_amp = 1:numel(A)
    for ct_freq = 1:numel(w)
        % Compute the input amplitude to the nonlinearity by solving the
        % analytical equation.
        opt = optimset('Display','off');
        A_DF(ct_amp,ct_freq) = ...
            fzero(@(A_DF) solveForSatAmp(A_DF,L,w(ct_freq),A(ct_amp)),...
            A(ct_amp),opt);
    end
end
```

Next, for each amplitude, compute the analytical frequency response of the closed-loop system from the reference to the output using the describing function. Store the results in an array of `frd` objects.

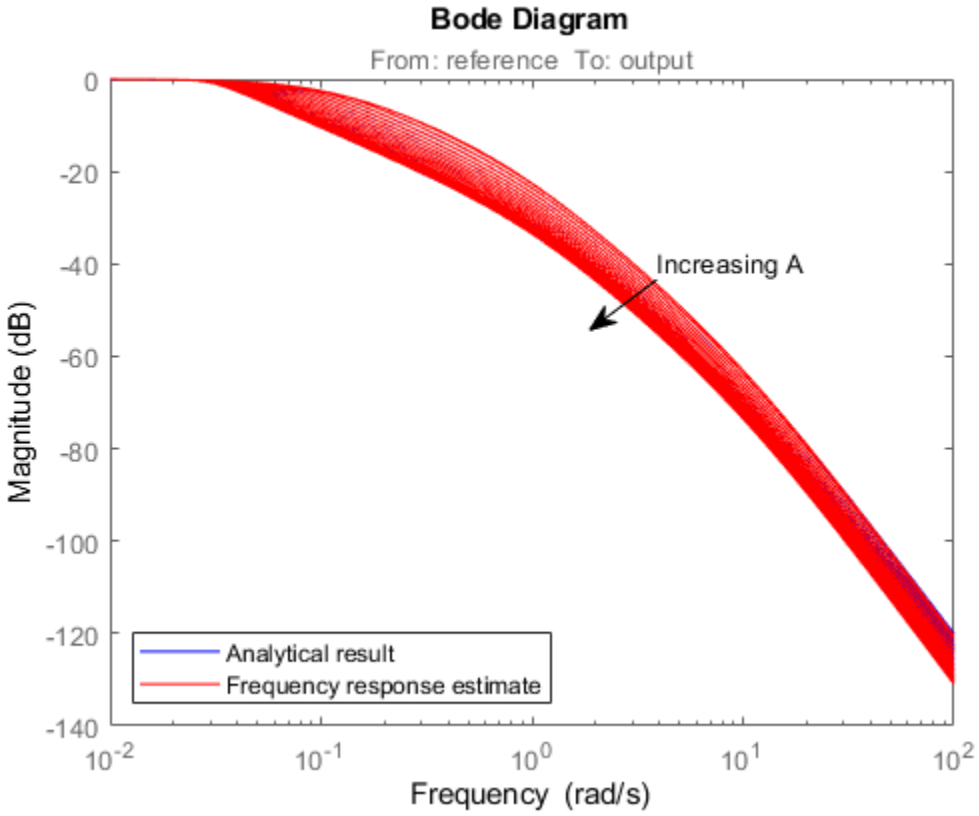
```
L_w = freqresp(L,w);
for ct = 1:numel(A)
    N_A = saturationDF(0.5./A_DF(ct,:));
    cl_resp = N_A(:).*L_w(:)./(1+N_A(:).*L_w(:));
    cl(1,1,ct) = frd(cl_resp,w);
end
```

Obtain the frequency response for the closed-loop system using `frestimate` in a manner similar to the saturation describing function analysis.

```
io(1) = linio('scdsaturationDFcl/r',1,'input');
io(2) = linio('scdsaturationDFcl/Linear',1,'output');
for ct = 1:numel(A)
    in = frest.Sinestream('Frequency',w,'Amplitude',A(ct),...
        'NumPeriods',10,'SettlingPeriods',7);
    cl_withfrest(1,1,ct)  = frestimate(mdl,in,io);
end
```

Plot the analytically calculated closed-loop magnitude along with the result from `frestimate`. As expected, the results match. The arrow indicates the direction of increasing amplitude.

```
h = figure;
bodemag(cl,'b',cl_withfrest,'r')
annotation(h,'textarrow',[0.64 0.58],[0.64 0.58],'String','Increasing A');
legend('Analytical result','Frequency response estimate',...
    'Location','southwest')
```



Close the model.

```
bdclose(mdl)
```

See Also

`frestimate` | `frest.Sinestream`

Speed Up Frequency Response Estimation Using Parallel Computing

This example illustrates how to speed up frequency response estimation of Simulink® models using parallel computing. In some scenarios, the `frestimate` function estimates the frequency response of a Simulink model by performing multiple Simulink simulations. You can distribute these simulations to a pool of MATLAB® workers by using Parallel Computing Toolbox™ software.

This example requires Parallel Computing Toolbox software. You can optionally run simulations on a computer cluster using MATLAB Parallel Server™ software. This example uses the local worker functionality available in Parallel Computing Toolbox software.

Speed up Simulink Simulations Performed by `frestimate`

When you compute a frequency response using the `frestimate` function, the majority of computation time is spent in Simulink simulations. To reduce the total simulation time, you can:

1. Use rapid accelerator mode. Use this method when `frestimate` performs only one Simulink simulation. For an example, see “Validate Linearization in Frequency Domain at Command Line” on page 5-80.
2. Distribute simulations across workers in a MATLAB pool. Use this method when `frestimate` performs multiple Simulink simulations. `frestimate` performs more than one Simulink simulation when you specify the following:

- A sinestream input signal with the `SimulationOrder` parameter set to `'OneAtATime'`. In this case, each frequency in the sinestream signal is simulated separately.
- Linear analysis points with more than one input point or a nonscalar input point. In this case, each linearization input point or each channel in a nonscalar linearization input point yields a separate Simulink simulation.

Using the `frestimate` function with parallel computing also supports normal, accelerator, and rapid accelerator modes.

Configure a MATLAB Pool

To use parallel computing to speed up frequency response estimation, configure and start a pool of MATLAB workers before you run the `frestimate` function.

To check if a MATLAB pool is open, use the `gcp` function. If no pool is open, open one using the `parpool` function.

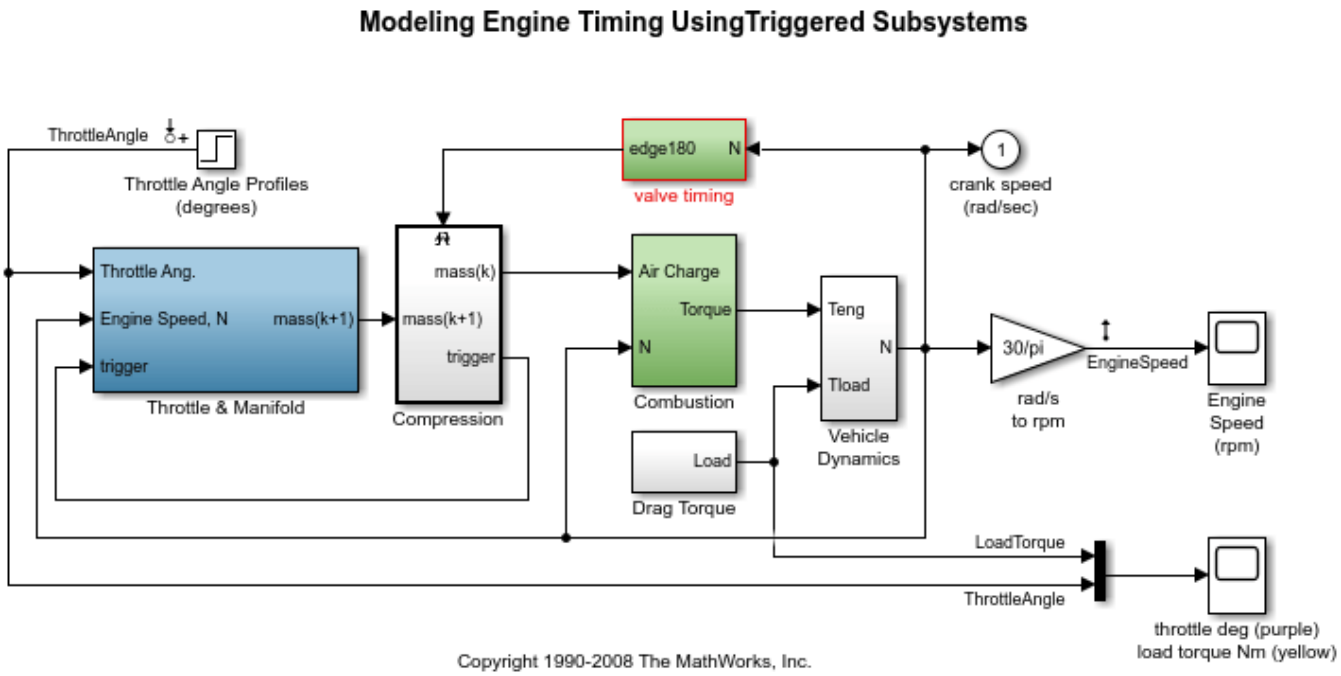
```
if isempty(gcp)
   parpool local
end
```

Distribute Simulink Simulations for Each Frequency in Sinestream Input

When you use a sinestream input signal with the `frestimate` function and you set the `SimulationOrder` parameter to `'OneAtATime'`, each frequency in the sinestream signal simulates in a separate Simulink simulation. If you enable the parallel computing option, the simulations corresponding to individual frequencies are distributed among workers in the MATLAB pool.

Open the model, and obtain the linear analysis points stored in the model.

```
mdl = 'scdengine';
open_system(mdl)
io = getlinio(mdl);
```



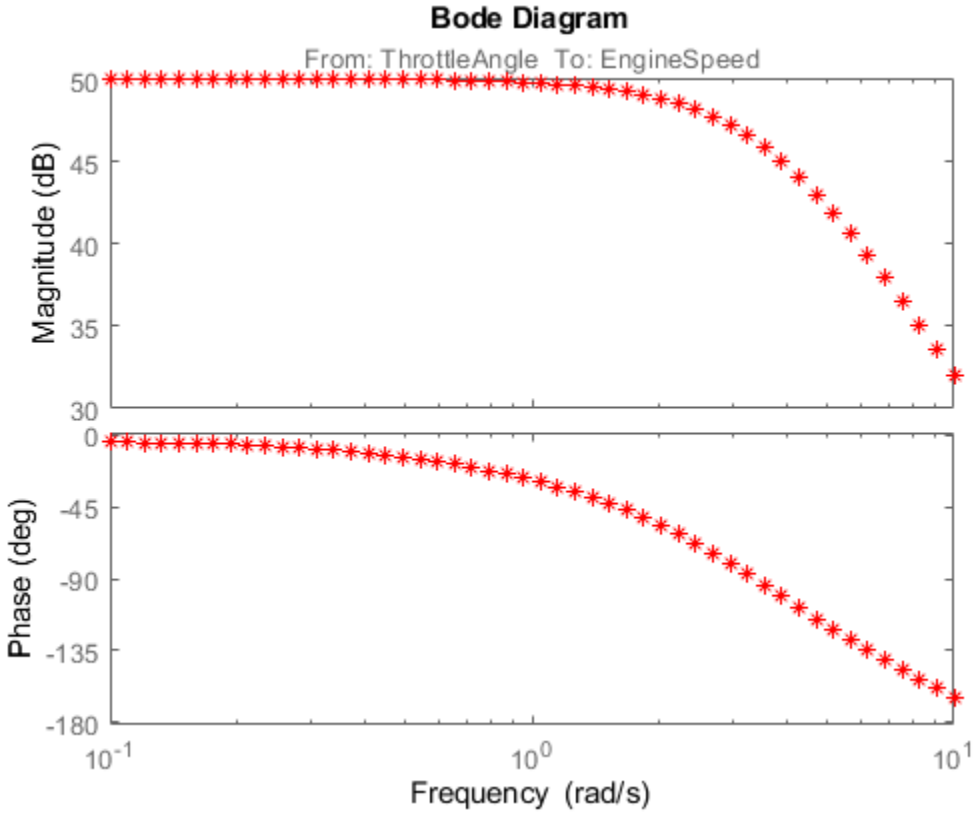
Create a sinestream input signal with a `'OneAtATime'` simulation order.

```
in = frest.Sinestream('Frequency',logspace(-1,1,50),'Amplitude',1e-3,...
    'SimulationOrder','OneAtATime');
```

In this model, there is a single linearization input point and a single linearization output point. There are 50 frequencies in the sinestream signal. The `frestimate` command performs 50 separate Simulink simulations because the `SimulationOrder` parameter is set to `'OneAtATime'`.

To distribute these simulations among workers, enable parallel computing for `frestimate`. Create an `frestimateOptions` object and set the `UseParallel` option to `'on'`. Use this object as an input argument for `frestimate`.

```
opt = frestimateOptions('UseParallel','on');
sysest = frestimate(mdl,io,in,opt);
bode(sysest,'r*')
```



In general, parallel computing significantly speeds up frequency response estimation using `frestimate`. The actual processing times and amount of improvement will depend on your computer setup and your Parallel Computing Toolbox configuration. For example, the amount of improvement can be affected by various factors including the overhead from client-to-worker data transfer and resource competition between worker processes and OS processes.

Close the model.

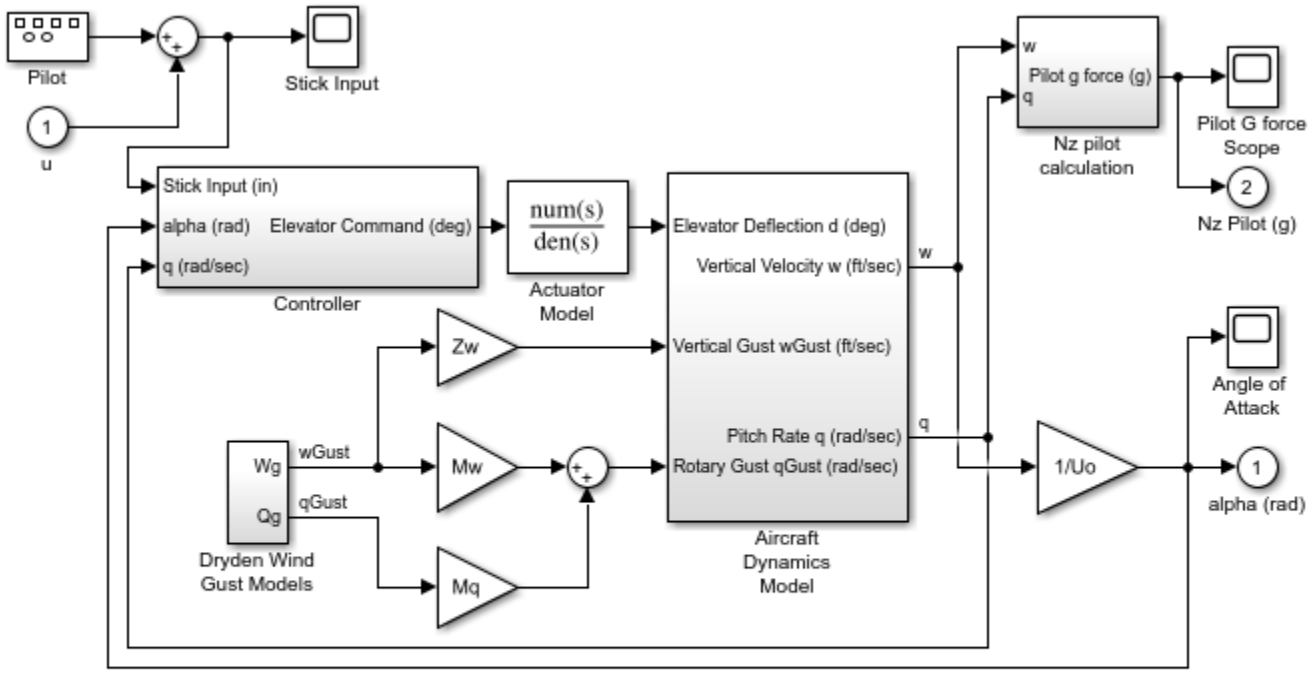
```
bdclose(mdl)
```

Distribute Simulink Simulations for Input Channels

When the number of linearization input points or the number of channels in a linearization input point is greater than one, the `frestimate` command distributes individual Simulink simulations corresponding to these input channels among workers in the MATLAB pool.

Open the model, and obtain the linear analysis points stored in the model.

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

io(1) = linio('scdplane/Sum1',1,'input');
io(2) = linio('scdplane/Actuator Model',1,'input');
io(3) = linio('scdplane/Gain5',1,'output');
```



Copyright 1990-2012 The MathWorks, Inc.

With the `linio` function, you specify two linearization input points, which are both located on scalar Simulink signals. If you run the `frestimate` command to estimate the frequency response for this model, two Simulink simulations occur, one for each input.

Linearize the model, and create an input signal using the linearization result.

```
sys = linearize(mdl,io);
in = frest.Sinestream(sys);
```

Before estimating the frequency response, find all source blocks in the signal paths of the linearization outputs that generate time-varying signals using the `findSources` function. Such time-varying signals can interfere with the signal at the linearization output points and produce inaccurate estimation results. To disable the time-varying source blocks, create an `frestimateOptions` option set and specify the `BlocksToHoldConstant` option.

```
srcblks = frest.findSources('scdplane',io);
opt = frestimateOptions('BlocksToHoldConstant',srcblks);
```

Enable parallel computing using the `UseParallel` estimation option, which distributes simulations among workers.

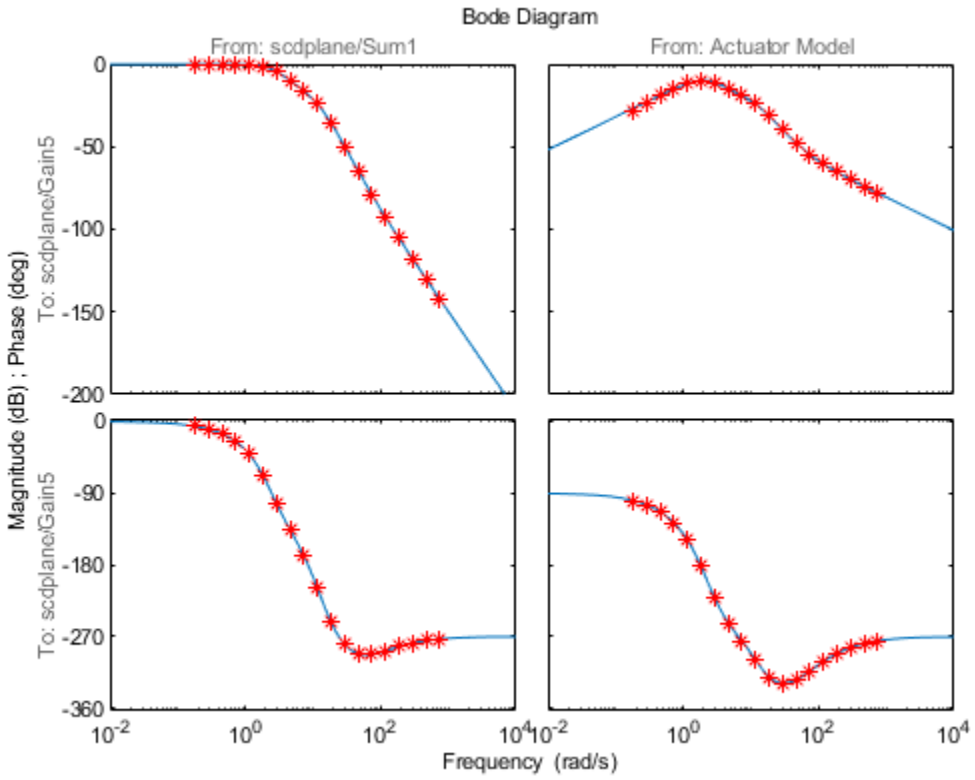
```
opt.UseParallel = 'on';
```

Run `frestimate` using parallel computing.

```
sysest = frestimate(mdl,io,in,opt);
```

Plot the estimation result against the analytical linearization

```
bodeopts = bodeoptions;
bodeopts.PhaseMatching = 'on';
bodeplot(sys,sysest,'r*',bodeopts);
```



Close the model, the open figure, and the parallel pool.

```
bdclose(mdl)
close(gcf)
delete(gcp)
```

See Also

`frestimate` | `frestimateOptions` | `parpool` | `gcp`

More About

- “Managing Estimation Speed and Memory” on page 5-68
- “Troubleshooting Frequency Response Estimation” on page 5-43

Frequency Response Estimation for Power Electronics Model Using Pseudorandom Binary Signal

This example shows how to identify a frequency domain model using a pseudorandom binary sequence (PRBS) for a power electronics system modeled in Simulink® using Simscape™ Electrical™ components. This example addresses the frequency response estimation process in the controller design workflow using a PRBS as the input signal.

Typically, power electronics systems cannot be linearized because they use high-frequency switching components, such as pulse-width modulation (PWM) generators. However, most Simulink Control Design™ PID tuning tools design PID gains based on a linearized plant model. To obtain such a model for a power electronics model that cannot be linearized, you can estimate the plant frequency response over a range of frequencies, as shown in this example.

To collect frequency response data, you can:

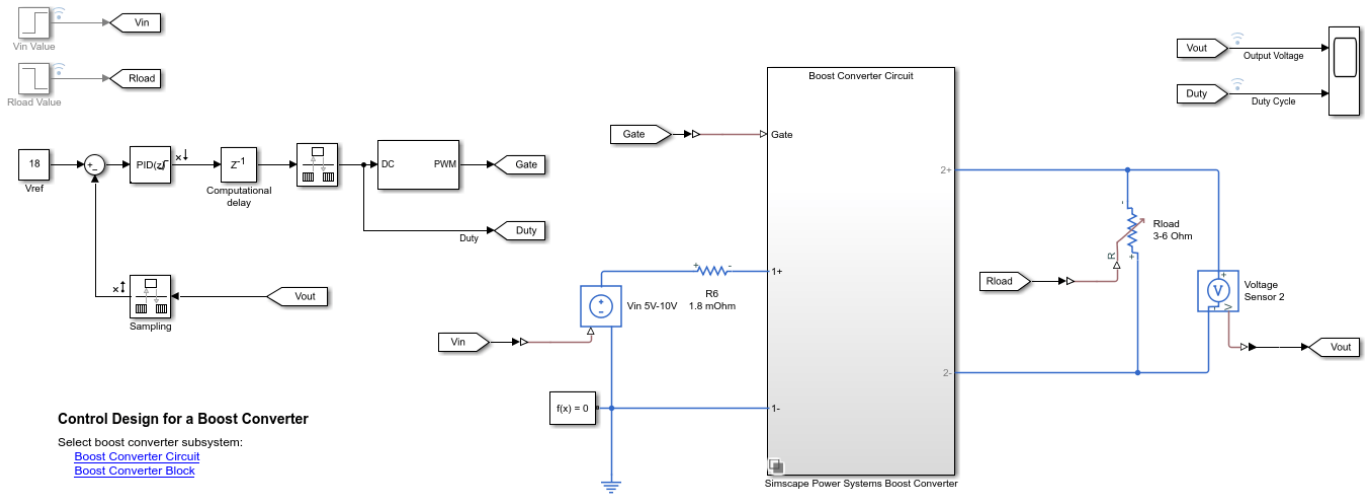
- Estimate the plant frequency response at the command line.
- Estimate the plant frequency response using the **Model Linearizer** app.

This example shows how to estimate the plant frequency response at the command line. To learn how to estimate the plant frequency response for a power electronics system in **Model Linearizer** using a PRBS input signal, see “Frequency Response Estimation in Model Linearizer Using Pseudorandom Binary Sequence” on page 5-101.

Boost Converter Model

This example uses a boost converter model as a power electronics system. A boost converter circuit converts one DC voltage to another, typically higher, DC voltage by controlled chopping or switching of the source voltage.

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



This model uses a MOSFET driven by a PWM signal for switching. The output voltage $Vout$ is regulated to the reference value $Vref$. A digital PID controller adjusts the PWM duty cycle, $Duty$,

based on the voltage error signal. For this example, you estimate the frequency response from the PWM duty cycle to the load voltage $Vout$.

Simscape Electrical software contains predefined blocks for many power electronics systems. This model contains a variant subsystem with two versions of the boost converter model:

- Boost converter circuit constructed using electrical power components. The parameters of the circuit components are based on [1].
- Boost converter block configured to have the same parameters as the boost converter circuit. For more information on this block, see Boost Converter (Simscape Electrical).

To use the Boost Converter block version of the subsystem, in the model, click **Boost Converter Block** or use the following command.

```
set_param([bdroot '/Simscape Power Systems Boost Converter'],...
    'LabelModeActiveChoice','block_boost_converter');
```

Find Model Operating Point

To estimate the frequency response for the boost converter, you must first determine the steady-state operating point at which you want the converter to operate. For more information on finding operating points, see “Find Steady-State Operating Points for Simscape Models” on page 1-104. For this example, use an operating point estimated from a simulation snapshot at 0.045 seconds.

```
opini = findop(mdl,0.045);
```

Initialize the model with the computed operating point.

```
set_param(mdl,'LoadInitialState','on','InitialState','getstatestruct(opini)');
```

Create Pseudorandom Binary Signal

A pseudorandom binary signal (PRBS) is a periodic, deterministic signal with white-noise-like properties that shifts between two values. A PRBS is an inherently periodic signal with a maximum period length of $2^n - 1$, where n is the PRBS order. For more information, see “PRBS Input Signals” on page 5-36.

Create a PRBS with the following configuration.

- To use a nonperiodic PRBS, set the number of periods to 1.
- Use a PRBS order of 14, producing a signal of length 16383. To obtain an accurate frequency response estimation, the length of the PRBS must be sufficiently large.
- Set the injection frequency of the PRBS to 200 kHz to match the sample time in the model. That is, specify a sample time of `5e-6` seconds.
- To ensure that the system is properly excited, set the perturbation amplitude to 0.05. If the input amplitude is too large, the boost converter operates in discontinuous-current mode. If the input amplitude is too small, the PRBS is indistinguishable from ripples in the power electronic circuits.

```
in_PRBS = frest.PRBS('Order',14,'NumPeriods',1,'Amplitude',0.05,'Ts',5e-6);
```

Collect Frequency Response Data

To collect frequency response data, you can estimate the plant frequency response at the command line. To do so, first get the input and output linear analysis points from the model.

```
io = getlinio(mdl);
```

Specify the operating point using the model initial conditions.

```
op = operpoint(mdl);
```

Find all source blocks in the signal paths of the linearization outputs that generate time-varying signals. Such time-varying signals can interfere with the signal at the linearization output points and produce inaccurate estimation results.

```
srcblks = frest.findSources(mdl,io);
```

To disable the time-varying source blocks, create an `frestimateOptions` option set and specify the `BlocksToHoldConstant` option.

```
opts = frestimateOptions;
opts.BlocksToHoldConstant = srcblks;
```

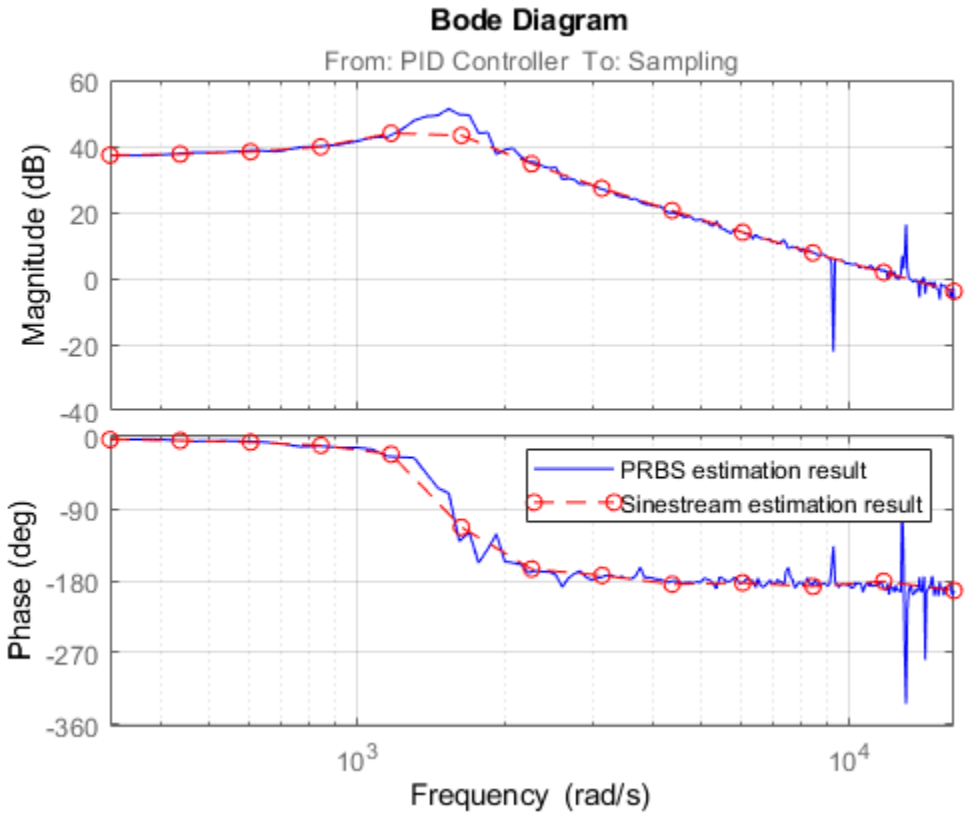
Estimate the frequency response using the PRBS input signal.

```
sysest_prbs = frestimate(mdl,io,op,in_PRBS,opts);
```

Compare Frequency Response Data to Sinestream FRE Results

Compare the estimation results when using PRBS signal to those found using a sinestream input signal. Compare the signals across the 15 logarithmically spaced frequencies used for the sinestream ranging from 50 Hz to 5 kHz.

```
load frdSinestream
wbode = estsysSinestream.Frequency;
bode(sysest_prbs,'b-');
hold on
bode(estsysSinestream,'ro--',wbode(1:end-2));
legend('PRBS estimation result','Sinestream estimation result',...
    'Location','northeast')
grid on
```



To find the simulation time taken for frequency response estimation by an input signal, you can use the `getSimulationTime` function.

```
tfinal_sinestream = in_sine1.getSimulationTime
tfinal_prbs = in_PRBS.getSimulationTime
```

```
tfinal_sinestream =

    0.2833


tfinal_prbs =

    0.0819
```

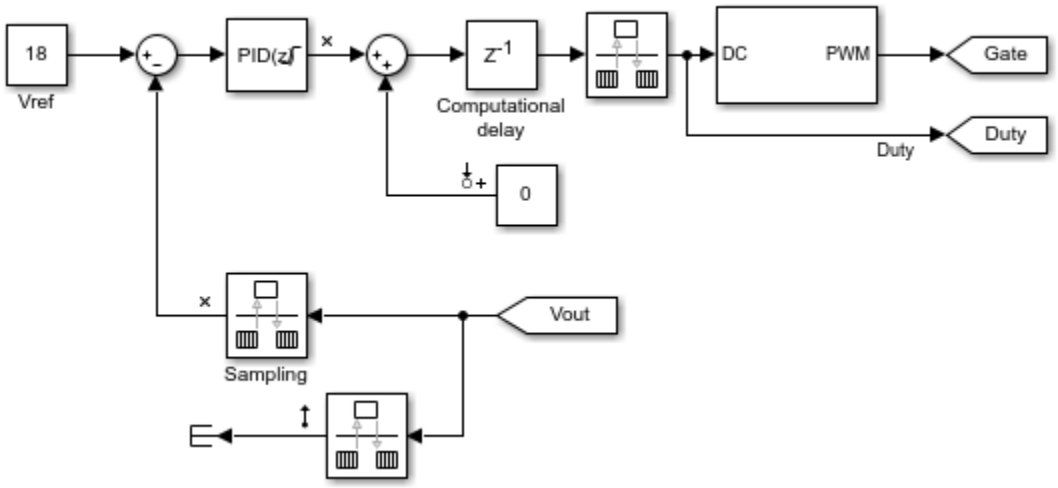
The simulation time with `in_PRBS` is about 30% of the time taken by `in_sine1` to estimate the frequency response of the model. This indicates that the frequency response estimation with PRBS input signal is much faster than the sinestream input signal.

The estimated frequency response result, `sysest_prbs`, closely matches `estsysSinestream`. Because the PRBS input signal estimates frequency responses with a large number of frequency points, the estimation result provides more information about the resonant characteristics of the system. To get similar results using sinestream input signal, you might need to increase the number of frequency points, which results in an increase in estimation time. You can use this approach to

obtain accurate frequency response estimation results in a shorter simulation time as compared to estimation with sinestream signals.

Improve Frequency Response Result at Low Frequencies

To improve the frequency response estimation result at lower frequencies, you can use a sample time slower than the sample time in the original boost converter model. To do so, modify the model to use a Constant block at the input analysis point and a Rate Transition block at the output analysis point.

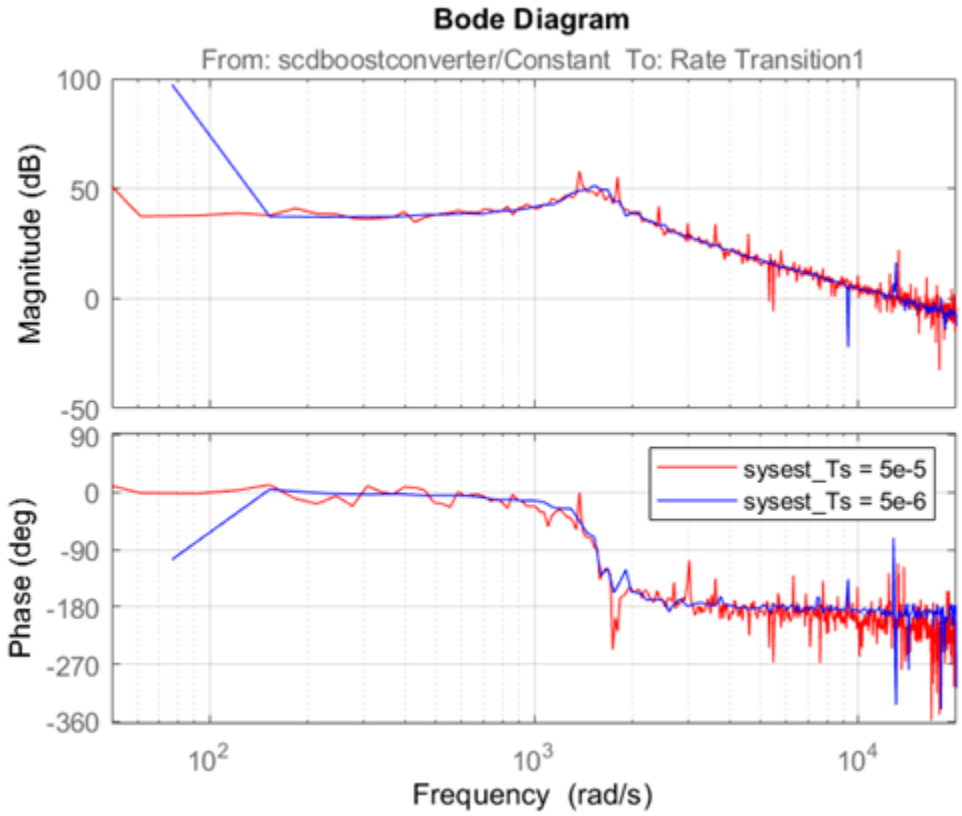


For both the Constant block and the Rate Transition block, use a sample time of `5e-5` seconds, which is ten times slower than the original sample time of `5e-6` seconds.

Create a PRBS input signal using the new sample time with an order of 12.

```
in_PRBS = frest.PRBS('Order',12,'NumPeriods',1,'Amplitude',0.05,'Ts',5e-5);
```

With the new PRBS input signal, you can obtain a frequency response that extends to lower frequencies.



The ability to change the sample time of the PRBS input signal time provides an additional degree of freedom in the frequency response estimation process. Using a larger sample time than in the original model, you can obtain a higher resolution frequency response estimation result over the low-frequency range with a short simulation time. Additionally, running estimation at lower sampling rate reduces processing requirements when deploying to hardware.

Close the model.

```
close_system(mdl,0)
```

References

[1] Lee, S. W. Practical Feedback Loop Analysis for Voltage-Mode Boost Converter. Application Report SLVA633. Texas Instruments, 2014. https://www.ti.com/lit/an/slva633/slva633.pdf.

See Also

`frest.PRBS` | `frestimate` | `frestimateOptions`

Related Examples

- “Frequency Response Estimation Basics” on page 5-2
- “Estimate Frequency Response at the Command Line” on page 5-14
- “PRBS Input Signals” on page 5-36
- “Estimation Input Signals” on page 5-24

- Frequency Response Estimation in Model Linearizer Using Pseudorandom Binary Sequence on page 5-101

Frequency Response Estimation in Model Linearizer Using Pseudorandom Binary Sequence

This example shows how to identify a frequency domain model using a pseudorandom binary sequence (PRBS) for a power electronics system modeled in Simulink® using Simscape™ Electrical™ components. This example addresses the frequency response estimation process in the controller design workflow using a PRBS as the input signal.

Typically, power electronics systems cannot be linearized because they use high-frequency switching components, such as pulse-width modulation (PWM) generators. However, most Simulink Control Design™ PID tuning tools design PID gains based on a linearized plant model. To obtain such a model for a power electronics model that cannot be linearized, you can estimate the plant frequency response over a range of frequencies, as shown in this example.

To collect frequency response data, you can:

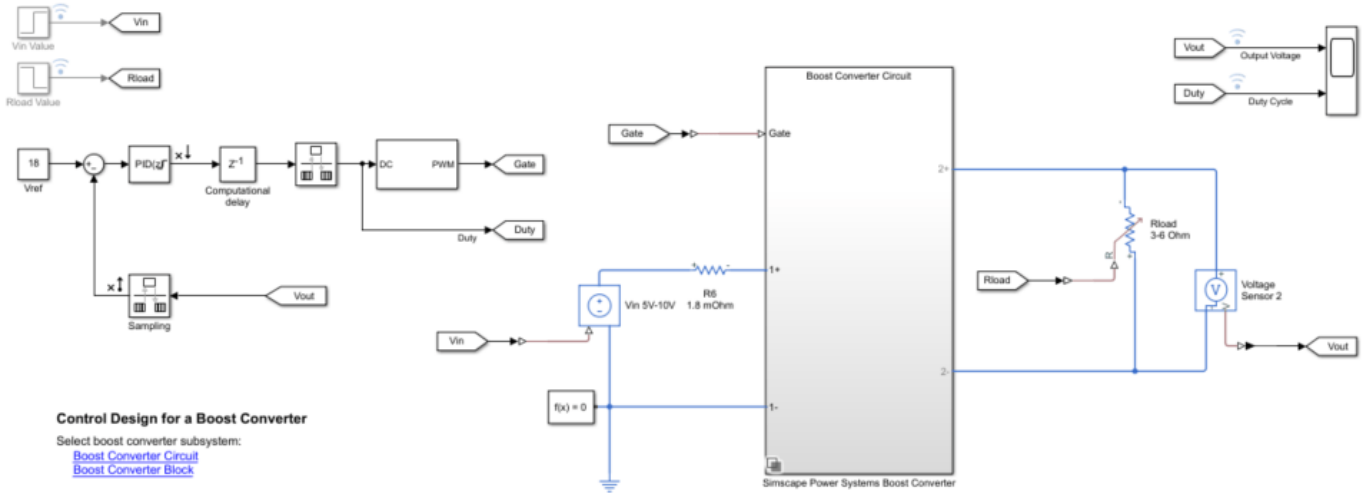
- Estimate the plant frequency response at the command line.
- Estimate the plant frequency response using the **Model Linearizer** app.

This example shows how to estimate the plant frequency response in **Model Linearizer**. To learn how to estimate the plant frequency response for a power electronics system at the command line using PRBS input signal, see “Frequency Response Estimation for Power Electronics Model Using Pseudorandom Binary Signal” on page 5-94.

Boost Converter Model

This example uses a boost converter model as a power electronics system. A boost converter circuit converts one DC voltage to another, typically higher, DC voltage by controlled chopping or switching of the source voltage.

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



This model uses a MOSFET driven by a PWM signal for switching. The output voltage *Vout* is regulated to the reference value *Vref*. A digital PID controller adjusts the PWM duty cycle, *Duty*,

based on the voltage error signal. For this example, you estimate the frequency response from the PWM duty cycle to the load voltage *Vout*.

Simscape Electrical software contains predefined blocks for many power electronics systems. This model contains a variant subsystem with two versions of the boost converter model:

- Boost converter circuit constructed using electrical power components. The parameters of the circuit components are based on [1] on page 5-0 .
- Boost converter block configured to have the same parameters as the boost converter circuit. For more information on this block, see Boost Converter (Simscape Electrical).

To use the Boost Converter block version of the subsystem, in the model, click **Boost Converter Block** or use the following command.

```
set_param([bdroot '/Simscape Power Systems Boost Converter'],...
    'OverrideUsingVariant','block_boost_converter');
```

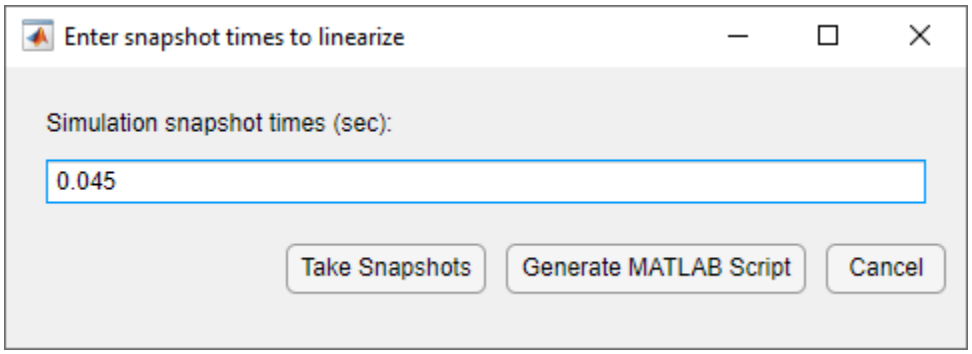
Open Model Linearizer and Find Model Operating Point

To open the **Model Linearizer**, in the Simulink model window, in the **Apps** gallery, click **Model Linearizer**.

To estimate the frequency response for the boost converter, you must first determine the steady-state operating point at which you want the converter to operate. For more information on finding operating points, see Find Steady-State Operating Points for Simscape Models. For this example, use an operating point estimated from a simulation snapshot at 0.045 seconds.

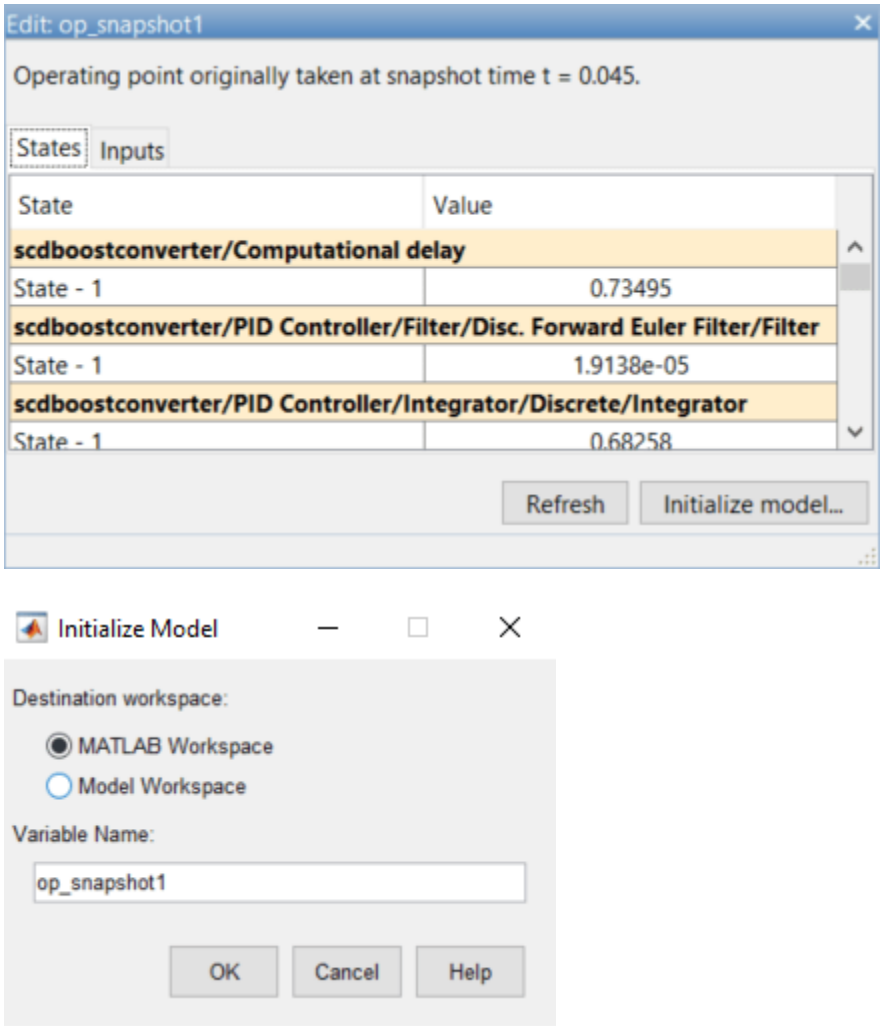
To specify the simulation snapshot time, in the **Model Linearizer**, on the **Linear Analysis** tab, in the **Operating Point** list, select `Take Simulation Snapshot`.

In the Enter snapshot times to linearize dialog box, in the **Simulation snapshot times** field, enter `0.045`.



Click **Take Snapshots** to find the model operating point. An operating point, `op_snapshot1`, appears in the **Data Browser**, in the **Linear Analysis Workspace** section.

To initialize the model with the computed operating point, double-click `op_snapshot1`, and then in the Edit dialog box click **Initialize Model**.



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

To export the operating point to the MATLAB® workspace and set the model initial condition to this operating point, click **OK**.

Specify Portion of Model to Estimate

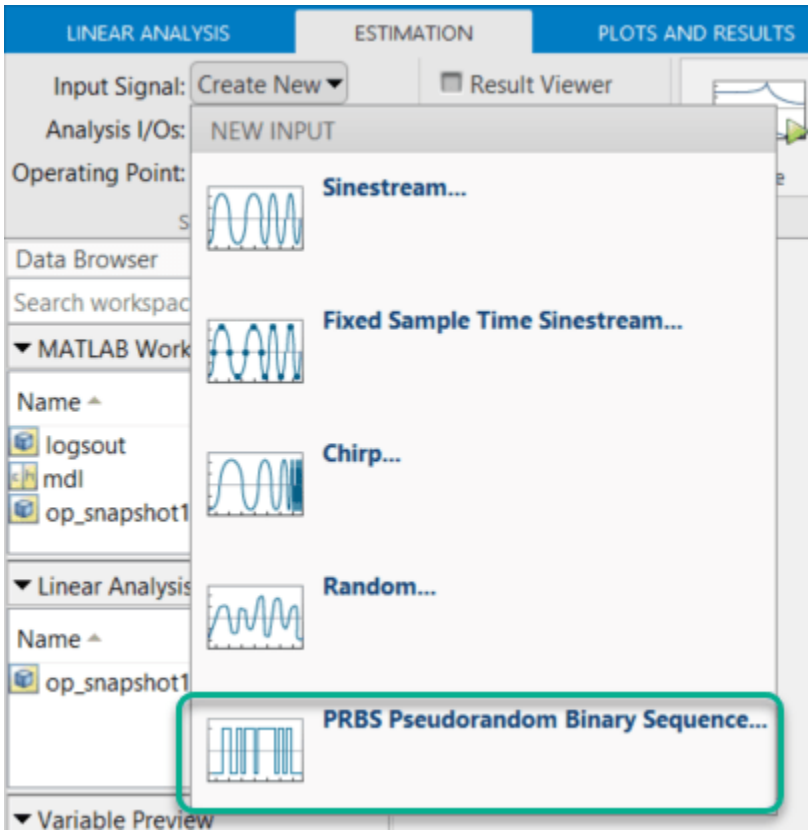
By default, **Model Linearizer** uses the linearization analysis points defined in the model (the *model I/Os*) to determine where to inject the test signal and where to measure the frequency response. The model `scdboostconverter` contains predefined linear analysis points: an input point at the PID controller output and an open-loop output before the negative feedback sum block.

If you want to obtain the frequency response of a different portion of the model, on the **Estimation** tab of **Model Linearizer**, use the **Analysis I/Os** drop-down list. Analysis points for estimation work the same way as analysis points for linearization. For more information about linear analysis points, see Specify Portion of Model to Linearize.

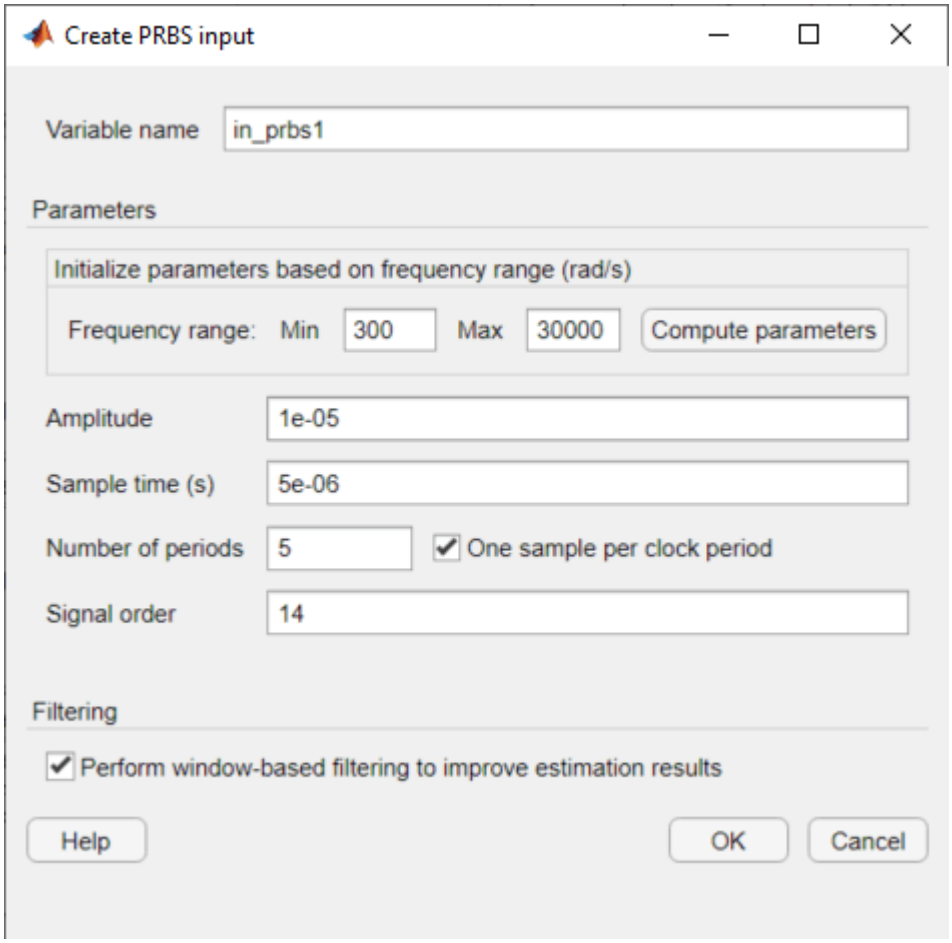
Create Pseudorandom Binary Sequence Input Signal

A PRBS is a periodic, deterministic signal with white-noise-like properties that shifts between two values. A PRBS is an inherently periodic signal with a maximum period length of $2^n - 1$, where n is the PRBS order.

To create a PRBS input signal, in the **Model Linearizer**, on the **Estimation** tab, under **Input Signal**, select **PRBS Pseudorandom Binary Sequence**.

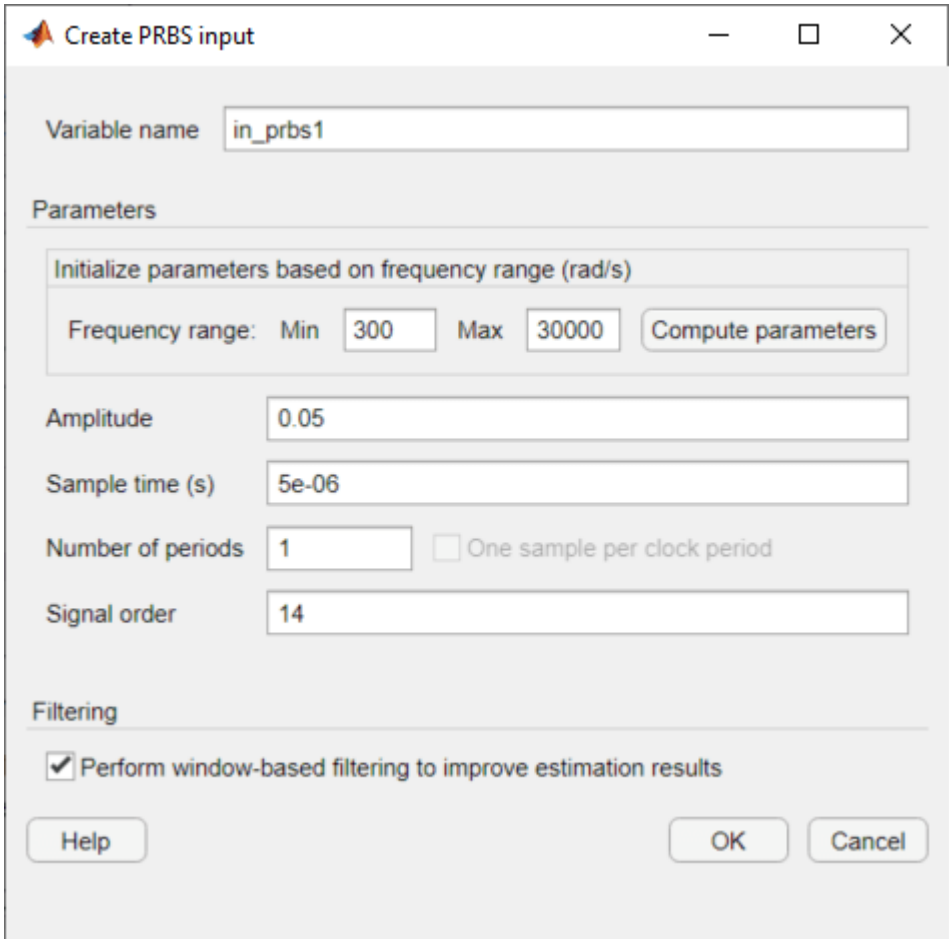


In the Create PRBS input dialog box, first set the **Sample time** to `5e-6` to match the sample time at the point of input signal injection in the model. Next, specify the frequency range as 300 rad/s to 30000 rad/s using the **Min** and **Max** parameters, and then click **Compute Parameters**. The software computes the signal parameters **Number of periods** and **Signal order** based on the specified frequency range. Automatic parameter determination helps create an input signal that leads to an accurate frequency response over a specified frequency range.



To use a nonperiodic PRBS, set the **Number of periods** parameter to `1`.

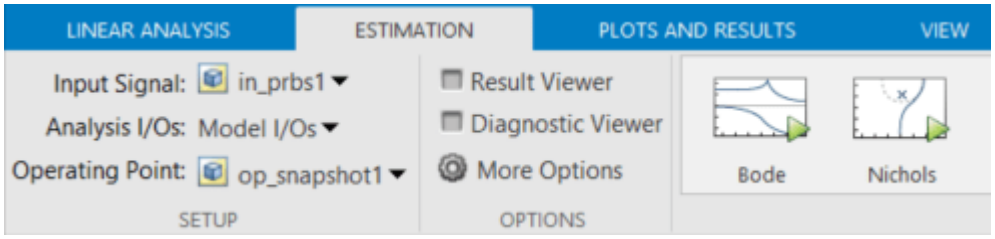
To ensure that the system is properly excited, set the perturbation amplitude to `0.05` using the **Amplitude** parameter. If the input amplitude is too large, the boost converter operates in discontinuous-current mode. If the input amplitude is too small, the PRBS is indistinguishable from ripples in the power electronic circuits.



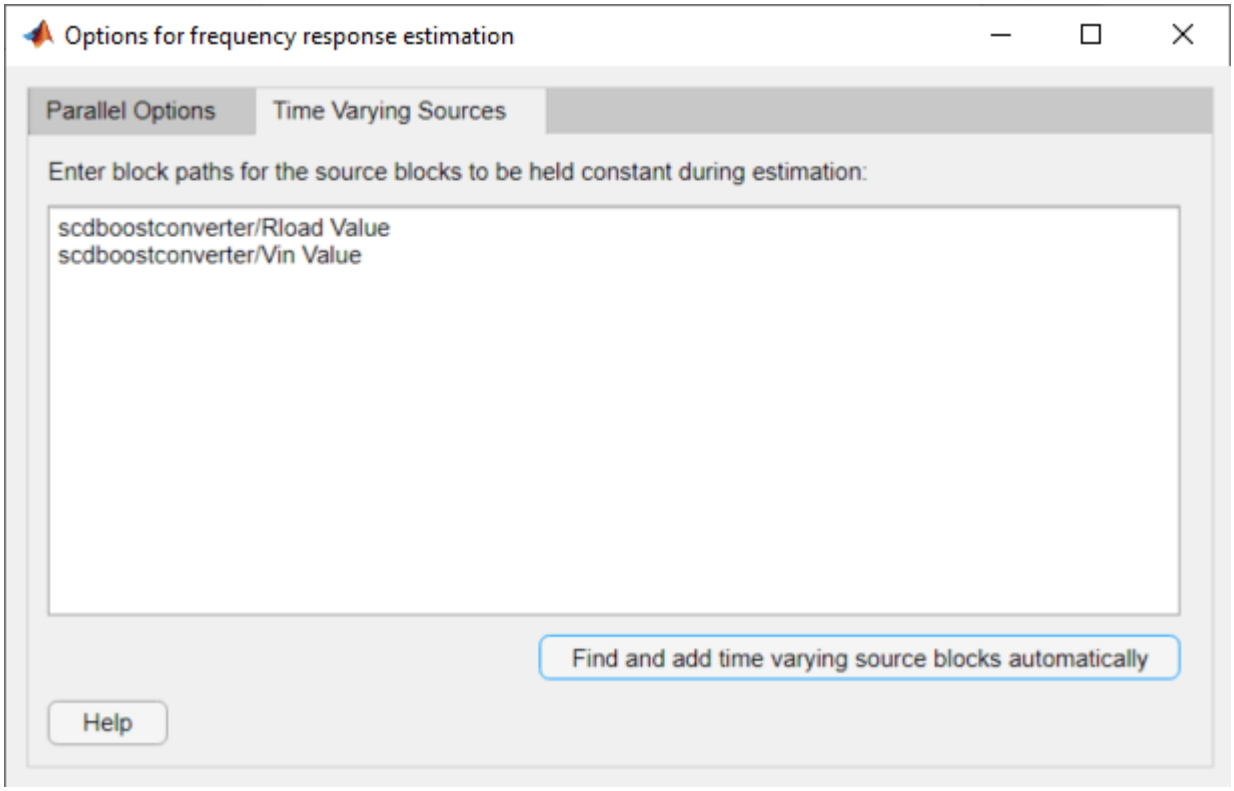
Click **OK**. The software adds the PRBS signal to the **Linear Analysis Workspace**.

Collect Frequency Response Data

In the **Estimation** tab, select `op_snapshot1` as the estimation operating point.

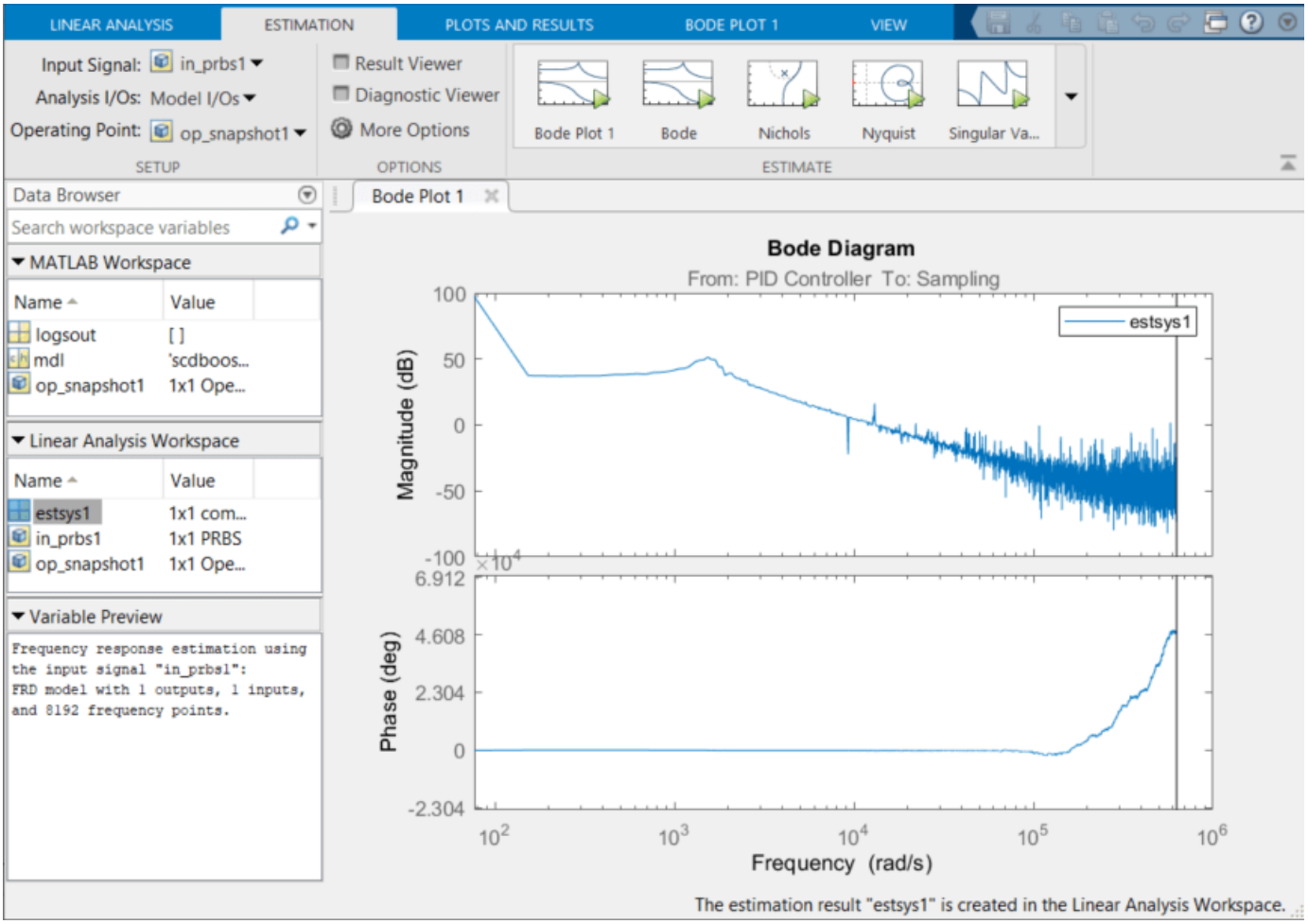


Find all source blocks in the signal paths of the linearization outputs that generate time-varying signals. Such time-varying signals can interfere with the signal at the linearization output points and produce inaccurate estimation results. To disable the time-varying source blocks, click **More Options**. In the Options for frequency response estimation dialog box, on the **Time Varying Sources** tab, click **Find and add time varying source blocks automatically**.



Close the dialog box after block paths appear.

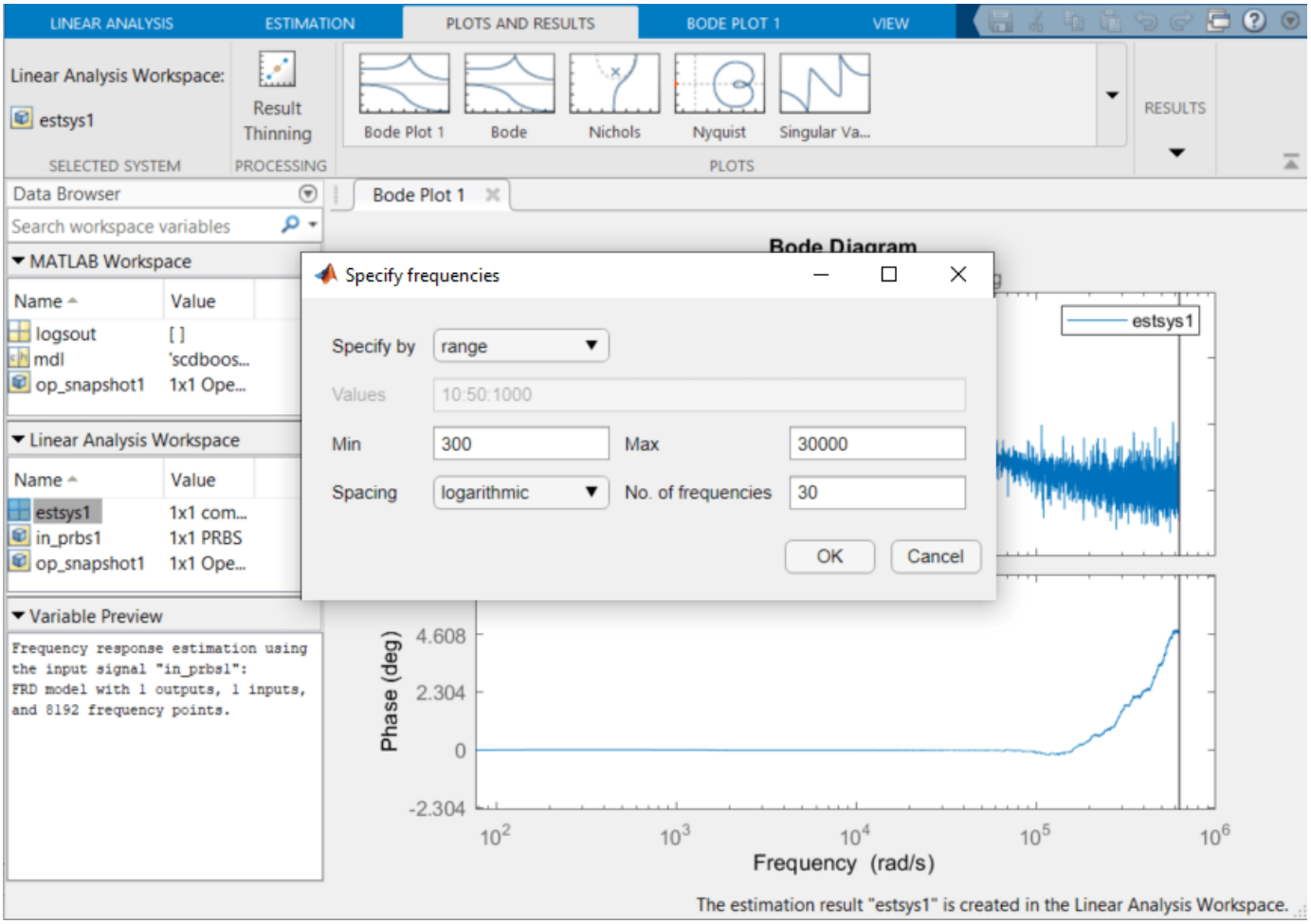
To estimate and plot the frequency response, on the **Estimation** tab, click **Bode**. The estimated system, `estsys1`, appears in the **Linear Analysis Workspace** and is added to **Bode Plot 1**.



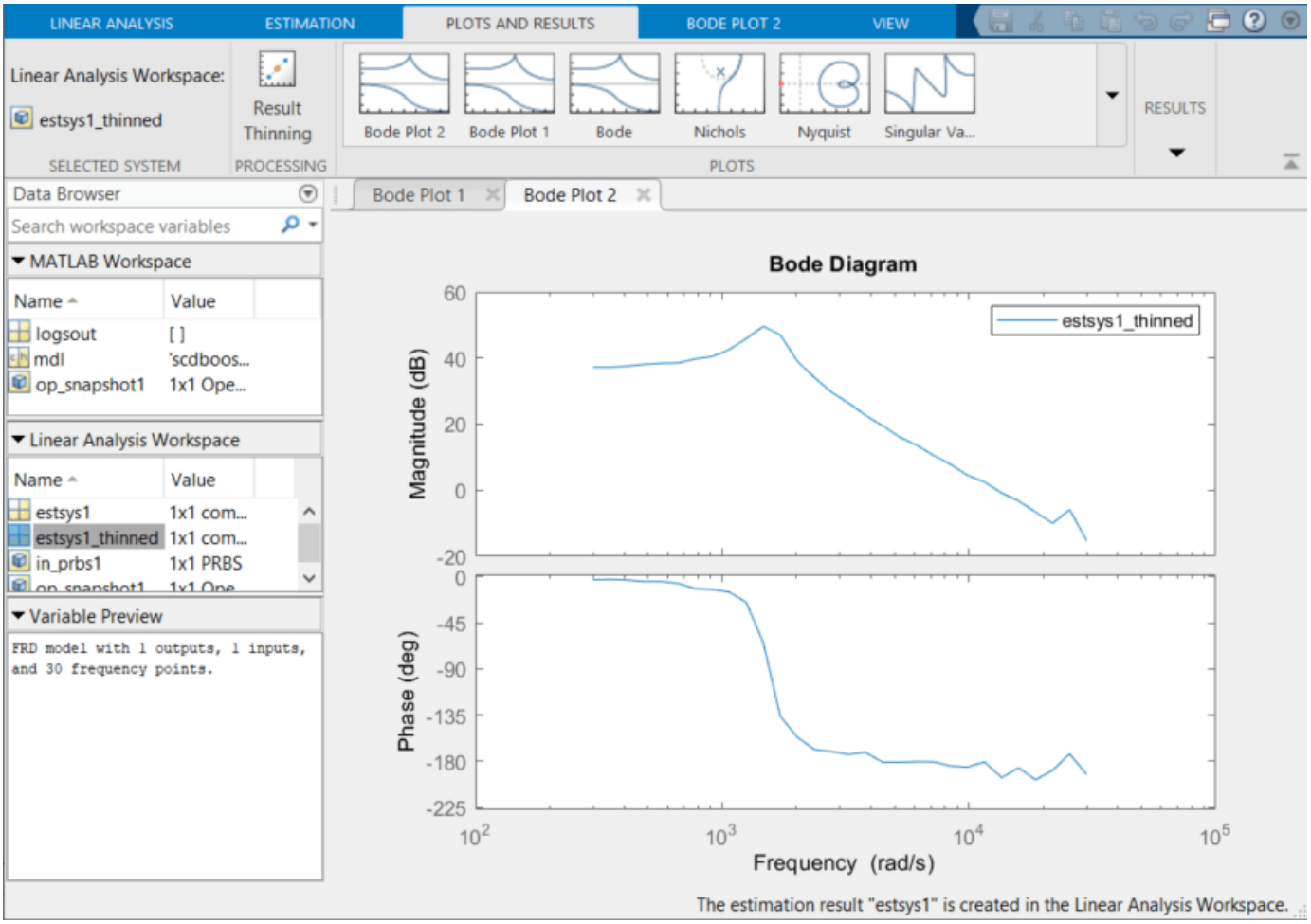
Frequency Response Estimation Result Thinning

Frequency response estimation with PRBS input signal produces results with a large number of frequency points. You can use the “Result Thinning” on page 5-23 functionality to extract an interpolated result from an estimated frequency response model across a specified frequency range and number of frequency points.

To extract the result over the desired frequency range of 300 rad/s to 30000 rad/s, select `estsys1` in the **Linear Analysis Workspace**, and on the **Plots and Results** tab click **Result Thinning.** In the Specify frequencies dialog box, enter a frequency range 300 rad/s to 30000 rad/s. Also, specify 30 logarithmically spaced frequency points.



Click **OK**. The estimated system, `estsys1_thinned`, appears in the **Linear Analysis Workspace.** To plot the thinned result, select `estsys1_thinned` and click **Bode.**



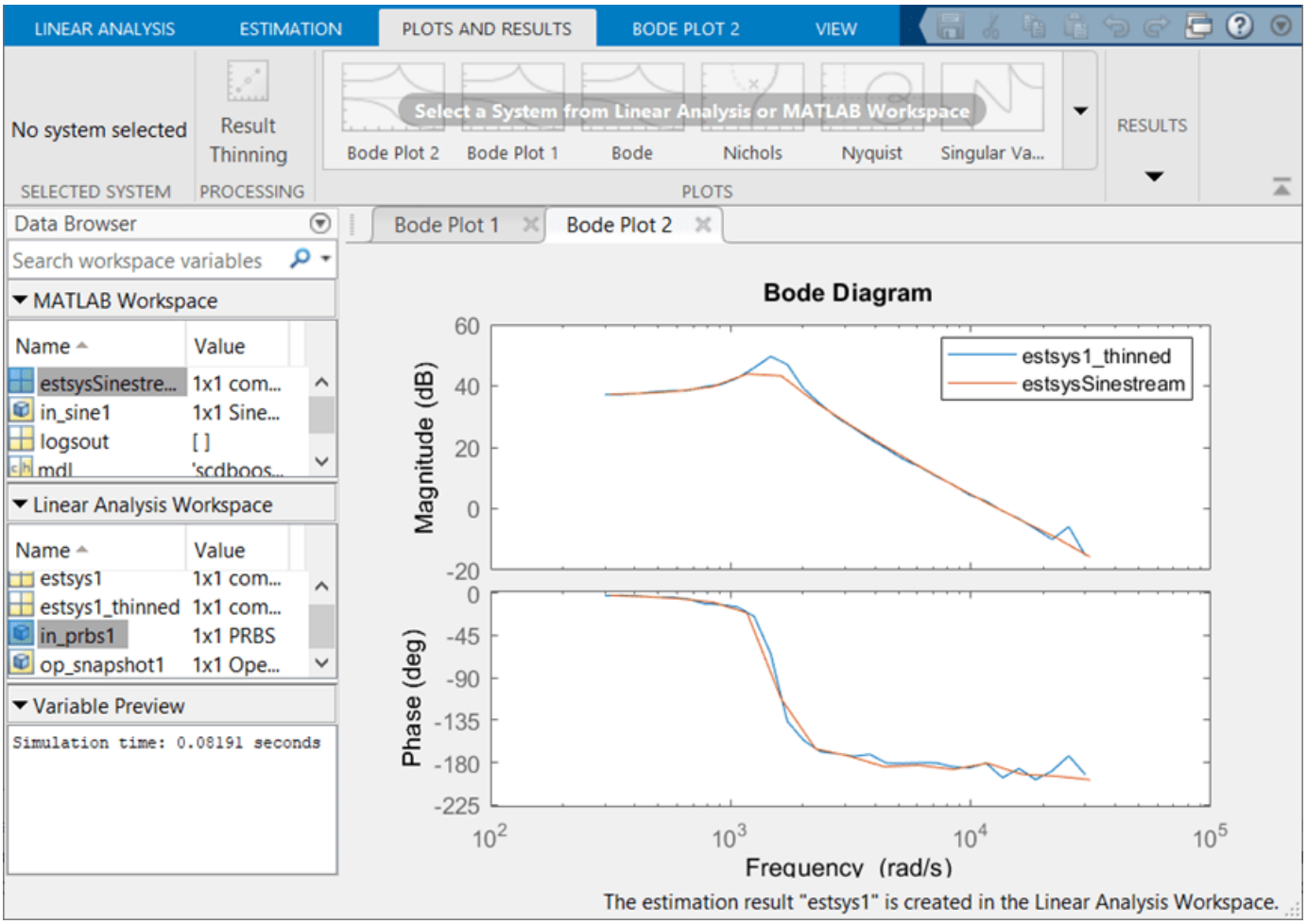
Compare Frequency Response Data to Sinestream FRE Results

In the **Model Linearizer**, you can also compare the frequency response data with the result obtained using a sinestream signal. Load the supplied frequency response estimation result using the following command.

```
load frdSinestream
```

The result `estsysSinestream` is a model with 15 logarithmically spaced frequencies estimated using a sinestream ranging from 50 Hz to 5 kHz.

To compare the sinestream result to the thinned result in the **Model Linearizer**, select `estsysSinestream` in the **MATLAB Workspace** pane of **Data Browser** and click **Bode Plot 2**.



To find the final simulation time taken for frequency response estimation by an input signal, select the input signal in the **Linear Analysis Workspace** and view the simulation time in the **Variable Preview** pane of the **Model Linearizer**. Alternatively, you can export the input signals to MATLAB Workspace and use the `getSimulationTime` function. Load a previously saved session for this example.

```
load boostconv_frdPRBS.mat
tfinal_sinestream = in_sine1.getSimulationTime
```

```
tfinal_sinestream = 0.2833
```

```
tfinal_prbs = in_prbs1.getSimulationTime
```

```
tfinal_prbs = 0.0819
```

The simulation time with `in_prbs1` is about 30% of the time taken by `in_sine1` to estimate the frequency response of the model. This indicates that the frequency response estimation with PRBS input signal is much faster than the sinestream input signal.

The estimated frequency response result `estsys1_thinned` closely matches `estsysSinestream`. Because the PRBS input signal estimates frequency responses with a large number of frequency points, the estimation result provides more information about the resonant characteristics of the system. To get similar results using a sinestream input signal, you might need to increase the number

of frequency points, which results in an increase in estimation time. You can use this approach to obtain accurate frequency response estimation results in a shorter simulation time as compared to estimation with sinestream signals.

Close the model.

```
close_system(mdl,0)
```

References

[1] Lee, S. W. Practical Feedback Loop Analysis for Voltage-Mode Boost Converter. Application Report SLVA633. Texas Instruments, 2014. https://www.ti.com/lit/an/slva633/slva633.pdf.

See Also

Model Linearizer

Related Examples

- “Frequency Response Estimation Basics” on page 5-2
- “PRBS Input Signals” on page 5-36
- “Estimation Input Signals” on page 5-24
- “Estimate Frequency Response Using Model Linearizer” on page 5-6
- “Estimate Frequency Response with Linearization-Based Input Using Model Linearizer” on page 5-10
- “Frequency Response Estimation for Power Electronics Model Using Pseudorandom Binary Signal” on page 5-94

Frequency Response Estimation for Permanent Magnet Synchronous Motor Model

This example shows the frequency response estimation workflow for a Field-Oriented Controlled (FOC) three-phase Permanent Magnet Synchronous Motor (PMSM) modeled using components from Motor Control Blockset™. This example uses **Model Linearizer** from Simulink® Control Design™ software to obtain a frequency-response model (`frd`) object, which you can use to estimate a parametric model for the motor.

PMSM Model

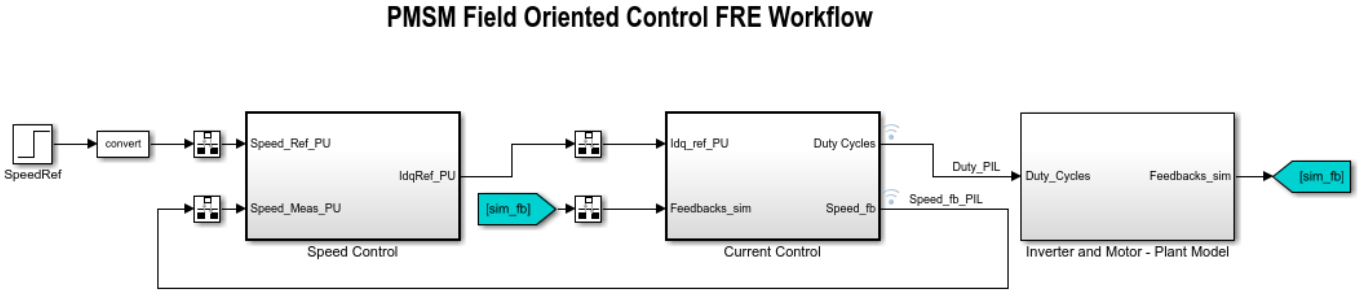
The PMSM model is based on the Motor Control Blockset™ `mcb_pmsm_foc_sim` model. The model includes:

- A subsystem to model inverter and PMSM dynamics
- Inner loop (current) and outer loop (speed) PI controllers to implement a field-oriented control algorithm for motor speed control

You can examine this model for more details. In this example, the original model is modified to ensure that the simulation starts from a steady state. The steady state serves as the operating point used in the frequency response estimation workflow.

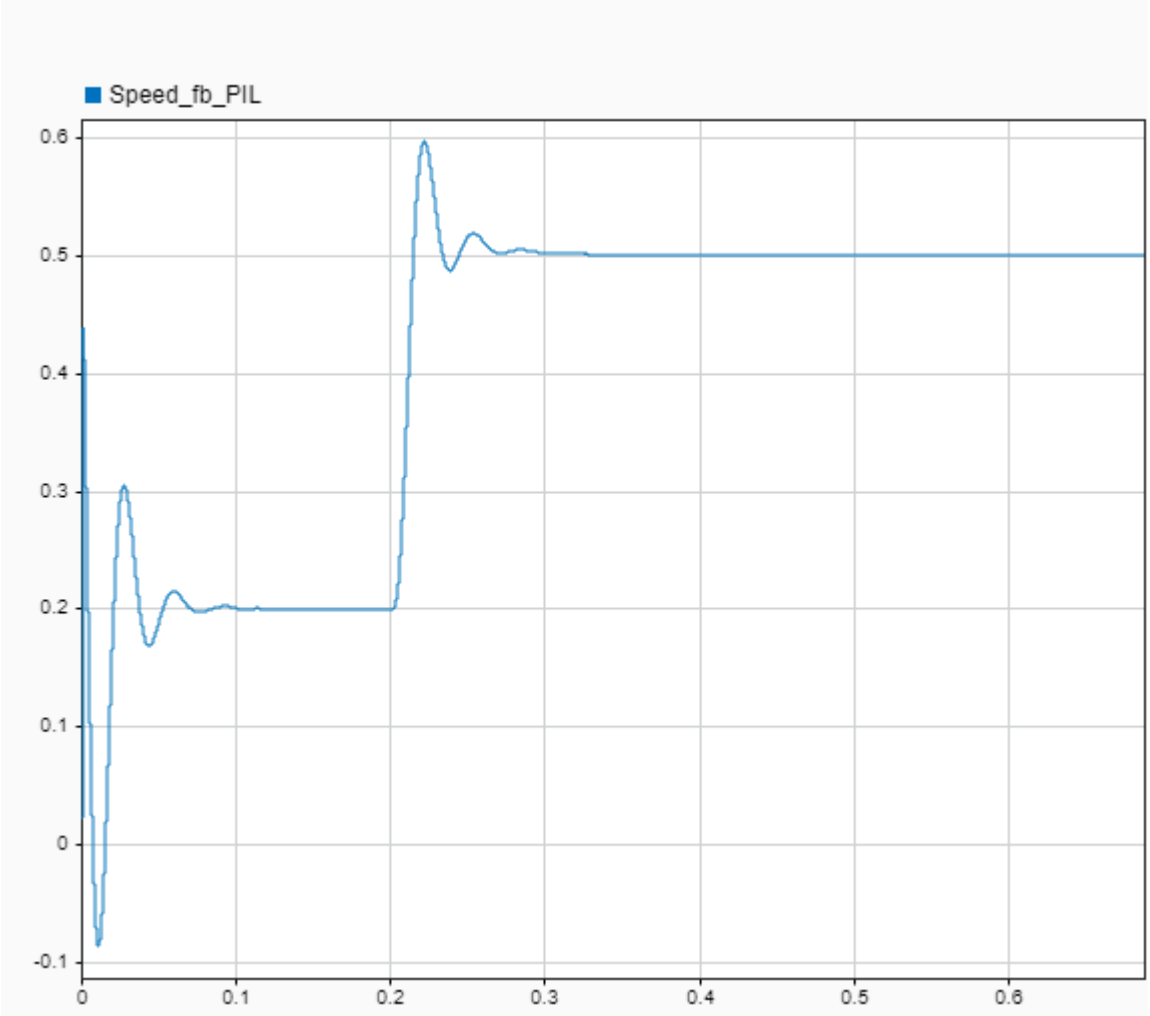
Open the Simulink® model.

```
model = 'scd_fre_mcb_pmsm_foc_sim';
open_system(model)
```



Modify Model to Start Simulation from Steady State

To ensure that the simulation starts from a steady state, modify initial conditions of the original model. To obtain these initial conditions, enable signal logging for the speed feedback signal and simulate the model to the steady state, with a speed of 0.5 p.u. To ensure that the speed reaches the desired steady state, after the simulation, examine the simulation result in the Simulation Data Inspector.



Based on the speed response in the preceding figure, you can use the simulation results after 0.6 seconds to obtain block initial conditions for frequency response estimation. In addition to the initial conditions, modify settings in filters associated with the speed control loop for faster speed reference tracking. The changes are intended to make the simulation start from a steady state but do not affect the motor plant model. Make the following changes to the model.

- In the SpeedRef block, set **Step time** to `0` s and **Initial value** to the steady state reference value `0.5` p.u.
- In the Speed Control subsystem, in the Zero_Cancellation block, set **Filter coefficient** to `1` for faster tracking.
- In the Speed Control > PI_Controller_Speed subsystem, in the Ki2 block, set the **Constant value** to `0.01725`. That is, the speed controller's initial value, `y0`.
- In the Current Control > Input Scaling > Calculate Position and Speed subsystem, in the Speed Measurement block, set the **Delays for speed calculation** to `1` for faster speed measurement.
- In the Current Control > Control_System > Closed Loop Control > Current_Controllers > PI_Controller_Id subsystem, in the Ki1 block, set the **Constant value** to `0.025`. That is, d-axis controller's initial value, `y0`.

- In the Current Control > Control_System > Closed Loop Control > Current_Controllers > PI_Controller_Iq subsystem, in the Kp1 block, set the **Constant value** to `0.435`. That is, q-axis controller's initial value, `y0`.
- In Inverter and Motor - Plant Model subsystem, in the Surface Mount PMSM block, set **Initial d-axis and q-axis current (idq0)** to `[-0.4 0.55]` and **Initial rotor mechanical speed (omega_init)** to `215`.

Alternatively, you can set the block parameters using the following commands.

```
set_param([model,'/SpeedRef'],'Time','0','Before','0.5')
set_param([model,'/Speed Control/Zero_Cancellation'],'Filter_constant','1')
set_param([model,'/Speed Control/PI_Controller_Speed/Ki2'],'Value','0.01725')
set_param([model,...
    '/Current Control/Input Scaling/ Calculate Position and Speed/Speed Measurement'],...
    'DelayLength','1')
currCtrlsPath = '/Current Control/Control_System/Closed Loop Control/Current_Controllers/';
set_param([model,currCtrlsPath,'PI_Controller_Id/Ki1'],'Value','0.025')
set_param([model,currCtrlsPath,'PI_Controller_Iq/Kp1'],'Value','0.435')
set_param([model,'/Inverter and Motor - Plant Model/Surface Mount PMSM'], ...
    'idq0','[-0.4 0.55]','omega_init','215')
```

Frequency Response Estimation Using Fixed Sample Sinestream Signal

After the necessary changes, the simulation starts from a steady state with the motor speed around 0.5 p.u. You can then use **Model Linearizer** to conduct frequency response estimation. To open the **Model Linearizer**, in the Simulink model window, in the **Apps** gallery, click **Model Linearizer**.

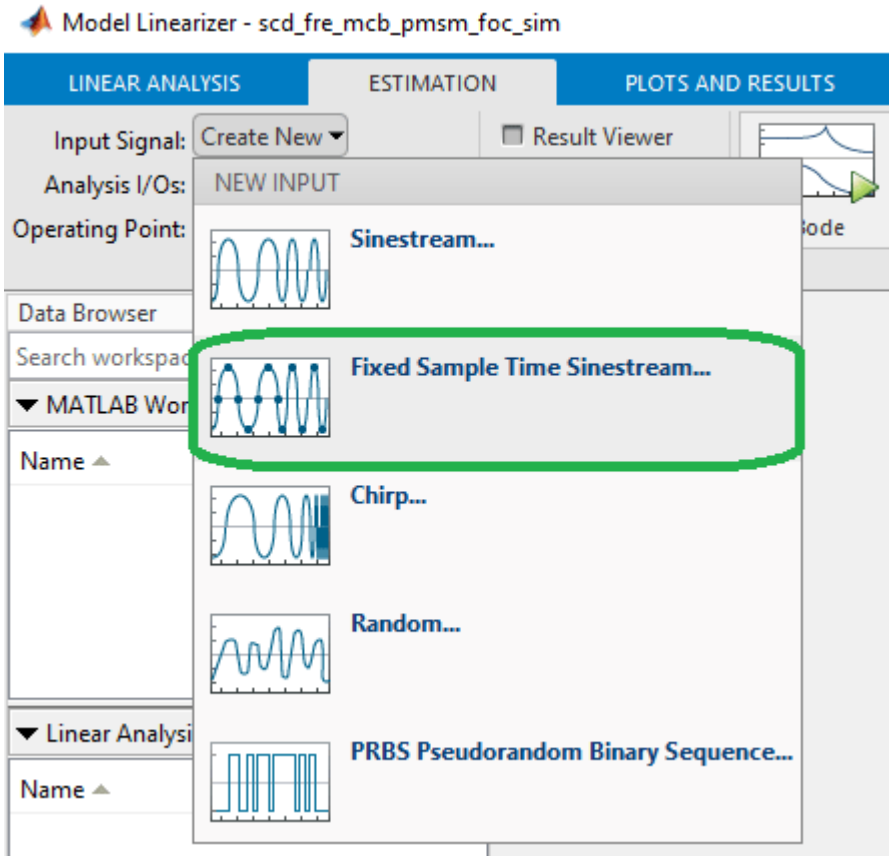
To collect frequency response data, you must first specify the portion of model to estimate. By default, **Model Linearizer** uses the linearization analysis points defined in the model (the *model I/Os*) to determine where to inject the test signal and where to measure the frequency response. The model `scd_fre_mcb_pmsm_foc_sim` contains predefined linear analysis points:

- Input points at the outputs of d-axis and q-axis current PI controllers
- Output points at speed, d-axis current, and q-axis current feedback signals

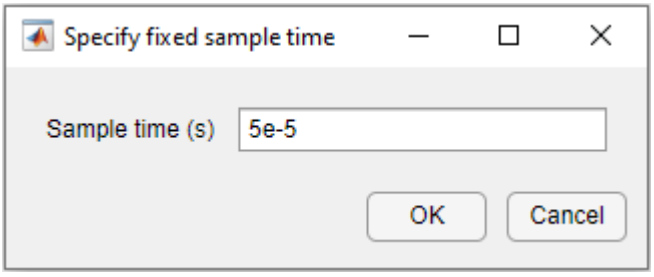
To view or edit these analysis points, on the **Estimation** tab of **Model Linearizer**, on the **Analysis I/Os** list, click `Edit Model I/Os`. Analysis points for estimation work the same way as analysis points for linearization. For more information about linear analysis points, see “Specify Portion of Model to Linearize” on page 2-10.

Frequency response estimation injects an input signal into the input analysis points you specify for estimation. For this example, use a fixed sample time sinestream input signal for estimation. Create a signal with 20 frequency points from 1 rad/s to 1000 rad/s and an amplitude of 0.05. For more information on defining sinestream input signals, see “Sinestream Input Signals” on page 5-29.

To create the signal, on the **Estimation** tab of the Model Linearizer, in the **Input Signal** list, select `Fixed Sample Time Sinestream`.



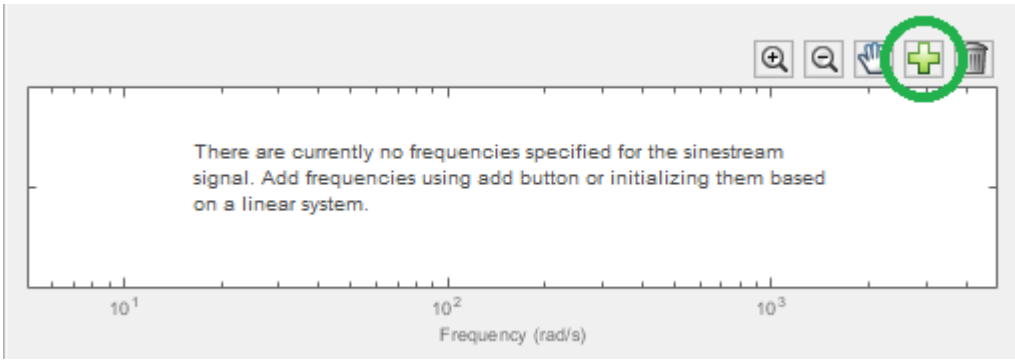
In the Specify fixed sample time dialog box, specify a **Sample time** of `5e-5` seconds.



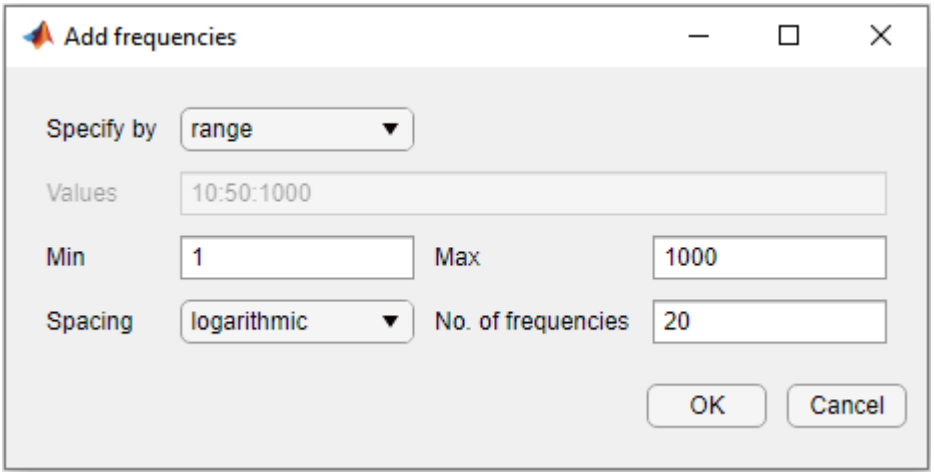
Click **OK**. The Create sinestream input with fixed sample time dialog box opens.

Specify the frequency units for estimation. In the **Frequency units** list, select `rad/s`.

To specify the frequencies at which to estimate the plant response, click the add frequencies icon.

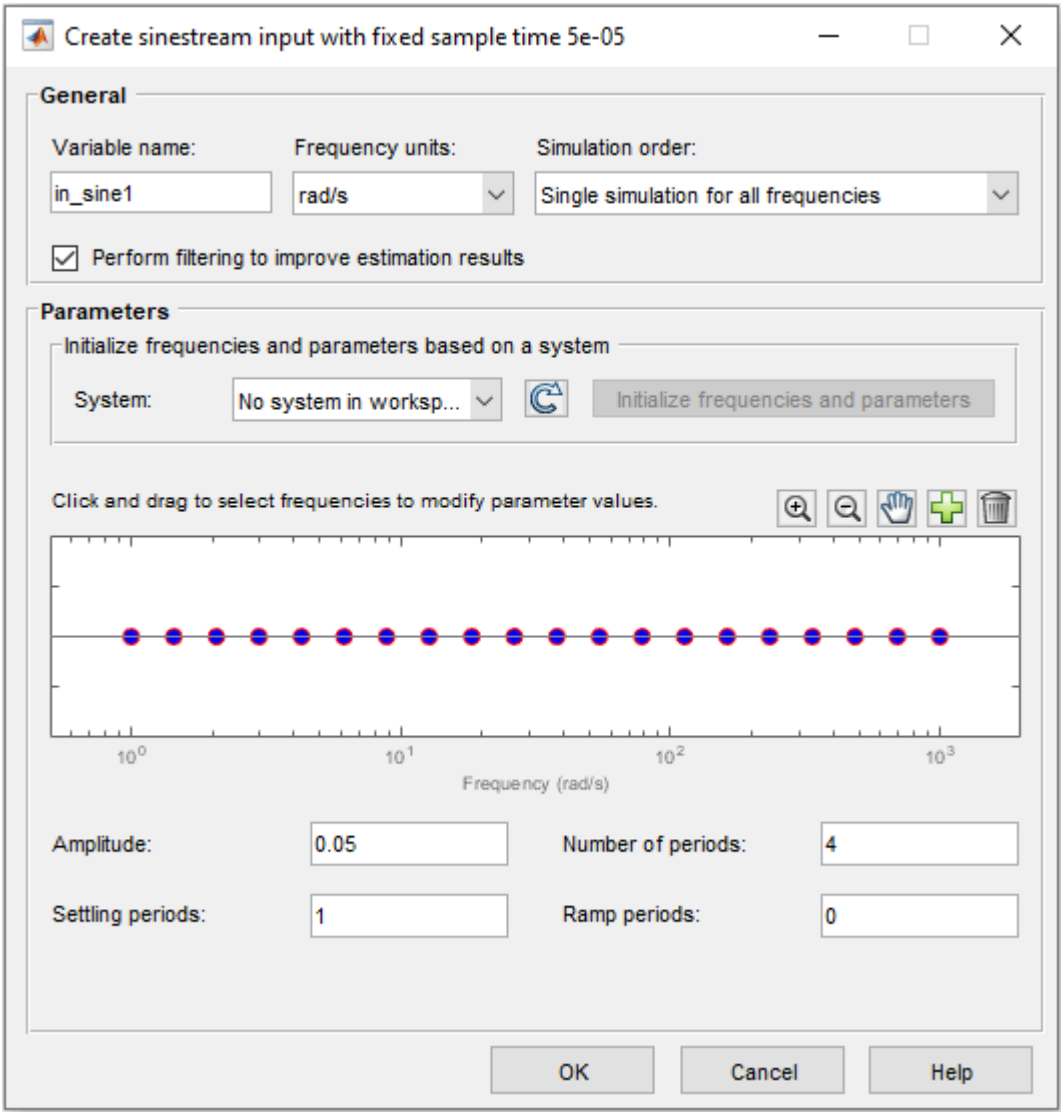


In the Add frequencies dialog box, specify 20 logarithmically spaced frequencies ranging from 1 rad/s to 1000 rad/s.



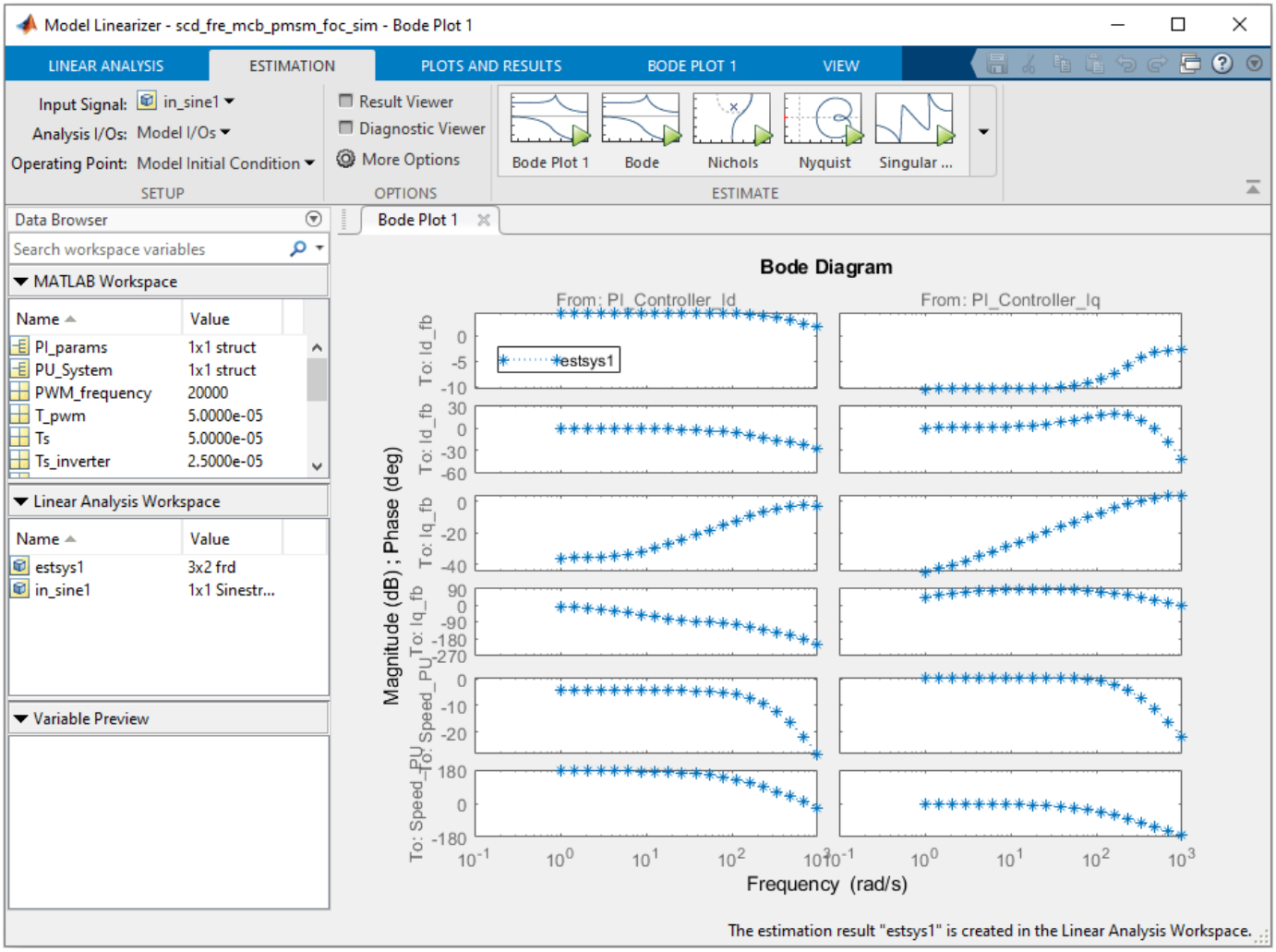
Click **OK**. The added points are visible in the frequency content viewer of the Create sinestream input with fixed sample time dialog box.

To specify the amplitude of the input signal, first select all the frequencies in the plot area. Then, in the **Amplitude** field, enter `0.05`. Use default values for the remaining parameters.



Click **OK** to create the fixed sample time sinestream signal.

To estimate and plot the frequency response, one the **Estimation** tab, click **Bode**. The estimated frequency response appears in the **Linear Analysis Workspace** as the `frd` model `estsys1` and response is added to **Bode Plot 1**.

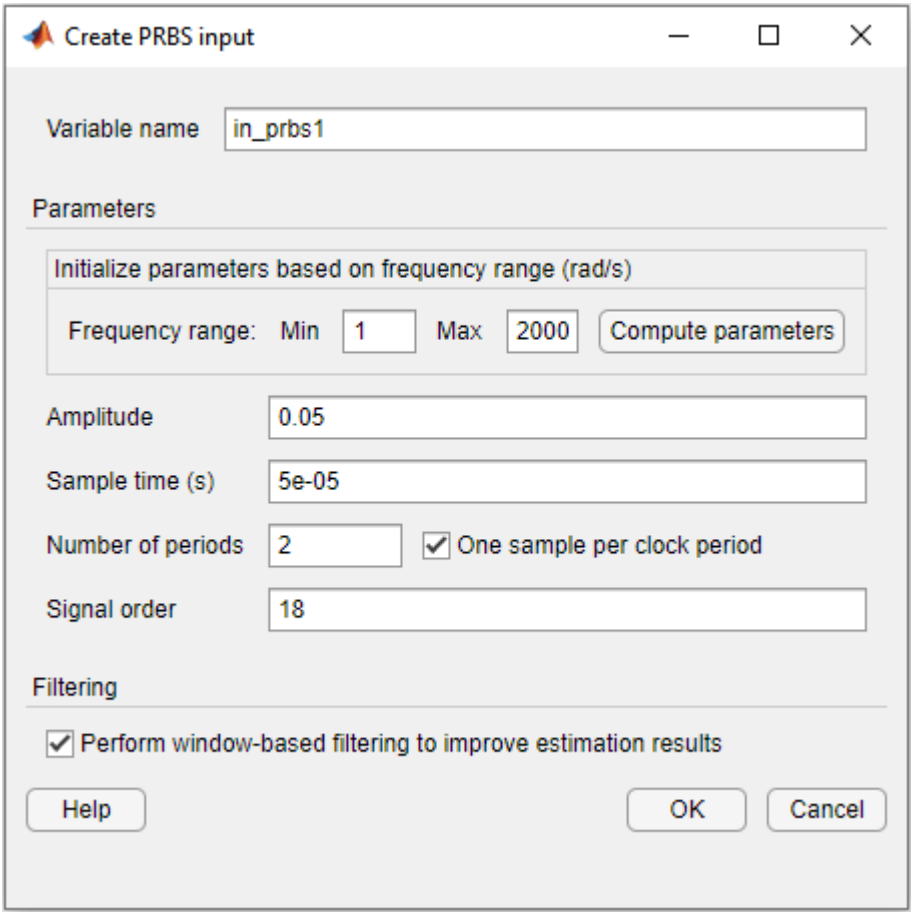


Frequency Response Estimation Using PRBS Input Signal

In addition to sinestream input signals, you can also use PRBS input signals in frequency response estimation. For this example, create an input signal with 2 periods, an order of 18, and an amplitude of 0.05.

To create the signal, on the **Estimation** tab of the **Model Linearizer**, in the **Input Signal** list, click `PRBS Pseudorandom Binary Sequence`.

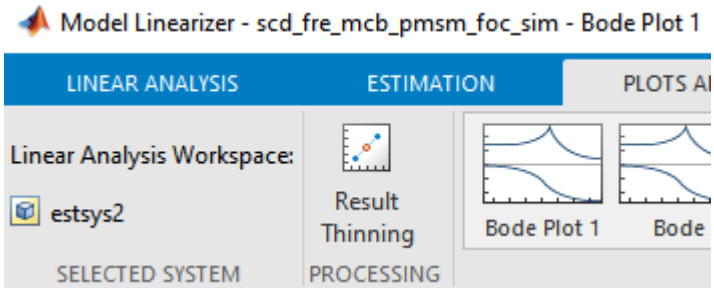
In the Create PRBS input dialog box, configure the parameters of the PRBS signal. The input dialog box assists you to set the signal order and number of periods based on the frequency range you are interested in. First, set the **Sample time** to `5e-5` seconds. Next, enter a frequency range between 1 rad/s and 2000 rad/s, and then click **Compute Parameters**. The software computes the signal parameters **Number of periods** and **Signal order**. To ensure that the system is properly excited, set the perturbation amplitude to 0.05 using the **Amplitude** parameter. Use default values for the remaining parameters.



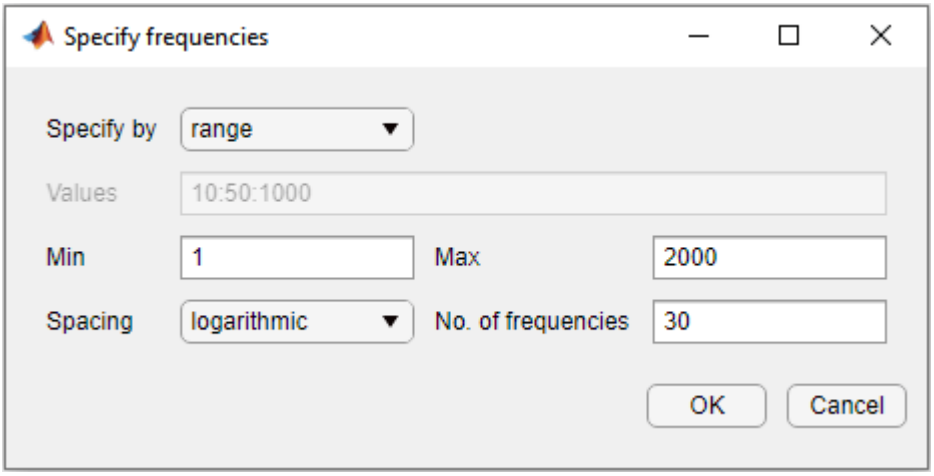
Click **OK** to create the input signal.

To estimate and plot the frequency response, on the **Estimation** tab, click **Bode**. The estimated frequency response appears in the **Linear Analysis Workspace** as the `frd` model `estsys2`. Frequency response estimation with PRBS input signal produces results with a large number of frequency points. You can use the “Result Thinning” on page 5-23 functionality to extract an interpolated result from an estimated frequency response model across a specified frequency range and number of frequency points.

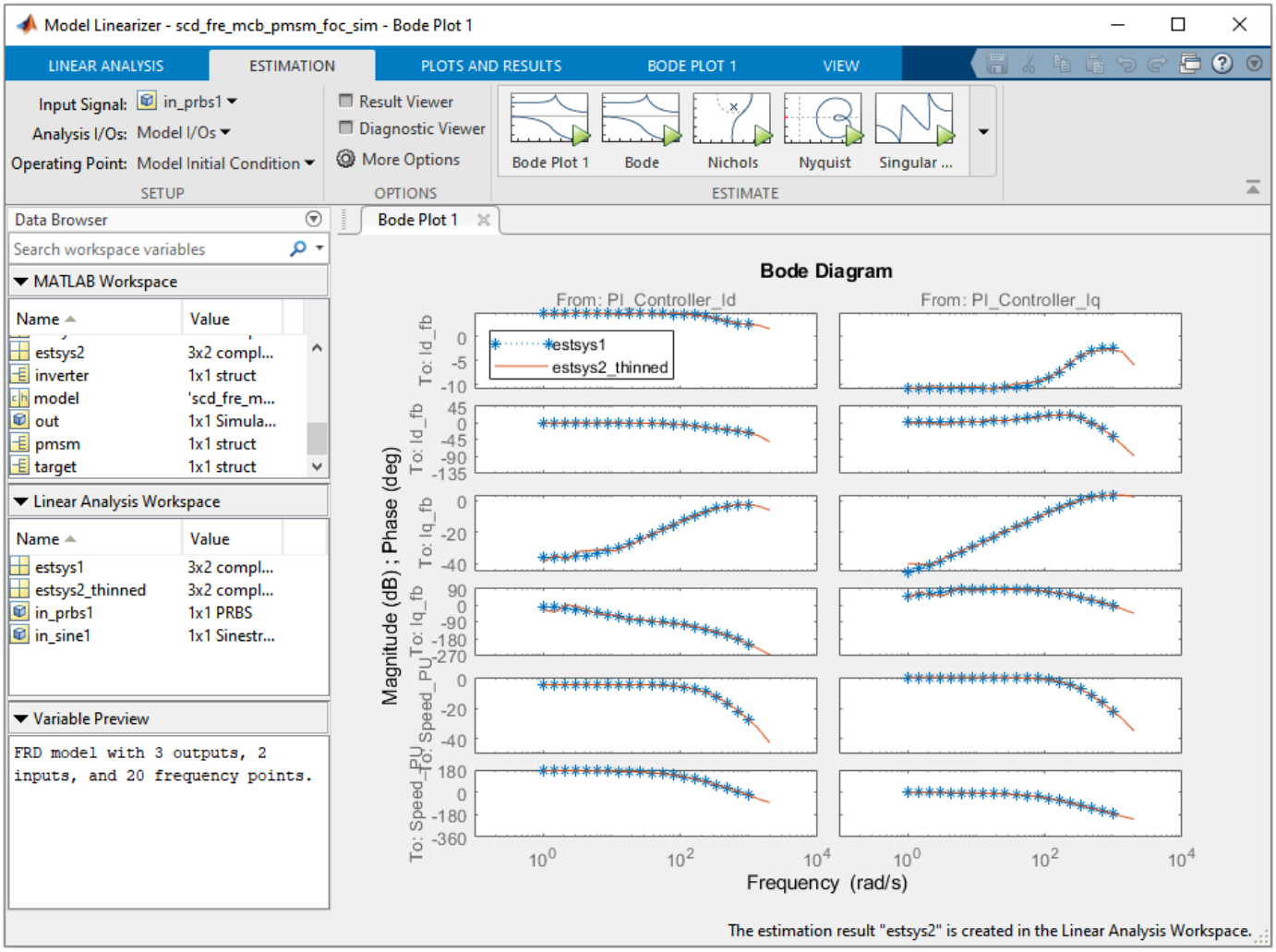
To apply thinning to the estimated result, select `estsys2` in the **Linear Analysis Workspace**, and on the **Plots and Results** tab click **Result Thinning**.



In the Specify frequencies dialog box, specify a frequency range between 1 rad/s and 2000 rad/s with 30 logarithmically spaced points.



Click **OK**. The thinned estimated system, `estsys2_thinned`, appears in the **Linear Analysis Workspace**. To compare the result with `estsys1`, click **Bode Plot 1**.



To find the final simulation time for frequency response estimation by an input signal, select the input signal in the **Linear Analysis Workspace** and view the simulation time in the **Variable Preview** area of the **Model Linearizer**. Alternatively, you can export the input signals to the MATLAB® workspace and use the `getSimulationTime` function. Load a previously saved session and display the simulation times.

```
load matlab_FRE_comparison.mat
tfinal_sinestream = in_sine1.getSimulationTime
tfinal_prbs = in_prbs1.getSimulationTime
```

```
tfinal_sinestream =

   82.3973

tfinal_prbs =

   26.2142
```

Both input signals provide similar performances in frequency response estimation. You can use PRBS input signals to obtain accurate frequency response estimation results in a shorter simulation time as compared to estimation with sinestream signals. However, since PRBS estimation results contain a large number of frequency points, you must thin them for accurate parametric estimation.

Estimate Parametric Model from Frequency Response Result

Parametric models, such as transfer function models and state-space models, are widely used in control design workflows. You can estimate a parametric model from the frequency response estimation result.

To estimate a parametric model for the PMSM motor, first export `estsys1` or `estsys2_thinned` to MATLAB workspace, then use `tfest` or `ssest` functions from the System Identification Toolbox™ software to estimate a transfer function model or a state-space model, respectively.

Close the model.

```
close_system(model,0)
```

See Also

Model Linearizer

Related Examples

- “Frequency Response Estimation Basics” on page 5-2
- “Estimation Input Signals” on page 5-24
- “Estimate Frequency Response Using Model Linearizer” on page 5-6
- “Estimate Frequency Response with Linearization-Based Input Using Model Linearizer” on page 5-10

Frequency Response Estimation to Measure Input Admittance of Boost Converter

This example shows how to measure the input admittance of a boost converter modeled in Simulink® using Simscape™ Electrical™ components. This example addresses the frequency response estimation process to measure the input admittance of an existing power electronics circuit model.

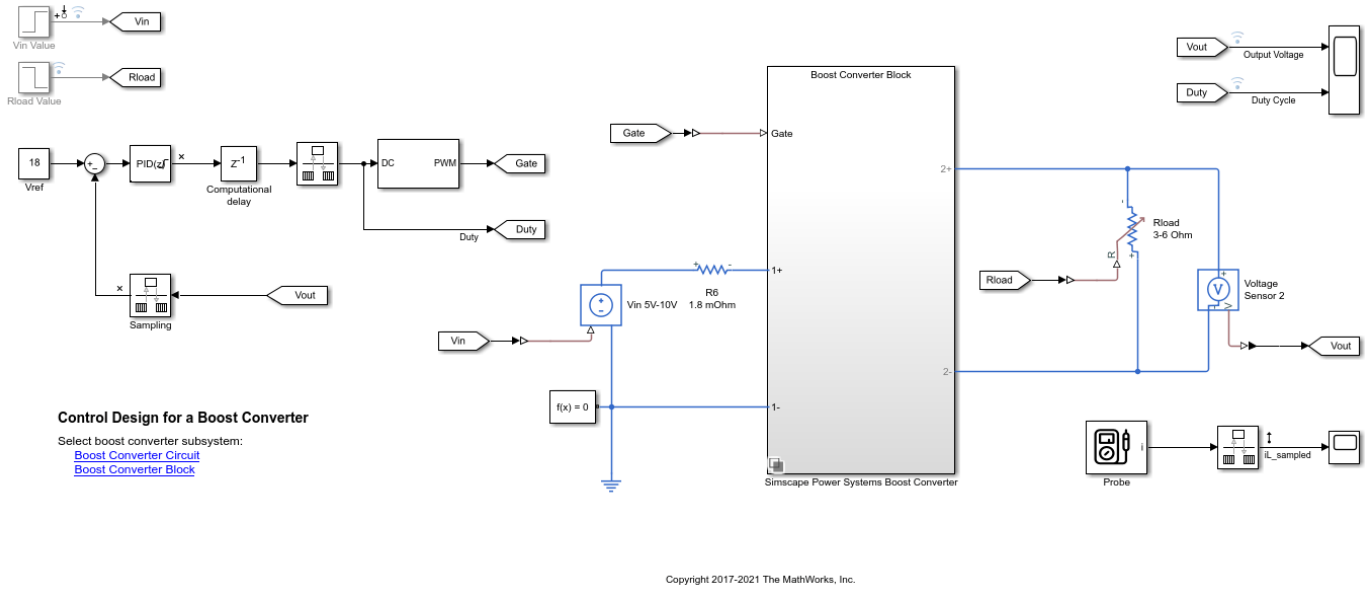
Typically, measuring input admittance of power electronics systems requires extensive modifications to these models. However, Simulink Control Design™ software provides tools to conduct frequency response estimation to measure this important property of a power electronics circuit. In this example, you also compare the estimated frequency response result to the analytical transfer function model constructed using component parameters.

Boost Converter Model

This example uses a boost converter model based on `scdboostconverter`. The model is modified with a Probe block and linear analysis points to measure the input current.

Open the model.

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



To use the Boost Converter block version of the subsystem, in the model, click **Boost Converter Block** or use the following command.

```
set_param([bdroot '/Simscape Power Systems Boost Converter'],...
    'OverrideUsingVariant','block_boost_converter')
```

Find Model Operating Point

To estimate the frequency response for the boost converter, you must first determine the steady-state operating point at which you want the converter to operate. For more information on finding

operating points, see “Find Steady-State Operating Points for Simscape Models” on page 1-104. For this example, use an operating point estimated from a simulation snapshot at 0.045 seconds.

```
opini = findop(mdl,0.045);
```

Initialize the model with the computed operating point.

```
set_param(mdl,'LoadInitialState','on','InitialState','getstatestruct(opini)')
```

Create Pseudorandom Binary and Sinestream Input Signals

A pseudorandom binary signal (PRBS) is a periodic, deterministic signal with white-noise-like properties that shifts between two values. A PRBS is an inherently periodic signal with a maximum period length of $2^n - 1$, where n is the PRBS order. For more information, see “PRBS Input Signals” on page 5-36.

Create a PRBS with the following configuration:

- To use a nonperiodic PRBS, set the number of periods to 1.
- Use a PRBS order of 14, producing a signal of length 16383. To obtain an accurate frequency response estimation, the length of the PRBS must be sufficiently large.
- Set the injection frequency of the PRBS to 200 kHz to match the sample time in the model. That is, specify a sample time of `5e-6` seconds.
- To ensure that the system is properly excited, set the perturbation amplitude to 0.5.

```
in_PRBS = frest.PRBS('Order',14,'NumPeriods',1,'Amplitude',0.5,'Ts',5e-6);
```

Create a sinestream input signal with 30 frequencies between 200 rad/s and 600000 rad/s. Set the amplitude of the input signal to 0.25 to make it consistent with the PRBS input signal.

```
in_Sinestream = frest.createFixedTsSinestream(5e-06,{200,6e5});
in_Sinestream.Amplitude = 0.25;
```

Using different input signals leads to a different simulation time for the frequency response estimation process. To achieve similar results, the PRBS input signal typically takes a much shorter simulation time than the sinestream input signal. Here, `in_PRBS` takes about 15% of the simulation time as `in_Sinestream`.

Following is the simulation time for frequency response estimation by `in_PRBS`.

```
in_PRBS.getSimulationTime


ans =

    0.0819
```

Following is the simulation time for frequency response estimation by `in_Sinestream`.

```
in_Sinestream.getSimulationTime


ans =

    0.5207
```

Collect Frequency Response Data

To collect frequency response data, you can estimate the plant frequency response at the command line. To do so, first get the input and output linear analysis points from the model.

```
io = getlinio(mdl);
```

Specify the operating point using the model initial conditions.

```
op = operpoint(mdl);
```

Find all source blocks in the signal paths of the linearization outputs that generate time-varying signals. Such time-varying signals can interfere with the signal at the linearization output points and produce inaccurate estimation results.

```
srcblks = frest.findSources(mdl,io);
```

To disable the time-varying source blocks, create an `frestimateOptions` option set and specify the `BlocksToHoldConstant` option.

```
opts = frestimateOptions;
opts.BlocksToHoldConstant = srcblks;
```

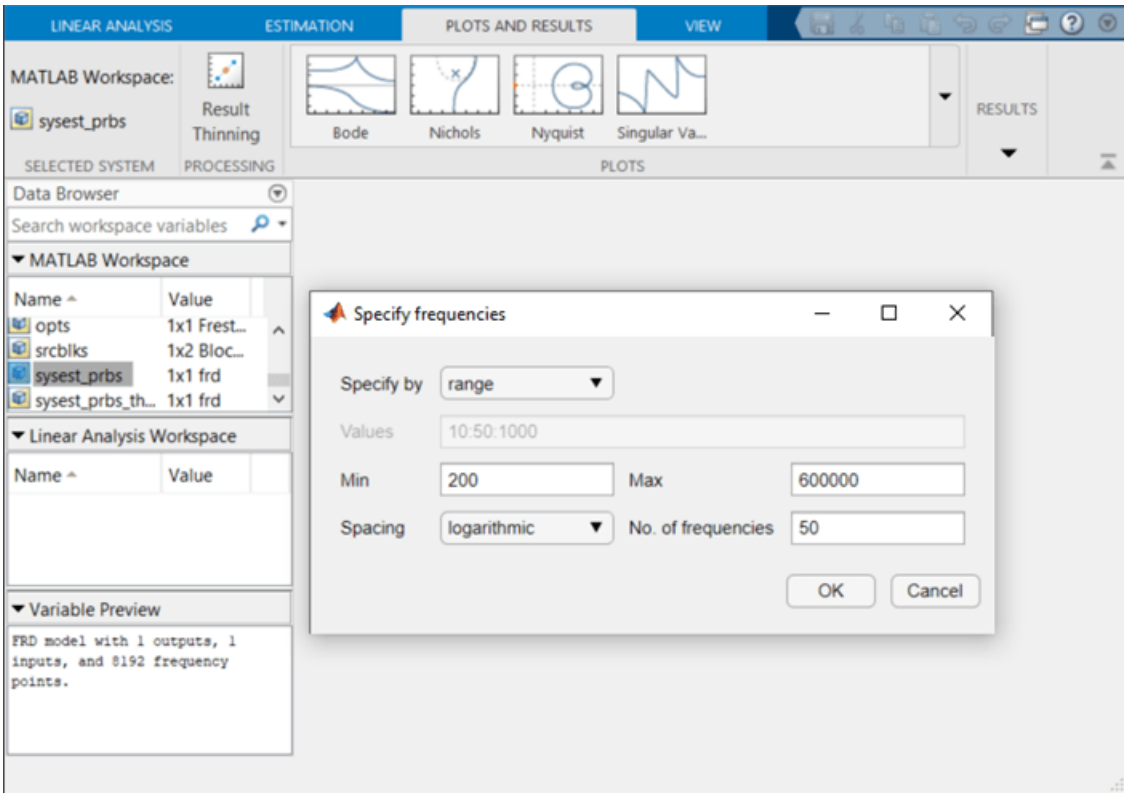
Estimate the frequency response using the PRBS input signal.

```
sysest_prbs = frestimate(mdl,io,op,in_PRBS,opts);
```

Frequency response estimation with a PRBS input signal produces results with a large number of frequency points. Simulink Control Design software lets you estimate plant frequency response using the **Model Linearizer** app. You can also use **Model Linearizer** to further improve a frequency response estimated at the command line.

You can use **Result Thinning** in **Model Linearizer** to extract an interpolated result from an estimated frequency response model across a specified frequency range and number of frequency points. For more information, see “Result Thinning” on page 5-23.

Apply thinning to `sysest_prbs` over frequencies between 200 rad/s and 600,000 rad/s with 50 logarithmically spaced frequency points.



Click **OK**. The thinned model, `sysest_prbs_thinned`, appears in the MATLAB® Workspace.

For comparison purposes, estimate the frequency response using the defined sinestream input signal.

```
sysest_sine = frestimate(mdl,io,op,in_Sinestream,opts);
```

Analytical Transfer Function of Boost Converter Model

Compare the estimation results to the analytical transfer function based on component parameters. Use the following parameters from the model.

```
L = 20e-6;
C = 1480e-6;
R = 6;
rC=8e-3;
rL=1.8e-3;
```

To compute the analytical transfer function, you require the actual duty cycle value. For this boost converter, use the logged duty cycle at the operating point.

```
D = 0.734785;
d = 1-0.734785;
```

Define the analytical transfer function, which is based on [1], using the component parameters in the boost converter model.

```
Yin = tf([C*(R+rC) 1], [L*C*(R+rC) L+C*rL*(R+rC)+C*R*rC-R*D*C*rC R+rL-R*D-R^2*D*d/(R+rC)]);
```

For a more accurate description, account for the delay due to PWM signal in the transfer function `Yin`.

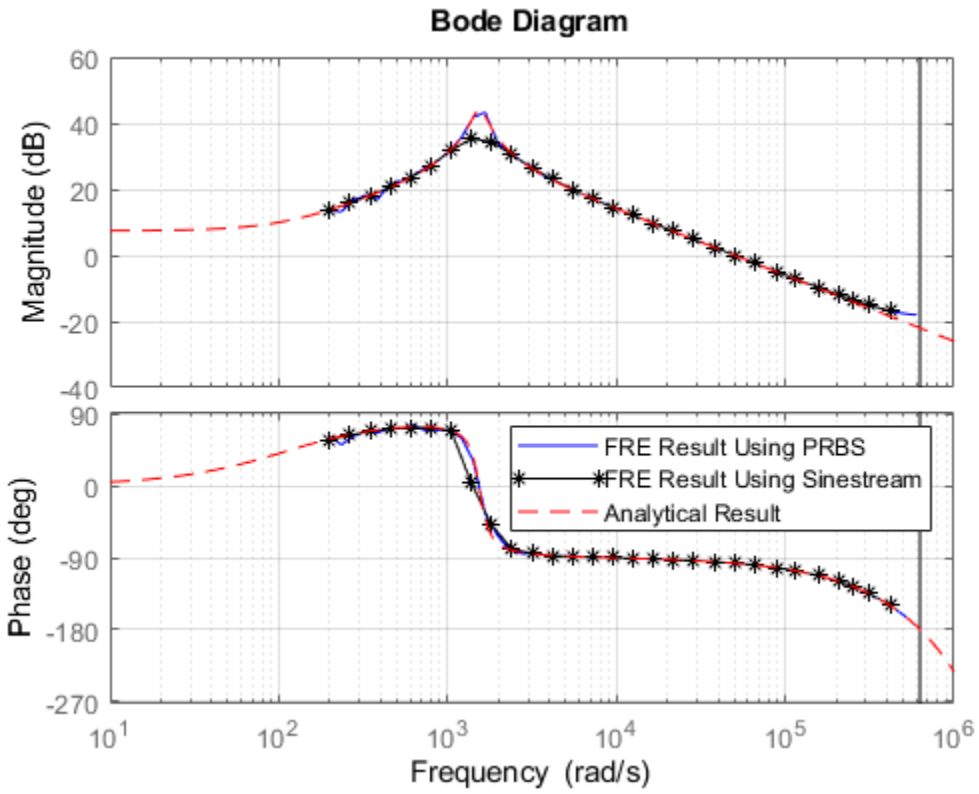
```
N = 0.5;
Ts = 5e-6;
iodelay = N*Ts; % 0.5 PWM
Yin.IODelay = iodelay;
```

Compare Frequency Response Data to Analytical Transfer Function

Compare FRE results using PRBS and sinestream input signals to the analytical transfer function over the frequency range of interest.

Load the thinned PRBS result from a previously saved session and plot the results.

```
load sysest_prbs_thinned_result
figure
bode(sysest_prbs_thinned,'b-')
hold on
bode(sysest_sine,'k*-')
bode(Yin,'r--',{10,1e6})
legend('FRE Result Using PRBS','FRE Result Using Sinestream','Analytical Result',...
    'Location','northeast')
grid on
```



Examine the Bode plot. The FRE result obtained using PRBS input signal captures interesting frequency-domain properties. For example, the frequency response estimation result with PRBS input signal, around the magnitude peak, matches the analytical transfer function much better than the frequency response estimation result with sinestream input signal.

Close the model.

```
close_system(mdl,0)
```

References

[1] Ahmadi, Reza, and Mehdi Ferdowsi. "Modeling Closed-Loop Input and Output Impedances of DC-DC Power Converters Operating inside Dc Distribution Systems." *2014 IEEE Applied Power Electronics Conference and Exposition - APEC 2014*, 2014 1131-38. https://doi.org/10.1109/APEC.2014.6803449.

See Also

`frest.PRBS` | `frest.createFixedTsSinestream` | `frestimate` | `frestimateOptions`

Related Examples

- “Frequency Response Estimation Basics” on page 5-2
- “Estimate Frequency Response at the Command Line” on page 5-14
- “Estimation Input Signals” on page 5-24
- “Frequency Response Estimation for Power Electronics Model Using Pseudorandom Binary Signal” on page 5-94
- Frequency Response Estimation in Model Linearizer Using Pseudorandom Binary Sequence on page 5-101

Online Frequency Response Estimation

- “Online Frequency Response Estimation Basics” on page 6-2
- “Online Estimation Using Plant Modeled in Simulink” on page 6-5
- “Deploy Frequency Response Estimation Algorithm for Real-Time Use” on page 6-9
- “Online Frequency Response Estimation During Simulation” on page 6-15
- “Collect Frequency Response Experiment Data for Offline Estimation” on page 6-18
- “Online Estimation of Frequency Responses of a Nonlinear Plant” on page 6-22

Online Frequency Response Estimation Basics

The Frequency Response Estimator block in Simulink Control Design lets you measure the frequency response of your system in operation. The block performs an experiment that injects signals into the plant and measures the plant output. If you have a code-generation product such as Simulink Coder™, you can generate code that implements the estimation algorithm on hardware. Deploying the algorithm to hardware lets you measure the frequency response of a physical plant in real time.

Embedded frequency-response estimation is a useful option when you have a physical plant and a test bed or control environment to operate in. In this case, you can deploy the Frequency Response Estimator block to your hardware. You trigger the tuning process via an input to the block, so you can tune your controller at any time. For details, see “Deploy Frequency Response Estimation Algorithm for Real-Time Use” on page 6-9.

If you have a plant model in Simulink, you can use the Frequency Response Estimator block to preview plant response and adjust the experiment settings before performing estimation in real time. Doing so helps ensure that real-time estimation does not drive your system out of the desirable operating range. For details, see “Online Estimation Using Plant Modeled in Simulink” on page 6-5. You can also use the block to obtain the frequency response of a plant that cannot be linearized in Simulink, as an alternative to offline frequency response estimation with **Model Linearizer** or `frestimate`.

When Not to Use Online Frequency-Response Estimation

You can use online frequency response estimation with any stable SISO plant. For an unstable plant, online estimation works in a closed-loop configuration, provided that the closed loop is internally stable. A closed-loop system is internally stable if and only if the roots of the nominal closed-loop characteristic equation all lie in the open left half-plane. For a plant with transfer function $G = N_G/D_G$ and controller $C = N_C/D_C$, the characteristic equation is:

$$D_G D_C + N_G N_C = 0.$$

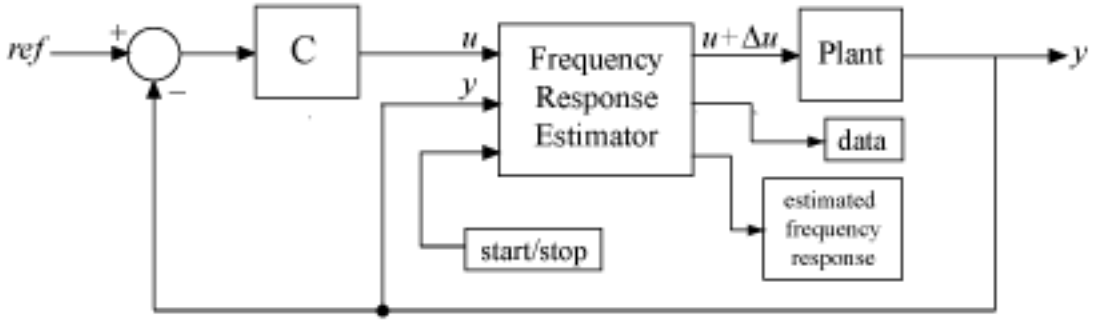
In practice, this condition means that no unstable poles in G are stabilized by pole-zero cancellation in GC. Do not use online estimation with an unstable plant that does not meet this condition.

Online frequency response estimation does not work well when there are large disturbances in the plant during the estimation experiment. Disturbances distort the plant response to the perturbation signals, yielding poor estimation results.

System Configurations for Online Frequency Response Estimation

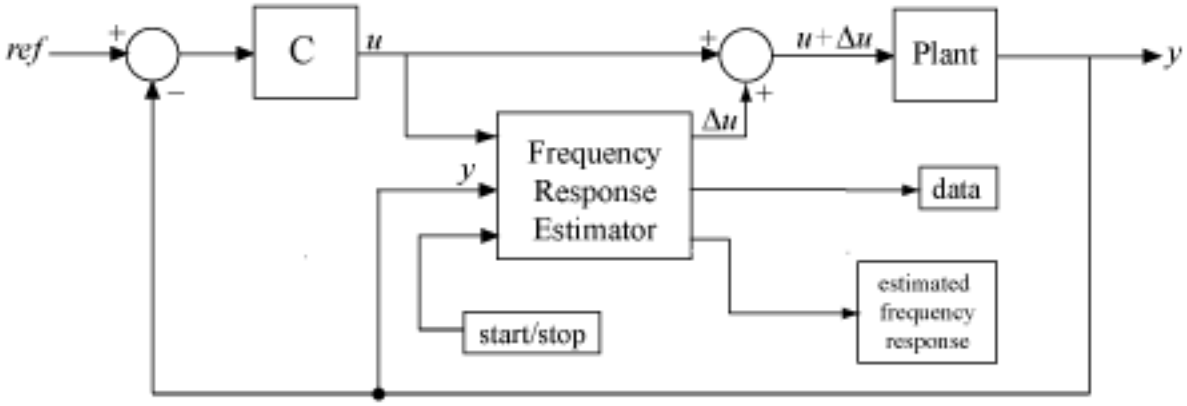
To use online frequency response estimation, you configure and deploy the Frequency Response Estimator block. The block performs an estimation experiment by injecting test signals in the plant and measuring system response.

The following schematic diagram illustrates a typical configuration for fitting the Frequency Response Estimator block into a closed-loop system.



In this configuration, the block receives a control signal u, adds a perturbation Δu to it, and injects the perturbed signal directly into the plant. It then measures the plant response y and uses the result to compute the estimated frequency response.

Alternatively, you can configure the block such that the output is the perturbation Δu only. You can then add this signal to the plant input yourself, as shown in the following schematic diagram.



As an alternative to the closed-loop configurations shown above, you can use the block in an open-loop plant, typically using a constant input signal u to drive the plant to the desired operating point. It is a good practice to use the closed-loop configuration, particularly for real-time estimation. In a closed-loop configuration, the controller works to suppress the injected disturbance and maintain safe plant operation.

Estimation Workflow

The general workflow for online frequency response estimation is as follows.

1 Incorporate the Frequency Response Estimator block into your system in one of the configurations described above.

2. Configure the start/stop signal that controls when the estimation experiment begins and ends. You can use this signal to initiate the experiment at any time.
3. Configure experiment parameters such as the frequencies at which to estimate the response and the amplitudes of the injected perturbations.
4. Start the experiment using the start/stop signal, and allow it to run long enough for the estimation process, based on the experiment length recommended by the block.
5. Stop the experiment and examine the estimated frequency response.

For detailed information on performing each of these steps, see:

- “Online Estimation Using Plant Modeled in Simulink” on page 6-5
- “Deploy Frequency Response Estimation Algorithm for Real-Time Use” on page 6-9

See Also

Frequency Response Estimator

More About

- “Frequency Response Estimation Basics” on page 5-2
- “Online Estimation Using Plant Modeled in Simulink” on page 6-5
- “Deploy Frequency Response Estimation Algorithm for Real-Time Use” on page 6-9

Online Estimation Using Plant Modeled in Simulink

When you have a plant modeled in Simulink, you can perform frequency response estimation using **Model Linearizer** or the `frestimate` command without changing the model. The Frequency Response Estimator block is an alternative that lets you incorporate the estimation experiment directly into your model and perform estimation while the model is running. This approach is especially useful when you plan to deploy the block for online estimation of a physical plant. Testing the estimation algorithm and experiment parameters against a Simulink model of the plant before deployment can help ensure that online estimation is safe for your plant.

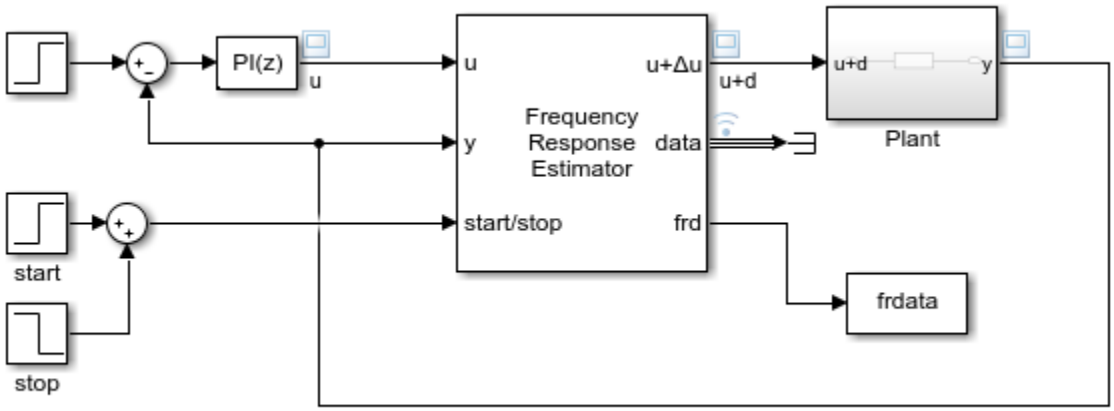
Workflow for Online Estimation in Simulink

The following steps provide a general overview of the workflow for online frequency response estimation of a plant modeled in Simulink.

1. Incorporate the Frequency Response Estimator block on page 6-5 into your model.
2. Configure the start/stop signal on page 6-6 that controls when the estimation experiment begins and ends.
3. Configure experiment parameters on page 6-7 such as the frequencies at which you want to perform estimation.
4. Run the model on page 6-7. Use the start/stop signal to initiate the estimation experiment. When you start the experiment, the block injects test signals and measures the response of the plant. When you end the experiment, you can examine the estimated frequency response.

Step 1. Incorporate Frequency Response Estimator into Model

The following illustration shows one way to incorporate a Frequency Response Estimator block into a closed-loop control system. In this configuration, you insert the block between the controller and the plant.



The control signal feeds into the **u** port of the Frequency Response Estimator block. The **u + Δu** port feeds into the plant input. Before you begin the estimation process, the block feeds the control signal directly from **u** to **u + Δu** without adding any perturbation. In that state, the block has no effect on

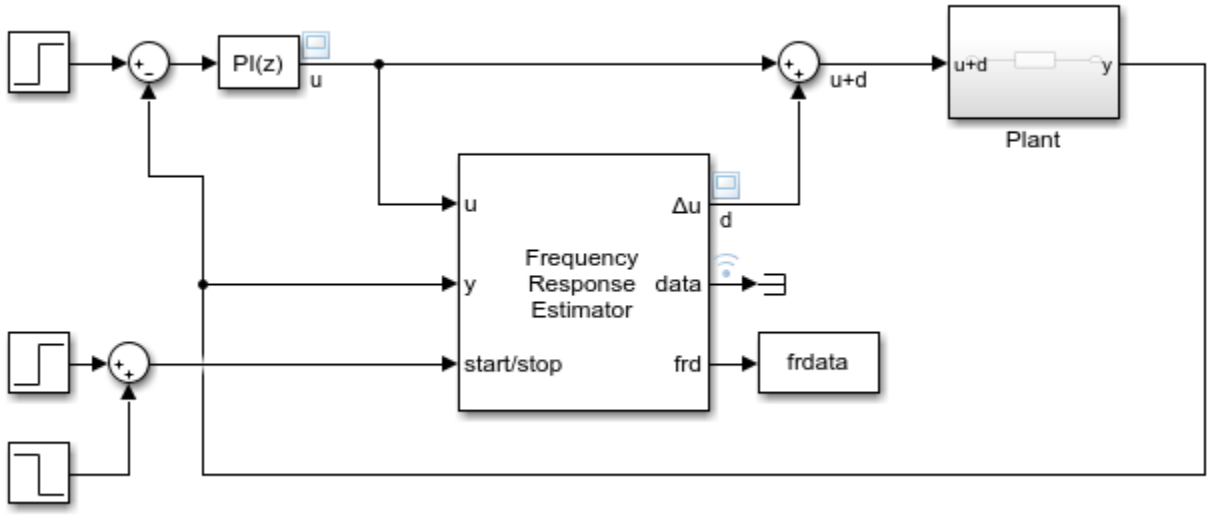
the system behavior. (You can perform frequency response estimation in an open-loop configuration by connecting **u** to a constant source that drives your plant to the desired operating point for estimation. However, it is a good practice to use the closed-loop configuration, particularly for real-time estimation. In a closed-loop configuration, the controller works to suppress the injected disturbance and maintain safe plant operation.)

The **start/stop** signal controls when the estimation process begins and ends (see “Step 2. Configure Start/Stop Signal” on page 6-6). Start the experiment when the plant is in steady state at the desired operating point. When the **start/stop** signal is positive, the block injects test signals at **u + Δu** and measures the response at **y**. The block calculates the estimated frequency response and returns it at the **frd** port.

For an example, see “Online Frequency Response Estimation During Simulation” on page 6-15.

Apply Perturbation Signal Only

The default configuration requires inserting the block between the controller and the plant. If you want to add the perturbation signal to the control signal yourself, in the Frequency Response Estimator block parameters, set **Output Signal Configuration** to **perturbation only**. In this configuration, the block output contains the perturbation signal only, at the port **Δu**. You inject this perturbation signal into the plant using, for example, a sum block, as in the following diagram.



In this configuration, because the Frequency Response Estimator is not part of the closed loop, you can optionally comment it out without disrupting the loop configuration.

Step 2. Configure Start/Stop Signal

To start and stop the frequency response estimation experiment, use a signal at the **start/stop** port. When the experiment is not running, the block generates no perturbation signal. In this state, the block has no impact on plant behavior. The frequency-response estimation experiment begins and ends when the block receives a rising or falling signal at the **start/stop** port, respectively. In the systems illustrated in “Step 1. Incorporate Frequency Response Estimator into Model” on page 6-5,

staggered step signals start and stop the experiment. You can configure any other logic appropriate for your application to control the start and stop times of the experiment. For example, you can use a Signal Editor block to configure a start/stop signal to carry out multiple experiments in a single simulation run.

The block provides a recommended experiment length in the **Experiment Length** section of the block parameters. Typically, you configure the start/stop signal such that there is at least that much time between the rising and falling signals. You must also make sure that the simulation does not stop before the experiment stops. For more information about how the block determines the recommended experiment length, see the Frequency Response Estimator block reference page.

Step 3. Set Experiment Parameters

The frequency-response estimation experiment injects sinusoidal signals at the frequencies you specify with the **Frequencies** parameter (or at the `w` port) of the Frequency Response Estimation block. Specify the perturbation amplitudes using the **Amplitudes** parameter (or at the `amp` port).

The block can apply the perturbation at each frequency separately (sinestream mode) or simultaneously (superposition mode). To specify which mode to use, set the **Experiment mode** parameter.

- **Sinestream** mode — Applies the perturbation one frequency at a time. Sinestream mode can be more accurate and can accommodate a wider range of frequencies than superposition mode.
- **Superposition** — Applies the perturbation as a superposition signal containing all frequencies at once. The estimation experiment is generally faster in superposition mode.

You can also specify parameters that tell the block how long to let the system settle when the perturbation is applied, and how long to measure the response for the estimation. For further details about the two signal types and their relative advantages, see the **Experiment mode** parameter description on the Frequency Response Estimator block reference page.

Step 4. Run Model and Examine Estimated Frequency Response

After you have configured all the parameters for the estimation experiment, run the model. Allow the model to run long enough to complete the estimation experiment, based on the recommended experiment length provided by the block. If you select **Display Bode plot**, the block generates a Bode plot to visualize the estimated frequency response during the experiment.

During the experiment, the block updates the estimated frequency response at the **frd** port. The signal at this port is a vector with one value for each frequency specified by **Frequencies**. You can write this signal to the MATLAB workspace using a To Workspace block, or use Simulink data logging to write the data to the workspace as a `Simulink.SimulationData.Dataset` object. The logged values show the convergence of the frequency responses during the experiment. The most meaningful value is the value when the experiment stops. For that reason, you can discard all values except the last one.

For an example of a model configured to perform online frequency response estimation, see “Online Frequency Response Estimation During Simulation” on page 6-15.

See Also

Frequency Response Estimator

More About

- “Deploy Frequency Response Estimation Algorithm for Real-Time Use” on page 6-9
- “Online Frequency Response Estimation During Simulation” on page 6-15
- “Online Estimation of Frequency Responses of a Nonlinear Plant” on page 6-22
- “Online Frequency Response Estimation Basics” on page 6-2

Deploy Frequency Response Estimation Algorithm for Real-Time Use

You can use the online frequency-response estimation algorithm in a standalone application for real-time estimation of a physical plant. To do so, you must deploy the Frequency Response Estimator block into your own system by creating a Simulink model for deployment. You can configure this model with the experiment parameters. Or, you can configure it to supply such parameters externally from elsewhere in your system. Once deployed to your own system, the estimator model injects signals into your plant and receives the plant response, without using Simulink to control the experiment. Deploying the estimation algorithm requires a code-generation product such as Simulink Coder.

Workflow

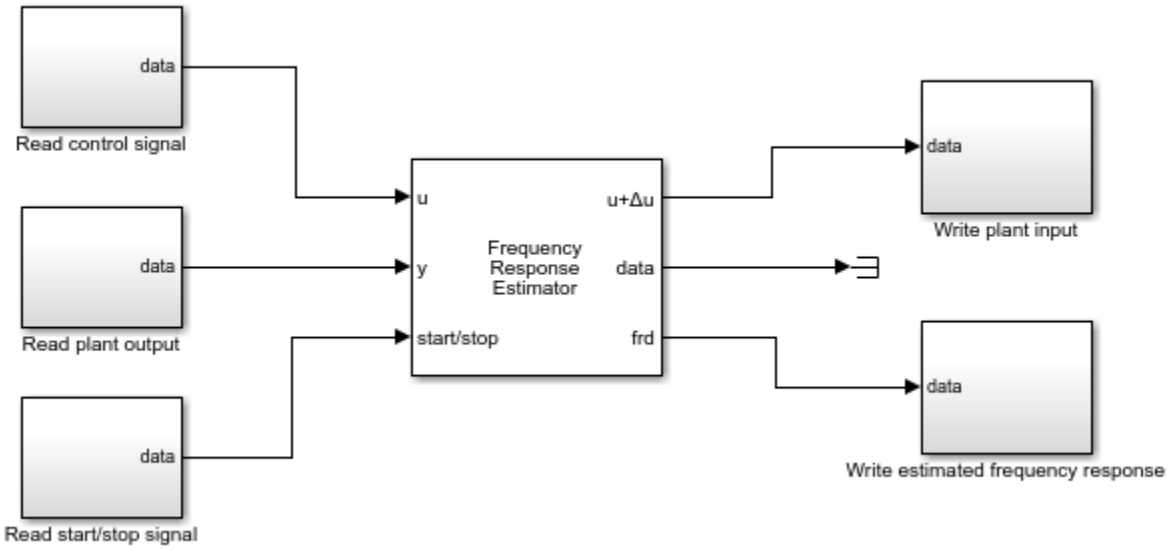
In overview, the workflow for deploying the Frequency Response Estimator for real-time tuning is:

1. Create a Simulink model on page 6-9 for deploying the block to your system.
2. Configure the start/stop signal on page 6-12 that controls when the estimation experiment begins and ends.
3. Configure experiment parameters on page 6-12 such as the frequencies at which you want to perform estimation.
4. Deploy the model to your system, and run the estimation experiment on page 6-13 against your physical plant. When you end the experiment, you can examine the estimated frequency response.

In practice, for real-time estimation, you might want to specify some parameters at run time, such as the estimation frequencies or perturbation amplitudes. For information about specifying parameters in your deployed application, see “Access Experiment Parameters After Deployment” on page 6-13.

Step 1. Create Deployable Simulink Model with Frequency Response Estimator Block

Using a Frequency Response Estimator block for real-time estimation requires creating a Simulink model for deployment. In the most basic form, a model for deploying real-time estimation resembles the following illustration.



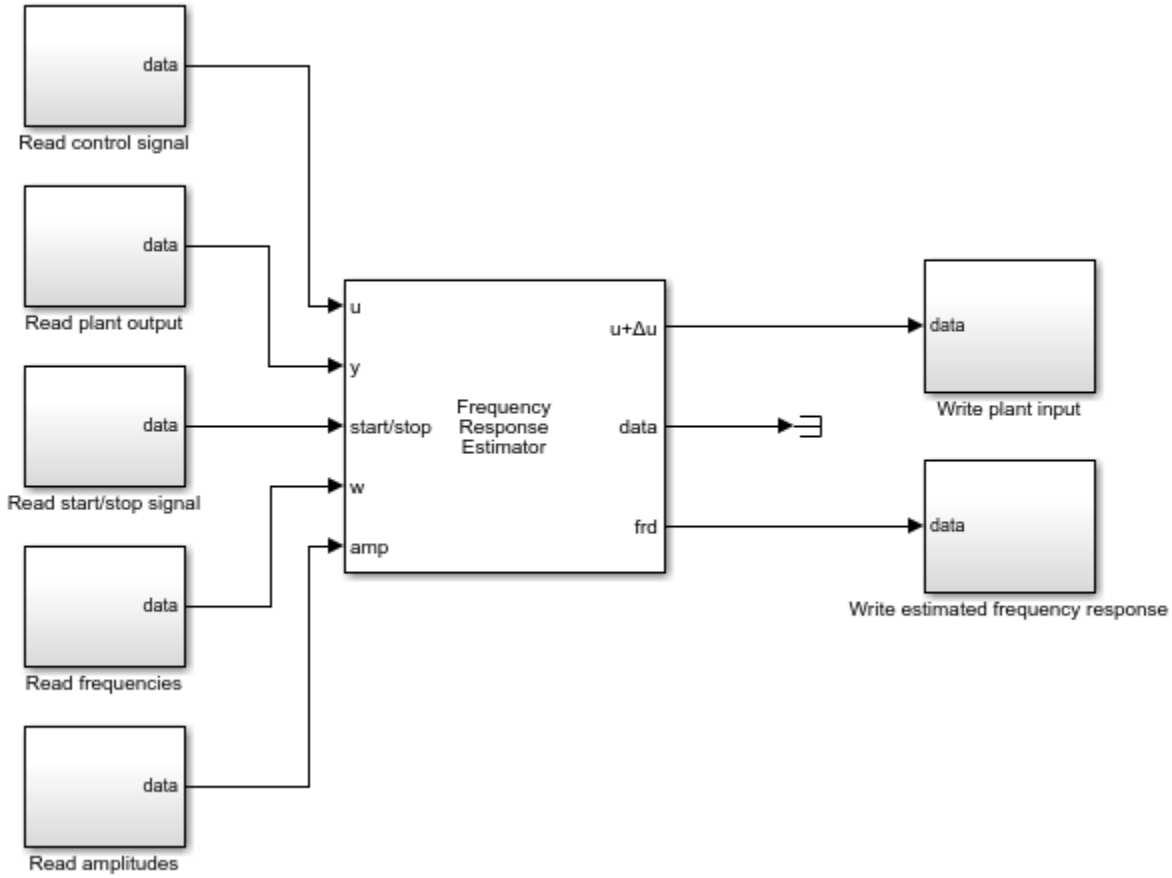
Here, the blocks connected to the inputs and outputs of the Frequency Response Estimator block represent hardware interfaces that read or write real-time data for your system. For example, the `Read control signal` block can be an interface for receiving serial data, a UDP Receive block for receiving UDP packets, or an interface for receiving other signals via wireless network. Similarly the blocks for writing data, such as `Write plant input`, can be interfaces for serial, UDP, or other interfaces for writing data to hardware.

The default ports of the Frequency Response Estimator block are:

- `u` — Receives the control signal.
- `y` — Receives the plant output.
- `start/stop` — Receives the signal that begins and ends the estimation experiment.
- `u + Δu` — Outputs the signal to feed to the plant input. When the experiment is not running, `u + Δu` outputs the control signal as received at `u`. When the experiment is running, the block adds the perturbation `Δu` to this signal.
- `data` — Outputs the simulation data collected during the estimation experiment. This data includes the perturbation applied to the plant input and the response received at `y`.
- `frd` — Outputs the estimated frequency responses.

For more details about all ports, see the Frequency Response Estimator block reference page.

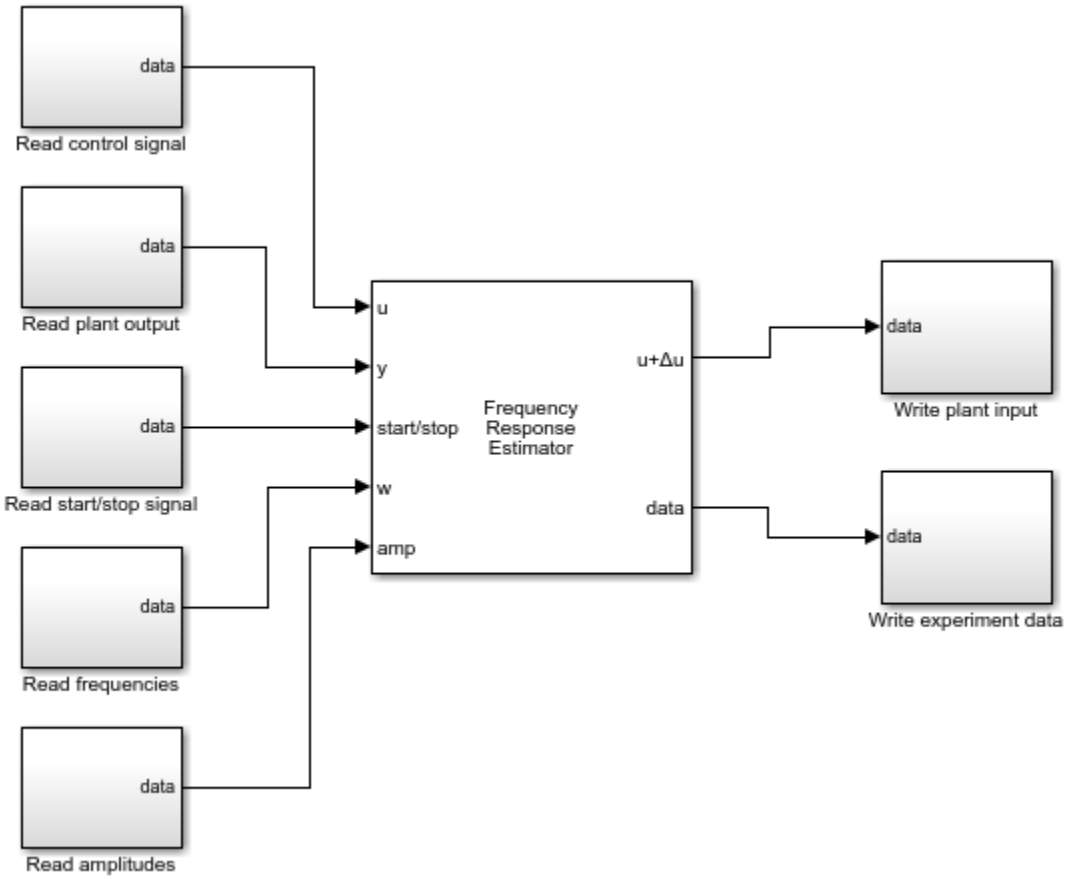
In the illustrated configuration, the frequencies at which to perform estimation and the amplitudes of the perturbation to apply at each frequency are hard-wired into the block. If you want to set these values after deployment, set the block parameter **Excitation Signal Source** to **External ports**. Doing so adds the `w` and `amp` ports to the block, as shown in the following illustration.



In this configuration, the deployed module can read frequencies and perturbation amplitudes for the estimation experiment at run time.

Store Data for Offline Estimation

The previously illustrated configurations discard the `data` output port, which provides the input and response signals collected during the estimation experiment. If you want to use this experiment data, you can store the output from this port. For instance, to conserve resources in a deployed environment, you can configure the block to collect the experiment data without performing the estimation. You can then perform the estimation in MATLAB using `frestimate`. A model configured this way for deployment resembles the following illustration.



Step 2. Configure Start/Stop Signal

To start and stop the frequency response estimation experiment, use a signal at the **start/stop** port. When the experiment is not running, the block generates no perturbation signal. In this state, the block has no impact on plant behavior. The frequency-response estimation experiment begins and ends when the block receives a rising or falling signal at the **start/stop** port, respectively. You can configure any logic appropriate for your application to control the start and stop times of the experiment.

The block provides a recommended experiment length in the **Experiment Length** section of the block parameters. Typically, you configure the start/stop signal such that there is at least that much time between the rising and falling signals. In a deployed environment when you are setting estimation parameters at run time, you must be aware of how experiment parameters such as estimation frequencies affect the required experiment length. For more information about determining the appropriate length, see the Frequency Response Estimator block reference page.

Step 3. Set Experiment Parameters

The frequency-response estimation experiment injects sinusoidal signals at the frequencies you specify with the **Frequencies** parameter (or at the `w` port) of the Frequency Response Estimation block. Specify the perturbation amplitudes using the **Amplitudes** parameter (or at the `amp` port).

The block can apply the perturbation at each frequency separately (sinestream mode) or simultaneously (superposition mode). To specify which mode to use, set the **Experiment mode** parameter.

- **Sinestream** mode — Applies the perturbation one frequency at a time. Sinestream mode can be more accurate and can accommodate a wider range of frequencies than superposition mode.
- **Superposition** — Applies the perturbation as a superposition signal containing all frequencies at once. The estimation experiment is generally faster in superposition mode.

You can also specify parameters that tell the block how long to let the system settle when the perturbation is applied, and how long to measure the response for the estimation. For further details about the two signal types and their relative advantages, see the **Experiment mode** parameter description on the Frequency Response Estimator block reference page.

Step 4. Run Experiment

After you deploy the estimation module to your system, use a rising `start/stop` signal to begin the estimation experiment. The deployed module injects the test signals into your physical plant in real time. After an appropriate time, your falling `start/stop` signal ends the experiment. (For more information about determining the appropriate length, see the Frequency Response Estimator block reference page.)

When the experiment is complete, you can obtain the estimated frequency response at the `frd` port.

If your deployed environment is short of resources for the online estimation computation, you can configure the block to collect experiment data only, and perform the estimation offline later. For an example, see “Collect Frequency Response Experiment Data for Offline Estimation” on page 6-18.

Access Experiment Parameters After Deployment

Some of the parameters that you set to configure the estimation experiment are tunable, such that you can access them in the generated code. Most parameters, however, are not tunable. For those parameters, you must configure them in the block before deployment, or use an external block port for the parameters for which one is available.

Tunable Parameters

The following parameters of the Frequency Response Estimator block are tunable after deployment. For more information about all these parameters, see the block reference pages.

Parameter	Description
Number of estimation periods	Number of periods after settling to use for estimation (sinestream mode)
Number of settling periods	Number of periods to wait for settling of transients (sinestream mode)
Number of periods of the lowest frequency used for estimation	Duration of data-collection window (superposition mode)

Non-Tunable Parameters

The remaining parameters of the Frequency Response Estimator are not tunable after deployment. For the **Frequencies** and **Amplitudes** parameters, you can enable external ports that allow you to

supply experiment frequencies and perturbation amplitudes after deployment. To enable the **w** and **amp** block inputs, in the **Excitation Signal Source** parameter, select **External ports**.

Modify Experiment Sample Time After Deployment

The **Sample time (Ts)** parameter is not tunable. As a consequence, you cannot access it directly in generated code when you deploy the block. To change the controller sample time in the deployed block at run time:

1. Set **Controller sample time (sec)** to –1.
2. Put the block in a Triggered Subsystem.
3. Trigger the subsystem at the desired sample time.

If you use this approach, you must make sure at run time that your sample time is fast enough to keep your estimation frequencies below the Nyquist frequency.

See Also

Frequency Response Estimator

More About

- “Online Frequency Response Estimation Basics” on page 6-2
- “Online Estimation Using Plant Modeled in Simulink” on page 6-5
- “Collect Frequency Response Experiment Data for Offline Estimation” on page 6-18

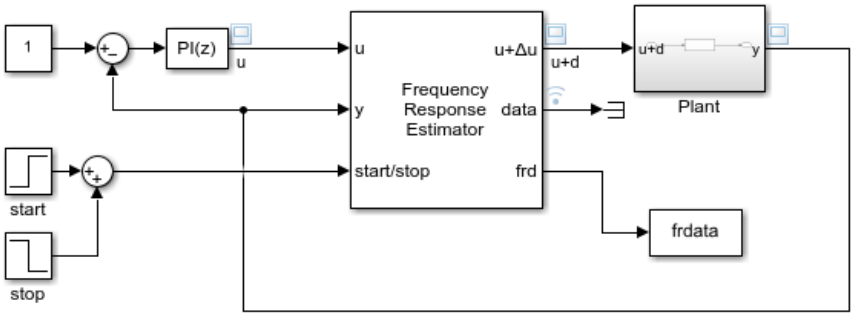
Online Frequency Response Estimation During Simulation

This example shows how to use the Frequency Response Estimator block to perform online frequency response estimation during simulation of the model in Simulink®. This approach can be useful when you plan to deploy the block for online estimation of a physical plant. Testing the estimation algorithm and experiment parameters against a Simulink model of the plant before deployment can help ensure that online estimation is safe for your plant.

Control System Model

This example uses a model that already contains the Frequency Response Estimator block configured for estimation. Open the model.

```
mdl = "OnlineFreqRespEstimEx.slx";
open_system(mdl)
```



Copyright 2018 The MathWorks, Inc.

Copyright 2018-2021 The MathWorks, Inc.

The model contains a plant in a closed-loop configuration with a PI controller. The Frequency Response Estimator block accepts the control signal as the input u. It feeds the control signal plus a perturbation into the plant input. You specify properties of the perturbation signal using parameters of the block.

Experiment Parameters

The Frequency Response Estimator block is configured to run the experiment in sinestream mode, which means that it injects a separate perturbation at each frequency. The block is also configured to use the same amplitude, 1, for each frequency in the perturbation signal.

The block is further configured to estimate frequency responses at the frequencies `w = logspace(0,2,20)`. To ensure that the experiment sampling rate is fast enough to accommodate the highest frequency, it is a good practice to set the sampling time to about 0.6 / `wmax` or faster, where `wmax` is the highest frequency in rad/s. For this example, the experiment sample time is 0.005 seconds, is fast enough for the `wmax` of 100 rad/s.

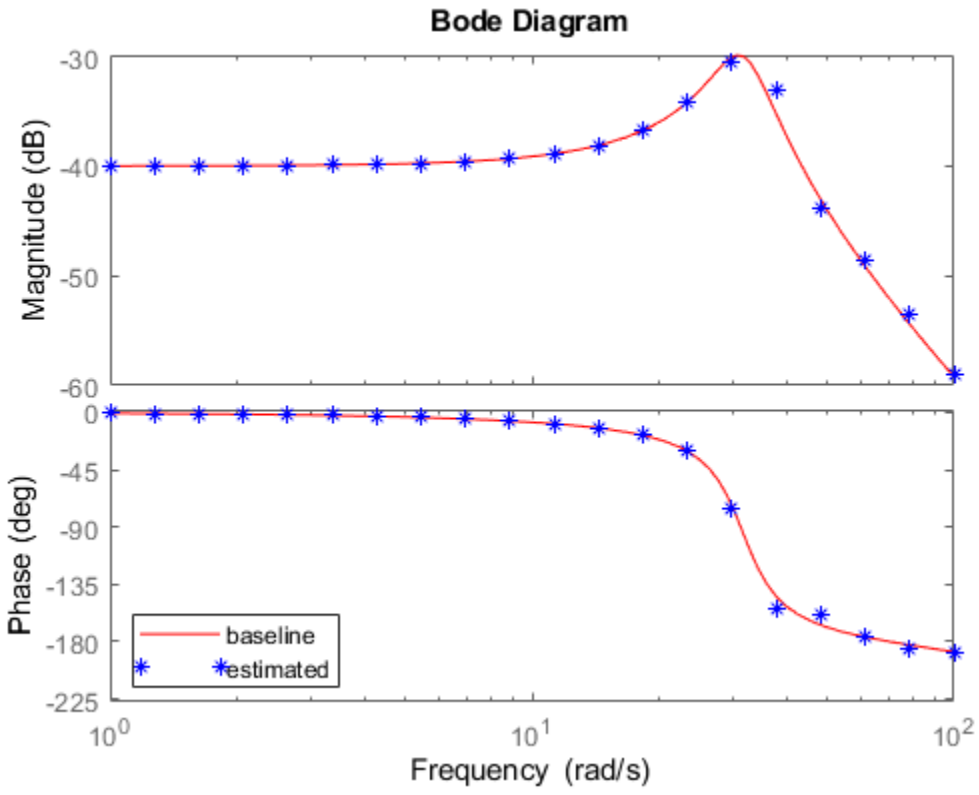
Start/Stop Signal

The step blocks connected to the `start/stop` input port turn the experiment on with a rising signal at $t = 5$, when the model is at steady state. The block provides a recommended experiment length of about 174 s. This value is based on the specified frequencies `w`, the number of settling periods to wait at each frequency, and the number of periods to use for estimation. To ensure that the experiment runs long enough for a good result, the `start/stop` signal stops the experiment at $t = 180$. (For details about the recommended experiment length, see Frequency Response Estimator.)

Estimation Results

Simulate the model. You can use the scope to visualize the control signal, the perturbation signal, and the plant output. Because the **Display Bode Plot** block parameter is selected, the block automatically generates a plot of the specified baseline model and updates it periodically with the estimated frequency response.

```
sim(mdl)
```



The signal at the `frd` port is a vector containing the current values of the estimated response at each frequency in `w`. The To Workspace block connected to that port writes the signal to the MATLAB®

workspace variable `frdata`. In the To Workspace block, the **Limit data points to last** parameter is set to 1, so that `frdata` contains only the final estimated responses at each frequency. Convert `frdata` to a `frd` model object.

```
sys_estim = frd(frdata,w);
size(sys_estim)
```

```
FRD model with 1 outputs, 1 inputs, and 20 frequency points.
```

You can now use `sys_estim` with Control System Toolbox™ analysis and control design commands that accept `frd` models as input, such as `bode` and `pidtune`. Alternatively, if you have System Identification Toolbox™ software, you can use the frequency response data to estimate a parametric model of your system.

Logged Experiment Data

The model is also configured to log the estimation data at the block output port data (see “Export Signal Data Using Signal Logging” for information about data logging). The data is stored in the MATLAB workspace as the `Simulink.SimulationData.Dataset` object `logsout`. For information about how to use this data, see “Collect Frequency Response Experiment Data for Offline Estimation” on page 6-18.

See Also

Frequency Response Estimator

More About

- “Online Frequency Response Estimation Basics” on page 6-2
- “Online Estimation Using Plant Modeled in Simulink” on page 6-5
- “Collect Frequency Response Experiment Data for Offline Estimation” on page 6-18

See Also

Collect Frequency Response Experiment Data for Offline Estimation

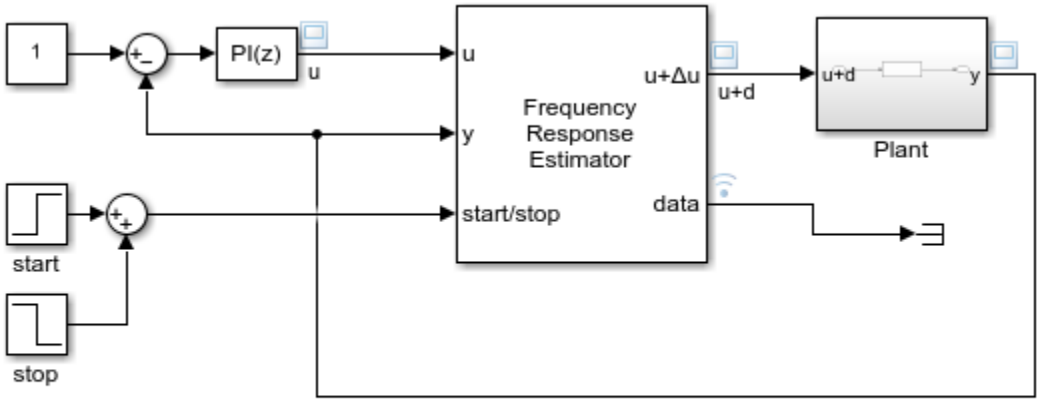
This example shows how to use the Frequency Response Estimator block to perform a frequency response estimation experiment and store the data for later estimation offline. In practice, you can use this approach to perform the experiment in real time against a physical plant, when your deployed environment is short of resources for the online estimation computation. In this example, for illustration purposes, you perform the experiment on a plant modeled in Simulink®.

Model and Experiment Parameters

This example uses a model that already contains a Frequency Response Estimator block configured to collect experiment data for offline estimation.

Open the model.

```
mdl = "CollectFreqRespEstimDataEx.slx";
open_system(mdl)
```



Copyright 2018-2021 The MathWorks, Inc.

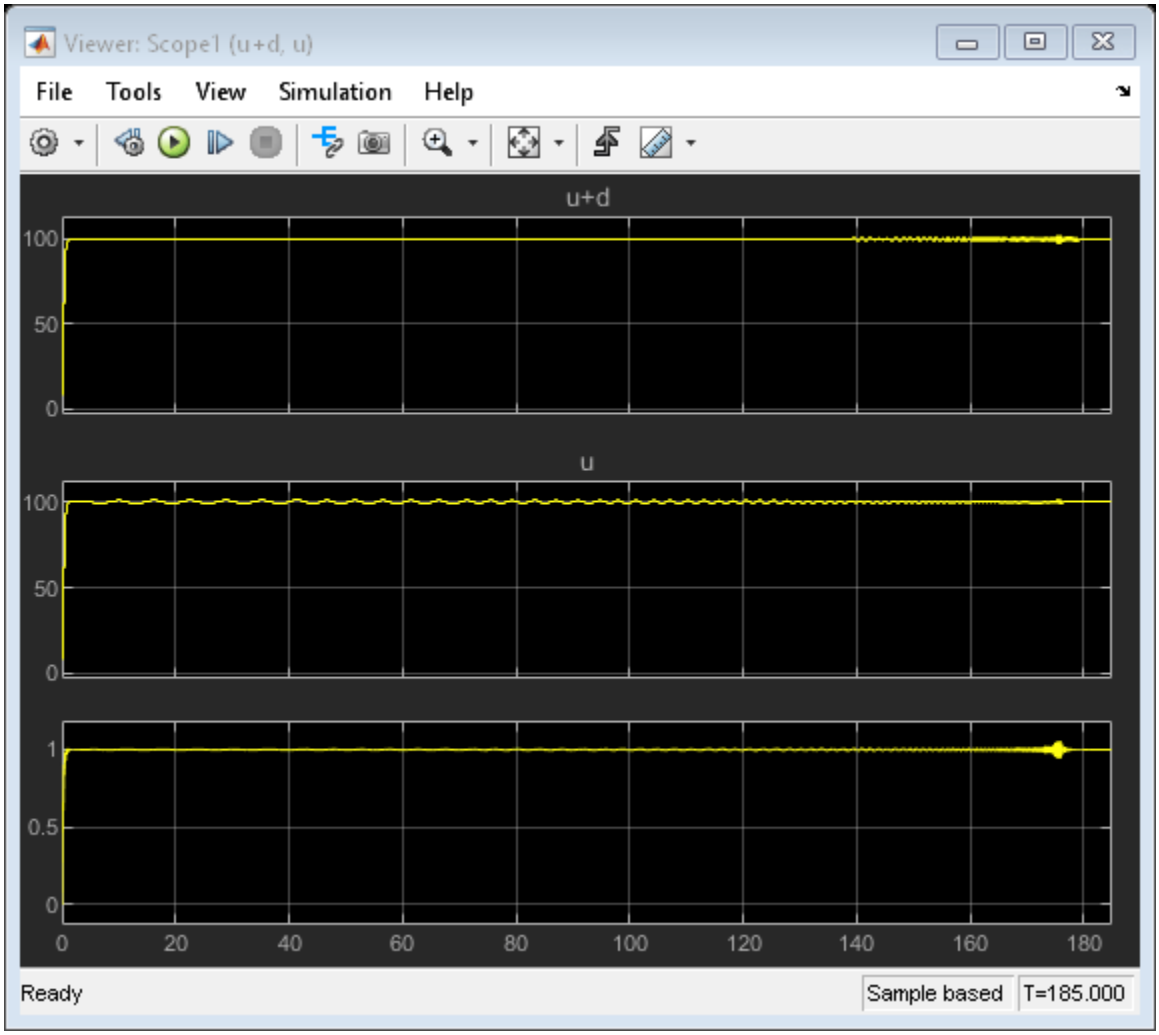
The model contains a plant in a closed-loop configuration with a PI controller. The Frequency Response Estimator block accepts the control signal as the input `u`. It feeds the control signal plus a perturbation into the plant input.

The Frequency Response Estimator block is configured to run the experiment in sinestream mode, with the same experiment parameters used in the example “Online Frequency Response Estimation During Simulation” on page 6-15. In this example, however, the **Estimation Mode** parameter is set to **Offline**. In this configuration, block injects the specified perturbation signals and collects the response data, but does not perform the estimation. The block is configured to use a sinestream signal at the frequencies `w = logspace(0,2,20)`.

Collect Experiment Data

Simulate the model. The block performs the experiment and collects the response data. The scope shows the applied sinestream signal and the system response.

```
sim(mdl)
open_system('CollectFreqRespEstimDataEx/Scope1')
```



The model is configured to log the estimation data at the block output port `data` (see “Export Signal Data Using Signal Logging” for information about data logging). The data is stored in the MATLAB workspace as the `Simulink.SimulationData.Dataset` object `logsout`. Because `data` is the only logged port, you can access the logged data in the first entry in `logsout`. The `Values` field of that entry is a structure containing four fields.

```
logdata = logsout{1}.Values
```

```
logdata =

  struct with fields:

          Ready: [1x1 timeseries]
    Perturbation: [1x1 timeseries]
      PlantInput: [1x1 timeseries]
     PlantOutput: [1x1 timeseries]
```

The `Ready` field is a timeseries containing a logical signal that indicates which time steps contain the data to used for the estimation. For a sinestream signal, this field indicates which perturbation periods for the estimation to discard (settling periods). `Perturbation` contains the sinestream perturbation applied to the plant. The `PlantInput` and `PlantOutput` timeseries contain the signals at the block inputs `u` and `y`, respectively.

Estimate Frequency Response

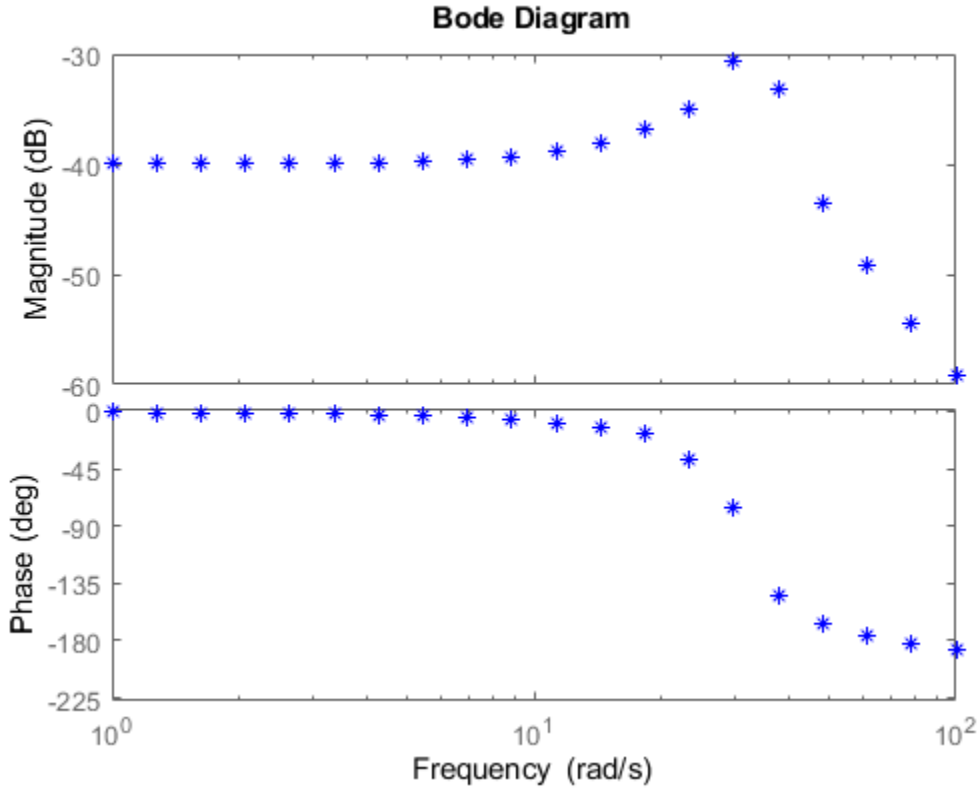
If you collect this data in a deployed environment with limited computational resources, you can use the data to perform frequency response estimation offline, using the `frestimate` command. Give `frestimate` the `logdata` structure and the same frequencies you used for the **Frequencies** parameter in the block. `frestimate` processes `logdata` to obtain a frequency response data (`frd`) model containing the estimated responses at those frequencies.

```
sys_estim = frestimate(logdata,w,'rad/s');
size(sys_estim)
```

```
FRD model with 1 outputs, 1 inputs, and 20 frequency points.
```

Examine the estimated frequency response.

```
figure
bode(sys_estim,'b*')
```



See Also

Frequency Response Estimator

More About

- “Online Frequency Response Estimation Basics” on page 6-2
- “Online Estimation Using Plant Modeled in Simulink” on page 6-5
- “Online Estimation of Frequency Responses of a Nonlinear Plant” on page 6-22
- “Deploy Frequency Response Estimation Algorithm for Real-Time Use” on page 6-9

Online Estimation of Frequency Responses of a Nonlinear Plant

This example shows how to use the Frequency Response Estimator block to perform online estimation of the plant frequency responses. For a nonlinear plant, estimation at different nominal operating points produces different frequency responses.

Real Time Frequency Response Estimation

Frequency response describes the steady state response of a system to a sinusoidal input signal. If the system is linear $G(s)$, the output signal is a sine wave of the same frequency with a different magnitude and a phase shift. A frequency response data (`frd`) model that stores frequency response information at multiple frequencies is useful for tasks such as analyzing plant dynamics, validating linearization results, designing a control system, and estimating a parametric model.

There are different ways to obtain an `frd` model in the Simulink environment. The most common approach is to linearize the Simulink model and calculate the frequency responses directly from the obtained state-space system. When the Simulink model cannot be linearized, you can use the `frestimate` command or use the **Model Linearizer** app to run simulation with some perturbation signals. Afterwards, the plant frequency responses are estimated offline based on the collected experiment data. This approach is called *offline estimation*.

This example shows an alternative *online estimation* approach using the Frequency Response Estimator block is to conduct an experiment and estimate the frequency response during simulation. Although this example uses a plant modeled in Simulink, if you do not have a plant model in Simulink, you can deploy the block on your target system and carry out frequency response estimation against a physical plant in real time. For more information, see “Online Frequency Response Estimation Basics” on page 6-2.

Nonlinear Plant Model

This example uses a stable nonlinear SISO plant. The plant has two states. Trim the model to find an initial steady-state operating point at which the plant output is zero.

```
plantMDL = 'scdfrePlant';
y0 = 0;
op = operspec(plantMDL);
op.Outputs.Known = true;
op.Outputs.y = y0;
options = findopOptions('DisplayReport','off');
[op_point, op_report] = findop(plantMDL,op,options);
x0 = [op_report.States(1).x;op_report.States(2).x];
y0 = op_report.Outputs.y;
u0 = op_report.Inputs.u;
```

A goal of this example is to obtain plant frequency responses from 0.1 rad/s to 10 rad/s at two other steady-state operating points, plant output = 0.5 and plant output = -0.5. To bring the plant to these operating points, design a discrete PID controller at the initial operating point. Use a controller sample time of 0.01 sec and an open-loop bandwidth is 20 rad/s.

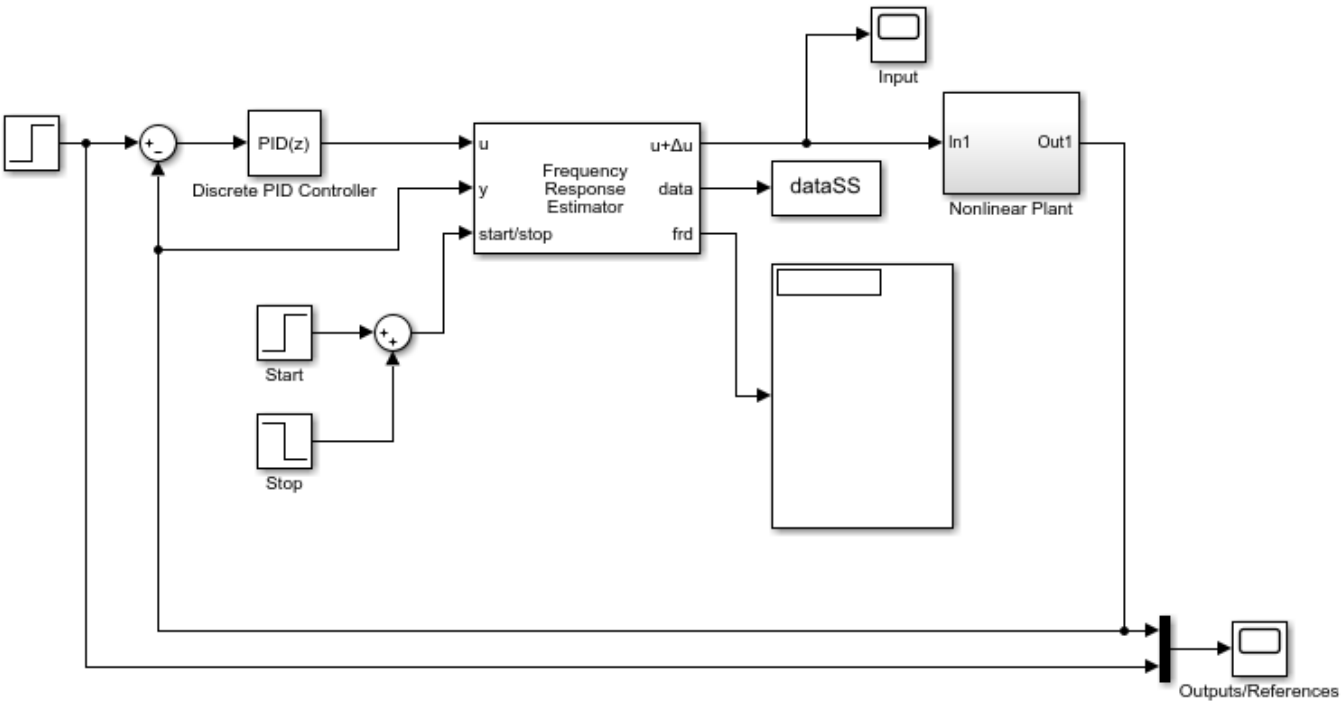
```
Ts = 0.01;
G0 = c2d(linearize(plantMDL,op_point),Ts);
c = pidtune(G0,'pidf',20);
```

Online Estimation Using Sinestream Mode

The model `scdfreSinestream` includes the plant in a PID control loop using the controller `c`. It also contains the Frequency Response Estimator block in the **control action + perturbation** output

configuration. In that configuration, you insert the block into the control loop between controller and plant.

```
mdlSS = 'scdfreSinestream';
open_system(mdlSS);
```



Copyright 1990-2018 The MathWorks, Inc.

You can use the **start/stop** signal to start and stop an online estimation experiment. When the block is idle, the control signal passes through the block without any change.

During the experiment, when the **Experiment Mode** is **Sinestream**, the block injects sinusoidal signals into to the plant one frequency after another, from the lowest to the highest. Compared with the **Superposition** mode, the sinestream experiment is less intrusive and more accurate. However, it requires a much longer time to conduct the experiment.

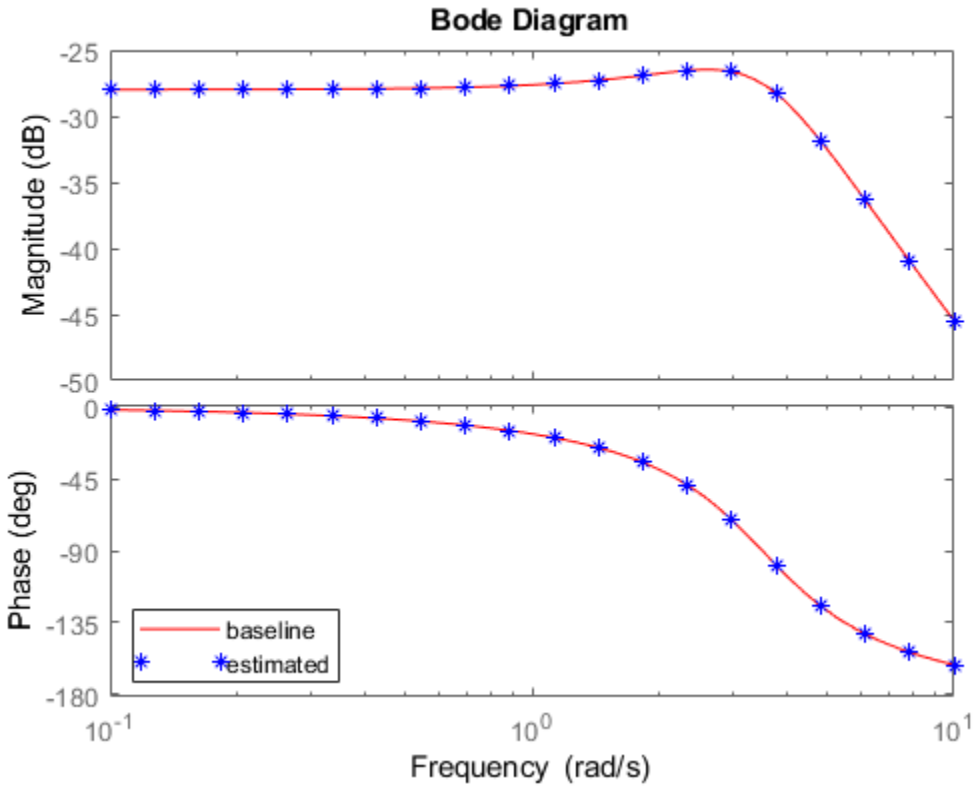
In this example, you can obtain an exact linearization of the model. Therefore, you can use it as the baseline in the block, letting you directly compare estimation result with this "truth" at run time. The block is configured to use a model `G1` as the baseline for the Bode plot that it generates. Find a steady-state operating point at which the plant output is 0.5, and linearize at this operating point to obtain `G1`.

```
op.Outputs.y = 0.5;
op_point = findop(plantMDL,op,options);
G1 = c2d(linearize(plantMDL,op_point),Ts);
```

The experiment starts at 10 sec after PID controller moves the plant to the new operating point (y = 0.5). After the experiment starts, the PID controller tries to reject the injected sine waves, which are

effectively load disturbance. Thus the controller ensures that the plant does not move too far away from the nominal operating point during the experiment, and reduces the impact of plant nonlinearity on the estimation result. Simulate the model and observe on the Bode plot how the estimated response evolves during the experiment. The estimation result matches the baseline very well.

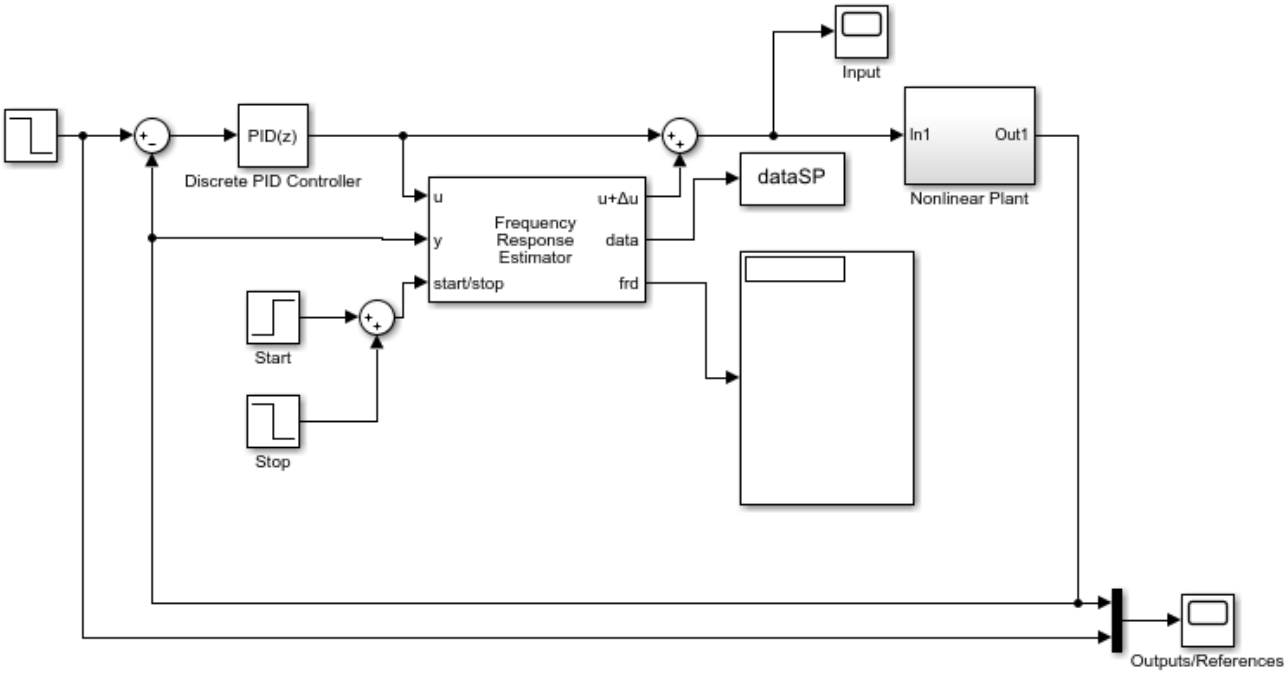
```
sim(mdlSS);
figSS = gcf;
hold on;
```



Online Estimation Using Superposition Mode

Open another model, `scdfreeSuperposition`. In this model, the Frequency Response Estimator block is configured for **perturbation only** output. In this configuration, you can position to block outside of the control loop. When the block is idle, the perturbation signal entering the Sum block is 0, so the loop is unaffected.

```
mdlSP = 'scdfreSuperposition';
open_system(mdlSP);
```



Copyright 1990-2018 The MathWorks, Inc.

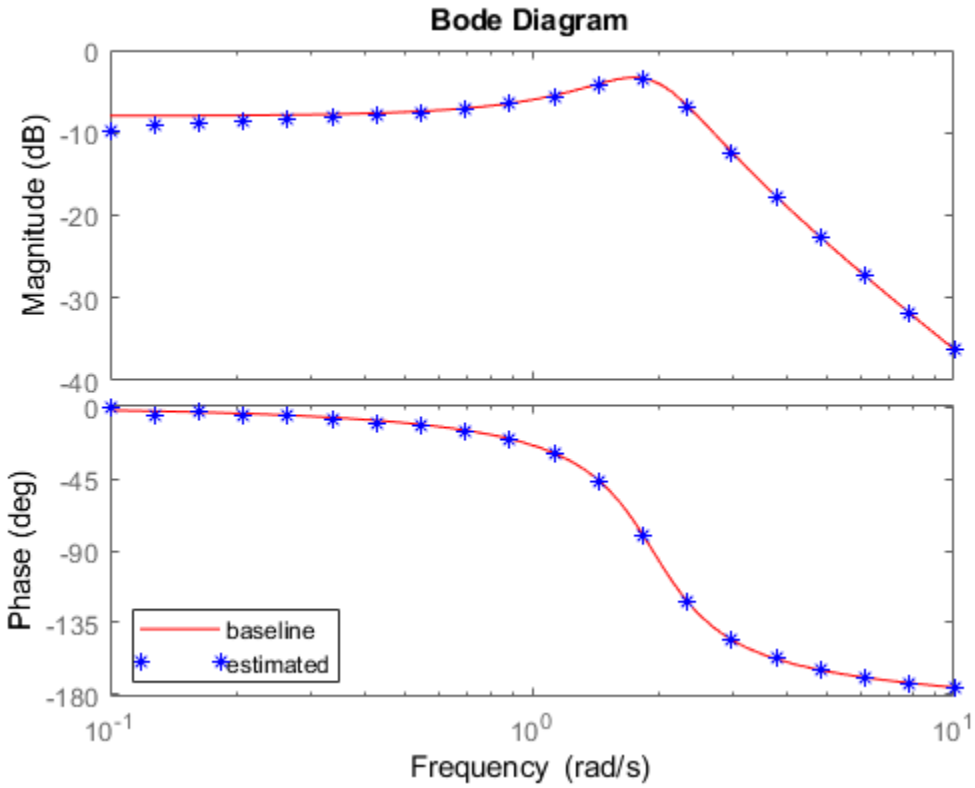
This model has the same plant and controller. However, the Frequency Response Estimator block in this model is configured to use the **Superposition** experiment mode. In this mode, all the sine waves are added together and injected to the plant at the same time. Compared with the sinestream experiment, the superposition experiment is must faster (especially when you are targeting low frequencies).

Find a steady-state operating point with plant output of -0.5. Linearize the plant to find a baseline response at this operating point, `G2`. The Frequency Response Estimator block is configured to use this baseline to generate a Bode plot.

```
op.Outputs.y = -0.5;
op_point = findop(plantMDL,op,options);
G2 = c2d(linearize(plantMDL,op_point),Ts);
```

The experiment starts at 10 sec after the PID controller moves the plant to the new operating point (y = -0.5). Note that the recommended experiment length is 377 seconds, much shorter than 1738 seconds used in the sinestream experiment. Simulate the model and again observe the progress of the estimation on the Bode plot.

```
sim(mdlSP);
figSP = gcf;
hold on;
```



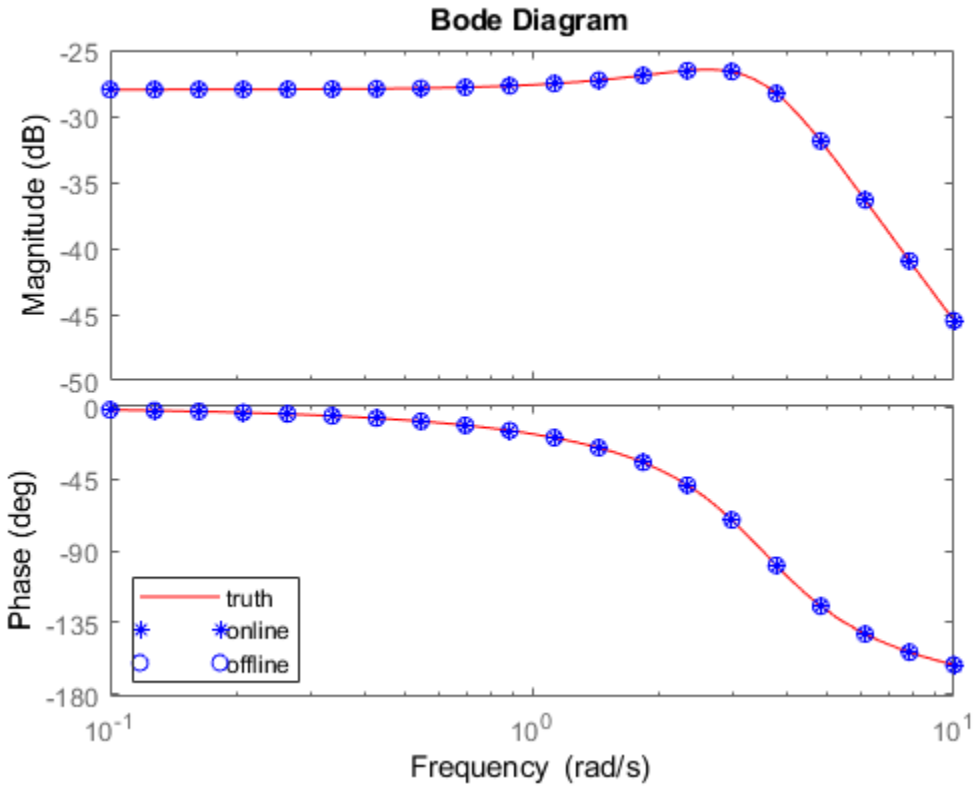
Offline Estimation Using Logged Experiment Data

Frequency Response Estimator block has a `data` outport that allows you to log the experiment data from simulation or in real-time. You can process that data set offline with the `frestimate` command to generate an `frd` object.

```
w = logspace(-1,1,20);
```

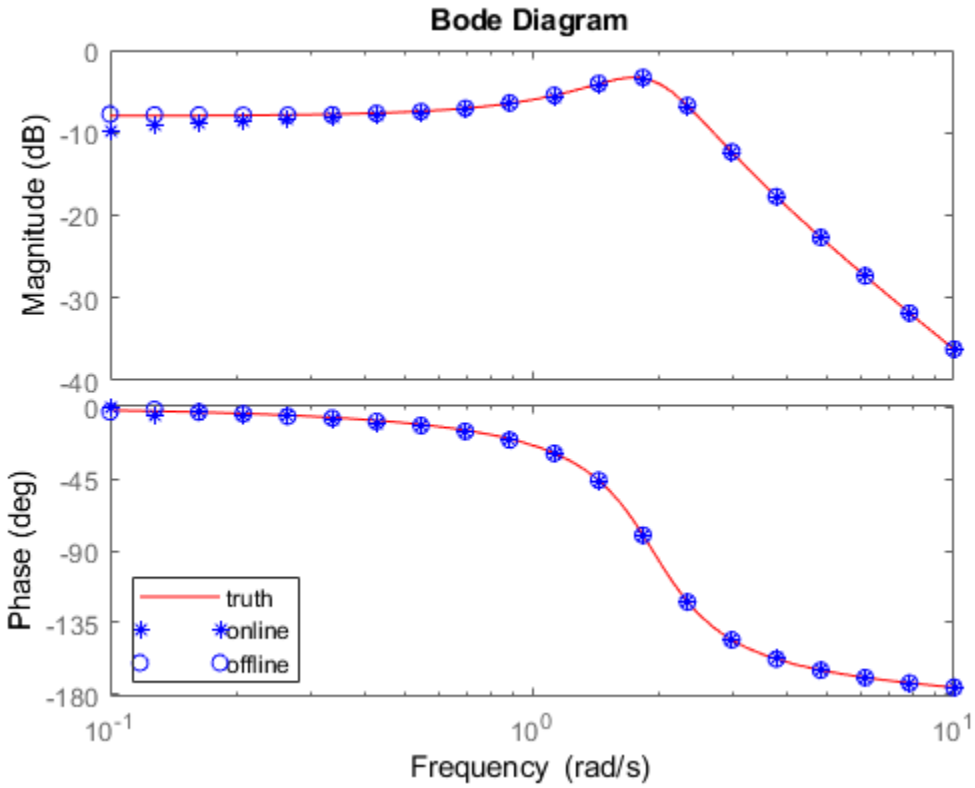
Compare online and offline estimation results from the sinestream experiment.

```
G1frd = frestimate(dataSS,w,'rad/s');
figure(figSS);
bodeplot(gca,G1frd,w,'o');
legend('truth','online','offline')
```



Compare online and offline estimation results from the superposition experiment.

```
G2frd = frestimate(dataSP,w,'rad/s');
figure(figSP);
bodeplot(gca,G2frd,w,'o');
legend('truth','online','offline')
```



For more information about using experiment data for offline estimation, see “Collect Frequency Response Experiment Data for Offline Estimation” on page 6-18.

See Also

Frequency Response Estimator

More About

- “Online Frequency Response Estimation Basics” on page 6-2
- “Online Estimation Using Plant Modeled in Simulink” on page 6-5

PID Controller Tuning

- “Introduction to Model-Based PID Tuning in Simulink” on page 7-2
- “Open PID Tuner” on page 7-5
- “Analyze Design in PID Tuner” on page 7-8
- “Verify the PID Design in Your Simulink Model” on page 7-13
- “Tune at a Different Operating Point” on page 7-14
- “Tune PID Controller to Favor Reference Tracking or Disturbance Rejection” on page 7-17
- “Design Two-Degree-of-Freedom PID Controllers” on page 7-26
- “Tune PID Controller Within Model Reference” on page 7-30
- “Specify PI-D and I-PD Controllers” on page 7-33
- “Design PID Controller from Plant Frequency-Response Data” on page 7-36
- “Frequency-Response Based Tuning” on page 7-37
- “Design PID Controller Using Plant Frequency Response Near Bandwidth” on page 7-43
- “Import Measured Response Data for Plant Estimation” on page 7-51
- “Interactively Estimate Plant from Measured or Simulated Response Data” on page 7-55
- “System Identification for PID Control” on page 7-61
- “Preprocess Data” on page 7-64
- “Input/Output Data for Identification” on page 7-67
- “Choosing Identified Plant Structure” on page 7-68
- “Design Controller for Power Electronics Model Using Frequency Response Data” on page 7-76
- “Design Controller for Power Electronics Model Using Simulated I/O Data” on page 7-94
- “Design PID Controller Using Simulated I/O Data” on page 7-108
- “Design PID Controller Using Estimated Frequency Response” on page 7-124
- “Design Family of PID Controllers for Multiple Operating Points” on page 7-132
- “Implement Gain-Scheduled PID Controllers” on page 7-139
- “Design Controller for Vehicle Platooning” on page 7-145
- “Plant Cannot Be Linearized or Linearizes to Zero” on page 7-153
- “Cannot Find a Good Design in PID Tuner” on page 7-154
- “Simulated Response Does Not Match PID Tuner Response” on page 7-155
- “Cannot Find Acceptable PID Design in Simulated Model” on page 7-157
- “Controller Performance Deteriorates When Switching Time Domains” on page 7-158
- “When Tuning the PID Controller, the D Gain Has a Different Sign from the I Gain” on page 7-159
- “Tune Field-Oriented Controllers Using SYSTUNE” on page 7-160
- “Frequency Response Based PID Tuner” on page 7-175

Introduction to Model-Based PID Tuning in Simulink

You can use **PID Tuner** to interactively tune PID gains in a Simulink model containing a PID Controller, Discrete PID Controller, PID Controller (2DOF), or Discrete PID Controller (2DOF) block. **PID Tuner** allows you to achieve a good balance between performance and robustness for either one-degree-of-freedom or two-degree-of-freedom PID controllers. When you use **PID Tuner**, it:

- Automatically computes a linear model of the plant in your model. **PID Tuner** considers the plant to be the combination of all blocks between the PID controller output and input. Thus, the plant includes all blocks in the control loop, other than the controller itself. See “What Plant Does PID Tuner See?” on page 7-2.
- Automatically computes an initial PID design with a balance between performance and robustness. **PID Tuner** bases the initial design upon the open-loop frequency response of the linearized plant. See “PID Tuning Algorithm” on page 7-3.
- Provides tools and response plots to help you interactively refine the performance of the PID controller to meet your design requirements. See “Open PID Tuner” on page 7-5.

For plants that do not linearize or that linearize to zero, there are several alternatives for obtaining a plant model for tuning. These alternatives include:

- “Design PID Controller from Plant Frequency-Response Data” on page 7-36 — Use the frequency-response estimation command `frestimate` or the Frequency Response Based PID Tuner to obtain estimated frequency responses of the plant by simulation.
- “Interactively Estimate Plant from Measured or Simulated Response Data” on page 7-55 — If you have System Identification Toolbox, you can use PID Tuner to estimate the parameters of a linear plant model based on time-domain response data. PID Tuner then tunes a PID controller for the resulting estimated model. The response data can be either measured from your real-world system, or obtained by simulating your Simulink® model.

You can use **PID Tuner** to design one-degree-of-freedom or two-degree-of-freedom PID controllers. You can often achieve both good setpoint tracking and good disturbance rejection using a one-degree-of-freedom PID controller. However, depending upon the dynamics in your model, using a one-degree-of-freedom PID controller can require a tradeoff between setpoint tracking and disturbance rejection. In such cases, if you need both good setpoint tracking and good disturbance rejection, use a two-degree-of-freedom PID Controller.

For examples of tuning one- and two-degree-of-freedom PID compensators, see:

- “PID Controller Tuning in Simulink”
- “Tune PID Controller to Favor Reference Tracking or Disturbance Rejection” on page 7-17

What Plant Does PID Tuner See?

PID Tuner considers as the plant all blocks in the loop between the PID Controller block output and input. The blocks in your plant can include nonlinearities. Because automatic tuning requires a linear model, **PID Tuner** computes a linearized approximation of the plant in your model. This linearized model is an approximation to a nonlinear system, which is valid in a small region around a given operating point of the system.

By default, **PID Tuner** linearizes your plant using the initial conditions specified in your Simulink model as the operating point. The linearized plant can be of any order and can include any time delays. The **PID tuner** designs a controller for the linearized plant.

In some circumstances, however, you want to design a PID controller for a different operating point from the one defined by the model initial conditions. For example:

- The Simulink model has not yet reached steady-state at the operating point specified by the model initial conditions, and you want to design a controller for steady-state operation.
- You are designing multiple controllers for a gain-scheduling application and must design each controller for a different operating point.

In such cases, change the operating point used by **PID Tuner**. See “Opening PID Tuner” on page 7-5.

For more information about linearization, see “Linearize Nonlinear Models” on page 2-3.

PID Tuning Algorithm

Typical PID tuning objectives include:

- Closed-loop stability — The closed-loop system output remains bounded for bounded input.
- Adequate performance — The closed-loop system tracks reference changes and suppresses disturbances as rapidly as possible. The larger the loop bandwidth (the frequency of unity open-loop gain), the faster the controller responds to changes in the reference or disturbances in the loop.
- Adequate robustness — The loop design has enough gain margin and phase margin to allow for modeling errors or variations in system dynamics.

MathWorks algorithm for tuning PID controllers meets these objectives by tuning the PID gains to achieve a good balance between performance and robustness. By default, the algorithm chooses a crossover frequency (loop bandwidth) based on the plant dynamics, and designs for a target phase margin of 60°. When you interactively change the response time, bandwidth, transient response, or phase margin using the **PID Tuner** interface, the algorithm computes new PID gains.

For a given robustness (minimum phase margin), the tuning algorithm chooses a controller design that balances the two measures of performance, reference tracking and disturbance rejection. You can change the design focus to favor one of these performance measures. To do so, use the **Options** dialog box in **PID Tuner**.

When you change the design focus, the algorithm attempts to adjust the gains to favor either reference tracking or disturbance rejection, while achieving the same minimum phase margin. The more tunable parameters there are in the system, the more likely it is that the PID algorithm can achieve the desired design focus without sacrificing robustness. For example, setting the design focus is more likely to be effective for PID controllers than for P or PI controllers. In all cases, fine-tuning the performance of the system depends strongly on the properties of your plant. For some plants, changing the design focus has little or no effect.

See Also

Apps
PID Tuner

Blocks
PID Controller | Discrete PID Controller | PID Controller (2DOF) | Discrete PID Controller (2DOF)

More About

- “Choose a Control Design Approach” on page 9-2

Open PID Tuner

You can use **PID Tuner** to interactively tune PID gains in a Simulink model containing a PID Controller, Discrete PID Controller, PID Controller (2DOF), or Discrete PID Controller (2DOF) block. For more information, see “Introduction to Model-Based PID Tuning in Simulink” on page 7-2.

Prerequisites for PID Tuning

Before you can use **PID Tuner**, you must:

- Create a Simulink model containing a PID Controller, Discrete PID Controller, PID Controller (2DOF), or Discrete PID Controller (2DOF) block. Your model can have one or more PID blocks, but you can only tune one PID block at a time.

  - If you are tuning a multi-loop control system with coupling between the loops, consider using other Simulink Control Design tools instead of **PID Tuner**. For more information, see “Choose a Control Design Approach” on page 9-2.
  - The PID Controller blocks support vector signals. However, using **PID Tuner** requires scalar signals at the block inputs. That is, the PID block must represent a single PID controller.

  Your plant (all blocks in the control loop other than the controller) can be linear or nonlinear. The plant can also be of any order, and have any time delays.
- Configure the PID block settings, such as controller type, controller form, time domain, sample time. For more information on these block settings, see the individual block reference pages:

  - PID Controller
  - Discrete PID Controller
  - PID Controller (2DOF)
  - Discrete PID Controller (2DOF)

Opening PID Tuner

To open **PID Tuner** and view the initial compensator design:

1. Open the Simulink model by typing the model name at the MATLAB command prompt.
2. To open the block dialog box, double-click the PID controller block.
3. In the block dialog box, in the **Select Tuning Method** drop-down list, select `Transfer Function Based (PID Tuner App)`. To open **PID Tuner**, click **Tune**.

When you open **PID Tuner**, the following actions occur:

- **PID Tuner** automatically linearizes the plant at the operating point specified by the model initial conditions, as described in “What Plant Does PID Tuner See?” on page 7-2. If you want to design a controller for a different operating point, see “Tune at a Different Operating Point” on page 7-14.

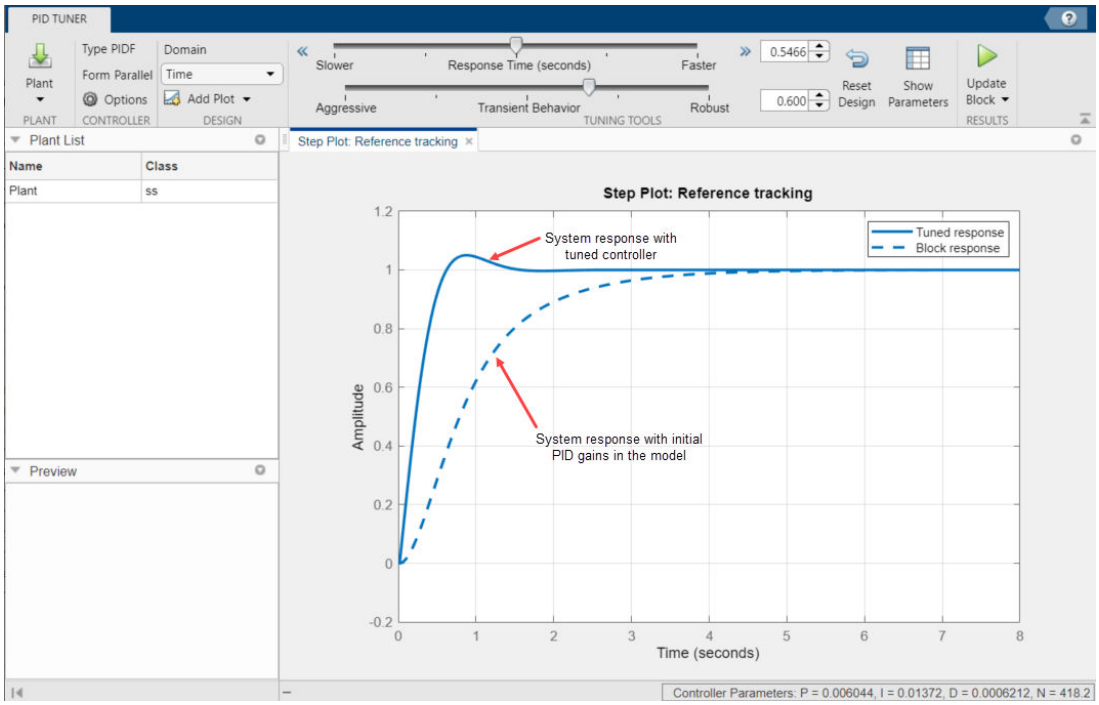
  Note If the plant model in the PID loop linearizes to zero, **PID Tuner** provides the **Obtain plant model** dialog box. This dialog box allows you to obtain a new plant model by either:

  - Linearizing at a different operating point (see “Tune at a Different Operating Point” on page 7-14).

- Importing an LTI model object representing the plant. For example, you can import frequency response data (`frd` model) obtained by frequency response estimation. For more information, see “Design PID Controller Using Estimated Frequency Response” on page 7-124.
- Identifying a linear plant model from simulated or measured response data (requires System Identification Toolbox software). **PID Tuner** uses system identification to estimate a linear plant model from the time-domain response of your plant to an applied input. For an example, see “Interactively Estimate Plant from Measured or Simulated Response Data” on page 7-55.

As an alternative, you can exit **PID Tuner** and use the **Frequency Response Based PID Tuner**, which runs simulations to perturb the plant and estimate frequency responses at frequencies near the control bandwidth. See “Frequency-Response Based Tuning” on page 7-37.

- **PID Tuner** computes an initial compensator design for the linearized plant model using the algorithm described in “PID Tuning Algorithm” on page 7-3.
- **PID Tuner** displays the closed-loop step reference tracking response for the initial compensator design. For comparison, the display also includes the closed-loop response for the gains specified in the PID controller block, if that closed loop is stable, as shown in the following figure.



Tip After the tuner opens, you can close the controller block dialog box.

See Also

Apps
PID Tuner

Blocks
PID Controller | Discrete PID Controller | PID Controller (2DOF) | Discrete PID Controller (2DOF)

More About

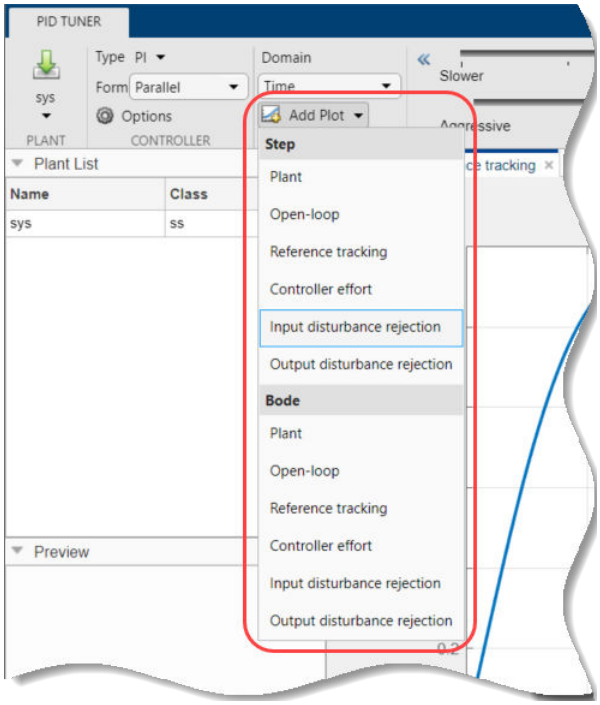
- “Introduction to Model-Based PID Tuning in Simulink” on page 7-2

Analyze Design in PID Tuner

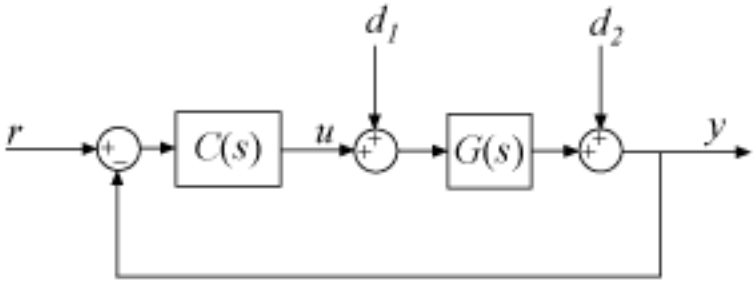
To determine whether your PID controller meets your requirements, you can analyze the system response using the **PID Tuner** response plots.

Plot System Responses

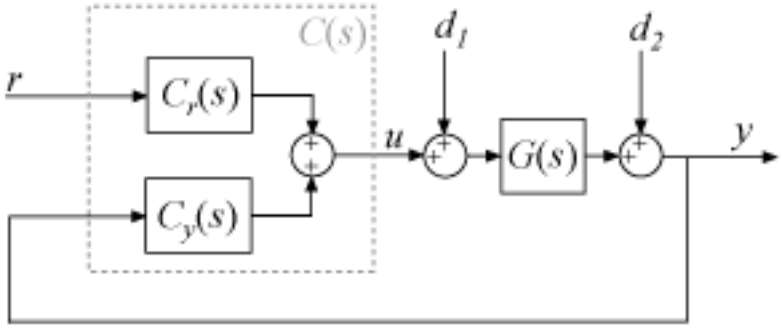
To determine whether the compensator design meets your requirements, you can analyze the system response using the response plots. On the **PID Tuner** tab, select a response plot from the **Add Plot** menu. The **Add Plot** menu also lets you choose from several step plots (time-domain response) or Bode plots (frequency-domain response).



For 1-DOF PID controller types such as PI, PIDF, and PDF, the software computes system responses based upon the following single-loop control architecture, where G is your specified plant and C is the PID controller:



For 2-DOF PID controller types such as PI2, PIDF2, and I-PD, the software computes responses based upon the following architecture:



The system responses are based on the decomposition of the 2-DOF PID controller, C, into a setpoint component Cr and a feedback component Cy, as described in “Two-Degree-of-Freedom PID Controllers”.

The following table summarizes the available responses for analysis plots.

Response	Plotted System (1-DOF)	Plotted System (2-DOF)	Description
`Plant`	G	G	Plant response. Use to examine plant dynamics.
`Open-loop`	GC	$-GC_y$	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.
`Reference tracking`	$\frac{GC}{1+GC}$ (from r to y)	$\frac{GC_r}{1-GC_y}$ (from r to y)	Closed-loop system response to a step change in setpoint. Use when your design specifications include setpoint tracking.
`Controller effort`	$\frac{C}{1+GC}$ (from r to u)	$\frac{C_r}{1-GC_y}$ (from r to u)	Closed-loop controller output response to a step change in setpoint. Use when your design is limited by practical constraints, such as controller saturation.

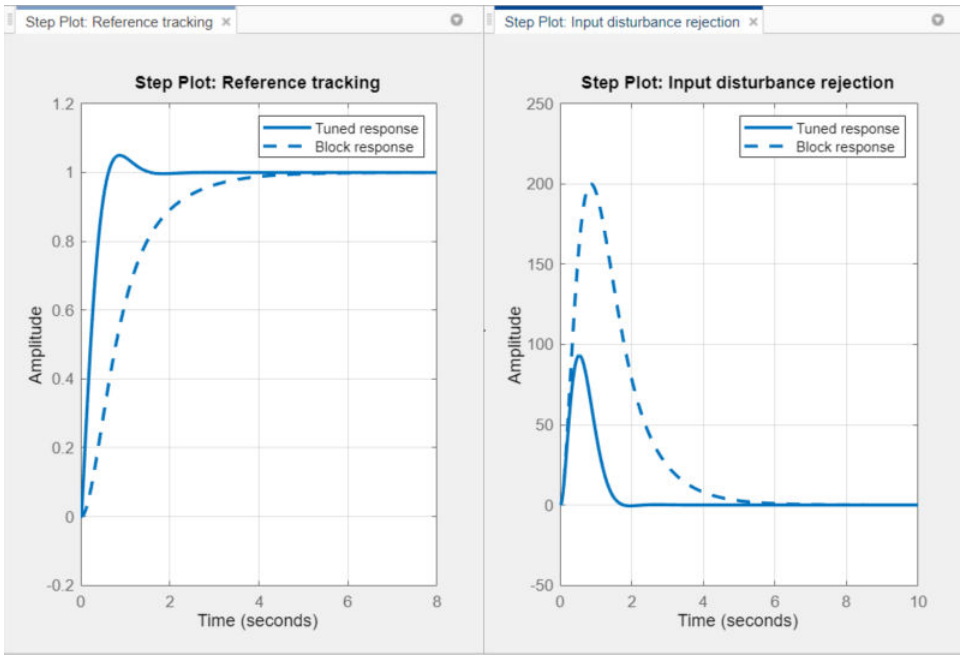
Response	Plotted System (1-DOF)	Plotted System (2-DOF)	Description
`Input disturbance rejection`	$\frac{G}{1+GC}$ (from d_1 to y)	$\frac{G}{1-GC_y}$ (from d_1 to y)	Closed-loop system response to load disturbance (a step disturbance at the plant input). Use when your design specifications include input disturbance rejection.
`Output disturbance rejection`	$\frac{1}{1+GC}$ (from d_2 to y)	$\frac{1}{1-GC_y}$ (from d_2 to y)	Closed-loop system response to a step disturbance at plant output. Use when you want to analyze sensitivity to modeling errors.

Compare Tuned Response to Block Response

By default, **PID Tuner** plots system responses using both:

- The PID coefficient values in the controller block in the Simulink model (Block response).
- The PID coefficient values of the current **PID Tuner** design (Tuned response).

As you adjust the current **PID Tuner** design, such as by moving the sliders, the Tuned response plots change, while the Block response plots do not.



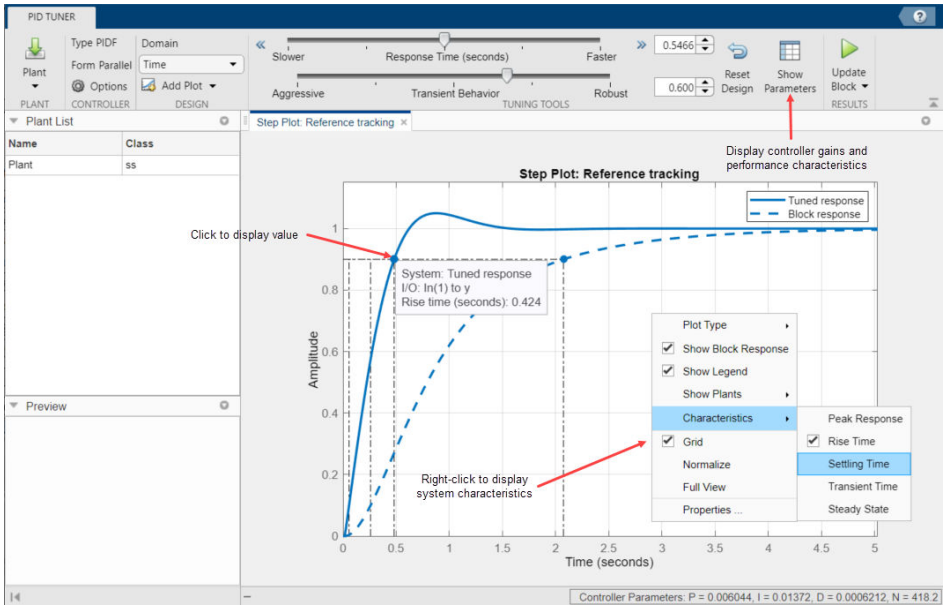
To write the current **PID Tuner** design to the Simulink model, click . When you do so, the current Tuned response becomes the Block response. Further adjustment of the current design creates a new Tuned response line.

To hide the Block response, click **Options**, and clear **Show Block Response**.

View Numeric Values of System Characteristics

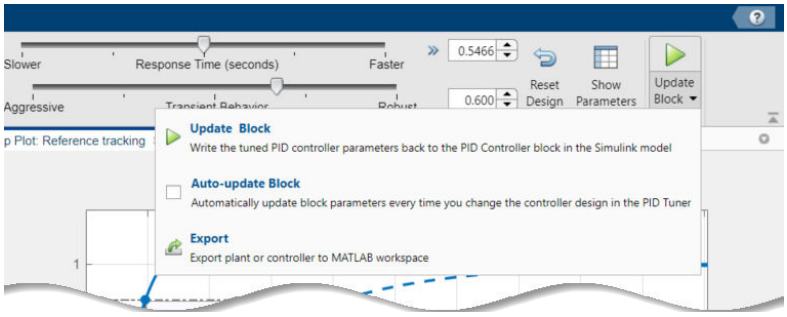
You can view the values for system characteristics, such as peak response and gain margin, either:

- Directly on the response plot — Use the right-click menu to add characteristics, which appear as blue markers. Then, left-click the marker to display the corresponding data panel.
- In the **Performance and robustness** table — To display this table, click **Show Parameters**.



Export Plant or Controller to MATLAB Workspace

You can export the linearized plant model computed by **PID Tuner** to the MATLAB workspace for further analysis. To do so, click **Update Block** and select **Export**.



In the Export dialog box, check the models that you want to export. Click **OK** to export the plant or controller to the MATLAB workspace as state-space (`ss`) model object or `pid` object, respectively.

Refine the Design

If the response of the initial controller design does not meet your requirements, you can interactively adjust the design. **PID Tuner** gives you two **Domain** options for refining the controller design:

- `Time` domain (default) — Use the **Response Time** slider to make the closed-loop response of the control system faster or slower. Use the **Transient Behavior** slider to make the controller more aggressive at disturbance rejection or more robust against plant uncertainty.
- `Frequency` — Use the **Bandwidth** slider to make the closed-loop response of the control system faster or slower (the response time is $2/w_c$, where w_c is the bandwidth). Use the **Phase Margin** slider to make the controller more aggressive at disturbance rejection or more robust against plant uncertainty.

In both modes, there is a tradeoff between reference tracking and disturbance rejection performance. For an example that shows how to use the sliders to adjust this tradeoff, see “Tune PID Controller to Favor Reference Tracking or Disturbance Rejection” on page 7-17.

Once you find a compensator design that meets your requirements, verify that it behaves in a similar way in the nonlinear Simulink model. For instructions, see “Verify the PID Design in Your Simulink Model” on page 7-13.

Tip To revert to the initial controller design after moving the sliders, click **Reset Design**.

See Also

PID Tuner

More About

- “Introduction to Model-Based PID Tuning in Simulink” on page 7-2

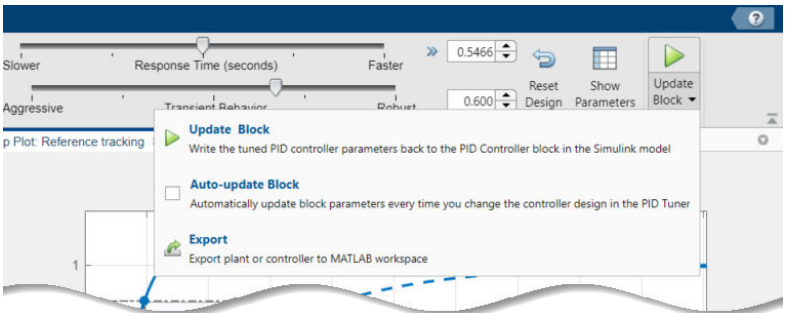
Verify the PID Design in Your Simulink Model

In **PID Tuner**, you tune the compensator using a linear model of your plant. First, you find a good compensator design in **PID Tuner**. Then, verify that the tuned controller meets your design requirements when applied to the nonlinear plant in your Simulink model.

To verify the compensator design in the nonlinear Simulink model:

1. In the **PID Tuner** tab, click to update the Simulink PID controller block with the tuned PID parameters.

   Tip To update PID block parameters automatically as you tune the controller in **PID Tuner**, click **Update Block** and check **Auto-update block**.

2. Simulate the Simulink model, and evaluate whether the simulation output meets your design requirements.

Because **PID Tuner** works with a linear model of your plant, the simulated response sometimes does not match the response in **PID Tuner**. See “Simulated Response Does Not Match PID Tuner Response” on page 7-155 for more information.

If the simulated response does not meet your design requirements, see “Cannot Find Acceptable PID Design in Simulated Model” on page 7-157.

Tune at a Different Operating Point

By default, **PID Tuner** linearizes your plant and designs a controller at the operating point specified by the initial conditions in your Simulink model. Sometimes, this operating point differs from the operating point for which you want to design a controller. For example, you want to design a controller for your system at steady-state. However, the Simulink model is not generally at steady-state at the initial condition. In this case, change the operating point that **PID Tuner** uses for linearizing your plant and designing a controller.

To set a new operating point for **PID Tuner**, use one of the following methods. The method you choose depends upon the information you have about your desired operating point.

Known State Values Yield the Desired Operating Conditions

In this case, set the state values in the model directly.

1. Close **PID Tuner**.
2. Set the initial conditions of the components of your model to the values that yield the desired operating conditions.
3. Click **Tune** in the PID controller dialog box to open **PID Tuner**. **PID Tuner** linearizes the plant using the new default operating point and designs a new initial controller for the new linear plant model.

After **PID Tuner** generates a new initial controller design, continue from “Analyze Design in PID Tuner” on page 7-8.

Model Reaches Desired Operating Conditions at a Finite Time

In this case, use **PID Tuner** to relinearize the model at a particular simulation time.

1. In the **PID Tuner** tab, in the **Plant** menu, select **Re-linearize Closed Loop**.
2. In the **Closed Loop Re-Linearization** tab, click **Run Simulation** to simulate the model for the time specified in the **Simulation Time** text box.

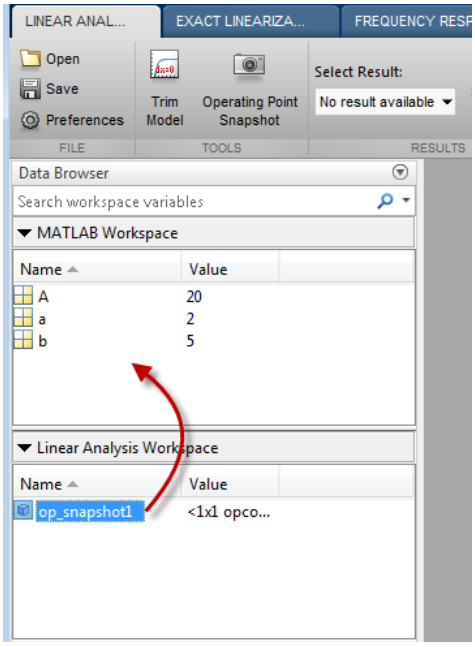
   PID Tuner plots the error signal as a function of time. You can use this plot to identify a time at which the model is in steady-state. Slide the vertical bar to a snapshot time at which you want to linearize the model.
3. Click **Linearize** to linearize the model at the selected snapshot time. **PID Tuner** computes a new linearized plant and saves it to the **PID Tuner** workspace. **PID Tuner** automatically designs a controller for the new plant, and displays a response plot for the new closed-loop system. **PID Tuner** returns you **PID Tuner** tab, where the **Plant** menu reflects that the new plant is selected for the current controller design.

   Note For models with Trigger-Based Operating Point Snapshot blocks, the software captures an operating point at events that trigger before the simulation reaches the snapshot time.

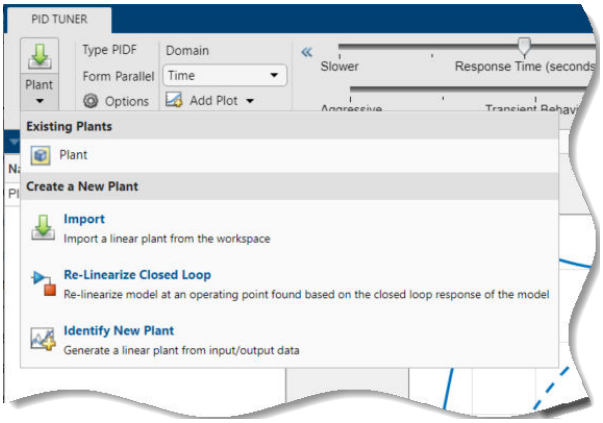
After **PID Tuner** generates a new initial controller design, continue from “Analyze Design in PID Tuner” on page 7-8.

You Computed an Operating Point in Model Linearizer

1 In the **Model Linearizer** app, drag the saved operating point object from the Linear Analysis Workspace to the MATLAB Workspace.



2 In **PID Tuner**, in the **PID Tuner** tab, in the **Plant** menu, select **Import**.



3 Select **Importing an LTI system or linearizing at an operating point defined in MATLAB workspace**. Select your exported operating point in the table.

4 Click **OK**. **PID Tuner** computes a new linearized plant and saves it to the **PID Tuner** workspace. **PID Tuner** automatically designs a controller for the new plant, and displays a response plot for the new closed-loop system. **PID Tuner** returns you **PID Tuner** tab, where the **Plant** menu reflects that the new plant is selected for the current controller design.

After **PID Tuner** generates a new initial controller design, continue from “Analyze Design in PID Tuner” on page 7-8.

See Also

More About

- “About Operating Points” on page 1-2
- “Compute Steady-State Operating Points” on page 1-5

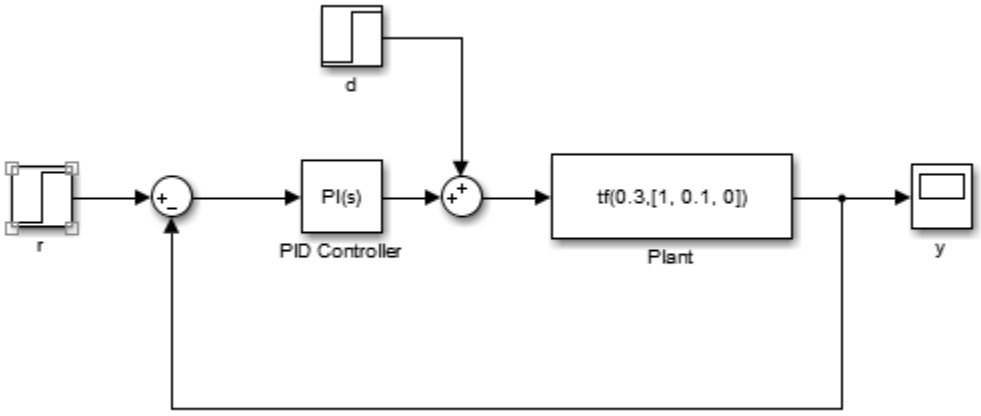
Tune PID Controller to Favor Reference Tracking or Disturbance Rejection

This example shows how to tune a PID controller to reduce overshoot in reference tracking or to improve rejection of a disturbance at the plant input. Using the **PID Tuner** app, the example illustrates the tradeoff between reference tracking and disturbance-rejection performance in PI and PID control systems.

Design Initial PI Controller

Load a Simulink model that contains a PID controller block.

```
open_system('singlePILoop')
```

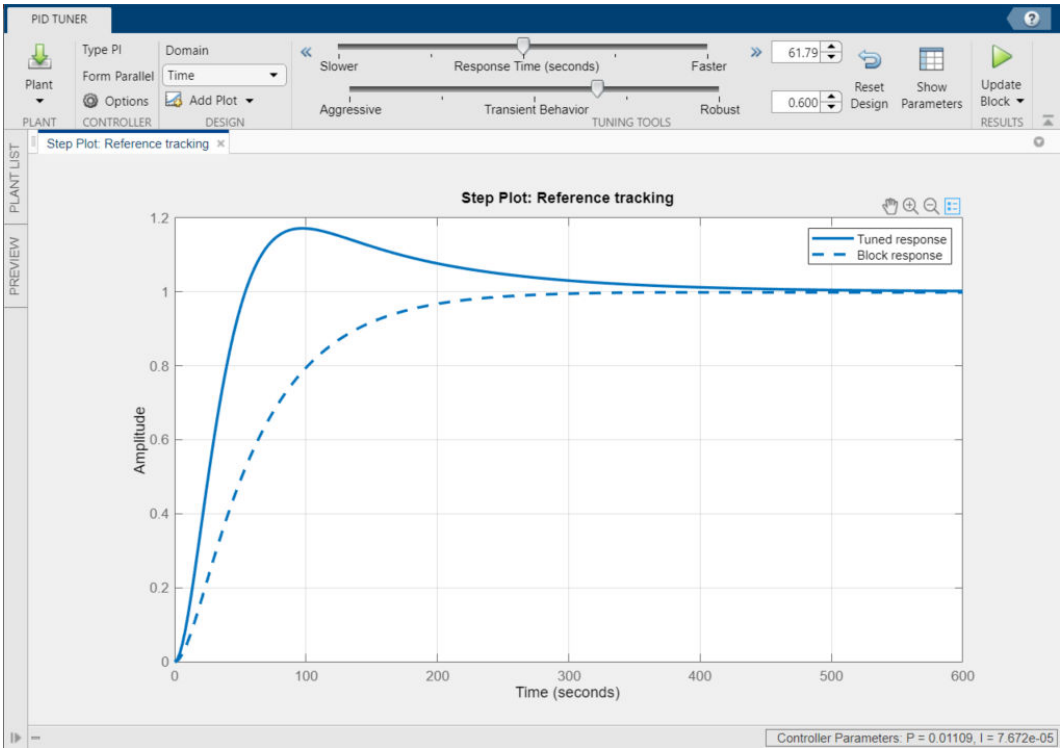


The plant in this example is:

$$\text{Plant} = \frac{0.3}{s^2 + 0.1s}.$$

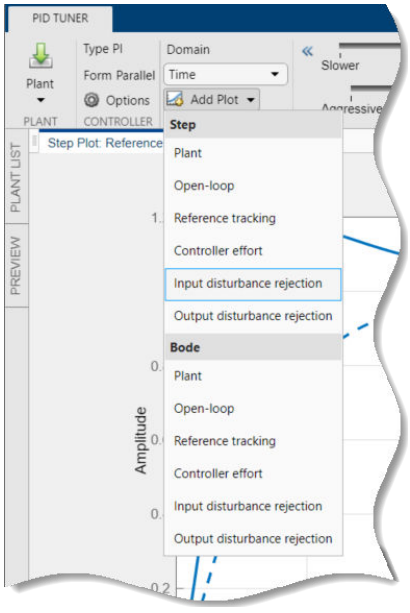
The model also includes a reference signal and a step disturbance at the plant input. Reference tracking is the response at `y` to the reference signal, `r`. Disturbance rejection is a measure of the suppression at `y` of the injected disturbance, `d`. When you use **PID Tuner** to tune the controller, you can adjust the design to favor reference tracking or disturbance rejection as your application requires.

Design an initial controller for the plant. To do so, double-click the PID controller block to open the Block Parameters dialog box, and click **Tune**. **PID Tuner** opens and automatically computes an initial controller design.

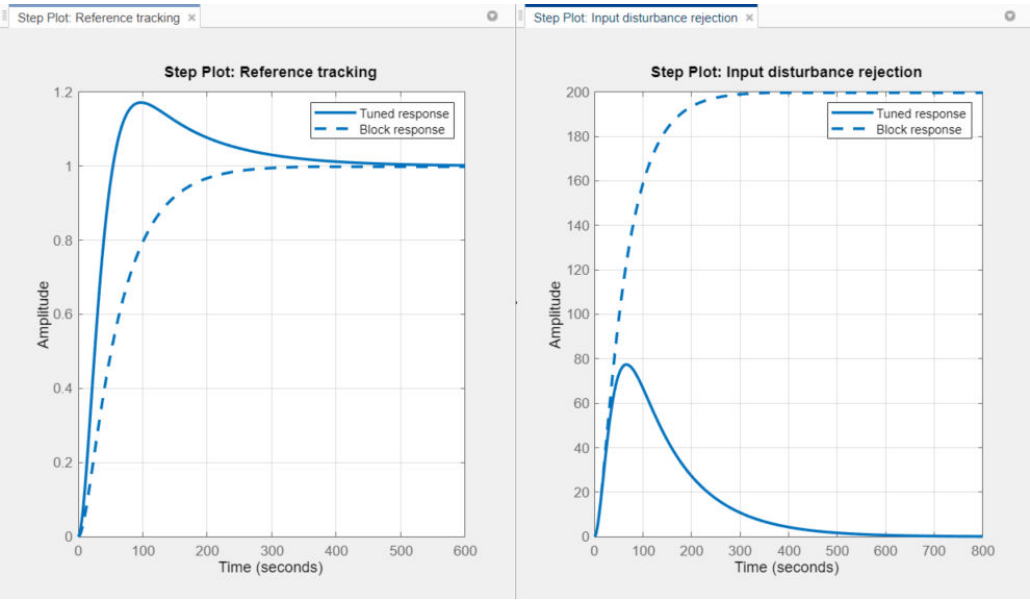


The controller in the Simulink model is configured as a PI-type controller. Therefore, the initial controller designed by **PID Tuner** is also of PI-type.

Add a step response plot of the input disturbance rejection. Select **Add Plot > Input Disturbance Rejection**.



PID Tuner tiles the disturbance-rejection plot in a new tab.



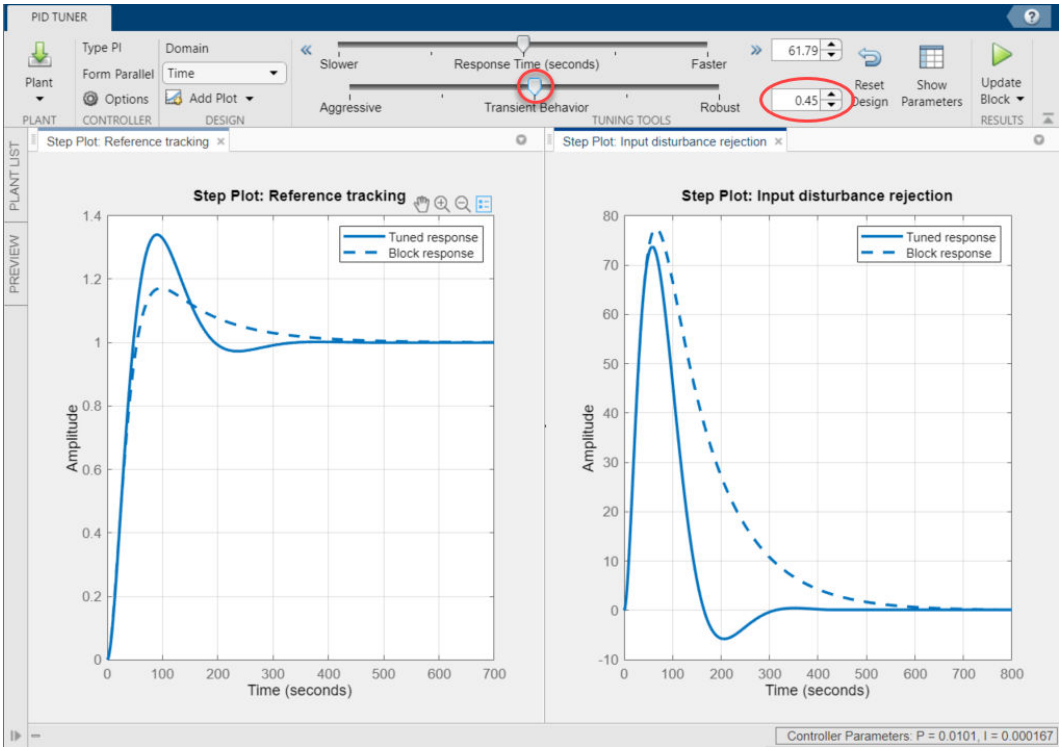
Tip Click and drag the tab to position the plots.

By default, for a given bandwidth and phase margin, **PID Tuner** tunes the controller to achieve a balance between reference tracking and disturbance rejection. In this case, the controller yields some overshoot in the reference-tracking response. The controller also suppresses the input disturbance with a longer settling time than the reference tracking, after an initial peak.

Click to update the Simulink model with this initial controller design. Doing so also updates the Block Response plots in **PID Tuner**, so that as you change the controller design, you can compare the results with the initial design.

Adjust Transient Behavior

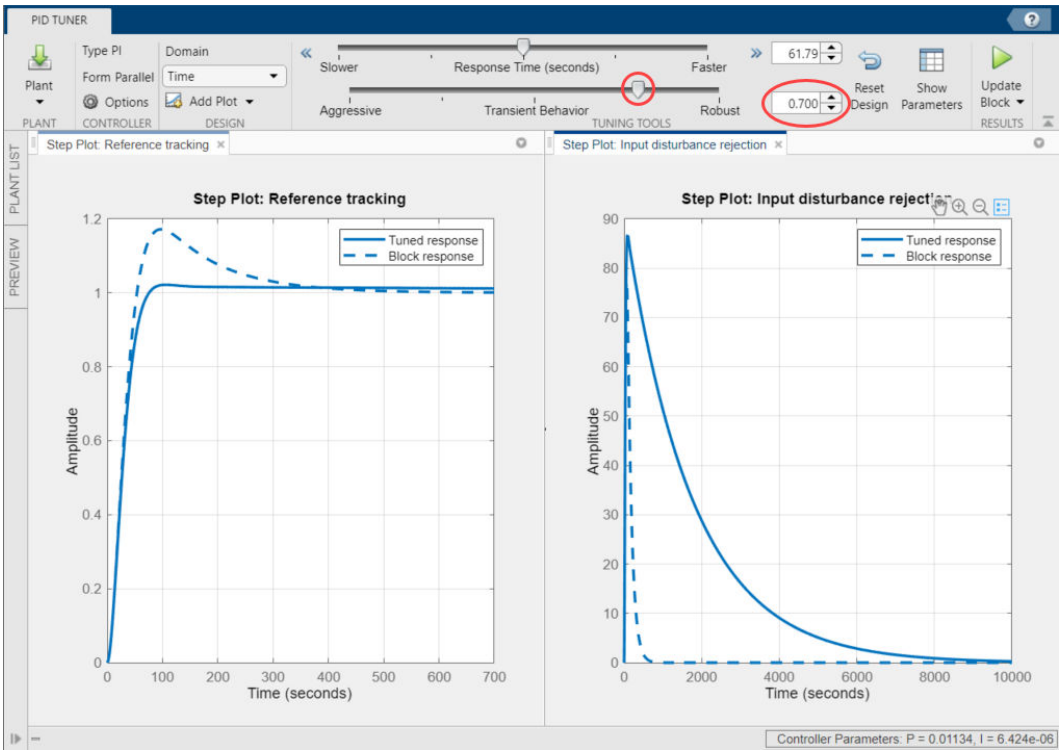
Depending on your application, you might want to alter the balance between reference tracking and disturbance rejection to favor one or the other. For a PI controller, you can alter this balance using the **Transient Behavior** slider. Move the **Transient behavior** slider to the left to improve the disturbance rejection. The responses with the initial controller design are now displayed as the `Block response` (dotted line).



Lowering the transient-behavior coefficient to 0.45 speeds up disturbance rejection, but also increases overshoot in the reference-tracking response.

Tip Right-click the reference-tracking plot and select **Characteristics > Peak Response** to obtain a numerical value for the overshoot.

Move the **Transient behavior** to the right until the overshoot in the reference-tracking response is minimized.

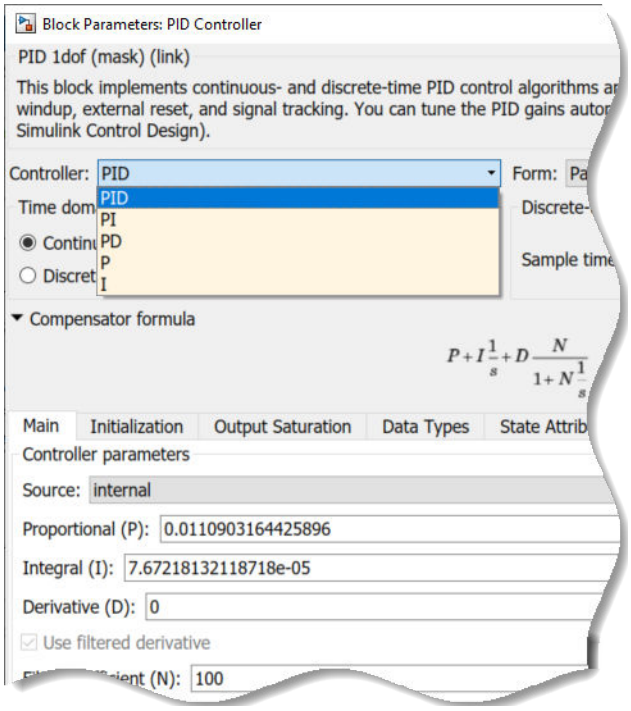


Increasing the transient-behavior coefficient to 0.70 nearly eliminates the overshoot, but results in sluggish disturbance rejection. You can try moving the **Transient behavior** slider until you find a suitable balance between reference tracking and disturbance rejection for your application. How much the slider affects the balance depends on the plant model. For some plant models, the effect is not as large as shown in this example.

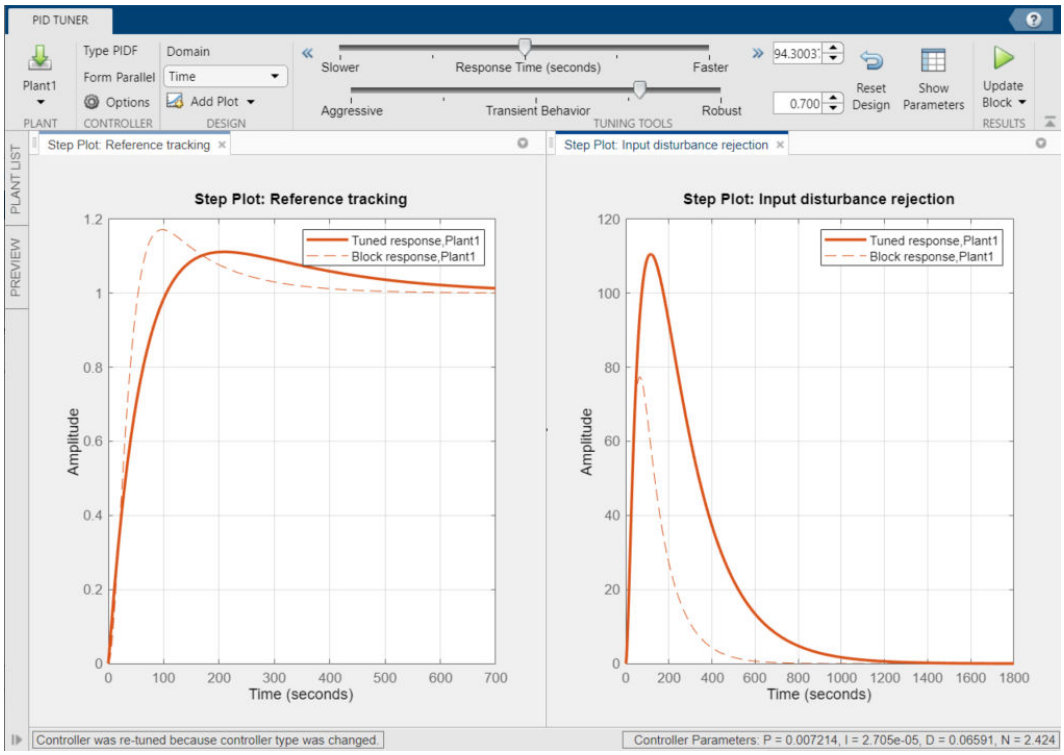
Change PID Tuning Design Focus

So far, the response time of the control system has remained fixed while you have changed the transient-behavior coefficient. These operations are equivalent to fixing the bandwidth and varying the target minimum phase margin of the system. If you want to fix both the bandwidth and target phase margin, you can still change the balance between reference tracking and disturbance rejection. To tune a controller that favors either disturbance rejection or reference tracking, you change the design focus of the PID tuning algorithm.

Changing the **PID Tuner** design focus is more effective the more tunable parameters there are in the control system. Therefore, it does not have much effect when used with a PI controller. To see its effect, change the controller type to PID. In the Simulink model, double-click the PID controller block. In the block parameters dialog box, in the **Controller** drop-down menu, select `PID`.

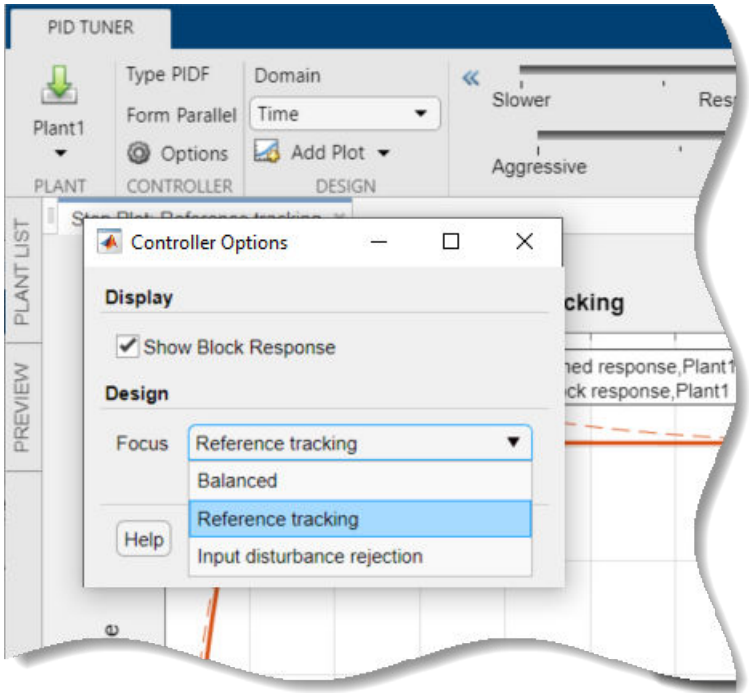


Click **Apply**. Then, click **Tune**. This action updates **PID Tuner** with a new controller design, this time for a PID controller. Click to the Simulink model with this initial PID controller design, so that you can compare the results when you change design focus.

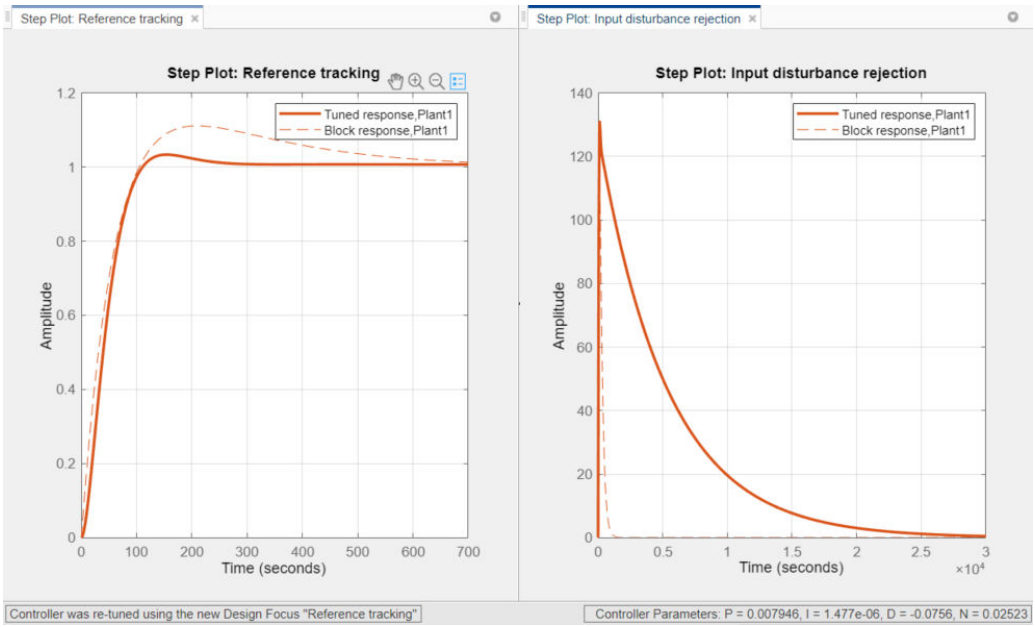


As in the PI case, the initial PID design balances reference tracking and disturbance rejection. In this case as well, the controller yields some overshoot in the reference-tracking response, and suppresses the input disturbance with a longer settling time.

Change the **PID Tuner** design focus to favor reference tracking without changing the response time or the transient-behavior coefficient. To do so, click **Options**, and in the **Focus** menu, select `Reference tracking`.

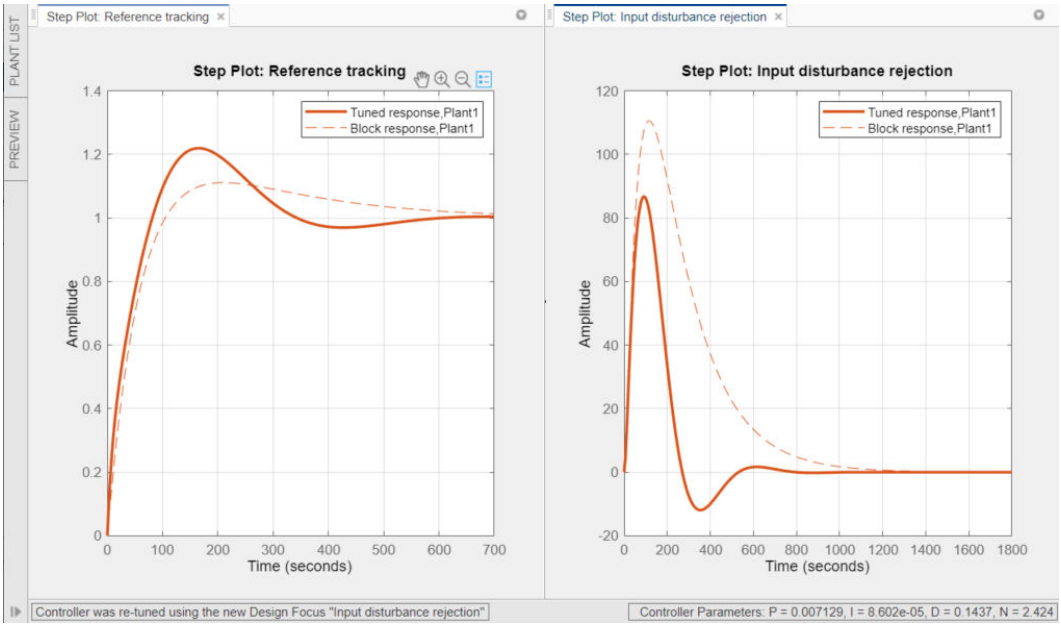


PID Tuner automatically retunes the controller coefficients with a focus on reference-tracking performance.



The responses with the balanced controller are now displayed as the `Block response`, and the controller tuned with a focus reference-tracking is the `Tuned response`. The plots show that the resulting controller tracks the reference input with considerably less overshoot and a faster settling time than the balanced controller design. However, the design yields much poorer disturbance rejection.

Finally, change the design focus to favor disturbance rejection. In the **Options** dialog box, in the **Focus** menu, select `Input disturbance rejection`.



This controller design yields improved disturbance rejection, but results in some increased overshoot in the reference-tracking response.

When you use design focus option, you can still adjust the **Transient Behavior** slider for further fine-tuning of the balance between these two measures of performance. Use the design focus and the sliders together to achieve the performance balance that best meets your design requirements. The effect of this fine-tuning on system performance depends strongly on the properties of your plant. For some plants, moving the **Transient Behavior** slider or changing the **Focus** option has little or no effect.

To obtain independent control over reference tracking and disturbance rejection, you can use a two-degree-of-freedom controller, PID Controller (2DOF), instead of a single degree-of-freedom controller.

See Also

More About

- “PID Tuning Algorithm” on page 7-3
- “Analyze Design in PID Tuner” on page 7-8
- “Verify the PID Design in Your Simulink Model” on page 7-13
- “Design Two-Degree-of-Freedom PID Controllers” on page 7-26

Design Two-Degree-of-Freedom PID Controllers

Using **PID Tuner**, you can tune two-degree-of-freedom PID Controller (2DOF) and Discrete 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) and Discrete PID Controller (2DOF) blocks, the setpoint weights `b` and `c` determine the strength of the proportional and derivative action in the feedforward compensator. For more information, see the PID Controller (2DOF) and Discrete PID Controller (2DOF) block reference pages.

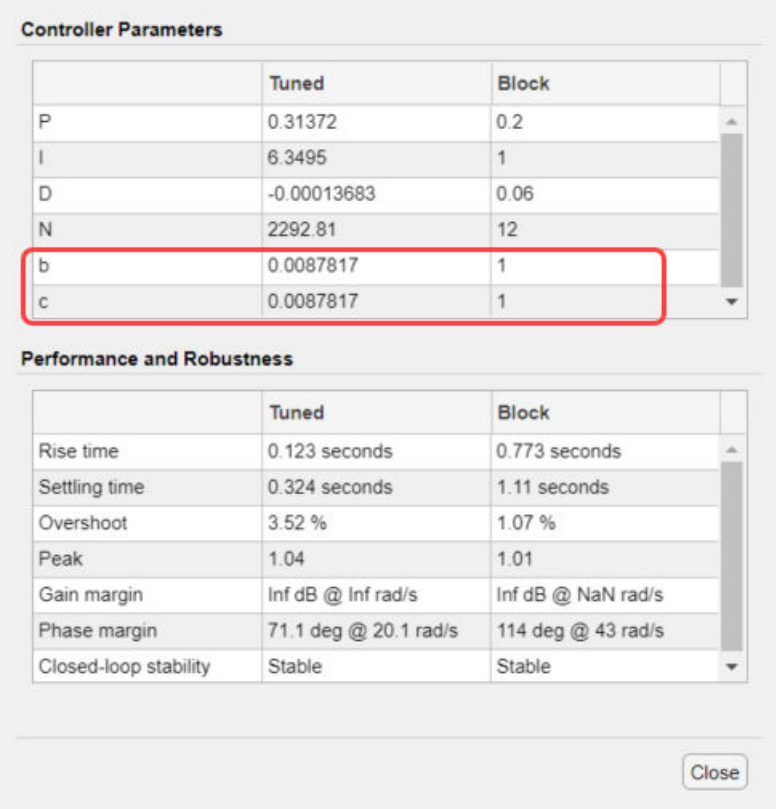
Tuning Two-Degree-of-Freedom PID Controllers

PID Tuner tunes the PID gains `P`, `I`, `D`, and `N`. For the PID Controller (2DOF) blocks, the tuner also automatically tunes the setpoint weights `b` and `c`. You can use the same techniques to refine and analyze the design that you use for tuning one-degree-of-freedom PID controllers.

To tune a 2-DOF PID controller block in a Simulink model:

1. Double-click the block. In the block parameters dialog box, click **Tune**.

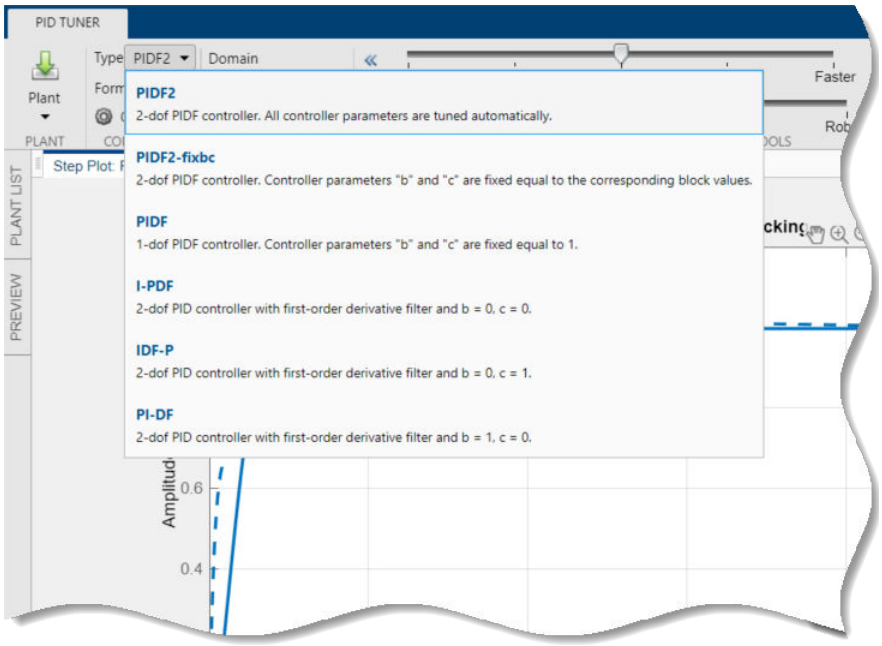
   PID Tuner opens, linearizes the model at the model initial conditions, and automatically computes an initial controller design that balances performance and robustness. In this design, **PID Tuner** adjusts the setpoint weights `b` and `c` if necessary, as well as the PID gains. To see the tuned values of all coefficients, click **Show Parameters**.



2. Analyze and refine the initial design, as described in “Analyze Design in PID Tuner” on page 7-8. All the same response plots, design adjustments, and options are available for tuning 2-DOF PID controllers as in the single-degree-of-freedom case.
3. Verify the controller design, as described in “Verify the PID Design in Your Simulink Model” on page 7-13.

Fixed-Weight Controller Types

When you tune a PID Controller (2DOF) block in **PID Tuner**, the **Type** menu shows additional options for specifying the controller type. These options include controllers with fixed setpoint weights, such as the controllers described in “Specify PI-D and I-PD Controllers” on page 7-33.



The availability of some type options depends on the **Controller** setting in the PID Controller (2DOF) block dialog box.

Type	Description	Controller Setting in Block
`PIDF2`	2-DOF PID controller with filter on derivative term. **PID Tuner** tunes all controller parameters, including setpoint weights.	`PID`
`PIDF2-fixbc`	2-DOF PID controller with filter on derivative term. **PID Tuner** fixes setpoint weights at the values specified in the controller block.	`PID`
`PIDF`	2-DOF controller with action equivalent to a 1-DOF PIDF controller, with fixed `b = 1` and `c = 1`.	`PID`
`I-PDF`	2-DOF PID controller with filter on derivative term, with fixed `b = 0` and `c = 0`.	`PID`
`IDF-P`	2-DOF PID controller with filter on derivative term, with fixed `b = 0` and `c = 1`.	`PID`
`PI-DF`	2-DOF PID controller with filter on derivative term, with fixed `b = 1` and `c = 0`.	`PID`
`PI2`	2-DOF PI controller. **PID Tuner** tunes all controller parameters, including setpoint weight on proportional term, `b`.	`PI`

Type	Description	Controller Setting in Block
`PI2-fixbc`	2-DOF PI controller with filter on derivative term. **PID Tuner** fixes setpoint weight `b` at the value specified in the controller block.	`PI`
`PI`	2-DOF controller with action equivalent to a 1-DOF PI controller, with fixed `b = 1`.	`PI`
`PDF2`	2-DOF PD controller with filter on derivative term (no integrator). **PID Tuner** tunes all controller parameters, including setpoint weights.	`PD`
`PDF2-fixbc`	2-DOF PD controller with filter on derivative term. **PID Tuner** fixes setpoint weights at the values specified in the controller block.	`PD`
`PD`	2-DOF controller with action equivalent to a 1-DOF PD controller, with fixed `b = 1` and `c = 1`.	`PD`

See Also

More About

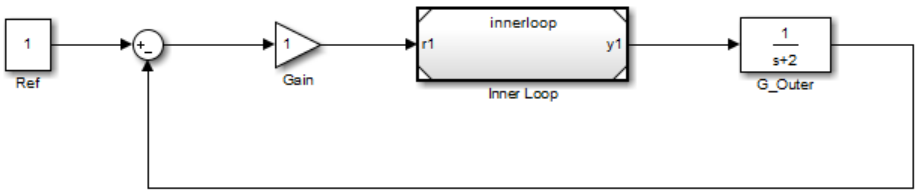
- “Analyze Design in PID Tuner” on page 7-8
- “Verify the PID Design in Your Simulink Model” on page 7-13
- “Specify PI-D and I-PD Controllers” on page 7-33

Tune PID Controller Within Model Reference

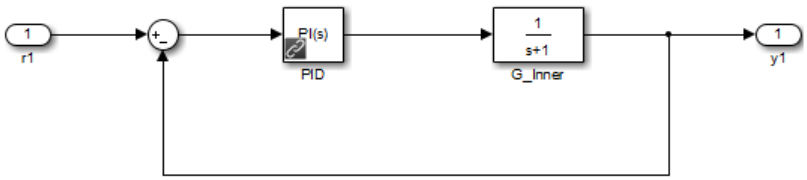
In Simulink, you can include one model inside another using model referencing (see “Model Reference Basics”). When using **PID Tuner** or **Frequency Response Based PID Tuner** to tune a PID controller block in a referenced model, there are some constraints to be aware of.

In general, you can tune a PID controller block in a referenced model using either **PID Tuner** or **Frequency Response Based PID Tuner**. When you open either tuner, the software prompts you to specify which model to use as the top-level model for linearization and tuning (**PID Tuner**) or estimation and tuning (**Frequency Response Based PID Tuner**). For example, consider the model `model_ref_pid`.

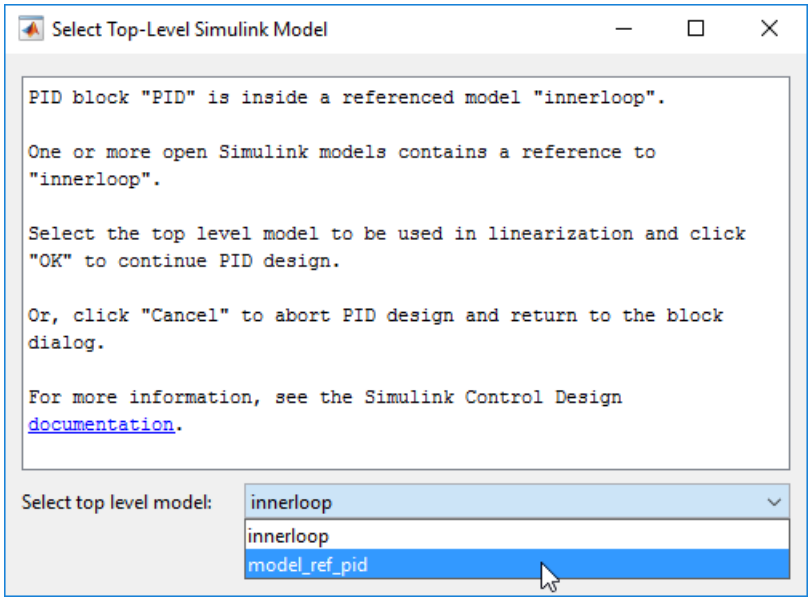
```
open('model_ref_pid');
```



The block `Inner Loop` is a referenced model that contains the controller block to tune. Open the referenced model.



`Inner Loop` contains a PID controller block, `PID`. Open that block. In the **Select Tuning Method** drop-down list, select `Transfer Function Based (PID Tuner App)`, and click **Tune** to open **PID Tuner**. The software prompts you to select which open model is the top-level model for linearization and tuning. (Selecting `Frequency Response Based` to open **Frequency Response Based PID Tuner** results in a similar prompt.)



The available choices for top-level model include the referenced model itself, plus any open model in which the referenced model:

- Appears exactly once, and
- Is configured for normal simulation mode.

The tuning tools do not detect models that contain the model reference but are not open.

Selecting `innerloop` causes the tuner to disregard `model_ref_pid`. Instead, the tuner tunes the PID Controller block for the plant `G_Inner` alone, as if there were no outer loop.

Alternatively, you can select `model_ref_pid` as the top-level model. When you do so, the tuner considers the dynamics of both the inner and outer loops, and tunes with both loops closed. In this case, PID controller sees the effective plant `(1+G_Outer*Gain)*G_Inner`.

Select the desired top-level model, and click **OK**. The tuner you selected with the **Select Tuning Method** opens for tuning the specified top-level model.

Models with Multiple Instances of the Referenced Model

Sometimes, tuning can proceed when the referenced model appears multiple times in an open model. If the following conditions are met, you can tune the PID controller block, using the referenced model as the top-level model:

- The only open models that contain the model reference have multiple instances of it, and
- At least one of these instances is in normal mode.

When this condition occurs, the software issues a warning. In this case, because the tuner can only tune with respect to the referenced model, you cannot specify a top-level model.

Referenced Model in Accelerated or Other Simulation Modes

If there is no normal mode instance of the referenced model in any open model, tuning cannot proceed. In this case, the software issues an error. To tune the PID controller block, convert some instance of the referenced model in an open model to normal simulation mode.

See Also

More About

- “Model Reference Basics”
- “Choosing a Simulation Mode”
- “Introduction to Model-Based PID Tuning in Simulink” on page 7-2

Specify PI-D and I-PD Controllers

PI-D and I-PD controllers are used to mitigate the influence of changes in the reference signal on the control signal. These controllers are variants of the 2DOF PID controller.

The general formula of a parallel-form 2DOF PID controller is:

$$u = P(br - y) + I\frac{1}{s}(r - y) + D\frac{N}{1 + N\frac{1}{s}}(cr - y)\,.$$

Here, r and y are the reference input and measured output, respectively. u is the controller output, also called the control signal. P, I, and D specify the proportional, integral, and derivative gains, respectively. N specifies the derivative filter coefficient. b and c specify setpoint weights for the proportional and derivative components, respectively. For a 1DOF PID controller, b and c are equal to 1.

If r is nonsmooth or discontinuous, the derivative and proportional components can contribute large spikes or offsets in u, which can be infeasible. For example, a step input can lead to a large spike in u because of the derivative component. For a motor actuator, such an aggressive control signal could damage the motor.

To mitigate the influence of r on u, set b or c, or both, to 0. Use one of the following setpoint-weight-based forms:

- PI-D ($b = 1$ and $c = 0$) — Derivative component does not directly propagate changes in r to u, whereas the proportional component does. However, the derivative component, which has a greater impact, is suppressed. Also referred to as the derivative of output controller.

  The general formula for this controller form is:

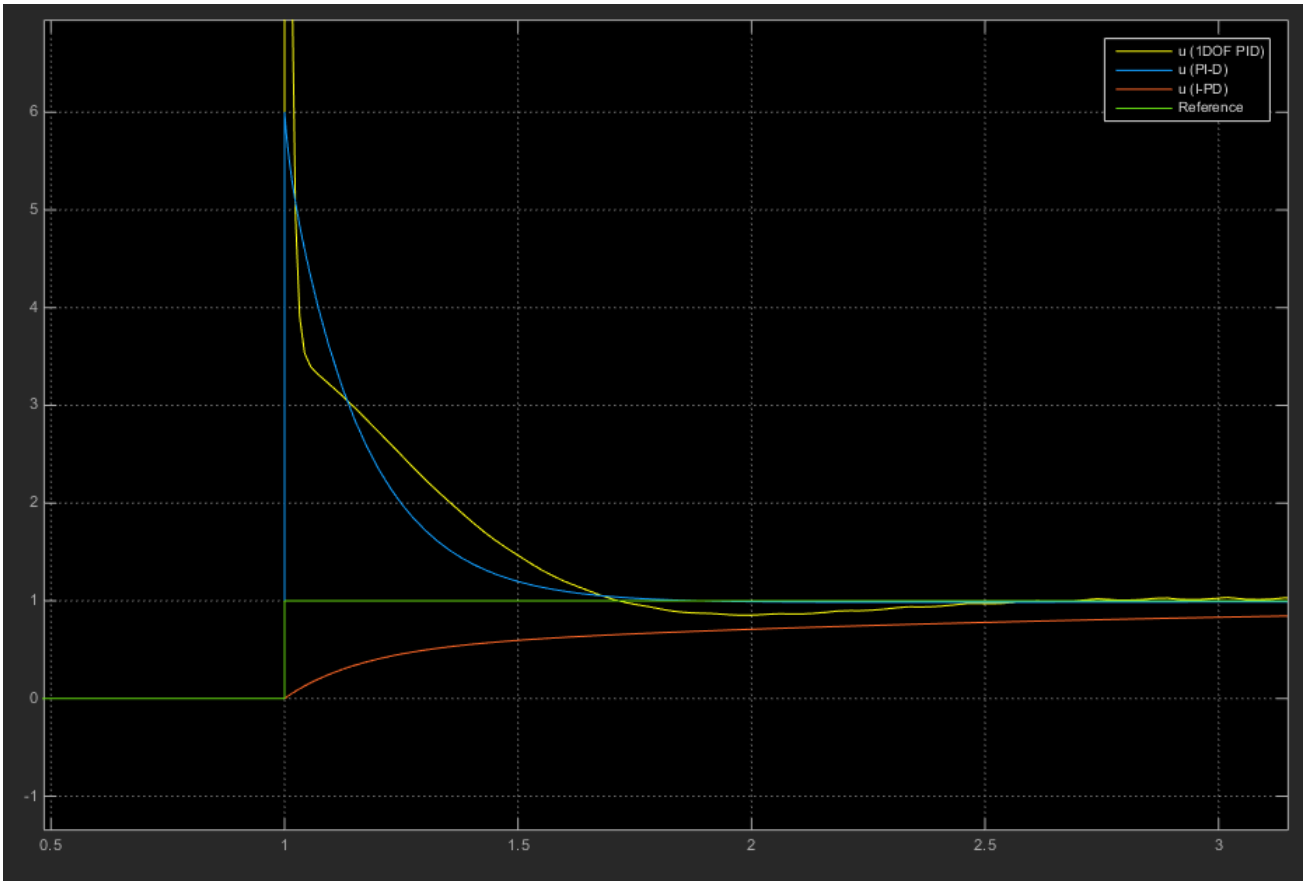
  $$u = P(r - y) + I\frac{1}{s}(r - y) - D\frac{N}{1 + N\frac{1}{s}}y\,.$$

- I-PD ($b = 0$ and $c = 0$) — Proportional and derivative components do not directly propagate changes in r to u.

  The general formula for this controller form is:

  $$u = -Py + I\frac{1}{s}(r - y) - D\frac{N}{1 + N\frac{1}{s}}y\,.$$

The following plot shows u for different PID forms for a step reference. The 1DOF PID controller results in a large spike when the reference changes from 0 to 1. The PI-D form results in a smaller jump. In contrast, the I-PD form does not react as much to the change in r.



You can tune the *P*, *I*, *D*, and *N* coefficients of a PI-D or I-PD controller to achieve the desired disturbance rejection and reference tracking.

Specify PI-D and I-PD Controllers Using PID Controller (2DOF) Block

To specify a PI-D or I-PD Controller using the PID Controller (2DOF) or Discrete PID Controller (2DOF) blocks, open the block dialog. In the **Controller** menu, select `PID`.

- For a PI-D controller, enter `1` in the **Setpoint weight (b)** box, and `0` in the **Setpoint weight (c)** box.
- For an I-PD controller, enter `0` in the **Setpoint weight (b)** box, and `0` in the **Setpoint weight (c)** box.

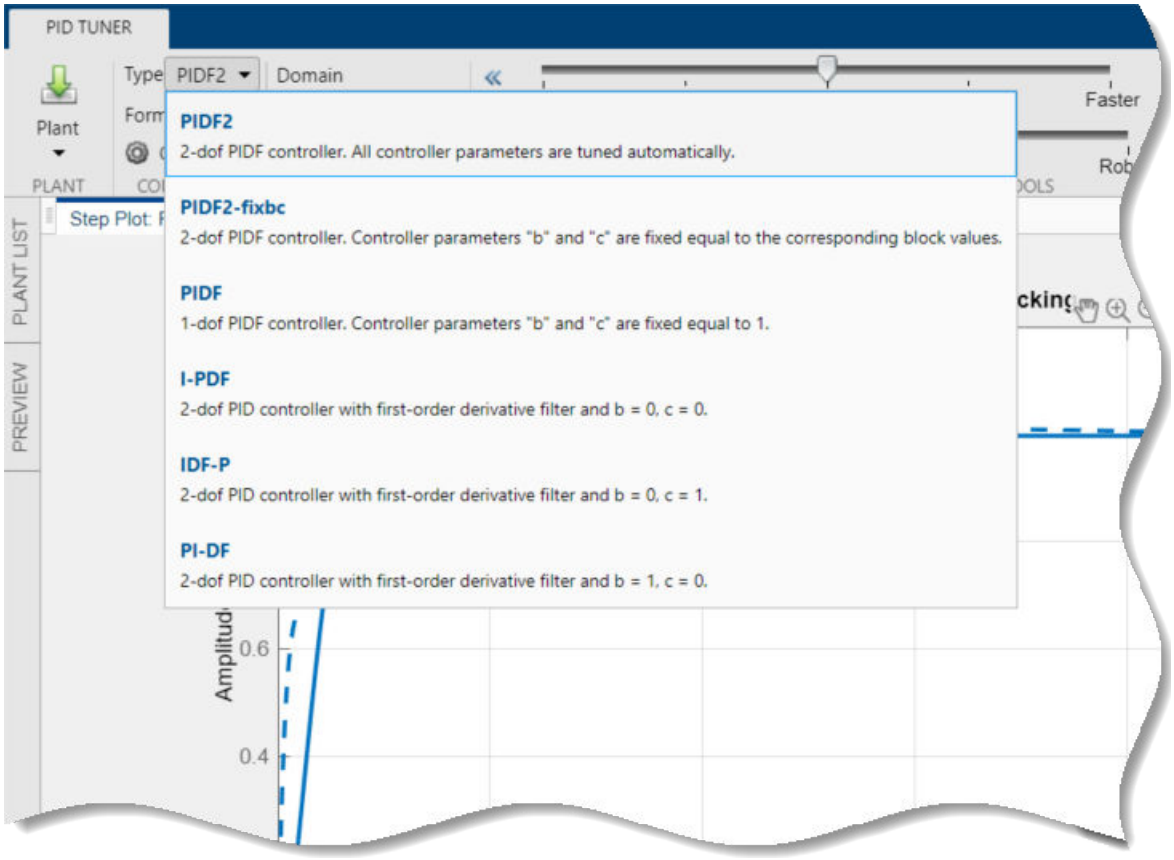
For an example that demonstrates the PI-D and I-PD controller forms, type `ex_scd_pid2dof_setpoint_based_controllers`. This opens a model that compares the performance of a 1DOF PID, a PI-D, and an I-PD controller.

Automatic Tuning of PI-D and I-PD Controllers

You can use **PID Tuner** to automatically tune PI-D and I-PD controllers while preserving the fixed *b* and *c* values. To do so:

1. In the model, open the block. In the block dialog box, in the **Controller** menu, select `PID`.

2. Click **Tune**. **PID Tuner** opens.
3. In **PID Tuner**, in the **Type** menu, select `PI-DF` or `I-PDF`. **PID Tuner** retunes the controller gains, fixing $b = 1$ and $c = 0$ for PI-D, and $b = 0$ and $c = 0$ for I-PD.



You can now analyze system responses as described in “Analyze Design in PID Tuner” on page 7-8.

See Also

PID Controller (2DOF) | Discrete PID Controller (2DOF)

More About

- “Tune PID Controller to Favor Reference Tracking or Disturbance Rejection” on page 7-17
- “Design Two-Degree-of-Freedom PID Controllers” on page 7-26

Design PID Controller from Plant Frequency-Response Data

Most Simulink Control Design PID tuning tools design PID gains based on a linearized plant model. When your plant model does not linearize or linearizes to zero, one option is to design a PID controller based on simulated frequency-response data. Simulink Control Design gives you several ways to do so.

Use Frequency Response Based PID Tuner

Use **Frequency Response Based PID Tuner** to design a PID controller using estimated plant frequency responses near the target open-loop bandwidth. Advantages of this approach include:

- **Frequency Response Based PID Tuner** works even if disturbances are present in the plant model.
- You can configure the estimation and tuning in one dialog box, making tuning less complex than using `frestimate` or **Model Linearizer** to estimate the frequency response.

For more information about using **Frequency Response Based PID Tuner**, see “Frequency-Response Based Tuning” on page 7-37.

Use frestimate or Model Linearizer

Use the `frestimate` command or the frequency-response estimation workflow in **Model Linearizer** to estimate the plant frequency response over a range of frequencies that you specify. This approach results in a frequency-response data (`frd`) model object that you then import into **PID Tuner**. Advantages of this approach include:

- You do not have to specify a control bandwidth ahead of time. **PID Tuner** chooses an initial control bandwidth, which you can adjust to achieve the desired balance between performance and robustness.
- You can use the interactive tuning and analysis tools of **PID Tuner** to examine the estimated linear response of the tuned system in the frequency domain. Also, you can use the `frd` model of the plant for other analysis tasks.
- Depending on the particulars of your model, this approach can be faster, because **Frequency Response Based PID Tuner** simulates your model twice.

For more information, see:

- “Design PID Controller Using Estimated Frequency Response” on page 7-124
- “Estimate Frequency Response Using Model Linearizer” on page 5-6

See Also

More About

- “Introduction to Model-Based PID Tuning in Simulink” on page 7-2
- “Frequency-Response Based Tuning” on page 7-37
- “Design PID Controller Using Estimated Frequency Response” on page 7-124

Frequency-Response Based Tuning

Frequency Response Based PID Tuner simulates the model to estimate the plant frequency responses at a few frequencies near the control bandwidth. It then uses the estimated frequency response to tune the gains in your PID Controller. This tuner is a useful alternative when **PID Tuner** cannot linearize the plant at the operating point you want to use for tuning.

Frequency Response Based PID Tuner can tune the **P**, **I**, **D**, and **N** parameters in PID Controller and PID Controller (2DOF) blocks in both continuous time and discrete time. For PID Controller (2DOF) blocks, the tuner does not tune the setpoint weights **b** and **c**.

How Frequency Response Based PID Tuner Works

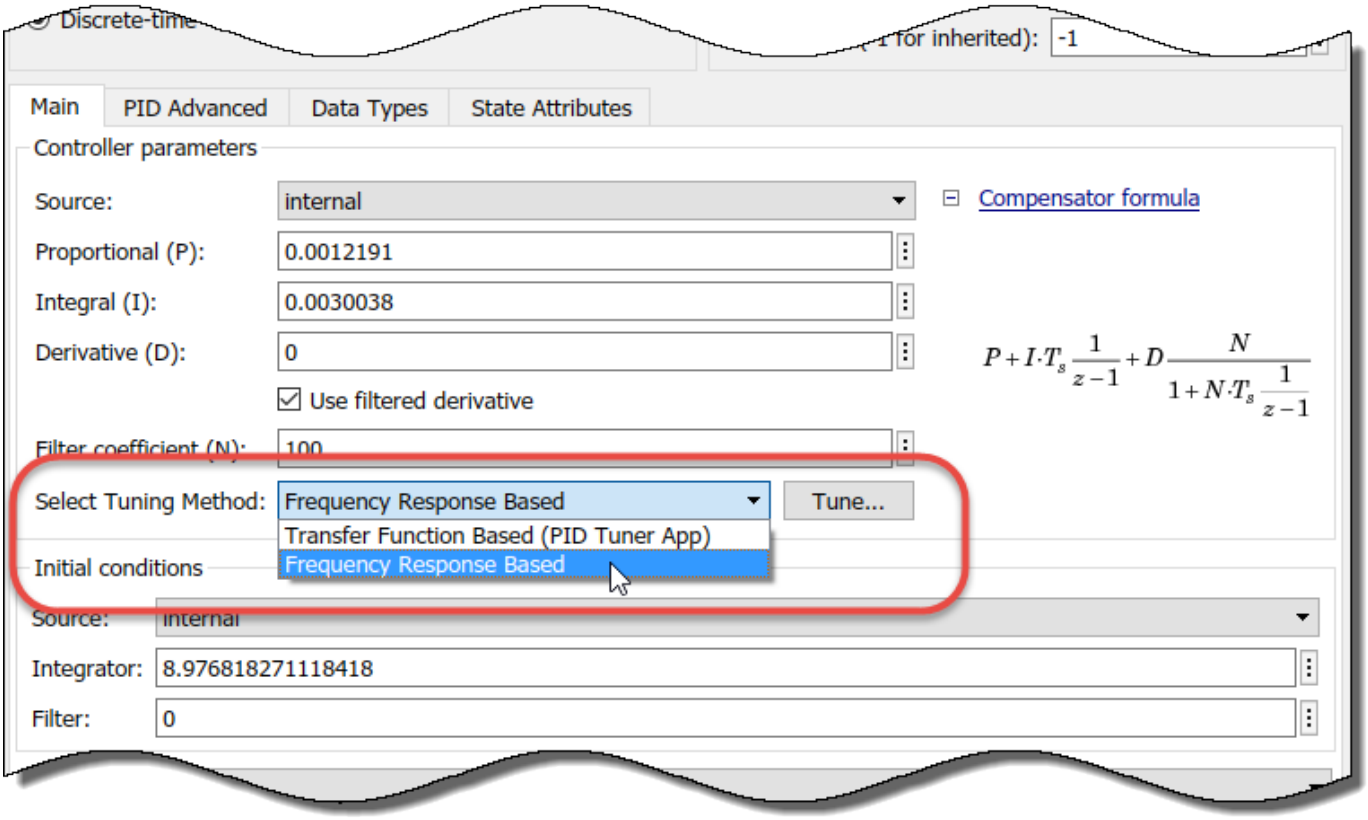
Like the interactive **PID Tuner**, the **Frequency Response Based PID Tuner** considers the plant to be all blocks in the loop between the PID Controller block output and input. The **Frequency Response Based PID Tuner** performs a perturbation experiment to estimate the open-loop frequency response of the plant. To do so, the tuner performs the following steps:

1. Breaks the feedback loop at the controller output and simulates the model, applying perturbation signals to the plant. The perturbations include sinusoidal signals at frequencies $[1/3,1,3,10]\omega_c$, where ω_c is the target bandwidth you specify for tuning. If the plant is asymptotically stable, the applied signal also includes a step perturbation.
2. Measures the response to the perturbation at the controller input.
3. Uses the resulting data to estimate the plant frequency response at the four frequencies. For asymptotically stable plants, the tuner also uses the response to the step perturbation to estimate the plant DC gain.
4. Uses the estimated frequency response to compute PID gains that balance performance and robustness.

If your model includes disturbances, the tuner can run two simulations: a simulation without perturbation to get a baseline response, and a simulation with the perturbations applied to the plant. The tuner then uses the difference between the two responses to remove the effects of disturbances in the model. In this case, the estimated frequency response used for tuning is based on this disturbance-free response.

Open Frequency Response Based PID Tuner

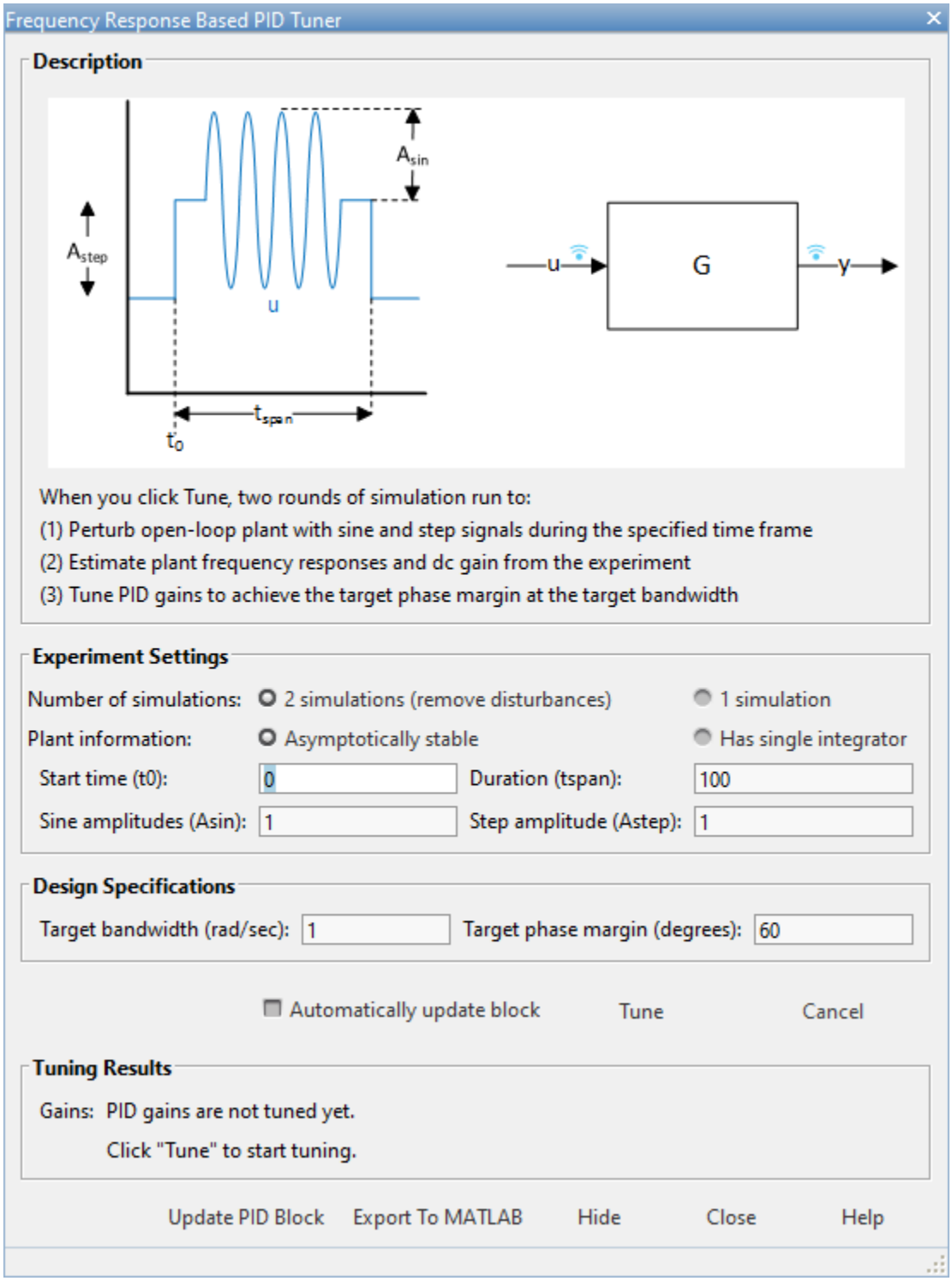
To open the **Frequency Response Based PID Tuner**, in the PID Controller block dialog box, in the **Select Tuning Method** drop-down list, select `Frequency Response Based`.



Click **Tune**. The **Frequency Response Based PID Tuner** opens. The tuner reads some parameters from the PID Controller block. These parameters include:

- Controller type (such as PI, PD, or PID)
- Controller form (parallel or ideal)
- Controller time domain (continuous-time or discrete-time)
- Controller sample time

In the **Frequency Response Based PID Tuner**, You configure the settings for the estimation experiment and the tuning goals.



Configure Experiment Settings

In the **Experiment Settings** section, you specify parameters that control the frequency-response estimation experiment. For more details about these settings, click **Help**.

1. Specify whether to run two simulations (default) or one. If your model includes disturbances that can affect the result of the frequency-response estimation experiment, select **2 simulations (remove disturbances)**. With this option selected, the tuner runs a baseline simulation and subtracts the resulting frequency response from the perturbed simulation to remove the effects of disturbances. If your model does not include any such disturbances, skip the baseline simulation by selecting **1 simulation**.
2. Specify whether the plant is asymptotically stable or has a single integrator. If the plant is asymptotically stable, the estimation experiment includes an estimation of the plant DC gain. The **Frequency Response Based PID Tuner** performs this estimation by injecting a step signal into the plant.

   Caution Do not use the **Frequency Response Based PID Tuner** with an unstable plant or a plant containing multiple integrators.
3. Specify the start time of the experiment in the **Start time (t0)** field. Start the experiment when the plant is at the desired equilibrium operating point. For instance, if you know that your simulation must run to 10 s for the plant to reach such an operating point, specify a start time of 10.
4. Specify the experiment duration in the **Duration (tspan)** field. Let the experiment run long enough for the frequency-response estimation algorithm to collect sufficient data for a good estimate at all frequencies it probes. A conservative estimate for the experiment duration is $100/\omega_c$, where ω_c is the target bandwidth for tuning that you specify.
5. Specify the perturbation amplitudes. During the tuning experiment, the **Frequency Response Based PID Tuner** injects a sinusoidal signal into the plant at four frequencies, $[1/3,1,3,10]\omega_c$. Use the **Sine amplitudes (Asin)** field to specify the amplitudes of these injected signals. You can provide a scalar value to inject the same amplitude at each frequency, or a vector of length 4 to specify different amplitudes for each.

   In a typical plant with typical target bandwidth, the magnitudes of the plant responses at the experiment frequencies do not vary widely. In such cases, you can use a scalar value to apply the same magnitude perturbation at all frequencies. However, if you know that the response decays sharply over the frequency range, consider decreasing the amplitude of the lower-frequency inputs and increasing the amplitude of the higher-frequency inputs. It is numerically better for the estimation experiment when all the plant responses have comparable magnitudes.

   The perturbation amplitudes must be:

   - Large enough that the perturbation overcomes any deadband in the plant actuator and generates a response above the noise level
   - Small enough to keep the plant running within the approximately linear region near the nominal operating point, and to avoid saturating the plant input or output

   In the experiment, the sinusoidal signals are superimposed (with the step perturbation, if any, in the case of open-loop tuning). Thus, the perturbation can be at least as large as the sum of all amplitudes. Therefore, to obtain appropriate values for the amplitudes, consider:

   - Actuator limits. Make sure that the largest possible perturbation is within the range of your plant actuator. Saturating the actuator can introduce errors into the estimated frequency response.
   - How much the plant response changes in response to a given actuator input at the nominal operating point for tuning. For instance, suppose that you are tuning a PID controller used in

engine-speed control. You have determined that at frequencies around the target bandwidth, a 1° change in throttle angle causes a change of about 200 rpm in the engine speed. Suppose further that to preserve linear performance the speed must not deviate by more than 100 rpm from the nominal operating point. In this case, choose amplitudes to ensure that the perturbation signal is no greater than 0.5 (assuming that value is within actuator limits).

If your plant is asymptotically stable, specify amplitude of the step perturbation in the **Step amplitudes (Astep)** field. The considerations for choosing a step amplitude are the same as the considerations for specifying the step amplitudes.

Configure Design Goals

In the **Design Specifications** section of the dialog box, you specify your goals for PID tuning.

Specify the target bandwidth in the **Target bandwidth (rad/sec)** field. The target bandwidth is the target value for the 0-dB gain crossover frequency of the tuned open-loop response CP, where P is the plant response, and C is the controller response. This crossover frequency roughly sets the control bandwidth. For a desired rise-time τ, a good guess for the target bandwidth is $2/\tau$.

In the **Target phase margin (degrees)** field, specify a target minimum phase margin for the tuned open-loop response at the crossover frequency. The target phase margin reflects desired robustness of the tuned system. Typically, choose a value in the range of about 45°– 60°. In general, higher phase margin improves overshoot, but can limit response speed. The default value, 60°, tends to balance performance and robustness, yielding about 5-10% overshoot, depending on the characteristics of your plant.

For more details about these settings, click **Help**.

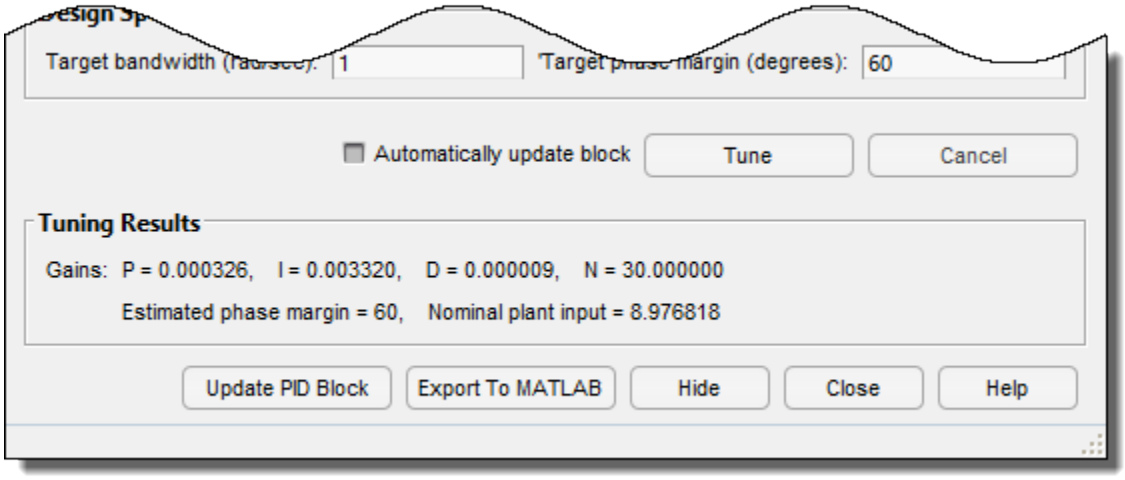
Tune and Validate Controller Gains

Click **Tune** to initiate the frequency-response estimation experiment. While the estimation experiment is running, the tuner:

- Closes the open PID Controller block.
- Clears any previous tuning results displayed in the tuner dialog box.
- Replaces the PID Controller block in your model with an unnamed subsystem.

Note When the estimation experiment is completed or canceled, the tuner restores the PID Controller block. This process might result in some displacement of signal wires on the model canvas, and puts your Simulink model in a state with unsaved changes.

When the estimation experiment ends, the tuner computes new PID gains and displays them in the **Tuning Results** section of the dialog box. (For more information about the tuning results, click **Help**.)



If **Automatically update block** is selected, the **Frequency Response Based PID Tuner** writes the new PID gains to the PID Controller block when tuning is completed. Otherwise, click **Update PID Block** to write the tuned gains to the block. Simulate the model to validate the tuned gains against your full nonlinear system.

For an example illustrating the use of the **Frequency Response Based PID Tuner** to tune a PID Controller block in a Simulink model that does not linearize, see “Design PID Controller Using Plant Frequency Response Near Bandwidth” on page 7-43.

See Also

More About

- “Introduction to Model-Based PID Tuning in Simulink” on page 7-2
- “Design PID Controller Using Plant Frequency Response Near Bandwidth” on page 7-43
- “Design PID Controller Using Estimated Frequency Response” on page 7-124

Design PID Controller Using Plant Frequency Response Near Bandwidth

This example shows one of several ways to tune a PID controller for plants that cannot be linearized. In this example, you use the Frequency Response Based PID Tuner to automatically characterize the frequency response of a buck converter around the control bandwidth and then tune the PID controller.

Buck Converter Model

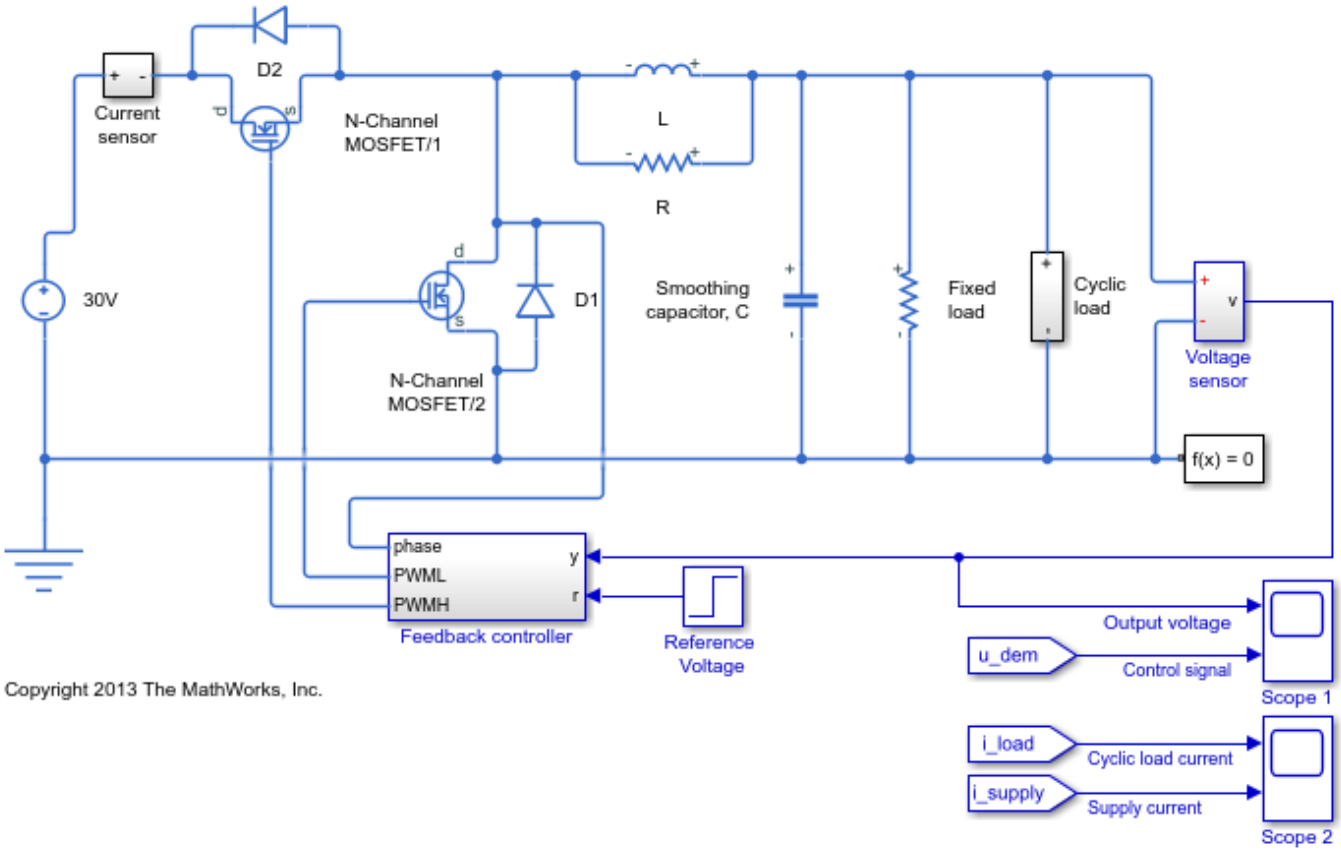
Buck converters convert DC to DC. The model in this example uses a switching power supply to convert a 30V DC supply into a regulated DC supply. The converter is modeled using MOSFETs rather than ideal switches to ensure that device on-resistances are correctly represented. The converter response from reference voltage to measured voltage includes the MOSFET switches. Traditional PID design requires a linear model of the system from the "reference voltage" (controller output) to measured voltage. However, because of the switches in this model, automated linearization results in a zero system. When a model linearizes to zero, several alternatives are available.

- **Relinearize the system**. Linearize the model at a different operating point or simulation snapshot time.
- **Identify a new plant**. Use measured or simulated data to identify a plant model (requires System Identification Toolbox™ software).
- **Frequency response based tuning**. Use simulated data to obtain the frequency response for the plant.

For this example, use the **Frequency Response Based PID Tuner** to estimate the frequency responses of the system and tune the PID controller. For an example that uses system identification to identify a plant model, see “Design PID Controller Using Simulated I/O Data” on page 7-108.

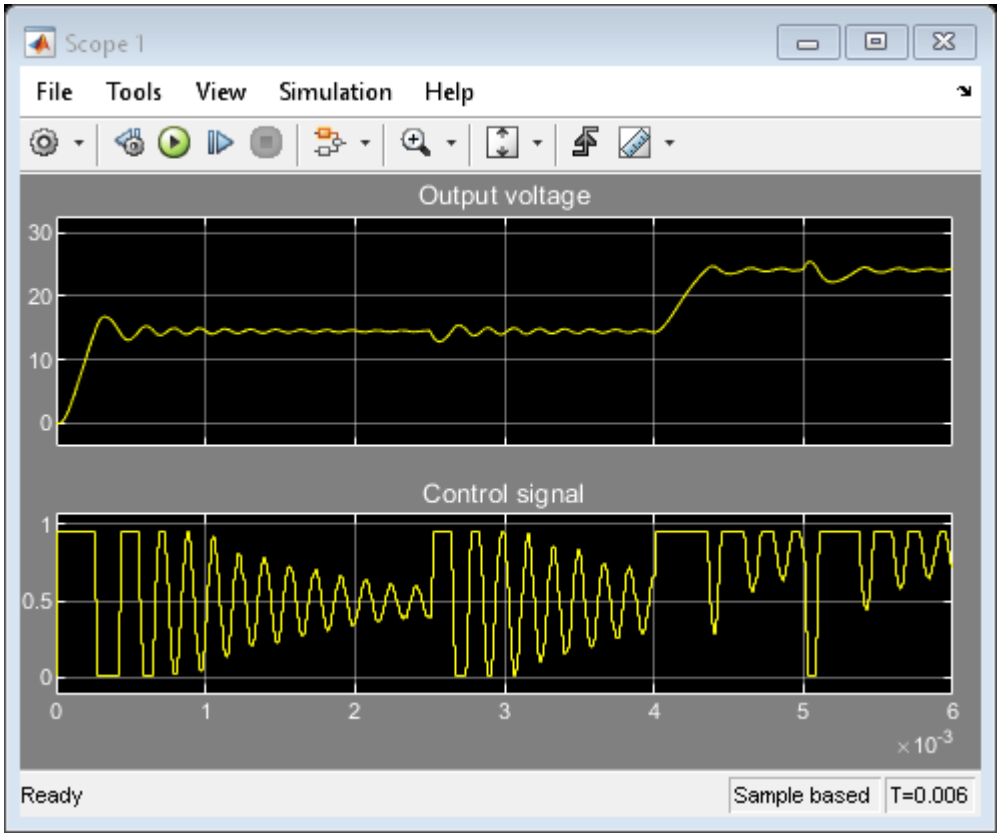
For more information on creating a buck converter model, see “Buck Converter” (Simscape Electrical).

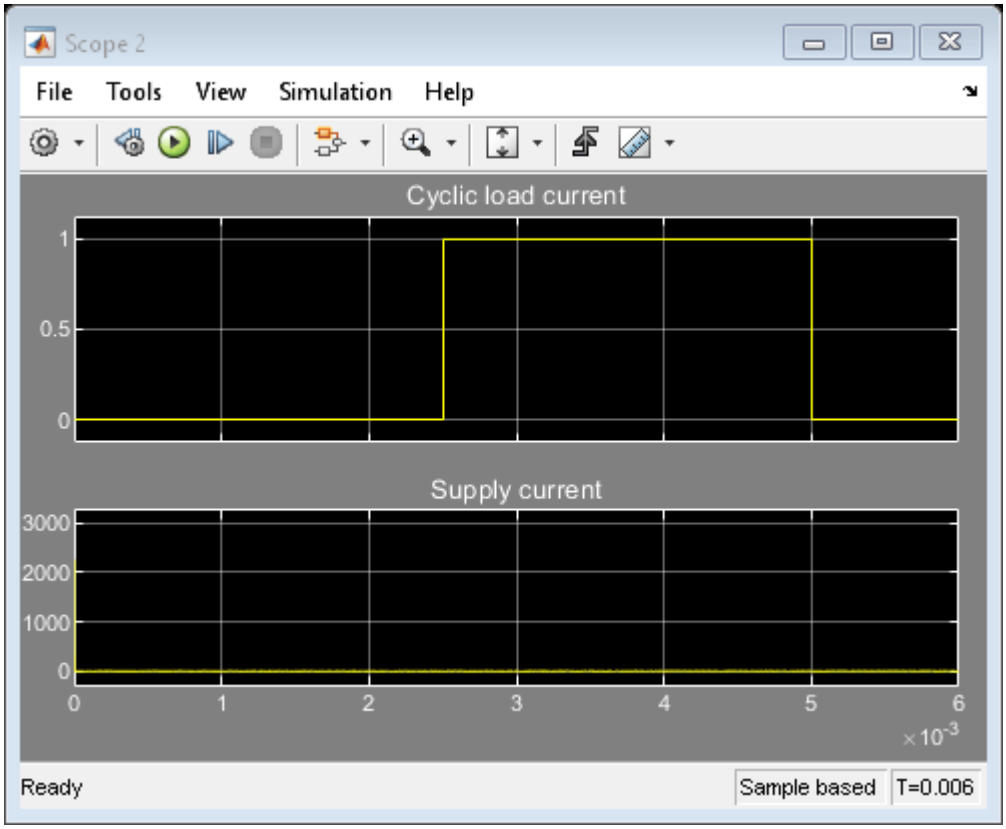
```
open_system('scdbuckconverter')
```



The model uses a reference voltage that switches from 15 to 25 Volts at 0.004 seconds and a load current that is active from 0.0025 to 0.005 seconds. The controller is initialized with default gains that produce overshoot and a slow settling time. Simulating the model shows the underdamping and slow response of the system.

```
sim('scdbuckconverter')
open_system('scdbuckconverter/Scope 1')
open_system('scdbuckconverter/Scope 2')
```


Scope 1
File
Tools
View
Simulation
Help
Output voltage
30
20
10
0
Control signal
1
0.5
0
0
1
2
3
4
5
6
×10^-3
Ready
Sample based
T=0.006



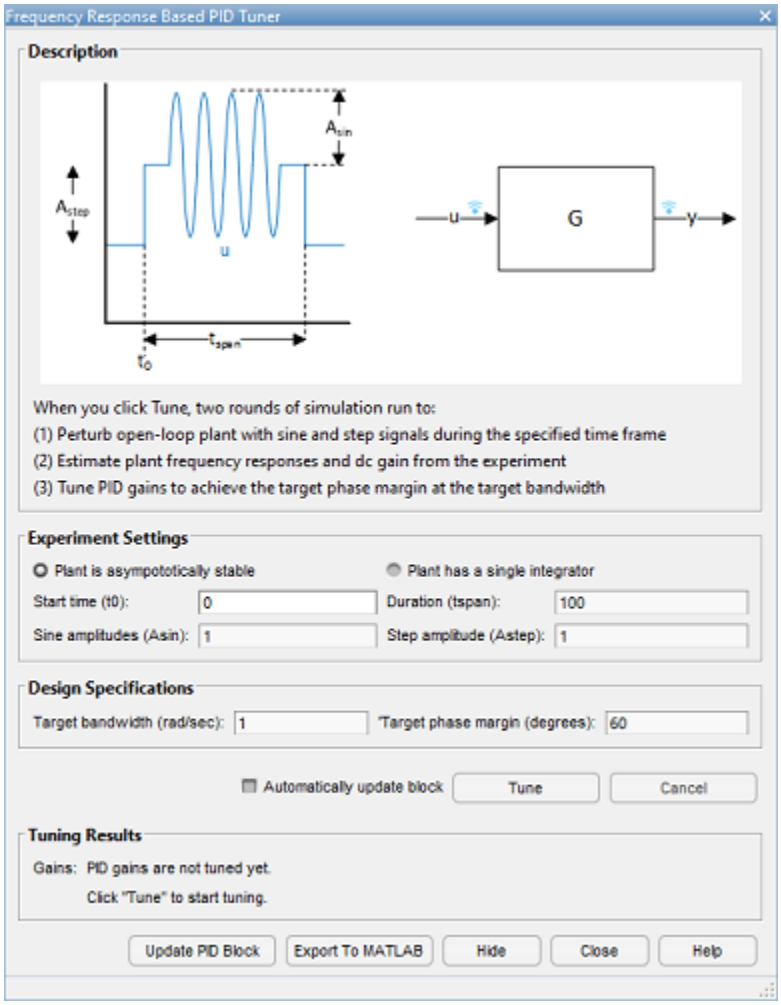
For this example, improve the bandwidth and phase margin of the system to achieve better performance by characterizing the system using frequency response estimation and tuning the PID gains. When tuning the PID controller, consider the following characteristics of the buck converter system.

- No system process or sensor noise
- Controller input is the PWM signal
- PWM signal is limited (saturated) to be between 0 and 1
- Nominal output of the controller at steady-state is 0.5

For buck converter systems, it is desired to have a system with a low rise time and low overshoot. For this example, tune the controller to achieve a rise time of 250e-6 seconds and an overshoot of less than 10%.

Open Frequency Response Based PID Tuner

Open the **Feedback controller** subsystem and then open the **PID Controller** block dialog. In **Select Tuning Method**, select **Frequency Response Based** and click **Tune**. The **Frequency Response Based PID Tuner** opens for the buck converter controller.



The **Frequency Response Based PID Tuner** automatically tunes a PID controller for the plant using two simulations. The first simulation generates a baseline response. The second simulation breaks the loop at the plant input, and perturbs the plant with sine and step signals. The tuner takes the difference between the two simulated responses, which removes the effect of any disturbances in the model. The tuner then uses the resulting data to estimate the plant frequency response. Finally, it uses the estimated frequency response to compute PID gains.

When you open the **Frequency Response Based PID Tuner**, it reads parameters from the PID Controller block to determine the structure of your PID controller. These parameters include:

- PID Controller Type (P, I, PI, PID etc.)
- PID Controller Form (Parallel, Ideal)
- Integrator Method, if applicable (Forward Euler, Trapezoidal etc.)
- Derivative Filter Method, if applicable (Forward Euler, Trapezoidal etc.)
- Sample Time, if applicable

Specify Experiment Settings

Before tuning, specify parameters of the experiment the tuner performs to estimate the frequency response of the plant.

Start time is the time, in seconds, at which the tuner begins applying the perturbation signals to the plant. Choose a start time at which the plant is at the nominal operating point you want to use for tuning. For this example, the buck converter has an initial transient that falls off by 0.002 seconds. Therefore, enter 0.002 for **Start Time**.

Specify the **Duration** of the perturbation experiment. A conservative estimate for the duration of the experiment is 100 divided by the target bandwidth. The target bandwidth is approximately 2/τ, where τ is the desired rise time. For this example, the desired rise time is 250e-6 seconds which results in a target bandwidth of 8000 radians per second. In this example, a conservative estimate for the duration would then be 100/8000 or 0.0125 seconds. Choose 0.0125 seconds for the **Duration**.

During the experiment, the tuner injects sinusoidal signals into the plant at four frequencies, [1/3, 1, 3, 10] ω_c, where ω_c is the target bandwidth you specify for tuning. Specify the amplitudes of the injected sine waves in the **Sine Amplitudes** field.

Choose signal amplitudes which have magnitudes above the noise floor of the system and will not saturate the system. For this example there is no noise in the system to consider. However, the controller output (duty cycle of the PWM) is limited to [0 1] and the nominal output of the controller at steady-state is 0.5. To remain within these limits, specify a sine amplitude of 0.1. Specifying a scalar value uses the same amplitude at all four frequencies.

For an asymptotically stable plant, the tuner also injects a step signal to estimate the plant DC gain. Choose an amplitude for this step signal based on the same considerations you used to choose the sine amplitudes. For this example, enter 0.1 in the **Step Amplitude** field as well.

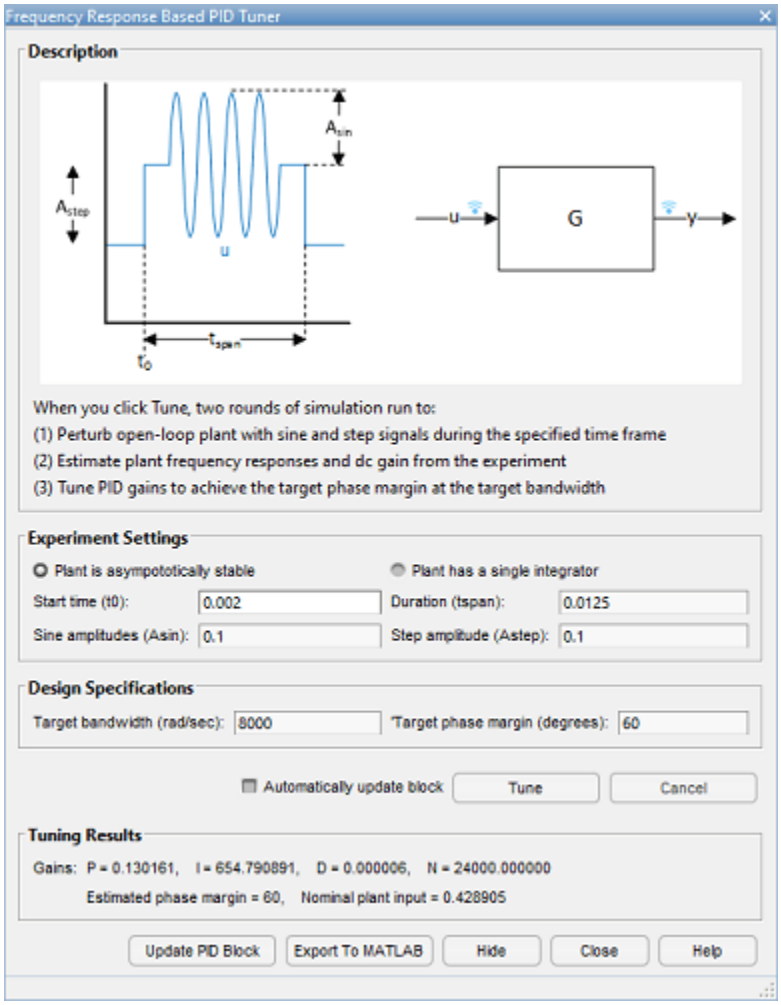
Specify Design Goals

Finally, specify the target bandwidth for tuning. As noted previously, the target bandwidth is 8000 radians per second. Enter 8000 in the **Bandwidth** field. The default target phase margin, 60 degrees, corresponds to an overshoot of about 10% or better.

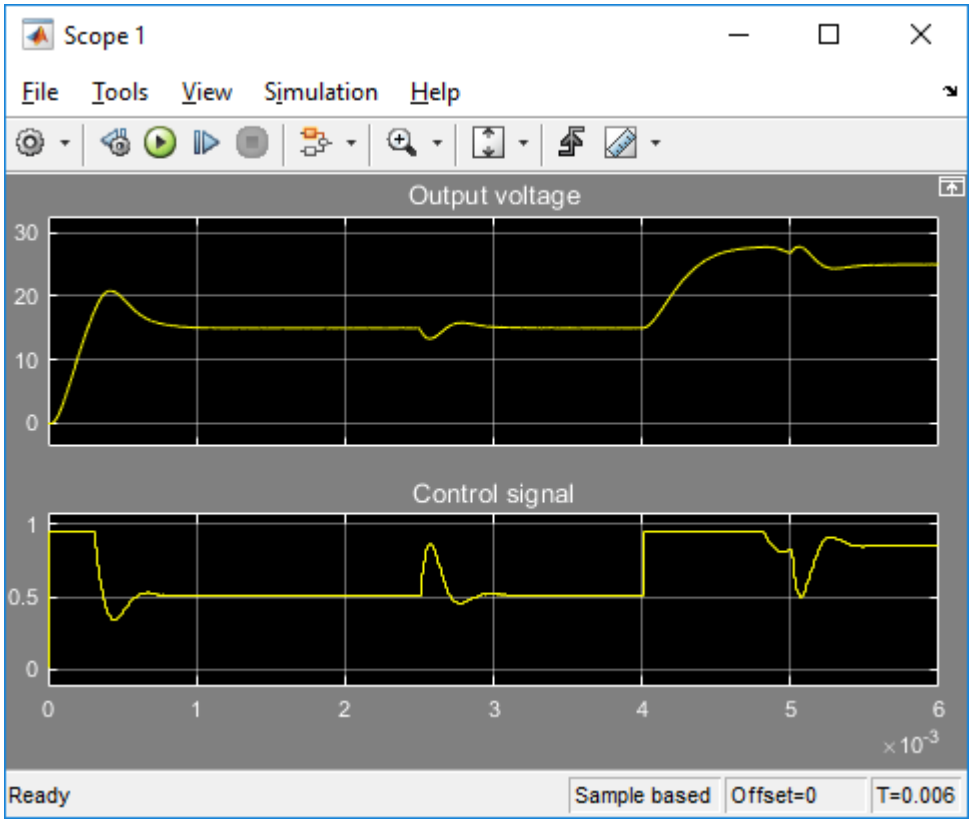
Tune PID Controller and Validate Results

Click **Tune** to begin the two simulations of the buck converter and tune the PID Controller.

At the conclusion of the tuning procedure, the tuned gains, estimated phase margin, and nominal plant input are displayed in the **Frequency Response Based PID Tuner** dialog in the **Tuning Results** section. Check the estimated phase margin to ensure that it is close to the **Target phase margin**.



To verify the results, simulate the model using the tuned PID gains. Click **Update PID Block** to write the tuned gains to the PID Controller block. Then, simulate the model to confirm the PID controller performance.




```
bdclose('scdbuckconverter')
```

See Also

PID Controller | PID Controller (2DOF) | Discrete PID Controller | Discrete PID Controller (2DOF)

More About

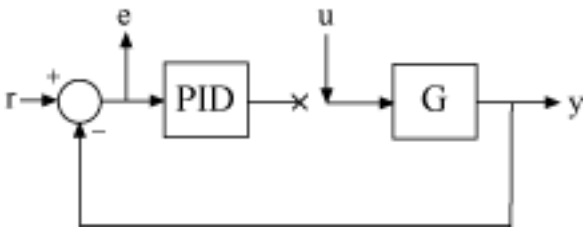
- “Frequency-Response Based Tuning” on page 7-37

Import Measured Response Data for Plant Estimation

This example shows how to use **PID Tuner** to import measured response data for plant estimation.

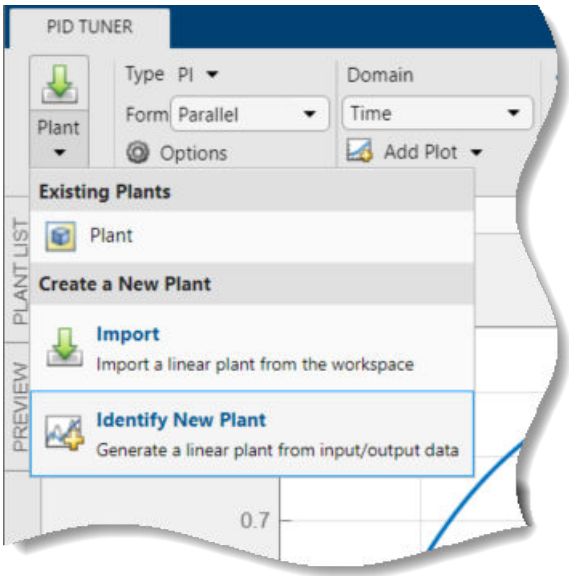
If you have System Identification Toolbox software, you can use **PID Tuner** to estimate the parameters of a linear plant model based on time-domain response data. **PID Tuner** then tunes a PID controller for the resulting estimated model. The response data can be either measured from your real-world system, or obtained by simulating your Simulink model. Plant estimation is especially useful when your Simulink model cannot be linearized or linearizes to zero. For plant identification, you must specify a finite value for the Simulink model stop time.

When you import response data, **PID Tuner** assumes that your measured data represents a plant connected to the PID controller in a negative-feedback loop. In other words, **PID Tuner** assumes the following structure for your system. **PID Tuner** assumes that you injected an input signal at `u` and measured the system response at `y`, as shown.



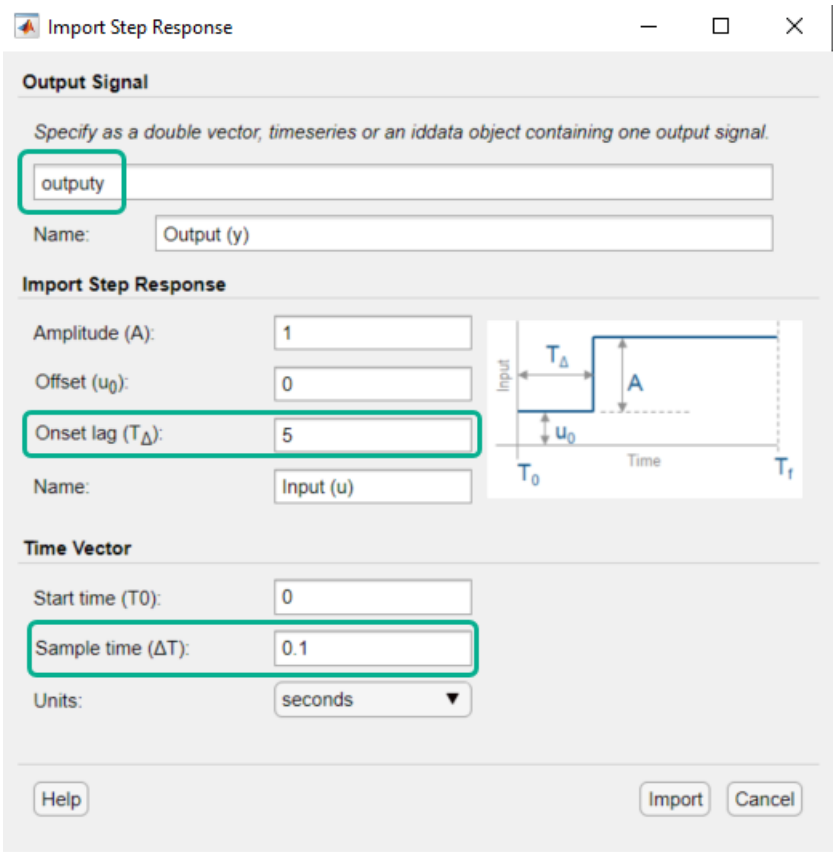
You can import response data stored in the MATLAB workspace as a numeric array, a `timeseries` object, or an `iddata` object. To import response data:

1. In **PID Tuner**, in the **PID Tuner** tab, in the **Plant** menu, select `Identify New Plant`.

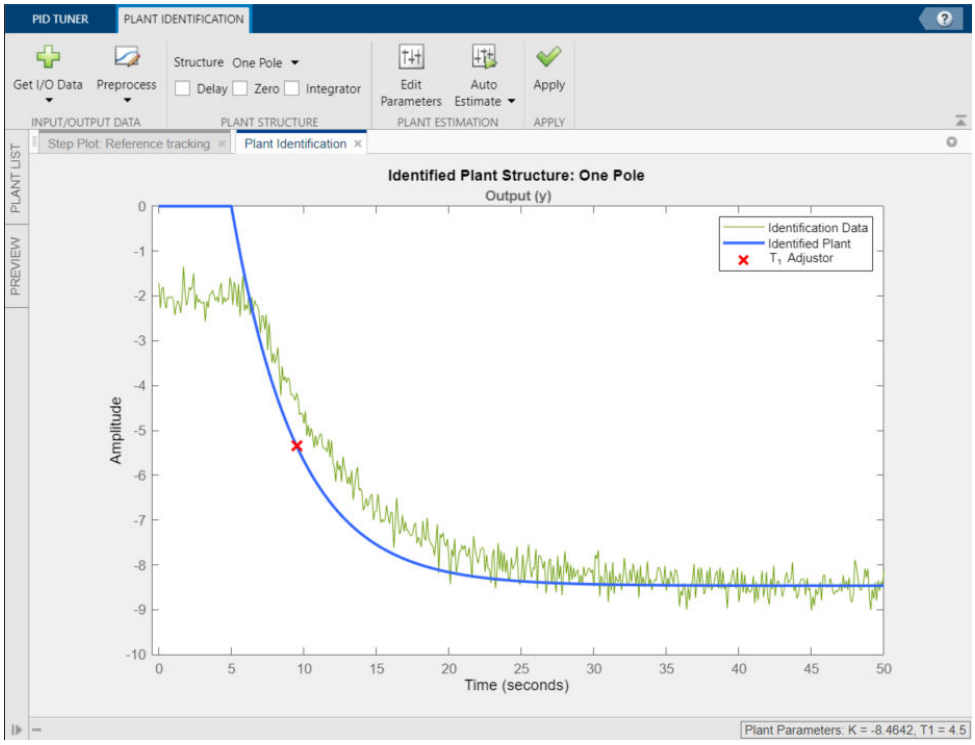


2. In the **Plant Identification** tab, click **Get I/O data**. Select the type of measured response data you have. For example, if you measured the response of your plant to a step input, select **Step Response**. To import the response of your system to an arbitrary stimulus, select **Arbitrary I/O Data**.

3 In the Import Response dialog box, enter information about your response data. For example, for step-response data stored in a variable `outputy` and sampled every 0.1s:

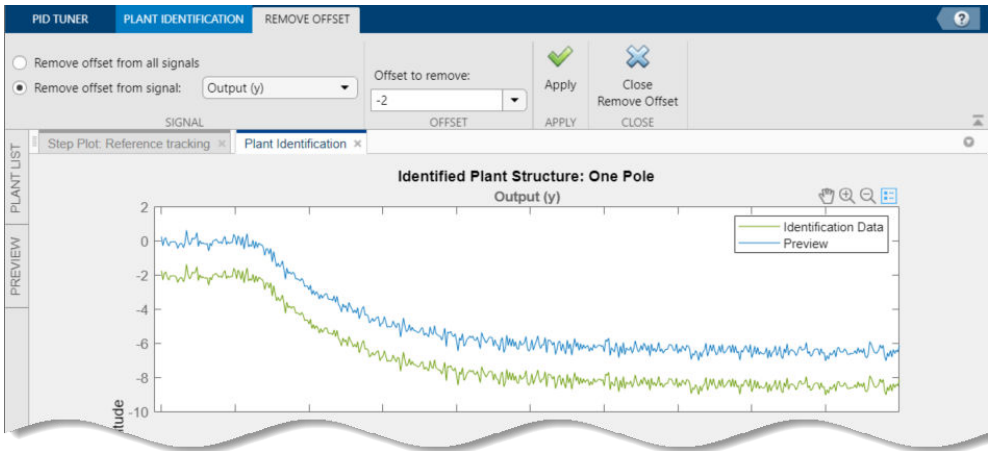


Click **Import**. The **Plant Identification** tab opens, displaying the response data and the response of an initial estimated plant.



4 Depending on the quality and features of your response data, you might want to perform some preprocessing on the data to improve the estimated plant results. The **Preprocess** menu gives you several options for preprocessing response data, such as removing offsets, filtering, or extracting on a subset of the data. In particular, when the response data has an offset, it is important for good identification results to remove the offset.

In the **Plant Identification** tab, click **Preprocess** and select the preprocessing option you want to use. A tab opens with a figure that displays the original and preprocessed data. Use the options in the tab to specify preprocessing parameters.



(For more information about preprocessing options, see “Preprocess Data” on page 7-64.)

When you are satisfied with the preprocessed signal, click **Update** to save the change to the signal. Click to return to the **Plant Identification** tab.

PID Tuner automatically adjusts the plant parameters to create a new initial guess for the plant based on the preprocessed response signal.

You can now adjust the structure and parameters of the estimated plant to obtain the estimated linear plant model for PID Tuning. See “Interactively Estimate Plant from Measured or Simulated Response Data” on page 7-55 for more information.

See Also

More About

- “System Identification for PID Control” on page 7-61
- “Input/Output Data for Identification” on page 7-67
- “Interactively Estimate Plant from Measured or Simulated Response Data” on page 7-55

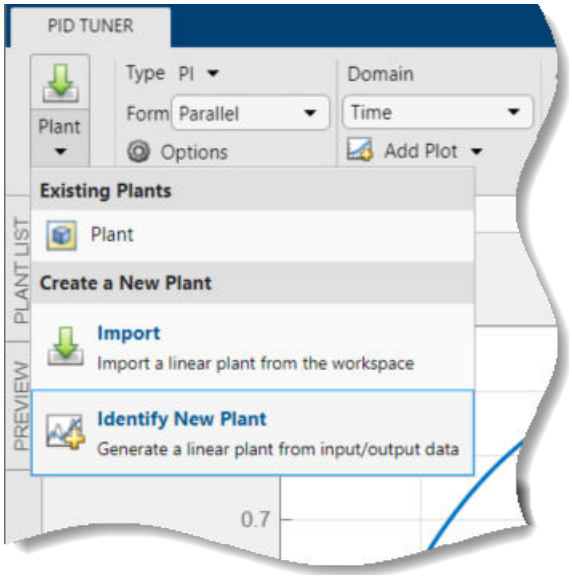
Interactively Estimate Plant from Measured or Simulated Response Data

If you have System Identification Toolbox software, **PID Tuner** lets you estimate the parameters of a linear plant model based on time-domain response data . **PID Tuner** then tunes a PID controller for the resulting estimated model. The response data can be either measured from your real-world system, or obtained by simulating your Simulink model. Plant estimation is especially useful when your Simulink model cannot be linearized or linearizes to zero. For plant identification, you must specify a finite value for the Simulink model stop time.

PID Tuner gives you several techniques to graphically, manually, or automatically adjust the estimated model to match your response data. This topic illustrates some of those techniques.

Obtain Response Data for Identification

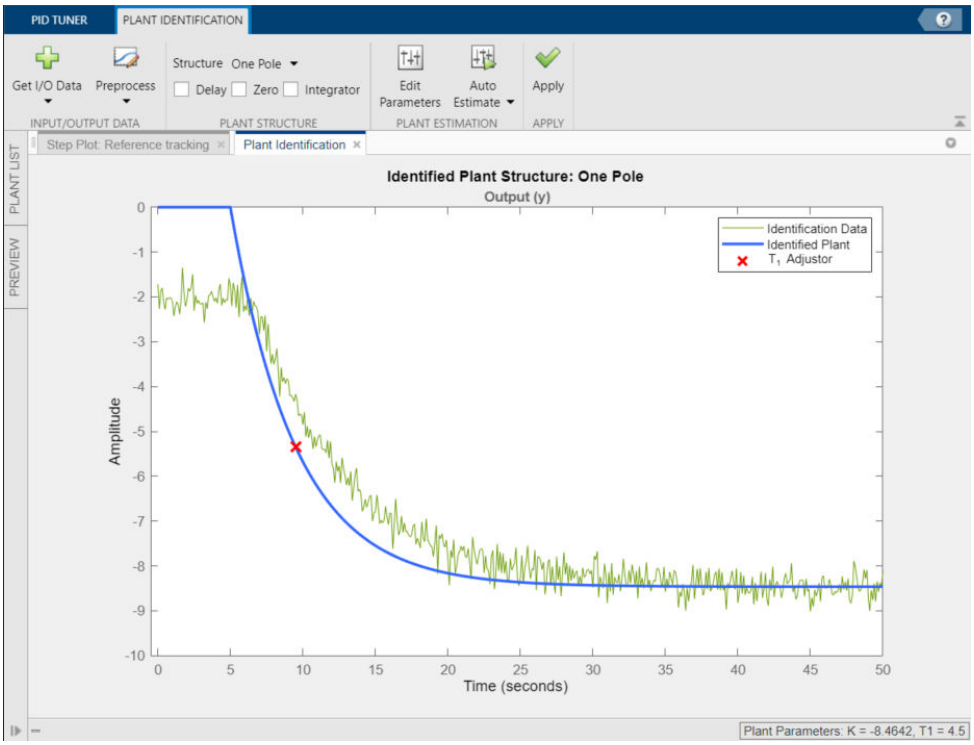
In **PID Tuner**, in the **PID Tuner** tab, in the **Plant** menu, select `Identify New Plant`.



In the **Plant Identification** tab, click **Get I/O data**. This menu allows you to obtain system response data in one of two ways:

- **Simulate Data**. Obtain system response data by simulating the response of your Simulink model to an input signal. For more information, see “Design PID Controller Using Simulated I/O Data” on page 7-108.
- **Import I/O Data**. Import measured system response data as described in “Import Measured Response Data for Plant Estimation” on page 7-51.

Once you have imported or simulated data, the **Plant Identification** plot displays the response data and the response of an initial estimated plant. You can now select the plant structure and adjust the estimated plant parameters until the response of the estimated plant is a good fit to the response data.



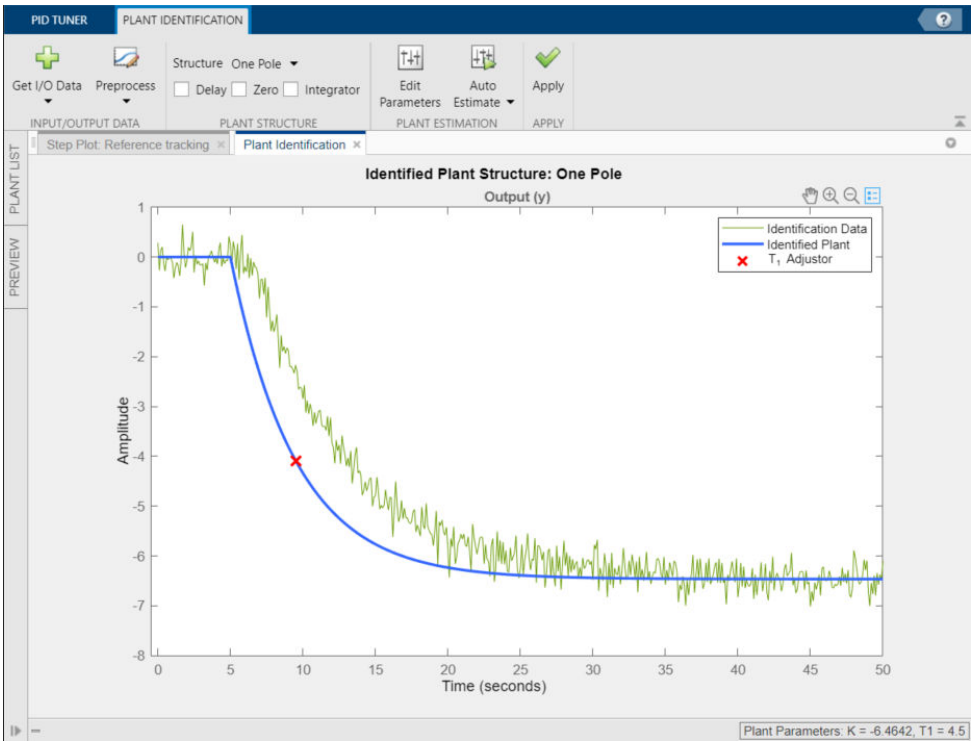
Preprocess Data

Depending on the quality and features of your imported or simulated data, you might want to perform some preprocessing on the data to improve the estimated plant results. **PID Tuner** provides several options for preprocessing response data, such as removing offsets, filtering, or extracting a subset of the data. For information, see “Preprocess Data” on page 7-64.

Adjust Plant Structure and Parameters

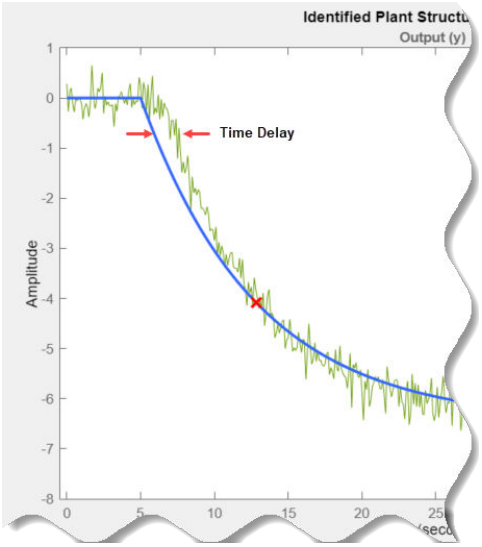
PID Tuner allows you to specify a plant structure, such as **One Pole**, **Two Real Poles**, or **State-Space Model**. In the **Structure** menu, choose the plant structure that best matches your response. You can also add a transport delay, a zero, or an integrator to your plant.

In the following sample plot, the one-pole structure gives the qualitatively correct response. You can make further adjustments to the plant structure and parameter values to make the response of the estimated system a better match to the measured response data.



PID Tuner gives you several ways to adjust the plant parameters:

- Graphically adjust the response of the estimated system by dragging the adjustors on the plot. In this example, drag the red `x` to adjust the estimated plant time constant. **PID Tuner** recalculates system parameters as you do so. As you change the estimated system's response, it becomes apparent that there is some time delay between the application of the step input at `t` = 5 s, and the response of the system to that step input.



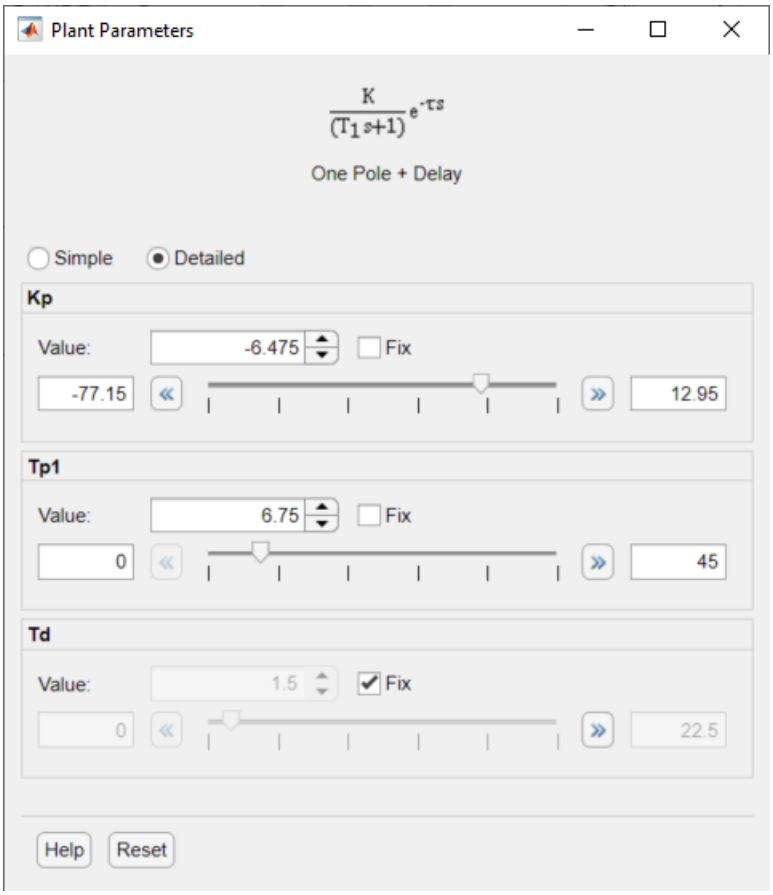
To add a transport delay to the estimated plant model, in the **Plant Structure** section, check **Delay**. A vertical line appears on the plot, indicating the current value of the delay. Drag the line

left or right to change the delay, and make further adjustments to the system response by dragging the red x.

- Adjust the numerical values of system parameters such as gains, time constants, and time delays.

  To numerically adjust the values of system parameters, click **Edit Parameters**.

  Suppose that you know from an independent measurement that the transport delay in your system is 1.5 seconds. In the **Plant Parameters** dialog box, enter 1.5 for τ. Check **Fix** to fix the parameter value. When you check **Fix** for a parameter, neither graphical nor automatic adjustments to the estimated plant model affect that parameter value.



- Automatically optimize the system parameters to match the measured response data. Click **Auto Estimate** to update the estimated system parameters using the current values as an initial guess.

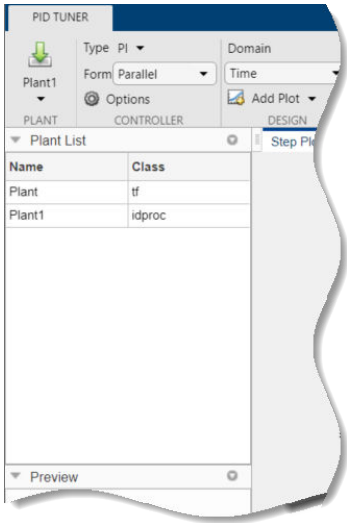
You can continue to iterate using any of these methods to adjust plant structure and parameter values until the estimated system response adequately matches the measured response.

Save Plant and Tune PID Controller

When you are satisfied with the fit, click **Save Plant**. Doing so saves the estimated plant, `Plant1`, to **PID Tuner** workspace. Doing so also selects the **Step Plot: Reference Tracking** figure

and returns you to the **PID Tuner** tab. **PID Tuner** automatically designs a PI controller for `Plant1`, and displays a response plot for the new closed-loop system. The **Plant** menu reflects that `Plant1` is selected for the current controller design.

Tip To examine variables stored in the **PID Tuner** workspace, use the **Plant List** area.

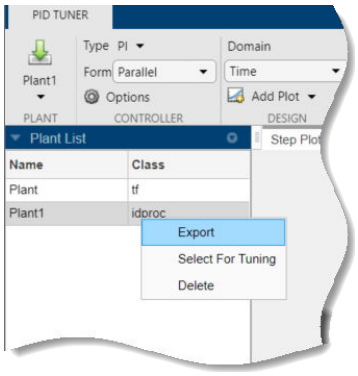


You can now use the **PID Tuner** tools to refine the controller design for the estimated plant and examine tuned system responses.

You can also export the identified plant from the **PID Tuner** workspace to the MATLAB workspace for further analysis. In the **PID Tuner** tab, click **Export**. Check the plant model you want to export to the MATLAB workspace. For this example, export `Plant1`, the plant you identified from response data. You can also export the tuned PID controller. Click **OK**. The models you selected are saved to the MATLAB workspace.

Identified plant models are saved as identified LTI models, such as `idproc` or `idss`.

Tip Alternatively, right-click a plant in the **Plant List** to select it for tuning or export it to the MATLAB workspace.



See Also

More About

- “Choosing Identified Plant Structure” on page 7-68
- “Input/Output Data for Identification” on page 7-67
- “System Identification for PID Control” on page 7-61
- “Import Measured Response Data for Plant Estimation” on page 7-51

System Identification for PID Control

Plant Identification

In many situations, a dynamic representation of the system you want to control is not readily available. One solution to this problem is to obtain a dynamical model using identification techniques. The system is excited by a measurable signal and the corresponding response of the system is collected at some sample rate. The resulting input-output data is then used to obtain a model of the system such as a transfer function or a state-space model. This process is called system identification or estimation. The goal of system identification is to choose a model that yields the best possible fit between the measured system response to a particular input and the model's response to the same input.

If you have a Simulink model of your control system, you can simulate input/output data instead of measuring it. The process of estimation is the same. The system response to some known excitation is simulated, and a dynamical model is estimated based upon the resulting simulated input/output data.

Whether you use measured or simulated data for estimation, once a suitable plant model is identified, you impose control objectives on the plant based on your knowledge of the desired behavior of the system that the plant model represents. You then design a feedback controller to meet those objectives.

If you have System Identification Toolbox software, you can use **PID Tuner** for both plant identification and controller design in a single interface. You can import input/output data and use it to identify one or more plant models. Or, you can obtain simulated input/output data from a Simulink model and use that to identify one or more plant models. You can then design and verify PID controllers using these plants. **PID Tuner** also allows you to directly import plant models, such as one you have obtained from an independent identification task.

For an overview of system identification, see About System Identification (System Identification Toolbox).

Linear Approximation of Nonlinear Systems for PID Control

The dynamical behavior of many systems can be described adequately by a linear relationship between the system's input and output. Even when behavior becomes nonlinear in some operating regimes, there are often regimes in which the system dynamics are linear. For example, the behavior of an operational amplifier or the lift-vs-force dynamics of aerodynamic bodies can be described by linear models, within a certain limited operating range of inputs. For such a system, you can perform an experiment (or a simulation) that excites the system only in its linear range of behavior and collect the input/output data. You can then use the data to estimate a linear plant model, and design a PID controller for the linear model.

In other cases, the effects of nonlinearities are small. In such a case, a linear model can provide a good approximation, such that the nonlinear deviations are treated as disturbances. Such approximations depend heavily on the input profile, the amplitude and frequency content of the excitation signal.

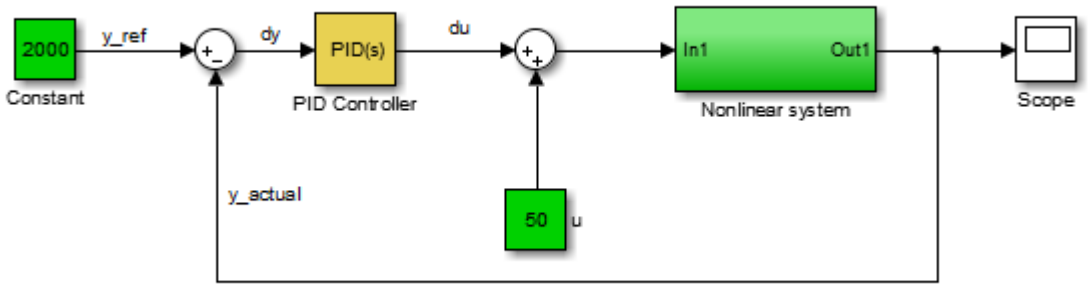
Linear models often describe the deviation of the response of a system from some equilibrium point, due to small perturbing inputs. Consider a nonlinear system whose output, $y(t)$, follows a prescribed trajectory in response to a known input, $u(t)$. The dynamics are described by $dx(t)/dt = f(x, u)$, $y = g(x,u)$. Here, x is a vector of internal states of the system, and y is the vector of output variables. The

functions f and g, which can be nonlinear, are the mathematical descriptions of the system and measurement dynamics. Suppose that when the system is at an equilibrium condition, a small perturbation to the input, Δu, leads to a small perturbation in the output, Δy:

$$\Delta \dot{x} = \frac{\partial f}{\partial x}\Delta x + \frac{\partial f}{\partial u}\Delta u,$$

$$\Delta y = \frac{\partial g}{\partial x}\Delta x + \frac{\partial g}{\partial u}\Delta u\,.$$

For example, consider the system of the following Simulink block diagram:



When operating in a disturbance-free environment, the nominal input of value 50 keeps the plant along its constant trajectory of value 2000. Any disturbances would cause the plant to deviate from this value. The PID Controller's task is to add a small correction to the input signal that brings the system back to its nominal value in a reasonable amount of time. The PID Controller thus needs to work only on the linear deviation dynamics even though the actual plant itself might be nonlinear. Thus, you might be able to achieve effective control over a nonlinear system in some regimes by designing a PID controller for a linear approximation of the system at equilibrium conditions.

Linear Process Models

A common use case is designing PID controllers for the steady-state operation of manufacturing plants. In these plants, a model relating the effect of a measurable input variable on an output quantity is often required in the form of a SISO plant. The overall system may be MIMO in nature, but the experimentation or simulation is carried out in a way that makes it possible to measure the incremental effect of one input variable on a selected output. The data can be quite noisy, but since the expectation is to control only the dominant dynamics, a low-order plant model often suffices. Such a proxy is obtained by collecting or simulating input-output data and deriving a process model (low order transfer function with unknown delay) from it. The excitation signal for deriving the data can often be a simple bump in the value of the selected input variable.

Advanced System Identification Tasks

In **PID Tuner**, you can only identify single-input, single output, continuous-time plant models. Additionally, **PID Tuner** cannot perform the following system identification tasks:

- Identify transfer functions of arbitrary number of poles and zeros. (**PID Tuner** can identify transfer functions up to three poles and one zero, plus an integrator and a time delay. **PID Tuner** can identify state-space models of arbitrary order.)

- Estimate the disturbance component of a model, which can be useful for separating measured dynamics from noise dynamics.
- Validate estimation by comparing the plant response against an independent dataset.
- Perform residual analysis.

If you need these enhanced identification features, import your data into the **System Identification** app (**System Identification**). Use the **System Identification** app to perform model identification and export the identified model to the MATLAB workspace. Then import the identified model into **PID Tuner** for PID controller design.

For more information about the System Identification Tool, see “Identify Linear Models Using System Identification App” (System Identification Toolbox).

See Also

System Identification

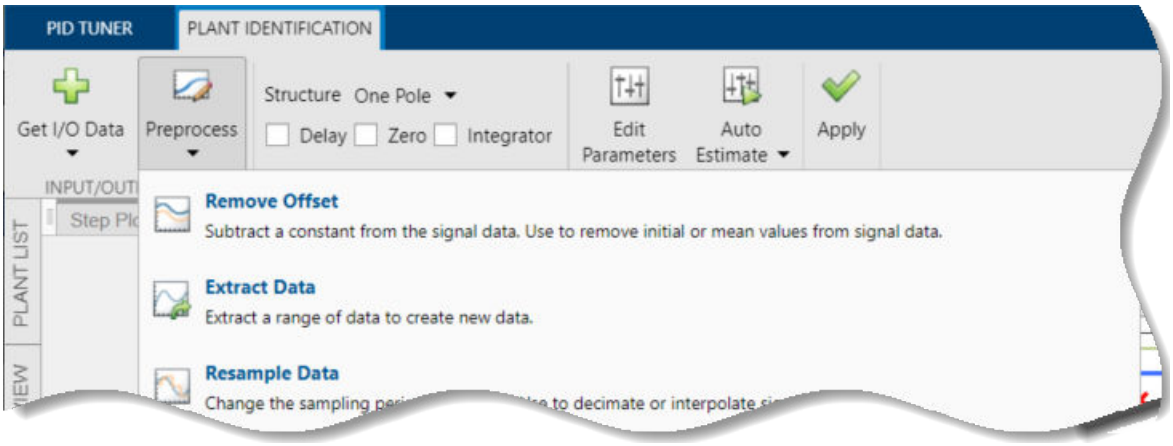
More About

- “Input/Output Data for Identification” on page 7-67
- “Choosing Identified Plant Structure” on page 7-68
- “Interactively Estimate Plant from Measured or Simulated Response Data” on page 7-55

Preprocess Data

Ways to Preprocess Data

In **PID Tuner**, you can preprocess plant data before you use it for estimation. After you import I/O data, on the **Plant Identification** tab, use the **Preprocess** menu to select a preprocessing operation.



- “Remove Offset” on page 7-64 — Remove mean values, a constant value, or an initial value from the data.
- “Scale Data” on page 7-65 — Scale data by a constant value, signal maximum value, or signal initial value.
- “Extract Data” on page 7-65 — Select a subset of the data to use in the . You can graphically select the data to extract, or enter start and end times in the text boxes.
- “Filter Data” on page 7-65 — Process data using a low-pass, high-pass, or band-pass filter.
- “Resample Data” on page 7-65 –– Resample data using zero-order hold or linear interpolation.
- “Replace Data” on page 7-66 –– Replace data with a constant value, region initial value, region final value, or a line. You can use this functionality to replace outliers.

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

Remove Offset

It is important for good results to remove data offsets. In the **Remove Offset** tab, you can remove offset from all signals at once or select a particular signal using the **Remove offset from signal** drop down list. Specify the value to remove using the **Offset to remove** drop down list. The options are:

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

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

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

Scale Data

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

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

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

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

Extract Data

Select a subset of data to use in **Extract Data** tab. You can extract data graphically or by specifying start time and end time. To extract data graphically, click and drag the vertical bars to select a region of the data to use.

Filter Data

You can filter your data using a low-pass, high-pass, or band-pass filter. A low-pass filter blocks high frequency signals, a high-pass filter blocks low frequency signals, and a band-pass filter combines the properties of both low- and high-pass filters.

On the **Low-Pass Filter**, **High-Pass Filter**, or **Band-Pass Filter** tab, you can choose to filter all signals or specify a particular signal. For the low-pass and high-pass filtering, you can specify the normalized cutoff frequency of the signal. Where, a normalized frequency of 1 corresponds to half the sampling rate. For the band-pass filter, you can specify the normalized start and end frequencies. Specify the frequencies by either entering the value in the associated field on the tab. Alternatively, you can specify filter frequencies graphically, by dragging the vertical bars in the frequency-domain plot of your data.

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

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

Resample Data

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

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

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

The modified data is shown in preview in the plot.

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

Replace Data

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

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

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

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

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

See Also

More About

- “Input/Output Data for Identification” on page 7-67
- “System Identification for PID Control” on page 7-61
- “Import Measured Response Data for Plant Estimation” on page 7-51

Input/Output Data for Identification

Data Preparation

Identification of a plant model for PID tuning requires a single-input, single-output data set.

If you have measured data, use the data import dialogs to bring in identification data. Some common sources of identification data are transient tests such as bump test and impact test. For such data, **PID Tuner** provides dedicated dialogs that require you to specify data for only the output signal while characterizing the input by its shape. For an example, see “Interactively Estimate Plant from Measured or Simulated Response Data” on page 7-55.

If you want to obtain input/output data by simulating a Simulink model, the **PID Tuner** interface lets you specify the shape of the input stimulus used to generate the response. For an example, see the Simulink Control Design example “Design a PID Controller Using Simulated I/O Data.”

Data Preprocessing

PID Tuner lets you preprocess your imported or simulated data. **PID Tuner** provides various options for detrending, scaling, and filtering the data.

It is strongly recommended to remove any equilibrium-related signal offsets from the input and output signals before proceeding with estimation. You can also filter the data to focus the signal contents to the frequency band of interest.

Some data processing actions can alter the nature of the data, which can result in transient data (step, impulse or wide pulse responses) to be treated as arbitrary input/output data. When that happens the identification plot does not show markers for adjusting the model time constants and damping coefficient.

For an example that includes a data-preprocessing step, see: “Interactively Estimate Plant from Measured or Simulated Response Data” on page 7-55.

For further information about data-preprocessing options, see “Preprocess Data” on page 7-64.

Choosing Identified Plant Structure

PID Tuner provides two types of model structures for representing the plant dynamics: process models and state-space models.

Use your knowledge of system characteristics and the level of accuracy required by your application to pick a model structure. In absence of any prior information, you can gain some insight into the order of dynamics and delays by analyzing the experimentally obtained step response and frequency response of the system. For more information see the following in the System Identification Toolbox documentation:

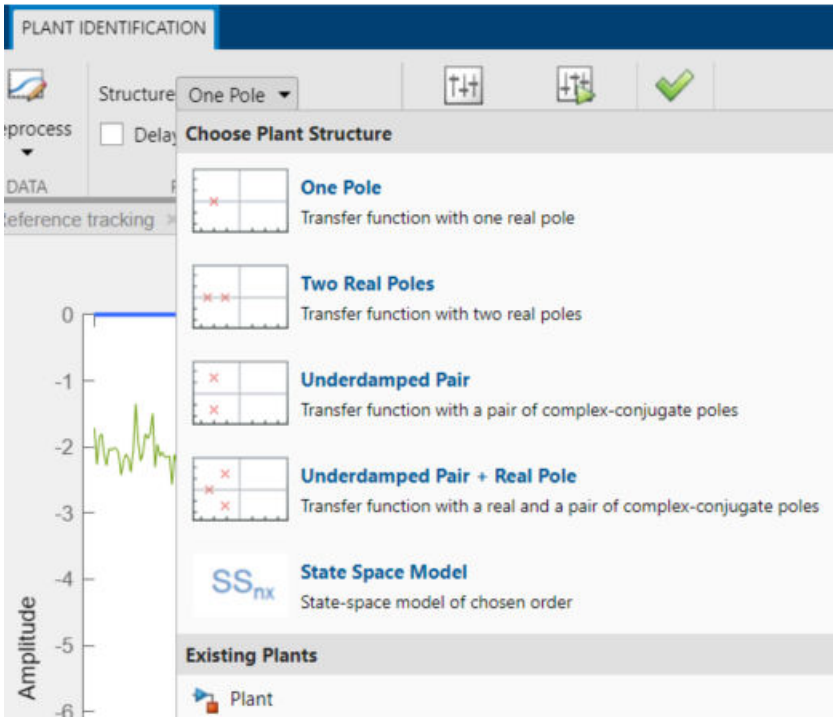
- “Correlation Models” (System Identification Toolbox)
- “Frequency-Response Models” (System Identification Toolbox)

Each model structure you choose has associated dynamic elements, or model parameters. You adjust the values of these parameters manually or automatically to find an identified model that yields a satisfactory match to your measured or simulated response data. In many cases, when you are unsure of the best structure to use, it helps to start with the simplest model structure, transfer function with one pole. You can progressively try identification with higher-order structures until a satisfactory match between the plant response and measured output is achieved. The state-space model structure allows an automatic search for optimal model order based on an analysis of the input-output data.

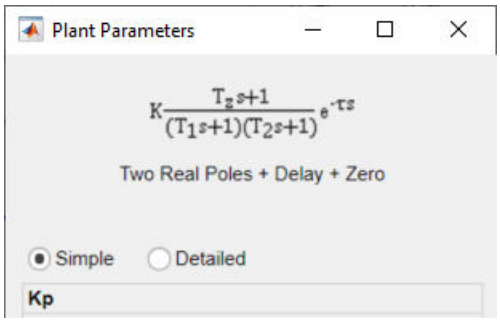
When you begin the plant identification task, a transfer function model structure with one real pole is selected by default. This default set up is not sensitive to the nature of the data and may not be a good fit for your application. It is therefore recommended that you choose a suitable model structure before performing parameter identification.

Process Models

Process models are transfer functions with 3 or fewer poles, and can be augmented by addition of zero, delay and integrator elements. Process models are parameterized by model parameters representing time constants, gain, and time delay. In **PID Tuner**, choose a process model in the **Plant Identification** tab using the **Structure** menu.



For any chosen structure you can optionally add a delay, a zero and/or an integrator element using the corresponding checkboxes. Click **Edit Parameters** to view the model transfer function configured by these choices.



The simplest available process model is a transfer function with one real pole and no zero or delay elements:

$$H(s) = \frac{K}{T_1 s + 1}.$$

This model is defined by the parameters K, the gain, and T_1, the first time constant. The most complex process-model structure choose has three poles, an additional integrator, a zero, and a time delay, such as the following model, which has one real pole and one complex conjugate pair of poles:

$$H(s) = K\frac{T_z s + 1}{s(T_1 s + 1)\left(T_\omega^2 s^2 + 2\zeta T_\omega s + 1\right)}e^{-\tau s}.$$

In this model, the configurable parameters include the time constants associated with the poles and the zero, T_1, T_ω, and T_z. The other parameters are the damping coefficient ζ, the gain K, and the time delay τ.

When you select a process model type, **PID Tuner** automatically computes initial values for the plant parameters and displays a plot showing both the estimated model response and your measured or simulated data. You can edit the parameter values graphically using indicators on the plot, or numerically using the Plant Parameters editor. For an example illustrating this process, see "Interactively Estimate Plant Parameters from Response Data".

The following table summarizes the various parameters that define the available types of process models.

Parameter	Used By	Description
K — Gain	All transfer functions	Can take any real value. In the plot, drag the plant response curve (blue) up or down to adjust K.
T_1 — First time constant	Transfer function with one or more real poles	Can take any value between 0 and T, the time span of measured or simulated data. In the plot, drag the red x left (towards zero) or right (towards T) to adjust T_1.
T_2— Second time constant	Transfer function with two real poles	Can take any value between 0 and T, the time span of measured or simulated data. In the plot, drag the magenta x left (towards zero) or right (towards T) to adjust T_2.
T_ω — Time constant associated with the natural frequency ω_n, where $T_\omega = 1/\omega_n$	Transfer function with underdamped pair (complex conjugate pair) of poles	Can take any value between 0 and T, the time span of measured or simulated data. In the plot, drag one of the orange response envelope curves left (towards zero) or right (towards T) to adjust T_ω.
ζ — Damping coefficient	Transfer function with underdamped pair (complex conjugate pair) of poles	Can take any value between 0 and 1. In the plot, drag one of the orange response envelope curves left (towards zero) or right (towards T) to adjust ζ.

Parameter	Used By	Description
τ — Transport delay	Any transfer function	Can take any value between 0 and T, the time span of measured or simulated data. In the plot, drag the orange vertical bar left (towards zero) or right (towards T) to adjust τ.
T_z — Model zero	Any transfer function	Can take any value between –T and T, the time span of measured or simulated data. In the plot, drag the red circle left (towards –T) or right (towards T) to adjust T_z.
Integrator	Any transfer function	Adds a factor of $1/s$ to the transfer function. There is no associated parameter to adjust.

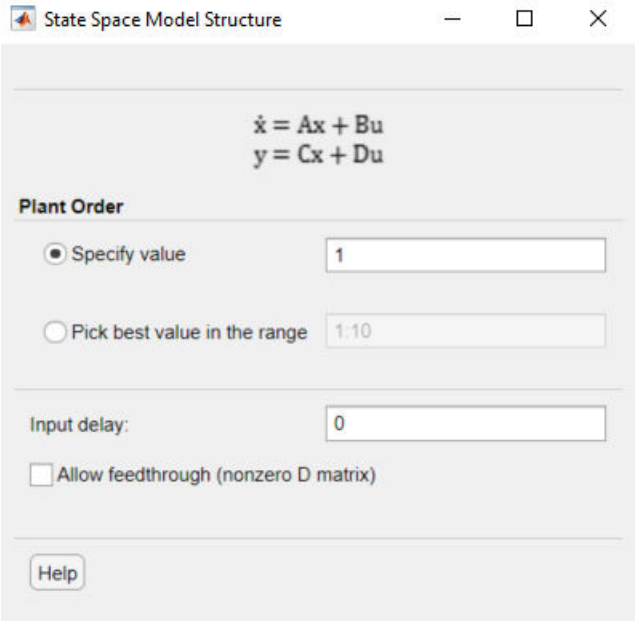
State-Space Models

The state-space model structure for identification is primarily defined by the choice of number of states, the model order. Use the state-space model structure when higher order models than those supported by process model structures are required to achieve a satisfactory match to your measured or simulated I/O data. In the state-space model structure, the system dynamics are represented by the state and output equations:

$$\dot{x} = Ax + Bu,$$
$$y = Cx + Du\,.$$

x is a vector of state variables, automatically chosen by the software based on the selected model order. u represents the input signal, and y the output signals.

To use a state-space model structure, in the **Plant Identification** tab, in the **Structure** menu, select `State-Space Model`. Then click **Configure Structure** to open the **State-Space Model Structure** dialog box.



Use the dialog box to specify model order, delay and feedthrough characteristics. If you are unsure about the order, select **Pick best value in the range**, and enter a range of orders. In this case, when you click **Estimate** in the **Plant Estimation** tab, the software displays a bar chart of Hankel singular values. Choose a model order equal to the number of Hankel singular values that make significant contributions to the system dynamics.

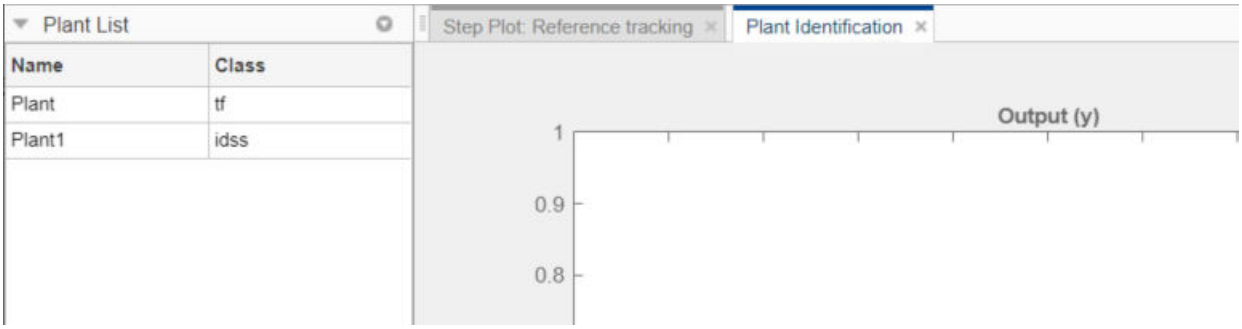
When you choose a state-space model structure, the identification plot shows a plant response (blue) curve only if a valid estimated model exists. For example, if you change structure after estimating a process model, the state-space equivalent of the estimated model is displayed. If you change the model order, the plant response curve disappears until a new estimation is performed.

When using the state-space model structure, you cannot directly interact with the model parameters. The identified model should thus be considered unstructured with no physical meaning attached to the state variables of the model.

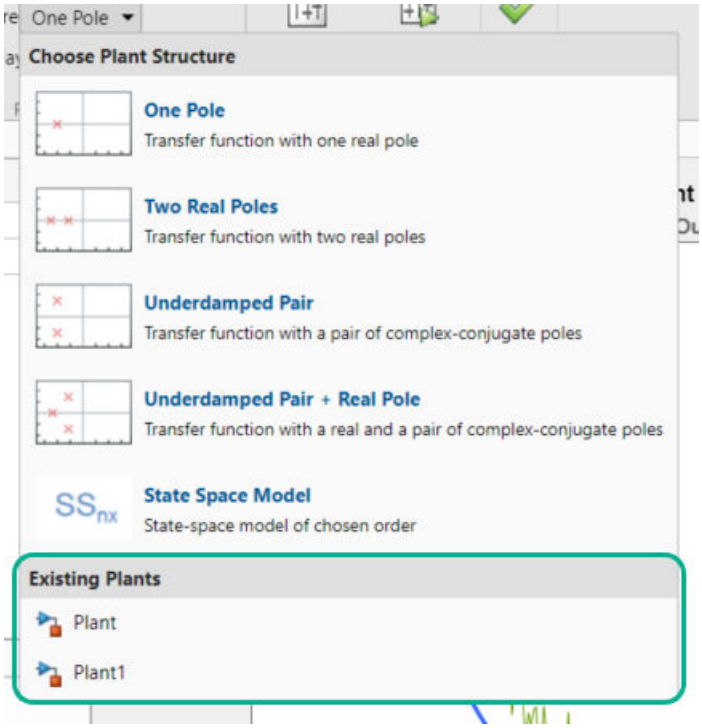
However, you can graphically adjust the input delay and the overall gain of the model. When you select a state-space model with a time delay, the delay is represented on the plot by a vertical orange bar is shown on the plot. Drag this bar horizontally to change the delay value. Drag the plant response (blue) curve up and down to adjust the model gain.

Existing Plant Models

Any previously imported or identified plant models are listed the **Plant List** area.



You can define the model structure and initialize the model parameter values using one of these plants. To do so, in the **Plant Identification** tab, in the **Structure** menu, select the linear plant model you want to use for structure an initialization.



If the plant you select is a process model (`idproc` object), **PID Tuner** uses its structure. If the plant is any other model type, **PID Tuner** uses the state-space model structure.

Switching Between Model Structures

When you switch from one model structure to another, the software preserves the model characteristics (pole/zero locations, gain, delay) as much as possible. For example, when you switch from a one-pole model to a two-pole model, the existing values of T_1, T_z, τ and K are retained, T_2 is initialized to a default (or previously assigned, if any) value.

Estimating Parameter Values

Once you have selected a model structure, you have several options for manually or automatically adjusting parameter values to achieve a good match between the estimated model response and your measured or simulated input/output data. For an example that illustrates all these options, see:

- “Interactively Estimate Plant Parameters from Response Data” (Control System Toolbox)
- “Interactively Estimate Plant from Measured or Simulated Response Data” on page 7-55 Simulink Control Design)

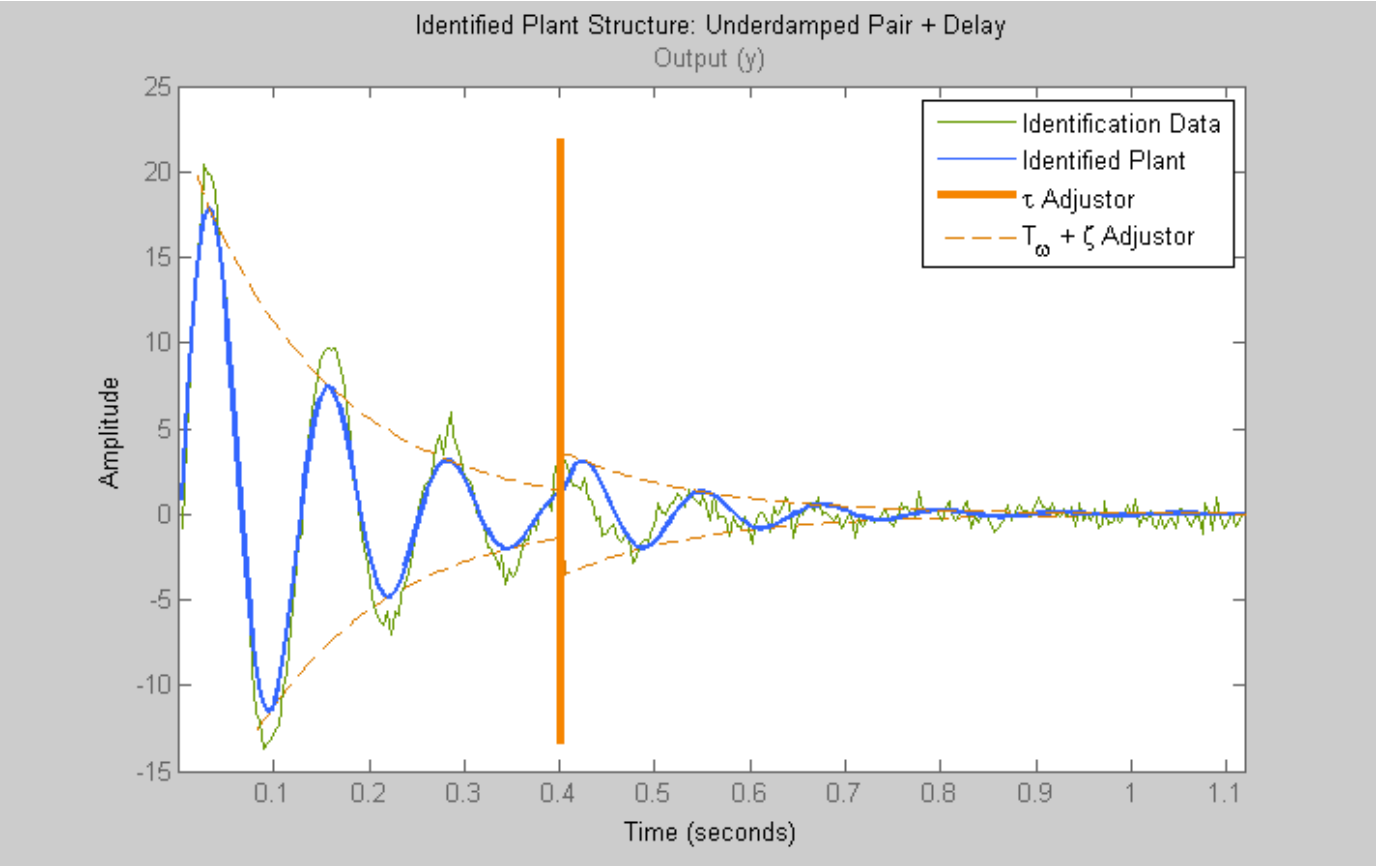
PID Tuner does not perform a smart initialization of model parameters when a model structure is selected. Rather, the initial values of the model parameters, reflected in the plot, are arbitrarily-chosen middle of the range values. If you need a good starting point before manually adjusting the parameter values, use the **Initialize and Estimate** option from the **Plant Identification** tab.

Handling Initial Conditions

In some cases, the system response is strongly influenced by the initial conditions. Thus a description of the input to output relationship in the form of a transfer function is insufficient to fit the observed data. This is especially true of systems containing weakly damped modes. **PID Tuner** allows you to estimate initial conditions in addition to the model parameters such that the sum of the initial condition response and the input response matches the observed output well. Use the **Estimation Options** dialog box to specify how the initial conditions should be handled during automatic estimation. By default, the initial condition handling (whether to fix to zero values or to estimate) is automatically performed by the estimation algorithm. However, you can enforce a certain choice by using the Initial Conditions menu.

Initial conditions can only be estimated with automatic estimation. Unlike the model parameters, they cannot be modified manually. However, once estimated they remain fixed to their estimated values, unless the model structure is changed or new identification data is imported.

If you modify the model parameters after having performed an automatic estimation, the model response will show a fixed contribution (i.e., independent of model parameters) from initial conditions. In the following plot, the effects of initial conditions were identified to be particularly significant. When the delay is adjusted afterwards, the portion of the response to the left of the input delay marker (the τ Adjustor) comes purely from initial conditions. The portion to the right of the τ Adjustor contains the effects of both the input signal as well as the initial conditions.



See Also

More About

- “System Identification for PID Control” on page 7-61
- “Interactively Estimate Plant from Measured or Simulated Response Data” on page 7-55

Design Controller for Power Electronics Model Using Frequency Response Data

This example shows how to design a PID controller for a power electronics system modeled in Simulink® using Simscape™ Electrical™ components.

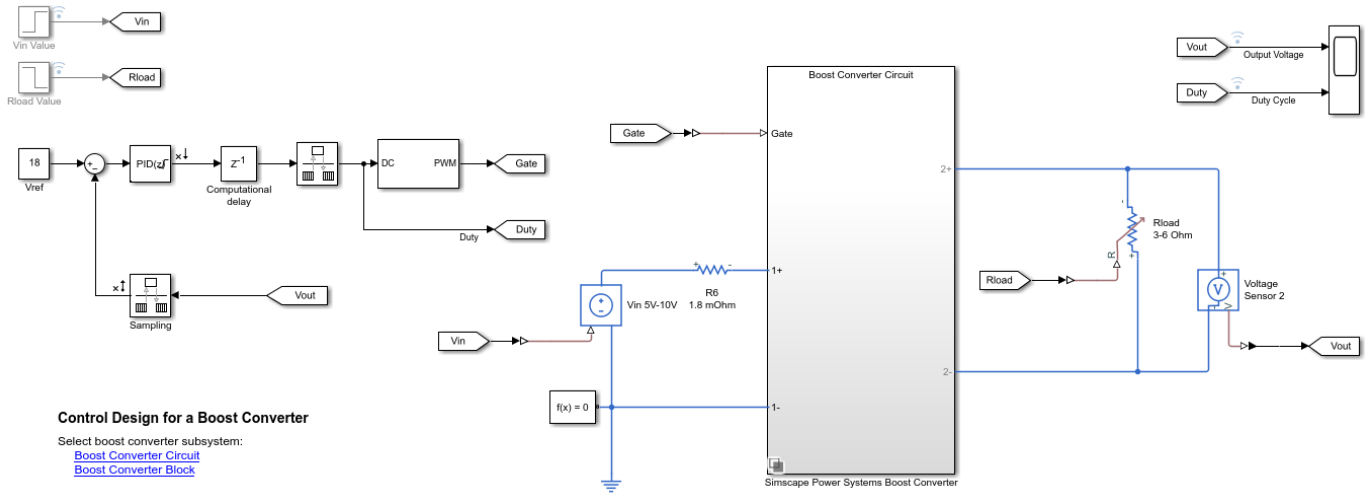
Typically, power electronics systems cannot be linearized because they use high-frequency switching components, such as pulse-width modulation (PWM) generators. However, most Simulink® Control Design™ PID tuning tools design PID gains based on a linearized plant model. To obtain such a model for a power electronics model that cannot be linearized, you can:

- Estimate the plant frequency response over a range of frequencies as shown in this example.
- Estimate the parameters of a linear model of the plant using System Identification Toolbox™ software. For an example see, “Design Controller for Power Electronics Model Using Simulated I/O Data” on page 7-94.

Boost Converter Model

This example uses a boost converter model as an example of a power electronics system. A boost converter circuit converts one DC voltage to another, typically higher, DC voltage by controlled chopping or switching of the source voltage.

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



In this model, a MOSFET driven by a pulse-width modulation (PWM) signal is used for switching. The output voltage $Vout$ should be regulated to the reference value $Vref$. A digital PID controller adjusts the PWM duty cycle, $Duty$, based on the voltage error signal. For this example, you estimate the frequency response from the PWM duty cycle to the load voltage $Vout$.

Simscape Electrical software contains predefined blocks for many power electronics systems. This model contains a variant subsystem with two versions of the boost converter model:

- Boost converter circuit constructed using electrical power components. The parameters of the circuit components are based on [1].

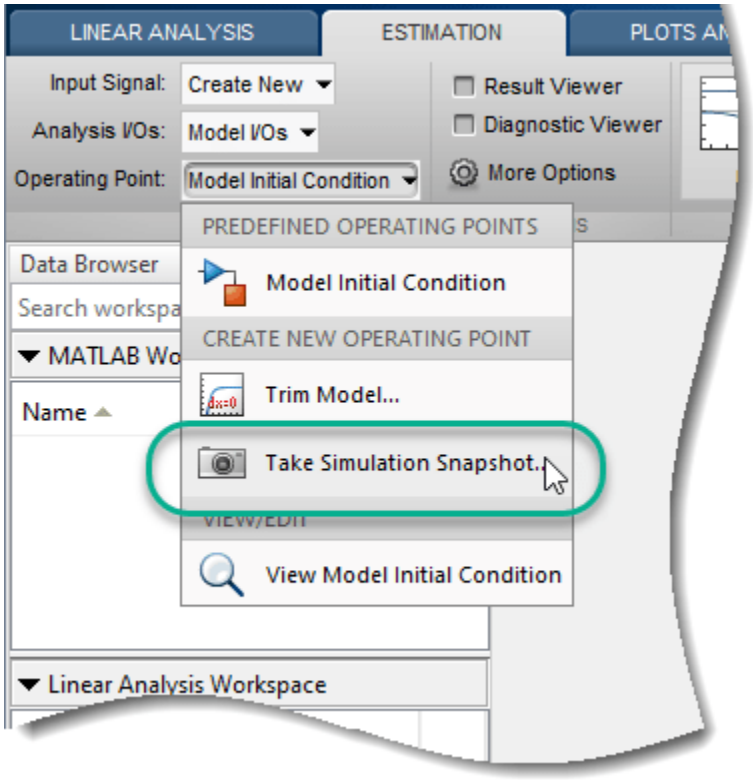
- Boost converter block configured to have the same parameters as the boost converter circuit. For more information on this block, see Boost Converter (Simscape Electrical).

Find Model Operating Point

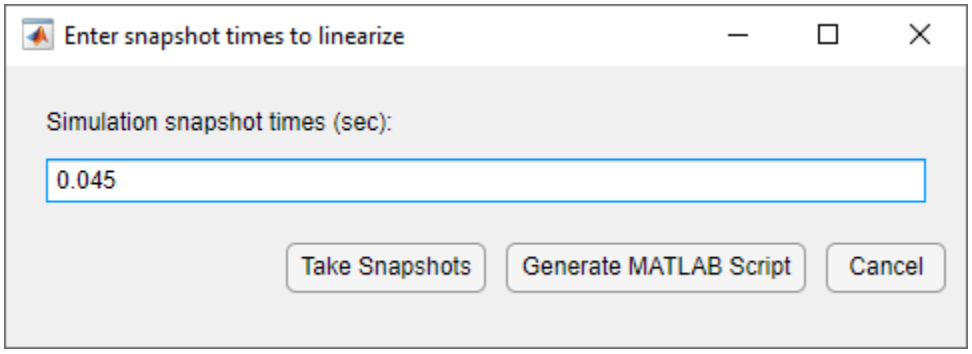
To design a controller for the boost converter, you must first determine the steady-state operating point at which you want the converter to operate. For more information on finding operating points, see “Find Steady-State Operating Points for Simscape Models” on page 1-104. For this example, use an operating point estimated from a simulation snapshot.

To find the operating point, use the **Model Linearizer**. To open **Model Linearizer**, in the Simulink model window, on the **Apps** tab, click **Model Linearizer**.

In the **Model Linearizer**, on the **Linear Analysis** tab, in the **Operating Point** drop-down list, select **Take Simulation Snapshot**.

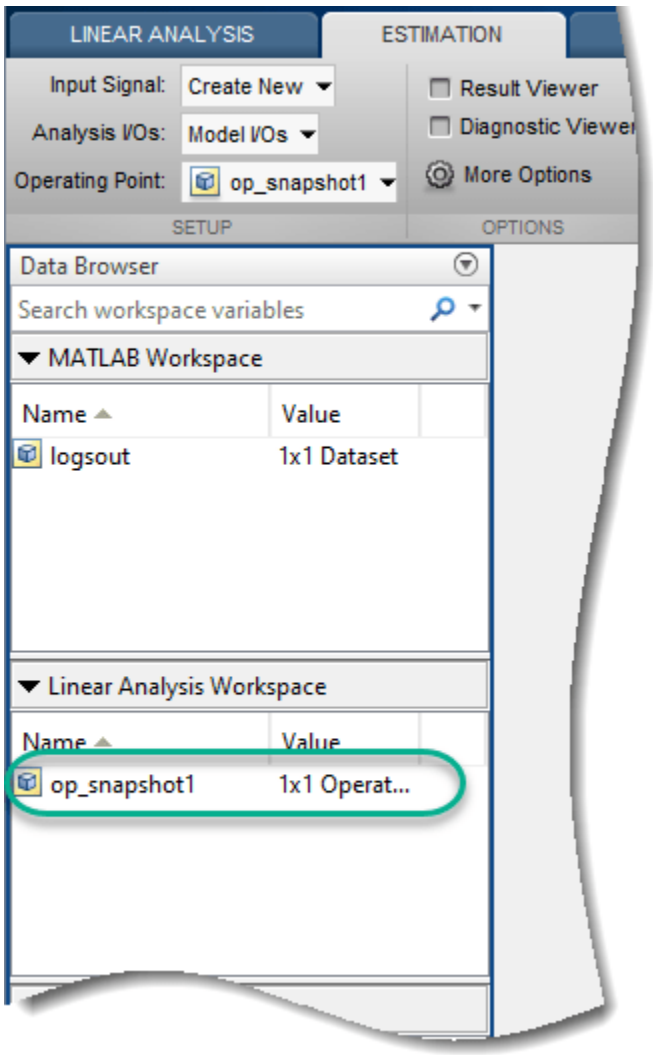


In the Enter snapshot times to linearize dialog box, in the **Simulation snapshot times** field, enter `0.045`, which is enough time for the closed-loop system to reach steady state.

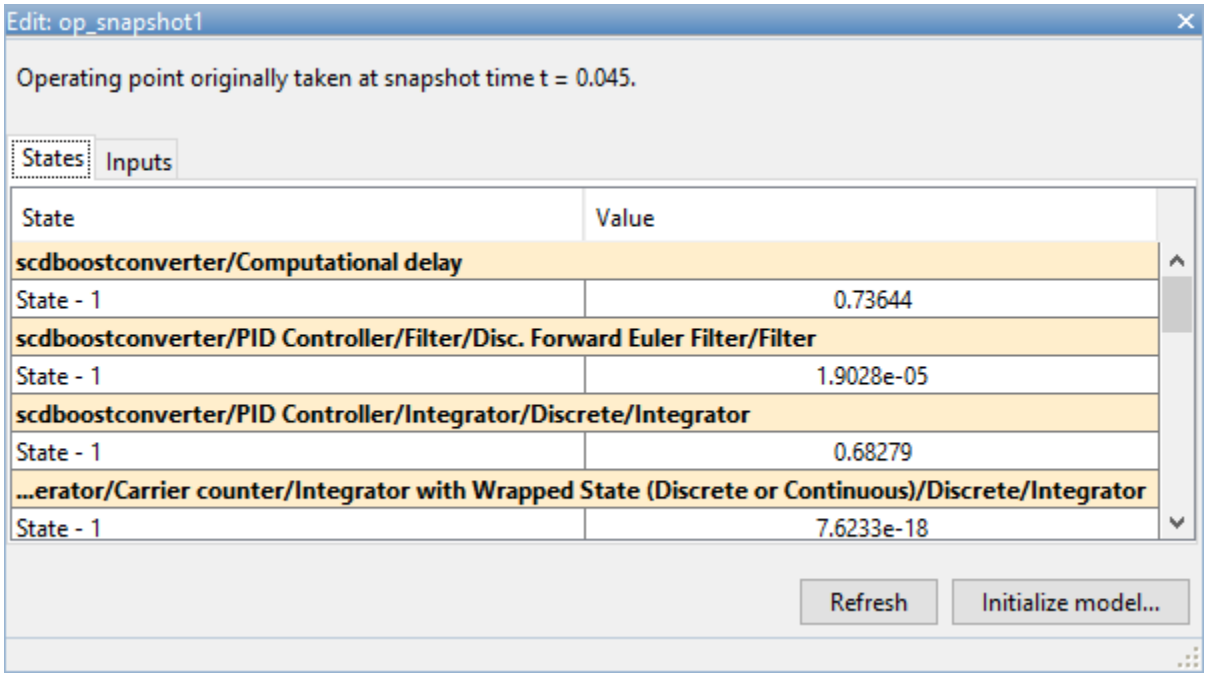


Click **Take Snapshots**.

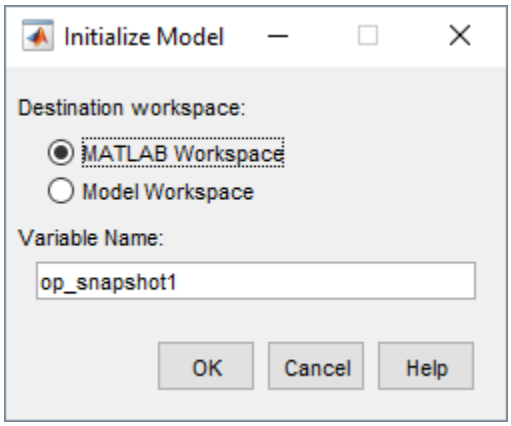
The software simulates the model and creates an operating point that contains the input and state values of the model at the specified snapshot time. This operating point, `op_snapshot1`, is added to the **Linear Analysis Workspace**.



To initialize the model with the computed operating point, double-click `op_snapshot1`.



In the Edit dialog box, click **Initialize model**.



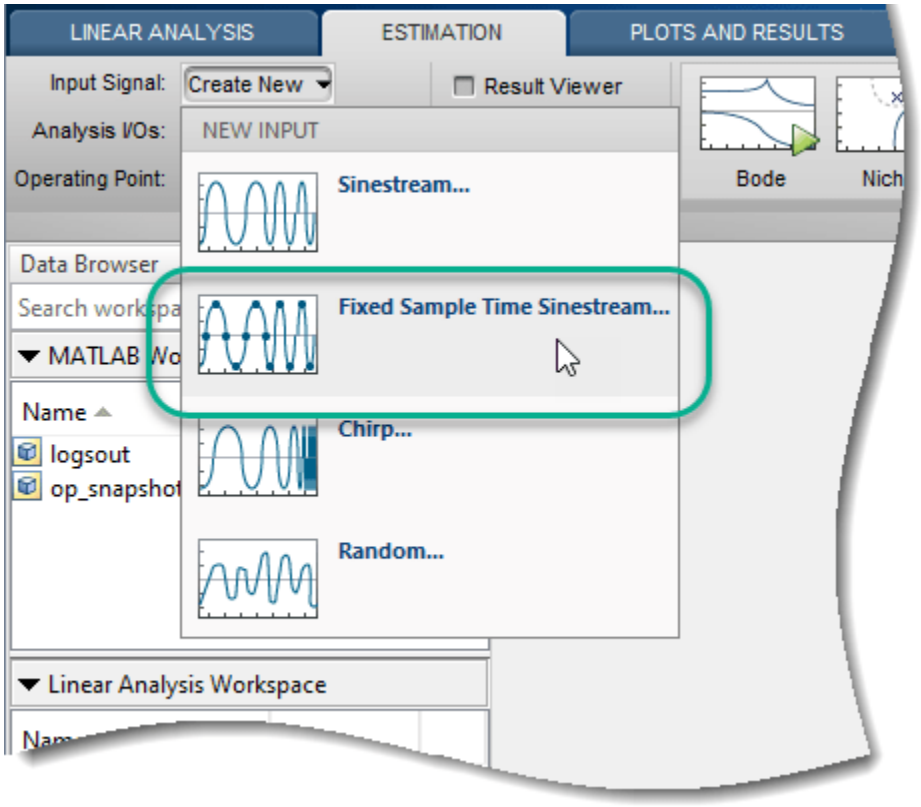
In the Initialize Model dialog box, select **MATLAB Workspace**, and click **OK**. The software exports the operating point to the MATLAB® workspace and initializes the model with the inputs and states in the operating point.

Collect Frequency Response Data

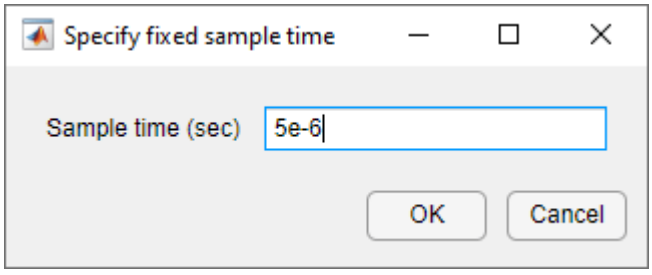
Before collecting frequency response data, you must first specify the portion of the model for which you want to find the frequency response. For this example, the model contains open-loop input and output linear analysis points at the output and input of the PID controller block.

To collect frequency response data, you must also specify an input signal. For this example, use a fixed-step sinestream signal. For more information on defining sinestream input signals, see “Sinestream Input Signals” on page 5-29.

On the **Estimation** tab, in the **Input Signal** drop-down list, click **Fixed Sample Time Sinestream**.



In the Specify fixed sample time dialog box, specify a **Sample time** of `5e-6` seconds. The sample time of the Sinestream input signal must match the sample time at the input linear analysis point.



Click **OK**.

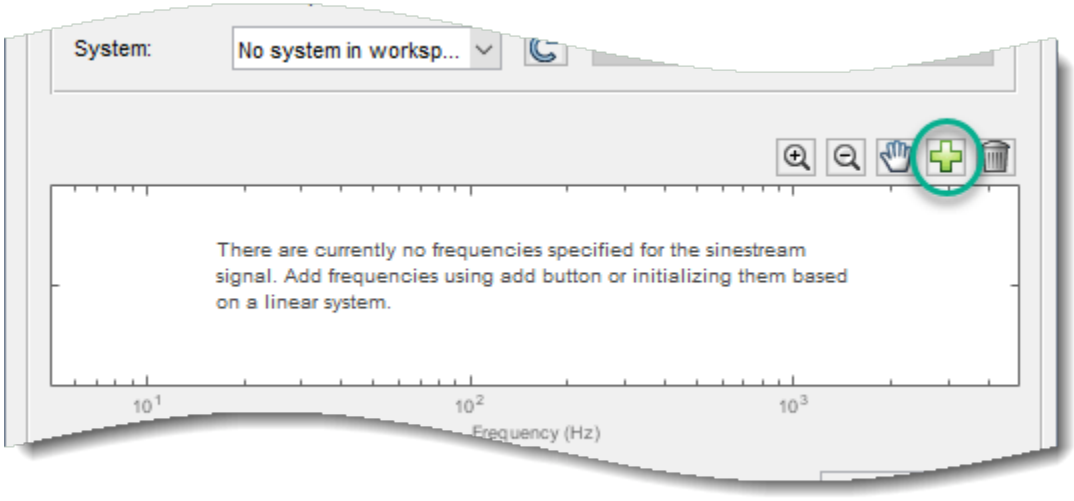
In the Create sinestream input with fixed sample time dialog box, configure the parameters of the sinestream signal.

Specify the frequency units for estimation. In the **Frequency units** drop-down list, select `Hz`.

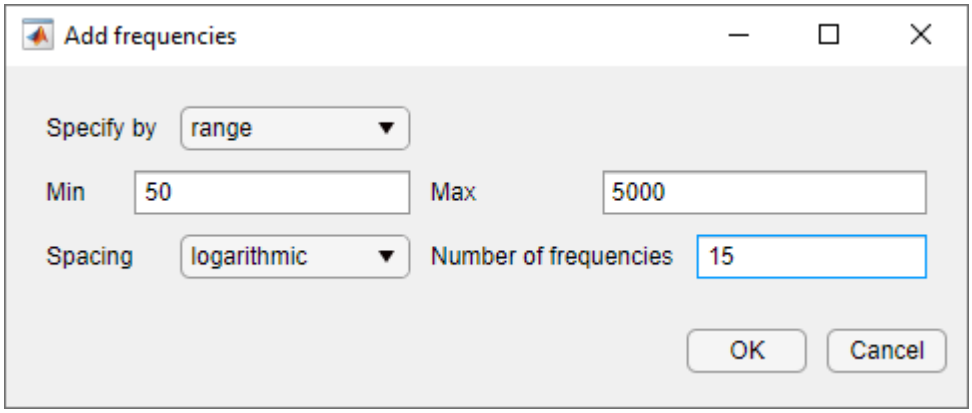
For this example, the frequency response estimation can either use one simulation per frequency or one simulation for all frequencies. In the **Simulation order** drop-down list, select the default option **Single simulation for all frequencies**. If you have Parallel Computing Toolbox™ software, you can speed up the frequency response estimation by choosing **One frequency per simulation** and

enabling the parallel pool for estimation. To enable the parallel pool, on the **Estimation** tab, click **More Options**, then in the dialog box, select **Use parallel pool during estimation**.

To specify the frequencies at which to estimate the plant response, click the + icon.



In the Add frequencies dialog box, specify `15` logarithmically-spaced frequencies ranging from `50 Hz` to `5 kHz`.

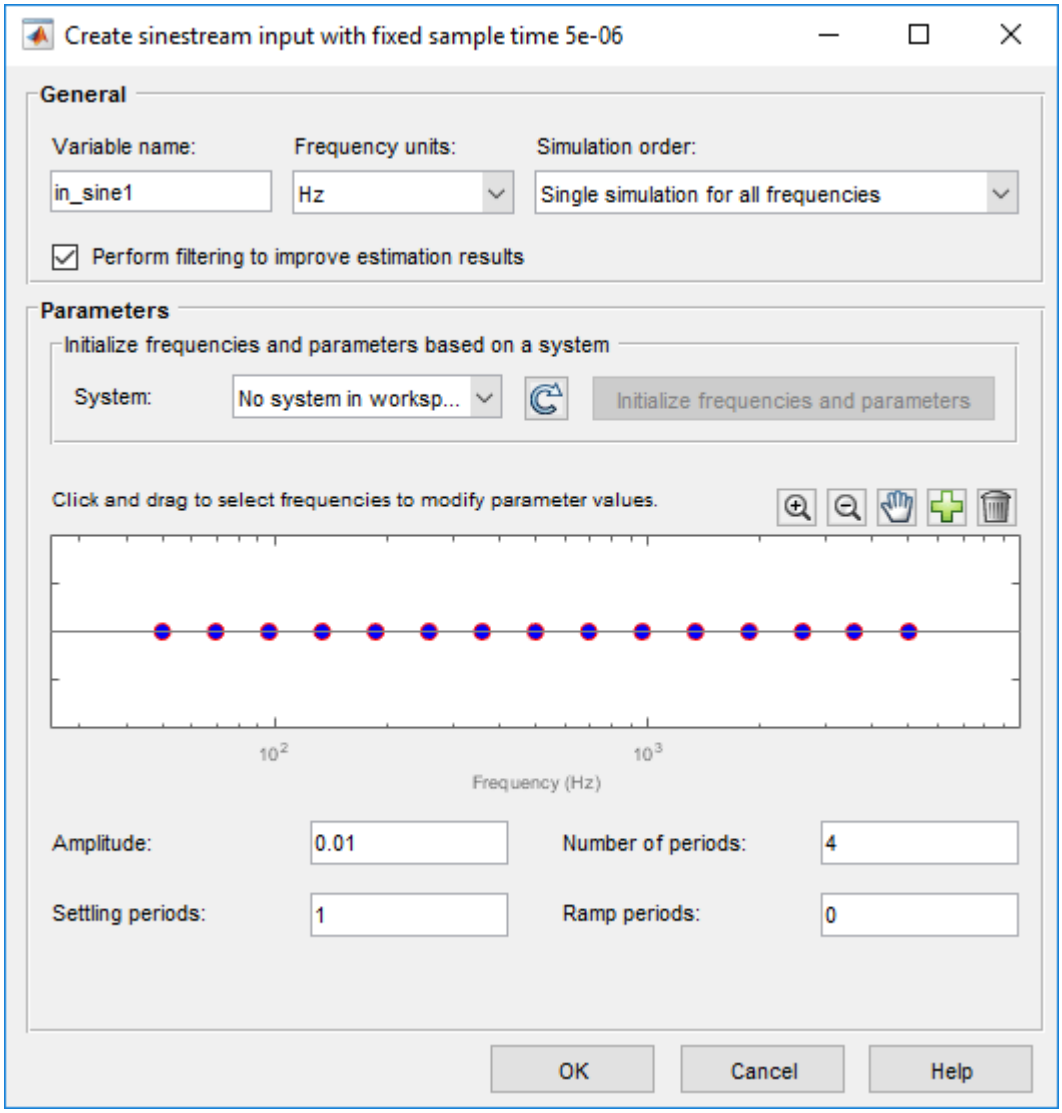


Click **OK**.

To ensure the system is properly excited, set the amplitude at all frequencies. If the input amplitude is too large, the boost converter will operate in discontinuous-current mode. If the input amplitude is too small, the sinestream will be indistinguishable from ripples in the power electronics circuits. Both situations produce inaccurate frequency response estimation results.

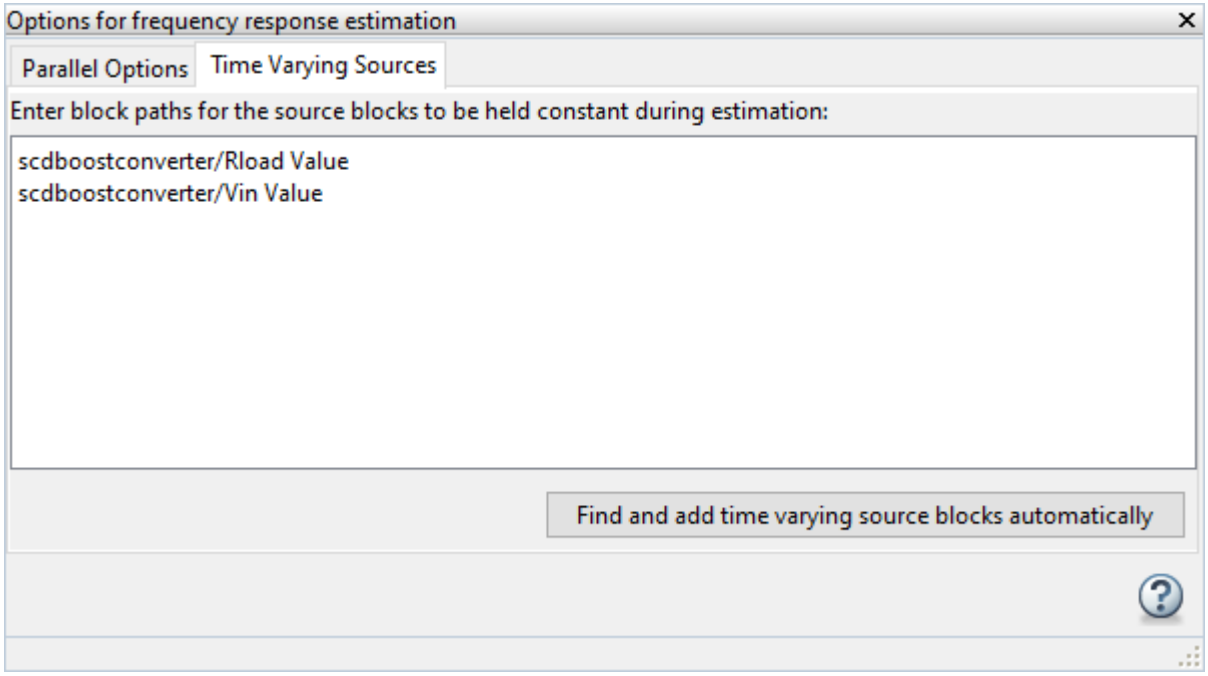
To set the amplitude, first select all the frequencies in the plot area. Then, in the **Amplitude** field, type `0.01`.

Leave all other sinestream settings at their default values.

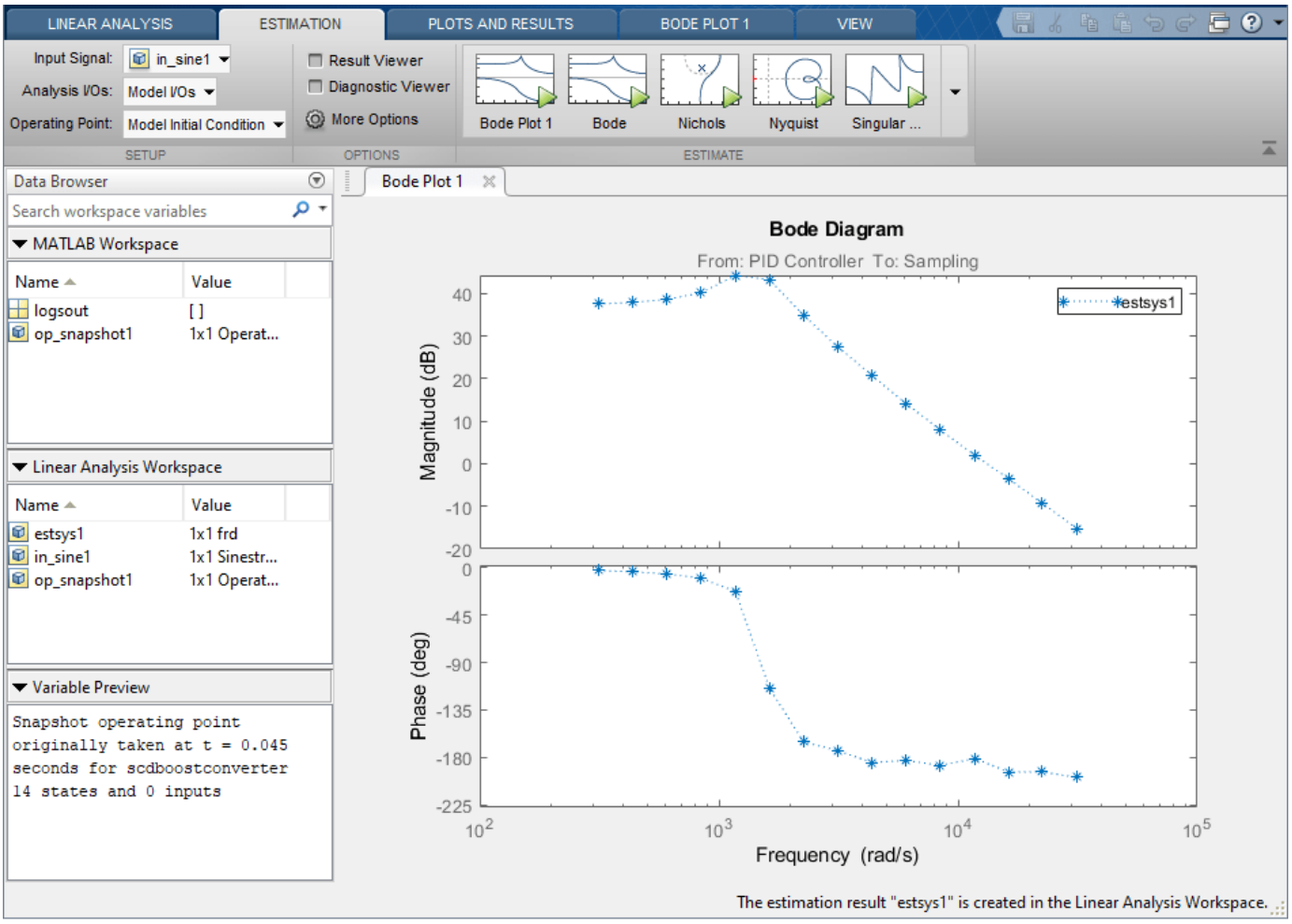


To create the sinestream signal, click **OK**.

The model has time-varying line and load disturbances modeled as step functions that will interfere with the frequency response estimation. To hold these disturbances constant during the simulation, click **More Options**. Then, in the Options for frequency response estimation dialog box, on the **Time Varying Sources** tab, click **Find and add time varying source blocks automatically**.

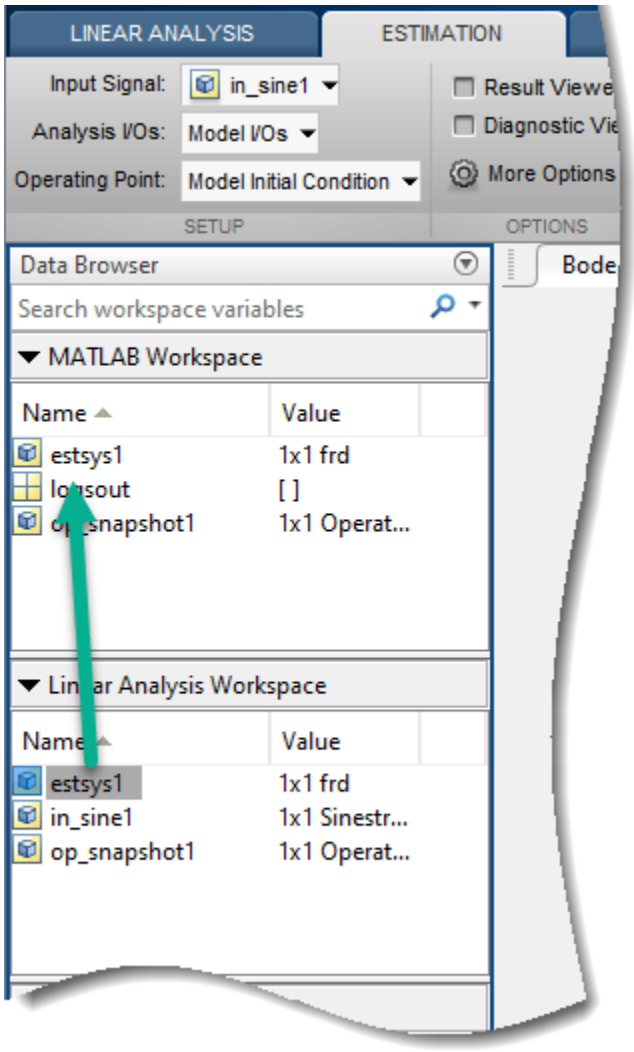


To estimate and plot the frequency response, on the **Estimation** tab, click **Bode**.



The software estimates the frequency response and displays the result in **Bode Plot 1**. The frequency response is plotted using discrete points and shows the peak response between 1200 and 1600 rad/s.

To tune your PID controller, you must export the frequency response to the MATLAB workspace. In the **Data Browser**, drag `estsys1` from the **Linear Analysis Workspace** to the **MATLAB Workspace**.



Specify Controller Structure

Before tuning a PID Controller block using **PID Tuner**, you must first specify your controller structure. To do so, double-click the PID Controller block. Then, specify the following controller parameters:

- **Controller**
- **Form**
- **Time domain**
- **Discrete-time settings**
- Other settings, such as the controller initial conditions, output saturation levels, and anti-windup configuration

For this example, use the current controller configuration; that is, a discrete-time parallel-form PID controller without anti-windup.

Using the **PID Tuner**, you can tune the parameters of the following controller blocks:

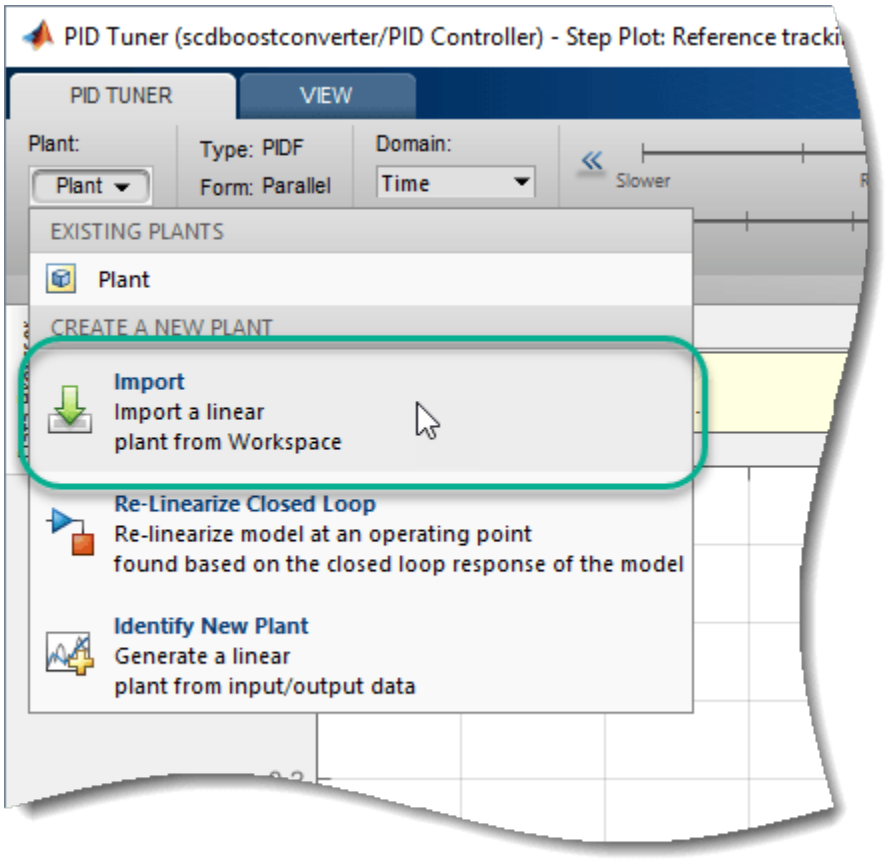
- PID Controller
- PID Controller (2DOF)
- Discrete PID Controller
- Discrete PID Controller (2DOF)

If your model uses the Simscape Electrical Discrete PI Controller (Simscape Electrical) block or Discrete PI Controller with Integral Anti-Windup (Simscape Electrical) block, you must replace this block with a PID Controller block before tuning.

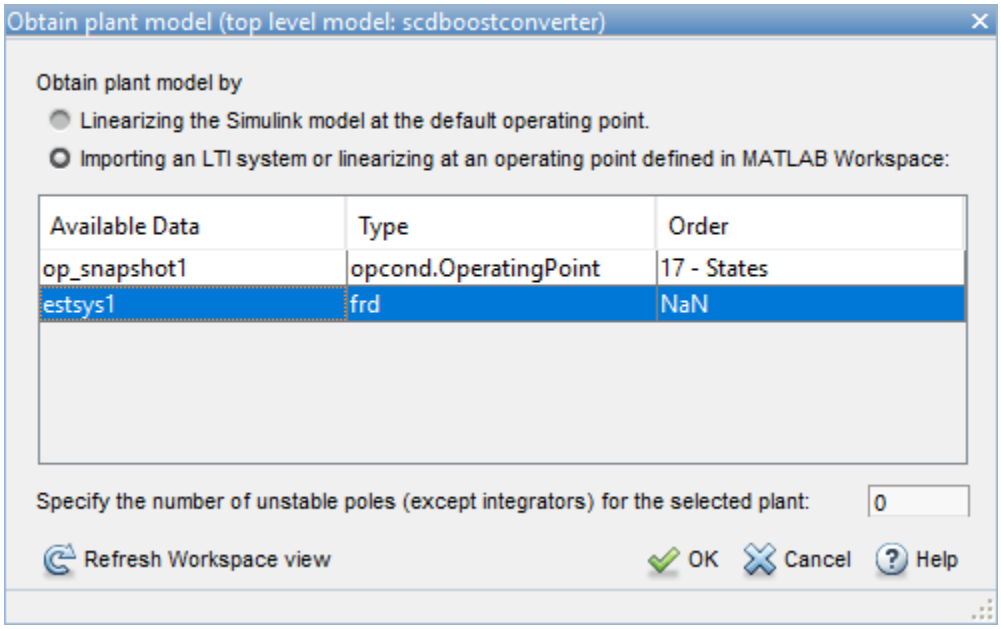
Tune Controller

To open the **PID Tuner**, click **Tune**. When **PID Tuner** first opens, it attempts to linearize the model. Due to the PWM components, the model analytically linearizes to zero.

For this example, you tune the controller using the estimated frequency response data as your plant model. To import the frequency response data, on the **PID Tuner** tab, click **Plant**, and then, under **Create a New Plant**, click **Import**.

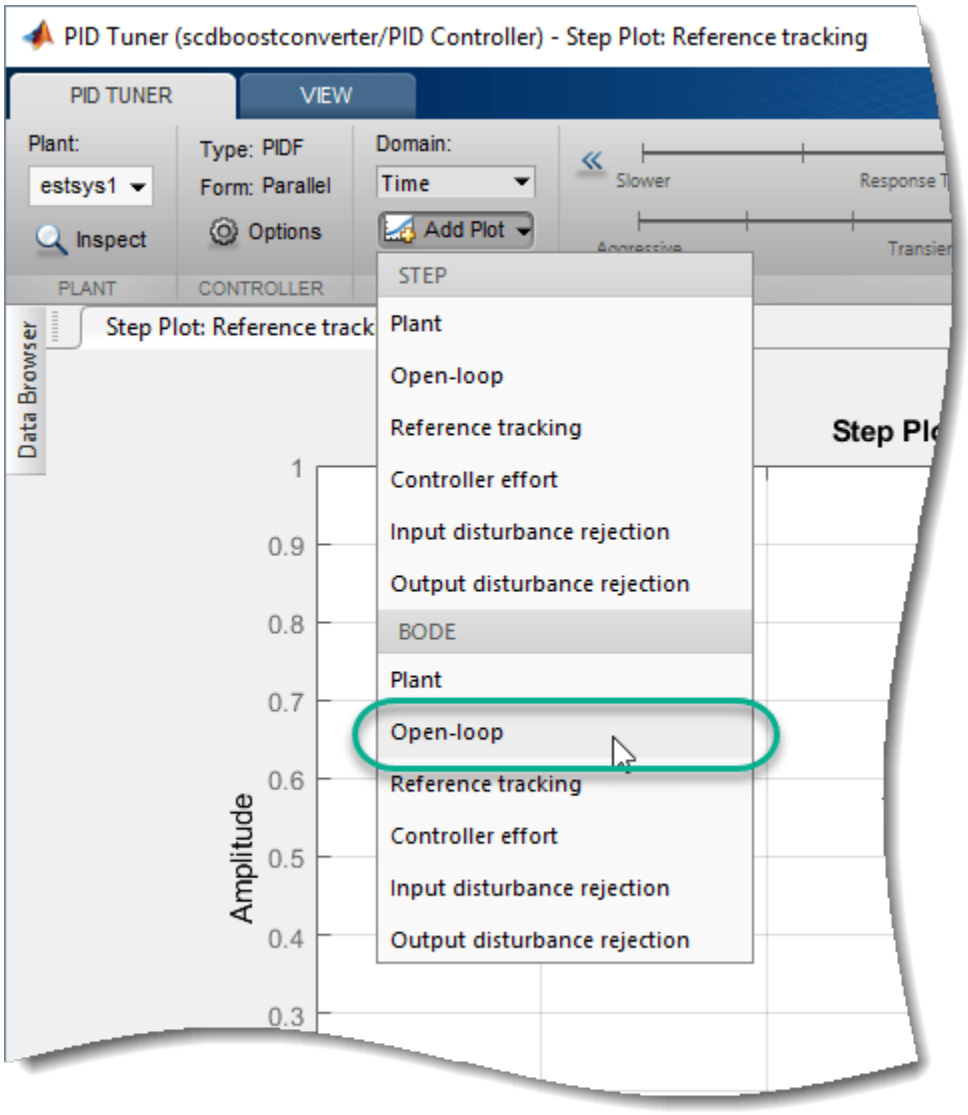


In the **Obtain plant model** dialog box, select **Importing an LTI System**, and in the table, select `estsys1`.



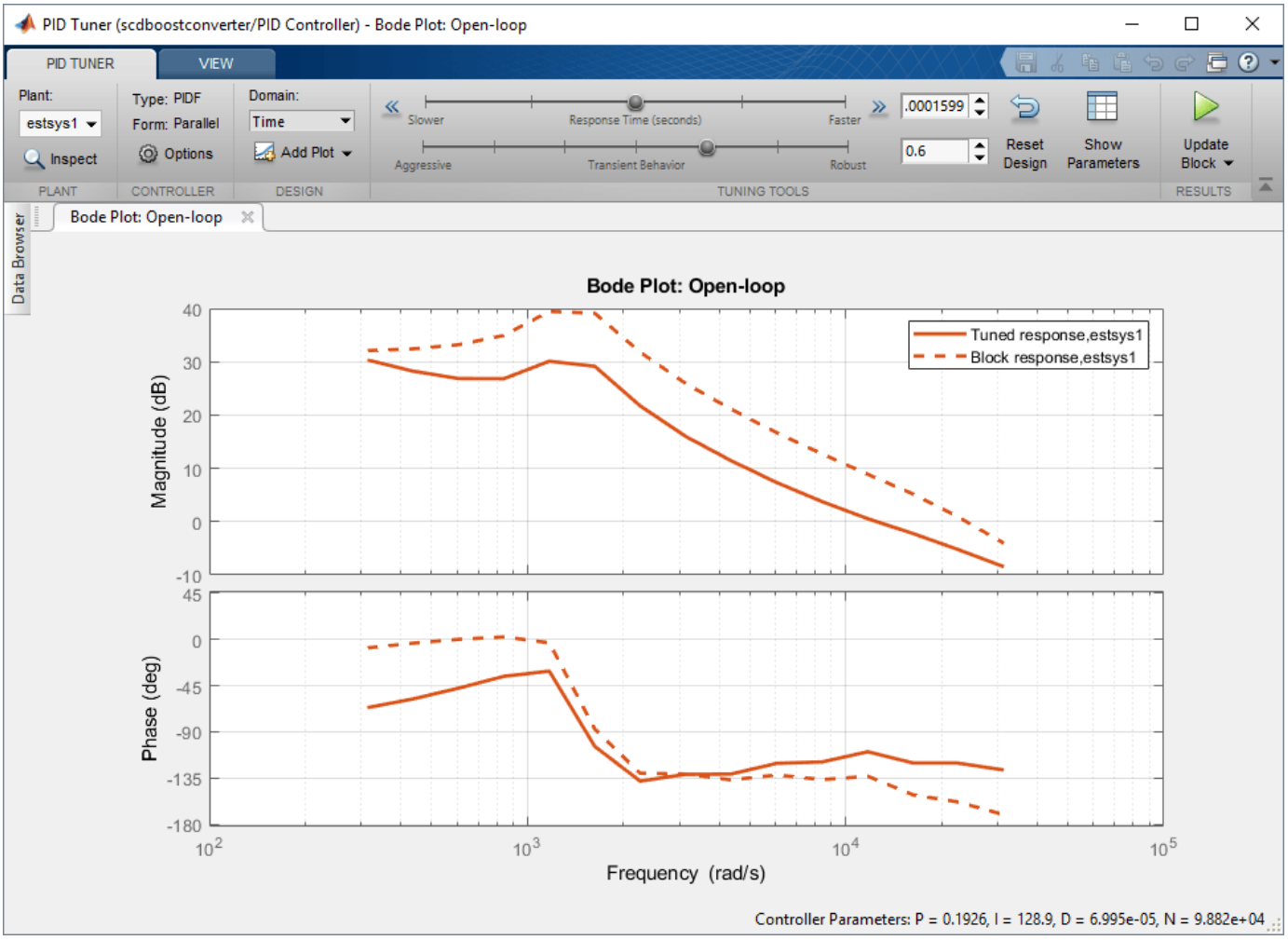
Click **OK**.

Since you are using an estimated frequency response, **PID Tuner** cannot plot a step response. To view the frequency response, click **Add Plot**, and under **Bode**, click `Open-loop`.



Close the **Step Plot** document.

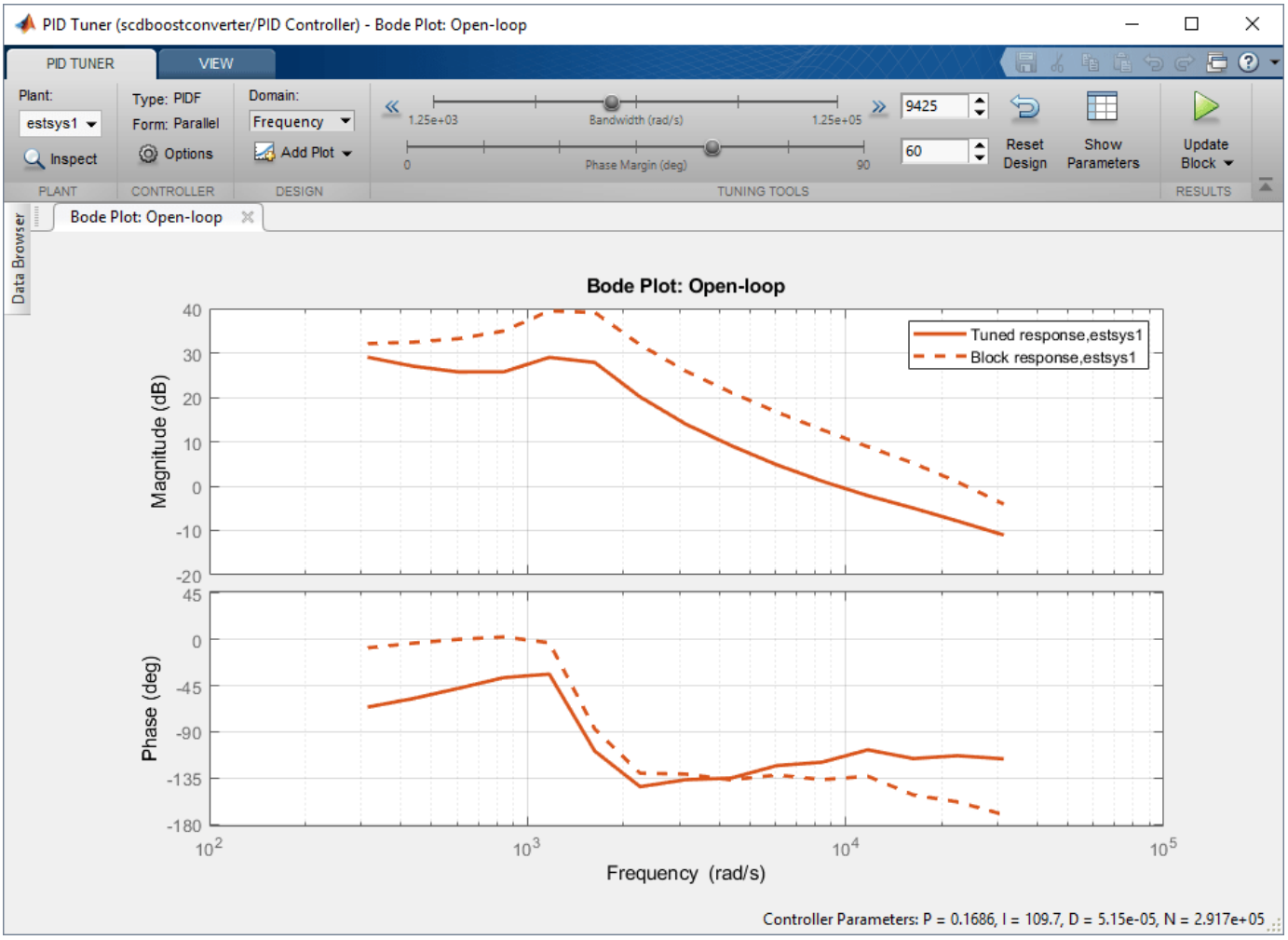
The Bode plot shows a block response (dashed line) and a tuned response (solid line). The block response is the open-loop response for the current PID gains in the PID Controller block. The tuned response is the open-loop response using the tuned PID gains in **PID Tuner**.



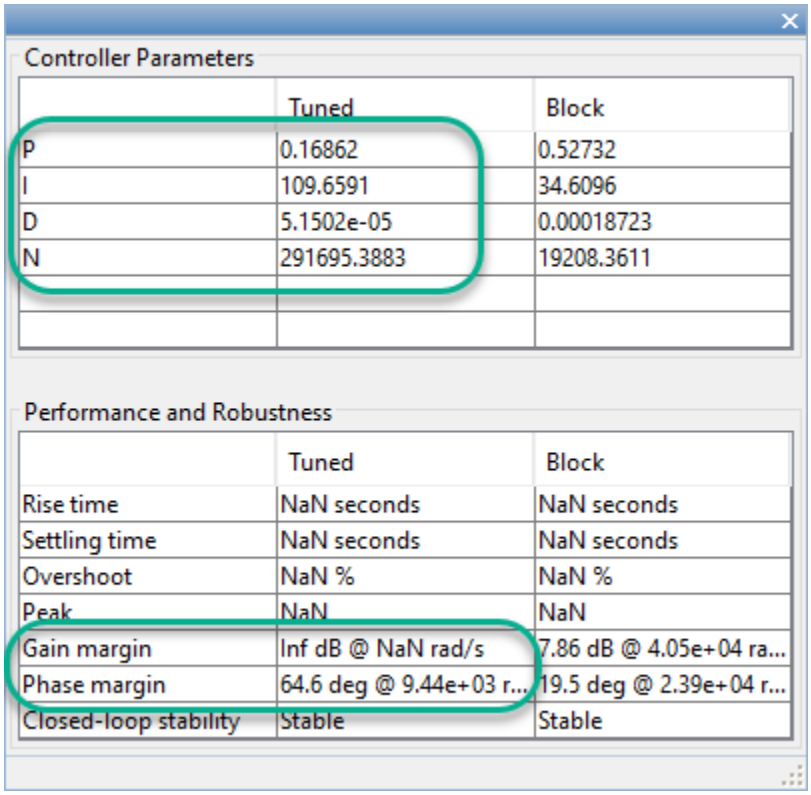
To tune the controller in terms of bandwidth and phase margin, design the controller in the frequency domain. In the **Domain** drop-down list, select `Frequency`.

For this example, set the **Bandwidth** and **Phase Margin** to 9425 rad/s (1.5 kHz) and 60 deg, respectively, according to the design criteria specified in [1].

PID Tuner selects controller parameters that meet these design specifications.



To view the tuned controller parameters and performance metrics, including the gain and phase margins, click **Show Parameters**. The tuned result has an infinite gain margin and 65 deg phase margin at about 9425 rad/s.



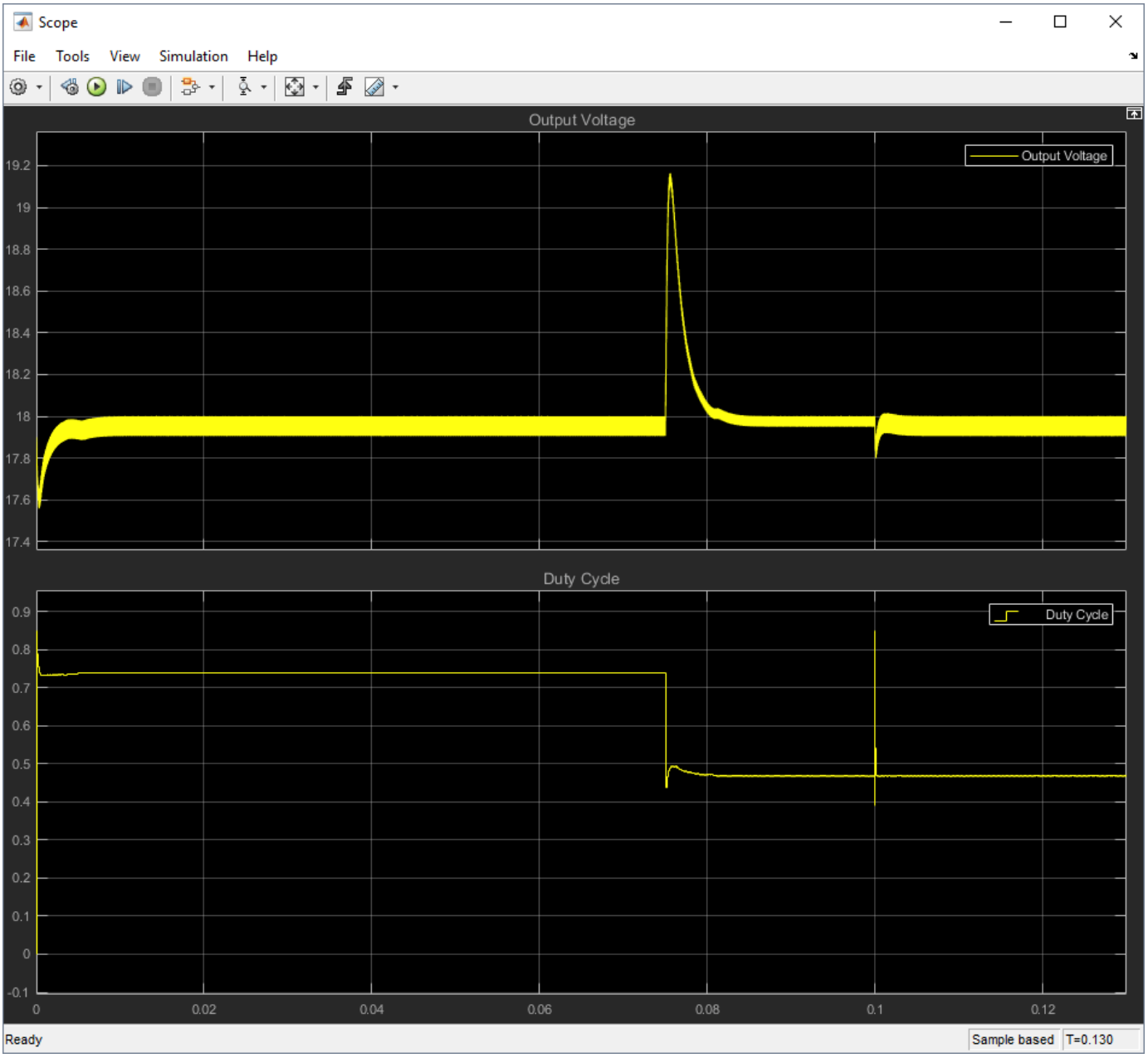
To update the PID Controller block with the tuned gains, click **Update Block**.

Validate Controller

You can examine the tuned controller performance using a simulation with line and load disturbances. To examine the controller dynamic performance, the Simulink model uses the following disturbances:

- Line disturbance at t = 0.075 sec, which increases the input voltage, Vin, from 5V to 10V.
- Load disturbance at t = 0.1 sec, which increases the load resistance, $Rload$ from 3 ohms to 6 ohms.

Simulate the model.



The controller rejects the line and load disturbances well.

References

[1] Lee, S. W. "Practical Feedback Loop Analysis for Voltage-Mode Boost Converter." Application Report No. SLVA633. Texas Instruments. January 2014. www.ti.com/lit/an/slva633/slva633.pdf

See Also

PID Tuner

More About

- “Design Controller for Power Electronics Model Using Simulated I/O Data” on page 7-94

Design Controller for Power Electronics Model Using Simulated I/O Data

This example shows how to design a PID controller for a power electronics system modeled in Simulink using Simscape Electrical components.

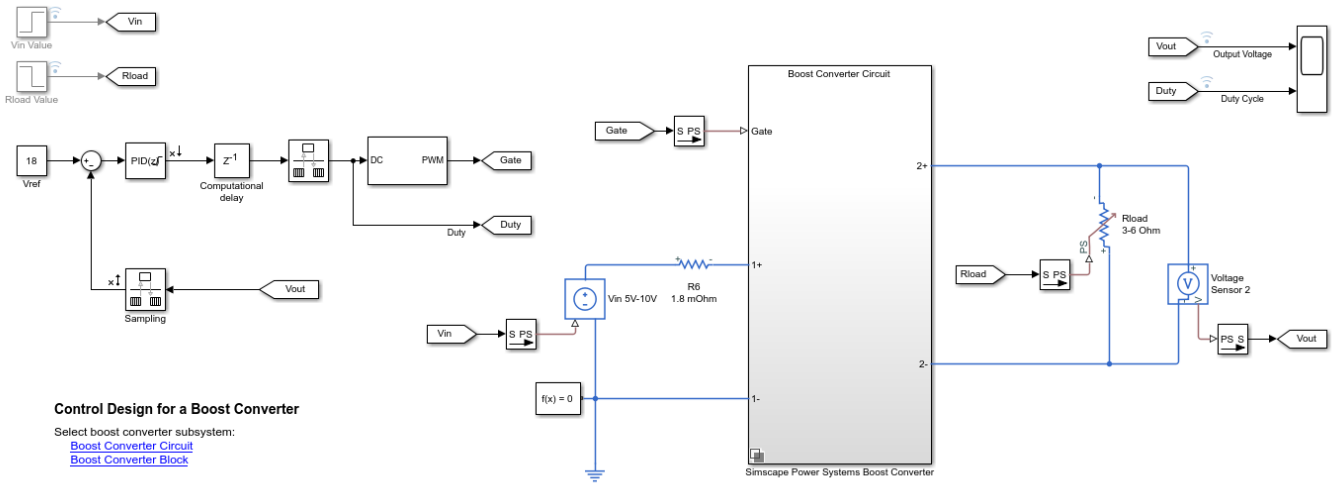
Many power electronics systems cannot be linearized because they use high-frequency switching components, such as pulse-width modulation (PWM) generators. However, most Simulink Control Design PID tuning tools design PID gains based on a linearized plant model. To obtain such a model for a power electronics model that cannot be linearized, you can:

- Estimate the parameters of a linear model of the plant using System Identification Toolbox software as shown in this example.
- Estimate the plant frequency response over a range of frequencies. For an example, see “Design Controller for Power Electronics Model Using Frequency Response Data” on page 7-76.

Boost Converter Model

This example uses a boost converter model as an example of a power electronics system. A boost converter circuit converts one DC voltage to another, typically higher, DC voltage by controlled chopping or switching of the source voltage.

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



In this model, a MOSFET driven by a pulse-width modulation (PWM) signal is used for switching. The output voltage *Vout* should be regulated to the reference value *Vref*. A digital PID controller adjusts the PWM duty cycle, *Duty*, based on the voltage error signal. For this example, you estimate a linear model from the PWM duty cycle to the load voltage *Vout*.

Simscape Electrical software contains predefined blocks for many power electronics systems. This model contains a variant subsystem with two versions of the boost converter model:

- Boost converter circuit constructed using electrical power components. The parameters of the circuit components are based on [1].

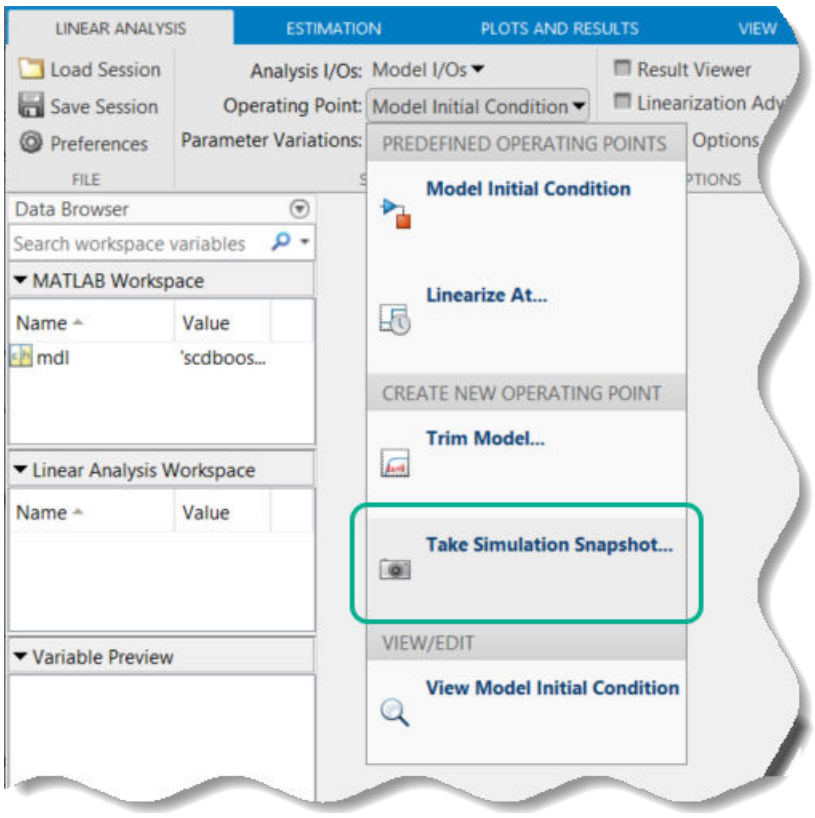
- Boost converter block configured to have the same parameters as the boost converter circuit. For more information on this block, see Boost Converter.

Find Model Operating Point

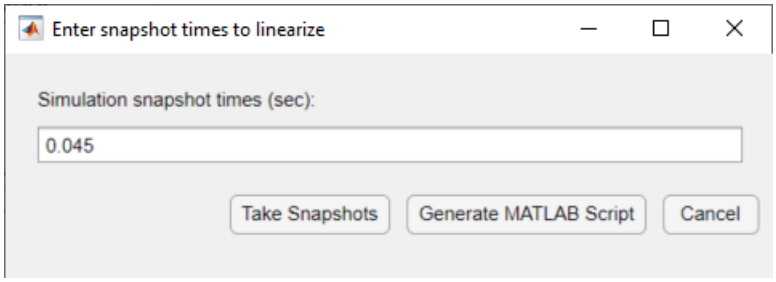
To design a controller for the boost converter, you must first determine the steady-state operating point at which you want the converter to operate. For more information on finding operating points, see “Find Steady-State Operating Points for Simscape Models” on page 1-104. For this example, use an operating point estimated from a simulation snapshot.

To find the operating point, use the **Model Linearizer** app. To open **Model Linearizer**, in the Simulink model window, in the **Apps** gallery, click **Model Linearizer**.

In the **Model Linearizer**, on the **Linear Analysis** tab, in the **Operating Point** drop-down list, select **Take Simulation Snapshot**.

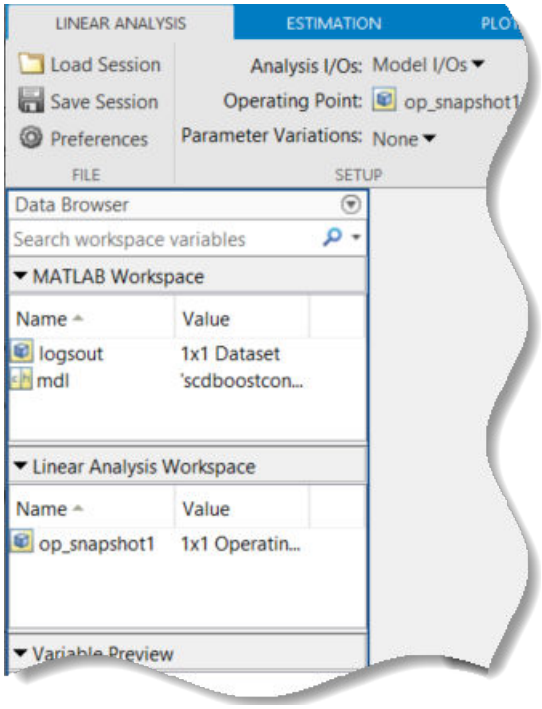


In the Enter snapshot times to linearize dialog box, in the **Simulation snapshot times** field, enter `0.045`, which is enough time for the closed-loop system to reach steady state.

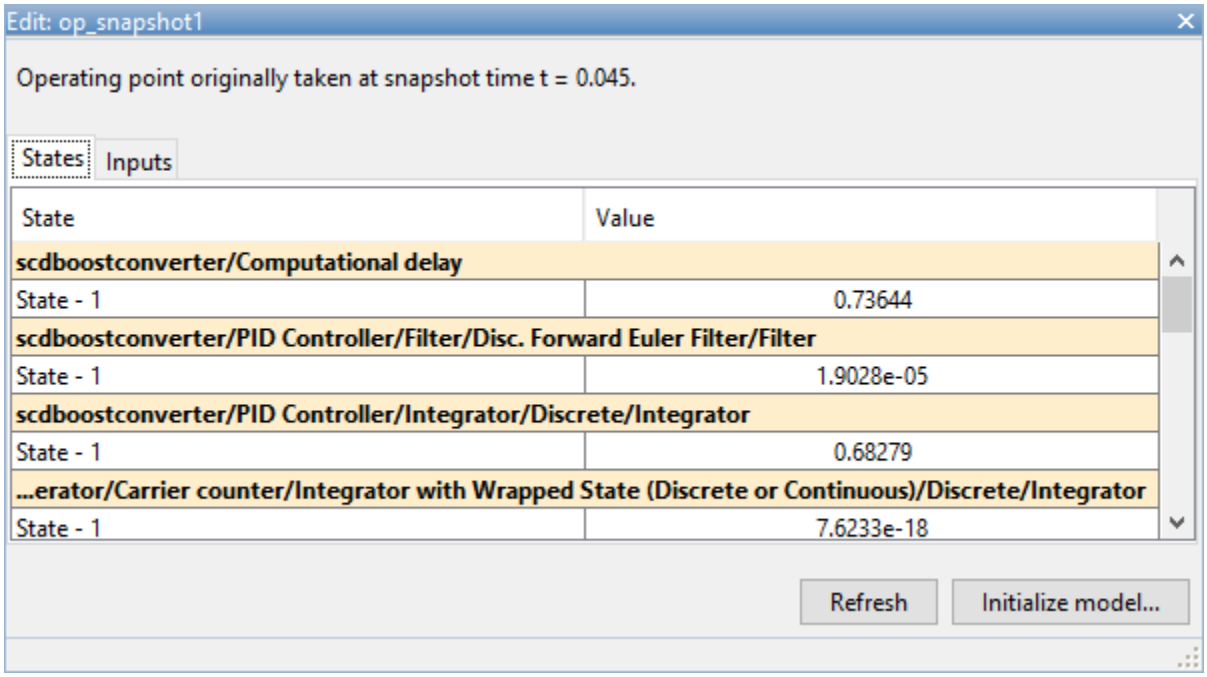


Click **Take Snapshots**.

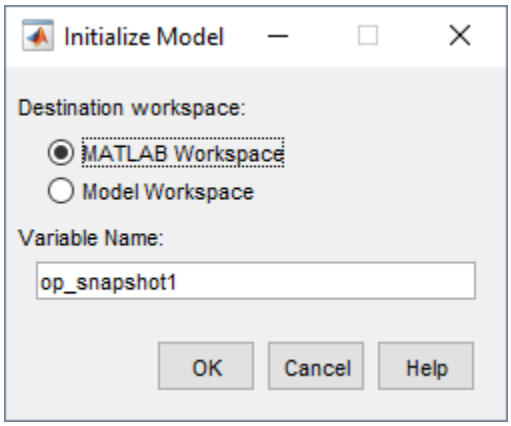
The software simulates the model and creates an operating point that contains the input and state values of the model at the specified snapshot time. This operating point, `op_snapshot1`, is added to the **Linear Analysis Workspace**.



To initialize the model with the computed operating point, double-click `op_snapshot1`.



In the Edit dialog box, click **Initialize model**.



In the Initialize Model dialog box, select **MATLAB Workspace**, and click **OK**. The software exports the operating point to the MATLAB workspace and initializes the model with the inputs and states in the operating point.

Specify Controller Structure

Before tuning a PID controller block using **PID Tuner**, you must specify your controller structure. To do so, double-click the controller block. Then, specify the following controller parameters:

- **Controller**
- **Form**
- **Time domain**
- **Discrete-time settings**

- Other settings such as the controller initial conditions, output saturation levels, and anti-windup configuration.

For this example, use the current controller configuration; that is, a discrete-time parallel-form PID controller without anti-windup.

Using **PID Tuner**, you can tune the parameters of the following controller blocks:

- PID Controller
- PID Controller (2DOF)
- Discrete PID Controller
- Discrete PID Controller (2DOF)

If your model uses the Simscape Electrical Discrete PI Controller block or Discrete PI Controller with Integral Anti-Windup block, you must replace this block with a Discrete PID Controller block before tuning.

Identify Plant Model

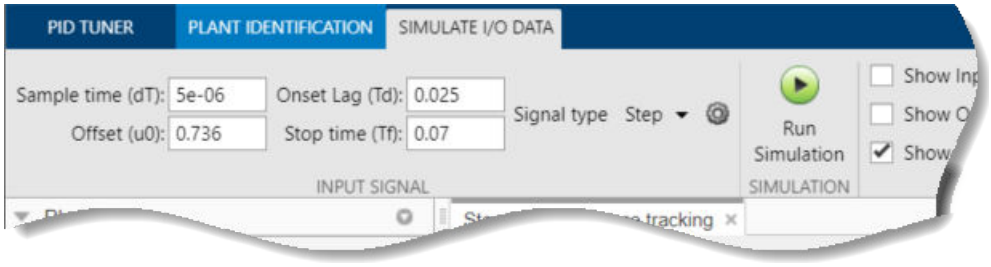
To open the **PID Tuner**, click **Tune**. When **PID Tuner** first opens, it attempts to linearize the model. Due to the PWM components, the model analytically linearizes to zero.

To obtain a linear plant model, on the **PID Tuner** tab, click **Plant**, and then under **Create a New Plant**, click **Identify New Plant**.

To identify a plant model, first obtain input/output data by simulating your model. On the **Plant Identification** tab, click **Get I/O Data > Simulate Data**. For plant identification, you must specify a finite value for the Simulink model stop time.

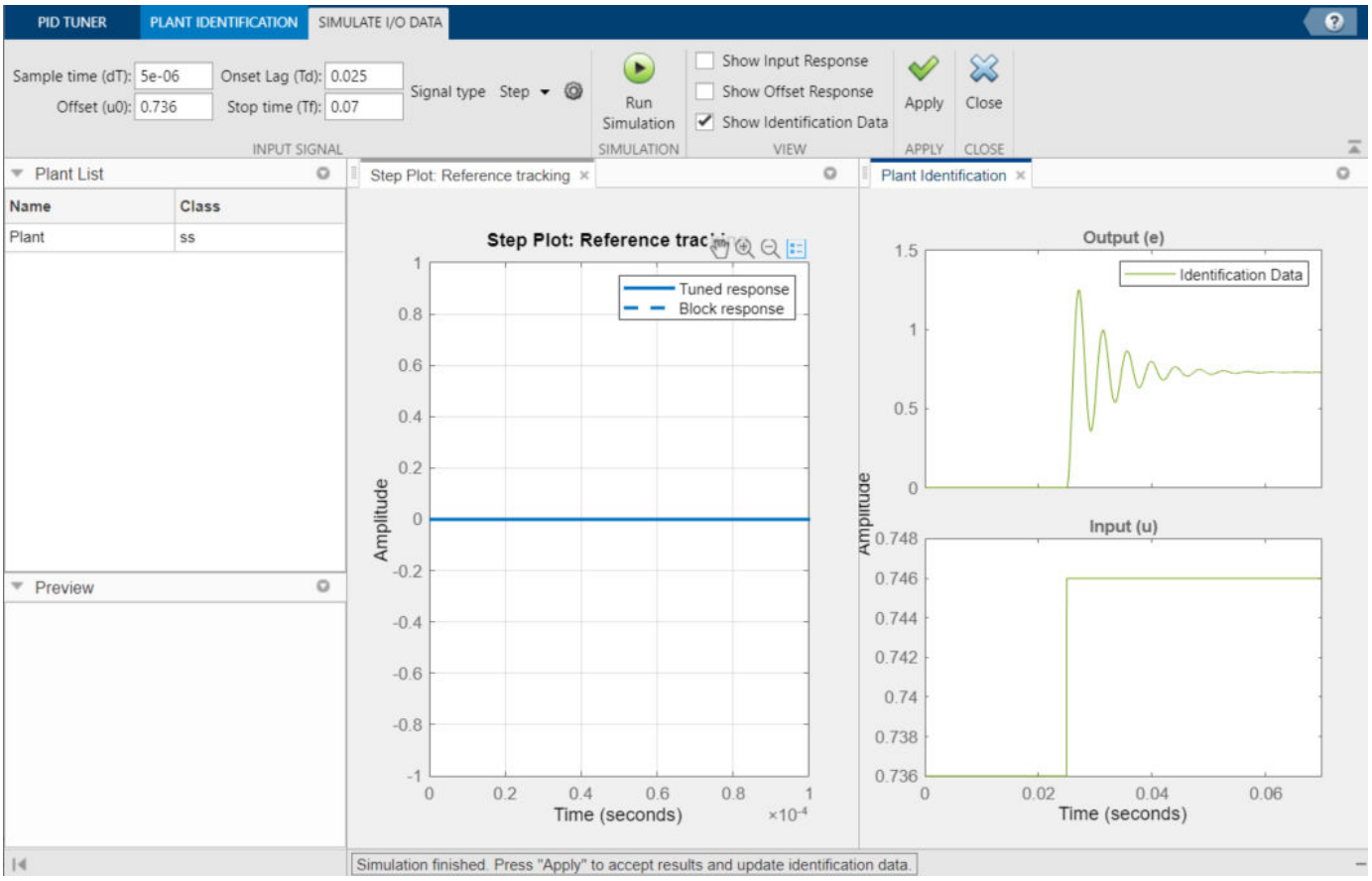
On the **Simulate I/O Data** tab, configure the input signal with the following settings.

- **Signal Type** of `Step`.
- **Sample Time** of `5e-06`.
- **Onset Lag** of `0.025`, which is enough time for the plant to reach steady state.
- **Stop Time** of `0.07`, which is enough time for the plant output to return to steady state after the step input.
- **Offset** of `0.736`, which is the value of the PID Controller block output at the computed operating point. For this model, the offset corresponds to the value of the state in the Computational delay block. If you do not have such a corresponding state in your model, you can attach a scope to the output of the PID Controller block and simulate the model at the computed operating point.



To specify the step amplitude, click . Then, in the Step Input Specifications dialog box, in the **Amplitude** field, type `0.01`. This value is large enough to sufficiently excite the system and small enough to prevent the controller from entering discontinuous-current mode.

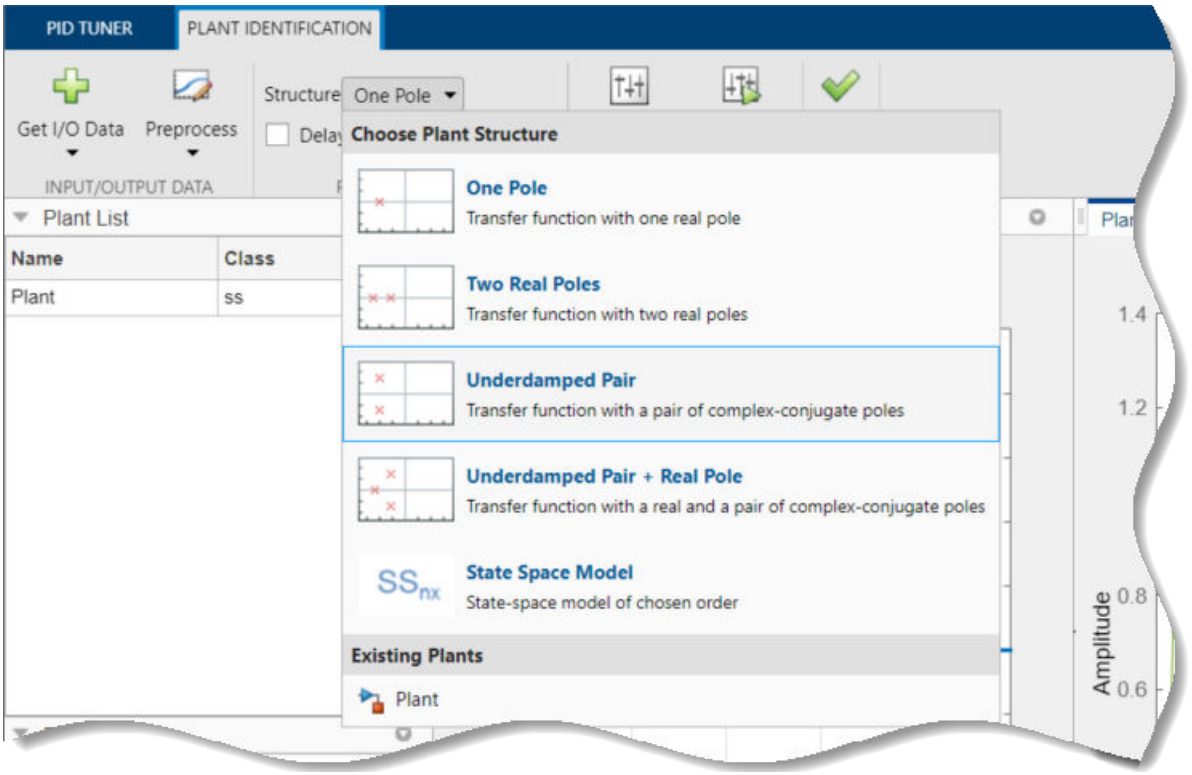
Click **Run Simulation**. To obtain the input/output response of the plant, **PID Tuner** injects the specified input signal at the output of the PID Controller block and measures the corresponding output response at the input of the controller. The software runs two simulations, an offset response without the input signal and an input response with the input signal. The difference between these responses is the output response.



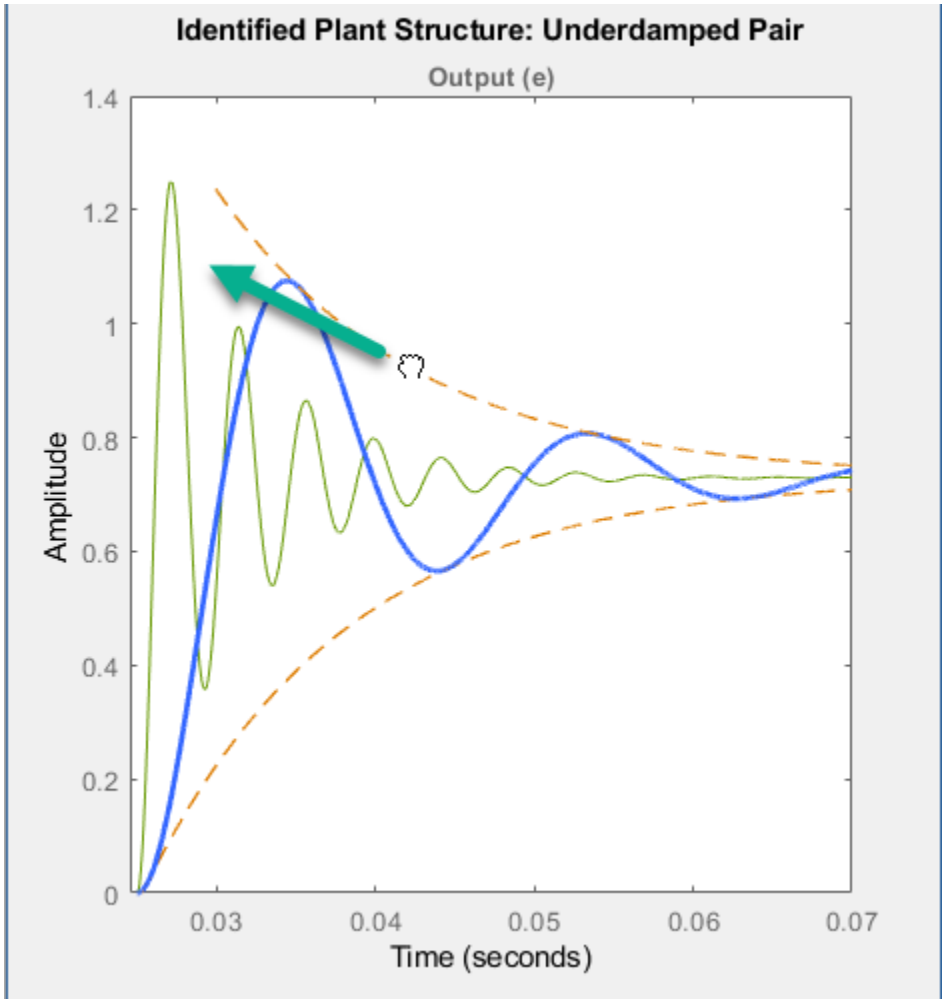
In the **Plant Identification** figure, the **Input** plot shows the specified input signal, and the **Output** plot shows the corresponding output response.

To use this simulated input/output data, click **Apply**. Then, to close the **Simulate I/O Data** tab, click **Close**.

On the **Plant Identification** tab, select the plant structure to identify based on your knowledge of the plant and the appearance of the output step response. For this example, the output response looks like an underdamped second-order response. In the **Structure** drop-down list, select **Underdamped Pair**.



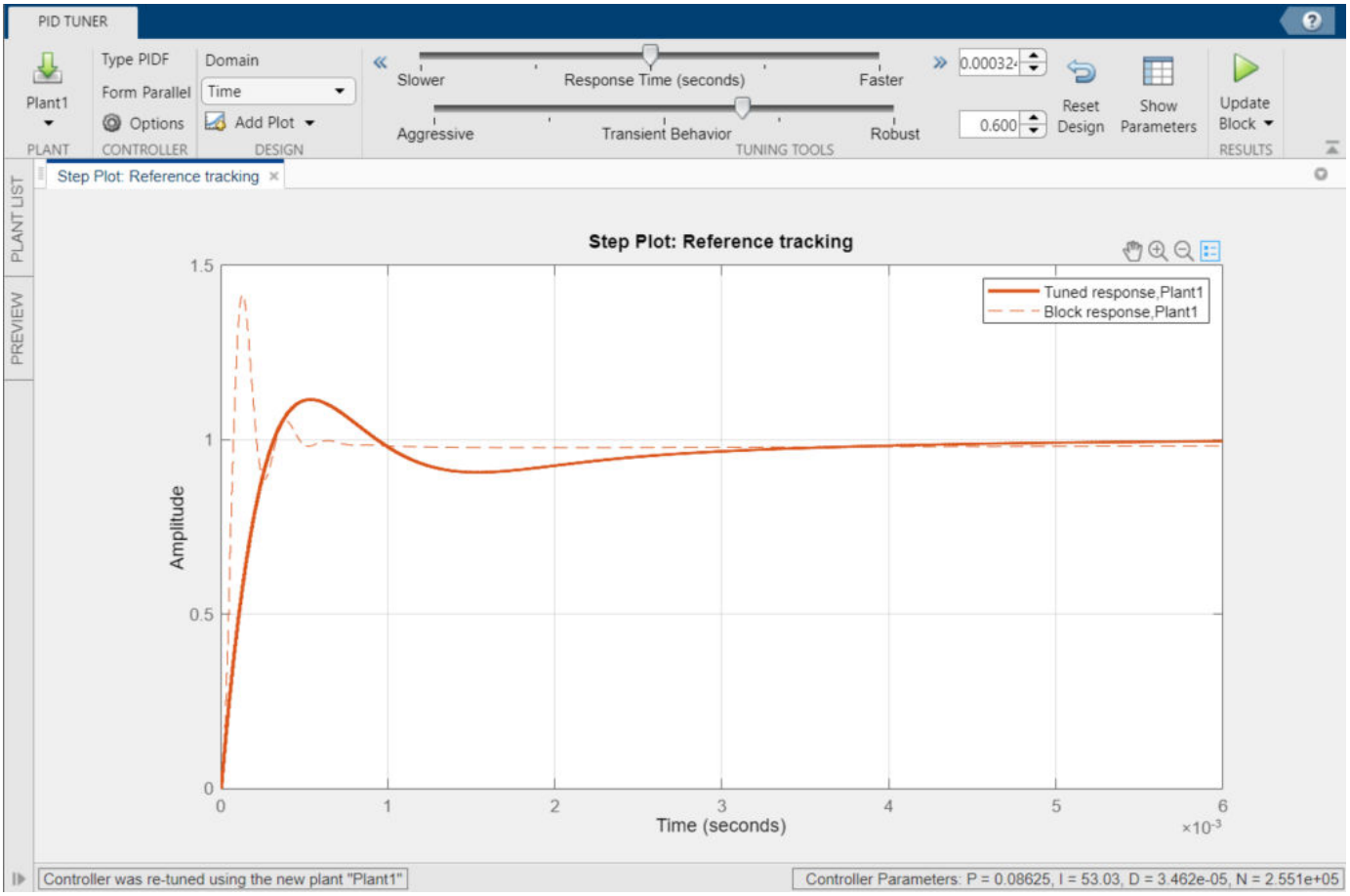
To obtain a rough approximation of the identified plant, in the **Identified Plant Structure** plot, drag the dashed lines that correspond to the envelope of the step response. Adjust the response so that it approximates the output response.



To fine-tune the approximate response, click **Auto Estimate**. The software estimates the parameters of the identified plant model using the current parameters as an initial guess.

The Plant Identification Progress dialog box shows the results of the estimation process. For this example, the fit to the estimation data is greater than 98%. To use this identified plant, on the **Plant Identification** tab, click **Apply**.

The **PID Tuner** updates its identified plant model, selects controller parameters to meet the tuning requirements in the **Tuning Tools** section, and plots the tuned response of this controller. To expand the plot, close the **Plant Identification** figure.



The step response shows a block response (dashed line) and a tuned response (solid line). The block response corresponds to the current PID gains in the PID Controller block. The tuned response corresponds to the tuned PID gains in **PID Tuner**.

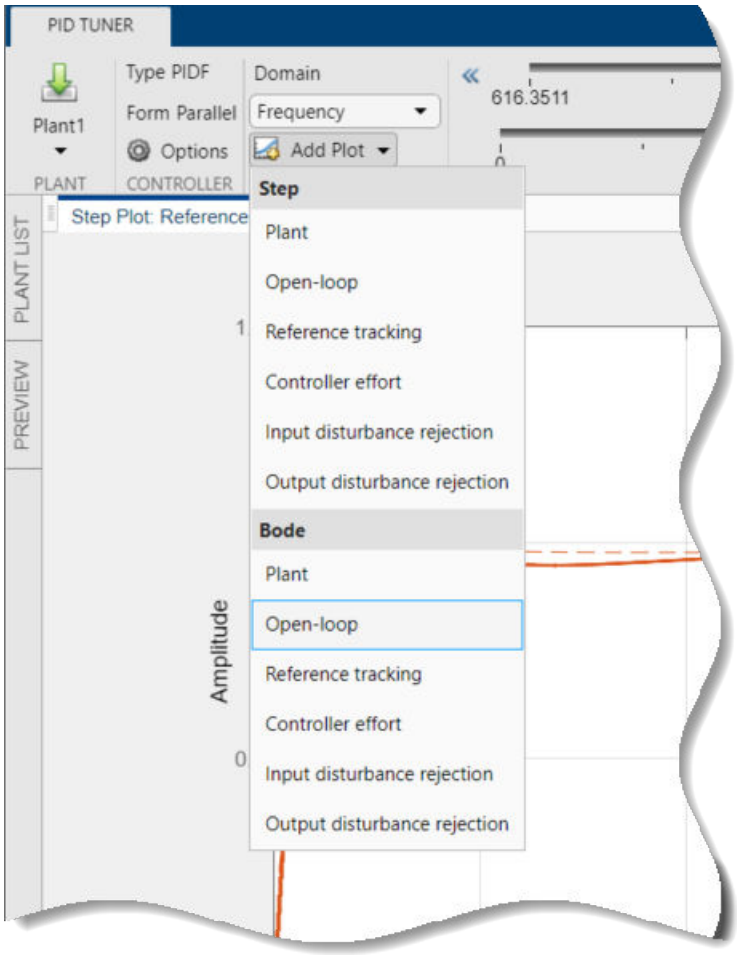
Tune Controller

To tune the controller based on bandwidth and phase margin, on the **PID Tuner** tab, in the **Domain** drop-down list, select **Frequency**.

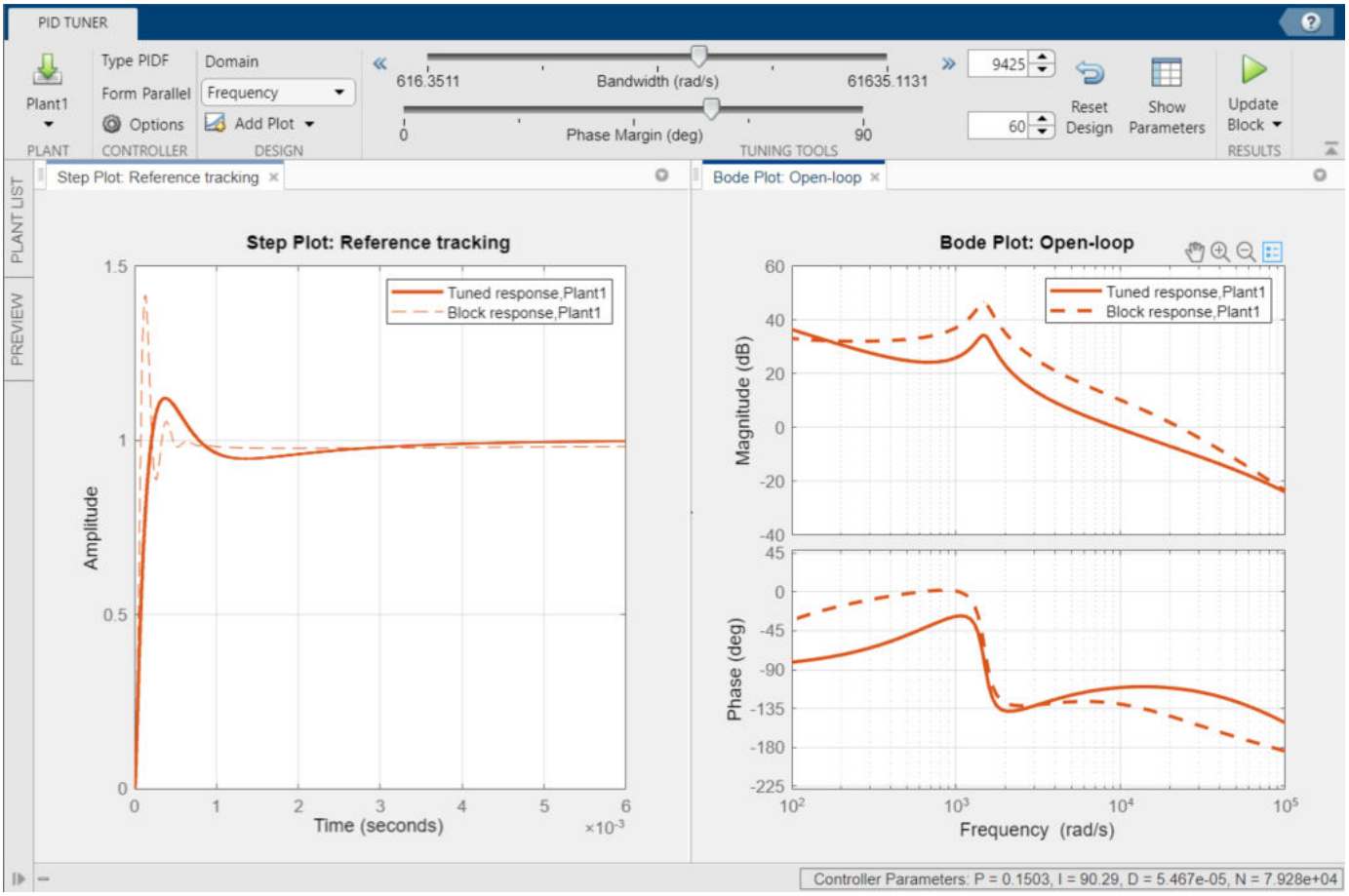
For this example, set the **Bandwidth** and **Phase Margin** to `9425` rad/s (1.5 kHz) and `60` deg, respectively, according to the design criteria specified in [1].

PID Tuner selects controller parameters that meet these design specifications.

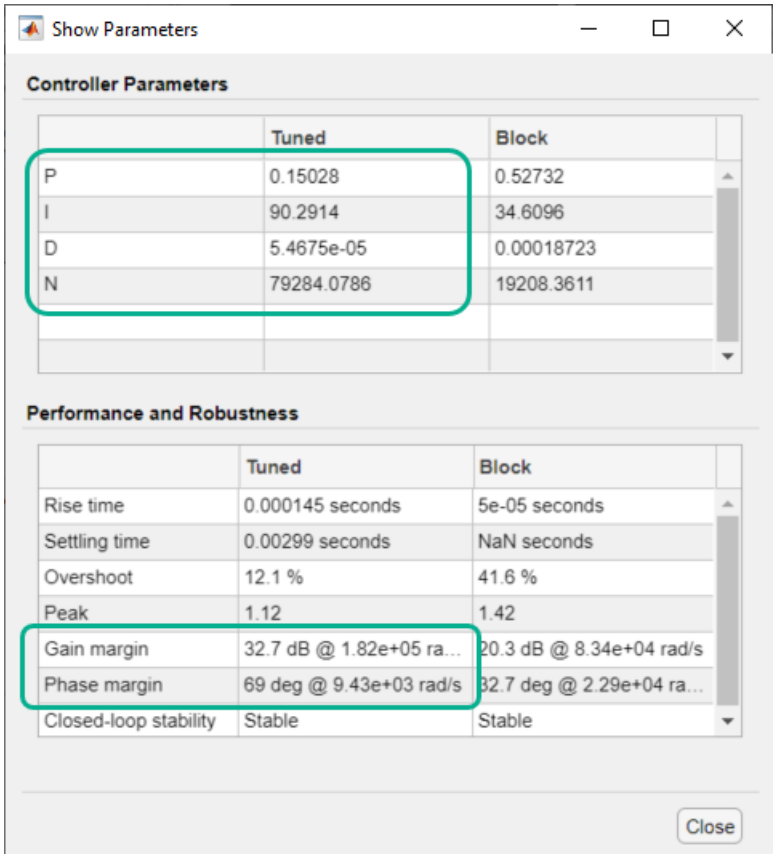
To view the frequency response of the tuned system, click **Add Plot**, and under **Bode**, click **Open-loop**.



To adjust the limits of the Bode plot, right-click the plot area, and select **Properties**. Then, in the Property Editor dialog box, on the **Limits** tab, set the axis limits.



To view the tuned controller parameters and performance metrics, including the gain and phase margins, click **Show Parameters**. The tuned result has a 32.7 dB gain margin and 69 deg phase margin at about 9425 rad/s.



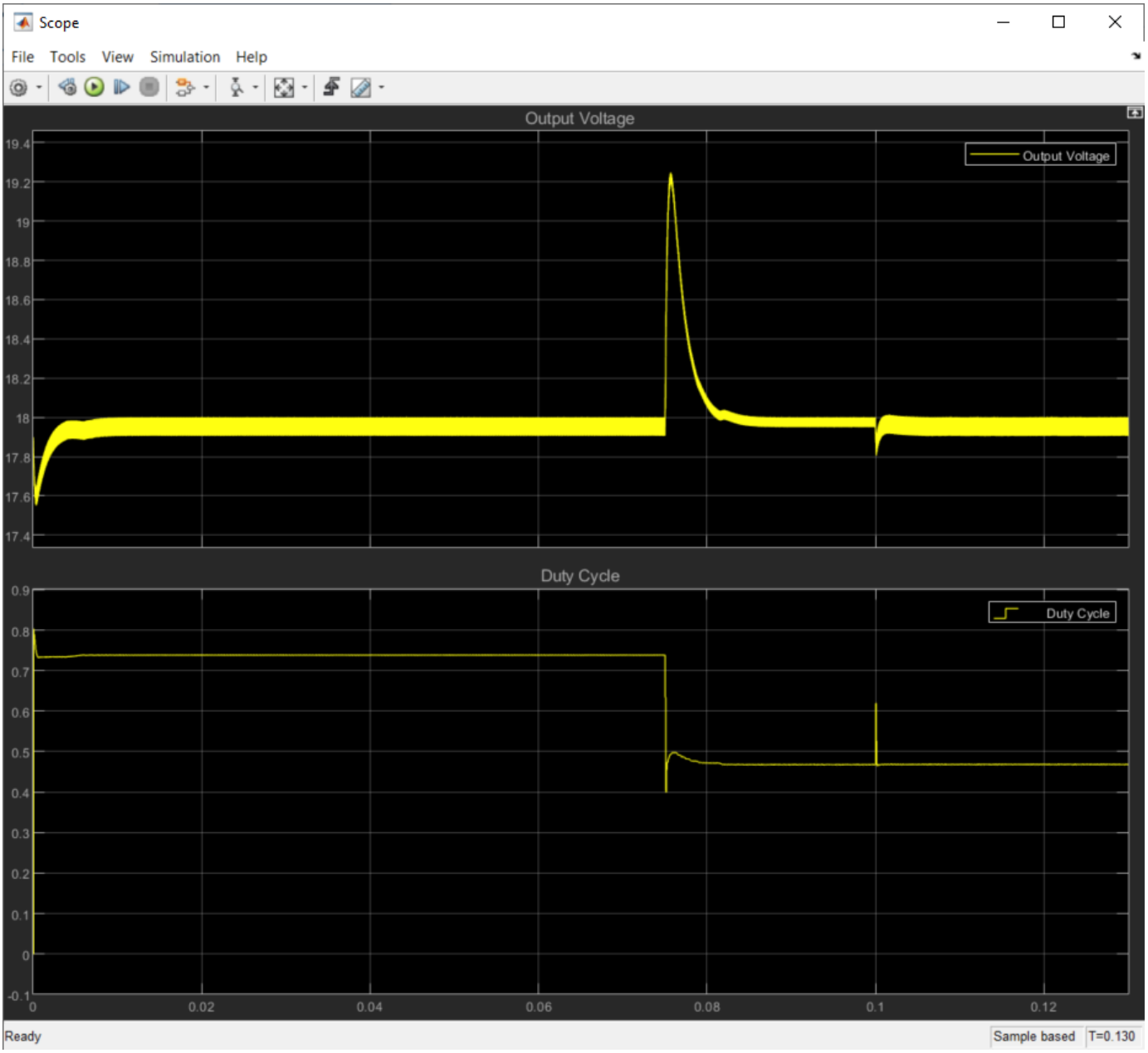
To update the PID Controller block with the tuned gains, on the **PID Tuner** tab, click **Update Block**.

Validate Controller

You can examine the tuned controller performance using a simulation with line and load disturbances. To examine the controller dynamic performance, the Simulink model uses the following disturbances:

- Line disturbance at t = 0.075 sec, which increases the input voltage, *Vin*, from 5V to 10V
- Load disturbance at t = 0.1 sec, which increases the load resistance, *Rload*, from 3 ohms to 6 ohms

Simulate the model.



The controller rejects the line and load disturbances well.

References

[1] Lee, S. W. "Practical Feedback Loop Analysis for Voltage-Mode Boost Converter." Application Report No. SLVA057. Texas Instruments. January 2014. www.ti.com/lit/an/slva633/slva633.pdf

See Also

PID Tuner

More About

- “Design Controller for Power Electronics Model Using Frequency Response Data” on page 7-76

Design PID Controller Using Simulated I/O Data

This example shows how to tune a PID controller for plants that cannot be linearized. You use **PID Tuner** to identify a plant for your model. Then tune the PID controller using the identified plant.

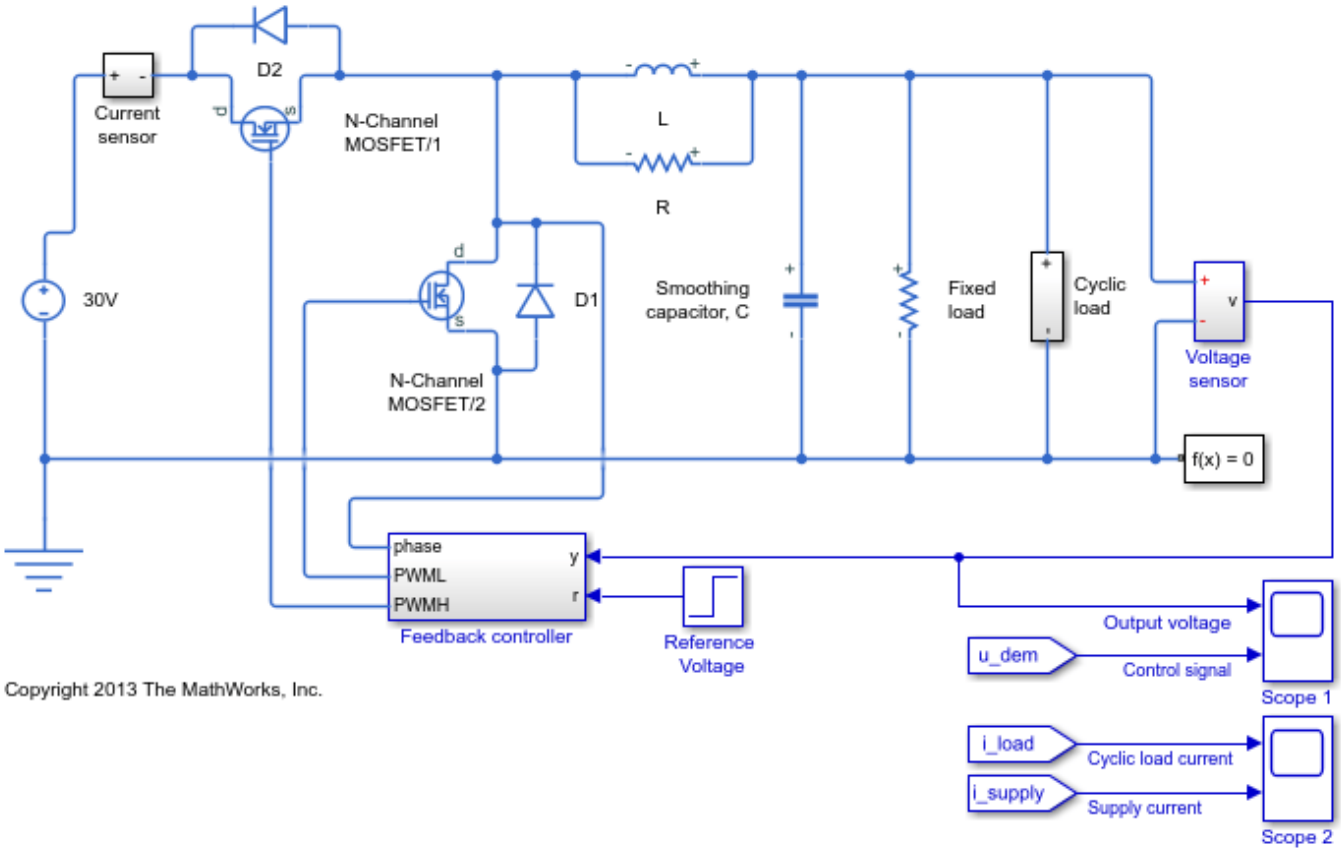
This example uses a buck converter model that requires Simscape™ Electrical™ software.

Buck Converter Model

Buck converters convert DC to DC. This model uses a switching power supply to convert a 30V DC supply into a regulated DC supply. The converter is modeled using MOSFETs rather than ideal switches to ensure that device on-resistances are correctly represented. The converter response from reference voltage to measured voltage includes the MOSFET switches. PID design requires a linear model of the system from the reference voltage to the measured voltage. However, because of the switches, automated linearization results in a zero system. In this example, using **PID Tuner**, you identify a linear model of the system using simulation instead of linearization.

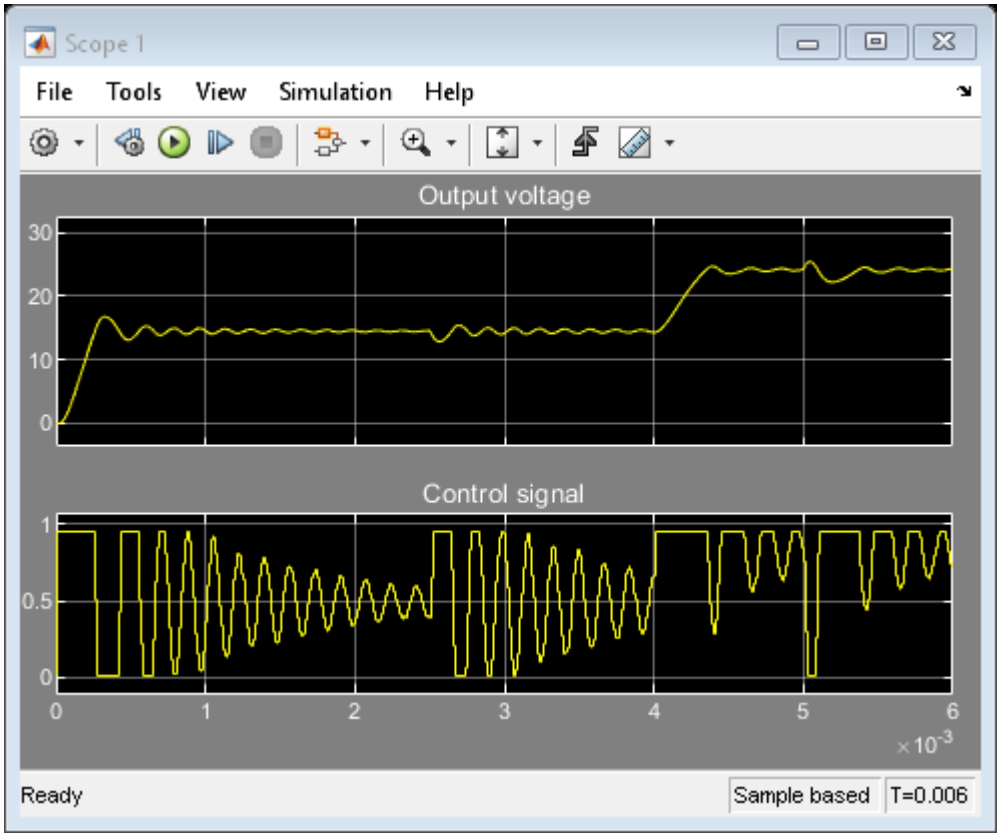
For more information on creating a buck converter model, see “Buck Converter” (Simscape Electrical).

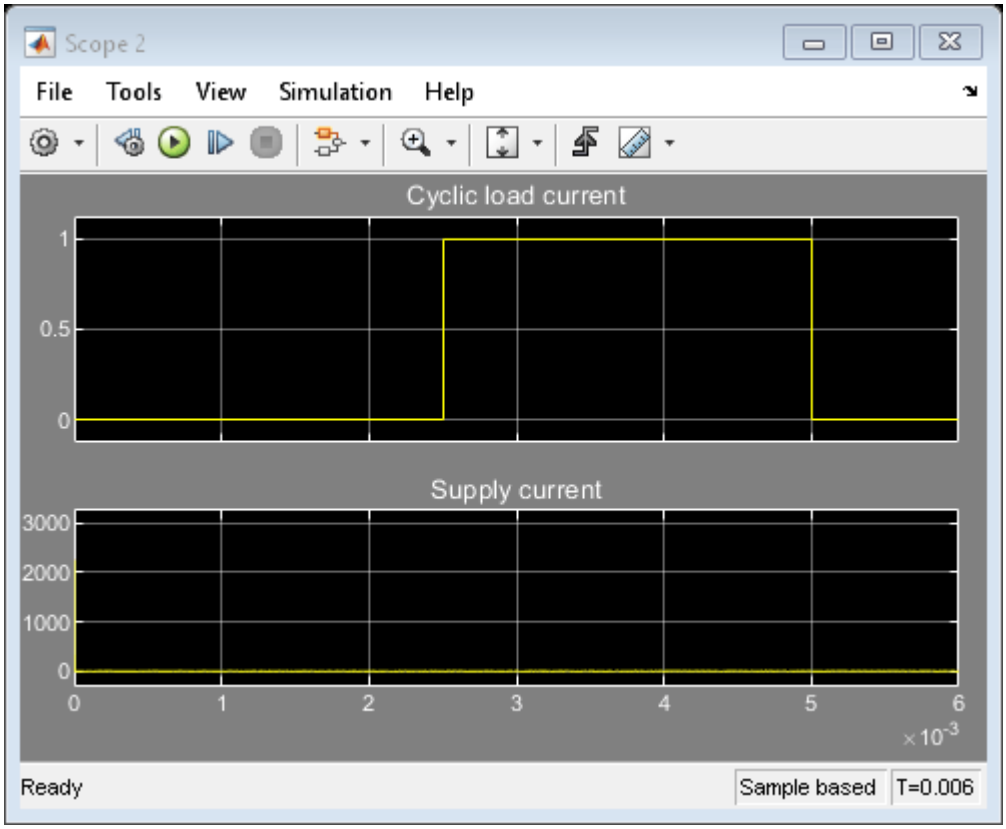
```
open_system('scdbuckconverter')
sim('scdbuckconverter')
```



The model is configured with a reference voltage that switches from 15 to 25 Volts at 0.004 seconds and a load current that is active from 0.0025 to 0.005 seconds. The controller is initialized with default gains and results in overshoot and slow settling time.

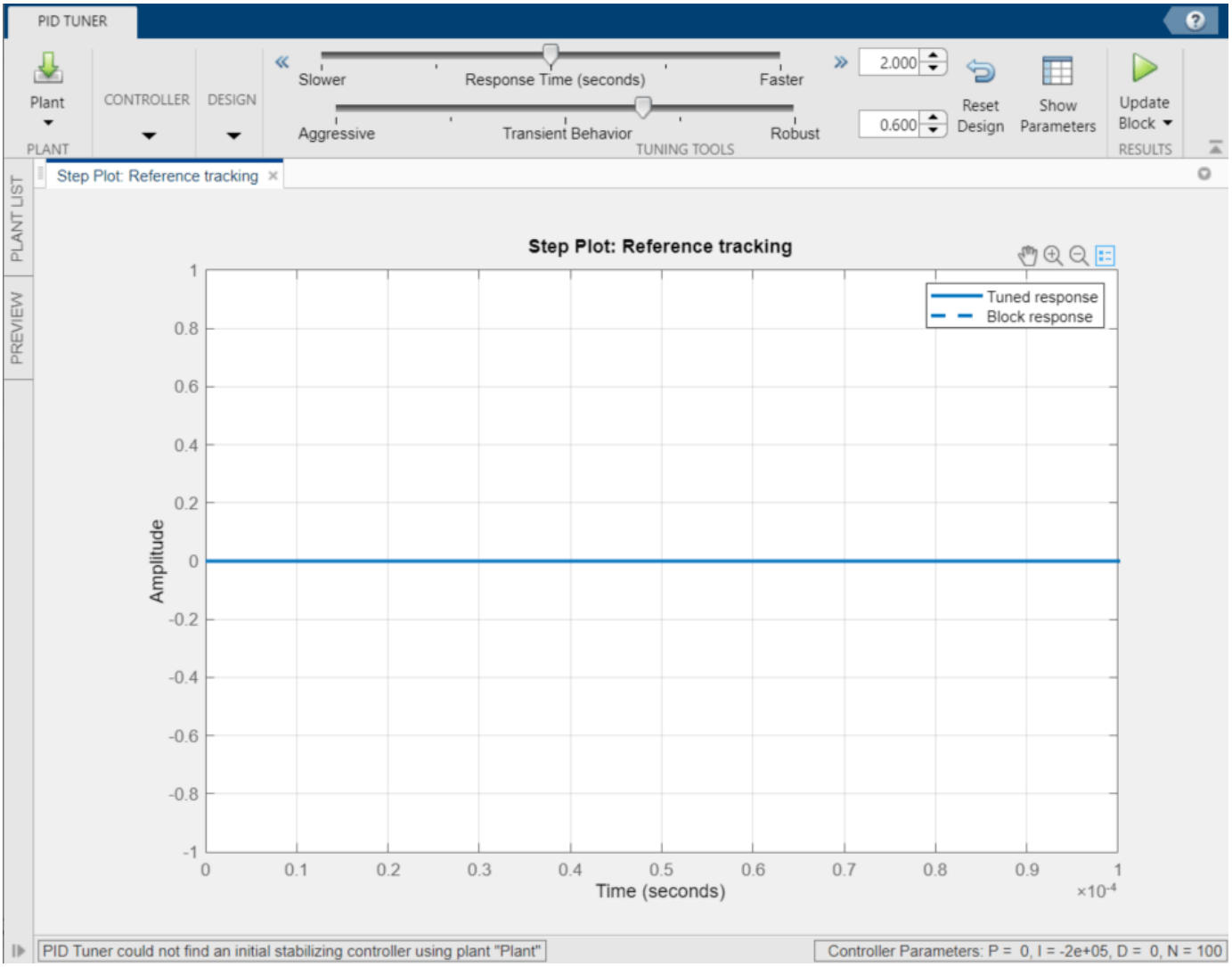
```
open_system('scdbuckconverter/Scope 1')
open_system('scdbuckconverter/Scope 2')
```





Simulate Model to Generate I/O Data

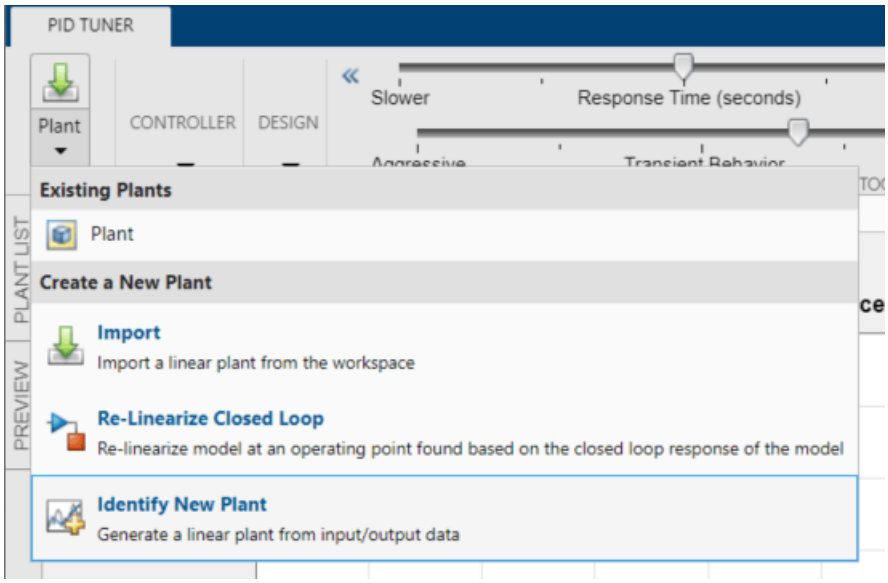
To open the PID Tuner, in the **Feedback controller** subsystem, open the **PID Controller** block dialog, and click **Tune**. **PID Tuner** indicates that the model cannot be linearized and returned a zero system.



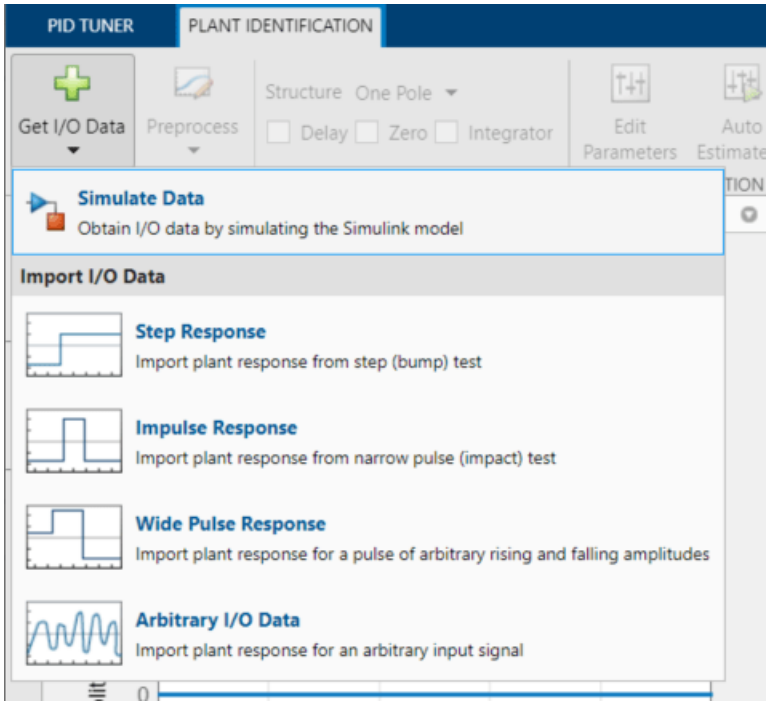
PID Tuner provides several alternatives when linearization fails. In the **Plant** drop-down list, you can select one of the following methods:

- **Import** - Import a linear model from the MATLAB workspace.
- **Re-linearize Closed Loop** - Linearize the model at different simulation snapshot times.
- **Identify New Plant** - Identify a plant model using measured data.

For this example, click **Identify New Plant** to open the Plant Identification tool. For plant identification, you must specify a finite value for the Simulink model stop time.



To open a tool that simulates the model to collect data for plant identification, on the **Plant Identification** tab, click **Get I/O Data > Simulate Data**.



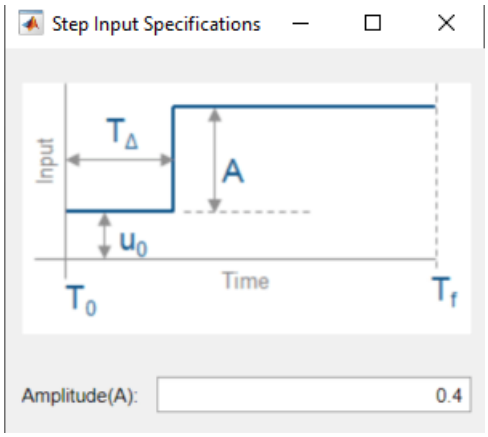
On the **Simulate I/O Data** tab, you simulate the plant seen by the controller. The software temporarily:

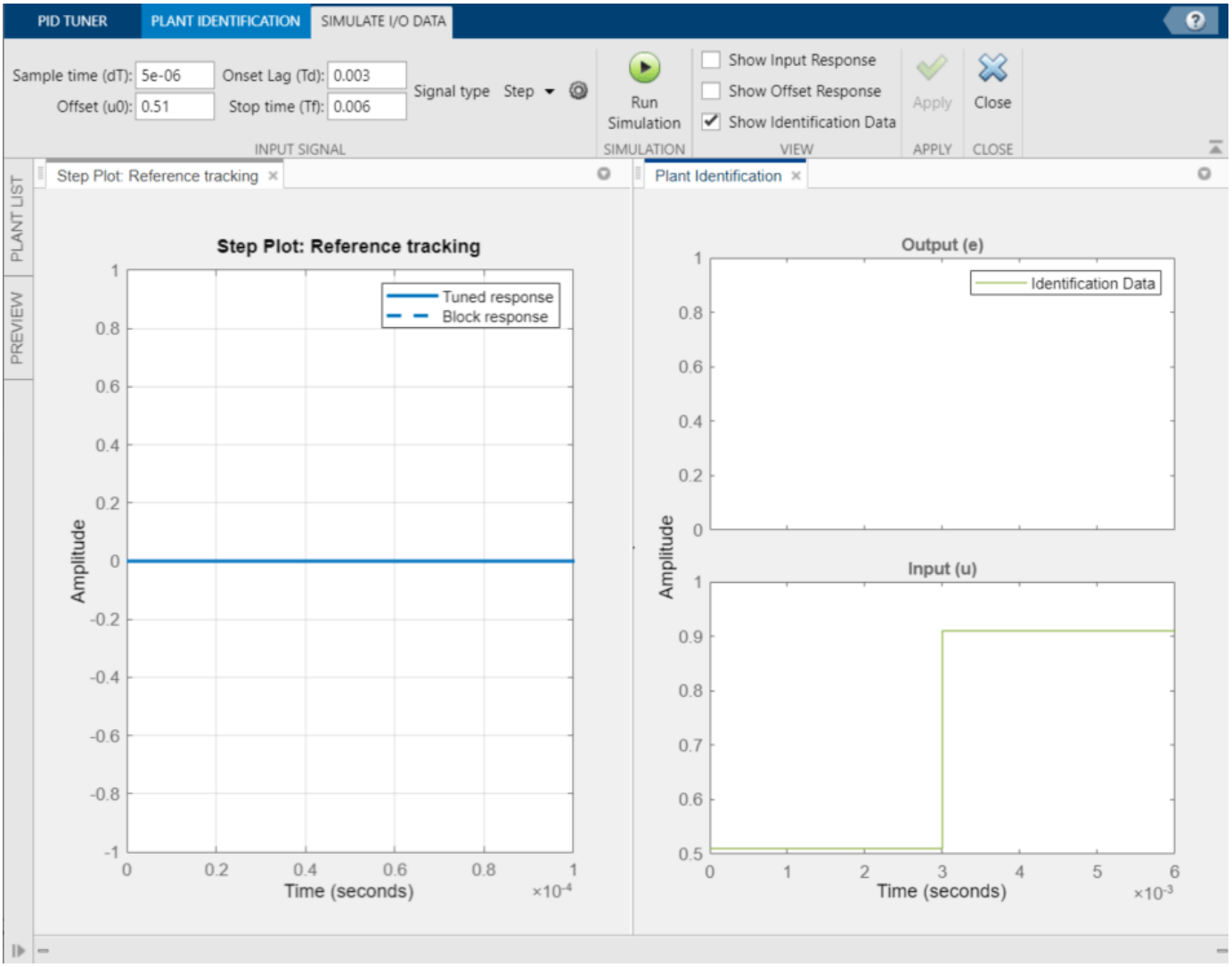
- Removes the PID Controller block from the model.
- Injects a signal where the output of the PID block used to be.
- Measures the resulting signal where the input to the PID block used to be.

This data describes the response of the plant seen by the controller. The **PID Tuner** uses this response data to estimate a linear plant model.

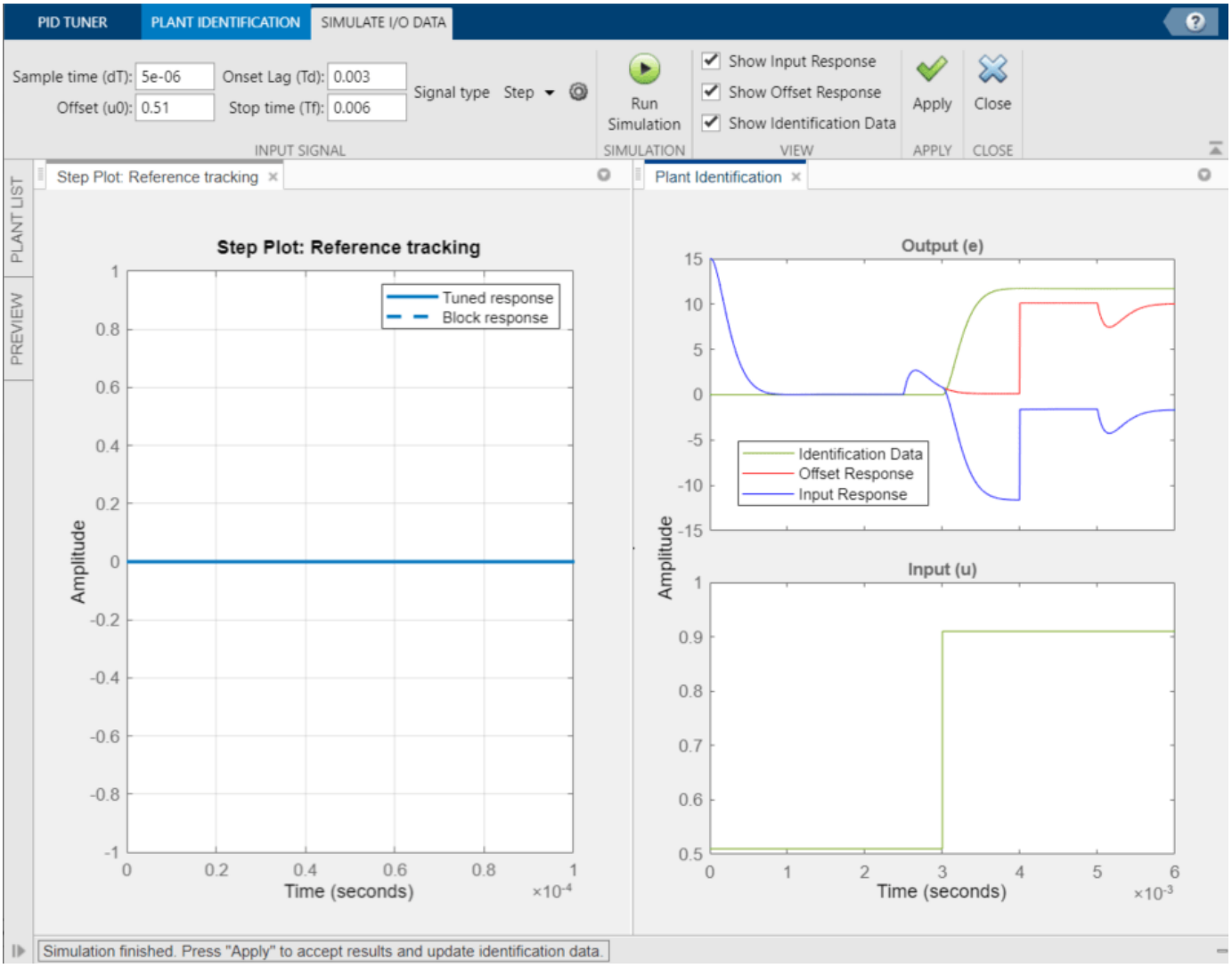
Configure the input signal as a step input with the following properties:

- **Sample Time (ΔT)** = 5e-6 - Controller sample rate.
- **Offset (u_0)** = 0.51 - Output offset value that puts the converter in a state where the output voltage is near 15V and gives the operating point around which to tune the controller.
- **Onset Time (T_Δ)** = 0.003 - Delay to allow sufficient time for the converter to reach the 15V steady state before applying the step change.
- **Step Amplitude (A)** = 0.4 - Step size of the controller output (plant input) to apply to the model. This value is added to the offset value u_0 so that the actual plant input steps from 0.51 to 0.91. The controller output (plant input) is limited to the range [0.01 0.95].





Select **Show Input Response**, **Show Offset Response**, and **Show Identification Data**. Then, click the **Run Simulation**. The **Plant Identification** plot is updated.



The red curve is the offset response. The offset response is the plant response to a constant input of u_0. The response shows that the model has some transients with a constant input, in particular:

- The [0 0.001] second range where the converter reaches the 15V steady state. Recall that this signal is the control error signal and hence drops to zero as steady state is reached.
- The [0.0025 0.004] second range where the converter reacts to the current load being applied while the reference voltage is maintained at 15V.
- The 0.004 second point where the reference voltage signal is changed from 15V to 25V resulting in a larger control error signal.
- The [0.005 0.006] second range where the converter reacts to the current load being removed.

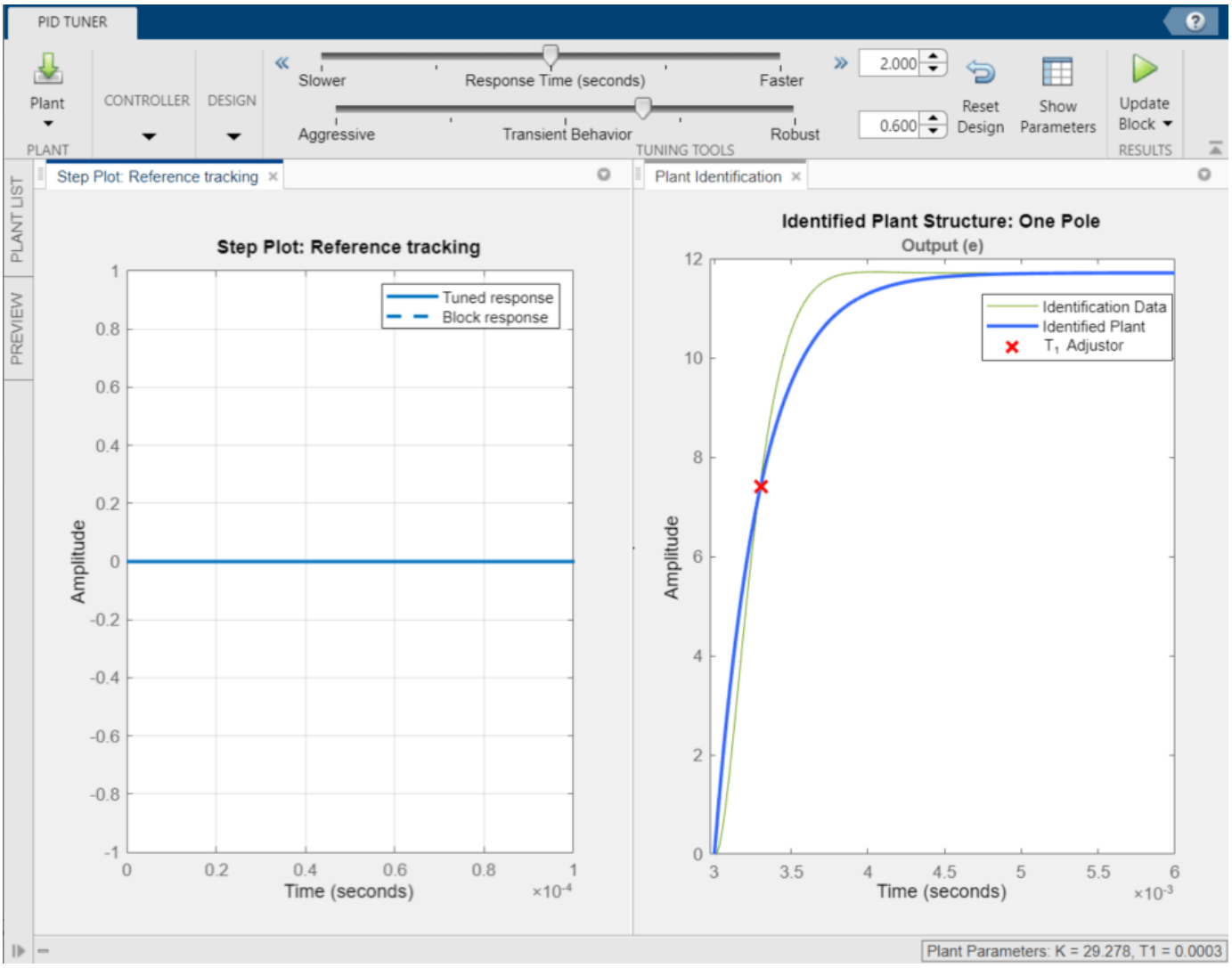
The blue curve shows the complete plant response that contains the contributions from the initial transients (significant for times < 0.001 seconds), the response to the cyclic current load (time durations 0.0025 to 0.005 seconds), reference voltage change (at 0.004 seconds), and response to the step test signal (applied at time 0.003 seconds). In contrast, the red curve is the response to only the initial transients, reference voltage step, and cyclic current load.

The green curve is the data that will be used for plant identification. This curve is the change in response due to the step test signal, which is the difference between the blue (input response) and red (offset response) curves taking into account the negative feedback sign.

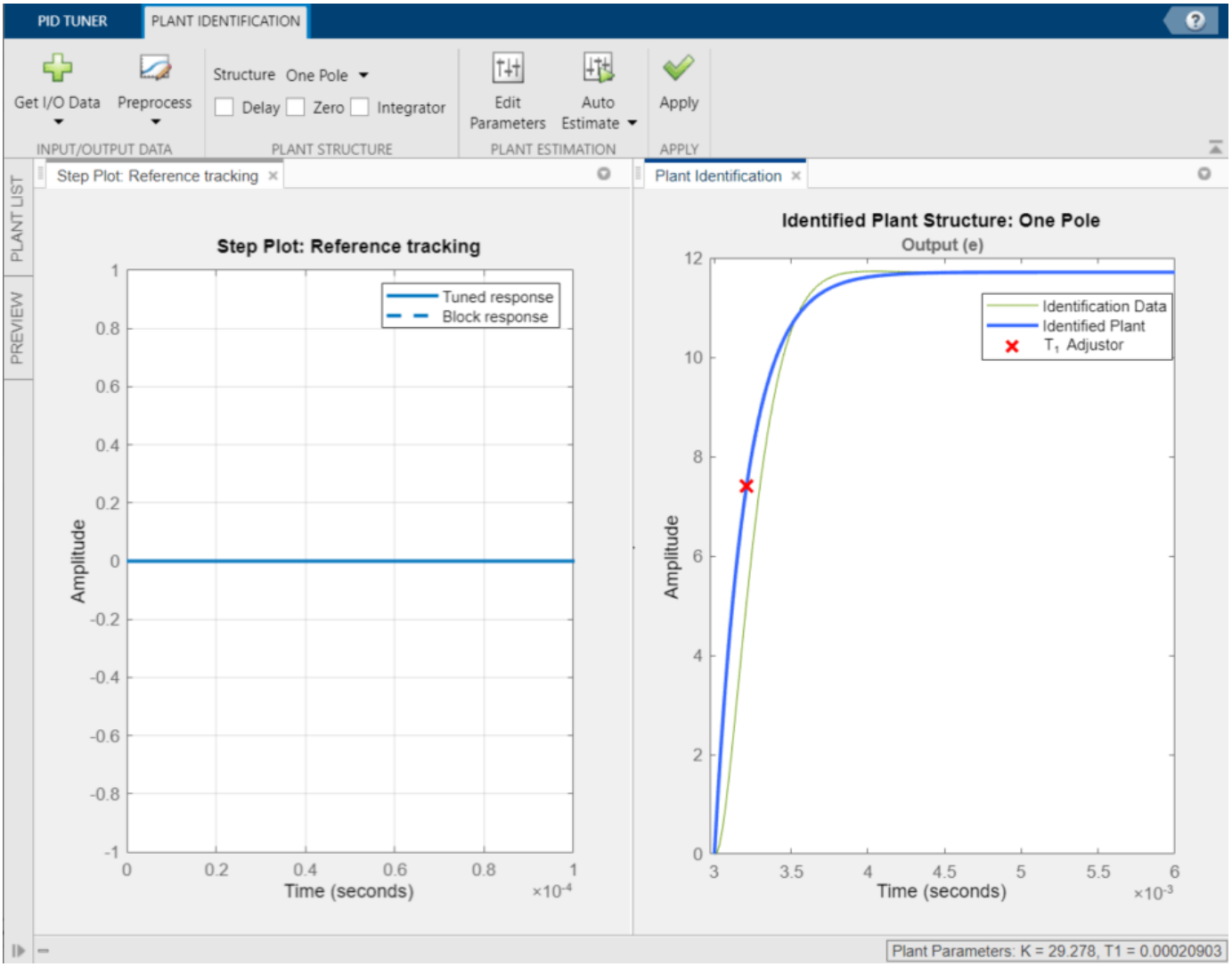
To use the measured data to identify a plant model, click **Apply**. Then, to return to plant identification, click **Close**.

Plant Identification

PID Tuner identifies a plant model using the data generated by simulating the model. You tune the identified plant parameters so that the identified plant response, when provided the measured input, matches the measured output.

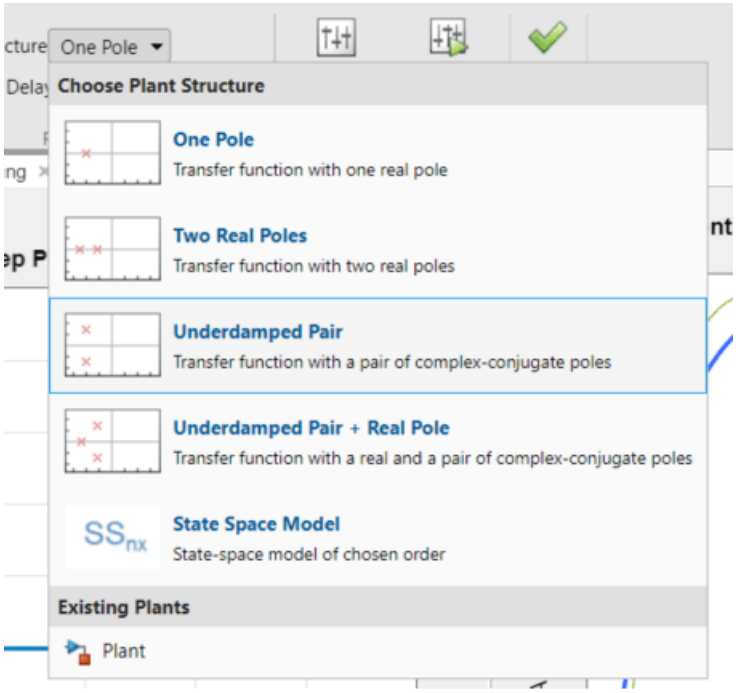


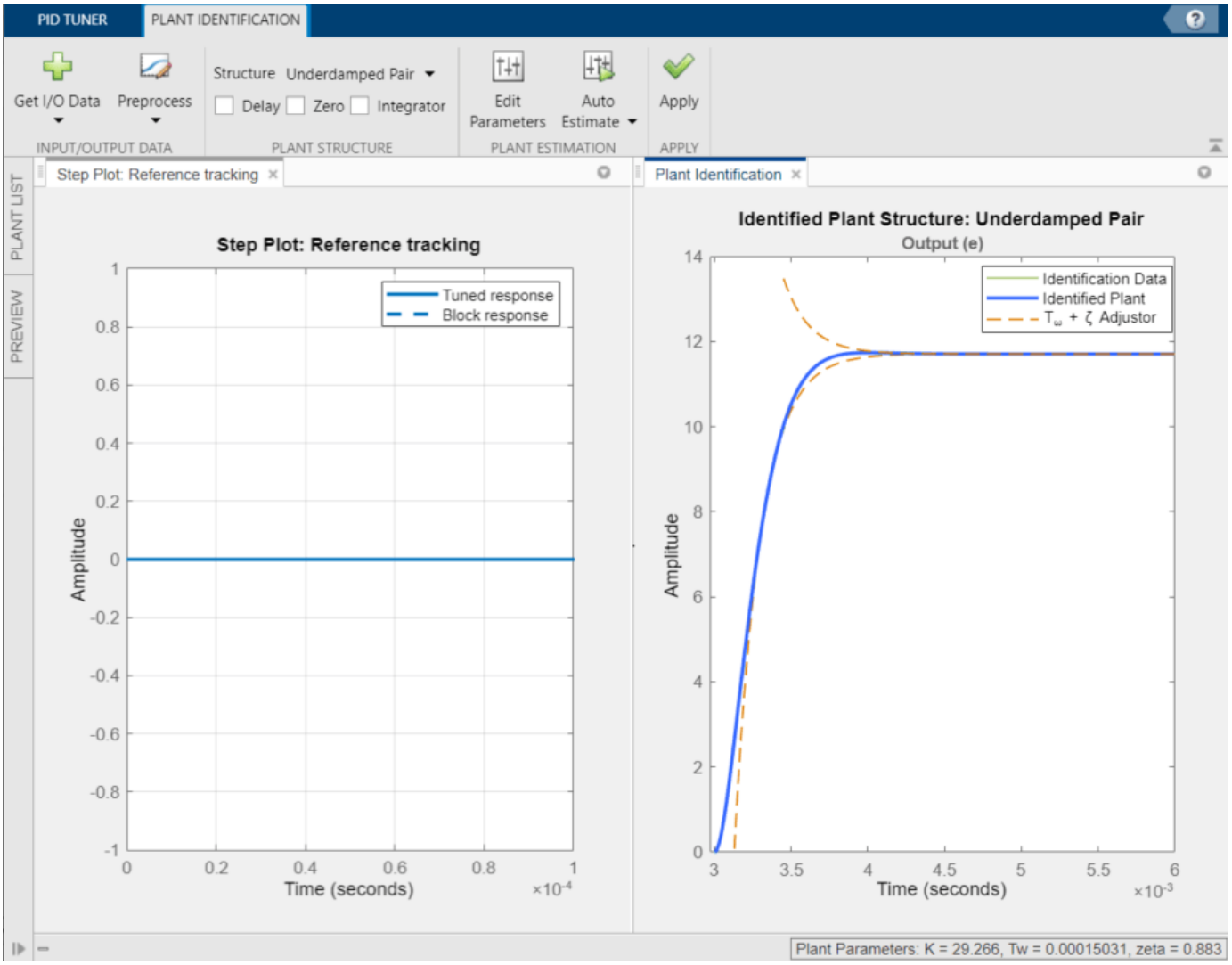
You can manually adjust the estimated model. Click and drag the plant curve and pole location (X) to adjust the identified plant response so that it matches the identification data as closely as possible.



To tune the identified plant using automated identification, click **Auto Estimate**. The automated tuning response is not much better than the interactive tuning. The identified plant and identification data do not match well. Change the plant structure to get a better match.

- In the Structure drop-down list, select **Underdamped pair**.
- Click and drag the 2nd order envelope to match the identified data as closely as possible (almost critically damped).
- Click **Auto Estimate** to fine tune the plant model.

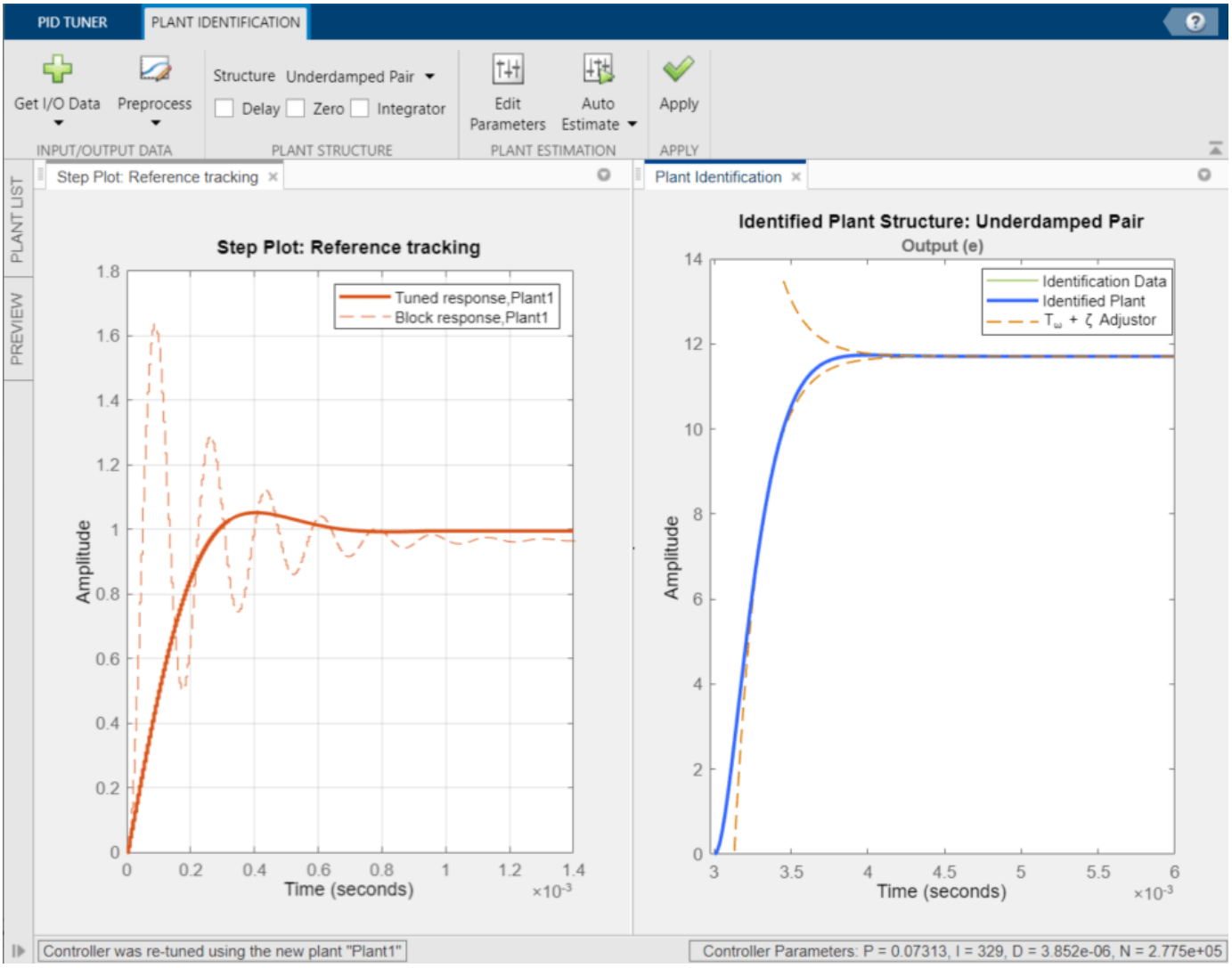

One Pole
Choose Plant Structure
One Pole
Transfer function with one real pole
Two Real Poles
Transfer function with two real poles
Underdamped Pair
Transfer function with a pair of complex-conjugate poles
Underdamped Pair + Real Pole
Transfer function with a real and a pair of complex-conjugate poles
SSnx
State Space Model
State-space model of chosen order
Existing Plants
Plant



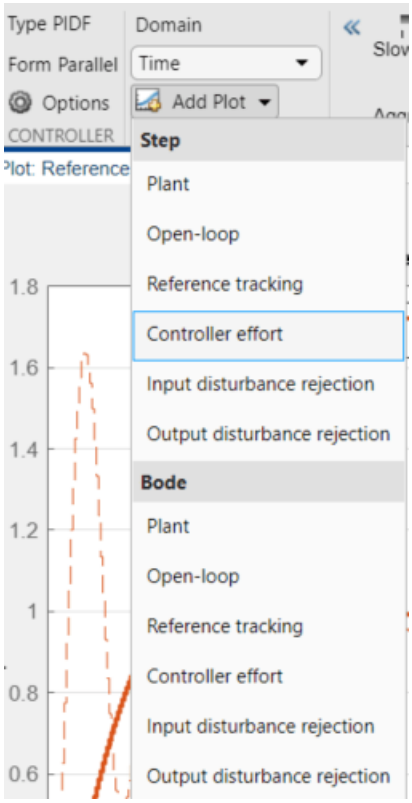
To designate the identified model as the current plant for controller tuning, Click **Apply**. **PID Tuner** then automatically tunes a controller for the identified plant and updates the **Reference Tracking** step plot.

Controller Tuning

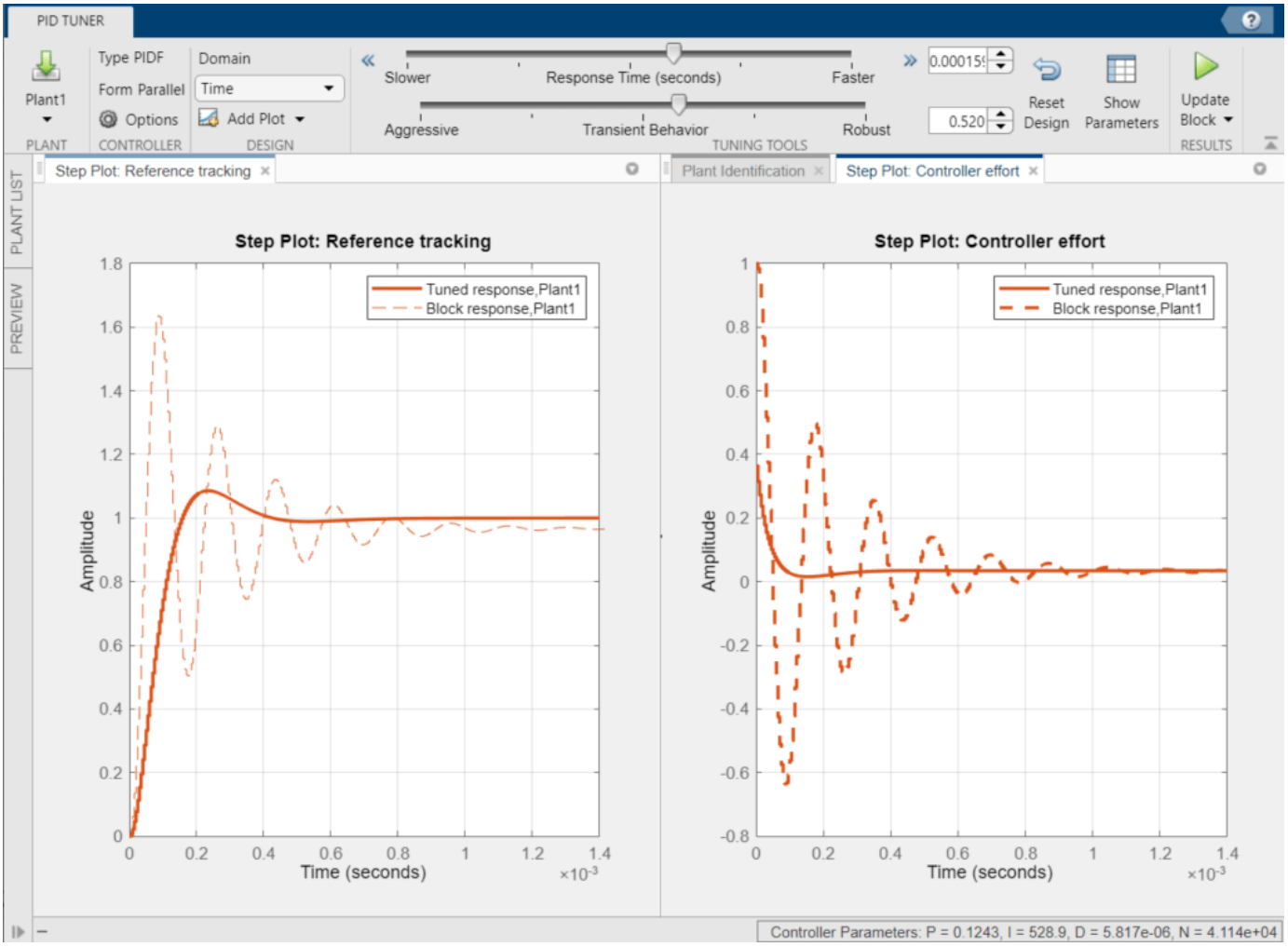
The PID Tuner automatically tunes a PID controller for the identified plant. The tuned controller response has about 5% overshoot and a settling time of around 0.0006 seconds. Click the **Reference Tracking** step plot to make it the current figure.



The controller output is the duty cycle for the PWM system and must be limited to [0.01 0.95]. To confirm that the controller output satisfies these bounds, create a controller effort plot. On the **PID Tuner** tab, in the **Add Plot** drop-down list, under **Step**, click **Controller effort**. Move the newly created **Controller effort** plot to the second plot group.

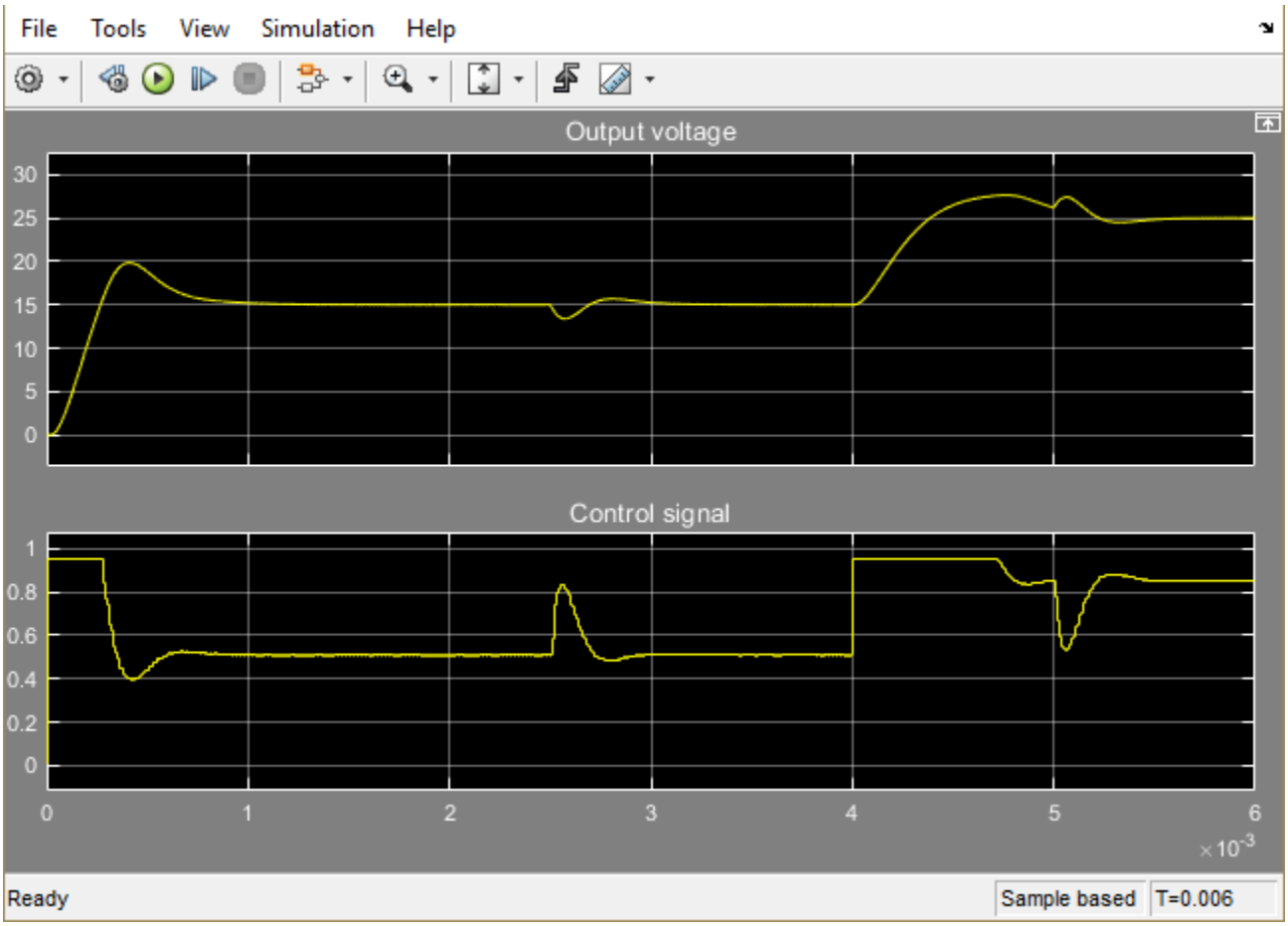


In the **Controller effort** plot, the tuned response (solid line) shows a large control effort required at the start of the simulation. To achieve a settling time of about 0.0004 seconds and overshoot of 9%, adjust the **Response Time** and **Transient Behavior** sliders. These adjustments reduce the maximum control effort to the acceptable range.



To update the Simulink block with the tuned controller values, click **Update Block**.

To confirm the PID controller performance, simulate the Simulink model.




```
bdclose('scdbuckconverter')
```

See Also

PID Tuner

More About

- “System Identification for PID Control” on page 7-61
- “Input/Output Data for Identification” on page 7-67
- “Interactively Estimate Plant from Measured or Simulated Response Data” on page 7-55
- “Choosing Identified Plant Structure” on page 7-68

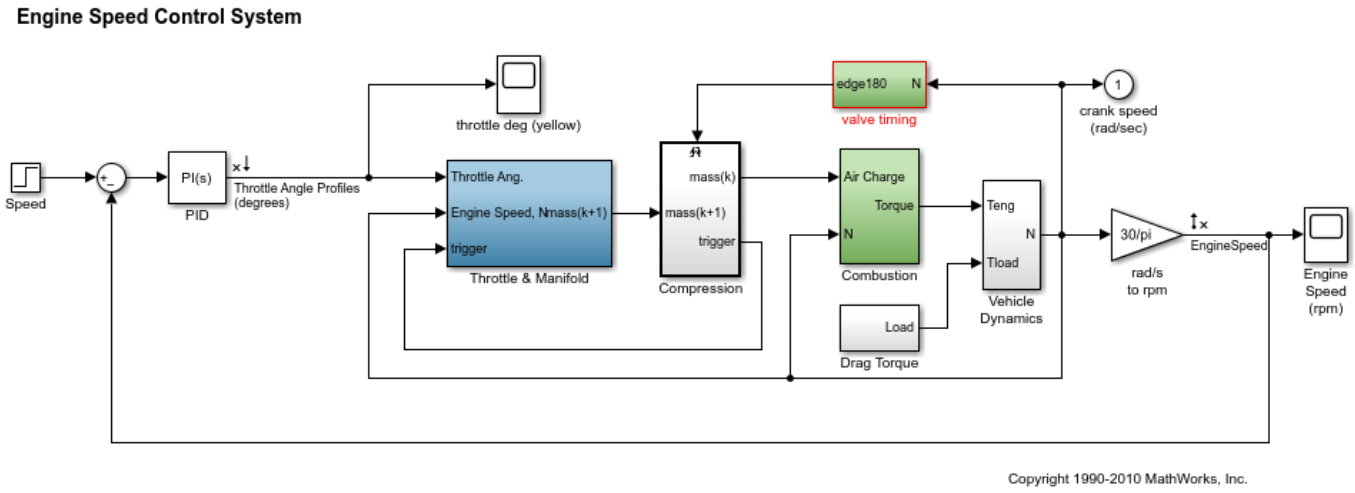
Design PID Controller Using Estimated Frequency Response

This example shows how to design a PI controller using a frequency response estimated from a Simulink model. 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).

Open the Model

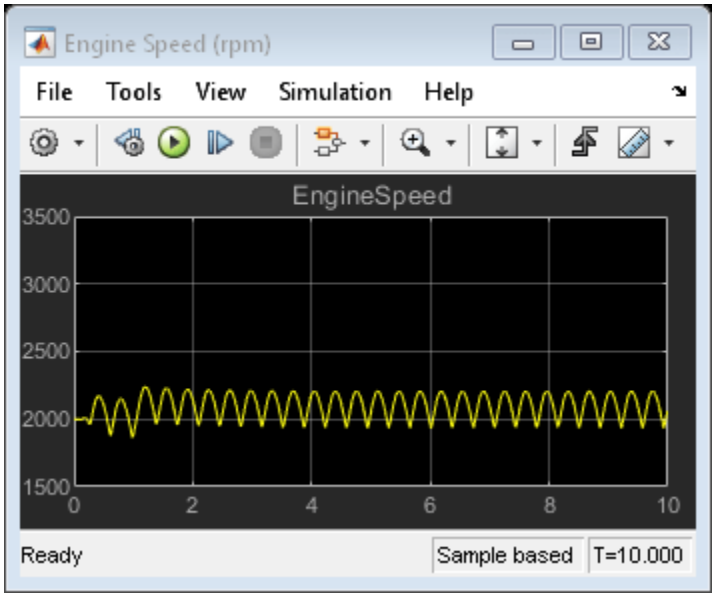
Open the engine control model and take a few moments to explore it.

```
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)
```



Close the scope.

```
close_system([mdl '/Engine Speed (rpm)'])
```

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.

To verify the zero linearization, first obtain the linearization input and output points from the model.

```
io = getlinio(mdl);
```

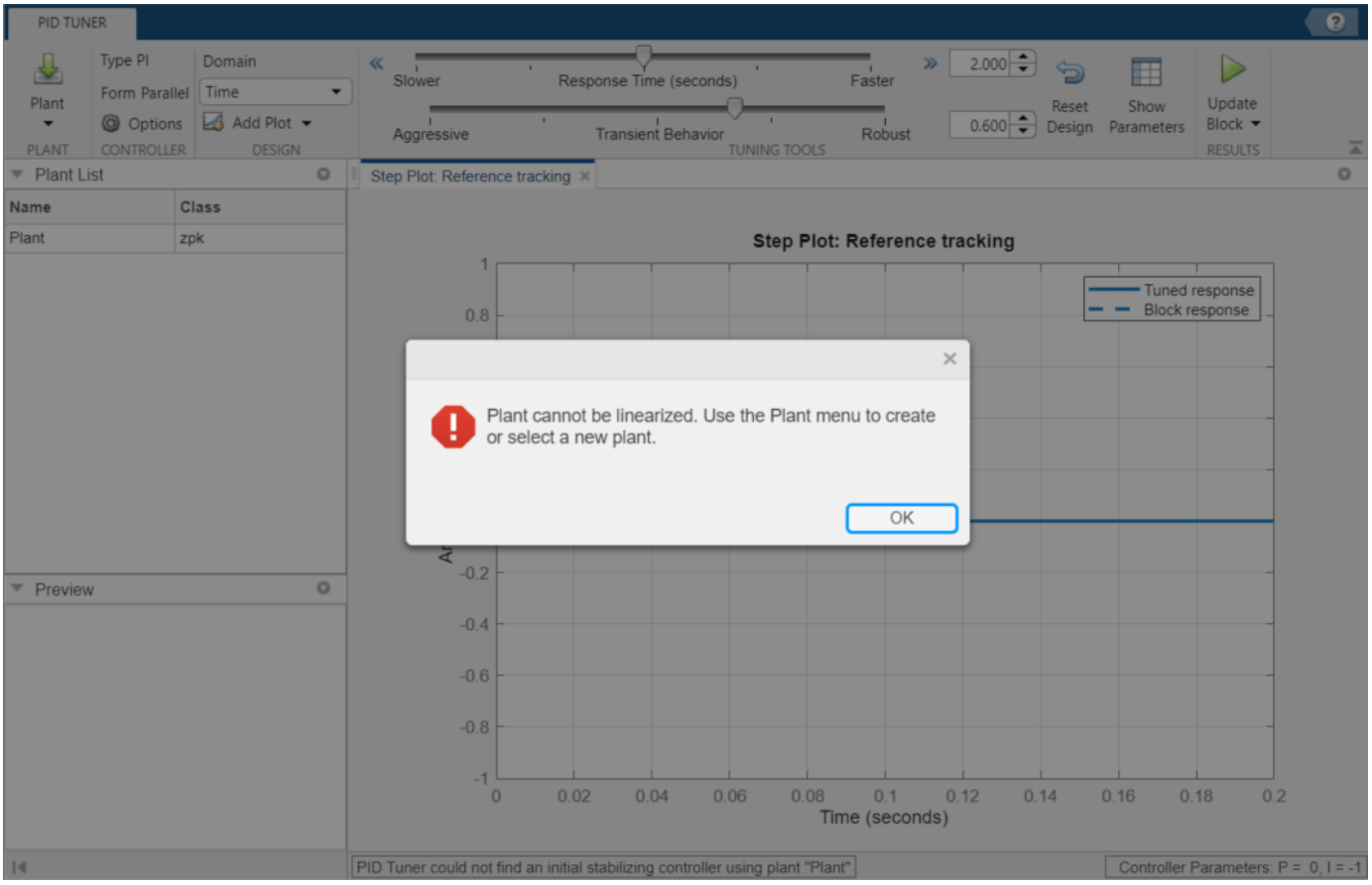
Then, linearize the plant at its initial operating point.

```
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 event-based subsystems. Since **PID Tuner** uses the same approach to obtain a linear plant model, **PID Tuner** also obtains a plant model with zero gain and rejects it during the launching process.

To launch the **PID Tuner**, open the PID block dialog, and click **Tune**. An information dialog opens and indicates that the plant model linearized at the initial operating point has zero gain and cannot be used to design a PID controller.



An 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 the MATLAB workspace, and import it back to **PID Tuner** to continue PID design.

Obtain Estimated Frequency Response Data Using Sinestream Signals

The sinestream input signal is the most reliable input signal for estimating an accurate frequency response of a Simulink model using the `frestimate` function. For more information on how to use `frestimate`, see “Frequency Response Estimation Using Simulation-Based Techniques” on page 5-74.

In this example, create a sine stream that sweeps frequency from 0.1 to 10 rad/sec with an amplitude of 1e-3. You can inspect the estimation results using the bode plot.

Construct the sinestream signal.

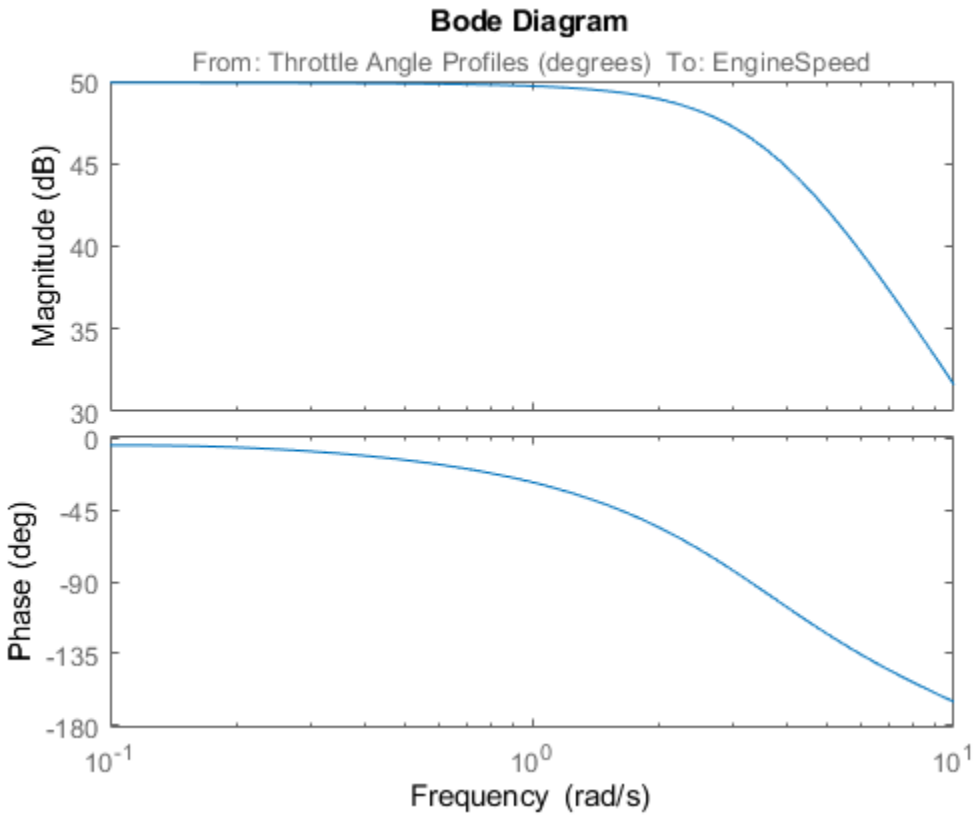
```
in = frest.Sinestream('Frequency',logspace(-1,1,50),'Amplitude',1e-3);
```

Estimate the frequency response. This process can take a few minutes.

```
sys = frestimate(mdl,io,in);
```

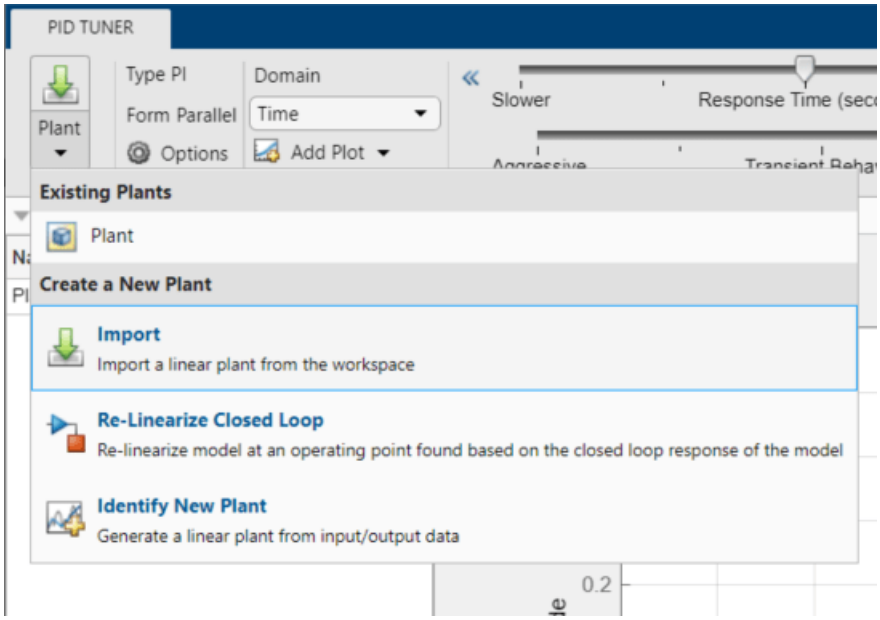
Display the estimated frequency response.

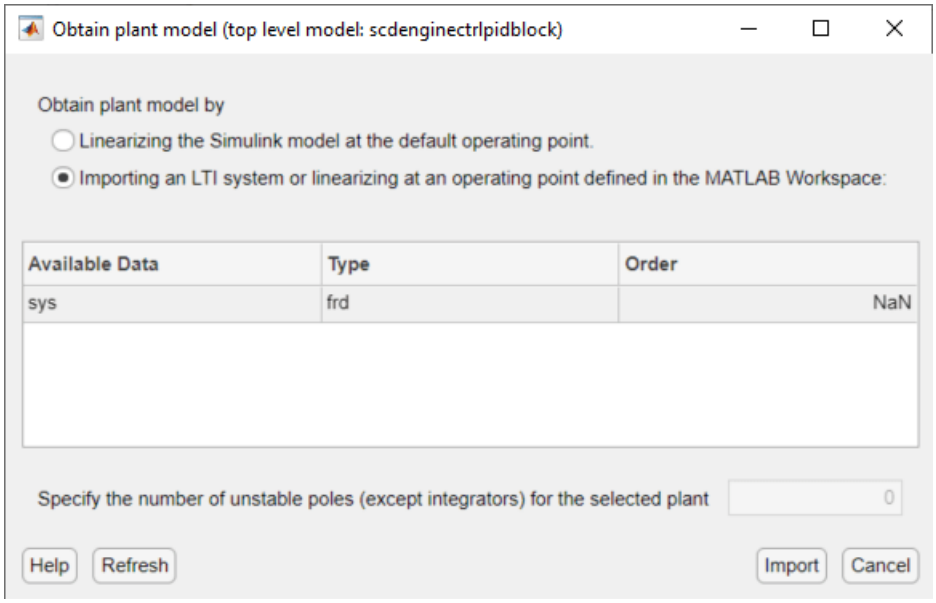
```
bode(sys)
```



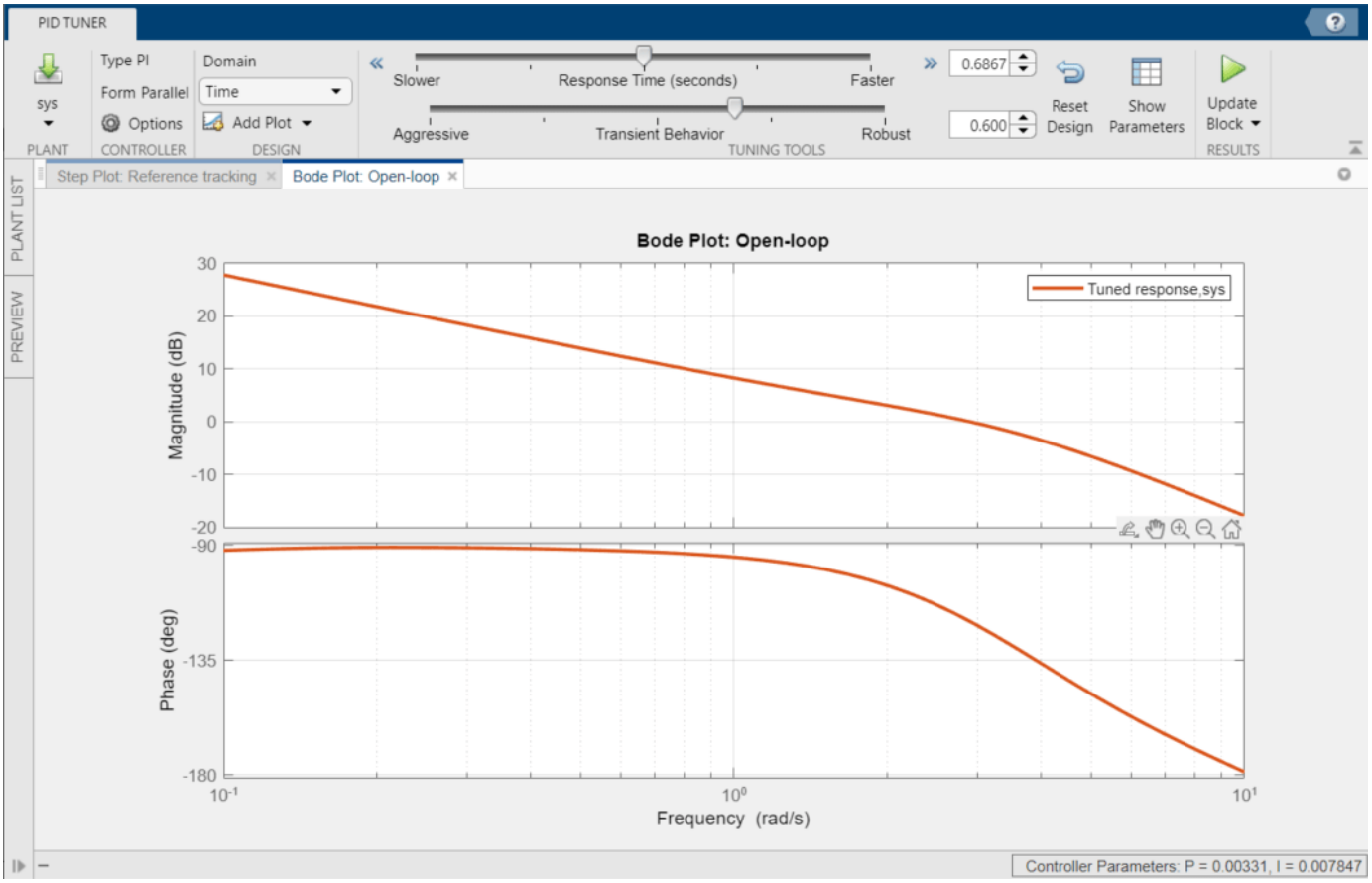
Design PI Controller

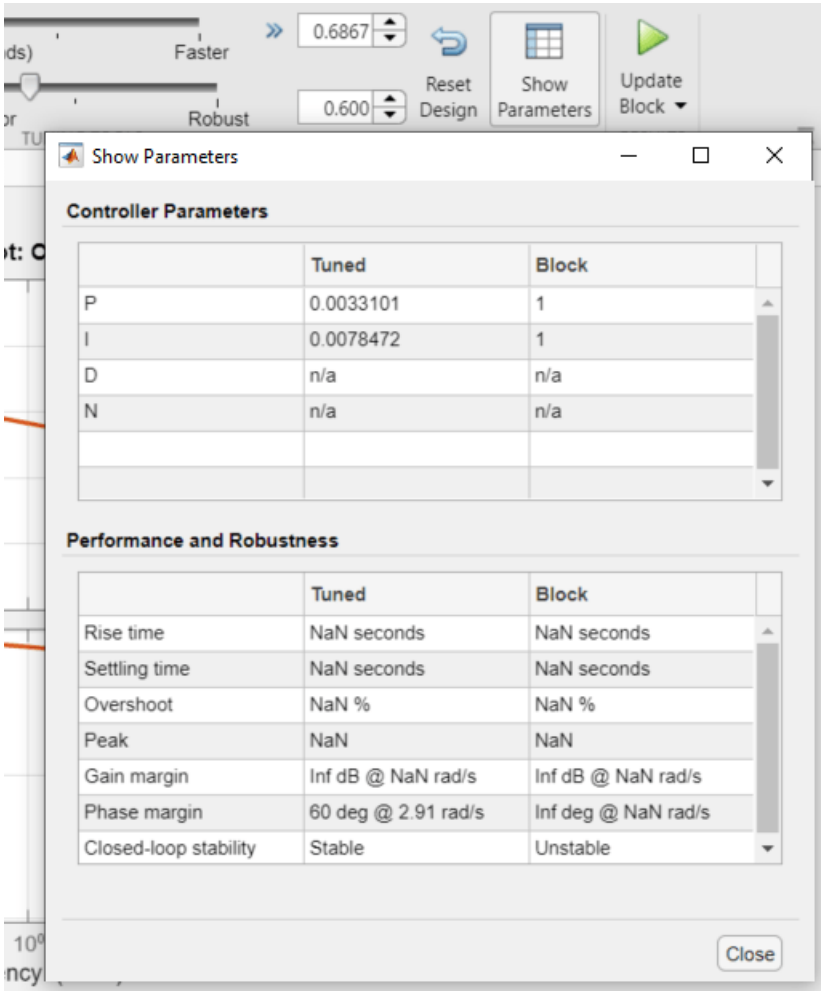
`sys` is an `frd` system that represents the plant frequency response at the initial operating point. To use it in **PID Tuner**, we need to import it after **PID Tuner** is launched. Click **Plant**, and select **Import**. The sampling rate of the imported `frd` plant must match the sampling rate of the PID Controller block.





Click **Importing an LTI system**, and in the list, select **sys**. Then, click "OK" to import the `frd` system into **PID Tuner**. The automated design returns a stabilizing controller. Click **Add Plot**, and select **Open-Loop** Bode plot. The plot shows reasonable gain and phase margin. Click **Show Parameters** to see the gain and phase margin values. Time domain response plots are not available for `frd` plant models.

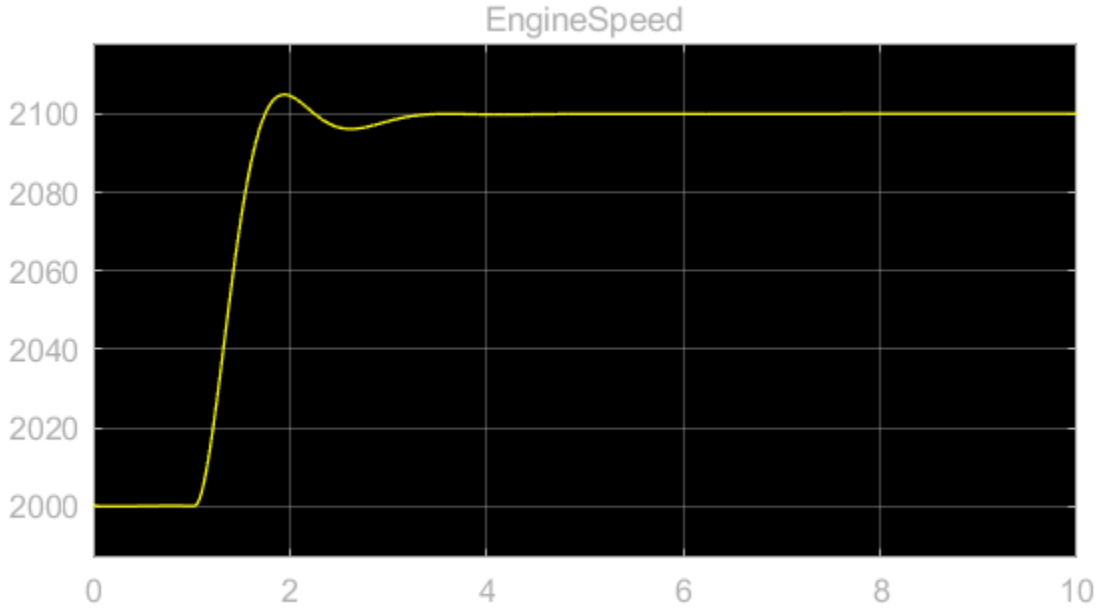

PID TUNER
Type PI
Form Parallel
Options
sys
PLANT
CONTROLLER
Domain
Time
Add Plot
DESIGN
Slower
Response Time (seconds)
Faster
Aggressive
Transient Behavior
Robust
TUNING TOOLS
0.6867
0.600
Reset Design
Show Parameters
Update Block
RESULTS
PLANT LIST
PREVIEW
Step Plot: Reference tracking
Bode Plot: Open-loop
Bode Plot: Open-loop
Tuned response,sys
Magnitude (dB)
30
20
10
0
-10
-20
Phase (deg)
-90
-135
-180
10-1
100
101
Frequency (rad/s)
Controller Parameters: P = 0.00331, I = 0.007847



To update the PID block P and I gains, click **Update Block**.

Simulate 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)
```

See Also

Apps
PID Tuner

Functions
`linearize` | `frestimate`

Related Examples

- “Design PID Controller from Plant Frequency-Response Data” on page 7-36
- “Design PID Controller Using Plant Frequency Response Near Bandwidth” on page 7-43
- “Design Controller for Power Electronics Model Using Frequency Response Data” on page 7-76

Design Family of PID Controllers for Multiple Operating Points

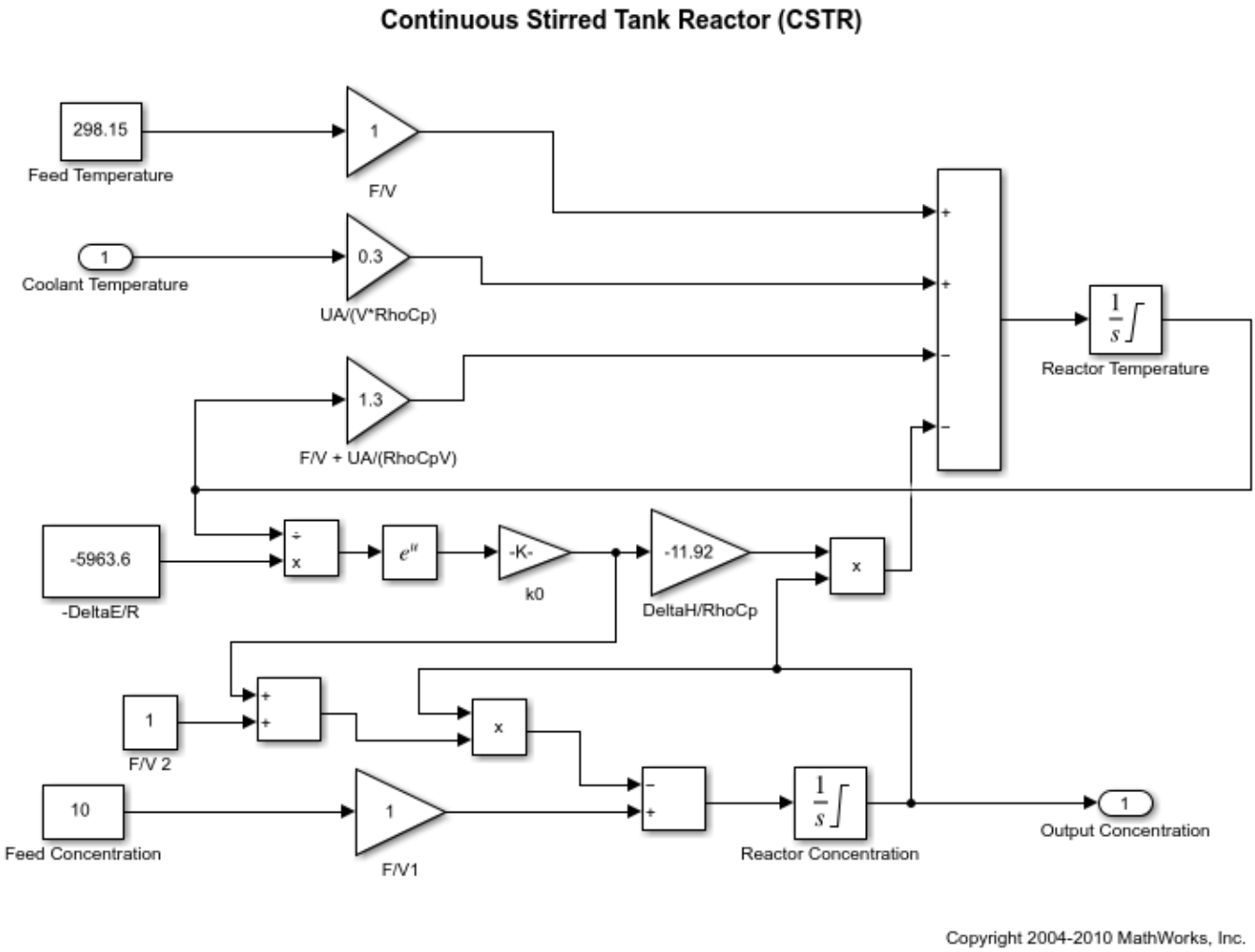
This example shows how to design an array of PID controllers for a nonlinear plant in Simulink® that operates over a wide range of operating points.

Open 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 for which the PID controller is designed. However, since the plant is a strongly nonlinear system, control performance degrades if the operating point changes significantly. The closed-loop system can even become unstable.

Open the CSTR plant model.

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



For more information on this system, see [1].

Introduction to Gain Scheduling

A common approach to solve the nonlinear control problem is to use 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 controllers, 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 information on gain scheduling, see [2].

This example focuses on designing a family of PID controllers for the nonlinear CSTR plant.

Obtain 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 a low conversion rate (`C = 9`) and a high conversion rate (`C = 2`). In this example, divide the operating range into eight regions represented by `C = 2` through `9`.

Specify the operating regions.

```
C = [2 3 4 5 6 7 8 9];
```

Create an array of default operating point specifications.

```
op = operspec(mdl,numel(C));
```

Initialize the operating point specifications by specifying that the output concentration is a known value and specifying the output concentration value.

```
for ct = 1:numel(C)
    op(ct).Outputs.Known = true;
    op(ct).Outputs.y = C(ct);
end
```

Compute the equilibrium operating points corresponding to the values of `C`.

```
opoint = findop(mdl,op,findopOptions('DisplayReport','off'));
```

Linearize the plant at these operating points.

```
Plants = linearize(mdl,opoint);
```

Since the CSTR plant is nonlinear, the linear models display different characteristics. For example, plant models with high and low conversion rates are stable, while the others are not.

```
isstable(Plants,'elem')'
```

```
ans =
```

```
  1x8 logical array

   1   1   0   0   0   0   1   1
```

Design PID Controllers for the Plant Models

To design multiple PID controllers in batch, use the `pidtune` function. The following commands 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.

```
Controllers = pidtune(Plants,'pidf',1);
```

Display the controller for `C = 4`.

```
Controllers(:,:,4)
```

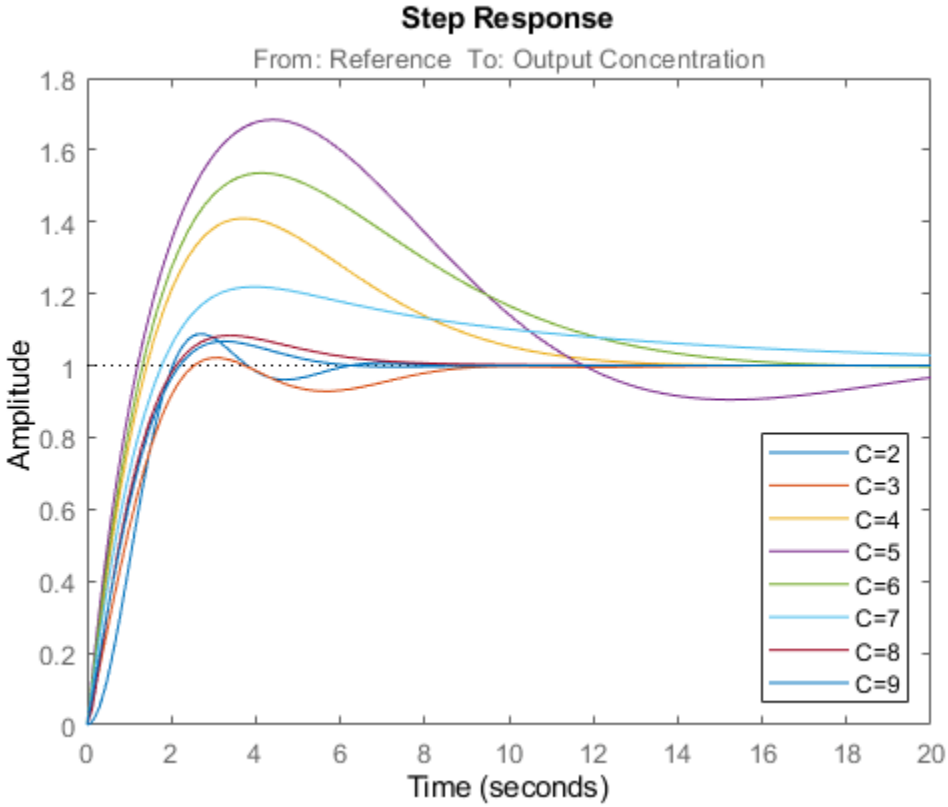
```
ans =

             1            s
  Kp + Ki * --- + Kd * --------
             s          Tf*s+1

  with Kp = -12.4, Ki = -1.74, Kd = -16, Tf = 0.00875

Continuous-time PIDF controller in parallel form.
```

To analyze the closed-loop responses for step setpoint tracking, first construct the closed-loop systems.

```
clsys = feedback(Plants*Controllers,1);
```

Plot the closed-loop responses.

```
figure
hold on
for ct = 1:length(C)
    % Select a system from the LTI array
    sys = clsys(:,:,ct);
    sys.Name = ['C=',num2str(C(ct))];
    sys.InputName = 'Reference';
    % Plot step response
    stepplot(sys,20);
end
legend('show','location','southeast')
```



All the closed loops are stable, but the overshoots of the loops with unstable plants (`C = 4`, through 7) are too large. To improve the results for the unstable plant models, increase the target open-loop bandwidth to `10` rad/sec.

```
Controllers = pidtune(Plants,'pidf',10);
```

Display the controller for `C = 4`.

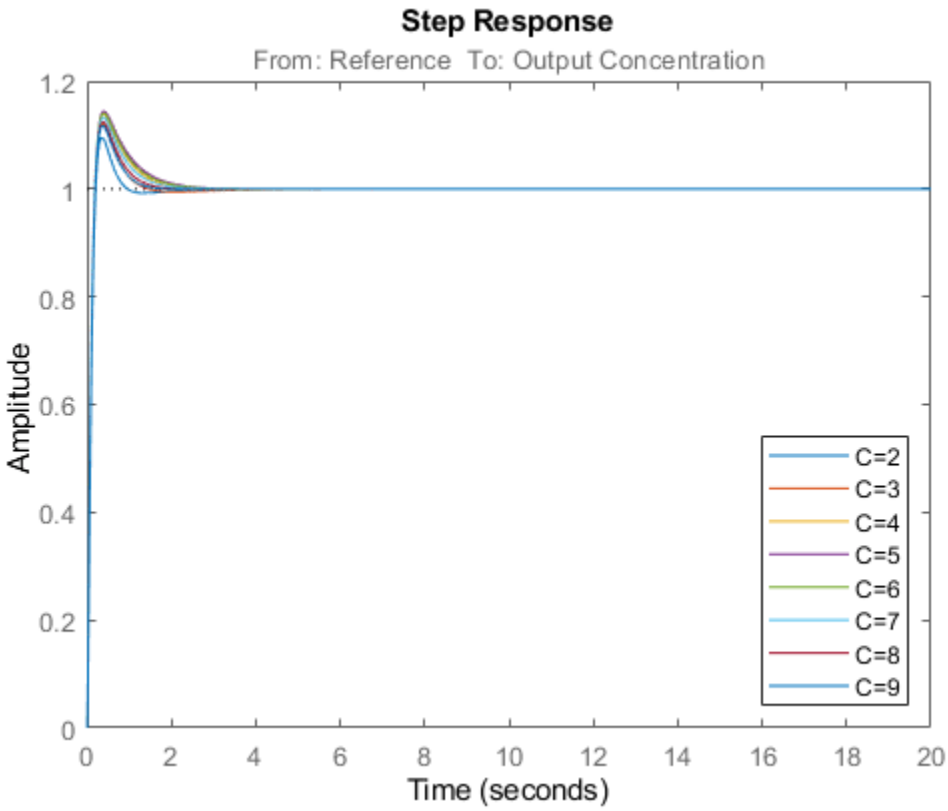
```
Controllers(:,:,4)
```

```
ans =

             1             s
  Kp + Ki * --- + Kd * --------
             s          Tf*s+1

  with Kp = -283, Ki = -151, Kd = -128, Tf = 0.0183

Continuous-time PIDF controller in parallel form.
```

Construct the closed-loop systems, and plot the closed-loop step responses for the new controllers.

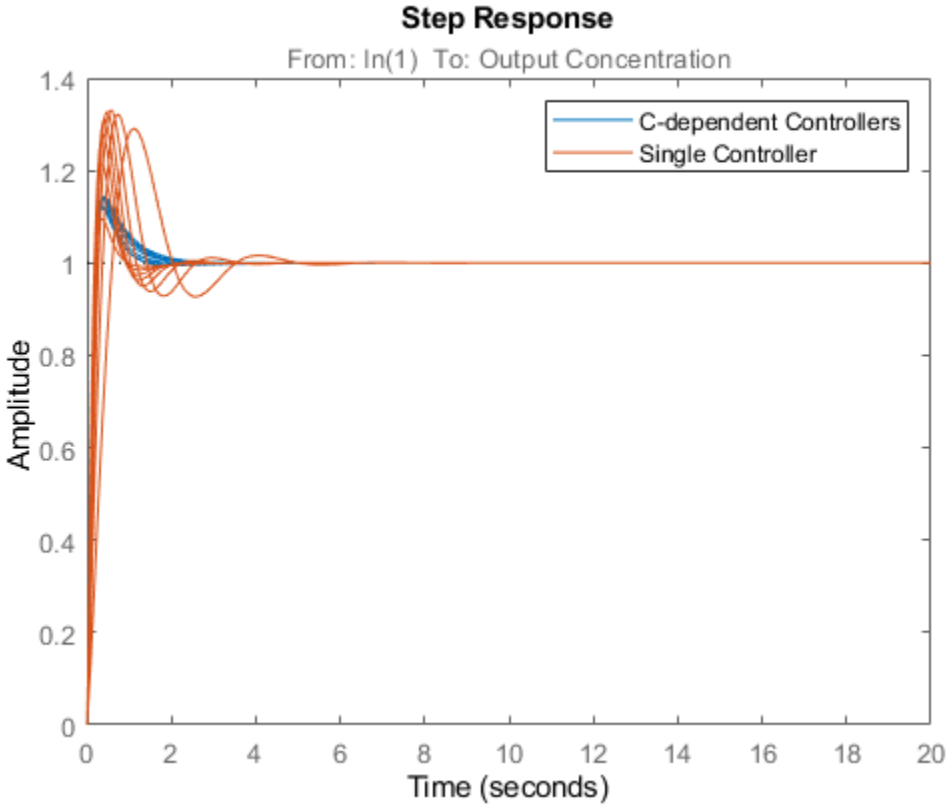
```
clsys = feedback(Plants*Controllers,1);
figure
hold on
for ct = 1:length(C)
```

```
    % Select a system from the LTI array.
    sys = clsys(:,:,ct);
    set(sys,'Name',['C=',num2str(C(ct))],'InputName','Reference');
    % Plot the step response.
    stepplot(sys,20)
end
legend('show','location','southeast')
```



All the closed-loop responses are now satisfactory. For comparison, examine the response when you use the same controller at all operating points. Create another set of closed-loop systems, where each one uses the C = 2 controller, and plot their responses.

```
clsys_flat = feedback(Plants*Controllers(:,:,1),1);

figure
stepplot(clsys,clsys_flat,20)
legend('C-dependent Controllers','Single Controller')
```



The array of PID controllers designed separately for each concentration gives considerably better performance than a single controller.

However, the closed-loop responses shown above are computed based on linear approximations of the full nonlinear system. To validate the design, implement the scheduling mechanism in your model using the PID Controller block as shown in “Implement Gain-Scheduled PID Controllers” on page 7-139.

Close the model.

```
bdclose(mdl)
```

References

[1] Seborg, Dale E., Thomas F. Edgar, and Duncan A. Mellichamp. *Process Dynamics and Control*. 2nd ed., John Wiley & Sons, Inc, 2004, pp. 34-36.

[2] Rugh, Wilson J., and Jeff S. Shamma. 'Research on Gain Scheduling'. *Automatica* 36, no. 10 (October 2000): 1401-1425.

See Also

`operspec` | `findop` | `pidtune`

More About

- “Implement Gain-Scheduled PID Controllers” on page 7-139
- “Design Controller for Power Electronics Model Using Frequency Response Data” on page 7-76

Implement Gain-Scheduled PID Controllers

This example shows how to implement gain-scheduled control in a Simulink model using a family of PID controllers. The PID controllers are tuned for a series of steady-state operating points of the plant, which is highly nonlinear.

This example builds on the work done in "Design Family of PID Controllers for Multiple Operating Points" on page 7-132. In that example, the continuous stirred tank reactor (CSTR) plant model is linearized at steady-state operating points that have output concentrations C = 2, 3, ..., 8, 9. The nonlinearity in the CSTR plant yields different linearized dynamics at different output concentrations. The example uses the `pidtune` command to generate and tune a separate PID controller for each output concentration.

You can expect each controller to perform well in a small operating range around its corresponding output concentration. This example shows how to use the PID Controller block to implement all of these controllers in a gain-scheduled configuration. In such a configuration, the PID gains change as the output concentration changes. This configuration ensures good PID control at any output concentration within the operating range of the control system.

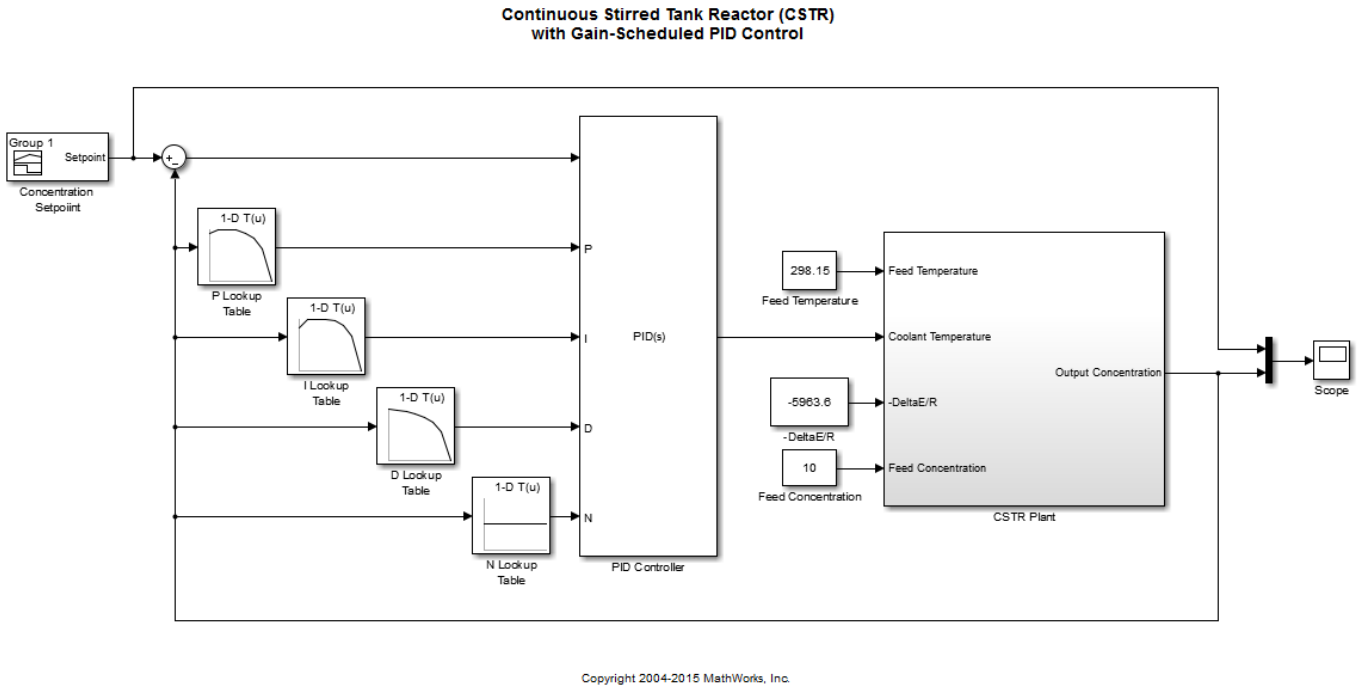
Begin with the controllers generated in "Design Family of PID Controllers for Multiple Operating Points" on page 7-132. If these controllers are not already in the MATLAB workspace, load them from the data file `PIDGainSchedExample.mat`.

```
load PIDGainSchedExample
```

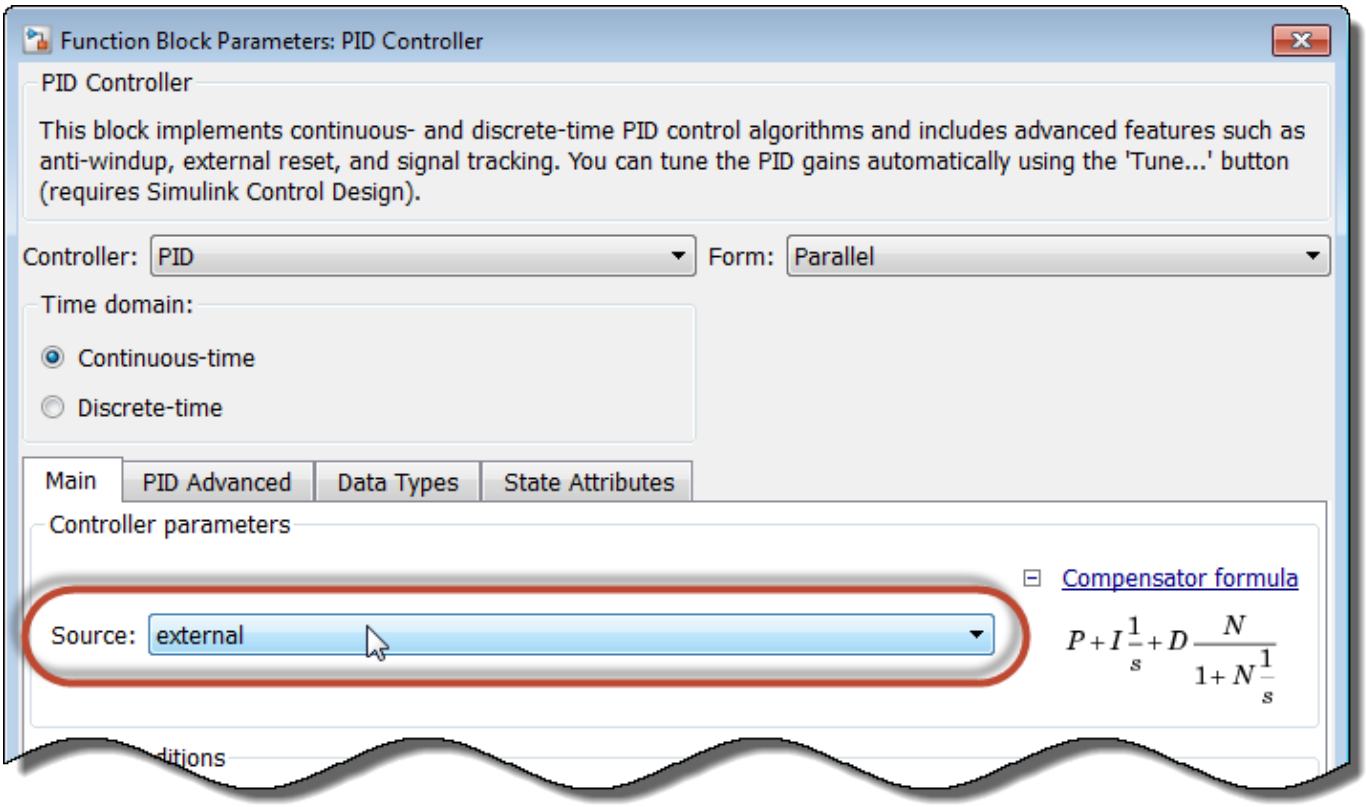
This operation puts two variables in the MATLAB workspace, `Controllers` and `C`. The model array `Controllers` contains eight `pid` models, each tuned for one output concentration in the vector `C`.

To implement these controllers in a gain-scheduled configuration, create lookup tables that associate each output concentration with the corresponding set of PID gains. The Simulink model `PIDGainSchedCSTRExampleModel` contains such lookup tables, configured to provide gain-scheduled control for the CSTR plant. Open this model.

```
open_system('PIDGainSchedCSTRExampleModel')
```

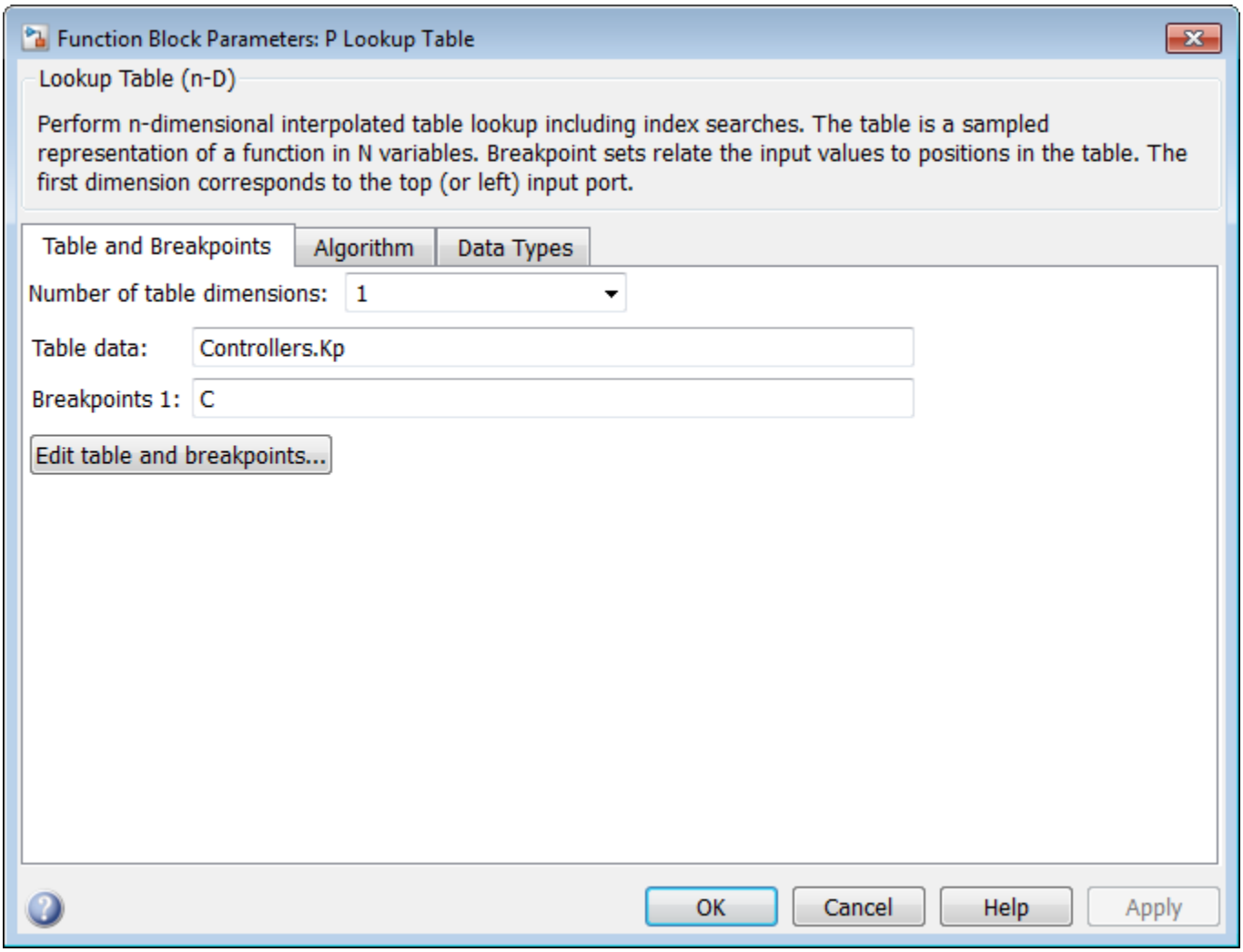


In this model, the PID Controller block is configured to have external input ports for the PID coefficients. Using external inputs allows the coefficients to vary as the output concentration varies. Open the block to examine the configuration.



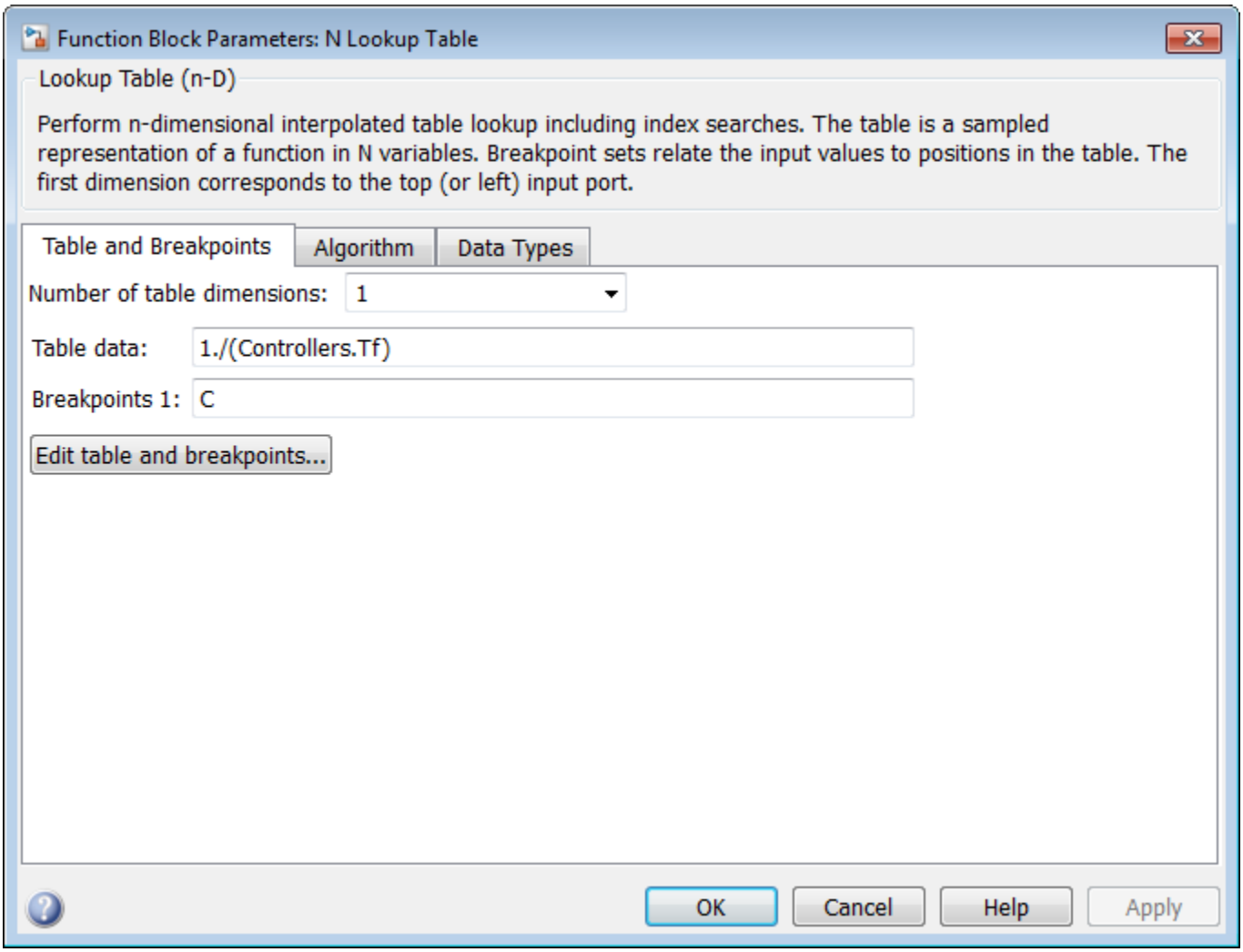
Setting the **Source** parameter to `external` enables the input ports for the coefficients.

The model uses a 1-D Lookup Table block for each of the PID coefficients. In general, for gain-scheduled PID control, use your scheduling variable as the lookup-table input, and the corresponding controller coefficient values as the output. In this example, the CSTR plant output concentration is the lookup table input, and the output is the PID coefficient corresponding to that concentration. To see how the lookup tables are configured, open the `P Lookup Table` block.

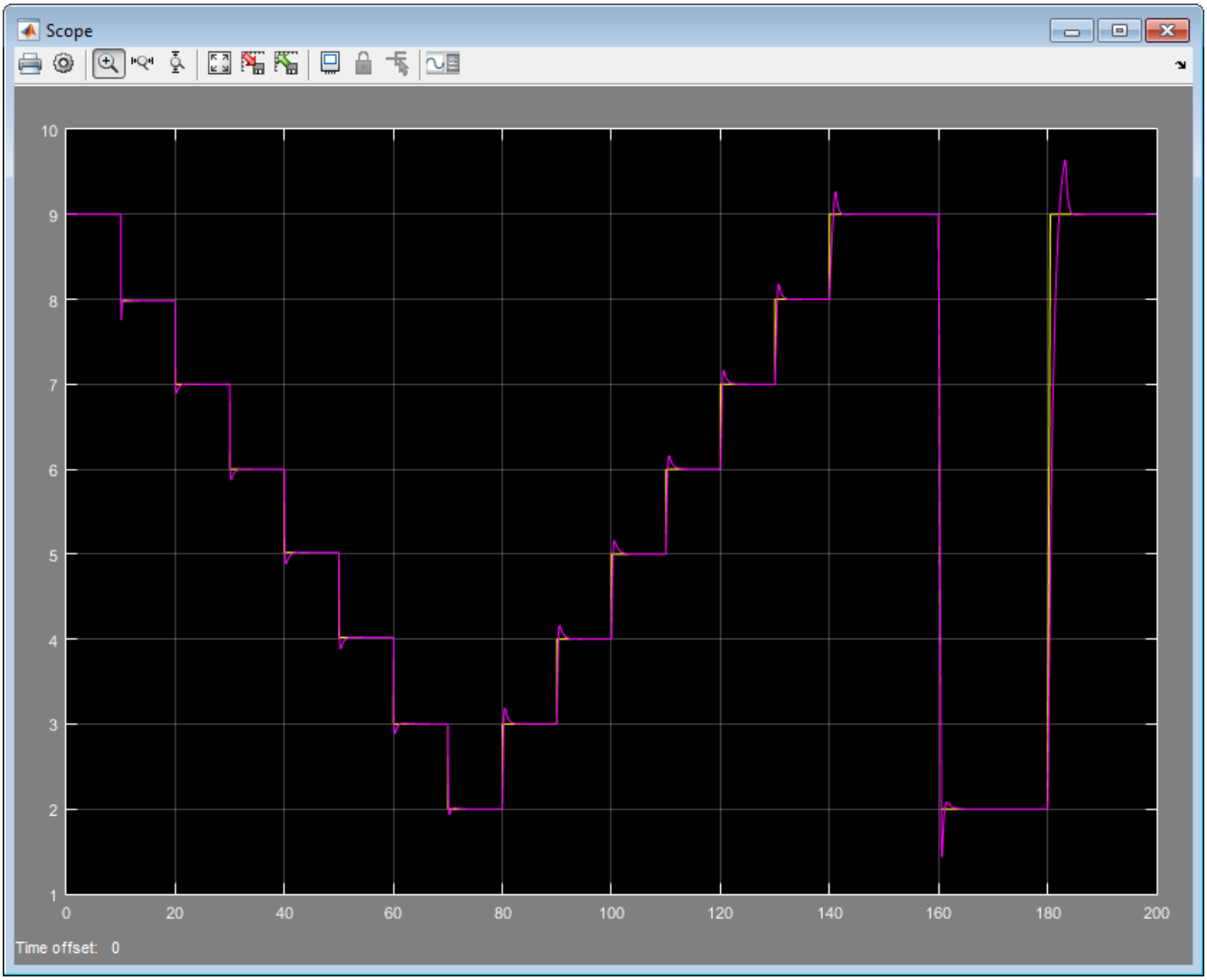


The **Table data** parameter contains the array of proportional coefficients for each controller, `Controllers.Kp`. (For more information about the properties of the `pid` models in the `Controllers` array, see `pid`.) Each entry in this array corresponds to an entry in the array `C` that is entered in the **Breakpoints 1** parameter. For concentration values that fall between entries in `C`, the `P Lookup Table` block performs linear interpolation to determine the value of the proportional coefficient. To set up lookup tables for the integral and derivative coefficients, configure the `I Lookup Table` and `D Lookup Table` blocks using `Controllers.Ki` and `Controllers.Kd`, respectively. For this example, this configuration is already done in the model.

The `pid` models in the `Controllers` array express the derivative filter coefficient as a time constant, `Controllers.Tf` (see the `pid` reference page for more information). However, the PID Controller block expresses the derivative filter coefficient as the inverse constant, `N`. Therefore, the `N Lookup Table` block must be configured to use the inverse of each value in `Controllers.Tf`. Open the `N Lookup Table` block to see the configuration.



Simulate the model. The `Concentration Setpoint` block is configured to step through a sequence of setpoints that spans the operating range between `C = 2` and `C = 9` (shown in yellow on the scope). The simulation shows that the gain-scheduled configuration achieves good setpoint tracking across this range (pink on the scope).



As was shown in “Design Family of PID Controllers for Multiple Operating Points” on page 7-132, the CSTR plant is unstable in the operating range between `C = 4` and `C = 7`. The gain-scheduled PID controllers stabilize the plant and yield good setpoint tracking through the entire unstable region. To fully validate the control design against the nonlinear plant, apply a variety of setpoint test sequences that test the tracking performance for steps of different sizes and directions across the operating range. You can also compare the performance against a design without gain scheduling, by setting all entries in the `Controllers` array equal.

See Also

`pid` | `pidtune` | PID Controller | n-D Lookup Table

More About

- “Design Family of PID Controllers for Multiple Operating Points” on page 7-132

Design Controller for Vehicle Platooning

This example shows how to design a controller for vehicle platooning applications. In a platoon of vehicles, every following vehicle maintains a constant spacing from its preceding vehicle. Vehicles travelling in tightly spaced platoons can improve traffic flow, safety, and fuel economy.

Platooning has the following control objectives [1].

- Individual vehicle stability — Spacing error for each following vehicle converges to zero if the preceding vehicle is traveling at constant speed.
- String stability — Spacing errors do not amplify as they propagate towards the tail of the vehicle string.

Platooning Model

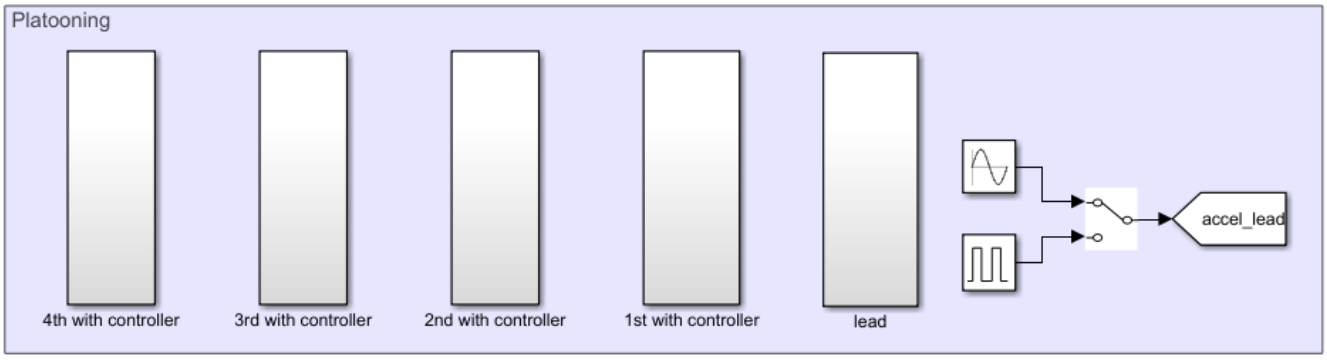
In this example, there are five vehicles in the platoon. Every vehicle is modeled as a truck-trailer system with the following parameters. All lengths are in meters.

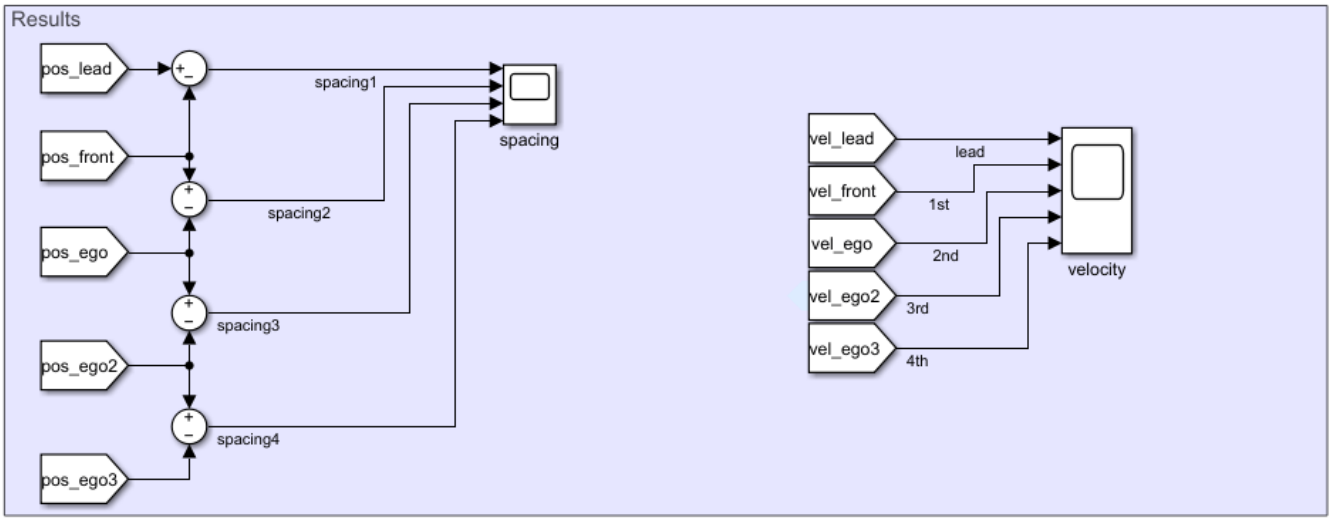
```
L1 = 6;  % Truck length
L2 = 10; % Trailer length
M1 = 1;  % Interconnection length
```

The lead vehicle follows a given acceleration profile. Each trailing vehicle has a controller that controls its acceleration.

Open the Simulink® model.

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





Controller Design

In this example, the trailing vehicles all use the same controller design, which has the following structure.

$$a_{\text{ego}} = C_1 a_{\text{lead}} + (1 - C_1) a_{\text{front}} - K_1 \left(v_{\text{ego}} - v_{\text{lead}} \right) - K_2 \left(x_{\text{ego}} - x_{\text{front}} + L \right)$$

Here:

- a_{ego}, v_{ego}, and x_{ego} are the respective acceleration, velocity, and position of the ego vehicle, that is, the trailing vehicle under control.
- a_{front} and v_{front} are the acceleration and velocity of the vehicle directly in front of the ego vehicle.
- a_{lead} and x_{lead} are the acceleration and position of the lead vehicle in the platoon.

Each vehicle obtains this information from onboard sensors and wireless communication with the other vehicles in the platoon.

The parameters for the controller are as follows.

- L is the desired following distance.
- C_1 is a constant.

- K_1 is a constant.
- K_2 is a PID controller.

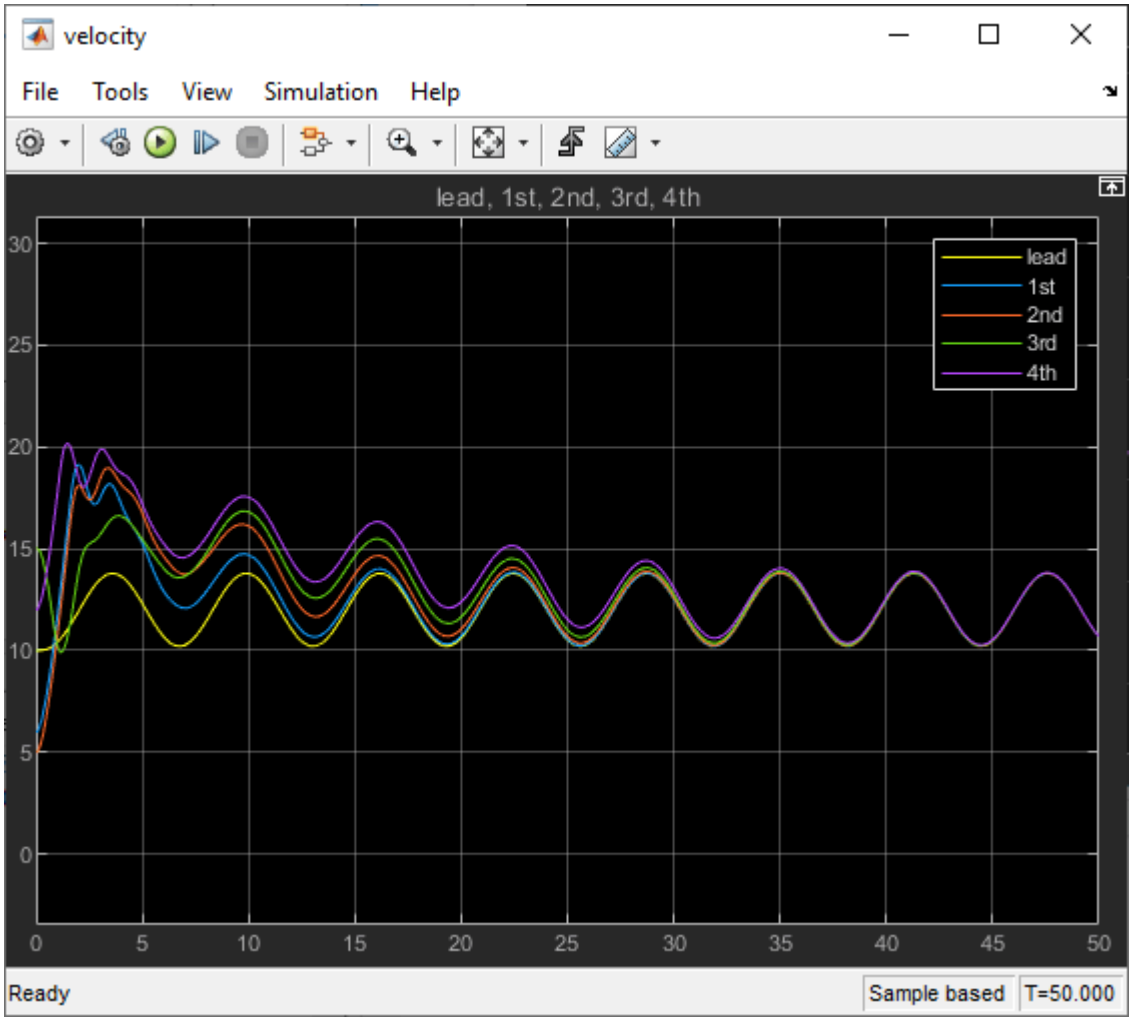
The controller minimizes the velocity error $v_{\text{lead}} - v_{\text{ego}}$ using controller K_1, and minimizes the spacing error $x_{\text{ego}} - x_{\text{front}} + L$ using K_2.

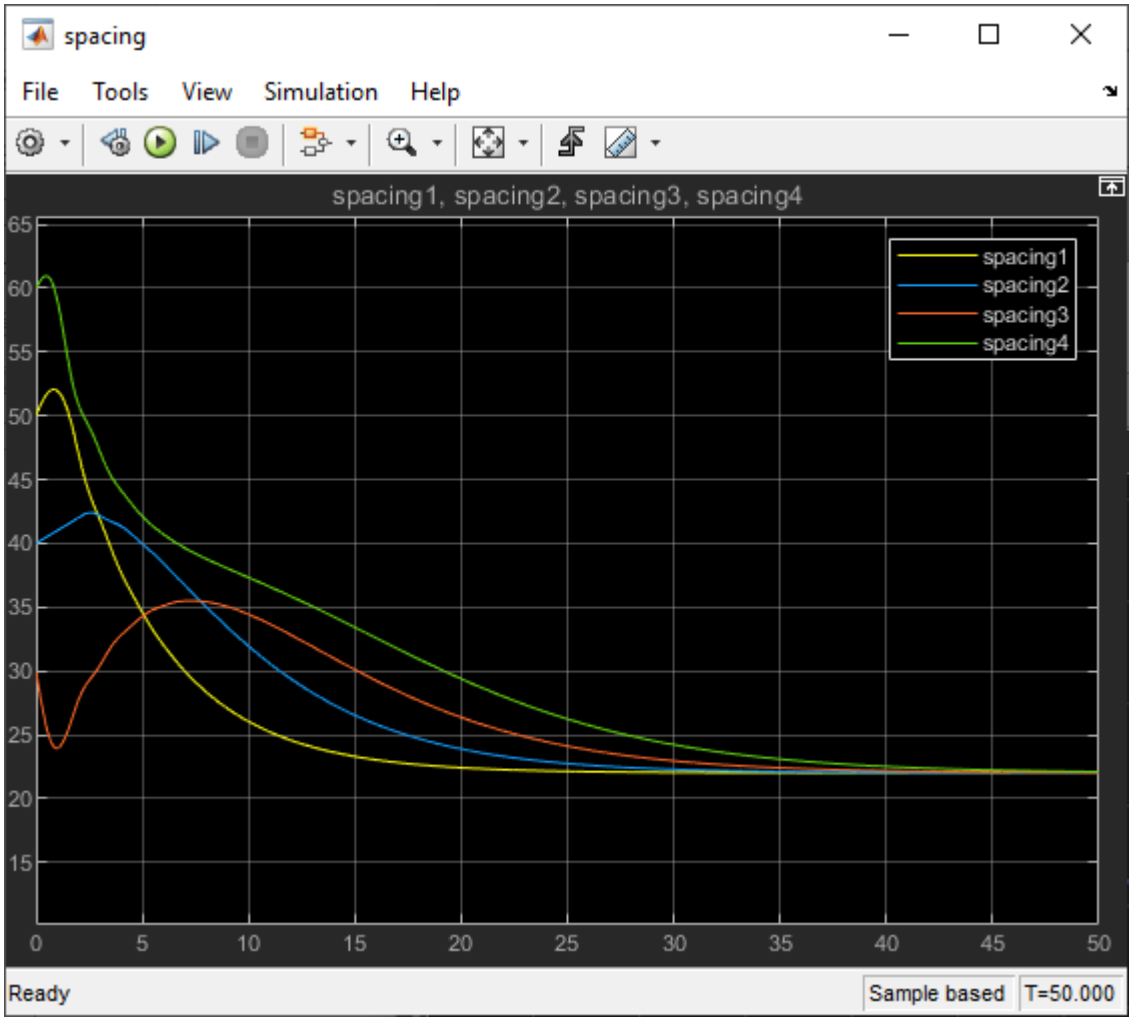
Define the initial controller parameters.

```
L = L1 + L2 + M1 + 5;      % Front-to-front vehicle spacing
C1 = 0.8;                  % Constant gain
K1 = 8;                    % Constant gain
K2 = [2,1];                % PD gains for spacing control [P,D]
```

Run the simulation.

```
sim(mdl);
```





In the top plot, the velocity of each following vehicle converges to the lead velocity. In the bottom plot, the spacing between vehicles converges to the desired spacing.

PID Tuning

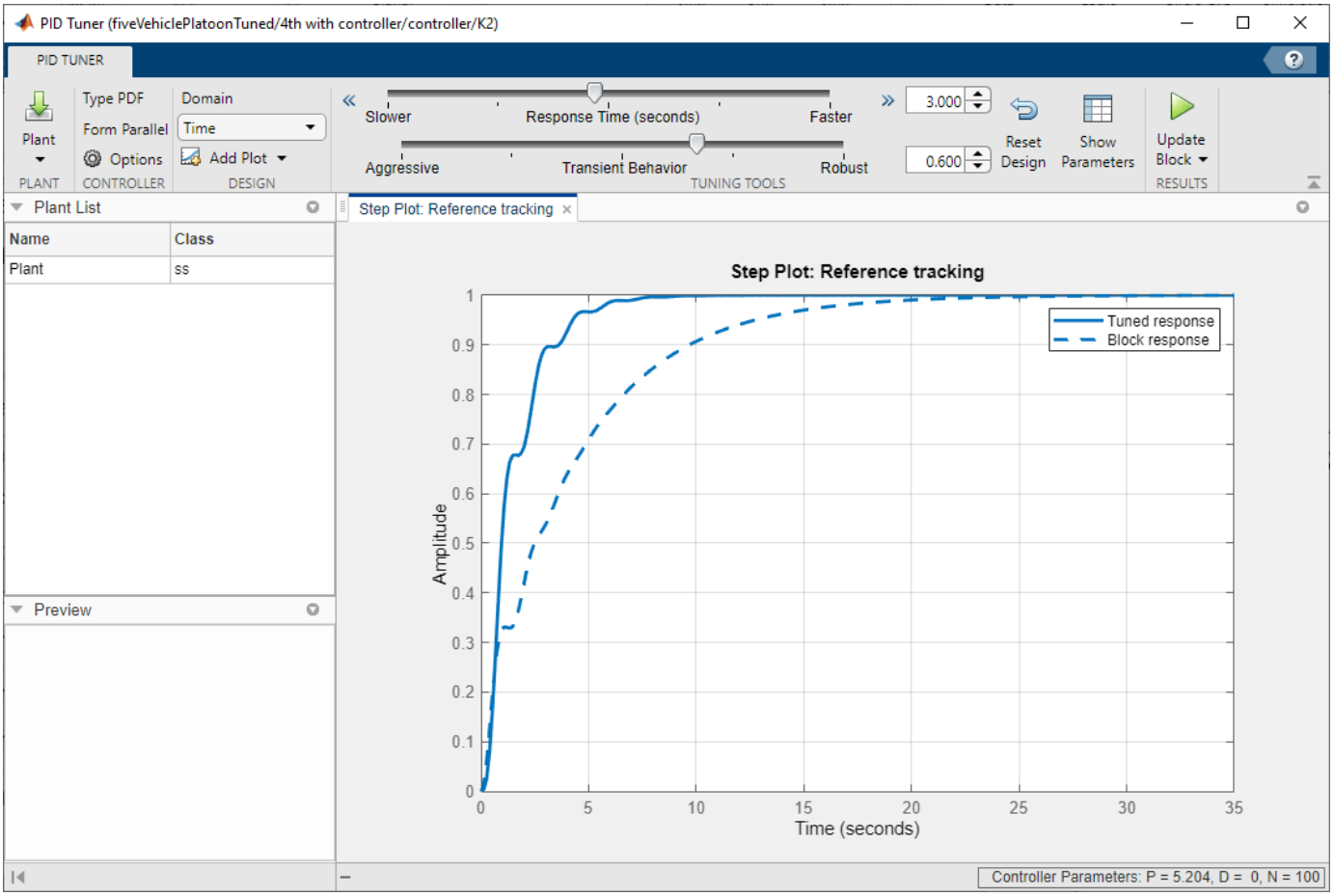
To improve performance, you can tune the gains of the PID controller. As an example, tune the spacing PID controller K_2 for the 4th vehicle in the platoon.

Open the controller.

```
open_system([mdl '/4th with controller/controller/K2'])
```

In the block parameters, under **Select tuning method**, select `Transfer Function Based (PID Tuner App)`. Click **Tune**.

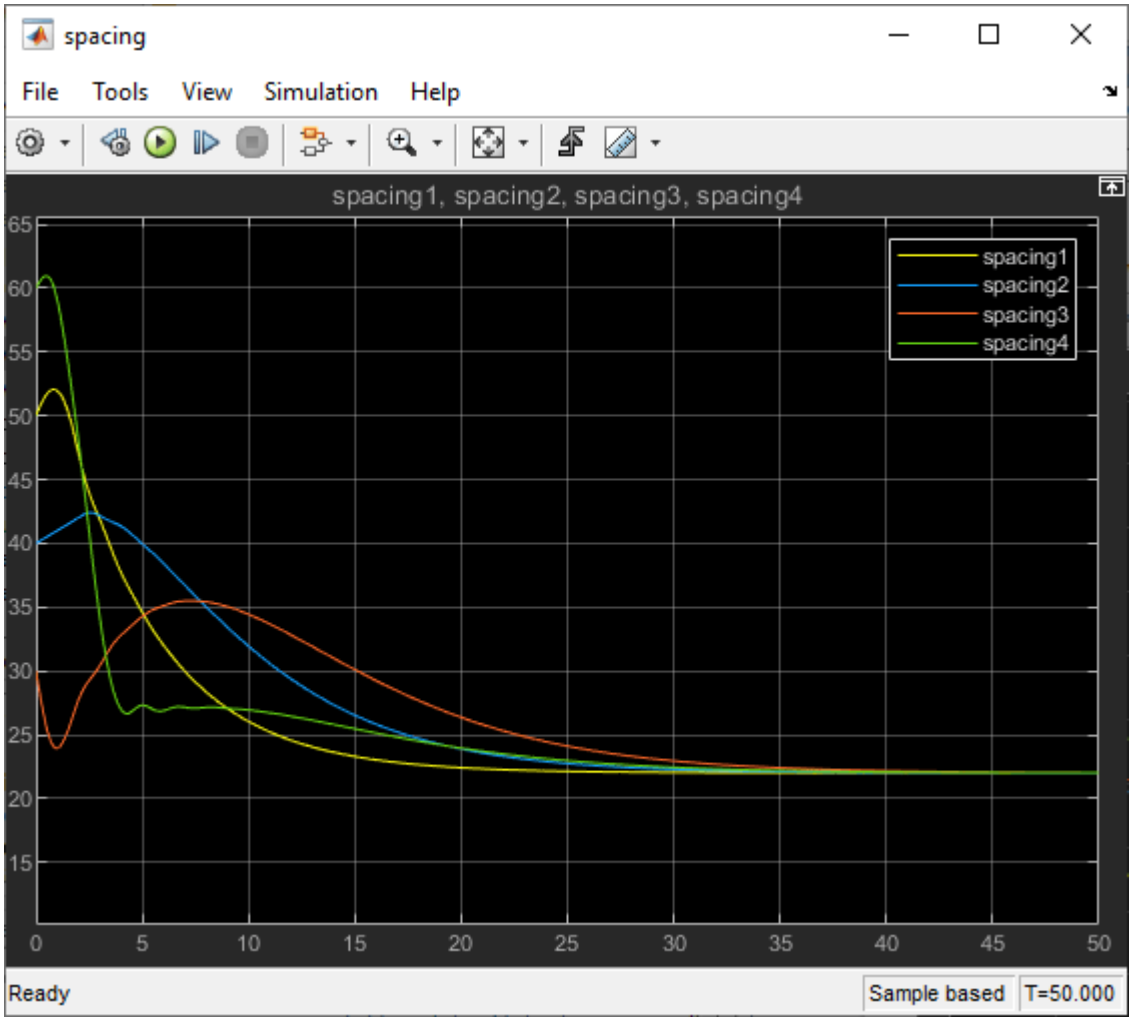
In the **PID Tuner** app, tune the response time and transient behavior of the controller. For example, a **Response Time** 3 seconds and a **Transient Behavior** factor of 0.6 produce a PID controller with a faster response and no overshoot.



To update the controller parameters in the model, click **Update Block**.

For this example, open and simulate a Simulink model with the tuned parameters of the 4th controller set.

```
mdlTuned = 'fiveVehiclePlatoonTuned';
open_system(mdlTuned)
sim(mdlTuned);
```



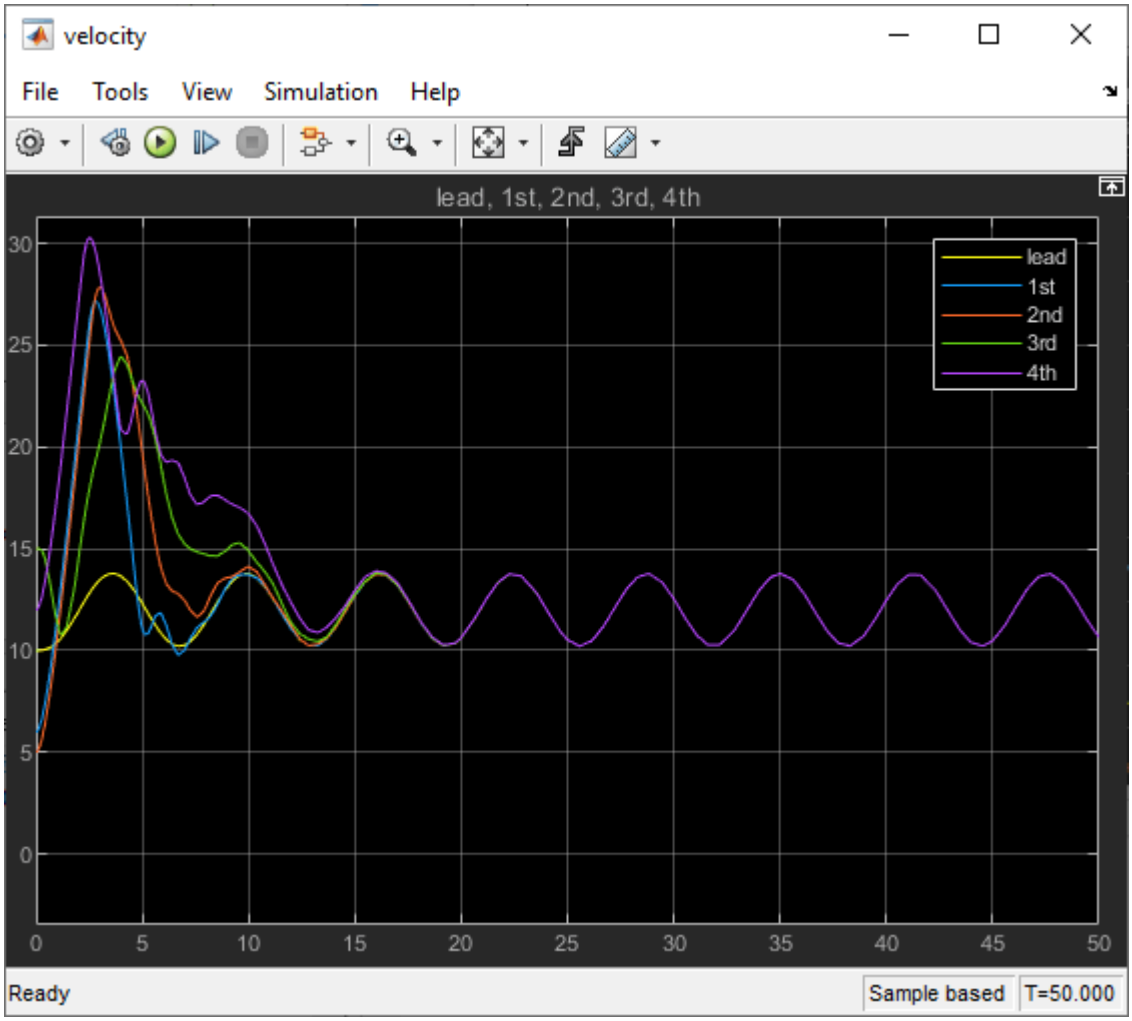
In the spacing plot, the error for the 4th vehicle converges faster than in the previous simulation.

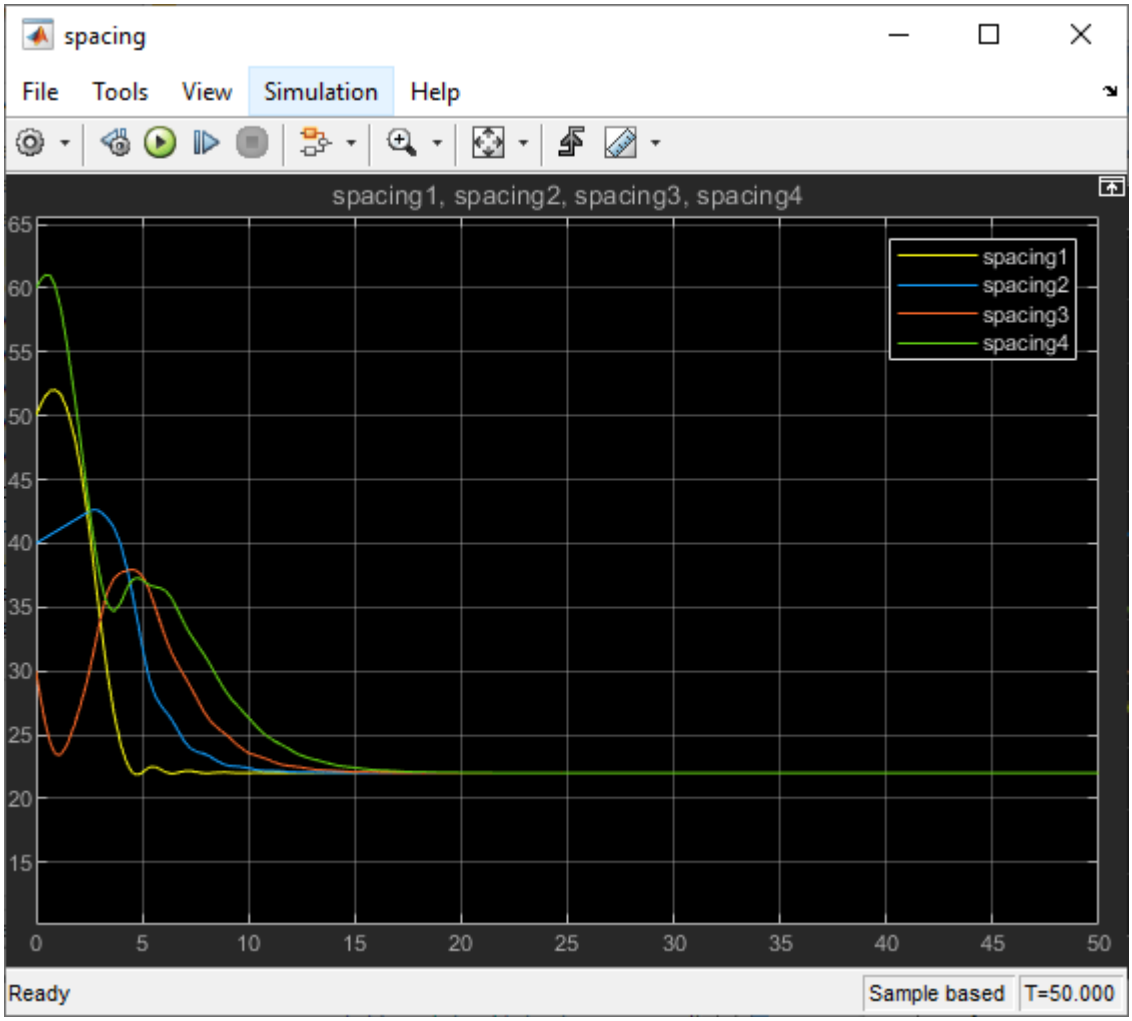
Since the trailing vehicles use the same controller design, set the K_2 gains for the other trailing vehicles to the tuned gains from the 4th vehicle.

```
% Obtained tuned controller gains.
pTuned = get_param([mdlTuned '/4th with controller/controller/K2'],'P');
dTuned = get_param([mdlTuned '/4th with controller/controller/K2'],'D');

% Set gains in other controllers.
set_param([mdlTuned '/3rd with controller/controller/K2'],'P',pTuned)
set_param([mdlTuned '/3rd with controller/controller/K2'],'D',dTuned)
set_param([mdlTuned '/2nd with controller/controller/K2'],'P',pTuned)
set_param([mdlTuned '/2nd with controller/controller/K2'],'D',dTuned)
set_param([mdlTuned '/1st with controller/controller/K2'],'P',pTuned)
set_param([mdlTuned '/1st with controller/controller/K2'],'D',dTuned)
```

Simulate the model with all the trailing vehicle controllers K_2 controllers tuned.

```
sim(mdlTuned);
```


velocity
File
Tools
View
Simulation
Help
lead, 1st, 2nd, 3rd, 4th
lead
1st
2nd
3rd
4th
30
25
20
15
10
5
0
0
5
10
15
20
25
30
35
40
45
50
Ready
Sample based
T=50.000



The error for all the trailing vehicles converge faster than in the original simulation.

References

[1] Rajamani, Rajesh. *Vehicle Dynamics and Control*. 2. ed. Mechanical Engineering Series. New York, NY Heidelberg: Springer, 2012.

See Also

Apps
PID Tuner

Blocks
PID Controller

Related Examples

- “Introduction to Model-Based PID Tuning in Simulink” on page 7-2

Plant Cannot Be Linearized or Linearizes to Zero

When you open **PID Tuner**, it attempts to linearize the model at the operating point specified by the model initial conditions. Sometimes, **PID Tuner** cannot obtain a nonzero linear system for the plant as seen by the PID controller.

How to Fix It

If the plant model in the PID loop cannot be linearized or linearizes to zero, you have several options for obtaining a linear plant model for PID tuning. The following table summarizes some of the options and when they are useful.

Approach	Useful When	More Information
Linearize at a different operating point	There is a known operating point suitable for tuning, such as: • A simulation snapshot time at which the plant is in a linearizable steady state. • Known state values or a previously trimmed operating point at which the plant is linearizable.	"Tune at a Different Operating Point" on page 7-14
Import a linear model of the plant to **PID Tuner**	You have an LTI model of the plant at the desired operating condition for tuning in the MATLAB workspace.	In **PID Tuner**, in the **Plant** menu, select **Import**.
Tune the controller using simulated plant frequency-response data	The plant is not linearizable in any operating condition suitable for tuning.	"Design PID Controller from Plant Frequency-Response Data" on page 7-36
Use system identification to estimate a linear plant model from measured or simulated response data	You have System Identification Toolbox software. An advantage of this approach is that it yields an analytic plant model that you can use for further analysis.	"Interactively Estimate Plant from Measured or Simulated Response Data" on page 7-55

See Also

More About

- "Cannot Find a Good Design in PID Tuner" on page 7-154
- "Introduction to Model-Based PID Tuning in Simulink" on page 7-2

Cannot Find a Good Design in PID Tuner

After adjusting the **PID Tuner** sliders, sometimes you cannot find a design that meets your design requirements when you analyze the **PID Tuner** response plots.

How to Fix It

Try a different PID controller type. It is possible that your controller type is not the best choice for your plant or your requirements.

For example, the closed-loop step response of a P- or PD-controlled system can settle on a value that is offset from the setpoint. If you require a zero steady-state offset, adding an integrator (using a PI or PID controller) can give better results.

As another example, sometimes 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 controller block dialog box:

- Select a different controller type from the **Controller** drop-down menu.
- Click **Apply** to save the change.
- Click **Tune** to instruct **PID Tuner** to tune the parameters for the new controller type.

If you cannot find any satisfactory controller with **PID Tuner**, PID control possibly is not sufficient for your requirements. You can design more complex controllers using **Control System Designer**.

See Also

PID Controller | Discrete PID Controller | PID Controller (2DOF) | Discrete PID Controller (2DOF)

More About

- “Simulated Response Does Not Match PID Tuner Response” on page 7-155
- “Control System Designer Tuning Methods” on page 9-4
- “Introduction to Model-Based PID Tuning in Simulink” on page 7-2

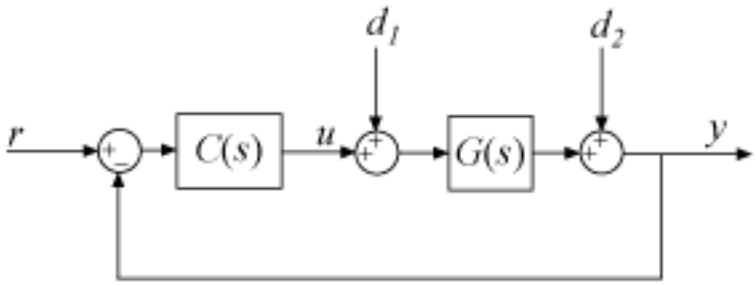
Simulated Response Does Not Match PID Tuner Response

When you run your Simulink model using the PID gains computed by **PID Tuner**, the simulation output differs from the **PID Tuner** response plot.

There are several reasons why the simulated model can differ from the **PID Tuner** response plot. If the simulated result meets your design requirements (despite differing from the **PID Tuner** response), you do not need to refine the design further. If the simulated result does not meet your design requirements, see “Cannot Find Acceptable PID Design in Simulated Model” on page 7-157.

Some causes for a difference between the simulated and **PID Tuner** responses include:

- The reference signals or disturbance signals in your Simulink model differ from the step signals that **PID Tuner** uses. If you need step signals to evaluate the performance of the PID controller in your model, change the reference signals in your model to step signals.
- The structure of your model differs from the loop structure that **PID Tuner** designs for. **PID Tuner** assumes the loop configuration shown in the following figure.



  As the figure illustrates, **PID Tuner** designs for a PID controller in the feedforward path of a unity-gain feedback loop. If your Simulink model differs from this structure, or injects a disturbance signal in a different location, your simulated response differs from the **PID Tuner** response.
- You have enabled nonlinear features in the PID Controller block in your model, such as saturation limits or anti-windup circuitry. **PID Tuner** ignores nonlinear settings in the PID Controller block, which can cause **PID Tuner** to give a different response from the simulation.
- Your Simulink model has strong nonlinearities in the plant that make the linearization invalid over the full operating range of the simulation.
- You selected an operating point using **PID Tuner** that is different from the operating point saved in the model. In this case, **PID Tuner** has designed a controller for a different operating point than the operating point that begins the simulation. Simulate your model using the **PID Tuner** operating point by initializing your Simulink model with this operating point. See “Simulate Simulink Model at Specific Operating Point” on page 1-93.

See Also

PID Controller | Discrete PID Controller | PID Controller (2DOF) | Discrete PID Controller (2DOF)

More About

- “Cannot Find Acceptable PID Design in Simulated Model” on page 7-157
- “Introduction to Model-Based PID Tuning in Simulink” on page 7-2

Cannot Find Acceptable PID Design in Simulated Model

When you run your Simulink model using the PID gains computed by **PID Tuner**, the simulation output may not meet your design requirements.

How to Fix It

Sometimes, PID control is not adequate to meet the control requirements for your plant. If you cannot find a design that meets your requirements when you simulate your model, consider designing a more complex controller using **Control System Designer**.

If you have enabled saturation limits in the PID Controller block without antiwindup circuitry, enable antiwindup circuitry. You can enable antiwindup circuitry in two ways:

- Activate the PID Controller block antiwindup circuitry on the **PID Advanced** tab of the block dialog box.
- Use the PID Controller block tracking mode to implement your own antiwindup circuitry external to the block. Activate the PID Controller block tracking mode on the **PID Advanced** tab of the block dialog box.

To learn more about both ways of implementing antiwindup circuitry, see “Anti-Windup Control Using a PID Controller”.

After enabling antiwindup circuitry, run the simulation again to see whether controller performance is acceptable.

If the loop response is still unacceptable, try slowing the response of the PID controller. To do so, reduce the response time or the bandwidth in **PID Tuner**. See “Refine the Design” on page 7-12.

You can also try implementing gain-scheduled PID control to help account for nonlinearities in your system. See “Design Family of PID Controllers for Multiple Operating Points” on page 7-132 and “Implement Gain-Scheduled PID Controllers” on page 7-139.

If you still cannot get acceptable performance with PID control, consider using a more complex controller. See **Control System Designer**.

See Also

PID Controller | Discrete PID Controller | PID Controller (2DOF) | Discrete PID Controller (2DOF)

More About

- “Simulated Response Does Not Match PID Tuner Response” on page 7-155
- “Controller Performance Deteriorates When Switching Time Domains” on page 7-158
- “Control System Designer Tuning Methods” on page 9-4
- “Introduction to Model-Based PID Tuning in Simulink” on page 7-2

Controller Performance Deteriorates When Switching Time Domains

After you obtain a well-tuned, continuous-time controller using **PID Tuner**, you can discretize the controller using the **Time Domain** selector button in the PID Controller block dialog box. Sometimes, the resulting discrete-time controller performs poorly or even becomes unstable.

How to Fix It

In some cases, you can improve performance by adjusting the sample time by trial and error. However, this procedure can yield a poorly tuned controller, especially where your application imposes a limit on the sample time. Instead, if you change time domains and the response deteriorates, click **Tune** in the PID Controller block dialog box to design a new controller.

Note If the plant and controller time domains differ, **PID Tuner** discretizes the plant (or converts the plant to continuous time) to match the controller time domain. If the plant and controller both use discrete time, but have different sample times, **PID Tuner** resamples the plant to match the controller. All conversions use the `tustin` method (see “Continuous-Discrete Conversion Methods”.

See Also

PID Controller | Discrete PID Controller | PID Controller (2DOF) | Discrete PID Controller (2DOF)

More About

- “When Tuning the PID Controller, the D Gain Has a Different Sign from the I Gain” on page 7-159
- “Introduction to Model-Based PID Tuning in Simulink” on page 7-2

When Tuning the PID Controller, the D Gain Has a Different Sign from the I Gain

When you design a controller using **PID Tuner**, the resulting derivative gain, D, can have a different sign from the integral gain I. **PID Tuner** always returns a stable controller, even if one or more gains are negative.

For example, the following expression gives the PID controller transfer function in `Ideal` form:

$$c = P\left(1 + \frac{I}{s} + \frac{Ds}{\frac{s}{N} + 1}\right) = P\frac{(1 + DN)s^2 + (I + N)s + IN}{s(s + N)}$$

For a stable controller, all three numerator coefficients require positive values. Because N is positive, $IN > 0$ requires that I is also positive. However, the only restriction on D is $(1 + DN) > 0$. Therefore, as long as $DN > -1$, a negative D still yields a stable PID controller.

Similar reasoning applies for any controller type and for the `Parallel` controller form. For more information about controller transfer functions, see the PID controller block reference pages.

See Also

PID Controller | Discrete PID Controller | PID Controller (2DOF) | Discrete PID Controller (2DOF)

More About

- “Introduction to Model-Based PID Tuning in Simulink” on page 7-2

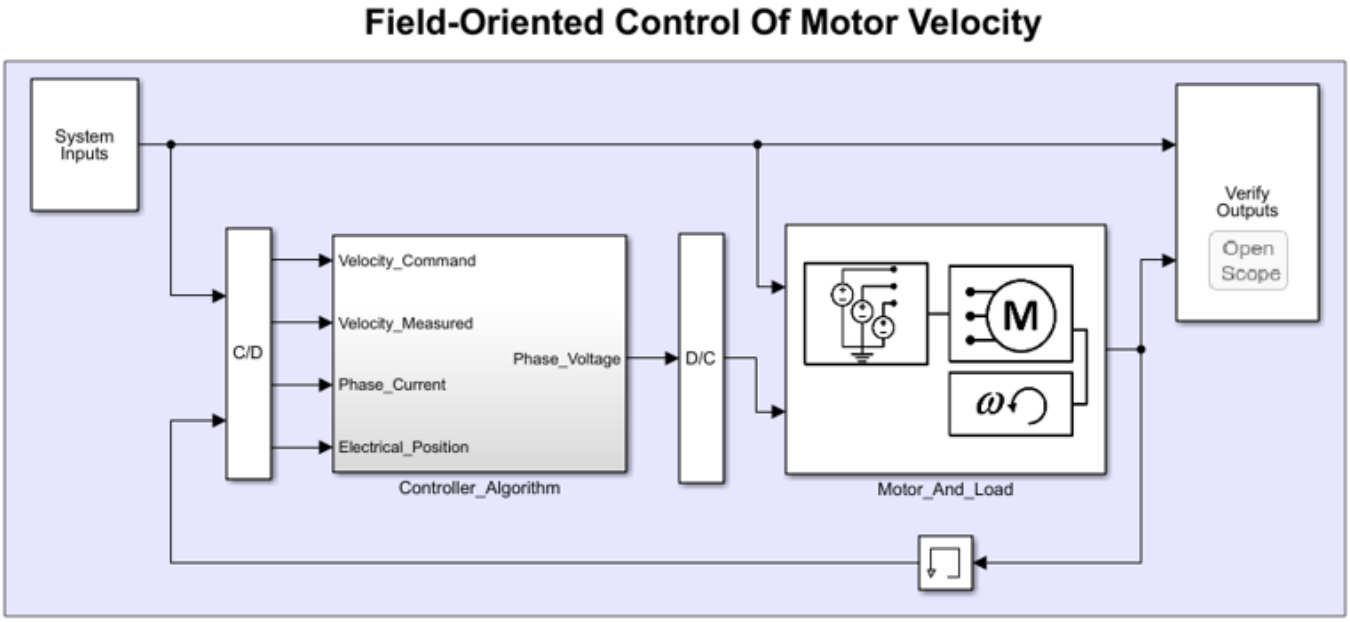
Tune Field-Oriented Controllers Using SYSTUNE

This example shows how to use the `systune` command to tune Field-Oriented Control (FOC) for a permanent magnet synchronous machine (PMSM) based on a frequency response estimation (FRE) result.

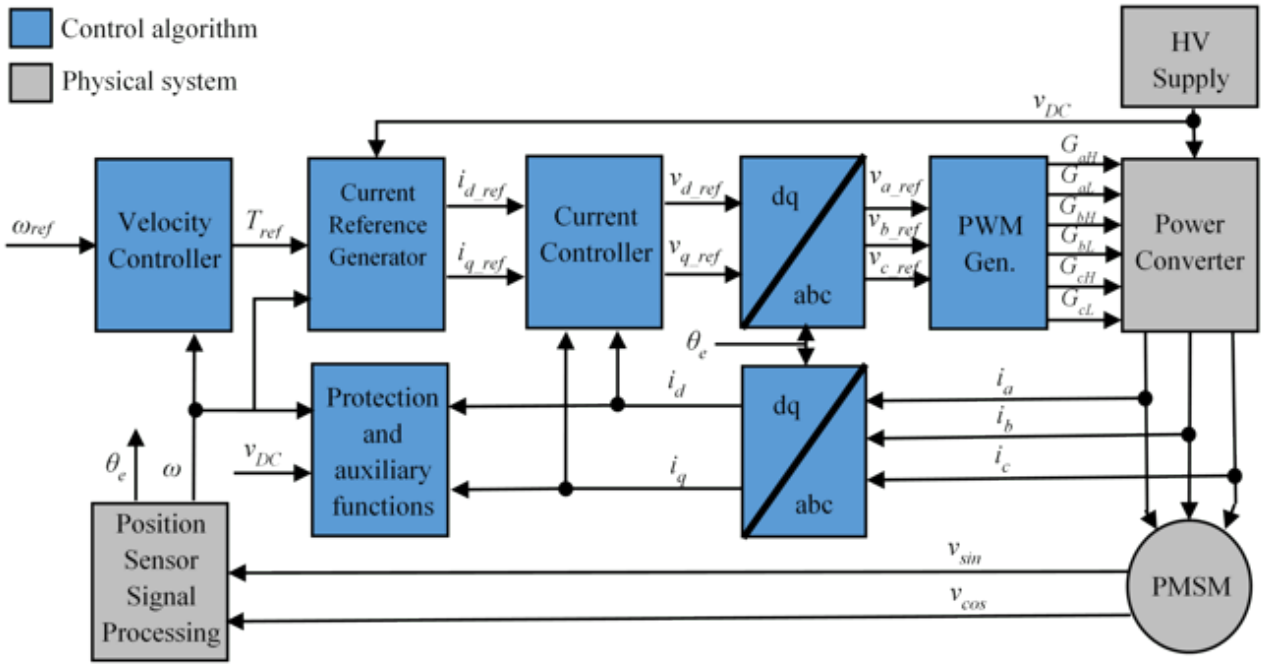
Field-Oriented Control

In this example, field-oriented control (FOC) for a permanent magnet synchronous machine (PMSM) is modeled in Simulink® using Simscape™ Electrical™ components.

```
mdl = 'scdfocmotorSystune';
open_system(mdl)
SignalBuilderPath = [mdl,'/System_Inputs/Signal_Builder_Experiments'];
```



Field-oriented control controls 3-phase stator currents as a vector. FOC is based on projections, which transform a 3-phase time-dependent and speed-dependent system into a two-coordinate time-invariant system. These transformations are the Clarke Transformation, Park Transformation, and their respective inverse transforms. These transformations are implemented as blocks within the Controller_Algorithm subsystem.



The advantages of using FOC to control AC motors include:

- Torque and flux controlled directly and separately
- Accurate transient and steady-state management
- Similar performance compared to DC motors

The Controller_Algorithm subsystem contains all three PI controllers. The outer-loop PI controller regulates the speed of the motor. The two inner-loop PI controllers control the d-axis and q-axis currents separately. The command from the outer loop PI controller directly feeds to the q-axis to control torque. The command for the d-axis is zero for PMSM because the rotor flux is fixed with a permanent magnet for this type of AC motor.

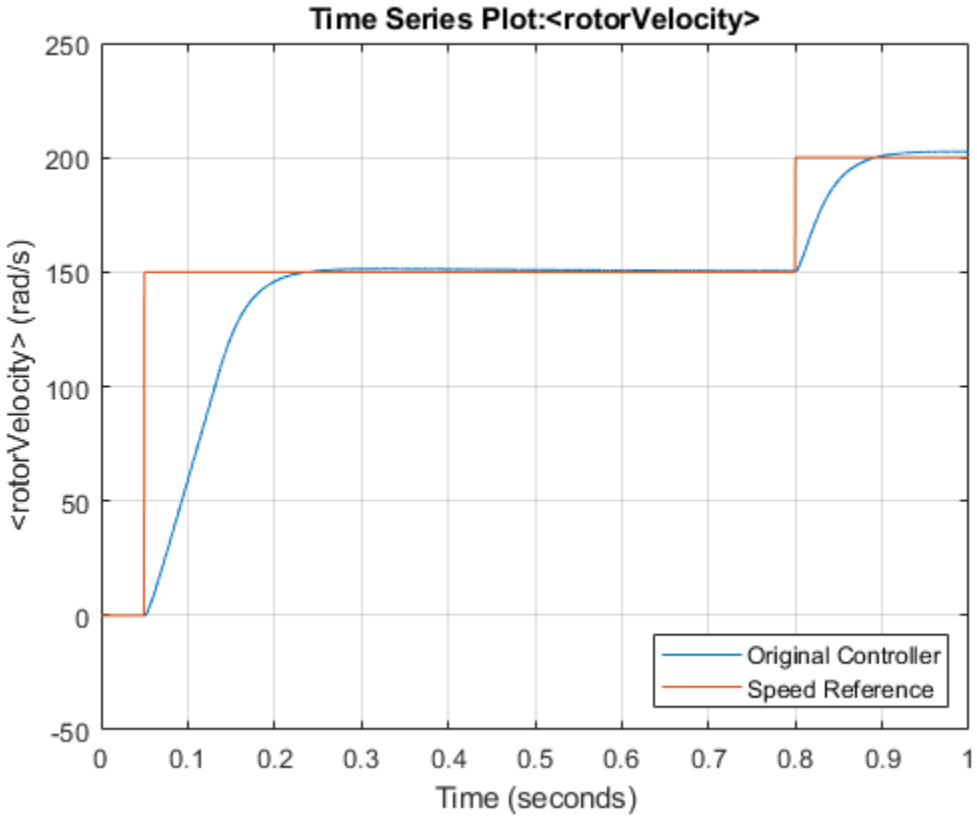
Before tuning controllers, examine the speed responses with the original controllers, and save the simulation results to a MAT-file, `SystunedSpeed.mat`. The existing speed PI controller has gains of P = 0.08655 and I = 0.1997. The current PI controllers both have gains of P = 1 and I = 200.

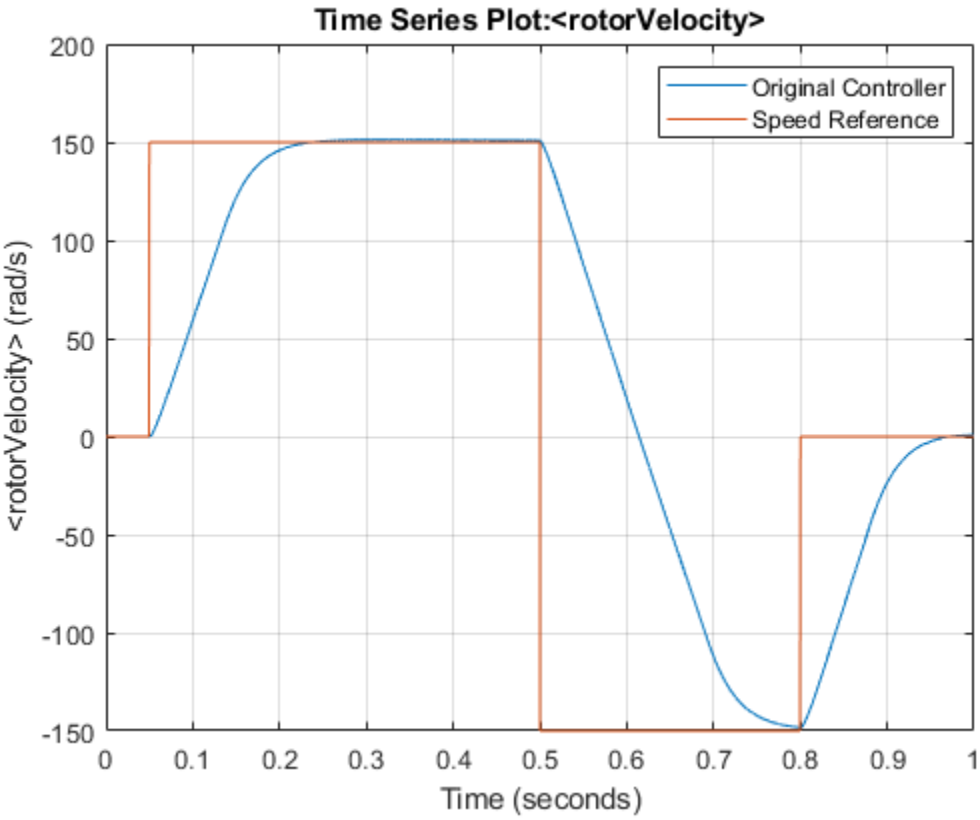
```
scdfocmotorSystuneOriginalResponse
```

Plot the speed response with the original controllers. The plots exhibit steady-state errors and relatively slow transient behavior. You can tune the controllers to achieve a better performance.

```
figure
plot(logsout_original_oneside{2}.Values);
hold on
plot(logsout_original_oneside{1}.Values);
legend('Original Controller','Speed Reference','Location','southeast');
grid on
hold off

figure
plot(logsout_original_twoside{2}.Values);
```

```
hold on
plot(logsout_original_twoside{1}.Values);
legend('Original Controller','Speed Reference','Location','northeast');
grid on
hold off
```



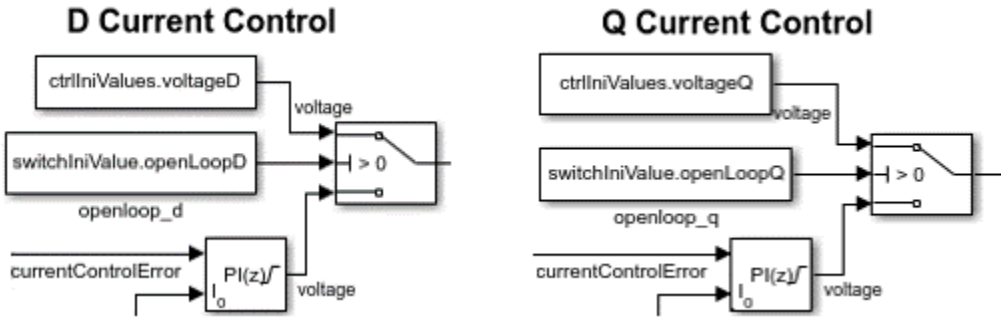


Collect Frequency Response Data

To collect frequency response data, find an operating point at a speed of `150` rad/sec, specify linear analysis points, define input signals, and estimate the frequency response.

Disconnect the original controllers, and simulate the open-loop system model with VD and VQ commands. To reach the operating point, specify initial voltages of -0.1 V for VD and 3.465 V for VQ using the `ctrlIniValues` structure. Constant voltage command blocks are connected by setting switch signals in the `switchIniValue` structure.

```
switchIniValue.openLoopD = 1;
switchIniValue.openLoopQ = 1;
ctrlIniValues.voltageD = -0.1;
ctrlIniValues.voltageQ = 3.465;
```



Capture a simulation snapshot at 3 sec as the operating point for frequency response estimation.

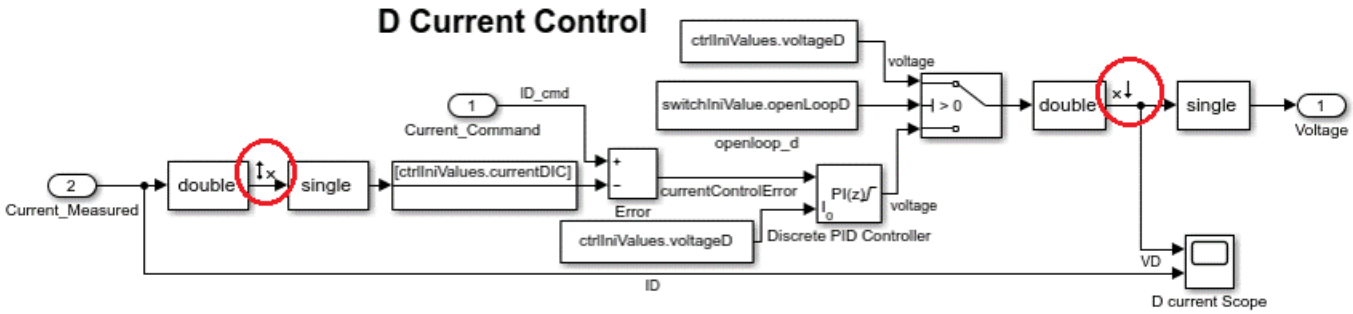
```
signalbuilder(SignalBuilderPath, 'activegroup', 1);
op = findop(mdl,3);
```

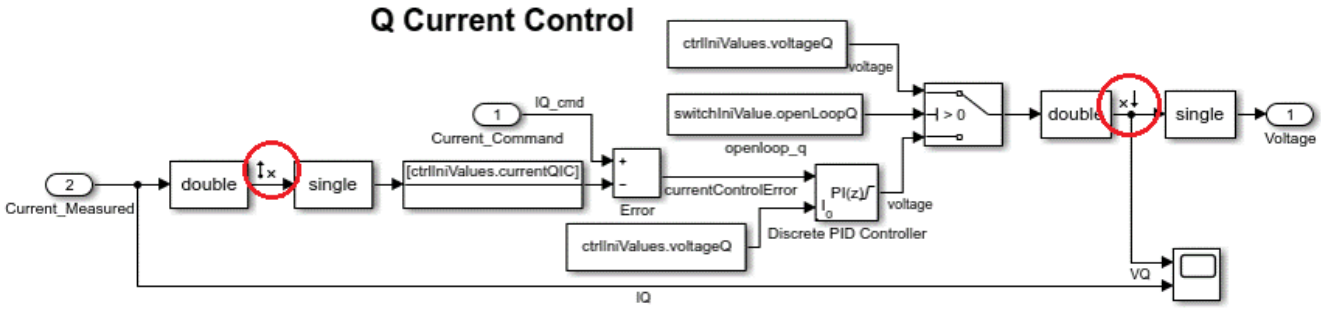
Use the simulation snapshot operating point as the initial condition of the model. Change the model initial values in the `ctrlIniValues` structure to be at this steady state. For the d-axis current controller, the current ID is `0` A. For the q-axis current controller, the current IQ is `0.1` A. For the outer-loop speed controller, the reference current is `0.122` A and the speed is at `150` rad/s. For the PMSM plant, set the rotor velocity in the `pmsm` structure to `150` rad/s.

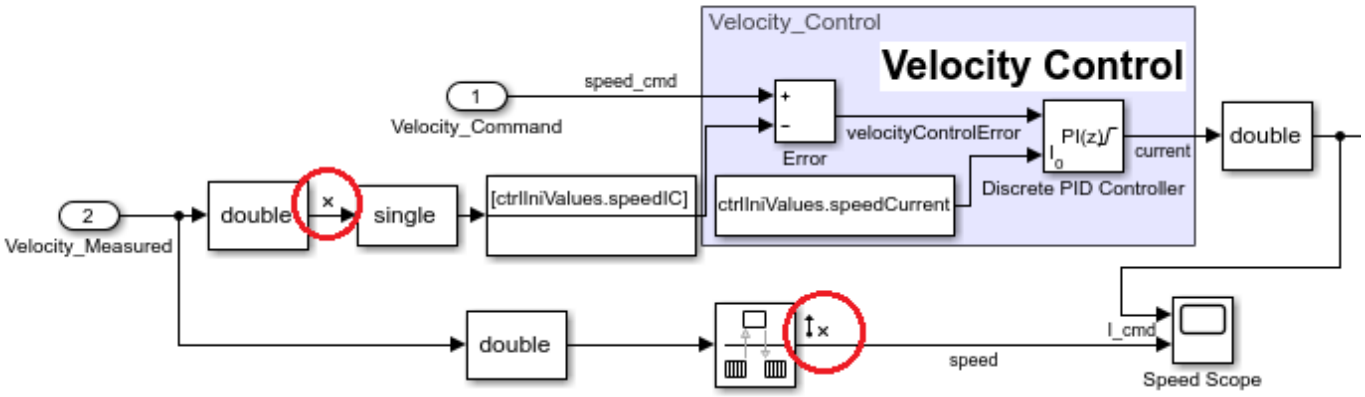
```
set_param(mdl,'LoadInitialState','on');
set_param(mdl,'InitialState','getstatestruct(op)');
ctrlIniValues.currentDIC = 0;
ctrlIniValues.currentQIC = 0.1;
ctrlIniValues.speedIC = 150;
ctrlIniValues.speedCurrent = 0.122;
pmsm.RotorVelocityInit = 150;
```

Add linear analysis points to the model for frequency response estimation. Add open-loop input points to VD and VQ. Add open-loop output points to ID, IQ, and speed. In addition, add a loop break analysis point to the speed measurement.

```
io = getlinio(mdl);
```





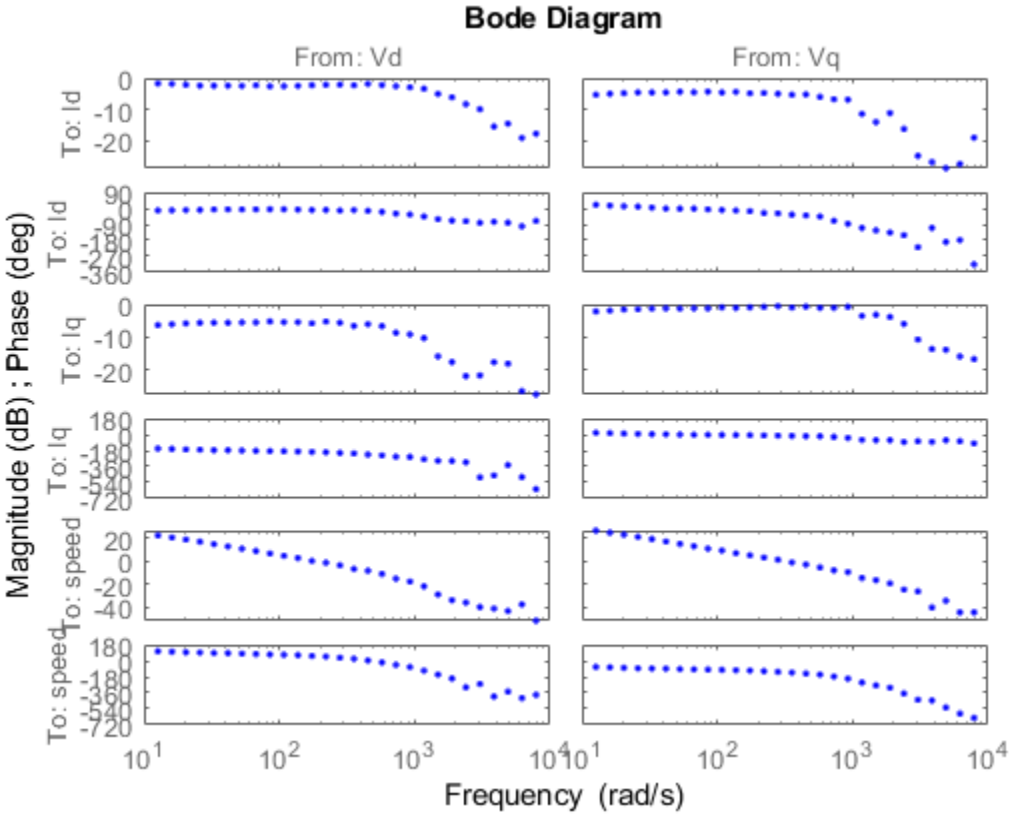


Define input sinestream signal from `10` to `10,000` rad/s with a fixed sample time of `4e-6` s, that is, the sample time of the current control loop `sampleTime.CurrentControl`. The sinestream signal magnitude is `0.25` V. This magnitude ensures that the plant is properly excited within the saturation limit. If the excitation amplitude is either too large or too small, it produces inaccurate frequency response estimation results.

```
in = frest.createFixedTsSinestream(sampleTime.CurrentControl,{10,1e4});
in.Amplitude = 0.25;
```

Estimate the frequency response at the specified steady state operating point `op`, using the linear analysis points in `io` and the input signals in `in`. After finishing the frequency response estimation, modify the input and output channel names in the resulting model, and plot the frequency response.

```
estsys = frestimate(mdl,op,io,in);
estsys.InputName = {'Vd','Vq'};
estsys.OutputName = {'Id','Iq','speed'};
bode(estsys,'.')
```

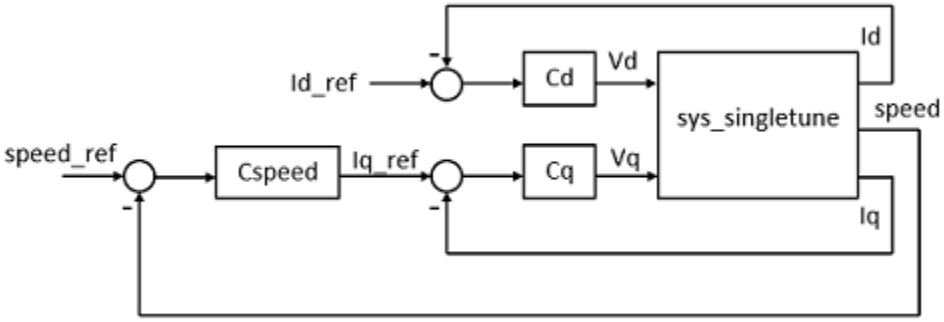


Tune Control System Using `systune`

Obtain a state-space linear system model from the frequency response estimation result. Using an option set for the `ssest` function, set the numerical search method used for this iterative parameter estimation as the Levenberg-Marquardt least-squares search. Estimate a state-space model with four states and a period of `4e-6` seconds. This step requires System Identification Toolbox™ software.

```
optssest = ssestOptions('SearchMethod','lm');
optssest.Regularization.Lambda = 0.1;
sys_singletune = ssest(estsys,4,'Ts',sampleTime.CurrentControl,optssest);
```

In order to tune all three PI controllers in the PMSM FOC model, construct a control system as shown in the following block diagram.



Define three tunable discrete-time PID blocks and their I/Os for d-axis current control, q-axis current control, and speed control. The sample times of these discrete-time PID controllers must be consistent, which is the same as the current control loop sample time. To ensure a better approximation of faster controllers as compared to the original slower controllers, set the discrete integrator formula for each PID controller to `'Trapezoidal'`.

```
Cd = tunablePID('Cd','pi',sampleTime.CurrentControl);
Cd.IFormula = 'Trapezoidal';
Cd.u = 'Id_e';
Cd.y = 'Vd';

Cq = tunablePID('Cq','pi',sampleTime.CurrentControl);
Cq.IFormula = 'Trapezoidal';
Cq.u = 'Iq_e';
Cq.y = 'Vq';

Cspeed = tunablePID('Cspeed','pi',sampleTime.CurrentControl);
Cspeed.IFormula = 'Trapezoidal';
Cspeed.u = 'speed_e';
Cspeed.y = 'Iq_ref';
```

Create three summing junctions for the inner and outer feedback loops.

```
sum_speed = sumblk('speed_e = speed_ref - speed');
sum_id = sumblk('Id_e = Id_ref - Id');
sum_iq = sumblk('Iq_e = Iq_ref - Iq');
```

Define inputs, outputs, and analysis points for controller tuning.

```
input = {'Id_ref','speed_ref'};
output = {'Id','Iq','speed'};
APs = {'Iq_ref','Vd','Vq','Id','Iq','speed'};
```

Finally, assemble the complete control system, `ST0`, using these components.

```
ST0 = connect(sys_singletune,Cd,Cq,Cspeed,sum_speed,sum_id,sum_iq,input,output,APs);
```

Define tuning goals, including tracking and loop shape goals to ensure command tracking, as well as gain goals to prevent saturations. For the speed controller, set the tracking bandwidth to `150` rad/s. This bandwidth is used in both the tracking and loop shape goals. Additionally, set the DC error to `0.001` to reflect a maximum steady-state error of `0.1`%. Set the peak error to `10`. For the d-axis current controller, set the tracking bandwidth to `2500` rad/s, which is much faster than the outer-loop

speed controller. To prevent saturating controllers, specify goals to constrain the gains for all three controllers.

```
TR1 = TuningGoal.Tracking('speed_ref','speed',2/150,0.001,10);
TR2 = TuningGoal.Tracking('Id_ref','Id',2/2500);
LS1 = TuningGoal.LoopShape('Id',2500);
LS2 = TuningGoal.LoopShape('speed',150);
MG1 = TuningGoal.Gain('speed_ref','Iq_ref',2);
MG2 = TuningGoal.Gain('speed_ref','Vq',50);
MG3 = TuningGoal.Gain('Id_ref','Vd',20);
```

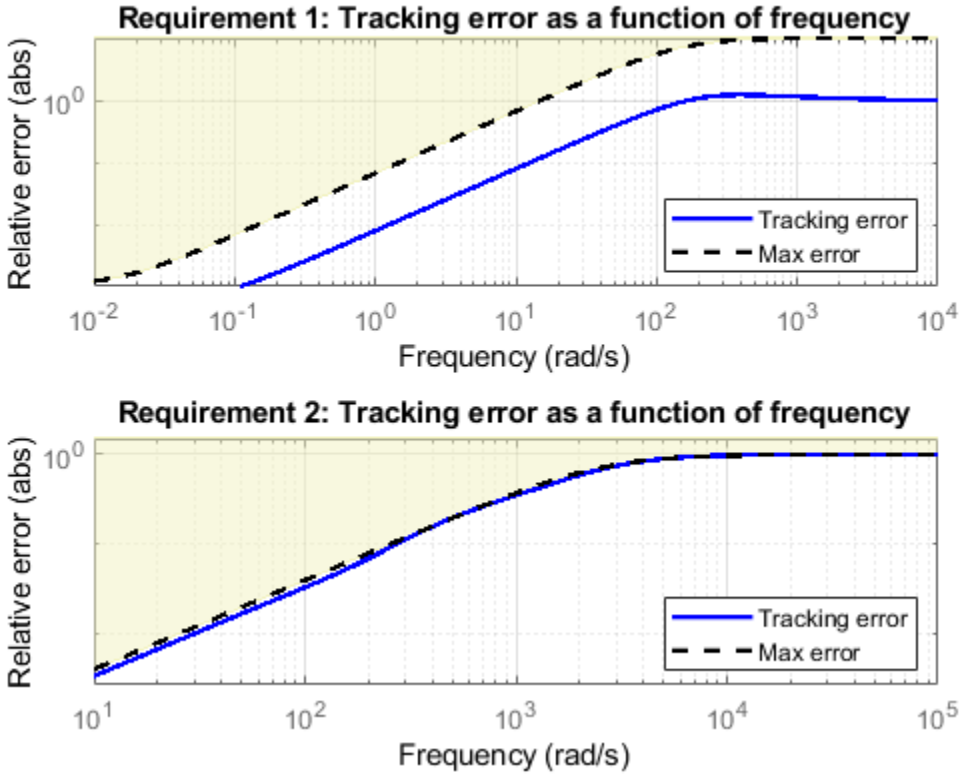
Tune all three PI controllers using `systune` with all tuning goals based on the constructed model `ST0`. To increase the likelihood of finding parameter values that meet all design requirements, set options for `systune` to run five additional optimizations starting from five randomly generated parameter values.

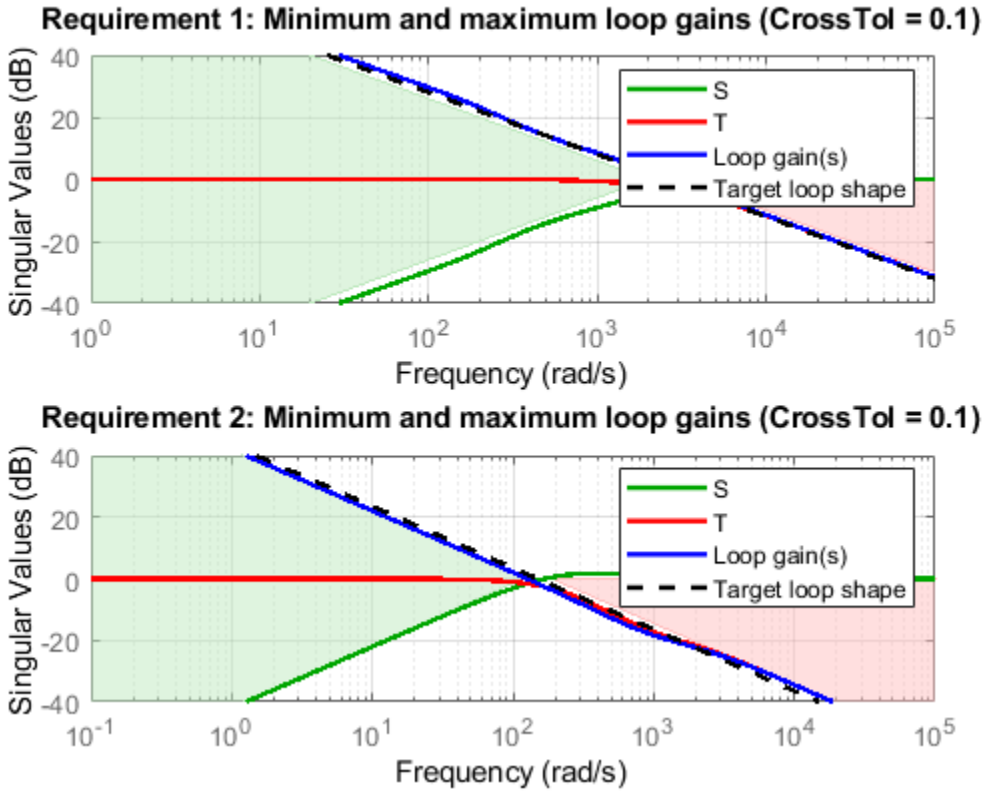
```
opt = systuneOptions('RandomStart',5);
rng(2)
[ST1,fSoft] = systune(ST0,[TR1,TR2,LS1,LS2,MG1,MG2,MG3],opt);
```

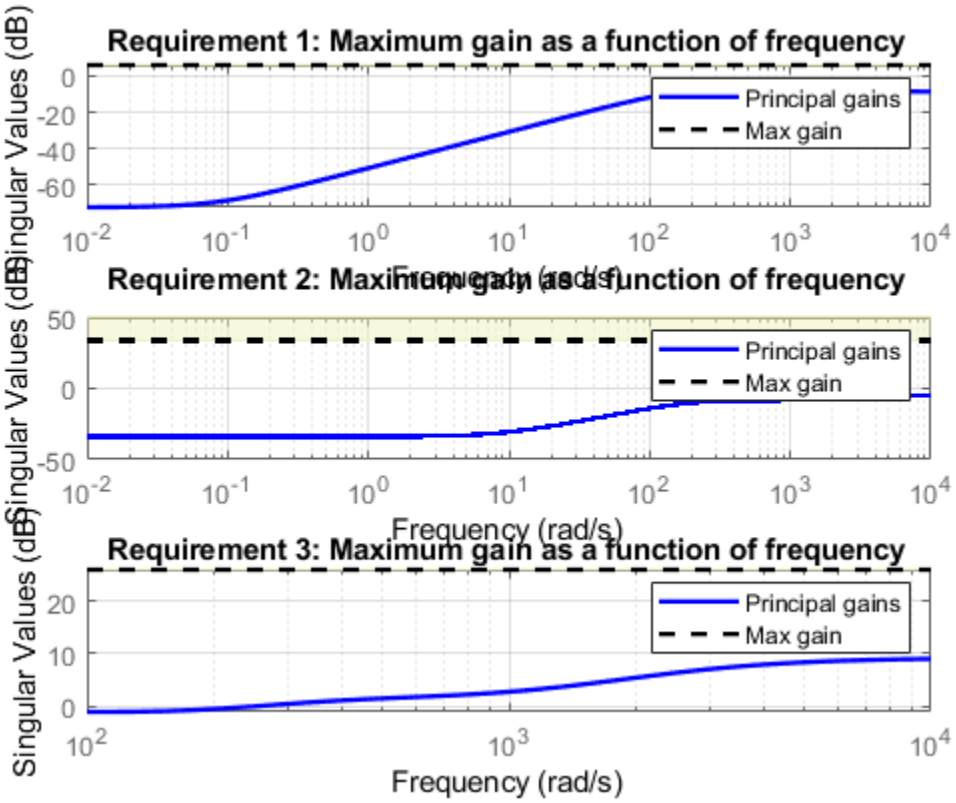
```
Final: Soft = 1.01, Hard = -Inf, Iterations = 73
       Some closed-loop poles are marginally stable (decay rate near 1e-07)
Final: Soft = 1.01, Hard = -Inf, Iterations = 87
Final: Soft = 1.01, Hard = -Inf, Iterations = 59
       Some closed-loop poles are marginally stable (decay rate near 1e-07)
Final: Soft = 1.01, Hard = -Inf, Iterations = 66
       Some closed-loop poles are marginally stable (decay rate near 1e-07)
Final: Soft = 1.01, Hard = -Inf, Iterations = 61
       Some closed-loop poles are marginally stable (decay rate near 1e-07)
Final: Soft = 1.01, Hard = -Inf, Iterations = 60
```

After finding a solution using `systune`, show how tuning goals are met in the tuned model `ST1`. Show the tracking, loop shape, and gain tuning goals separately. Dashed lines in the following figures represent tuning goals and solid lines are the result of the tuned controllers.

```
figure
viewGoal([TR1,TR2],ST1)
figure
viewGoal([LS1,LS2],ST1)
figure
viewGoal([MG1,MG2,MG3],ST1)
```


Requirement 1: Tracking error as a function of frequency
Relative error (abs)
10^0
Tracking error
Max error
10^{-2}
10^{-1}
10^0
10^1
10^2
10^3
10^4
Frequency (rad/s)
Requirement 2: Tracking error as a function of frequency
Relative error (abs)
10^0
Tracking error
Max error
10^1
10^2
10^3
10^4
10^5
Frequency (rad/s)


Requirement 1: Minimum and maximum loop gains (CrossTol = 0.1)
Singular Values (dB)
40
20
0
-20
-40
10^0
10^1
10^2
10^3
10^4
10^5
Frequency (rad/s)
S
T
Loop gain(s)
Target loop shape
Requirement 2: Minimum and maximum loop gains (CrossTol = 0.1)
Singular Values (dB)
40
20
0
-20
-40
10^{-1}
10^0
10^1
10^2
10^3
10^4
10^5
Frequency (rad/s)
S
T
Loop gain(s)
Target loop shape



After verifying tuning goals, extract controller parameters from the tuned model `ST1`. Use tuned PI controller parameters to update the workspace parameters for the PI controller blocks.

```
Cd = getBlockValue(ST1,'Cd');
Cq = getBlockValue(ST1,'Cq');
Cspeed = getBlockValue(ST1,'Cspeed');
```

The d-axis current PI controller has tuned gains:

```
paramCurrentControlPD = Cd.Kp
paramCurrentControlID = Cd.Ki
```

```
paramCurrentControlPD =

    2.8898


paramCurrentControlID =

   2.6370e+03
```

The q-axis current PI controller has tuned gains:

```
paramCurrentControlPQ = Cq.Kp
paramCurrentControlIQ = Cq.Ki
```

```
paramCurrentControlPQ =

    1.4964


paramCurrentControlIQ =

  679.7979
```

The speed PI controller has tuned gains:

```
paramVelocityControlTuneP = Cspeed.Kp
paramVelocityControlTuneI = Cspeed.Ki
```

```
paramVelocityControlTuneP =

    0.3713


paramVelocityControlTuneI =

    0.0179
```

After tuning all three controllers together using `systune`, the controller gains are significantly different from the original values. The PID controller in the speed control loop has a different sample time, which is `0.001` second. The tuned result uses a different sample time of `4e-6` second but the controller gains are the same. To make sure controller performances are identical with different sample times, the discrete integrator format of the PID controllers is 'Trapezoidal' in this example.

Validate Tuned Controller

Examine the performances using the tuned controller gains. First, initialize the model to its zero initial conditions using `ctrlIniValues`. Connect the PID controller blocks by setting switch signals in the `switchIniValue` and set proper initial conditions for the PMSM plant model.

```
switchIniValue.openLoopQ = 0;
switchIniValue.openLoopD = 0;
ctrlIniValues.currentDIC = 0;
ctrlIniValues.voltageD = 0;
ctrlIniValues.currentQIC = 0;
ctrlIniValues.voltageQ = 0;
ctrlIniValues.speedIC = 0;
ctrlIniValues.speedCurrent = 0;
pmsm.RotorVelocityInit = 0;
set_param(mdl,'LoadInitialState','off')
```

Configure the model to use a one-sided speed command signal and simulate the model. Show the speed response of the model to the one-sided speed command that rises from `0` rad/s to `150` rad/s at `0.05` s, and then to `200` rad/s at `0.8` s. Save the simulation result to `logsout_tuned_oneside` in the MAT-file, `SystunedSpeed.mat`.

```
signalbuilder(SignalBuilderPath, 'activegroup', 2);
sim(mdl);
```

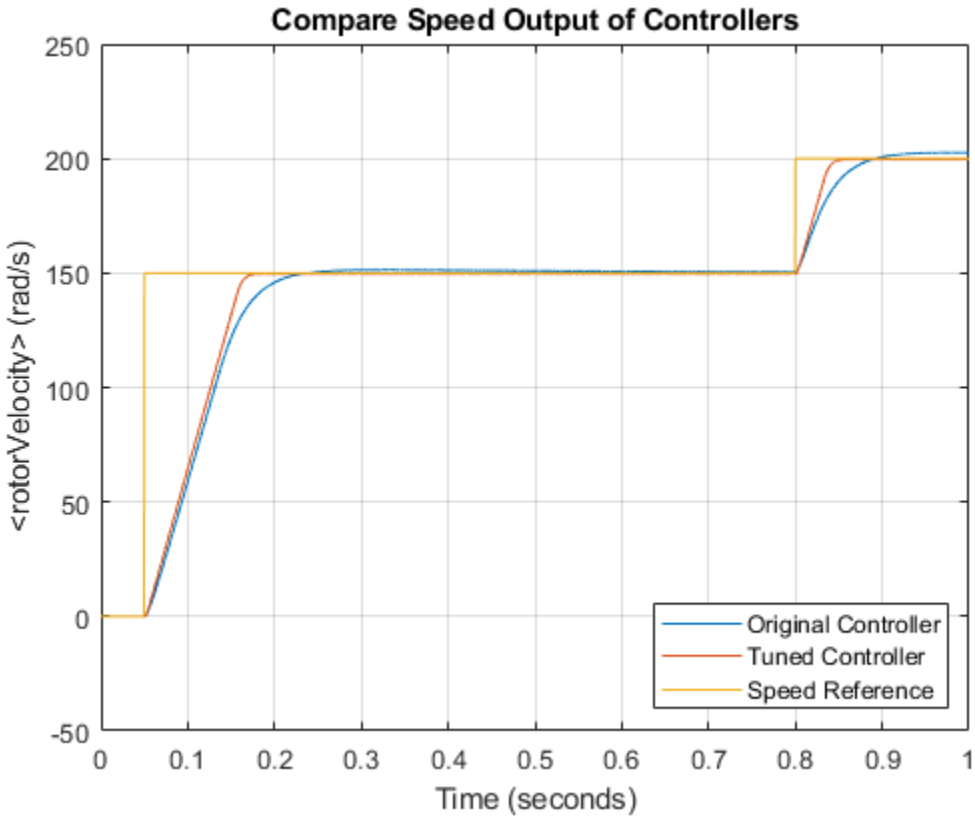
```
logsout_tuned_oneside = logsout;
save('SystunedSpeed','logsout_tuned_oneside','-append')
```

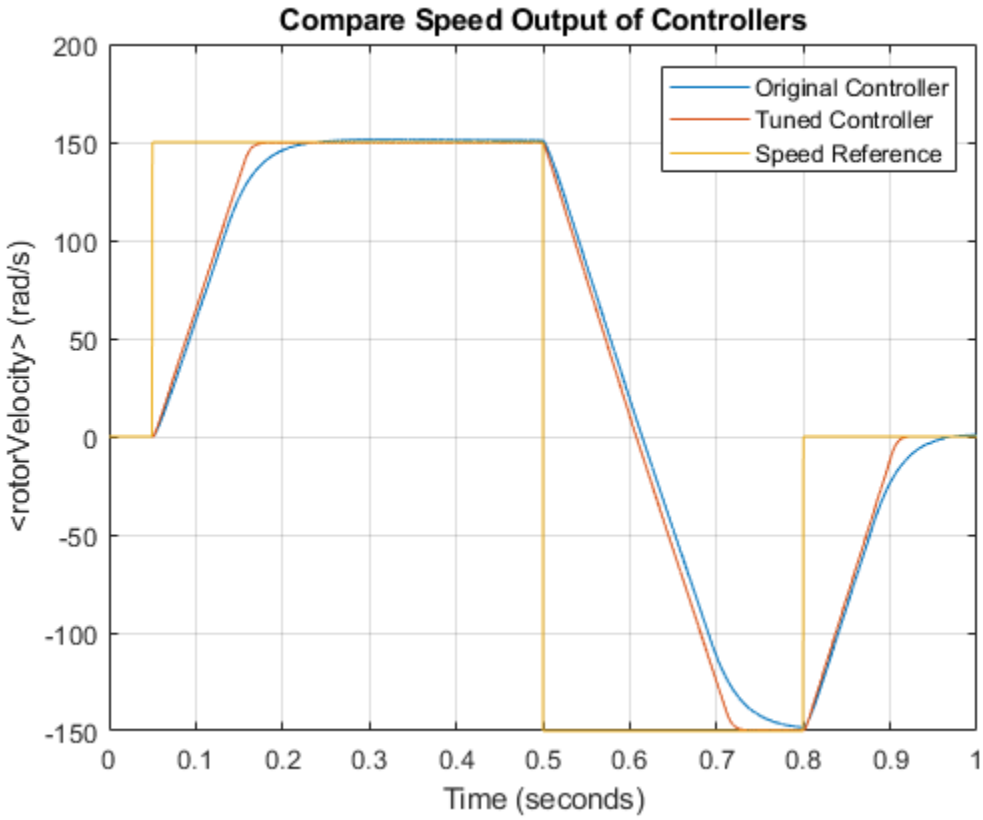
Configure the model to use a two-sided speed command signal and simulate the model. Show the speed response of the model to the two-sided speed command that rises from `0` rad/s to `150` rad/s at `0.05` s, reverses direction at `0.5` s and then back to `0` rad/s at `0.8` s. Save the simulation result to `logsout_tuned_twoside` in the MAT-file, `SystunedSpeed.mat`.

```
signalbuilder(SignalBuilderPath, 'activegroup', 3);
sim(mdl);
logsout_tuned_twoside = logsout;
save('SystunedSpeed','logsout_tuned_twoside','-append')
```

Compare the motor speed responses between the existing controller gains and the tuned result. The speed responses are shown side-by-side over the one-second simulation. The speed response follows more closely to the step command. The steady-state error also decreases after the PI controllers are tuned with `systune`.

```
scdfocmotorSystunePlotSpeed
```





After tuning the controllers, the motor response improves with faster transient response and smaller steady-state error under both types of speed commands.

```
bdclose(mdl)
```

See Also

`systune`

More About

- “Tune Field-Oriented Controllers Using Closed-Loop PID Autotuner Block” on page 8-47

Frequency Response Based PID Tuner

Frequency Response Based PID Tuner tunes the gains of PID controller based on a simulation experiment that estimates the frequency-response of the plant in your model. It is particularly useful for tuning or retuning the gains of a PID Controller for a plant that you cannot linearize.

The frequency-response based PID tuning process begins with an estimation experiment that breaks the loop at the plant input, and perturbs the plant with sine and step signals. The tuner then uses the resulting data to estimate the plant frequency response. Finally, it uses the estimated frequency response to compute PID gains to balance performance and robustness.

Use the settings in this dialog box to specify parameters for the frequency-response estimation experiment and the goals for PID tuning. Then, click **Tune** to run the experiment and tune the PID gains.

For more information about how Frequency Response Based PID Tuner works, see “Frequency-Response Based Tuning” on page 7-37.

Note While the estimation experiment is running, the tuner replaces the PID Controller block in your model with an unnamed subsystem. When the estimation experiment is completed or canceled, the tuner restores the PID Controller block. This process might result in some displacement of signal wires on the model canvas.

Experiment Settings

Use the settings in this section to specify parameters of the estimation experiment.

Number of simulations

Specify whether to perform two simulations to remove the effects of disturbances in the model.

- **2 simulations (remove disturbances)** — Select when your model includes disturbances that have a large enough effect on the plant response to interfere with the estimation experiment. In this case, the tuner performs two simulations:
  - A simulation without any perturbations, to generate a baseline plant response, including plant input and output values at the nominal operating point.
  - A simulation with perturbations on the plant input.

  The tuner uses the difference between these two simulations to compute the estimated frequency response that it uses for tuning. This process removes the effect of disturbances in the model, which would otherwise distort the estimated frequency response.
- **1 simulation** — Select when your model does not include disturbances that affect the frequency-response estimation. Selecting this option skips the baseline simulation, cutting the overall PID tuning time in half.

Default: 2 simulations (remove disturbances)

Plant information

Specify whether your plant is asymptotically stable or has an integrator.

- **Asymptotically stable** — Select when your plant is stable and has no integrators. When this option is selected, the estimation experiment includes an estimation of the plant DC gain. The Frequency Response Based PID Tuner performs this estimation by injecting a step signal into the plant.
- **Has single integrator** — Select when your plant contains one integrator. When this option is selected, the experiment does not include a step perturbation and DC-gain estimate.

Caution Do not use the Frequency Response Based PID Tuner with an unstable plant or a plant containing multiple integrators.

Default: Asymptotically stable

Start time (t0)

Specify the experiment start time. Start the experiment when the plant is at the desired equilibrium operating point. For instance, if you know that your simulation must run to 10 s for transient effects to decay, specify a start time of 10.

Default: 0

Duration (tspan)

Specify how long to let the frequency-response estimation experiment run. Let the experiment run long enough for the frequency-response estimation algorithm to collect sufficient data for a good estimate at all frequencies it probes. A conservative estimate for the experiment duration is $100/\omega_c$, where ω_c is the target bandwidth for tuning that you specify with the **Target bandwidth (rad/sec)** parameter.

The Frequency Response Based PID Tuner computes tuned PID gains when the experiment ends.

Default: 100

Sine amplitudes (Asin)

During the tuning experiment, the Frequency Response Based PID Tuner injects a sinusoidal signal into the plant at four frequencies, $[1/3,1,3,10]\omega_c$, where ω_c is the target bandwidth for tuning. Use **Sine Amplitudes (Asin)** to specify the amplitude of these injected signals. Specify a:

- Scalar value to inject the same amplitude at each frequency.
- Vector of length 4 to specify a different amplitude at each of $[1/3,1,3,10]\omega_c$, respectively.

In a typical plant with typical target bandwidth, the magnitudes of the plant responses at the experiment frequencies do not vary widely. In such cases, you can use a scalar value to apply the same magnitude perturbation at all frequencies. However, if you know that the response decays sharply over the frequency range, consider decreasing the amplitude of the lower-frequency inputs and increasing the amplitude of the higher-frequency inputs. It is numerically better for the estimation experiment when all the plant responses have comparable magnitudes.

The perturbation amplitudes must be:

- Large enough that the perturbation overcomes any deadband in the plant actuator and generates a response above the noise level

- Small enough to keep the plant running within the approximately linear region near the nominal operating point, and to avoid saturating the plant input or output

In the experiment, the sinusoidal signals are superimposed (with the step perturbation, if any, in the case of open-loop tuning). Thus, the perturbation can be at least as large as the sum of all amplitudes. Therefore, to obtain appropriate values for the amplitudes, consider:

- Actuator limits. Make sure that the largest possible perturbation is within the range of your plant actuator. Saturating the actuator can introduce errors into the estimated frequency response.
- How much the plant response changes in response to a given actuator input at the nominal operating point for tuning. For instance, suppose that you are tuning a PID controller used in engine-speed control. You have determined that at frequencies around the target bandwidth, a 1° change in throttle angle causes a change of about 200 rpm in the engine speed. Suppose further that to preserve linear performance the speed must not deviate by more than 100 rpm from the nominal operating point. In this case, choose amplitudes to ensure that the perturbation signal is no greater than 0.5 (assuming that value is within actuator limits).

Default: 1

Step amplitude (Astep)

If **Plant is asymptotically stable** is selected, the Frequency Response Based PID Tuner estimates the DC gain by injecting a step signal into the plant. Use this parameter to set the amplitude of the signal. The considerations for choosing a step amplitude are the same as the considerations for specifying **Sine amplitudes (Asin)**.

Default: 1

Design Specifications

Use the settings in this section to specify tuning goals.

Target bandwidth (rad/sec)

Specify the target value for the 0-dB gain crossover frequency of the tuned open-loop response CP, where P is the plant response, and C is the controller response. This crossover frequency roughly sets the control bandwidth. For a desired rise-time τ, a good guess for the target bandwidth is $2/\tau$.

When tuning a discrete-time controller with sample time T_s, the target bandwidth, ω_c, must satisfy $\omega_c T_s \leq 0.3$. This requirement ensures that the highest frequency in the estimation experiment, $10\omega_c$, is less than the Nyquist frequency. Because of this condition, the fastest rise time you can enforce for discrete-time tuning is about $1.67T_s$. If this rise time does not meet your design goals, consider reducing T_s.

Default: 1

Target phase margin (degrees)

Specify a target minimum phase margin for the tuned open-loop response at the crossover frequency. The target phase margin reflects desired robustness of the tuned system. Typically, choose a value in the range of about 45°– 60°. In general, higher phase margin improves overshoot, but can limit response speed. The default value, 60°, tends to balance performance and robustness, yielding about 5-10% overshoot, depending on the characteristics of your plant.

Default: 60

Automatically Update Block

When this option is selected, the Frequency Response Based PID Tuner automatically updates the gains in the PID Controller block when the experiment and tuning is complete. If you want to examine the tuning results before updating the block, clear this option. In that case, click **Update PID Block** to write the tuned gains to the block.

Tune and Cancel

Click **Tune** to initiate the frequency-response estimation experiment. While the estimation experiment is running, the tuner:

- Closes the open PID Controller block.
- Clears any previous tuning results displayed in the tuner dialog box.
- Replaces the PID Controller block in your model with an unnamed subsystem.

Note When the estimation experiment is completed or canceled, the tuner restores the PID Controller block. This process might result in some displacement of signal wires on the model canvas, and puts your Simulink model in a state with unsaved changes.

To abort the experiment before completion, click **Cancel**. When you do so, the tuner does not compute new gains.

Tuning Results

While the experiment is running, this section displays the progress of the estimation experiment. When the experiment and tuning is complete, the resulting PID gains are displayed. Also displayed are:

- **Estimated phase margin** — Estimated phase margin achieved by the tuned system, in degrees. The tuner calculates this value from the angle of $G(j\omega_c)C(j\omega_c)$, where G is the plant, C is the tuned controller, and ω_c is the crossover frequency (bandwidth). The estimated phase margin might differ from the target phase margin you specify in the **Target phase margin (degrees)** parameter. It is an indicator of the robustness and stability achieved by the tuned system.
  - Typically, the estimated phase margin is near the target phase margin. In general, the larger the value, the more robust is the tuned system, and the less overshoot there is.
  - A negative phase margin indicates that the closed-loop system might be unstable.
- **Nominal plant input** — Plant input at the nominal operating point, when the experiment begins.

For additional information about the experiment and tuning results, click **Export To MATLAB**. When you do so, the tuner creates a structure in the MATLAB workspace, `OnlinePIDTuningResult`, containing the following fields:

- `P`, `I`, `D`, `N` — Tuned PID gains. The structure contains whichever of these fields are necessary for the controller type of your PID Controller block. For instance, if you are tuning a PI controller, the structure contains `P` and `I`, but not `D` and `N`.

- `TargetBandwidth` — The value you specified in the **Target bandwidth (rad/sec)** parameter.
- `TargetPhaseMargin` — The value you specified in the **Target phase margin (degrees)** parameter.
- `EstimatedPhaseMargin` — Estimated phase margin achieved by the tuned system.
- `Controller` — The tuned PID controller, returned as a `pid` (for parallel form) or `pidstd` (for standard form) model object.
- `Plant` — The estimated plant, returned as an `frd` model object. This `frd` contains the response data obtained at the four frequencies [1/3, 1, 3, 10]ω_c.
- `PlantNominalInput` — The plant input at the nominal operating point when the experiment begins.
- `PlantDCGain` — The estimated DC gain of the system in absolute units, if **Plant is asymptotically stable** is selected during tuning.

See Also

More About

- “Frequency-Response Based Tuning” on page 7-37

PID Autotuning

- “When to Use PID Autotuning” on page 8-2
- “How PID Autotuning Works” on page 8-5
- “PID Autotuning for a Plant Modeled in Simulink” on page 8-7
- “PID Autotuning in Real Time” on page 8-13
- “Control Real-Time PID Autotuning in Simulink” on page 8-20
- “Tune PID Controller in Real Time Using Open-Loop PID Autotuner Block” on page 8-23
- “Tune PID Controller in Real Time Using Closed-Loop PID Autotuner Block” on page 8-29
- “BLDC Motor Speed Control with Cascade PI Controllers” on page 8-36
- “Tune Field-Oriented Controllers Using Closed-Loop PID Autotuner Block” on page 8-47
- “Tune Field-Oriented Controllers for an Asynchronous Machine Using Closed-Loop PID Autotuner Block” on page 8-54
- “Tune Field-Oriented Controllers for a PMSM Using Closed-Loop PID Autotuner Block” on page 8-62
- “Design PID Controllers for Three-Phase Rectifier using Closed-Loop PID Autotuner Block” on page 8-69
- “PID Autotuning for UAV Quadcopter” on page 8-75
- “Tune Gain-Scheduled Controller Using Closed-Loop PID Autotuner Block” on page 8-88

When to Use PID Autotuning

The PID autotuner blocks in Simulink Control Design let you tune a PID controller without a parametric plant model or an initial controller design. If you have a code-generation product such as Simulink Coder, you can generate code that implements the tuning algorithm on hardware. Deploying the algorithm to hardware lets you tune a controller for a physical plant, with or without using Simulink to manage the tuning process.

To achieve model-free tuning, use the Closed-Loop PID Autotuner or Open-Loop PID Autotuner blocks. These blocks perform a frequency-response estimation experiment that injects signals into the plant and measures the plant output with the feedback loop closed or open, respectively. The blocks use the resulting estimated frequency response to tune PID gains for the plant.

PID autotuning works with any asymptotically stable or integrating SISO plant, whether low-order or high-order, with or without time delay, and with or without direct feedthrough. It can tune any type of PID controller. You trigger the tuning process via an input to the block, so you can tune your controller at any time.

PID Autotuning for a Physical Plant

Embedded PID autotuning is a useful option when you have a PID-controlled system and a test bed or control environment to operate in. In this case, you can deploy an autotuner block to your hardware and automatically tune the gains of the PID controller in your system.

In practice, you can manage the PID autotuning process in several ways, including:

- Deploy the autotuning algorithm as a standalone embedded module and manage the tuning process in your own software and hardware environment. For details, see “PID Autotuning in Real Time” on page 8-13.
- Initiate, monitor, and analyze the autotuning process via Simulink. For details, see “Control Real-Time PID Autotuning in Simulink” on page 8-20.

PID Autotuning for a Plant Model in Simulink

If you have a plant model in Simulink, you can use PID autotuning to:

- Obtain an initial PID design for your plant, which you can refine by tuning against the physical plant.
- Preview plant response and adjust the settings for PID autotuning before tuning the controller in real time. Doing so helps ensure that real-time tuning does not drive your system out of the desirable operating range.

For more information, see “PID Autotuning for a Plant Modeled in Simulink” on page 8-7.

Closed-Loop vs. Open-Loop PID Autotuning

The PID autotuning tools let you tune:

- In a closed-loop configuration, with your plant under control of an existing PID controller (Closed-Loop PID Autotuner block).

- In an open-loop configuration (Open-Loop PID Autotuner block). With open-loop autotuning, if the plant is in a feedback loop, the autotuner opens the loop for the duration of the tuning process.

In general, if you do not have an initial PID design, start with open-loop autotuning, and switch to closed-loop autotuning for retuning or refinement. If you have an initial PID design for your plant, use closed-loop tuning, which is safer for your plant. With closed-loop autotuning, the controller remains in the loop to:

- Reject unexpected plant disturbances to maintain safe operation of the plant during the estimation experiment.
- Reduce the risk that the perturbations used for the experiment drive the plant away from the desired operating point.

Additional advantages of the closed-loop autotuning approach include:

- Closed-loop tuning works with multiple-integrator plants. In contrast, you cannot use open-loop autotuning for multiple-integrator plants. Even single-integrator plants risk drifting away from the desired operating point during open-loop tuning.
- Because the feedback loop remains closed, there is no concern about controller saturation during the tuning process. In contrast, with open-loop autotuning, a controller with integral action can saturate while the loop is open. Such saturation can create a jump at the plant input when the tuning process ends. With open-loop tuning, you must take additional steps to ensure that the controller continues to track autotuner block output during tuning. (See, for instance, “PID Autotuning for a Plant Modeled in Simulink” on page 8-7.)

If safe operation of your plant is not a practical concern (such as when tuning against a plant model in Simulink), open-loop autotuning has these advantages:

- Open-loop tuning can result in more accurate frequency-response estimation and tuning. In closed-loop tuning, the controller suppresses injected perturbations, which can result in less accurate frequency-response estimation and poorer tuning results.
- Open-loop tuning is faster. Closed-loop tuning uses a lower-frequency perturbation signal, which makes the process about three times longer.
- The memory footprint of the deployed algorithm is slightly smaller.

Caution

- Do not use either closed-loop or open-loop PID autotuning with an unstable plant.
- Do not use open-loop PID autotuning with a plant that has more than one integrator. You can use closed-loop PID autotuning with a multiple-integrator plant.

To get started with either type of PID autotuner, see “How PID Autotuning Works” on page 8-5.

When Not to Use PID Autotuning

PID Autotuning is not suitable for unstable plants. The perturbations applied in open-loop tuning can drive an unstable plant to operating conditions that are unsafe for the plant. Although closed-loop autotuning does not have that risk, it does not yield meaningful tuning results for unstable plants.

PID autotuning does not work well when there are large disturbances in the plant during the estimation experiment. Disturbances distort the plant response to the perturbation signals, yielding poor estimation results.

See Also

Closed-Loop PID Autotuner | Open-Loop PID Autotuner

More About

- “How PID Autotuning Works” on page 8-5

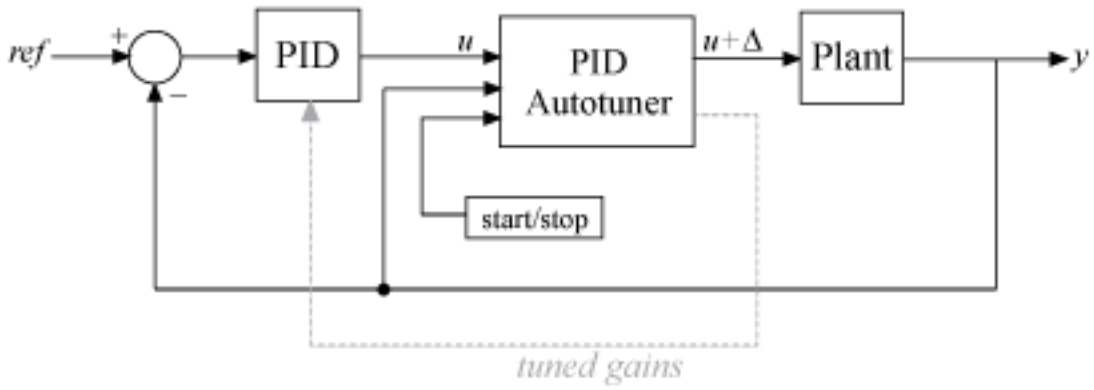
How PID Autotuning Works

To use PID autotuning, configure and deploy one of the PID autotuner blocks, Closed-Loop PID Autotuner or Open-Loop PID Autotuner.

Autotuning Process

The PID autotuner blocks work by performing a frequency-response estimation experiment. The blocks inject test signals into your plant and tune PID gains based on an estimated frequency response.

The following schematic diagram illustrates generally how a PID autotuner block fits into a control system.



Until the autotuning process begins, the autotuner block relays the control signal directly from **u** to the plant input at **u+Δu**. In that state, the module has no effect on the performance of your system.

When the autotuning process begins, the block injects a test signal at `u out` to collect plant input-output data and estimate frequency response in real time.

- If you use the Open-Loop PID Autotuner block, the block opens the feedback loop between **u** and **u +Δu** for the duration of the estimation experiment. It injects into **u+Δu** a superposition of sinusoidal signals at frequencies [1/3, 1, 3, 10]ω_c, where ω_c is your specified target bandwidth for tuning. For nonintegrating plants, the block can also inject a step signal to estimate the plant DC gain. All test signals are injected on top of the nominal plant input, which is the value of the signal at **u** when the experiment begins.
- If you use the Closed-Loop PID Autotuner block, the plant remains under control of the PID controller with its current gains during the experiment. Closed-loop tuning uses sinusoidal test signals at the frequencies [1/10,1/3, 1, 3, 10]ω_c.

When the experiment ends, the block uses the estimated frequency response to compute PID gains. The tuning algorithm aims to balance performance and robustness while achieving the control bandwidth and phase margin that you specify. You can configure logic to transfer the tuned gains from the block to your PID controller, allowing you to validate closed-loop performance in real time.

Workflow for PID Autotuning

The following steps provide a general overview of the workflow for PID autotuning.

1. Incorporate a PID autotuner block into your system, as shown in the schematic diagram.
2. Configure the start/stop signal that controls when the tuning experiment begins and ends. You can use this signal to initiate the PID autotuning process at any time. When you stop the experiment, the block returns tuned PID gains.
3. Specify controller parameters such as controller type and the target bandwidth for tuning.
4. Configure experiment parameters such as the amplitudes of the perturbations injected during the frequency-response experiment.
5. Start the autotuning process using the start/stop signal, and allow it to run long enough to complete the frequency-response estimation experiment.
6. Stop the autotuning process. When the experiment stops, the autotuner computes and returns tuned PID gains.
7. Transfer the tuned gains from the block to your PID controller. You can then validate the performance of the tuned controller in Simulink or in real time.

For detailed information on performing each of these steps, see:

- “PID Autotuning for a Plant Modeled in Simulink” on page 8-7
- “PID Autotuning in Real Time” on page 8-13

See Also

Closed-Loop PID Autotuner | Open-Loop PID Autotuner

More About

- “When to Use PID Autotuning” on page 8-2

PID Autotuning for a Plant Modeled in Simulink

To use PID autotuning for a plant modeled in Simulink, you incorporate a PID autotuner block into the model. You can control the autotuning process while the model is running. When tuning is complete you can validate tuned controller parameters against the simulated plant. Using PID autotuning this way can be useful for generating an initial PID design that you later refine with real-time autotuning.

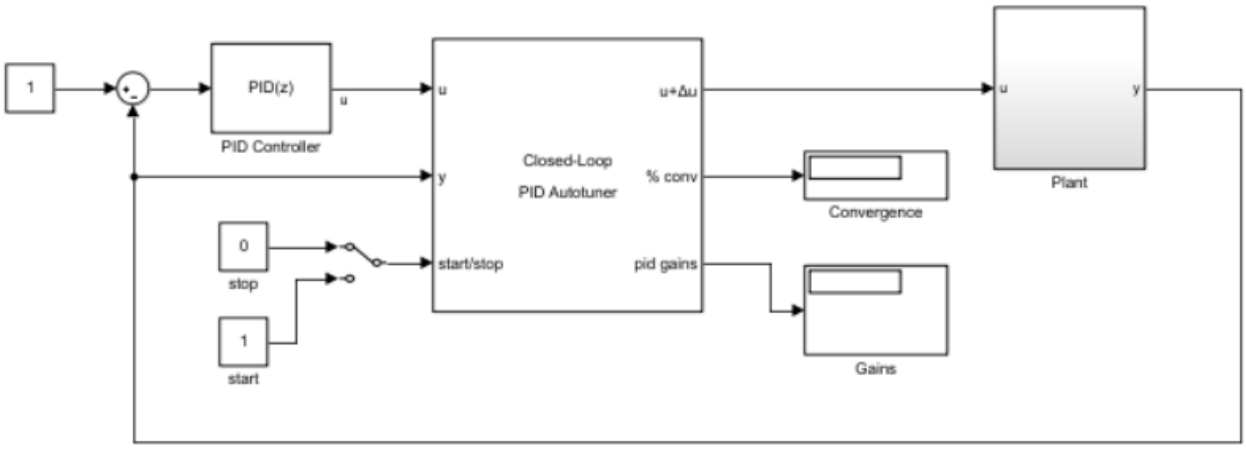
Workflow for Autotuning in Simulink

The following steps provide a general overview of the workflow for PID autotuning in Simulink using the Closed-Loop PID Autotuner or Open-Loop PID Autotuner blocks.

1. Incorporate a PID autotuner block on page 8-7 into your model between the PID controller and the plant.
2. Configure the start/stop signal on page 8-9 that controls when the tuning experiment begins and ends.
3. Specify controller parameters on page 8-9 such as controller type and the target bandwidth for tuning.
4. Configure experiment parameters on page 8-10 such as the amplitudes of the perturbations injected during the frequency-response experiment.
5. Run the model and initiate tuning on page 8-11. Use the start/stop signal to initiate the PID autotuning process. When you start the process, the autotuner block injects test signals and measures the response of the plant.
6. Stop the experiment on page 8-11 with the start/stop signal. When the experiment stops, the autotuner block computes and returns tuned PID gains. You can examine the tuned gains for reasonableness.
7. Transfer the tuned gains on page 8-11 from the autotuner block to your PID controller. You can then validate the performance of the tuned controller in Simulink.

Step 1. Incorporate Autotuner into Model

The following illustration shows one way to incorporate a Closed-Loop PID Autotuner block in between your PID controller and your plant.



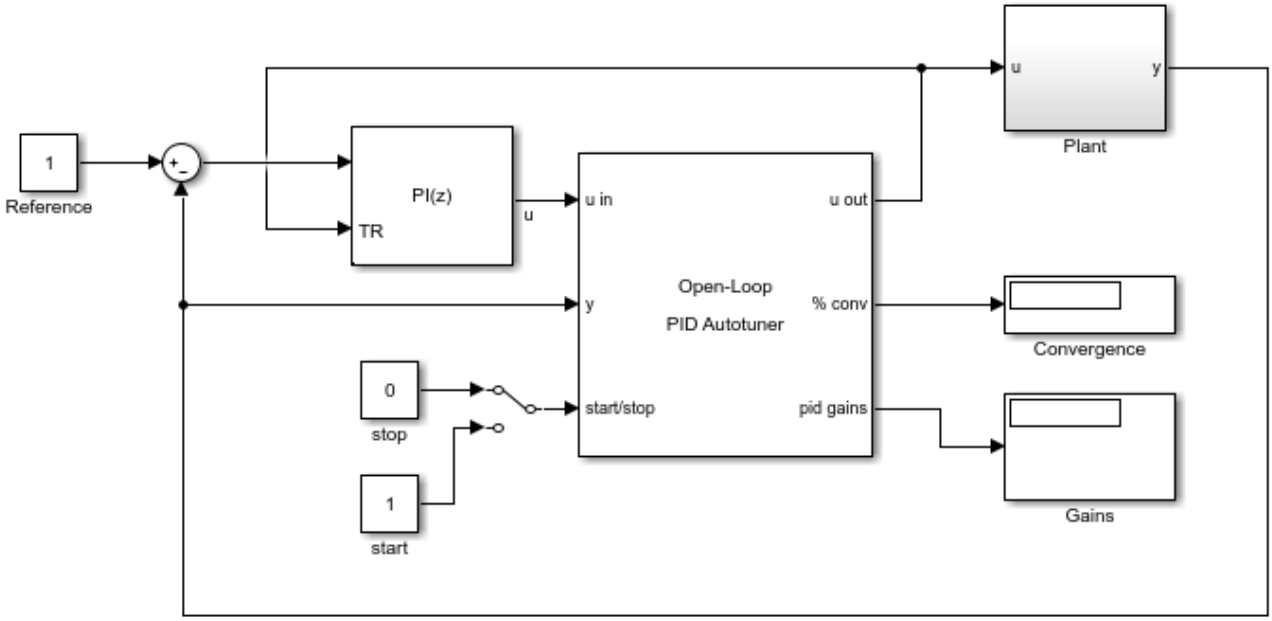
The control signal `u` from the PID controller feeds into the **u** port of the autotuner block. The **u+Δu** port feeds into the plant input. Before you begin the autotuning process, the autotuner block feeds the PID control signal directly from **u** to **u+Δu** and the plant input. In that state, the autotuner block has no effect on plant or controller behavior. During the autotuning process, the block injects test signals at the plant input and measures the response at `y`.

The `start/stop` signal controls when the autotuning process begins and ends (see “Step 2. Configure Start/Stop Signal” on page 8-9). When the experiment ends, the block calculates PID gains and returns them at the `pid gains` port.

For a more detailed example of a Simulink model configured for closed-loop PID autotuning, see “Tune PID Controller in Real Time Using Closed-Loop PID Autotuner Block” on page 8-29.

Bumpless Transfer for Open-Loop Tuning

The Open-Loop PID Autotuner block opens the loop between **u** and **u+Δu** during the estimation experiment. If your controller includes integral action, you can use signal tracking to avoid integrator windup while the loop is open. Signal tracking enables the PID controller to continue to track the real plant input while it is out of the loop. Without it, your system can experience a bump when the control loop is closed at the end of the tuning process. In system of the following illustration, the PID controller is a Simulink PID Controller block with the **Enable tracking mode** parameter on. The plant input feeds into the tracking input of the controller block.



For a more detailed example of a Simulink model configured for open-loop PID autotuning, see “Tune PID Controller in Real Time Using Open-Loop PID Autotuner Block” on page 8-23.

Step 2. Configure Start/Stop Signal

To start and stop the autotuning process, use a signal at the `start/stop` port. When the experiment is not running, the block passes signals unchanged from **u** to **u+Δu**. In this state, the block has no impact on plant or controller behavior.

The frequency-response estimation experiment begins and ends when the block receives a rising or falling signal at the `start/stop` port, respectively. In the systems illustrated in “Step 1. Incorporate Autotuner into Model” on page 8-7, the start/stop signal is a simple switch. While the model is running, you can use the switch to begin and end the experiment. When you end the experiment, the algorithm generates the tuned PID gains and the block returns them at the `pid gains` port.

As an alternative to a manual switch, you can configure the `start/stop` signal to begin and end the experiment automatically at particular simulation times. For example, you can use the sum of two Step blocks: Configure one Step block to step from 0 to 1 at the experiment start time, and a second Step block to step from 1 to 0 at the end time. Feed the sum of the two signals into the `start/stop` port of the PID autotuner block.

You can configure any other logic appropriate for your application to control the start and stop times of the experiment. For more information about when to start and stop the experiment, see “Step 5. Run Model and Initiate Tuning Experiment” on page 8-11.

Step 3. Specify Controller Parameters and Tuning Goals

In the PID autotuner block, specify the configuration of the PID controller you are tuning, using the following block parameters:

- **Type**
- **Form**
- **Time Domain**
- **Controller sample time (sec)**
- **Integrator method**
- **Filter method**

Then, specify the target bandwidth and phase margin for tuning with the **Target bandwidth (rad/sec)** and **Target phase margin (degrees)** parameters, respectively.

The target bandwidth, specified in rad/sec, is the target value for the 0-dB gain crossover frequency of the tuned open-loop response CP, where P is the plant response, and C is the controller response. This crossover frequency roughly sets the control bandwidth. For a desired rise-time τ seconds, a good guess for the target bandwidth is $2/\tau$ rad/sec.

The target phase margin reflects your desired robustness of the tuned system. Typically, choose a value in the range of about 45°– 60°. In general, higher phase margin improves overshoot, but can limit response speed. The default value, 60°, tends to balance performance and robustness, yielding about 5-10% overshoot, depending on the characteristics of your plant.

For more information about setting these parameters, see the Closed-Loop PID Autotuner or Open-Loop PID Autotuner block reference pages.

Step 4. Set Experiment Parameters

The frequency-response estimation experiment injects sinusoidal signals at frequencies around the target bandwidth ω_c:

- $[1/3, 1, 3, 10]\omega_c$ for the Open-Loop PID Autotuner block
- $[1/10, 1/3, 1, 3, 10]\omega_c$ for the Closed-Loop PID Autotuner block

Use the **Sine Amplitudes** parameter of the blocks to specify the amplitudes of these signals.

If your plant is asymptotically stable, the Open-Loop PID Autotuner block can estimate the plant DC gain with a step perturbation. Specify the amplitude of this perturbation with the **Step Amplitude** parameter. If your plant has a single integrator, clear the **Estimate DC gain with step signal** parameter.

Caution

- Do not use either closed-loop or open-loop PID autotuning with an unstable plant.
- Do not use open-loop PID autotuning with a plant that has more than one integrator. You can use closed-loop PID autotuning with a multiple-integrator plant.

All the perturbation amplitudes must be:

- Large enough that the perturbation overcomes any deadband in the plant actuator and generates a response above the noise level.

- Small enough to keep the plant running within the approximately linear region near the nominal operating point, and to avoid saturating the plant input or output.

For more information about setting the experiment parameters, see the Closed-Loop PID Autotuner and Open-Loop PID Autotuner block reference pages.

Step 5. Run Model and Initiate Tuning Experiment

After you have configured all the parameters for tuning, run the model.

- If you have configured a manual `start/stop` signal, begin the experiment when your plant has reached steady-state.
- If you have configured the `start/stop` signal to begin and end the tuning process at specific times, allow the simulation to run long enough to begin the experiment.

Step 6. Stop Experiment and Examine Tuned Gains

The frequency-response estimation experiment ends when the `start/stop` signal falls.

- If you have configured a manual `start/stop` signal, end the experiment when the signal at the `% conv` output stabilizes near 100%.
- If you have configured the `start/stop` signal to begin and end the tuning process at specific times, allow the simulation to run through the end of the experiment.

In either case, a conservative estimate for the experiment time is $200/\omega_c$ for closed-loop tuning or $100/\omega_c$ for open-loop tuning, where ω_c is your target bandwidth.

When you stop experiment, the block computes new PID gains based on the estimated frequency response of the system and your specified tuning goals. Examine them for reasonableness. For instance, if you have an initial PID controller, you might expect the tuned gains to be roughly the same magnitude as the gains of the initial design. There are several ways to see the tuned gains:

- View the output of the `pid gains` port of the autotuner block. One way to view this output is to connect the output to a Simulink Display block.
- In the block, in the **Block** tab, click **Export to MATLAB**. The block creates a structure in the MATLAB workspace, `OnlinePIDTuningResult`. For more information about the contents of this structure, see the Closed-Loop PID Autotuner or Open-Loop PID Autotuner block reference pages.

Step 7. Update PID Controller with Tuned Gains

The autotuner block can write tuned controller parameters directly to the PID controller block, if your PID controller is either:

- A Simulink PID Controller block.
- A custom PID controller for which the following conditions are both true:
  - The custom controller is a masked subsystem.
  - The PID gains are mask parameters named `P`, `I`, `D`, and `N`. (You do not need to use all four parameters. For example, if you use a custom PI controller, then you only need mask parameters `P` and `I`.)

To configure the autotuner block to write tuned gains to your controller, designate the controller as the associated PID block in the PID autotuner block parameters. (For more information, see the see the Closed-Loop PID Autotuner or Open-Loop PID Autotuner block reference pages.) Then, update your controller by clicking **Update PID Block**. You can update the PID gains while the simulation is running. Doing so is useful for immediately validating tuned PID gains.

Note At any time during simulation, you can change tuning or experiment parameters, start the experiment again, and push the new tuned gains to the PID block. You can then observe the behavior of your plant as simulation continues with the new gains.

Manual Update of PID Gains

If your custom PID controller does not satisfy the conditions for direct update, you must transfer the tuned gains to your controller some other way, such as manually or with your own logic.

When you examine these gains and transfer them to your own controller, be aware of the meaning of these gains in the PID autotuner blocks. In discrete time, the blocks assume the following PID controller transfer function:

$$C = P + IF_i(z) + D\left[\frac{N}{1 + NF_d(z)}\right],$$

in parallel form, or in ideal form,

$$C = P\left[1 + IF_i(z) + D\left(\frac{N}{1 + NF_d(z)}\right)\right].$$

$F_i(z)$ and $F_d(z)$ depend on the values you specify for the **Integrator method** and **Filter method** formulas, respectively. For more details, see the Closed-Loop PID Autotuner or Open-Loop PID Autotuner block reference pages.

See Also

Closed-Loop PID Autotuner | Open-Loop PID Autotuner

More About

- “When to Use PID Autotuning” on page 8-2
- “PID Autotuning in Real Time” on page 8-13
- “Control Real-Time PID Autotuning in Simulink” on page 8-20

PID Autotuning in Real Time

To use the PID autotuning algorithm in a standalone application for real-time tuning against your physical plant, you must deploy the PID autotuner block into your own system. To do so, you create a Simulink model for deployment. You can configure this model with the experiment and tuning parameters. Or, you can configure it to supply such parameters externally from elsewhere in your system. Once deployed to your own system, the autotuner model injects signals into your plant and receives the plant response, without using Simulink to control the tuning process. Deploying the PID autotuning algorithm requires a code-generation product such as Simulink Coder.

As an alternative, you can tune in real time against your physical plant while using Simulink to control the experiment. For more information, see “Control Real-Time PID Autotuning in Simulink” on page 8-20.

Workflow

In overview, the workflow for deploying a PID autotuning algorithm for real-time tuning is:

1. Create a Simulink model on page 8-13 for deploying a PID autotuner block into your system.
2. Configure the start/stop signal on page 8-16 that controls when the tuning experiment begins and ends. After deployment, you can use this signal to initiate the PID autotuning process at any time.
3. Specify controller parameters on page 8-16 such as controller type and the target bandwidth for tuning.
4. Configure experiment parameters on page 8-17 such as the amplitudes of the perturbations injected during the frequency-response experiment.
5. Deploy the model to your system, and initiate the autotuning process on page 8-18 against your physical plant. You can validate closed-loop performance in real time.

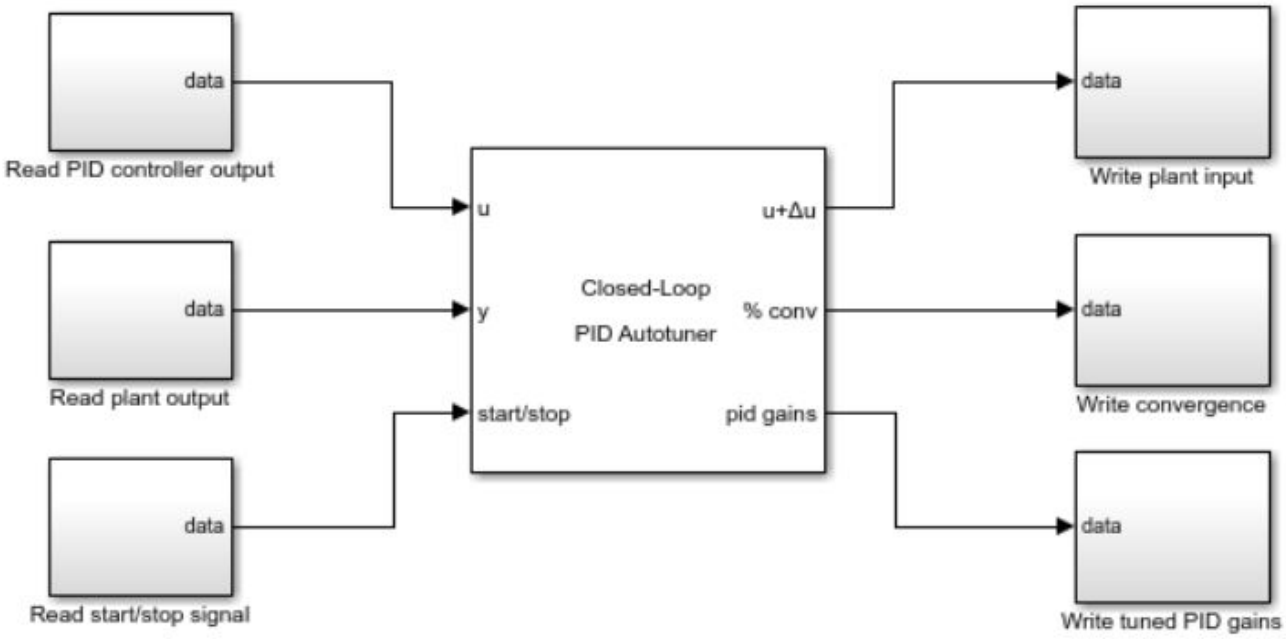
In practice, for real-time tuning, you might want to specify some parameters at run time, such as the target bandwidth or perturbation amplitudes. For information about specifying parameters in your deployed application, see “Access Autotuning Parameters After Deployment” on page 8-18.

Step 1. Create Deployable Simulink Model with PID Autotuner Block

Using a PID autotuner block for real-time tuning requires creating a Simulink model for deployment. There are several ways to do so.

Deployable Module with Autotuner Only

In the most basic form, a model for deploying real-time PID autotuning resembles the following illustration, using either the Closed-Loop PID Autotuner or the Open-Loop PID Autotuner block. An advantage of this approach is that it lets you switch between and tune different PID controllers at run time.



Here, the blocks connected to the inputs and outputs of the PID autotuner block represent hardware interfaces that read or write real-time data for your system. For example, the `Read PID controller output` block can be an interface for receiving serial data, a UDP Receive block for receiving UDP packets, or an interface for receiving other signals via wireless network. Similarly the blocks for writing data, such as `Write plant input`, can be interfaces for serial, UDP, or other interfaces for writing data to hardware.

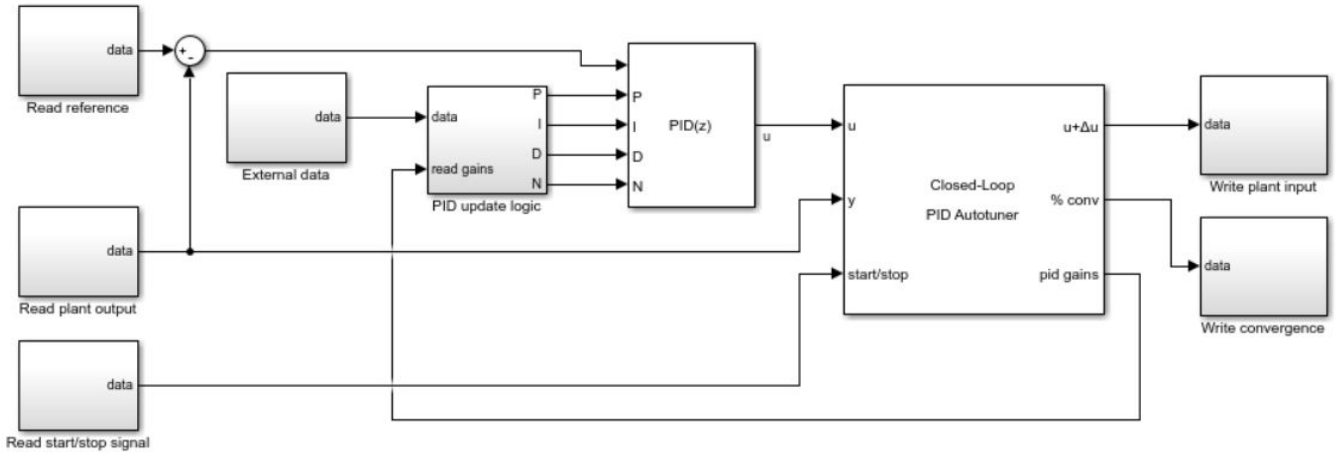
The default ports of the autotuner block are:

- `u` — Receives the control signal.
- `y` — Receives the plant output.
- `start/stop` — Receives the signal that begins and ends the tuning process.
- `u+Δu` — Outputs the signal to feed to the plant input. When the experiment is not running, `u+Δu` outputs the control signal as input at `u`. When the experiment is running the block and injects the test signals at `u+Δu`. For open-loop tuning only, the block breaks the loop between `u` and `u+Δu` for the duration of the experiment. When the experiment ends, the block restores the connection between `u` and `u+Δu`.
- `% conv` — Outputs a numeric indicator of the progress of the frequency-response estimation experiment.
- `pid gains` — Outputs the tuned PID gains when the tuning process stops.

In this configuration, the PID controller itself exists in another module of your system. When tuning is complete, you use your own logic to write the tuned PID gains from the `pid gains` port of the autotuning block to your PID controller.

Deployable Module with Controller

Alternatively, you can deploy a module that includes both the PID controller and the PID autotuning algorithm, such as shown in the following illustration. An advantage of this approach is that it facilitates retuning a specific controller in an individual system.



In this illustration, the PID controller is implemented as a Simulink PID Controller block. Because the PID gains of that block are tunable, you can configure your system to write the tuned gains to the deployed controller. Alternatively, you can also use your own custom PID controller subsystem in the model that you deploy.

You can implement any logic appropriate to your application to determine whether and how to update the PID controller with the tuned gains. In the illustrated system, the `PID update logic` subsystem represents such a module. The `External data` block represents whatever other information your logic requires to determine whether to update the controller.

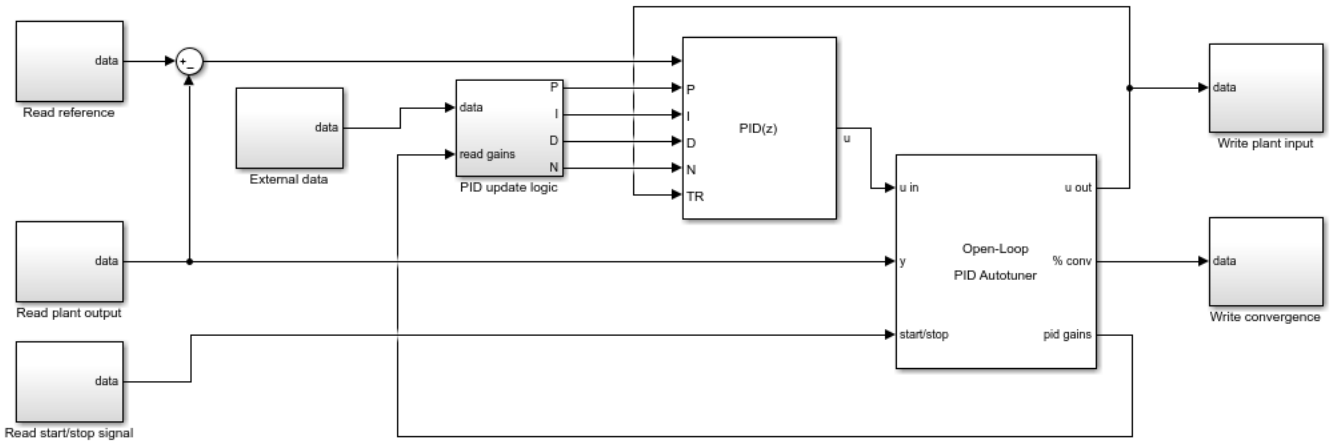
Note When you are using the Closed-Loop PID Autotuner block, feeding the `pid gains` outputs directly into the PID Controller gain inputs can introduce an algebraic loop that prevents code generation. To avoid this problem, you can introduce a state in your PID update logic that breaks the algebraic loop. For example, you can try one of the following approaches:

- Use a `Unit Delay` block to keep the controller output one time step ahead of the controller inputs.
- Use a Data Store Memory block, as illustrated in “Tune PID Controller in Real Time Using Closed-Loop PID Autotuner Block” on page 8-29.

Bumpless Transfer for Open-Loop Tuning

When you use the Open-Loop PID Autotuner, if your controller includes integrator action, consider implementing signal tracking to avoid integrator windup during the tuning experiment. Signal tracking enables the PID controller to continue to track the real plant input while it is out of the loop. Without it, your system can experience a bump when the control loop is closed at the end of the tuning process.

If your PID controller is a Simulink PID Controller block, you can use the **Enable tracking mode** parameter of the controller block to avoid this bump. The following diagram illustrates a module containing an Open-Loop PID Autotuner block and a PID Controller block with tracking mode configured. The plant input feeds into the tracking input of the controller block.



Step 2. Configure Start/Stop Signal

To start and stop the autotuning process, use a signal at the `start/stop` port. When the experiment is not running, the block passes signals unchanged from `u` to `u+Δu`. In this state, the block has no impact on plant or controller behavior.

The frequency-response estimation experiment begins and ends with a rising or falling signal at the `start/stop` port, respectively. Thus, after deployment, to begin the autotuning process, use a rising signal at the `start/stop` port. After an appropriate time, or after the `% conv` signal settles near 100, use a falling signal to end the experiment. When the experiment ends, the algorithm generates the tuned PID gains and returns them at the `pid gains` port. A conservative estimate for the experiment time is $200/\omega_c$ for closed-loop tuning or $100/\omega_c$ for open-loop tuning, where ω_c is your target bandwidth. For more detailed information about how to configure the start-stop signal, see the Closed-Loop PID Autotuner or Open-Loop PID Autotuner block reference pages.

Step 3. Set PID Tuning Parameters

To specify the configuration of the PID controller in your system, use the following parameters of the autotuner block:

- **Type**
- **Form**
- **Time Domain**
- **Controller sample time (sec)**
- **Integrator method**
- **Filter method**

For Closed-Loop PID Autotuner, you can also specify a sample time for tuning that is different from the **Controller sample time (sec)**. The PID gain tuning algorithm is computationally intensive, and

when you want to deploy the block to hardware and tune a controller with a fast sample time, some hardware might not complete PID gain calculation in a single time step. To reduce the hardware throughput requirements, enable **Tune at different sample time** parameter, and then specify a tuning sample time slower than the controller sample time using the **Tuning sample time (sec)** parameter.

Then, specify the target bandwidth and phase margin for tuning with the **Target bandwidth (rad/sec)** and **Target phase margin (degrees)** parameters, respectively.

The target bandwidth, specified in rad/sec, is the target value for the 0-dB gain crossover frequency of the tuned open-loop response CP, where P is the plant response, and C is the controller response. This crossover frequency roughly sets the control bandwidth. For a rise-time τ seconds, a good guess for the target bandwidth is $2/\tau$ rad/sec.

The target phase margin sets the robustness of the tuned system. Typically, choose a value in the range of about 45°– 60°. In general, higher phase margin improves overshoot, but can limit response speed. The default value, 60°, tends to balance performance and robustness, yielding about 5-10% overshoot, depending on the characteristics of your plant.

You can set most tuning parameters in your own application after deployment, instead of fixing them in the PID autotuner block before deployment. See “Access Autotuning Parameters After Deployment” on page 8-18.

For more details about the values to use for these parameters, see the Closed-Loop PID Autotuner or Open-Loop PID Autotuner block reference pages.

Step 4. Set Experiment Parameters

The frequency-response estimation experiment injects sinusoidal signals at frequencies around the target bandwidth ω_c:

- $[1/3, 1, 3, 10]\omega_c$ for the Open-Loop PID Autotuner block
- $[1/10,1/3, 1, 3, 10]\omega_c$ for the Closed-Loop PID Autotuner block

Use the **Sine Amplitudes** parameter of the blocks to specify the amplitudes of these signals.

If your plant is asymptotically stable, the Open-Loop PID Autotuner block can estimate the plant DC gain with a step perturbation. Specify the amplitude of this perturbation with the **Step Amplitude** parameter. If your plant has a single integrator, clear the **Estimate DC gain with step signal** parameter.

Caution

- Do not use either closed-loop or open-loop PID autotuning with an unstable plant.
- Do not use open-loop PID autotuning with a plant that has more than one integrator. You can use closed-loop PID autotuning with a multiple-integrator plant.

All the perturbation amplitudes must be:

- Large enough that the perturbation overcomes any deadband in the plant actuator and generates a response above the noise level.

- Small enough to keep the plant running within the approximately linear region near the nominal operating point, and to avoid saturating the plant input or output.

For more information about setting the experiment parameters, see the Closed-Loop PID Autotuner and Open-Loop PID Autotuner block reference pages.

Step 5. Tune and Validate

After you deploy the autotuner module to your system, use a rising `start/stop` signal to begin the autotuning process. The deployed module injects the test signals into your physical plant in real time. After an appropriate time, or when the `% conv` signal stabilizes near 100%, use a falling `start/stop` signal to end the experiment. A conservative estimate for the experiment time is $200/\omega_c$ for closed-loop tuning or $100/\omega_c$ for open-loop tuning, where ω_cc is your target bandwidth. When the experiment stops, the module computes new PID gains based on the estimated frequency response at the system and your specified tuning goals. You can examine the tuned PID gains using the `pid gains` signal.

When you examine these gains and transfer them to your own controller, be aware of the meaning of these gains in the PID autotuner blocks. In discrete time, the blocks assume the following PID controller transfer function:

$$C = P + IF_i(z) + D\left[\frac{N}{1 + NF_d(z)}\right],$$

in parallel form, or in ideal form,

$$C = P\left[1 + IF_i(z) + D\left(\frac{N}{1 + NF_d(z)}\right)\right].$$

$F_i(z)$ and $F_d(z)$ depend on the values you specify for the **Integrator method** and **Filter method** formulas, respectively. For more details, see the Closed-Loop PID Autotuner or Open-Loop PID Autotuner block reference pages.

After you transfer the tuned gains to your PID controller, you can observe and validate the continued performance of your system with the new gains.

Access Autotuning Parameters After Deployment

Some of the parameters that you set to configure the autotuner are tunable, such that you can access them in the generated code. For the parameters that are not tunable, you must configure them in the block before deployment.

Tunable Parameters

The following parameters of the PID autotuner blocks are tunable after deployment. For more information about all these parameters, see the Closed-Loop PID Autotuner or Open-Loop PID Autotuner block reference pages.

Parameter	Description
Target bandwidth (rad/sec)	Target crossover frequency of open-loop response

Parameter	Description
Target phase margin (degrees)	Target minimum phase margin of open-loop response
Sine Amplitudes	Amplitude of sinusoidal perturbations
Estimate DC gain with step signal	Inject step signal into plant
Step Amplitude	Amplitude of step perturbation
Type	PID controller type (such as PI, PD, or PID)
Form	PID controller form
Integrator method	Discrete integration formula for integrator term
Filter method	Discrete integration formula for derivative filter term

Non-Tunable Parameters

The following parameters of the PID autotuner blocks are not tunable after deployment. You must specify them in the block before code generation, and their values remain fixed in your application. For more information about all these parameters, see the Closed-Loop PID Autotuner or Open-Loop PID Autotuner block reference pages.

Parameter	Description
Time Domain	PID controller time domain
Controller sample time (sec)	Sample time of PID controller (see “Modify Sample Times After Deployment” on page 8-19)
Reduce memory and avoid task overrun (external mode only)	Deploy tuning algorithm only
Data Type	Floating-point precision

Modify Sample Times After Deployment

The **Controller sample time (sec)** parameter is not tunable. As a consequence, you cannot access it directly in generated code when you deploy the block. To change the controller sample time in the deployed block at run time:

1. Set **Controller sample time (sec)** to –1.
2. Put the autotuner block in a Triggered Subsystem.
3. Trigger the subsystem at the desired sample time.

See Also

Closed-Loop PID Autotuner | Open-Loop PID Autotuner

More About

- “How PID Autotuning Works” on page 8-5
- “Control Real-Time PID Autotuning in Simulink” on page 8-20
- “PID Autotuning for a Plant Modeled in Simulink” on page 8-7

Control Real-Time PID Autotuning in Simulink

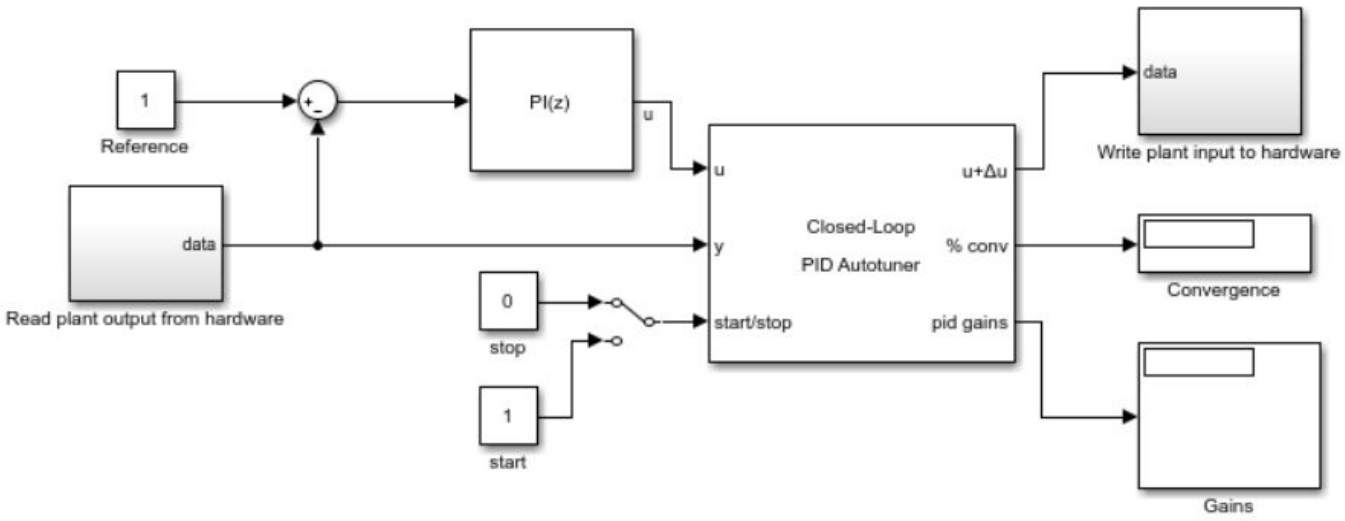
Deploying the PID autotuner blocks lets you tune your system in real time without Simulink in the loop. However, it can be useful to run the autotuning algorithm on hardware while controlling the experiment from Simulink.

One way to do so is to use a model that contains a PID controller and a PID autotuner block, and run this model in external simulation mode. External mode allows communication between the Simulink block diagram and the standalone program that is built from the generated code. In this mode, Simulink serves as a real-time monitoring interface in which you can interact with the tuning algorithm running on hardware. For instance, you can start and stop the experiment or change tuning goals from the Simulink interface while the model is running.

When tuning in external mode, you can deploy the experiment algorithm only, such that the PID tuning part of the calculation is performed in Simulink. Doing so can save memory on your target hardware. Running the PID autotuning algorithm in external mode requires a code-generation product such as Simulink Coder.

Simulink Model for External-Mode Tuning

A Simulink model for PID autotuning in external mode resembles the following illustration.

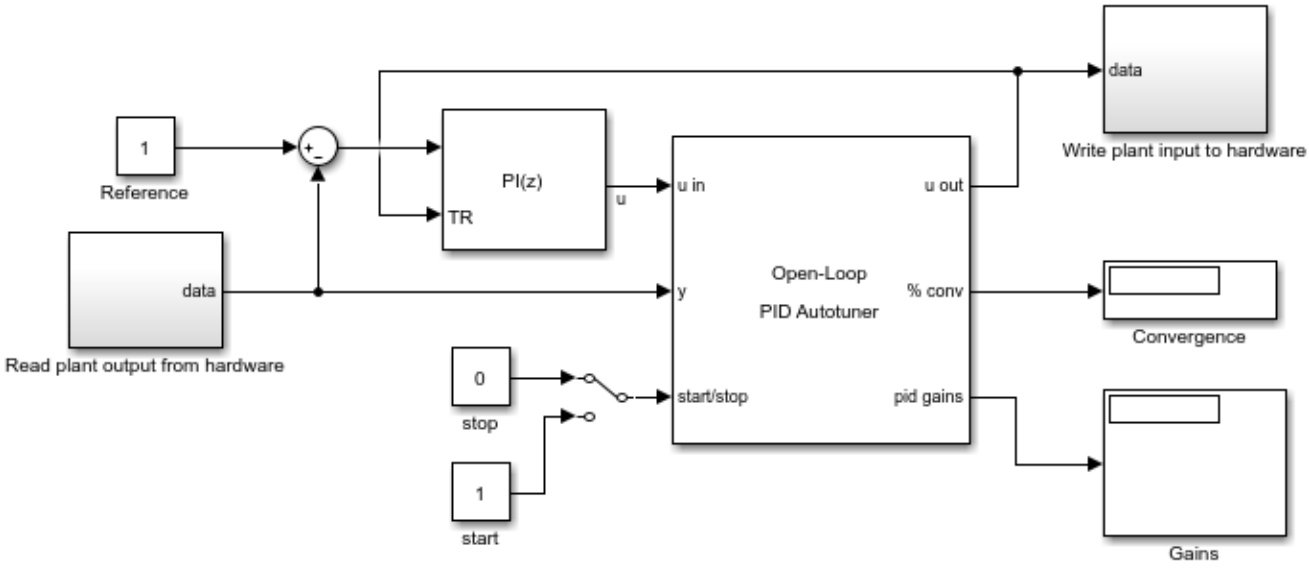


Here, the blocks marked `Read plant output from hardware` and `Write plant input to hardware` represent hardware interfaces that read data from or write data to your physical plant. When you are ready for tuning, you run this model in external simulation mode.

Bumpless Transfer for Open-Loop Tuning

When you use the Open-Loop PID Autotuner, if your controller includes integrator action, consider implementing signal tracking to avoid integrator windup during the tuning experiment. Signal tracking enables the PID controller to continue to track the real plant input while it is out of the loop. Without it, your system can experience a bump when the control loop is closed at the end of the tuning process.

If your PID controller is a Simulink PID Controller block, you can use the **Enable tracking mode** parameter of the controller block to avoid this bump. The following diagram illustrates a module containing an Open-Loop PID Autotuner block and a PID Controller block with tracking mode configured. The plant input feeds into the tracking input of the controller block.

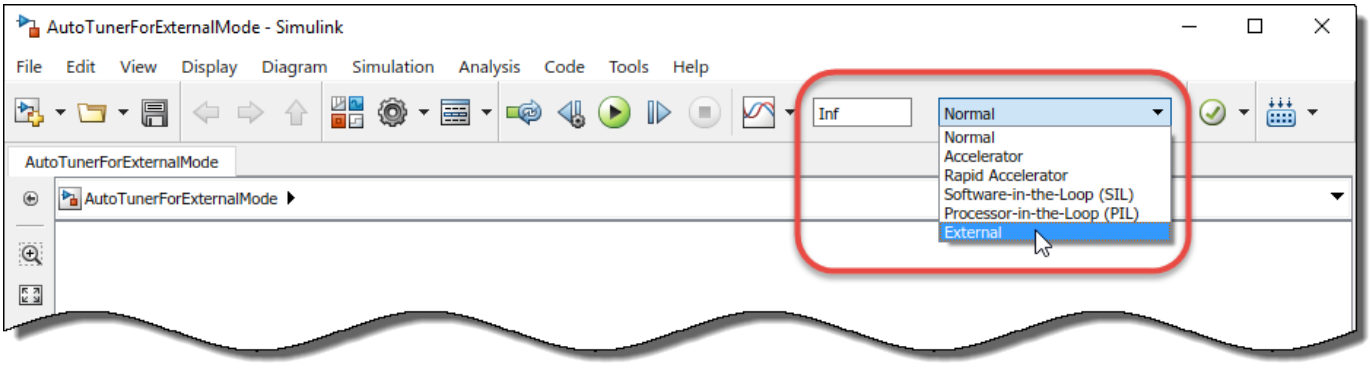


For external-mode tuning, you configure the `start-stop` signal as described in “PID Autotuning for a Plant Modeled in Simulink” on page 8-7. The models illustrated here use a simple switch with a binary signal to start and stop the experiment manually.

You also configure controller parameters, tuning goals, and experiment parameters as described in “PID Autotuning for a Plant Modeled in Simulink” on page 8-7.

Run the Model and Tune the Controller Gains

After configuring the block parameters for the experiment, in the model, select `external mode`, set the simulation time to infinite, and run the model.



Simulink compiles the model and deploys it to your connected hardware.

- If you have configured the `start/stop` signal to begin and end the tuning process at specific times, allow the simulation to run through the end of the experiment.

- If you have configured a manual `start/stop` signal, begin the experiment when your plant has reached steady-state. Observe the signal at the `% conv` output, and stop the experiment when the signal stabilizes near 100%.

When tuning is complete, examine and validate the tuned gains as described in “PID Autotuning for a Plant Modeled in Simulink” on page 8-7.

For a more detailed example that illustrates the use of external mode to control the autotuning process via Simulink, see “Tune PID Controller in Real Time Using Open-Loop PID Autotuner Block” on page 8-23.

Reduce Memory Footprint When Using External Mode

The autotuner blocks contain two modules, one that performs the real-time frequency-response estimation, and one that uses the resulting estimated response to tune the PID gains. When you run a Simulink model containing the block in the external simulation mode, by default both modules are deployed. You can save memory on the target hardware by deploying the estimation module only. In this case, the tuning algorithm runs on the Simulink host computer instead of the target hardware. To do so, use the **Reduce memory and avoid task overrun** option in the autotuner block. When this option is selected, the deployed algorithm uses about a third as much memory as when the option is cleared.

The PID gain calculation demands more computational load than the frequency-response estimation. For fast controller sample times, some hardware might not finish the gain calculation within one execution cycle. Therefore, when using hardware with limited computing power, selecting this option lets you tune a PID controller with a fast sample time.

Additionally, when you enable this option, there can be a delay of several sampling periods between when the tuning experiment ends and when the new PID gains arrive at the **pid gains** output port. Before pushing gains to the controller, first confirm the change at the **pid gains** output port instead of using **start/stop** signal as the trigger for the update.

Caution When you use this option, the model must be configured such that numeric block parameters are tunable in generated code, not inlined. To specify tunable parameters:

- In the model editor: In **Configuration Parameters**, in **Code Generation** > **Optimization**, set **Default parameter behavior** to `Tunable`.
- At the command line: Use `set_param(mdl,'DefaultParameterBehavior','Tunable')`.

See Also

Closed-Loop PID Autotuner | Open-Loop PID Autotuner

More About

- “PID Autotuning in Real Time” on page 8-13
- “When to Use PID Autotuning” on page 8-2

Tune PID Controller in Real Time Using Open-Loop PID Autotuner Block

This example shows how to use Open-Loop PID Autotuner block to tune a PI controller for an engine speed control system in both simulation and real time.

Open-Loop PID Autotuner Block

The Open-Loop PID Autotuner block allows you to tune a single-loop PID controller in real time. It carries out an open-loop experiment that injects perturbation signals to the plant and computes PID gains based on the plant frequency responses estimated near the desired bandwidth.

The Open-Loop PID Autotuner block supports two typical PID tuning scenarios in real time applications.

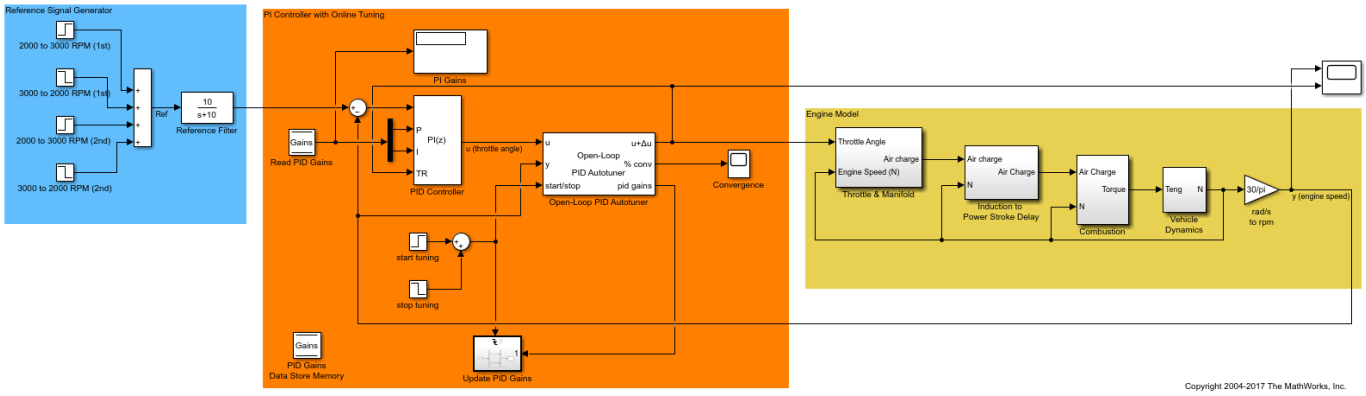
1. Deploy the block on hardware and use it in a standalone real-time application without the presence of Simulink®.
2. Deploy the block on hardware but monitor and manage the real-time tuning process in Simulink, using external mode. External mode allows communication between the Simulink block diagram running on the host computer and the generated code running on the hardware.

This example focuses on the second scenario, where the Open-Loop PID Autotuner block is used to tune an engine speed control system in real time using external mode.

Engine Speed Model

The Simulink model contains a PID block, an Open-Loop PID Autotuner block, and an engine model.

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



The PI controller has initial gains of P = 0.01 and I = 0.01, provided externally to the PID block via "P" and "I" inports. Having external P and I gains allows you to change them after new gains are computed by the Open-Loop PID Autotuner block.

The Open-Loop PID Autotuner block is inserted between the PID block and the engine model. The start/stop signal is used to start and stop an open-loop experiment. When no experiment is running, the Open-Loop PID Autotuner block behaves like a unity gain block, where the "u" signal directly passes to "u+Δu". When the experiment ends, the block tunes PID gains and outputs them at the "pid gains" port.

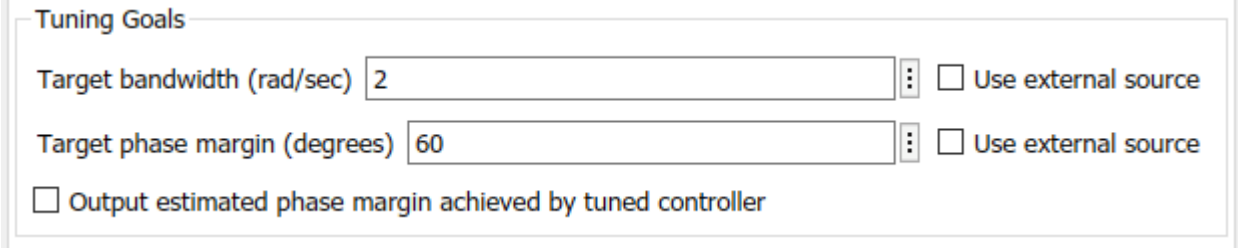
There are a few important considerations when using the Open-Loop PID Autotuner block with a physical plant in real time.

- The plant must be asymptotically stable because an open-loop experiment is conducted during the tuning process. If your plant has a single integrator, you can still use the block by choosing not to estimate the plant DC gain. However, in both cases, you must closely monitor the plant behavior during the tuning process and intervene promptly if the plant gets too close to an undesirable operating condition.
- To help estimate plant frequency responses more accurately in real time, there should be minimum load disturbance occurring during the tuning process. The block expects the plant output to be the response to the injected perturbation signals only, and load disturbance distorts this output.
- The "tracking mode" (the TR inport) in the PID block is turned on, which enables the PID block to track the real plant input "u+Δu" during the tuning process. This feature should always be used to provide a bumpless transfer when the loop is closed and PID block resumes control after the tuning process is completed.

Configure Open-Loop PID Autotuner Block

After properly connecting the Open-Loop PID Autotuner block with the plant model and PID block, open the block dialog and specify tuning and experiment settings.

On the **Tuning** tab, there are two main tuning settings.



- **Target bandwidth**: Determines how fast you want the controller to respond. In this example, choose 2 rad/sec because the desired rise time is 1 sec.
- **Target phase margin**: Determines how robust you want the controller to be. In this example, choose the default value of 60 degrees, which leads to about 5% overshoot in general.

On the **Experiment** tab, there are two main experiment settings.

- **Sine Amplitudes**: Specifies amplitudes of the injected sine waves. In this example, choose 0.1 for all four sine waves, a fraction of the nominal plant input of 9. During the tuning process, the plant output varies between 1900 and 2100 rpm, which is about +/- 5% of the nominal plant output of

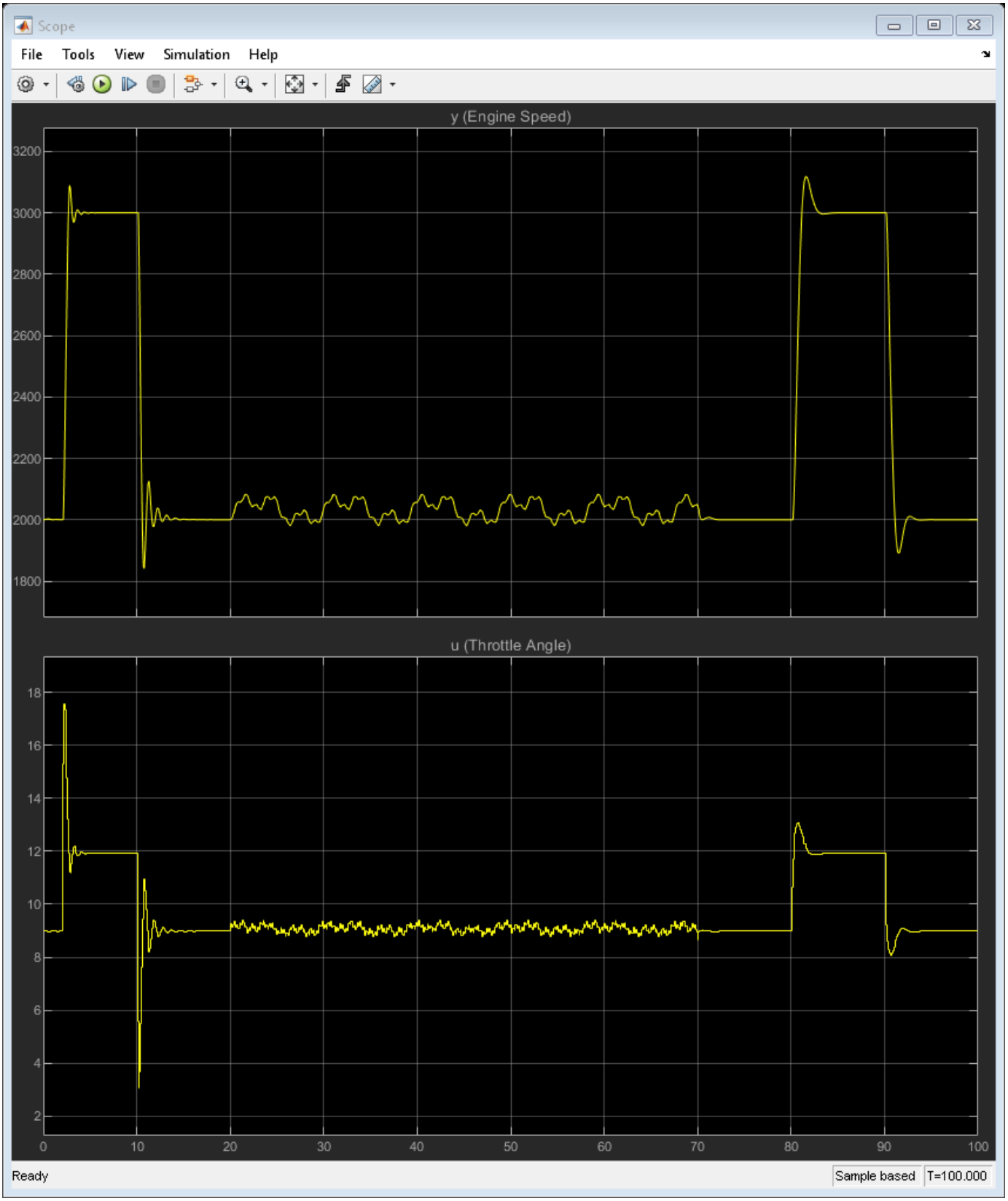
2000. The goal is to keep the plant operating near the nominal operating point to avoid exciting nonlinear plant behavior.

- **Step Amplitude**: Specifies the amplitude of the injected step signal. In this example, choose 0.1 as well. If the plant has a single integrator, you must not estimate DC gain. In this case, clear the **Estimate DC gain with step signal** parameter. As a result, no step signal is injected to the plant.

Simulate Open-Loop PID Autotuner Block in Normal Mode

If you have a plant model built in Simulink, it is recommended to simulate the Open-Loop PID Autotuner block against the plant model in normal mode before using the block in external mode for real-time tuning. Simulation helps you identify issues in signal connection and block settings so that you can adjust them before generating code.

```
sim(mdl);
```


Scope
File Tools View Simulation Help
y (Engine Speed)
3200
3000
2800
2600
2400
2200
2000
1800
u (Throttle Angle)
18
16
14
12
10
8
6
4
2
0 10 20 30 40 50 60 70 80 90 100
Ready
Sample based
T=100.000

In this example, the engine speed reference signal goes from 2000 to 3000 rpm and then back to 2000 rpm in the first 20 seconds. The original gains of P = 0.01 and I = 0.01 cause strong oscillation in the transient and must be retuned.

At 20 seconds, the plant is running at the nominal operating point of 2000 rpm and online PID tuning starts. The experiment duration is 50 seconds, because a conservative guideline suggests that it takes about 100/bandwidth seconds for online frequency response estimation to converge.

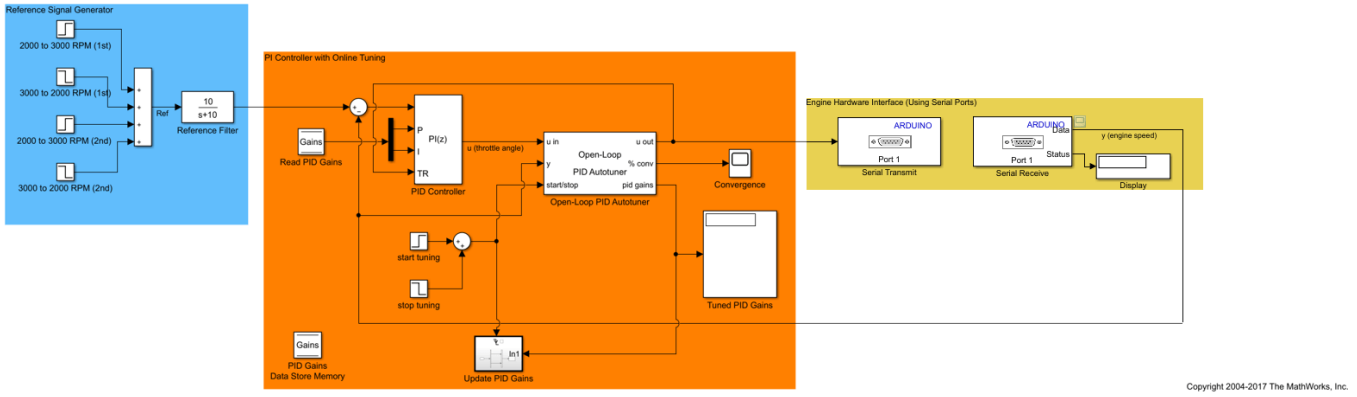
When PID tuning stops at 70 seconds, new gains P = 0.0026 and I = 0.0065 are immediately available at the "pid gains" outport and sent to the external P and I port of PID block, overwriting the original gains. There is almost no bump in transient when the loop is closed and the PID block resumes control.

The engine speed reference signal goes from 2000 to 3000 rpm and then back to 2000 rpm again between 80 and 100 seconds. The new PI gains provide a much better closed-loop response.

Using Open-Loop PID Autotuner Block in External Mode

To tune the PI controller against a physical engine in the external mode, you replace the Engine Model section in the Simulink model with hardware interface blocks that provide the rpm measurement as "y" and send throttle angle to the actuator as "u".

As an example, the following Simulink diagram is configured to tune in external mode, assuming your PI controller is running on an Arduino® DUE board and communicating with your physical engine via serial ports.



To make the original model work in external mode, the following changes were made (in order) to the original Simulink model.

1. Have a host computer that runs Simulink and communicates with an Arduino DUE board via a USB connection.
2. Install Simulink Support Package for Arduino Hardware software. You must install a different hardware support package if your hardware is different.
3. In the Configuration Parameters dialog, in the **Solver** pane, select the "Fixed-Step" solver type. In the **Hardware Implementation** pane, select the "Arduino DUE" hardware board.
4. Replace the engine model section in the original model with two serial interface blocks. In real time, the Open-Loop PID Autotuner block running on the Arduino board collects plant output from the Serial Receive block (from sensor) and sends the experiment signals to the engine using the Serial Transmit block (to actuator).

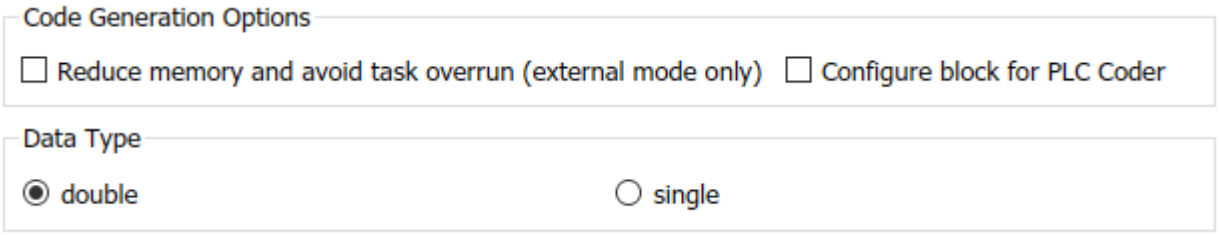
5 For more flexibility in real-time operation, start and stop the tuning process by flipping a manual "Tuning Switch" instead of based on the simulation clock. Similarly, update the PI gains by flipping a "Gain Switch" and change the reference signal by flipping a "Ref Switch".

6 Choose "External Mode" in the Simulink model and the set simulation time to "infinite".

Run the simulation. First, Simulink generates code for the whole model and downloads it to the Arduino DUE board. After the program starts running on the board, you can monitor the plant input and output from the scope in real time. When the plant reaches the nominal operating point of 2000 rpm, use the three manual switches to tune, update, and validate the controller.

Reduce Memory and Avoid Task Overrun in External Mode

On the **Block** tab, the **Reduce memory and avoid task overrun (external mode only)** option can help deploy the generated code on hardware with limited memory resources or very fast sample time.



If your hardware has low memory on board, use this option when tuning in external mode. With this option, Simulink only generates code for the online frequency response estimation functionality. Because no code is deployed for the PID design functionality, the result is reduced memory usage on the hardware. In this case, after the estimation is done, the PID gains are computed in Simulink on the host computer and then sent back to the autotuner block.

The PID gain calculation at the end of the tuning process demands much more computation load than the online frequency response estimation. If the controller sample time is very fast, some hardware might not be able to finish the calculation within an execution cycle. Therefore, having the host computer to perform the PID gain calculation also enables you to tune a PID controller with fast sample time on hardware with limited computing power.

```
bdclose(mdl)
```

See Also

Open-Loop PID Autotuner

More About

- “How PID Autotuning Works” on page 8-5
- “PID Autotuning for a Plant Modeled in Simulink” on page 8-7
- “Tune PID Controller in Real Time Using Closed-Loop PID Autotuner Block” on page 8-29

Tune PID Controller in Real Time Using Closed-Loop PID Autotuner Block

This example shows how to use the Closed-Loop PID Autotuner block to tune a PID controller for a boost converter plant in both simulation and real time.

Closed-Loop PID Autotuner Block

The Closed-Loop PID Autotuner block allows you to tune a single-loop PID controller in both simulation and real time. The block injects sinusoidal perturbation signals at the plant input and measures the plant output during a closed-loop experiment. When the experiment stops, the block computes PID gains based on the plant frequency responses estimated near the desired bandwidth.

The Closed-Loop PID Autotuner block supports two typical PID tuning scenarios in real-time applications.

1. Deploy the block on hardware and use it in a standalone real-time application, without the presence of Simulink®.
2. Deploy the block on hardware but monitor and manage the real-time tuning process in Simulink, using external simulation mode. External mode allows communication between the Simulink block diagram running on the host computer and the generated code running on the hardware.

This example focuses on the first scenario, deploying the block to perform the real-time tuning.

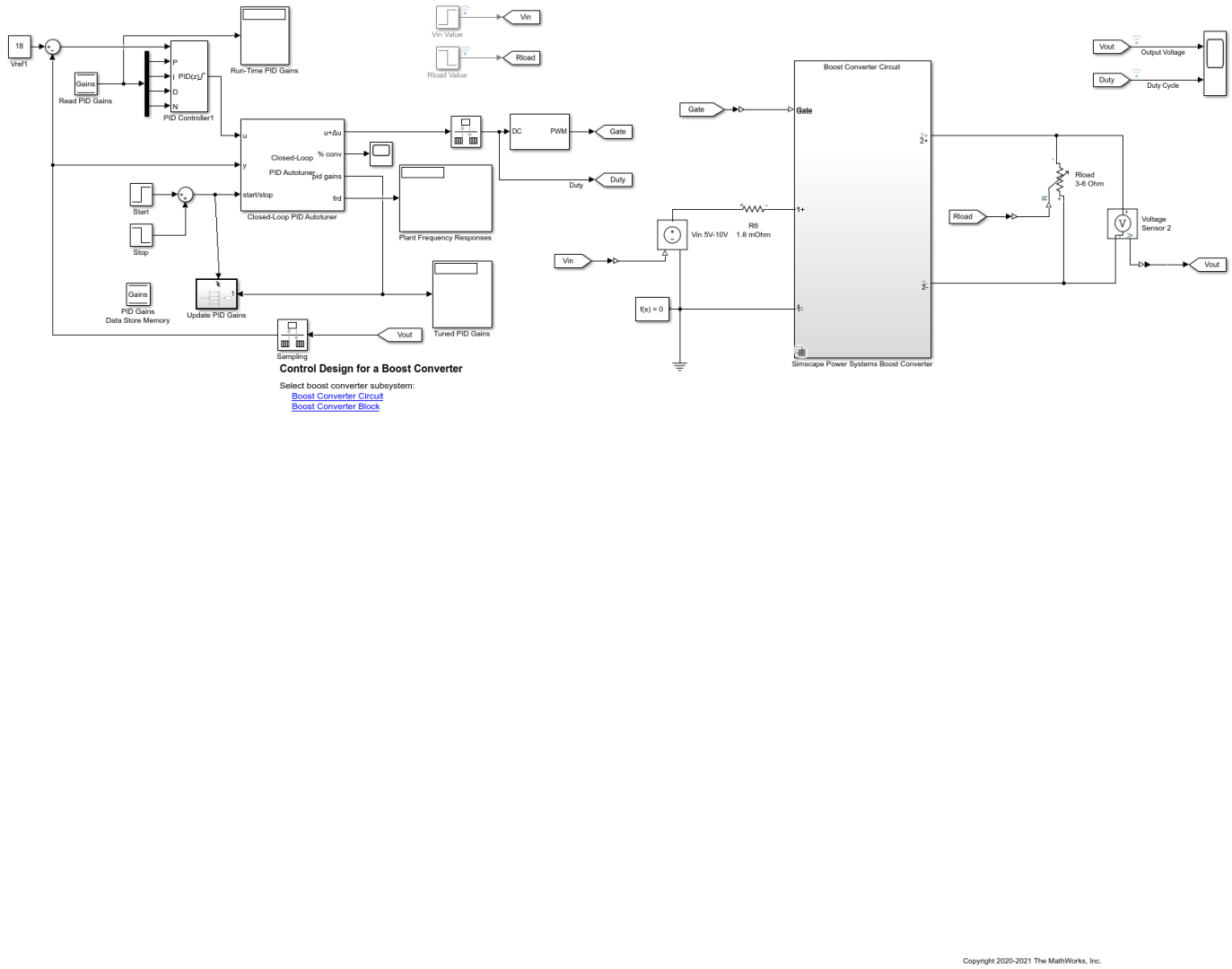
Simulink Control Design™ software also provides an Open-Loop PID Autotuner block for real-time PID tuning. The main difference between the two autotuner blocks is that the Open-Loop PID Autotuner block carries out the experiment with the feedback loop open (that is, the existing controller is not in action). To decide which autotuner block is best for your application, consider the following points:

- If you do not have an initial controller, use the Open-Loop PID Autotuner block to obtain one. You can continue using it to retune the controller or replace it with the Closed-Loop PID Autotuner block.
- If you have an initial controller, use the Closed-Loop PID Autotuner block for retuning. The major benefits are: (1) if there is an unexpected disturbance during the experiment, it is rejected by the existing controller to ensure safe operation; (2) the existing controller keeps the plant running near its nominal operating point by suppressing the perturbation signals as well.

Voltage-Mode Controlled Boost Converter

In this example, a voltage-mode boost converter is modeled in Simulink using Simscape™ Electrical™ components. The parameters of these components are based on [1].

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



A boost converter circuit converts a DC voltage to another, typically higher, DC voltage by controlled chopping or switching of the source voltage. In this model, a MOSFET driven by a pulse-width modulation (PWM) signal is used for switching. A digital PID controller adjusts the PWM duty cycle to maintain the load voltage $Vout$ at its reference $Vref$.

At the nominal operating point, the load voltage is 18 volts and the duty cycle is about 0.74. The duty cycle can vary from 0.1 to 0.85 during boost converter operation.

The existing PID controller has gains of P = 0.02, I = 160, D = 0.00005, and N = 20000. These gains are stored in a Data Store Memory block and provided externally to the PID Controller block. Having external gain input ports allows you to change the values after new gains are computed by the Closed-Loop PID Autotuner block.

Connect Autotuner Block with Plant and Controller

Insert the Closed-Loop PID Autotuner block between the PID Controller block and the plant, as shown in the boost converter model. The start/stop signal starts and stops the closed-loop experiment. When

no experiment is running, the Closed-Loop PID Autotuner block behaves like a unity gain block, where the u signal passes directly to u+Δu.

When using the Closed-Loop PID Autotuner block in either simulation or real-time applications, consider the following points.

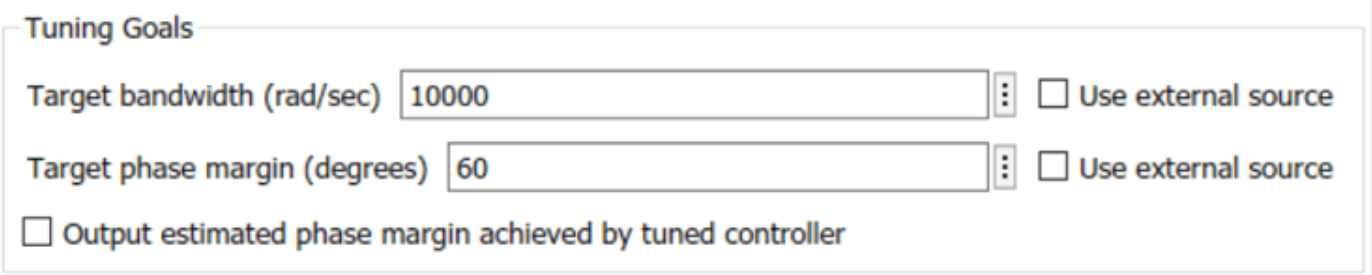
- The plant must be either asymptotically stable (all poles strictly stable) or integrating. The autotuner block does not work with an unstable plant.
- The feedback loop with the existing controller must be stable.
- To estimate plant frequency responses more accurately in real time, minimize the occurrence of any load disturbance in the plant during the experiment. The autotuner block expects the plant output to be the response to the injected perturbation signals only, and load disturbances distort this output.
- Because the feedback loop is closed during the experiment, the existing controller suppresses the injected perturbation signals as well. The advantage of using closed-loop experiment is that the controller keeps the plant running near the nominal operating point and maintains safe operation. The disadvantage is that it reduces the accuracy of frequency response estimation if your target bandwidth is far away from the current bandwidth.

Configure Autotuner Block

After properly connecting the Closed-Loop PID Autotuner block with the plant model and PID Controller block, use the block parameters to specify tuning and experiment settings.

On the **Tuning** tab, there are two main tuning settings.

- **Target bandwidth**: Determines how fast you want the controller to respond. In this example, choose `10000` rad/sec, which is typical for a boost converter.
- **Target phase margin**: Determines how robust you want the controller to be. In this example, choose the default value of `60` degrees.



On the **Experiment** tab, there are three main experiment settings.

- **Plant Type**: Specifies whether the plant is asymptotically stable or integrating. In this example, the boost converter plant is stable.
- **Plant Sign**: Specifies whether the plant has a positive or negative sign. The plant sign is positive if a positive change in the plant input at the nominal operating point results in a positive change in the plant output when the plant reaches a new steady state. Otherwise, the plant sign is negative. If a plant is stable, plant sign is equivalent to the sign of its DC gain. If a plant is integrating, the plant sign is positive or negative if the plant output keeps increasing or decreasing, respectively. In this example, the boost converter plant has a positive plant sign.

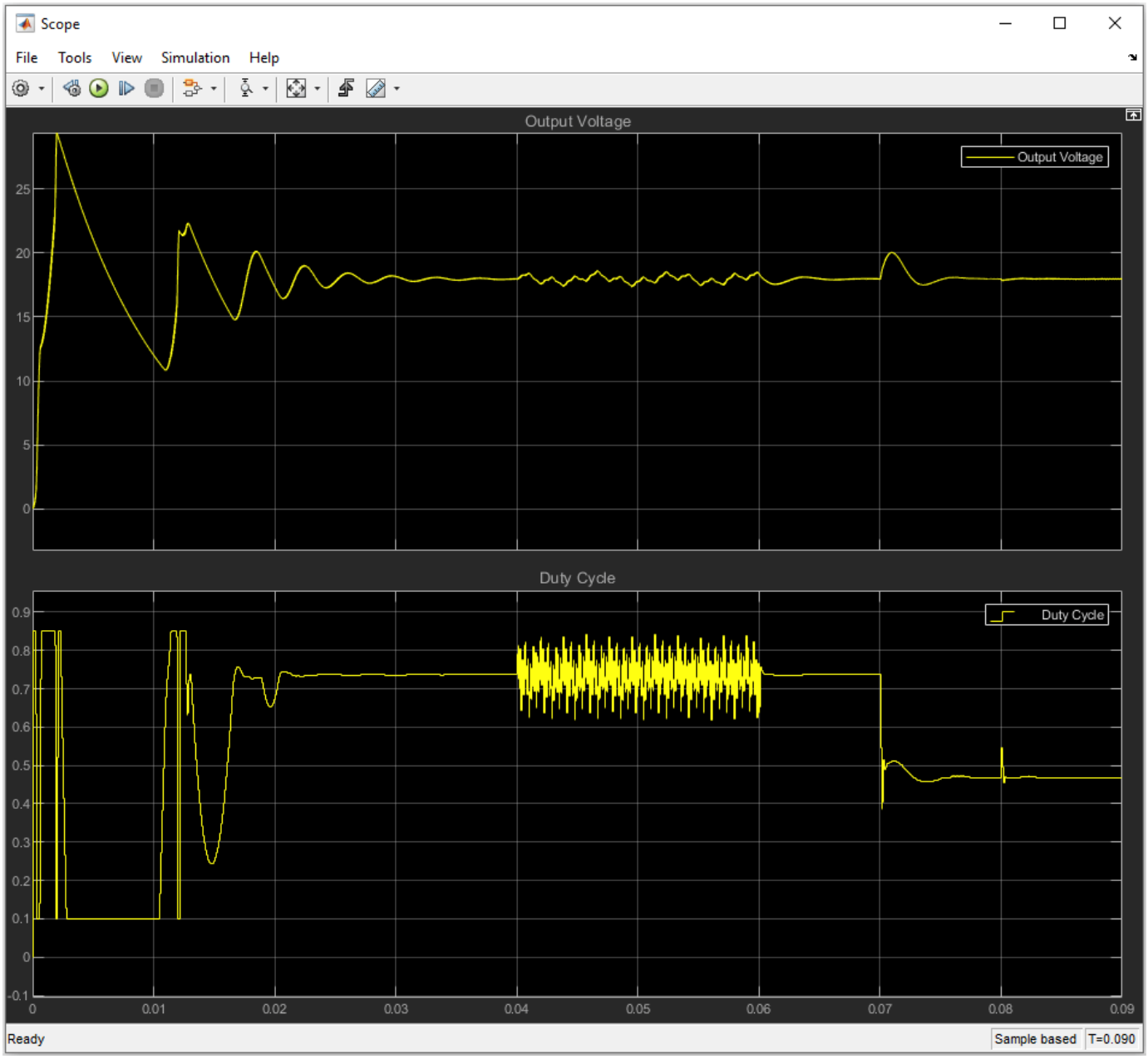
- **Sine Amplitudes**: Specifies amplitudes of the injected sine waves. In this example, choose `0.03` for all five frequencies of the perturbation signal to ensure the plant is properly excited within the saturation limit. If the excitation amplitude is too large, the boost converter operates in discontinuous-current mode. If the input amplitude is too small, the sinusoidal signals are indistinguishable from ripples in the power electronics circuits. Both situations produce inaccurate frequency response estimation results.

Simulate Autotuner Block in Normal Mode

If you have a plant model built in Simulink, it is recommended to simulate the Closed-Loop PID Autotuner block against the plant model in normal mode before deploying it for real-time tuning. Simulation helps you identify issues with signal connections and block settings so that you can adjust them before generating code.

Simulation of the boost converter plant usually takes a few minutes because of the fast sample time of the PWM generator. `Vout` is the plant output and `Duty Cycle` is the plant input.

```
sim(mdl)
```



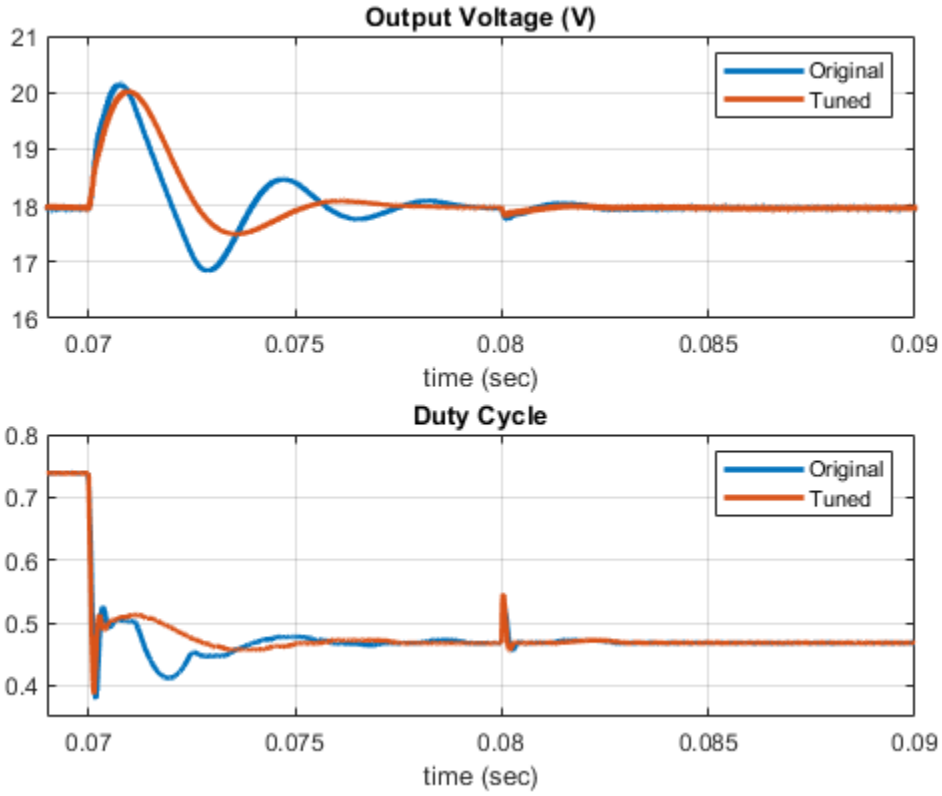
In this example, it takes the PID controller about `0.04` seconds to bring the boost converter to the nominal operating point of 18 volts. The initial transient contains strong oscillations, which indicates that the existing controller must be retuned.

At `0.04` seconds, the autotuning process starts. The experiment lasts `0.02` seconds, because the number of seconds it takes for the online frequency response estimation to converge is about 200 divided by the bandwidth.

For a different nominal operating point, it can take a longer time for the boost converter to reach the reference voltage. You must modify the start/stop time signal such that the autotuning process always starts from the nominal operating point.

When PID tuning stops at `0.06` seconds, the block calculates new gains, P = 0.04, I = 100, D = 0.00006, and N = 30000. The new gains are immediately written to the data store memory and sent to the external gain input ports of the PID Controller block, which overwrites the original gains.

The model has a line disturbance (Vin from 5V to 10V) and a load current disturbance (Load from 6A to 3A), which occur at 0.07 and 0.08 seconds, respectively. You can use these disturbances to examine controller performance. The new set of PID gains provides an improved closed-loop response with much less oscillation.



Use Autotuner Block in Standalone Application

To tune a PID controller against a physical boost converter in a standalone real-time application, you must generate C/C++ code from the Closed-Loop PID Autotuner block and deploy it on your hardware.

You can change the following tunable parameters at run time.

- PID controller type
- PID controller form
- PID integrator and filter methods (discrete time only)
- Target bandwidth
- Target phase margin
- Plant type

- Plant sign
- Amplitudes of sine waves

The sample time of the Closed-Loop PID Autotuner block is not a tunable parameter. To use the autotuner block with a different sample time without recompiling the model, set the **Controller sample time** parameter of the block to `-1` and put the autotuner block inside a triggered subsystem. Doing so runs the autotuner at the sample time of the triggered subsystem.

```
close_system(mdl,0)
```

References

[1] Lee, S. W. "Practical Feedback Loop Analysis for Voltage-Mode Boost Converter." Application Report No. SLVA633. Texas Instruments. January 2014. www.ti.com/lit/an/slva633/slva633.pdf

See Also

Closed-Loop PID Autotuner

More About

- “How PID Autotuning Works” on page 8-5
- “Tune PID Controller in Real Time Using Open-Loop PID Autotuner Block” on page 8-23

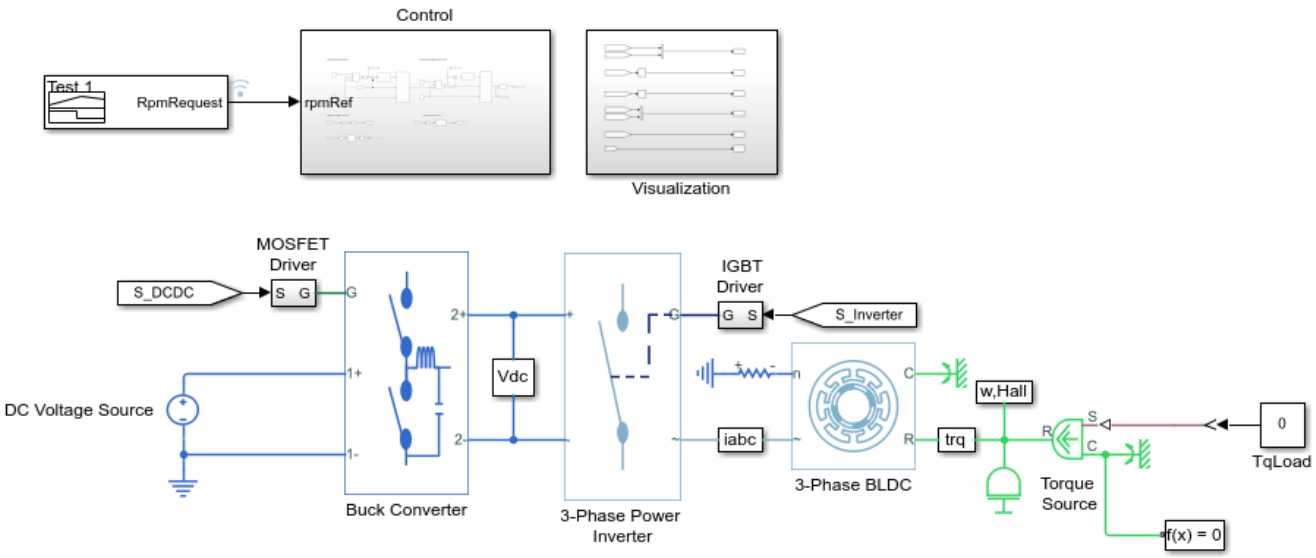
BLDC Motor Speed Control with Cascade PI Controllers

This example shows one of several ways to tune a PID controller for an existing plant in Simulink. Here, you use Closed-Loop PID Autotuner blocks to tune two PI controllers in a cascade configuration. The Autotuner blocks perturb the plant and perform PID tuning based on the plant frequency response estimated near the desired bandwidth. In contrast to the Open-Loop PID Autotuner block, here the feedback loop remains closed and the initial controller gains do not change during the autotuning process.

BLDC Motor Model

The model in this example uses a 3-phase BLDC motor coupled with a buck converter and a 3-phase inverter power link. The buck converter is modeled with MOSFETs and the inverter with IGBTs rather than ideal switches so that the device on-resistances and characteristics are represented properly. Both the voltages of the DC-DC converter link and the inverter can be controlled by changing the semiconductor gate triggers, which control the speed of the motor.

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



BLDC Speed Control with Cascade PI Controllers
This example shows how to control the rotor speed in a BLDC based electrical drive. An ideal torque source provides the load. The Control subsystem uses a PI-based cascade control structure with an outer speed control loop and an inner dc-link voltage control loop. The dc-link voltage is adjusted through a DC-DC buck converter. The BLDC is fed by a controlled three-phase inverter. The gate signals for the inverter are obtained from hall signals. The simulation uses speed steps. The Scopes subsystem contains scopes that allow you to see the simulation results.

The motor model parameters are as follows.

```
p    = 4;        % Number of pole pairs
Rs   = 0.1;      % Stator resistance per phase           [Ohm]
Ls   = 1e-4;     % Stator self-inductance per phase, Ls  [H]
Ms   = 1e-5;     % Stator mutual inductance, Ms          [H]
psim = 0.0175;   % Maximum permanent magnet flux linkage [Wb]
Jm   = 0.0005;   % Rotor inertia                         [Kg*m^2]
Ts   = 5e-6;     % Fundamental sample time               [s]
```

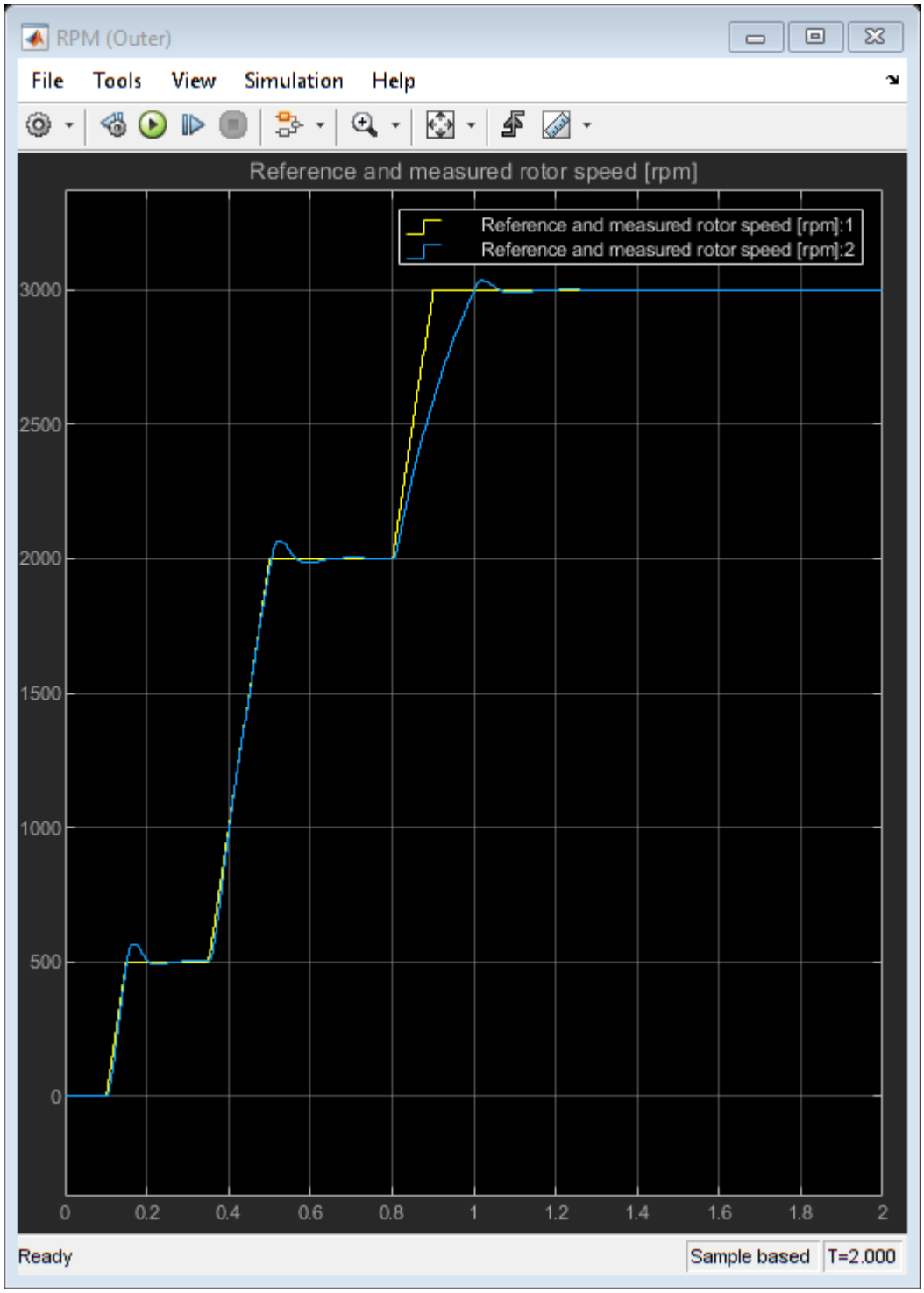
```
Tsc = 1e-4;      % Sample time for inner control loop    [s]
Vdc = 48;        % Maximum DC link voltage               [V]
```

The model is preconfigured to have stable closed-loop operation with two cascaded PI controllers, one for the inner DC link voltage loop, and one for the outer motor-speed loop.

```
Kpw = 0.1;    % Proportional gain for speed controller
Kiw = 15;     % Integrator gain for speed controller
Kpv = 0.1;    % Proportional gain for voltage controller
Kiv = 0.5;    % Integrator gain for voltage controller
```

The signal for testing the tracking performance is a series of speed ramps from 0-500 RPM, 500-2000 RPM, and 2000-3000 RPM. Simulating the model with initial controller gains shows slow tracking response, indicating that controller recalibration is needed.

```
open_system([mdl '/Visualization/RPM (Outer)'])
sim(mdl)
```

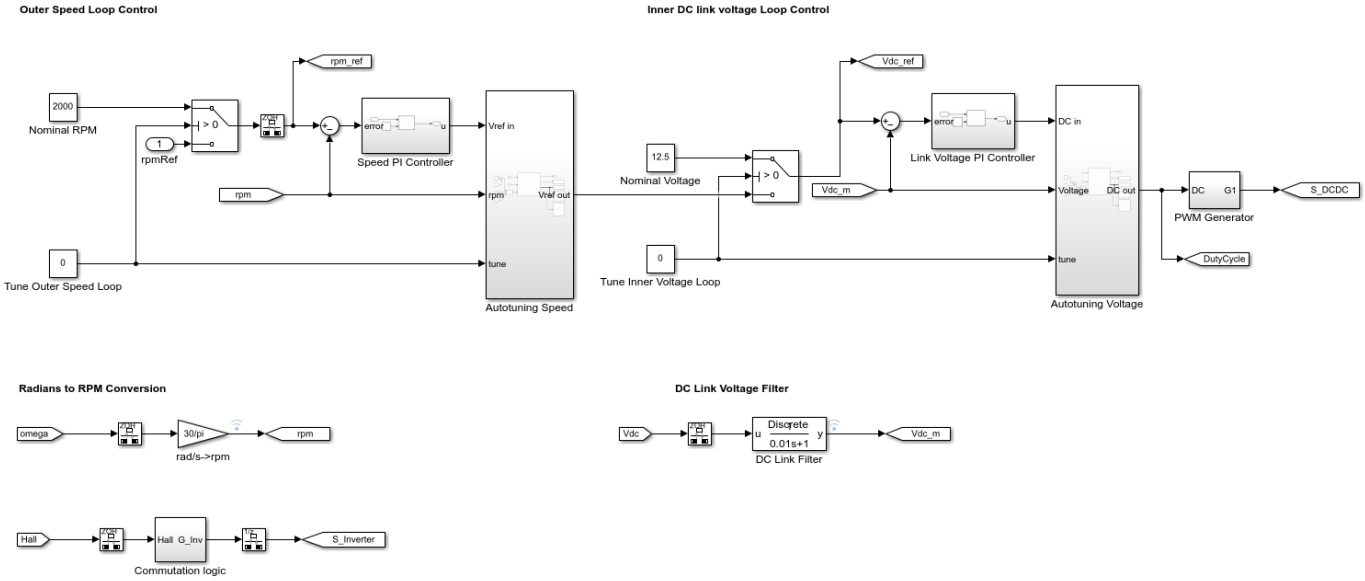


Configure Closed-Loop PID Autotuner Blocks

In this example, you improve the controller performance using Closed-Loop PID Autotuner blocks. These blocks estimate the plant frequency response with the loop closed during the experiment and

then tune the controller gains. Examine the Control subsystem to see the Closed-Loop PID Autotuner blocks in the Autotuning Speed and Autotuning Voltage subsystems.

```
open_system([mdl '/Control'])
```



Following the typical cascade loop tuning practice, first tune the inner voltage loop with the outer speed loop open. Then, tune the outer speed loop with the inner voltage loop closed.

To specify tuning requirements for the PID controllers, use the parameters on the **Tuning** tab of each of the PID autotuner blocks. In this example, the controllers are parallel, discrete-time, PI controllers. The controller sample time is 100 microseconds.

A **Target Phase Margin** of 60 degrees for both controllers gives a good balance between performance and robustness.

For the outer-loop controller, choose a **Target Bandwidth** of 100 rad/sec. For the inner-loop controller, choose an estimated target bandwidth of 400 rad/sec. These values ensure that the inner-loop controller has a faster response than the outer-loop controller.

The Closed-Loop PID Autotuner block performs a closed-loop experiment to obtain the plant frequency response. You specify parameters for this experiment on the **Experiment** tab of the block parameters. Here, **Plant Sign** is **Positive**, as a positive change in the plant input at the nominal operating point results in a positive change in the plant output, when the plant reaches a new steady state. When the plant is stable, as in this example, the plant sign is equivalent to the sign of its DC gain.

For the amplitude of the sine waves injected during the autotuning process, use 1 to ensure that the plant is suitably excited while remaining within the plant saturation limit. If the amplitude you choose is too small, the autotuner block has difficulty distinguishing the response signals from ripple in the power electronics circuits.

Tune Inner-Loop PI Controller

For tuning cascade controllers, set up the model for tuning the inner voltage loop first, followed by the outer speed loop.

To enable the tuning process for the inner-loop controller, in the Autotuning Voltage subsystem, set the Tune Inner Voltage Loop constant block value to 1. Setting this value opens the outer loop and configures the inner loop to uses a constant nominal voltage reference of 12.5 instead.

```
set_param([mdl '/Control/Tune Inner Voltage Loop'],'Value','1')
```

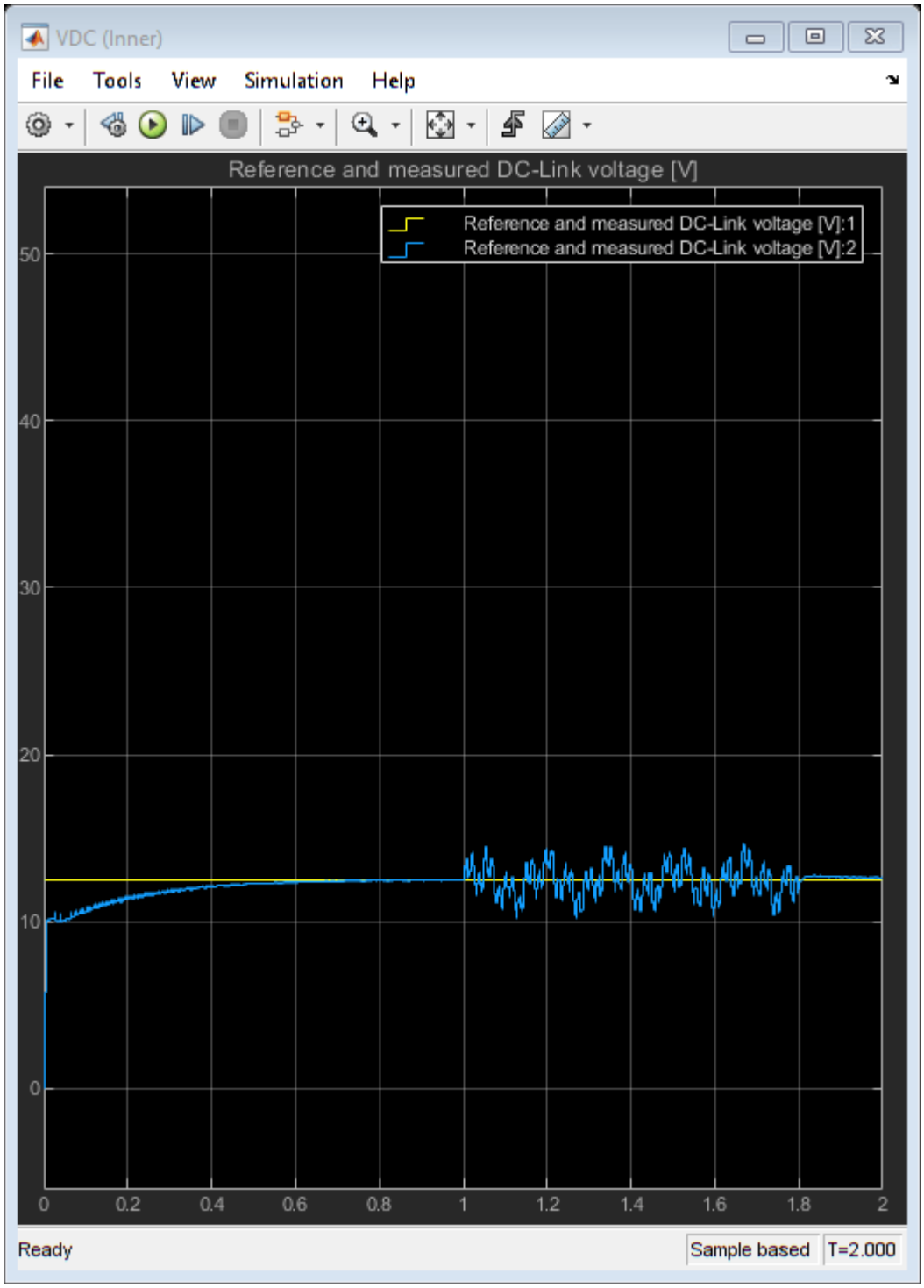
Also, to disable outer loop tuning, set the Tune Outer Speed Loop constant block value to 0.

```
set_param([mdl '/Control/Tune Outer Speed Loop'],'Value','0')
```

This setting enables the Closed-Loop PID Autotuner block which is configured to run a closed-loop tuning experiment from 1 to 1.8 seconds of simulation time. The plant uses the first second to reach a steady-state operating condition. A good estimate for a closed-loop experiment duration is $200/b_t$, where b_t is the target bandwidth. You can use the `% conv` output of the Closed-Loop PID Autotuner block to monitor the progress of the experiment and stop it when the `% conv` signal stabilizes near 100%.

Run the simulation. When the experiment concludes, the Closed-Loop PID Autotuner block returns the tuned PID controller gains for the inner voltage loop. The model sends them to the MATLAB workspace as the array `VoltageLoopGains`.

```
close_system([mdl '/Visualization/RPM (Outer)'])
open_system([mdl '/Visualization/VDC (Inner)'])
sim(mdl)
```



Update the inner loop PI controller with the new gains.

```
Kpv = VoltageLoopGains(1);
Kiv = VoltageLoopGains(2);
```

Tuning Outer Loop PI Controller

Next, tune the outer speed loop. In the Autotuning Voltage subsystem, change the value of the Tune Inner Voltage Loop constant block value to 0, which disables the inner voltage loop tuning. The inner-loop controller uses the newly tuned gains, `Kpv` and `Kiv`.

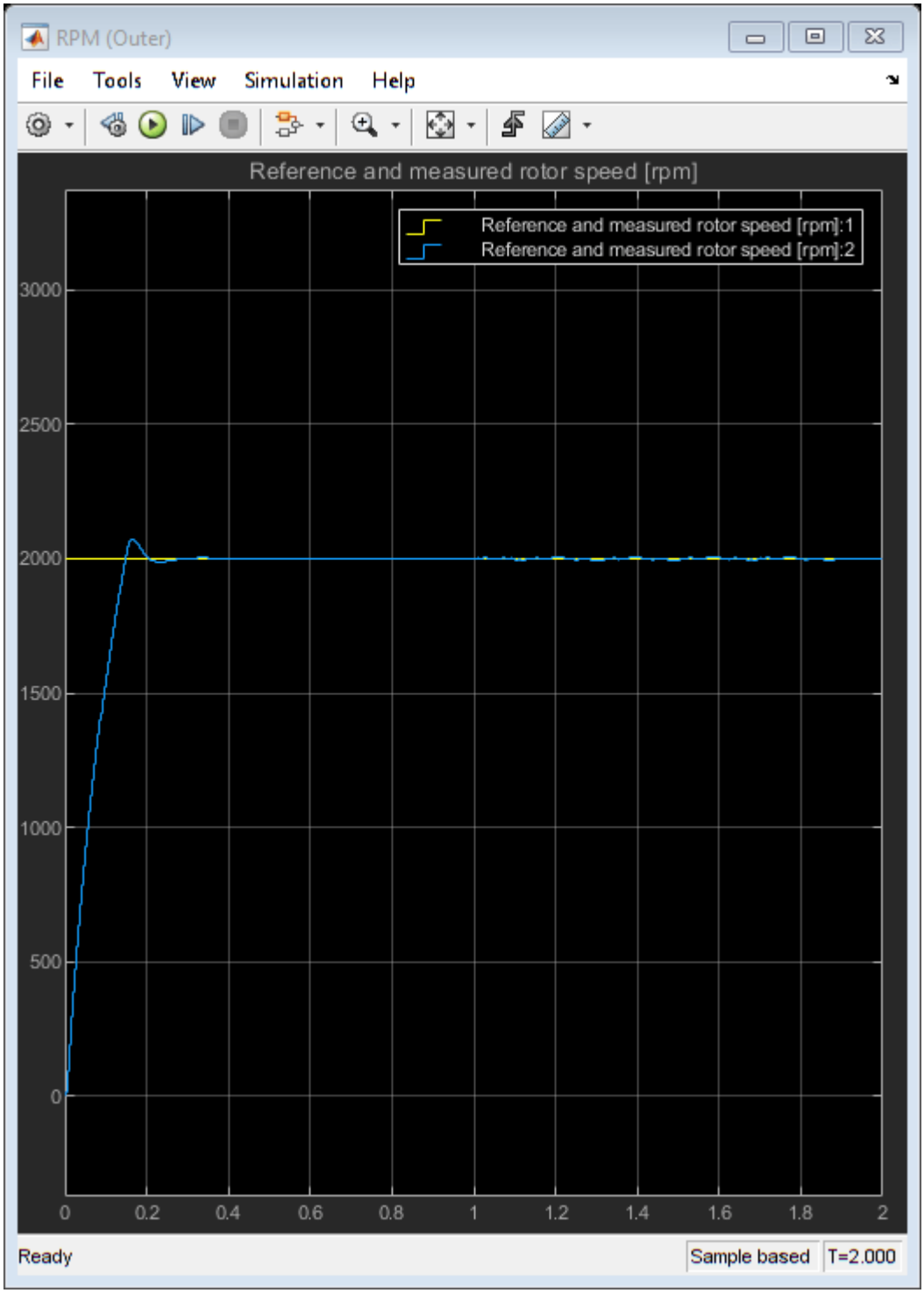
```
set_param([mdl '/Control/Tune Inner Voltage Loop'],'Value','0')
```

Similarly, in the Autotuning Speed subsystem, change the Tune Outer Speed Loop constant block value to 1, which enables the outer speed loop tuning. For this loop, use a closed-loop autotuning duration of 0.9 seconds, beginning at 1 second. The nominal speed for tuning is 2000 RPM.

```
set_param([mdl '/Control/Tune Outer Speed Loop'],'Value','1')
```

Run the simulation again. When the experiment concludes, the Closed-Loop PID Autotuner block returns the tuned PID controller gains for the outer speed loop. The model sends them to the MATLAB workspace as the array `SpeedLoopGains`.

```
close_system([mdl '/Visualization/VDC (Inner)'])
open_system([mdl '/Visualization/RPM (Outer)'])
sim(mdl)
```



Update the outer-loop PI controller with the new gains.

```
Kpw = SpeedLoopGains(1);
Kiw = SpeedLoopGains(2);
```

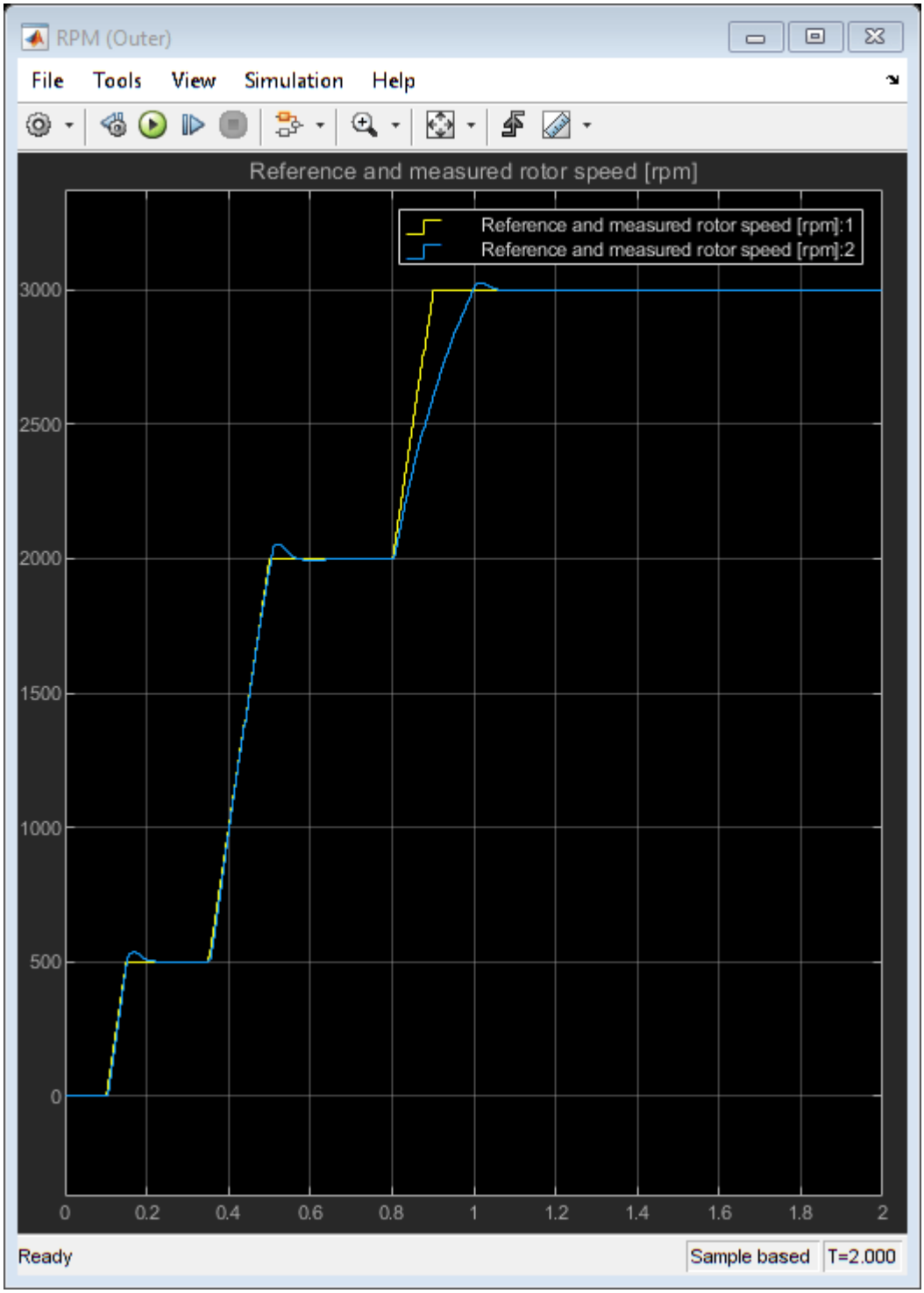
Improved Tracking Performance After Autotuning

To check the tuned controller performance, disable tuning in both loops.

```
set_param([mdl '/Control/Tune Inner Voltage Loop'],'Value','0')
set_param([mdl '/Control/Tune Outer Speed Loop'],'Value','0')
```

The tuned gains result in better tracking of the test ramp signals.

```
sim(mdl)
```



See Also

Closed-Loop PID Autotuner

More About

- “How PID Autotuning Works” on page 8-5
- “Tune PID Controller in Real Time Using Open-Loop PID Autotuner Block” on page 8-23

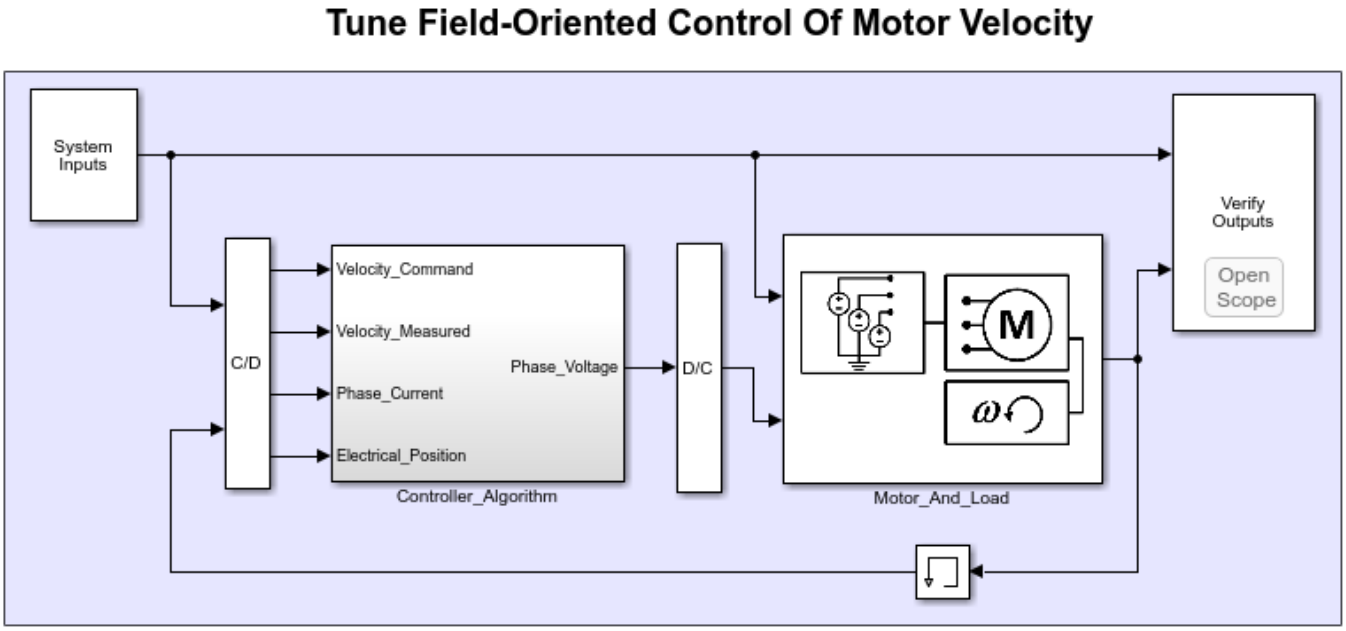
Tune Field-Oriented Controllers Using Closed-Loop PID Autotuner Block

This example shows how to use the Closed-Loop PID Autotuner block to tune Field-Oriented Control (FOC) for a permanent magnet synchronous machine (PMSM) in just one simulation.

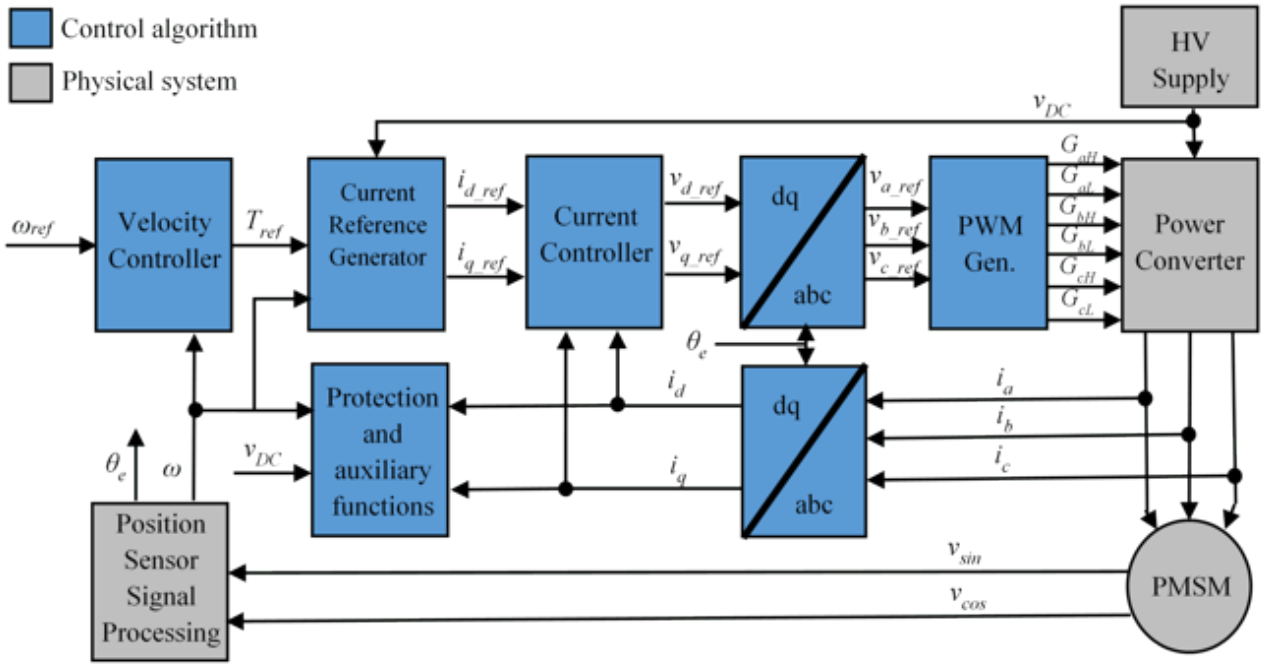
Introduction of Field-Oriented Control

In this example, field-oriented control (FOC) for a permanent magnet synchronous machine (PMSM) is modeled in Simulink® using Simscape™ Electrical™ components.

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



Field-oriented control (FOC) controls 3-phase stator currents as a vector. FOC is based on projections, which transform a 3-phase time- and speed-dependent system into a two coordinate time-invariant system. These transformations are the Clarke Transformation, Park Transformation, and their respective inverse transforms. These transformations are implemented as blocks within the Controller_Algorithm subsystem.



The advantages of using FOC to control AC motors include:

- Torque and flux controlled directly and separately
- Accurate transient and steady-state management
- Similar performance compared to DC motors

The Controller_Algorithm subsystem contains all three PI controllers. The outer-loop PI controller regulates the speed of the motor. The two inner-loop PI controllers control the d-axis and q-axis currents separately. The command from the outer loop PI controller directly feeds to the q-axis to control torque. The command for the d-axis is zero for PMSM because the rotor flux is fixed with a permanent magnet for this type of AC motor.

The existing speed PI controller has gains of P = 0.08655 and I = 0.1997. The current PI controllers both have gains of P = 1 and I = 200.

The controller gains are stored in a Data Store Memory block and provided externally to each PID block. When the tuning process for a controller is complete, the new tuned gains are written to the Data Store Memory block. This configuration allows you to update your controller gains in real-time during the simulation.

Closed-Loop PID Autotuner Block

The Closed-Loop PID Autotuner block allows you to tune one PID controller at a time. It injects sinusoidal perturbation signals at the plant input and measures the plant output during a closed-loop experiment. When the experiment stops, the block computes PID gains based on the plant frequency responses estimated at a small number of points near the desired bandwidth. For this FOC PMSM model, the Closed-Loop PID Autotuner block can be used for each of the three PI controllers.

This workflow applies when you have initial controllers that you want to retune using the Closed-Loop PID Autotuner block. The benefits of this approach are:

1. If there is an unexpected disturbance during the experiment, it will be rejected by the existing controller to ensure safe operation.
2. The existing controller will keep the plant running near its nominal operating point by suppressing the perturbation signals.

When using the Closed-Loop PID Autotuner block for both simulations and real-time applications:

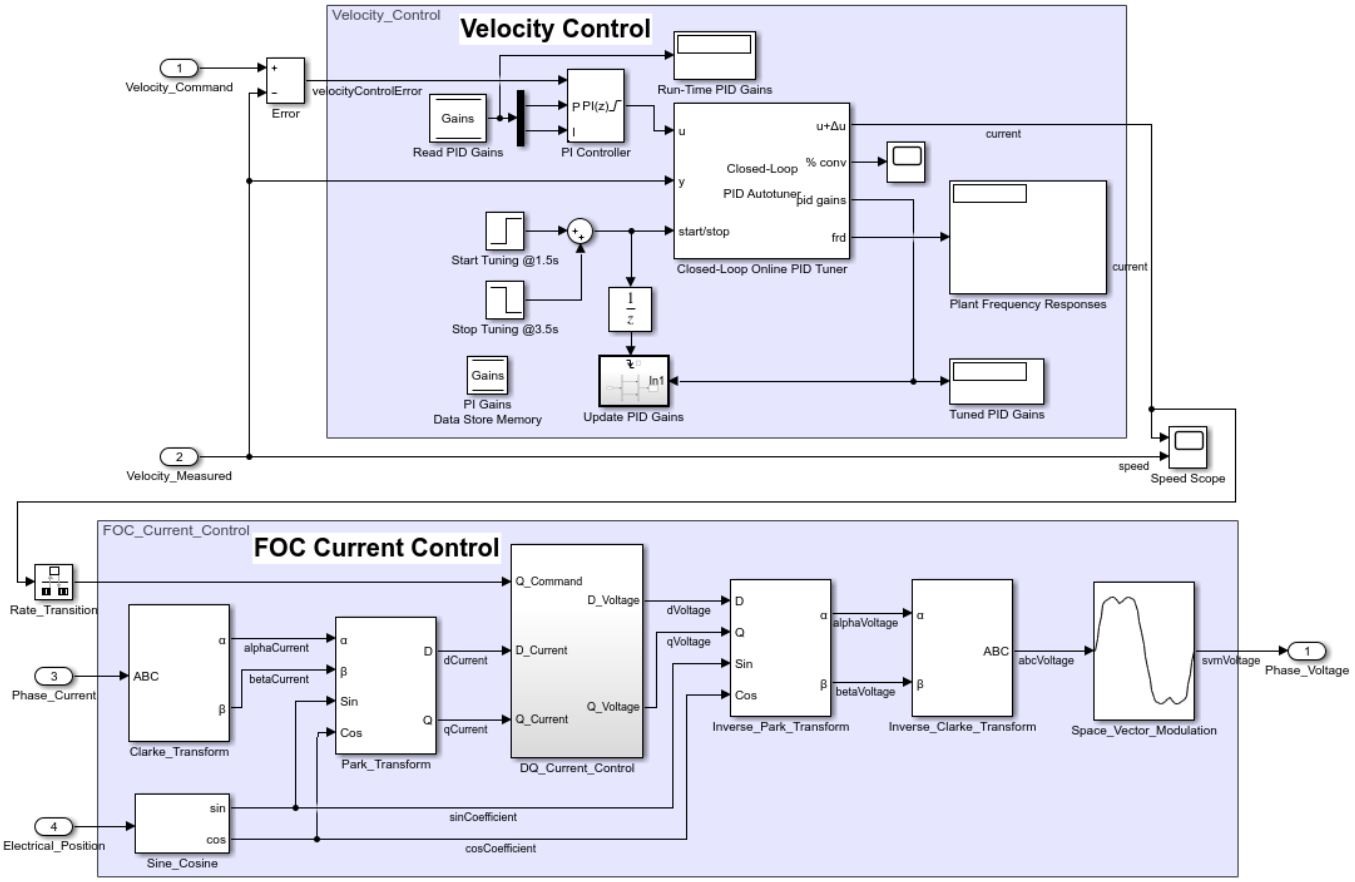
- The plant must be either asymptotically stable (all the poles are strictly stable) or integrating. The autotuner block does not work with an unstable plant.
- The feedback loop with the existing controller must be stable.
- To estimate plant frequency responses more accurately in real time, minimize the occurrence of any disturbance in the FOC PMSM model during the experiment. The autotuner block expects the plant output to be the response to the injected perturbation signals only.
- Because the feedback loop is closed during the experiment, the existing controller suppresses the injected perturbation signals as well. The advantage of using closed-loop experiment is that the controller keeps the plant running near the nominal operating point and maintains safe operation. The disadvantage is that it reduces the accuracy of frequency response estimation if your target bandwidth is far away from the current bandwidth.

Connect Autotuner with Plant and Controller

Insert the Closed-Loop PID Autotuner block between the PID block and the plant for all three PI controllers, as shown in the FOC PMSM model. The `start/stop` signal starts and stops the closed-loop experiment. When no experiment is running, the Closed-Loop PID Autotuner block behaves like a unity gain block, where the u signal directly passes to $u + \Delta u$.

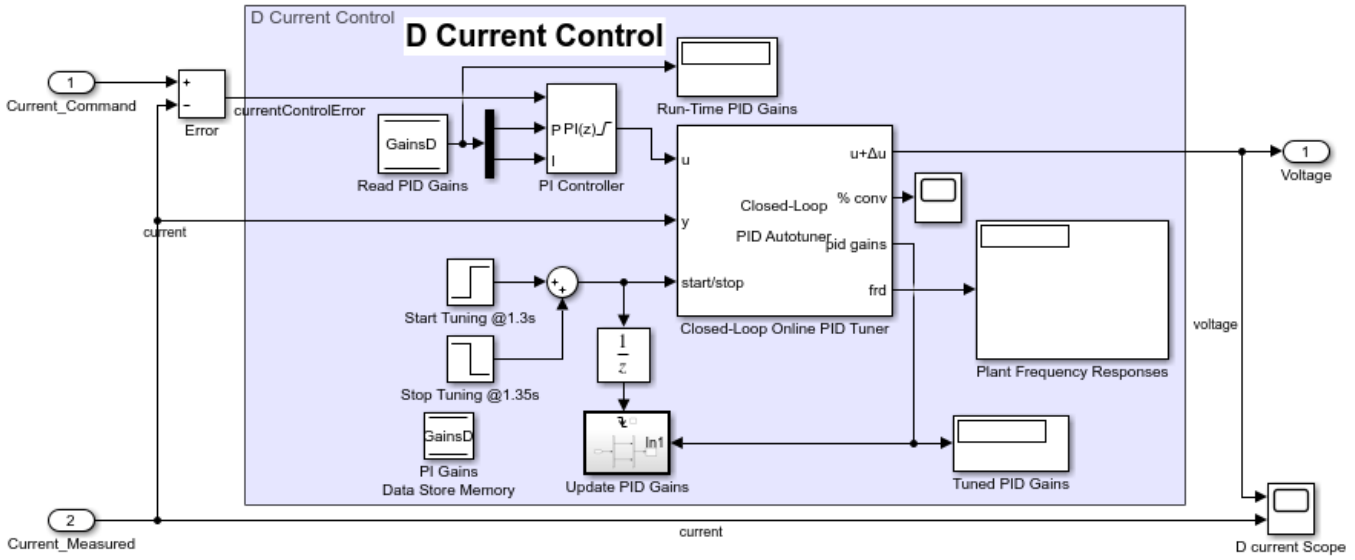
To view the modified outer-loop control structure, open the Controller_Algorithm subsystem.

```
controlSubsystem = [mdl '/Controller_Algorithm'];
open_system(controlSubsystem)
```



View the modified d-axis current controller. The modified q-axis controller has an identical structure.

```
open_system([controlSubsystem '/DQ_Current_Control/D_Current_Control'])
```



Configure Autotuner Block

After connecting the Closed-Loop PID Autotuner block with the plant model and PID block, configure the tuning and experiment settings.

On the **Tuning** tab, there are two main tuning settings:

- **Target bandwidth** - Determines how fast you want the controller to respond. In this example, choose `5000` rad/sec for current control and `100` rad/sec for speed control.
- **Target phase margin** - Determines how robust you want the controller to be. In this example, choose `70` degrees for current control and `90` degree for speed control.

On the **Experiment** tab, there are three main experiment settings:

- **Plant Type** - Specifies whether the plant is asymptotically stable or integrating. In this example, the FOC PMSM model is stable.
- **Plant Sign** - Specifies whether the plant has a positive or negative sign. The plant sign is positive if a positive change in the plant input at the nominal operating point results in a positive change in the plant output when the plant reaches a new steady state. Otherwise, the plant sign is negative. If a plant is stable, the plant sign is equivalent to the sign of its dc gain. If a plant is integrating, the plant sign is positive (or negative) if the plant output keeps increasing (or decreasing). In this example, the FOC PMSM model has a positive plant sign.
- **Sine Amplitudes** - Specifies the amplitudes of the injected sine waves. In this example, choose `0.25` for the current controllers and `0.01` for the speed controller to ensure the plant is properly excited within the saturation limit. If the excitation amplitude is either too large or too small, it will produce inaccurate frequency response estimation results.

Tuning Cascaded Feedback Loops

Because the Closed-Loop PID Autotuner block only tunes one PI controller at a time, the three controllers must be tuned separately in the FOC PMSM model. Tune the inner-loop controllers first, and then tune the outer-loop controller.

- The d-axis current controller is tuned between 1.3 and 1.35 sec.
- The q-axis current controller is tuned between 1.4 and 1.45 sec.
- The speed controller is tuned between 1.5 and 3.5 sec.

After tuning each PI controller, the controller gains are updated through the Data Store Memory block.

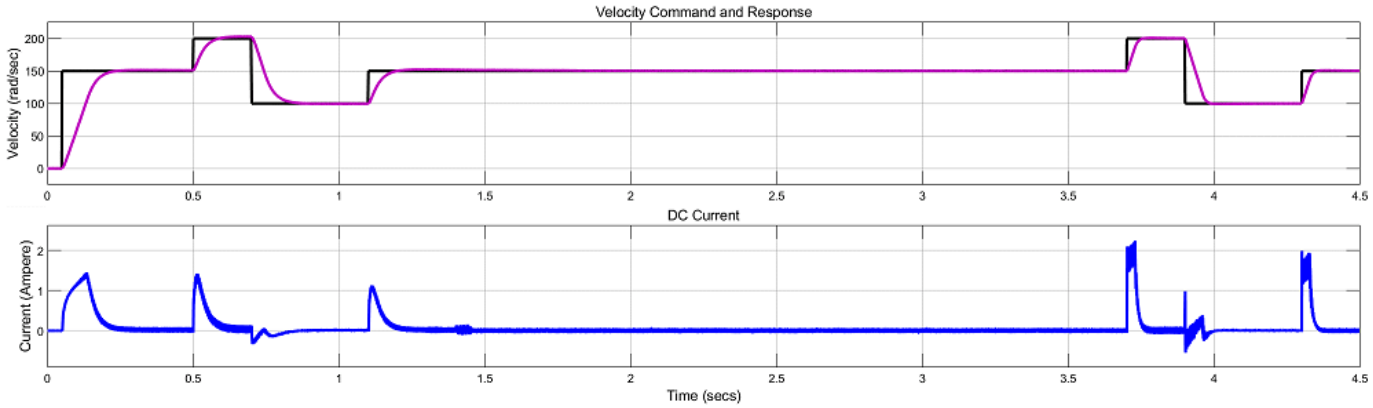
Simulating Autotuner Block in Normal Mode

In this example, the FOC PMSM model is built in Simulink. All three controllers are tuned in one simulation. In addition, responses are compared between speed responses before and after tuning the controllers.

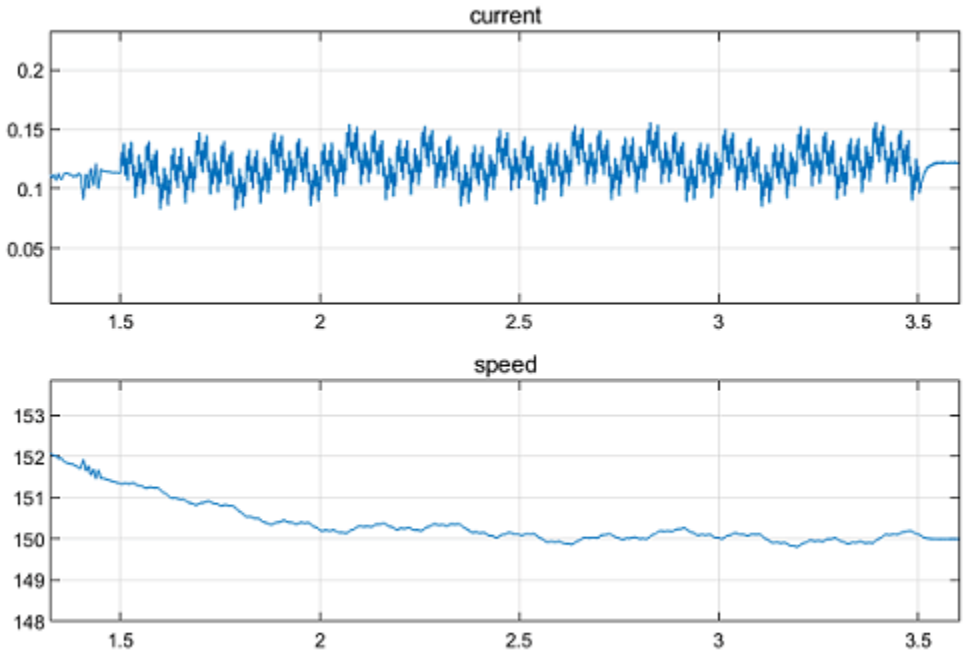
Simulation of the FOC PMSM model usually takes a few minutes on your computer due to the small sample time of the power electronics controller of the motor.

```
sim(mdl)
logsout_autotuned = logsout;
save('AutotunedSpeed','logsout_autotuned')
```

The following figure shows the overall simulation result.



The following figure shows the current and speed responses during tuning, from 1.3 to 3.5 seconds. The change in current is within `0.1` A and the change in motor speed is within 2 rad/sec (about 1% deviation).

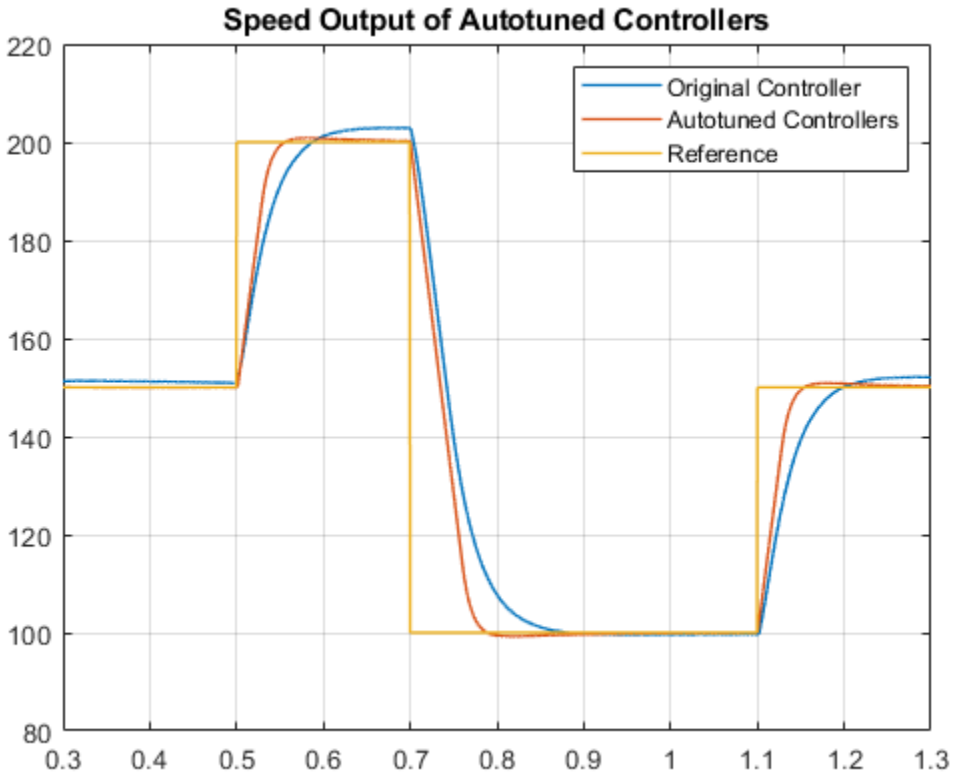


The three PI controllers are tuned with new gains.

- The speed PI controller has gains of P = 0.2785 and I = 2.678.
- The d-axis current PI controller has gains of P = 5.135 and I = 8663.
- The q-axis current PI controller has gains of P = 4.59 and I = 8026.

The same velocity commands are applied before and after the autotuning process. Plot the speed responses before and after the controllers are tuned using the Closed-Loop PID Autotuner block. The speed response curves are aligned in time to compare controller performances side-by-side.

```
scdfocmotorPIDTuningPlotSpeed
```



After tuning the controllers, the speed response of the AC motor has a faster transient response and smaller steady-state error.

```
bdclose(mdl)
```

See Also

Closed-Loop PID Autotuner

More About

- “How PID Autotuning Works” on page 8-5
- “Tune PID Controller in Real Time Using Open-Loop PID Autotuner Block” on page 8-23
- “Tune Field-Oriented Controllers for an Asynchronous Machine Using Closed-Loop PID Autotuner Block” on page 8-54
- “Tune Field-Oriented Controllers Using SYSTUNE” on page 7-160

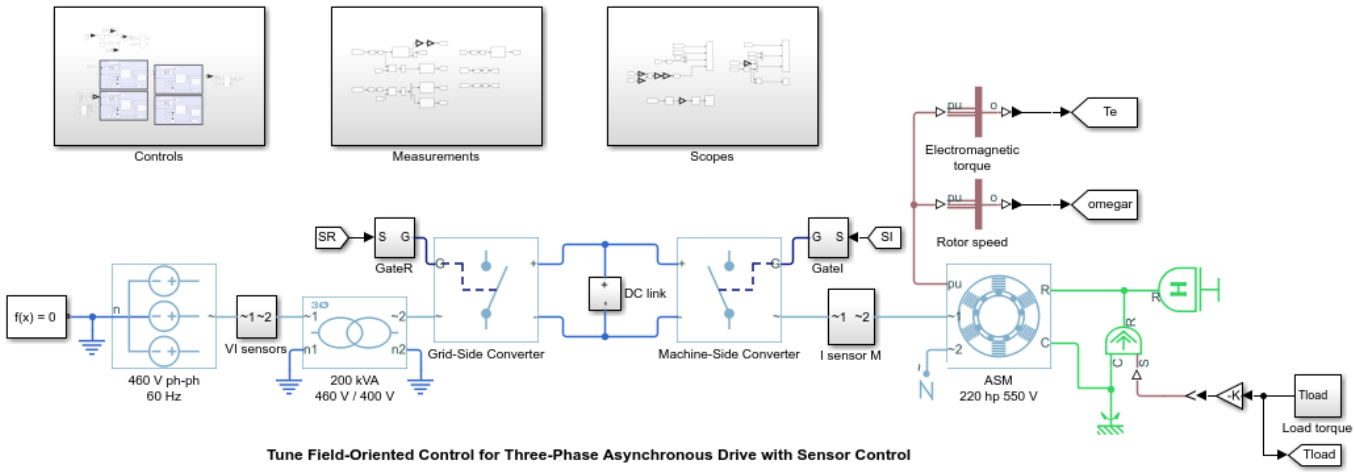
Tune Field-Oriented Controllers for an Asynchronous Machine Using Closed-Loop PID Autotuner Block

This example shows how to use the Closed-Loop PID Autotuner block to tune Field-Oriented Control (FOC) for an asynchronous machine (ASM) in just one simulation.

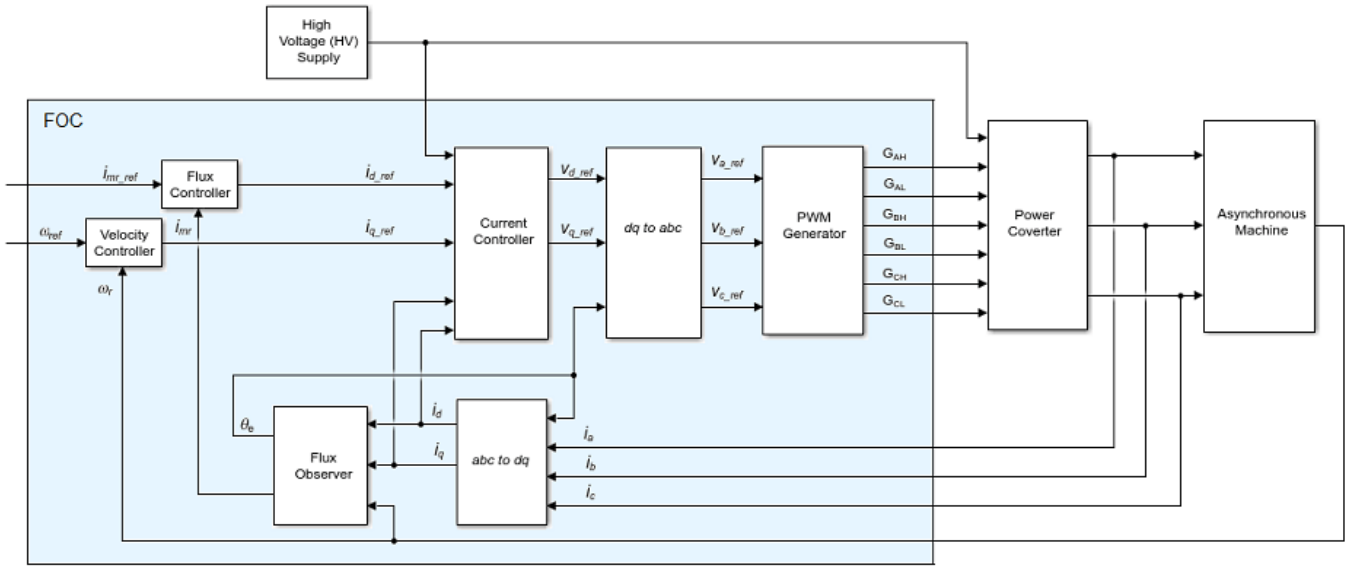
Introduction of Field-Oriented Control

In this example, field-oriented control (FOC) for an asynchronous machine (ASM) is modeled in Simulink® using Simscape™ Electrical™ components. The model is based on the Simscape example "Three-Phase Asynchronous Drive with Sensor Control" (Simscape Electrical).

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



Field-oriented control controls 3-phase stator currents as a vector. FOC is based on projections, which transform a 3-phase time-dependent and speed-dependent system into a two coordinate time-invariant system. These transformations are the Clarke Transformation, Park Transformation, and their respective inverse transforms. These transformations are implemented as blocks within the Controls subsystem.



The advantages of using FOC to control AC motors include:

- Torque and flux controlled directly and separately
- Accurate transient and steady-state management
- Similar performance compared to DC motors

The Controls subsystem contains all four PI controllers. The outer-loop speed PI controller regulates the speed of the motor. The outer-loop flux PI controller regulates the flux of stator. The two inner-loop PI controllers control the d-axis and q-axis currents separately. The command from the outer-loop speed PI controller directly feeds to the q-axis to control torque. The command for the d-axis is nonzero for ASM and is a result of the outer-loop flux PI controller.

The existing PI controllers have the following gains:

- Speed PI controller has gains of P = 65.47 and I = 3134.24.
- Flux PI controller has gains of P = 52.22 and I = 2790.51.
- D-axis PI controller has gains of P = 1.08 and I = 207.58.
- Q-axis PI controller has gains of P = 1.08 and I = 210.02.

The controller gains are stored in a Data Store Memory block and provided externally to each PID block. When the tuning process for a controller is complete, the new tuned gains are written to the Data Store Memory block. This configuration allows you to update your controller gains in real-time during the simulation.

Closed-Loop PID Autotuner Block

The Closed-Loop PID Autotuner block allows you to tune one PID controller at a time. It injects sinusoidal perturbation signals at the plant input and measures the plant output during a closed-loop experiment. When the experiment stops, the block computes PID gains based on the plant frequency responses estimated at a small number of points near the desired bandwidth. For this FOC ASM model, the Closed-Loop PID Autotuner block can be used for each of the four PI controllers.

This workflow applies when you have initial controllers that you want to retune using the Closed-Loop PID Autotuner block. The benefits of this approach are:

1. If there is an unexpected disturbance during the experiment, it is rejected by the existing controller to ensure safe operation.
2. The existing controller keeps the plant running near its nominal operating point by suppressing the perturbation signals.

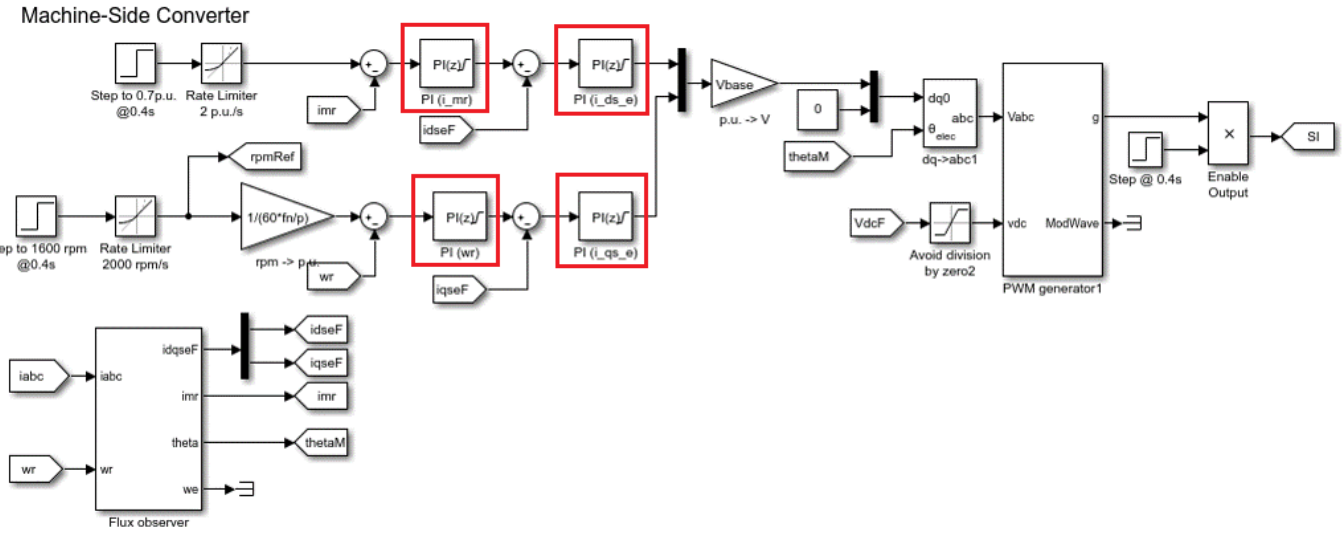
When using the Closed-Loop PID Autotuner block for both simulations and real-time applications:

- The plant must be either asymptotically stable (all the poles are strictly stable) or integrating. The autotuner block does not work with an unstable plant.
- The feedback loop with the existing controller must be stable.
- To estimate plant frequency responses more accurately in real time, minimize the occurrence of any disturbance in the FOC ASM model during the experiment. The autotuner block expects the plant output to be the response to the injected perturbation signals only.
- Because the feedback loop is closed during the experiment, the existing controller suppresses the injected perturbation signals as well. The advantage of using closed-loop experiment is that the controller keeps the plant running near the nominal operating point and maintains safe operation. The disadvantage is that it reduces the accuracy of frequency response estimation if your target bandwidth is far away from the current bandwidth.

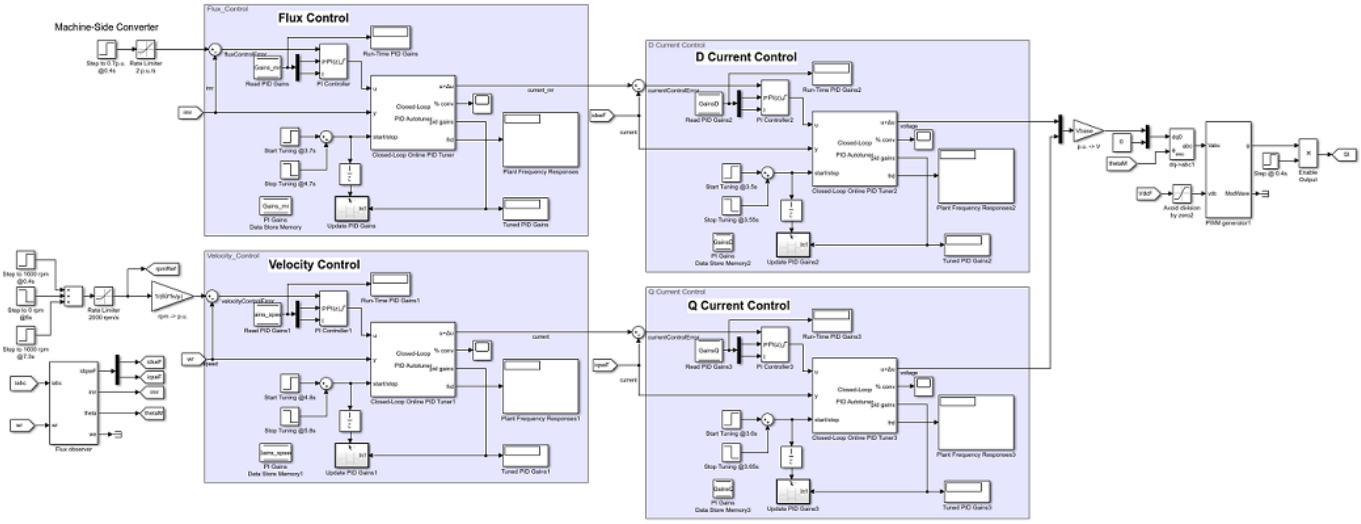
Connect Autotuner with Plant and Controller

Insert the Closed-Loop PID Autotuner block between the PID block and the plant for all four PI controllers, as shown in the FOC ASM model. The `start/stop` signal starts and stops the closed-loop experiment. When no experiment is running, the Closed-Loop PID Autotuner block behaves like a unity gain block, where the u signal directly passes to $u + \Delta u$.

View the original control structure for the machine-side converter with four PI controllers.



To modify the control structure, incorporate the Closed-Loop PID Autotuner Block to each of the PI controllers. View the modified control structure for the machine-side converter.



Configure Autotuner Block

After connecting the Closed-Loop PID Autotuner block with the plant model and PID block, configure the tuning and experiment settings.

On the **Tuning** tab, there are two main tuning settings:

- **Target bandwidth** - Determines how fast you want the controller to respond. In this example, choose `5000` rad/sec for inner-loop current control and `200` rad/sec for outer-loop control.
- **Target phase margin** - Determines how robust you want the controller to be. In this example, choose `70` degrees for inner-loop current control and `90` degrees for outer-loop control.

On the **Experiment** tab, there are three main experiment settings:

- **Plant Type** - Specifies whether the plant is asymptotically stable or integrating. In this example, the FOC ASM model is stable.
- **Plant Sign** - Specifies whether the plant has a positive or negative sign. The plant sign is positive if a positive change in the plant input at the nominal operating point results in a positive change in the plant output when the plant reaches a new steady state. Otherwise, the plant sign is negative. If a plant is stable, the plant sign is equivalent to the sign of its dc gain. If a plant is integrating, the plant sign is positive (or negative) if the plant output keeps increasing (or decreasing). In this example, the FOC ASM model has a positive plant sign.
- **Sine Amplitudes** - Specifies the amplitudes of the injected sine waves. In this example, choose `0.25` for the inner-loop controllers and `0.01` for the outer-loop controllers to ensure the plant is properly excited within the saturation limit. If the excitation amplitude is either too large or too small, it will produce inaccurate frequency response estimation results.

Tuning Cascaded Feedback Loops

Because the Closed-Loop PID Autotuner block only tunes one PI controller at a time, the four controllers must be tuned separately in the FOC ASM model. Tune the inner-loop controllers first, and then tune the outer-loop controllers.

- The d-axis current controller is tuned between 3.5 and 3.55 sec.

- The q-axis current controller is tuned between 3.6 and 3.65 sec.
- The flux controller is tuned between 3.7 and 4.7 sec.
- The speed controller is tuned between 4.8 and 5.8 sec.

After tuning each PI controller, the controller gains are updated through the Data Store Memory block.

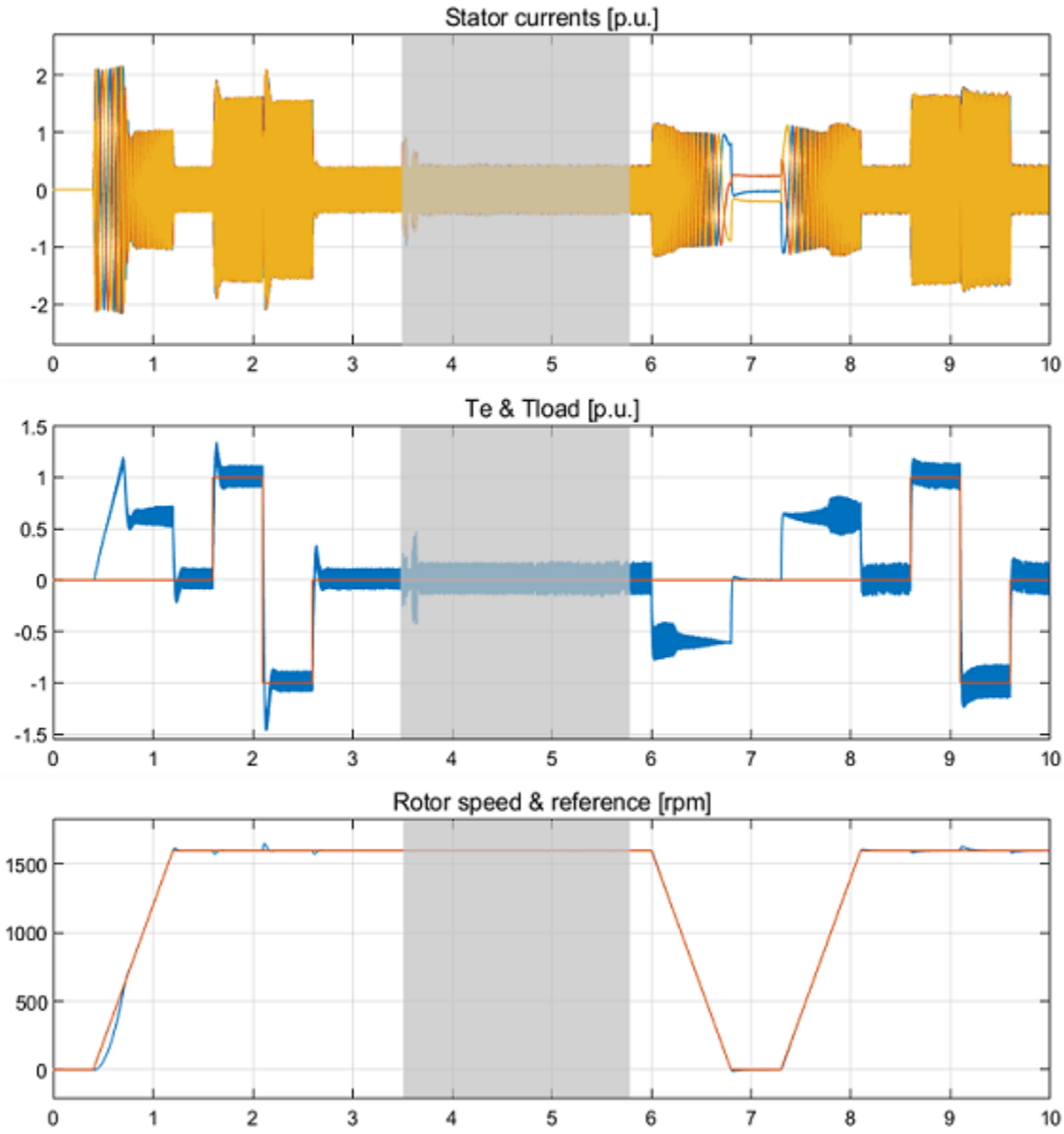
Simulating Autotuner Block in Normal Mode

In this example, the FOC ASM model is built in Simulink. All four controllers are tuned in one simulation. In addition, speed responses are compared before and after tuning the controllers. Scenarios under test include the acceleration process and torque load changes (magnitude of 1 p.u.).

Simulation of the FOC ASM model usually takes a few minutes on your computer due to the small sample time of the power electronics controller of the motor.

```
sim(mdl)
logsout_autotuned = logsout;
save('AutotunedSpeed','logsout_autotuned');
```

The following figure shows the overall simulation result.



The gray area in the previous figure shows the current and speed responses during tuning, from 3.5 to 5.8 seconds. The changes in current and in motor speed are very small. The motor speed reaches the nominal `1600` rpm before the autotuning process begins.

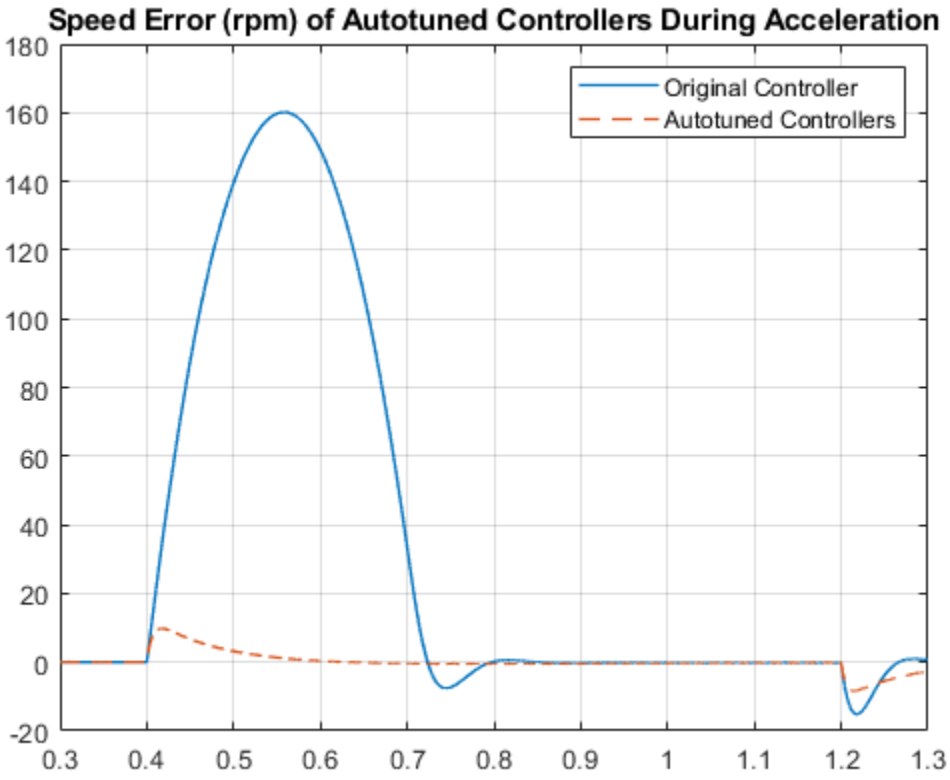
The four PI controllers are tuned with new gains.

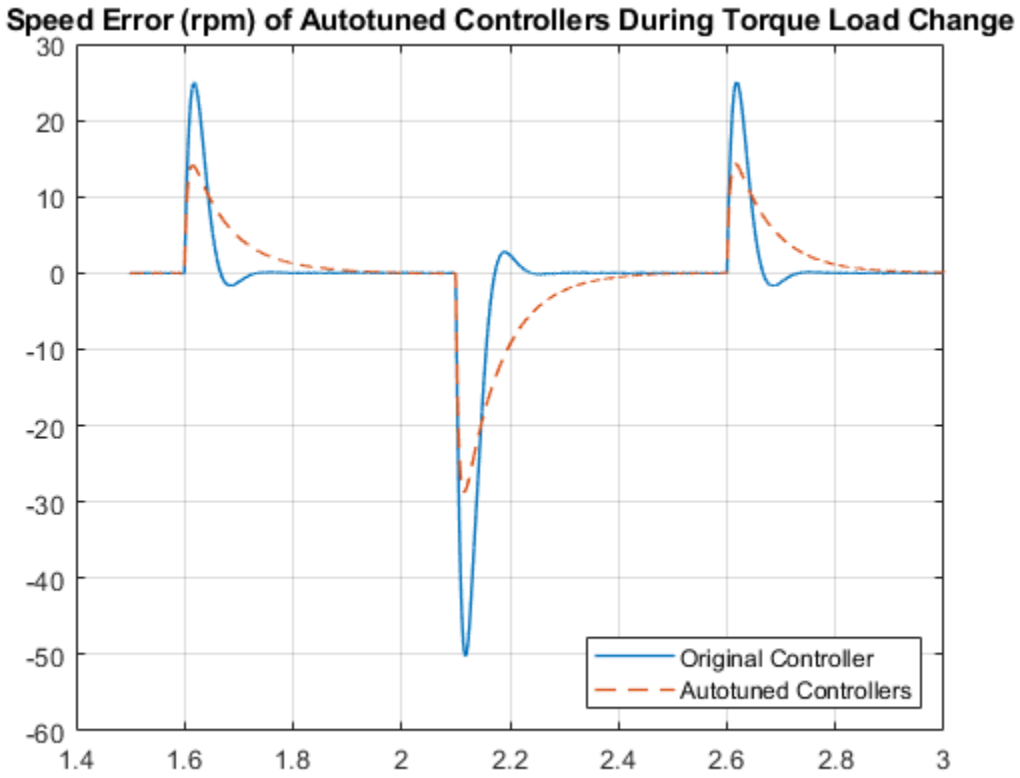
- The speed PI controller has gains of P = 158.8 and I = 2110.
- The flux PI controller has gains of P = 129.3 and I = 1732.
- The d-axis PI controller has gains of P = 1.611 and I = 627.6.
- The q-axis PI controller has gains of P = 2.029 and I = 829.9.

The same rotor speed references and torque loads are applied before and after the autotuning process. Plot rotor speed errors with respect to the nominal `1600` rpm before and after the

controllers are tuned using the Closed-Loop PID Autotuner block. The speed error curves are aligned in time to compare controller performances side-by-side.

```
scdfocasmPIDTuningPlotSpeed
```





After tuning the controllers, the speed response of the asynchronous motor has a faster transient response and smaller steady-state error during acceleration and when torque load changes.

```
bdclose(mdl)
```

See Also

Closed-Loop PID Autotuner

More About

- “How PID Autotuning Works” on page 8-5
- “Tune PID Controller in Real Time Using Open-Loop PID Autotuner Block” on page 8-23
- “Tune Field-Oriented Controllers Using Closed-Loop PID Autotuner Block” on page 8-47

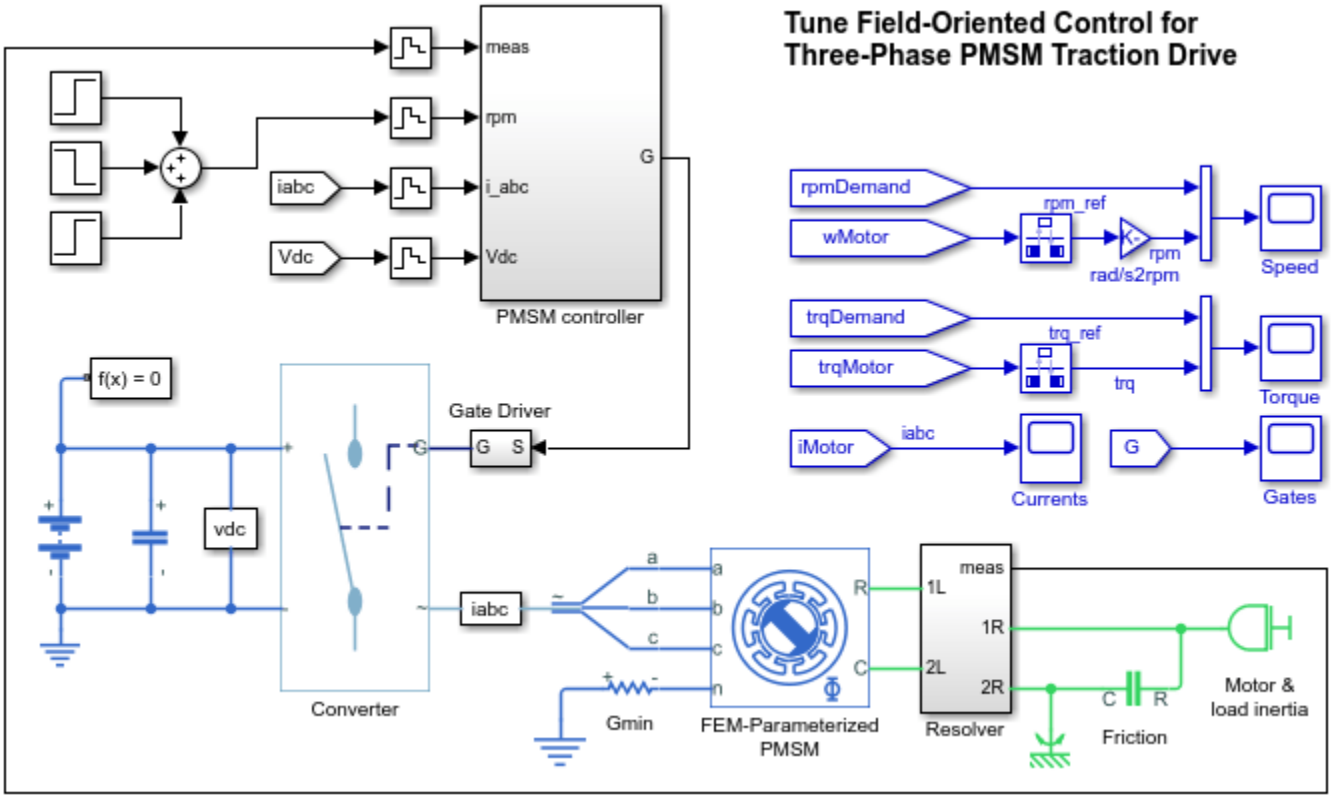
Tune Field-Oriented Controllers for a PMSM Using Closed-Loop PID Autotuner Block

This example shows how to tune a field-oriented controller for a PMSM-based electrical-traction drive in just one simulation using the Closed-Loop PID Autotuner block.

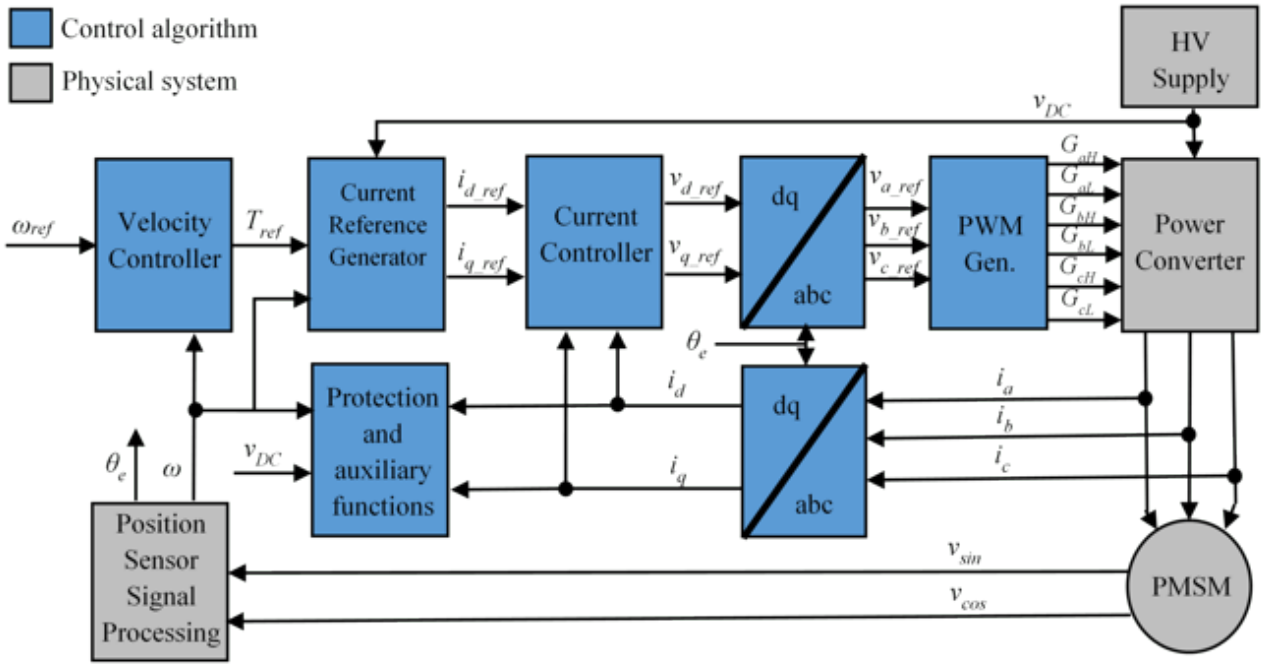
Field-Oriented Control

In this example, a field-oriented controller for a permanent magnet synchronous machine (PMSM) based electrical-traction drive is modeled in Simulink® using Simscape™ Electrical™ components. The model is based on the Simscape example “Three-Phase PMSM Traction Drive” (Simscape Electrical).

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



Field-oriented control (FOC) controls three-phase stator currents as a vector. FOC is based on projections, which transform a three-phase time- and speed-dependent system into a two-coordinate time-invariant system. These transformations are the Clarke Transformation, Park Transformation, and their respective inverse transforms. These transformations are implemented as blocks within the PMSM controller subsystem.



The advantages of using FOC to control AC motors include:

- Torque and flux controlled directly and separately
- Accurate transient and steady-state management
- Similar performance compared to DC motors

The PMSM controller subsystem contains all three PI controllers. The outer-loop PI controller regulates the speed of the motor. The two inner-loop PI controllers control the d-axis and q-axis currents separately. The command from the outer-loop PI controller directly feeds to the q-axis to control torque. The command for the d-axis is zero for PMSM because the rotor flux is fixed with a permanent magnet for this type of AC motor.

The existing PI controllers have the following gains:

- Speed PI controller has gains of P = 20 and I = 500.
- D-axis PI controller has gains of P = 0.8779 and I = 710.3.
- Q-axis PI controller has gains of P = 1.0744 and I = 1061.5.

The controller gains are stored in a Data Store Memory block and provided externally to each PID block. When the tuning process for a controller is complete, the new tuned gains are written to the Data Store Memory block. This configuration allows you to update your controller gains in real-time during the simulation.

Closed-Loop PID Autotuner Block

The Closed-Loop PID Autotuner block allows you to tune one PID controller at a time. It injects sinusoidal perturbation signals at the plant input and measures the plant output during a closed-loop experiment. When the experiment stops, the block computes PID gains based on the plant frequency responses estimated at a small number of points near the desired bandwidth. For this PMSM-based

electrical-traction drive model, the Closed-Loop PID Autotuner block can be used for each of the three PI controllers.

This workflow applies when you have initial controllers that you want to retune using the Closed-Loop PID Autotuner block. The benefits of this approach are:

1. If there is an unexpected disturbance during the experiment, it will be rejected by the existing controllers to ensure safe operation.
2. The existing controllers will keep the plant running near its nominal operating point by suppressing the perturbation signals.

When using the Closed-Loop PID Autotuner block for both simulations and real-time applications:

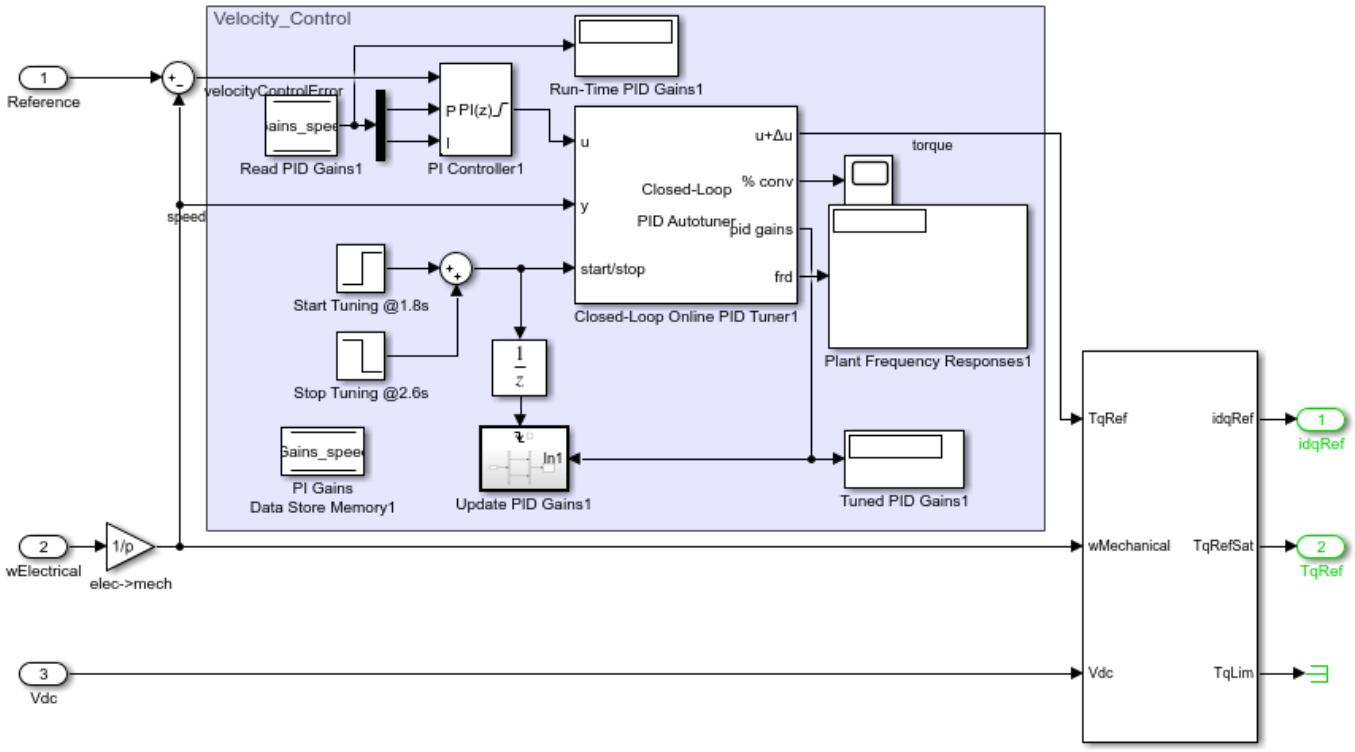
- The plant must be either asymptotically stable (all the poles are strictly stable) or integrating. The autotuner block does not work with an unstable plant.
- The feedback loop with the existing controller must be stable.
- To estimate plant frequency responses more accurately in real time, minimize the occurrence of any disturbance in the PMSM based electrical-traction drive model during the experiment. The autotuner block expects the plant output to be the response to the injected perturbation signals only.
- Because the feedback loop is closed during the experiment, the existing controller suppresses the injected perturbation signals as well. The advantage of using closed-loop experiment is that the controller keeps the plant running near the nominal operating point and maintains safe operation. The disadvantage is that it reduces the accuracy of frequency response estimation if your target bandwidth is far away from the current bandwidth.

Connect Autotuner with Plant and Controller

Insert the Closed-Loop PID Autotuner block between the PID block and the plant for all three PI controllers, as shown in the PMSM-based electrical-traction drive model. The `start/stop` signal starts and stops the closed-loop experiment. When no experiment is running, the Closed-Loop PID Autotuner block behaves like a unity gain block, where the u signal directly passes to $u + \Delta u$.

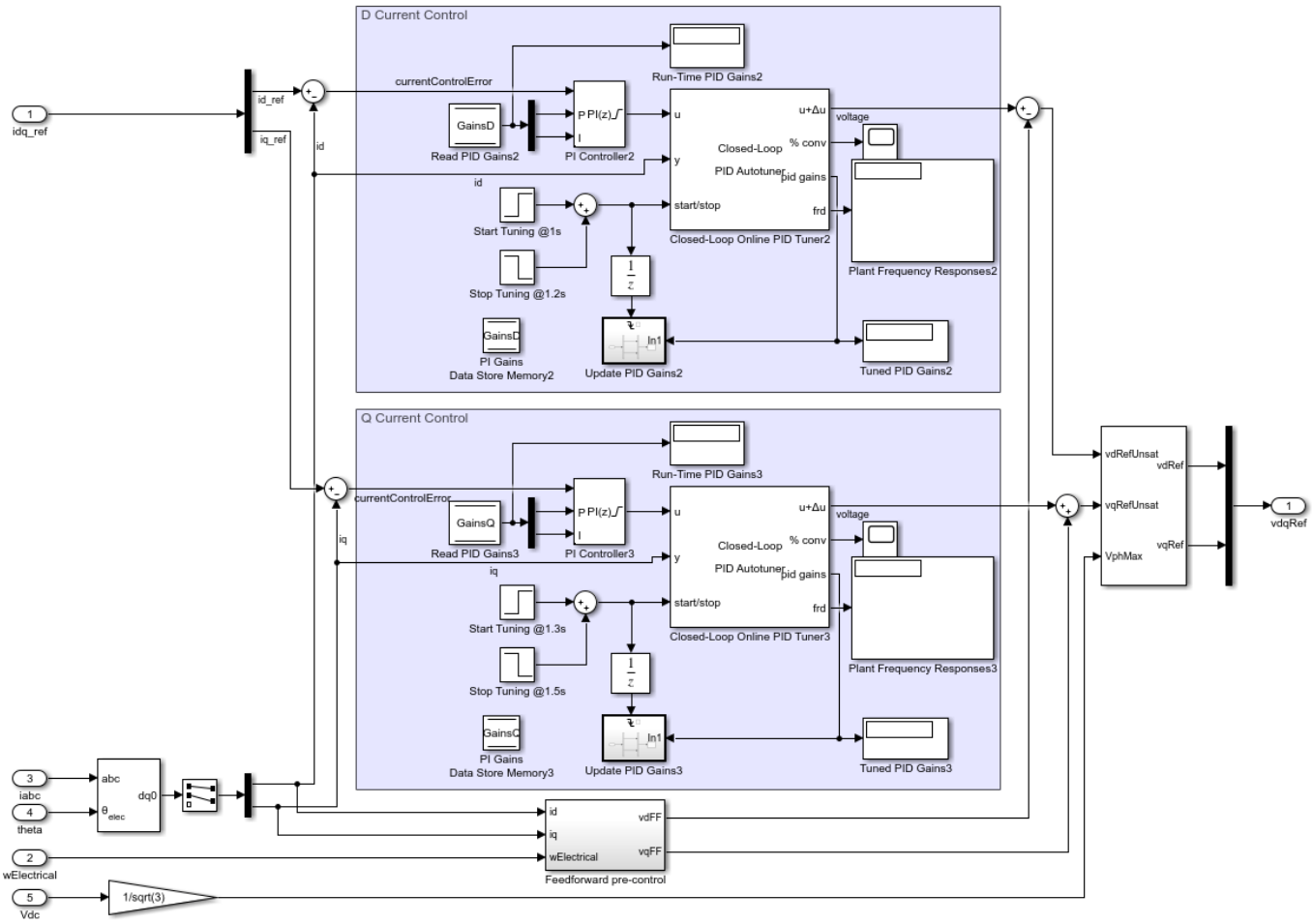
To view the modified outer-loop control structure, open the Outer loop control subsystem in the PMSM controller subsystem.

```
controlSubsystem = [mdl '/PMSM controller'];
open_system([controlSubsystem '/Outer loop control'])
```



View the modified current controllers in the Inner loop control subsystem.

```
open_system([controlSubsystem '/Inner loop control'])
```



Configure Autotuner Block

After connecting the Closed-Loop PID Autotuner block with the plant model and PID block, configure the tuning and experiment settings.

On the **Tuning** tab, there are two main tuning settings:

- **Target bandwidth** - Determines how fast you want the controller to respond. In this example, choose `300` rad/s for speed control, `2500` rad/s for d-axis current control and `2200` rad/s for q-axis current control.
- **Target phase margin** - Determines how robust you want the controller to be. In this example, choose `60` degrees for inner-loop current control and `70` degrees for outer-loop control.

On the **Experiment** tab, there are three main experiment settings:

- **Plant Type** - Specifies whether the plant is asymptotically stable or integrating. In this example, the PMSM-based electrical-traction drive model is stable.
- **Plant Sign** - Specifies whether the plant has a positive or negative sign. The plant sign is positive if a positive change in the plant input at the nominal operating point results in a positive change in the plant output when the plant reaches a new steady state. Otherwise, the plant sign is negative. If a plant is stable, the plant sign is equivalent to the sign of its dc gain. If a plant is integrating,

the plant sign is positive (or negative) if the plant output keeps increasing (or decreasing). In this example, the PMSM-based electrical-traction drive model has a positive plant sign.

- **Sine Amplitudes** - Specifies the amplitudes of the injected sine waves. In this example, choose `5` for the inner-loop controllers and `5` for the outer-loop controller to ensure the plant is properly excited within the saturation limit. If the excitation amplitude is either too large or too small, it will produce inaccurate frequency response estimation results.

Tuning Cascaded Feedback Loops

Because the Closed-Loop PID Autotuner block only tunes one PI controller at a time, the three controllers must be tuned separately in the PMSM-based electrical-traction drive model. Tune the inner-loop controllers first, and then tune the outer-loop controller.

- The d-axis current controller is tuned between 1.0 and 1.2 sec.
- The q-axis current controller is tuned between 1.3 and 1.5 sec.
- The speed controller is tuned between 1.8 and 2.6 sec.

After tuning each PI controller, the controller gains are updated through the Data Store Memory block.

Simulating Autotuner Block in Normal Mode

In this example, the PMSM-based electrical-traction drive model is built in Simulink. All three PI controllers are tuned in one simulation. In addition, speed responses are compared before and after tuning the controllers during acceleration processes.

Simulation of the PMSM-based electrical-traction drive model usually takes a few minutes on your computer due to the small sample time of the power electronics controller of the motor.

```
sim(mdl);
save('AutotunedSpeed','SpeedData');
```

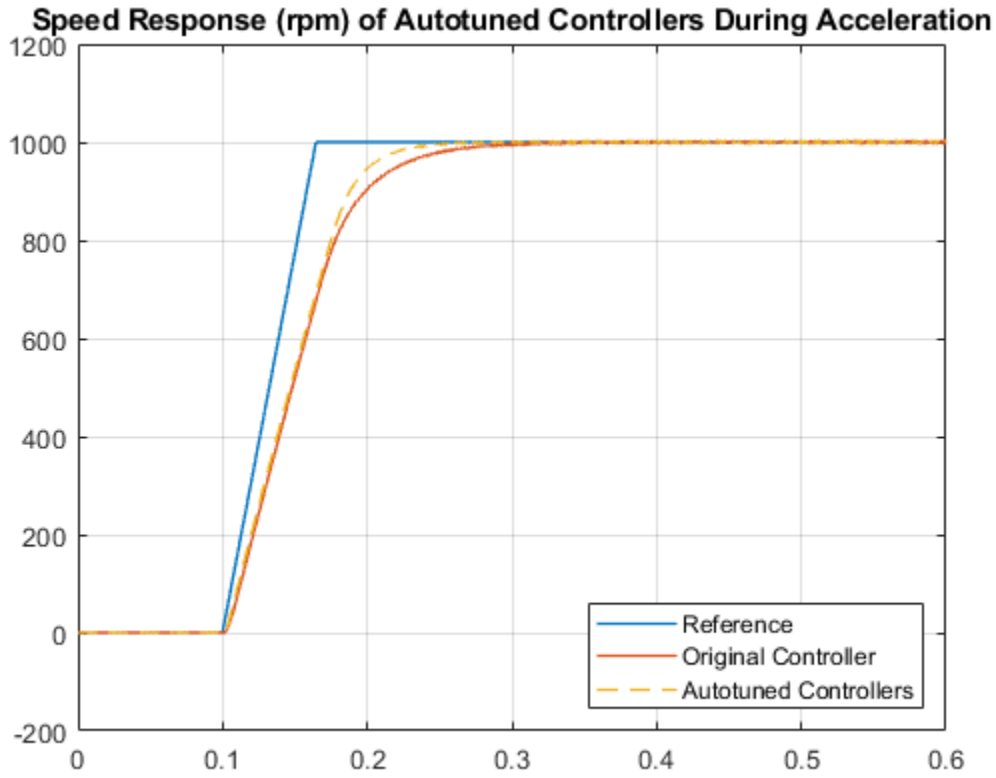
During the autotuning process from 1.0 to 2.6 seconds, the changes in current and in motor speed are very small. The motor speed reaches the nominal `1000` rpm before the autotuning process begins.

The three PI controllers are tuned with new gains.

- The d-axis PI controller has gains of P = 0.6332 and I = 331.7.
- The q-axis PI controller has gains of P = 0.937 and I = 351.1.
- The speed PI controller has gains of P = 32.35 and I = 1322.

The same rotor speed reference is applied before and after the autotuning process. Plot the rotor speed responses with respect to the nominal `1000` rpm before and after tuning the controllers. The speed response curves are aligned in time to compare controller performances side-by-side.

```
scdfocpmsmPIDTuningPlotSpeed
```



After tuning the controllers, the speed response of the PMSM has a faster transient response during acceleration.

```
bdclose(mdl)
```

See Also

Closed-Loop PID Autotuner

More About

- “How PID Autotuning Works” on page 8-5
- “Tune PID Controller in Real Time Using Open-Loop PID Autotuner Block” on page 8-23
- “Tune Field-Oriented Controllers for an Asynchronous Machine Using Closed-Loop PID Autotuner Block” on page 8-54
- “Tune Field-Oriented Controllers Using SYSTUNE” on page 7-160

Design PID Controllers for Three-Phase Rectifier using Closed-Loop PID Autotuner Block

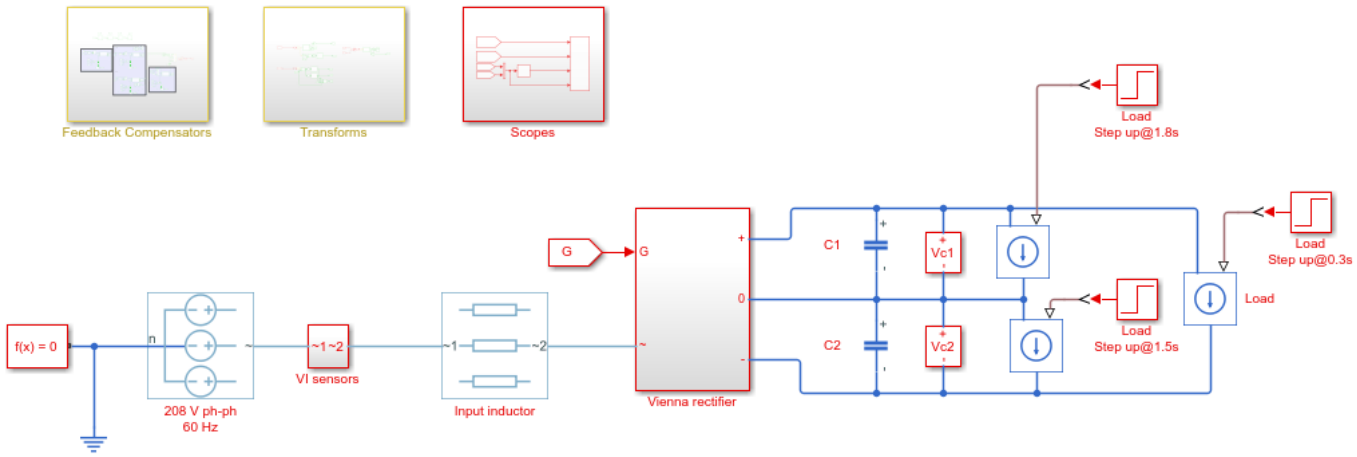
This example shows how to use the Closed-Loop PID Autotuner block to tune the DC-link voltage, DQ axis current, and voltage neutral controllers for a Vienna-rectifier-based power factor corrector.

Power Factor Correction Model

This example uses the power factor correction circuit described in “Vienna Rectifier Control” (Simscape Electrical). Power factor correction preconverters correct the power factor of loads, which increases the energy efficiency of the distribution system. This correction is useful when nonlinear impedances, such as switched-mode power supplies, are connected to the AC grid.

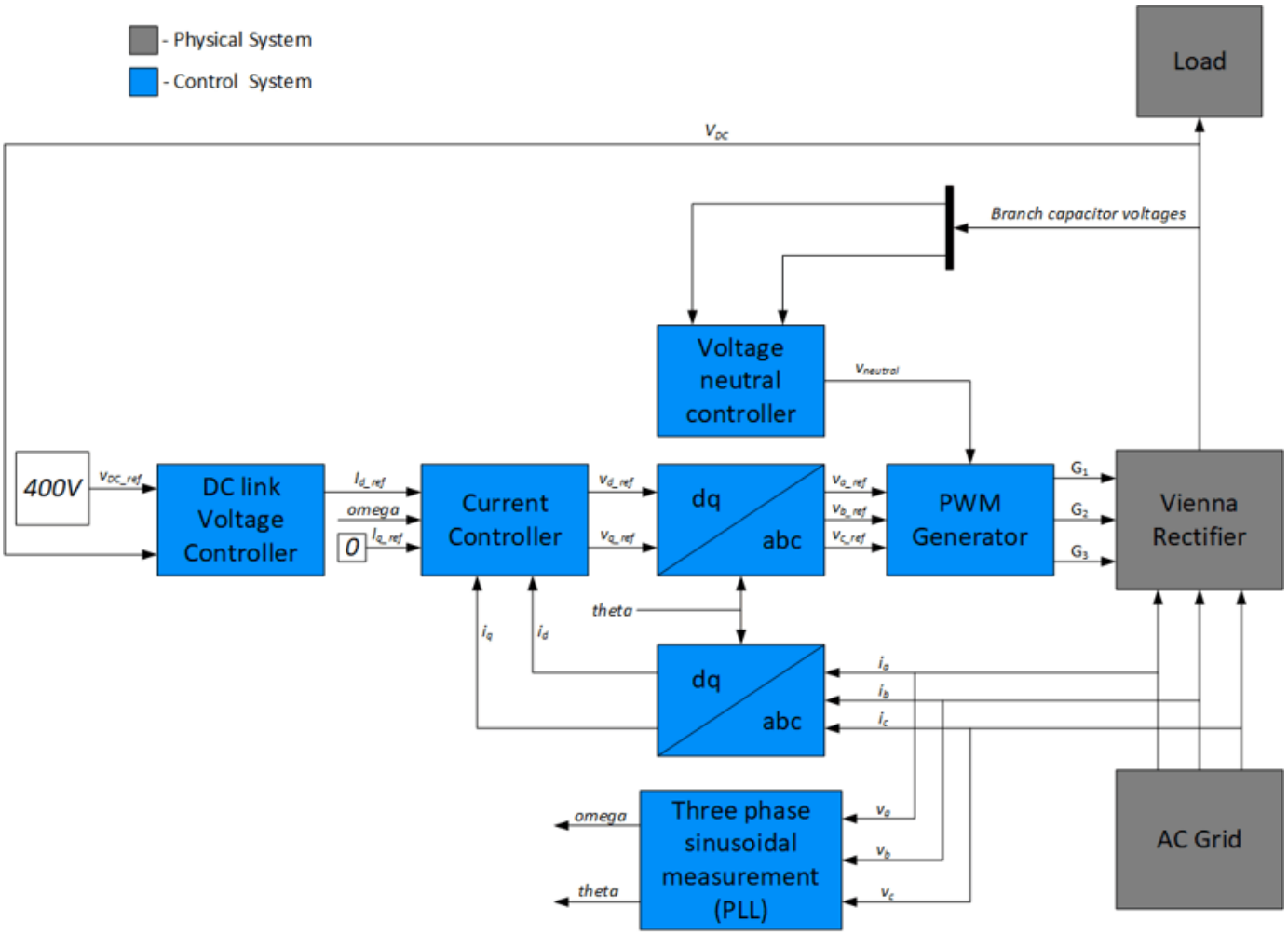
This model uses a Vienna rectifier and a switched-mode power supply to convert a three-phase 120V AC supply to a regulated 400V DC supply. To ensure that the device on-resistances are correctly represented, the semiconductor components are modeled using MOSFETs rather than ideal switches. The model simulation is configured to run in accelerator mode using the partitioning solver.

```
open_system('PWM_Rectifier_Vienna_SC')
```



DQ-Axis Current Control

For this example, the DQ-axis controller for the Vienna rectifier is modeled as shown in the following diagram.



In DQ-axis control, the time-dependent, three-phase currents are transformed into a time-invariant, two-coordinate vector using projections. These transformations are the Clarke Transformation, the Park Transformation, and their respective inverse transformations. These transformations are implemented as blocks within the Measurements subsystem. To maintain a power factor close to 1, the reactive power being drawn from the grid should be close to zero. Therefore, commanding a zero Q-axis current from the controller allows the power factor to be close to 1.

In the model, the controllers have the following gains:

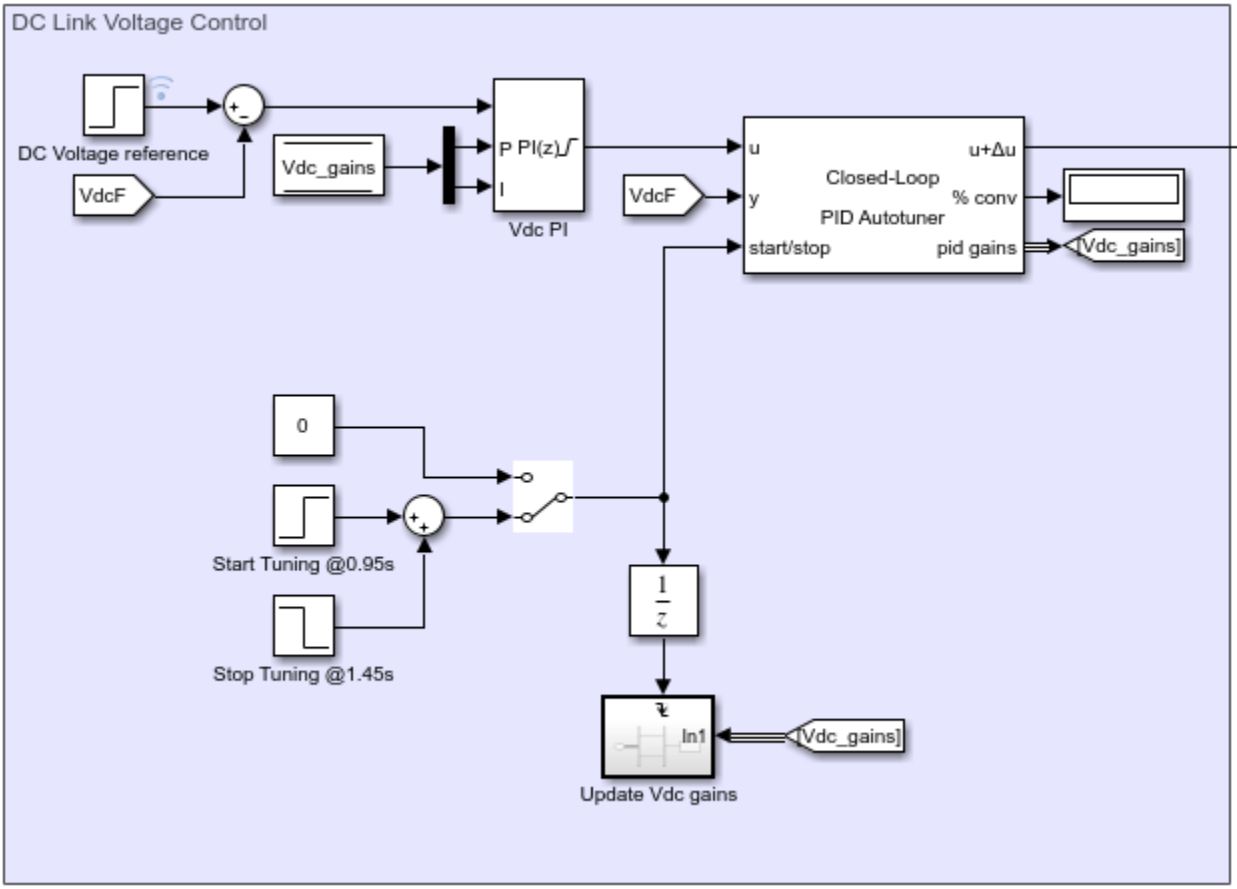
- DC-link voltage PI controller: P = 2 and I = 20
- Both DQ-axis current PI controllers: P = 5 and I = 500
- Voltage neutral P controller: P = 0.001

The controller gains are stored in a Data Store Memory block and provided externally to each PID block. When the tuning process for a controller is complete, the new tuned gains are written to the Data Store Memory block. This configuration allows you to update your controller gains in real-time during the simulation.

For this example, you retune these controllers using Closed-Loop PID Autotuner blocks.

Closed-Loop PID Autotuner Block

The Closed-Loop PID Autotuner block allows you to tune one PID controller at a time. It injects sinusoidal perturbation signals at the plant input and measures the resulting plant output during a closed-loop experiment. When the experiment stops, the block computes PID gains based on the plant frequency responses estimated at a small number of points near the desired bandwidth. For this Vienna rectifier model, the Closed-Loop PID Autotuner block can be used for each of the controllers, as shown for the DC link voltage loop below.



This workflow applies when you have initial controllers that you want to retune using the Closed-Loop PID Autotuner block. The benefits of this approach are:

1. If there is an unexpected disturbance during the experiment, it is rejected by the existing controller to ensure safe operation.
2. The existing controller keeps the plant running near its nominal operating point by suppressing the perturbation signals.

When using the Closed-Loop PID Autotuner block for both simulations and real-time applications:

- The plant must be either asymptotically stable (all the poles are strictly stable) or integrating. The autotuner block does not work with an unstable plant.
- The feedback loop with the existing controller must be stable.

- To estimate plant frequency responses more accurately in real time, minimize the occurrence of any disturbance in the Vienna rectifier model during the experiment. The autotuner block expects the plant output to be the response to the injected perturbation signals only.
- Because the feedback loop is closed during the experiment, the existing controller suppresses the injected perturbation signals as well, which reduces the accuracy of frequency response estimation when your target bandwidth is far away from the current bandwidth.

Tuning Cascaded Feedback Loops

Since the Closed-Loop PID Autotuner block only tunes one PID controller at a time, the four controllers must be tuned separately in the model. Therefore, you tune the inner current controllers first, followed by the DC-link voltage controller, and then the voltage neutral controller.

During the model simulation:

- The D-axis current controller is tuned between 0.65 and 0.75 sec.
- The Q-axis current controller is tuned between 0.8 and 0.9 sec.
- The DC-link voltage controller is tuned between 0.95 and 1.45 sec.
- The voltage neutral controller is tuned between 1.7 and 1.72 sec.

After tuning each of the controllers, the controller gains are updated through the Data Store Memory block.

Configure Autotuner Block

After connecting the Closed-Loop PID Autotuner blocks with the plant and the PID blocks, configure the tuning and experiment settings for each of them. On the **Tuning** tab, there are two main tuning settings:

- **Target bandwidth** - Determines how fast you want the controller to respond. In this example, choose 3000 rad/sec for the current control, 400 rad/s for the DC-link voltage control, and 20000 rad/s for the voltage neutral control.
- **Target Phase Margin** - Determines how robust you want the controller to be. In this example, choose 60 degrees for all the controllers.

On the **Experiment** tab, there are three main experiment settings:

- **Plant Type** - Specifies whether the plant is asymptotically stable or integrating. In this example, the Vienna rectifier model is stable.
- **Plant Sign** - Specifies whether the plant has a positive or negative sign. The plant sign is positive if a positive change in the plant input at the nominal operating point results in a positive change in the plant output when the plant reaches a new steady state. Otherwise, the plant sign is negative. If a plant is stable, the plant sign is equivalent to the sign of its DC gain. If a plant is integrating, the plant sign is positive (or negative) if the plant output keeps increasing (or decreasing). In this example, the Vienna rectifier model has a positive plant sign.
- **Sine Amplitudes** - Specifies the amplitudes of the injected sine waves. In this example, to ensure that the plant is properly excited within the saturation limit, choose `0.6` for the D-axis controller, `0.19` for the Q-axis controller, `1` for the DC-link voltage controller, and `0.01` for the voltage neutral controller. If the excitation amplitude is either too large or too small, it will produce inaccurate frequency response estimation results for these experiments.

Simulate Autotuner Block in Accelerator Mode

In this example, the Vienna rectifier model is run in accelerator mode and all four controllers are tuned in one simulation. The simulation of the model usually takes a few minutes due to the small sample time of the power electronics controller.

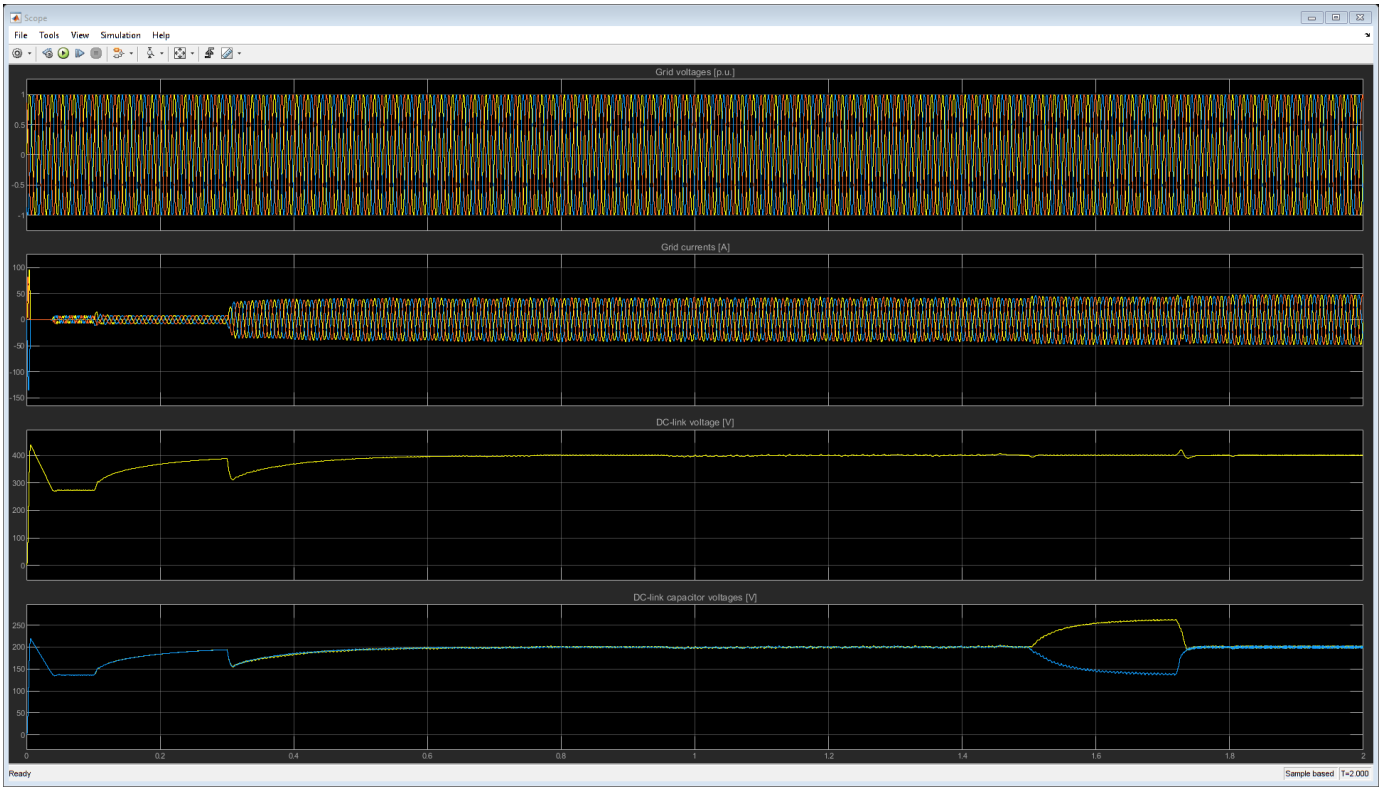
To tune the controllers, simulate the model.

```
sim('PWM_Rectifier_Vienna_SC')
```

```
ans =

  Simulink.SimulationOutput:
                logsout: [1x1 Simulink.SimulationData.Dataset]
                   tout: [4000001x1 double]

     SimulationMetadata: [1x1 Simulink.SimulationMetadata]
           ErrorMessage: [0x0 char]
```

The graph below shows the DC-link voltage profile during the current and voltage controller tuning from 0.65 to 1.45 seconds. It also shows the introduction of an unbalanced load at 1.5 seconds and the subsequent voltage neutral controller tuning at 1.7 seconds.

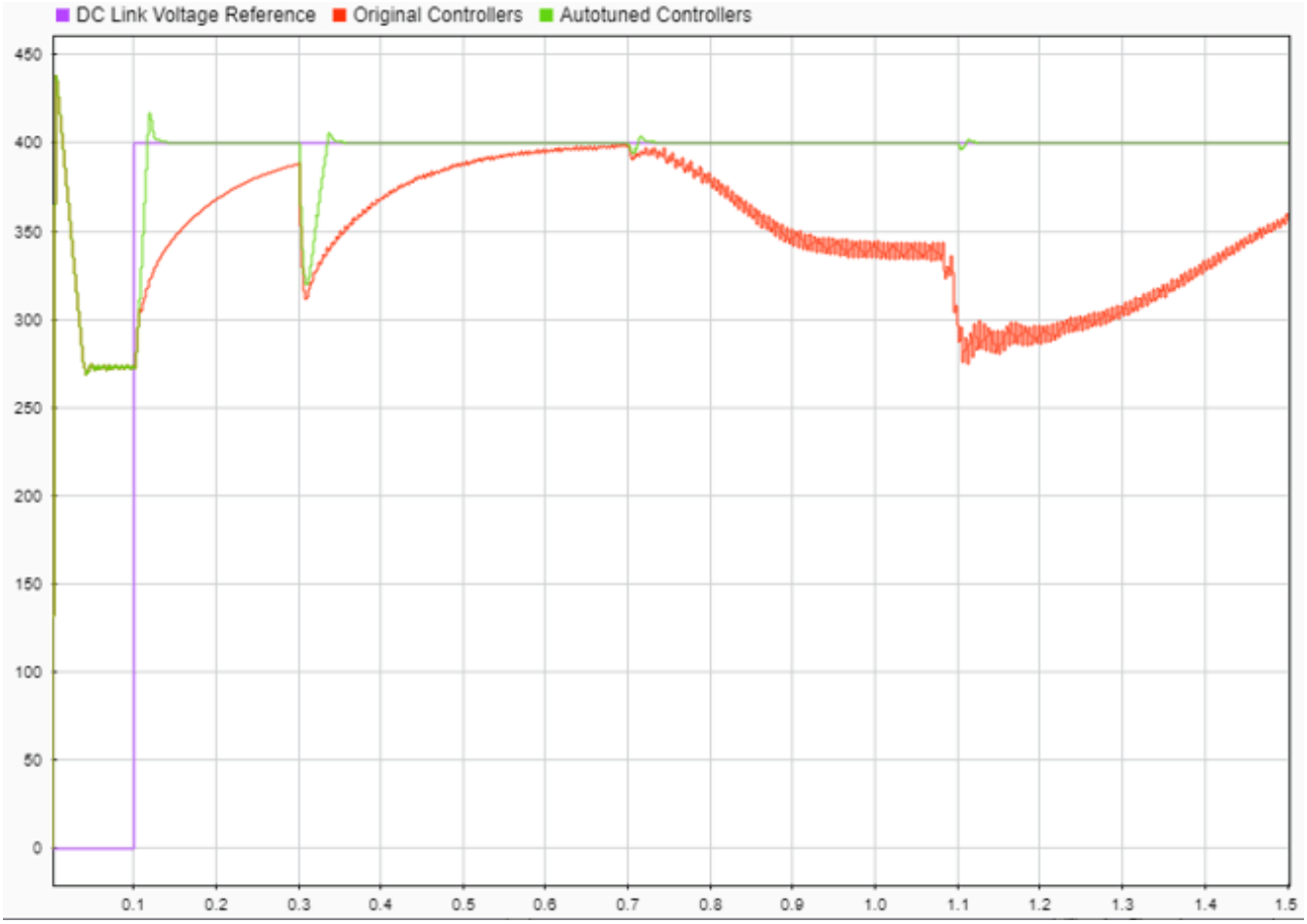
```
open_system('PWM_Rectifier_Vienna_SC/Scopes/Scope')
```



The four controllers are tuned with the new gains.

- DC-link voltage PI controller: P = 0.7386 and I = 135.6
- D-axis current PI controllers: P = 8.407 and I = 1127
- Q-axis current PI controllers: P = 11.91 and I = 3706
- Voltage neutral P controller: P = 6.628

The graph below shows the DC-link voltage response in comparison to the reference before and after tuning the controllers. The original controller (red) is unable to maintain the DC-link voltage after the introduction of unbalanced loads at 0.7 and 1.1 seconds. On the other hand, the autotuned controller decreases the rise time with minimal overshoot and a good settling time to the steady-state value.



See Also

Closed-Loop PID Autotuner

More About

- “How PID Autotuning Works” on page 8-5
- “Tune Field-Oriented Controllers for an Asynchronous Machine Using Closed-Loop PID Autotuner Block” on page 8-54

PID Autotuning for UAV Quadcopter

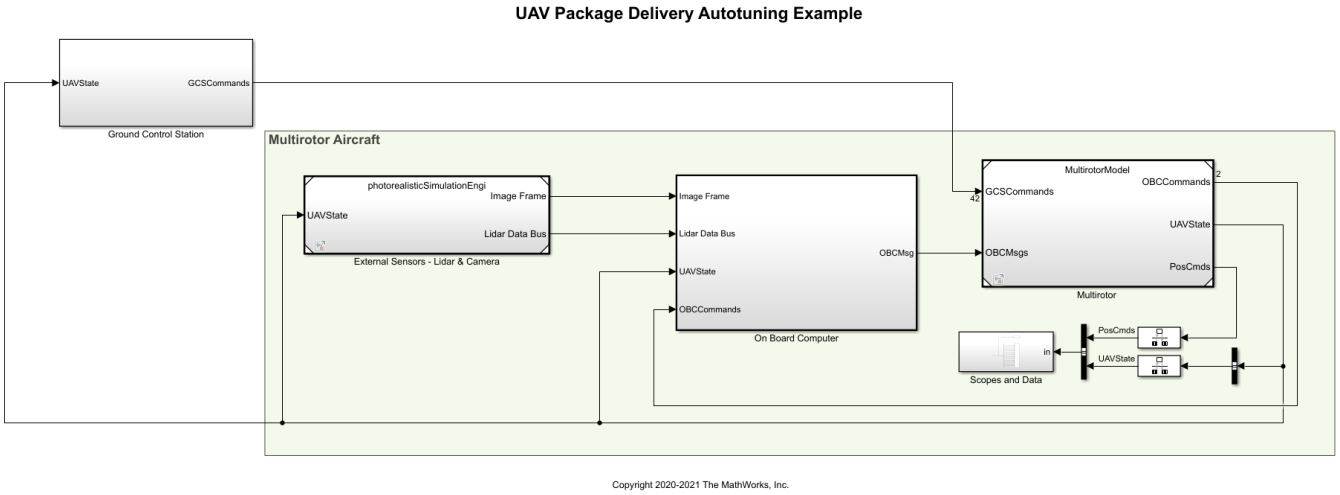
This example shows how to tune PID Controllers used in the attitude and position control of a small quadcopter in only one simulation using the Closed-Loop PID Autotuner block.

UAV Package Delivery Model

In this example, a multirotor is modeled in Simulink® using UAV Toolbox components. This model is based on the UAV Toolbox `uavPackageDelivery` model. For more information, see “UAV Package Delivery” (UAV Toolbox).

To get started, set up and open the Simulink Project.

```
run('scdUAVPIDAutotuningStart.m')
```



Model Architecture and Conventions

The top model consists of the following subsystems and model references:

1. **Ground Control Station** — Used to control and monitor the aircraft while in flight.
2. **External Sensors - Lidar & Camera** — Used to connect to a previously-designed scenario or a photorealistic simulation environment. These produce Lidar readings from the environment as the aircraft flies through it.
3. **On Board Computer** — Used to implement algorithms meant to run in an onboard computer independent from the Autopilot.
4. **Multirotor** — Includes a low-fidelity and mid-fidelity mode, as well as a flight controller including its guidance logic.

The model's design data is contained in a Simulink data dictionary in the **data** folder (`uavPackageDeliveryDataDict.sldd`). Additionally, the model uses “Variant Subsystems” to manage different configurations of the model. Variables placed in the base workspace configure these variants without the need to modify the data dictionary.

PID Controller Autotuning

This example uses the Closed-Loop PID Autotuner block from Simulink Control Design™ software to tune eight controllers used in the attitude and position control of a multirotor. You can use many ways to tune controllers, including manual tuning and empirical calculations. By using the Closed-Loop PID Autotuner, you can set up the control system ahead of time and then perform tuning of all eight loops with a one-click process. This makes the entire tuning process repeatable and easily adjustable for future tuning. In this example, you use a six degree-of-freedom model of a multirotor in Simulink. However, you can also use the Closed-Loop PID Autotuner on hardware to perform the same process using a real multirotor. Most other tuning techniques are difficult to implement on actual multirotors, can take a long time, and are not easily repeatable.

By using the Closed-Loop PID Autotuner for tuning the controllers in this example you do not need to have advanced knowledge of control tuning techniques.

Modify UAV Package Delivery Model for PID Autotuning

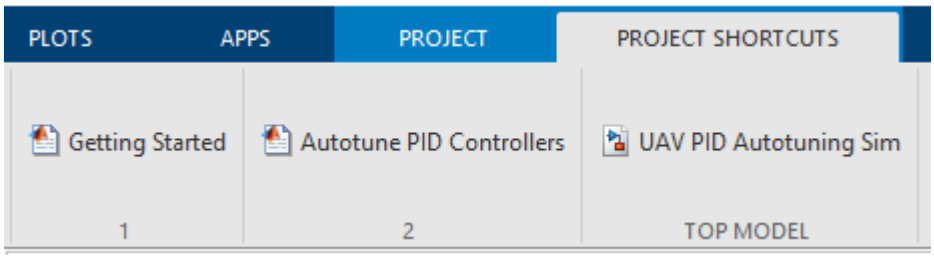
To facilitate PID autotuning, the original UAV package delivery model is modified with these changes:

- Hover mode added to the Full Guidance Logic subsystem
- Third Mission variant added to the Ground Control Station subsystem
- Four Closed-Loop PID Autotuner blocks added to the Attitude Controller and Position Controller subsystems in the High Fidelity Model
- PID Controllers in the Attitude Controller and Position Controller subsystems Controller Parameters **Source** changed from `internal` to `external`
- Data Store Memory blocks added to the Multirotor subsystem
- Default controller gains changed
- To Workspace added to root level model

These changes allow for the multirotor to take off and remain at a fixed altitude while autotuning takes place and to update the gains of the PID Controllers, all in a single simulation.

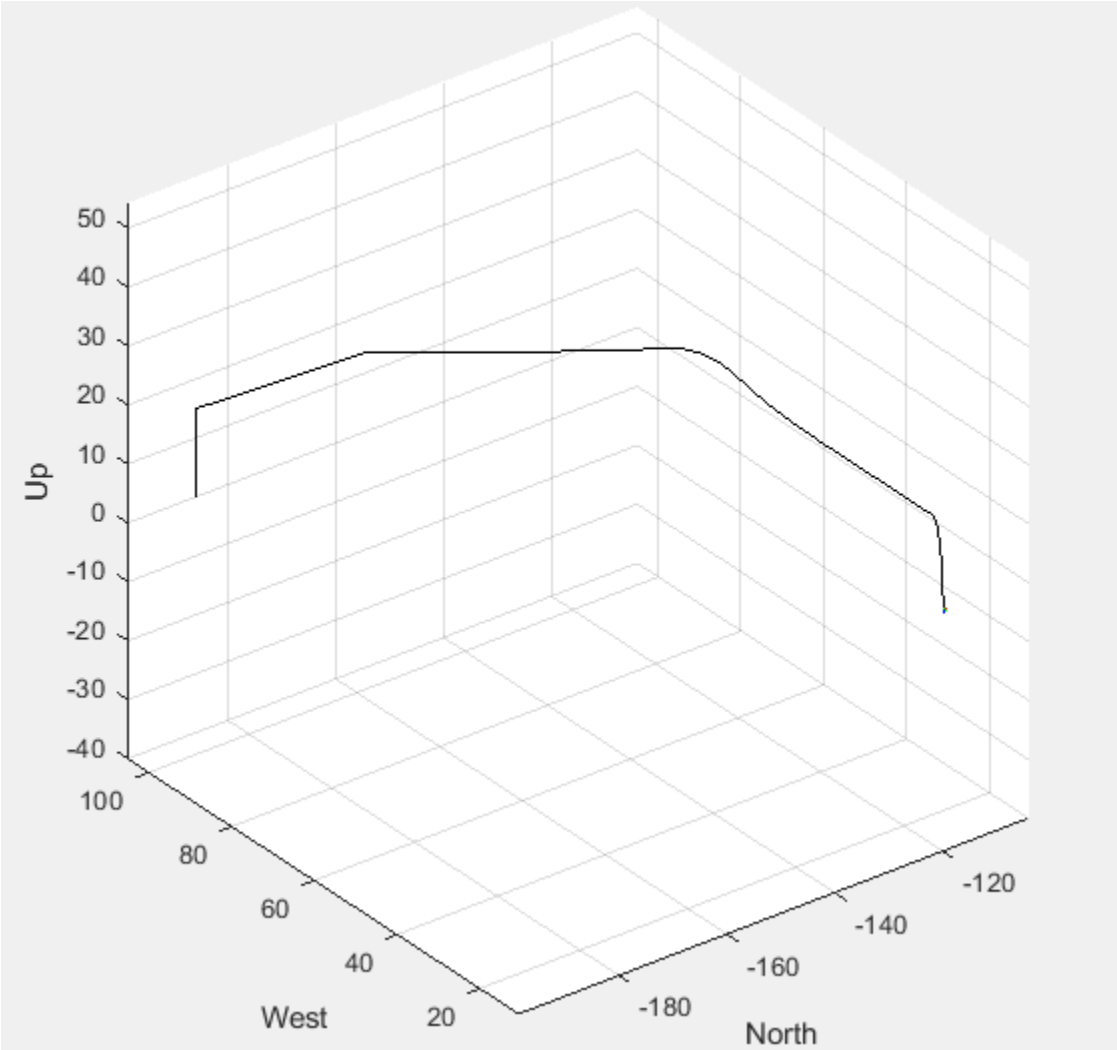
Following Example Steps

Use the **Project Shortcuts** to step through the example. Each shortcut sets up the required variables for the project.

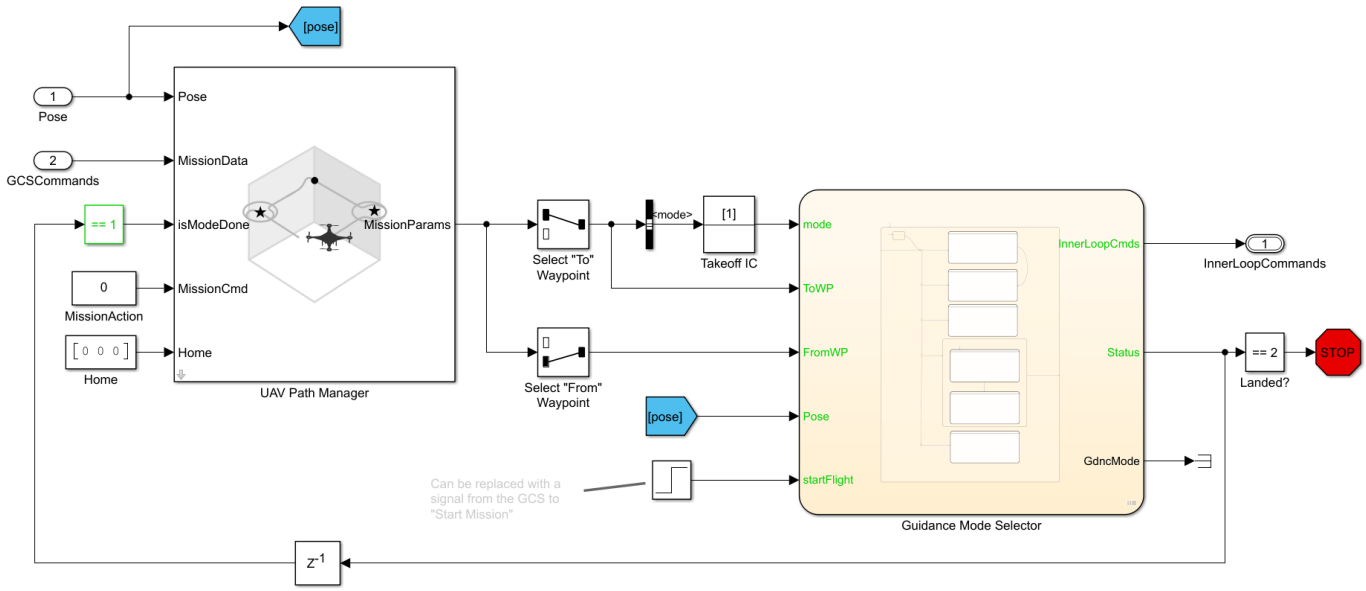


Getting Started

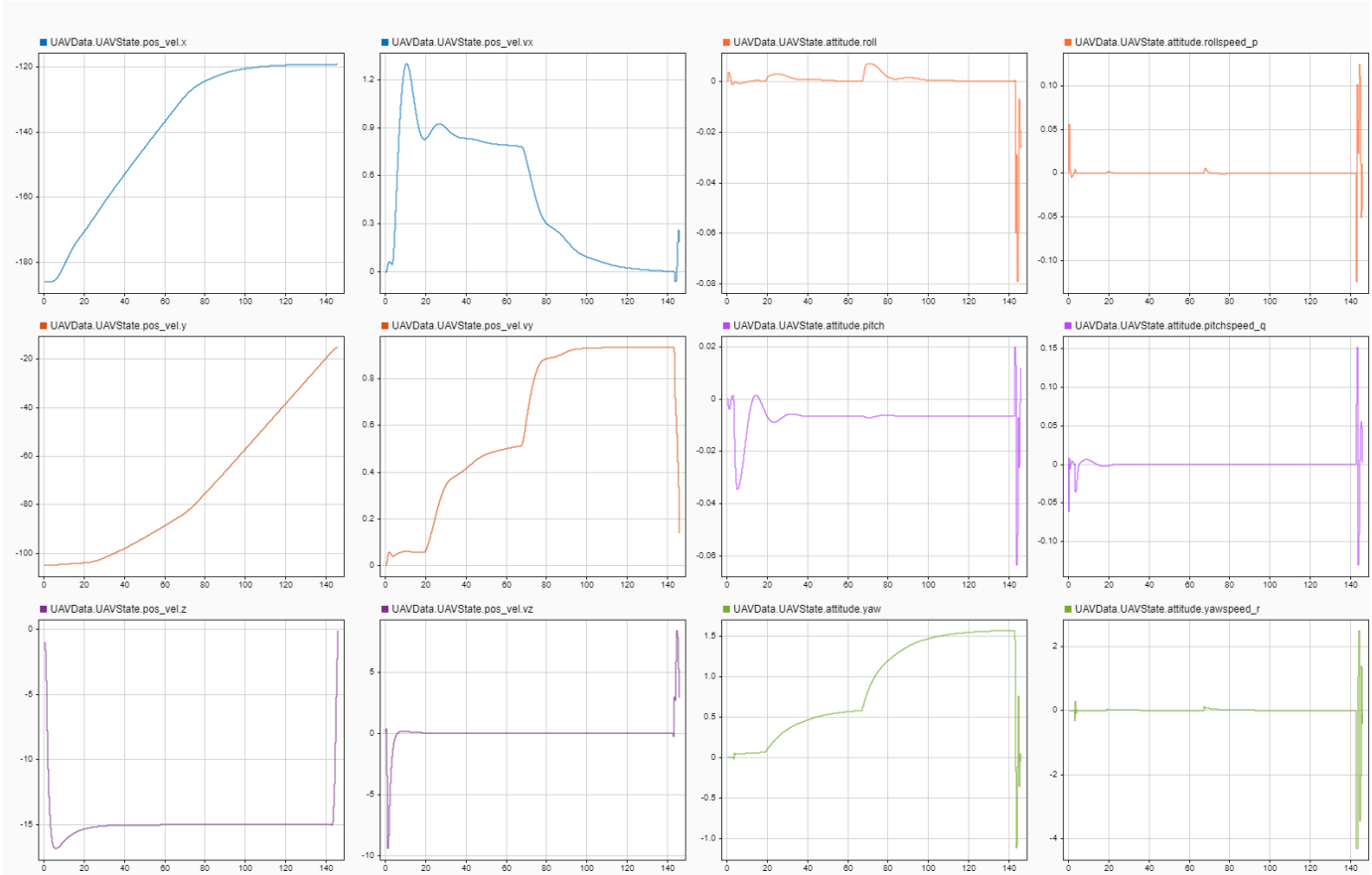
Click the **Getting Started** project shortcut, which sets up the model for a four-waypoint mission using a high-fidelity multirotor plant model. **Run** the `uavPIDAutotuning` model, which shows the multirotor takeoff, fly, and land in a 3-D plot.



The model uses UAV Path Manager block to determine which is the active waypoint throughout the flight. The active waypoint is passed into the `Guidance Mode Selector` Stateflow® chart to generate the necessary inner loop control commands.



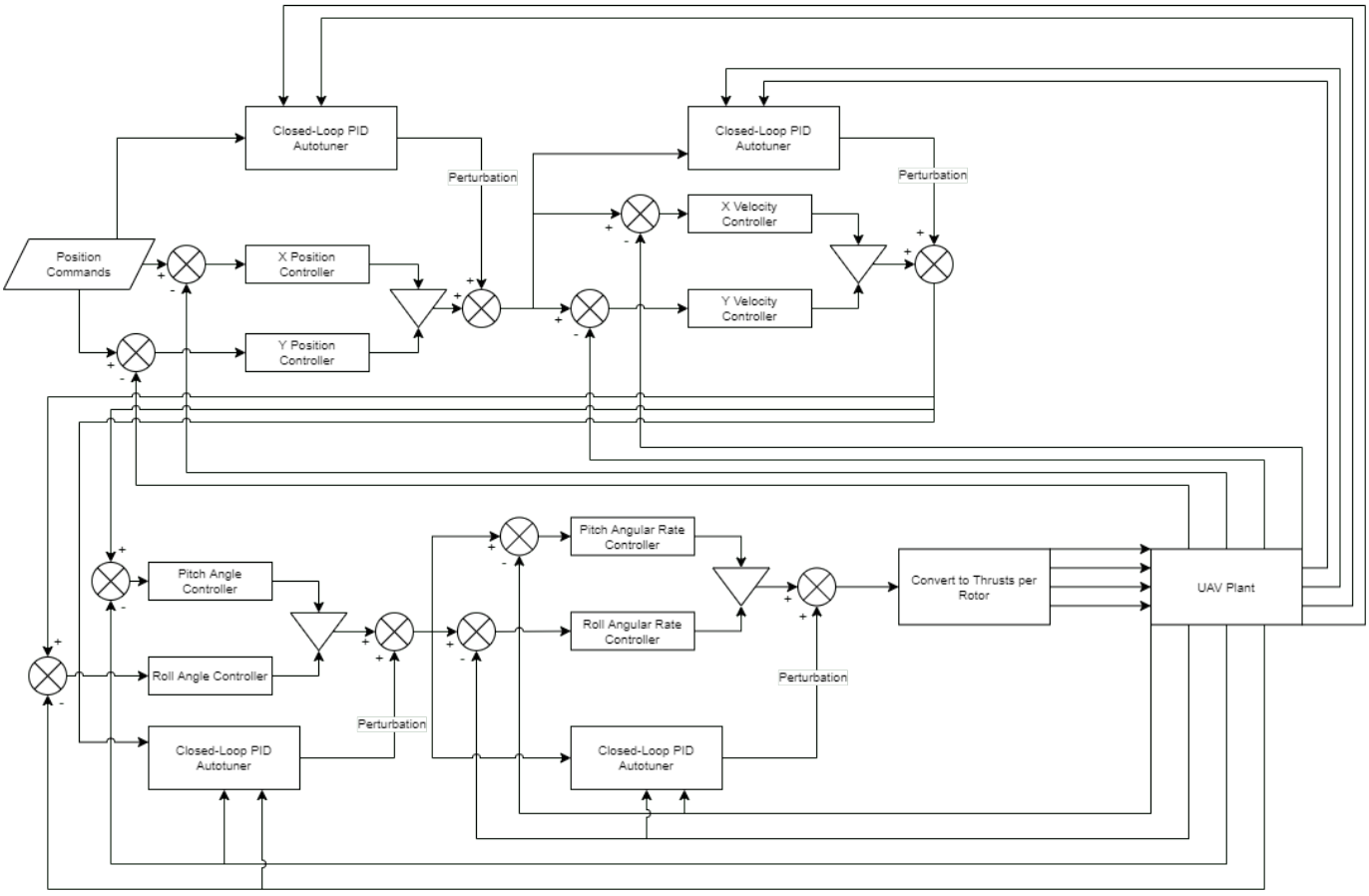
Use the Simulation Data Inspector to visualize the **UAVState** output of the multirotor model.



You can see that the multirotor takes almost 150 seconds to complete the four waypoint path with the baseline set of gains. In order to improve this performance, retune the PID Controllers used in the multirotor.

Run Autotuning Mission

Once you are able to fly a basic mission, you are ready to autotune the attitude and position control loops to improve the performance of the multirotor. The control system for this example contains eight PID Controllers. The system has four cascading control loops. Each loop contains two controllers, one for each axis. This diagram shows how the eight controllers are set up with the Closed-Loop PID Autotuner blocks in order to perform autotuning.



The Closed-Loop PID Autotuner blocks inject perturbation signals to the output of each of the eight existing PID Controllers. The autotuners then use the feedback signals and the output of the PID Controllers in order to perform the autotuning process. With the exception of the innermost control loops, pitch and roll rate, the two axes being controlled are decoupled from each other. For example, the x velocity and the y velocity loops are decoupled from each other. This allows you to tune these two loops simultaneously which reduces the overall time to perform autotuning. For the pitch rate and roll rate loops, tune the control loops sequentially because they are coupled. This results in the following sequence for tuning the PID Controllers:

1. Pitch Rate
2. Roll Rate

3 Pitch and Roll
4 X and Y Velocity
5 X and Y Position

Click the **Autotune PID Controllers** project shortcut, which sets up the model to hover at a low altitude and automatically tune the four PID Controllers, then runs the same four-waypoint mission from first step.

The Closed-Loop PID Autotuner blocks for each control loop are set up with different performance criteria depending on the control loop. For cascaded control, such as that used in this example, the inner loop should have a higher bandwidth than the outer loop to avoid instabilities. For this example, that means the pitch and roll rate control loops have the highest bandwidth while the x and y position control loops have the slowest bandwidths.

The settings used for the pitch and roll rate loops are:

- Bandwidth — 50 rad/sec
- Phase margin — 60 degrees
- Perturbation amplitude — 0.001

The settings used for the pitch and roll loops are:

- Bandwidth — 20 rad/sec
- Phase margin — 60 degrees
- Perturbation amplitude — 0.1

The settings used for the x and y velocity loops are:

- Bandwidth — 5 rad/sec
- Phase margin — 60 degrees
- Perturbation amplitude — 0.02

The settings used for the x and y position loops are:

- Bandwidth — 1 rad/sec
- Phase margin — 60 degrees
- Perturbation amplitude — 0.1

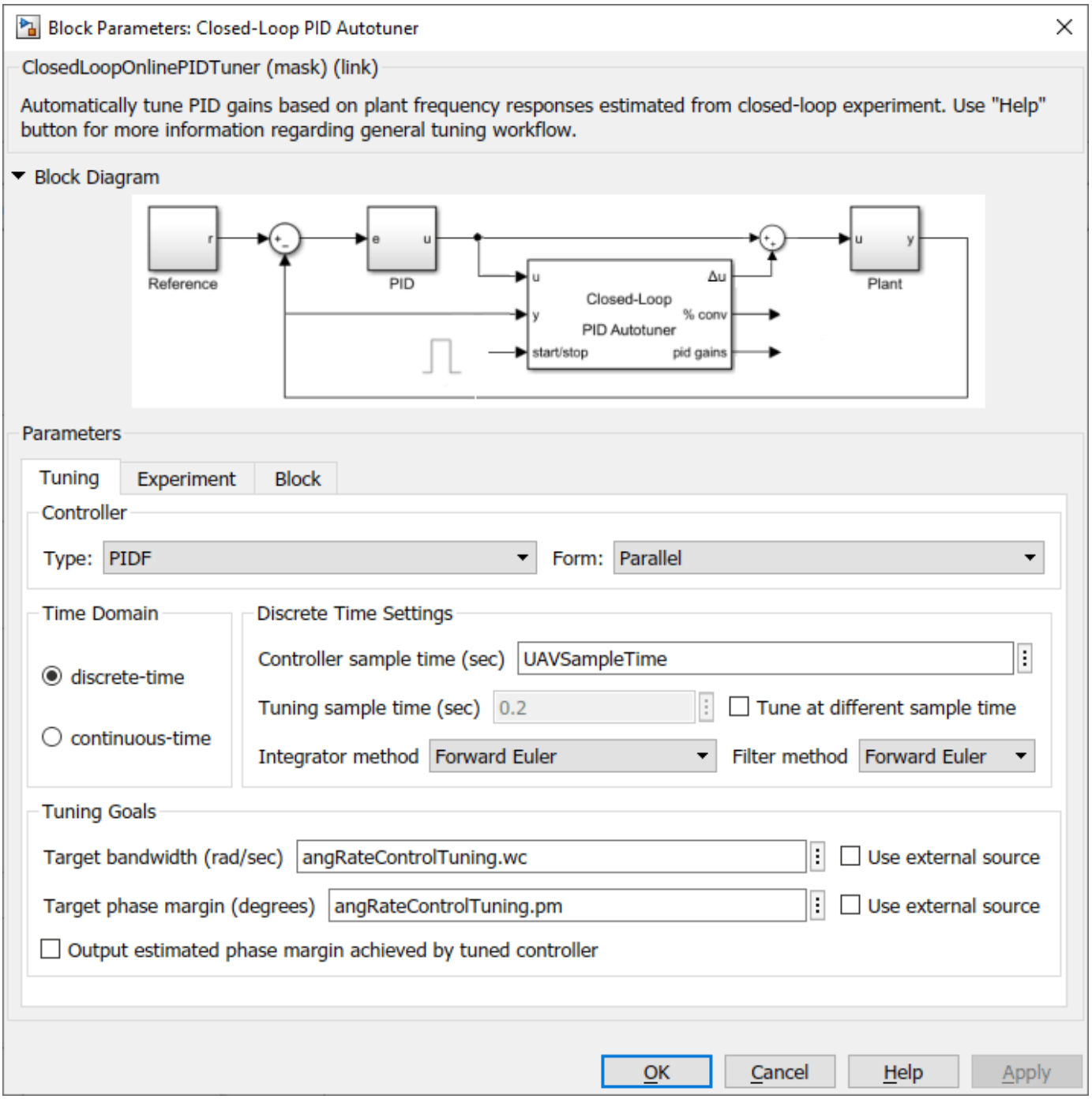
To maximize performance, the bandwidth for the pitch and roll rate loops is set to 50 rad/sec. The sampling time T_s of the UAV control system is 0.005 seconds, and the Closed-Loop PID Autotuner requires that the bandwidth ω must satisfy $\omega T_s \leq 0.3$, which means that bandwidth must be 60 rad/sec or less. Choose the bandwidth such that it is less than the required 60 rad/sec. The other bandwidths are set to be as large as possible while not causing stability issues with inner loops.

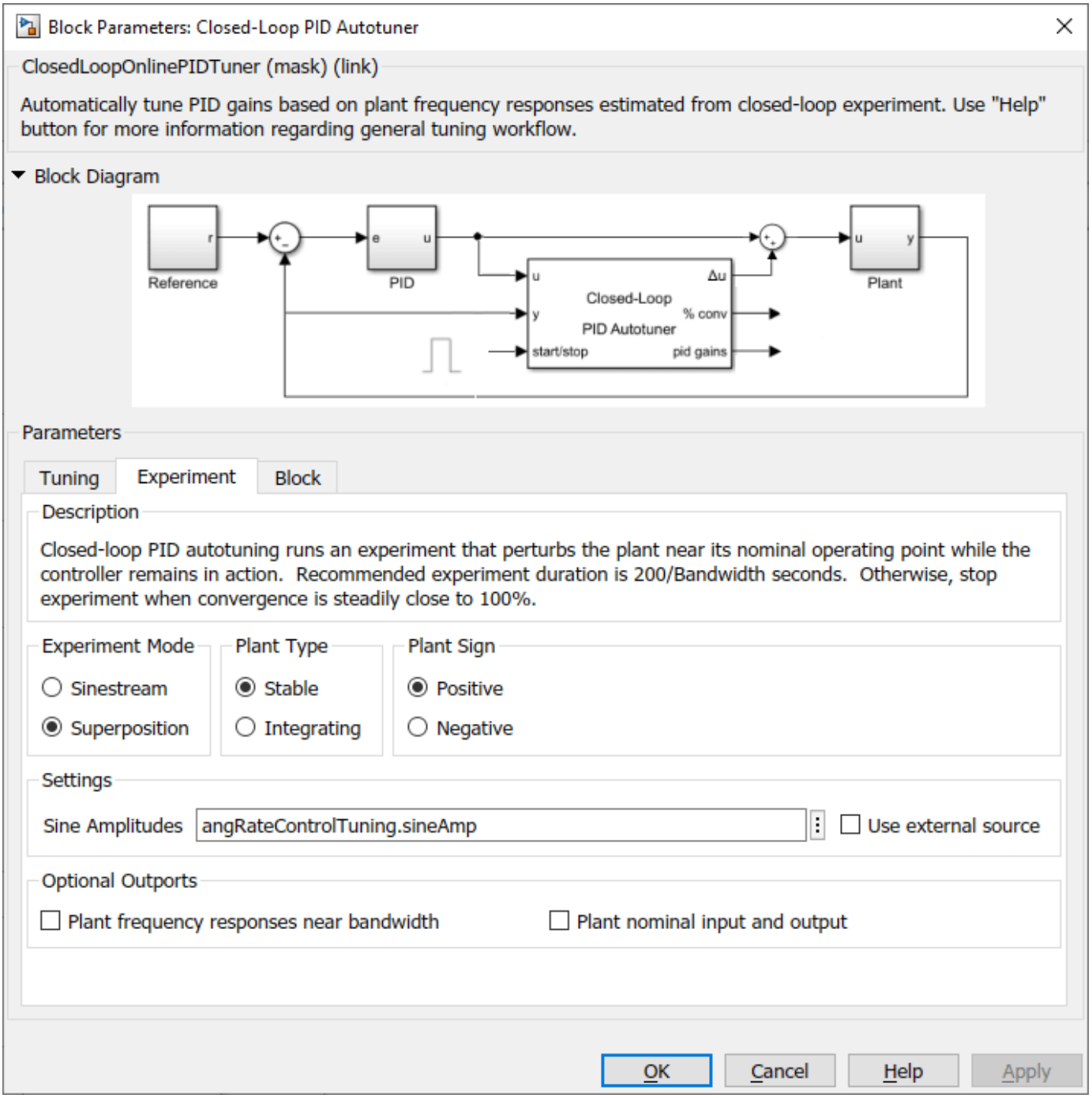
The phase margin for each loop is set to 60 degrees as this value is typically a good compromise between performance and damping. This margin is the default setting for the Closed-Loop PID Autotuner block.

The perturbation amplitudes are set such that they are less than 5% of the maximum expected output of the individual controllers. If the value for the perturbations is too high, it can cause the multirotor to become unstable during tuning. If the value for the perturbations is too low, the autotuner might

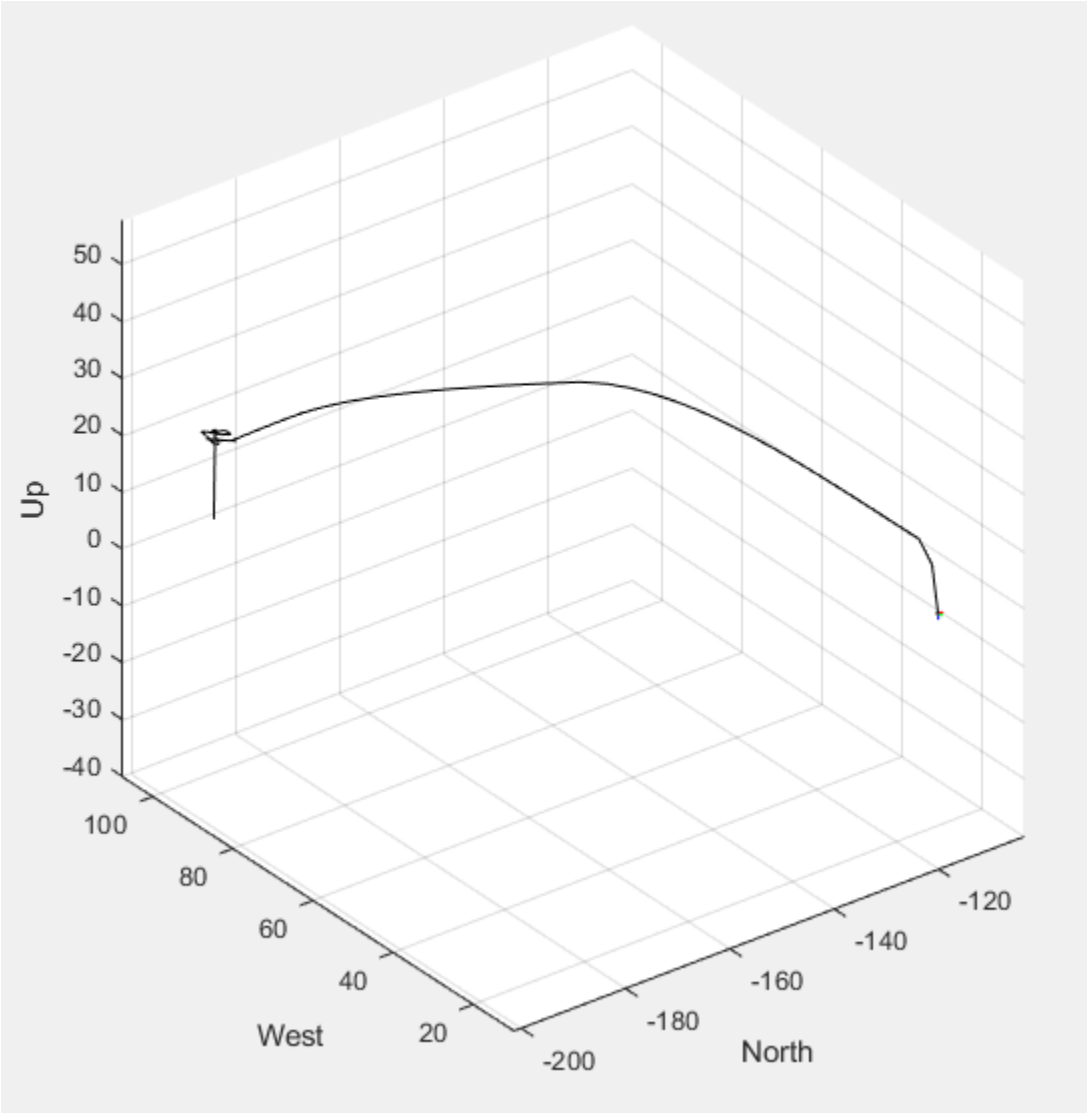
not get an accurate estimate of the plant and the calculated gains might not meet the desired bandwidth or phase margin.

The settings for the individual loops are contained in the data dictionary, `uavPackageDeliveryDataDict.sldd`. The following images show a sample of how to enter these settings in the Closed-Loop PID Autotuner blocks used to tune the pitch angular rate.

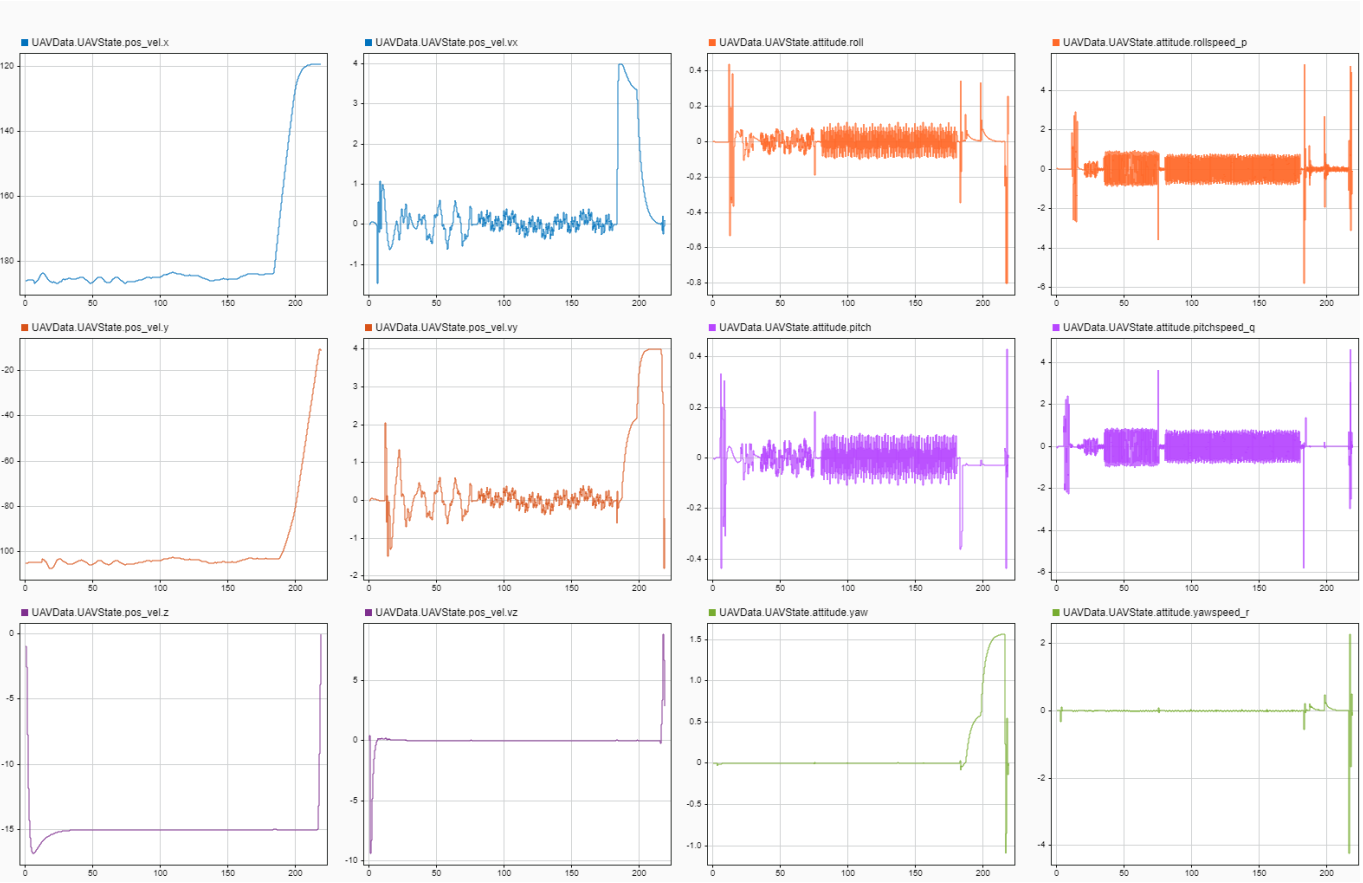




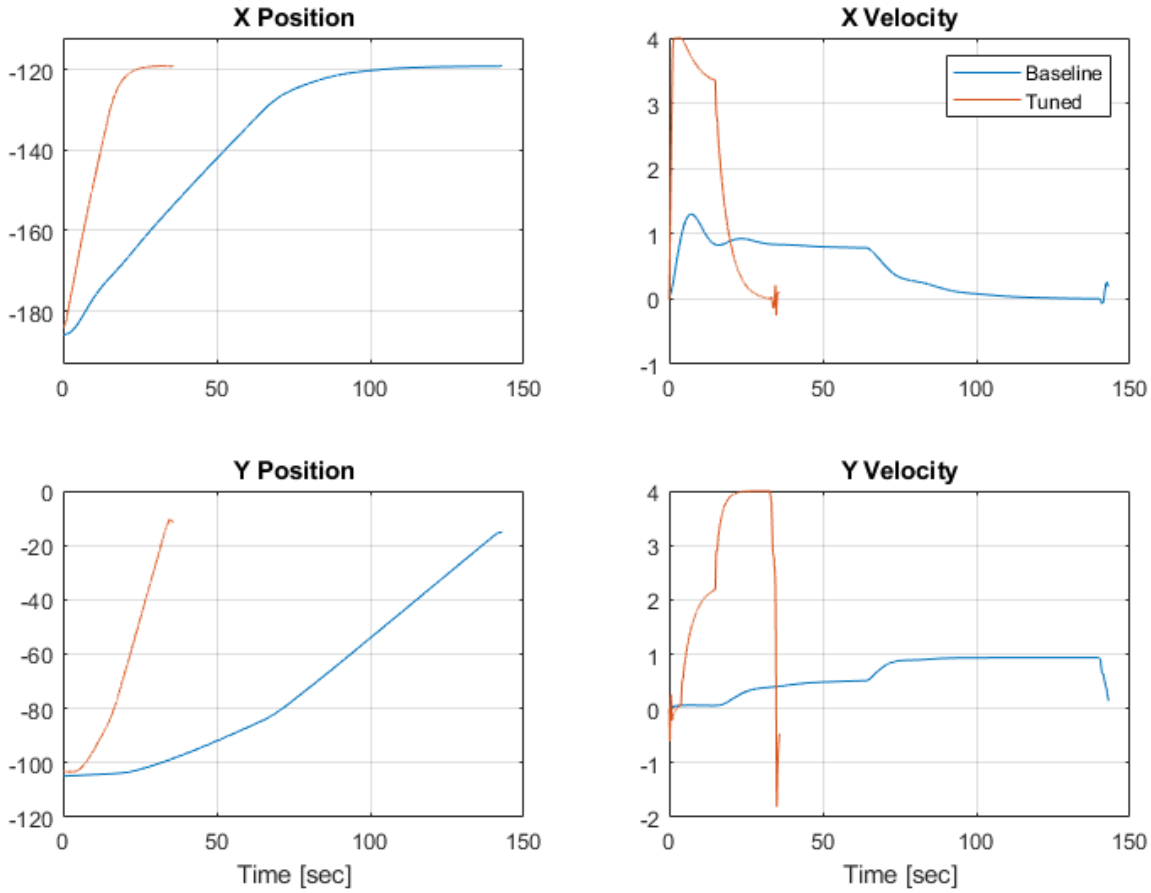
Run the `uavPIDAutotuning` model, which shows the multirotor takeoff, hover, autotune the PID Controllers, fly, and land in a 3-D plot.

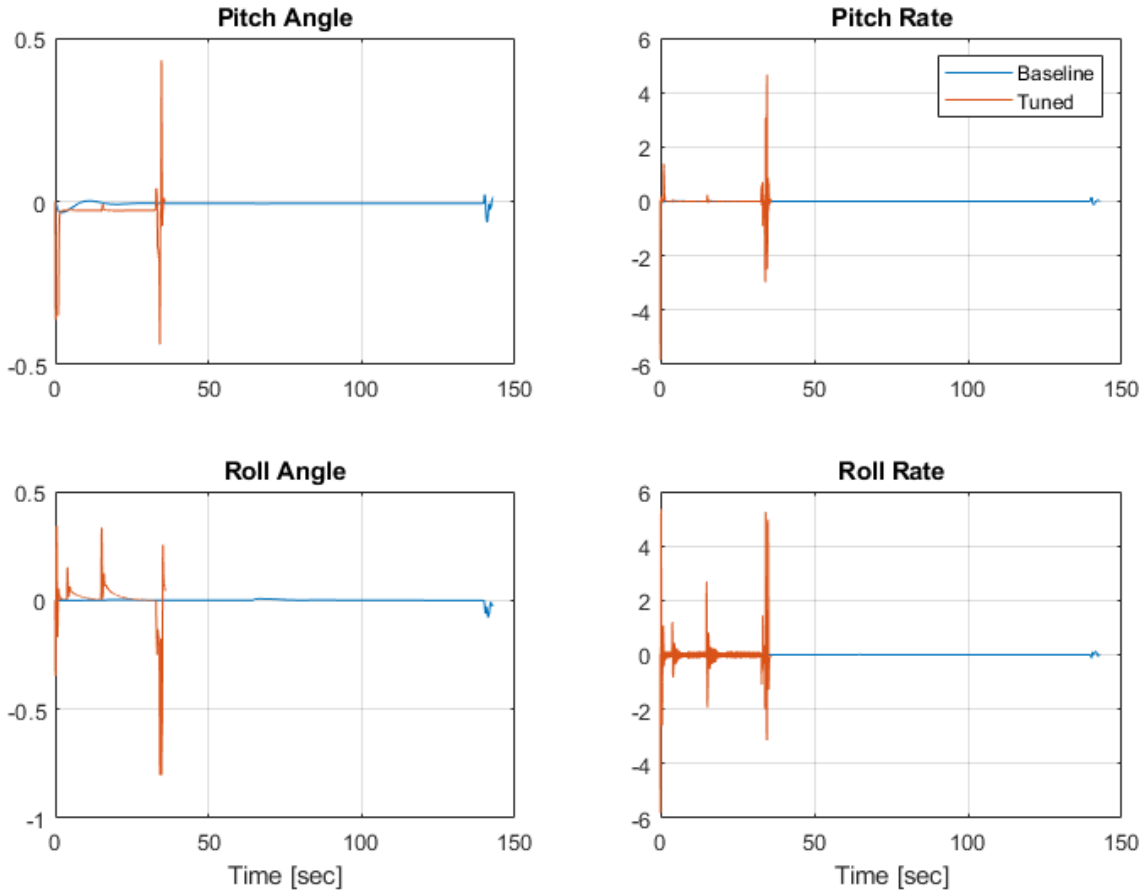


Use the Simulation Data Inspector to visualize the **UAVState** output of the multirotor model.



As you can see, the multirotor hovers for a period of time in order to perform autotuning. After the autotuning process is complete, around 185 seconds into the simulation, the multirotor follows the same four-waypoint path as in project first step, but the quadcopter is able to complete the path in a much shorter time due to the tuned gains increasing performance.


X Position
X Velocity
Baseline
Tuned
Y Position
Y Velocity
Time [sec]
Time [sec]



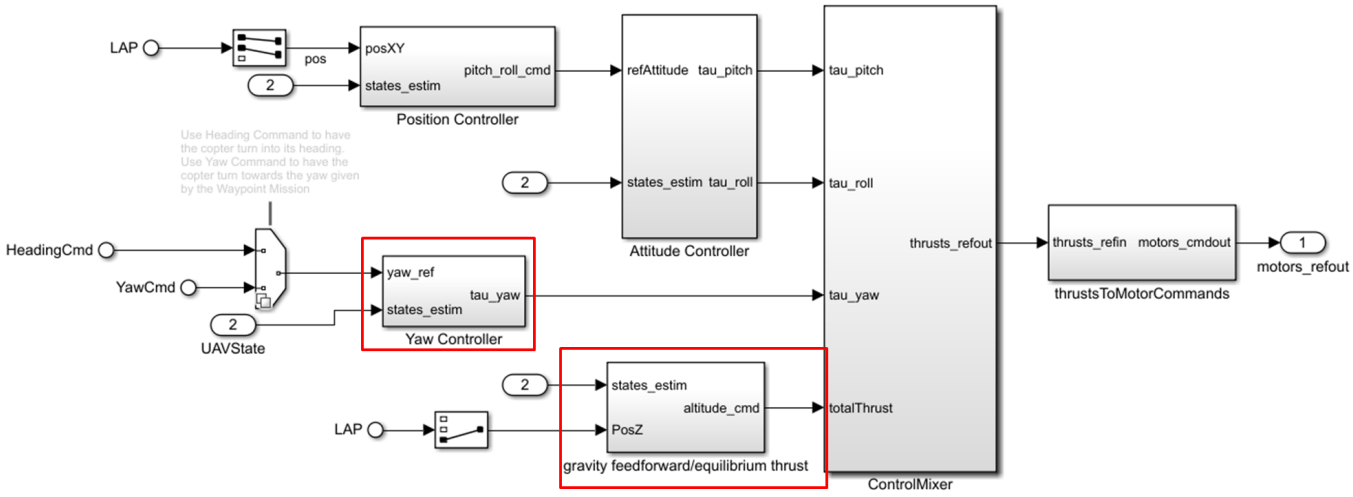
These plots show the position and attitude responses for the multirotor over the path. The blue line shows the multirotor performance with the baseline set of gains while the red line shows the multirotor performance with the tuned gains. With the tuned set of gains, the multirotor is able to complete the path in about 45 seconds. Meanwhile, with the baseline set of gains, the multirotor takes almost 150 seconds.

During the autotuning process the gains are updated for the eight controllers:

- Pitch rate — Kp = 0.00425, Ki = 0.01479, Kd = 0.0000045, N = 398
- Roll rate — Kp = 0.003477, Ki = 0.01215, Kd = 0.0000031, N = 398
- Pitch angle — Kp = 19.38
- Roll angle — Kp = 18.95
- X velocity — Kp = 0.5153, Ki = 0.2581
- Y velocity — Kp = 0.5201, Ki = 0.2979
- X position — Kp = 0.9365
- Y position — Kp = 0.9291

Autotuning Altitude and Heading/Yaw Control Loops

In this example you learned how to automatically tune the P, I, D and N gains for four separate paired control loops used for the attitude and position control. However, this example model contains two more control loops, one for altitude and one for the heading or yaw angle. Using the same methodology for autotuning presented in this example, you can add the Closed-Loop PID Autotuner to one or both of these other control loops and perform the autotuning process.



In order to tune the controllers for either of these loops, ensure that the tuning happens only when the tuning is not running for the other controllers. You can either disable tuning of the other controllers or you can tune these controllers after tuning has completed for position and attitude controllers.

When you are done exploring the models, close the project file.

```
close(prj)
```

See Also

Closed-Loop PID Autotuner

Related Examples

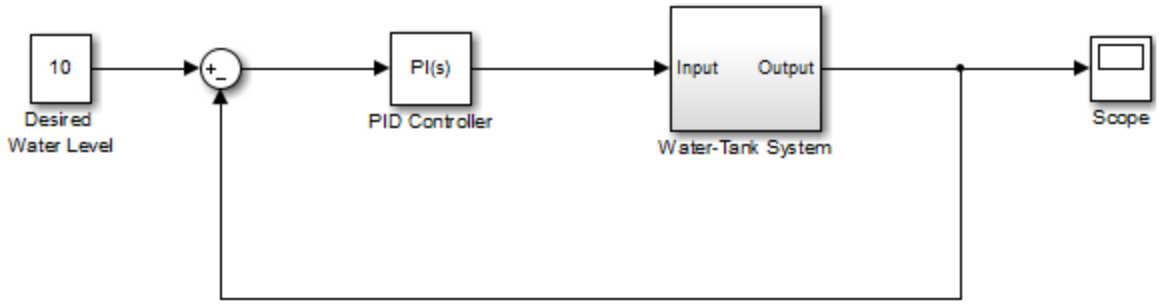
- “How PID Autotuning Works” on page 8-5
- “PID Autotuning for a Plant Modeled in Simulink” on page 8-7
- “UAV Inflight Failure Recovery” on page 14-2

Tune Gain-Scheduled Controller Using Closed-Loop PID Autotuner Block

This example shows how to use the Closed-Loop PID Autotuner block to tune a gain-scheduled controller in one simulation.

Water-Tank System Model

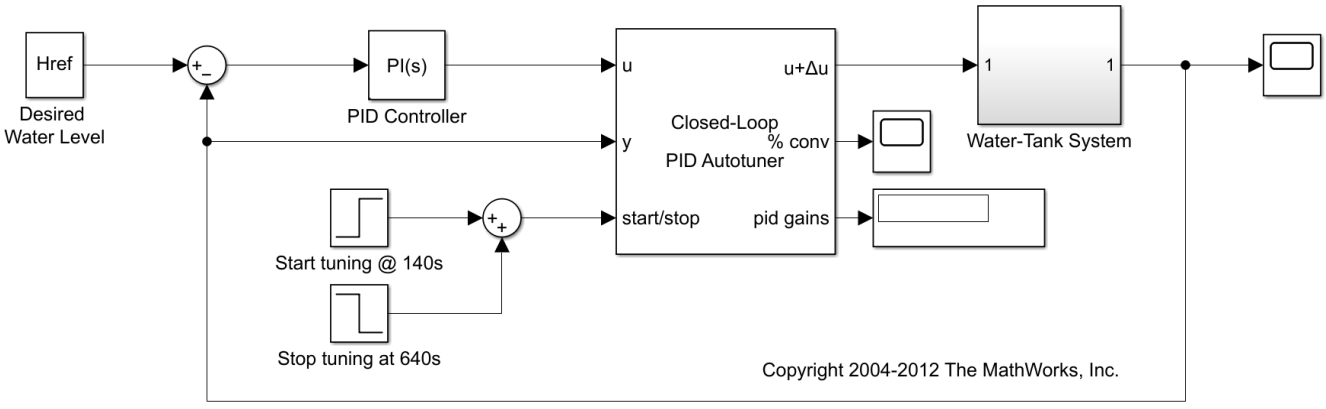
This example uses a gain-scheduled controller to control the water level of a nonlinear Water-Tank System plant. The Water-Tank System plant is originally controlled by a single PI controller in the `watertank` Simulink® model. For more details on the nonlinear Water-Tank System plant, see “watertank Simulink Model”.



The following sections describe how to modify the `watertank` model for tuning and validating gain-scheduled controller. Alternatively, use the `watertank_gainscheduledcontrol` model provided with this example.

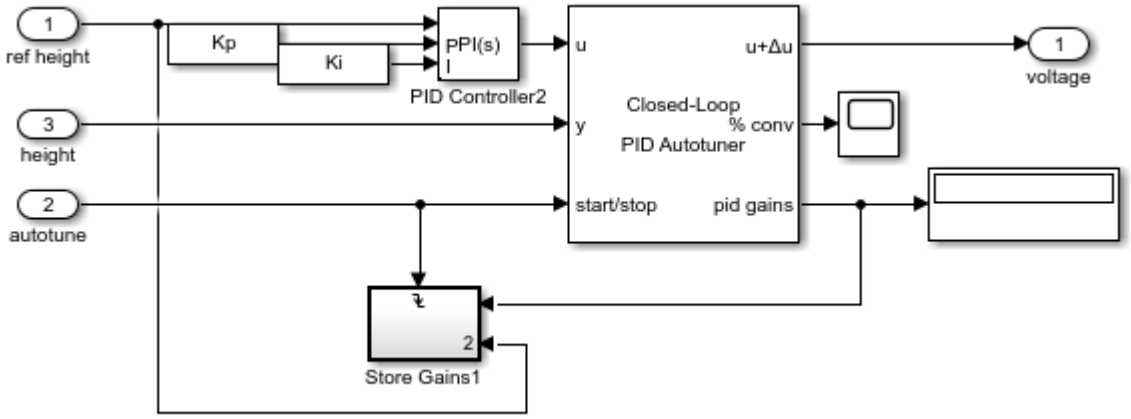
Connect Closed-Loop PID Autotuner Block with Plant and Controller

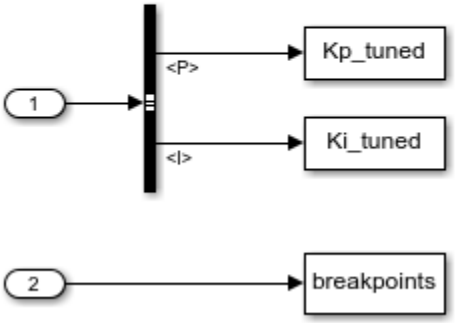
Insert the Closed-Loop PID Autotuner block between the controller and plant as shown in the following diagram. The **start/stop** signal starts and stops the closed-loop experiment. When no experiment is running, the Closed-Loop PID Autotuner block behaves like a unity gain block, where the **u** signal passes directly to **u+Δu**.



Connect Blocks to Store Tuned Gains

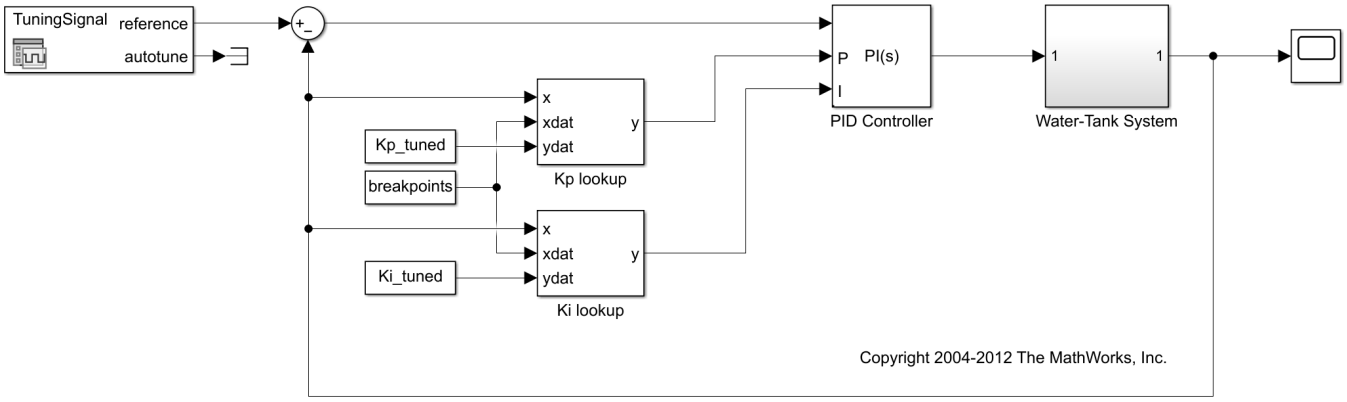
To create a gain schedule, the autotuned gains are recorded at each operating point. In this example, a triggered subsystem is used to write the reference heights and controller gains to the workspace upon falling edges of the autotuner **start/stop** signal. Simulating this model produces an array of tuned gains and breakpoints for easy use with dynamic lookup tables to test the controller.





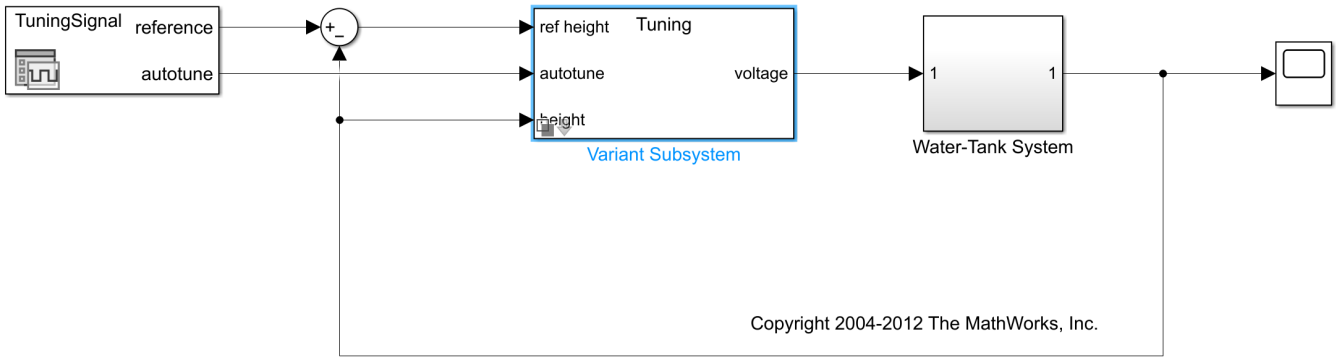
Validate Performance of Gain-Scheduled Controller

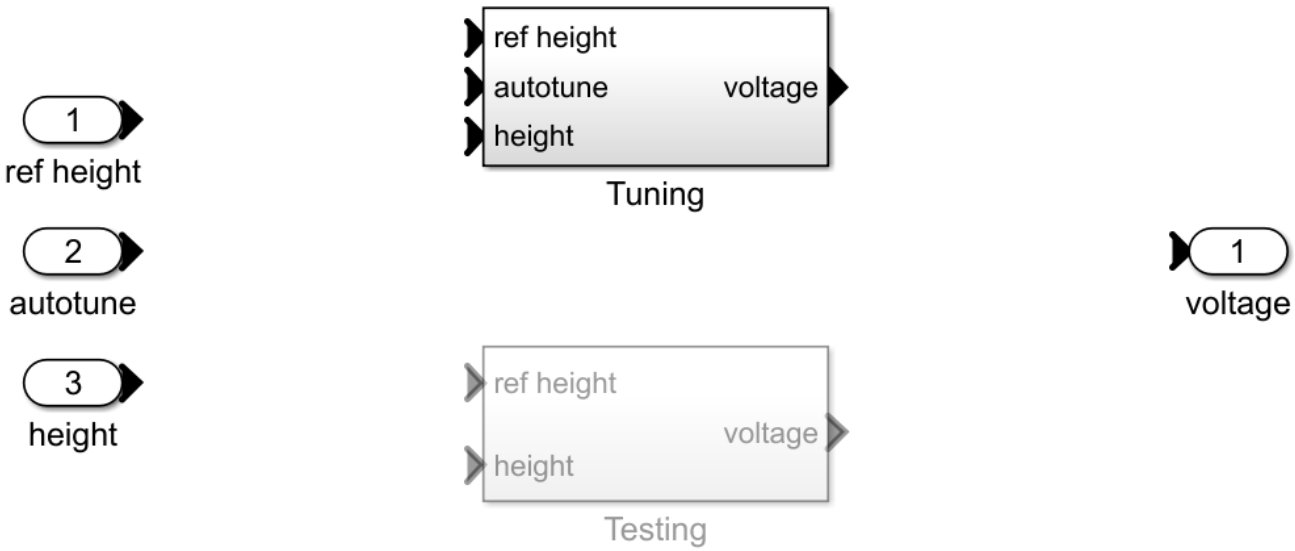
After you obtain a set of breakpoints and tuned gains, test the tuned gain-scheduled controller with the Water-Tank System plant. To do so, remove the autotuner block, change the source of the PID Controller block to external, and insert Lookup Table Dynamic blocks as shown in the diagram.



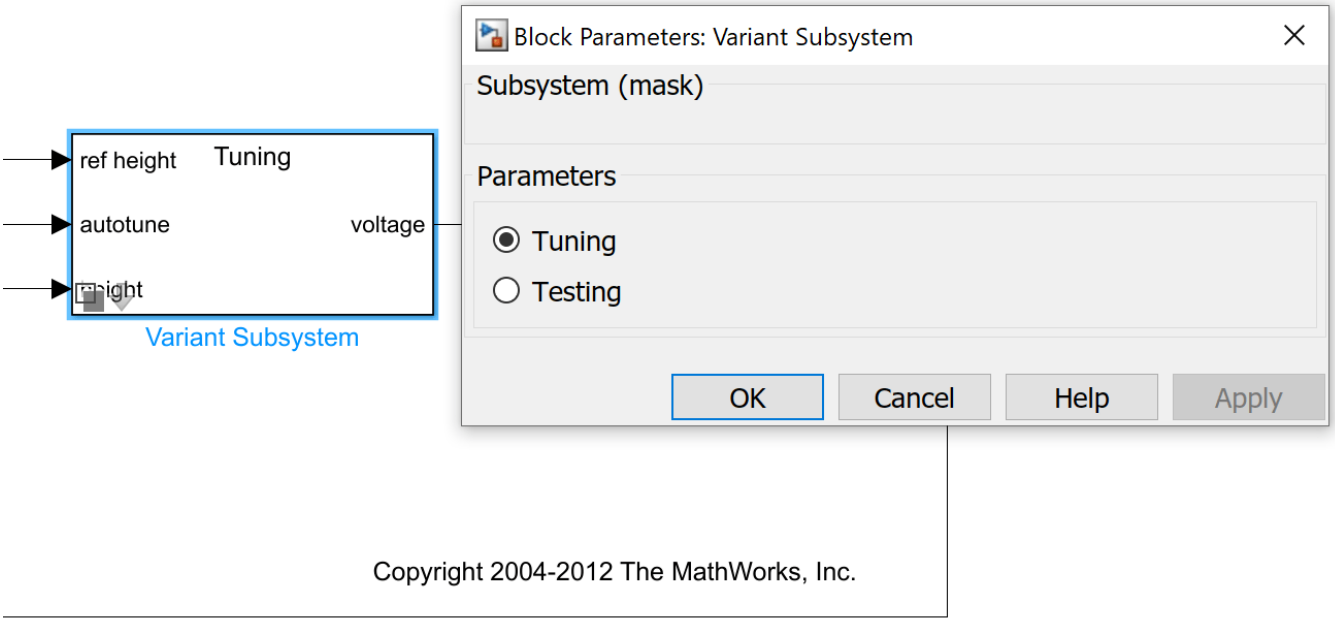
Integrate Both Tuning and Testing in Example Model

In this example, a gain-scheduled controller is tuned using the Closed-Loop PID Autotuner block and its performance is then tested in the same model. The example model uses a variant subsystem to organize the tuning and testing workflows.





To switch between Tuning and Testing modes, double-click the Variant Subsystem block.



Tune Controller at Single Operating Point

Before tuning the gain-scheduled controller at multiple operating points, tuning at single operating point helps you configure the Closed-Loop PID Autotuner block. Open the example model `watertank_gainscheduledcontrol` with controller gains used by the `watertank` Simulink model.

```
mdl = 'watertank_gainscheduledcontrol';
Kp = 1.599340;
Ki = 0.079967;
open_system(mdl);
set_param([mdl,'/Variant Subsystem'],'SimMode','Tuning');
```

Configure Closed-Loop PID Autotuner Block

After connecting the Closed-Loop PID Autotuner block with the Water-Tank System plant model and PID Controller block, use the block parameters to specify tuning and experiment settings. This example uses the same design requirements found in the example “Design Compensator Using Automated PID Tuning and Graphical Bode Design” on page 9-11. These design requirements are in the form of closed-loop step response characteristics.

- Overshoot less than 5%
- Rise time less than 5 seconds

To tune the PID controller to meet the above design requirements, parameters of the Closed-Loop PID Autotuner block are pre-populated. The **Tuning** tab has three main tuning settings.

- **Target bandwidth —** Determines how fast you want the controller to respond. The target bandwidth is roughly 2/desired rise time. For a desired rise time of 4 seconds, set target bandwidth = 2/4 = 0.5 rad/s.
- **Target phase margin —** Determines how robust you want the controller to be. In this example, start with the default value of 60 degrees.
- **Experiment sample time —** Sample time for the experiment performed by the autotuner block. Use the recommended 0.02/bandwidth for sample time = 0.02/0.5 = 0.04s.

The **Experiment** tab has three main experiment settings.

- **Plant Type —** Specifies whether the plant is asymptotically stable or integrating. In this example, the Water-Tank System plant is integrating.
- **Plant Sign —** Specifies whether the plant has a positive or negative sign. The plant sign is positive if a positive change in the plant input at the nominal operating point results in a positive change in the plant output when the plant reaches a new steady state. In this example, the Water-Tank System plant has a positive plant sign.
- **Sine Amplitudes —** Specifies amplitudes of the injected sine wave perturbations. In this example, specify a sine amplitude of 0.3.

Simulate at One Operating Point

Start the experiment at 140 seconds to ensure that the water level has reached steady-state $H = 10$. The recommended experiment duration is 200/bandwidth seconds = 200/0.4 = 500s. With start time of 140 seconds, the stop time is 640 seconds**.** The simulation stop time is further increased to capture the full experiment.

```
set_param([mdl,'/Variant Subsystem/Tuning/Closed-Loop PID Autotuner1'],'TargetPM','60');
set_param([mdl,'/Signal Editor'],'ActiveScenario','TuningSignal_OnePoint');
simOut = sim(mdl,'StopTime','800');
simOut.Kp_tuned
```

```
ans = 1.8257
```

```
simOut.Ki_tuned
```

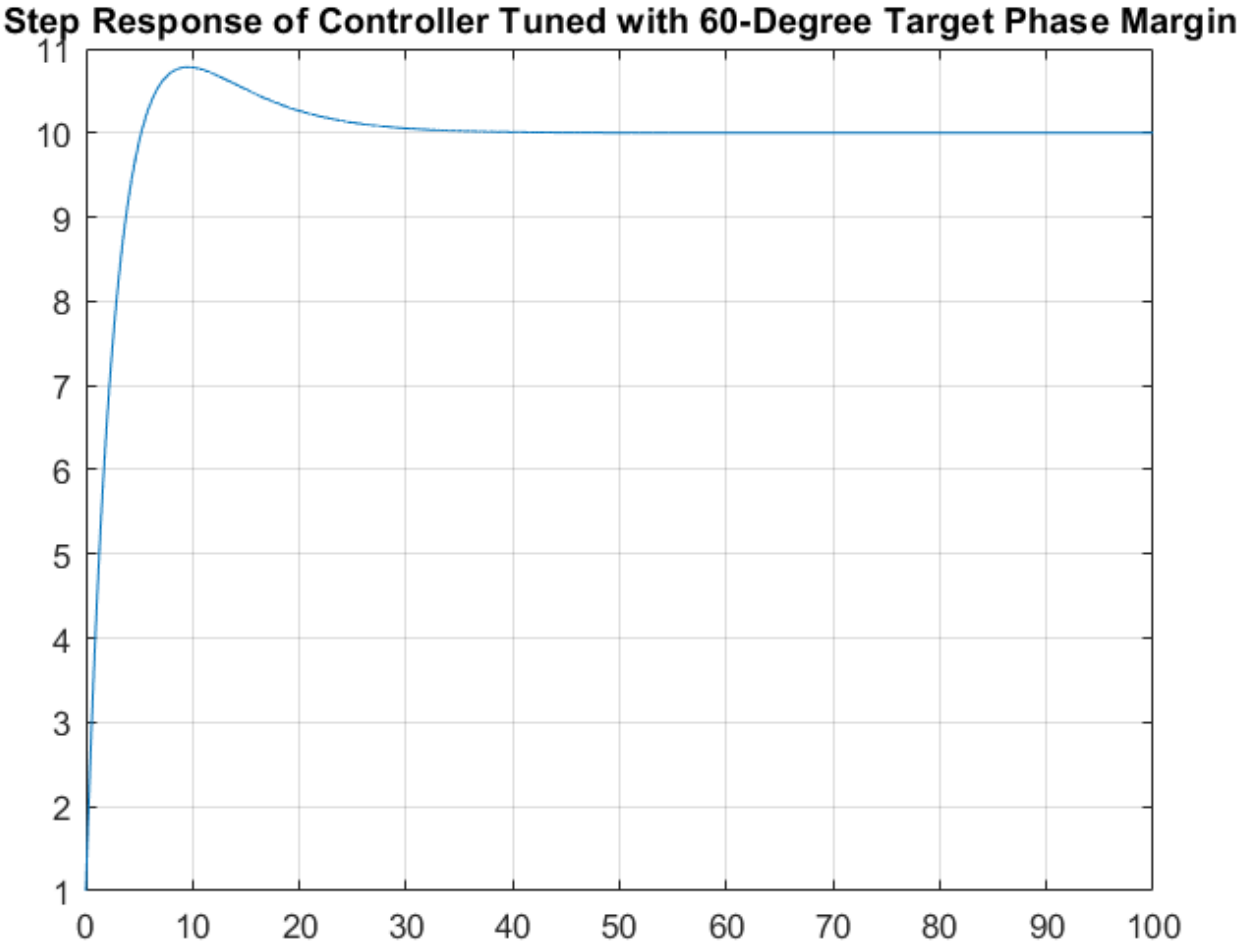
```
ans = 0.2037
```

In the `watertank` Simulink model, initial PI controller gains are Kp = 1.599340 and Ki = 0.079967. After tuning, the controller gains are Kp = 1.82567 and Ki = 0.20373.

Check Tuning Result and Adjust Autotuning Parameters

Replace controller gains with the new autotuned gains and validate the design requirements.

```
Kp = simOut.Kp_tuned;
Ki = simOut.Ki_tuned;
simOut = sim(mdl,'StopTime','100');
figure;
plot(simOut.ScopeDataGS.time,simOut.ScopeDataGS.signals.values);
grid on
title('Step Response of Controller Tuned with 60-Degree Target Phase Margin');
```




```
StepPerformance_OnePoint = stepinfo(simOut.ScopeDataGS.signals.values(:),simOut.ScopeDataGS.time
```

```
StepPerformance_OnePoint = struct with fields:
        RiseTime: 3.6251
```

```
    TransientTime: 22.5219
     SettlingTime: 22.5219
      SettlingMin: 9.1089
      SettlingMax: 10.7821
        Overshoot: 8.6895
       Undershoot: 0
             Peak: 9.7821
         PeakTime: 9.5500
```

The step response has a rise time of 3.6251 seconds and overshoot of 8.6895%. The overshoot is larger than desired; increase target phase margin to 75 degrees to improve the closed-loop transient response.

```
set_param([mdl,'/Variant Subsystem/Tuning/Closed-Loop PID Autotuner1'],'TargetPM','75');
```

Examine the simulation result. The system is at steady-state when experiment starts and returns to steady-state after tuning is completed. As an indication of controller tuning performance, the Closed-Loop PID Autotuner block reaches 100% convergence level sooner than the recommended 500 seconds. As a result, reduce experiment duration to 300 seconds, meaning a stop time of 440 seconds. Accordingly, decrease the simulation stop time from 800 seconds to 500 seconds.

```
Kp = 1.599340;
Ki = 0.079967;
set_param([mdl,'/Signal Editor'],'ActiveScenario','TuningSignal_OnePointAdjusted');
simOut = sim(mdl,'StopTime','500');
simOut.Kp_tuned
```

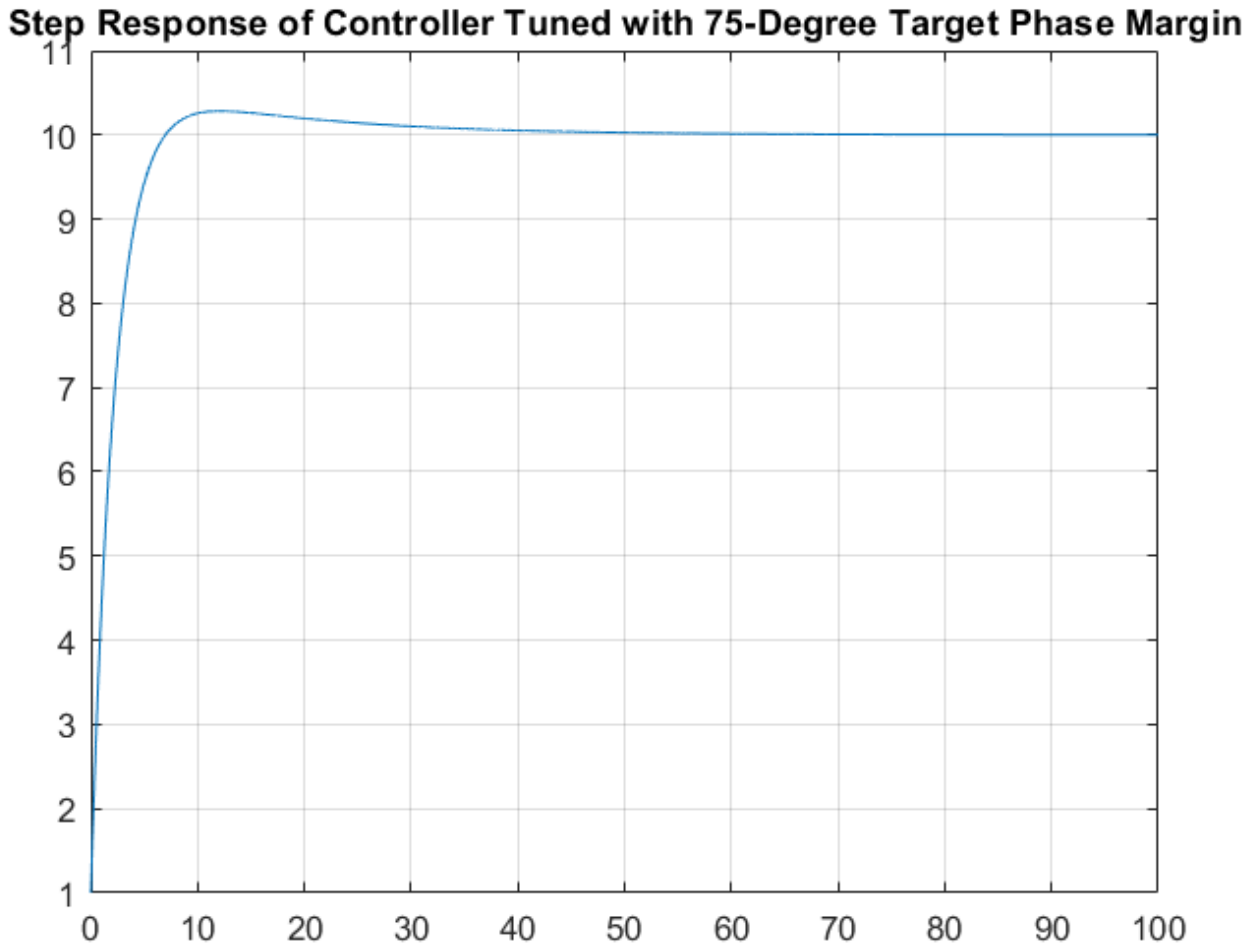
```
ans = 1.9351
```

```
simOut.Ki_tuned
```

```
ans = 0.1141
```

Simulating with new experiment parameters produces tuned gains of Kp = 1.93514 and Ki = 0.11415**.** Examine the step response again using gains tuned with the increased target phase margin value.

```
Kp = simOut.Kp_tuned;
Ki = simOut.Ki_tuned;
simOut = sim(mdl,'StopTime','100');
figure;
plot(simOut.ScopeDataGS.time,simOut.ScopeDataGS.signals.values);
grid on
title('Step Response of Controller Tuned with 75-Degree Target Phase Margin');
```




```
StepPerformance_OnePointAdjusted = stepinfo(simOut.ScopeDataGS.signals.values(:),simOut.ScopeData
```

```
StepPerformance_OnePointAdjusted = struct with fields:
         RiseTime: 4.1398
    TransientTime: 21.4047
     SettlingTime: 21.4047
      SettlingMin: 9.1044
      SettlingMax: 10.2829
        Overshoot: 3.1438
       Undershoot: 0
             Peak: 9.2829
         PeakTime: 12.1100
```

The step response has a rise time of 4.1398 seconds and overshoot of 3.1438%, both of which meet the design requirements.

Simulate the model with tuned gains for multiple operating points H = [5, 10, 15, 20].

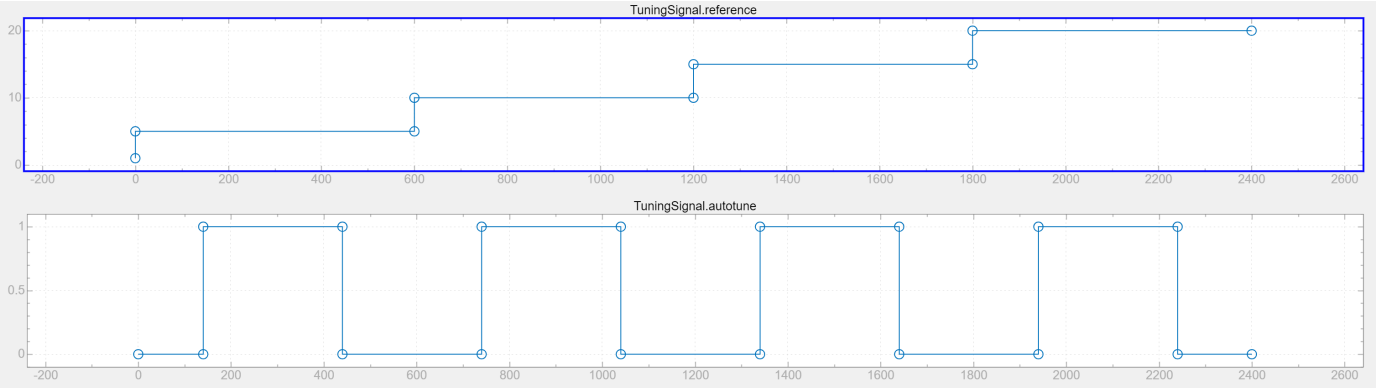
```
set_param([mdl,'/Signal Editor'],'ActiveScenario','TuningSignal_SinglePID');
simOut_single = sim(mdl,'StopTime','2400');
```

Tune Gain-Scheduled Controller at Multiple Operating Points

The set of tuned gains produces a desired response. You can now perform tuning at multiple operating points to create a gain-scheduled controller.

Create Input Tuning Signal

The operating range of scheduling variable H from 1 to 20 is covered by the operating points for autotuning. In this example, the gain-scheduled controller gains are tuned at four operating points with H = [5, 10, 15, 20]. To tune at multiple operating points, use the Signal Editor block to create the reference and autotuner start/stop signal



Simulate Multiple Operating Points

Using the input signal, simulate the `watertank_gainscheduledcontrol` model for the entire length of the autotuning process. At the end of simulation, save both tuned gains and breakpoints as vectors in the MATLAB® Workspace.

```
Kp = 1.599340;
Ki = 0.079967;
set_param([mdl,'/Signal Editor'],'ActiveScenario','TuningSignal');
simOut = sim(mdl,'StopTime','2400');
Kp_tuned = simOut.Kp_tuned
```

```
Kp_tuned = 4×1

    1.9283
    1.9351
    1.9382
    1.9400
```

```
Ki_tuned = simOut.Ki_tuned
```

```
Ki_tuned = 4×1

    0.1277
    0.1141
    0.1082
    0.1046
```

```
breakpoints = simOut.breakpoints
```

```
breakpoints = 4×1

    0.1550
    0.1579
    0.1592
    0.1600
```

Performance Improvements of Gain-Scheduled Controller

To examine the performance of the gain-scheduled controller, set the Variant Subsystem to Testing mode and simulate the model.

```
set_param([mdl,'/Variant Subsystem'],'SimMode','Testing');
simOut_GS = sim(mdl,'StopTime','2400');
```

Using the gain-scheduled controller, step responses of the water level in the Water-Tank System plant are much faster and have less overshoot than the untuned controller used in “watertank Simulink Model”.

In addition, the tuned gain-scheduled controller leads to a better transient performance than the single set of gains tuned at water level $H = 10$.

Use the `compareControllers_watertank` script to compute the step-response characteristics for the PID controller tuned at $H = 10$ and the gain-scheduled controller. The script generates two tables, which contain the rise time (in seconds) and percentage overshoot for the gain-scheduled controller and a single set of controller gains.

```
compareControllers_watertank
```

```
RiseTime=2×4 table
                    H = 1 to 5    H = 5 to 10    H = 10 to 15    H = 15 to 20
                    __________    ___________    ____________    ____________

    Single PID        4.6721        3.7818          3.715          3.6826
    Gain-Scheduled    4.8012         3.845         3.7744          3.7402


Overshoot=2×4 table
                    H = 1 to 5    H = 5 to 10    H = 10 to 15    H = 15 to 20
                    __________    ___________    ____________    ____________

    Single PID        0.69606       5.2553          5.888          6.2236
    Gain-Scheduled    0.14074       4.6827         5.3208          5.6592
```

The gain-scheduled controller leads to a smaller overshoot for a comparable rise time, compared to a single set of gains tuned at one operating point. This workflow is useful when you want to tune a gain-scheduled controller using the Closed-Loop PID Autotuner block.

Close the model.

```
close_system(mdl,0);
```

See Also

Closed-Loop PID Autotuner | PID Controller | Signal Editor | Lookup Table Dynamic | Variant Subsystem, Variant Model

Related Examples

- “watertank Simulink Model”
- “How PID Autotuning Works” on page 8-5
- “Design Compensator Using Automated PID Tuning and Graphical Bode Design” on page 9-11

Classical Control Design

- “Choose a Control Design Approach” on page 9-2
- “Control System Designer Tuning Methods” on page 9-4
- “What Blocks Are Tunable?” on page 9-8
- “Designing Compensators for Plants with Time Delays” on page 9-9
- “Design Compensator Using Automated PID Tuning and Graphical Bode Design” on page 9-11
- “Analyze Designs Using Response Plots” on page 9-27
- “Compare Performance of Multiple Designs” on page 9-36
- “Update Simulink Model and Validate Design” on page 9-40
- “Single Loop Feedback/Prefilter Compensator Design” on page 9-41
- “Cascaded Multiloop Feedback Design” on page 9-49
- “Tune Custom Masked Subsystems” on page 9-59
- “Tune Simulink Blocks Using Compensator Editor” on page 9-67
- “Reference Tracking of DC Motor with Parameter Variations” on page 9-74
- “Regulate Pressure in Drum Boiler” on page 9-79
- “Model Computational Delay and Sampling Effects” on page 9-86

Choose a Control Design Approach

Simulink Control Design lets you design and tune many types of control systems in Simulink. There are also deployable PID autotuning tools that let you tune your controller in real time against a physical plant.

Design in Simulink

Simulink Control Design provides several approaches to tuning Simulink blocks, such as Transfer Fcn and PID Controller blocks. Use the following table to determine which approach best supports what you want to do.

	Model-Based PID Tuning	Classical Control Design	Multiloop, Multiobjective Tuning
Supported Blocks	• PID Controller • Discrete PID Controller • PID Controller (2DOF) • Discrete PID Controller (2DOF)	Linear Blocks (see "What Blocks Are Tunable?" on page 9-8)	Any blocks; only some blocks are automatically parameterized (See "How Tuned Simulink Blocks Are Parameterized")
Architecture	1-DOF and 2-DOF PID loops	Control systems that contain one or more SISO compensators	Any structure, including any number of SISO or MIMO feedback loops
Control Design Approach	Automatically tune PID gains to balance performance and robustness	• Graphically tune poles and zeros on design plots, such as Bode, root locus, and Nichols • Automatically tune compensators using response optimization (Simulink Design Optimization), LQG synthesis, or IMC tuning	Automatically tune controller parameters to meet design requirements you specify, such as setpoint tracking, stability margins, disturbance rejection, and loop shaping (see "Tuning Goals")
Analysis of Control System Performance	Time and frequency responses for reference tracking and disturbance rejection	Any combination of system responses	Any combination of system responses

	Model-Based PID Tuning	Classical Control Design	Multiloop, Multiobjective Tuning
Interface	• Graphical tuning using **PID Tuner** (see “Introduction to Model-Based PID Tuning in Simulink” on page 7-2) • Tuning of plants that do not linearize (see “Frequency-Response Based Tuning” on page 7-37) • Programmatic tuning using `pidtune` (see “PID Controller Design at the Command Line”)	Graphical tuning using **Control System Designer**	• Graphical tuning using **Control System Tuner** • Programmatic tuning using `slTuner` (see “Programmatic Tuning”)

Real-Time PID Autotuning

The real-time PID autotuning tools in Simulink Control Design let you deploy an automatic tuning algorithm as a stand-alone application for PID tuning against a physical plant. Real-time PID autotuning lets you tune a PID controller to achieve a specified bandwidth and phase margin without a parametric plant model or an initial controller design.

The real-time PID autotuning algorithm can tune PID gains in Simulink PID Controller blocks or in your own custom PID blocks. You can tune against your physical plant with or without Simulink in the loop. Deploying the real-time PID autotuning algorithm requires a code-generation product such as Simulink Coder.

For more information, see “When to Use PID Autotuning” on page 8-2.

See Also

More About

- “PID Controller Tuning”
- “Classical Control Design”
- “Tuning with Control System Tuner”
- “Programmatic Tuning”

Control System Designer Tuning Methods

Using **Control System Designer**, you can tune compensators using various graphical and automated tuning methods.

Graphical Tuning Methods

Use graphical tuning methods to interactively add, modify, and remove controller poles, zeros, and gains.

Tuning Method	Description	Useful For
Bode Editor	Tune your compensator to achieve a specific open-loop frequency response (loop shaping).	Adjusting open-loop bandwidth and designing to gain and phase margin specifications.
Closed-Loop Bode Editor	Tune your prefilter to improve closed-loop system response.	Improving reference tracking, input disturbance rejection, and noise rejection.
Root Locus Editor	Tune your compensator to produce closed-loop pole locations that satisfy your design specifications.	Designing to time-domain design specifications, such as maximum overshoot and settling time.
Nichols Editor	Tune your compensator to achieve a specific open-loop response (loop shaping), combining gain and phase information on a Nichols plot.	Adjusting open-loop bandwidth and designing to gain and phase margin specifications.

When using graphical tuning, you can modify the compensator either directly from the editor plots or using the compensator editor. A common design approach is to roughly tune your compensator using the editor plots, and then use the compensator editor to fine-tune the compensator parameters. For more information, see “Edit Compensator Dynamics”

The graphical tuning methods are not mutually exclusive. For example, you can tune your compensator using both the Bode editor and root locus editor simultaneously. This option is useful when designing to both time-domain and frequency-domain specifications.

For examples of graphical tuning, see the following:

- “Bode Diagram Design”
- “Root Locus Design”
- “Nichols Plot Design”

Automated Tuning Methods

Use automated tuning methods to automatically tune compensators based on your design specifications.

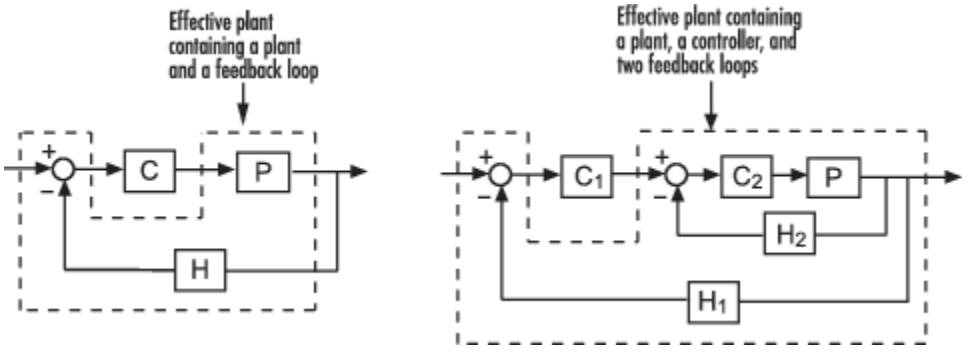
Tuning Method	Description	Requirements and Limitations
PID Tuning	Automatically tune PID gains to balance performance and robustness or tune controllers using classical PID tuning formulas.	Classical PID tuning formulas require a stable or integrating effective plant.
Optimization Based Tuning	Optimize compensator parameters using design requirements specified in graphical tuning and analysis plots.	Requires Simulink Design Optimization software. Tunes the parameters of a previously defined controller structure.
LQG Synthesis	Design a full-order stabilizing feedback controller as a linear-quadratic-Gaussian (LQG) tracker.	Maximum controller order depends on the effective plant dynamics.
Loop Shaping	Find a full-order stabilizing feedback controller with a specified open-loop bandwidth or shape.	Requires Robust Control Toolbox software. Maximum controller order depends on the effective plant dynamics.
Internal Model Control (IMC) Tuning	Obtain a full-order stabilizing feedback controller using the IMC design method.	Assumes that your control system uses an IMC architecture that contains a predictive model of your plant dynamics. Maximum controller order depends on the effective plant dynamics.

A common design approach is to generate an initial compensator using PID tuning, LQG synthesis, loop shaping, or IMC tuning. You can then improve the compensator performance using either optimization-based tuning or graphical tuning.

For more information on automated tuning methods, see “Design Compensator Using Automated Tuning Methods”.

Effective Plant for Tuning

An *effective plant* is the system controlled by a compensator that contains all elements of the open loop in your model other than the compensator you are tuning. The following diagrams show examples of effective plants:



Knowing the properties of the effective plant seen by your compensator can help you understand which tuning methods work for your system. For example, some automated tuning methods apply only to compensators whose open loops ($L = C\hat{P}$) have stable effective plants ($\hat{P}$). Also, for tuning methods such as IMC and loop shaping, the maximum controller order depends on the dynamics of the effective plant.

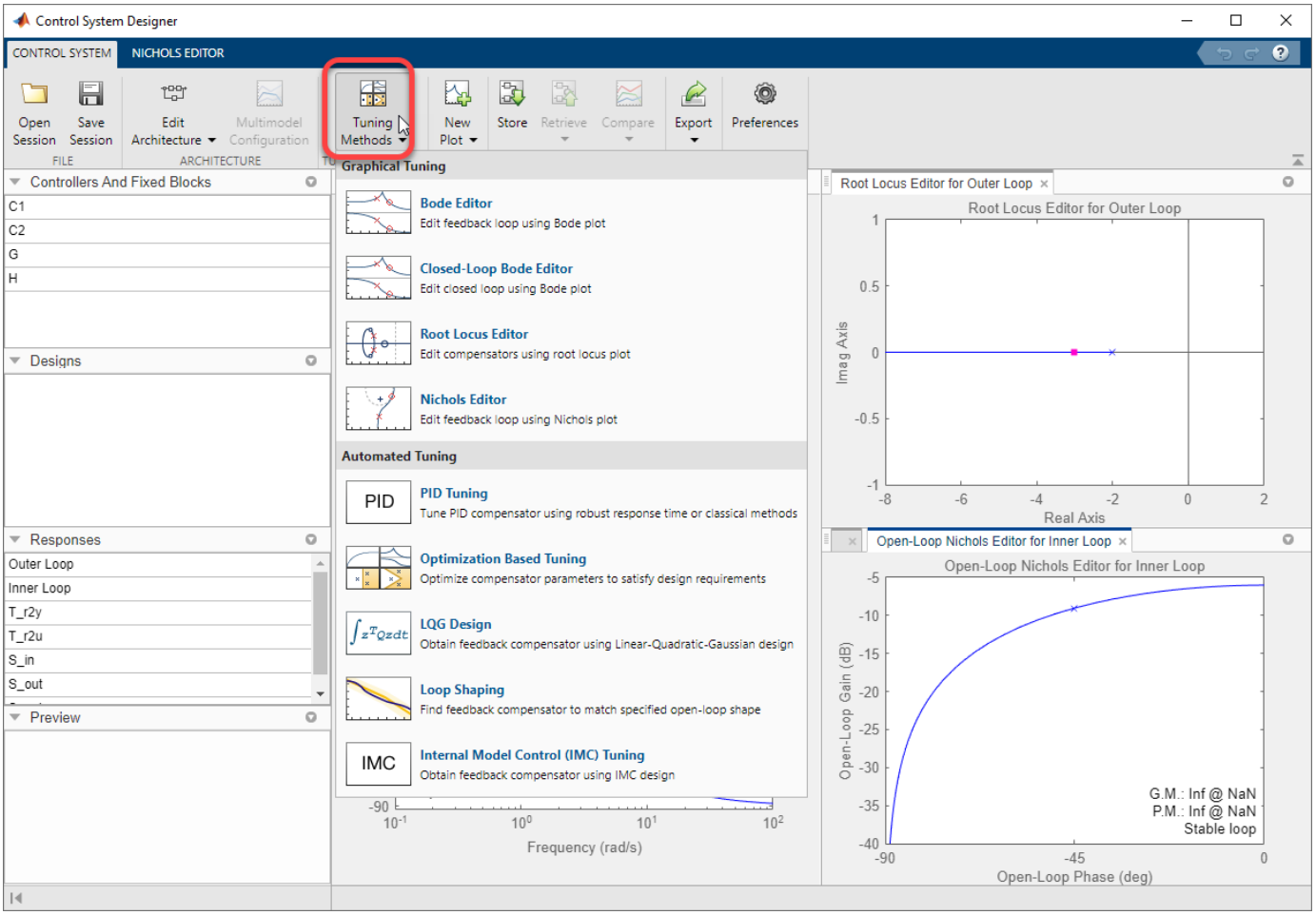
Tuning Compensators In Simulink

If the compensator in your Simulink model has constraints on its poles, zeros, or gain, you cannot use LQG synthesis, loop shaping, or IMC tuning. For example, you cannot tune the parameters of a PID Controller block using these methods. If your application requires controller constraints, use an alternative automated or graphical tuning method.

Also, any compensator constraints in your Simulink model limit the structure of your tuned compensator. For example, if you are using PID tuning and you configure your PID Controller block as a PI controller, your tuned compensator must have a zero derivative parameter.

Select a Tuning Method

To select a tuning method, in **Control System Designer**, click **Tuning Methods**.



See Also

Control System Designer

Related Examples

- “Bode Diagram Design”
- “Root Locus Design”
- “Nichols Plot Design”
- “Design Compensator Using Automated Tuning Methods”

What Blocks Are Tunable?

You can tune parameters in the following Simulink blocks using Simulink Control Design software. The block input and output signals for tunable blocks must have scalar, double-precision values.

Tunable Block	Description
Gain	Constant gain
LTI System	Linear time-invariant system
Discrete Filter	Discrete-time infinite impulse response filter
PID Controller	One-degree-of-freedom PID controller
Discrete PID Controller	Discrete-time one-degree-of-freedom PID controller
State-Space	Continuous-time state-space model
Discrete State-Space	Discrete-time state-space model
Zero-Pole	Continuous-time zero-pole-gain transfer function
Discrete Zero-Pole	Discrete-time zero-pole-gain transfer function
Transfer Fcn	Continuous-time transfer function model
Discrete Transfer Fcn	Discrete-time transfer function model

Additionally, you can tune the linear State-Space, Zero-Pole, and Transfer Fcn blocks in the Simulink Extras Additional Linear library.

You can tune the following versions of the listed tunable blocks:

- Blocks with custom configuration functions associated with a masked subsystem
- Blocks discretized using the Simulink Model Discretizer

Note If your model contains Model blocks with normal-mode model references to other models, you can select tunable blocks in the referenced models for compensator design.

See Also

Control System Designer

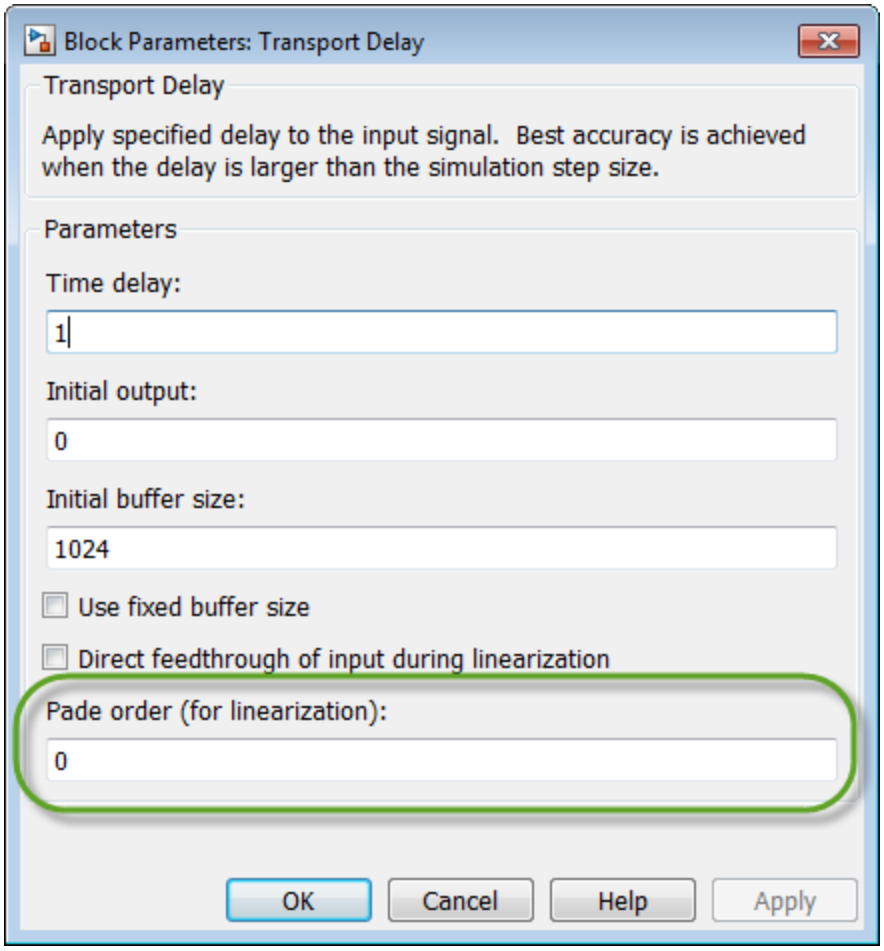
Related Examples

- “Control System Designer Tuning Methods” on page 9-4

Designing Compensators for Plants with Time Delays

You can design compensators for plants with time delays using Simulink Control Design software. When opened from Simulink, **Control System Designer** creates a linear model of your plant. Within this model, you can represent time delays using either Padé approximations or exact delays.

To represent time delays using Padé approximations, specify the Padé order in the Block Parameters dialog box for each Simulink block with delays.



If you do not specify a Padé order for a block, **Control System Designer** uses exact delays by default when possible.

However, the following design methods and plots do not support systems with exact time delays:

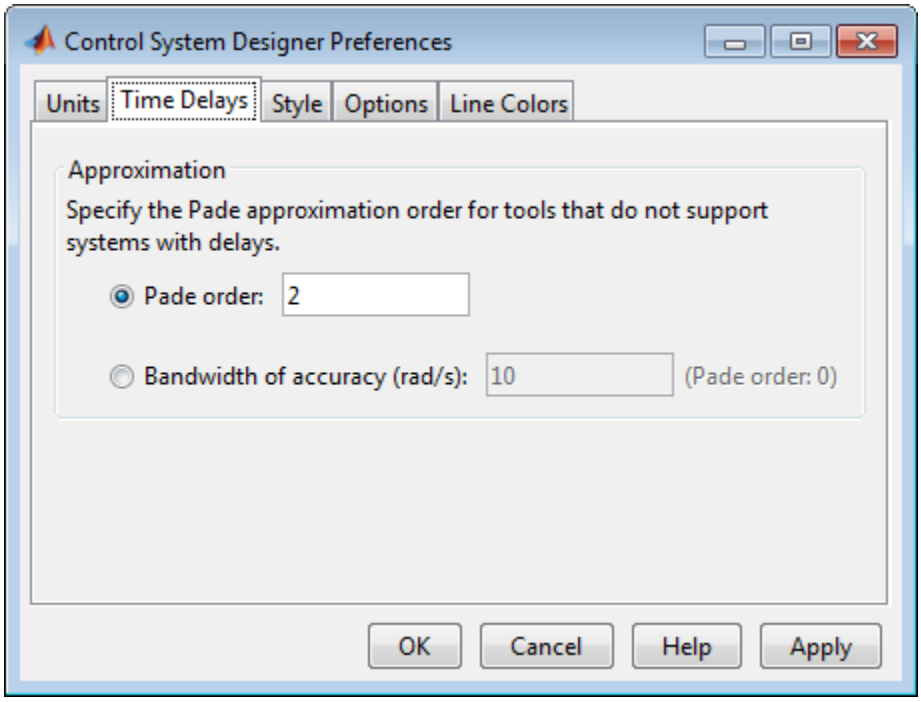
- Root locus plots
- Pole-zero maps
- PID automated tuning
- IMC automated tuning
- LQG automated tuning
- Loop shaping automated tuning

For these design methods and plots, the app automatically computes a Padé approximation for any exact delays in your model using a default Padé order.

To specify the default order, in **Control System Designer**, on the **Control System** tab, click **Preferences**.

In the Control System Designer Preferences dialog box, on the **Time Delays** tab, specify one of the following:

- **Padé order** — Specific Padé order.
- **Bandwidth of accuracy** — Highest frequency at which the approximated response matches the exact response. The app computes and displays the corresponding Padé order.



For more information on designing compensators using **Control System Designer**, see “Control System Designer Tuning Methods” on page 9-4.

See Also

pade

More About

- “Choose a Control Design Approach” on page 9-2
- “What Blocks Are Tunable?” on page 9-8

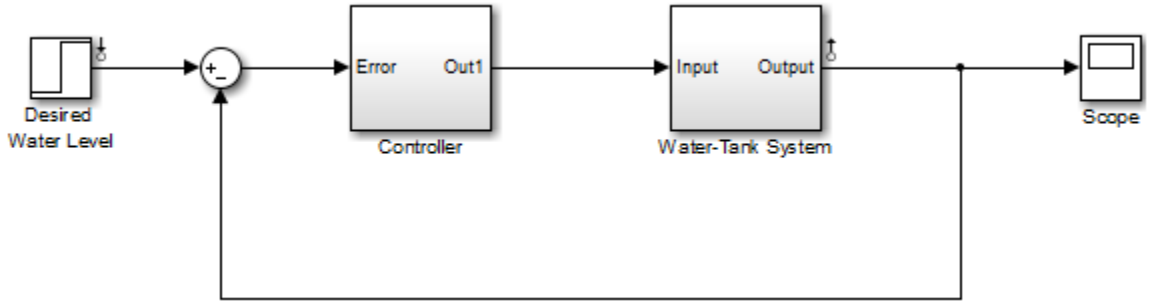
Design Compensator Using Automated PID Tuning and Graphical Bode Design

This example shows how to design a compensator for a Simulink model using automated PID tuning in the **Control System Designer** app. It then shows how to fine tune the compensator design using the open-loop Bode editor.

System Model

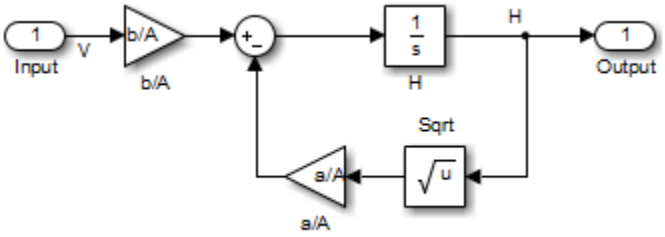
This example uses the `watertank_comp_design` Simulink model. To open the model, at the MATLAB command line, enter:

```
open_system('watertank_comp_design')
```

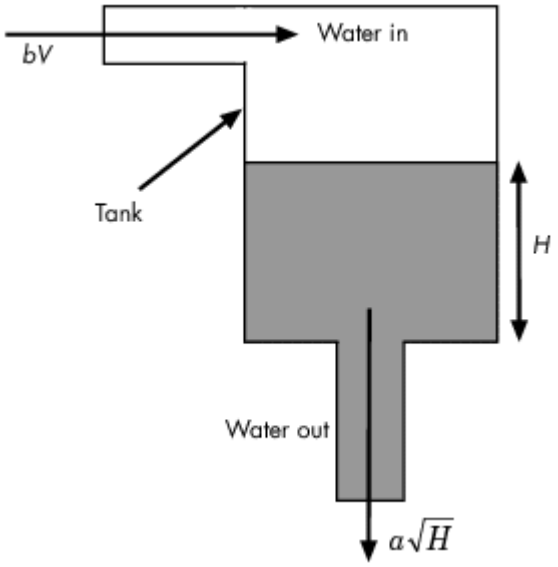


This model contains a Water-Tank System plant model and a PID controller in a single-loop feedback system.

To view the water tank model, open the Water-Tank System subsystem.



This model represents the following water tank system.



Here:

- H is the height of water in the tank.
- Vol is the volume of water in the tank.
- V is the voltage applied to the pump.
- A is the cross-sectional area of the tank.
- b is a constant related to the flow rate into the tank.
- a is a constant related to the flow rate out of the tank.

Water enters the tank from the top at a rate proportional to the voltage applied to the pump. The water leaves through an opening in the tank base at a rate that is proportional to the square root of the water height in the tank. The presence of the square root in the water flow rate results in a nonlinear plant. Based on these flow rates, the rate of change of the tank volume is:

$$\frac{d}{dt}Vol = A\frac{dH}{dt} = bV - a\sqrt{H}$$

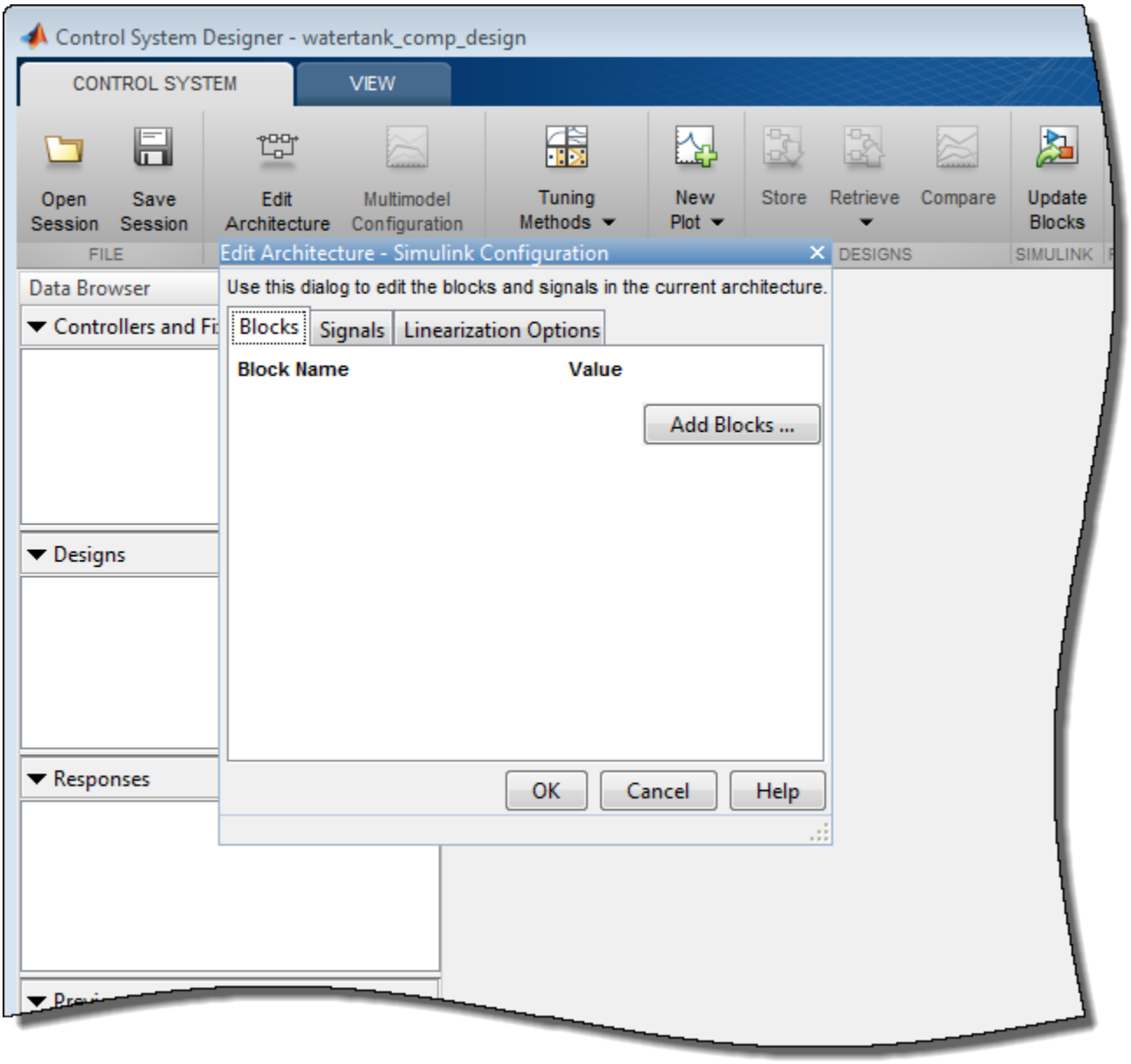
Design Requirements

Tune the PID controller to meet the following closed-loop step response design requirements:

- Overshoot less than 5%
- Rise time less than five seconds

Open Control System Designer

To open **Control System Designer**, in the Simulink model window, in the **Apps** gallery, click **Control System Designer**.

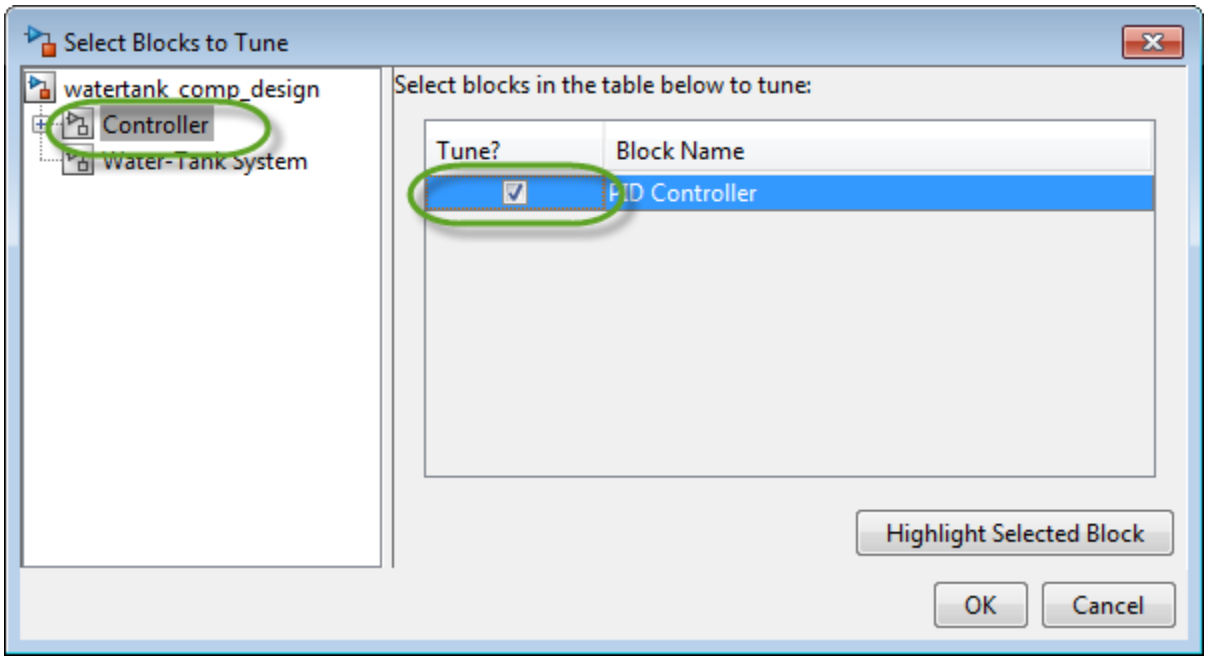


Control System Designer opens and automatically opens the Edit Architecture dialog box.

Specify Blocks to Tune

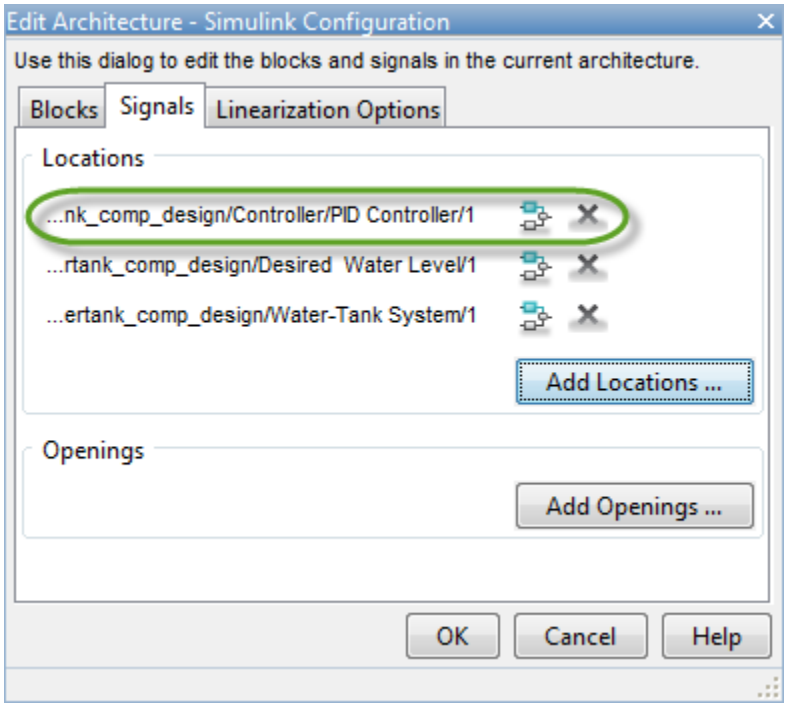
To specify the compensator to tune, in the Edit Architecture dialog box, click **Add Blocks**.

In the Select Blocks to Tune dialog box, in the left pane, click the Controller subsystem. In the **Tune** column, check the box for the PID Controller.



Click **OK**.

In the Edit Architecture dialog box, the app adds the selected controller block to the list of blocks to tune on the **Blocks** tab. On the **Signals** tab, the app also adds the output of the PID Controller block to the list of analysis point **Locations**.



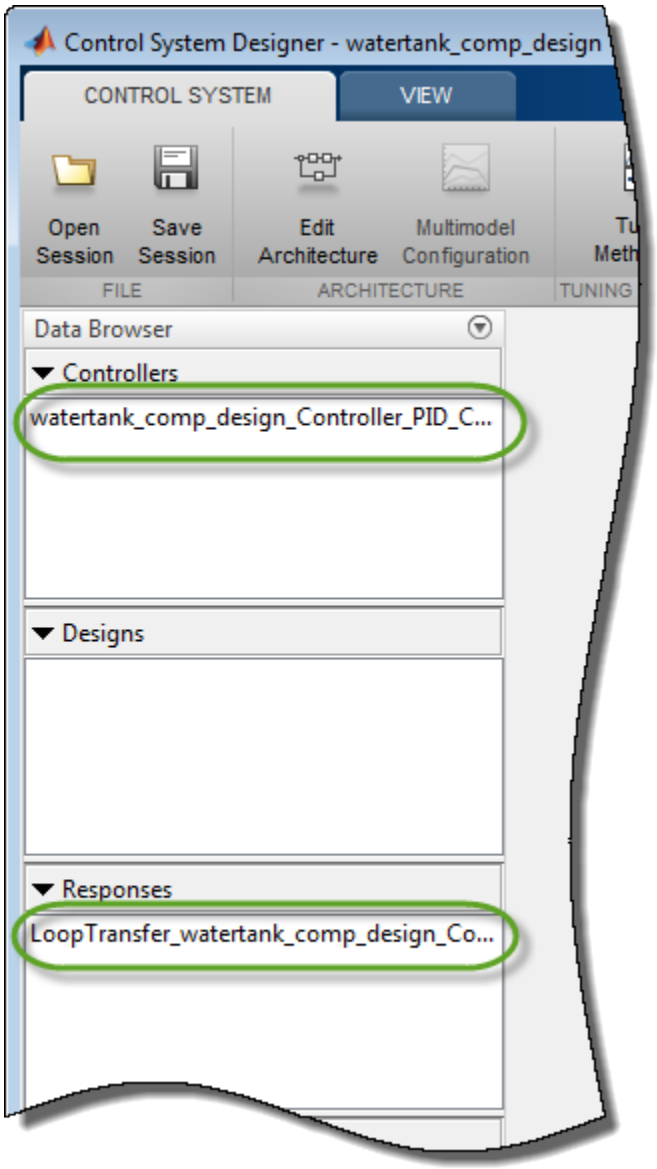
When **Control System Designer** opens, it adds any analysis points previously defined in the Simulink model to the **Locations** list. For the `watertank_comp_design`, there are two such signals.

- Desired Water Level block output — Reference signal for the closed-loop step response
- Water-Tank System block output — Output signal for the closed-loop step response

To linearize the Simulink model and set the control architecture, click **OK**.

By default, **Control System Designer** linearizes the plant model at the model initial conditions.

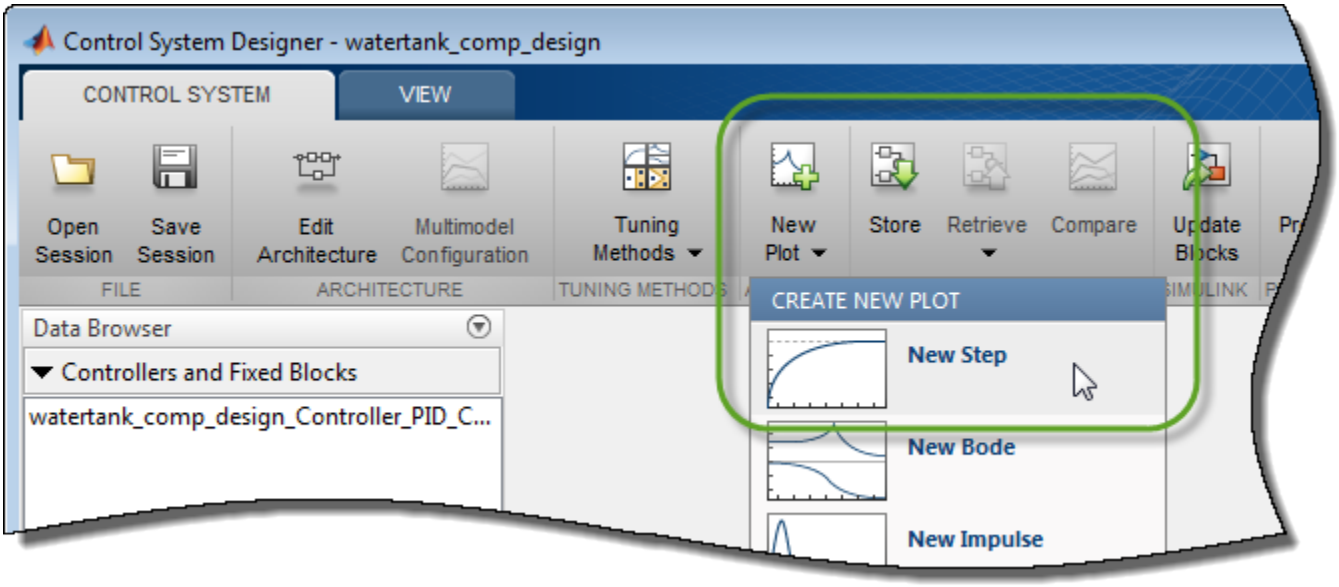
The app adds the PID controller to the **Data Browser**, in the **Controllers and Fixed Blocks** section. The app also computes the open-loop transfer function at the output of the PID Controller block and adds this response to the **Data Browser**.



Plot Closed-Loop Step Response

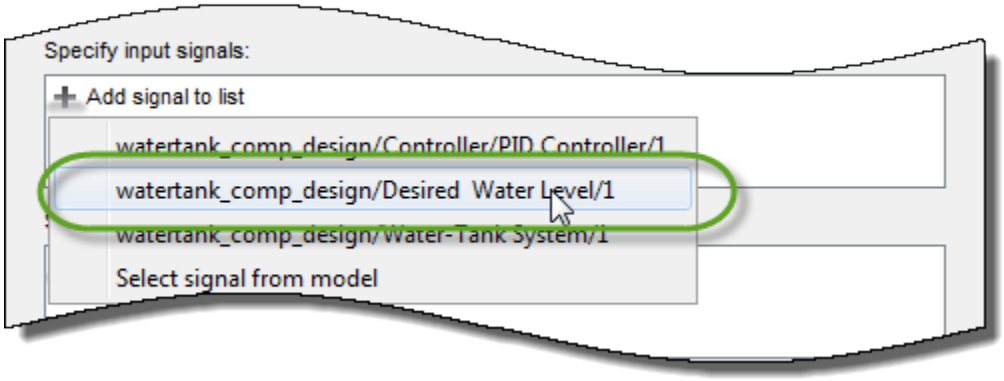
To analyze the controller design, create a closed-loop transfer function of the system and plot its step response.

On the **Control System** tab, click **New Plot**, and select `New Step`.

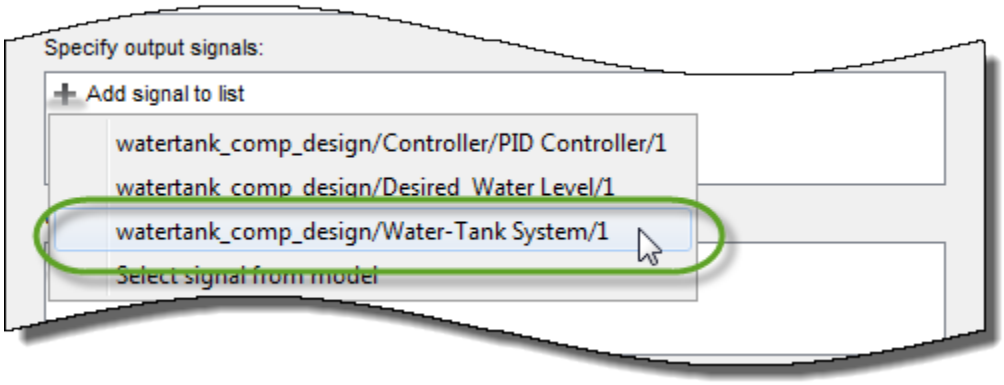


In the New Step to plot dialog box, in the **Select Response to Plot** drop-down list, select `New Input-Output Transfer Response`.

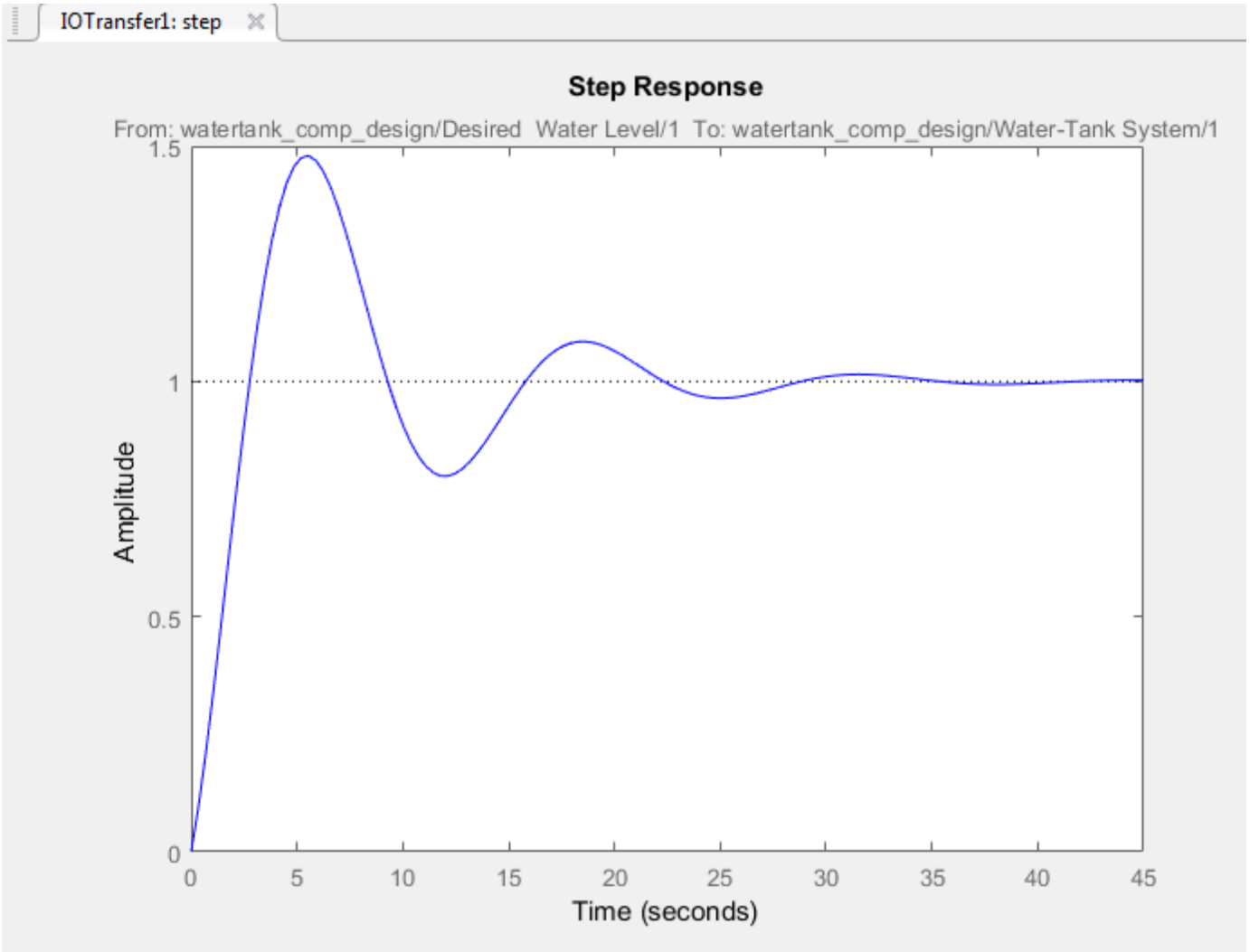
To add an input signal, in the **Specify input signals** area, click **+**. In the drop-down list, select the output of the Desired Water Level block.



To add an output signal, in the **Specify output signals** area, click **+**. In the drop-down list, select the output of the Water-Tank System block.

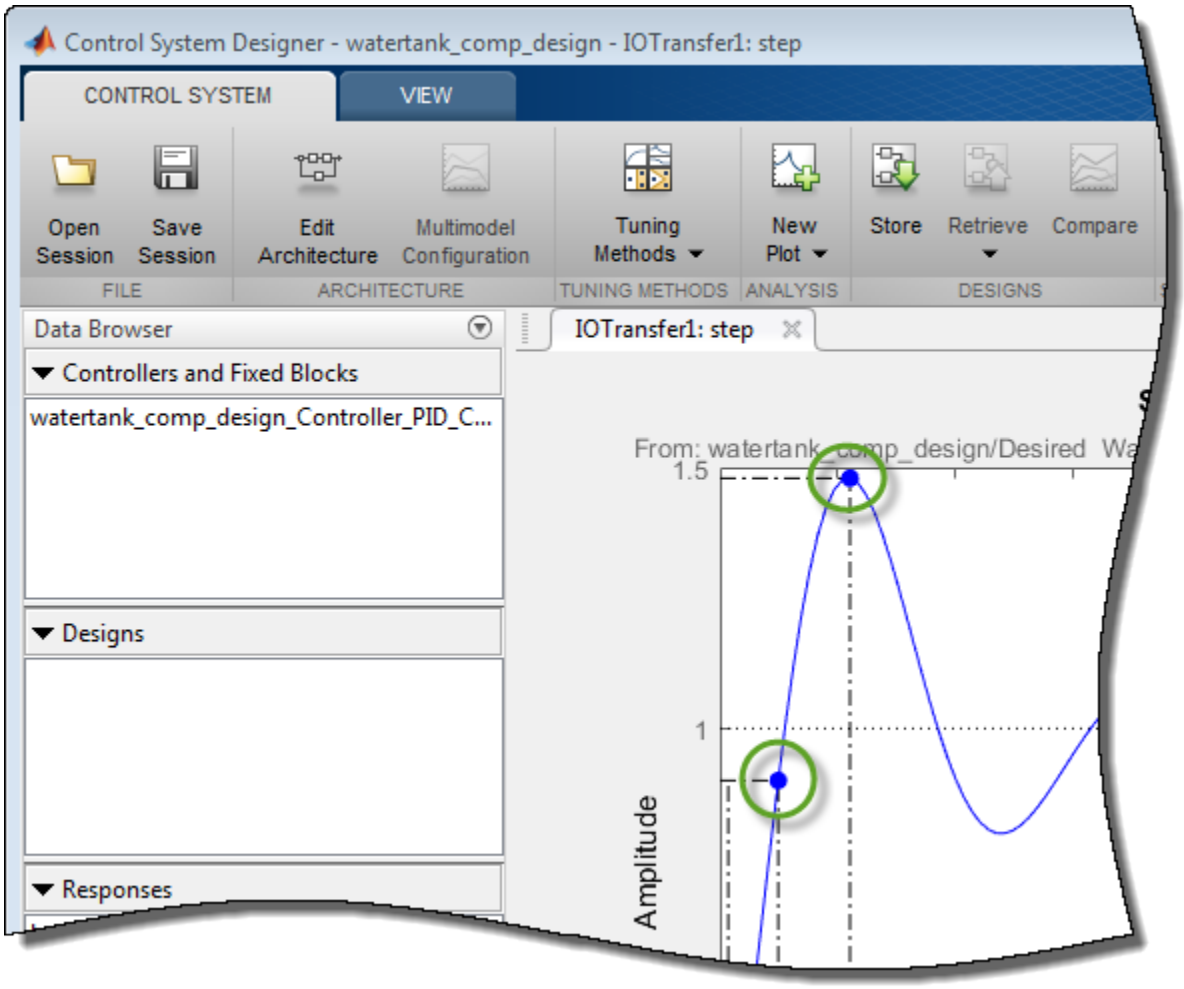


To create the closed-loop transfer function and plot the step response, click **Plot**.



To view the maximum overshoot on the response plot, right-click the plot area, and select **Characteristics > Peak Response**.

To view the rise time on the response plot, right-click the plot area, and select **Characteristics** > **Rise Time**.



Mouse-over the characteristic indicators to view their values. The current design has a:

- Maximum overshoot of 47.9%.
- Rise time of 2.13 seconds.

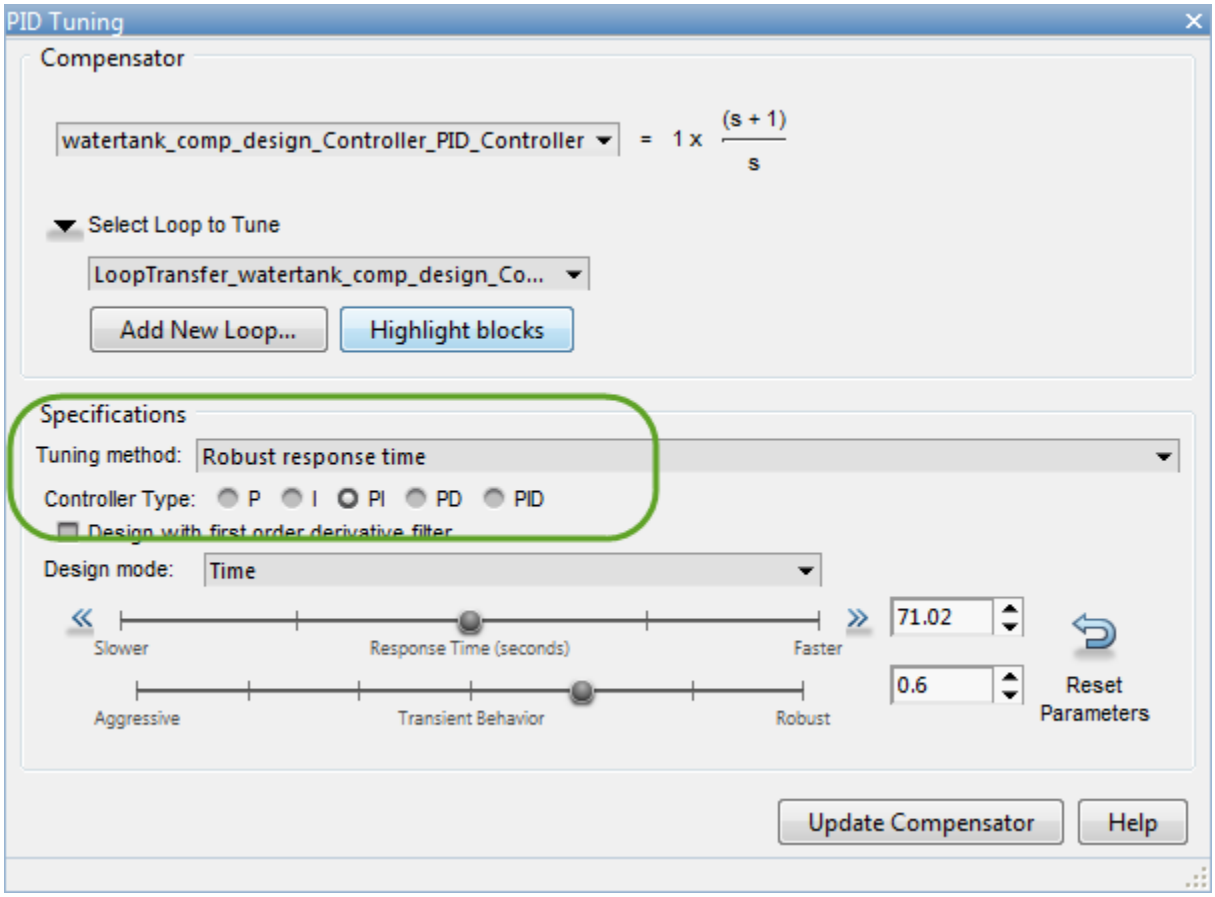
This response does not satisfy the 5% overshoot design requirement.

Tune Compensator Using Automated PID Tuning

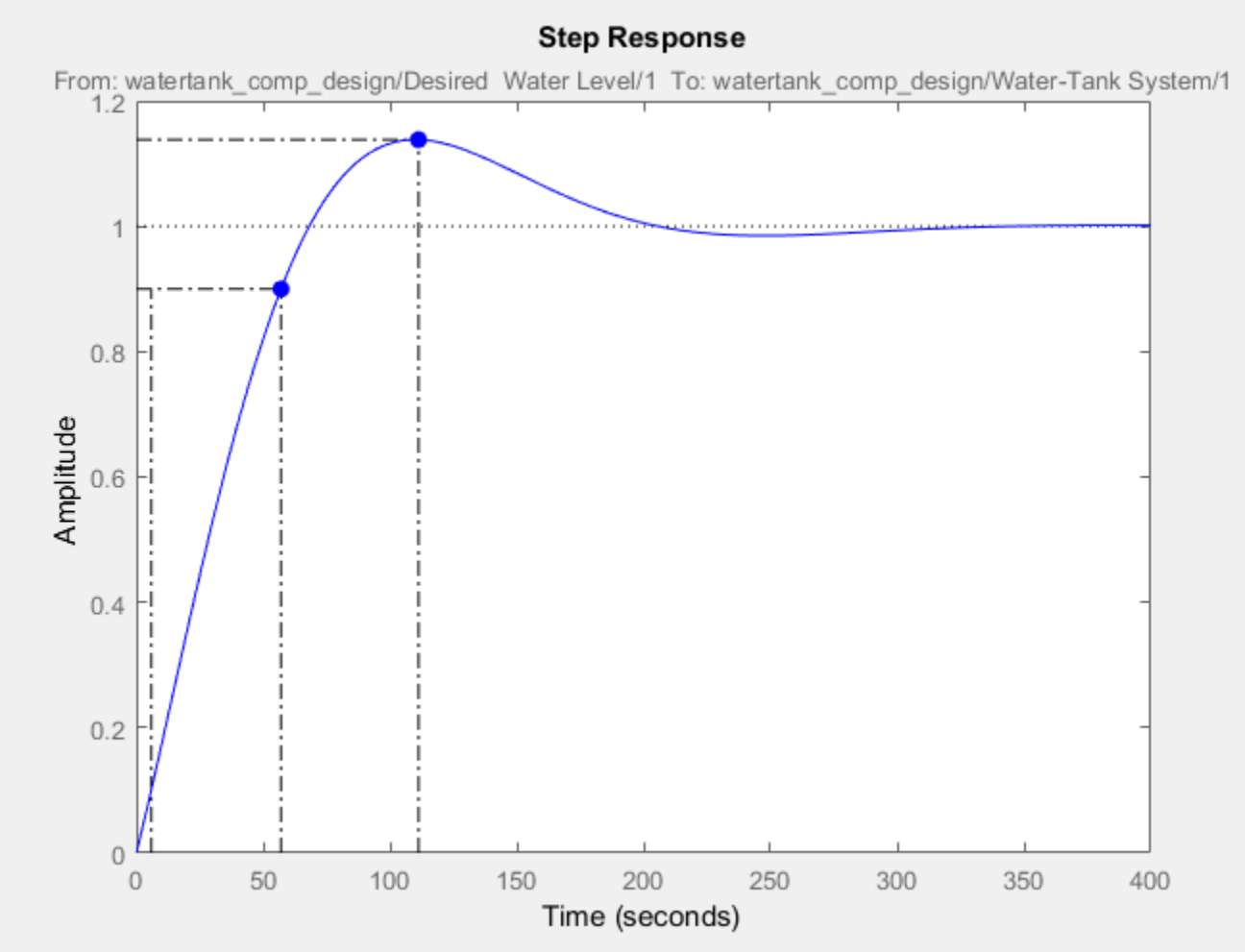
To tune the compensator using automated PID tuning, click **Tuning Methods**, and select `PID Tuning`.

In the PID Tuning dialog box, in the **Specifications** section, select the following options:

- **Tuning method** — `Robust response time`
- **Controller Type** — **PI**



Click **Update Compensator**. The app updates the closed-loop response for the new compensator settings and updates the step response plot.



To check the system performance, mouse over the response characteristic markers. The system response with the tuned compensator has a:

- Maximum overshoot of 13.8%.
- Rise time of 51.2 seconds.

This response exceeds the maximum allowed overshoot of 5%. The rise time is much slower than the required rise time of five seconds.

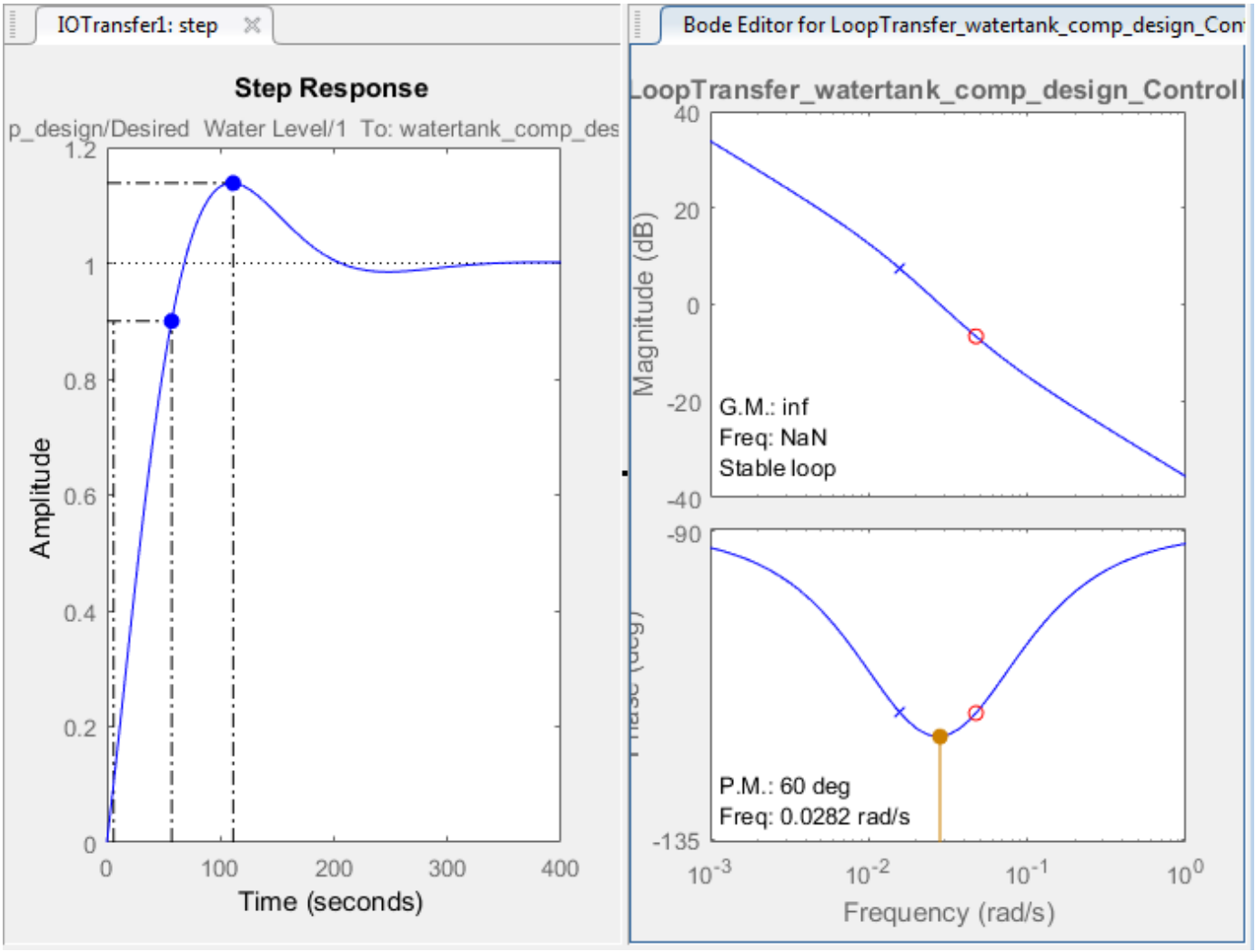
Tune Compensator Using Bode Graphical Tuning

To decrease the rise time, interactively increase the compensator gain using graphical Bode Tuning.

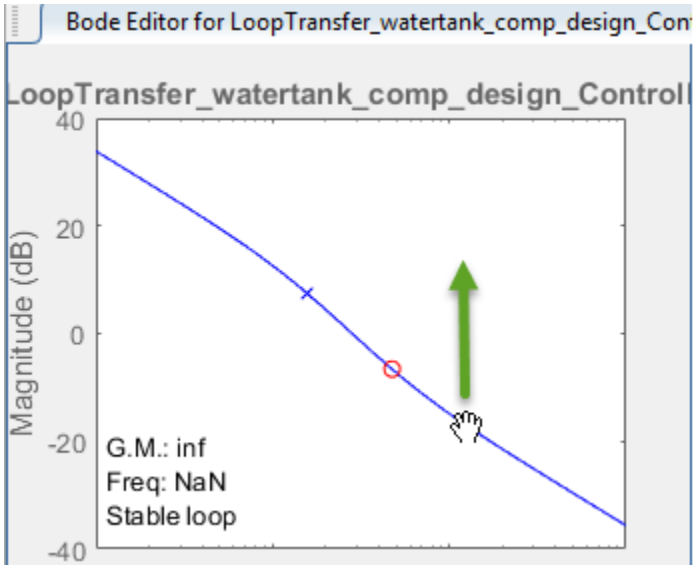
To open the open-loop Bode editor, click **Tuning Methods**, and select `Bode Editor`.

In the Select Response to Edit dialog box, the open-loop response at the output of the PID Controller block is already selected. To open the Bode editor for this response, click **Plot**.

To view the **Bode Editor** and **Step Response** plots side-by-side, on the **View** tab, click **Left/Right**.

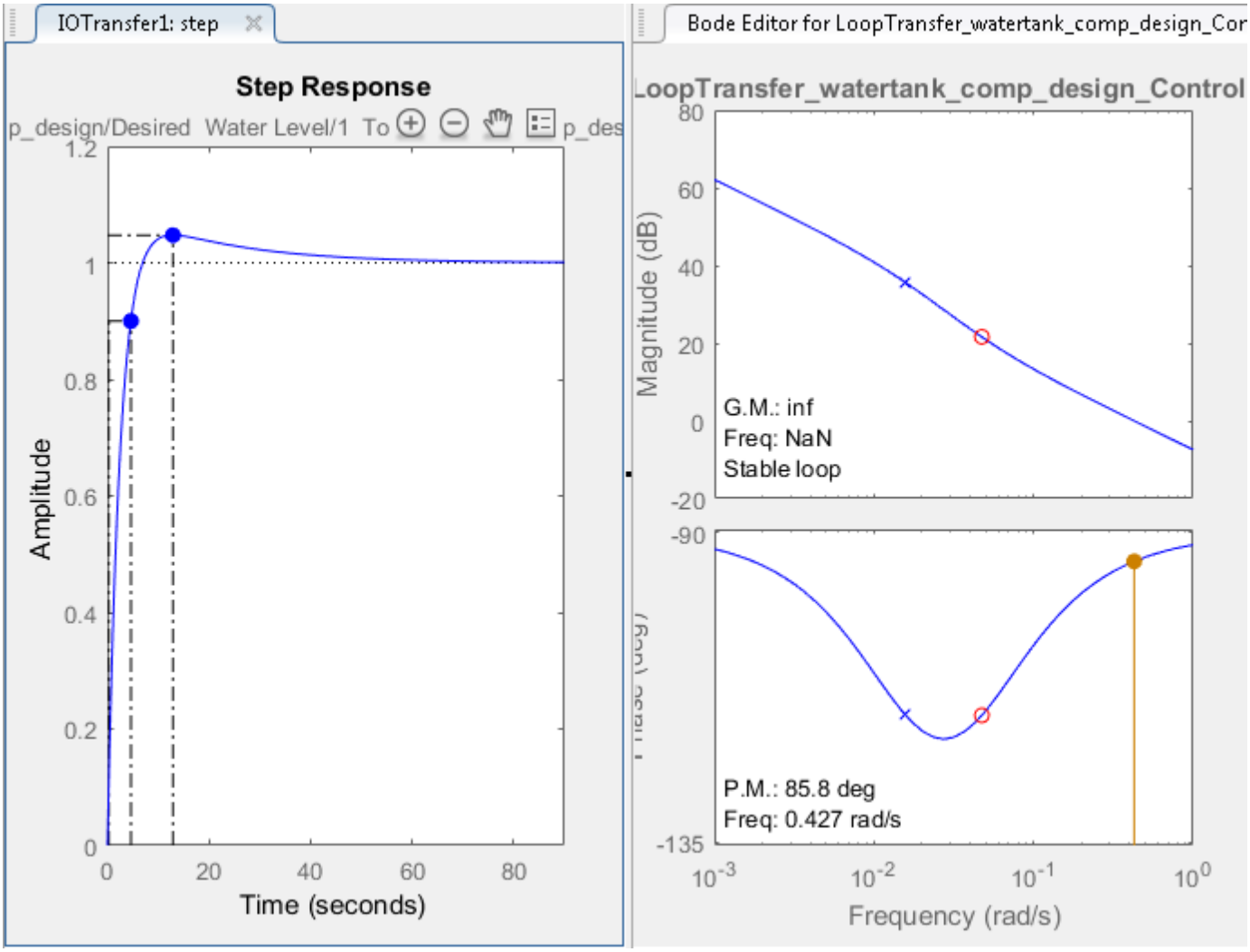


In the **Bode Editor** plot, drag the magnitude response up to increase the compensator gain. By increasing the gain, you increase the bandwidth and speed up the response.



As you drag the Bode response upward, the app automatically updates the compensator and the associated response plots. Also, when you release the plot, in the status bar, on the right side, the app displays the updated gain value.

Increase the compensator gain until the step response meets the design requirements. One potential solution is to set the gain to `1.7`.



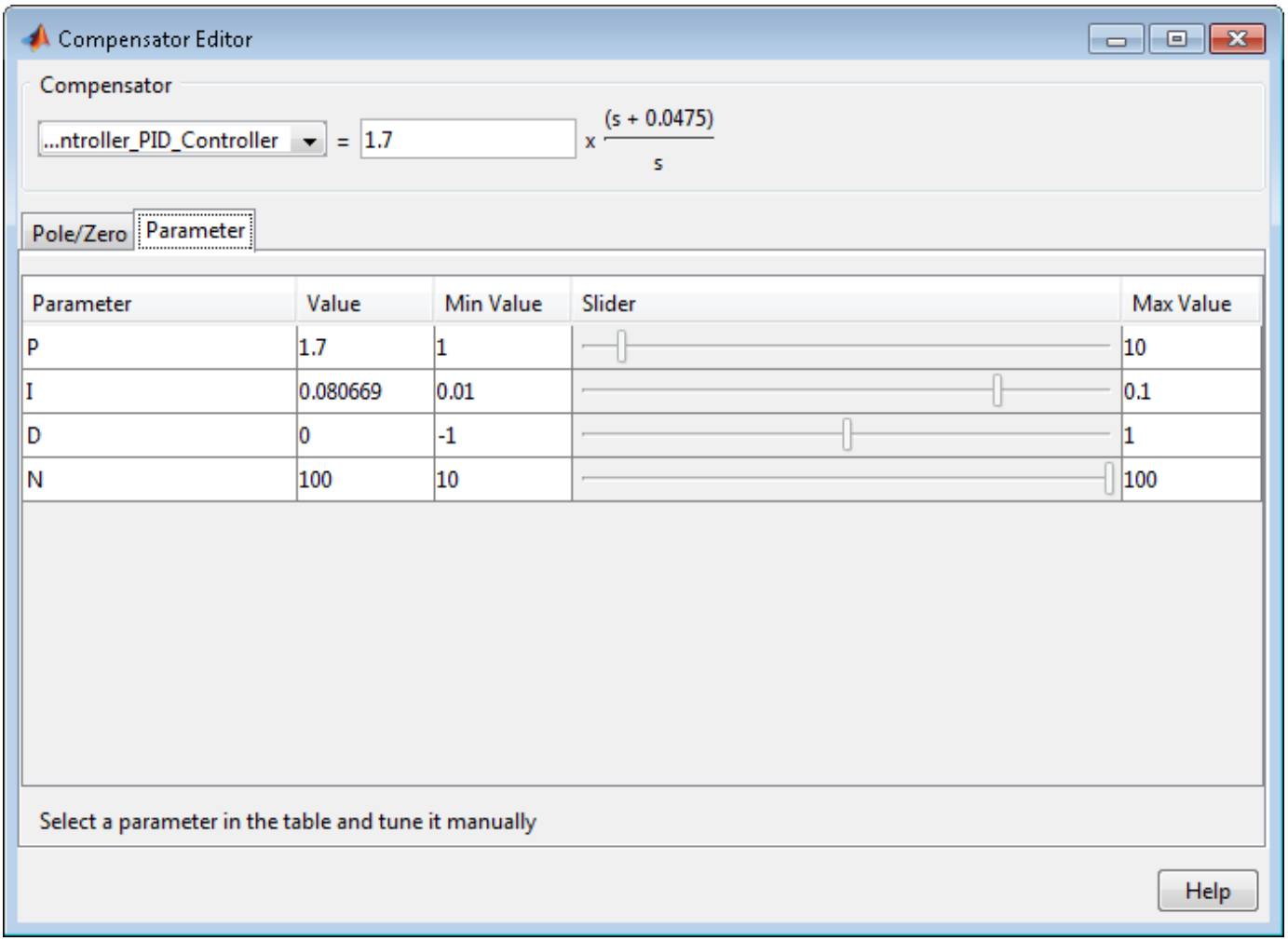
At this gain value, the closed loop response has a:

- Maximum overshoot of 4.74%.
- Rise time of 4.36 seconds.

Fine Tune Controller Using Compensator Editor

To tune the parameters of your compensator directly, use the compensator editor. In the **Bode Editor**, right-click the plot area, and select **Edit Compensator**.

In the Compensator Editor dialog box, on the **Parameter** tab, tune the PID controller gains. For more information on editing compensator parameters, see “Tune Simulink Blocks Using Compensator Editor” on page 9-67.



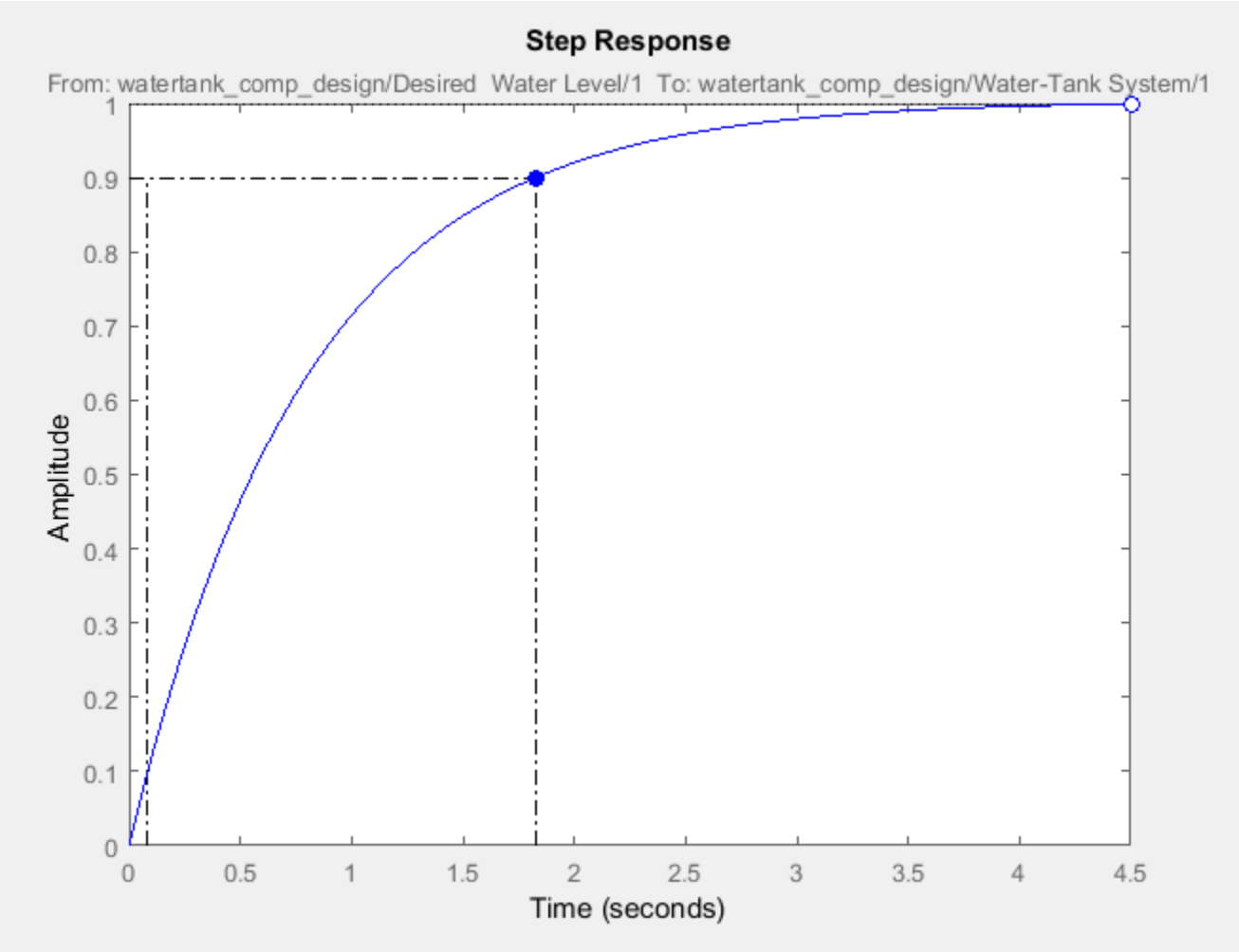
While the tuned compensator meets the design requirements, the settling time is over 30 seconds. To improve the settling time, adjust the **P** and **I** parameters of the controller manually.

For example, set the compensator parameters to:

- **P** = 4
- **I** = 0.1

This compensator produces a closed-loop response with a:

- Maximum overshoot of 0.206%.
- Rise time of 1.74 seconds.
- Settling time of around three seconds.



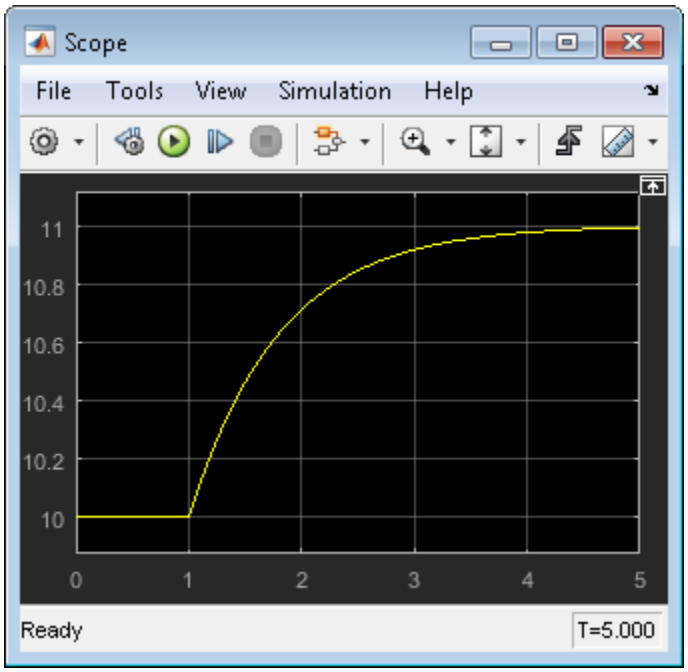
Simulate Closed-Loop System in Simulink

Validate your compensator design by simulating the nonlinear Simulink model with the tuned controller parameters.

To write the tuned compensator parameters to the PID Controller block, in **Control System Designer**, on the **Control System** tab, click **Update Blocks**.

In the Simulink model window, run the simulation.

To view the closed-loop simulation output, double-click the Scope block.



The closed-loop response of the nonlinear system satisfies the design requirements with a rise time of less than five seconds and minimal overshoot.

See Also

Control System Designer

More About

- “Control System Designer Tuning Methods” on page 9-4
- “Analyze Designs Using Response Plots” on page 9-27
- “Update Simulink Model and Validate Design” on page 9-40

Analyze Designs Using Response Plots

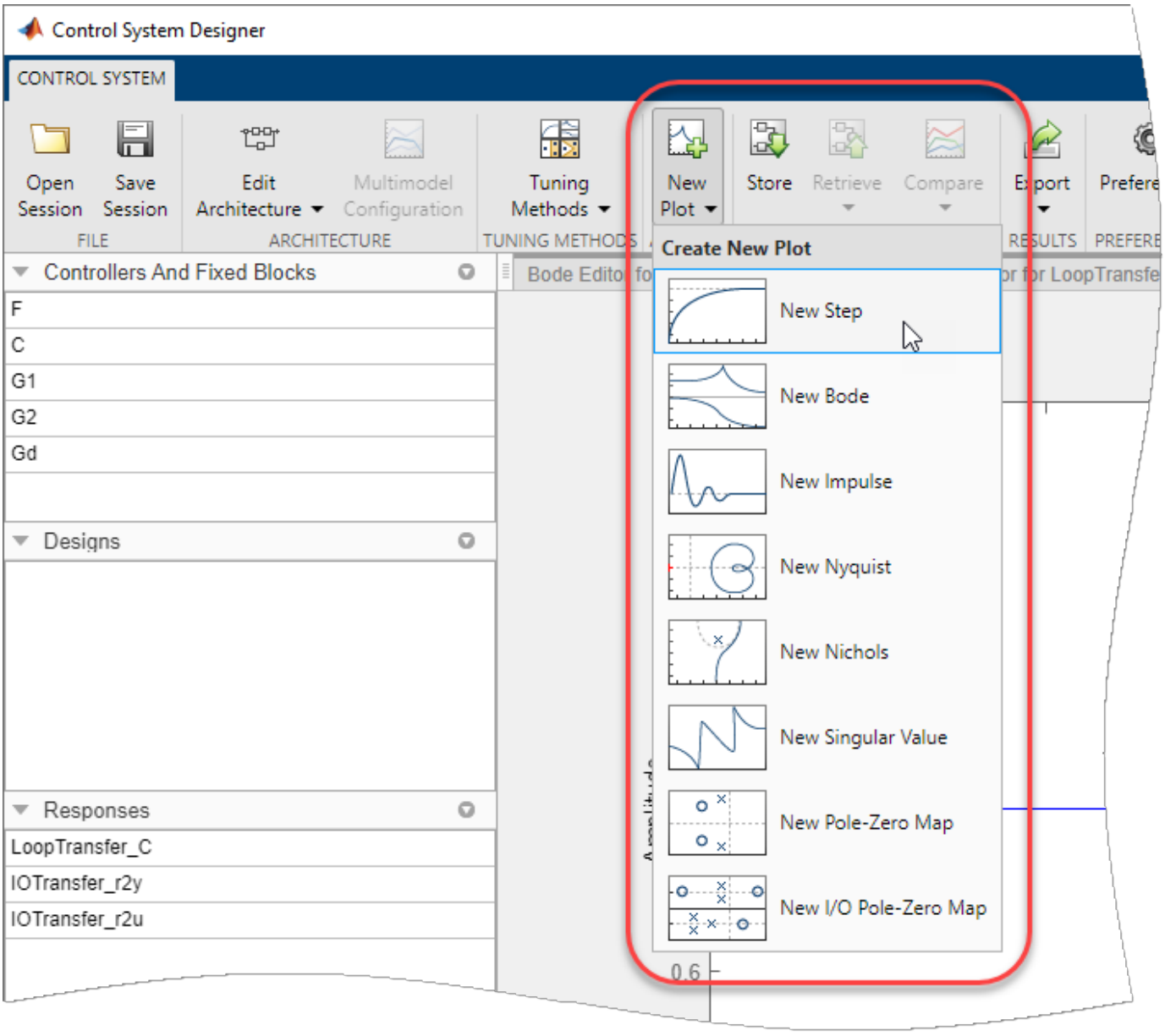
This example shows how to analyze your control system designs using the plotting tools in **Control System Designer**. There are two types of **Control System Designer** plots:

- Analysis plots — Use these plots to visualize your system performance and display response characteristics.
- Editor plots — Use these plots to visualize your system performance and interactively tune your compensator dynamics using graphical tuning methods.

Analysis Plots

Use analysis plots to visualize your system performance and display response characteristics. You can also use these plots to compare multiple control system designs. For more information, see “Compare Performance of Multiple Designs” on page 9-36.

To create a new analysis plot, in **Control System Designer**, on the **Control System** tab, click **New Plot**, and select the type of plot to add.



In the new plot dialog box, specify an existing or new response to plot.

Note Using analysis plots, you can compare the performance of multiple designs stored in the **Data Browser**. For more information, see “Compare Performance of Multiple Designs” on page 9-36.

Plot Existing Response

To plot an existing response, in the **Select Response to Plot** drop-down list, select an existing response from the **Data Browser**. The details for the selected response are displayed in the text box.

To plot the selected response, click **Plot**.

Plot New Response

To plot a new response, specify the following:

- **Select Response to Plot** — Select the type of response to create.
  - `New Input-Output Transfer Response` — Create a transfer function response for specified input signals, output signals, and loop openings.
  - `New Open-Loop Response` — Create an open-loop transfer function response at a specified location with specified loop openings.
  - `New Sensitivity Transfer Response` — Create a sensitivity response at a specified location with specified loop openings.
- **Response Name** — Enter a name in the text box.
- Signal selection boxes — Specify signals as inputs, outputs, or loop openings by clicking . If you open **Control System Designer** from:
  - MATLAB — Select a signal using the **Architecture** block diagram for reference.
  - Simulink — Select an existing signal from the current control system architecture, or add a signal from the Simulink model.

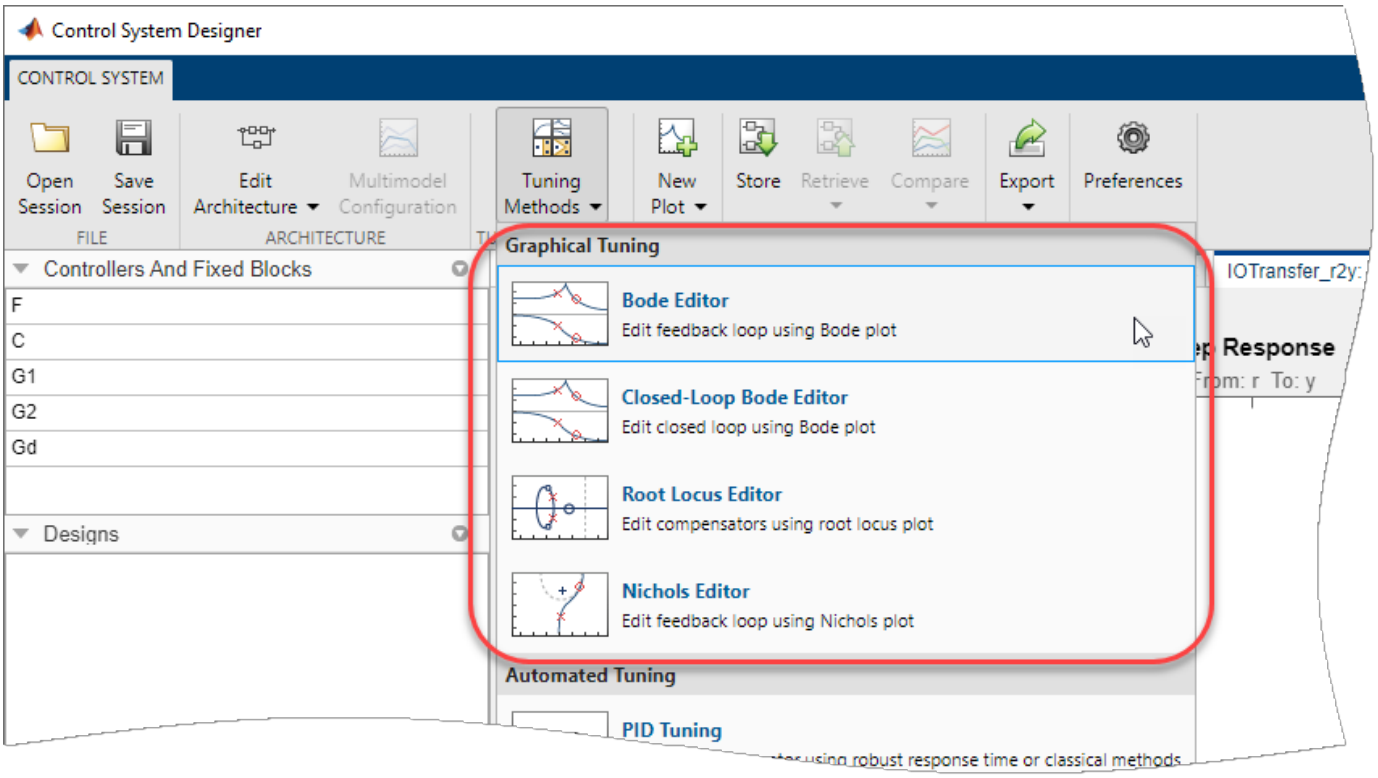
  Use , , and to reorder and delete signals.

To add the specified response to the **Data Browser** and create the selected plot, click **Plot**.

Editor Plots

Use editor plots to visualize your system performance and interactively tune your compensator dynamics using graphical tuning methods.

To create a new editor plot, in **Control System Designer**, on the **Control System** tab, click **Tuning Methods**, and select one of the **Graphical Tuning** methods.



For examples of graphical tuning using editor plots, see:

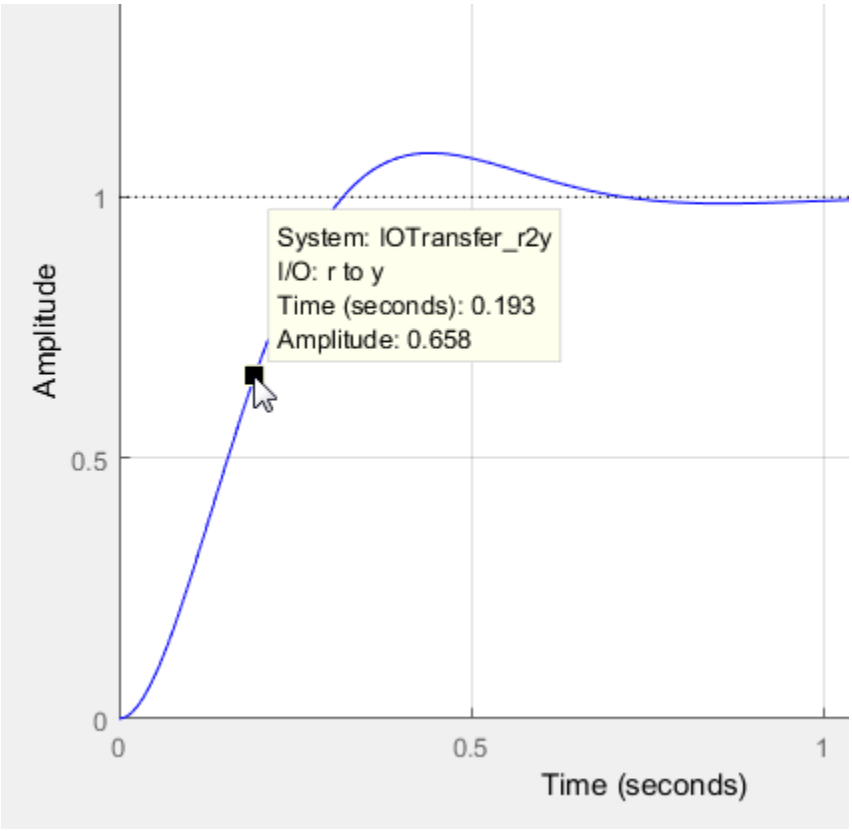
- “Bode Diagram Design”
- “Root Locus Design”
- “Nichols Plot Design”

For more information on interactively editing compensator dynamics, see “Edit Compensator Dynamics”.

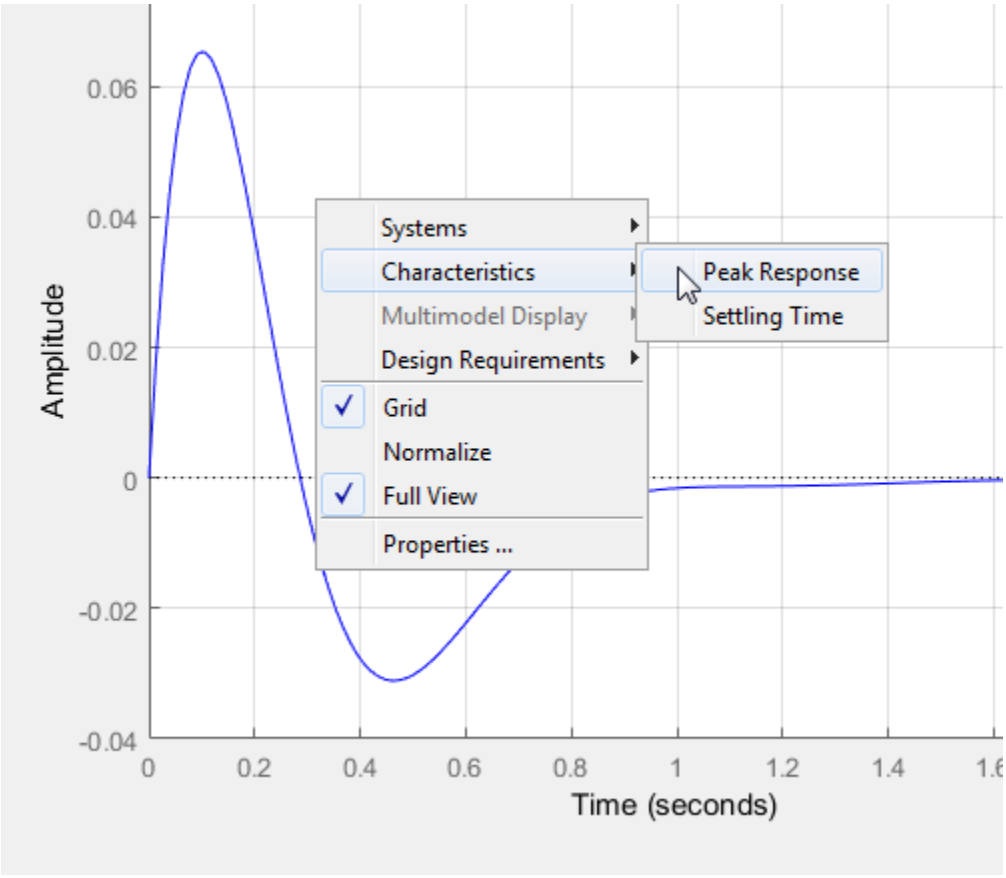
Plot Characteristics

On any analysis plot in **Control System Designer**:

- To see response information and data values, click a line on the plot.

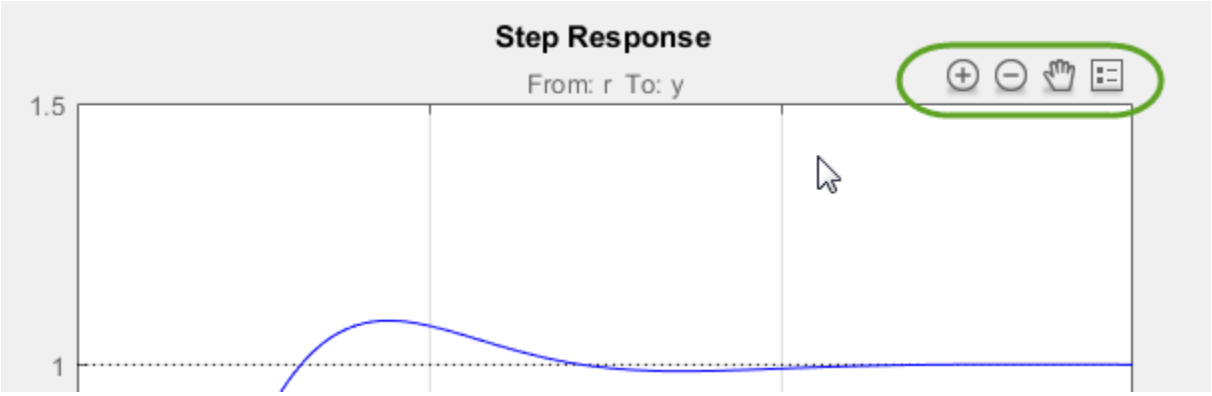


- To view system characteristics, right-click anywhere on the plot, as described in “Frequency-Domain Characteristics on Response Plots”.

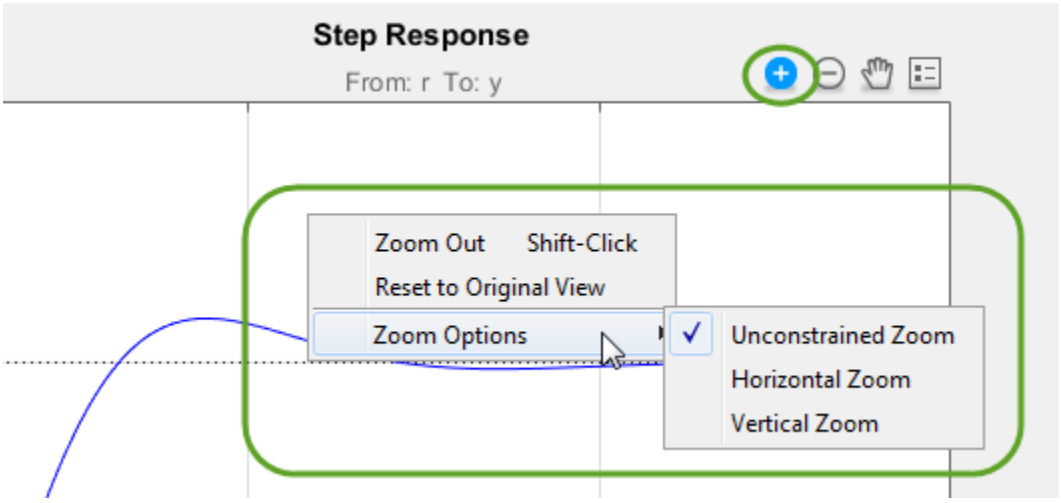


Plot Tools

Mouse over any analysis plot to access plot tools at the upper right corner of the plot.



- and — Zoom in and zoom out. Click to activate, and drag the cursor over the region to zoom. The zoom icon turns dark when zoom is active. Right-click while zoom is active to access additional zoom options. Click the icon again to deactivate.

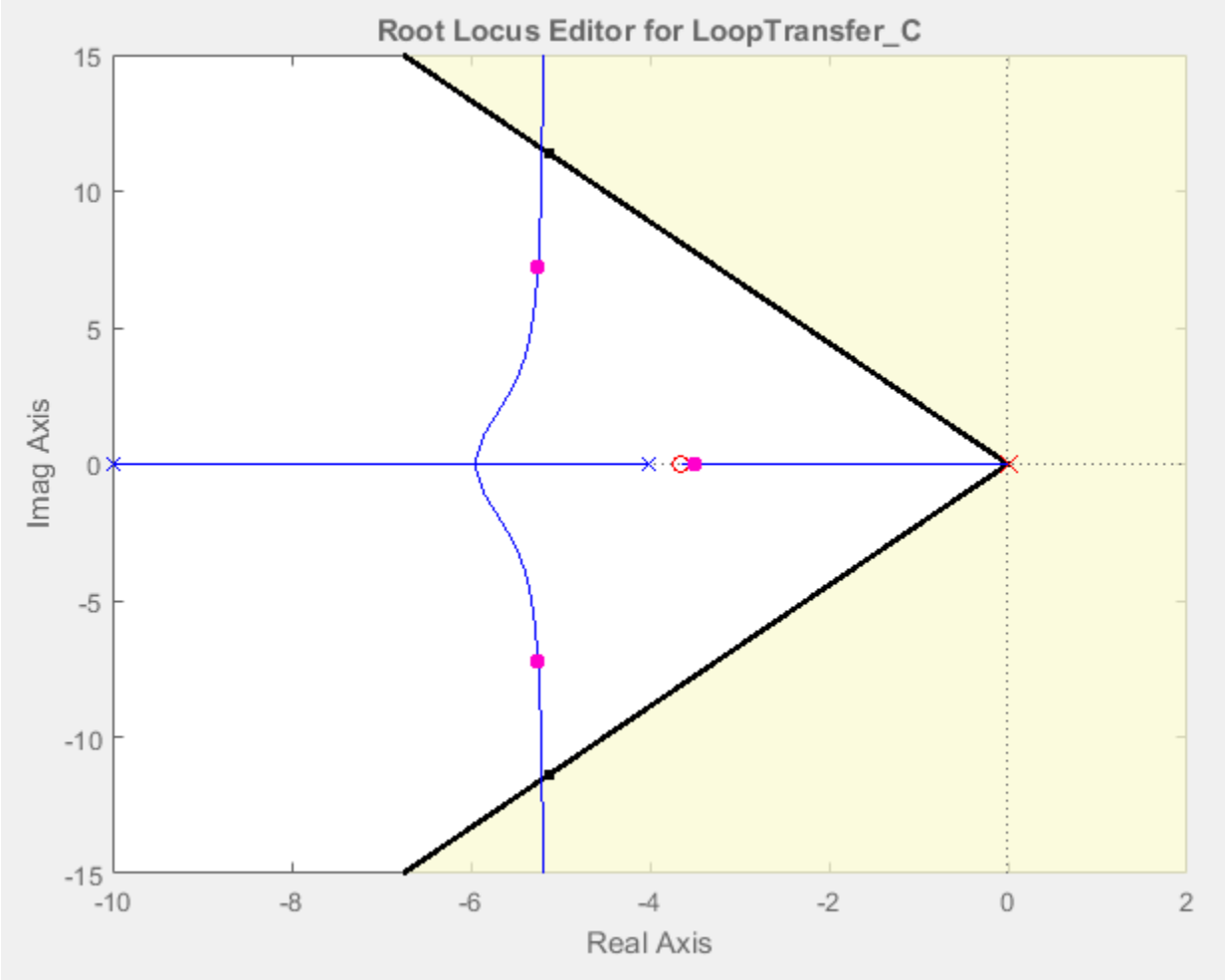


- — Pan. Click to activate, and drag the cursor across the plot area to pan. The pan icon turns dark when pan is active. Right-click while pan is active to access additional pan options. Click the icon again to deactivate.
- — Legend. By default, the plot legend is inactive. To toggle the legend on and off, click this icon. To move the legend, drag it to a new location on the plot.

To change the way plots are tiled or sorted, click and drag the plots to the desired location.

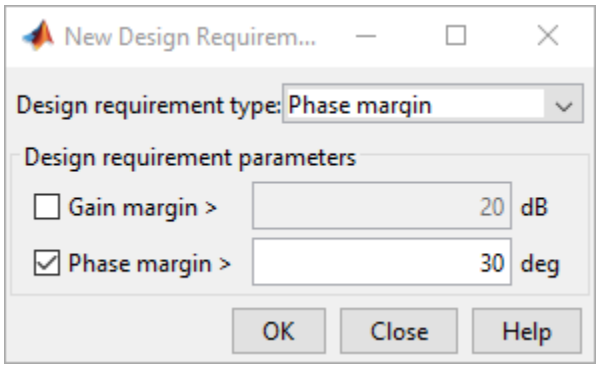
Design Requirements

You can add graphical representations of design requirements to any editor or analysis plots. These requirements define shaded exclusion regions in the plot area.



Use these regions as guidelines when analyzing and tuning your compensator designs. To meet a design requirement, your response plots must remain outside of the corresponding shaded area.

To add design requirements to a plot, right-click anywhere on the plot and select **Design Requirements > New**.



In the New Design Requirement dialog box, specify the **Design requirement type**, and define the **Design requirement parameters**. Each type of design requirement has a different set of

parameters to configure. For more information on adding design requirements to analysis and editor plots, see “Design Requirements”.

See Also

More About

- “Control System Designer Tuning Methods” on page 9-4
- “Compare Performance of Multiple Designs” on page 9-36
- “Design Requirements”

Compare Performance of Multiple Designs

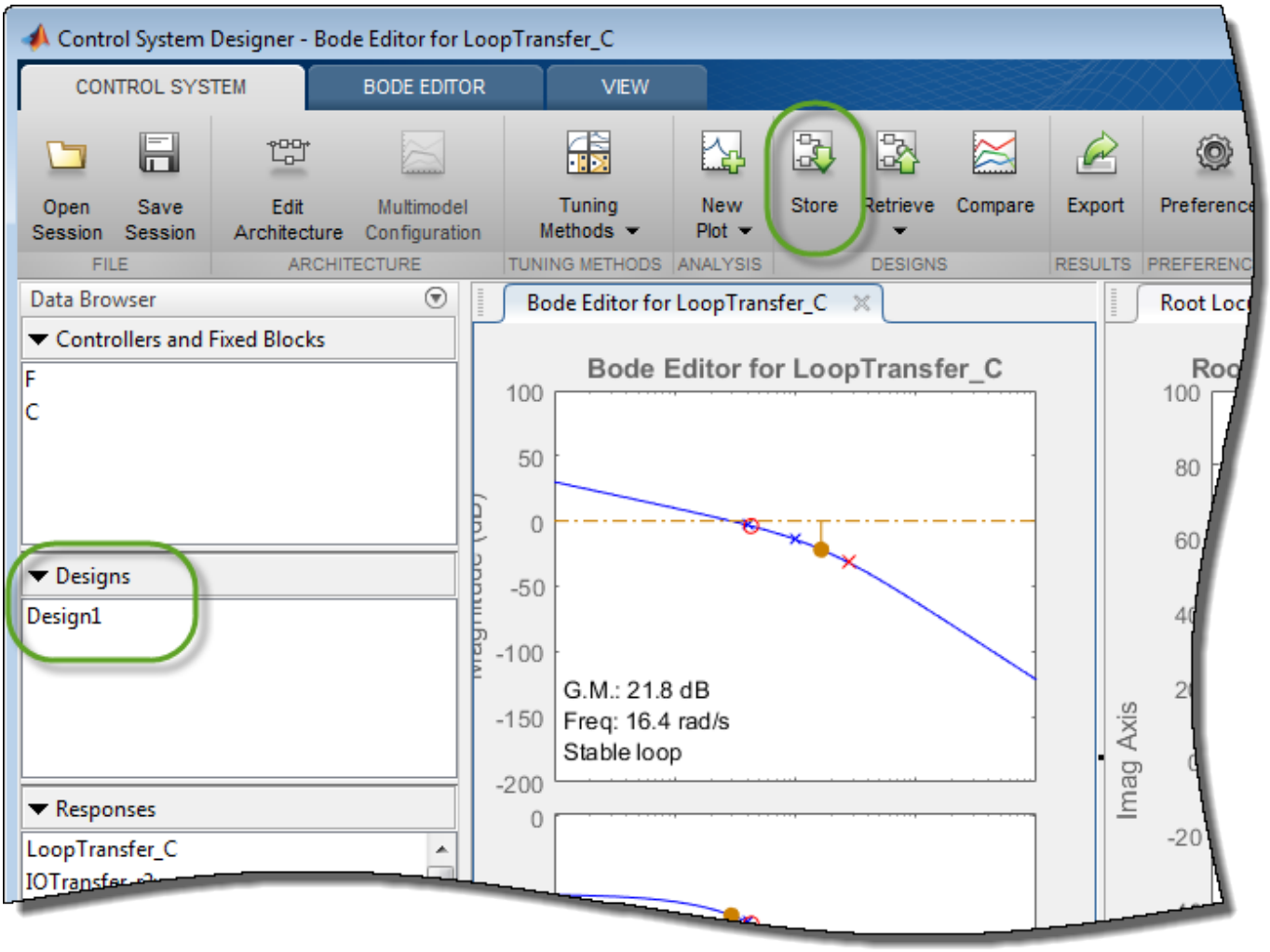
This example shows how to compare the performance of two different control system designs. Such comparison is useful, for example, to see the effects of different tuning methods or compensator structures.

Store First Design

In this example, the first design is the compensator tuned graphically in “Bode Diagram Design”.

After tuning the compensator with this first tuning method, store the design in **Control System Designer**.

On the **Control System** tab, in the **Designs** section, click **Store**. The stored design appears in the **Data Browser** in the **Designs** area.



The stored design contains the tuned values of the controller and filter blocks. The app does not store the values of any fixed blocks.

To rename the stored design, in the **Data Browser**, double-click the design, and specify a new name.

Compute New Design

On the **Control System** tab, tune the compensator using a different tuning method.

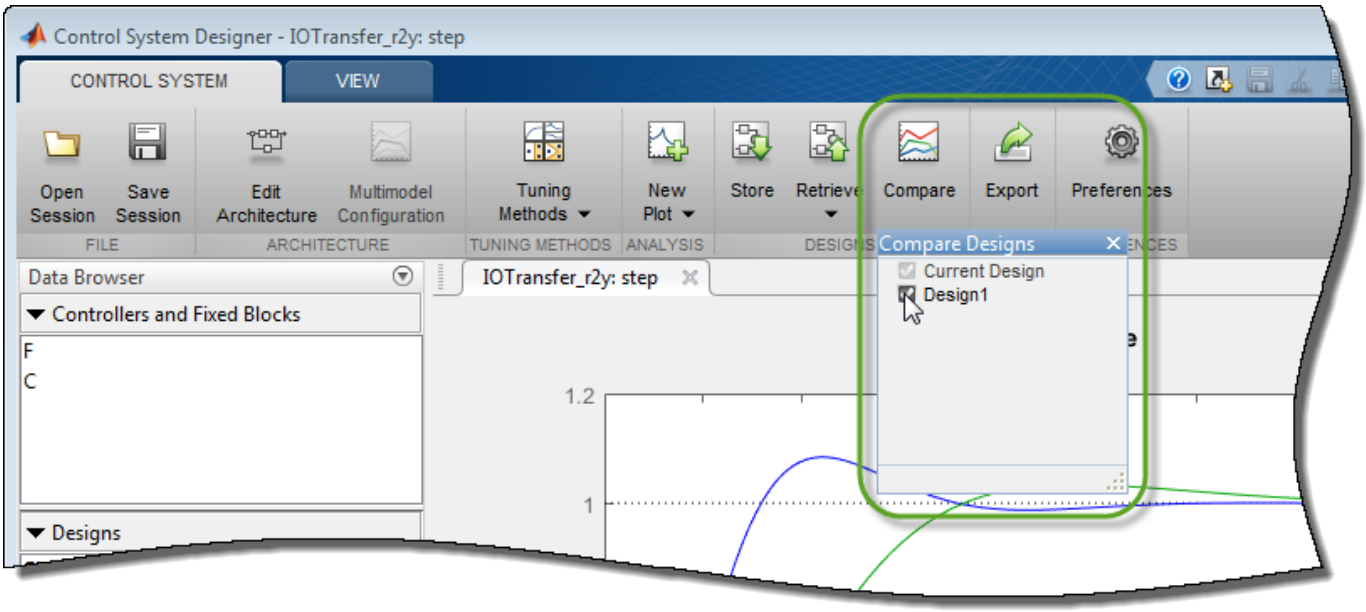
Under **Tuning Methods**, select `PID Tuning`.

To design a controller with the default `Robust response time` specifications, in the PID Tuning dialog box, click **Update Compensator**.

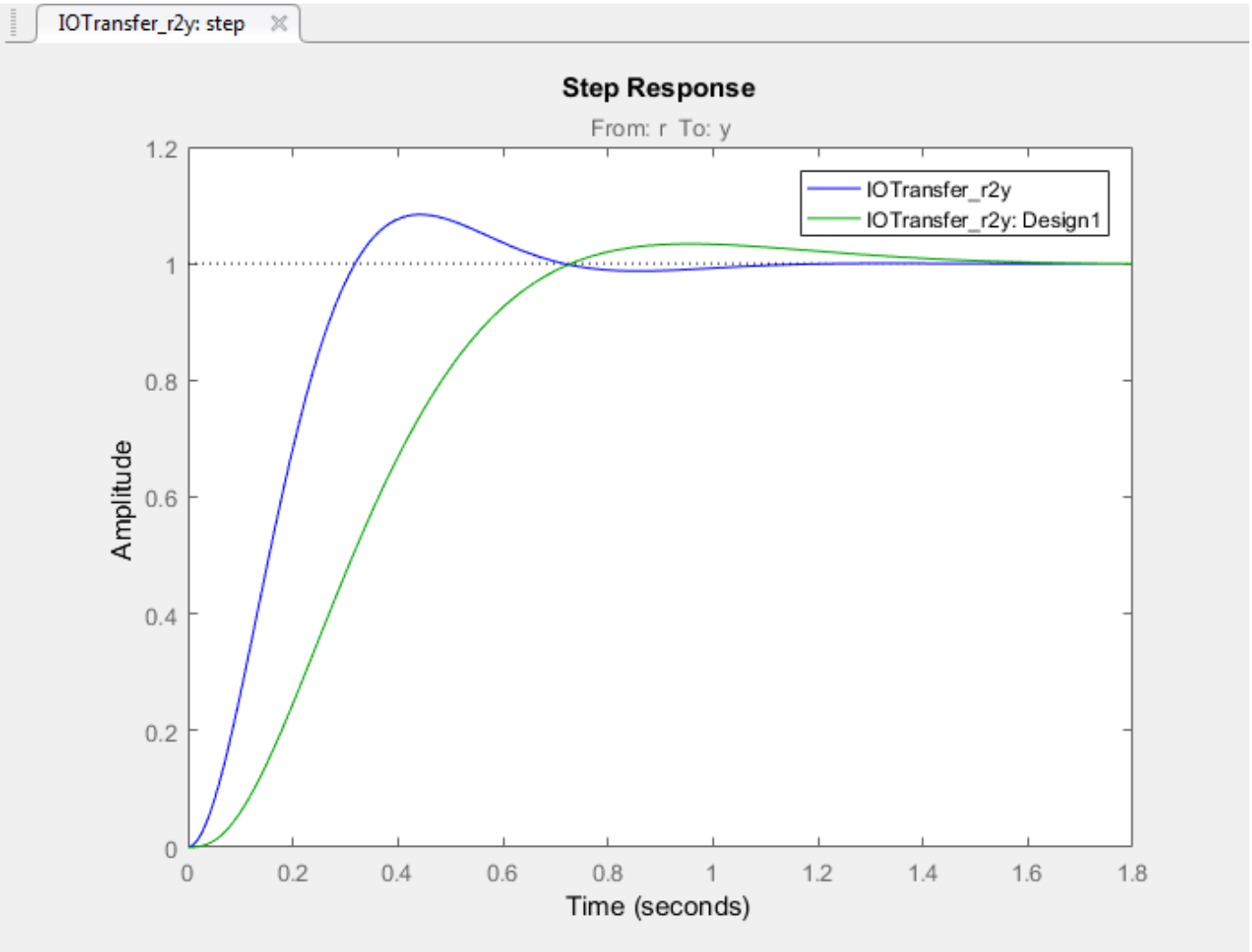
Compare New Design with Stored Design

Update all plots to reflect both the new design and the stored design.

On the **Control System** tab, click **Compare**.



In the **Compare Designs** dialog box, the current design is checked by default. To compare a design with the current design, check the corresponding box. All analysis plots update to reflect the checked designs. The blue trace corresponds to the current design. Refer to the plot legend to identify the responses for other designs.

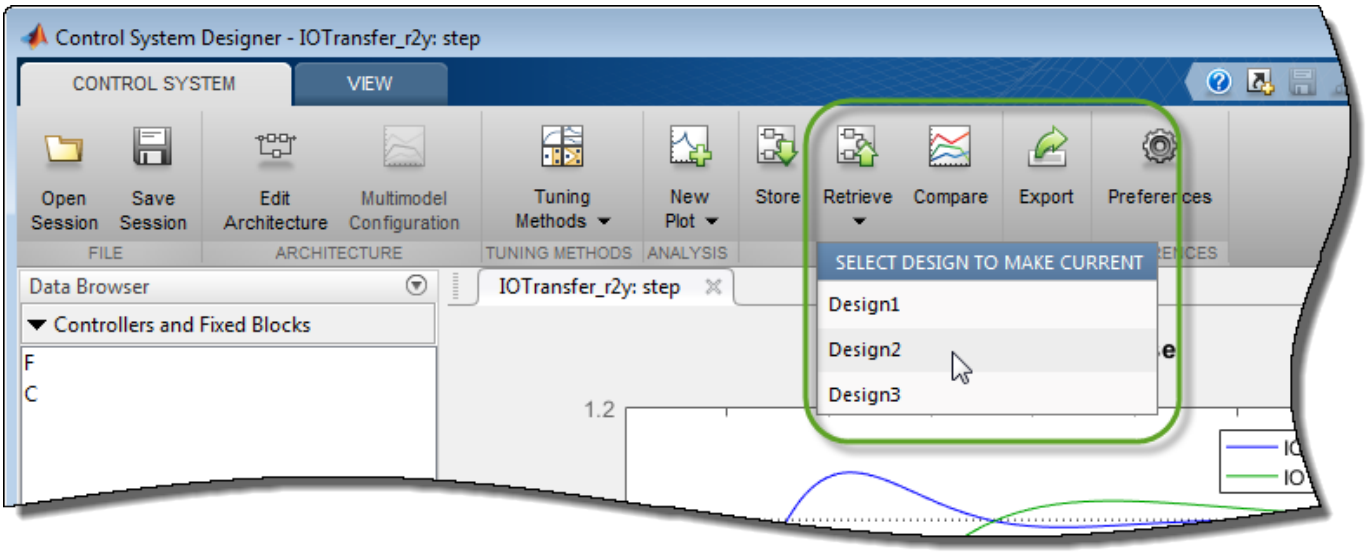


To compare a stored design with the current design, the sample times of the current design and stored design must be the same. To modify the sample time of the current design to match that of a stored design, on the **Control System** tab, click **Edit Architecture**. Then, in the Edit Architecture dialog box, on the **Linearization Options** tab, specify the working domain and rate conversion method.

Restore Previously Saved Design

Under some conditions, it is useful to restore a previously stored design. For example, when designing a compensator for a Simulink model, you can write the current compensator values to the model (see “Update Simulink Model and Validate Design” on page 9-40). To test a stored compensator in your model, first restore the stored design as the current design.

To do so, in **Control System Designer**, click **Retrieve**. Select the stored design that you want to make current.



As with design comparison, to retrieve a stored design, the sample times of the current design and stored design must be the same.

Note The retrieved design overwrites the current design. If necessary, store the current design before retrieving a previously stored design.

See Also

More About

- “Analyze Designs Using Response Plots” on page 9-27
- “Control System Designer Tuning Methods” on page 9-4

Update Simulink Model and Validate Design

This example shows how to update compensator blocks in a Simulink model and validate a control system design.

To tune a control system for a nonlinear Simulink model, **Control System Designer** linearizes the system. Therefore, it is good practice to validate your tuned control system in Simulink.

1. Tune your control system using **Control System Designer**.

   For an example, see “Design Compensator Using Automated PID Tuning and Graphical Bode Design” on page 9-11.
2. Insure that the control system satisfies the design requirements.

   In **Control System Designer**, analyze the controller design. For more information, see “Analyze Designs Using Response Plots” on page 9-27.
3. Write tuned compensator parameters to your Simulink model.

   In **Control System Designer**, on the **Control System** tab, click **Update Blocks**.
4. Simulate the updated model.

   In the Simulink model window, click .
5. Verify whether your compensator satisfies the design requirements when simulated with your nonlinear Simulink model.

See Also

Control System Designer

More About

- “Design Compensator Using Automated PID Tuning and Graphical Bode Design” on page 9-11

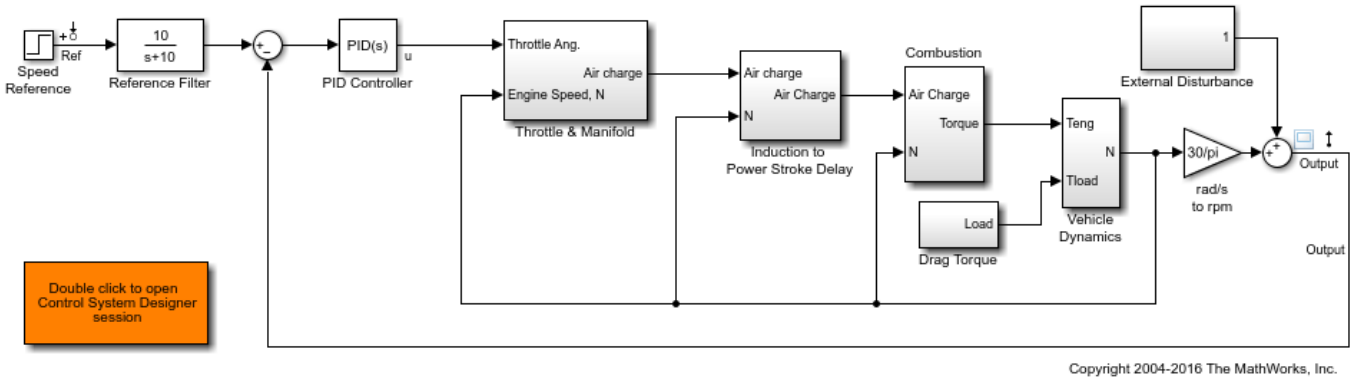
Single Loop Feedback/Prefilter Compensator Design

This example shows how to tune multiple compensators (feedback and prefilter) to control a single loop using **Control System Designer**.

Open the Model

Open the engine speed control model and take a few moments to explore it.

```
open_system('scdspeedctrl')
```



Design Overview

This example introduces the process of designing a single-loop control system with both feedback and prefilter compensators. The goal of the design is to:

- Track the reference signal from a Simulink step block `scdspeedctrl/Speed Reference`. The design requirement is to have a settling time of under 5 seconds and zero steady-state error to the step reference input.
- Reject an unmeasured output disturbance specified in the subsystem `scdspeedctrl/External Disturbance`. The design requirement is to reduce the peak deviation to 190 RPM and to have zero steady-state error for a step disturbance input.

In this example, the stabilization of the feedback loop and the rejection of the output disturbance are achieved by designing the PID compensator `scdspeedctrl/PID Controller`. The prefilter `scdspeedctrl/Reference Filter` is used to tune the response of the feedback system to changes in the reference tracking.

Open Control System Designer

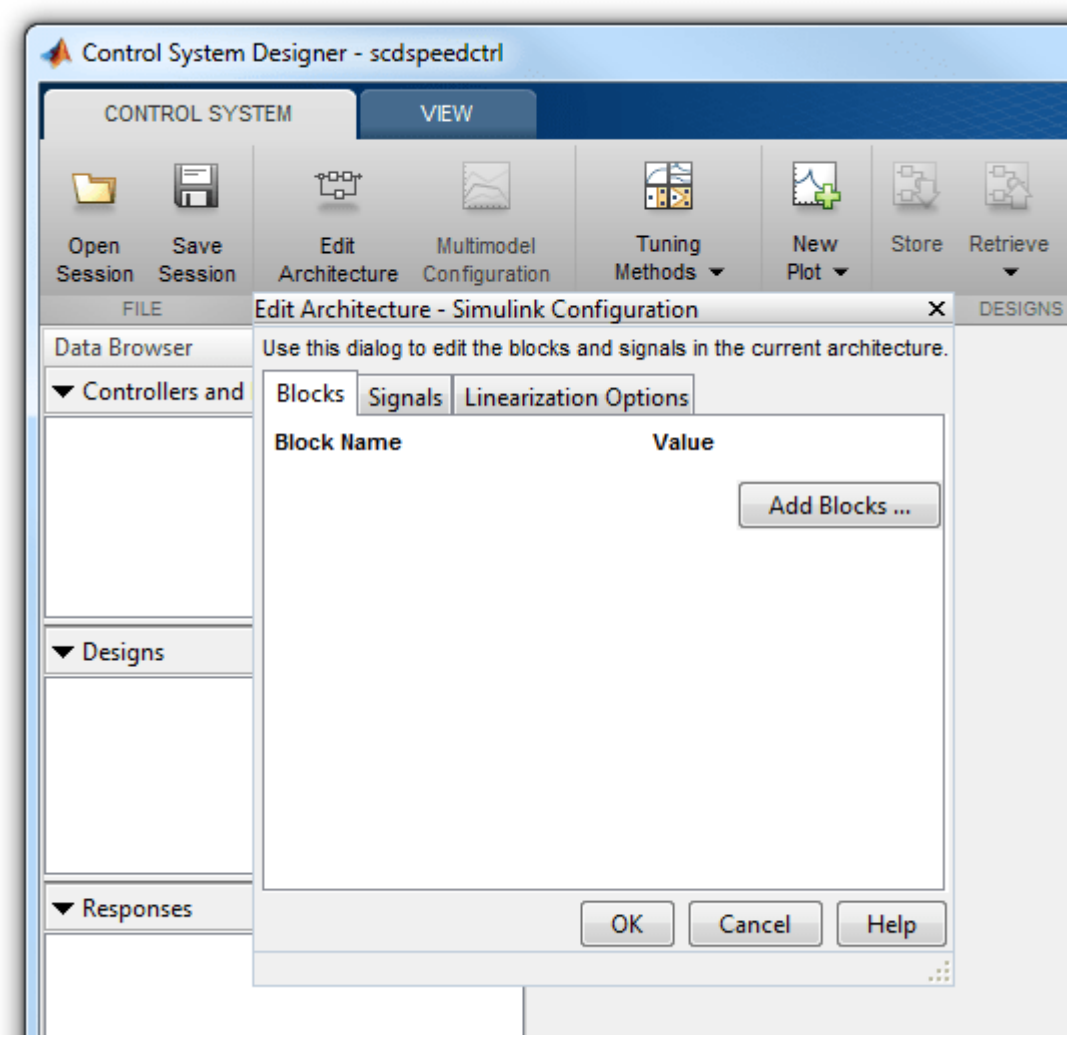
This example uses **Control System Designer** to tune the compensators in the feedback system. To open the **Control System Designer**

- Launch a pre-configured **Control System Designer** session by double-clicking the subsystem in the lower left corner of the model.
- Configure **Control System Designer** using the following procedure.

Start a New Design

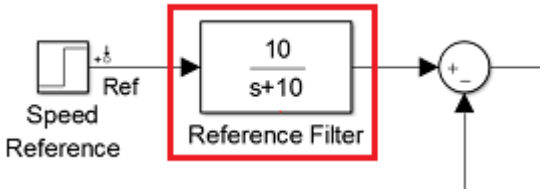
To open **Control System Designer**, in the Simulink model window, in the **Apps** gallery, click **Control System Designer**.

The **Edit Architecture** dialog box opens when the **Control System Designer** launches.

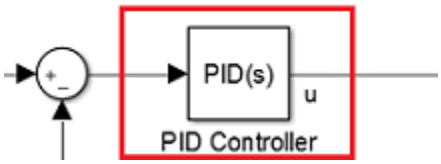


In the **Edit Architecture** dialog box, on the **Blocks** tab, click **Add Blocks**, and select the following blocks to tune:

- `scdspeedctrl/Reference Filter`

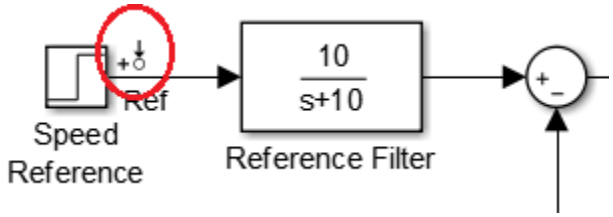


- `scdspeedctrl/PID Controller`

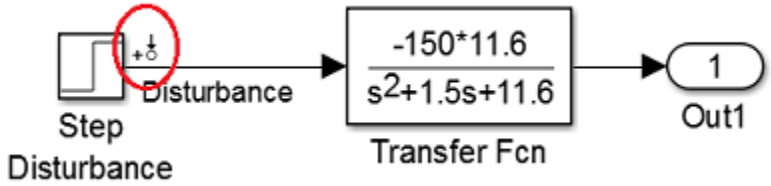


On the **Signals** tab, the analysis points defined in the Simulink model are automatically added as **Locations**.

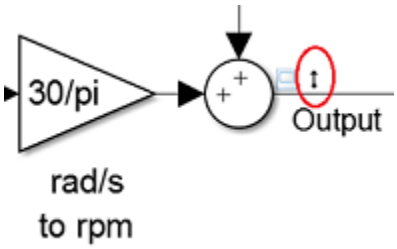
- Input: `scdspeedctrl/Speed Reference` output port 1



- Input `scdspeedctrl/External Disturbance/Step Disturbance` output port 1

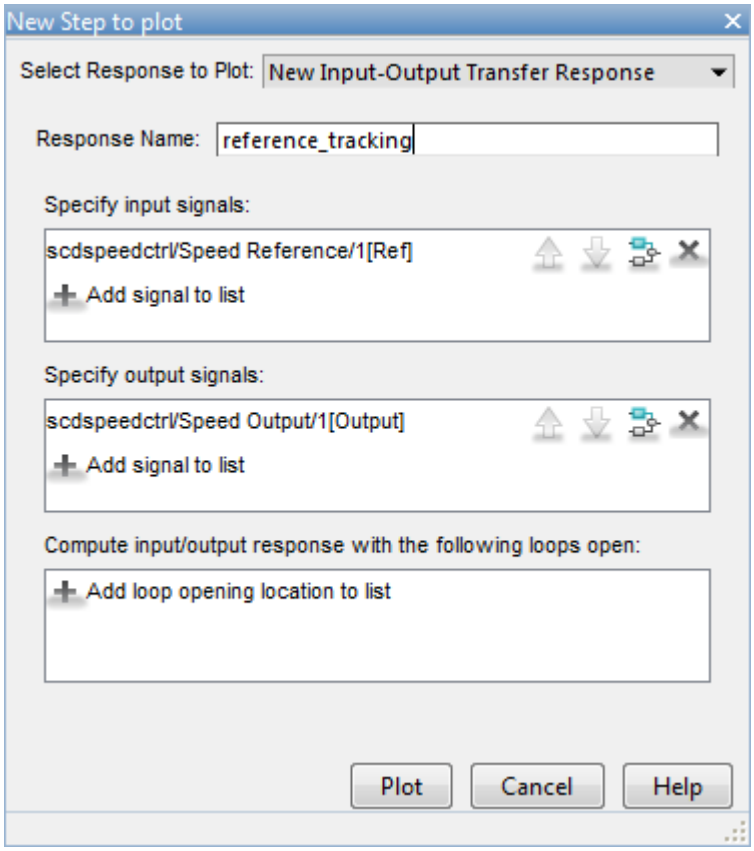


- Output `scdspeedctrl/Speed Output` output port 1



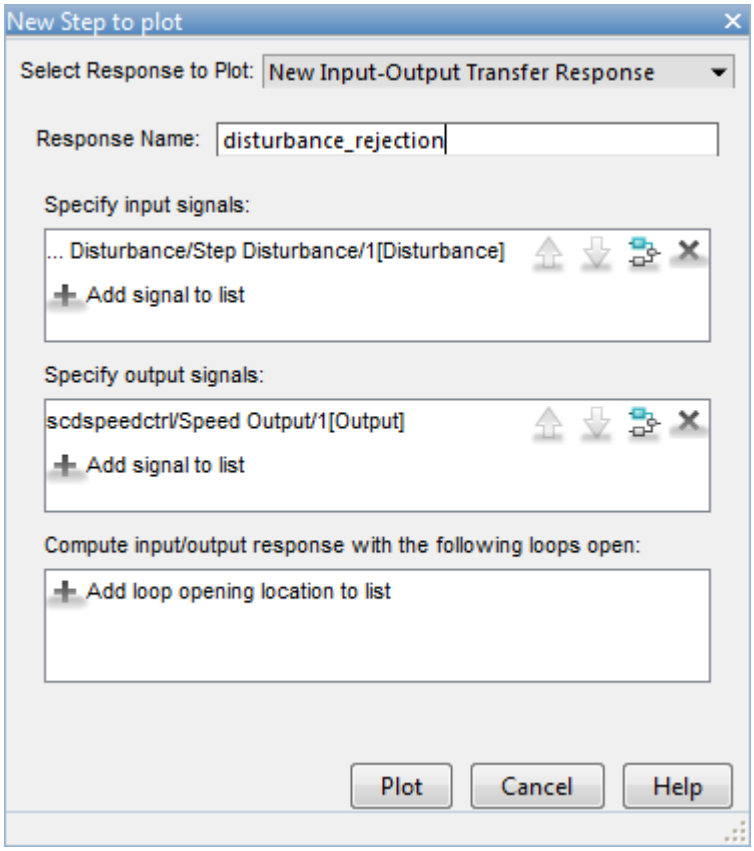
On the **Linearization Options** tab, in the **Operating Point** drop-down list, select **Model Initial Condition**.

Create new plots to view the step responses while tuning the controllers. In **Control System Designer**, click **New Plot**, and select **New Step**. In the **Select Response to Plot** drop-down menu, select **New Input-Output Transfer Response**. Configure the response as follows:



To view the response, click **Plot**.

Similarly, create a step response plot to show the disturbance rejection. In the New Step to plot dialog box, configure the response as follows:



Tune Compensators

Control System Designer contains several methods tuning a control system:

- Manually tune the parameters of each compensator using the compensator editor. For more information, see “Tune Simulink Blocks Using Compensator Editor” on page 9-67.
- Graphically tune the compensator poles, zeros, and gains using open/closed-loop Bode, root locus, or Nichols editor plots. Click **Tuning Methods**, and select an editor under **Graphical Tuning**.
- Optimize compensator parameters using both time-domain and frequency-domain design requirements (requires Simulink Design Optimization™ software). Click **Tuning Methods**, and select **Optimization based tuning**. For more information, see “Enforcing Time and Frequency Requirements on a Single-Loop Controller Design” (Simulink Design Optimization).
- Compute initial compensator parameters using automated tuning based on parameters such as closed-loop time constants. Click **Tuning Methods**, and select either **PID Tuning**, **Internal Model Control (IMC) Tuning**, **Loop Shaping** (requires Robust Control Toolbox™ software), or **LQG Synthesis**.

Completed Design

The following compensator parameters satisfy the design requirements:

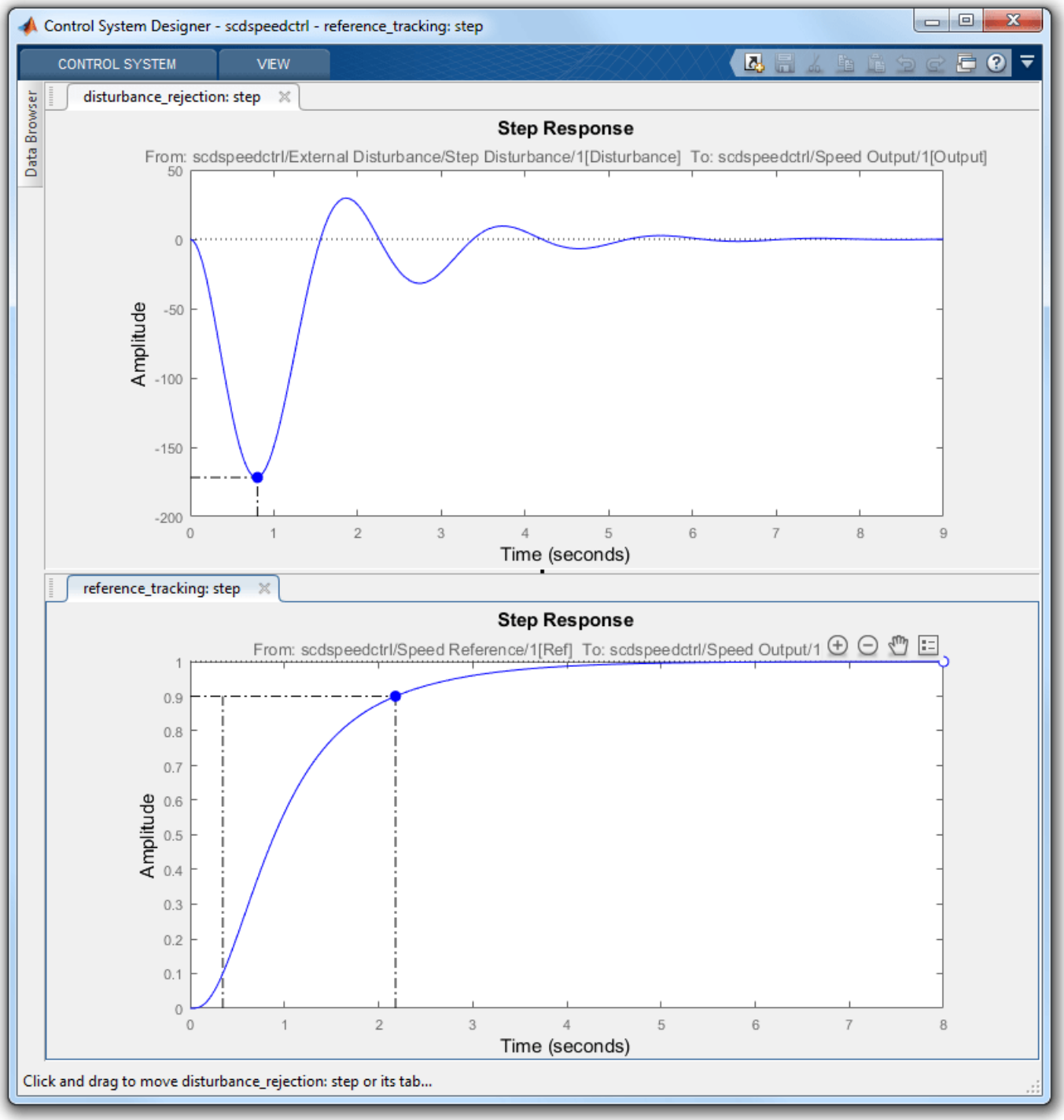
- `scdspeedctrl/PID Controller` has parameters:

```
P = 0.0012191
I = 0.0030038
```

- `scdspeedctrl/Reference Filter`:

```
Numerator = 10
Denominator = [1 10]
```

The responses of the closed-loop system are shown below:



Update Simulink Model

To write the compensator parameters back to the Simulink model, click **Update Blocks**. You can then test your design on the nonlinear model.

```
bdclose('scdspeedctrl')
```

See Also

Control System Designer

More About

- “Control System Designer Tuning Methods” on page 9-4
- “Analyze Designs Using Response Plots” on page 9-27

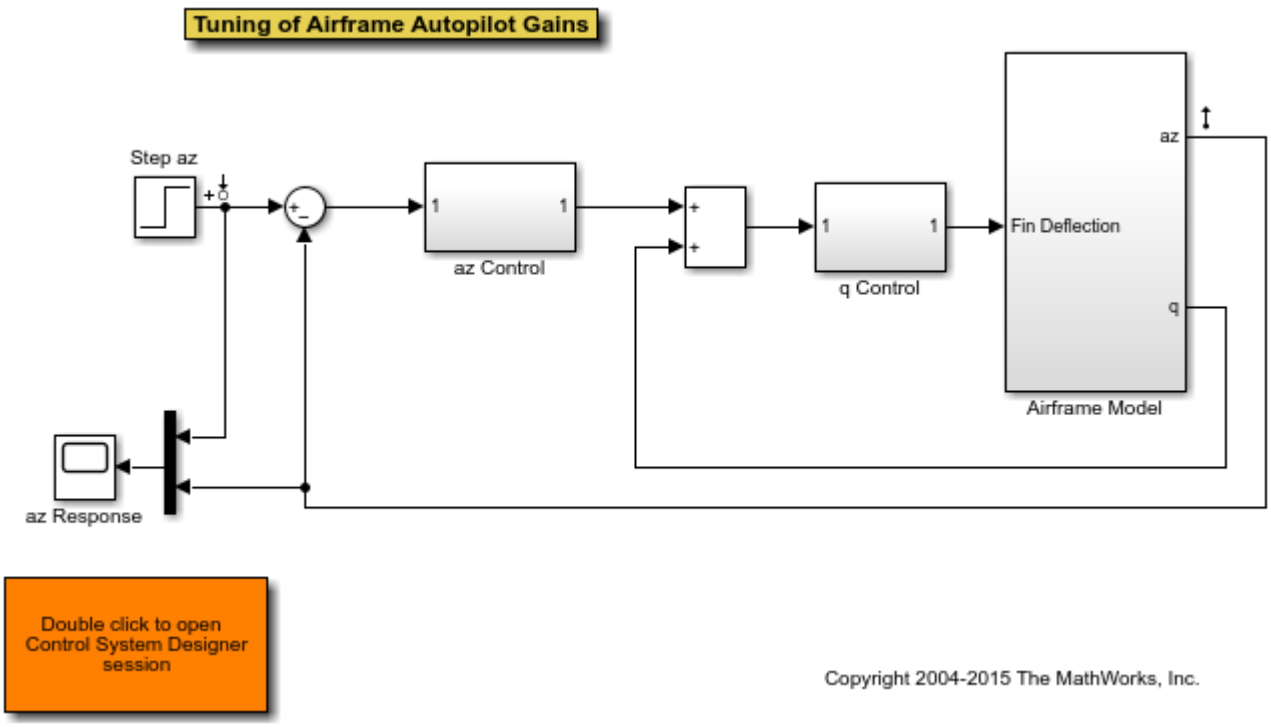
Cascaded Multiloop Feedback Design

This example shows how to tune two cascaded feedback loops in Simulink® Control Design™ using **Control System Designer**.

This example designs controllers for two cascaded feedback loops in an airframe model such that the acceleration component (`az`) tracks reference signals with a maximum rise time of 0.5 seconds. The feedback loop structure in this example uses the body rate (`q`) as an inner feedback loop and the acceleration (`az`) as an outer feedback loop.

Open the airframe model.

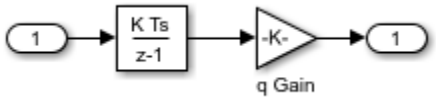
```
open_system('scdairframectrl')
```



The two feedback controllers are:

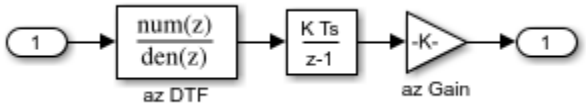
- `scdairframectrl/q Control` - A discrete-time integrator and a gain block stabilize the inner loop.

```
open_system('scdairframectrl/q Control')
```



- `scdairframectrl/az Control` - A discrete-time integrator, a discrete transfer function, and a gain block stabilize the outer loop.

```
open_system('scdairframectrl/az Control')
```



Decoupling Loops in Multiloop Systems

The typical design procedure for cascaded feedback systems is to first design the inner loop and then the outer loop. In **Control System Designer**, it is possible to design both loops simultaneously; by default, when designing a multi-loop feedback system the coupling effects between loops are taken into account. However, when designing two feedback loops simultaneously, it can be necessary decouple the feedback loops; that is, remove the effect of the outer loop when tuning the inner loop. In this example, you design the inner feedback loop (`q`) with the effect of the outer loop (`az`) removed.

Configure Control System Designer

To design a controller using **Control System Designer**, you must:

- Select the controller blocks that you want to tune.
- Create the open-loop and closed-loop responses that you want to view.

For this example, you can:

- Launch a preconfigured **Control System Designer** session by double-clicking the subsystem in the lower left corner of the model.
- Configure **Control System Designer** using the following procedure.

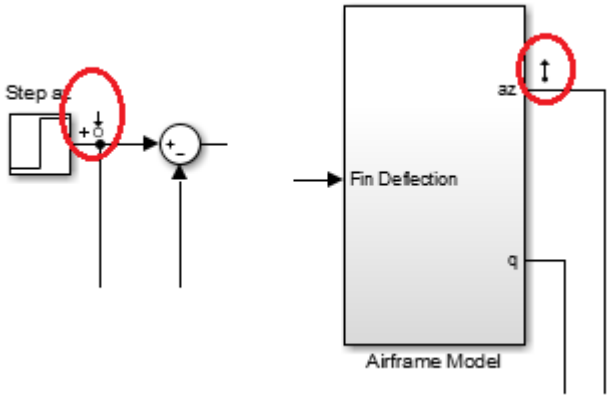
To open **Control System Designer**, in the Simulink model, in the **Apps** gallery, click **Control System Designer**.

In the Edit Architecture dialog box, on the **Blocks** tab, click **Add Blocks**. In the Select Blocks to Tune dialog box, select the following blocks, and click **OK**.

- `scdairframectrl/q Control/q Gain`
- `scdairframectrl/az Control/az Gain`
- `scdairframectrl/az Control/az DTF`

On the **Signals** tab, the analysis points defined in the Simulink model are automatically added as **Locations**.

- Input: `scdairframectrl/Step az` - Output port 1
- Output: `scdairframectrl/Airframe Model` - Output port 1



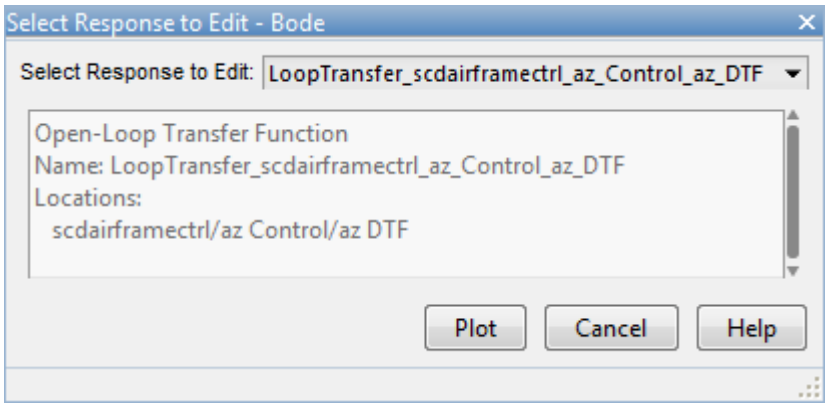
To use the selected blocks and signals, click **OK**.

In the **Data Browser**, the **Responses** section contains the following open-loop responses, which **Control System Designer** automatically recognizes as potential feedback loops for open-loop design.

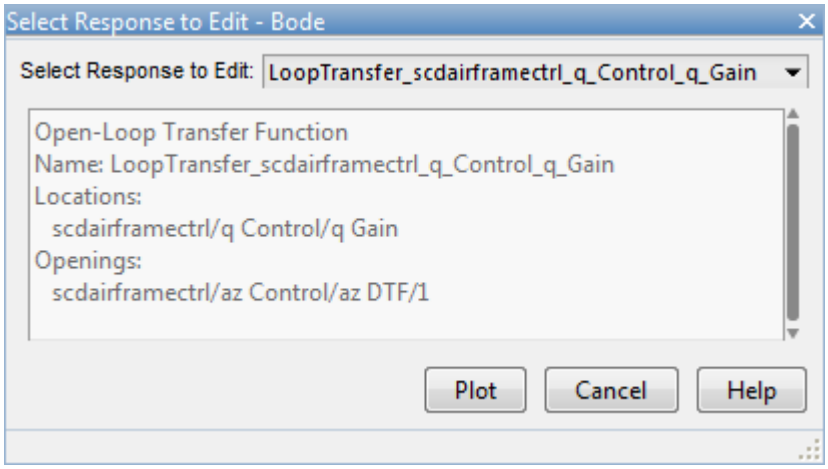
- Output port 1 of `scdairframectrl/az Control/az DTF`
- Output port 1 of `scdairframectrl/az Control/az Gain`
- Output port 1 of `scdairframectrl/q Control/q Gain`

Open graphical Bode editors for each of the following responses. In **Control System Designer**, select **Tuning Methods > Bode Editor**. Then, in the **Select Response to Edit** drop-down list, select the corresponding open-loop responses, and click **Plot**.

- Open Loop at outport 1 of `scdairframectrl/az Control/az DTF`

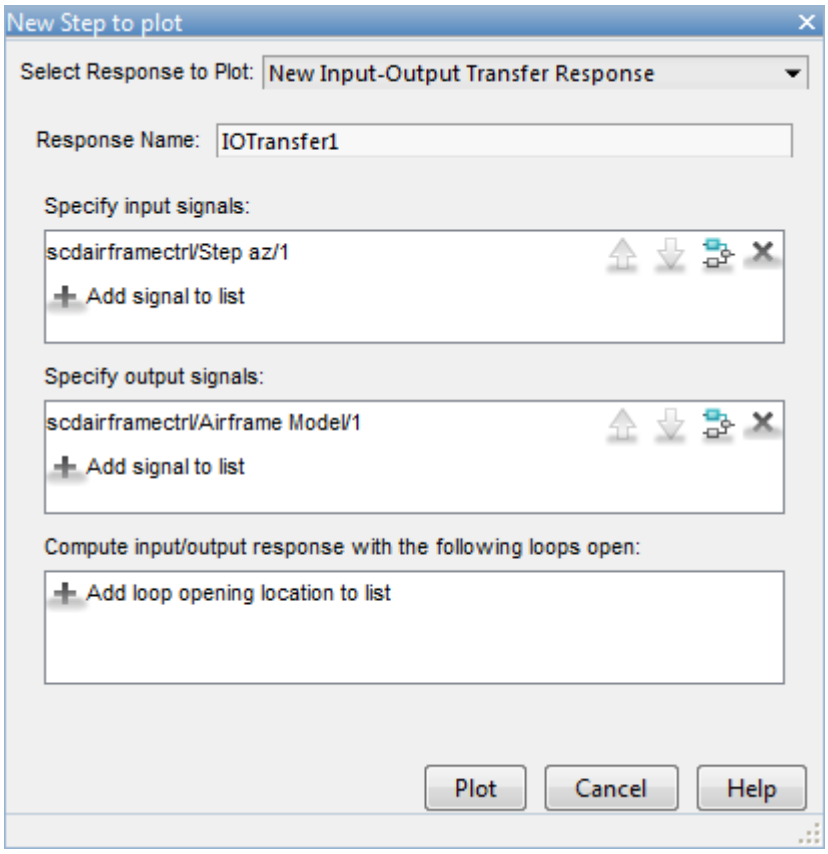


- Open Loop at outport 1 of `scdairframectrl/q Control/q Gain`



To view the closed-loop response of the feedback system, create a step plot for a new input-output transfer function response. Select **New Plot > New Step**. Then, in the New Step to plot dialog box, in the **Select Response to Plot** drop-down list, select **New Input-Output Transfer Response**.

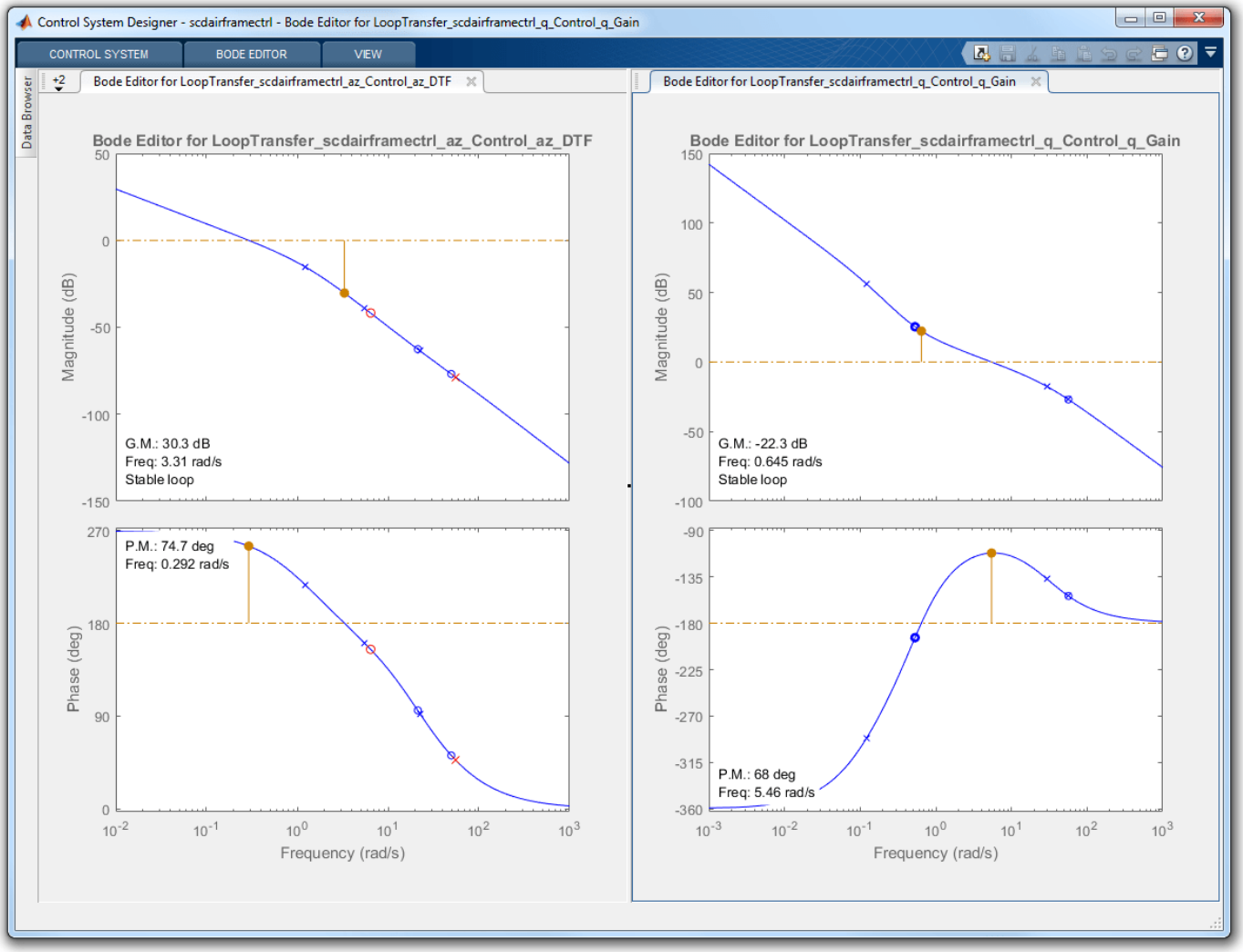
Add `scdairframectrl/Step az/1` as an input signal and `scdairframectrl/Airframe Model/1` as an output signal.



Click **Plot**.

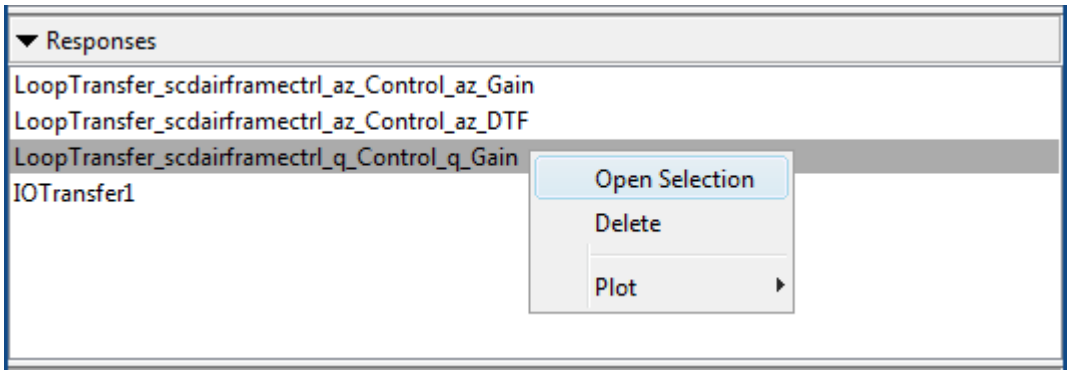
Remove Effect of Outer Feedback Loop

In the outer-loop bode editor plot, **Bode Editor for LoopTransfer_scdairframectrl_az_Control_az_DTF**, increase the gain of the feedback loop by dragging the magnitude response upward. The inner-loop bode editor plot, **Bode Editor for LoopTransfer_scdairframectrl_q_Control_q_Gain**, also changes. This change is a result of the coupling between the feedback loops. A more systematic approach is to first design the inner feedback loop, with the outer loop open.

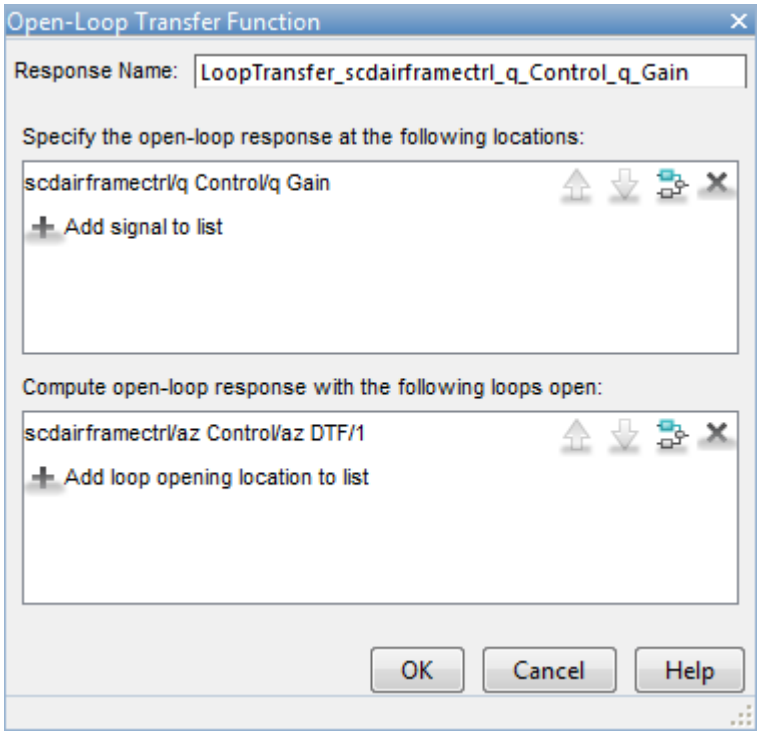


To remove the effect of the outer loop when designing the inner loop, add a loop opening to the open-loop response of the inner loop.

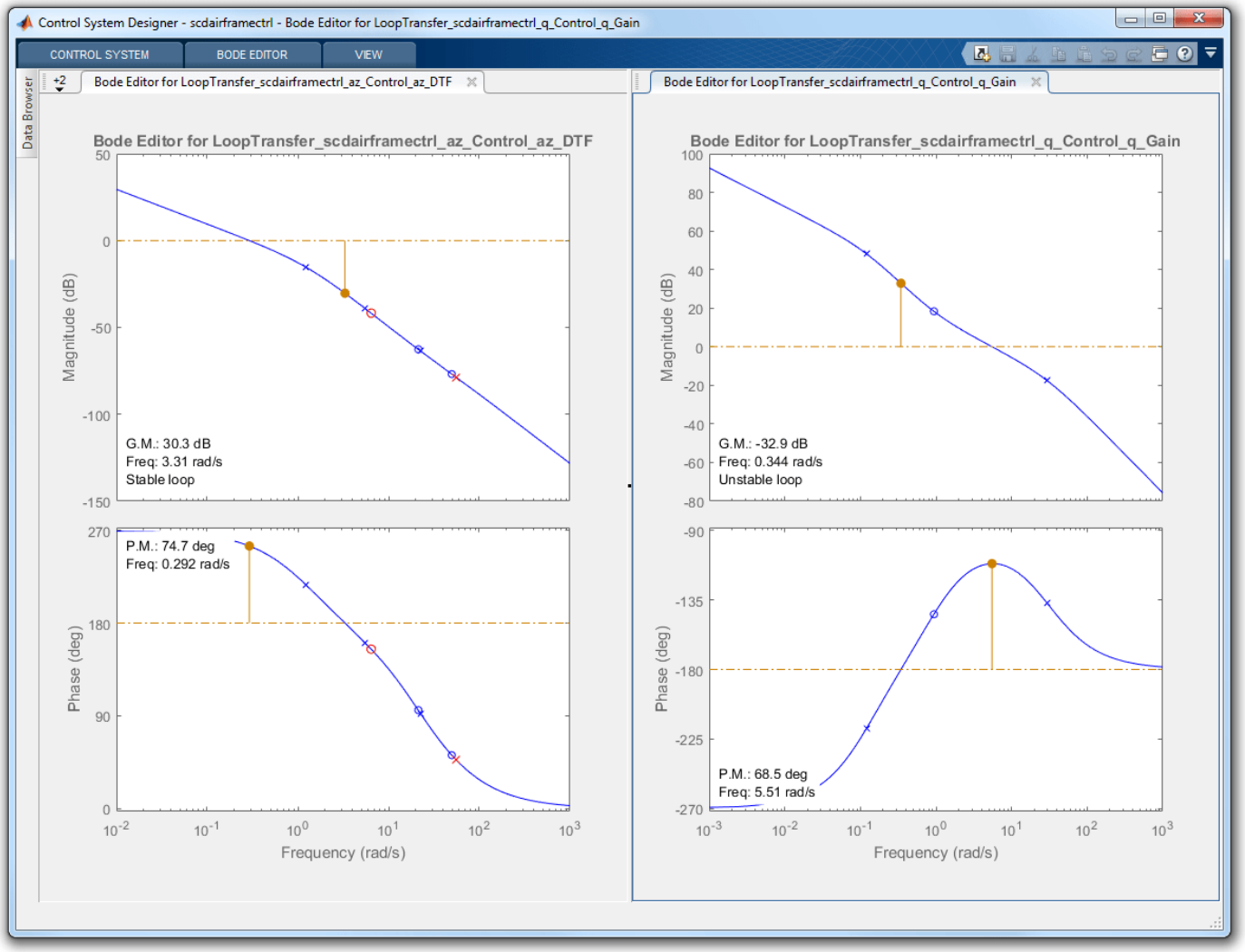
In the **Data Browser**, in the **Responses** area, right-click the inner loop response, and select **Open Selection**.



In the Open-Loop Transfer Function dialog box, specify `scdairframectrl/az Control/az DTF/1` as the loop opening. Click **OK**.



In the outer-loop Bode editor plot, increase the gain by dragging the magnitude response. Since the loops are decoupled, the inner-loop Bode editor plot does not change.



You can now complete the design of the inner loop without the effect of the outer loop and simultaneously design the outer loop while taking the effect of the inner loop into account.

Tune Compensators

Control System Designer contains several methods tuning a control system:

- Manually tune the parameters of each compensator using the compensator editor. For more information, see “Tune Simulink Blocks Using Compensator Editor” on page 9-67.
- Graphically tune the compensator poles, zeros, and gains using open/closed-loop Bode, root locus, or Nichols editor plots. Click **Tuning Methods**, and select an editor under **Graphical Tuning**.
- Optimize compensator parameters using both time-domain and frequency-domain design requirements (requires Simulink Design Optimization™ software). Click **Tuning Methods**, and select **Optimization based tuning**. For more information, see “Enforcing Time and Frequency Requirements on a Single-Loop Controller Design” (Simulink Design Optimization).
- Compute initial compensator parameters using automated tuning based on parameters such as closed-loop time constants. Click **Tuning Methods**, and select either **PID Tuning**, **Internal**

Model Control (IMC) Tuning, **Loop Shaping** (requires Robust Control Toolbox™ software), or **LQG Synthesis**.

Complete Design

The following compensator parameters satisfy the design requirements:

- `scdairframectrl/q Control/q Gain`:

```
K_q = 2.7717622
```

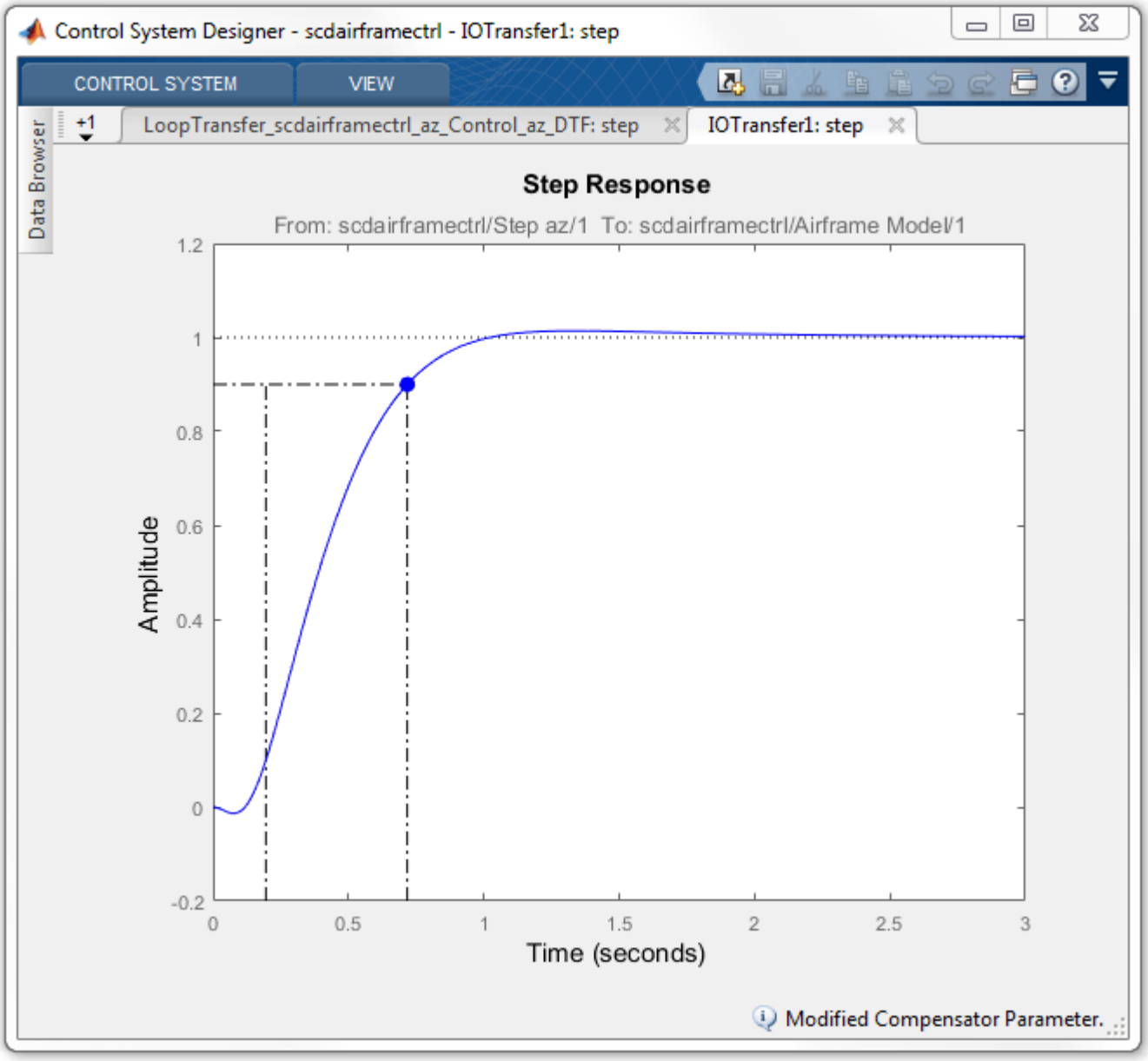
- `scdairframectrl/az Control/az Gain`:

```
K_az = 0.00027507
```

- `scdairframectrl/az Control/az DTF`:

```
Numerator = [100.109745 -99.109745]
Denominator = [1 -0.88893]
```

The response of the closed-loop system is shown below:



Update Simulink Model

To write the compensator parameters back to the Simulink model, click **Update Blocks**. You can then test your design on the nonlinear model.

See Also

Control System Designer

More About

- “Control System Designer Tuning Methods” on page 9-4

- “Analyze Designs Using Response Plots” on page 9-27
- “Design Multiloop Control System”

Tune Custom Masked Subsystems

This example shows how to enable custom masked subsystems in **Control System Designer**. Once configured, you can tune a custom masked subsystem in the same way as any supported blocks in Simulink® Control Design™. For more information, see “What Blocks Are Tunable?” on page 9-8.

Lead-Lag Library Block

For this example, you tune the Lead-Lag Controller block in the `scdexblks` library.

```
open_system('scdexblks')
```

This block implements a compensator with a single zero, a single pole, and a gain. To configure the controller, you can specify the following block parameters.

- **Gain** (K)
- **Zero Frequency** (W_z)
- **Pole Frequency** (W_p)

The Lead-Lag Controller block implements the following transfer function.

$$G(s) = K\frac{\frac{s}{W_z} + 1}{\frac{s}{W_p} + 1}$$

Configure Subsystem for Control System Designer

To configure a masked subsystem for tuning with **Control System Designer**, you specify a configuration function. In this example, the block uses the configuration function in `scdleadexample.m`. To open this file, at the MATLAB® command line, type `edit scdleadexample`.

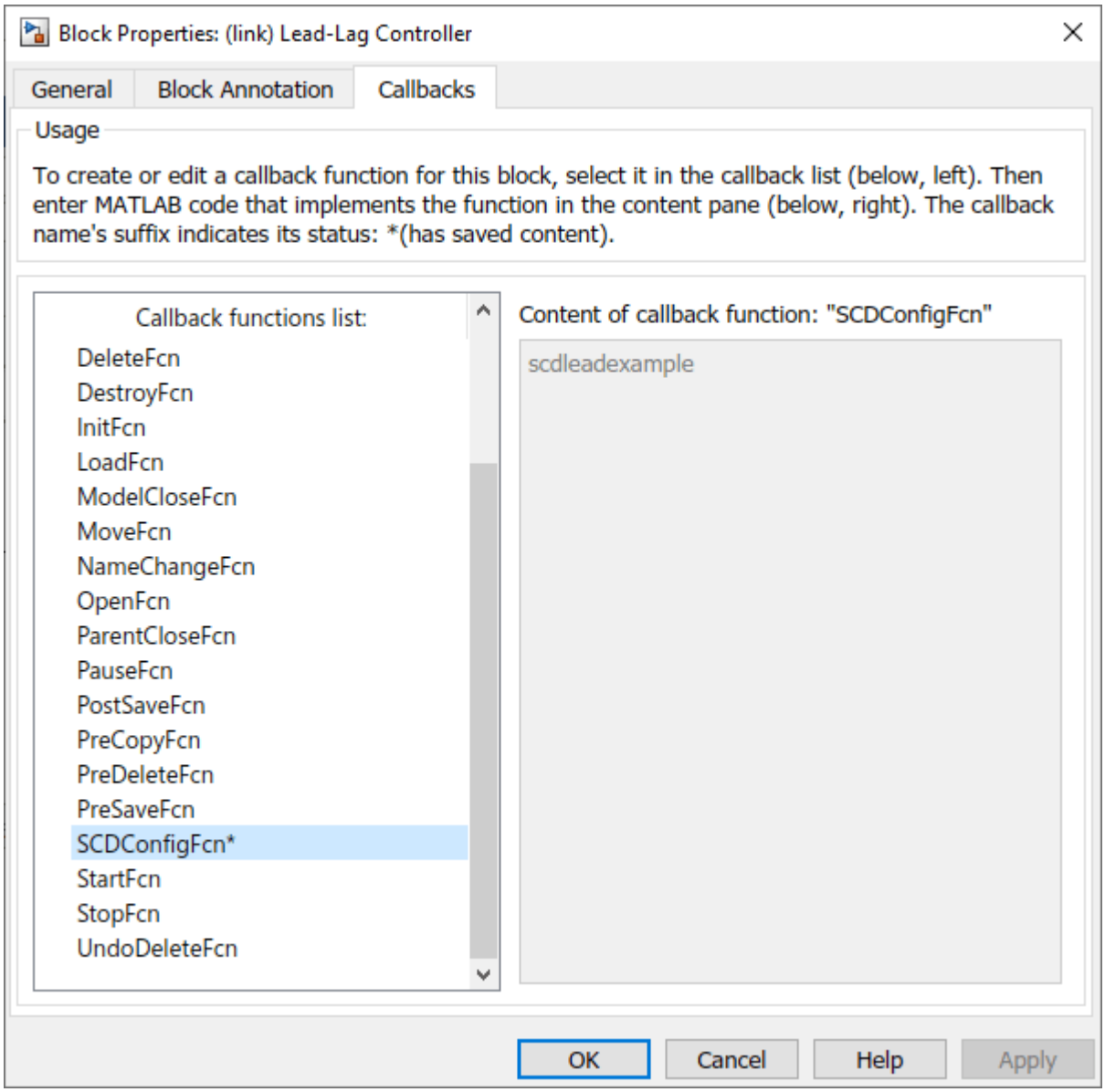
This function returns a structure with the following fields.

- `TunableParameters` — Structure array with one element for each tunable parameter (gain, zero, pole)
- `EvalFcn` — Handle to function that converts block parameters to zero-pole-gain form
- `InvFcn` — Handle to function that computes block parameters given zero-pole-gain values
- `Constraints` — Structure specifying constraints on the block, such as the number of poles and zeros
- `Inport` — Port number for the controller input
- `Outport` — Port number for the controller output

The `scdleadexample` configuration function specifies the following constraints for the controller block.

- There is only one pole allowed (`MaxPoles` constraint)
- There is only one zero allowed (`MaxZeros` constraint)
- The gain is tunable (`isStaticGainTunable` constraint)

To use a configuration function, specify it as the `SCDConfigfcn` callback function for the block. To do so, right-click the Lead-Lag Controller block and select **Properties**. Then, in the Block Properties dialog box, on the **Callbacks** tab, set `SCDConfigFcn` to `scdleadexample`.



Alternatively, you can set `SCDConfigFcn` using the `set_param` function.

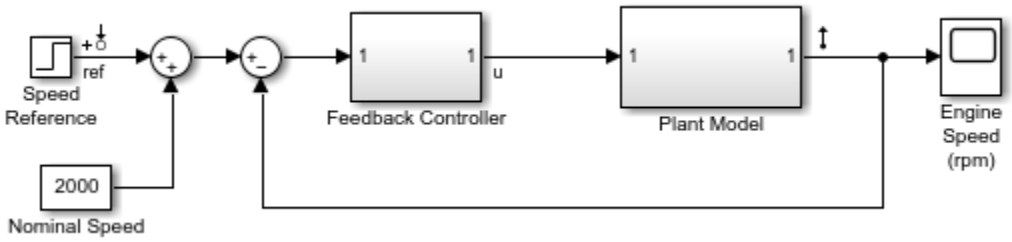
```
set_param(blockpath,'SCDConfigFcn','scdleadexample')
```

Once you set the block configuration function, you can tune the controller using **Control System Designer**.

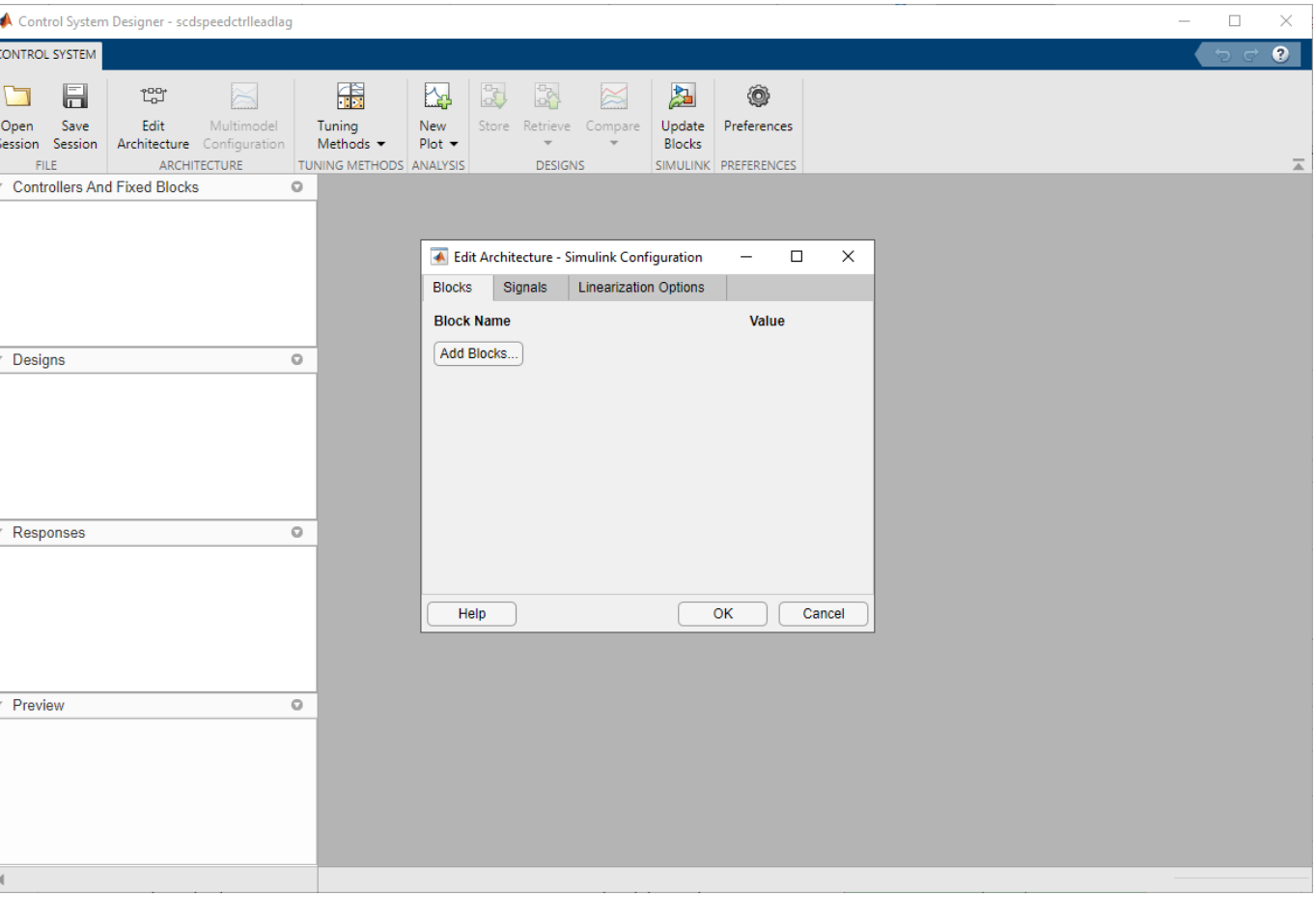
Speed Control System

The `scdspeedctrlleadlag` model uses the Lead-Lag Controller block to tune the feedback loop in “Single Loop Feedback/Prefilter Compensator Design” on page 9-41.

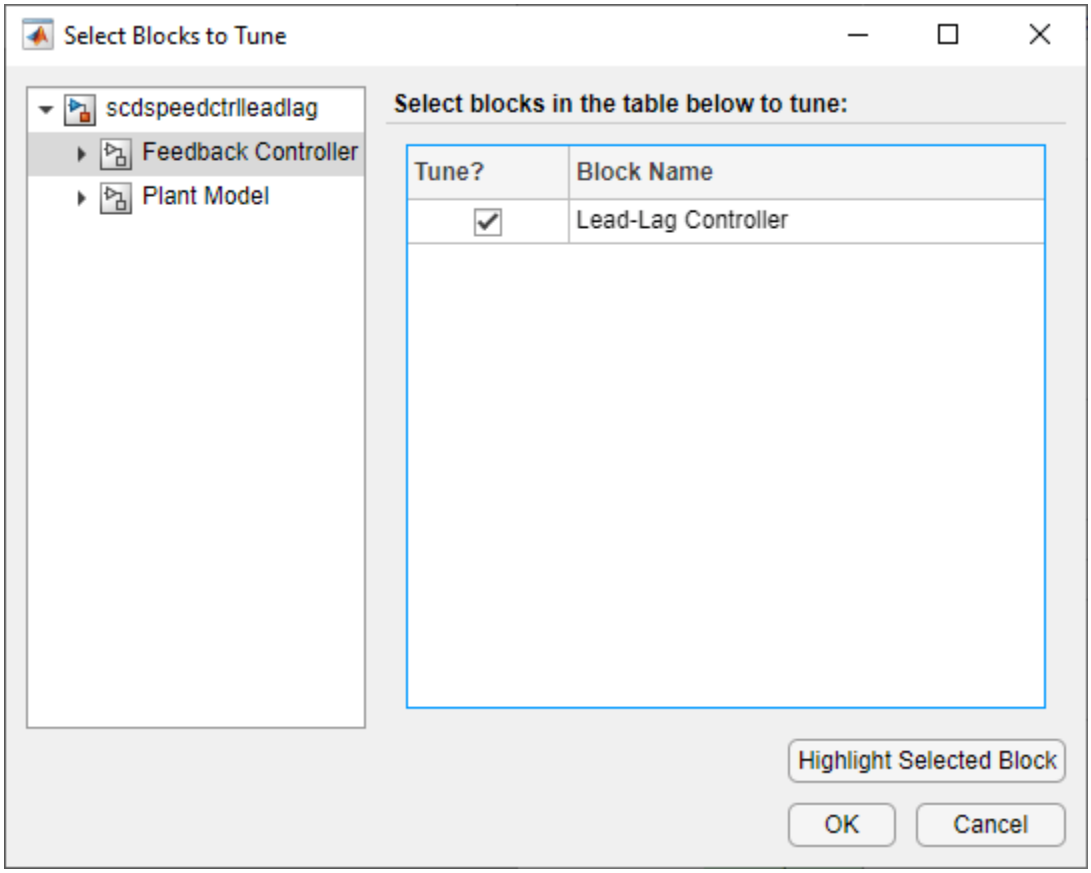
```
open_system('scdspeedctrlleadlag')
```



To open **Control System Designer**, in the Simulink model window, on the **Apps** tab, in the **Apps** gallery, click **Control System Designer**.

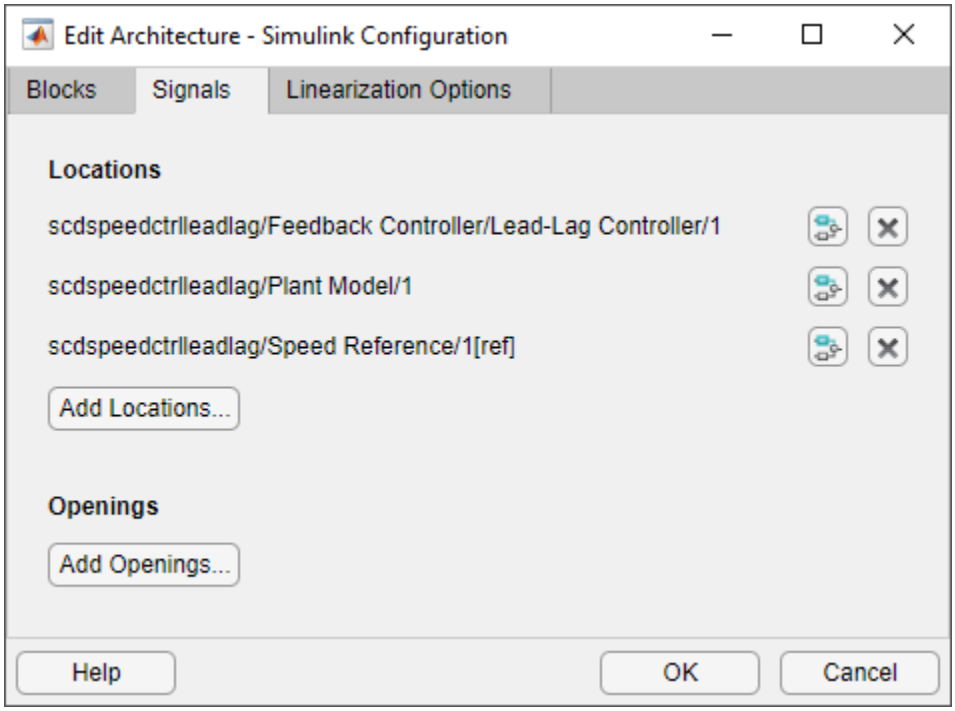


In the Edit Architecture dialog box, on the **Blocks** tab, click **Add Blocks**. Then, in the Select Blocks to Tune dialog box, click `Feedback Controller`, and select `Lead-Lag Controller`.

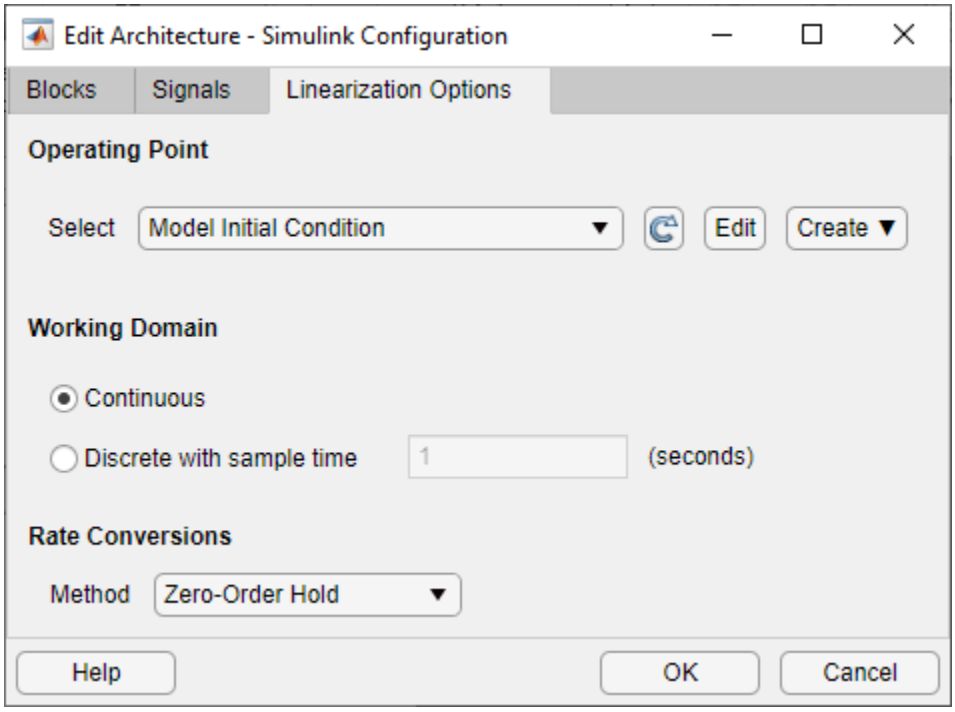


Click **OK**.

In the **Edit Architecture** dialog box, on the **Signals** tab, the analysis points defined in the Simulink model are automatically added as **Locations**.



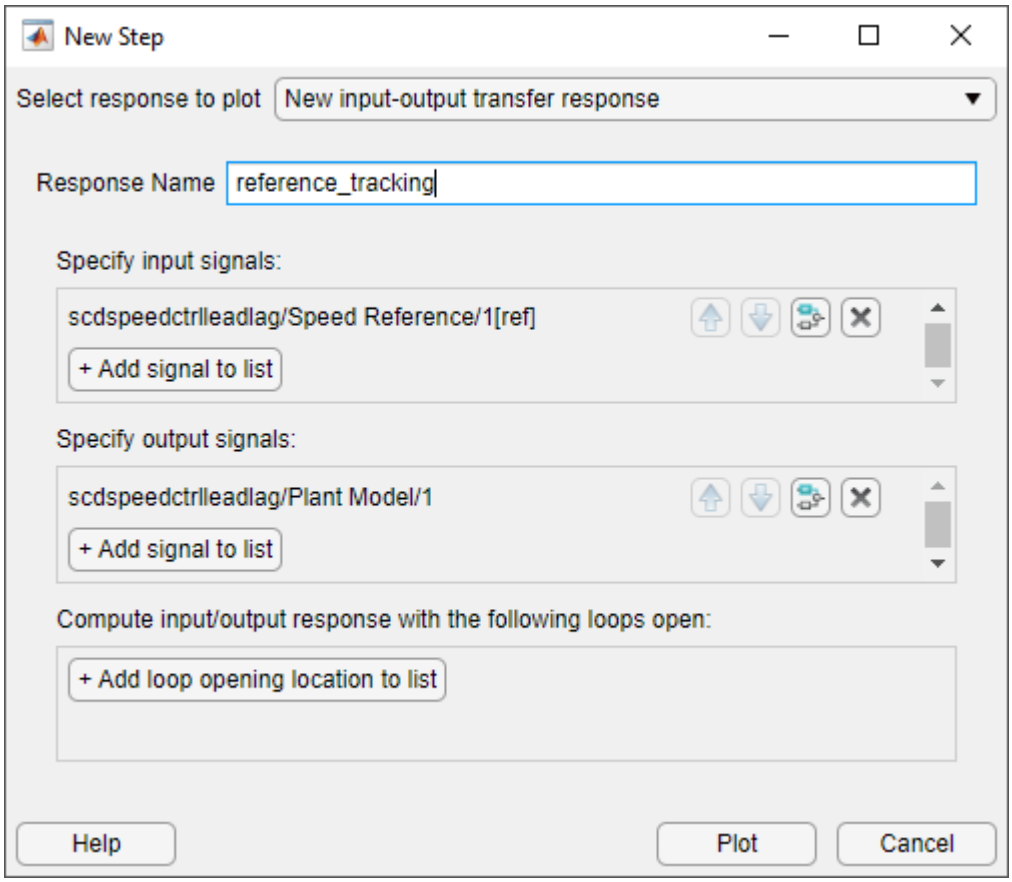
On the **Linearization Options** tab, in the **Operating Point** drop-down list, select `Model Initial Condition`.



Click **OK**.

Create new plots to view the step responses while tuning the controllers. In **Control System Designer**, select **New Plot > New Step**.

In the New Step dialog box, in the **Select response to plot** drop-down menu, select **New input-output transfer response**. Configure the response as shown in the following figure.

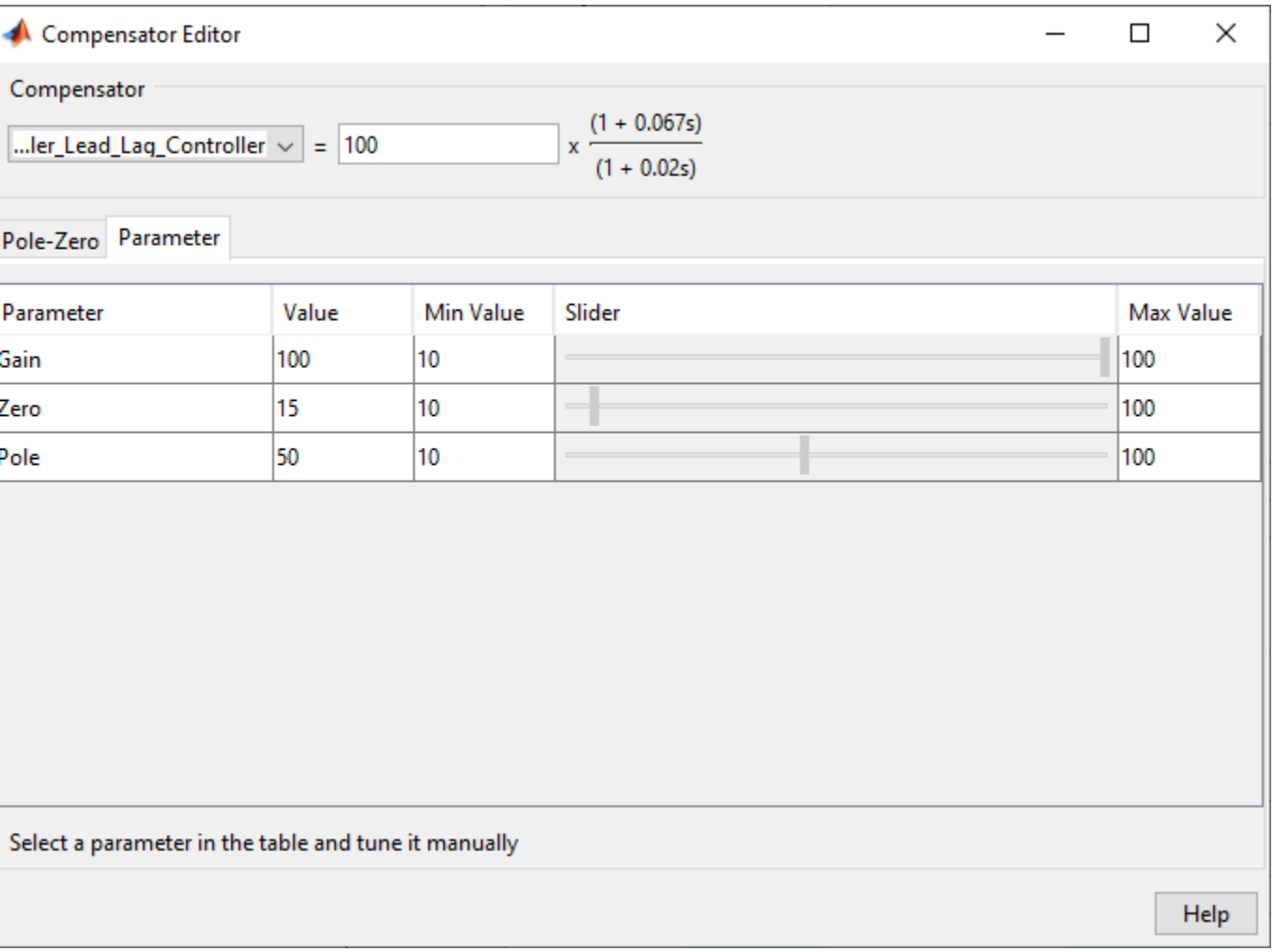


To view the response, click **Plot**.

Tune Compensator

The **Control System Designer** app contains four methods to tune a control system.

- Manually tune the parameters of the `Lead-Lag Controller` using the Compensator Editor. For more information, see “Tune Simulink Blocks Using Compensator Editor” on page 9-67.



- Graphically tune the compensator poles, zeros, and gains using open-loop/closed-loop Bode, root locus, or Nichols editor plots. Click **Tuning Methods**, and select an editor under **Graphical Tuning**.
- Optimize compensator parameters using both time-domain and frequency-domain design requirements (requires Simulink Design Optimization™ software). Click **Tuning Methods**, and select **Optimization based tuning**. For more information, see “Enforcing Time and Frequency Requirements on a Single-Loop Controller Design” (Simulink Design Optimization).
- Compute initial compensator parameters using automated tuning based on parameters such as closed-loop time constants. Click **Tuning Methods**, and select either **PID tuning**, **IMC tuning**, **Loop shaping** (requires Robust Control Toolbox™ software), or **LQG synthesis**.

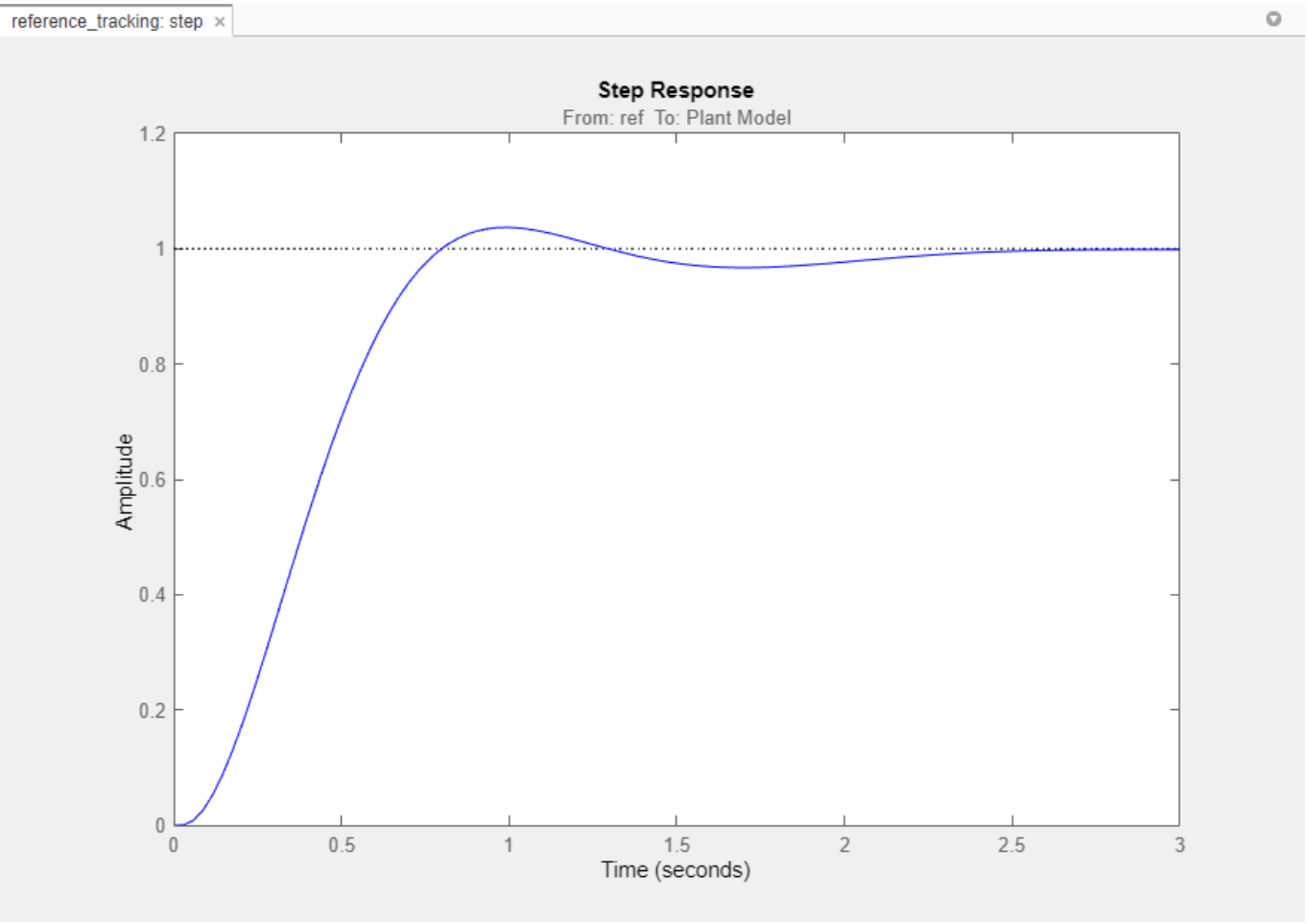
Complete Design

The design requirements for the reference step response in “Single Loop Feedback/Prefilter Compensator Design” on page 9-41 can be met with the following Lead-Lag Controller block parameters.

- **Gain** = 0.0075426
- **Zero Frequency** (rad/s) = 2

- **Pole Frequency** (rad/s) = 103.59

The following figure shows the closed-loop system response for these controller parameters.



Update Simulink Model

To write the compensator parameters back to the Simulink model, click **Update Blocks**. You can then test your design on the nonlinear model.

```
bdclose('scdexblks')
bdclose('scdspeedctrlleadlag')
```

See Also

Control System Designer

More About

- “Control System Designer Tuning Methods” on page 9-4
- “Tune Simulink Blocks Using Compensator Editor”
- “Single Loop Feedback/Prefilter Compensator Design” on page 9-41

Tune Simulink Blocks Using Compensator Editor

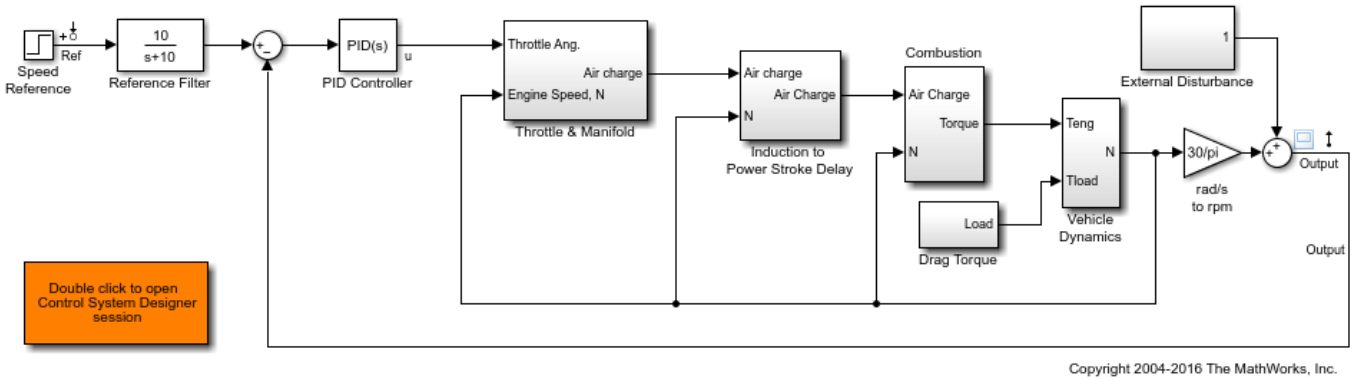
This example shows how to tune Simulink® blocks using the Compensator Editor dialog box in **Control System Designer**.

Open the Model

This example uses a model of a speed control system for a sparking ignition engine. The initial compensator has been designed in a fashion similar to the method shown in “Single Loop Feedback/ Prefilter Compensator Design” on page 9-41.

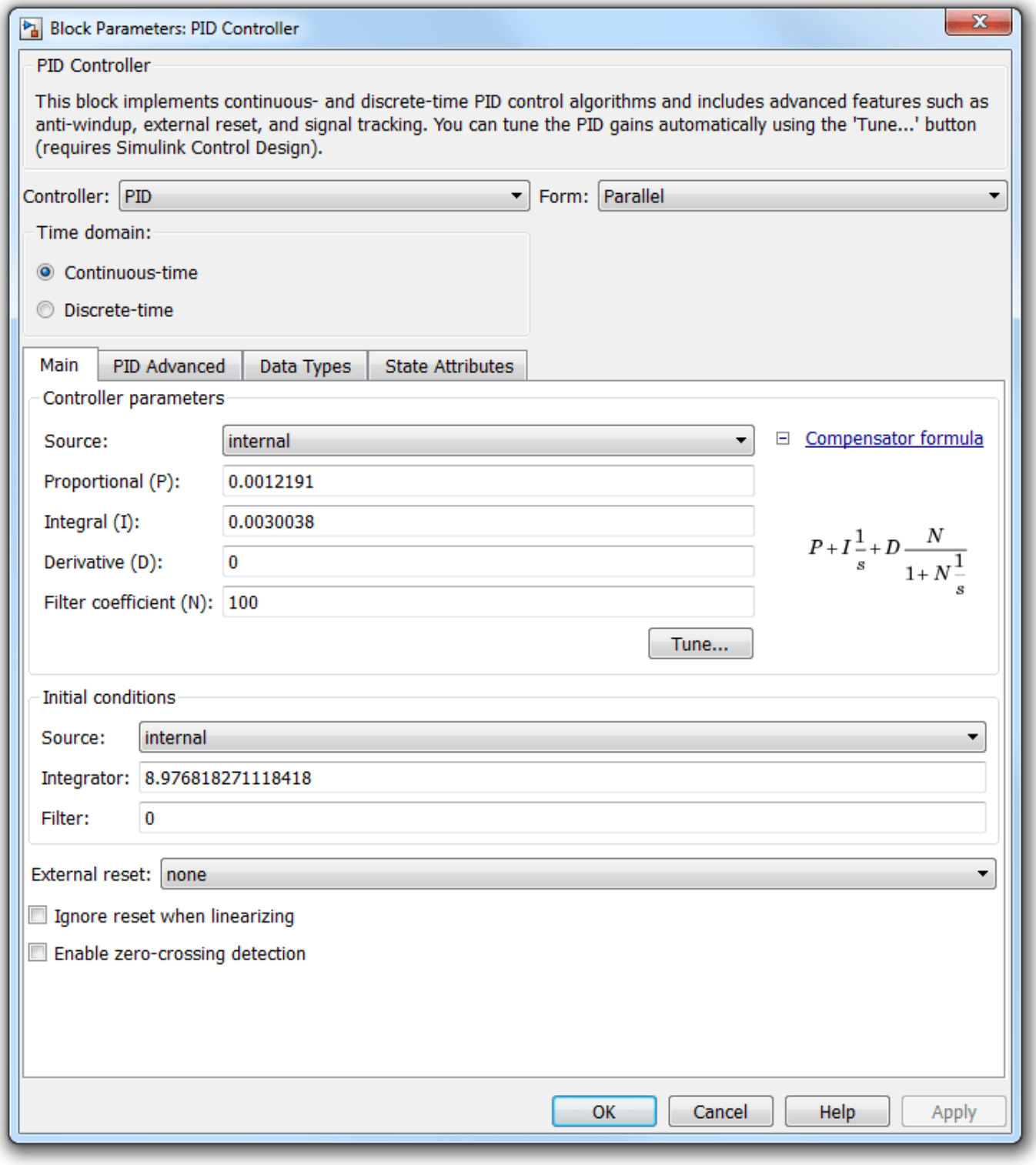
Open and explore the engine speed control model.

```
open_system('scdspeedctrl')
```



Introduction

This example uses the **Compensator Editor** to tune Simulink blocks. When tuning a block in Simulink using **Control System Designer**, you can tune the block parameters directly or you can tune a zero-pole-gain representation of the block. For example, in the speed control example there is a PID controller with filtered derivative `scdspeedctrl/PID Controller`:



This block implements the traditional PID with filtered derivative as:

$$G(s) = P + \frac{I}{s} + \frac{Ds}{Ns+1}$$

In this block `P`, `I`, `D`, and `N` are the parameters that are available for tuning. Another approach is to reformulate the block transfer function to use zero-pole-gain format:

$$G(s) = \frac{Ps(Ns+1) + I(Ns+1) + Ds^s}{s(Ns+1)} = \frac{K(s^2 + 2\zeta\omega_n + w_n^2)}{s(s+z)}$$

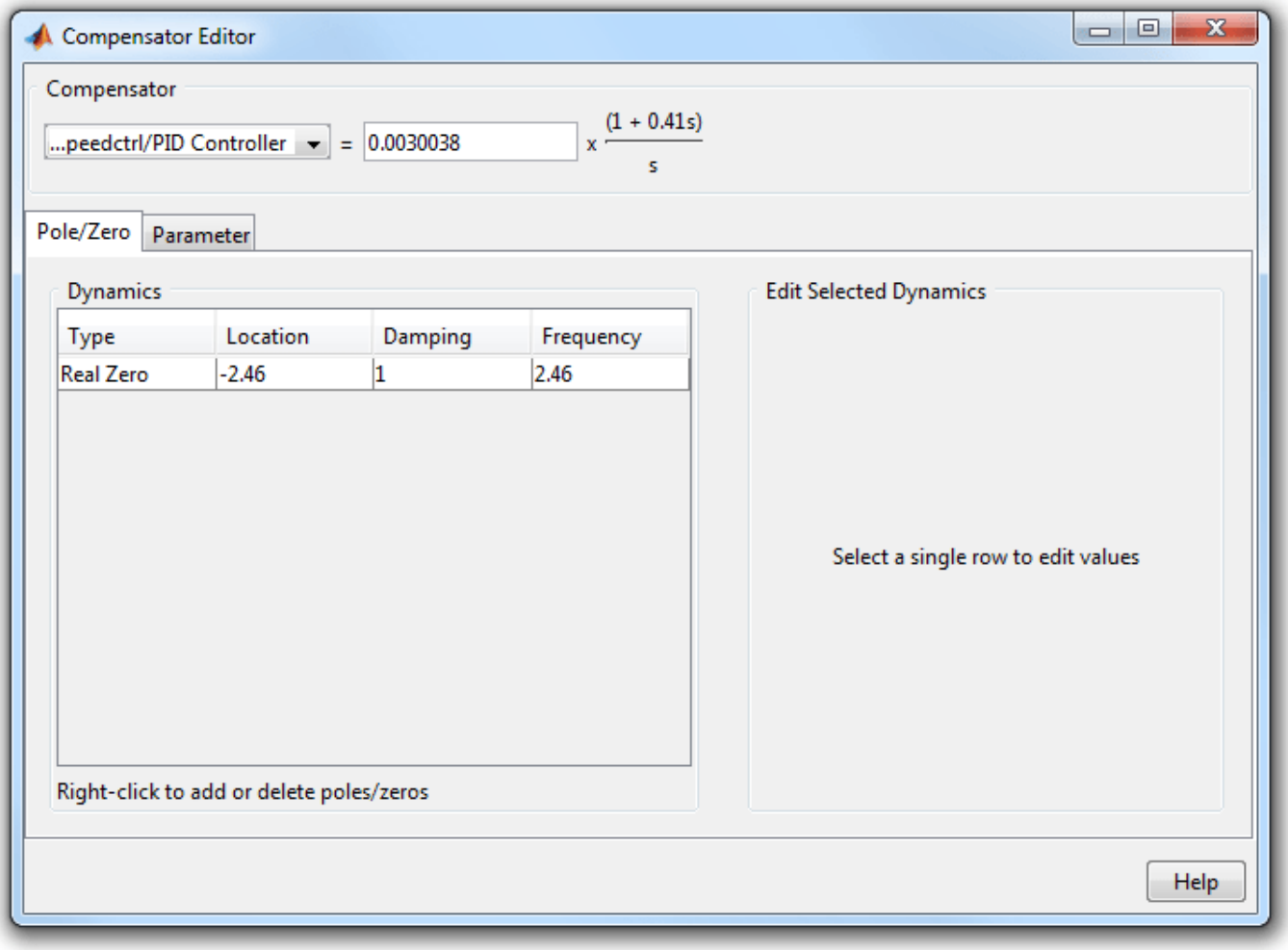
This formulation of poles, zeros, and gains allows for direct graphical tuning on design plots such as Bode, root locus, and Nichols plots. Additionally, **Control System Designer** allows for both representations to be tuned using the compensator editor. The tuning of both representations is available for all supported blocks in Simulink Control Design™. For more information, see “What Blocks Are Tunable?” on page 9-8.

Open Control System Designer

In this example, to tune the compensators in this feedback system, open a preconfigured **Control System Designer** session by double clicking the subsystem in the lower left-hand corner of the model.

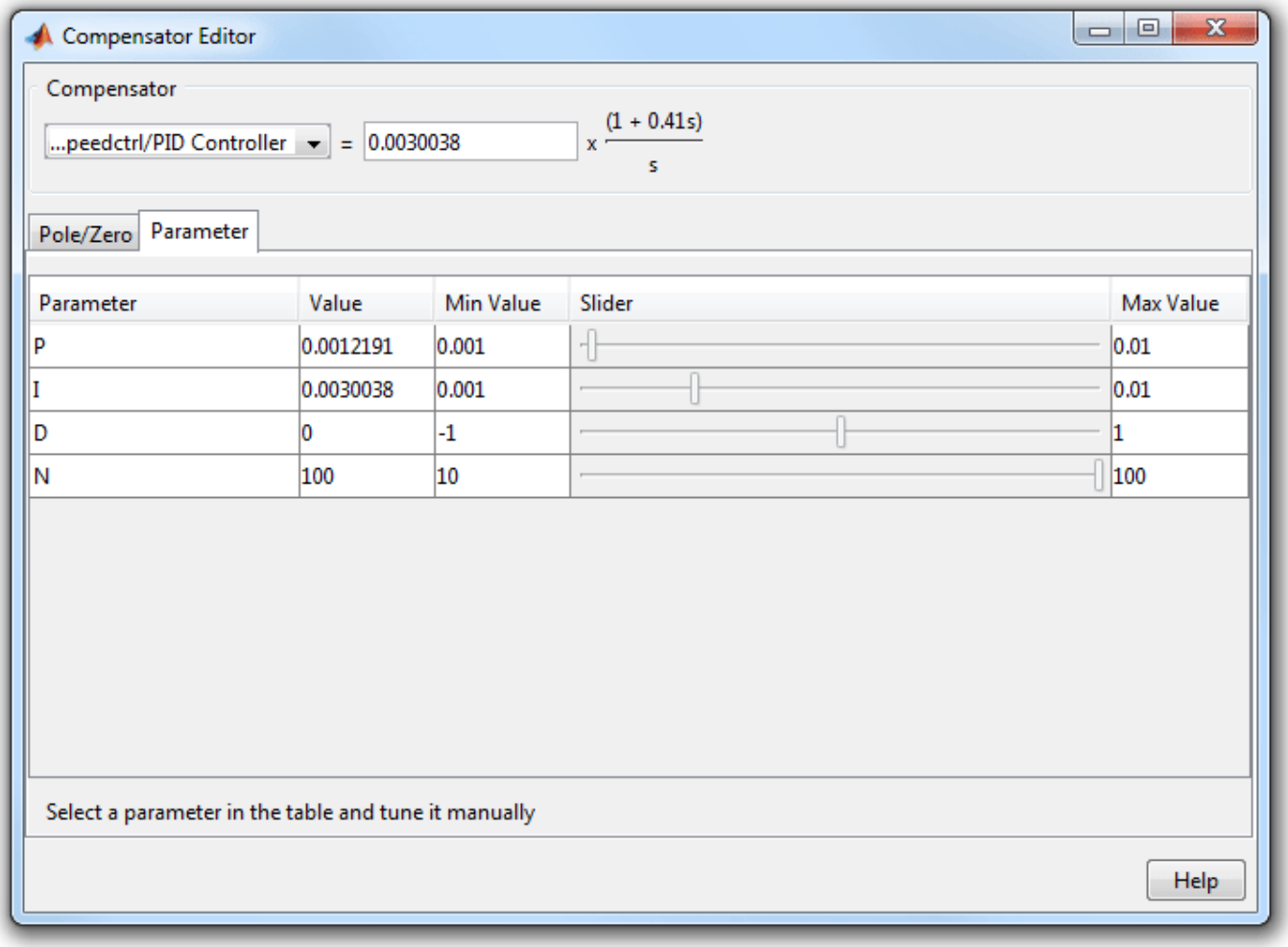
Compensator Editor Dialog Box

You can view the representations of the PID compensator using the Compensator Editor dialog box. To open the Compensator Editor, in the **Data Browser**, in the **Controllers and Fixed Blocks** section, double-click `scdspeedctrl_PID_Controller`. In the Compensator Editor dialog box, in the Compensator section, you can view and edit any of the compensators in your system.



On the **Pole/Zero** tab, you can add, delete, and edit compensator poles and zeros. Since the PID with filtered derivative is fixed in structure, the number of poles and zeros is limited to having up to two zeros, one pole, and an integrator at `s = 0`.

On the **Parameter** tab, you can independently tune the `P`, `I`, `D`, and `N` parameters.



Enter new parameters values in the **Value** column. To interactively tune the parameters, use the sliders. You can change the slider limits using the **Min Value** and **Max Value** columns.

When you change parameter values, any associated editor and analysis plots automatically update.

Complete Design

The design requirements in “Single Loop Feedback/Prefilter Compensator Design” on page 9-41 can be met with the following controller parameters:

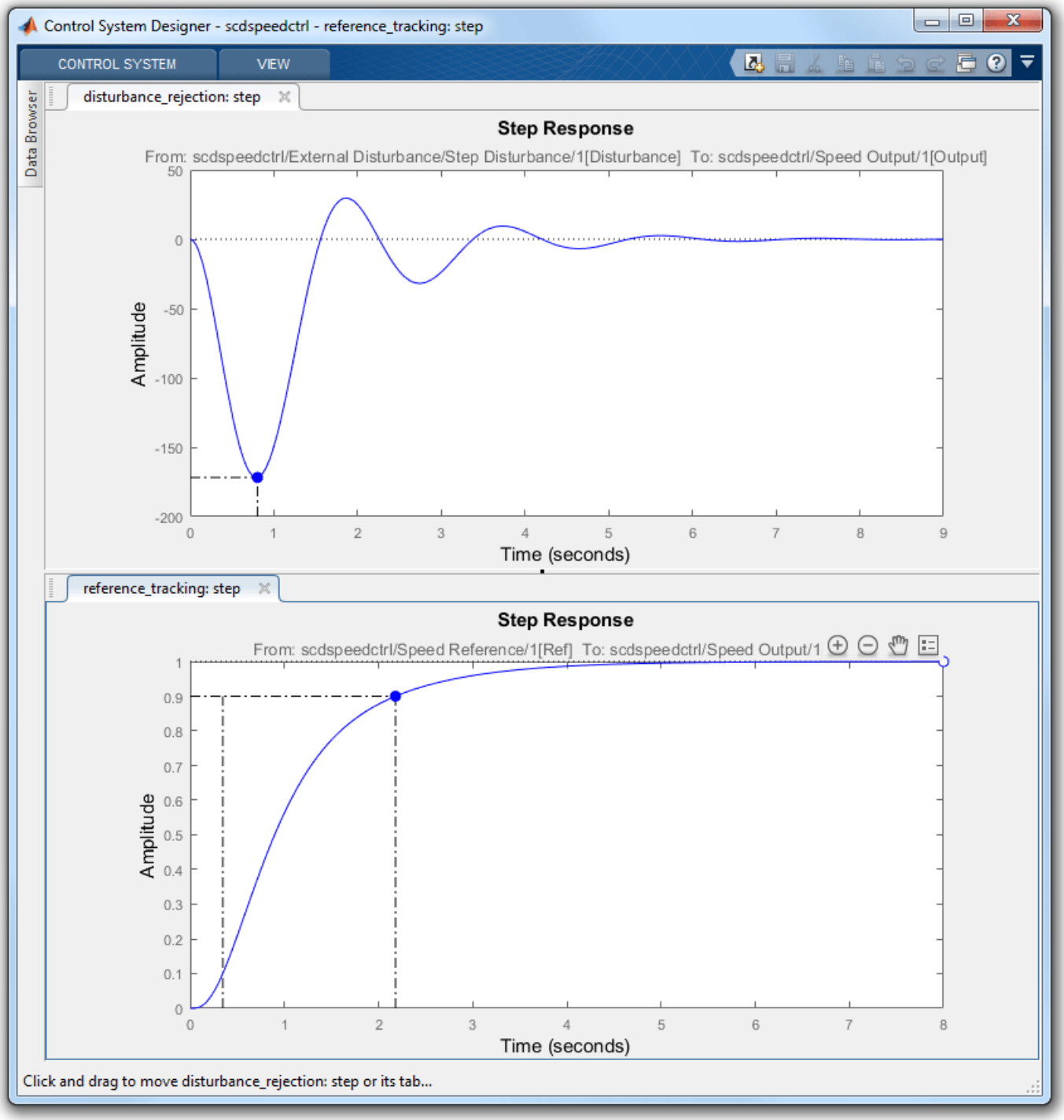
- `scdspeedctrl/PID Controller`:

```
P = 0.0012191
I = 0.0030038
```

- `scdspeedctrl/Reference Filter`:

```
Numerator = 10
Denominator = [1 10]
```

In the Compensator Editor dialog box, specify these parameters. Then, in **Control System Designer**, view the closed-loop responses.



Update Simulink Model

To write the compensator parameters back to the Simulink model, click **Update Blocks**. You can then test your design on the nonlinear model.

```
bdclose('scdspeedctrl')
```

See Also

Control System Designer

More About

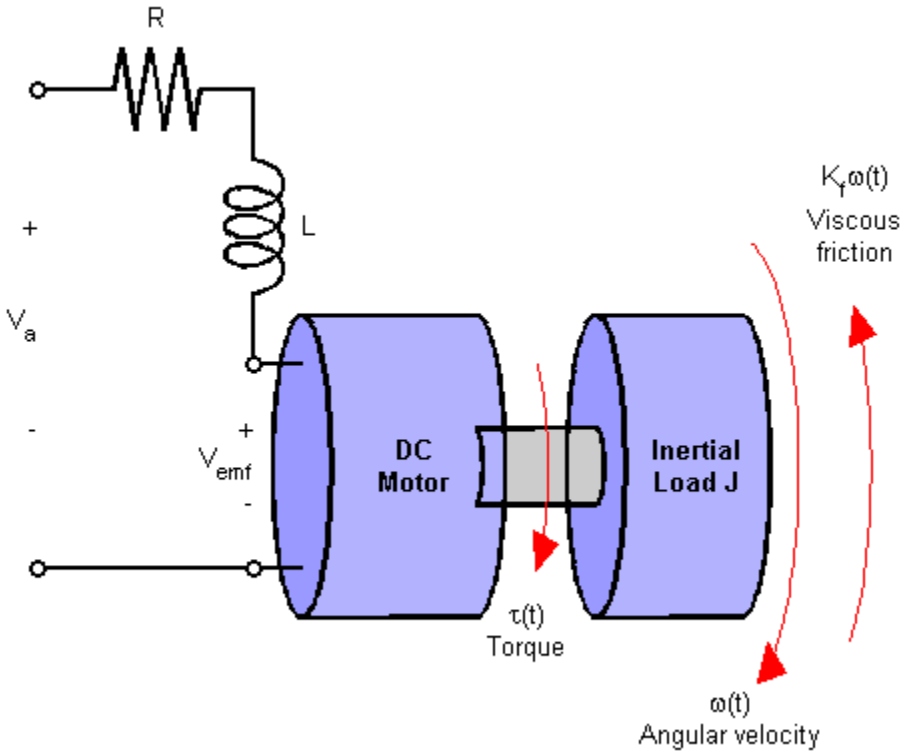
- “Edit Compensator Dynamics”
- “Update Simulink Model and Validate Design” on page 9-40
- “Control System Designer Tuning Methods” on page 9-4
- “Analyze Designs Using Response Plots” on page 9-27

Reference Tracking of DC Motor with Parameter Variations

This example shows how to generate an array of LTI models that represent the plant variations of a control system from a Simulink® model. This array of models is used in **Control System Designer** for control design.

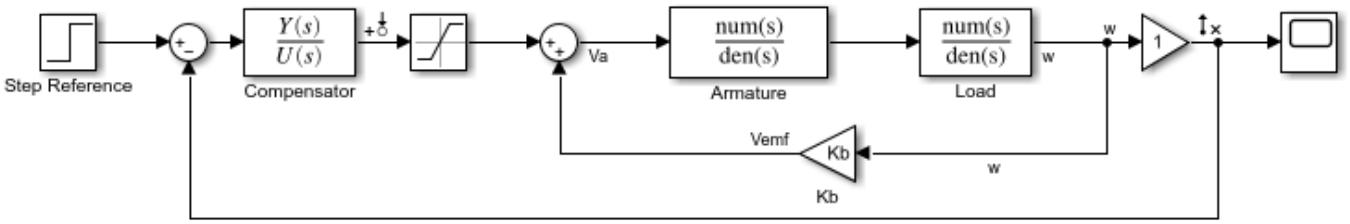
DC Motor Model

In armature-controlled DC motors, the applied voltage `Va` controls the angular velocity ω of the shaft. A simplified model of the DC motor is shown below.



Open the Simulink model for the DC motor.

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



Copyright 2013 The MathWorks, Inc.

Perform Batch Linearization

The goal of the controller is to provide tracking to step changes in reference angular velocity.

For this example, the physical constants for the motor are:

- `R` = 2.0 +/- 10% Ohms
- `L` = 0.5 Henrys
- `Km` = 0.1 Torque constant
- `Kb` = 0.1 Back emf constant
- `Kf` = 0.2 Nms
- `J` = 0.02 +/- .01 kg m^2/s^2

Note that parameters `R` and `J` are specified as a range of values.

To design a controller which will work for all physical parameter values, create a representative set of plants by sampling these values.

For parameters `R` and `J`, use their nominal, minimum, and maximum values.

```
R = [2,1.8,2.2];
J = [.02,.03,.01];
```

To create an LTI array of plant models, batch linearize the DC motor plant. For each combination of the sample values of `R` and `J`, linearize the Simulink model. To do so, specify a linearization input point at the output of the controller block and a linearization output point with a loop opening at the output of the load block as shown in the model.

Get the linearization analysis points specified in the model.

```
io = getlinio(mdl);
```

Vary the plant parameters `R` and `J`.

```
[R_grid,J_grid] = ndgrid(R,J);
params(1).Name = 'R';
params(1).Value = R_grid;
params(2).Name = 'J';
params(2).Value = J_grid;
```

Linearize the model for each parameter value combination.

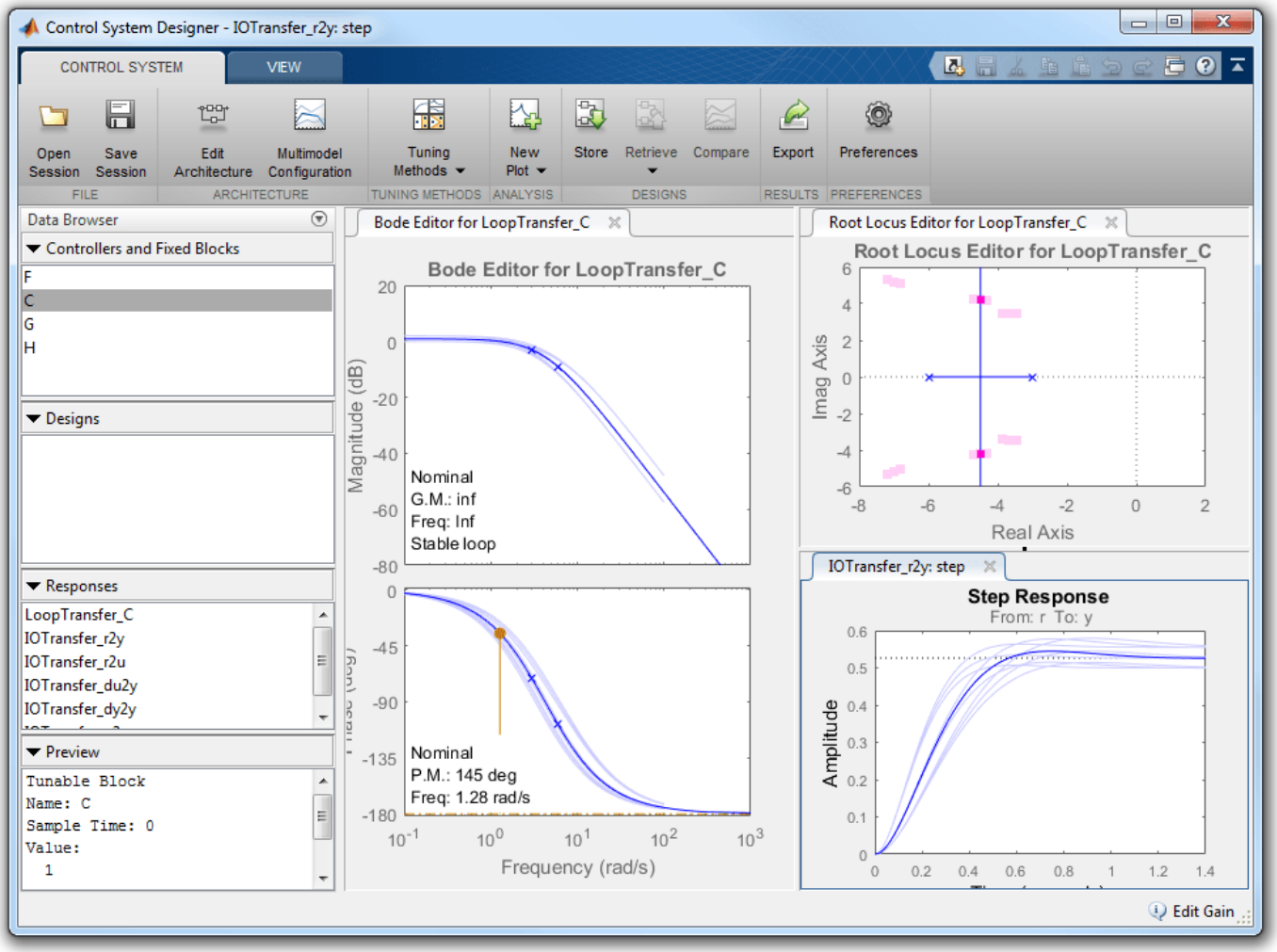
```
sys = linearize(mdl,io,params);
```

Open Control System Designer

Open **Control System Designer**, and import the array of plant models. using the following command.

```
controlSystemDesigner(sys)
```

Using **Control System Designer**, you can design a controller for the nominal plant model while simultaneously visualizing the effect on the other plant models as shown below.



The root locus editor displays the root locus for the nominal model and the closed-loop pole locations associated with the other plant models.

The Bode editor displays both the nominal model response and the responses of the other plant models.

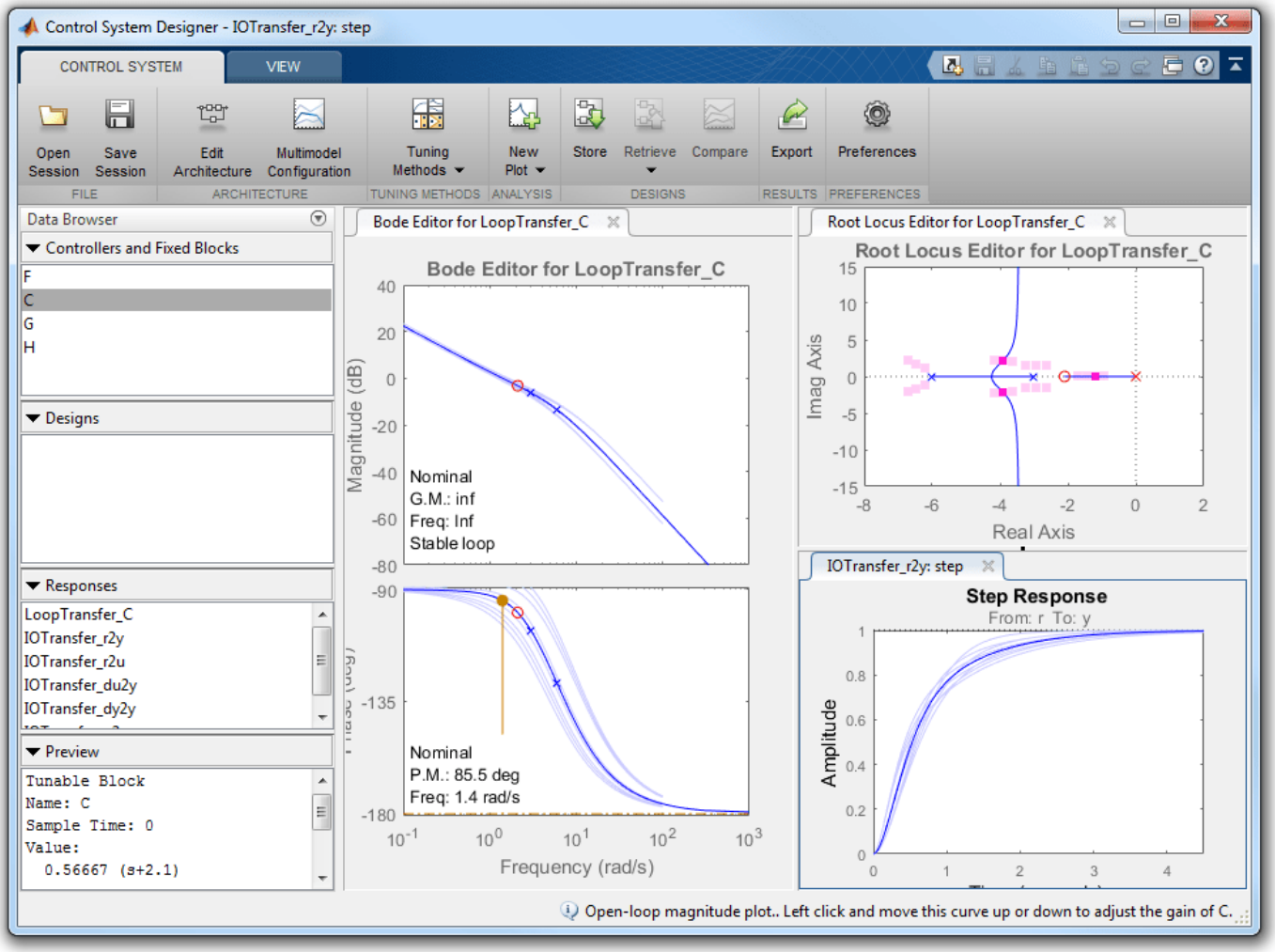
The step responses show that reference tracking is not achieved for any of the plant models.

Design Controller

Using the tools in **Control System Designer**, design the following compensator for reference tracking.

$$C(s) = 1.19\frac{(s+2.1)}{s}$$

The resulting design is shown below. The closed-loop step response shows that the goal of reference tracking is achieved with zero steady-state error for all models defined in the plant set. However, if a zero percent overshoot requirement is necessary, not all responses would satisfy this requirement.



Export Design and Validate in Simulink Model

To export the designed controller to the MATLAB workspace, click **Export**. In the Export Model dialog box, select **C**, and click **Export**. Write the controller parameters to the Simulink model.

```
[Cnum,Cden] = tfdata(C,'v');
hws = get_param(mdl, 'modelworkspace');
assignin(hws,'Cnum',Cnum)
assignin(hws,'Cden',Cden)
```

More Information

For more information on using the multimodel features of **Control System Designer**, see "Multimodel Control Design".

```
bdclose('scdDCMotor')
```

See Also

Control System Designer

Related Examples

- "Control System Designer Tuning Methods" on page 9-4
- "Analyze Designs Using Response Plots" on page 9-27

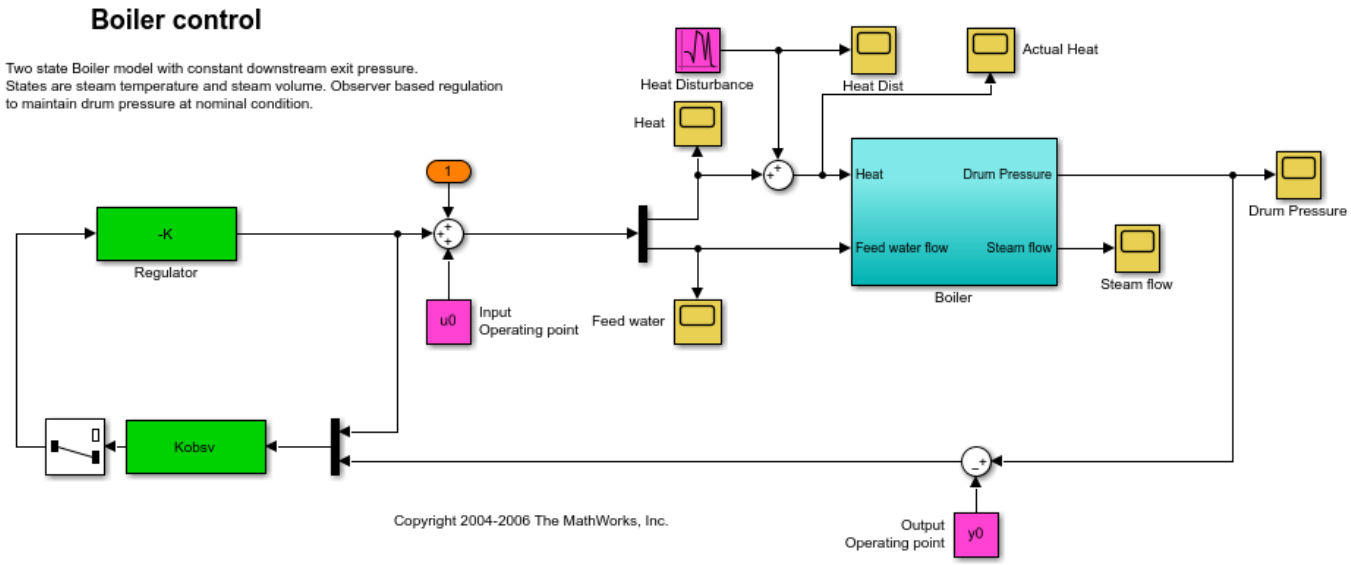
Regulate Pressure in Drum Boiler

This example shows how to use Simulink® Control Design™ software, using a drum boiler as an example application. Using the operating point search function, the example illustrates model linearization as well as subsequent state observer and LQR design. In this drum-boiler model, the control problem is to regulate boiler pressure in the face of random heat fluctuations from the furnace by adjusting the feed water flow rate and the nominal heat applied. For this example, 95% of the random heat fluctuations are less than 50% of the nominal heating value, which is not unusual for a furnace-fired boiler.

Open the Model

Open the Simulink model.

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



When you open boiler control model the software initializes the controller sizes. `u0` and `y0` are set after the operating point computation and are therefore initially set to zero. The observer and regulator are computed during the controller design step and are also initially set to zero.

Find Nominal Operating Point and Linearize Model

The model initial state values are defined in the Simulink model. Using these state values, find the steady-state operating point using the `findop` function.

Create an operating point specification where the state values are known.

```
opspec = operspec(mdl);
opspec.States(1).Known = 1;
opspec.States(2).Known = 1;
opspec.States(3).Known = [1;1];
```

Adjust the operating point specification to indicate that the inputs must be computed and that they are lower-bounded.

```
opspec.Inputs(1).Known = [0;0];     % Inputs unknown
opspec.Inputs(1).Min = [0;0];       % Input minimum value
```

Add an output specification to the operating point specification; this is necessary to ensure that the output operating point is computed during the solution process.

```
opspec = addoutputspec(opspec,[mdl '/Boiler'],1);
opspec.Outputs(1).Known = 0;    % Outputs unknown
opspec.Outputs(1).Min = 0;      % Output minimum value
```

Compute the operating point, and generate an operating point search report.

```
[opSS,opReport] = findop(mdl,opspec);
```

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

opreport =


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

Operating point specifications were successfully met.
States:
----------
    <strong>Min</strong>     <strong>x</strong>     <strong>Max</strong>    <strong>dxMin</strong>
    <strong>___</strong>    <strong>___</strong>    <strong>___</strong>    <strong>_____</strong>

(1.) Boiler_Demo/Boiler/Steam volume
    5.6    5.6    5.6      0       7.8501e-13      0
(2.) Boiler_Demo/Boiler/Temperature
    180    180    180      0      -5.9262e-14      0
(3.) Boiler_Demo/Observer/Internal
      0      0      0      0                0      0
      0      0      0      0                0      0

Inputs:
----------
    <strong>Min</strong>        <strong>u</strong>         <strong>Max</strong>
    <strong>___</strong>    <strong>__________</strong>    <strong>___</strong>

(1.) Boiler_Demo/Input
     0     2.4107e+05    Inf
     0         100.13    Inf

Outputs:
----------
    <strong>Min</strong>      <strong>y</strong>       <strong>Max</strong>
    <strong>___</strong>    <strong>______</strong>    <strong>___</strong>

(1.) Boiler_Demo/Boiler
     0     1002.4    Inf
```

Before linearizing the model around this point, specify the input and output signals for the linear model. First, specify the input points for linearization.

```
Boiler_io(1) = linio([mdl '/Sum'],1,'input');
Boiler_io(2) = linio([mdl '/Demux'],2,'input');
```

Next, specify the open-loop output points for linearization.

```
Boiler_io(3) = linio([mdl '/Boiler'],1,'openoutput');
setlinio(mdl,Boiler_io);
```

Find a linear model around the chosen operating point.

```
Lin_Boiler = linearize(mdl,opSS,Boiler_io);
```

Finally, using the `minreal` function, make sure that the model is a minimum realization.

```
Lin_Boiler = minreal(Lin_Boiler);
```

```
1 state removed.
```

Design Regulator and State Observer

Using this linear model, design an LQR regulator and Kalman filter state observer. First, find the controller offsets to make sure that the controller is operating around the chosen linearization point by retrieving the computed operating point.

```
u0 = opReport.Inputs.u;
y0 = opReport.Outputs.y;
```

Now, design the regulator using the `lqry` function. Tight regulation of the output is required while input variation should be limited.

```
Q = diag(1e8);                  % Output regulation
R = diag([1e2,1e6]);            % Input limitation
[K,S,E] = lqry(Lin_Boiler,Q,R);
```

Design the Kalman state observer using the `kalman` function. For this example, the main noise source is process noise. The noise enters the system only through one input, hence the form of `G` and `H`.

```
[A,B,C,D] = ssdata(Lin_Boiler);
G = B(:,1);
H = 0;
QN = 1e4;
RN = 1e-1;
NN = 0;
[Kobsv,L,P] = kalman(ss(A,[B G],C,[D H]),QN,RN);
```

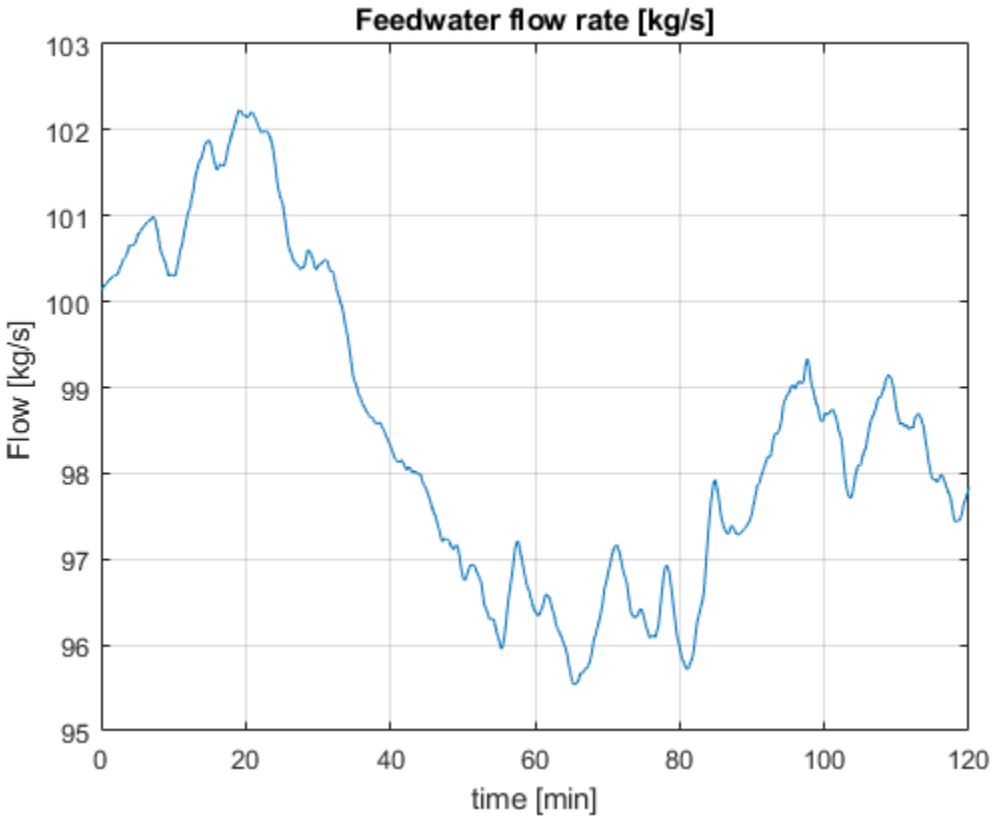
Simulate Model

Simulate the model for the designed controller.

```
sim(mdl)
```

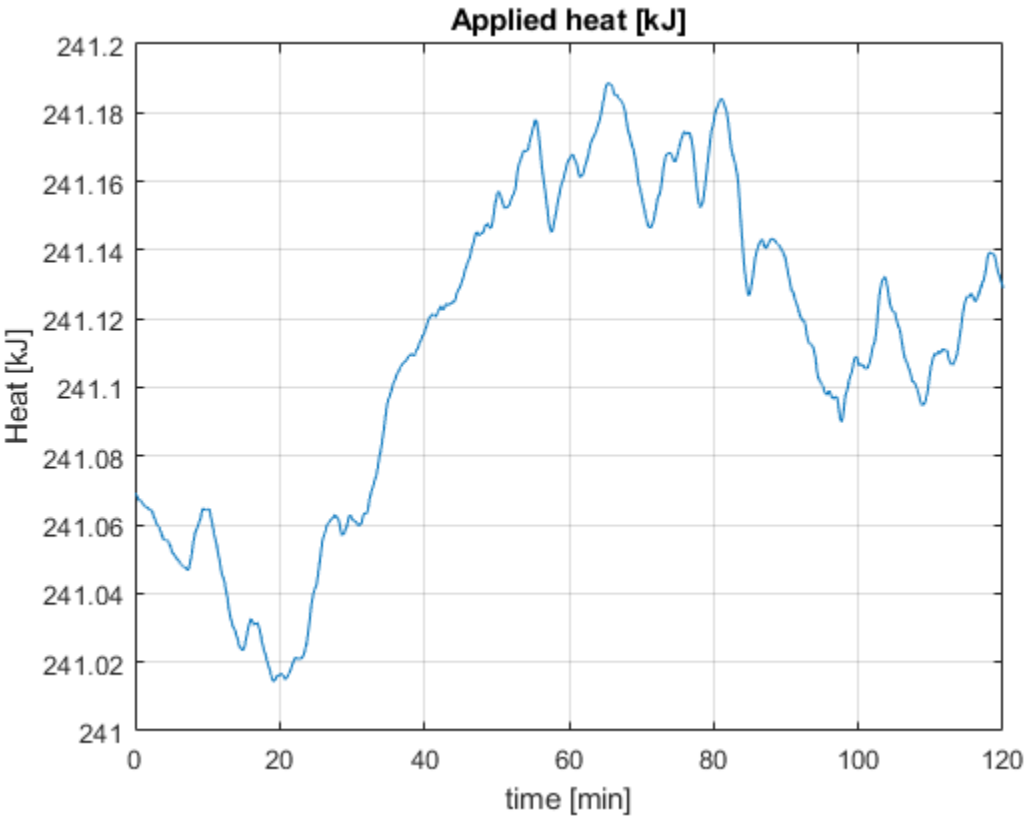
Plot the process input and output signals. The following figure shows the feed water actuation signal in kg/s.

```
plot(FeedWater.time/60,FeedWater.signals.values)
title('Feedwater flow rate [kg/s]');
ylabel('Flow [kg/s]')
xlabel('time [min]')
grid on
```



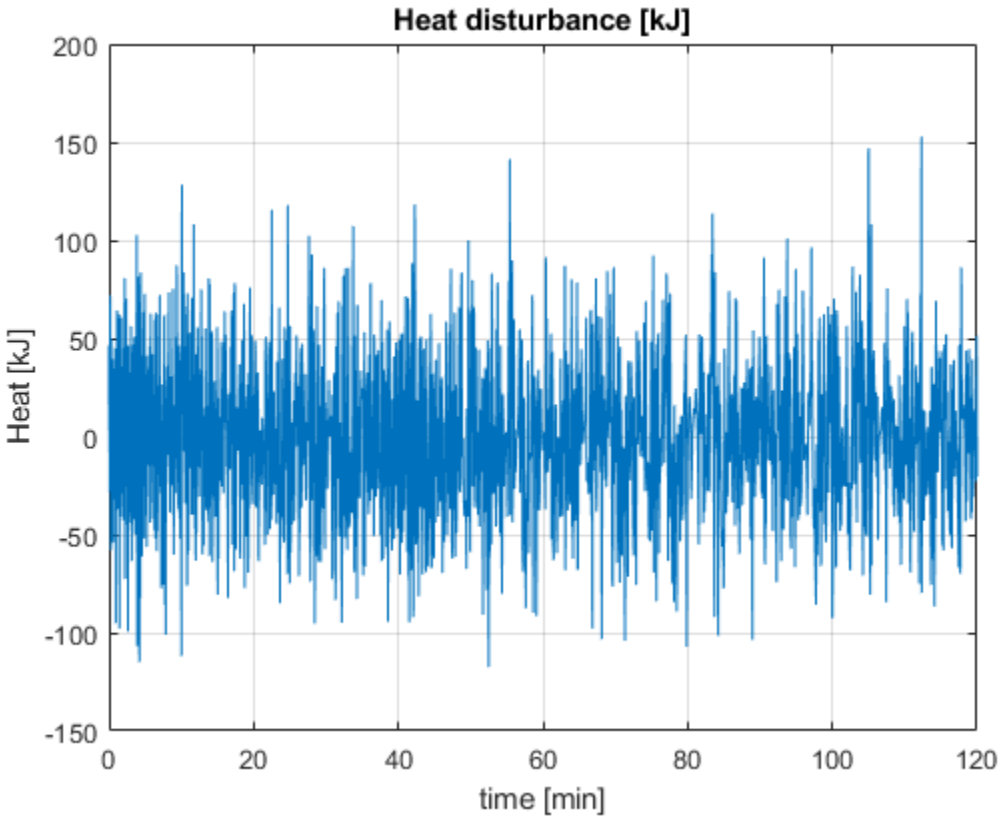
The following plot shows the heat actuation signal in kJ.

```
plot(Heat.time/60,Heat.signals.values/1000)
title('Applied heat [kJ]');
ylabel('Heat [kJ]')
xlabel('time [min]')
grid on
```



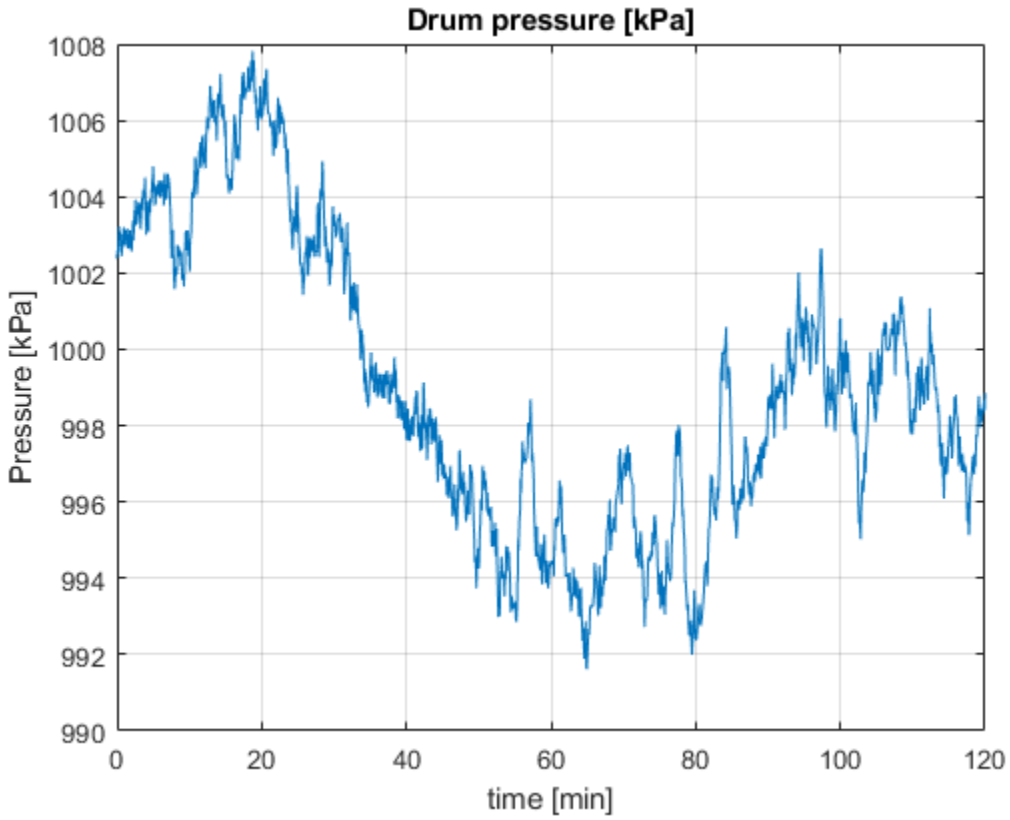
The next figure shows the heat disturbance in kJ. The disturbance varies by as much as 50% of the nominal heat value.

```
plot(HeatDist.time/60,HeatDist.signals.values/1000)
title('Heat disturbance [kJ]');
ylabel('Heat [kJ]')
xlabel('time [min]')
grid on
```



The figure below shows the corresponding drum pressure in kPa. The pressure varies by about 1% of the nominal value even though the disturbance is relatively large.

```
plot(DrumPressure.time/60,DrumPressure.signals.values)
title('Drum pressure [kPa]');
ylabel('Pressure [kPa]')
xlabel('time [min]')
grid on

bdclose(mdl)
```



See Also

`kalman` | `lqry` | `findop` | `linearize`

Related Examples

- “Kalman Filtering”
- “Linear-Quadratic-Gaussian (LQG) Design”

Model Computational Delay and Sampling Effects

This example shows how to model computational delay and sampling effects using Simulink® Control Design™ software.

Computational delays and sampling effects can critically effect the performance of a control system. Typically, the closed-loop responses of a system become oscillatory and unstable if these factors are not taken into account. Therefore, when modeling a control system, you should include computational delays and sampling effects to accurately design and simulate a closed-loop system.

There are two approaches for designing compensators with the effects of computational delay and sampling. The first approach is to design a controller in the discrete domain to capture the effects of sampling by discretizing the plant. The second approach is to design a controller in the continuous domain. This approach is sometimes more convenient, but in this case you must account for the effects of computational delay and sampling. In this example, you apply both approaches to redesign a control system using Simulink Control Design software.

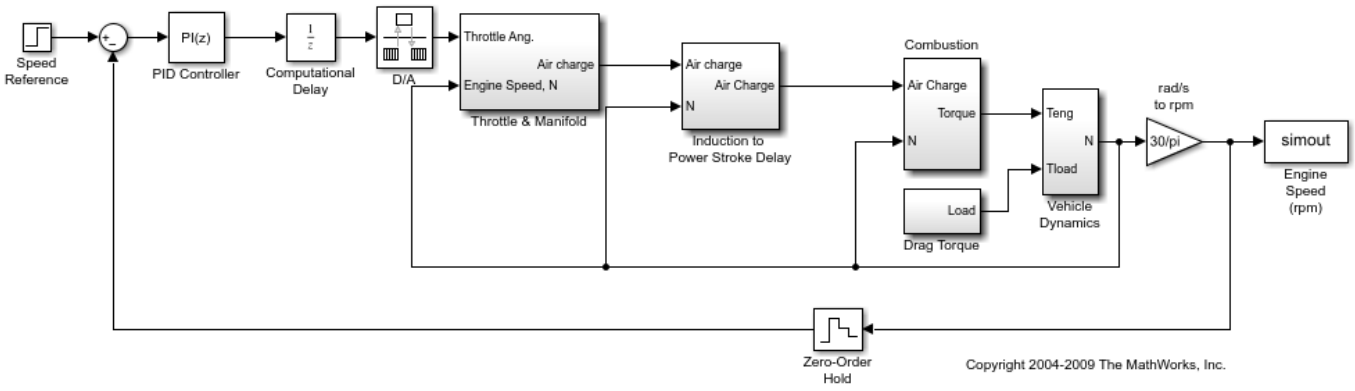
Simulate Discrete and Continuous Controllers

In the following example model, the initial compensator has the following gains from the compensator in the example “Single Loop Feedback/Prefilter Compensator Design” on page 9-41.

```
P = 0.0018222
I = 0.0052662
```

The first model has a discrete implementation of the control system.

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



In this model, the Computational Delay block models the effects of the computational delay. The delay is equal to the sample time of the controller, which is the worst case. The Zero-Order Hold block models the effect of sampling on the response of the system. Finally, the speed controller, which is implemented using a PID Controller block, is discretized using a Forward Euler sampling method.

You can see the effect of the sampling by simulating the response of the system.

First, discretize the controller with a sample time of Ts = 0.1 seconds and simulate the model.

```
Ts = 0.1;
sim(mdl);
```

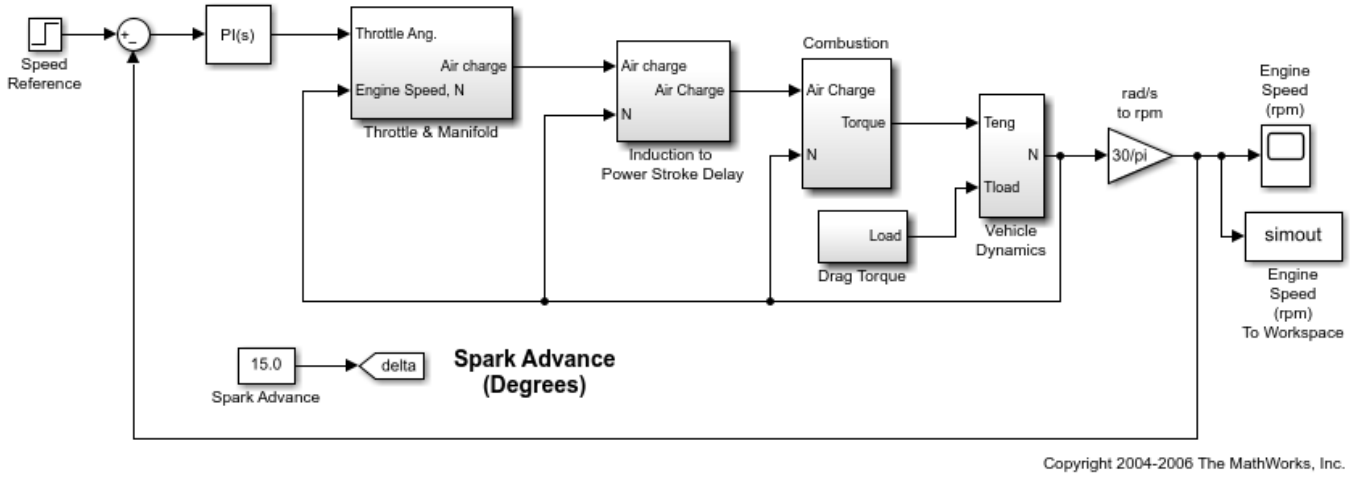
```
T2 = simout.time;
Y2 = simout.signals.values;
```

Next, simulate the model with an increased sample time of Ts = 0.25 seconds.

```
Ts = 0.25;
sim(mdl);
T3 = simout.time;
Y3 = simout.signals.values;
```

The second model is a continuous model.

```
mdl_continuous = 'scdspeed_contcomp';
open_system(mdl_continuous)
```

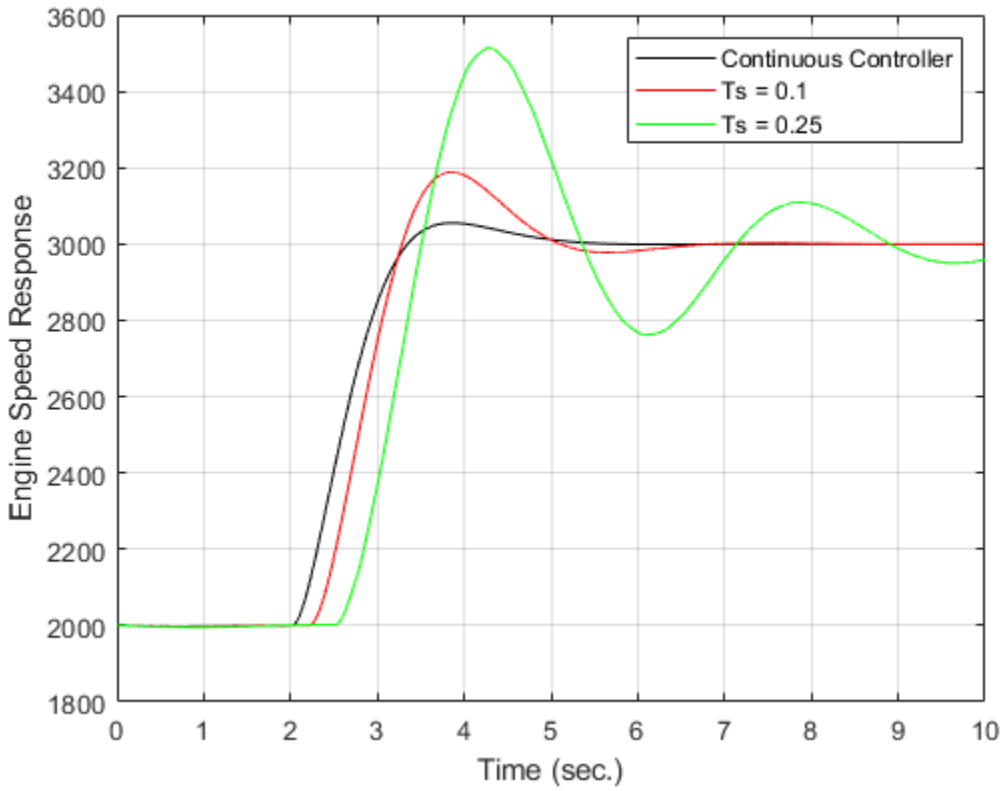


Simulate the response of the continuous model.

```
sim(mdl_continuous);
T1 = simout.time;
Y1 = simout.signals.values;
```

Plot the simulation of both the discrete and continuous models. The response becomes more oscillatory as the sample time increases.

```
plot(T1,Y1,'k',T2,Y2,'r',T3,Y3,'g')
xlabel('Time (sec.)')
ylabel('Engine Speed Response')
legend('Continuous Controller','Ts = 0.1','Ts = 0.25')
grid
```



Design Compensator in Discrete Domain

To remove the oscillatory effects of the closed-loop system with the slowest sample time of Ts = 0.25, you must redesign the compensator.

First, redesign the controller using a discretized version of the plant. You can redesign the compensator in a fashion similar to “Single Loop Feedback/Prefilter Compensator Design” on page 9-41. The tuned compensator has the following gains.

```
P = 0.00066155
I = 0.0019118795

set_param('scdspeed_compdelay/PID Controller','P','0.00066155')
set_param('scdspeed_compdelay/PID Controller','I','0.0019118795')
```

Simulate the resulting closed-loop system with a sample time of Ts = 0.25 seconds. You examine these results later in the example.

```
Ts = 0.25;
sim(mdl);
Td = simout.time;
Yd = simout.signals.values;
```

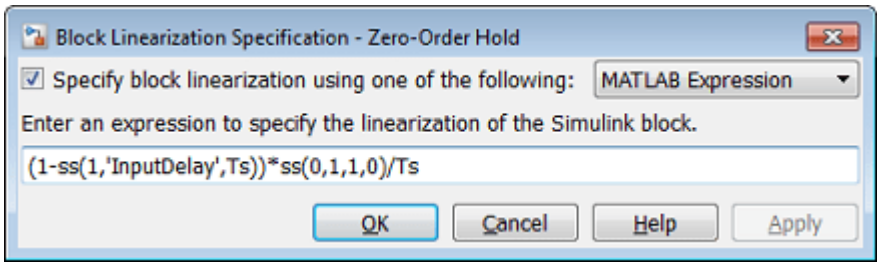
Account for Delays and Sampling in Continuous Domain

As a second approach, redesign the controller with the continuous equivalents of the unit delay and zero-order hold.

For this example, use the following zero-order hold dynamics.

$$ZOH(s) = \frac{1 - e^{-sT_s}}{sT_s}$$

To specify these dynamics in the model, right-click the Zero-Order Hold block and select **Linear Analysis > Specify Selected Block Linearization**. In the Block Linearization dialog box, specify the expression for the zero-order hold dynamics.



Equivalently, you can specify the block linearization using the following code.

```
zohblk = 'scdspeed_compdelay/Zero-Order Hold';
set_param(zohblk,'SCDEnableBlockLinearizationSpecification','on')
rep = struct('Specification','(1-ss(1,''InputDelay'',Ts))*ss(0,1,1,0)/Ts',...
             'Type','Expression',...
             'ParameterNames','',...
             'ParameterValues','');
set_param(zohblk,'SCDBlockLinearizationSpecification',rep)
```

Specify the linearization of the Computational Delay using a continuous transport delay.

$$DELAY(s) = e^{-sT_s}$$

Specify the linearization for the delay block using the following code.

```
delayblk = 'scdspeed_compdelay/Computational Delay';
set_param(delayblk,'SCDEnableBlockLinearizationSpecification','on')
rep = struct('Specification','ss(1,''InputDelay'',Ts)',...
             'Type','Expression',...
             'ParameterNames','',...
             'ParameterValues','');
set_param(delayblk,'SCDBlockLinearizationSpecification',rep)
```

With the continuous-time models for the delay and sampling effects, the analysis of the controller design then remains in the continuous domain.

Next, you linearize the model with delays of Ts = 0.1 and 0.25 seconds. To do so, first set the input and output linear analysis points.

```
io(1) = linio('scdspeed_compdelay/PID Controller',1,'input');
io(2) = linio('scdspeed_compdelay/Zero-Order Hold',1,'openoutput');
```

Linearize the model at Ts = 0.1.

```
Ts = 0.1;
sys2 = linearize(mdl,io);
```

Linearize the model at Ts = 0.25.

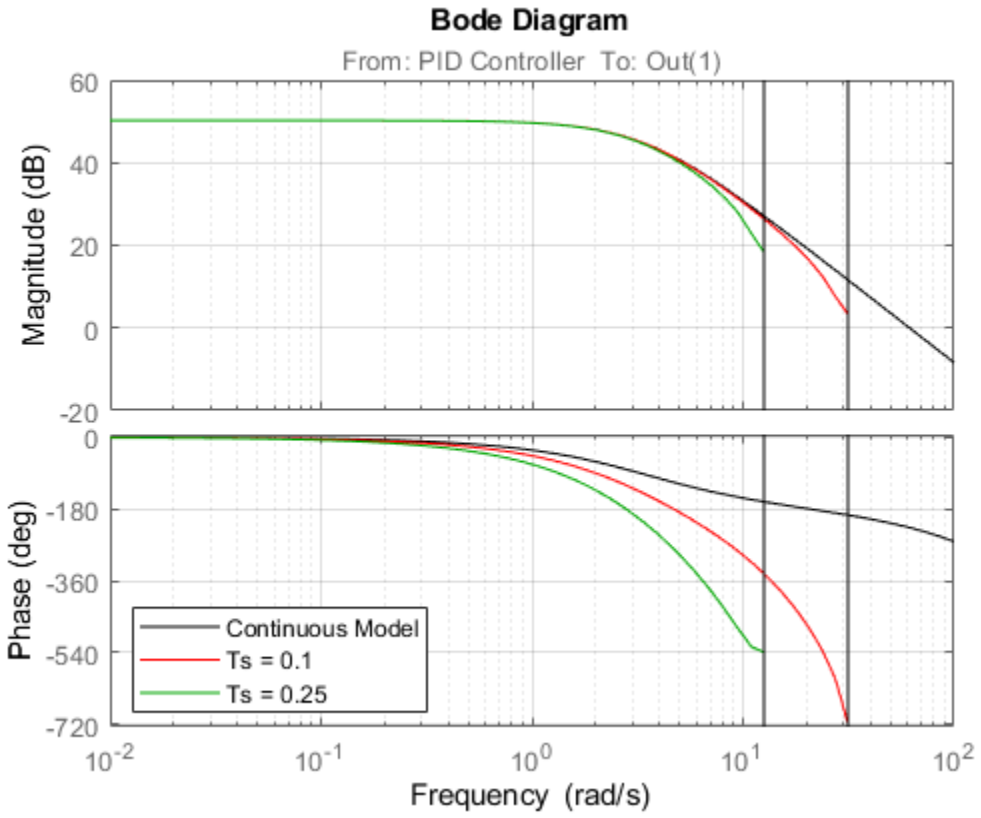
```
Ts = 0.25;
sys3 = linearize(mdl,io);
```

Finally, linearize the model without the effects of sampling and the computational delay.

```
io(1) = linio('scdspeed_contcomp/PID Controller',1,'input');
io(2) = linio('scdspeed_contcomp/rad//s to rpm',1,'openoutput');
sys1 = linearize(mdl_continuous,io);
```

You can use the linear models of the engine to examine the effects of the computational delay on the frequency response. In this case, the phase response of the system is significantly reduced due to the delay introduced by sampling.

```
p = bodeoptions('cstprefs');
p.Grid = 'on';
p.PhaseMatching = 'on';
bodeplot(sys1,'k',sys2,'r',sys3,'g', {1e-2,1e2},p)
legend('Continuous Model','Ts = 0.1','Ts = 0.25','Location','southwest')
```



Using the model with the slowest sample time, redesign the compensator using the techniques in “Single Loop Feedback/Prefilter Compensator Design” on page 9-41. Doing so gives the following PI Gains.

```
P = 0.00065912
I = 0.001898342
```

Set these gains in the controller.

```
set_param('scdspeed_compdelay/PID Controller','P','0.00065912')
set_param('scdspeed_compdelay/PID Controller','I','0.001898342')
```

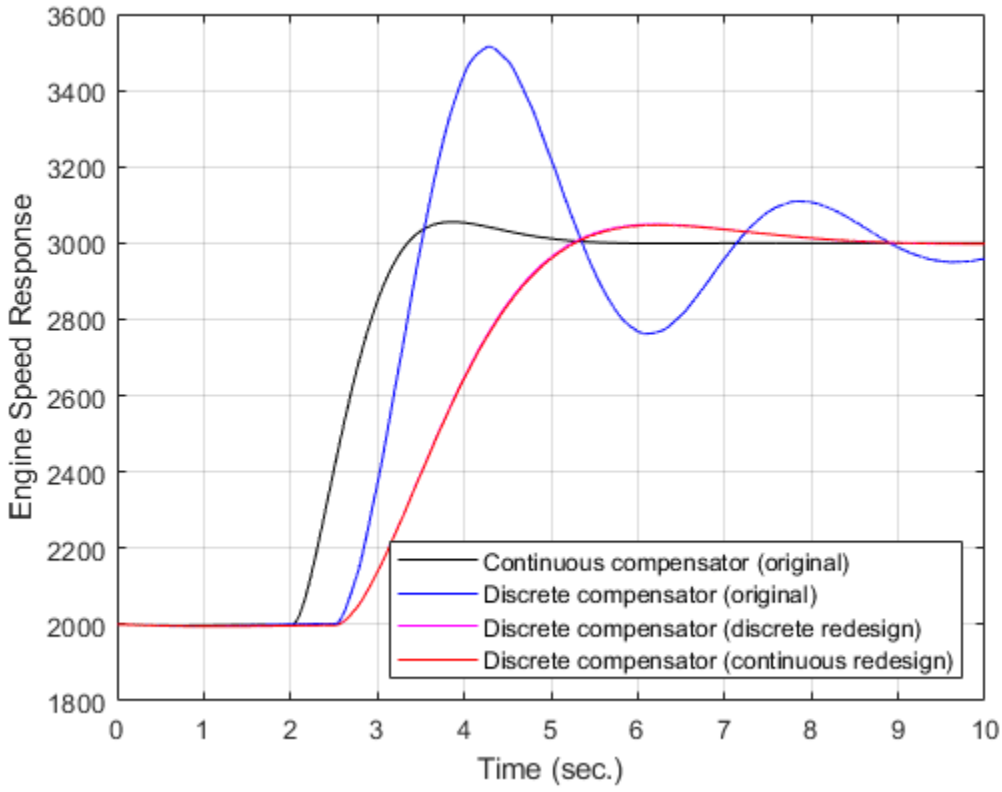
Simulate the resulting closed-loop system with a sample time Ts = 0.25.

```
sim(mdl)
Tc = simout.time;
Yc = simout.signals.values;
```

Compare Responses

Plot the responses of the design. The redesign of the control system using both approaches yields similar controllers. This example shows the effects of the computational delay and discretization. These effects reduce the stability margins of the system, but when you properly model a control system you can achieve the desired closed-loop behavior.

```
plot(T1,Y1,'k',T3,Y3,'b',Td,Yd,'m',Tc,Yc,'r')
xlabel('Time (sec.)')
ylabel('Engine Speed Response')
legend('Continuous compensator (original)','Discrete compensator (original)', ...
    'Discrete compensator (discrete redesign)',...
    'Discrete compensator (continuous redesign)',...
    'Location','southeast')
grid
```



```
bdclose('scdspeed_contcomp')
bdclose('scdspeed_compdelay')
```

See Also

Control System Designer

Related Examples

- “Designing Compensators for Plants with Time Delays” on page 9-9
- “Control System Designer Tuning Methods” on page 9-4
- “Single Loop Feedback/Prefilter Compensator Design” on page 9-41

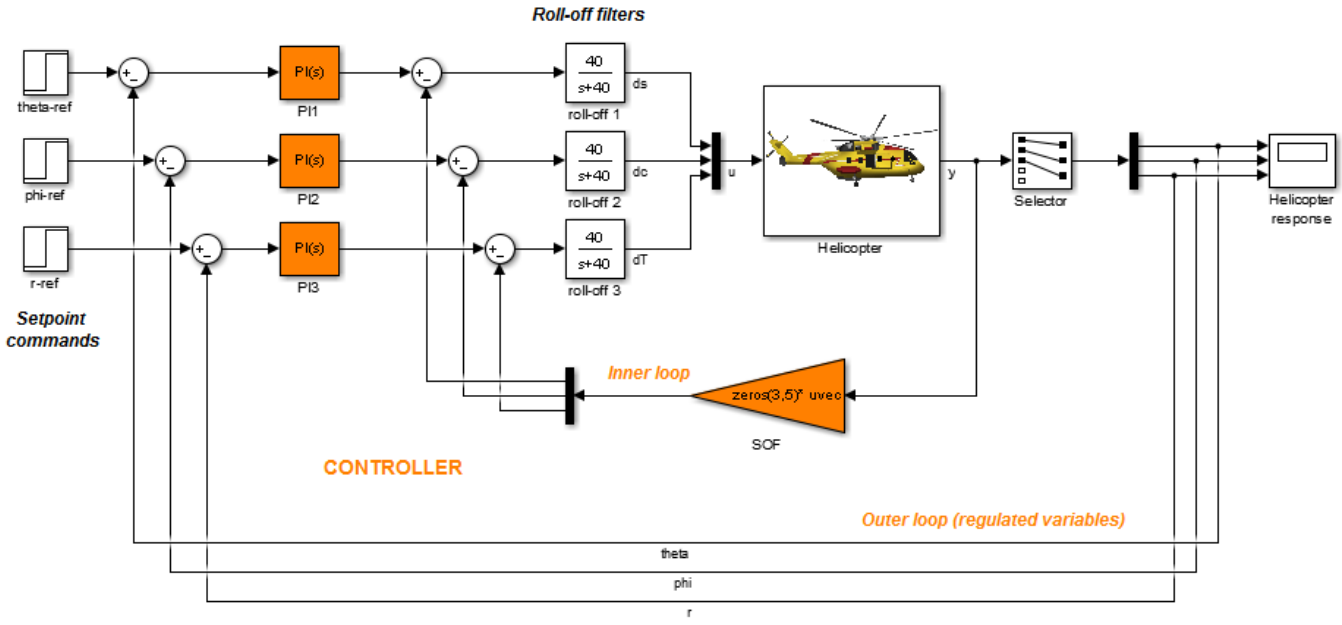
Control System Tuning

- “Automated Tuning Overview” on page 10-3
- “Choosing an Automated Tuning Approach” on page 10-4
- “Automated Tuning Workflow” on page 10-6
- “Specify Control Architecture in Control System Tuner” on page 10-7
- “Open Control System Tuner for Tuning Simulink Model” on page 10-10
- “Specify Operating Points for Tuning in Control System Tuner” on page 10-11
- “Specify Blocks to Tune in Control System Tuner” on page 10-17
- “View and Change Block Parameterization in Control System Tuner” on page 10-19
- “Setup for Tuning Control System Modeled in MATLAB” on page 10-26
- “How Tuned Simulink Blocks Are Parameterized” on page 10-27
- “Specify Goals for Interactive Tuning” on page 10-29
- “Quick Loop Tuning of Feedback Loops in Control System Tuner” on page 10-34
- “Quick Loop Tuning” on page 10-42
- “Step Tracking Goal” on page 10-45
- “Step Rejection Goal” on page 10-50
- “Transient Goal” on page 10-54
- “LQR/LQG Goal” on page 10-59
- “Gain Goal” on page 10-63
- “Variance Goal” on page 10-67
- “Reference Tracking Goal” on page 10-71
- “Overshoot Goal” on page 10-76
- “Disturbance Rejection Goal” on page 10-80
- “Sensitivity Goal” on page 10-85
- “Weighted Gain Goal” on page 10-89
- “Weighted Variance Goal” on page 10-92
- “Minimum Loop Gain Goal” on page 10-96
- “Maximum Loop Gain Goal” on page 10-101
- “Loop Shape Goal” on page 10-106
- “Margins Goal” on page 10-111
- “Passivity Goal” on page 10-115
- “Conic Sector Goal” on page 10-119
- “Weighted Passivity Goal” on page 10-124
- “Poles Goal” on page 10-128
- “Controller Poles Goal” on page 10-132
- “Manage Tuning Goals” on page 10-135

- “Generate MATLAB Code from Control System Tuner for Command-Line Tuning” on page 10-136
- “Interpret Numeric Tuning Results” on page 10-139
- “Visualize Tuning Goals” on page 10-143
- “Create Response Plots in Control System Tuner” on page 10-149
- “Examine Tuned Controller Parameters in Control System Tuner” on page 10-155
- “Compare Performance of Multiple Tuned Controllers” on page 10-157
- “Create and Configure slTuner Interface to Simulink Model” on page 10-161
- “Stability Margins in Control System Tuning” on page 10-165
- “Tune Control System at the Command Line” on page 10-170
- “Speed Up Tuning with Parallel Computing Toolbox Software” on page 10-171
- “Validate Tuned Control System” on page 10-172
- “Extract Responses from Tuned MATLAB Model at the Command Line” on page 10-176

Automated Tuning Overview

The control system tuning tools such as `systune` and **Control System Tuner** automatically tune control systems from high-level tuning goals you specify, such as reference tracking, disturbance rejection, and stability margins. The software jointly tunes all the free parameters of your control system regardless of control system architecture or the number of feedback loops it contains. For example, the model of the following illustration represents a multiloop control system for a helicopter.



This control system includes a number of fixed elements, such as the helicopter model itself and the roll-off filters. The inner control loop provides static output feedback for decoupling. The outer loop includes PI controllers for setpoint tracking. The tuning tools jointly optimize the gains in the `SOF` and `PI` blocks to meet setpoint tracking, stability margin, and other requirements that you specify. These tools allow you to specify any control structure and designate which blocks in your system are tunable.

Control systems are tuned to meet your specific performance and robustness goals subject to feasibility constraints such as actuator limits, sensor accuracy, computing power, or energy consumption. The library of tuning goals lets you capture these objectives in a form suitable for fast automated tuning. This library includes standard control objectives such as reference tracking, disturbance rejection, loop shapes, closed-loop damping, and stability margins. Using these tools, you can perform multi-objective tuning of control systems having any structure.

See Also

`systune` | **Control System Designer**

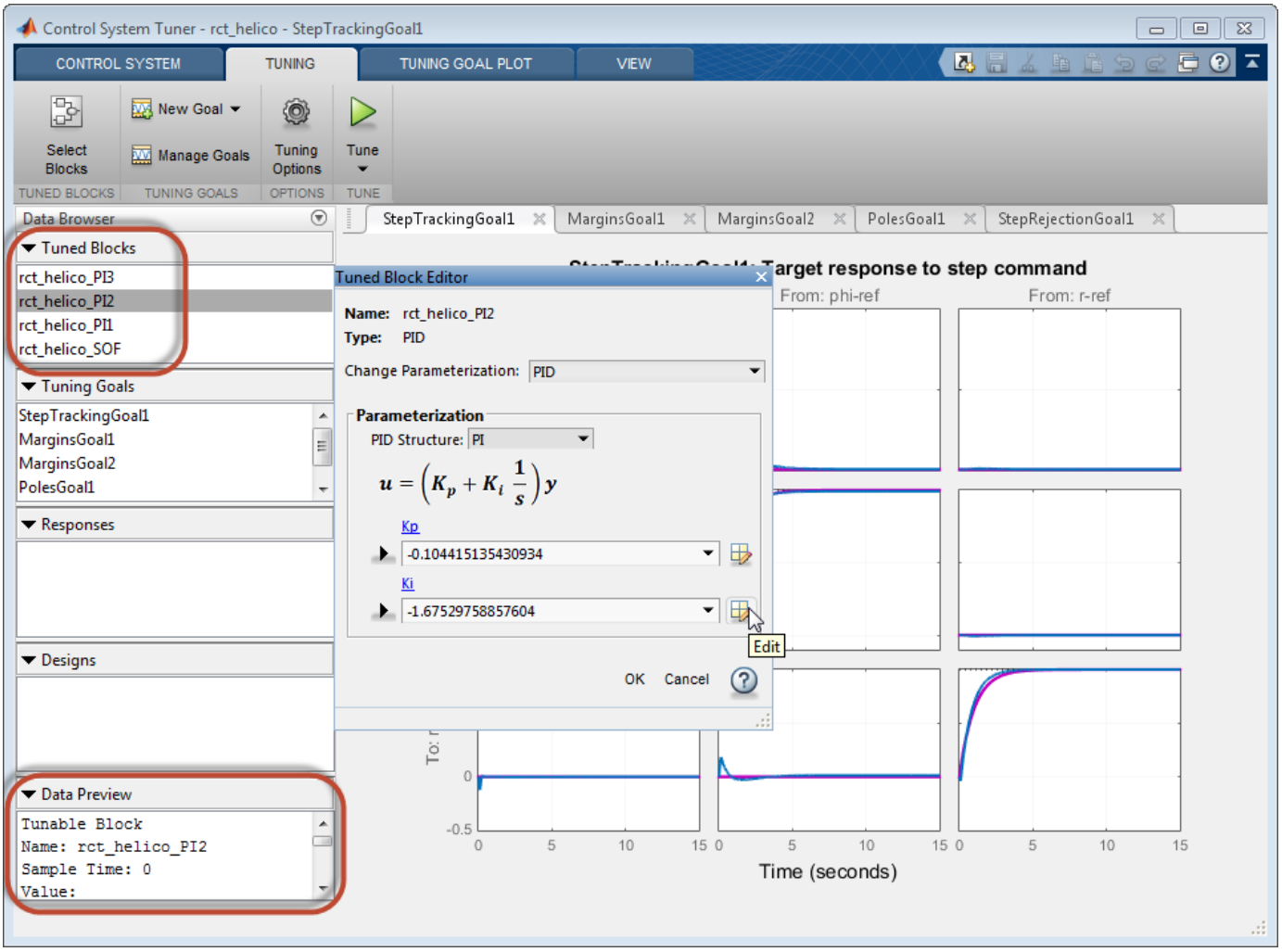
More About

- “Choosing an Automated Tuning Approach” on page 10-4
- “Automated Tuning Workflow” on page 10-6

Choosing an Automated Tuning Approach

You can tune control systems at the MATLAB command line or using the Control System Tuner app.

Control System Tuner provides an interactive graphical interface for specifying your tuning goals and validating the performance of the tuned control system.



Use **Control System Tuner** to tune control systems consisting of any number of feedback loops, with tunable components having any structure (such as PID, gain block, or state-space). You can represent your control architecture in MATLAB as a tunable generalized state-space (`genss`) model. If you have Simulink Control Design software, you can tune a control system represented by a Simulink model. Use the graphical interface to configure your tuning goals, examine response plots, and validate your controller design.

The `systune` command can perform all the same tuning tasks as **Control System Tuner**. Tuning at the command line allows you to write scripts for repeated tuning tasks. `systune` also provides advanced techniques such as tuning a controller for multiple plants, or designing gain-scheduled controllers. To use the command-line tuning tools, you can represent your control architecture in MATLAB as a tunable generalized state-space (`genss`) model. If you have Simulink Control Design

software, you can tune a control system represented by a Simulink model using an `slTuner` interface. Use the `TuningGoal` requirement objects to configure your tuning goals. Analysis commands such as `getIOTransfer` and `viewGoal` let you examine and validate the performance of your tuned system.

See Also

`systune` | **Control System Designer**

More About

- “Automated Tuning Workflow” on page 10-6

Automated Tuning Workflow

Whether you are tuning a control system at the command line or using **Control System Tuner**, the basic workflow includes the following steps:

1. Define your control architecture, by building a model of your control system from fixed-value blocks and blocks with tunable parameters. You can do so in one of several ways:

   - Create a Simulink model of your control system. (Tuning a Simulink model requires Simulink Control Design software.)
   - Use a predefined control architecture available in **Control System Tuner**.
   - At the command line, build a tunable `genss` model of your control system out of numeric LTI models and tunable control design blocks.

   For more information, see “Specify Control Architecture in Control System Tuner” on page 10-7.

2. Set up your model for tuning.

   - In **Control System Tuner**, identify which blocks of the model you want to tune. See Model Setup for Control System Tuner.
   - If tuning a Simulink model at the command line, create and configure the `slTuner` interface to the model. See Model Setup for Tuning at the Command Line.

3. Specify your tuning goals. Use the library of tuning goals to capture requirements such as reference tracking, disturbance rejection, stability margins, and more.

   - In **Control System Tuner**, use the graphical interface to specify tuning goals. See Tuning Goals (Control System Tuner).
   - At the command-line, use the `TuningGoal` requirement objects to specify your tuning goals. See Tuning Goals (programmatic tuning).

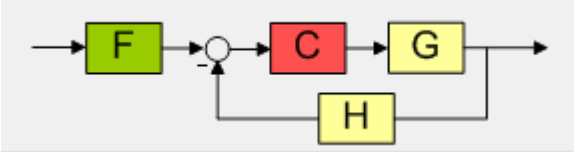
4. Tune the model. Use the `systune` command or **Control System Tuner** to optimize the tunable parameters of your control system to best meet your specified tuning goals. Then, analyze the tuned system responses and validate the design. Whether at the command line or in **Control System Tuner**, you can plot system responses to examine any aspects of system performance you need to validate your design.

   - For tuning and validating in **Control System Tuner**, see Tuning, Analysis, and Validation (Control System Tuner).
   - For tuning at the command line, see Tuning, Analysis, and Validation (programmatic tuning).

Specify Control Architecture in Control System Tuner

About Control Architecture

Control System Tuner lets you tune a control system having any architecture. Control system architecture defines how your controllers interact with the system under control. The architecture comprises the tunable control elements of your system, additional filter and sensor components, the system under control, and the interconnections among all these elements. For example, a common control system architecture is the single-loop feedback configuration of the following illustration:



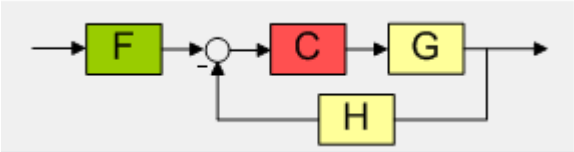
G is the plant model, and *H* the sensor dynamics. These are usually the fixed components of the control system. The prefilter *F* and feedback controller *C* are the tunable elements. Because control systems are so conveniently expressed in this block diagram form, these elements are referred to as fixed blocks and tunable blocks.

Control System Tuner gives you several ways to define your control system architecture:

- Use the predefined feedback structure of the illustration.
- Model any control system architecture in MATLAB by building a generalized state-space (`genss`) model from fixed LTI components and tunable control design blocks.
- Model your control system in Simulink and specify the blocks to tune in **Control System Tuner** (requires Simulink Control Design software).

Predefined Feedback Architecture

If your control system has the single-loop feedback configuration of the following illustration, use the predefined feedback structure built into **Control System Tuner**.



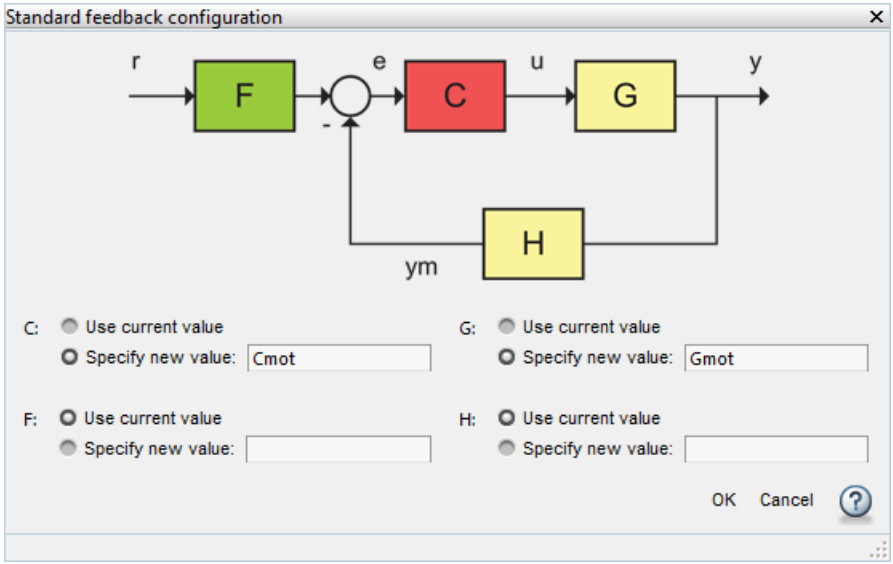
For example, suppose you have a DC motor for which you want to tune a PID controller. The response of the motor is modeled as $G(s) = 1/(s + 1)^2$. Create a fixed LTI model representing the plant, and a tunable PID controller model.

```
Gmot = zpk([],[-1,-1],1);
Cmot = tunablePID('Cmot','PID');
```

Open **Control System Tuner**.

```
controlSystemTuner
```

Control System Tuner opens, set to tune this default architecture. Next, specify the values of the blocks in the architecture. Click to open the **Standard feedback configuration** dialog box.



Enter the values for *C* and *G* that you created. **Control System Tuner** reads these values from the MATLAB workspace. Click **OK**.

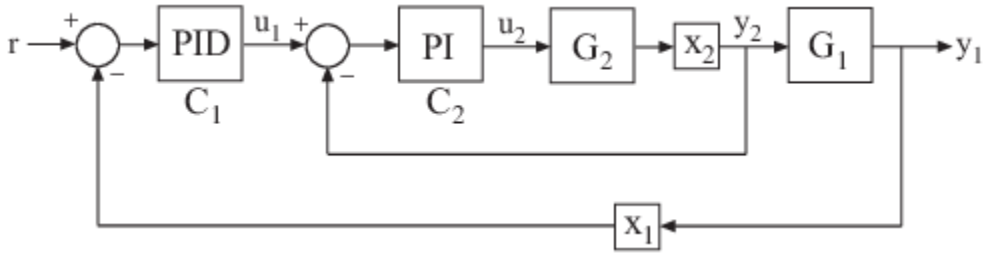
The default value for the sensor dynamics is a fixed unity-gain transfer function. The default value for the filter *F* is a tunable gain block.

You can now select blocks to tune, create tuning goals, and tune the control system.

Arbitrary Feedback Control Architecture

If your control architecture does not match the predefined control architecture of **Control System Tuner**, you can create a generalized state-space (`genss`) model with tunable components representing your controller elements. For example, suppose you want to tune the cascaded control system of the following illustration, that includes two tunable PID controllers.

.



Create tunable control design blocks for the controllers, and fixed LTI models for the plant components, G_1 and G_2. Also include optional loop-opening locations x_1 and x_2. These locations indicate where you can open loops or inject signals for the purpose of specifying requirements for tuning the system.

```
G2 = zpk([],-2,3);
G1 = zpk([],[-1 -1 -1],10);
```

```
C20 = tunablePID('C2','pi');
C10 = tunablePID('C1','pid');

X1 = AnalysisPoint('X1');
X2 = AnalysisPoint('X2');
```

Connect these components to build a model of the entire closed-loop control system.

```
InnerLoop = feedback(X2*G2*C20,1);
CL0 = feedback(G1*InnerLoop*C10,X1);
CL0.InputName = 'r';
CL0.OutputName = 'y';
```

`CL0` is a tunable `genss` model. Specifying names for the input and output channels allows you to identify them when you specify tuning requirements for the system.

Open **Control System Tuner** to tune this model.

```
controlSystemTuner(CL0)
```

You can now select blocks to tune, create tuning goals, and tune the control system.

Control System Architecture in Simulink

If you have Simulink Control Design software, you can model an arbitrary control system architecture in a Simulink model and tune the model in **Control System Tuner**.

See “Open Control System Tuner for Tuning Simulink Model” on page 10-10.

See Also

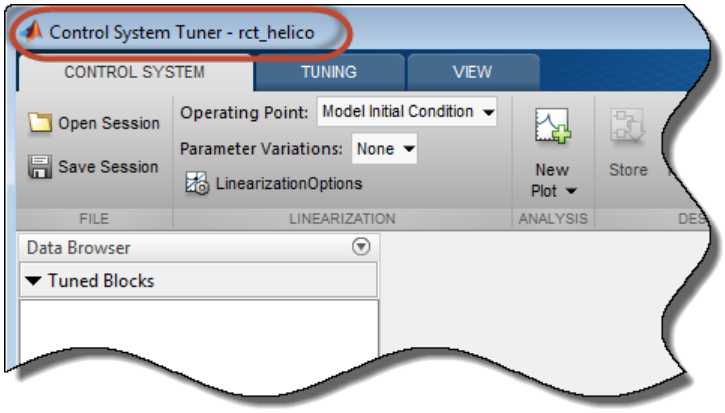
More About

- “Building Tunable Models”
- “Specify Blocks to Tune in Control System Tuner” on page 10-17
- “Specify Goals for Interactive Tuning” on page 10-29

Open Control System Tuner for Tuning Simulink Model

To open **Control System Tuner** for tuning a Simulink model, open the model. In the Simulink model window, in the **Apps** gallery, click **Control System Tuner**.

Each instance of **Control System Tuner** is linked to the Simulink model from which it is opened. The title bar of the **Control System Tuner** window reflects the name of the associated Simulink model.



Command-Line Equivalents

At the MATLAB command line, use the `controlSystemTuner` command to open **Control System Tuner** for tuning a Simulink model. For example, the following command opens **Control System Tuner** for the model `rct_helico.slx`.

```
controlSystemTuner('rct_helico')
```

If `SLT0` is an `slTuner` interface to the Simulink model, the following command opens **Control System Tuner** using the information in the interface, such as blocks to tune and analysis points.

```
controlSystemTuner(SLT0)
```

See Also

Related Examples

- “Specify Operating Points for Tuning in Control System Tuner” on page 10-11
- “Specify Blocks to Tune in Control System Tuner” on page 10-17

More About

- “Automated Tuning Workflow” on page 10-6

Specify Operating Points for Tuning in Control System Tuner

About Operating Points in Control System Tuner

When you use **Control System Tuner** with a Simulink model, the software computes system responses and tunes controller parameters for a linearization of the model. That linearization can depend on model operating conditions.

By default, **Control System Tuner** linearizes at the operating point specified in the model, which comprises the initial state values in the model (the model initial conditions). You can specify one or more alternate operating points for tuning the model. **Control System Tuner** lets you compute two types of alternate operating points:

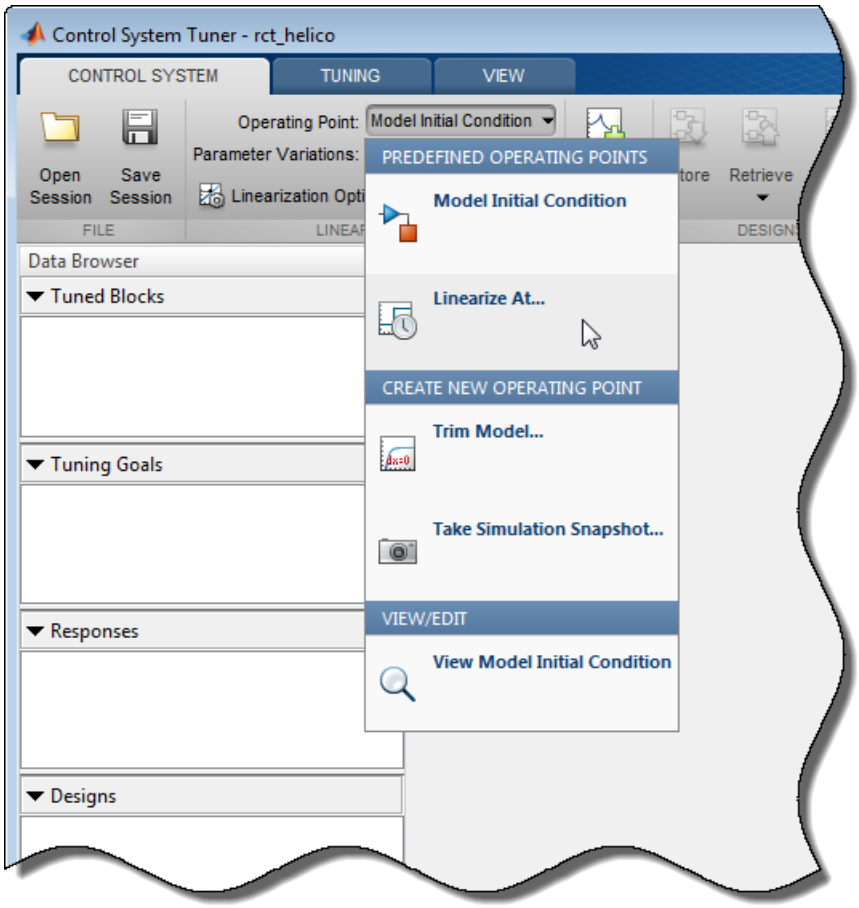
- Simulation snapshot time. **Control System Tuner** simulates the model for the amount of time you specify, and linearizes using the state values at that time. Simulation snapshot linearization is useful, for instance, when you know your model reaches an equilibrium state after a certain simulation time.
- Steady-state operating point. **Control System Tuner** finds a steady-state operating point at which some specified condition is met (trimming). For example, if your model represents an automobile motor, you can compute an operating point at which the motor operates steadily at 2000 rpm.

For more information on finding steady-state operating points, see “About Operating Points” on page 1-2 and “Compute Steady-State Operating Points” on page 1-5.

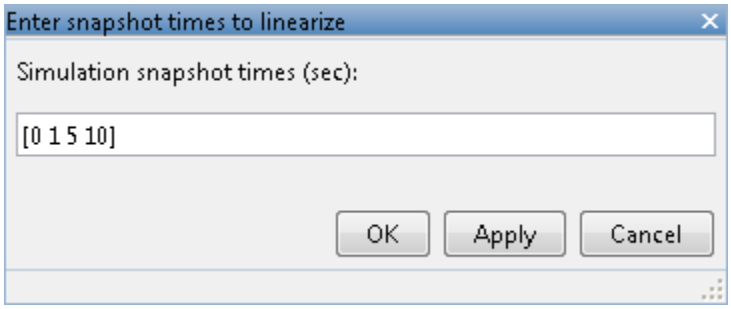
Linearize at Simulation Snapshot Times

This example shows how to compute linearizations at one or more simulation snapshot times.

In the **Control System** tab, in the **Operating Point** menu, select `Linearize At`.



In the **Enter snapshot times to linearize** dialog box, specify one or more simulation snapshot times. Click **OK**.



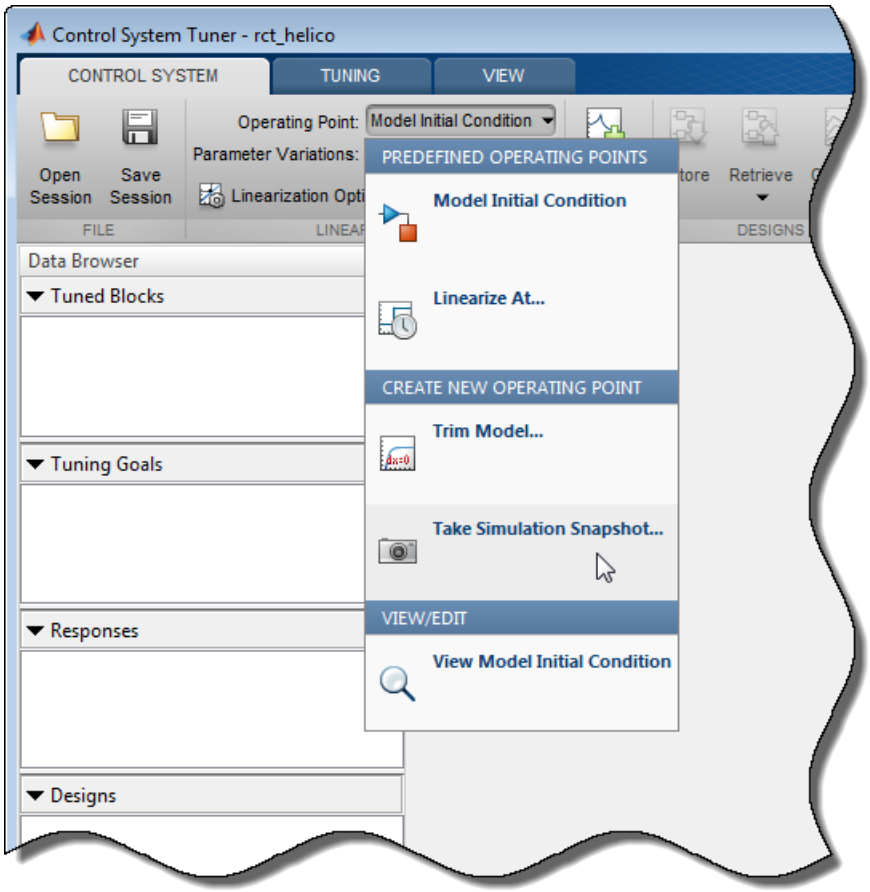
When you are ready to analyze system responses or tune your model, **Control System Tuner** computes linearizations at the specified snapshot times. If you enter multiple snapshot times, **Control System Tuner** computes an array of linearized models, and displays analysis plots that reflect the multiple linearizations in the array. In this case, **Control System Tuner** also takes into

account all linearizations when tuning parameters. This helps to ensure that your tuned controller meets your design requirements at a variety of operating conditions.

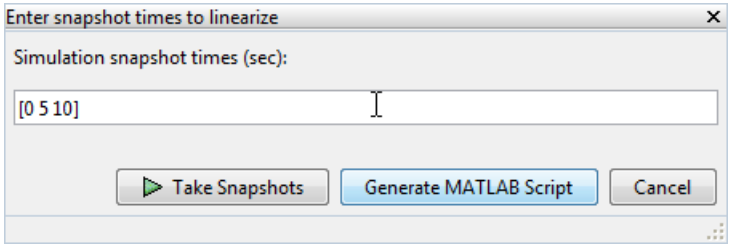
Compute Operating Points at Simulation Snapshot Times

This example shows how to compute operating points at one or more simulation snapshot times. Doing so stores the operating point within **Control System Tuner**. When you later want to analyze or tune the model at a stored operating point, you can select the stored operating point from the **Operating Point** menu.

In the **Control System** tab, in the **Operating Point** menu, select `Take simulation snapshot`.



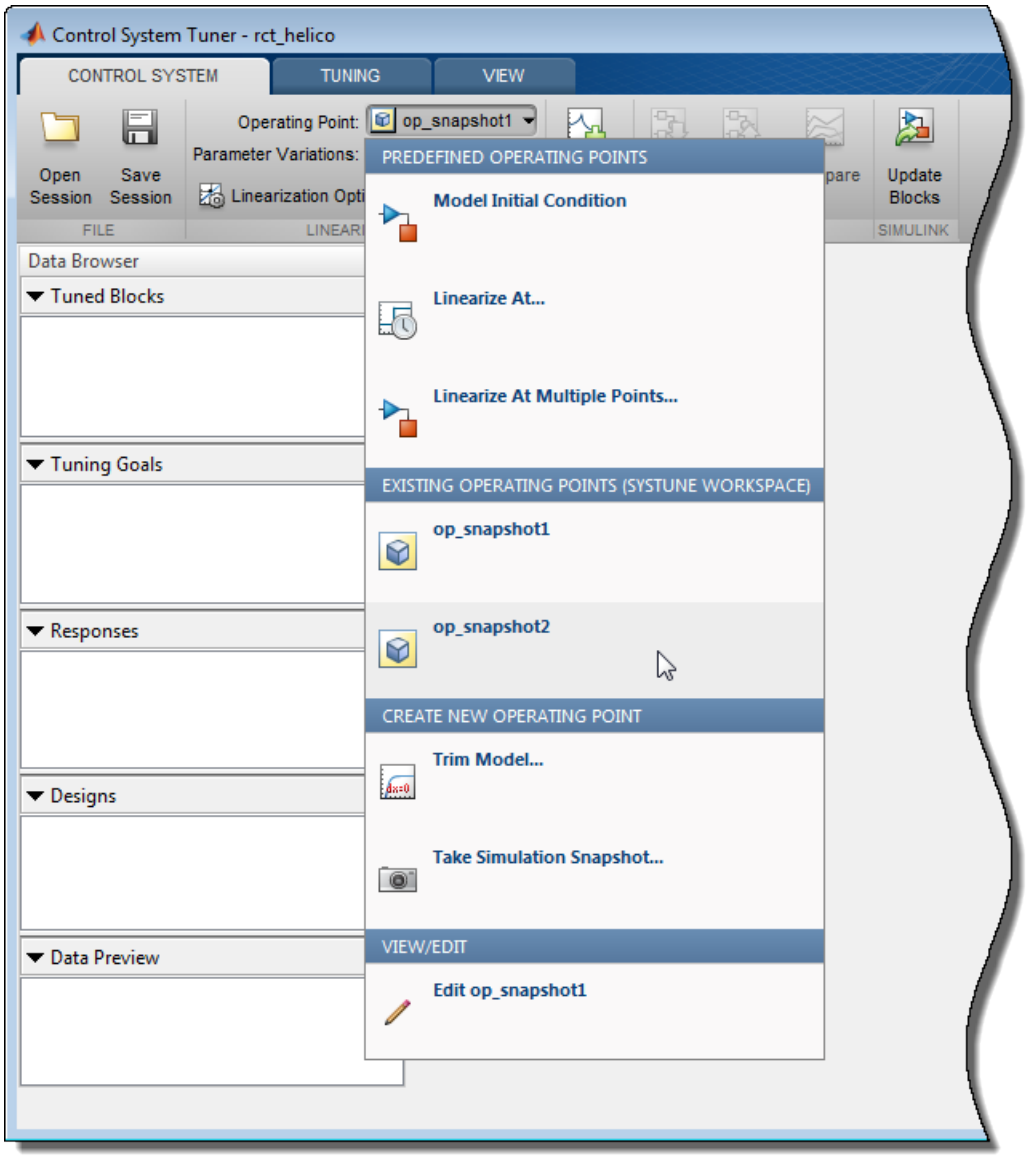
In the **Enter snapshot times to linearize** dialog box, in the **Simulation snapshot times** field, enter one or more simulation snapshot times. Enter multiple snapshot times as a vector.



Click **Take Snapshots**. **Control System Tuner** simulates the model and computes the snapshot operating points.

Compute additional snapshot operating points if desired. Enter additional snapshot times and click **Take Snapshots**. Close the dialog box when you are done.

When you are ready to analyze responses or tune your model, select the operating point at which you want to linearize the model. In the **Control System** tab, in the **Operating Point** menu, stored operating points are available.

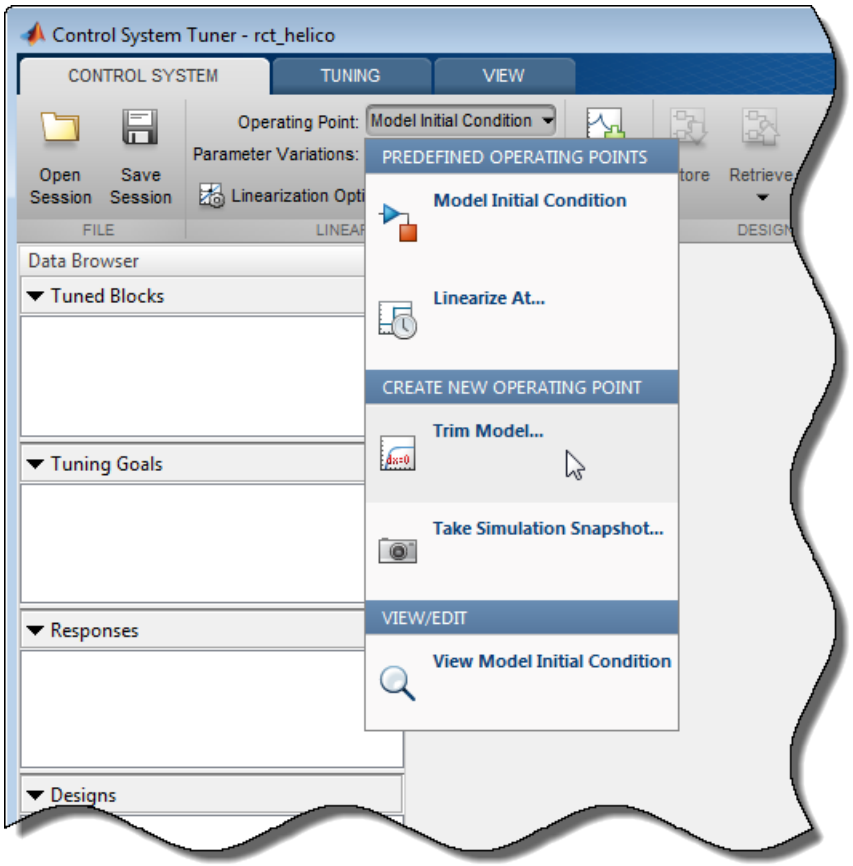


If you entered a vector of snapshot times, all the resulting operating points are stored together in an operating-point vector. You can use this vector to tune a control system at several operating points simultaneously.

Compute Steady-State Operating Points

This example shows how to compute a steady-state operating point with specified conditions. Doing so stores the operating point within **Control System Tuner**. When you later want to analyze or tune the model at a stored operating point, you can select the stored operating point from the **Operating Point** menu.

In the **Control System** tab, in the **Operating Point** menu, select `Trim model`.

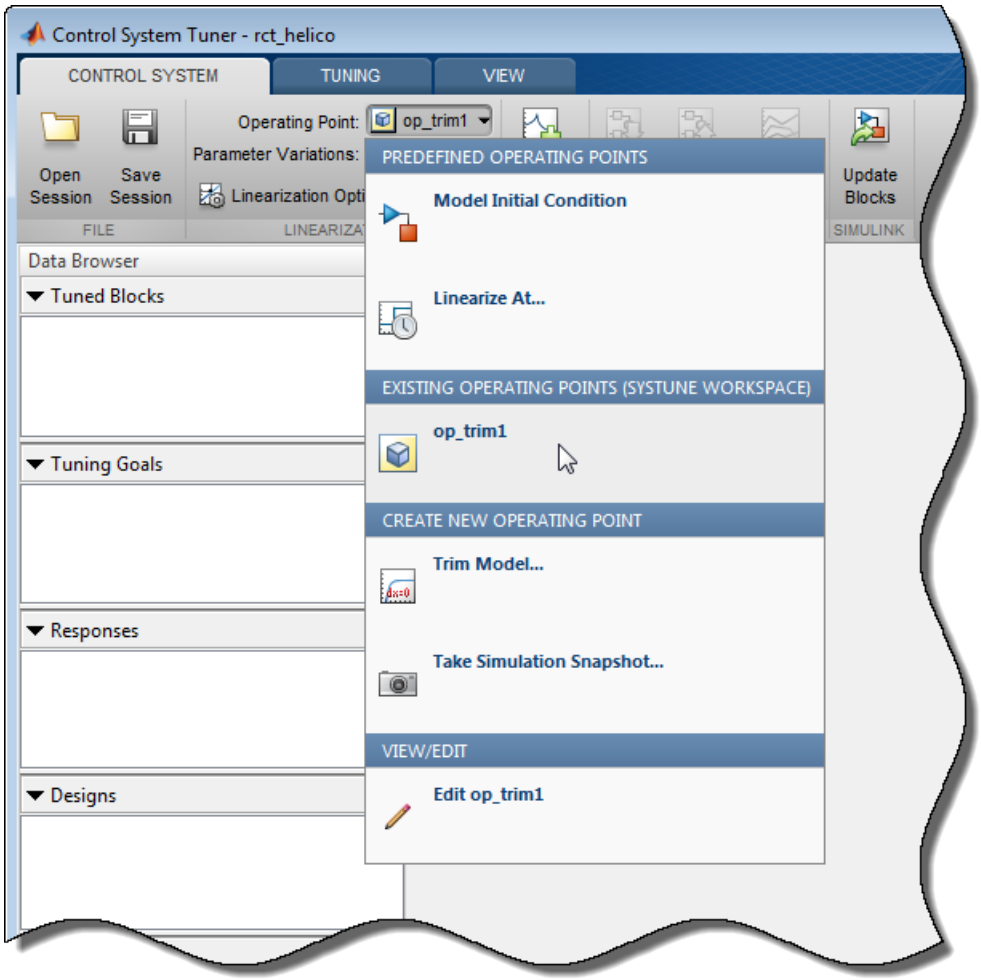


In the **Trim the model** dialog box, enter the specifications for the steady-state state values at which you want to find an operating point.

For an example showing how to use the **Trim the model** dialog box to specify the conditions for a steady-state operating point search, see “Compute Operating Points from Specifications Using Model Linearizer” on page 1-31.

When you have entered your state specifications, click **Start trimming**. **Control System Tuner** finds an operating point that meets the state specifications and stores it.

When you are ready to analyze responses or tune your model, select the operating point at which you want to linearize the model. In the **Control System** tab, in the **Operating Point** menu, stored operating points are available.



See Also

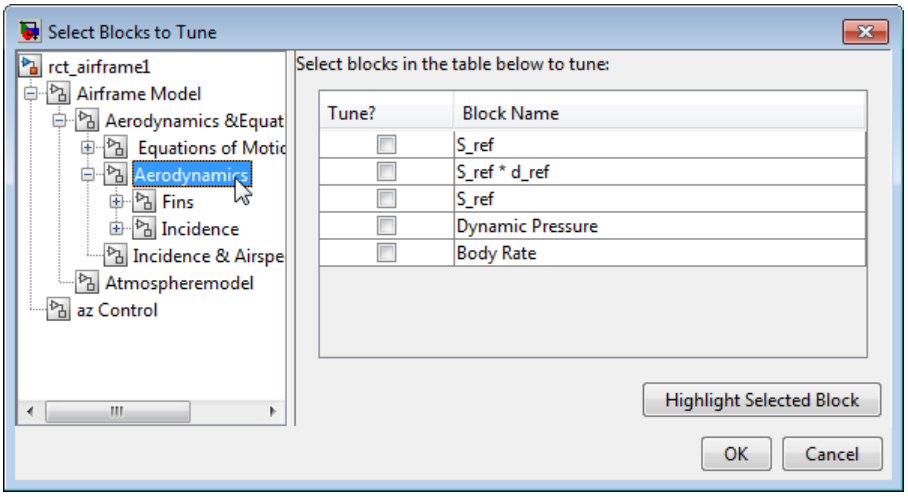
Related Examples

- “Specify Blocks to Tune in Control System Tuner” on page 10-17
- “Robust Tuning Approaches” (Robust Control Toolbox)

Specify Blocks to Tune in Control System Tuner

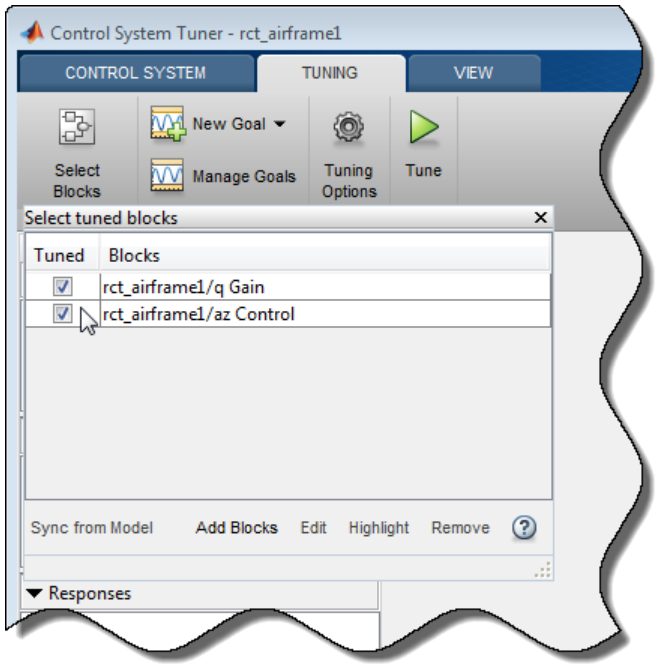
To select which blocks of your Simulink model to tune in **Control System Tuner**:

1. In the **Tuning** tab, click **Select Blocks**. The **Select tuned Blocks** dialog opens.
2. Click **Add Blocks**. **Control System Tuner** analyzes your model to find blocks that can be tuned.
3. In the **Select Blocks to Tune** dialog box, use the nodes in the left panel to navigate through your model structure to the subsystem that contains blocks you want to tune. Check **Tune?** for the blocks you want to tune. The parameters of blocks you do not check remain constant when you tune the model.



Tip To find a block in your model, select the block in the **Block Name** list and click **Highlight Selected Block**.

4. Click **OK**. The **Select tuned blocks** dialog box now reflects the blocks you added.



To import the current value of a block from your model into the current design in **Control System Tuner**, select the block in the **Blocks** list and click **Sync from Model**. Doing so is useful when you have tuned a block in **Control System Tuner**, but wish to restore that block to its original value. To store the current design before restoring a block value, in the **Control System** tab, click **Store**.

See Also

Related Examples

- “View and Change Block Parameterization in Control System Tuner” on page 10-19

More About

- “How Tuned Simulink Blocks Are Parameterized” on page 10-27

View and Change Block Parameterization in Control System Tuner

Control System Tuner parameterizes every block that you designate for tuning.

- When you tune a Simulink model, **Control System Tuner** automatically assigns a default parameterization to tunable blocks in the model. The default parameterization depends on the type of block. For example, a PID Controller block configured for PI structure is parameterized by proportional gain and integral gain as follows:

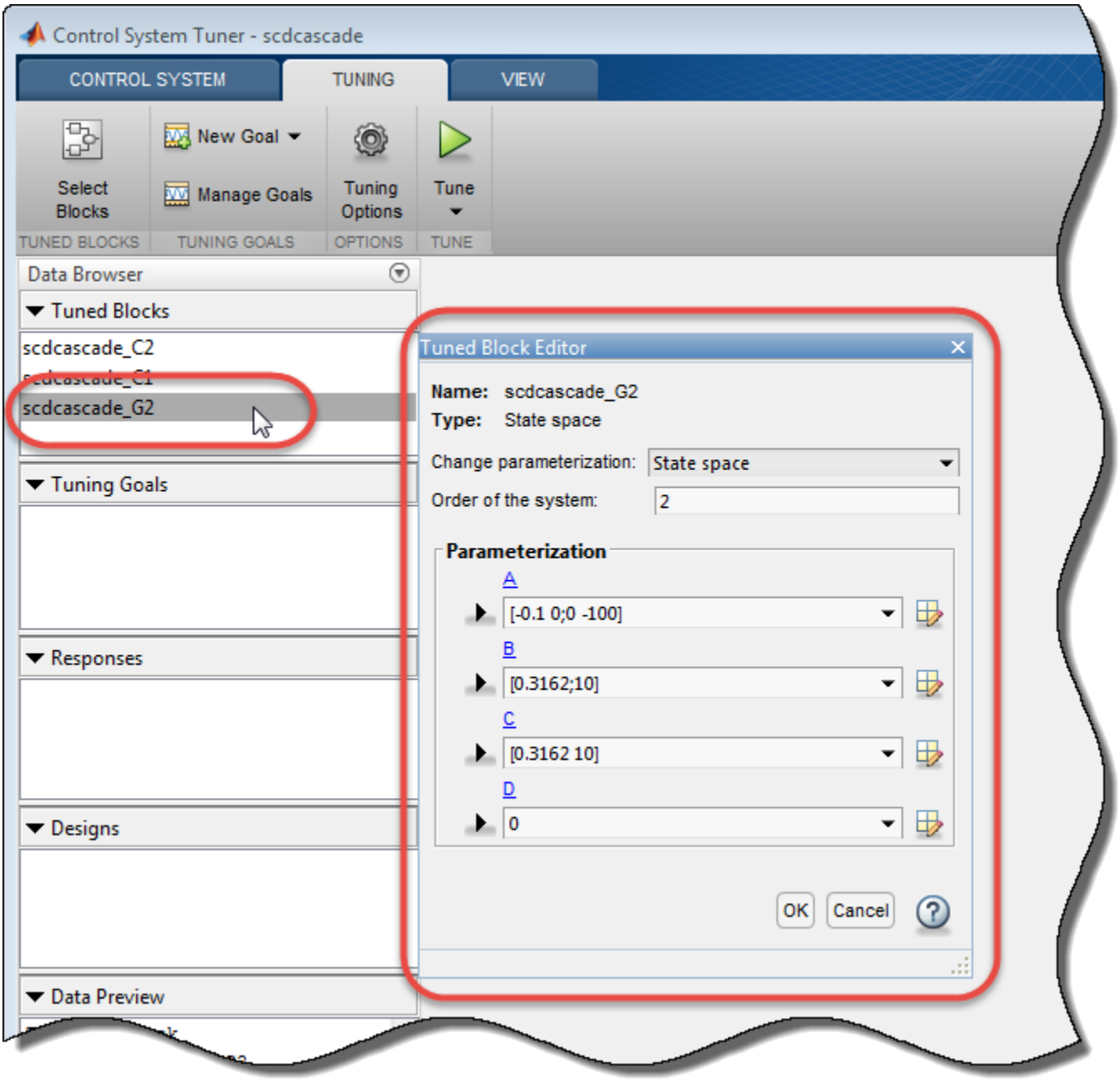
  $$u = K_p + K_i\frac{1}{s}.$$

  K_p and K_i are the tunable parameters whose values are optimized to satisfy your specified tuning goals.
- When you tune a predefined control architecture or a MATLAB (generalized state-space) model, you define the parameterization of each tunable block when you create it at the MATLAB command line. For example, you can use `tunablePID` to create a tunable PID block.

Control System Tuner lets you view and change the parameterization of any block to be tuned. Changing the parameterization can include changing the structure or current parameter values. You can also designate individual block parameters fixed (non-tunable) or limit their tuning range.

View Block Parameterization

To access the parameterization of a block that you have designated as a tuned block, in the **Data Browser**, in the **Tuned Blocks** area, double-click the name of a block. The Tuned Block Editor dialog box opens, displaying the current block parameterization.



The fields of the **Tuned Block Editor** display the type of parameterization, such as `PID`, `State-Space`, or `Gain`. For more specific information about the fields, click .

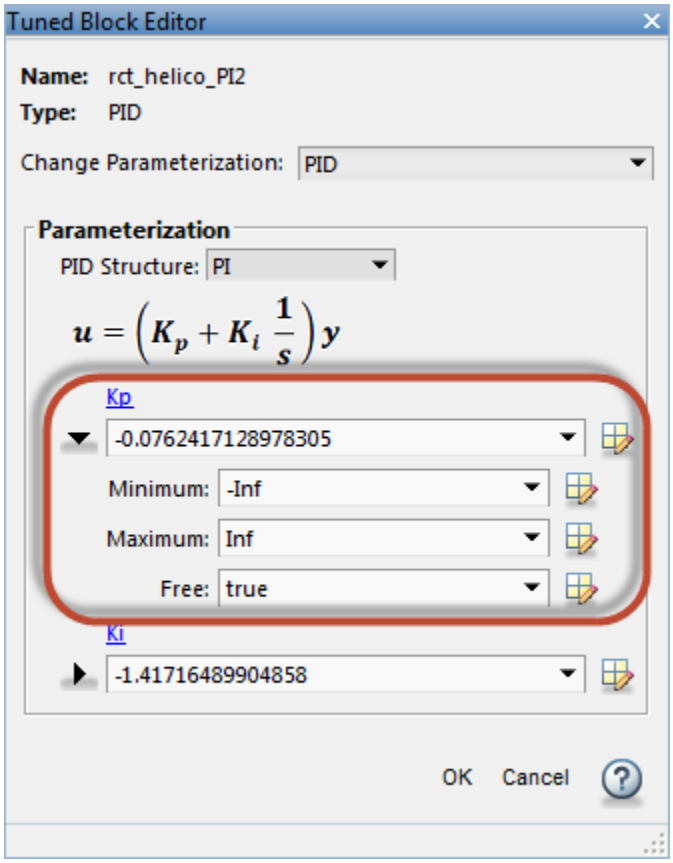
Note To find a tuned block in the Simulink model, right-click the block name in the **Data Browser** and select **Highlight**.

Fix Parameter Values or Limit Tuning Range

You can change the current value of a parameter, fix its current value (make the parameter nontunable), or limit the parameter's tuning range.

To change a current parameter value, type a new value in its text box. Alternatively, click to use a variable editor to change the current value. If you attempt to enter an invalid value, the parameter returns to its previous value.

Click to access and edit additional properties of each parameter.



- **Minimum** — Minimum value that the parameter can take when the control system is tuned.
- **Maximum** — Maximum value that the parameter can take when the control system is tuned.
- **Free** — When the value is `true`, Control System Toolbox tunes the parameter. To fix the value of the parameter, set **Free** to `false`.

For array-valued parameters, you can set these properties independently for each entry in the array. For example, for a vector-valued gain of length 3, enter `[1 10 100]` to set the current value of the three gains to 1, 10, and 100 respectively. Alternatively, click to use a variable editor to specify such values.

For vector or matrix-valued parameters, you can use the **Free** parameter to constrain the structure of the parameter. For example, to restrict a matrix-valued parameter to be a diagonal matrix, set the current values of the off-diagonal elements to 0, and set the corresponding entries in **Free** to `false`.

Custom Parameterization

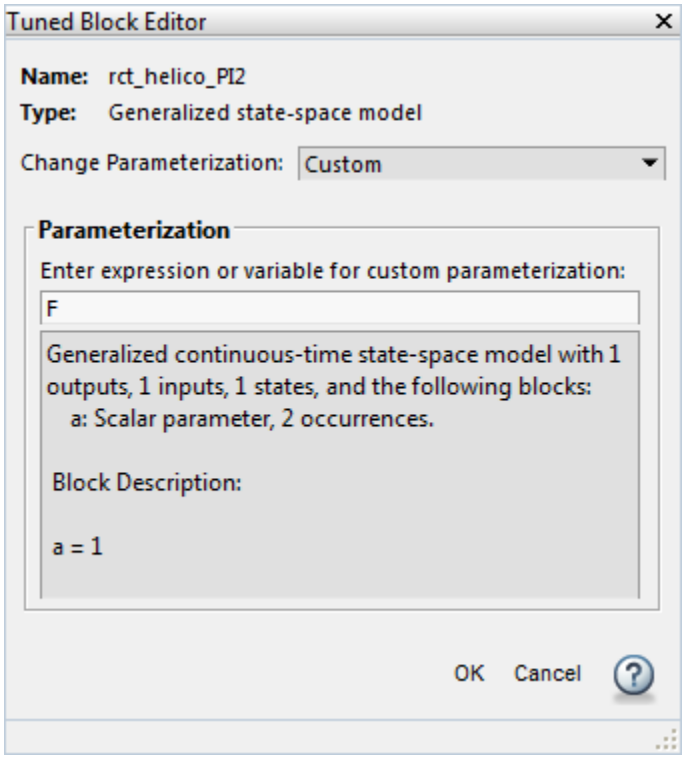
When tuning a control system represented by a Simulink model or by a “Predefined Feedback Architecture” on page 10-7, you can specify a custom parameterization for any tuned block using a generalized state-space (`genss`) model. To do so, create and configure a `genss` model in the MATLAB workspace that has the desired parameterization, initial parameter values, and parameter properties. In the **Change parameterization** dialog box, select `Custom`. In the **Parameterization** area, the variable name of the `genss` model.

For example, suppose you want to specify a tunable low-pass filter, $F = a/(s+a)$, where a is the tunable parameter. First, at the MATLAB command line, create a tunable `genss` model that represents the low-pass filter structure.

```
a = realp('a',1);
F = tf(a,[1 a]);
```

```
F =

  Generalized continuous-time state-space model with 1 outputs, 1 inputs,
  1 states, and the following blocks:
    a: Scalar parameter, 2 occurrences.

Type "ss(F)" to see the current value, "get(F)" to see all properties, and
"F.Blocks" to interact with the blocks.
```

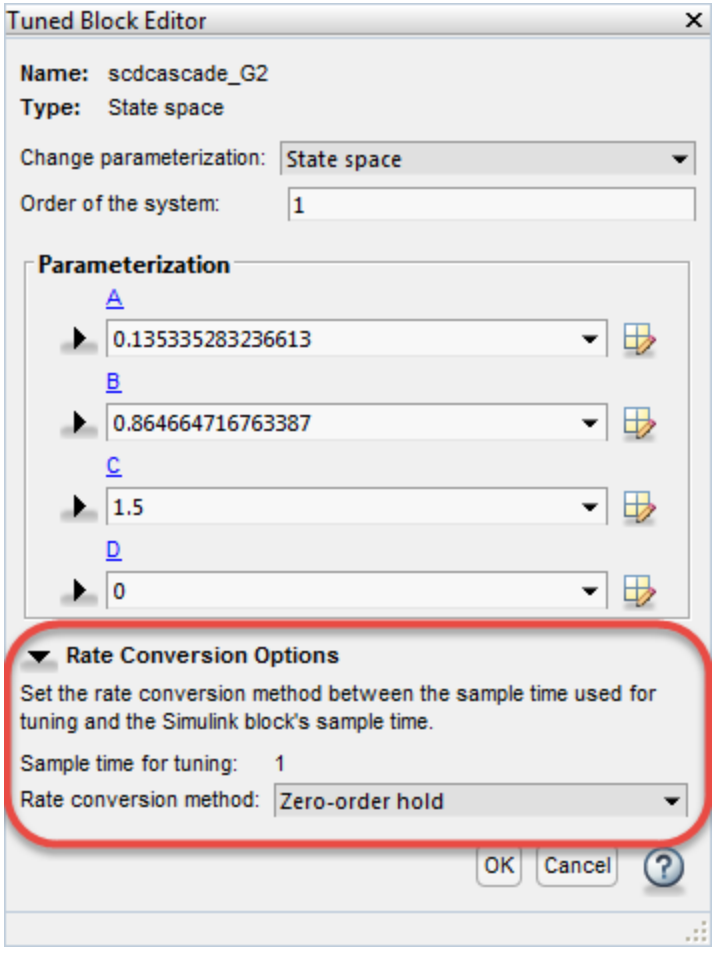
Then, in the Tuned Block Editor, enter `F` in the **Parameterization** area.



When you specify a custom parameterization for a Simulink block, you might not be able to write the tuned block value back to the Simulink model. When writing values to Simulink blocks, **Control System Tuner** skips blocks that cannot represent the tuned value in a straightforward and lossless manner. For example, if you reparameterize a PID Controller Simulink block as a third-order state-space model, **Control System Tuner** will not write the tuned value back to the block.

Block Rate Conversion

When **Control System Tuner** writes tuned parameters back to the Simulink model, each tuned block value is automatically converted from the sample time used for tuning, to the sample time of the Simulink block. When the two sample times differ, the Tuned Block Editor contains additional rate conversion options that specify how this resampling operation is performed for the corresponding block.



By default, **Control System Tuner** performs linearization and tuning in continuous time (sample time = 0). You can specify discrete-time linearization and tuning and change the sample time. To do so, on the **Control System** tab, click **Linearization Options**. **Sample time for tuning** reflects the sample time specified in the **Linearization Options** dialog box.

The remaining rate conversion options depend on the parameterized block.

Rate Conversion for Parameterized PID Blocks

For parameterization of continuous-time PID Controller and PID Controller (2-DOF) blocks, you can independently specify the rate-conversion methods as discretization formulas for the integrator and derivative filter. Each has the following options:

- `Trapezoidal` (default) — Integrator or derivative filter discretized as `(Ts/2)*(z+1)/(z-1)`, where `Ts` is the target sample time.
- `Forward Euler` — `Ts/(z-1)`.
- `Backward Euler` — `Ts*z/(z-1)`.

For more information about PID discretization formulas, see “Discrete-Time Proportional-Integral-Derivative (PID) Controllers”.

For discrete-time PID Controller and PID Controller (2-DOF) blocks, you set the integrator and derivative filter methods in the block dialog box. You cannot change them in the Tuned Block Editor.

Rate Conversion for Other Parameterized Blocks

For blocks other than PID Controller blocks, the following rate-conversion methods are available:

- `Zero-order hold` — Zero-order hold on the inputs. For most dynamic blocks this is the default rate-conversion method.
- `Tustin` — Bilinear (Tustin) approximation.
- `Tustin with prewarping` — Tustin approximation with better matching between the original and rate-converted dynamics at the prewarp frequency. Enter the frequency in the **Prewarping frequency** field.
- `First-order hold` — Linear interpolation of inputs.
- `Matched` (SISO only) — Zero-pole matching equivalents.

For more detailed information about these rate-conversion methods, see “Continuous-Discrete Conversion Methods”.

Blocks with Fixed Rate Conversion Methods

For the following blocks, you cannot set the rate-conversion method in the Tuned Block Editor.

- Discrete-time PID Controller and PID Controller (2-DOF) block. Set the integrator and derivative filter methods in the block dialog box.
- Gain block, because it is static.
- Transfer Fcn Real Zero block. This block can only be tuned at the sample time specified in the block.
- Block that has been discretized using the Model Discretizer. Sample time for this block is specified in the Model Discretizer itself.

See Also

Related Examples

- “Specify Blocks to Tune in Control System Tuner” on page 10-17

More About

- “How Tuned Simulink Blocks Are Parameterized” on page 10-27

Setup for Tuning Control System Modeled in MATLAB

To model your control architecture in MATLAB for tuning in **Control System Tuner**, construct a tunable model of the control system that identifies and parameterizes its tunable elements. You do so by combining numeric LTI models of the fixed elements with parametric models of the tunable elements. The result is a tunable generalized state-space `genss` model.

Building a tunable `genss` model for **Control System Tuner** is the same as building such a model for tuning at the command line. For information about building such models, “Setup for Tuning MATLAB Models”.

When you have a tunable `genss` model of your control system, use the `controlSystemTuner` command to open **Control System Tuner**. For example, if `T0` is the `genss` model, the following command opens **Control System Tuner** for tuning `T0`:

```
controlSystemTuner(T0)
```

See Also

Related Examples

- “Specify Goals for Interactive Tuning” on page 10-29

How Tuned Simulink Blocks Are Parameterized

Blocks With Predefined Parameterization

When you tune a Simulink model, either with **Control System Tuner** or at the command line through an `slTuner` interface, the software automatically assigns a predefined parameterization to certain Simulink blocks. For example, for a PID Controller block set to the PI controller type, the software automatically assigns the parameterization $K_p + K_i/s$, where K_p and K_i are the tunable parameters. For blocks that have a predefined parameterization, you can write tuned values back to the Simulink model for validating the tuned controller.

Blocks that have a predefined parameterization include the following:

Simulink Library	Blocks with Predefined Parameterization
Math Operations	Gain
Continuous	• State-Space • Transfer Fcn • Zero-Pole • PID Controller • PID Controller (2DOF)
Discrete	• Discrete State-Space • Discrete Transfer Fcn • Discrete Zero-Pole • Discrete Filter • Discrete PID Controller • Discrete PID Controller (2DOF)
Lookup Tables	• 1-D Lookup Table • 2-D Lookup Table • n-D Lookup Table
Control System Toolbox	LTI System
Discretizing (Model Discretizer Blocks)	• Discretized State-Space • Discretized Transfer Fcn • Discretized Zero-Pole • Discretized LTI System • Discretized Transfer Fcn (with initial states)
Simulink Extras/Additional Linear	State-Space (with initial outputs)

Scalar Expansion

The following tunable blocks support scalar expansion:

- Discrete Filter
- Gain

- 1-D Lookup Table, 2-D Lookup Table, n-D Lookup Table
- PID Controller, PID Controller (2DOF)

Scalar expansion means that the block parameters can be scalar values even when the input and output signals are vectors. For example, you can use a Gain block to implement `y = k*u` with scalar `k` and vector `u` and `y`. To do so, you set the **Multiplication** mode of the block to `Element-wise(K.*u)`, and set the gain value to the scalar `k`.

When a tunable block uses scalar expansion, its default parameterization uses tunable scalars. For example, in the `y = k*u` Gain block, the software parameterizes the scalar `k` as a tunable real scalar (`realp` of size `[1 1]`). If instead you want to tune different gain values for each channel, replace the scalar gain `k` by a `N-by-1` gain vector in the block dialog, where `N` is the number of channels, the length of the vectors `u` and `y`. The software then parameterizes the gain as a `realp` of size `[N 1]`.

Blocks Without Predefined Parameterization

You can specify blocks for tuning that do not have a predefined parameterization. When you do so, the software assigns a state-space parameterization to such blocks based upon the block linearization. For blocks that do not have a predefined parameterization, the software cannot write tuned values back to the block, because there is no clear mapping between the tuned parameters and the block. To validate a tuned control system that contains such blocks, you can specify a block linearization in your model using the value of the tuned parameterization. (See “Specify Linear System for Block Linearization Using MATLAB Expression” on page 2-123 for more information about specifying block linearization.)

View and Change Block Parameterization

You can view and edit the current parameterization of every block you designate for tuning.

- In **Control System Tuner**, see “View and Change Block Parameterization in Control System Tuner” on page 10-19.
- At the command line, use `getBlockParam` to view the current block parameterization. Use `setBlockParam` to change the block parameterization.

Specify Goals for Interactive Tuning

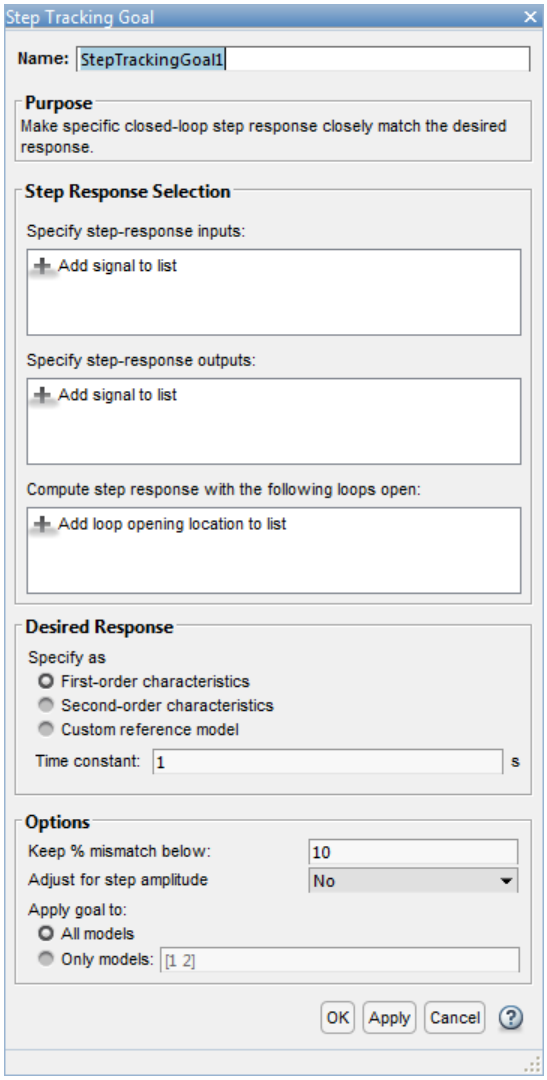
This example shows how to specify your tuning goals for automated tuning in **Control System Tuner**.

Use the **New Goal** menu to create a tuning goal such as a tracking requirement, disturbance rejection specification, or minimum stability margins. Then, when you are ready to tune your control system, use **Manage Goals** to designate which goals to enforce.

This example creates tuning goals for tuning the sample model `rct_helico`.

Choose Tuning Goal Type

In **Control System Tuner**, in the **Tuning** tab, click **New Goal**. Select the type of goal you want to create. A tuning goal dialog box opens in which you can provide the detailed specifications of your goal. For example, select **Tracking of step commands** to make a particular step response of your control system match a desired response.

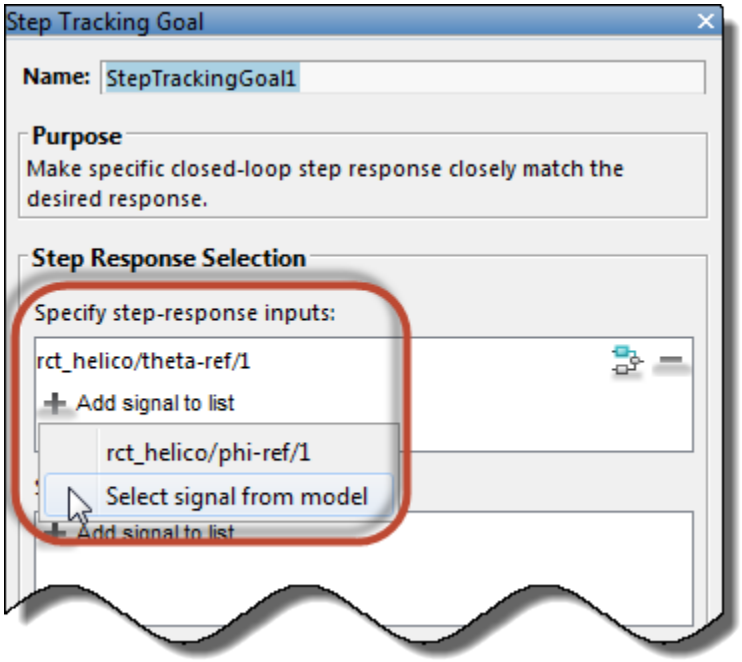


Choose Signal Locations for Evaluating Tuning Goal

Specify the signal locations in your control system at which the tuning goal is evaluated. For example, the step response goal specifies that a step signal applied at a particular input location yields a desired response at a particular output location. Use the **Step Response Selection** section of the dialog box to specify these input and output locations. (Other tuning goal types, such as loop-shape or stability margins, require you to specify only one location for evaluation. The procedure for specifying the location is the same as illustrated here.)

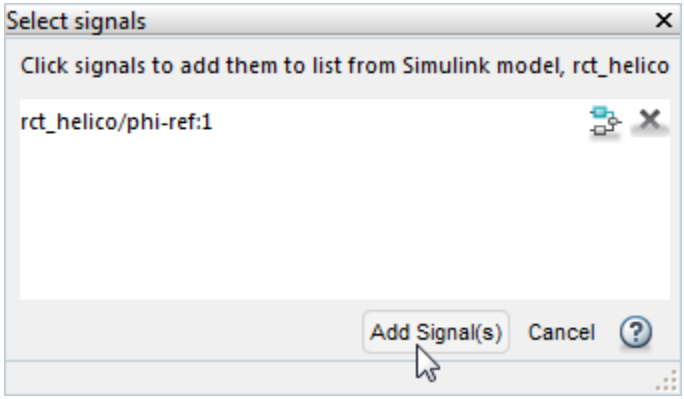
Under **Specify step-response inputs**, click **Add signal to list**. A list of available input locations appears.

If the signal you want to designate as a step-response input is in the list, click the signal to add it to the step-response inputs. If the signal you want to designate does not appear, and you are tuning a Simulink model, click **Select signal from model**.



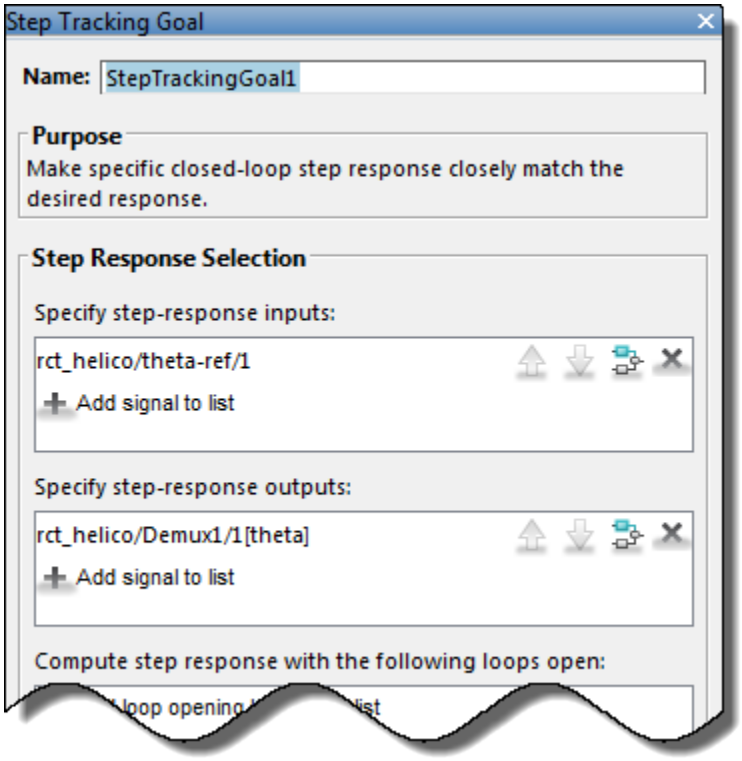
In the **Select signals** dialog box, build a list of the signals you want. To do so, click signals in the Simulink model editor. The signals that you click appear in the **Select signals** dialog box. Click one signal to create a SISO tuning goal, and click multiple signals to create a MIMO tuning goal.

Click **Add signal(s)**. The **Select signals** dialog box closes, returning you to the new tuning-goal specification dialog box.



The signals you selected now appear in the list of step-response inputs in the tuning goal dialog box.

Similarly, specify the locations at which the step response is measured to the step-response outputs list. For example, the following configuration constrains the response to a step input applied at `theta-ref` and measured at `theta` in the Simulink model `rct_helico`.



Tip To highlight any selected signal in the Simulink model, click . To remove a signal from the input or output list, click . When you have selected multiple signals, you can reorder them using and .

Specify Loop Openings

Most tuning goals can be enforced with loops open at one or more locations in the control system. Click **Add loop opening location to list** to specify such locations for the tuning goal.

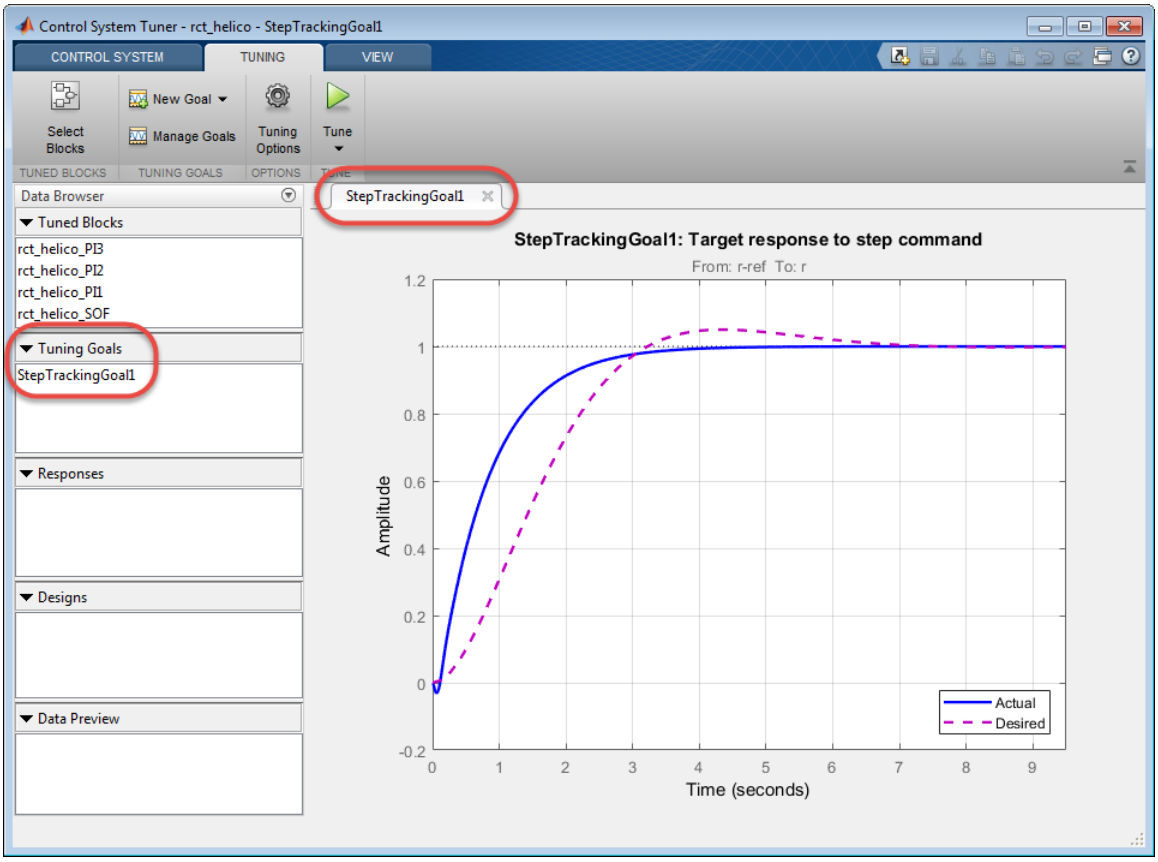
Define Other Specifications of the Tuning Goal

The tuning goal dialog box prompts you to specify other details about the tuning goal. For example, to create a step response requirement, you provide details of the desired step response in the **Desired Response** area of the **Step Response Goal** dialog box. Some tuning goals have additional options in an **Options** section of the dialog box.

For information about the fields for specifying a particular tuning goal, click in the tuning goal dialog box.

Store the Tuning Goal for Tuning

When you have finished specifying the tuning goal, click **OK** in the tuning goal dialog box. The new tuning goal appears in the **Tuning Goals** section of the Data Browser. A new figure opens displaying a graphical representation of the tuning goal. When you tune your control system, you can refer to this figure to evaluate graphically how closely the tuned system satisfies the tuning goal.



Tip To edit the specifications of the tuning goal, double-click the tuning goal in the Data Browser.

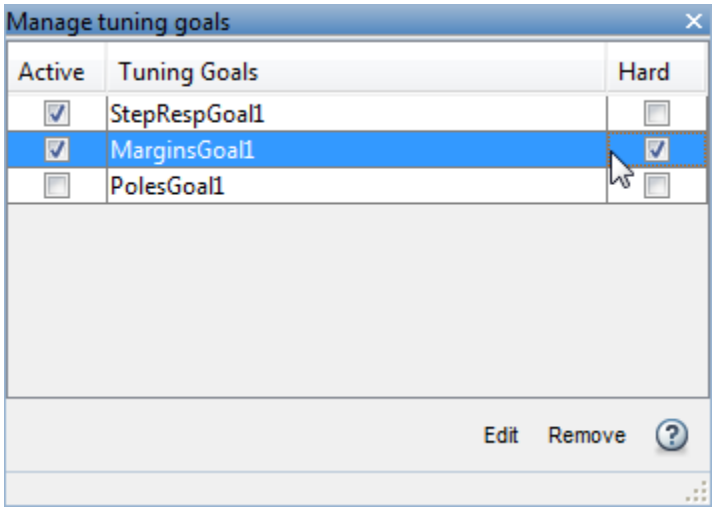
Activate the Tuning Goal for Tuning

When you have saved your tuning goal, click **New Goal** to create additional tuning goals.

When you are ready to tune your control system, click **Manage Goals** to select which tuning goals are active for tuning. In the **Manage Tuning Goals** dialog box, **Active** is checked by default for any new goals. Clear **Active** for any tuning goal that you do not want enforced.

You can also designate one or more tuning goals as **Hard** goals. **Control System Tuner** attempts to satisfy hard requirements, and comes as close as possible to satisfying remaining (soft) requirements subject to the hard constraints. By default, new goals are designated soft goals. Check **Hard** for any goal to designate it a hard goal.

For example, if you tune with the following configuration, **Control System Tuner** optimizes `StepRespGoal1`, subject to `MarginsGoal1`. The tuning goal `PolesGoal1` is ignored.



Deactivating tuning goals or designating some goals as soft requirements can be useful when investigating the tradeoffs between different tuning requirements. For example, if you do not obtain satisfactory performance with all your tuning goals active and hard, you might try another design in which less crucial goals are designated as soft or deactivated entirely.

See Also

Related Examples

- “Manage Tuning Goals” on page 10-135
- “Quick Loop Tuning of Feedback Loops in Control System Tuner” on page 10-34
- “Create Response Plots in Control System Tuner” on page 10-149

Quick Loop Tuning of Feedback Loops in Control System Tuner

This example shows how to tune a Simulink model of a control system to meet a specified bandwidth and specified stability margins in **Control System Tuner**, without explicitly creating tuning goals that capture these requirements. You can use a similar approach for quick loop tuning of control systems modeled in MATLAB.

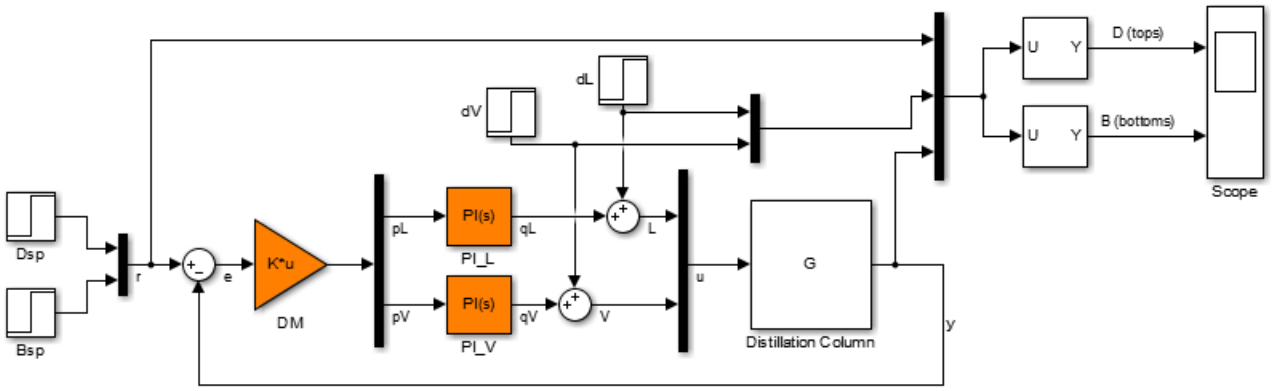
This example demonstrates how the **Quick Loop Tuning** option of **Control System Tuner** generates tuning goals from your crossover frequency and gain and phase margin specifications. This option lets you quickly set up SISO or MIMO feedback loops for tuning using a loop-shaping approach. The example also shows how to add further tuning requirements to the control system after using the **Quick Loop Tuning** option.

Quick Loop Tuning is the **Control System Tuner** equivalent of the `looptune` command.

Set up the Model for Tuning

Open the Simulink model.

```
open_system('rct_distillation')
```



This model represents a distillation column, captured in the two-input, two-output plant `G`. The tunable elements are the decoupling gain matrix `DM`, and the two PI controllers, `PI_L` and `PI_V`. (For more information about this model, see “Decoupling Controller for a Distillation Column”.)

Suppose your goal is to tune the MIMO feedback loop between `r` and `y` to a bandwidth between 0.1 and 0.5 rad/s. Suppose you also require a gain margin of 7 dB and a phase margin of 45 degrees. You can use the **Quick Loop Tuning** option to quickly configure **Control System Tuner** for these goals.

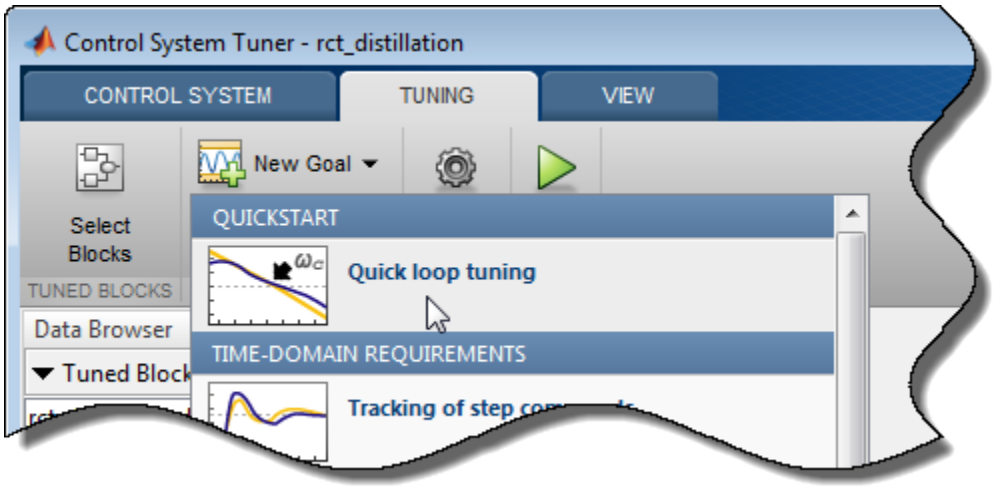
Open **Control System Tuner**. In the Simulink model window, in the **Apps** gallery, click **Control System Tuner**.

Designate the blocks you want to tune. In the **Tuning** tab of **Control System Tuner**, click **Select Blocks**. In the **Select tuned blocks** dialog box, click **Add blocks**. Then, select `DM`, `PI_L`, and `PI_V` for tuning. (For more information about selecting tuned blocks, see “Specify Blocks to Tune in Control System Tuner” on page 10-17.)

The model is now ready to tune to the target bandwidth and stability margins.

Specify the Goals for Quick Loop Tuning

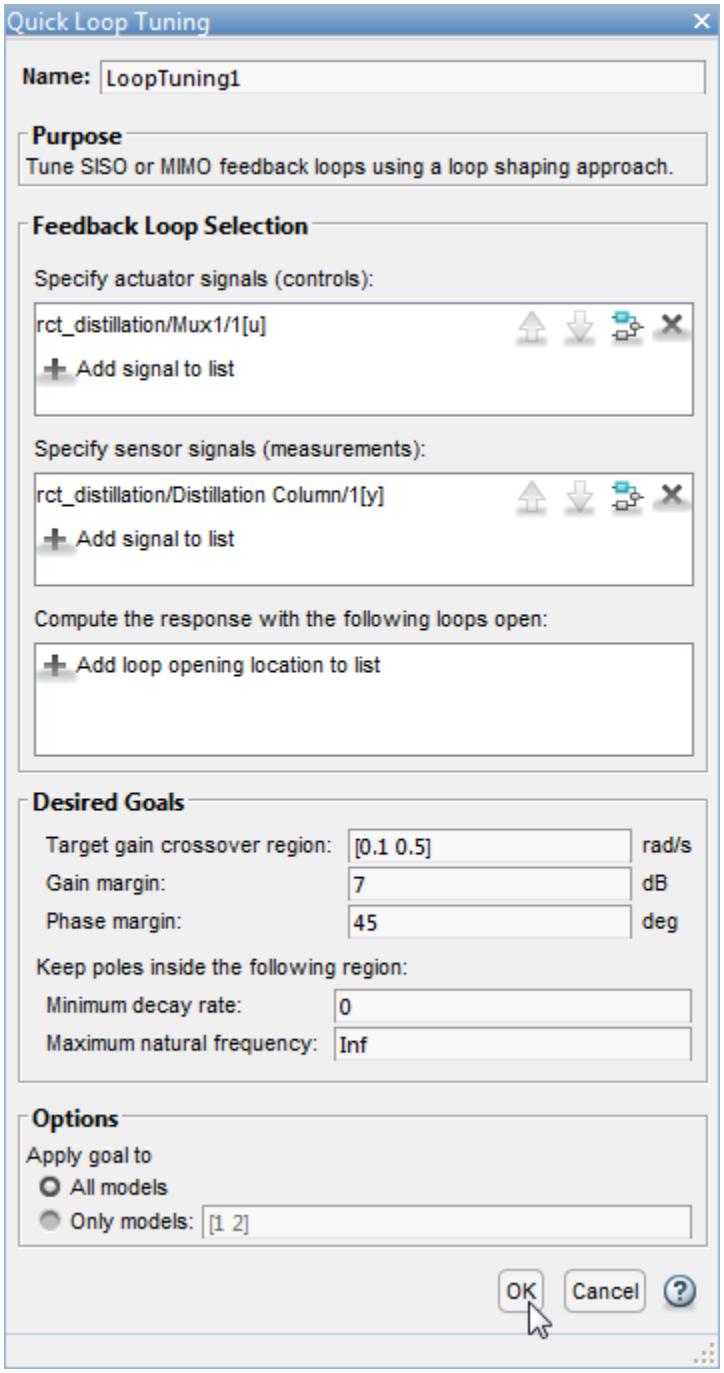
In the **Tuning** tab, select **New Goal** > **Quick Loop Tuning**.



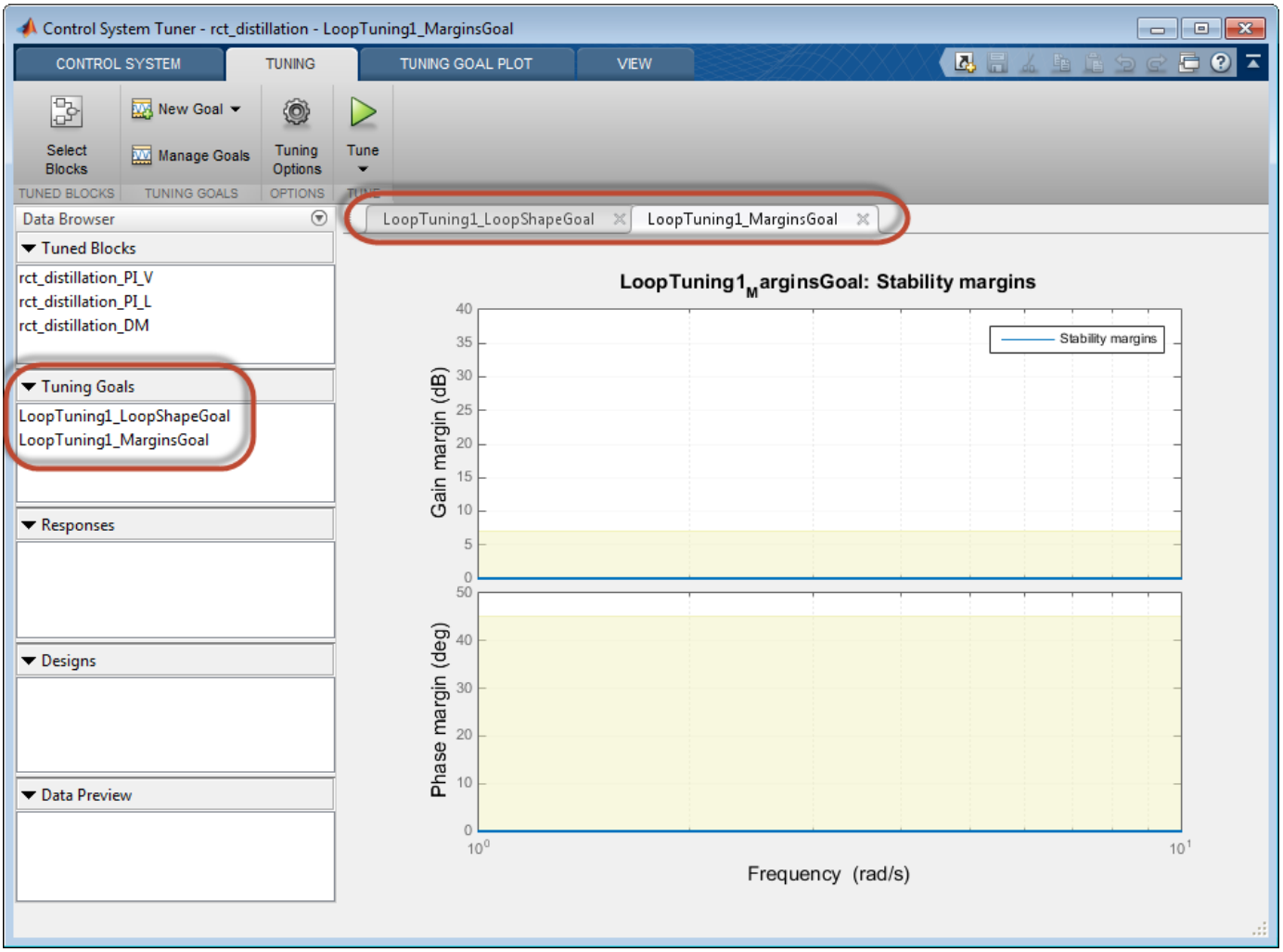
For Quick Loop Tuning, you need to identify the actuator signals and sensor signals that separate the plant portion of the control system from the controller, which for the purpose of Quick Loop Tuning is the rest of the control system. The actuator signals are the controller outputs that drive the plant, or the plant inputs. The sensor signals are the measurements of plant output that feed back into the controller. In this control system, the actuator signals are represented by the vector signal `u`, and the sensor signals by the vector signal `y`.

In the **Quick Loop Tuning** dialog box, under **Specify actuator signals (controls)**, add the actuator signal, `u`. Similarly, under **Specify sensor signals (measurements)**, add the sensor signal, `y` (For more information about specifying signals for tuning, see “Specify Goals for Interactive Tuning” on page 10-29.)

Under **Desired Goals**, in the **Target gain crossover region** field, enter the target bandwidth range, `[0.1 0.5]`. Enter the desired gain margin and phase margin in the corresponding fields.



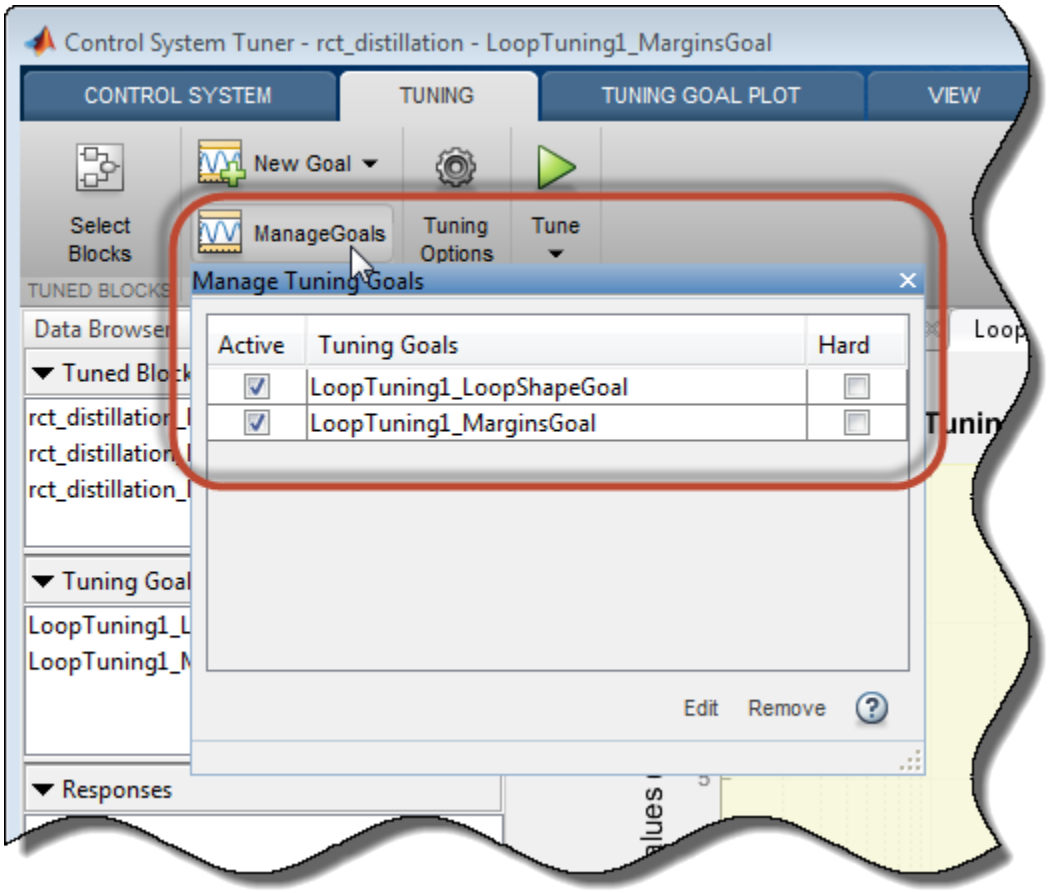
Click **OK**. **Control System Tuner** automatically generates tuning goals that capture the desired goals you entered in the dialog box.



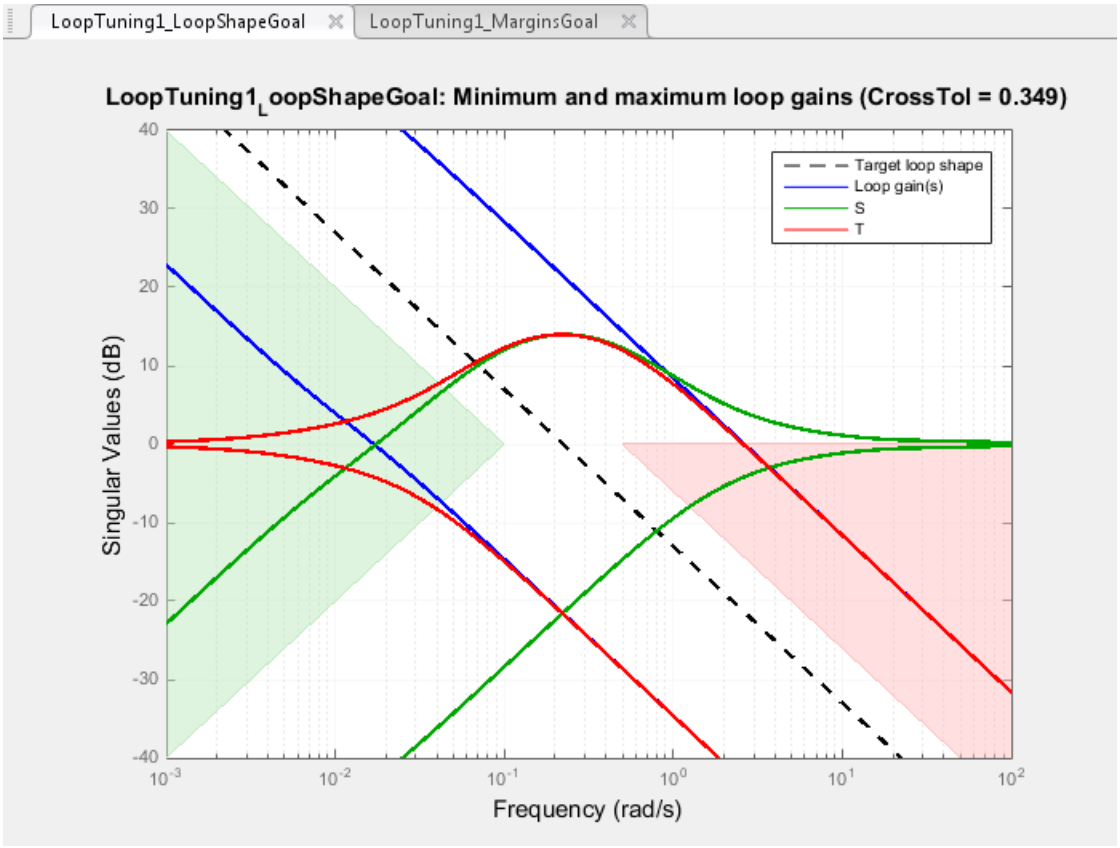
Examine the Automatically-Created Tuning Goals

In this example, **Control System Tuner** creates a Loop Shape Goal and a Margins Goal. If you had changed the pole-location settings in the **Quick Loop Tuning** dialog box, a Poles goal would also have been created.

Click **Manage Goals** to examine the automatically-created goals. By default, the goals are active and designated as soft tuning goals.



You can double-click the tuning goals to examine their parameters, which are automatically computed and populated. You can also examine the graphical representations of the tuning goals. In the **Tuning** tab, examine the **LoopTuning1_LoopShapeGoal** plot.



The target crossover range is expressed as a Loop Shape goal with an integrator open-loop gain profile. The shaded areas of the graph show that the permitted crossover range is `[0.1 0.5]` rad/s, as you specified in the **Quick Loop Tuning** dialog box.

Similarly, your margin requirements are captured in the **LoopTuning1_MarginsGoal** plot.

Tune the Model

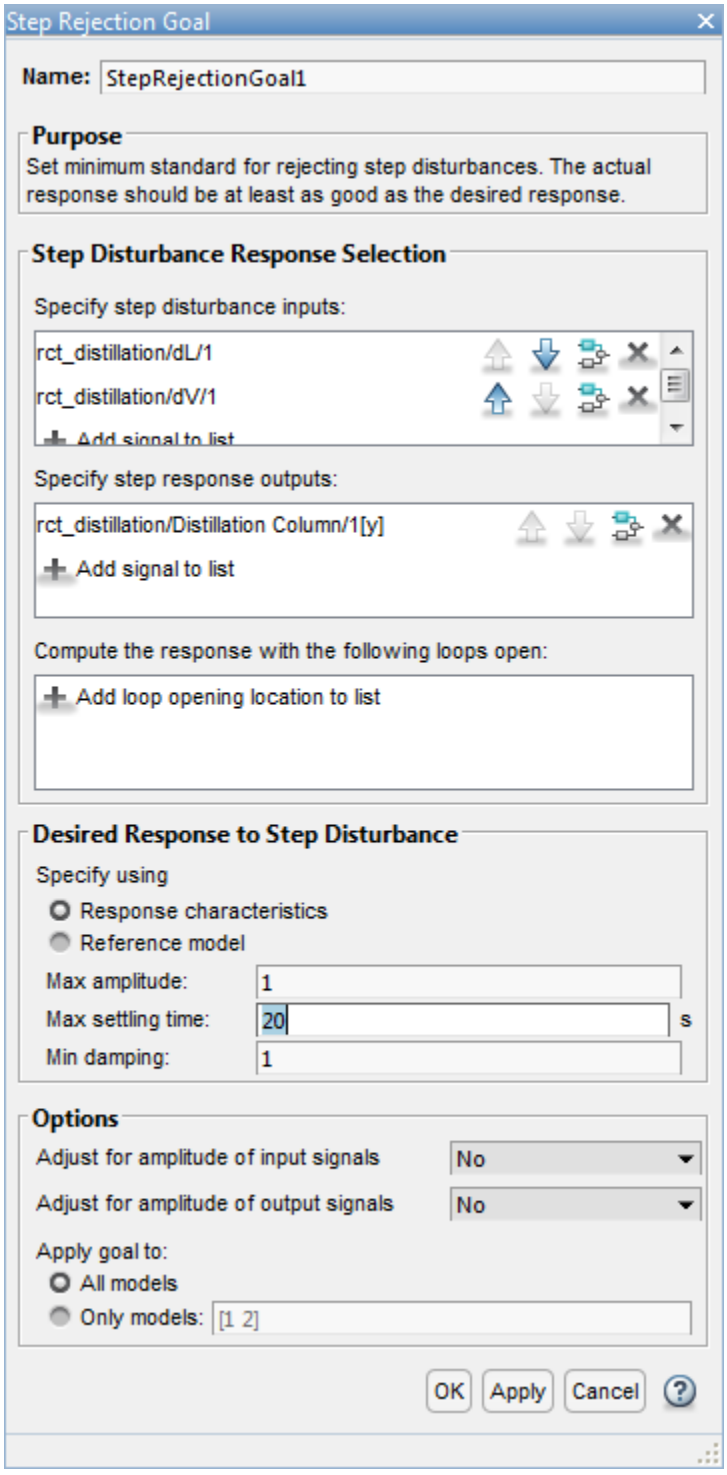
Click **Tune** to tune the model to meet the automatically-created tuning goals. In the tuning goal plots, you can see that the requirements are satisfied.

To create additional plots for examining other system responses, see “Create Response Plots in Control System Tuner” on page 10-149.

Change Design Requirements

If you want to change your design requirements after using Quick Loop Tuning, you can edit the automatically-created tuning goals and tune the model again. You can also create additional tuning goals.

For example, add a requirement that limits the response to a disturbance applied at the plant inputs. Limit the response to a step command at `dL` and `dV` at the outputs, `y`, to be well damped, to settle in less than 20 seconds, and not exceed 4 in amplitude. Select **New Goal > Rejection of step disturbances** and enter appropriate values in the Step Rejection Goal dialog box. (For more information about creating tuning goals, see “Specify Goals for Interactive Tuning” on page 10-29.)



You can now retune the model to meet all these tuning goals.

See Also

`looptune (for slTuner)`

Related Examples

- “Specify Operating Points for Tuning in Control System Tuner” on page 10-11
- “Manage Tuning Goals” on page 10-135
- “Setup for Tuning Control System Modeled in MATLAB” on page 10-26
- “Stability Margins in Control System Tuning”

Quick Loop Tuning

Purpose

Tune SISO or MIMO feedback loops using a loop-shaping approach in **Control System Tuner**.

Description

Quick Loop Tuning lets you tune your system to meet open-loop gain crossover and stability margin requirements without explicitly creating tuning goals that capture these requirements. You specify the feedback loop whose open-loop gain you want to shape by designating the actuator signals (controls) and sensor signals (measurements) that form the loop. Actuator signals are the signals that drive the plant. The sensor signals are the plant outputs that feed back into the controller.

You enter the target loop bandwidth and desired gain and phase margins. You can also specify constraints on pole locations of the tuned system, to eliminate fast dynamics. **Control System Tuner** automatically creates Tuning Goals that capture your specifications and ensure integral action at frequencies below the target loop bandwidth.

Creation

In the **Tuning** tab of **Control System Tuner**, select **New Goal > Quick Loop Tuning** to specify loop-shaping requirements.

Command-Line Equivalent

When tuning control systems at the command line, use `looptune (for slTuner)` or `looptune` for tuning feedback loops using a loop-shaping approach.

Feedback Loop Selection

Use this section of the dialog box to specify input, output, and loop-opening locations for evaluating the tuning goal.

- **Specify actuator signals (controls)**

  Designate one or more signals in your model as actuator signals. These are the input signals that drive the plant. To tune a SISO feedback loop, select a single-valued input signal. To tune MIMO loop, select multiple signals or a vector-valued signal.

- **Specify sensor signals (measurements)**

  Designate one or more signals in your model as sensor signals. These are the plant outputs that provide feedback into the controller. To tune a SISO feedback loop, select a single-valued input signal. To tune MIMO loop, select multiple signals or a vector-valued signal.

- **Compute the response with the following loops open**

  Designate additional locations at which to open feedback loops for the purpose of tuning the loop defined by the control and measurement signals.

  Quick Loop Tuning tunes the open-loop response of the loop defined by the control and measurement signals. If you want your specifications for that loop to apply with other feedback

loops in the system opened, specify loop-opening locations in this section of the dialog box. For example, if you are tuning a cascaded-loop control system with an inner loop and an outer loop, you might want to tune the inner loop with the outer loop open.

Tip To highlight any selected signal in the Simulink model, click . To remove a signal from the input or output list, click . When you have selected multiple signals, you can reorder them using and . For more information on how to specify signal locations for a tuning goal, see “Specify Goals for Interactive Tuning” on page 10-29.

Desired Goals

Use this section of the dialog box to specify desired characteristics of the tuned system. **Control System Tuner** converts these into Loop Shape, Margin, and Poles goals.

- **Target gain crossover region**

  Specify a frequency range in which the open-loop gain should cross 0 dB. Specify the frequency range as a row vector of the form `[min,max]`, expressed in frequency units of your model. Alternatively, if you specify a single target frequency, `wc`, the target range is taken as `[wc/10^0.1,wc*10^0.1]`, or `wc` ± 0.1 decade.

- **Gain margin (db)**

  Specify the desired gain margin in decibels. For MIMO feedback loops, this requirement guarantees stability for gain variations across all feedback channels. The gain can change in all feedback channels simultaneously, and by a different amount in each channel. For information about disk margins, see “Stability Analysis Using Disk Margins” (Robust Control Toolbox).

- **Phase margin (degrees)**

  Specify the desired phase margin in degrees. For MIMO feedback loops, this requirement guarantees stability for phase variations across all feedback channels. The phase can change in all feedback channels simultaneously, and by a different amount in each channel. For information about disk margins, see “Stability Analysis Using Disk Margins” (Robust Control Toolbox).

- **Keep poles inside the following region**

  Specify minimum decay rate and maximum natural frequency for the closed-loop poles of the tuned system. While the other Quick Loop Tuning options specify characteristics of the open-loop response, these specifications apply to the closed-loop dynamics.

  The minimum decay rate you enter constrains the closed-loop pole locations to:

  - `Re(s) < -mindecay`, for continuous-time systems.
  - `log(|z|) < -mindecay*Ts`, for discrete-time systems with sample time `Ts`.

  The maximum frequency you enter constrains the closed-loop poles to satisfy `|s| < maxfreq` for continuous time, or `|log(z)| < maxfreq*Ts` for discrete-time systems with sample time `Ts`. This constraint prevents fast dynamics in the closed-loop system.

Options

Use this section of the dialog box to specify additional characteristics.

- **Apply goal to**

  Use this option when tuning multiple models at once, such as an array of models obtained by linearizing a Simulink model at different operating points or block-parameter values. By default, active tuning goals are enforced for all models. To enforce a tuning requirement for a subset of models in an array, select **Only Models**. Then, enter the array indices of the models for which the goal is enforced. For example, suppose you want to apply the tuning goal to the second, third, and fourth models in a model array. To restrict enforcement of the requirement, enter `2:4` in the **Only Models** text box.

  For more information about tuning for multiple models, see “Robust Tuning Approaches” (Robust Control Toolbox).

Algorithms

Control System Tuner uses `looptuneSetup (for slTuner)` or `looptuneSetup` to convert Quick Loop Tuning specifications into tuning goals.

See Also

Related Examples

- “Quick Loop Tuning of Feedback Loops in Control System Tuner” on page 10-34
- “Specify Goals for Interactive Tuning” on page 10-29
- “Visualize Tuning Goals” on page 10-143
- “Manage Tuning Goals” on page 10-135
- “Stability Margins in Control System Tuning”

Step Tracking Goal

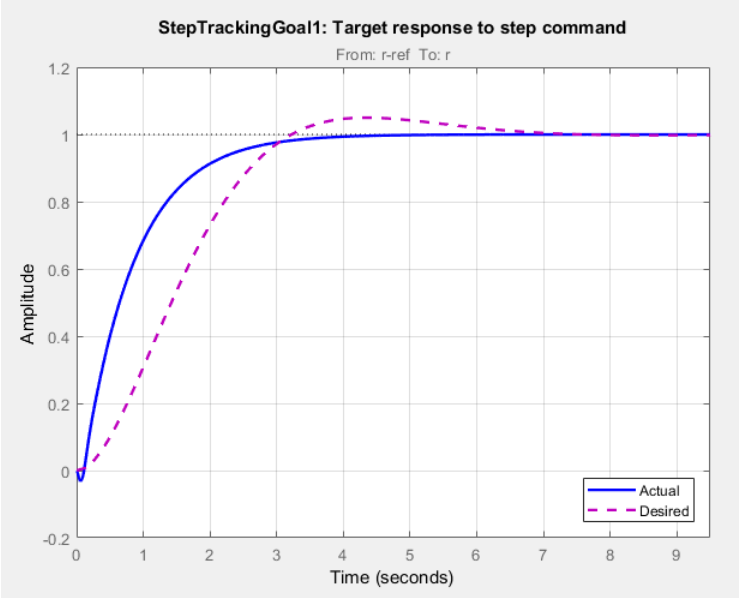
Purpose

Make the step response from specified inputs to specified outputs closely match a target response, when using **Control System Tuner**.

Description

Step Tracking Goal constrains the step response between the specified signal locations to match the step response of a stable reference system. The constraint is satisfied when the relative difference between the tuned and target responses falls within the tolerance you specify. You can use this goal to constrain a SISO or MIMO response of your control system.

You can specify the reference system for the target step response in terms of first-order system characteristics (time constant) or second-order system characteristics (natural frequency and percent overshoot). Alternatively, you can specify a custom reference system as a numeric LTI model.



Creation

In the **Tuning** tab of **Control System Tuner**, select **New Goal > Tracking of step commands** to create a Step Tracking Goal.

Command-Line Equivalent

When tuning control systems at the command line, use `TuningGoal.StepTracking` to specify a step response goal.

Step Response Selection

Use this section of the dialog box to specify input, output, and loop-opening locations for evaluating the tuning goal.

- **Specify step-response inputs**

  Select one or more signal locations in your model at which to apply the step input. To constrain a SISO response, select a single-valued input signal. For example, to constrain the step response from a location named `'u'` to a location named `'y'`, click **Add signal to list** and select `'u'`. To constrain a MIMO response, select multiple signals or a vector-valued signal.

- **Specify step-response outputs**

  Select one or more signal locations in your model at which to measure the response to the step input. To constrain a SISO response, select a single-valued output signal. For example, to constrain the step response from a location named `'u'` to a location named `'y'`, click **Add signal to list** and select `'y'`. To constrain a MIMO response, select multiple signals or a vector-valued signal. For MIMO systems, the number of outputs must equal the number of inputs.

- **Compute step response with the following loops open**

  Select one or more signal locations in your model at which to open a feedback loop for the purpose of evaluating this tuning goal. The tuning goal is evaluated against the open-loop configuration created by opening feedback loops at the locations you identify. For example, to evaluate the tuning goal with an opening at a location named `'x'`, click **Add signal to list** and select `'x'`.

Tip To highlight any selected signal in the Simulink model, click . To remove a signal from the input or output list, click . When you have selected multiple signals, you can reorder them using and . For more information on how to specify signal locations for a tuning goal, see “Specify Goals for Interactive Tuning” on page 10-29.

Desired Response

Use this section of the dialog box to specify the shape of the desired step response.

- **First-order characteristics**

  Specify the desired step response (the reference model H_{ref}) as a first-order response with time constant τ:

  $$H_{ref} = \frac{1/\tau}{s + 1/\tau}.$$

  Enter the desired value for τ in the **Time Constant** text box. Specify τ in the time units of your model.

- **Second-order characteristics**

  Specify the desired step response as a second-order response with time constant τ, and natural frequency $1/\tau$.

  Enter the desired value for τ in the **Time Constant** text box. Specify τ in the time units of your model.

  Enter the target overshoot percentage in the **Overshoot** text box.

The second-order reference system has the form:

$$H_{ref} = \frac{(1/\tau)^2}{s^2 + 2(\zeta/\tau)s + (1/\tau)^2}.$$

The damping constant ζ is related to the overshoot percentage by ζ = `cos(atan2(pi,-log(overshoot/100)))`.

- **Custom reference model**

  Specify the reference system for the desired step response as a dynamic system model, such as a `tf`, `zpk`, or `ss` model.

  Enter the name of the reference model in the MATLAB workspace in the **LTI model to match** text field. Alternatively, enter a command to create a suitable reference model, such as `tf(1,[1 1.414 1])`.

  The reference model must be stable and must have DC gain of 1 (zero steady-state error). The model can be continuous or discrete. If the model is discrete, it can include time delays which are treated as poles at `z = 0`.

  The reference model can be MIMO, provided that it is square and that its DC singular value (`sigma`) is 1. Then number of inputs and outputs of the reference model must match the dimensions of the inputs and outputs specified for the step response goal.

  For best results, the reference model should also include intrinsic system characteristics such as non-minimum-phase zeros (undershoot).

If your selected inputs and outputs define a MIMO system and you apply a SISO reference system, the software attempts to match the diagonal channels of the MIMO system. In that case, cross-couplings tend to be minimized.

Options

Use this section of the dialog box to specify additional characteristics of the step response goal.

- **Keep % mismatch below**

  Specify the relative matching error between the actual (tuned) step response and the target step response. Increase this value to loosen the matching tolerance. The relative matching error, e_{rel}, is defined as:

  $$e_{rel} = \frac{\|y(t) - y_{ref}(t)\|_2}{\|1 - y_{ref}(t)\|_2}.$$

  $y(t) - y_{ref}(t)$ is the response mismatch, and $1 - y_{ref}(t)$ is the step-tracking error of the target model. $\|\cdot\|_2$ denotes the signal energy (2-norm).

- **Adjust for step amplitude**

  For a MIMO tuning goal, when the choice of units results in a mix of small and large signals in different channels of the response, this option allows you to specify the relative amplitude of each entry in the vector-valued step input. This information is used to scale the off-diagonal terms in the transfer function from reference to tracking error. This scaling ensures that cross-couplings are measured relative to the amplitude of each reference signal.

For example, suppose that tuning goal is that outputs `'y1' and 'y2'` track reference signals `'r1'and 'r2'`. Suppose further that you require the outputs to track the references with less than 10% cross-coupling. If `r1` and `r2` have comparable amplitudes, then it is sufficient to keep the gains from `r1` to `y2` and `r2` and `y1` below 0.1. However, if `r1` is 100 times larger than `r2`, the gain from `r1` to `y2` must be less than 0.001 to ensure that `r1` changes `y2` by less than 10% of the `r2` target. To ensure this result, set **Adjust for step amplitude** to `Yes`. Then, enter `[100,1]` in the **Amplitudes of step commands** text box. Doing so tells **Control System Tuner** to take into account that the first reference signal is 100 times greater than the second reference signal.

The default value, `No` , means no scaling is applied.

- **Apply goal to**

  Use this option when tuning multiple models at once, such as an array of models obtained by linearizing a Simulink model at different operating points or block-parameter values. By default, active tuning goals are enforced for all models. To enforce a tuning requirement for a subset of models in an array, select **Only Models**. Then, enter the array indices of the models for which the goal is enforced. For example, suppose you want to apply the tuning goal to the second, third, and fourth models in a model array. To restrict enforcement of the requirement, enter `2:4` in the **Only Models** text box.

  For more information about tuning for multiple models, see “Robust Tuning Approaches” (Robust Control Toolbox).

Algorithms

When you tune a control system, the software converts each tuning goal into a normalized scalar value $f(x)$. Here, x is the vector of free (tunable) parameters in the control system. The software then adjusts the parameter values to minimize $f(x)$ or to drive $f(x)$ below 1 if the tuning goal is a hard constraint.

For **Step Response Goal**, $f(x)$ is given by:

$$f(x) = \frac{\left\| \frac{1}{s}\left(T(s,x) - H_{ref}(s)\right) \right\|_2}{e_{rel}\left\| \frac{1}{s}\left(H_{ref}(s) - I\right) \right\|_2}.$$

$T(s,x)$ is the closed-loop transfer function between the specified inputs and outputs, evaluated with parameter values x. $H_{ref}(s)$ is the reference model. e_{rel} is the relative error (see “Options” on page 10-47). $\| \cdot \|_2$ denotes the H_2 norm (see `norm`).

This tuning goal also imposes an implicit stability constraint on the closed-loop transfer function between the specified inputs to outputs, evaluated with loops opened at the specified loop-opening locations. The dynamics affected by this implicit constraint are the stabilized dynamics for this tuning goal. The **Minimum decay rate** and **Maximum natural frequency** tuning options control the lower and upper bounds on these implicitly constrained dynamics. If the optimization fails to meet the default bounds, or if the default bounds conflict with other requirements, on the **Tuning** tab, use **Tuning Options** to change the defaults.

See Also

Related Examples

- “Specify Goals for Interactive Tuning” on page 10-29
- “Manage Tuning Goals” on page 10-135
- “Visualize Tuning Goals” on page 10-143

Step Rejection Goal

Purpose

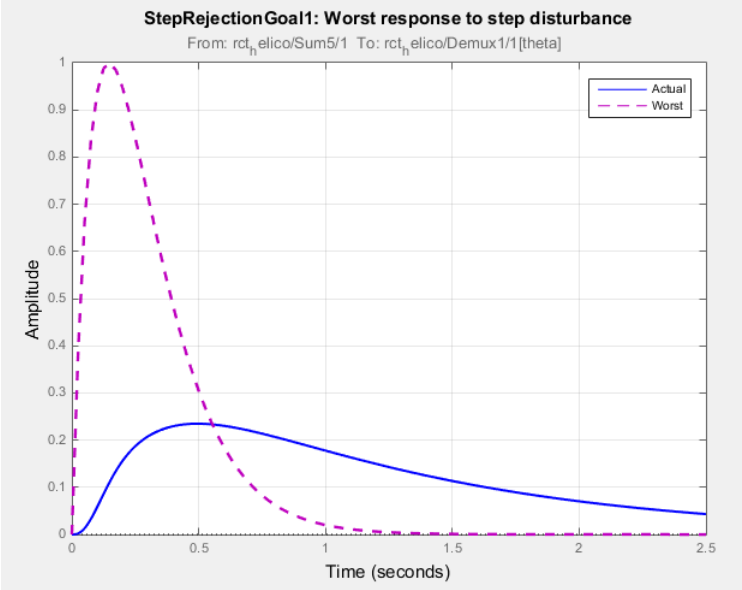
Set a minimum standard for rejecting step disturbances, when using **Control System Tuner**.

Description

Use **Step Rejection Goal** to specify how a step disturbance injected at a specified location in your control system affects the signal at a specified output location.

You can specify the desired response in time-domain terms of peak value, settling time, and damping ratio. **Control System Tuner** attempts to make the actual rejection at least as good as the desired response. Alternatively, you can specify the response as a stable reference model having DC-gain. In that case, the tuning goal is to reject the disturbance as well as or better than the reference model.

To specify disturbance rejection in terms of a frequency-domain attenuation profile, use **Disturbance Rejection Goal**.



When you create a tuning goal in **Control System Tuner**, a tuning-goal plot is generated. The dotted line shows the target step response you specify. The solid line is the current corresponding response of your system.

Creation

In the **Tuning** tab of **Control System Tuner**, select **New Goal > Rejection of step disturbance** to create a Step Rejection Goal.

Command-Line Equivalent

When tuning control systems at the command line, use `TuningGoal.StepRejection` to specify a step response goal.

Step Disturbance Response Selection

Use this section of the dialog box to specify input, output, and loop-opening locations for evaluating the tuning goal.

- **Specify step disturbance inputs**

  Select one or more signal locations in your model at which to apply the input. To constrain a SISO response, select a single-valued input signal. For example, to constrain the step-disturbance response from a location named `'u'` to a location named `'y'`, click **Add signal to list** and select `'u'`. To constrain a MIMO response, select multiple signals or a vector-valued signal.

- **Specify step response outputs**

  Select one or more signal locations in your model at which to measure the response to the step disturbance. To constrain a SISO response, select a single-valued output signal. For example, to constrain the transient response from a location named `'u'` to a location named `'y'`, click **Add signal to list** and select `'y'`. To constrain a MIMO response, select multiple signals or a vector-valued signal. For MIMO systems, the number of outputs must equal the number of inputs.

- **Compute the response with the following loops open**

  Select one or more signal locations in your model at which to open a feedback loop for the purpose of evaluating this tuning goal. The tuning goal is evaluated against the open-loop configuration created by opening feedback loops at the locations you identify. For example, to evaluate the tuning goal with an opening at a location named `'x'`, click **Add signal to list** and select `'x'`.

Tip To highlight any selected signal in the Simulink model, click . To remove a signal from the input or output list, click . When you have selected multiple signals, you can reorder them using and . For more information on how to specify signal locations for a tuning goal, see “Specify Goals for Interactive Tuning” on page 10-29.

Desired Response to Step Disturbance

Use this section of the dialog box to specify the shape of the desired response to the step disturbance. **Control System Tuner** attempts to make the actual response at least as good as the desired response.

- **Response Characteristics**

  Specify the desired response in terms of time-domain characteristics. Enter the maximum amplitude, maximum settling time, and minimum damping constant in the text boxes.

- **Reference Model**

  Specify the desired response in terms of a reference model.

  Enter the name of the reference model in the MATLAB workspace in the **Reference Model** text field. Alternatively, enter a command to create a suitable reference model, such as `tf([1 0],[1 1.414 1])`.

The reference model must be stable and must have zero DC gain. The model can be continuous or discrete. If the model is discrete, it can include time delays which are treated as poles at `z = 0`.

For best results, the reference model and the open-loop response from the disturbance to the output should have similar gains at the frequency where the reference model gain peaks.

Options

Use this section of the dialog box to specify additional characteristics of the step rejection goal.

- **Adjust for amplitude of input signals** and **Adjust for amplitude of output signals**

  For a MIMO tuning goal, when the choice of units results in a mix of small and large signals in different channels of the response, this option allows you to specify the relative amplitude of each entry in the vector-valued signals. This information is used to scale the off-diagonal terms in the transfer function from the tuning goal inputs to outputs. This scaling ensures that cross-couplings are measured relative to the amplitude of each reference signal.

  When these options are set to `No`, the closed-loop transfer function being constrained is not scaled for relative signal amplitudes. When the choice of units results in a mix of small and large signals, using an unscaled transfer function can lead to poor tuning results. Set the option to `Yes` to provide the relative amplitudes of the input signals and output signals of your transfer function.

  For example, suppose the tuning goal constrains a 2-input, 2-output transfer function. Suppose further that second input signal to the transfer function tends to be about 100 times greater than the first signal. In that case, select `Yes` and enter `[1,100]` in the **Amplitudes of input signals** text box.

  Adjusting signal amplitude causes the tuning goal to be evaluated on the scaled transfer function $D_o^{-1}T(s)D_i$, where $T(s)$ is the unscaled transfer function. D_o and D_i are diagonal matrices with the **Amplitudes of output signals** and **Amplitudes of input signals** values on the diagonal, respectively.

  The default value, `No`, means no scaling is applied.

- **Apply goal to**

  Use this option when tuning multiple models at once, such as an array of models obtained by linearizing a Simulink model at different operating points or block-parameter values. By default, active tuning goals are enforced for all models. To enforce a tuning requirement for a subset of models in an array, select **Only Models**. Then, enter the array indices of the models for which the goal is enforced. For example, suppose you want to apply the tuning goal to the second, third, and fourth models in a model array. To restrict enforcement of the requirement, enter `2:4` in the **Only Models** text box.

  For more information about tuning for multiple models, see “Robust Tuning Approaches” (Robust Control Toolbox).

Algorithms

Evaluating Tuning Goals

When you tune a control system, the software converts each tuning goal into a normalized scalar value $f(x)$. Here, x is the vector of free (tunable) parameters in the control system. The software then

adjusts the parameter values to minimize $f(x)$ or to drive $f(x)$ below 1 if the tuning requirement is a hard constraint.

Step Rejection Goal aims to keep the gain from disturbance to output below the gain of the reference model. The scalar value of the requirement $f(x)$ is given by:

$$f(x) = \left\| W_F(s) T_{dy}(s,x) \right\|_\infty,$$

or its discrete-time equivalent. Here, $T_{dy}(s,x)$ is the closed-loop transfer function of the constrained response, and $\| \cdot \|_\infty$ denotes the H_∞ norm (see `norm`). W_F is a frequency weighting function derived from the step-rejection profile you specify in the tuning goal. The gain of W_F roughly matches the inverse of the reference model for gain values within 60 dB of the peak gain. For numerical reasons, the weighting function levels off outside this range, unless you specify a reference model that changes slope outside this range. This adjustment is called regularization. Because poles of W_F close to $s = 0$ or $s =$ `Inf` might lead to poor numeric conditioning for tuning, it is not recommended to specify reference models with very low-frequency or very high-frequency dynamics. For more information about regularization and its effects, see “Visualize Tuning Goals” on page 10-143.

Implicit Constraints

This tuning goal also imposes an implicit stability constraint on the closed-loop transfer function between the specified inputs to outputs, evaluated with loops opened at the specified loop-opening locations. The dynamics affected by this implicit constraint are the stabilized dynamics for this tuning goal. The **Minimum decay rate** and **Maximum natural frequency** tuning options control the lower and upper bounds on these implicitly constrained dynamics. If the optimization fails to meet the default bounds, or if the default bounds conflict with other requirements, on the **Tuning** tab, use **Tuning Options** to change the defaults.

See Also

Related Examples

- “Specify Goals for Interactive Tuning” on page 10-29
- “Manage Tuning Goals” on page 10-135
- “Visualize Tuning Goals” on page 10-143

Transient Goal

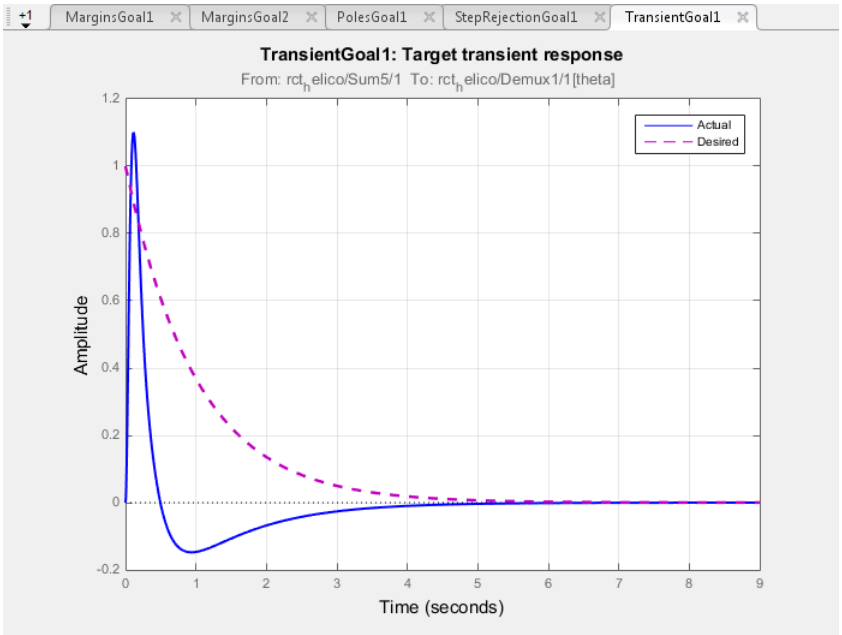
Purpose

Shape how the closed-loop system responds to a specific input signal when using **Control System Tuner**. Use a reference model to specify the desired transient response.

Description

Transient Goal constrains the transient response from specified input locations to specified output locations. This requirement specifies that the transient response closely match the response of a reference model. The constraint is satisfied when the relative difference between the tuned and target responses falls within the tolerance you specify.

You can constrain the response to an impulse, step, or ramp input signal. You can also constrain the response to an input signal that is given by the impulse response of an input filter you specify.



Creation

In the **Tuning** tab of **Control System Tuner**, select **New Goal > Transient response matching** to create a Transient Goal.

Command-Line Equivalent

When tuning control systems at the command line, use `TuningGoal.Transient` to specify a step response goal.

Response Selection

Use this section of the dialog box to specify input, output, and loop-opening locations for evaluating the tuning goal.

- **Specify response inputs**

  Select one or more signal locations in your model at which to apply the input. To constrain a SISO response, select a single-valued input signal. For example, to constrain the transient response from a location named `'u'` to a location named `'y'`, click **Add signal to list** and select `'u'`. To constrain a MIMO response, select multiple signals or a vector-valued signal.

- **Specify response outputs**

  Select one or more signal locations in your model at which to measure the transient response. To constrain a SISO response, select a single-valued output signal. For example, to constrain the transient response from a location named `'u'` to a location named `'y'`, click **Add signal to list** and select `'y'`. To constrain a MIMO response, select multiple signals or a vector-valued signal. For MIMO systems, the number of outputs must equal the number of inputs.

- **Compute the response with the following loops open**

  Select one or more signal locations in your model at which to open a feedback loop for the purpose of evaluating this tuning goal. The tuning goal is evaluated against the open-loop configuration created by opening feedback loops at the locations you identify. For example, to evaluate the tuning goal with an opening at a location named `'x'`, click **Add signal to list** and select `'x'`.

Tip To highlight any selected signal in the Simulink model, click . To remove a signal from the input or output list, click . When you have selected multiple signals, you can reorder them using and . For more information on how to specify signal locations for a tuning goal, see “Specify Goals for Interactive Tuning” on page 10-29.

Initial Signal Selection

Select the input signal shape for the transient response you want to constrain in **Control System Tuner**.

- `Impulse` — Constrain the response to a unit impulse.
- `Step` — Constrain the response to a unit step. Using `Step` is equivalent to using a Step Tracking Goal.
- `Ramp` — Constrain the response to a unit ramp, `u = t`.
- `Other` — Constrain the response to a custom input signal. Specify the custom input signal by entering a transfer function (`tf` or `zpk`model) in the **Use impulse response of filter** field. The custom input signal is the response of this transfer function to a unit impulse.

  This transfer function represents the Laplace transform of the desired custom input signal. For example, to constrain the transient response to a unit-amplitude sine wave of frequency `w`, enter `tf(w,[1,0,w^2])`. This transfer function is the Laplace transform of sin(*wt*).

  The transfer function you enter must be continuous, and can have no poles in the open right-half plane. The series connection of this transfer function with the reference system for the desired transient response must have no feedthrough term.

Desired Transient Response

Specify the reference system for the desired transient response as a dynamic system model, such as a `tf`, `zpk`, or `ss` model. The Transient Goal constrains the system response to closely match the response of this system to the input signal you specify in **Initial Signal Selection**.

Enter the name of the reference model in the MATLAB workspace in the **Reference Model** field. Alternatively, enter a command to create a suitable reference model, such as `tf(1,[1 1.414 1])`. The reference model must be stable, and the series connection of the reference model with the input shaping filter must have no feedthrough term.

Options

Use this section of the dialog box to specify additional characteristics of the transient response goal.

- **Keep % mismatch below**

  Specify the relative matching error between the actual (tuned) transient response and the target response. Increase this value to loosen the matching tolerance. The relative matching error, e_{rel}, is defined as:

  $$\text{gap} = \frac{\|y(t) - y_{ref}(t)\|_2}{\|y_{ref(tr)}(t)\|_2}.$$

  $y(t) - y_{ref}(t)$ is the response mismatch, and $1 - y_{ref(tr)}(t)$ is the transient portion of y_{ref} (deviation from steady-state value or trajectory). $\|\cdot\|_2$ denotes the signal energy (2-norm). The gap can be understood as the ratio of the root-mean-square (RMS) of the mismatch to the RMS of the reference transient.

- **Adjust for amplitude of input signals** and **Adjust for amplitude of output signals**

  For a MIMO tuning goal, when the choice of units results in a mix of small and large signals in different channels of the response, this option allows you to specify the relative amplitude of each entry in the vector-valued signals. This information is used to scale the off-diagonal terms in the transfer function from the tuning goal inputs to outputs. This scaling ensures that cross-couplings are measured relative to the amplitude of each reference signal.

  When these options are set to `No`, the closed-loop transfer function being constrained is not scaled for relative signal amplitudes. When the choice of units results in a mix of small and large signals, using an unscaled transfer function can lead to poor tuning results. Set the option to `Yes` to provide the relative amplitudes of the input signals and output signals of your transfer function.

  For example, suppose the tuning goal constrains a 2-input, 2-output transfer function. Suppose further that second input signal to the transfer function tends to be about 100 times greater than the first signal. In that case, select `Yes` and enter `[1,100]` in the **Amplitudes of input signals** text box.

  Adjusting signal amplitude causes the tuning goal to be evaluated on the scaled transfer function $D_o^{-1}T(s)D_i$, where $T(s)$ is the unscaled transfer function. D_o and D_i are diagonal matrices with the **Amplitudes of output signals** and **Amplitudes of input signals** values on the diagonal, respectively.

  The default value, `No`, means no scaling is applied.

- **Apply goal to**

  Use this option when tuning multiple models at once, such as an array of models obtained by linearizing a Simulink model at different operating points or block-parameter values. By default, active tuning goals are enforced for all models. To enforce a tuning requirement for a subset of models in an array, select **Only Models**. Then, enter the array indices of the models for which the goal is enforced. For example, suppose you want to apply the tuning goal to the second, third, and fourth models in a model array. To restrict enforcement of the requirement, enter `2:4` in the **Only Models** text box.

  For more information about tuning for multiple models, see “Robust Tuning Approaches” (Robust Control Toolbox).

Tips

- When you use this requirement to tune a control system, **Control System Tuner** attempts to enforce zero feedthrough ($D = 0$) on the transfer that the requirement constrains. Zero feedthrough is imposed because the H_2 norm, and therefore the value of the tuning goal (see “Algorithms” on page 10-57), is infinite for continuous-time systems with nonzero feedthrough.

  Control System Tuner enforces zero feedthrough by fixing to zero all tunable parameters that contribute to the feedthrough term. **Control System Tuner** returns an error when fixing these tunable parameters is insufficient to enforce zero feedthrough. In such cases, you must modify the requirement or the control structure, or manually fix some tunable parameters of your system to values that eliminate the feedthrough term.

  When the constrained transfer function has several tunable blocks in series, the software’s approach of zeroing all parameters that contribute to the overall feedthrough might be conservative. In that case, it is sufficient to zero the feedthrough term of one of the blocks. If you want to control which block has feedthrough fixed to zero, you can manually fix the feedthrough of the tuned block of your choice.

  To fix parameters of tunable blocks to specified values, see “View and Change Block Parameterization in Control System Tuner” on page 10-19.
- This tuning goal also imposes an implicit stability constraint on the closed-loop transfer function between the specified inputs to outputs, evaluated with loops opened at the specified loop-opening locations. The dynamics affected by this implicit constraint are the stabilized dynamics for this tuning goal. The **Minimum decay rate** and **Maximum natural frequency** tuning options control the lower and upper bounds on these implicitly constrained dynamics. If the optimization fails to meet the default bounds, or if the default bounds conflict with other requirements, on the **Tuning** tab, use **Tuning Options** to change the defaults.

Algorithms

When you tune a control system, the software converts each tuning goal into a normalized scalar value $f(x)$. Here, x is the vector of free (tunable) parameters in the control system. The software then adjusts the parameter values to minimize $f(x)$ or to drive $f(x)$ below 1 if the tuning requirement is a hard constraint.

For **Transient Goal**, $f(x)$ is based upon the relative gap between the tuned response and the target response:

$$\text{gap} = \frac{\|y(t) - y_{ref}(t)\|_2}{\|y_{ref(tr)}(t)\|_2}.$$

$y(t) - y_{ref}(t)$ is the response mismatch, and $1 - y_{ref(tr)}(t)$ is the transient portion of y_{ref} (deviation from steady-state value or trajectory). $\|\cdot\|_2$ denotes the signal energy (2-norm). The gap can be understood as the ratio of the root-mean-square (RMS) of the mismatch to the RMS of the reference transient.

See Also

Related Examples

- “Specify Goals for Interactive Tuning” on page 10-29
- “Manage Tuning Goals” on page 10-135
- “Visualize Tuning Goals” on page 10-143

LQR/LQG Goal

Purpose

Minimize or limit Linear-Quadratic-Gaussian (LQG) cost in response to white-noise inputs, when using **Control System Tuner**.

Description

LQR/LQG Goal specifies a tuning requirement for quantifying control performance as an LQG cost. It is applicable to any control structure, not just the classical observer structure of optimal LQG control.

The LQG cost is given by:

$J = E(z(t)' \, QZ \, z(t))$.

$z(t)$ is the system response to a white noise input vector $w(t)$. The covariance of $w(t)$ is given by:

$E(w(t)w(t)') = QW$.

The vector $w(t)$ typically consists of external inputs to the system such as noise, disturbances, or command. The vector $z(t)$ includes all the system variables that characterize performance, such as control signals, system states, and outputs. $E(x)$ denotes the expected value of the stochastic variable x.

The cost function J can also be written as an average over time:

$$J = \lim_{T \to \infty} E\left(\frac{1}{T}\int_0^T z(t)' \, QZ \, z(t)dt\right).$$

Creation

In the **Tuning** tab of **Control System Tuner**, select **New Goal > LQR/LQG objective** to create an LQR/LQG Goal.

Command-Line Equivalent

When tuning control systems at the command line, use `TuningGoal.LQG` to specify an LQR/LQG goal.

Signal Selection

Use this section of the dialog box to specify noise input locations and performance output locations. Also specify any locations at which to open loops for evaluating the tuning goal.

- **Specify noise inputs (w)**

  Select one or more signal locations in your model as noise inputs. To constrain a SISO response, select a single-valued input signal. For example, to constrain the LQG cost for a noise input `'u'` and performance output `'y'`, click ➕ **Add signal to list** and select `'u'`. To constrain the LQG cost for a MIMO response, select multiple signals or a vector-valued signal.

- **Specify performance outputs (z)**

  Select one or more signal locations in your model as performance outputs. To constrain a SISO response, select a single-valued output signal. For example, to constrain the LQG cost for a noise input `'u'` and performance output `'y'`, click **Add signal to list** and select `'y'`. To constrain the LQG cost for a MIMO response, select multiple signals or a vector-valued signal.

- **Evaluate LQG objective with the following loops open**

  Select one or more signal locations in your model at which to open a feedback loop for the purpose of evaluating this tuning goal. The tuning goal is evaluated against the open-loop configuration created by opening feedback loops at the locations you identify. For example, to evaluate the tuning goal with an opening at a location named `'x'`, click **Add signal to list** and select `'x'`.

Tip To highlight any selected signal in the Simulink model, click . To remove a signal from the input or output list, click . When you have selected multiple signals, you can reorder them using and . For more information on how to specify signal locations for a tuning goal, see “Specify Goals for Interactive Tuning” on page 10-29.

LQG Objective

Use this section of the dialog box to specify the noise covariance and performance weights for the LQG goal.

- **Performance weight Qz**

  Performance weights, specified as a scalar or a matrix. Use a scalar value to specify a multiple of the identity matrix. Otherwise specify a symmetric nonnegative definite matrix. Use a diagonal matrix to independently scale or penalize the contribution of each variable in z.

  The performance weights contribute to the cost function according to:

  $J = E(z(t)' Qz\, z(t))$.

  When you use the LQG goal as a hard goal, the software tries to drive the cost function $J < 1$. When you use it as a soft goal, the cost function J is minimized subject to any hard goals and its value is contributed to the overall objective function. Therefore, select Qz values to properly scale the cost function so that driving it below 1 or minimizing it yields the performance you require.

- **Noise Covariance Qw**

  Covariance of the white noise input vector $w(t)$, specified as a scalar or a matrix. Use a scalar value to specify a multiple of the identity matrix. Otherwise specify a symmetric nonnegative definite matrix with as many rows as there are entries in the vector $w(t)$. A diagonal matrix means the entries of $w(t)$ are uncorrelated.

  The covariance of $w(t$ is given by:

  $E(w(t)w(t)') = QW$.

When you are tuning a control system in discrete time, the LQG goal assumes:

$E(w[k]w[k]') = QW/T_s$.

T_s is the model sample time. This assumption ensures consistent results with tuning in the continuous-time domain. In this assumption, $w[k]$ is discrete-time noise obtained by sampling continuous white noise $w(t)$ with covariance QW. If in your system $w[k]$ is a truly discrete process with known covariance QWd, use the value T_s*QWd for the QW value.

Options

Use this section of the dialog box to specify additional characteristics of the LQG goal.

- **Apply goal to**

  Use this option when tuning multiple models at once, such as an array of models obtained by linearizing a Simulink model at different operating points or block-parameter values. By default, active tuning goals are enforced for all models. To enforce a tuning requirement for a subset of models in an array, select **Only Models**. Then, enter the array indices of the models for which the goal is enforced. For example, suppose you want to apply the tuning goal to the second, third, and fourth models in a model array. To restrict enforcement of the requirement, enter `2:4` in the **Only Models** text box.

  For more information about tuning for multiple models, see “Robust Tuning Approaches” (Robust Control Toolbox).

Tips

When you use this requirement to tune a control system, **Control System Tuner** attempts to enforce zero feedthrough ($D = 0$) on the transfer that the requirement constrains. Zero feedthrough is imposed because the H_2 norm, and therefore the value of the tuning goal, is infinite for continuous-time systems with nonzero feedthrough.

Control System Tuner enforces zero feedthrough by fixing to zero all tunable parameters that contribute to the feedthrough term. **Control System Tuner** returns an error when fixing these tunable parameters is insufficient to enforce zero feedthrough. In such cases, you must modify the requirement or the control structure, or manually fix some tunable parameters of your system to values that eliminate the feedthrough term.

When the constrained transfer function has several tunable blocks in series, the software’s approach of zeroing all parameters that contribute to the overall feedthrough might be conservative. In that case, it is sufficient to zero the feedthrough term of one of the blocks. If you want to control which block has feedthrough fixed to zero, you can manually fix the feedthrough of the tuned block of your choice.

To fix parameters of tunable blocks to specified values, see “View and Change Block Parameterization in Control System Tuner” on page 10-19.

Algorithms

When you tune a control system, the software converts each tuning goal into a normalized scalar value $f(x)$. Here, x is the vector of free (tunable) parameters in the control system. The software then

adjusts the parameter values to minimize $f(x)$ or to drive $f(x)$ below 1 if the tuning goal is a hard constraint.

For **LQR/LQG Goal**, $f(x)$ is given by the cost function J:

$J = E(z(t)' \, Qz \, z(t))$.

When you use the LQG requirement as a hard goal, the software tries to drive the cost function $J < 1$. When you use it as a soft goal, the cost function J is minimized subject to any hard goals and its value is contributed to the overall objective function. Therefore, select `Qz` values to properly scale the cost function so that driving it below 1 or minimizing it yields the performance you require.

See Also

Related Examples

- “Specify Goals for Interactive Tuning” on page 10-29
- “Manage Tuning Goals” on page 10-135

Gain Goal

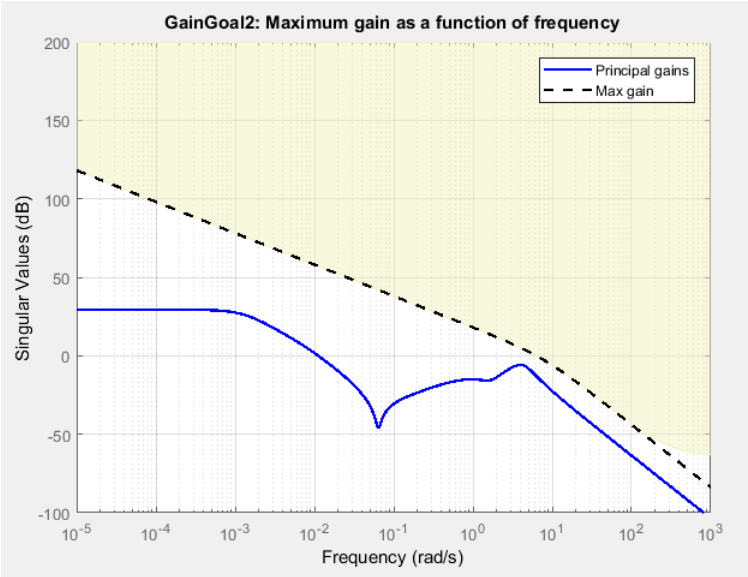
Purpose

Limit gain of a specified input/output transfer function, when using **Control System Tuner**.

Description

Gain Goal limits the gain from specified inputs to specified outputs. If you specify multiple inputs and outputs, Gain Goal limits the largest singular value of the transfer matrix. (See `sigma` for more information about singular values.) You can specify a constant maximum gain at all frequencies. Alternatively, you can specify a frequency-dependent gain profile.

Use Gain Goal, for example, to enforce a custom roll-off rate in a particular frequency band. To do so, specify a maximum gain profile in that band. You can also use Gain Goal to enforce disturbance rejection across a particular input/output pair by constraining the gain to be less than 1.



When you create a tuning goal in **Control System Tuner**, a tuning-goal plot is generated. The dotted line shows the gain profile you specify. The shaded area on the plot represents the region in the frequency domain where the gain goal is not satisfied.

By default, Gain Goal constrains a closed-loop gain. To constrain a gain computed with one or more loops open, specify loop-opening locations in the **I/O Transfer Selection** section of the dialog box.

Creation

In the **Tuning** tab of **Control System Tuner**, select **New Goal** > **Gain limits** to create a Gain Goal.

Command-Line Equivalent

When tuning control systems at the command line, use `TuningGoal.Gain` to specify a maximum gain goal.

I/O Transfer Selection

Use this section of the dialog box to specify the inputs and outputs of the transfer function that the tuning goal constrains. Also specify any locations at which to open loops for evaluating the tuning goal.

- **Specify input signals**

  Select one or more signal locations in your model as inputs to the transfer function that the tuning goal constrains. To constrain a SISO response, select a single-valued input signal. For example, to constrain the gain from a location named `'u'` to a location named `'y'`, click **Add signal to list** and select `'u'`. To constrain the largest singular value of a MIMO response, select multiple signals or a vector-valued signal.
- **Specify output signals**

  Select one or more signal locations in your model as outputs of the transfer function that the tuning goal constrains. To constrain a SISO response, select a single-valued output signal. For example, to constrain the gain from a location named `'u'` to a location named `'y'`, click **Add signal to list** and select `'y'`. To constrain the largest singular value of a MIMO response, select multiple signals or a vector-valued signal.
- **Compute input/output gain with the following loops open**

  Select one or more signal locations in your model at which to open a feedback loop for the purpose of evaluating this tuning goal. The tuning goal is evaluated against the open-loop configuration created by opening feedback loops at the locations you identify. For example, to evaluate the tuning goal with an opening at a location named `'x'`, click **Add signal to list** and select `'x'`.

Tip To highlight any selected signal in the Simulink model, click . To remove a signal from the input or output list, click . When you have selected multiple signals, you can reorder them using and . For more information on how to specify signal locations for a tuning goal, see “Specify Goals for Interactive Tuning” on page 10-29.

Options

Use this section of the dialog box to specify additional characteristics of the gain goal.

- **Limit gain to**

  Enter the maximum gain in the text box. You can specify a scalar value or a frequency-dependent gain profile. To specify a frequency-dependent gain profile, enter a SISO numeric LTI model. For example, you can specify a smooth transfer function (`tf`, `zpk`, or `ss` model). Alternatively, you can sketch a piecewise maximum gain using an `frd` model. When you do so, the software automatically maps the profile to a smooth transfer function that approximates the desired minimum disturbance rejection. For example, to specify a gain profile that rolls off at –40dB/decade in the frequency band from 8 to 800 rad/s, enter `frd([0.8 8 800],[10 1 1e-4])`.

  You must specify a SISO transfer function. If you specify multiple input signals or output signals, the gain profile applies to all I/O pairs between these signals.

If you are tuning in discrete time, you can specify the maximum gain profile as a discrete-time model with the same sampling time as you use for tuning. If you specify the gain profile in continuous time, the tuning software discretizes it. Specifying the gain profile in discrete time gives you more control over the gain profile near the Nyquist frequency.

- **Stabilize I/O transfer**

  By default, the tuning goal imposes a stability requirement on the closed-loop transfer function from the specified inputs to outputs, in addition to the gain constraint. If stability is not required or cannot be achieved, select `No` to remove the stability requirement. For example, if the gain constraint applies to an unstable open-loop transfer function, select `No`.

- **Enforce goal in frequency range**

  Limit the enforcement of the tuning goal to a particular frequency band. Specify the frequency band as a row vector of the form `[min,max]`, expressed in frequency units of your model. For example, to create a tuning goal that applies only between 1 and 100 rad/s, enter `[1,100]`. By default, the tuning goal applies at all frequencies for continuous time, and up to the Nyquist frequency for discrete time.

- **Adjust for signal amplitude**

  When this option is set to `No`, the closed-loop transfer function being constrained is not scaled for relative signal amplitudes. When the choice of units results in a mix of small and large signals, using an unscaled transfer function can lead to poor tuning results. Set the option to `Yes` to provide the relative amplitudes of the input signals and output signals of your transfer function.

  For example, suppose the tuning goal constrains a 2-input, 2-output transfer function. Suppose further that second input signal to the transfer function tends to be about 100 times greater than the first signal. In that case, select `Yes` and enter `[1,100]` in the **Amplitude of input signals** text box.

  Adjusting signal amplitude causes the tuning goal to be evaluated on the scaled transfer function $D_o^{-1}T(s)D_i$, where $T(s)$ is the unscaled transfer function. D_o and D_i are diagonal matrices with the **Amplitude of output signals** and **Amplitude of input signals** values on the diagonal, respectively.

- **Apply goal to**

  Use this option when tuning multiple models at once, such as an array of models obtained by linearizing a Simulink model at different operating points or block-parameter values. By default, active tuning goals are enforced for all models. To enforce a tuning requirement for a subset of models in an array, select **Only Models**. Then, enter the array indices of the models for which the goal is enforced. For example, suppose you want to apply the tuning goal to the second, third, and fourth models in a model array. To restrict enforcement of the requirement, enter `2:4` in the **Only Models** text box.

  For more information about tuning for multiple models, see “Robust Tuning Approaches” (Robust Control Toolbox).

Algorithms

Evaluating Tuning Goals

When you tune a control system, the software converts each tuning goal into a normalized scalar value $f(x)$. Here, x is the vector of free (tunable) parameters in the control system. The software then

adjusts the parameter values to minimize $f(x)$ or to drive $f(x)$ below 1 if the tuning goal is a hard constraint.

For **Gain Goal**, $f(x)$ is given by:

$$f(x) = \left\| W_F(s) D_o^{-1} T(s,x) D_i \right\|_\infty,$$

or its discrete-time equivalent. Here, $T(s,x)$ is the closed-loop transfer function between the specified inputs and outputs, evaluated with parameter values x. D_o and D_i are the scaling matrices described in "Options" on page 10-64. $\| \cdot \|_\infty$ denotes the H_∞ norm (see `getPeakGain`).

The frequency weighting function W_F is the regularized gain profile, derived from the maximum gain profile you specify. The gain of W_F roughly matches the inverse of the gain profile you specify, inside the frequency band you set in the **Enforce goal in frequency range** field of the tuning goal. W_F is always stable and proper. Because poles of $W_F(s)$ close to $s = 0$ or $s =$ `Inf` might lead to poor numeric conditioning for tuning, it is not recommended to specify maximum gain profiles with very low-frequency or very high-frequency dynamics. For more information about regularization and its effects, see "Visualize Tuning Goals" on page 10-143.

Implicit Constraints

This tuning goal also imposes an implicit stability constraint on the closed-loop transfer function between the specified inputs to outputs, evaluated with loops opened at the specified loop-opening locations. The dynamics affected by this implicit constraint are the stabilized dynamics for this tuning goal. The **Minimum decay rate** and **Maximum natural frequency** tuning options control the lower and upper bounds on these implicitly constrained dynamics. If the optimization fails to meet the default bounds, or if the default bounds conflict with other requirements, on the **Tuning** tab, use **Tuning Options** to change the defaults.

See Also

Related Examples

- "Specify Goals for Interactive Tuning" on page 10-29
- "Manage Tuning Goals" on page 10-135
- "Visualize Tuning Goals" on page 10-143

Variance Goal

Purpose

Limit white-noise impact on specified output signals, when using **Control System Tuner**.

Description

Variance Goal imposes a noise attenuation constraint that limits the impact on specified output signals of white noise applied at specified inputs. The noise attenuation is measured by the ratio of the noise variance to the output variance.

For stochastic inputs with a nonuniform spectrum (colored noise), use “Weighted Variance Goal” on page 10-92 instead.

Creation

In the **Tuning** tab of **Control System Tuner**, select **New Goal** > **Signal variance attenuation** to create a Variance Goal.

Command-Line Equivalent

When tuning control systems at the command line, use `TuningGoal.Variance` to specify a constraint on noise amplification.

I/O Transfer Selection

Use this section of the dialog box to specify noise input locations and response outputs. Also specify any locations at which to open loops for evaluating the tuning goal.

- **Specify stochastic inputs**

  Select one or more signal locations in your model as noise inputs. To constrain a SISO response, select a single-valued input signal. For example, to constrain the gain from a location named `'u'` to a location named `'y'`, click **Add signal to list** and select `'u'`. To constrain the noise amplification of a MIMO response, select multiple signals or a vector-valued signal.

- **Specify stochastic outputs**

  Select one or more signal locations in your model as outputs for computing response to the noise inputs. To constrain a SISO response, select a single-valued output signal. For example, to constrain the gain from a location named `'u'` to a location named `'y'`, click **Add signal to list** and select `'y'`. To constrain the noise amplification of a MIMO response, select multiple signals or a vector-valued signal.

- **Compute output variance with the following loops open**

  Select one or more signal locations in your model at which to open a feedback loop for the purpose of evaluating this tuning goal. The tuning goal is evaluated against the open-loop configuration created by opening feedback loops at the locations you identify. For example, to evaluate the tuning goal with an opening at a location named `'x'`, click **Add signal to list** and select `'x'`.

Tip To highlight any selected signal in the Simulink model, click . To remove a signal from the input or output list, click . When you have selected multiple signals, you can reorder them using and . For more information on how to specify signal locations for a tuning goal, see “Specify Goals for Interactive Tuning” on page 10-29.

Options

Use this section of the dialog box to specify additional characteristics of the variance goal.

- **Attenuate input variance by a factor**

  Enter the desired noise attenuation from the specified inputs to outputs. This value specifies the maximum ratio of noise variance to output variance.

  When you tune a control system in discrete time, this requirement assumes that the physical plant and noise process are continuous, and interprets the desired noise attenuation as a bound on the continuous-time H_2 norm. This assumption ensures that continuous-time and discrete-time tuning give consistent results. If the plant and noise processes are truly discrete, and you want to bound the discrete-time H_2 norm instead, divide the desired attenuation value by $\sqrt{T_s}$, where T_s is the sample time of the model you are tuning.

- **Adjust for signal amplitude**

  When this option is set to `No`, the closed-loop transfer function being constrained is not scaled for relative signal amplitudes. When the choice of units results in a mix of small and large signals, using an unscaled transfer function can lead to poor tuning results. Set the option to `Yes` to provide the relative amplitudes of the input signals and output signals of your transfer function.

  For example, suppose the tuning goal constrains a 2-input, 2-output transfer function. Suppose further that second input signal to the transfer function tends to be about 100 times greater than the first signal. In that case, select `Yes` and enter `[1,100]` in the **Amplitude of input signals** text box.

  Adjusting signal amplitude causes the tuning goal to be evaluated on the scaled transfer function $D_o^{-1}T(s)D_i$, where $T(s)$ is the unscaled transfer function. D_o and D_i are diagonal matrices with the **Amplitude of output signals** and **Amplitude of input signals** values on the diagonal, respectively.

- **Apply goal to**

  Use this option when tuning multiple models at once, such as an array of models obtained by linearizing a Simulink model at different operating points or block-parameter values. By default, active tuning goals are enforced for all models. To enforce a tuning requirement for a subset of models in an array, select **Only Models**. Then, enter the array indices of the models for which the goal is enforced. For example, suppose you want to apply the tuning goal to the second, third, and fourth models in a model array. To restrict enforcement of the requirement, enter `2:4` in the **Only Models** text box.

  For more information about tuning for multiple models, see “Robust Tuning Approaches” (Robust Control Toolbox).

Tips

- When you use this requirement to tune a control system, **Control System Tuner** attempts to enforce zero feedthrough ($D = 0$) on the transfer that the requirement constrains. Zero feedthrough is imposed because the H_2 norm, and therefore the value of the tuning goal (see "Algorithms" on page 10-69), is infinite for continuous-time systems with nonzero feedthrough.

  Control System Tuner enforces zero feedthrough by fixing to zero all tunable parameters that contribute to the feedthrough term. **Control System Tuner** returns an error when fixing these tunable parameters is insufficient to enforce zero feedthrough. In such cases, you must modify the requirement or the control structure, or manually fix some tunable parameters of your system to values that eliminate the feedthrough term.

  When the constrained transfer function has several tunable blocks in series, the software's approach of zeroing all parameters that contribute to the overall feedthrough might be conservative. In that case, it is sufficient to zero the feedthrough term of one of the blocks. If you want to control which block has feedthrough fixed to zero, you can manually fix the feedthrough of the tuned block of your choice.

  To fix parameters of tunable blocks to specified values, see "View and Change Block Parameterization in Control System Tuner" on page 10-19.
- This tuning goal also imposes an implicit stability constraint on the closed-loop transfer function between the specified inputs to outputs, evaluated with loops opened at the specified loop-opening locations. The dynamics affected by this implicit constraint are the stabilized dynamics for this tuning goal. The **Minimum decay rate** and **Maximum natural frequency** tuning options control the lower and upper bounds on these implicitly constrained dynamics. If the optimization fails to meet the default bounds, or if the default bounds conflict with other requirements, on the **Tuning** tab, use **Tuning Options** to change the defaults.

Algorithms

When you tune a control system, the software converts each tuning goal into a normalized scalar value $f(x)$. Here, x is the vector of free (tunable) parameters in the control system. The software then adjusts the parameter values to minimize $f(x)$ or to drive $f(x)$ below 1 if the tuning goal is a hard constraint.

For **Variance Goal**, $f(x)$ is given by:

$$f(x) = \|\text{Attenuation} \cdot T(s,x)\|_2 .$$

$T(s,x)$ is the closed-loop transfer function from `Input` to `Output`. $\|\cdot\|_2$ denotes the H_2 norm (see `norm`).

For tuning discrete-time control systems, $f(x)$ is given by:

$$f(x) = \left\| \frac{\text{Attenuation}}{\sqrt{T_s}} T(z,x) \right\|_2 .$$

T_s is the sample time of the discrete-time transfer function $T(z,x)$.

See Also

Related Examples

- “Specify Goals for Interactive Tuning” on page 10-29
- “Manage Tuning Goals” on page 10-135
- “Visualize Tuning Goals” on page 10-143

Reference Tracking Goal

Purpose

Make specified outputs track reference inputs with prescribed performance and fidelity, when using **Control System Tuner**. Limit cross-coupling in MIMO systems.

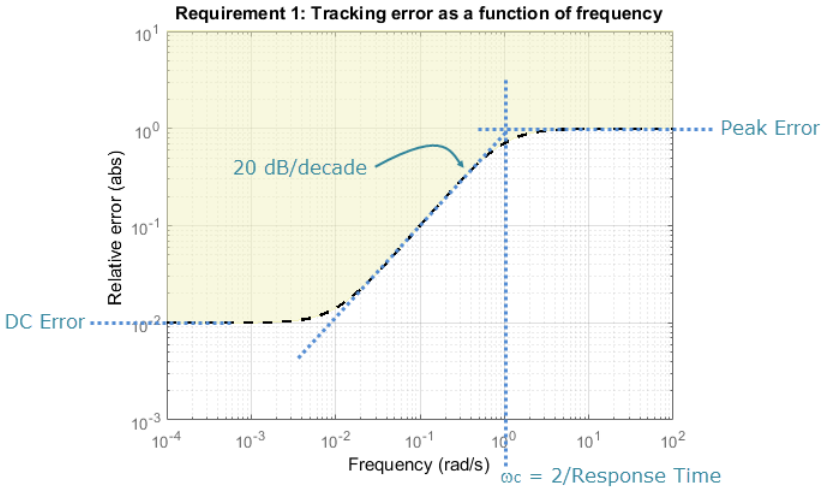
Description

Reference Tracking Goal constrains tracking between the specified signal locations. The constraint is satisfied when the maximum relative tracking error falls below the value you specify at all frequencies. The relative error is the gain from reference input to tracking error as a function of frequency.

You can specify the maximum error profile directly as a function of frequency. Alternatively, you can specify the tracking goal a target DC error, peak error, and response time. These parameters are converted to the following transfer function that describes the maximum frequency-domain tracking error:

$$\text{MaxError} = \frac{(\text{PeakError})s + \omega_c(\text{DCError})}{s + \omega_c}.$$

Here, ω_c is 2/(response time). The following plot illustrates these relationships for an example set of values.



When you create a tuning goal in **Control System Tuner**, a tuning-goal plot is generated. The dotted line shows the error profile you specify. The shaded area on the plot represents the region in the frequency domain where the tuning goal is not satisfied.

Creation

In the **Tuning** tab of **Control System Tuner**, select **New Goal > Reference Tracking** to create a Reference Tracking Goal.

Command-Line Equivalent

When tuning control systems at the command line, use `TuningGoal.Tracking` to specify a tracking goal.

Response Selection

Use this section of the dialog box to specify input, output, and loop-opening locations for evaluating the tuning goal.

- **Specify reference inputs**

  Select one or more signal locations in your model as reference signals. To constrain a SISO response, select a single-valued reference signal. For example, to constrain the response from a location named `'u'` to a location named `'y'`, click **Add signal to list** and select `'u'`. To constrain a MIMO response, select multiple signals or a vector-valued signal.
- **Specify reference-tracking outputs**

  Select one or more signal locations in your model as reference-tracking outputs. To constrain a SISO response, select a single-valued output signal. For example, to constrain the step response from a location named `'u'` to a location named `'y'`, click **Add signal to list** and select `'y'`. To constrain a MIMO response, select multiple signals or a vector-valued signal. For MIMO systems, the number of outputs must equal the number of inputs.
- **Evaluate tracking performance with the following loops open**

  Select one or more signal locations in your model at which to open a feedback loop for the purpose of evaluating this tuning goal. The tuning goal is evaluated against the open-loop configuration created by opening feedback loops at the locations you identify. For example, to evaluate the tuning goal with an opening at a location named `'x'`, click **Add signal to list** and select `'x'`.

Tip To highlight any selected signal in the Simulink model, click . To remove a signal from the input or output list, click . When you have selected multiple signals, you can reorder them using and . For more information on how to specify signal locations for a tuning goal, see “Specify Goals for Interactive Tuning” on page 10-29.

Tracking Performance

Use this section of the dialog box to specify frequency-domain constraints on the tracking error.

Response time, DC error, and peak error

Select this option to specify the tracking error in terms of response time, percent steady-state error, and peak error across all frequencies. These parameters are converted to the following transfer function that describes the maximum frequency-domain tracking error:

$$\text{MaxError} = \frac{(\text{PeakError})s + \omega_c(\text{DCError})}{s + \omega_c}.$$

When you select this option, enter the following parameters in the text boxes:

- **Response Time** — Enter the target response time. The tracking bandwidth is given by ω_c = 2/Response Time. Express the target response time in the time units of your model.
- **Steady-state error (%)** — Enter the maximum steady-state fractional tracking error, expressed in percent. For MIMO tracking goals, this steady-state error applies to all I/O pairs. The steady-state error is the DC error expressed as a percentage, `DCError`/100.
- **Peak error across frequency (%)** — Enter the maximum fractional tracking error across all frequencies, expressed in percent.

Maximum error as a function of frequency

Select this option to specify the maximum tracking error profile as a function of frequency.

Enter a SISO numeric LTI model in the text box. For example, you can specify a smooth transfer function (`tf`, `zpk`, or `ss` model). Alternatively, you can sketch a piecewise error profile using an `frd` model. When you do so, the software automatically maps the error profile to a smooth transfer function that approximates the desired error profile. For example, to specify a maximum error of 0.01 below about 1 rad/s, gradually rising to a peak error of 1 at 100 rad/s, enter `frd([0.01 0.01 1], [0 1 100])`.

For MIMO tracking goals, this error profile applies to all I/O pairs.

If you are tuning in discrete time, you can specify the maximum error profile as a discrete-time model with the same sampling time as you use for tuning. If you specify the attenuation profile in continuous time, the tuning software discretizes it. Specifying the error profile in discrete time gives you more control over the profile near the Nyquist frequency.

Options

Use this section of the dialog box to specify additional characteristics of the tracking goal.

- **Enforce goal in frequency range**

  Limit the enforcement of the tuning goal to a particular frequency band. Specify the frequency band as a row vector of the form `[min,max]`, expressed in frequency units of your model. For example, to create a tuning goal that applies only between 1 and 100 rad/s, enter `[1,100]`. By default, the tuning goal applies at all frequencies for continuous time, and up to the Nyquist frequency for discrete time.

- **Adjust for step amplitude**

  For a MIMO tuning goal, when the choice of units results in a mix of small and large signals in different channels of the response, this option allows you to specify the relative amplitude of each entry in the vector-valued step input. This information is used to scale the off-diagonal terms in the transfer function from reference to tracking error. This scaling ensures that cross-couplings are measured relative to the amplitude of each reference signal.

  For example, suppose that tuning goal is that outputs `'y1' and 'y2'` track reference signals `'r1'and 'r2'`. Suppose further that you require the outputs to track the references with less than 10% cross-coupling. If `r1` and `r2` have comparable amplitudes, then it is sufficient to keep the gains from `r1` to `y2` and `r2` and `y1` below 0.1. However, if `r1` is 100 times larger than `r2`, the gain from `r1` to `y2` must be less than 0.001 to ensure that `r1` changes `y2` by less than 10% of the `r2` target. To ensure this result, set **Adjust for step amplitude** to `Yes`. Then, enter `[100,1]` in

the **Amplitudes of step commands** text box. Doing so tells **Control System Tuner** to take into account that the first reference signal is 100 times greater than the second reference signal.

The default value, `No` , means no scaling is applied.

- **Apply goal to**

  Use this option when tuning multiple models at once, such as an array of models obtained by linearizing a Simulink model at different operating points or block-parameter values. By default, active tuning goals are enforced for all models. To enforce a tuning requirement for a subset of models in an array, select **Only Models**. Then, enter the array indices of the models for which the goal is enforced. For example, suppose you want to apply the tuning goal to the second, third, and fourth models in a model array. To restrict enforcement of the requirement, enter `2:4` in the **Only Models** text box.

  For more information about tuning for multiple models, see “Robust Tuning Approaches” (Robust Control Toolbox).

Algorithms

Evaluating Tuning Goals

When you tune a control system, the software converts each tuning goal into a normalized scalar value $f(x)$. Here, x is the vector of free (tunable) parameters in the control system. The software then adjusts the parameter values to minimize $f(x)$ or to drive $f(x)$ below 1 if the tuning goal is a hard constraint.

For **Tracking Goal**, $f(x)$ is given by:

$$f(x) = \|W_F(s)(T(s,x) - I)\|_\infty,$$

or its discrete-time equivalent. Here, $T(s,x)$ is the closed-loop transfer function between the specified inputs and outputs, and $\|\cdot\|_\infty$ denotes the H_∞ norm (see `getPeakGain`). W_F is a frequency weighting function derived from the error profile you specify in the tuning goal. The gain of W_F roughly matches the inverse of the error profile for gain values between –20 dB and 60 dB. For numerical reasons, the weighting function levels off outside this range, unless you specify a reference model that changes slope outside this range. This adjustment is called regularization. Because poles of W_F close to $s = 0$ or $s =$ `Inf` might lead to poor numeric conditioning of the `systune` optimization problem, it is not recommended to specify error profiles with very low-frequency or very high-frequency dynamics. For more information about regularization and its effects, see “Visualize Tuning Goals” on page 10-143.

Implicit Constraints

This tuning goal also imposes an implicit stability constraint on the closed-loop transfer function between the specified inputs to outputs, evaluated with loops opened at the specified loop-opening locations. The dynamics affected by this implicit constraint are the stabilized dynamics for this tuning goal. The **Minimum decay rate** and **Maximum natural frequency** tuning options control the lower and upper bounds on these implicitly constrained dynamics. If the optimization fails to meet the default bounds, or if the default bounds conflict with other requirements, on the **Tuning** tab, use **Tuning Options** to change the defaults.

See Also

Related Examples

- “Specify Goals for Interactive Tuning” on page 10-29
- “Visualize Tuning Goals” on page 10-143
- “Manage Tuning Goals” on page 10-135

Overshoot Goal

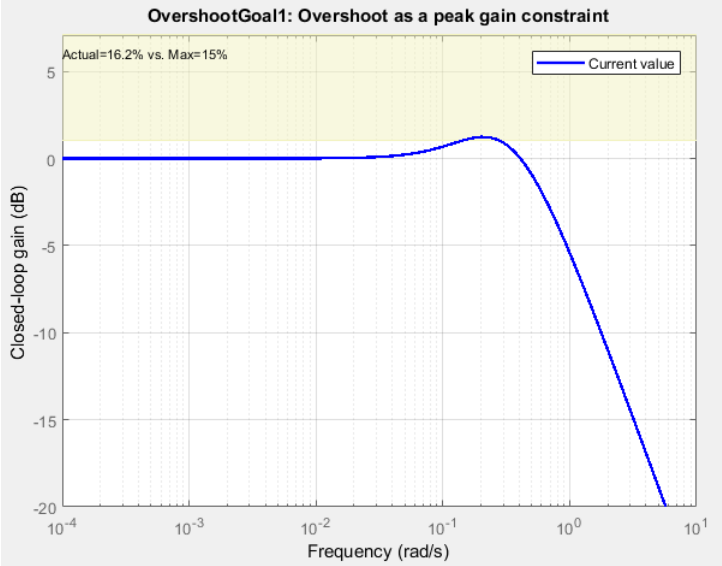
Purpose

Limit overshoot in the step response from specified inputs to specified outputs, when using **Control System Tuner**.

Description

Overshoot Goal limits the overshoot in the step response between the specified signal locations. The constraint is satisfied when the overshoot in the tuned response is less than the target overshoot

The software maps the maximum overshoot to a peak gain constraint, assuming second-order system characteristics. Therefore, for tuning higher-order systems, the overshoot constraint is only approximate. In addition, the Overshoot Goal cannot reliably reduce the overshoot below 5%.



When you create a tuning goal in **Control System Tuner**, a tuning-goal plot is generated. The shaded area on the plot represents the region in the frequency domain where the tuning goal is not satisfied.

Creation

In the **Tuning** tab of **Control System Tuner**, select **New Goal > Maximum overshoot** to create an Overshoot Goal.

Command-Line Equivalent

When tuning control systems at the command line, use `TuningGoal.Overshoot` to specify a step response goal.

Response Selection

Use this section of the dialog box to specify input, output, and loop-opening locations for evaluating the tuning goal.

- **Specify step-response inputs**

  Select one or more signal locations in your model at which to apply the step input. To constrain a SISO response, select a single-valued input signal. For example, to constrain the step response from a location named `'u'` to a location named `'y'`, click **Add signal to list** and select `'u'`. To constrain a MIMO response, select multiple signals or a vector-valued signal.
- **Specify step-response outputs**

  Select one or more signal locations in your model at which to measure the response to the step input. To constrain a SISO response, select a single-valued output signal. For example, to constrain the step response from a location named `'u'` to a location named `'y'`, click **Add signal to list** and select `'y'`. To constrain a MIMO response, select multiple signals or a vector-valued signal. For MIMO systems, the number of outputs must equal the number of inputs.
- **Evaluate overshoot with the following loops open**

  Select one or more signal locations in your model at which to open a feedback loop for the purpose of evaluating this tuning goal. The tuning goal is evaluated against the open-loop configuration created by opening feedback loops at the locations you identify. For example, to evaluate the tuning goal with an opening at a location named `'x'`, click **Add signal to list** and select `'x'`.

Tip To highlight any selected signal in the Simulink model, click . To remove a signal from the input or output list, click . When you have selected multiple signals, you can reorder them using and . For more information on how to specify signal locations for a tuning goal, see “Specify Goals for Interactive Tuning” on page 10-29.

Options

Use this section of the dialog box to specify additional characteristics of the overshoot goal.

- **Limit % overshoot to**

  Enter the maximum percent overshoot. Overshoot Goal cannot reliably reduce the overshoot below 5%
- **Adjust for step amplitude**

  For a MIMO tuning goal, when the choice of units results in a mix of small and large signals in different channels of the response, this option allows you to specify the relative amplitude of each entry in the vector-valued step input. This information is used to scale the off-diagonal terms in the transfer function from reference to tracking error. This scaling ensures that cross-couplings are measured relative to the amplitude of each reference signal.

  For example, suppose that tuning goal is that outputs `'y1' and 'y2'` track reference signals `'r1'and 'r2'`. Suppose further that you require the outputs to track the references with less

than 10% cross-coupling. If `r1` and `r2` have comparable amplitudes, then it is sufficient to keep the gains from `r1` to `y2` and `r2` and `y1` below 0.1. However, if `r1` is 100 times larger than `r2`, the gain from `r1` to `y2` must be less than 0.001 to ensure that `r1` changes `y2` by less than 10% of the `r2` target. To ensure this result, set **Adjust for step amplitude** to `Yes`. Then, enter `[100,1]` in the **Amplitudes of step commands** text box. Doing so tells **Control System Tuner** to take into account that the first reference signal is 100 times greater than the second reference signal.

The default value, `No` , means no scaling is applied.

- **Apply goal to**

  Use this option when tuning multiple models at once, such as an array of models obtained by linearizing a Simulink model at different operating points or block-parameter values. By default, active tuning goals are enforced for all models. To enforce a tuning requirement for a subset of models in an array, select **Only Models**. Then, enter the array indices of the models for which the goal is enforced. For example, suppose you want to apply the tuning goal to the second, third, and fourth models in a model array. To restrict enforcement of the requirement, enter `2:4` in the **Only Models** text box.

  For more information about tuning for multiple models, see “Robust Tuning Approaches” (Robust Control Toolbox).

Algorithms

When you tune a control system, the software converts each tuning goal into a normalized scalar value $f(x)$. Here, x is the vector of free (tunable) parameters in the control system. The software then adjusts the parameter values to minimize $f(x)$ or to drive $f(x)$ below 1 if the tuning goal is a hard constraint.

For **Overshoot Goal**, $f(x)$ reflects the relative satisfaction or violation of the goal. The percent deviation from $f(x) = 1$ roughly corresponds to the percent deviation from the specified overshoot target. For example, $f(x) = 1.2$ means the actual overshoot exceeds the target by roughly 20%, and $f(x) = 0.8$ means the actual overshoot is about 20% less than the target.

Overshoot Goal uses $\|T\|_\infty$ as a proxy for the overshoot, based on second-order model characteristics. Here, T is the closed-loop transfer function that the requirement constrains. The overshoot is tuned in the range from 5% ($\|T\|_\infty = 1$) to 100% ($\|T\|_\infty$). **Overshoot Goal** is ineffective at forcing the overshoot below 5%.

This tuning goal also imposes an implicit stability constraint on the closed-loop transfer function between the specified inputs to outputs, evaluated with loops opened at the specified loop-opening locations. The dynamics affected by this implicit constraint are the stabilized dynamics for this tuning goal. The **Minimum decay rate** and **Maximum natural frequency** tuning options control the lower and upper bounds on these implicitly constrained dynamics. If the optimization fails to meet the default bounds, or if the default bounds conflict with other requirements, on the **Tuning** tab, use **Tuning Options** to change the defaults.

See Also

Related Examples

- “Specify Goals for Interactive Tuning” on page 10-29

- “Manage Tuning Goals” on page 10-135
- “Visualize Tuning Goals” on page 10-143

Disturbance Rejection Goal

Purpose

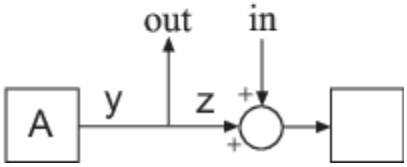
Attenuate disturbances at particular locations and in particular frequency bands, when using **Control System Tuner**.

Description

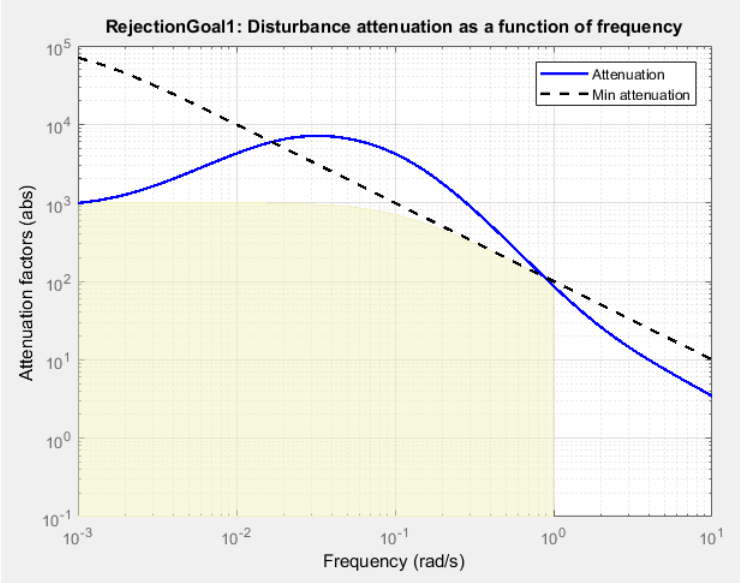
Disturbance Rejection Goal specifies the minimum attenuation of a disturbance injected at a specified location in a control system.

When you use this tuning goal, the software attempts to tune the system so that the attenuation of a disturbance at the specified location exceeds the minimum attenuation factor you specify. This attenuation factor is the ratio between the open- and closed-loop sensitivities to the disturbance, and is a function of frequency.

The following diagram illustrates how the attenuation factor is calculated. Suppose you specify a location in your control system, `y`, which is the output of a block `A`. In that case, the software calculates the closed-loop sensitivity at `out` to a signal injected at `in`. The software also calculates the sensitivity with the control loop opened at the location `z`.



To specify a Disturbance Rejection Goal, you specify one or more locations at which to attenuate disturbance. You also provide the frequency-dependent minimum attenuation factor as a numeric LTI model. You can achieve disturbance attenuation only inside the control bandwidth. The loop gain must be larger than one for the disturbance to be attenuated (attenuation factor > 1).



When you create a tuning goal in **Control System Tuner**, a tuning-goal plot is generated. The dotted line shows the gain profile you specify. The shaded area on the plot represents the region in the frequency domain where the tuning goal is not satisfied. The solid line is the current corresponding response of your system.

If you prefer to specify sensitivity to disturbance at a location, rather than disturbance attenuation, you can use “Sensitivity Goal” on page 10-85.

Creation

In the **Tuning** tab of **Control System Tuner**, select **New Goal** > **Disturbance rejection** to create a Disturbance Rejection Goal.

Command-Line Equivalent

When tuning control systems at the command line, use `TuningGoal.Rejection` to specify a disturbance rejection goal.

Disturbance Scenario

Use this section of the dialog box to specify the signal locations at which to inject the disturbance. You can also specify loop-opening locations for evaluating the tuning goal.

- **Inject disturbances at the following locations**

  Select one or more signal locations in your model at which to measure the disturbance attenuation. To constrain a SISO response, select a single-valued location. For example, to attenuate disturbance at a location named `'y'`, click ➕ **Add signal to list** and select `'y'`. To constrain a MIMO response, select multiple signals or a vector-valued signal.
- **Evaluate disturbance rejection with the following loops open**

  Select one or more signal locations in your model at which to open a feedback loop for the purpose of evaluating this tuning goal. The tuning goal is evaluated against the open-loop

configuration created by opening feedback loops at the locations you identify. For example, to evaluate the tuning goal with an opening at a location named `'x'`, click **Add signal to list** and select `'x'`.

Tip To highlight any selected signal in the Simulink model, click . To remove a signal from the input or output list, click . When you have selected multiple signals, you can reorder them using and . For more information on how to specify signal locations for a tuning goal, see “Specify Goals for Interactive Tuning” on page 10-29.

Rejection Performance

Specify the minimum disturbance attenuation as a function of frequency.

Enter a SISO numeric LTI model whose magnitude represents the desired attenuation profile as a function of frequency. For example, you can specify a smooth transfer function (`tf`, `zpk`, or `ss` model). Alternatively, you can sketch a piecewise minimum disturbance rejection using an `frd` model. When you do so, the software automatically maps the profile to a smooth transfer function that approximates the desired minimum disturbance rejection. For example, to specify an attenuation factor of 100 (40 dB) below 1 rad/s, that gradually drops to 1 (0 dB) past 10 rad/s, enter `frd([100 100 1 1],[0 1 10 100])`.

If you are tuning in discrete time, you can specify the attenuation profile as a discrete-time model with the same sampling time as you use for tuning. If you specify the attenuation profile in continuous time, the tuning software discretizes it. Specifying the attenuation profile in discrete time gives you more control over the profile near the Nyquist frequency.

Options

Use this section of the dialog box to specify additional characteristics of the disturbance rejection goal.

- **Enforce goal in frequency range**

  Limit the enforcement of the tuning goal to a particular frequency band. Specify the frequency band as a row vector of the form `[min,max]`, expressed in frequency units of your model. For example, to create a tuning goal that applies only between 1 and 100 rad/s, enter `[1,100]`. By default, the tuning goal applies at all frequencies for continuous time, and up to the Nyquist frequency for discrete time.

  Regardless of the limits you enter, a disturbance rejection goal can only be enforced within the control bandwidth.

- **Equalize cross-channel effects**

  For multiloop or MIMO disturbance rejection requirements, the feedback channels are automatically rescaled to equalize the off-diagonal (loop interaction) terms in the open-loop transfer function. Select `Off` to disable such scaling and shape the unscaled open-loop response.

- **Apply goal to**

Use this option when tuning multiple models at once, such as an array of models obtained by linearizing a Simulink model at different operating points or block-parameter values. By default, active tuning goals are enforced for all models. To enforce a tuning requirement for a subset of models in an array, select **Only Models**. Then, enter the array indices of the models for which the goal is enforced. For example, suppose you want to apply the tuning goal to the second, third, and fourth models in a model array. To restrict enforcement of the requirement, enter `2:4` in the **Only Models** text box.

For more information about tuning for multiple models, see “Robust Tuning Approaches” (Robust Control Toolbox).

Algorithms

Evaluating Tuning Goals

When you tune a control system, the software converts each tuning goal into a normalized scalar value $f(x)$. Here, x is the vector of free (tunable) parameters in the control system. The software then adjusts the parameter values to minimize $f(x)$ or to drive $f(x)$ below 1 if the tuning goal is a hard constraint.

For **Disturbance Rejection Goal**, $f(x)$ is given by:

$$f(x) = \max_{\omega \in \Omega} \|W_S(j\omega)S(j\omega,x)\|_\infty,$$

or its discrete-time equivalent. Here, $S(j\omega,x)$ is the closed-loop sensitivity function measured at the disturbance location. Ω is the frequency interval over which the requirement is enforced, specified in the **Enforce goal in frequency range** field. W_S is a frequency weighting function derived from the attenuation profile you specify. The gains of W_S and the specified profile roughly match for gain values ranging from –20 dB to 60 dB. For numerical reasons, the weighting function levels off outside this range, unless the specified gain profile changes slope outside this range. This adjustment is called regularization. Because poles of W_S close to $s = 0$ or $s = \texttt{Inf}$ might lead to poor numeric conditioning for tuning, it is not recommended to specify loop shapes with very low-frequency or very high-frequency dynamics. For more information about regularization and its effects, see “Visualize Tuning Goals” on page 10-143.

Implicit Constraints

This tuning goal imposes an implicit stability constraint on the closed-loop sensitivity function measured at the specified, evaluated with loops opened at the specified loop-opening locations. The dynamics affected by this implicit constraint are the stabilized dynamics for this tuning goal. The **Minimum decay rate** and **Maximum natural frequency** tuning options control the lower and upper bounds on these implicitly constrained dynamics. If the optimization fails to meet the default bounds, or if the default bounds conflict with other requirements, on the **Tuning** tab, use **Tuning Options** to change the defaults.

See Also

Related Examples

- “Specify Goals for Interactive Tuning” on page 10-29
- “Manage Tuning Goals” on page 10-135

- “Visualize Tuning Goals” on page 10-143

Sensitivity Goal

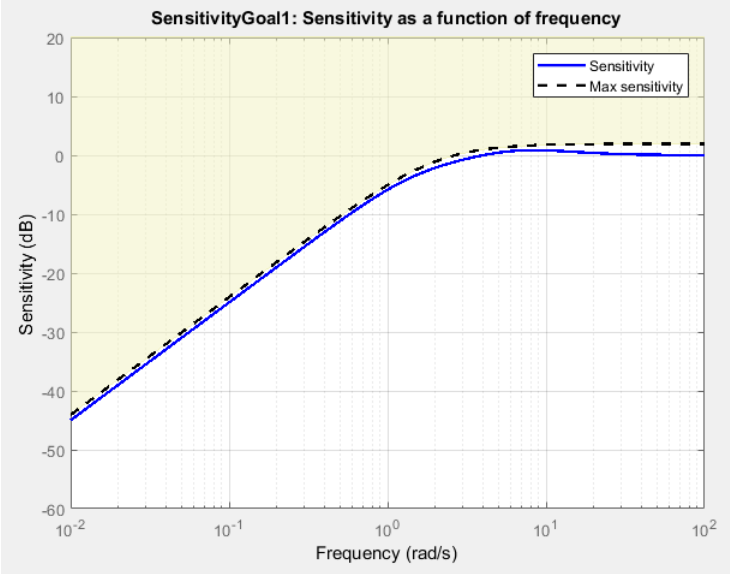
Purpose

Limit sensitivity of feedback loops to disturbances, when using **Control System Tuner**.

Description

Sensitivity Goal limits the sensitivity of a feedback loop to disturbances. You specify the maximum sensitivity as a function of frequency. Constrain the sensitivity to be smaller than one at frequencies where you need good disturbance rejection.

To specify a Sensitivity Goal, you specify one or more locations at which to limit sensitivity. You also provide the frequency-dependent maximum sensitivity as a numeric LTI model whose magnitude represents the desired sensitivity as a function of frequency.



When you create a tuning goal in **Control System Tuner**, a tuning-goal plot is generated. The dotted line shows the gain profile you specify. The shaded area on the plot represents the region in the frequency domain where the tuning goal is not satisfied.

If you prefer to specify disturbance attenuation at a particular location, rather than sensitivity to disturbance, you can use “Disturbance Rejection Goal” on page 10-80.

Creation

In the **Tuning** tab of **Control System Tuner**, select **New Goal > Sensitivity of feedback loops** to create a Sensitivity Goal.

Command-Line Equivalent

When tuning control systems at the command line, use `TuningGoal.Sensitivity` to specify a disturbance rejection goal.

Sensitivity Evaluation

Use this section of the dialog box to specify the signal locations at which to compute the sensitivity to disturbance. You can also specify loop-opening locations for evaluating the tuning goal.

- **Measure sensitivity at the following locations**

  Select one or more signal locations in your model at which to measure the sensitivity to disturbance. To constrain a SISO response, select a single-valued location. For example, to limit sensitivity at a location named `'y'`, click **Add signal to list** and select `'y'`. To constrain a MIMO response, select multiple signals or a vector-valued signal.
- **Evaluate disturbance rejection with the following loops open**

  Select one or more signal locations in your model at which to open a feedback loop for the purpose of evaluating this tuning goal. The tuning goal is evaluated against the open-loop configuration created by opening feedback loops at the locations you identify. For example, to evaluate the tuning goal with an opening at a location named `'x'`, click **Add signal to list** and select `'x'`.

Tip To highlight any selected signal in the Simulink model, click . To remove a signal from the input or output list, click . When you have selected multiple signals, you can reorder them using and . For more information on how to specify signal locations for a tuning goal, see “Specify Goals for Interactive Tuning” on page 10-29.

Sensitivity Bound

Specify the maximum sensitivity as a function of frequency.

Enter a SISO numeric LTI model whose magnitude represents the desired sensitivity bound as a function of frequency. For example, you can specify a smooth transfer function (`tf`, `zpk`, or `ss` model). Alternatively, you can sketch a piecewise maximum sensitivity using an `frd` model. When you do so, the software automatically maps the profile to a smooth transfer function that approximates the desired sensitivity. For example, to specify a sensitivity that rolls up at 20 dB per decade and levels off at unity above 1 rad/s, enter `frd([0.01 1 1],[0.001 0.1 100])`.

If you are tuning in discrete time, you can specify the maximum sensitivity profile as a discrete-time model with the same sampling time as you use for tuning. If you specify the sensitivity profile in continuous time, the tuning software discretizes it. Specifying the profile in discrete time gives you more control over the profile near the Nyquist frequency.

Options

Use this section of the dialog box to specify additional characteristics of the sensitivity goal.

- **Enforce goal in frequency range**

  Limit the enforcement of the tuning goal to a particular frequency band. Specify the frequency band as a row vector of the form `[min,max]`, expressed in frequency units of your model. For

example, to create a tuning goal that applies only between 1 and 100 rad/s, enter `[1,100]`. By default, the tuning goal applies at all frequencies for continuous time, and up to the Nyquist frequency for discrete time.

- **Equalize cross-channel effects**

  For multiloop or MIMO sensitivity requirements, the feedback channels are automatically rescaled to equalize the off-diagonal (loop interaction) terms in the open-loop transfer function. Select `Off` to disable such scaling and shape the unscaled open-loop response.

- **Apply goal to**

  Use this option when tuning multiple models at once, such as an array of models obtained by linearizing a Simulink model at different operating points or block-parameter values. By default, active tuning goals are enforced for all models. To enforce a tuning requirement for a subset of models in an array, select **Only Models**. Then, enter the array indices of the models for which the goal is enforced. For example, suppose you want to apply the tuning goal to the second, third, and fourth models in a model array. To restrict enforcement of the requirement, enter `2:4` in the **Only Models** text box.

  For more information about tuning for multiple models, see “Robust Tuning Approaches” (Robust Control Toolbox).

Algorithms

Evaluating Tuning Goals

When you tune a control system, the software converts each tuning goal into a normalized scalar value $f(x)$. Here, x is the vector of free (tunable) parameters in the control system. The software then adjusts the parameter values to minimize $f(x)$ or to drive $f(x)$ below 1 if the tuning goal is a hard constraint.

For **Sensitivity Goal**, $f(x)$ is given by:

$$f(x) = \|W_S(s)S(s,x)\|_\infty,$$

or its discrete-time equivalent. Here, $S(s,x)$ is the closed-loop sensitivity function measured at the location specified in the tuning goal. $\|\cdot\|_\infty$ denotes the H_∞ norm (see `norm`). W_S is a frequency weighting function derived from the sensitivity profile you specify. The gain of W_S roughly matches the inverse of the specified profile for gain values ranging from –20 dB to 60 dB. For numerical reasons, the weighting function levels off outside this range, unless the specified gain profile changes slope outside this range. This adjustment is called regularization. Because poles of W_S close to $s = 0$ or $s = \texttt{Inf}$ might lead to poor numeric conditioning for tuning, it is not recommended to specify sensitivity profiles with very low-frequency or very high-frequency dynamics. For more information about regularization and its effects, see “Visualize Tuning Goals” on page 10-143.

Implicit Constraint

This tuning goal imposes an implicit stability constraint on the closed-loop sensitivity function measured at the specified, evaluated with loops opened at the specified loop-opening locations. The dynamics affected by this implicit constraint are the stabilized dynamics for this tuning goal. The **Minimum decay rate** and **Maximum natural frequency** tuning options control the lower and upper bounds on these implicitly constrained dynamics. If the optimization fails to meet the default bounds, or if the default bounds conflict with other requirements, on the **Tuning** tab, use **Tuning Options** to change the defaults.

See Also

Related Examples

- “Specify Goals for Interactive Tuning” on page 10-29
- “Manage Tuning Goals” on page 10-135
- “Visualize Tuning Goals” on page 10-143

Weighted Gain Goal

Purpose

Frequency-weighted gain limit for tuning with **Control System Tuner**.

Description

Weighted Gain Goal limits the gain of the frequency-weighted transfer function $WL(s)H(s)WR(s)$, where $H(s)$ is the transfer function between inputs and outputs you specify. $WL(s)$ and $WR(s)$ are weighting functions that you can use to emphasize particular frequency bands. Weighted Gain Goal constrains the peak gain of $WL(s)H(s)WR(s)$ to values less than 1. If $H(s)$ is a MIMO transfer function, Weighted Gain Goal constrains the largest singular value of $H(s)$.

By default, Weighted Gain Goal constrains a closed-loop gain. To constrain a gain computed with one or more loops open, specify loop-opening locations in the **I/O Transfer Selection** section of the dialog box.

Creation

In the **Tuning** tab of **Control System Tuner**, select **New Goal > Frequency-weighted gain limit** to create a Weighted Gain Goal.

Command-Line Equivalent

When tuning control systems at the command line, use `TuningGoal.WeightedGain` to specify a weighted gain goal.

I/O Transfer Selection

Use this section of the dialog box to specify the inputs and outputs of the transfer function that the tuning goal constrains. Also specify any locations at which to open loops for evaluating the tuning goal.

- **Specify input signals**

  Select one or more signal locations in your model as inputs to the transfer function that the tuning goal constrains. To constrain a SISO response, select a single-valued input signal. For example, to constrain the gain from a location named `'u'` to a location named `'y'`, click **Add signal to list** and select `'u'`. To constrain the largest singular value of a MIMO response, select multiple signals or a vector-valued signal.

- **Specify output signals**

  Select one or more signal locations in your model as outputs of the transfer function that the tuning goal constrains. To constrain a SISO response, select a single-valued output signal. For example, to constrain the gain from a location named `'u'` to a location named `'y'`, click **Add signal to list** and select `'y'`. To constrain the largest singular value of a MIMO response, select multiple signals or a vector-valued signal.

- **Compute input/output gain with the following loops open**

Select one or more signal locations in your model at which to open a feedback loop for the purpose of evaluating this tuning goal. The tuning goal is evaluated against the open-loop configuration created by opening feedback loops at the locations you identify. For example, to evaluate the tuning goal with an opening at a location named `'x'`, click **Add signal to list** and select `'x'`.

Tip To highlight any selected signal in the Simulink model, click . To remove a signal from the input or output list, click . When you have selected multiple signals, you can reorder them using and . For more information on how to specify signal locations for a tuning goal, see “Specify Goals for Interactive Tuning” on page 10-29.

Weights

Use the **Left weight WL** and **Right weight WR** text boxes to specify the frequency-weighting functions for the tuning goal. The tuning goal ensures that the gain $H(s)$ from the specified input to output satisfies the inequality:

$$||WL(s)H(s)WR(s)||_\infty < 1.$$

WL provides the weighting for the output channels of $H(s)$, and WR provides the weighting for the input channels. You can specify scalar weights or frequency-dependent weighting. To specify a frequency-dependent weighting, use a numeric LTI model whose magnitude represents the desired weighting function. For example, enter `tf(1,[1 0.01])` to specify a high weight at low frequencies that rolls off above 0.01 rad/s.

If the tuning goal constrains a MIMO transfer function, scalar or SISO weighting functions automatically expand to any input or output dimension. You can specify different weights for each channel by specifying matrices or MIMO weighting functions. The dimensions $H(s)$ must be commensurate with the dimensions of WL and WR. For example, if the constrained transfer function has two inputs, you can specify `diag([1 10])` as WR.

If you are tuning in discrete time, you can specify the weighting functions as discrete-time models with the same sampling time as you use for tuning. If you specify the weighting functions in continuous time, the tuning software discretizes them. Specifying the weighting functions in discrete time gives you more control over the weighting functions near the Nyquist frequency.

Options

Use this section of the dialog box to specify additional characteristics of the weighted gain goal.

- **Stabilize I/O transfer**

  By default, the tuning goal imposes a stability requirement on the closed-loop transfer function from the specified inputs to outputs, in addition to the gain constraint. If stability is not required or cannot be achieved, select `No` to remove the stability requirement. For example, if the gain constraint applies to an unstable open-loop transfer function, select `No`.

- **Enforce goal in frequency range**

Limit the enforcement of the tuning goal to a particular frequency band. Specify the frequency band as a row vector of the form `[min,max]`, expressed in frequency units of your model. For example, to create a tuning goal that applies only between 1 and 100 rad/s, enter `[1,100]`. By default, the tuning goal applies at all frequencies for continuous time, and up to the Nyquist frequency for discrete time.

- **Apply goal to**

  Use this option when tuning multiple models at once, such as an array of models obtained by linearizing a Simulink model at different operating points or block-parameter values. By default, active tuning goals are enforced for all models. To enforce a tuning requirement for a subset of models in an array, select **Only Models**. Then, enter the array indices of the models for which the goal is enforced. For example, suppose you want to apply the tuning goal to the second, third, and fourth models in a model array. To restrict enforcement of the requirement, enter `2:4` in the **Only Models** text box.

  For more information about tuning for multiple models, see “Robust Tuning Approaches” (Robust Control Toolbox).

Algorithms

When you tune a control system, the software converts each tuning goal into a normalized scalar value $f(x)$. Here, x is the vector of free (tunable) parameters in the control system. The software then adjusts the parameter values to minimize $f(x)$ or to drive $f(x)$ below 1 if the tuning goal is a hard constraint.

For **Weighted Gain Goal**, $f(x)$ is given by:

$$f(x) = \|WL\, H(s,x)\; WR\|_\infty .$$

$H(s,x)$ is the closed-loop transfer function between the specified inputs and outputs, evaluated with parameter values x. Here, $\|\cdot\|_\infty$ denotes the H_∞ norm (see `getPeakGain`).

This tuning goal also imposes an implicit stability constraint on the weighted closed-loop transfer function between the specified inputs to outputs, evaluated with loops opened at the specified loop-opening locations. The dynamics affected by this implicit constraint are the stabilized dynamics for this tuning goal. The **Minimum decay rate** and **Maximum natural frequency** tuning options control the lower and upper bounds on these implicitly constrained dynamics. If the optimization fails to meet the default bounds, or if the default bounds conflict with other requirements, on the **Tuning** tab, use **Tuning Options** to change the defaults.

See Also

Related Examples

- “Specify Goals for Interactive Tuning” on page 10-29
- “Visualize Tuning Goals” on page 10-143
- “Manage Tuning Goals” on page 10-135

Weighted Variance Goal

Purpose

Frequency-weighted limit on noise impact on specified output signals for tuning with **Control System Tuner**.

Description

Weighted Variance Goal limits the noise impact on the outputs of the frequency-weighted transfer function $WL(s)H(s)WR(s)$, where $H(s)$ is the transfer function between inputs and outputs you specify. $WL(s)$ and $WR(s)$ are weighting functions you can use to model a noise spectrum or emphasize particular frequency bands. Thus, you can use Weighted Variance Goal to tune the system response to stochastic inputs with a nonuniform spectrum such as colored noise or wind gusts.

Weighted Variance minimizes the response to noise at the inputs by minimizing the H_2 norm of the frequency-weighted transfer function. The H_2 norm measures:

- The total energy of the impulse response, for deterministic inputs to the transfer function.
- The square root of the output variance for a unit-variance white-noise input, for stochastic inputs to the transfer function. Equivalently, the H_2 norm measures the root-mean-square of the output for such input.

Creation

In the **Tuning** tab of **Control System Tuner**, select **New Goal > Frequency-weighted variance attenuation** to create a Weighted Variance Goal.

Command-Line Equivalent

When tuning control systems at the command line, use `TuningGoal.WeightedVariance` to specify a weighted gain goal.

I/O Transfer Selection

Use this section of the dialog box to specify noise input locations and response outputs. Also specify any locations at which to open loops for evaluating the tuning goal.

- **Specify stochastic inputs**

  Select one or more signal locations in your model as noise inputs. To constrain a SISO response, select a single-valued input signal. For example, to constrain the gain from a location named `'u'` to a location named `'y'`, click **Add signal to list** and select `'u'`. To constrain the noise amplification of a MIMO response, select multiple signals or a vector-valued signal.

- **Specify stochastic outputs**

  Select one or more signal locations in your model as outputs for computing response to the noise inputs. To constrain a SISO response, select a single-valued output signal. For example, to constrain the gain from a location named `'u'` to a location named `'y'`, click **Add signal to list** and select `'y'`. To constrain the noise amplification of a MIMO response, select multiple signals or a vector-valued signal.

- **Compute output variance with the following loops open**

  Select one or more signal locations in your model at which to open a feedback loop for the purpose of evaluating this tuning goal. The tuning goal is evaluated against the open-loop configuration created by opening feedback loops at the locations you identify. For example, to evaluate the tuning goal with an opening at a location named `'x'`, click **Add signal to list** and select `'x'`.

Tip To highlight any selected signal in the Simulink model, click . To remove a signal from the input or output list, click . When you have selected multiple signals, you can reorder them using and . For more information on how to specify signal locations for a tuning goal, see “Specify Goals for Interactive Tuning” on page 10-29.

Weights

Use the **Left weight WL** and **Right weight WR** text boxes to specify the frequency-weighting functions for the tuning goal.

WL provides the weighting for the output channels of $H(s)$, and *WR* provides the weighting for the input channels.

You can specify scalar weights or frequency-dependent weighting. To specify a frequency-dependent weighting, use a numeric LTI model whose magnitude represents the desired weighting as a function of frequency. For example, enter `tf(1,[1 0.01])` to specify a high weight at low frequencies that rolls off above 0.01 rad/s. To limit the response to a nonuniform noise distribution, enter as *WR* an LTI model whose magnitude represents the noise spectrum.

If the tuning goal constrains a MIMO transfer function, scalar or SISO weighting functions automatically expand to any input or output dimension. You can specify different weights for each channel by specifying MIMO weighting functions. The dimensions $H(s)$ must be commensurate with the dimensions of *WL* and *WR*. For example, if the constrained transfer function has two inputs, you can specify `diag([1 10])` as *WR*.

If you are tuning in discrete time, you can specify the weighting functions as discrete-time models with the same sampling time as you use for tuning. If you specify the weighting functions in continuous time, the tuning software discretizes them. Specifying the weighting functions in discrete time gives you more control over the weighting functions near the Nyquist frequency.

Options

Use this section of the dialog box to specify additional characteristics of the weighted variance goal.

- **Apply goal to**

  Use this option when tuning multiple models at once, such as an array of models obtained by linearizing a Simulink model at different operating points or block-parameter values. By default, active tuning goals are enforced for all models. To enforce a tuning requirement for a subset of models in an array, select **Only Models**. Then, enter the array indices of the models for which the goal is enforced. For example, suppose you want to apply the tuning goal to the second, third, and

fourth models in a model array. To restrict enforcement of the requirement, enter `2:4` in the **Only Models** text box.

For more information about tuning for multiple models, see “Robust Tuning Approaches” (Robust Control Toolbox).

Tips

- When you use this requirement to tune a control system, **Control System Tuner** attempts to enforce zero feedthrough ($D = 0$) on the transfer that the requirement constrains. Zero feedthrough is imposed because the H_2 norm, and therefore the value of the tuning goal (see “Algorithms” on page 10-94), is infinite for continuous-time systems with nonzero feedthrough.

  Control System Tuner enforces zero feedthrough by fixing to zero all tunable parameters that contribute to the feedthrough term. **Control System Tuner** returns an error when fixing these tunable parameters is insufficient to enforce zero feedthrough. In such cases, you must modify the requirement or the control structure, or manually fix some tunable parameters of your system to values that eliminate the feedthrough term.

  When the constrained transfer function has several tunable blocks in series, the software’s approach of zeroing all parameters that contribute to the overall feedthrough might be conservative. In that case, it is sufficient to zero the feedthrough term of one of the blocks. If you want to control which block has feedthrough fixed to zero, you can manually fix the feedthrough of the tuned block of your choice.

  To fix parameters of tunable blocks to specified values, see “View and Change Block Parameterization in Control System Tuner” on page 10-19.
- This tuning goal also imposes an implicit stability constraint on the weighted closed-loop transfer function between the specified inputs to outputs, evaluated with loops opened at the specified loop-opening locations. The dynamics affected by this implicit constraint are the stabilized dynamics for this tuning goal. The **Minimum decay rate** and **Maximum natural frequency** tuning options control the lower and upper bounds on these implicitly constrained dynamics. If the optimization fails to meet the default bounds, or if the default bounds conflict with other requirements, on the **Tuning** tab, use **Tuning Options** to change the defaults.

Algorithms

When you tune a control system, the software converts each tuning goal into a normalized scalar value $f(x)$. Here, x is the vector of free (tunable) parameters in the control system. The software then adjusts the parameter values to minimize $f(x)$ or to drive $f(x)$ below 1 if the tuning goal is a hard constraint.

For **Weighted Variance Goal**, $f(x)$ is given by:

$$f(x) = \|WL\, H(s,x)\, WR\|_2 .$$

$H(s,x)$ is the closed-loop transfer function between the specified inputs and outputs, evaluated with parameter values x. $\|\cdot\|_2$ denotes the H_2 norm (see `norm`).

For tuning discrete-time control systems, $f(x)$ is given by:

$$f(x) = \frac{1}{\sqrt{T_s}} \|WL(z)\, H(z,x)\, WR(z)\|_2 .$$

T_s is the sample time of the discrete-time transfer function $H(z,x)$.

See Also

Related Examples

- “Specify Goals for Interactive Tuning” on page 10-29
- “Visualize Tuning Goals” on page 10-143
- “Manage Tuning Goals” on page 10-135

Minimum Loop Gain Goal

Purpose

Boost gain of feedback loops at low frequency when using **Control System Tuner**.

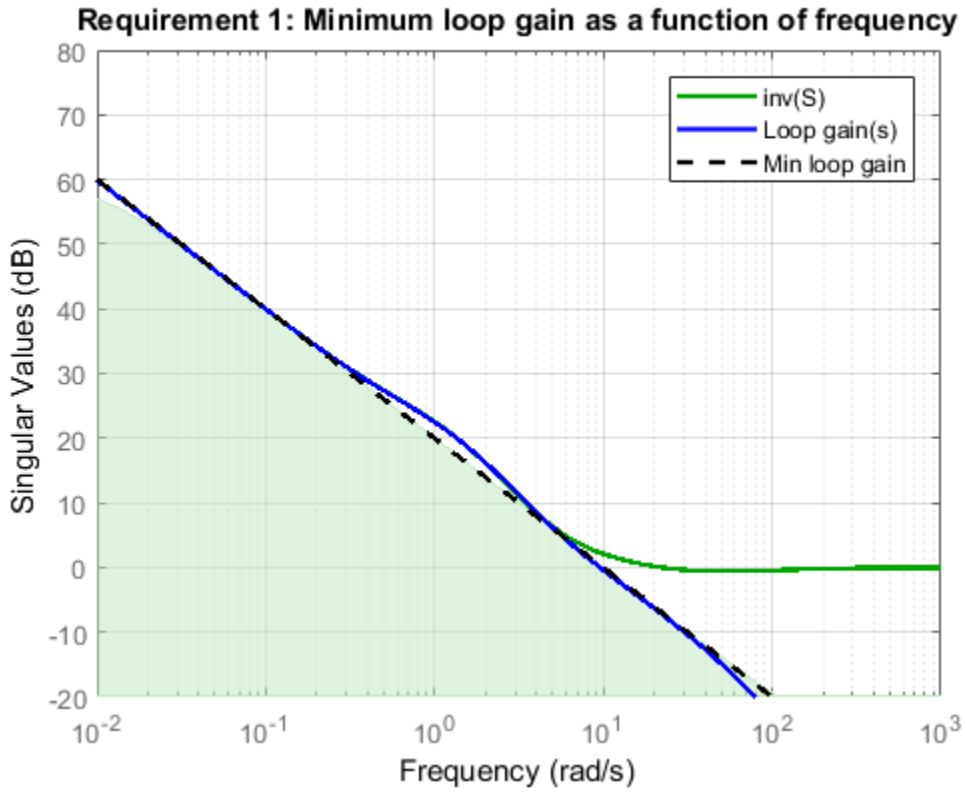
Description

Minimum Loop Gain Goal enforces a minimum loop gain in a particular frequency band. This tuning goal is useful, for example, for improving disturbance rejection at a particular location.

Minimum Loop Gain Goal imposes a minimum gain on the open-loop frequency response (L) at a specified location in your control system. You specify the minimum open-loop gain as a function of frequency (a minimum gain profile). For MIMO feedback loops, the specified gain profile is interpreted as a lower bound on the smallest singular value of L.

When you tune a control system, the minimum gain profile is converted to a minimum gain constraint on the inverse of the sensitivity function, $\text{inv}(S) = (I + L)$.

The following figure shows a typical specified minimum gain profile (dashed line) and a resulting tuned loop gain, L (blue line). The green region represents gain profile values that are forbidden by this requirement. The figure shows that when L is much larger than 1, imposing a minimum gain on $\text{inv}(S)$ is a good proxy for a minimum open-loop gain.



Minimum Loop Gain Goal is a constraint on the open-loop gain of the specified control loop. Thus, the loop gain is computed with the loop open at the specified location. To compute the gain with loop openings at other points in the control system, use the **Compute response with the following loops open** option in the **Open-Loop Response Selection** section of the dialog box.

Minimum Loop Gain Goal and Maximum Loop Gain Goal specify only low-gain or high-gain constraints in certain frequency bands. When you use these requirements, the software determines the best loop shape near crossover. When the loop shape near crossover is simple or well understood (such as integral action), you can use “Loop Shape Goal” on page 10-106 to specify that target loop shape.

Creation

In the **Tuning** tab of **Control System Tuner**, select **New Goal > Minimum gain for open-loop response** to create a Minimum Gain Goal.

Command-Line Equivalent

When tuning control systems at the command line, use `TuningGoal.MinLoopGain` to specify a minimum loop gain goal.

Open-Loop Response Selection

Use this section of the dialog box to specify the signal locations at which to compute the open-loop gain. You can also specify additional loop-opening locations for evaluating the tuning goal.

- **Shape open-loop response at the following locations**

  Select one or more signal locations in your model at which to compute and constrain the open-loop gain. To constrain a SISO response, select a single-valued location. For example, to constrain the open-loop gain at a location named `'y'`, click **Add signal to list** and select `'y'`. To constrain a MIMO response, select multiple signals or a vector-valued signal.
- **Compute response with the following loops open**

  Select one or more signal locations in your model at which to open a feedback loop for the purpose of evaluating this tuning goal. The tuning goal is evaluated against the open-loop configuration created by opening feedback loops at the locations you identify. For example, to evaluate the tuning goal with an opening at a location named `'x'`, click **Add signal to list** and select `'x'`.

Tip To highlight any selected signal in the Simulink model, click . To remove a signal from the input or output list, click . When you have selected multiple signals, you can reorder them using and . For more information on how to specify signal locations for a tuning goal, see “Specify Goals for Interactive Tuning” on page 10-29.

Desired Loop Gain

Use this section of the dialog box to specify the target minimum loop gain.

- **Pure integrator K/s**

  Check to specify a pure integrator shape for the target minimum loop gain. The software chooses the integrator constant, *K*, based on the values you specify for a target minimum gain and frequency. For example, to specify an integral gain profile with crossover frequency 10 rad/s, enter 1 in the **Choose K to keep gain above** text box. Then, enter 10 in the **at the frequency** text box. The software chooses the integrator constant such that the minimum loop gain is 1 at 10 rad/s.

- **Other gain profile**

  Check to specify the minimum gain profile as a function of frequency. Enter a SISO numeric LTI model whose magnitude represents the desired gain profile. For example, you can specify a smooth transfer function (`tf`, `zpk`, or `ss` model). Alternatively, you can sketch a piecewise target loop gain using an `frd` model. When you do so, the software automatically maps the profile to a smooth transfer function that approximates the desired minimum loop gain. For example, to specify minimum gain of 100 (40 dB) below 0.1 rad/s, rolling off at a rate of –20 dB/dec at higher frequencies, enter `frd([100 100 10],[0 1e-1 1])`.

  If you are tuning in discrete time, you can specify the minimum gain profile as a discrete-time model with the same sampling time as you use for tuning. If you specify the gain profile in continuous time, the tuning software discretizes it. Specifying the profile in discrete time gives you more control over the profile near the Nyquist frequency.

Options

Use this section of the dialog box to specify additional characteristics of the minimum loop gain goal.

- **Enforce goal in frequency range**

  Limit the enforcement of the tuning goal to a particular frequency band. Specify the frequency band as a row vector of the form `[min,max]`, expressed in frequency units of your model. For example, to create a tuning goal that applies only between 1 and 100 rad/s, enter `[1,100]`. By default, the tuning goal applies at all frequencies for continuous time, and up to the Nyquist frequency for discrete time.

- **Stabilize closed loop system**

  By default, the tuning goal imposes a stability requirement on the closed-loop transfer function from the specified inputs to outputs, in addition to the gain constraint. If stability is not required or cannot be achieved, select `No` to remove the stability requirement. For example, if the gain constraint applies to an unstable open-loop transfer function, select `No`.

- **Equalize loop interactions**

  For multi-loop or MIMO loop gain constraints, the feedback channels are automatically rescaled to equalize the off-diagonal (loop interaction) terms in the open-loop transfer function. Select `Off` to disable such scaling and shape the unscaled open-loop response.

- **Apply goal to**

  Use this option when tuning multiple models at once, such as an array of models obtained by linearizing a Simulink model at different operating points or block-parameter values. By default, active tuning goals are enforced for all models. To enforce a tuning requirement for a subset of models in an array, select **Only Models**. Then, enter the array indices of the models for which the goal is enforced. For example, suppose you want to apply the tuning goal to the second, third, and

fourth models in a model array. To restrict enforcement of the requirement, enter `2:4` in the **Only Models** text box.

For more information about tuning for multiple models, see “Robust Tuning Approaches” (Robust Control Toolbox).

Algorithms

Evaluating Tuning Goals

When you tune a control system, the software converts each tuning goal into a normalized scalar value $f(x)$. Here, x is the vector of free (tunable) parameters in the control system. The software then adjusts the parameter values to minimize $f(x)$ or to drive $f(x)$ below 1 if the tuning goal is a hard constraint.

For **Minimum Loop Gain Goal**, $f(x)$ is given by:

$$f(x) = \left\| W_S\left(D^{-1}SD\right)\right\|_\infty .$$

D is a diagonal scaling (for MIMO loops). S is the sensitivity function at `Location`. W_S is a frequency-weighting function derived from the minimum loop gain profile you specify. The gain of this function roughly matches the specified loop gain for values ranging from –20 dB to 60 dB. For numerical reasons, the weighting function levels off outside this range, unless the specified gain profile changes slope outside this range. This adjustment is called regularization. Because poles of W_S close to $s = 0$ or $s =$ `Inf` might lead to poor numeric conditioning for tuning, it is not recommended to specify gain profiles with very low-frequency or very high-frequency dynamics. For more information about regularization and its effects, see “Visualize Tuning Goals” on page 10-143.

Although S is a closed-loop transfer function, driving $f(x) < 1$ is equivalent to enforcing a lower bound on the open-loop transfer function, L, in a frequency band where the gain of L is greater than 1. To see why, note that $S = 1/(1 + L)$. For SISO loops, when $|L| >> 1$, $|S| \approx 1/|L|$. Therefore, enforcing the open-loop minimum gain requirement, $|L| > |W_S|$, is roughly equivalent to enforcing $|W_sS| < 1$. For MIMO loops, similar reasoning applies, with $||S|| \approx 1/\sigma_{\min}(L)$, where $\sigma_{\min}$ is the smallest singular value.

Implicit Constraints

This tuning goal imposes an implicit stability constraint on the closed-loop sensitivity function measured at the specified, evaluated with loops opened at the specified loop-opening locations. The dynamics affected by this implicit constraint are the stabilized dynamics for this tuning goal. The **Minimum decay rate** and **Maximum natural frequency** tuning options control the lower and upper bounds on these implicitly constrained dynamics. If the optimization fails to meet the default bounds, or if the default bounds conflict with other requirements, on the **Tuning** tab, use **Tuning Options** to change the defaults.

See Also

Related Examples

- “Specify Goals for Interactive Tuning” on page 10-29
- “Manage Tuning Goals” on page 10-135

- “Visualize Tuning Goals” on page 10-143

Maximum Loop Gain Goal

Purpose

Suppress gain of feedback loops at high frequency when using **Control System Tuner**.

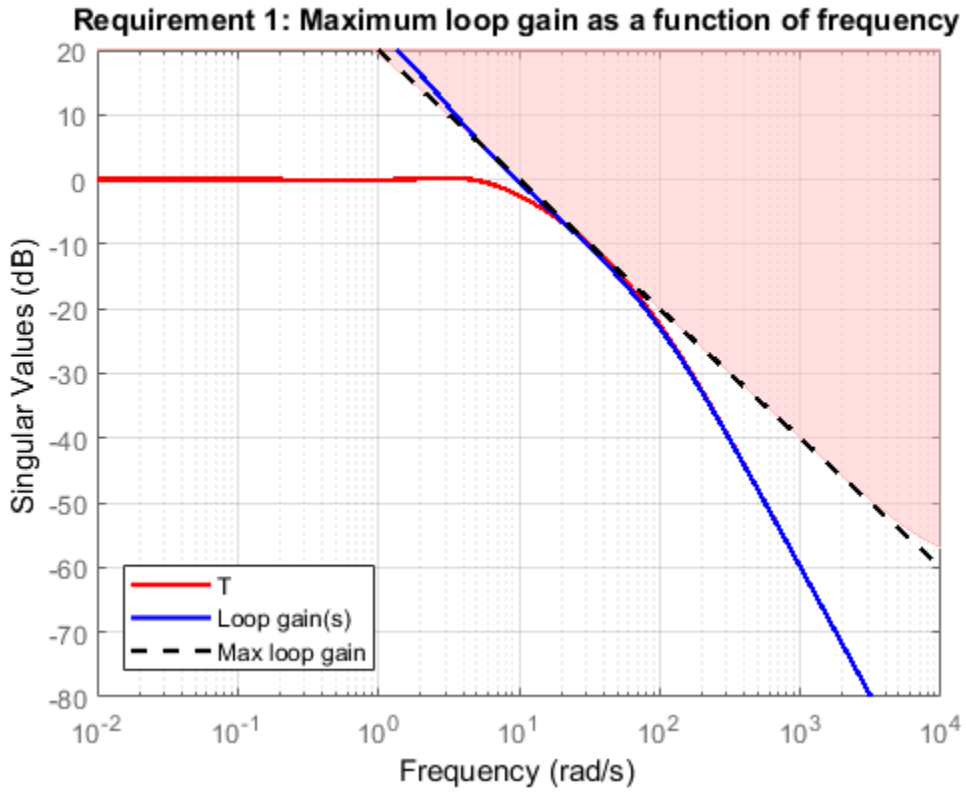
Description

Maximum Loop Gain Goal enforces a maximum loop gain in a particular frequency band. This tuning goal is useful, for example, for increasing system robustness to unmodeled dynamics.

Maximum Loop Gain Goal imposes a maximum gain on the open-loop frequency response (L) at a specified location in your control system. You specify the maximum open-loop gain as a function of frequency (a maximum gain profile). For MIMO feedback loops, the specified gain profile is interpreted as an upper bound on the largest singular value of L.

When you tune a control system, the maximum gain profile is converted to a maximum gain constraint on the complementary sensitivity function, $T = L/(I + L)$.

The following figure shows a typical specified maximum gain profile (dashed line) and a resulting tuned loop gain, L (blue line). The shaded region represents gain profile values that are forbidden by this requirement. The figure shows that when L is much smaller than 1, imposing a maximum gain on T is a good proxy for a maximum open-loop gain.



Maximum Loop Gain Goal is a constraint on the open-loop gain of the specified control loop. Thus, the loop gain is computed with the loop open at the specified location. To compute the gain with loop openings at other points in the control system, use the **Compute response with the following loops open** option in the **Open-Loop Response Selection** section of the dialog box.

Maximum Loop Gain Goal and Minimum Loop Gain Goal specify only high-gain or low-gain constraints in certain frequency bands. When you use these requirements, the software determines the best loop shape near crossover. When the loop shape near crossover is simple or well understood (such as integral action), you can use “Loop Shape Goal” on page 10-106 to specify that target loop shape.

Creation

In the **Tuning** tab of **Control System Tuner**, select **New Goal > Maximum gain for open-loop response** to create a Maximum Gain Goal.

Command-Line Equivalent

When tuning control systems at the command line, use `TuningGoal.MaxLoopGain` to specify a maximum loop gain goal.

Open-Loop Response Selection

Use this section of the dialog box to specify the signal locations at which to compute the open-loop gain. You can also specify additional loop-opening locations for evaluating the tuning goal.

- **Shape open-loop response at the following locations**

  Select one or more signal locations in your model at which to compute and constrain the open-loop gain. To constrain a SISO response, select a single-valued location. For example, to constrain the open-loop gain at a location named `'y'`, click **Add signal to list** and select `'y'`. To constrain a MIMO response, select multiple signals or a vector-valued signal.
- **Compute response with the following loops open**

  Select one or more signal locations in your model at which to open a feedback loop for the purpose of evaluating this tuning goal. The tuning goal is evaluated against the open-loop configuration created by opening feedback loops at the locations you identify. For example, to evaluate the tuning goal with an opening at a location named `'x'`, click **Add signal to list** and select `'x'`.

Tip To highlight any selected signal in the Simulink model, click . To remove a signal from the input or output list, click . When you have selected multiple signals, you can reorder them using and . For more information on how to specify signal locations for a tuning goal, see “Specify Goals for Interactive Tuning” on page 10-29.

Desired Loop Gain

Use this section of the dialog box to specify the target maximum loop gain.

- **Pure integrator K/s**

  Check to specify a pure integrator shape for the target maximum loop gain. The software chooses the integrator constant, *K*, based on the values you specify for a target maximum gain and frequency. For example, to specify an integral gain profile with crossover frequency 10 rad/s, enter 1 in the **Choose K to keep gain below** text box. Then, enter 10 in the **at the frequency** text box. The software chooses the integrator constant such that the maximum loop gain is 1 at 10 rad/s.

- **Other gain profile**

  Check to specify the maximum gain profile as a function of frequency. Enter a SISO numeric LTI model whose magnitude represents the desired gain profile. For example, you can specify a smooth transfer function (`tf`, `zpk`, or `ss` model). Alternatively, you can sketch a piecewise target loop gain using an `frd` model. When you do so, the software automatically maps the profile to a smooth transfer function that approximates the desired maximum loop gain. For example, to specify maximum gain of 100 (40 dB) below 0.1 rad/s, rolling off at a rate of –20 dB/dec at higher frequencies, enter `frd([100 100 10],[0 1e-1 1])`.

  If you are tuning in discrete time, you can specify the maximum gain profile as a discrete-time model with the same sampling time as you use for tuning. If you specify the gain profile in continuous time, the tuning software discretizes it. Specifying the profile in discrete time gives you more control over the profile near the Nyquist frequency.

Options

Use this section of the dialog box to specify additional characteristics of the maximum loop gain goal.

- **Enforce goal in frequency range**

  Limit the enforcement of the tuning goal to a particular frequency band. Specify the frequency band as a row vector of the form `[min,max]`, expressed in frequency units of your model. For example, to create a tuning goal that applies only between 1 and 100 rad/s, enter `[1,100]`. By default, the tuning goal applies at all frequencies for continuous time, and up to the Nyquist frequency for discrete time.

- **Stabilize closed loop system**

  By default, the tuning goal imposes a stability requirement on the closed-loop transfer function from the specified inputs to outputs, in addition to the gain constraint. If stability is not required or cannot be achieved, select `No` to remove the stability requirement. For example, if the gain constraint applies to an unstable open-loop transfer function, select `No`.

- **Equalize loop interactions**

  For multi-loop or MIMO loop gain constraints, the feedback channels are automatically rescaled to equalize the off-diagonal (loop interaction) terms in the open-loop transfer function. Select `Off` to disable such scaling and shape the unscaled open-loop response.

- **Apply goal to**

  Use this option when tuning multiple models at once, such as an array of models obtained by linearizing a Simulink model at different operating points or block-parameter values. By default, active tuning goals are enforced for all models. To enforce a tuning requirement for a subset of models in an array, select **Only Models**. Then, enter the array indices of the models for which the goal is enforced. For example, suppose you want to apply the tuning goal to the second, third, and

fourth models in a model array. To restrict enforcement of the requirement, enter `2:4` in the **Only Models** text box.

For more information about tuning for multiple models, see “Robust Tuning Approaches” (Robust Control Toolbox).

Algorithms

Evaluating Tuning Goals

When you tune a control system, the software converts each tuning goal into a normalized scalar value $f(x)$. Here, x is the vector of free (tunable) parameters in the control system. The software then adjusts the parameter values to minimize $f(x)$ or to drive $f(x)$ below 1 if the tuning goal is a hard constraint.

For **Maximum Loop Gain Goal**, $f(x)$ is given by:

$$f(x) = \left\| W_T\left(D^{-1}TD\right) \right\|_\infty .$$

Here, D is a diagonal scaling (for MIMO loops). T is the complementary sensitivity function at the specified location. W_T is a frequency-weighting function derived from the maximum loop gain profile you specify. The gain of this function roughly matches the inverse of the specified loop gain for values ranging from –60 dB to 20 dB. For numerical reasons, the weighting function levels off outside this range, unless the specified gain profile changes slope outside this range. This adjustment is called regularization. Because poles of W_T close to $s = 0$ or $s = \texttt{Inf}$ might lead to poor numeric conditioning for tuning, it is not recommended to specify gain profiles with very low-frequency or very high-frequency dynamics. For more information about regularization and its effects, see “Visualize Tuning Goals” on page 10-143.

Although T is a closed-loop transfer function, driving $f(x) < 1$ is equivalent to enforcing an upper bound on the open-loop transfer, L, in a frequency band where the gain of L is less than one. To see why, note that $T = L/(I + L)$. For SISO loops, when $|L| << 1$, $|T| \approx |L|$. Therefore, enforcing the open-loop maximum gain requirement, $|L| < 1/|W_T|$, is roughly equivalent to enforcing $|W_T T| < 1$. For MIMO loops, similar reasoning applies, with $||T|| \approx \sigma_{\max}(L)$, where $\sigma_{\max}$ is the largest singular value.

Implicit Constraints

This tuning goal imposes an implicit stability constraint on the closed-loop sensitivity function measured at the specified, evaluated with loops opened at the specified loop-opening locations. The dynamics affected by this implicit constraint are the stabilized dynamics for this tuning goal. The **Minimum decay rate** and **Maximum natural frequency** tuning options control the lower and upper bounds on these implicitly constrained dynamics. If the optimization fails to meet the default bounds, or if the default bounds conflict with other requirements, on the **Tuning** tab, use **Tuning Options** to change the defaults.

See Also

Related Examples

- “Specify Goals for Interactive Tuning” on page 10-29
- “Manage Tuning Goals” on page 10-135

- “Visualize Tuning Goals” on page 10-143

Loop Shape Goal

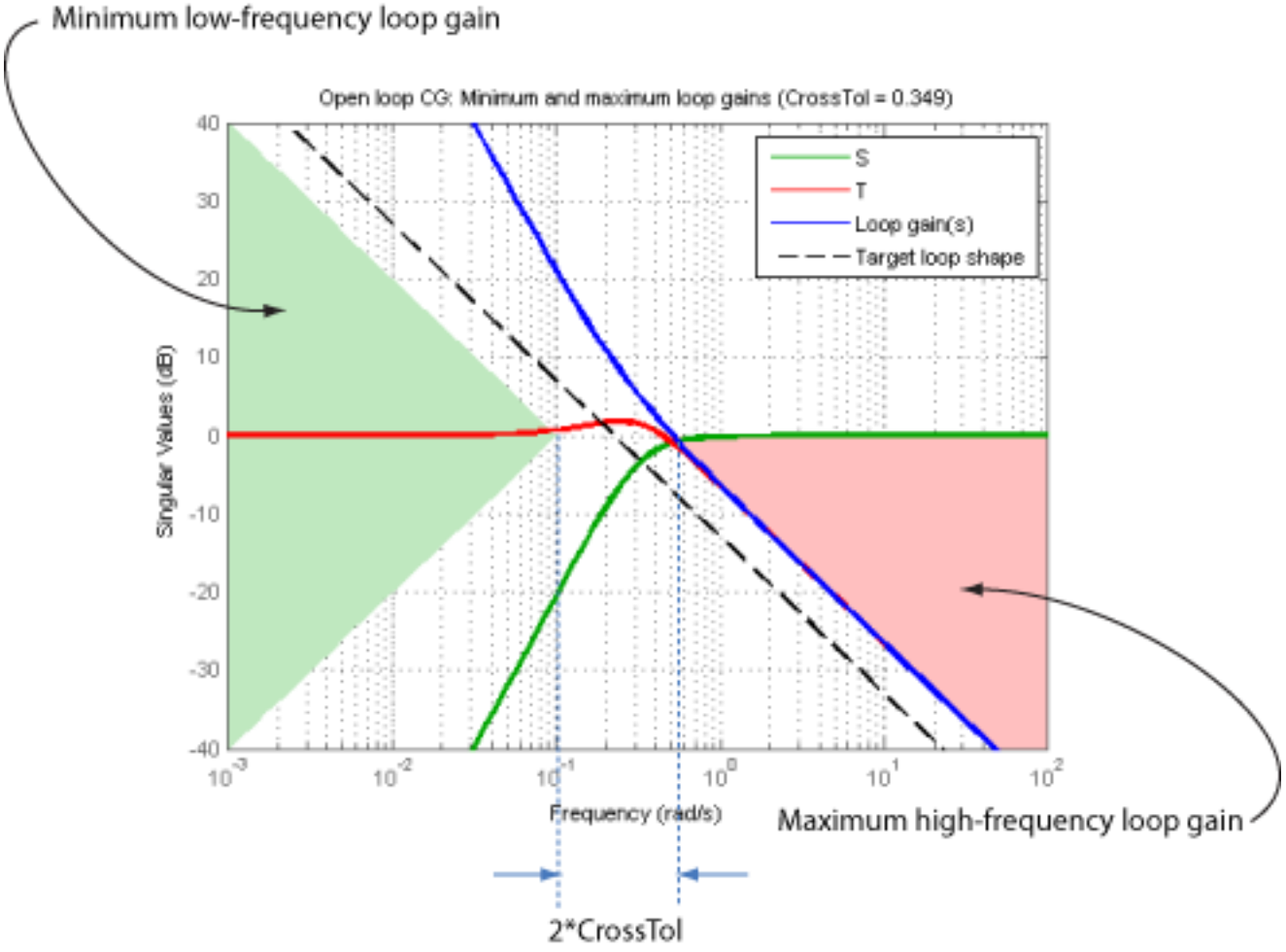
Purpose

Shape open-loop response of feedback loops when using **Control System Tuner**.

Description

Loop Shape Goal specifies a target gain profile (gain as a function of frequency) of an open-loop response. Loop Shape Goal constrains the open-loop, point-to-point response (L) at a specified location in your control system.

When you tune a control system, the target open-loop gain profile is converted into constraints on the inverse sensitivity function inv(S) = ($I + L$) and the complementary sensitivity function $T = 1–S$. These constraints are illustrated for a representative tuned system in the following figure.



Where L is much greater than 1, a minimum gain constraint on inv(S) (green shaded region) is equivalent to a minimum gain constraint on L. Similarly, where L is much smaller than 1, a maximum gain constraint on T (red shaded region) is equivalent to a maximum gain constraint on L. The gap

between these two constraints is twice the crossover tolerance, which specifies the frequency band where the loop gain can cross 0 dB.

For multi-input, multi-output (MIMO) control systems, values in the gain profile greater than 1 are interpreted as minimum performance requirements. Such values are lower bounds on the smallest singular value of the open-loop response. Gain profile values less than one are interpreted as minimum roll-off requirements, which are upper bounds on the largest singular value of the open-loop response. For more information about singular values, see `sigma`.

Use Loop Shape Goal when the loop shape near crossover is simple or well understood (such as integral action). To specify only high gain or low gain constraints in certain frequency bands, use “Minimum Loop Gain Goal” on page 10-96 or “Maximum Loop Gain Goal” on page 10-101. When you do so, the software determines the best loop shape near crossover.

Creation

In the **Tuning** tab of **Control System Tuner**, select **New Goal** > **Target shape for open-loop response** to create a Loop Shape Goal.

Command-Line Equivalent

When tuning control systems at the command line, use `TuningGoal.LoopShape` to specify a loop-shape goal.

Open-Loop Response Selection

Use this section of the dialog box to specify the signal locations at which to compute the open-loop gain. You can also specify additional loop-opening locations for evaluating the tuning goal.

- **Shape open-loop response at the following locations**

  Select one or more signal locations in your model at which to compute and constrain the open-loop gain. To constrain a SISO response, select a single-valued location. For example, to constrain the open-loop gain at a location named `'y'`, click **Add signal to list** and select `'y'`. To constrain a MIMO response, select multiple signals or a vector-valued signal.

- **Compute response with the following loops open**

  Select one or more signal locations in your model at which to open a feedback loop for the purpose of evaluating this tuning goal. The tuning goal is evaluated against the open-loop configuration created by opening feedback loops at the locations you identify. For example, to evaluate the tuning goal with an opening at a location named `'x'`, click **Add signal to list** and select `'x'`.

Tip To highlight any selected signal in the Simulink model, click . To remove a signal from the input or output list, click . When you have selected multiple signals, you can reorder them using and . For more information on how to specify signal locations for a tuning goal, see “Specify Goals for Interactive Tuning” on page 10-29.

Desired Loop Shape

Use this section of the dialog box to specify the target loop shape.

- **Pure integrator wc/s**

  Check to specify a pure integrator and crossover frequency for the target loop shape. For example, to specify an integral gain profile with crossover frequency 10 rad/s, enter 10 in the **Crossover frequency wc** text box.

- **Other gain profile**

  Check to specify the target loop shape as a function of frequency. Enter a SISO numeric LTI model whose magnitude represents the desired gain profile. For example, you can specify a smooth transfer function (`tf`, `zpk`, or `ss` model). Alternatively, you can sketch a piecewise target loop shape using an `frd` model. When you do so, the software automatically maps the profile to a smooth transfer function that approximates the desired loop shape. For example, to specify a target loop shape of 100 (40 dB) below 0.1 rad/s, rolling off at a rate of –20 dB/decade at higher frequencies, enter `frd([100 100 10],[0 1e-1 1])`.

  If you are tuning in discrete time, you can specify the loop shape as a discrete-time model with the same sample time that you are using for tuning. If you specify the loop shape in continuous time, the tuning software discretizes it. Specifying the loop shape in discrete time gives you more control over the loop shape near the Nyquist frequency.

Options

Use this section of the dialog box to specify additional characteristics of the loop shape goal.

- **Enforce loop shape within**

  Specify the tolerance in the location of the crossover frequency, in decades. For example, to allow gain crossovers within half a decade on either side of the target crossover frequency, enter 0.5. Increase the crossover tolerance to increase the ability of the tuning algorithm to enforce the target loop shape for all loops in a MIMO control system.

- **Enforce goal in frequency range**

  Limit the enforcement of the tuning goal to a particular frequency band. Specify the frequency band as a row vector of the form `[min,max]`, expressed in frequency units of your model. For example, to create a tuning goal that applies only between 1 and 100 rad/s, enter `[1,100]`. By default, the tuning goal applies at all frequencies for continuous time, and up to the Nyquist frequency for discrete time.

- **Stabilize closed loop system**

  By default, the tuning goal imposes a stability requirement on the closed-loop transfer function from the specified inputs to outputs, in addition to the gain constraint. If stability is not required or cannot be achieved, select `No` to remove the stability requirement. For example, if the gain constraint applies to an unstable open-loop transfer function, select `No`.

- **Equalize loop interactions**

  For multi-loop or MIMO loop gain constraints, the feedback channels are automatically rescaled to equalize the off-diagonal (loop interaction) terms in the open-loop transfer function. Select `Off` to disable such scaling and shape the unscaled open-loop response.

- **Apply goal to**

  Use this option when tuning multiple models at once, such as an array of models obtained by linearizing a Simulink model at different operating points or block-parameter values. By default, active tuning goals are enforced for all models. To enforce a tuning requirement for a subset of models in an array, select **Only Models**. Then, enter the array indices of the models for which the goal is enforced. For example, suppose you want to apply the tuning goal to the second, third, and fourth models in a model array. To restrict enforcement of the requirement, enter `2:4` in the **Only Models** text box.

  For more information about tuning for multiple models, see “Robust Tuning Approaches” (Robust Control Toolbox).

Algorithms

Evaluating Tuning Goals

When you tune a control system, the software converts each tuning goal into a normalized scalar value $f(x)$. Here, x is the vector of free (tunable) parameters in the control system. The software then adjusts the parameter values to minimize $f(x)$ or to drive $f(x)$ below 1 if the tuning goal is a hard constraint.

For **Loop Shape Goal**, $f(x)$ is given by:

$$f(x) = \left\| \begin{matrix} W_S S \\ W_T T \end{matrix} \right\|_\infty .$$

$S = D^{-1}[I - L(s,x)]^{-1}D$ is the scaled sensitivity function.

$L(s,x)$ is the open-loop response being shaped.

D is an automatically-computed loop scaling factor. (If **Equalize loop interactions** is set to `Off`, then $D = I$.)

$T = S - I$ is the complementary sensitivity function.

W_S and W_T are frequency weighting functions derived from the specified loop shape. The gains of these functions roughly match your specified loop shape and its inverse, respectively, for values ranging from –20 dB to 60 dB. For numerical reasons, the weighting functions level off outside this range, unless the specified gain profile changes slope outside this range. Because poles of W_S or W_T close to $s = 0$ or $s = \texttt{Inf}$ might lead to poor numeric conditioning for tuning, it is not recommended to specify loop shapes with very low-frequency or very high-frequency dynamics. For more information about regularization and its effects, see “Visualize Tuning Goals” on page 10-143.

Implicit Constraints

This tuning goal imposes an implicit stability constraint on the closed-loop sensitivity function measured at the specified, evaluated with loops opened at the specified loop-opening locations. The dynamics affected by this implicit constraint are the stabilized dynamics for this tuning goal. The **Minimum decay rate** and **Maximum natural frequency** tuning options control the lower and upper bounds on these implicitly constrained dynamics. If the optimization fails to meet the default bounds, or if the default bounds conflict with other requirements, on the **Tuning** tab, use **Tuning Options** to change the defaults.

See Also

Related Examples

- “Specify Goals for Interactive Tuning” on page 10-29
- “Manage Tuning Goals” on page 10-135
- “Visualize Tuning Goals” on page 10-143

Margins Goal

Purpose

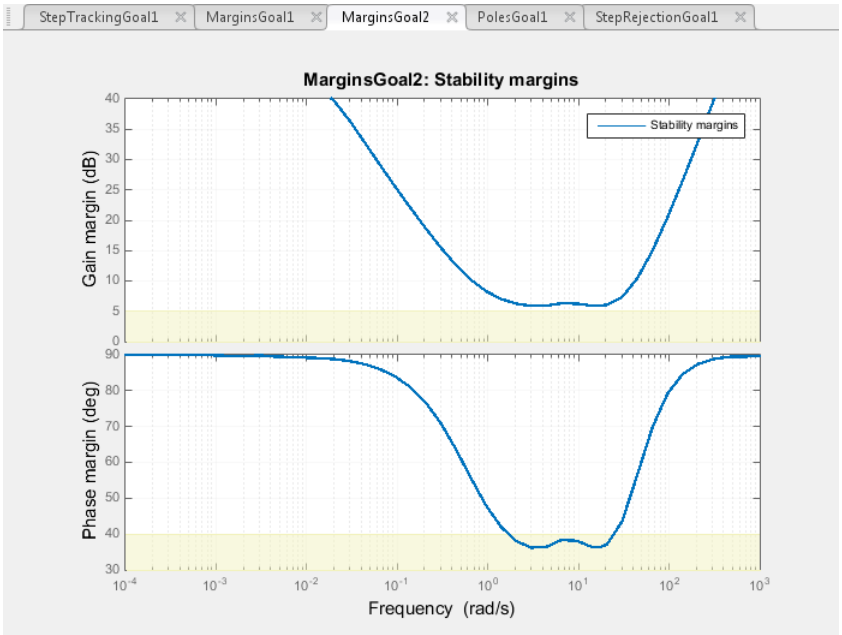
Enforce specified gain and phase margins when using **Control System Tuner**.

Description

Margins Goal uses the notion of disk margin to enforce specified gain and phase margins on SISO or MIMO feedback loops. Disk margins provide a more complete picture of robust stability as they take into account all frequency and loop interactions. Therefore, disk-based margins provide a stronger guarantee of stability than the classical gain and phase margins.

- For SISO feedback loops, the disk-based gain and phase margins are typically smaller but similar to the classical gain and phase margins.
- For MIMO feedback loops, the disk-based margins account for loop interactions and can be much smaller than classical loop-at-a-time gain and phase margins. The disk-based margins guarantee stability against gain or phase variations across all feedback channels. The gain or phase can change in all channels simultaneously, and by a different amount in each channel.

For information about disk margins, see “Stability Analysis Using Disk Margins” (Robust Control Toolbox).



In **Control System Tuner**, the shaded area on the plot represents the region in the frequency domain where the margins goal is not met. For more information about interpreting this plot, see “Stability Margins in Control System Tuning” on page 10-165.

Creation

In the **Tuning** tab of **Control System Tuner**, select **New Goal > Minimum stability margins** to create a Margins Goal.

Command-Line Equivalent

When tuning control systems at the command line, use `TuningGoal.Margins` to specify a stability margin goal.

Feedback Loop Selection

Use this section of the dialog box to specify the signal locations at which to measure stability margins. You can also specify additional loop-opening locations for evaluating the tuning goal.

- **Measure stability margins at the following locations**

  Select one or more signal locations in your model at which to compute and constrain the stability margins. To constrain a SISO loop, select a single-valued location. For example, to constrain the stability margins at a location named `'y'`, click **Add signal to list** and select `'y'`. To constrain a MIMO loop, select multiple signals or a vector-valued signal.

- **Measure stability margins with the following loops open**

  Select one or more signal locations in your model at which to open a feedback loop for the purpose of evaluating this tuning goal. The tuning goal is evaluated against the open-loop configuration created by opening feedback loops at the locations you identify. For example, to evaluate the tuning goal with an opening at a location named `'x'`, click **Add signal to list** and select `'x'`.

Tip To highlight any selected signal in the Simulink model, click . To remove a signal from the input or output list, click . When you have selected multiple signals, you can reorder them using and . For more information on how to specify signal locations for a tuning goal, see “Specify Goals for Interactive Tuning” on page 10-29.

Desired Margins

Use this section of the dialog box to specify the minimum gain and phase margins for the feedback loop.

- **Gain margin (dB)**

  Enter the required minimum gain margin for the feedback loop, specified as a scalar value in dB. The tuning goal uses disk-based gain and phase margins, which provide a stronger guarantee of stability than the classical gain and phase margins. (For details about disk margins, see “Stability Analysis Using Disk Margins” (Robust Control Toolbox).)

  The gain margin indicates how much the gain of the open-loop response can increase or decrease without loss of stability. For instance,

  - For a SISO system, entering `3` specifies a requirement that the closed-loop system remain stable for changes in the open-loop gain of up to ±3 dB.
  - For a MIMO system, entering `3` specifies a requirement that the closed-system remain stable for gain changes up to ±3 dB in each feedback channel. The gain can change in all channels simultaneously, and by a different amount in each channel.

- **Phase margin (degrees)**

  Required minimum phase margin for the feedback loop, specified as a scalar value in degrees. The tuning goal uses disk-based gain and phase margins, which provide a stronger guarantee of stability than the classical gain and phase margins. (For details about disk margins, see “Stability Analysis Using Disk Margins” (Robust Control Toolbox).)

  The phase margin indicates how much the phase of the open-loop response can increase or decrease without loss of stability. For instance,

  - For a SISO system, entering `45` specifies a requirement that the closed-loop system remain stable for changes of up to ±45° in the phase of the open-loop response.
  - For a MIMO system, entering `45` specifies a requirement that the closed-system remain stable for phase changes up to ±45° in each feedback channel. The phase can change in all channels simultaneously, and by a different amount in each channel.

Options

Use this section of the dialog box to specify additional characteristics of the stability margin goal.

- **Enforce goal in frequency range**

  Limit the enforcement of the tuning goal to a particular frequency band. Specify the frequency band as a row vector of the form `[min,max]`, expressed in frequency units of your model. For example, to create a tuning goal that applies only between 1 and 100 rad/s, enter `[1,100]`. By default, the tuning goal applies at all frequencies for continuous time, and up to the Nyquist frequency for discrete time.

  For best results with stability margin requirements, pick a frequency band extending about one decade on each side of the gain crossover frequencies.

- **D scaling order**

  This value controls the order (number of states) of the scalings involved in computing MIMO stability margins. Static scalings (scaling order 0) are used by default. Increasing the order may improve results at the expense of increased computations. If the stability margin plot shows a large gap between the optimized and actual margins, consider increasing the scaling order. See “Stability Margins in Control System Tuning” on page 10-165.

- **Apply goal to**

  Use this option when tuning multiple models at once, such as an array of models obtained by linearizing a Simulink model at different operating points or block-parameter values. By default, active tuning goals are enforced for all models. To enforce a tuning requirement for a subset of models in an array, select **Only Models**. Then, enter the array indices of the models for which the goal is enforced. For example, suppose you want to apply the tuning goal to the second, third, and fourth models in a model array. To restrict enforcement of the requirement, enter `2:4` in the **Only Models** text box.

  For more information about tuning for multiple models, see “Robust Tuning Approaches” (Robust Control Toolbox).

Algorithms

When you tune a control system, the software converts each tuning goal into a normalized scalar value $f(x)$. Here, x is the vector of free (tunable) parameters in the control system. The software then adjusts the parameter values to minimize $f(x)$ or to drive $f(x)$ below 1 if the tuning goal is a hard constraint.

For **Margins Goal**, $f(x)$ is given by:

$$f(x) = \|2\alpha S - \alpha I\|_\infty .$$

$S = D^{-1}[I - L(s,x)]^{-1}D$ is the scaled sensitivity function.

$L(s,x)$ is the open-loop response being shaped.

D is an automatically-computed loop scaling factor. For more information about D, see “Stability Margins in Control System Tuning” on page 10-165.

α is a scalar parameter computed from the specified gain and phase margin. For more information about α, see “Stability Analysis Using Disk Margins” (Robust Control Toolbox).

This tuning goal imposes an implicit stability constraint on the closed-loop sensitivity function measured at the specified, evaluated with loops opened at the specified loop-opening locations. The dynamics affected by this implicit constraint are the stabilized dynamics for this tuning goal. The **Minimum decay rate** and **Maximum natural frequency** tuning options control the lower and upper bounds on these implicitly constrained dynamics. If the optimization fails to meet the default bounds, or if the default bounds conflict with other requirements, on the **Tuning** tab, use **Tuning Options** to change the defaults.

See Also

Related Examples

- “Specify Goals for Interactive Tuning” on page 10-29
- “Manage Tuning Goals” on page 10-135
- “Visualize Tuning Goals” on page 10-143
- “Stability Margins in Control System Tuning” on page 10-165

Passivity Goal

Purpose

Enforce passivity of specific input/output map when using **Control System Tuner**.

Description

Passivity Goal enforces passivity of the response of the transfer function between the specified signal locations. A system is passive if all its I/O trajectories ($u(t)$,$y(t)$) satisfy:

$$\int_0^T y(t)^T u(t) dt > 0,$$

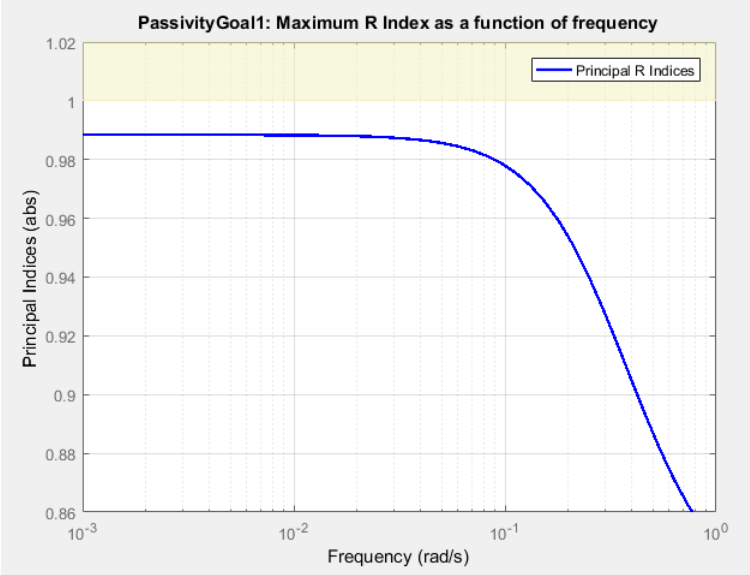
for all $T > 0$. Equivalently, a system is passive if its frequency response is positive real, which means that for all $\omega > 0$,

$$G(j\omega) + G(j\omega)^H > 0$$

Passivity Goal creates a constraint that enforces:

$$\int_0^T y(t)^T u(t) dt > \nu \int_0^T u(t)^T u(t) dt + \rho \int_0^T y(t)^T y(t) dt,$$

for all $T > 0$. To enforce the overall passivity condition, set the minimum input passivity index (ν) and the minimum output passivity index (ρ) to zero. To enforce an excess of passivity at the inputs or outputs, set ν or ρ to a positive value. To permit a shortage of passivity, set ν or ρ to a negative value. See “About Passivity and Passivity Indices” for more information about these indices.



In **Control System Tuner**, the shaded area on the plot represents the region in the frequency domain in which the tuning goal is not met. The plot shows the value of the index described in “Algorithms” on page 10-117.

Creation

In the **Tuning** tab of **Control System Tuner**, select **New Goal > Passivity Goal**.

Command-Line Equivalent

When tuning control systems at the command line, use `TuningGoal.Passivity` to specify a passivity constraint.

I/O Transfer Selection

Use this section of the dialog box to specify the inputs and outputs of the transfer function that the tuning goal constrains. Also specify any locations at which to open loops for evaluating the tuning goal.

- **Specify input signals**

  Select one or more signal locations in your model as inputs to the transfer function that the tuning goal constrains. To constrain a SISO response, select a single-valued input signal. For example, to constrain the gain from a location named `'u'` to a location named `'y'`, click **Add signal to list** and select `'u'`. To constrain the passivity of a MIMO response, select multiple signals or a vector-valued signal.

- **Specify output signals**

  Select one or more signal locations in your model as outputs of the transfer function that the tuning goal constrains. To constrain a SISO response, select a single-valued output signal. For example, to constrain the gain from a location named `'u'` to a location named `'y'`, click **Add signal to list** and select `'y'`. To constrain the passivity of a MIMO response, select multiple signals or a vector-valued signal.

- **Compute input/output gain with the following loops open**

  Select one or more signal locations in your model at which to open a feedback loop for the purpose of evaluating this tuning goal. The tuning goal is evaluated against the open-loop configuration created by opening feedback loops at the locations you identify. For example, to evaluate the tuning goal with an opening at a location named `'x'`, click **Add signal to list** and select `'x'`.

Tip To highlight any selected signal in the Simulink model, click . To remove a signal from the input or output list, click . When you have selected multiple signals, you can reorder them using and . For more information on how to specify signal locations for a tuning goal, see “Specify Goals for Interactive Tuning” on page 10-29.

Options

Use this section of the dialog box to specify additional characteristics of the passivity goal.

- **Minimum input passivity index**

Enter the target value of ν in the text box. To enforce an excess of passivity at the specified inputs, set $\nu > 0$. To permit a shortage of passivity, set $\nu < 0$. By default, the passivity goal enforces $\nu = 0$, passive at the inputs with no required excess of passivity.

- **Minimum output passivity index**

  Enter the target value of ρ in the text box. To enforce an excess of passivity at the specified outputs, set $\rho > 0$. To permit a shortage of passivity, set $\rho < 0$. By default, the passivity goal enforces $\rho = 0$, passive at the outputs with no required excess of passivity.

- **Enforce goal in frequency range**

  Limit the enforcement of the tuning goal to a particular frequency band. Specify the frequency band as a row vector of the form `[min,max]`, expressed in frequency units of your model. For example, to create a tuning goal that applies only between 1 and 100 rad/s, enter `[1,100]`. By default, the tuning goal applies at all frequencies for continuous time, and up to the Nyquist frequency for discrete time.

- **Apply goal to**

  Use this option when tuning multiple models at once, such as an array of models obtained by linearizing a Simulink model at different operating points or block-parameter values. By default, active tuning goals are enforced for all models. To enforce a tuning requirement for a subset of models in an array, select **Only Models**. Then, enter the array indices of the models for which the goal is enforced. For example, suppose you want to apply the tuning goal to the second, third, and fourth models in a model array. To restrict enforcement of the requirement, enter `2:4` in the **Only Models** text box.

  For more information about tuning for multiple models, see “Robust Tuning Approaches” (Robust Control Toolbox).

Algorithms

When you tune a control system, the software converts each tuning goal into a normalized scalar value $f(x)$. Here, x is the vector of free (tunable) parameters in the control system. The software then adjusts the parameter values to minimize $f(x)$ or to drive $f(x)$ below 1 if the tuning goal is a hard constraint.

For **Passivity Goal**, for a closed-loop transfer function $G(s,x)$ from the specified inputs to the specified outputs, $f(x)$ is given by:

$$f(x) = \frac{R}{1 + R/R_{\max}}, \quad R_{\max} = 10^6 .$$

R is the relative sector index (see `getSectorIndex`) of $[G(s,x); I]$, for the sector represented by:

$$Q = \begin{pmatrix} 2\rho & -I \\ -I & 2\nu \end{pmatrix},$$

where ρ is the minimum output passivity index and ν is the minimum input passivity index specified in the dialog box. $R_{\max}$ is fixed at 10_6, included to avoid numeric errors for very large R.

This tuning goal imposes an implicit minimum-phase constraint on the transfer function $G + I$. The transmission zeros of $G + I$ are the stabilized dynamics for this tuning goal. The **Minimum decay rate** and **Maximum natural frequency** tuning options control the lower and upper bounds on these

implicitly constrained dynamics. If the optimization fails to meet the default bounds, or if the default bounds conflict with other requirements, on the **Tuning** tab, use **Tuning Options** to change the defaults.

See Also

Related Examples

- “Specify Goals for Interactive Tuning” on page 10-29
- “Manage Tuning Goals” on page 10-135
- “Visualize Tuning Goals” on page 10-143
- “Passive Control of Water Tank Level”
- “About Passivity and Passivity Indices”

Conic Sector Goal

Purpose

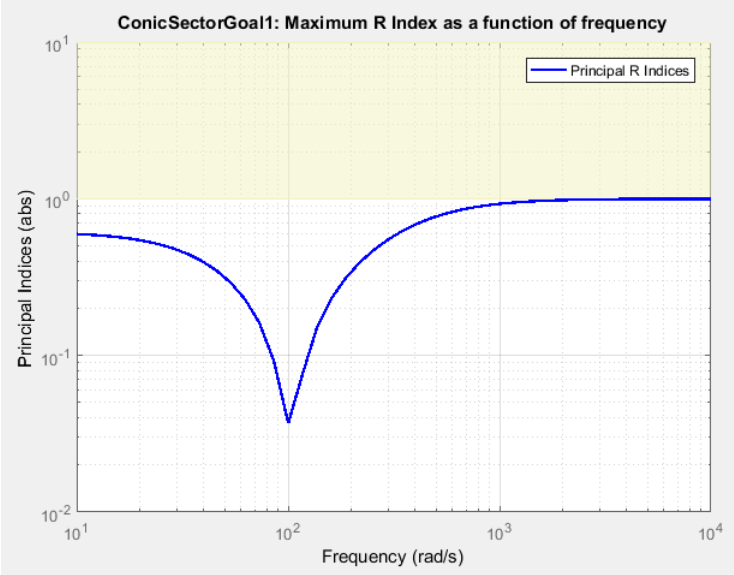
Enforce sector bound on specific input/output map when using **Control System Tuner**.

Description

Conic Sector Goal creates a constraint that restricts the output trajectories of a system. If for all nonzero input trajectories $u(t)$, the output trajectory $z(t) = (Hu)(t)$ of a linear system H satisfies:

$$\int_0^T z(t)^T Q\, z(t) dt < 0,$$

for all $T \geq 0$, then the output trajectories of H lie in the conic sector described by the symmetric indefinite matrix Q. Selecting different Q matrices imposes different conditions on the system response. When you create a Conic Sector Goal, you specify the input signals, output signals, and the sector geometry.



In **Control System Tuner**, the shaded area on the plot represents the region in the frequency domain in which the tuning goal is not met. The plot shows the value of the R-index described in "About Sector Bounds and Sector Indices".

Creation

In the **Tuning** tab of **Control System Tuner**, select **New Goal** > **Conic Sector Goal**.

Command-Line Equivalent

When tuning control systems at the command line, use `TuningGoal.ConicSector` to specify a step response goal.

I/O Transfer Selection

Use this section of the dialog box to specify the inputs and outputs of the transfer function that the tuning goal constrains. Also specify any locations at which to open loops for evaluating the tuning goal.

- **Specify input signals**

  Select one or more signal locations in your model as inputs to the transfer function that the tuning goal constrains. To constrain a SISO response, select a single-valued input signal. For example, to constrain the gain from a location named `'u'` to a location named `'y'`, click **Add signal to list** and select `'u'`. To constrain the passivity of a MIMO response, select multiple signals or a vector-valued signal.
- **Specify output signals**

  Select one or more signal locations in your model as outputs of the transfer function that the tuning goal constrains. To constrain a SISO response, select a single-valued output signal. For example, to constrain the gain from a location named `'u'` to a location named `'y'`, click **Add signal to list** and select `'y'`. To constrain the passivity of a MIMO response, select multiple signals or a vector-valued signal.
- **Compute input/output gain with the following loops open**

  Select one or more signal locations in your model at which to open a feedback loop for the purpose of evaluating this tuning goal. The tuning goal is evaluated against the open-loop configuration created by opening feedback loops at the locations you identify. For example, to evaluate the tuning goal with an opening at a location named `'x'`, click **Add signal to list** and select `'x'`.

Tip To highlight any selected signal in the Simulink model, click . To remove a signal from the input or output list, click . When you have selected multiple signals, you can reorder them using and . For more information on how to specify signal locations for a tuning goal, see “Specify Goals for Interactive Tuning” on page 10-29.

Options

Specify additional characteristics of the conic sector goal using this section of the dialog box.

- **Conic Sector Matrix**

  Enter the sector geometry Q, specified as:

  - A matrix, for constant sector geometry. `Q` is a symmetric square matrix that is `ny` on a side, where `ny` is the number of output signals you specify for the goal. The matrix `Q` must be indefinite to describe a well-defined conic sector. An indefinite matrix has both positive and negative eigenvalues. In particular, `Q` must have as many negative eigenvalues as there are input signals specified for the tuning goal (the size of the vector input signal $u(t)$).
  - An LTI model, for frequency-dependent sector geometry. `Q` satisfies $Q(s)' = Q(-s)$. In other words, $Q(s)$ evaluates to a Hermitian matrix at each frequency.

For more information, see “About Sector Bounds and Sector Indices”.

- **Regularization**

  Regularization parameter, specified as a real nonnegative scalar value. Regularization keeps the evaluation of the tuning goal numerically tractable when other tuning goals tend to make the sector bound ill-conditioned at some frequencies. When this condition occurs, set **Regularization** to a small (but not negligible) fraction of the typical norm of the feedthrough term in H. For example, if you anticipate the norm of the feedthrough term of H to be of order 1 during tuning, try setting **Regularization** to 0.001.

  For more information about the conditions that require regularization, see the `Regularization` property of `TuningGoal.ConicSector`.

- **Enforce goal in frequency range**

  Limit the enforcement of the tuning goal to a particular frequency band. Specify the frequency band as a row vector of the form `[min,max]`, expressed in frequency units of your model. For example, to create a tuning goal that applies only between 1 and 100 rad/s, enter `[1,100]`. By default, the tuning goal applies at all frequencies for continuous time, and up to the Nyquist frequency for discrete time.

- **Apply goal to**

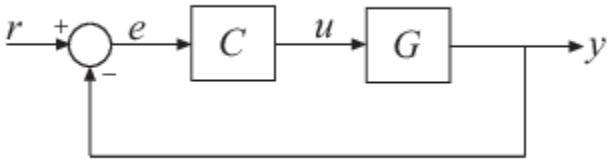
  Use this option when tuning multiple models at once, such as an array of models obtained by linearizing a Simulink model at different operating points or block-parameter values. By default, active tuning goals are enforced for all models. To enforce a tuning requirement for a subset of models in an array, select **Only Models**. Then, enter the array indices of the models for which the goal is enforced. For example, suppose you want to apply the tuning goal to the second, third, and fourth models in a model array. To restrict enforcement of the requirement, enter `2:4` in the **Only Models** text box.

  For more information about tuning for multiple models, see “Robust Tuning Approaches” (Robust Control Toolbox).

Tips

Constraining Input and Output Trajectories to Conic Sector

Consider the following control system.



Suppose that the signal u is marked as an analysis point in the model you are tuning. Suppose also that G is the closed-loop transfer function from u to y. A common application is to create a tuning goal that constrains all the I/O trajectories $\{u(t),y(t)\}$ of G to satisfy:

$$\int_0^T \begin{pmatrix} y(t) \\ u(t) \end{pmatrix}^T Q \begin{pmatrix} y(t) \\ u(t) \end{pmatrix} dt < 0,$$

for all $T \geq 0$. Constraining the I/O trajectories of G is equivalent to restricting the output trajectories $z(t)$ of the system $H = [G;I]$ to the sector defined by:

$$\int_0^T z(t)^T Q\, z(t)\, dt < 0.$$

(See “About Sector Bounds and Sector Indices” for more details about this equivalence.) To specify a constraint of this type using Conic Sector Goal, specify u as the input signal, and specify y and u as output signals. When you specify u as both input and output, Conic Sector Goal sets the corresponding transfer function to the identity. Therefore, the transfer function that the goal constrains is $H = [G;I]$ as intended. This treatment is specific to Conic Sector Goal. For other tuning goals, when the same signal appears in both inputs and outputs, the resulting transfer function is zero in the absence of feedback loops, or the complementary sensitivity at that location otherwise. This result occurs because when the software processes analysis points, it assumes that the input is injected after the output. See “Mark Signals of Interest for Control System Analysis and Design” on page 2-38 for more information about how analysis points work.

Algorithms

Let

$$Q = W_1 W_1^{\mathrm{T}} - W_2 W_2^{\mathrm{T}}$$

be an indefinite factorization of Q, where $W_1^{\mathrm{T}} W_2 = 0$. If $W_2^{\mathrm{T}} H(s)$ is square and minimum phase, then the time-domain sector bound

$$\int_0^T z(t)^T Q\, z(t) dt < 0,$$

is equivalent to the frequency-domain sector condition,

$$H(-j\omega) Q H(j\omega) \;<\; 0$$

for all frequencies. Conic Sector Goal uses this equivalence to convert the time-domain characterization into a frequency-domain condition that **Control System Tuner** can handle in the same way it handles gain constraints. To secure this equivalence, Conic Sector Goal also makes $W_2^{\mathrm{T}} H(s)$ minimum phase by making all its zeros stable. The transmission zeros affected by this minimum-phase condition are the stabilized dynamics for this tuning goal. The **Minimum decay rate** and **Maximum natural frequency** tuning options control the lower and upper bounds on these implicitly constrained dynamics. If the optimization fails to meet the default bounds, or if the default bounds conflict with other requirements, on the **Tuning** tab, use **Tuning Options** to change the defaults.

For sector bounds, the R-index plays the same role as the peak gain does for gain constraints (see “About Sector Bounds and Sector Indices”). The condition

$$H(-j\omega) Q H(j\omega) \;<\; 0$$

is satisfied at all frequencies if and only if the R-index is less than one. The plot that **Control System Tuner** displays for Conic Sector Goal shows the R-index value as a function of frequency (see `sectorplot`).

When you tune a control system, the software converts each tuning goal into a normalized scalar value $f(x)$, where x is the vector of free (tunable) parameters in the control system. The software then

adjusts the parameter values to minimize $f(x)$ or to drive $f(x)$ below 1 if the tuning goal is a hard constraint.

For Conic Sector Goal, for a closed-loop transfer function `H(s,x)` from the specified inputs to the specified outputs, $f(x)$ is given by:

$$f(x) = \frac{R}{1 + R/R_{\max}}, \quad R_{\max} = 10^6.$$

R is the relative sector index (see `getSectorIndex`) of `H(s,x)`, for the sector represented by `Q`.

See Also

Related Examples

- “About Sector Bounds and Sector Indices”
- “Specify Goals for Interactive Tuning” on page 10-29
- “Manage Tuning Goals” on page 10-135
- “Visualize Tuning Goals” on page 10-143

Weighted Passivity Goal

Purpose

Enforce passivity of a frequency-weighted transfer function when tuning in **Control System Tuner**.

Description

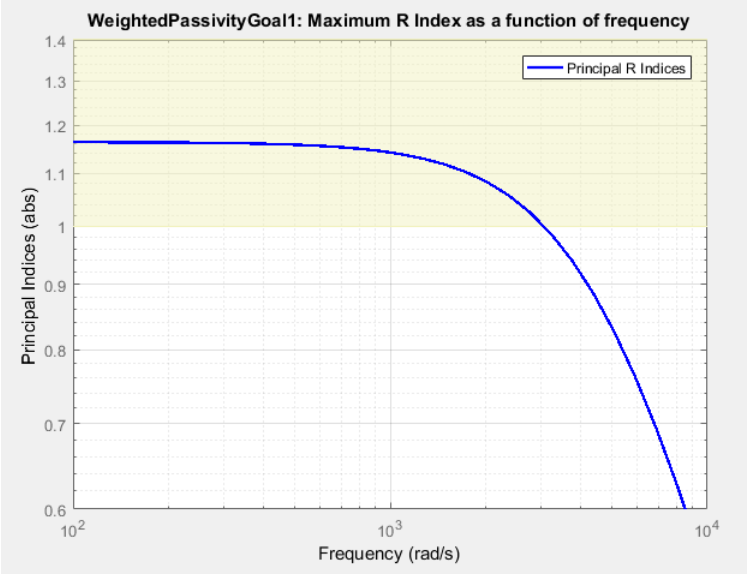
Weighted Passivity Goal enforces the passivity of $H(s) = W_L(s)T(s)W_R(s)$, where $T(s)$ is the transfer function from specified inputs to outputs. $W_L(s)$ and $W_R(s)$ are frequency weights used to emphasize particular frequency bands. A system is passive if all its I/O trajectories $(u(t),y(t))$ satisfy:

$$\int_0^T y(t)^T u(t)dt > 0,$$

for all $T > 0$. Weighted Passivity Goal creates a constraint that enforces:

$$\int_0^T y(t)^T u(t)dt > \nu \int_0^T u(t)^T u(t)dt + \rho \int_0^T y(t)^T y(t)dt,$$

for the trajectories of the weighted transfer function $H(s)$, for all $T > 0$. To enforce the overall passivity condition, set the minimum input passivity index (ν) and the minimum output passivity index (ρ) to zero. To enforce an excess of passivity at the inputs or outputs of the weighted transfer function, set ν or ρ to a positive value. To permit a shortage of passivity, set ν or ρ to a negative value. See `getPassiveIndex` for more information about these indices.



In **Control System Tuner**, the shaded area on the plot represents the region in the frequency domain in which the tuning goal is not met. The plot shows the value of the index described in “Algorithms” on page 10-126.

Creation

In the **Tuning** tab of **Control System Tuner**, select **New Goal** > **Weighted Passivity Goal**.

Command-Line Equivalent

When tuning control systems at the command line, use `TuningGoal.WeightedPassivity` to specify a step response goal.

I/O Transfer Selection

Use this section of the dialog box to specify the inputs and outputs of the transfer function that the tuning goal constrains. Also specify any locations at which to open loops for evaluating the tuning goal.

- **Specify input signals**

  Select one or more signal locations in your model as inputs to the transfer function that the tuning goal constrains. To constrain a SISO response, select a single-valued input signal. For example, to constrain the gain from a location named `'u'` to a location named `'y'`, click **Add signal to list** and select `'u'`. To constrain the passivity of a MIMO response, select multiple signals or a vector-valued signal.

- **Specify output signals**

  Select one or more signal locations in your model as outputs of the transfer function that the tuning goal constrains. To constrain a SISO response, select a single-valued output signal. For example, to constrain the gain from a location named `'u'` to a location named `'y'`, click **Add signal to list** and select `'y'`. To constrain the passivity of a MIMO response, select multiple signals or a vector-valued signal.

- **Compute input/output gain with the following loops open**

  Select one or more signal locations in your model at which to open a feedback loop for the purpose of evaluating this tuning goal. The tuning goal is evaluated against the open-loop configuration created by opening feedback loops at the locations you identify. For example, to evaluate the tuning goal with an opening at a location named `'x'`, click **Add signal to list** and select `'x'`.

Tip To highlight any selected signal in the Simulink model, click . To remove a signal from the input or output list, click . When you have selected multiple signals, you can reorder them using and . For more information on how to specify signal locations for a tuning goal, see “Specify Goals for Interactive Tuning” on page 10-29.

Weights

Use the **Left weight WL** and **Right weight WR** text boxes to specify the frequency-weighting functions for the tuning goal. $H(s) = W_L(s)T(s)W_R(s)$, where $T(s)$ is the transfer function from specified inputs to outputs.

W_L provides the weighting for the output channels of $H(s)$, and W_R provides the weighting for the input channels. You can specify scalar weights or frequency-dependent weighting. To specify a frequency-dependent weighting, use a numeric LTI model whose magnitude represents the desired

weighting function. For example, enter `tf(1,[1 0.01])` to specify a high weight at low frequencies that rolls off above 0.01 rad/s.

If the tuning goal constrains a MIMO transfer function, scalar or SISO weighting functions automatically expand to any input or output dimension. You can specify different weights for each channel by specifying matrices or MIMO weighting functions. The dimensions $H(s)$ must be commensurate with the dimensions of W_L and W_R. For example, if the constrained transfer function has two inputs, you can specify `diag([1 10])` as *WR*.

If you are tuning in discrete time, you can specify the weighting functions as discrete-time models with the same sampling time as you use for tuning. If you specify the weighting functions in continuous time, the tuning software discretizes them. Specifying the weighting functions in discrete time gives you more control over the weighting functions near the Nyquist frequency.

Options

Use this section of the dialog box to specify additional characteristics of the step response goal.

- **Minimum input passivity index**

  Enter the target value of ν in the text box. To enforce an excess of passivity at the specified inputs, set $\nu > 0$. To permit a shortage of passivity, set $\nu < 0$. By default, the passivity goal enforces $\nu = 0$, passive at the inputs with no required excess of passivity.
- **Minimum output passivity index**

  Enter the target value of ρ in the text box. To enforce an excess of passivity at the specified outputs, set $\rho > 0$. To permit a shortage of passivity, set $\rho < 0$. By default, the passivity goal enforces $\rho = 0$, passive at the outputs with no required excess of passivity.
- **Enforce goal in frequency range**

  Limit the enforcement of the tuning goal to a particular frequency band. Specify the frequency band as a row vector of the form `[min,max]`, expressed in frequency units of your model. For example, to create a tuning goal that applies only between 1 and 100 rad/s, enter `[1,100]`. By default, the tuning goal applies at all frequencies for continuous time, and up to the Nyquist frequency for discrete time.
- **Apply goal to**

  Use this option when tuning multiple models at once, such as an array of models obtained by linearizing a Simulink model at different operating points or block-parameter values. By default, active tuning goals are enforced for all models. To enforce a tuning requirement for a subset of models in an array, select **Only Models**. Then, enter the array indices of the models for which the goal is enforced. For example, suppose you want to apply the tuning goal to the second, third, and fourth models in a model array. To restrict enforcement of the requirement, enter `2:4` in the **Only Models** text box.

  For more information about tuning for multiple models, see “Robust Tuning Approaches” (Robust Control Toolbox).

Algorithms

When you tune a control system, the software converts each tuning goal into a normalized scalar value $f(x)$. Here, x is the vector of free (tunable) parameters in the control system. The software then

adjusts the parameter values to minimize $f(x)$ or to drive $f(x)$ below 1 if the tuning goal is a hard constraint.

For **Weighted Passivity Goal**, for a closed-loop transfer function $T(s,x)$ from the specified inputs to the specified outputs, and the weighted transfer function $H(s,x) = W_L(s)T(s,x)W_R(s)$, $f(x)$ is given by:

$$f(x) = \frac{R}{1 + R/R_{\max}}, \quad R_{\max} = 10^6.$$

R is the relative sector index (see `getSectorIndex`) of $[H(s,x); I]$, for the sector represented by:

$$Q = \begin{pmatrix} 2\rho & -I \\ -I & 2\nu \end{pmatrix},$$

where ρ is the minimum output passivity index and ν is the minimum input passivity index specified in the dialog box. $R_{\max}$ is fixed at 10_6, included to avoid numeric errors for very large R.

This tuning goal imposes an implicit minimum-phase constraint on the weighted transfer function $H + I$. The transmission zeros of $H + I$ are the stabilized dynamics for this tuning goal. The **Minimum decay rate** and **Maximum natural frequency** tuning options control the lower and upper bounds on these implicitly constrained dynamics. If the optimization fails to meet the default bounds, or if the default bounds conflict with other requirements, on the **Tuning** tab, use **Tuning Options** to change the defaults.

See Also

Related Examples

- “Specify Goals for Interactive Tuning” on page 10-29
- “Manage Tuning Goals” on page 10-135
- “Visualize Tuning Goals” on page 10-143
- “About Passivity and Passivity Indices”

Poles Goal

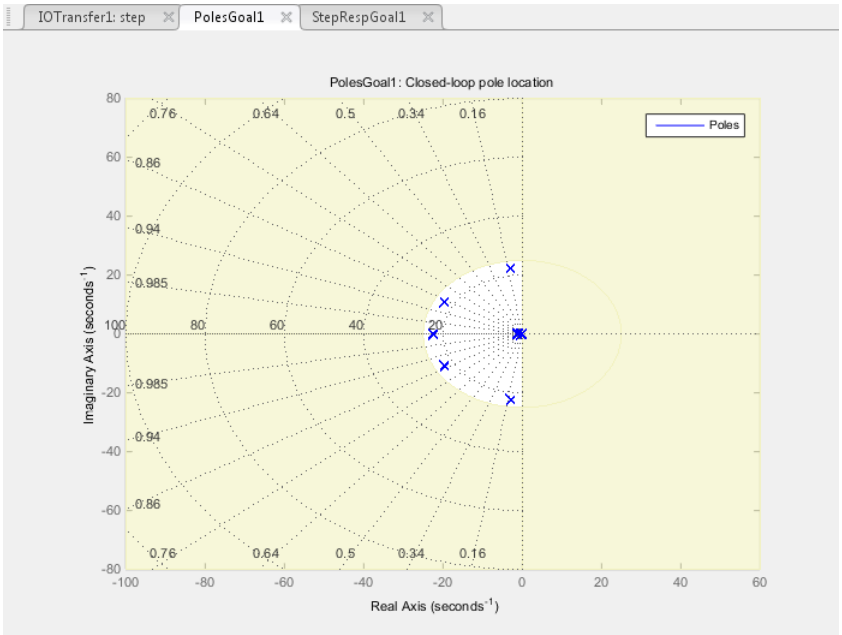
Purpose

Constrain the dynamics of the closed-loop system, specified feedback loops, or specified open-loop configurations, when using **Control System Tuner**.

Description

Poles Goal constrains the dynamics of your entire control system or of specified feedback loops of your control system. Constraining the dynamics of a feedback loop means constraining the dynamics of the sensitivity function measured at a specified location in the control system.

Using Poles Goal, you can specify finite minimum decay rate or minimum damping for the poles in the control system or specified loop. You can specify a maximum natural frequency for these poles, to eliminate fast dynamics in the tuned control system.



In **Control System Tuner**, the shaded area on the plot represents the region in the frequency domain where the pole location constraints are not met.

To constrain dynamics or ensure stability of a single tunable component of the control system, use “Controller Poles Goal” on page 10-132.

Creation

In the **Tuning** tab of **Control System Tuner**, select **New Goal > Constraint on closed-loop dynamics** to create a Poles Goal.

Command-Line Equivalent

When tuning control systems at the command line, use `TuningGoal.Poles` to specify a disturbance rejection goal.

Feedback Configuration

Use this section of the dialog box to specify the portion of the control system for which you want to constrain dynamics. You can also specify loop-opening locations for evaluating the tuning goal.

- **Entire system**

  Select this option to constrain the locations of closed-loop poles of the control system.

- **Specific feedback loop(s)**

  Select this option to specify one or more feedback loops to constrain. Specify a feedback loop by selecting a signal location in your control system. Poles Goal constrains the dynamics of the sensitivity function measured at that location. (See `getSensitivity` for information about sensitivity functions.)

  To constrain the dynamics of a SISO loop, select a single-valued location. For example, to constrain the dynamics of the sensitivity function measured at a location named `'y'`, click **Add signal to list** and select `'y'`. To constrain the dynamics of a MIMO loop, select multiple signals or a vector-valued signal.

- **Compute poles with the following loops open**

  Select one or more signal locations in your model at which to open a feedback loop for the purpose of evaluating this tuning goal. The tuning goal is evaluated against the open-loop configuration created by opening feedback loops at the locations you identify. For example, to evaluate the tuning goal with an opening at a location named `'x'`, click **Add signal to list** and select `'x'`.

Tip To highlight any selected signal in the Simulink model, click . To remove a signal from the input or output list, click . When you have selected multiple signals, you can reorder them using and . For more information on how to specify signal locations for a tuning goal, see “Specify Goals for Interactive Tuning” on page 10-29.

Pole Location

Use this section of the dialog box to specify the limits on pole locations.

- **Minimum decay rate**

  Enter the target minimum decay rate for the system poles. Closed-loop system poles that depend on the tunable parameters are constrained to satisfy `Re(s) < -MinDecay` for continuous-time systems, or `log(|z|) < -MinDecay*Ts` for discrete-time systems with sample time `Ts`. This constraint helps ensure stable dynamics in the tuned system.

  Enter 0 to impose no constraint on the decay rate.

- **Minimum damping**

  Enter the target minimum damping of closed-loop poles of tuned system, as a value between 0 and 1. Closed-loop system poles that depend on the tunable parameters are constrained to satisfy

`Re(s) < -MinDamping*|s|`. In discrete time, the damping ratio is computed using `s = log(z)/Ts`.

Enter 0 to impose no constraint on the damping ratio.

- **Maximum natural frequency**

  Enter the target maximum natural frequency of poles of tuned system, in the units of the control system model you are tuning. When you tune the control system using this requirement, closed-loop system poles that depend on the tunable parameters are constrained to satisfy `|s| < MaxFrequency` for continuous-time systems, or `|log(z)| < MaxFrequency*Ts` for discrete-time systems with sample time `Ts`. This constraint prevents fast dynamics in the control system.

  Enter `Inf` to impose no constraint on the natural frequency.

Options

Use this section of the dialog box to specify additional characteristics of the poles goal.

- **Enforce goal in frequency range**

  Limit the enforcement of the tuning goal to a particular frequency band. Specify the frequency band as a row vector of the form `[min,max]`, expressed in frequency units of your model. For example, to create a tuning goal that applies only between 1 and 100 rad/s, enter `[1,100]`. By default, the tuning goal applies at all frequencies for continuous time, and up to the Nyquist frequency for discrete time.

  The Poles Goal applies only to poles with natural frequency within the range you specify.

- **Apply goal to**

  Use this option when tuning multiple models at once, such as an array of models obtained by linearizing a Simulink model at different operating points or block-parameter values. By default, active tuning goals are enforced for all models. To enforce a tuning requirement for a subset of models in an array, select **Only Models**. Then, enter the array indices of the models for which the goal is enforced. For example, suppose you want to apply the tuning goal to the second, third, and fourth models in a model array. To restrict enforcement of the requirement, enter `2:4` in the **Only Models** text box.

  For more information about tuning for multiple models, see “Robust Tuning Approaches” (Robust Control Toolbox).

Algorithms

When you tune a control system, the software converts each tuning goal into a normalized scalar value $f(x)$. Here, x is the vector of free (tunable) parameters in the control system. The software then adjusts the parameter values to minimize $f(x)$ or to drive $f(x)$ below 1 if the tuning goal is a hard constraint.

For **Poles Goal**, $f(x)$ reflects the relative satisfaction or violation of the goal. For example, if your Poles Goal constrains the closed-loop poles of a feedback loop to a minimum damping of $\zeta = 0.5$, then:

- $f(x) = 1$ means the smallest damping among the constrained poles is $\zeta = 0.5$ exactly.

- $f(x) = 1.1$ means the smallest damping $\zeta = 0.5/1.1 = 0.45$, roughly 10% less than the target.
- $f(x) = 0.9$ means the smallest damping $\zeta = 0.5/0.9 = 0.55$, roughly 10% better than the target.

See Also

Related Examples

- “Specify Goals for Interactive Tuning” on page 10-29
- “Manage Tuning Goals” on page 10-135
- “Visualize Tuning Goals” on page 10-143

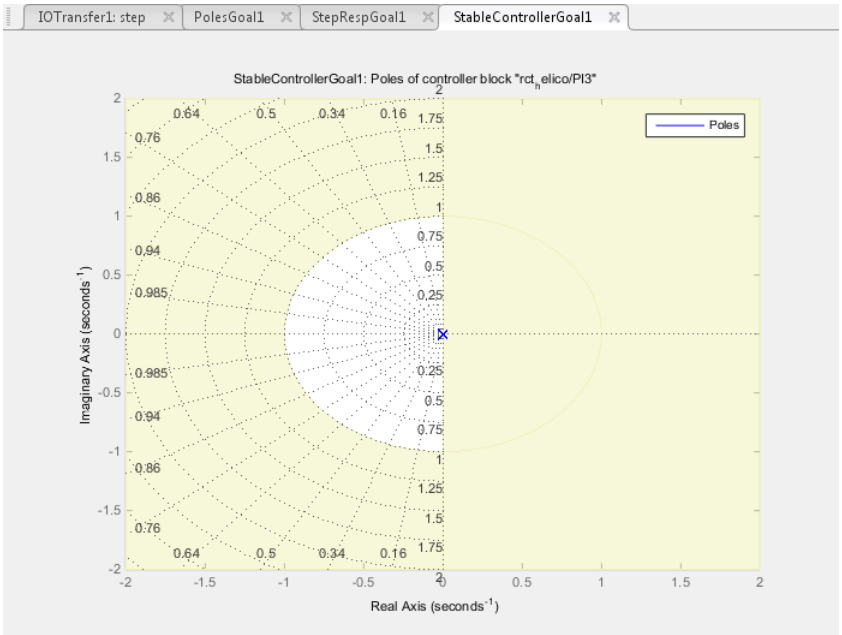
Controller Poles Goal

Purpose

Constrain the dynamics of a specified tunable block in the tuned control system, when using **Control System Tuner**.

Description

Controller Poles Goal constrains the dynamics of a tunable block in your control system model. Controller Poles Goal can impose a stability constraint on the specified block. You can also specify a finite minimum decay rate, a minimum damping rate, or a maximum natural frequency for the poles of the block. These constraints allow you to eliminate fast dynamics and control ringing in the response of the tunable block.



In **Control System Tuner**, the shaded area on the plot represents the region in the frequency domain where the pole location constraints are not met. The constraint applies to all poles in the block except fixed integrators, such as the I term of a PID controller.

To constrain dynamics or ensure stability of an entire control system or a feedback loop in the control system, use “Poles Goal” on page 10-128.

Creation

In the **Tuning** tab of **Control System Tuner**, select **New Goal > Constraint on controller dynamics** to create a Controller Poles Goal.

Command-Line Equivalent

When tuning control systems at the command line, use `TuningGoal.ControllerPoles` to specify a controller poles goal.

Constrain Dynamics of Tuned Block

From the drop-down menu, select the tuned block in your control system to which to apply the Controller Poles Goal.

If the block you want to constrain is not in the list, add it to the Tuned Blocks list. In **Control System Tuner**, in the **Tuning** tab, click **Select Blocks**. For more information about adding tuned blocks, see “Specify Blocks to Tune in Control System Tuner” on page 10-17.

Keep Poles Inside the Following Region

Use this section of the dialog box to specify the limits on pole locations.

- **Minimum decay rate**

  Enter the desired minimum decay rate for the poles of the tunable block. Poles of the block are constrained to satisfy `Re(s) < -MinDecay` for continuous-time blocks, or `log(|z|) < -MinDecay*Ts` for discrete-time blocks with sample time `Ts`.

  Specify a nonnegative value to ensure that the block is stable. If you specify a negative value, the tuned block can include unstable poles.

- **Minimum damping**

  Enter the desired minimum damping ratio of poles of the tunable block, as a value between 0 and 1. Poles of the block that depend on the tunable parameters are constrained to satisfy `Re(s) < -MinDamping*|s|`. In discrete time, the damping ratio is computed using `s=log(z)/Ts`.

- **Maximum natural frequency**

  Enter the target maximum natural frequency of poles of the tunable block, in the units of the control system model you are tuning. Poles of the block are constrained to satisfy `|s| < MaxFrequency` for continuous-time blocks, or `|log(z)| < MaxFrequency*Ts` for discrete-time blocks with sample time `Ts`. This constraint prevents fast dynamics in the tunable block.

Algorithms

When you tune a control system, the software converts each tuning goal into a normalized scalar value $f(x)$. Here, x is the vector of free (tunable) parameters in the control system. The software then adjusts the parameter values to minimize $f(x)$ or to drive $f(x)$ below 1 if the tuning goal is a hard constraint.

For **Controller Poles Goal**, $f(x)$ reflects the relative satisfaction or violation of the goal. For example, if your Controller Poles Goal constrains the pole of a tuned block to a minimum damping of $\zeta = 0.5$, then:

- $f(x) = 1$ means the damping of the pole is $\zeta = 0.5$ exactly.
- $f(x) = 1.1$ means the damping is $\zeta = 0.5/1.1 = 0.45$, roughly 10% less than the target.
- $f(x) = 0.9$ means the damping is $\zeta = 0.5/0.9 = 0.55$, roughly 10% better than the target.

See Also

Related Examples

- “Specify Goals for Interactive Tuning” on page 10-29
- “Manage Tuning Goals” on page 10-135
- “Visualize Tuning Goals” on page 10-143

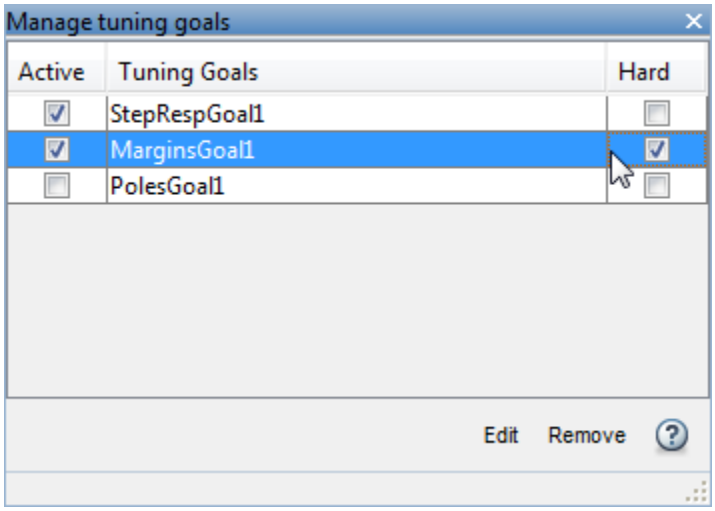
Manage Tuning Goals

Control System Tuner lets you designate one or more tuning goals as hard goals. This designation gives you a way to differentiate must-have goals from nice-to-have goals. **Control System Tuner** attempts to satisfy hard requirements by driving their associated cost functions below 1. Subject to that constraint, the software comes as close as possible to satisfying remaining (soft) requirements. For best results, make sure you can obtain a reasonable design with all goals treated as soft goals before attempting to enforce any goal as a hard constraint.

By default, new goals are designated soft goals. In the **Tuning** tab, click **Manage Goals** to open the **Manage tuning goals** dialog box. Check **Hard** for any goal to designate it a hard goal.

You can also designate any tuning goal as inactive for tuning. In this case the software ignores the tuning goal entirely. Use this dialog box to select which tuning goals are active when you tune the control system. **Active** is selected by default for any new goals. Clear **Active** for any design goal that you do not want enforced.

For example, if you tune with the following configuration, **Control System Tuner** optimizes `StepRespGoal1`, subject to `MarginsGoal1`. The tuning goal `PolesGoal1` is ignored.



All tuning goals you have created in the **Control System Tuner** session are listed in the dialog box. To edit an existing tuning goal, select it in the list and click **Edit**. To delete a tuning goal from the list, select it and click **Remove**.

To add more tuning goals to the list, in **Control System Tuner**, in the **Tuning** tab, click **New Goal**. For more information about creating tuning goals, see “Specify Goals for Interactive Tuning” on page 10-29.

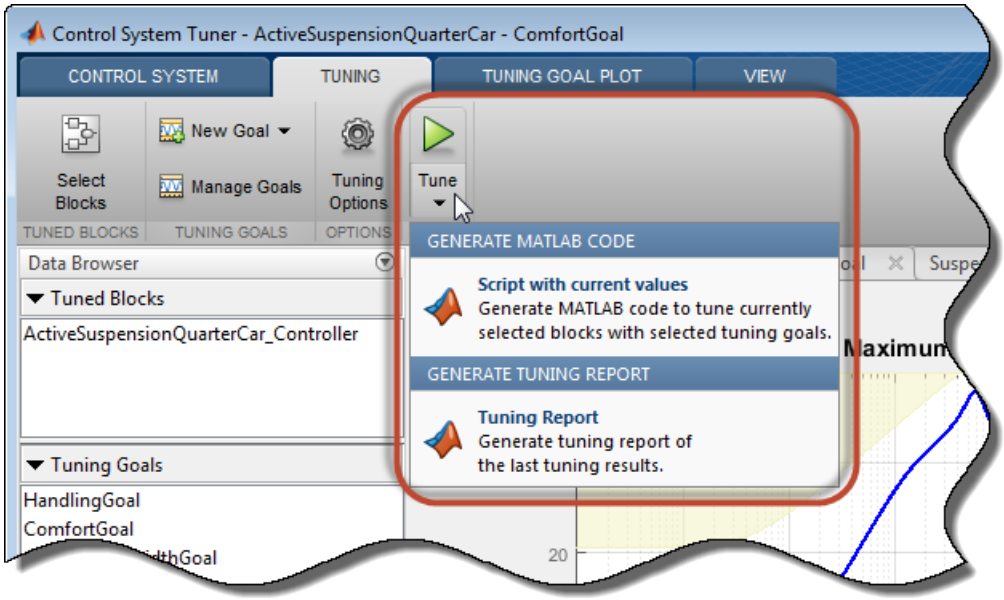
Generate MATLAB Code from Control System Tuner for Command-Line Tuning

You can generate a MATLAB script in **Control System Tuner** for tuning a control system at the command line. Generated scripts are useful when you want to programmatically reproduce a result you obtained interactively. A generated MATLAB script also enables you to programmatically perform multiple tuning operations with variations in tuning goals, system parameters, or model conditions such as operating point.

Tip You can also save a **Control System Tuner** session to reproduce within **Control System Tuner**.

To do so, in the **Control System** tab, click **Save Session**.

To generate a MATLAB script in **Control System Tuner**, in the **Tuning** tab, click **Tune**. Select **Script with current values**.



The MATLAB Editor displays the generated script, which script reproduces programmatically the current tuning configuration of Control System Tuner.

For example, suppose you generate a MATLAB script after completing all steps in the example “Control of a Linear Electric Actuator Using Control System Tuner”. The generated script computes the operating point used for tuning, designates the blocks to tune, creates the tuning goals, and performs other operations to reproduce the result at the command line.

The first section of the script creates the `slTuner` interface to the Simulink model (`rct_linact` in this example). The `slTuner` interface stores a linearization of the model and parameterizations of the blocks to tune.

```
%% Create system data with slTuner interface
TunedBlocks = {'rct_linact/Current Controller/Current PID'; ...
               'rct_linact/Speed Controller/Speed PID'};
AnalysisPoints = {'rct_linact/Speed Demand (rpm)/1'; ...
```

```
                'rct_linact/Current Sensor/1'; ...
                'rct_linact/Hall Effect Sensor/1'; ...
                'rct_linact/Speed Controller/Speed PID/1'; ...
                'rct_linact/Current Controller/Current PID/1'};
OperatingPoints = 0.5;
% Specify the custom options
Options = slTunerOptions('AreParamsTunable',false);
% Create the slTuner object
CL0 = slTuner('rct_linact',TunedBlocks,AnalysisPoints,OperatingPoints,Options);
```

The `slTuner` interface also specifies the operating point at which the model is linearized, and marks as analysis points all the signal locations required to specify the tuning goals for the example. (See “Create and Configure slTuner Interface to Simulink Model” on page 10-161.)

If you are tuning a control system modeled in MATLAB instead of Simulink, the first section of the script constructs a `genss` model that has equivalent dynamics and parameterization to the `genss` model of the control system that you specified **Control System Tuner**.

Next, the script creates the three tuning goals specified in the example. The script uses `TuningGoal` objects to capture these tuning goals. For instance, the script uses `TuningGoal.Tracking` to capture the Tracking Goal of the example.

```
%% Create tuning goal to follow reference commands with prescribed performance
% Inputs and outputs
Inputs = {'rct_linact/Speed Demand (rpm)/1'};
Outputs = {'rct_linact/Hall Effect Sensor/1[rpm]'};
% Tuning goal specifications
ResponseTime = 0.1; % Approximately reciprocal of tracking bandwidth
DCError = 0.001; % Maximum steady-state error
PeakError = 1; % Peak error across frequency
% Create tuning goal for tracking
TR = TuningGoal.Tracking(Inputs,Outputs,ResponseTime,DCError,PeakError);
TR.Name = 'TR'; % Tuning goal name
```

After creating the tuning goals, the script sets any algorithm options you had set in **Control System Tuner**. The script also designates tuning goals as soft or hard goals, according to the configuration of tuning goals in **Control System Tuner**. (See “Manage Tuning Goals” on page 10-135.)

```
%% Create option set for systune command
Options = systuneOptions();

%% Set soft and hard goals
SoftGoals = [ TR ; ...
              MG1 ; ...
              MG2 ];
HardGoals = [];
```

In this example, all the goals are designated as soft goals when the script is generated. Therefore, `HardGoals` is empty.

Finally, the script tunes the control system by calling `systune` on the `slTuner` interface using the tuning goals and options.

```
%% Tune the parameters with soft and hard goals
[CL1,fSoft,gHard,Info] = systune(CL0,SoftGoals,HardGoals,Options);
```

The script also includes an optional call to `viewGoal`, which displays graphical representations of the tuning goals to aid you in interpreting and validating the tuning results. Uncomment this line of code to generate the plots.

```
%% View tuning results
% viewGoal([SoftGoals;HardGoals],CL1);
```

You can add calls to functions such `getIOTransfer` to make the script generate additional analysis plots.

See Also

Related Examples

- “Create and Configure slTuner Interface to Simulink Model” on page 10-161
- “Tune Control System at the Command Line” on page 10-170
- “Validate Tuned Control System” on page 10-172

Interpret Numeric Tuning Results

When you tune a control system with `systune` or **Control System Tuner**, the software provides reports that give you an overview of how well the tuned control system meets your design requirements. Interpreting these reports requires understanding how the tuning algorithm optimizes the system to satisfy your tuning goals. (The software also provides visualizations of the tuning goals and system responses to help you see where and by how much your requirements are not satisfied. For information about using these plots, see “Visualize Tuning Goals” on page 10-143.)

Tuning-Goal Scalar Values

The tuning software converts each tuning goal into a normalized scalar value which it then constrains (hard goals) or minimizes (soft goals). Let $f_i(x)$ and $g_j(x)$ denote the scalar values of the soft and hard goals, respectively. Here, x is the vector of tunable parameters in the control system to tune. The tuning algorithm solves the minimization problem:

Minimize $\max_i f_i(x)$ subject to $\max_j g_j(x) < 1$, for $x_{\min} < x < x_{\max}$.

x_{min} and x_{max} are the minimum and maximum values of the free parameters of the control system. (For information about the specific functions used to evaluate each type of requirement, see the reference pages for each tuning goal.)

When you use both soft and hard tuning goals, the software solves the optimization as a sequence of subproblems of the form:

$$\min_x \max(\alpha f(x), g(x)) .$$

The software adjusts the multiplier α so that the solution of the subproblems converges to the solution of the original constrained optimization problem.

The tuning software reports the final scalar values for each tuning goal. When the final value of $f_i(x)$ or $g_j(x)$ is less than 1, the corresponding tuning goal is satisfied. Values greater than 1 indicate that the tuning goal is not satisfied for at least some conditions. For instance, a tuning goal that describes a frequency-domain constraint might be satisfied at some frequencies and not at others. The closer the value is to 1, the closer the tuning goal is to being satisfied. Thus these values give you an overview of how successfully the tuned system meets your requirements.

The form in which the software presents the optimized tuning-goal values depends on whether you are tuning with **Control System Tuner** or at the command line.

Tuning Results at the Command Line

The `systune` command returns the control system model or `slTuner` interface with the tuned parameter values. `systune` also returns the best achieved values of each $f_i(x)$ and $g_j(x)$ as the vector-valued output arguments `fSoft` and `gHard`, respectively. See the `systune` reference page for more information. (To obtain the final tuning goal values on their own, use `evalGoal`.)

By default, `systune` displays the best achieved final values of the tuning goals in the command window. For instance, in the example “PID Tuning for Setpoint Tracking vs. Disturbance Rejection”, `systune` is called with one soft requirement, `R1`, and two hard requirements `R2` and `R3`.

```
T1 = systune(T0,R1,[R2 R3]);
```

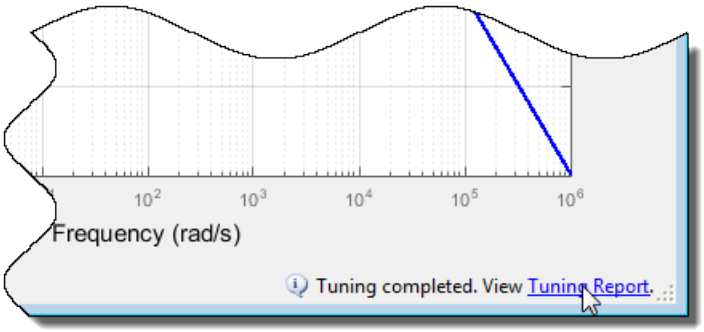
```
Final: Soft = 1.12, Hard = 0.99988, Iterations = 143
```

This display indicates that the largest optimized value of the hard tuning goals is less than 1, so both hard goals are satisfied. The soft goal value is slightly greater than one, indicating that the soft goal is nearly satisfied. You can use tuning-goal plots to see in what regimes and by how much the tuning goals are violated. (See “Visualize Tuning Goals” on page 10-143.)

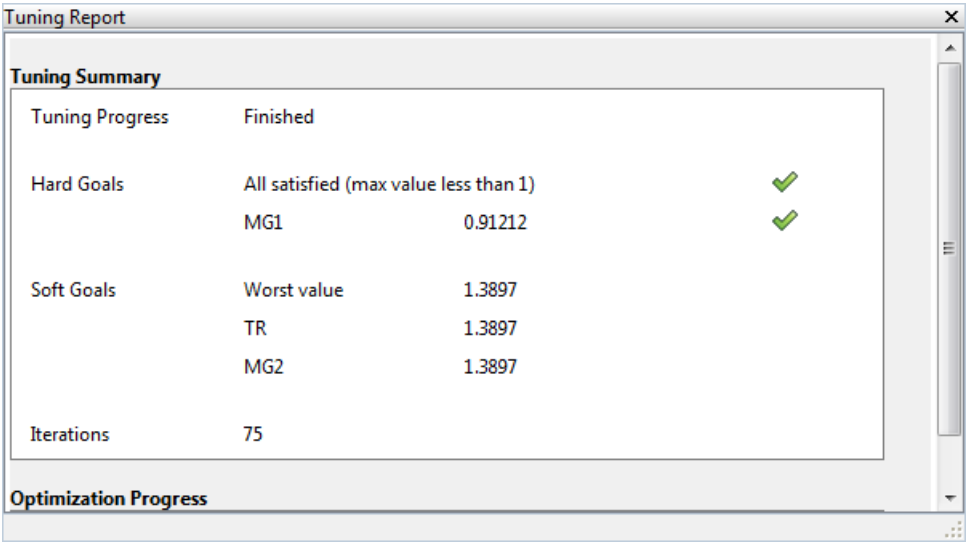
You can obtain additional information about the optimization progress and values using the `info` output of `systune`. To make `systune` display additional information during tuning, use `systuneOptions`.

Tuning Results in Control System Tuner

In **Control System Tuner**, when you click , the app compiles a Tuning Report summarizing the best achieved values of $f_i(x)$ and $g_j(x)$. To view the tuning report immediately after tuning a control system, click **Tuning Report** at the bottom-right corner of **Control System Tuner**.



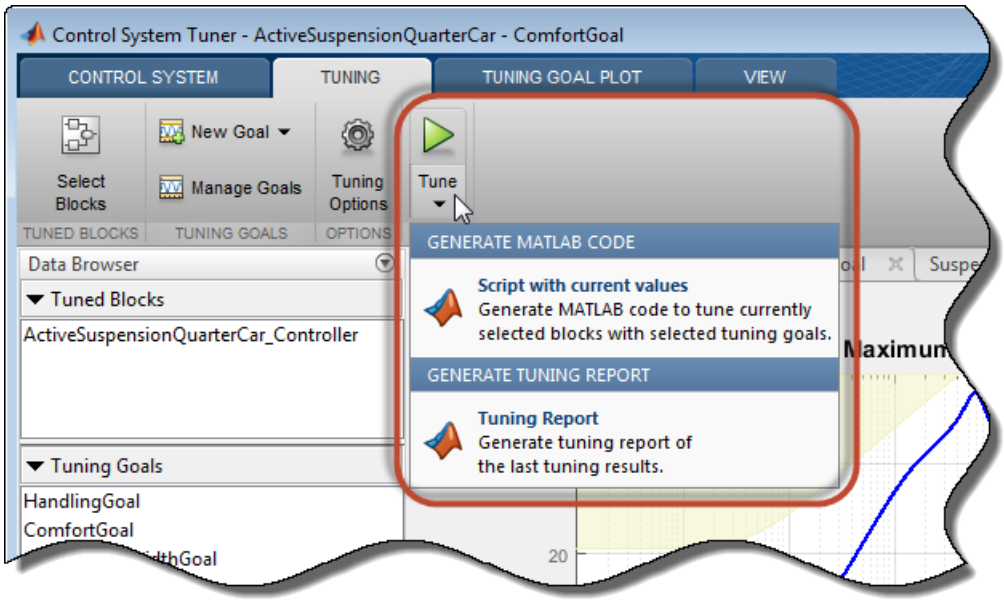
The tuning report displays the final $f_i(x)$ and $g_j(x)$ values obtained by the algorithm.



The **Hard Goals** area shows the minimized $g_i(x)$ values and indicates which are satisfied. The **Soft Goals** area highlights the largest of the minimized $f_i(x)$ values as **Worst Value**, and lists the values for all the requirements. In this example, the hard goal is satisfied, while the soft goals are nearly

satisfied. As in the command-line case, you can use tuning-goal plots to see where and by how much tuning goals are violated. (See “Visualize Tuning Goals” on page 10-143.)

Tip You can view a report from the most recent tuning run at any time. In the **Tuning** tab, click **Tune**, and select `Tuning Report`.



Improve Tuning Results

If the tuning results do not adequately meet your design requirements, adjust your set of tuning goals to improve the results. For example:

- Designate tuning goals that are must-have requirements as hard goals. Or, relax tuning goals that are not absolute requirements by designating them as soft goals.
- Limit the frequency range in which frequency-domain goals are enforced.
  - In **Control System Tuner**, use the **Enforce goal in frequency range** field of the tuning goal dialog box.
  - At the command line, use the `Focus` property of the `TuningGoal` object.

If the tuning results do satisfy your design requirements, you can validate the tuned control system as described in “Validate Tuned Control System” on page 10-172.

See Also

`systune` | `systune (for slTuner)` | `viewGoal` | `evalGoal`

Related Examples

- “Visualize Tuning Goals” on page 10-143

- “Validate Tuned Control System” on page 10-172

Visualize Tuning Goals

When you tune a control system with `systune` or **Control System Tuner**, use tuning-goal plots to visualize your design requirements against the tuned control system responses. Tuning-goal plots show graphically where and by how much tuning goals are satisfied or violated. This visualization lets you examine how close your control system is to ideal performance. It can also help you identify problems with tuning and provide clues on how to improve your design.

Tuning-Goal Plots

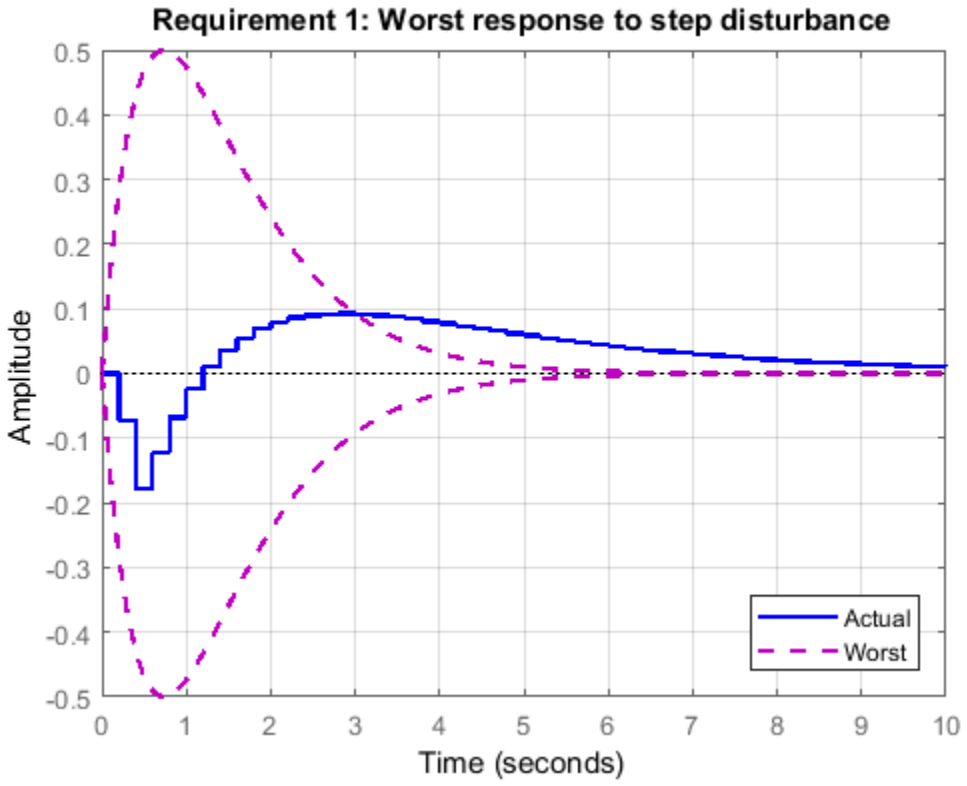
How you obtain tuning-goal plots depends on your work environment.

- At the command line, use `viewGoal`.
- In **Control System Tuner**, each tuning goal that you create generates a tuning-goal plot. When you tune the control system, these plots update to reflect the tuned design.

The form of the tuning-goal plot depends on the specific tuning goal you use.

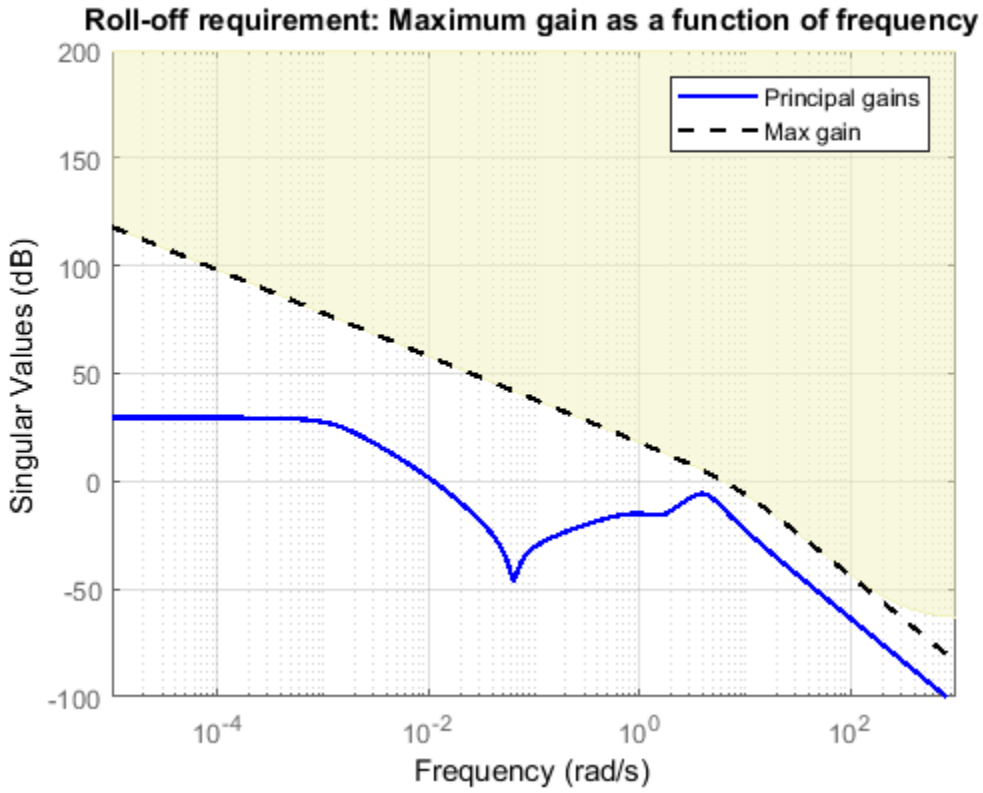
Time-Domain Goals

For time-domain tuning goals, the tuning-goal plot is a time-domain plot of the relevant system response. The following plot, adapted from the example “MIMO Control of Diesel Engine”, shows a typical tuning-goal plot for a time-domain disturbance-rejection goal. The dashed lines represent the worst acceptable step response specified in the tuning goal. The solid line shows the corresponding response of the tuned system.



Frequency-Domain Goals

The plots for frequency-domain tuning goals show the target response and the tuned response in the frequency domain. The following plot, adapted from the example “Fixed-Structure Autopilot for a Passenger Jet”, shows a plot for a gain goal (`TuningGoal.Gain` at the command line). This tuning goal limits the gain between a specified input and output to a frequency-dependent profile. In the plot, the dashed line shows the gain profile specified in the tuning goal. If the tuned system response (solid line) enters the shaded region, the tuning goal is violated. In this case, the tuning goal is satisfied at all frequencies.



Margin Goals

For information about interpreting tuning-goal plots for stability-margin goals, see “Stability Margins in Control System Tuning” on page 10-165.

Difference Between Dashed Line and Shaded Region

With some frequency-domain tuning goals, there might be a difference between the gain profile you specify in the tuning goal, and the profile the software uses for tuning. In this case, the shaded region of the plot reflects the profile that the software uses for tuning. The gain profile you specify and the gain profile used for tuning might differ if:

- You tune a control system in discrete time, but specify the gain profile in continuous time.
- The software modifies the asymptotes of the specified gain profile to improve numeric stability.

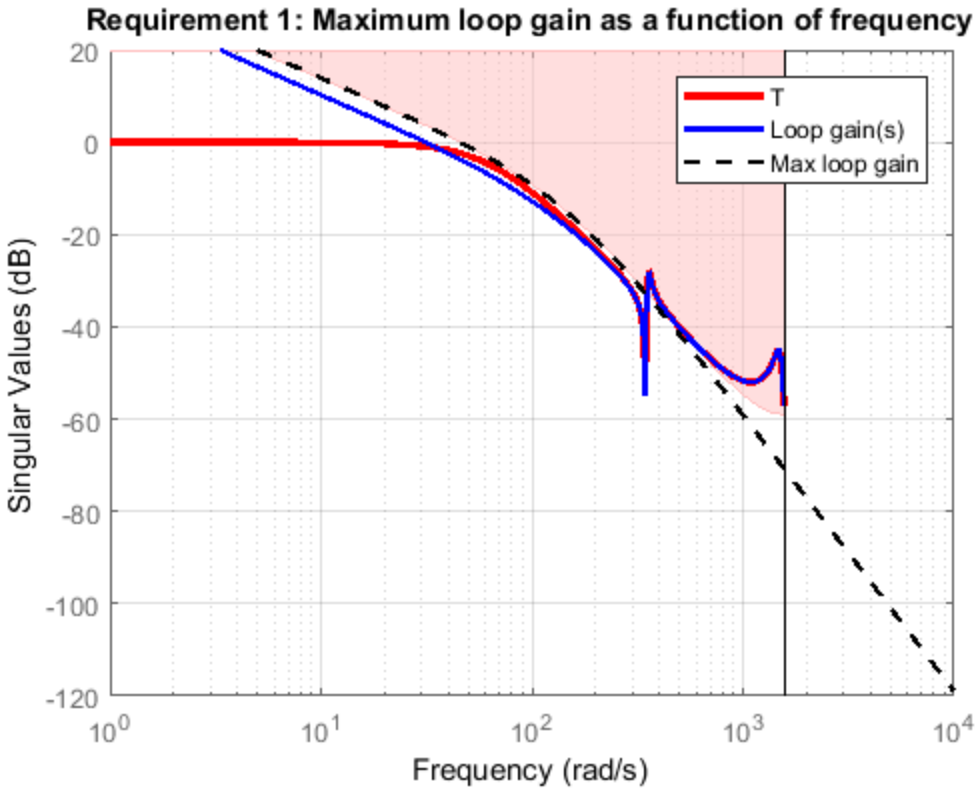
Continuous-Time Gain Profile for Discrete-Time Tuning

When you tune a discrete-time control system, you can specify frequency-dependent tuning goals using discrete-time or continuous-time transfer functions. If you use a continuous-time transfer function, the tuning algorithm discretizes the transfer function before tuning. For instance, suppose that you specify a tuning goal as follows.

```
W = zpk([],[0 -150 -150],1125000);
Req = TuningGoal.MaxLoopGain('Xloc',W);
```

Suppose further that you use the tuning goal with `systune` to tune a discrete-time `genss` model or `slTuner` interface. `CL` is the resulting tuned control system. To examine the result, generate a tuning-goal plot.

```
viewGoal(Req,CL)
```



The plot shows `W`, the continuous-time maximum loop gain that you specified, as a dashed line. The shaded region shows the discretized version of `W` that `systune` uses for tuning. The discretized maximum loop gain cuts off at the Nyquist frequency corresponding to the sample time of `CL`. Near that cutoff, the shaded region diverges from the dashed line.

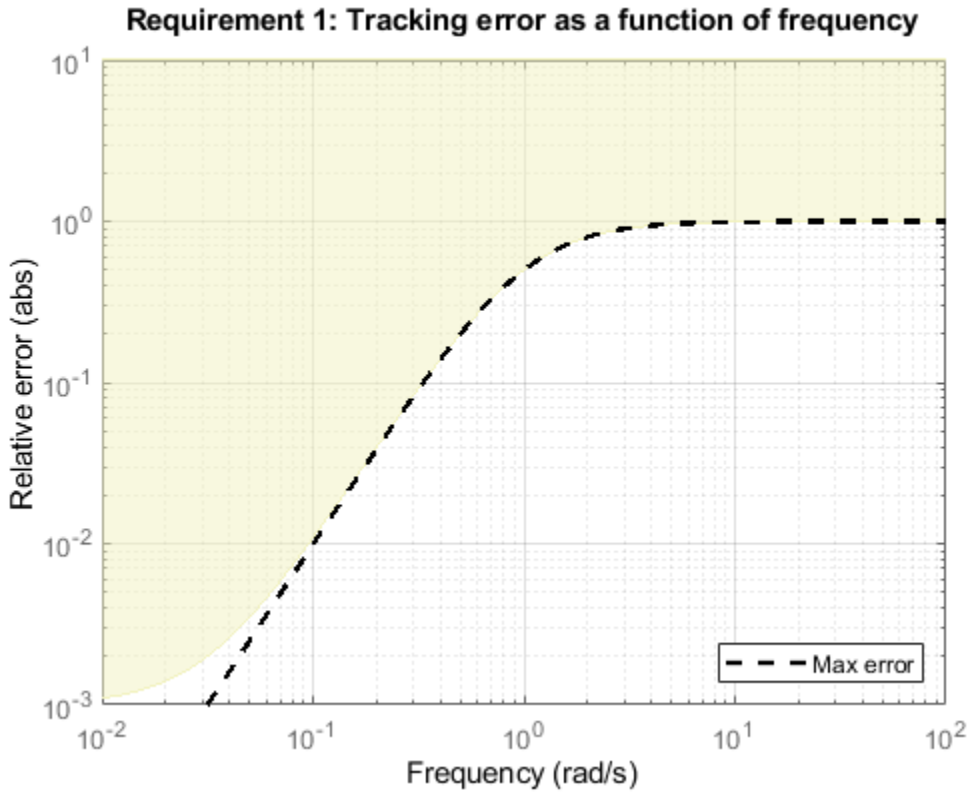
The plot highlights that sometimes it is preferable to specify tuning goals for discrete-time tuning using discrete-time gain profiles. In particular, specifying a discrete-time profile gives you more control over the behavior of the gain profile near the Nyquist frequency.

Modifications for Numeric Stability

When you use a tuning goal with a frequency-dependent specification, the tuning algorithm uses a frequency-weighting function to compute the normalized value of the tuning goal. This weighting function is derived from the gain profile that you specify. For numeric tractability, weighting functions must be stable and proper. For numeric stability, their dynamics must be in the same frequency range as the control system dynamics. For these reasons, the software might adjust the specified gain profile to eliminate undesirable low-frequency or high-frequency dynamics or asymptotes. The process of modifying the tuning goal for better numeric conditioning is called regularization.

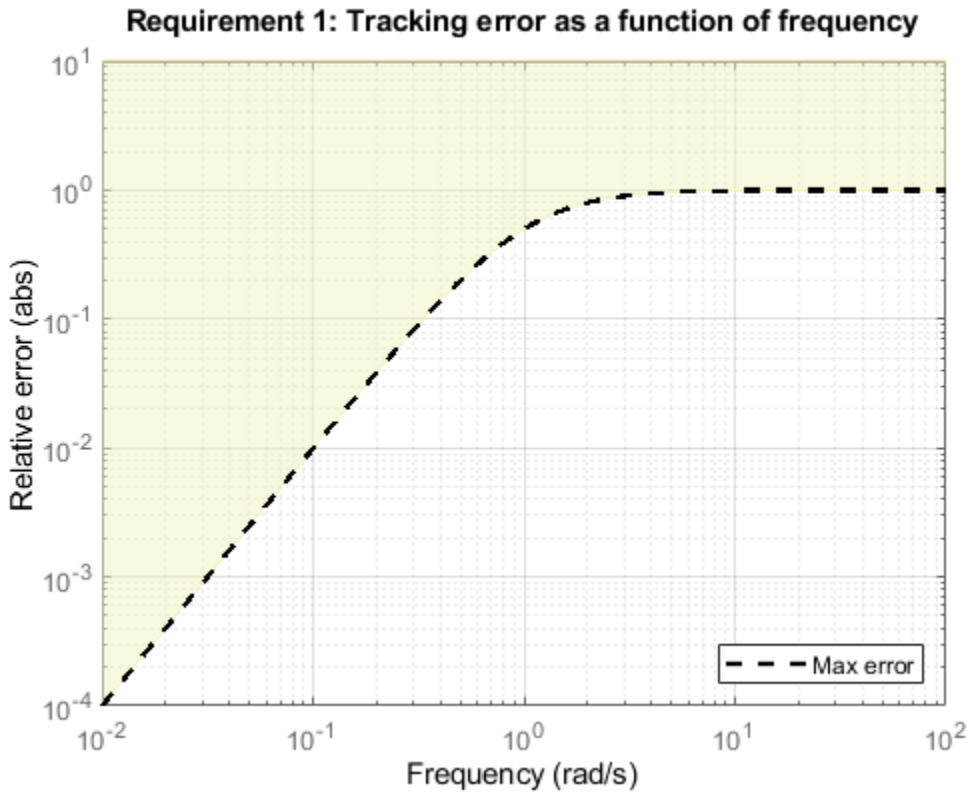
For example, consider the following tracking goal.

```
R1 = TuningGoal.Tracking('r','y',tf([1 0 0],[1 2 1]));
viewGoal(R1)
```



Here the control bandwidth is about 1 rad/s and the gain profile has two zeros at s = 0, which become unstable poles in the weighting function (see `TuningGoal.Tracking` for details). The regularization moves these zeros to about 0.01 rad/s, and the maximum tracking error levels off at about 10^{-3} (0.1%). If you need better tracking accuracy, you can explicitly specify the cutoff frequency in the error profile.

```
R2 = TuningGoal.Tracking('r','y',tf([1 0 5e-8],[1 2 1]));
viewGoal(R2)
set(gca,'Ylim',[1e-4,10])
```



However, for numeric safety, the regularized weighting function always levels off at very low and very high frequencies, regardless of the specified gain profile.

Access the Regularized Functions

When you are working at the command line, you can obtain the regularized gain profile using the `getWeight` or `getWeights` commands. For details, see the reference pages for the individual tuning goals for which the tuning algorithm performs regularization:

- `TuningGoal.Gain`
- `TuningGoal.LoopShape`
- `TuningGoal.MaxLoopGain`
- `TuningGoal.MinLoopGain`
- `TuningGoal.Rejection`
- `TuningGoal.Sensitivity`
- `TuningGoal.StepRejection`
- `TuningGoal.Tracking`

In **Control System Tuner**, you cannot view the regularized weighting functions directly. Instead, use the tuning-goal commands to generate an equivalent tuning goal, and use `getWeight` or `getWeights` to access the regularized functions.

Improve Tuning Results

If the tuning results do not adequately meet your design requirements, adjust your set of tuning goals to improve the results. For example:

- Designate tuning goals that are must-have requirements as hard goals. Or, relax tuning goals that are not absolute requirements by designating them as soft goals.
- Limit the frequency range in which frequency-domain goals are enforced.
  - In **Control System Tuner**, use the **Enforce goal in frequency range** field of the tuning goal dialog box.
  - At the command line, use the `Focus` property of the `TuningGoal` object.

If the tuning results do satisfy your design requirements, you can validate the tuned control system as described in “Validate Tuned Control System” on page 10-172.

See Also

`viewGoal`

Related Examples

- “Interpret Numeric Tuning Results” on page 10-139
- “Create Response Plots in Control System Tuner” on page 10-149
- “Validate Tuned Control System” on page 10-172

Create Response Plots in Control System Tuner

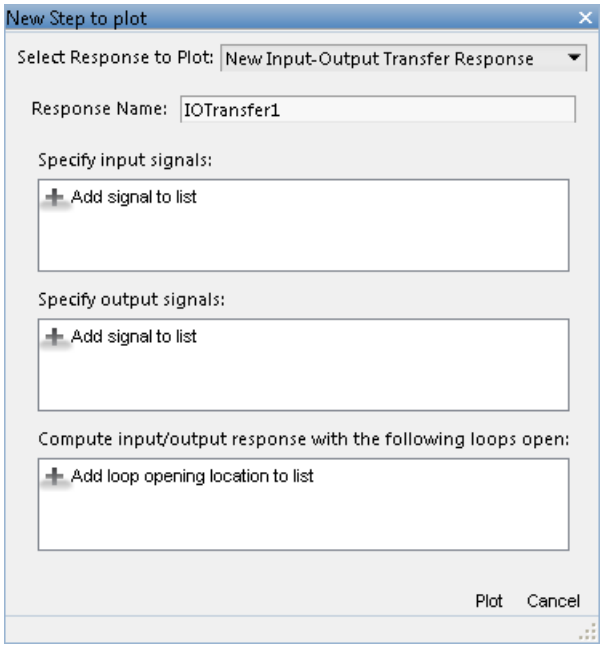
This example shows how to create response plots for analyzing system performance in **Control System Tuner**. **Control System Tuner** can generate many types of response plots in the time and frequency domains. You can view responses of SISO or MIMO transfer functions between inputs and outputs at any location in your model. When you tune your control system, **Control System Tuner** updates the response plots to reflect the tuned design. Use response plots to validate the performance of the tuned control system.

This example creates response plots for analyzing the sample model `rct_helico`.

Choose Response Plot Type

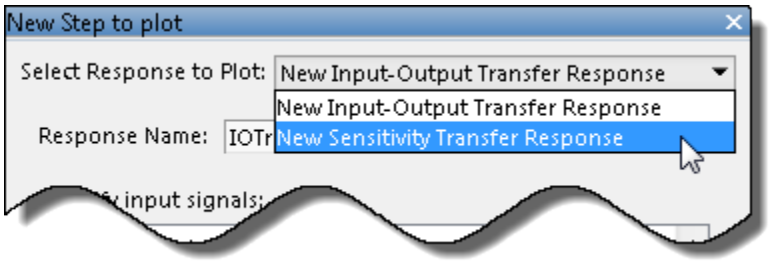
In **Control System Tuner**, in the **Control System** tab, click **New Plot**. Select the type of plot you want to create.

A new plot dialog box opens in which you specify the inputs and outputs of the portion of your control system whose response you want to plot. For example, select **New step** to create a step response plot from specified inputs to specified outputs of your system.



Specify Transfer Function

Choose which transfer function associated with the specified inputs and outputs you want to analyze.



For most response plots types, the **Select Response to Plot** menu lets you choose one of the following transfer functions:

- `New Input-Output Transfer Response` — Transfer function between specified inputs and outputs, computed with loops open at any additionally specified loop-opening locations.
- `New Sensitivity Transfer Response` — Sensitivity function computed at the specified location and with loops open at any specified loop-opening locations.
- `New Open-Loop Response` — Open loop point-to-point transfer function computed at the specified location and with loops open at any additionally specified loop-opening locations.
- `Entire System Response` — For Pole/Zero maps and I/O Pole/Zero maps only. Plot the pole and zero locations for the entire closed-loop control system.
- `Response of Tuned Block` — For Pole/Zero maps and I/O Pole/Zero maps only. Plot the pole and zero locations of tuned blocks.

Name the Response

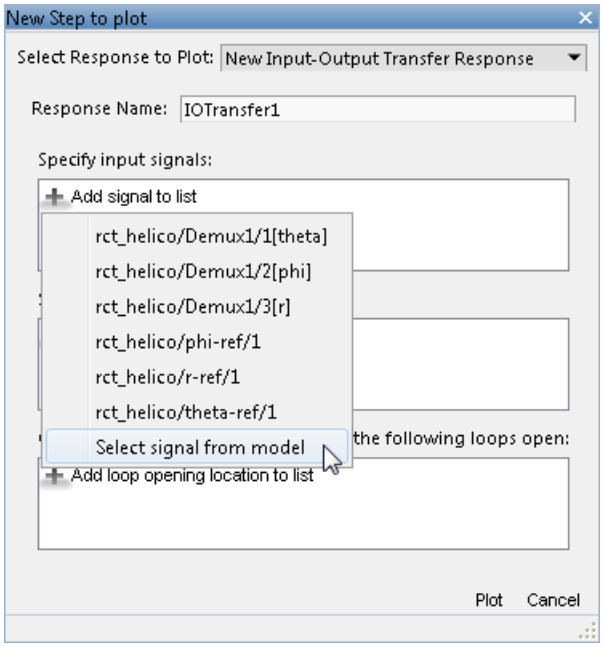
Type a name for the response in the **Response Name** text box. Once you have specified signal locations defining the response, **Control System Tuner** stores the response under this name. When you create additional new response plots, the response appears by this name in **Select Response to Plot** menu.

Choose Signal Locations for Evaluating System Response

Specify the signal locations in your control system at which to evaluate the selected response. For example, the step response plot displays the response of the system at one or more output locations to a unit step applied at one or more input locations. Use the **Specify input signals** and **Specify output signals** sections of the dialog box to specify these locations. (Other tuning goal types, such as loop-shape or stability margins, require you to specify only one location for evaluation. The procedure for specifying the location is the same as illustrated here.)

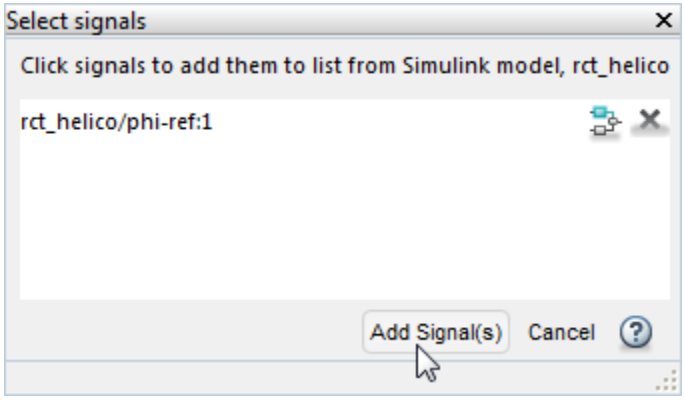
Under **Specify input signals**, click **Add signal to list**. A list of available input locations appears.

If the signal you want to designate as a step-response input is in the list, click the signal to add it to the step-response inputs. If the signal you want to designate does not appear, and you are tuning a Simulink model, click **Select signal from model**.



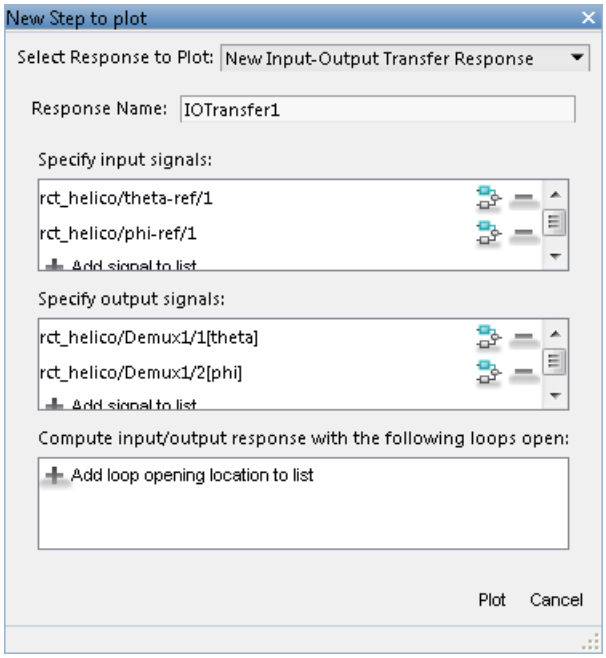
In the **Select signals** dialog box, build a list of the signals you want. To do so, click signals in the Simulink model editor. The signals that you click appear in the **Select signals** dialog box. Click one signal to create a SISO response, and click multiple signals to create a MIMO response.

Click **Add signal(s)**. The **Select signals** dialog box closes.



The signal or signals you selected now appear in the list of step-response inputs in the new-plot dialog box.

Similarly, specify the locations at which the step response is measured to the step-response outputs list. For example, the following configuration plots the MIMO response to a step input applied at `theta-ref` and `phi-ref` and measured at `theta` and `phi` in the Simulink model `rct_helico`.



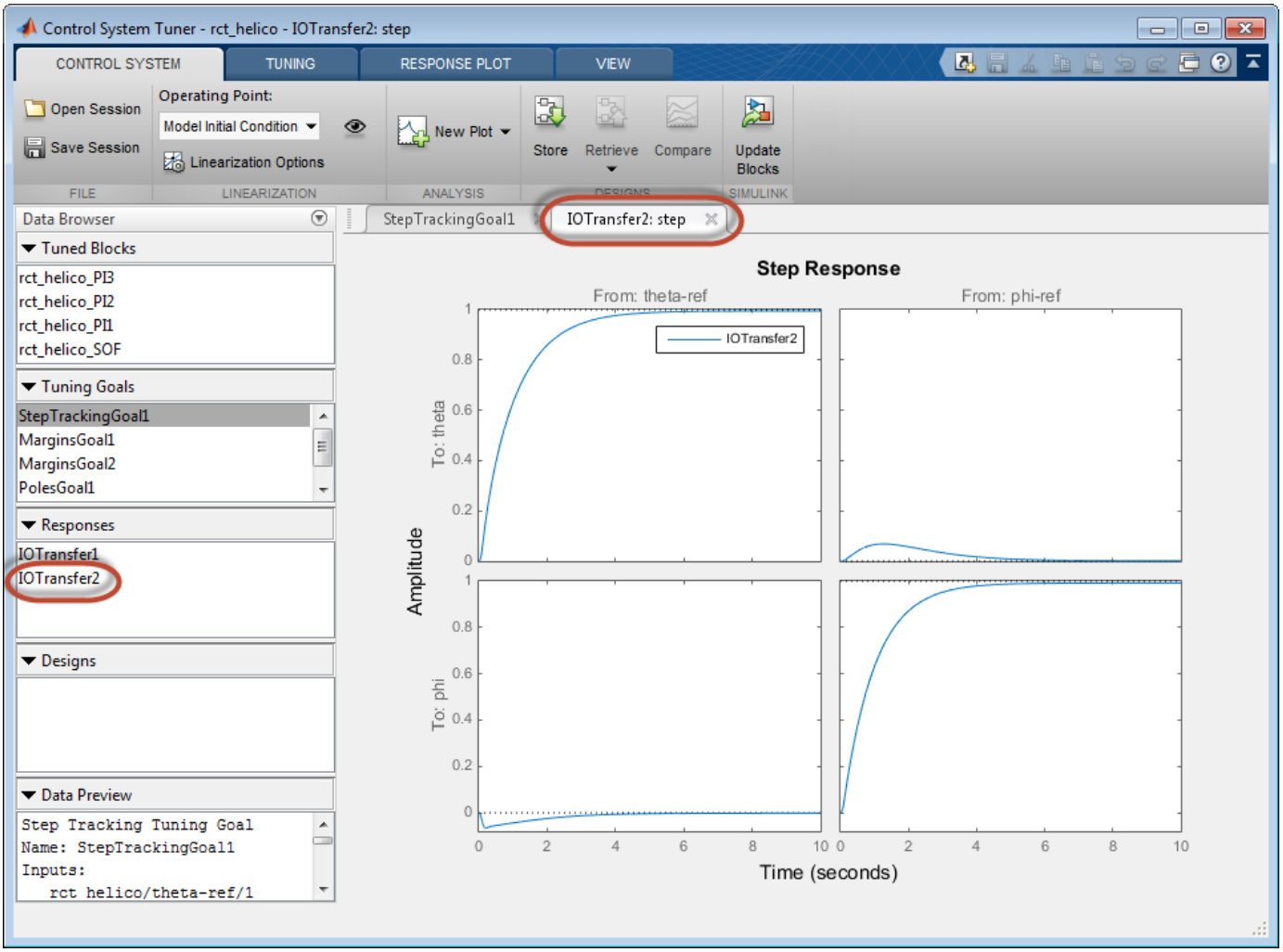
Tip To highlight any selected signal in the Simulink model, click . To remove a signal from the input or output list, click . When you have selected multiple signals, you can reorder them using and .

Specify Loop Openings

You can evaluate most system responses with loops open at one or more locations in the control system. Click **Add loop opening location to list** to specify such locations for the response.

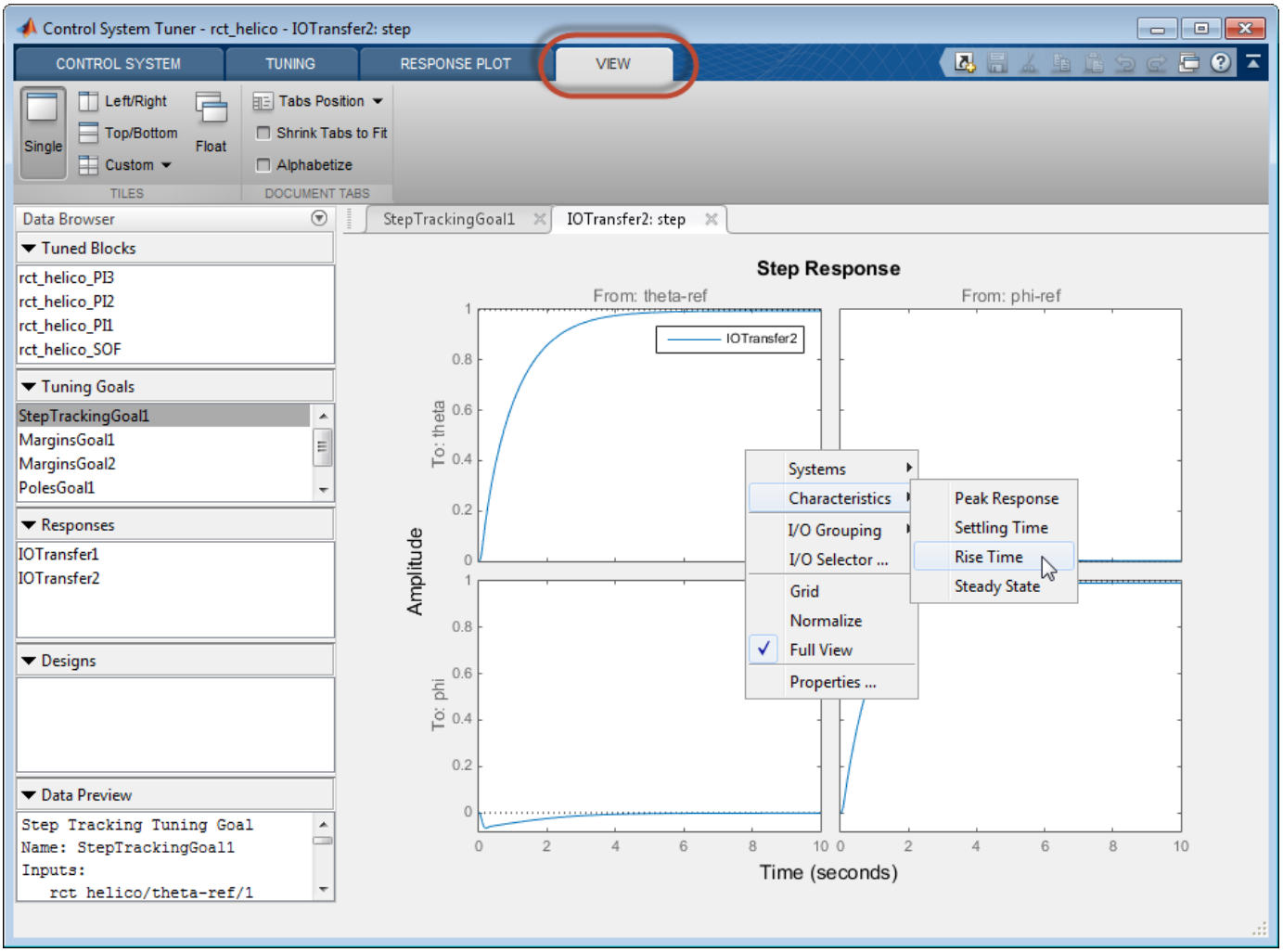
Store and Plot the Response

When you have finished specifying the response, click **Plot** in the new plot dialog box. The new response appears in the **Responses** section of the Data Browser. A new figure opens displaying the response plot. When you tune your control system, you can refer to this figure to evaluate the performance of the tuned system.



Tip To edit the specifications of the response, double-click the response in the Data Browser. Any plots using that response update to reflect the edited response.

View response characteristics such as rise-times or peak values by right-clicking on the plot. Other options for managing and organizing multiple plots are available in the **View** tab.



See Also

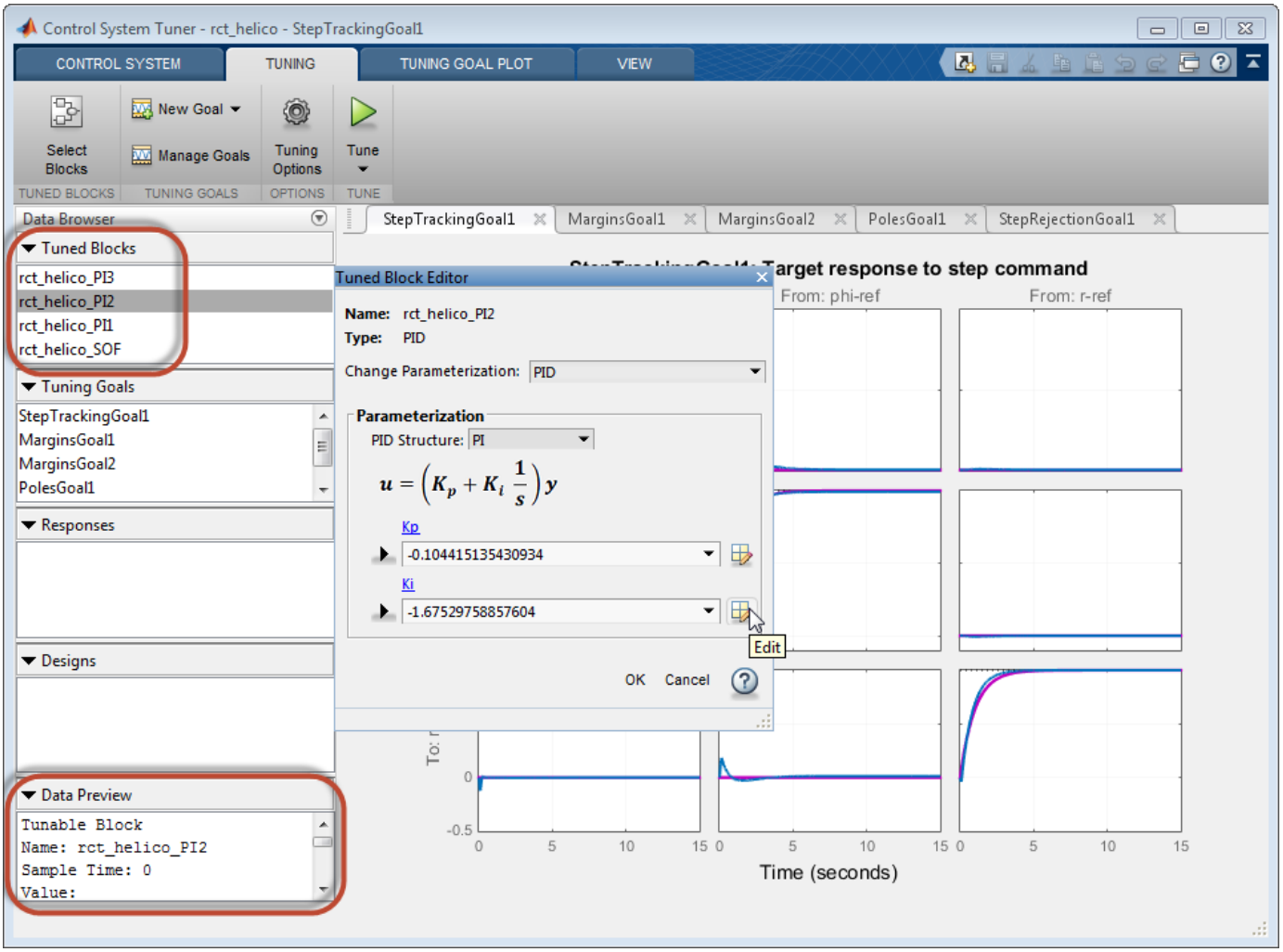
Related Examples

- “Compare Performance of Multiple Tuned Controllers” on page 10-157
- “Examine Tuned Controller Parameters in Control System Tuner” on page 10-155
- “Visualize Tuning Goals” on page 10-143

Examine Tuned Controller Parameters in Control System Tuner

After you tune your control system, **Control System Tuner** gives you two ways to view the current values of the tuned block parameters:

- In the Data Browser, in the **Tuned Blocks** area, select the block whose parameters you want to view. A text summary of the block and its current parameter values appears in the Data Browser in the **Data Preview** area.
- In the Data Browser, in the **Tuned Blocks** area, double-click the block whose parameters you want to view. The Tuned Block Editor opens, displaying the current values of the parameters. For array-valued parameters, click to open a variable editor displaying values in the array.



Note To find a tuned block in the Simulink model, right-click the block name in the **Data Browser** and select **Highlight**.

See Also

Related Examples

- “View and Change Block Parameterization in Control System Tuner” on page 10-19

Compare Performance of Multiple Tuned Controllers

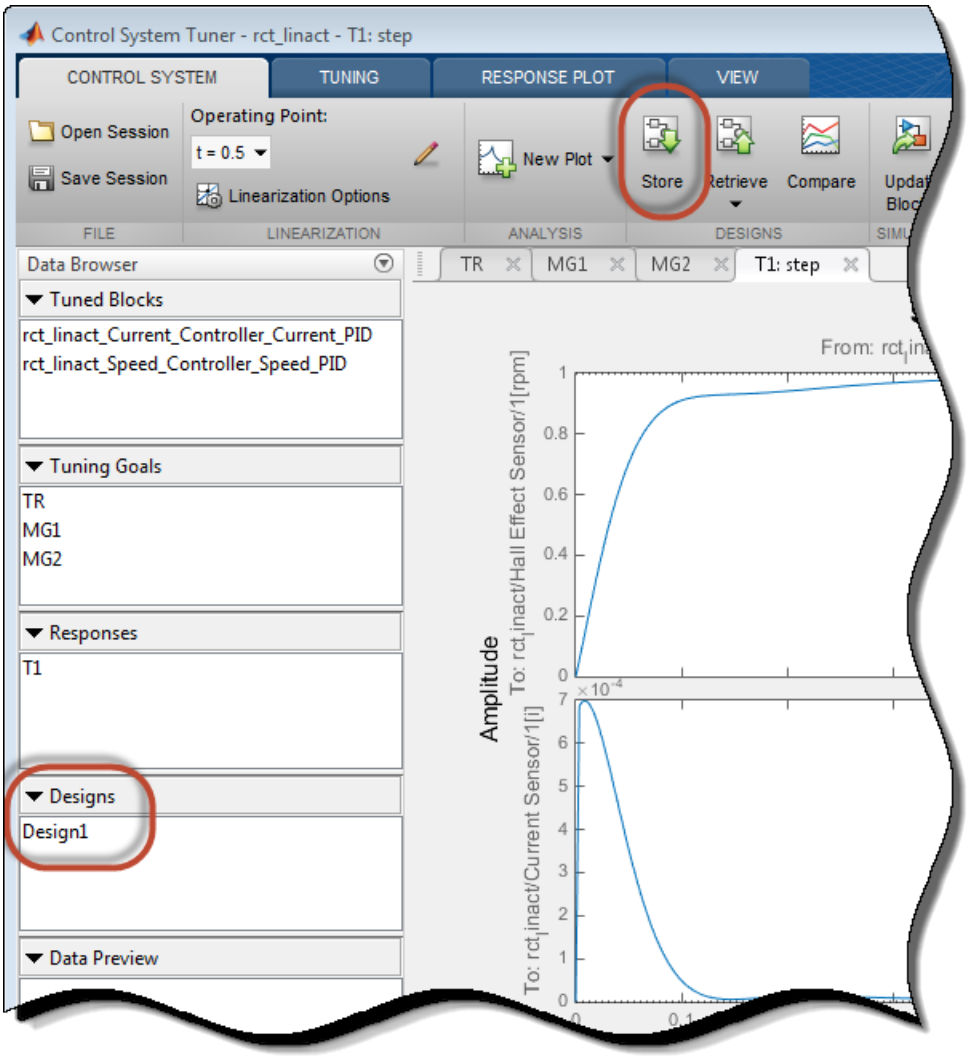
Control System Tuner lets you compare the performance of a control system tuned with two different sets of tuning goals. Such comparison is useful, for example, to see the effect on performance of changing a tuning goal from hard goal to soft goal. Comparing performance is also useful to see the effect of adding an additional tuning goal when an initial design fails to satisfy all your performance requirements either in the linearized system or when validated against a full nonlinear model.

This example compares tuning results for the sample model `rct_linact`.

Store First Design

After tuning a control system with a first set of design requirements, store the design in **Control System Tuner**.

In the **Control System** tab, click **Store**. The stored design appears in the Data Browser in the **Designs** area.



Change the name of the stored design, if desired, by right-clicking on the data browser entry.

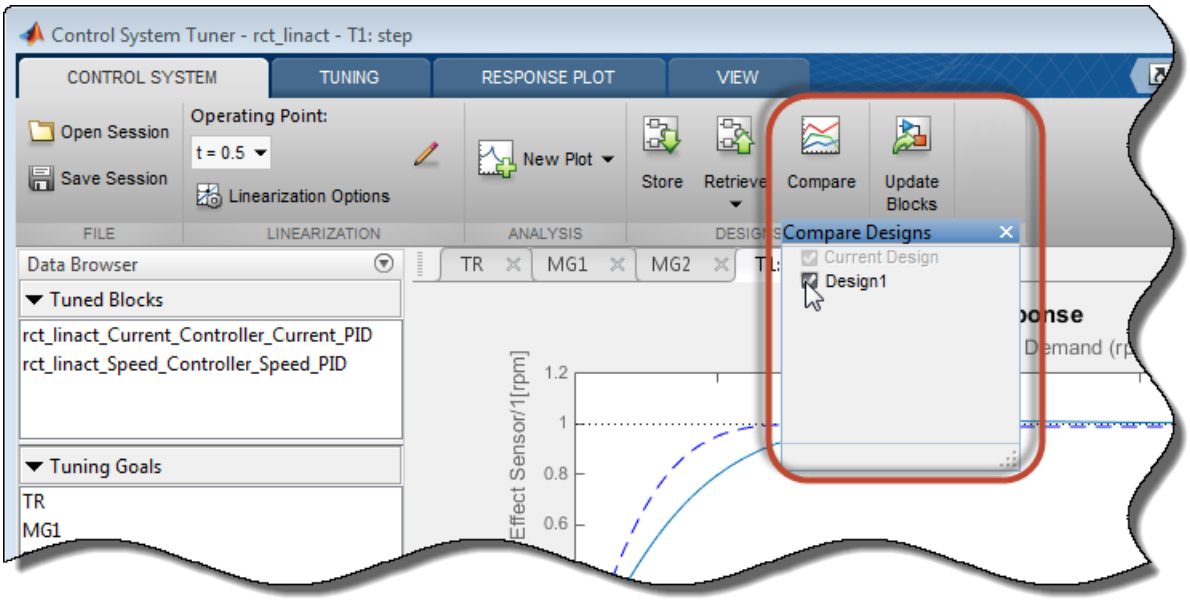
Compute New Design

In the **Tuning** tab, make any desired changes to the tuning goals for the second design. For example, add new tuning goals or edit existing tuning goals to change specifications. Or, in **Manage Goals**, change which tuning goals are active and which are designated hard constraints or soft requirements.

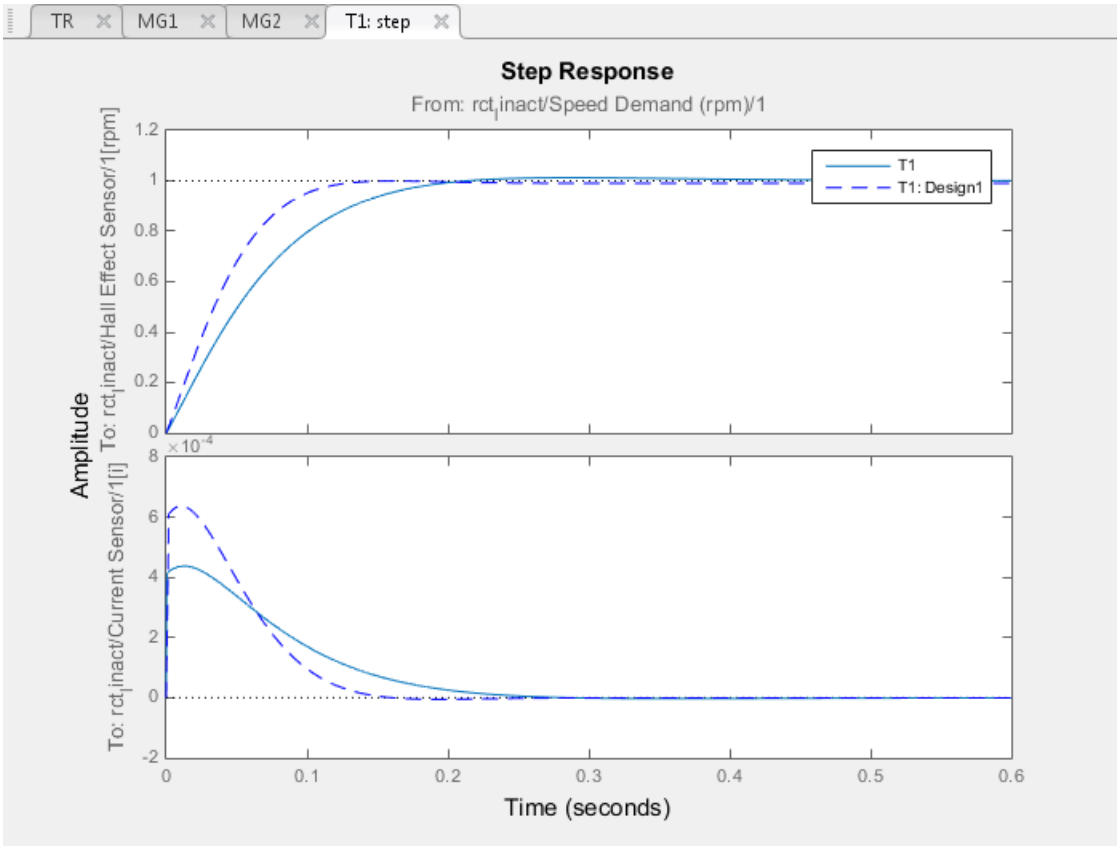
When you are ready, retune the control system with the new set of tuning goals. Click **Tune**. **Control System Tuner** updates the current design (the current set of controller parameters) with the new tuned design. All existing plots, which by default show the current design, are updated to reflect the new current design.

Compare New Design with Stored Design

Update all plots to reflect both the new design and the stored design. In the **Control System** tab, click **Compare**. The **Compare Designs** dialog box opens.



In the **Compare Designs** dialog box, the current design is checked by default. Check the box for the design you want to compare to the current design. All response plots and tuning goal plots update to reflect the checked designs. The solid trace corresponds to the current design. Other designs are identified by name in the plot legend.

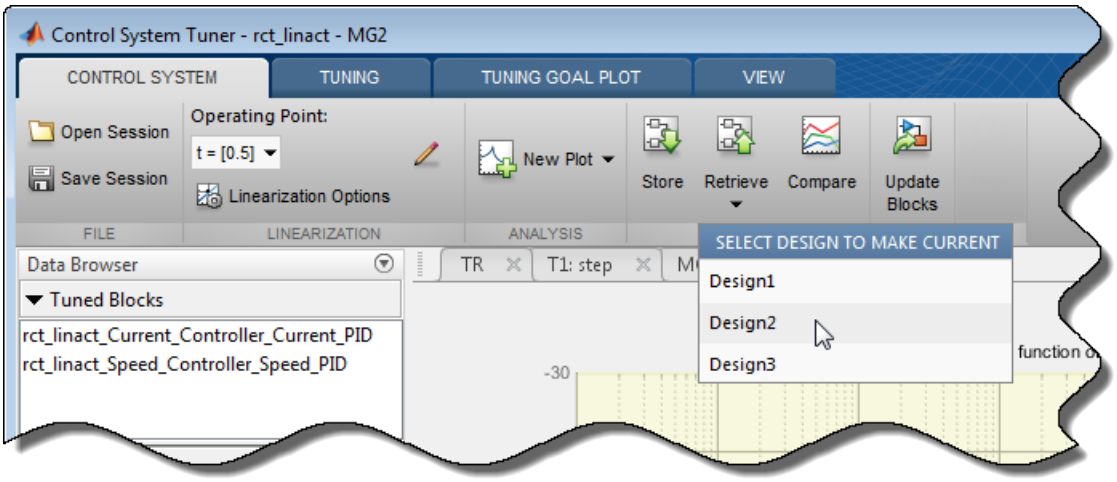


Use the same procedure save and compare as many designs as you need.

Restore Previously Saved Design

Under some conditions, it is useful to restore the tuned parameter values from a previously saved design as the current design. For example, clicking **Update Blocks** writes the current parameter values to the Simulink model. If you decide to test a stored controller design in your full nonlinear model, you must first restore those stored values as the current design.

To do so, click **Retrieve**. Select the stored design that you want to make the current design.



See Also

Related Examples

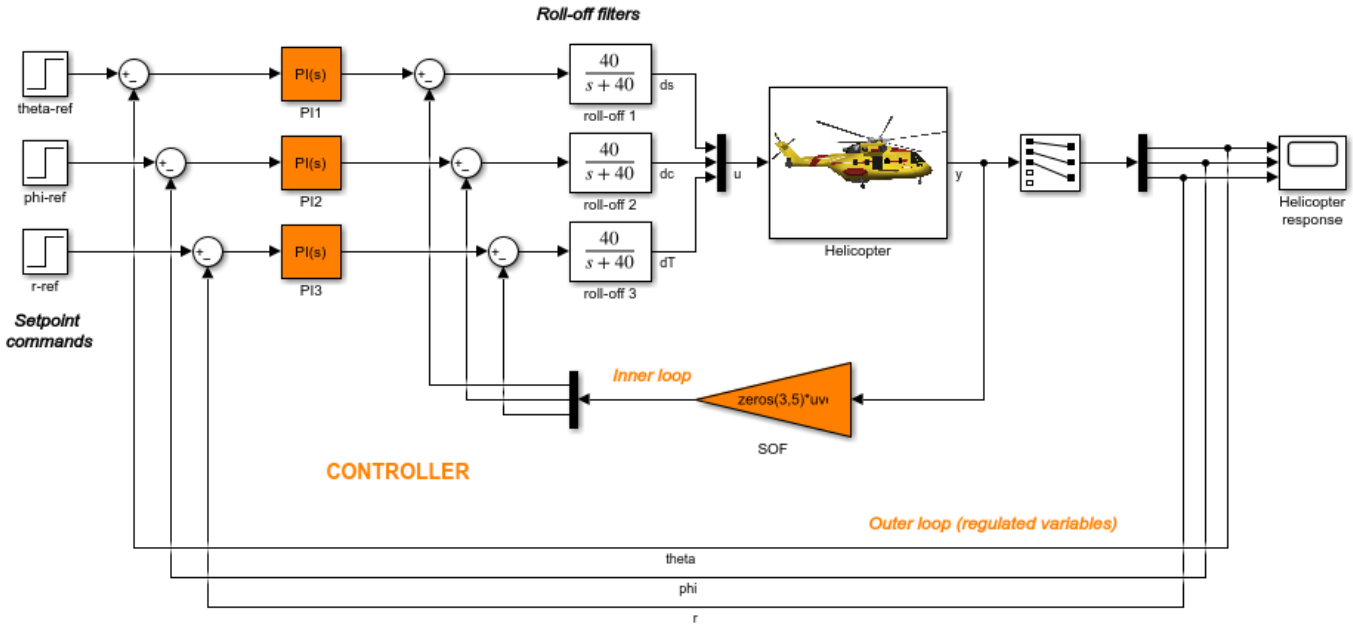
- “Create Response Plots in Control System Tuner” on page 10-149

Create and Configure slTuner Interface to Simulink Model

This example shows how to create and configure an `slTuner` interface for a Simulink® model. The `slTuner` interface parameterizes blocks in your model that you designate as tunable and allows you to tune them using `systune`. The `slTuner` interface generates a linearization of your Simulink model, and also allows you to extract linearized system responses for analysis and validation of the tuned control system.

For this example, create and configure an `slTuner` interface for tuning the Simulink model `rct_helico`, a multiloop controller for a rotorcraft. Open the model.

```
open_system('rct_helico');
```



The control system consists of two feedback loops. The inner loop (static output feedback) provides stability augmentation and decoupling. The outer loop (PI controllers) provides the desired setpoint tracking performance.

Suppose that you want to tune this model to meet the following control objectives:

- Track setpoint changes in `theta`, `phi`, and `r` with zero steady-state error, specified rise times, minimal overshoot, and minimal cross-coupling.
- Limit the control bandwidth to guard against neglected high-frequency rotor dynamics and measurement noise.
- Provide strong multivariable gain and phase margins (robustness to simultaneous gain/phase variations at the plant inputs and outputs).

The `systune` command can jointly tune the controller blocks `SOF` and the PI controllers to meet these design requirements. The `slTuner` interface sets up this tuning task.

Create the `slTuner` interface.

```
ST0 = slTuner('rct_helico',{'PI1','PI2','PI3','SOF'});
```

This command initializes the `slTuner` interface with the three PI controllers and the `SOF` block designated as tunable. Each tunable block is automatically parameterized according to its type and initialized with its value in the Simulink model.

To configure the `slTuner` interface, designate as analysis points any signal locations of relevance to your design requirements. First, add the outputs and reference inputs for the tracking requirements.

```
addPoint(ST0,{'theta-ref','theta','phi-ref','phi','r-ref','r'});
```

When you create a `TuningGoal.Tracking` object that captures the tracking requirement, this object references the same signals.

Configure the `slTuner` interface for the stability margin requirements. Designate as analysis points the plant inputs and outputs (control and measurement signals) where the stability margins are measured.

```
addPoint(ST0,{'u','y'});
```

Display a summary of the `slTuner` interface configuration in the command window.

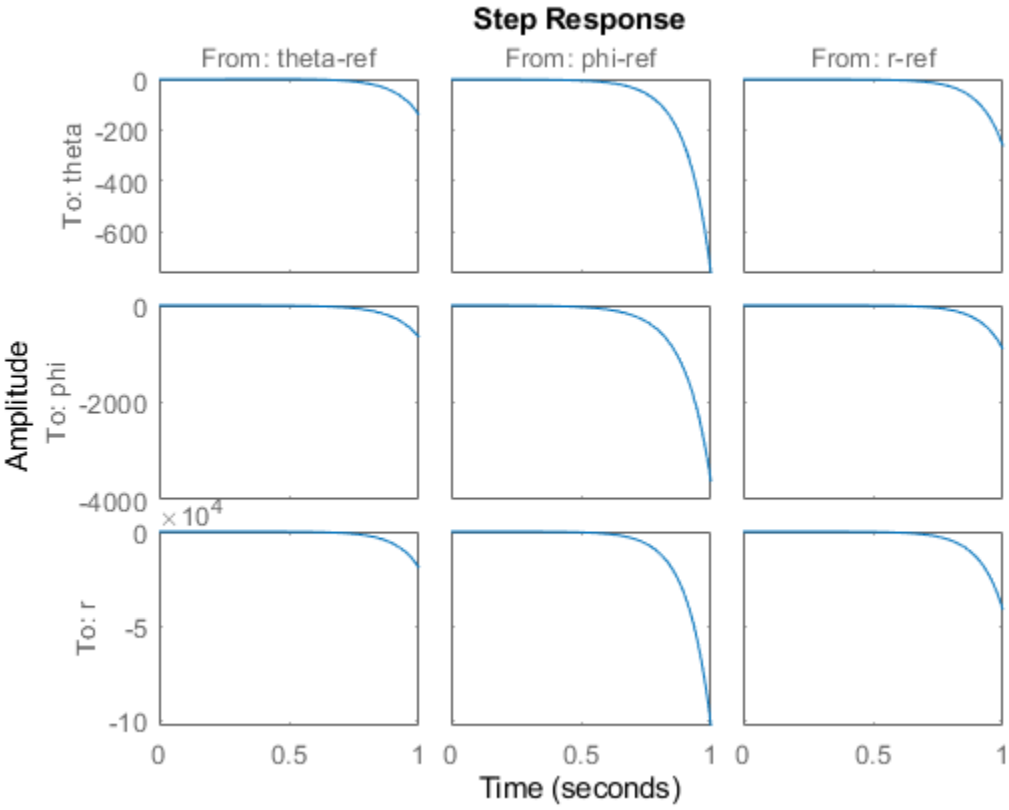
```
ST0
```

```
slTuner tuning interface for "rct_helico":

4 Tuned blocks: (Read-only TunedBlocks property)
--------------------------
Block 1: rct_helico/PI1
Block 2: rct_helico/PI2
Block 3: rct_helico/PI3
Block 4: rct_helico/SOF

8 Analysis points:
--------------------------
Point 1: 'Output Port 1' of rct_helico/theta-ref
Point 2: Signal "theta", located at 'Output Port 1' of rct_helico/Demux1
Point 3: 'Output Port 1' of rct_helico/phi-ref
Point 4: Signal "phi", located at 'Output Port 2' of rct_helico/Demux1
Point 5: 'Output Port 1' of rct_helico/r-ref
Point 6: Signal "r", located at 'Output Port 3' of rct_helico/Demux1
Point 7: Signal "u", located at 'Output Port 1' of rct_helico/Mux3
Point 8: Signal "y", located at 'Output Port 1' of rct_helico/Helicopter

No permanent openings. Use the addOpening command to add new permanent openings.
Properties with dot notation get/set access:
      Parameters         : []
      OperatingPoints    : [] (model initial condition will be used.)
      BlockSubstitutions : []
      Options            : [1x1 linearize.SlTunerOptions]
      Ts                 : 0
```

In the command window, click on any highlighted signal to see its location in the Simulink model.

In addition to specifying design requirements, you can use analysis points for extracting system responses. For example, extract and plot the step responses between the reference signals and `'theta'`, `'phi'`, and `'r'`.

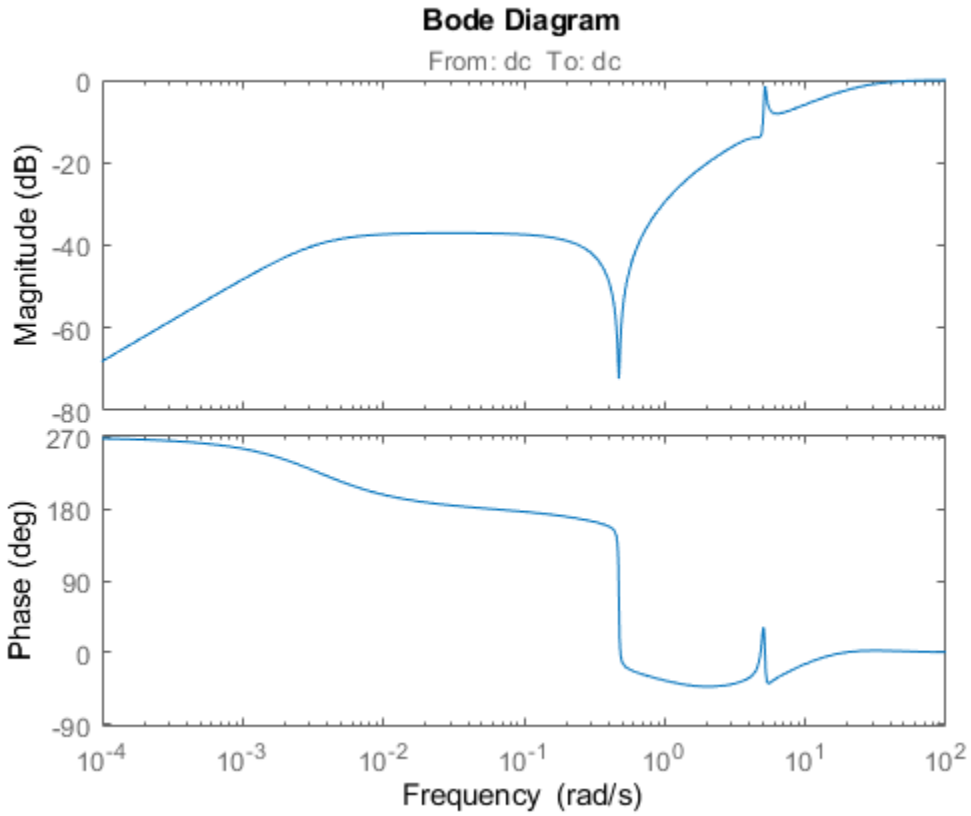
```
T0 = getIOTransfer(ST0,{'theta-ref','phi-ref','r-ref'},{'theta','phi','r'});
stepplot(T0,1)
```



All the step responses are unstable, including the cross-couplings, because this model has not yet been tuned.

After you tune the model, you can similarly use the designated analysis points to extract system responses for validating the tuned system. If you want to examine system responses at locations that are not needed to specify design requirements, add these locations to the `slTuner` interface as well. For example, plot the sensitivity function measured at the output of the block `roll-off 2`.

```
addPoint(ST0,'dc')
dcS0 = getSensitivity(ST0,'dc');
bodeplot(dcS0)
```



Suppose you want to change the parameterization of tunable blocks in the `slTuner` interface. For example, suppose that after tuning the model, you want to test whether changing from PI to PID controllers yields improved results. Change the parameterization of the three PI controllers to PID controllers.

```
PID0 = pid(0,0.001,0.001,.01);  % initial value for PID controllers
PID1 = tunablePID('C1',PID0);
PID2 = tunablePID('C2',PID0);
PID3 = tunablePID('C3',PID0);

setBlockParam(ST0,'PI1',PID1,'PI2',PID2,'PI3',PID3);
```

After you configure the `slTuner` interface to your Simulink model, you can create tuning goals and tune the model using `systune` or `looptune`.

See Also

`slTuner` | `addBlock` | `addPoint` | `setBlockParam` | `getIOTransfer` | `getSensitivity`

Related Examples

- “Mark Signals of Interest for Control System Analysis and Design” on page 2-38
- “Multiloop Control of a Helicopter”
- “Control of a Linear Electric Actuator”

Stability Margins in Control System Tuning

In control system tuning, you specify target gain and phase margins using “Margins Goal” on page 10-111 (for **Control System Tuner**) or `TuningGoal.Margins` (for `systune`). The software provides tools to help you visualize and interpret the gain and phase margins in your tuned system.

Gain and Phase Margins

Gain and phase margins measure the tolerance of a control loop to variations in the open-loop system response. The Margins Goal and `TuningGoal.Margins` rely on the notion of a disk margin to compute gain and phase margins. Like classical gain and phase margins, disk margins quantify the stability of a closed-loop system against gain or phase variations in the open-loop response. Disk margins also take into account all frequencies and loop interactions. Therefore, disk-based margin analysis provides a stronger guarantee of stability than the classical gain and phase margins. For more information about disk margins, see “Stability Analysis Using Disk Margins” (Robust Control Toolbox).

For a SISO system, the gain and phase margins indicate how much the gain or phase of the open-loop response L can change without loss of stability.

For MIMO systems, gain and phase margins are interpreted as follows:

- Gain margin — Stability is preserved when the gain changes up to the gain margin value in each feedback channel. The gain can change in all channels simultaneously, and by a different amount in each channel.
- Phase margin — Stability is preserved when the phase changes up to the phase margin value in each feedback channel. The phase can change in all channels simultaneously, and by a different amount in each channel.

Gain and phase margins typically vary across frequencies. For example, in a SISO loop, a gain margin of 5 dB at 2 rad/s indicates that closed-loop stability is maintained when the loop gain increases or decreases by as much as 5 dB at this frequency. For control system tuning, you specify target values for the minimum (worst) margins across all frequencies. The margin tuning goal assumes symmetric ranges of variation, such as ±5 dB or ±30°.

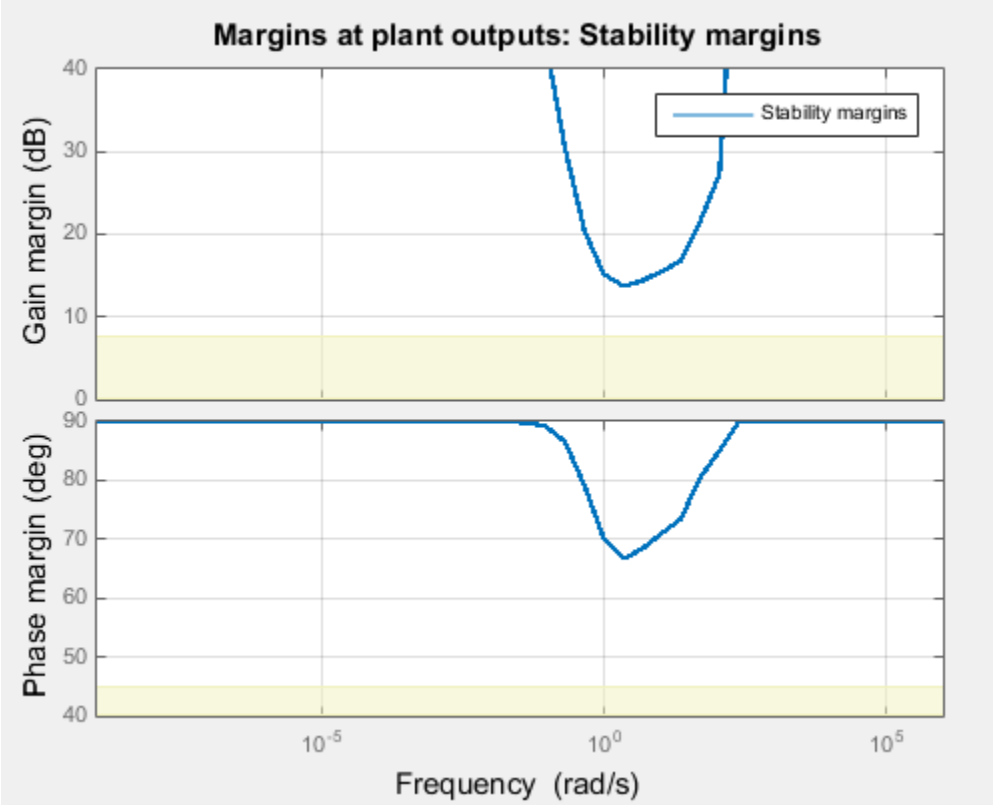
Interpret Gain and Phase Margin Plots

For control system tuning, visualize system stability margins to help evaluate the performance of the tuned system.

- In **Control System Tuner**, use a “Margins Goal” on page 10-111 or “Quick Loop Tuning” on page 10-42.
- At the command line, use `viewGoal`. For instance, if `S` is the control system, and `Req` is a `TuningGoal.Margins` goal, enter the following.

  ```
  viewGoal(Req,S)
  ```

`viewGoal` produces a plot with a yellow shaded region where the target margins are not met. The plot also shows the gain and phase margins for the current values of the tunable parameters in the control system. These margins appear as a blue trace that typically varies across frequencies. For instance, the following plot shows a typical result.

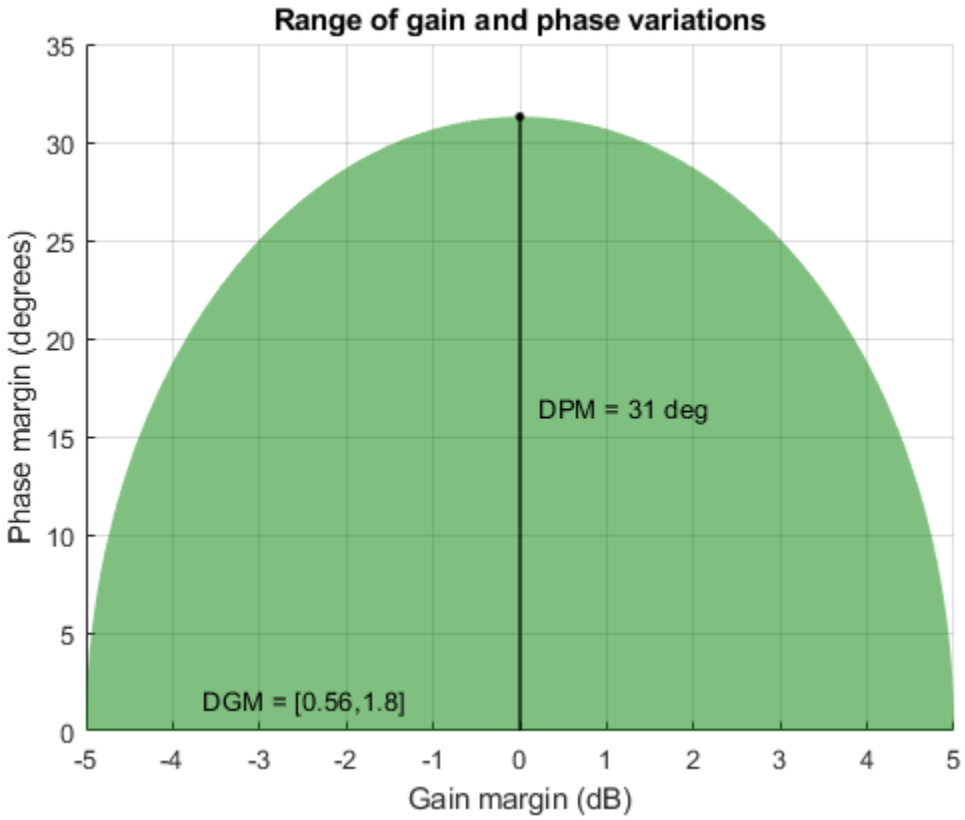


The plot shows that the frequency of the gain or phase variation can affect how large a perturbation the system can tolerate without going unstable. The minimum (worst) gain and phase margins occur at about 2 rad/s. At this frequency, the system can tolerate changes in open-loop gain of about ±14 dB, or changes in phase of about ±66°. For this system, the margins at all frequencies are well above the target margins used for tuning, shown in yellow.

Simultaneous Gain and Phase Variations

In general, gain margins are determined assuming no phase variation, and phase margins are determined assuming no gain variation. In practice, your system can experience simultaneous gain and phase variations. Disk-based margin analysis also gives you a range of simultaneous gain and phase variations that the system can tolerate. For instance, suppose that your system has a disk-based gain margin of 5 dB. This system remains stable for gain changes of ±5 dB, assuming no phase variation. Use the `diskmarginplot` command to visualize the region of simultaneous gain and phase variations that the system can tolerate.

```
diskmarginplot(db2mag(5))
```



The shaded region shows the stable range of combined gain and phase variations for a disk-based gain margin of 5 dB. With no phase variation, the system can tolerate the full range of gain variation, –5 dB to 5 dB, or gain that changes by a factor within the range `DGM = [0.56,1.8]`. Adding in phase variation reduces the tolerable gain variation. For instance, If the phase is allowed to vary by ±25°, the tolerable gain variation drops to a range of about ±3 dB. The disk-based phase margin is the allowable phase variation when there is no gain variation, in this case about ±31°, shown in the plot as `DPM`.

For more information about disk margins, see “Stability Analysis Using Disk Margins” (Robust Control Toolbox).

Algorithm

The gain and phase margin values are both derived from the disk margin. The disk margin measures the radius of a circular exclusion region centered near the critical point. (See “Stability Analysis Using Disk Margins” (Robust Control Toolbox).) For a system with open-loop response $L(j\omega)$, this radius α is a function of the scaled norm:

$$\frac{1}{\alpha} = \min_{D\,\text{diagonal}} \left\| D(j\omega)^{-1}(I - L(j\omega))(I + L(j\omega))^{-1} D(j\omega) \right\|_\infty .$$

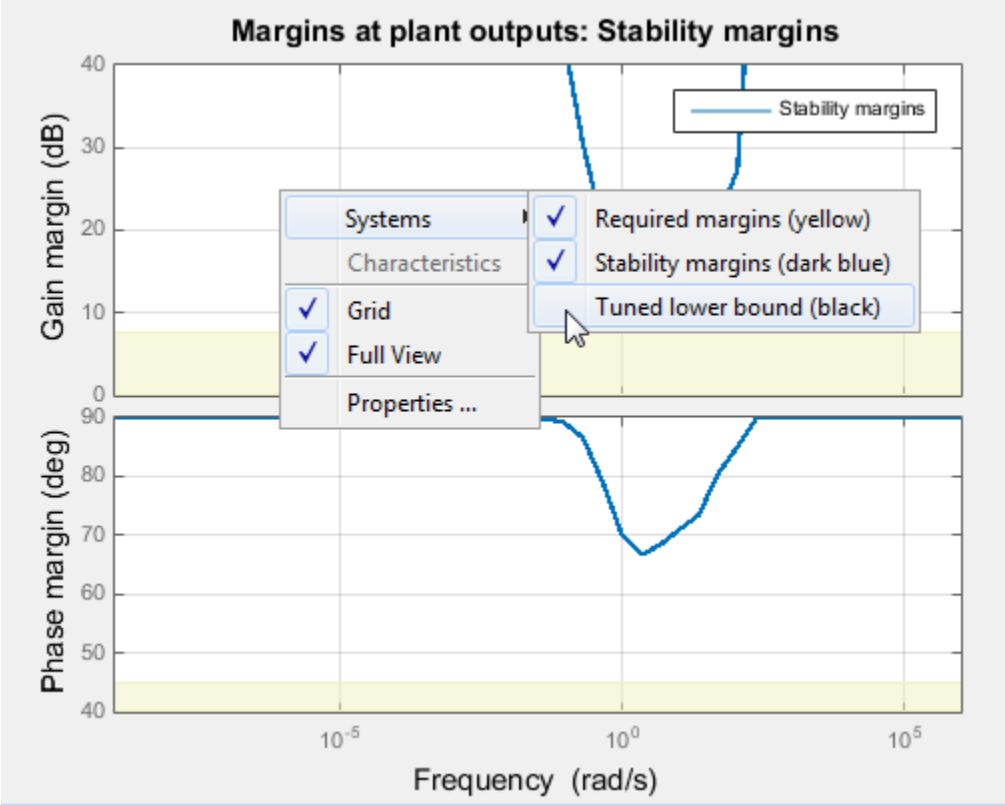
Unlike classical gain and phase margins, the disk margins and associated gain and phase margins guarantee that the open-loop response stays at a safe distance from the critical point at all frequencies.

Impact of Scaling

The frequency dependence of the gain and phase margins can be obtained by an exact calculation involving μ-analysis. However, for computational efficiency, the tuning algorithm uses an approximate calculation with a constant scaling D instead of the frequency-dependent scaling $D(j\omega)$:

$$\frac{1}{\alpha} = \min_{D \text{ diagonal}} \left\| D^{-1}(I - L(j\omega))(I + L(j\omega))^{-1} D \right\|_\infty .$$

This approximation is an upper bound on $1/\alpha$, or a lower bound on α. It can therefore yield smaller margins in parts of the frequency range, especially at frequencies away from the frequency at which the minimum margin occurs. The smaller margin is still a guaranteed margin, but it might be more conservative than the true margin. To see the lower bound used by the tuning algorithm, right-click on the stability-margins plot and select **Systems** > **Tuned lower bound**.



If you see a significant gap between the actual margins of the tuned system (blue curve) and the lower-bound approximation used for tuning (black curve), try increasing the D-scaling order to introduce some frequency dependence into the scaling. For tuning in **Control System Tuner**, set the D-scaling order in the Margins Goal dialog box. For command-line tuning, set this value using the `ScalingOrder` property of `TuningGoal.Margins`. The default order is zero (static scaling).

See Also

`TuningGoal.Margins` | `diskmargin` | `viewGoal`

More About

- “Loop Shape and Stability Margin Specifications” on page 13-34
- “Margins Goal” on page 10-111
- “Stability Analysis Using Disk Margins” (Robust Control Toolbox)

Tune Control System at the Command Line

After specifying your tuning goals using `TuningGoal` objects (see “Tuning Goals”), use `systune` to tune the parameters of your model.

The `systune` command lets you designate one or more design goals as hard goals. This designation gives you a way to differentiate must-have goals from nice-to-have tuning goals.`systune` attempts to satisfy hard requirements by driving their associated cost functions below 1. Subject to that constraint, the software comes as close as possible to satisfying remaining (soft) requirements. For best results, make sure you can obtain a reasonable design with all goals treated as soft goals before attempting to enforce any goal as a hard constraint.

Organize your `TuningGoal` objects into a vector of soft requirements and a vector of hard requirements. For example, suppose you have created a tracking requirement, a rejection requirement, and stability margin requirements at the plant inputs and outputs. The following commands tune the control system represented by `T0`, treating the stability margins as hard goals, the tracking and rejection requirements as soft goals. (`T0` is either a `genss` model or an `slTuner` interface previously configured for tuning.)

```
SoftReqs = [Rtrack,Rreject];
HardReqs = [RmargIn,RmargOut];
[T,fSoft,gHard] = systune(T0,SoftReqs,HardReqs);
```

`systune` converts each tuning requirement into a normalized scalar value, f for the soft constraints and g for the hard constraints. The command adjusts the tunable parameters of `T0` to minimize the f values, subject to the constraint that each $g < 1$. `systune` returns the vectors `fSoft` and `gHard` that contain the final normalized values for each tuning goal in `SoftReqs` and `HardReqs`.

Use `systuneOptions` to configure additional options for the `systune` algorithm, such as the number of independent optimization runs, convergence tolerance, and output display options.

See Also

`systune` | `systune (for slTuner)` | `systuneOptions`

More About

- “Interpret Numeric Tuning Results” on page 10-139

Speed Up Tuning with Parallel Computing Toolbox Software

If you have the Parallel Computing Toolbox software installed, you can speed up the tuning of fixed-structure control systems. When you run multiple randomized optimization starts with `systune`, `looptune`, or `hinfstruct`, parallel computing speeds up tuning by distributing the optimization runs among workers.

To distribute randomized optimization runs among workers:

If **Automatically create a parallel pool** is not selected in your Parallel Computing Toolbox preferences (Parallel Computing Toolbox), manually start a parallel pool using `parpool`. For example:

```
parpool;
```

If **Automatically create a parallel pool** is selected in your preferences, you do not need to manually start a pool.

Create a `systuneOptions`, `looptuneOptions`, or `hinfstructOptions` set that specifies multiple random starts. For example, the following options set specifies 20 random restarts to run in parallel for tuning with `looptune`:

```
options = systuneOptions('RandomStart',20,'UseParallel',true);
```

Setting `UseParallel` to `true` enables parallel processing by distributing the randomized starts among available workers in the parallel pool.

Use the options set when you call the tuning command. For example, if you have already created a tunable control system model, `CL0`, and tunable controller, and tuning requirement vectors `SoftReqs` and `HardReqs`, the following command uses parallel computing to tune the control system of `CL0` with `systune`.

```
[CL,fSoft,gHard,info] = systune(CL0,SoftReq,Hardreq,options);
```

To learn more about configuring a parallel pool, see the Parallel Computing Toolbox documentation.

See Also

`parpool`

Related Examples

- “Using Parallel Computing to Accelerate Tuning”

More About

- “Specify Your Parallel Preferences” (Parallel Computing Toolbox)

Validate Tuned Control System

When you tune a control system using `systune` or **Control System Tuner**, you must validate the results of tuning. The tuning results provide numeric and graphical indications of how well your tuning goals are satisfied. (See “Interpret Numeric Tuning Results” on page 10-139 and “Visualize Tuning Goals” on page 10-143.) Often, you want to examine other system responses using the tuned controller parameters. If you are tuning a Simulink model, you must also validate the tuned controller against the full nonlinear system. At the command line and in **Control System Tuner**, there are several tools to help you validate the tuned control system.

Extract and Plot System Responses

In addition to the system responses corresponding to your tuning goals (see “Visualize Tuning Goals” on page 10-143), you can evaluate the tuned system performance by plotting other system responses. For instance, evaluate reference tracking or overshoot performance by plotting the step response of transfer function from the reference input to the controlled output. Or, evaluate stability margins by examining an open-loop transfer function. You can extract any transfer function you need for analysis from the tuned model of your control system.

Extract System Responses at the Command Line

The tuning tools include analysis functions that let you extract responses from your tuned control system.

For generalized state-space (`genss`) models, use:

- `getIOTransfer`
- `getLoopTransfer`
- `getSensitivity`
- `getCompSensitivity`

For an `slTuner` interface, use:

- `getIOTransfer (for slTuner)`
- `getLoopTransfer (for slTuner)`
- `getSensitivity (for slTuner)`
- `getCompSensitivity (for slTuner)`

In either case, the extracted responses are represented by state-space (`ss`) models. You can analyze these models using commands such as `step`, `bode`, `sigma`, or `margin`.

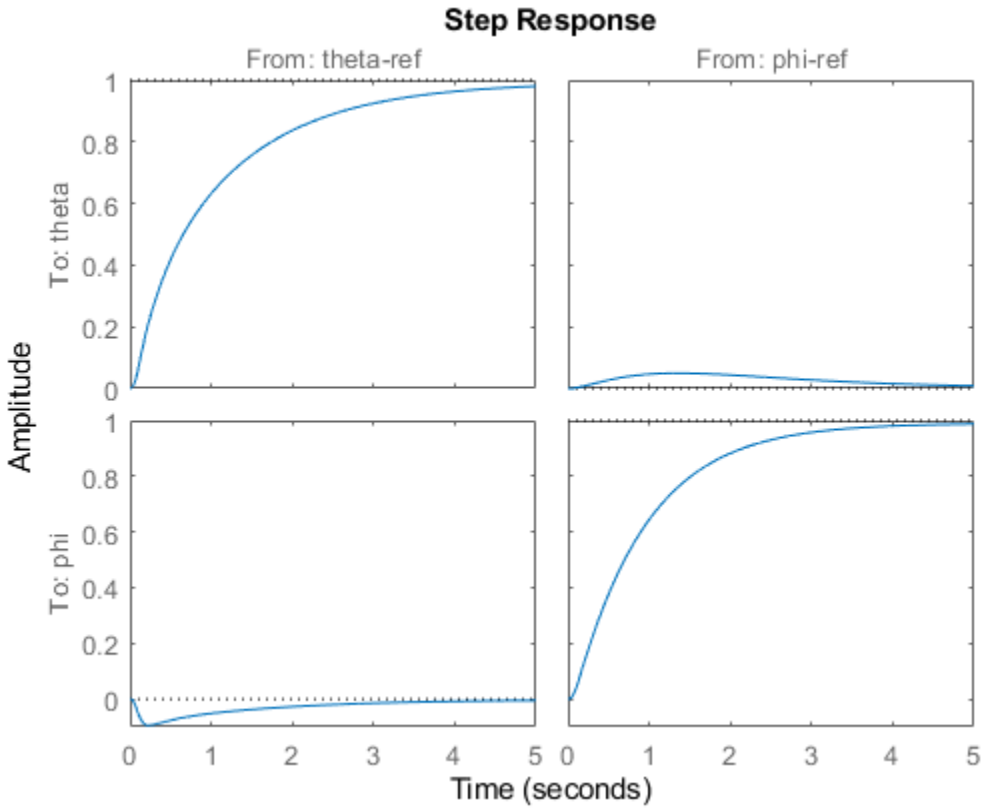
For instance, suppose that you are tuning the control system of the example “Multiloop Control of a Helicopter”. You have created an `slTuner` interface `ST0` for the Simulink model. You have also specified tuning goals `TrackReq`, `MarginReq1`, `MarginReq2`, and `PoleReq`. You tune the control system using `systune`.

```
AllReqs = [TrackReq,MarginReq1,MarginReq2,PoleReq];
ST1 = systune(ST0,AllReqs);
```

```
Final: Soft = 1.12, Hard = -Inf, Iterations = 71
```

Suppose also that `ST0` has analysis points that include signals named `theta-ref`, `theta`, `phi-ref`, and `phi`. Use `getIOTransfer` to extract the tuned transfer functions from `theta-ref` and `phi-ref` to `theta` and `phi`.

```
T1 = getIOTransfer(ST1,{'theta-ref','phi-ref'},{'theta','phi'});
step(T1,5)
```



The step plot shows that the extracted transfer function is the 2-input, 2-output response from the specified reference inputs to the specified outputs.

For an example that shows how to extract responses from a tuned `genss` model, see “Extract Responses from Tuned MATLAB Model at the Command Line” on page 10-176.

For additional examples, see “Validating Results” on page 13-43.

System Responses in Control System Tuner

For information about extracting and plotting system responses in **Control System Tuner**, see “Create Response Plots in Control System Tuner” on page 10-149.

Validate Design in Simulink Model

When you tune a Simulink model, the software evaluates tuning goals for a linearization of the model. Similarly, analysis commands such as `getIOTransfer` extract linearized system responses. Therefore, you must validate the tuned controller parameters by simulating the full nonlinear model

with the tuned controller parameters, even if the tuned linear system meets all your design requirements. To do so, write the tuned parameter values to the model.

Tip If you tune the Simulink model at an operating point other than the model initial condition, initialize the model at the same operating point before validating the tuned controller parameters. See “Simulate Simulink Model at Specific Operating Point” on page 1-93.

Write Parameters at the Command Line

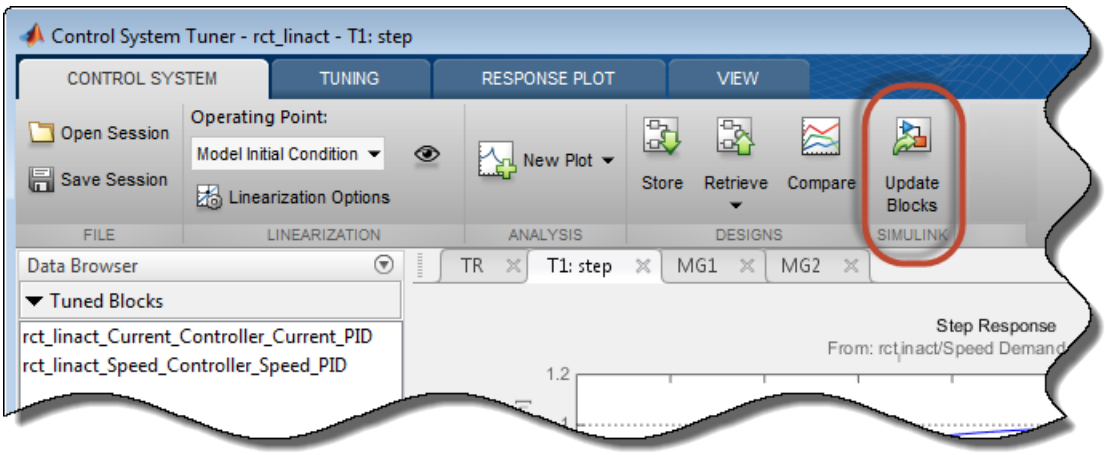
To write tuned block values from a tuned `slTuner` interface to the corresponding Simulink model, use the `writeBlockValue` command. For example, suppose `ST1` is the tuned `slTuner` interface returned by `systune`. The following command writes the tuned parameters from `ST1` to the associated Simulink model.

```
writeBlockValue(ST1)
```

Simulate the Simulink model to evaluate system performance with the tuned parameter values.

Write Parameters in Control System Tuner

To write tuned block parameters to a Simulink model, in the **Control System** tab, click **Update Blocks**.



Control System Tuner transfers the current values of the tuned block parameters to the corresponding blocks in the Simulink model. Simulate the model to evaluate system performance using the tuned parameter values.

To update Simulink model with parameter values from a previous design stored in **Control System Tuner**, click **Retrieve** and select the stored design that you want to make the current design. Then click **Update Blocks**.

See Also

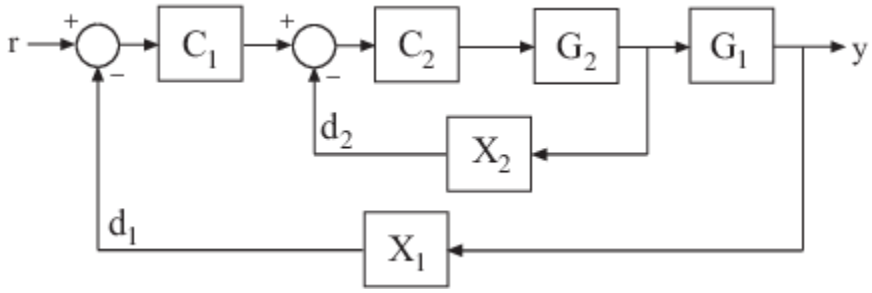
Related Examples

- “Extract Responses from Tuned MATLAB Model at the Command Line” on page 10-176
- “Create Response Plots in Control System Tuner” on page 10-149
- “Visualize Tuning Goals” on page 10-143

Extract Responses from Tuned MATLAB Model at the Command Line

This example shows how to analyze responses of a tuned control system by using `getIOTransfer` to compute responses between various inputs and outputs of a closed-loop model of the system. You can obtain other responses using similar functions such as `getLoopTransfer` and `getSensitivity`.

Consider the following control system.



Suppose you have used `systune` to tune a `genss` model of this control system. The result is a `genss` model, `T`, which contains tunable blocks representing the controller elements `C1` and `C2`. The tuned model also contains `AnalysisPoint` blocks that represent the analysis-point locations, `X1` and `X2`.

Analyze the tuned system performance by examining various system responses extracted from `T`. For example, examine the response at the output, y, to a disturbance injected at the point d_1.

```
H1 = getIOTransfer(T,'X1','y');
```

`H1` represents the closed-loop response of the control system to a disturbance injected at the implicit input associated with the `AnalysisPoint` block `X1`, which is the location of d_1:

`H1` is a `genss` model that includes the tunable blocks of `T`. `H1` allows you to validate the disturbance response of your tuned system. For example, you can use analysis commands such as `bodeplot` or `stepplot` to analyze `H1`. You can also use `getValue` to obtain the current value of `H1`, in which all the tunable blocks are evaluated to their current numeric values.

Similarly, examine the response at the output to a disturbance injected at the point d_2.

```
H2 = getIOTransfer(T,'X2','y');
```

You can also generate a two-input, one-output model representing the response of the control system to simultaneous disturbances at both d_1 and d_2. To do so, provide `getIOTransfer` with a cell array that specifies the multiple input locations.

```
H = getIOTransfer(T,{'X1','X2'},'y');
```

See Also

`getIOTransfer` | `getLoopTransfer` | `getSensitivity` | `getCompSensitivity` | `AnalysisPoint`

Related Examples

- “Interpret Numeric Tuning Results” on page 10-139

Gain-Scheduled Controllers

Gain Scheduling Basics

Gain scheduling is an approach to control of nonlinear systems using a family of linear controllers, each providing satisfactory control for a different operating point of the system. Gain-scheduled control is typically implemented using a controller whose gains are automatically adjusted as a function of scheduling variables that describe the current operating point. Such variables can include time, external operating conditions, or system states such as orientation or velocity.

Gain-scheduled control systems are often designed by choosing a small set of operating points, the design points, and designing a suitable linear controller for each point. In operation, the system switches or interpolates between these controllers according to the current values of the scheduling variables.

Gain scheduling is most suitable when the scheduling variables are external parameters that vary slowly compared to the control bandwidth, such as the ambient temperature of a chemical reaction or the speed of a cruising aircraft. Gain scheduling is most challenging when the scheduling variables depend on fast-varying states of the system. Because local linear performance near operating points is no guarantee of global performance in nonlinear systems, extensive simulation-based validation is required. See [1] for an overview of gain scheduling and its challenges.

To design a gain-scheduled control system, you need:

- An operating range, defined as a set of ranges within which the values of relevant system parameters remain during operation. For instance, if your system is a cruising aircraft, then the operating range might be an incidence angle between –20° and 20° and airspeed in the range 200-250 m/s.
- Some measurable variables that indicate where in the operating range the system is at a given time. These signals are the scheduling variables. For the aircraft system, the scheduling variables might be the incidence angle and the airspeed.
- A gain schedule, which comprises the formulas or data tables that return the appropriate controller gains for given values of the scheduling variables. For the aircraft system, the gain schedule gives appropriate controller gains for any combination of incidence angle and airspeed within the operating range.

Gain Scheduling in Simulink

Control System Toolbox provides blocks that help you model gain-scheduled control systems in Simulink. These blocks let you implement common control-system elements with variable parameters. For instance, the Varying PID Controller block accepts PID gains as inputs. In your model, you use blocks such as n-D Lookup Table or MATLAB Function blocks to implement the gain schedule. For more information and examples, see “Model Gain-Scheduled Control Systems in Simulink” on page 11-4.

Tune Gain Schedules

You can use `systune` to tune gain schedules to achieve a control system that meets performance objectives across the entire operating range. For more information, see “Tune Gain Schedules in Simulink” on page 11-12.

References

[1] Rugh, W.J., and J.S. Shamma, “Research on Gain Scheduling”, *Automatica*, 36 (2000), pp. 1401-1425.

See Also

More About

- “Model Gain-Scheduled Control Systems in Simulink” on page 11-4
- “Tune Gain Schedules in Simulink” on page 11-12

Model Gain-Scheduled Control Systems in Simulink

In Simulink, you can model gain-scheduled control systems in which controller gains or coefficients depend on scheduling variables such as time, operating conditions, or model parameters. The library of linear parameter-varying blocks in Control System Toolbox lets you implement common control-system elements with variable gains. Use blocks such as lookup tables or MATLAB Function blocks to implement the gain schedule, which gives the dependence of these gains on the scheduling variables.

To model a gain-scheduled control system in Simulink:

1. Identify the scheduling variables and the signals that represent them in your model. For instance, if your system is a cruising aircraft, then the scheduling variables might be the incidence angle and the airspeed of the aircraft.
2. Use a lookup table block or a MATLAB Function block to implement a gain or coefficient that depends on the scheduling variables. If you do not have lookup table values or MATLAB expressions for gain schedules that meet your performance requirements, you can use `systune` to tune them. See “Tune Gain Schedules in Simulink” on page 11-12.
3. Replace ordinary control elements with gain-scheduled elements. For instance, instead of a fixed-coefficient PID controller, use a Varying PID Controller block, in which the gain schedules determine the PID gains.
4. Add scheduling logic and safeguards to your model as needed.

Model Scheduled Gains

A gain schedule converts the current values of the scheduling variables into controller gains. There are several ways to implement a gain schedule in Simulink.

Available blocks for implementing lookup tables include:

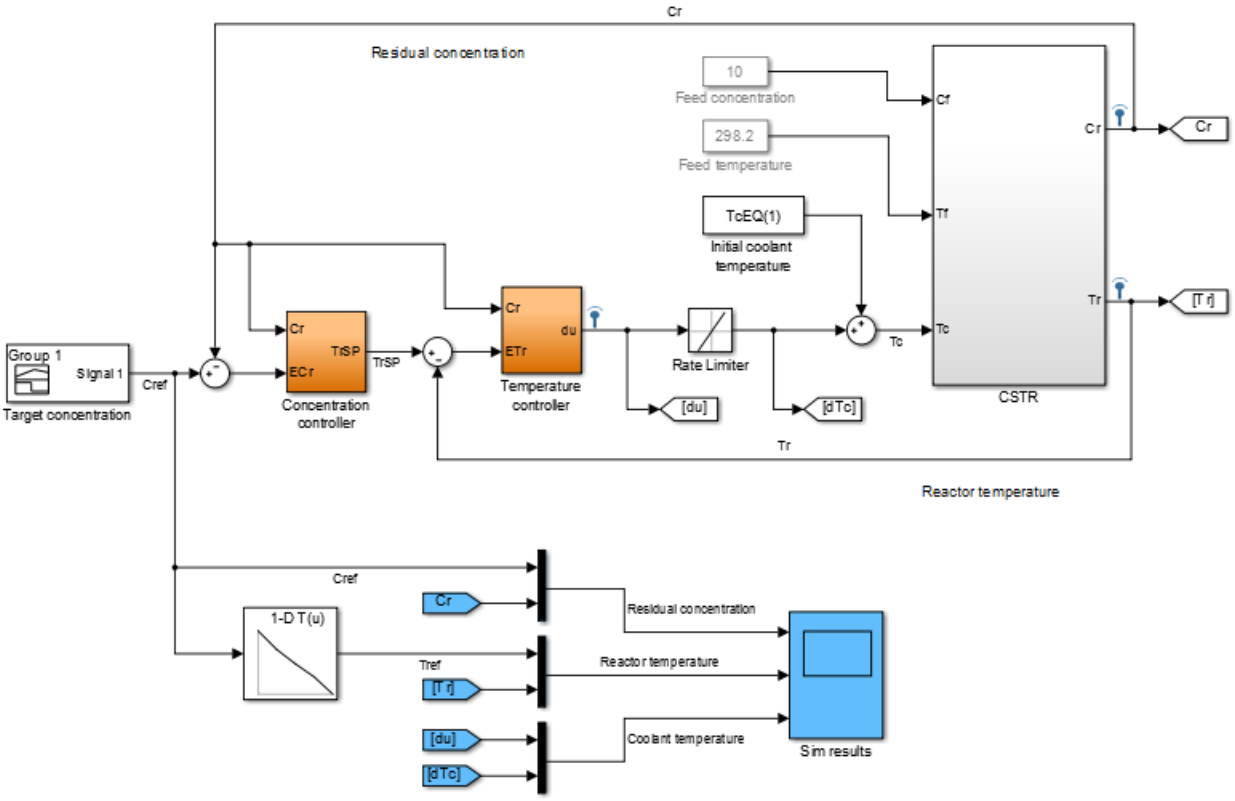
- Lookup tables — A lookup table is a list of breakpoints and corresponding gain values. When the scheduling variables fall between breakpoints, the lookup table interpolates between the corresponding gains. Use the following blocks to implement gain schedules as lookup tables.
  - 1-D Lookup Table, 2-D Lookup Table, n-D Lookup Table — For a scalar gain that depends on one, two, or more scheduling variables.
  - Matrix Interpolation — For a matrix-valued gain that depends on one, two, or three scheduling variables. (This block is in the **Simulink Extras** library.)
- MATLAB Function block — When you have a functional expression relating the gains to the scheduling variables, use a MATLAB Function block. If the expression is a smooth function, using a MATLAB function can result in smoother gain variations than a lookup table. Also, if you use a code-generation product such as Simulink Coder to implement the controller in hardware, a MATLAB function can result in a more memory-efficient implementation than a lookup table.

You can use `systune` to tune gain schedules implement as either lookup tables or MATLAB functions. See “Tune Gain Schedules in Simulink” on page 11-12.

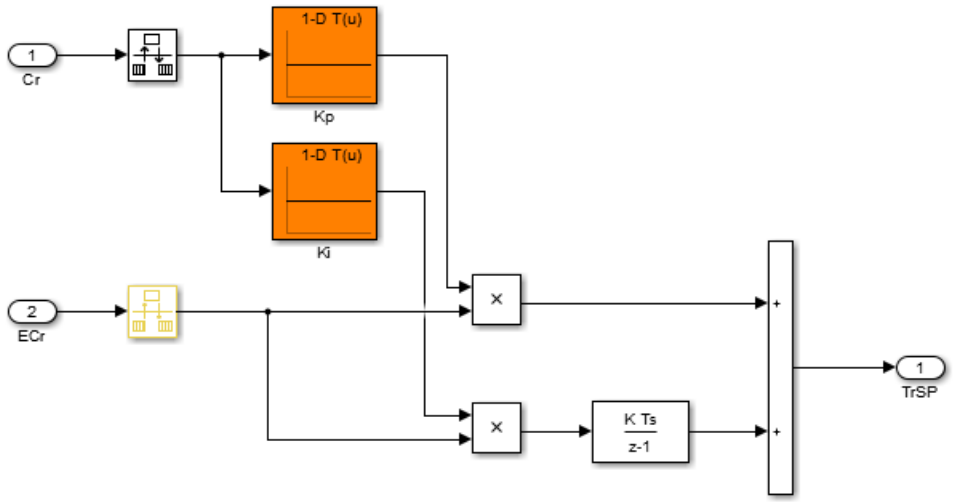
Scheduled Gain in Controller

As an example, The model `rct_CSTR` includes a PI controller and a lead compensator in which the controller gains are implemented as lookup tables using 1-D Lookup Table blocks. Open that model and examine the controllers.

```
open_system(fullfile(matlabroot,'examples','controls_id','main','rct_CSTR.slx'))
```



Both the `Concentration controller` and `Temperature controller` blocks take the `CSTR` plant output, `Cr`, as an input. This value is both the controlled variable of the system and the scheduling variable on which the controller action depends. Double-click the `Concentration controller` block.



Gain-scheduled PID controller Kp + Ki * Ts/(z-1)

This block is a PI controller in which the proportional gain `Kp` and integrator gain `Ki` are determined by feeding the scheduling parameter `Cr` into a 1-D Lookup Table block. Similarly, the `Temperature controller` block contains three gains implemented as lookup tables.

Gain-Scheduled Equivalents for Commonly Used Control Elements

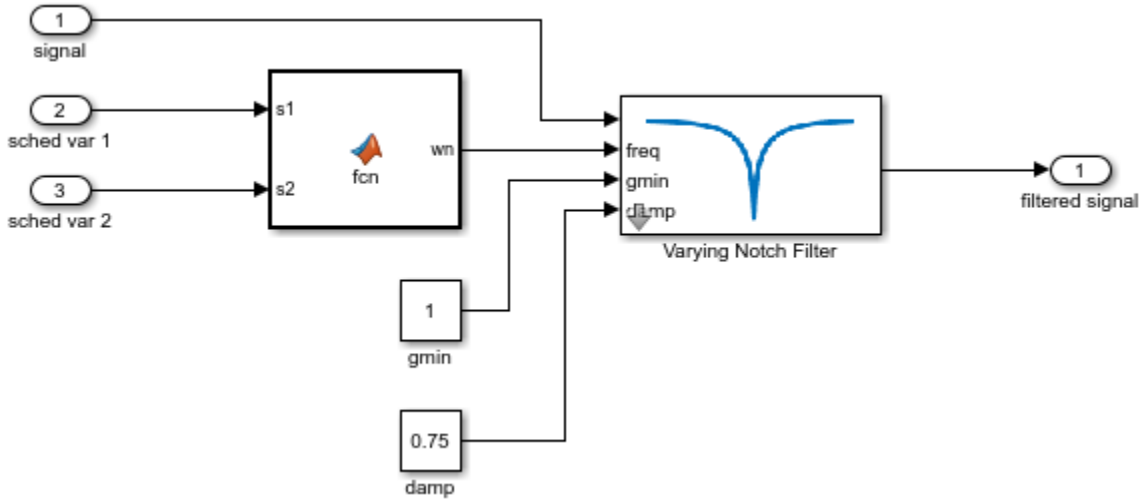
Use the **Linear Parameter Varying** block library of Control System Toolbox to implement common control elements with variable parameters or coefficients. These blocks provide common elements in which the gains or parameters are available as external inputs. The following table lists some applications of these blocks.

Block	Application
• Varying Lowpass Filter • Discrete Varying Lowpass	Use these blocks to implement a Butterworth lowpass filter in which the cutoff frequency varies with scheduling variables.
• Varying Notch Filter • Discrete Varying Notch	Use these blocks to implement a notch filter in which the notch frequency, width, and depth vary with scheduling variables.
• Varying PID Controller • Discrete Varying PID • Varying 2DOF PID • Discrete Varying 2DOF PID	These blocks are preconfigured versions of the PID Controller and PID Controller (2DOF) blocks. Use them to implement PID controllers in which the PID gains vary with scheduling variables.
• Varying Transfer Function • Discrete Varying Transfer Function	Use these blocks to implement a transfer function of any order in which the polynomial coefficients of the numerator and denominator vary with scheduling variables.

Block	Application
• Varying State Space • Discrete Varying State Space	Use these blocks to implement a state-space controller in which the A, B, C, and D matrices vary with the scheduling variables.
• Varying Observer Form • Discrete Varying Observer Form	Use these blocks to implement a gain-scheduled observer-form state-space controller, such as an LQG controller. In such a controller, the A, B, C, D matrices and the state-feedback and state-observer gain matrices vary with the scheduling variables.

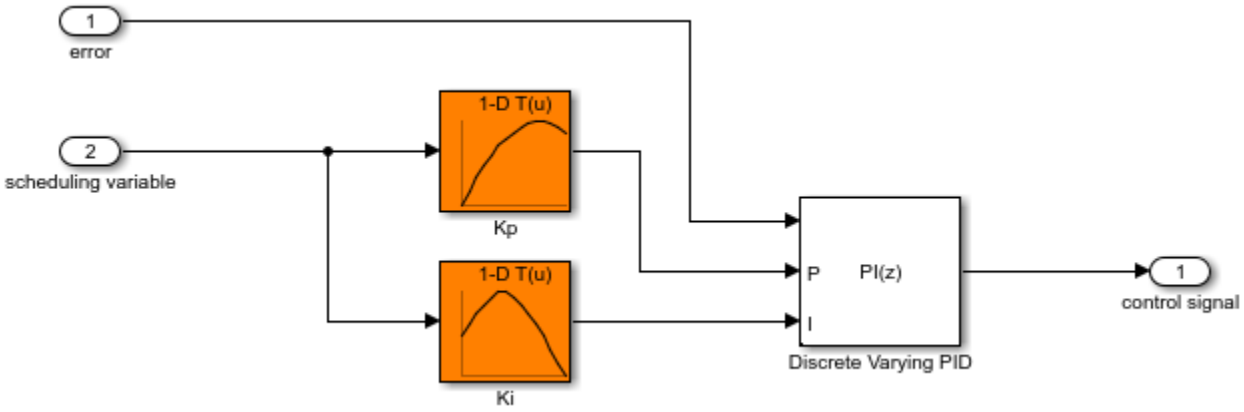
Gain-Scheduled Notch Filter

For example, the subsystem in the following illustration uses a Varying Notch Filter block to implement a filter whose notch frequency varies as a function of two scheduling variables. The relationship between the notch frequency and the scheduling variables is implemented in a MATLAB function.



Gain-Scheduled PI Controller

As another example, the following subsystem is a gain-scheduled discrete-time PI controller in which both the proportional and integral gains depend on the same scheduling variable. This controller uses 1-D Lookup Table blocks to implement the gain schedules.



Matrix-Valued Gain Schedules

You can also implement matrix-valued gain schedules Simulink. A matrix-valued gain schedule takes one or more scheduling variables and returns a matrix rather than a scalar value. For instance, suppose that you want to implement a time-varying LQG controller of the form:

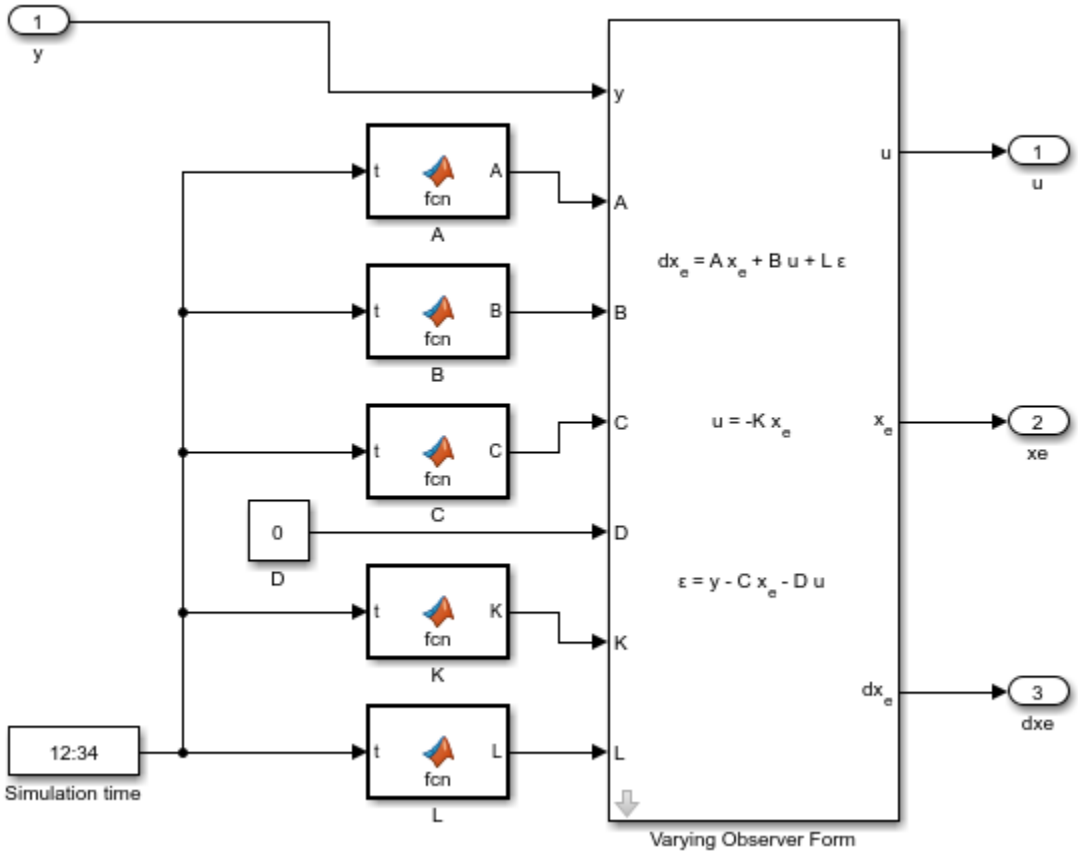
$$dx_e = Ax_e + Bu + L(y - Cx_e - Du)$$
$$u = -Kx_e,$$

where, in general, the state-space matrices A, B, C, and D, the state-feedback matrix K, and the observer-gain matrix L all vary with time. In this case, time is the scheduling variable, and the gain schedule determines the values of the matrices at a given time.

In your Simulink model, you can implement matrix-valued gain schedules using:

- MATLAB Function block — Specify a MATLAB function that takes scheduling variables and returns matrix values.
- Matrix Interpolation block — Specify a lookup table to associate a matrix value with each scheduling-variable breakpoint. Between breakpoints, the block interpolates the matrix elements. (This block is in the **Simulink Extras** library.)

For the LQG controller, use either MATLAB Function blocks or Matrix Interpolation blocks to implement the time-varying matrices as inputs to a Varying Observer Form block. For example:

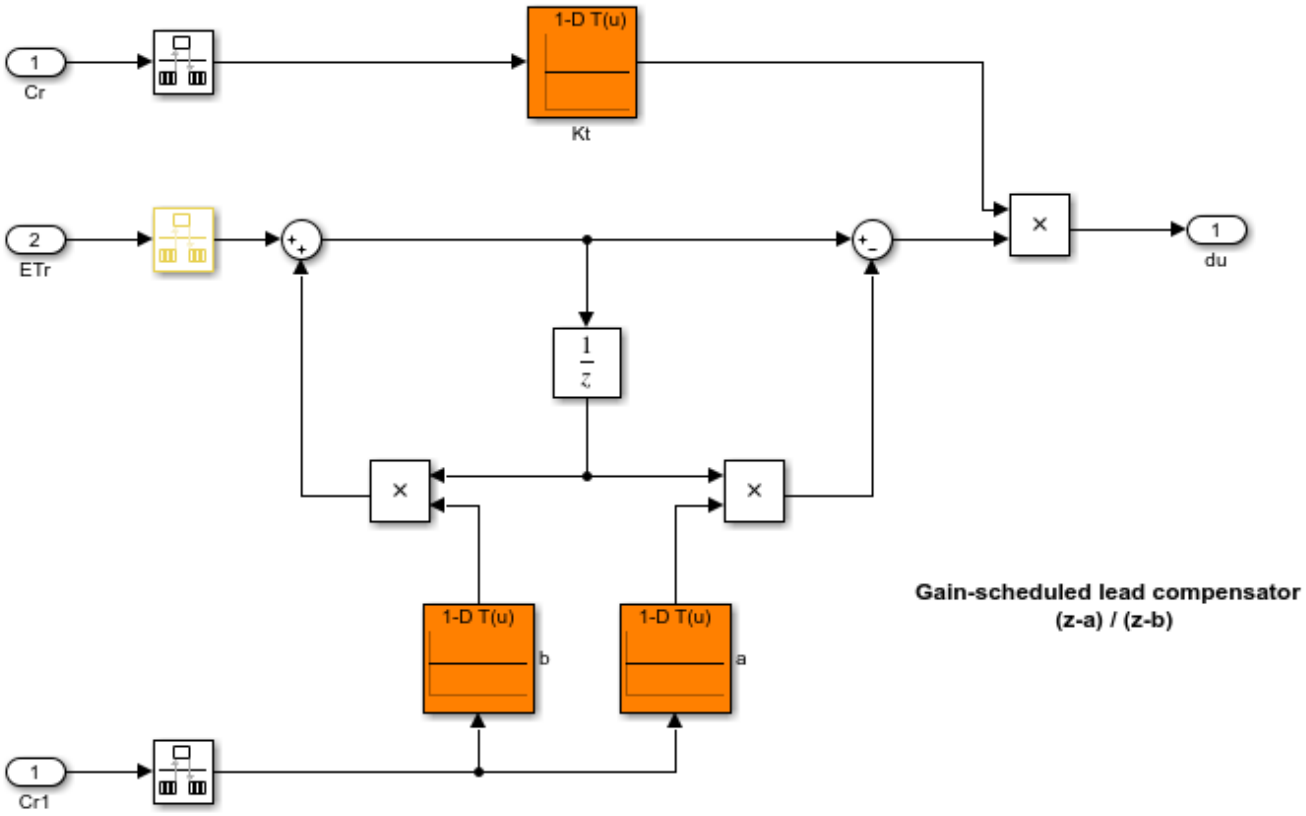


In this implementation, the time-varying matrices are each implemented as a MATLAB Function block in which the associated function takes the simulation time and returns a matrix of appropriate dimensions.

You can tune matrix-valued gain schedules implemented as either MATLAB Function blocks or as Matrix Interpolation blocks. However, to tune a Matrix Interpolation block, you must set **Simulate using** to `Interpreted execution`. See the Matrix Interpolation block reference page for information about simulation modes.

Custom Gain-Scheduled Control Structures

You can also use the scheduled gains to build your own control elements. For example, the model `rct_CSTR` includes a gain-scheduled lead compensator with three coefficients that depend on the scheduling variable, `CR`. To see how this compensator is implemented, open the model and examine the `Temperature controller` subsystem.



Here, the overall gain `Kt`, the zero location `a`, and the pole location `b` are each implemented as a 1-D lookup table that takes the scheduling variable as input. The lookup tables feed directly into product blocks.

Tunability of Gain Schedules

For a lookup table or MATLAB Function block that implements a gain schedule to be tunable with `systune`, it must ultimately feed into either:

- A block in the Linear Parameter Varying block library.
- A Product block that applies the gain to a given signal. For instance, if the Product block takes as inputs a scheduled gain $g(\alpha)$ and a signal $u(t)$, then the output signal of the block is $y(t) = g(\alpha)u(t)$.

There can be one or more of the following blocks between the lookup table or MATLAB Function block and the Product block or parameter-varying block:

- Gain
- Bias
- Blocks that are equivalent to a unit gain in the linear domain, including:
  - Transport Delay, Variable Transport Delay
  - Saturate, Deadzone

  - Rate Limiter, Rate Transition
  - Quantizer, Memory, Zero-Order Hold
  - MinMax
  - Data Type Conversion
  - Signal Specification
- Switch blocks, including:
  - Switch
  - Multiport Switch
  - Manual Switch

Inserting such blocks can be useful, for example, to constrain the gain value to a certain range, or to specify how often the gain schedule is updated.

See Also

Related Examples

- “Tune Gain Schedules in Simulink” on page 11-12
- “Gain-Scheduled Control of a Chemical Reactor”

Tune Gain Schedules in Simulink

Typically, gain-scheduled controllers are fixed single-loop or multiloop control structures in which controller gains vary with operating condition. A gain schedule converts the scheduling variables that describe the current operating condition into appropriate controller gains. In Simulink, you can implement gain schedules using lookup tables or MATLAB functions. (See “Model Gain-Scheduled Control Systems in Simulink” on page 11-4.)

You can use `systune` to tune these gain schedules so that the full nonlinear system meets your design requirements. Tuning gain schedules amounts to identifying appropriate values for lookup-table data or finding the right function to embed in a MATLAB Function block. For `systune`, you parameterize the gain schedules as functions of the scheduling variables with tunable coefficients.

Workflow for Tuning Gain Schedules

The general workflow for tuning gain-scheduled control systems is:

1. Select a set of design points that adequately covers the operating range over which you are tuning. A design point is a set of scheduling-variable values that describe a particular operating condition. The set of design points can be a regular grid of values or a scattered set. Typically, you start with a few design points. If the performance that your tuned system achieves at the design points is not maintained between design points, add more design points and retune.
2. Obtain a collection of linear models describing the linearized plant dynamics at the selected design points. Ways to obtain the array of linear models include:

   - Linearize a Simulink model at each operating condition represented in the grid of design points. For example, if each design point corresponds to a steady-state operating condition, you can trim the plant at each design point and linearize at the resulting operating point. Or, if your scheduling variable is time, you can linearize at a series of simulation snapshots.
   - Sample an LPV model of the plant at the design points.

   For more information, see “Plant Models for Gain-Scheduled Controller Tuning” on page 11-14.
3. Create an `slTuner` interface for tuning the Simulink. When you do so, you substitute the array of linear models for the plant, so that the `slTuner` interface contains a set of closed-loop tunable models corresponding to each design point. For more information, see “Multiple Design Points in slTuner Interface” on page 11-20.
4. Model the gain schedules as parametric gain surfaces. A parametric gain surface is a basis-function expansion with tunable coefficients. For a vector σ of scheduling variables, such expansion is of the form:

   $$K(\sigma) = K_0 + K_1F_1(n(\sigma)) + ... + K_MF_M(n(\sigma)) .$$

   $n(\sigma)$ is a normalization function. For tuning with `systune`, you use `tunableSurface` to represent the parametric gain surface $K(\sigma)$. In the `slTuner` interface you create for tuning, use `setBlockParam` to associate the resulting gain surface with the block that represents the gain schedule. `systune` tunes the coefficients $K_0,...,K_M$ over all the design points.

   For more information, see “Parameterize Gain Schedules” on page 11-24.
5. Specify your tuning goals using `TuningGoal` objects. You can specify tuning goals that apply at all design points or at a subset of design points. You can also specify tuning goals that vary from

design point to design point. For example, you might define a minimum gain margin that becomes increasingly stringent as a particular scheduling variable increases in magnitude.

For information about specifying tuning goals that vary with design point, see “Change Requirements with Operating Condition” on page 11-33.

For information about specifying tuning goals generally, see “Tuning Goals”.

6. Use `systune` to tune the control system. `systune` tunes the set of parameters, $K_0,...,K_M$, against all plant models in the design grid simultaneously (multimodel tuning).
7. Validate the tuning results. You can examine the tuned gain surfaces and validate the performance of the linearized system at each design point. However, local linear performance does not guarantee global performance in nonlinear systems. Therefore, it is important to perform simulation-based validation using the tuned gain schedules.

   For more information, see “Validate Gain-Scheduled Control Systems” on page 11-36.

See Also

More About

- “Model Gain-Scheduled Control Systems in Simulink” on page 11-4
- “Gain-Scheduled Control of a Chemical Reactor”
- “Tuning of Gain-Scheduled Three-Loop Autopilot”

Plant Models for Gain-Scheduled Controller Tuning

Gain scheduling is a control approach for controlling a nonlinear plant. To tune a gain-scheduled control system, you need a collection of linear models that approximate the nonlinear dynamics near selected design points. Generally, the dynamics of the plant are described by nonlinear differential equations of the form:

$$\dot{x} = f(x, u, \sigma)$$
$$y = g(x, u, \sigma)\,.$$

Here, x is the state vector, u is the plant input, and y is the plant output. These nonlinear differential equations can be known explicitly for a particular system. More commonly, they are specified implicitly, such as by a Simulink model.

You can convert these nonlinear dynamics into a family of linear models that describe the local behavior of the plant around a family of operating points ($x(\sigma)$,$u(\sigma)$), parameterized by the scheduling variables, σ. Deviations from the nominal operating condition are defined as:

$$\delta x = x - x(\sigma), \quad \delta u = u - u(\sigma)\,.$$

These deviations are governed, to first order, by linear parameter-varying dynamics:

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

$$A(\sigma) = \frac{\partial f}{\partial x}(x(\sigma), u(\sigma)) \quad B(\sigma) = \frac{\partial f}{\partial u}(x(\sigma), u(\sigma))$$

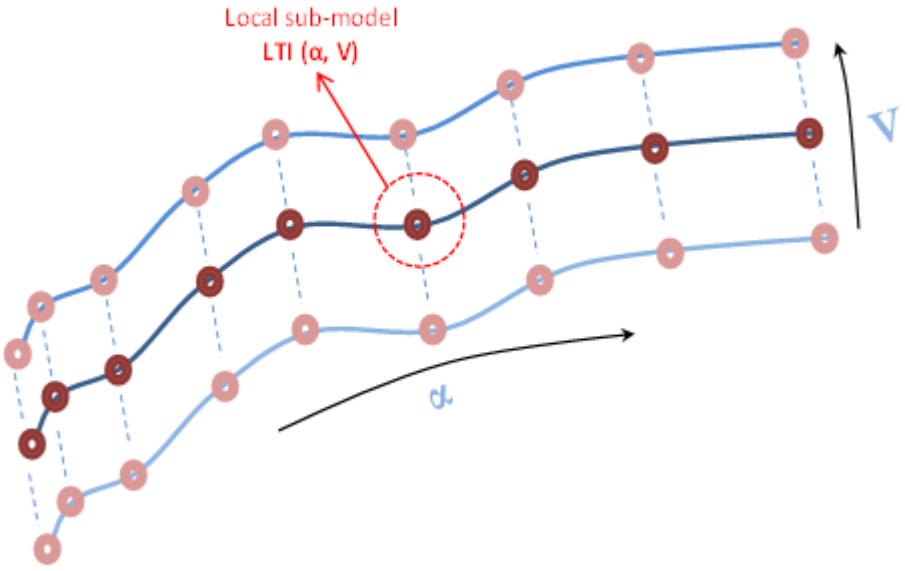
$$C(\sigma) = \frac{\partial g}{\partial x}(x(\sigma), u(\sigma)) \quad D(\sigma) = \frac{\partial g}{\partial u}(x(\sigma), u(\sigma))\,.$$

This continuum of linear approximations to the nonlinear dynamics is called a linear parameter-varying (LPV) model:

$$\frac{dx}{dt} = A(\sigma)x + B(\sigma)u$$
$$y = C(\sigma)x + D(\sigma)u\,.$$

The LPV model describes how the linearized plant dynamics vary with time, operating condition, or any other scheduling variable. For example, the pitch axis dynamics of an aircraft can be approximated by an LPV model that depends on incidence angle, α, air speed, V, and altitude, h.

In practice, you replace this continuum of plant models by a finite set of linear models obtained for a suitable grid of σ values This replacement amounts to sampling the LPV dynamics over the operating range and selecting a representative set of σ values, your design points.



Gain-scheduled controllers yield best results when the plant dynamics vary smoothly between design points.

Obtaining the Family of Linear Models

If you do not have this family of linear models, there are several approaches to obtaining it, including:

- If you have a Simulink model, trim and linearize the model at the design points on page 11-15.
- Linearize the Simulink model using parameter variation on page 11-18.
- If the scheduling variable is time, linearize the model at a series of simulation snapshots on page 11-18.
- If you have nonlinear differential equations that describe the plant, linearize them at the design points.

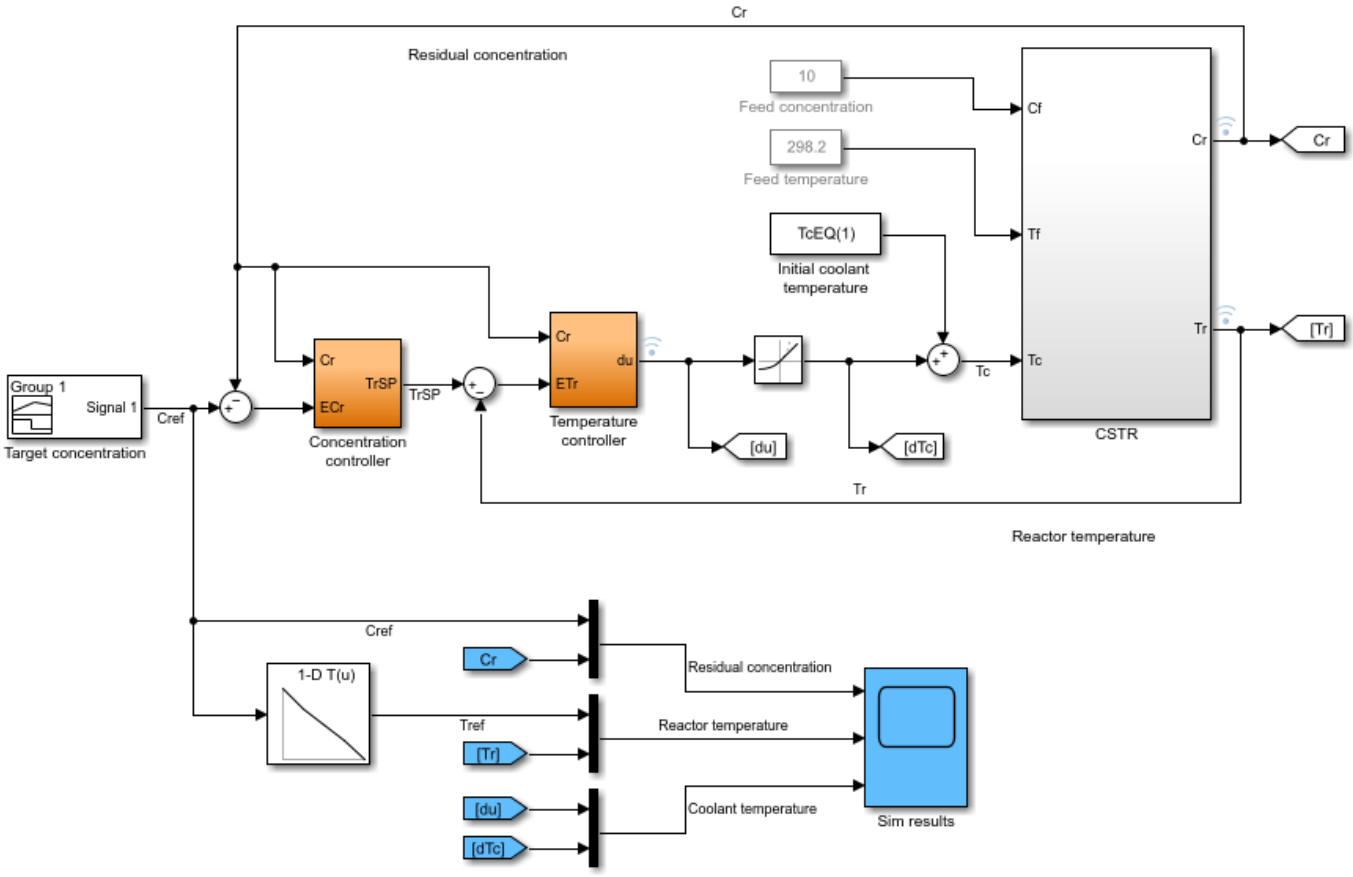
For tuning gain schedules, after you obtain the family of linear models, you must associate it with an `slTuner` interface to build a family of tunable closed-loop models. To do so, use block substitution, as described in “Multiple Design Points in slTuner Interface” on page 11-20.

Set Up for Gain Scheduling by Linearizing at Design Points

This example shows how to linearize a plant model at a set of design points for tuning of a gain-scheduled controller. The example then uses the resulting linearized models to configure an `slTuner` interface for tuning the gain schedule.

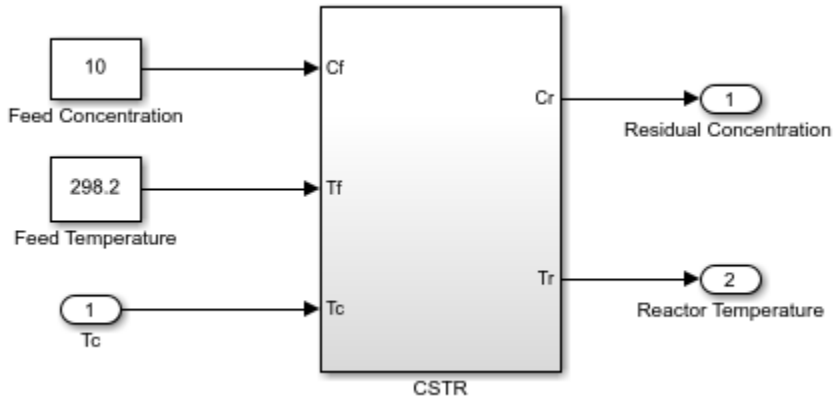
Open the `rct_CSTR` model.

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



In this model, the `Concentration controller` and `Temperature controller` both depend on the output concentration `Cr`. To set up this gain-scheduled system for tuning, you linearize the plant at a set of steady-state operating points that correspond to different values of the scheduling parameter `Cr`. Sometimes, it is convenient to use a separate model of the plant for trimming and linearization under various operating conditions. For example, in this case, the most straightforward way to obtain these linearizations is to use a separate open-loop model of the plant, `rct_CSTR_OL`.

```
mdl_OL = 'rct_CSTR_OL';
open_system(mdl_OL)
```



Trim Plant at Design Points

Suppose that you want to control this plant at a range of `Cr` values from 4 to 8. Trim the model to find steady-state operating points for a set of values in this range. These values are the design points for tuning.

```
Cr = (4:8)';        % concentrations
for k=1:length(Cr)
   opspec = operspec(mdl_OL);
   % Set desired residual concentration
   opspec.Outputs(1).y = Cr(k);
   opspec.Outputs(1).Known = true;
   % Compute equilibrium condition
   [op(k),report(k)] = findop(mdl_OL,opspec,findopOptions('DisplayReport','off'));
end
```

`op` is an array of steady-state operating points. For more information about steady-state operating points, see “About Operating Points” on page 1-2.

Linearize at Design Points

Linearizing the plant model using `op` returns an array of LTI models, each linearized at the corresponding design point.

```
G = linearize(mdl_OL,'rct_CSTR_OL/CSTR',op);
```

Create `slTuner` Interface with Block Substitution

To tune the control system `rct_CSTR`, create an `slTuner` interface that linearizes the system at those design points. Use block substitution to replace the plant in `rct_CSTR` with the linearized plant-model array `G`.

```
blocksub.Name = 'rct_CSTR/CSTR';
blocksub.Value = G;
tunedblocks = {'Kp','Ki'};
ST0 = slTuner(mdl,tunedblocks,blocksub);
```

For this example, only the PI coefficients in the `Concentration controller` are designated as tuned blocks. In general, however, `tunedblocks` lists all the blocks to tune.

For more information about using block substitution to configure an `slTuner` interface for gain-scheduled controller tuning, see “Multiple Design Points in slTuner Interface”.

For another example that illustrates using trimming and linearization to generate a family of linear models for gain-scheduled controller tuning, see “Trimming and Linearization of the HL-20 Airframe”.

Sample System at Simulation Snapshots

If you are controlling the system around a reference trajectory ($x(\sigma)$,$u(\sigma)$), use snapshot linearization to sample the system at various points along the σ trajectory. Use this approach for time-varying systems where the scheduling variable is time.

To linearize a system at a set of simulation snapshots, use a vector of positive scalars as the `op` input argument of `linearize`, `slLinearizer`, or `slTuner`. These scalars are the simulation times at which to linearize the model. Use the same set of time values as the design points in tunable surfaces for the system.

Sample System at Varying Parameter Values

If the scheduling variable is a parameter in the Simulink model, you can use parameter variation to sample the control system over a parameter grid. For example, suppose that you want to tune a model named `suspension_gs` that contains two parameters, `Ks` and `Bs`. These parameters each can vary over some known range, and a controller gain in the model varies as a function of both parameters.

To set up such a model for tuning, create a grid of parameter values. For this example, let `Ks` vary from 1 – 5, and let `Bs` vary from 0.6 – 0.9.

```
Ks = 1:5;
Bs = [0.6:0.1:0.9];
[Ksgrid,Bsgrid] = ndgrid(Ks,Bs);
```

These values are the design points at which to sample and tune the system. For example, create an `slTuner` interface to the model, assuming one tunable block, a Lookup Table block named `K` that models the parameter-dependent gain.

```
params(1) = struct('Name','Ks','Value',Ksgrid);
params(2) = struct('Name','Bs','Value',Bsgrid);
ST0 = slTuner('suspension_gs','K',params);
```

`slTuner` samples the model at all `(Ksgrid,Bsgrid)` values specified in `params`.

Next, use the same design points to create a tunable gain surface for parameterizing `K`.

```
design = struct('Ks',Ksgrid,'Bs',Bsgrid);
shapefcn = @(Ks,Bs)[Ks,Bs,Ks*Bs];
K = tunableSurface('K',1,design,shapefcn);
setBlockParam(ST0,'K',K);
```

After you parameterize all the scheduled gains, you can create your tuning goals and tune the system with `systune`.

Eliminate Samples at Unneeded Design Points

Sometimes, your sampling grid includes points that represent irrelevant or unphysical design points. You can eliminate such design points from the model grid entirely, so that they do not contribute to any stage of tuning or analysis. To do so, use `voidModel`, which replaces specified models in a model array with `NaN`. `voidModel` replaces specified models in a model array with `NaN`. Using `voidModel` lets your design over a grid of design points that is almost regular.

There are other tools for controlling which models contribute to design and analysis. For instance, you might want to:

- Keep a model in the grid for analysis, but exclude it from tuning.
- Keep a model in the grid for tuning, but exclude it from a particular design goal.

For more information, see “Change Requirements with Operating Condition” on page 11-33.

LPV Plants in MATLAB

In MATLAB, you can use an array of LTI plant models to represent an LPV system sampled at varying values of σ. To associate each linear model in the set with the underlying design points, use the `SamplingGrid` property of the LTI model array σ. One way to obtain such an array is to create a parametric generalized state-space (`genss`) model of the system and sample the model with parameter variation to generate the array. For an example, see “Study Parameter Variation by Sampling Tunable Model”.

See Also

`slTuner` | `findop` | `voidModel`

Related Examples

- “Parameterize Gain Schedules” on page 11-24
- “Tune Gain Schedules in Simulink” on page 11-12
- “Multiple Design Points in slTuner Interface” on page 11-20

Multiple Design Points in slTuner Interface

For tuning a gain-scheduled control system, you must make your Simulink model linearize to an array of LTI models corresponding to the various operating conditions that are your design points. Thus, after you obtain a family of linear plant models as described in “Plant Models for Gain-Scheduled Controller Tuning” on page 11-14, you must associate it with the `slTuner` interface to your Simulink model. To do so, you use block substitution to cause `slTuner` replace the plant subsystem of the model with the array of linear models. This process builds a family of tunable closed-loop models within the `slTuner` interface.

Block Substitution for Plant

Suppose that you have an array of linear plant models obtained at each operating point in your design grid. In the most straightforward case, the following conditions are met:

- The linear models in the array correspond exactly to the plant subsystem in your model.
- Other than the elements you want to tune, nothing else in the model varies with the scheduling variables.

For a Simulink model `mdl` containing plant subsystem `G`, and a linear model array `Garr` that represents the plant at a grid of design points, the following commands create an `slTuner` interface:

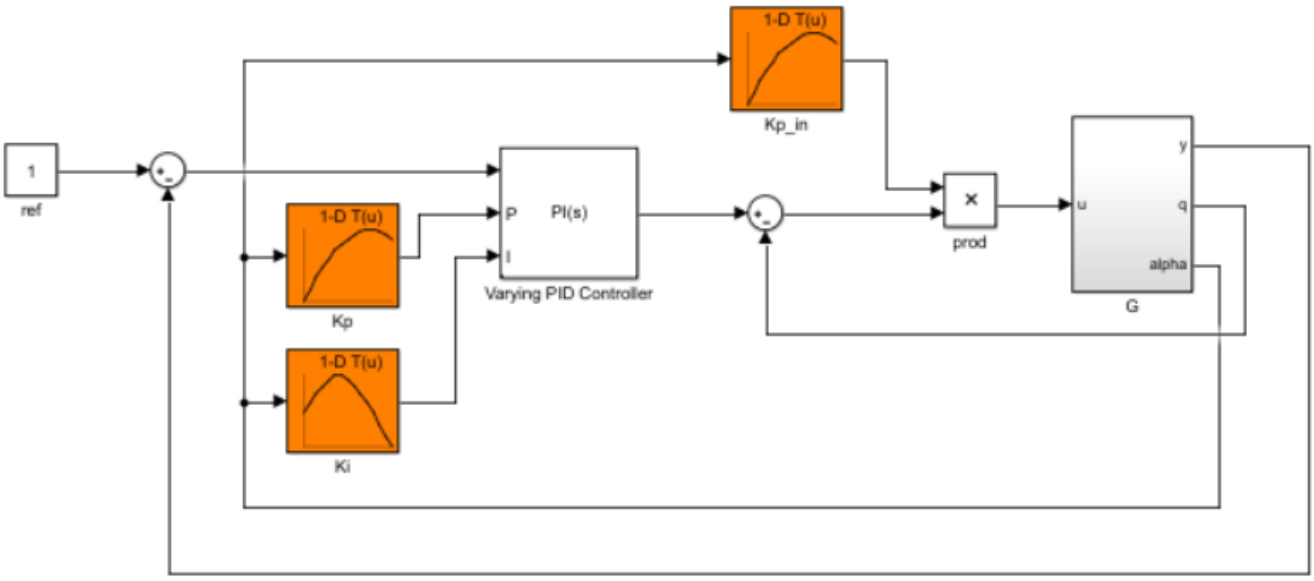
```
BlockSubs = struct('Name','mdl/G','Value',Garr);
st0 = slTuner('mdl',{'Kp','Ki'},BlockSubs);
```

`st0` contains a family of closed-loop linear models, each linearized at a design point, and each with the corresponding linear plant inserted for `G`. If `'Kp'` and `'Ki'` are the gain schedules you want to tune (such as lookup tables), you can parameterize them with tunable gain surfaces, as described in “Parameterize Gain Schedules” on page 11-24, and tune them.

Multiple Block Substitutions

In other cases, the linearized array of plant models you have might not correspond exactly to the plant subsystem in your Simulink model. Or, you might need to replace other parts of the model that vary with operating condition. In such cases, more care is needed in constructing the correct block substitution. The following sections highlight several such cases.

For instance, consider the model of the following illustration.



This model has an inner loop with a proportional-only gain-scheduled controller. The controller is represented by the lookup table `Kp_in` and the product block `prod`. The outer loop includes a PI controller with gain-scheduled proportional and integral coefficients represented by the lookup tables `Kp` and `Ki`. All the gain schedules depend on the same scheduling variable `alpha`.

Suppose you want to tune the inner-loop gain schedule `Kp_in` with the outer loop open. To that end, you obtain an array of linear models `G_in` from input `u` to outputs `{q,alpha}`. This model array has the wrong I/O dimensions to use as a block substitution for `G`. Therefore, you must "pad" `G_in` with an extra output dimension.

```
Garr = [0; G_in];
BlockSubs1 = struct('Name','mdl/G','Value',Garr);
```

In addition, you can remove all effect of the outer loop by replacing the Varying PID Controller block with a system that linearizes to zero at all operating conditions. Because this block has three inputs, replace it with a 3-input, one-output zero system.

```
BlockSubs2 = struct('Name','mdl/Varying PID Controller','Value',ss([0 0 0]));
```

With those block substitutions, the following commands create an `slTuner` interface that you might use to tune the inner-loop gain schedule.

```
st0 = slTuner('mdl','Kp_in');
st0.BlockSubstitutions = [BlockSubs1; BlockSubs2];
```

See the example “Angular Rate Control in the HL-20 Autopilot” on page 11-75 for a another case in which several elements other than the plant itself are replaced by block substitution.

Substituting Blocks that Depend on the Scheduling Variables

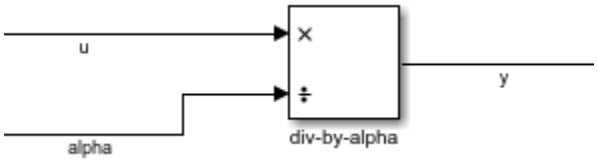
Next, suppose that you have already tuned the inner-loop gain schedule, and have obtained an array `Kp_in_tuned`, of values of `Kp_in` that correspond to each design point (each value of `alpha` at

which you linearized the plant). Suppose also that you have a new `Garr` that is the full plant from `u` to `{y,q,alpha}` linearized with the tuned inner loop closed. To tune the outer-loop gain schedules, you must replace the product block with the array `Kp_in_tuned`. It is important to note that you replace the injection point, the product block `prod`, rather than the lookup table `Kp_in`. Replacing the product block effectively converts it to a varying gain. Also, you must zero out the first input of the product block to remove the effect of the lookup table `Kp_in`.

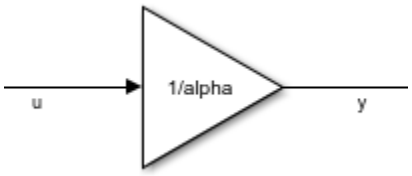
```
prodsub = [0 ss(Kp_in_tuned)];
BlockSubs1 = struct('Name','mdl/prod','Value',prodsub);
BlockSubs2 = struct('Name','mdl/G','Value',Garr);

st0 = slTuner('mdl',{'Kp','Ki'});
st0.BlockSubstitutions = [BlockSubs1; BlockSubs2];
```

For another example that shows this kind of substitution for a previously-tuned lookup table, see “Attitude Control in the HL-20 Autopilot - SISO Design” on page 11-81.

The following illustration of a portion of a model highlights another scenario in which you might need to replace blocks that vary with the scheduling variable. Suppose the scheduling variable is `alpha`, and somewhere in your model, an signal `u` gets divided by `alpha`.



To ensure that `slTuner` linearizes this block correctly at all values of `alpha` in the design grid, you must replace it by an array of linear models, one for each `alpha` value. This block is equivalent to sending `u` through a gain of 1/`alpha`:



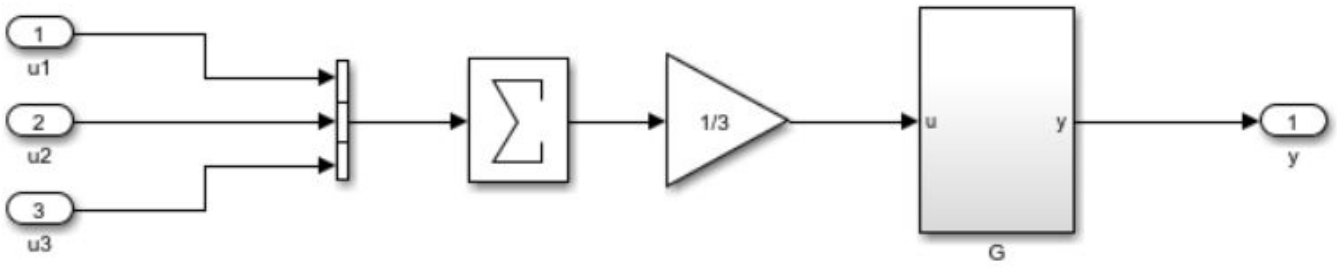
Therefore, you can use the following block substitution in your `slTuner` interface, where `alphagrid` is an array of `alpha` values at your design points.

```
divsub = ss[(1/alphagrid), 0]
BlockSubs = struct('Name','mdl/div-by-alpha','Value',divsub);
st0.BlockSubstitutions = [st0.BlockSubstitutions; BlockSubs]
```

Each entry in model array `divsub` divides its first input by the corresponding entry in `alphagrid`, and zeros out its second input. Thus, this substitution gives the desired result `y = u/alpha`.

Resolving Mismatches Between a Block and its Substitution

Sometimes, the linear model array you have is not an exact replacement for the part of the model you want to replace. For example, consider the following illustration of a three-input, one-output subsystem.



G_full

Suppose you have an array of linearized models `Garr` corresponding to `G`. You can configure a block substitution for the entire subsystem `G_full` by constructing a substitution model that reproduces the effect of averaging the three inputs, as follows:

```
Gsub = Garr*[1/3 1/3 1/3];
BlockSubs = struct('Name','mdl/G_full','Value',Gsub);
```

Sometimes, you can resolve a mismatch in I/O dimensions by padding inputs or outputs with zeros, as shown in “Multiple Block Substitutions” on page 11-20. In still other cases, you might need to perform other model arithmetic, using commands like `series`, `feedback`, or `connect` to build a suitable replacement.

Block Substitution for LPV Blocks

If the plant in your Simulink model is represented by an LPV System , you must still perform block substitution when creating the `slTuner` interface for tuning gain schedules. `slTuner` cannot read the linear model array directly from the LPV System block. However, you can use the linear model array specified in the block for the block substitution, if it corresponds to the design points for which you are tuning. For instance, suppose your plant is an LPV System block, `LPVPlant`, that specifies a model array `PlantArray`. You can configure a block substitution for `LPVPlant` as follows:

```
BlockSubs = struct('Name','mdl/LPVPlant','Value',PlantArray);
```

See Also

`slTuner`

More About

- “Tune Gain Schedules in Simulink” on page 11-12
- “Plant Models for Gain-Scheduled Controller Tuning” on page 11-14
- “Parameterize Gain Schedules” on page 11-24

Parameterize Gain Schedules

Typically, gain-scheduled control systems in Simulink use lookup tables or MATLAB Function blocks to specify gain values as a function of the scheduling variables. For tuning, you replace these blocks by parametric gain surfaces. A parametric gain surface is a basis-function expansion whose coefficients are tunable. For example, you can model a time-varying gain $k(t)$ as a cubic polynomial in t:

$$k(t) = k_0 + k_1 t + k_2 t^2 + k_3 t^3.$$

Here, $k_0,\ldots,k_3$ are tunable coefficients. When you parameterize scheduled gains in this way, `systune` can tune the gain-surface coefficients to meet your control objectives at a representative set of operating conditions. For applications where gains vary smoothly with the scheduling variables, this approach provides explicit formulas for the gains, which the software can write directly to MATLAB Function blocks. When you use lookup tables, this approach lets you tune a few coefficients rather than many individual lookup-table entries, drastically reducing the number of parameters and ensuring smooth transitions between operating points.

Basis Function Parameterization

In a gain-scheduled controller, the scheduled gains are functions of the scheduling variables, σ. For example, a gain-scheduled PI controller has the form:

$$C(s,\sigma) = K_p(\sigma) + \frac{K_i(\sigma)}{s}.$$

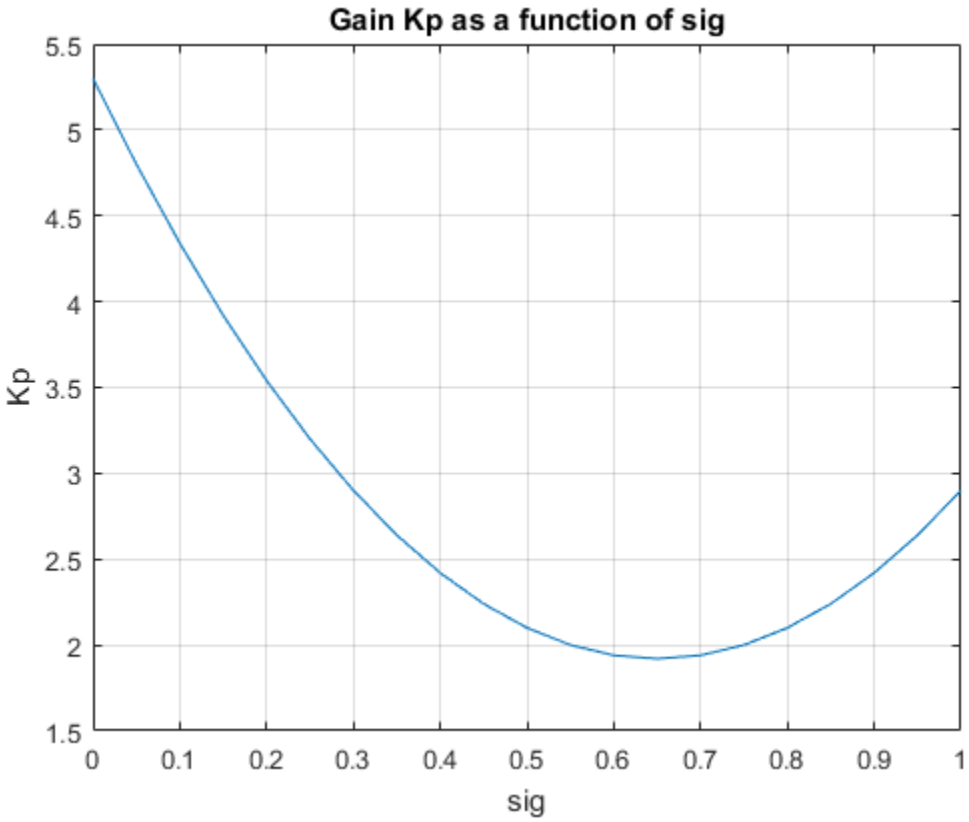
Tuning this controller requires determining the functional forms $K_p(\sigma)$ and $K_i(\sigma)$ that yield the best system performance over the operating range of σ values. However, tuning arbitrary functions is difficult. Therefore, it is necessary either to consider the function values at only a finite set of points, or restrict the generality of the functions themselves.

In the first approach, you choose a collection of design points, σ, and tune the gains K_p and K_i independently at each design point. The resulting set of gain values is stored in a lookup table driven by the scheduling variables, σ. A drawback of this approach is that tuning might yield substantially different values for neighboring design points, causing undesirable jumps when transitioning from one operating point to another.

Alternatively, you can model the gains as smooth functions of σ, but restrict the generality of such functions by using specific basis function expansions. For example, suppose σ is a scalar variable. You can model $K_p(\sigma)$ as a quadratic function of σ:

$$K_p(\sigma) = k_0 + k_1\sigma + k_2\sigma^2.$$

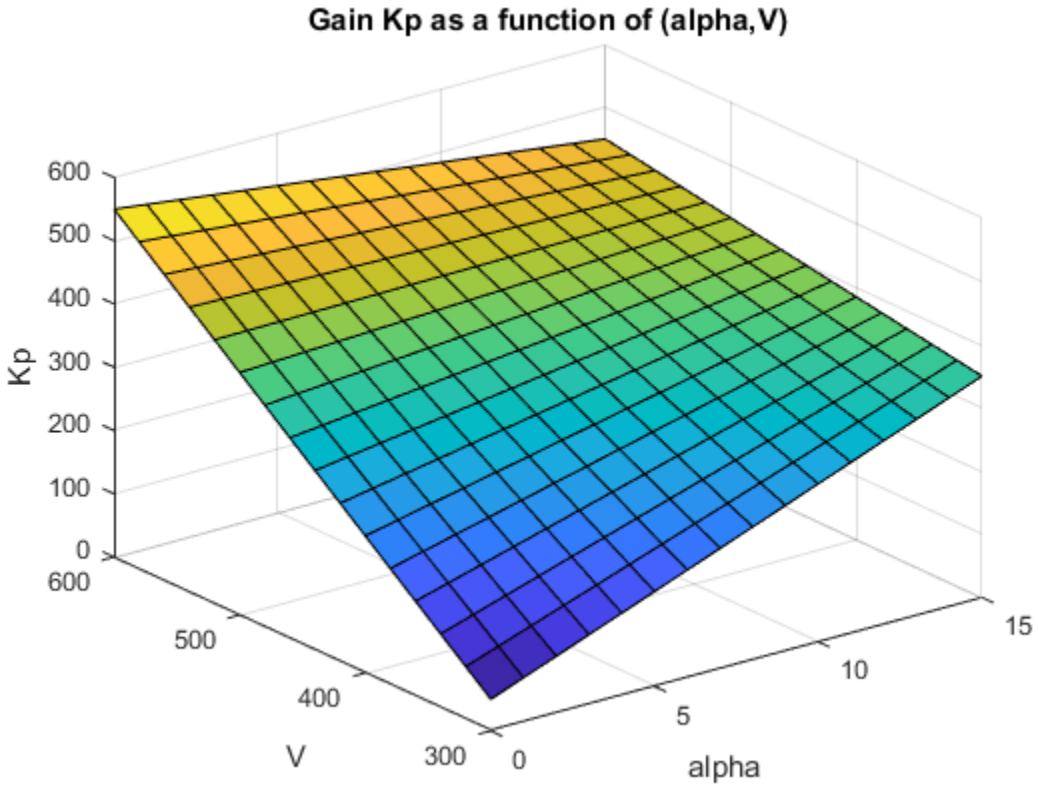
After tuning, this parametric gain might have a profile such as the following (the specific shape of the curve depends on the tuned coefficient values and range of σ):



Or, suppose that σ consists of two scheduling variables, α and V. Then, you can model $K_p(\sigma)$ as a bilinear function of α and V:

$$K_p(\alpha, V) = k_0 + k_1\alpha + k_2V + k_3\alpha V\,.$$

After tuning, this parametric gain might have a profile such as the following. Here too, the specific shape of the curve depends on the tuned coefficient values and ranges of σ values:



For tuning gain schedules with `systune`, you use a parametric gain surface that is a particular expansion of the gain in basis functions of σ:

$$K(\sigma) = K_0 + K_1F_1(n(\sigma)) + ... + K_MF_M(n(\sigma)) .$$

The basis functions $F_1,...,F_M$ are user-selected and fixed. These functions operate on $n(\sigma)$, where n is a function that scales and normalizes the scheduling variables to the interval [–1,1] (or an interval you specify). The coefficients of the expansion, $K_0,...,K_M$, are the tunable parameters of the gain surface. $K_0,...,K_M$ can be scalar or matrix-valued, depending on the I/O size of the gain $K(\sigma)$. The choice of basis function is problem-dependent, but in general, try low-order polynomial expansions first.

Tunable Gain Surfaces

Use the `tunableSurface` command to construct a tunable model of a gain surface sampled over a grid of design points (σ values). For example, consider the gain with bilinear dependence on two scheduling variables, α and V:

$$K_p(\alpha, V) = K_0 + K_1\alpha + K_2V + K_3\alpha V .$$

Suppose that α is an angle of incidence that ranges from 0° to 15°, and V is a speed that ranges from 300 m/s to 700 m/s. Create a grid of design points that span these ranges. These design points must match the parameter values at which you sample your varying or nonlinear plant. (See “Plant Models for Gain-Scheduled Controller Tuning” on page 11-14.)

```
[alpha,V] = ndgrid(0:5:15,300:100:700);
domain = struct('alpha',alpha,'V',V);
```

Specify the basis functions for the parameterization of this surface, α, V, and αV. The `tunableSurface` command expects the basis functions to be arranged as a vector of functions of two input variables. You can use an anonymous function to express the basis functions.

```
shapefcn = @(alpha,V)[alpha,V,alpha*V];
```

Alternatively, use `polyBasis`, `fourierBasis`, or `ndBasis` to generate basis functions of as many scheduling variables as you need.

Create the tunable surface using the design points and basis functions.

```
Kp = tunableSurface('Kp',1,domain,shapefcn);
```

`Kp` is a tunable model of the gain surface. `tunableSurface` parameterizes the surface as:

$$K_p(\alpha, V) = \bar{K}_0 + \bar{K}_1\bar{\alpha} + \bar{K}_2\bar{V} + \bar{K}_3\bar{\alpha}\bar{V},$$

where

$$\bar{\alpha} = \frac{\alpha - 7.5}{7.5}, \quad \bar{V} = \frac{V - 500}{200}.$$

The surface is expressed in terms of the normalized variables, $\bar{\alpha}, \bar{V} \in [-1, 1]^2$ rather than in terms of α and V. This normalization, which `tunableSurface` performs by default, improves the conditioning of the optimization performed by `systune`. If needed, you can change the default scaling and normalization. (See `tunableSurface`).

The second input argument to `tunableSurface` specifies the initial value of the constant coefficient, K_0. By default, K_0 is the gain when all the scheduling variables are at the center of their ranges. `tunableSurface` takes the I/O dimensions of the gain surface from K_0. Therefore, you can create array-valued tunable gains by providing an array for that input.

```
Karr = tunableSurface('Karr',ones(2),domain,shapefcn);
```

`Karr` is a 2-by-2 matrix in which each entry is a bilinear function of the scheduling variables with independent coefficients.

Tunable Gain with Two Independent Scheduling Variables

This example shows how to model a scalar gain K with a bilinear dependence on two scheduling variables. You do so by creating a grid of design points representing the independent dependence of the two variables.

Suppose that the first variable α is an angle of incidence that ranges from 0 to 15 degrees, and the second variable V is a speed that ranges from 300 to 600 m/s. By default, the normalized variables are:

$$x = \frac{\alpha - 7.5}{7.5}, \quad y = \frac{V - 450}{150}.$$

The gain surface is modeled as:

$$K(\alpha, V) = K_0 + K_1x + K_2y + K_3xy,$$

where $K_0, \ldots, K_3$ are the tunable parameters.

Create a grid of design points, (α,V), that are linearly spaced in α and V. These design points are the scheduling-variable values used for tuning the gain-surface coefficients. They must correspond to parameter values at which you have sampled the plant.

```
[alpha,V] = ndgrid(0:3:15,300:50:600);
```

These arrays, `alpha` and `V`, represent the independent variation of the two scheduling variables, each across its full range. Put them into a structure to define the design points for the tunable surface.

```
domain = struct('alpha',alpha,'V',V);
```

Create the basis functions that describe the bilinear expansion.

```
shapefcn = @(x,y) [x,y,x*y];  % or use polyBasis('canonical',1,2)
```

In the array returned by `shapefcn`, the basis functions are:

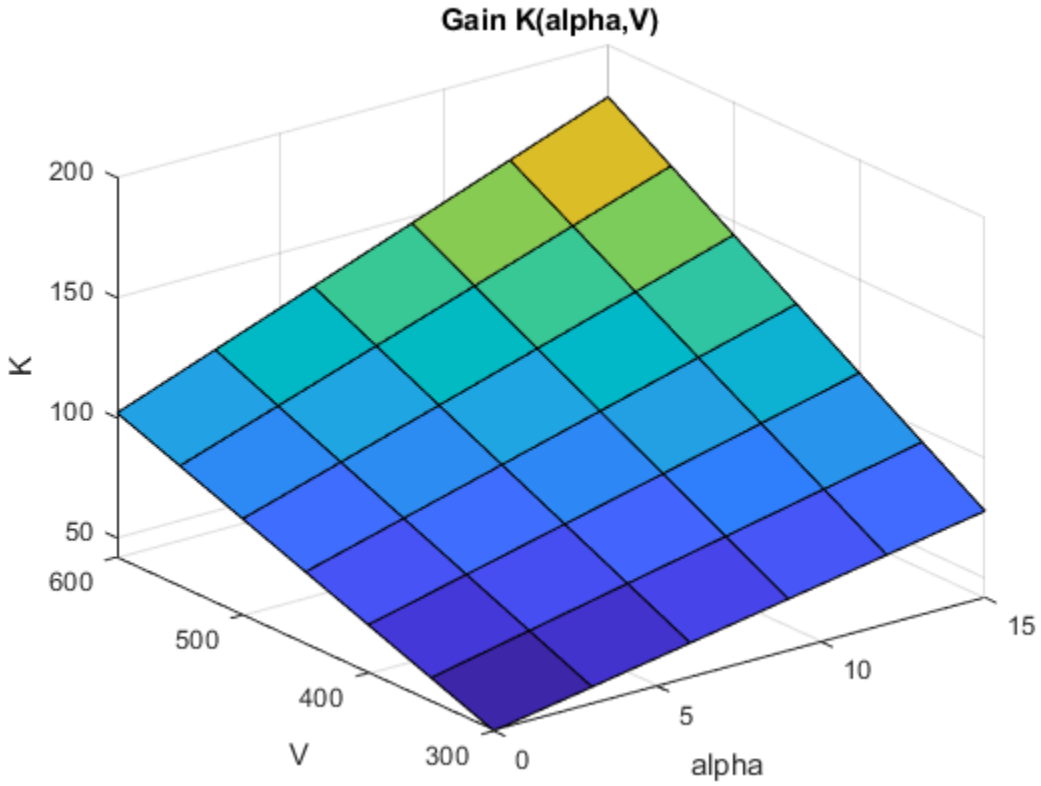
$$F_1(x,y) = x$$
$$F_2(x,y) = y$$
$$F_3(x,y) = xy\,.$$

Create the tunable gain surface.

```
K = tunableSurface('K',1,domain,shapefcn);
```

You can use the tunable surface as the parameterization for a lookup table block or a MATLAB Function block in a Simulink model. Or, use model interconnection commands to incorporate it as a tunable element in a control system modeled in MATLAB. After you tune the coefficients, you can examine the resulting gain surface using the `viewSurf` command. For this example, instead of tuning, manually set the coefficients to non-zero values and view the resulting gain.

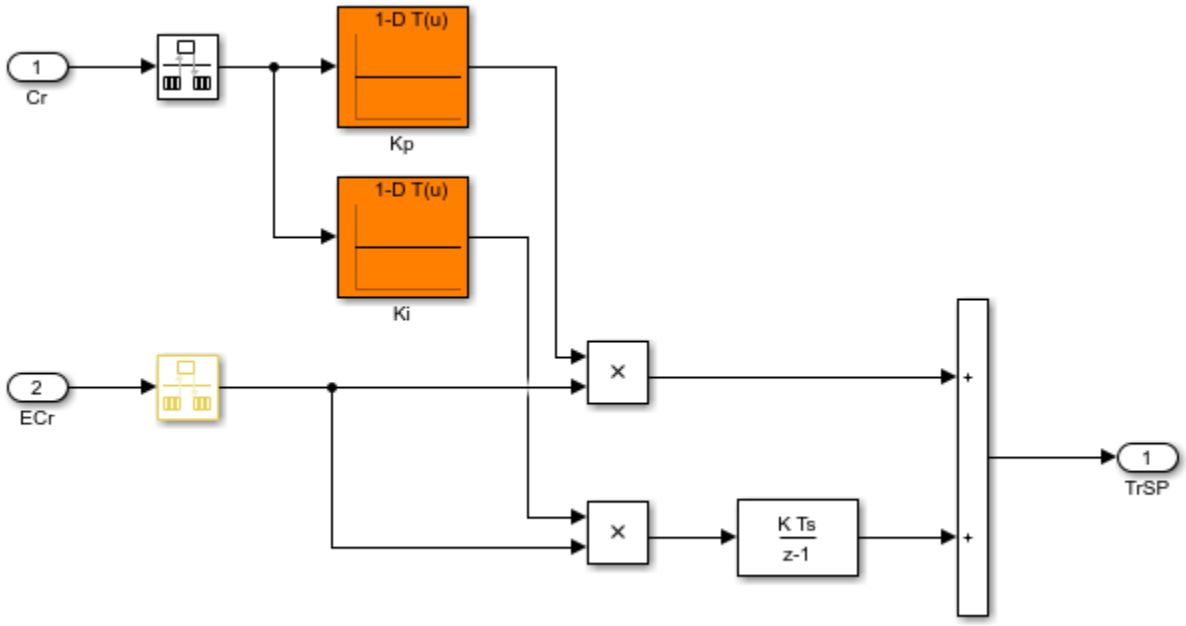
```
Ktuned = setData(K,[100,28,40,10]);
viewSurf(Ktuned)
```



`viewSurf` displays the gain surface as a function of the scheduling variables, for the ranges of values specified by `domain` and stored in the `SamplingGrid` property of the gain surface.

Tunable Surfaces in Simulink

In your Simulink model, you model gain schedules using lookup table blocks, MATLAB Function blocks, or Matrix Interpolation blocks, as described in “Model Gain-Scheduled Control Systems in Simulink” on page 11-4. To tune these gain surfaces, use `tunableSurface` to create a gain surface for each block. In the `slTuner` interface to the model, designate each gain schedule as a block to tune, and set its parameterization to the corresponding gain surface. For instance, the `rct_CSTR` model includes a gain-scheduled PI controller, the `Concentration controller` subsystem, in which the gains `Kp` and `Ki` vary with the scheduling variable `Cr`.



To tune the lookup tables `Kp` and `Ki`, create a tunable surface for each one. Suppose that `CrEQ` is the vector of design points, and that you expect the gains to vary quadratically with `Cr`.

```
TuningGrid = struct('Cr',CrEQ);
ShapeFcn = @(Cr) [Cr , Cr^2];

Kp = tunableSurface('Kp',0,TuningGrid,ShapeFcn);
Ki = tunableSurface('Ki',-2,TuningGrid,ShapeFcn);
```

Suppose that you have an array `Gd` of linearizations of the plant subsystem, `CSTR`, at each of the design points in `CrEQ`. (See “Plant Models for Gain-Scheduled Controller Tuning” on page 11-14). Create an `slTuner` interface that substitutes this array for the plant subsystem and designates the two lookup-table blocks for tuning.

```
BlockSubs = struct('Name','rct_CSTR/CSTR','Value',Gd);
ST0 = slTuner('rct_CSTR',{'Kp','Ki'},BlockSubs);
```

Finally, use the tunable surfaces to parameterize the lookup tables.

```
ST0.setBlockParam('Kp',Kp);
ST0.setBlockParam('Ki',Ki);
```

When you tune `ST0`, `systune` tunes the coefficients of the tunable surfaces `Kp` and `Ki`, so that each tunable surface represents the tuned relationship between `Cr` and the gain. When you write the tuned values back to the block for validation, `setBlockParam` automatically generates tuned lookup-table data by evaluating the tunable surfaces at the breakpoints you specify in the corresponding blocks.

For more details about this example, see “Gain-Scheduled Control of a Chemical Reactor”.

Tunable Surfaces in MATLAB

For a control system modeled in MATLAB, use tunable surfaces to construct more complex gain-scheduled control elements, such as gain-scheduled PID controllers, filters, or state-space controllers. For example, suppose that you create two gain surfaces `Kp` and `Ki` using `tunableSurface`. The following command constructs a tunable gain-scheduled PI controller.

```
C0 = pid(Kp,Ki);
```

Similarly, suppose that you create four matrix-valued gain surfaces `A`, `B`, `C`, `D`. The following command constructs a tunable gain-scheduled state-space controller.

```
C1 = ss(A,B,C,D);
```

You then incorporate the gain-scheduled controller into a generalized model of your entire control system. For example, suppose `G` is an array of models of your plant sampled at the design points that are specified in `Kp` and `Ki`. Then, the following command builds a tunable model of a gain-scheduled single-loop PID control system.

```
T0 = feedback(G*C0,1);
```

When you interconnect a tunable surface with other LTI models, the resulting model is an array of tunable generalized `genss` models. The design points in the tunable surface determine the dimensions of the array. Thus, each entry in the array represents the system at the corresponding scheduling variable value. The `SamplingGrid` property of the array stores those design points.

```
T0 = feedback(G*Kp,1)
```

```
T0 =

  4x5 array of generalized continuous-time state-space models.
  Each model has 1 outputs, 1 inputs, 3 states, and the following blocks:
    Kp: Parametric 1x4 matrix, 1 occurrences.

Type "ss(T0)" to see the current value, "get(T0)" to see all properties, and
"T0.Blocks" to interact with the blocks.
```

The resulting generalized model has tunable blocks corresponding to the gain surfaces used to create the model. In this example, the system has one gain surface, `Kp`, which has the four tunable coefficients corresponding to K_0, K_1, K_2, and K_3. Therefore, the tunable block is a vector-valued `realp` parameter with four entries.

When you tune the control system with `systune`, the software tunes the coefficients for each of the design points specified in the tunable surface.

For an example illustrating the entire workflow in MATLAB, see the section "Controller Tuning in MATLAB" in “Gain-Scheduled Control of a Chemical Reactor”.

See Also

`tunableSurface`

Related Examples

- “Model Gain-Scheduled Control Systems in Simulink” on page 11-4
- “Multiple Design Points in slTuner Interface” on page 11-20

- “Tune Gain Schedules in Simulink” on page 11-12

Change Requirements with Operating Condition

When tuning a gain-scheduled control system, it is sometimes useful to enforce different design requirements at different points in the design grid. For instance, you might want to:

- Specify a variable tuning goal that depends explicitly or implicitly on the design point.
- Enforce a tuning goal at a subset of design points, but ignore it at other design points.
- Exclude a design point from a particular run of `systune`, but retain it for analysis or other tuning operations.
- Eliminate a design point from all stages of design and analysis.

Define Variable Tuning Goal

There are several ways to define a tuning goal that changes across design points.

Create Varying Goals

The `varyingGoal` command lets you construct tuning goals that depend implicitly or explicitly on the design point.

For example, create a tuning goal that specifies variable gain and phase margins across a grid of design points. Suppose that you use the following 5-by-5 grid of design points to tune your controller.

```
[alpha,V] = ndgrid(linspace(0,20,5),linspace(700,1300,5));
```

Suppose further that you have 5-by-5 arrays of target gain margins and target phase margins corresponding to each of the design points, such as the following.

```
[GM,PM] = ndgrid(linspace(7,20,5),linspace(45,70,5));
```

To enforce the specified margins at each design point, first create a template for the margins goal. The template is a function that takes gain and phase margin values and returns a `TuningGoal.Margins` object with those margins.

```
FH = @(gm,pm) TuningGoal.Margins('u',gm,pm);
```

Use the template and the margin arrays to create the varying goal.

```
VG = varyingGoal(FH,GM,PM);
```

To make it easier to trace which goal applies to which design point, use the `SamplingGrid` property to attach the design-point information to `VG`.

```
VG.SamplingGrid = struct('alpha',alpha,'V',V);
```

Use `VG` with `systune` as you would use any other tuning goal. Use `viewGoal` to visualize the tuning goal and identify design points that fail to meet the target margins. For varying tuning goals, the `viewGoal` plot includes sliders that let you examine the goal and system performance for particular design points. See “Validate Gain-Scheduled Control Systems” on page 11-36.

The template function allows great flexibility in constructing the design goals. For example, you can write a function, `goalspec(a,b)`, that constructs the target overshoot as a nontrivial function of the parameters `(a,b)`, and save the function in a MATLAB file. Your template function then calls `goalspec`:

```
FH = @(a,b) TuningGoal.Overshoot('r',y',goalspec(a,b));
```

For more information about configuring varying goals, see the `varyingGoal` reference page.

Create Separate Requirement for Each Design Point

Another way to enforce a requirement that varies with design point is to create a separate instance of the requirement for each design point. This approach can be useful when you have a goal that only applies to a few of models in the design array. For example, suppose that you want to enforce a $1/s$ loop shape on the first five design points only, with a crossover frequency that depends on the scheduling variables. Suppose also that you have created a vector, `wc`, that contains the target bandwidth for each design point. Then you can construct one `TuningGoal.LoopShape` requirement for each design point. Associate each `TuningGoal.LoopShape` requirement with the corresponding design point using the `Models` property of the requirement.

```
for ct = 1:length(wc)
   R(ct) = TuningGoal.LoopShape('u',wc(ct));
   R(ct).Model = ct;
end
```

If `wc` covers all the design points in your grid, this approach is equivalent to using a `varyingGoal` object. It is a useful alternative to `varyingGoal` when you only want to constrain a few design points.

Build Variation into the Model

Instead of creating varying requirements, you can incorporate the varying portion of the requirement into the closed-loop model of the control system. This approach is a form of goal normalization that makes it possible to cover all design points with a single uniform goal.

For example, suppose that you want to limit the gain from `d` to `y` to a quantity that depends on the scheduling variables. Suppose that `T0` is an array of models of the closed-loop system at each design point. Suppose further that you have created a table, `gmax`, of the maximum gain values for each design point, σ. Then you can add another output `ys = y/gmax` to the closed-loop model, as follows.

```
% Create array of scalar gains 1/gmax
yScaling = reshape(1./gmax,[1 1 size(gmax)]);
yScaling = ss(yScaling,'InputName','y','OutputName','ys');

% Connect these gains in series to y output of T0
T0 = connect(T0,yScaling,T0.InputName,[T0.OutputName ; {'ys'}]);
```

The maximum gain changes at each design point according to the table `gmax`. You can then use a single requirement that limits to 1 the gain from `d` to the scaled output `ys`.

```
R = TuningGoal.Gain('d','ys',1);
```

Such effective normalization of requirements moves the requirement variability from the requirement object, `R`, to the closed-loop model, `T0`.

In Simulink, you can use a similar approach by feeding the relevant model inputs and outputs through a gain block. Then, when you linearize the model, change the gain value of the block with the operating condition. For example, set the gain to a MATLAB variable, and use the `Parameters` property in `slLinearizer` to change the variable value with each linearization condition.

Enforce Tuning Goal at Subset of Design Points

You can restrict application of a tuning goal to a subset of models in the design grid using the `Models` property of the tuning goal. Specify models by their linear index in the model array. For instance, suppose that you have a tuning goal, `Req`. Configure `Req` to apply to the first and last models in a 3-by-3 design grid.

```
Req.Models = [1,9];
```

When you call `systune` with `Req` as a hard or soft goal, `systune` enforces `Req` for these models and ignores it for the rest of the grid.

Exclude Design Points from systune Run

You can exclude one or more design points from tuning without removing the corresponding model from the array or reconfiguring your tuning goals. Doing so can be useful, for example, to identify problematic design points when tuning over the entire design grid fails to meet your design requirements. It can also be useful when there are design points that you want to exclude from a particular tuning run, but preserve for performance analysis or further tuning.

The `SkipModels` option of `systuneOptions` lets you specify models in the design grid to exclude from tuning. Specify models by their linear index in the model array. For instance, configure `systuneOptions` to skip the first and last models in a 3-by-3 design grid.

```
opt = systuneOptions;
opt.SkipModels = [1,9];
```

When you call `systune` with `opt`, the tuning algorithm ignores these models.

As an alternative, you can eliminate design points from the model grid entirely, so that they do not contribute to any stage of tuning or analysis. To do so, use `voidModel`, which replaces specified models in a model array with `NaN`. This option is useful when your sampling grid includes points that represent irrelevant or unphysical design points. Using `voidModel` lets you design over a grid of design points that is almost regular.

See Also

`viewGoal` | `varyingGoal` | `systuneOptions`

More About

- “Validate Gain-Scheduled Control Systems” on page 11-36
- “Tune Gain Schedules in Simulink” on page 11-12

Validate Gain-Scheduled Control Systems

Tuned gain schedules require careful validation. The tuning process guarantees suitable performance only near each design point. In addition, the tuning ignores dynamic couplings between the plant state variables and the scheduling variables (see Section 4.3, “Hidden Coupling”, in [1]). Best practices for validation include:

- Examine tuned gain surfaces to make sure that they are smooth and well-behaved.
- Visualize tuning goals against system responses at all design points.
- Check linear performance of the tuned control system between design points.
- Validate gain schedules in simulation of the full nonlinear system.

Check linear performance on a denser grid of σ values than you used for design. If adequate linear performance is not maintained between design points, you can add more design points and retune.

Perform nonlinear simulations that drive the closed-loop system through its entire operating range. Pay special attention to maneuvers that cause rapid variations of the scheduling variables.

Examine Tuned Gain Surfaces

After tuning, examine the tuned gains as a function of the scheduling variables to make sure that they are smooth and well-behaved over the operating range. Visualize tuned gain surfaces using the `viewSurf` command.

Visualize Tuning Goals

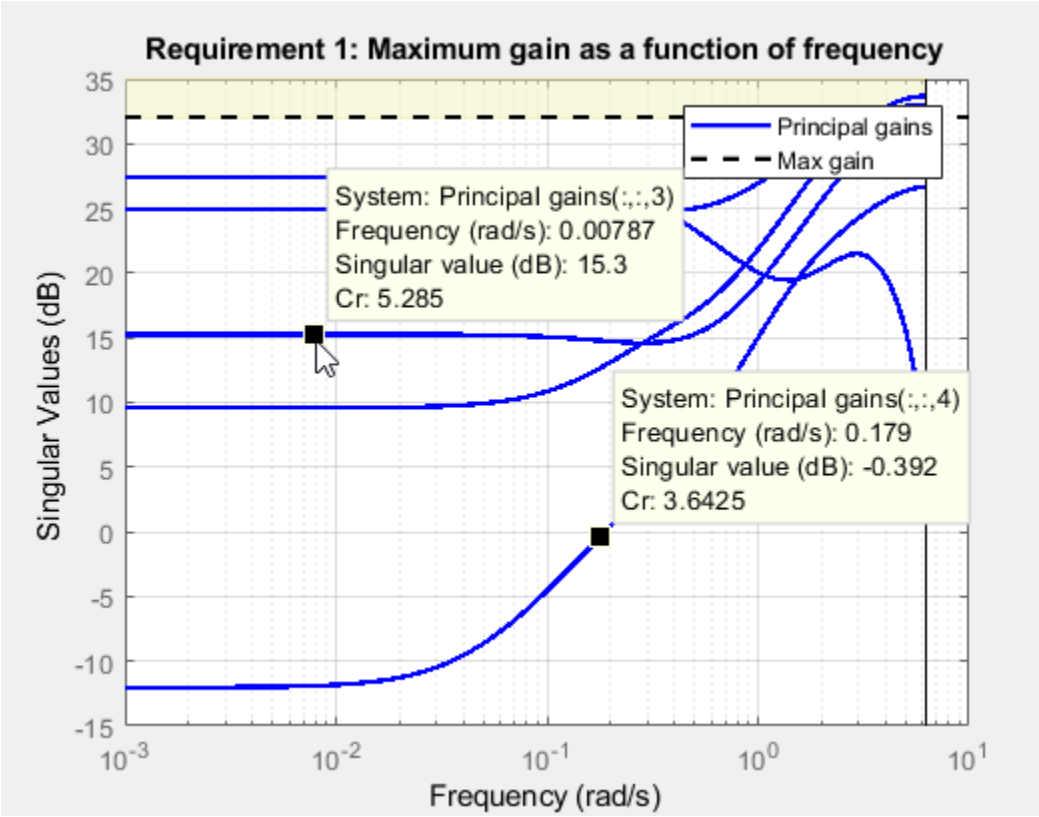
Use tuning-goal plots to visualize your design requirements against the linear response of the tuned control system. Tuning-goal plots show graphically where and by how much tuning goals are satisfied or violated. This visualization lets you examine how close your control system is to ideal performance. It can also help you identify problems with tuning and provide clues on how to improve your design.

For general information about using tuning-goal plots, see “Visualize Tuning Goals” on page 10-143. For gain-scheduled control systems, the tuning-goal plots you generate with `viewGoal` provide additional information that helps you evaluate how each tuning goal contributes to the result.

Fixed Tuning Goals

For fixed tuning goals that apply to multiple design points, `viewGoal` plots the relevant system response at all those design points. For instance, suppose that you tune an `slTuner` interface, `ST`, for the `rct_CSTR` model described in “Gain-Scheduled Control of a Chemical Reactor”. You can use `viewGoal` to see how well each of the five design points of that example satisfies the gain goal `R3`. The resulting plot shows the relevant gain profile at all five design points. Click any of the gain lines for a display that shows the corresponding value of the scheduling variable `Cr`.

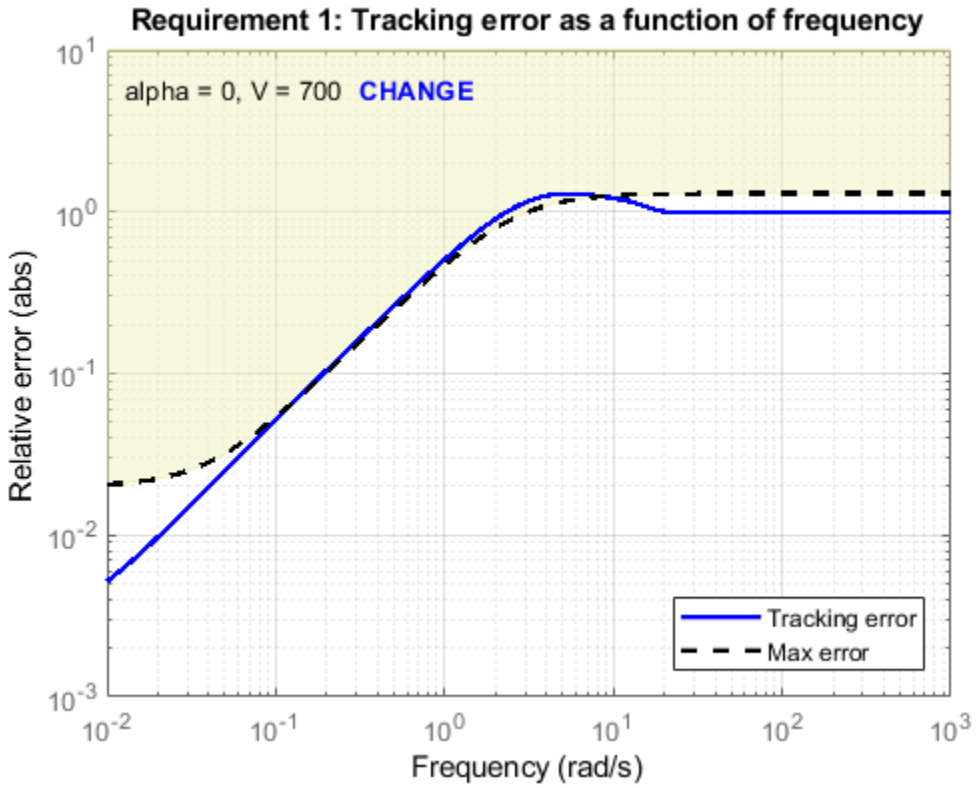
```
viewGoal(R3,ST)
```



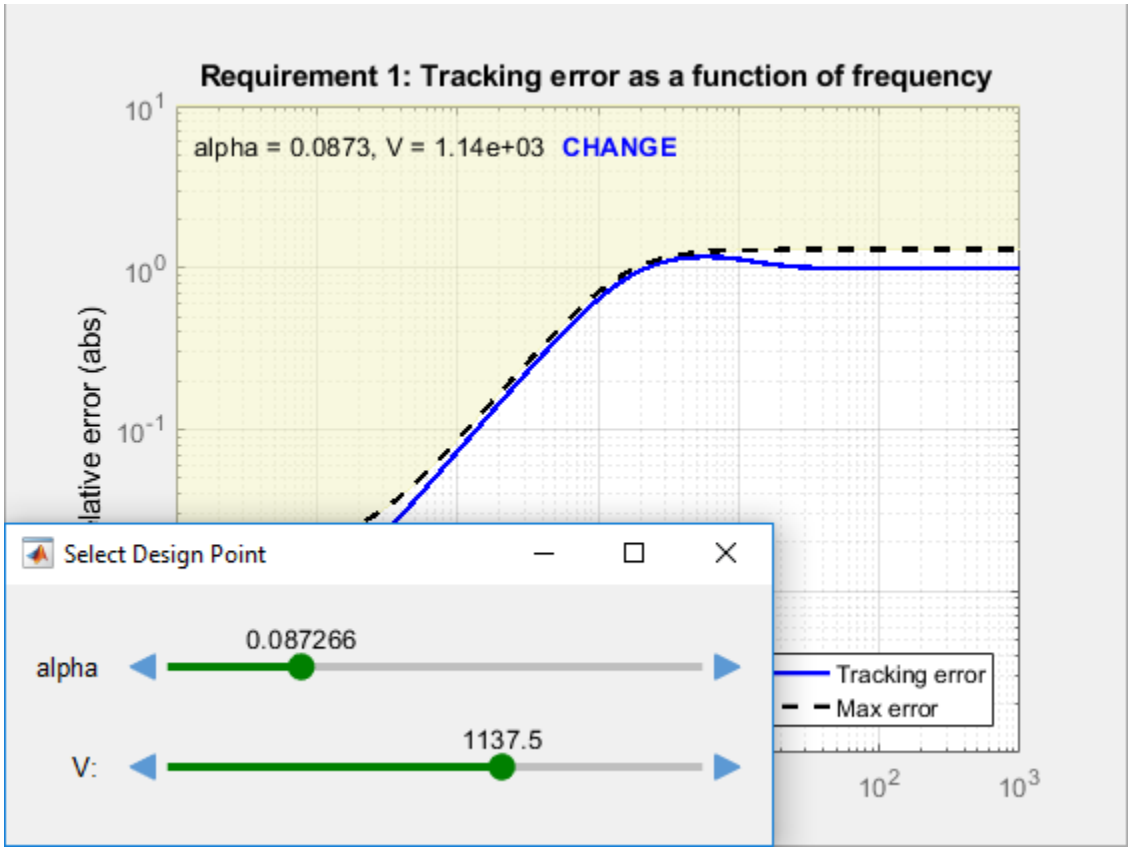
Varying Tuning Goals

Varying goals that you create using `varyingGoal` apply a different target response at each design point. When you use viewGoal to examine a varying goal, the plot initially displays the target and tuned responses at the first design point in the design grid. For instance, suppose that you tune a control system ST over a design grid of two scheduling variables, using a varying goal Rv that varies across the entire grid. After tuning, examine Rv.

```
viewGoal(Rv,ST)
```



Click **CHANGE** to open sliders that let you select a design point at which to view the target and tuned responses.



Check Linear Performance

In addition to examining linear responses associated with tuning goals, check other linear responses of the system to make sure that the behavior is suitable. You can do so by extracting and plotting system responses as described generally in “Validate Tuned Control System” on page 10-172.

For gain-scheduled systems, it is good practice to check linear performance on a denser grid of operating points than you used for design. If the system does not maintain adequate linear performance between design points, then you can add more design points and retune.

Validate Gain Schedules in Nonlinear System

Because `systune` tunes gain schedules against a linearization obtained at each design point, it is important to test the tuning results in simulation of the full nonlinear system. Perform nonlinear simulations that drive the closed-loop system through its entire operating range. Pay special attention to maneuvers that cause rapid variations of the scheduling variables.

After tuning an `slTuner` interface, use `writeBlockValue` to write tuned controller parameters to the Simulink model for such simulation. This command can write tuned gain schedules to lookup table blocks, Matrix Interpolation blocks, and MATLAB Function blocks for which you have specified a `tunableSurface` parameterization.

Lookup Tables

For lookup table blocks and Matrix Interpolation blocks, `writeBlockValue` automatically evaluates the tuned gain surface at the breakpoints specified in the block. These breakpoints do not need to be the same as the design points used for tuning. Because the `tunableSurface` describes the gain schedule in parametric form, `writeBlockValue` can evaluate the gain at any scheduling-variable value.

If you have retuned a subset of design points, you can use `writeLookupTableData` to update a portion of the lookup-table data while leaving the rest intact.

MATLAB Function Blocks

For gain schedules implemented as MATLAB Function blocks, `writeBlockValue` automatically generates MATLAB code and pushes it to the block. The generated MATLAB function takes the scheduling variables and returns the gain value given by the tuned parametric expression of the `tunableSurface`. To see this MATLAB code for a particular gain surface, use the `codegen` command.

References

[1] Rugh, W.J., and J.S. Shamma, "Research on Gain Scheduling", *Automatica*, 36 (2000), pp. 1401-1425.

See Also

`viewSurf` | `codegen` | `writeBlockValue` | `writeLookupTableData` | `viewGoal`

Related Examples

- “Tuning of Gain-Scheduled Three-Loop Autopilot”
- “Gain-Scheduled Control of a Chemical Reactor”
- “Validate Tuned Control System” on page 10-172

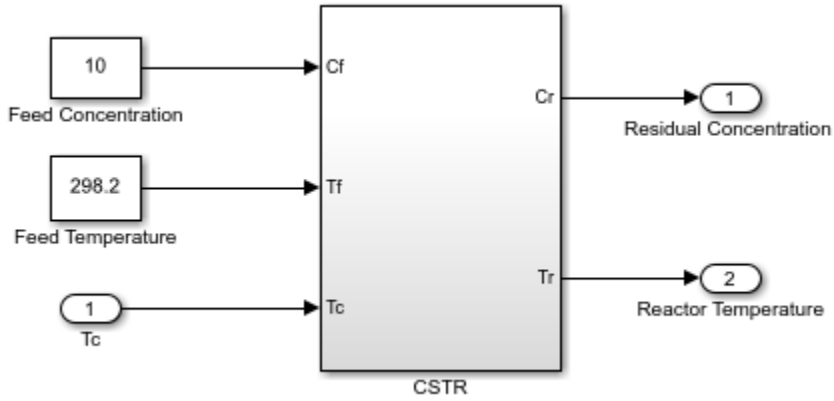
Gain-Scheduled Control of a Chemical Reactor

This example shows how to design and tune a gain-scheduled controller for a chemical reactor transitioning from low to high conversion rate. For background, see Seborg, D.E. et al., "Process Dynamics and Control", 2nd Ed., 2004, Wiley, pp. 34-36.

Continuous Stirred Tank Reactor

The process considered here is a continuous stirred tank reactor (CSTR) during transition from low to high conversion rate (high to low residual concentration). Because the chemical reaction is exothermic (produces heat), the reactor temperature must be controlled to prevent a thermal runaway. The control task is complicated by the fact that the process dynamics are nonlinear and transition from stable to unstable and back to stable as the conversion rate increases. The reactor dynamics are modeled in Simulink. The controlled variables are the residual concentration `Cr` and the reactor temperature `Tr`, and the manipulated variable is the temperature `Tc` of the coolant circulating in the reactor's cooling jacket.

```
open_system('rct_CSTR_OL')
```



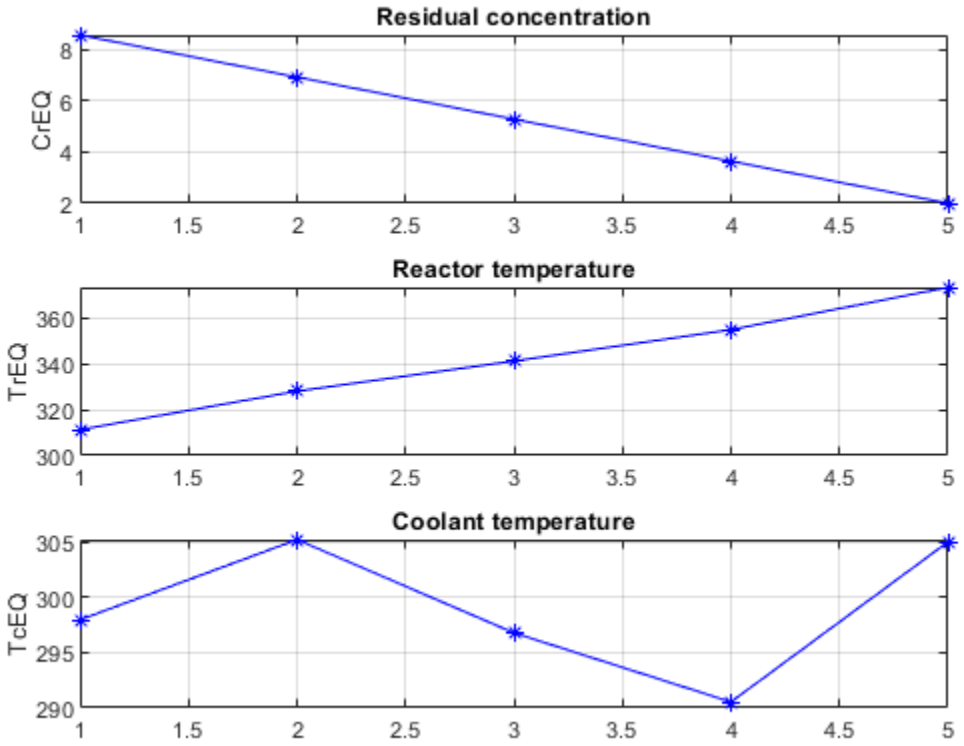
We want to transition from a residual concentration of 8.57 kmol/m^3 initially down to 2 kmol/m^3. To understand how the process dynamics evolve with the residual concentration `Cr`, find the equilibrium conditions for five values of `Cr` between 8.57 and 2 and linearize the process dynamics around each equilibrium. Log the reactor and coolant temperatures at each equilibrium point.

```
CrEQ = linspace(8.57,2,5)';  % concentrations
TrEQ = zeros(5,1);           % reactor temperatures
TcEQ = zeros(5,1);           % coolant temperatures

% Specify trim conditions
opspec = operspec('rct_CSTR_OL',5);
for k=1:5
   % Set desired residual concentration
   opspec(k).Outputs(1).y = CrEQ(k);
   opspec(k).Outputs(1).Known = true;
end

% Compute equilibrium condition and log corresponding temperatures
[op,report] = findop('rct_CSTR_OL',opspec,...
   findopOptions('DisplayReport','off'));
for k=1:5
```

```
   TrEQ(k) = report(k).Outputs(2).y;
   TcEQ(k) = op(k).Inputs.u;
end

% Linearize process dynamics at trim conditions
G = linearize('rct_CSTR_OL', 'rct_CSTR_OL/CSTR', op);
G.InputName = {'Cf','Tf','Tc'};
G.OutputName = {'Cr','Tr'};
```

Plot the reactor and coolant temperatures at equilibrium as a function of concentration.

```
subplot(311), plot(CrEQ,'b-*'), grid, title('Residual concentration'), ylabel('CrEQ')
subplot(312), plot(TrEQ,'b-*'), grid, title('Reactor temperature'), ylabel('TrEQ')
subplot(313), plot(TcEQ,'b-*'), grid, title('Coolant temperature'), ylabel('TcEQ')
```



An open-loop control strategy consists of following the coolant temperature profile above to smoothly transition between the `Cr`=8.57 and `Cr`=2 equilibria. However, this strategy is doomed by the fact that the reaction is unstable in the mid range and must be properly cooled to avoid thermal runaway. This is confirmed by inspecting the poles of the linearized models for the five equilibrium points considered above (three out of the five models are unstable).

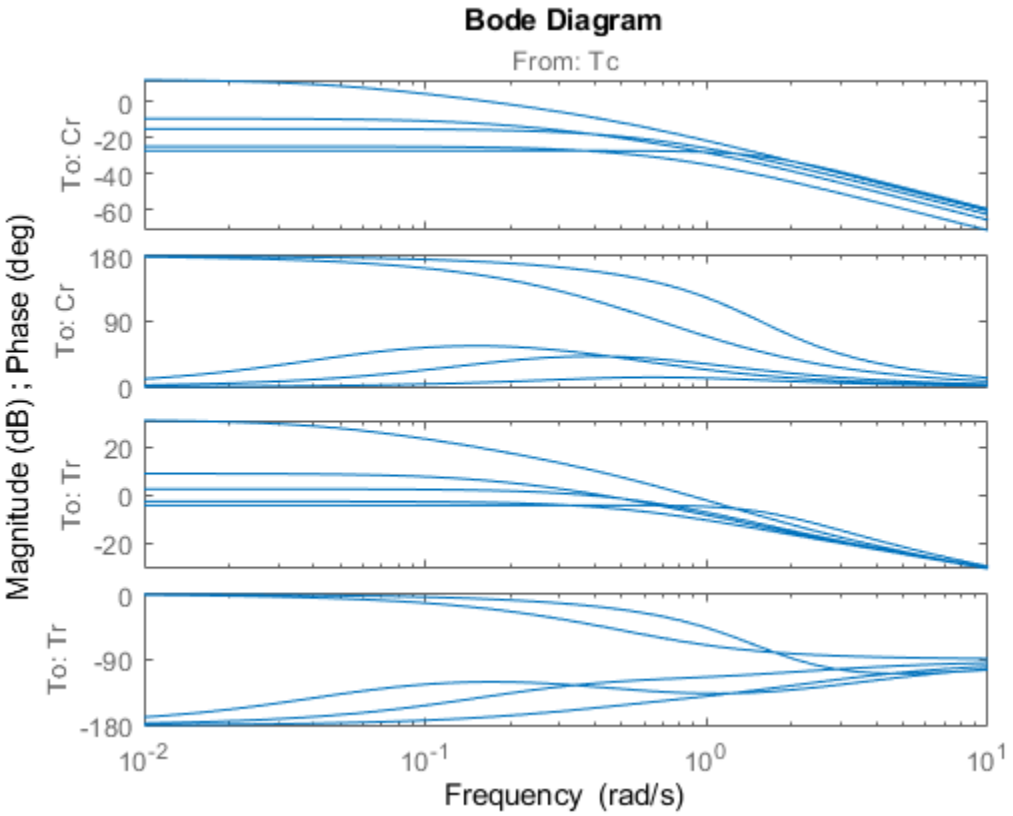
```
pole(G)
```

```
ans(:,:,1) =

  -0.5225 + 0.0000i
  -0.8952 + 0.0000i
```

```
ans(:,:,2) =

   0.1733 + 0.0000i
  -0.8866 + 0.0000i


ans(:,:,3) =

   0.5114 + 0.0000i
  -0.8229 + 0.0000i


ans(:,:,4) =

   0.0453 + 0.0000i
  -0.4991 + 0.0000i


ans(:,:,5) =

  -1.1077 + 1.0901i
  -1.1077 - 1.0901i
```

The Bode plot further highlights the significant variations in plant dynamics while transitioning from `Cr`=8.57 to `Cr`=2.

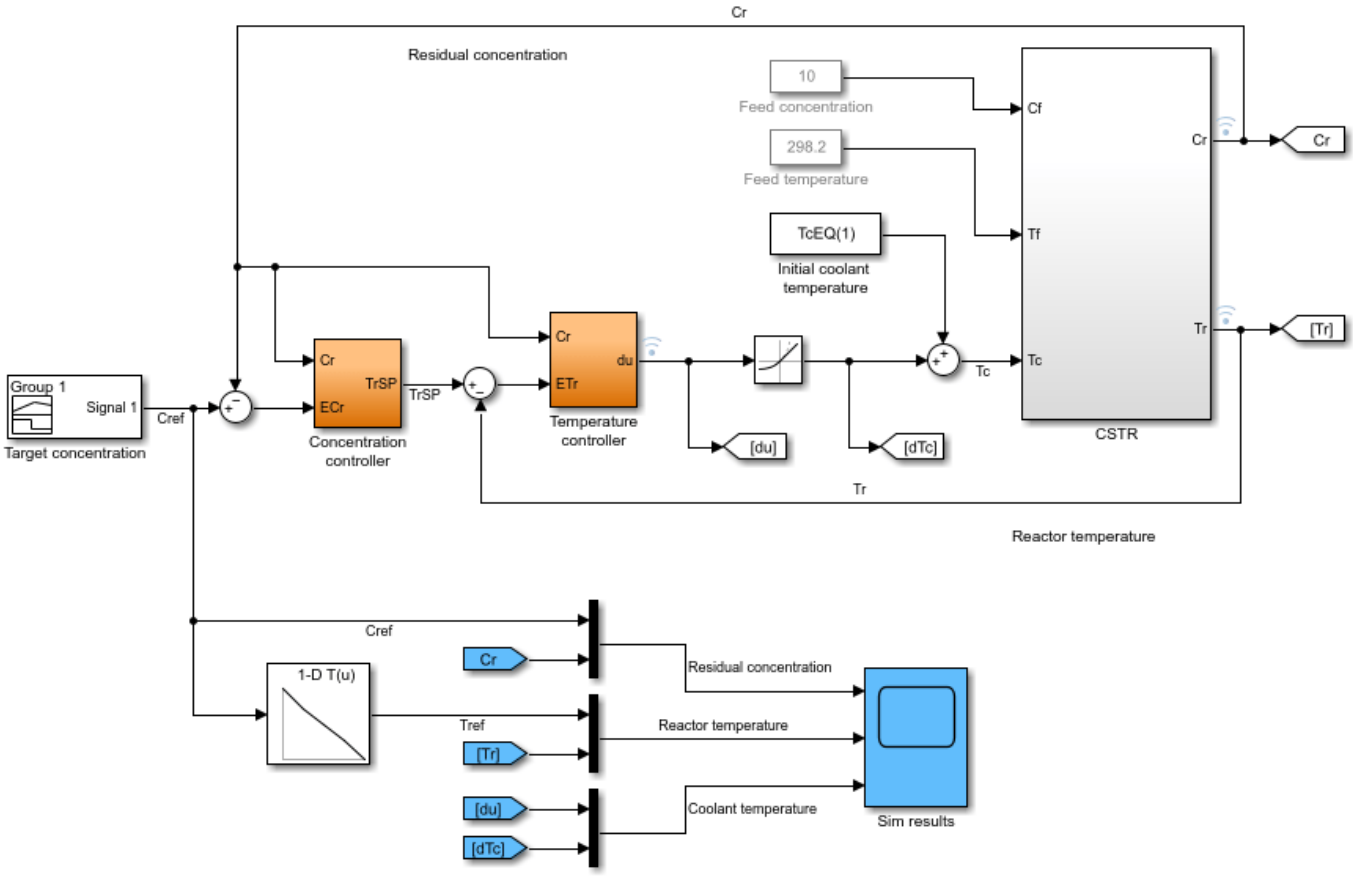
```
clf, bode(G(:,'Tc'),{0.01,10})
```



Feedback Control Strategy

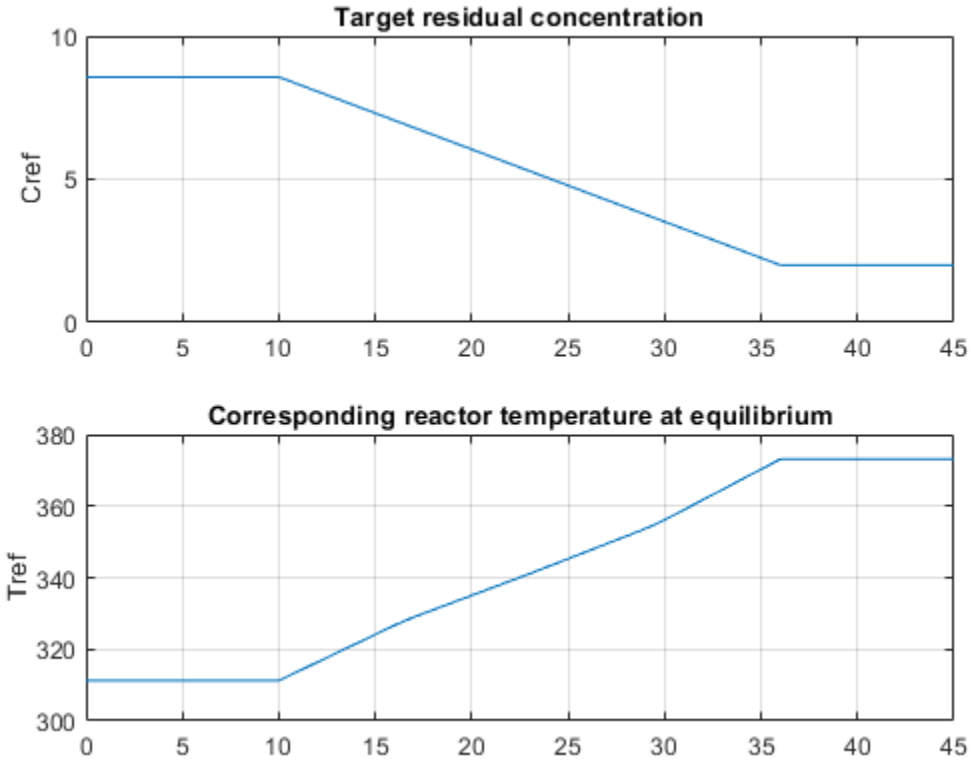
To prevent thermal runaway while ramping down the residual concentration, use feedback control to adjust the coolant temperature `Tc` based on measurements of the residual concentration `Cr` and reactor temperature `Tr`. For this application, we use a cascade control architecture where the inner loop regulates the reactor temperature and the outer loop tracks the concentration setpoint. Both feedback loops are digital with a sampling period of 0.5 minutes.

```
open_system('rct_CSTR')
```



The target concentration `Cref` ramps down from 8.57 kmol/m^3 at t=10 to 2 kmol/m^3 at t=36 (the transition lasts 26 minutes). The corresponding profile `Tref` for the reactor temperature is obtained by interpolating the equilibrium values `TrEQ` from trim analysis. The controller computes the coolant temperature adjustment `dTc` relative to the initial equilibrium value `TcEQ(1)`=297.98 for `Cr`=8.57. Note that the model is set up so that initially, the output `TrSP` of the "Concentration controller" block matches the reactor temperature, the adjustment `dTc` is zero, and the coolant temperature `Tc` is at its equilibrium value `TcEQ(1)`.

```
clf
t = [0 10:36 45];
C = interp1([0 10 36 45],[8.57 8.57 2 2],t);
subplot(211), plot(t,C), grid, set(gca,'ylim',[0 10])
title('Target residual concentration'), ylabel('Cref')
subplot(212), plot(t,interp1(CrEQ,TrEQ,C))
title('Corresponding reactor temperature at equilibrium'), ylabel('Tref'), grid
```



Control Objectives

Use `TuningGoal` objects to capture the design requirements. First, `Cr` should follow setpoints `Cref` with a response time of about 5 minutes.

```
R1 = TuningGoal.Tracking('Cref','Cr',5);
```

The inner loop (temperature) should stabilize the reaction dynamics with sufficient damping and fast enough decay.

```
MinDecay = 0.2;
MinDamping = 0.5;
% Constrain closed-loop poles of inner loop with the outer loop open
R2 = TuningGoal.Poles('Tc',MinDecay,MinDamping);
R2.Openings = 'TrSP';
```

The Rate Limit block at the controller output specifies that the coolant temperature `Tc` cannot vary faster than 10 degrees per minute. This is a severe limitation on the controller authority which, when ignored, can lead to poor performance or instability. To take this rate limit into account, observe that `Cref` varies at a rate of 0.25 kmol/m^3/min. To ensure that `Tc` does not vary faster than 10 degrees/min, the gain from `Cref` to `Tc` should be less than 10/0.25=40.

```
R3 = TuningGoal.Gain('Cref','Tc',40);
```

Finally, require at least 7 dB of gain margin and 45 degrees of phase margin at the plant input `Tc`.

```
R4 = TuningGoal.Margins('Tc',7,45);
```

Gain-Scheduled Controller

To achieve these requirements, we use a PI controller in the outer loop and a lead compensator in the inner loop. Due to the slow sampling rate, the lead compensator is needed to adequately stabilize the chemical reaction at the mid-range concentration `Cr` = 5.28 kmol/m^3/min. Because the reaction dynamics vary substantially with concentration, we further schedule the controller gains as a function of concentration. This is modeled in Simulink using Lookup Table blocks as shown in Figures 1 and 2.

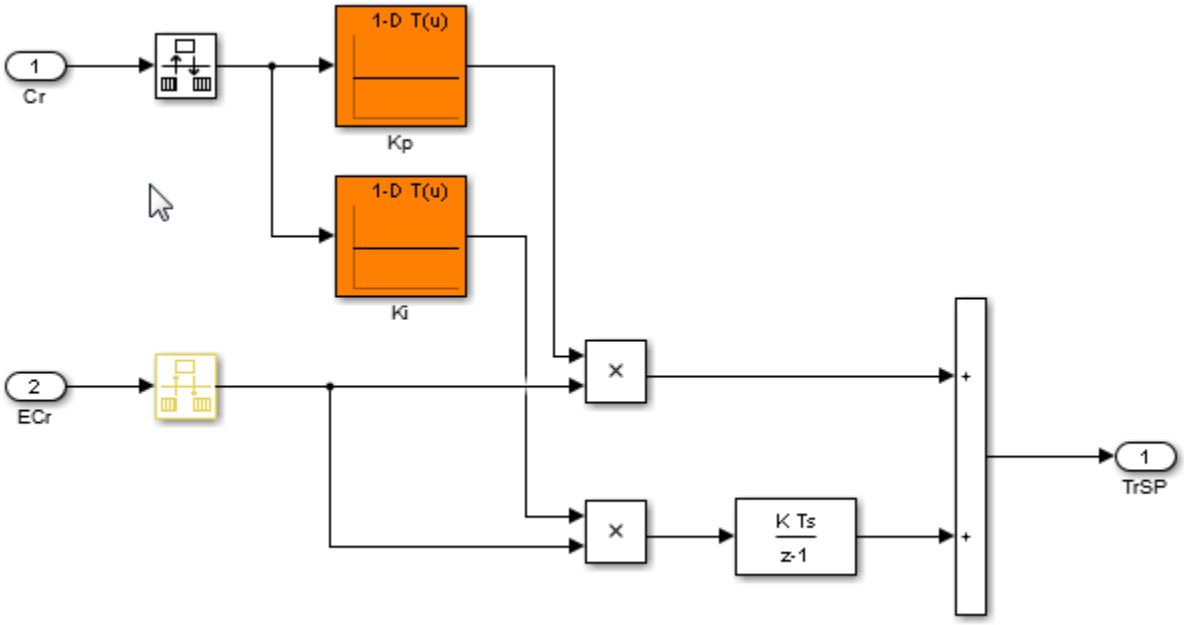


Figure 1: Gain-scheduled PI controller for concentration loop.

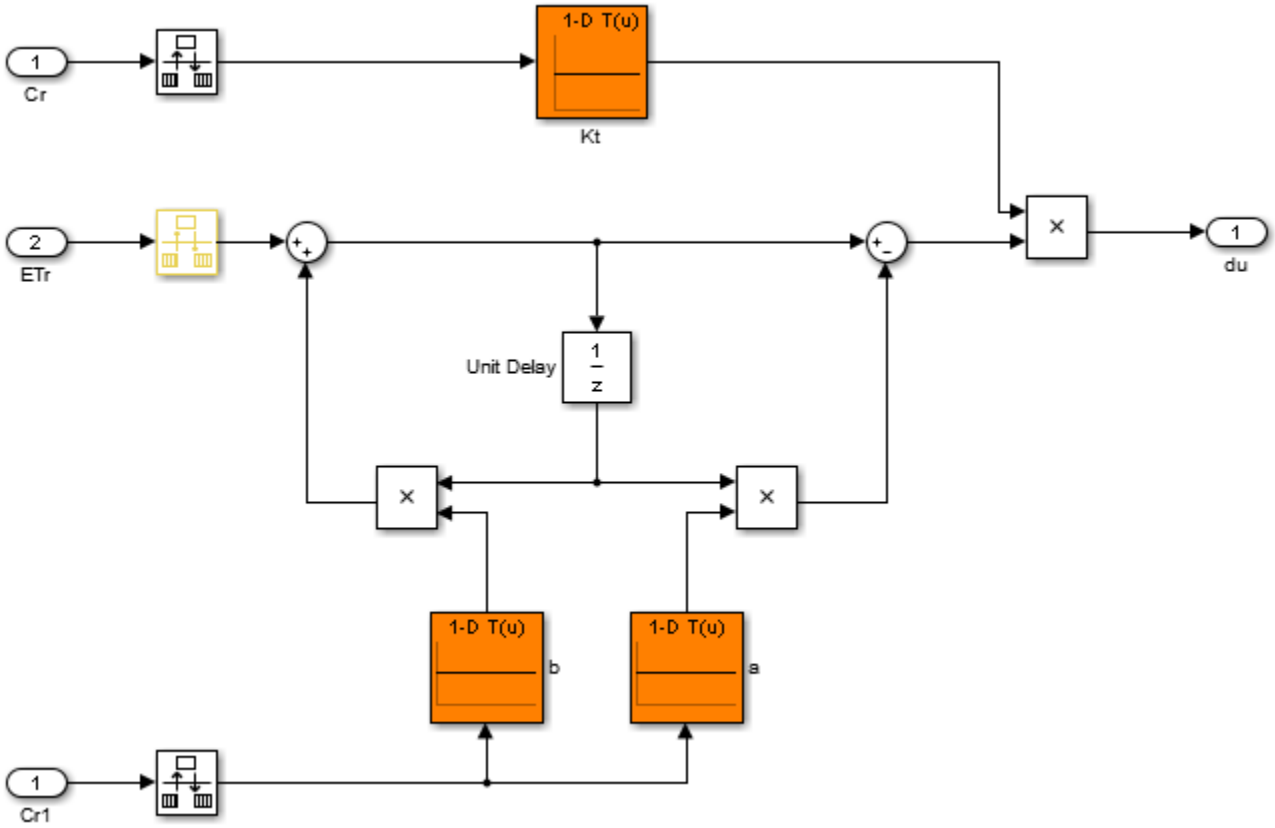


Figure 2: Gain-scheduled lead compensator for temperature loop.

Tuning this gain-scheduled controller amounts to tuning the look-up table data over a range of concentration values. Rather than tuning individual look-up table entries, parameterize the controller gains `Kp,Ki,Kt,a,b` as quadratic polynomials in `Cr`, for example,

$$K_p(C_r) = K_{p0} + K_{p1}C_r + K_{p2}C_r^2.$$

Besides reducing the number of variables to tune, this approach ensures smooth gain transitions as `Cr` varies. Using `systune`, you can automatically tune the coefficients $K_{p0}, K_{p1}, K_{p2}, K_{i0}, \cdots$ to meet the requirements `R1-R4` at the five equilibrium points computed above. This amounts to tuning the gain-scheduled controller at five design points along the `Cref` trajectory. Use the `tunableSurface` object to parameterize each gain as a quadratic function of `Cr`. The "tuning grid" is set to the five concentrations `CrEQ` and the basis functions for the quadratic parameterization are C_r, C_r^2. Most gains are initialized to be identically zero.

```
TuningGrid = struct('Cr',CrEQ);
ShapeFcn = @(Cr) [Cr , Cr^2];

Kp = tunableSurface('Kp', 0, TuningGrid, ShapeFcn);
Ki = tunableSurface('Ki', -2, TuningGrid, ShapeFcn);
Kt = tunableSurface('Kt', 0, TuningGrid, ShapeFcn);
a = tunableSurface('a', 0, TuningGrid, ShapeFcn);
b = tunableSurface('b', 0, TuningGrid, ShapeFcn);
```

Controller Tuning

Because the target bandwidth is within a decade of the Nyquist frequency, it is easier to tune the controller directly in the discrete domain. Discretize the linearized process dynamics with sample time of 0.5 minutes. Use the ZOH method to reflect how the digital controller interacts with the continuous-time plant.

```
Ts = 0.5;
Gd = c2d(G,Ts);
```

Create an `slTuner` interface for tuning the quadratic gain schedules introduced above. Use block substitution to replace the nonlinear plant model by the five discretized linear models `Gd` obtained at the design points `CrEQ`. Use `setBlockParam` to associate the tunable gain functions `Kp`, `Ki`, `Kt`, `a`, `b` with the Lookup Table blocks of the same name.

```
BlockSubs = struct('Name','rct_CSTR/CSTR','Value',Gd);
ST0 = slTuner('rct_CSTR',{'Kp','Ki','Kt','a','b'},BlockSubs);
ST0.Ts = Ts;  % sample time for tuning

% Register points of interest
ST0.addPoint({'Cref','Cr','Tr','TrSP','Tc'})

% Parameterize look-up table blocks
ST0.setBlockParam('Kp',Kp);
ST0.setBlockParam('Ki',Ki);
ST0.setBlockParam('Kt',Kt);
ST0.setBlockParam('a',a);
ST0.setBlockParam('b',b);
```

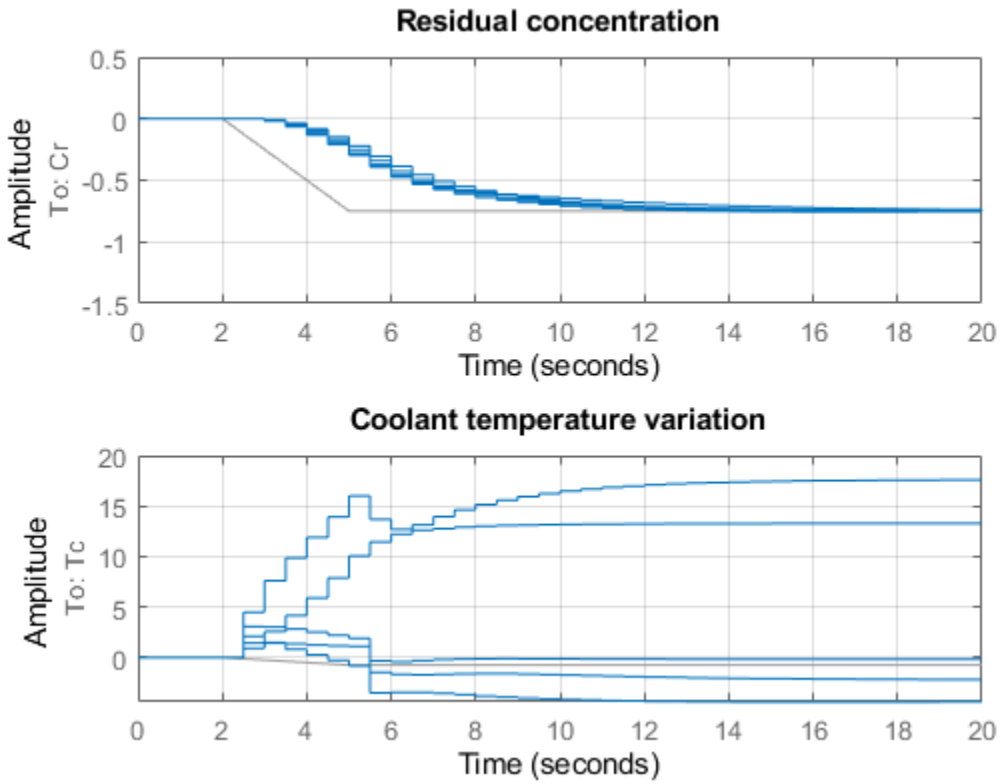
You can now use `systune` to tune the controller coefficients against the requirements `R1-R4`. Make the stability margin requirement a hard constraints and optimize the remaining requirements.

```
ST = systune(ST0,[R1 R2 R3],R4);
```

```
Final: Soft = 1.23, Hard = 0.99989, Iterations = 196
```

The resulting design satisfies the hard constraint (`Hard<1`) and nearly satisfies the remaining requirements (`Soft` close to 1). To validate this design, simulate the responses to a ramp in concentration with the same slope as `Cref`. Each plot shows the linear responses at the five design points `CrEQ`.

```
t = 0:Ts:20;
uC = interp1([0 2 5 20],(-0.25)*[0 0 3 3],t);
subplot(211), lsim(getIOTransfer(ST,'Cref','Cr'),uC)
grid, set(gca,'ylim',[-1.5 0.5]), title('Residual concentration')
subplot(212), lsim(getIOTransfer(ST,'Cref','Tc'),uC)
grid, title('Coolant temperature variation')
```

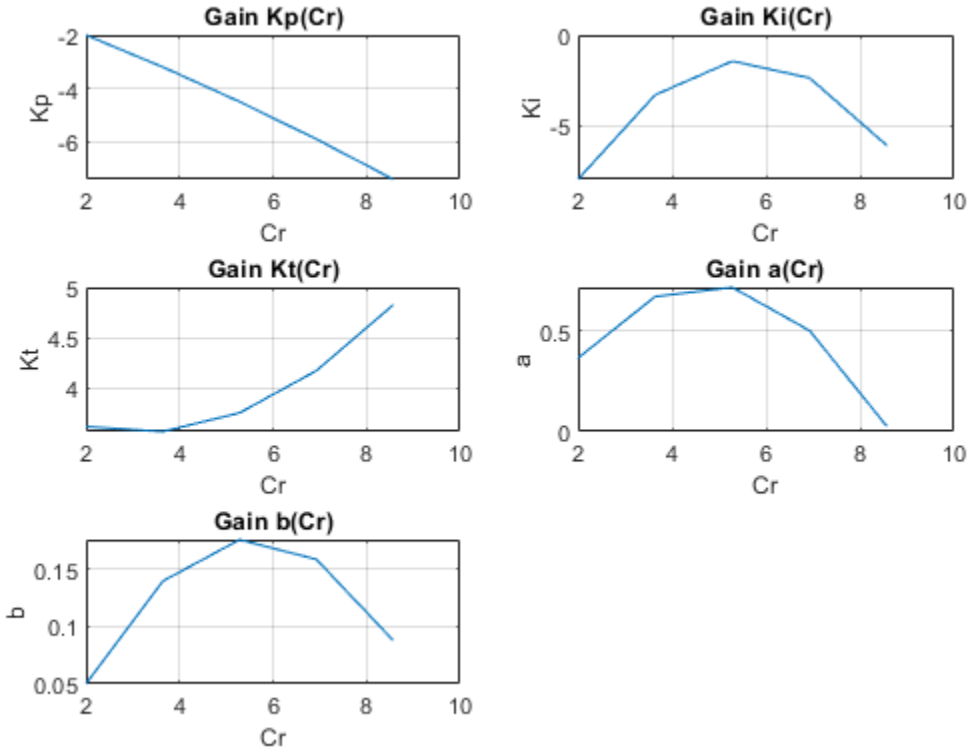


Note that rate of change of the coolant temperature remains within the physical limits (10 degrees per minute or 5 degrees per sample period).

Controller Validation

Inspect how each gain varies with `Cr` during the transition.

```
% Access tuned gain schedules
TGS = getBlockParam(ST);

% Plot gain profiles
clf
subplot(321), viewSurf(TGS.Kp), ylabel('Kp')
subplot(322), viewSurf(TGS.Ki), ylabel('Ki')
subplot(323), viewSurf(TGS.Kt), ylabel('Kt')
subplot(324), viewSurf(TGS.a), ylabel('a')
subplot(325), viewSurf(TGS.b), ylabel('b')
```



To validate the gain-scheduled controller in Simulink, first use `writeBlockValue` to apply the tuning results to the Simulink model. For each Lookup Table block, this evaluates the corresponding quadratic gain formula at the table breakpoints and updates the table data accordingly.

```
writeBlockValue(ST)
```

Next push the Play button to simulate the response with the tuned gain schedules. The simulation results appear in Figure 3. The gain-scheduled controller successfully drives the reaction through the transition with adequate response time and no saturation of the rate limits (controller output `du` matches effective temperature variation `dTc`). The reactor temperature stays close to its equilibrium value `Tref`, indicating that the controller keeps the reaction near equilibrium while preventing thermal runaway.

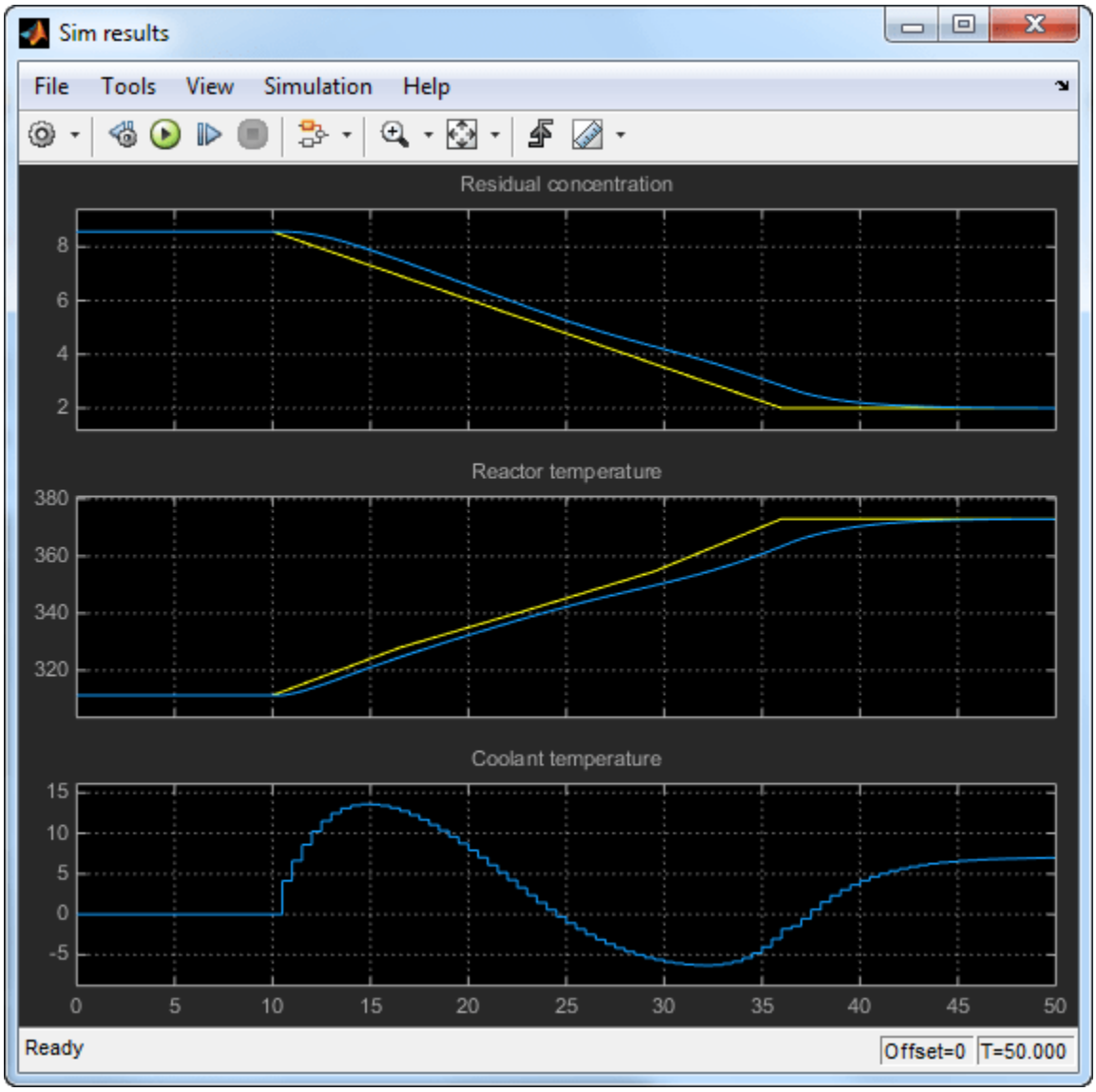


Figure 3: Transition with gain-scheduled cascade controller.

Controller Tuning in MATLAB

Alternatively, you can tune the gain schedules directly in MATLAB without using the `slTuner` interface. First parameterize the gains as quadratic functions of `Cr` as done above.

```
TuningGrid = struct('Cr',CrEQ);
ShapeFcn = @(Cr) [Cr , Cr^2];

Kp = tunableSurface('Kp', 0, TuningGrid, ShapeFcn);
Ki = tunableSurface('Ki', -2, TuningGrid, ShapeFcn);
Kt = tunableSurface('Kt', 0, TuningGrid, ShapeFcn);
a = tunableSurface('a', 0, TuningGrid, ShapeFcn);
b = tunableSurface('b', 0, TuningGrid, ShapeFcn);
```

Use these gains to build the PI and lead controllers.

```
PI = pid(Kp,Ki,'Ts',Ts,'TimeUnit','min');
PI.u = 'ECr';   PI.y = 'TrSP';

LEAD = Kt * tf([1 -a],[1 -b],Ts,'TimeUnit','min');
LEAD.u = 'ETr';   LEAD.y = 'Tc';
```

Use `connect` to build a closed-loop model of the overall control system at the five design points. Mark the controller outputs `TrSP` and `Tc` as "analysis points" so that loops can be opened and stability margins evaluated at these locations. The closed-loop model `T0` is a 5-by-1 array of linear models depending on the tunable coefficients of `Kp,Ki,Kt,a,b`. Each model is discrete and sampled every half minute.

```
Gd.TimeUnit = 'min';
S1 = sumblk('ECr = Cref - Cr');
S2 = sumblk('ETr = TrSP - Tr');
T0 = connect(Gd(:,'Tc'),LEAD,PI,S1,S2,'Cref','Cr',{'TrSP','Tc'});
```

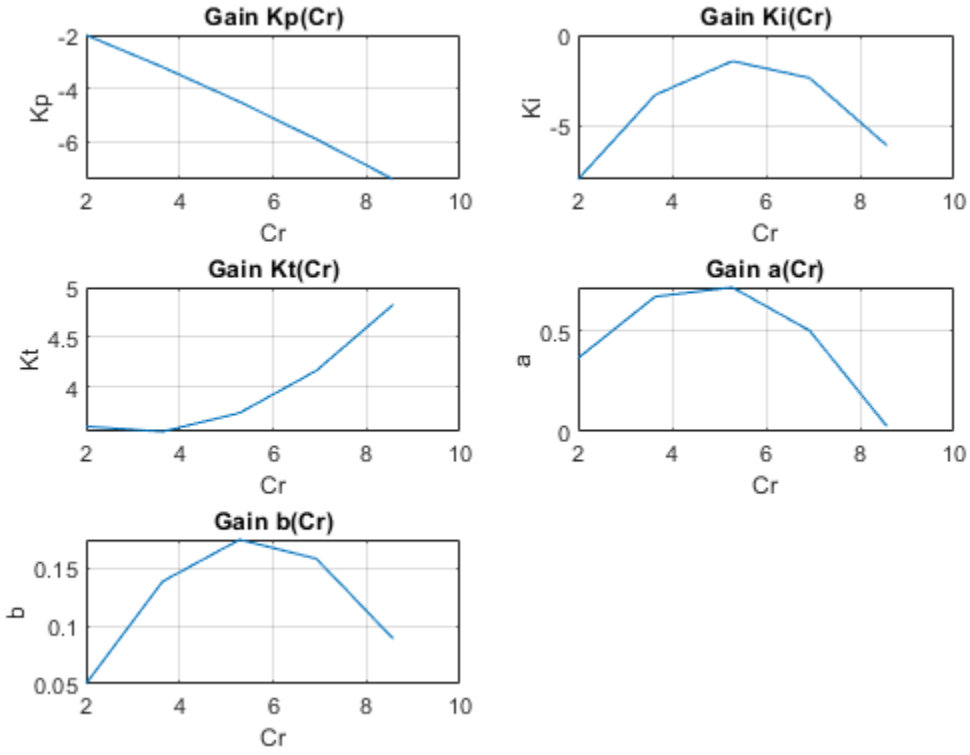
Finally, use `systune` to tune the gain schedule coefficients.

```
T = systune(T0,[R1 R2 R3],R4);
```

```
Final: Soft = 1.23, Hard = 0.99986, Iterations = 193
```

The result is similar to the one obtained above. Confirm by plotting the gains as a function of `Cr` using the tuned coefficients in `T`.

```
clf
subplot(321), viewSurf(setBlockValue(Kp,T)), ylabel('Kp')
subplot(322), viewSurf(setBlockValue(Ki,T)), ylabel('Ki')
subplot(323), viewSurf(setBlockValue(Kt,T)), ylabel('Kt')
subplot(324), viewSurf(setBlockValue(a,T)), ylabel('a')
subplot(325), viewSurf(setBlockValue(b,T)), ylabel('b')
```



You can further validate the design by simulating the linear responses at each design point. However, you need to return to Simulink to simulate the nonlinear response of the gain-scheduled controller.

See Also

`slTuner` | `tunableSurface` | `setBlockParam`

Related Examples

- “Model Gain-Scheduled Control Systems in Simulink” on page 11-4
- “Tuning of Gain-Scheduled Three-Loop Autopilot” on page 11-55

More About

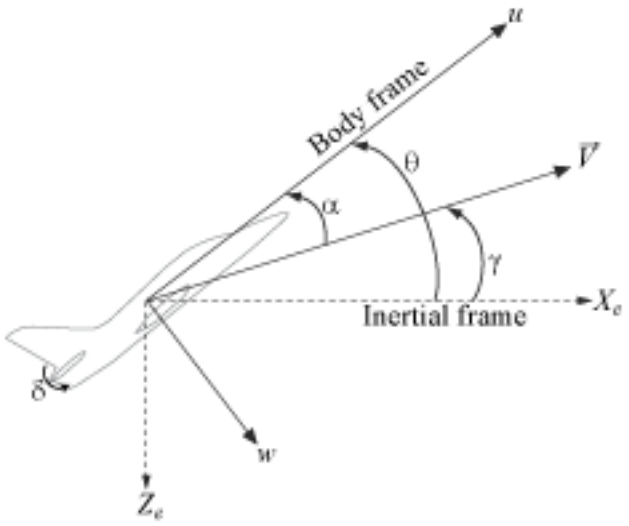
- “Parameterize Gain Schedules” on page 11-24

Tuning of Gain-Scheduled Three-Loop Autopilot

This example uses `systune` to generate smooth gain schedules for a three-loop autopilot.

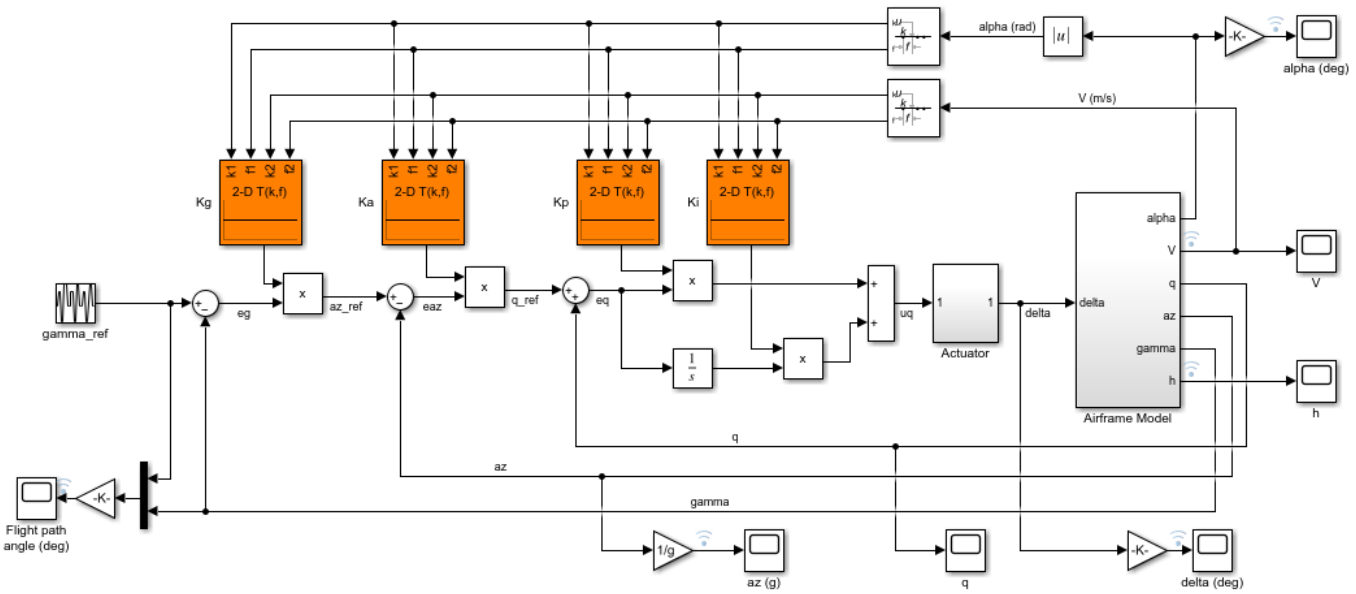
Airframe Model and Three-Loop Autopilot

This example uses a three-degree-of-freedom model of the pitch axis dynamics of an airframe. The states are the Earth coordinates (X_e, Z_e), the body coordinates (u, w), the pitch angle θ, and the pitch rate $q = \dot{\theta}$. The following figure summarizes the relationship between the inertial and body frames, the flight path angle γ, the incidence angle α, and the pitch angle θ.



We use a classic three-loop autopilot structure to control the flight path angle γ. This autopilot adjusts the flight path by delivering adequate bursts of normal acceleration a_z (acceleration along w). In turn, normal acceleration is produced by adjusting the elevator deflection δ to cause pitching and vary the amount of lift. The autopilot uses Proportional-Integral (PI) control in the pitch rate loop q and proportional control in the a_z and γ loops. The closed-loop system (airframe and autopilot) are modeled in Simulink.

```
open_system('rct_airframeGS')
```



Autopilot Gain Scheduling

The airframe dynamics are nonlinear and the aerodynamic forces and moments depend on speed V and incidence α. To obtain suitable performance throughout the (α, V) flight envelope, the autopilot gains must be adjusted as a function of α and V to compensate for changes in plant dynamics. This adjustment process is called "gain scheduling" and α, V are called the scheduling variables. In the Simulink model, gain schedules are implemented as look-up tables driven by measurements of α and V.

Gain scheduling is a linear technique for controlling nonlinear or time-varying plants. The idea is to compute linear approximations of the plant at various operating conditions, tune the controller gains at each operating condition, and swap gains as a function of operating condition during operation. Conventional gain scheduling involves the following three major steps.

1. Trim and linearize the plant at each operating condition
2. Tune the controller gains for the linearized dynamics at each operating condition
3. Reconcile the gain values to provide smooth transition between operating conditions.

In this example, you combine steps 2 and 3 by parameterizing the autopilot gains as first-order polynomials in α, V and directly tuning the polynomial coefficients for the entire flight envelope. This approach eliminates step 3 and guarantees smooth gain variations as a function of α and V. Moreover, the gain schedule coefficients can be automatically tuned with `systune`.

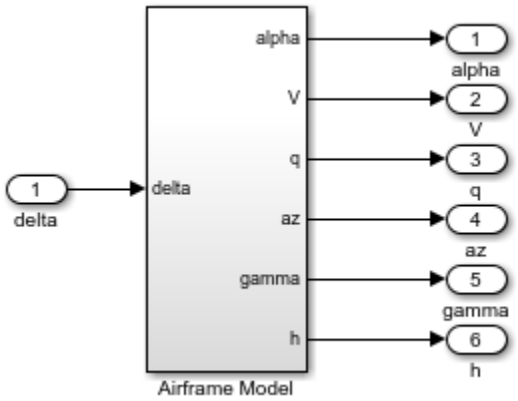
Trimming and Linearization

Assume that the incidence α varies between -20 and 20 degrees and that the speed V varies between 700 and 1400 m/s. When neglecting gravity, the airframe dynamics are symmetric in α. Therefore, consider only positive values of α. Use a 5-by-9 grid of linearly spaced (α, V) pairs to cover the flight envelope.

```
nA = 5;  % number of alpha values
nV = 9;  % number of V values
[alpha,V] = ndgrid(linspace(0,20,nA)*pi/180,linspace(700,1400,nV));
```

For each flight condition (α, V), linearize the airframe dynamics at trim (zero normal acceleration and pitching moment). This requires computing the elevator deflection δ and pitch rate q that result in steady w and q. To do this, first isolate the airframe model in a separate Simulink model.

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



Use `operspec` to specify the trim condition, use `findop` to compute the trim values of δ and q, and linearize the airframe dynamics for the resulting operating points. For details, see “Trim and Linearize an Airframe” on page 2-181. Repeat these steps for the 45 flight conditions (α, V).

Compute the trim condition for each (α, V) pair.

```
for ct=1:nA*nV
   alpha_ini = alpha(ct);      % Incidence [rad]
   v_ini = V(ct);              % Speed [m/s]

   % Specify trim condition
   opspec(ct) = operspec(mdl);
   % Xe,Ze: known, not steady
   opspec(ct).States(1).Known = [1;1];
   opspec(ct).States(1).SteadyState = [0;0];
   % u,w: known, w steady
   opspec(ct).States(3).Known = [1 1];
   opspec(ct).States(3).SteadyState = [0 1];
   % theta: known, not steady
   opspec(ct).States(2).Known = 1;
   opspec(ct).States(2).SteadyState = 0;
   % q: unknown, steady
   opspec(ct).States(4).Known = 0;
   opspec(ct).States(4).SteadyState = 1;
end
opspec = reshape(opspec,[nA nV]);
```

Trim the model for the given specifications.

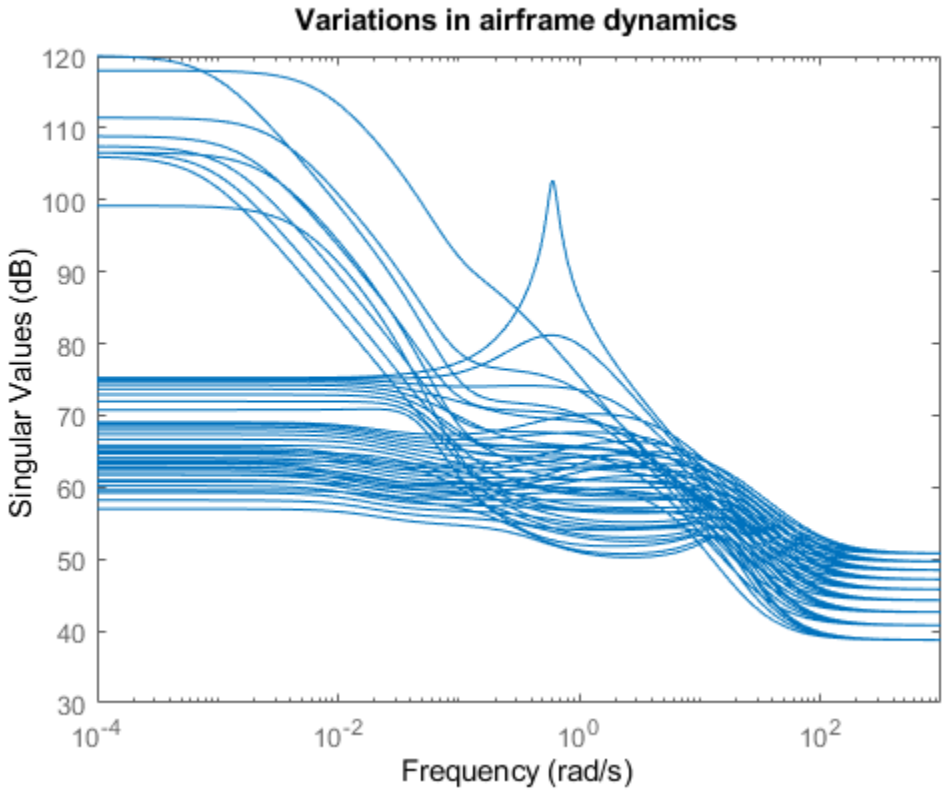
```
Options = findopOptions('DisplayReport','off');
op = findop(mdl,opspec,Options);
```

Linearize the model at the trim conditions.

```
G = linearize(mdl,op);
G.u = 'delta';
G.y = {'alpha','V','q','az','gamma','h'};
G.SamplingGrid = struct('alpha',alpha,'V',V);
```

This process produces a 5-by-9 array of linearized plant models at the 45 flight conditions (α, V). The plant dynamics vary substantially across the flight envelope.

```
sigma(G)
title('Variations in airframe dynamics')
```



Tunable Gain Surface

The autopilot consists of four gains K_p, K_i, K_a, K_g to be "scheduled" (adjusted) as a function of α and V. Practically, this means tuning 88 values in each of the corresponding four look-up tables. Rather than tuning each table entry separately, parameterize the gains as a two-dimensional gain surfaces, for example, surfaces with a simple multi-linear dependence on α and V:

$$K(\alpha, V) = K_0 + K_1\alpha + K_2 V + K_3 \alpha V .$$

This cuts the number of variables from 88 down to 4 for each lookup table. Use the `tunableSurface` object to parameterize each gain surface. Note that:

- `TuningGrid` specifies the "tuning grid" (design points). This grid should match the one used for linearization but needs not match the loop-up table breakpoints
- `ShapeFcn` specifies the basis functions for the surface parameterization (α, V, and αV)

Each surface is initialized to a constant gain using the tuning results for α = 10 deg and V = 1050 m/s (mid-range design).

```
TuningGrid = struct('alpha',alpha,'V',V);
ShapeFcn = @(alpha,V) [alpha,V,alpha*V];

Kp = tunableSurface('Kp',0.1, TuningGrid,ShapeFcn);
Ki = tunableSurface('Ki',2, TuningGrid,ShapeFcn);
Ka = tunableSurface('Ka',0.001, TuningGrid,ShapeFcn);
Kg = tunableSurface('Kg',-1000, TuningGrid,ShapeFcn);
```

Next create an `slTuner` interface for tuning the gain surfaces. Use block substitution to replace the nonlinear plant model by the linearized models over the tuning grid. Use `setBlockParam` to associate the tunable gain surfaces `Kp`, `Ki`, `Ka`, `Kg` with the Interpolation blocks of the same name.

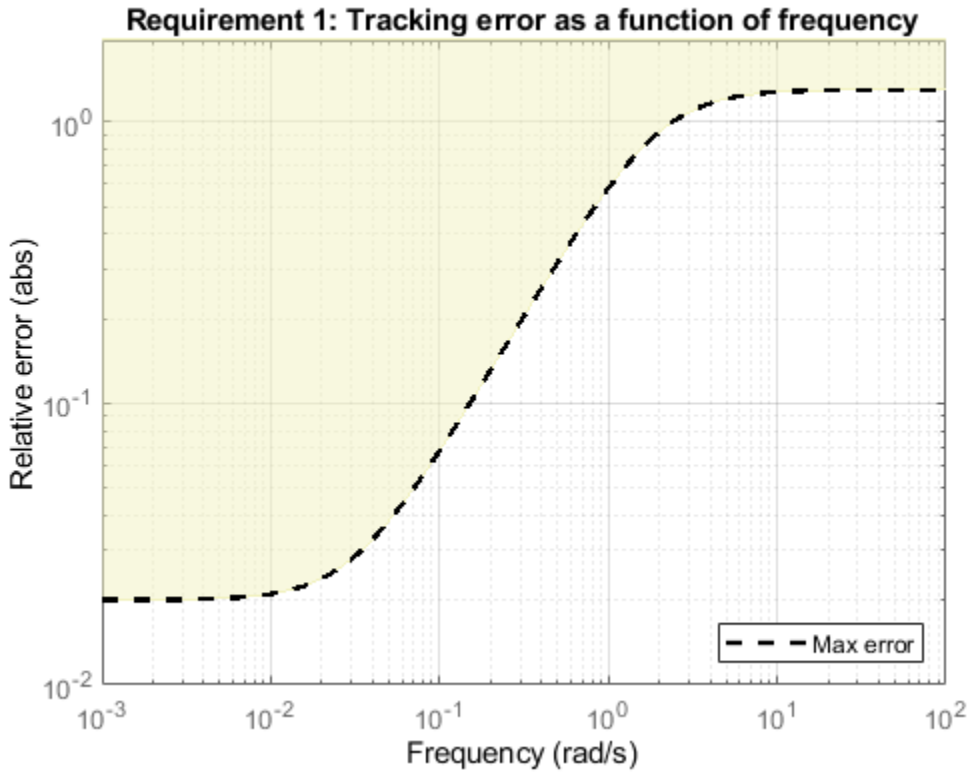
```
BlockSubs = struct('Name','rct_airframeGS/Airframe Model','Value',G);
ST0 = slTuner('rct_airframeGS',{'Kp','Ki','Ka','Kg'},BlockSubs);

% Register points of interest
ST0.addPoint({'az_ref','az','gamma_ref','gamma','delta'})

% Parameterize look-up table blocks
ST0.setBlockParam('Kp',Kp,'Ki',Ki,'Ka',Ka,'Kg',Kg);
```

Autopilot Tuning

`systune` can automatically tune the gain surface coefficients for the entire flight envelope. Use `TuningGoal` objects to specify the performance objectives:

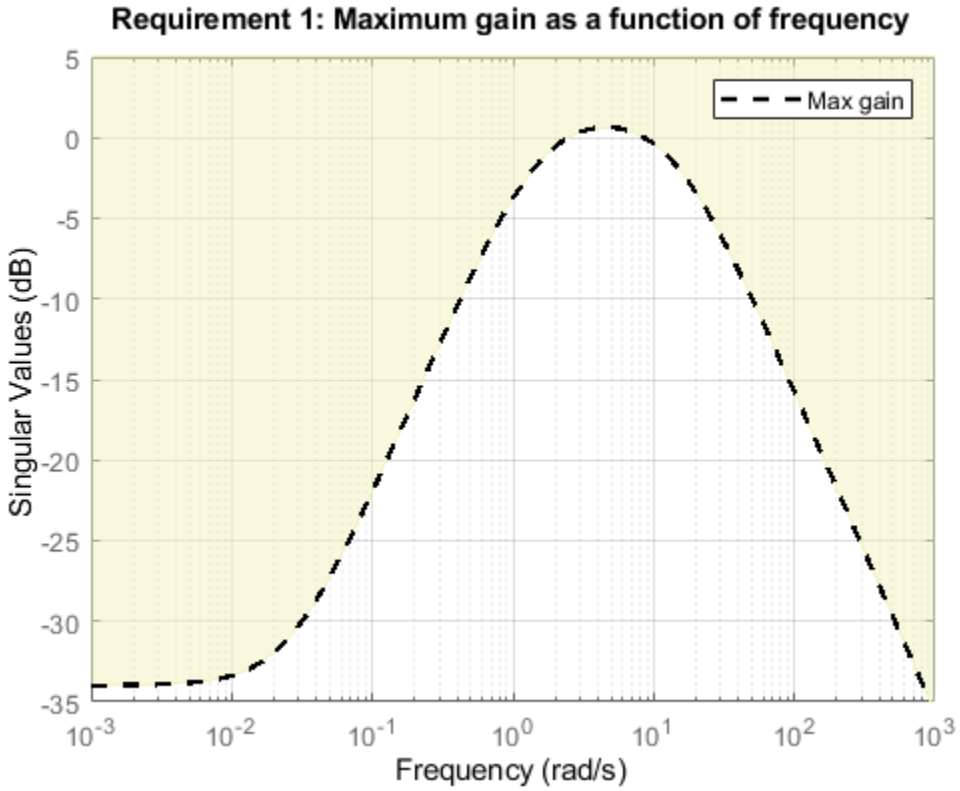
- γ loop: Track the setpoint with a 1 second response time, less than 2% steady-state error, and less than 30% peak error.

```
Req1 = TuningGoal.Tracking('gamma_ref','gamma',1,0.02,1.3);
viewGoal(Req1)
```



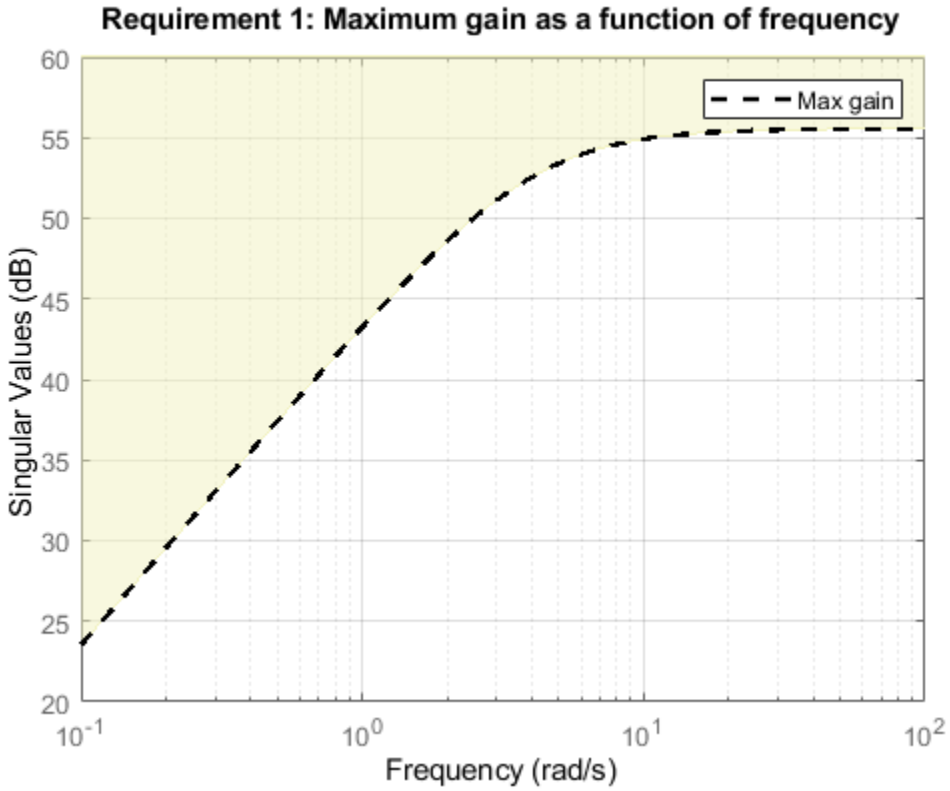
- a_z loop: Ensure good disturbance rejection at low frequency (to track acceleration demands) and past 10 rad/s (to be insensitive to measurement noise). The disturbance is injected at the `az_ref` location.

```
RejectionProfile = frd([0.02 0.02 1.2 1.2 0.1],[0 0.02 2 15 150]);
Req2 = TuningGoal.Gain('az_ref','az',RejectionProfile);
viewGoal(Req2)
```



- q loop: Ensure good disturbance rejection up to 10 rad/s. The disturbance is injected at the plant input `delta`.

```
Req3 = TuningGoal.Gain('delta','az',600*tf([0.25 0],[0.25 1]));
viewGoal(Req3)
```



- Transients: Ensure a minimum damping ratio of 0.35 for oscillation-free transients

```
MinDamping = 0.35;
Req4 = TuningGoal.Poles(0,MinDamping);
```

Using `systune`, tune the 16 gain surface coefficients to best meet these performance requirements at all 45 flight conditions.

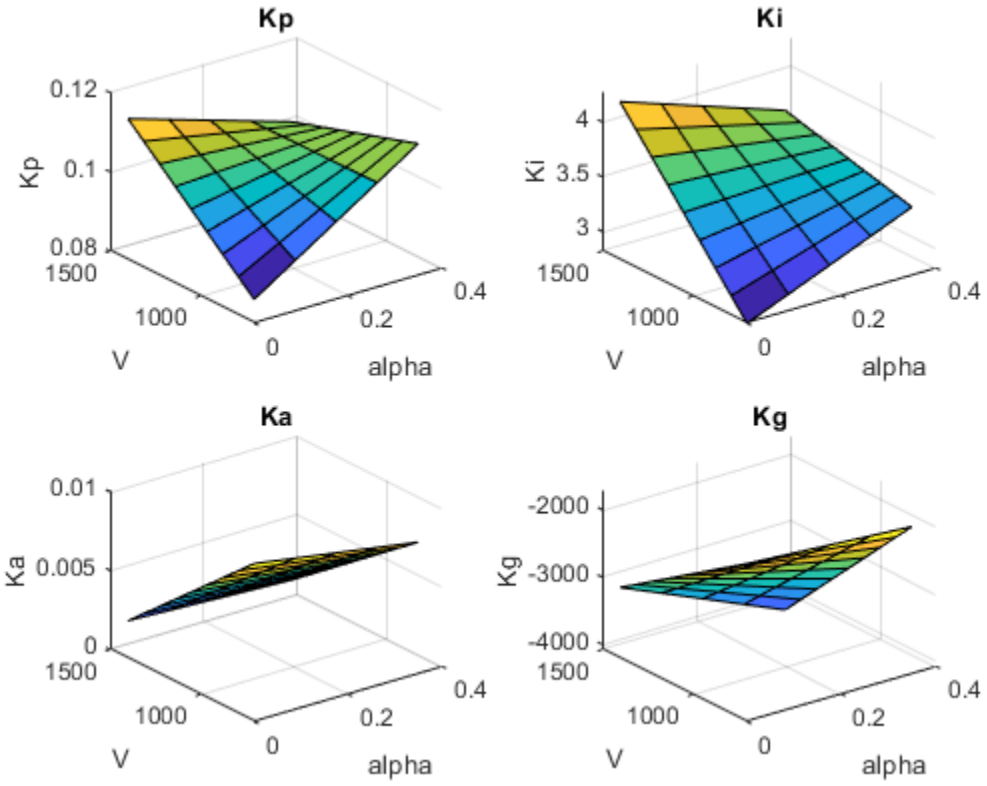
```
ST = systune(ST0,[Req1 Req2 Req3 Req4]);
```

```
Final: Soft = 1.13, Hard = -Inf, Iterations = 57
```

The final value of the combined objective is close to 1, indicating that all requirements are nearly met. Visualize the resulting gain surfaces.

```
% Get tuned gain surfaces.
TGS = getBlockParam(ST);

% Plot gain surfaces.
clf
subplot(2,2,1)
viewSurf(TGS.Kp)
title('Kp')
subplot(2,2,2)
viewSurf(TGS.Ki)
title('Ki')
subplot(2,2,3)
viewSurf(TGS.Ka)
```

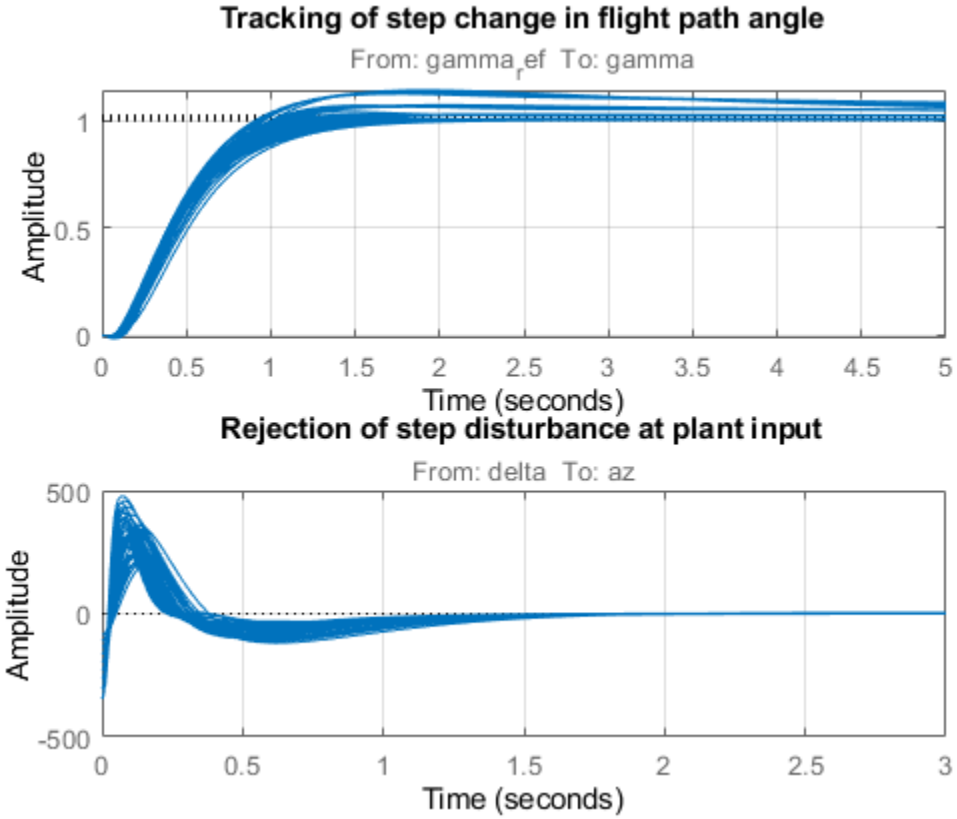
```
title('Ka')
subplot(2,2,4)
viewSurf(TGS.Kg)
title('Kg')
```



Validation

First validate the tuned autopilot at the 45 flight conditions considered above. Plot the response to a step change in flight path angle and the response to a step disturbance in elevator deflection.

```
clf
subplot(2,1,1)
step(getIOTransfer(ST,'gamma_ref','gamma'),5)
grid
title('Tracking of step change in flight path angle')
subplot(2,1,2)
step(getIOTransfer(ST,'delta','az'),3)
grid
title('Rejection of step disturbance at plant input')
```



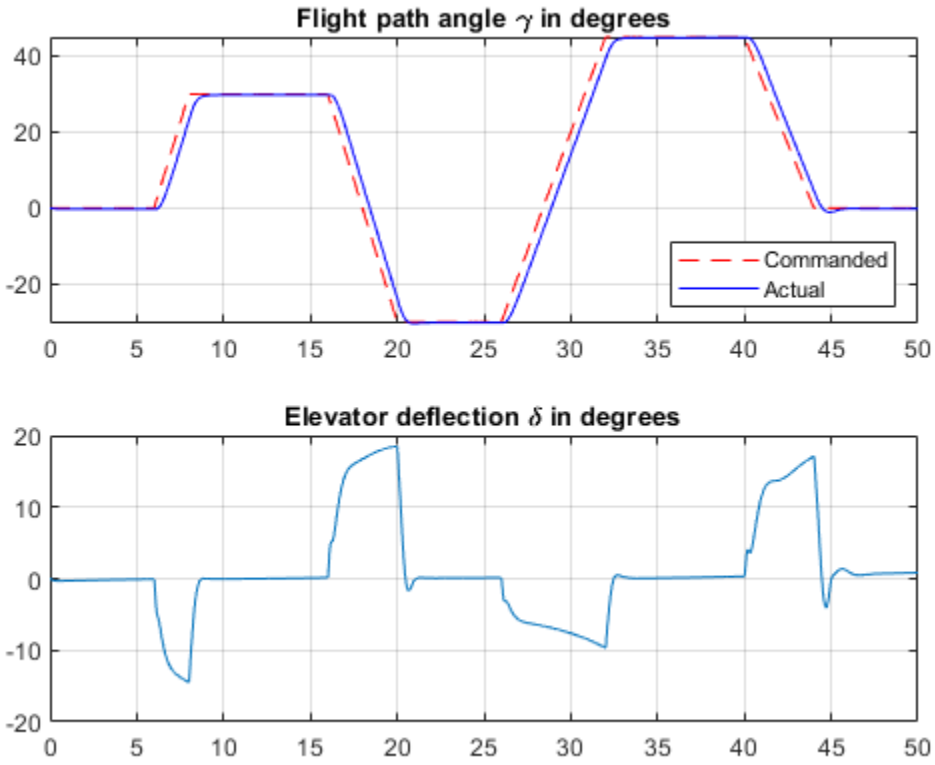
The responses are satisfactory at all flight conditions. Next validate the autopilot against the nonlinear airframe model. First use `writeBlockValue` to apply the tuning results to the Simulink model. This evaluates each gain surface formula at the breakpoints specified in the two Prelookup blocks and writes the result in the corresponding Interpolation block.

```
writeBlockValue(ST)
```

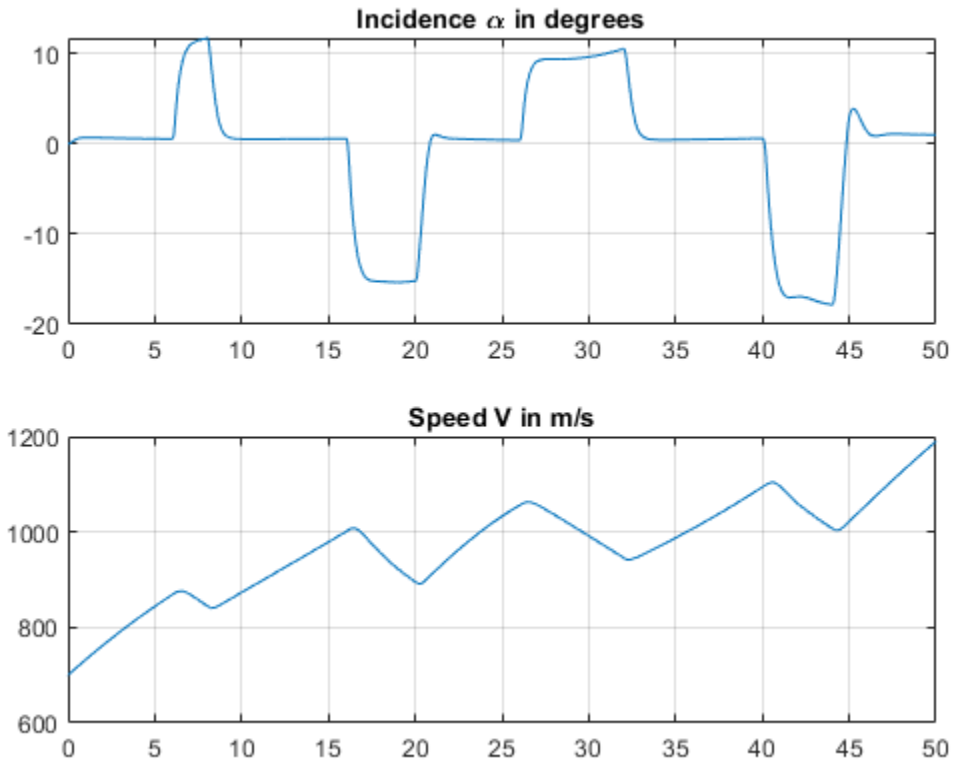
Simulate the autopilot performance for a maneuver that takes the airframe through a large portion of its flight envelope. The code below is equivalent to pressing the Play button in the Simulink model and inspecting the responses in the Scope blocks.

```
% Specify the initial conditions.
h_ini = 1000;
alpha_ini = 0;
v_ini = 700;

% Simulate the model.
SimOut = sim('rct_airframeGS', 'ReturnWorkspaceOutputs', 'on');

% Extract simulation data.
SimData = get(SimOut,'sigsOut');
Sim_gamma = getElement(SimData,'gamma');
Sim_alpha = getElement(SimData,'alpha');
Sim_V = getElement(SimData,'V');
Sim_delta = getElement(SimData,'delta');
Sim_h = getElement(SimData,'h');
Sim_az = getElement(SimData,'az');
```

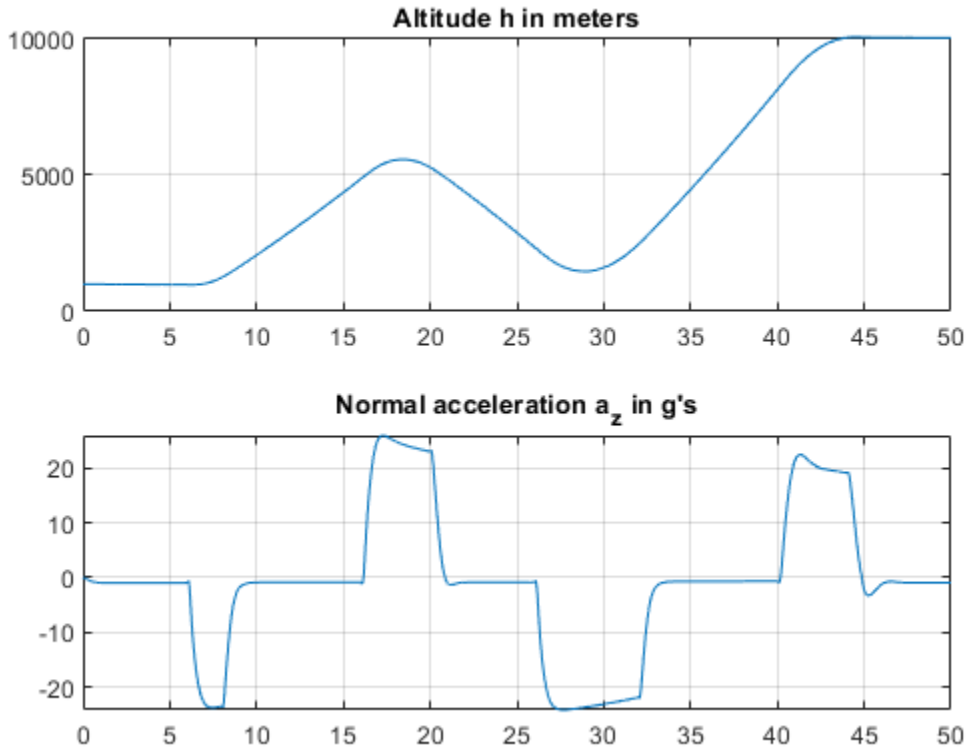
```
t = Sim_gamma.Values.Time;

% Plot the main flight variables.
clf
subplot(2,1,1)
plot(t,Sim_gamma.Values.Data(:,1),'r--',t,Sim_gamma.Values.Data(:,2),'b')
grid
legend('Commanded','Actual','location','SouthEast')
title('Flight path angle \gamma in degrees')
subplot(2,1,2)
plot(t,Sim_delta.Values.Data)
grid
title('Elevator deflection \delta in degrees')
```




```
subplot(2,1,1)
plot(t,Sim_alpha.Values.Data)
grid
title('Incidence \alpha in degrees')
subplot(2,1,2)
plot(t,Sim_V.Values.Data)
grid
title('Speed V in m/s')
```




```
subplot(2,1,1)
plot(t,Sim_h.Values.Data)
grid
title('Altitude h in meters')
subplot(2,1,2)
plot(t,Sim_az.Values.Data)
grid
title('Normal acceleration a_z in g''s')
```



Tracking of the flight path angle profile remains good throughout the maneuver. Note that the variations in incidence α and speed V cover most of the flight envelope considered here ([-20,20] degrees for α and [700,1400] for V). And while the autopilot was tuned for a nominal altitude of 3000 m, it fares well for altitude changing from 1,000 to 10,000 m.

The nonlinear simulation results confirm that the gain-scheduled autopilot delivers consistently high performance throughout the flight envelope. The "gain surface tuning" procedure provides simple explicit formulas for the gain dependence on the scheduling variables. Instead of using look-up tables, you can use these formulas directly for an more memory-efficient hardware implementation.

See Also

`slTuner` | `tunableSurface` | `setBlockParam`

Related Examples

- “Model Gain-Scheduled Control Systems in Simulink” on page 11-4
- “Gain-Scheduled Control of a Chemical Reactor”

More About

- “Gain Scheduling Basics” on page 11-2
- “Parameterize Gain Schedules” on page 11-24

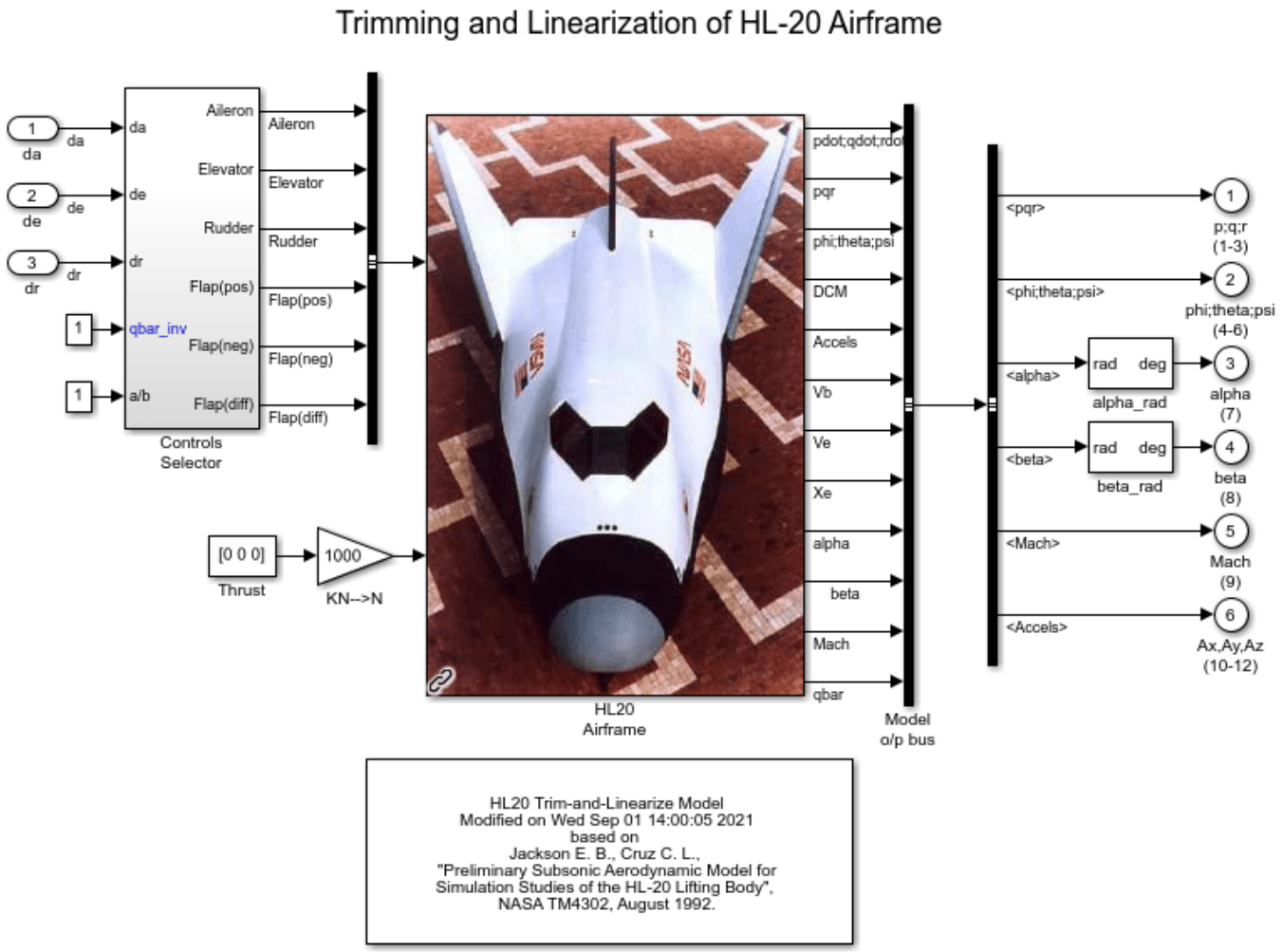
Trimming and Linearization of the HL-20 Airframe

This is Part 1 of a five-part example series on design and tuning of the flight control system for the HL-20 vehicle. This part deals with trimming and linearization of the airframe.

HL-20 Model

The HL-20 model is adapted from the model described in “NASA HL-20 Lifting Body Airframe” (Aerospace Blockset). This is a 6-DOF model of the vehicle during the final descent and landing phase of the flight. No thrust is used during this phase and the airframe is gliding to the landing strip.

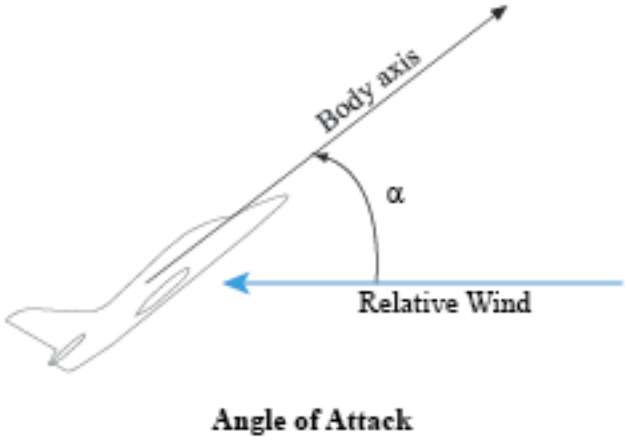
```
open_system('csthl20_trim')
```

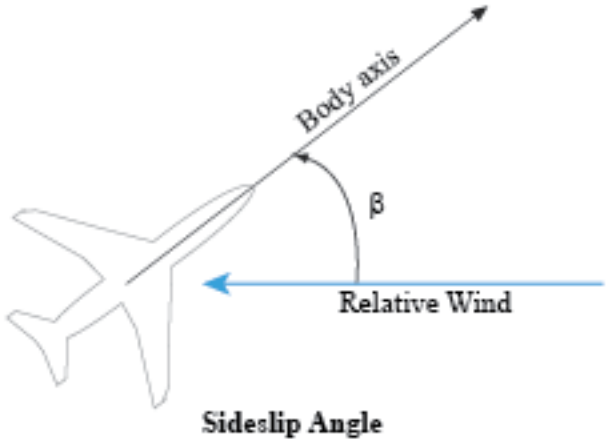


This version of the model includes the equations of motion (EOM), the force and moment calculation from the aerodynamic tables, the environment model, and the "Controls Selector" block which maps aileron, elevator, and rudder demands to deflections of the six control surfaces.

Batch Trimming

Trimming consists of calculating aileron, elevator, and rudder deflections that zero out the forces and moments on the airframe, or equivalently, keep the body velocities ub,vb,wb and angular rates p,q,r steady. Because thrust is not used during descent, one degree-of-freedom is lost and the trim condition must be relaxed to let ub to vary. The trim values of the deflections da,de,dr depend on the airframe orientation relative to the wind. This orientation is characterized by the angle-of-attack (AoA) alpha and the sideslip angle (AoS) beta.





With the `operspec` and `findop` functions, you can efficiently compute the trim deflections over a grid of (alpha,beta) values covering the operating range of the vehicle. Here we trim the model for 8 values of alpha ranging from -10 to 25 degrees, and 5 values of beta ranging from -10 to +10 degrees. The nominal altitude and speed are set to 10,000 feet and Mach 0.6.

```
d2r = pi/180;            % degrees to radians
m2ft = 3.28084;          % meter to feet
Altitude = 10000/m2ft;   % Nominal altitude
Mach = 0.6;              % Nominal Mach
alpha_vec = -10:5:25;     % Alpha Range
beta_vec = -10:5:10;     % Beta Range
[alpha,beta] = ndgrid(alpha_vec,beta_vec);   % (Alpha,Beta) grid
```

Use `operspec` to create an array of operating point specifications.

```
opspec = operspec('csthl20_trim',size(alpha));

opspec(1)


ans =


 Operating point specification for the Model csthl20_trim.
 (Time-Varying Components Evaluated at time t=0)

States:
----------
       <strong>x</strong>        <strong>Known</strong>    <strong>SteadyState</strong>    <stro
```

```
    <strong>________</strong>    <strong>_____</strong>    <strong>___________</strong>    <stron

(1.) csthl20_trim/HL20 Airframe/6DOF (Euler Angles)/Calculate DCM & Euler Angles/phi theta psi
           0    false       true        -Inf    Inf    -Inf      Inf
    -0.19945    false       true        -Inf    Inf    -Inf      Inf
           0    false       true        -Inf    Inf    -Inf      Inf
(2.) csthl20_trim/HL20 Airframe/6DOF (Euler Angles)/p,q,r
           0    false       true        -Inf    Inf    -Inf      Inf
           0    false       true        -Inf    Inf    -Inf      Inf
           0    false       true        -Inf    Inf    -Inf      Inf
(3.) csthl20_trim/HL20 Airframe/6DOF (Euler Angles)/ub,vb,wb
      202.67    false       true        -Inf    Inf    -Inf      Inf
           0    false       true        -Inf    Inf    -Inf      Inf
      23.257    false       true        -Inf    Inf    -Inf      Inf
(4.) csthl20_trim/HL20 Airframe/6DOF (Euler Angles)/xe,ye,ze
      -12072    false       true        -Inf    Inf    -Inf      Inf
           0    false       true        -Inf    Inf    -Inf      Inf
       -3048    false       true        -Inf    Inf    -Inf      Inf

Inputs:
----------
    <strong>u</strong>    <strong>Known</strong>    <strong>Min</strong>     <strong>Max</strong>
    <strong>_</strong>    <strong>_____</strong>    <strong>____</strong>    <strong>___</strong>

(1.) csthl20_trim/da
    0    false    -Inf    Inf
(2.) csthl20_trim/de
    0    false    -Inf    Inf
(3.) csthl20_trim/dr
    0    false    -Inf    Inf

Outputs:
----------
    <strong>y</strong>    <strong>Known</strong>    <strong>Min</strong>     <strong>Max</strong>
    <strong>_</strong>    <strong>_____</strong>    <strong>____</strong>    <strong>___</strong>

(1.) csthl20_trim/p;q;r (1-3)
    0    false    -Inf    Inf
    0    false    -Inf    Inf
    0    false    -Inf    Inf
(2.) csthl20_trim/phi;theta;psi (4-6)
    0    false    -Inf    Inf
    0    false    -Inf    Inf
    0    false    -Inf    Inf
(3.) csthl20_trim/alpha (7)
    0    false    -Inf    Inf
(4.) csthl20_trim/beta (8)
    0    false    -Inf    Inf
(5.) csthl20_trim/Mach (9)
    0    false    -Inf    Inf
(6.) csthl20_trim/Ax,Ay,Az (10-12)
    0    false    -Inf    Inf
    0    false    -Inf    Inf
    0    false    -Inf    Inf
```

Specify the equilibrium conditions for each orientation of the airframe. To do this:

- Specify the orientation by fixing the outputs alpha and beta to their desired values.

- Specify the airframe speed by fixing the Mach output to 0.6.
- Mark the angular rates p,q,r as steady.
- Mark the velocities vb and wb as steady.

```
for ct=1:40
   % Specify alpha angle
   opspec(ct).Outputs(3).y = alpha(ct);
   opspec(ct).Outputs(3).Known = true;
   % Specify beta angle
   opspec(ct).Outputs(4).y = beta(ct);
   opspec(ct).Outputs(4).Known = true;
   % Specify Mach speed
   opspec(ct).Outputs(5).y = Mach;
   opspec(ct).Outputs(5).Known = true;
   % Mark p,q,r as steady
   opspec(ct).States(2).SteadyState = true(3,1);
   % Mark vb,wb as steady
   opspec(ct).States(3).SteadyState = [false;true;true];
   % (phi,theta,psi) and (Xe,Ye,Ze) are not steady
   opspec(ct).States(1).SteadyState = false(3,1);
   opspec(ct).States(4).SteadyState = false(3,1);
end
```

To fully characterize the trim condition, also

- Set p=0 to prevent rolling.
- Set the roll/pitch/yaw angles (phi,theta,psi) to (0,alpha,beta) to align the wind and earth frames.
- Specify the airframe position (Xe,Ye,Ze) as (0,0,-Altitude).

```
for ct=1:40
   % Set (phi,theta,psi) to (0,alpha,beta)
   opspec(ct).States(1).x = [0 ; alpha(ct)*d2r ; beta(ct)*d2r];
   opspec(ct).States(1).Known = true(3,1);
   % Set p=0 (no rolling)
   opspec(ct).States(2).x(1) = 0;
   opspec(ct).States(2).Known(1) = true;
   % Set (Xe,Ye,Ze) to (0,0,-Altitude)
   opspec(ct).States(4).x = [0 ; 0 ; -Altitude];
   opspec(ct).States(4).Known = true(3,1);
end
```

Now use `findop` to compute the trim conditions for all 40 (alpha,beta) combinations in one go. This batch mode approach involves a single compilation of the model. FINDOP uses optimization to solve the nonlinear equations characterizing each equilibrium. Here we use the "SQP" algorithm for this task.

```
% Set options for FINDOP solver
TrimOptions = findopOptions;
TrimOptions.OptimizationOptions.Algorithm = 'sqp';
TrimOptions.DisplayReport = 'off';

% Trim model
[ops,rps] = findop('csthl20_trim',opspec,TrimOptions);
```

This returns 8-by-5 arrays OPS (operating conditions) and RPS (optimization reports). You can use RPS to verify that each trim condition was successfully calculated. Results for the first (alpha,beta) pair are shown below.

```
[alpha(1) beta(1)]


ans =

   -10   -10


ops(1)


ans =


 Operating point for the Model csthl20_trim.
 (Time-Varying Components Evaluated at time t=0)

States:
----------
       <strong>x</strong>
    <strong>________</strong>

(1.) csthl20_trim/HL20 Airframe/6DOF (Euler Angles)/Calculate DCM & Euler Angles/phi theta psi
           0
    -0.17453
    -0.17453
(2.) csthl20_trim/HL20 Airframe/6DOF (Euler Angles)/p,q,r
           0
    -0.15825
    0.008004
(3.) csthl20_trim/HL20 Airframe/6DOF (Euler Angles)/ub,vb,wb
      191.09
     -34.214
     -33.695
(4.) csthl20_trim/HL20 Airframe/6DOF (Euler Angles)/xe,ye,ze
           0
           0
       -3048

Inputs:
----------
       <strong>u</strong>
    <strong>_______</strong>

(1.) csthl20_trim/da
    -23.984
(2.) csthl20_trim/de
    -6.4896
(3.) csthl20_trim/dr
     4.0858

rps(1).TerminationString


ans =
```

```
    'Operating point specifications were successfully met.'
```

Batch Linearization

The gains of the flight control system are typically scheduled as a function of alpha and beta, see Part 2 (“Angular Rate Control in the HL-20 Autopilot”) for more details. To tune these gains, you need linearized models of the HL-20 airframe at the 40 trim conditions. Use `linearize` to compute these models from the trim operating conditions `ops`.

```
% Linearize airframe dynamics at each trim condition
G = linearize('csthl20_trim','csthl20_trim/HL20 Airframe',ops);

size(G)
```

```
8x5 array of state-space models.
Each model has 34 outputs, 9 inputs, and 12 states.
```

The linear equivalent of the "Controls Selector" block depends on the amount of elevator deflection and should be computed for qbar_inv=1 (nominal dynamic pressure at Mach=0.6). For convenience, also linearize this block at the 40 trim conditions.

```
CS = linearize('csthl20_trim','csthl20_trim/Controls Selector',ops);

% Zero out a/b and qbar_inv channels
CS = [CS(:,1:3) zeros(6,2)];
```

Linear Model Simplification

The linearized airframe models have 12 states:

```
xG = G.StateName
```

```
xG =

  12x1 cell array

    {'phi theta psi(1)'}
    {'phi theta psi(2)'}
    {'phi theta psi(3)'}
    {'p,q,r (1)'       }
    {'p,q,r (2)'       }
    {'p,q,r (3)'       }
    {'ub,vb,wb(1)'     }
    {'ub,vb,wb(2)'     }
    {'ub,vb,wb(3)'     }
    {'xe,ye,ze(1)'     }
    {'xe,ye,ze(2)'     }
    {'xe,ye,ze(3)'     }
```

Some states are not under the authority of the roll/pitch/yaw autopilot and other states contribute little to the design of this autopilot. For control purposes, the most important states are the roll angle phi, the body velocities ub,vb,wb, and the angular rates p,q,r. Accordingly, use `modred` to obtain a 7th-order model that only retains these states.

```
G7 = G;
xKeep = {...
   'phi theta psi(1)'
   'ub,vb,wb(1)'
   'ub,vb,wb(2)'
   'ub,vb,wb(3)'
   'p,q,r(1)'
   'p,q,r(2)'
   'p,q,r(3)'};
[~,xElim] = setdiff(xG,xKeep);
for ct=1:40
   G7(:,:,ct) = modred(G(:,:,ct),xElim,'truncate');
end
```

With these linearized models in hand, you can move to the task of tuning and scheduling the flight control system gains. See “Angular Rate Control in the HL-20 Autopilot” for Part 2 of this example.

See Also

More About

- “Angular Rate Control in the HL-20 Autopilot”
- “Model Gain-Scheduled Control Systems in Simulink”

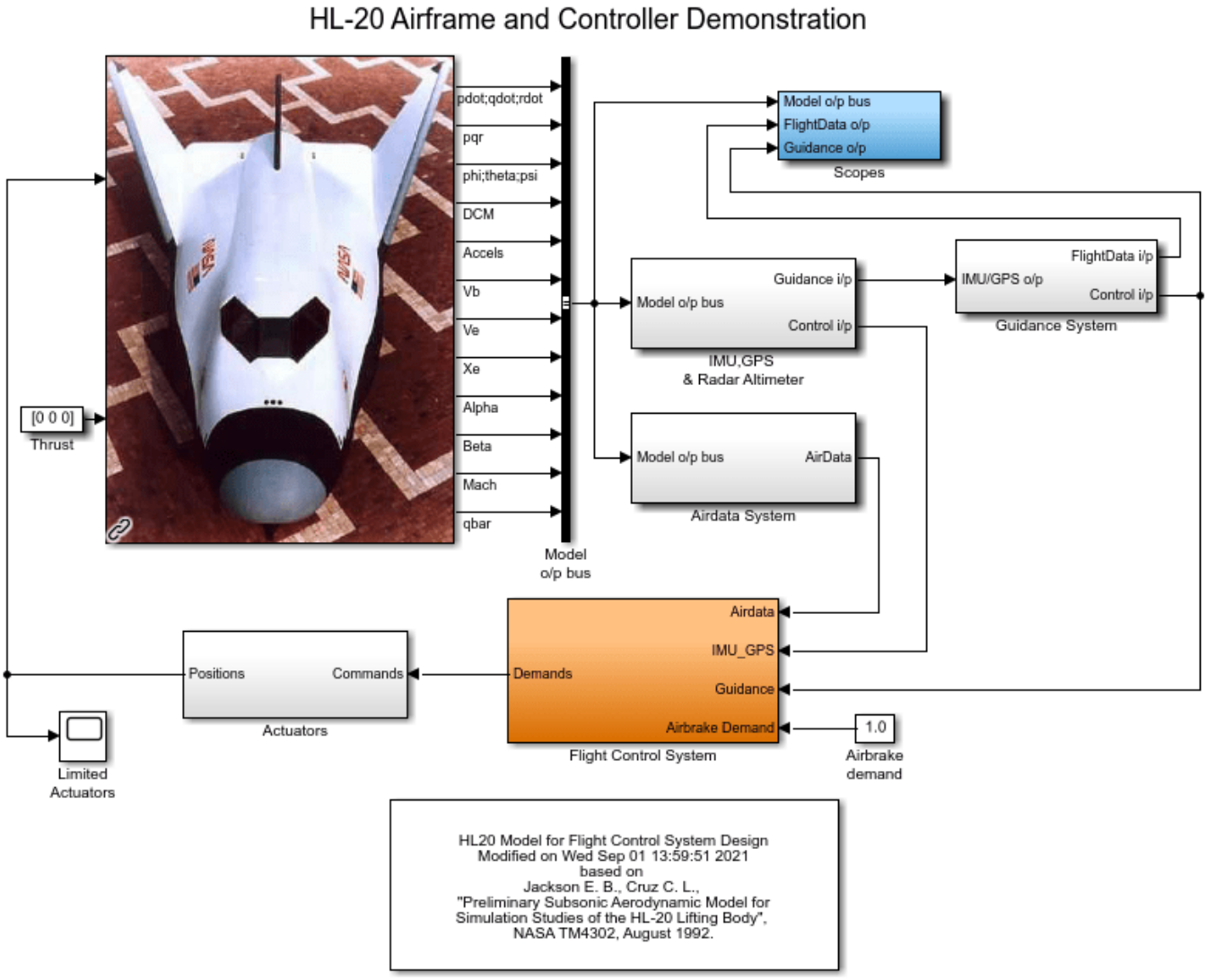
Angular Rate Control in the HL-20 Autopilot

This is Part 2 of the example series on design and tuning of the flight control system for the HL-20 vehicle. This part deals with closing the inner loops controlling the body angular rates.

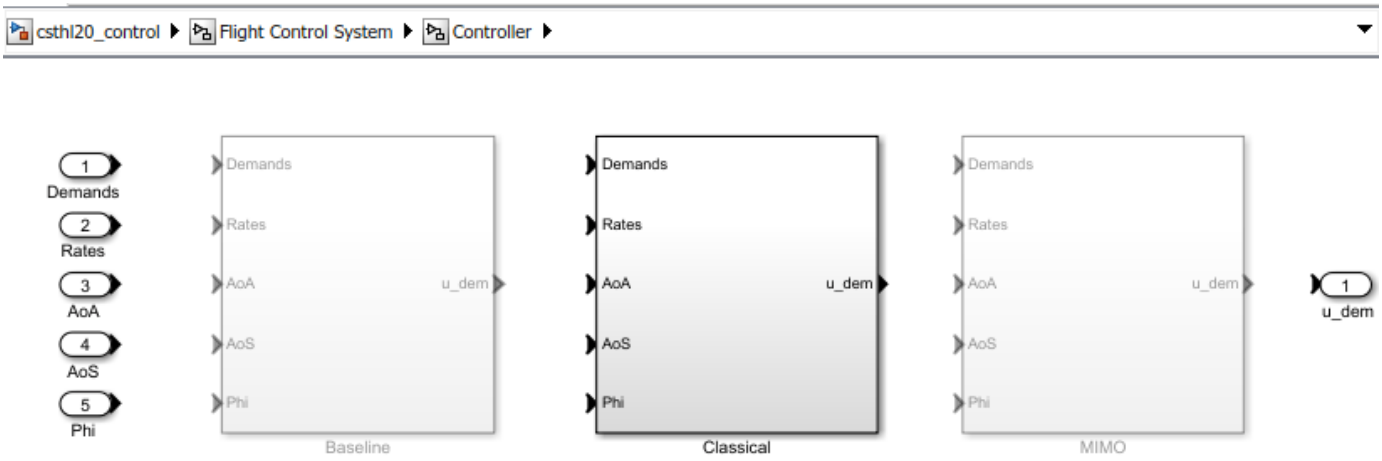
Control Architecture

Open the HL-20 model with its flight control system.

```
open_system('csthl20_control')
```



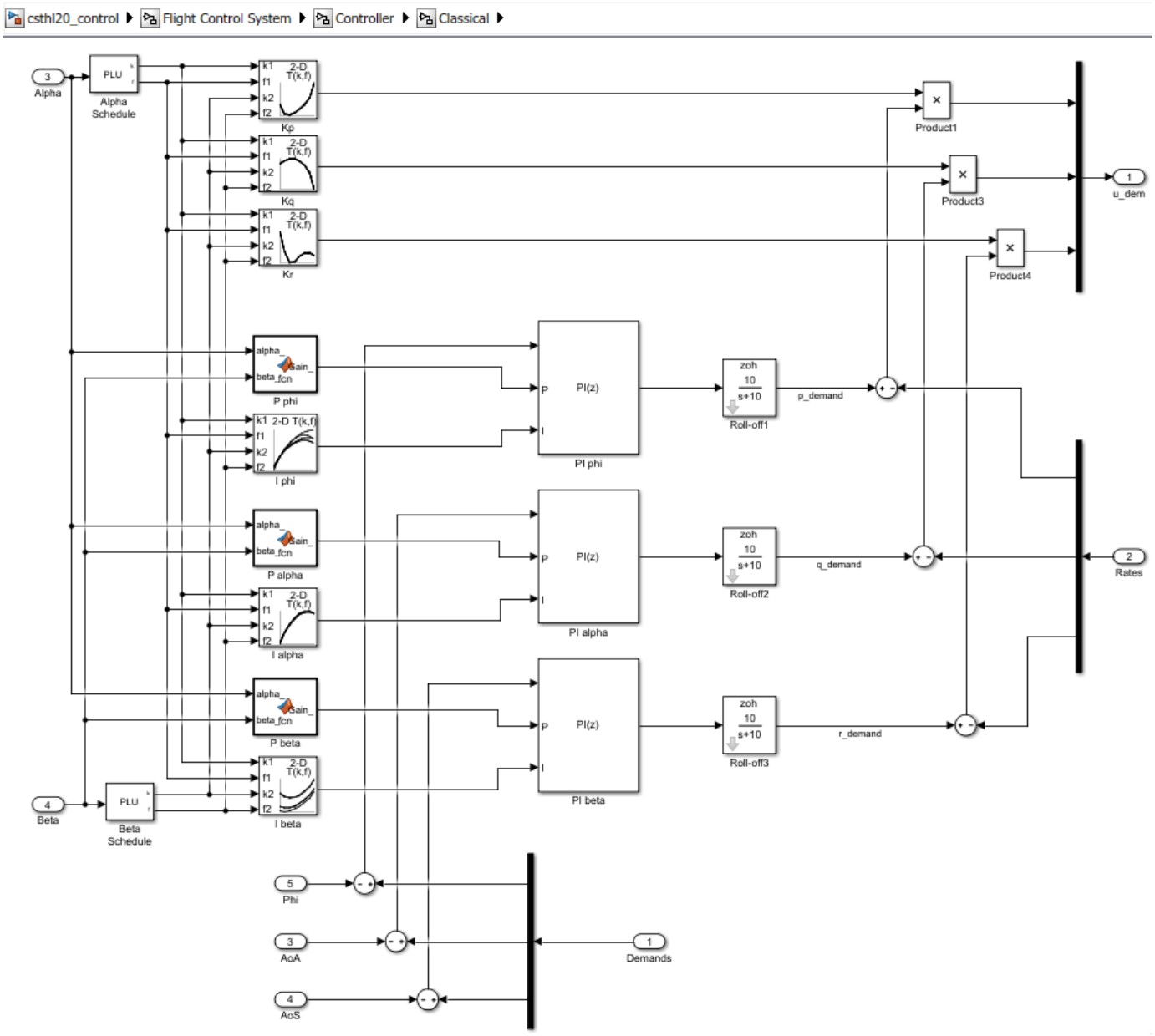
This 6-DOF model is adapted from “NASA HL-20 Lifting Body Airframe” (Aerospace Blockset). The model is configured to simulate the final approach to the landing site. The "Guidance System" generates the glideslope trajectory and corresponding roll, angle of attack (alpha), and sideslip angle (beta) commands. The "Flight Control System" is tasked with adjusting the control surfaces to track these commands. The "Controller" block inside the "Flight Control System" is a variant subsystem with different autopilot configurations.



The "Baseline" and "Classical" controllers use a classic cascaded-loop architecture with three inner P-only loops to control the angular rates p,q,r, and three outer PI loops to control the angular positions phi,alpha,beta. The six proportional gains and three integral gains are all scheduled as a function of alpha and beta. The "Baseline" variant contains the baseline design featured in “NASA HL-20 Lifting Body Airframe” (Aerospace Blockset). Parts 2 and 3 of this series use the "Classical" variant to walk through the tuning process. The active variant is controlled by the workspace variable CTYPE. Set its value to 2 to activate the "Classical" variant of the controller.

```
% Select "Classical" variant of controller
CTYPE = 2;

% call model update to make sure only active variant signals are analyzed during linearization
set_param('csthl20_control', 'SimulationCommand', 'update');
```

Note that this variant uses a mix of lookup tables and MATLAB Function blocks to schedule the autopilot gains.



Setup for Controller Tuning

In Part 1 of this series (“Trimming and Linearization of the HL-20 Airframe”), we obtained linearized models of the "HL20 Airframe" and "Controls Selector" blocks for 40 different aircraft orientations (40 different pairs of (alpha,beta) values). Load these arrays of linearized models.

```
load csthl20_TrimData G7 CS
```

```
size(G7)
```

```
8x5 array of state-space models.
Each model has 34 outputs, 9 inputs, and 7 states.
```

```
size(CS)
```

```
8x5 array of state-space models.
Each model has 6 outputs, 5 inputs, and 0 states.
```

The `slTuner` interface is a convenient way to obtain linearized models of "csthl20_control" that are suitable for control system design and analysis. Through this interface you can designate the signals and points of interest in the model and specify which blocks you want to tune.

```
ST0 = slTuner('csthl20_control');
ST0.Ts = 0;   % ask for continuous-time linearizations
```

Here the points of interest include the angular and rate demands, the corresponding responses, and the deflections da,de,dr.

```
AP = {'da;de;dr'
   'HL20 Airframe/pqr'
   'Alpha_deg'
   'Beta_deg'
   'Phi_deg'
   'Controller/Classical/Demands'  % angular demands
   'p_demand'
   'q_demand'
   'r_demand'};
ST0.addPoint(AP)
```

Since we already obtained linearized models of the "HL20 Airframe" and "Controls Selector" blocks as a function of (alpha,beta), the simplest way to linearize the entire model "csthl20_control" is to replace each nonlinear component by a family of linear models. This is called "block substitution" and is often the most effective way to linearize complex models at multiple operating conditions.

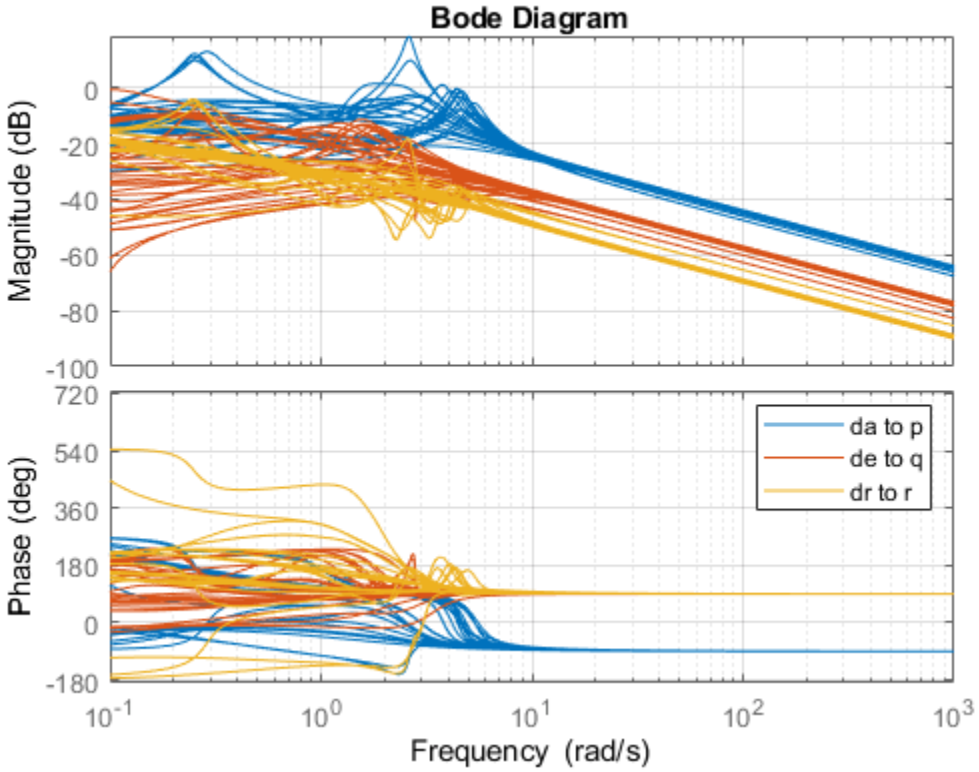
```
% Replace "HL20 Airframe" block by 8-by-5 array of linearized models G7
BlockSub1 = struct('Name','csthl20_control/HL20 Airframe','Value',G7);

% Replace "Controls Selector" by CS
BlockSub2 = struct('Name','csthl20_control/Flight Control System/Controls Selector','Value',CS);

% Replace "Actuators" by direct feedthrough (ignore saturations and second-order actuator dynamic
BlockSub3 = struct('Name','csthl20_control/Actuators','Value',eye(6));

ST0.BlockSubstitutions = [BlockSub1 ; BlockSub2 ; BlockSub3];
```

You are now ready for the control design part.

Closing the Inner Loops

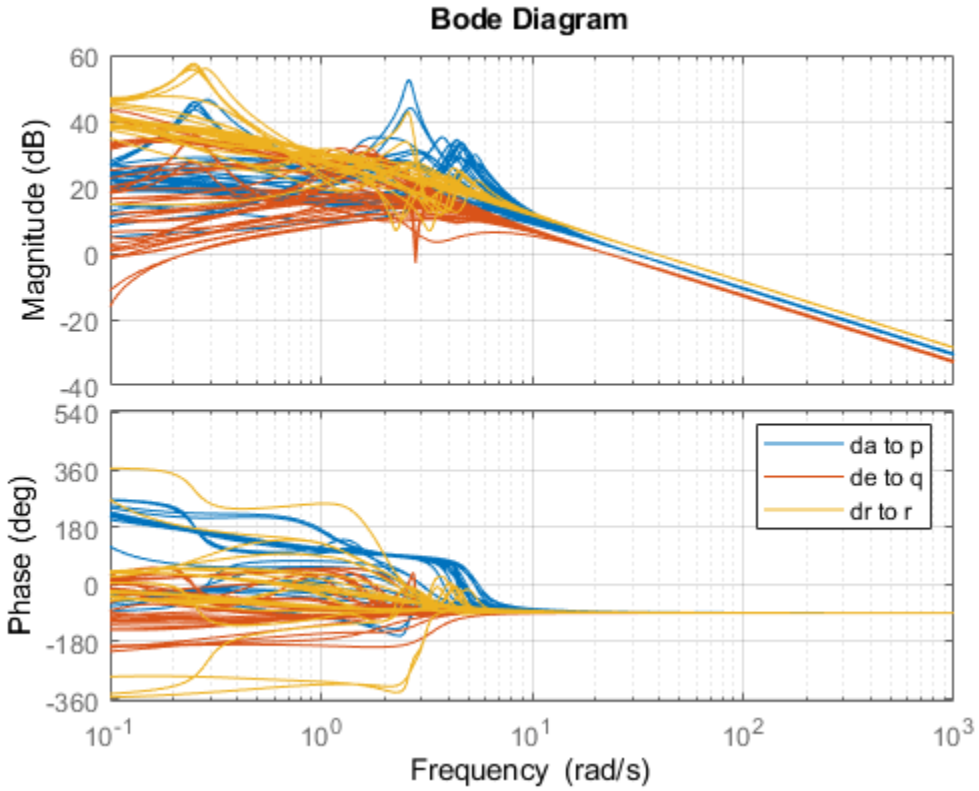
Begin with the three inner loops controlling the angular rates p,q,r. To get oriented, plot the open-loop transfer function from deflections (da,de,dr) to angular rates (p,q,r). With the `slTuner` interface, you can query the model for any transfer function of interest.

```
% NOTE: The second 'da;de;dr' opens all feedback loops at the plant input
Gpqr = getIOTransfer(ST0,'da;de;dr','pqr','da;de;dr');

bode(Gpqr(1,1),Gpqr(2,2),Gpqr(3,3),{1e-1,1e3}), grid
legend('da to p','de to q','dr to r')
```



This Bode plot suggests that the diagonal terms behave as integrators (up to the sign) beyond 5 rad/s. This justifies using proportional-only control. Consistent with the baseline design, set the target bandwidth for the p,q,r loops to 30, 22.5, and 37.5 rad/s, respectively. The gains Kp, Kq, Kr for each (alpha,beta) value are readily obtained from the plant frequency response at these frequencies, and the phase plots indicate that Kp should be positive (negative feedback) and Kq, Kr should be negative (positive feedback).

```
% Compute Kp,Kq,Kr for each (alpha,beta) condition. Resulting arrays
% have size [1 1 8 5]
Kp = 1./abs(evalfr(Gpqr(1,1),30i));
Kq = -1./abs(evalfr(Gpqr(2,2),22.5i));
Kr = -1./abs(evalfr(Gpqr(3,3),37.5i));

bode(Gpqr(1,1)*Kp,Gpqr(2,2)*Kq,Gpqr(3,3)*Kr,{1e-1,1e3}), grid
legend('da to p','de to q','dr to r')
```



To conclude the inner-loop design, push these gain values to the corresponding lookup tables in the Simulink model and refresh the `slTuner` object.

```
MWS = get_param('csthl20_control','ModelWorkspace');
MWS.assignin('Kp',squeeze(Kp))
MWS.assignin('Kq',squeeze(Kq))
MWS.assignin('Kr',squeeze(Kr))

refresh(ST0)
```

Next you need to tune the outer loops controlling roll, angle of attack, and sideslip angle. Part 3 of this series (“Attitude Control in the HL-20 Autopilot - SISO Design”) shows how to tune a classic SISO architecture and Part 4 (“Attitude Control in the HL-20 Autopilot - MIMO Design”) looks into the benefits of a MIMO architecture.

See Also

More About

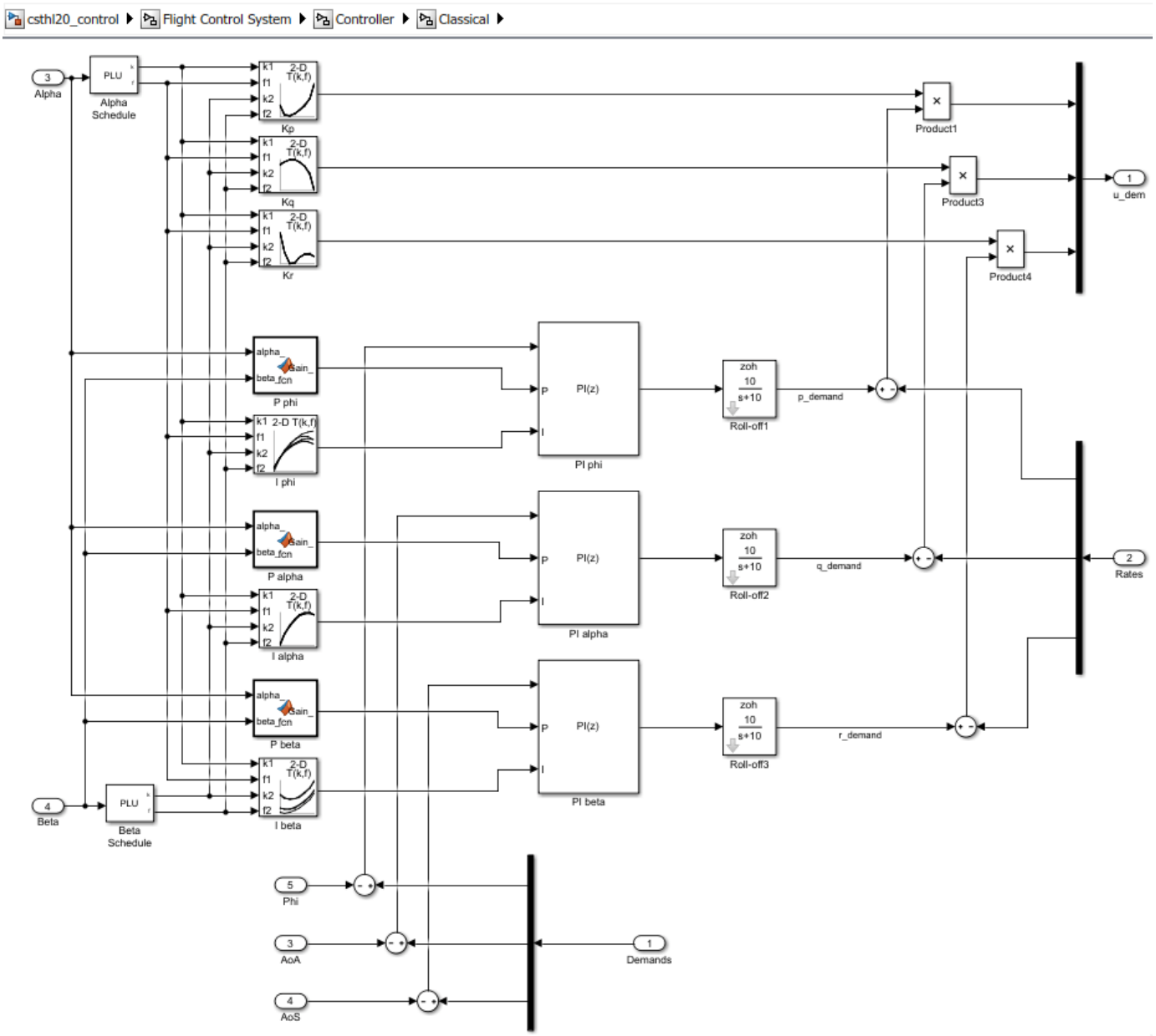
- “Trimming and Linearization of the HL-20 Airframe”
- “Attitude Control in the HL-20 Autopilot - SISO Design”
- “Tune Gain Schedules in Simulink”

Attitude Control in the HL-20 Autopilot - SISO Design

This is Part 3 of the example series on design and tuning of the flight control system for the HL-20 vehicle. This part shows how to tune a classic SISO architecture for controlling the roll, pitch, and yaw of the vehicle.

Background

This example uses the HL-20 model adapted from “NASA HL-20 Lifting Body Airframe” (Aerospace Blockset), see Part 1 of the series (“Trimming and Linearization of the HL-20 Airframe”) for details. The autopilot controlling the attitude of the aircraft consists of three inner loops and three outer loops.



In Part 2 (“Angular Rate Control in the HL-20 Autopilot”), we showed how to close the inner loops controlling the angular rates p,q,r. The following commands recap the corresponding steps. Note that this creates and configures an `slTuner` interface `ST0` for interacting with the Simulink model.

```
load_system('csthl20_control')
CTYPE = 2;  % Select SISO architecture
HL20recapPart2

ST0
```

```
slTuner tuning interface for "csthl20_control":

No tuned blocks. Use the addBlock command to add new blocks.
9 Analysis points:
--------------------------
Point 1: Signal "da;de;dr", located at 'Output Port 1' of csthl20_control/Flight Control System/C
Point 2: Signal "pqr", located at 'Output Port 2' of csthl20_control/HL20 Airframe
Point 3: 'Output Port 1' of csthl20_control/Flight Control System/Alpha_deg
Point 4: 'Output Port 1' of csthl20_control/Flight Control System/Beta_deg
Point 5: 'Output Port 1' of csthl20_control/Flight Control System/Phi_deg
Point 6: 'Output Port 1' of csthl20_control/Flight Control System/Controller/Classical/Demands
Point 7: Signal "p_demand", located at 'Output Port 1' of csthl20_control/Flight Control System/C
Point 8: Signal "q_demand", located at 'Output Port 1' of csthl20_control/Flight Control System/C
Point 9: Signal "r_demand", located at 'Output Port 1' of csthl20_control/Flight Control System/C

No permanent openings. Use the addOpening command to add new permanent openings.
Properties with dot notation get/set access:
      Parameters         : []
      OperatingPoints    : [] (model initial condition will be used.)
      BlockSubstitutions : [3x1 struct]
      Options            : [1x1 linearize.SlTunerOptions]
      Ts                 : 0
```

Setup for Outer Loop Tuning

We now shift focus to the three gain-scheduled PI loops controlling roll (phi), angle of attack (alpha), and sideslip angle (beta). These loops could be tuned one at a time (3 loops and 40 operating points equals 120 design points). You could also use `pidtune` to tune the PI gains in batch mode for specific target bandwidth and phase margin requirements. Both approaches have caveats:

- It is difficult to account for loop interactions.
- The gains obtained at each design point may be inconsistent and require smoothing across operating points.

An alternative approach is the concept of "Gain Surface Tuning" [1] where you parameterize the gain schedules P(alpha,beta) and I(alpha,beta) as polynomial surfaces and use `systune` to tune the polynomial coefficients. This approach tackles all operating points at once and can account for loop interactions, in particular for stability margin considerations. This is the approach showcased here.

To tune the outer loops, we must close the inner loops and obtain a linearized model of the "plant" seen by the outer loops at each (alpha,beta) condition. We could ask `slTuner` to compute the corresponding transfer function, but this would effectively fix the inner-loop gains Kp,Kq,Kr to their values at the default operating condition. To get the correct linearization, we must tell `slTuner` that these gains vary with (alpha,beta). Block substitution is again the simplest way to do this. To mark Kp as varying, find the Product block used to multiply the error signal by Kp, and replace it by an array of gains, one for each (alpha,beta) condition.

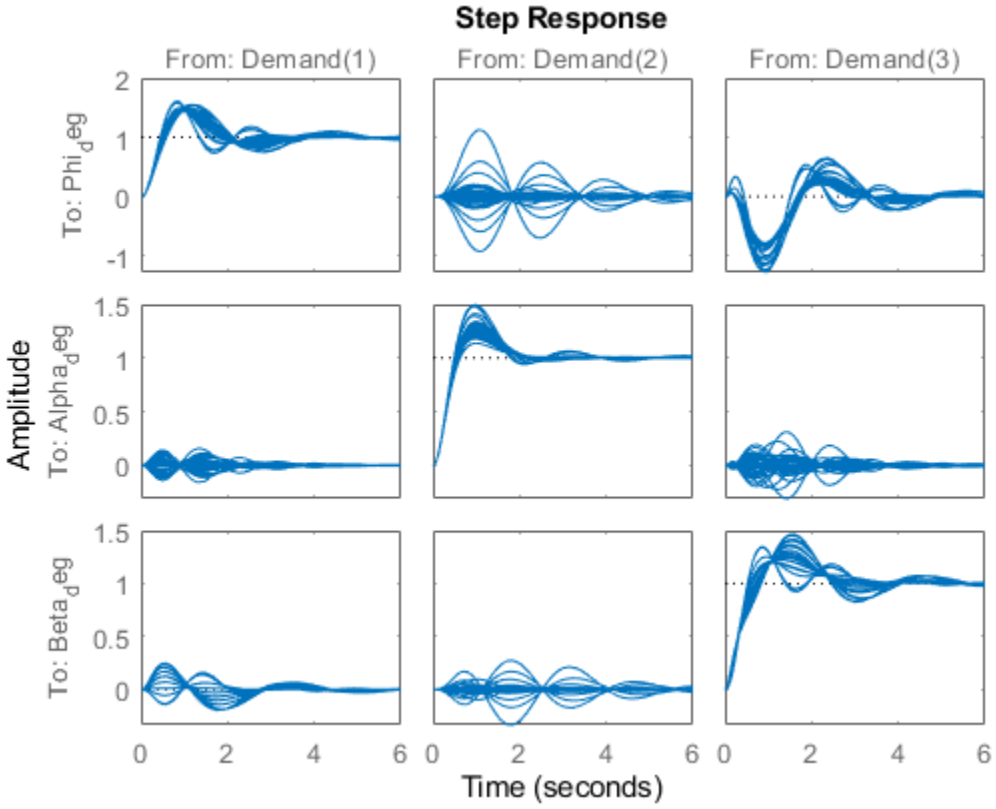
```
ProductBlk = 'csthl20_control/Flight Control System/Controller/Classical/Product1';
BlockSub4 = struct('Name',ProductBlk,'Value',[0 ss(Kp)]);
```

It is easily verified that this block linearization amounts to multiplying the error signal by the varying quantity Kp computed above. Similarly, replace the corresponding Product blocks for Kq and Kr by varying gains.

```
ProductBlk = 'csthl20_control/Flight Control System/Controller/Classical/Product3';
BlockSub5 = struct('Name',ProductBlk,'Value',[0 ss(Kq)]);

ProductBlk = 'csthl20_control/Flight Control System/Controller/Classical/Product4';
BlockSub6 = struct('Name',ProductBlk,'Value',[0 ss(Kr)]);

ST0.BlockSubstitutions = [ST0.BlockSubstitutions ; BlockSub4 ; BlockSub5 ; BlockSub6];
```

You can now plot the angular responses for the initial gain-schedule settings in the model.

```
T0 = getIOTransfer(ST0,'Demand',{'Phi_deg','Alpha_deg','Beta_deg'});
step(T0,6)
```



Tuning Goals

Basic control objectives include bandwidth (response time) and stability margins. Use the "MinLoopGain" and "MaxLoopGain" goals to set the gain crossover of the outer loops between 0.5 and 5 rad/s. Since all loop variables are expressed in degrees, no additional scaling is needed.

```
R1 = TuningGoal.MinLoopGain({'Phi_deg','Alpha_deg','Beta_deg'},0.5,1);
R1.LoopScaling = 'off';
```

```
R2 = TuningGoal.MaxLoopGain({'Phi_deg','Alpha_deg','Beta_deg'},tf(50,[1 10 0]));
R2.LoopScaling = 'off';
```

Use the "Margins" goal to impose adequate stability margins in each loop and across loops. This goal is based on the notion of disk margins which guarantees stability in the face of **concurrent** gain and phase variations in all three loops. Because the target margins of 7 dB and 40 degrees are difficult to obtain for extreme orientations (corners of the (alpha,beta) grid), we use a varying goal to relax the gain and phase margin requirements at the corners.

```
% Gain margins vs (alpha,beta)
GM = [...
   6     6     6     6     6
   6     6     7     6     6
   7     7     7     7     7
   7     7     7     7     7
   7     7     7     7     7
   7     7     7     7     7
   6     6     7     6     6
   6     6     6     6     6];

% Phase margins vs (alpha,beta)
PM = [...
   40         40          40         40        40
   40         40          45         40        40
   45         45          45         45        45
   45         45          45         45        45
   45         45          45         45        45
   45         45          45         45        45
   40         40          45         40        40
   40         40          40         40        40];

% Create varying goal
FH = @(gm,pm) TuningGoal.Margins('da;de;dr',gm,pm);
R3 = varyingGoal(FH,GM,PM);
```

Gain Schedule Tuning

To tune the P and I gain schedules for the outer loop, mark the three MATLAB Function blocks and three lookup table blocks as tunable.

```
TunedBlocks = {'P phi','P alpha','P beta','I phi','I alpha','I beta'};
ST0.addBlock(TunedBlocks)
```

Parameterize each tuned gain schedule as a polynomial surface in alpha and beta. Here we use quadratic surfaces for the proportional gains and multilinear surfaces for the integral gains.

```
% Grid of (alpha,beta) design points
alpha_vec = -10:5:25;     % Alpha Range
beta_vec = -10:5:10;     % Beta Range
[alpha,beta] = ndgrid(alpha_vec,beta_vec);
SG = struct('alpha',alpha,'beta',beta);

% Proportional gains
alphabetaBasis = polyBasis('canonical',2,2);
P_PHI = tunableSurface('Pphi', 0.05, SG, alphabetaBasis);
P_ALPHA = tunableSurface('Palpha', 0.05, SG, alphabetaBasis);
P_BETA = tunableSurface('Pbeta', -0.05, SG, alphabetaBasis);
ST0.setBlockParam('P phi',P_PHI);
```

```
ST0.setBlockParam('P alpha',P_ALPHA);
ST0.setBlockParam('P beta',P_BETA);

% Integral gains
alphaBasis = @(alpha) alpha;
betaBasis = @(beta) abs(beta);
alphabetaBasis = ndBasis(alphaBasis,betaBasis);
I_PHI = tunableSurface('Iphi', 0.05, SG, alphabetaBasis);
I_ALPHA = tunableSurface('Ialpha', 0.05, SG, alphabetaBasis);
I_BETA = tunableSurface('Ibeta', -0.05, SG, alphabetaBasis);
ST0.setBlockParam('I phi',I_PHI);
ST0.setBlockParam('I alpha',I_ALPHA);
ST0.setBlockParam('I beta',I_BETA);
```

Note that we initialized each gain surface to a fixed value suggested by the baseline design. In general, it is not recommended to start from a zero or random initial point because the difficulty of the problem increases the likelihood of getting stuck in uninteresting local minima. Instead, a better strategy consists of tuning a fixed (non-scheduled) set of gains against the full set (or a relevant subset) of design points. Such "robust design" typically provides a good starting point for gain surface tuning.

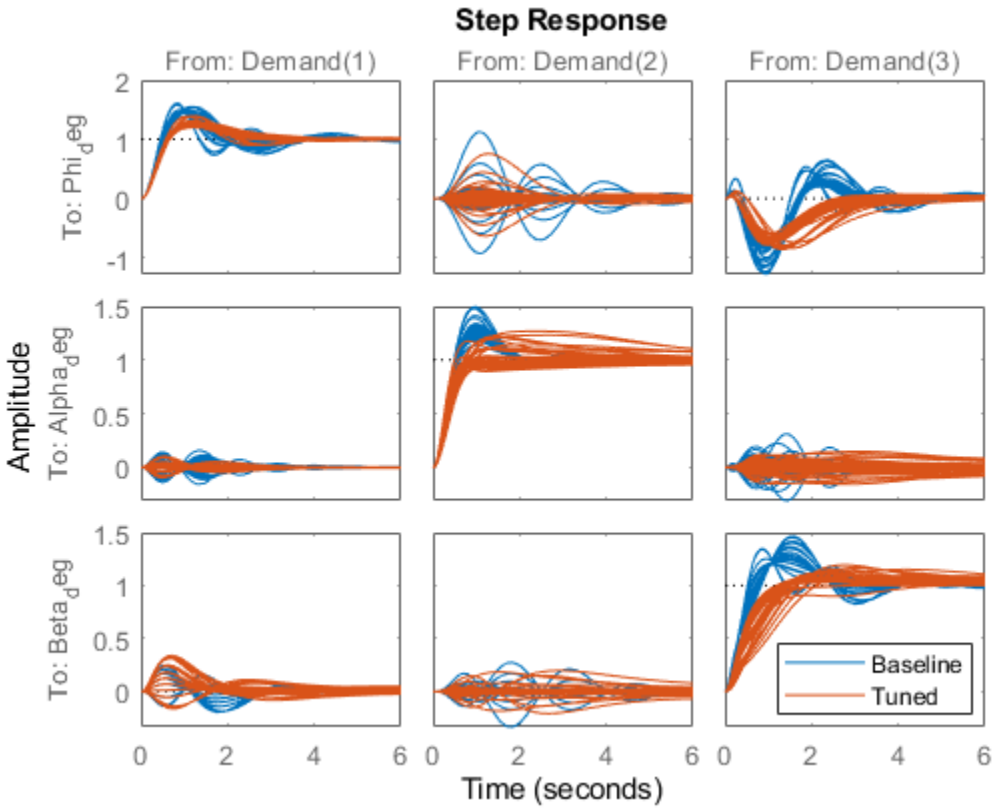
You can now use `systune` to tune the 6 gain surfaces against the three tuning goals.

```
ST = systune(ST0,[R1 R2 R3]);
```

```
Final: Soft = 1.03, Hard = -Inf, Iterations = 41
```

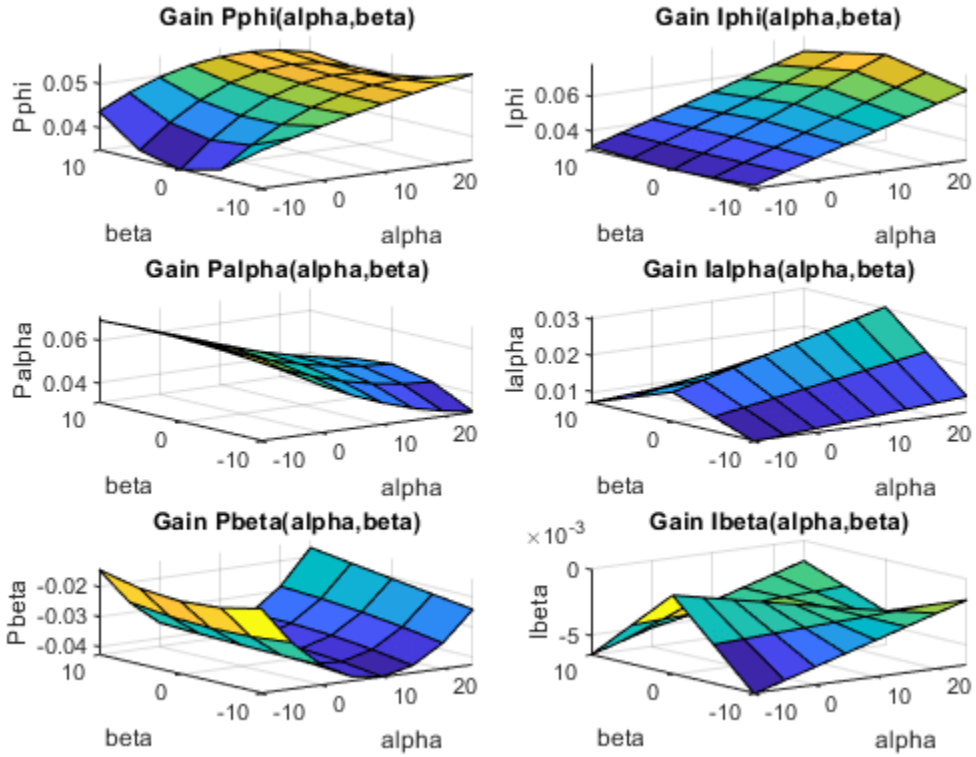
The final objective value is close to 1 so the tuning goals are essentially met. Plot the closed-loop angular responses and compare with the baseline design.

```
T = getIOTransfer(ST,'Demand',{'Phi_deg','Alpha_deg','Beta_deg'});
step(T0,T,6)
legend('Baseline','Tuned','Location','SouthEast')
```



The results are comparable with the baseline with less oscillations in the roll and sideslip responses and a reduced amount of cross-coupling. Use `viewSurf` to inspect the tuned gain surfaces.

```
TV = getTunedValue(ST);
clf
% NOTE: setBlockValue updates each gain surface with the tuned coefficients in TV
subplot(3,2,1)
viewSurf(setBlockValue(P_PHI,TV))
subplot(3,2,3)
viewSurf(setBlockValue(P_ALPHA,TV))
subplot(3,2,5)
viewSurf(setBlockValue(P_BETA,TV))
subplot(3,2,2)
viewSurf(setBlockValue(I_PHI,TV))
subplot(3,2,4)
viewSurf(setBlockValue(I_ALPHA,TV))
subplot(3,2,6)
viewSurf(setBlockValue(I_BETA,TV))
```



Validation

To further validate this design, push the tuned gain surfaces to the Simulink model.

```
writeBlockValue(ST)
```

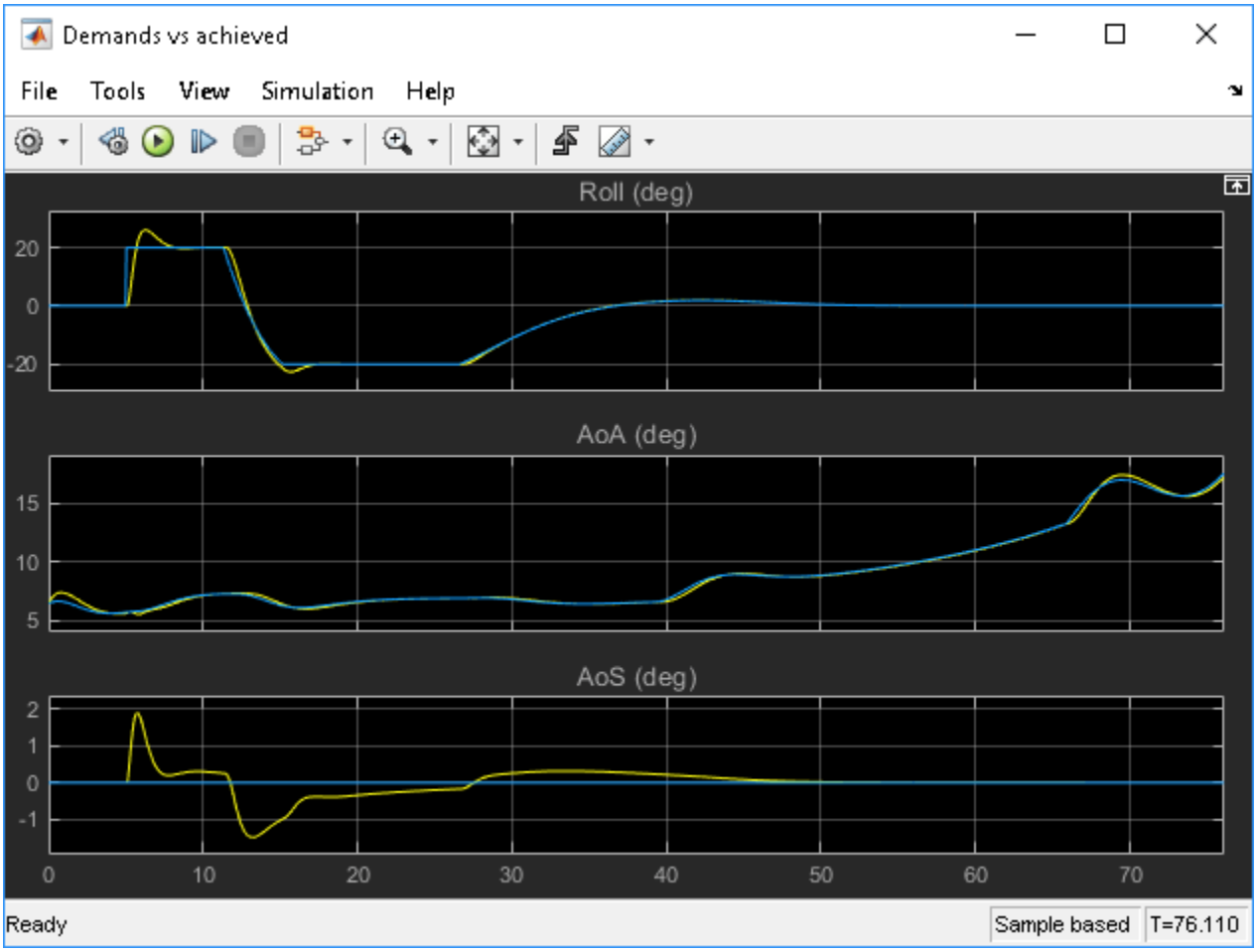
For the three lookup table blocks "I phi", "I alpha", "I beta", `writeBlockValue` samples the gain surfaces at the table breakpoints and updates the table data in the model workspace. For the MATLAB Function blocks "P phi", "P alpha", "P beta", `writeBlockValue` generates MATLAB code for the gain surface equations. For example, the code for the "P phi" block looks like

```
function Gain_ = fcn(alpha_,beta_)
%#codegen

% Type casting
ZERO = zeros(1,1,'like',alpha_+beta_);
alpha_ = cast(alpha_,'like',ZERO);
beta_ = cast(beta_,'like',ZERO);

% Tuned gain surface coefficients
Coeffs = cast([0.0450831184220361 0.00640313776431574 -0.00770552214181431 -0.001451573971(
Offsets = cast([7.5 0],'like',ZERO);
Scalings = cast([17.5 10],'like',ZERO);

% Normalization
alpha_ = (alpha_ - Offsets(1))/Scalings(1);
beta_ = (beta_ - Offsets(2))/Scalings(2);

% Evaluate monic terms for variable alpha_
deg = 2;
Y1 = coder.nullcopy(zeros(deg+1,1,'like',ZERO));
Y1(1) = 1;
Y1(2) = alpha_;
for i1=2:deg
   Y1(i1+1) = alpha_ * Y1(i1);
end

% Evaluate monic terms for variable beta_
deg = 2;
Y2 = coder.nullcopy(zeros(deg+1,1,'like',ZERO));
Y2(1) = 1;
Y2(2) = beta_;
for i2=2:deg
   Y2(i2+1) = beta_ * Y2(i2);
end

% Compute weighted sum of Yj's outer product:
%     Gain = sum  Coeffs(i1,i2,...,iN) Y1(i1) Y2(i2) ... YN(iN)
Gain_ = ZERO;
k = 1;
for i2=1:numel(Y2)
   for i1=1:numel(Y1)
      Gain_ = Gain_ + Coeffs(k) * Y1(i1) * Y2(i2);
      k = k+1;
   end
end
```

Simulink Coder automatically turns this MATLAB code into efficient embedded C code. Whether to use lookup tables or MATLAB Function blocks depends on the application. The MATLAB Function option ensures smooth variation of the gains as a function of alpha and beta (no kinks at breakpoints). It can also be more memory-efficient as it only needs to store the coefficients of the polynomial

equation for the gain surface. On the other hand, evaluating the gain at a given (alpha,beta) point may take a few more operations than in a lookup table, and further adjustment of the gains is easier in a lookup table.

Once you pushed the gains to Simulink, the autopilot tuning is complete and you can simulate its behavior during the landing approach.



The performance is satisfactory but the linear responses showed a significant amount of cross-coupling between axes and we could not quite meet the stability margins target at the corner points of the (alpha,beta) range. Would it be beneficial to use a MIMO architecture that combines all three measurements of phi, alpha, beta to calculate the surface deflections? This idea is further explored in Part 4 of this series (“Attitude Control in the HL-20 Autopilot - MIMO Design”).

References

[1] P. Gahinet and P. Apkarian, "Automated tuning of gain-scheduled control systems," in Proc. IEEE Conf. Decision and Control, Dec 2013.

See Also

`tunableSurface`

More About

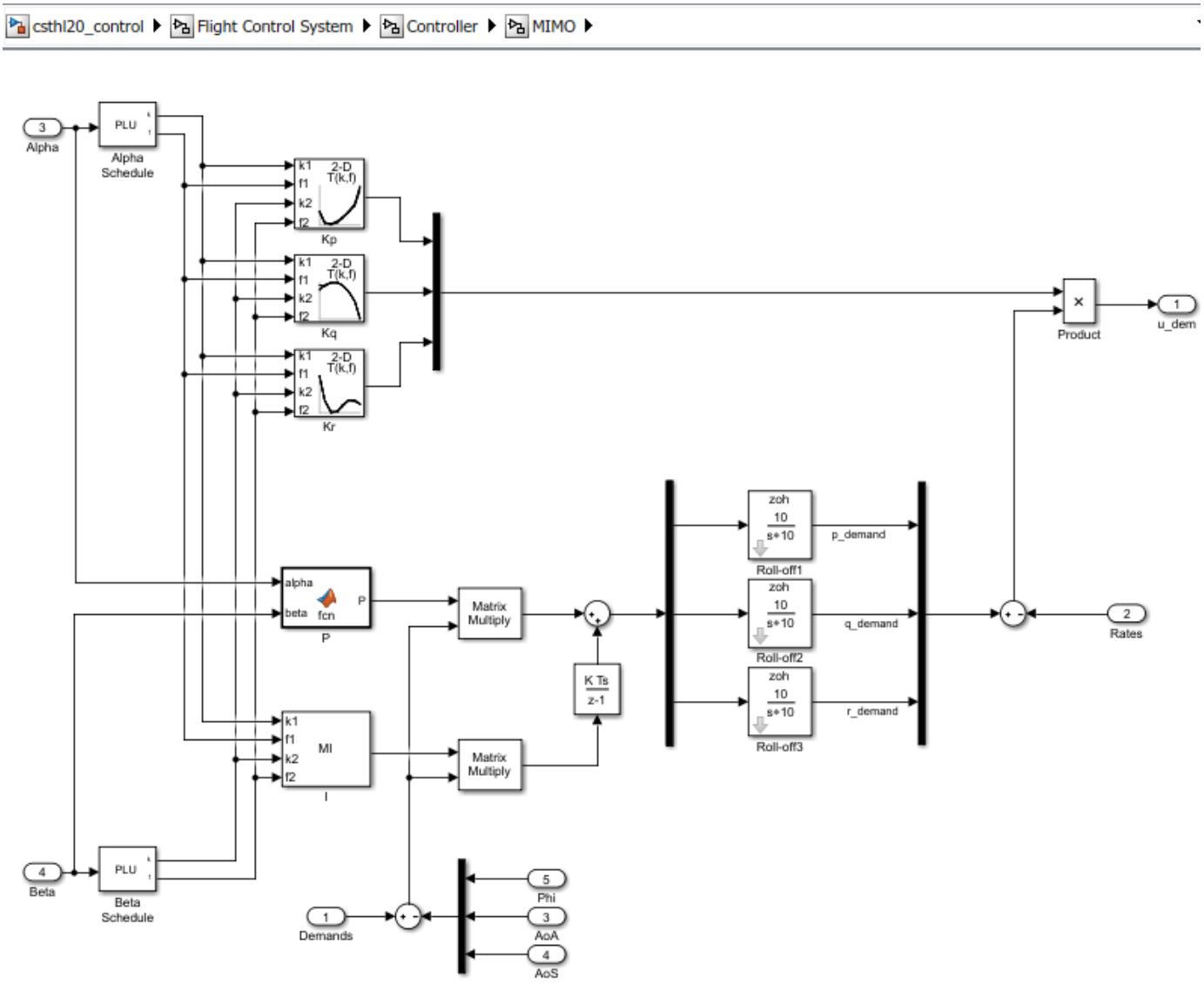
- “Attitude Control in the HL-20 Autopilot - MIMO Design”
- “Tune Gain Schedules in Simulink”

Attitude Control in the HL-20 Autopilot - MIMO Design

This is Part 4 of the example series on design and tuning of the flight control system for the HL-20 vehicle. This part shows how to tune a MIMO PI architecture for controlling the roll, pitch, and yaw of the vehicle.

Background

This example uses the HL-20 model adapted from “NASA HL-20 Lifting Body Airframe” (Aerospace Blockset), see Part 1 of the series (“Trimming and Linearization of the HL-20 Airframe”) for details. In Parts 2 and 3, we showed how to close the inner loops and tune the outer loops of a classic SISO architecture for the HL-20 autopilot, see “Angular Rate Control in the HL-20 Autopilot” and “Attitude Control in the HL-20 Autopilot - SISO Design” for details. In this example, we explore the benefits of switching to a MIMO architecture for the outer loops.



In this architecture, the three PI loops for pitch, alpha, beta are replaced by a 3-input, 3-output PI controller that blends the pitch, alpha, and beta measurements to calculate the inner-loop setpoints p_demand, q_demand, r_demand. Intuitively, this architecture should be more successful at reducing cross-couplings between axes. Note that the P and I gains are 3-by-3 matrices scheduled as a function of alpha and beta.

To get started, load the model, set `CTYPE` to 3 to select the MIMO variant of the Controller block, and reapply the steps of Part 2 for closing the inner loops (this part of the design is unchanged). Note that this creates and configures an `slTuner` interface `ST0` for interacting with the Simulink model.

```
load_system('csthl20_control')
CTYPE = 3;  % MIMO architecture
HL20recapPart2

ST0
```

```
slTuner tuning interface for "csthl20_control":

No tuned blocks. Use the addBlock command to add new blocks.
9 Analysis points:
--------------------------
Point 1: Signal "da;de;dr", located at 'Output Port 1' of csthl20_control/Flight Control System/C
Point 2: Signal "pqr", located at 'Output Port 2' of csthl20_control/HL20 Airframe
Point 3: 'Output Port 1' of csthl20_control/Flight Control System/Alpha_deg
Point 4: 'Output Port 1' of csthl20_control/Flight Control System/Beta_deg
Point 5: 'Output Port 1' of csthl20_control/Flight Control System/Phi_deg
Point 6: 'Output Port 1' of csthl20_control/Flight Control System/Controller/MIMO/Demands
Point 7: Signal "p_demand", located at 'Output Port 1' of csthl20_control/Flight Control System/C
Point 8: Signal "q_demand", located at 'Output Port 1' of csthl20_control/Flight Control System/C
Point 9: Signal "r_demand", located at 'Output Port 1' of csthl20_control/Flight Control System/C

No permanent openings. Use the addOpening command to add new permanent openings.
Properties with dot notation get/set access:
      Parameters         : []
      OperatingPoints    : [] (model initial condition will be used.)
      BlockSubstitutions : [3x1 struct]
      Options            : [1x1 linearize.SlTunerOptions]
      Ts                 : 0
```

Setup for Outer Loop Tuning

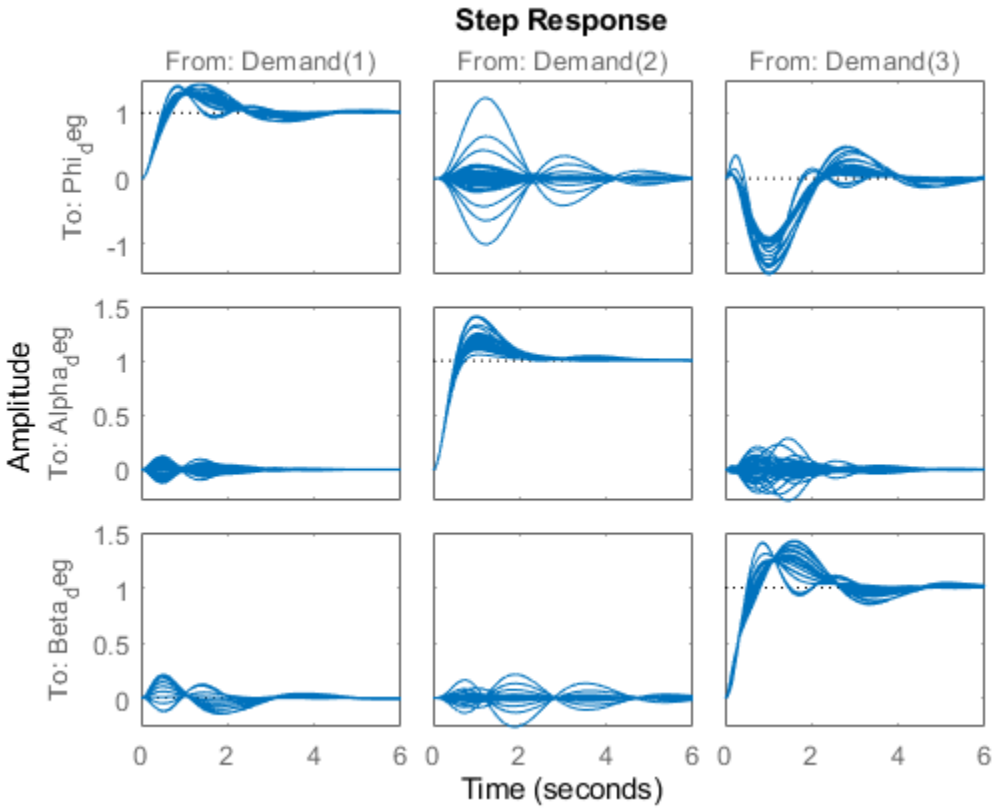
As in the SISO design (“Attitude Control in the HL-20 Autopilot - SISO Design”), the first step is to obtain a linearized model of the "plant" seen by the outer loops at each (alpha,beta) condition. To account for the fact that the inner-loop gains Kp,Kq,Kr vary with (alpha,beta), replace the "MIMO/Product" block by its linear equivalent, which is the diagonal gain matrix

$$\begin{pmatrix} Kp(\alpha,\beta) & 0 & 0 \\ 0 & Kq(\alpha,\beta) & 0 \\ 0 & 0 & Kr(\alpha,\beta) \end{pmatrix}.$$

```
Blk = 'csthl20_control/Flight Control System/Controller/MIMO/Product';
Subs = [zeros(3) append(ss(Kp),ss(Kq),ss(Kr))];
BlockSub4 = struct('Name',Blk,'Value',Subs);

ST0.BlockSubstitutions = [ST0.BlockSubstitutions ; BlockSub4];
```

The gain schedules "P" and "I" are initialized to the constant diagonal matrices `diag([0.05, 0.05, -0.05])`. Plot the angular responses for these initial settings.

```
T0 = getIOTransfer(ST0,'Demand',{'Phi_deg','Alpha_deg','Beta_deg'});
step(T0,6)
```

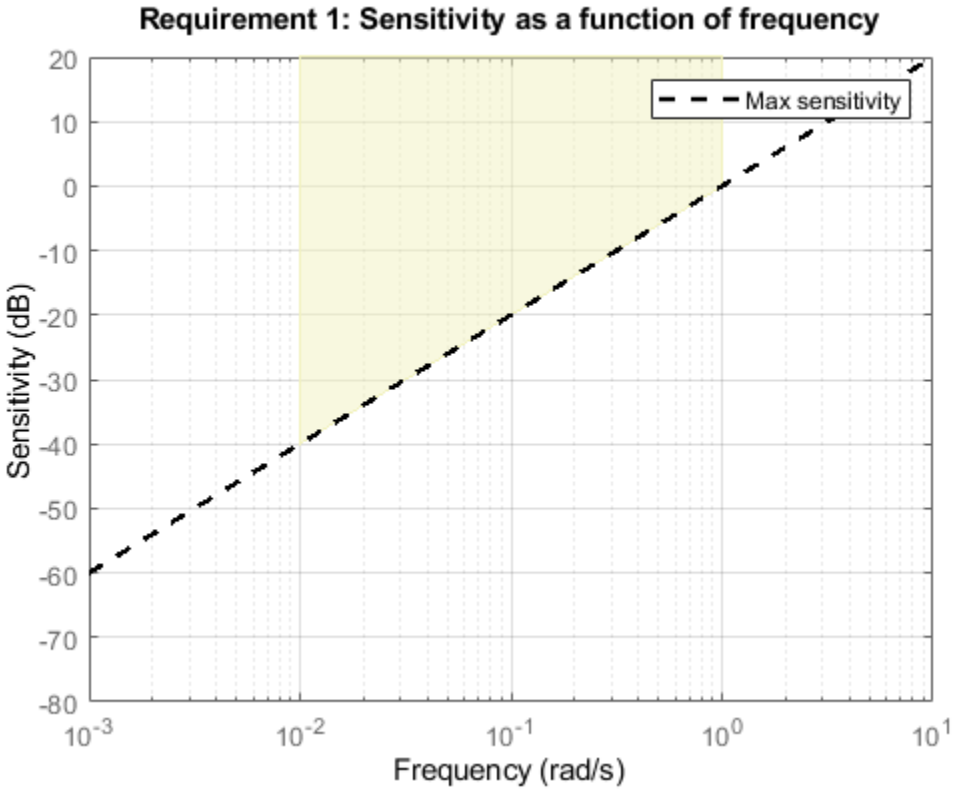


Tuning Goals

To tune the MIMO gain schedules we use the following three tuning goals:

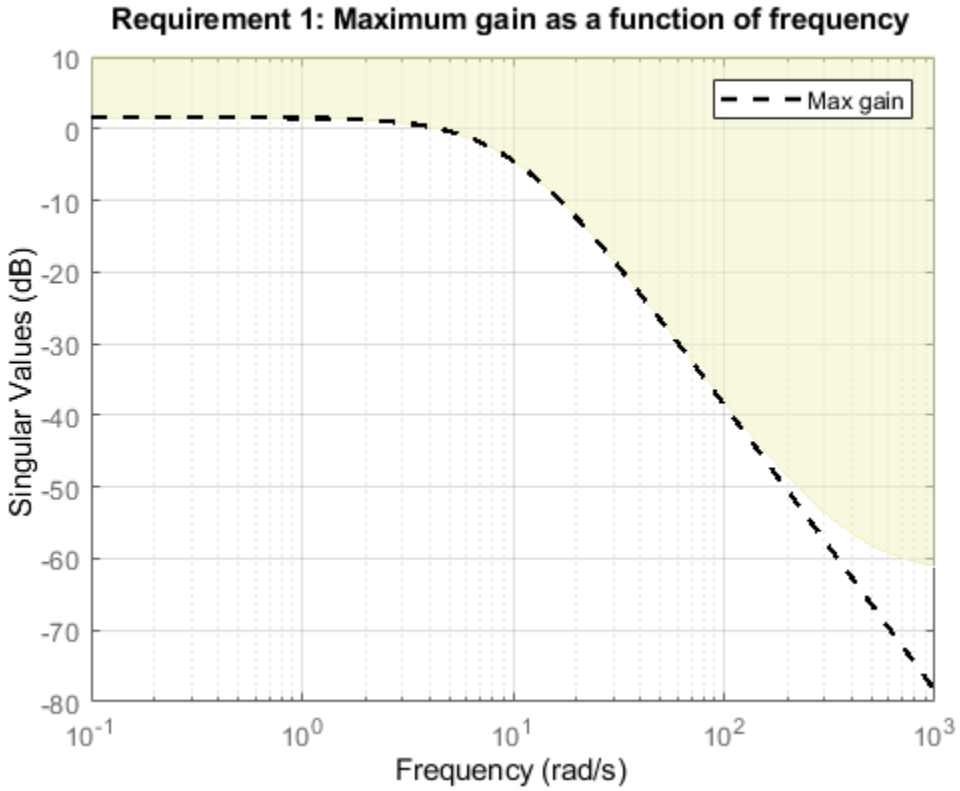
- A "Sensitivity" goal to specify the desired bandwidth (response time) and maximize decoupling at low frequency.

```
s = tf('s');
R1 = TuningGoal.Sensitivity({'Phi_deg','Alpha_deg','Beta_deg'},s);
R1.Focus = [1e-2 1];
R1.LoopScaling = 'off';

viewGoal(R1)
```



- A "Gain" constraint on the closed-loop transfer from angular demands to angular responses. The gain profile is chosen to enforce adequate roll-off and limit overshoot (which is related to the hump near crossover).

```
MaxGain = 1.2 * (10/(s+10))^2;  % max gain profile
R2 = TuningGoal.Gain('Demands',{'Phi_deg','Alpha_deg','Beta_deg'},MaxGain);

viewGoal(R2)
```



- A "Margins" goal to require gain margins of at least 7 dB and phase margins of at least 45 degrees (in the disk margin sense).

```
R3 = TuningGoal.Margins('da;de;dr',7,45);
```

Gain Schedule Tuning

The gain schedules for the MIMO PI controller are specified by the "P" and "I" blocks in the MIMO architecture. Recall that these blocks output 3-by-3 matrices and implement the MIMO transfer function:

$$\begin{pmatrix} p_{\text{demand}} \\ q_{\text{demand}} \\ r_{\text{demand}} \end{pmatrix} = \frac{10}{s+10}(P + I/s) \begin{pmatrix} \phi_{\text{deg}} \\ \alpha_{\text{deg}} \\ \beta_{\text{deg}} \end{pmatrix}.$$

For illustration sake, we use a MATLAB Function block to implement the proportional gain schedule, and a Matrix Interpolation block to implement the integral gain schedule. The Matrix Interpolation block lives in the "Simulink Extras" library and is a lookup table where each table entry is a matrix.

To tune the P and I gain schedules, mark the corresponding blocks as tunable in the `slTuner` interface.

```
TunedBlocks = {'MIMO/P' , 'MIMO/I'};
ST0.addBlock(TunedBlocks)
```

Parameterize each tuned gain schedule as a polynomial surface in alpha and beta. Again we use a quadratic surface for the proportional gain and a multilinear surface for the integral gain.

```
% Grid of (alpha,beta) design points
alpha_vec = -10:5:25;     % Alpha Range
beta_vec = -10:5:10;     % Beta Range
[alpha,beta] = ndgrid(alpha_vec,beta_vec);
SG = struct('alpha',alpha,'beta',beta);

% Proportional gain matrix
alphabetaBasis = polyBasis('canonical',2,2);
P0 = diag([0.05 0.05 -0.05]);  % initial (constant) value
PS = tunableSurface('P', P0, SG, alphabetaBasis);
ST0.setBlockParam('P',PS);

% Integral gain matrix
alphaBasis = @(alpha) alpha;
betaBasis = @(beta) abs(beta);
alphabetaBasis = ndBasis(alphaBasis,betaBasis);
I0 = diag([0.05 0.05 -0.05]);
IS = tunableSurface('I', I0, SG, alphabetaBasis);
ST0.setBlockParam('I',IS);
```

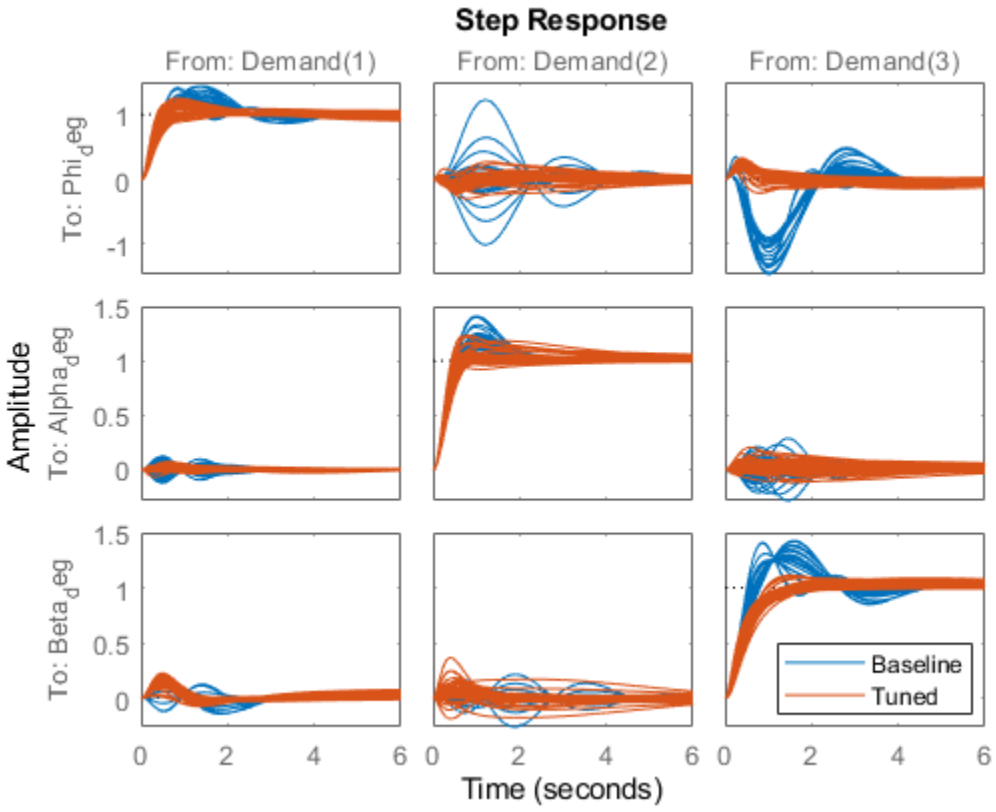
Finally, use `systune` to tune the 6 gain surfaces against the three tuning goals.

```
ST = systune(ST0,[R1 R2 R3]);
```

```
Final: Soft = 1.13, Hard = -Inf, Iterations = 104
```

The final value of the objective function indicates that the tuning goals are nearly met (a tuning goal is satisfied when its "value" is less than one). Plot the closed-loop angular responses and compare with the baseline design.

```
T = getIOTransfer(ST,'Demand',{'Phi_deg','Alpha_deg','Beta_deg'});
step(T0,T,6)
legend('Baseline','Tuned','Location','SouthEast')
```



These responses show significant reductions in overshoot and cross-coupling when compared to the SISO design.

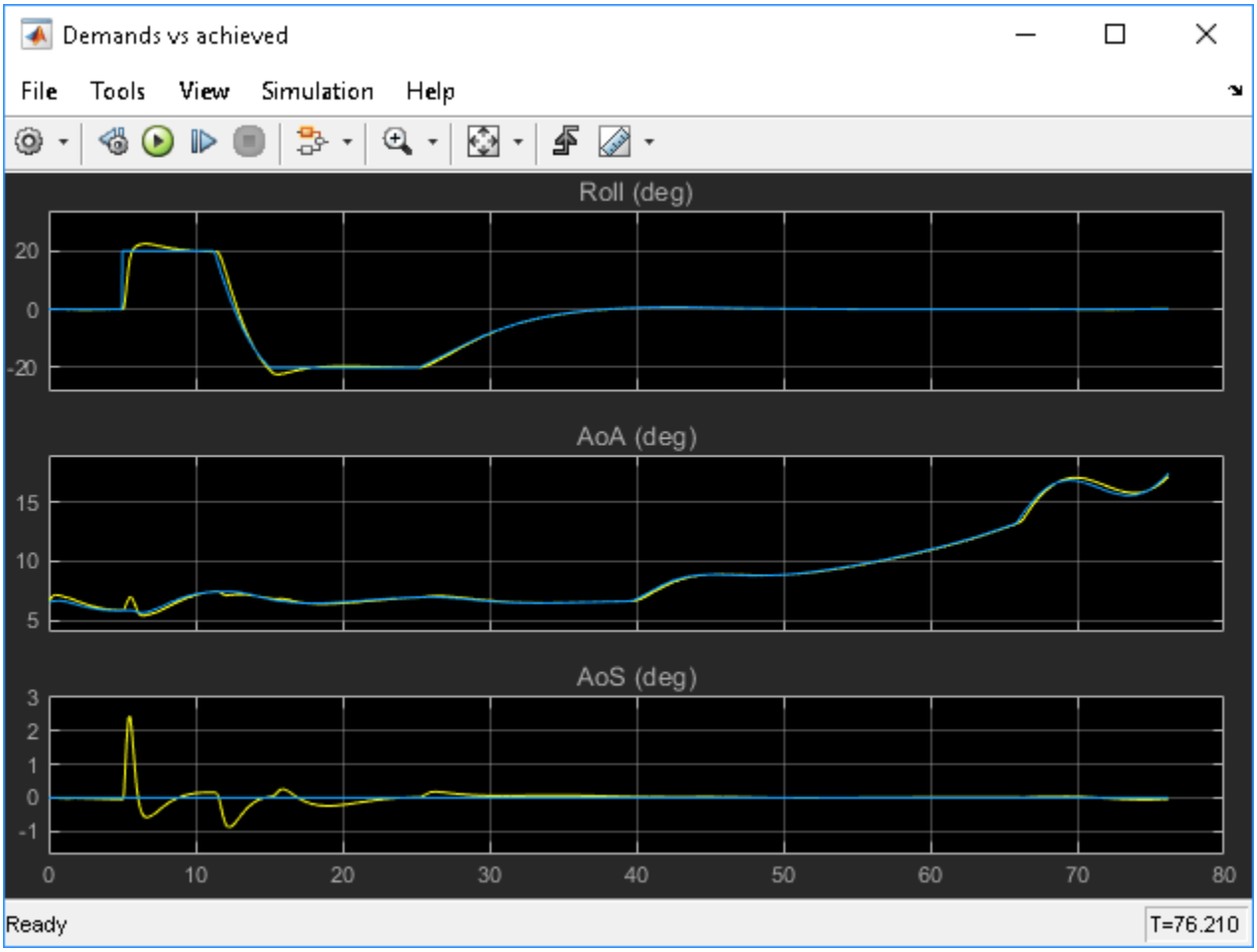
Validation

To further validate this design, push the tuned gain surfaces to the Simulink model.

```
writeBlockValue(ST)
```

For the Matrix Interpolation block "I", this samples the gain surface at the table breakpoints and updates the table data in the model workspace. For the MATLAB Function block "P", this generates MATLAB code for the gain surface equations. You can see this code by double-clicking on the block.

Once you push the gains to Simulink, tuning of the MIMO architecture is complete and you can simulate its behavior during the landing approach.



These responses are not very different from the SISO design (“Attitude Control in the HL-20 Autopilot - SISO Design”) due to the mild demands throughout the maneuver. The benefits of the MIMO design would be more visible in a more challenging maneuver.

See Also

`tunableSurface`

More About

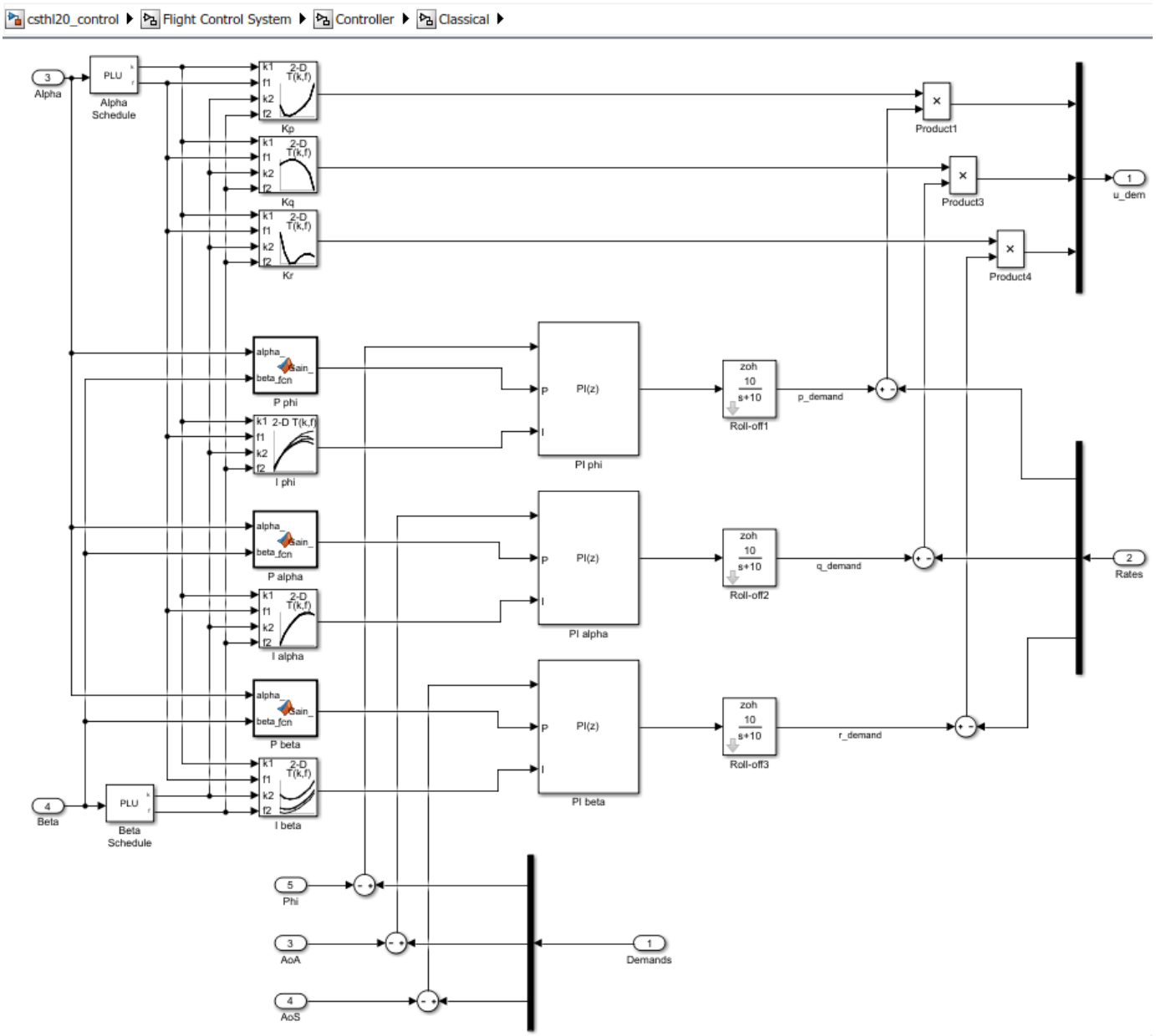
- “Attitude Control in the HL-20 Autopilot - SISO Design”
- “Tune Gain Schedules in Simulink”

MATLAB Workflow for Tuning the HL-20 Autopilot

This is Part 5 of the example series on design and tuning of the flight control system for the HL-20 vehicle. This part shows how to perform most of the design in MATLAB® without interacting with the Simulink® model.

Background

This example uses the HL-20 model adapted from “NASA HL-20 Lifting Body Airframe” (Aerospace Blockset), see Part 1 of the series (“Trimming and Linearization of the HL-20 Airframe”) for details. The autopilot controlling the attitude of the aircraft consists of three inner loops and three outer loops.



In Part 2 (“Angular Rate Control in the HL-20 Autopilot”) and Part 3 (“Attitude Control in the HL-20 Autopilot - SISO Design”), we showed how to close the inner loops and tune the gain schedules for the outer loops. These examples made use of the `slTuner` interface to interact with the Simulink model, obtain linearized models and control system responses, and push tuned values back to Simulink.

For simple architectures and rapid design iterations, it can be preferable (and conceptually simpler) to manipulate the linearized models in MATLAB and use basic commands like `feedback` to close loops. This example shows how to perform the design steps of Parts 2 and 3 in MATLAB.

Obtaining the Plant Models

To tune the autopilot, we need linearized models of the transfer function from deflections to angular position and rates. To do this, start from the results from the "Trim and Linearize" step (see “Trimming and Linearization of the HL-20 Airframe”). Recall that `G7` is a seven-state linear model of the airframe at 40 different (alpha,beta) conditions, and `CS` is the linearization of the Controls Selector block.

```
load csthl20_TrimData G7 CS
```

Using the Simulink model "csthl20_trim" as reference for selecting I/Os, build the desired plant models by connecting `G7` and `CS` in series. Do not forget to convert phi,alpha,beta from radians to degrees.

```
r2d = 180/pi;
G = diag([1 1 1 r2d r2d r2d]) * G7([4:7 31:32],1:6) * CS(:,1:3);

G.InputName = {'da','de','dr'};
G.OutputName = {'p','q','r','Phi_deg','Alpha_deg','Beta_deg'};

size(G)
```

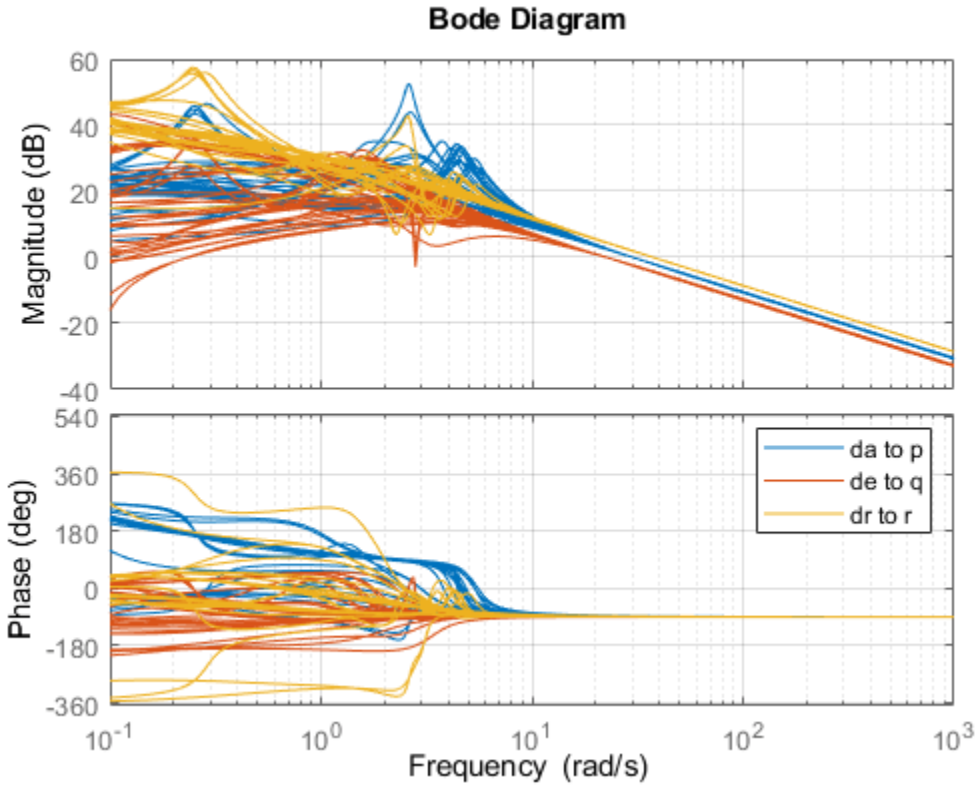
```
8x5 array of state-space models.
Each model has 6 outputs, 3 inputs, and 7 states.
```

This gives us an array of plant models over the 8-by-5 grid of (alpha,beta) conditions used for trimming.

Closing the Inner Loops

To close the inner loops, we follow the same procedure as in Part 2 (“Angular Rate Control in the HL-20 Autopilot”). This consists of selecting the gain Kp,Kq,Kr to set the crossover frequency of the p,q,r loops to 30, 22.5, and 37.5 rad/s, respectively.

```
% Compute Kp,Kq,Kr for each (alpha,beta) condition.
Gpqr = G({'p','q','r'},:);
Kp = 1./abs(evalfr(Gpqr(1,1),30i));
Kq = -1./abs(evalfr(Gpqr(2,2),22.5i));
Kr = -1./abs(evalfr(Gpqr(3,3),37.5i));

bode(Gpqr(1,1)*Kp,Gpqr(2,2)*Kq,Gpqr(3,3)*Kr,{1e-1,1e3}), grid
legend('da to p','de to q','dr to r')
```



Use `feedback` to close the three inner loops. Insert an analysis point at the plant inputs da,de,dr for later evaluation of the stability margins.

```
Cpqr = append(ss(Kp),ss(Kq),ss(Kr));
APu = AnalysisPoint('u',3);  APu.Location = {'da','de','dr'};

Gpos = feedback(G * APu * Cpqr, eye(3), 1:3, 1:3);
Gpos.InputName = {'p_demand','q_demand','r_demand'};

size(Gpos)
```

```
8x5 array of generalized state-space models.
Each model has 6 outputs, 3 inputs, 7 states, and 1 blocks.
```

Note that these commands seamlessly manage the fact that we are dealing with arrays of plants and gains corresponding to the various (alpha,beta) conditions.

Tuning the Outer Loops

Next move to the outer loops. We already have an array of linear models `Gpos` for the "plant" seen by the outer loops. As done in Part 3 (“Attitude Control in the HL-20 Autopilot - SISO Design”), parameterize the six gain schedules as polynomial surfaces in alpha and beta. Again we use quadratic surfaces for the proportional gains and multilinear surfaces for the integral gains.

```
% Grid of (alpha,beta) design points
alpha_vec = -10:5:25;     % Alpha Range
beta_vec = -10:5:10;     % Beta Range
```

```
[alpha,beta] = ndgrid(alpha_vec,beta_vec);
SG = struct('alpha',alpha,'beta',beta);

% Proportional gains
alphabetaBasis = polyBasis('canonical',2,2);
P_PHI = tunableSurface('Pphi', 0.05, SG, alphabetaBasis);
P_ALPHA = tunableSurface('Palpha', 0.05, SG, alphabetaBasis);
P_BETA = tunableSurface('Pbeta', -0.05, SG, alphabetaBasis);

% Integral gains
alphaBasis = @(alpha) alpha;
betaBasis = @(beta) abs(beta);
alphabetaBasis = ndBasis(alphaBasis,betaBasis);
I_PHI = tunableSurface('Iphi', 0.05, SG, alphabetaBasis);
I_ALPHA = tunableSurface('Ialpha', 0.05, SG, alphabetaBasis);
I_BETA = tunableSurface('Ibeta', -0.05, SG, alphabetaBasis);
```

The overall controller for the outer loop is a diagonal 3-by-3 PI controller taken the errors on angular positions phi,alpha,beta and calculating the rate demands p_demand,q_demand,r_demand.

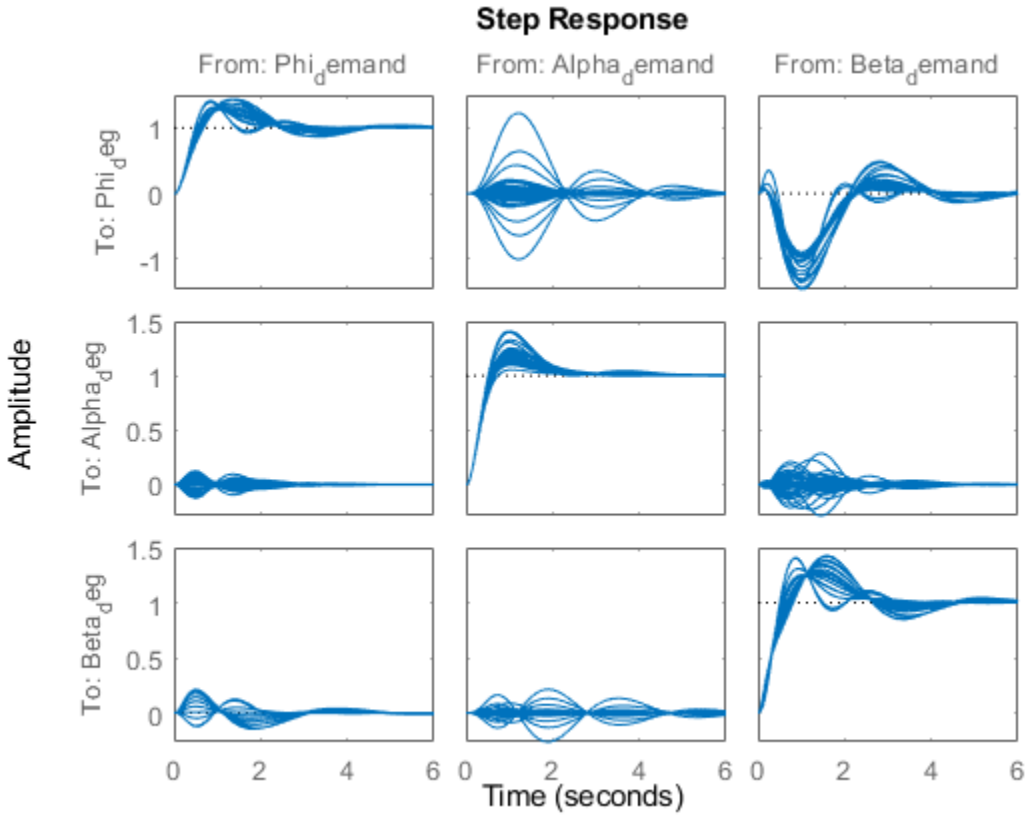
```
KP = append(P_PHI,P_ALPHA,P_BETA);
KI = append(I_PHI,I_ALPHA,I_BETA);
Cpos = KP + KI * tf(1,[1 0]);
```

Finally, use `feedback` to obtain a tunable closed-loop model of the outer loops. To enable tuning and closed-loop analysis, insert analysis points at the plant outputs.

```
RollOffFilter = tf(10,[1 10]);
APy = AnalysisPoint('y',3);  APy.Location = {'Phi_deg','Alpha_deg','Beta_deg'};

T0 = feedback(APy * Gpos(4:6,:) * RollOffFilter * Cpos ,eye(3));
T0.InputName = {'Phi_demand','Alpha_demand','Beta_demand'};
T0.OutputName = {'Phi_deg','Alpha_deg','Beta_deg'};
```

You can plot the closed-loop responses for the initial gain surface settings (constant gains of 0.05).

```
step(T0,6)
```



Tuning Goals

Use the same tuning goals as in Part 3 (“Attitude Control in the HL-20 Autopilot - SISO Design”). These include "MinLoopGain" and "MaxLoopGain" goals to set the gain crossover of the outer loops between 0.5 and 5 rad/s.

```
R1 = TuningGoal.MinLoopGain({'Phi_deg','Alpha_deg','Beta_deg'},0.5,1);
R1.LoopScaling = 'off';
R2 = TuningGoal.MaxLoopGain({'Phi_deg','Alpha_deg','Beta_deg'},tf(50,[1 10 0]));
R2.LoopScaling = 'off';
```

These also include a varying "Margins" goal to impose adequate stability margins in each loop and across loops.

```
% Gain margins vs (alpha,beta)
GM = [...
   6     6     6     6     6
   6     6     7     6     6
   7     7     7     7     7
   7     7     7     7     7
   7     7     7     7     7
   7     7     7     7     7
   6     6     7     6     6
   6     6     6     6     6];

% Phase margins vs (alpha,beta)
PM = [...
```

```
    40          40          40          40          40
    40          40          45          40          40
    45          45          45          45          45
    45          45          45          45          45
    45          45          45          45          45
    45          45          45          45          45
    40          40          45          40          40
    40          40          40          40          40];

% Create varying goal
FH = @(gm,pm) TuningGoal.Margins({'da','de','dr'},gm,pm);
R3 = varyingGoal(FH,GM,PM);
```

Gain Schedule Tuning

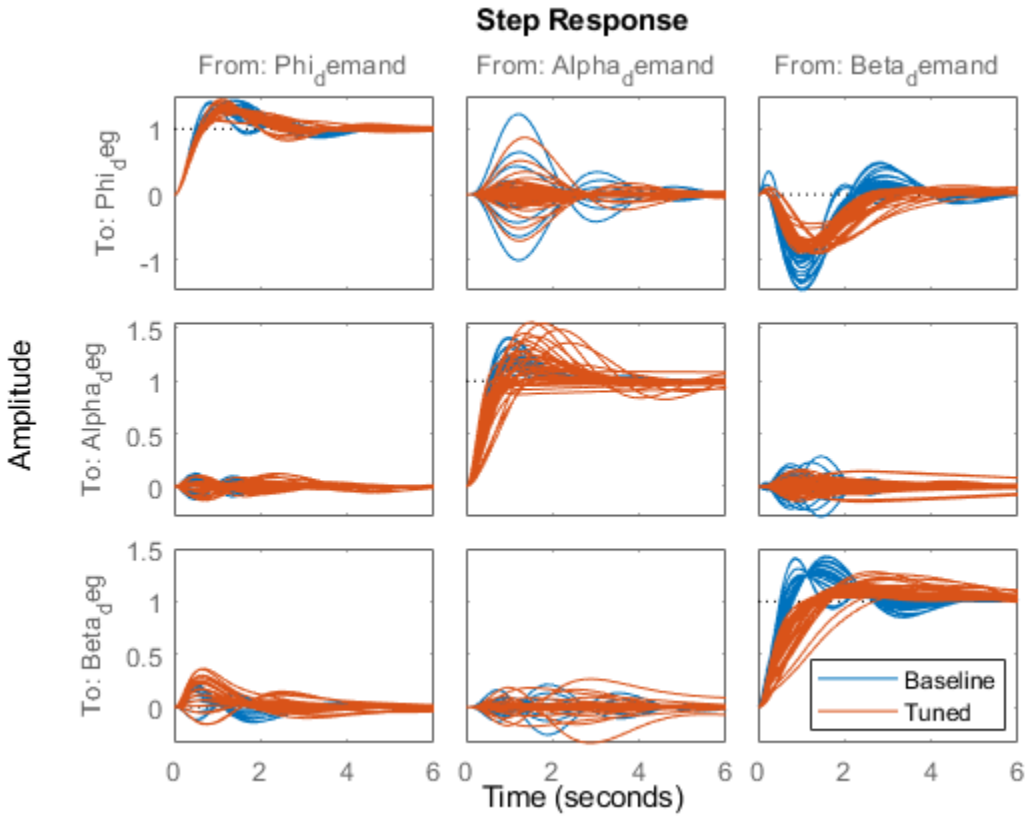
You can now use `systune` to shape the six gain surfaces against the tuning goals at all 40 design points.

```
T = systune(T0,[R1 R2 R3]);
```

```
Final: Soft = 1.03, Hard = -Inf, Iterations = 53
```

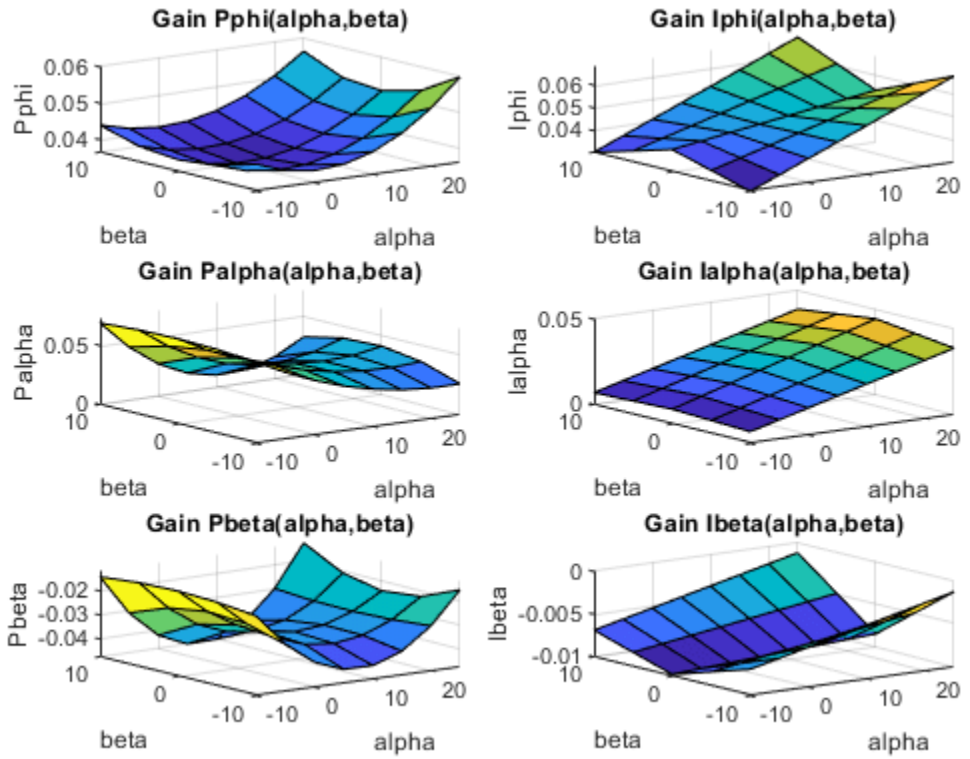
The final objective value is close to 1 so the tuning goals are essentially met. Plot the closed-loop angular responses and compare with the initial settings.

```
step(T0,T,6)
legend('Baseline','Tuned','Location','SouthEast')
```



The results match those obtained in Parts 2 and 3. The tuned gain surfaces are also similar.

```
clf
% NOTE: setBlockValue updates each gain surface with the tuned coefficients in T
subplot(3,2,1), viewSurf(setBlockValue(P_PHI,T))
subplot(3,2,3), viewSurf(setBlockValue(P_ALPHA,T))
subplot(3,2,5), viewSurf(setBlockValue(P_BETA,T))
subplot(3,2,2), viewSurf(setBlockValue(I_PHI,T))
subplot(3,2,4), viewSurf(setBlockValue(I_ALPHA,T))
subplot(3,2,6), viewSurf(setBlockValue(I_BETA,T))
```



You could now use `evalSurf` to sample the gain surfaces and update the lookup tables in the Simulink model. You could also use the `codegen` method to generate code for the gain surface equations. For example

```
% Generate code for "P phi" block
MCODE = codegen(setBlockValue(P_PHI,T));

% Get tuned values for the "I phi" lookup table
Kphi = evalSurf(setBlockValue(I_PHI,T),alpha_vec,beta_vec);
```

See Also

`tunableSurface`

More About

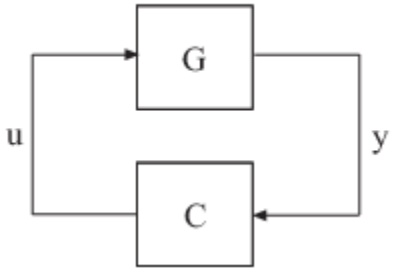
- “Trimming and Linearization of the HL-20 Airframe”
- “Tune Gain Schedules in Simulink”

Loop-Shaping Design

- “Structure of Control System for Tuning With looptune” on page 12-2
- “Set Up Your Control System for Tuning with looptune” on page 12-3
- “Tune MIMO Control System for Specified Bandwidth” on page 12-4
- “Decoupling Controller for a Distillation Column” on page 12-10
- “Tuning of a Digital Motion Control System” on page 12-21

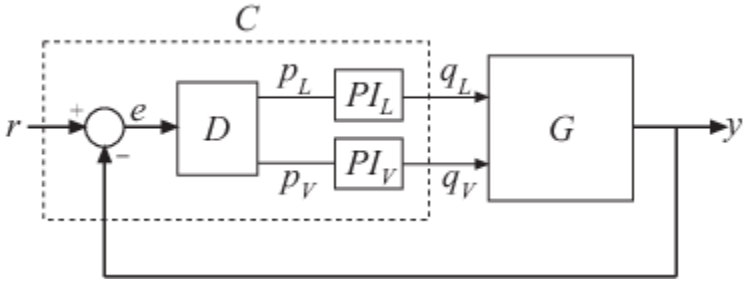
Structure of Control System for Tuning With looptune

`looptune` tunes the feedback loop illustrated below to meet default requirements or requirements that you specify.



`C` represents the controller and `G` represents the plant. The sensor outputs `y` (measurement signals) and actuator outputs `u` (control signals) define the boundary between plant and controller. The controller is the portion of your control system whose inputs are measurements, and whose outputs are control signals. Conversely, the plant is the remainder—the portion of your control system that receives control signals as inputs, and produces measurements as outputs.

For example, in the control system of the following illustration, the controller `C` receives the measurement `y`, and the reference signal `r`. The controller produces the controls `qL` and `qV` as outputs.



The controller `C` has a fixed internal structure. `C` includes a gain matrix `D` , the PI controllers `PI_L` and `PI_V`, and a summing junction. The `looptune` command tunes free parameters of `C` such as the gains in `D` and the proportional and integral gains of `PI_L` and `PI_V`. You can also use `looptune` to co-tune free parameters in both `C` and `G`.

Set Up Your Control System for Tuning with looptune

Set Up Your Control System for looptunein MATLAB

To set up your control system in MATLAB for tuning with `looptune`:

1. Parameterize the tunable elements of your controller. You can use predefined structures such as `tunablePID`, `tunableGain`, and `tunableTF`. Or, you can create your own structure from elementary tunable parameters (`realp`).
2. Use model interconnection commands such as `series` and `connect` to build a tunable `genss` model representing the controller `C0`.
3. Create a Numeric LTI model representing the plant `G`. For co-tuning the plant and controller, represent the plant as a tunable `genss` model.

Set Up Your Control System for looptune in Simulink

To set up your control system in Simulink for tuning with `systune` (requires Simulink Control Design software):

1. Use `slTuner` to create an interface to the Simulink model of your control system. When you create the interface, you specify which blocks to tune in your model.
2. Use `addPoint` to specify the control and measurement signals that define the boundaries between plant and controller. Use `addOpening` to mark optional loop-opening or signal injection sites for specifying and assessing open-loop requirements.

The `slTuner` interface automatically linearizes your Simulink model. The `slTuner` interface also automatically parametrizes the blocks that you specify as tunable blocks. For more information about this linearization, see the `slTuner` reference page and “How Tuned Simulink Blocks Are Parameterized” on page 10-27.

See Also

Related Examples

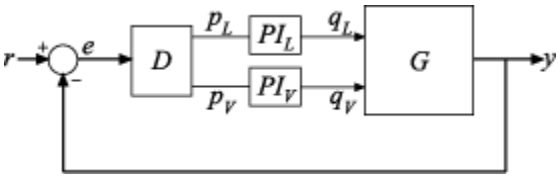
- “Tune MIMO Control System for Specified Bandwidth” on page 12-4
- “Tune Feedback Loops using looptune”

More About

- “Structure of Control System for Tuning With looptune” on page 12-2

Tune MIMO Control System for Specified Bandwidth

This example shows how to tune the following control system to achieve a loop crossover frequency between 0.1 and 1 rad/s using `looptune`.



The plant, `G`, is a two-input, two-output model (*y* is a two-element vector signal). For this example, the transfer function of `G` is given by:

$$G(s) = \frac{1}{75s+1}\begin{bmatrix} 87.8 & -86.4 \\ 108.2 & -109.6 \end{bmatrix}.$$

This sample plant is based on the distillation column described in more detail in the example "Decoupling Controller for a Distillation Column".

To tune this control system, you first create a numeric model of the plant. Then you create tunable models of the controller elements and interconnect them to build a controller model. Then you use `looptune` to tune the free parameters of the controller model. Finally, examine the performance of the tuned system to confirm that the tuned controller yields desirable performance.

Create a model of the plant.

```
s = tf('s');
G = 1/(75*s+1)*[87.8 -86.4; 108.2 -109.6];
G.InputName = {'qL','qV'};
G.OutputName = 'y';
```

When you tune the control system, `looptune` uses the channel names `G.InputName` and `G.OutputName` to interconnect the plant and controller. Therefore, assign these channel names to match the illustration. When you set `G.OutputName = 'y'`, the `G.OutputName` is automatically expanded to `{'y(1)';'y(2)'}`. This expansion occurs because `G` is a two-output system.

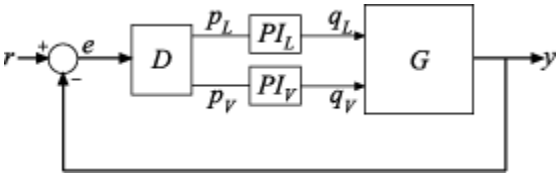
Represent the components of the controller.

```
D = tunableGain('Decoupler',eye(2));
D.InputName = 'e';
D.OutputName = {'pL','pV'};

PI_L = tunablePID('PI_L','pi');
PI_L.InputName = 'pL';
PI_L.OutputName = 'qL';

PI_V = tunablePID('PI_V','pi');
PI_V.InputName = 'pV';
PI_V.OutputName = 'qV';

sum1 = sumblk('e = r - y',2);
```

The control system includes several tunable control elements. `PI_L` and `PI_V` are tunable PI controllers. These elements represented by `tunablePID` models. The fixed control structure also

includes a decoupling gain matrix `D`, represented by a tunable `tunableGain` model. When the control system is tuned, `D` ensures that each output of `G` tracks the corresponding reference signal `r` with minimal crosstalk.

Assigning `InputName` and `OutputName` values to these control elements allows you to interconnect them to create a tunable model of the entire controller `C` as shown.



When you tune the control system, `looptune` uses these channel names to interconnect `C` and `G`. The controller `C` also includes the summing junction `sum1`. This a two-channel summing junction, because `r` and `y` are vector-valued signals of dimension 2.

Connect the controller components.

```
C0 = connect(PI_L,PI_V,D,sum1,{'r','y'},{'qL','qV'});
```

`C0` is a tunable `genss` model that represents the entire controller structure. `C0` stores the tunable controller parameters and contains the initial values of those parameters.

Tune the control system.

The inputs to `looptune` are `G` and `C0`, the plant and initial controller models that you created. The input `wc = [0.1,1]` sets the target range for the loop bandwidth. This input specifies that the crossover frequency of each loop in the tuned system fall between 0.1 and 1 rad/min.

```
wc = [0.1,1];
[G,C,gam,Info] = looptune(G,C0,wc);
```

```
Final: Peak gain = 0.991, Iterations = 26
Achieved target gain value TargetGain=1.
```

The displayed `Peak Gain = 0.949` indicates that `looptune` has found parameter values that achieve the target loop bandwidth. `looptune` displays the final peak gain value of the optimization run, which is also the output `gam`. If `gam` is less than 1, all tuning requirements are satisfied. A value greater than 1 indicates failure to meet some requirement. If `gam` exceeds 1, you can increase the target bandwidth range or relax another tuning requirement.

`looptune` also returns the tuned controller model `C`. This model is the tuned version of `C0`. It contains the PI coefficients and the decoupling matrix gain values that yield the optimized peak gain value.

Display the tuned controller parameters.

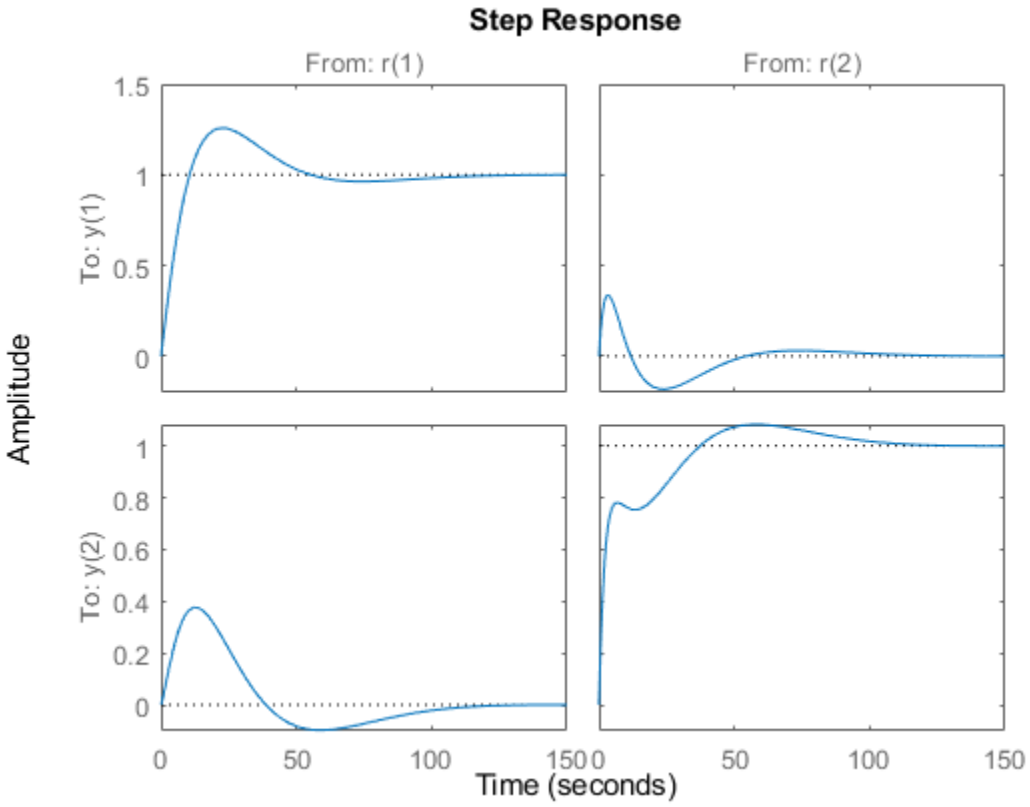
```
showTunable(C)
```

```
Decoupler =

  D =
           u1       u2
   y1    1.229  -0.8621
   y2   -1.517    1.239
```

```
Name: Decoupler
Static gain.
-----------------------------------
PI_L =

            1
  Kp + Ki * ---
            s

  with Kp = 2.22, Ki = 0.125

Name: PI_L
Continuous-time PI controller in parallel form.
-----------------------------------
PI_V =

            1
  Kp + Ki * ---
            s

  with Kp = -1.76, Ki = -0.0843

Name: PI_V
Continuous-time PI controller in parallel form.
```

Check the time-domain response for the control system with the tuned coefficients. To produce a plot, construct a closed-loop model of the tuned control system. Plot the step response from reference to output.

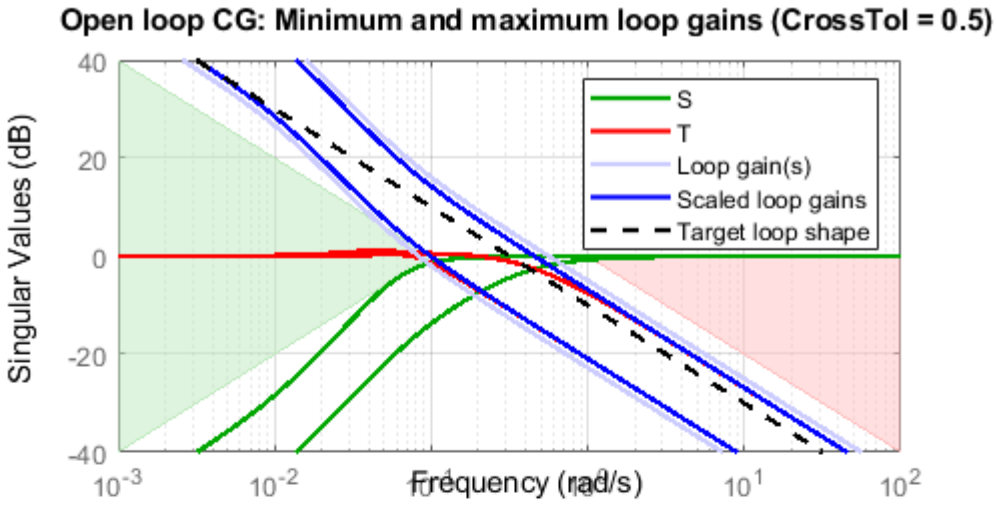
```
T = connect(G,C,'r','y');
step(T)
```

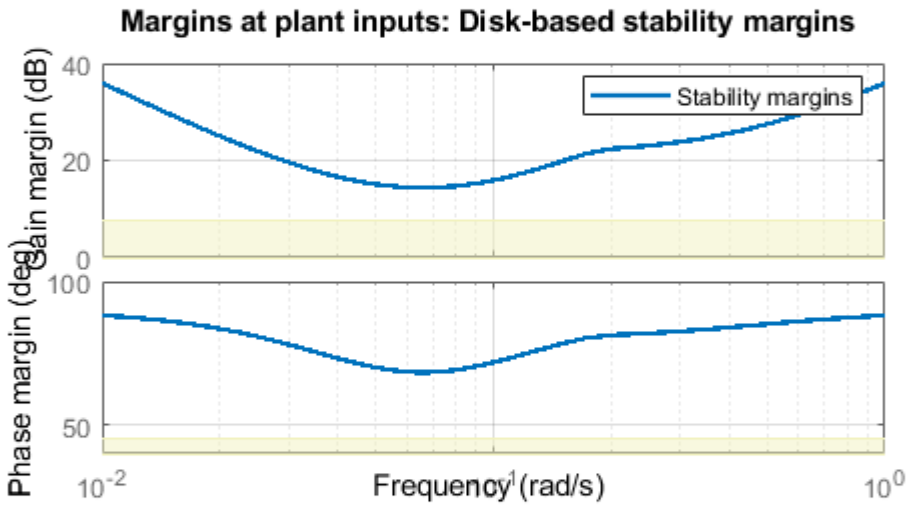


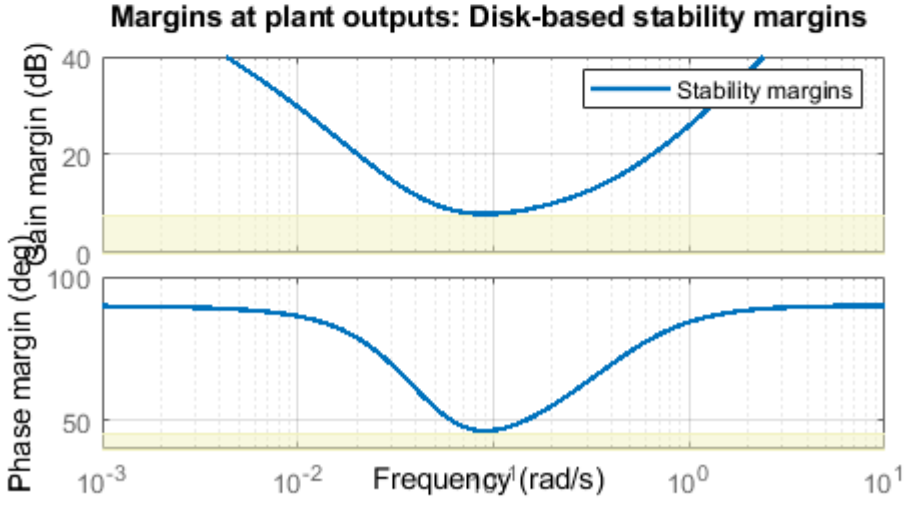
The decoupling matrix in the controller permits each channel of the two-channel output signal `y` to track the corresponding channel of the reference signal `r`, with minimal crosstalk. From the plot, you can how well this requirement is achieved when you tune the control system for bandwidth alone. If the crosstalk still exceeds your design requirements, you can use a `TuningGoal.Gain` requirement object to impose further restrictions on tuning.

Examine the frequency-domain response of the tuned result as an alternative method for validating the tuned controller.

```
figure('Position',[100,100,520,1000])
loopview(G,C,Info)
```


Open loop CG: Minimum and maximum loop gains (CrossTol = 0.5)
Singular Values (dB)
Frequency (rad/s)
S
T
Loop gain(s)
Scaled loop gains
Target loop shape


Margins at plant inputs: Disk-based stability margins
Gain margin (dB)
Phase margin (deg)
Frequency (rad/s)
Stability margins


Margins at plant outputs: Disk-based stability margins
Gain margin (dB)
Phase margin (deg)
Frequency (rad/s)
Stability margins

The first plot shows that the open-loop gain crossovers fall within the specified interval `[0.1,1]`. This plot also includes the maximum and tuned values of the sensitivity function $S = (I - GC)^{-1}$ and complementary sensitivity $T = I - S$. The second and third plots show that the MIMO stability margins of the tuned system (blue curve) do not exceed the upper limit (yellow curve).

See Also

Related Examples

- “Decoupling Controller for a Distillation Column”

More About

- “Structure of Control System for Tuning With looptune” on page 12-2

Decoupling Controller for a Distillation Column

This example shows how to use `looptune` to decouple the two main feedback loops in a distillation column.

Distillation Column Model

This example uses a simple model of the distillation column shown below.

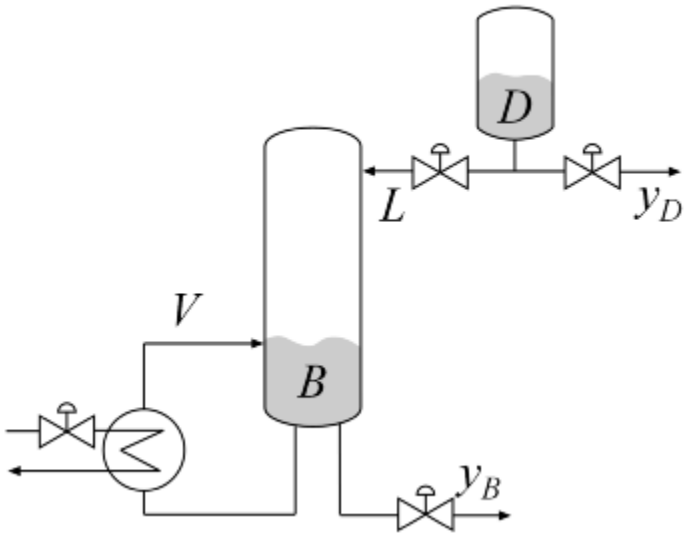


Figure 1: Distillation Column

In the so-called LV configuration, the controlled variables are the concentrations `yD` and `yB` of the chemicals `D` (tops) and `B` (bottoms), and the manipulated variables are the reflux `L` and boilup `V`. This process exhibits strong coupling and large variations in steady-state gain for some combinations of L and V. For more details, see Skogestad and Postlethwaite, *Multivariable Feedback Control*.

The plant is modeled as a first-order transfer function with inputs `L,V` and outputs `yD,yB`:

$$G(s) = \frac{1}{75s+1}\begin{pmatrix} 87.8 & -86.4 \\ 108.2 & -109.6 \end{pmatrix}$$

The unit of time is minutes (all plots are in minutes, not seconds).

```
s = tf('s');
G = [87.8 -86.4 ; 108.2 -109.6]/(75*s+1);
G.InputName = {'L','V'};
G.OutputName = {'yD','yB'};
```

Control Architecture

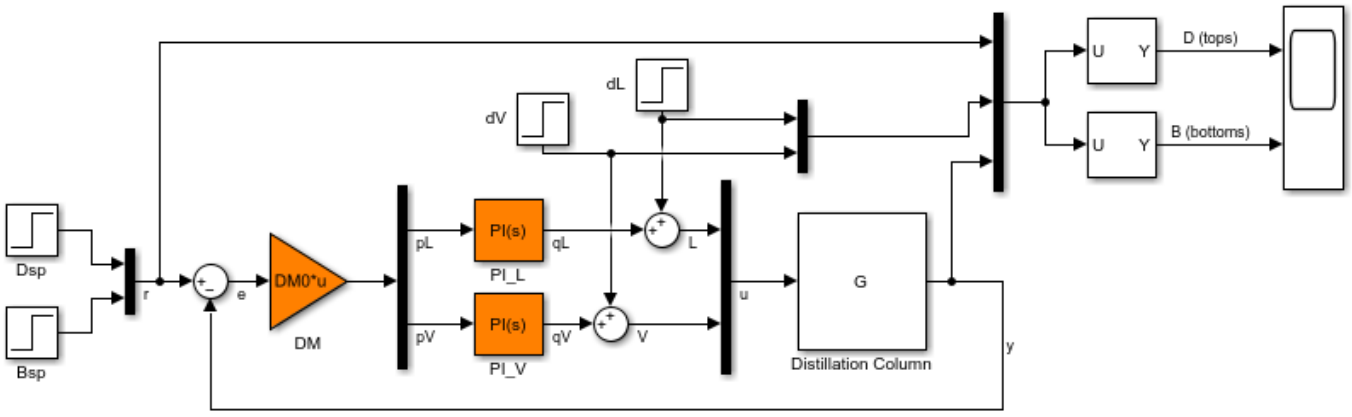
The control objectives are as follows:

- Independent control of the tops and bottoms concentrations by ensuring that a change in the tops setpoint `Dsp` has little impact on the bottoms concentration `B` and vice versa

- Response time of about 4 minutes with less than 15% overshoot
- Fast rejection of input disturbances affecting the effective reflux `L` and boilup `V`

To achieve these objectives we use the control architecture shown below. This architecture consists of a static decoupling matrix `DM` in series with two PI controllers for the reflux `L` and boilup `V`.

```
open_system('rct_distillation')
```



Decoupling controller for a distillation column

Controller Tuning in Simulink with LOOPTUNE

The `looptune` command provides a quick way to tune MIMO feedback loops. When the control system is modeled in Simulink, you just specify the tuned blocks, the control and measurement signals, and the desired bandwidth, and `looptune` automatically sets up the problem and tunes the controller parameters. `looptune` shapes the open-loop response to provide integral action, roll-off, and adequate MIMO stability margins.

Use the `slTuner` interface to specify the tuned blocks, the controller I/Os, and signals of interest for closed-loop validation.

```
ST0 = slTuner('rct_distillation',{'PI_L','PI_V','DM'});

% Signals of interest
addPoint(ST0,{'r','dL','dV','L','V','y'})
```

Set the control bandwidth by specifying the gain crossover frequency for the open-loop response. For a response time of 4 minutes, the crossover frequency should be approximately 2/4 = 0.5 rad/min.

```
wc = 0.5;
```

Use `TuningGoal` objects to specify the remaining control objectives. The response to a step command should have less than 15% overshoot. The response to a step disturbance at the plant input should be well damped, settle in less than 20 minutes, and not exceed 4 in amplitude.

```
OS = TuningGoal.Overshoot('r','y',15);

DR = TuningGoal.StepRejection({'dL','dV'},'y',4,20);
```

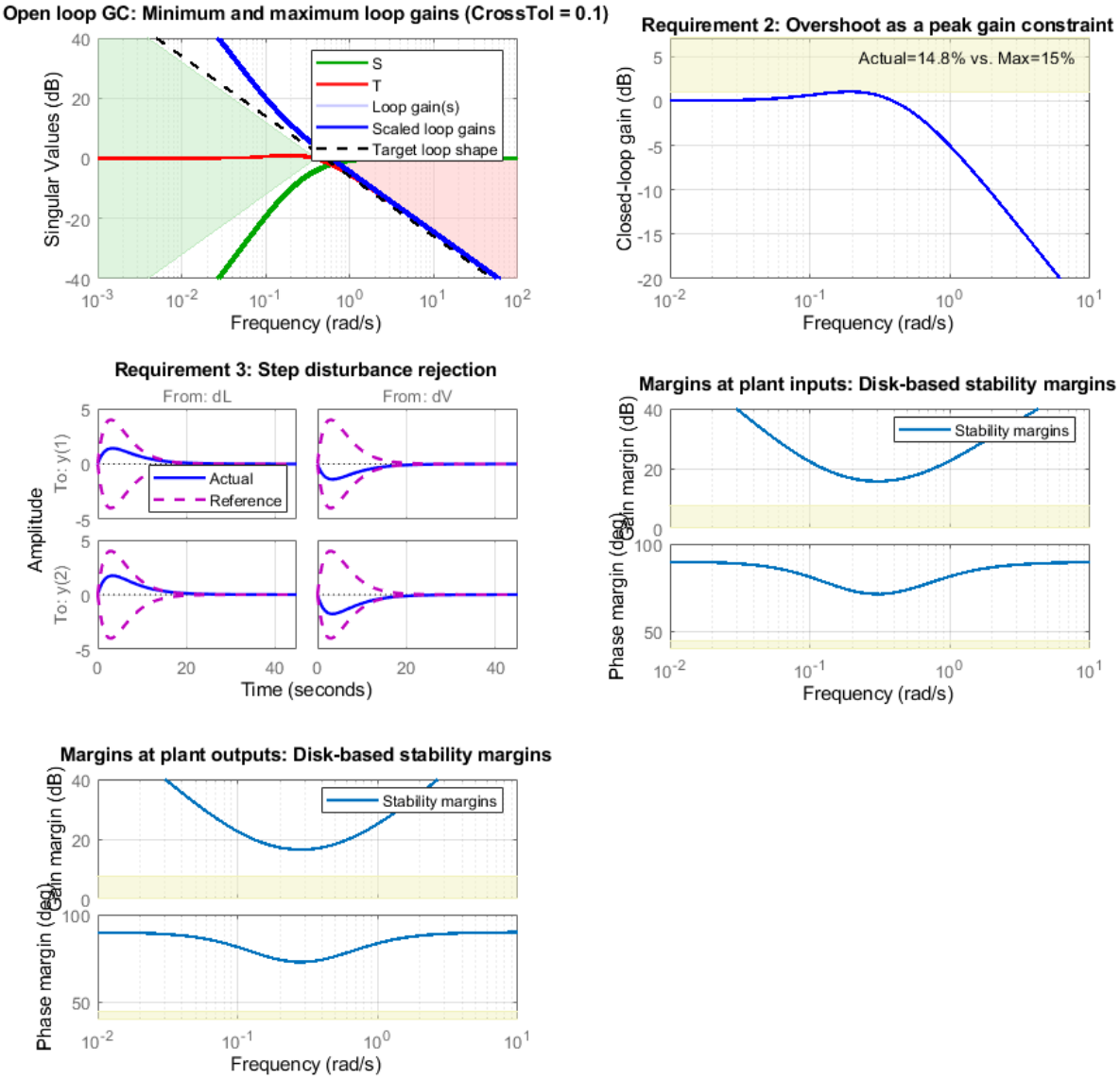
Next use `looptune` to tune the controller blocks `PI_L`, `PI_V`, and `DM` subject to the disturbance rejection requirement.

```
Controls = {'L','V'};
Measurements = 'y';
[ST,gam,Info] = looptune(ST0,Controls,Measurements,wc,OS,DR);
```

```
Final: Peak gain = 0.997, Iterations = 68
Achieved target gain value TargetGain=1.
```

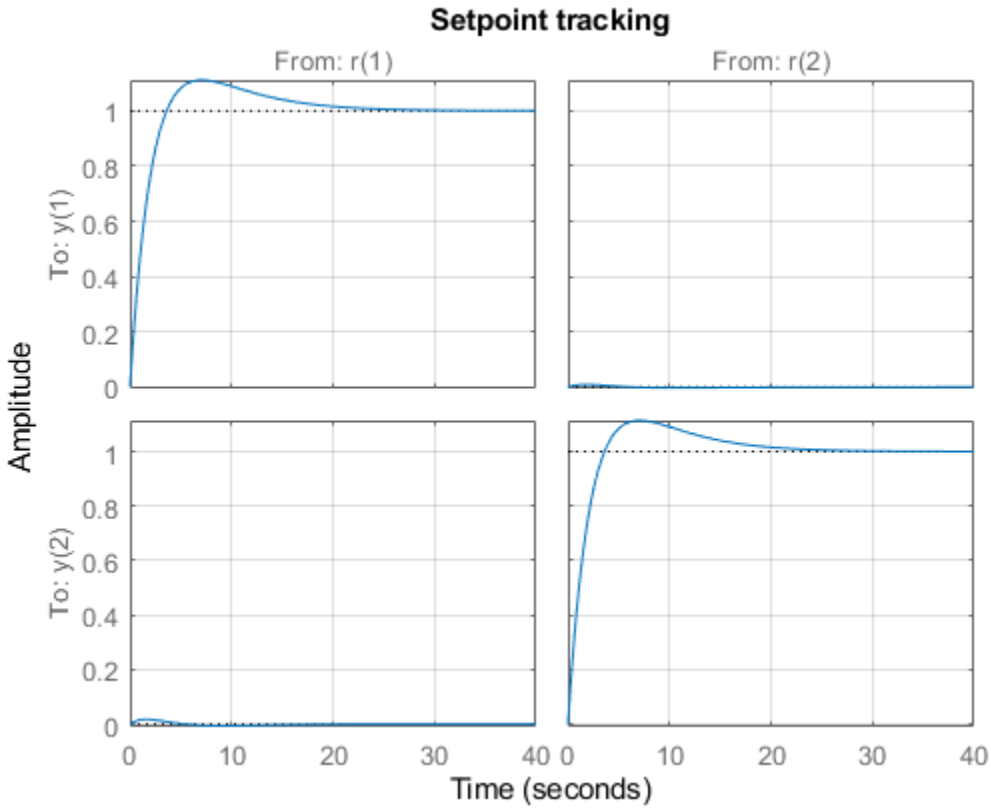
The final value is near 1 which indicates that all requirements were met. Use `loopview` to check the resulting design. The responses should stay outside the shaded areas.

```
figure('Position',[0,0,1000,1200])
loopview(ST,Info)
```

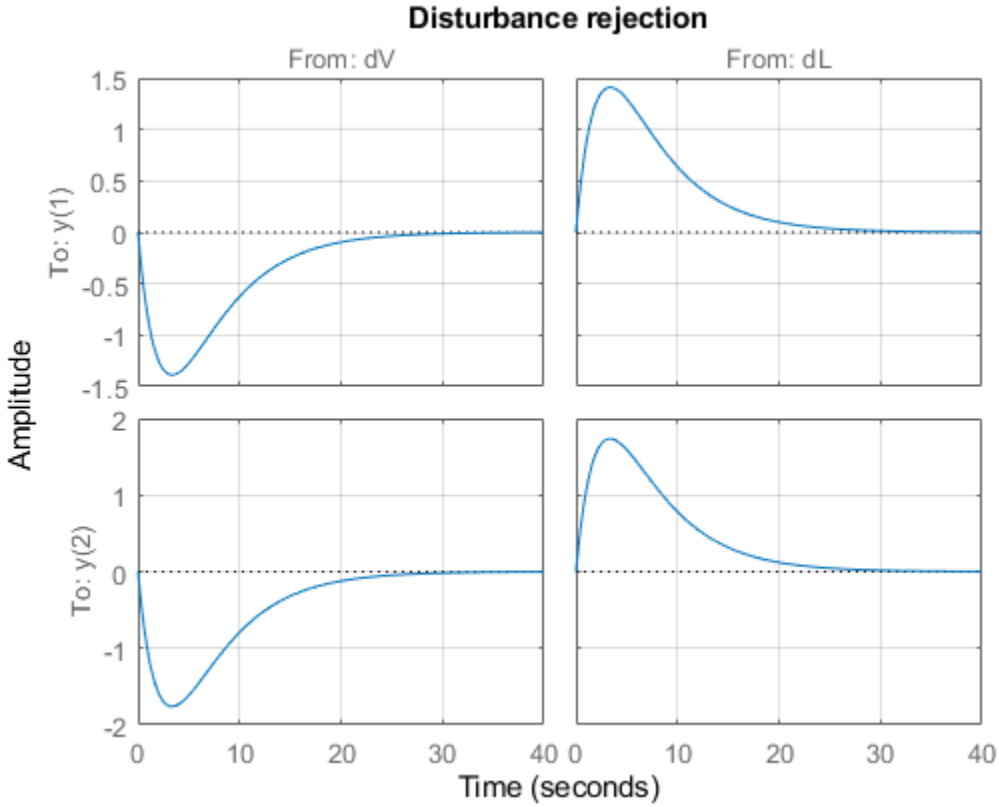


Use `getIOTransfer` to access and plot the closed-loop responses from reference and disturbance to the tops and bottoms concentrations. The tuned responses show a good compromise between tracking and disturbance rejection.

```
figure
Ttrack = getIOTransfer(ST,'r','y');
step(Ttrack,40), grid, title('Setpoint tracking')
```

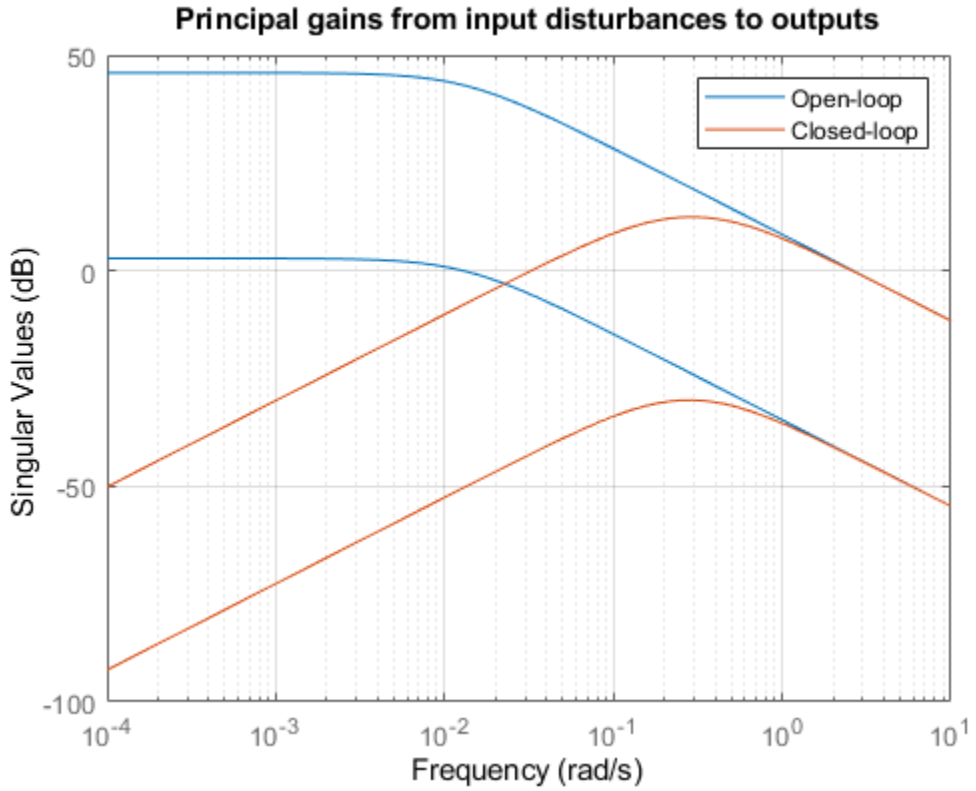



```
Treject = getIOTransfer(ST,{'dV','dL'},'y');
step(Treject,40), grid, title('Disturbance rejection')
```



Comparing the open- and closed-loop disturbance rejection characteristics in the frequency domain shows a clear improvement inside the control bandwidth.

```
clf, sigma(G,Treject), grid
title('Principal gains from input disturbances to outputs')
legend('Open-loop','Closed-loop')
```



Adding Constraints on the Tuned Variables

Inspection of the controller obtained above shows that the second PI controller has negative gains.

```
getBlockValue(ST,'PI_V')
```

```
ans =

             1
  Kp + Ki * ---
             s

  with Kp = -3.3, Ki = -0.455

Name: PI_V
Continuous-time PI controller in parallel form.
```

This is due to the negative signs in the second input channels of the plant G. In addition, the tunable elements are over-parameterized because multiplying `DM` by two and dividing the PI gains by two does not change the overall controller. To address these issues, fix the (1,1) entry of `DM` to 1 and the (2,2) entry to -1.

```
DM = getBlockParam(ST0,'DM');
DM.Gain.Value = diag([1 -1]);
DM.Gain.Free = [false true;true false];
setBlockParam(ST0,'DM',DM)
```

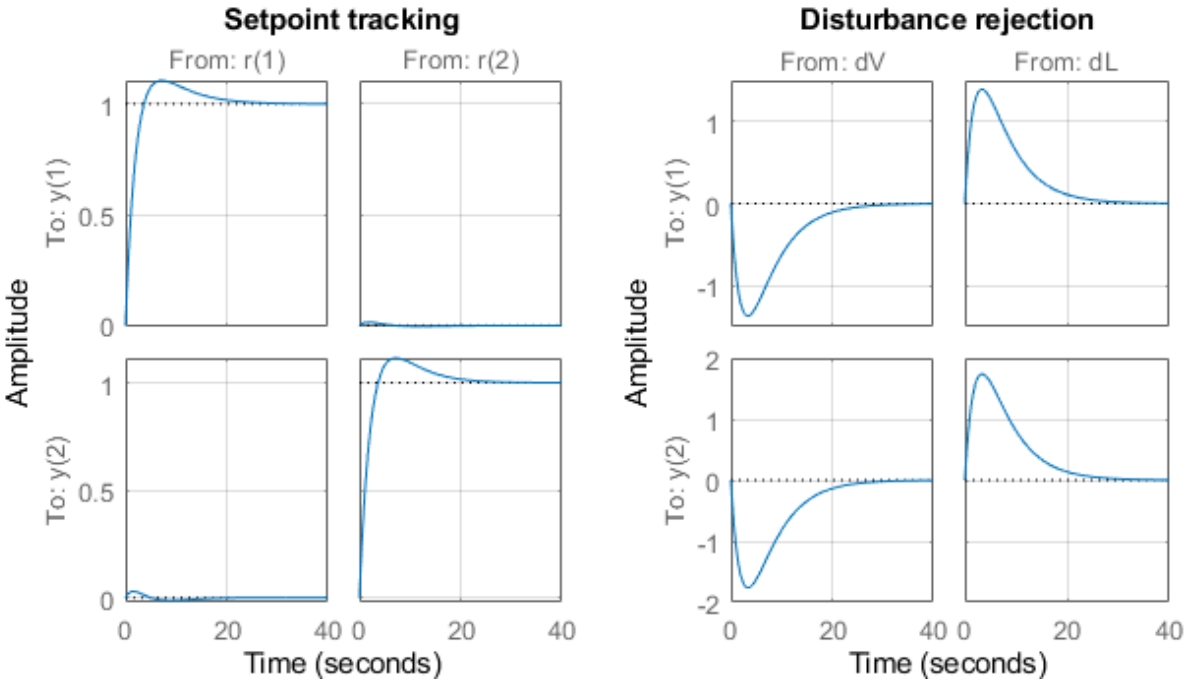
Re-tune the controller for the reduced set of tunable parameters.

```
[ST,gam,Info] = looptune(ST0,Controls,Measurements,wc,OS,DR);
```

```
Final: Peak gain = 0.999, Iterations = 81
Achieved target gain value TargetGain=1.
```

The step responses look similar but the values of DM and the PI gains are more suitable for implementation.

```
figure('Position',[0,0,700,350])

subplot(121)
Ttrack = getIOTransfer(ST,'r','y');
step(Ttrack,40), grid, title('Setpoint tracking')

subplot(122)
Treject = getIOTransfer(ST,{'dV','dL'},'y');
step(Treject,40), grid, title('Disturbance rejection')
```




```
showTunable(ST)
```

```
Block 1: rct_distillation/PI_L =

             1
  Kp + Ki * ---
             s

  with Kp = 16.9, Ki = 2.25

Name: PI_L
Continuous-time PI controller in parallel form.
```

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

Block 2: rct_distillation/PI_V =

             1
  Kp + Ki * ---
             s

  with Kp = 13.3, Ki = 1.77

Name: PI_V
Continuous-time PI controller in parallel form.

-----------------------------------

Block 3: rct_distillation/DM =

  D =
           u1      u2
   y1       1  -0.773
   y2   1.253      -1

Name: DM
Static gain.
```

Equivalent Workflow in MATLAB

If you do not have a Simulink model of the control system, you can use LTI objects and Control Design blocks to create a MATLAB representation of the following block diagram.

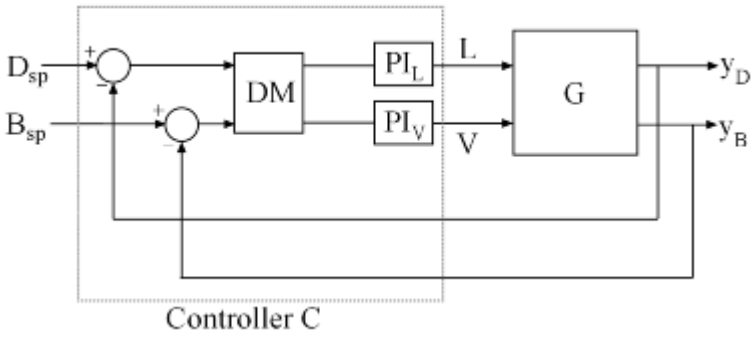


Figure 2: Block Diagram of Control System

First parameterize the tunable elements using Control Design blocks. Use the `tunableGain` object to parameterize `DM` and fix `DM(1,1)=1` and `DM(2,2)=-1`. This creates a 2x2 static gain with the off-diagonal entries as tunable parameters.

```
DM = tunableGain('Decoupler',diag([1 -1]));
DM.Gain.Free = [false true;true false];
```

Similarly, use the `tunablePID` object to parameterize the two PI controllers:

```
PI_L = tunablePID('PI_L','pi');
PI_V = tunablePID('PI_V','pi');
```

Next construct a model `C0` of the controller C in Figure 2.

```
C0 = blkdiag(PI_L,PI_V) * DM * [eye(2) -eye(2)];

% Note: I/O names should be consistent with those of G
C0.InputName = {'Dsp','Bsp','yD','yB'};
C0.OutputName = {'L','V'};
```

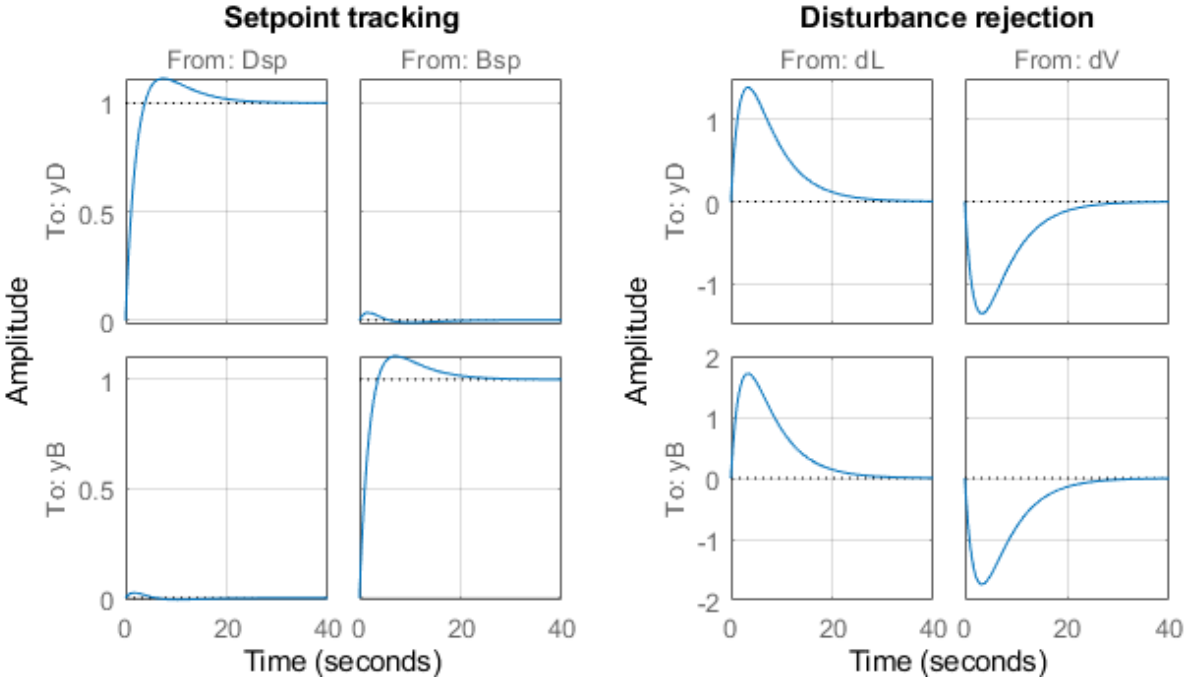
Now tune the controller parameters with `looptune` as done previously.

```
% Crossover frequency
wc = 0.5;

% Overshoot and disturbance rejection requirements
OS = TuningGoal.Overshoot({'Dsp','Bsp'},{'yD','yB'},15);
DR = TuningGoal.StepRejection({'L','V'},{'yD','yB'},4,20);

% Tune controller gains
[~,C] = looptune(G,C0,wc,OS,DR);
```

```
Final: Peak gain = 0.999, Iterations = 64
Achieved target gain value TargetGain=1.
```

To validate the design, close the loop with the tuned compensator `C` and simulate the step responses for setpoint tracking and disturbance rejection.

```
Tcl = connect(G,C,{'Dsp','Bsp','L','V'},{'yD','yB'});

figure('Position',[0,0,700,350])

subplot(121)
Ttrack = Tcl(:,[1 2]);
step(Ttrack,40), grid, title('Setpoint tracking')

subplot(122)
Treject = Tcl(:,[3 4]);
Treject.InputName = {'dL','dV'};
step(Treject,40), grid, title('Disturbance rejection')
```



The results are similar to those obtained in Simulink.

See Also

`looptune` | `looptune (slTuner)`

More About

- “Tuning of a Digital Motion Control System”

Tuning of a Digital Motion Control System

This example shows how to use Control System Toolbox™ to tune a digital motion control system.

Motion Control System

The motion system under consideration is shown below.

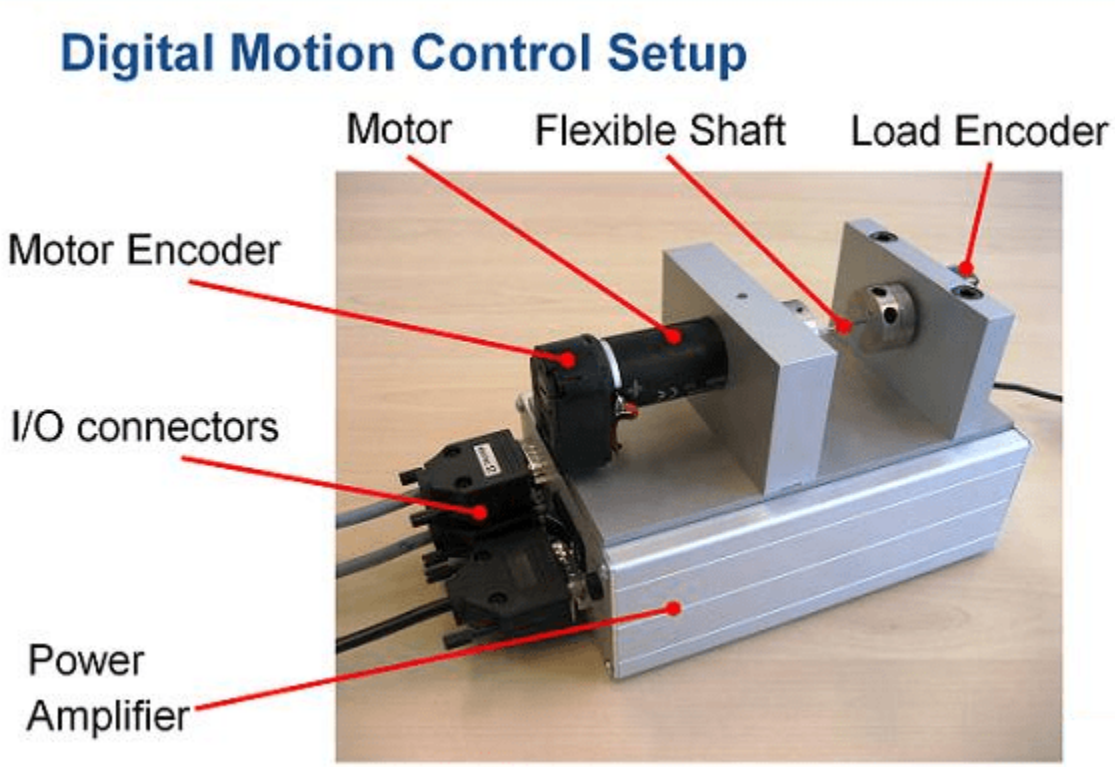
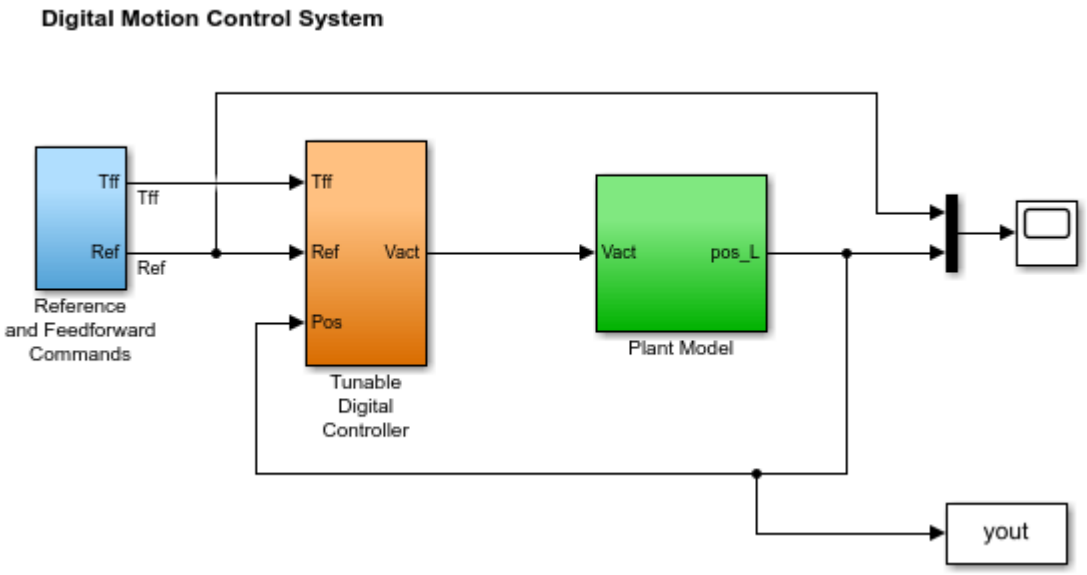


Figure 1: Digital motion control hardware

This device could be part of some production machine and is intended to move some load (a gripper, a tool, a nozzle, or anything else that you can imagine) from one angular position to another and back again. This task is part of the "production cycle" that has to be completed to create each product or batch of products.

The digital controller must be tuned to maximize the production speed of the machine without compromising accuracy and product quality. To do this, we first model the control system in Simulink® using a 4th-order model of the inertia and flexible shaft:

```
open_system('rct_dmc')
```



The "Tunable Digital Controller" consists of a gain in series with a lead/lag controller.

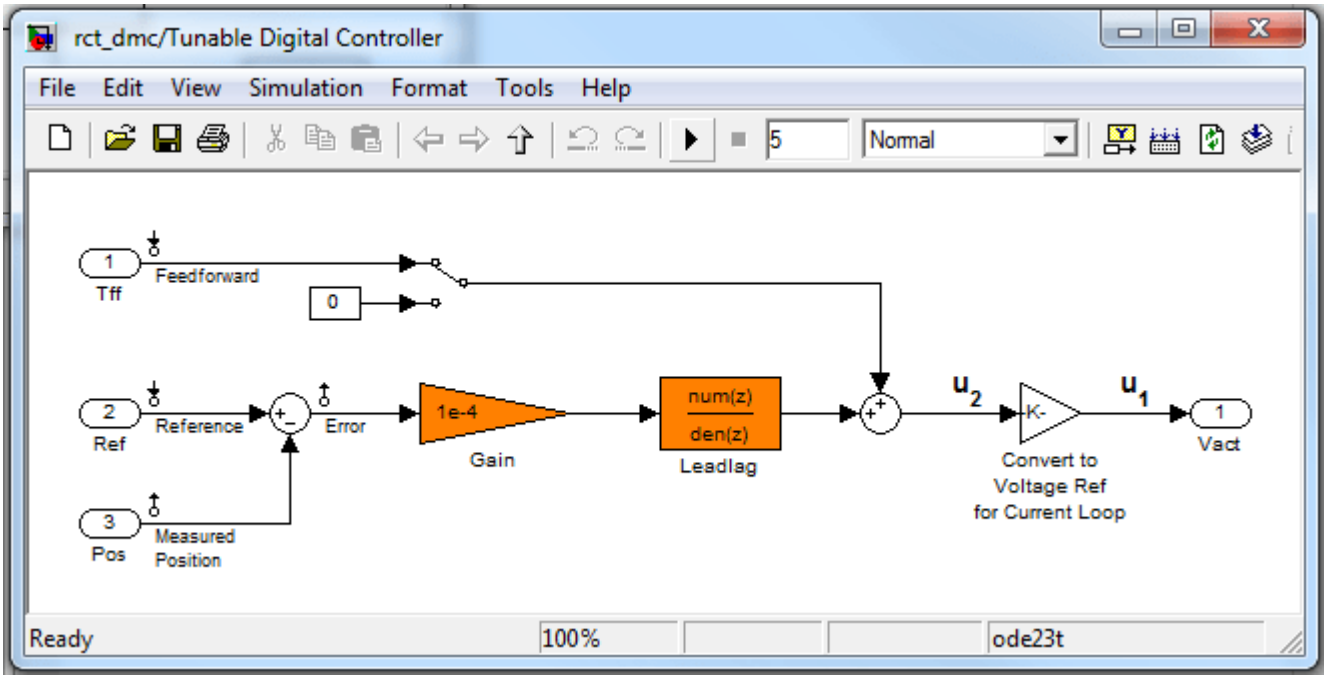
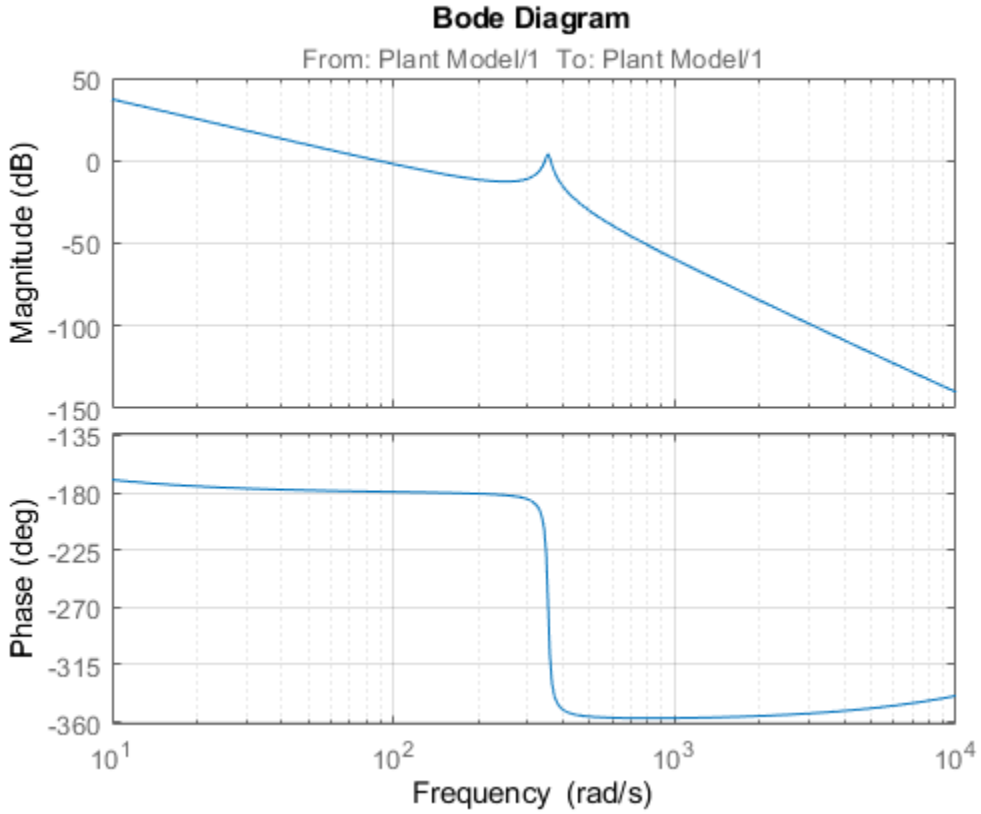


Figure 2: Digital controller

Tuning is complicated by the presence of a flexible mode near 350 rad/s in the plant:

```
G = linearize('rct_dmc','rct_dmc/Plant Model');
bode(G,{10,1e4}), grid
```



Compensator Tuning

We are seeking a 0.5 second response time to a step command in angular position with minimum overshoot. This corresponds to a target bandwidth of approximately 5 rad/s. The `looptune` command offers a convenient way to tune fixed-structure compensators like the one in this application. To use `looptune`, first instantiate the `slTuner` interface to automatically acquire the control structure from Simulink. Note that the signals of interest are already marked as Linear Analysis Points in the Simulink model.

```
ST0 = slTuner('rct_dmc',{'Gain','Leadlag'});
```

Next use `looptune` to tune the compensator parameters for the target gain crossover frequency of 5 rad/s:

```
Measurement = 'Measured Position';  % controller input
Control = 'Leadlag';                % controller output
ST1 = looptune(ST0,Control,Measurement,5);
```

```
Final: Peak gain = 0.979, Iterations = 19
Achieved target gain value TargetGain=1.
```

A final value below or near 1 indicates success. Inspect the tuned values of the gain and lead/lag filter:

```
showTunable(ST1)
```

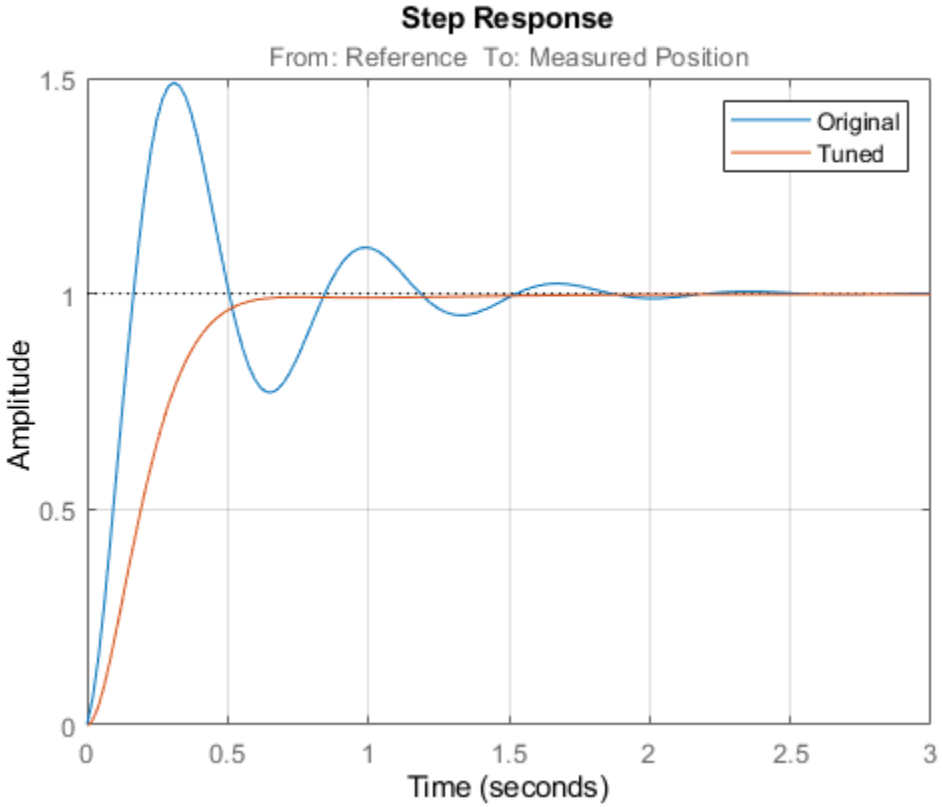
```
Block 1: rct_dmc/Tunable Digital Controller/Gain =
```

```
  D =
              u1
   y1  1.869e-05

Name: Gain
Static gain.

-----------------------------------

Block 2: rct_dmc/Tunable Digital Controller/Leadlag =

  3.856 s + 6.323
  ---------------
     s + 13.35

Name: Leadlag
Continuous-time transfer function.
```

Design Validation

To validate the design, use the `slTuner` interface to quickly access the closed-loop transfer functions of interest and compare the responses before and after tuning.

```
T0 = getIOTransfer(ST0,'Reference','Measured Position');
T1 = getIOTransfer(ST1,'Reference','Measured Position');
step(T0,T1), grid
legend('Original','Tuned')
```



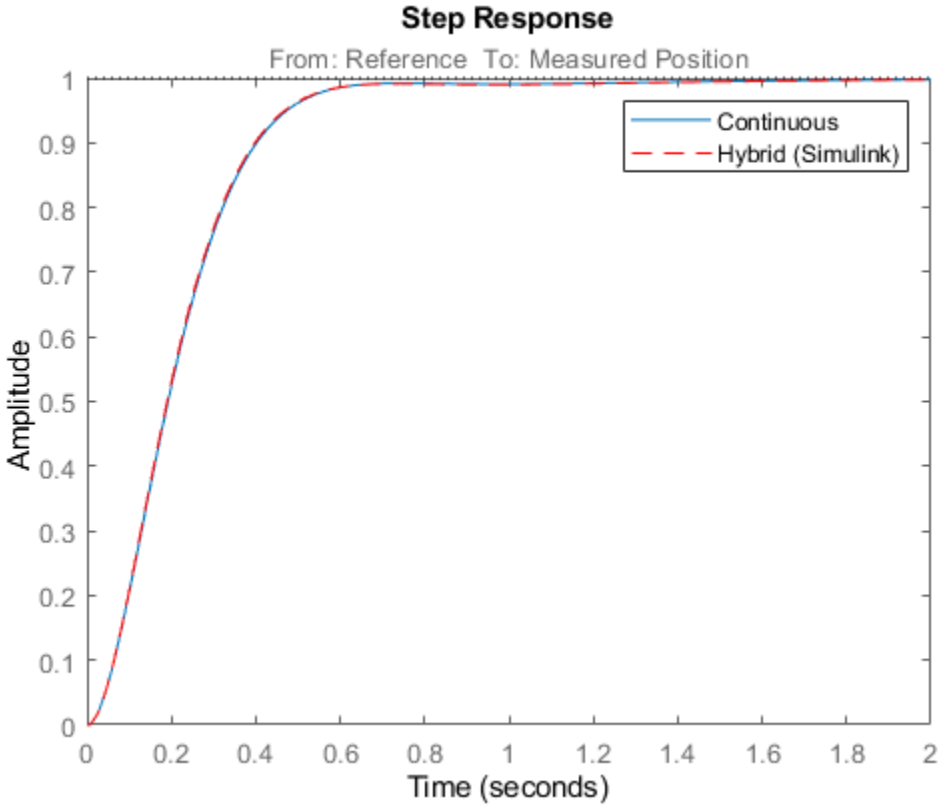
The tuned response has significantly less overshoot and satisfies the response time requirement. However these simulations are obtained using a continuous-time lead/lag compensator (`looptune` operates in continuous time) so we need to further validate the design in Simulink using a digital implementation of the lead/lag compensator. Use `writeBlockValue` to apply the tuned values to the Simulink model and automatically discretize the lead/lag compensator to the rate specified in Simulink.

```
writeBlockValue(ST1)
```

You can now simulate the response of the continuous-time plant with the digital controller:

```
sim('rct_dmc');  % angular position logged in "yout" variable
t = yout.time;
y = yout.signals.values;
step(T1), hold, plot(t,y,'r--')
legend('Continuous','Hybrid (Simulink)')
```

```
Current plot held
```



The simulations closely match and the coefficients of the digital lead/lag can be read from the "Leadlag" block in Simulink.

Tuning an Additional Notch Filter

Next try to increase the control bandwidth from 5 to 50 rad/s. Because of the plant resonance near 350 rad/s, the lead/lag compensator is no longer sufficient to get adequate stability margins and small overshoot. One remedy is to add a notch filter as shown in Figure 3.

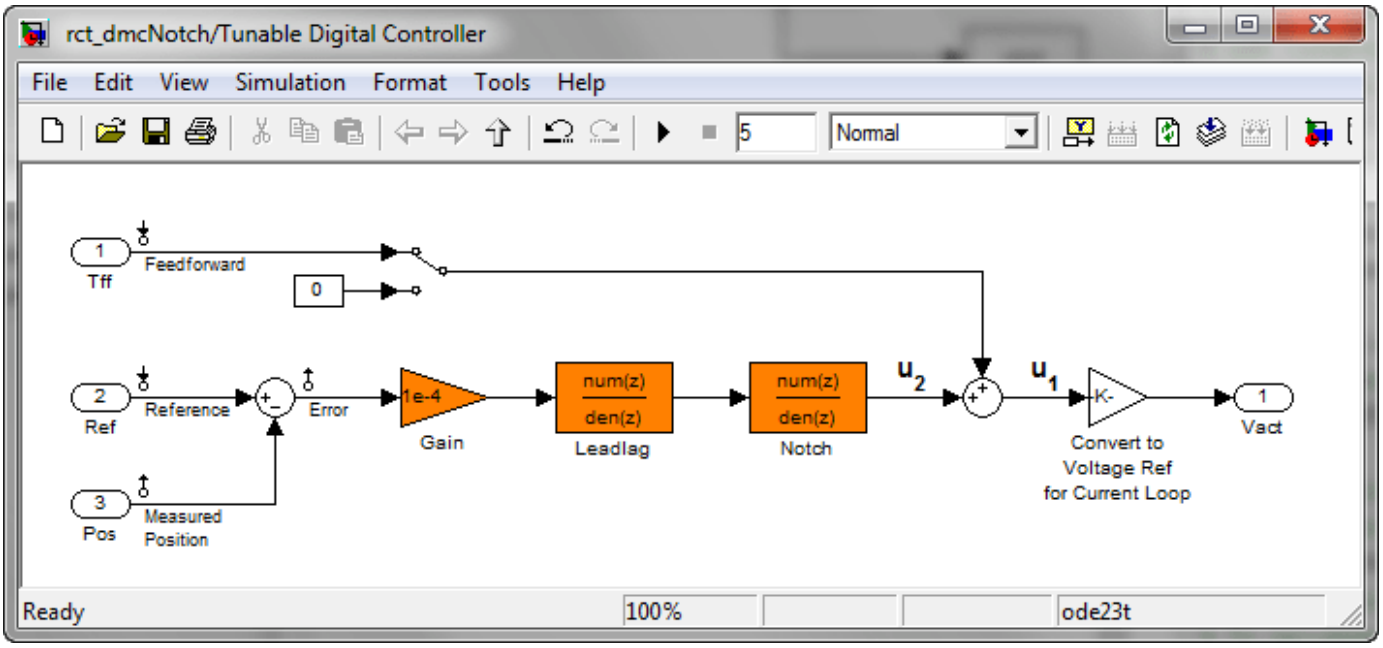


Figure 3: Digital Controller with Notch Filter

To tune this modified control architecture, create an `slTuner` instance with the three tunable blocks.

```
ST0 = slTuner('rct_dmcNotch',{'Gain','Leadlag','Notch'});
```

By default the "Notch" block is parameterized as any second-order transfer function. To retain the notch structure

$$N(s) = \frac{s^2 + 2\zeta_1\omega_n s + \omega_n^2}{s^2 + 2\zeta_2\omega_n s + \omega_n^2},$$

specify the coefficients $\omega_n, \zeta_1, \zeta_2$ as real parameters and create a parametric model `N` of the transfer function shown above:

```
wn = realp('wn',300);
zeta1 = realp('zeta1',1);
zeta2 = realp('zeta2',1);
zeta1.Minimum = 0;   zeta1.Maximum = 1;   % 0 <= zeta1 <= 1
zeta2.Minimum = 0;   zeta2.Maximum = 1;   % 0 <= zeta2 <= 1
N = tf([1 2*zeta1*wn wn^2],[1 2*zeta2*wn wn^2]);  % tunable notch filter
```

Then associate this parametric notch model with the "Notch" block in the Simulink model. Because the control system is tuned in the continuous time, you can use a continuous-time parameterization of the notch filter even though the "Notch" block itself is discrete.

```
setBlockParam(ST0,'Notch',N);
```

Next use `looptune` to jointly tune the "Gain", "Leadlag", and "Notch" blocks with a 50 rad/s target crossover frequency. To eliminate residual oscillations from the plant resonance, specify a target loop shape with a -40 dB/decade roll-off past 50 rad/s.

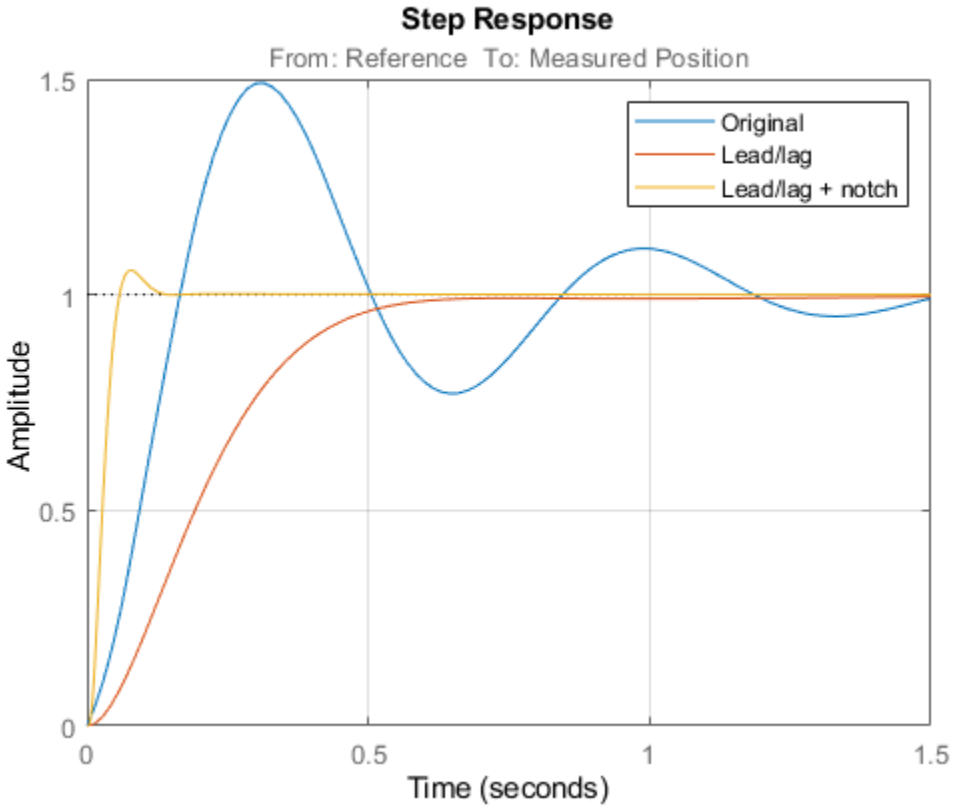
```
% Specify target loop shape with a few frequency points
Freqs = [5 50 500];
```

```
Gains = [10 1 0.01];
TLS = TuningGoal.LoopShape('Notch',frd(Gains,Freqs));

Measurement = 'Measured Position';  % controller input
Control = 'Notch';                  % controller output
ST2 = looptune(ST0,Control,Measurement,TLS);
```

```
Final: Peak gain = 1.05, Iterations = 60
```

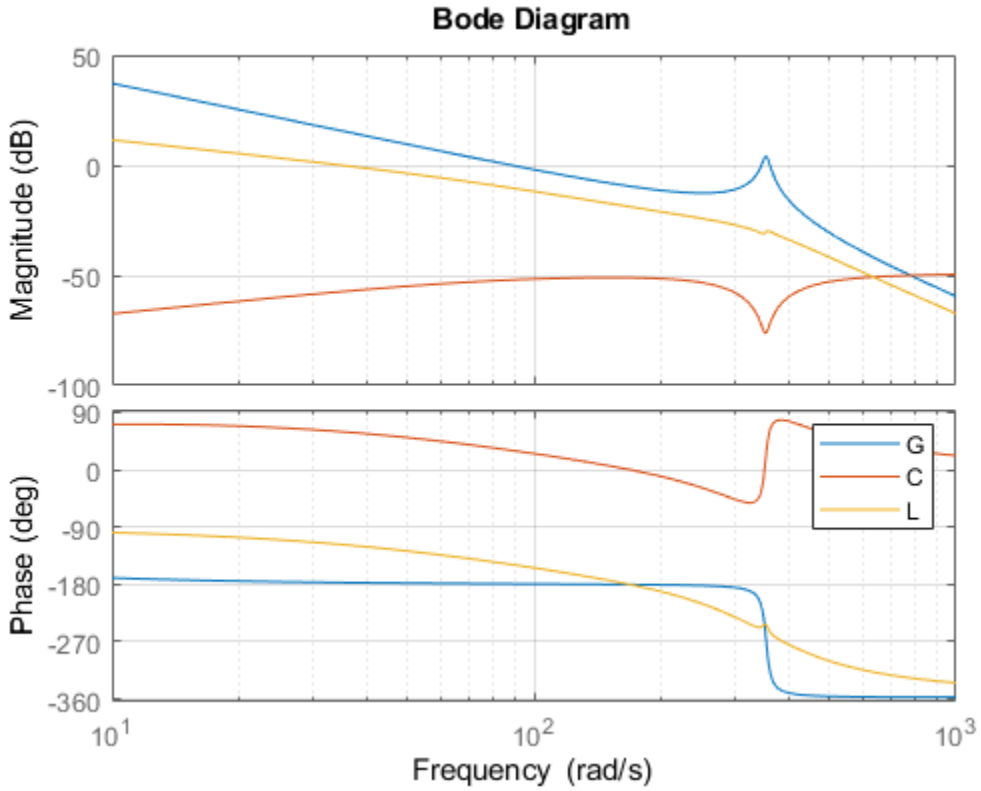
The final gain is close to 1, indicating that all requirements are met. Compare the closed-loop step response with the previous designs.

```
T2 = getIOTransfer(ST2,'Reference','Measured Position');
clf
step(T0,T1,T2,1.5), grid
legend('Original','Lead/lag','Lead/lag + notch')
```



To verify that the notch filter performs as expected, evaluate the total compensator `C` and the open-loop response `L` and compare the Bode responses of `G`, `C`, `L`:

```
% Get tuned block values (in the order blocks are listed in ST2.TunedBlocks)
[g,LL,N] = getBlockValue(ST2,'Gain','Leadlag','Notch');
C = N * LL * g;

L = getLoopTransfer(ST2,'Notch',-1);

bode(G,C,L,{1e1,1e3}), grid
legend('G','C','L')
```



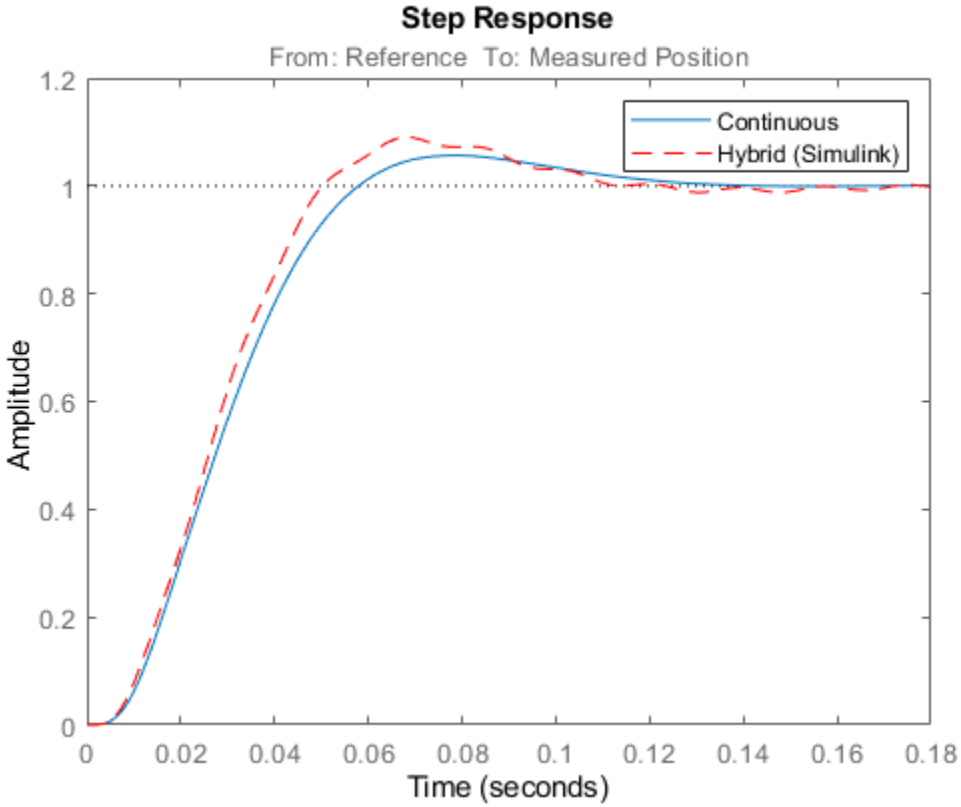
This Bode plot confirms that the plant resonance has been correctly "notched out."

Discretizing the Notch Filter

Again use `writeBlockValue` to discretize the tuned lead/lag and notch filters and write their values back to Simulink. Compare the MATLAB and Simulink responses:

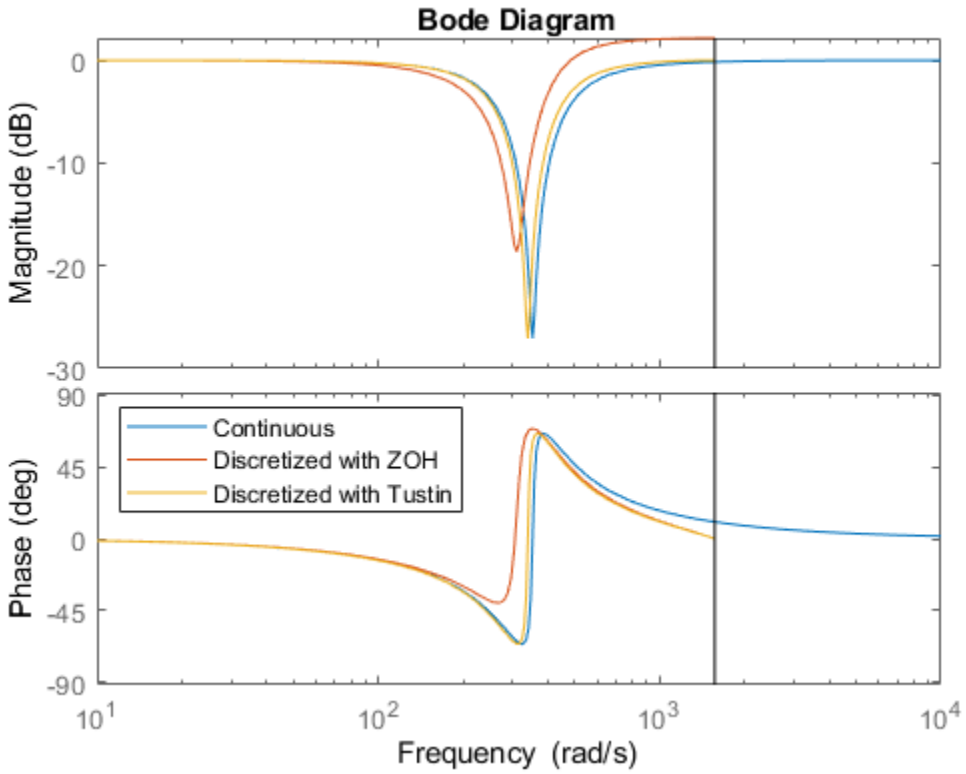
```
writeBlockValue(ST2)

sim('rct_dmcNotch');
t = yout.time;
y = yout.signals.values;
step(T2), hold, plot(t,y,'r--')
legend('Continuous','Hybrid (Simulink)')
```

```
Current plot held
```



The Simulink response exhibits small residual oscillations. The notch filter discretization is the likely culprit because the notch frequency is close to the Nyquist frequency `pi/0.002=1570` rad/s. By default the notch is discretized using the ZOH method. Compare this with the Tustin method prewarped at the notch frequency:

```
wn = damp(N);  % natural frequency of the notch filter
Ts = 0.002;    % sample time of discrete notch filter

Nd1 = c2d(N,Ts,'zoh');
Nd2 = c2d(N,Ts,'tustin',c2dOptions('PrewarpFrequency',wn(1)));

clf, bode(N,Nd1,Nd2)
legend('Continuous','Discretized with ZOH','Discretized with Tustin',...
    'Location','NorthWest')
```

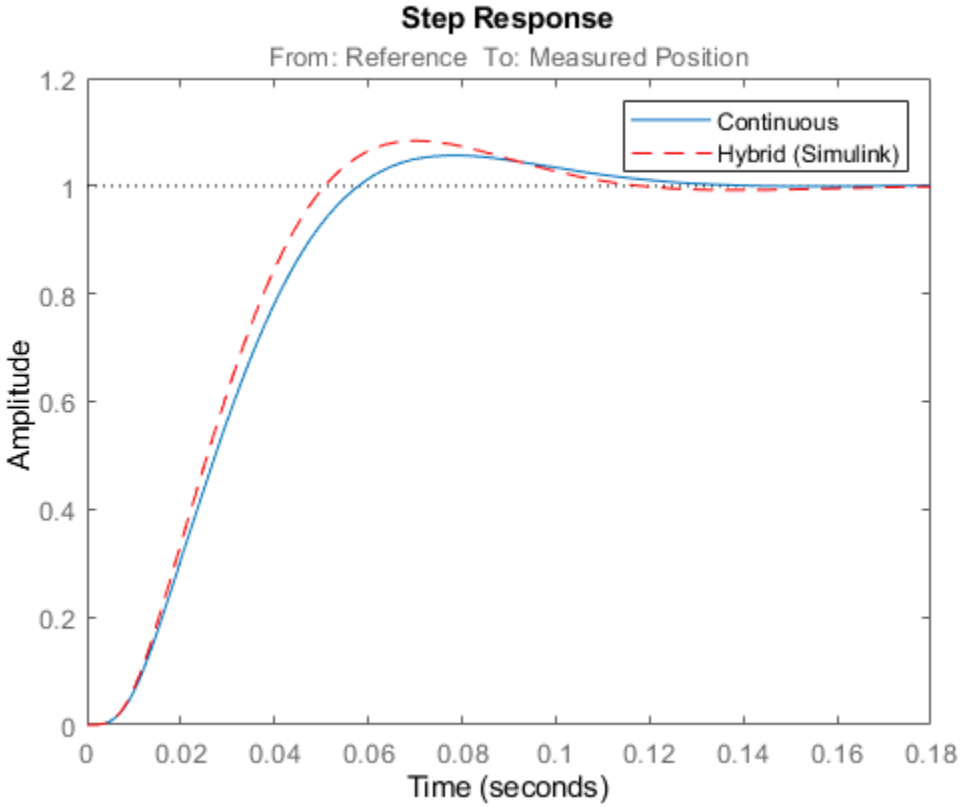


The ZOH method has significant distortion and prewarped Tustin should be used instead. To do this, specify the desired rate conversion method for the notch filter block:

```
setBlockRateConversion(ST2,'Notch','tustin',wn(1))

writeBlockValue(ST2)
```

`writeBlockValue` now uses Tustin prewarped at the notch frequency to discretize the notch filter and write it back to Simulink. Verify that this gets rid of the oscillations.

```
sim('rct_dmcNotch');
t = yout.time;
y = yout.signals.values;
step(T2), hold, plot(t,y,'r--')
legend('Continuous','Hybrid (Simulink)')

Current plot held
```



Direct Discrete-Time Tuning

Alternatively, you can tune the controller directly in discrete time to avoid discretization issues with the notch filter. To do this, specify that the Simulink model should be linearized and tuned at the controller sample time of 0.002 seconds:

```
ST0 = slTuner('rct_dmcNotch',{'Gain','Leadlag','Notch'});
ST0.Ts = 0.002;
```

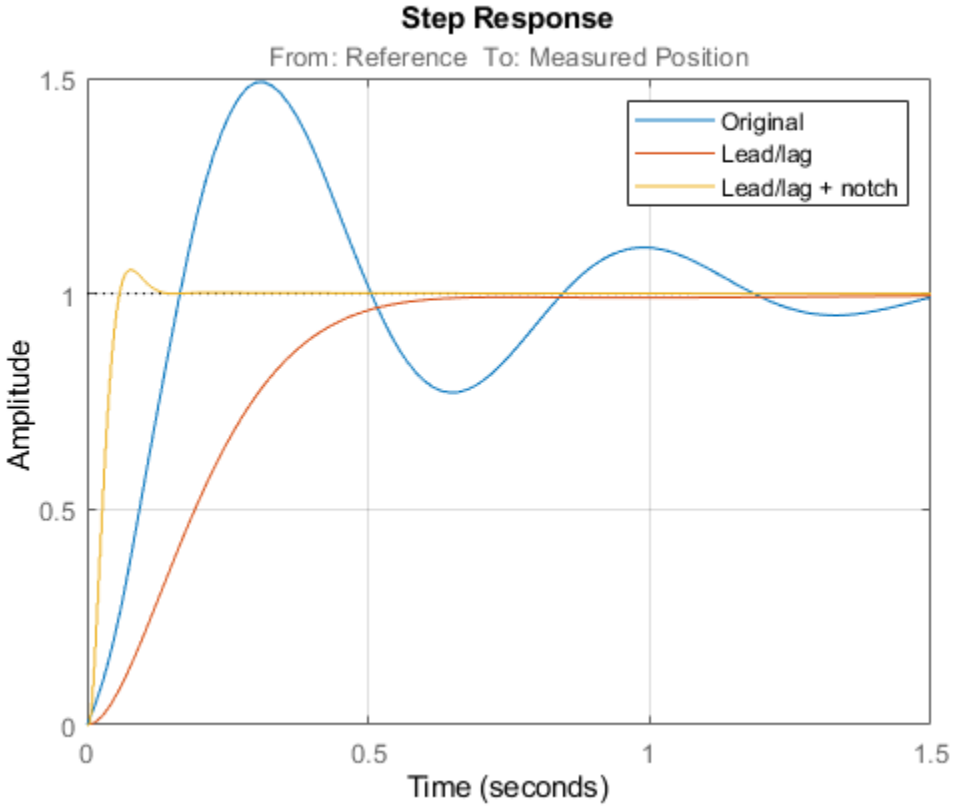
To prevent high-gain control and saturations, add a requirement that limits the gain from reference to control signal (output of Notch block).

```
GL = TuningGoal.Gain('Reference','Notch',0.01);
```

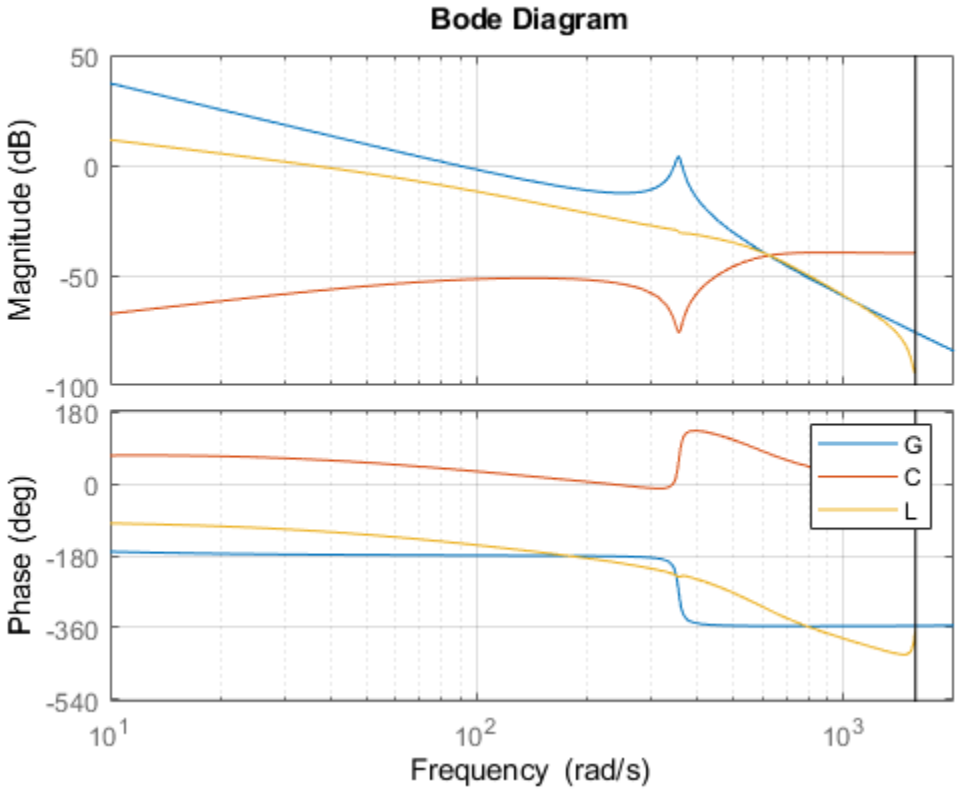
Now retune the controller at the specified sampling rate and verify the tuned open- and closed-loop responses.

```
ST2 = looptune(ST0,Control,Measurement,TLS,GL);

% Closed-loop responses
T2 = getIOTransfer(ST2,'Reference','Measured Position');
clf
step(T0,T1,T2,1.5), grid
legend('Original','Lead/lag','Lead/lag + notch')
```

```
Final: Peak gain = 1.04, Iterations = 36
```



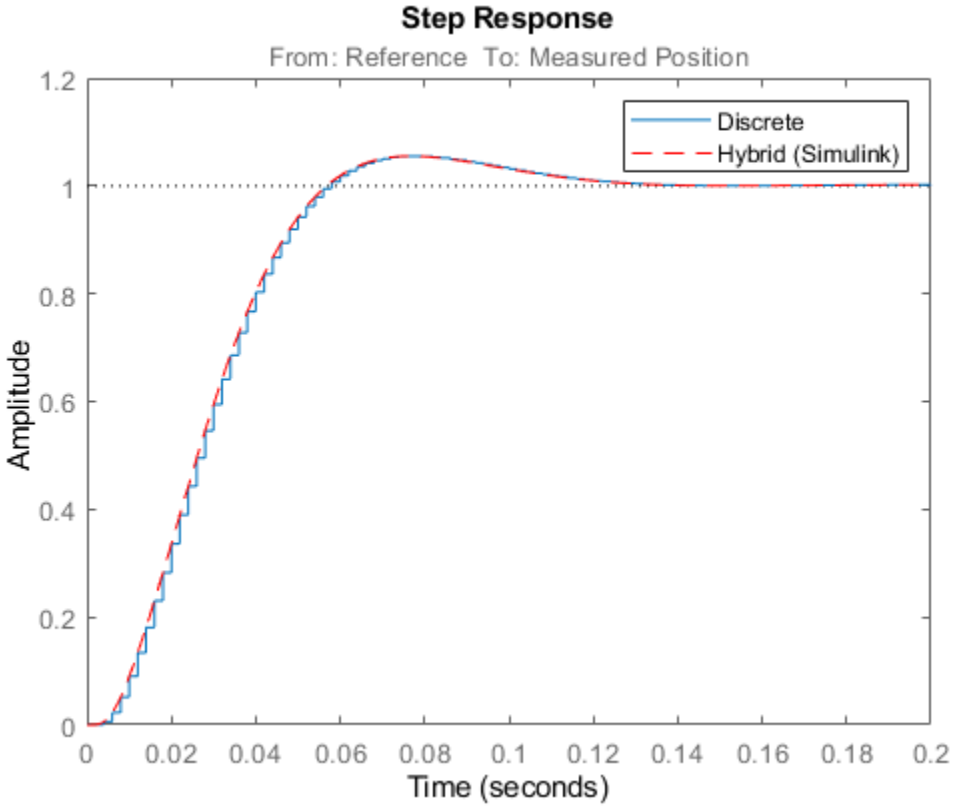

```
% Open-loop responses
[g,LL,N] = getBlockValue(ST2,'Gain','Leadlag','Notch');
C = N * LL * g;
L = getLoopTransfer(ST2,'Notch',-1);
bode(G,C,L,{1e1,2e3}), grid
legend('G','C','L')
```



The results are similar to those obtained when tuning the controller in continuous time. Now validate the digital controller against the continuous-time plant in Simulink.

```
writeBlockValue(ST2)

sim('rct_dmcNotch');
t = yout.time;
y = yout.signals.values;
step(T2), hold, plot(t,y,'r--')
legend('Discrete','Hybrid (Simulink)')
```

```
Current plot held
```



This time, the hybrid response closely matches its discrete-time approximation and no further adjustment of the notch filter is required.

See Also

`looptune (slTuner)`

More About

- “Tune Feedback Loops using looptune”

Control System Tuning Examples

- “Tuning Multiloop Control Systems” on page 13-2
- “PID Tuning for Setpoint Tracking vs. Disturbance Rejection” on page 13-11
- “Time-Domain Specifications” on page 13-20
- “Frequency-Domain Specifications” on page 13-26
- “Loop Shape and Stability Margin Specifications” on page 13-34
- “System Dynamics Specifications” on page 13-40
- “Configuring Design Requirements” on page 13-42
- “Validating Results” on page 13-43
- “Tune Control Systems in Simulink” on page 13-51
- “Tune a Control System Using Control System Tuner” on page 13-59
- “Using Parallel Computing to Accelerate Tuning” on page 13-75
- “Control of a Linear Electric Actuator” on page 13-79
- “Control of a Linear Electric Actuator Using Control System Tuner” on page 13-89
- “Multi-Loop PI Control of a Robotic Arm” on page 13-114
- “Control of an Inverted Pendulum on a Cart” on page 13-129
- “Digital Control of Power Stage Voltage” on page 13-136
- “MIMO Control of Diesel Engine” on page 13-145
- “Tuning of a Two-Loop Autopilot” on page 13-158
- “Multiloop Control of a Helicopter” on page 13-173
- “Fixed-Structure Autopilot for a Passenger Jet” on page 13-180
- “Fault-Tolerant Control of a Passenger Jet” on page 13-191
- “Passive Control of Water Tank Level” on page 13-200
- “Tuning for Multiple Values of Plant Parameters” on page 13-213

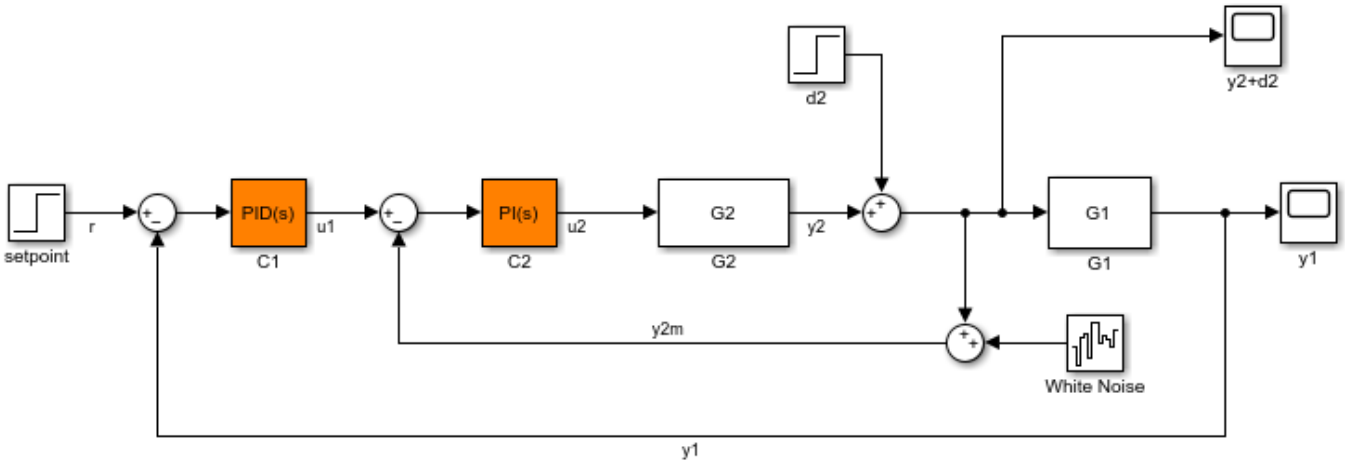
Tuning Multiloop Control Systems

This example shows how to jointly tune the inner and outer loops of a cascade architecture with the `systune` command.

Cascaded PID Loops

Cascade control is often used to achieve smooth tracking with fast disturbance rejection. The simplest cascade architecture involves two control loops (inner and outer) as shown in the block diagram below. The inner loop is typically faster than the outer loop to reject disturbances before they propagate to the outer loop. (Simulink® is not supported in MATLAB® Online.)

```
open_system('rct_cascade')
```



Plant Models and Bandwidth Requirements

In this example, the inner loop plant `G2` is

$$G_2(s) = \frac{3}{s+2}$$

and the outer loop plant `G1` is

$$G_1(s) = \frac{10}{(s+1)^3}$$

```
G2 = zpk([],-2,3);
G1 = zpk([],[-1 -1 -1],10);
```

We use a PI controller in the inner loop and a PID controller in the outer loop. The outer loop must have a bandwidth of at least 0.2 rad/s and the inner loop bandwidth should be ten times larger for adequate disturbance rejection.

Tuning the PID Controllers with SYSTUNE

When the control system is modeled in Simulink, use the `slTuner` interface in Simulink Control Design™ to set up the tuning task. List the tunable blocks, mark the signals `r` and `d2` as inputs of

interest, and mark the signals `y1` and `y2` as locations where to measure open-loop transfers and specify loop shapes.

```
ST0 = slTuner('rct_cascade',{'C1','C2'});
addPoint(ST0,{'r','d2','y1','y2'})
```

You can query the current values of `C1` and `C2` in the Simulink model using `showTunable`. The control system is unstable for these initial values as confirmed by simulating the Simulink model.

```
showTunable(ST0)
```

```
Block 1: rct_cascade/C1 =

             1
  Kp + Ki * ---
             s

  with Kp = 0.1, Ki = 0.1

Name: C1
Continuous-time PI controller in parallel form.

-----------------------------------

Block 2: rct_cascade/C2 =

             1
  Kp + Ki * ---
             s

  with Kp = 0.1, Ki = 0.1

Name: C2
Continuous-time PI controller in parallel form.
```

Next use "LoopShape" requirements to specify the desired bandwidths for the inner and outer loops. Use $0.2/s$ as the target loop shape for the outer loop to enforce integral action with a gain crossover frequency at 0.2 rad/s:

```
% Outer loop bandwidth = 0.2
s = tf('s');
Req1 = TuningGoal.LoopShape('y1',0.2/s); % loop transfer measured at y1
Req1.Name = 'Outer Loop';
```

Use $2/s$ for the inner loop to make it ten times faster (higher bandwidth) than the outer loop. To constrain the inner loop transfer, make sure to open the outer loop by specifying `y1` as a loop opening:

```
% Inner loop bandwidth = 2
Req2 = TuningGoal.LoopShape('y2',2/s); % loop transfer measured at y2
Req2.Openings = 'y1';                  % with outer loop opened at y1
Req2.Name = 'Inner Loop';
```

You can now tune the PID gains in `C1` and `C2` with `systune`:

```
ST = systune(ST0,[Req1,Req2]);
```

```
Final: Soft = 0.859, Hard = -Inf, Iterations = 70
```

Use `showTunable` to see the tuned PID gains.

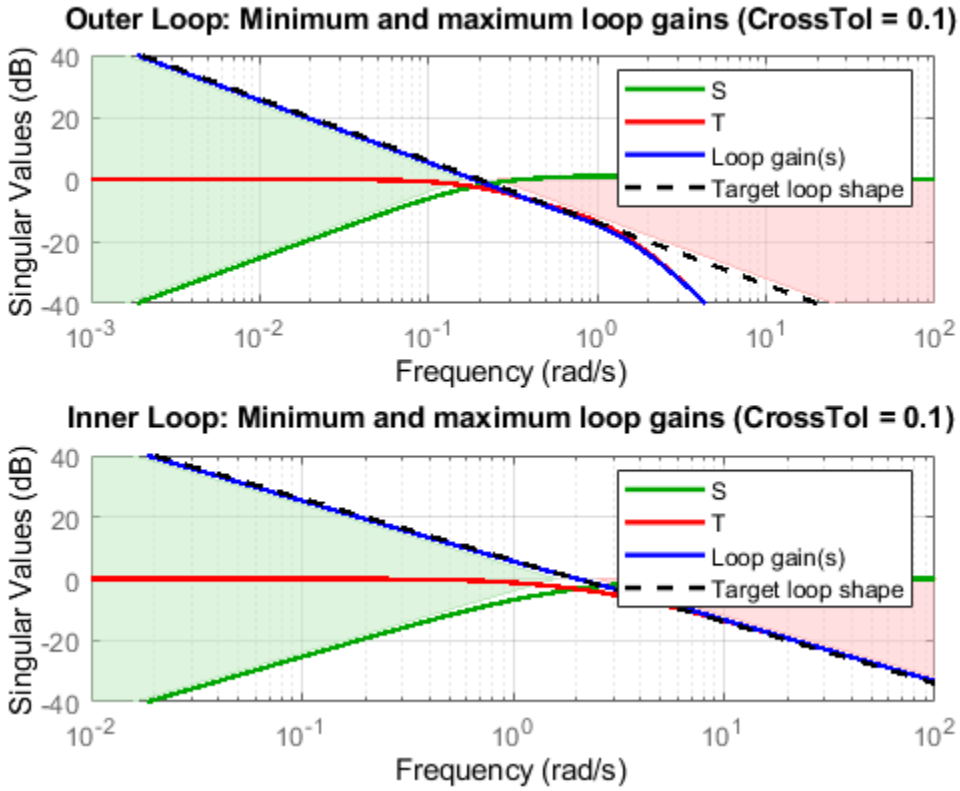
```
showTunable(ST)
```

```
Block 1: rct_cascade/C1 =

             1            s
  Kp + Ki * --- + Kd * --------
             s          Tf*s+1

  with Kp = 0.0521, Ki = 0.0186, Kd = 0.0474, Tf = 0.00602

Name: C1
Continuous-time PIDF controller in parallel form.

-----------------------------------

Block 2: rct_cascade/C2 =

             1
  Kp + Ki * ---
             s

  with Kp = 0.721, Ki = 1.23

Name: C2
Continuous-time PI controller in parallel form.
```

Validating the Design

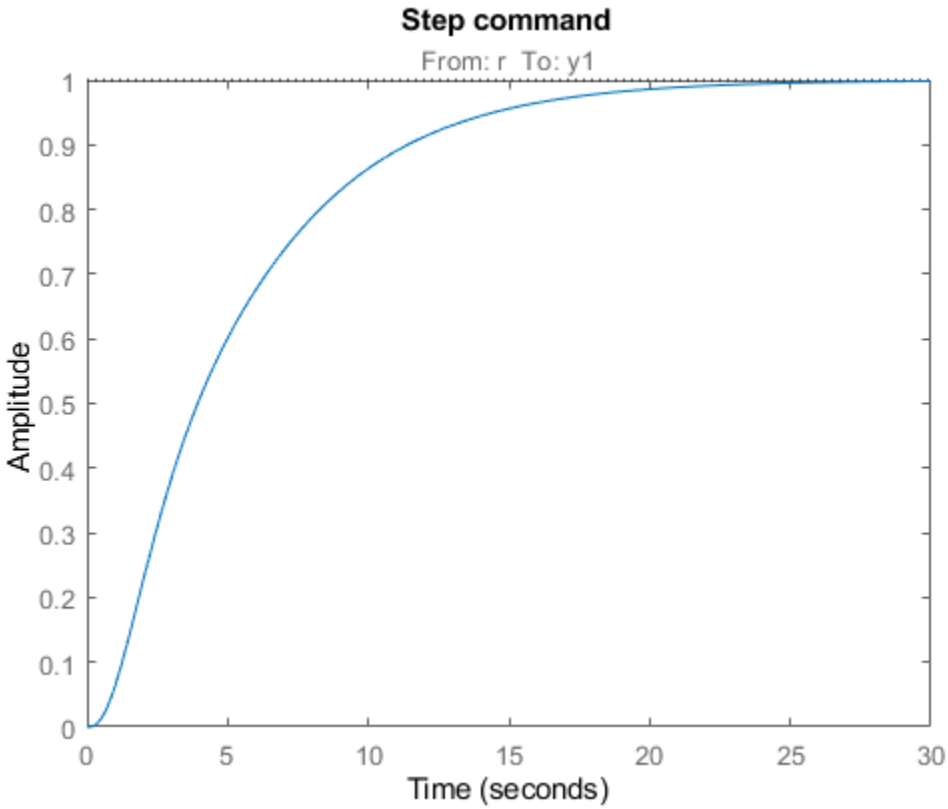
The final value is less than 1 which means that `systune` successfully met both loop shape requirements. Confirm this by inspecting the tuned control system `ST` with `viewGoal`

```
viewGoal([Req1,Req2],ST)
```

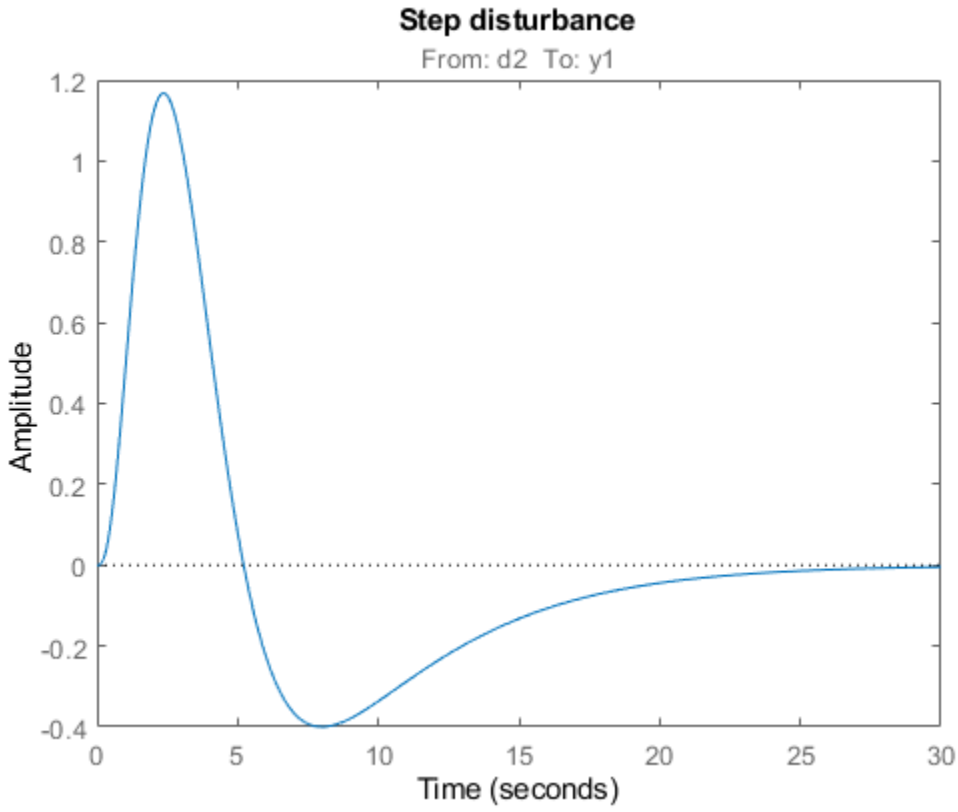


Note that the inner and outer loops have the desired gain crossover frequencies. To further validate the design, plot the tuned responses to a step command r and step disturbance d2:

```
% Response to a step command
H = getIOTransfer(ST,'r','y1');
clf, step(H,30), title('Step command')
```




```
% Response to a step disturbance
H = getIOTransfer(ST,'d2','y1');
step(H,30), title('Step disturbance')
```



Once you are satisfied with the linear analysis results, use `writeBlockValue` to write the tuned PID gains back to the Simulink blocks. You can then conduct a more thorough validation in Simulink.

```
writeBlockValue(ST)
```

Equivalent Workflow in MATLAB

If you do not have a Simulink model of the control system, you can perform the same steps using LTI models of the plant and Control Design blocks to model the tunable elements.

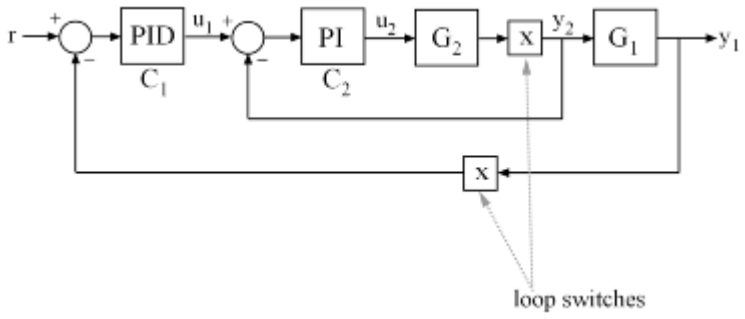


Figure 1: Cascade Architecture

First create parametric models of the tunable PI and PID controllers.

```
C1 = tunablePID('C1','pid');
C2 = tunablePID('C2','pi');
```

Then use "analysis point" blocks to mark the loop opening locations `y1` and `y2`.

```
LS1 = AnalysisPoint('y1');
LS2 = AnalysisPoint('y2');
```

Finally, create a closed-loop model `T0` of the overall control system by closing each feedback loop. The result is a generalized state-space model depending on the tunable elements `C1` and `C2`.

```
InnerCL = feedback(LS2*G2*C2,1);
T0 = feedback(G1*InnerCL*C1,LS1);
T0.InputName = 'r';
T0.OutputName = 'y1';
```

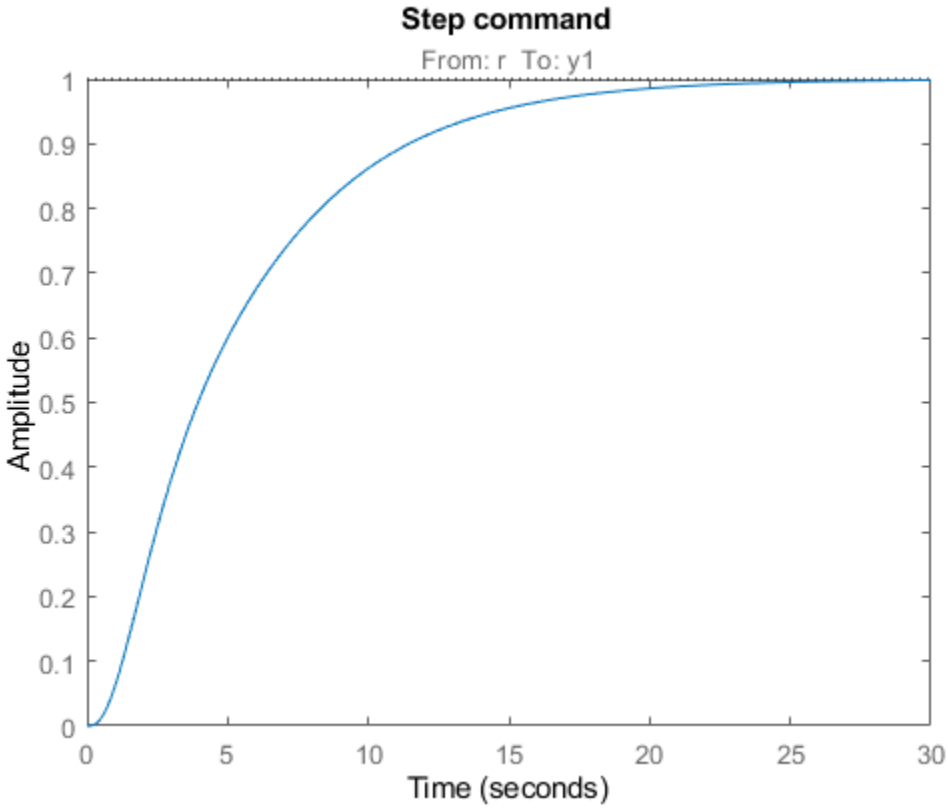
You can now tune the PID gains in `C1` and `C2` with `systune`.

```
T = systune(T0,[Req1,Req2]);
```

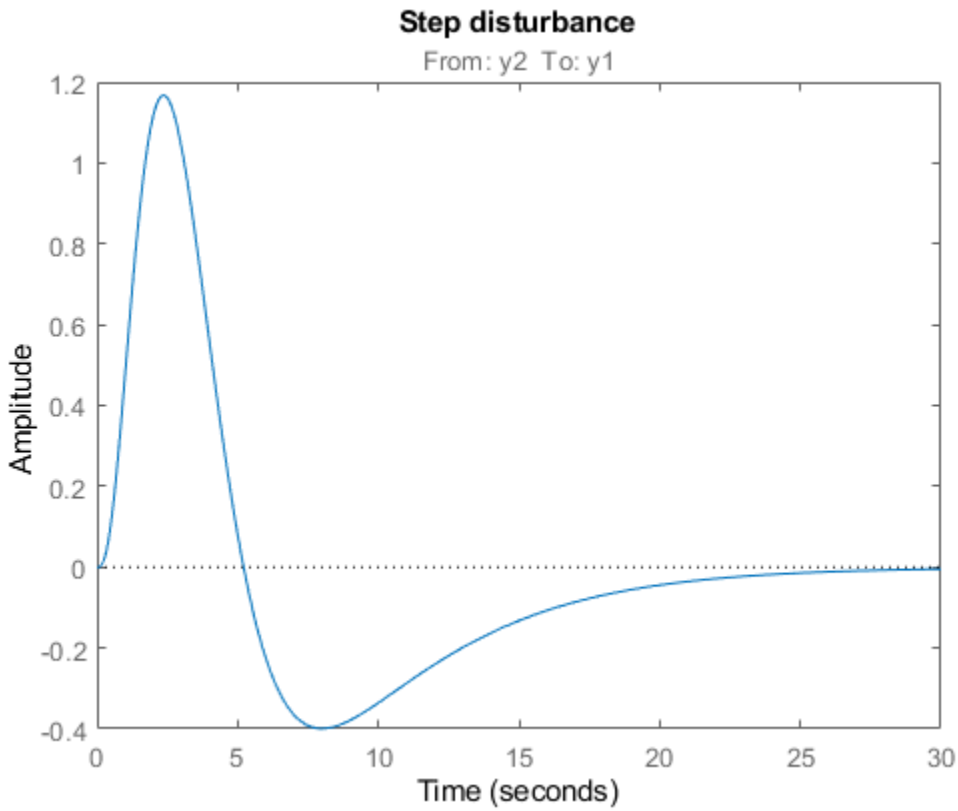
```
Final: Soft = 0.86, Hard = -Inf, Iterations = 114
```

As before, use `getIOTransfer` to compute and plot the tuned responses to a step command `r` and step disturbance entering at the location `y2`:

```
% Response to a step command
H = getIOTransfer(T,'r','y1');
clf, step(H,30), title('Step command')
```

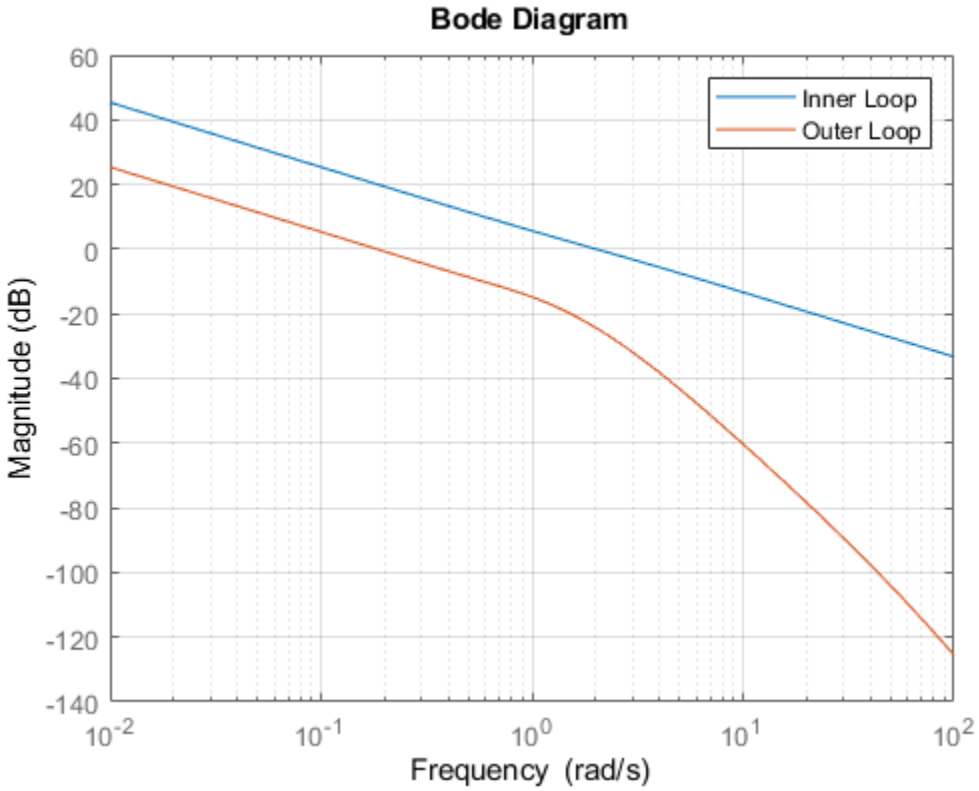


```
% Response to a step disturbance
H = getIOTransfer(T,'y2','y1');
step(H,30), title('Step disturbance')
```



You can also plot the open-loop gains for the inner and outer loops to validate the bandwidth requirements. Note the -1 sign to compute the negative-feedback open-loop transfer:

```
L1 = getLoopTransfer(T,'y1',-1);       % crossover should be at .2
L2 = getLoopTransfer(T,'y2',-1,'y1');  % crossover should be at 2
bodemag(L2,L1,{1e-2,1e2}), grid
legend('Inner Loop','Outer Loop')
```



See Also

`slTuner` | `systune (slTuner)`

Related Examples

- “PID Tuning for Setpoint Tracking vs. Disturbance Rejection”

PID Tuning for Setpoint Tracking vs. Disturbance Rejection

This example uses `systune` to explore trade-offs between setpoint tracking and disturbance rejection when tuning PID controllers.

PID Tuning Trade-Offs

When tuning 1-DOF PID controllers, it is often impossible to achieve good tracking and fast disturbance rejection at the same time. Assuming the control bandwidth is fixed, faster disturbance rejection requires more gain inside the bandwidth, which can only be achieved by increasing the slope near the crossover frequency. Because a larger slope means a smaller phase margin, this typically comes at the expense of more overshoot in the response to setpoint changes.

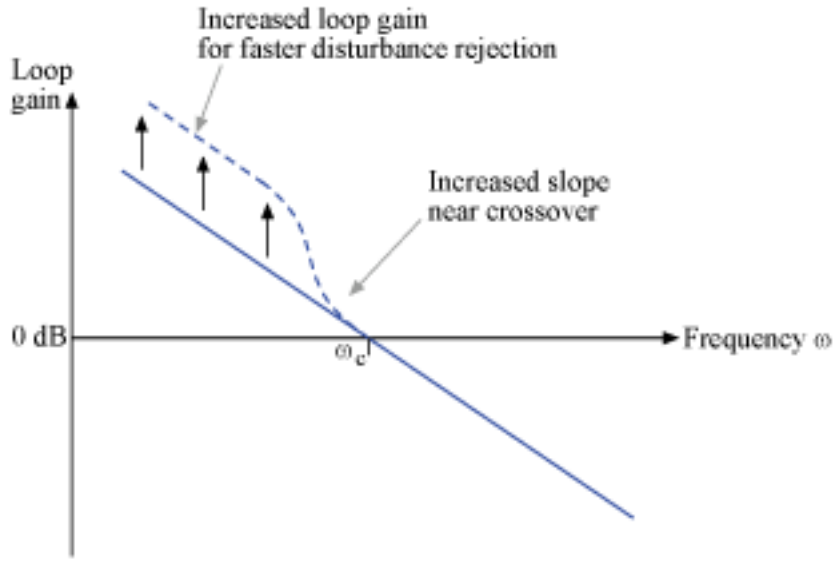


Figure 1: Trade-off in 1-DOF PID Tuning.

This example uses `systune` to explore this trade-off and find the right compromise for your application. See also `pidtool` for a more direct way to make such trade-off (see "Design Focus" under Controller Options).

Tuning Setup

Consider the PID loop of Figure 2 with a load disturbance at the plant input.

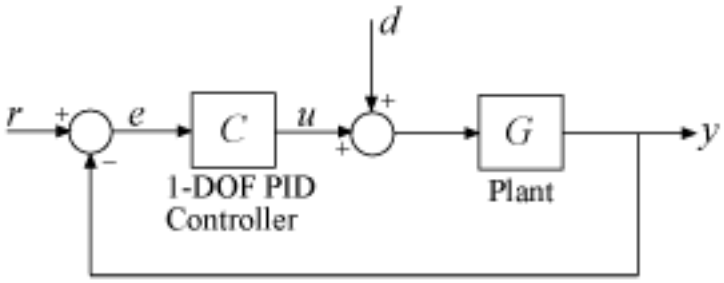


Figure 2: PID Control Loop.

For this example we use the plant model

$$G(s) = \frac{10(s+5)}{(s+1)(s+2)(s+10)}.$$

The target control bandwidth is 10 rad/s. Create a tunable PID controller and fix its derivative filter time constant to $T_f = 0.01$ (10 times the bandwidth) so that there are only three gains to tune (proportional, integral, and derivative gains).

```
G = zpk(-5,[-1 -2 -10],10);
C = tunablePID('C','pid');
C.Tf.Value = 0.01;  C.Tf.Free = false;  % fix Tf=0.01
```

Construct a tunable model `T0` of the closed-loop transfer from `r` to `y`. Use an "analysis point" block to mark the location `u` where the disturbance enters.

```
LS = AnalysisPoint('u');
T0 = feedback(G*LS*C,1);
T0.u = 'r'; T0.y = 'y';
```

The gain of the open-loop response $L = GC$ is a key indicator of the feedback loop behavior. The open-loop gain should be high (greater than one) inside the control bandwidth to ensure good disturbance rejection, and should be low (less than one) outside the control bandwidth to be insensitive to measurement noise and unmodeled plant dynamics. Accordingly, use three requirements to express the control objectives:

- "Tracking" requirement to specify a response time of about 0.2 seconds to step changes in `r`
- "MaxLoopGain" requirement to force a roll-off of -20 dB/decade past the crossover frequency 10 rad/s
- "MinLoopGain" requirement to adjust the integral gain at frequencies below 0.1 rad/s.

```
s = tf('s');
wc = 10; % target crossover frequency

% Tracking
R1 = TuningGoal.Tracking('r','y',2/wc);

% Bandwidth and roll-off
R2 = TuningGoal.MaxLoopGain('u',wc/s);

% Disturbance rejection
R3 = TuningGoal.MinLoopGain('u',wc/s);
R3.Focus = [0 0.1];
```

Tuning of 1-DOF PID Controller

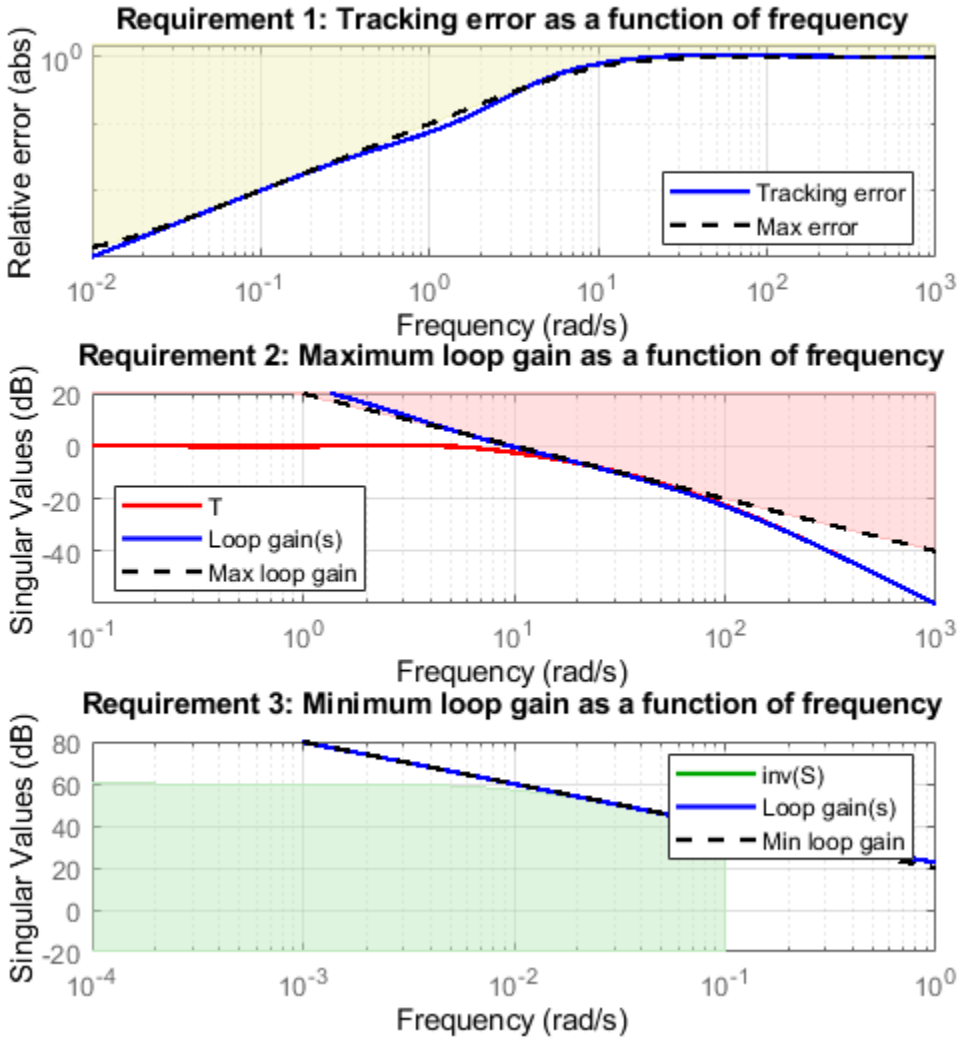
Use `systune` to tune the PID gains to meet these requirements. Treat the bandwidth and disturbance rejection goals as hard constraints and optimize tracking subject to these constraints.

```
T1 = systune(T0,R1,[R2 R3]);
```

```
Final: Soft = 1.12, Hard = 0.9998, Iterations = 158
```

Verify that all three requirements are nearly met. The blue curves are the achieved values and the yellow patches highlight regions where the requirements are violated.

```
figure('Position',[100,100,560,580])
viewGoal([R1 R2 R3],T1)
```



Tracking vs. Rejection

To gain insight into the trade-off between tracking and disturbance rejection, increase the minimum loop gain in the frequency band [0,0.1] rad/s by a factor α. Re-tune the PID gains for the values $\alpha = 2, 4$.

```
% Increase loop gain by factor 2
alpha = 2;
R3.MinGain = alpha*wc/s;
T2 = systune(T0,R1,[R2 R3]);
```

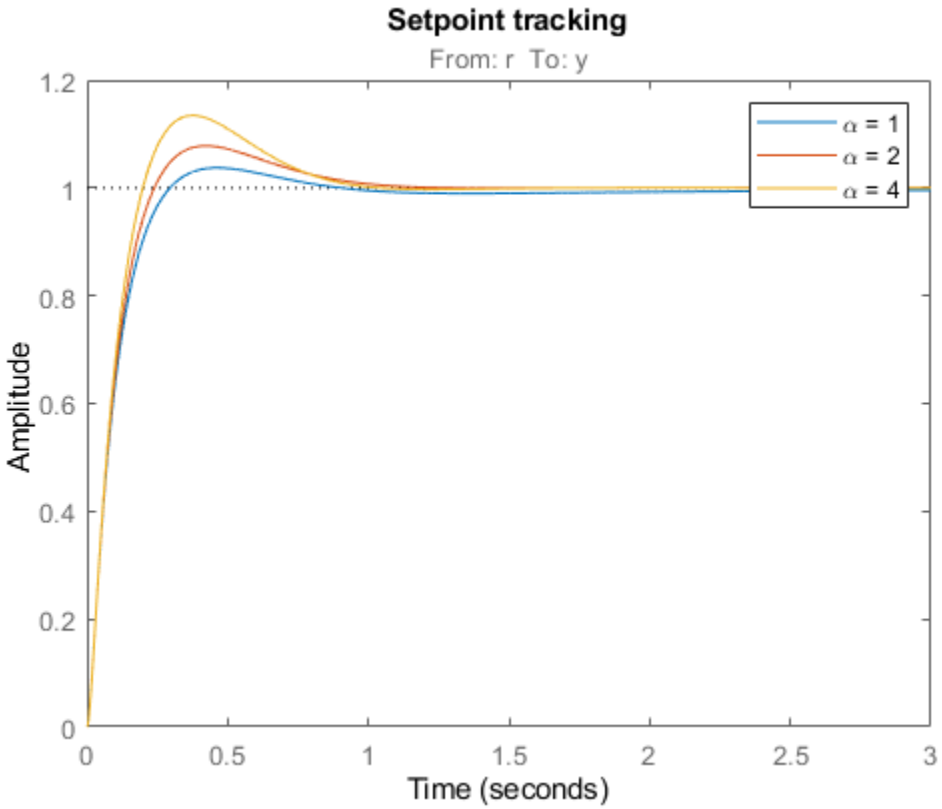
```
Final: Soft = 1.17, Hard = 0.99954, Iterations = 115
```

```
% Increase loop gain by factor 4
alpha = 4;
R3.MinGain = alpha*wc/s;
T3 = systune(T0,R1,[R2 R3]);
```

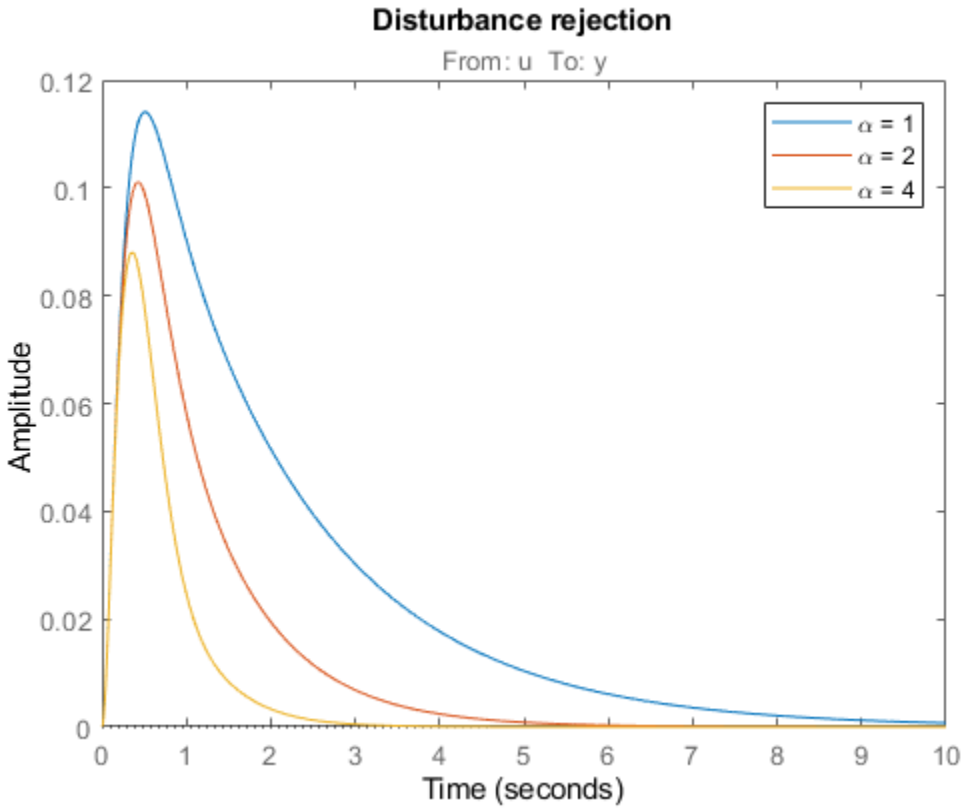
```
Final: Soft = 1.25, Hard = 0.99994, Iterations = 166
```

Compare the responses to a step command `r` and to a step disturbance `d` entering at the plant input `u`.

```
figure, step(T1,T2,T3,3)
title('Setpoint tracking')
legend('\alpha = 1','\alpha = 2','\alpha = 4')
```

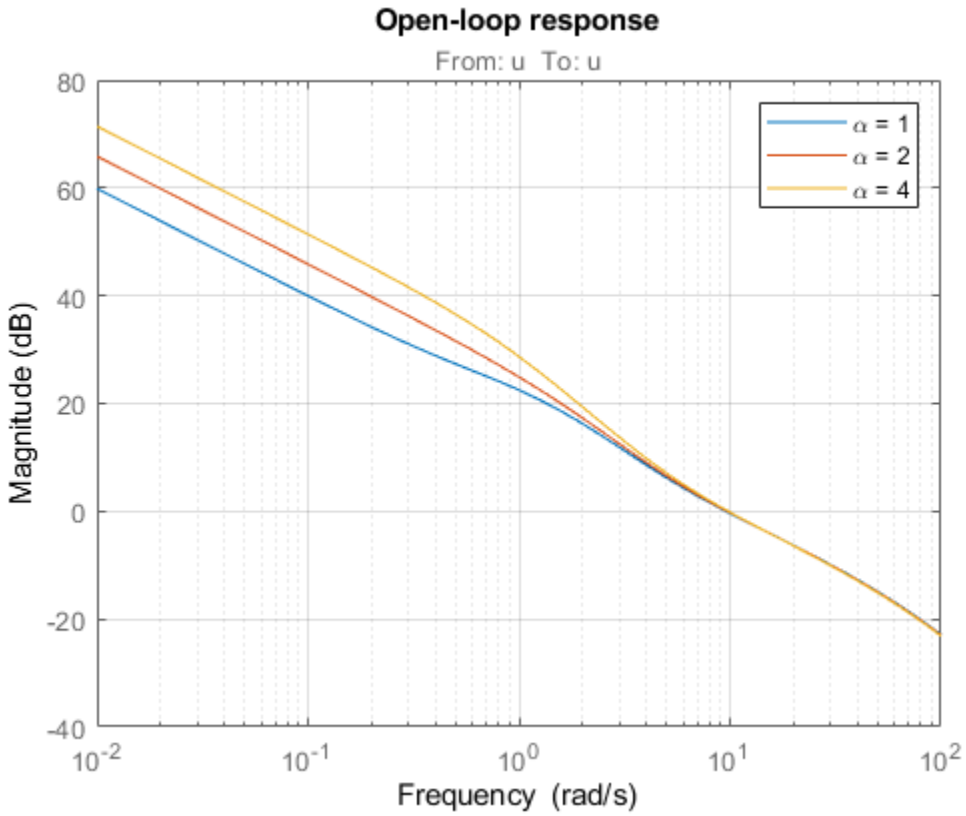



```
% Compute closed-loop transfer from u to y
D1 = getIOTransfer(T1,'u','y');
D2 = getIOTransfer(T2,'u','y');
D3 = getIOTransfer(T3,'u','y');
step(D1,D2,D3,10)
title('Disturbance rejection')
legend('\alpha = 1','\alpha = 2','\alpha = 4')
```



Note how disturbance rejection improves as `alpha` increases, but at the expense of increased overshoot in setpoint tracking. Plot the open-loop responses for the three designs, and note how the slope before crossover (0dB) increases with `alpha`.

```
L1 = getLoopTransfer(T1,'u');
L2 = getLoopTransfer(T2,'u');
L3 = getLoopTransfer(T3,'u');
bodemag(L1,L2,L3,{1e-2,1e2}), grid
title('Open-loop response')
legend('\alpha = 1','\alpha = 2','\alpha = 4')
```



Which design is most suitable depends on the primary purpose of the feedback loop you are tuning.

Tuning of 2-DOF PID Controller

If you cannot compromise tracking to improve disturbance rejection, consider using a 2-DOF architecture instead. A 2-DOF PID controller is capable of fast disturbance rejection without significant increase of overshoot in setpoint tracking.

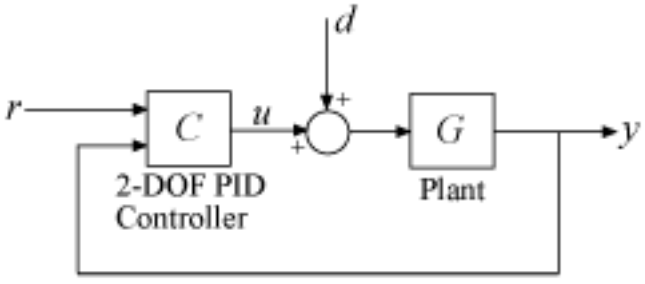


Figure 3: 2-DOF PID Control Loop.

Use the `tunablePID2` object to parameterize the 2-DOF PID controller and construct a tunable model `T0` of the closed-loop system in Figure 3.

```
C = tunablePID2('C','pid');
C.Tf.Value = 0.01;  C.Tf.Free = false;  % fix Tf=0.01
```

```
T0 = feedback(G*LS*C,1,2,1,+1);
T0 = T0(:,1);
T0.u = 'r';  T0.y = 'y';
```

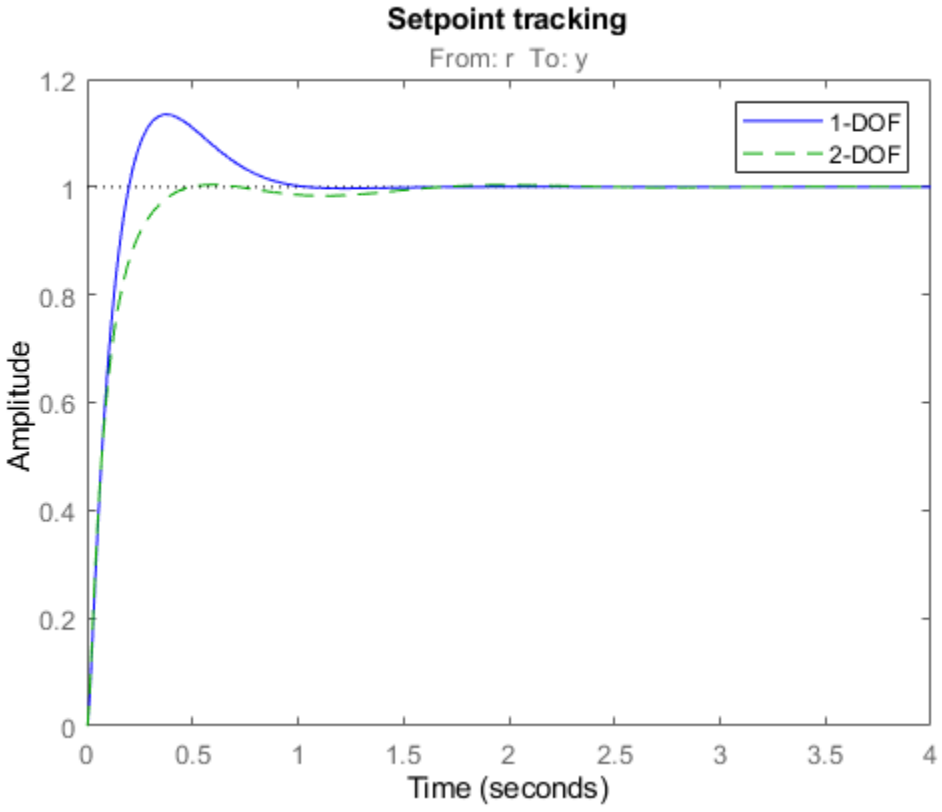
Next tune the 2-DOF PI controller for the largest loop gain tried earlier ($\alpha = 4$).

```
% Minimum loop gain inside bandwidth (for disturbance rejection)
alpha = 4;
R3.MinGain = alpha*wc/s;

% Tune 2-DOF PI controller
T4 = systune(T0,R1,[R2 R3]);
```

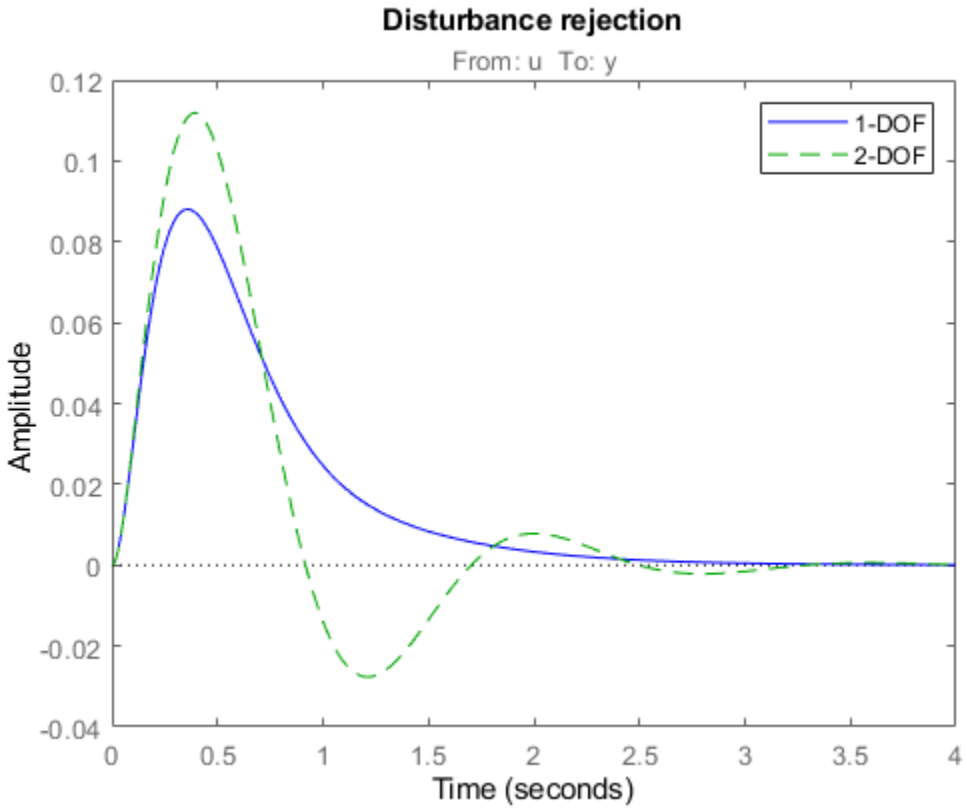
```
Final: Soft = 1.09, Hard = 0.85727, Iterations = 78
```

Compare the setpoint tracking and disturbance rejection properties of the 1-DOF and 2-DOF designs for $\alpha = 4$.

```
clf, step(T3,'b',T4,'g--',4)
title('Setpoint tracking')
legend('1-DOF','2-DOF')
```




```
D4 = getIOTransfer(T4,'u','y');
step(D3,'b',D4,'g--',4)
title('Disturbance rejection')
legend('1-DOF','2-DOF')
```



The responses to a step disturbance are similar but the 2-DOF controller eliminates the overshoot in the response to a setpoint change. You can use `showTunable` to compare the tuned gains in the 1-DOF and 2-DOF controllers.

```
showTunable(T3)  % 1-DOF PI
```

```
C =

             1            s
  Kp + Ki * --- + Kd * --------
             s          Tf*s+1

  with Kp = 9.51, Ki = 14.9, Kd = 0.89, Tf = 0.01

Name: C
Continuous-time PIDF controller in parallel form.
```

```
showTunable(T4)  % 2-DOF PI
```

```
C =

                       1                 s
  u = Kp (b*r-y) + Ki --- (r-y) + Kd -------- (c*r-y)
                       s               Tf*s+1

  with Kp = 5.71, Ki = 20.5, Kd = 0.839, Tf = 0.01, b = 0.681, c = 1.31
```

```
Name: C
Continuous-time 2-DOF PIDF controller in parallel form.
```

See Also

`systune`

Related Examples

- “Multi-Loop PI Control of a Robotic Arm” on page 13-114

Time-Domain Specifications

This example gives a tour of available time-domain requirements for control system tuning with `systune` or `looptune`.

Background

The `systune` and `looptune` commands tune the parameters of fixed-structure control systems subject to a variety of time- and frequency-domain requirements. The `TuningGoal` package is the repository for such design requirements.

Step Command Following

The `TuningGoal.StepTracking` requirement specifies how the tuned closed-loop system should respond to a step input. You can specify the desired response either in terms of first- or second-order characteristics, or as an explicit reference model. This requirement is satisfied when the relative gap between the actual and desired responses is small enough in the least-squares sense. For example,

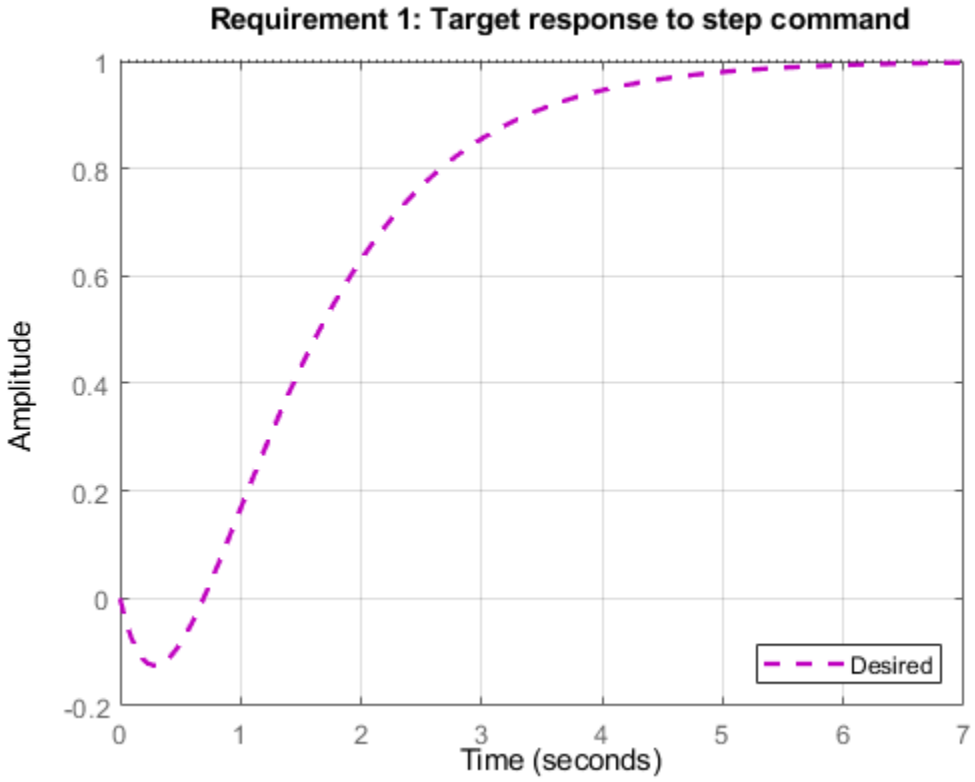
```
R1 = TuningGoal.StepTracking('r','y',0.5);
```

stipulates that the closed-loop response from `r` to `y` should behave like a first-order system with time constant 0.5, while

```
R2 = TuningGoal.StepTracking('r','y',zpk(2,[-1 -2],-1));
```

specifies a second-order, non-minimum-phase behavior. Use `viewGoal` to visualize the desired response.

```
viewGoal(R2)
```



This requirement can be used to tune both SISO and MIMO step responses. In the MIMO case, the requirement ensures that each output tracks the corresponding input with minimum cross-couplings.

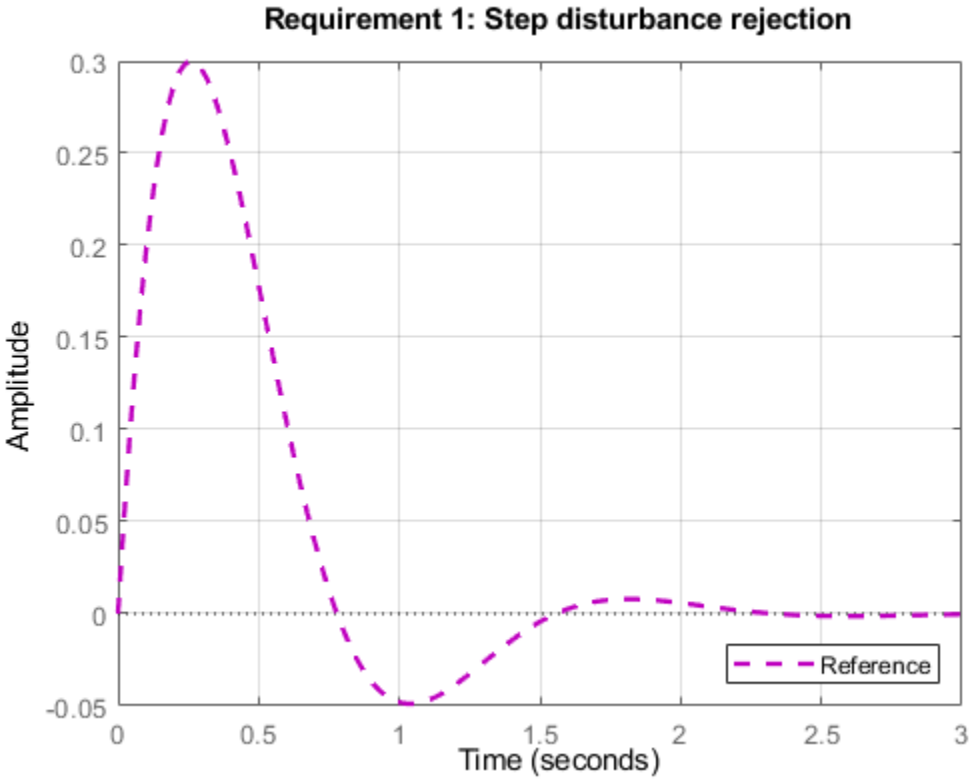
Step Disturbance Rejection

The `TuningGoal.StepRejection` requirement specifies how the tuned closed-loop system should respond to a step disturbance. You can specify worst-case values for the response amplitude, settling time, and damping of oscillations. For example,

```
R1 = TuningGoal.StepRejection('d','y',0.3,2,0.5);
```

limits the amplitude of $y(t)$ to 0.3, the settling time to 2 time units, and the damping ratio to a minimum of 0.5. Use `viewGoal` to see the corresponding time response.

```
viewGoal(R1)
```



You can also use a "reference model" to specify the desired response. Note that the actual and specified responses may differ substantially when better disturbance rejection is possible. Use the `TuningGoal.Transient` requirement when a close match is desired. For best results, adjust the gain of the reference model so that the actual and specified responses have similar peak amplitudes (see `TuningGoal.StepRejection` documentation for details).

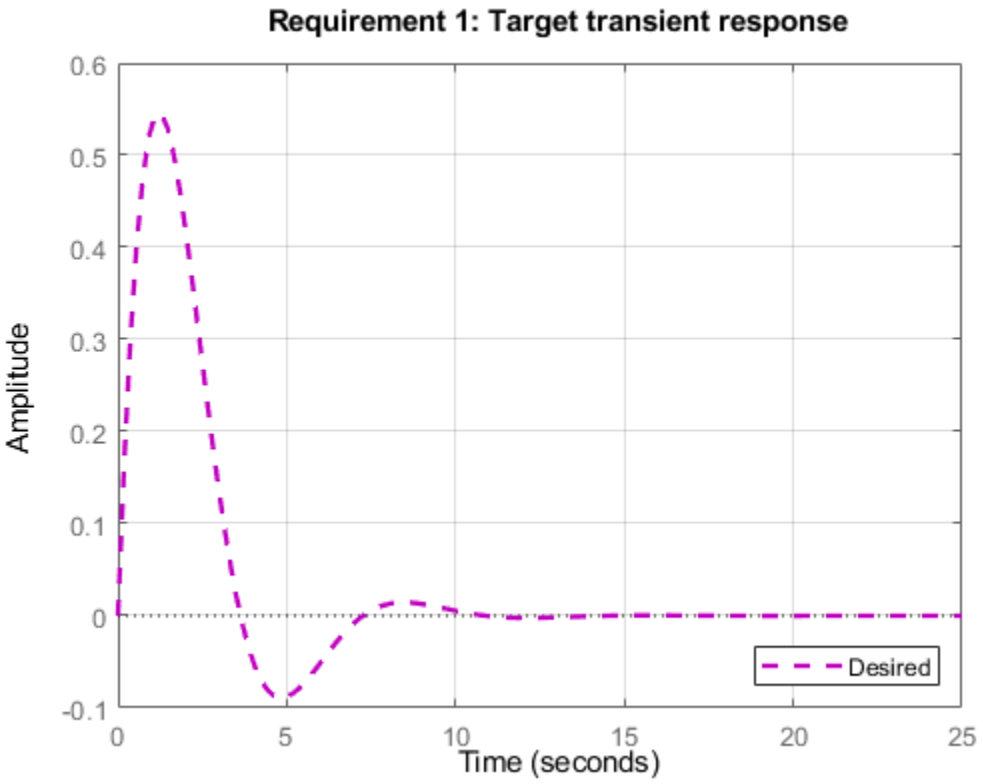
Transient Response Matching

The `TuningGoal.Transient` requirement specifies the transient response for a specific input signal. This is a generalization of the `TuningGoal.StepTracking` requirement. For example,

```
R1 = TuningGoal.Transient('r','y',tf(1,[1 1 1]),'impulse');
```

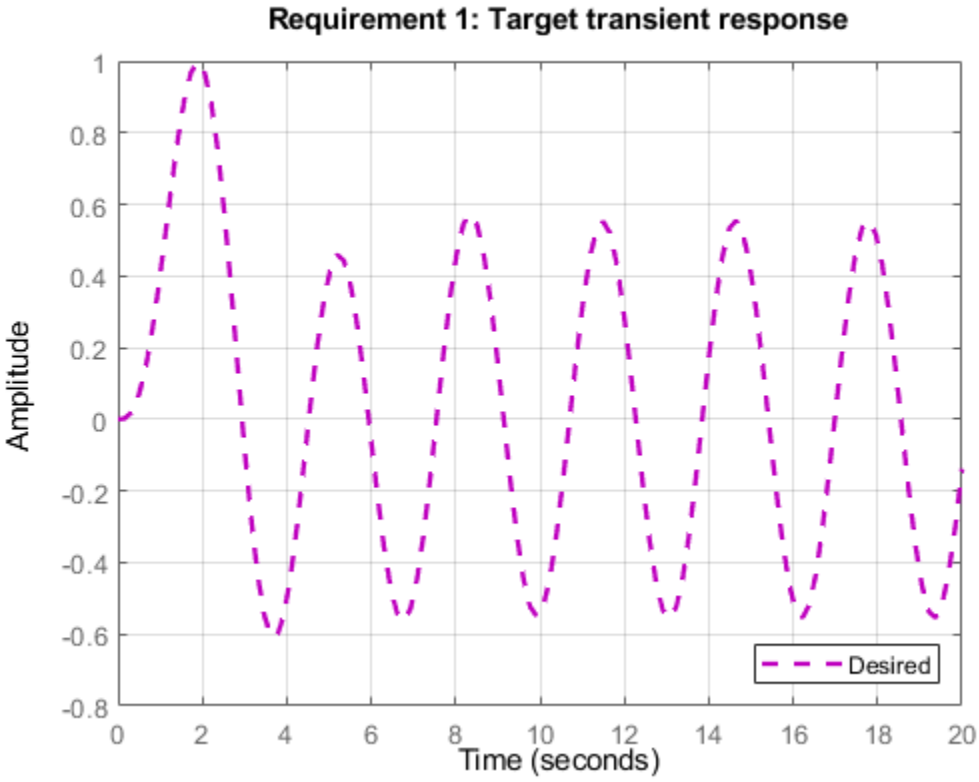
requires that the tuned response from r to y look like the impulse response of the reference model $1/(s^2+s+1)$.

```
viewGoal(R1)
```



The input signal can be an impulse, a step, a ramp, or a more general signal modeled as the impulse response of some input shaping filter. For example, a sine wave with frequency ω_0 can be modeled as the impulse response of $\omega_0^2/(s^2 + \omega_0^2)$.

```
w0 = 2;
F = tf(w0^2,[1 0 w0^2]);  % input shaping filter
R2 = TuningGoal.Transient('r','y',tf(1,[1 1 1]),F);
viewGoal(R2)
```



LQG Design

Use the `TuningGoal.LQG` requirement to create a linear-quadratic-Gaussian objective for tuning the control system parameters. This objective is applicable to any control structure, not just the classical observer structure of LQG control. For example, consider the simple PID loop of Figure 2 where d and n are unit-variance disturbance and noise inputs, and S_d and S_n are lowpass and highpass filters that model the disturbance and noise spectral contents.

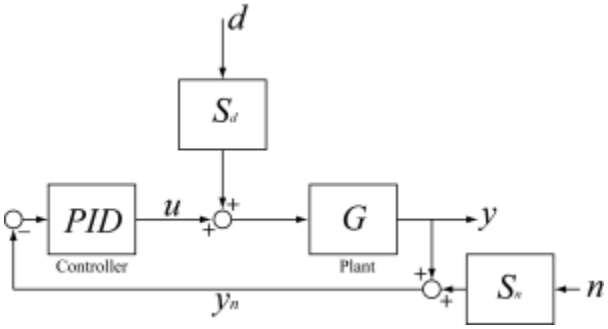


Figure 2: Regulation loop.

To regulate y around zero, you can use the following LQG criterion:

$$J = lim_{T \to \infty} E\left(\frac{1}{T}\int_0^T (y^2(t) + 0.05u^2)dt\right)$$

The first term in the integral penalizes the deviation of $y(t)$ from zero, and the second term penalizes the control effort. Using `systune`, you can tune the PID controller to minimize the cost J. To do this, use the LQG requirement

```
Qyu = diag([1 0.05]);  % weighting of y^2 and u^2
R4 = TuningGoal.LQG({'d','n'},{'y','u'},1,Qyu);
```

See Also

`TuningGoal.StepTracking` | `TuningGoal.StepRejection` | `TuningGoal.Transient` | `TuningGoal.LQG`

Related Examples

- “Frequency-Domain Specifications”

Frequency-Domain Specifications

This example gives a tour of available frequency-domain requirements for control system tuning with `systune` or `looptune`.

Background

The `systune` and `looptune` commands tune the parameters of fixed-structure control systems subject to a variety of time- and frequency-domain requirements. The `TuningGoal` package is the repository for such design requirements.

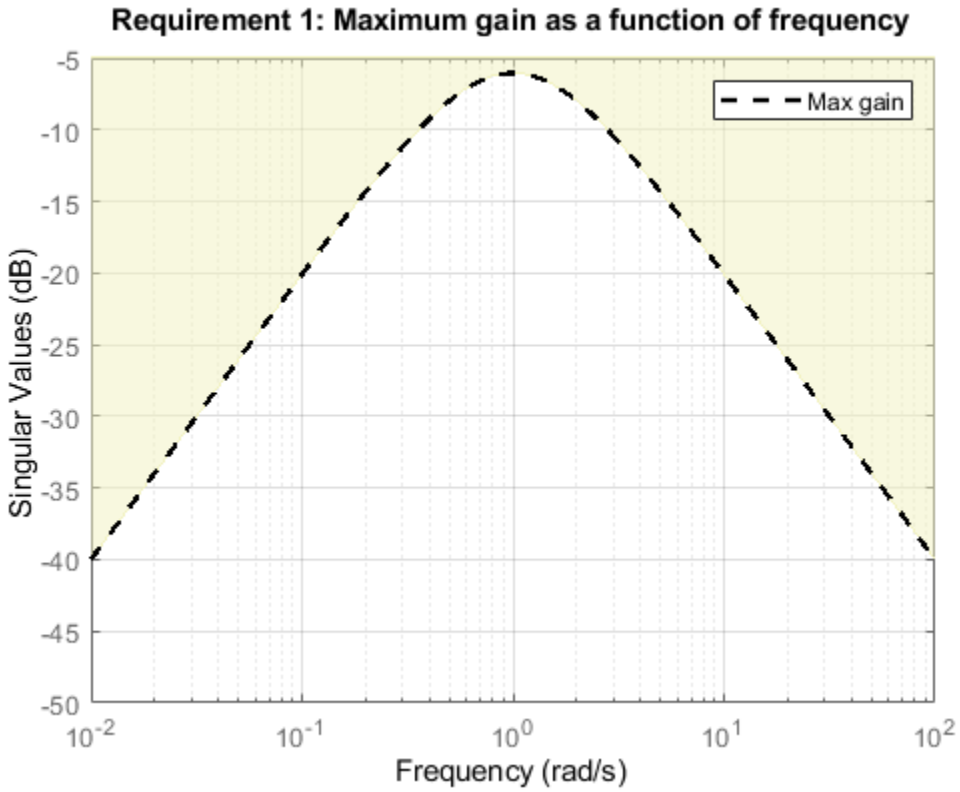
Gain Limit

The `TuningGoal.Gain` requirement enforces gain limits on SISO or MIMO closed-loop transfer functions. This requirement is useful to enforce adequate disturbance rejection and roll off, limit sensitivity and control effort, and prevent saturation. For MIMO transfer functions, "gain" refers to the largest singular value of the frequency response matrix. The gain limit can be frequency dependent. For example

```
s = tf('s');
R1 = TuningGoal.Gain('d','y',s/(s+1)^2);
```

specifies that the gain from `d` to `y` should not exceed the magnitude of the transfer function $s/(s+1)^2$.

```
viewGoal(R1)
```

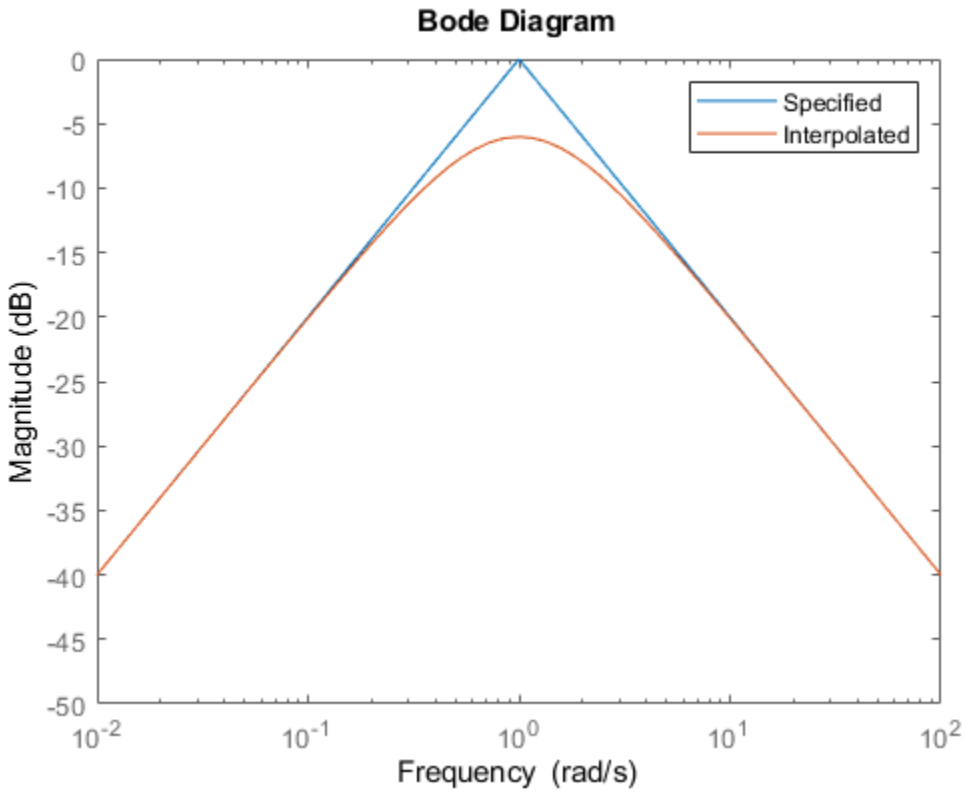


It is often convenient to just sketch the asymptotes of the desired gain profile. For example, instead of the transfer function $s/(s+1)^2$, we could just specify gain values of 0.01,1,0.01 at the frequencies 0.01,1,100, the point (1,1) being the breakpoint of the two asymptotes s and $1/s$.

```
Asymptotes = frd([0.01,1,0.01],[0.01,1,100]);
R2 = TuningGoal.Gain('d','y',Asymptotes);
```

The requirement object automatically turns this discrete gain profile into a gain limit defined at all frequencies.

```
bodemag(Asymptotes,R2.MaxGain)
legend('Specified','Interpolated')
```



Variance Amplification

The `TuningGoal.Variance` requirement limits the noise variance amplification from specified inputs to specified outputs. In technical terms, this requirement constrains the H_2 norm of a closed-loop transfer function. This requirement is preferable to `TuningGoal.Gain` when the input signals are random processes and the average gain matters more than the peak gain. For example,

```
R = TuningGoal.Variance('n','y',0.1);
```

limits the output variance of `y` to 0.1^2 for a unit-variance white-noise input `n`.

Reference Tracking and Overshoot Reduction

The `TuningGoal.Tracking` requirement enforces reference tracking and loop decoupling objectives in the frequency domain. For example

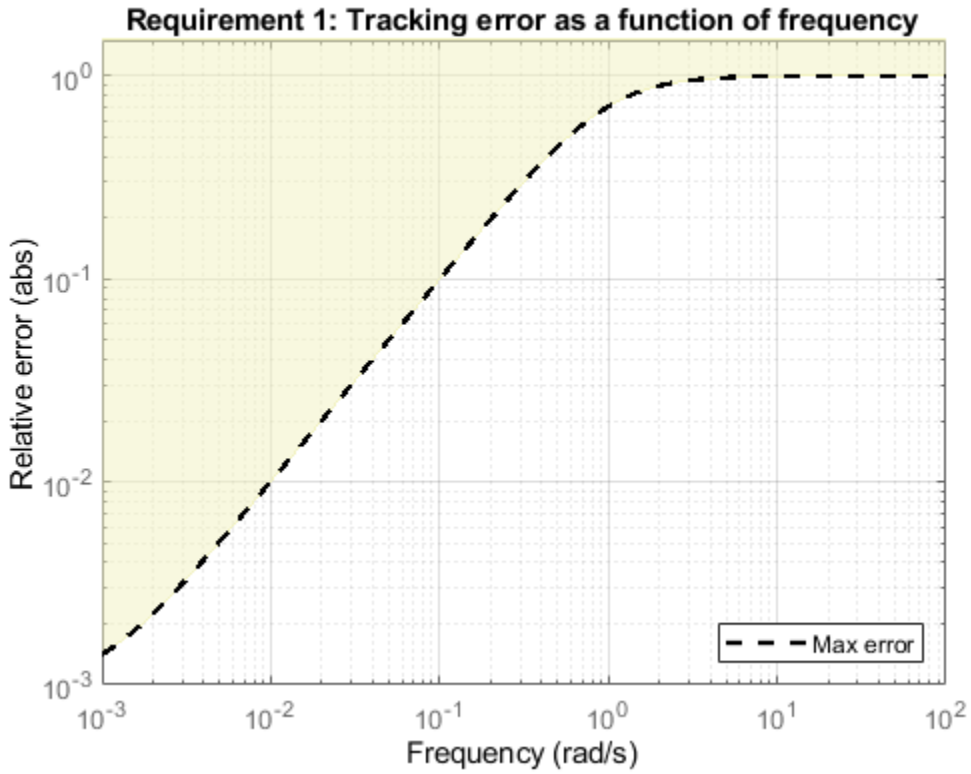
```
R1 = TuningGoal.Tracking('r','y',2);
```

specifies that the output `y` should track the reference `r` with a two-second response time. Similarly

```
R2 = TuningGoal.Tracking({'Vsp','wsp'},{'V','w'},2);
```

specifies that `V` should track `Vsp` and `w` should track `wsp` with minimum cross-coupling between the two responses. Tracking requirements are converted into frequency-domain constraints on the tracking error as a function of frequency. For the first requirement `R1`, for example, the gain from `r` to the tracking error `e = r-y` should be small at low frequency and approach 1 (100%) at frequencies greater than 1 rad/s (bandwidth for a two-second response time). You can use `viewGoal` to visualize this frequency-domain constraint. Note that the yellow region indicates where the requirement is violated.

```
viewGoal(R1)
```



If the response has excessive overshoot, use the `TuningGoal.Overshoot` requirement in conjunction with the `TuningGoal.Tracking` requirement. For example, you can limit the overshoot from `r` to `y` to 10% using

```
R3 = TuningGoal.Overshoot('r','y',10);
```

Disturbance Rejection

In feedback loops such as the one shown in Figure 1, the open- and closed-loop responses from disturbance d to output y are related by

$$G_{CL}(s) = \frac{G_{OL}(s)}{1 + L(s)}$$

where $L(s)$ is the loop transfer function measured at the disturbance entry point. The gain of $1 + L$ is the disturbance attenuation factor, the ratio between the open- and closed-loop sensitivities to the disturbance. Its reciprocal $S = 1/(1 + L)$ is the sensitivity at the disturbance input.

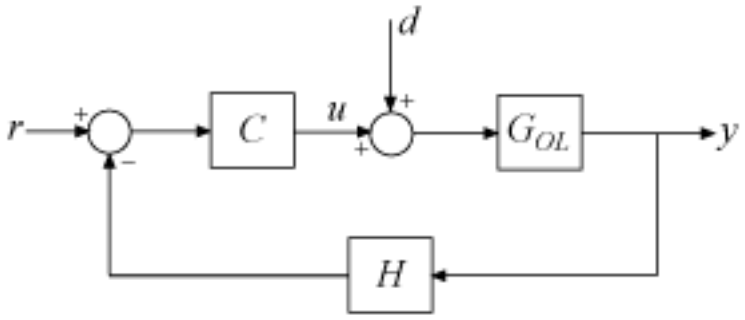


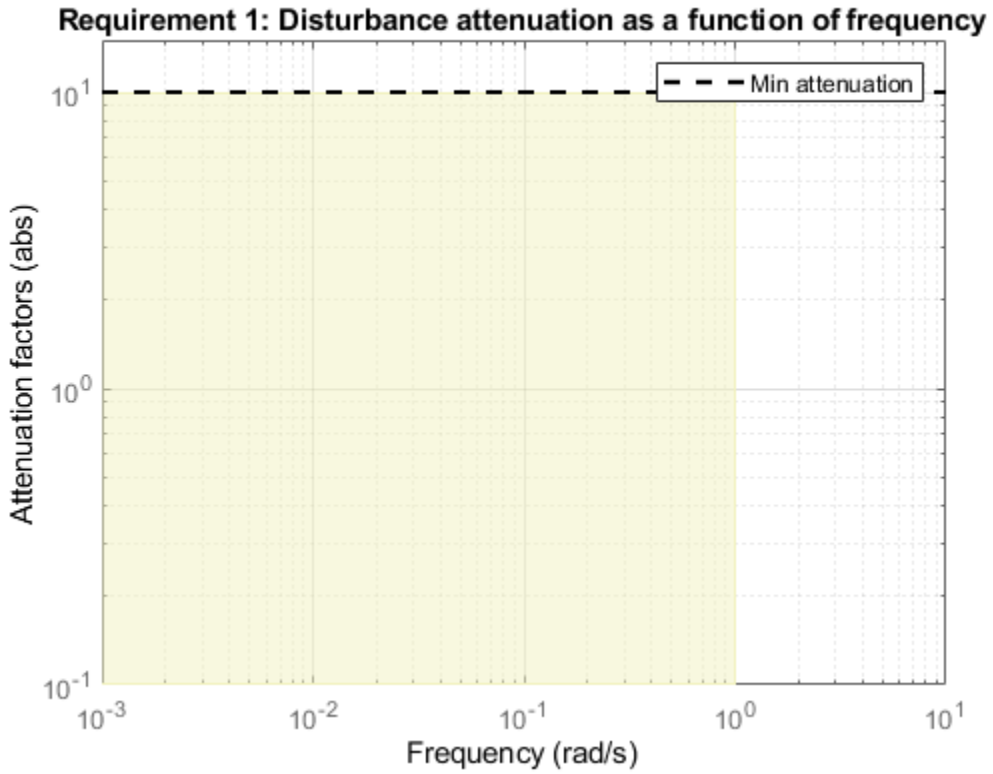
Figure 1: Sample feedback loop.

The `TuningGoal.Rejection` requirement specifies the disturbance attenuation as a function of frequency. The attenuation factor is greater than one inside the control bandwidth since feedback control reduces the impact of disturbances. As a rule of thumb, a 10-times-larger attenuation requires a 10-times-larger loop gain. For example

```
R1 = TuningGoal.Rejection('u',10);
R1.Focus = [0 1];
```

specifies that a disturbance entering at the plant input "u" should be attenuated by a factor 10 in the frequency band from 0 to 1 rad/s.

```
viewGoal(R1)
```

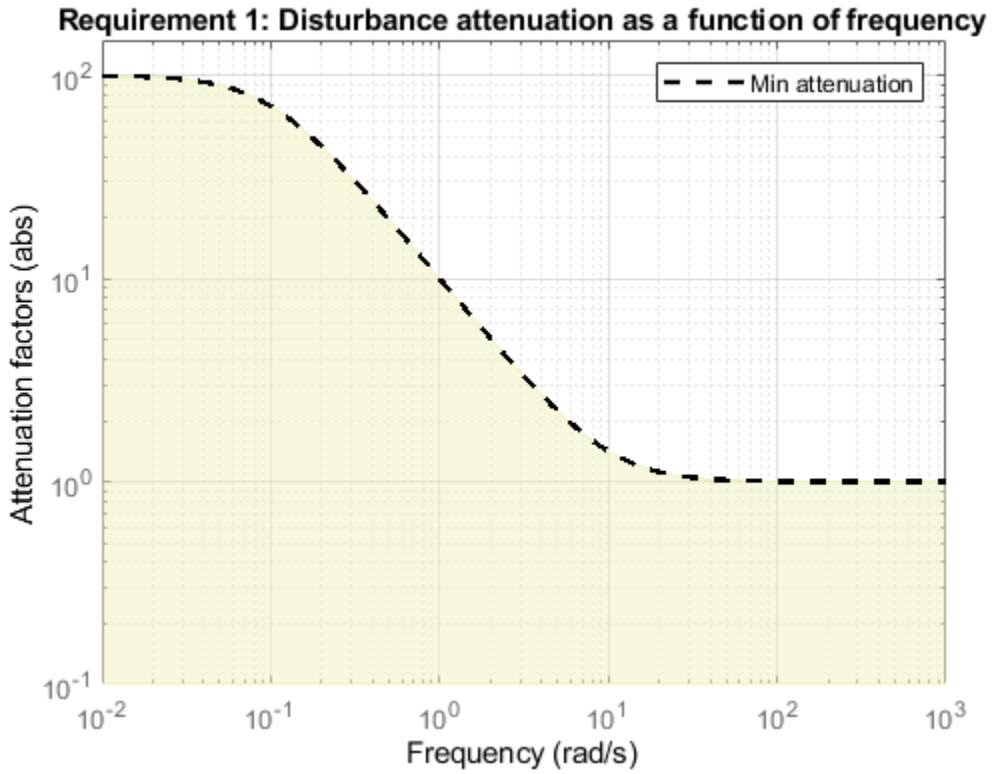


More generally, you can specify a frequency-dependent attenuation profile, for example

```
s = tf('s');
R2 = TuningGoal.Rejection('u',(s+10)/(s+0.1));
```

specifies an attenuation factor of 100 below 0.1 rad/s gradually decreasing to 1 (no attenuation) after 10 rad/s.

```
viewGoal(R2)
```

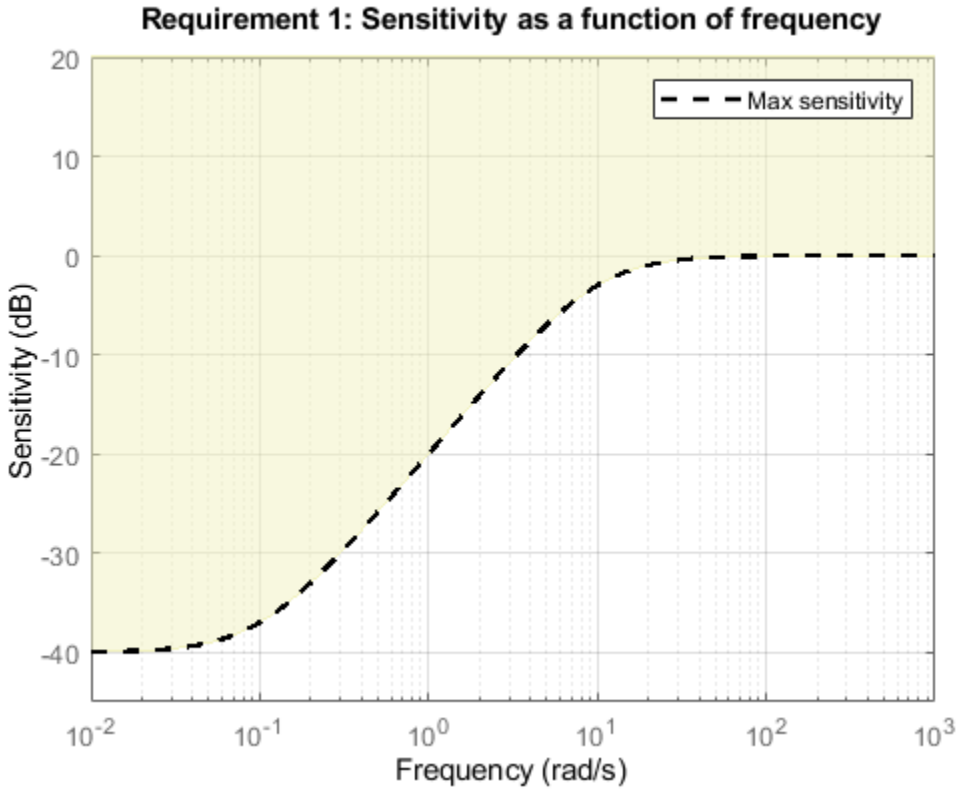


Instead of specifying the minimum attenuation, you can use the `TuningGoal.Sensitivity` requirement to specify the maximum sensitivity, that is, the maximum gain of $S = 1/(1+L)$. For example,

```
R3 = TuningGoal.Sensitivity('u',(s+0.1)/(s+10));
```

is equivalent to the rejection requirement R2 above. The sensitivity increases from 0.01 (1%) below 0.1 rad/s to 1 (100%) above 10 rad/s.

```
viewGoal(R3)
```



Frequency-Weighted Gain and Variance

The `TuningGoal.WeightedGain` and `TuningGoal.WeightedVariance` requirements are generalizations of the `TuningGoal.Gain` and `TuningGoal.Variance` requirements. These requirements constrain the H_∞ or H_2 norm of a frequency-weighted closed-loop transfer function $W_L(s)T(s)W_R(s)$, where $W_L(s)$ and $W_R(s)$ are user-defined weighting functions. For example

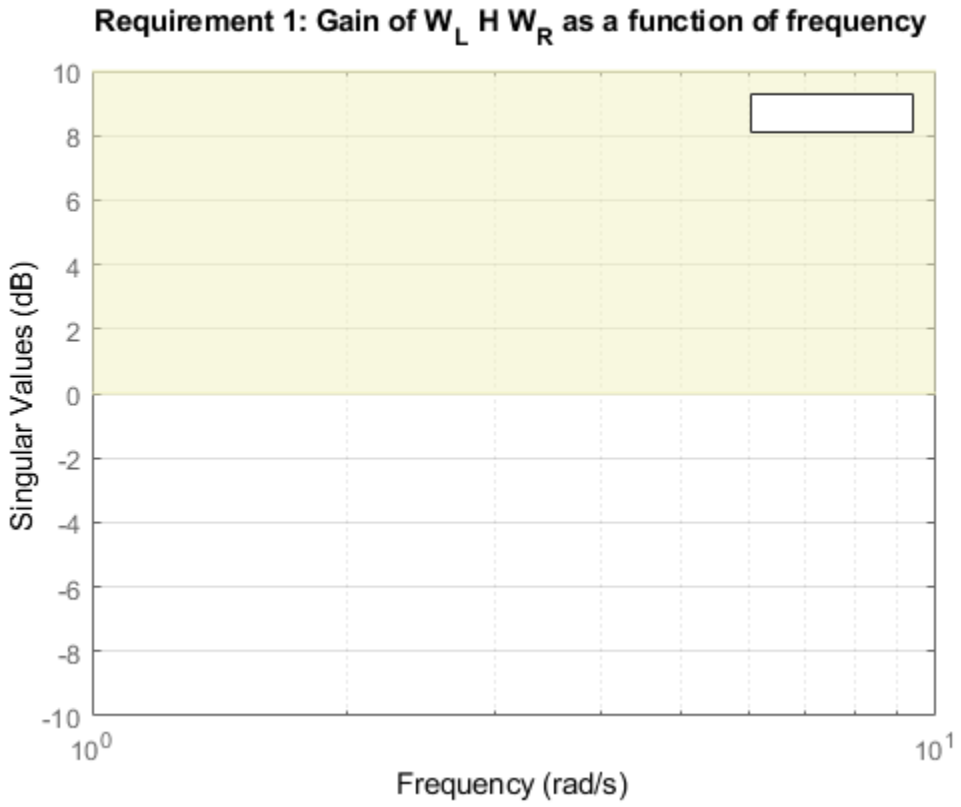
```
WL = blkdiag(1/(s+0.001),s/(0.001*s+1));
WR = [];
R = TuningGoal.WeightedGain('r',{'e','y'},WL,[]);
```

specifies the constraint

$$\left\| \begin{pmatrix} \frac{1}{s+0.001} T_{re} \\ \frac{s}{0.001s+1} T_{ry} \end{pmatrix} \right\|_\infty < 1$$

Note that this is a normalized gain constraint (unit bound across frequency).

```
viewGoal(R)
```



See Also

`TuningGoal.Gain` | `TuningGoal.Variance` | `TuningGoal.Tracking` | `TuningGoal.Overshoot` | `TuningGoal.Rejection` | `TuningGoal.Sensitivity` | `TuningGoal.WeightedGain` | `TuningGoal.WeightedVariance`

Related Examples

- “Time-Domain Specifications”
- “Loop Shape and Stability Margin Specifications”

Loop Shape and Stability Margin Specifications

This example shows how to specify loop shapes and stability margins when tuning control systems with `systune` or `looptune`.

Background

The `systune` and `looptune` commands tune the parameters of fixed-structure control systems subject to a variety of time- and frequency-domain requirements. The `TuningGoal` package is the repository for such design requirements.

Loop Shape

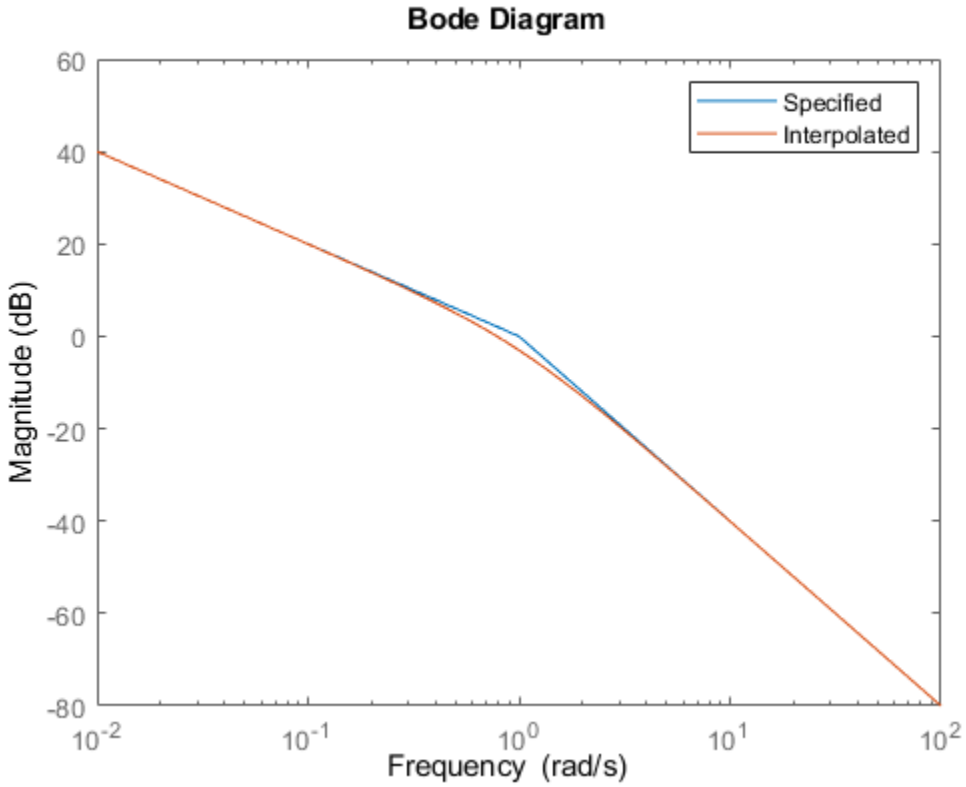
The `TuningGoal.LoopShape` requirement is used to shape the open-loop response gain(s), a design approach known as *loop shaping*. For example,

```
s = tf('s');
R1 = TuningGoal.LoopShape('u',1/s);
```

specifies that the open-loop response measured at the location "u" should look like a pure integrator (as far as its gain is concerned). In MATLAB, use an `AnalysisPoint` block to mark the location "u", see the "*Building Tunable Models*" example for details. In Simulink, use the `addPoint` method of the `slTuner` interface to mark "u" as a point of interest.

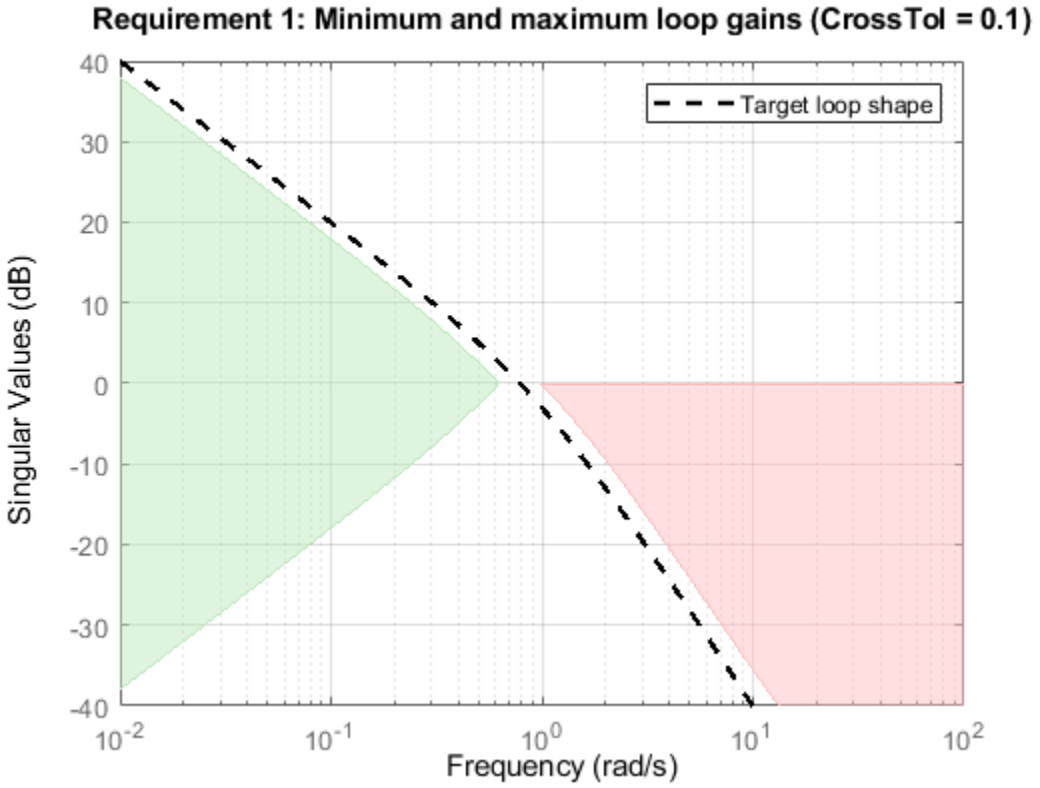
As with other gain specifications, you can just specify the asymptotes of the desired loop shape using a few frequency points. For example, to specify a loop shape with gain crossover at 1 rad/s, -20 dB/decade slope before 1 rad/s, and -40 dB/decade slope after 1 rad/s, just specify that the gain at the frequencies 0.1,1,10 should be 10,1,0.01, respectively.

```
LS = frd([10,1,0.01],[0.1,1,10]);
R2 = TuningGoal.LoopShape('u',LS);

bodemag(LS,R2.LoopGain)
legend('Specified','Interpolated')
```



Loop shape requirements are constraints on the open-loop response L. For tuning purposes, they are converted into closed-loop gain constraints on the sensitivity function $S = 1/(1 + L)$ and complementary sensitivity function $T = L/(1 + L)$. Use `viewGoal` to visualize the target loop shape and corresponding gain bounds on S (green) and T (red).

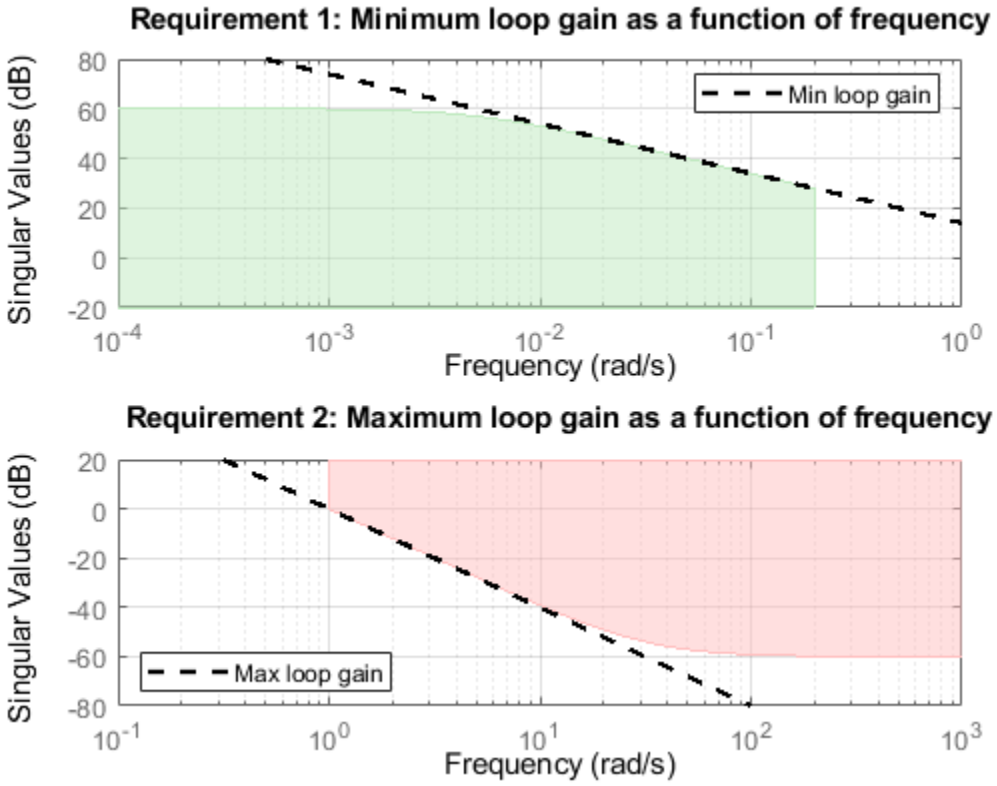
```
viewGoal(R2)
```



Minimum and Maximum Loop Gain

Instead of `TuningGoal.LoopShape`, you can use `TuningGoal.MinLoopGain` and `TuningGoal.MaxLoopGain` to specify minimum or maximum values for the loop gain in a particular frequency band. This is useful when the actual loop shape near crossover is best left to the tuning algorithm to figure out. For example, the following requirements specify the minimum loop gain inside the bandwidth and the roll-off characteristics outside the bandwidth, but do not specify the actual crossover frequency nor the loop shape near crossover.

```
MinLG = TuningGoal.MinLoopGain('u',5/s);  % integral action
MinLG.Focus = [0 0.2];

MaxLG = TuningGoal.MaxLoopGain('u',1/s^2);  % -40dB/decade roll off
MaxLG.Focus = [1 Inf];

viewGoal([MinLG MaxLG])
```



The `TuningGoal.MaxLoopGain` requirement rests on the fact that the open- and closed-loop gains are comparable when the loop gain is small ($|L| \ll 1$). As a result, it can be ineffective at keeping the loop gain below some value close to 1. For example, suppose that flexible modes cause gain spikes beyond the crossover frequency and that you need to keep these spikes below 0.5 (-6 dB). Instead of using `TuningGoal.MaxLoopGain`, you can directly constrain the gain of L using `TuningGoal.Gain` with a loop opening at "u".

```
MaxLG = TuningGoal.Gain('u','u',0.5);
MaxLG.Opening = 'u';
```

If the open-loop response is unstable, make sure to further disable the implicit stability constraint associated with this requirement.

```
MaxLG.Stabilize = false;
```

Figure 1 shows this requirement evaluated for an open-loop response with flexible modes.

Figure 1: Gain constraint on L

Stability Margins

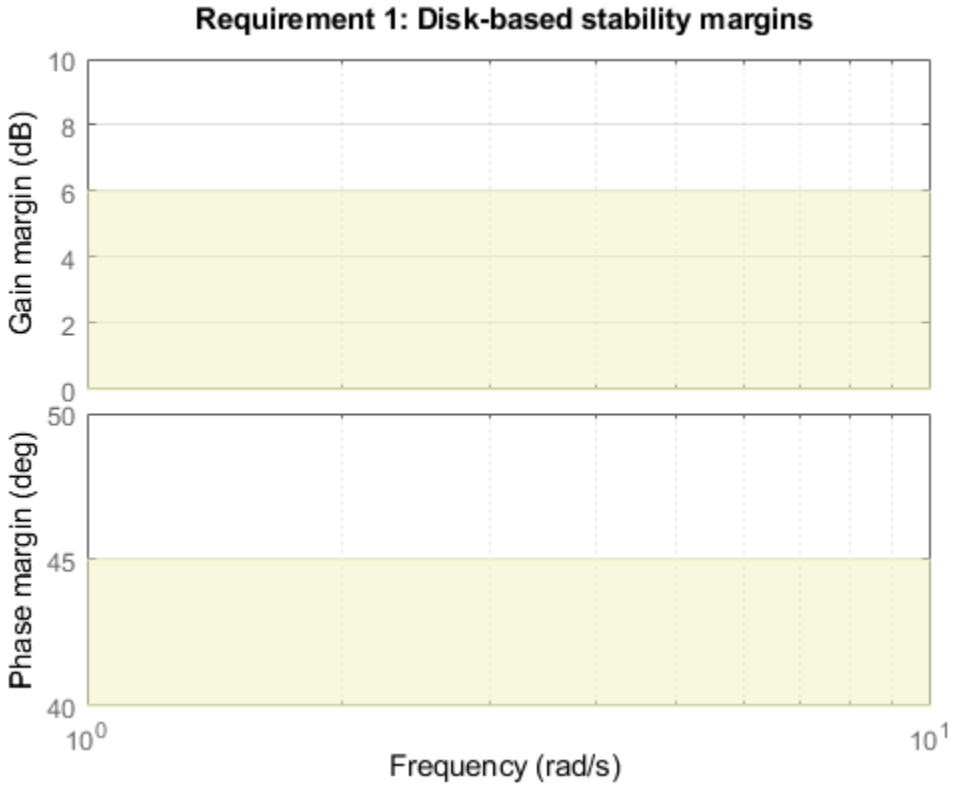
The `TuningGoal.Margins` requirement uses the notion of disk margin to enforce minimum amounts of gain and phase margins at the specified loop opening site(s). For MIMO feedback loops, this requirement guarantees stability for gain or phase variations in each feedback channel. The gain or phase can change in all channels simultaneously, and by a different amount in each channel. See “Stability Margins in Control System Tuning” for details. For example,the following code enforces ±6 dB of gain margin and 45 degrees of phase margin at a location "u".

```
R = TuningGoal.Margins('u',6,45);
```

In MATLAB, use an `AnalysisPoint` block to mark the location "u" (see “Building Tunable Models” for details). In Simulink, use the `addPoint` method of the `slTuner` interface to mark "u" as a point of interest (see “Create and Configure slTuner Interface to Simulink Model” on page 10-161). Stability margins are typically measured at the plant inputs or plant outputs or both.

The target gain and phase margin values are converted into a normalized gain constraint on some appropriate closed-loop transfer function. The desired margins are achieved at frequencies where the gain is less than 1. Use `viewGoal` to examine the requirement you have configured.

```
viewGoal(R)
```



The shaded region indicates where the constraint is violated. After tuning, for a tuned model `T`, you can use `viewGoal(R,T)` to see the tuned frequency-dependent margins on this plot.

See Also

`TuningGoal.MinLoopGain` | `TuningGoal.MaxLoopGain` | `TuningGoal.LoopShape` | `TuningGoal.Margins`

Related Examples

- “Stability Margins in Control System Tuning”
- “Frequency-Domain Specifications”

System Dynamics Specifications

This example shows how to constrain the poles of a control system tuned with `systune` or `looptune`.

Background

The `systune` and `looptune` commands tune the parameters of fixed-structure control systems subject to a variety of time- and frequency-domain requirements. The `TuningGoal` package is the repository for such design requirements.

Closed-Loop Poles

The `TuningGoal.Poles` goal constrains the location of the closed-loop poles. You can enforce some minimum decay rate

$$Re(s) < -\alpha,$$

impose some minimum damping ratio

$$Re(s) < -\zeta|s|,$$

or constrain the pole magnitude to

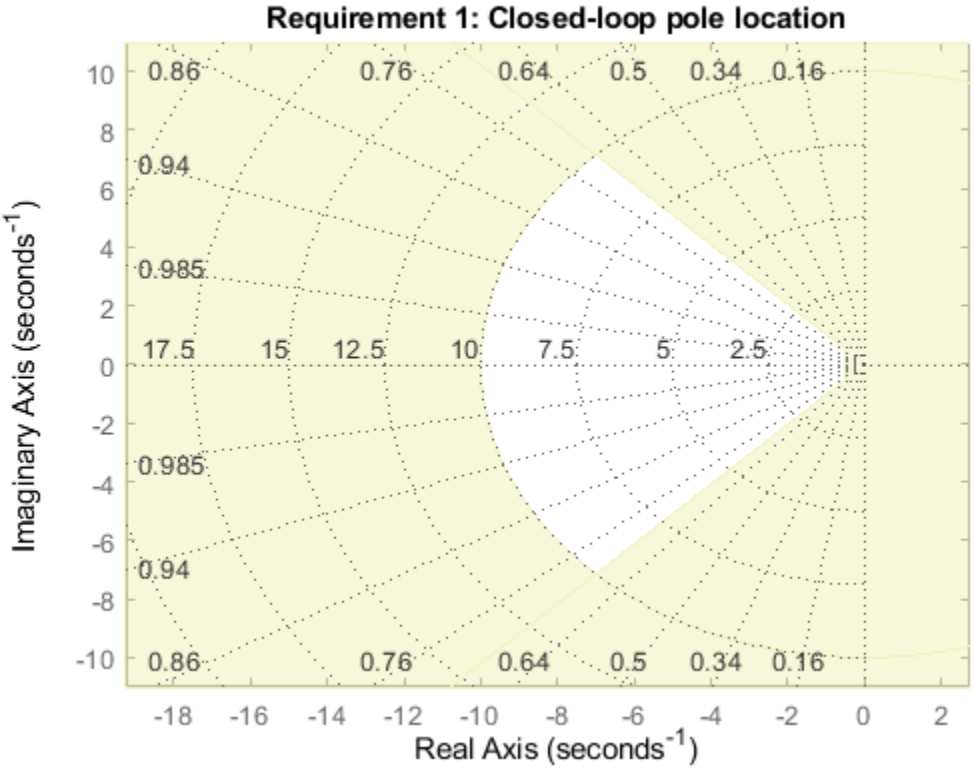
$$|s| < \omega_{\max}.$$

For example

```
MinDecay = 0.5;
MinDamping = 0.7;
MaxFrequency = 10;
R = TuningGoal.Poles(MinDecay,MinDamping,MaxFrequency);
```

constrains the closed-loop poles to lie in the white region below.

```
viewGoal(R)
```



Increasing the `MinDecay` value results in faster transients. Increasing the `MinDamping` value results in better damped transients. Decreasing the `MaxFrequency` value prevents fast dynamics.

Controller Poles

The `TuningGoal.ControllerPoles` goal constrains the pole locations for tuned elements such as filters and compensators. The tuning algorithm may produce unstable compensators for unstable plants. To prevent this, use the `TuningGoal.ControllerPoles` goal to keep the compensator poles in the left-half plane. For example, if your compensator is parameterized as a second-order transfer function,

```
C = tunableTF('C',1,2);
```

you can force it to have stable dynamics by adding the requirement

```
MinDecay = 0;
R = TuningGoal.ControllerPoles('C',MinDecay);
```

See Also

`TuningGoal.Poles` | `TuningGoal.ControllerPoles`

Related Examples

- “Loop Shape and Stability Margin Specifications”

Configuring Design Requirements

This example shows how to configure additional attributes of design requirements for use with `systune` or `looptune`.

All `TuningGoal` requirements are objects that can be further configured by modifying their default attributes. The display shows the list of such attributes. For example

```
R = TuningGoal.Gain('d','y',1)
```

```
R =
  Gain with properties:

         MaxGain: [1x1 zpk]
           Focus: [0 Inf]
       Stabilize: 1
    InputScaling: []
   OutputScaling: []
           Input: {'d'}
          Output: {'y'}
          Models: NaN
        Openings: {0x1 cell}
            Name: ''
```

Three attributes are shared by multiple requirements. The `Focus` property specifies the frequency band in which the requirement is active. For example,

```
R.Focus = [1 20];
```

limits the gain from `d` to `y` in the frequency interval [1,20] only. The `Models` property specifies which models the requirement applies to (in the context of tuning for multiple plant models). For example,

```
R.Models = [2 3 5];
```

indicates that the requirement only applies to the second, third, and fifth model in the model array supplied to `systune`. Finally, the `Openings` property lets you specify additional loop openings. For example

```
R = TuningGoal.Margins('Inner',6,45);
R.Openings = 'Outer';
```

specifies stability margins for the inner loop with the outer loop open. In MATLAB, use `AnalysisPoint` blocks to mark loop opening locations. In Simulink, use the `addPoint` method of the `slTuner` interface to flag such locations.

Validating Results

This example shows how to interpret and validate tuning results from `systune`.

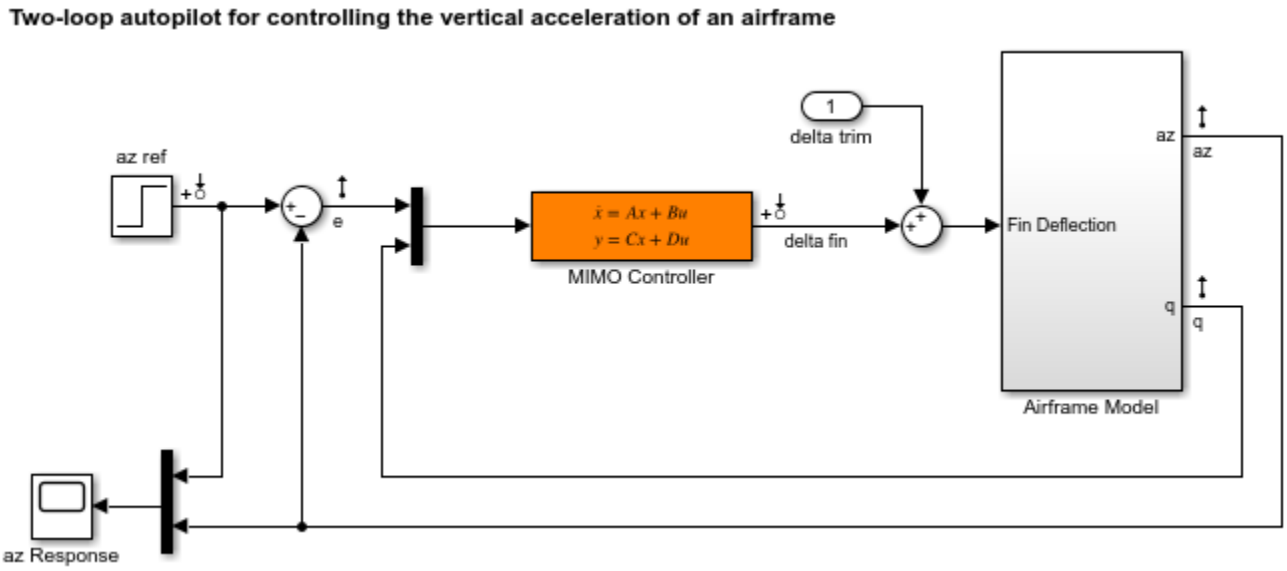
Background

You can tune the parameters of your control system with `systune` or `looptune`. The design specifications are captured using `TuningGoal` requirement objects. This example shows how to interpret the results from `systune`, graphically verify the design requirements, and perform additional open- and closed-loop analysis.

Controller Tuning with SYSTUNE

We use an autopilot tuning application as illustration, see the "*Tuning of a Two-Loop Autopilot*" example for details. The tuned compensator is the "MIMO Controller" block highlighted in orange in the model below.

```
open_system('rct_airframe2')
```



The setup and tuning steps are repeated below for completeness.

```
ST0 = slTuner('rct_airframe2','MIMO Controller');

% Compensator parameterization
C0 = tunableSS('C',2,1,2);
C0.D.Value(1) = 0;
C0.D.Free(1) = false;
setBlockParam(ST0,'MIMO Controller',C0)

% Requirements
Req1 = TuningGoal.Tracking('az ref','az',1);                  % tracking
Req2 = TuningGoal.Gain('delta fin','delta fin',tf(25,[1 0])); % roll-off
Req3 = TuningGoal.Margins('delta fin',7,45);                  % margins
MaxGain = frd([2 200 200],[0.02 2 200]);
```

```
Req4 = TuningGoal.Gain('delta fin','az',MaxGain);  % disturbance rejection

% Tuning
Opt = systuneOptions('RandomStart',3);
rng('default')
[ST1,fSoft] = systune(ST0,[Req1,Req2,Req3,Req4],Opt);

Final: Soft = 1.51, Hard = -Inf, Iterations = 52
Final: Soft = 1.15, Hard = -Inf, Iterations = 92
Final: Soft = 1.15, Hard = -Inf, Iterations = 74
Final: Soft = 1.16, Hard = -Inf, Iterations = 107
```

Interpreting Results

`systune` run three optimizations from three different starting points and returned the best overall result. The first output `ST` is an `slTuner` interface representing the tuned control system. The second output `fSoft` contains the final values of the four requirements for the best design.

```
fSoft
```

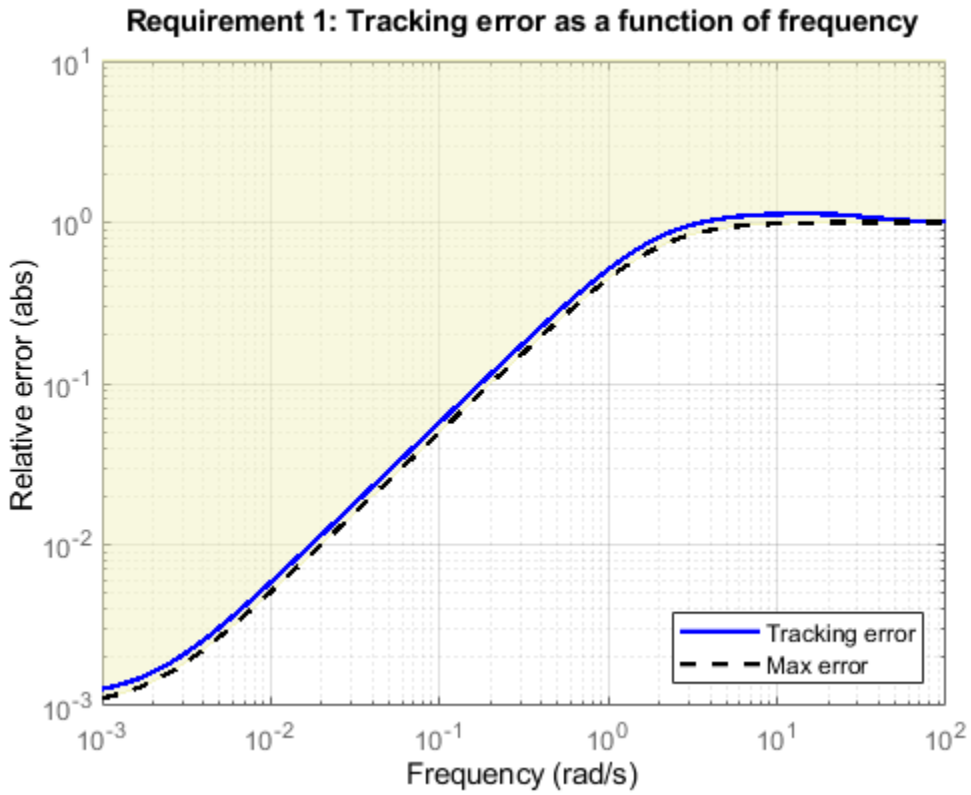
```
fSoft =

    1.1476    1.1476    0.5461    1.1476
```

Requirements are normalized so a requirement is satisfied if and only if its value is less than 1. Inspection of `fSoft` reveals that Requirements 1,2,4 are active and slightly violated while Requirement 3 (stability margins) is satisfied.

Verifying Requirements

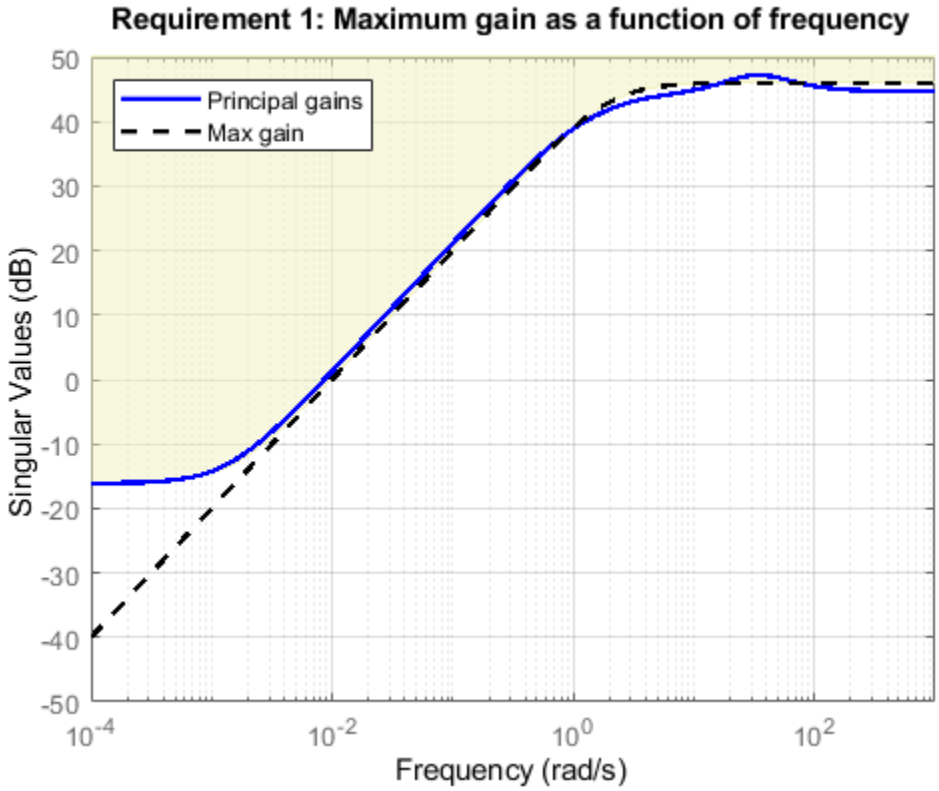
Use `viewGoal` to graphically inspect each requirement. This is useful to understand whether small violations are acceptable or what causes large violations. First verify the tracking requirement.

```
viewGoal(Req1,ST1)
```



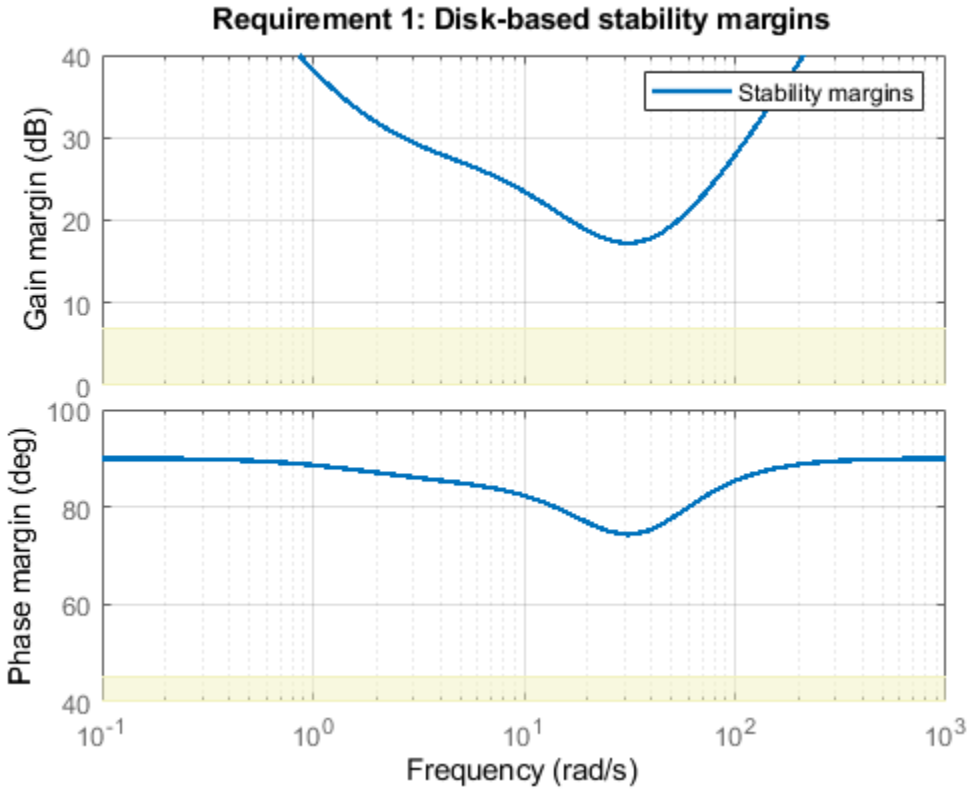
We observe a slight violation across frequency, suggesting that setpoint tracking will perform close to expectations. Similarly, verify the disturbance rejection requirement.

```
viewGoal(Req4,ST1)
legend('location','NorthWest')
```



Most of the violation is at low frequency with a small bump near 35 rad/s, suggesting possible damped oscillations at this frequency. Finally, verify the stability margin requirement.

```
viewGoal(Req3,ST1)
```



This requirement is satisfied at all frequencies, with the smallest margins achieved near the crossover frequency as expected.

Evaluating Requirements

You can also use `evalGoal` to evaluate each requirement, that is, compute its contribution to the soft and hard constraints. For example

```
[H1,f1] = evalGoal(Req1,ST1);
```

returns the value `f1` of the requirement and the underlying frequency-weighted transfer function `H1` used to computed it. You can verify that `f1` matches the first entry of `fSoft` and coincides with the peak gain of `H1`.

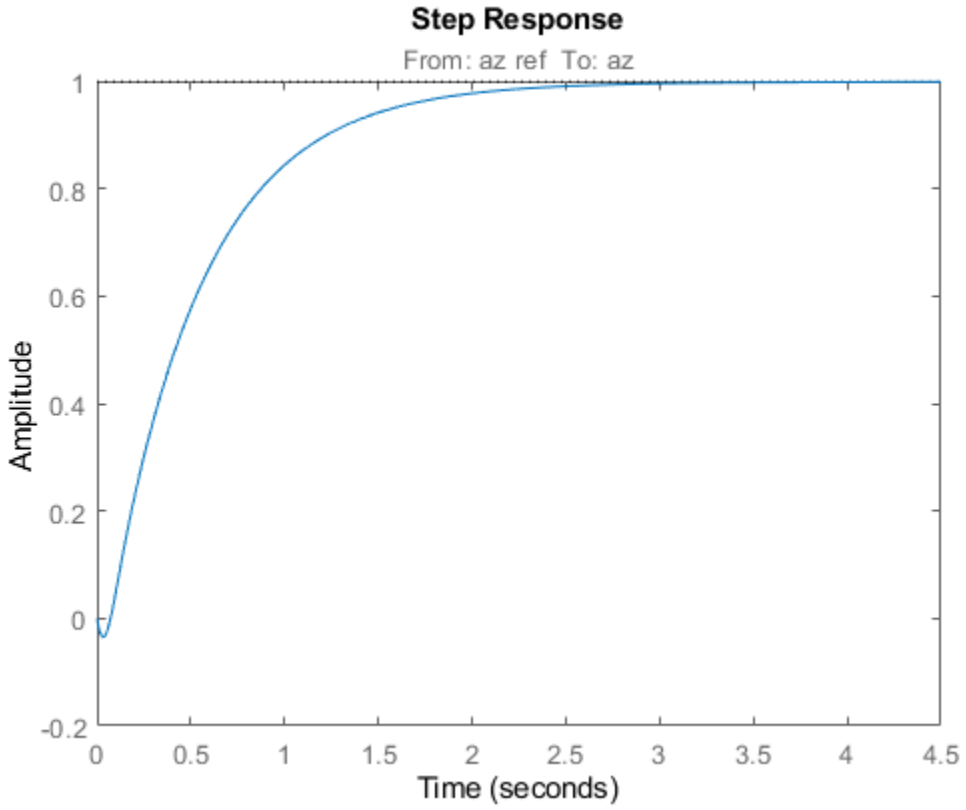
```
[f1 fSoft(1) getPeakGain(H1,1e-6)]
```

```
ans =

    1.1476    1.1476    1.1476
```

Analyzing System Responses

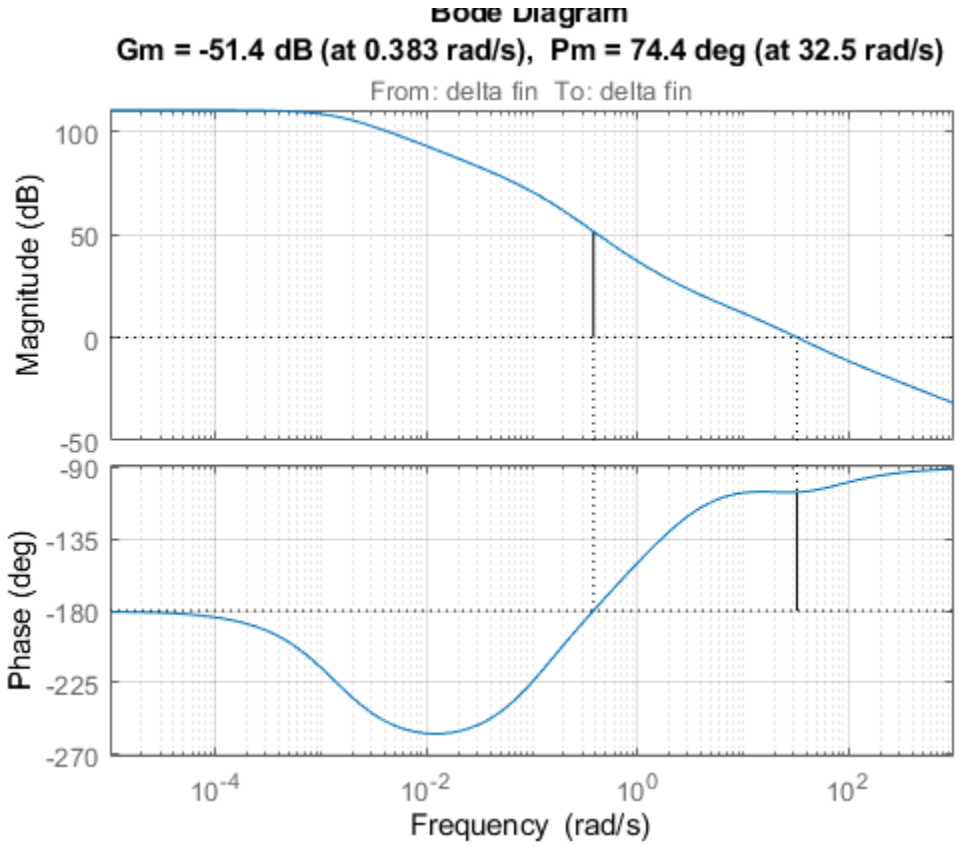
In addition to verifying requirements, you can perform basic open- and closed-loop analysis using `getIOTransfer` and `getLoopTransfer`. For example, verify tracking performance in the time domain by plotting the response `az` to a step command `azref` for the tuned system `ST1`.

```
T = ST1.getIOTransfer('az ref','az');
step(T)
```



Also plot the open-loop response measured at the plant input `delta fin`. You can use this plot to assess the classical gain and phase margins at the plant input.

```
L = ST1.getLoopTransfer('delta fin',-1);  % negative-feedback loop transfer
margin(L)
grid
```



Soft vs Hard Requirements

So far we have treated all four requirements equally in the objective function. Alternatively, you can use a mix of soft and hard constraints to differentiate between must-have and nice-to-have requirements. For example, you could treat Requirements 3,4 as hard constraints and optimize the first two requirements subject to these constraints. For best results, do this only after obtaining a reasonable design with all requirements treated equally.

```
[ST2,fSoft,gHard] = systune(ST1,[Req1 Req2],[Req3 Req4]);
```

```
Final: Soft = 1.29, Hard = 0.99995, Iterations = 168
```

```
fSoft
```

```
fSoft =

    1.2824    1.2870
```

```
gHard
```

```
gHard =

    0.4663    1.0000
```

Here `fSoft` contains the final values of the first two requirements (soft constraints) and `gHard` contains the final values of the last two requirements (hard constraints). The hard constraints are satisfied since all entries of `gHard` are less than 1. As expected, the best value of the first two requirements went up as the optimizer strived to strictly enforce the fourth requirement.

```
bdclose('all')
```

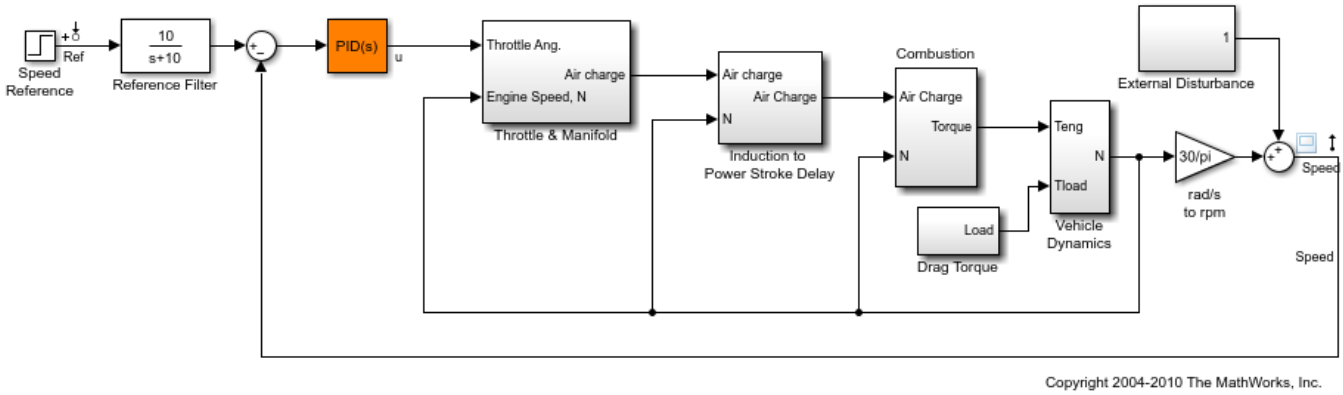
Tune Control Systems in Simulink

This example shows how to use `systune` or `looptune` to automatically tune control systems modeled in Simulink®.

Engine Speed Control

For this example we use the following model of an engine speed control system:

```
open_system('rct_engine_speed')
```



The control system consists of a single PID loop and the PID controller gains must be tuned to adequately respond to step changes in the desired speed. Specifically, we want the response to settle in less than 5 seconds with little or no overshoot. While `pidtune` is a faster alternative for tuning a single PID controller, this simple example is well suited for an introduction to the `systune` and `looptune` workflows in Simulink.

Controller Tuning with SYSTUNE

The `slTuner` interface provides a convenient gateway to `systune` for control systems modeled in Simulink. This interface lets you specify which blocks in the Simulink model are tunable and what signals are of interest for open- or closed-loop validation. Create an `slTuner` instance for the `rct_engine_speed` model and list the "PID Controller" block (highlighted in orange) as tunable. Note that all Linear Analysis points in the model (signals "Ref" and "Speed" here) are automatically available as points of interest for tuning.

```
ST0 = slTuner('rct_engine_speed','PID Controller');
```

The PID block is initialized with its value in the Simulink model, which you can access using `getBlockValue`. Note that the proportional and derivative gains are initialized to zero.

```
getBlockValue(ST0,'PID Controller')
```

```
ans =

        1
  Ki * ---
        s

  with Ki = 0.01
```

```
Name: PID_Controller
Continuous-time I-only controller.
```

Next create a step tracking requirement to capture the target settling time of 5 seconds. Use the signal names in the Simulink model to refer to the reference and output signals, and specify the target response as a first-order response with time constant of 1 second.

```
TrackReq = TuningGoal.StepTracking('Ref','Speed',1);
```

You can now tune the control system `ST0` subject to the requirement `TrackReq`.

```
ST1 = systune(ST0,TrackReq);
```

```
Final: Soft = 0.282, Hard = -Inf, Iterations = 67
```

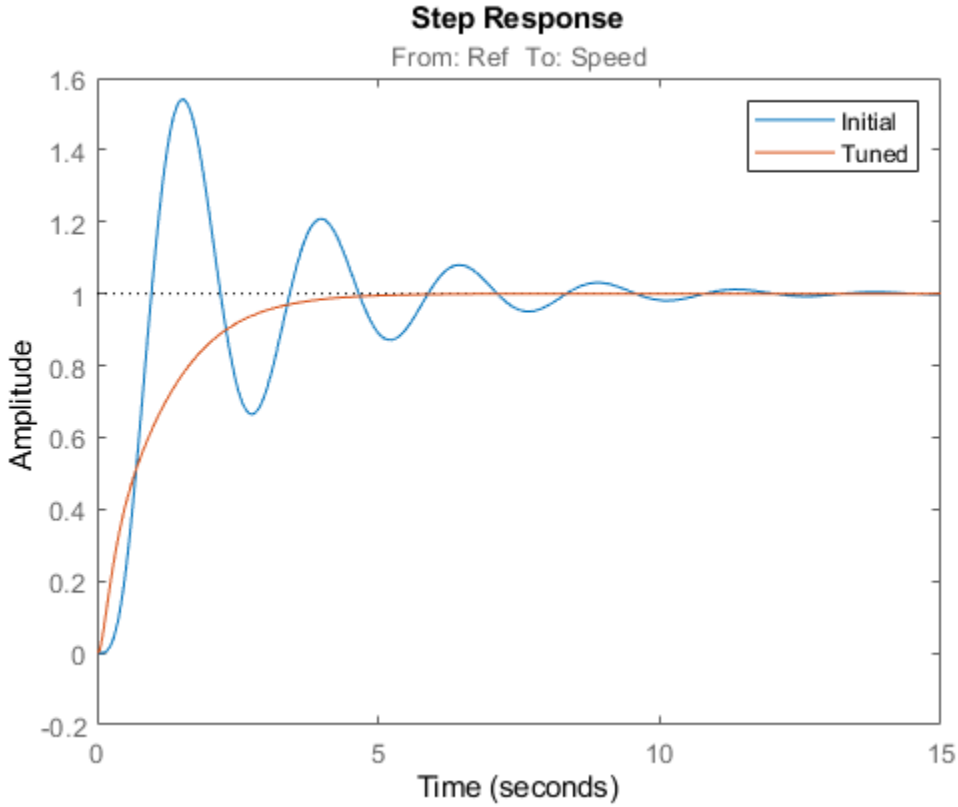
The final value is close to 1 indicating that the tracking requirement is met. `systune` returns a "tuned" version `ST1` of the control system described by `ST0`. Again use `getBlockValue` to access the tuned values of the PID gains:

```
getBlockValue(ST1,'PID Controller')
```

```
ans =

             1            s
  Kp + Ki * --- + Kd * --------
             s          Tf*s+1

  with Kp = 0.00217, Ki = 0.00341, Kd = 0.000512, Tf = 1.08e-06

Name: PID_Controller
Continuous-time PIDF controller in parallel form.
```

To simulate the closed-loop response to a step command in speed, get the initial and tuned transfer functions from speed command "Ref" to "Speed" output and plot their step responses:

```
T0 = getIOTransfer(ST0,'Ref','Speed');
T1 = getIOTransfer(ST1,'Ref','Speed');
step(T0,T1)
legend('Initial','Tuned')
```



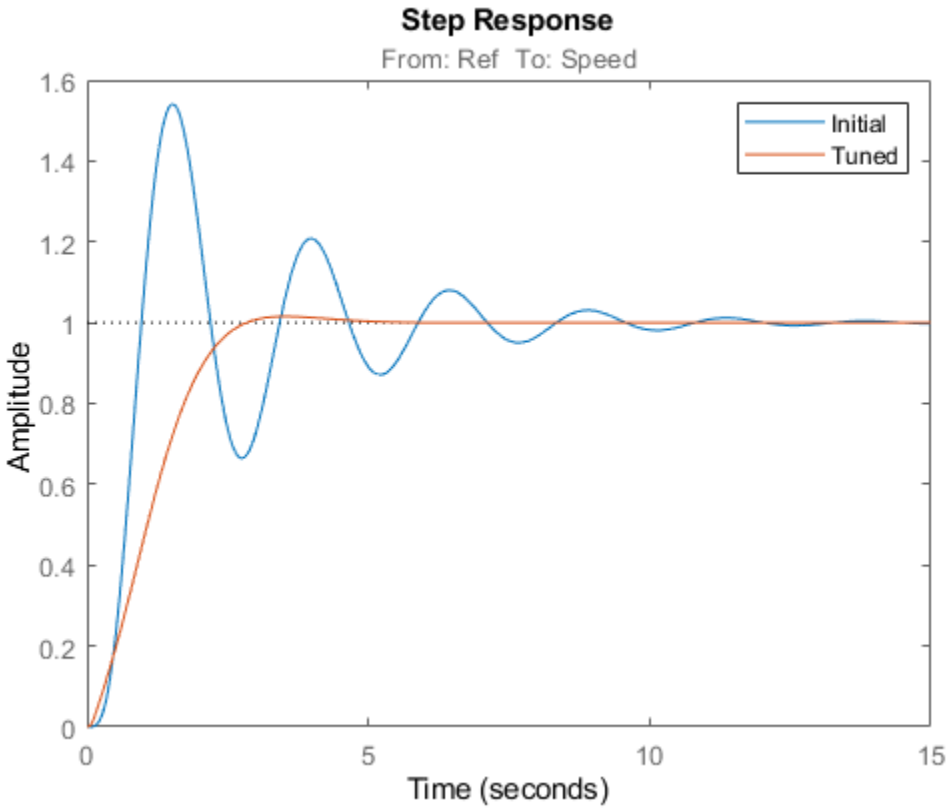
Controller Tuning with LOOPTUNE

You can also use `looptune` to tune control systems modeled in Simulink. The `looptune` workflow is very similar to the `systune` workflow. One difference is that `looptune` needs to know the boundary between the plant and controller, which is specified in terms of *controls* and *measurements* signals. For a single loop the performance is essentially captured by the response time, or equivalently by the open-loop crossover frequency. Based on first-order characteristics the crossover frequency should exceed 1 rad/s for the closed-loop response to settle in less than 5 seconds. You can therefore tune the PID loop using 1 rad/s as target 0-dB crossover frequency.

```
% Mark the signal "u" as a point of interest
addPoint(ST0,'u')

% Tune the controller parameters
Control = 'u';
Measurement = 'Speed';
wc = 1;
ST1 = looptune(ST0,Control,Measurement,wc);

Final: Peak gain = 0.979, Iterations = 4
Achieved target gain value TargetGain=1.
```

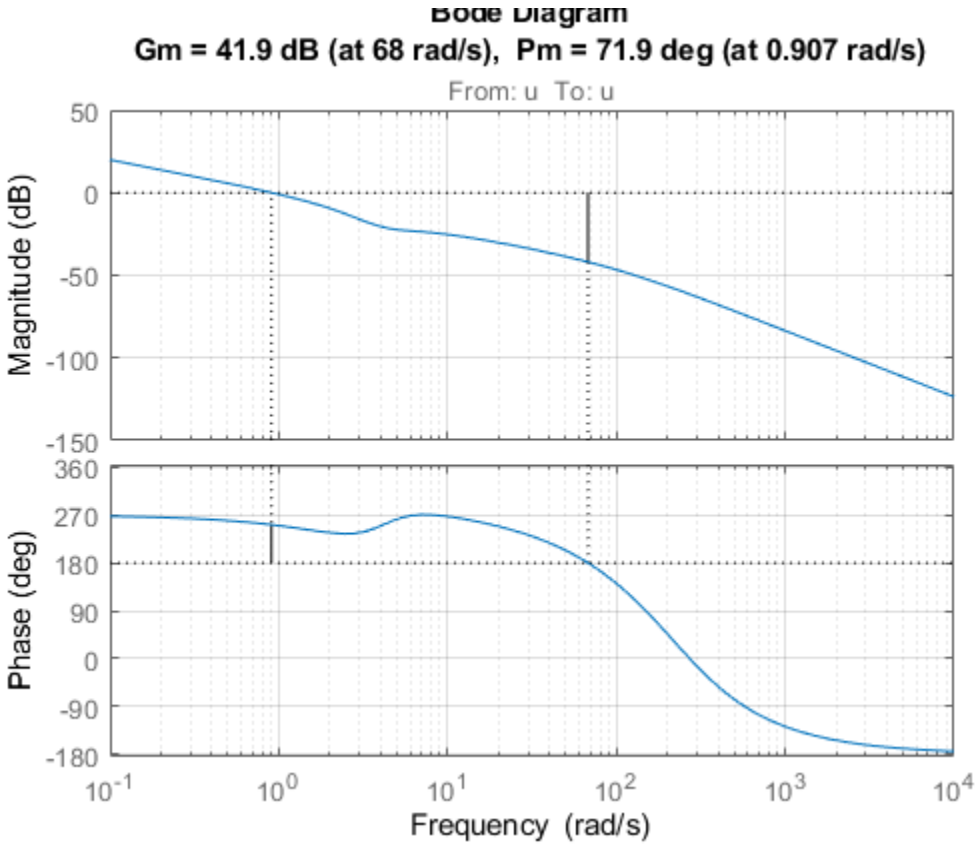
Again the final value is close to 1, indicating that the target control bandwidth was achieved. As with `systune`, use `getIOTransfer` to compute and plot the closed-loop response from speed command to actual speed. The result is very similar to that obtained with `systune`.

```
T0 = getIOTransfer(ST0,'Ref','Speed');
T1 = getIOTransfer(ST1,'Ref','Speed');
step(T0,T1)
legend('Initial','Tuned')
```



You can also perform open-loop analysis, for example, compute the gain and phase margins at the plant input u.

```
% Note: -1 because |margin| expects the negative-feedback loop transfer
L = getLoopTransfer(ST1,'u',-1);

margin(L), grid
```

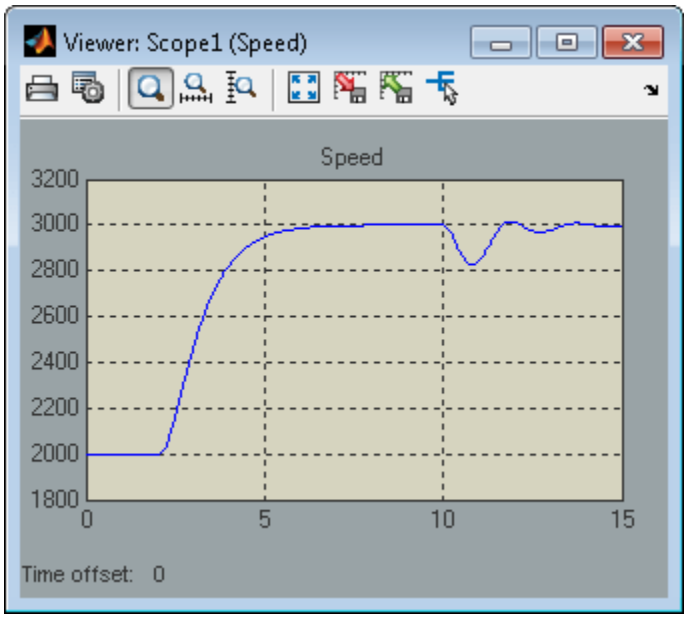


Validation in Simulink

Once you are satisfied with the `systune` or `looptune` results, you can upload the tuned controller parameters to Simulink for further validation with the nonlinear model.

```
writeBlockValue(ST1)
```

You can now simulate the engine response with the tuned PID controller.



The nonlinear simulation results closely match the linear responses obtained in MATLAB.

Constraints on PID Gains

It is often useful to constrain the range of tuned parameters to weed out undesirable solutions. For example, you may require that the proportional and derivative gains of the PID controller be nonnegative. To do this, access the tuned block parameterization.

```
C = getBlockParam(ST0,'PID Controller')
```

```
C =

  Tunable continuous-time PID controller "PID_Controller" with formula:

            1            s
  Kp + Ki * --- + Kd * --------
            s          Tf*s+1

  and tunable parameters Kp, Ki, Kd, Tf.

Type "pid(C)" to see the current value and "get(C)" to see all properties.
```

Set the "Minimum" value of the tunable parameters `Kp` and `Kd` to 0.

```
C.Kp.Minimum = 0;
C.Kd.Minimum = 0;
```

Finally, associate the modified parameterization with the tuned block.

```
setBlockParam(ST0,'PID Controller',C)
```

Retune the PID gains and verify that the proportional and derivative gains are indeed nonnegative.

```
ST1 = looptune(ST0,Control,Measurement,wc);

showTunable(ST1)

Final: Peak gain = 0.964, Iterations = 4
Achieved target gain value TargetGain=1.

Block 1: rct_engine_speed/PID Controller =

             1            s
  Kp + Ki * --- + Kd * --------
             s          Tf*s+1

  with Kp = 0.00091, Ki = 0.00253, Kd = 0.000146, Tf = 0.01

Name: PID_Controller
Continuous-time PIDF controller in parallel form.
```

Comparison of PI and PID Controllers

Closer inspection of the tuned PID gains suggests that the contribution of the derivative term is minor. This suggests using a simpler PI controller instead. To do this, override the default parameterization for the "PID Controller" block:

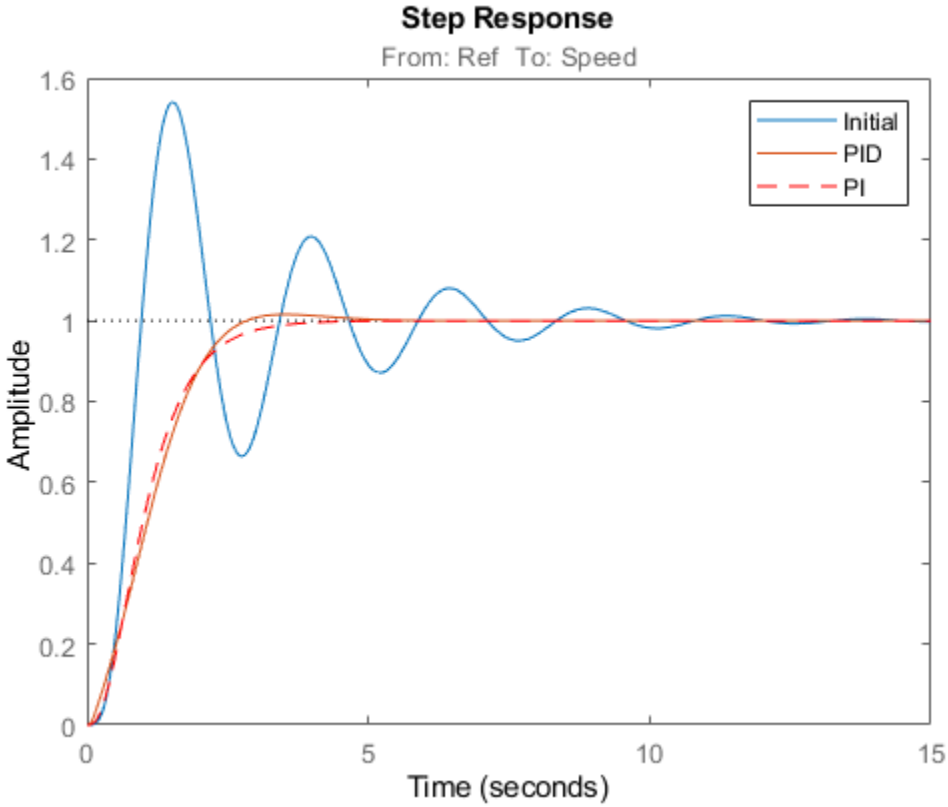
```
setBlockParam(ST0,'PID Controller',tunablePID('C','pi'))
```

This specifies that the "PID Controller" block should now be parameterized as a mere PI controller. Next re-tune the control system for this simpler controller:

```
ST2 = looptune(ST0,Control,Measurement,wc);

Final: Peak gain = 0.95, Iterations = 4
Achieved target gain value TargetGain=1.
```

Again the final value is less than one indicating success. Compare the closed-loop response with the previous ones:

```
T2 = getIOTransfer(ST2,'Ref','Speed');
step(T0,T1,T2,'r--')
legend('Initial','PID','PI')
```



Clearly a PI controller is sufficient for this application.

See Also

`systune (slTuner)` | `slTuner` | `TuningGoal.Tracking`

Related Examples

- “Create and Configure slTuner Interface to Simulink Model”

Tune a Control System Using Control System Tuner

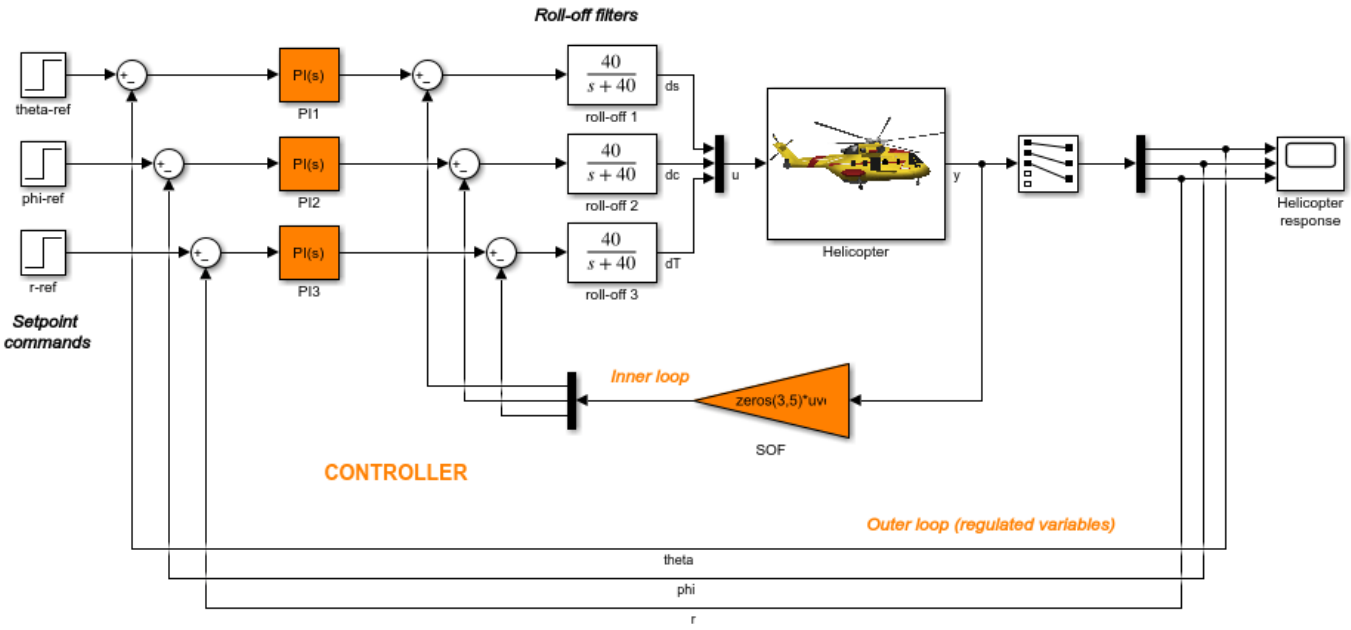
This example shows how to use the Control System Tuner app to tune a MIMO, multiloop control system modeled in Simulink®.

Control System Tuner lets you model any control architecture and specify the structure of controller components, such as PID controllers, gains, and other elements. You specify which blocks in the model are tunable. Control System Tuner parameterizes those blocks and tunes the free parameters system to meet design requirements that you specify, such as setpoint tracking, disturbance rejection, and stability margins.

Control System Model

This example uses the Simulink model `rct_helico`. Open the model.

```
open_system('rct_helico')
```



The plant, `Helicopter`, is an 8-state helicopter model trimmed to a steady-state hovering condition. The state vector `x = [u,w,q,theta,v,p,phi,r]` consists of:

- Longitudinal velocity `u` (m/s)
- Normal velocity `w` (m/s)
- Pitch rate `q` (deg/s)
- Pitch angle `theta` (deg)
- Lateral velocity `v` (m/s)
- Roll rate `p` (deg/s)
- Roll angle `phi` (deg)
- Yaw rate `r` (deg/s)

The control system of the model has two feedback loops. The inner loop provides static output feedback for stability augmentation and decoupling, represented in the model by the gain block `SOF`. The outer loop has a PI controller for each of the three attitude angles. The controller generates commands `ds,dc,dT` in degrees for the longitudinal cyclic, lateral cyclic, and tail rotor collective using measurements of `theta`, `phi`, `p`, `q`, and `r`. This loop provides the desired setpoint tracking for the three angles.

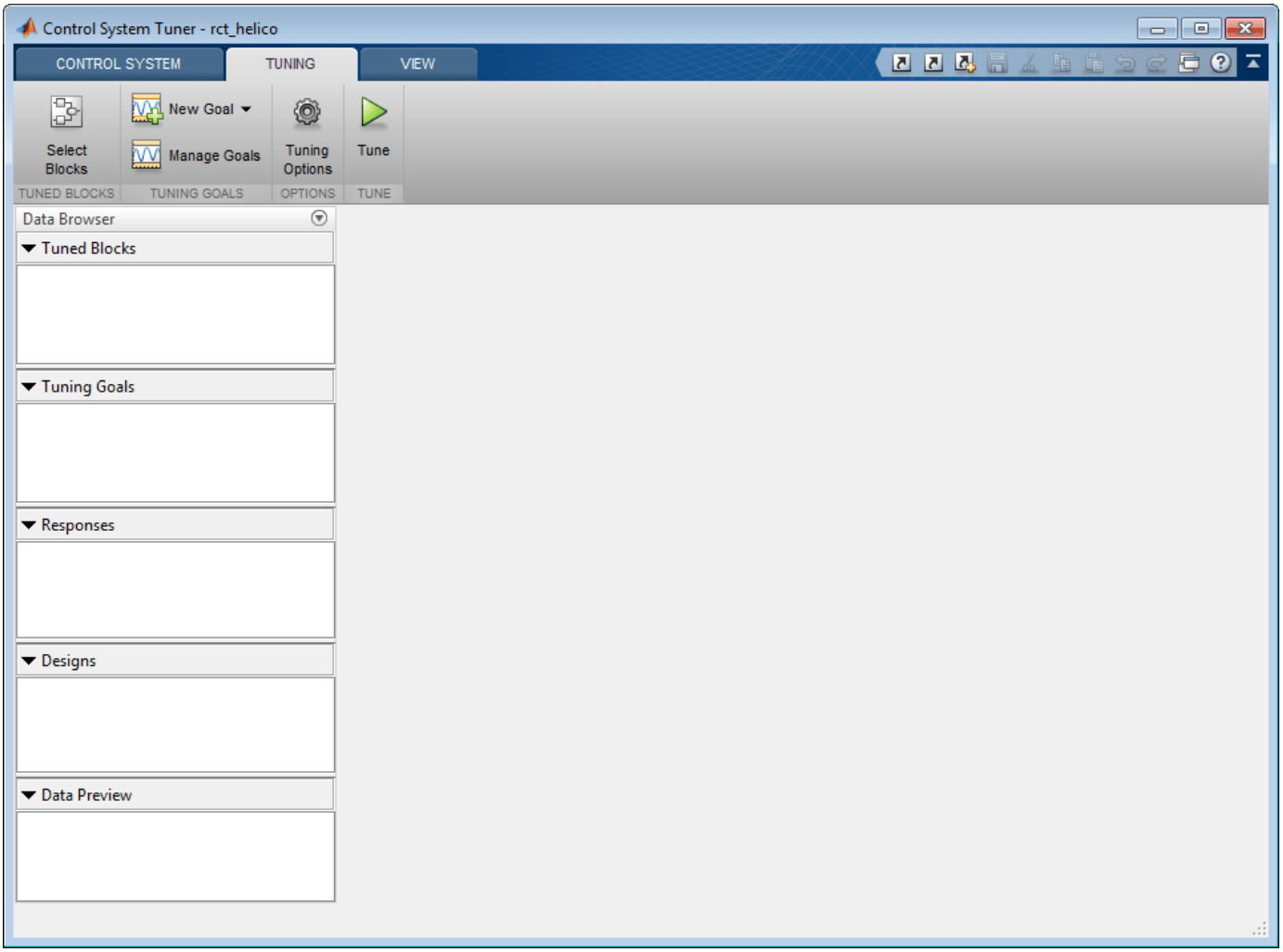
This example uses these control objectives:

- Track setpoint changes in `theta`, `phi`, and `r` with zero steady-state error, rise times of about 2 seconds, minimal overshoot, and minimal cross-coupling.
- Limit the control bandwidth to guard against neglected high-frequency rotor dynamics and measurement noise. (The model contains low-pass filters that partially enforce this objective.)
- Provide strong multivariable gain and phase margins. (Multivariable margins measure robustness to simultaneous gain or phase variations at the plant inputs and outputs. See the `diskmargin` reference page for details.)

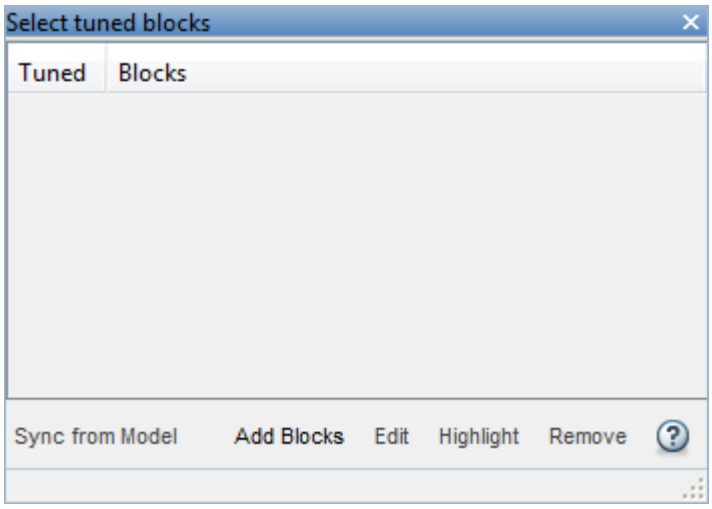
Set Up the Model for Tuning

Using Control System Tuner, you can jointly tune the inner and outer loops to meet all the design requirements. To set up the model for tuning, open the app and specify which blocks of the Simulink model you want to tune.

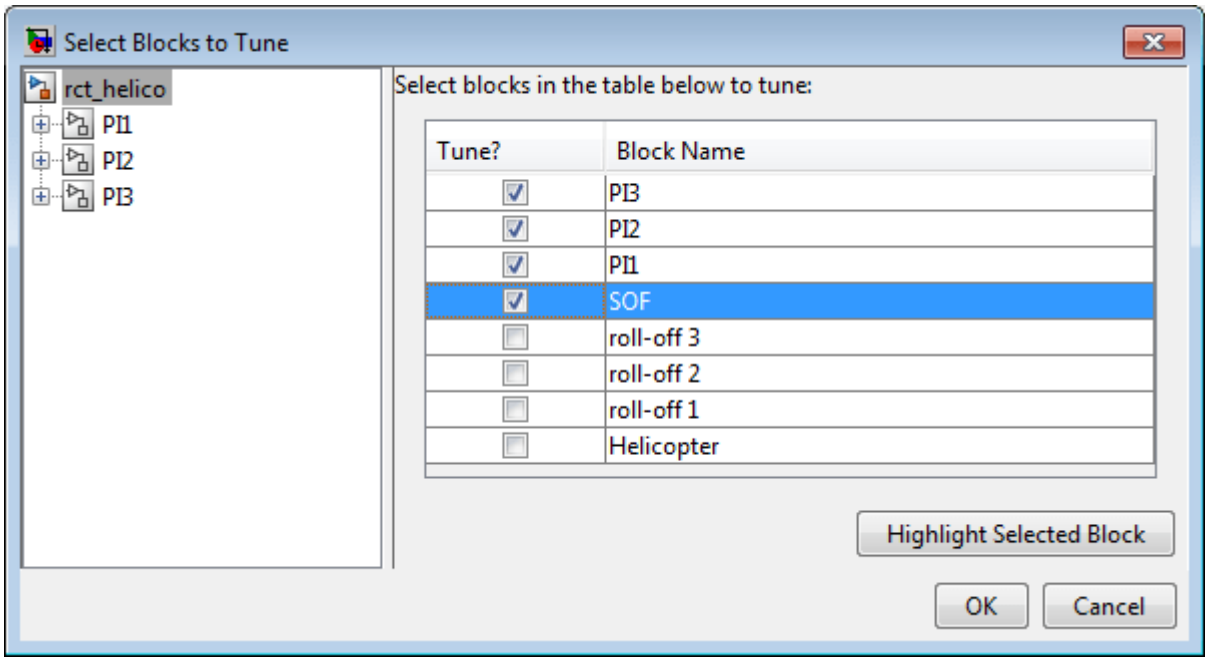
In the Simulink model window, under **Control Systems** in the **Apps** tab, select **Control System Tuner**.



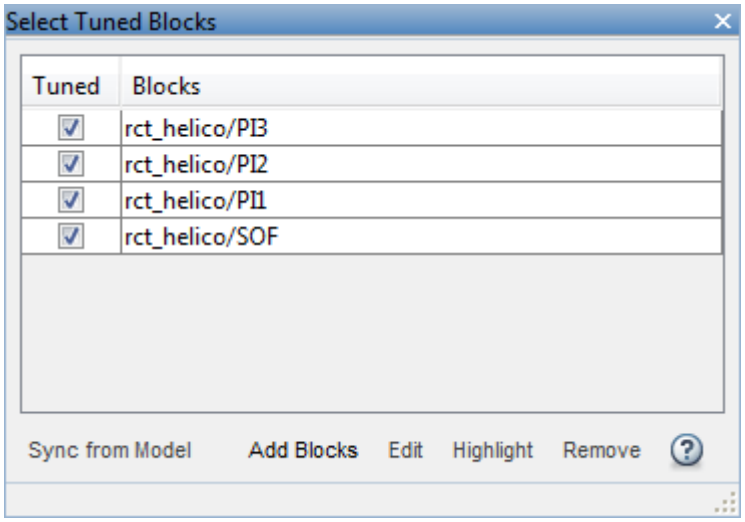
In Control System Tuner, on the **Tuning** tab, click **Select Blocks**. Use the Select tuned blocks dialog box to specify the blocks to tune.



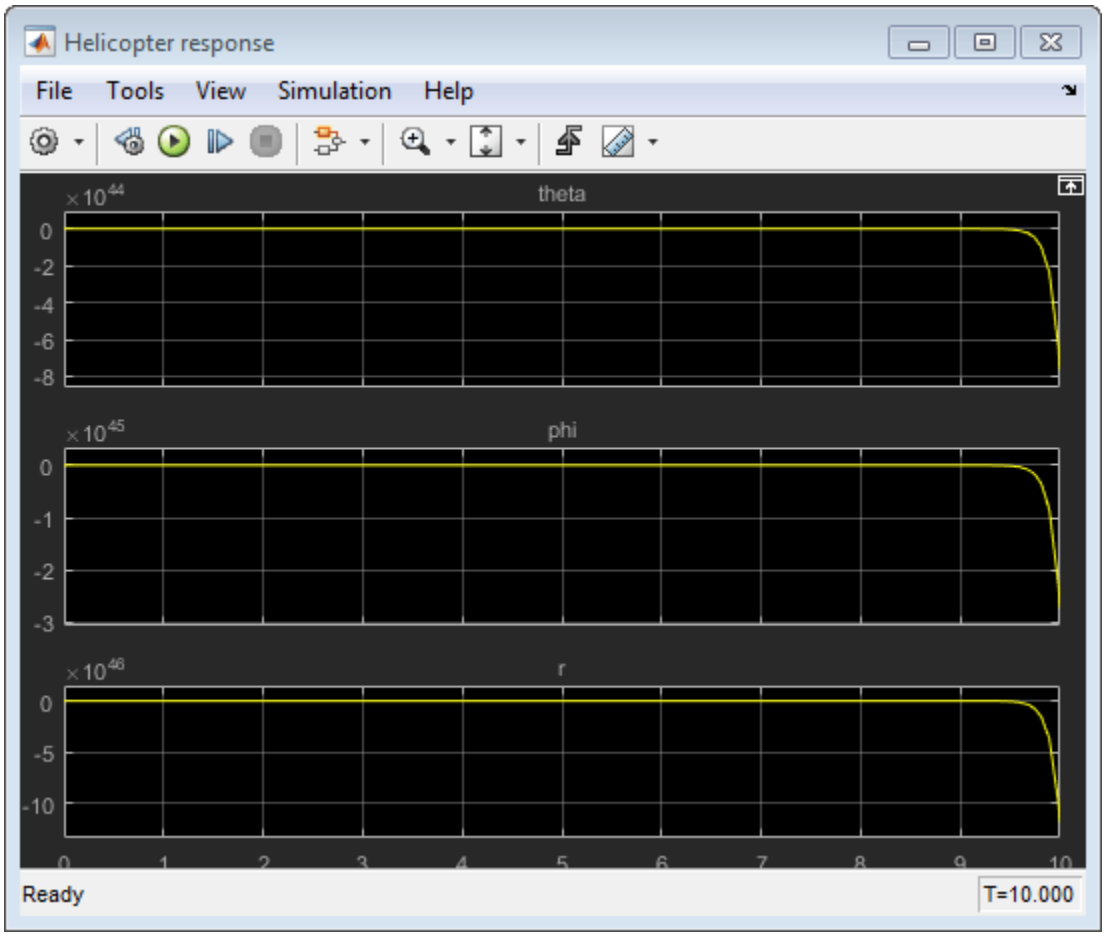
Click **Add Blocks**. Control System Tuner analyzes your model to find blocks that can be tuned. For this example, the controller blocks to tune are the three PI controllers and the gain block. Check the corresponding blocks `PI1`, `PI2`, `PI3`, and `SOF`.



Click **OK**. The Select tuned blocks dialog box now reflects the blocks you added.



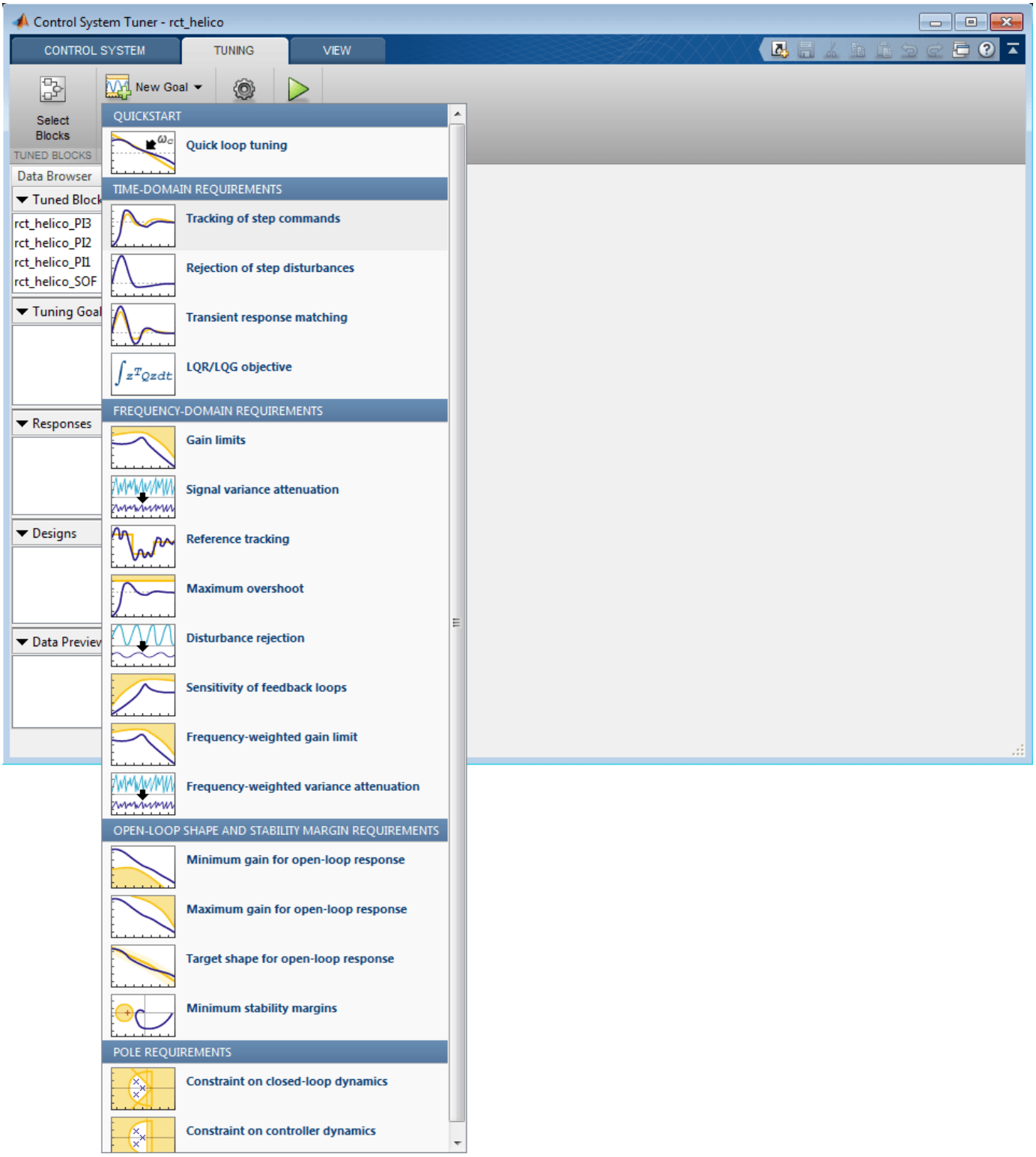
When you select a block to tune, Control System Tuner automatically parameterizes the block according to its type and initializes the parameterization with the block value in the Simulink model. In this example, the PI controllers are initialized to $1 + 1/s$ and the static output-feedback gain is initialized to zero on all channels. Simulating the model shows that the control system is unstable for these initial values.



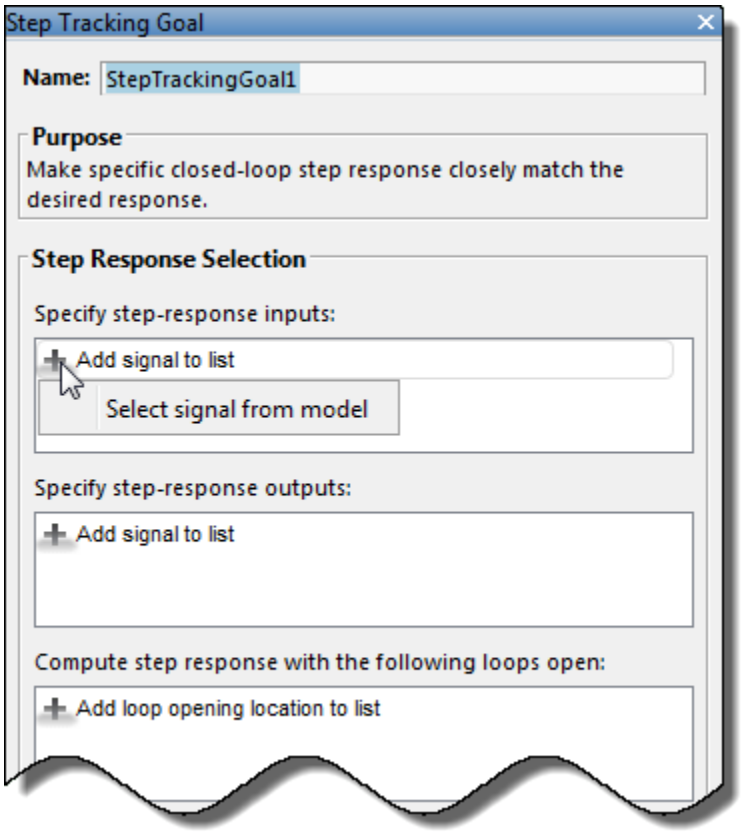
Specify Tuning Goals

The design requirements for this system, discussed previously, include setpoint tracking, minimum stability margins, and a limit on fast dynamics. In Control System Tuner, you capture design requirements using *tuning goals*.

First, create a tuning goal for the setpoint-tracking requirement on `theta`, `phi`, and `r`. On the **Tuning** tab, in the **New Goal** drop-down list, select `Tracking of step commands`.



In the Step Tracking Goal dialog, specify the reference signals for tracking. Under **Specify step-response inputs**, click **Add signal to list**. Then click **Select signal from model**.

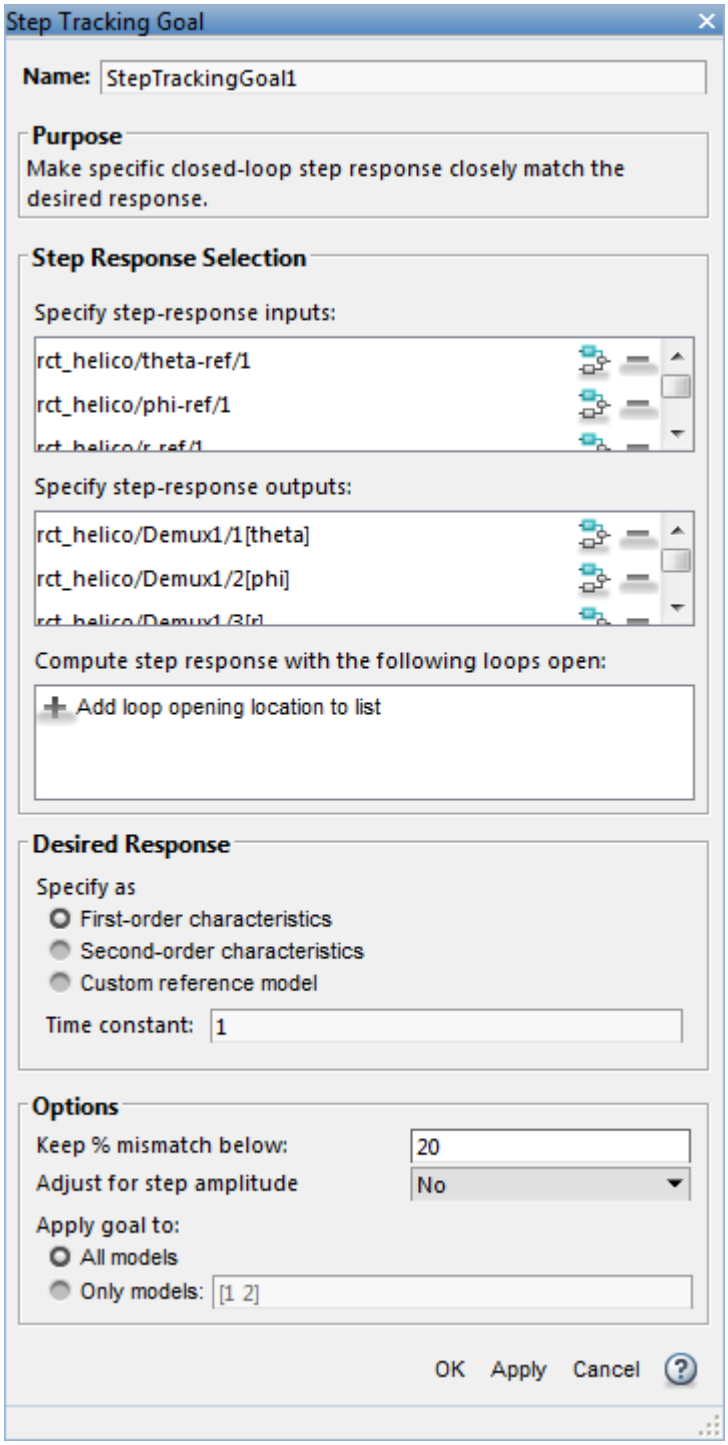


In the Simulink model editor, select the reference signals `theta_ref`, `phi_ref`, and `r_ref`. These signals appear in the Select signals dialog box. Click **Add Signal(s)** to add them to the step tracking goal.

Next, specify the outputs that you want to track those references. Under **Specify step-response outputs**, add the outputs `theta`, `phi`, and `r`.

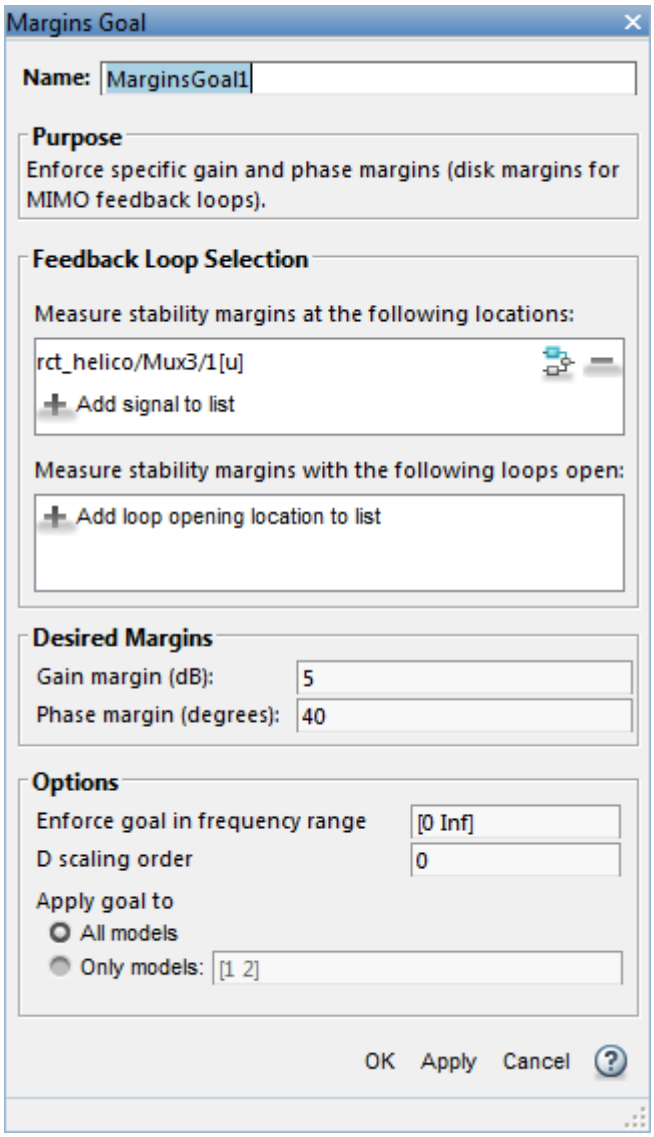
The requirement is that the responses at the outputs track the reference commands with a first-order response that has a one-second time constant. Enter these values in the **Desired Response** section of the dialog box. Also, for this example set **Keep mismatch below** to 20. This value sets a 20% relative mismatch between the target first-order response and the tuned response.

This figure shows the configuration of the Step Tracking Goal dialog box. Click **OK** to save the tuning goal.

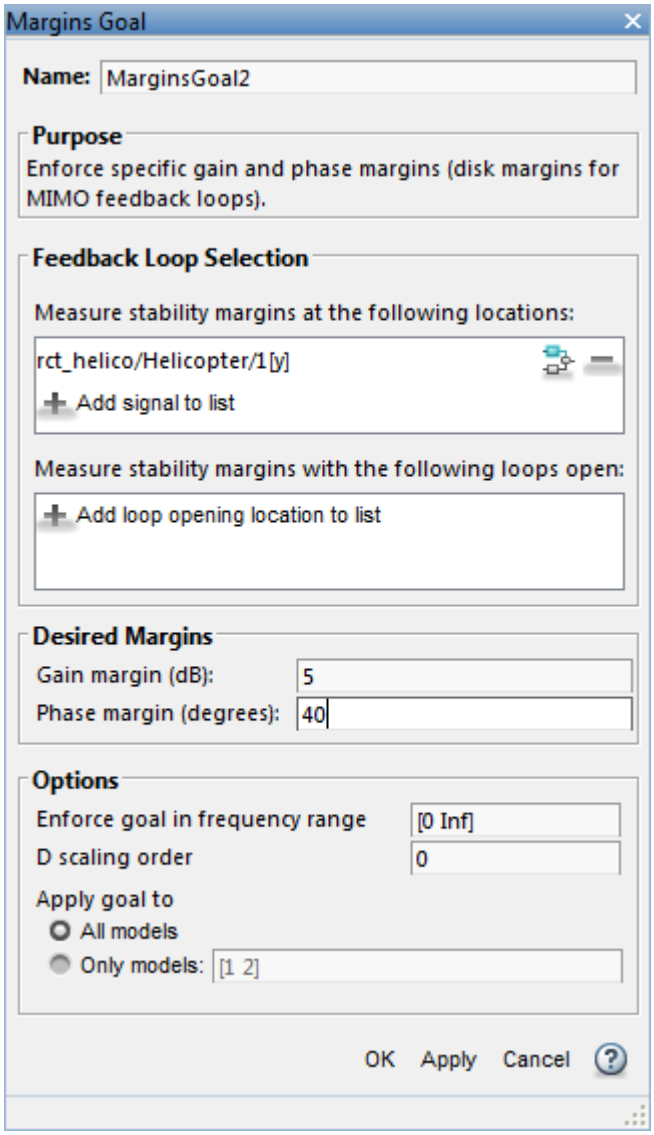


Next, create tuning goals for the desired stability margin requirements. For this example, the multivariable gain and phase margins at the plant inputs u and plant outputs y must be at least 5 dB and 40 degrees. Create separate tuning goals for the input and output margin constraints. In the **New Goal** drop-down list, select **Minimum stability margins**. In the Margins Goal dialog box, add the input signal u under **Measure stability margins at the following locations**. Also, enter the

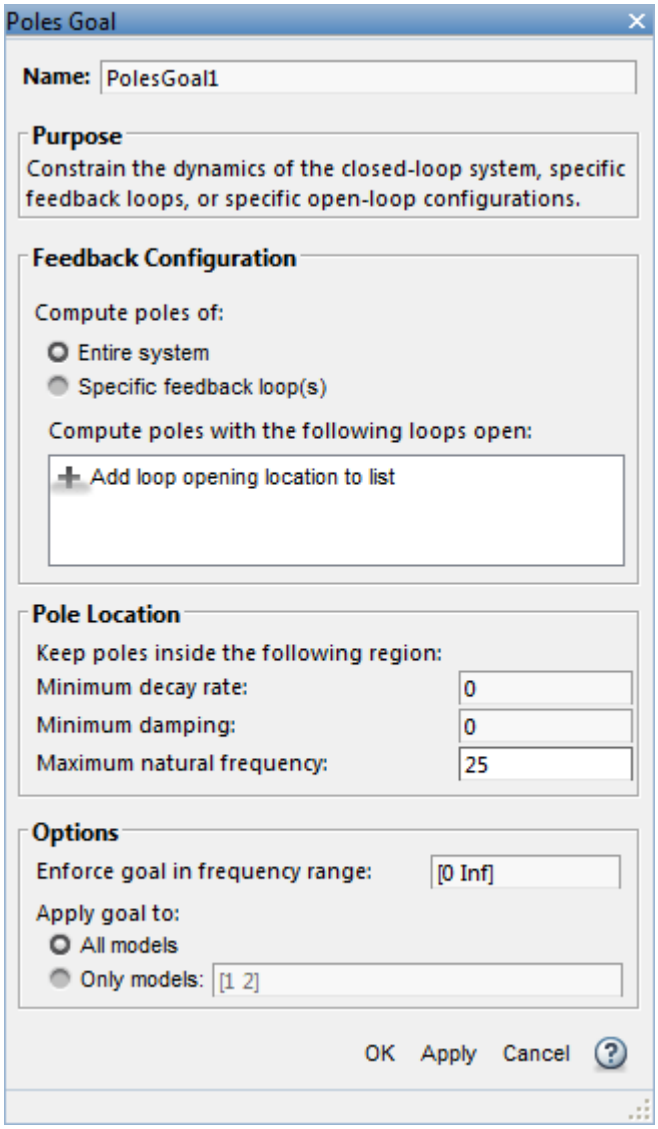
gain and phase values 5 and 40 in the **Desired Margins** section of the dialog box. Click **OK** to save the input stability margin goal.



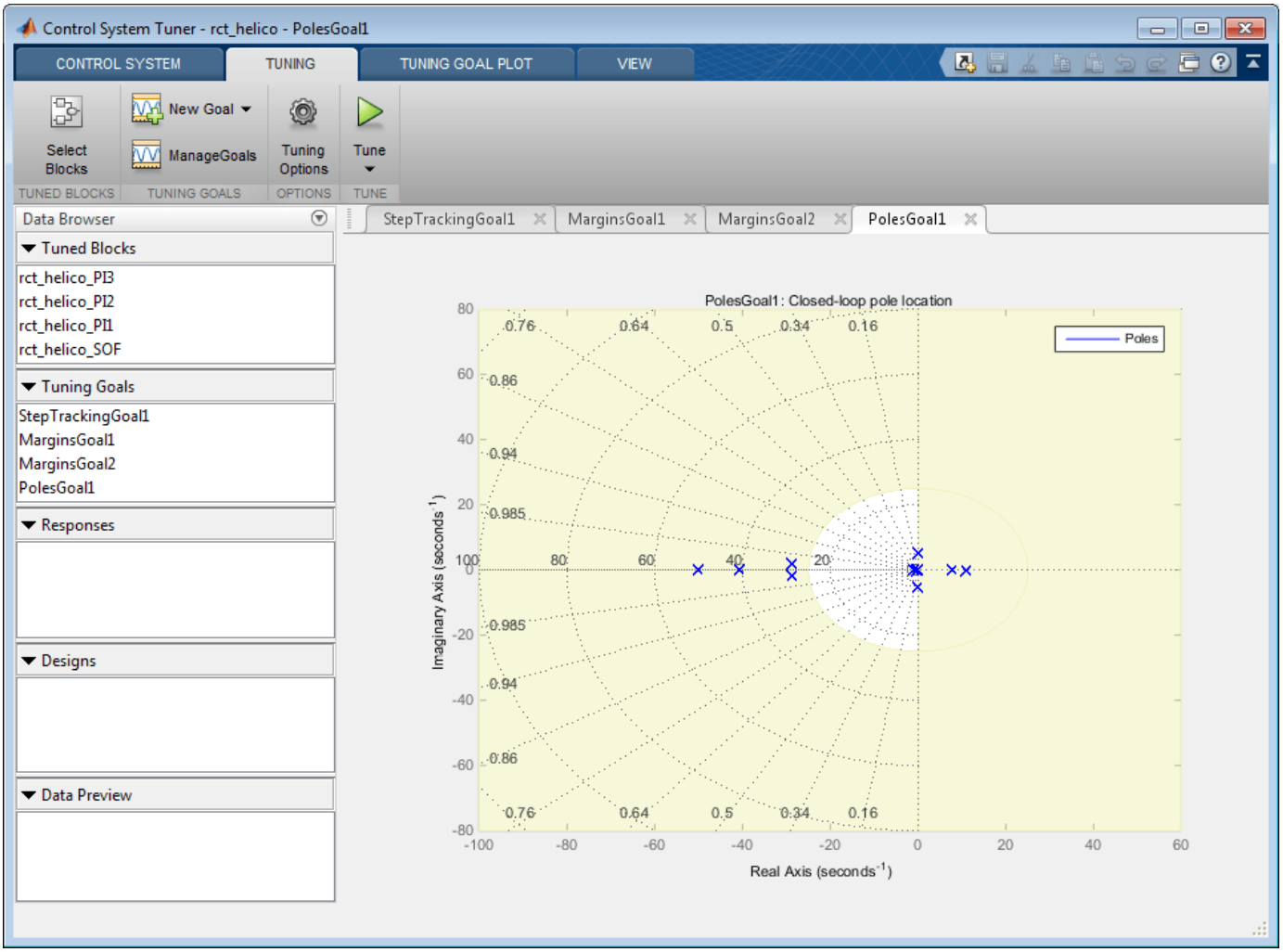
Create another Margins Goal for the output stability margin. Specify the output signal `y` and the target margins, as shown, and save the output stability margin goal.



The last requirement is to limit fast dynamics and jerky transients. To achieve this, create a tuning goal that constrains the magnitude of the closed-loop poles to less than 25 rad/s. In the **New Goal** drop-down list, select **Constraint on closed-loop dynamics**. In the **Poles Goal** dialog box, specify the maximum natural frequency of 25, and click **OK** to save the tuning goal.



As you create each tuning goal, Control System Tuner creates a new figure that displays a graphical representation of the tuning goal. When you tune your control system, you can refer to this figure for a graphical representation of how closely the tuned system satisfies the tuning goal.

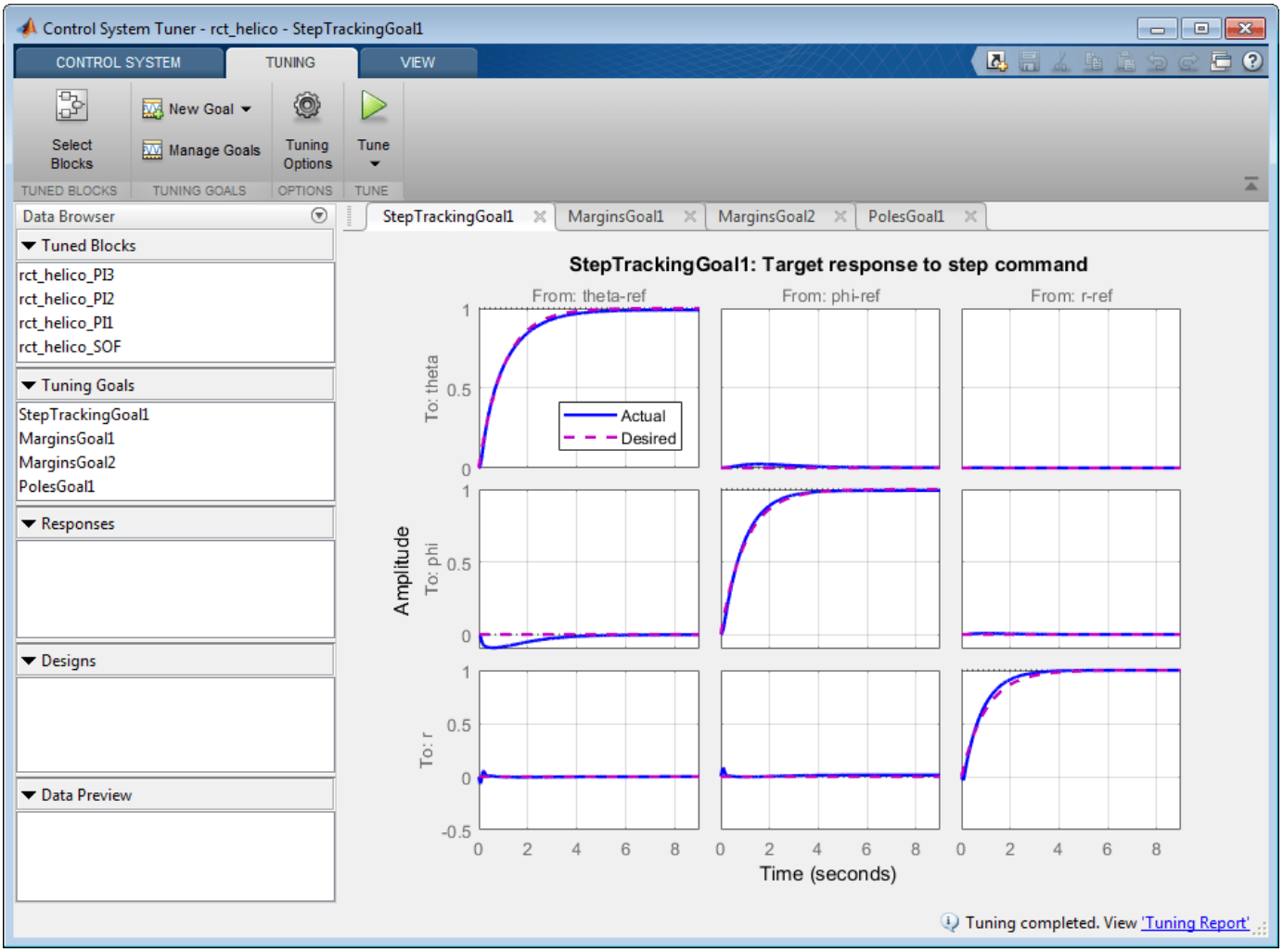


Tune the Control System

Tune the control system to meet the design requirements you have specified.

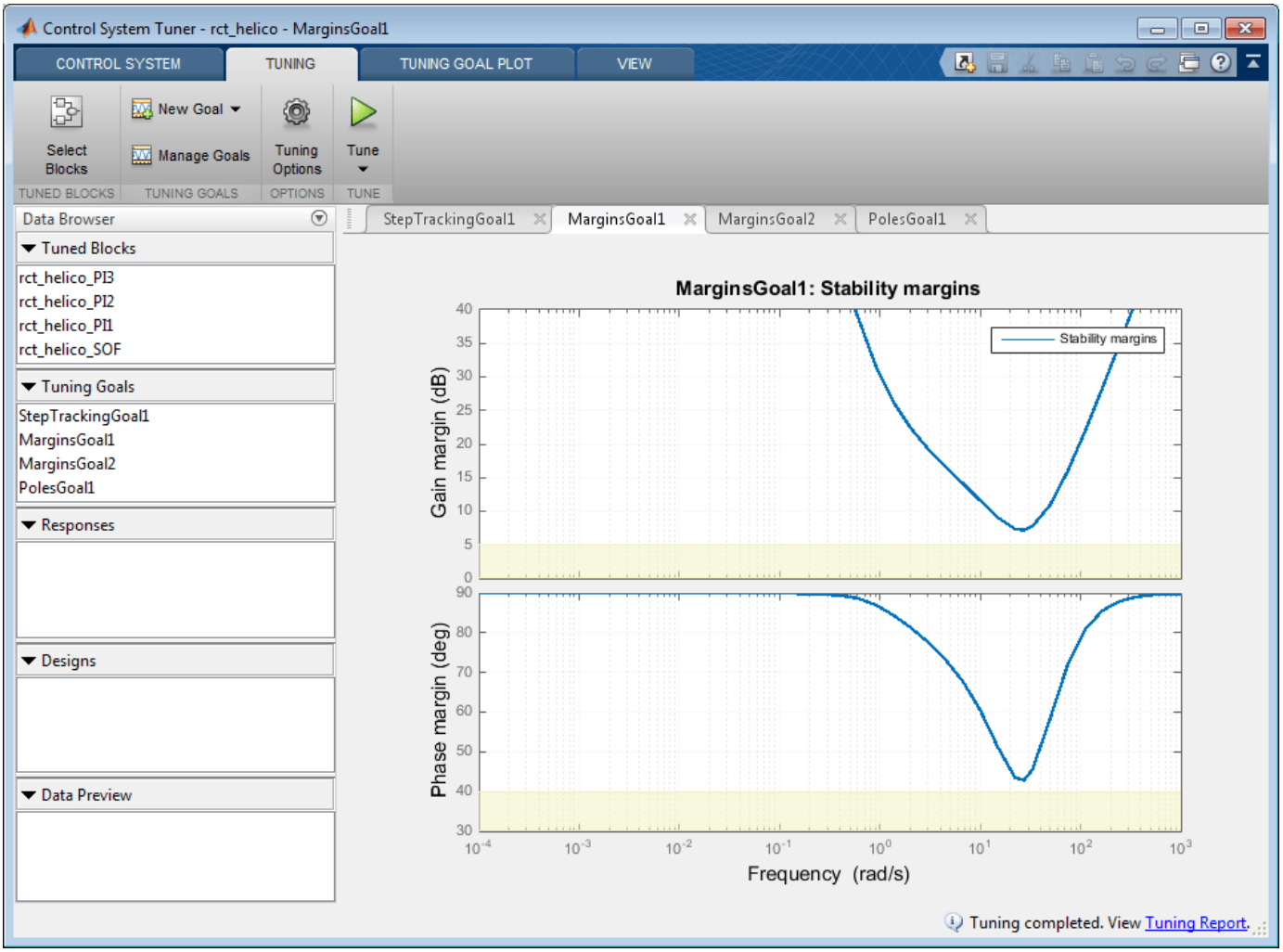
On the **Tuning** tab, click **Tune**. Control System Tuner adjusts the tunable parameters to values that best meet those requirements.

Control System Tuner automatically updates the tuning-goal plots to reflect the tuned parameter values. Examine these plots to see how well the requirements are satisfied by the design. For instance, examine the tuned step responses of tracking requirements.

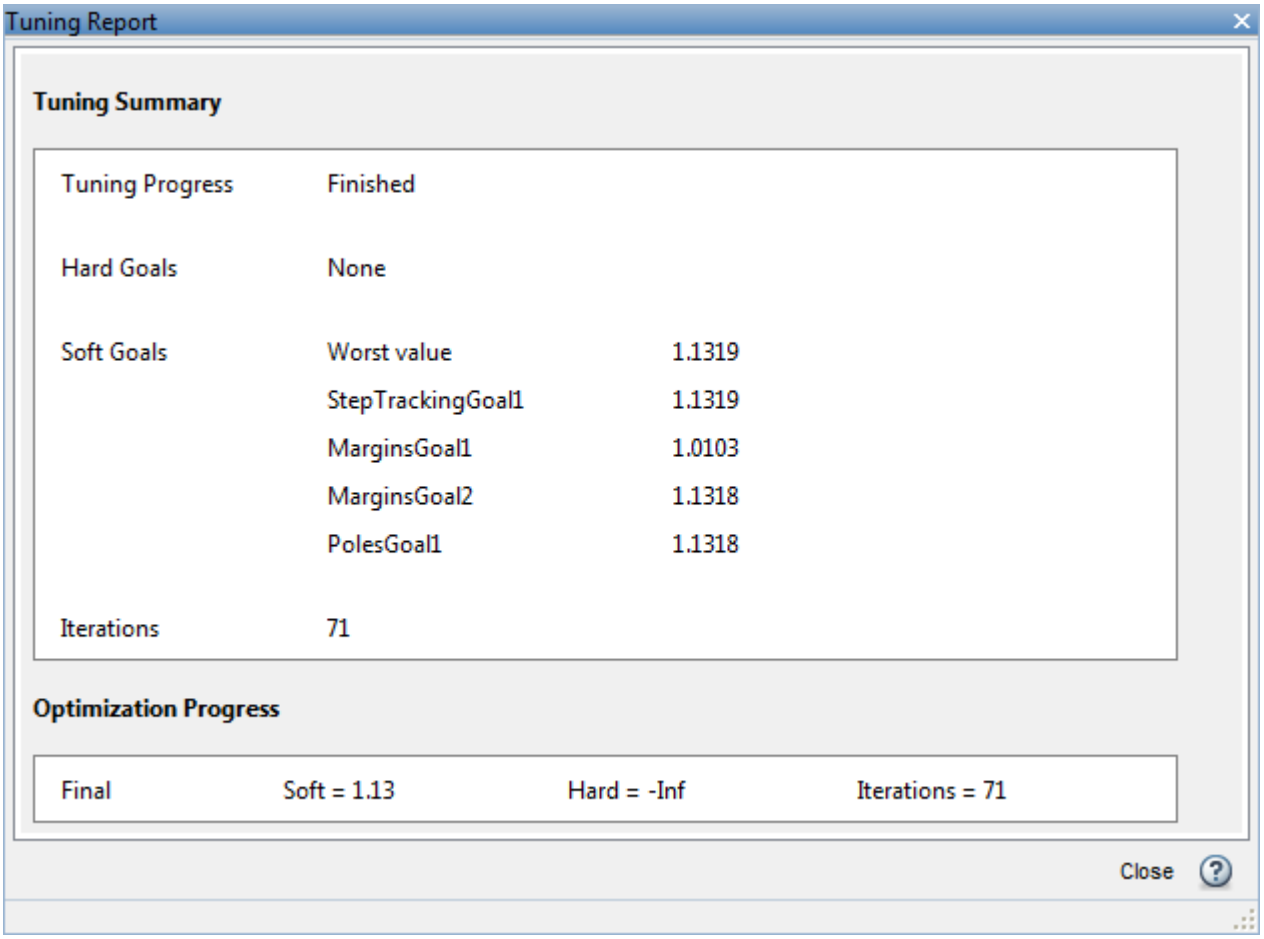


The blue line shows that the tuned response is very close to the target response, in pink. The rise time is about two seconds, and there is no overshoot and little cross-coupling.

Similarly, the `MarginsGoal1` and `MarginsGoal2` plots provide a visual assessment of the multivariable stability margins. (See the `diskmargin` reference page for more information about multivariable stability margins.) These plots show that the stability margin is out of the shaded region, satisfying the requirement at all frequencies.



You can also view a numeric report of the tuning results. Click the **Tuning Report** at the bottom right of Control System Tuner.

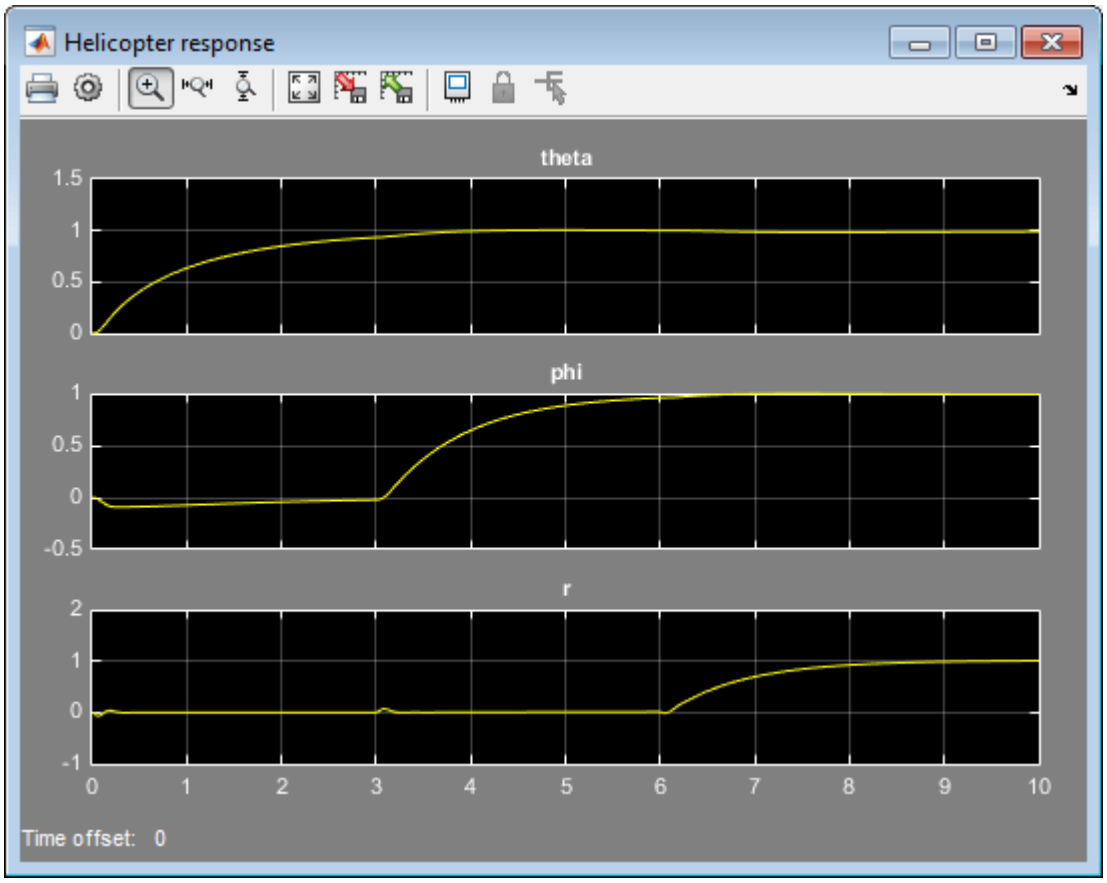


When you tune the model, Control System Tuner converts each tuning goal to a function of the tunable parameters of the system and adjusts the parameters to minimize the value of those functions. For this example, the tuning report shows that the final values for all tuning goals are close to 1, which indicates that all the requirements are nearly met.

Validate the Tuned Design

In general, your Simulink model represents a nonlinear system. Control System Tuner linearizes the model at the operating point you specify in the app, and tunes parameters using the linear approximation of your system. Therefore, it is important to validate the controller design on the full Simulink model.

To do so, write the tuned parameter values back to the Simulink model. On the **Control System** tab, click **Update Blocks**. In the Simulink model window, simulate the model with the new parameter values. Observe the response to the step changes in setpoint commands, `theta-ref`, `phi-ref`, and `r-ref` at 0, 3, and 6 seconds respectively.



Examine the simulation to confirm that you get the desired responses in the Simulink model. Here, the rise time of each response is about 2 seconds with no overshoot, no steady-state error, and minimal cross-coupling, as specified in the design requirements.

See Also

Control System Tuner

Related Examples

- “Specify Operating Points for Tuning in Control System Tuner”
- “Tuning for Multiple Values of Plant Parameters” on page 13-213

Using Parallel Computing to Accelerate Tuning

This example shows how to leverage the Parallel Computing Toolbox™ to accelerate multi-start strategies for tuning fixed-structure control systems.

Background

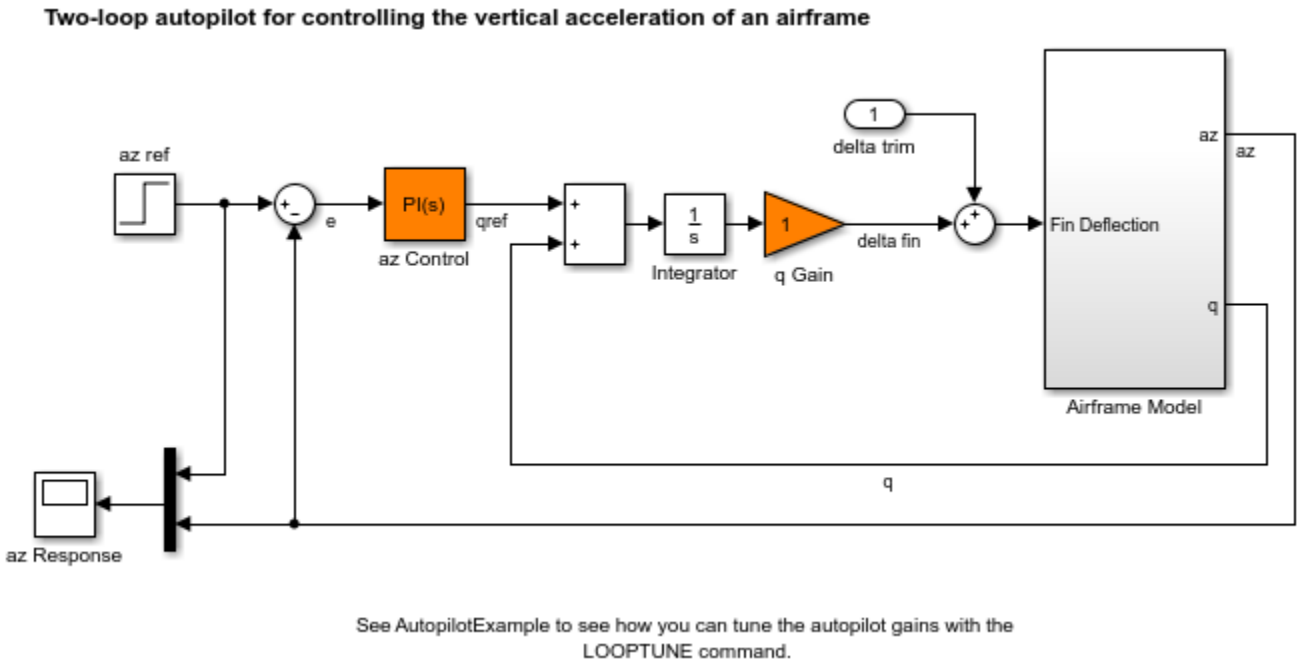
Both `systune` and `looptune` use local optimization methods for tuning the control architecture at hand. To mitigate the risk of ending up with a locally optimal but globally poor design, it is recommended to run several optimizations starting from different randomly generated initial points. If you have a multi-core machine or have access to distributed computing resources, you can significantly speed up this process using the Parallel Computing Toolbox.

This example shows how to parallelize the tuning of an airframe autopilot with `looptune`. See the example "Tuning of a Two-Loop Autopilot" for more details about this application of `looptune`.

Autopilot Tuning

The airframe dynamics and autopilot are modeled in Simulink.

```
open_system('rct_airframe1')
```



The autopilot consists of two cascaded loops whose tunable elements include two PI controller gains ("az Control" block) and one gain in the pitch-rate loop ("q Gain" block). The vertical acceleration `az` should track the command `azref` with a 1 second response time. Use `slTuner` to configure this tuning task (see "Tuning of a Two-Loop Autopilot" example for details):

```
ST0 = slTuner('rct_airframe1',{'az Control','q Gain'});
addPoint(ST0,{'az ref','delta fin','az','q'})

% Design requirements
```

```
wc = [3,12];   % bandwidth
TrackReq = TuningGoal.Tracking('az ref','az',1);  % tracking
```

Parallel Tuning with LOOPTUNE

We are ready to tune the autopilot gains with `looptune`. To minimize the risk of getting a poor-quality local minimum, run 30 optimizations starting from 30 randomly generated values of the three gains. Configure the `looptune` options to enable parallel processing of these 30 runs:

```
rng('default')
Options = looptuneOptions('RandomStart',30,'UseParallel',true);
```

Next call `looptune` to launch the tuning algorithm. The 30 runs are automatically distributed across available computing resources:

```
Controls = 'delta fin';
Measurements = {'az','q'};
[ST,gam,Info] = looptune(ST0,Controls,Measurements,wc,TrackReq,Options);
```

```
Starting parallel pool (parpool) using the 'local' profile ... connected to 6 workers.
Final: Failed to enforce closed-loop stability (max Re(s) = 0.042)
Final: Failed to enforce closed-loop stability (max Re(s) = 0.039)
Final: Failed to enforce closed-loop stability (max Re(s) = 0.041)
Final: Failed to enforce closed-loop stability (max Re(s) = 0.041)
Final: Failed to enforce closed-loop stability (max Re(s) = 0.041)
Final: Failed to enforce closed-loop stability (max Re(s) = 0.082)
Final: Failed to enforce closed-loop stability (max Re(s) = 0.041)
Final: Peak gain = 1.23, Iterations = 53
Final: Peak gain = 62, Iterations = 92
       Some closed-loop poles are marginally stable (decay rate near 1e-07)
Final: Peak gain = 62, Iterations = 128
       Some closed-loop poles are marginally stable (decay rate near 1e-07)
Final: Peak gain = 1.23, Iterations = 128
Final: Failed to enforce closed-loop stability (max Re(s) = 0.041)
Final: Peak gain = 1.23, Iterations = 130
Final: Failed to enforce closed-loop stability (max Re(s) = 0.041)
Final: Failed to enforce closed-loop stability (max Re(s) = 0.041)
Final: Failed to enforce closed-loop stability (max Re(s) = 0.04)
Final: Failed to enforce closed-loop stability (max Re(s) = 0.041)
Final: Failed to enforce closed-loop stability (max Re(s) = 0.082)
Final: Failed to enforce closed-loop stability (max Re(s) = 0.041)
Final: Peak gain = 1.23, Iterations = 98
Final: Peak gain = 61.9, Iterations = 79
Final: Failed to enforce closed-loop stability (max Re(s) = 0.039)
Final: Failed to enforce closed-loop stability (max Re(s) = 0.041)
Final: Failed to enforce closed-loop stability (max Re(s) = 0.041)
Final: Failed to enforce closed-loop stability (max Re(s) = 0.041)
Final: Failed to enforce closed-loop stability (max Re(s) = 0.041)
Final: Failed to enforce closed-loop stability (max Re(s) = 0.082)
Final: Failed to enforce closed-loop stability (max Re(s) = 0.051)
Final: Failed to enforce closed-loop stability (max Re(s) = 0.041)
Final: Failed to enforce closed-loop stability (max Re(s) = 0.041)
Final: Peak gain = 1.23, Iterations = 42
```

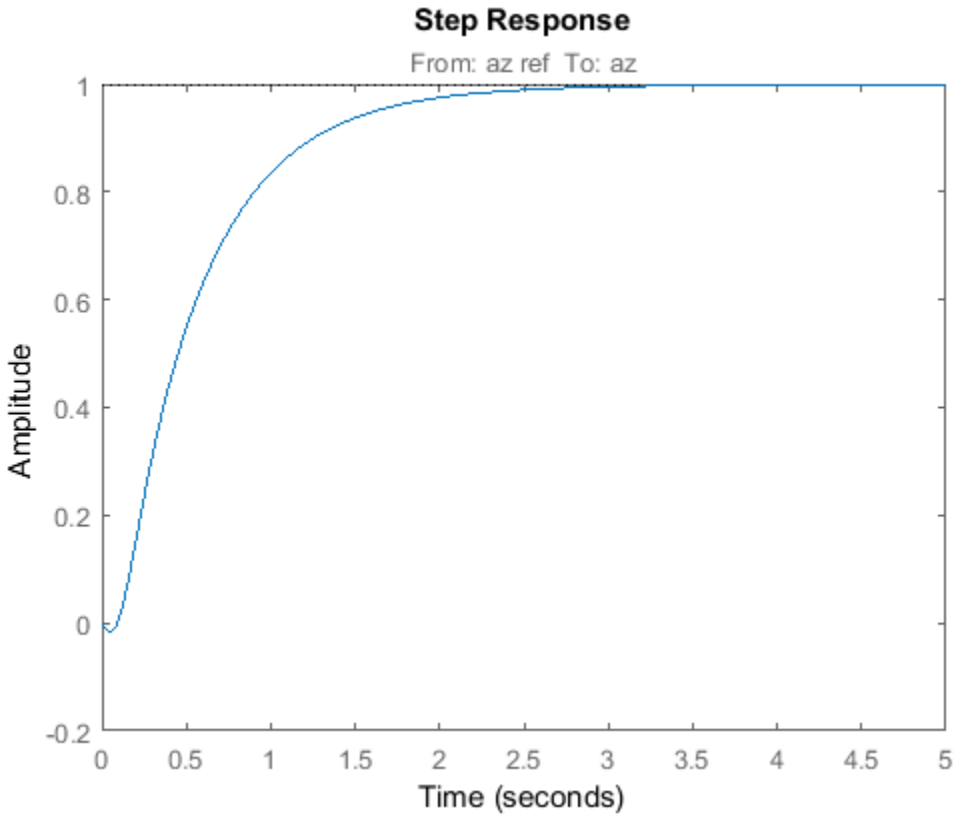
Most runs return 1.23 as optimal gain value, suggesting that this local minimum has a wide region of attraction and is likely to be the global optimum. Use `showBlockValue` to see the corresponding gain values:

```
showBlockValue(ST)
```

```
AnalysisPoints_ =

  D =
       u1  u2  u3  u4
   y1   1   0   0   0
   y2   0   1   0   0
   y3   0   0   1   0
   y4   0   0   0   1

Name: AnalysisPoints_
Static gain.
-----------------------------------
az_Control =

             1
  Kp + Ki * ---
             s

  with Kp = 0.00165, Ki = 0.00166

Name: az_Control
Continuous-time PI controller in parallel form.
-----------------------------------
q_Gain =

  D =
          u1
   y1  1.985

Name: q_Gain
Static gain.
```

Plot the closed-loop response for this set of gains:

```
T = getIOTransfer(ST,'az ref','az');
step(T,5)
```



See Also

`systune (slTuner)` | `slTuner` | `systune`

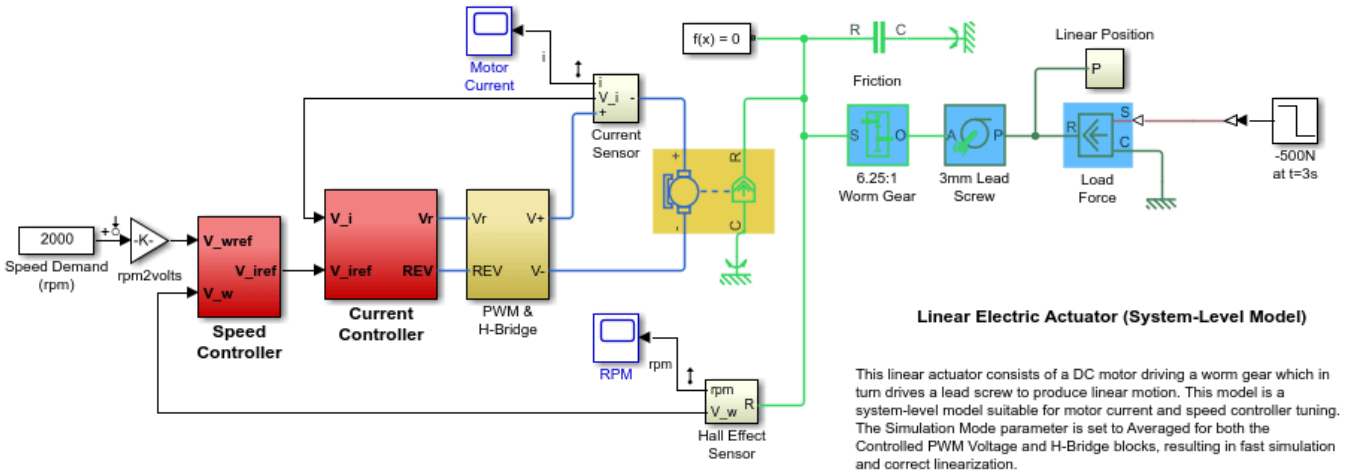
Control of a Linear Electric Actuator

This example shows how to use `slTuner` and `systune` to tune the current and velocity loops in a linear electric actuator with saturation limits.

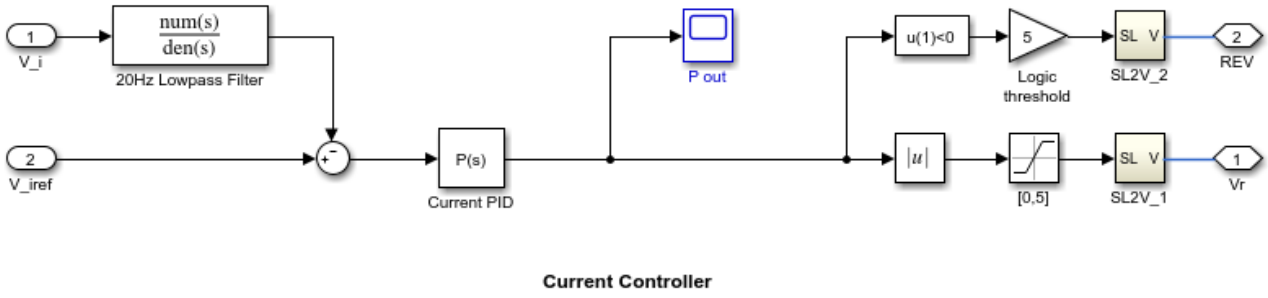
Linear Electric Actuator Model

Open the Simulink model of the linear electric actuator:

```
open_system('rct_linact')
```



The electrical and mechanical components are modeled using Simscape Electrical. The control system consists of two cascaded feedback loops controlling the driving current and angular speed of the DC motor.



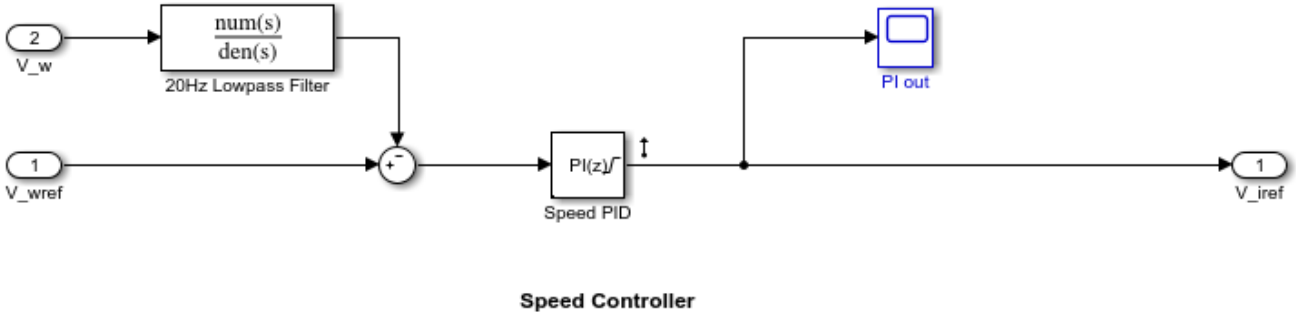


Figure 1: Current and Speed Controllers.

Note that the inner-loop (current) controller is a proportional gain while the outer-loop (speed) controller has proportional and integral actions. The output of both controllers is limited to plus/minus 5.

Design Specifications

We need to tune the proportional and integral gains to respond to a 2000 rpm speed demand in about 0.1 seconds with minimum overshoot. The initial gain settings in the model are P=50 and PI(s)=0.2+0.1/s and the corresponding response is shown in Figure 2. This response is too slow and too sensitive to load disturbances.

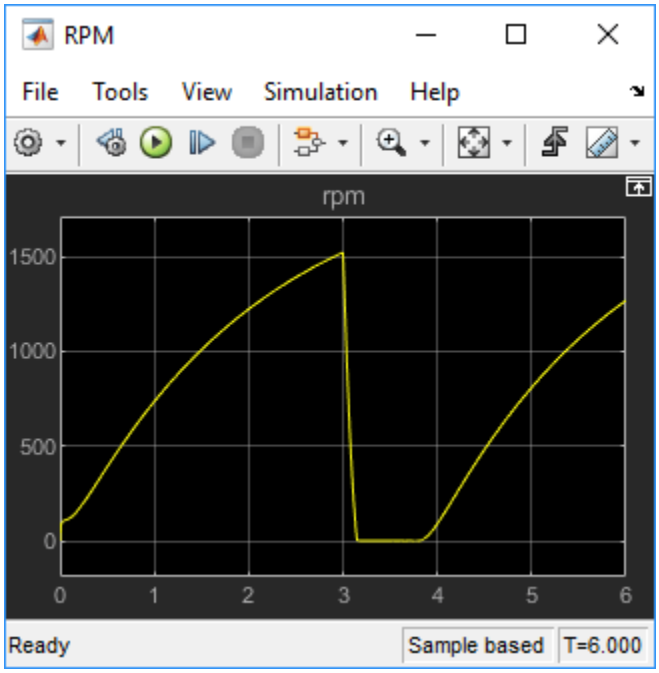


Figure 2: Untuned Response.

Control System Tuning

You can use `systune` to jointly tune both feedback loops. To set up the design, create an instance of the `slTuner` interface with the list of tuned blocks. All blocks and signals are specified by their names in the model. The model is linearized at t=0.5 to avoid discontinuities in some derivatives at t=0.

```
TunedBlocks = {'Current PID','Speed PID'};
tLinearize = 0.5;  % linearize at t=0.5

% Create tuning interface
ST0 = slTuner('rct_linact',TunedBlocks,tLinearize);
addPoint(ST0,{'Current PID','Speed PID'})
```

The data structure `ST0` contains a description of the control system and its tunable elements. Next specify that the DC motor should follow a 2000 rpm speed demand in 0.1 seconds:

```
TR = TuningGoal.Tracking('Speed Demand (rpm)','rpm',0.1);
```

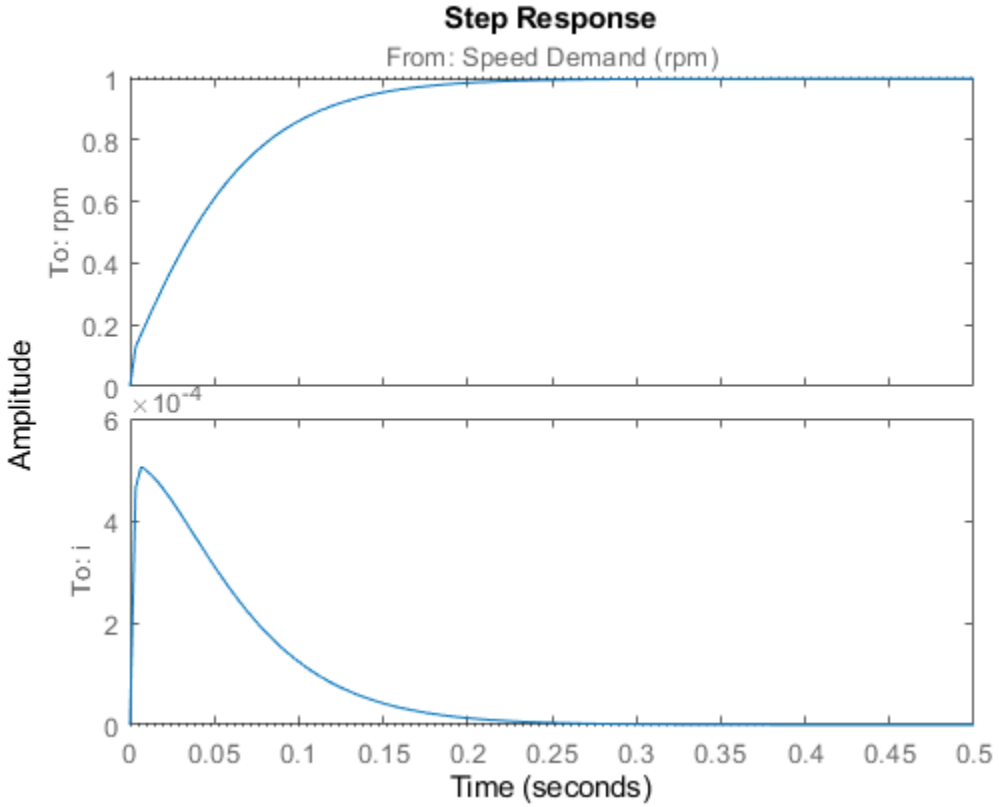
You can now tune the proportional and integral gains with `looptune`:

```
ST1 = systune(ST0,TR);
```

```
Final: Soft = 1.04, Hard = -Inf, Iterations = 40
```

This returns an updated description `ST1` containing the tuned gain values. To validate this design, plot the closed-loop response from speed demand to speed:

```
T1 = getIOTransfer(ST1,'Speed Demand (rpm)',{'rpm','i'});
figure
step(T1,0.5)
```



The response looks good in the linear domain so push the tuned gain values to Simulink and further validate the design in the nonlinear model.

```
writeBlockValue(ST1)
```

The nonlinear simulation results appear in Figure 3. The nonlinear behavior is far worse than the linear approximation. Figure 4 shows saturation and oscillations in the inner (current) loop.

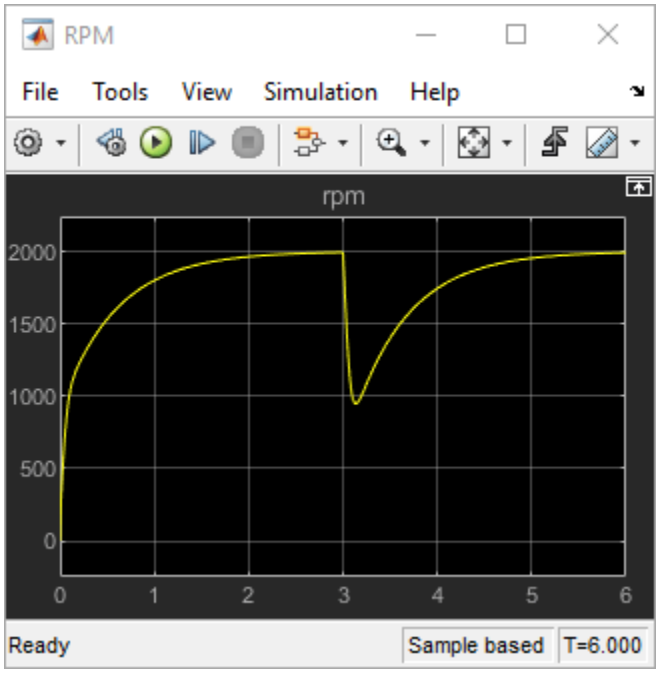


Figure 3: Nonlinear Simulation of Tuned Controller.

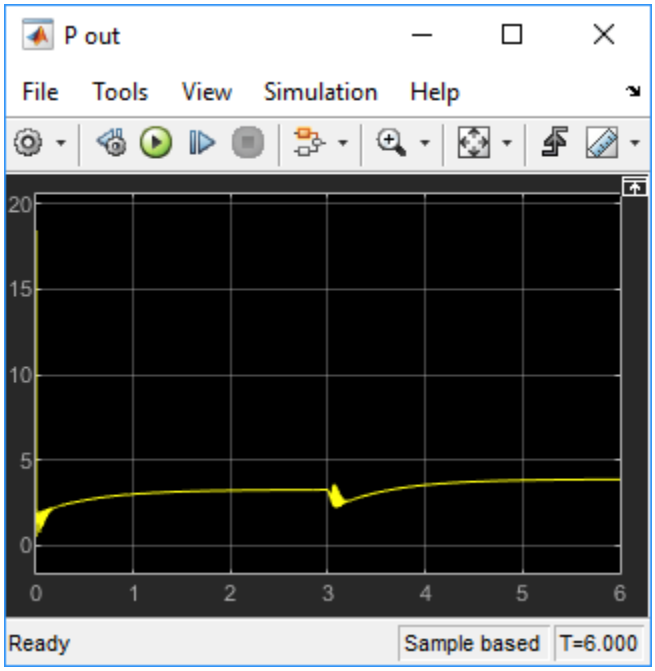


Figure 4: Current Controller Output.

Preventing Saturations

So far we have only specified a desired response time for the outer (speed) loop. This leaves `systune` free to allocate the control effort between the inner and outer loops. Saturation in the inner loop suggests that the proportional gain is too high and that some rebalancing is needed. One possible

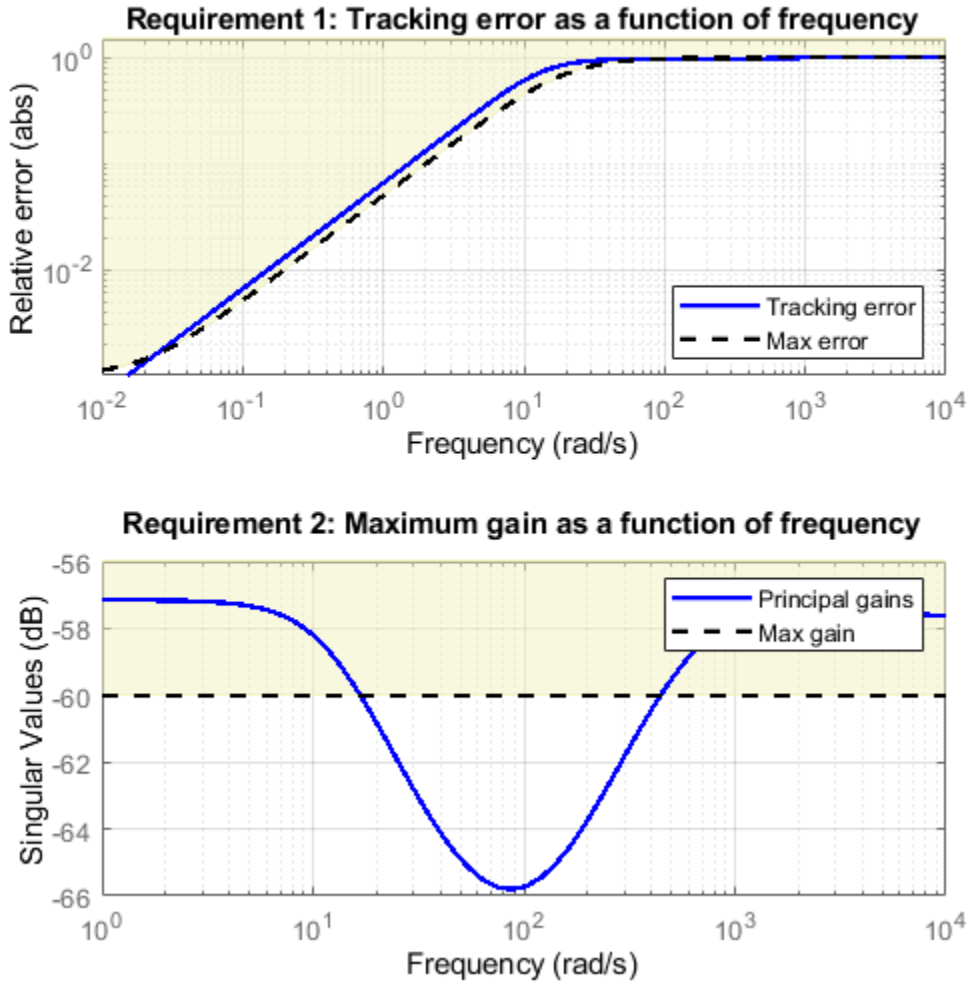
remedy is to explicitly limit the gain from the speed command to the "Current PID" output. For a speed reference of 2000 rpm and saturation limits of plus/minus 5, the average gain should not exceed 5/2000 = 0.0025. To be conservative, try keeping the gain from speed reference to "Current PID" below 0.001. To do this, add a gain constraint and retune the controller gains with both requirements in place.

```
% Mark the "Current PID" output as a point of interest
addPoint(ST0,'Current PID')

% Limit gain from speed demand to "Current PID" output to avoid saturation
MG = TuningGoal.Gain('Speed Demand (rpm)','Current PID',0.001);

% Retune with this additional goal
ST2 = systune(ST0,[TR,MG]);
```

```
Final: Soft = 1.39, Hard = -Inf, Iterations = 52
```

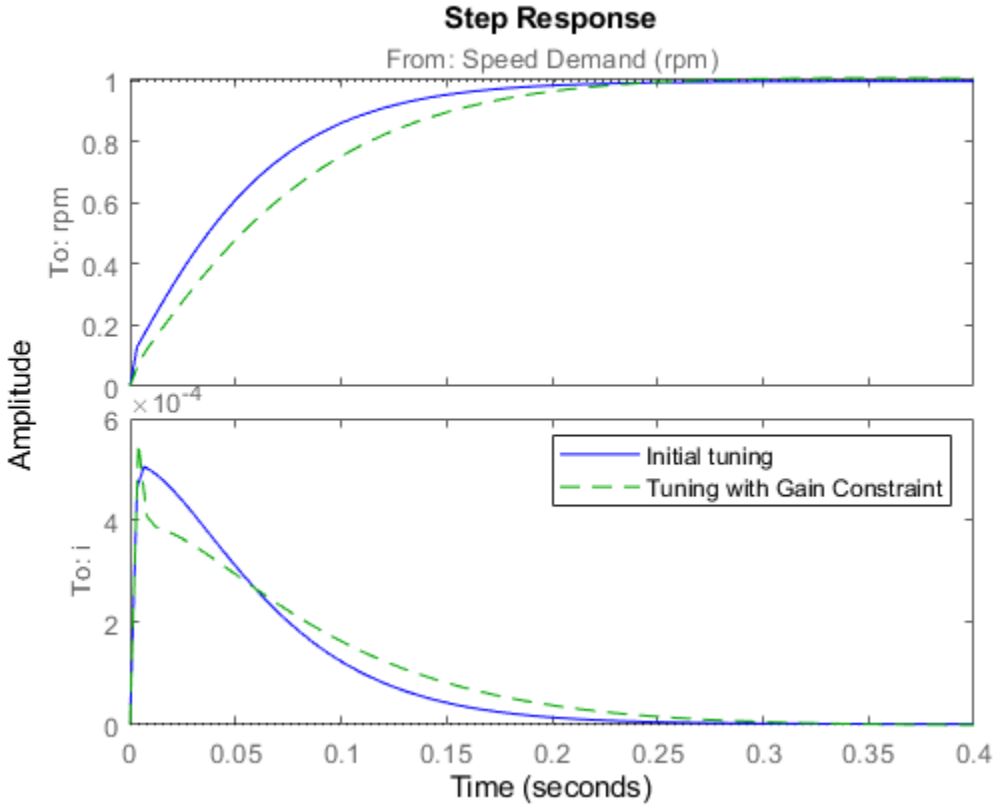
The final gain 1.39 indicates that the requirements are nearly but not exactly met (all requirements are met when the final gain is less than 1). Use `viewGoal` to inspect how the tuned controllers fare against each goal.

```
figure('Position',[100,100,560,550])
viewGoal([TR,MG],ST2)
```



Next compare the two designs in the linear domain.

```
T2 = getIOTransfer(ST2,'Speed Demand (rpm)',{'rpm','i'});
figure
step(T1,'b',T2,'g--',0.4)
legend('Initial tuning','Tuning with Gain Constraint')
```



The second design is less aggressive but still meets the response time requirement. A comparison of the tuned PID gains shows that the proportional gain in the current loop was reduced from 18 to about 2.

```
showTunable(ST1)  % initial tuning
```

```
Block 1: rct_linact/Current Controller/Current PID =

  Kp = 30

Name: Current_PID
P-only controller.

----------------------------------

Block 2: rct_linact/Speed Controller/Speed PID =

             1
  Kp + Ki * ---
             s

  with Kp = 0.391, Ki = 0.412

Name: Speed_PID
Continuous-time PI controller in parallel form.
```

```
showTunable(ST2)  % retuning
```

```
Block 1: rct_linact/Current Controller/Current PID =

  Kp = 2.41

Name: Current_PID
P-only controller.

-----------------------------------

Block 2: rct_linact/Speed Controller/Speed PID =

             1
  Kp + Ki * ---
             s

  with Kp = 0.433, Ki = 4.46

Name: Speed_PID
Continuous-time PI controller in parallel form.
```

To validate this new design, push the new tuned gain values to the Simulink model and simulate the response to a 2000 rpm speed demand and 500 N load disturbance. The simulation results appear in Figure 5 and the current controller output is shown in Figure 6.

```
writeBlockValue(ST2)
```

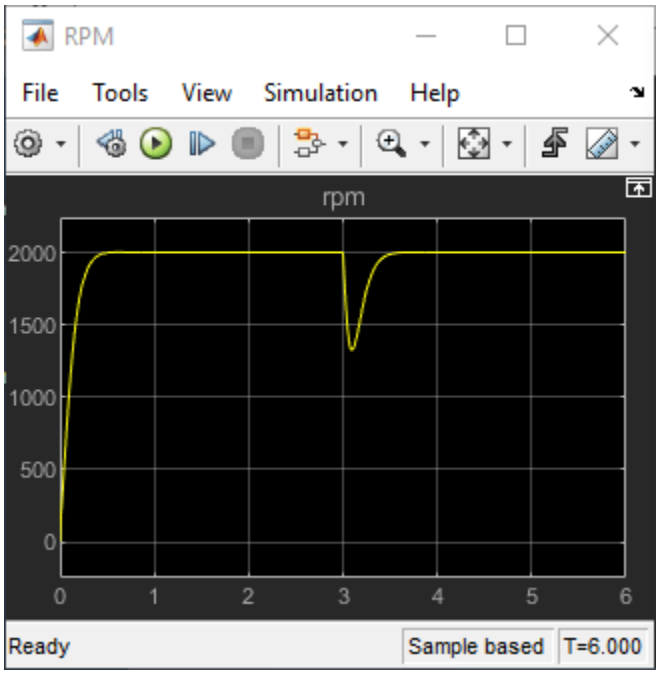


Figure 5: Nonlinear Response of Tuning with Gain Constraint.

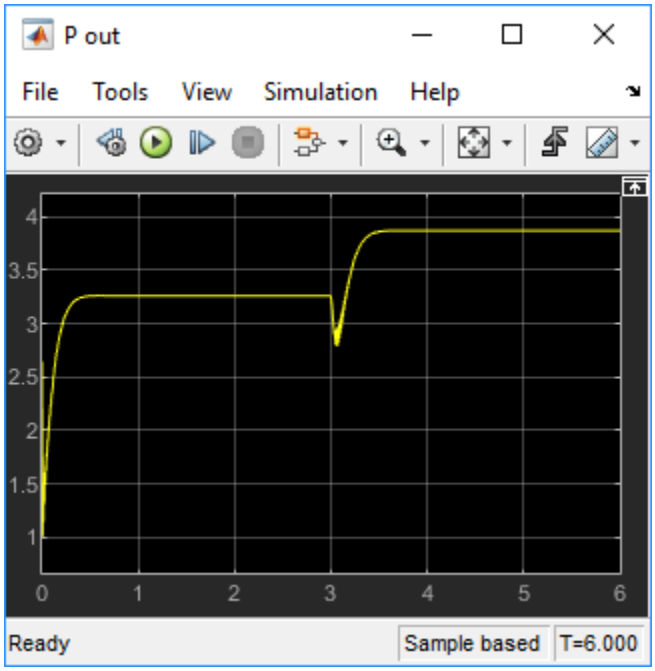


Figure 6: Current Controller Output.

The nonlinear responses are now satisfactory and the current loop is no longer saturating. The additional gain constraint has successfully rebalanced the control effort between the inner and outer loops.

See Also

`systune (slTuner)` | `slTuner` | `writeBlockValue` | `TuningGoal.Tracking` | `TuningGoal.Gain`

Related Examples

- “Control of a Linear Electric Actuator Using Control System Tuner” on page 13-89
- “Tune Control Systems in Simulink” on page 13-51
- “Tuning Control Systems with SYSTUNE”

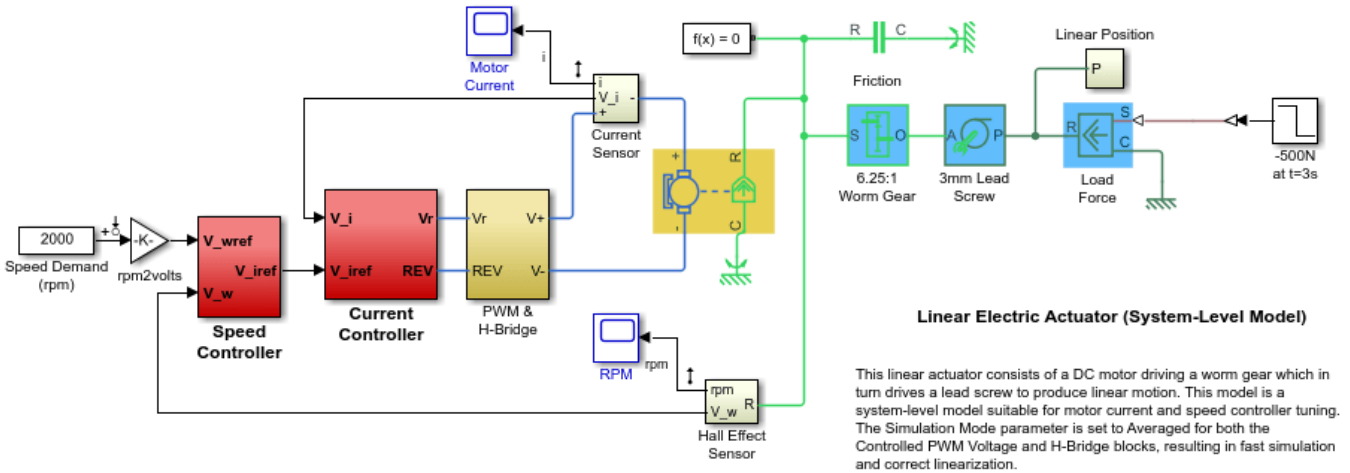
Control of a Linear Electric Actuator Using Control System Tuner

This example shows how to use the Control System Tuner app to tune the current and velocity loops in a linear electric actuator with saturation limits.

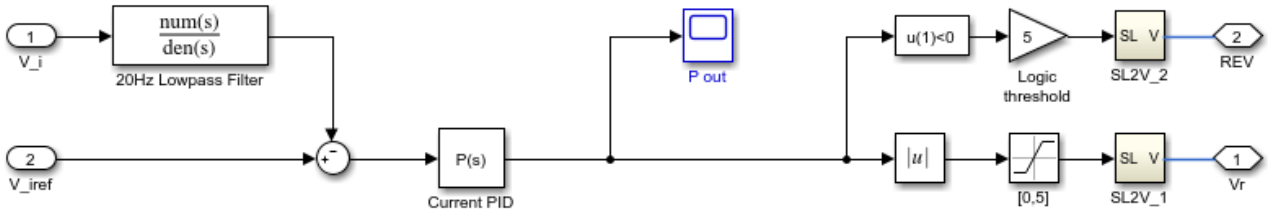
Linear Electric Actuator Model

Open the Simulink® model of the linear electric actuator:

```
open_system('rct_linact')
```



The electrical and mechanical components are modeled using Simulink and Simscape Electrical. The control system consists of two cascaded feedback loops controlling the driving current and angular speed of the DC motor.



Current Controller

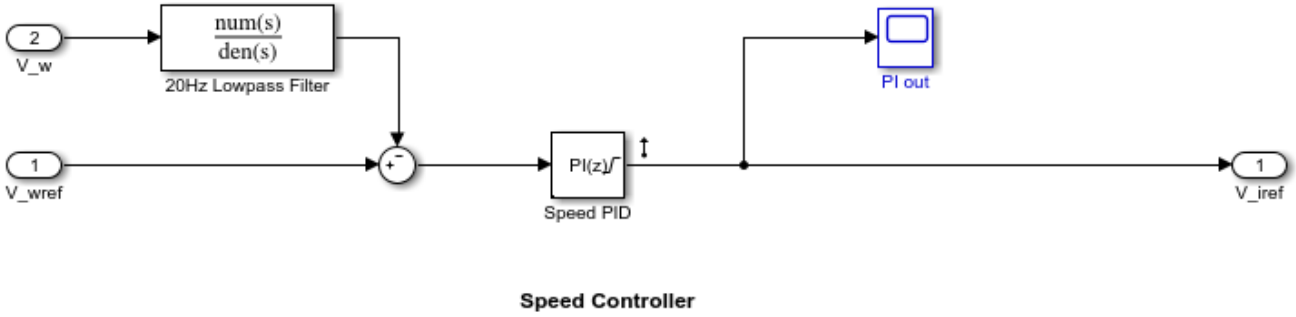


Figure 1: Current and Speed Controllers.

Note that the inner-loop (current) controller is a proportional gain while the outer-loop (speed) controller has proportional and integral actions. The output of both controllers is limited to plus/minus 5.

Design Specifications

We need to tune the proportional and integral gains to respond to a 2000 rpm speed demand in about 0.1 seconds with minimum overshoot. The initial gain settings in the model are P=50 and PI(s)=0.2+0.1/s and the corresponding response is shown in Figure 2. This response is too slow and too sensitive to load disturbances.

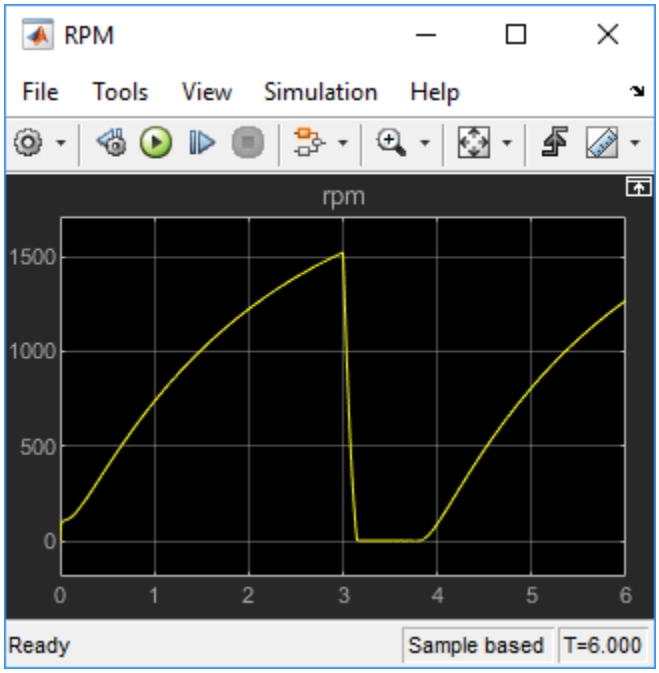


Figure 2: Untuned Response.

Control System Tuning

You can use Control System Tuner to jointly tune both feedback loops. First, open Control System Tuner from the Apps tab.

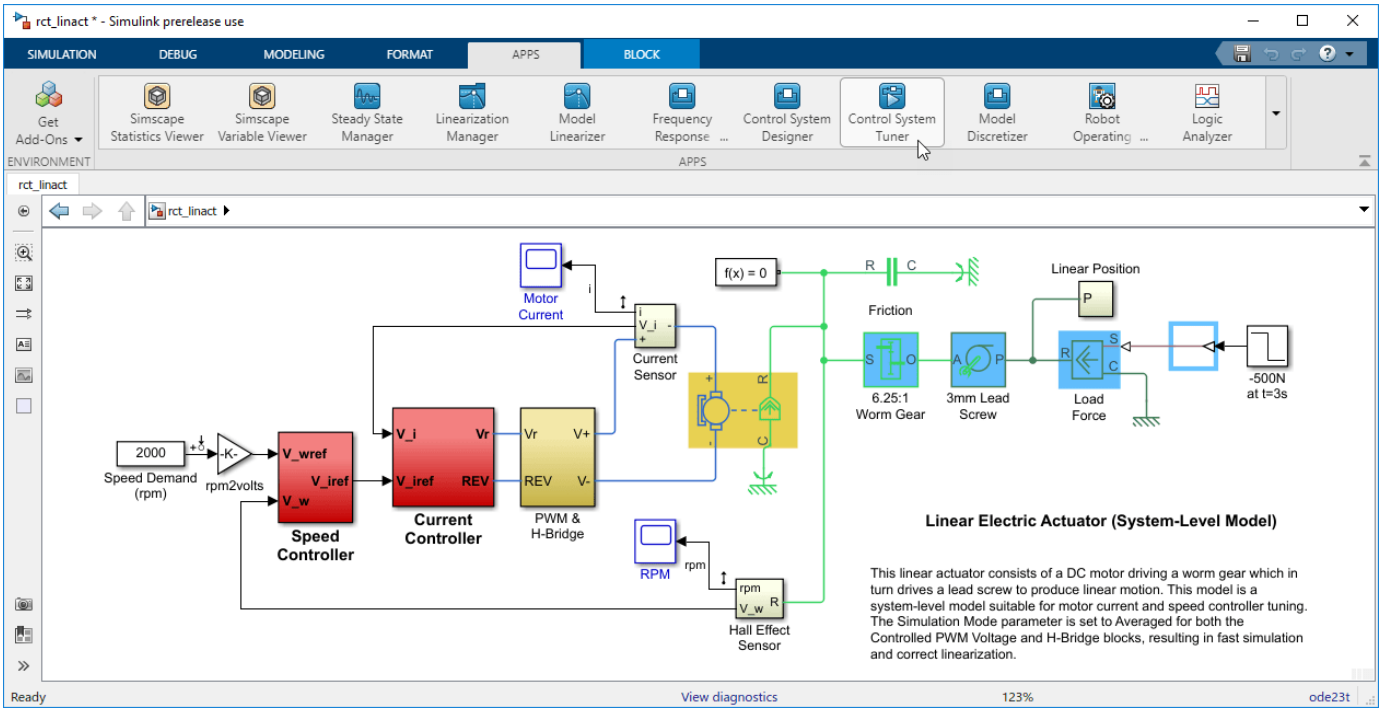


Figure 3: Opening Control System Tuner.

This opens Control System Tuner.

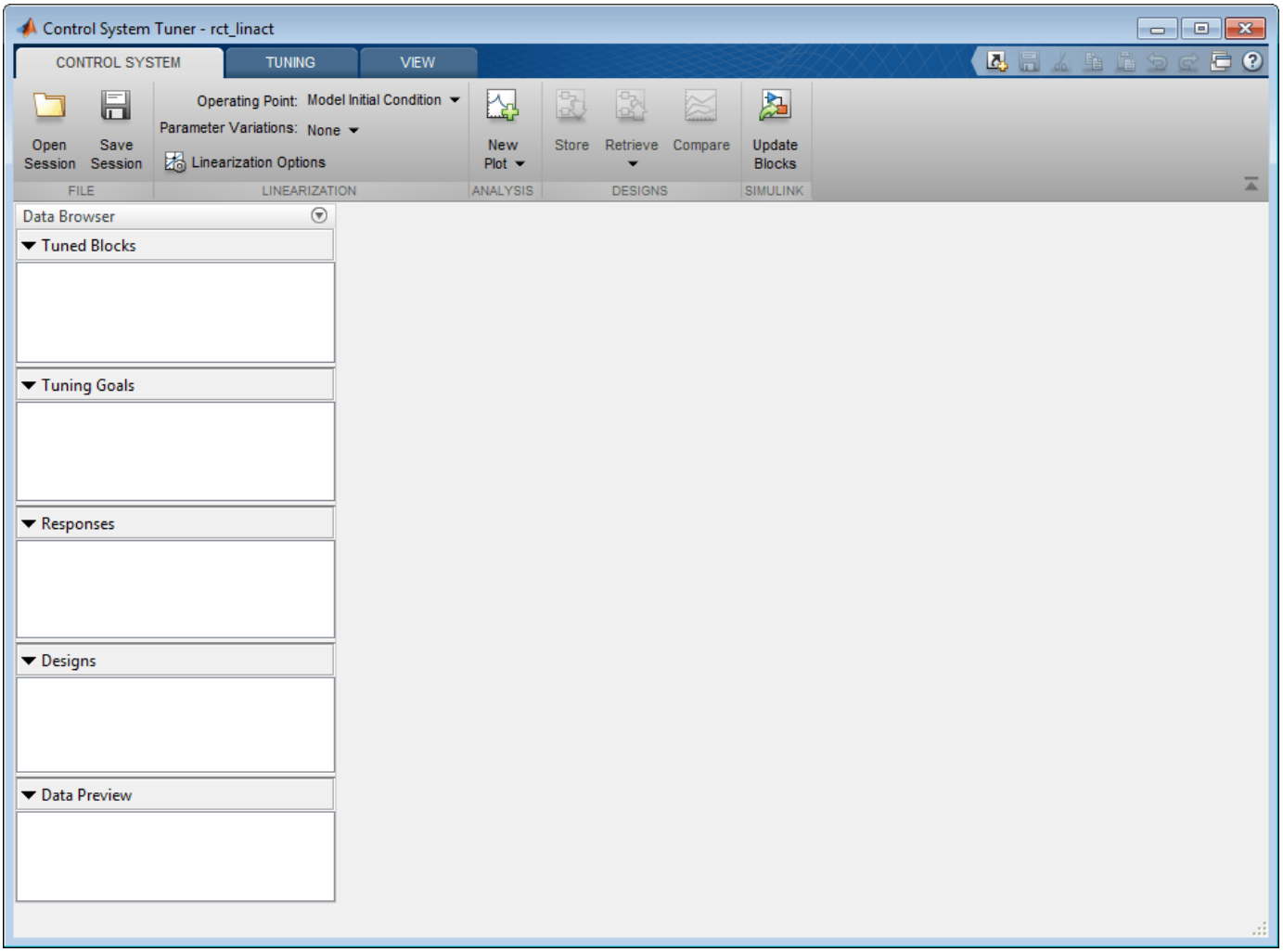


Figure 4: Control System Tuner.

You linearize the model at t=0.5 to avoid discontinuities in some derivatives at t=0. You can set the operating point in `Linearize At...`.

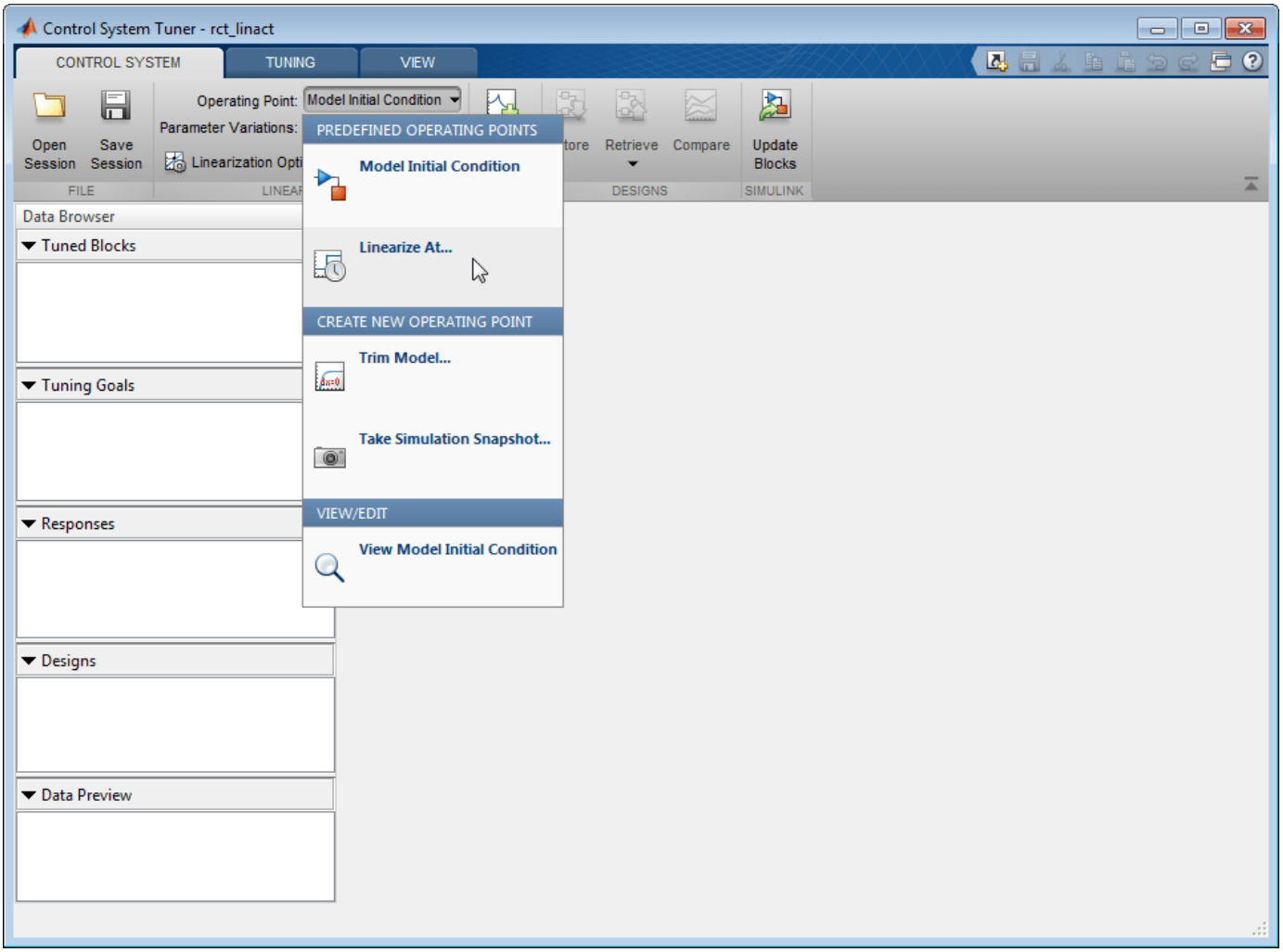


Figure 5: Setting Operating Point for Linearization.

Set the linearization snapshot time at t=0.5.

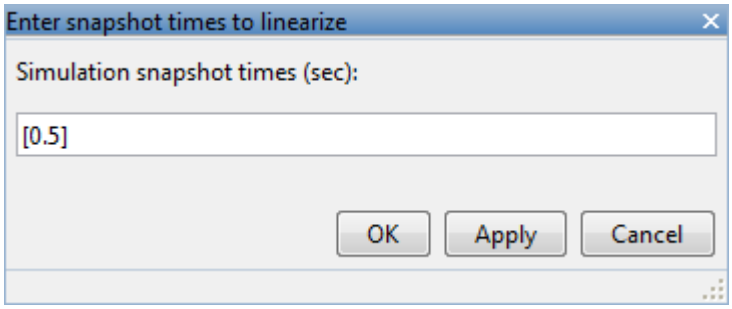


Figure 6: Setting the Linearization Snapshot Time.

In order to set the tuned blocks of the control system, open `Select Blocks` from `Tuning` tab.

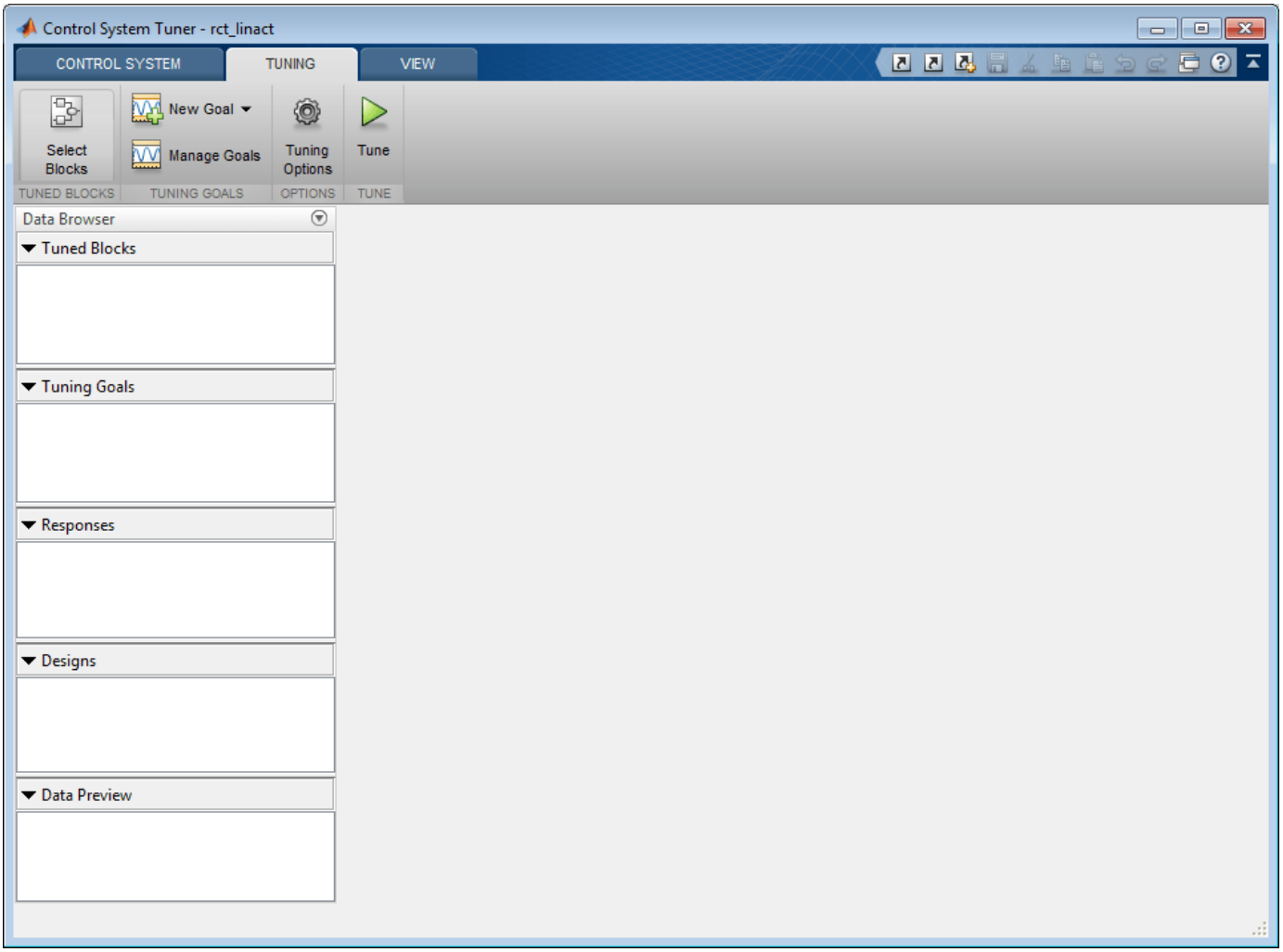


Figure 7: Tuning Tab of Control System Tuner.

This shows the editor for tuned blocks where you can `Add Blocks`.

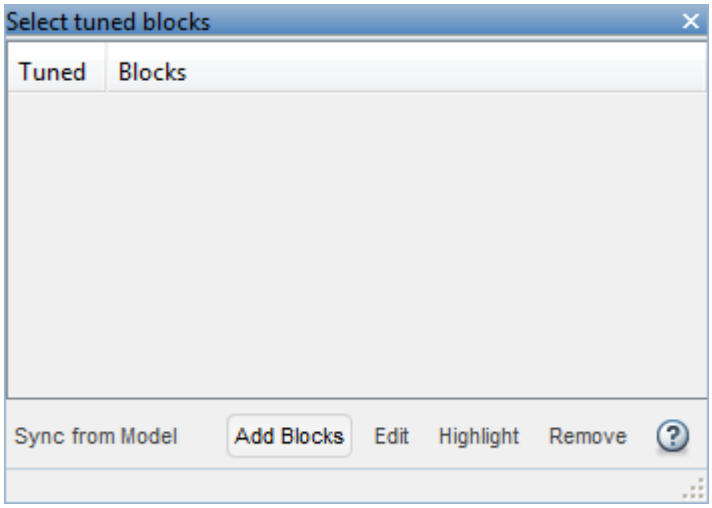


Figure 8: Editor for Tuned Blocks.

Set the tuned blocks `Current PID` and `Speed PID` by navigating through the tree on the left.

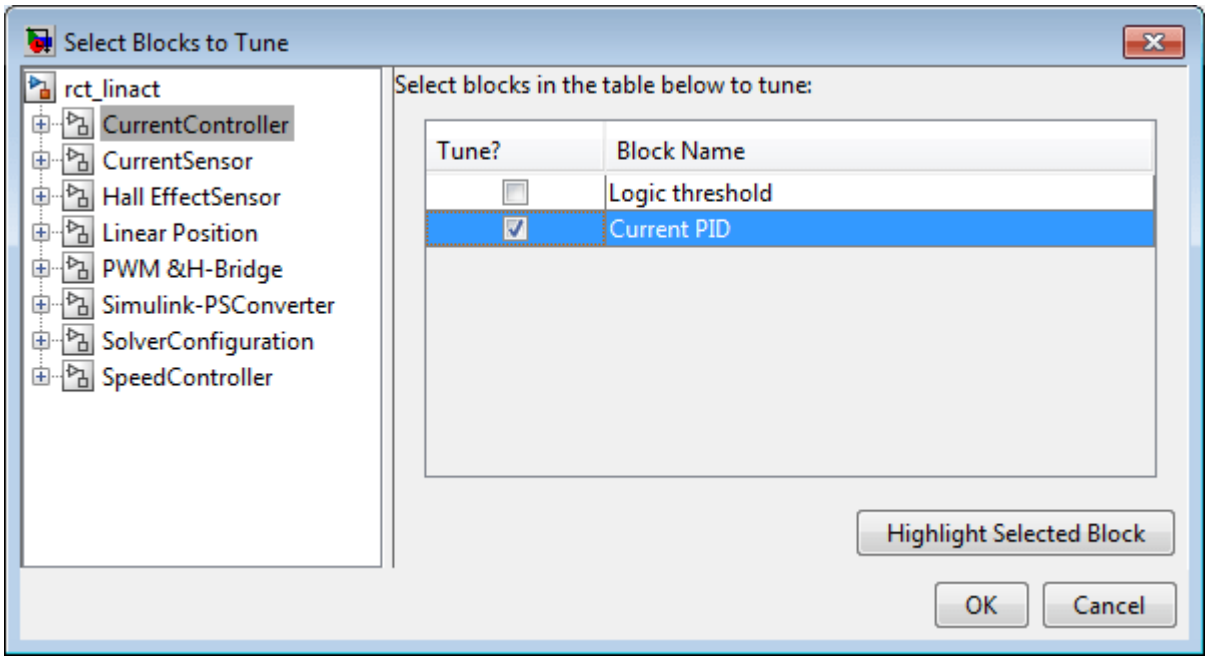


Figure 9: Selecting Tuned Block `Current PID`.

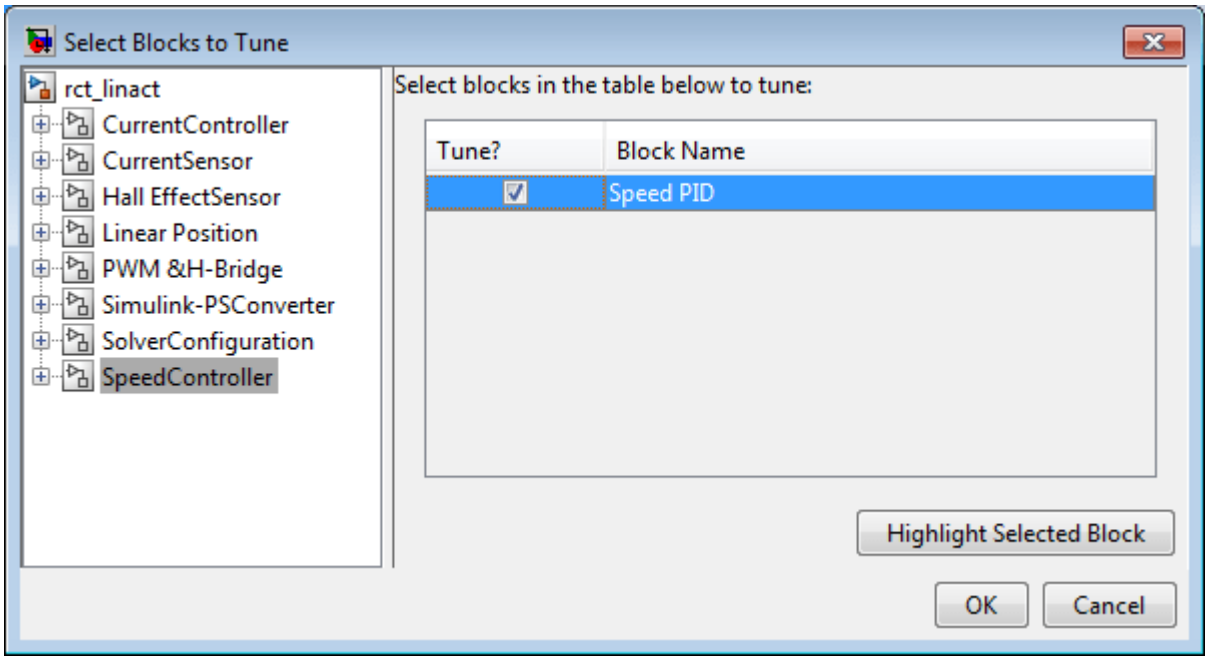


Figure 10: Selecting Tuned Block `Speed PID`.

Selected tuned blocks `Current PID` and `Speed PID` show in the editor for tuned blocks.

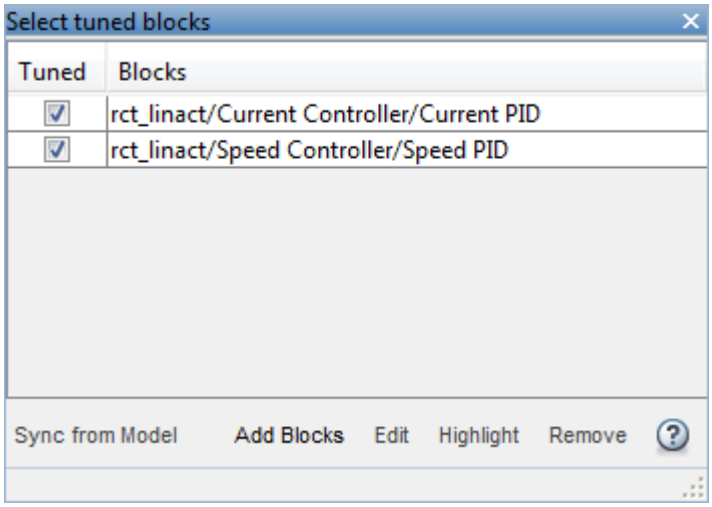


Figure 11: Editor Updated with Selected Tuned Blocks.

They also appear in the `Tuned Blocks` section of Data Browser on the left side of Control System Tuner.

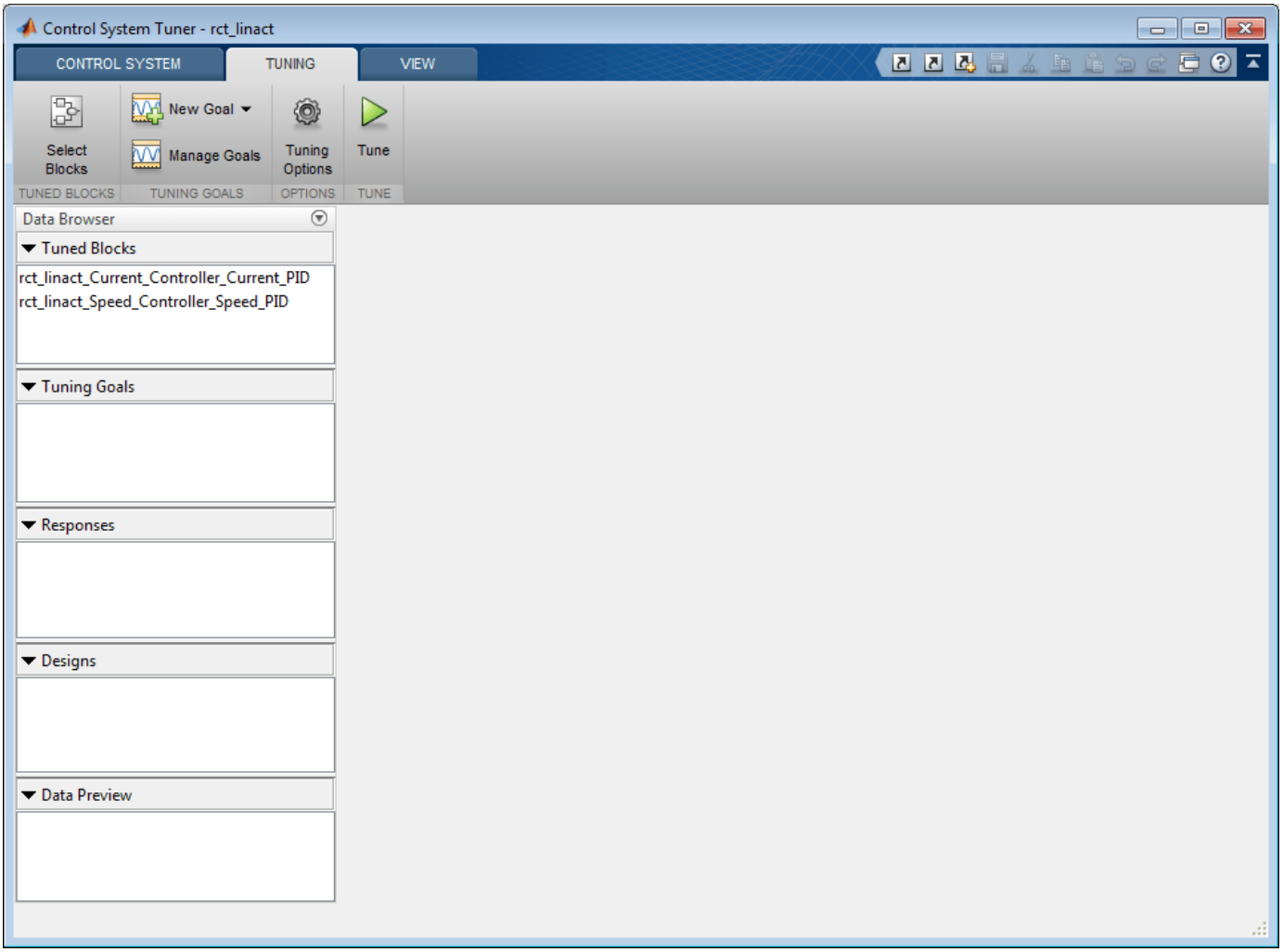


Figure 12: Updated Tuned Blocks in Control System Tuner.

Next specify the tracking goal that the DC motor should follow a 2000 rpm speed demand in 0.1 seconds. See different types of goals under `New Goal` and select `Reference Tracking`.

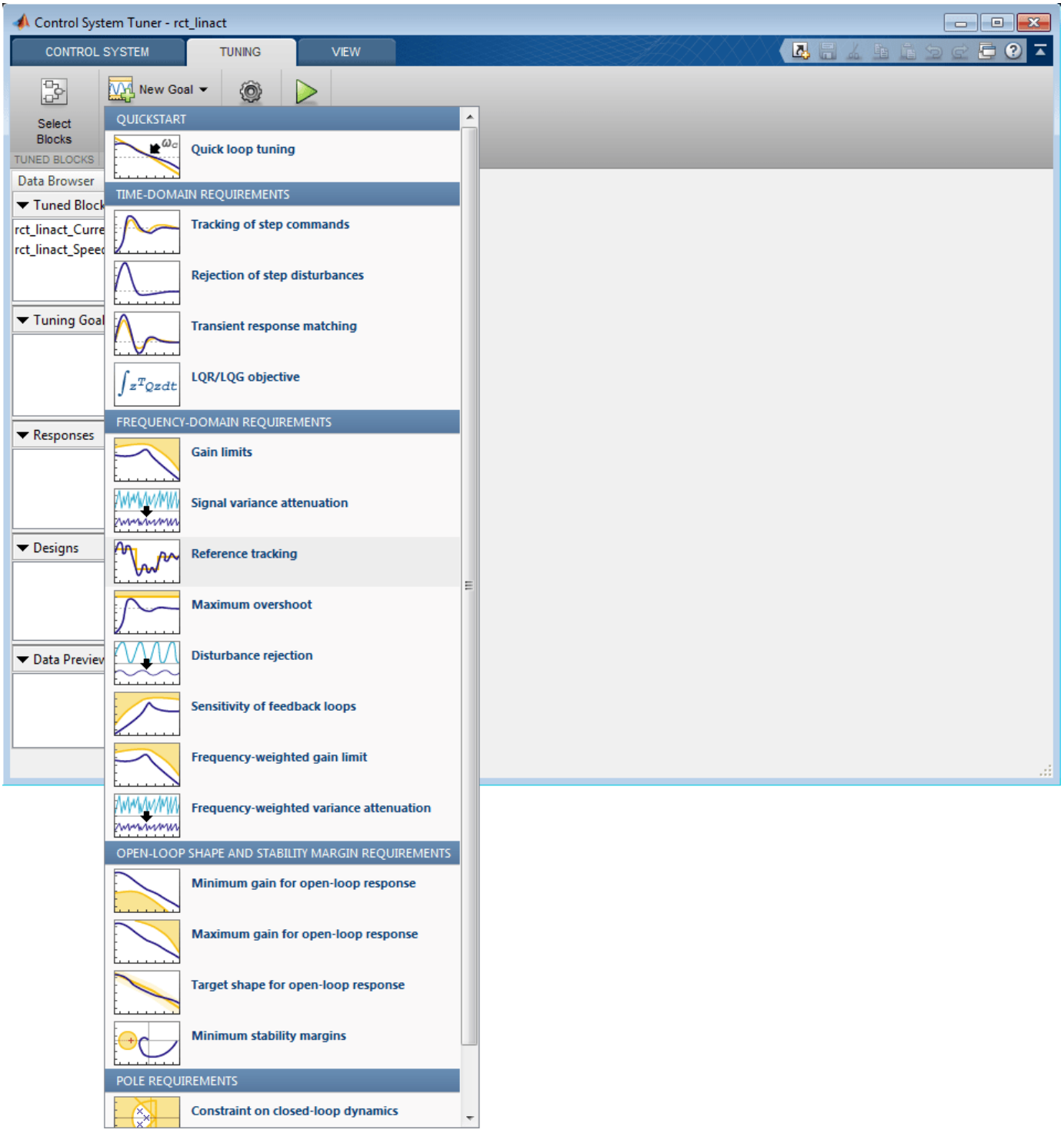


Figure 13: Available Goals for Selection in Control System Tuner.

Name the tracking goal as TR, specify the tracking goal from the reference input `rct_linact/Speed Demand(rpm)/1` to the reference-tracking output `rct_linact/Hall Effect Sensor/1[rpm]` with the response time 0.1 seconds.

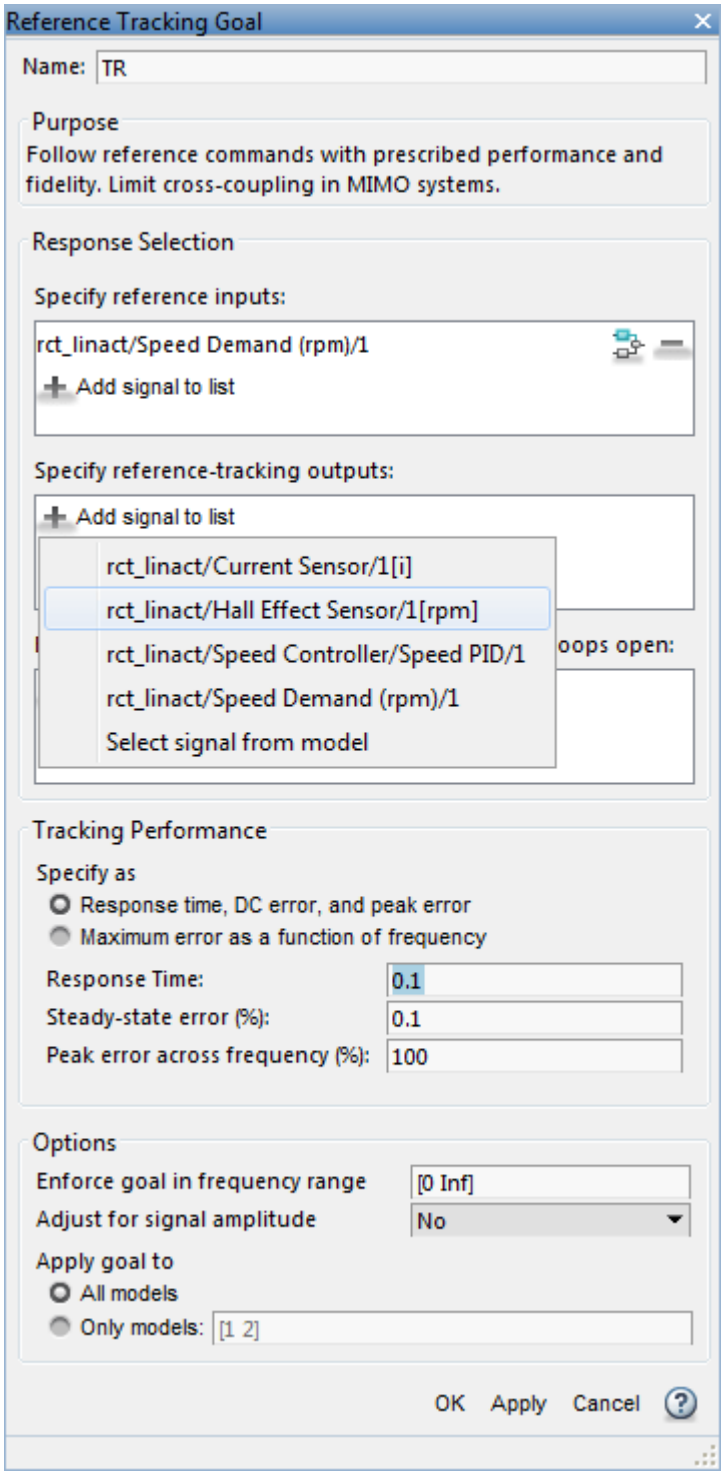


Figure 14: Reference Tracking Dialog in Control System Tuner.

The plot for specified tracking goal appears in Control System Tuner and `Tuning Goals` section of Data Browser on the left side is updated.

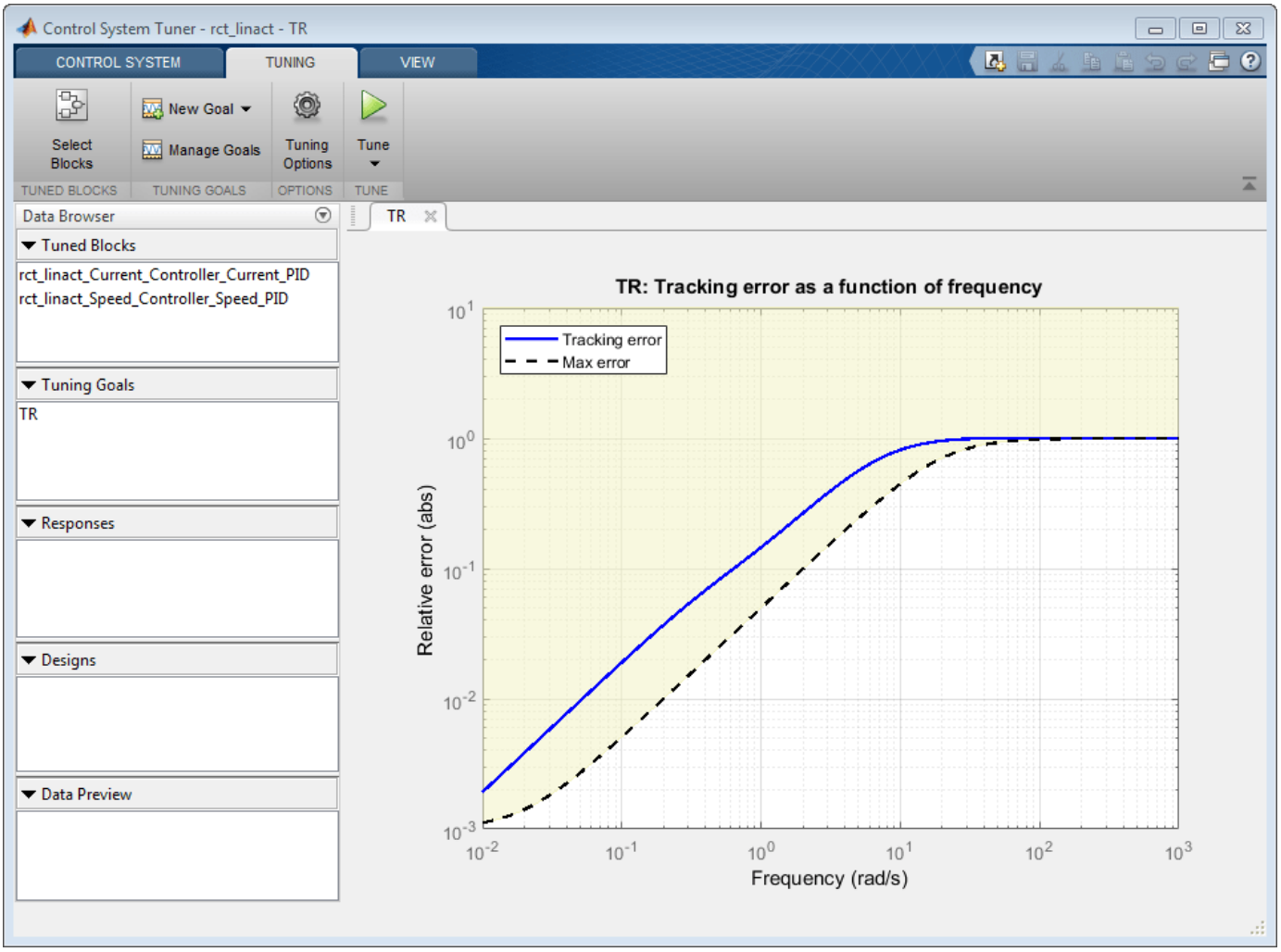


Figure 15: Tracking Tuning Goal in Control System Tuner.

You can now tune the proportional and integral gains with Control System Tuner from clicking Tune button. The plot for tracking goal is updated

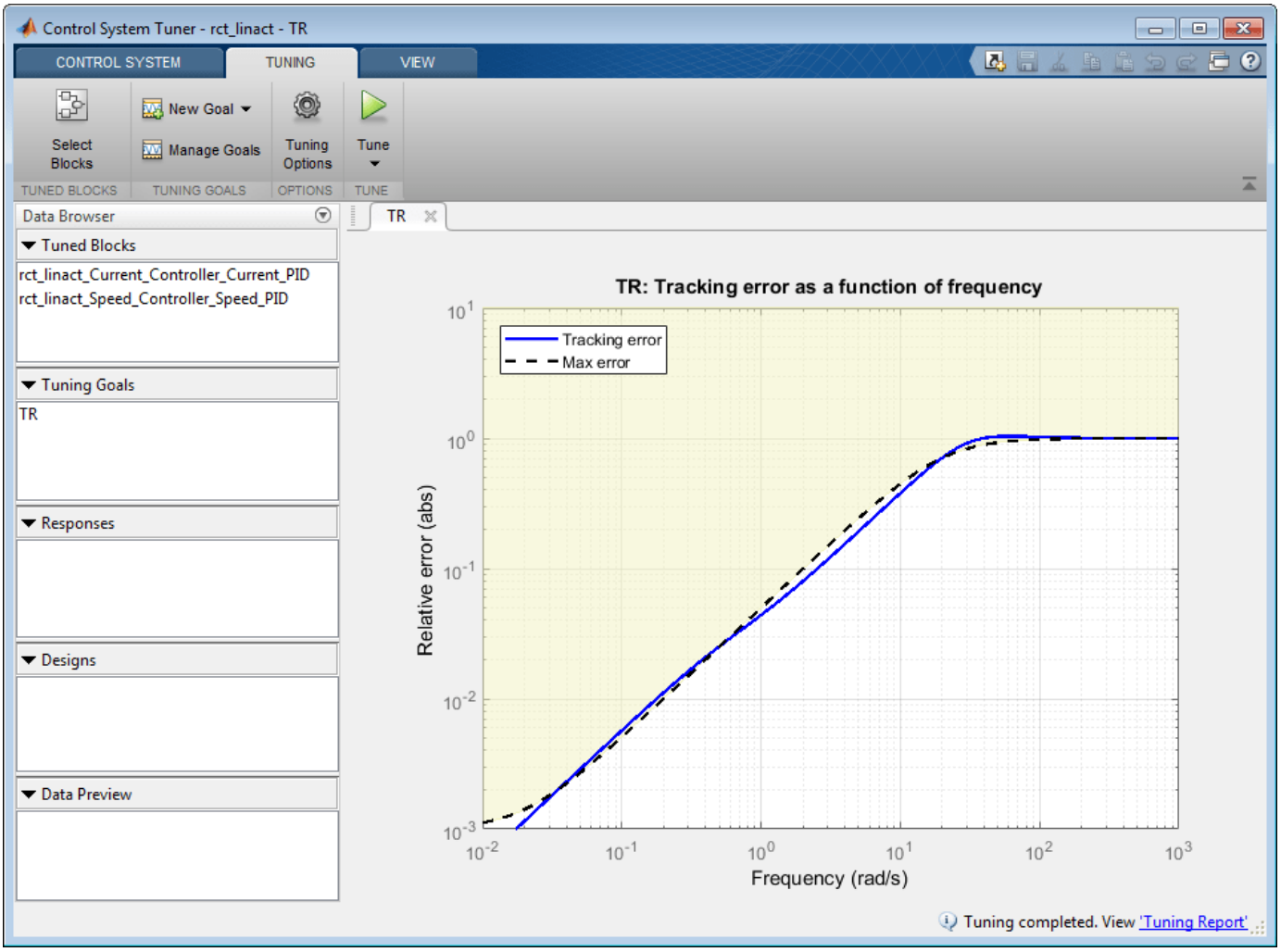


Figure 16: Updated Tracking Goal Plot with Tuned Blocks in Control System Tuner.

Tuned blocks are updated with the tuned gain values. To validate this design, plot the closed-loop response from speed demand to speed from `New Plot` of `Control System` Tab.

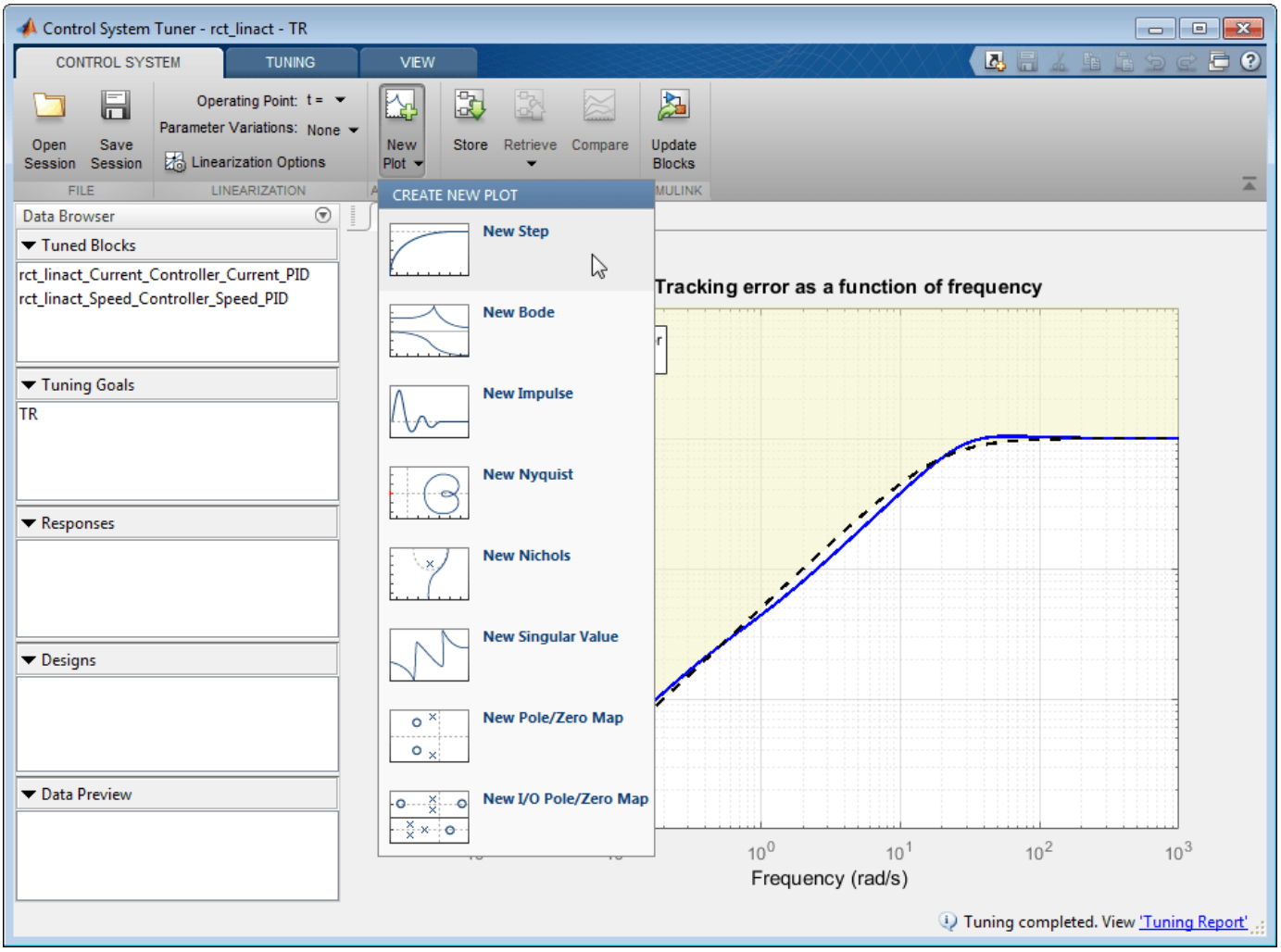


Figure 17: New Plot in Control System Tuner.

Specify the closed-loop response from speed demand to speed by the step plot dialog.

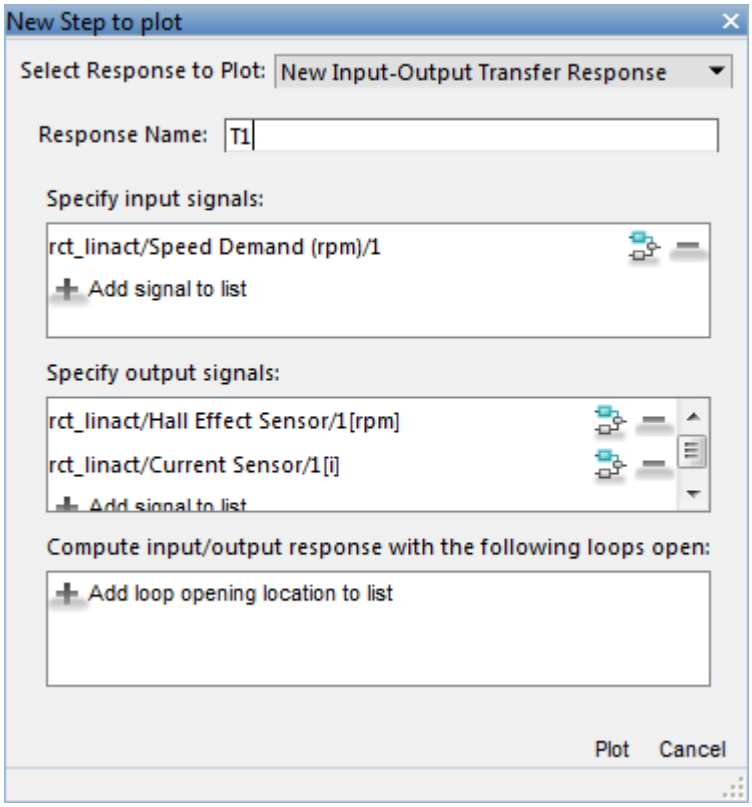


Figure 18: Step Plot Dialog in Control System Tuner.

You see the step plot of the response in Control System Tuner.

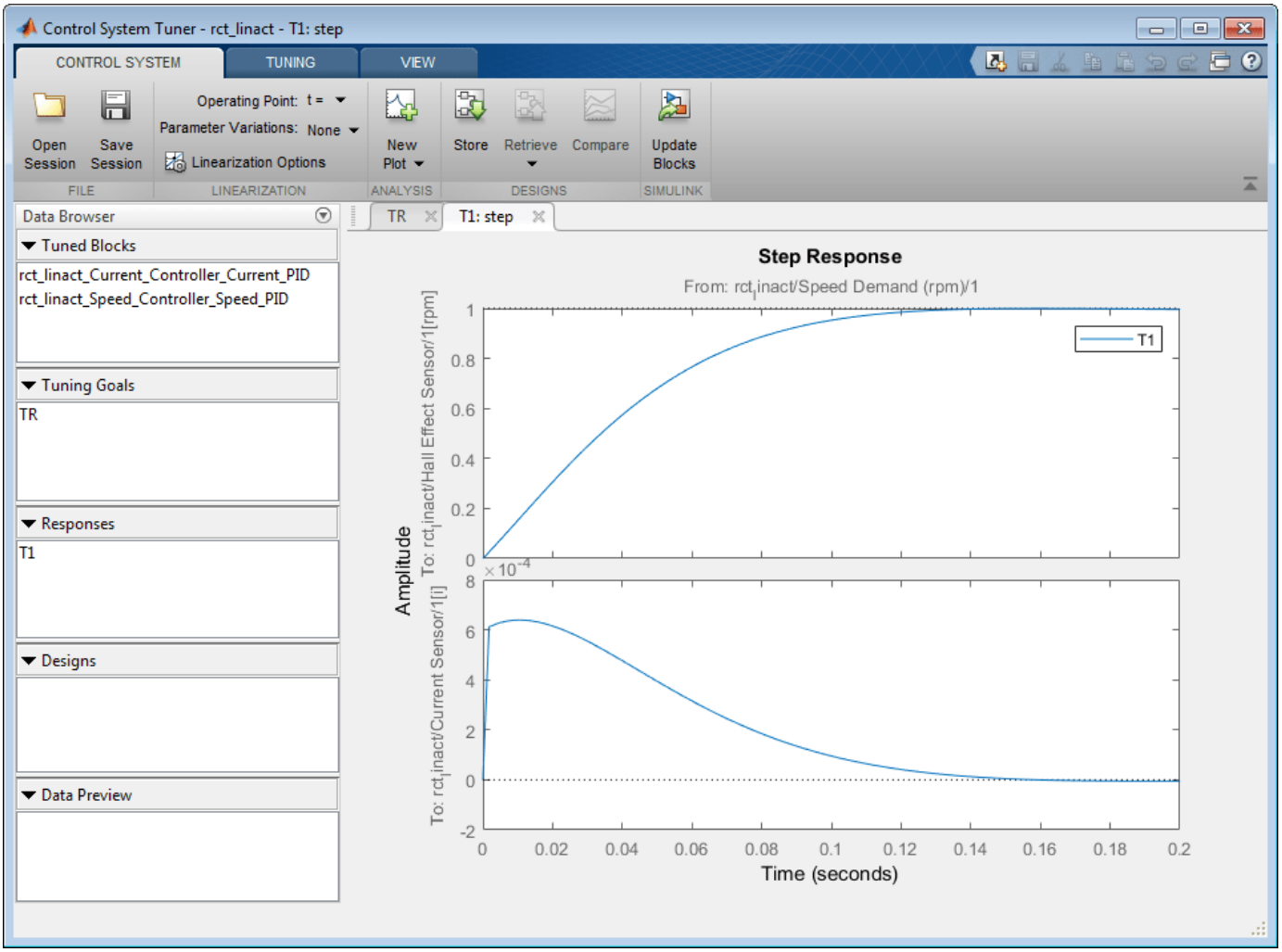


Figure 19: Step Plot in Control System Tuner.

The response looks good in the linear domain so first store the current design by clicking `Store` and push the tuned gain values to Simulink by clicking `Update Blocks` and further validate the design in the nonlinear model.

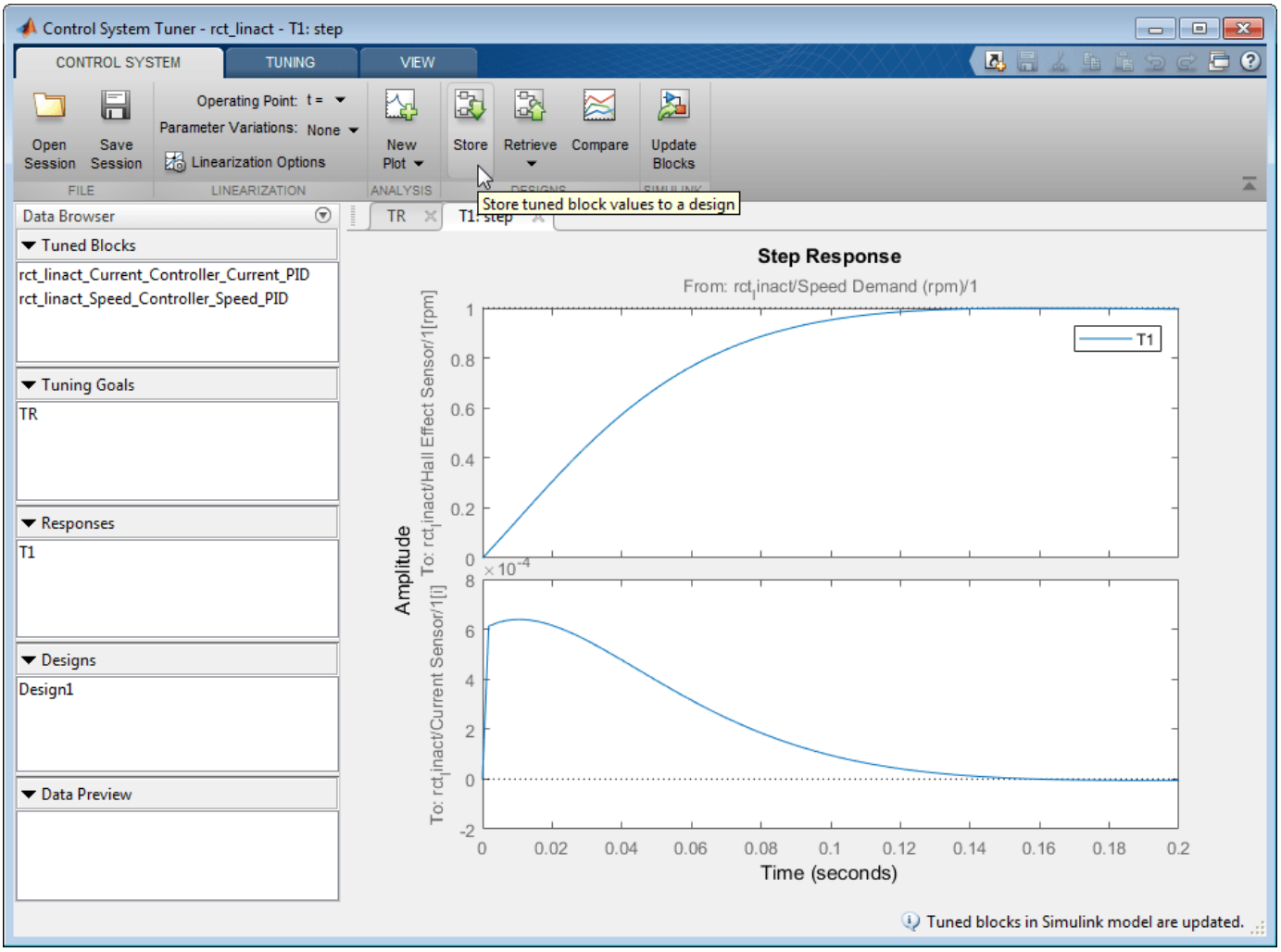


Figure 20: Stored Values of Tuned Blocks in Control System Tuner.

The nonlinear simulation results appear in Figure 21. The nonlinear behavior is far worse than the linear approximation, a discrepancy that can be traced to saturations in the inner loop (see Figure 22).

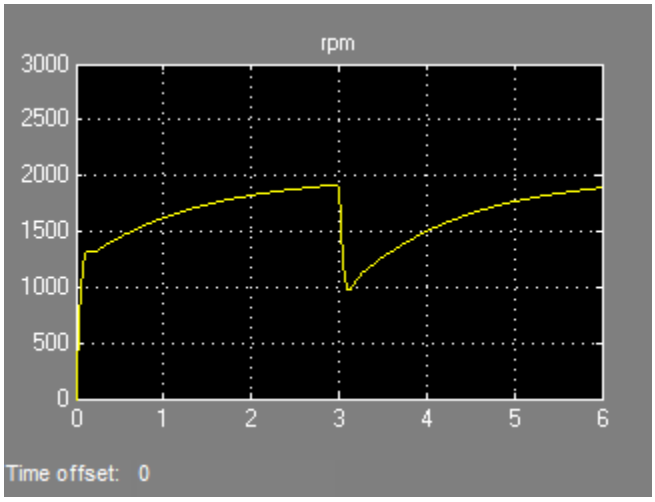


Figure 21: Nonlinear Simulation of Tuned Controller.

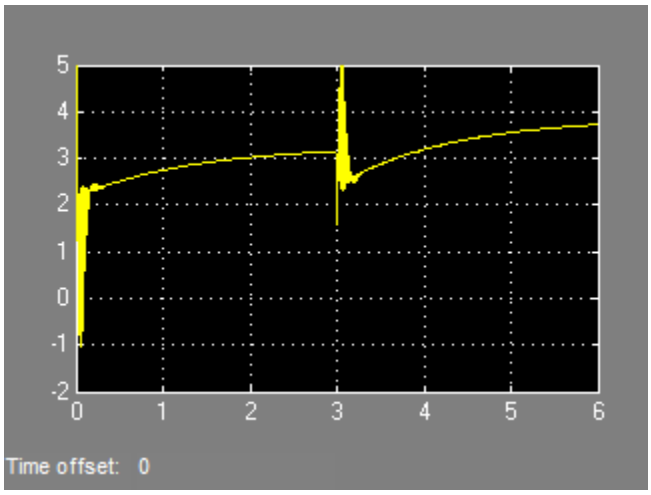


Figure 22: Current Controller Output (limited to plus/minus 5).

Preventing Saturations

So far we have only specified a desired response time for the outer (speed) loop. This leaves `systune` free to allocate the control effort between the inner and outer loops. Saturations in the inner loop suggest that the proportional gain is too high and that some rebalancing is needed. One possible remedy is to explicitly limit the gain from the speed command to the outputs of the P and PI controllers. For a speed reference of 2000 rpm and saturation limits of plus/minus 5, the average gain should not exceed 5/2000 = 0.0025. To be conservative, we can try to keep the gain from speed reference to controller outputs below 0.001. To do this, add two gain requirements and retune the controller gains with all three requirements in place.

Limit gain from speed demand to control signals to avoid saturation by specifying two new goals from Tuning tab. You need to select control signals from Simulink model since they are not defined previously.

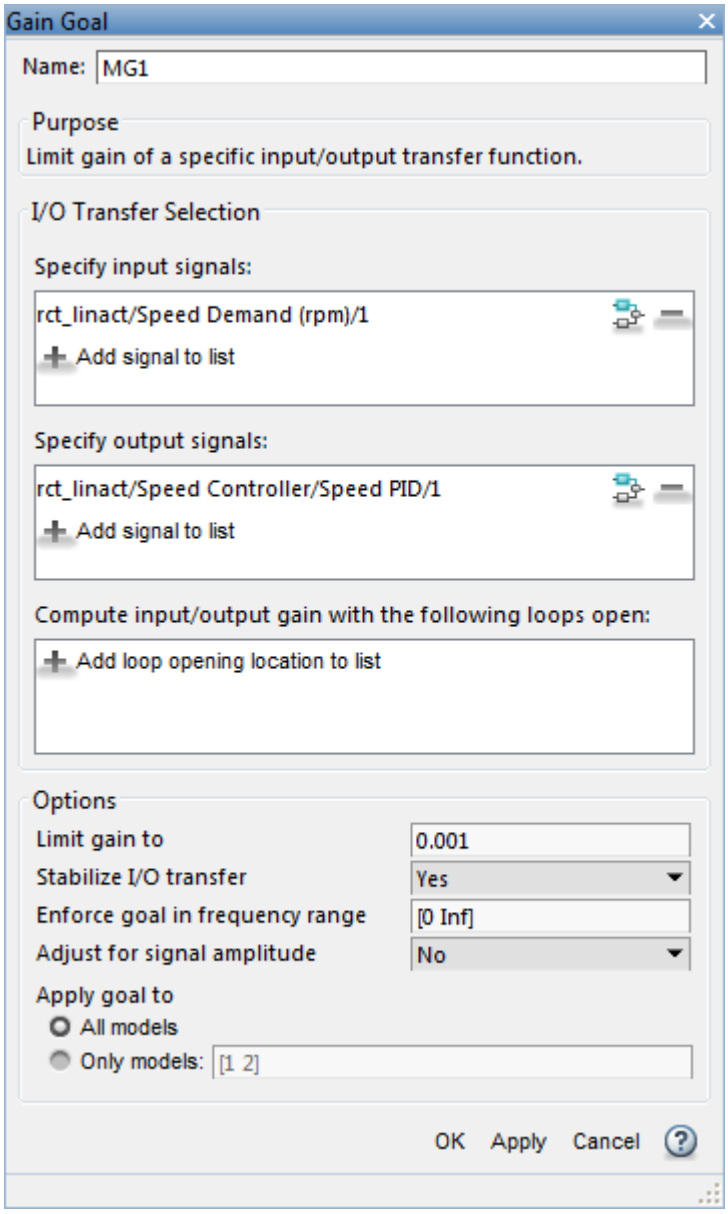


Figure 23: Gain Goal Dialog from Speed Demand to Control Signal of Speed PID.

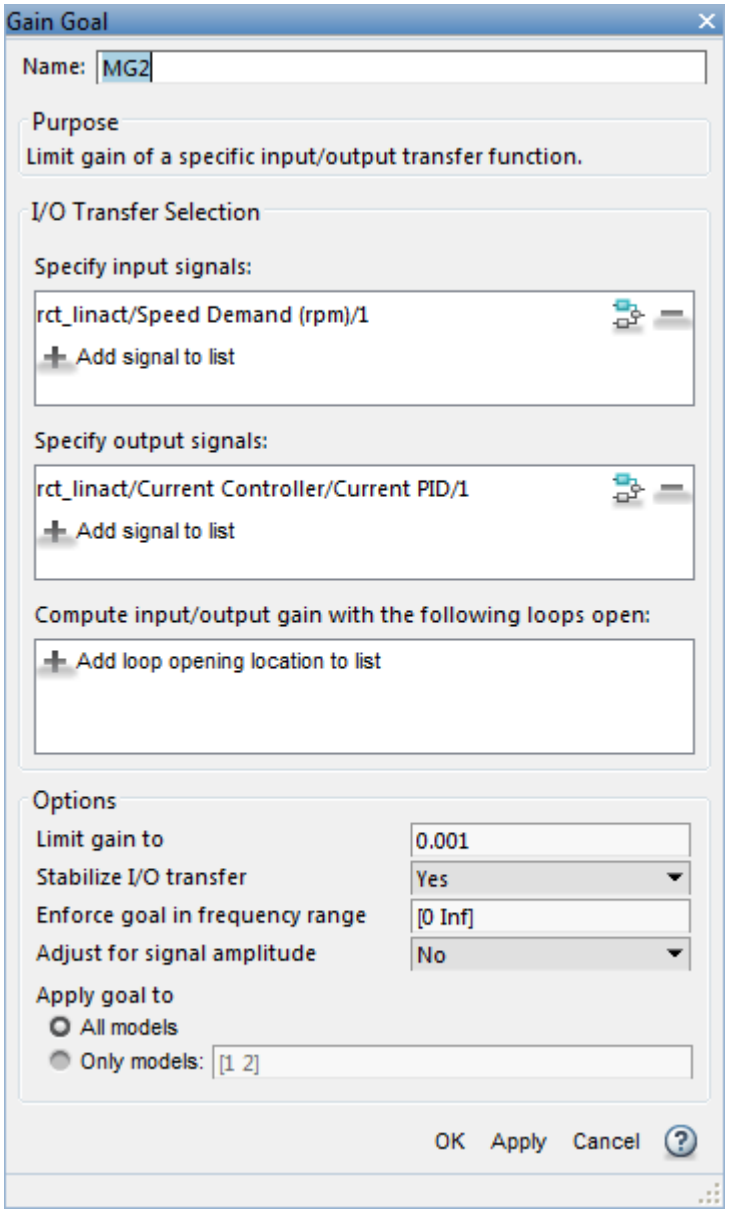


Figure 24: Gain Goal Dialog from Speed Demand to Control Signal of Current PID.

New gain goals appear in Tuning Goals section of Control System Tuner.

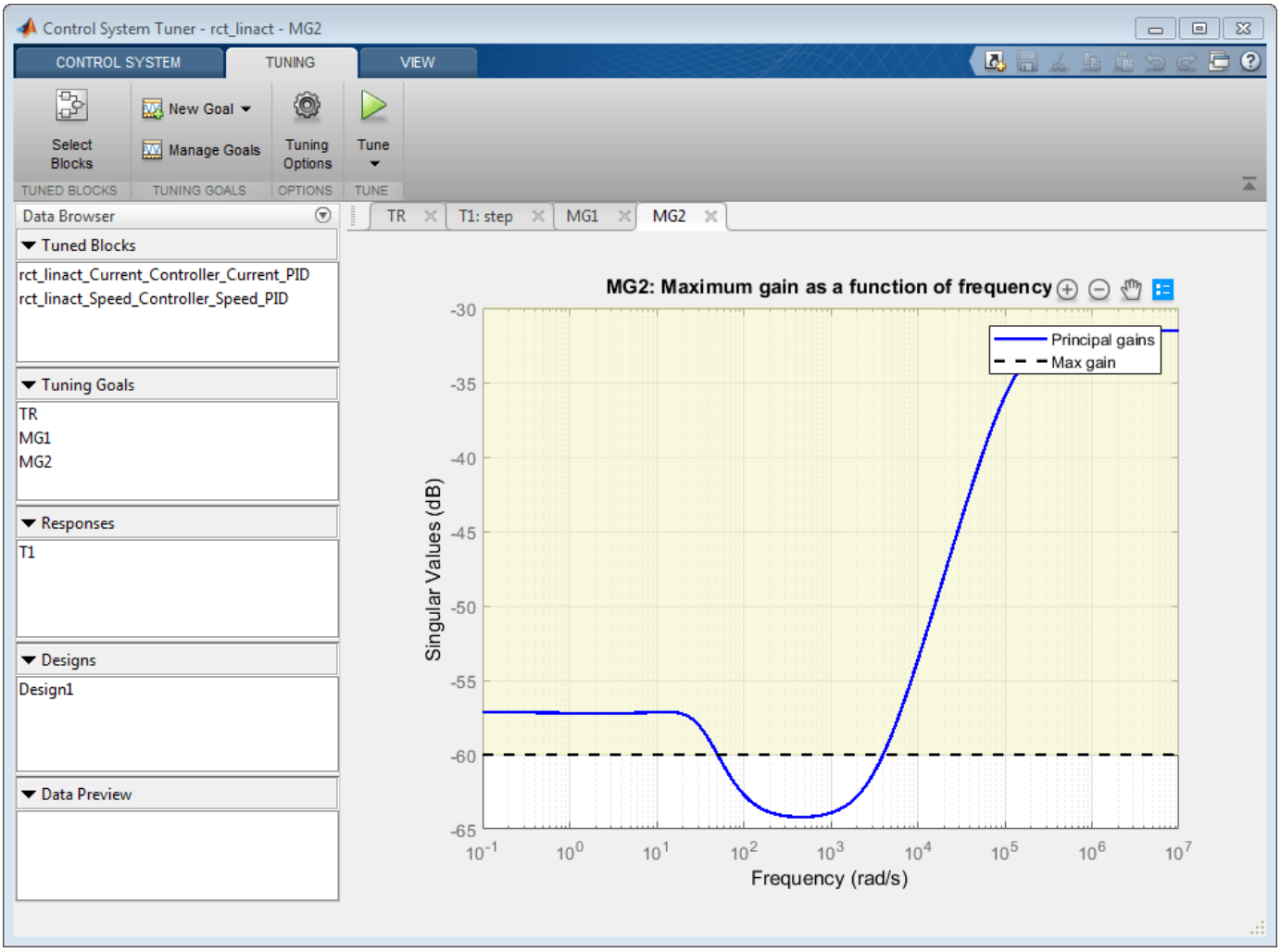


Figure 25: Two Gain Goals Added to Control System Tuner.

Retune with these additional requirements. Tuning Report accessed at the bottom right of the tool shows the worst gain 1.39 indicating that the requirements are nearly but not exactly met (all requirements are met when the final gain is less than 1).

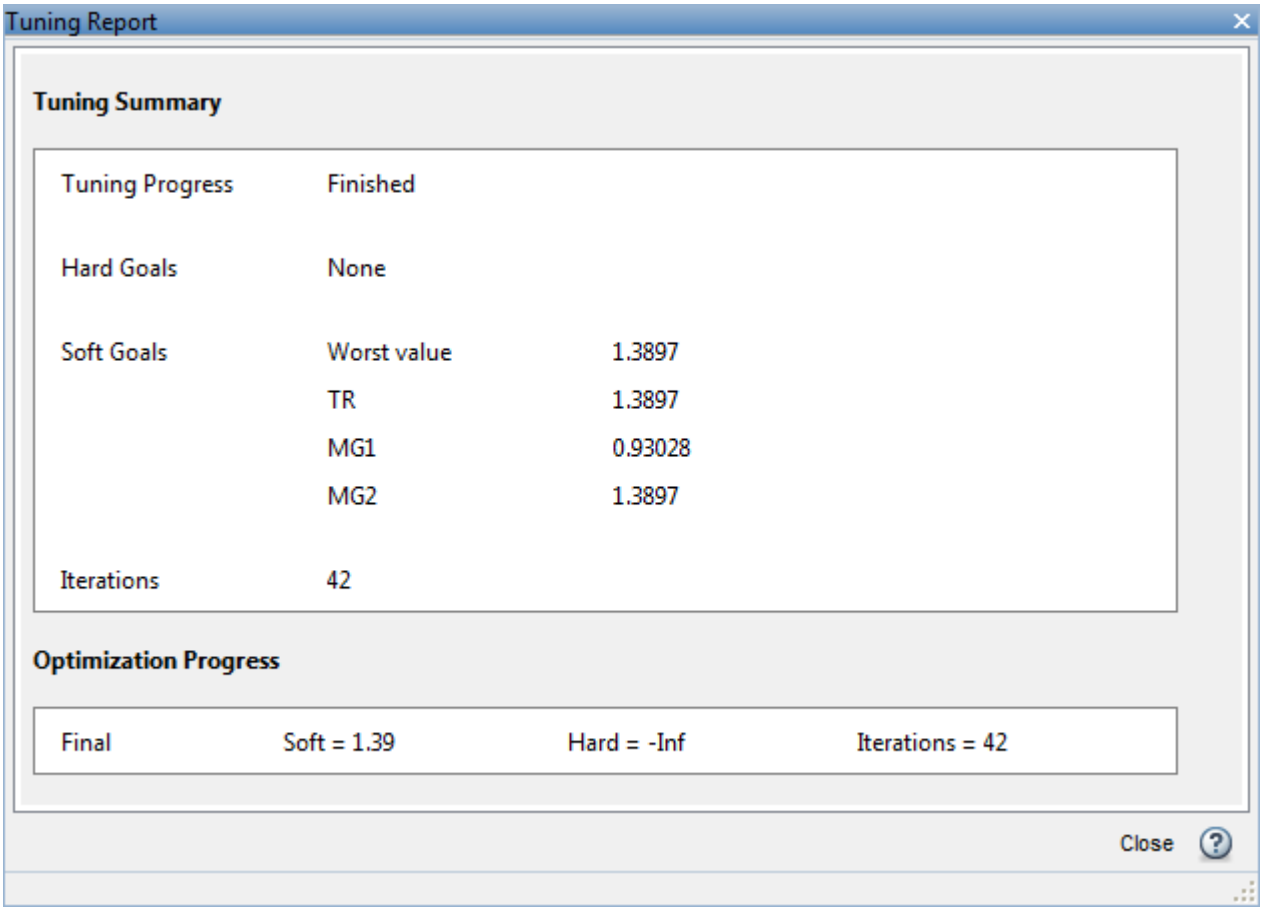


Figure 26: Tuning Report After Retuning.

Next compare the two designs in the linear domain by clicking `Compare` in `Control System` tab.

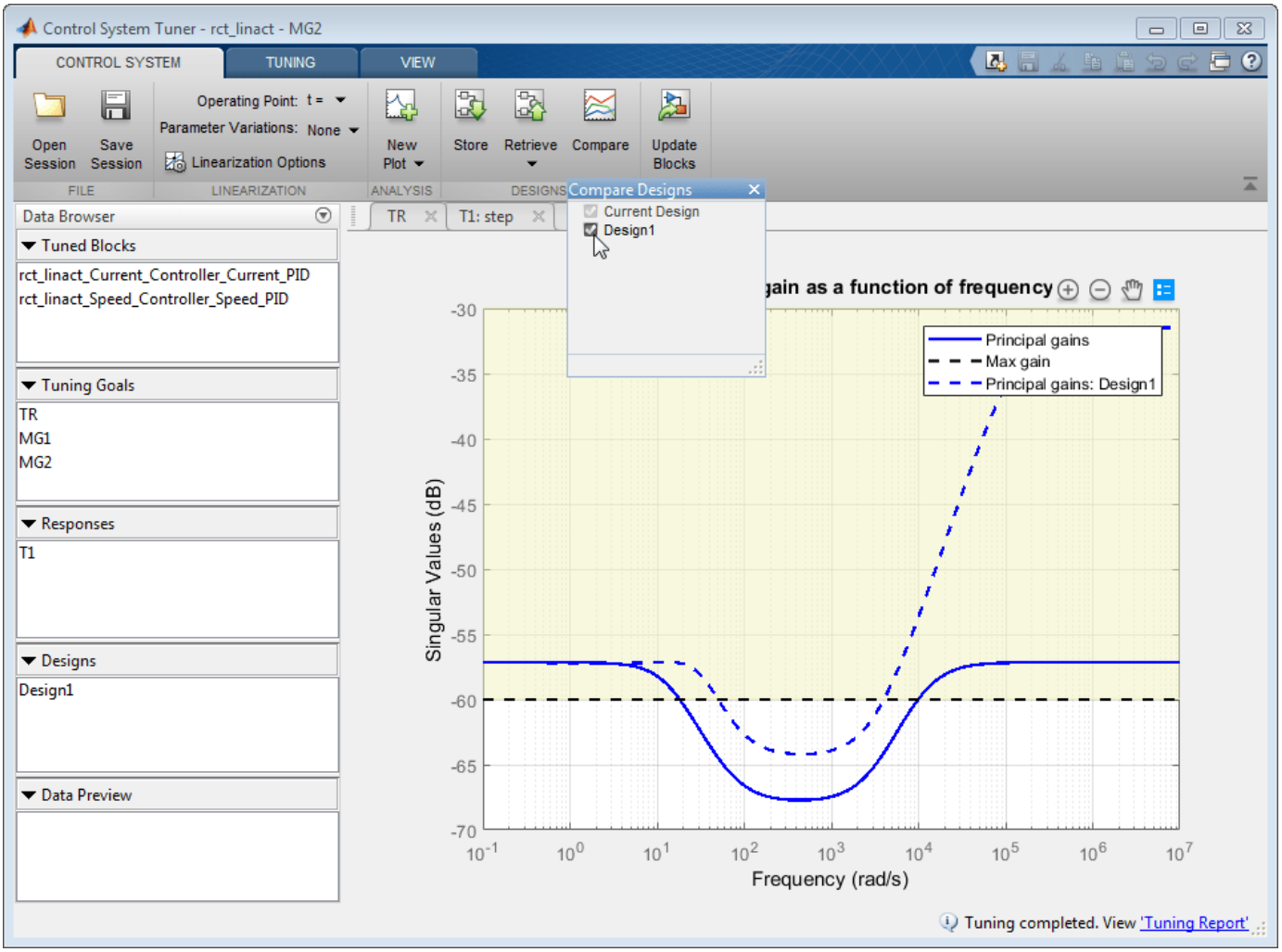


Figure 27: Comparing Two Designs.

The second design is less aggressive but still meets the response time requirement.

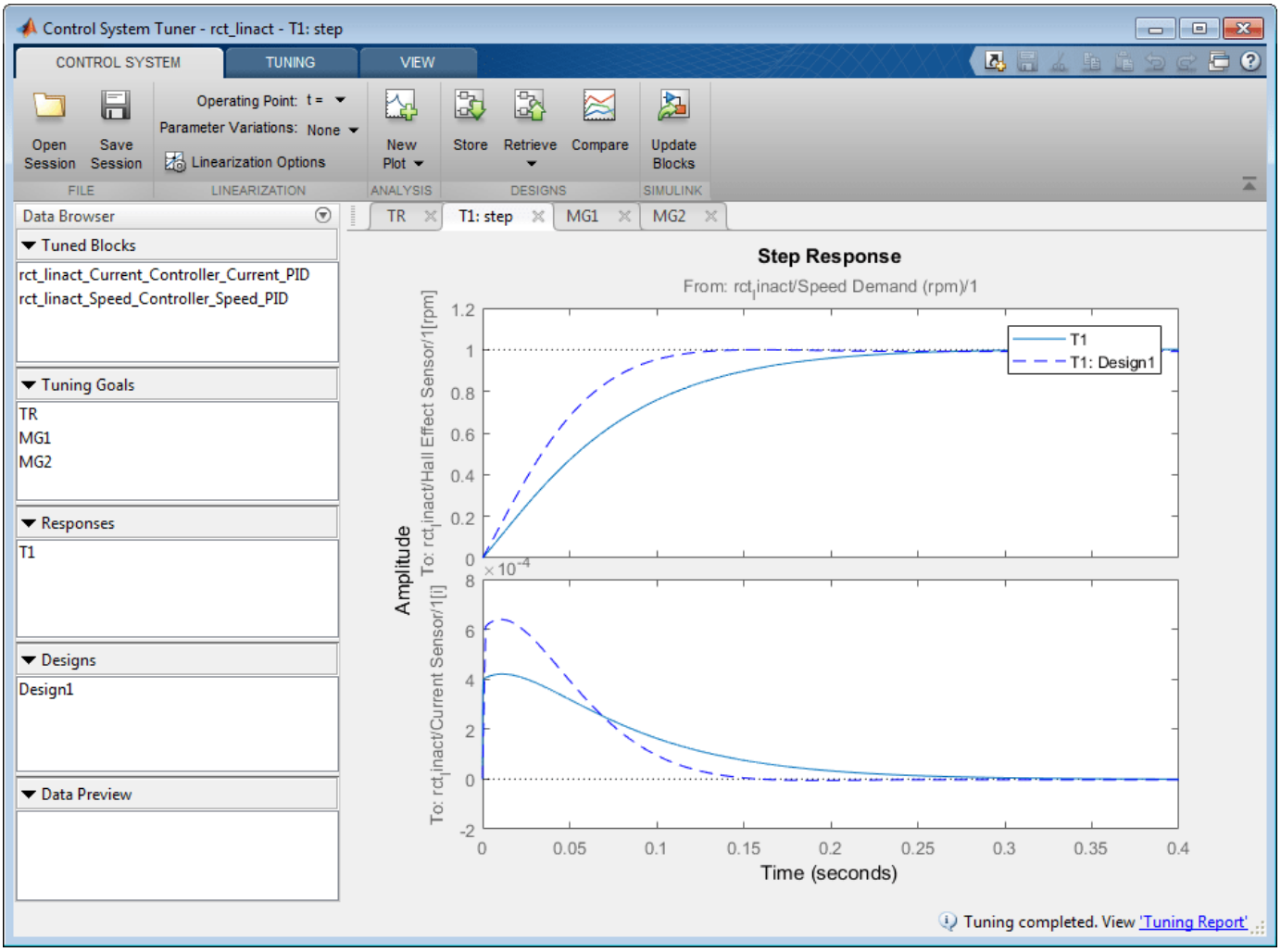


Figure 28: Step Responses of Two Designs.

Finally, push the new tuned gain values to the Simulink model by `Update Blocks` and simulate the response to a 2000 rpm speed demand and 500 N load disturbance. The simulation results appear in Figure 29 and the current controller output is shown in Figure 30.

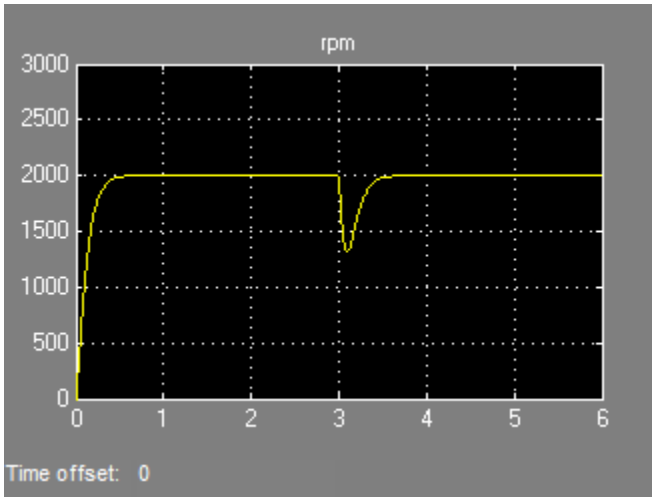


Figure 29: Nonlinear Response of Tuning with Gain Constraints.

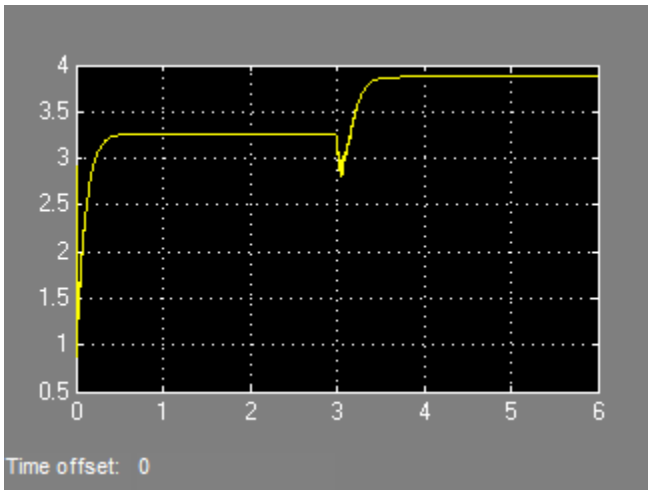


Figure 30: Current Controller Output.

The nonlinear responses are now satisfactory and the current loop is no longer saturating. The additional gain constraints have forced `systune` to re-distribute the control effort between the inner and outer loops so as to avoid saturation.

See Also

Control System Tuner

Related Examples

- “Control of a Linear Electric Actuator”

Multi-Loop PI Control of a Robotic Arm

This example shows how to use `looptune` to tune a multi-loop controller for a 6-DOF robotic arm manipulator.

Robotic Arm Model and Controller

This example uses the six degree-of-freedom robotic arm shown below. This arm consists of six joints labeled from base to tip: "Turntable", "Bicep", "Forearm", "Wrist", "Hand", and "Gripper". Each joint is actuated by a DC motor except for the Bicep joint which uses two DC motors in tandem.

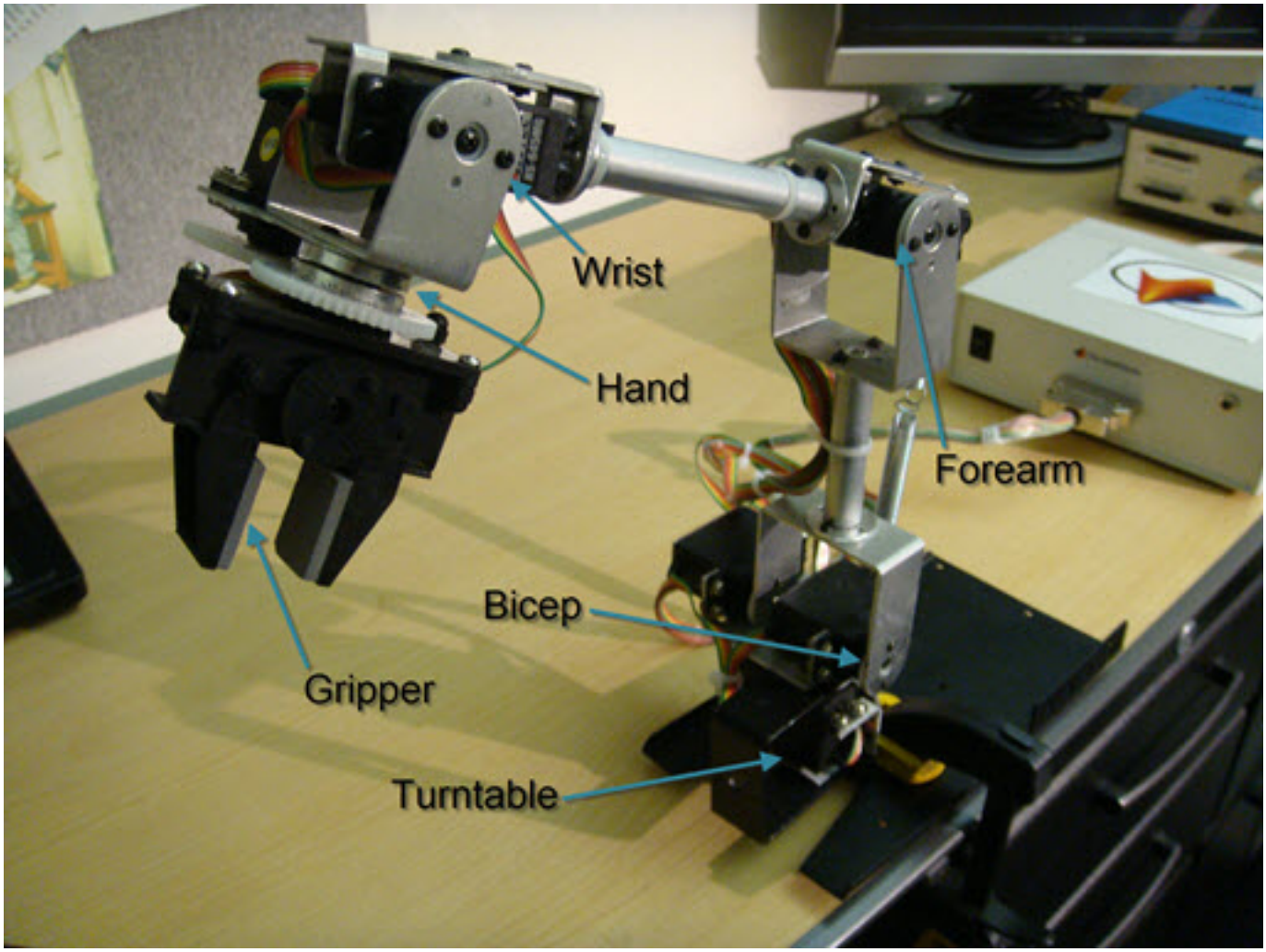


Figure 1: Robotic arm manipulator.

The file "cst_robotarm.slx" contains a Simulink model of the electrical and mechanical components of this system.

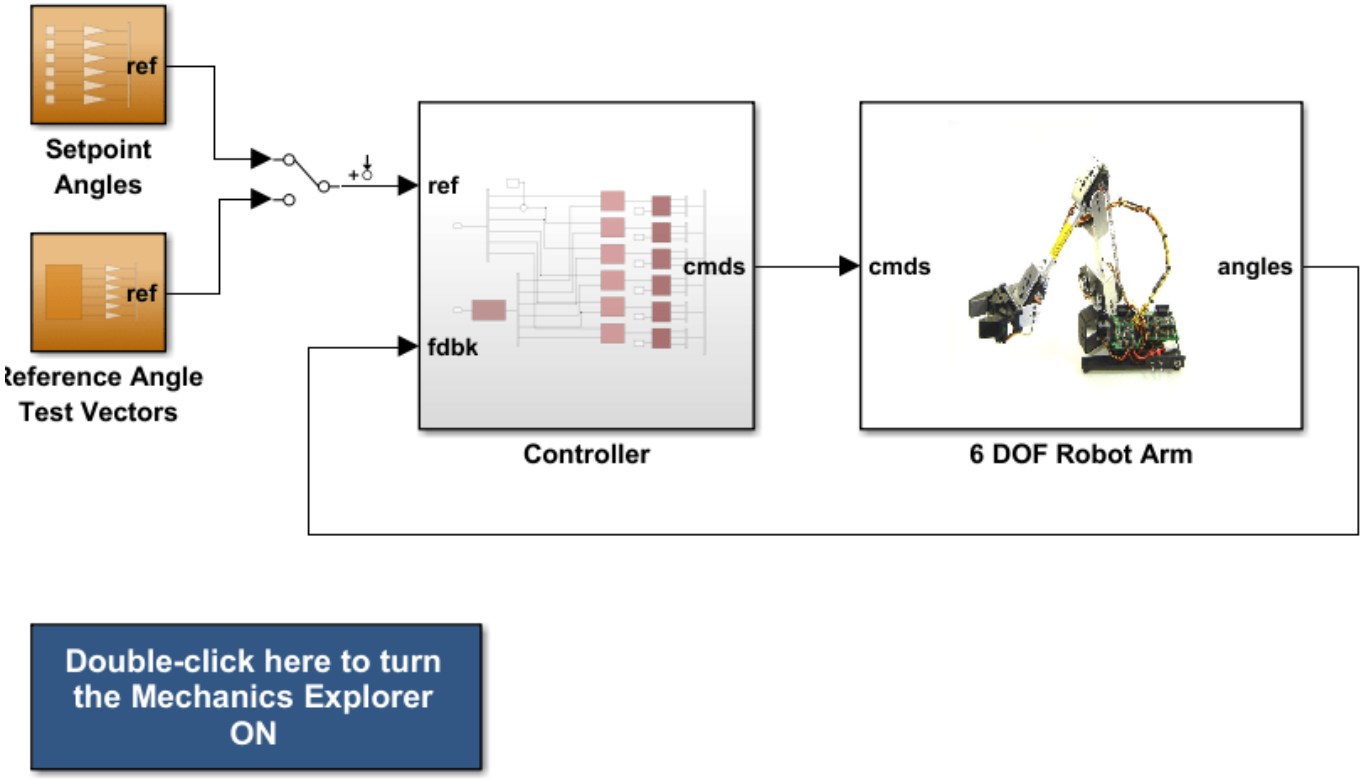


Figure 2: Simulink model of robotic arm.

The "Controller" subsystem consists of six digital PI controllers (one per joint). Each PI controller is implemented using the "2-DOF PID Controller" block from the Simulink library (see *PID Tuning for Setpoint Tracking vs. Disturbance Rejection* example for motivation). The control sample time is Ts=0.1 (10 Hz).

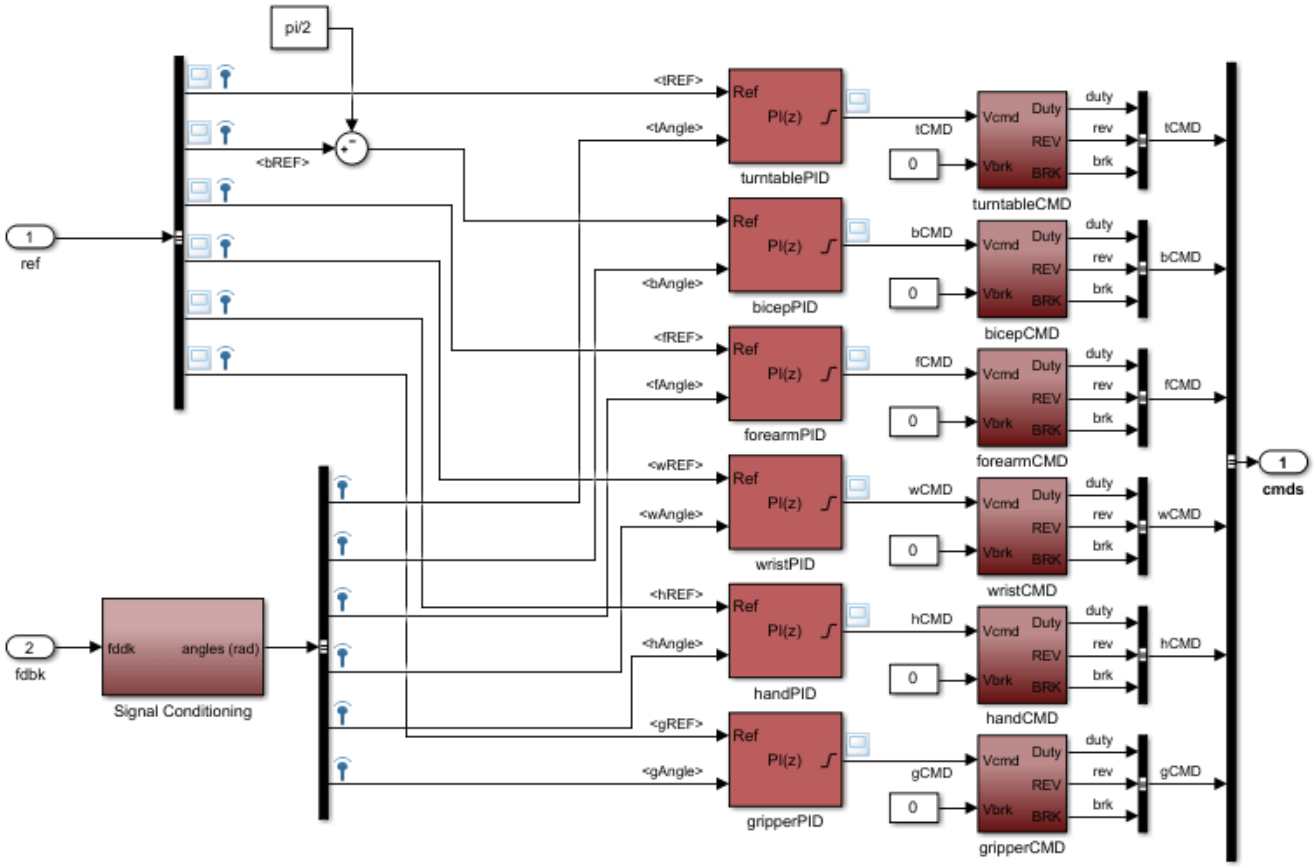
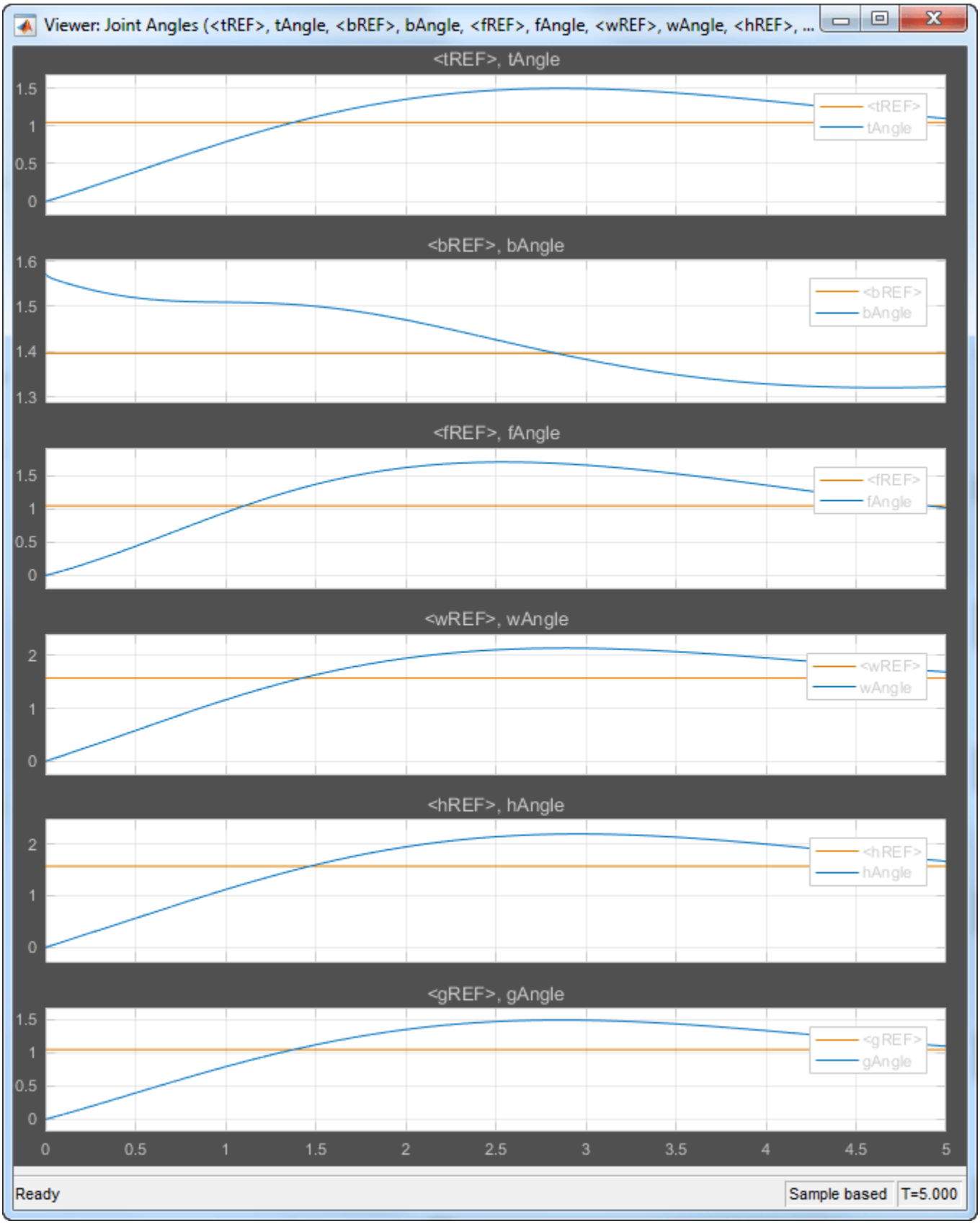


Figure 3: Controller structure.

Typically, such multi-loop controllers are tuned sequentially by tuning one PID loop at a time and cycling through the loops until the overall behavior is satisfactory. This process can be time consuming and is not guaranteed to converge to the best overall tuning. Alternatively, you can use `systune` or `looptune` to jointly tune all six PI loops subject to system-level requirements such as response time and minimum cross-coupling.

In this example, the arm must move to a particular configuration in about 1 second with smooth angular motion at each joint. The arm starts in a fully extended vertical position with all joint angles at zero except for the Bicep angle at ninety degrees. The end configuration is specified by the angular positions: Turntable = 60 deg, Bicep = 80 deg, Forearm = 60 deg, Wrist = 90 deg, Hand = 90 deg, and Gripper = 60 deg.

Press the "Play" button in the Simulink model to simulate the angular trajectories for the PI gain values specified in the model. You can first double-click on the blue button to also show a 3D animation of the robotic arm. The angular responses and the 3D animation appear below. Clearly the response is too sluggish and imprecise.


Viewer: Joint Angles (<tREF>, tAngle, <bREF>, bAngle, <fREF>, fAngle, <wREF>, wAngle, <hREF>, ...
<tREF>, tAngle
<tREF>
tAngle
<bREF>, bAngle
<bREF>
bAngle
<fREF>, fAngle
<fREF>
fAngle
<wREF>, wAngle
<wREF>
wAngle
<hREF>, hAngle
<hREF>
hAngle
<gREF>, gAngle
<gREF>
gAngle
0 0.5 1 1.5 2 2.5 3 3.5 4 4.5 5
Ready
Sample based
T=5.000

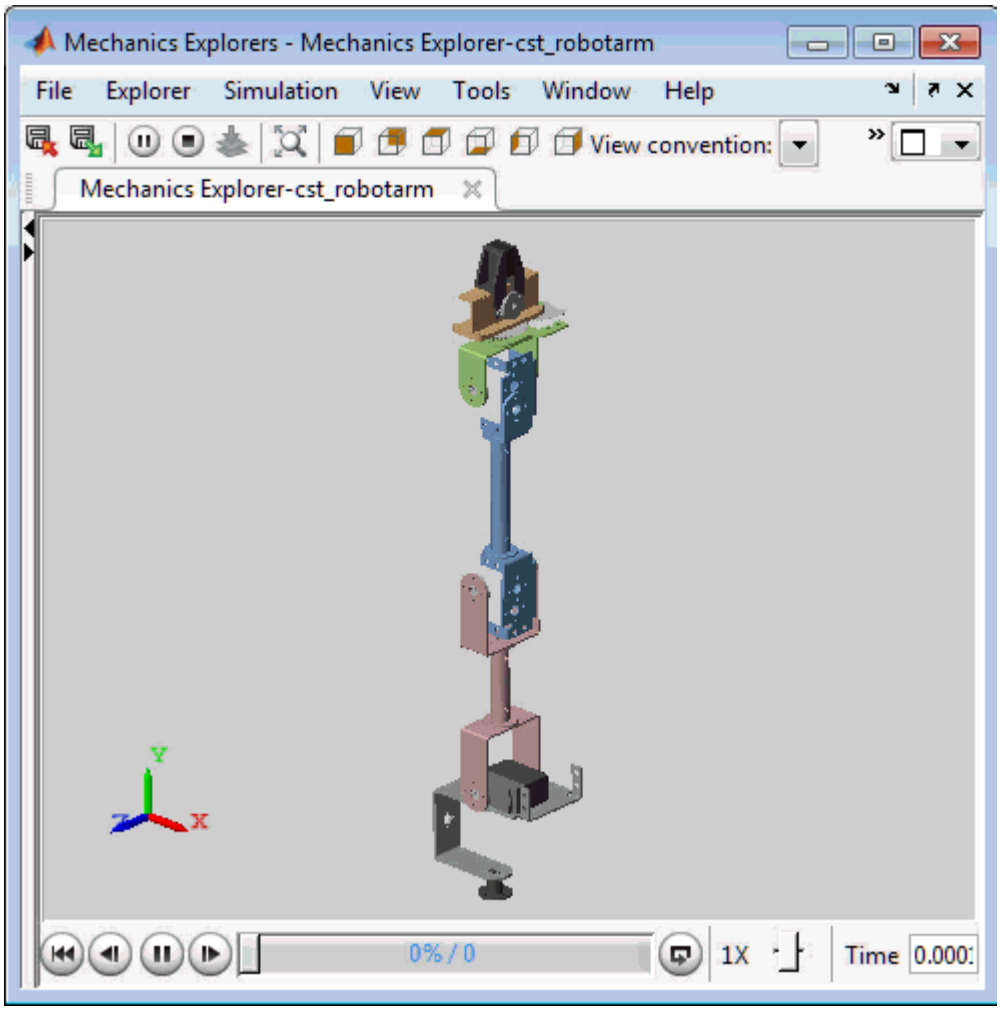


Figure 4: Untuned response.

Linearizing the Plant

The robot arm dynamics are nonlinear. To understand whether the arm can be controlled with one set of PI gains, linearize the plant at various points (snapshot times) along the trajectory of interest. Here "plant" refers to the dynamics between the control signals (outputs of PID blocks) and the measurement signals (output of "6 DOF Robot Arm" block).

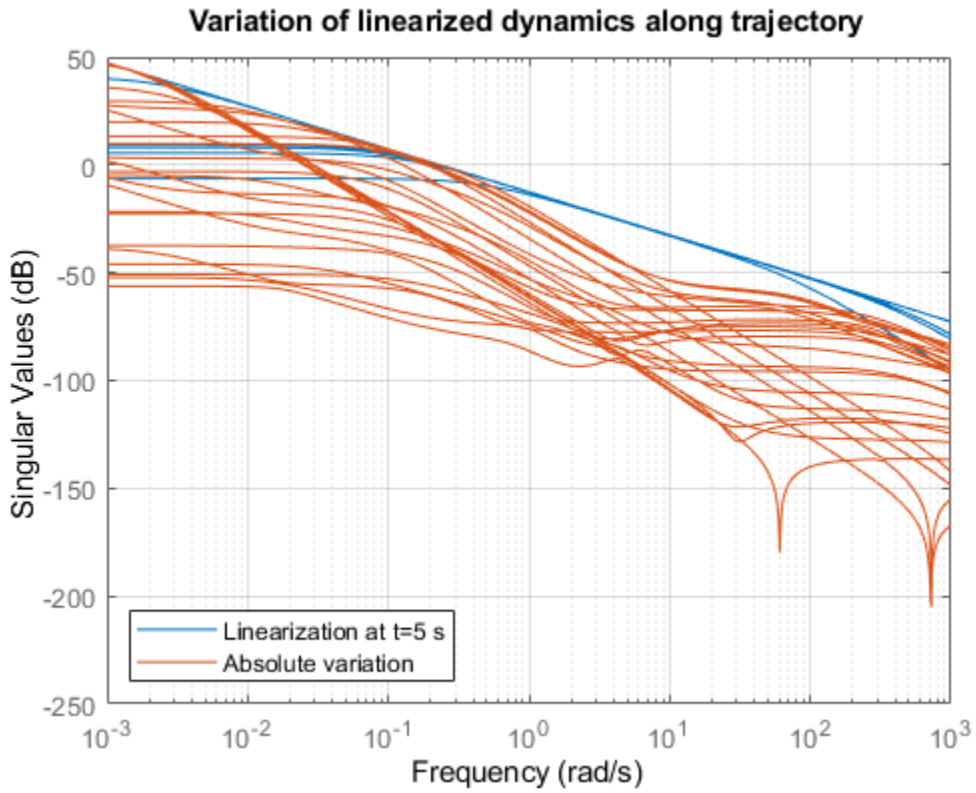
```
SnapshotTimes = 0:1:5;
% Plant is from PID outputs to Robot Arm outputs
LinIOs = [...
   linio('cst_robotarm/Controller/turntablePID',1,'openinput'),...
   linio('cst_robotarm/Controller/bicepPID',1,'openinput'),...
   linio('cst_robotarm/Controller/forearmPID',1,'openinput'),...
   linio('cst_robotarm/Controller/wristPID',1,'openinput'),...
   linio('cst_robotarm/Controller/handPID',1,'openinput'),...
   linio('cst_robotarm/Controller/gripperPID',1,'openinput'),...
   linio('cst_robotarm/6 DOF Robot Arm',1,'output')];
LinOpt = linearizeOptions('SampleTime',0);  % seek continuous-time model
G = linearize('cst_robotarm',LinIOs,SnapshotTimes,LinOpt);
```

```
size(G)
```

```
6x1 array of state-space models.
Each model has 6 outputs, 6 inputs, and 19 states.
```

Plot the gap between the linearized models at t=0,1,2,3,4 seconds and the final model at t=5 seconds.

```
G5 = G(:,:,end);  % t=5
G5.SamplingGrid = [];
sigma(G5,G(:,:,2:5)-G5,{1e-3,1e3}), grid
title('Variation of linearized dynamics along trajectory')
legend('Linearization at t=5 s','Absolute variation',...
       'location','SouthWest')
```



While the dynamics vary significantly at low and high frequency, the variation drops to less than 10% near 10 rad/s, which is roughly the desired control bandwidth. Small plant variations near the target gain crossover frequency suggest that we can control the arm with a single set of PI gains and need not resort to gain scheduling.

Tuning the PI Controllers with LOOPTUNE

With `looptune`, you can directly tune all six PI loops to achieve the desired response time with minimal loop interaction and adequate MIMO stability margins. The controller is tuned in continuous time and automatically discretized when writing the PI gains back to Simulink. Use the `slTuner` interface to specify which blocks must be tuned and to locate the plant/controller boundary.

```
% Linearize the plant at t=3s
tLinearize = 3;

% Create slTuner interface
TunedBlocks = {'turntablePID','bicepPID','forearmPID',...
               'wristPID','handPID','gripperPID'};
ST0 = slTuner('cst_robotarm',TunedBlocks,tLinearize);

% Mark outputs of PID blocks as plant inputs
addPoint(ST0,TunedBlocks)

% Mark joint angles as plant outputs
addPoint(ST0,'6 DOF Robot Arm')

% Mark reference signals
RefSignals = {...
   'ref Select/tREF',...
   'ref Select/bREF',...
   'ref Select/fREF',...
   'ref Select/wREF',...
   'ref Select/hREF',...
   'ref Select/gREF'};
addPoint(ST0,RefSignals)
```

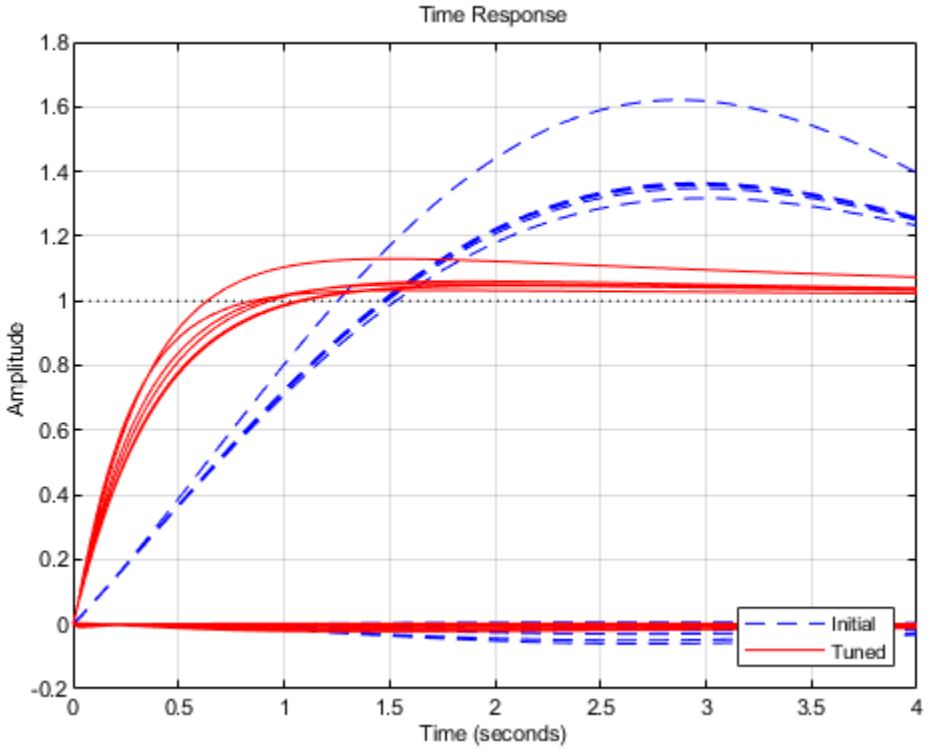
In its simplest use, `looptune` only needs to know the target control bandwidth, which should be at least twice the reciprocal of the desired response time. Here the desired response time is 1 second so try a target bandwidth of 3 rad/s (bearing in mind that the plant dynamics vary least near 10 rad/s).

```
wc = 3;  % target gain crossover frequency
Controls = TunedBlocks;      % actuator commands
Measurements = '6 DOF Robot Arm';  % joint angle measurements
ST1 = looptune(ST0,Controls,Measurements,wc);
```

```
Final: Peak gain = 0.957, Iterations = 10
Achieved target gain value TargetGain=1.
```

A final value near or below 1 indicates that `looptune` achieved the requested bandwidth. Compare the responses to a step command in angular position for the initial and tuned controllers.

```
T0 = getIOTransfer(ST0,RefSignals,Measurements);
T1 = getIOTransfer(ST1,RefSignals,Measurements);

opt = timeoptions; opt.IOGrouping = 'all'; opt.Grid = 'on';
stepplot(T0,'b--',T1,'r',4,opt)
legend('Initial','Tuned','location','SouthEast')
```



The six curves settling near y=1 represent the step responses of each joint, and the curves settling near y=0 represent the cross-coupling terms. The tuned controller is a clear improvement, but there is some overshoot and the Bicep response takes a long time to settle.

Exploiting the Second Degree of Freedom

The 2-DOF PI controllers have a feedforward and a feedback component.

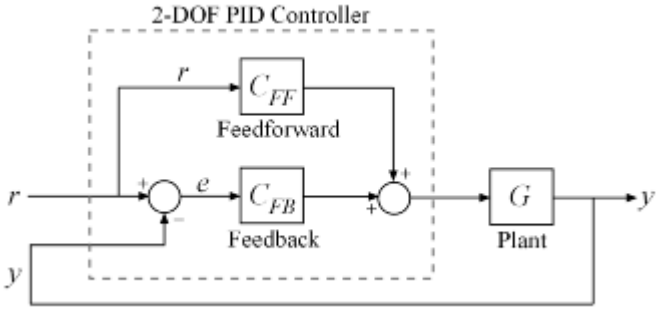


Figure 5: Two degree-of-freedom PID controllers.

By default, `looptune` only tunes the feedback loop and does not "see" the feedforward component. This can be confirmed by verifying that the b parameters of the PI controllers remain set to their initial value $b = 1$ (type `showTunable(ST1)` to see the tuned values). To take advantage of the

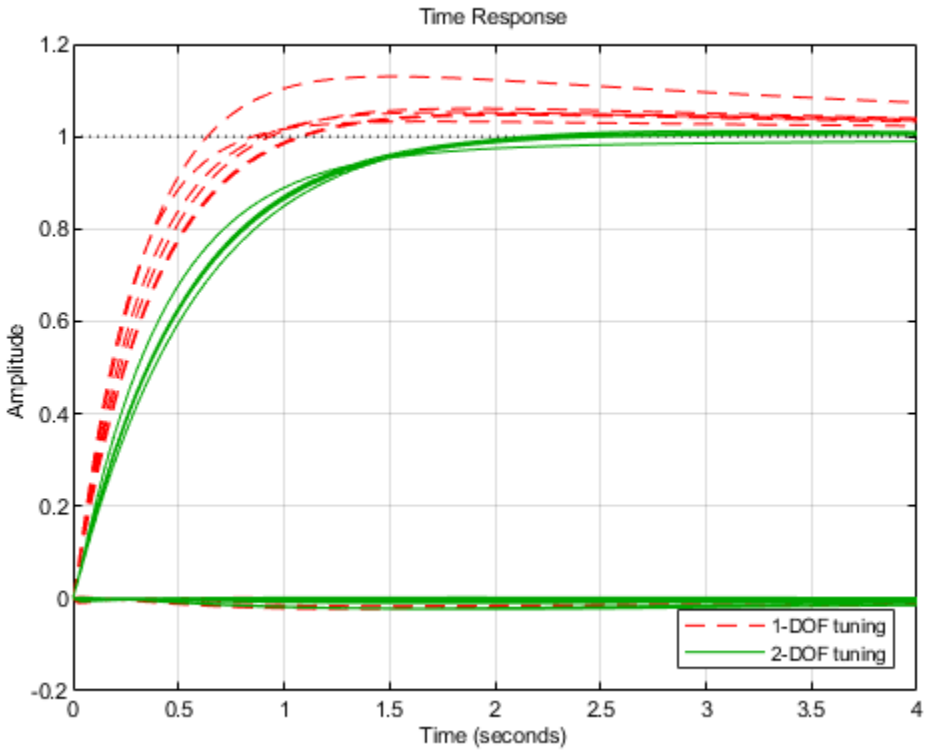
feedforward action and reduce overshoot, replace the bandwidth target by an explicit step tracking requirement from reference angles to joint angles.

```
TR = TuningGoal.StepTracking(RefSignals,Measurements,0.5);
ST2 = looptune(ST0,Controls,Measurements,TR);
```

```
Final: Peak gain = 0.766, Iterations = 13
Achieved target gain value TargetGain=1.
```

The 2-DOF tuning eliminates overshoot and improves the Bicep response.

```
T2 = getIOTransfer(ST2,RefSignals,Measurements);
stepplot(T1,'r--',T2,'g',4,opt)
legend('1-DOF tuning','2-DOF tuning','location','SouthEast')
```



Validating the Tuned Controller

The tuned linear responses look satisfactory so write the tuned values of the PI gains back to the Simulink blocks and simulate the overall maneuver. The simulation results appear in Figure 6.

```
writeBlockValue(ST2)
```

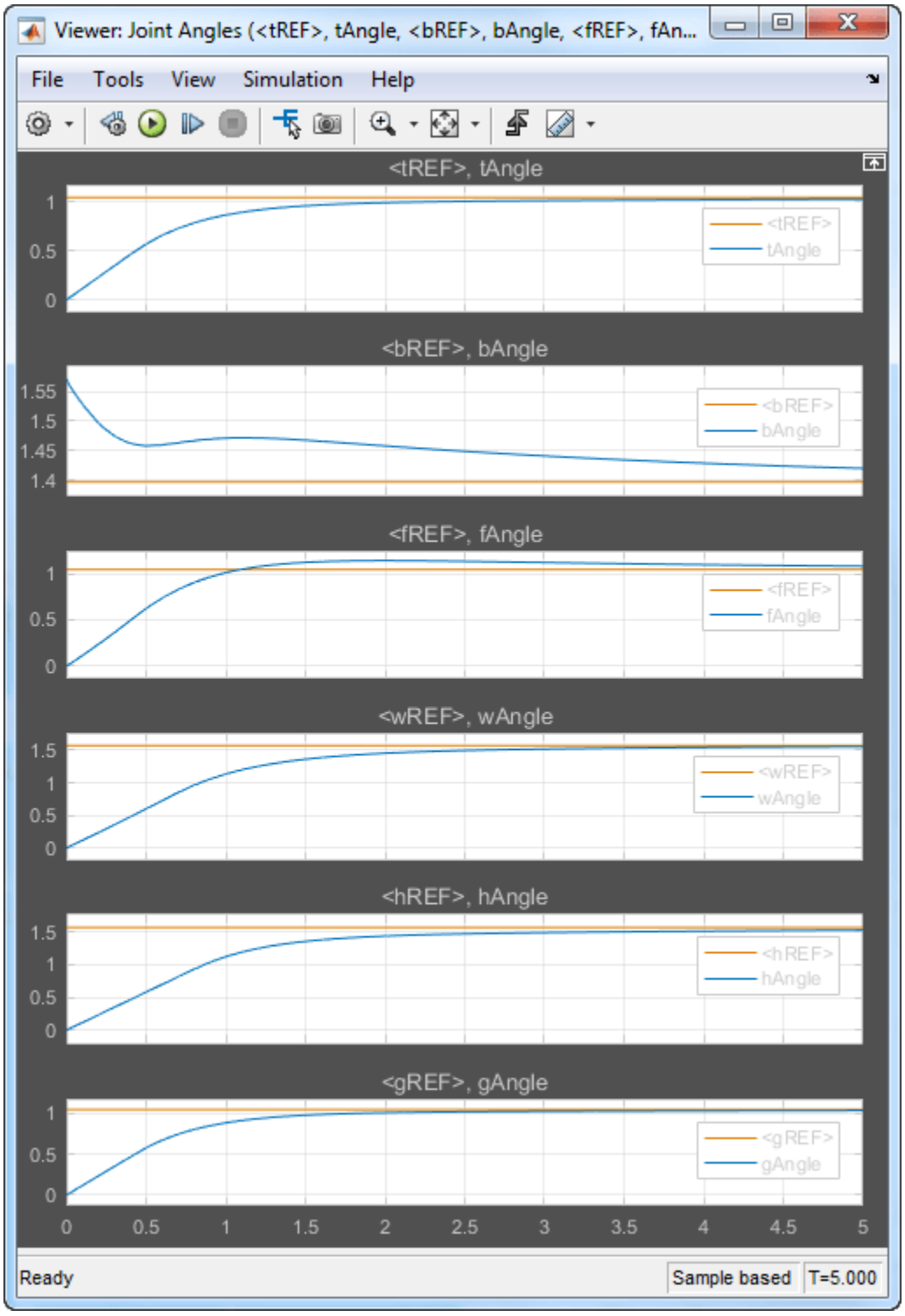


Figure 6: Tuned angular responses.

The nonlinear response of the Bicep joint noticeably undershoots. Further investigation suggests two possible culprits. First, the PI controllers are too aggressive and saturate the motors (the input voltage is limited to ± 5 V).

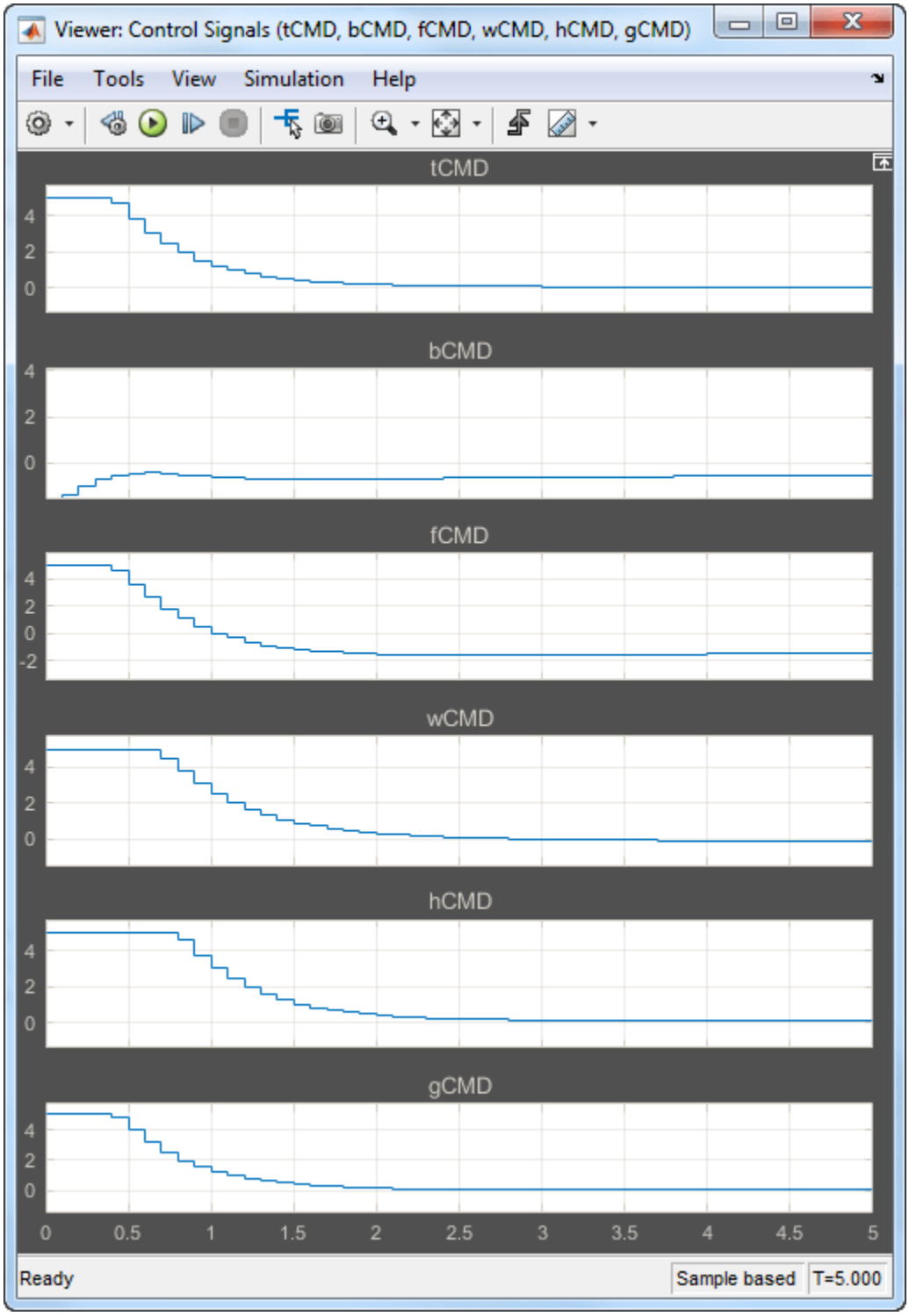
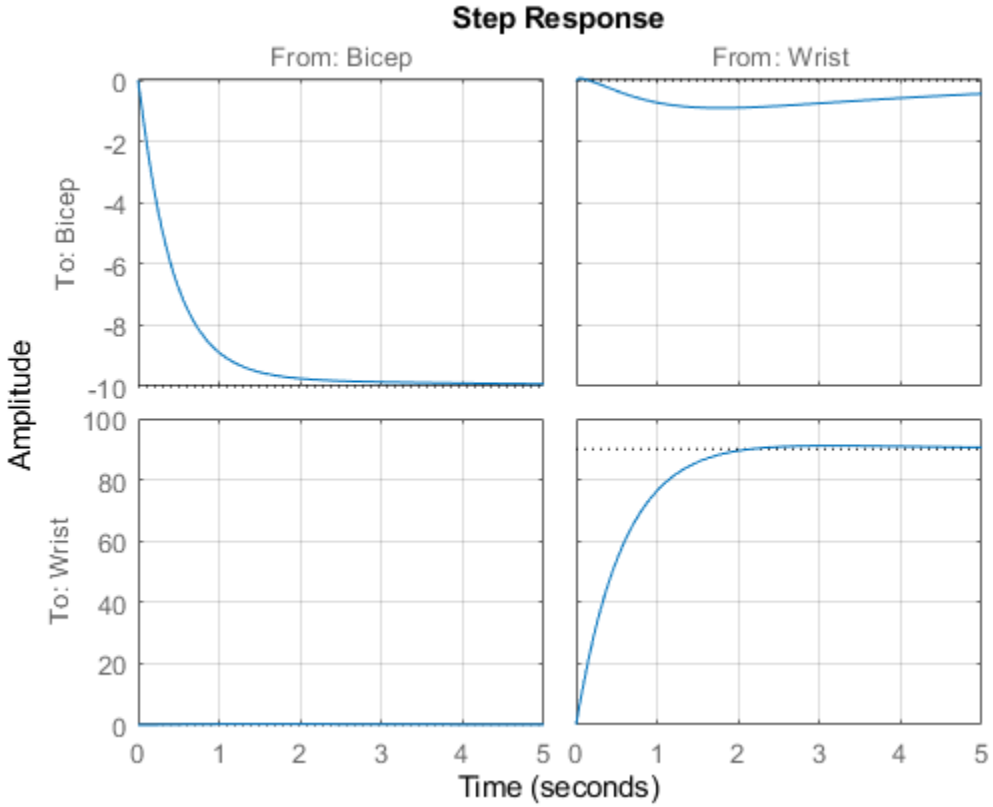


Figure 7: Input voltage to DC motors (control signal).

Second, cross-coupling effects between the Wrist and Bicep, when brought to scale, have a significant and lasting impact on the Bicep response. To see this, plot the step response of these three joints for the **actual** step changes occurring during the maneuver (-10 deg for the Bicep joint and 90 degrees for the Wrist joint).

```
H2 = T2([2 4],[2 4]) * diag([-10 90]);  % scale by step amplitude
H2.u = {'Bicep','Wrist'};
H2.y = {'Bicep','Wrist'};
step(H2,5), grid
```



Refining the Design

To improve the Bicep response for this specific arm maneuver, we must keep the cross-couplings effects small *relative to* the final angular displacements in each joint. To do this, scale the cross-coupling terms in the step tracking requirement by the reference angle amplitudes.

```
JointDisp = [60 10 60 90 90 60];  % commanded angular displacements, in degrees
TR.InputScaling = JointDisp;
```

To reduce saturation of the actuators, limit the gain from reference signals to control signals.

```
UR = TuningGoal.Gain(RefSignals,Controls,6);
```

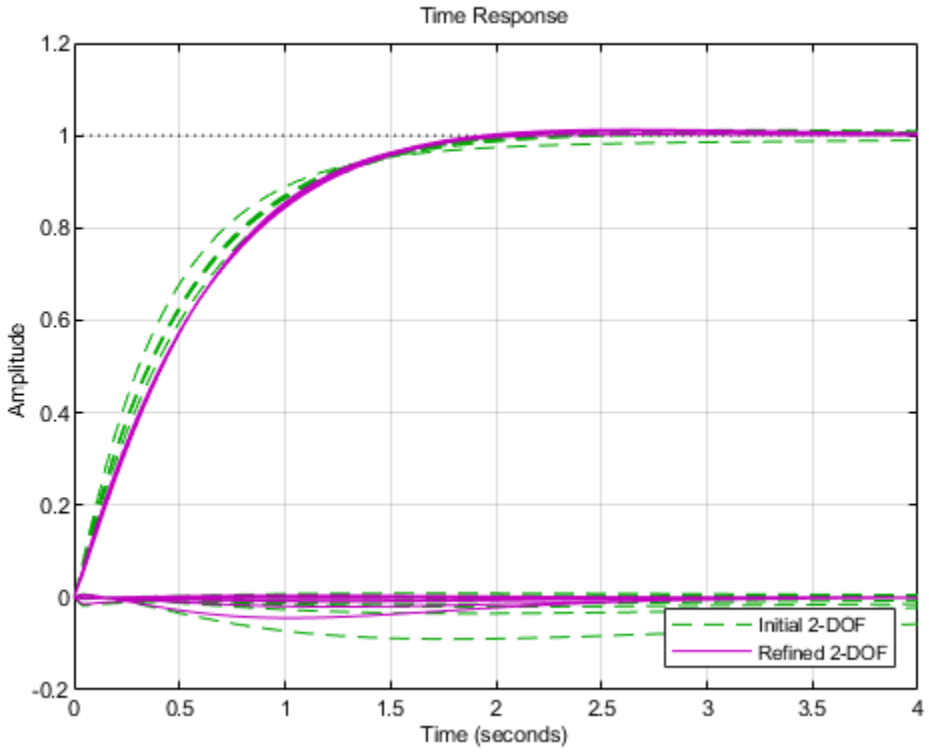
Retune the controller with these refined tuning goals.

```
ST3 = looptune(ST0,Controls,Measurements,TR,UR);
```

```
Final: Peak gain = 1.14, Iterations = 160
```

Compare the scaled responses with the previous design. Notice the significant reduction of the coupling between Wrist and Bicep motion, both in peak value and total energy.

```
T2s = diag(1./JointDisp) * T2 * diag(JointDisp);
T3s = diag(1./JointDisp) * getIOTransfer(ST3,RefSignals,Measurements) * diag(JointDisp);
stepplot(T2s,'g--',T3s,'m',4,opt)
legend('Initial 2-DOF','Refined 2-DOF','location','SouthEast')
```



Push the retuned values to Simulink for further validation.

```
writeBlockValue(ST3)
```

The simulation results appear in Figure 8. The Bicep response is now on par with the other joints in terms of settling time and smooth transient, and there is less actuator saturation than in the previous design.

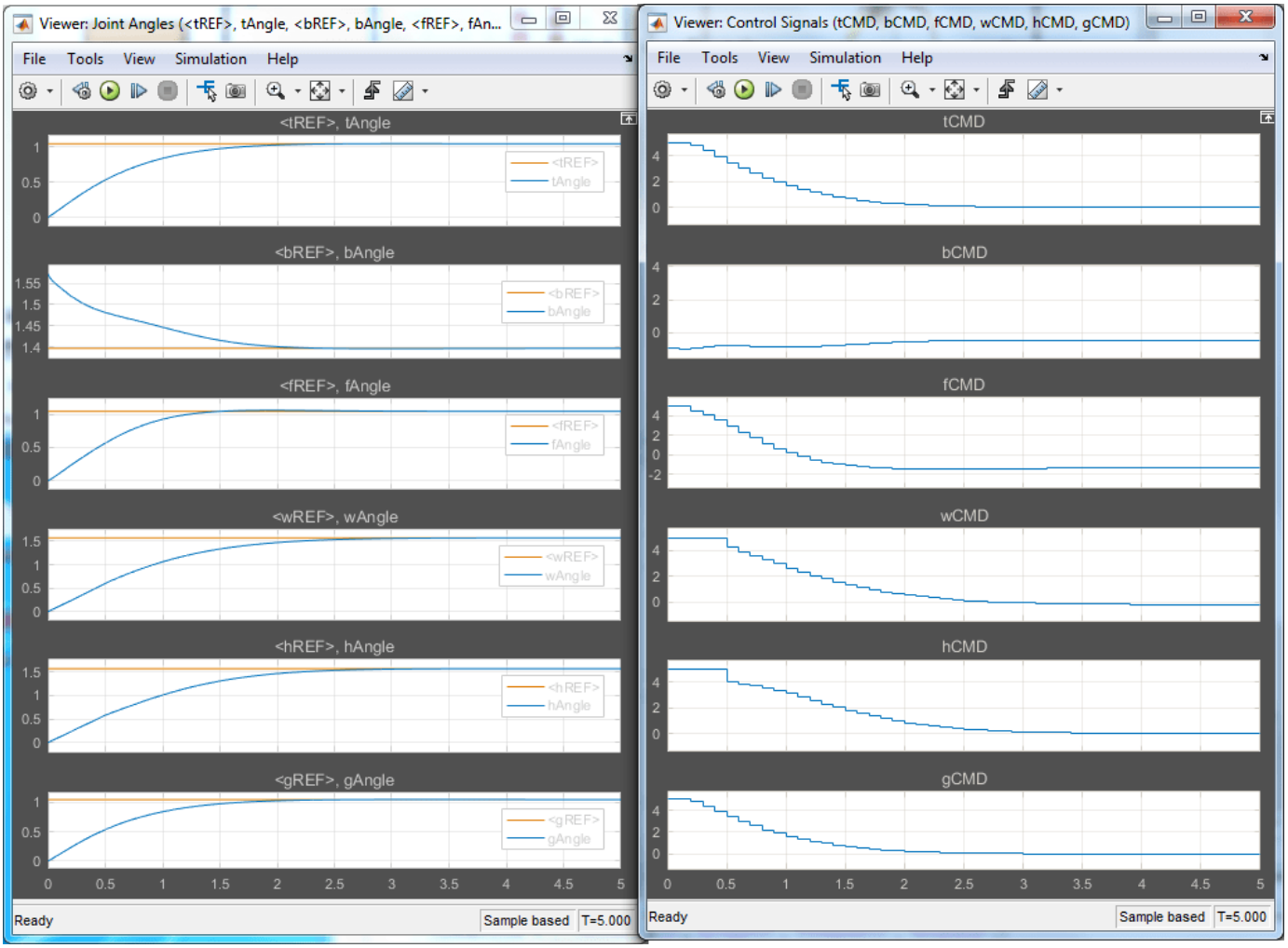


Figure 8: Angular positions and control signals with refined controller.

The 3D animation confirms that the arm now moves quickly and precisely to the desired configuration.

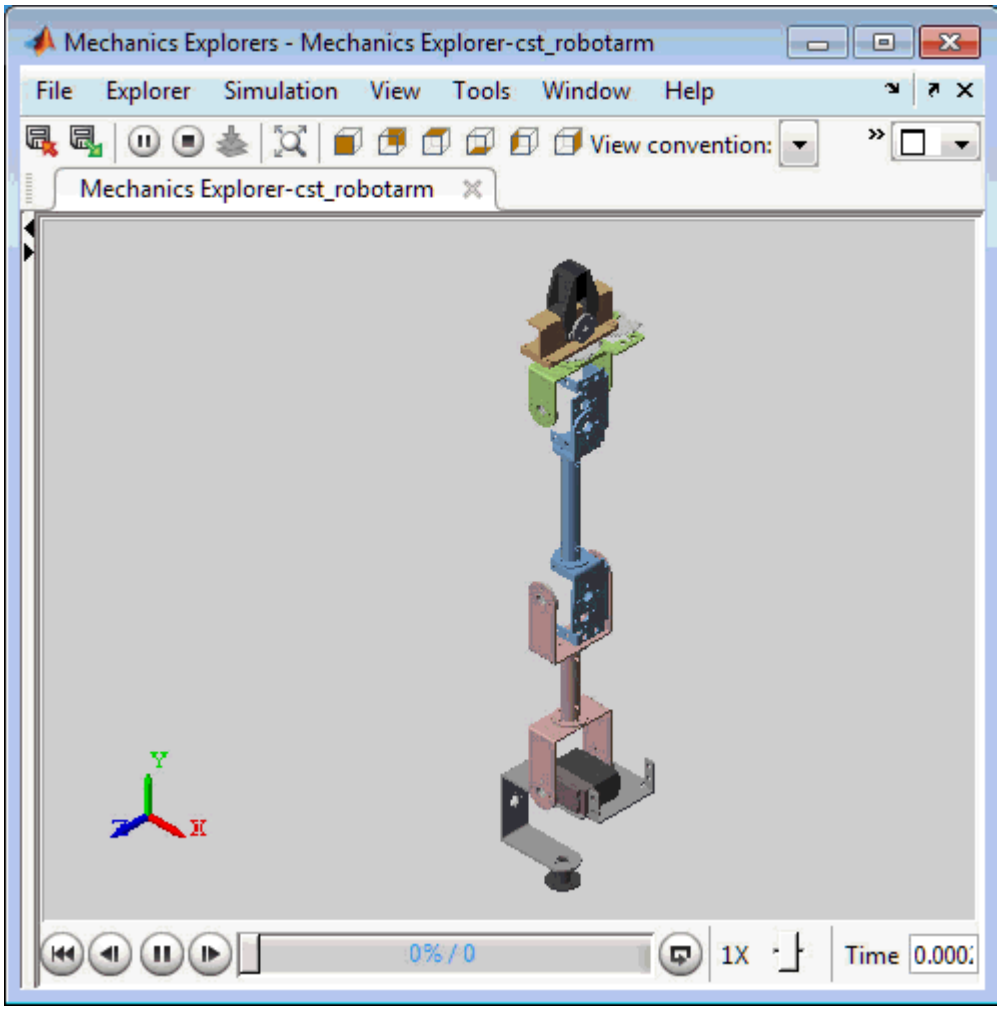


Figure 9: Fine-tuned response.

See Also

`systune` | `TuningGoal.Tracking` | `TuningGoal.MinLoopGain` | `TuningGoal.MaxLoopGain`

Related Examples

- "Active Vibration Control in Three-Story Building"

Control of an Inverted Pendulum on a Cart

This example uses `systune` to control an inverted pendulum on a cart.

Pendulum/Cart Assembly

The cart/pendulum assembly is depicted in Figure 1 and modeled in Simulink® using Simscape™ Multibody™.

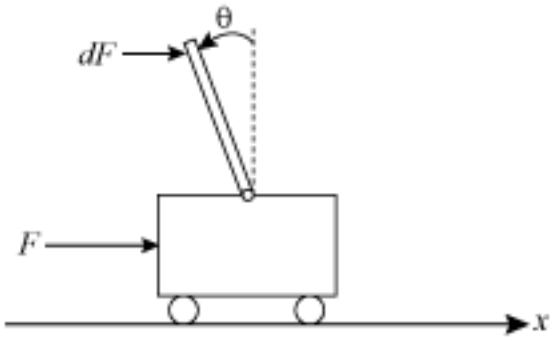


Figure 1: Inverted pendulum on a cart

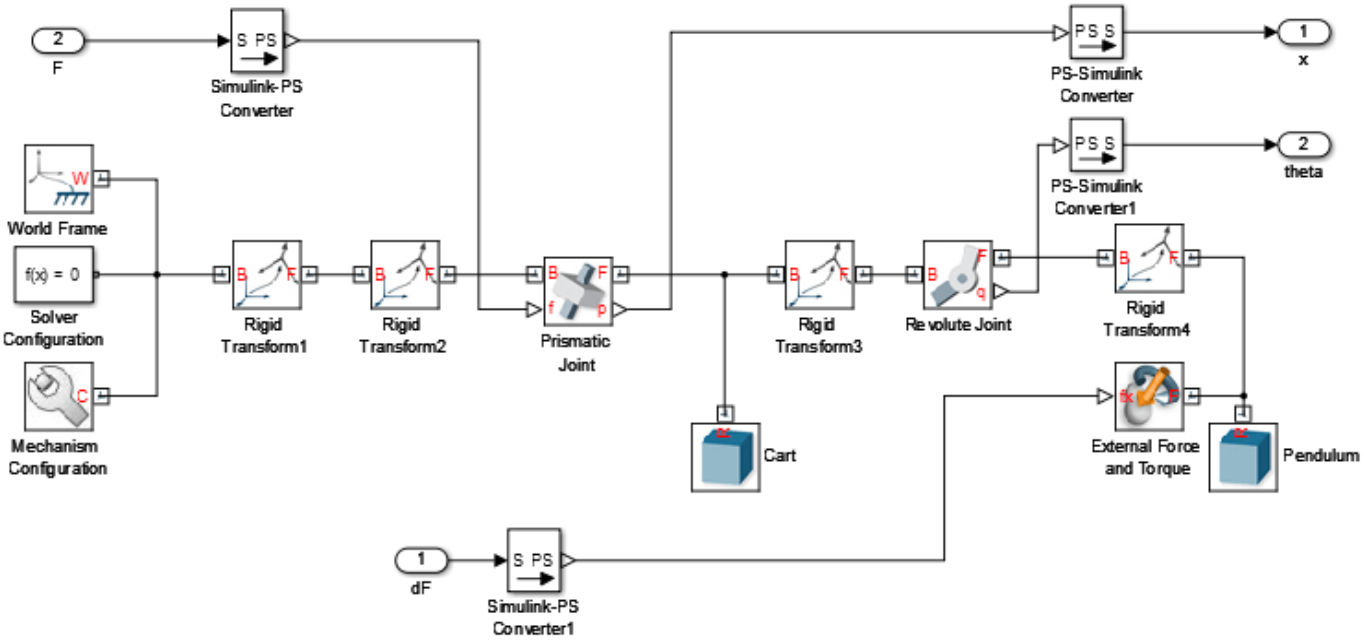


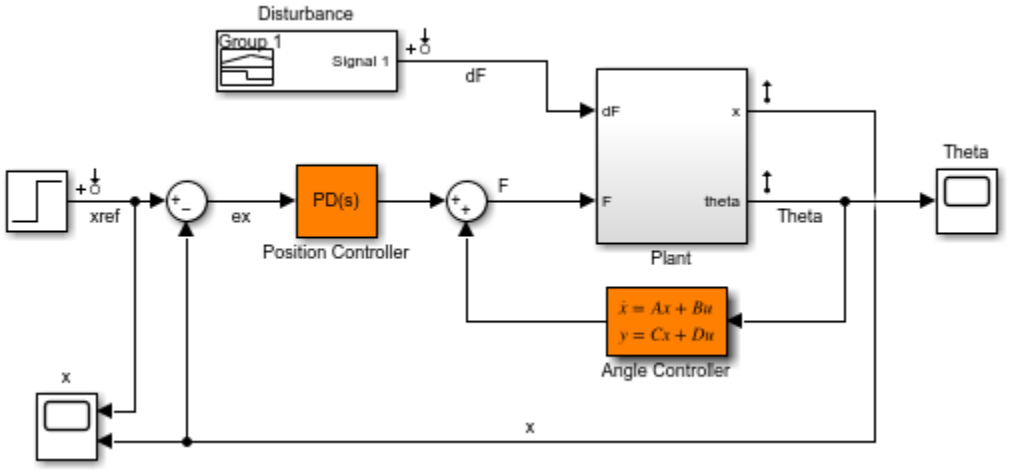
Figure 2: Simscape Multibody model

This system is controlled by exerting a variable force F on the cart. The controller needs to keep the pendulum upright while moving the cart to a new position or when the pendulum is nudged forward (impulse disturbance dF).

Control Structure

The upright position is an unstable equilibrium for the inverted pendulum. The unstable nature of the plant makes the control task more challenging. For this example, you use the following two-loop control structure:

```
open_system('rct_pendulum.slx')
set_param('rct_pendulum','SimMechanicsOpenEditorOnUpdate','off');
```



The inner loop uses a second-order state-space controller to stabilize the pendulum in its upright position (θ control), while the outer loop uses a Proportional-Derivative (PD) controller to control the cart position. You use a PD rather than PID controller because the plant already provides some integral action.

Design Requirements

Use `TuningGoal` requirements to specify the desired closed-loop behavior. Specify a response time of 3 seconds for tracking a setpoint change in cart position x.

```
% Tracking of x command
req1 = TuningGoal.Tracking('xref','x',3);
```

To adequately reject impulse disturbances dF on the tip of the pendulum, use an LQR penalty of the form

$$\int_0^\infty (16\theta^2(t) + x^2(t) + 0.01F^2(t))dt$$

that emphasizes a small angular deviation θ and limits the control effort F.

```
% Rejection of impulse disturbance dF
Qxu = diag([16 1 0.01]);
req2 = TuningGoal.LQG('dF',{'Theta','x','F'},1,Qxu);
```

For robustness, require at least 6 dB of gain margin and 40 degrees of phase margin at the plant input.

```
% Stability margins
req3 = TuningGoal.Margins('F',6,40);
```

Finally, constrain the damping and natural frequency of the closed-loop poles to prevent jerky or underdamped transients.

```
% Pole locations
MinDamping = 0.5;
MaxFrequency = 45;
req4 = TuningGoal.Poles(0,MinDamping,MaxFrequency);
```

Control System Tuning

The closed-loop system is unstable for the initial values of the PD and state-space controllers (1 and $2/s$, respectively). You can use `systune` to jointly tune these two controllers. Use the `slTuner` interface to specify the tunable blocks and register the plant input `F` as an analysis point for measuring stability margins.

```
ST0 = slTuner('rct_pendulum',{'Position Controller','Angle Controller'});
addPoint(ST0,'F');
```

Next, use `systune` to tune the PD and state-space controllers subject to the performance requirements specified above. Optimize the tracking and disturbance rejection performance (soft requirements) subject to the stability margins and pole location constraints (hard requirements).

```
rng(0)
Options = systuneOptions('RandomStart',5);
[ST, fSoft] = systune(ST0,[req1,req2],[req3,req4],Options);
```

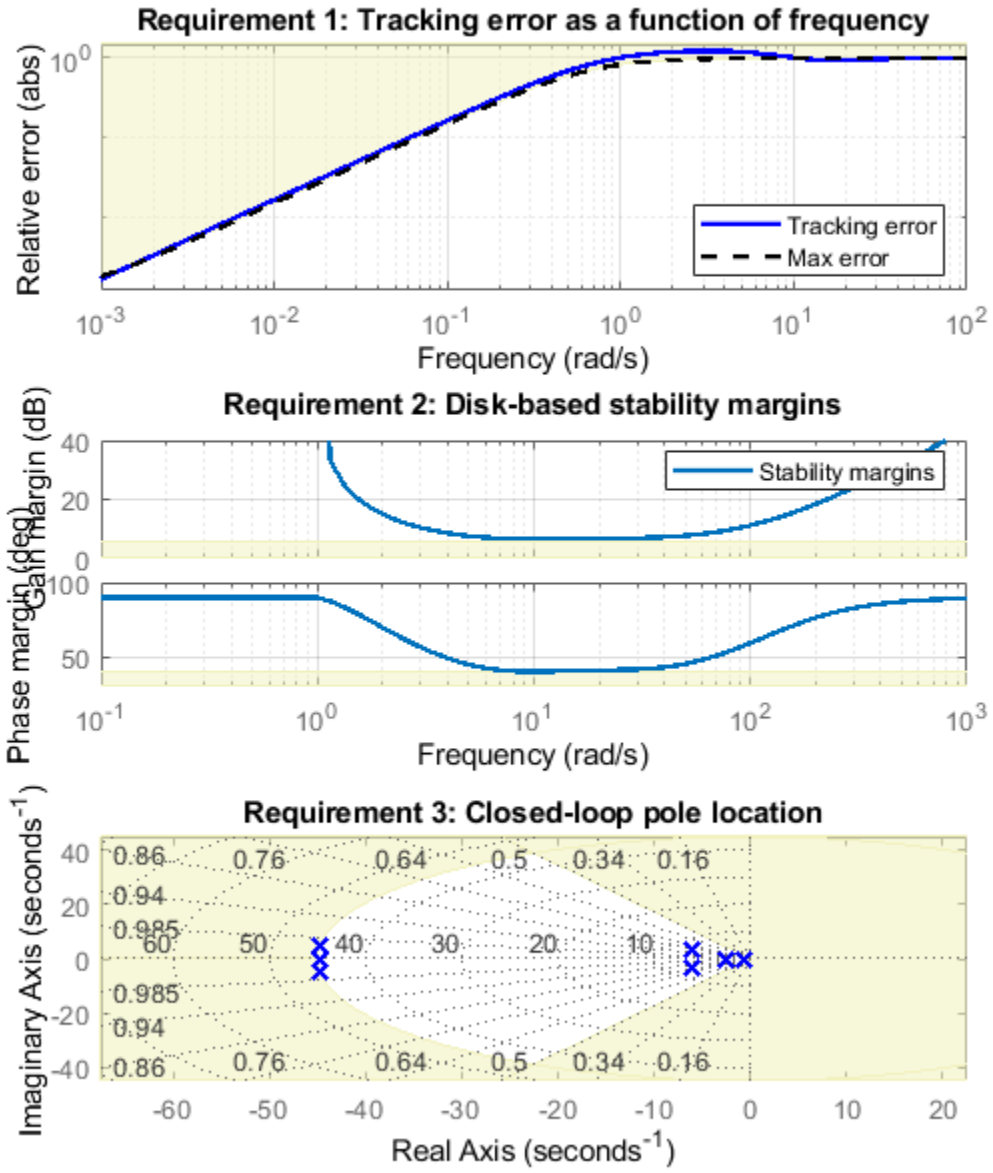
```
Final: Soft = 1.38, Hard = 0.99957, Iterations = 346
Final: Soft = 1.44, Hard = 0.99932, Iterations = 140
Final: Soft = 1.44, Hard = 0.9993, Iterations = 334
Final: Soft = 1.26, Hard = 0.99893, Iterations = 292
Final: Soft = 1.44, Hard = 0.9995, Iterations = 248
Final: Soft = 1.4, Hard = 0.99956, Iterations = 264
```

The best design achieves a value close to 1 for the soft requirements while satisfying the hard requirements (`Hard`<1). This means that the tuned control system nearly achieves the target performance for tracking and disturbance rejection while satisfying the stability margins and pole location constraints.

Validation

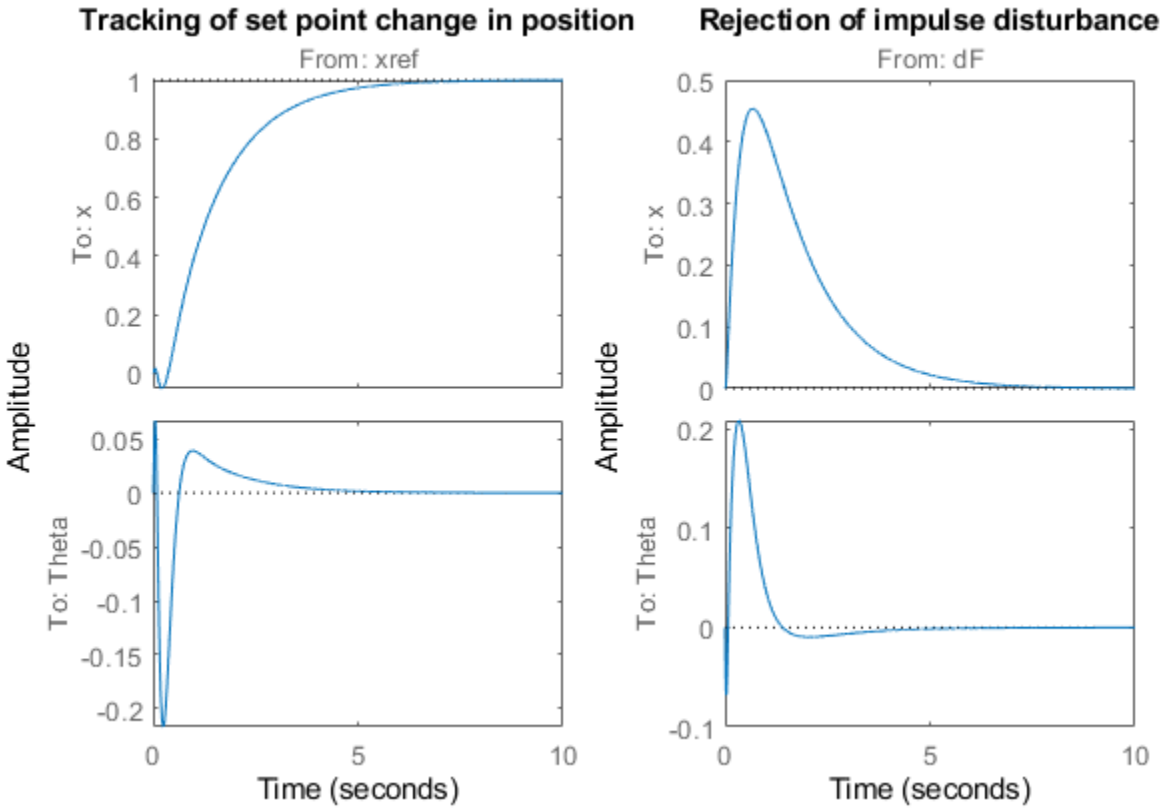
Use `viewGoal` to further analyze how the best design fares against each requirement.

```
figure('Position',[100   100   575   660])
viewGoal([req1,req3,req4],ST)
```



These plots confirm that the first two requirements are nearly satisfied while the last two are strictly enforced. Next, plot the responses to a step change in position and to a force impulse on the cart.

```
T = getIOTransfer(ST,{'xref','dF'},{'x','Theta'});
figure('Position',[100   100   650   420]);
subplot(121), step(T(:,1),10)
title('Tracking of set point change in position')
subplot(122), impulse(T(:,2),10)
title('Rejection of impulse disturbance')
```



The responses are smooth with the desired settling times. Inspect the tuned values of the controllers.

```
C1 = getBlockValue(ST,'Position Controller')
```

```
C1 =

                s
  Kp + Kd * --------
             Tf*s+1

  with Kp = 6.05, Kd = 2.05, Tf = 0.0528

Name: Position_Controller
Continuous-time PDF controller in parallel form.
```

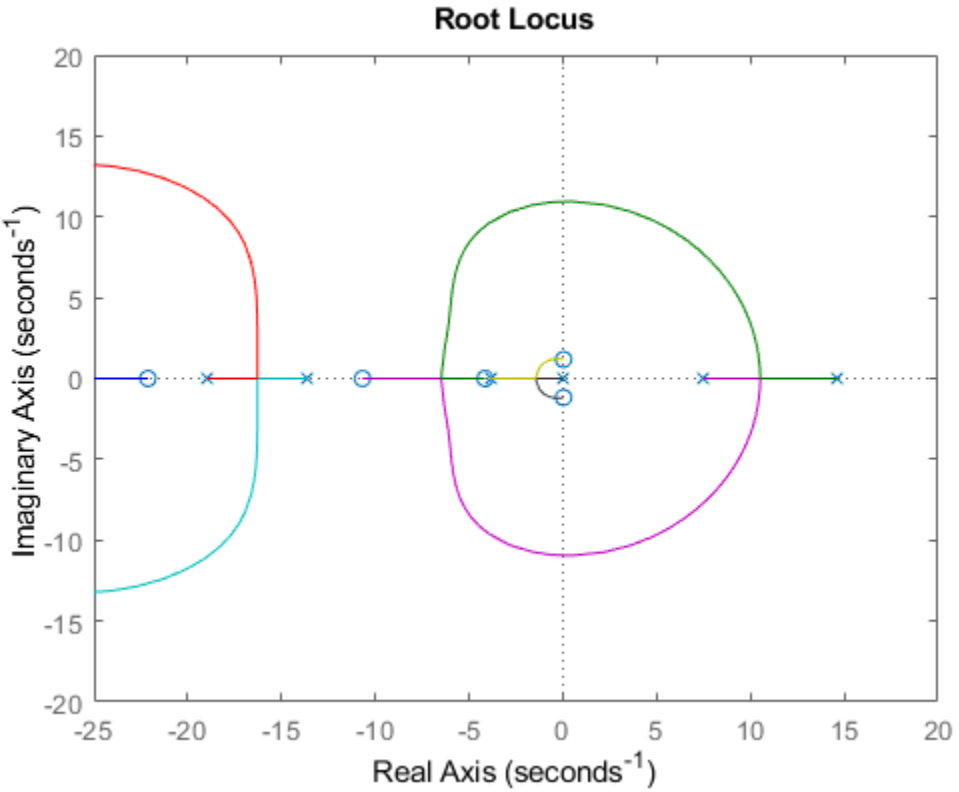
```
C2 = zpk(getBlockValue(ST,'Angle Controller'))
```

```
C2 =

  -1636.5 (s+12.92) (s+4.417)
  ---------------------------
      (s+135.6) (s-14.66)

Name: Angle_Controller
```

```
Continuous-time zero/pole/gain model.
```

Note that the angle controller has an unstable pole that pairs up with the plant unstable pole to stabilize the inverted pendulum. To see this, get the open-loop transfer at the plant input and plot the root locus.

```
L = getLoopTransfer(ST,'F',-1);
figure
rlocus(L)
set(gca,'XLim',[-25 20],'YLim',[-20 20])
```



To complete the validation, upload the tuned values to Simulink and simulate the nonlinear response of the cart/pendulum assembly. A video of the resulting simulation appears below.

```
writeBlockValue(ST)
```

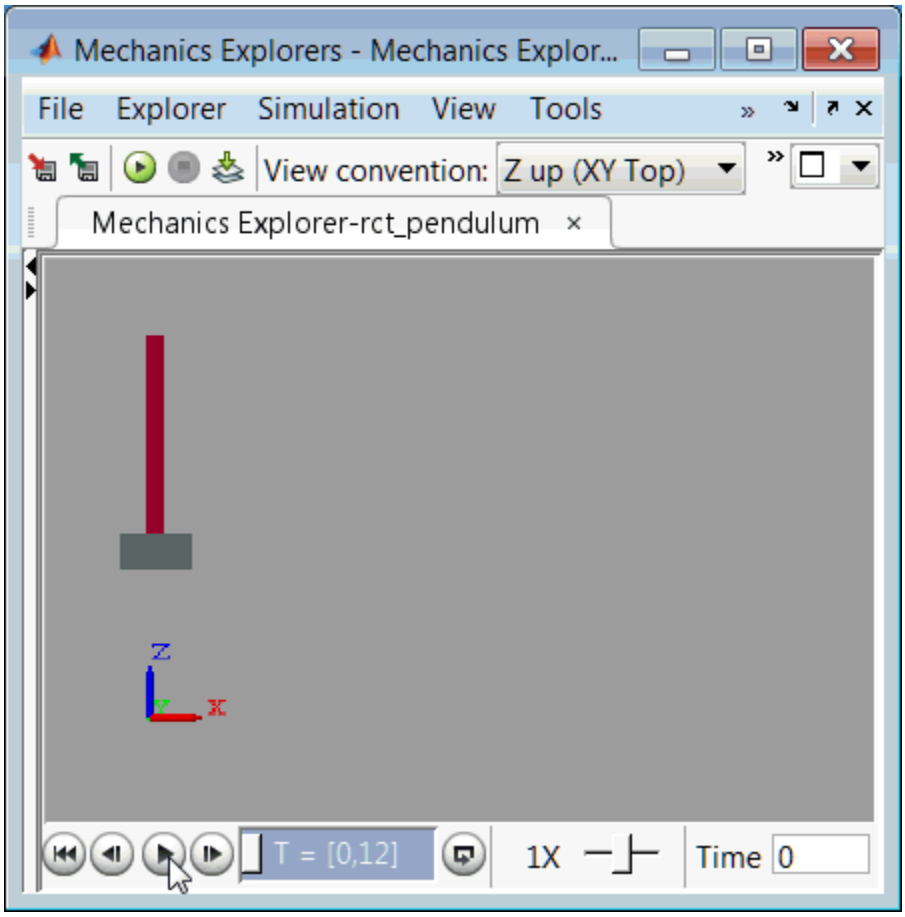


Figure 3: Cart/pendulum simulation with tuned controllers.

Close the model after simulation.

```
set_param('rct_pendulum','SimMechanicsOpenEditorOnUpdate','on');
close_system('rct_pendulum',0);
```

See Also

`systune` | `slTuner`

Related Examples

- “Mark Signals of Interest for Control System Analysis and Design” on page 2-38
- “Create and Configure slTuner Interface to Simulink Model” on page 10-161

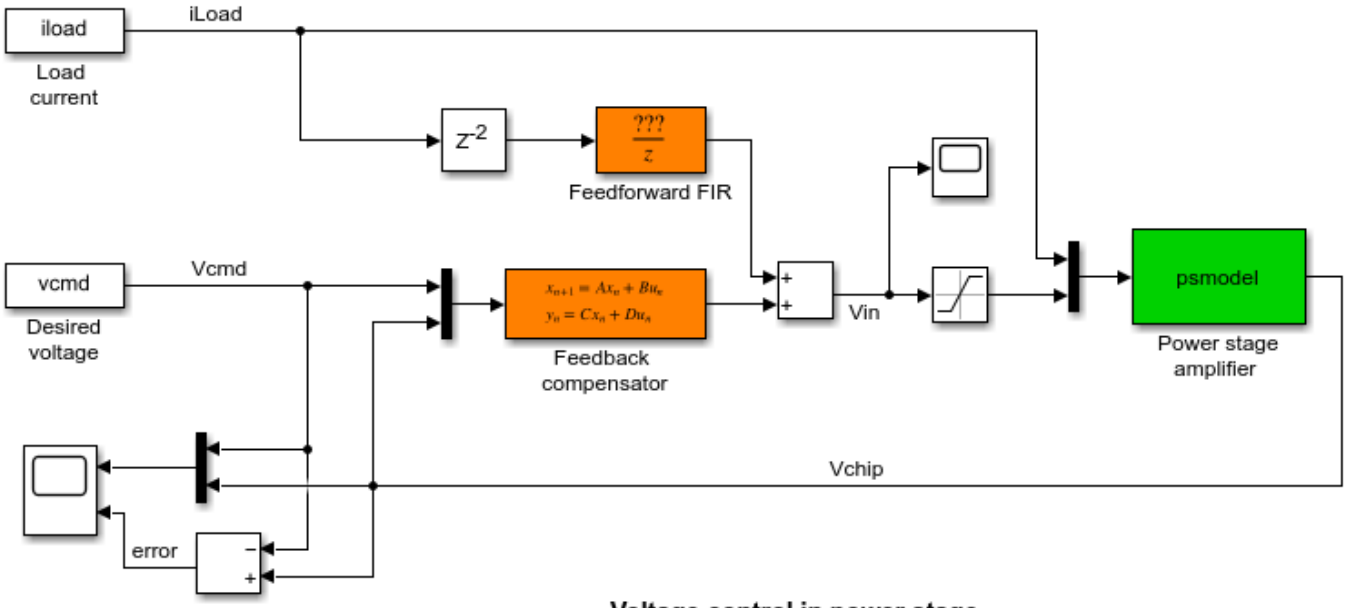
Digital Control of Power Stage Voltage

This example shows how to tune a high-performance digital controller with bandwidth close to the sampling frequency.

Voltage Regulation in Power Stage

We use Simulink to model the voltage controller in the power stage for an electronic device:

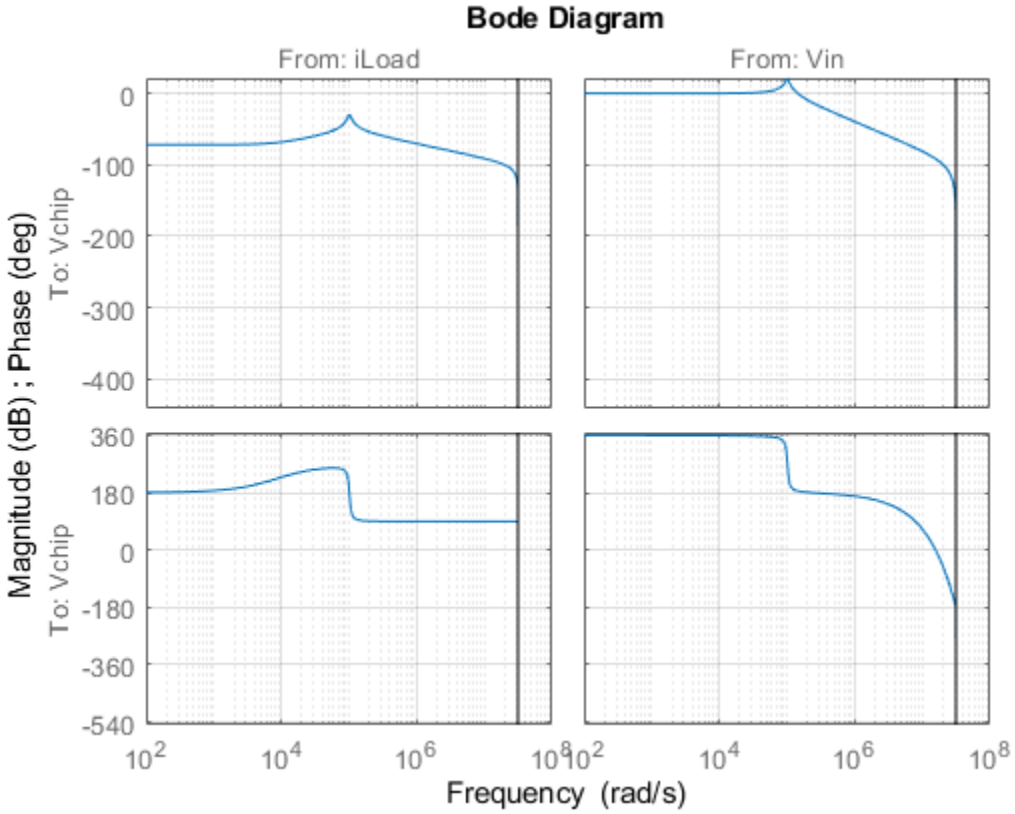
```
open_system('rct_powerstage')
```



Voltage control in power stage

The power stage amplifier is modeled as a second-order linear system with the following frequency response:

```
bode(psmodel)
grid
```



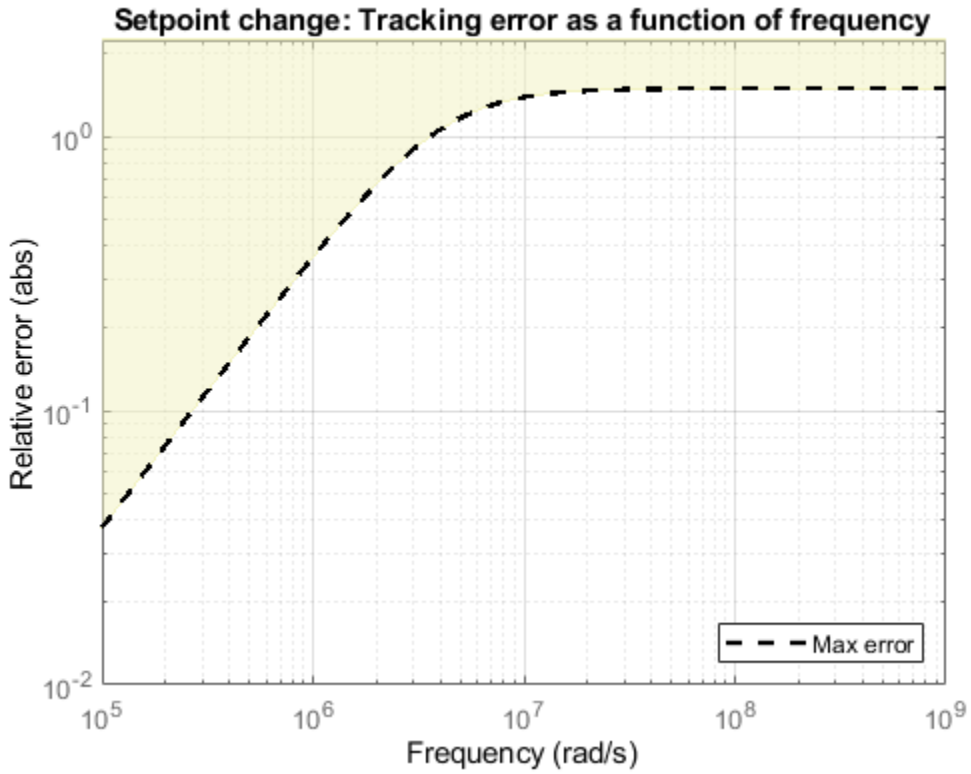
The controller must regulate the voltage `Vchip` delivered to the device to track the setpoint `Vcmd` and be insensitive to variations in load current `iLoad`. The control structure consists of a feedback compensator and a disturbance feedforward compensator. The voltage `Vin` going into the amplifier is limited to $V_{\max} = 12V$. The controller sampling rate is 10 MHz (sample time `Tm` is 1e-7 seconds).

Performance Requirements

This application is challenging because the controller bandwidth must approach the Nyquist frequency `pi/Tm` = 31.4 MHz. To avoid aliasing troubles when discretizing continuous-time controllers, it is preferable to tune the controller directly in discrete time.

The power stage should respond to a setpoint change in desired voltage `Vcmd` in about 5 sampling periods with a peak error (across frequency) of 50%. Use a tracking requirement to capture this objective.

```
Req1 = TuningGoal.Tracking('Vcmd','Vchip',5*Tm,0,1.5);
Req1.Name = 'Setpoint change';

viewGoal(Req1)
```



The power stage should also quickly reject load disturbances `iLoad`. Express this requirement in terms of gain from `iLoad` to `Vchip`. This gain should be low at low frequency for good disturbance rejection.

```
s = tf('s');
nf = pi/Tm;   % Nyquist frequency

Req2 = TuningGoal.Gain('iLoad','Vchip',1.5e-3 * s/nf);
Req2.Focus = [nf/1e4, nf];
Req2.Name = 'Load disturbance';
```

High-performance demands may lead to high control effort and saturation. For the ramp profile `vcmd` specified in the Simulink model (from 0 to 1 in about 250 sampling periods), we want to avoid hitting the saturation constraint $V_{\text{in}} \le V_{\text{max}}$. Use a rate-limiting filter to model the ramp command, and require that the gain from the rate-limiter input to V_{in} be less than V_{max}.

```
RateLimiter = 1/(250*Tm*s); % models ramp command in Simulink

% |RateLimiter * (Vcmd->Vin)| < Vmax
Req3 = TuningGoal.Gain('Vcmd','Vin',Vmax/RateLimiter);
Req3.Focus = [nf/1000, nf];
Req3.Name = 'Saturation';
```

To ensure adequate robustness, require at least 7 dB gain margin and 45 degrees phase margin at the plant input.

```
Req4 = TuningGoal.Margins('Vin',7,45);
Req4.Name = 'Margins';
```

Finally, the feedback compensator has a tendency to cancel the plant resonance by notching it out. Such plant inversion may lead to poor results when the resonant frequency is not exactly known or subject to variations. To prevent this, impose a minimum closed-loop damping of 0.5 to actively damp of the plant's resonant mode.

```
Req5 = TuningGoal.Poles(0,0.5,3*nf);
Req5.Name = 'Damping';
```

Tuning

Next use `systune` to tune the controller parameters subject to the requirements defined above. First use the `slTuner` interface to configure the Simulink model for tuning. In particular, specify that there are two tunable blocks and that the model should be linearized and tuned at the sample time `Tm`.

```
TunedBlocks = {'compensator','FIR'};
ST0 = slTuner('rct_powerstage',TunedBlocks);
ST0.Ts = Tm;

% Register points of interest for open- and closed-loop analysis
addPoint(ST0,{'Vcmd','iLoad','Vchip','Vin'});
```

We want to use an FIR filter as feedforward compensator. To do this, create a parameterization of a first-order FIR filter and assign it to the "Feedforward FIR" block in Simulink.

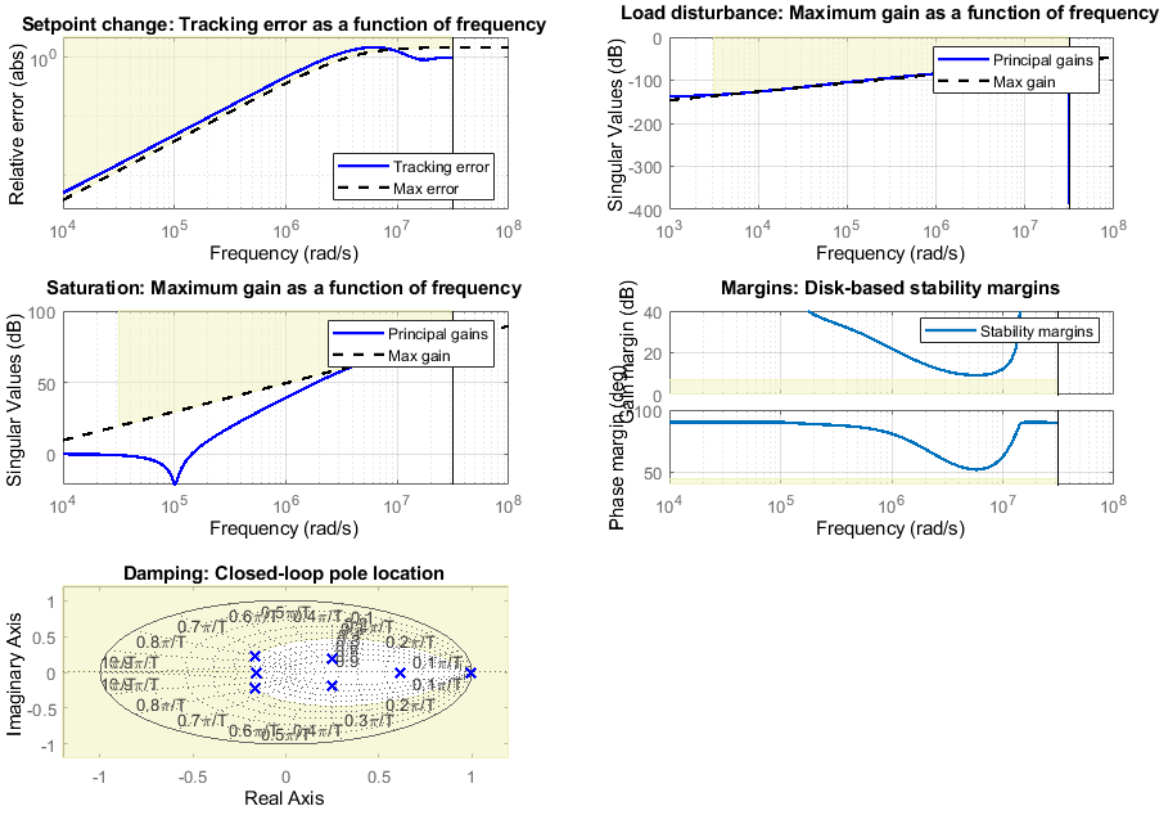
```
FIR = tunableTF('FIR',1,1,Tm);
% Fix denominator to z^n
FIR.Denominator.Value = [1 0];
FIR.Denominator.Free = false;
setBlockParam(ST0,'FIR',FIR);
```

Note that `slTuner` automatically parameterizes the feedback compensator as a third-order state-space model (the order specified in the Simulink block). Next tune the feedforward and feedback compensators with `systune`. Treat the damping and margin requirements as hard constraints and try to best meet the remaining requirements.

```
rng(0)
topt = systuneOptions('RandomStart',6);
ST = systune(ST0,[Req1 Req2 Req3],[Req4 Req5],topt);

Final: Soft = 1.29, Hard = 0.90217, Iterations = 369
Final: Soft = 1.29, Hard = 0.99776, Iterations = 405
Final: Soft = 1.31, Hard = 0.99514, Iterations = 530
Final: Soft = 1.29, Hard = 0.94586, Iterations = 351
Final: Soft = 1.29, Hard = 0.99439, Iterations = 385
Final: Soft = 1.29, Hard = 0.99805, Iterations = 405
Final: Soft = 1.29, Hard = 0.97975, Iterations = 488
```

The best design satisfies the hard constraints (`Hard` less than 1) and nearly satisfies the other constraints (`Soft` close to 1). Verify this graphically by plotting the tuned responses for each requirement.

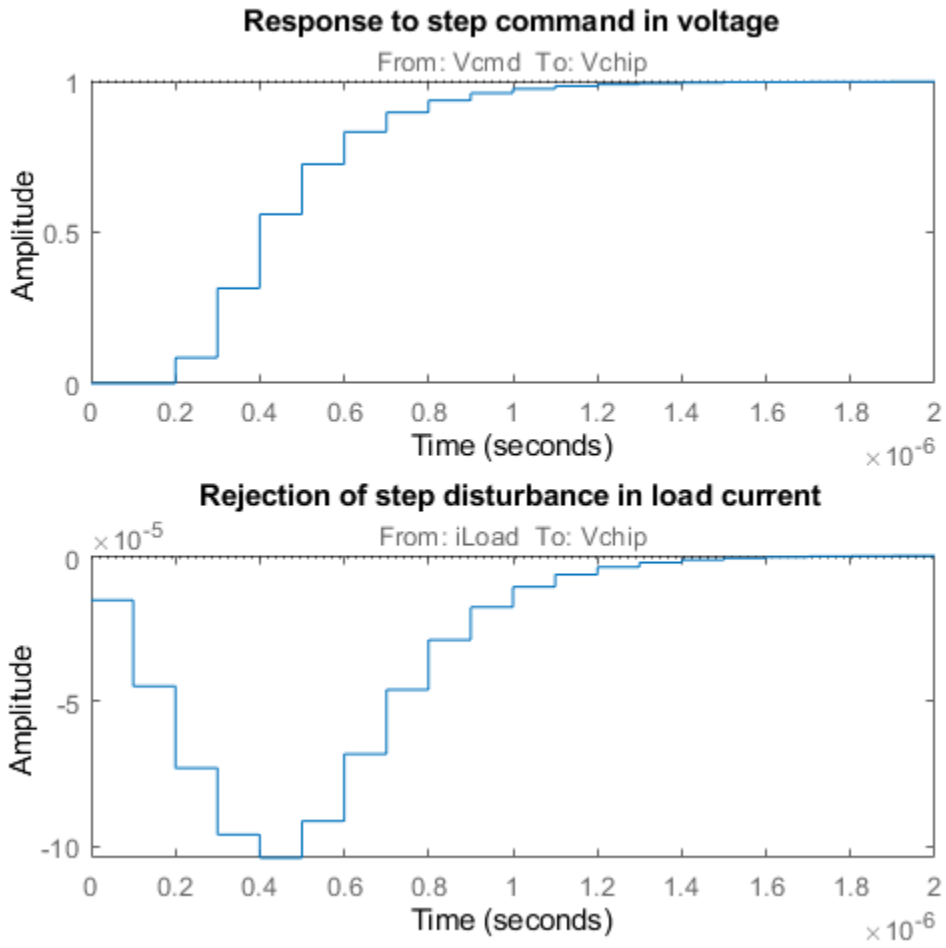
```
figure('Position',[10,10,1071,714])
viewGoal([Req1 Req2 Req3 Req4 Req5],ST)
```



Validation

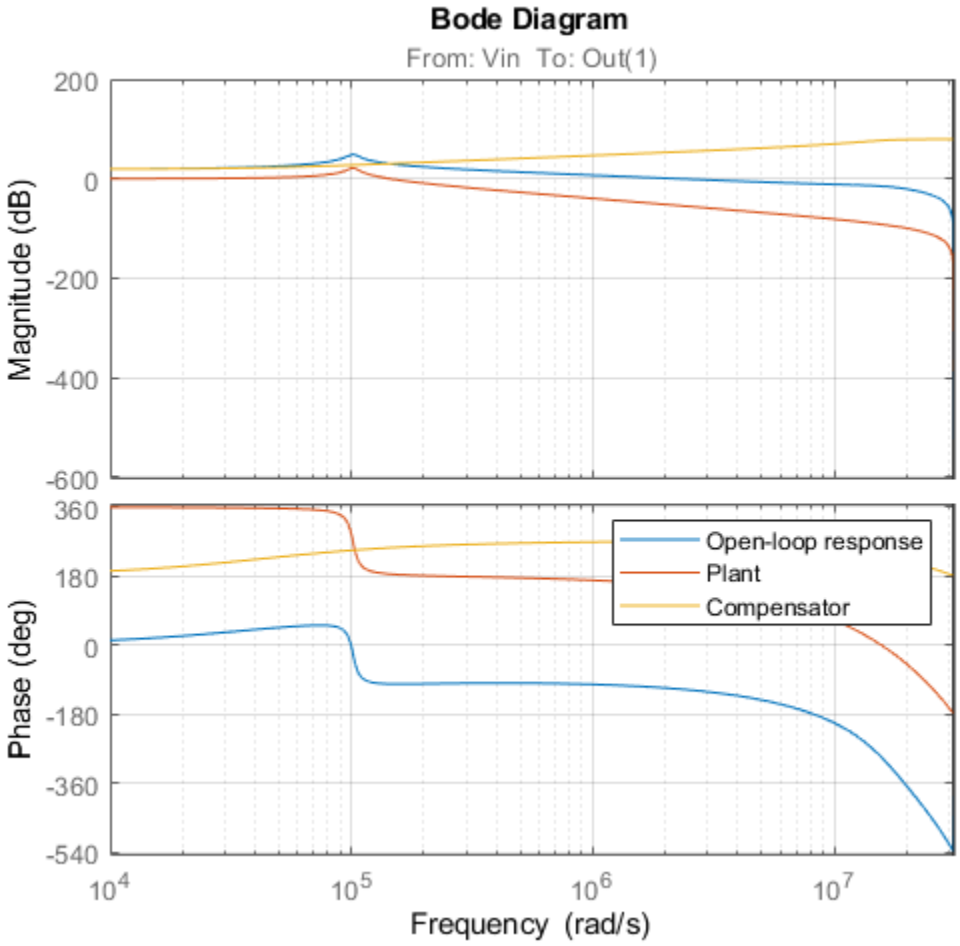
First validate the design in the linear domain using the `slTuner` interface. Plot the closed-loop response to a step command `Vcmd` and a step disturbance `iLoad`.

```
figure('Position',[100,100,560,500])
subplot(2,1,1)
step(getIOTransfer(ST,'Vcmd','Vchip'),20*Tm)
title('Response to step command in voltage')
subplot(2,1,2)
step(getIOTransfer(ST,'iLoad','Vchip'),20*Tm)
title('Rejection of step disturbance in load current')
```



Use `getLoopTransfer` to compute the open-loop response at the plant input and superimpose the plant and feedback compensator responses.

```
clf
L = getLoopTransfer(ST,'Vin',-1);
C = getBlockValue(ST,'compensator');
bodeplot(L,psmodel(2),C(2),{1e-3/Tm pi/Tm})
grid
legend('Open-loop response','Plant','Compensator')
```



The controller achieves the desired bandwidth and the responses are fast enough. Apply the tuned parameter values to the Simulink model and simulate the tuned responses.

```
writeBlockValue(ST)
```

The results from the nonlinear simulation appear below. Note that the control signal `Vin` remains approximately within $\pm 12V$ saturation bounds for the setpoint tracking portion of the simulation.

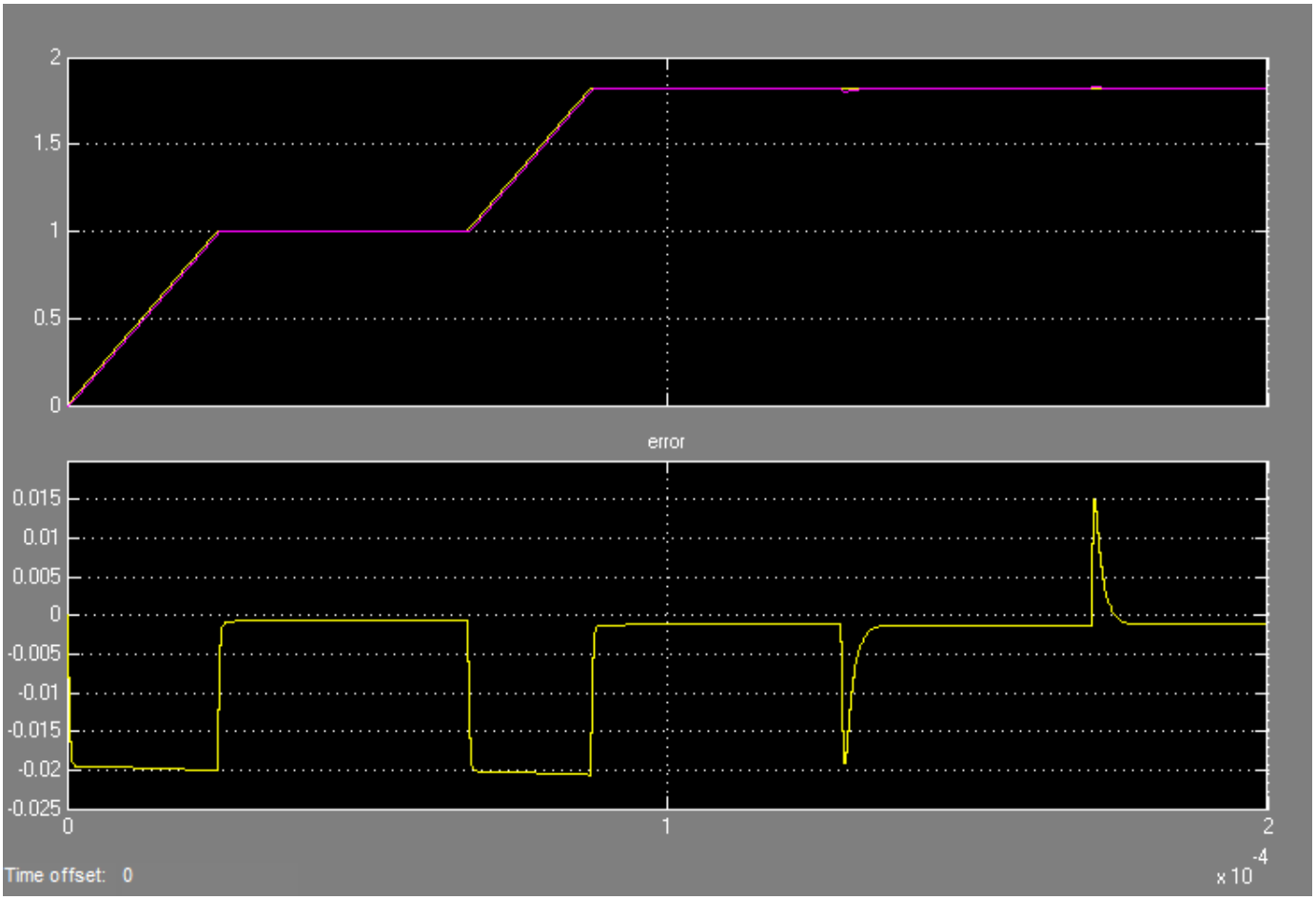


Figure 1: Response to ramp command and step load disturbances.

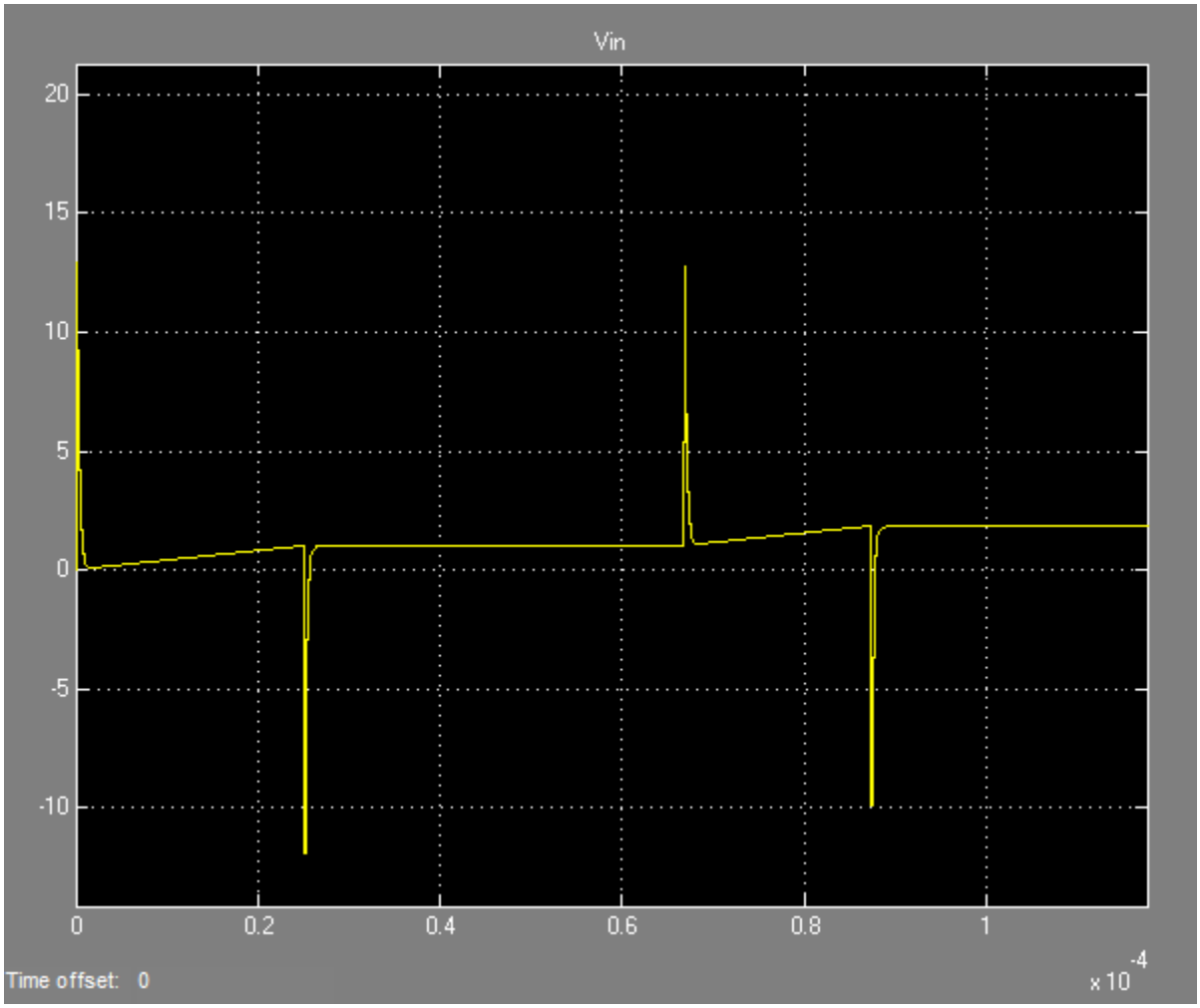


Figure 2: Amplitude of input voltage `Vin` during setpoint tracking phase.

See Also

`systune (slTuner)` | `slTuner` | `TuningGoal.Tracking` | `TuningGoal.Gain` | `TuningGoal.Margins`

Related Examples

- “MIMO Control of Diesel Engine”

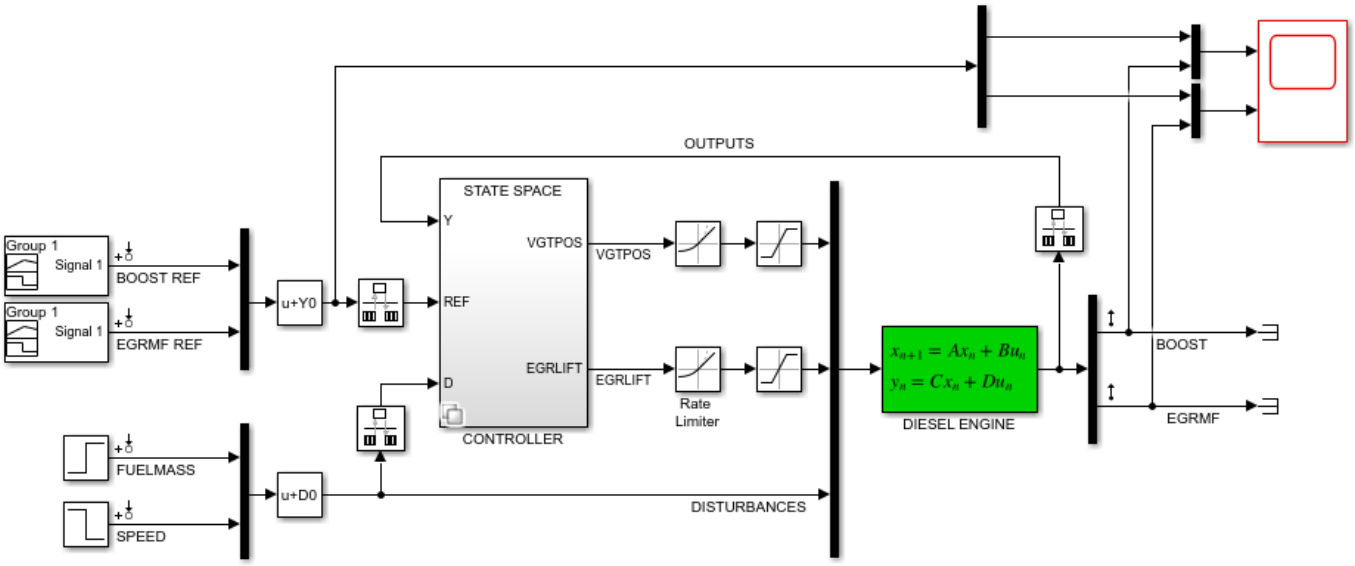
MIMO Control of Diesel Engine

This example uses `systune` to design and tune a MIMO controller for a Diesel engine. The controller is tuned in discrete time for a single operating condition.

Diesel Engine Model

Modern Diesel engines use a variable geometry turbocharger (VGT) and exhaust gas recirculation (EGR) to reduce emissions. Tight control of the VGT boost pressure and EGR massflow is necessary to meet strict emission targets. This example shows how to design and tune a MIMO controller that regulates these two variables when the engine operates at 2100 rpm with a fuel mass of 12 mg per injection-cylinder.

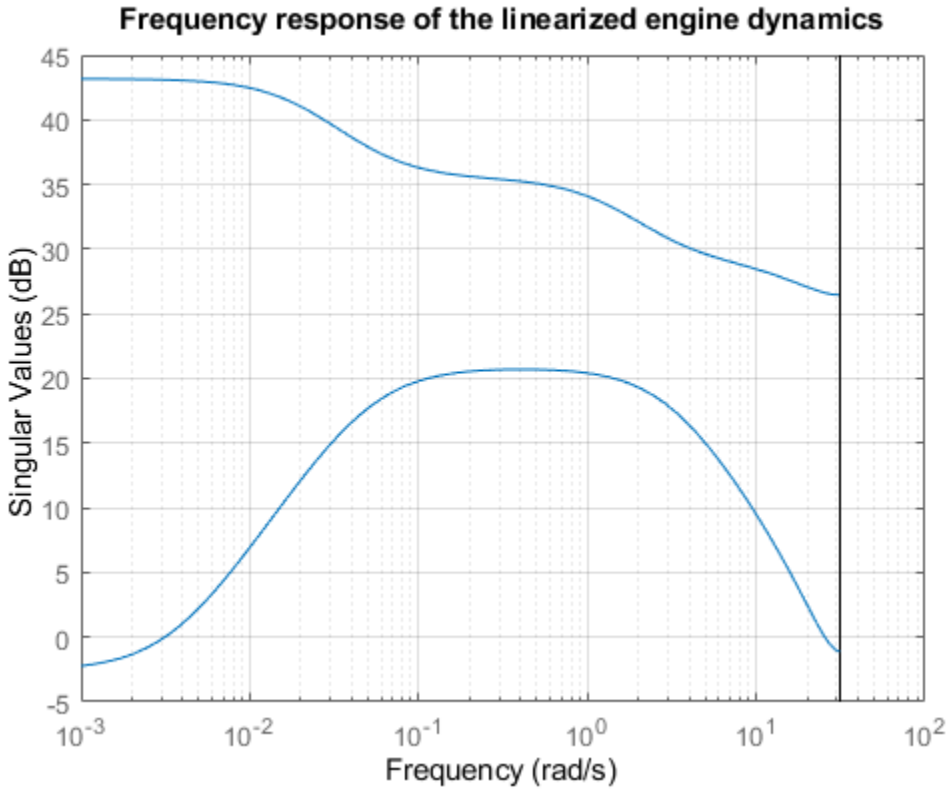
```
open_system('rct_diesel')
```



The VGT/EGR control system is modeled in Simulink. The controller adjusts the positions `EGRLIFT` and `VGTPOS` of the EGR and VGT valves. It has access to the boost pressure and EGR massflow targets and measured values, as well as fuel mass and engine speed measurements. Both valves have rate and saturation limits. The plant model is sampled every 0.1 seconds and the control signals `EGRLIFT` and `VGTPOS` are refreshed every 0.2 seconds. This example considers step changes of +10 KPa in boost pressure and +3 g/s in EGR massflow, and disturbances of +5 mg in fuel mass and -200 rpm in speed.

For the operating condition under consideration, we used System Identification to derive a linear model of the engine from experimental data. The frequency response from the manipulated variables `EGRLIFT` and `VGTPOS` to the controlled variables `BOOST` and `EGR MF` appears below. Note that the plant is ill conditioned at low frequency which makes independent control of boost pressure and EGR massflow difficult.

```
sigma(Plant(:,1:2)), grid
title('Frequency response of the linearized engine dynamics')
```



Control Objectives

There are two main control objectives:

1. Respond to step changes in boost pressure and EGR massflow in about 5 seconds with minimum cross-coupling
2. Be insensitive to (small) variations in speed and fuel mass.

Use a tracking requirement for the first objective. Specify the amplitudes of the step changes to ensure that cross-couplings are small *relative* to these changes.

```
% 5-second response time, steady-state error less than 5%
TR = TuningGoal.Tracking({'BOOST REF';'EGRMF REF'},{'BOOST';'EGRMF'},5,0.05);
TR.Name = 'Setpoint tracking';
TR.InputScaling = [10 3];
```

For the second objective, treat the speed and fuel mass variations as step disturbances and specify the peak amplitude and settling time of the resulting variations in boost pressure and EGR massflow. Also specify the signal amplitudes to properly reflect the relative contribution of each disturbance.

```
% Peak<0.5, settling time<5
DR = TuningGoal.StepRejection({'FUELMASS';'SPEED'},{'BOOST';'EGRMF'},0.5,5);
DR.Name = 'Disturbance rejection';
DR.InputScaling = [5 200];
DR.OutputScaling = [10 3];
```

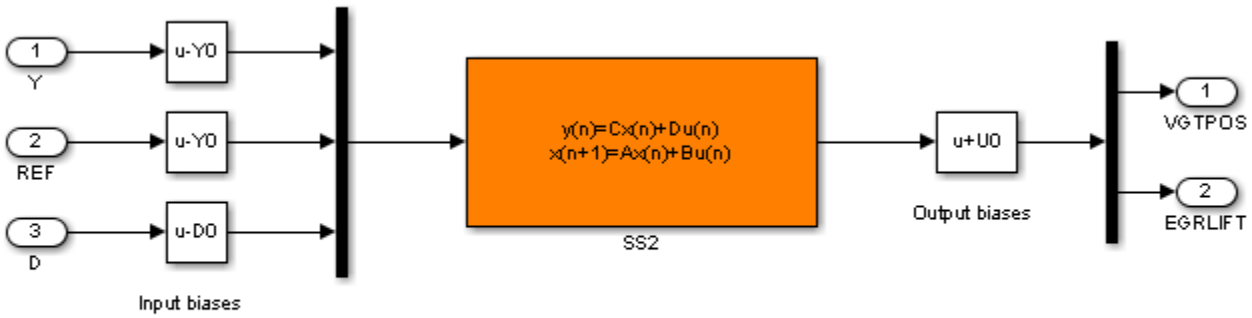
To provide adequate robustness to unmodeled dynamics and aliasing, limit the control bandwidth and impose sufficient stability margins at both the plant inputs and outputs. Because we are dealing with

2-by-2 MIMO feedback loops, this requirement guarantees stability for gain or phase variations in each feedback channel. The gain or phase can change in both channels simultaneously, and by a different amount in each channel. See “Stability Margins in Control System Tuning” and `TuningGoal.Margins` for details.

```
% Roll off of -20 dB/dec past 1 rad/s
RO = TuningGoal.MaxLoopGain({'EGRLIFT','VGTPOS'},1,1);
RO.LoopScaling = 'off';
RO.Name = 'Roll-off';

% 7 dB of gain margin and 45 degrees of phase margin
M1 = TuningGoal.Margins({'EGRLIFT','VGTPOS'},7,45);
M1.Name = 'Plant input';
M2 = TuningGoal.Margins('DIESEL ENGINE',7,45);
M2.Name = 'Plant output';
```

Tuning of Blackbox MIMO Controller

Without a-priori knowledge of a suitable control structure, first try "blackbox" state-space controllers of various orders. The plant model has four states, so try a controller of order four or less. Here we tune a second-order controller since the "SS2" block in the Simulink model has two states.



2-DOF MIMO PID with disturbance feedforward

Figure 1: Second-order blackbox controller.

Use the `slTuner` interface to configure the Simulink model for tuning. Mark the block "SS2" as tunable, register the locations where to assess margins and loop shapes, and specify that linearization and tuning should be performed at the controller sampling rate.

```
ST0 = slTuner('rct_diesel','SS2');
ST0.Ts = 0.2;
addPoint(ST0,{'EGRLIFT','VGTPOS','DIESEL ENGINE'})
```

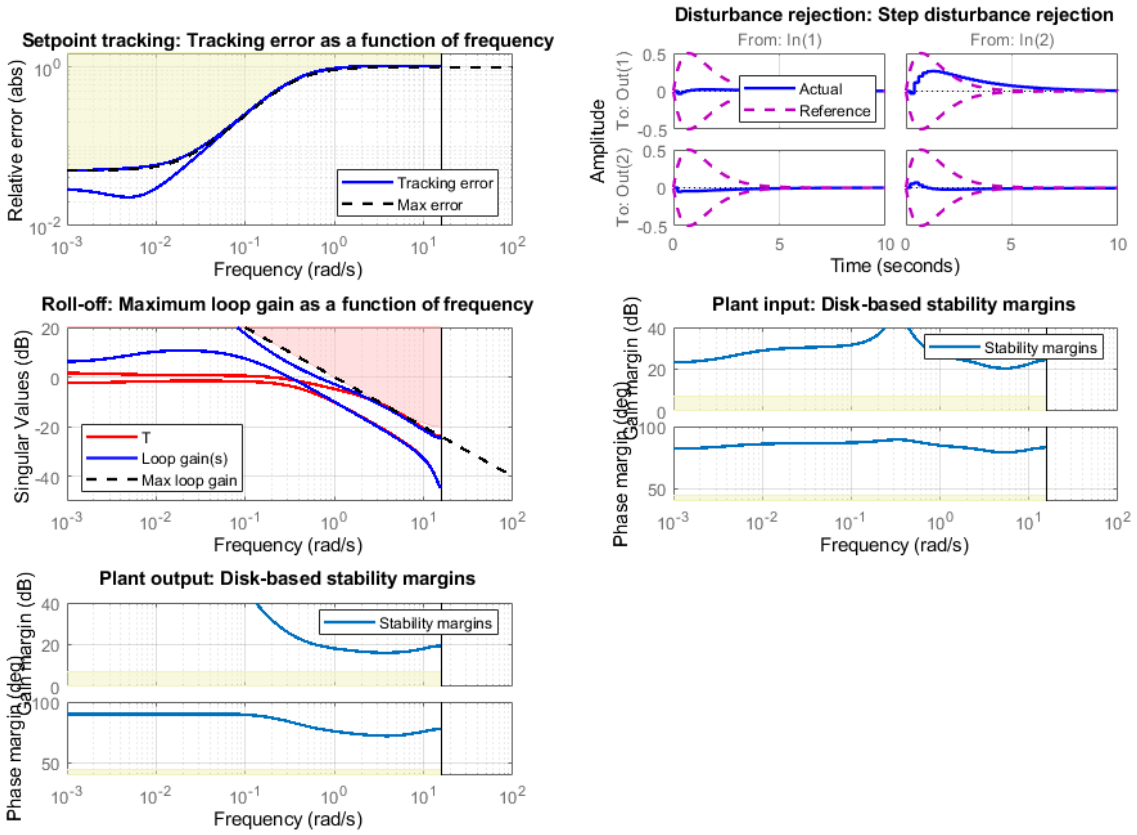
Now use `systune` to tune the state-space controller subject to our control objectives. Treat the stability margins and roll-off target as hard constraints and try to best meet the remaining objectives (soft goals). Randomize the starting point to reduce exposure to undesirable local minima.

```
Opt = systuneOptions('RandomStart',2);
rng(0), ST1 = systune(ST0,[TR DR],[M1 M2 RO],Opt);
```

```
Final: Soft = 1.05, Hard = 0.90488, Iterations = 435
Final: Soft = 1.05, Hard = 0.9666, Iterations = 581
Final: Soft = 1.12, Hard = 0.91154, Iterations = 496
```

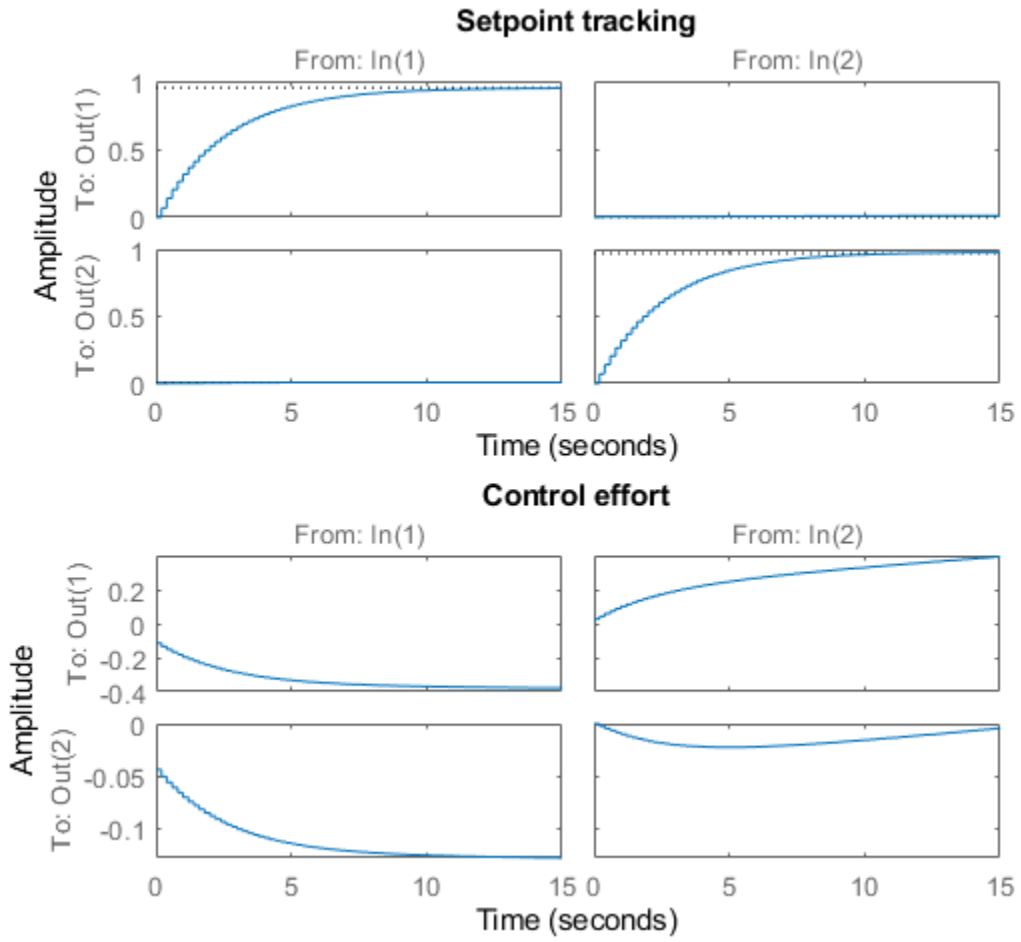
All requirements are nearly met (a requirement is satisfied when its normalized value is less than 1). Verify this graphically.

```
figure('Position',[10,10,1071,714])
viewGoal([TR DR RO M1 M2],ST1)
```

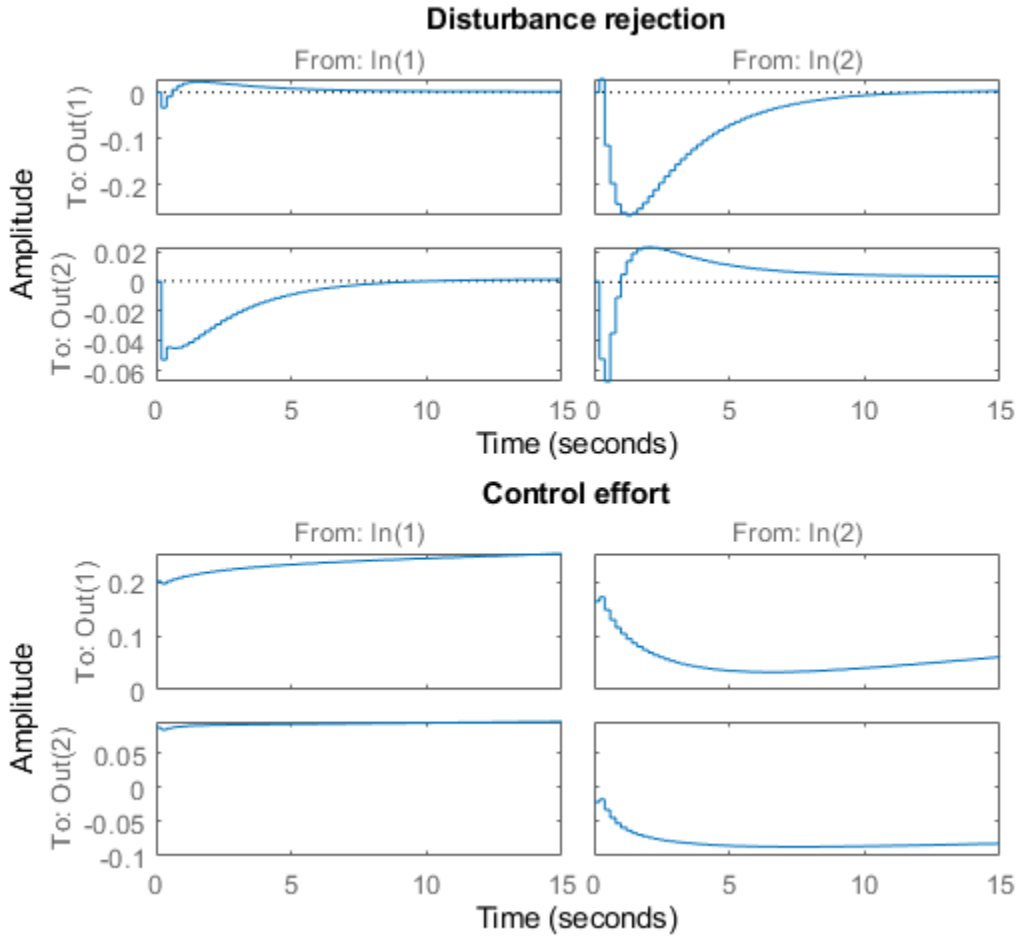


Plot the setpoint tracking and disturbance rejection responses. Scale by the signal amplitudes to show normalized effects (boost pressure changes by +10 KPa, EGR massflow by +3 g/s, fuel mass by +5 mg, and speed by -200 rpm).

```
figure('Position',[100,100,560,500])
T1 = getIOTransfer(ST1,{'BOOST REF';'EGRMF REF'},{'BOOST','EGRMF','EGRLIFT','VGTPOS'});
T1 = diag([1/10 1/3 1 1]) * T1 * diag([10 3]);
subplot(211), step(T1(1:2,:),15), title('Setpoint tracking')
subplot(212), step(T1(3:4,:),15), title('Control effort')
```




```
D1 = getIOTransfer(ST1,{'FUELMASS';'SPEED'},{'BOOST','EGRMF','EGRLIFT','VGTPOS'});
D1 = diag([1/10 1/3 1 1]) * D1 * diag([5 -200]);
subplot(211), step(D1(1:2,:),15), title('Disturbance rejection')
subplot(212), step(D1(3:4,:),15), title('Control effort')
```

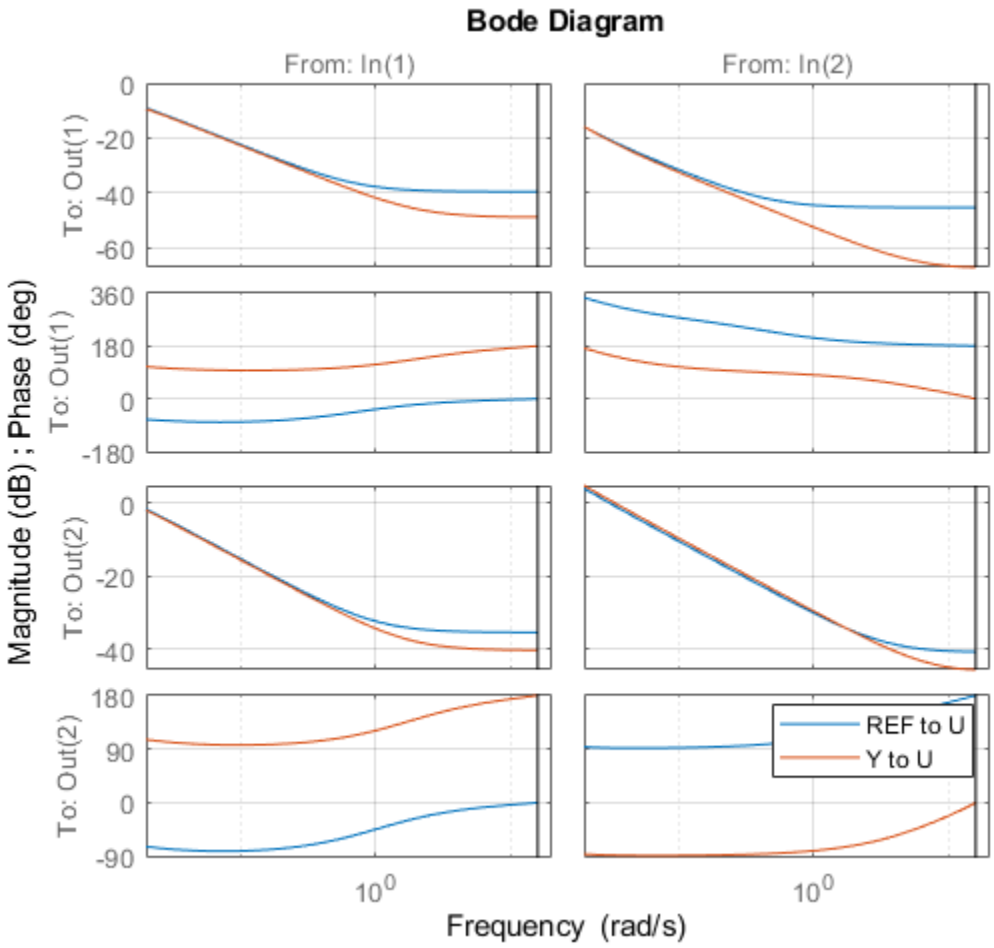


The controller responds in less than 5 seconds with minimum cross-coupling between the `BOOST` and `EGRMF` variables.

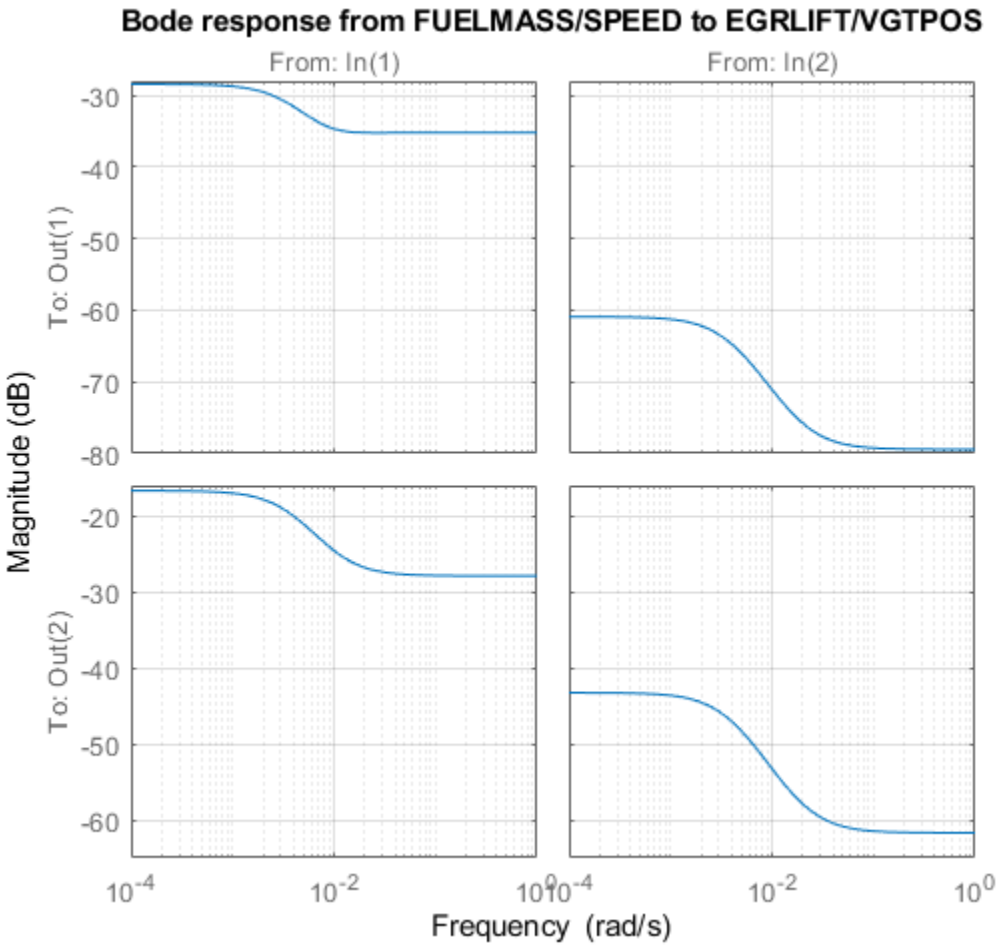
Tuning of Simplified Control Structure

The state-space controller could be implemented as is, but it is often desirable to boil it down to a simpler, more familiar structure. To do this, get the tuned controller and inspect its frequency response

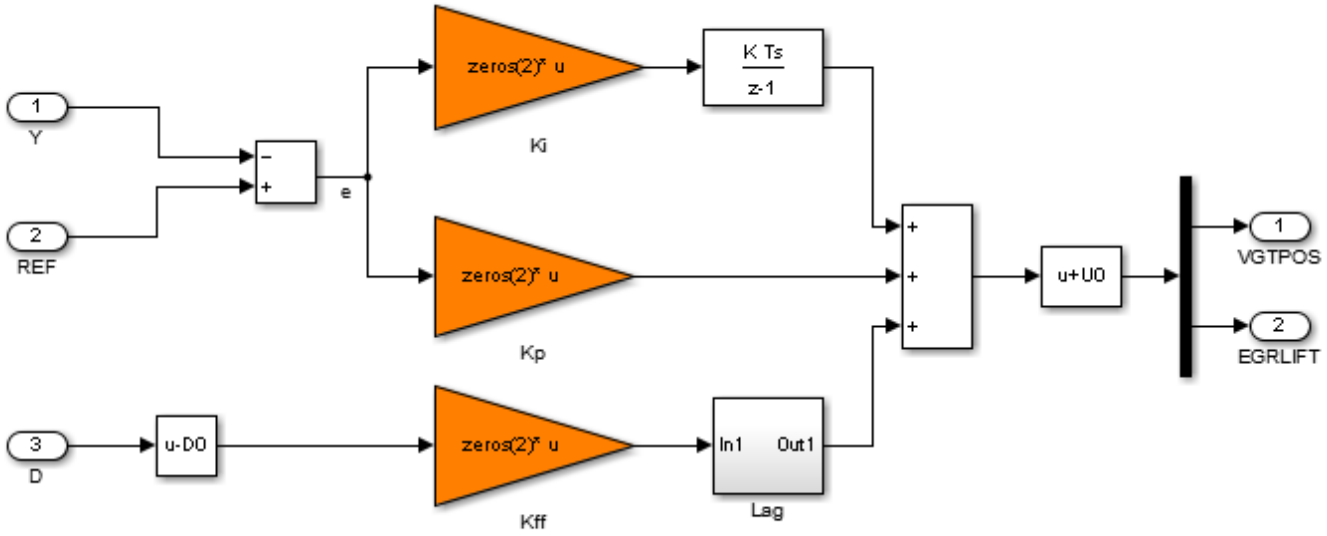
```
C = getBlockValue(ST1,'SS2');

clf
bode(C(:,1:2),C(:,3:4),{.02 20}), grid
legend('REF to U','Y to U')
```




```
bodemag(C(:,5:6)), grid
title('Bode response from FUELMASS/SPEED to EGRLIFT/VGTPOS')
```



The first plot suggests that the controller essentially behaves like a PI controller acting on REF-Y (the difference between the target and actual values of the controlled variables). The second plot suggests that the transfer from measured disturbance to manipulated variables could be replaced by a gain in series with a lag network. Altogether this suggests the following simplified control structure consisting of a MIMO PI controller with a first-order disturbance feedforward.



MIMO PID with disturbance feedforward

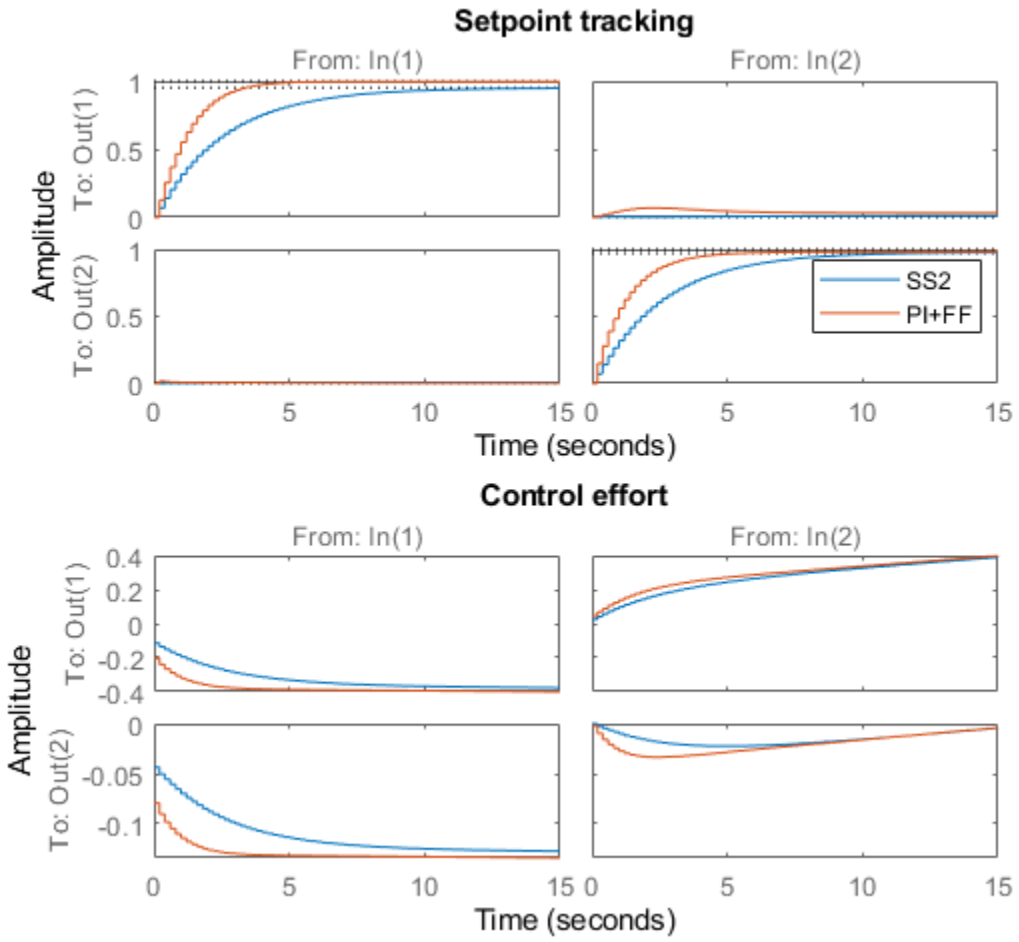
Figure 2: Simplified control structure.

Using variant subsystems, you can implement both control structures in the same Simulink model and use a variable to switch between them. Here setting `MODE=2` selects the MIMO PI structure. As before, use `systune` to tune the three 2-by-2 gain matrices `Kp`, `Ki`, `Kff` in the simplified control structure.

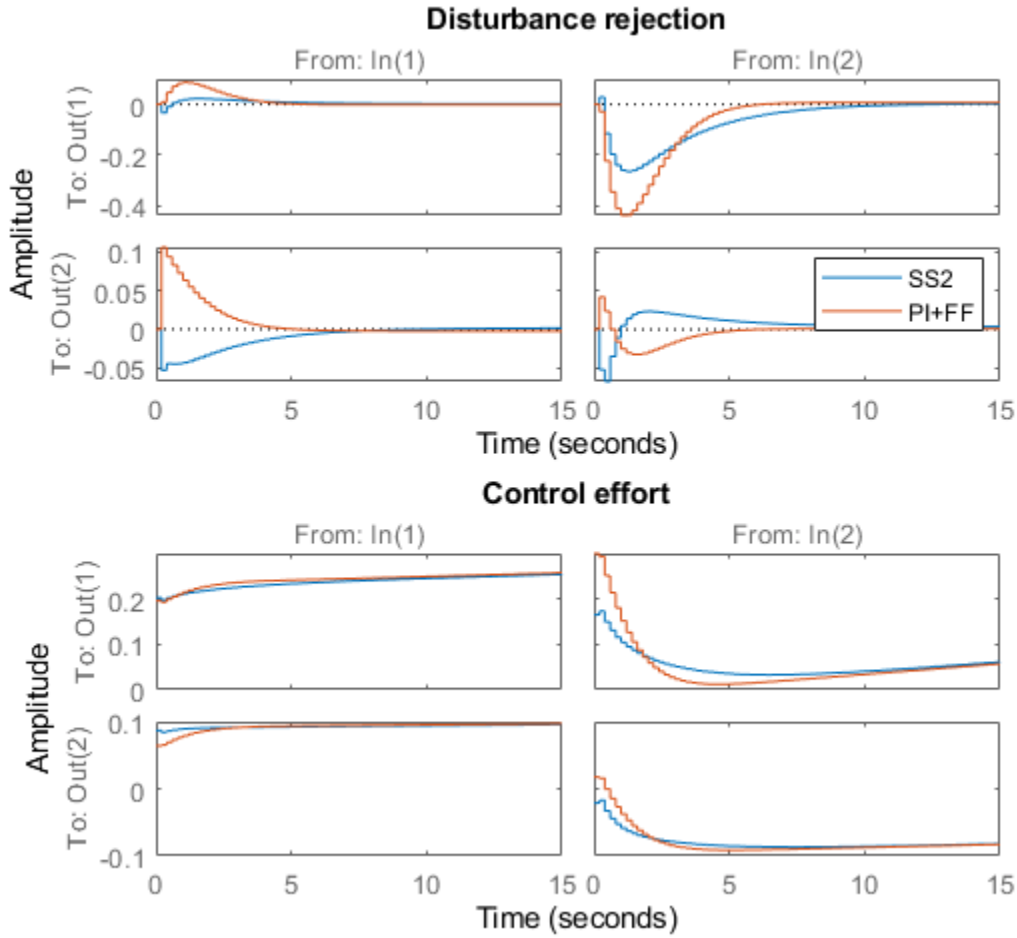
```
% Select "MIMO PI" variant in "CONTROLLER" block
MODE = 2;

% Configure tuning interface
ST0 = slTuner('rct_diesel',{'Kp','Ki','Kff'});
ST0.Ts = 0.2;
addPoint(ST0,{'EGRLIFT','VGTPOS','DIESEL ENGINE'})

% Tune MIMO PI controller.
ST2 = systune(ST0,[TR DR],[M1 M2 RO]);

Final: Soft = 1.09, Hard = 0.99933, Iterations = 197
```

Again all requirements are nearly met. Plot the closed-loop responses and compare with the state-space design.

```
clf
T2 = getIOTransfer(ST2,{'BOOST REF';'EGRMF REF'},{'BOOST','EGRMF','EGRLIFT','VGTPOS'});
T2 = diag([1/10 1/3 1 1]) * T2 * diag([10 3]);
subplot(211), step(T1(1:2,:),T2(1:2,:),15), title('Setpoint tracking')
legend('SS2','PI+FF')
subplot(212), step(T1(3:4,:),T2(3:4,:),15), title('Control effort')
```




```
D2 = getIOTransfer(ST2,{'FUELMASS';'SPEED'},{'BOOST','EGRMF','EGRLIFT','VGTPOS'});
D2 = diag([1/10 1/3 1 1]) * D2 * diag([5 -200]);
subplot(211), step(D1(1:2,:),D2(1:2,:),15), title('Disturbance rejection')
legend('SS2','PI+FF')
subplot(212), step(D1(3:4,:),D2(3:4,:),15), title('Control effort')
```



The blackbox and simplified control structures deliver similar performance. Inspect the tuned values of the PI and feedforward gains.

```
showTunable(ST2)
```

```
Block 1: rct_diesel/CONTROLLER/MIMO PID/Kp =

  D =
              u1          u2
   y1   -0.007959  -0.0004826
   y2    -0.02043     0.01568

Name: Kp
Static gain.

-----------------------------------

Block 2: rct_diesel/CONTROLLER/MIMO PID/Ki =

  D =
            u1        u2
```

```
   y1  -0.01051  -0.01439
   y2  -0.03029   0.04719

Name: Ki
Static gain.

-----------------------------------

Block 3: rct_diesel/CONTROLLER/MIMO PID/Kff =

  D =
              u1          u2
   y1     0.01283  -8.854e-05
   y2     0.03908   -0.001497

Name: Kff
Static gain.
```

Nonlinear Validation

To validate the MIMO PI controller in the Simulink model, push the tuned controller parameters to Simulink and run the simulation.

```
writeBlockValue(ST2)
```

The simulation results are shown below and confirm that the controller adequately tracks setpoint changes in boost pressure and EGR massflow and quickly rejects changes in fuel mass (at t=90) and in speed (at t=110).

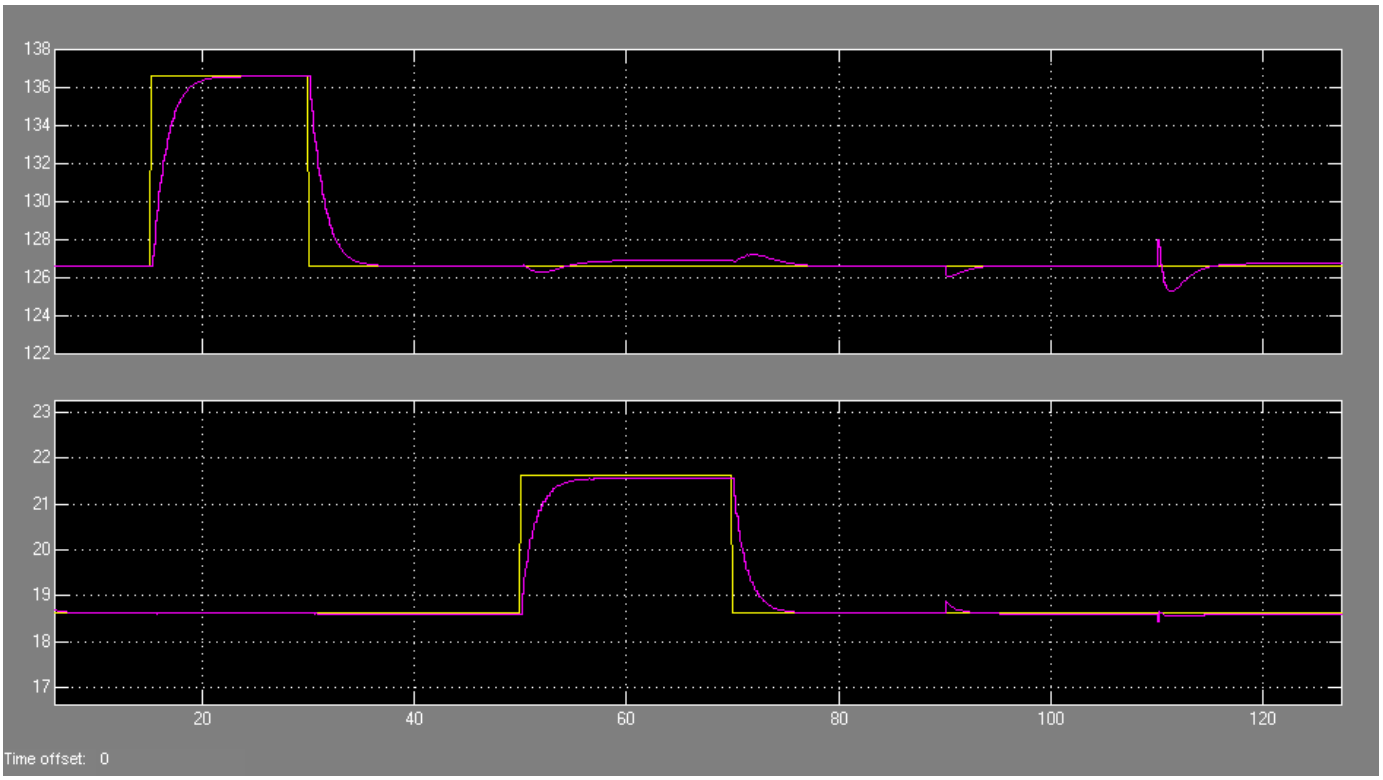


Figure 3: Simulation results with simplified controller.

See Also

`systune (slTuner)` | `slTuner` | `TuningGoal.Tracking` | `TuningGoal.StepRejection` | `TuningGoal.MaxLoopGain` | `TuningGoal.Margins`

Related Examples

- “Digital Control of Power Stage Voltage”

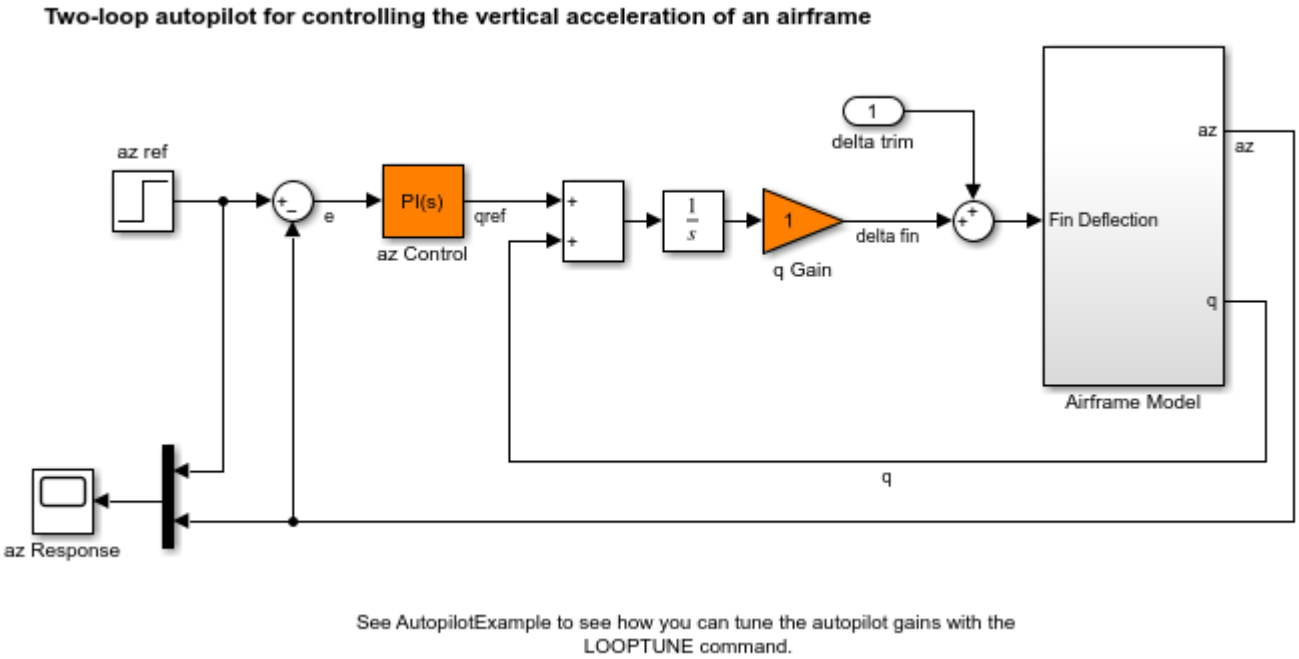
Tuning of a Two-Loop Autopilot

This example shows how to use Simulink Control Design to tune a two-loop autopilot controlling the pitch rate and vertical acceleration of an airframe.

Model of Airframe Autopilot

The airframe dynamics and the autopilot are modeled in Simulink.

```
open_system('rct_airframe1')
```



The autopilot consists of two cascaded loops. The inner loop controls the pitch rate q, and the outer loop controls the vertical acceleration `az` in response to the pilot stick command `azref`. In this architecture, the tunable elements include the PI controller gains ("az Control" block) and the pitch-rate gain ("q Gain" block). The autopilot must be tuned to respond to a step command `azref` in about 1 second with minimal overshoot. In this example, we tune the autopilot gains for one flight condition corresponding to zero incidence and a speed of 984 m/s.

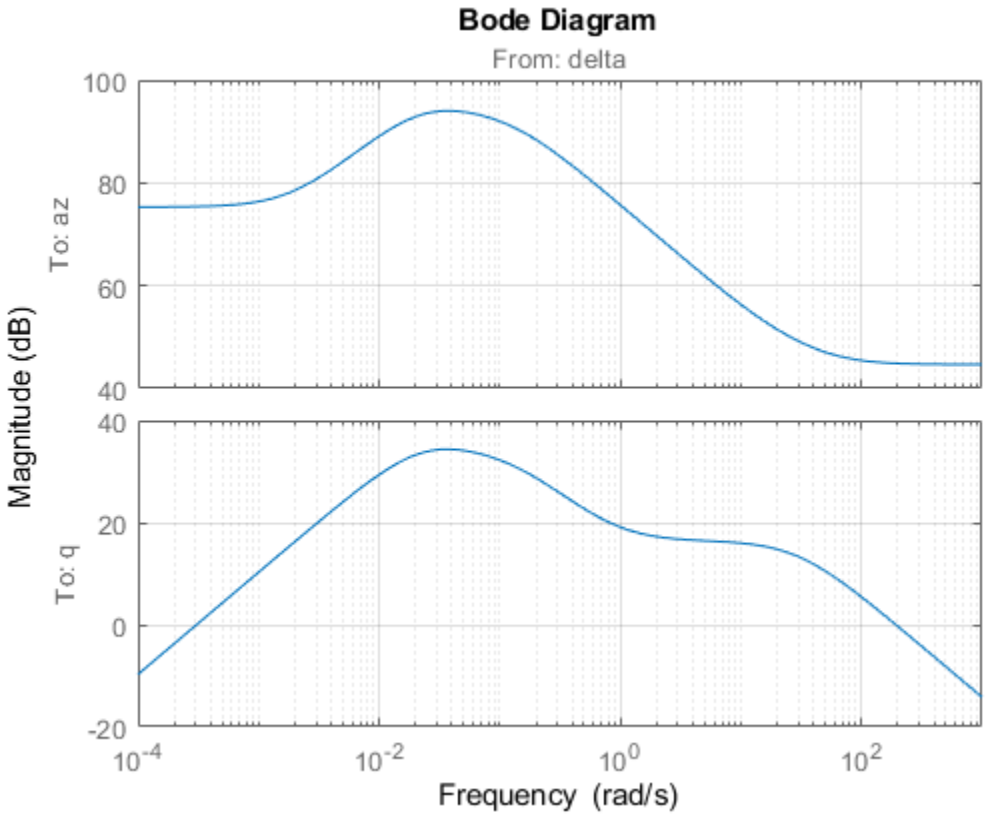
To analyze the airframe dynamics, trim the airframe for $\alpha = 0$ and $V = 984m/s$. The trim condition corresponds to zero normal acceleration and pitching moment (w and q steady). Use `findop` to compute the corresponding closed-loop operating condition. Note that we added a "delta trim" input port so that `findop` can adjust the fin deflection to produce the desired equilibrium of forces and moments.

```
opspec = operspec('rct_airframe1');

% Specify trim condition
% Xe,Ze: known, not steady
opspec.States(1).Known = [1;1];
opspec.States(1).SteadyState = [0;0];
```

```
% u,w: known, w steady
opspec.States(3).Known = [1 1];
opspec.States(3).SteadyState = [0 1];
% theta: known, not steady
opspec.States(2).Known = 1;
opspec.States(2).SteadyState = 0;
% q: unknown, steady
opspec.States(4).Known = 0;
opspec.States(4).SteadyState = 1;
% integrator states unknown, not steady
opspec.States(5).SteadyState = 0;
opspec.States(6).SteadyState = 0;

op = findop('rct_airframe1',opspec);


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

opreport =


 Operating point search report for the Model rct_airframe1.
 (Time-Varying Components Evaluated at time t=0)

Operating point specifications were successfully met.
States:
----------
     <strong>Min</strong>         <strong>x</strong>          <strong>Max</strong>     <strong>dx
    <strong>_____</strong>    <strong>__________</strong>    <strong>_____</strong>    <strong>__

(1.) rct_airframe1/Airframe Model/Aerodynamics & Equations of Motion/ Equations of Motion (Body A
        0             0        0    -Inf             984     Inf
    -3048         -3048    -3048    -Inf               0     Inf
(2.) rct_airframe1/Airframe Model/Aerodynamics & Equations of Motion/ Equations of Motion (Body A
        0             0        0    -Inf      -0.0097235     Inf
(3.) rct_airframe1/Airframe Model/Aerodynamics & Equations of Motion/ Equations of Motion (Body A
      984           984      984    -Inf           22.69     Inf
        0             0        0       0     -1.4371e-11       0
(4.) rct_airframe1/Airframe Model/Aerodynamics & Equations of Motion/ Equations of Motion (Body A
     -Inf    -0.0097235      Inf       0      1.1477e-16       0
(5.) rct_airframe1/Integrator
     -Inf    0.00070807      Inf    -Inf      -0.0097235     Inf
(6.) rct_airframe1/az Control/Integrator/Continuous/Integrator
     -Inf             0      Inf    -Inf      0.00024207     Inf

Inputs:
----------
    <strong>Min</strong>         <strong>u</strong>          <strong>Max</strong>
    <strong>____</strong>    <strong>__________</strong>    <strong>___</strong>

(1.) rct_airframe1/delta trim
    -Inf    0.00070807    Inf

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

Linearize the "Airframe Model" block for the computed trim condition `op` and plot the gains from the fin deflection `delta` to `az` and `q`:

```
G = linearize('rct_airframe1','rct_airframe1/Airframe Model',op);
G.InputName = 'delta';
G.OutputName = {'az','q'};

bodemag(G), grid
```



Note that the airframe model has an unstable pole:

```
pole(G)
```

```
ans =

   -0.0320
   -0.0255
    0.1253
  -29.4685
```

Frequency-Domain Tuning with LOOPTUNE

You can use the `looptune` function to automatically tune multi-loop control systems subject to basic requirements such as integral action, adequate stability margins, and desired bandwidth. To apply `looptune` to the autopilot model, create an instance of the `slTuner` interface and designate the

Simulink blocks "az Control" and "q Gain" as tunable. Also specify the trim condition `op` to correctly linearize the airframe dynamics.

```
ST0 = slTuner('rct_airframe1',{'az Control','q Gain'},op);
```

Mark the reference, control, and measurement signals as points of interest for analysis and tuning.

```
addPoint(ST0,{'az ref','delta fin','az','q'});
```

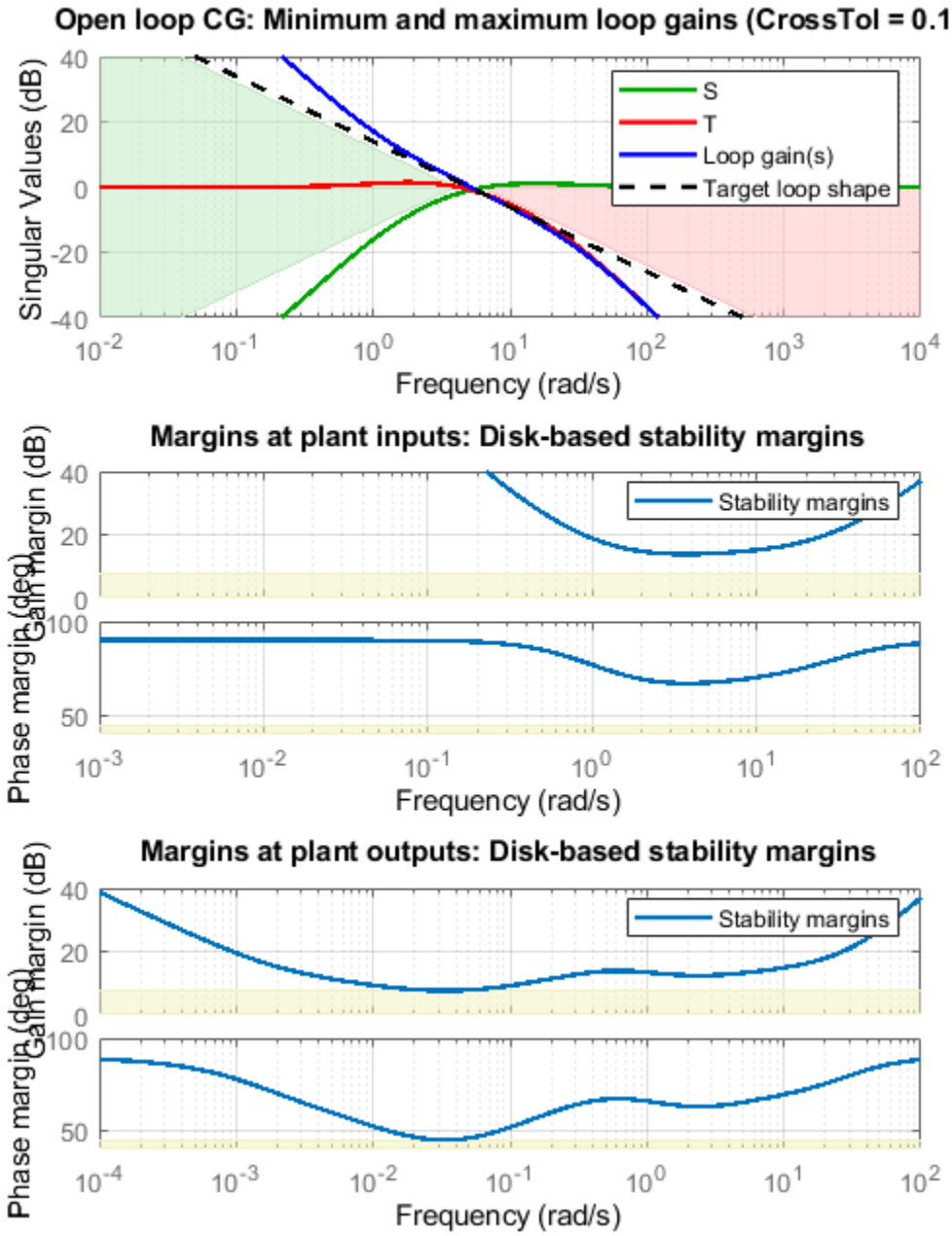
Finally, tune the control system parameters to meet the 1 second response time requirement. In the frequency domain, this roughly corresponds to a gain crossover frequency `wc = 5` rad/s for the open-loop response at the plant input "delta fin".

```
wc = 5;
Controls = 'delta fin';
Measurements = {'az','q'};
[ST,gam,Info] = looptune(ST0,Controls,Measurements,wc);
```

```
Final: Peak gain = 1.01, Iterations = 71
```

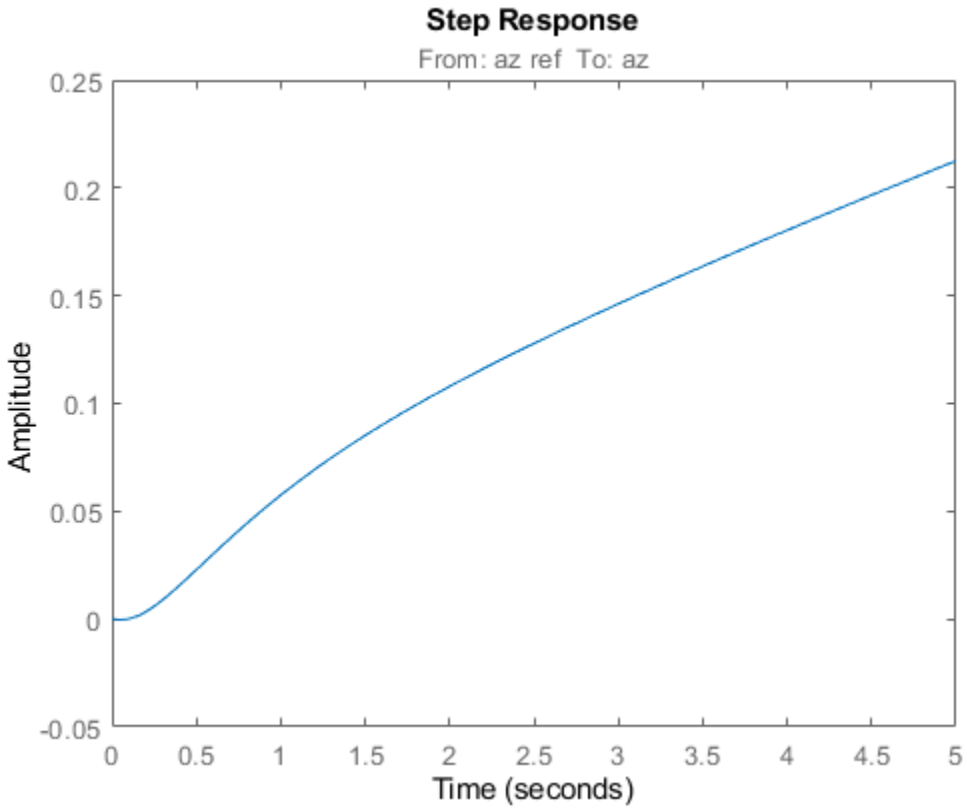
The requirements are normalized so a final value near 1 means that all requirements are met. Confirm this by graphically validating the design.

```
figure('Position',[100,100,560,714])
loopview(ST,Info)
```



The first plot confirms that the open-loop response has integral action and the desired gain crossover frequency while the second plot shows that the MIMO stability margins are satisfactory (the blue curve should remain below the yellow bound). Next check the response from the step command `azref` to the vertical acceleration `az`:

```
T = getIOTransfer(ST,'az ref','az');
figure
step(T,5)
```



The acceleration `az` does not track `azref` despite the presence of an integrator in the loop. This is because the feedback loop acts on the two variables `az` and `q` and we have not specified which one should track `azref`.

Adding a Tracking Requirement

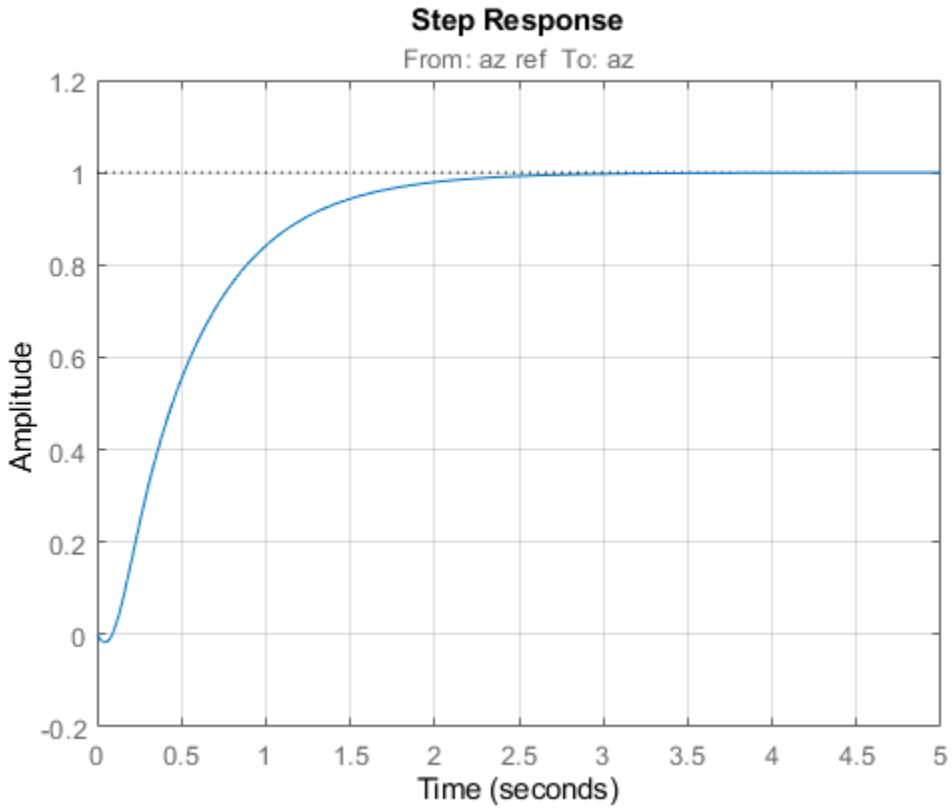
To remedy this issue, add an explicit requirement that `az` should follow the step command `azref` with a 1 second response time. Also relax the gain crossover requirement to the interval [3,12] to let the tuner find the appropriate gain crossover frequency.

```
TrackReq = TuningGoal.Tracking('az ref','az',1);
ST = looptune(ST0,Controls,Measurements,[3,12],TrackReq);
```

```
Final: Peak gain = 1.23, Iterations = 54
```

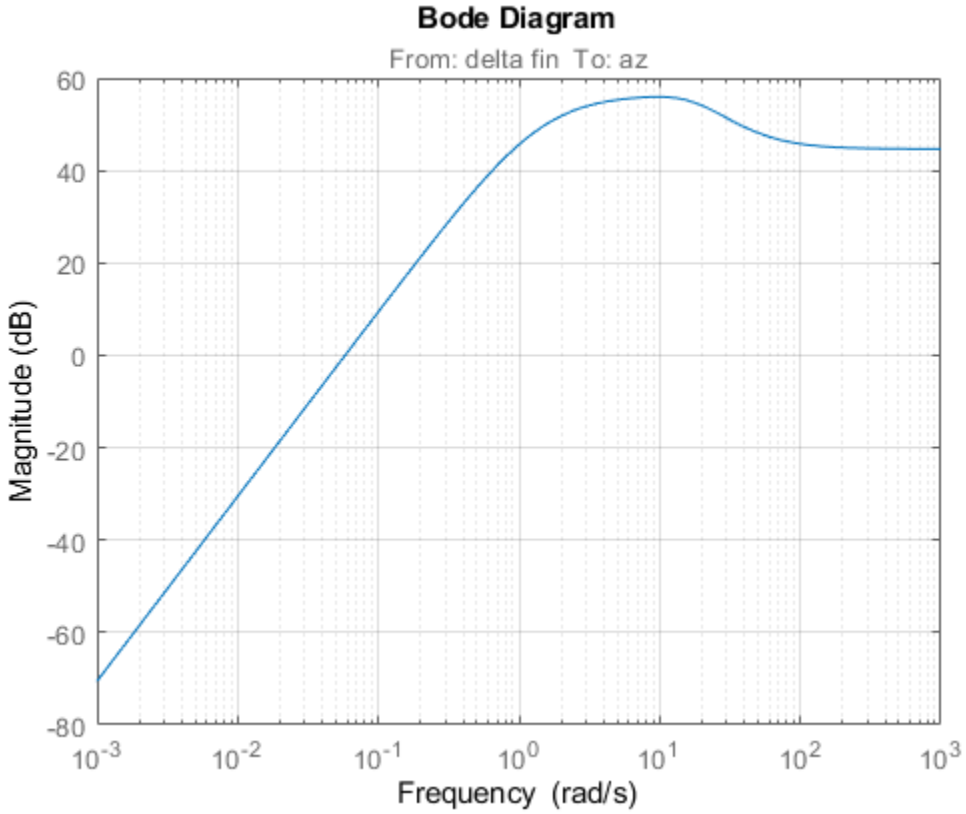
The step response from `azref` to `az` is now satisfactory:

```
Tr1 = getIOTransfer(ST,'az ref','az');
step(Tr1,5)
grid
```

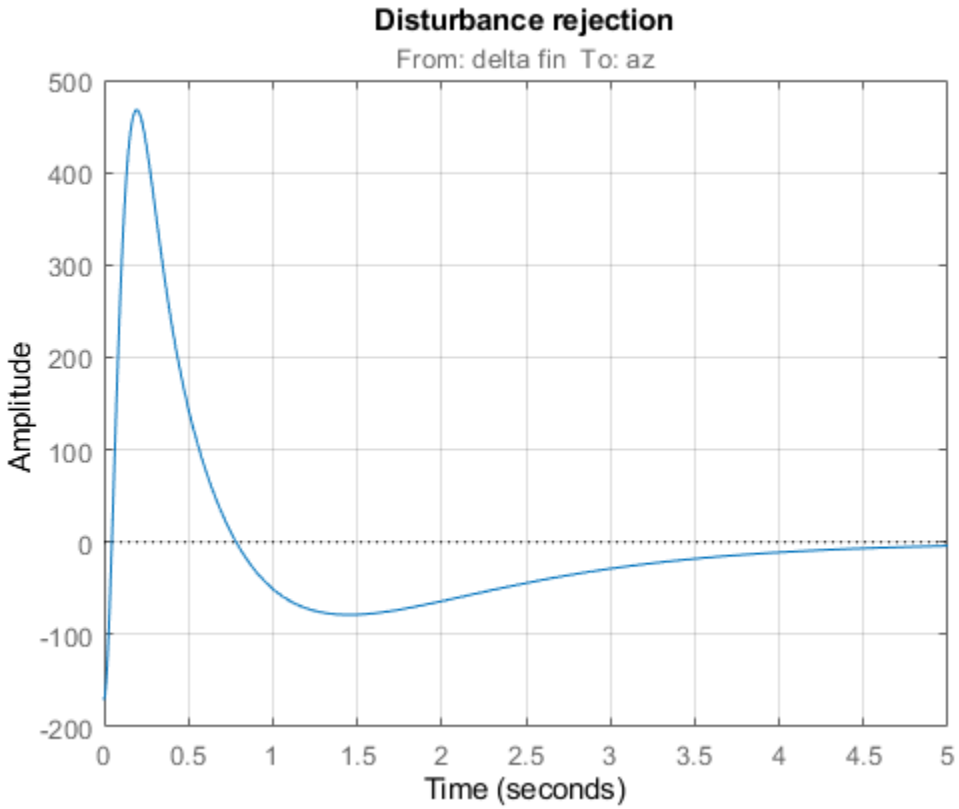


Also check the disturbance rejection characteristics by looking at the responses from a disturbance entering at the plant input

```
Td1 = getIOTransfer(ST,'delta fin','az');
bodemag(Td1)
grid
```




```
step(Td1,5)
grid
title('Disturbance rejection')
```



Use `showBlockValue` to see the tuned values of the PI controller and inner-loop gain

```
showBlockValue(ST)
```

```
AnalysisPoints_ =

  D =
       u1  u2  u3  u4
   y1   1   0   0   0
   y2   0   1   0   0
   y3   0   0   1   0
   y4   0   0   0   1

Name: AnalysisPoints_
Static gain.
----------------------------------
az_Control =

             1
  Kp + Ki * ---
             s

  with Kp = 0.00166, Ki = 0.0017

Name: az_Control
Continuous-time PI controller in parallel form.
----------------------------------
q_Gain =
```

```
  D =
         u1
   y1  1.985

Name: q_Gain
Static gain.
```

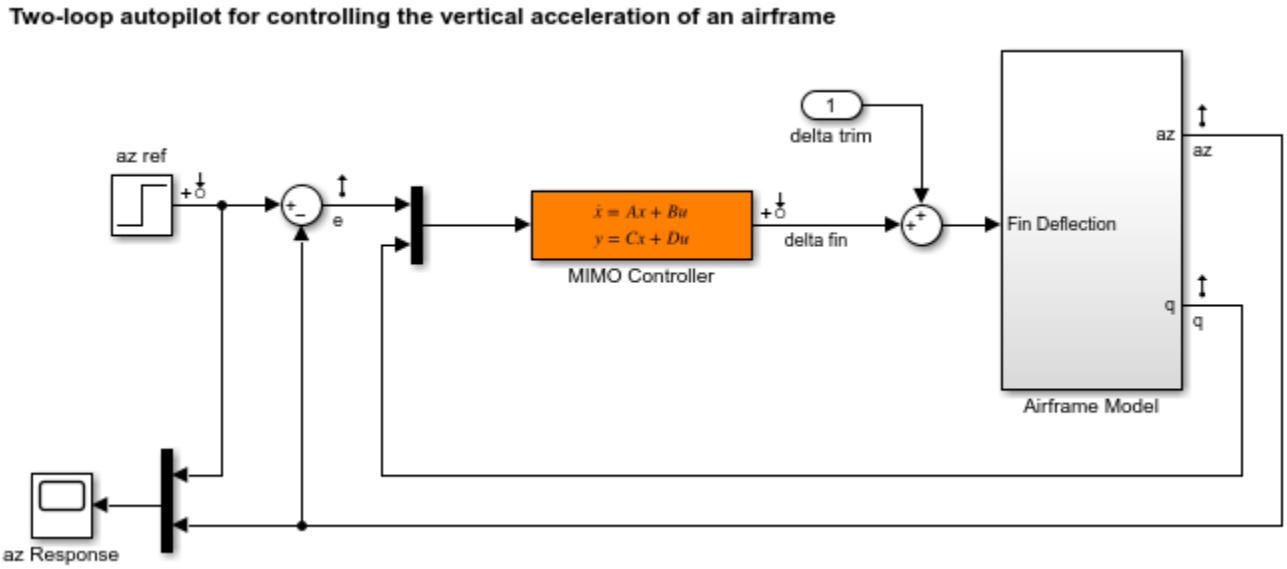
If this design is satisfactory, use `writeBlockValue` to apply the tuned values to the Simulink model and simulate the tuned controller in Simulink.

```
writeBlockValue(ST)
```

MIMO Design with SYSTUNE

Cascaded loops are commonly used for autopilots. Yet one may wonder how a single MIMO controller that uses both `az` and `q` to generate the actuator command `delta fin` would compare with the two-loop architecture. Trying new control architectures is easy with `systune` or `looptune`. For variety, we now use `systune` to tune the following MIMO architecture.

```
open_system('rct_airframe2')
```



As before, compute the trim condition for $\alpha = 0$ and $V = 984 m/s$.

```
opspec = operspec('rct_airframe2');

% Specify trim condition
% Xe,Ze: known, not steady
opspec.States(1).Known = [1;1];
opspec.States(1).SteadyState = [0;0];
% u,w: known, w steady
opspec.States(3).Known = [1 1];
opspec.States(3).SteadyState = [0 1];
% theta: known, not steady
opspec.States(2).Known = 1;
```

```
opspec.States(2).SteadyState = 0;
% q: unknown, steady
opspec.States(4).Known = 0;
opspec.States(4).SteadyState = 1;
% controller states unknown, not steady
opspec.States(5).SteadyState = [0;0];

op = findop('rct_airframe2',opspec);


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

opreport =


 Operating point search report for the Model rct_airframe2.
 (Time-Varying Components Evaluated at time t=0)

Operating point specifications were successfully met.
States:
----------
     <strong>Min</strong>         <strong>x</strong>          <strong>Max</strong>     <strong>dx
    <strong>_____</strong>    <strong>__________</strong>    <strong>_____</strong>    <strong>__

(1.) rct_airframe2/Airframe Model/Aerodynamics & Equations of Motion/ Equations of Motion (Body A
        0             0        0    -Inf             984     Inf
    -3048         -3048    -3048    -Inf               0     Inf
(2.) rct_airframe2/Airframe Model/Aerodynamics & Equations of Motion/ Equations of Motion (Body A
        0             0        0    -Inf      -0.0097235     Inf
(3.) rct_airframe2/Airframe Model/Aerodynamics & Equations of Motion/ Equations of Motion (Body A
      984           984      984    -Inf           22.69     Inf
        0             0        0       0      2.4587e-11       0
(4.) rct_airframe2/Airframe Model/Aerodynamics & Equations of Motion/ Equations of Motion (Body A
     -Inf    -0.0097235      Inf       0     -1.7215e-16       0
(5.) rct_airframe2/MIMO Controller
     -Inf    0.00065361      Inf    -Inf      -0.0089973     Inf
     -Inf    4.1313e-19      Inf    -Inf        0.030259     Inf

Inputs:
----------
    <strong>Min</strong>         <strong>u</strong>          <strong>Max</strong>
    <strong>____</strong>    <strong>__________</strong>    <strong>___</strong>

(1.) rct_airframe2/delta trim
    -Inf    0.00043574    Inf

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

As with `looptune`, use the `slTuner` interface to configure the Simulink model for tuning. Note that the signals of interest are already marked as Linear Analysis points in the Simulink model.

```
ST0 = slTuner('rct_airframe2','MIMO Controller',op);
```

Try a second-order MIMO controller with zero feedthrough from `e` to `delta fin`. To do this, create the desired controller parameterization and associate it with the "MIMO Controller" block using `setBlockParam`:

```
C0 = tunableSS('C',2,1,2);          % Second-order controller
C0.D.Value(1) = 0;                  % Fix D(1) to zero
C0.D.Free(1) = false;
setBlockParam(ST0,'MIMO Controller',C0)
```

Next create the tuning requirements. Here we use the following four requirements:

1. **Tracking**: `az` should respond in about 1 second to the `azref` command
2. **Bandwidth and roll-off**: The loop gain at `delta fin` should roll off after 25 rad/s with a -20 dB/decade slope
3. **Stability margins**: The margins at `delta fin` should exceed 7 dB and 45 degrees
4. **Disturbance rejection**: The attenuation factor for input disturbances should be 40 dB at 1 rad/s increasing to 100 dB at 0.001 rad/s.

```
% Tracking
Req1 = TuningGoal.Tracking('az ref','az',1);

% Bandwidth and roll-off
Req2 = TuningGoal.MaxLoopGain('delta fin',tf(25,[1 0]));

% Margins
Req3 = TuningGoal.Margins('delta fin',7,45);

% Disturbance rejection
% Use an FRD model to sketch the desired attenuation profile with a few points
Freqs = [0 0.001 1];
MinAtt = [100 100 40];  % in dB
Req4 = TuningGoal.Rejection('delta fin',frd(db2mag(MinAtt),Freqs));
Req4.Focus = [0 1];
```

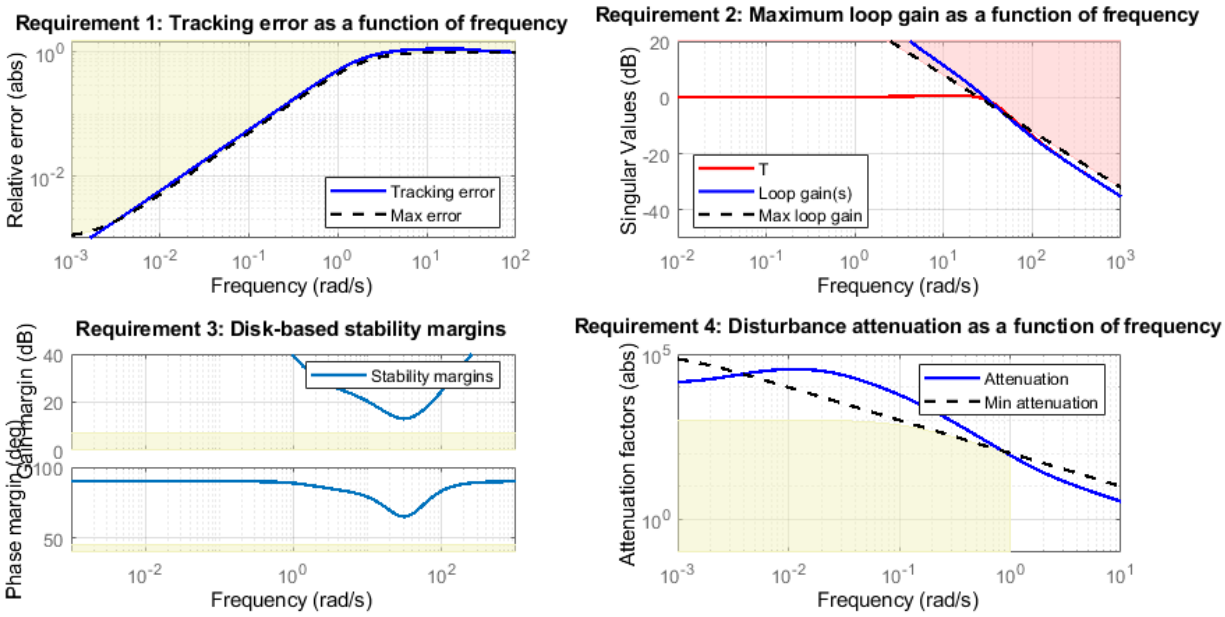
You can now use `systune` to tune the controller parameters subject to these requirements.

```
AllReqs = [Req1,Req2,Req3 Req4];
Opt = systuneOptions('RandomStart',3);

rng(0)
[ST,fSoft] = systune(ST0,AllReqs,Opt);
```

```
Final: Soft = 1.42, Hard = -Inf, Iterations = 47
Final: Soft = 1.42, Hard = -Inf, Iterations = 62
Final: Soft = 1.14, Hard = -Inf, Iterations = 84
Final: Soft = 1.14, Hard = -Inf, Iterations = 115
```

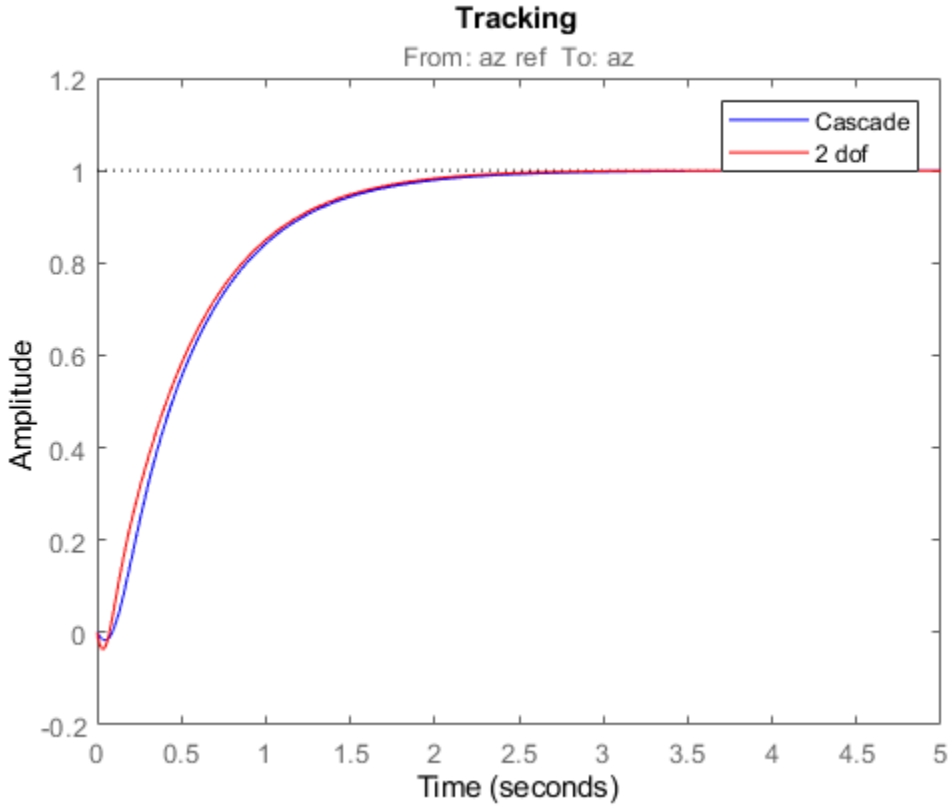
The best design has an overall objective value close to 1, indicating that all four requirements are nearly met. Use `viewGoal` to inspect each requirement for the best design.

```
figure('Position',[100,100,987,474])
viewGoal(AllReqs,ST)
```

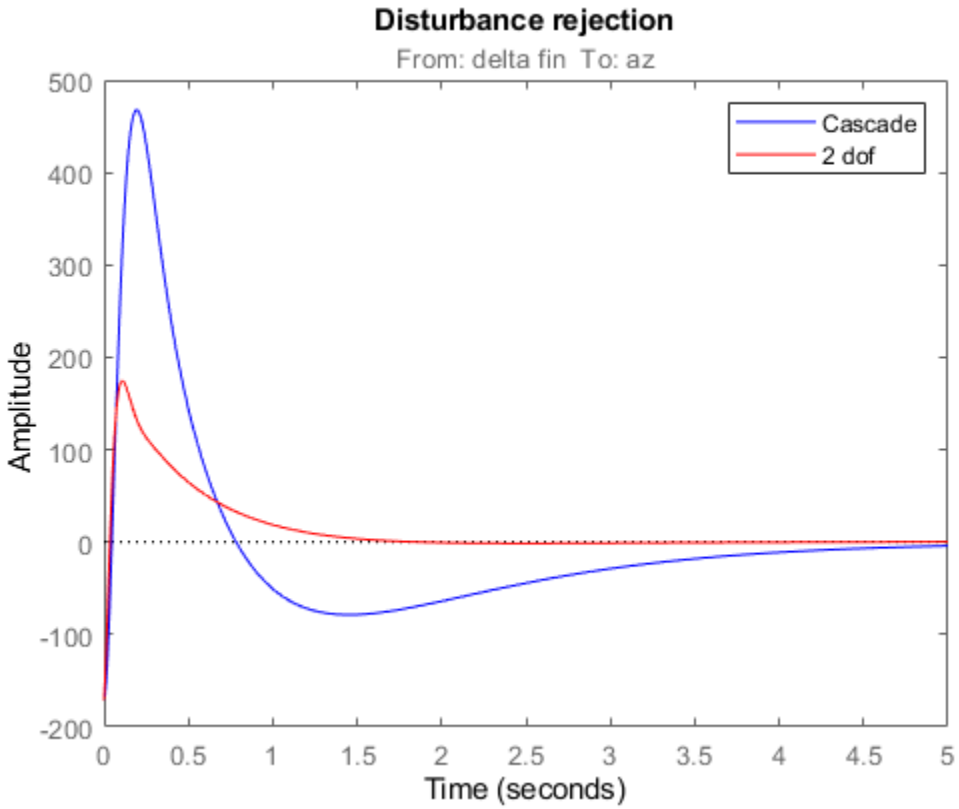


Compute the closed-loop responses and compare with the two-loop design.

```
T = getIOTransfer(ST,{'az ref','delta fin'},'az');

figure
step(Tr1,'b',T(1),'r',5)
title('Tracking')
legend('Cascade','2 dof')
```




```
step(Td1,'b',T(2),'r',5)
title('Disturbance rejection')
legend('Cascade','2 dof')
```



The tracking performance is similar but the second design has better disturbance rejection properties.

See Also

`looptune (slTuner)` | `slTuner`

Related Examples

- “Multi-Loop PI Control of a Robotic Arm”
- “Decoupling Controller for a Distillation Column”

Multiloop Control of a Helicopter

This example shows how to use `slTuner` and `systune` to tune a multiloop controller for a rotorcraft.

Helicopter Model

This example uses an 8-state helicopter model at the hovering trim condition. The state vector `x = [u,w,q,theta,v,p,phi,r]` consists of

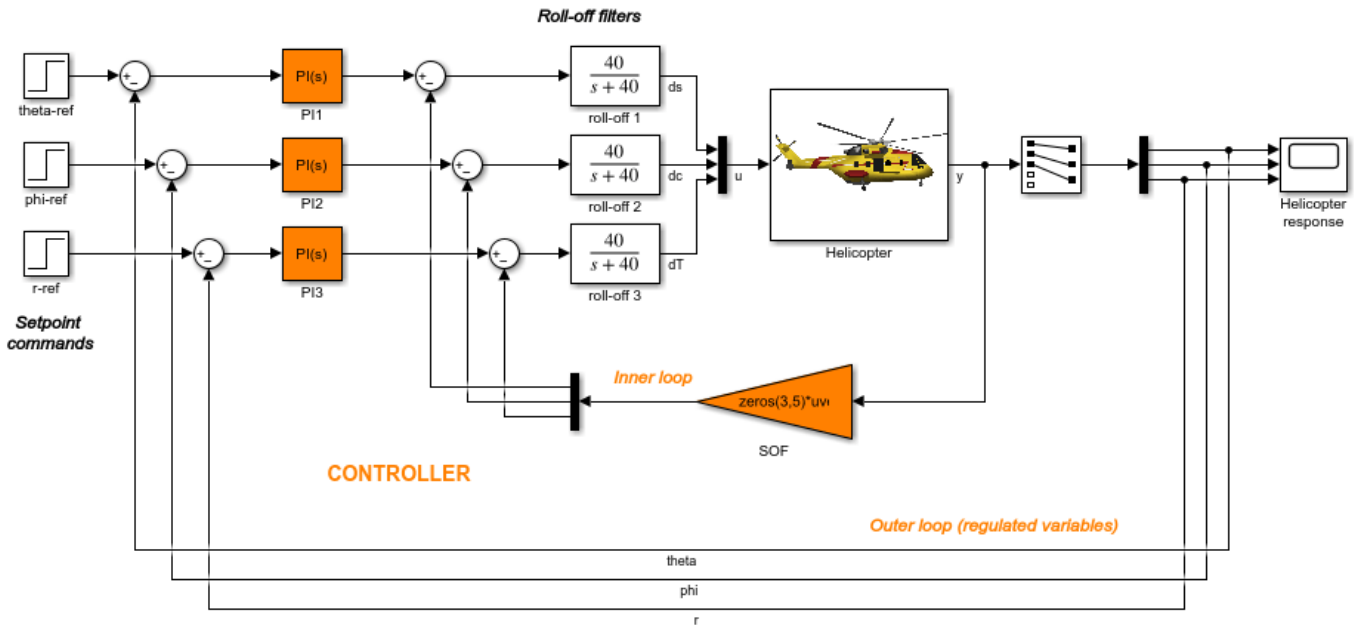
- Longitudinal velocity `u` (m/s)
- Lateral velocity `v` (m/s)
- Normal velocity `w` (m/s)
- Pitch angle `theta` (deg)
- Roll angle `phi` (deg)
- Roll rate `p` (deg/s)
- Pitch rate `q` (deg/s)
- Yaw rate `r` (deg/s).

The controller generates commands `ds,dc,dT` in degrees for the longitudinal cyclic, lateral cyclic, and tail rotor collective using measurements of `theta`, `phi`, `p`, `q`, and `r`.

Control Architecture

The following Simulink model depicts the control architecture:

```
open_system('rct_helico')
```



The control system consists of two feedback loops. The inner loop (static output feedback) provides stability augmentation and decoupling. The outer loop (PI controllers) provides the desired setpoint tracking performance. The main control objectives are as follows:

- Track setpoint changes in `theta`, `phi`, and `r` with zero steady-state error, rise times of about 2 seconds, minimal overshoot, and minimal cross-coupling
- Limit the control bandwidth to guard against neglected high-frequency rotor dynamics and measurement noise
- Provide strong multivariable gain and phase margins (robustness to simultaneous gain/phase variations at the plant inputs and outputs, see `diskmargin` for details).

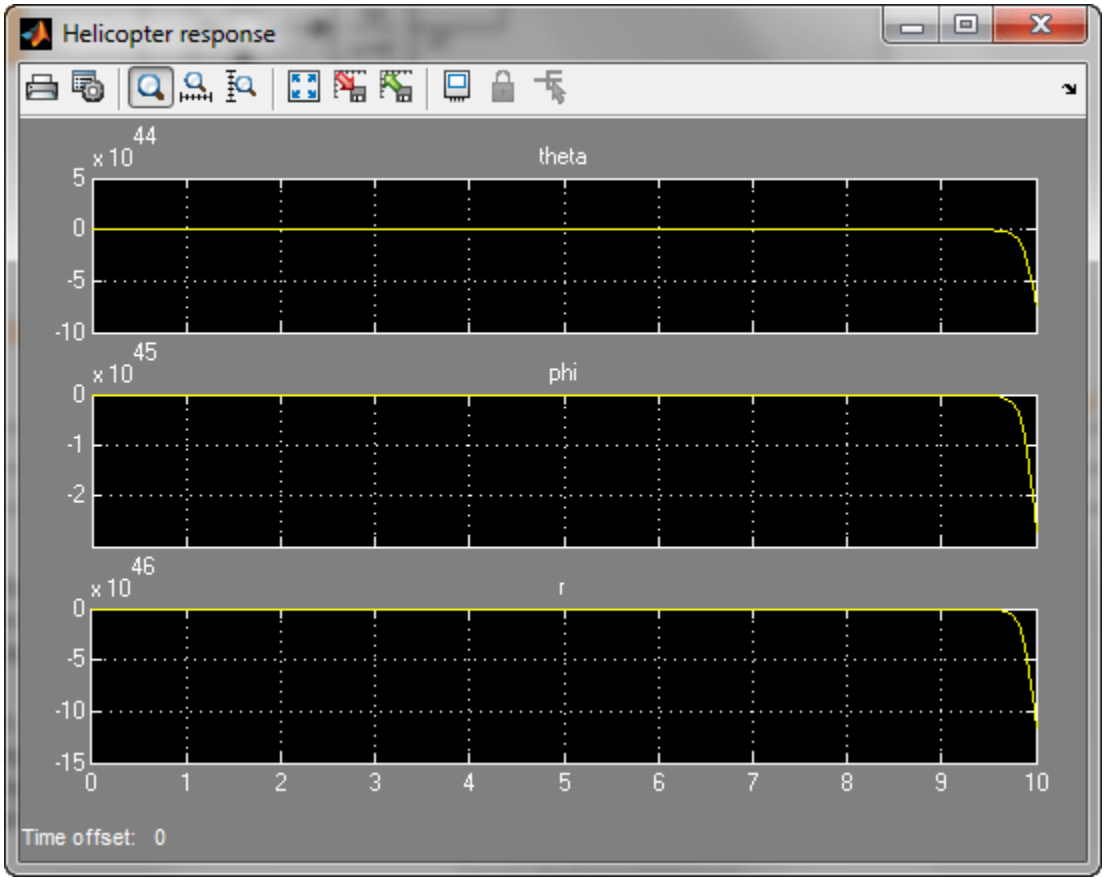
We use lowpass filters with cutoff at 40 rad/s to partially enforce the second objective.

Controller Tuning

You can jointly tune the inner and outer loops with the `systune` command. This command only requires models of the plant and controller along with the desired bandwidth (which is function of the desired response time). When the control system is modeled in Simulink, you can use the `slTuner` interface to quickly set up the tuning task. Create an instance of this interface with the list of blocks to be tuned.

```
ST0 = slTuner('rct_helico',{'PI1','PI2','PI3','SOF'});
```

Each tunable block is automatically parameterized according to its type and initialized with its value in the Simulink model ($1 + 1/s$ for the PI controllers and zero for the static output-feedback gain). Simulating the model shows that the control system is unstable for these initial values:



Mark the I/O signals of interest for setpoint tracking, and identify the plant inputs and outputs (control and measurement signals) where the stability margin are measured.

```
addPoint(ST0,{'theta-ref','phi-ref','r-ref'})   % setpoint commands
addPoint(ST0,{'theta','phi','r'})               % corresponding outputs
addPoint(ST0,{'u','y'});
```

Finally, capture the design requirements using `TuningGoal` objects. We use the following requirements for this example:

- **Tracking requirement**: The response of `theta`, `phi`, `r` to step commands `theta_ref`, `phi_ref`, `r_ref` must resemble a decoupled first-order response with a one-second time constant
- **Stability margins**: The multivariable gain and phase margins at the plant inputs `u` and plant outputs `y` must be at least 5 dB and 40 degrees
- **Fast dynamics**: The magnitude of the closed-loop poles must not exceed 25 to prevent fast dynamics and jerky transients

```
% Less than 20% mismatch with reference model 1/(s+1)
TrackReq = TuningGoal.StepTracking({'theta-ref','phi-ref','r-ref'},{'theta','phi','r'},1);
TrackReq.RelGap = 0.2;

% Gain and phase margins at plant inputs and outputs
MarginReq1 = TuningGoal.Margins('u',5,40);
MarginReq2 = TuningGoal.Margins('y',5,40);

% Limit on fast dynamics
MaxFrequency = 25;
PoleReq = TuningGoal.Poles(0,0,MaxFrequency);
```

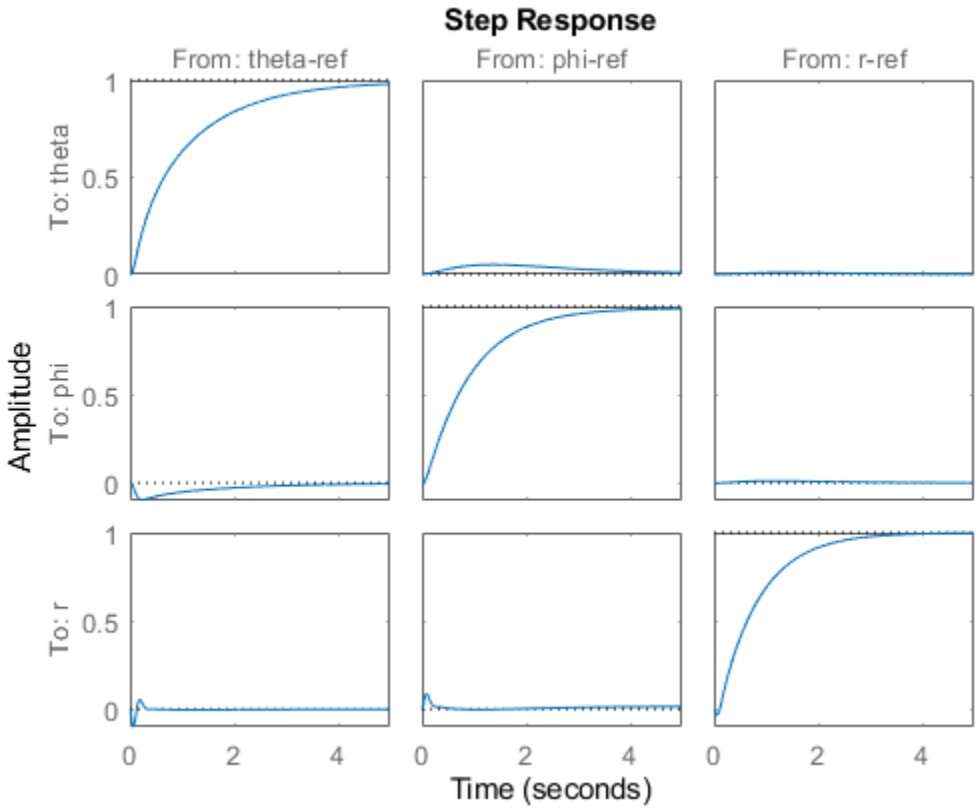
You can now use `systune` to jointly tune all controller parameters. This returns the tuned version `ST1` of the control system `ST0`.

```
AllReqs = [TrackReq,MarginReq1,MarginReq2,PoleReq];
ST1 = systune(ST0,AllReqs);
```

```
Final: Soft = 1.12, Hard = -Inf, Iterations = 71
```

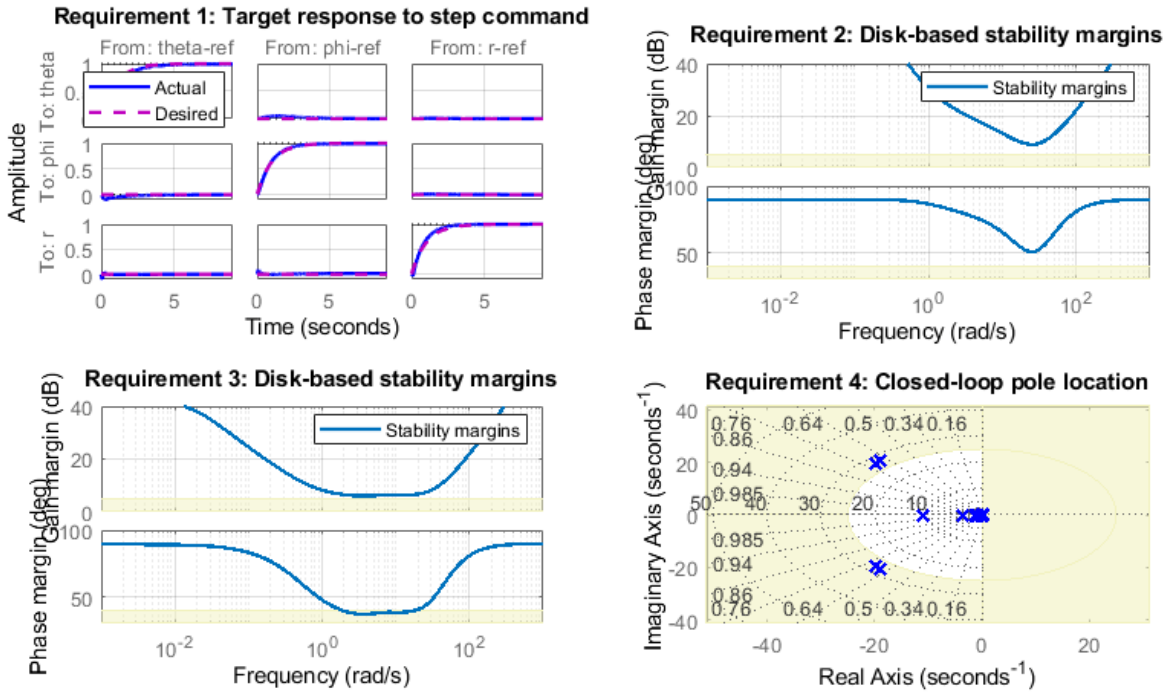
The final value is close to 1 so the requirements are nearly met. Plot the tuned responses to step commands in theta, phi, r:

```
T1 = getIOTransfer(ST1,{'theta-ref','phi-ref','r-ref'},{'theta','phi','r'});
step(T1,5)
```



The rise time is about two seconds with no overshoot and little cross-coupling. You can use `viewGoal` for a more thorough validation of each requirement, including a visual assessment of the multivariable stability margins (see `diskmargin` for details):

```
figure('Position',[100,100,900,474])
viewGoal(AllReqs,ST1)
```



Inspect the tuned values of the PI controllers and static output-feedback gain.

```
showTunable(ST1)
```

```
Block 1: rct_helico/PI1 =

             1
  Kp + Ki * ---
             s

  with Kp = 1.04, Ki = 2.08

Name: PI1
Continuous-time PI controller in parallel form.

-----------------------------------

Block 2: rct_helico/PI2 =

             1
  Kp + Ki * ---
             s

  with Kp = -0.0981, Ki = -1.35

Name: PI2
Continuous-time PI controller in parallel form.

-----------------------------------

Block 3: rct_helico/PI3 =
```

```
             1
  Kp + Ki * ---
             s

  with Kp = 0.14, Ki = -2.2

Name: PI3
Continuous-time PI controller in parallel form.

-----------------------------------

Block 4: rct_helico/SOF =

  D =
              u1         u2         u3         u4         u5
   y1      2.217     -0.309  -0.003329     0.7856   -0.01513
   y2     -0.192     -1.286    0.01831   -0.08421    -0.1192
   y3   -0.02129   -0.01223     -1.898   -0.00144    0.06835

Name: SOF
Static gain.
```

Benefit of the Inner Loop

You may wonder whether the static output feedback is necessary and whether PID controllers aren't enough to control the helicopter. This question is easily answered by re-tuning the controller with the inner loop open. First break the inner loop by adding a loop opening after the SOF block:

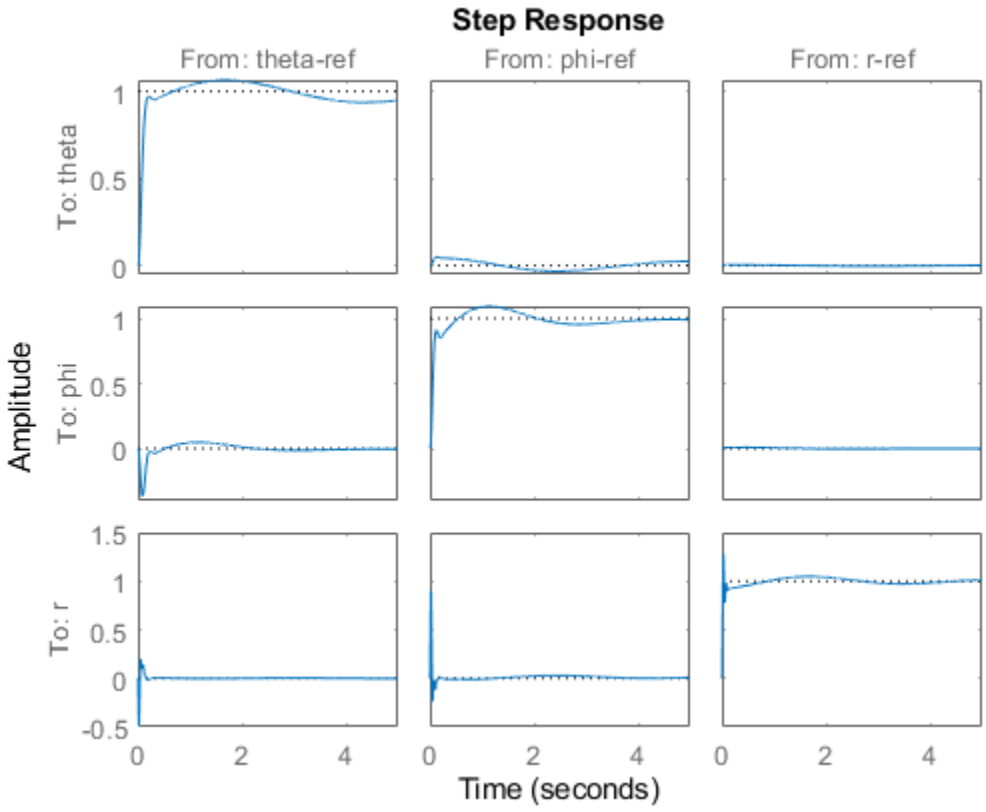
```
addOpening(ST0,'SOF')
```

Then remove the SOF block from the tunable block list and re-parameterize the PI blocks as full-blown PIDs with the correct loop signs (as inferred from the first design).

```
PID = pid(0,0.001,0.001,.01);  % initial guess for PID controllers

removeBlock(ST0,'SOF');
setBlockParam(ST0,...
   'PI1',tunablePID('C1',PID),...
   'PI2',tunablePID('C2',-PID),...
   'PI3',tunablePID('C3',-PID));
```

Re-tune the three PID controllers and plot the closed-loop step responses.

```
ST2 = systune(ST0,AllReqs);

Final: Soft = 4.94, Hard = -Inf, Iterations = 67

T2 = getIOTransfer(ST2,{'theta-ref','phi-ref','r-ref'},{'theta','phi','r'});
figure, step(T2,5)
```



The final value is no longer close to 1 and the step responses confirm the poorer performance with regard to rise time, overshoot, and decoupling. This suggests that the inner loop has an important stabilizing effect that should be preserved.

See Also

`systune (slTuner)` | `slTuner` | `TuningGoal.StepTracking` | `TuningGoal.Margins` | `TuningGoal.Poles`

Related Examples

- “Fixed-Structure Autopilot for a Passenger Jet”

Fixed-Structure Autopilot for a Passenger Jet

This example shows how to use `slTuner` and `systune` to tune the standard configuration of a longitudinal autopilot. We thank Professor D. Alazard from Institut Superieur de l'Aeronautique et de l'Espace for providing the aircraft model and Professor Pierre Apkarian from ONERA for developing the example.

Aircraft Model and Autopilot Configuration

The longitudinal autopilot for a supersonic passenger jet flying at Mach 0.7 and 5000 ft is depicted in Figure 1. The autopilot main purpose is to follow vertical acceleration commands N_{zc} issued by the pilot. The feedback structure consists of an inner loop controlling the pitch rate q and an outer loop controlling the vertical acceleration N_z. The autopilot also includes a feedforward component and a reference model $G_{ref}(s)$ that specifies the desired response to a step command N_{zc}. Finally, the second-order roll-off filter

$$F_{ro}(s) = \frac{\omega_n^2}{s^2 + 2\zeta\omega_n s + \omega_n^2}$$

is used to attenuate noise and limit the control bandwidth as a safeguard against unmodeled dynamics. The tunable components are highlighted in orange.

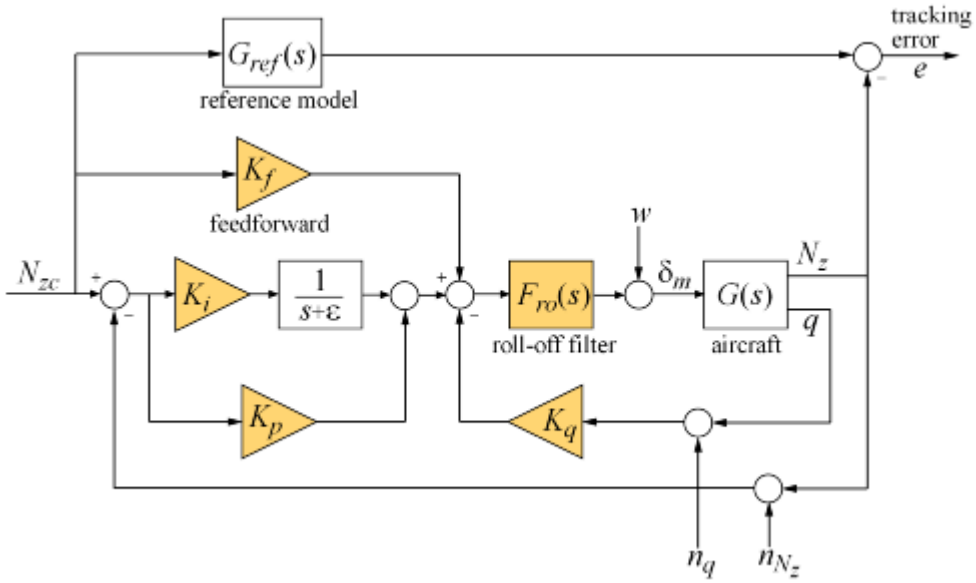
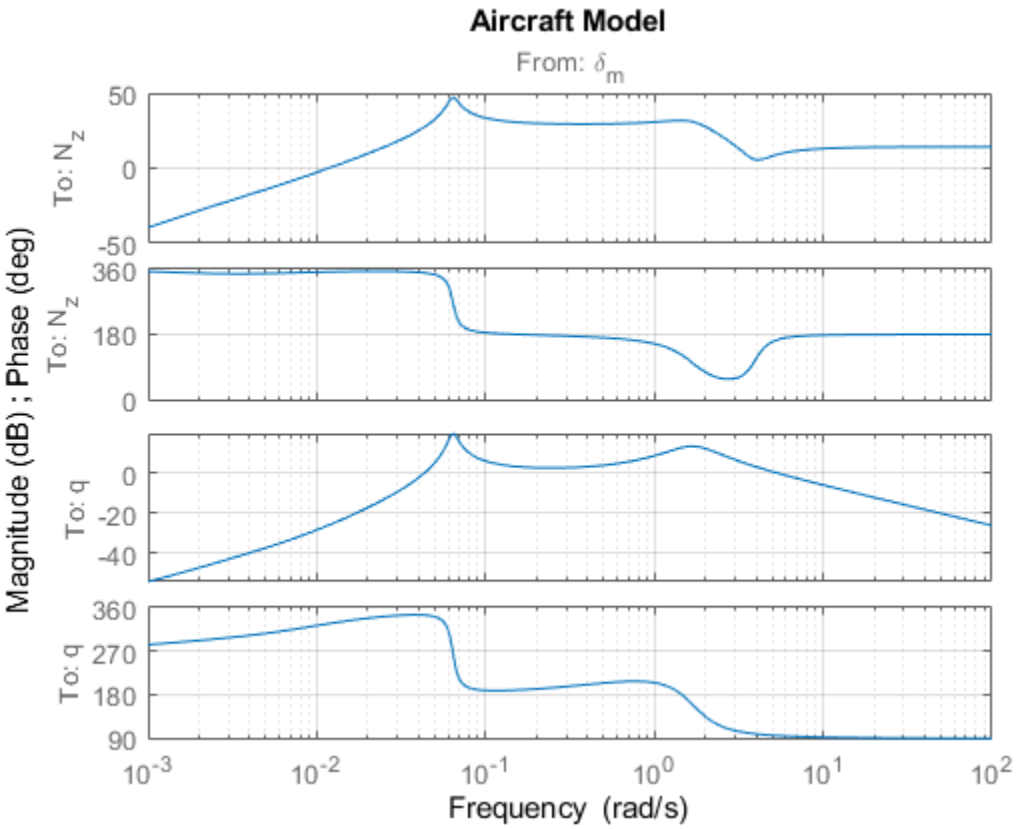


Figure 1: Longitudinal Autopilot Configuration.

The aircraft model $G(s)$ is a 5-state model, the state variables being the aerodynamic speed V (m/s), the climb angle γ (rad), the angle of attack α (rad), the pitch rate q (rad/s), and the altitude H (m). The elevator deflection δ_m (rad) is used to control the vertical load factor N_z. The open-loop dynamics include the α oscillation with frequency and damping ratio ω_n = 1.7 (rad/s) and ζ = 0.33, the phugoid mode ω_n = 0.64 (rad/s) and ζ = 0.06, and the slow altitude mode λ = -0.0026.

```
load ConcordeData G
bode(G,{1e-3,1e2}), grid
title('Aircraft Model')
```

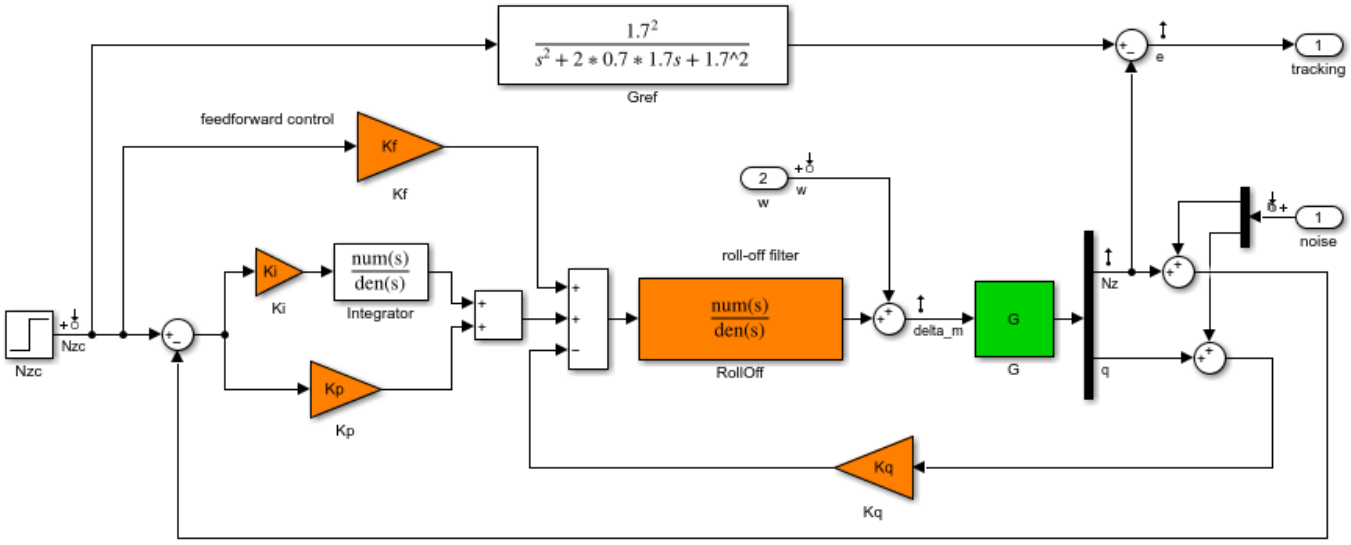


Note the zero at the origin in $G(s)$. Because of this zero, we cannot achieve zero steady-state error and must instead focus on the transient response to acceleration commands. Note that acceleration commands are transient in nature so steady-state behavior is not a concern. This zero at the origin also precludes pure integral action so we use a pseudo-integrator $1/(s+\epsilon)$ with ϵ = 0.001.

Tuning Setup

When the control system is modeled in Simulink, you can use the `slTuner` interface to quickly set up the tuning task. Open the Simulink model of the autopilot.

```
open_system('rct_concorde')
```



Longitudinal autopilot for a passenger jet.

You can tune this autopilot with the SYSTUNE command, see concorde_demo for details

Copyright 2004-2012 The MathWorks, Inc.

Configure the `slTuner` interface by listing the tuned blocks in the Simulink model (highlighted in orange). This automatically picks all Linear Analysis points in the model as points of interest for analysis and tuning.

```
ST0 = slTuner('rct_concorde',{'Ki','Kp','Kq','Kf','RollOff'});
```

This also parameterizes each tuned block and initializes the block parameters based on their values in the Simulink model. Note that the four gains `Ki,Kp,Kq,Kf` are initialized to zero in this example.

By default the roll-off filter $F_{ro}(s)$ is parameterized as a generic second-order transfer function. To parameterize it as

$$F_{ro}(s) = \frac{\omega_n^2}{s^2 + 2\zeta\omega_n s + \omega_n^2},$$

create real parameters ζ, ω_n, build the transfer function shown above, and associate it with the `RollOff` block.

```
wn = realp('wn', 3);                % natural frequency
zeta = realp('zeta',0.8);           % damping
Fro = tf(wn^2,[1 2*zeta*wn wn^2]); % parametric transfer function

setBlockParam(ST0,'RollOff',Fro)   % use Fro to parameterize "RollOff" block
```

Design Requirements

The autopilot must be tuned to satisfy three main design requirements:

1. **Setpoint tracking**: The response N_z to the command N_{zc} should closely match the response of the reference model:

$$G_{ref}(s) = \frac{1.7^2}{s^2 + 2 \times 0.7 \times 1.7s + 1.7^2}.$$

This reference model specifies a well-damped response with a 2 second settling time.

2. **High-frequency roll-off**: The closed-loop response from the noise signals to δ_m should roll off past 8 rad/s with a slope of at least -40 dB/decade.

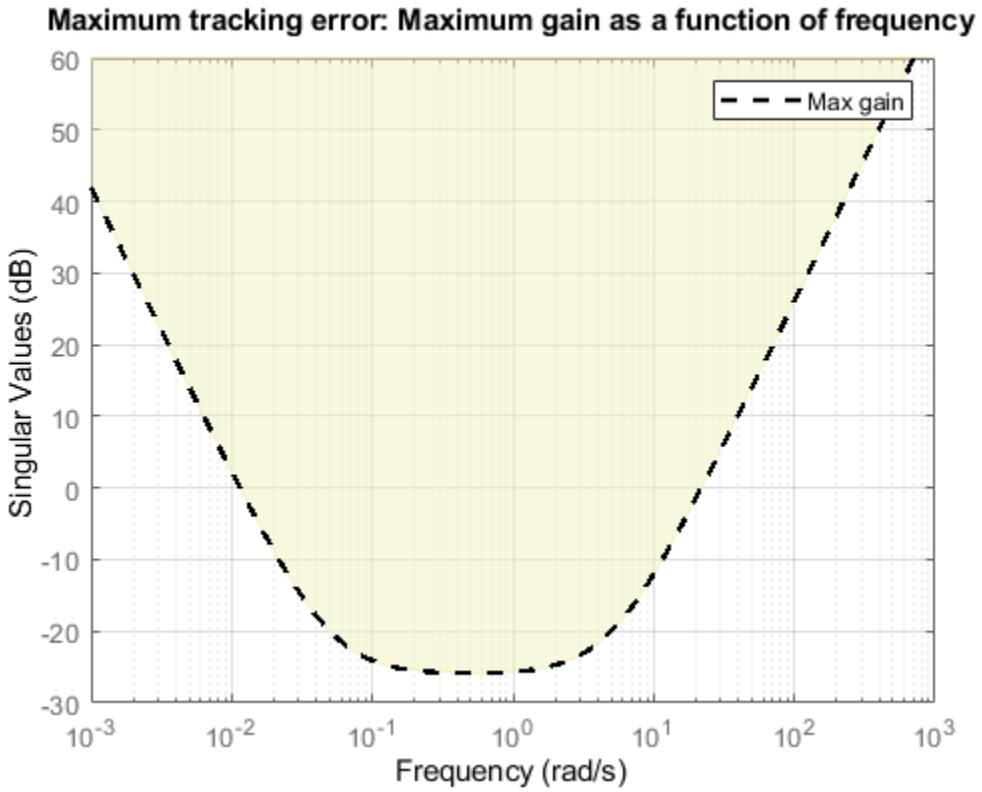
3. **Stability margins**: The stability margins at the plant input δ_m should be at least 7 dB and 45 degrees.

For setpoint tracking, we require that the gain of the closed-loop transfer from the command N_{zc} to the tracking error e be small in the frequency band [0.05,5] rad/s (recall that we cannot drive the steady-state error to zero because of the plant zero at s=0). Using a few frequency points, sketch the maximum tracking error as a function of frequency and use it to limit the gain from N_{zc} to e.

```
Freqs = [0.005 0.05 5 50];
Gains = [5 0.05 0.05 5];
Req1 = TuningGoal.Gain('Nzc','e',frd(Gains,Freqs));
Req1.Name = 'Maximum tracking error';
```

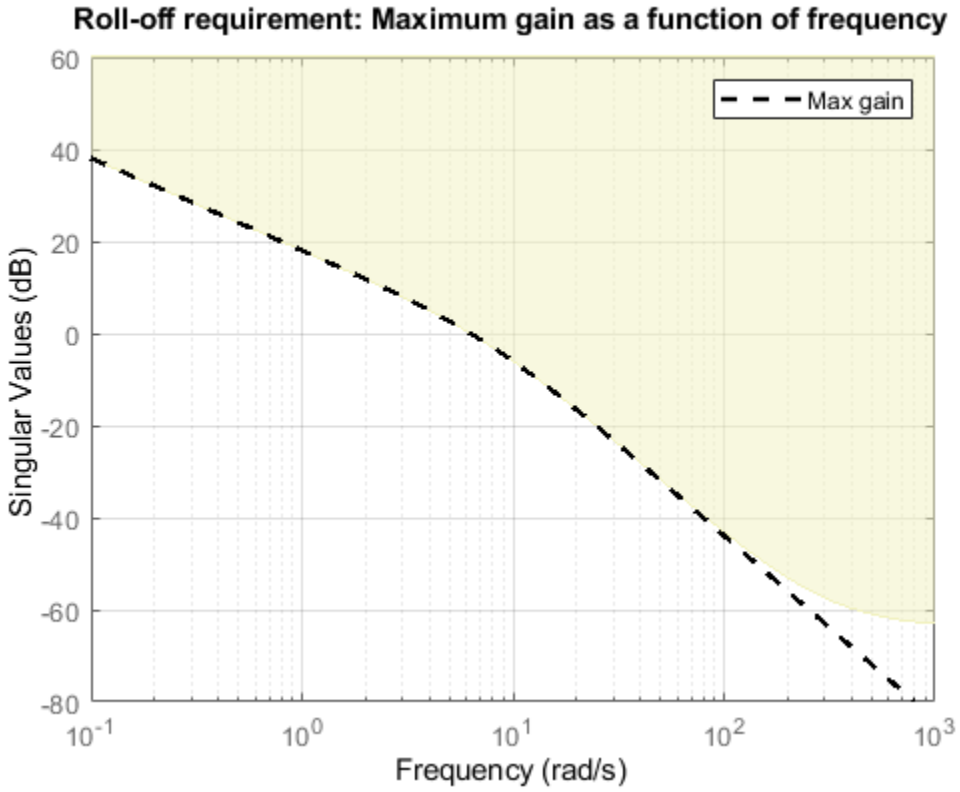
The `TuningGoal.Gain` constructor automatically turns the maximum error sketch into a smooth weighting function. Use `viewGoal` to graphically verify the desired error profile.

```
viewGoal(Req1)
```



Repeat the same process to limit the high-frequency gain from the noise inputs to δ_m and enforce a -40 dB/decade slope in the frequency band from 8 to 800 rad/s

```
Freqs = [0.8 8 800];
Gains = [10 1 1e-4];
Req2 = TuningGoal.Gain('n','delta_m',frd(Gains,Freqs));
Req2.Name = 'Roll-off requirement';

viewGoal(Req2)
```



Finally, register the plant input δ_m as a site for open-loop analysis and use `TuningGoal.Margins` to capture the stability margin requirement.

```
addPoint(ST0,'delta_m')

Req3 = TuningGoal.Margins('delta_m',7,45);
```

Autopilot Tuning

We are now ready to tune the autopilot parameters with `systune`. This command takes the untuned configuration `ST0` and the three design requirements and returns the tuned version `ST` of `ST0`. All requirements are satisfied when the final value is less than one.

```
[ST,fSoft] = systune(ST0,[Req1 Req2 Req3]);
```

```
Final: Soft = 0.965, Hard = -Inf, Iterations = 80
```

Use `showTunable` to see the tuned block values.

```
showTunable(ST)
```

```
Block 1: rct_concorde/Ki =

  D =
         u1
   y1  -0.03

Name: Ki
Static gain.

-----------------------------------

Block 2: rct_concorde/Kp =

  D =
             u1
   y1  -0.009673

Name: Kp
Static gain.

-----------------------------------

Block 3: rct_concorde/Kq =

  D =
           u1
   y1  -0.2869

Name: Kq
Static gain.

-----------------------------------

Block 4: rct_concorde/Kf =

  D =
            u1
   y1  -0.02274

Name: Kf
Static gain.

-----------------------------------

wn = 4.82
-----------------------------------
zeta = 0.514
```

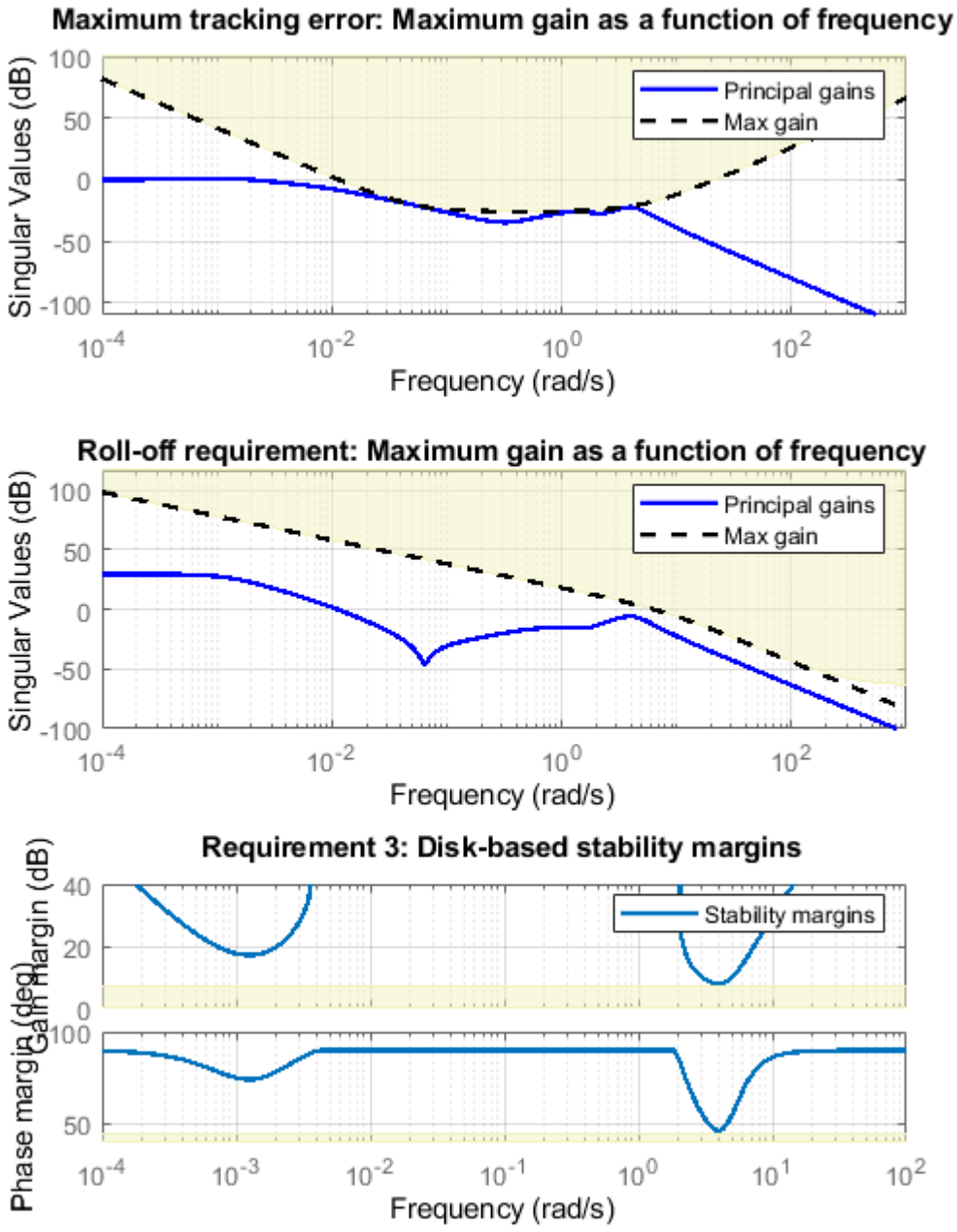
To get the tuned value of $F_{ro}(s)$, use `getBlockValue` to evaluate `Fro` for the tuned parameter values in `ST`:

```
Fro = getBlockValue(ST,'RollOff');
tf(Fro)
```

```
ans =

          23.22
  ---------------------
  s^2 + 4.952 s + 23.22

Continuous-time transfer function.
```

Finally, use `viewGoal` to graphically verify that all requirements are satisfied.

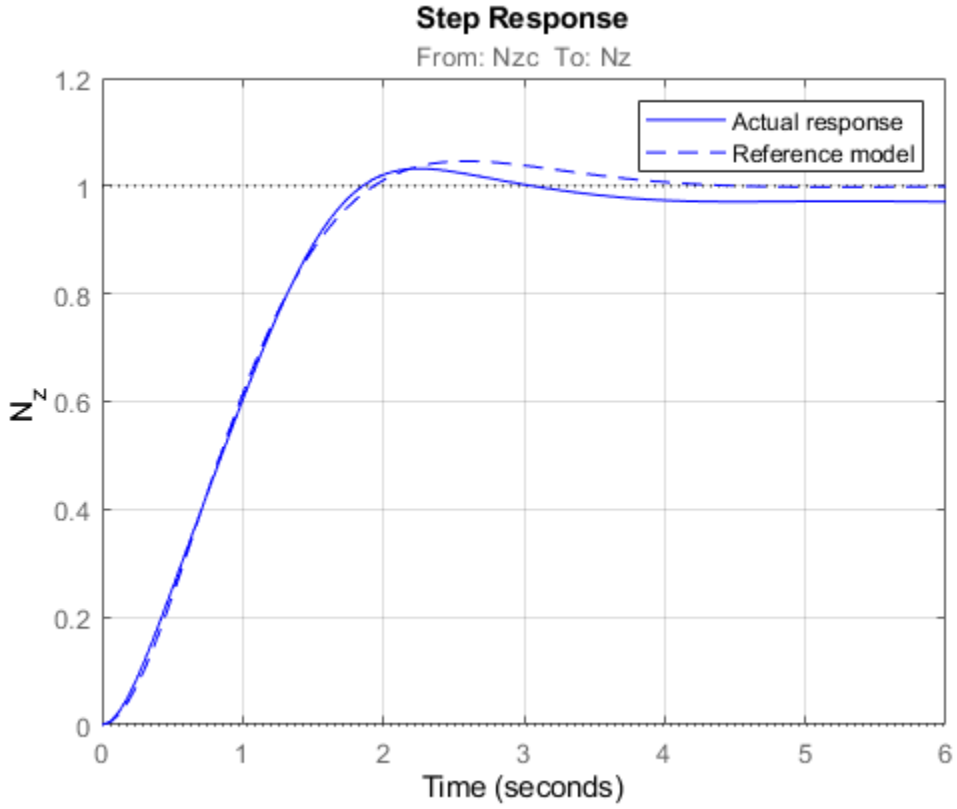
```
figure('Position',[100,100,550,710])
viewGoal([Req1 Req2 Req3],ST)
```



Closed-Loop Simulations

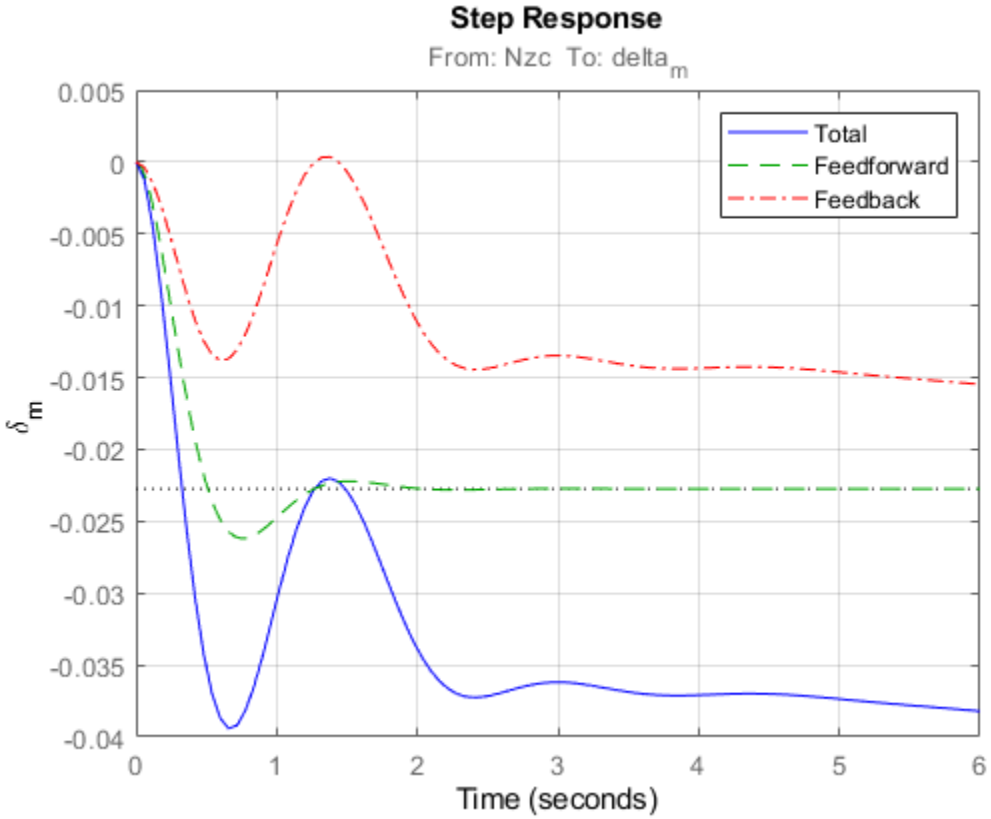
We now verify that the tuned autopilot satisfies the design requirements. First compare the step response of N_z with the step response of the reference model $G_{ref}(s)$. Again use `getIOTransfer` to compute the tuned closed-loop transfer from `Nzc` to `Nz`:

```
Gref = tf(1.7^2,[1 2*0.7*1.7 1.7^2]);    % reference model

T = getIOTransfer(ST,'Nzc','Nz');  % transfer Nzc -> Nz

figure, step(T,'b',Gref,'b--',6), grid,
ylabel('N_z'), legend('Actual response','Reference model')
```



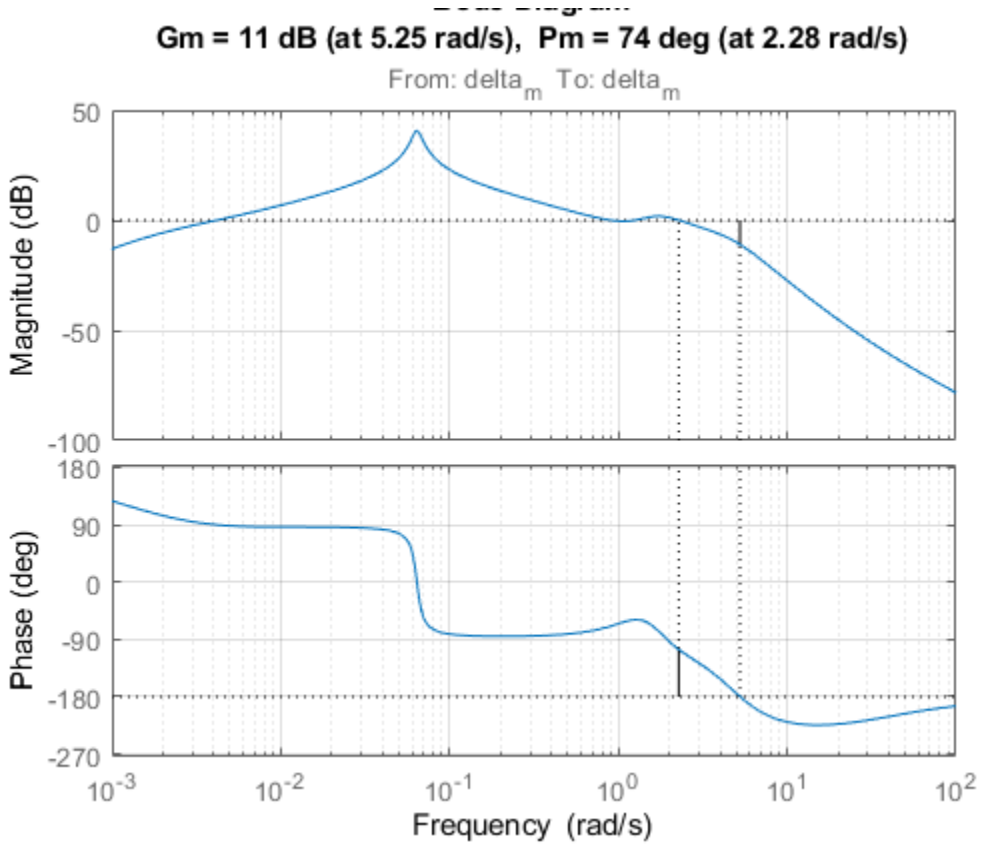
Also plot the deflection δ_m and the respective contributions of the feedforward and feedback paths:

```
T = getIOTransfer(ST,'Nzc','delta_m');  % transfer Nzc -> delta_m
Kf = getBlockValue(ST,'Kf');            % tuned value of Kf
Tff = Fro*Kf;         % feedforward contribution to delta_m

step(T,'b',Tff,'g--',T-Tff,'r-.',6), grid
ylabel('\delta_m'), legend('Total','Feedforward','Feedback')
```



Finally, check the roll-off and stability margin requirements by computing the open-loop response at δ_m.

```
OL = getLoopTransfer(ST,'delta_m',-1); % negative-feedback loop transfer
margin(OL);
grid;
xlim([1e-3,1e2]);
```



The Bode plot confirms a roll-off of -40 dB/decade past 8 rad/s and indicates gain and phase margins in excess of 10 dB and 70 degrees.

See Also

`systune (slTuner)` | `slTuner` | `TuningGoal.Gain` | `TuningGoal.Margins`

Related Examples

- “Fault-Tolerant Control of a Passenger Jet”

Fault-Tolerant Control of a Passenger Jet

This example shows how to tune a fixed-structure controller for multiple operating modes of the plant.

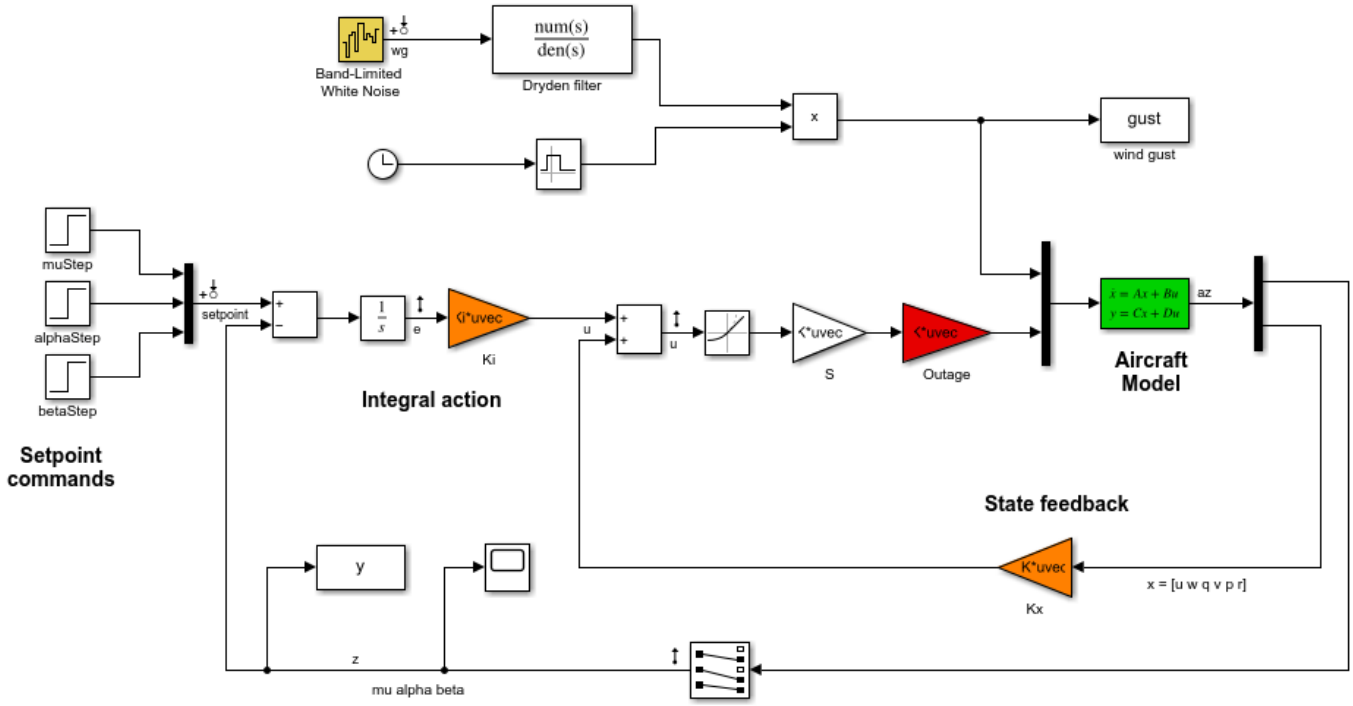
Background

This example deals with fault-tolerant flight control of passenger jet undergoing outages in the elevator and aileron actuators. The flight control system must maintain stability and meet performance and comfort requirements in both nominal operation and degraded conditions where some actuators are no longer effective due to control surface impairment. Wind gusts must be alleviated in all conditions. This application is sometimes called *reliable control* as aircraft safety must be maintained in extreme flight conditions.

Aircraft Model

The control system is modeled in Simulink.

```
addpath(fullfile(matlabroot,'examples','control','main')) % add example data
open_system('faultTolerantAircraft')
```



The aircraft is modeled as a rigid 6th-order state-space system with the following state variables (units are mph for velocities and deg/s for angular rates):

- u: x-body axis velocity
- w: z-body axis velocity
- q: pitch rate
- v: y-body axis velocity

- p: roll rate
- r: yaw rate

The state vector is available for control as well as the flight-path bank angle rate mu (deg/s), the angle of attack alpha (deg), and the sideslip angle beta (deg). The control inputs are the deflections of the right elevator, left elevator, right aileron, left aileron, and rudder. All deflections are in degrees. Elevators are grouped symmetrically to generate the angle of attack. Ailerons are grouped anti-symmetrically to generate roll motion. This leads to 3 control actions as shown in the Simulink model.

The controller consists of state-feedback control in the inner loop and MIMO integral action in the outer loop. The gain matrices `Ki` and `Kx` are 3-by-3 and 3-by-6, respectively, so the controller has 27 tunable parameters.

Actuator Failures

We use a 9x5 matrix to encode the nominal mode and various actuator failure modes. Each row corresponds to one flight condition, a zero indicating outage of the corresponding deflection surface.

```
OutageCases = [...
   1 1 1 1 1; ... % nominal operational mode
   0 1 1 1 1; ... % right elevator outage
   1 0 1 1 1; ... % left elevator outage
   1 1 0 1 1; ... % right aileron outage
   1 1 1 0 1; ... % left aileron outage
   1 0 0 1 1; ... % left elevator and right aileron outage
   0 1 0 1 1; ... % right elevator and right aileron outage
   0 1 1 0 1; ... % right elevator and left aileron outage
   1 0 1 0 1; ... % left elevator and left aileron outage
   ];
```

Design Requirements

The controller should:

1. Provide good tracking performance in mu, alpha, and beta in nominal operating mode with adequate decoupling of the three axes
2. Maintain performance in the presence of wind gust of 10 mph
3. Limit stability and performance degradation in the face of actuator outage.

To express the first requirement, you can use an LQG-like cost function that penalizes the integrated tracking error `e` and the control effort `u`:

$$J = lim_{T\to\infty} E\left(\frac{1}{T}\int_0^T \|W_e e\|^2 + \|W_u u\|^2 dt\right).$$

The diagonal weights W_e and W_u are the main tuning knobs for trading responsiveness and control effort and emphasizing some channels over others. Use the `WeightedVariance` requirement to express this cost function, and relax the performance weight W_e by a factor 2 for the outage scenarios.

```
We = diag([10 20 15]);   Wu = eye(3);

% Nominal tracking requirement
SoftNom = TuningGoal.WeightedVariance('setpoint',{'e','u'}, blkdiag(We,Wu), []);
```

```
SoftNom.Models = 1;    % nominal model

% Tracking requirement for outage conditions
SoftOut = TuningGoal.WeightedVariance('setpoint',{'e','u'}, blkdiag(We/2,Wu), []);
SoftOut.Models = 2:9;  % outage scenarios
```

For wind gust alleviation, limit the variance of the error signal `e` due to the white noise `wg` driving the wind gust model. Again use a less stringent requirement for the outage scenarios.

```
% Nominal gust alleviation requirement
HardNom = TuningGoal.Variance('wg','e',0.02);
HardNom.Models = 1;

% Gust alleviation requirement for outage conditions
HardOut = TuningGoal.Variance('wg','e',0.1);
HardOut.Models = 2:9;
```

Controller Tuning for Nominal Flight

Set the wind gust speed to 10 mph and initialize the tunable state-feedback and integrators gains of the controller.

```
GustSpeed = 10;
Ki = eye(3);
Kx = zeros(3,6);
```

Use the `slTuner` interface to set up the tuning task. List the blocks to be tuned and specify the nine flight conditions by varying the `outage` variable in the Simulink model. Because you can only vary scalar parameters in `slTuner`, independently specify the values taken by each entry of the `outage` vector.

```
OutageData = struct(...
   'Name',{'outage(1)','outage(2)','outage(3)','outage(4)','outage(5)'},...
   'Value',mat2cell(OutageCases,9,[1 1 1 1 1]));
ST0 = slTuner('faultTolerantAircraft',{'Ki','Kx'},OutageData);
```

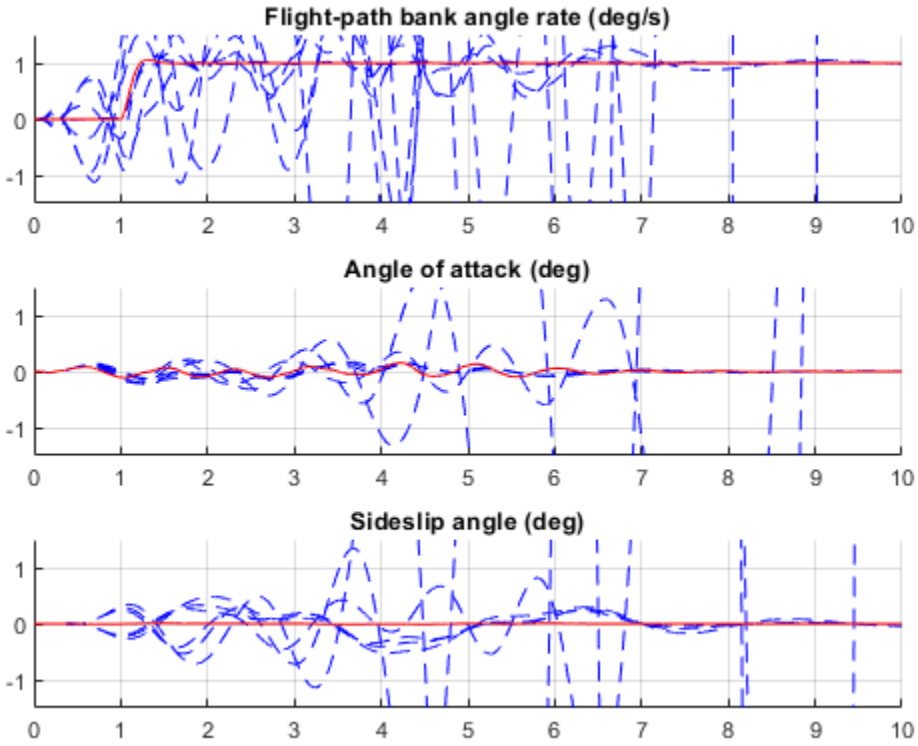
Use `systune` to tune the controller gains subject to the nominal requirements. Treat the wind gust alleviation as a hard constraint.

```
[ST,fSoft,gHard]  = systune(ST0,SoftNom,HardNom);
```

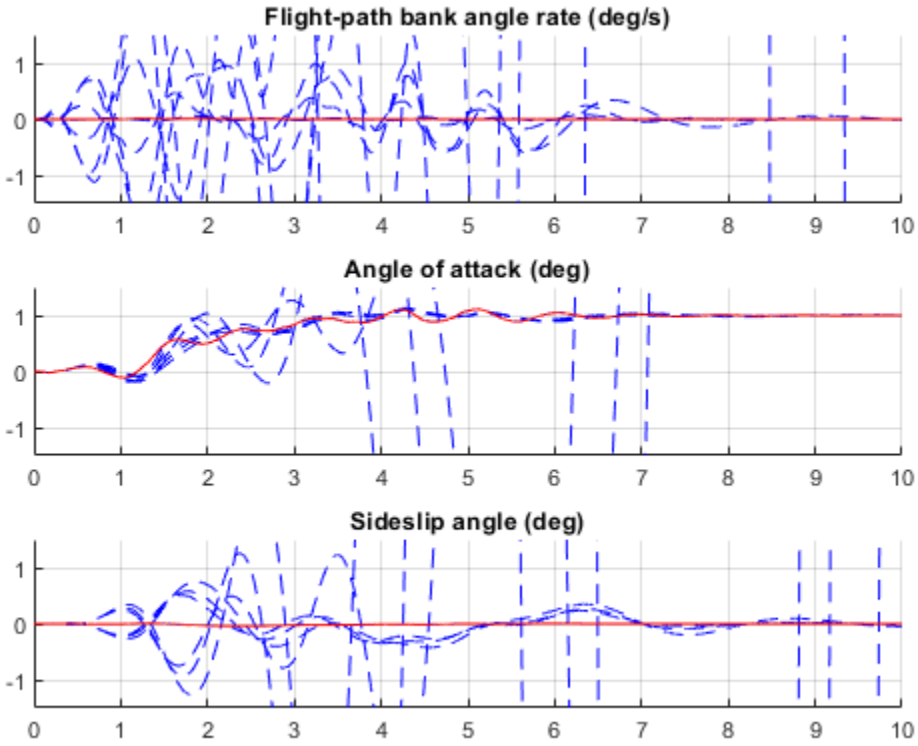
```
Final: Soft = 22.6, Hard = 0.99989, Iterations = 296
```

Retrieve the gain values and simulate the responses to step commands in `mu, alpha, beta` for the nominal and degraded flight conditions. All simulations include wind gust effects, and the red curve is the nominal response.

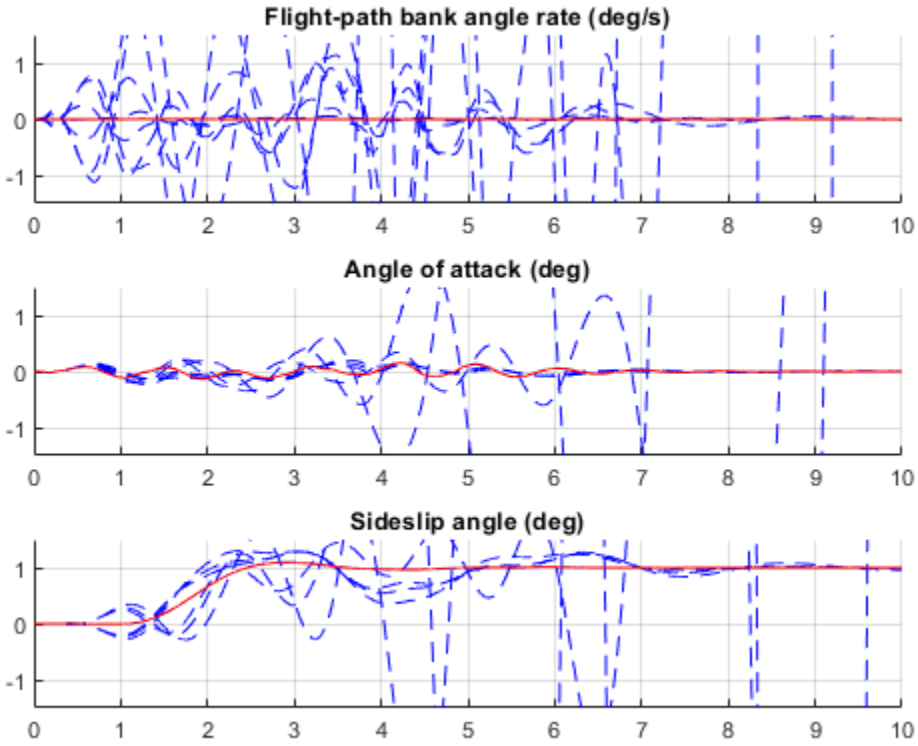
```
Ki = getBlockValue(ST, 'Ki');  Ki = Ki.d;
Kx = getBlockValue(ST, 'Kx');  Kx = Kx.d;

% Bank-angle setpoint simulation
plotResponses(OutageCases,1,0,0);
```




```
% Angle-of-attack setpoint simulation
plotResponses(OutageCases,0,1,0);
```




```
% Sideslip-angle setpoint simulation
plotResponses(OutageCases,0,0,1);
```



The nominal responses are good but the deterioration in performance is unacceptable when faced with actuator outage.

Controller Tuning for Impaired Flight

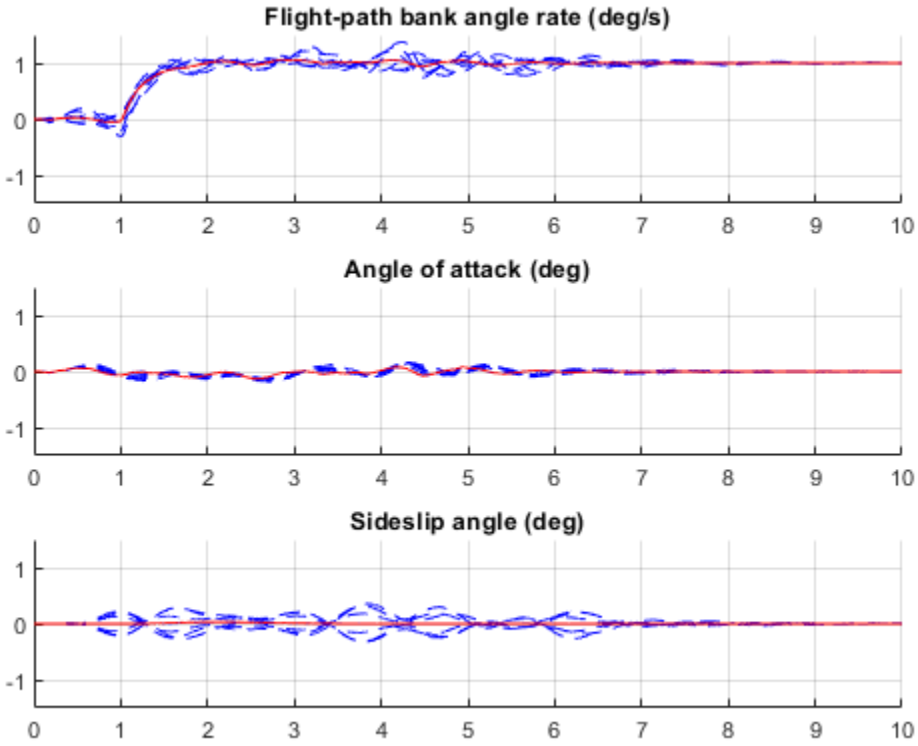
To improve reliability, retune the controller gains to meet the nominal requirement for the nominal plant as well as the relaxed requirements for all eight outage scenarios.

```
[ST,fSoft,gHard]  = systune(ST0,[SoftNom;SoftOut],[HardNom;HardOut]);
```

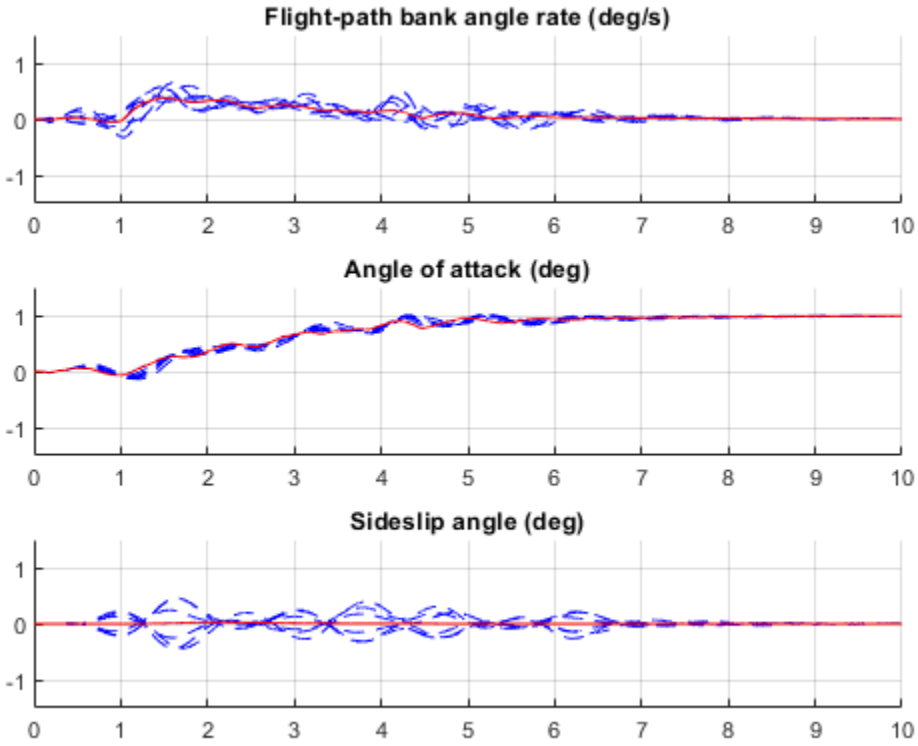
```
Final: Soft = 25.7, Hard = 0.99966, Iterations = 489
```

The optimal performance (square root of LQG cost J) is only slightly worse than for the nominal tuning (26 vs. 23). Retrieve the gain values and rerun the simulations (red curve is the nominal response).

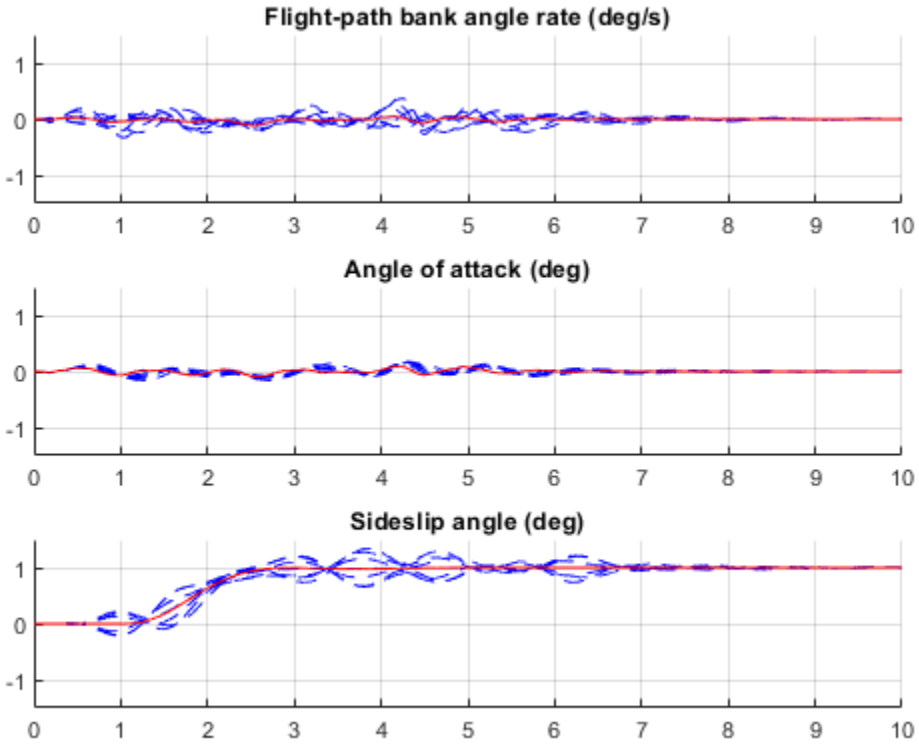
```
Ki = getBlockValue(ST, 'Ki');  Ki = Ki.d;
Kx = getBlockValue(ST, 'Kx');  Kx = Kx.d;

% Bank-angle setpoint simulation
plotResponses(OutageCases,1,0,0);
```




```
% Angle-of-attack setpoint simulation
plotResponses(OutageCases,0,1,0);
```




```
% Sideslip-angle setpoint simulation
plotResponses(OutageCases,0,0,1);
```



The controller now provides acceptable performance for all outage scenarios considered in this example. The design could be further refined by adding specifications such as minimum stability margins and gain limits to avoid actuator rate saturation.

```
rmpath(fullfile(matlabroot,'examples','control','main')) % remove example data
```

See Also

`systune (slTuner)` | `slTuner` | `TuningGoal.WeightedVariance` | `TuningGoal.Variance`

Related Examples

- “Fixed-Structure Autopilot for a Passenger Jet”

Passive Control of Water Tank Level

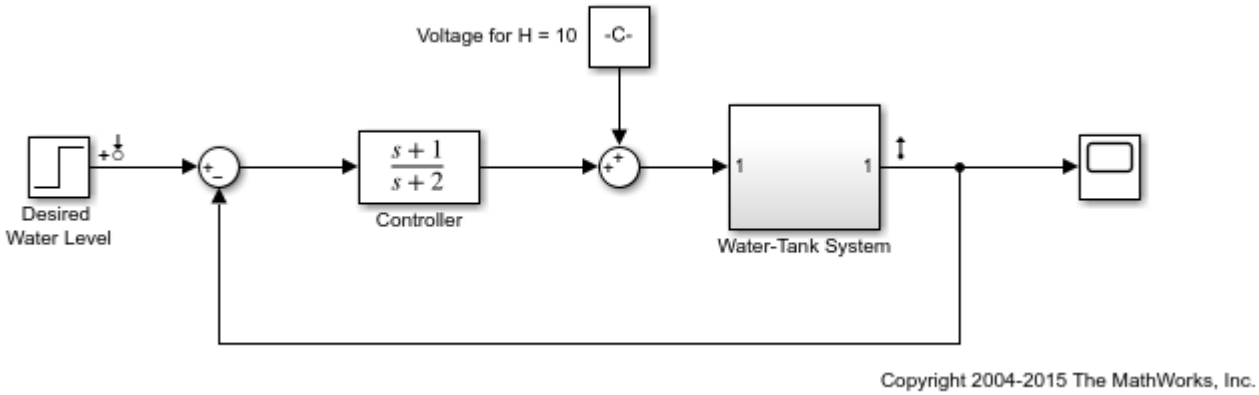
In this example, you learn how to use **Control System Tuner** app to design a controller for a nonlinear plant modeled in Simulink®. You accomplish the following tasks:

- Configure the model and app for compensator tuning
- Tune a first-order compensator using passivity-based design
- Simulate the closed-loop nonlinear response.

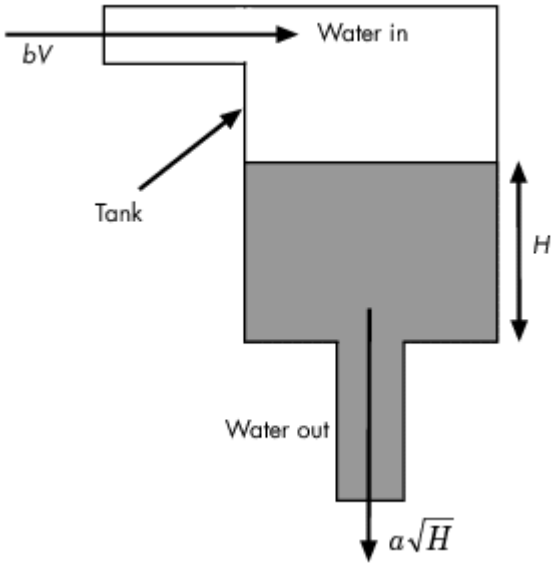
Simulink Model of the Control System

The watertank_comp_design model, shown in the figure below, models a feedback loop for regulating the water level in a water tank. The Controller block contains the first-order compensator to be tuned.

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



The Water Tank subsystem models the water-tank dynamics. Water enters the tank from the top at a rate proportional to the voltage, V, applied to the pump. The water leaves through an opening in the tank base at a rate that is proportional to the square root of the water height, H, in the tank. The presence of the square root in the water flow rate makes the plant nonlinear.



The nonlinear model for the water flow is

$$A\dot{x} = bu - a\sqrt{x}$$

$$y = x$$

where

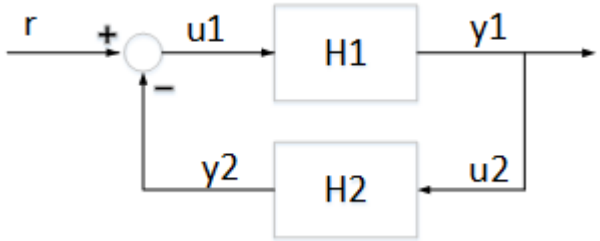
- $x = H$ denotes the height of water in the tank
- u denotes the voltage applied to the pump
- A denotes the cross-sectional area of the tank
- a and b are constants related to the flow rate into and out of the tank

This system is passive with storage function $V(x) = \frac{A}{2b}x^2$ since

$$\dot{V}(x) - uy = -\frac{a}{b}x\sqrt{x} \leq 0$$

Passivity-Based Control

By the Passivity Theorem, the negative-feedback interconnection of two strictly passive systems H_1 and H_2 is always stable.



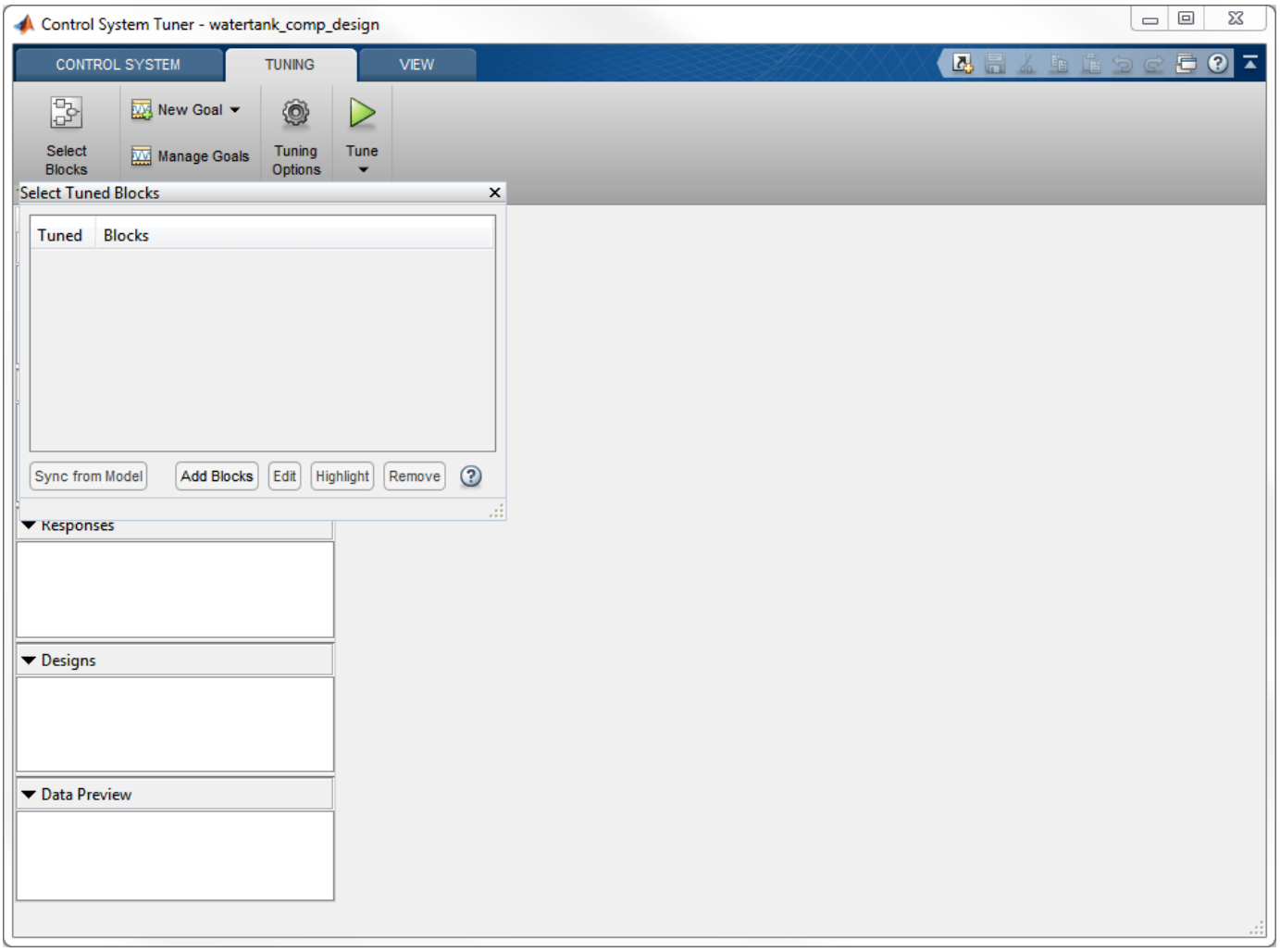
Since the water tank system is passive, it makes sense to require that the controller be strictly passive to guarantee closed-loop stability even when the plant model is inaccurate.

Compensator Tuning Using Control System Tuner

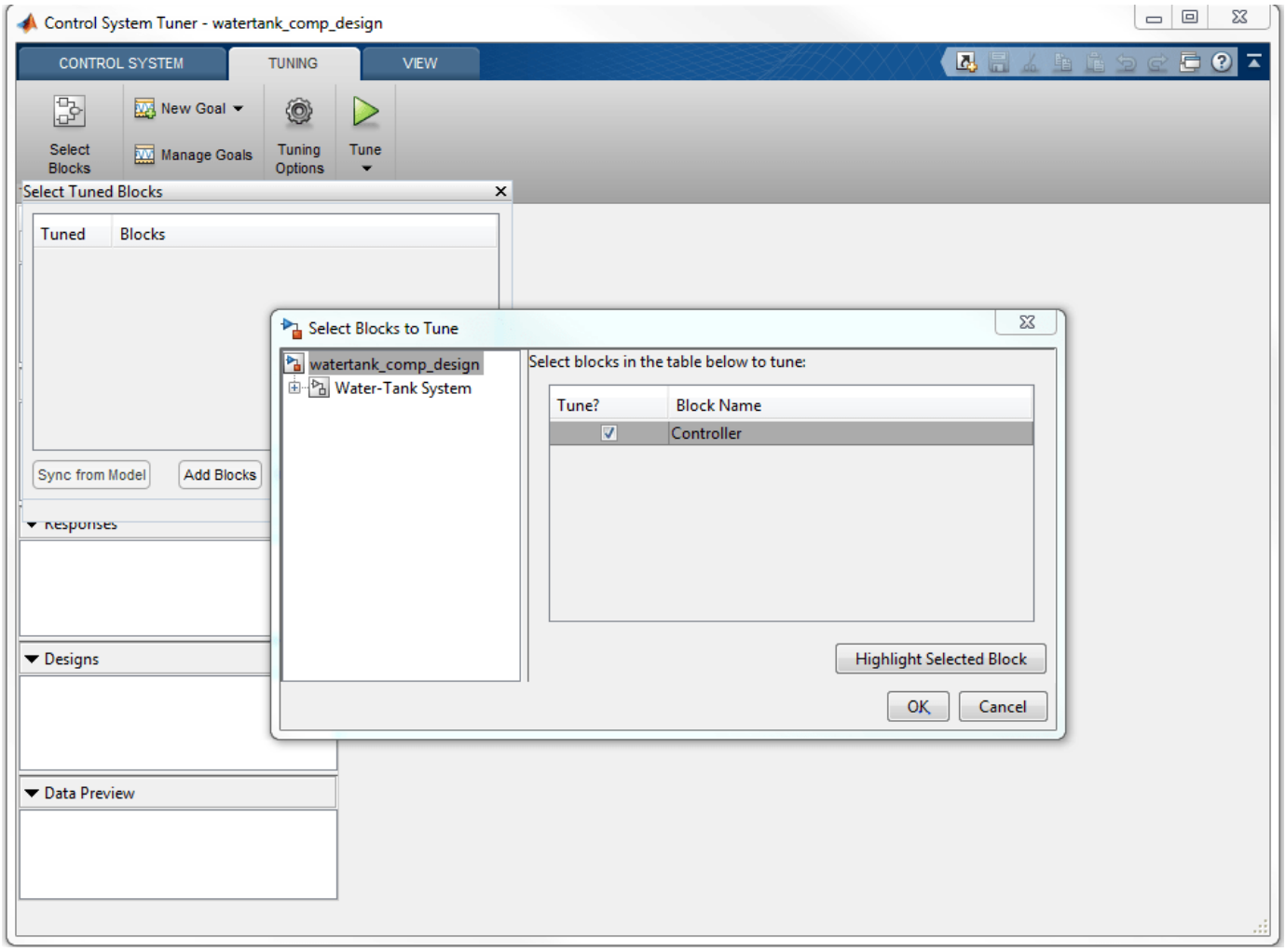
You can use the **Control System Tuner** app to tune the Controller block.

Step 1: Open the **Control System Tuner** app. In the Simulink model window, on the **Apps** tab, in the **Apps** gallery, click **Control System Tuner**.

Step 2: Launch the tuned block selector from the **Select Blocks** button in the **Tuning** tab



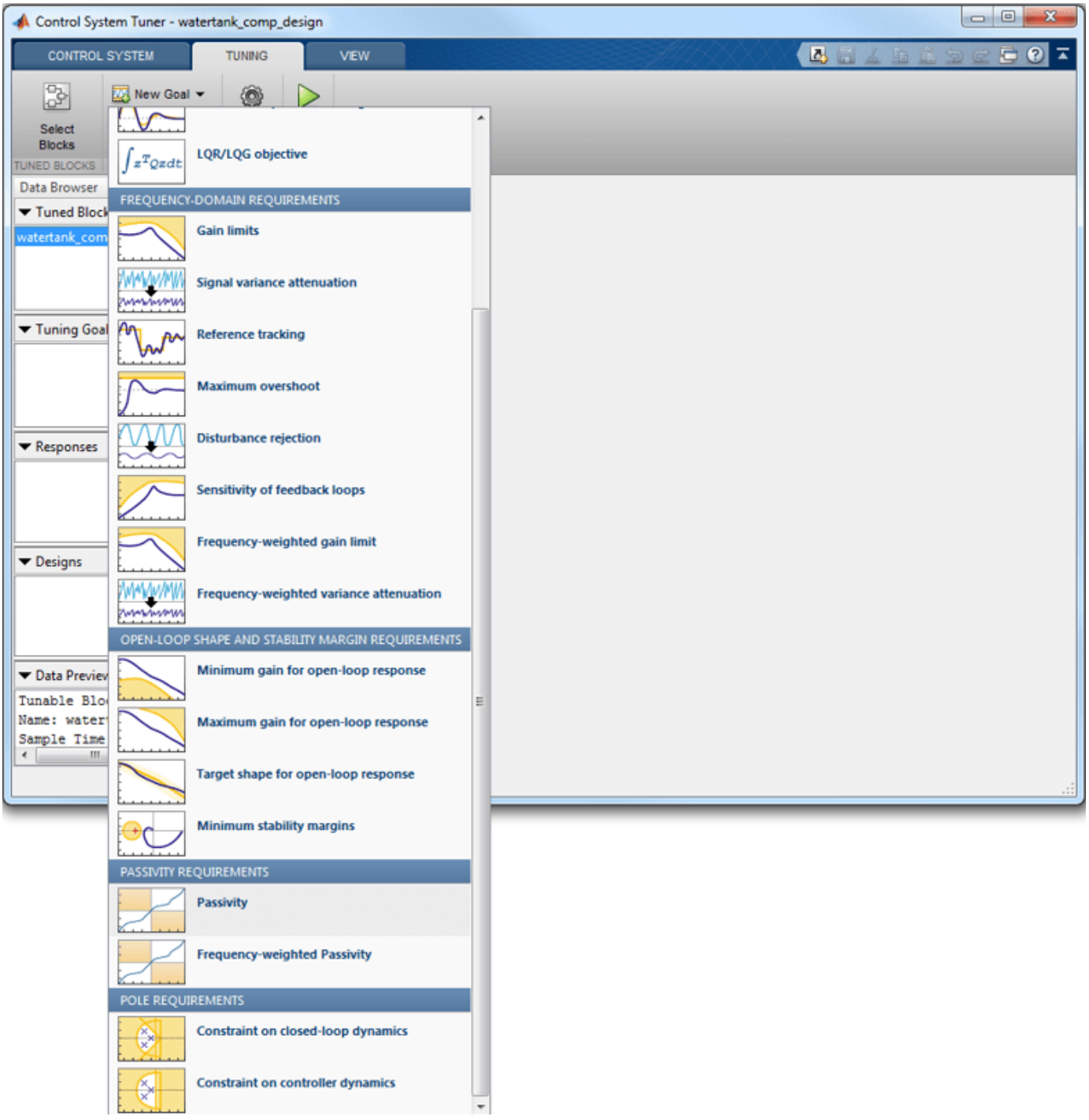
Step 3: Select the Controller block and Click OK. This block now appears in the Tuned Blocks list.



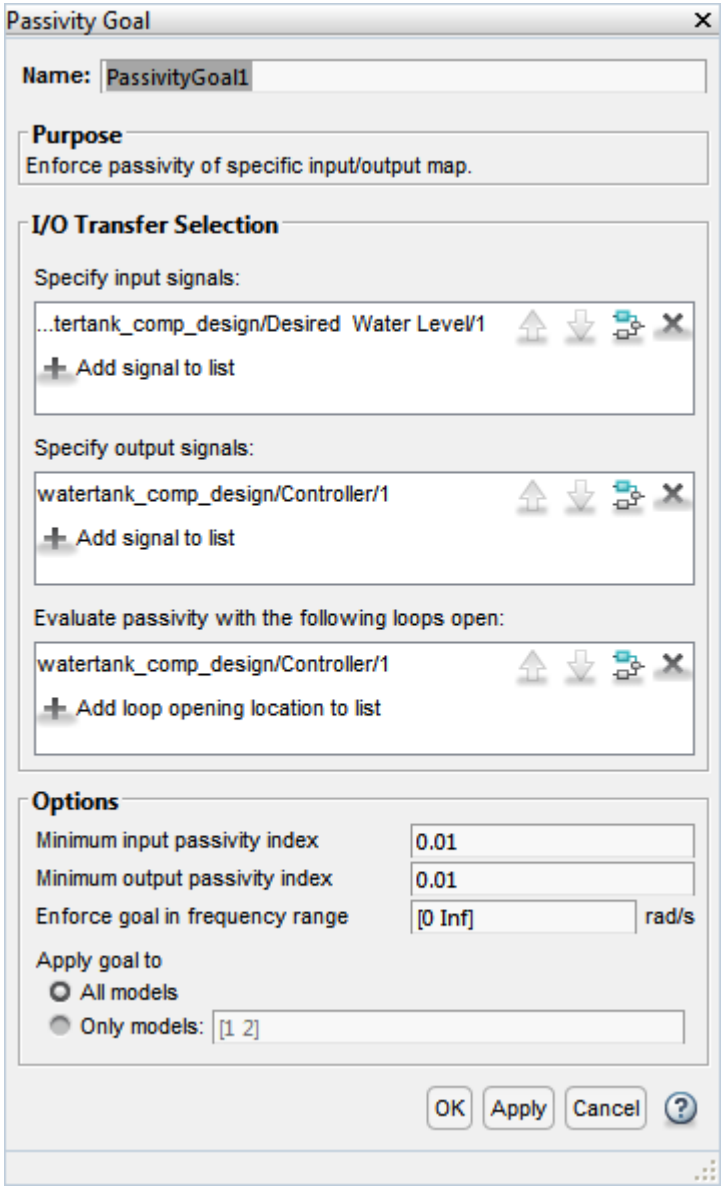
Step 4: Specify the tuning goals. Here, there are two main goals:

1. Track step changes in water level
2. Make the controller passive

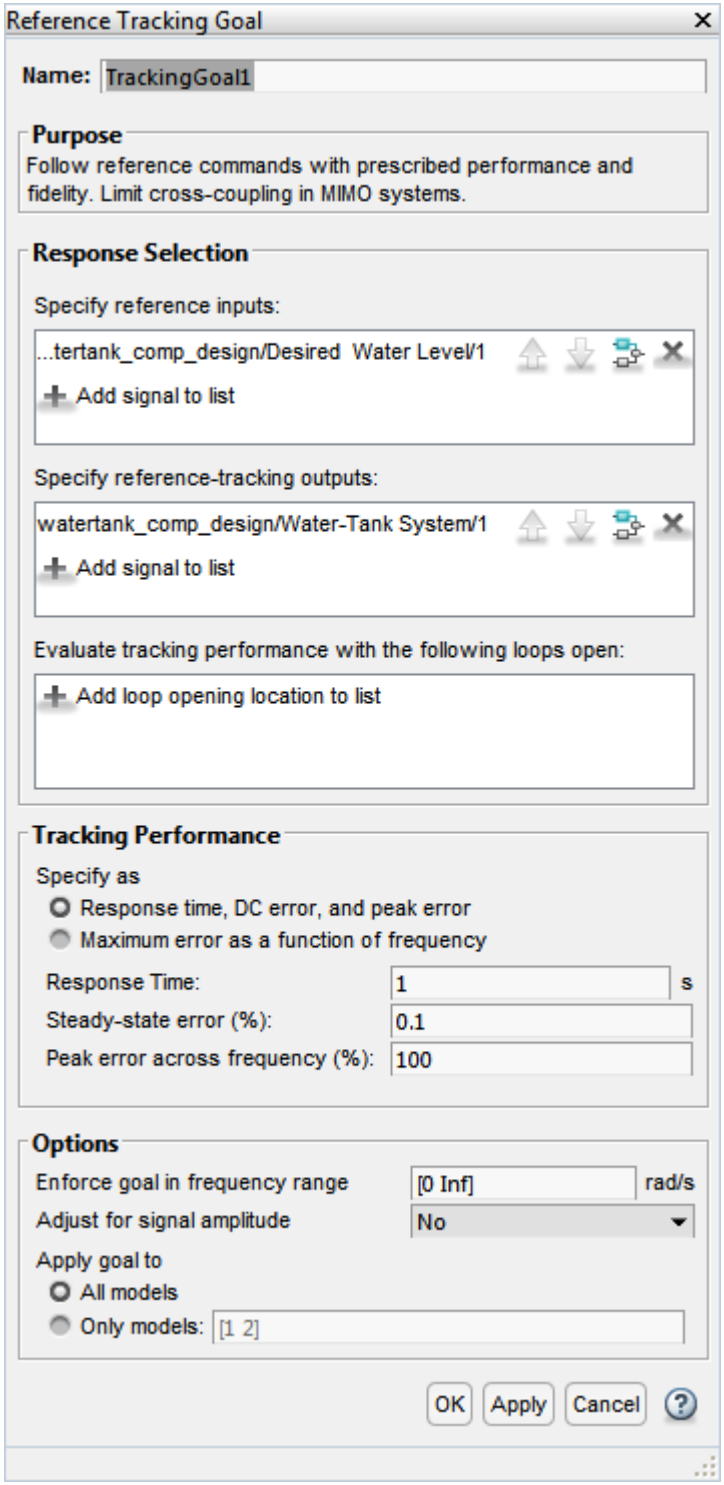
Click the **New Goal** drop-down list, and first add a **Passivity** goal.



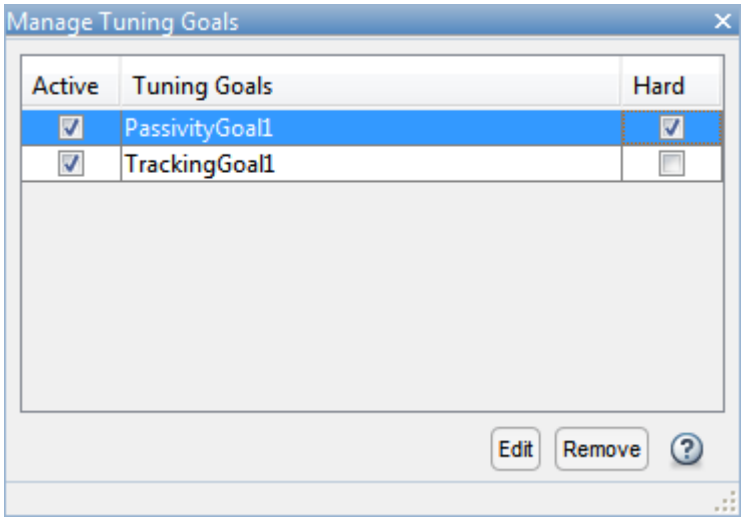
Configure this goal to apply to the Controller block only. This is done by setting the input signal to be the "Desired Water Level", the output signal to be the output of the Controller block, and the loop opening to be at the Controller block output. Also specify minimum passivity indices of 0.01 at the inputs and outputs to enforce strict passivity.



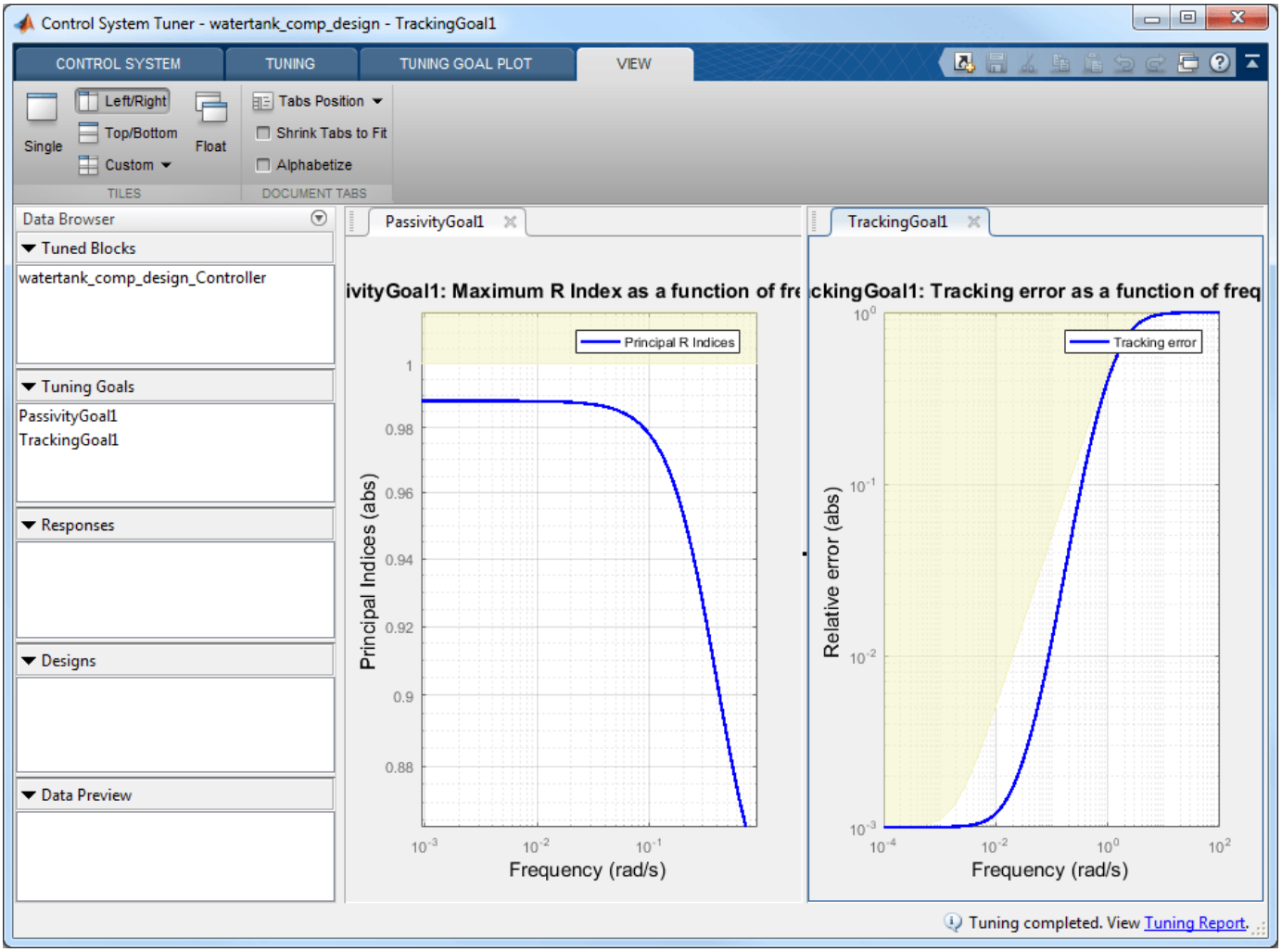
Next, add a **Reference Tracking** goal from the **New Goal** drop-down list. Configure this goal for a 1 second response time.



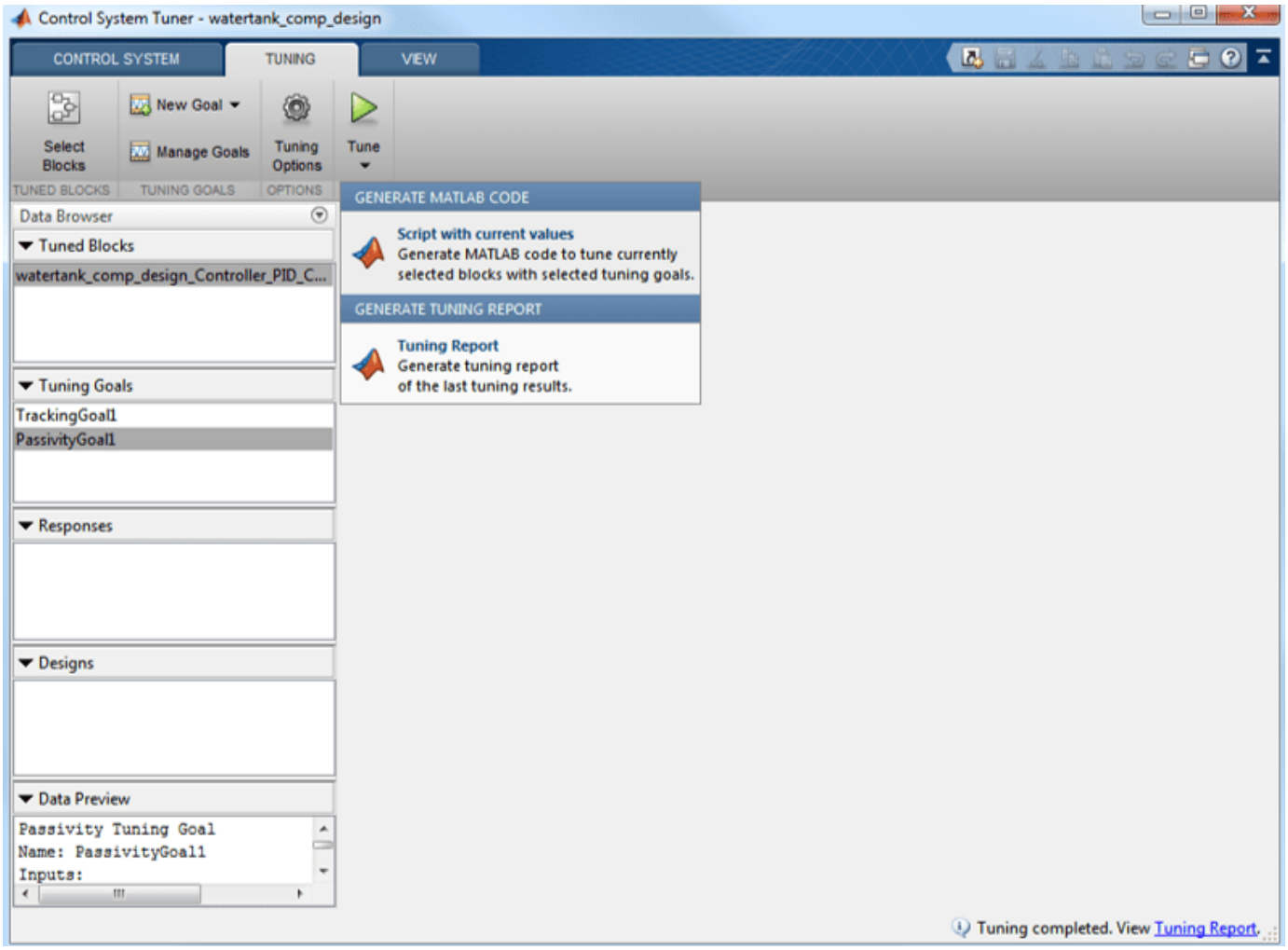
Finally, click on the **Manage Goals** button off the **Tuning** tab and mark the Passivity goal as a hard tuning constraint.



Step 5: You are ready to tune the Controller block. Click the Tune button. You can view the tuning results side-by-side by selecting **Left/Right** in the **View** tab.

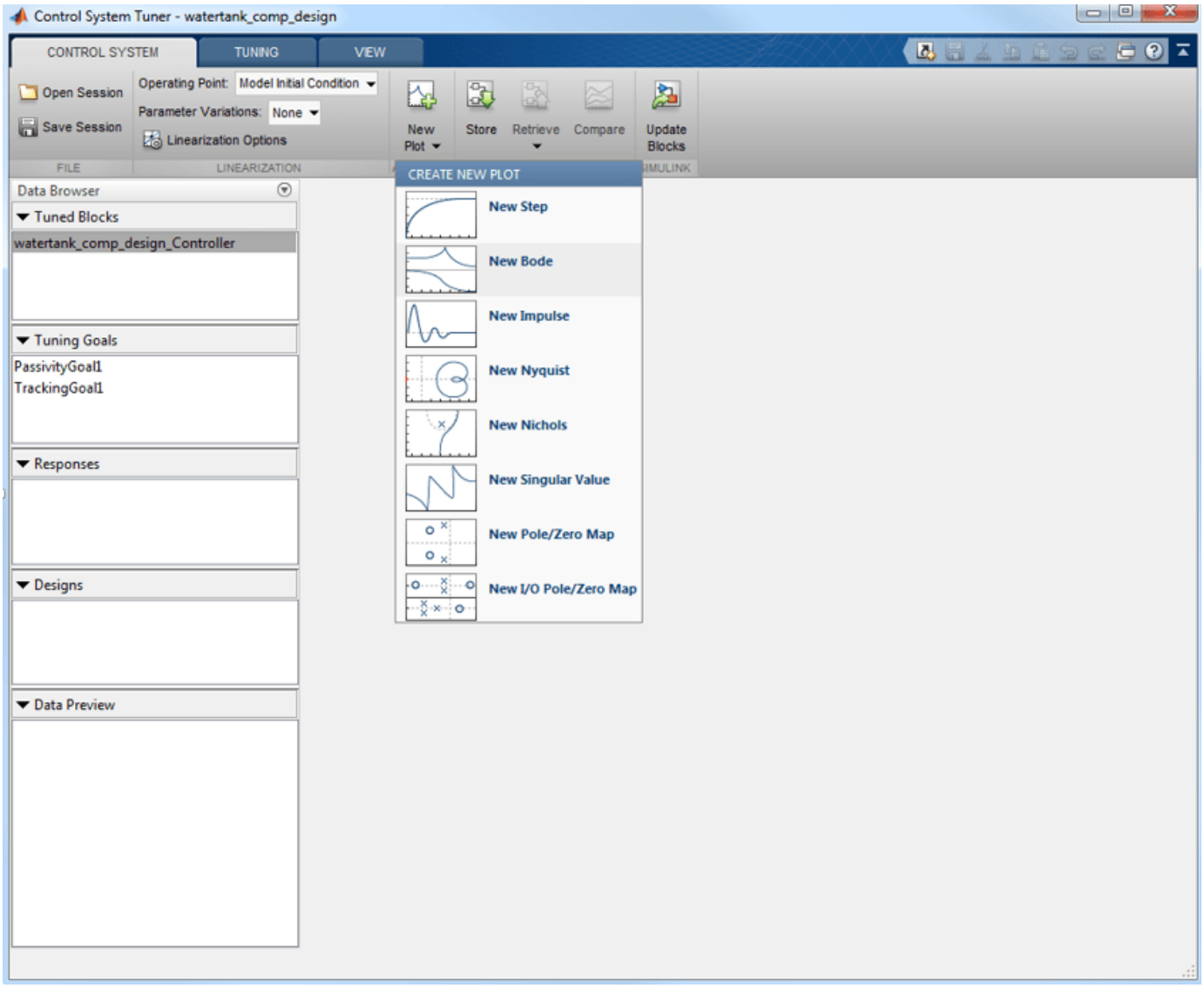


You can further analyze these results by generating a MATLAB script that reproduces this tuning process.

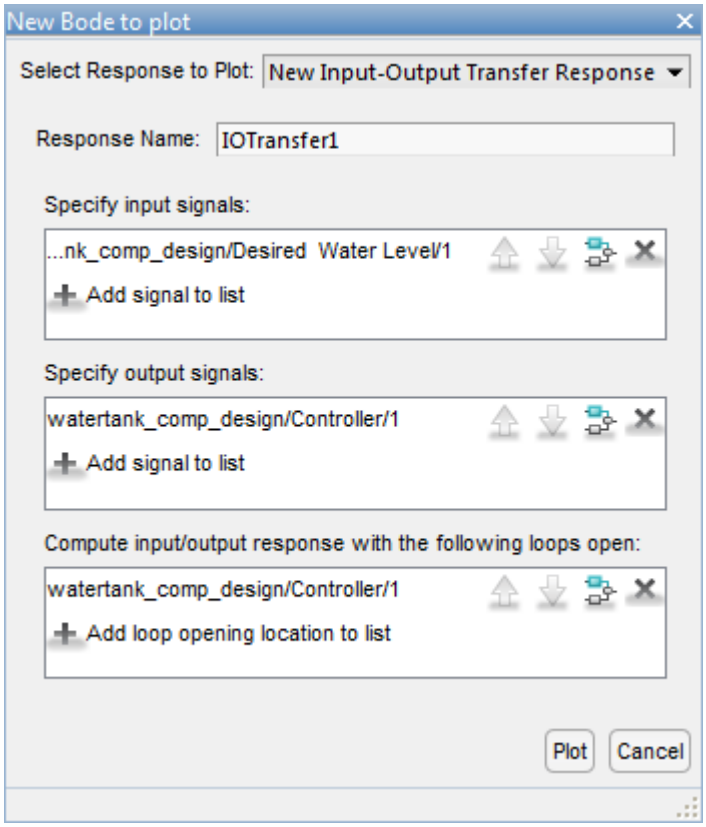


Closed-Loop Simulation

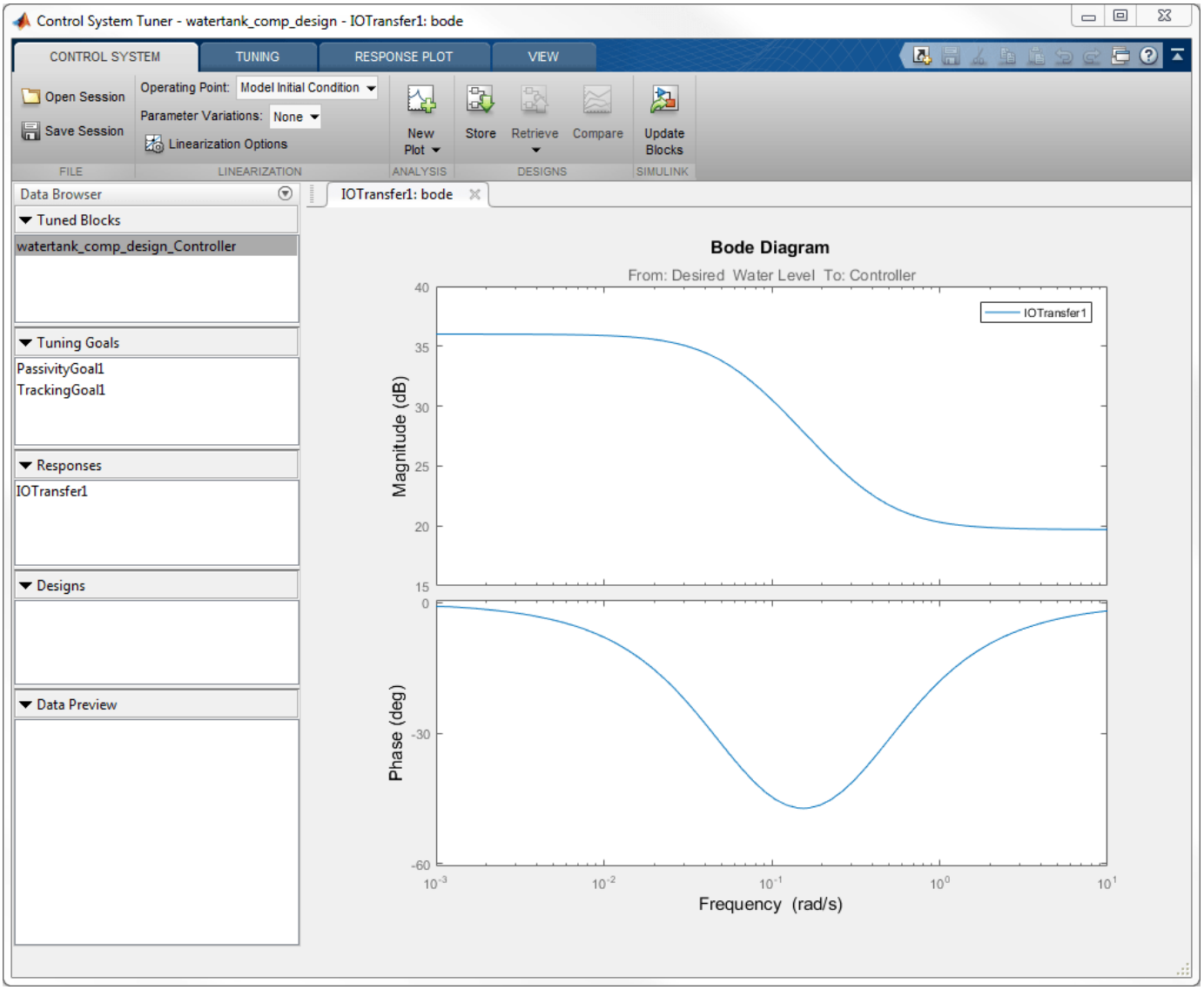
You can view the Bode plot of the tuned controller. Click on the **New Plot** button off the **Control System** tab. Select **New Bode** from the drop-down list.



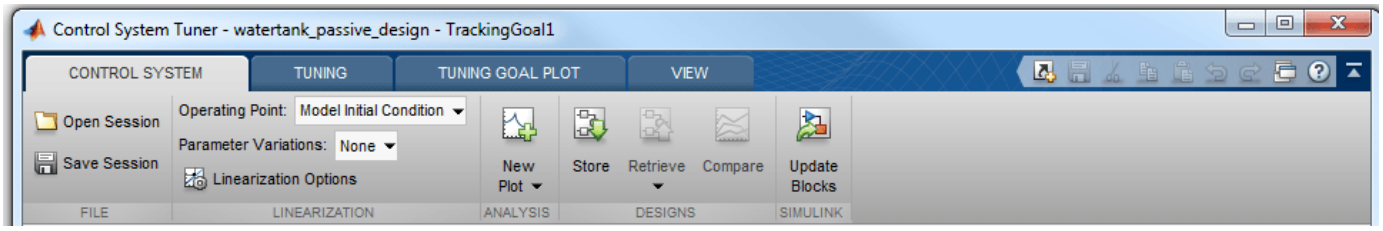
The controller response can be specified as follows.



Click on the **Plot** button. The bode plot is shown in the following figure.



You can also simulate the closed-loop nonlinear response with the tuned controller. First, update the Controller block by clicking **Update Blocks** in the **Control System** tab.



In the Simulink model, double click the Scope block to open the Scope window, then simulate the model.

The nonlinear response of the tuned control system appears in the Scope window. This simulation shows that the tracking performance is satisfactory.

See Also

Control System Tuner

Related Examples

- “About Passivity and Passivity Indices”
- “Vibration Control in Flexible Beam”

Tuning for Multiple Values of Plant Parameters

This example shows how to use **Control System Tuner** to tune a control system when there are parameter variations in the plant. The control system used in this example is an active suspension of a quarter-car model. The example uses **Control System Tuner** to tune the system to meet performance objectives when parameters in the plant vary from their nominal values.

Quarter-Car Model and Active Suspension Control

A simple quarter-car model of an active suspension system is shown in Figure 1. The quarter-car model consists of two masses, a car chassis with mass m_b and a wheel assembly of mass m_w. There is a spring k_s and damper b_s between the masses, which models the passive spring and shock absorber. The tire between the wheel assembly and the road is modeled by the spring k_t.

Active suspension introduces a force f_s between the chassis and wheel assembly and allows the designer to balance driving objectives such as passenger comfort and road handling with the use of a feedback controller.

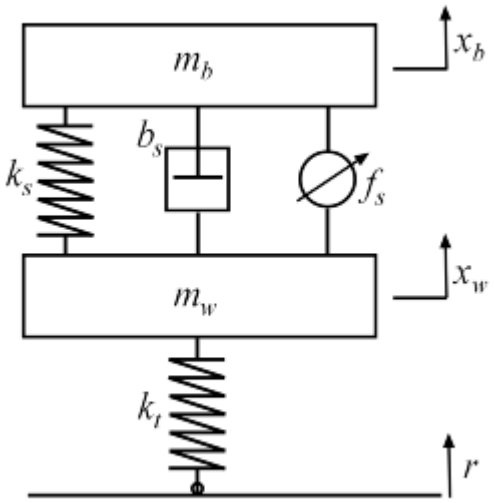
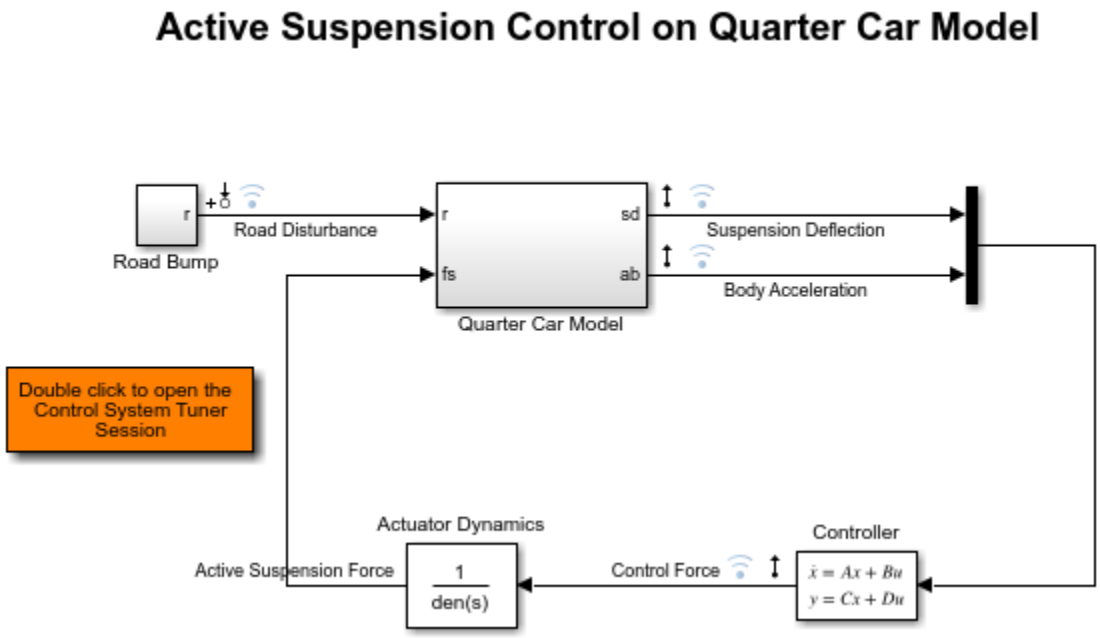


Figure 1: Quarter-car model of active suspension.

Control Architecture

The quarter-car model is implemented using Simscape. The following Simulink model contains the quarter-car model with active suspension, controller and actuator dynamics. Its inputs are road disturbance and the force for the active suspension. Its outputs are the suspension deflection and body acceleration. The controller uses these measurements to send a control signal to the actuator that creates the force for active suspension.

```
mdl = 'rct_suspension.slx';
open_system(mdl)
```



Control Objectives

The example has the following three control objectives:

- Good handling defined from road disturbance to suspension deflection.
- User comfort defined from road disturbance to body acceleration.
- Reasonable control bandwidth.

The nominal values of the spring constant k_s and damper b_s between the body and the wheel assembly are not exact and due to the imperfections in the materials, these values can be constant but different. Assess the impact on the system control using a variety of parameter values.

Model the road disturbance of magnitude seven centimeters and use a constant weight.

```
Wroad = ss(0.07);
```

Define the closed-loop target for handling from road disturbance to suspension deflection as

```
HandlingTarget = 0.044444 * tf([1/8 1],[1/80 1]);
```

Define the target for comfort from road disturbance to body acceleration.

```
ComfortTarget = 0.6667 * tf([1/0.45 1],[1/150 1]);
```

Limit the control bandwidth by the weight function from road disturbance to the control signal.

```
Wact = tf(0.1684*[1 500],[1 50]);
```

For more information on selecting the closed-loop targets and the weight function, see “Robust Control of an Active Suspension” (Robust Control Toolbox).

Controller Tuning

To open a **Control System Tuner** session for active suspension control, in the Simulink model, Double click to the orange block. Tuned block is set to the second order Controller and three tuning goals are defined to achieve the handling, comfort and control bandwidth as described above. In order to see the performance of the tuning, the step responses from road disturbance to suspension deflection, the body acceleration and the control force are plotted.

Handling, comfort, and control bandwidth goals are defined as gain limits, `HandlingTarget/Wroad`, `ComfortTarget/Wroad` and `Wact/Wroad`. All gain functions are divided by `Wroad` to incorporate the road disturbance.

The open-loop system with zero controller violates the handling goal and results in highly oscillatory behavior for both suspension deflection and body acceleration with long settling time.

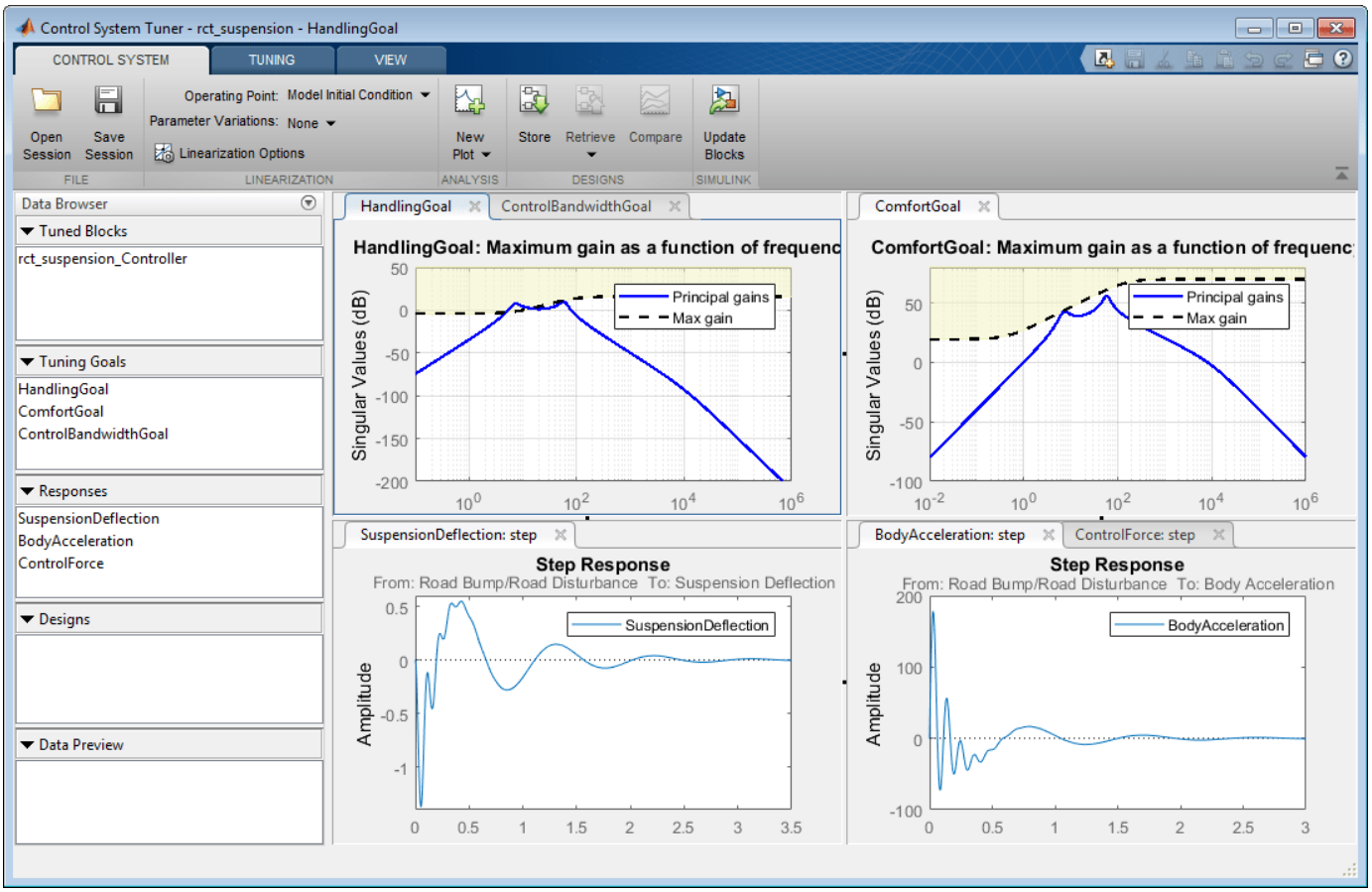


Figure 2: Control System Tuner with Session File.

To tune the controller using **Control System Tuner**, on the **Tuning** tab, click **Tune**. As shown in Figure 3, this design satisfies the tuning goals and the responses are less oscillatory and converges quickly to zero.

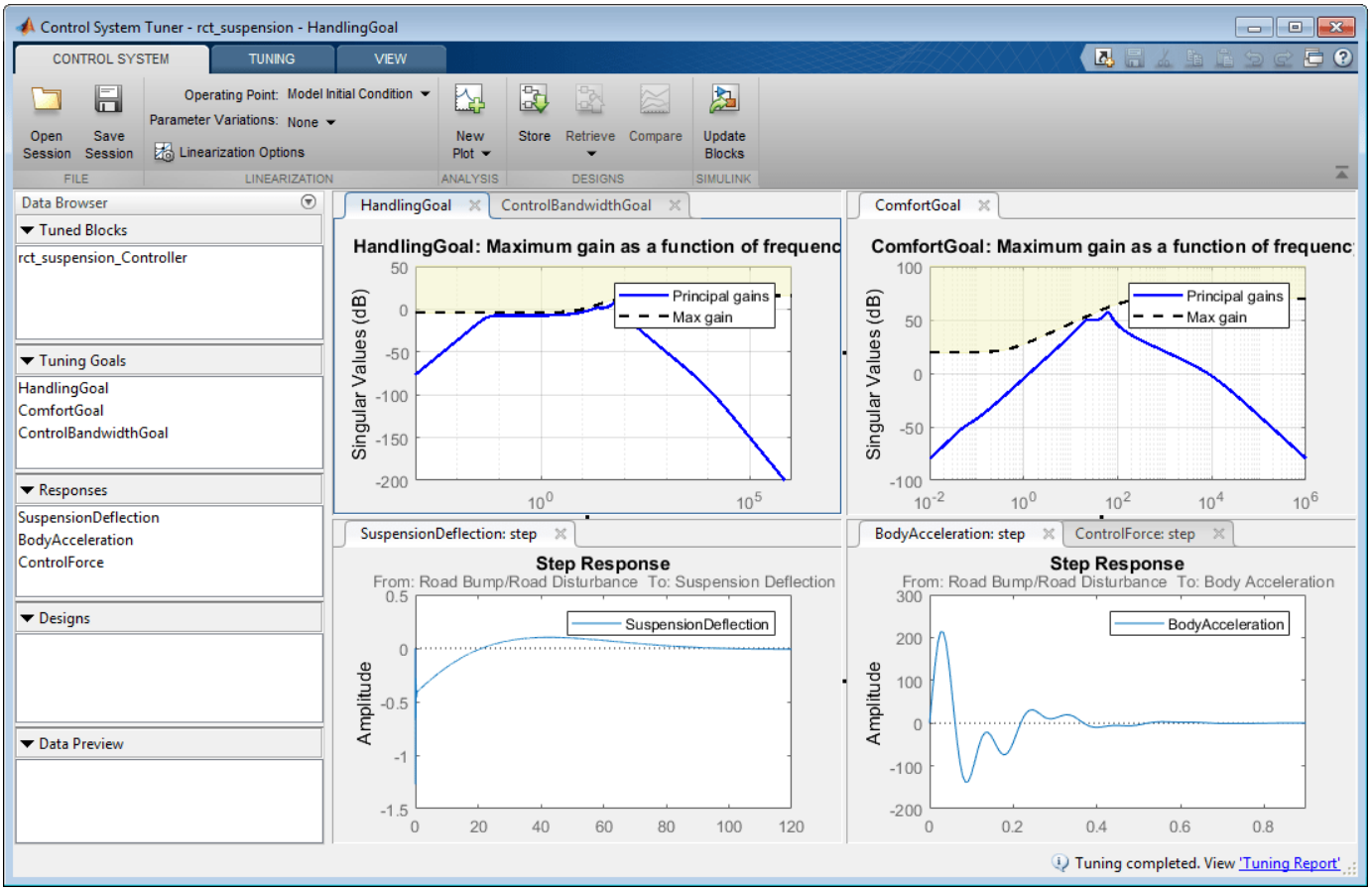


Figure 3: Control System Tuner after tuning.

Controller Tuning for Multiple Parameter Values

Now, try to tune the controller for multiple parameter values. The default value for car chassis of mass m_b is 300 kg. Vary the mass to 100 kg, 200 kg and 300 kg for different operation conditions.

In order to vary these parameters in **Control System Tuner**, on the **Control System** tab, under **Parameter Variations**, select **Select parameters to Vary**. Define the parameters in the dialog that opens.

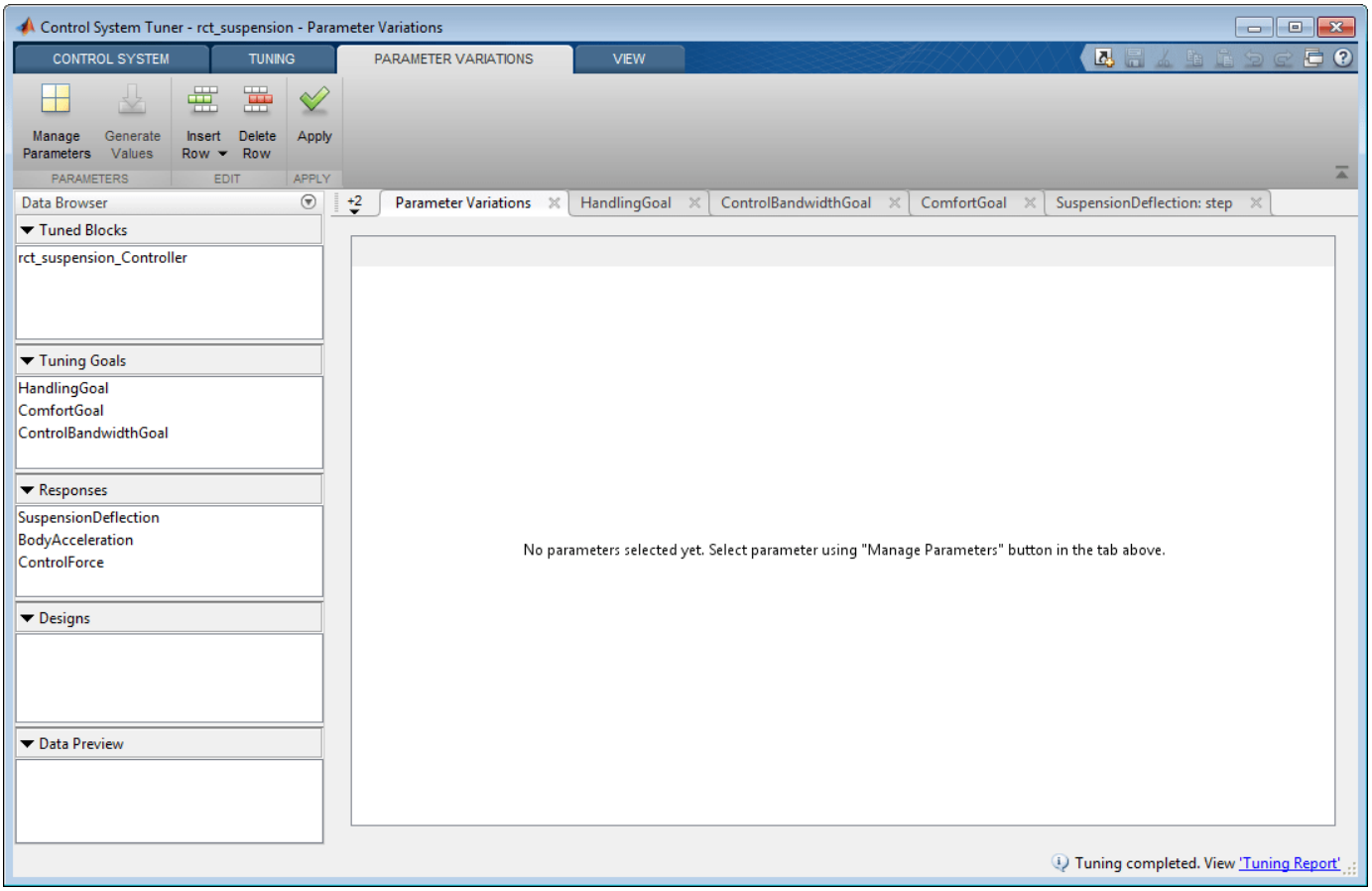


Figure 4: Defining parameter variations.

On the **Parameter Variations** tab, click **Manage Parameters**. In the Select model variables dialog box, select `Mb`.

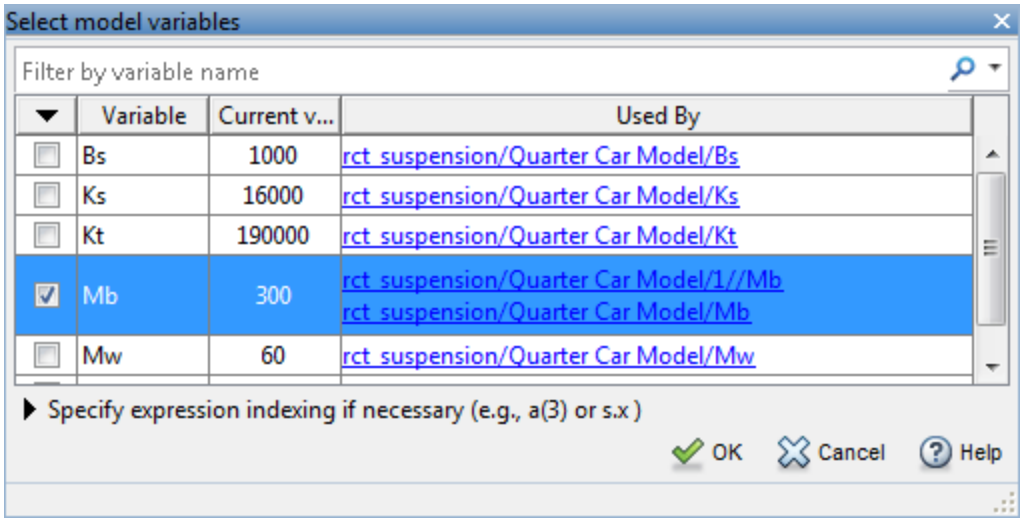


Figure 5: Select a parameter to vary from the model.

Now, the parameter `Mb` is added with default values in the parameter variations table.

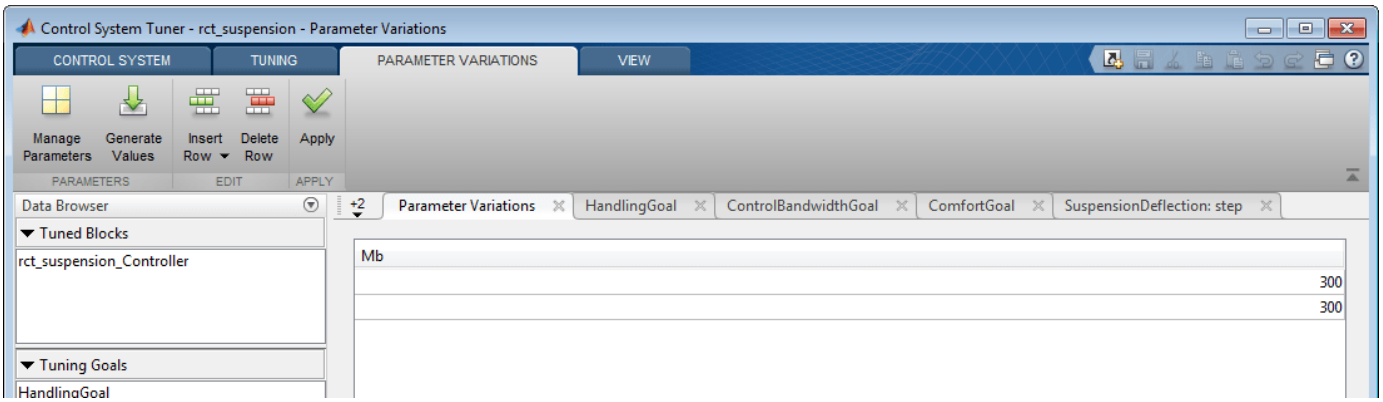


Figure 6: Parameter variations table with default values.

To generate variations quickly, click **Generate Values**. In the Generate Parameter Values dialog box, define values 100, 200, 300 for `Mb`, and click `Overwrite`.

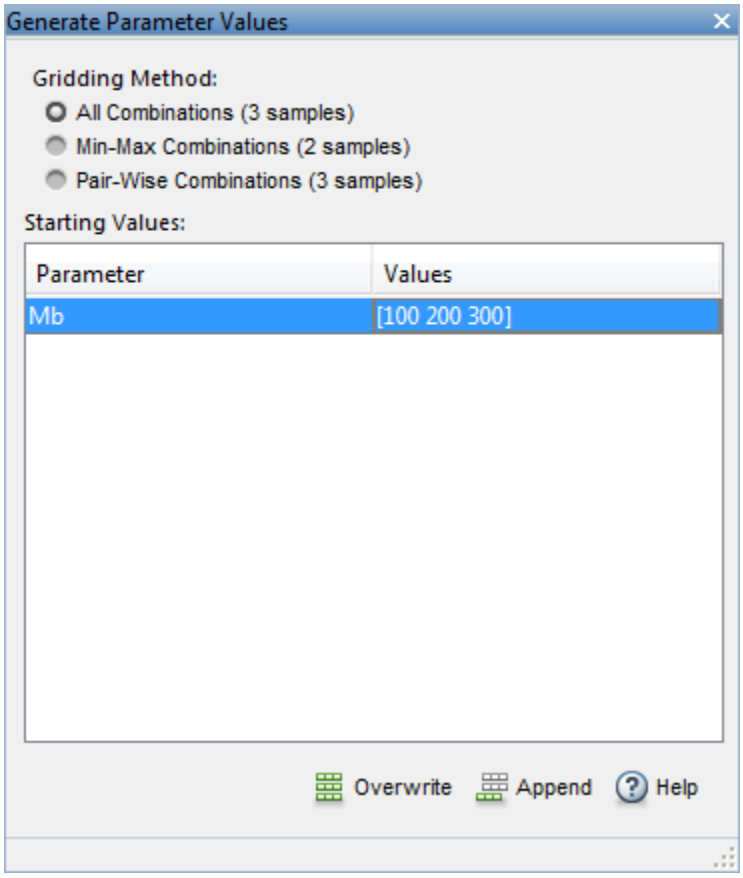


Figure 7: Generate values window.

All values are populated in the parameter variations table. To set the parameter variations to **Control System Tuner**, click **Apply**.

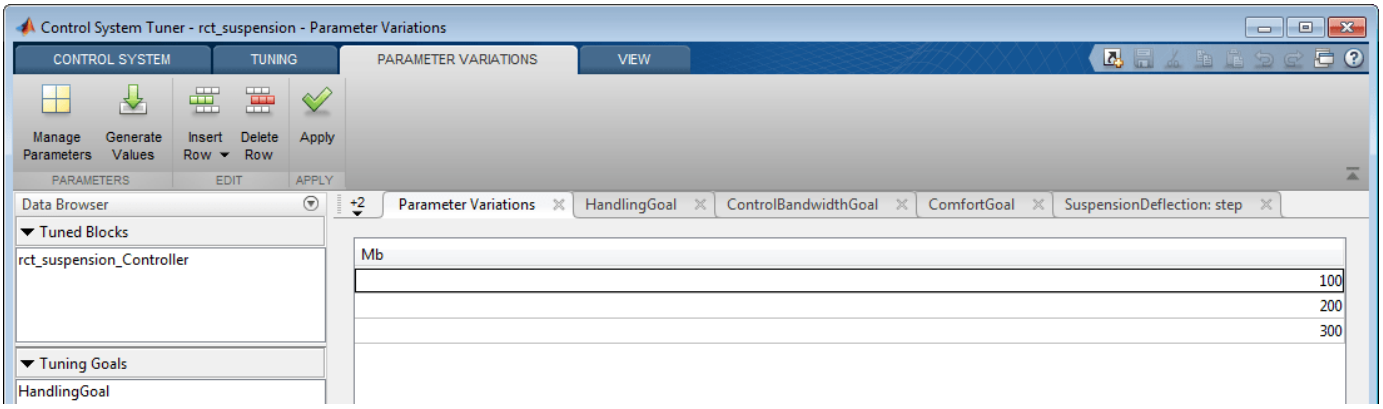


Figure 8: Parameter variations table with updated values.

Multiple lines appear in the tuning goal and response plots due to the varying parameters. The controller obtained for these nominal parameter values results in an unstable closed-loop system.

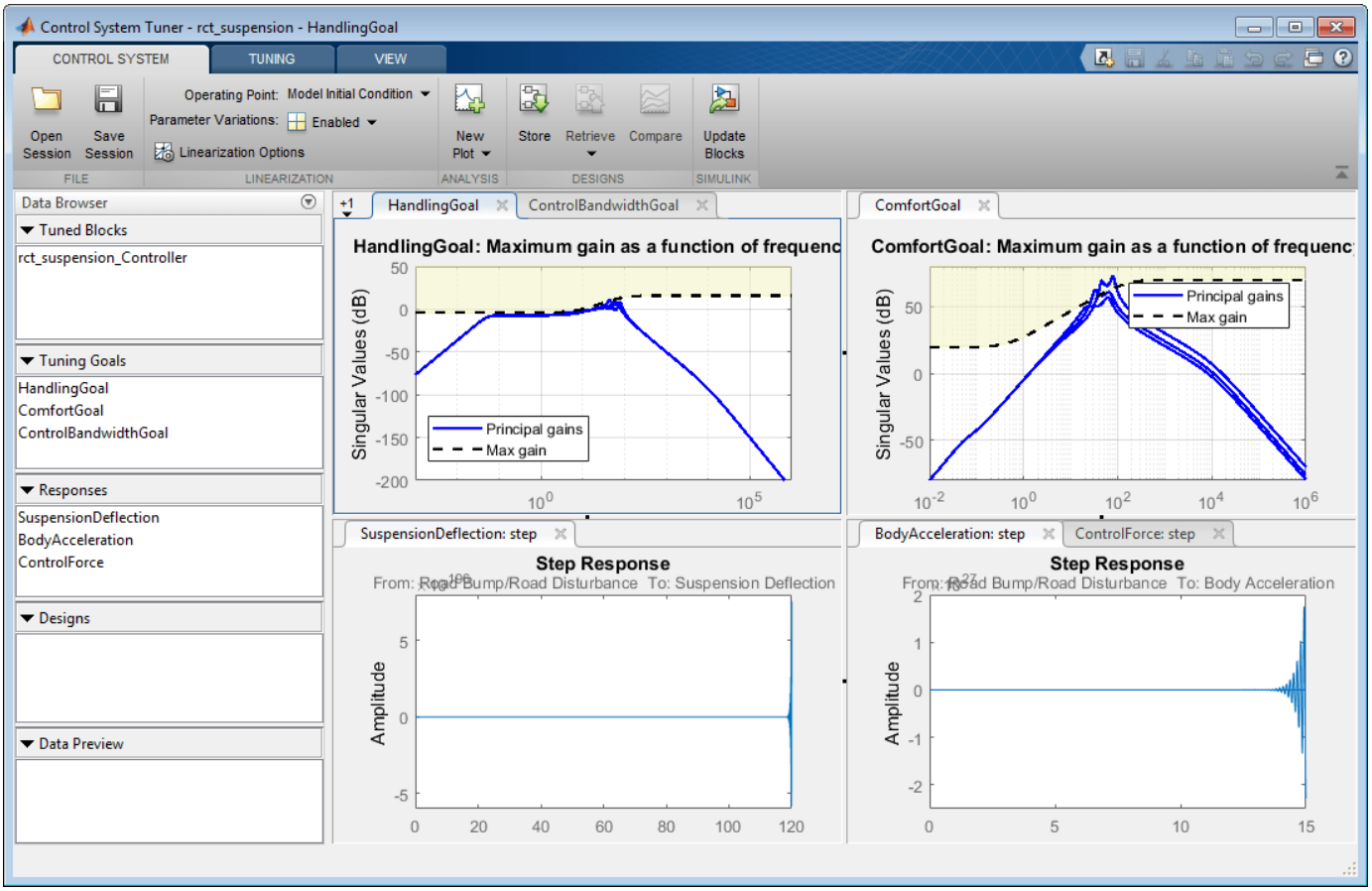


Figure 9: Control System Tuner with multiple parameter variations.

Tune the controller to satisfy the handling, comfort, and control bandwidth objectives by clicking **Tune** in **Tuning** tab. The tuning algorithm tries to satisfy these objectives for the nominal parameters

and for all parameter variations. This is a challenging task in contrast to nominal design as shown in Figure 10.

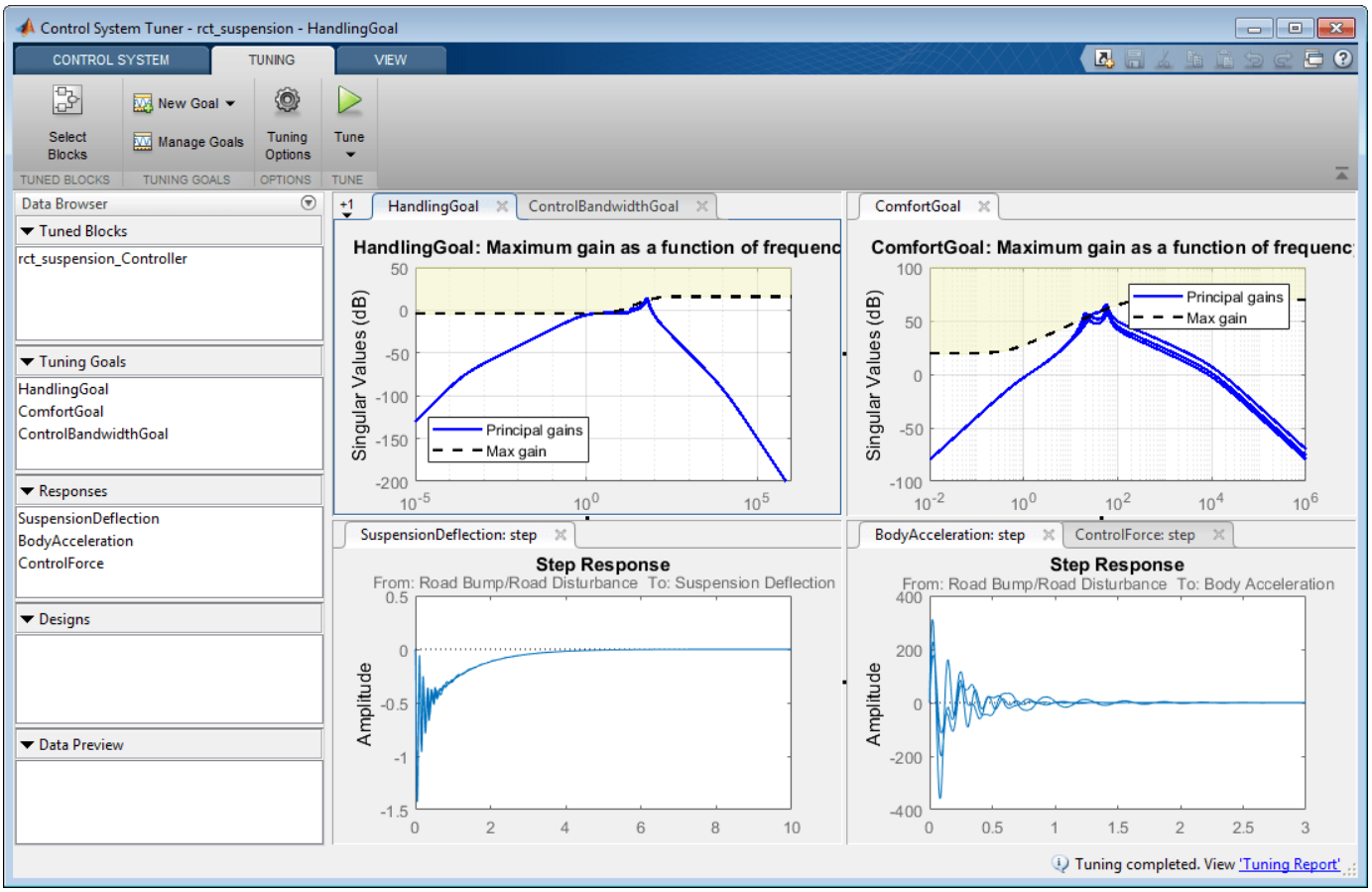


Figure 10: Control System Tuner with multiple parameter variations (Tuned).

Control System Tuner tunes the controller parameters for the linearized control system. To examine the performance of the tuned parameters on the Simulink model, update the controller in the Simulink model by clicking **Update Blocks** on the **Control System** tab.

Simulate the model for each of the parameter variations. Then, using the Simulation Data Inspector, examine the results for all simulations. The results are shown in Figure 11. For all three parameter variations, the controller tries to minimize the suspension deflection and body acceleration with minimal control effort.

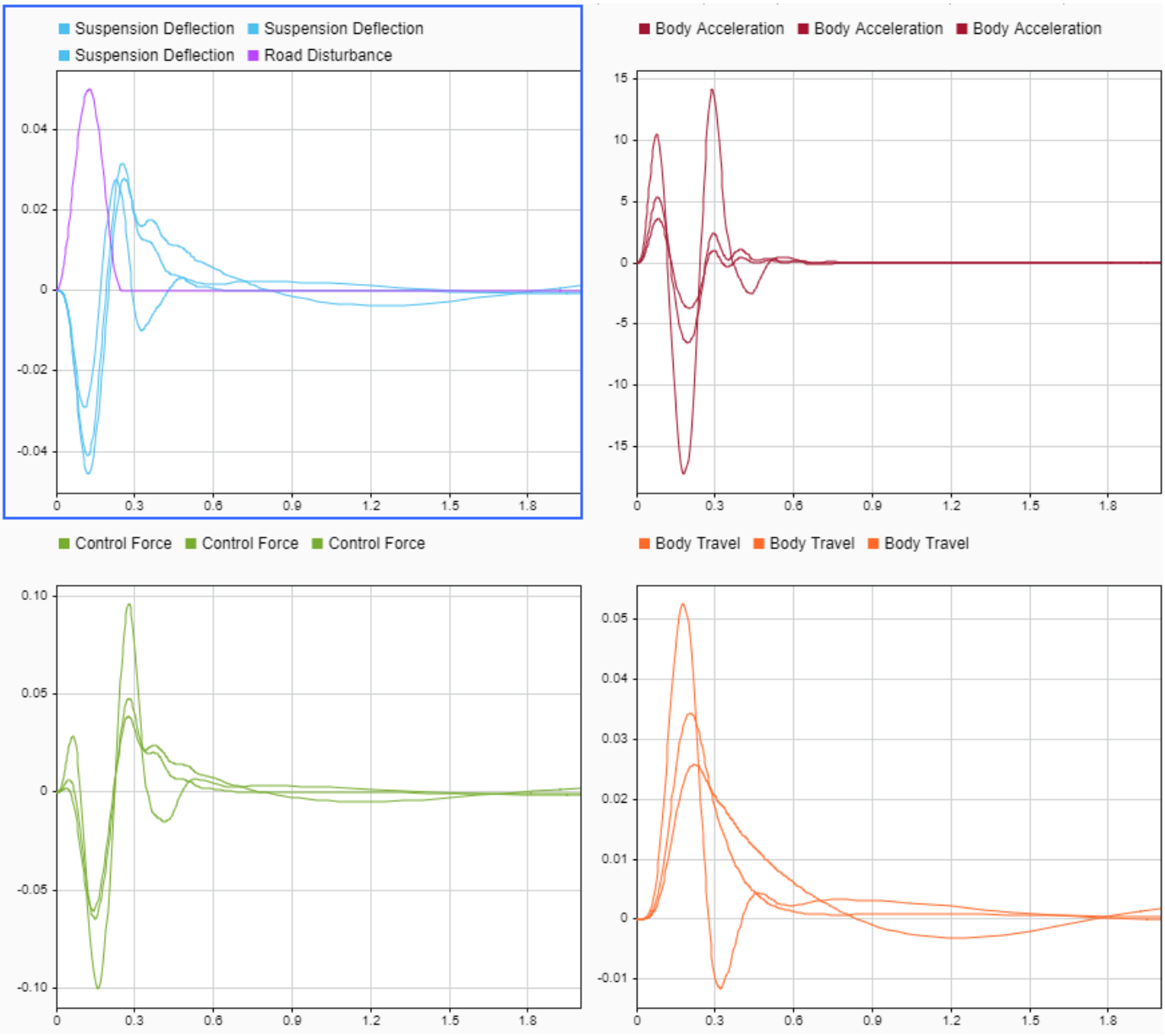


Figure 11: Controller performance on the Simulink model.

See Also

Control System Tuner

More About

- “Create Response Plots in Control System Tuner”

Control System Tuning Applications

UAV Inflight Failure Recovery

This example shows how to use **Control System Tuner** to tune the fixed-structure PID controllers of a multicopter for nominal flight conditions and fault conditions. Here, you use a gain-scheduled approach to recover from a single rotor failure and land the UAV.

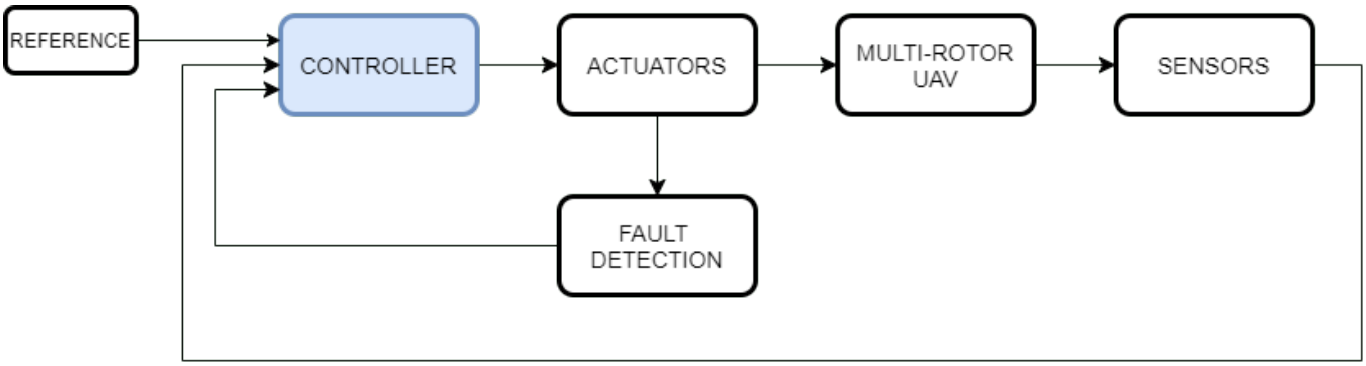
UAV Package Delivery Model

This example uses a model based on the model discussed in the “UAV Package Delivery” (UAV Toolbox) example. The high-fidelity 6-DOF plant model is based on Simulink Drone Reference Application.

Open the Simulink® Project.

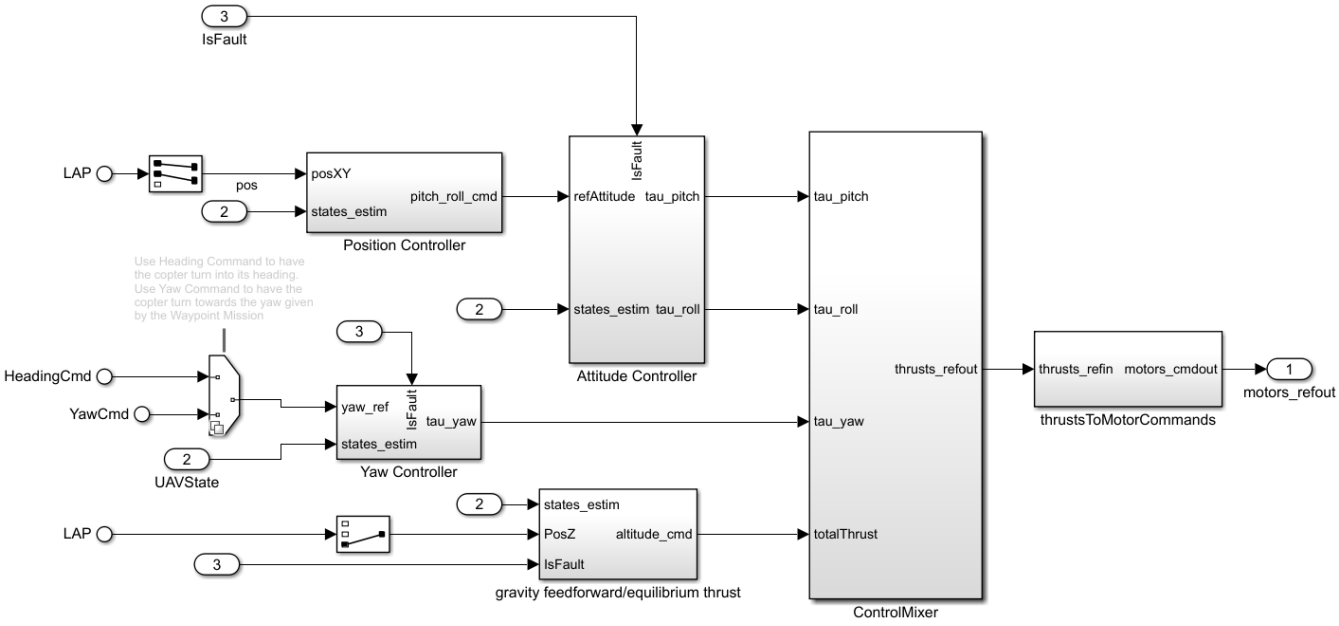
```
run('scdUAVInflightFailureRecoveryStart.m')
```

In this example, a single rotor failure is simulated by injecting a multiplicative gain vector in the plant. To demonstrate the simulation results of the gain-scheduled controllers, the actuator commands are used as features to design a fault detection algorithm, and a threshold based on nominal and fault induced simulations is used to detect the fault.



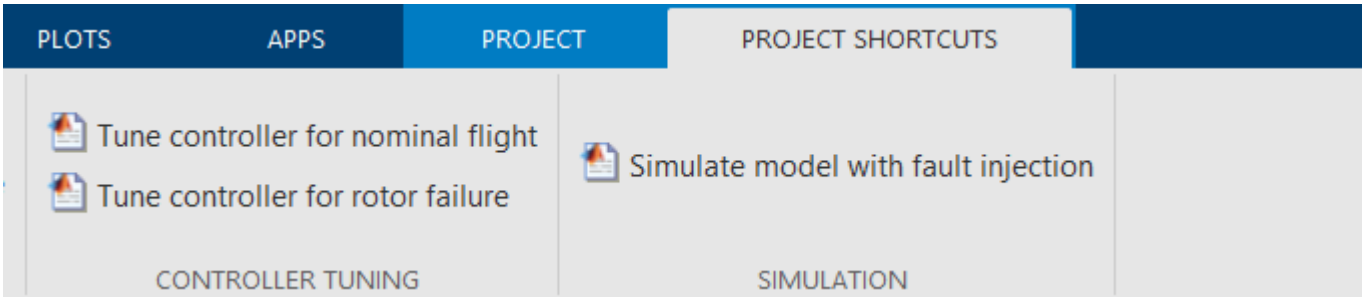
The controller consists of a position (X, Y) and attitude (pitch, roll) loop, a yaw control loop, and an altitude (Z) control loop. The Rotor Fault Injection Gains block is used to define the fault condition for the plant model. **Control System Tuner** is used to tune the inner attitude controller and the altitude controller. The gains are tuned for nominal flight conditions and fault conditions. A fault-recovery control strategy is implemented using Varying PID Controller and Lookup Table blocks scheduled based on the fault detection indicator. The following figure shows the controller subsystem used for tuning.

```
open_system('MultirotorModel/Inner Loop and Plant Model/High-FidelityModel_RotorFault/Controller,
```



Controller Tuning

This section describes how to specify tunable elements and create tuning goals using **Control System Tuner**. To launch a preconfigured session instead, use **Project Shortcuts**.



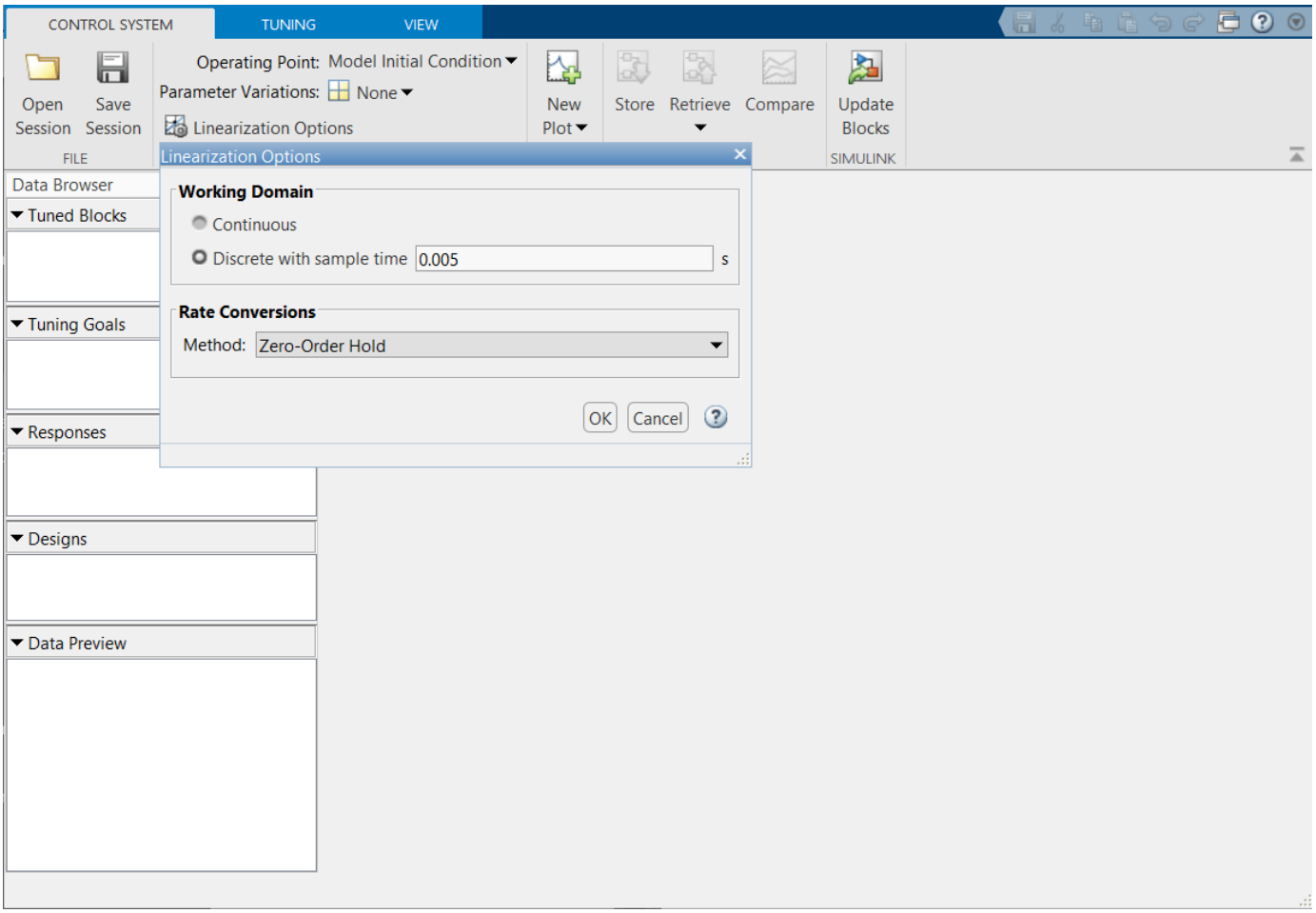
The controller and plant are extracted in a separate model, `MultirotorModelControlDesign.slx`, set up for tuning controller gains using **Control System Tuner**. The initial condition for the integral gains is set as 0.01 to suppress overshoot while maintaining zero steady-state error. The rotor gain multiplier parameter is set to indicate nominal mode with all rotors operational.

Open the model.

```
open_system("MultirotorModelControlDesign.slx")
```

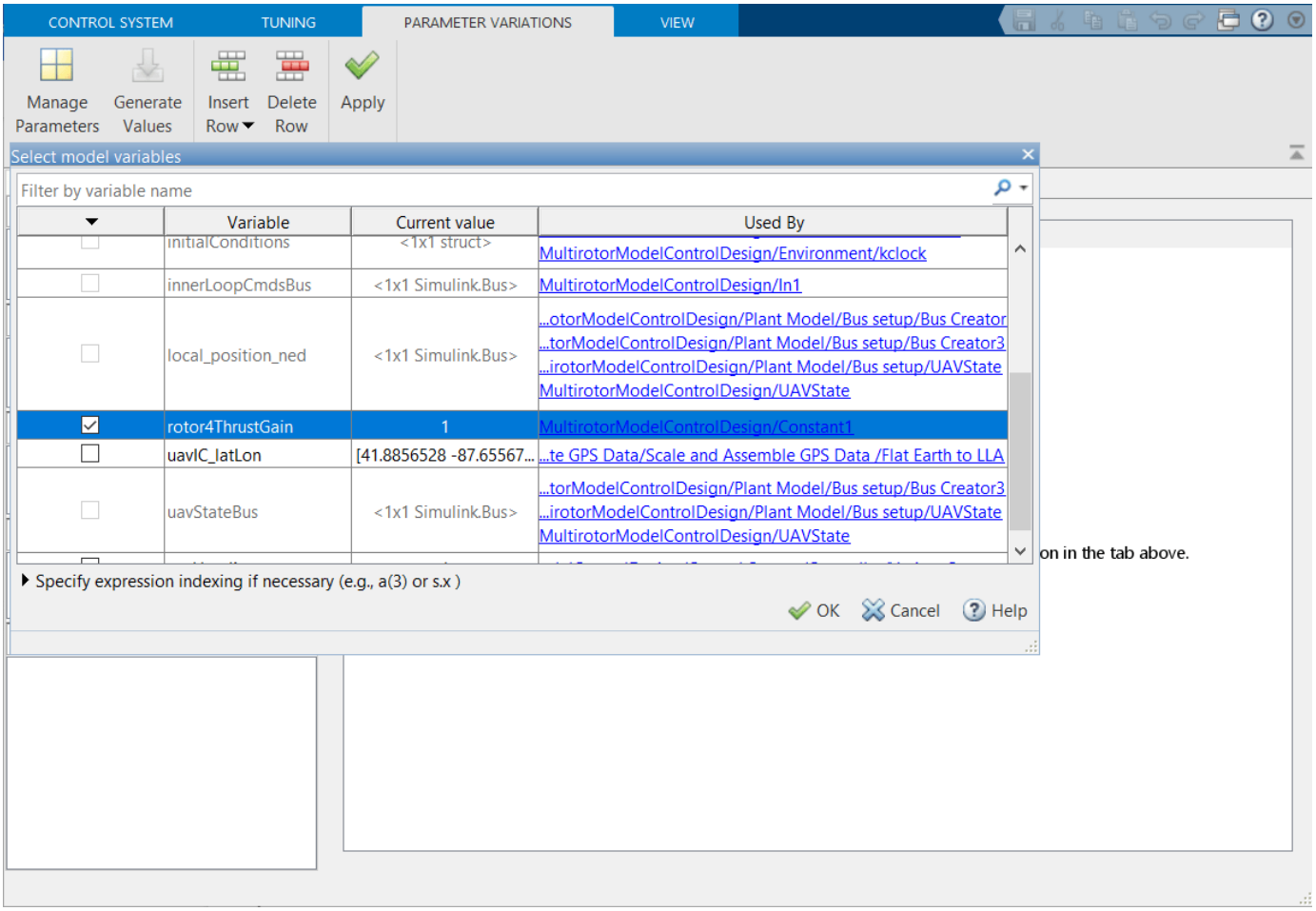
To launch the **Control System Tuner** app, in the Simulink model window, in the **Apps** gallery, click **Control System Tuner.**

Specify to linearize the model at a sample time of 0.005 seconds. To do so, click **Linearization Options**, select **Discrete with sample time**, and enter `0.005`.

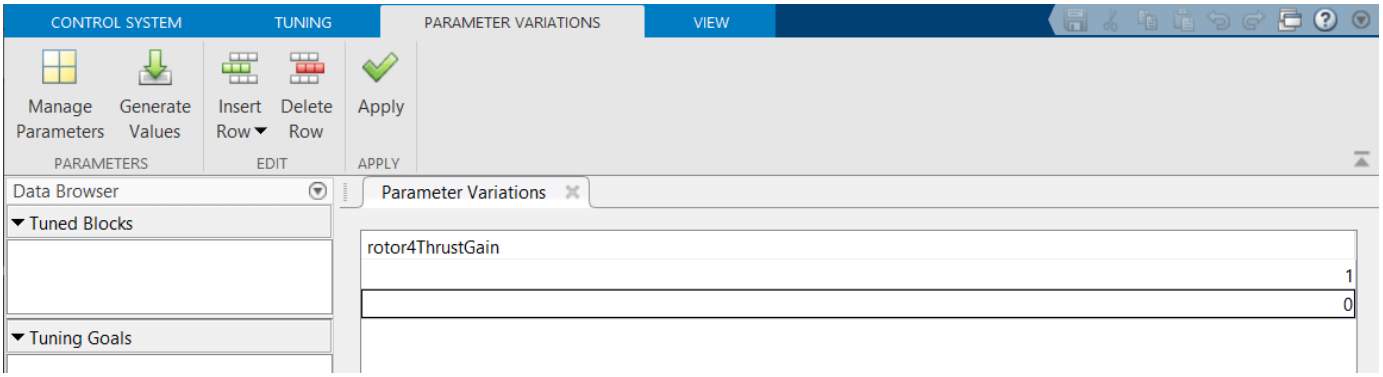


Parameter Variations for Multiple Models

To specify parameter variations, on the **Control System** tab, click `Select parameters to vary` from the **Parameter Variations** list. To select model variables, on the **Parameter Variations** tab, click **Manage Parameters.** Select the `rotor4ThrustGain` parameter and click **OK** to add it to the **Parameter Variations** table.

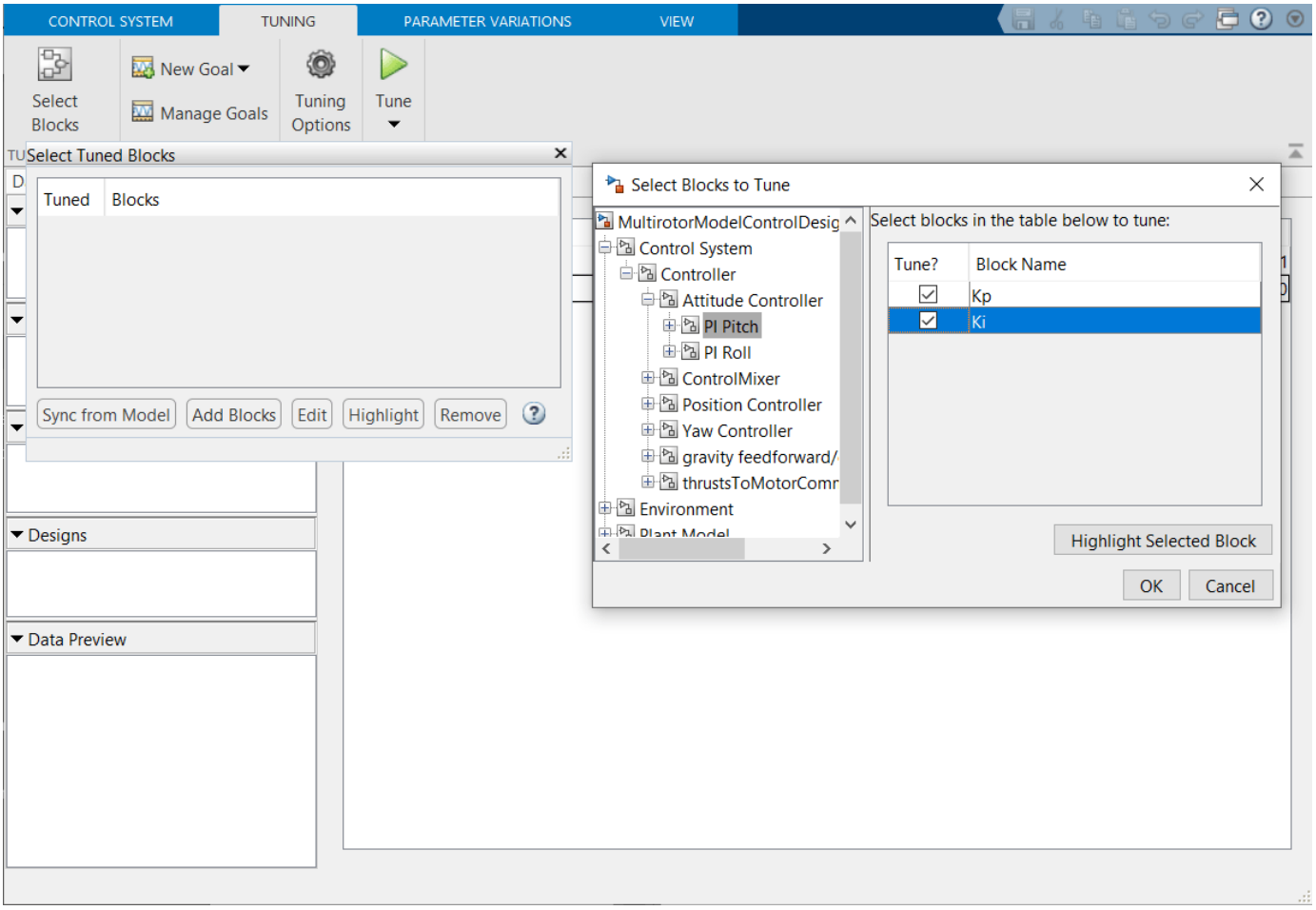


Enter `0` for the second row value. The parameter variation defines the nominal flight (`rotor4ThrustGain` = 1) and single rotor failure (`rotor4ThrustGain` = 0) conditions.

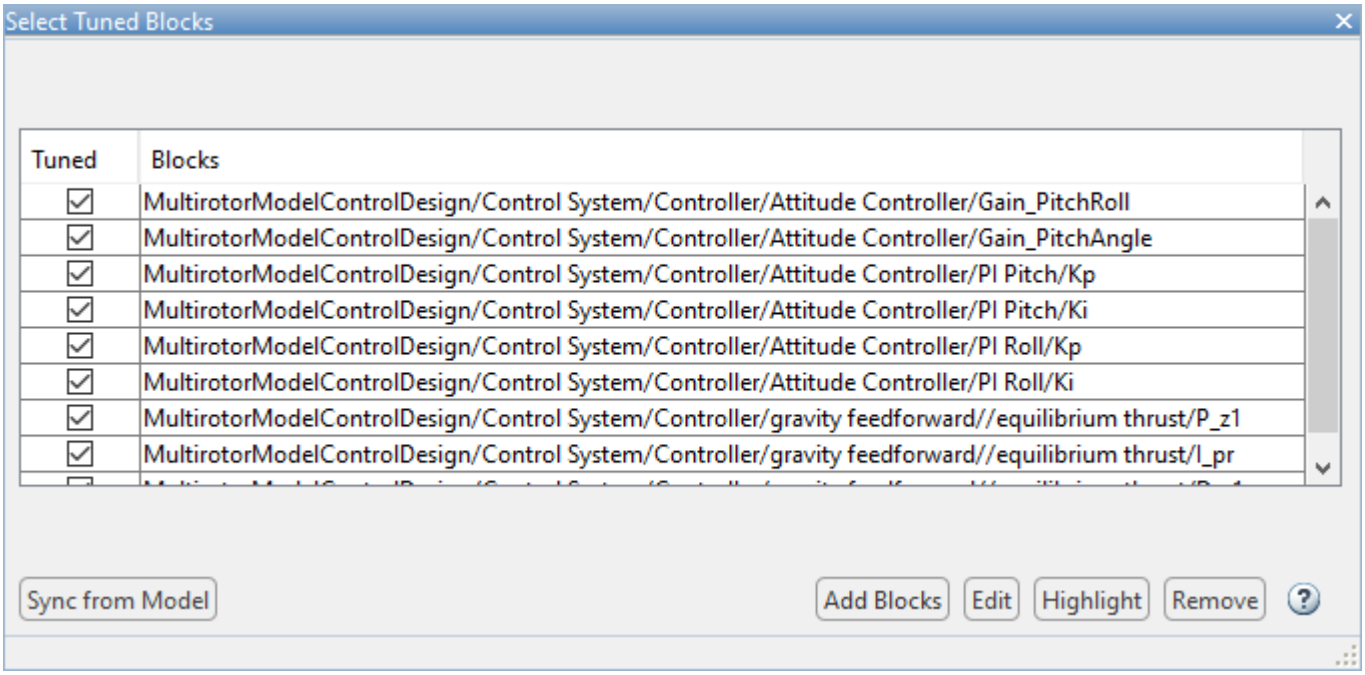


Select Tunable Blocks

To select blocks for tuning, on the **Tuning** tab, click **Select Blocks**. Then, on the Select Tuned Blocks dialog, click **Add Blocks.** This opens the editor for tuned blocks where you can specify which blocks are tunable.

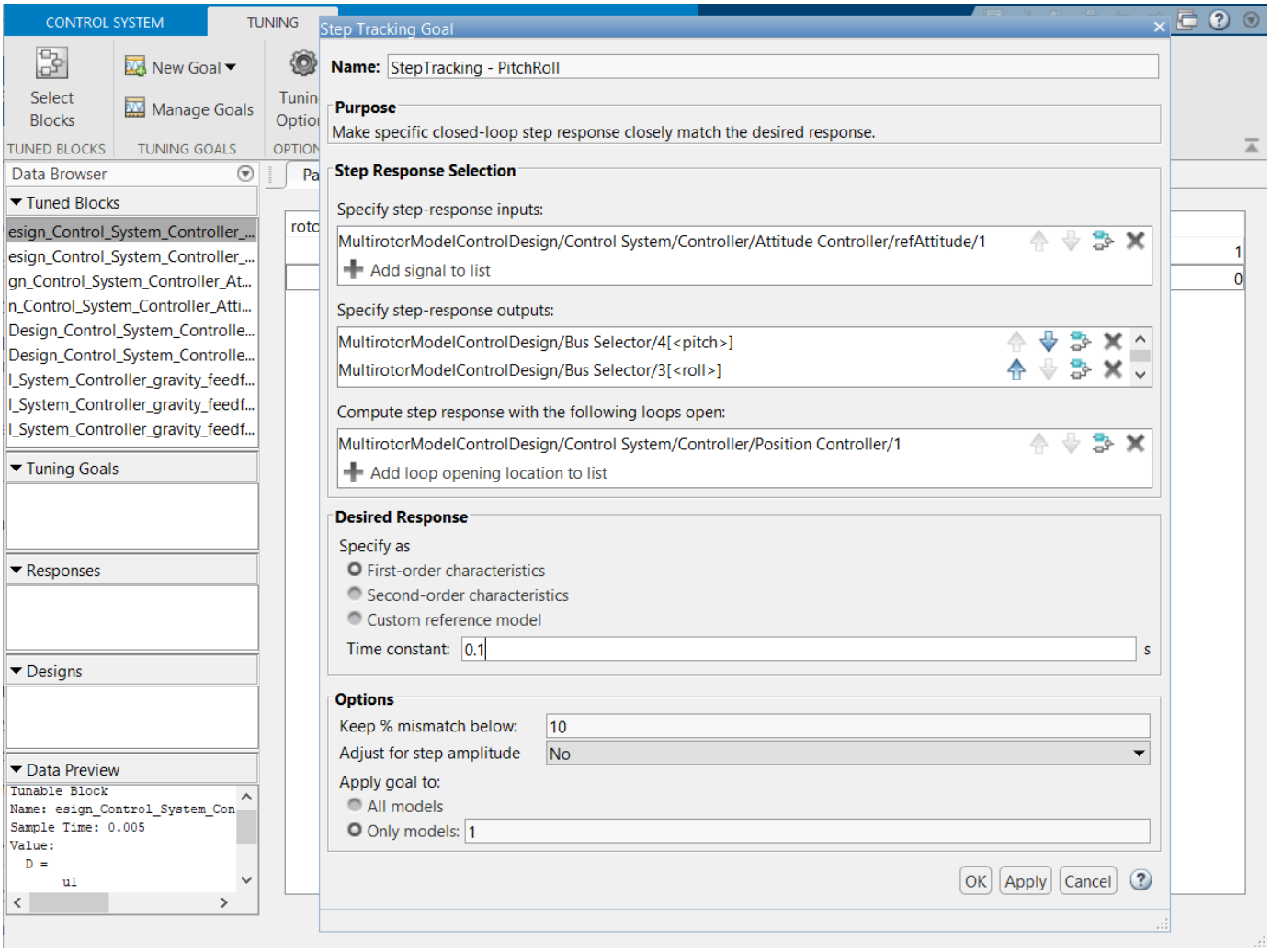


Select the controller gains in the attitude controller and altitude controller subsystems as tunable.



Specify Tuning Goals

To specify a tuning goal, on the **Tuning** tab, click **New Goal**. The Step Tracking Goal is used to specify the response and desired characteristics for controlling the pitch and roll angles. The reference pitch and roll signals and measured signals in the **UAVState** signal are marked as inputs and outputs for the tuning goal, respectively. The position controller loop is opened to tune the attitude control loop in isolation. Enter the **Time constant** parameter as 0.1 s and apply the tuning goal to the nominal model (first row of **Parameter Variation**).



Similarly, use the Step Tracking Goal and Loop Shaping Goal dialogs to create the full set of tuning goals required to tune the attitude and altitude controller gains.

The control objectives for the attitude loop are:

- Step tracking tuning goal to control pitch and roll with a desired first order response with a time constant of 0.1 seconds
- Loop shape goal to suppress gain of feedback of pitch and roll loops at high frequency, specified as an integrator with bandwidth of 10 Hz

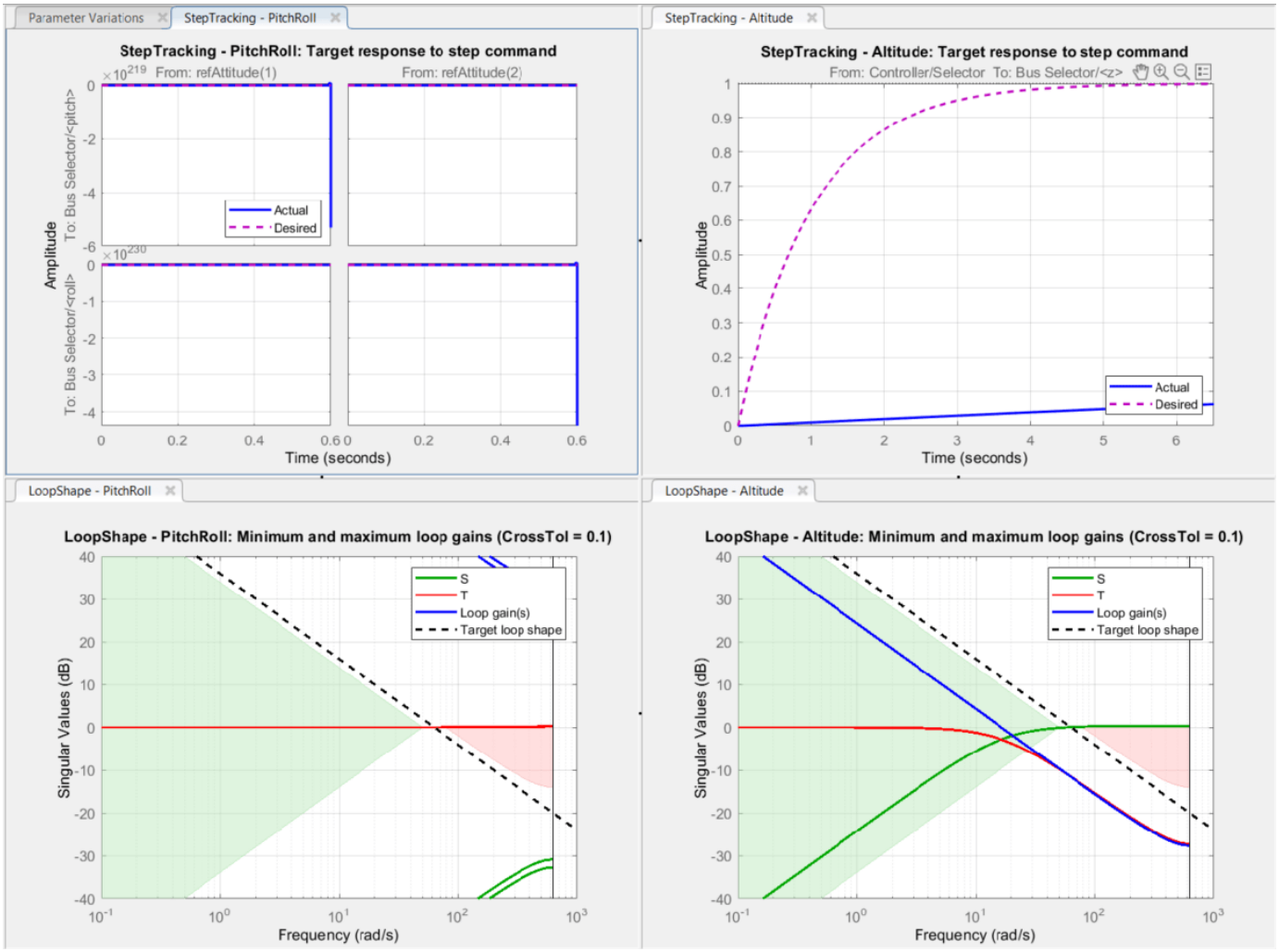
The control objectives for the altitude loop are:

- Step tracking tuning goal with a desired first order response with a time constant of 1 second
- Loop shape goal to suppress gain of feedback at high frequency, specified as an integrator with bandwidth of 10 Hz

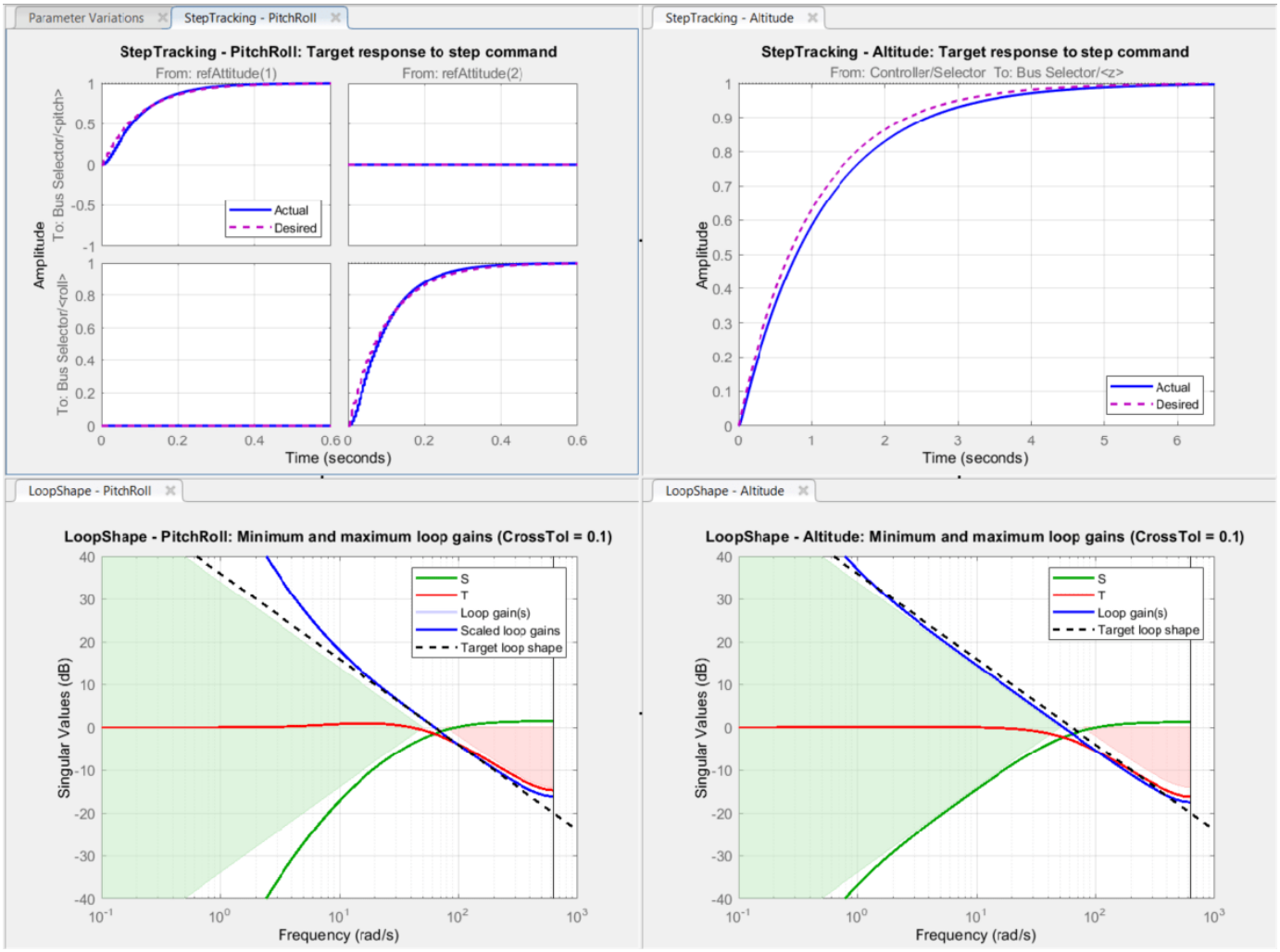
Tuning Controller Parameters for Nominal Model

Follow the preceding section to set up the **Control System Tuner** session for tuning controller gains for nominal flight mode. Alternatively, use the shortcut **Tune controller for nominal flight** in the project to launch **Control System Tuner** with a preconfigured session file.

The tuning goal plots show that with the untuned controller gains, the attitude control loop is unstable and the altitude control loop does not have the desired response. Use the **Manage Goals** option to select and edit a tuning goal.



Click **Tune** to adjust the values of the tunable blocks to achieve the tuning goals. Both the attitude and altitude control loops are tuned together with the yaw control loop fixed.



To see the values of tuned controller gains, select a block in the **Tuned Blocks** and view the value in the **Data Preview** area of the **Control System Tuner**. For more information, see “Examine Tuned Controller Parameters in Control System Tuner” on page 10-155.

The tuned controller gains are:

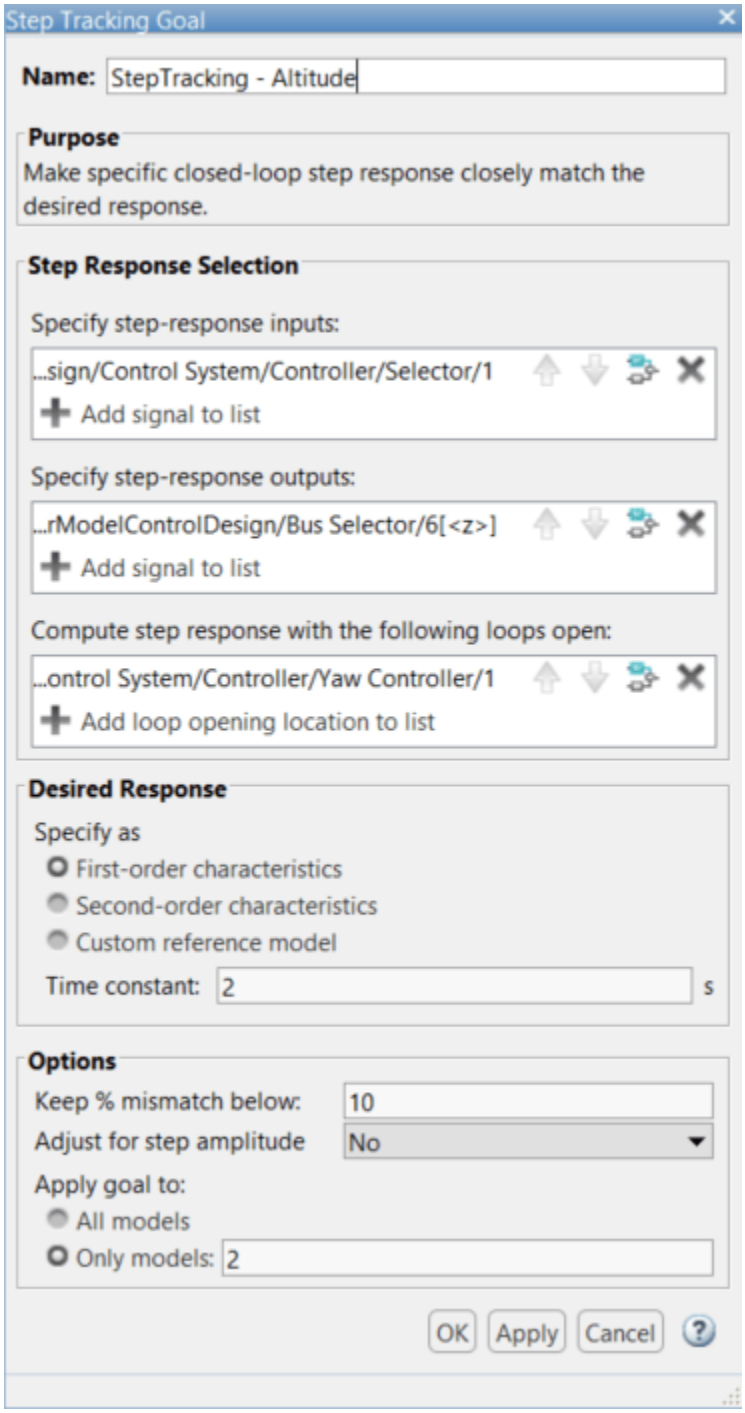
- Outer proportional loop control for pitch — $K_p = 9.669$
- Inner PI control for pitch angular velocity — $K_p = 0.004296, K_i = 0.01$
- Outer proportional loop control for roll — $K_p = 9.572$
- Inner PI control for roll angular velocity — $K_p = 0.003493, K_i = 0.01$
- PID control for altitude — $K_p = 3.004, K_i = 0.01, K_d = 3.308$

Tune Parameters for Fault Model

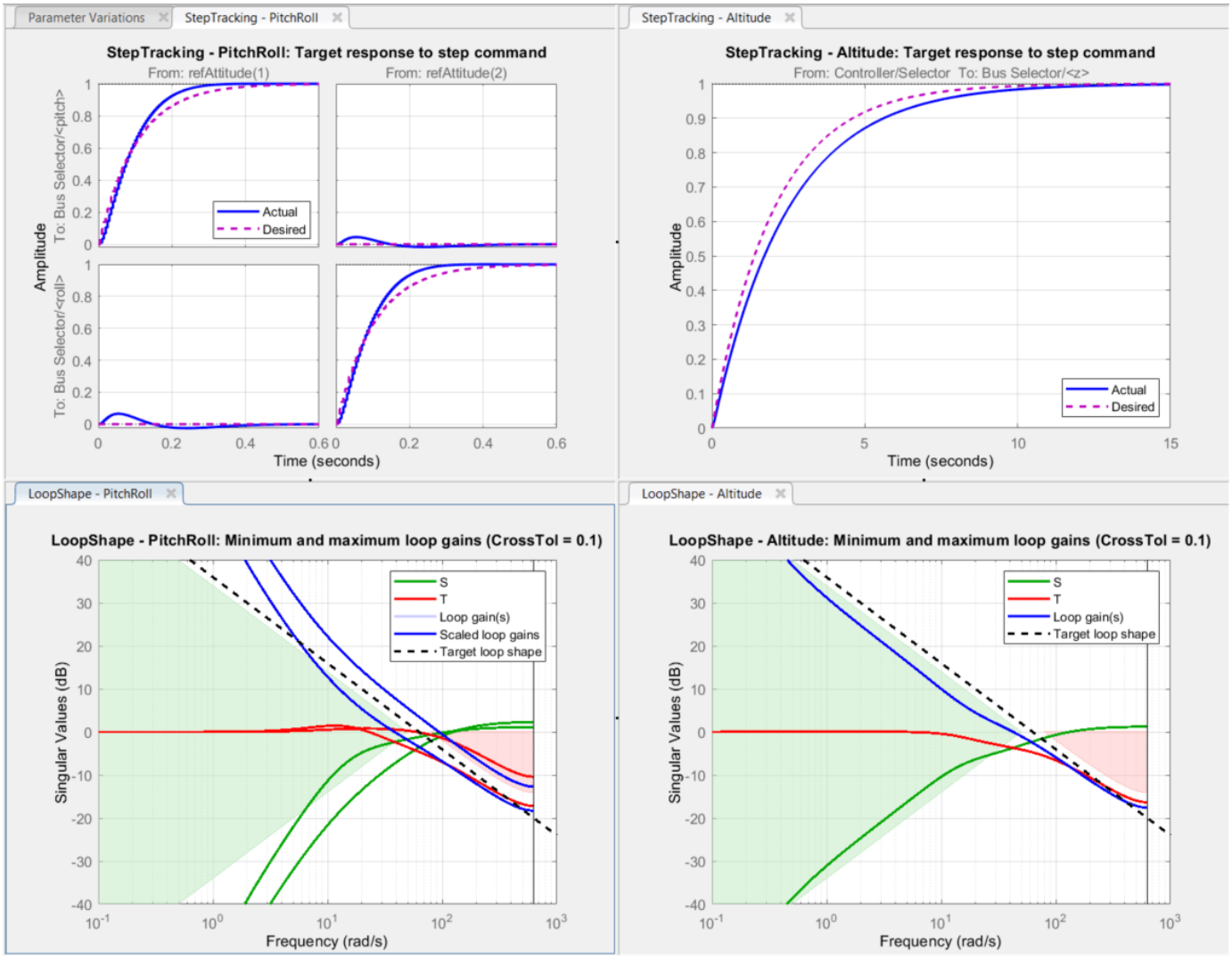
The defined parameter variation, `rotor4ThrustGain` = 0, generates the model for single rotor failure. Change the tuning goals to apply to the fault model instead of the nominal model. The yaw control loop is set to open for all tuning goals because with the loss of thrust to one rotor the diagonal rotor pairs are unbalanced and the yaw is uncontrolled. The desired time constant for the

`StepTracking - Altitude` tuning goal is modified to 2 seconds, which reduces the landing velocity of the multicopter.

Double-click to open and modify each objective under **Tuning Goals** in **Data Browser**. Alternatively, use the shortcut **Tune controller for rotor failure** in the project to launch **Control System Tuner** with a preconfigured session file.



Click **Tune** to retune the parameters based on the fault model. As seen in the following plots, small overshoots and oscillations exist in the step response of pitch and roll as a result of the rotor failure.



The tuned controller gains for a plant with a single rotor failure are:

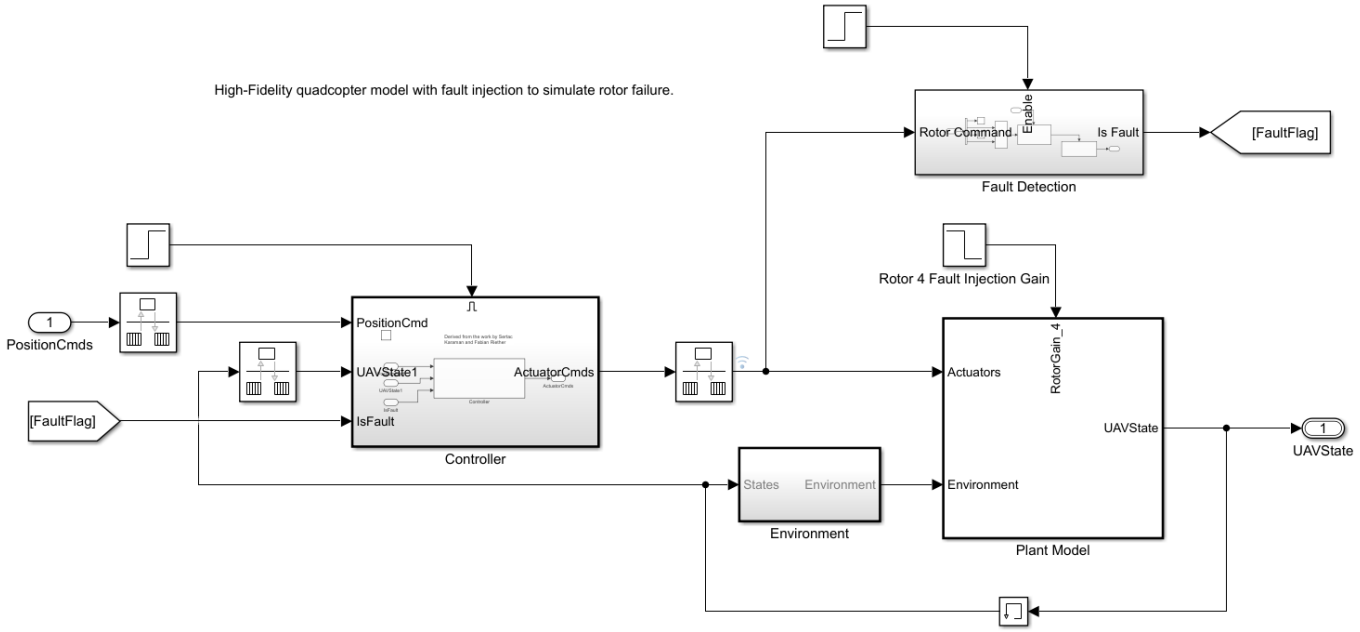
- Outer proportional loop control for pitch — $K_p = 11.02$
- Inner PI control for pitch angular velocity — $K_p = 0.006416, K_i = 0.01$
- Outer proportional loop control for roll — $K_p = 11.42$
- Inner PI control for roll angular velocity — $K_p = 0.005209, K_i = 0.01$
- PID control for altitude — $K_p = 1.667, K_i = 0.01, K_d = 4.098$

Simulation with Injected Fault and Gain-Scheduled PID Controllers

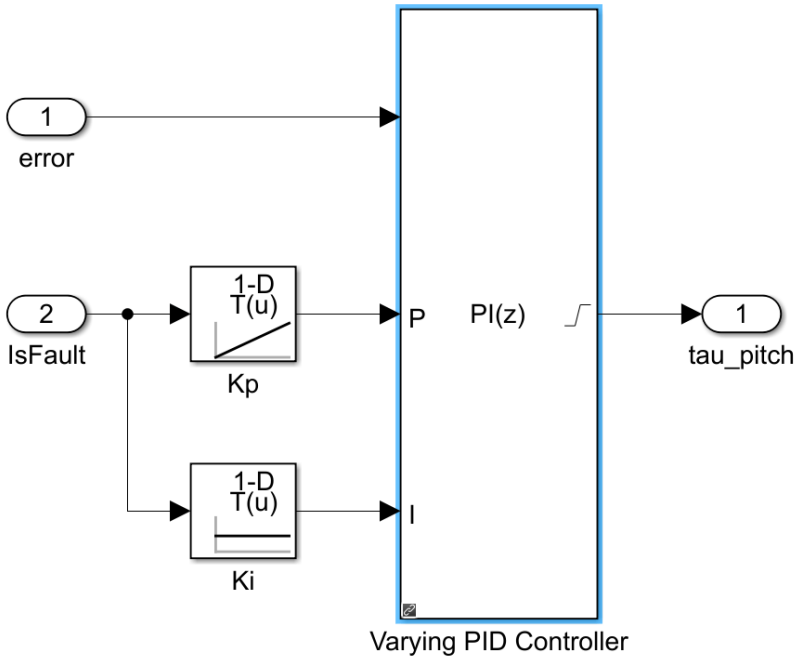
The simulation model implements a fault detection subsystem that extracts features from the actuator commands and sets a threshold to detect the fault in rotor 4. The fault detection indicator is used as the scheduling variable for the gain-scheduled controllers in the attitude loop and altitude loop. Also,

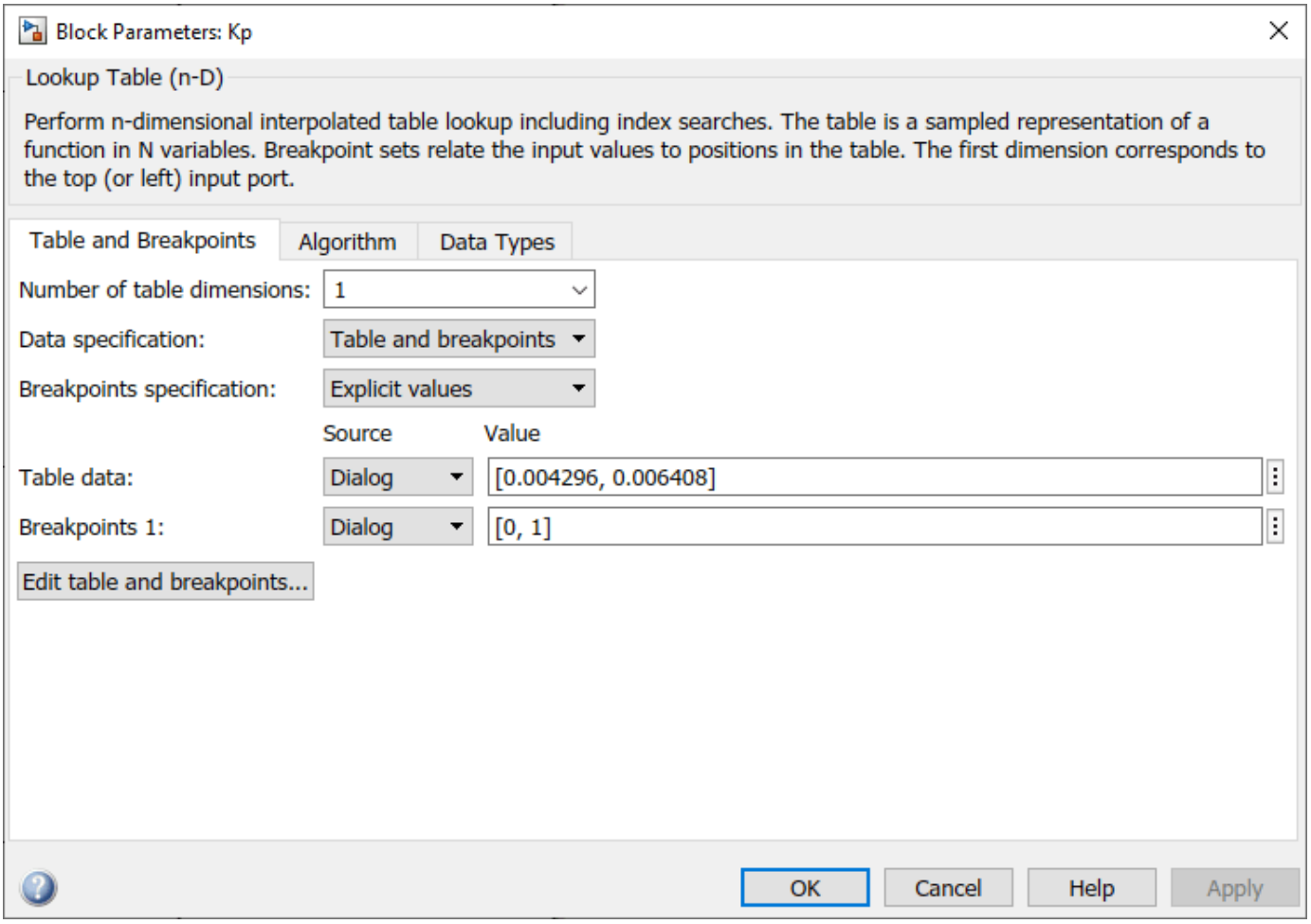
the pitch, roll, and altitude reference inputs are reconfigured to command the multicopter to land with a safe velocity.

```
load_system("MultirotorModel");
open_system("MultirotorModel/Inner Loop and Plant Model/High-FidelityModel_RotorFault");
```



The gain-scheduled controllers are implemented using the Varying PID Controller block and Lookup Table blocks to specify the gains, as shown for the pitch angular velocity controller. Update the gains computed in the previous section in the table data of the blocks defined at breakpoints of nominal operation (0) and fault (1).


1
error
2
IsFault
1-D
T(u)
Kp
1-D
T(u)
Ki
P
I
PI(z)
1
tau_pitch
Varying PID Controller



Similarly, update the gains for the remaining controllers.

Alternatively, you can set the block parameters using the following commands. If you modify any tuning goal, replace the provided values with your tuned controller gain values.

Set the pitch controller parameters as follows.

```
set_param(['MultirotorModel/Inner Loop and Plant Model/High-FidelityModel_RotorFault/' ...
    'Controller/Controller/Attitude Controller/Gain_PitchAngle'],'TableData','[9.669, 11.02]');

set_param(['MultirotorModel/Inner Loop and Plant Model/High-FidelityModel_RotorFault/' ...
    'Controller/Controller/Attitude Controller/PI Pitch/Kp'],'TableData','[0.004296, 0.006416]');

set_param(['MultirotorModel/Inner Loop and Plant Model/High-FidelityModel_RotorFault/' ...
    'Controller/Controller/Attitude Controller/PI Pitch/Ki'],'TableData','[0.01, 0.01]');
```

Set the roll controller parameters as follows.

```
set_param(['MultirotorModel/Inner Loop and Plant Model/High-FidelityModel_RotorFault/' ...
    'Controller/Controller/Attitude Controller/Gain_RollAngle'],'TableData','[9.572, 11.42]');

set_param(['MultirotorModel/Inner Loop and Plant Model/High-FidelityModel_RotorFault/' ...
    'Controller/Controller/Attitude Controller/PI Roll/Kp'],'TableData','[0.003493, 0.005209]');
```

```
set_param(['MultirotorModel/Inner Loop and Plant Model/High-FidelityModel_RotorFault/' ...
    'Controller/Controller/Attitude Controller/PI Roll/Ki'],'TableData','[0.01, 0.01]');
```

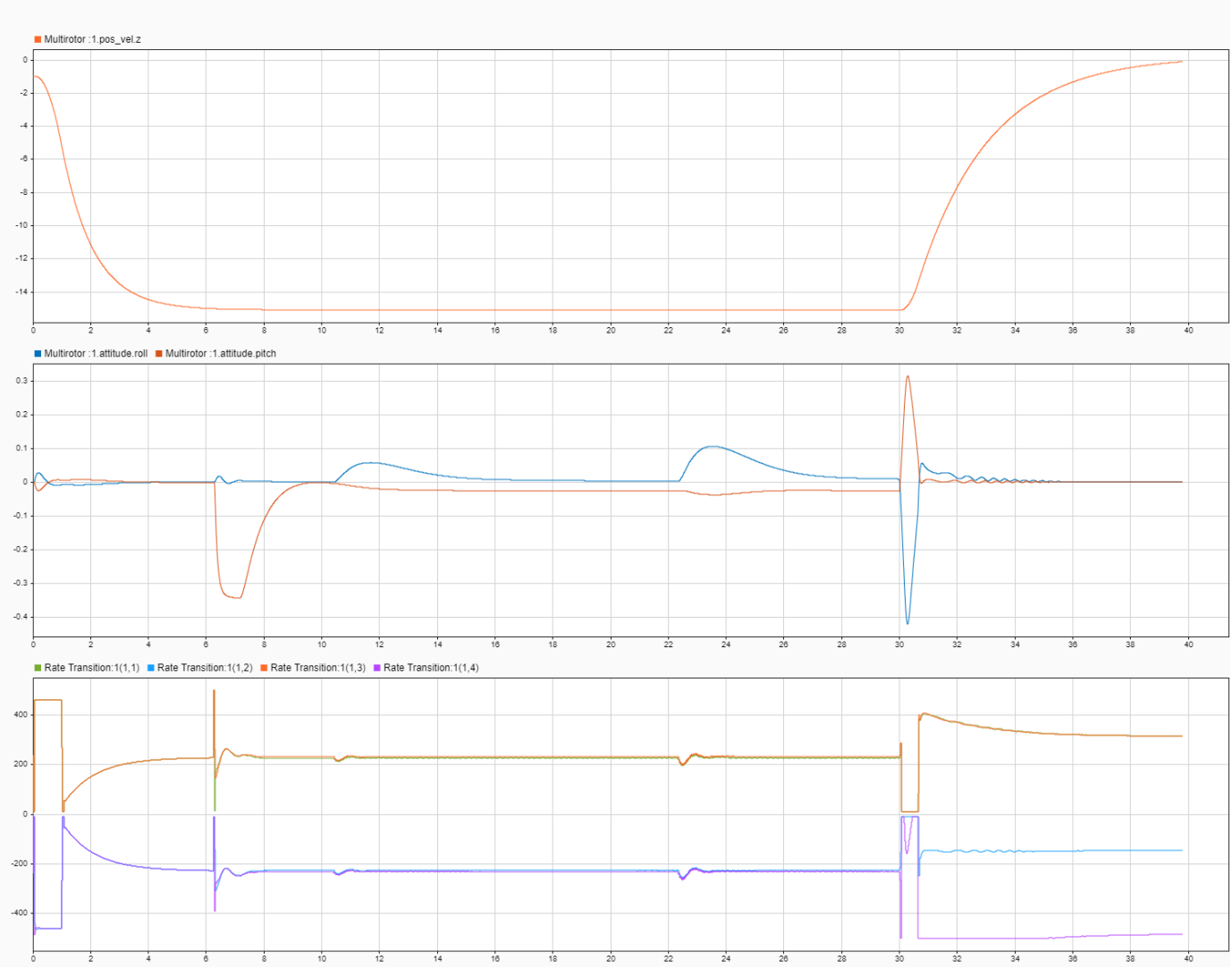
Set the altitude controller parameters as follows.

```
set_param(['MultirotorModel/Inner Loop and Plant Model/High-FidelityModel_RotorFault/' ...
    'Controller/Controller/gravity feedforward//equilibrium thrust/Kp'],'TableData','[3.004, 1.60

set_param(['MultirotorModel/Inner Loop and Plant Model/High-FidelityModel_RotorFault/' ...
    'Controller/Controller/gravity feedforward//equilibrium thrust/Ki'],'TableData','[0.01, 0.01]

set_param(['MultirotorModel/Inner Loop and Plant Model/High-FidelityModel_RotorFault/' ...
    'Controller/Controller/gravity feedforward//equilibrium thrust/Kd'],'TableData','[3.308, 4.09
```

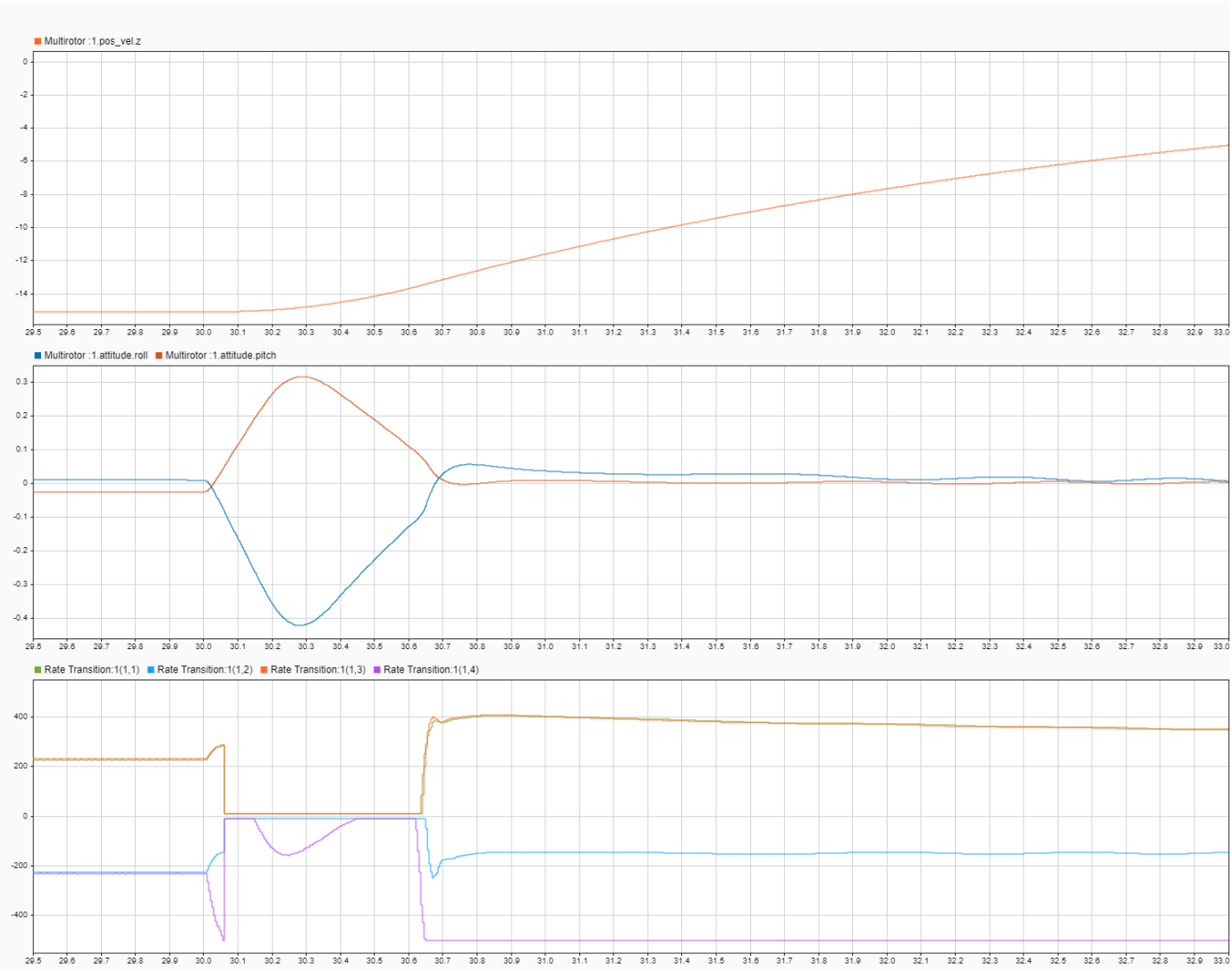
Use the **Simulate model with fault injection** project shortcut to simulate the model to takeoff and fly the multicopter based on position commands from the guidance logic subsystem. A rotor fault is introduced at 30 seconds by reducing the rotor thrust to 30%. Results for altitude position (top row) and the attitude (middle row) in the **Simulation Data Inspector** show that the UAV settles to within 5% of its desired altitude and has smooth pitch and roll to enable it to track X and Y positions. The bottom row shows the actuator commands for the four rotors.

When a rotor fails at 30 seconds, the UAV starts pitching and rolling. The controller is reconfigured as the failure is detected. The pitch and roll are controlled to settle at 0 radians and the UAV lands while maintaining a low velocity. As expected, the yaw is uncontrolled and the UAV does spin around the vertical axis. The maximum yaw rate that the UAV reaches is verified through simulation, and the attitude and altitude controllers are retuned with modified tuning goals, if necessary.

The UAV is visualized in a photorealistic environment and shows the UAV flying in a realistic world. As the simulation starts, press `'F'` to set the camera mode to `Free` in the `AutoVrtlEnv` window and use the mouse scroll wheel to increase the camera distance from the UAV. Using these controls, you can visualize the landing sequence following the rotor failure.



The following plot shows the transients after occurrence of the fault at 30 seconds.



Close the project.

```
close(prj);
```

See Also

Control System Tuner

Related Examples

- “Specify Goals for Interactive Tuning” on page 10-29
- “View and Change Block Parameterization in Control System Tuner” on page 10-19
- “Tune a Control System Using Control System Tuner” on page 13-59
- “PID Autotuning for UAV Quadcopter” on page 8-75

Adaptive Control

- “Extremum Seeking Control” on page 15-2
- “Extremum Seeking Control for Reference Model Tracking of Uncertain Systems” on page 15-8
- “Anti-Lock Braking Using Extremum Seeking Control” on page 15-15
- “Adaptive Cruise Control Using Extremum Seeking Control” on page 15-21
- “Model Reference Adaptive Control” on page 15-28
- “Model Reference Adaptive Control of Satellite Spin” on page 15-33
- “Model Reference Adaptive Control of Aircraft Undergoing Wing Rock ” on page 15-41

Extremum Seeking Control

Extremum seeking control (ESC) is a model-free, real-time adaptive control algorithm that is useful for adapting parameters to unknown system dynamics and unknown mappings from control parameters to an objective function. You can use extremum seeking to solve static optimization problems and to optimize parameters of dynamic systems.

The extremum seeking algorithm uses the following stages to tune a parameter value.

1. Modulation — Perturb the value of the parameter being optimized using a low-amplitude sinusoidal signal.
2. System Response — The system being optimized reacts to the parameter perturbations. This reaction causes a corresponding change in the objective function value.
3. Demodulation — Multiply the objective function signal by a sinusoid with the same frequency as the modulation signal. This stage includes an optional high-pass filter to remove bias from the objective function signal.
4. Parameter Update — Update the parameter value by integrating the demodulated signal. The parameter value corresponds to the state of the integrator. This stage includes an optional low-pass filter to remove high-frequency noise from the demodulated signal.

Simulink Control Design software implements this algorithm using the Extremum Seeking Control block. For examples of extremum-seeking control, see:

- “Extremum Seeking Control for Reference Model Tracking of Uncertain Systems” on page 15-8
- “Anti-Lock Braking Using Extremum Seeking Control” on page 15-15
- “Adaptive Cruise Control Using Extremum Seeking Control” on page 15-21

Time Domain

Using the Extremum Seeking Control block, you can implement both continuous-time and discrete-time controllers. The ESC algorithm is the same in both cases. Changing the time-domain of the controller affects the time domain of the high-pass filters, low-pass filters, and integrators used in the tuning loops.

To generate hardware-deployable code for the Extremum Seeking Control block, use a discrete-time controller.

The following table shows the continuous-time and discrete-time transfer functions for the filters and integrators in the Extremum Seeking Control block.

Controller Element	Continuous-Time Transfer Function	Discrete-Time Transfer Function
High-pass filter	$\frac{s}{s+\omega_h}$	$\frac{1-z^{-1}}{1-\omega_h z^{-1}}$
Low-pass Filter	$\frac{1}{s+\omega_l}$	$\frac{1}{1-\omega_l z^{-1}}$

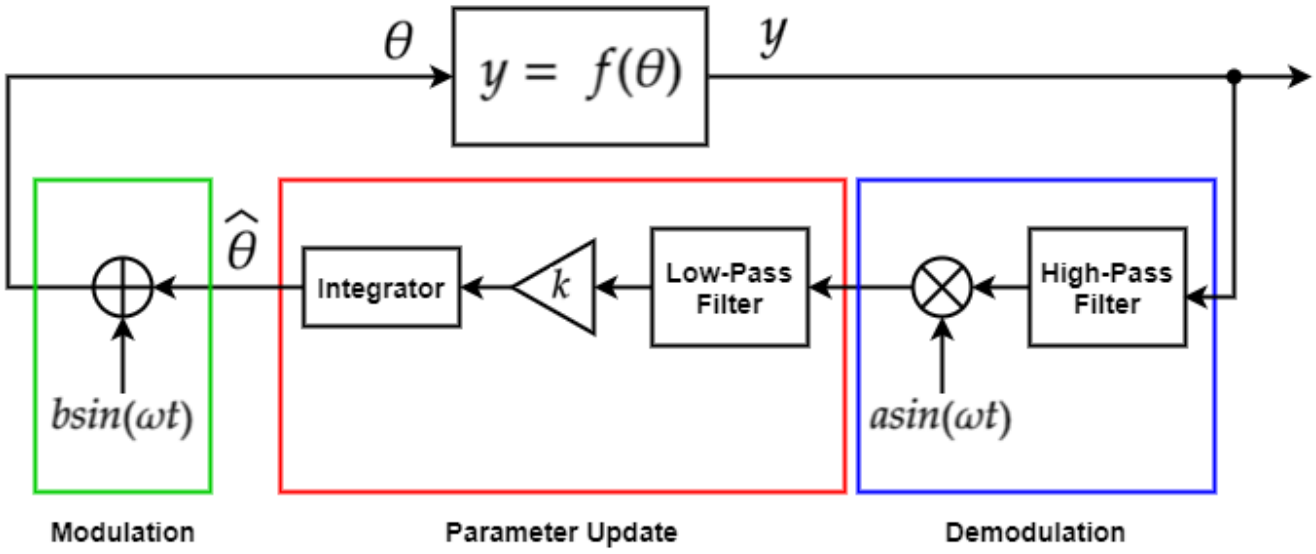
Controller Element	Continuous-Time Transfer Function	Discrete-Time Transfer Function
Integrator	$\frac{1}{s}$	Forward Euler: $\frac{T_s}{z-1}$ Backward Euler: $\frac{T_s z}{z-1}$ Trapezoidal: $\frac{T_s}{2}\frac{z+1}{z-1}$

Here:

- ω_l is the low-pass filter cutoff frequency.
- ω_h is the high-pass filter cutoff frequency.
- T_s is the sample time of the discrete-time controller learning rate.

Static Optimization

To demonstrate extremum seeking, consider the following static optimization problem.



Here:

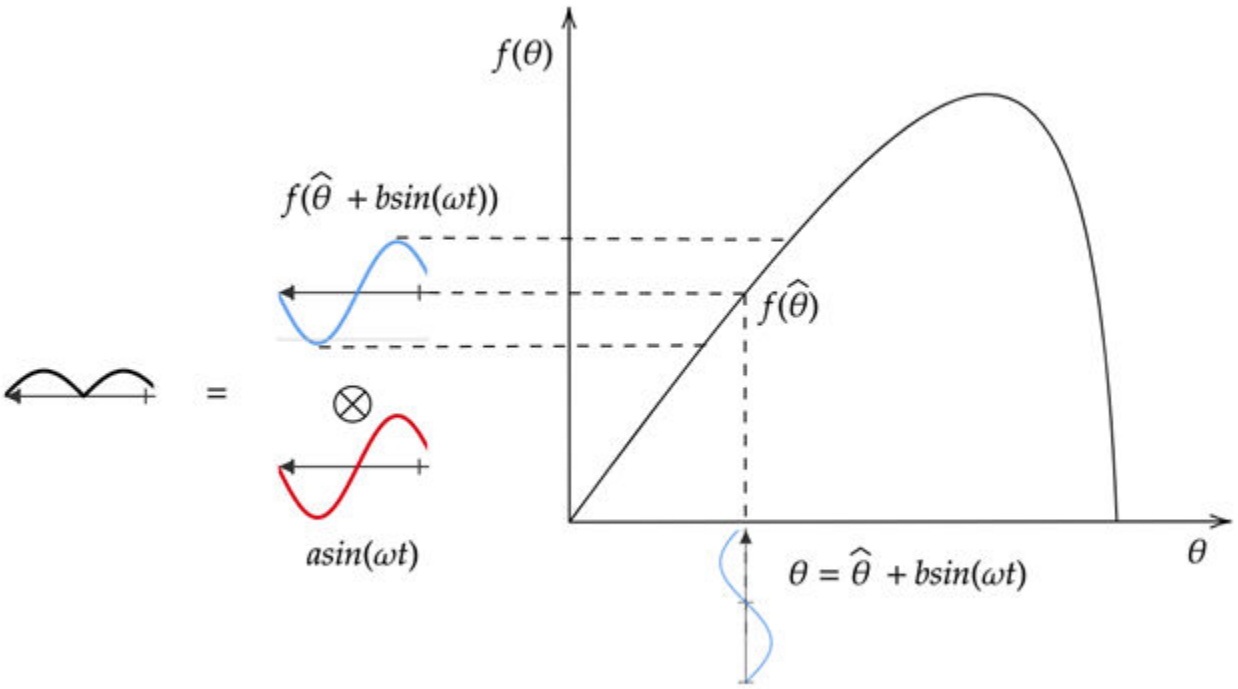
- $\widehat{\theta}$ is the estimated parameter value.
- θ is the modulation signal

- $y = f(\theta)$ is the function output being maximized, that is, the objective function.
- ω is the forcing frequency of the modulation and demodulation signals.
- $b \cdot \sin(\omega t)$ is the modulation signal.
- $a \cdot \sin(\omega t)$ is the demodulation signal.
- k is the learning rate.

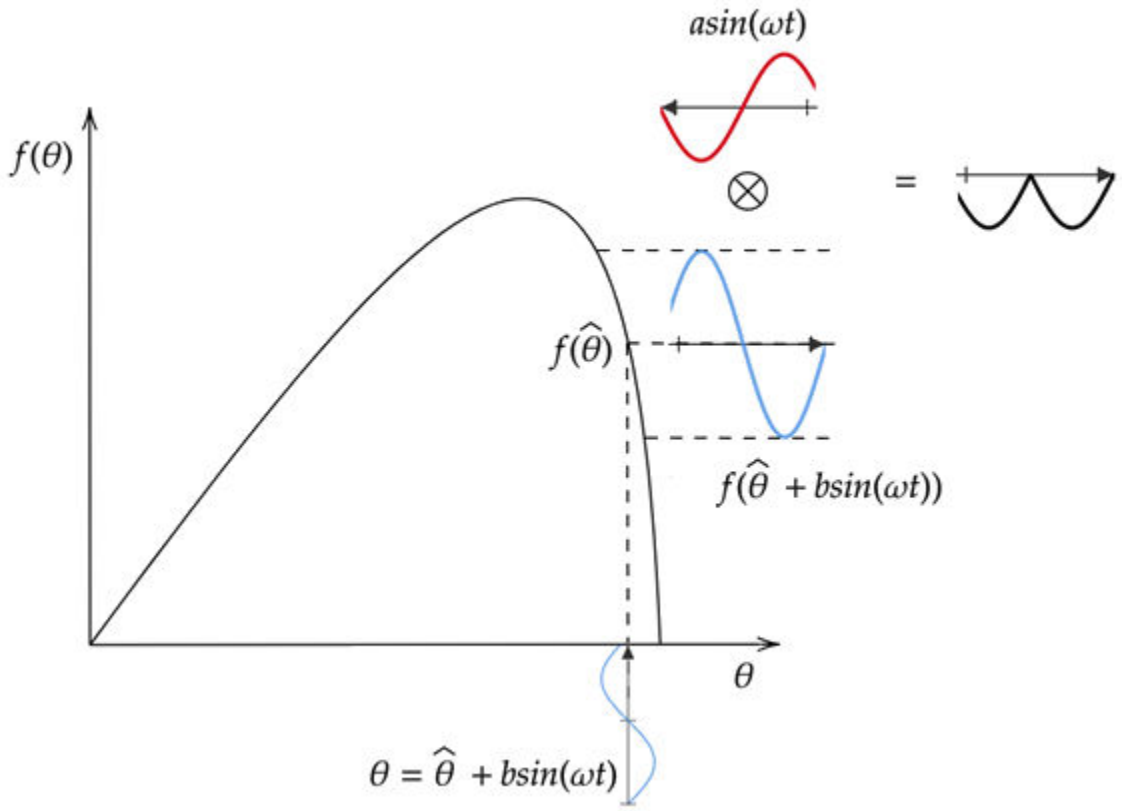
The optimum parameter value, θ^*, occurs at the maximum value of $f(\theta)$.

To optimize multiple parameters, you use a separate tuning loop for each parameter.

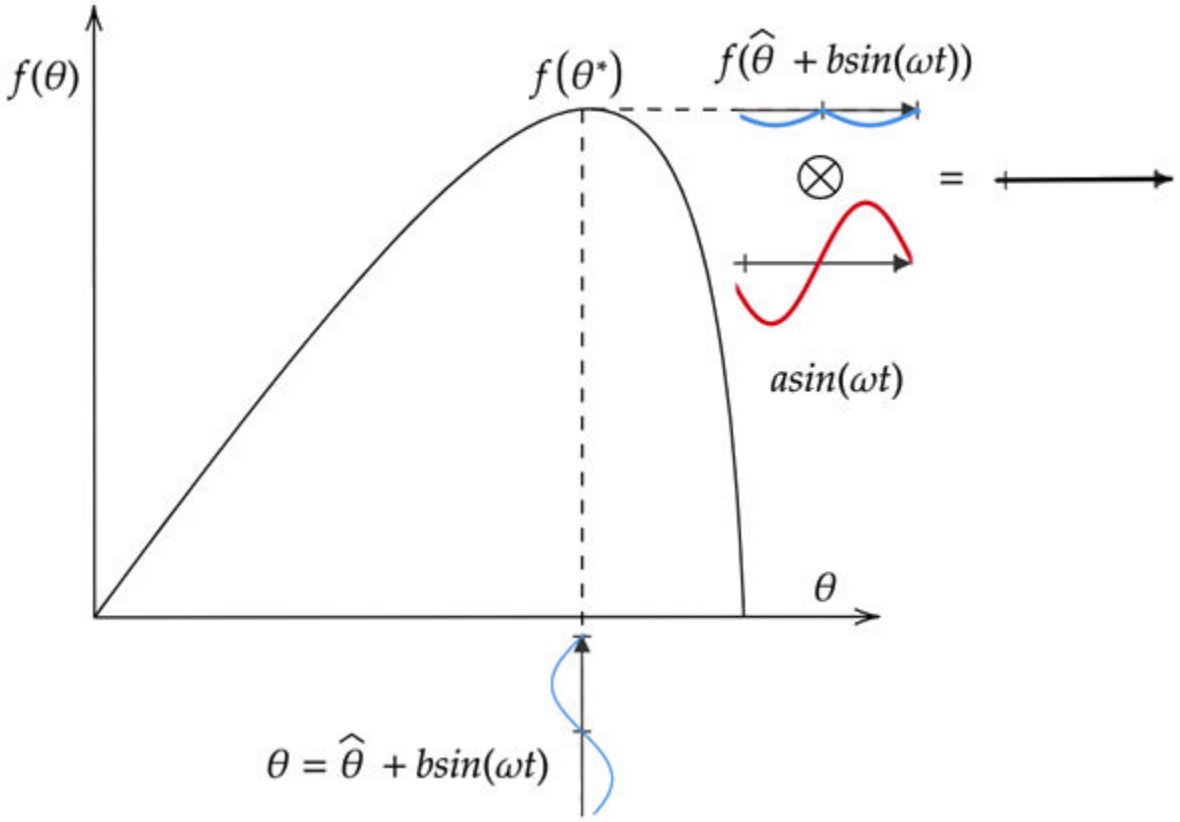
The following figure demonstrates extremum seeking for an *increasing portion* of the objective function curve. The modulated signal θ is the sum of the current estimated parameter and the modulation signal. Applying $f(\theta)$ produces a perturbed objective function with the same phase as the modulation signal. Multiplying the perturbed objective function by the demodulation signal produces a positive signal. Integrating this signal increases the value of θ, which moves it closer to the peak of the objective function.



The following figure demonstrates extremum seeking for a *decreasing portion* of the objective function curve. In this case, applying $f(\theta)$ produces a perturbed objective function that is 180 degrees out of phase from the modulation signal. Multiplying by the demodulation signal produces a negative signal. Integrating this signal decreases the value of θ, which moves it closer to the peak of the objective function.

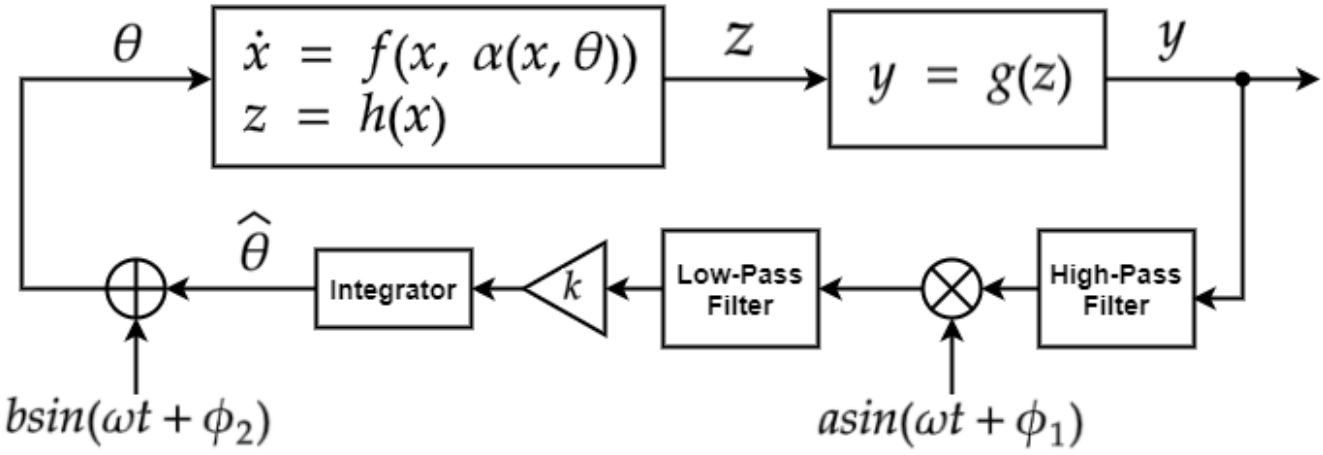


The following figure demonstrates extremum seeking for a *flat portion* of the objective function curve, that is, a portion of the curve near the maximum. In this case, applying $f(\theta)$ produces a near-zero perturbed objective function. Multiplying by the demodulation signal and integrating this signal does not significantly change the value of θ, which is already near its optimum value θ^*.



Dynamic System Optimization

Extremum seeking optimization of a dynamic system occurs in a similar fashion as static optimization. However, in this case, the parameter θ affects the output of a time-dependent dynamic system. The objective function to be maximized is computed from the system output. The following figure shows the general tuning loop for a dynamic system.



Here:

- $\dot{x} = f(x, \alpha(x, \theta))$ is the state function of the dynamic system.
- $z = h(x)$ is the output of the dynamic system.
- $y = g(z)$ is the objective function derived from the output of the dynamic system.
- ϕ_1 is the phase of the demodulation signal.
- ϕ_2 is the phase of the modulation signal.

ESC Design Guidelines

When designing an extremum-seeking controller, consider the following guidelines.

- Ensure that the system dynamics are on the fastest time scale, the forcing frequencies are on the medium time scale, and the filter cutoff frequencies are on the slowest time scale.
- Specify an amplitude for the demodulation signal that is much greater than the modulation signal amplitude ($a \gg b$).
- Select phase angles for the modulation and demodulation signals such that $\cos(\phi_1 - \phi_2) > 0$.
- When tuning multiple parameters, the forcing frequency for each tuning loop must be different.
- Try designing your system without high-pass and low-pass filters. If the performance is not satisfactory, you can then consider adding one or both filters.

References

[1] Ariyur, Kartik B., and Miroslav Krstić. *Real Time Optimization by Extremum Seeking Control*. Hoboken, NJ: Wiley Interscience, 2003.

See Also

Blocks
Extremum Seeking Control

Related Examples

- “Extremum Seeking Control for Reference Model Tracking of Uncertain Systems” on page 15-8
- “Anti-Lock Braking Using Extremum Seeking Control” on page 15-15
- “Adaptive Cruise Control Using Extremum Seeking Control” on page 15-21

Extremum Seeking Control for Reference Model Tracking of Uncertain Systems

This example shows the design of feedback and feedforward gains for a system, a common controller design technique. Here, you use an extremum seeking controller to track a given reference plant model by adapting feedback and feedforward gains for an uncertain dynamical system.

Adaptive Gain Tuning for Uncertain Linear Systems

For this example, consider the following first-order linear system.

$$\dot{x}(t) = a_0 x(t) + b_0 u(t)$$

Here, $x(t)$ and $u(t)$ are the state and control input of the system, respectively. The constants a_0 and b_0 are unknown.

The goal of this example is to track the performance of the following reference plant model, which defines the required transient and steady-state behavior.

$$\dot{x}_{ref}(t) = a^* x_{ref}(t) + b^* r(t)$$

Here, $x_{\mathrm{ref}}(t)$ is the state of the reference plant and $r(t)$ is the reference signal.

The aim of the control signal $u(t)$ is to make the states $x(t)$of the uncertain system track the reference states $x_{\mathrm{ref}}(t)$.

$$u(t) = Kx(t) - Kr(t)$$

The designed controller contains a feedback term, $Kx(t)$, and a feedforward term, $-Kr(t)$.

Substitute this control signal into the unknown linear system dynamics.

$$\dot{x}(t) = a_0 x(t) + b_0(Kx(t) - Kr(t))$$

You can rewrite this expression as shown in the following equation.

$$\dot{x}(t) = (a_0 + b_0 K)x(t) - b_0 K r(t)$$

In the ideal case, if the coefficients a_0 and b_0 of the nominal system dynamics are known, then you can determine the controller gain K using pole-placement techniques. Doing so produces the following matching condition.

$$a_0 + b_0 K = a^*, b_0 K = b^*$$

When you use a single gain value as both the feedforward and feedback gain, this matching condition might not be satisfied for all the possible values of a_0 and b_0. For a more general solution, you can tune two different gain values (multiparameter tuning).

For this example, use the following unknown system and reference dynamics.

$$\dot{x}(t) = -1x(t) + u(t)$$

$$\dot{x}_{ref}(t) = -3x_{ref}(t) + 2r(t)$$

In this case, the ideal control gain is $K = -2$.

Extremum Seeking Control Adaptive Gain Tuning

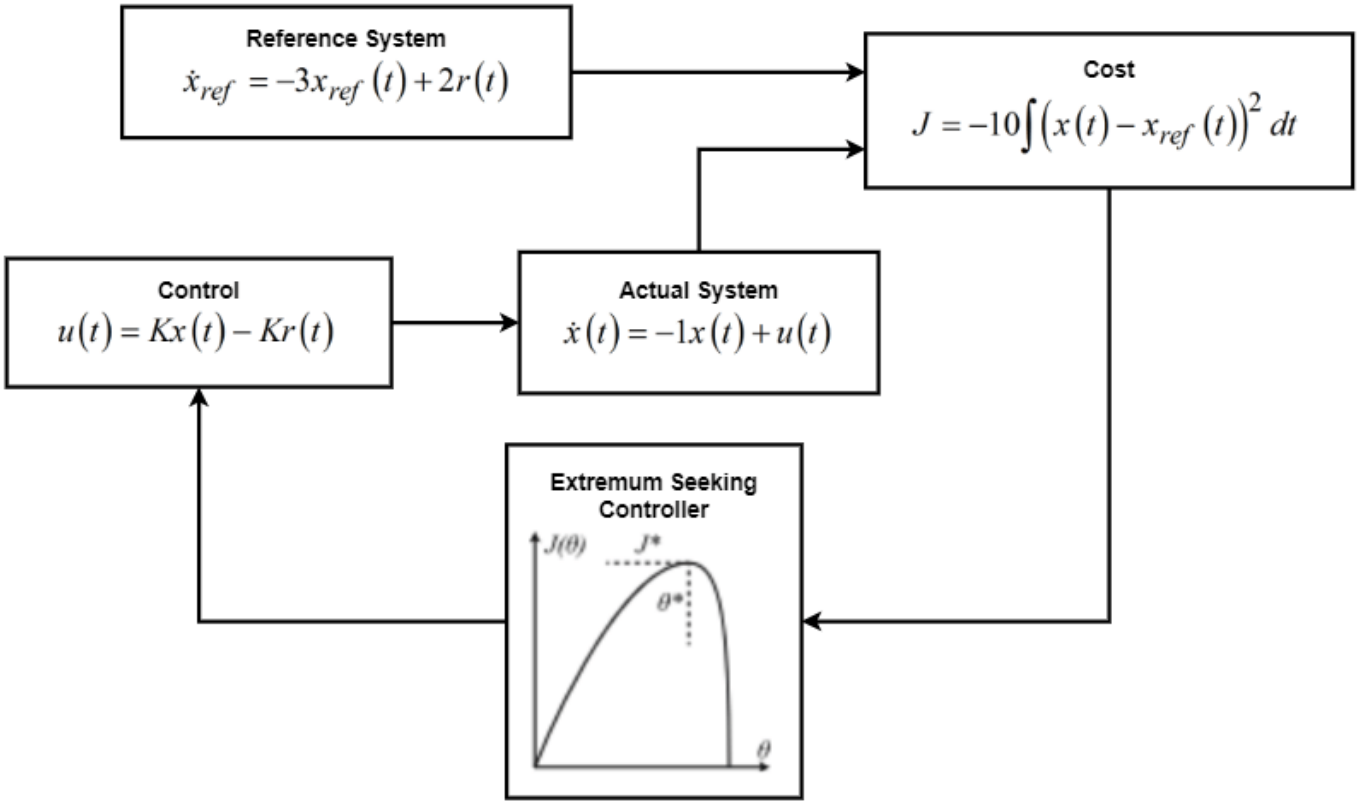
To implement an extremum seeking control (ESC) approach to the preceding problem, you define an objective function, which the ESC controller then maximizes to find the controller gain K.

For this example, use the following objective function.

$$J = -10\int \left(x(t) - x_{ref}(t)\right)^2 dt$$

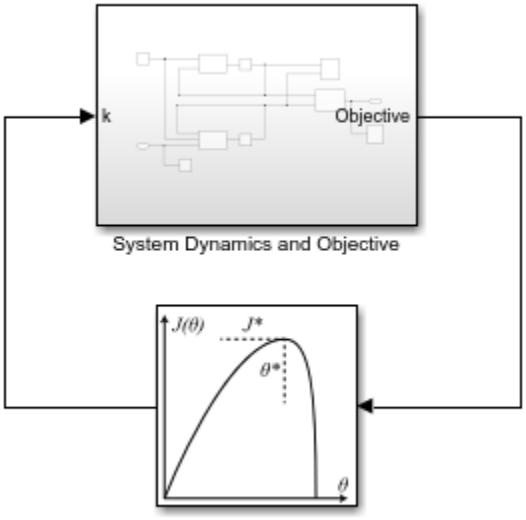
The following figure shows the setup for extremum seeking control.

- The cost function is computed from the outputs of the reference system and the actual system.
- The extremum seeking controller updates the gain parameter.
- The control action is updated using the new gain value.
- This control action is applied to the actual system.



The `firstOrderRefTracking_Esc` Simulink model implements this problem configuration.

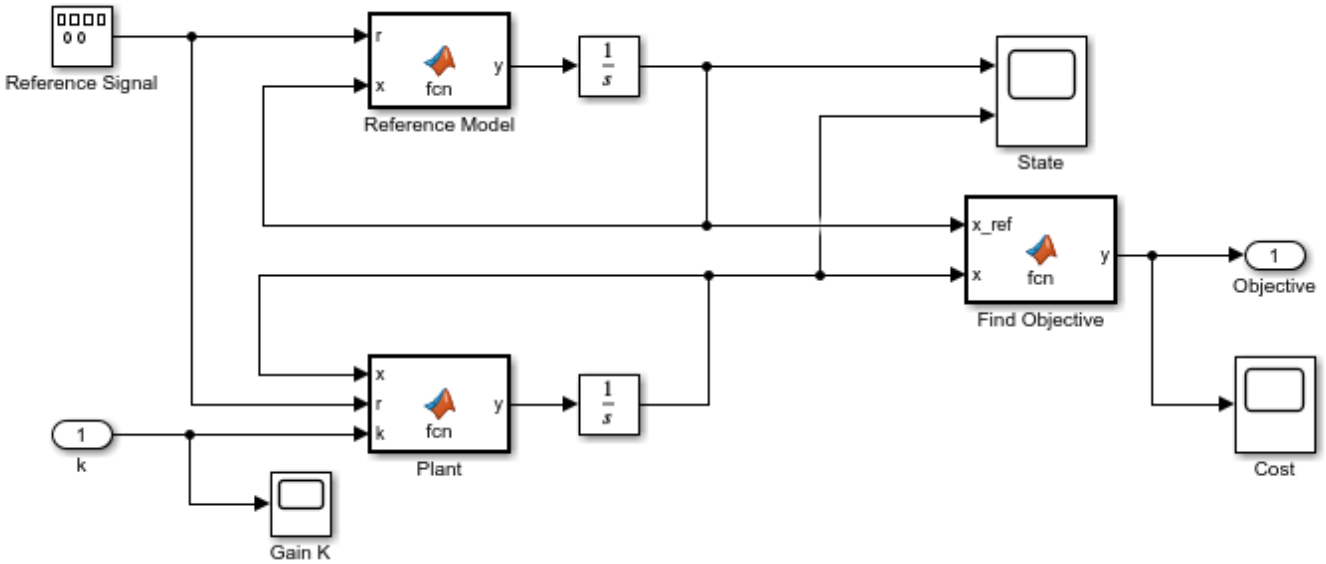
```
mdl = 'firstOrderRefTracking_Esc';
open_system(mdl)
```



In this model, you use the Extremum Seeking Control block to optimize the gain value.

The System Dynamics and Objective subsystem contains the reference model, the plant (including the actual system and control action), and the objective function computation. These elements are all implemented using MATLAB Function blocks.

```
open_system([mdl '/System Dynamics and Objective'])
```



Specify an initial guess for the gain value.

```
IC = 0;
```

The Extremum Seeking Control block perturbs the parameter value using a modulation signal. It then demodulates the resulting change in the objective function signal before computing the parameter update. Configure the extremum seeking control parameters for this block.

First specify the number of parameters to be tuned (`N`) and the learning rate (`lr`).

```
N = 1;
lr = 0.55;
```

Configure the demodulation and modulation signals by specifying their frequency (`omega`), phases (`phi_1` and `phi_2`), and their amplitudes (`a` and `b`).

```
omega = 5; % Forcing frequency
a = 1;     % Demodulation amplitude
b = 0.1;   % Modulation amplitude
phi_1 = 0; % Demodulation phase
phi_2 = 0; % Modulation phase
```

For this example, the Extremum Seeking Control block is configured to remove high-frequency noise from the demodulated signal. Set the cutoff frequency for the corresponding low-pass filter.

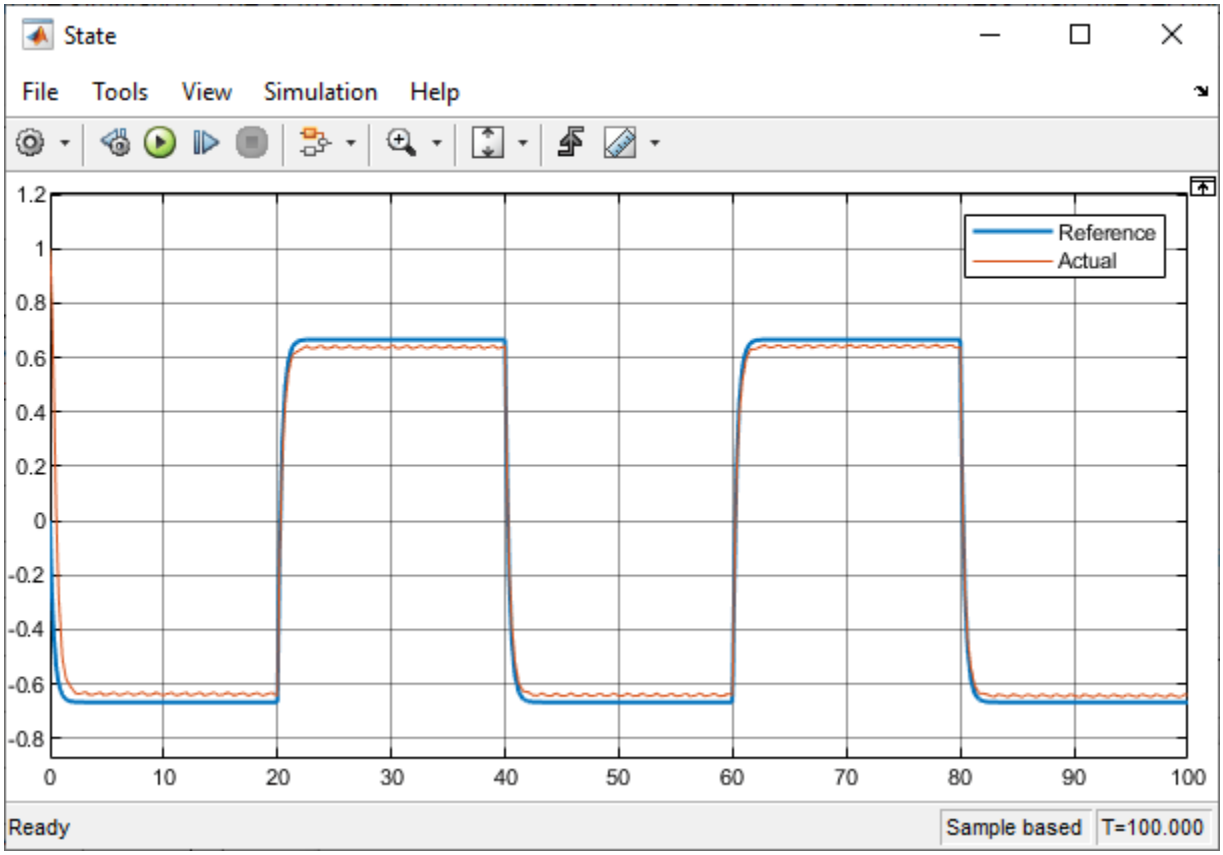
```
omega_lpf = 1;
```

Simulate the model.

```
sim(mdl);
```

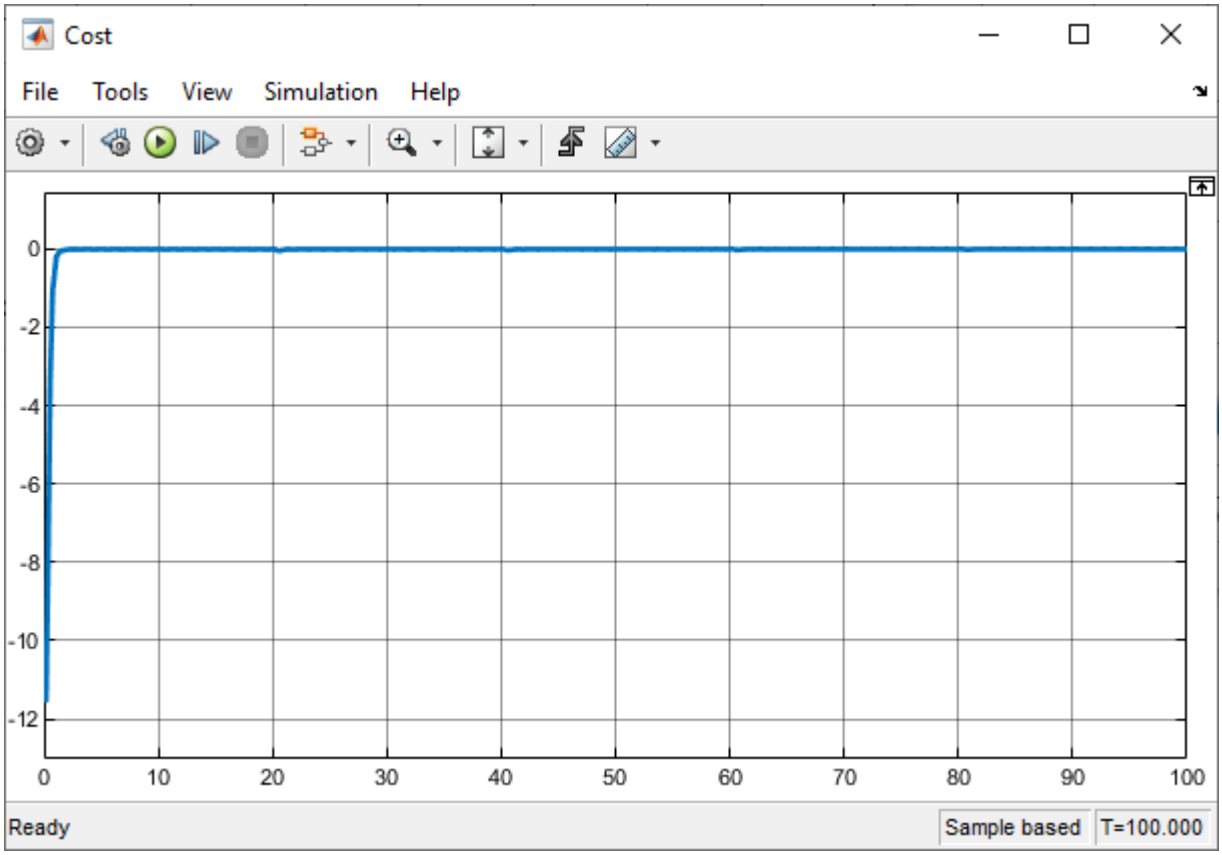
To check the reference tracking performance, view the state trajectories from the simulation. The actual trajectory converges to the reference trajectory in less than five seconds.

```
open_system([mdl '/System Dynamics and Objective/State'])
```



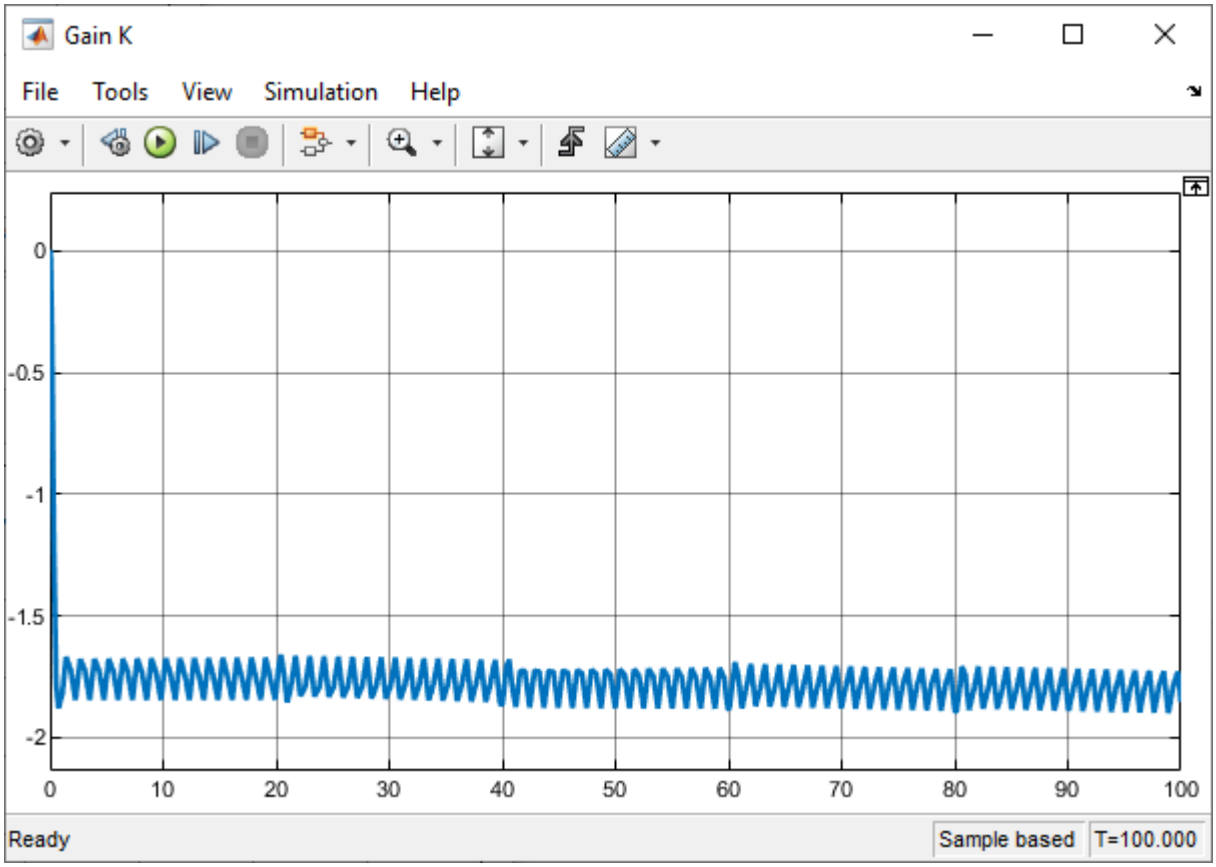
To examine the behavior of the ESC controller, first view the objective function, which reaches its maximum value quickly.

```
open_system([mdl '/System Dynamics and Objective/Cost'])
```



By maximizing the objective function, the ESC controller optimizes the control gain value near its ideal value of –2. The fluctuations in the gain value are due to the modulation signal from the Extremum Seeking Control block.

```
open_system([mdl '/System Dynamics and Objective/Gain K'])
```




```
bdclose(mdl)
```

See Also

Blocks
Extremum Seeking Control

Related Examples

- “Anti-Lock Braking Using Extremum Seeking Control” on page 15-15
- “Adaptive Cruise Control Using Extremum Seeking Control” on page 15-21

Anti-Lock Braking Using Extremum Seeking Control

This example shows how to use the extremum seeking control (ESC) to optimize braking torque for an anti-lock braking system (ABS).

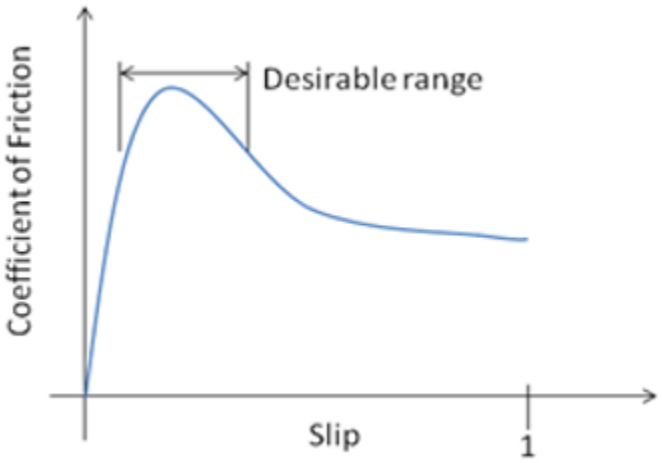
Anti-Lock Braking System

An anti-lock braking system prevents vehicle brakes from locking up by adjusting the braking torque for each wheel. For such a system, the following function defines the slip coefficient of a wheel.

$$slip = 1 - \frac{\omega_w}{\omega_v}$$

Here, ω_w is the wheel angular velocity and ω_v is the wheel angular velocity under a nonbraking condition (vehicle speed divided by wheel radius). Based on this equation, slip is zero when the wheel speed and vehicle speed are equal, and slip equals one when the wheel is locked (ω_w is zero). A desirable slip value for braking is 0.2, which means that the number of wheel revolutions equals 0.8 times the number of revolutions under nonbraking conditions with the same vehicle velocity. This slip value maximizes the adhesion between the tire and road and minimizes the stopping distance for the available friction.

The friction coefficient μ between the tire and the road surface is a function of slip, known as the *mu-slip curve*.



The frictional force F_f acting on the circumference of the tire is the product of the friction coefficient μ multiplied by the weight on the wheel W. F_f divided by the vehicle mass is equal to the vehicle deceleration, which you can integrate to obtain vehicle velocity.

Ideally, an anti-lock braking controller uses *bang-bang control* based upon the error between the actual slip and desired slip. The desired slip value is a constant and corresponds to the slip value for which the mu-slip curve reaches its peak value. For more information, see “Modeling an Anti-Lock Braking System”.

Specify Vehicle Model Parameters

Define the following vehicle parameters for this example.

- `m` — Vehicle mass
- `W` — Vehicle weight
- `B` — Wheel damping torque coefficient
- `Rr` — Wheel radius
- `I` — Wheel inertia

```
m = 400;
W = m*9.81;
B = 0.01;
Rr = 0.3;
I = 1;
```

Also, specify the initial vehicle forward velocity `v0` and initial wheel angular velocity `x0`.

```
v0 = 120/3.6;
w0 = 400/3.6;
```

Extremum Seeking Control for Anti-Lock Braking System

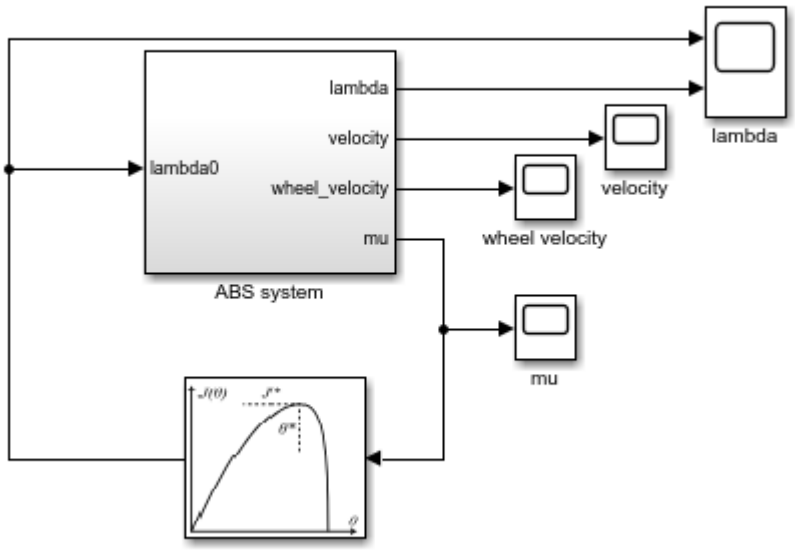
For this ABS example, you design an extremum seeking controller that maximizes the friction coefficient, which is a function of the slip coefficient as shown in the following equation.

$$\mu = \frac{2\mu^*\lambda^*\lambda}{\left(\lambda^{*2} + \lambda^2\right)}$$

Here, μ^* and λ^* are the ideal friction and slip coefficients, respectively. The actual friction μ equals μ^* when the achieved slip coefficient λ equals the ideal slip coefficient λ^*. The ABS achieves this objective of maximum deceleration, and hence the shortest stopping distance, by controlling the braking torque, which is a function of the slip and friction coefficients.

Simulink Control Design software implements the ESC algorithm using the Extremum Seeking Control block. For this example, open the `ExtremumSeekingControlABS` model, which includes this block along with an ABS system model.

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



The output of the Extremum Seeking Control block is the slip coefficient λ. Since the ESC controller maximizes the value of μ, use this value as the objective function input for the block.

Specify an initial guess for the slip coefficient.

```
IC = 0.15;
```

Also, specify the ideal slip coefficient `lambda_star` and ideal friction coefficient `mu_star`.

```
lambda_star = 0.25;
mu_star = 0.6;
```

The Extremum Seeking Control block perturbs the parameter value using a modulation signal. It then demodulates the resulting change in the objective function signal before computing the parameter update. Configure the extremum seeking control parameters for this block.

First specify the number of parameters to be tuned (`N`) and the learning rate (`lr`).

```
N = 1;
lr = 0.3;
```

Configure the demodulation and modulation signals by specifying their frequency (`omega`), phases (`phi_1` and `phi_2`), and amplitudes (`a` and `b`).

```
omega = 0.7; % Forcing frequency
a = 1; % Demodulation amplitude
b = 0.02; % Modulation amplitude
phi_1 = pi/2; % Demodulation phase
phi_2 = 0; % Modulation phase
```

For this example, use a low-pass filter to remove high-frequency noise from the demodulated signal and a high-pass filter to remove bias from the perturbed objective function signal. Specify the cutoff frequencies for these filters.

```
omega_lpf = 1;
omega_hpf = 0.5;
```

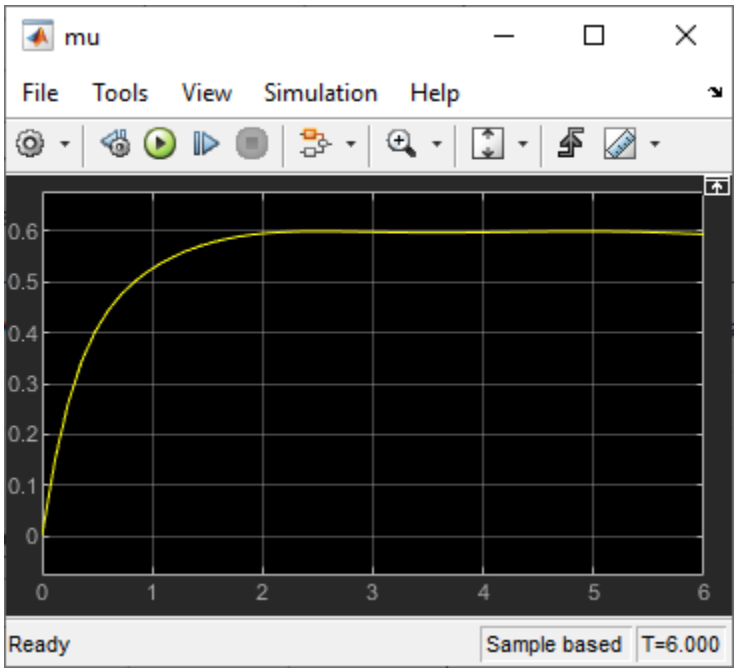
Simulate Anti-Lock Braking System

Simulate the model.

```
sim(mdl);
```

View the friction coefficient simulation result. Within two seconds, μ reaches its maximum value.

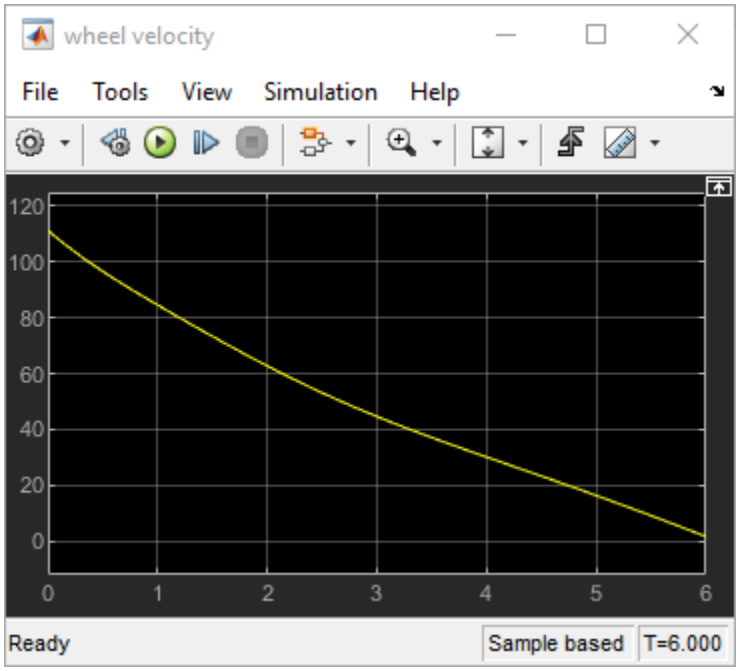
```
open_system([mdl '/mu'])
```



View the vehicle velocity and the wheel angular velocity, both of which decrease to zero during the braking simulation.

```
open_system([mdl '/velocity'])
```

```
open_system([mdl '/wheel velocity'])
```




```
bdclose('ExtremumSeekingControlABS')
```

Reference

[1] Ariyur, Kartik B., and Miroslav Krstić. *Real Time Optimization by Extremum Seeking Control*. Hoboken, NJ: Wiley Interscience, 2003.

See Also

Blocks

Extremum Seeking Control

Related Examples

- “Extremum Seeking Control for Reference Model Tracking of Uncertain Systems” on page 15-8
- “Adaptive Cruise Control Using Extremum Seeking Control” on page 15-21

Adaptive Cruise Control Using Extremum Seeking Control

This example shows how to implement adaptive cruise control using an extremum seeking control (ESC) approach. In this example, the goal is to make an ego car travel at a set velocity while maintaining a safe distance from a lead car by controlling longitudinal acceleration and braking.

Adaptive Cruise Control System

Adaptive cruise control (ACC) is a system designed to help vehicles maintain a safe following distance and stay within the speed limit. A vehicle equipped with an ACC system (ego car) uses radar to measure relative distance (D_{rel}) and relative velocity (V_{rel}) with respect to the leading vehicle. The ACC system is designed to maintain a desired cruising speed (V_{set}) or maintain a relative safe distance (D_{safe}) from the leading car. The switch in the control objective is determined based on the following conditions.

- If $D_{rel} > D_{safe}$, the ACC system follows the desired reference cruise velocity commanded by the driver.
- If $D_{rel} < D_{safe}$, the ACC system controls the relative position of the ego car with respect to the lead car.

This example uses the same ego and lead car model as the “Adaptive Cruise Control System Using Model Predictive Control” (Model Predictive Control Toolbox).

Implement the longitudinal vehicle dynamics as a simple second-order linear model.

```
G = tf(1,[0.5,1,0]);
```

Configure the ACC parameters for the example.

```
D_default = 10; % Default spacing (m)
t_gap = 1.4;    % Time gap (s)
v_set = 30;     % Driver-set velocity (m/s)
amin_ego = -3;  % Minimum acceleration for driver comfort (m/s^2)
amax_ego = 2;   % Maximum acceleration for driver comfort (m/s^2)
Ts = 0.1;       % Sample time (s)
Tf = 150;       % Duration (s)
```

Specify the initial position and velocity for both the lead car and ego car.

```
x0_lead = 50; % Initial lead car position (m)
v0_lead = 25; % Initial lead car velocity (m/s)
x0_ego = 10;  % Initial ego car position (m)
v0_ego = 20;  % Initial ego car velocity (m/s)
```

ACC Using Extremum Seeking Control

An extremum seeking controller achieves satisfactory control performance by adjusting control parameters to maximize an objective function in real time. For this example, use the following objective function, which depends on relative distance, safe distance, relative velocity, and set velocity.

$$J = -\int Q_d\left(D_{rel} - D_{safe}\right)^2 + Q_v\left(v_{rel} - v_{set}\right)^2$$

Here, Q_d and Q_v are objective function weights for the distance error and velocity error terms, respectively.

```
Qd = 0.5;
Qv = 1;
```

The extremum seeking controller adapts the following controller gains.

- K_{xerr} — Position error gain
- K_{verr} — Velocity error gain
- K_{vrel} — Relative velocity gain

Specify initial guesses for the gain values.

```
Kverr = 1;   % ACC velocity error gain
Kxerr = 1;   % ACC spacing error gain
Kvrel = 0.5; % ACC relative velocity gain
```

Specify Extremum Seeking Control Parameters

Simulink Control Design software implements the ESC algorithm using the Extremum Seeking Control block. Configure the parameters for this block.

Specify the number of parameters to tune (the three controller gains). The controller uses a separate tuning loop for each parameter.

```
N = 3;
```

Specify the initial conditions for the parameter update integrators by scaling the initial gain values with respect to the learning rate for each parameter `lr`.

```
lr = 0.02*[2 3 1];
IC = [Kverr,Kxerr,Kvrel];
```

Configure the demodulation and modulation signals by specifying their frequencies (`omega`), phases (`phi_1` and `phi_2`), and amplitudes (`a` and `b`). Each parameter must use a different forcing frequency. For this example, use the same modulation and demodulation phases and amplitudes for all parameters.

```
omega = 0.8*[5,7,8]; % Forcing frequency (rad/s)
a = 0.01;            % Demodulation amplitude
b = 0.5*lr;          % Modulation amplitude
phi_1 = 0;           % Demodulation phase (rad)
phi_2 = pi/4;        % Modulation phase (rad)
```

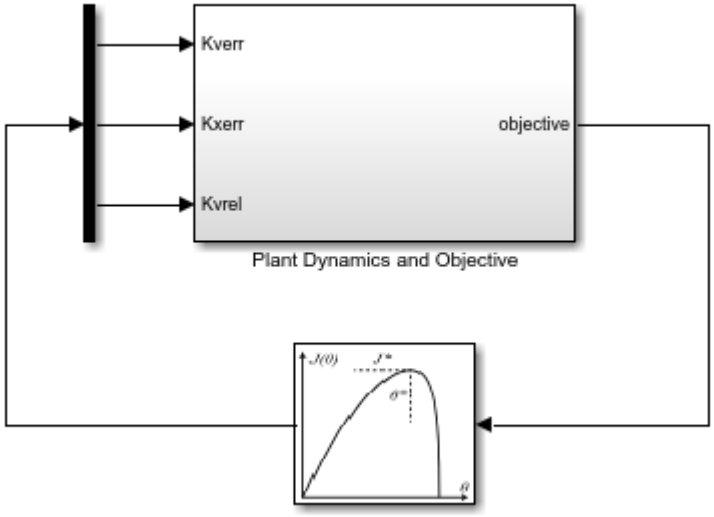
Use a low-pass filter to remove high-frequency noise from the demodulated signal and a high-pass filter to remove bias from the perturbed objective function signal. Specify the cutoff frequencies for these filters.

```
omega_lpf = 0.04;
omega_hpf = 0.01;
```

Simulate Adaptive Cruise Control System

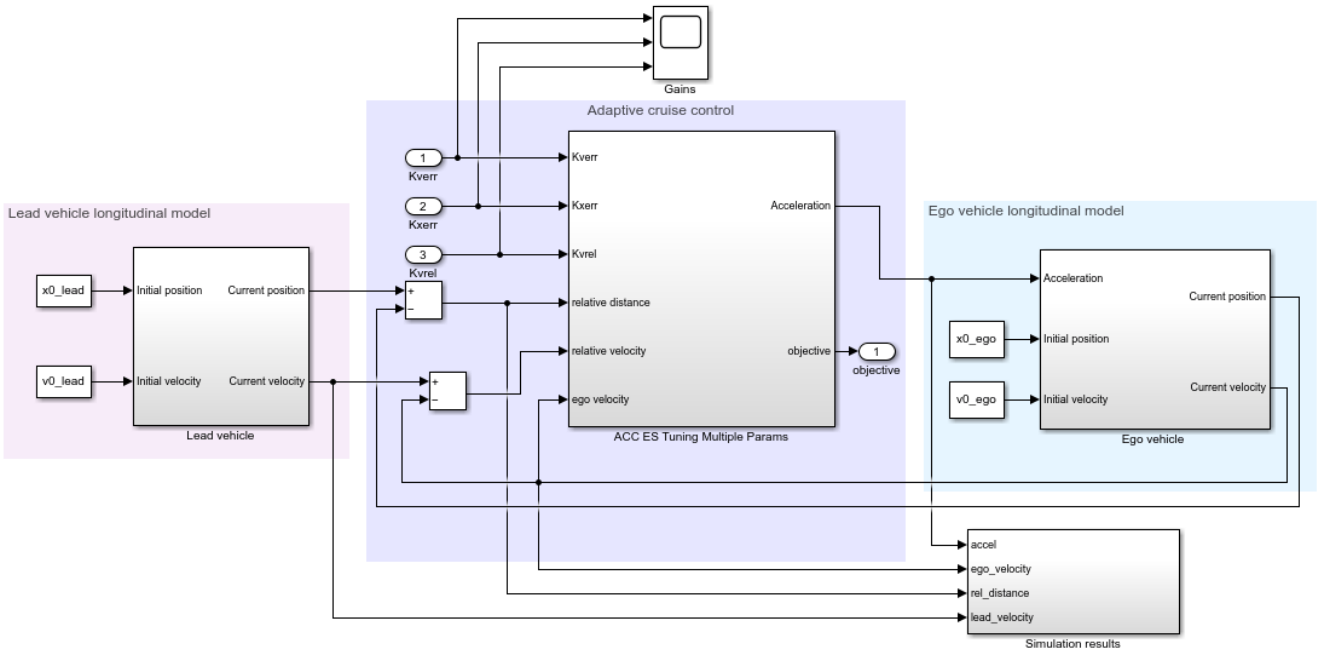
To simulate the ESC adaptive cruise controller, open the `ExtremumSeekingControlACC` model.

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



The Plant Dynamics and Objective subsystem contains the ACC models and computes the objective function for the ESC algorithm.

```
open_system([mdl '/Plant Dynamics and Objective'])
```



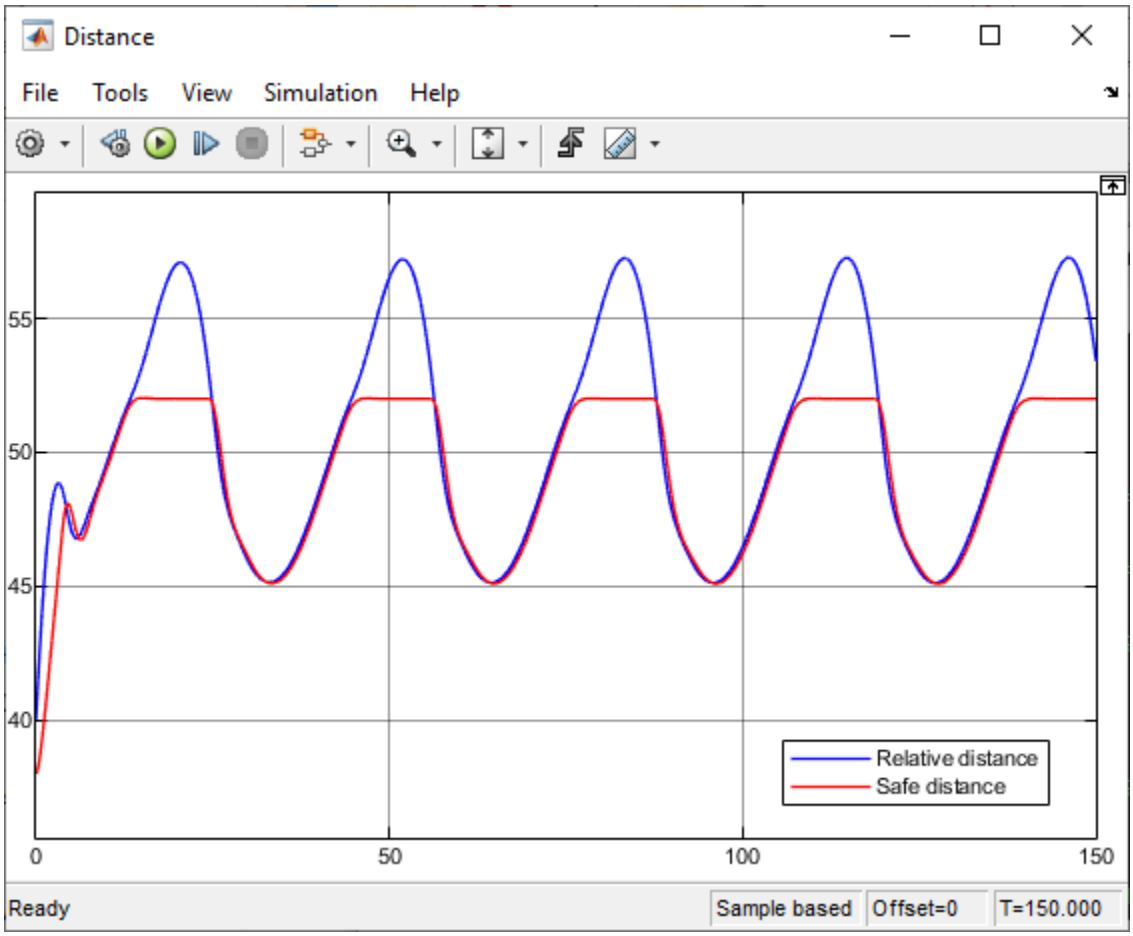
Simulate the model. During the simulation, the lead car velocity varies sinusoidally. Therefore, the ego car must adjust its velocity to compensate.

```
sim(mdl);
```

The following plot shows the relative distance between the lead and ego cars and the safe distance.

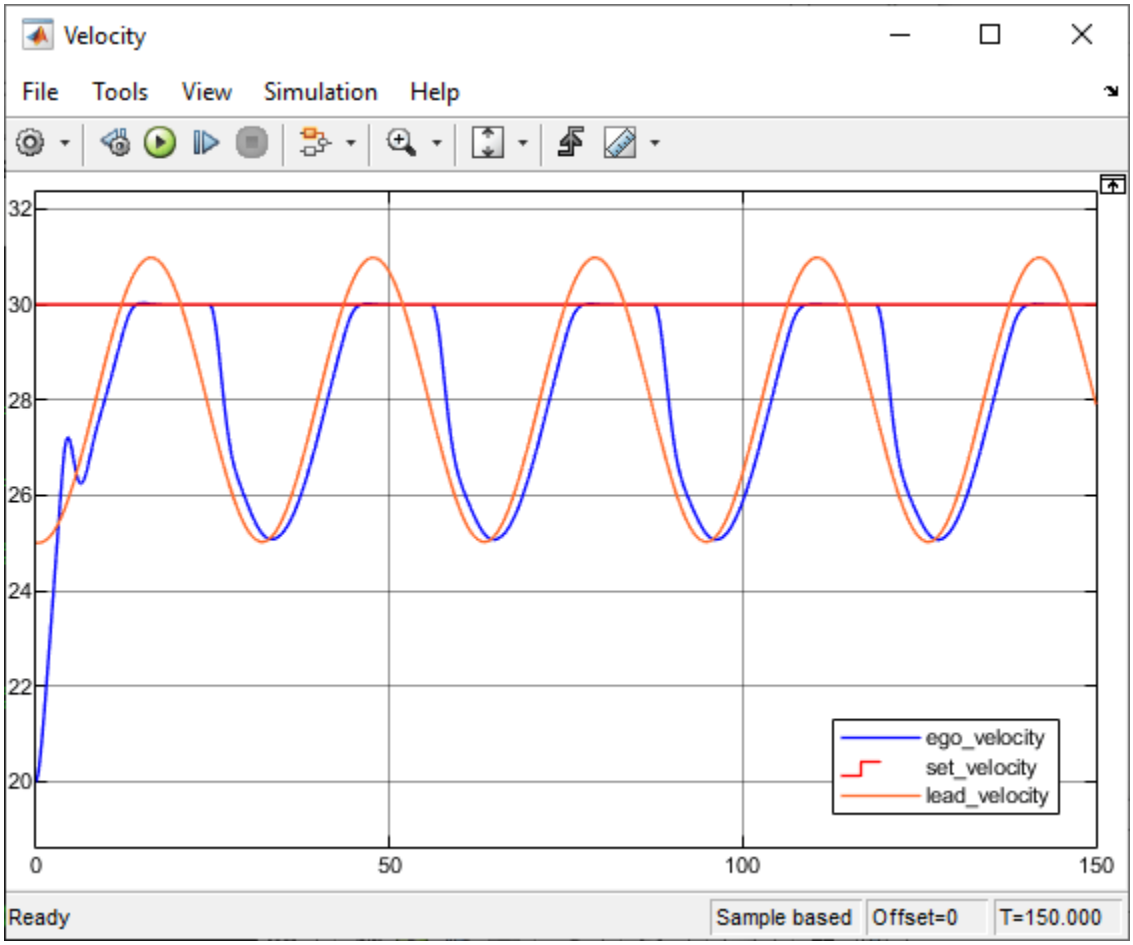
- The safe distance varies as the ego car velocity changes.
- The relative distance between the ego and lead cars occasionally drops slightly below the safe distance. This result is because the ACC system enforces the relative distance using a soft constraint.

```
open_system([mdl '/Plant Dynamics and Objective/Simulation results/Distance'])
```



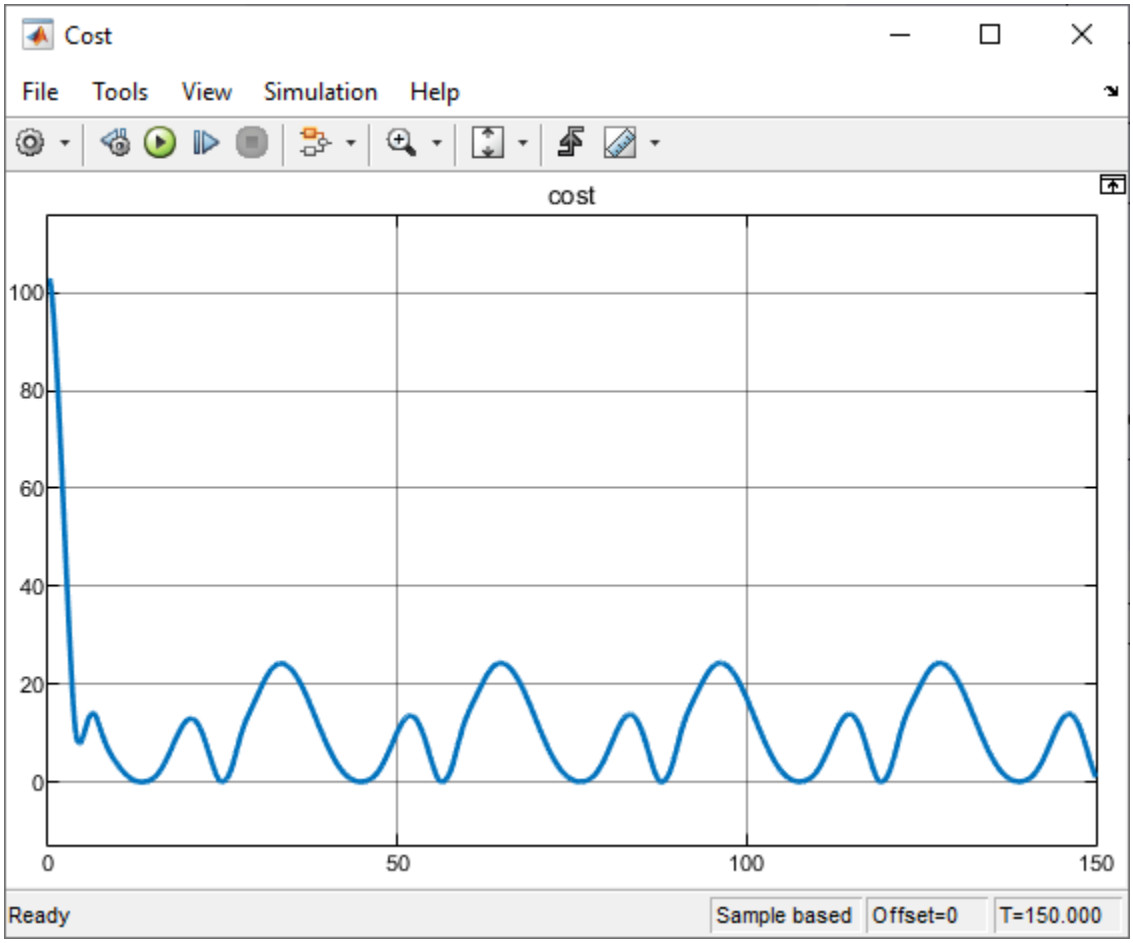
View the velocities of the ego and lead cars along with the ego car set velocity. To maintain a safe distance the ACC system adjusts the ego car velocity as the lead car velocity changes. When the lead car velocity is greater than the set velocity, the ego car stops tracking the lead car velocity and cruises at the set velocity.

```
open_system([mdl '/Plant Dynamics and Objective/Simulation results/Velocity'])
```



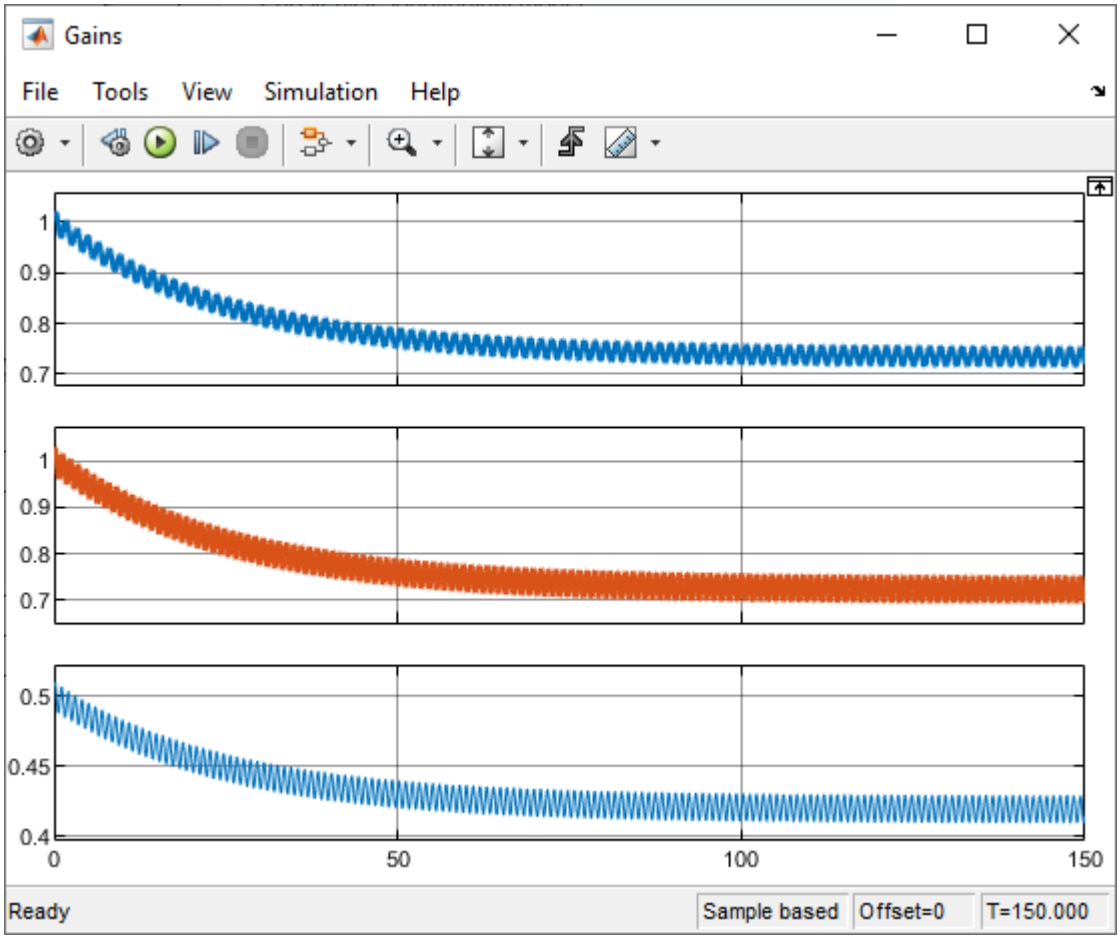
The next plot shows the cost function that ESC seeks to optimize when searching for optimal control gains.

```
open_system([mdl '/Plant Dynamics and Objective/Simulation results/Cost'])
```



View the resulting controller gains, which adapt over the course of the simulation. The top plot is K_{verr}, the middle plot is K_{xerr}, and the bottom plot is K_{vrel}. Fluctuations in the gain values are due to the modulation signals from the Extremum Seeking Control block.

```
open_system([mdl '/Plant Dynamics and Objective/Gains'])
```




```
bdclose('ExtremumSeekingControlACC')
```

See Also

Blocks
Extremum Seeking Control

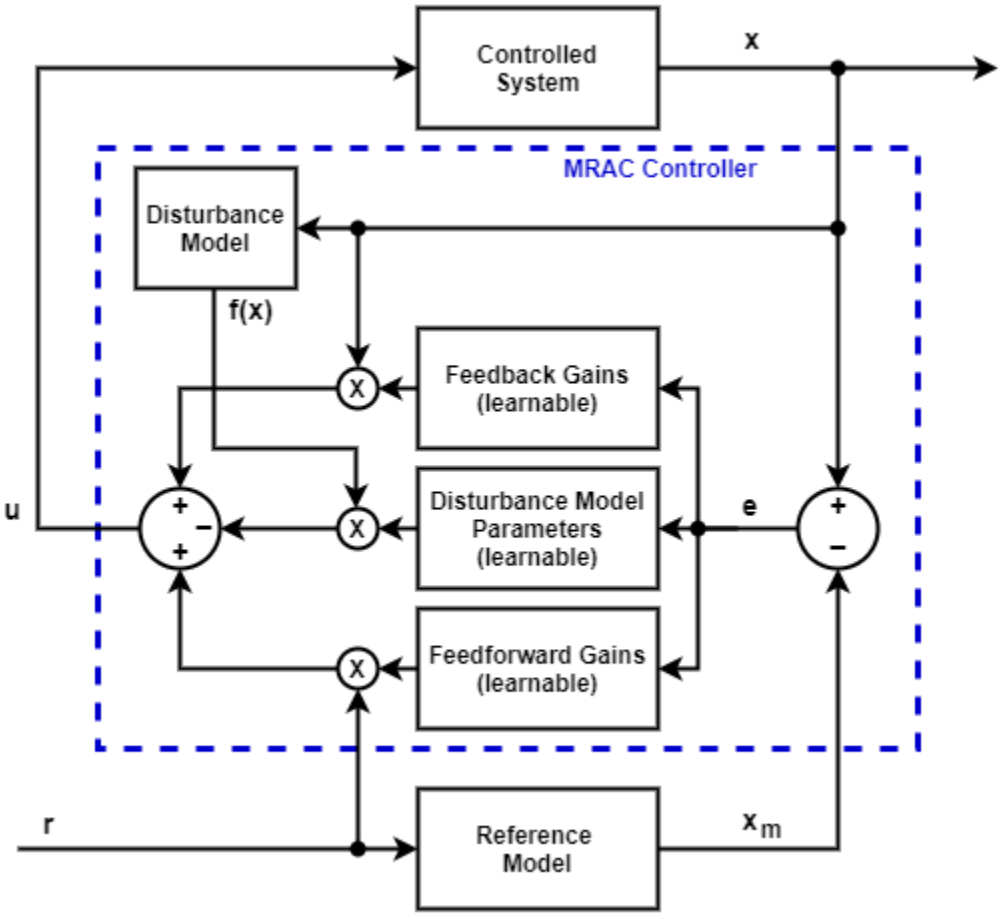
Related Examples

- “Extremum Seeking Control for Reference Model Tracking of Uncertain Systems” on page 15-8
- “Anti-Lock Braking Using Extremum Seeking Control” on page 15-15

Model Reference Adaptive Control

Model reference adaptive control (MRAC) is a model-based, real-time adaptive control algorithm that computes control actions to make an uncertain controlled system track the behavior of a given reference plant model. Using Simulink Control Design software, you can implement MRAC using the Model Reference Adaptive Control block.

The following figure shows the structure of an MRAC controller.



The goal of the controller is to make the states (x) of the controlled system track the states (x_m) of the reference model for a given reference signal (r). For more information on the controlled system and reference models, see “Nominal and Reference Models” on page 15-29. The MRAC controller uses the following process to generate a control signal.

1. Compute the error e between the states of the controlled system and the states of the reference model.
2. Compute the features $\phi(x)$ for the internal model of the system disturbances and uncertainty.
3. Update the weight parameters of the disturbance model based on the error e.
4. Update the values of the feedback and feedforward gains based on error e. You can choose to have the controller adapt the feedforward gains, the feedback gains, or both.

5 Apply the updated gains and parameters to the reference signal, plant states, and disturbance model features.

6 Compute the control action u. For more information, see “Control Structure” on page 15-29.

Nominal and Reference Models

The controlled system, which typically exhibits modeling uncertainty and external disturbances, has the following nominal state equation. This model represents the nominal behavior that you expect from the plant.

$$\dot{x}(t) = Ax(t) + B(u(t) + f(x))$$

Here:

- $x(t)$ is the state of the system you want to control.
- $u(t)$ is the control input.
- A is a constant state-transition matrix.
- B is a constant control effective matrix.
- $f(x)$ is the matched uncertainty in the system.

The reference plant model is the ideal system that characterizes the desired behavior that you want to achieve in practice.

$$\dot{x}_m(t) = A_m x_m(t) + B_m r(t)$$

Here:

- $r(t)$ is the external reference signal
- $x_m(t)$ is the state of the reference plant model. Since $r(t)$ is known, you can simulate the reference model to get $x_m(t)$.
- A_m is a constant state matrix. For a stable reference model, A_m must be a Hurwitz matrix for which every eigenvalue must have a strictly negative real part.
- B_m is a control effective matrix.

Control Structure

MRAC computes the control input $u(t)$ using the following control structure.

$$u(t) = k_x x(t) + k_r r(t) - u_{ad}$$

$$u_{ad} = w^T \phi(x)$$

Here:

- k_x is the feedback gain matrix.
- k_r is the feedforward gain matrix.
- u_{ad} is the adaptive control component derived from the disturbance model.
- $\phi(x)$ contains the disturbance and uncertainty model features. For more information on the model features, see “Disturbance and Uncertainty Model Features” on page 15-31.

- w is an adaptive control weight vector.

An MRAC controller can learn the values of k_x, k_r, and w in real time.

When you substitute the control structure into the nominal model and simplify the result, you get the following state equation.

$$\dot{x}(t) = (A + Bk_x)x(t) + Bk_r r(t) + B\left(f(x) - w^T\phi(x)\right)$$

Uncertainty and Error Dynamics

Using the universal approximation property of neural networks, you can assume that an ideal neural network with unknown ideal weights can parameterize the true uncertainty.

$$f(x) = w^{*T}\phi(x) + \varepsilon$$

Here, w^* contains the unknown ideal weights and ε is the approximation error.

Further, if you define the weight vector error as $\tilde{w} = w* - w$ and substitute it into the control structure, you get the following state equation.

$$\dot{x}(t) = (A + Bk_x)x(t) + Bk_r r(t) + B\left(\tilde{w}^T\phi(x) + \varepsilon\right)$$

The goal of the MRAC controller is the asymptotic convergence of the state tracking error $e(t)$ to zero.

$$e(t) = x(t) - x_m(t)$$
$$\dot{e}(t) = \dot{x}(t) - \dot{x}_m(t)$$

Substitute the control structure and reference model equations to obtain the following equation for the error dynamics.

$$\dot{e}(t) = (A + Bk_x)x(t) + Bk_r r(t) + B\left(\tilde{w}^T\phi(x) + \varepsilon\right) - A_m x_m(t) - B_m r(t)$$

You can rewrite this equation as follows.

$$\dot{e}(t) = A_m e(t) + B\tilde{k}_x x(t) + B\tilde{k}_r r(t) + B\left(\tilde{w}^T\phi(x) + \varepsilon\right)$$
$$\tilde{k}_x = k_x^* - k_x$$
$$\tilde{k}_r = k_r^* - k_r$$

Here, $\tilde{k}_x$ and $\tilde{k}_r$ are feedback and feedforward gain errors, respectively. k_x^* and k_r^* are unknown ideal feedback and feedforward gains.

Parameter Updates

To derive the MRAC parameter update equations, first define the following Lyapunov function based on the error dynamics.

$$V\left(e, \tilde{k}_x, \tilde{k}_r, \tilde{w}\right) = e^T Q e + \frac{\Gamma_x \tilde{k}_x^2}{2} + \frac{\Gamma_r \tilde{k}_r^2}{2} + \frac{\tilde{w}^T \Gamma_w \tilde{w}}{2}$$

Here, Γ_x, Γ_r, and Γ_w are learning rates for the parameter updates. Q is a weighting matrix for state tracking errors.

To obtain the following parameter update equations, take the Lie derivative of the preceding Lyapunov function along the error dynamics and equate it to zero.

$$\dot{k}_x = \Gamma_x x(t) e^T(t) PB$$
$$\dot{k}_r = \Gamma_r x(t) e^T(t) PB$$
$$\dot{w}_x = \Gamma_w \phi(t) e^T(t) PB$$

Here, P is the solution to the following Lyapunov function based on the reference model state matrix.

$$A_m^T P + P A_m + Q = 0$$

Learning Modification

To add robustness at higher learning rates, you can modify the parameter updates to include an optional momentum term. You can choose one of two possible learning modification methods: *sigma modification* and *e-modification*.

For sigma modification, the momentum term for each parameter update is the product of the momentum weight parameter σ and the current parameter value.

$$\dot{k}_x = \Gamma_x x(t) e^T(t) PB + \sigma k_x$$
$$\dot{k}_r = \Gamma_r x(t) e^T(t) PB + \sigma k_x$$
$$\dot{w}_x = \Gamma_w \phi(t) e^T(t) PB + \sigma w$$

For e-modification, the controller scales the sigma-modification momentum term by the norm of the error vector.

$$\dot{k}_x = \Gamma_x x(t) e^T(t) PB + \sigma |e(t)| k_x$$
$$\dot{k}_r = \Gamma_r x(t) e^T(t) PB + \sigma |e(t)| k_x$$
$$\dot{w}_x = \Gamma_w \phi(t) e^T(t) PB + \sigma |e(t)| w$$

To adjust the amount of the learning modification for either method, change the value of the momentum weight parameter σ.

Disturbance and Uncertainty Model Features

The Model Reference Adaptive Control block maintains an internal model u_{ad} of the disturbance and model uncertainty in the controlled system.

$$u_{ad} = w^T \phi(x)$$

Here, $\phi(x)$ is a vector of model features. w is an adaptive control weight vector that the controller updates in real time.

To define $\phi(x)$, you can use one of the following feature definitions.

- State vector of the controlled plant — This approach can under-represent the uncertainty and perform poorly. Use this option when the disturbance and model uncertainty are linear. Using the state vector can also be a useful starting point when you do not know the complexity of the disturbance and model uncertainty.
- Gaussian radial basis functions — Use this option when the disturbance and model uncertainty are nonlinear and structure of the disturbance model is unknown.
- External source provided to the controller block — Use this option to define your own custom feature vector. You can use this option when you know the structure of the disturbance and uncertainty model. For example, you can use a custom feature vector to identify specific unknown plant parameters.

See Also

Blocks
Model Reference Adaptive Control

Related Examples

- “Model Reference Adaptive Control of Satellite Spin” on page 15-33
- “Model Reference Adaptive Control of Aircraft Undergoing Wing Rock” on page 15-41

Model Reference Adaptive Control of Satellite Spin

This example shows how to control satellite spin using model reference adaptive control (MRAC) to make the unknown controlled system match an ideal reference model. The satellite system is modeled in Simulink® and the MRAC controller is implemented using the Model Reference Adaptive Control block provided by Simulink Control Design™ software.

Satellite-Spin Control System

In the model for this example, the cylindrical body of the satellite spins around its axis of symmetry (z axis) with constant angular rate Ω. [1] The goal is to independently control the angular rates ω_x and ω_y around the x and y axes using torques u_x and u_y. The equations of motion produce a second-order model with two inputs and two outputs.

$$\begin{bmatrix}\dot{\omega}_x\\ \dot{\omega}_y\end{bmatrix} = \begin{bmatrix}0 & a\\ -a & 0.5a\end{bmatrix}\begin{bmatrix}\omega_x\\ \omega_y\end{bmatrix} + \begin{bmatrix}u_x\\ u_y\end{bmatrix}$$

In this system, the controller does not observe the plant states directly. Instead, it views the states indirectly through following measurement model.

$$\begin{bmatrix}\nu_1\\ \nu_2\end{bmatrix} = \begin{bmatrix}1 & a\\ -a & 1\end{bmatrix}\begin{bmatrix}\omega_x\\ \omega_y\end{bmatrix}$$

For this example, assume that the actual plant model is unknown and that the MRAC controller uses the following nominal model of the plant dynamics.

$$\begin{bmatrix}\dot{\omega}_x\\ \dot{\omega}_y\end{bmatrix} = \begin{bmatrix}0 & 0.5a\\ -0.5a & 0\end{bmatrix}\begin{bmatrix}\omega_x\\ \omega_y\end{bmatrix} + \begin{bmatrix}u_x\\ u_y\end{bmatrix}$$

Given this uncertain nonlinear system, your goal is to design a controller that enables the system to track the following decoupled reference model.

$$\begin{bmatrix}\dot{x}_{m1}\\ \dot{x}_{m2}\end{bmatrix} = \begin{bmatrix}-5 & 0\\ 0 & -5\end{bmatrix}\begin{bmatrix}x_{m1}\\ x_{m2}\end{bmatrix} + \begin{bmatrix}r_1\\ r_2\end{bmatrix}$$

Here:

- x_m is the reference model state vector.
- $r(t)$ contains the angular rate reference signals.

Define Nominal and Reference Models

Define the dynamic model for the satellite system. For this system, `A` represents the true state matrix.

```
a = 10;                 % System Constant
A = [0 a;-a -a/2];      % True model parameters
B = eye(2);
C = [1 a;-a 1];
D = zeros(2,2);
```

Specify a plant mismatch matrix. This matrix is subtracted from the true `A` matrix to define the nominal plant used by the controller. The goal of the controller is to model this plant uncertainty.

```
deltaA = [0 0.5*a; -0.5*a 0];   % Plant mismatch
```

The reference system is the following stable second order system.

```
Am = [-5 0; 0 -5];    % Second-order decoupled integrator model
Bm = diag([5 5]);     % Nominal control effective matrix
```

Define the initial conditions for the satellite spin rates and reference model states.

```
x1_0 = 0.5;     % Initial x-axis angular rate (rad/s)
x2_0 = 0.5;     % Initial y-axis angular rate (rad/s)
xm_0 = [0;0];   % Initial conditions of the reference plant model
```

Model Reference Adaptive Control Structure

The goal of the MRAC controller is to achieve asymptotic convergence of the tracking error $e(t) = x(t) - x_m(t)$.

$$\lim_{t \to \infty} (x(t) - x_m(t)) \to 0$$

The MRAC controller has the following structure.

$$u(t) = -k_x x(t) + k_r r(t) - u_{ad}$$

Here:

- k_x contains feedback control gains.
- k_r contains feedforward control gains.
- u_{ad} is an adaptive control term that cancels the model uncertainty.

The Model Reference Adaptive Control block adjusts the adaptive control term to achieve the desired reference model tracking. Optionally, you can also adapt the feedback and feedforward control gains.

For this example, the controller adapts the feedback gains and keeps the feedforward gains static.

The initial feedback gain and the static feedforward gain are computed to satisfy the following model matching condition

$$A_m = A - Bk_x$$
$$B_m = Bk_r$$

Specify the computed controller gains and the feedback gain learning rate.

```
Kx = -place(A,B,eig(Am));  % State feedback gain for pole placement
gain_kx = 1;               % Feedback gain learning rate
Kr = Bm;                   % Feedforward gain for ref model matching
```

Configure Uncertainty Estimation Parameters

The MRAC controller estimates the model uncertainty online and generates an adaptive control action u_{ad} that cancels the uncertainty to recover the nominal system for the baseline controller. The adaptive control term models the system uncertainty using the following model.

$$u_{ad} = \mathbf{w}^T \phi(x)$$

Here:

- **w** contains the network weights that are adjusted by the controller.
- $\phi(x)$ is the uncertainty model feature vector.

Using the Model Reference Adaptive Control block, you can select one of the following feature vector definitions.

- System states, where $\phi(x) = x(t)$ — This approach is the simplest option, which can be a good starting point if you do not know the complexity of the system uncertainty. If you find that using the states as features does not adequately represent the nonlinear uncertainty, select one of the other approaches.
- Radial basis functions with Gaussian kernels, where $\phi(x) = \left[\exp\left(-\frac{(x-c_i)^2}{\sigma_i}\right)\right]_{i=1}^{N}$. You can configure the kernels by defining the feature centers c_i and bandwidth σ_i.
- Custom features provided by an optional input port.

For this example, the controller is configured to use the system states as the disturbance model features.

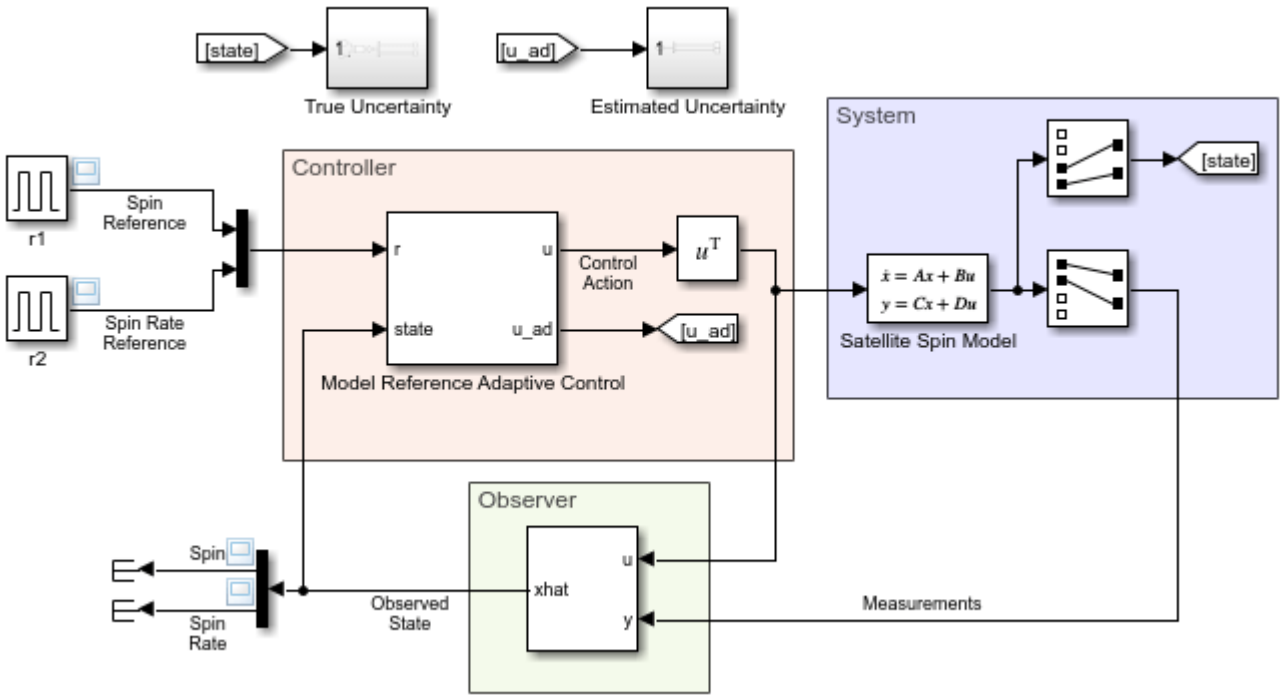
Define the model estimation learning rate `gamma_w` and tracking error weight `Q`.

```
gamma_w = 100;  % Learning rate
Q = 10;         % Tracking error weight
```

Simulate Controller

Open the Simulink model of the satellite-spin control system.

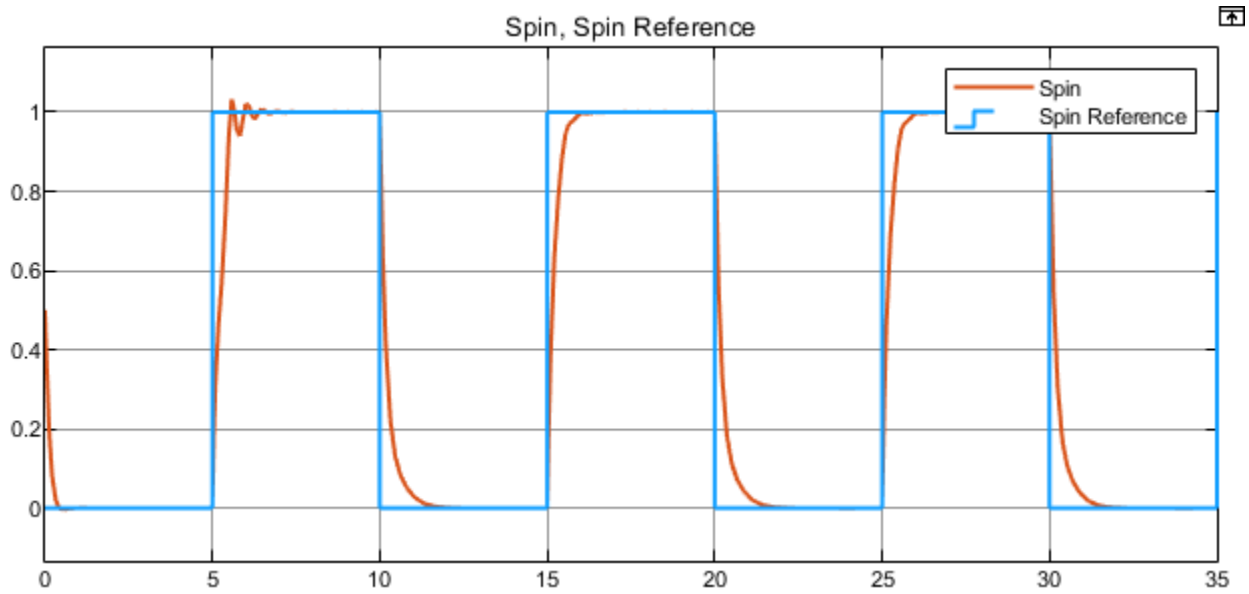
```
mdl = "satellitespin";
open_system(mdl)
```

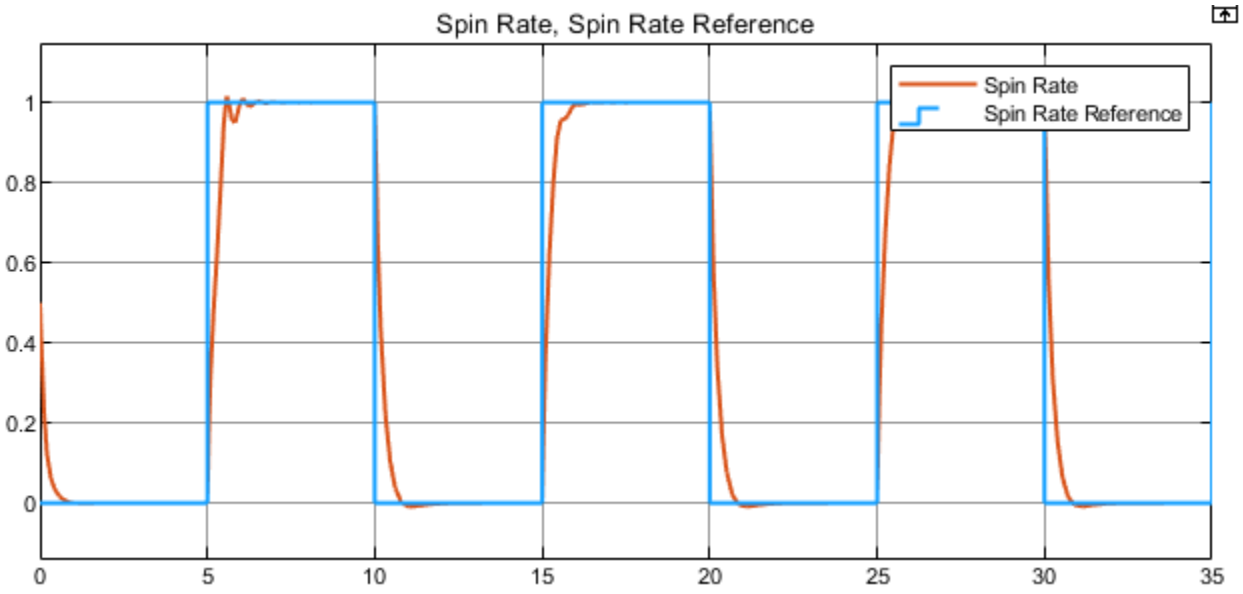


Simulate the model.

```
Tf = 35;          % duration (s)
sim(mdl);
```

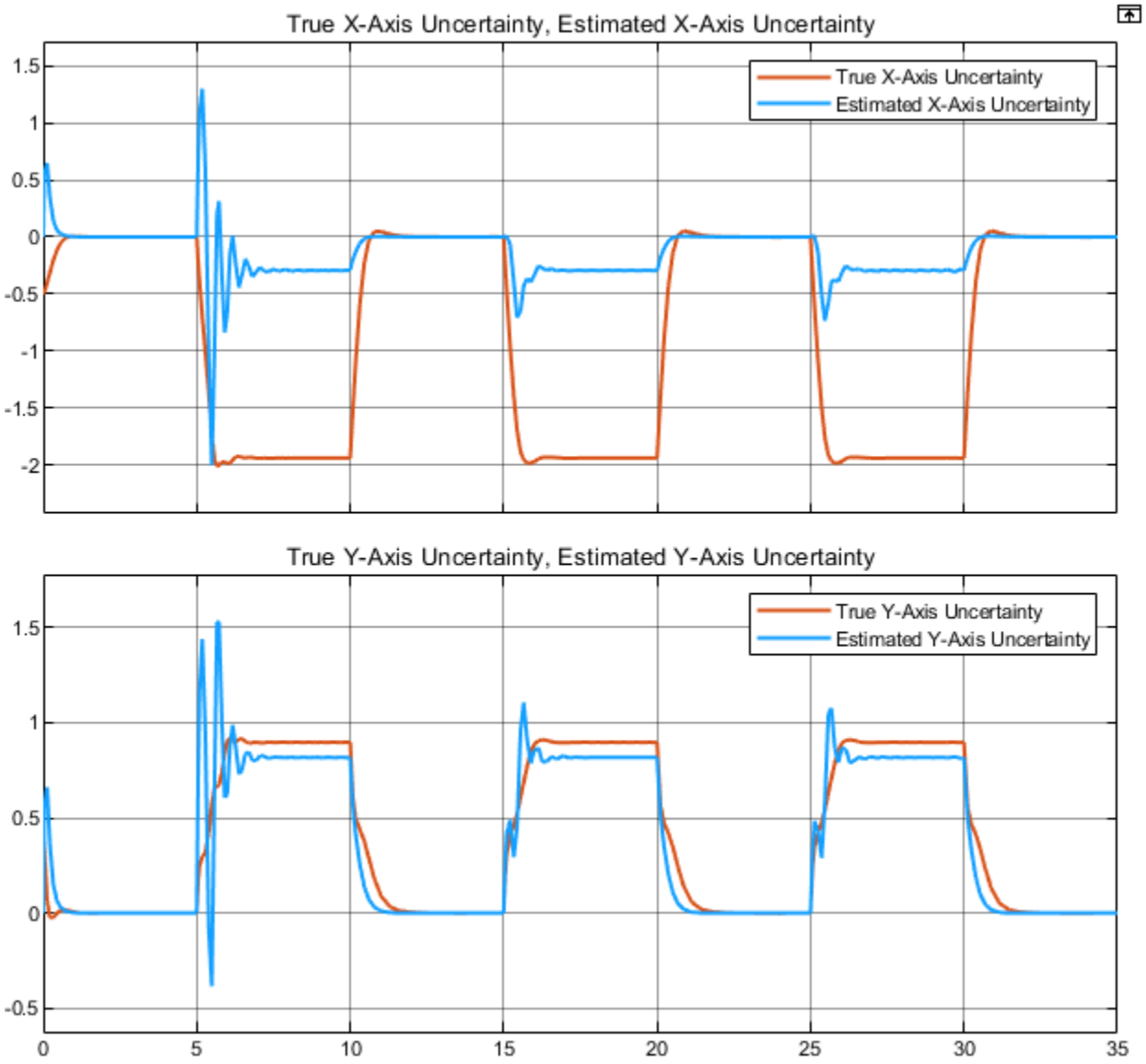
View the controller performance by comparing the spin response (top plot) and spin-rate response (bottom plot) to their respective reference signals.





The MRAC controller tracks the reference signals well.

View the uncertainty estimated by the controller and compare it to the true uncertainty.

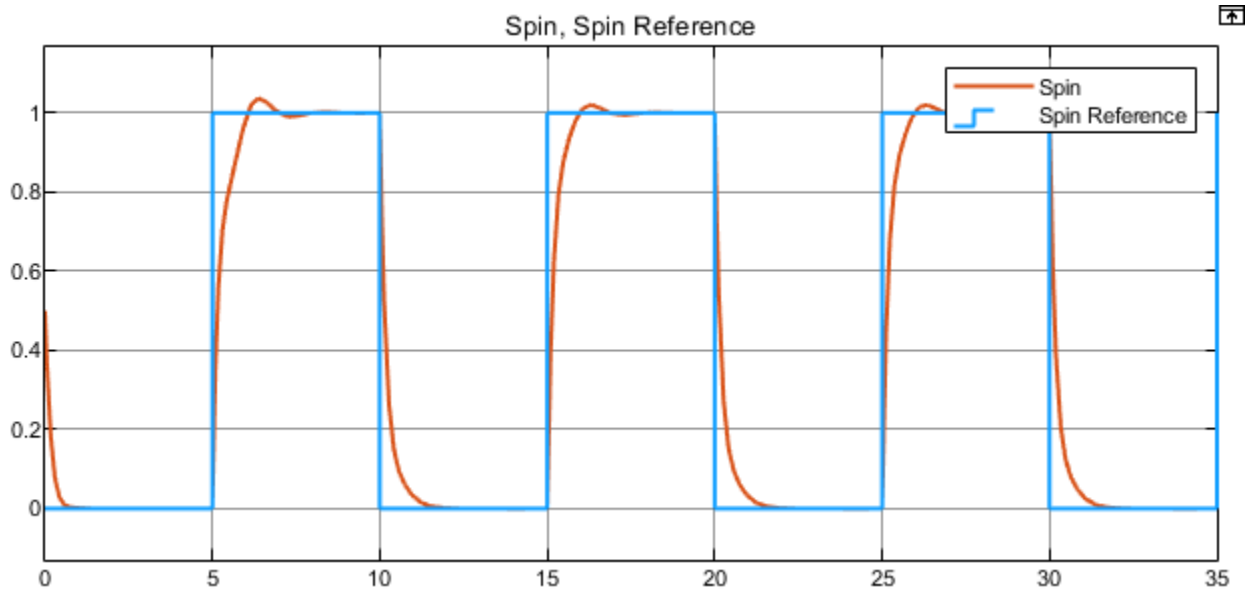


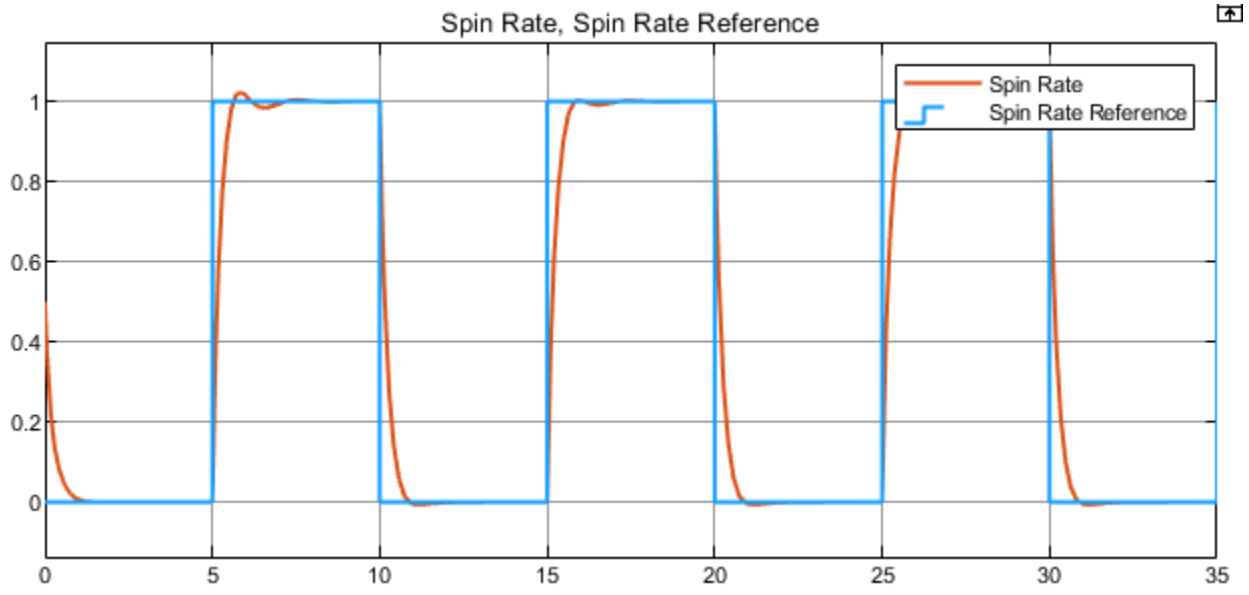
The controller estimates an accurate y-axis uncertainty model. The uncertainty model for the x-axis has the correct sign, though the magnitude is lower than that of the true uncertainty. Both uncertainty models show significant transients in the uncertainty estimate, especially during the first step change in the reference signals.

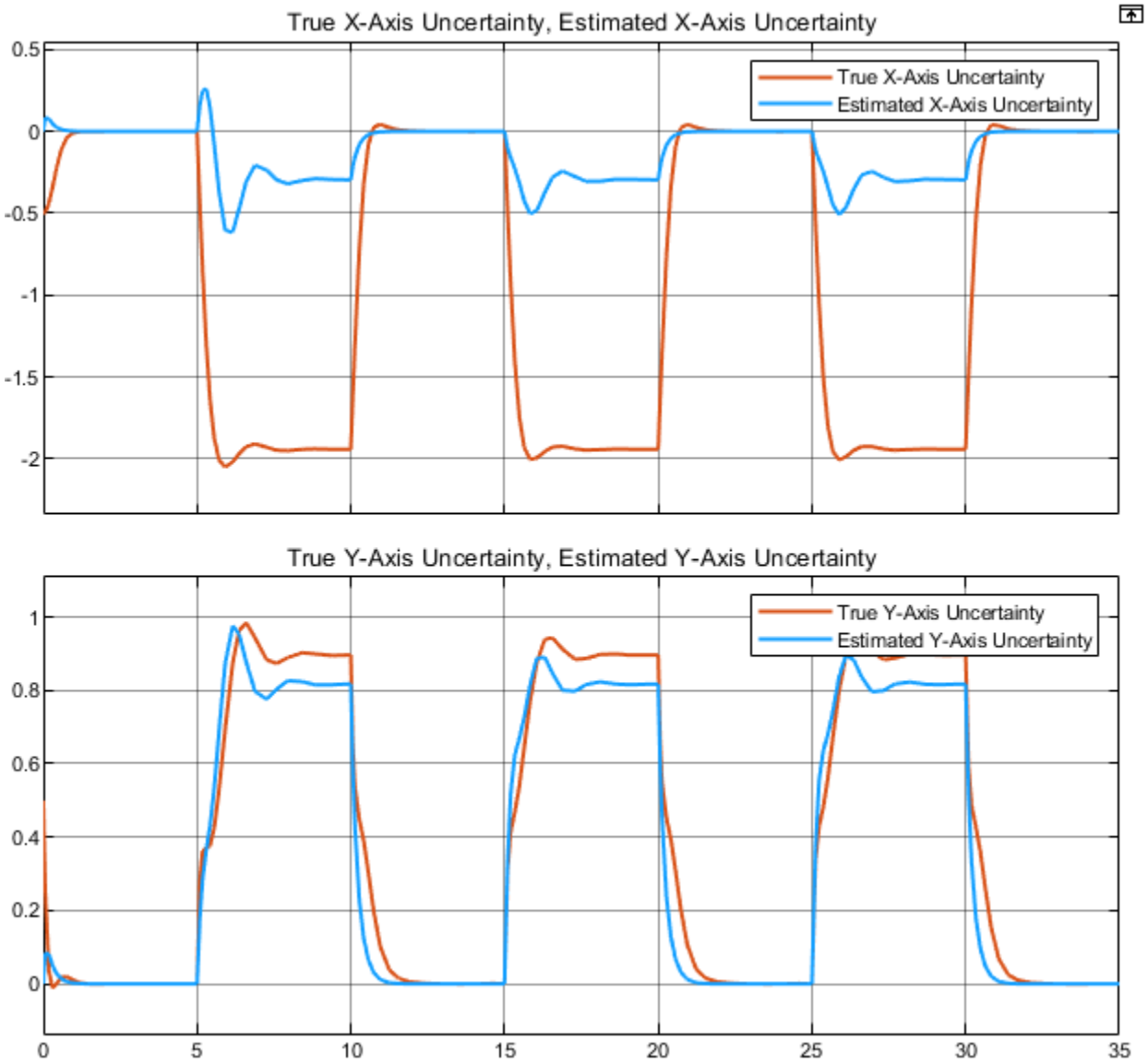
To reduce the transients in the estimated uncertainty responses, you can reduce the value of the tracking error rate `Q`. For example, reduce the value of `Q` to 1 and simulate the model.

```
Q = 1;
sim(mdl);
```

There are fewer transients in the model estimate responses. The trade off is that the spin and spin-rate responses have longer settling times.


Spin, Spin Reference
Spin
Spin Reference
1
0.8
0.6
0.4
0.2
0
0
5
10
15
20
25
30
35


Spin Rate, Spin Rate Reference
Spin Rate
Spin Rate Reference
1
0.8
0.6
0.4
0.2
0
0
5
10
15
20
25
30
35



Reference

[1] Zhou, K., J.C.Doyle, and K. Glover. *Robust and Optimal Control*, Englewood Cliffs, NJ: Prentice-Hall, 1996, Section 9.6.

See Also

Blocks

Model Reference Adaptive Control

Related Examples

- “Model Reference Adaptive Control” on page 15-28
- “Model Reference Adaptive Control of Aircraft Undergoing Wing Rock” on page 15-41

Model Reference Adaptive Control of Aircraft Undergoing Wing Rock

This example shows how to control roll and roll rate of a delta wing aircraft undergoing wing rock. For this example, the system model is unknown. Therefore, you use model reference adaptive control (MRAC) to make the controlled system match an ideal reference model. The aircraft is modeled in Simulink® and the MRAC controller is implemented using the Model Reference Adaptive Control block provided by Simulink Control Design™ software.

Wing-Rock Control System

Wing rock is a phenomenon observed in delta wing aircraft flying at low speeds and high angles of attack. The aircraft experiences undesired roll oscillations that make the aircraft more difficult for the pilot to control. The goal of the MRAC controller is to cancel the undesired roll oscillation. A baseline controller can then be designed to achieve a desired reference behavior.

The following equations define the dynamics for the wing-rock model.

$$\dot{\theta} = p$$
$$\dot{p} = \Delta(x) + L_\delta \delta_a$$
$$\Delta(x) = w_0^* + w_1^*\theta + w_2^* p + w_3^*|\theta| + w_4^*|p|p + w_5^*\theta^3$$

Here:

- $x = (\theta, p)$ is the system state vector, where θ is the roll angle and p is the roll rate.
- δ_a is the aileron angle control input for the aircraft.
- L_δ is the control effective matrix for which you know at least the sign.
- $\Delta(x)$ is the wing-rock disturbance
- w_i^* are unknown ideal weights.

For this uncertain nonlinear system, your goal is to design a controller that enables the system to track the following second-order reference model.

$$\ddot{x}_m = -4x_m - 2\dot{x}_m + 4r(t)$$

Here:

- x_m is the reference model state vector.
- $r(t)$ is the roll reference signal provided by the pilot.

Define Nominal and Reference Models

Specify the following simplified second-order nominal model for the roll dynamics.

```
A = [0 1; 0 0]; % Second intergrator model
B = [0;1];      % Nominal control effective matrix
```

Define the reference model as the stable second-order system defined previously. The controller adapts its uncertainty model to achieve the same second-order behavior as this model.

```
Am = [0 1; -4 -2]; % Second intergrator model
Bm = [0;4];        % Nominal control effective matrix
```

Specify the initial conditions for the nominal and reference models.

```
theta_0 = 0;   % Iitial roll angle (rad)
p0 = 0;        % Initial roll rate (rad/s)
xm = [0;0];    % Initial condition of the reference plant model
```

Model Reference Adaptive Control Structure

The goal of the MRAC controller is to achieve asymptotic convergence of the tracking error $e(t) = x(t) - x_m(t)$.

$$\lim_{t \to \infty} (x(t) - x_m(t)) \to 0$$

The MRAC controller has the following structure.

$$u(t) = -k_x x(t) + k_r r(t) - u_{ad}$$

Here:

- k_x contains feedback control gains.
- k_r contains feedforward control gains.
- u_{ad} is an adaptive control term that cancels the model uncertainty.

The Model Reference Adaptive Control block adjusts the adaptive control term to achieve the desired reference model tracking. Optionally, you can also adapt the feedback and feedforward control gains Though, for this example, the control gains are static.

The static feedback and feedforward control gains for the wing-rock system are computed to satisfy the following model-matching condition.

$$A_m = A - Bk_x$$
$$B_m = Bk_r$$

Specify the computed controller gains.

```
Kx = [-4 -2];   % Feedback gain
Kr = 4;         % Feedforward gain
```

Configure Uncertainty Estimation Parameters

The MRAC controller estimates the model uncertainty online and generates an adaptive control action u_{ad} that cancels the uncertainty to recover the nominal system for the baseline controller. The adaptive control term models the system uncertainty using the following model.

$$u_{ad} = \mathbf{w}^T \phi(x)$$

Here:

- $\mathbf{w}$ contains the network weights that are adjusted by the controller.
- $\phi(x)$ is the uncertainty model feature vector.

Using the Model Reference Adaptive Control block, you can select one of the following feature vector definitions.

- System states, where $\phi(x) = x(t)$ — This approach is the simplest option, which can be a good starting point if you do not know the complexity of the system uncertainty. If you find that using the states as features does not adequately represent the nonlinear uncertainty, select one of the other approaches.
- Radial basis functions with Gaussian kernels, where $\phi(x) = \left[\exp\left(-\frac{(x-c_i)^2}{\sigma_i}\right)\right]_{i=1}^{N}$. You can configure the kernels by defining the feature centers c_i and bandwidth σ_i.
- Custom features provided by an optional input port.

For this example, you first configure the controller to use the system states as the disturbance model features. You then simulate the system using radial basis functions as the model features.

Specify the parameters for the radial basis function kernels.

```
N = 20;          % Number of RBF kernels
cen_max = 1;     % Upper limit for kernel centers
cen_min = -1;    % Lower limit for kernel centers
bandwidth = 5;   % Kernel bandwidth
```

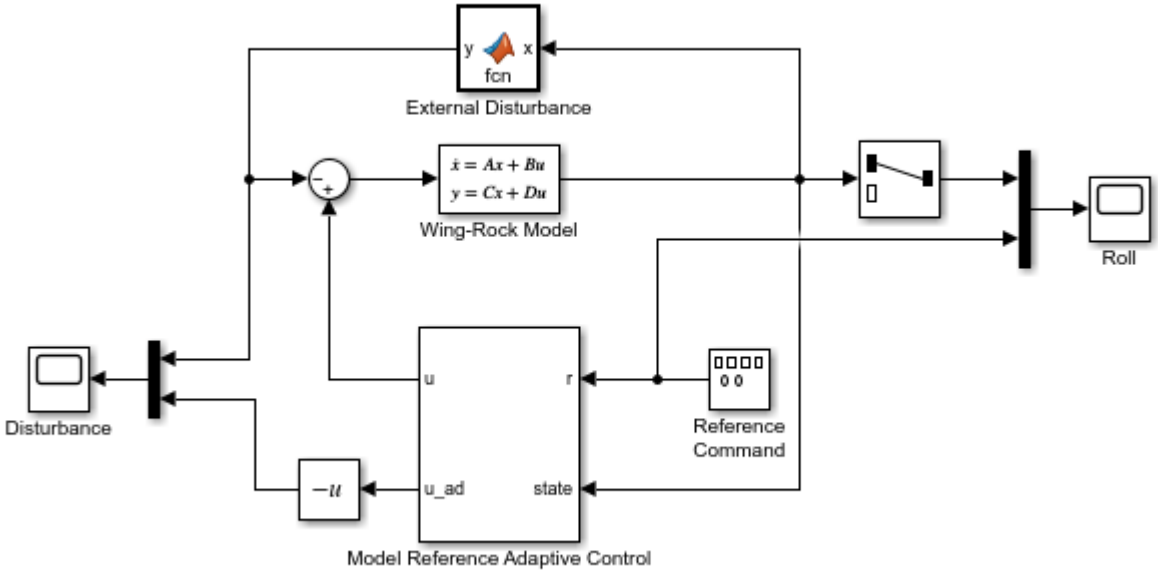
Define the model estimation learning rate `gamma_w` and tracking error weight `Q`.

```
gamma_w = 100;  % Learning rate
Q = 1;          % Tracking error weight
```

Simulate Controller

Open the Simulink model of the wing-rock control system.

```
mdl = "wingrock";
open_system(mdl)
```



In this model:

- The Wing-Rock Model block implements the nominal model of the roll dynamics.
- The External Disturbance block generates a wing-rock disturbance to the roll dynamics.
- The Reference Command block generates pilot reference signal.
- The Model Reference Adaptive Control block outputs the control action u_{ad}, which is an estimate of the wing-rock disturbance.

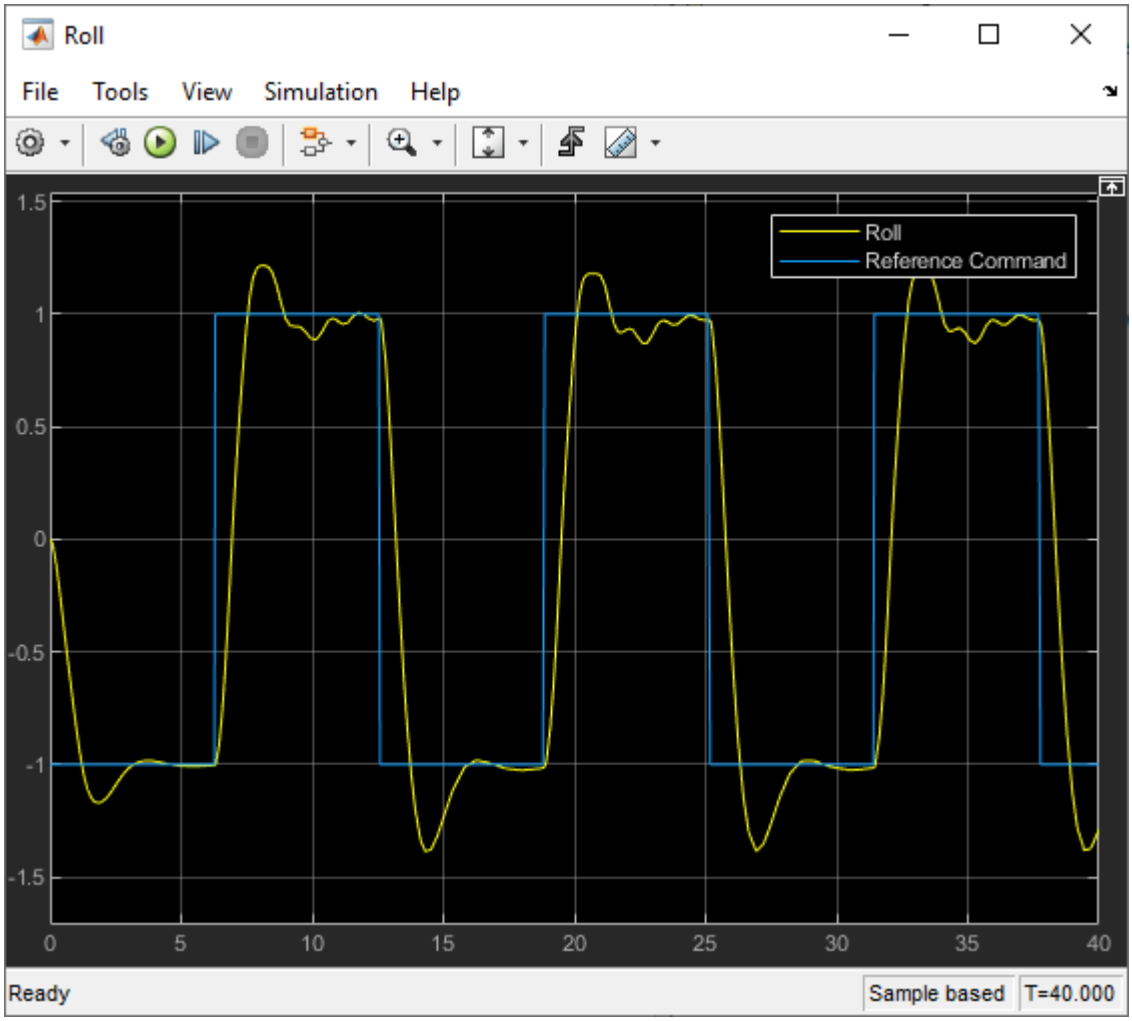
The Model Reference Adaptive Control block is initially configured to use the states from the Wing-Rock Model block as the disturbance model features.

Set the simulation duration and simulate the model.

```
Tf = 40;    % Simulation duration (s)
sim(mdl);
```

Compare the resulting aircraft roll to the reference command.

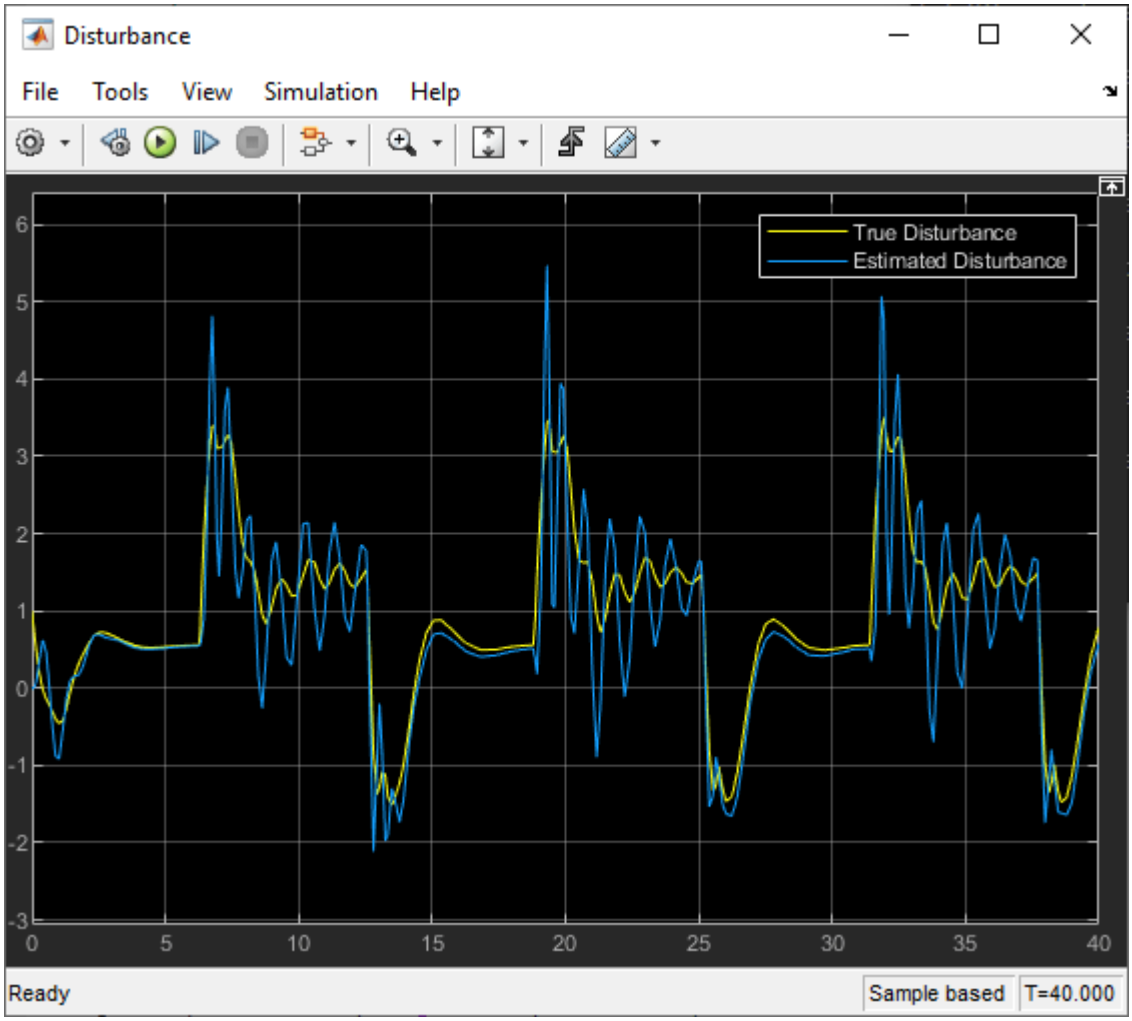
```
open_system(mdl + "/Roll")
```



The controller does not achieve a smooth second-order transient response.

Compare the disturbance model estimated by the MRAC controller with the true disturbance signal.

```
open_system(mdl + "/Disturbance")
```



When using the states as the disturbance model features, the controller does not accurately estimate the disturbance model.

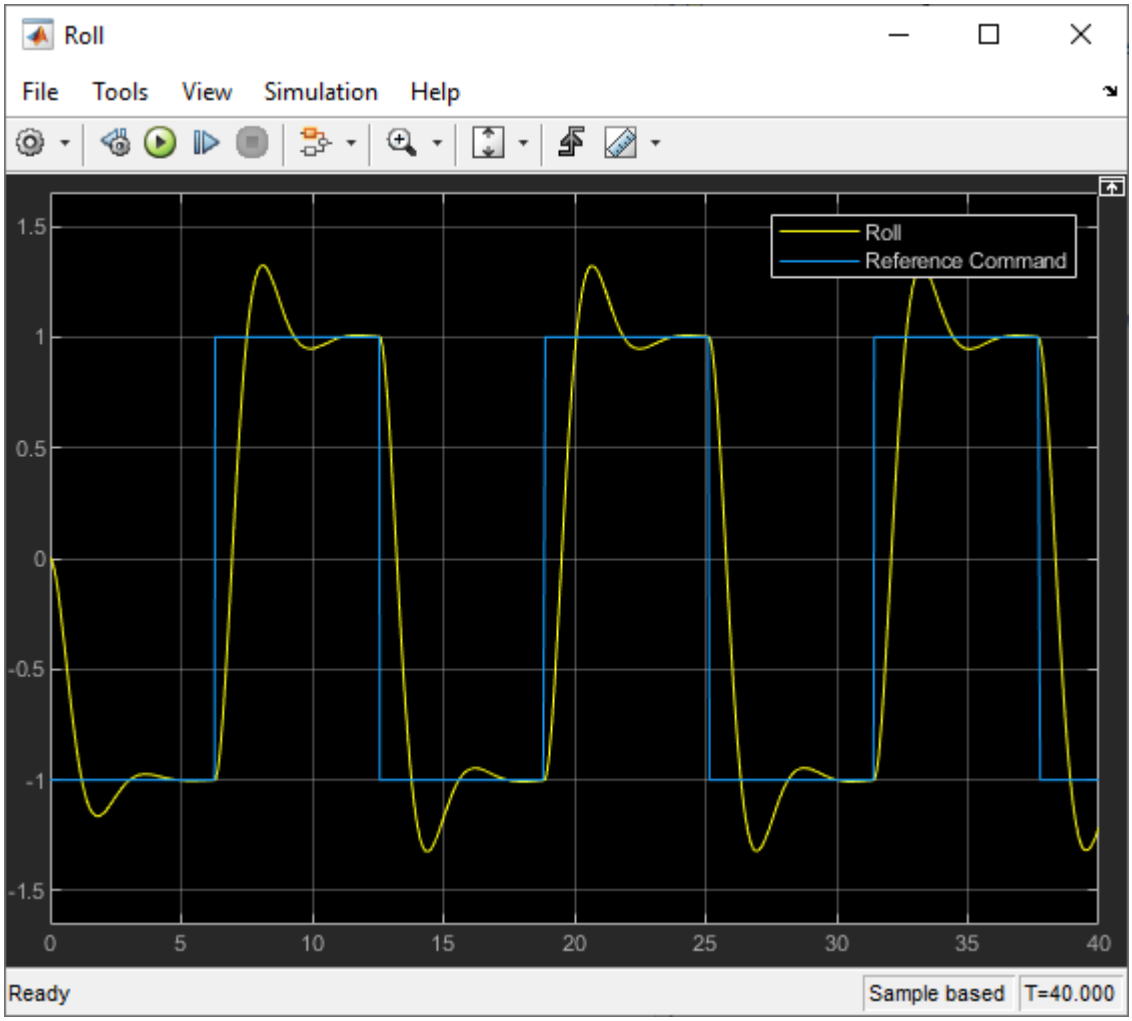
Modify the MRAC controller to use 20 radial basis functions with centers spanning the range [-1 1]. Use the same bandwidth for all the basis function kernels.

```
controller = mdl + "/Model Reference Adaptive Control";
set_param(controller,'FeatureTypeOptions','Radial Basis Function')
set_param(controller,'nCen','N')
set_param(controller,'cSpanMin','cen_min')
set_param(controller,'cSpanMax','cen_max')
set_param(controller,'cSig','bandwidth')
```

Simulate the model.

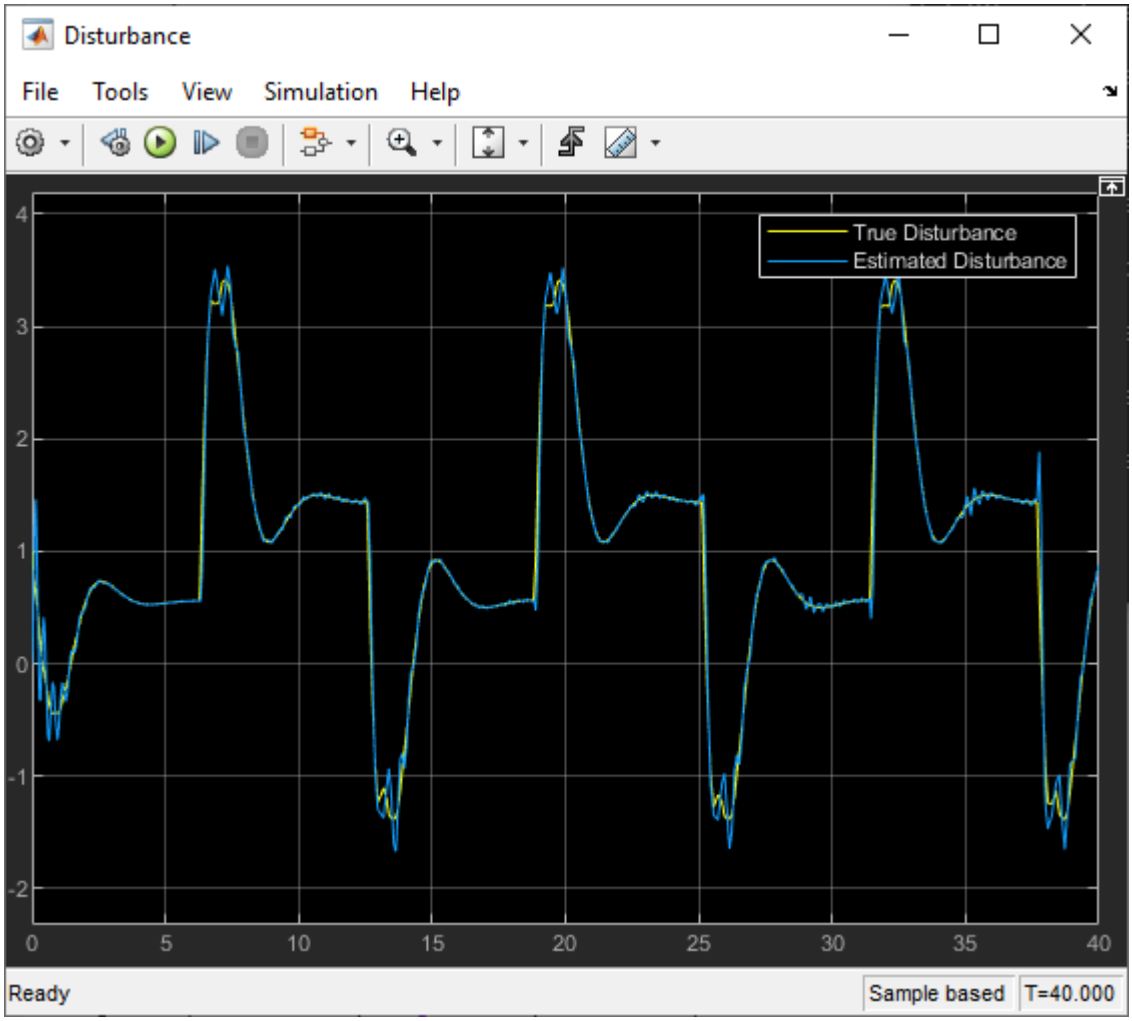
```
sim(mdl);
```

View the resulting controller performance.



The controller achieves a smoother second-order response for changes in the reference command.

Compare the disturbance estimated using the radial basis functions to the true disturbance.



As expected, using the radial basis functions as disturbance model features produces a more accurate model of the nonlinear disturbance.

See Also

Blocks
Model Reference Adaptive Control

Related Examples

- “Model Reference Adaptive Control” on page 15-28
- “Model Reference Adaptive Control of Satellite Spin” on page 15-33

Constraint Enforcement

- “Constraint Enforcement for Control Design” on page 16-2
- “Enforce Constraints for PID Controllers” on page 16-4
- “Learn and Apply Constraints for PID Controllers” on page 16-10
- “Train Reinforcement Learning Agent with Constraint Enforcement” on page 16-18
- “Train RL Agent for Adaptive Cruise Control with Constraint Enforcement” on page 16-29
- “Train RL Agent for Lane Keeping Assist with Constraint Enforcement” on page 16-39

Constraint Enforcement for Control Design

Some control applications require the controller to select control actions such that the plant states do not violate certain critical constraints. In many cases, the constraints are on plant states that the controller does not control directly. Instead, you define a constraint function that defines the constraint in terms of the control action signal. This constraint function can be a known relationship or one that you must learn from experimental data.

Constraint Enforcement Block

The Constraint Enforcement block, which requires Optimization Toolbox software, computes the modified control actions that are closest to specified control actions subject to constraints and action bounds. The block uses a quadratic programming (QP) solver to find the control action u that minimizes the function $|u - u_0|^2$ in real time. Here, u_0 is the unmodified control action from the controller.

The solver applies the following constraints to the optimization problem.

$$f_x + g_x u \le c$$

$$u_{\min} \le u \le u_{\max}$$

Here:

- f_x and g_x are coefficients of the constraint function which depend on the plant states x.
- c is a bound for the constraint function.
- $u_{\min}$ is a lower bound for the control action.
- $u_{\max}$ is an upper bound for the control action.

Since the Constraint Enforcement block modifies the original control action, the final closed-loop system might not achieve the design objectives of the original controller, such as stability margins.

You must verify that the combined controller and Constraint Enforcement block meet your original control objectives. If the system does not meet your original objectives, consider updating your original controller design. For example, you can add additional gain and phase margin to compensate for any potential performance degradation.

Constraint Function Coefficients

Depending on your application, the coefficients f_x and g_x of the constraint function can be linear or nonlinear functions of the plant states and can be either known or unknown.

For an example that uses known nonlinear constraint function coefficients, see “Enforce Constraints for PID Controllers” on page 16-4. This example derives the constraint function from the plant dynamics.

When you are unable to derive the constraint function from the plant directly, you must learn the coefficients using input/output data from experiments or simulations. To learn such constraints, you can create a function approximator and tune the approximator to reproduce the input-to-output mapping from simulation or experimental data.

To learn linear coefficient functions, you can find a least-squares solution from the data. For examples that use this approach, see “Train RL Agent for Adaptive Cruise Control with Constraint

Enforcement” on page 16-29 and “Train RL Agent for Lane Keeping Assist with Constraint Enforcement” on page 16-39.

For nonlinear coefficient functions, you must tune a nonlinear function approximator. Examples of such approximators include:

- Deep neural networks (requires Deep Learning Toolbox™ software)
- Nonlinear identified system models (requires System Identification Toolbox software)
- Fuzzy inference systems (requires Fuzzy Logic Toolbox™ software)

For examples that learn nonlinear coefficient function by training a deep neural network, see “Learn and Apply Constraints for PID Controllers” on page 16-10 and “Train Reinforcement Learning Agent with Constraint Enforcement” on page 16-18.

See Also

Blocks
Constraint Enforcement

Related Examples

- “Enforce Constraints for PID Controllers” on page 16-4
- “Learn and Apply Constraints for PID Controllers” on page 16-10
- “Train Reinforcement Learning Agent with Constraint Enforcement” on page 16-18

Enforce Constraints for PID Controllers

This example shows how to enforce known constraints for a PID controller application using the Constraint Enforcement block.

Overview

For this example, the plant dynamics are described by the following equations [1].

$$\dot{x}_1 = -x_1 + (x_1^2 + 1)u_1$$
$$\dot{x}_2 = -x_2 + (x_2^2 + 1)u_2$$

The goal for the plant is to track desired trajectories, defined as:

$$\dot{\theta} = 0.1\pi$$
$$\dot{x}_{1d} = -r\cos(\theta)$$
$$\dot{x}_{2d} = r\sin(\theta)$$

For an example that learns and applies an unknown constraint function for the same PID control application, see “Learn and Apply Constraints for PID Controllers” on page 16-10.

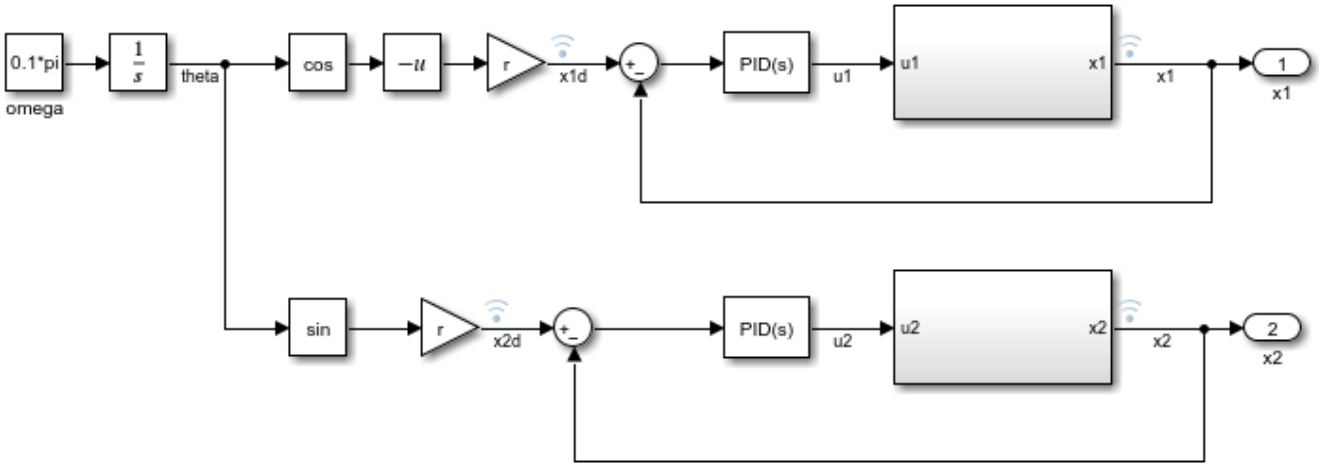
Configure model parameters and initial conditions.

```
r = 1.5;   % Radius for desired trajectory
Ts = 0.1;  % Sample time
Tf = 22;   % Duration
x0_1 = -r; % Initial condition for x1
x0_2 = 0;  % Initial condition for x2
```

Design PID Controllers

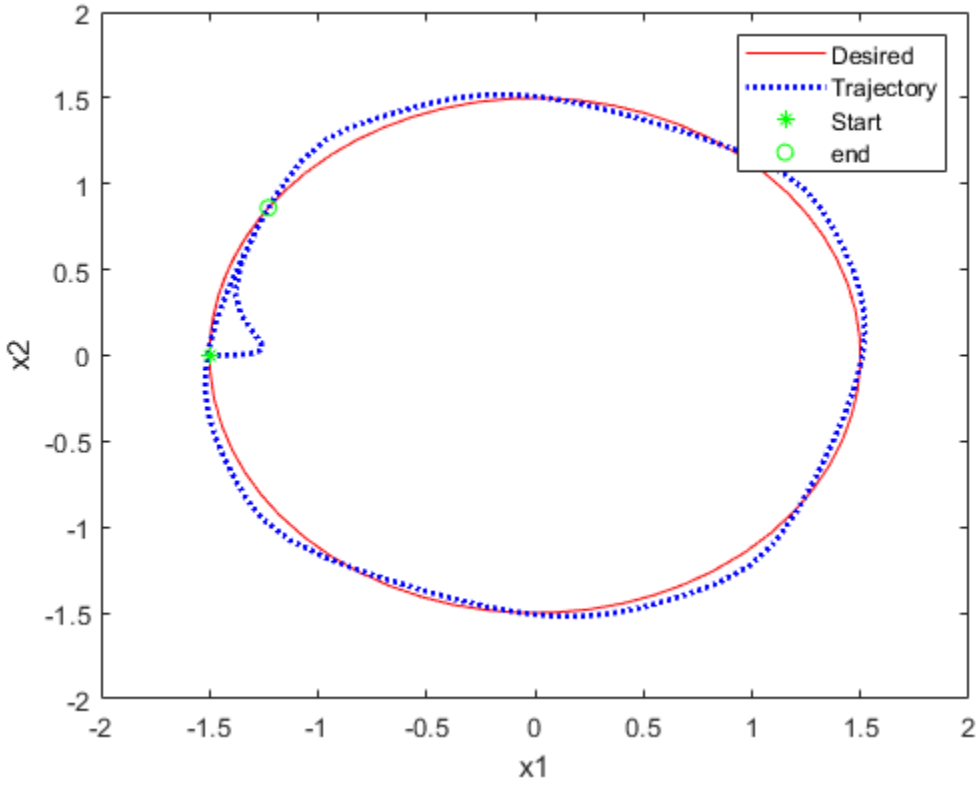
Before applying constraints, design PID controllers for tracking the reference trajectories. The `trackingWithPIDs` model contains two PID controllers with gains tuned using the PID Tuner app. For more information on tuning PID controllers in Simulink models, see “Introduction to Model-Based PID Tuning in Simulink” on page 7-2.

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



Simulate the PID controllers and plot their tracking performance.

```
% Simulate the model.
out = sim(mdl);

% Extract trajectories.
logData = out.logsout;
x1_traj = logData{3}.Values.Data;
x2_traj = logData{4}.Values.Data;
x1_des = logData{1}.Values.Data;
x2_des = logData{2}.Values.Data;

% Plot trajectories.
figure('Name','Tracking')
xlim([-2,2])
ylim([-2,2])
plot(x1_des,x2_des,'r')
xlabel('x1')
ylabel('x2')
hold on
plot(x1_traj,x2_traj,'b:','LineWidth',2)
hold on
plot(x1_traj(1),x2_traj(1),'g*')
hold on
plot(x1_traj(end),x2_traj(end),'go')
legend('Desired','Trajectory','Start','end')
```



Constraint Function

For this example, you apply known constraints to the application using the Constraint Enforcement block, which adjusts control actions to satisfy a constraint function.

In this example, the feasible region for the plant is given by $\{x : x_1 \le 1, x_x \le 1\}$. Therefore, the plant next-state condition $x_{k+1} = \begin{bmatrix} x_1(k+1) \\ x_2(k+1) \end{bmatrix}$ must satisfy $x_{k+1} \le \begin{bmatrix} 1 \\ 1 \end{bmatrix}$.

You can approximate the plant dynamics by the following equation.

$$x_{k+1} \approx \begin{bmatrix} (1 - T_s) & 0 \\ 0 & (1 - T_s) \end{bmatrix} x_k + \begin{bmatrix} T_s\left(1 + x_1^2(k)\right) & 0 \\ 0 & T_s\left(1 + x_2^2(k)\right) \end{bmatrix} u_k$$

Applying the constraints to this equation produces the following constraint function.

$$\begin{bmatrix} (1 - T_s) & 0 \\ 0 & (1 - T_s) \end{bmatrix} x_k + \begin{bmatrix} T_s\left(1 + x_1^2(k)\right) & 0 \\ 0 & T_s\left(1 + x_2^2(k)\right) \end{bmatrix} u_k \le \begin{bmatrix} 1 \\ 1 \end{bmatrix}$$

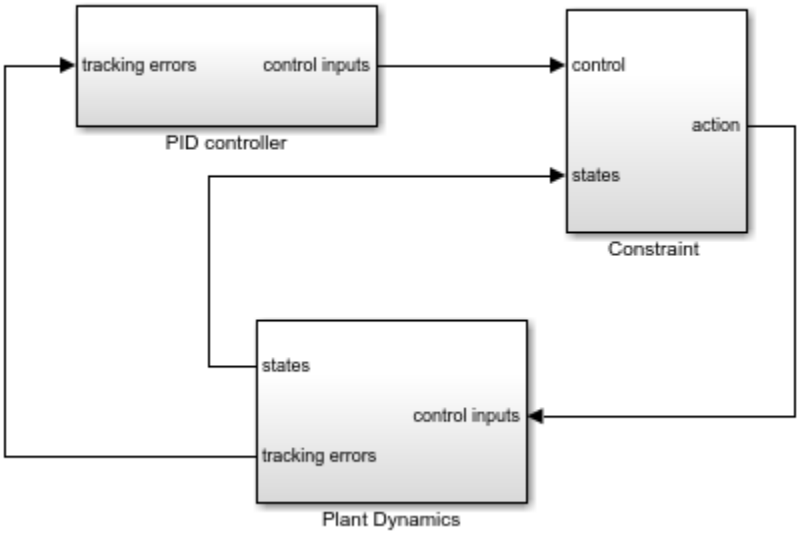
The Constraint Enforcement block accepts constraints of the form $f_x + g_x u \le c$. For this application, the coefficients of this constraint function are as follows.

$$f_x = \begin{bmatrix} (1 - T_s) & 0 \\ 0 & (1 - T_s) \end{bmatrix} x_k,\ g_x = \begin{bmatrix} T_s\left(1 + x_1^2(k)\right) & 0 \\ 0 & T_s\left(1 + x_2^2(k)\right) \end{bmatrix},\ c = \begin{bmatrix} 1 \\ 1 \end{bmatrix}$$

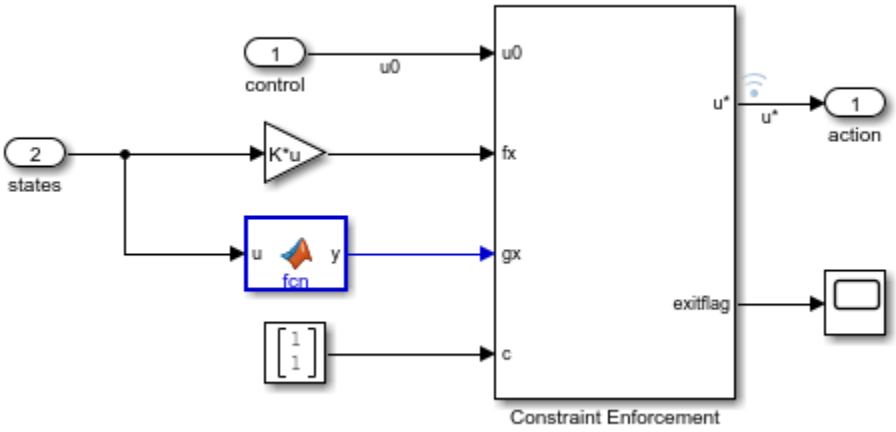
Simulate PID Controller with Constraint Enforcement

The `trackingWithConstraintPID` model contains the PID controllers, the plant dynamics and the constraint implementation.

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

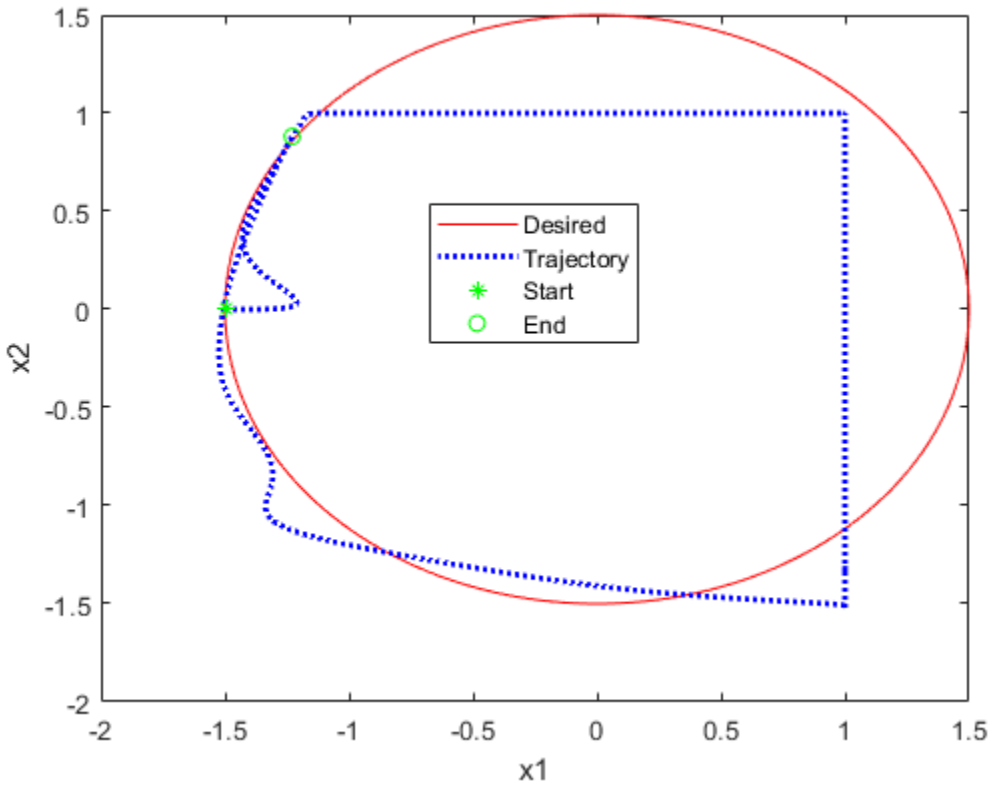


To view the constraint implementation, open the Constraint subsystem. Here, the model implements the known constraint function using a MATLAB Function block, and the Constraint Enforcement block enforces the constraint function.



Run the model and plot the simulation results. The plot shows that the plant states are less than one.

```
% Simulate the model.
out = sim(mdl);

% Extract trajectories.
logData = out.logsout;
x1_traj = zeros(size(out.tout));
x2_traj = zeros(size(out.tout));
for ct = 1:size(out.tout,1)
    x1_traj(ct) = logData{4}.Values.Data(:,:,ct);
    x2_traj(ct) = logData{5}.Values.Data(:,:,ct);
end

x1_des = logData{2}.Values.Data;
x2_des = logData{3}.Values.Data;

% Plot trajectories.
figure('Name','Tracking with Constraint');
plot(x1_des,x2_des,'r')
xlabel('x1')
ylabel('x2')
hold on
plot(x1_traj,x2_traj,'b:','LineWidth',2)
hold on
plot(x1_traj(1),x2_traj(1),'g*')
hold on
plot(x1_traj(end),x2_traj(end),'go')
legend('Desired','Trajectory','Start','End','Location','best')
```



The Constraint Enforcement block successfully constrains the control actions such that the plant states remain less than one.

```
bdclose('trackingWithPIDs')
bdclose('trackingWithKnownConstraintPID')
```

References

[1] Robey, Alexander, Haimin Hu, Lars Lindemann, Hanwen Zhang, Dimos V. Dimarogonas, Stephen Tu, and Nikolai Matni. "Learning Control Barrier Functions from Expert Demonstrations." Preprint, submitted April 7, 2020. https://arxiv.org/abs/2004.03315

See Also

Blocks
Constraint Enforcement

Apps
PID Tuner

Related Examples

- “Constraint Enforcement for Control Design” on page 16-2
- “Learn and Apply Constraints for PID Controllers” on page 16-10
- “Train Reinforcement Learning Agent with Constraint Enforcement” on page 16-18
- “Introduction to Model-Based PID Tuning in Simulink” on page 7-2

Learn and Apply Constraints for PID Controllers

This example shows how to learn constraints from data and apply these constraints for a PID control application. First, you learn the constraint function using a deep neural network, which requires Deep Learning Toolbox™ software. You then apply the constraints to the PID control actions using the Constraint Enforcement block.

For this example, the plant dynamics are described by the following equations [1].

$$\dot{x}_1 = -x_1 + (x_1^2 + 1)u_1$$
$$\dot{x}_2 = -x_2 + (x_2^2 + 1)u_2$$

The goal for the plant is to track the following trajectories.

$$\dot{\theta} = 0.1\pi$$
$$\dot{x}_{1d} = -r\cos(\theta)$$
$$\dot{x}_{2d} = r\sin(\theta)$$

For an example that applies a known constraint function to the same PID control application, see “Enforce Constraints for PID Controllers” on page 16-4.

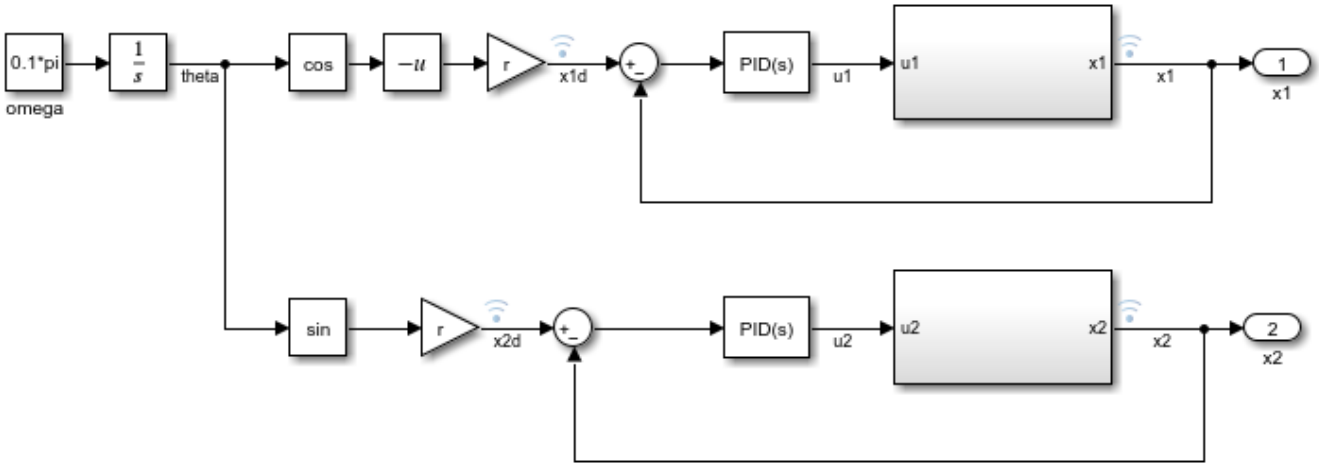
Set the random seed and configure model parameters and initial conditions.

```
rng(0);            % Random seed
r = 1.5;           % Radius for desired trajectory
Ts = 0.1;          % Sample time
Tf = 22;           % Duration
x0_1 = -r;         % Initial condition for x1
x0_2 = 0;          % Initial condition for x2
maxSteps = Tf/Ts; % Simulation steps
```

Design PID Controllers

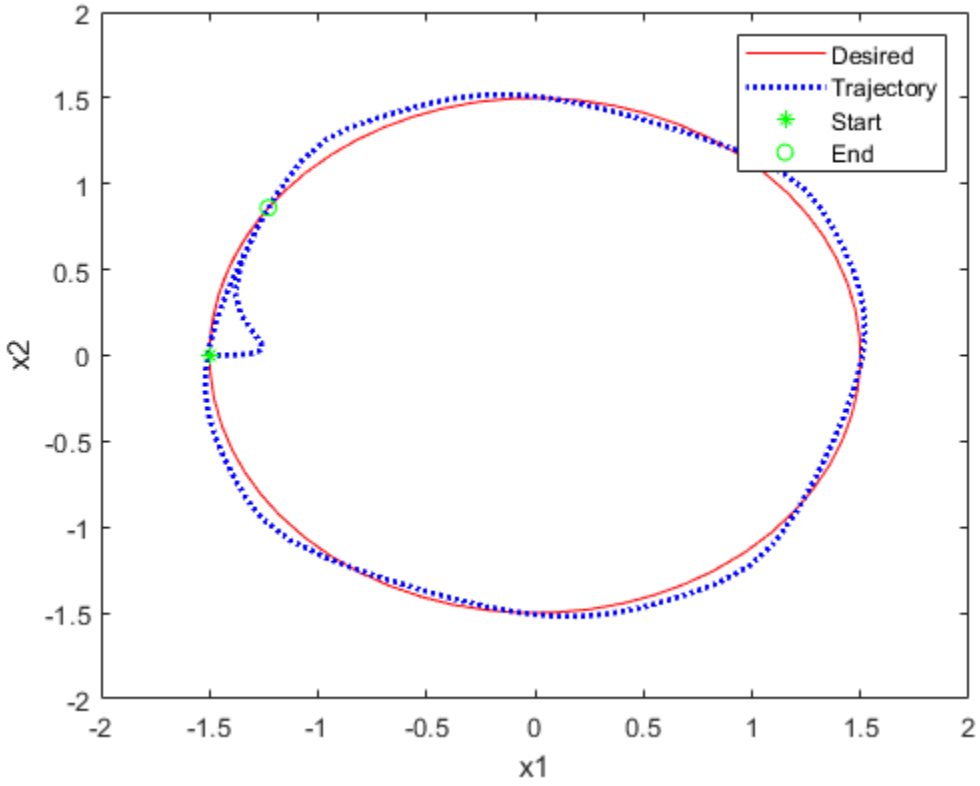
Before learning and applying constraints, design PID controllers for tracking the reference trajectories. The `trackingWithPIDs` model contains two PID controllers with gains tuned using the PID Tuner app. For more information on tuning PID controllers in Simulink models, see “Introduction to Model-Based PID Tuning in Simulink” on page 7-2.

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



Simulate the PID controllers and plot their tracking performance.

```
% Simulate the model.
out = sim(mdl);

% Extract trajectories.
logData = out.logsout;
x1_traj = logData{3}.Values.Data;
x2_traj = logData{4}.Values.Data;
x1_des = logData{1}.Values.Data;
x2_des = logData{2}.Values.Data;

% Plot trajectories.
figure('Name','Tracking')
xlim([-2,2])
ylim([-2,2])
plot(x1_des,x2_des,'r')
xlabel('x1')
ylabel('x2')
hold on
plot(x1_traj,x2_traj,'b:','LineWidth',2)
hold on
plot(x1_traj(1),x2_traj(1),'g*')
hold on
plot(x1_traj(end),x2_traj(end),'go')
legend('Desired','Trajectory','Start','End')
```



Constraint Function

In this example, you learn application constraints and modify the control actions of the PID controllers to satisfy these constraints.

The feasible region for the plant is given by the constraints $\{x: x_1 \le 1, x_2 \le 1\}$. Therefore, the trajectories $x_{k+1} = [x_1(k+1)\, x_2(k+1)]'$ must satisfy $x_{k+1} \le 1$.

You can approximate the plant dynamics by the following equation.

$$x_{k+1} \approx Ax_k + \begin{bmatrix} g_1(x_k) & 0 \\ 0 & g_2(x_k) \end{bmatrix} u_k$$

Applying the constraints to this equation produces the following constraint function.

$$Ax_k + \begin{bmatrix} g_1(x_k) & 0 \\ 0 & g_2(x_k) \end{bmatrix} u_k \le \begin{bmatrix} 1 \\ 1 \end{bmatrix}$$

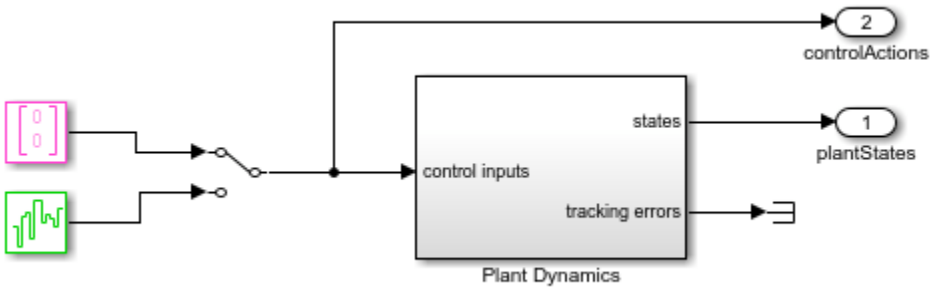
The Constraint Enforcement block accepts constraints of the form $f_x + g_x u \le c$. For this application, the coefficients of this constraint function are as follows.

$$f_x = Ax_k\,,\ g_x = \begin{bmatrix} g_1(x_k) & 0 \\ 0 & g_2(x_k) \end{bmatrix},\ c = \begin{bmatrix} 1 \\ 1 \end{bmatrix}$$

Learn Constraint Function

For this example, the coefficients of the constraint function are unknown. Therefore, you must derive them from training data. To collect training data, use the `rlCollectDataPID` model. This model allows you to pass either zero or random inputs to the plant model and record the resulting plant outputs.

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



To learn the unknown matrix A, set the plant inputs to zero. When you do so, the plant dynamics become $x_{k+1} \approx Ax_k$.

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

Collect training data using the `collectDataPID` helper function. This function simulates the model multiple times and extracts the input/output data. The function also formats the training data into an array with six columns: $x_1(k)$, $x_2(k)$, $u_1(k)$, $u_2(k)$, $x_1(k+1)$, and $x_2(k+1)$.

```
numSamples = 1000;
data = collectDataPID(mdl,numSamples);
```

Find a least-squares solution for A using the input/output data.

```
inputData = data(:,1:2);
outputData = data(:,5:6);
A = inputData\outputData;
```

Collect the training data. To learn the unknown function g_x, configure the model to use random input values that follow a normal distribution.

```
% Configure model to use random input data.
blk = [mdl '/Manual Switch'];
set_param(blk,'sw','0');

% Collect data.
data = collectDataPID(mdl,numSamples);
```

Train a deep neural network to approximate the g_x function using the `trainConstraintPID` helper function. This function formats the data for training then creates and trains a deep neural network. Training a deep neural network requires Deep Learning Toolbox software.

The inputs to the deep neural network are the plant states. Create the input training data by extracting the collected state information.

```
inputData = data(:,1:2);
```

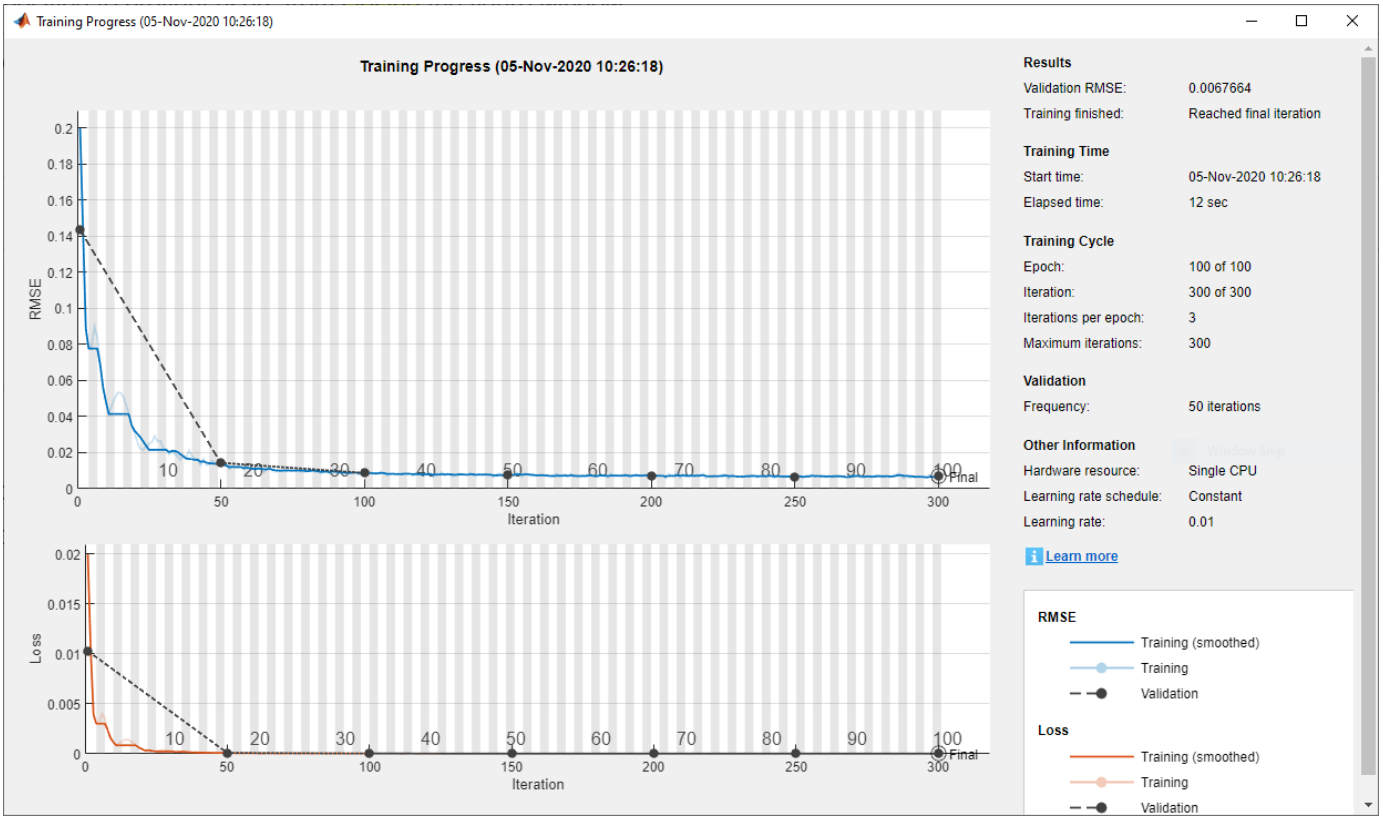
Since the output of the deep neural network corresponds to g_x, you must derive output training data using the collected input/output data and the computed A matrix.

```
u = data(:,3:4);
x_next = data(:,5:6);
fx = (A*inputData')';
outputData = (x_next - fx)./u;
```

For this example, to ensure reproducibility, load a pretrained network. To train the network yourself, set `trainConstraint` to `true`.

```
trainConstraint = false;
if trainConstraint
    network = trainConstraintPID(inputData,outputData);
else
   load trainedNetworkPID network
end
```

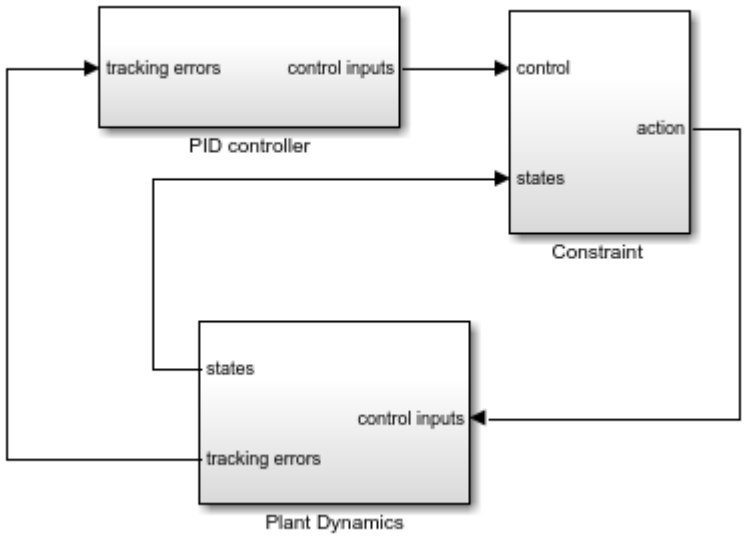
The following figure shows an example of the training progress.



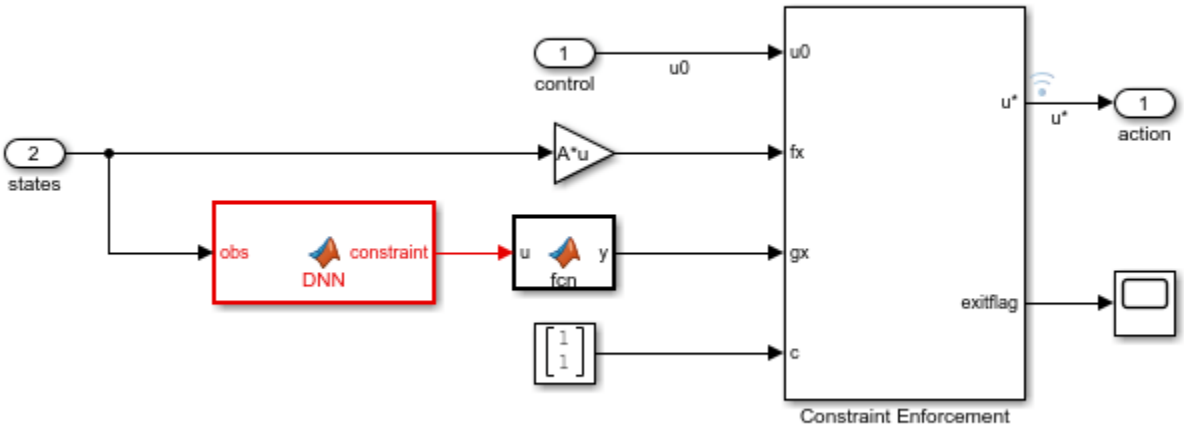
Simulate PID Controllers with Constraint Enforcement

To simulate the PID controllers with constraint enforcement, use the `trackingWithLearnedConstraintPID` model. This model constrains the controller outputs before applying them to the plant.

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

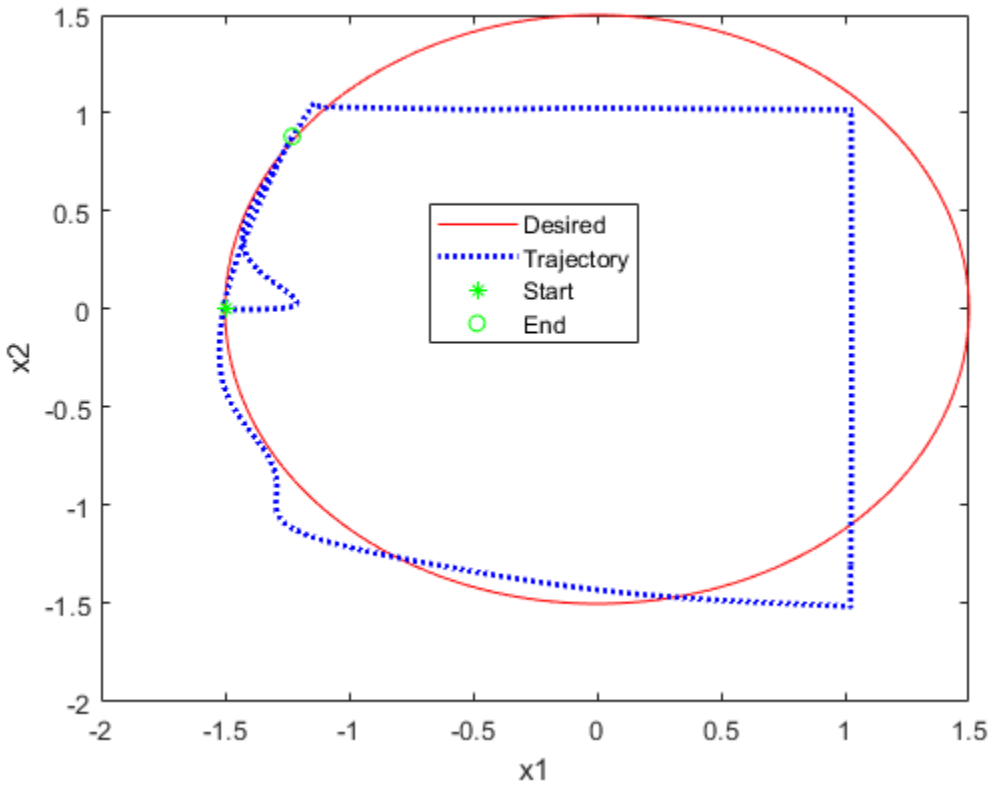


To view the constraint implementation, open the Constraint subsystem. Here, the trained deep neural network approximates g_x based on the current plant state, and the Constraint Enforcement block enforces the constraint function.



Simulate the model and plot the results.

```
% Simulate the model.
out = sim(mdl);

% Extract trajectories.
logData = out.logsout;
x1_traj = zeros(size(out.tout));
x2_traj = zeros(size(out.tout));
for ct = 1:size(out.tout,1)
    x1_traj(ct) = logData{4}.Values.Data(:,:,ct);
    x2_traj(ct) = logData{5}.Values.Data(:,:,ct);
end

x1_des = logData{2}.Values.Data;
x2_des = logData{3}.Values.Data;

% Plot trajectories.
figure('Name','Tracking with Constraint');
plot(x1_des,x2_des,'r')
xlabel('x1')
ylabel('x2')
hold on
plot(x1_traj,x2_traj,'b:','LineWidth',2)
hold on
plot(x1_traj(1),x2_traj(1),'g*')
hold on
plot(x1_traj(end),x2_traj(end),'go')
legend('Desired','Trajectory','Start','End','Location','best')
```



The Constraint Enforcement block successfully constrains the control actions such that the plant states remain less than one.

```
bdclose('rlCollectDataPID')
bdclose('trackingWithLearnedConstraintPID')
```

References

[1] Robey, Alexander, Haimin Hu, Lars Lindemann, Hanwen Zhang, Dimos V. Dimarogonas, Stephen Tu, and Nikolai Matni. "Learning Control Barrier Functions from Expert Demonstrations." Preprint, submitted April 7, 2020. https://arxiv.org/abs/2004.03315

See Also

Blocks
Constraint Enforcement

Apps
PID Tuner

Related Examples

- “Constraint Enforcement for Control Design” on page 16-2
- “Enforce Constraints for PID Controllers” on page 16-4
- “Train Reinforcement Learning Agent with Constraint Enforcement” on page 16-18
- “Introduction to Model-Based PID Tuning in Simulink” on page 7-2

Train Reinforcement Learning Agent with Constraint Enforcement

This example shows how to train a reinforcement learning (RL) agent with actions constrained using the Constraint Enforcement block. This block computes modified control actions that are closest to the actions output by the agent subject to constraints and action bounds. Training reinforcement learning agents requires Reinforcement Learning Toolbox™ software.

In this example, the goal of the agent is to bring a green ball as close as possible to the changing target position of a red ball [1].

The dynamics for the green ball from velocity v to position x are governed by Newton's law with a small damping coefficient τ.

$$\frac{1}{s(\tau s+1)}.$$

The feasible region for the ball position $0 \le x \le 1$ and the velocity of the green ball is limited to the range $[-1, 1]$.

The position of the target red ball is uniformly random across the range $[0, 1]$. The agent can observe only a noisy estimate of this target position.

Set the random seed and configure model parameters

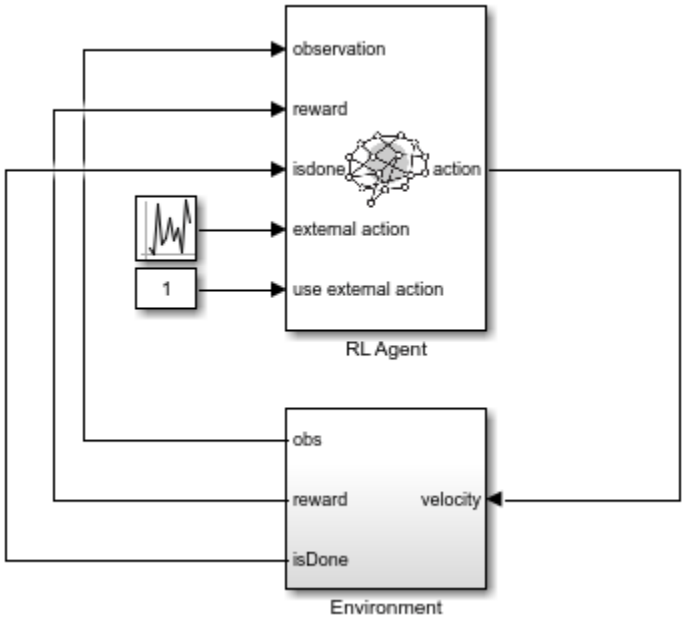
```
% set random seed
rng('default')
% parameters
Tv = 0.8;             % sample time for visualizer
Ts = 0.1;             % sample time for controller
tau = 0.01;           % damping constant for green ball
velLimit = 1;         % maximum speed for green ball
s0 = 200;             % random seed
s1 = 100;             % random seed
x0 = 0.2;             % initial position for ball
```

Create Environment and Agent for Collecting Data

In this example, a constraint function is represented using a trained deep neural network. To train the network, you must first collect training data from the environment.

To do so, first create an RL environment using the `rlBallOneDim` model. This model applies random external actions through an RL Agent block to the environment.

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



The Environment subsystem performs the following steps.

- Applies the input velocity to the environment model and generates the resulting output observations
- Computes the training reward $r = \left[1 - 10(x - x_r)^2\right]^+$, where x_r denotes the position of the red ball
- Sets the termination signal `isDone` to `true` if the ball position violates the constraint $0 \le x \le 1$

For this model, the observations from the environment include the position and velocity of the green ball and the noisy measurement of the red ball position. Define a continuous observation space for these three values.

```
obsInfo = rlNumericSpec([3 1]);
```

The action that the agent applies to the green ball is its velocity. Create a continuous action space and apply the required velocity limits.

```
actInfo = rlNumericSpec([1 1],'LowerLimit',-velLimit,'UpperLimit',velLimit);
```

Create an RL environment for this model.

```
agentblk = [mdl '/RL Agent'];
env = rlSimulinkEnv(mdl,agentblk,obsInfo,actInfo);
```

Specify a reset function, which randomly initializes the environment at the start of each training episode or simulation.

```
env.ResetFcn = @(in)localResetFcn(in);
```

Next, create a DDPG reinforcement learning agent, which supports continuous actions and observations, using the `createDDPGAgentBall` helper function. This function creates critic and

actor representations based on the action and observation specifications and uses the representations to create a DDPG agent.

```
agent = createDDPGAgentBall(Ts,obsInfo,actInfo);
```

In the `rlBallOneDim` model, the RL Agent block does not generate actions. Instead, it is configured to pass a random external action to the environment. The purpose for using a data-collection model with an inactive RL Agent block is to ensure that the environment model, action and observation signal configurations, and model reset function used during data collection match those used during subsequent agent training.

Learn Constraint Function

In this example, the ball position signal x_{k+1} must satisfy $0 \le x_{k+1} \le 1$. To allow for some slack, the constraint is set to be $0.1 \le x_{k+1} \le 0.9$. The dynamic model from velocity to position has a very small damping constant, thus it can be approximated by $x_{k+1} \approx x_k + h(x_k)u_k$. Therefore, the constraints for green ball are given by the following equation.

$$\begin{bmatrix} x_k \\ -x_k \end{bmatrix} + \begin{bmatrix} h(x_k) \\ -h(x_k) \end{bmatrix} u_k \le \begin{bmatrix} 0.9 \\ -0.1 \end{bmatrix}$$

The Constraint Enforcement block accepts constraints of the form $f_x + g_x u \le c$. For the above equation, the coefficients of this constraint function are as follows.

$$f_x = \begin{bmatrix} x_k \\ -x_k \end{bmatrix}, g_x = \begin{bmatrix} h(x_k) \\ -h(x_k) \end{bmatrix}, c = \begin{bmatrix} 0.9 \\ -0.1 \end{bmatrix}$$

The function $h(x_k)$ is approximated by a deep neural network that is trained on the data collected by simulating the RL agent within the environment. To learn the unknown function $h(x_k)$, the RL agent passes a random external action to the environment that is uniformly distributed in the range $[-1, 1]$.

To collect data, use the `collectDataBall` helper function. This function simulates the environment and agent and collects the resulting input and output data. The resulting training data has three columns: x_k, u_k, and x_{k+1}.

For this example, load precollected training data. To collect the data yourself, set `collectData` to `true`.

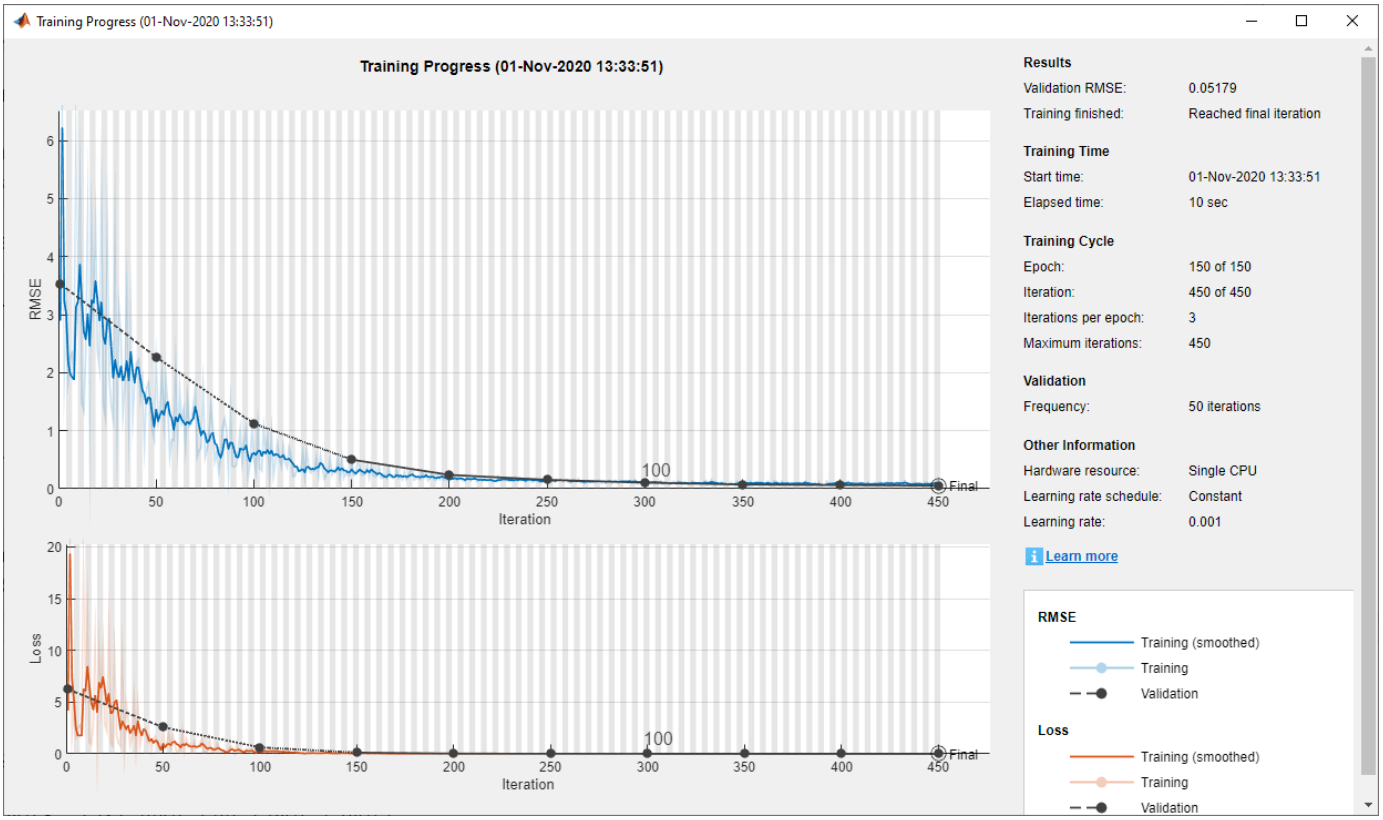
```
collectData = false;
if collectData
    count = 1050;
    data = collectDataBall(env,agent,count);
else
    load trainingDataBall data
end
```

Train a deep neural network to approximate the constraint function using the `trainConstraintBall` helper function. This function formats the data for training then creates and trains a deep neural network. Training a deep neural network requires Deep Learning Toolbox™ software.

For this example, to ensure reproducibility, load a pretrained network. To train the network yourself, set `trainConstraint` to `true`.

```
trainConstraint = false;
if trainConstraint
    network = trainConstraintBall(data);
else
    load trainedNetworkBall network
end
```

The following figure shows an example of the training progress.



Validate the trained neural network using the `validateNetworkBall` helper function. This function processes the input training data using the trained deep neural network. It then compares the network output with the training output and computes the root mean-squared error (RMSE).

```
validateNetworkBall(data,network)
```

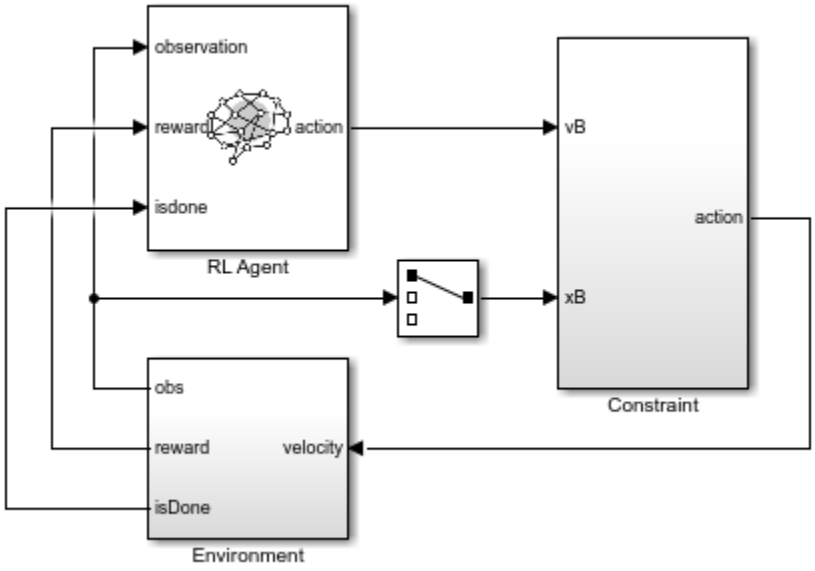
```
Test Data RMSE = 9.996700e-02
```

The small RMSE value indicates that the network successfully learned the constraint function.

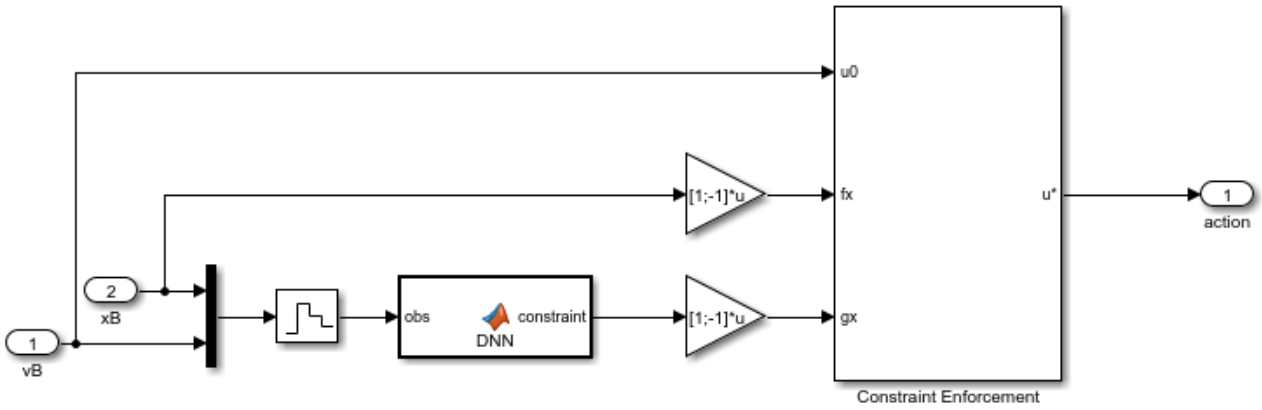
Train Agent with Constraint Enforcement

To train the agent with constraint enforcement, use the `rlBallOneDimWithConstraint` model. This model constrains the actions from the agent before applying them to the environment.

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



To view the constraint implementation, open the Constraint subsystem. Here, the trained deep neural network approximates $h(x_k)$, and the Constraint Enforcement block enforces the constraint function and velocity bounds.



For this example the following Constraint Enforcement block parameter settings are used.

- **Number of constraints** — 2
- **Number of actions** — 1
- **Constraint bound** — `[0.9;-0.1]`

Create an RL environment using this model. The observation and action specifications match those used for the previous data collection environment.

```
agentblk = [mdl '/RL Agent'];
env = rlSimulinkEnv(mdl,agentblk,obsInfo,actInfo);
env.ResetFcn = @(in)localResetFcn(in);
```

Specify options for training the agent. Train the RL agent for 300 episodes with 300 steps per episode.

```
trainOpts = rlTrainingOptions(...
    'MaxEpisodes',300, ...
    'MaxStepsPerEpisode', 300, ...
    'Verbose', false, ...
    'Plots','training-progress');
```

Train the agent. Training is a time-consuming process. For this example, load a pretrained agent using the `loadAgentParams` helper function. To train the agent yourself, set `trainAgent` to `true`.

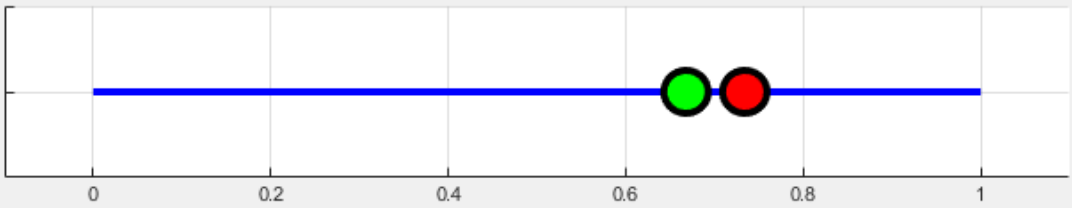
```
trainAgent = false;
if trainAgent
    trainingStats = train(agent,env,trainOpts);
else
    loadAgentParams(agent,'rlAgentBallParams')
end
```

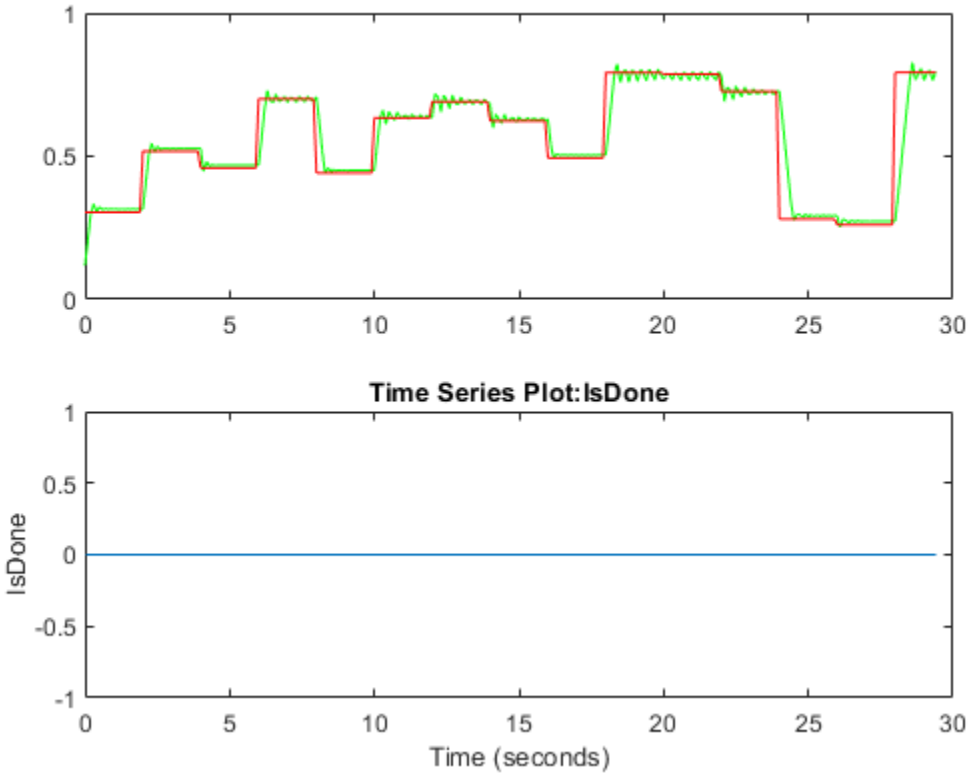
The following figure shows the training results. The training process converges to a good agent within 20 episodes.

Since **Total Number of Steps** equals the product of **Episode Number** and **Episode Steps**, each training episode runs to the end without early termination. Therefore, the Constraint Enforcement block ensures that the ball position x never violates the constraint $0 \le x \le 1$.

Simulate the trained agent using the `simWithTrainedAgentBall` helper function.

```
simWithTrainedAgentBall(env,agent)
```





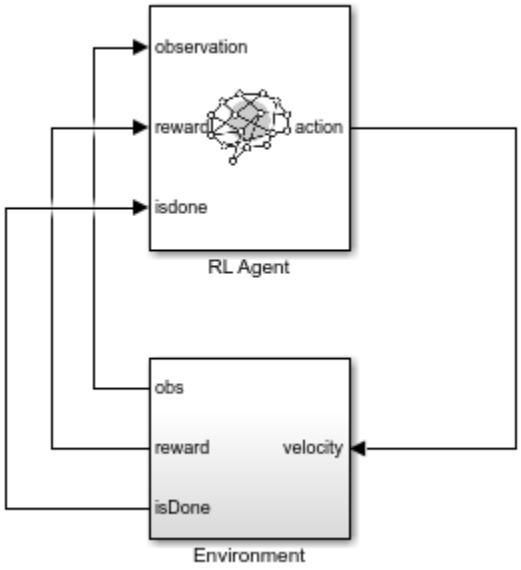
The agent successfully tracks the position of the red ball.

Train Agent Without Constraint Enforcement

To see the benefit of training an agent with constraint enforcement, you can train the agent without constraints and compare the training results to the constraint enforcement case.

To train the agent without constraints, use the `rlBallOneDimWithoutConstraint` model. This model applies the actions from the agent directly to the environment.

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



Create an RL environment using this model.

```
agentblk = [mdl '/RL Agent'];
env = rlSimulinkEnv(mdl,agentblk,obsInfo,actInfo);
env.ResetFcn = @(in)localResetFcn(in);
```

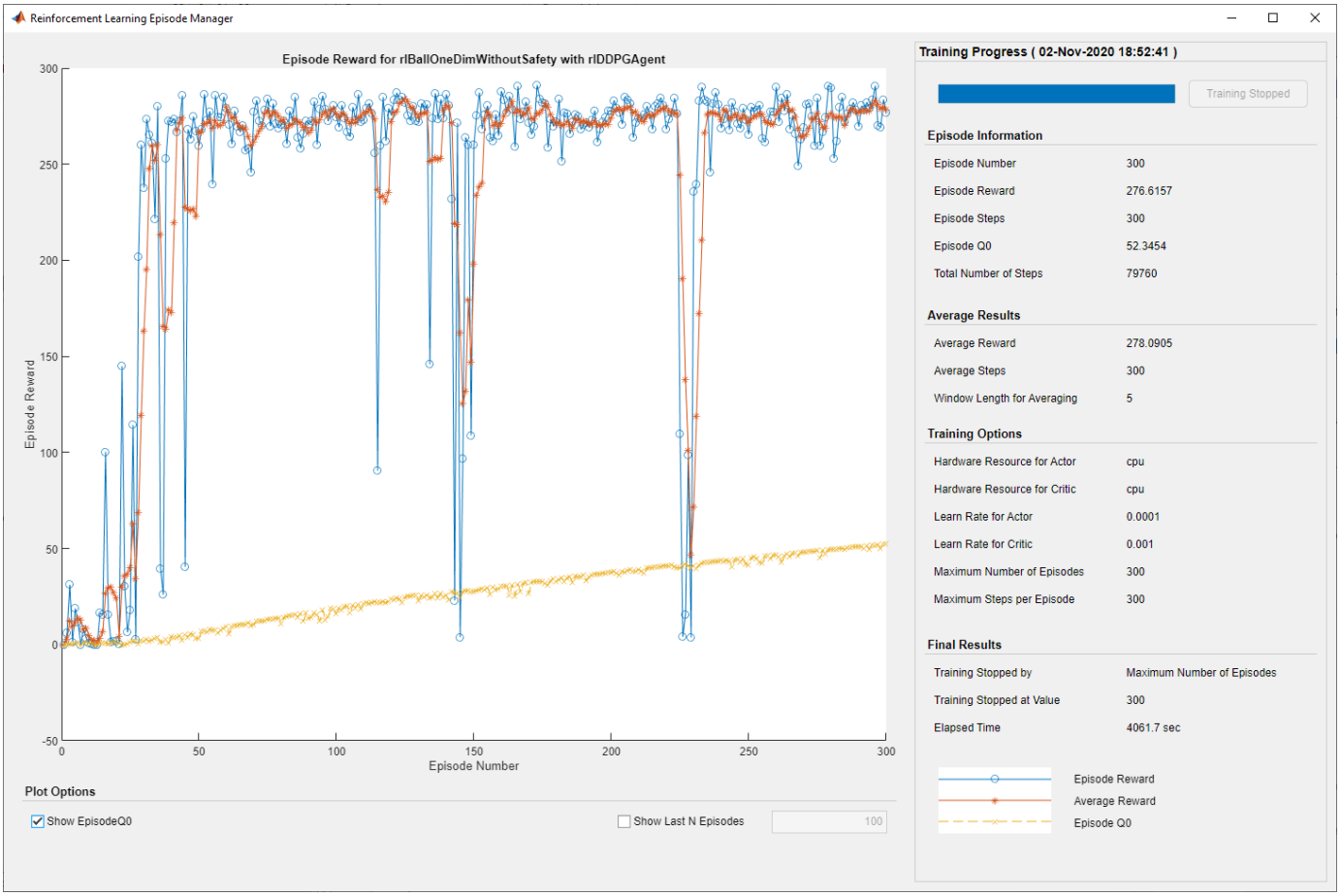
Create a new DDPG agent to train. This agent has the same configuration as the agent used in the previous training.

```
agent = createDDPGAgentBall(Ts,obsInfo,actInfo);
```

Train the agent using the same training options as in the constraint enforcement case. For this example, as with the previous training, load a pretrained agent. To train the agent yourself, set `trainAgent` to true.

```
trainAgent = false;
if trainAgent
    trainingStats2 = train(agent,env,trainOpts);
else
    loadAgentParams(agent,'rlAgentBallCompParams')
end
```

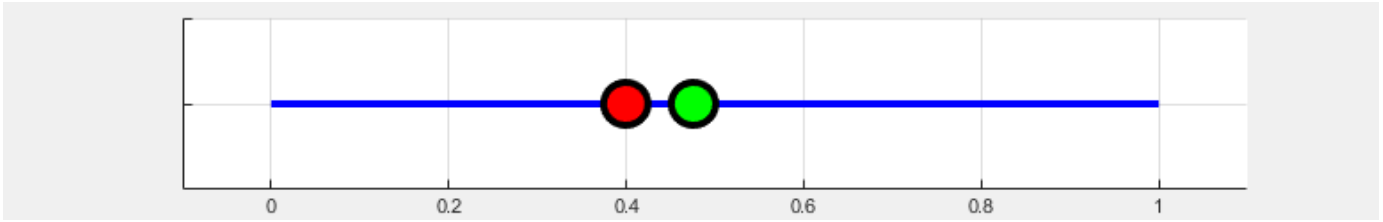
The following figure shows the training results. The training process converges to a good agent after 50 episodes. After that point, the agent has poor performance for some episodes, for example around episodes 140 and 230.

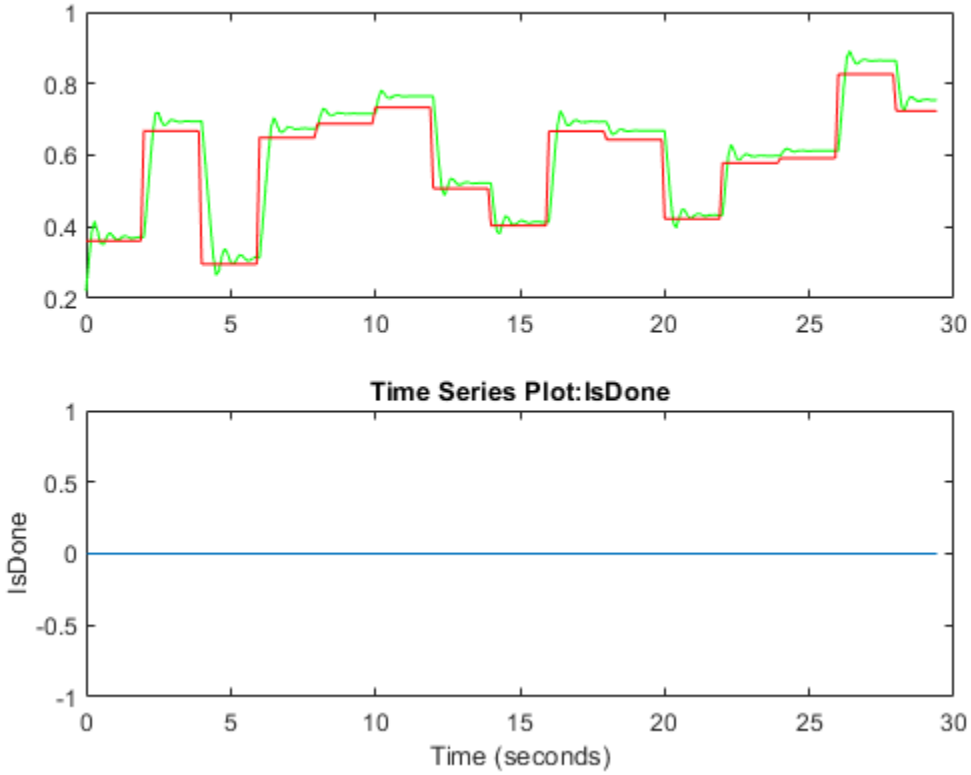


Since **Total Number of Steps** is less than the product of **Episode Number** and **Episode Steps**, the training includes episodes that terminated early due to constraint violations.

Simulate the trained agent.

```
simWithTrainedAgentBall(env,agent)
```





The agent successfully tracks the position of the red ball with more steady-state offset than the agent trained with constraints.

Conclusion

In this example, training an RL agent with the Constraint Enforcement block ensures that the actions applied to the environment never produce a constraint violation. As a result, the training process converges to a good agent quickly. Training the same agent without constraints produces slower convergence and poorer performance.

```
bdclose('rlBallOneDim')
bdclose('rlBallOneDimWithConstraint')
bdclose('rlBallOneDimWithoutConstraint')
close('Ball One Dim')
```

Local Reset Function

```
function in = localResetFcn(in)
% Reset function
in = setVariable(in,'x0',rand);
in = setVariable(in,'s0',randi(5000));
in = setVariable(in,'s1',randi(5000));
end
```

References

[1] Dalal, Gal, Krishnamurthy Dvijotham, Matej Vecerik, Todd Hester, Cosmin Paduraru, and Yuval Tassa. "Safe Exploration in Continuous Action Spaces." Preprint, submitted January 26, 2018. https://arxiv.org/abs/1801.08757

See Also

Blocks

Constraint Enforcement | RL Agent

Related Examples

- “Constraint Enforcement for Control Design” on page 16-2
- “Learn and Apply Constraints for PID Controllers” on page 16-10
- “Train RL Agent for Adaptive Cruise Control with Constraint Enforcement” on page 16-29
- “Train RL Agent for Lane Keeping Assist with Constraint Enforcement” on page 16-39

Train RL Agent for Adaptive Cruise Control with Constraint Enforcement

This example shows how to train a reinforcement learning (RL) agent for adaptive cruise control (ACC) using guided exploration with the Constraint Enforcement block.

Overview

In this example, the goal is to make an ego car travel at a set velocity while maintaining a safe distance from a lead car by controlling longitudinal acceleration and braking. This example uses the same vehicle models and parameters as the "Train DDPG Agent for Adaptive Cruise Control" (Reinforcement Learning Toolbox) example.

Set the random seed and configure model parameters.

```
% Set random seed.
rng('default')

% Parameters
x0_lead = 50;   % Initial position for lead car (m)
v0_lead = 25;   % Initial velocity for lead car (m/s)
x0_ego = 10;    % Initial position for ego car (m)
v0_ego = 20;    % Initial velocity for ego car (m/s)
D_default = 10; % Default spacing (m)
t_gap = 1.4;    % Time gap (s)
v_set = 30;     % Driver-set velocity (m/s)
amin_ego = -3;  % Minimum acceleration for driver comfort (m/s^2)
amax_ego = 2;   % Maximum acceleration for driver comfort (m/s^2)
Ts = 0.1;       % Sample time (s)
Tf = 60;        % Duration (s)
```

Learn Constraint Equation

For the ACC application, the safety signals are the ego car velocity v and relative distance d between the ego car and lead car. In this example, the constraints for these signals are $10 \le v \le 30.5$ and $d \ge 5$. The constraints depend on the following states in x: ego car actual acceleration, ego car velocity, relative distance, and lead car velocity.

The action u is the ego car acceleration command. The following equation describes the safety signals in terms of the action and states.

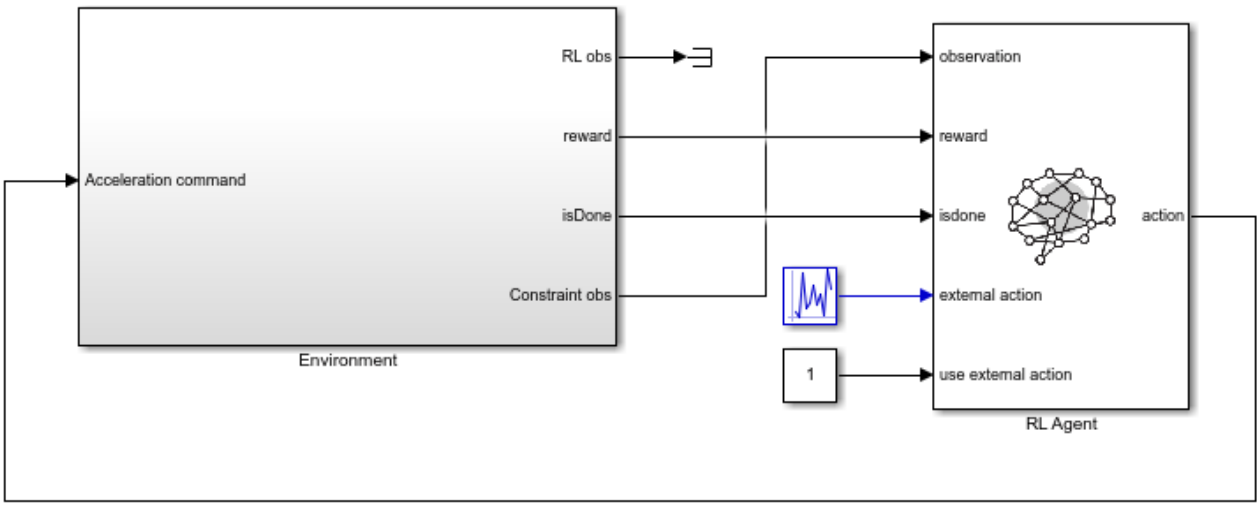
$$\begin{bmatrix} v_{k+1} \\ d_{k+1} \end{bmatrix} = \begin{bmatrix} f_1(x_k) \\ f_2(x_k) \end{bmatrix} + \begin{bmatrix} g_1(x_k) \\ g_2(x_k) \end{bmatrix} u_k$$

The Constraint Enforcement block accepts constraints of the form $f_x + g_x u \le c$. For this example, the coefficients of this constraint function are as follows.

$$f_x = \begin{bmatrix} -f_1(x_k) \\ -f_2(x_k) \\ f_1(x_k) \end{bmatrix},\ g_x = \begin{bmatrix} -g_1(x_k) \\ -g_2(x_k) \\ g_1(x_k) \end{bmatrix},\ c = \begin{bmatrix} -10 \\ -5 \\ 30.5 \end{bmatrix}$$

To learn the unknown functions f_i and g_i, you must first collect training data from the environment. To do so, first create an RL environment using the `rlLearnConstraintACC` model.

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



In this model, the RL Agent block does not generate actions. Instead, it is configured to pass a random external action to the environment. The purpose for using a data-collection model with an inactive RL Agent block is to ensure that the environment model, action and observation signal configurations, and model reset function used during data collection match those used during subsequent agent training.

The random external action signal is uniformly distributed in the range [10, 6]; that is, the ego car has a maximum braking power of -10 m/s^2 and a maximum acceleration power of 6 m/s^2.

For training, the four observations from the environment are the relative distance between the vehicles, the velocities of the lead and ego cars, and the ego car acceleration. Define a continuous observation space for these values.

```
obsInfo = rlNumericSpec([4 1]);
```

The action output by the agent is the acceleration command. Create a corresponding continuous action space with acceleration limits.

```
actInfo = rlNumericSpec([1 1],'LowerLimit',-3,'UpperLimit',2);
```

Create an RL environment for this model. Specify a reset function to set a random position for the lead car at the start of each training episode or simulation.

```
agentblk = [mdl '/RL Agent'];
env = rlSimulinkEnv(mdl,agentblk,obsInfo,actInfo);
env.ResetFcn = @(in)localResetFcn(in);
```

Next, create a DDPG reinforcement learning agent, which supports continuous actions and observations, using the `createDDPGAgentBACC` helper function. This function creates critic and actor representations based on the action and observation specifications and uses the representations to create a DDPG agent.

```
agent = createDDPGAgentACC(Ts,obsInfo,actInfo);
```

To collect data, use the `collectDataACC` helper function. This function simulates the environment and agent and collects the resulting input and output data. The resulting training data has nine columns.

- Relative distance between the cars
- Lead car velocity
- Ego car velocity
- Ego car actual acceleration
- Ego acceleration command
- Relative distance between the cars in the next time step
- Lead car velocity in the next time step
- Ego car velocity in the next time step
- Ego car actual acceleration in the next time step

For this example, load precollected training data. To collect the data yourself, set `collectData` to `true`.

```
collectData = false;
if collectData
    count = 1000;
    data = collectDataACC(env,agent,count);
else
    load trainingDataACC data
end
```

For this example, the dynamics of the ego car and lead car are linear. Therefore, you can find a least-squares solution for the safety-signal constraints; that is, $v = R_v I$ and $d = R_d I$, where I is $[x_k; u_k]$.

```
% Extract state and input data.
I = data(1:1000,[4,3,1,2,5]);
% Extract data for the relative distance in the next time step.
d = data(1:1000,6);
% Compute the relation from the state and input to relative distance.
Rd = I\d;
% Extract data for actual ego car velocity.
v = data(1:1000,8);
% Compute the relation from the state and input to ego car velocity.
Rv = I\v;
```

Validate the learned constraints using the `validateConstraintACC` helper function. This function processes the input training data using the learned constraints. It then compares the network output with the training output and computes the root mean squared error (RMSE).

```
validateConstraintACC(data,Rd,Rv)
```

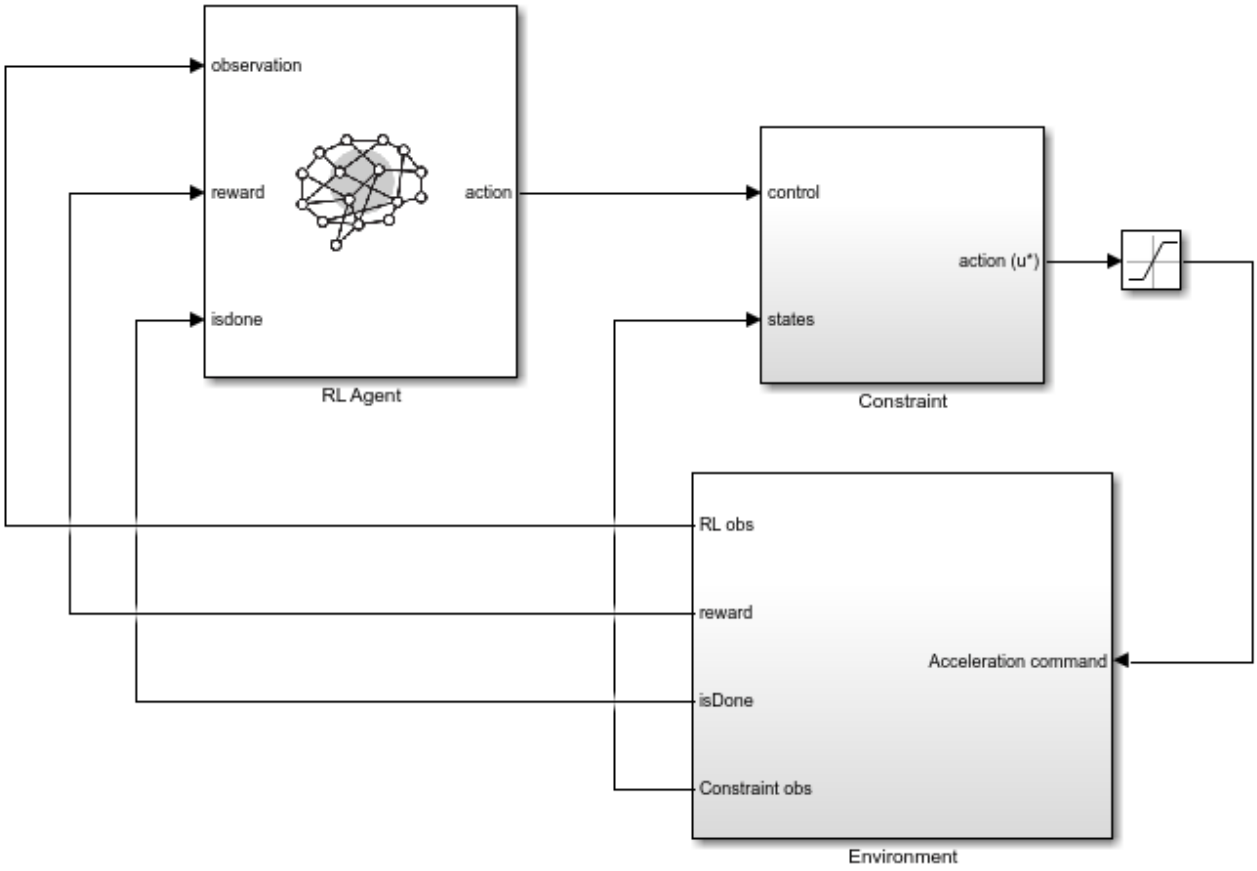
```
Test Data RMSE for Relative Distance  = 8.118162e-04
Test Data RMSE for Ego Velocity = 5.967300e-15
```

The small RMSE values indicate successful constraint learning.

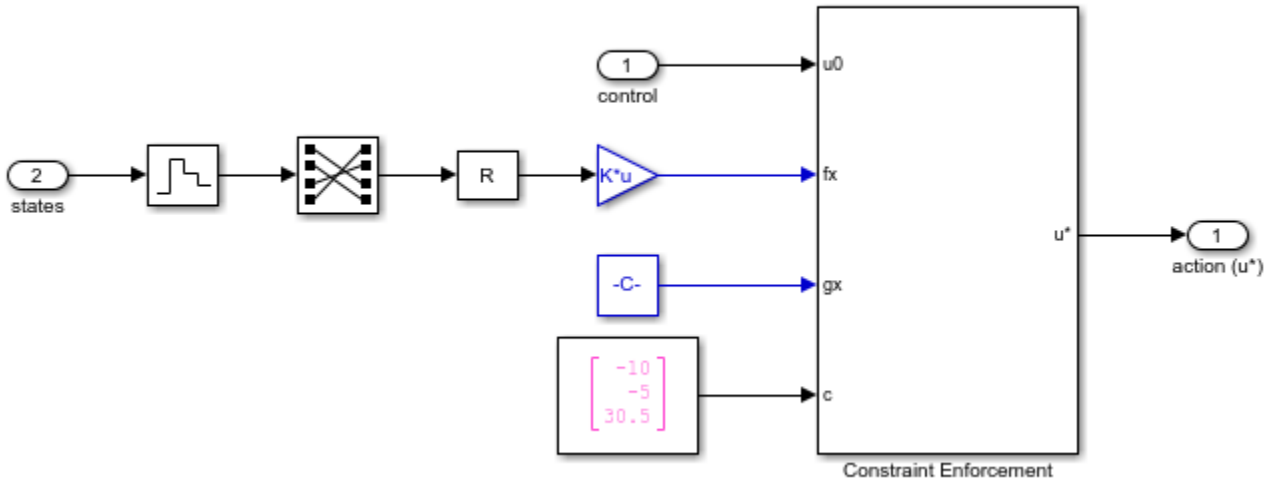
Train Agent with Constraint Enforcement

To train the agent with constraint enforcement, use the `rlACCwithConstraint` model. This model constrains the acceleration command from the agent before applying it to the environment.

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



To view the constraint implementation, open the Constraint subsystem. Here, the model generates the values of f_i and g_i from the linear constraint relations. The model sends these values along with the constraint bounds to the Constraint Enforcement block.



Create an RL environment using this model. The action specification is the same as for the constraint-learning environment. For training, the environment produces three observations: the integral of the velocity error, the velocity error, and the ego-car velocity.

The Environment subsystem generates an `isDone` signal when critical constraints are violated—either the ego car has negative velocity (moves backwards) or the relative distance is less than zero (ego car collides with lead car). The RL Agent block uses this signal to terminate training episodes early.

```
obsInfo = rlNumericSpec([3 1]);
agentblk = [mdl '/RL Agent'];
env = rlSimulinkEnv(mdl,agentblk,obsInfo,actInfo);
env.ResetFcn = @(in)localResetFcn(in);
```

Since the observation specification are different for training, you must also create a new DDPG agent.

```
agent = createDDPGAgentACC(Ts,obsInfo,actInfo);
```

Specify options for training the agent. Train the agent for at most 5000 episodes. Stop training if the episode reward exceeds 260.

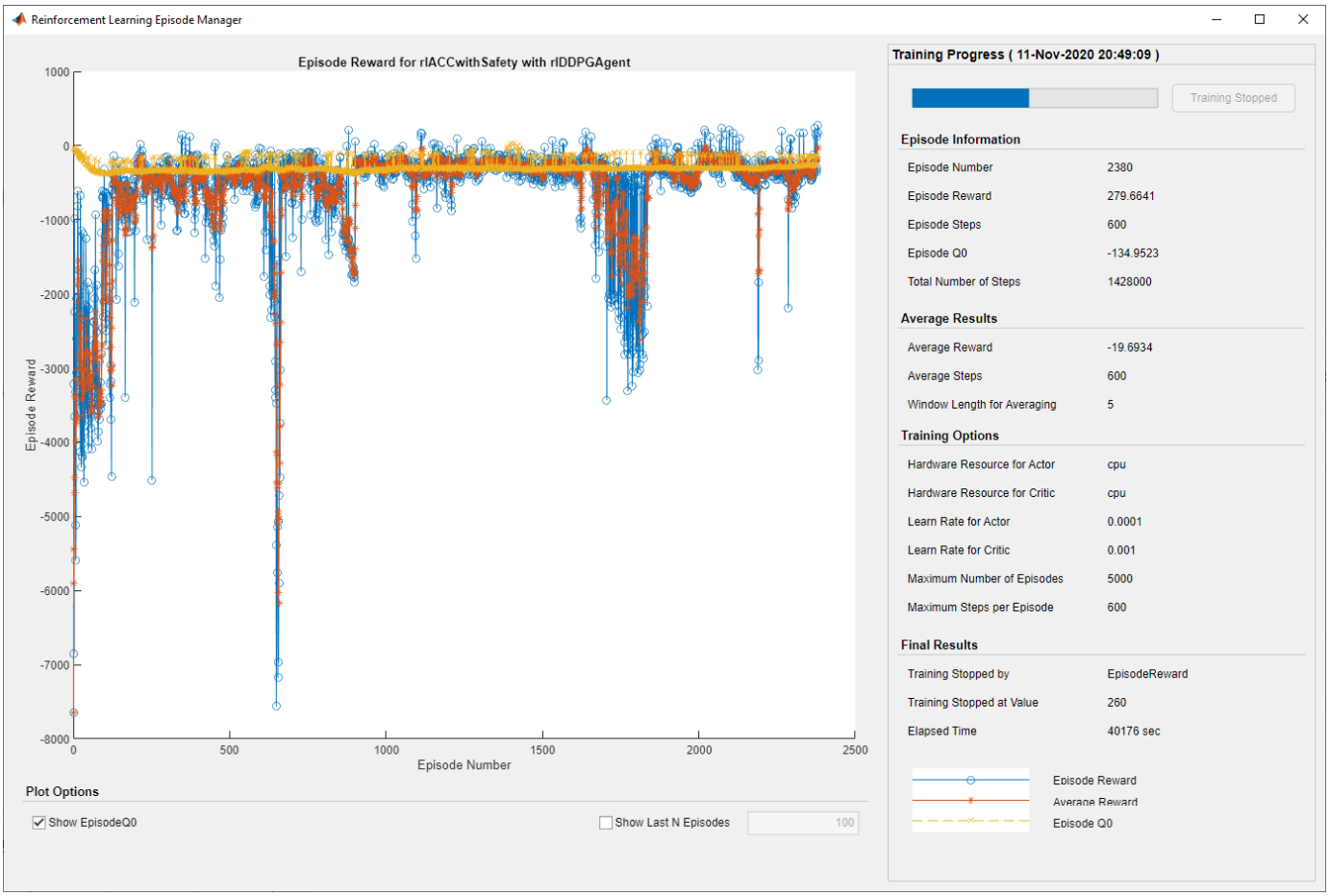
```
maxepisodes = 5000;
maxsteps = ceil(Tf/Ts);
trainingOpts = rlTrainingOptions(...
    'MaxEpisodes',maxepisodes,...
    'MaxStepsPerEpisode',maxsteps,...
    'Verbose',false,...
    'Plots','training-progress',...
    'StopTrainingCriteria','EpisodeReward',...
    'StopTrainingValue',260);
```

Train the agent. Training is a time-consuming process, so for this example, load a pretrained agent. To train the agent yourself instead, set `trainAgent` to `true`.

```
trainAgent = false;
if trainAgent
```

```
    trainingStats = train(agent,env,trainingOpts);
else
    load rlAgentConstraintACC agent
end
```

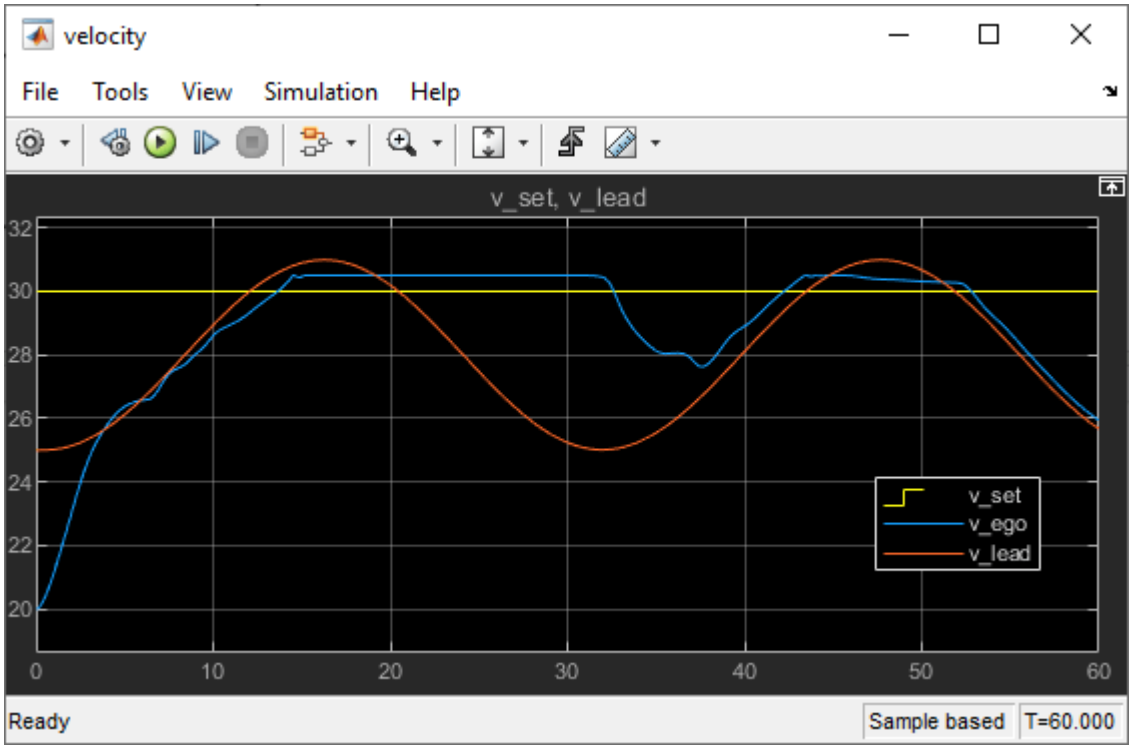
The following figure shows the training results.



Since **Total Number of Steps** equals the product of **Episode Number** and **Episode Steps**, each training episode runs to the end without early termination. Therefore, the Constraint Enforcement block ensures that the ego car never violates the critical constraints.

Run the trained agent and view the simulation results.

```
x0_lead = 80;
sim(mdl);
```

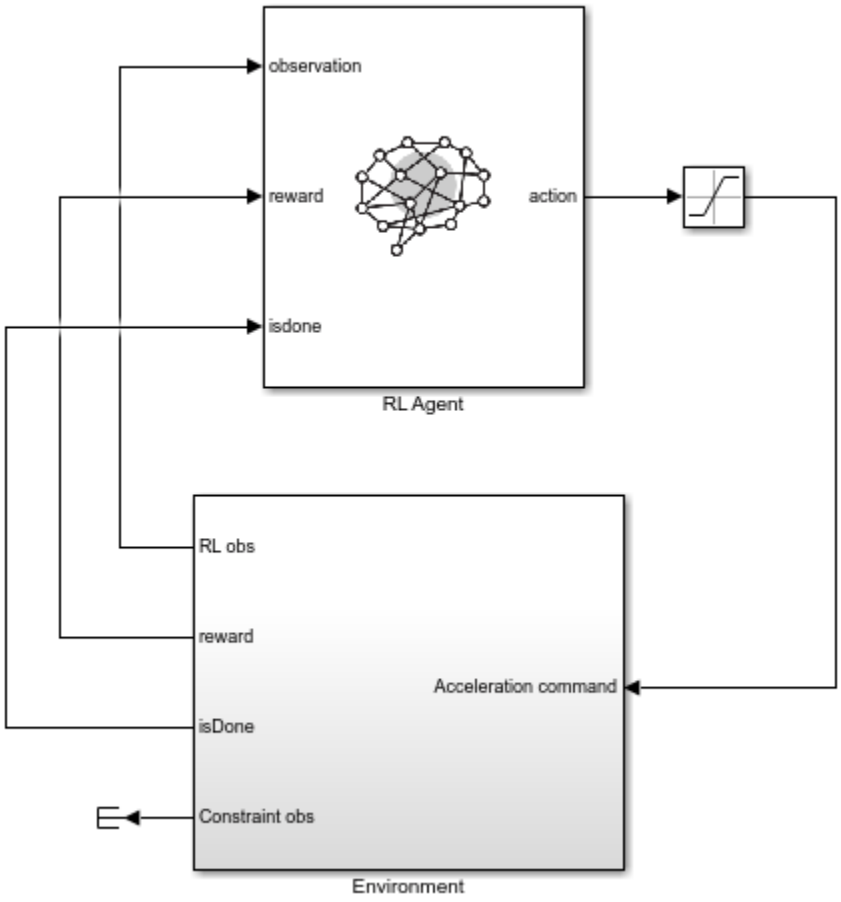


Train Agent Without Constraints

To see the benefit of training an agent with constraint enforcement, you can train the agent without constraints and compare the training results to the constraint enforcement case.

To train the agent without constraints, use the `rlACCwithoutConstraint` model. This model applies the actions from the agent directly to the environment, and the agent uses the same action and observation specifications.

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



Create an RL environment using this model.

```
agentblk = [mdl '/RL Agent'];
env = rlSimulinkEnv(mdl,agentblk,obsInfo,actInfo);
env.ResetFcn = @(in)localResetFcn(in);
```

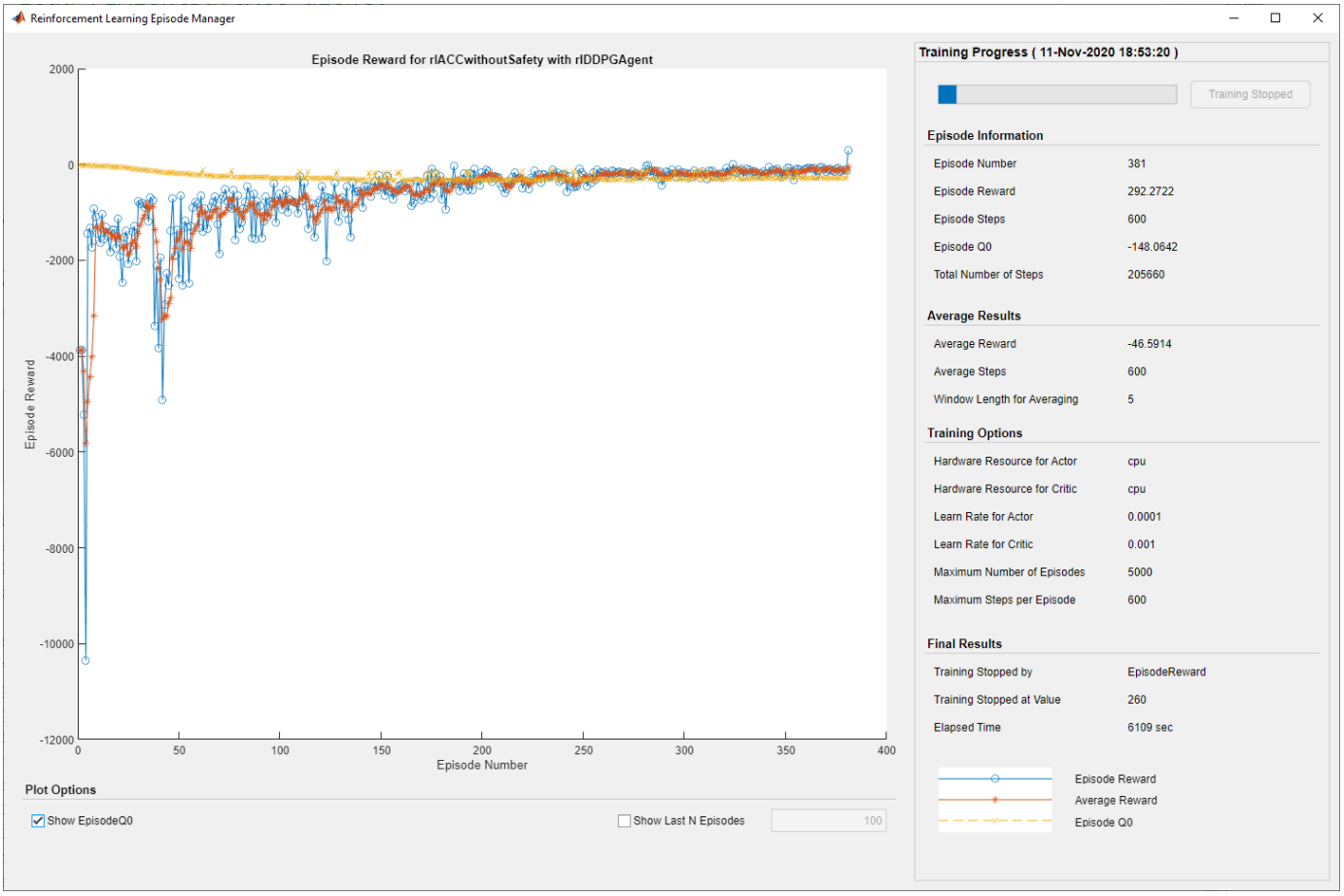
Create a new DDPG agent to train. This agent has the same configuration as the agent used in the previous training.

```
agent = createDDPGAgentACC(Ts,obsInfo,actInfo);
```

Train the agent using the same training options as the in the constraint enforcement case. For this example, as with the previous training, load a pretrained agent. To train the agent yourself, set `trainAgent` to true.

```
trainAgent = false;
if trainAgent
    trainingStats2 = train(agent,env,trainingOpts);
else
    load rlAgentACC agent
end
```

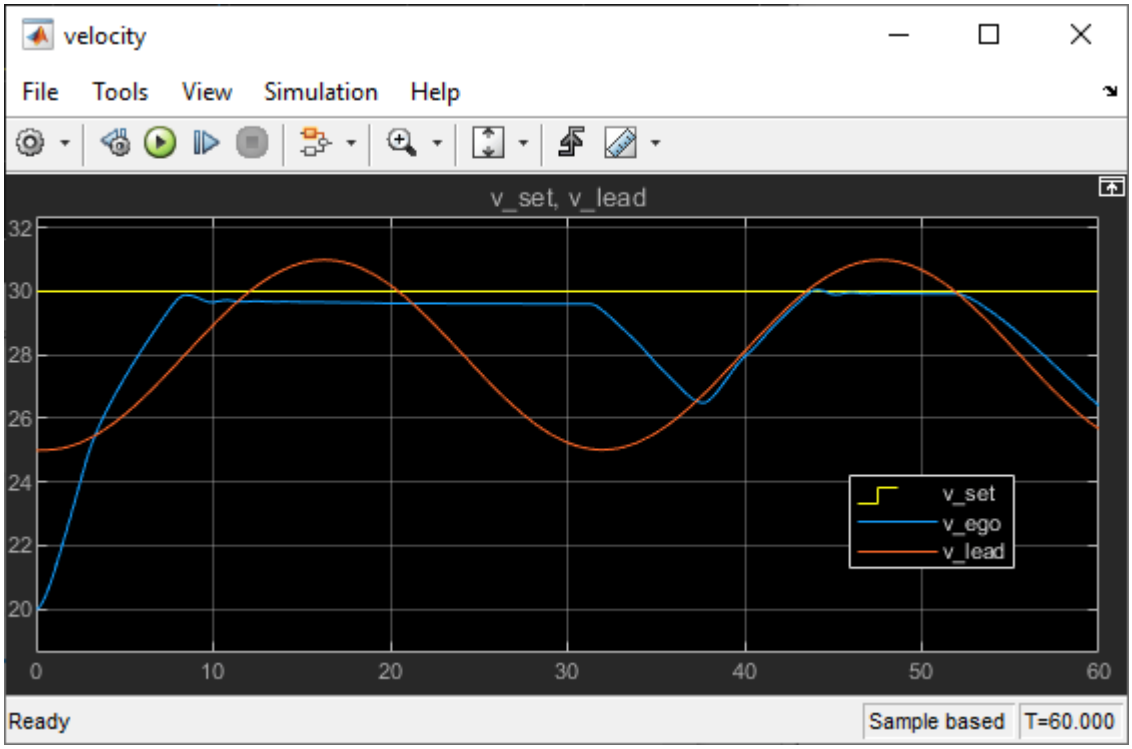
The following figure shows the training results.



Since **Total Number of Steps** is less than the product of **Episode Number** and **Episode Steps**, the training includes episodes that terminated early due to constraint violations.

Run the trained agent and plot the simulation results.

```
x0_lead = 80;
sim(mdl)
```




```
bdclose('rlLearnConstraintACC')
bdclose('rlACCwithConstraint')
bdclose('rlACCwithoutConstraint')
```

Local Reset Function

```
function in = localResetFcn(in)
% Reset the initial position of the lead car.
in = setVariable(in,'x0_lead',40+randi(60,1,1));
end
```

See Also

Blocks
Constraint Enforcement | RL Agent

Related Examples

- “Constraint Enforcement for Control Design” on page 16-2
- “Learn and Apply Constraints for PID Controllers” on page 16-10
- “Train Reinforcement Learning Agent with Constraint Enforcement” on page 16-18
- “Train RL Agent for Lane Keeping Assist with Constraint Enforcement” on page 16-39

Train RL Agent for Lane Keeping Assist with Constraint Enforcement

This example shows how to train an reinforcement learning (RL) agent for lane keeping assist (LKA) with constraints enforced using the Constraint Enforcement block.

Overview

In this example, the goal is to keep an ego car traveling along the centerline of a lane by adjusting the front steering angle. This example uses the same vehicle model and parameters as the “Train DQN Agent for Lane Keeping Assist Using Parallel Computing” (Reinforcement Learning Toolbox) example.

Set the random seed and configure model parameters.

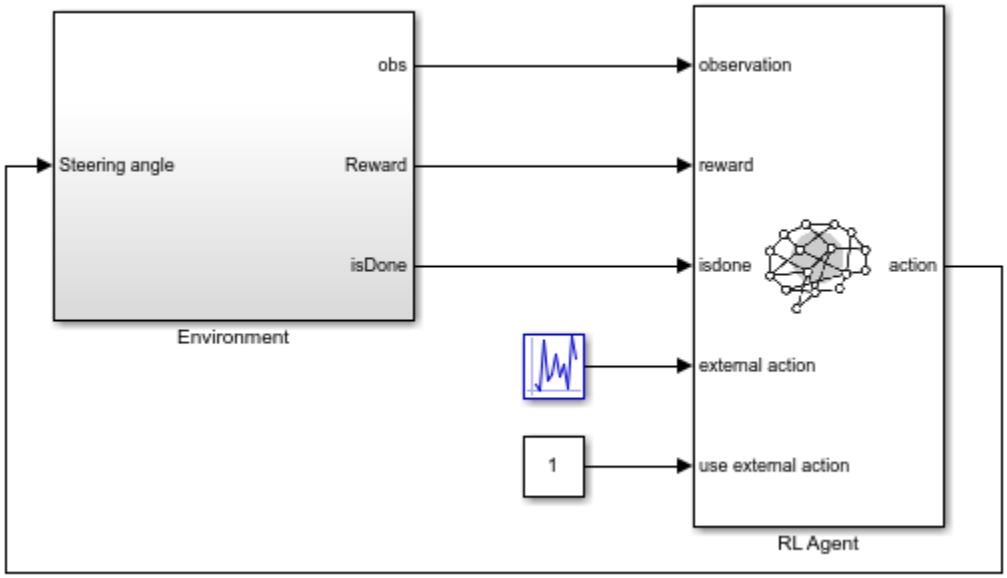
```
% Set random seed.
rng(0);

% Paramters
m = 1575;           % Total vehicle mass (kg)
Iz = 2875;          % Yaw moment of inertia (mNs^2)
lf = 1.2;           % Longitudinal distance from center of gravity to front tires (m)
lr = 1.6;           % Longitudinal distance from center of gravity to rear tires (m)
Cf = 19000;         % Cornering stiffness of front tires (N/rad)
Cr = 33000;         % Cornering stiffness of rear tires (N/rad)
Vx = 15;            % Longitudinal velocity (m/s)
Ts = 0.1;           % Sample time (s)
T = 15;             % Duration (s)
rho = 0.001;        % Road curvature (1/m)
e1_initial = 0.2;   % Initial lateral deviation from center line (m)
e2_initial = -0.1;  % Initial yaw angle error (rad)
steerLimit = 0.2618;% Maximum steering angle for driver comfort (rad)
```

Create Environment and Agent for Collecting Data

In this example, the constraint function enforced by the Constraint Enforcement block is unknown. To learn the function, you must first collect training data from the environment.

To do so, first create an RL environment using the `rlLearnConstraintLKA` model. This model applies random external actions through an RL Agent block to the environment.

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



The observations from the environment are the lateral deviation e_1, the relative yaw angle e_2, their derivatives, and their integrals. Create a continuous observation space for these six signals.

```
obsInfo = rlNumericSpec([6 1]);
```

The action from the RL Agent block is the front steering angle, which can take one of 31 possible values from -15 to 15 degrees. Create a discrete action space for this signal.

```
actInfo = rlFiniteSetSpec((-15:15)*pi/180);
```

Create an RL environment for this model.

```
agentblk = [mdl '/RL Agent'];
env = rlSimulinkEnv(mdl,agentblk,obsInfo,actInfo);
```

Specify a reset function, which randomly initializes the lateral deviation and relative yaw angle at the start of each training episode or simulation.

```
env.ResetFcn = @(in)localResetFcn(in);
```

Next, create a DQN reinforcement learning agent, which supports discrete actions and continuous observations, using the `createDQNAgentLKA` helper function. This function creates a critic representation based on the action and observation specifications and uses the representation to create a DQN agent.

```
agent = createDQNAgentLKA(Ts,obsInfo,actInfo);
```

In the `rlLearnConstraintLKA` model, the RL Agent block does not generate actions. Instead, it is configured to pass a random external action to the environment. The purpose for using a data-collection model with an inactive RL Agent block is to ensure that the environment model, action and observation signal configurations, and model reset function used during data collection match those used during subsequent agent training.

Learn Constraint Function

In this example, the safety signal is e_1. The constraint for this signal is $-1 \le e_1 \le 1$; that is, the distance from the centerline of the lane must be less than 1. The constraint depends on the states in x: the lateral deviation and its derivative, and the yaw angle error and its derivative. The action u is the front steering angle. The relatinship between the states and the lateral deviation is described by the following equation.

$$e_1(k+1) = f(x_k) + g(x_k)u_k$$

To allow for some slack, set the maximum lateral distance to be 0.9.

The Constraint Enforcement block accepts constraints of the form $f_x + g_x u \le c$. For the previous equation and constraints, the coefficients of the constraint function are:

$$f_x = \begin{bmatrix} f(x_k) \\ -f(x_k) \end{bmatrix},\ g_x = \begin{bmatrix} g(x_k) \\ -g(x_k) \end{bmatrix},\ c = \begin{bmatrix} 0.9 \\ 0.9 \end{bmatrix}$$

To learn the unknown functions f_x and g_x, the RL agent passes a random external action to the environment that is uniformly distributed in the range [–0.2618, 0.2618].

To collect data, use the `collectDataLKA` helper function. This function simulates the environment and agent and collects the resulting input and output data. The resulting training data has eight columns, the first six of which are the observations for the RL agent.

- Integral of lateral deviation
- Lateral deviation
- Integral of yaw angle error
- Yaw angle error
- Derivative of lateral deviation
- Derivative of yaw angle error
- Steering angle
- Lateral deviation in the next time step

For this example, load precollected training data. To collect the data yourself, set `collectData` to `true`.

```
collectData = false;
if collectData
    count = 1050;
    data = collectDataLKA(env,agent,count);
else
    load trainingDataLKA data
end
```

For this example, the dynamics of the ego car are linear. Therefore, you can find a least-squares solution for the lateral-deviation constraints.

You can apply linear approximations to learn the unknown functions f_x and g_x.

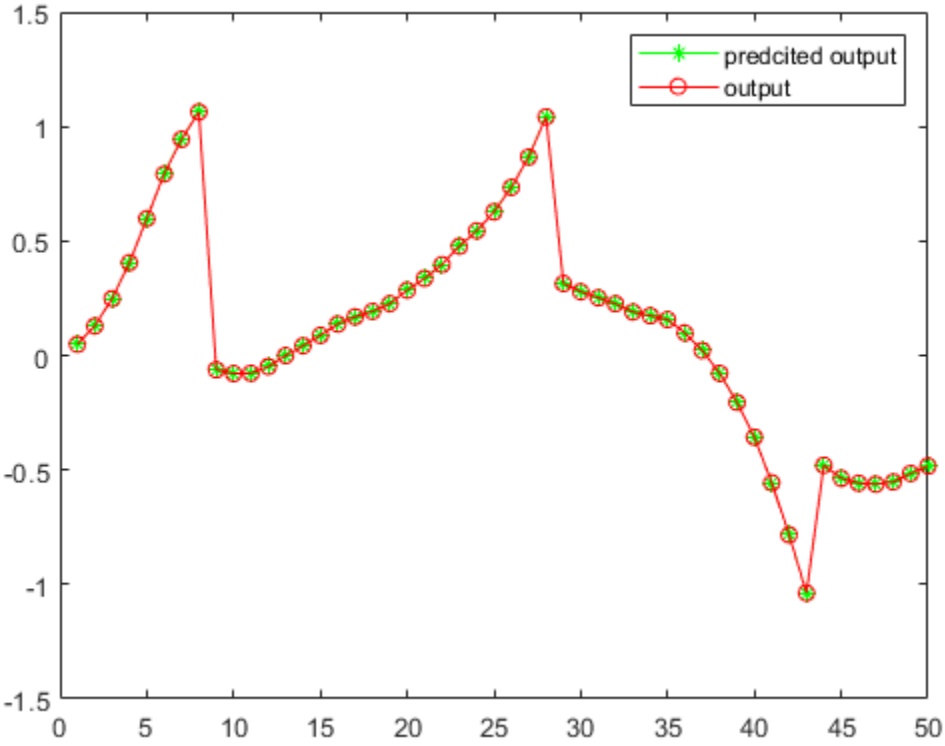
```
% Extract state and input data.
inputData = data(1:1000,[2,5,4,6,7]);
```

```
% Extract data for the lateral deviation in the next time step.
outputData = data(1:1000,8);
% Compute the relation from the state and input to the lateral deviation.
relation = inputData\outputData;
% Extract the components of the constraint function coefficients.
Rf = relation(1:4)';
Rg = relation(5);
```

Validate the learned constraints using the `validateConstraintLKA` helper function. This function processes the input training data using the learned constraints. It then compares the network output with the training output and computes the root mean squared error (RMSE).

```
validateConstraintLKA(data,Rf,Rg);
```

```
Test Data RMSE = 8.569169e-04
```

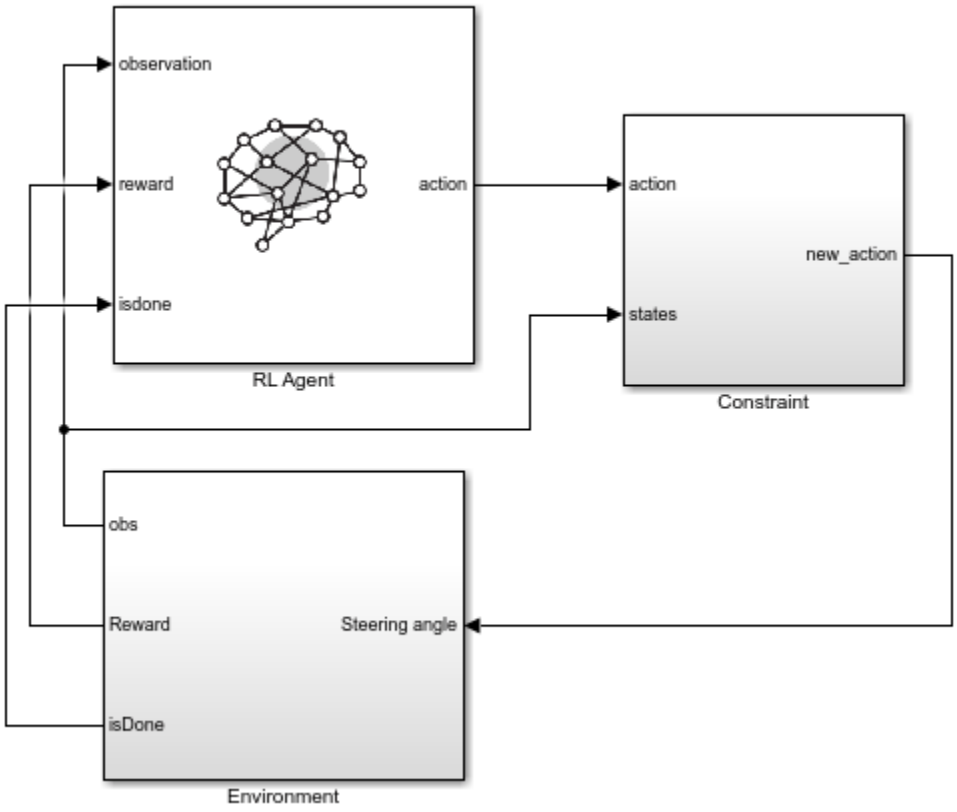


The small RMSE value indicates successful constraint learning.

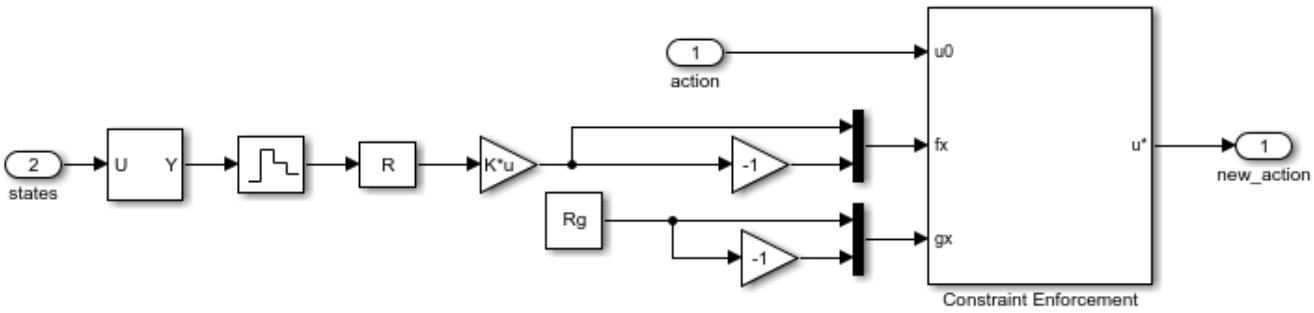
Train RL Agent with Constraint Enforcement

To train the agent with constraint enforcement, use the `rlLKAWithConstraint` model. This model constrains the actions from the agent before applying them to the environment.

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



To view the constraint implementation, open the Constraint subsystem. Here, the model generates the values of f_i and g_i from the linear constraint relations. The model sends these values along with the constraint bounds to the Constraint Enforcement block.



Create an RL environment using this model. The observation and action specifications are the same as for the constraint-learning environment.

The Environment subsystem creates an `isDone` signal that is `true` when the lateral deviation exceeds a specified constraint. The RL Agent block uses this signal to terminate training episodes early.

```
agentblk = [mdl '/RL Agent'];
env = rlSimulinkEnv(mdl,agentblk,obsInfo,actInfo);
env.ResetFcn = @(in)localResetFcn(in);
```

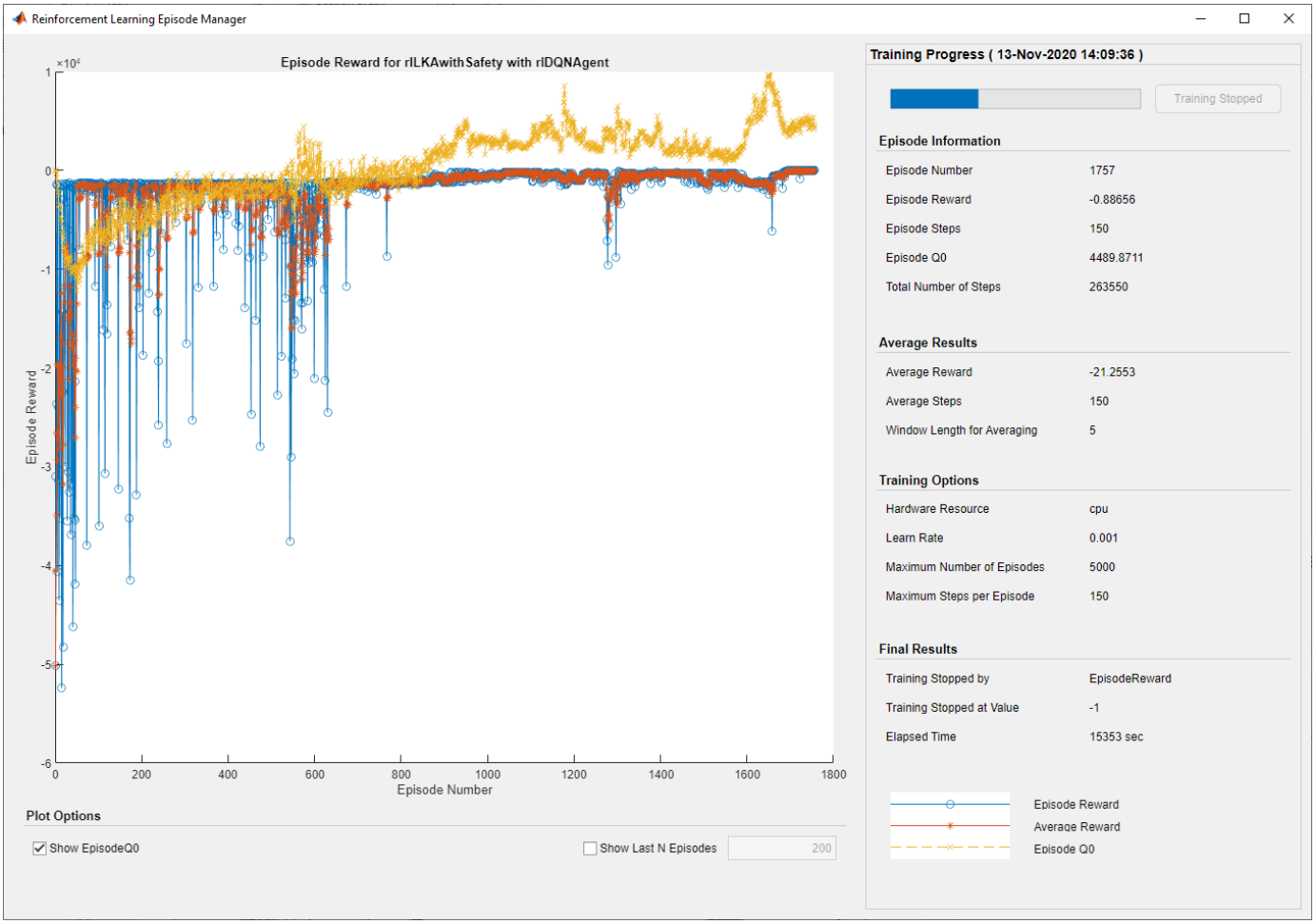
Specify options for training the agent. Train the agent for at most 5000 episodes. Stop training if the episode reward exceeds –1.

```
maxepisodes = 5000;
maxsteps = ceil(T/Ts);
trainingOpts = rlTrainingOptions(...
    'MaxEpisodes',maxepisodes,...
    'MaxStepsPerEpisode',maxsteps,...
    'Verbose',false,...
    'Plots','training-progress',...
    'StopTrainingCriteria','EpisodeReward',...
    'StopTrainingValue',-1);
```

Train the agent. Training is a time-consuming process, so for this example, load a pretrained agent. To train the agent yourself instead, set `trainAgent` to `true`.

```
trainAgent = false;
if trainAgent
    trainingStats = train(agent,env,trainingOpts);
else
    load rlAgentConstraintLKA agent
end
```

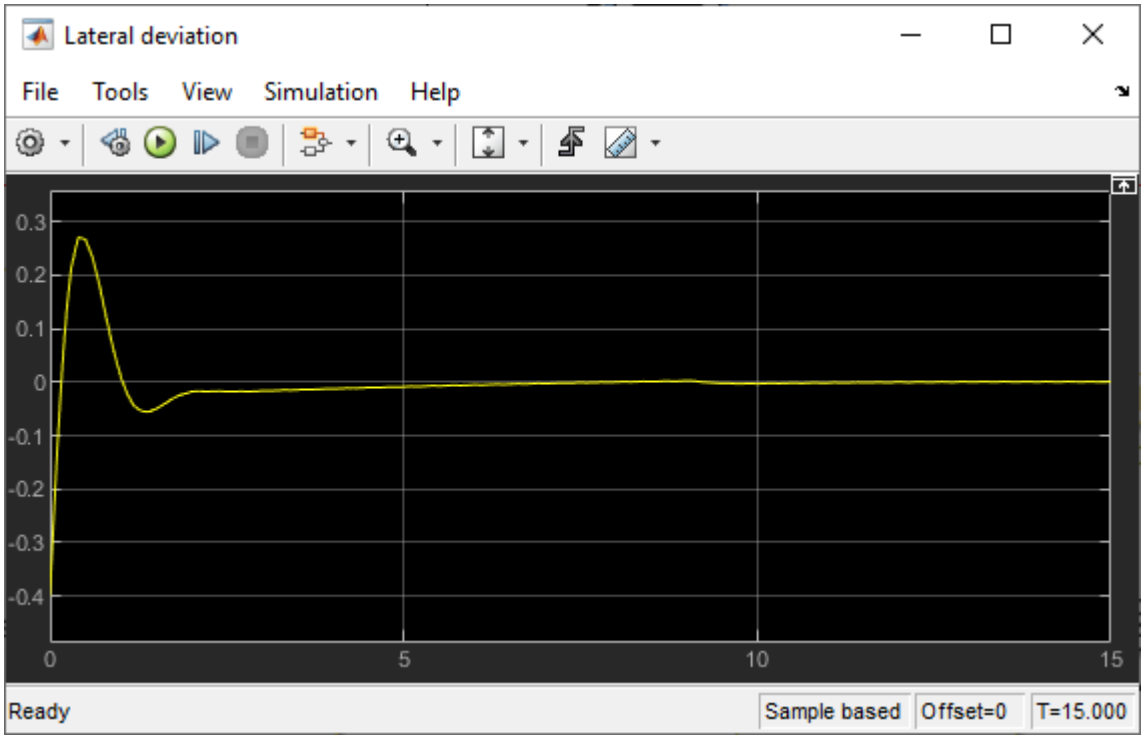
The following figure shows the training results.



Since **Total Number of Steps** equals the product of **Episode Number** and **Episode Steps**, each training episode runs to the end without early termination. Therefore, the Constraint Enforcement block ensures that the lateral deviation never violates its constraints.

Run the trained agent and view the simulation results.

```
e1_initial = -0.4;
e2_initial = 0.2;
sim(mdl);
```




```
bdclose('rlLearnConstraintLKA')
bdclose('rlLKAwithConstraint')
```

Local Reset Function

```
function in = localResetFcn(in)
% Set initial lateral deviation to random value.
in = setVariable(in,'e1_initial', 0.5*(-1+2*rand));
% Set initial relative yaw angle to random value.
in = setVariable(in,'e2_initial', 0.1*(-1+2*rand));
end
```

See Also

Blocks
Constraint Enforcement | RL Agent

Related Examples

- “Constraint Enforcement for Control Design” on page 16-2
- “Learn and Apply Constraints for PID Controllers” on page 16-10
- “Train Reinforcement Learning Agent with Constraint Enforcement” on page 16-18
- “Train RL Agent for Adaptive Cruise Control with Constraint Enforcement” on page 16-29

Model Verification

- “Monitor Linear System Characteristics in Simulink Models” on page 17-2
- “Define Linear System for Model Verification Blocks” on page 17-3
- “Verifiable Linear System Characteristics” on page 17-4
- “Verify Model at Default Simulation Snapshot Time” on page 17-5
- “Verify Model at Multiple Simulation Snapshots” on page 17-13
- “Verify Model Using Simulink Control Design and Simulink Verification Blocks” on page 17-20
- “Verify Frequency-Domain Characteristics of an Aircraft” on page 17-27

Monitor Linear System Characteristics in Simulink Models

Simulink Control Design software provides Model Verification blocks to monitor time- and frequency-domain characteristics of a linear system on page 17-3 computed from a nonlinear Simulink model during simulation.

Use these blocks to:

- Verify that the linear system characteristics of any nonlinear Simulink model, including the following, remain within specified bounds during simulation:
  - Continuous- or discrete-time models
  - Multi-rate models
  - Models with time delays, represented using exact delay or Padé approximation
  - Discretized linear models computed from continuous-time models
  - Continuous-time models computed from discrete-time models
  - Resampled discrete-time models

  The linear system can be Single-Input Single-Output (SISO) or Multi-Input Multi-Output (MIMO).
- View specified bounds and bound violations on linear analysis plots.

  Tip These blocks are same as the Linear Analysis Plots blocks except for different default settings of the bound parameters.
- Save the computed linear system to the MATLAB workspace.

The verification blocks assert when the linear system characteristic does not satisfy a specified bound, i.e., assertion fails. A warning message, reporting the assertion failure, appears at the MATLAB prompt. When assertion fails, you can:

- Stop the simulation and bring that block into focus.
- Evaluate a MATLAB expression.

You can use these blocks with the Simulink Model Verification blocks to design complex logic for assertion. For an example, see “Verify Model Using Simulink Control Design and Simulink Verification Blocks” on page 17-20.

You can use the Verification Manager tool in the Signal Builder to construct simulation tests for your model. For an example, see “Verify Frequency-Domain Characteristics of an Aircraft” on page 17-27.

Note These blocks do not support code generation and can only be used in `Normal` simulation mode.

Define Linear System for Model Verification Blocks

To assert that the linear system characteristics satisfy specified bounds, the Model Verification blocks compute a linear system from a nonlinear Simulink model.

For the software to compute a linear system, you must specify:

- Linearization inputs and outputs using the **Linearization inputs/outputs** block parameter.

  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 using the **Linearize on** block parameter.

  - Default simulation snapshot time. Typically, simulation snapshots are the times when your model reaches steady state.
  - Multiple simulation snapshots.
  - Trigger-based simulation events

For more information, see the following examples:

- “Verify Model at Default Simulation Snapshot Time” on page 17-5
- “Verify Model at Multiple Simulation Snapshots” on page 17-13

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

You can specify bound parameters on the **Bounds** tab of each block. You can also specify the bounds programmatically. For more information, see the corresponding block reference pages.

Verify Model at Default Simulation Snapshot Time

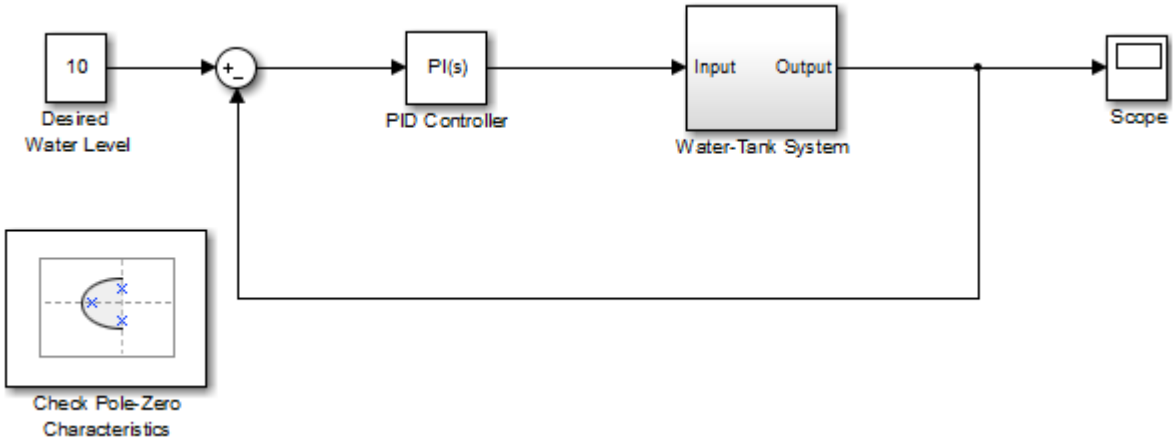
This example shows how to assert that bounds on the linear system characteristics of a nonlinear Simulink model, computed at the default simulation snapshot time of 0, are satisfied during simulation.

1. Open a nonlinear Simulink model. For example:

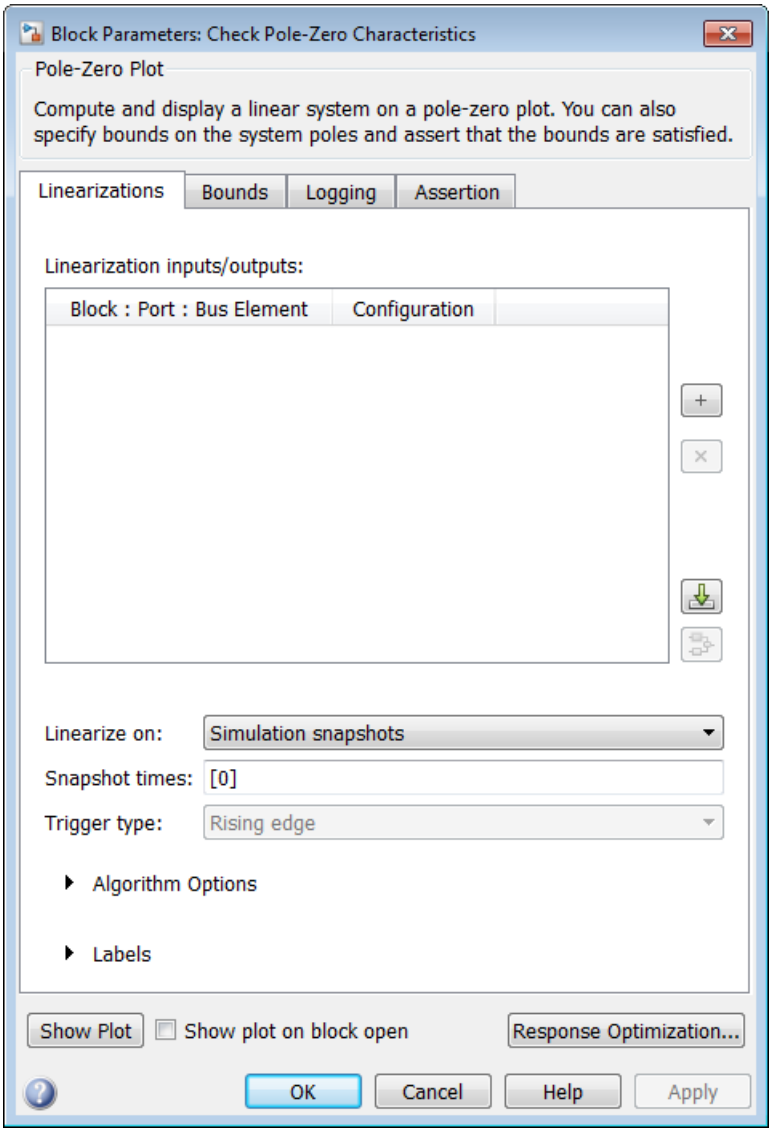
   ```
   watertank
   ```

2. Open the Simulink Library Browser. In the Simulink Editor, on the **Simulation** tab, click **Library Browser**.
3. Add a model verification block to the Simulink model.
   a. In the **Simulink Control Design** library, select **Model Verification**.
   b. Drag and drop a block, such as the Check Pole-Zero Characteristics block, into the Simulink Editor.

   The model now resembles the following figure.



4. Double-click the block to open the Block Parameters dialog box.



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

5 Specify the linearization input and output to compute the closed-loop poles and zeros.

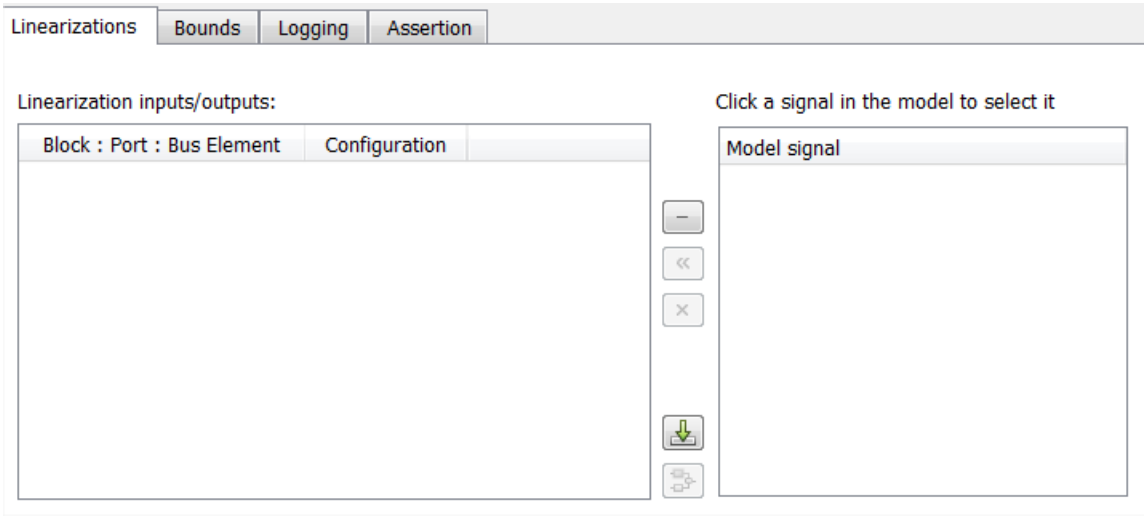
Tip If you have defined the linearization input and output in your Simulink model, click to automatically populate the **Linearization inputs/outputs** table with I/Os from the model.

---

a To specify an input:

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

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

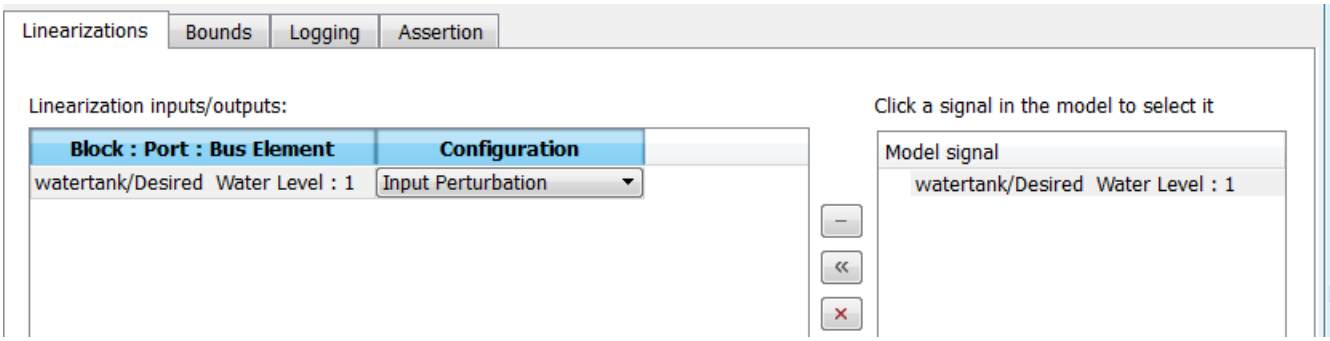


ii In the Simulink model, click the output signal of the `Desired Water Level` block to select it.

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

Tip You can select multiple signals at once in the Simulink model. All selected signals appear in the **Click a signal in the model to select it** area.

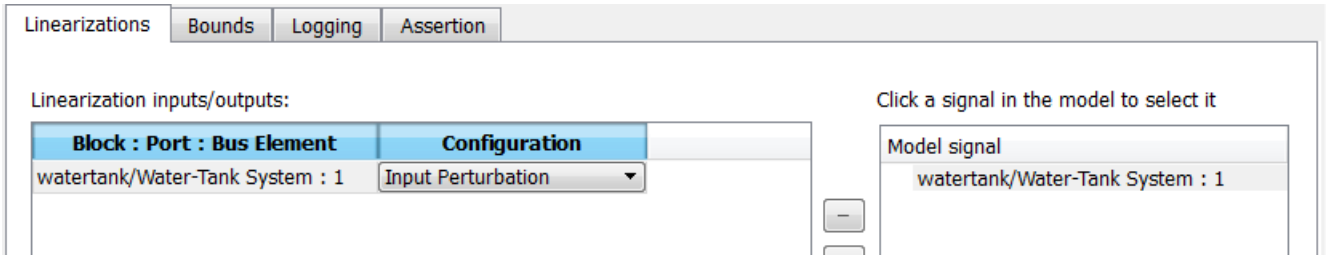
iii Click « to add the signal to the **Linearization inputs/outputs** table.



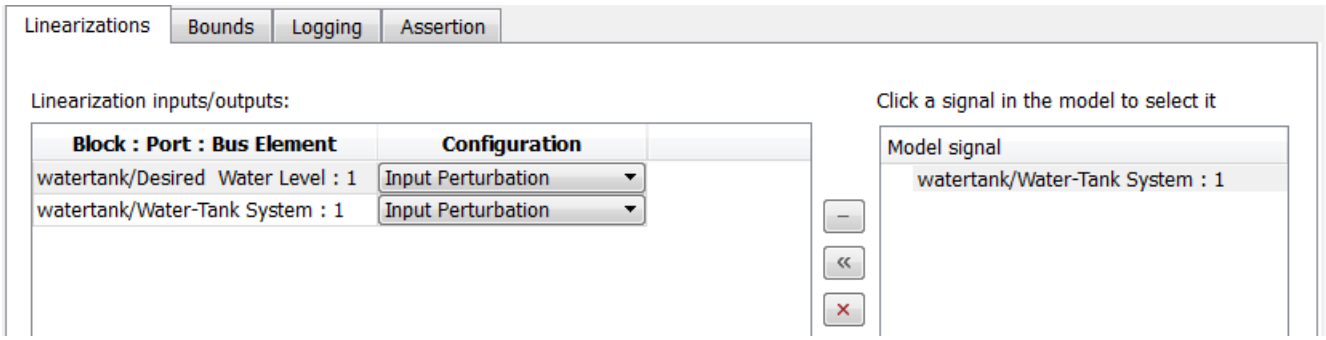
b To specify an output:

i In the Simulink model, click the output signal of the `Water-Tank System` block to select it.

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

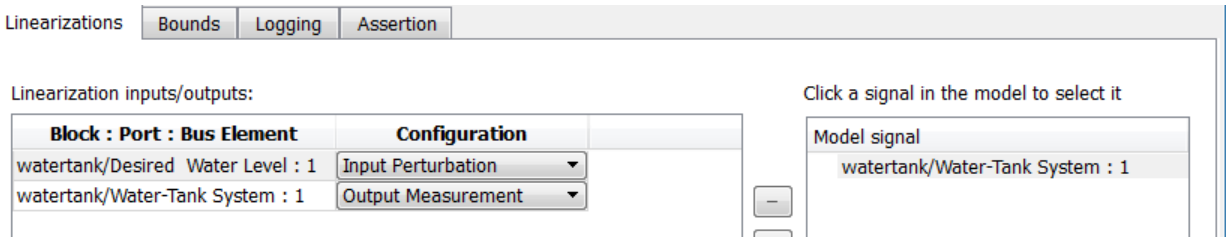


ii Click « to add the signal to the **Linearization inputs/outputs** table.



Note To find the location in the Simulink model corresponding to a signal in the **Linearization inputs/outputs** table, select the signal in the table and click .

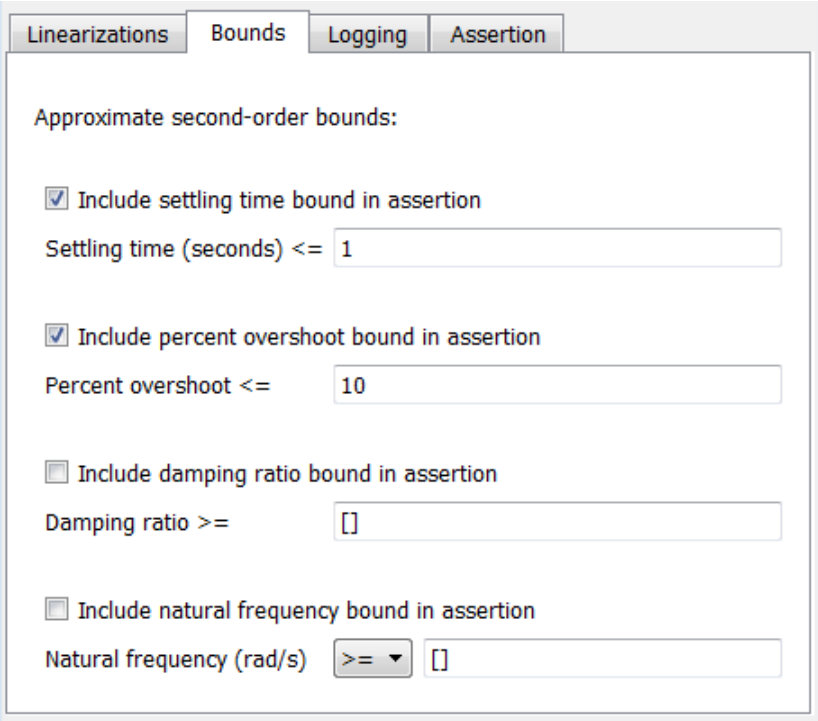
iii In the **Configuration** drop-down list of the **Linearization inputs/outputs** table, select `Output Measurement` for **watertank/Water-Tank System: 1**.



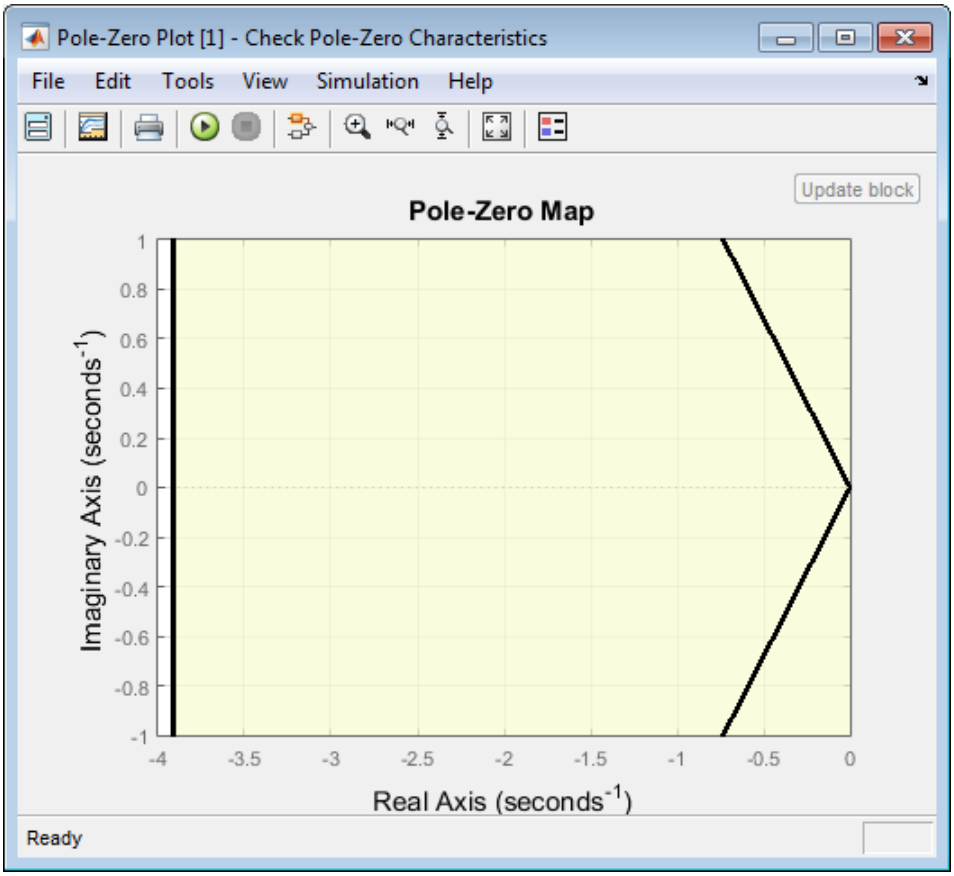
Note The I/Os include the feedback loop in the Simulink model. The software computes the poles and zeros of the closed-loop system.

iv Click − to collapse the **Click a signal in the model to select it** area.

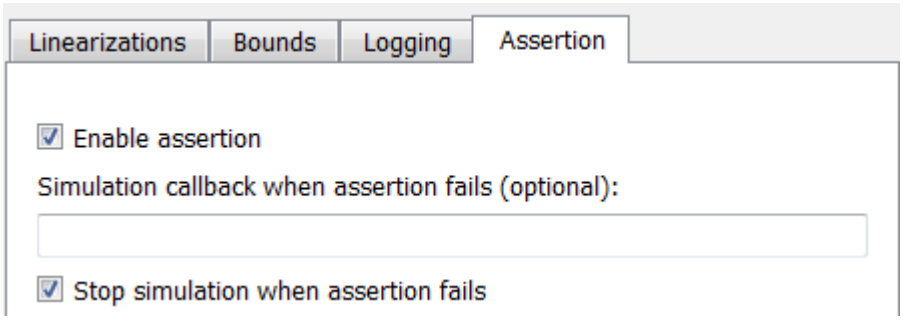
6 Specify bounds for assertion. In this example, you use the default approximate second-order bounds, specified in **Bounds** tab of the Block Parameters dialog box.



View the bounds on the pole-zero map by clicking **Show Plot** to open a plot window.



7 Stop the simulation if assertion fails by selecting **Stop simulation when assertion fails** in the **Assertion** tab.



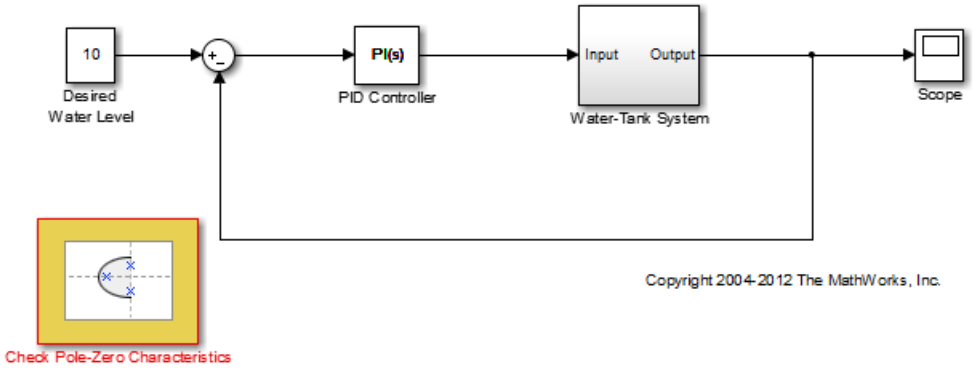
8 Click **Apply** to apply all changed settings to the block.

9 Simulate the model by clicking in the plot window.

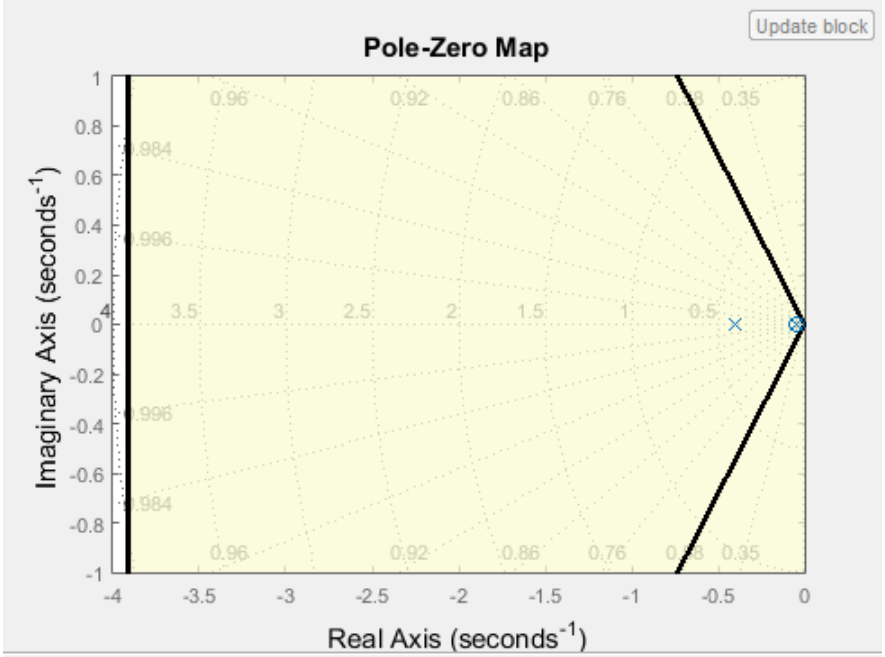
Alternatively, you can simulate the model from the Simulink Editor.

The software linearizes the portion of the model between the linearization input and output at the default simulation time of 0, specified in **Snapshot times** block parameter. When the software detects that a pole violates a specified bound, the simulation stops. The Diagnostics Viewer opens reporting the block that asserts.

Click **Open** to highlight the block that asserts in the Simulink model.



The closed-loop pole and zero locations of the computed linear system appear as x and o markings in the plot window. You can also view the bound violation in the plot.


Update block
Pole-Zero Map
Imaginary Axis (seconds-1)
Real Axis (seconds-1)
1
0.8
0.6
0.4
0.2
0
-0.2
-0.4
-0.6
-0.8
-1
-4
-3.5
-3
-2.5
-2
-1.5
-1
-0.5
0
0.96
0.92
0.86
0.76
0.35
0.984
0.996
3.5
3
2.5
2
1.5
1
0.5

Verify Model at Multiple Simulation Snapshots

This example shows how to:

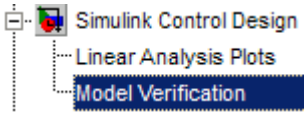
- Add multiple bounds.
- Check that the linear system characteristics of a nonlinear Simulink model satisfy the bounds at multiple simulation snapshots
- Modify bounds graphically
- Disable bounds during simulation

1. Open a nonlinear Simulink model. For example:

   ```
   watertank
   ```

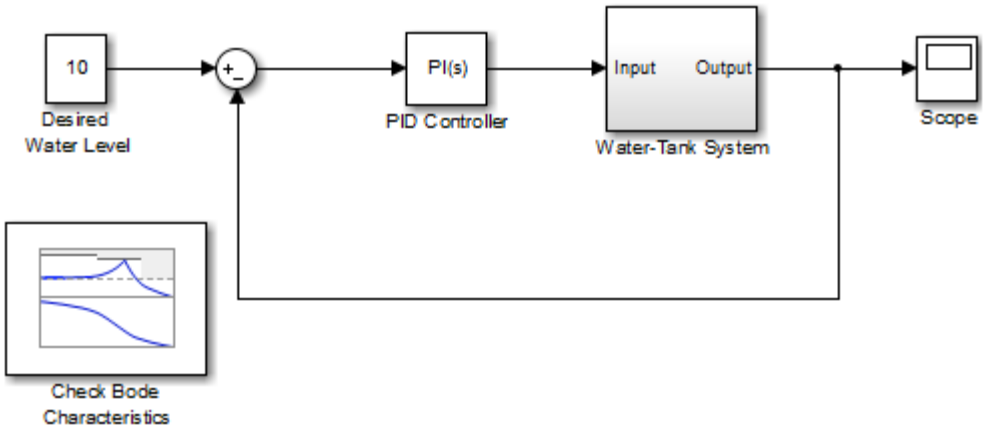
2. Open the Simulink Library Browser. In the Simulink Editor, on the **Simulation** tab, click **Library Browser**.
3. Add a model verification block to the Simulink model.

   a. In the **Simulink Control Design** library, select **Model Verification**.

   b. Drag and drop a block, such as the Check Bode Characteristics block, into the Simulink Editor.

   The model now resembles the following figure.



4. Double-click the block to open the Block Parameters dialog box.

   To learn more about the block parameters, see the block reference pages.
5. Specify the linearization I/O points.

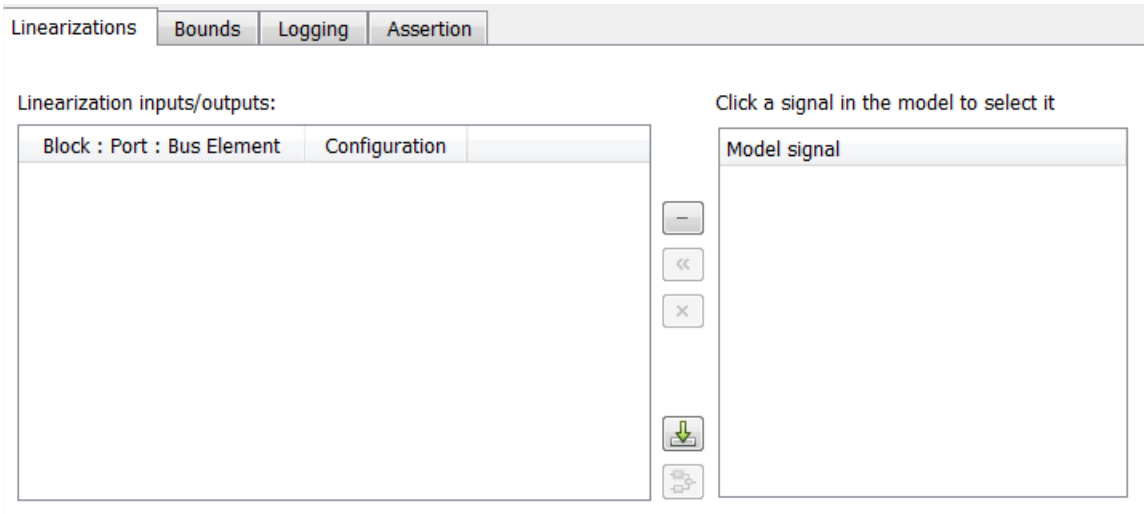
   The linear system is computed for the Water-Tank System.

Tip If your model already contains I/O points, the block automatically detects these points and displays them. Click at any time to update the **Linearization inputs/outputs** table with I/Os from the model.

a To specify an input:

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

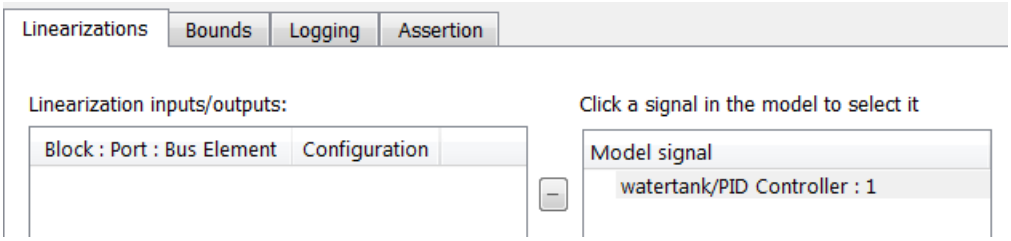
The Block Parameters dialog expands to display a **Click a signal in the model to select it** area.



Tip You can select multiple signals at once in the Simulink model. All selected signals appear in the **Click a signal in the model to select it** area.

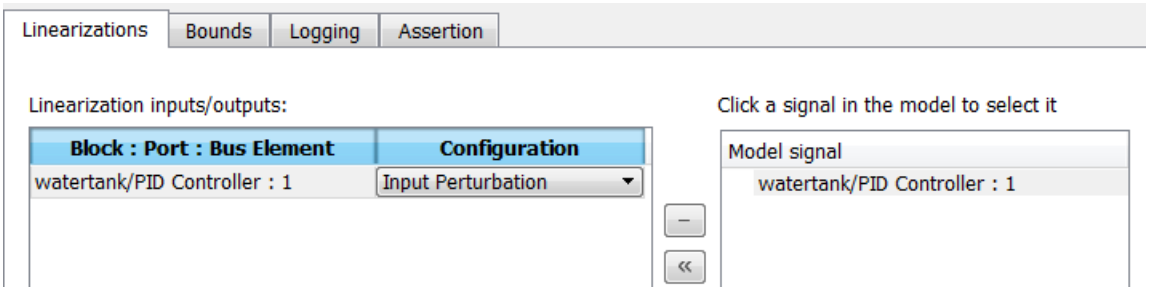
ii In the Simulink model, click the output signal of the `PID Controller` block to select it.

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



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

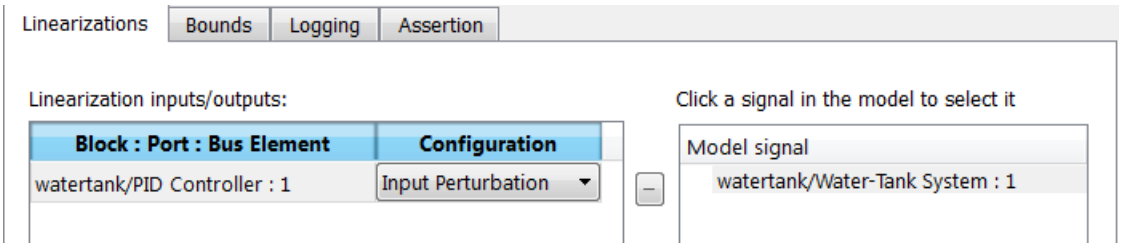
To remove a signal from the **Linearization inputs/outputs** table, select the signal and click .



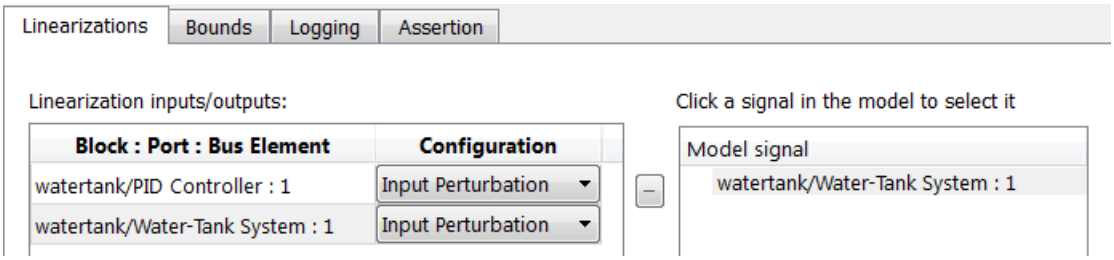
b To specify an output:

i In the Simulink model, click the output signal of the `Water-Tank System` block to select it.

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



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

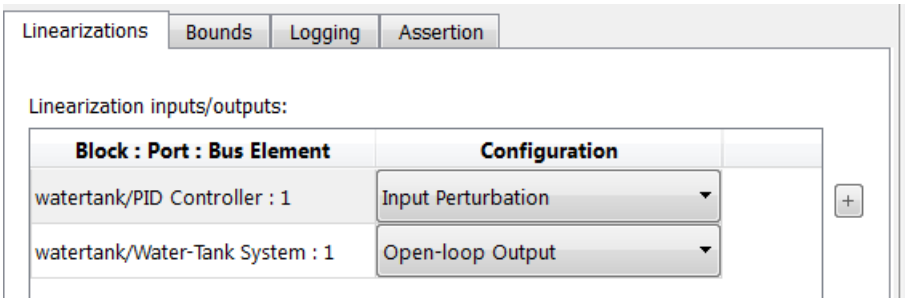


To remove a signal from the **Linearization inputs/outputs** table, select the signal and click ×.

Note To find the location in the Simulink model corresponding to a signal in the **Linearization inputs/outputs** table, select the signal in the table and click .

iii In the **Configuration** drop-down list of the **Linearization inputs/outputs** table, select `Open-loop Output` for **watertank/Water-Tank System : 1**.

The **Linearization inputs/outputs** table now resembles the following figure.



c Click − to collapse the **Click a signal in the model to select it** area.

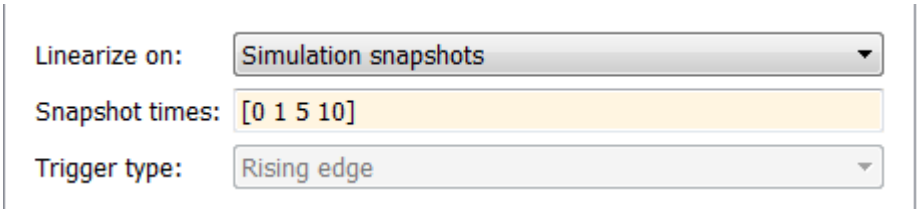
Tip Alternatively, before you add the Linear Analysis Plots block, right-click the signals in the Simulink model and select **Linear Analysis Points > Input Perturbation** and **Linear Analysis Points > Open-loop Output**. Linearization I/O annotations appear in the model and the selected signals appear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

---

6 Specify simulation snapshot times.

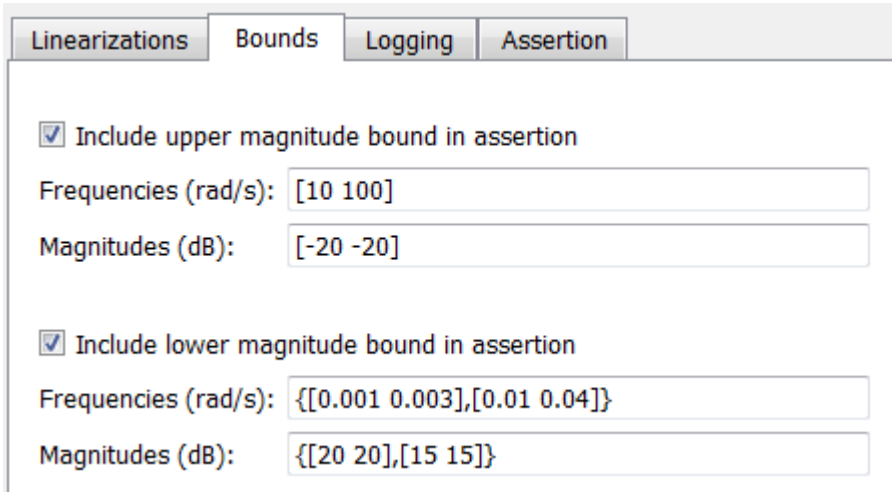
a In the **Linearizations** tab, verify that `Simulation snapshots` is selected in **Linearize on**.

b In the **Snapshot times** field, type `[0 1 5 10]`.



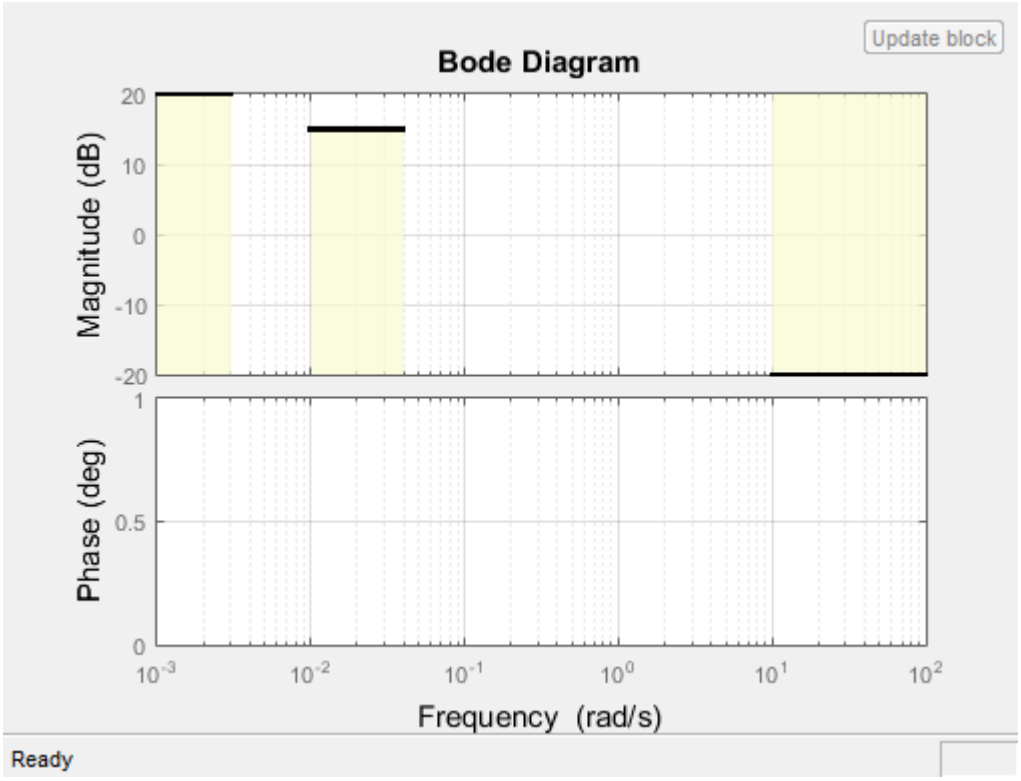
7 Specify multiple bound segments for assertion in the **Bounds** tab of the Block Parameters dialog box. In this example, enter the following lower magnitude bounds:

- **Frequencies (rad/s)** — `{[0.001 0.003],[0.01 0.04]}`
- **Magnitudes (dB)** — `{[20 20],[15 15]}`



Click **Apply** to apply the parameter changes to the block.

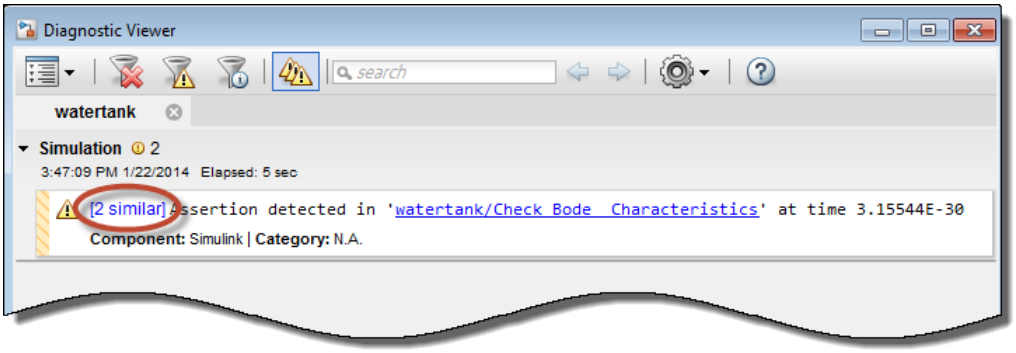
Click **Show Plot** to view the bounds on the Bode magnitude plot.



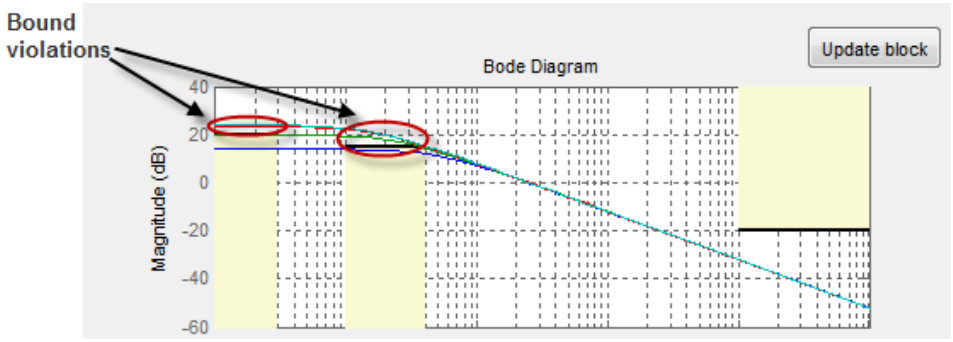
8 Simulate the model by clicking in the plot window.

Alternatively, you can simulate the model from the Simulink Editor.

The software linearizes the portion of the model between the linearization input and output at the simulation times of 0,1, 5 and 10. When the software detects that the linear system computed at times 0 and 1 violate a specified lower magnitude bound, warning messages appear in the Diagnostic Viewer window. Click the link at the bottom of the Simulink model to open this window. Click the link in the window to view the details of the assertion.

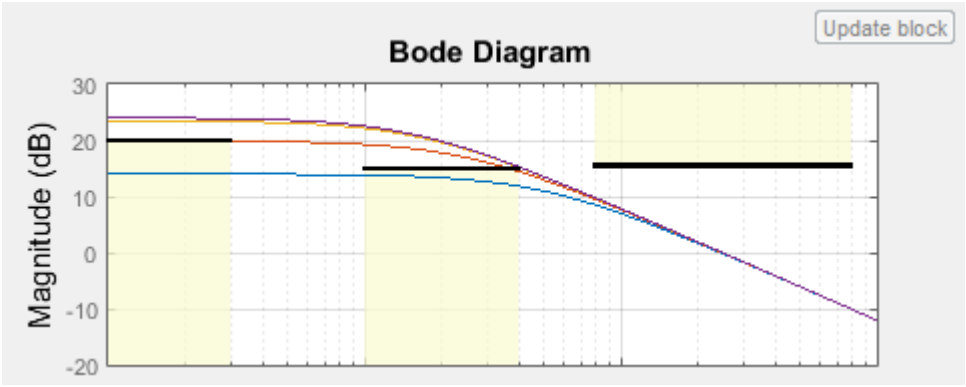


You can also view the bound violations on the plot window.

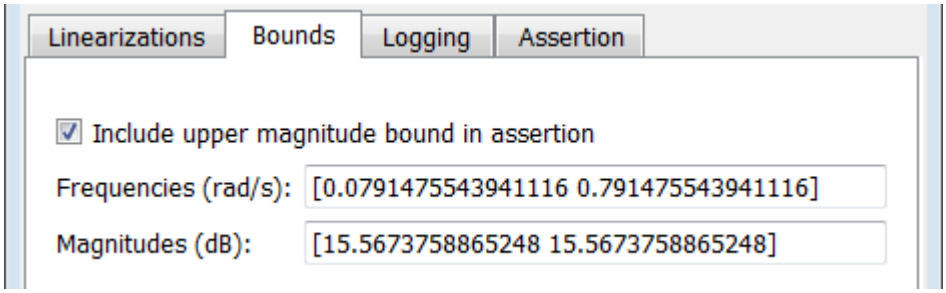


9 Modify a bound graphically. For example, to modify the upper magnitude bound graphically:

a In the plot window, click the bound segment to select it and then drag it to the desired location.

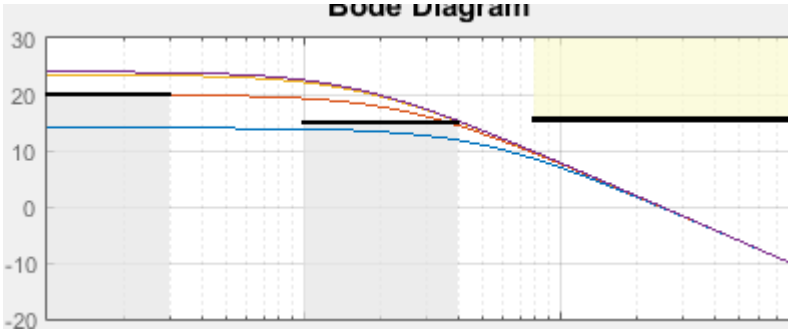


b Click **Update block** to update the new values in the Bounds tab of the Block Parameters dialog box.



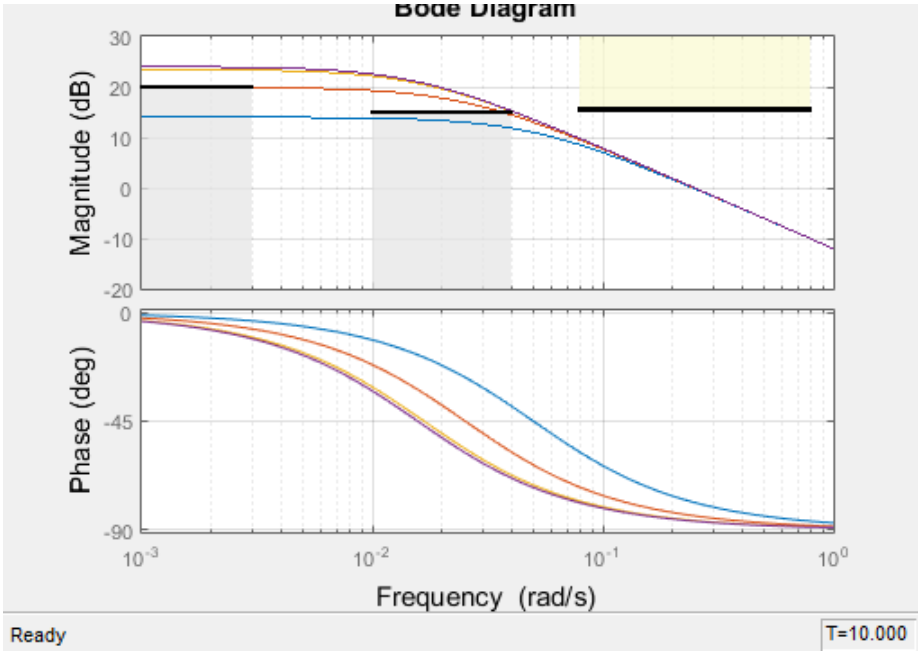
10 Disable the lower bounds to exclude them from asserting. Clear the **Include lower magnitude bounds in assertion** option in the Block Parameters dialog box. Then, click **Apply**.

The lower bounds are now grey-out in the plot window, indicating that they are excluded from assertion.



11 Resimulate the model to check if bounds are satisfied.

The software satisfies the specified upper magnitude bound, and therefore the software no longer reports an assertion failure.



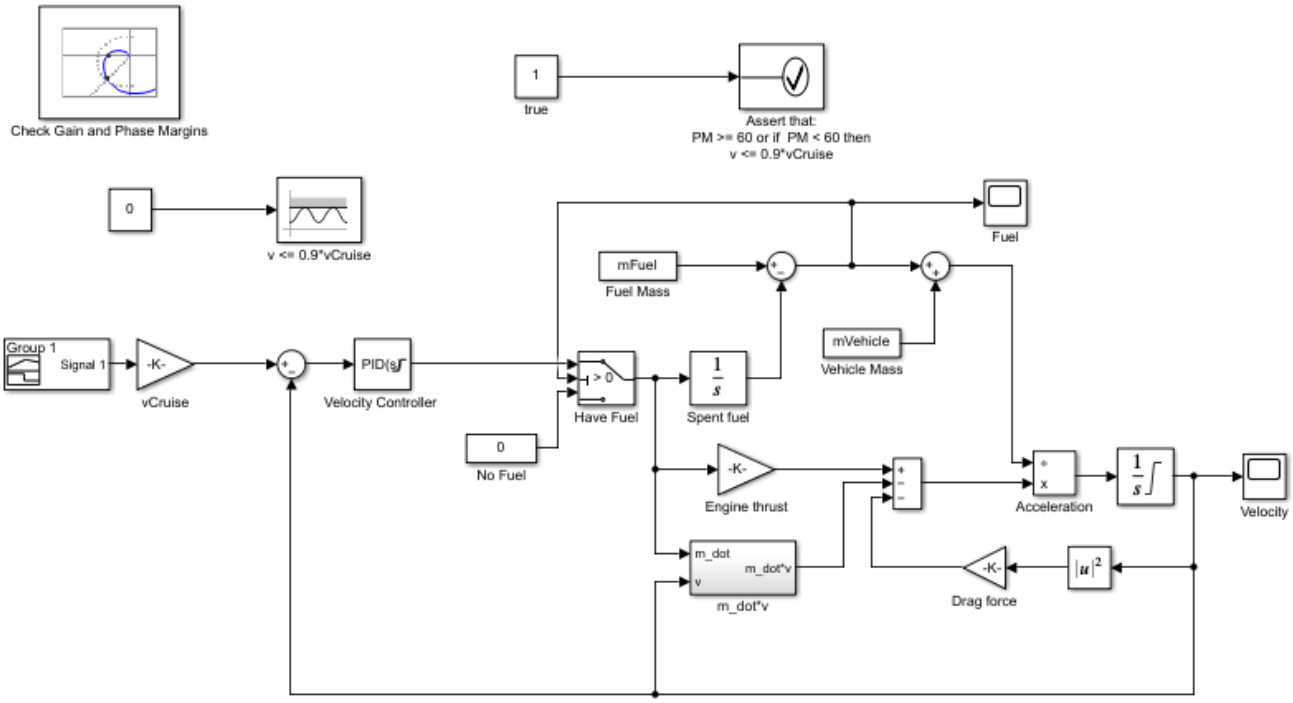
Verify Model Using Simulink Control Design and Simulink Verification Blocks

This example shows how to use a combination of Simulink® Control Design™ Simulink verification blocks to assert that the characteristics of a linear system for an aircraft satisfy one of the following conditions.

- Phase margin greater than 60 degrees
- Phase margin less than 60 degrees with velocity less than or equal to 90% of the cruise velocity

Open the aircraft Simulink model.

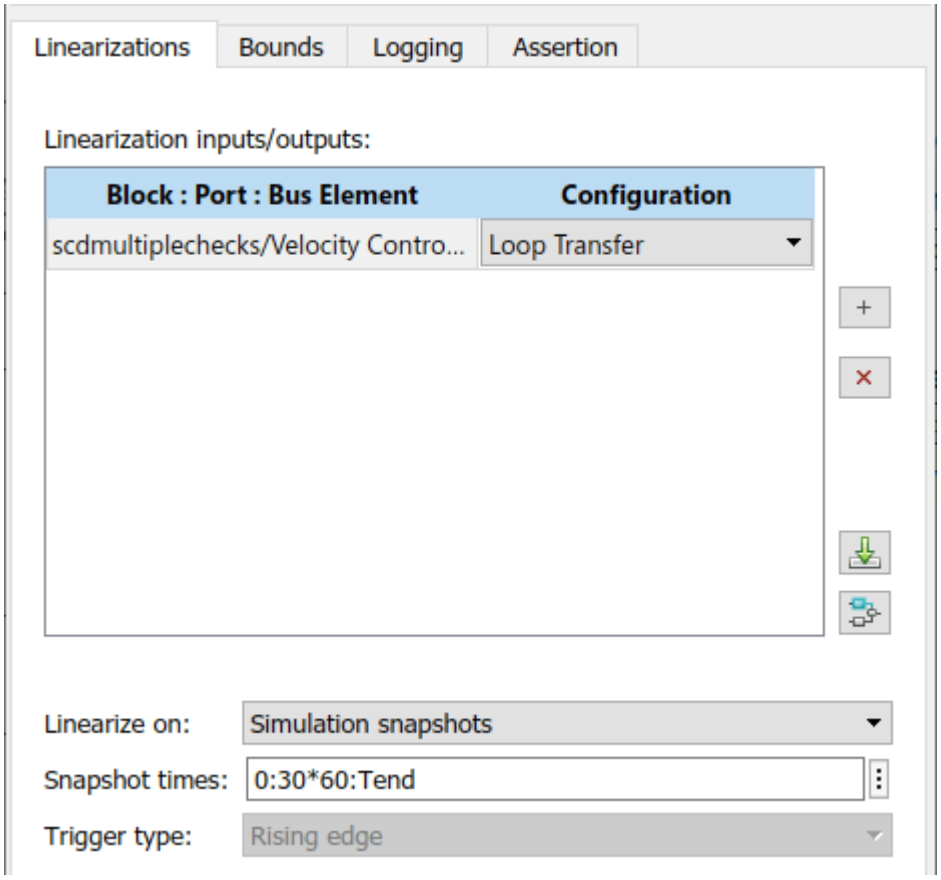
```
open_system('scdmultiplechecks')
```



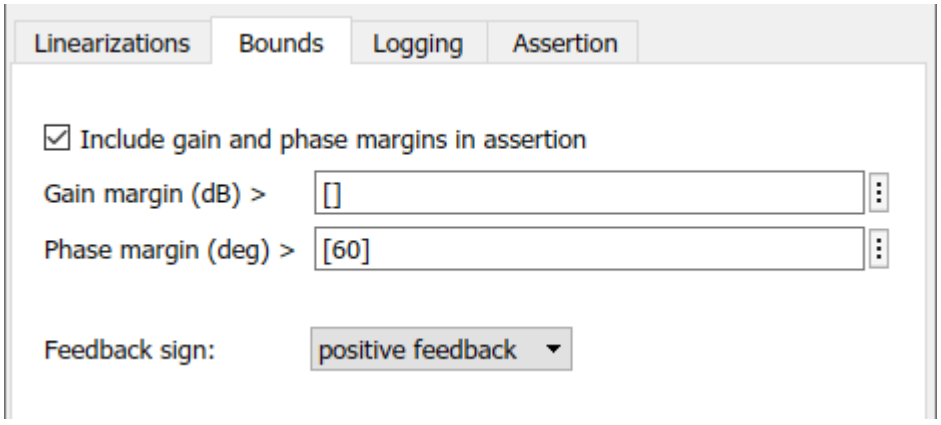
The aircraft model is based on a long-haul passenger aircraft flying at cruising altitude and speed. The aircraft starts with a full fuel load and follows a pre-specified eight-hour velocity profile. The model is a simplified version of a velocity control loop, which adjusts the fuel flow rate to control the aircraft velocity.

The 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 complex logic for assertion.

The Check Gain and Phase Margins block is configured to linearize the loop seen by the Velocity Controller block every 30 minutes of simulated time. To view the linearization settings, open the Check Gain and Phase Margins plot and open the **Linearization** tab.

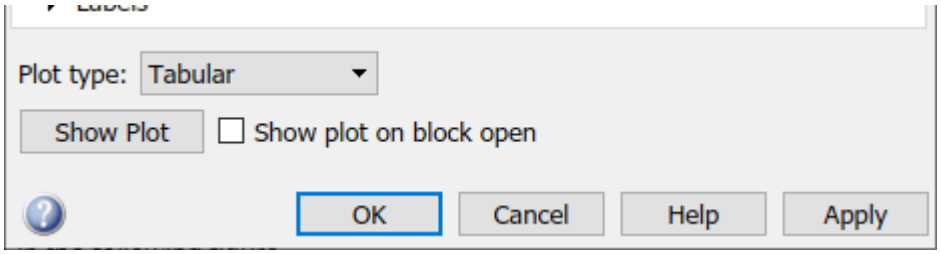


The Check Gain and Phase Margins block is configured to perform an assertion. The assertion fails when the phase margin of the linearized system is greater than 60 degrees. You can view the phase margin bound settings on the **Bounds** tab.



Since the loop seen by the controller contains the summation block with negative feedback, set the feedback sign for computing the phase margin `positive feedback`.

To view the computed phase margins in a tabular format during simulations, set the **Plot type** parameter to `Tabular` and click **Show Plot**.



Design assertion logic that causes the combination of verification blocks to fail their assertion when both of the following assertion conditions are false. In other words, the assertion passes if either condition is true

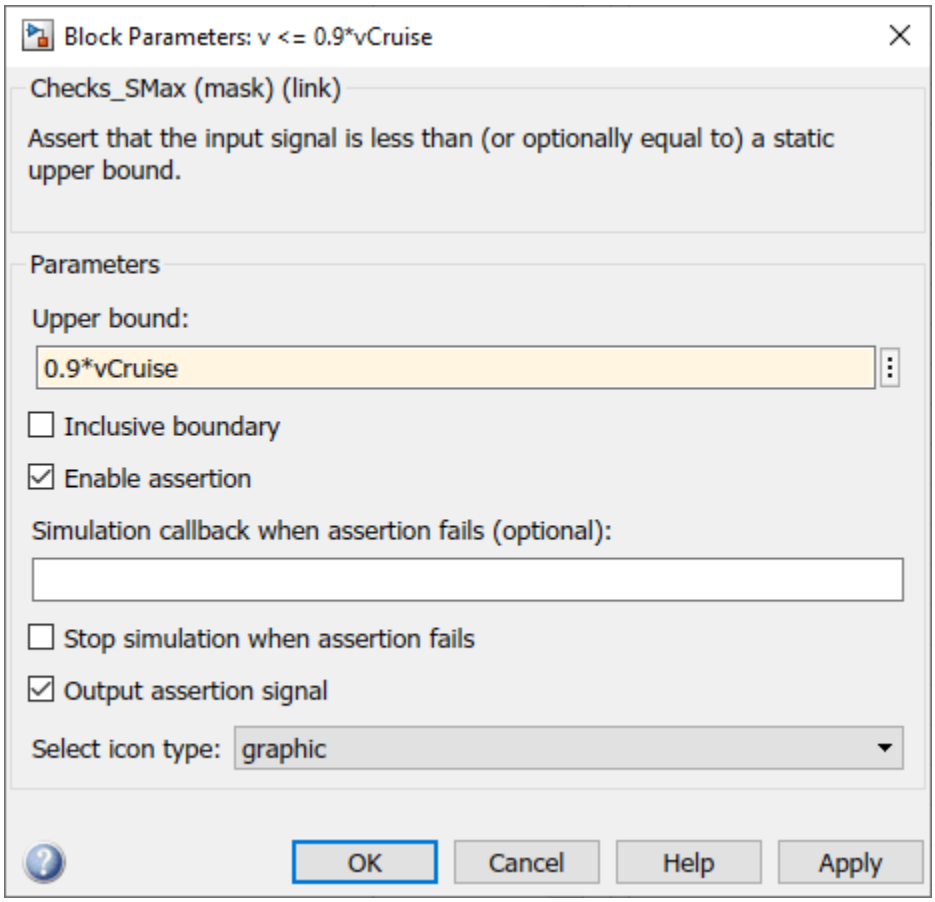
- Phase margin is greater than 60 degrees
- Phase margin is less than 60 degrees when the velocity is less than or equal to 90% of the cruise velocity

First, configure the Check Gain and Phase Margins block to output its assertion signal. To do so, on the **Assertion** tab, select **Output assertion signal**, and click **Apply**.

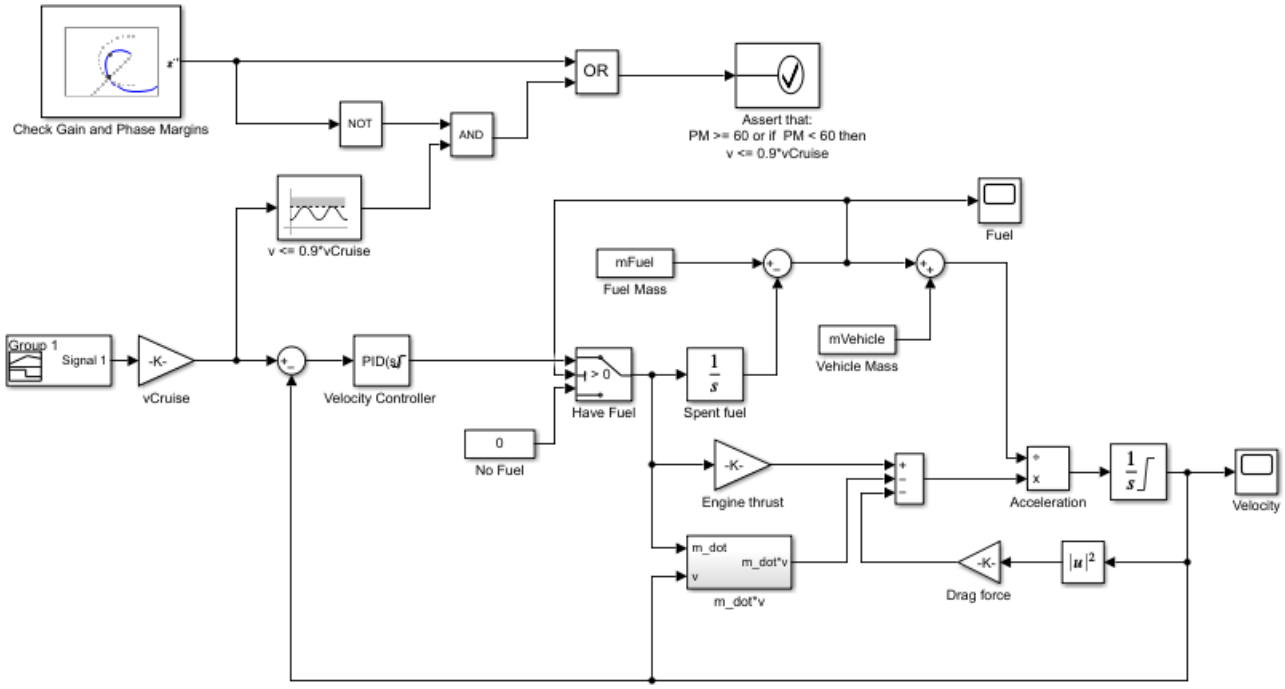
Next, configure the v <= 0.9*vCruise block to:

- Check if the aircraft velocity exceeds the cruise velocity by 0.9 times
- Add an assertion output port to the block
- Not stop the simulation when the assertion fails

To do so, open the bock and configure the parameters as shown in the following figure.



Finally, connect the verification blocks in the model as shown in the following figure. When both of assertion conditions are false, the input to the Assert that: PM >= 60 or if PM < 60 then v <= 0.9*vCruise block is zero. As a result, the block fails its assertion and stops the simulation.



The `scdmultiplechecks_final` model is configured with these settings and connections.

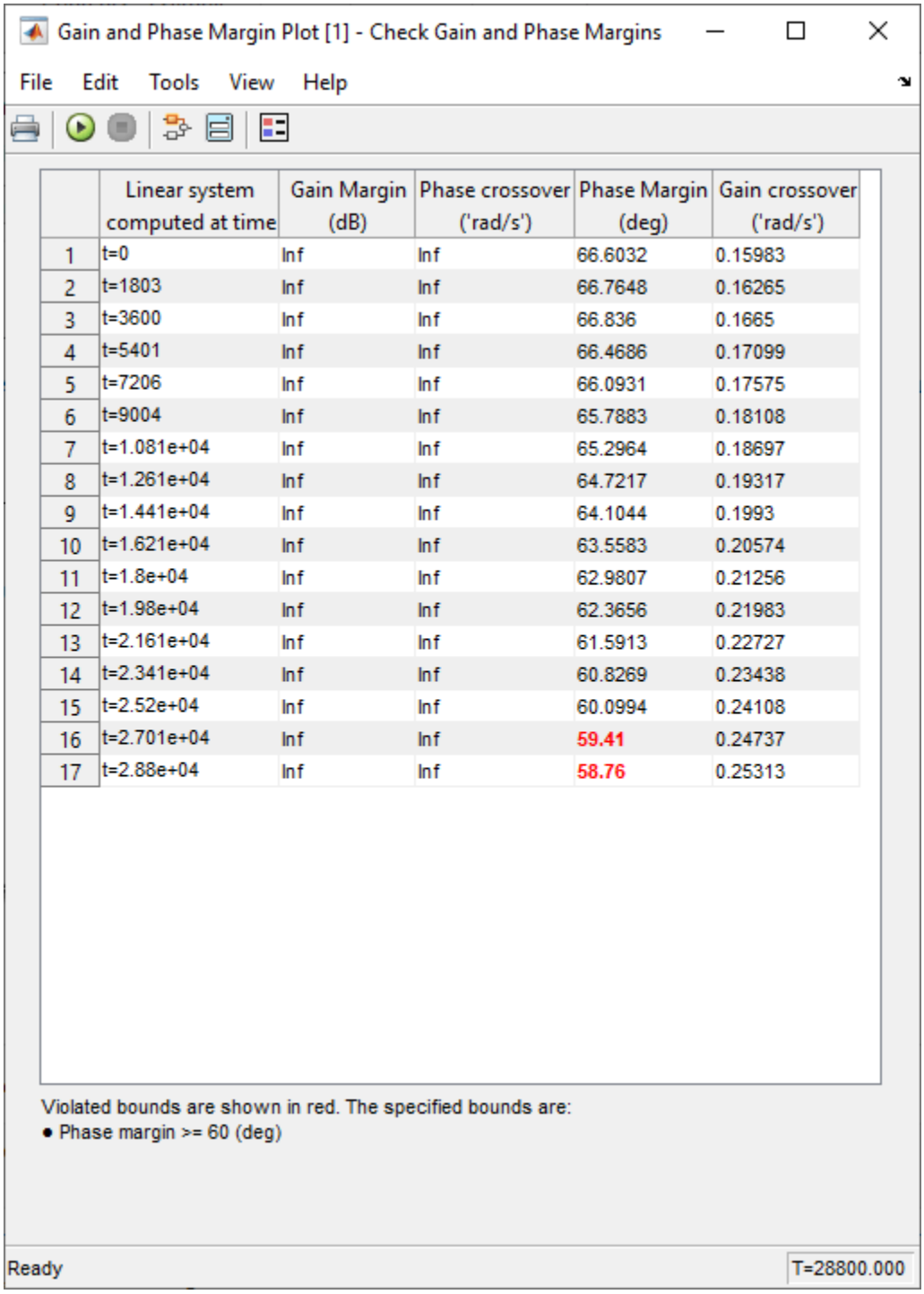
```
mdl = 'scdmultiplechecks_final';
open_system(mdl)
```

To simulate the model, run the following code.

```
sim(mdl)
```

During the simulation, the v <= 0.9*vCruise block asserts multiple times and the Check Gain and Phase Margins block asserts twice.

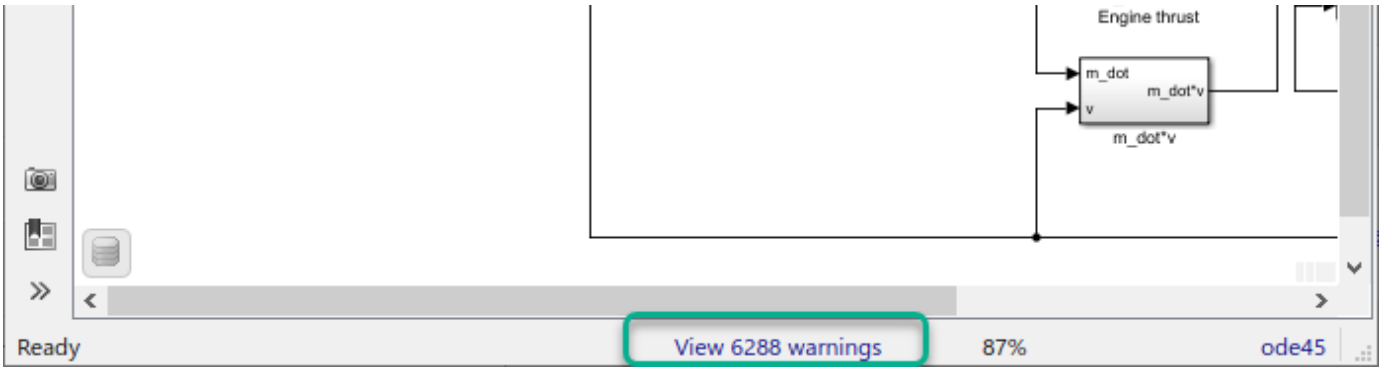
You can view the phase margins that violate the bound for the Check Gain and Phase Margins block in the table.



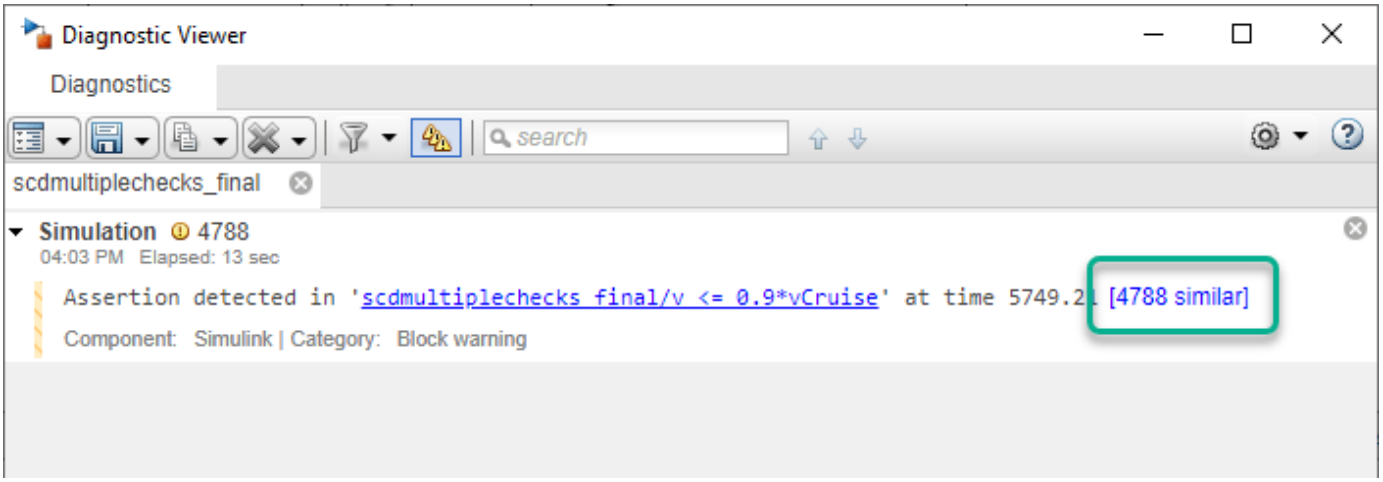
	Linear system computed at time	Gain Margin (dB)	Phase crossover ('rad/s')	Phase Margin (deg)	Gain crossover ('rad/s')
1	t=0	Inf	Inf	66.6032	0.15983
2	t=1803	Inf	Inf	66.7648	0.16265
3	t=3600	Inf	Inf	66.836	0.1665
4	t=5401	Inf	Inf	66.4686	0.17099
5	t=7206	Inf	Inf	66.0931	0.17575
6	t=9004	Inf	Inf	65.7883	0.18108
7	t=1.081e+04	Inf	Inf	65.2964	0.18697
8	t=1.261e+04	Inf	Inf	64.7217	0.19317
9	t=1.441e+04	Inf	Inf	64.1044	0.1993
10	t=1.621e+04	Inf	Inf	63.5583	0.20574
11	t=1.8e+04	Inf	Inf	62.9807	0.21256
12	t=1.98e+04	Inf	Inf	62.3656	0.21983
13	t=2.161e+04	Inf	Inf	61.5913	0.22727
14	t=2.341e+04	Inf	Inf	60.8269	0.23438
15	t=2.52e+04	Inf	Inf	60.0994	0.24108
16	t=2.701e+04	Inf	Inf	59.41	0.24737
17	t=2.88e+04	Inf	Inf	58.76	0.25313

The Assert that: PM >= 60 or if PM < 60 then v <= 0.9*vCruise does not encounter the assertion condition. Therefore, the simulation does not stop.

When a block asserts, the model generates warnings. To open the Diagnostic viewer, in the model window, click the warning link.



In the Diagnostic Viewer, you can view the details of the assertions by clicking the link.



See Also

Check Gain and Phase Margins | Check Static Upper Bound

Related Examples

- “Monitor Linear System Characteristics in Simulink Models” on page 17-2
- “Define Linear System for Model Verification Blocks” on page 17-3
- “Verify Model at Multiple Simulation Snapshots” on page 17-13

Verify Frequency-Domain Characteristics of an Aircraft

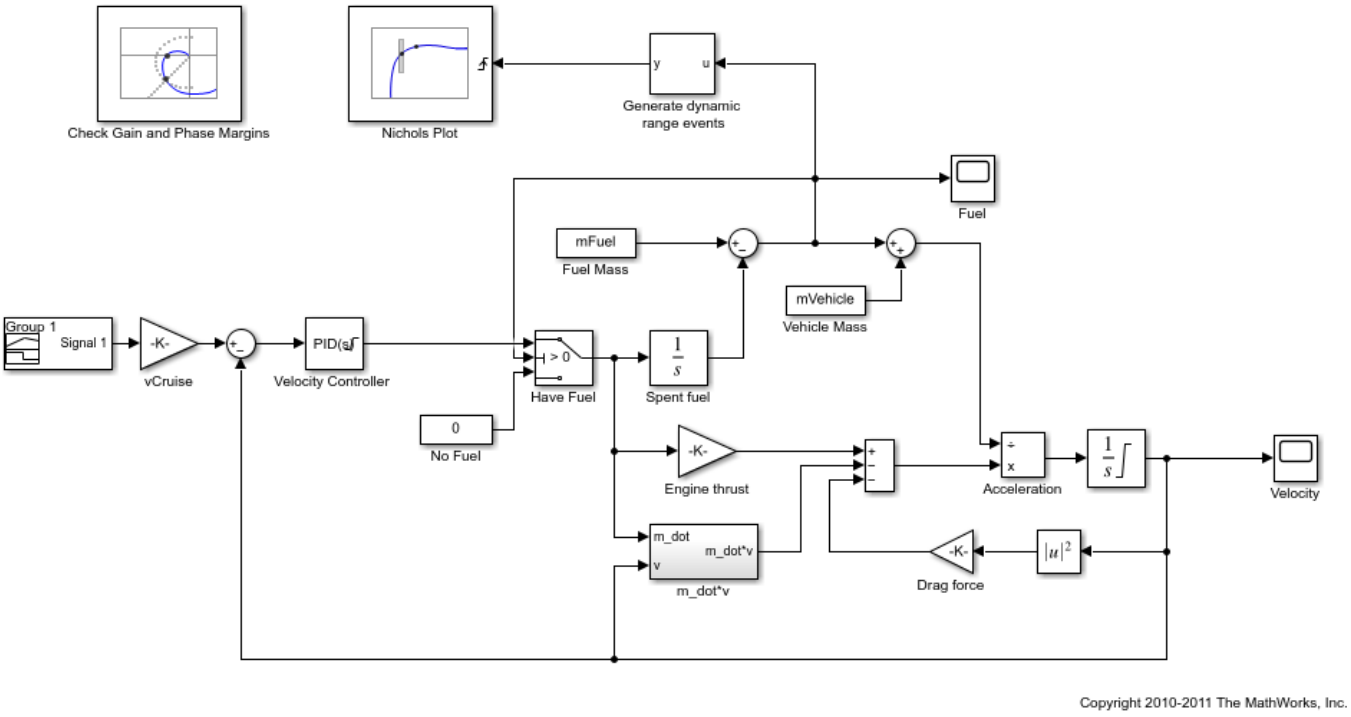
This example shows how to check the whether the linear response of a Simulink® model satisfies frequency-domain requirements during simulation. To do so, you can use the Linear Analysis Plots and Model Verification libraries of Simulink Control Design™.

In this example, you check the gain and phase margins of an aircraft velocity control loop as the fuel load changes.

Aircraft Model

Open the aircraft Simulink model.

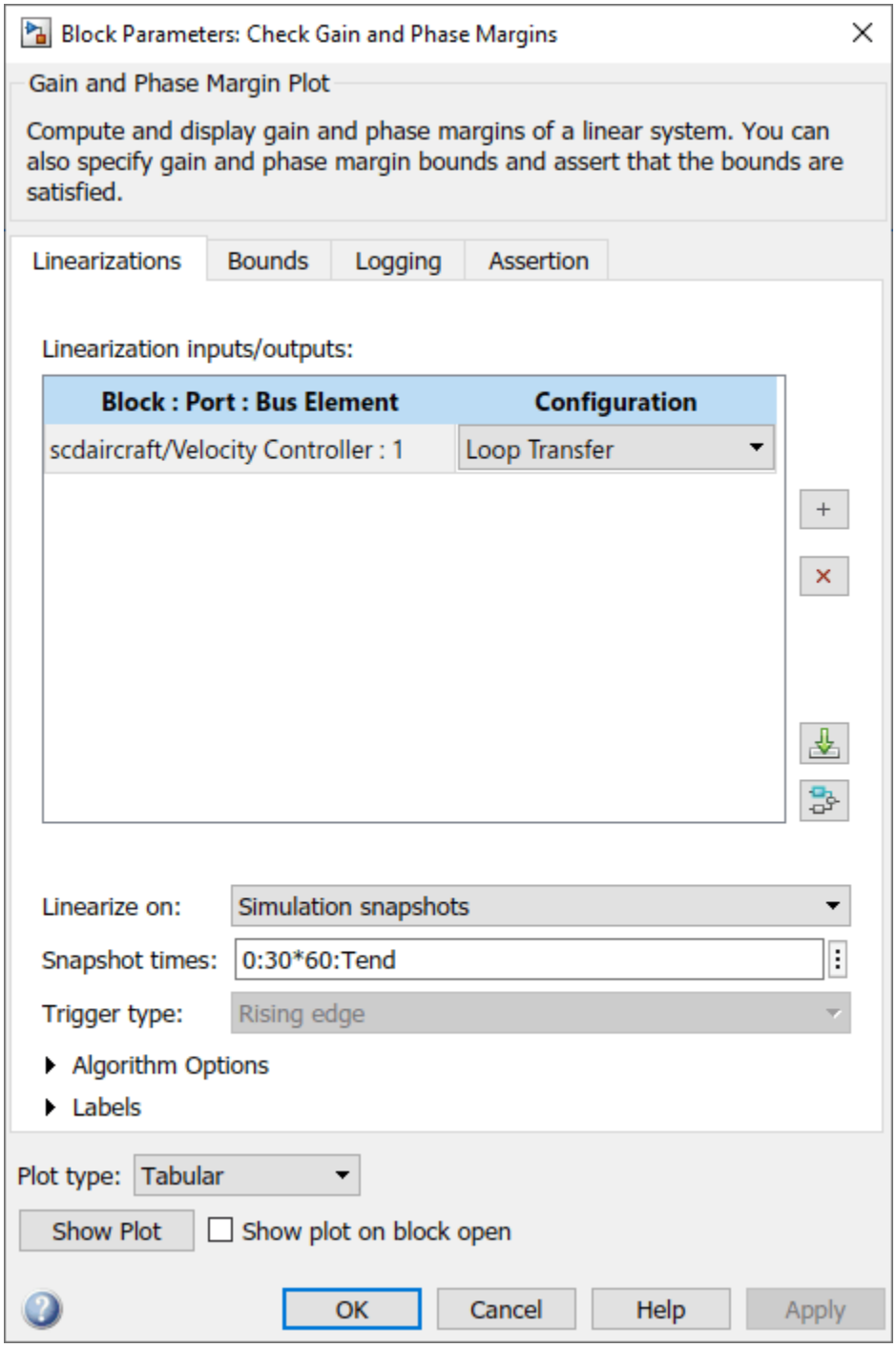
```
mdl = 'scdaircraft';
open_system(mdl)
```



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 prespecified eight hour velocity profile. The Simulink model is a simplified version of a velocity control loop that adjusts the fuel flow rate to control the aircraft velocity. The model includes elements to model fuel consumption and the resulting changes in aircraft mass as well as nonlinear draft effects limiting aircraft velocity. Constants used in the model, such as the drag coefficient, are defined in the model workspace.

Verify Loop Gain and Phase Margins

The aircraft model contains a Check Gain and Phase Margins block. This block computes the linearization of the loop seen by the Velocity Controller block every 30 minutes of simulated time. To configure



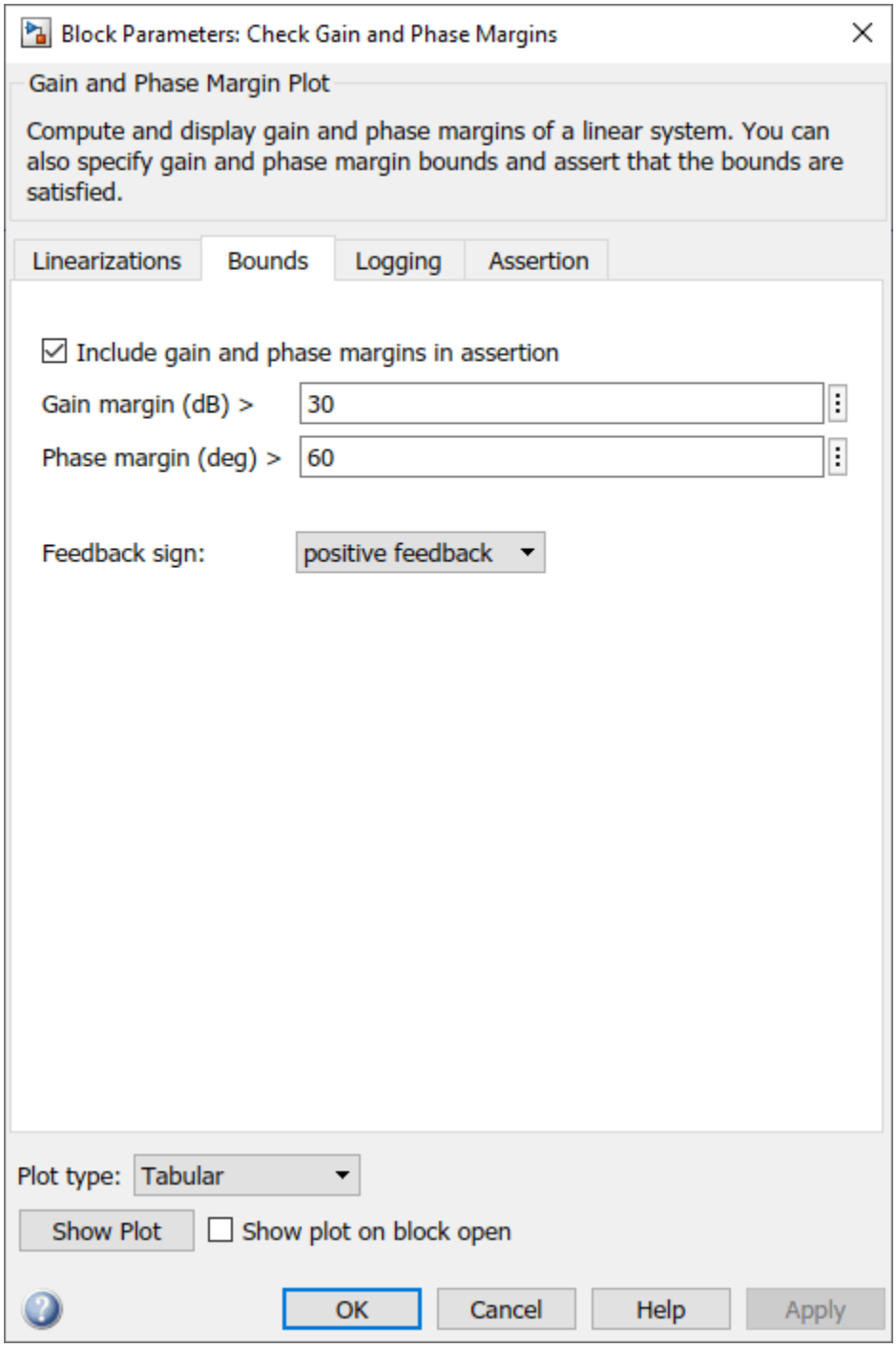
The block computes the loop gain and phase margins and checks whether the following conditions are satisfied.

- Gain margin greater than 30 dB
- Phase margin greater than 60 degrees

When computing the margins, the loop feedback sign must be specified. To determine the feedback sign, check if the linearization path defined by the linearization inputs and outputs includes the feedback summation.

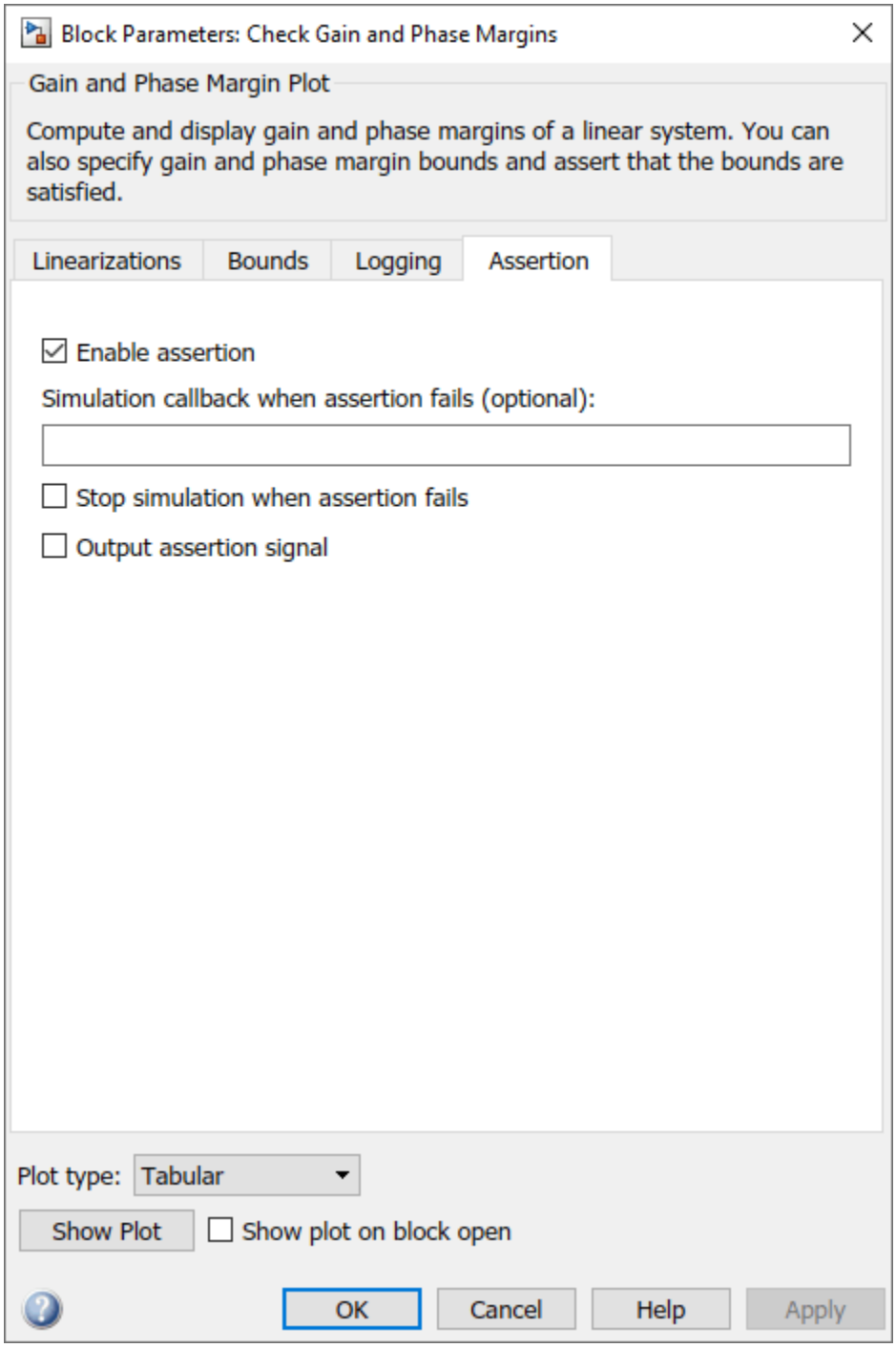
- If the path includes the summation block, use positive feedback.
- If the path does not include the summation block, use the feedback defined by the summation block.

In this case, the linearization defined in the Check Gain and Phase Margins block includes the summation block with negative feedback. Therefore, compute the gain and phase margins using a positive feedback sign.



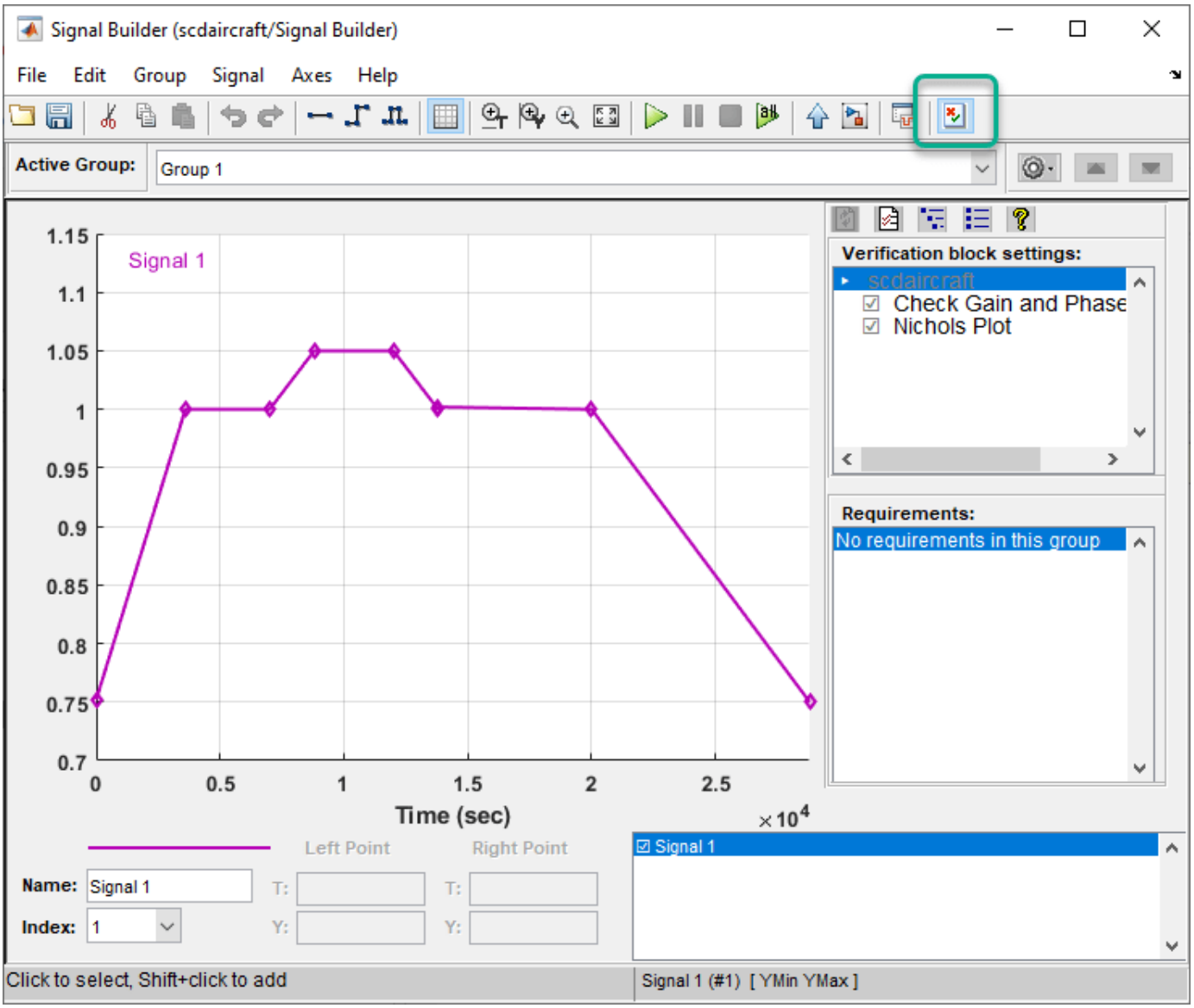
During the simulation, the block shows the computed gain and phase margins in a tabular format. To open the table, click **Show Plot**.

On the **Assertion** tab, the block is configured to throw a warning when the assertion fails, that is, when the gain and phase margins are not satisfied.



The model also includes a Nichols Plot block, which computes the loop response as the fuel mass drops during the simulation. The Generate dynamic range events block generates a rising edge whenever the fuel mass is a multiple of 10% of the maximum fuel mass. These rising edges trigger a linearization and display the results on the Nichols plot. To view the Nichols plot, open the Nichols Plot block and click **Show Plot**.

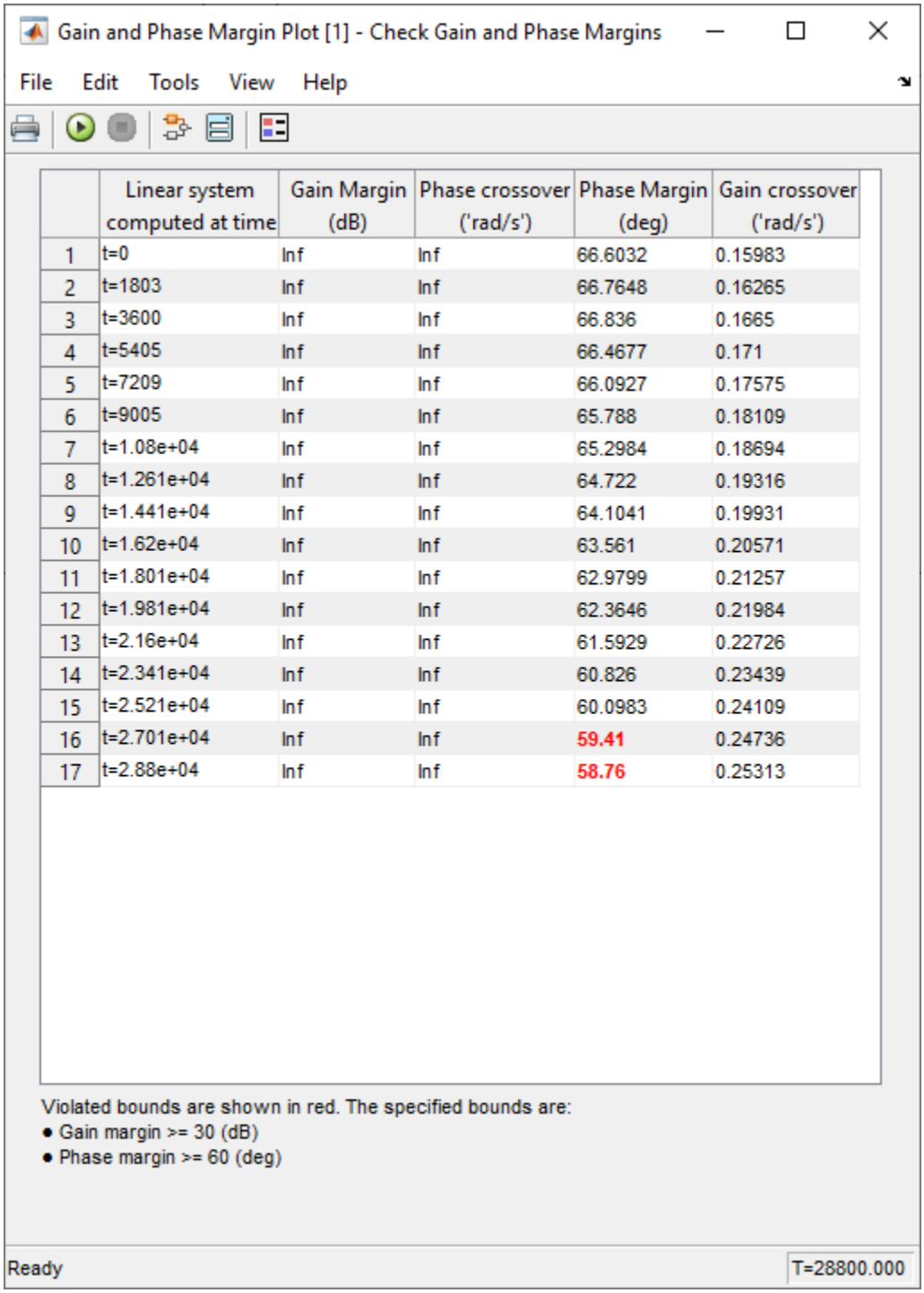
The Verification Manager displays the check blocks included in the model. To open the Verification Manager, double-click the Signal Builder block and select the **Show verification settings** icon. For more information, see “Construct Simulation Tests by Using the Verification Manager”.



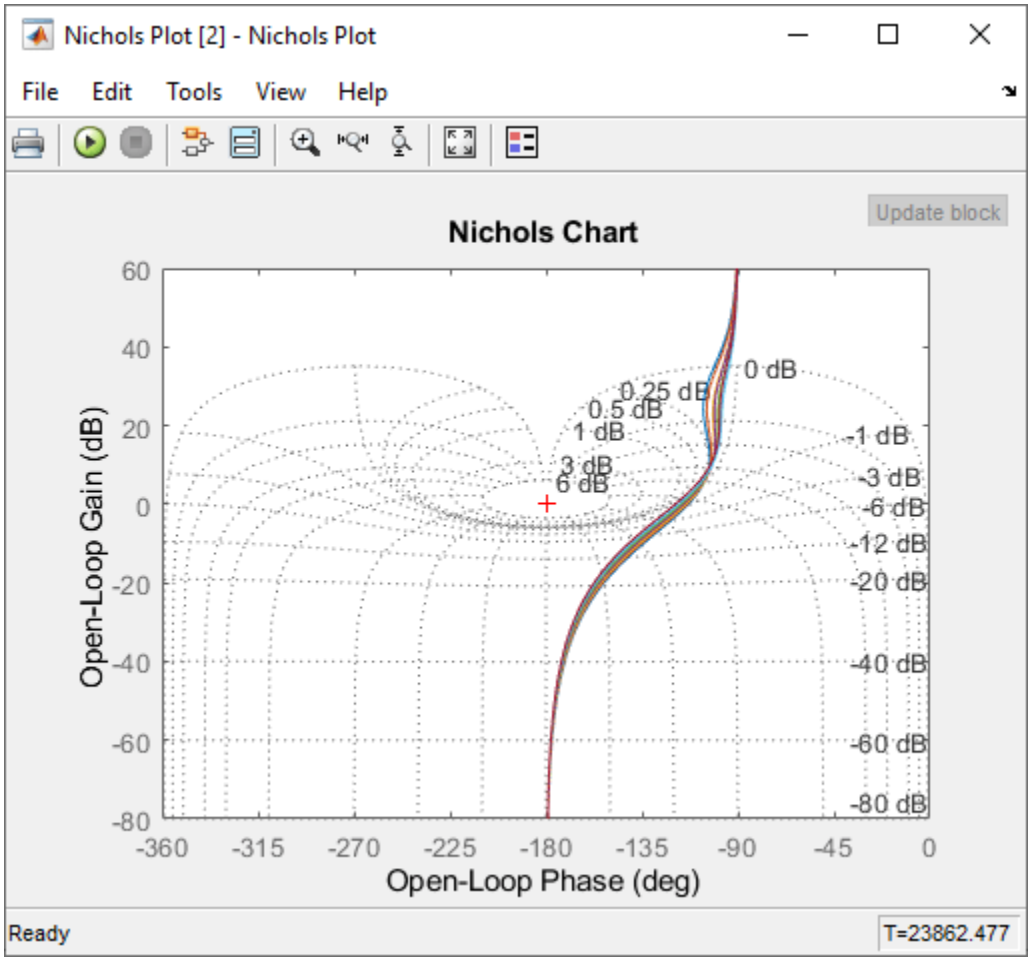
To check if the specified gain and phase margins are satisfied, simulate the model.

```
sim(mdl);
```

```
Warning: Assertion detected in 'scdaircraft/Check Gain and Phase Margins' at
time 27020.4
```



	Linear system computed at time	Gain Margin (dB)	Phase crossover ('rad/s')	Phase Margin (deg)	Gain crossover ('rad/s')
1	t=0	Inf	Inf	66.6032	0.15983
2	t=1803	Inf	Inf	66.7648	0.16265
3	t=3600	Inf	Inf	66.836	0.1665
4	t=5405	Inf	Inf	66.4677	0.171
5	t=7209	Inf	Inf	66.0927	0.17575
6	t=9005	Inf	Inf	65.788	0.18109
7	t=1.08e+04	Inf	Inf	65.2984	0.18694
8	t=1.261e+04	Inf	Inf	64.722	0.19316
9	t=1.441e+04	Inf	Inf	64.1041	0.19931
10	t=1.62e+04	Inf	Inf	63.561	0.20571
11	t=1.801e+04	Inf	Inf	62.9799	0.21257
12	t=1.981e+04	Inf	Inf	62.3646	0.21984
13	t=2.16e+04	Inf	Inf	61.5929	0.22726
14	t=2.341e+04	Inf	Inf	60.826	0.23439
15	t=2.521e+04	Inf	Inf	60.0983	0.24109
16	t=2.701e+04	Inf	Inf	59.41	0.24736
17	t=2.88e+04	Inf	Inf	58.76	0.25313



The tabular display from the Gain and Phase Margin block shows the following information.

- Times when the control loop is linearized
- Corresponding computed gain and phase margins.

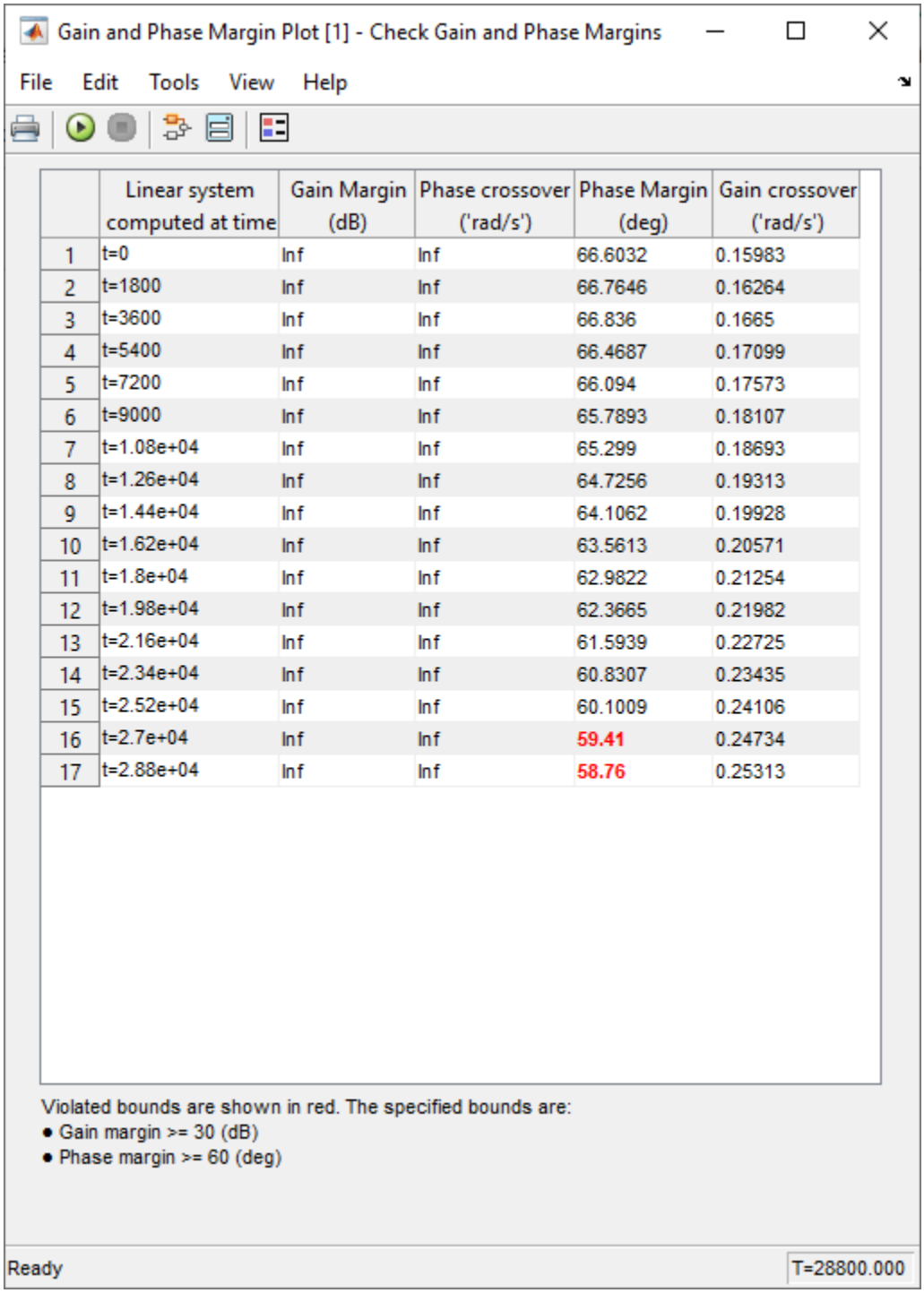
Margins that violate the specified bounds are shown in red. The phase margin bound is violated towards the end of the simulation when the fuel mass and airplane velocity have dropped. The Nichols plot indicates the small loop response variations as the fuel load and aircraft velocity change.

The table shows that the linearizations are not computed at exactly every 30 min but at small variations of 30 min. This is because zero-crossing detection for the block is not enabled. Enabling zero-crossing for the block ensures that the linearizations are computed at exactly 30 min intervals but may increase the time the simulation takes to run.

To enable zero-crossing detection, you can select the **Enable zero-crossing detection** option on the block **Linearizations** tab or use the following command.

```
set_param('scdaircraft/Check Gain and Phase Margins','ZeroCross','on')
sim(mdl);
```

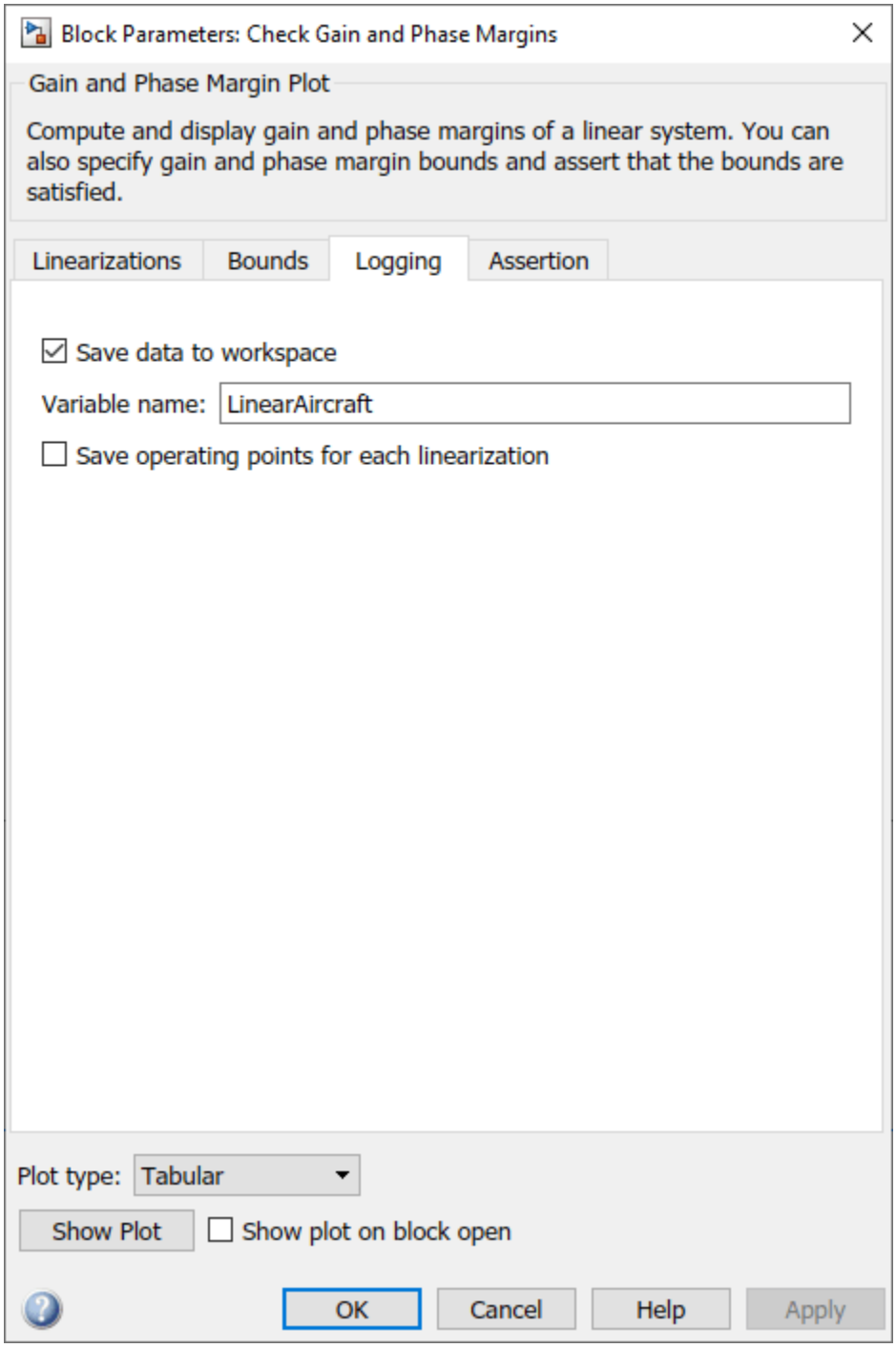
```
Warning: Assertion detected in 'scdaircraft/Check Gain and Phase Margins' at
time 27000
```



	Linear system computed at time	Gain Margin (dB)	Phase crossover ('rad/s')	Phase Margin (deg)	Gain crossover ('rad/s')
1	t=0	Inf	Inf	66.6032	0.15983
2	t=1800	Inf	Inf	66.7646	0.16264
3	t=3600	Inf	Inf	66.836	0.1665
4	t=5400	Inf	Inf	66.4687	0.17099
5	t=7200	Inf	Inf	66.094	0.17573
6	t=9000	Inf	Inf	65.7893	0.18107
7	t=1.08e+04	Inf	Inf	65.299	0.18693
8	t=1.26e+04	Inf	Inf	64.7256	0.19313
9	t=1.44e+04	Inf	Inf	64.1062	0.19928
10	t=1.62e+04	Inf	Inf	63.5613	0.20571
11	t=1.8e+04	Inf	Inf	62.9822	0.21254
12	t=1.98e+04	Inf	Inf	62.3665	0.21982
13	t=2.16e+04	Inf	Inf	61.5939	0.22725
14	t=2.34e+04	Inf	Inf	60.8307	0.23435
15	t=2.52e+04	Inf	Inf	60.1009	0.24106
16	t=2.7e+04	Inf	Inf	59.41	0.24734
17	t=2.88e+04	Inf	Inf	58.76	0.25313

Log the Linear Systems

On the **Logging** tab, you can configure Linear Analysis Plots and Model Verification to log the computed linear systems to the MATLAB® workspace.



For this model, the Check Gain and Phase Margins block is configured to save the lienar systems in the structure `LinearAircraft`. This structure contains the linear systems and corresponding simulation times in the `values` and `time` fields, respectively.

```
LinearAircraft
```

```
LinearAircraft =
```

```
  struct with fields:

              time: [17x1 double]
            values: [1x1x17x1 ss]
         blockName: 'scdaircraft/Check Gain and Phase Margins'
    assertionValue: [17x1 logical]
```

The `values` field stores the linear systems as an array of LTI state-space systems. For more information, see “Model Arrays”.

You can retrieve the individual systems by indexing into the `values` field.

```
L = LinearAircraft.values(:,:,17)
```

```
L =

  A =
                 scdaircraft/  Continuous/I        Filter
   scdaircraft/       -0.01122             0             0
   Continuous/I       -0.01184             0             0
   Filter               0.7492             0       -0.4326

  B =
                 Velocity Con
   scdaircraft/         0.3774
   Continuous/I              0
   Filter                    0

  C =
                 scdaircraft/  Continuous/I        Filter
   Velocity Con     -1.998e-15             1       -0.4326

  D =
                 Velocity Con
   Velocity Con              0

Continuous-time state-space model.
```

Close the model.

```
bdclose('scdaircraft')
```

See Also

More About

- “Monitor Linear System Characteristics in Simulink Models” on page 17-2
- “Verifiable Linear System Characteristics” on page 17-4

Functions

addoutputspec

Add output specification to operating point specification

Syntax

```
newOpspec = addoutputspec(opspec,block,port)
```

Description

`newOpspec = addoutputspec(opspec,block,port)` adds an output specification for a Simulink model to an existing operating point specification or array of operating point specifications. The output specification is added for the signal that originates from the specified output port `port` of block `block`.

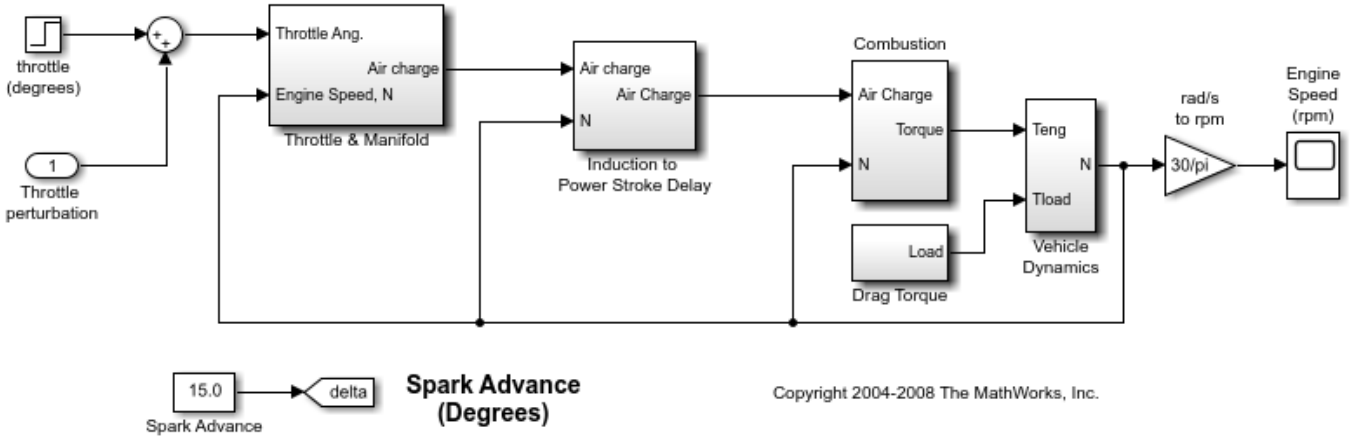
To find the width of the specified port, the `addoutputspec` command recompiles the model.

Examples

Add Output Specification to Operating Point Specification Object

Open the Simulink model.

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



Create a default operating point specification object for the model.

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

```
opspec =
```

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

States:
----------
      <strong>x</strong>       <strong>Known</strong>    <strong>SteadyState</strong>    <strong>
    <strong>______</strong>    <strong>_____</strong>    <strong>___________</strong>    <strong>

(1.) scdspeed/Throttle & Manifold/Intake Manifold/p0 = 0.543 bar
     0.543    false       true        -Inf    Inf    -Inf      Inf
(2.) scdspeed/Vehicle Dynamics/w = T//J w0 = 209 rad//s
    209.48    false       true        -Inf    Inf    -Inf      Inf

Inputs:
----------
    <strong>u</strong>    <strong>Known</strong>    <strong>Min</strong>     <strong>Max</strong>
    <strong>_</strong>    <strong>_____</strong>    <strong>____</strong>    <strong>___</strong>

(1.) scdspeed/Throttle  perturbation
    0    false    -Inf    Inf

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

The default operating point specification object has no output specifications because there are no root-level outports in the model.

Add an output specification to the outport of the rad/s to rpm block.

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

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

```
newspec.Outputs(1).Known = 1;
newspec.Outputs(1).y = 2000;
```

View the updated operating point specification.

```
newspec
```

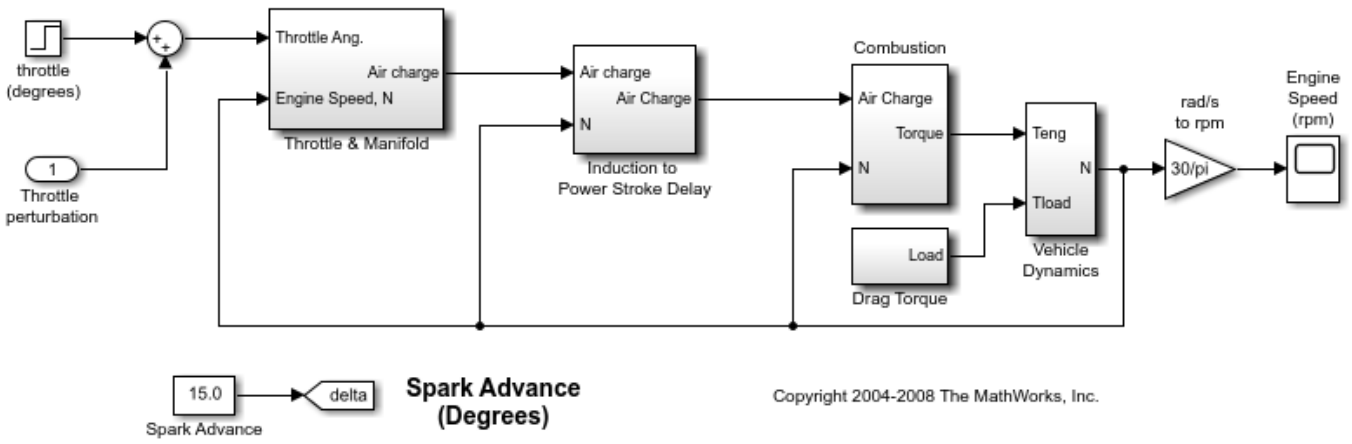
```
newspec =


 Operating point specification for the Model scdspeed.
 (Time-Varying Components Evaluated at time t=0)

States:
----------
      <strong>x</strong>       <strong>Known</strong>    <strong>SteadyState</strong>    <strong>
    <strong>______</strong>    <strong>_____</strong>    <strong>___________</strong>    <strong>

(1.) scdspeed/Throttle & Manifold/Intake Manifold/p0 = 0.543 bar
     0.543    false       true        -Inf    Inf    -Inf      Inf
(2.) scdspeed/Vehicle Dynamics/w = T//J w0 = 209 rad//s
    209.48    false       true        -Inf    Inf    -Inf      Inf

Inputs:
```

```
----------
    <strong>u</strong>    <strong>Known</strong>    <strong>Min</strong>     <strong>Max</strong>
    <strong>_</strong>    <strong>_____</strong>    <strong>____</strong>    <strong>___</strong>

(1.) scdspeed/Throttle  perturbation
    0    false    -Inf    Inf

Outputs:
----------
     <strong>y</strong>      <strong>Known</strong>    <strong>Min</strong>     <strong>Max</strong>
    <strong>____</strong>    <strong>_____</strong>    <strong>____</strong>    <strong>___</strong>

(1.) scdspeed/rad//s to rpm
    2000    true     -Inf    Inf
```

Add Output Specification to Multiple Operating Point Specification Objects

Open the Simulink model.

```
sys = 'scdspeed';
open_system(sys)
```



Create a 3-by-1 array of default operating point specification objects for the model.

```
opspec = operspec(sys,[3,1])
```

```
opspec =

Array of operating point specifications for the model scdspeed. To display an
 operating point specification, select an element from the array.
```

Add an output specification to the outport of the rad/s to rpm block.

```
newspec = addoutputspec(opspec,'scdspeed/rad//s to rpm',1);
```

This output specification is added to all of the operating point specification objects in `opspec`.

You can specify different output constraints for each specification in `opspec`. For example, specify different known values for each specification.

```
newspec(1,1).Outputs(1).Known = 1;
newspec(1,1).Outputs(1).y = 1900;

newspec(2,1).Outputs(1).Known = 1;
newspec(2,1).Outputs(1).y = 2000;

newspec(3,1).Outputs(1).Known = 1;
newspec(3,1).Outputs(1).y = 2100;
```

Input Arguments

`opspec` — Operating point specification

`OperatingSpec` object | array of `OperatingSpec` objects

Operating point specification for a Simulink model, specified as one of the following:

- `OperatingSpec` object — Add output specification to a single `OperatingSpec` object.
- Array of `OperatingSpec` objects — Add the same output specification to all `OperatingSpec` objects in the array. All the specification objects must have the same `Model` property.

To create an `OperatingSpec` object for your model, use the `operspec` function.

`block` — Simulink block

character vector | string

Simulink block to which to add the output specification, specified as a character vector or string that contains its block path. The `block` must be in the Simulink model specified in `opspec.Model`.

`port` — Output port

positive integer

Output port to which to add the output specification, specified as a positive integer in the range `[1,N]`, where `N` is the number of output ports on the specified `block`.

Output Arguments

`newOpspec` — Updated operating point specification

`OperatingSpec` object | array of `OperatingSpec` objects

Updated operating point specification, returned as an `OperatingSpec` object or an array of `OperatingSpec` objects with the same dimensions as `opspec`. `newOpspec` is the same as `opspec`, except that it contains the new output specification in its `Outputs` array.

You can modify the constraints and specifications for the new output specification using dot notation.

Alternative Functionality

Steady State Manager

You can interactively add output specifications when trimming your model using the **Steady State Manager**. For more information, see “Compute Operating Points from Specifications Using Steady State Manager” on page 1-20.

Simulink Model

You can add output specifications directly in your Simulink model. To do so, right-click the signal to which you want to add the specification, and select **Linear Analysis Points > Trim Output Constraint**.

See Also

`findop` | `operspec` | `operpoint`

Introduced before R2006a

advise

Package: `linearize.advisor`

Find blocks that are potentially problematic for linearization

Syntax

```
advise(advisor)
result = advise(advisor)
```

Description

When you linearize a Simulink model, you can create a `LinearizationAdvisor` object that contains diagnostic information about individual block linearizations. To search the `LinearizationAdvisor` object for diagnostics of blocks that are potentially problematic for linearization, use the `advise` function.

`advise(advisor)` opens the **Model Linearizer** with an **Advisor** tab open for troubleshooting the block linearizations in `advisor`. For more information, see “Troubleshoot Linearization Results in Model Linearizer” on page 4-16.

`result = advise(advisor)` returns a `LinearizationAdvisor` object that contains linearization diagnostic information for any blocks in `advisor` that are potentially problematic for linearization.

Examples

Open Linearization Advisor

Load Simulink model.

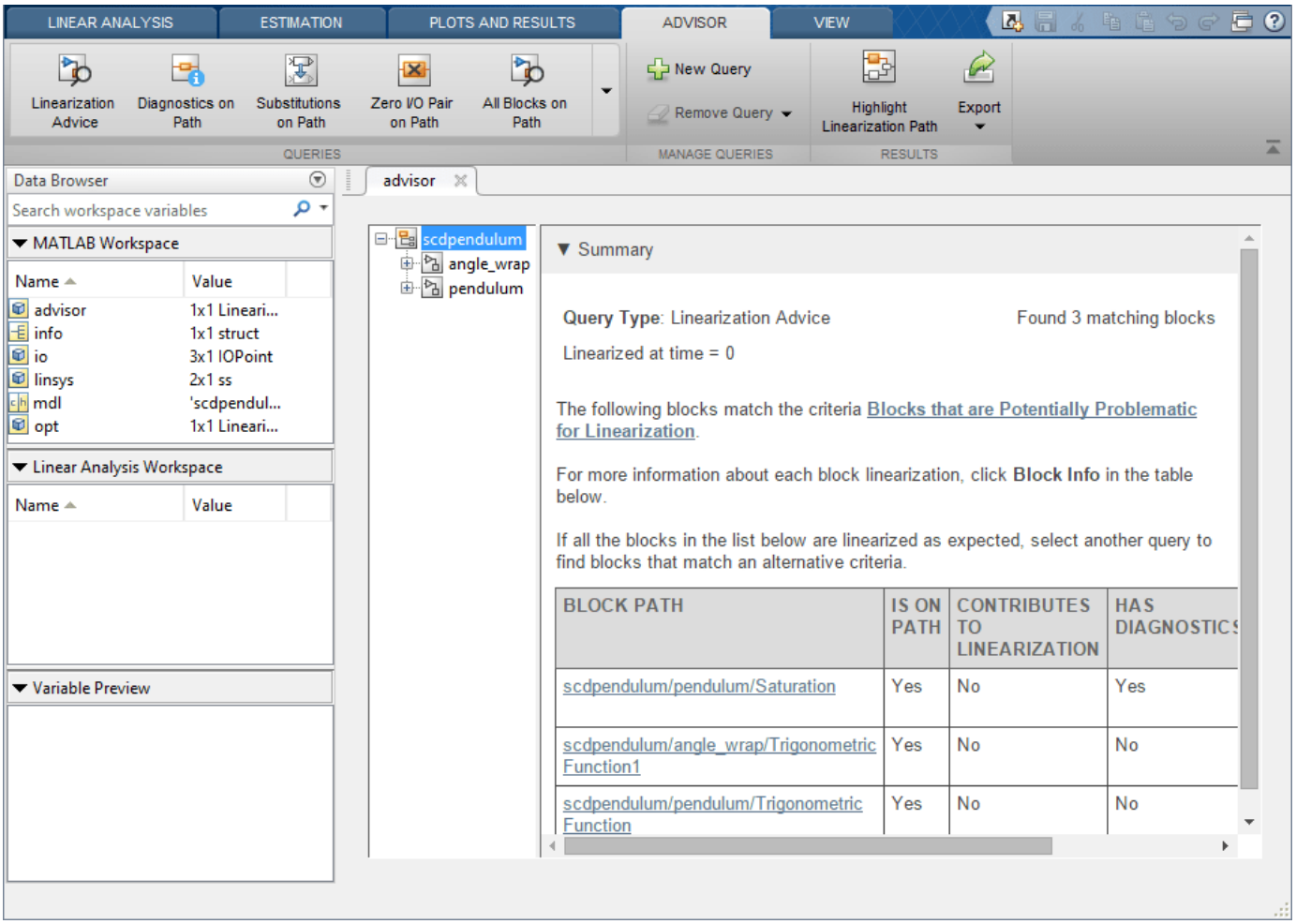
```
mdl = 'scdpendulum';
load_system(mdl)
```

Linearize model, and obtain `LinearizationAdvisor` object.

```
io = getlinio(mdl);
opt = linearizeOptions('StoreAdvisor',true);
[linsys,~,info] = linearize(mdl,io,opt);
advisor = info.Advisor;
```

Open the Linearization Advisor in the **Model Linearizer**.

```
advise(advisor)
```



Find Potentially Problematic Blocks for Linearization

Load Simulink model.

```
mdl = 'scdpendulum';
load_system(mdl)
```

Linearize model and obtain `LinearizationAdvisor` object.

```
io = getlinio(mdl);
opt = linearizeOptions('StoreAdvisor',true);
[linsys,~,info] = linearize(mdl,io,opt);
advisor = info.Advisor;
```

Find potentially problematic blocks for linearization.

```
result = advise(advisor)
```

```
result =
  LinearizationAdvisor with properties:
```

```
            Model: 'scdpendulum'
  OperatingPoint: [1x1 opcond.OperatingPoint]
BlockDiagnostics: [1x3 linearize.advisor.BlockDiagnostic]
       QueryType: 'Linearization Advice'
```

Input Arguments

`advisor` — Diagnostic information for block linearizations

`LinearizationAdvisor` object | array of `LinearizationAdvisor` objects

Diagnostic information for block linearizations, specified as a `LinearizationAdvisor` object or an array of `LinearizationAdvisor` objects.

Output Arguments

`result` — Diagnostic information for potentially problematic blocks

`LinearizationAdvisor` object

Diagnostic information for potentially problematic blocks in linearization results, returned as a `LinearizationAdvisor` object. `result` contains linearization diagnostic information for any blocks in `advisor` that are on the linearization path and satisfy at least one of the following criteria:

- Have diagnostic messages regarding the block linearization
- Linearize to zero
- Have substituted linearizations

Algorithms

Calling the `advise` function is equivalent to performing the following custom query with the `find` function:

```
qPath     = linqueryIsOnPath;
qZero     = linqueryIsZero;
qBlkRep   = linqueryIsBlockSubstituted;
qDiags    = linqueryHasDiagnostics;

q = qPath & (qZero | qDiags | qBlkRep);

advisor_new = find(advisor,q);
```

See Also

Apps
Model Linearizer

Functions
`find` | `getBlockInfo` | `getBlockPaths` | `highlight`

Objects
`LinearizationAdvisor`

Topics
“Troubleshoot Linearization Results at Command Line” on page 4-28
“Identify and Fix Common Linearization Issues” on page 4-6

Introduced in R2017b

copy

Copy operating point or operating point specification

Syntax

```
op_point2 = copy(op_point1)
op_spec2 = copy(op_spec1)
```

Description

`op_point2 = copy(op_point1)` returns a copy of the operating point object `op_point1`. You can create `op_point1` with the function `operpoint`.

`op_spec2 = copy(op_spec1)` returns a copy of the operating point specification object `op_spec1`. You can create `op_spec1` with the function `operspec`.

Note The command `op_point2 = op_point1` does not create a copy of `op_point1` but instead creates a pointer to `op_point1`. In this case, any changes made to `op_point2` are also made to `op_point1`. The same is true for operating point specifications. For an example, see “Copy an Operating-Point Specification” on page 18-185.

Examples

Copy an Operating-Point Specification

You can create new `operspec` variables in three ways:

- Using the `operspec` command
- Using assignment with the equals (=) operator
- Using the `copy` command

Using the = operator results in linked variables that both point to the same underlying data. Using the `copy` command results in an independent `operspec` object. In this example, create `operspec` objects both ways, and examine their behavior.

```
mdl = 'watertank';
open_system(mdl)
opspec1 = operspec(mdl)
```

```
opspec1 =
 Operating point specification for the Model watertank.
 (Time-Varying Components Evaluated at time t=0)

States:
----------
    x    Known    SteadyState    Min     Max    dxMin    dxMax
    _    _____    ___________    ____    ___    _____    _____
```

```
(1.) watertank/PID Controller/Integrator/Continuous/Integrator
    0    false       true        -Inf    Inf    -Inf      Inf
(2.) watertank/Water-Tank System/H
    1    false       true           0    Inf    -Inf      Inf

Inputs: None
----------

Outputs: None
----------
```

Create a new operating point specification object using assignment with the = operator.

```
opspec2 = opspec1;
```

`opspec2` is an `operspec` object that points to the same underlying data as `opspec1`. Because of this link, you cannot independently change properties of the two `operspec` objects. To see this, change a property of `opspec2`. For instance, change the initial value for the first state from 0 to 2. The change shows in the `States` section of the display.

```
opspec2.States(1).x = 2
```

```
opspec2 =
 Operating point specification for the Model watertank.
 (Time-Varying Components Evaluated at time t=0)

States:
----------
    x    Known    SteadyState    Min     Max    dxMin    dxMax
    _    _____    ___________    ____    ___    _____    _____

(1.) watertank/PID Controller/Integrator/Continuous/Integrator
    2    false       true        -Inf    Inf    -Inf      Inf
(2.) watertank/Water-Tank System/H
    1    false       true           0    Inf    -Inf      Inf

Inputs: None
----------

Outputs: None
----------
```

Examine the display of `opspec1` to see that the corresponding property value of `opspec1` also changes from 0 to 2.

```
opspec1
```

```
opspec1 =
 Operating point specification for the Model watertank.
 (Time-Varying Components Evaluated at time t=0)

States:
----------
    x    Known    SteadyState    Min     Max    dxMin    dxMax
    _    _____    ___________    ____    ___    _____    _____

(1.) watertank/PID Controller/Integrator/Continuous/Integrator
    2    false       true        -Inf    Inf    -Inf      Inf
```

```
(2.) watertank/Water-Tank System/H
    1    false       true           0    Inf    -Inf      Inf

Inputs: None
----------

Outputs: None
----------
```

To create an independent copy of an operating point specification, use the `copy` command.

```
opspec3 = copy(opspec1);
```

Now, when you change a property of `opspec3`, `opspec1` does not change. For instance, change the initial value for the first state from 2 to 4.

```
opspec3.States(1).x = 4
```

```
opspec3 =
 Operating point specification for the Model watertank.
 (Time-Varying Components Evaluated at time t=0)

States:
----------
    x    Known    SteadyState    Min     Max    dxMin    dxMax
    _    _____    ___________    ____    ___    _____    _____

(1.) watertank/PID Controller/Integrator/Continuous/Integrator
    4    false       true        -Inf    Inf    -Inf      Inf
(2.) watertank/Water-Tank System/H
    1    false       true           0    Inf    -Inf      Inf

Inputs: None
----------

Outputs: None
----------
```

In `opspec1`, the corresponding value remains 2.

```
opspec1.States(1).x
```

```
ans = 2
```

This copy behavior occurs because `operspec` is a *handle object*. For more information about handle objects, see “Handle Object Behavior”.

Copy an Operating Point

You can create new operating-point variables in three ways:

- Using the `operpoint` function
- Using assignment with the equals (=) operator
- Using the `copy` function

Using the = operator results in linked variables that both point to the same underlying data. Using the `copy` function results in an independent operating-point object. In this example, create operating-point objects both ways, and examine their behavior.

```
mdl = 'watertank';
open_system(mdl)
op1 = operpoint(mdl)
```

```
op1 =
 Operating point for the Model watertank.
 (Time-Varying Components Evaluated at time t=0)

States:
----------
    x
    -

(1.) watertank/PID Controller/Integrator/Continuous/Integrator
    0
(2.) watertank/Water-Tank System/H
    1

Inputs: None
----------
```

Create a new operating-point object using assignment with the = operator.

```
op2 = op1;
```

`op2` is an operating-point object that points to the same underlying data as `op1`. Because of this link, you cannot independently change properties of the two operating-point objects. To see this, change a property of `op2`. For instance, change the value for the first state from 0 to 2. The change shows in the `States` section of the display.

```
op2.States(1).x = 2
```

```
op2 =
 Operating point for the Model watertank.
 (Time-Varying Components Evaluated at time t=0)

States:
----------
    x
    -

(1.) watertank/PID Controller/Integrator/Continuous/Integrator
    2
(2.) watertank/Water-Tank System/H
    1

Inputs: None
----------
```

Examine the display of `op1` to see that the corresponding property value of `op1` also changes from 0 to 2.

```
op1
```

```
op1 =
 Operating point for the Model watertank.
 (Time-Varying Components Evaluated at time t=0)

States:
----------
    x
    -

(1.) watertank/PID Controller/Integrator/Continuous/Integrator
    2
(2.) watertank/Water-Tank System/H
    1

Inputs: None
----------
```

To create an independent copy of an operating-point object, use the `copy` function.

```
op3 = copy(op1);
```

Now, when you change a property of `op3`, `op1` does not change. For instance, change the value for the first state from 2 to 4.

```
op3.States(1).x = 4
```

```
op3 =
 Operating point for the Model watertank.
 (Time-Varying Components Evaluated at time t=0)

States:
----------
    x
    -

(1.) watertank/PID Controller/Integrator/Continuous/Integrator
    4
(2.) watertank/Water-Tank System/H
    1

Inputs: None
----------
```

In `op1`, the corresponding value remains 2.

```
op1.States(1).x
```

```
ans = 2
```

This copy behavior occurs because the operating-point object is a *handle object*. For more information about handle objects, see “Handle Object Behavior”.

See Also

`operpoint` | `operspec`

Introduced before R2006a

fastRestartForLinearAnalysis

Fast restart for linear analysis

Syntax

```
fastRestartForLinearAnalysis(model,'on')
fastRestartForLinearAnalysis(model,'on',Name,Value)
fastRestartForLinearAnalysis(model,'off')
```

Description

`fastRestartForLinearAnalysis(model,'on')` prepares the `model` for single compilation workflows by turning fast restart for linear analysis `'on'`. Once a compiling function is called, the model will remain compiled after the function is finished executing. Compiling functions can then be later called without any additional compilations. If the linear analysis points or block substitutions change with subsequent calls to compiling functions, the model is recompiled.

`fastRestartForLinearAnalysis(model,'on',Name,Value)` prepares the `model` for single compilation workflows with additional options specified by one or more `Name,Value` pair arguments.

`fastRestartForLinearAnalysis(model,'off')` turns fast restart for linear analysis off and restores the model parameters to their original value. Simulink does not let you close the `model` while it is in a complied state. Use this syntax to turn fast restart for linear analysis off before closing the `model`.

You can also click the link that appears on the top of the compiled Simulink model to turn `fastRestartForLinearAnalysis` off. For more information, see “Tips” on page 18-21.

Examples

Linear Analysis in a Loop with Fast Restart

Trim and linearize a closed-loop engine speed control model. Use fast restart for linear analysis to reduce the number of model compilations when looping over parameters.

Open the engine speed control model and get the analysis points for linearization. Doing so prevents recompilation between the first call to `findop` and `linearize`.

```
model = 'scdspeedctrl';
open_system(model)
io = getlinio(model);
fopt = findopOptions('DisplayReport','off');
```

Configure the PI controller to use the base workspace variables `kp` and `ki`.

```
block = [model,'/PID Controller'];
set_param(block,'P','kp');
set_param(block,'I','ki');
```

Create a grid of parameters to vary.

```
vp = 0.0005:0.0005:0.003;
vi = 0.0025:0.0005:0.005;
[KP,KI] = ndgrid(vp,vi);
N = numel(KP);
sz = size(KP);
```

Initialize the base workspace variables kp and `ki`.

```
kp = KP(1);
ki = KI(1);
```

Turn `fastRestartForLinearAnalysis` on and specify the analysis points using `io`.

```
fastRestartForLinearAnalysis(model,'on','AnalysisPoints',io)
```

Perform the linear analysis in a loop. When fast restart for linear analysis is on, calling `findop` sends the updated controller parameters to the model.

```
ops = operspec(model); % operating point specifications
for i = N:-1:1
    kp = KP(i);
    ki = KI(i);
    op = findop(model,ops,fopt); % trim the model
    [j,k] = ind2sub(sz,i);
    sysFastRestartLoop(:,:,j,k) = linearize(model,io,op); % linearize the model
end
```

Turn off `fastRestartForLinearAnalysis` and close the model.

```
fastRestartForLinearAnalysis(model,'off')
bdclose(model)
```

Linearize Model Using Multiple State Initial Conditions

When fast restart for linear analysis is on, calling compiling functions does not automatically apply changes to state initial conditions. Therefore, you must configure the initial state using an operating point object rather than using parameters or workspace variables.

Open the model and create linear analysis points.

```
mdl = 'magball';
open_system(mdl)
io(1) = linio([mdl '/Controller'],1,'input');
io(2) = linio([mdl '/Magnetic Ball Plant'],1,'openoutput');
```

Configure the ball height to use the `hInitial` workspace variable as the initial condition.

```
set_param([mdl '/Magnetic Ball Plant/height'],'InitialCondition','hInitial')
```

Turn on fast restart for linear analysis.

```
fastRestartForLinearAnalysis(mdl,'on')
```

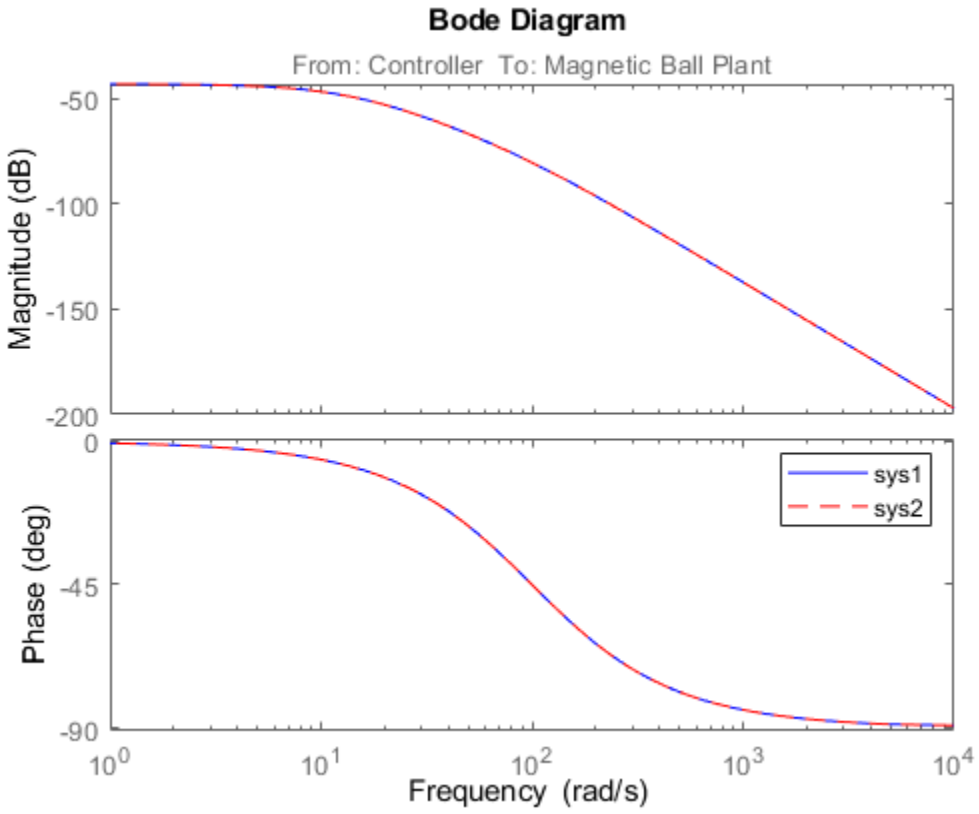
Linearize the model using different initial height values.

```
hInitial = 0.05;
sys1 = linearize(mdl,io);
```

```
hInitial = 0.1;
sys2 = linearize(mdl,io);
```

The frequency responses of the linearized models are the same. Therefore, the initial condition does not update when `hInitial` changes.

```
bode(sys1,'b',sys2,'r--')
legend('sys1','sys2')
```



To vary the initial state when fast restart is enabled, you must modify an operating point object instead. Create an operating point based on the model initial conditions.

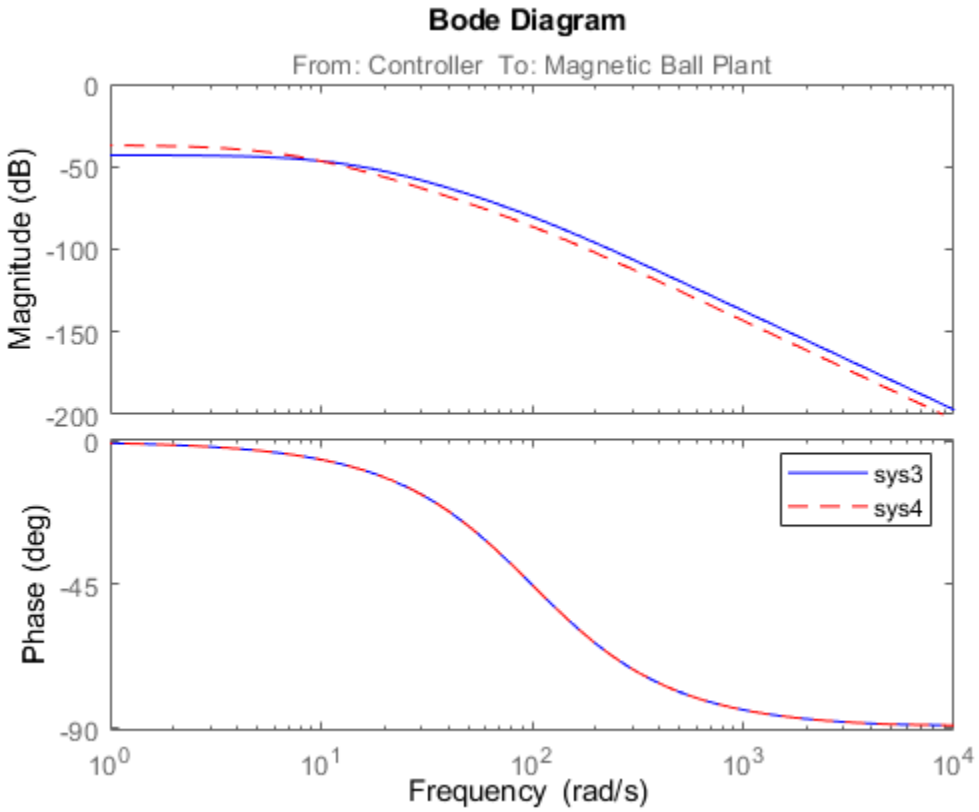
```
op = operpoint(mdl);
```

Modify the operating point for different initial conditions and linearize the model.

```
op.States(5).x = 0.05;
sys3 = linearize(mdl,io,op);
op.States(5).x = 0.1;
sys4 = linearize(mdl,io,op);
```

The frequency responses of the linearized models are different. Therefore, the initial condition changes when the operating point object changes.

```
bode(sys3,'b',sys4,'r--')
legend('sys3','sys4')
```



Turn off fast restart for linear analysis and close the model.

```
fastRestartForLinearAnalysis(mdl,'off')
bdclose(mdl)
```

Input Arguments

`model` — Simulink model name

character vector | string | `slTuner` object | `slLinearizer` object

Simulink model name, specified as a character vector, string, `slTuner` object, or `slLinearizer` object. The model must be in the current working folder or on the MATLAB path.

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 quotes. You can specify several name and value pair arguments in any order as `Name1,Value1,...,NameN,ValueN`.

Example: `...,'UseBusSignalLabels','on'`

`AnalysisPoints` — Analysis point set

[] (default) | linearization I/O object | vector of linearization I/O objects

Analysis point set that contains inputs, outputs, and openings, specified as the comma-separated pair consisting of `AnalysisPoints` and a linearization I/O object or a vector of linearization I/O objects. To create `AnalysisPoints`:

- Define the inputs, outputs, and openings using `linio`.
- If the inputs, outputs, and openings are specified in the Simulink model, extract these points from the model using `getlinio`.

Each linearization I/O object in `AnalysisPoints` must correspond to the Simulink model `model` or some normal mode model reference in the model hierarchy.

If the linear analysis points change with subsequent calls to compiling functions, the model is recompiled.

For more information on specifying linearization inputs, outputs, and openings, see “Specify Portion of Model to Linearize” on page 2-10.

`BlockSubstitutions` — Substitute linearizations for blocks and model subsystems
[] (default) | structure array | string array | character vector | cell array of character vectors

Substitute linearizations for blocks and model subsystems, specified as the comma-separated pair consisting of `BlockSubstitutions` and one of the following:

- A structure array
- A string array
- A character vector
- A cell array of character vectors

Use `BlockSubstitutions` to specify a custom linearization for a block or subsystem. For example, you can specify linearizations for blocks that do not have analytic linearizations, such as blocks with discontinuities or triggered subsystems.

If the block substitutions change with subsequent calls to compiling functions, the model is recompiled.

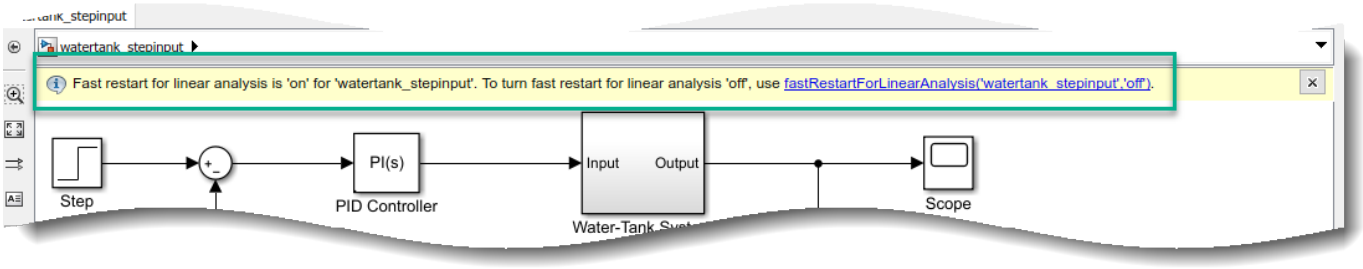
`UseBusSignalLabels` — Flag indicating whether to use bus signal channel numbers or names
`'off'` (default) | `'on'`

Flag indicating whether to use bus signal channel numbers or names to label the I/Os in the linearized model, specified as the comma-separated pair consisting of `UseBusSignalLabels` and one of the following:

- `'off'` — Use bus signal channel numbers to label I/Os on bus signals in the linearized model.
- `'on'` — Use bus signal names to label I/Os on bus signals in the linearized model. Bus signal names appear in the results when the I/O points are 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

Tips

- Simulink does not let you close the `model` while it is in a compiled state. Turn fast restart for linear analysis off to close the `model`. You can turn off `fastRestartForLinearAnalysis` in one of the following ways.
  - Use the syntax `fastRestartForLinearAnalysis(model,'off')`.
  - Click the link that appears on the top of the Simulink model.



- When fast restart for linear analysis is on, calling compiling functions does not automatically apply changes to state initial conditions. Therefore, you must configure the initial state using an operating point object rather than using parameters or workspace variables. For more information, see “Linearize Model Using Multiple State Initial Conditions” on page 18-17.

See Also

`slLinearizer` | `slTuner` | `linearize` | `findop` | `getlinio` | `operspec`

Topics

“Improve Linear Analysis Performance” on page 3-97

Introduced in R2019a

find

Package: `linearize.advisor`

Find blocks in linearization results that match specific criteria

Syntax

```
result = find(advisor,query)
```

Description

When you linearize a Simulink model, you can create a `LinearizationAdvisor` object that contains diagnostic information about individual block linearizations. To find block linearizations that satisfy specific criteria, you can use the `find` function with custom query objects. Alternatively, you can analyze linearization diagnostics using the Linearization Advisor in the **Model Linearizer**. For more information on finding specific blocks in linearization results, see “Find Blocks in Linearization Results Matching Specific Criteria” on page 4-37.

`result = find(advisor,query)` returns the subset of block diagnostics in `advisor` that match the search criteria specified in `query`.

Examples

Find Blocks on Linearization Path

Load Simulink model.

```
mdl = 'scdspeed';
load_system(mdl)
```

Linearize the model and obtain `LinearizationAdvisor` object.

```
opts = linearizeOptions('StoreAdvisor',true);
io(1) = linio('scdspeed/throttle (degrees)',1,'input');
io(2) = linio('scdspeed/rad//s to rpm',1,'output');
[sys,~,info] = linearize(mdl,io,opts);
advisor = info.Advisor;
```

Create a query object for finding blocks on the linearization path.

```
query = linqueryIsOnPath;
```

Find blocks using query object.

```
advOnPath = find(advisor,query)
```

```
advOnPath =
  LinearizationAdvisor with properties:

               Model: 'scdspeed'
      OperatingPoint: [1x1 opcond.OperatingPoint]
```

```
    BlockDiagnostics: [1x26 linearize.advisor.BlockDiagnostic]
           QueryType: 'On Linearization Path'
```

Find All SISO Blocks

Load the Simulink model.

```
mdl = 'scdspeed';
load_system(mdl)
```

Linearize the model and obtain the `LinearizationAdvisor` object.

```
opts = linearizeOptions('StoreAdvisor',true);
io(1) = linio('scdspeed/throttle (degrees)',1,'input');
io(2) = linio('scdspeed/rad//s to rpm',1,'output');
[sys,op,info] = linearize(mdl,io,opts);
advisor = info.Advisor;
```

Create compound query object for finding all blocks with one input and one output.

```
qSISO = linqueryHasInputs(1) & linqueryHasOutputs(1);
```

Find all SISO blocks using compound query object.

```
advSISO = find(advisor,qSISO)
```

```
advSISO =
  LinearizationAdvisor with properties:

               Model: 'scdspeed'
      OperatingPoint: [1x1 opcond.OperatingPoint]
    BlockDiagnostics: [1x10 linearize.advisor.BlockDiagnostic]
           QueryType: '(Has 1 Inputs & Has 1 Outputs)'
```

Input Arguments

`advisor` — Diagnostic information for block linearizations

`LinearizationAdvisor` object | array of `LinearizationAdvisor` objects

Diagnostic information for block linearizations, specified as a `LinearizationAdvisor` object or an array of `LinearizationAdvisor` objects.

`query` — Search criteria

`CompoundQuery` object | `linqueryIsOnPath` object | `linqueryHasDiagnostics` object | `linqueryHasOrder` object | ...

Search criteria, specified as one of the following query objects or a logical combination of query objects (`CompoundQuery` object).

Query Object	Find Blocks That...
`linqueryAdvise`	Are potentially problematic for linearization.
`linqueryAllBlocks`	Are in the `advisor` object.
`linqueryContributesToLinearization`	Numerically contribute to the model linearization result.
`linqueryHasDiagnostics`	Have diagnostic messages regarding their linearization.
`linqueryHasInputs`	Have a specified number of inputs.
`linqueryHasOrder`	Have a specified number of states.
`linqueryHasOutputs`	Have a specified number of outputs.
`linqueryHasSampleTime`	Have a specified sample time.
`linqueryHasZeroIOPair`	Have at least one input/output pair that linearizes to zero.
`linqueryIsBlockSubstituted`	Have a custom block linearization specified.
`linqueryIsBlockType`	Are of a specified type.
`linqueryIsExact`	Are linearized using their defined exact linearization.
`linqueryIsNumericallyPerturbed`	Are linearized using numerical perturbation.
`linqueryIsOnPath`	Are on the linearization path.
`linqueryIsZero`	Linearize to zero.

To create a compound query, combine these queries using AND (&), OR (|), and NOT (~) logical operations. For example, to find all blocks on the linearization path that do not contribute to the model linearization result, use:

```
compundQuery = linqueryIsOnPath & ~linqueryContributesToLinearization
```

Output Arguments

`result` — Diagnostic information for blocks that match the search criteria
`LinearizationAdvisor` object | array of `LinearizationAdvisor` objects

Diagnostic information for blocks that match the search criteria specified in `query`, returned as:

- `LinearizationAdvisor` object if `advisor` is a single `LinearizationAdvisor` object.
- A `LinearizationAdvisor` object with the same dimensions as `advisor` if `advisor` is an array.

See Also

Objects
`LinearizationAdvisor` | `CompoundQuery`

Functions
`advise` | `getBlockInfo` | `getBlockPaths` | `highlight`

Topics
“Troubleshoot Linearization Results at Command Line” on page 4-28

Introduced in R2017b

findop

Steady-state operating point from specifications (trimming) or simulation

Syntax

```
op = findop(mdl,opspec)
op = findop(mdl,opspec,param)

op = findop( ___ ,options)

[op,opreport] = findop( ___ )

op = findop(mdl,tsnapshot)
op = findop(mdl,tsnapshot,param)
```

Description

`op = findop(mdl,opspec)` returns the operating point of the model that meets the specifications in `opspec`. Typically, you trim the model at a steady-state operating point on page 18-37. The Simulink model must be open. If `opspec` is an array of operating points specifications, `findop` returns an array of corresponding operating points.

`op = findop(mdl,opspec,param)` batch trims the model for the parameter value variations specified in `param`.

`op = findop( ___ ,options)` trims the model using additional optimization algorithm `options`.

`[op,opreport] = findop( ___ )` returns an operating point search report, `opreport`, for any of the previous syntaxes.

`op = findop(mdl,tsnapshot)` simulates the model using the model initial conditions, and extracts operating points at simulation snapshot times specified in `tsnapshot`.

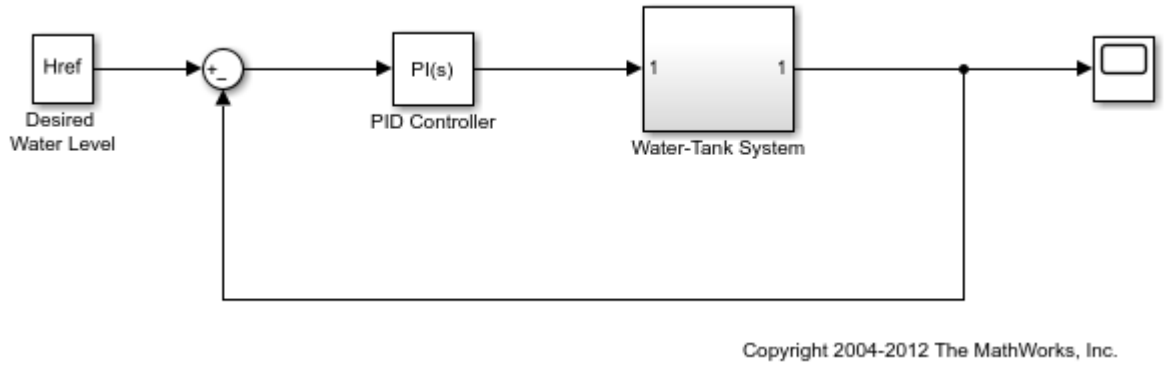
`op = findop(mdl,tsnapshot,param)` batch simulates the model using the parameter value variations specified in `param`.and extracts operating points at simulation snapshot times.

Examples

Trim Model to Meet State Specifications

Open the Simulink model.

```
mdl = 'watertank';
open_system(mdl)
```



Copyright 2004-2012 The MathWorks, Inc.

Trim the model to find a steady-state operating point where the water tank level is `10`.

Create default operating point specification object.

```
opspec = operspec(mdl);
```

Configure specifications for the first model state. The first state must be at steady state with a lower bound of `0`. Provide an initial guess of `2` for the state value.

```
opspec.States(1).SteadyState = 1;
opspec.States(1).x = 2;
opspec.States(1).Min = 0;
```

Configure the second model state as a known state with a value of `10`.

```
opspec.States(2).Known = 1;
opspec.States(2).x = 10;
```

Find the operating point that meets these specifications.

```
op = findop(mdl,opspec);
```

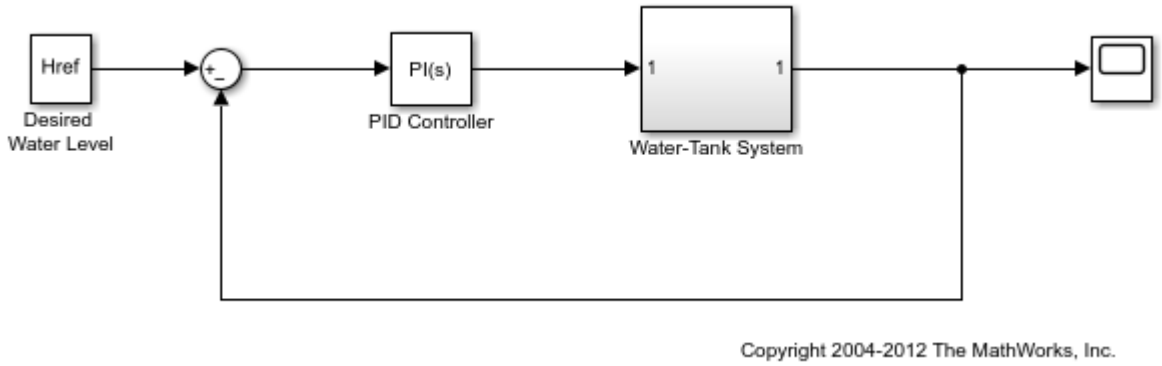
```
 Operating point search report:
---------------------------------

opreport =


 Operating point search report for the Model watertank.
 (Time-Varying Components Evaluated at time t=0)

Operating point specifications were successfully met.
States:
----------
    <strong>Min</strong>      <strong>x</strong>       <strong>Max</strong>    <strong>dxMin</st
    <strong>___</strong>    <strong>______</strong>    <strong>___</strong>    <strong>_____</st

(1.) watertank/PID Controller/Integrator/Continuous/Integrator
     0     1.2649    Inf      0      0       0
(2.) watertank/Water-Tank System/H
    10         10     10      0      0       0
```

```
Inputs: None
----------

Outputs: None
----------
```

Batch Trim Simulink Model for Parameter Variation

Open the Simulink model.

```
mdl = 'watertank';
open_system(mdl)
```



Vary parameters `A` and `b` within 10% of their nominal values, and create a 3-by-4 parameter grid.

```
[A_grid,b_grid] = ndgrid(linspace(0.9*A,1.1*A,3),...
                         linspace(0.9*b,1.1*b,4));
```

Create a parameter structure array, specifying the name and grid points for each parameter.

```
params(1).Name = 'A';
params(1).Value = A_grid;
params(2).Name = 'b';
params(2).Value = b_grid;
```

Create a default operating point specification for the model.

```
opspec = operspec(mdl);
```

Trim the model using the specified operating point specification and parameter grid.

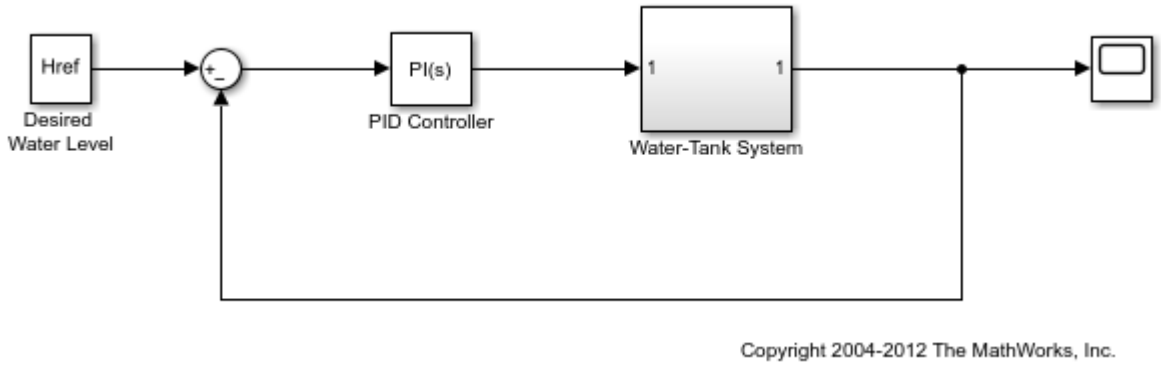
```
opt = findopOptions('DisplayReport','off');
op = findop(mdl,opspec,params,opt);
```

`op` is a 3-by-4 array of operating point objects that correspond to the specified parameter grid points.

Trim Model Using Specified Optimizer Type

Open the Simulink model.

```
mdl = 'watertank';
open_system(mdl)
```



Copyright 2004-2012 The MathWorks, Inc.

Create a default operating point specification object.

```
opspec = operspec(mdl);
```

Create an option set that sets the optimizer type to gradient descent and suppresses the search report display.

```
opt = findopOptions('OptimizerType','graddescent','DisplayReport','off');
```

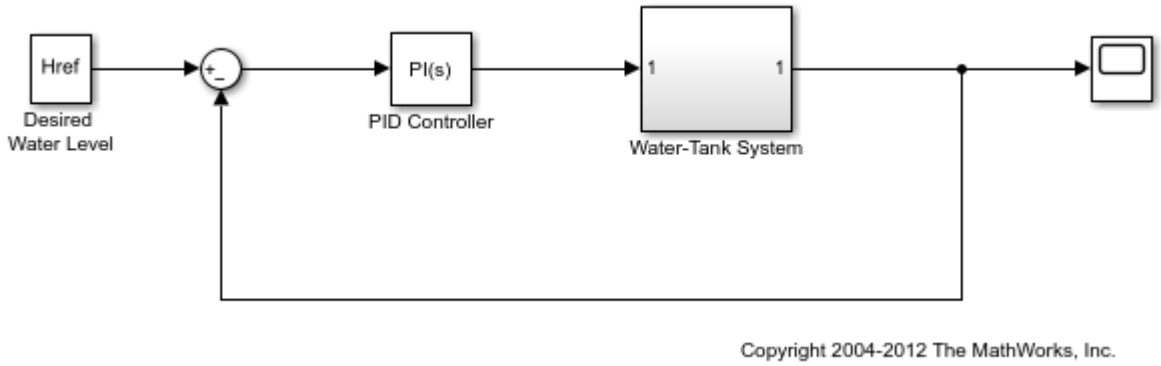
Trim the model using the specified option set.

```
op = findop(mdl,opspec,opt);
```

Obtain Operating Point Search Report

Open the Simulink model.

```
mdl = 'watertank';
open_system(mdl)
```



Copyright 2004-2012 The MathWorks, Inc.

Create default operating point specification object.

```
opspec = operspec(mdl);
```

Configure specifications for the first model state.

```
opspec.States(1).SteadyState = 1;
opspec.States(1).x = 2;
opspec.States(1).Min = 0;
```

Configure specifications for the second model state.

```
opspec.States(2).Known = 1;
opspec.States(2).x = 10;
```

Find the operating point that meets these specifications, and return the operating point search report. Create an option set to suppress the search report display.

```
opt = findopOptions('DisplayReport',false);
[op,opreport] = findop(mdl,opspec,opt);
```

`opreport` describes how closely the optimization algorithm met the specifications at the end of the operating point search.

```
opreport
```

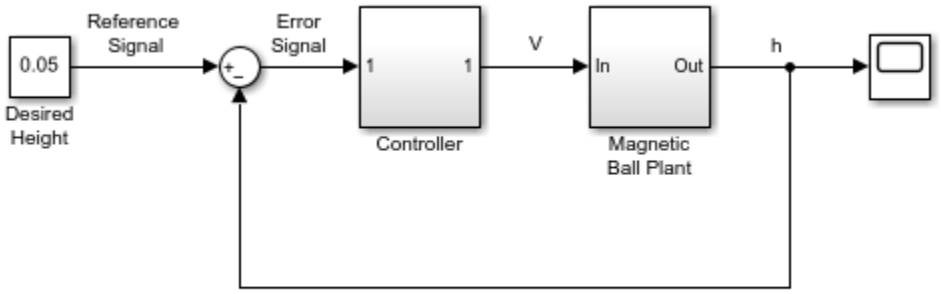
```
opreport =


 Operating point search report for the Model watertank.
 (Time-Varying Components Evaluated at time t=0)

Operating point specifications were successfully met.
States:
----------
    <strong>Min</strong>      <strong>x</strong>       <strong>Max</strong>    <strong>dxMin</str
    <strong>___</strong>    <strong>______</strong>    <strong>___</strong>    <strong>_____</str

(1.) watertank/PID Controller/Integrator/Continuous/Integrator
     0     1.2649    Inf      0      0       0
(2.) watertank/Water-Tank System/H
    10         10     10      0      0       0

Inputs: None
----------

Outputs: None
----------
```

`dx` is the time derivative for each state. Since all `dx` values are zero, the operating point is at steady state.

Extract Operating Points at Simulation Snapshots

Open the Simulink model.

```
mdl = 'magball';
open_system(mdl)
```



Copyright 2003-2006 The MathWorks, Inc.

Simulate the model, and extract operating points at `10` and `20` time units.

```
op = findop(mdl,[10,20]);
```

`op` is a column vector of operating points, with one element for each snapshot time.

Display the first operating point.

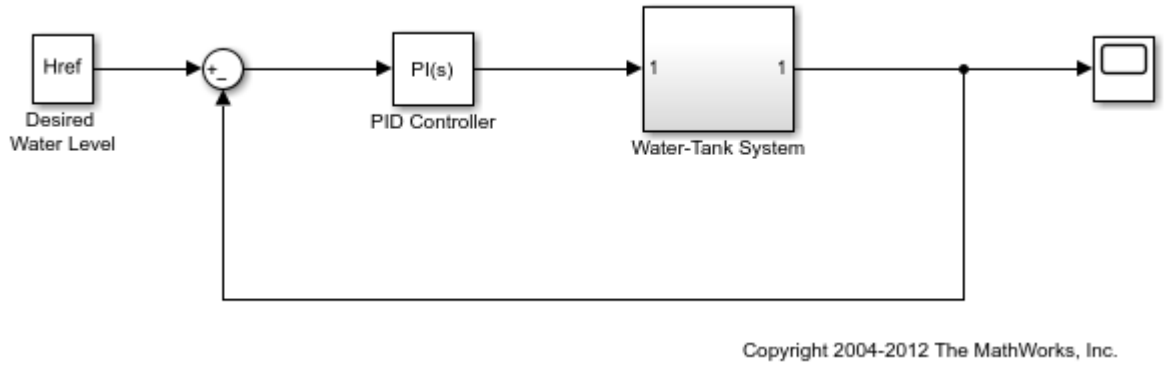
```
op(1)
```

```
ans =


 Operating point for the Model magball.
 (Time-Varying Components Evaluated at time t=10)

States:
----------
        <strong>x</strong>
    <strong>__________</strong>

(1.) magball/Controller/PID Controller/Filter/Cont. Filter/Filter
    5.4732e-07
(2.) magball/Controller/PID Controller/Integrator/Continuous/Integrator
        14.007
(3.) magball/Magnetic Ball Plant/Current
        7.0036
(4.) magball/Magnetic Ball Plant/dhdt
     8.443e-08
(5.) magball/Magnetic Ball Plant/height
          0.05

Inputs: None
----------
```

Vary Parameters and Extract Operating Points at Simulation Snapshots

Open Simulink model.

```
mdl = 'watertank';
open_system(mdl)
```



Copyright 2004-2012 The MathWorks, Inc.

Specify parameter values. The parameter grids are 5-by-4 arrays.

```
[A_grid,b_grid] = ndgrid(linspace(0.9*A,1.1*A,5),...
                         linspace(0.9*b,1.1*b,4));
params(1).Name = 'A';
params(1).Value = A_grid;
params(2).Name = 'b';
params(2).Value = b_grid;
```

Simulate the model and extract operating points at `0`, `5`, and `10` time units.

```
op = findop(mdl,[0 5 10],params);
```

`findop` simulates the model for each parameter value combination, and extracts operating points at the specified simulation times.

`op` is a 3-by-5-by-4 array of operating point objects.

```
size(op)


ans =

     3     5     4
```

Input Arguments

`mdl` — Simulink model name

character vector | string

Simulink model name, specified as a character vector or string. The model must be in the current working folder or on the MATLAB path.

`opspec` — Operating point specifications

`OperatingSpec` object | array of `OperatingSpec` objects

Operating point specifications for trimming the model, specified as an `OperatingSpec` object or an array of `OperatingSpec` objects created using the `operspec` function.

If `opspec` is an array, `findop` returns an array of corresponding operating points using a single model compilation.

`param` — Parameter samples

structure | structure array

Parameter samples for trimming, specified as one of the following:

- Structure — Vary the value of a single parameter by specifying `param` as a structure with the following fields:

  - `Name` — Parameter name, specified as a character vector or string. You can specify any model parameter that is a variable in the model workspace, the MATLAB workspace, or a data dictionary. If the variable used by the model is not a scalar variable, specify the parameter name as an expression that resolves to a numeric scalar value. For example, to use the first element of vector `V` as a parameter, use:

    ```
    param.Name = 'V(1)';
    ```

  - `Value` — Parameter sample values, specified as a double array.

  For example, vary the value of parameter `A` in the 10% range:

  ```
  param.Name = 'A';
  param.Value = linspace(0.9*A,1.1*A,3);
  ```

- Structure array — Vary the value of multiple parameters. For example, vary the values of parameters `A` and `b` in the 10% range:

  ```
  [A_grid,b_grid] = ndgrid(linspace(0.9*A,1.1*A,3),...
                           linspace(0.9*b,1.1*b,3));
  params(1).Name = 'A';
  params(1).Value = A_grid;
  params(2).Name = 'b';
  params(2).Value = b_grid;
  ```

When you specify parameter value variations, `findop` batch trims the model for each parameter value combination, and returns an array of corresponding operating points. If `param` specifies tunable parameters only, then the software batch trims the model using a single compilation.

If you specify `opspec` as a single `operspec` object and the parameter values in `param` produce states that conflict with known states in `opspec`, `findop` trims the model using the specifications in `opspec`. To trim the model at state values derived from the parameter values, specify `opspec` as an array of corresponding `operspec` objects. For an example, see “Batch Trim Simulink Model for Parameter Variation” on page 18-28.

`options` — Trimming options

`findopOptions` option set

Trimming options, specified as a `findopOptions` option set.

`tsnapshot` — Simulation snapshot times

scalar | vector

Simulation snapshot times at which to extract the operating point of the model, specified as a scalar for a single snapshot or a vector for multiple snapshots. `findop` simulates the model and computes an operating point for the state of the model at each snapshot time.

Output Arguments

op — Operating point

`OperatingPoint` object | array of `OperatingPoint` objects

Operating point, returned as an `OperatingPoint` object or an array of `OperatingPoint` objects. The dimensions of `op` depend on the specified parameter variations and either the operating-point specifications or the simulation snapshot time.

Parameter Variation	Find operating point for...	Resulting op Dimensions
No parameter variation	Single operating-point specification, specified by `opspec`	single operating-point object
	Single snapshot time, specified by `tsnapshot`	
	N_1-by-`...`-by-N_m array of operating-point specifications, specified by `opspec`	N_1-by-`...`-by-N_m
	N_s snapshots, specified by `tsnapshot`	Column vector of length N_s
N_1-by-`...`-by-N_m parameter grid, specified by `param`	Single operating-point specification, specified by `opspec`	N_1-by-`...`-by-N_m
	Single snapshot time, specified by `tsnapshot`	
	N_1-by-`...`-by-N_m array of operating-point specifications, specified by `opspec`	
	N_s snapshots, specified by `tsnapshot`	N_s-by-N_1-by-`...`-by-N_m.

For example, suppose:

- `opspec` is a single operating-point specification object and `param` specifies a 3-by-4-by-2 parameter grid. In this case, `op` is a 3-by-4-by-2 array of operating points.
- `tsnapshot` is a scalar and `param` specifies a 5-by-6 parameter grid. In this case, `op` is a 1-by-5-by-6 array of operating points.
- `tsnapshot` is a row vector with three elements and `param` specifies a 5-by-6 parameter grid. In this case, `op` is a 3-by-5-by-6 array of operating points.

Each operating-point object has the following properties:

Property	Description
`Model`	Simulink model name, returned as a character vector.

<table>
<tr><th>Property</th><th>Description</th></tr>
<tr><td>States</td><td>State operating point, returned as a vector of state objects. Each entry in States represents the supported states of one Simulink block.

For a list of supported states for operating point objects, see “Simulink Model States Included in Operating Point Object” on page 1-3.

Note If the block has multiple named continuous states, States contains one structure for each named state.

Each state object has the following fields:

<table>
<tr><th>Field</th><th>Description</th></tr>
<tr><td>Nx (read only)</td><td>Number of states in the block</td></tr>
<tr><td>Block</td><td>Block path, returned as a character vector.</td></tr>
<tr><td>StateName</td><td>State name</td></tr>
<tr><td>x</td><td>Values of all supported block states, returned as a vector of length Nx.</td></tr>
<tr><td>Ts</td><td>Sample time and offset of each supported block state, returned as a vector. For continuous-time systems, Ts is zero.</td></tr>
<tr><td>SampleType</td><td>State time rate, returned as one of the following:
• 'CSTATE' — Continuous-time state
• 'DSTATE' — Discrete-time state</td></tr>
<tr><td>inReferencedModel</td><td>Flag indicating whether the block is inside a reference model, returned as one of the following:
• 1 — Block is inside a reference model.
• 0 — Block is in the current model file.</td></tr>
<tr><td>Description</td><td>Block state description, returned as a character vector.</td></tr>
</table>
</td></tr>
<tr><td>Inputs</td><td>Input level at the operating point, returned as a vector of input objects. Each entry in Inputs represents the input levels of one root-level inport block in the model.

Each input object has the following fields:

<table>
<tr><th>Field</th><th>Description</th></tr>
<tr><td>Nu (read only)</td><td>Number of inport block signals</td></tr>
<tr><td>Block</td><td>Inport block name</td></tr>
<tr><td>PortDimensions</td><td>Dimension of signals accepted by the inport</td></tr>
<tr><td>u</td><td>Inport block input levels at the operating point, returned as a vector of length Nu.</td></tr>
<tr><td>Description</td><td>Inport block input description, returned as a character vector.</td></tr>
</table>
</td></tr>
</table>

Property	Description
`Time`	Times at which any time-varying functions in the model are evaluated, returned as a vector.
`Version`	Object version number

You can edit the properties of `op` using dot notation or the `set` function.

`opreport` — Operating point search report
`OperatingReport` object | array of `OperatingReport` objects

Operating point search report, returned as an `OperatingReport` object. If `op` is an array of `OperatingPoint` objects, then `opreport` is an array of corresponding `OperatingReport` objects.

This report displays automatically, even when you suppress the output using a semicolon. To hide the report, set the `DisplayReport` field in `options` to `'off'`.

Each operating point search report has the following properties:

Property	Description
`Model`	`Model` property value of `op`
`Inputs`	`Inputs` property value of `op`
`Outputs`	Output values at the computed operating point. This object contains the same fields as the `Outputs` property of `opspec`, with the addition of `yspec`, which is the desired output value.
`States`	`States` property value of `op` with the addition of `dx`, which contains the state derivative values. For discrete-time states, `dx` is the difference between the next state value and the current one; that is, $x(k+1) - x(k)$.
`Time`	`Time` property value of `op`
`TerminationString`	Optimization termination condition, returned as a character vector.

<table>
<tr><th>Property</th><th>Description</th></tr>
<tr><td><code>Optimization Output</code></td><td>Optimization algorithm search results, returned as a structure with the following fields:

<table>
<tr><th>Field</th><th>Description</th></tr>
<tr><td><code>iterations</code></td><td>Number of iterations performed during the optimization</td></tr>
<tr><td><code>funcCount</code></td><td>Number of function evaluations performed during the optimization</td></tr>
<tr><td><code>lssteplength</code></td><td>Size of line search step relative to search direction (active-set optimization algorithm only)</td></tr>
<tr><td><code>stepsize</code></td><td>Displacement in the state vector at the final iteration (active-set and interior-point optimization algorithms)</td></tr>
<tr><td><code>algorithm</code></td><td>Optimization algorithm used</td></tr>
<tr><td><code>firstorderopt</code></td><td>Measure of first-order optimization, for the trust-region-reflective optimization algorithm; <code>[]</code> for other algorithms</td></tr>
<tr><td><code>constrviolation</code></td><td>Maximum of constraint functions</td></tr>
<tr><td><code>message</code></td><td>Exit message</td></tr>
</table>
For more information about the optimization algorithm, see the Optimization Toolbox documentation.</td></tr>
</table>

More About

Steady-State Operating Point (Trim Condition)

A *steady-state operating point* of a model, also called an equilibrium or *trim* condition, includes state variables that do not change with time.

A model can have several steady-state operating points. For example, a hanging damped pendulum has two steady-state operating points at which the pendulum position does not change with time. A *stable steady-state operating point* occurs when a pendulum hangs straight down. When the pendulum position deviates slightly, the pendulum always returns to equilibrium. In other words, small changes in the operating point do not cause the system to leave the region of good approximation around the equilibrium value.

An *unstable steady-state operating point* occurs when a pendulum points upward. As long as the pendulum points *exactly* upward, it remains in equilibrium. However, when the pendulum deviates slightly from this position, it swings downward and the operating point leaves the region around the equilibrium value.

When using optimization search to compute operating points for nonlinear systems, your initial guesses for the states and input levels must be near the desired operating point to ensure convergence.

When linearizing a model with multiple steady-state operating points, it is important to have the right operating point. For example, linearizing a pendulum model around the stable steady-state operating point produces a stable linear model, whereas linearizing around the unstable steady-state operating point produces an unstable linear model.

Tips

- You can initialize an operating point search at a simulation snapshot or a previously computed operating point using `initopspec`.
- Linearize the model at the operating point `op` using `linearize`.

Algorithms

By default, `findop` uses the optimizer `graddescent-elim`. To use a different optimizer, change the value of `OptimizerType` in `options` using `findopOptions`.

`findop` automatically sets these Simulink model properties for optimization:

- `BufferReuse = 'off'`
- `RTWInlineParameters = 'on'`
- `BlockReductionOpt = 'off'`
- `SaveFormat = 'StructureWithTime'`

After the optimization completes, Simulink restores the original model properties.

Alternative Functionality

App

As an alternative to the `findop` command, you can find operating points in one of the following ways.

- Compute operating points using the **Steady State Manager**. For an example, see “Compute Operating Points from Specifications Using Steady State Manager” on page 1-20.
- If you are computing an operating point for linearization, you can find the operating point and linearize the model using the **Model Linearizer**. For an example, see “Compute Operating Points from Specifications Using Model Linearizer” on page 1-31.

Compatibility Considerations

`PortWidth` property of operating point inputs and outputs will be removed
Not recommended starting in R2021b

The input and output `PortWidth` properties of operating points and operating point search reports will be removed in a future release. Use the new `Nu` and `Ny` properties instead.

To update your code, change instances of `PortWidth` to either `Nu` or `Ny` as shown in the following table.

Not Recommended	Recommended

`[op,report] = findop('scdplane',10);`
`numOut = op.Outputs(1).PortWidth;`
`numIn = report.Inputs(1).PortWidth;`
	`[op,report] = findop('scdplane',10);`
	`numOut = op.Outputs(1).Ny;`
	`numIn = report.Inputs(1).Nu;`

See Also

`initopspec` | `linearize` | `findopOptions` | `operspec` | `addoutputspec`

Topics

“About Operating Points” on page 1-2
“Compute Steady-State Operating Points” on page 1-5
“Find Operating Points at Simulation Snapshots” on page 1-83

Introduced before R2006a

findopOptions

Set options for finding operating points from specifications

Syntax

```
options = findopOptions
options = findopOptions(Name,Value)
```

Description

`options = findopOptions` returns the default operating point search options.

`options = findopOptions(Name,Value)` returns an option set with additional options specified by one or more `Name,Value` pair arguments. Use this option set to specify options for the `findop` command.

Examples

Create Option Set for Operating Point Search

Create an option set for operating point search that sets the optimizer type to gradient descent and suppresses the display output of `findop`.

```
option = findopOptions('OptimizerType','graddescent','DisplayReport','off');
```

Alternatively, use dot notation to set the values of `options`.

```
options = findopOptions;
options.OptimizerType = 'graddescent';
options.DisplayReport = 'off';
```

Input Arguments

Name-Value Pair Arguments

Specify optional comma-separated pairs of `Name,Value` arguments. `Name` is the argument name and `Value` is the corresponding value. `Name` must appear inside quotes. You can specify several name and value pair arguments in any order as `Name1,Value1,...,NameN,ValueN`.

Example: `'DisplayReport','off'` suppresses the display of the operating point search report to the Command Window.

`OptimizerType` — Optimizer type used by the optimization algorithm
`'graddescent-elim'` (default) | `'graddescent'` | `'graddescent-proj'` | `'lsqnonlin'` | `'lsqnonlin-proj'` | `'simplex'`

Optimizer type used by the optimization algorithm, specified as the comma-separated pair consisting of `'OptimizerType'` and one of the following:

- `'graddescent-elim'` — Enforce an equality constraint to force the time derivatives of states to be zero (`dx/dt = 0`, `x(k+1) = x(k)`) and output signals to be equal to their specified known values. The optimizer fixes the states, `x`, and inputs, `u`, that are marked as `Known` in an operating point specification, and optimizes the remaining variables.
- `'graddescent'` — Enforce an equality constraint to force the time derivatives of states to be zero (`dx/dt = 0`, `x(k+1) = x(k)`) and the output signals to be equal to their specified known values. The optimizer also minimizes the error between the states, `x`, and inputs, `u`, and their respective known values from an operating point specification. If there are not any inputs or states marked as `Known`, `findop` attempts to minimize the deviation between the initial guesses for `x` and `u`, and their trimmed values.
- `'graddescent-proj'` — In addition to `'graddescent'`, enforce consistency of model initial conditions at each function evaluation. To specify whether constraints are hard or soft, use the `ConstraintType` option. This optimization method does not support analytical Jacobians.
- `'lsqnonlin'` — Fix the states, `x`, and inputs, `u`, marked as `Known` in an operating point specification, and optimize the remaining variables. The algorithm tries to minimize both the error in the time derivatives of the states (`dx/dt = 0`, `x(k+1) = x(k)`) and the error between the outputs and their specified known values.
- `'lsqnonlin-proj'` — In addition to `'lsqnonlin'`, enforce consistency of model initial conditions at each function evaluation. This optimization method does not support analytical Jacobians.
- `'simplex'` — Use the same cost function as `lsqnonlin` with the direct search optimization routine found in `fminsearch`.

For more information about these optimization algorithms, see `fmincon`, `lsqnonlin`, and `fminsearch`.

`OptimizationOptions` — Options for the optimization algorithm
structure

Options for the optimization algorithm, specified as the comma-separated pair consisting of `'OptimizationOptions'` and a structure created using the `optimset` function.

`DisplayReport` — Flag indicating whether to display the operating summary report
`'on'` (default) | `'off'` | `'iter'`

Flag indicating whether to display the operating point summary report, specified as the comma-separated pair consisting of `'DisplayReport'` and one of the following:

- `'on'` — Display the operating point summary report in the MATLAB command window when running `findop`.
- `'off'` — Suppress display of the summary report.
- `'iter'` — Display an iterative update of the optimization progress.

`AreParamsTunable` — Flag indicating whether to recompile the model when varying parameter values
`true` (default) | `false`

Flag indicating whether to recompile the model when varying parameter values for trimming, specified as the comma-separated pair consisting of `'AreParamsTunable'` and one of the following:

- `true` — Do not recompile the model when all varying parameters are tunable. If any varying parameters are not tunable, recompile the model for each parameter grid point, and issue a warning message.
- `false` — Recompile the model for each parameter grid point. Use this option when you vary the values of nontunable parameters.

`ConstraintType` — Constraint types for `'graddescent-proj'`
structure

Constraint types for `'graddescent-proj'` optimizer algorithm, specified as the comma-separated pair consisting of `'ConstraintType'` and a structure with the following fields:

- `dx` — Type for constraints on state derivatives
- `x` — Type for constraints on state values
- `y` — Type for constraints on output values

Specify each constraint as one of the following:

- `'hard'` — Enforce the constraints to be zero.
- `'soft'` — Minimize the constraints.

All constraint types are `'hard'` by default.

Output Arguments

`options` — Trimming options
`findopOptions` option set

Trimming options, returned as a `findopOptions` option set.

Compatibility Considerations

`'graddescent_elim'` value of the `Optimizer` property is now `'graddescent-elim'`
Behavior changed in R2017b

The `'graddescent_elim'` value of the `Optimizer` property of a `findopOptions` object is now `'graddescent-elim'`.

Update Code

To update your code, change the optimizer value from `graddescent_elim` to `graddescent-elim`. The following table shows the typical usage of this property value and how to update your code.

If your code has this form:	Use this code instead:
`opt = findopOptions('Optimizer',...` `'graddescent_elim');`	`opt = findopOptions('Optimizer',...` `'graddescent-elim')`
`opt = findopOptions;` `opt.Optimizer = 'graddescent_elim';`	`opt = findopOptions;` `opt.Optimizer = 'graddescent-elim';`

See Also
`findop`

Introduced in R2013b

frest.createFixedTsSinestream

Package: `frest`

Sinestream input signal with fixed sample time

Syntax

```
input = frest.createFixedTsSinestream(ts)
input = frest.createFixedTsSinestream(ts,{wmin wmax})
input = frest.createFixedTsSinestream(ts,w)
input = frest.createFixedTsSinestream(ts,sys)
input = frest.createFixedTsSinestream(ts,sys,{wmin wmax})
input = frest.createFixedTsSinestream(ts,sys,w)
```

Description

`input = frest.createFixedTsSinestream(ts)` creates sinestream input signal in which each frequency has the same fixed sample time `ts` in seconds. The signal has 30 frequencies between 1 and ω_s, where $\omega_s = \frac{2\pi}{t_s}$ is the sample rate in radians per second. The software adjusts the `SamplesPerPeriod` option to ensure that each frequency has the same sample time. Use when your Simulink model has linearization input I/Os on signals with discrete sample times.

`input = frest.createFixedTsSinestream(ts,{wmin wmax})` creates sinestream input signal with up to 30 frequencies logarithmically spaced between `wmin` and `wmax` in radians per second.

`input = frest.createFixedTsSinestream(ts,w)` creates sinestream input signal with frequencies `w`, specified as a vector of frequency values in radians per second. The values of `w` must satisfy $w = \frac{2\pi}{Nts}$ for integer N such that the sample rate $\omega_s = \frac{2\pi}{t_s}$ is an integer multiple of each element of `w`.

`input = frest.createFixedTsSinestream(ts,sys)` creates sinestream input signal with a fixed sample time `ts`. The signal's frequencies, settling periods, and number of periods automatically set based on the dynamics of a linear system `sys`.

`input = frest.createFixedTsSinestream(ts,sys,{wmin wmax})` creates sinestream input signal with up to 30 frequencies logarithmically spaced between `wmin` and `wmax` in radians per second.

`input = frest.createFixedTsSinestream(ts,sys,w)` creates sinestream input signal at frequencies `w`, specified as a vector of frequency values in radians per second. The values of `w` must satisfy $w = \frac{2\pi}{Nts}$ for integer N such that the sample rate `ts` is an integer multiple of each element of `w`.

Examples

Create a sinusoidal input signal with the following characteristics:

- Sample time of 0.02 sec
- Frequencies of the sinusoidal signal are between 1 rad/s and 10 rad/s

```
input = frest.createFixedTsSinestream(0.02,{1, 10});
```

See Also

`frest.Sinestream` | `frestimate`

Topics

“Estimation Input Signals” on page 5-24
“Estimate Frequency Response at the Command Line” on page 5-14
“Estimate Frequency Response Using Model Linearizer” on page 5-6

Introduced in R2009b

frest.createStep

Package: `frest`

Create step input signal

Syntax

```
input = frest.createStep(Name,Value)
```

Description

A step input signal has an initial value of `0` and transitions to a specified step size value after a specified step time. When performing frequency response estimation, step inputs are quick to simulate and can be useful as a first try when you do not have much knowledge about the system you are trying to estimate. However, the amplitude of the excitation decreases rapidly with increasing frequency. Therefore, step signals are best used to identify low-order plants where the slowest poles are dominant. Step inputs are not recommended for estimation across a wide range of frequencies.

When you use a step input signal for estimation, the frequencies returned in the estimated `frd` model depend on the length and sampling time of the signal. They are the frequencies obtained in the fast Fourier transform of the input signal (see the Algorithm section of `frestimate`).

For more information on input signals for frequency response estimation, see “Estimation Input Signals” on page 5-24.

`input = frest.createStep(Name,Value)` creates a step input signal for frequency response estimation using options specified using one or more name-value pair arguments.

Examples

Create Step Input Signal

Create a step input signal with a default sample time and the following properties:

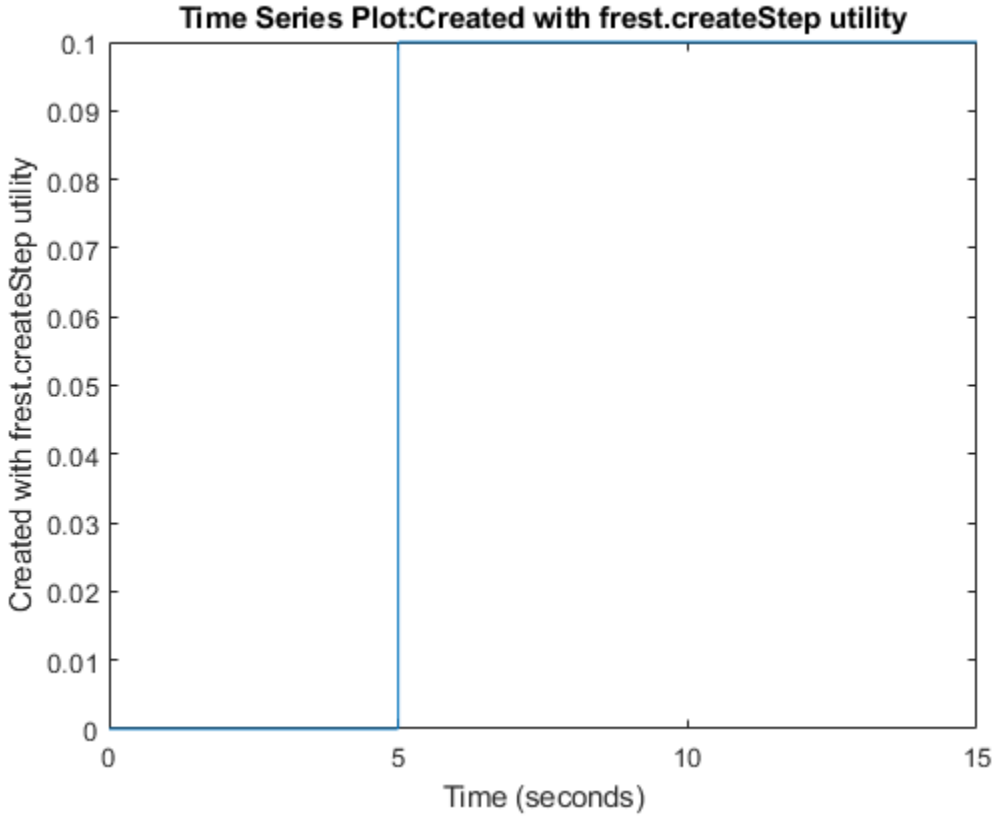
- Step time of `5` seconds
- Step size of `0.1`
- Total duration of `15` seconds

```
input = frest.createStep('StepTime',5,'StepSize',0.1,'FinalTime',15)
```

```
  timeseries

  Common Properties:
            Name: 'Created with frest.createStep utility'
            Time: [15001x1 double]
        TimeInfo: tsdata.timemetadata
            Data: [15001x1 double]
        DataInfo: tsdata.datametadata
```

Plot the step signal.

```
plot(input)
```



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 quotes. You can specify several name and value pair arguments in any order as `Name1,Value1,...,NameN,ValueN`.

Example: `'Ts',0.01` sets the input signal sample time to `0.01`

`Ts` — Sample time
`1e-3` (default) | positive scalar

Sample time of the input signal in seconds, specified as the comma-separated pair `'Ts'`, followed by a positive scalar.

`StepTime` — Step time
`1` (default) | positive scalar

Step time when the input signal transitions from `0` to `StepSize`, specified as the comma-separated pair `'StepTime'` followed by a positive scalar.

`StepTime` must be less than `FinalTime`

`StepSize` — Step size
`1` (default) | nonzero scalar

Step size, specified as the comma-separated pair `'StepSize'` followed by a nonzero scalar. The input signal has value `StepSize` after `StepTime` seconds.

`FinalTime` — Input signal duration
`10` (default) | positive scalar

Input signal duration in seconds, specified as the comma-separated pair `'FinalTime'` followed by a positive scalar.

`FinalTime` must be greater than `StepTime`

Output Arguments

`input` — Step input signal
`timeseries` object

Step input signal for frequency response estimation, returned as a `timeseries` object.

To view a plot of your input signal, type `plot(input)`.

See Also

`frest.simCompare` | `frestimate`

Topics
“Estimation Input Signals” on page 5-24
“Estimate Frequency Response at the Command Line” on page 5-14
“Estimate Frequency Response Using Model Linearizer” on page 5-6

Introduced in R2009b

frest.findDepend

Package: `frest`

List of model path dependencies

Syntax

```
dirs = frest.findDepend(model)
```

Description

`dirs = frest.findDepend(model)` returns paths containing Simulink model dependencies required for frequency response estimation using parallel computing. `model` is the Simulink model to estimate, specified as a character vector or a string. `dirs` is a cell array, where each element is a path character vector. `dirs` is empty when `frest.findDepend` does not detect any model dependencies. Append paths to `dirs` when the list of paths is empty or incomplete.

`frest.findDepend` does not return a complete list of model dependency paths when the dependencies are undetectable.

Examples

Specify Model Path Dependencies for Parallel Computing

To demonstrate a dependency on a file that is not in the current working folder, move the model files to a temporary folder and return the path to that folder. The `pathdepSetup` helper function also adds the temporary folder to the MATLAB® search path.

```
tempPath = pathdepSetup;
```

Open the Simulink® model.

```
mdl = 'scdpathdep';
open_system(mdl)
```

Obtain the model dependency path.

```
dirs = frest.findDepend(mdl)
```

```
dirs = 1×1 cell array
    {'C:/myTempFiles/tpd02d55f5_8b4c_489e_938c_ea004b9c771d'}
```

The resulting path is on the local drive `C:/`.

If you are using remote workers, 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 remote workers.

```
dirs = regexprep(dirs,'C:/','\\\\hostname\\C$\\')
```

Enable parallel computing and specify the model path dependencies.

```
options = frestimateOptions(...
    'UseParallel','on',...
    'ParallelPathDependencies',dirs);
```

You can now use these options for frequency response estimation using parallel computing.

```
io = getlinio(mdl);
in = frest.Sinestream('SimulationOrder','OneAtATime');
frd = frestimate(mdl,io,in,options);
```

After estimating the frequency response, you can close the model.

```
bdclose(mdl)
```

Return the model files to the current working folder and remove the temporary folder from the path.

```
pathdepCleanup(tempPath)
```

See Also

`frestimate`

Topics

“Speeding Up Estimation Using Parallel Computing” on page 5-69
“Dependency Analyzer Scope and Limitations”

Introduced in R2010a

frest.findSources

Package: `frest`

Identify time-varying source blocks

Syntax

```
blocks = frest.findSources(model)
blocks = frest.findSources(model,io)
```

Description

`blocks = frest.findSources(model)` finds all time-varying source blocks in the signal path of any linearization output point marked in the Simulink model `model`.

`blocks = frest.findSources(model,io)` finds all time-varying source blocks in the signal path of any linearization output point specified in the array of linear analysis points `io`.

Input Arguments

`model`

Character vector or string that contains the name of the Simulink model in which you are identifying time-varying source blocks for frequency response estimation.

`io`

Array of linearization I/O points.

The elements of `io` are linearization I/O objects that you create with `getlinio` or `linio`. `frest.findSources` uses only the output points to locate time-varying source blocks that can interfere with frequency response estimation. See “Algorithms” on page 18-54 for more information.

Output Arguments

`blocks`

Block paths of time-varying source blocks in `model` that can interfere with frequency response estimation, returned as an array of `Simulink.BlockPath` objects. `blocks` includes time-varying source blocks inside subsystems and normal-mode referenced models.

If you provide `io`, `blocks` contains all time-varying source blocks contributing to the signal at the output points in `io`.

If you do not provide `io`, `blocks` contains all time-varying source blocks contributing to the signal at the output points marked in `model`.

Examples

Estimate the frequency response of a model having time-varying source blocks. This example shows the use of `frest.findSources` to identify time-varying source blocks that interfere with frequency response estimation. You can also see the use of `BlocksToHoldConstant` option of `frestimateOptions` to disable time-varying source blocks in the estimation.

Load the model `scdspeed_ctrlloop`.

```
mdl = 'scdspeed_ctrlloop';
open_system(mdl)
% Convert referenced model to normal mode for accuracy
set_param('scdspeed_ctrlloop/Engine Model',...
                    'SimulationMode','Normal');
```

First, view the effects of time-varying source blocks on frequency response estimation. To do so, perform the estimation without disabling time-varying source blocks.

In this example, linearization I/O points are already defined in the model. Use the `getlinio` command to get the I/O points for `frestimate`.

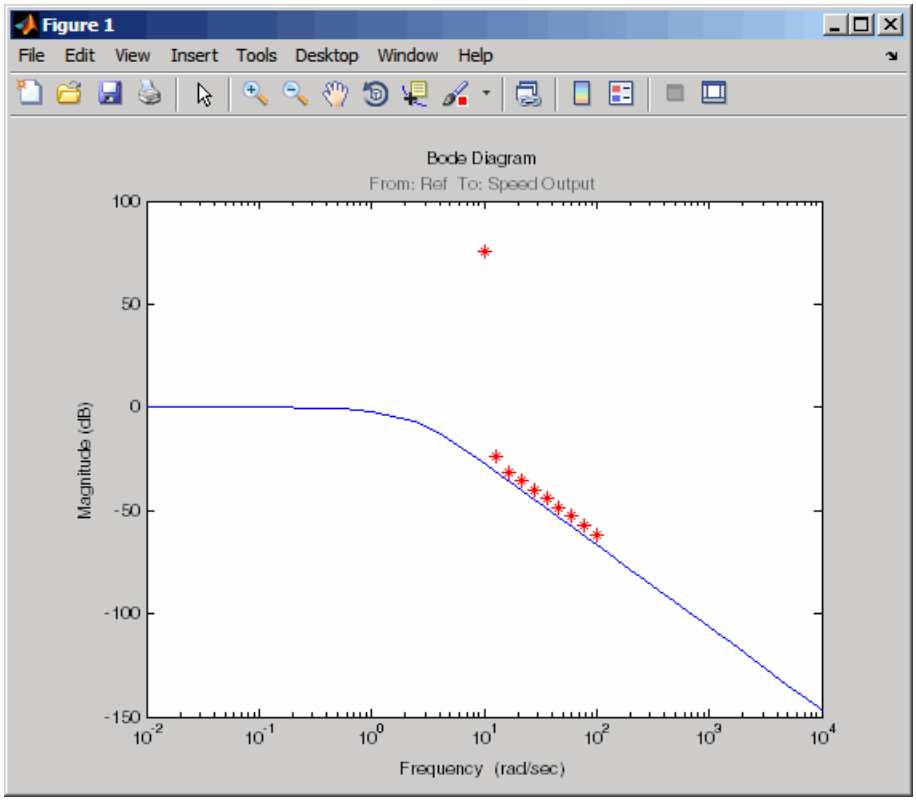
```
io = getlinio(mdl)
```

Define a sinestream signal and compute the estimated frequency response `sysest`.

```
in = frest.Sinestream('Frequency',logspace(1,2,10),...
              'NumPeriods',30,'SettlingPeriods',25);
[sysest,simout] = frestimate(mdl,io,in);
```

Perform exact linearization, and compare to the estimated response.

```
sys = linearize(mdl,io);
bodemag(sys,sysest,'r*')
```



The estimated frequency response does not match the exact linearization. The mismatch occurs because time-varying source blocks in the model prevent the response from reaching steady state.

Find the time-varying blocks using `frest.findSources`.

```
srcblks = frest.findSources(mdl);
```

`srcblks` is an array of block paths corresponding to the time-varying source blocks in the model. To examine the result, index into the array.

For example, entering

```
srcblks(2)
```

returns the result

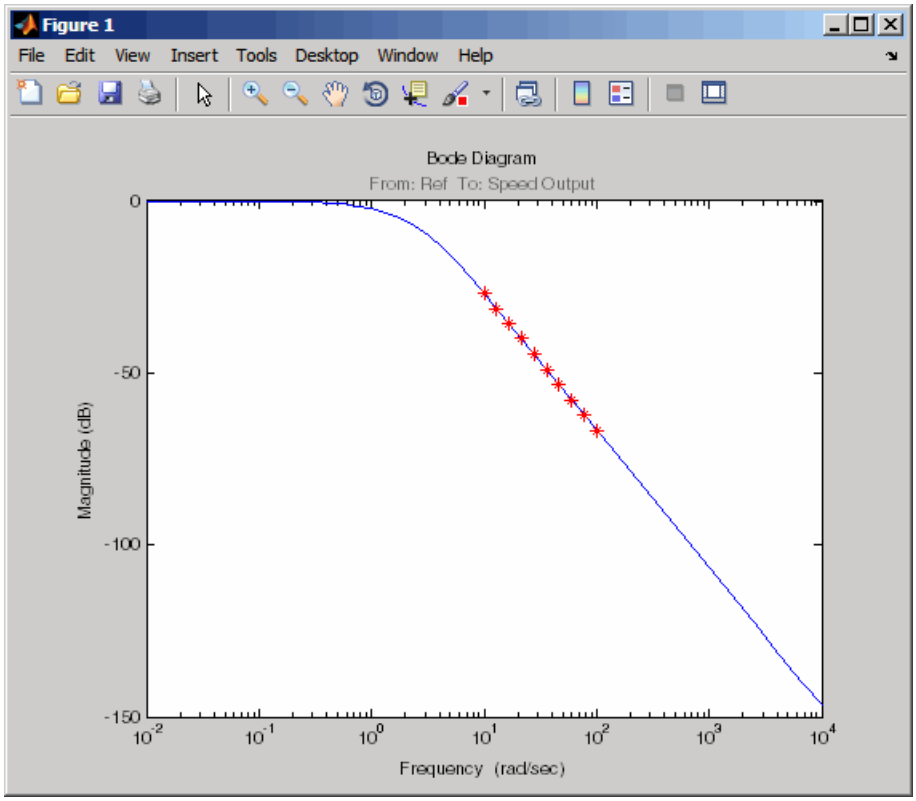
```
ans =

  Simulink.BlockPath
  Package: Simulink

  Block Path:
    'scdspeed_ctrlloop/Engine Model'
      'scdspeed_plantref/Drag Torque/Step1'
```

Now you can estimate the frequency response without the contribution of the time-varying source blocks. To do so, set the `BlocksToHoldConstant` option of `frestimateOptions` equal to `srcblks`, and run the estimation.

```
opts = frestimateOptions
opts.BlocksToHoldConstant = srcblks
% Run frestimate again with blocks disabled
 [sysest2,simout2] = frestimate(mdl,io,in,opts);
```

The frequency response estimate now provides a good match to the exact linearization result.

```
bodemag(sys,sysest2,'r*')
```



Tips

- Use `frest.findSources` to identify time-varying source blocks that can interfere with frequency response estimation. To disable such blocks to estimate frequency response, set the `BlocksToHoldConstant` option of `frestimateOptions` equal to `blocks` or a subset of `blocks`. Then, estimate the frequency response using `frestimate`.
- Sometimes, `model` includes referenced models containing source blocks in the signal path of a linearization output point. In such cases, set the referenced models to normal simulation mode to ensure that `frest.findSources` locates them. Use the `set_param` command to set `SimulationMode` of any referenced models to `Normal` before running `frest.FindSources`.

Algorithms

To locate time-varying source blocks that can interfere with frequency response estimation, `frest.findSources` begins at each linearization output point in the model. From each output point, the algorithm traces every signal path backward block by block. The algorithm reports any source block (a block with no input port) it discovers, unless that source block is a Constant or Ground block.

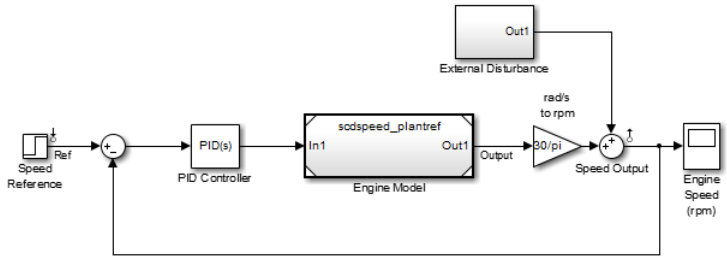
The `frest.findSources` algorithm traces every signal path that can affect the signal value at each linearization output point in the model. The paths traced include:

- Signal paths inside virtual and nonvirtual subsystems.
- Signal paths inside normal-mode referenced models. Set all referenced models to normal simulation mode before using `frest.findSources` to ensure that the algorithm identifies source blocks within the referenced models.
- Signals routed through From and Goto blocks, or through Data Store Read and Data Store Write blocks.
- Signals routed through switches. The `frest.findSources` algorithm assumes that any pole of a switch can be active during frequency response estimation. The algorithm therefore follows the signal back through all switch inputs.

Note Some time-varying source blocks might not be found by `frest.findSources`. If the internal signal path of a block does not contain a block with no input port, that block is not reported by the function.

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:

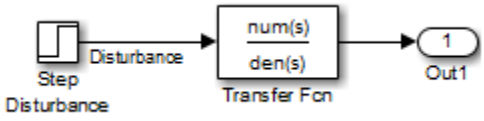
```
set_param('scdspeed_ctrlloop/Engine Model',...
                        'SimulationMode','Normal');
```



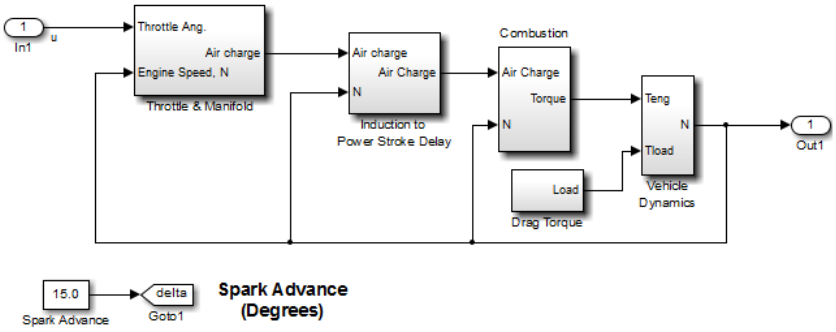
You can now run `frest.findSources` to identify the time-varying source blocks using the linearization output point defined in the model.

```
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 “Simulink Control Design Checks” on page 21-2. For more information about using the Model Advisor, see “Check Your Model Using the Model Advisor”.

See Also

`frestimate` | `frestimateOptions`

Topics

“Effects of Time-Varying Source Blocks on Frequency Response Estimation” on page 5-53

Introduced in R2010b

frest.simCompare

Package: `frest`

Plot time-domain simulation of nonlinear and linear models

Syntax

```
frest.simCompare(simout,sys,input)
frest.simCompare(simout,sys,input,x0)
[y,t] = frest.simCompare(simout,sys,input)
[y,t,x] = frest.simCompare(simout,sys,input,x0)
```

Description

`frest.simCompare(simout,sys,input)` plots both

- Simulation output, `simout`, of the nonlinear Simulink model

  You obtain the output from the `frestimate` command.
- Simulation output of the linear model `sys` for the input signal `input`

  The linear simulation results are offset by the initial output values in the `simout` data.

`frest.simCompare(simout,sys,input,x0)` plots the frequency response simulation output and the simulation output of the linear model with initial state `x0`. Because you specify the initial state, the linear simulation result is *not* offset by the initial output values in the `simout` data.

`[y,t] = frest.simCompare(simout,sys,input)` returns the linear simulation output response `y` and the time vector `t` for the linear model `sys` with the input signal `input`. This syntax does not display a plot. The matrix `y` has as many rows as time samples `(length(t))` and as many columns as system outputs.

`[y,t,x] = frest.simCompare(simout,sys,input,x0)` also returns the state trajectory `x` for the linear state space model `sys` with initial state `x0`.

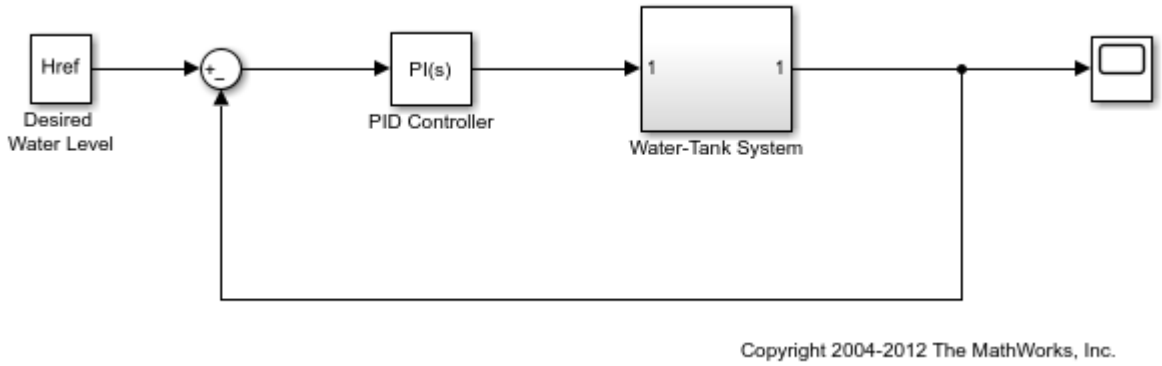
Examples

Compare Simulated Model and Linear Model in Time Domain

`frest.simCompare` lets you lets you examine the results of frequency response estimation in the time domain. You can compare the simulated model response to the response of a linear model of the system, such as one obtained by exact linearization.

Estimate the closed-loop response of the plant in the `watertank` model. First, open the model.

```
model = 'watertank';
open_system(model)
```



Copyright 2004-2012 The MathWorks, Inc.

Define a linearization I/O set that specifies the plant, and find a steady-state operating point for estimation.

```
io(1)=linio('watertank/PID Controller',1,'input');
io(2)=linio('watertank/Water-Tank System',1,'output');

watertank_spec = operspec(model);
opOpts = findopOptions('DisplayReport','off');
op = findop(model,watertank_spec,opOpts);
```

Create an input signal for estimation. For this example, use a step input.

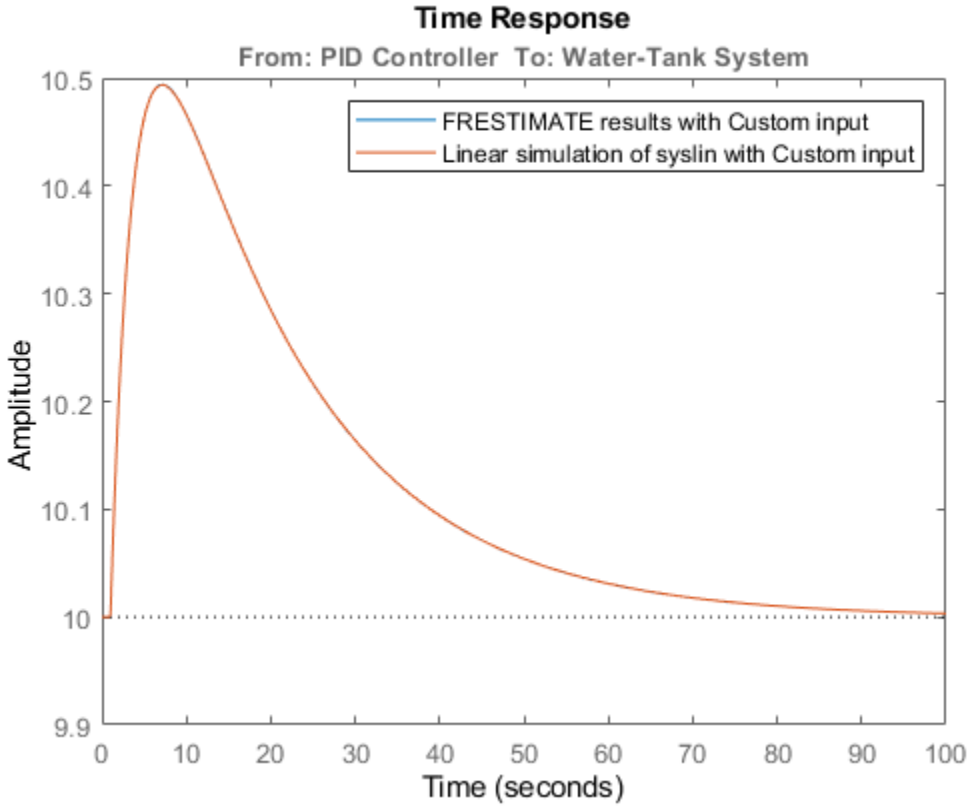
```
input = frest.createStep('FinalTime',100);
```

Estimate the frequency response of the specified portion of the model, using the `simout` output argument to store the data generated during the estimation process. Also, linearize the model using the same I/O set and operating point.

```
[sysest,simout] = frestimate(model,op,io,input);
syslin = linearize(model,io,op);
```

Examine the time-domain responses of the linearized model and the Simulink model to the same input signal.

```
frest.simCompare(simout,syslin,input)
legend
```



In this example, the responses are virtually identical.

See Also

`frestimate` | `frest.simView`

Introduced in R2009b

frest.simView

Package: `frest`

Plot frequency response model in time- and frequency-domain

Syntax

```
frest.simView(simout,input,sysest)
frest.simView(simout,input,sysest,sys)
```

Description

`frest.simView(simout,input,sysest)` plots the following frequency response estimation results:

- Time-domain simulation `simout` of the Simulink model
- FFT of time-domain simulation `simout`
- Bode of estimated system `sysest`

  This Bode plot is available when you create the input signal using `frest.Sinestream` or `frest.Chirp`. In this plot, you can interactively select frequencies or a frequency range for viewing the results in all three plots.

You obtain `simout` and `sysest` from the `frestimate` command using the input signal `input`.

`frest.simView(simout,input,sysest,sys)` includes the linear system `sys` in the Bode plot when you create the input signal using `frest.Sinestream` or `frest.Chirp`. Use this syntax to compare the linear system to the frequency response estimation results.

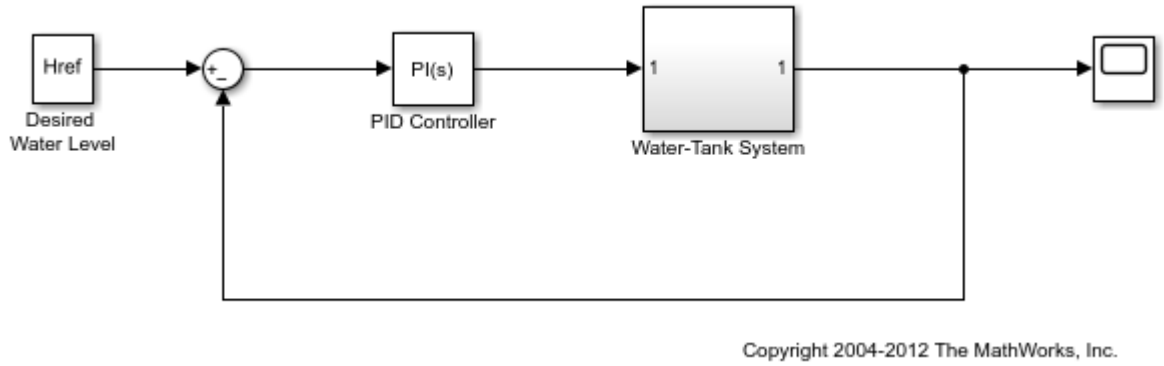
Examples

Examine Estimation Results Using Simulation Results Viewer

The Simulation Results Viewer lets you examine the results of frequency response estimation frequency by frequency. You open the viewer using the `frest.simView` command. To do so, store the simulation data using the `simout` output argument of `frestimate`.

Estimate the open-loop response of the plant in the `watertank` model. First, open the model.

```
model = 'watertank';
open_system(model)
```



Copyright 2004-2012 The MathWorks, Inc.

Define a linearization I/O set that specifies the plant, and find a steady-state operating point for estimation.

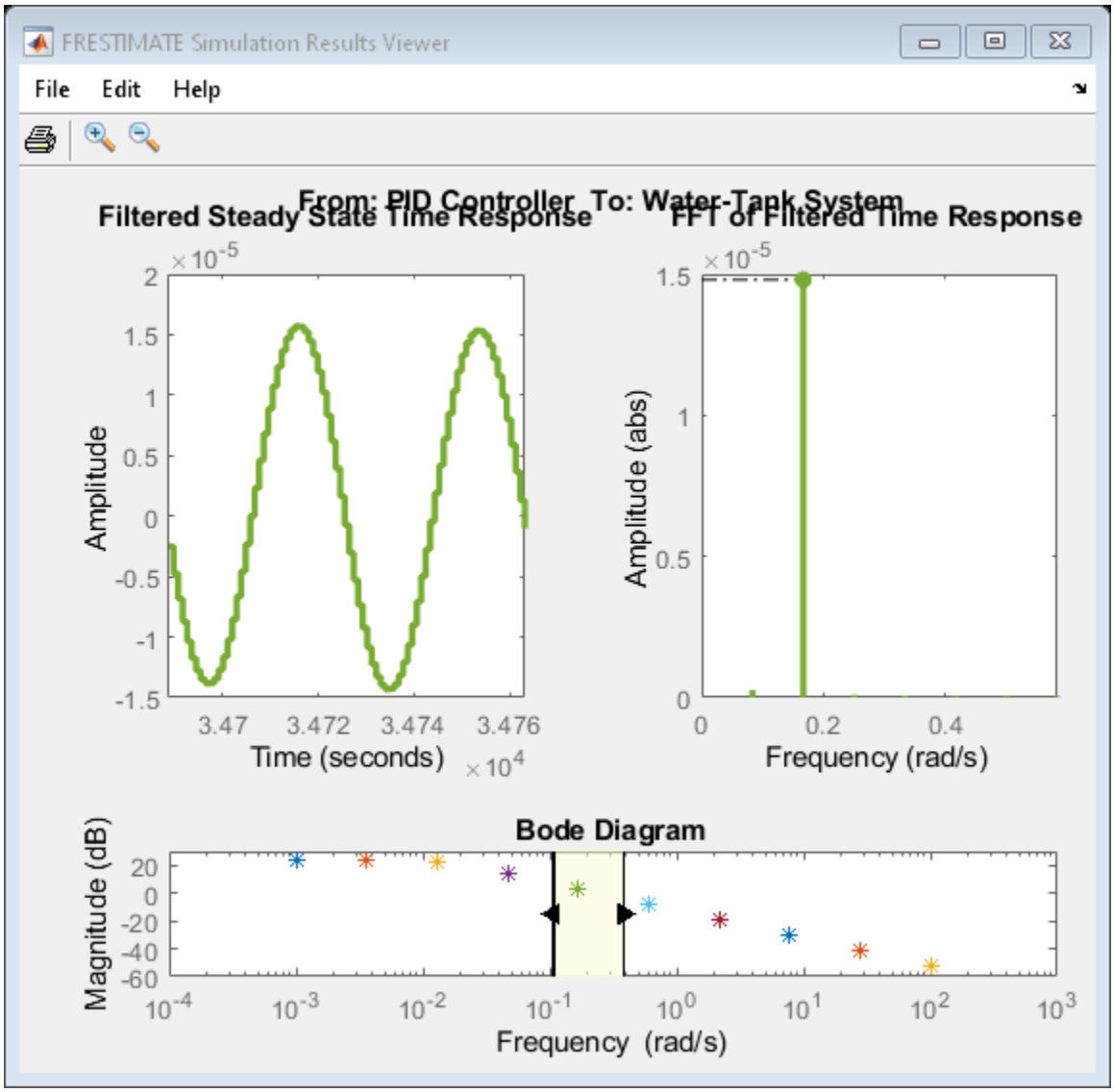
```
io(1)=linio('watertank/PID Controller',1,'input');
io(2)=linio('watertank/Water-Tank System',1,'openoutput');

watertank_spec = operspec(model);
opOpts = findopOptions('DisplayReport','off');
op = findop(model,watertank_spec,opOpts);
```

Then, create an input signal for estimation, and estimate the frequency response of the specified portion of the model. Use the `simout` output argument to store the estimation data.

```
input = frest.Sinestream('Frequency',logspace(-3,2,10));
[sysest,simout] = frestimate(model,op,io,input);
```

Open the Simulation Results Viewer.

```
frest.simView(simout,input,sysest)
```



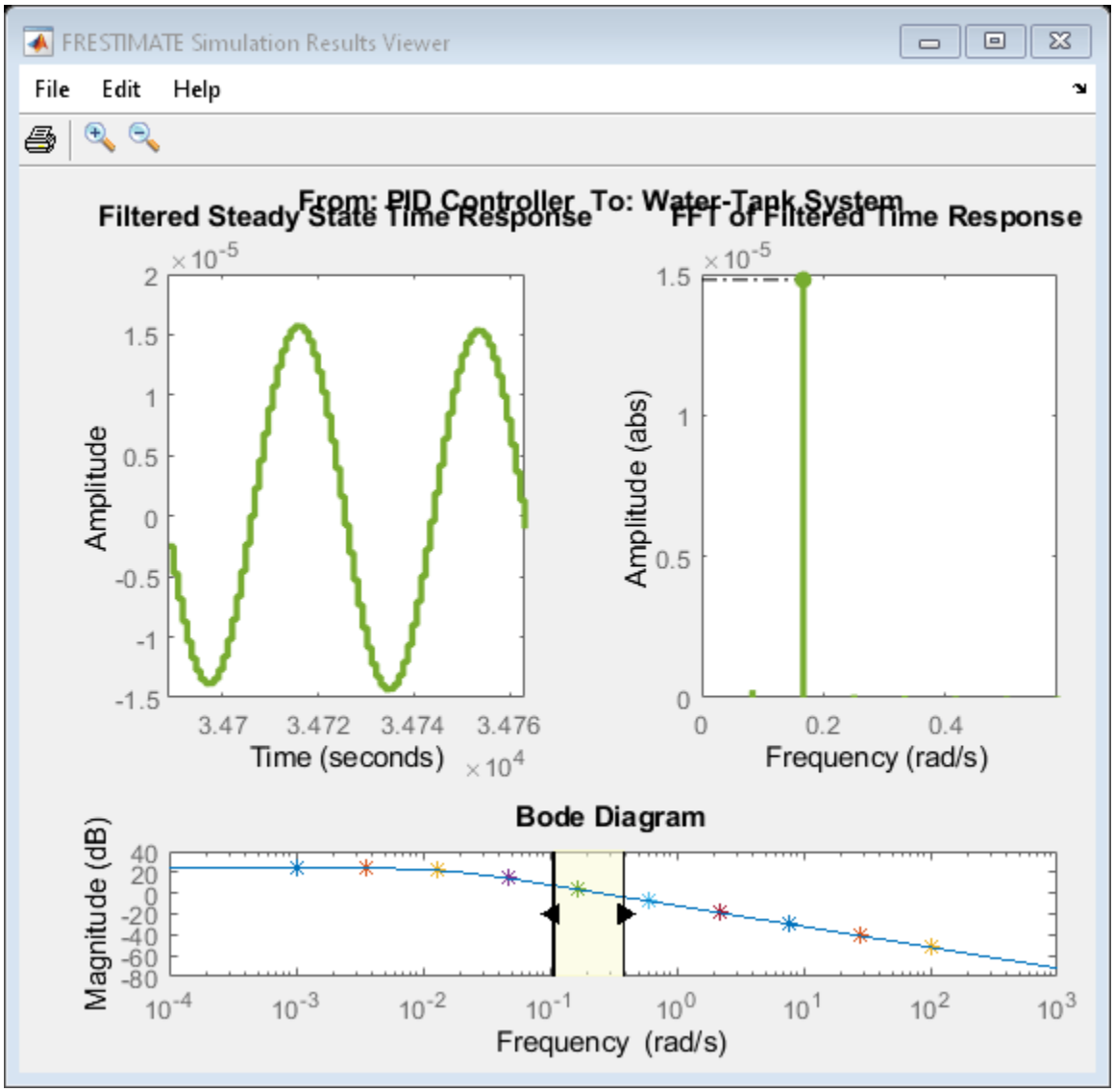
The viewer shows you the steady-state time response and the FFT of that response for all frequencies within the range you select on the Bode Diagram section of the viewer. These plots can help you identify when the response deviates from the expected response. For more information about using the Simulation Results Viewer, see “Analyze Estimated Frequency Response” on page 5-18.

If you have a linear model of the system you are estimating, you can use the model as a baseline response for comparison in the viewer. For instance, you can compare a model obtained by exact linearization to the estimated frequency response. Use the linearization I/O set and the operating point to compute an exact linearization of the `watertank` plant.

```
syslin = linearize(model,io,op);
```

Open the Simulation Results Viewer again, this time providing `syslin` as an input argument.

```
frest.simView(simout,input,sysest,syslin)
```



The Bode Diagram section of the viewer includes a line showing the exact response `syslin`. This view can be useful to identify particular frequencies where the estimated response deviates from the linearization.

See Also

`frestimate` | `frest.simCompare`

Topics

"Analyze Estimated Frequency Response" on page 5-18
"Troubleshooting Frequency Response Estimation" on page 5-43

Introduced in R2009b

frestimate

Frequency response estimation of Simulink models

Syntax

```
sysest = frestimate(model,io,input)
sysest = frestimate(model,op,io,input)
[sysest,simout] = frestimate(model,op,io,input)
[___] = frestimate(___,options)

sysest = frestimate(data,freqs,units)
```

Description

`sysest = frestimate(model,io,input)` estimates the frequency response of a Simulink model using the specified input signal, the operating point defined by the model initial conditions, and the analysis points specified in `io`.

`sysest = frestimate(model,op,io,input)` initializes the model at the operating point `op` before estimating the frequency response. If the model initial conditions are not at steady state or not the operating point of interest, use this syntax to specify a different operating point.

`[sysest,simout] = frestimate(model,op,io,input)` also returns the simulated model output. Use this syntax when you want to examine the estimation results using the Simulation Results Viewer (`frest.simView`).

`[___] = frestimate(___,options)` computes the frequency response using additional options. You can use this syntax with any of the previous input and output argument combinations.

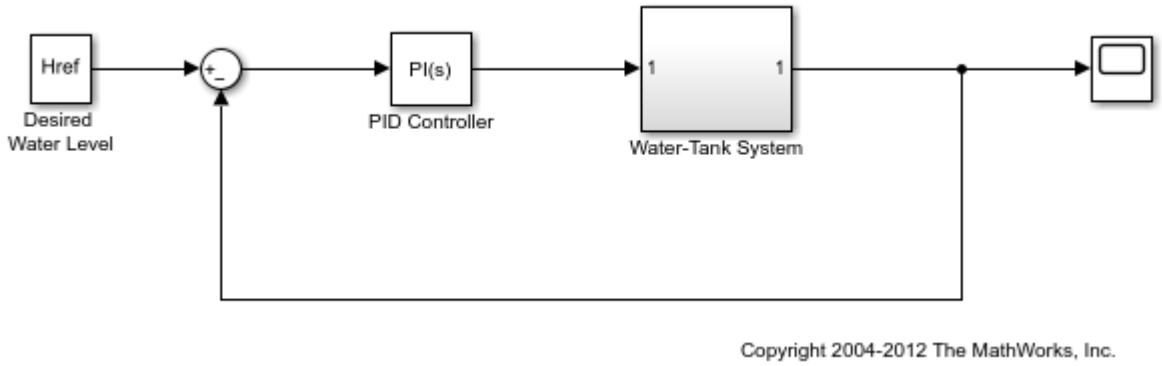
`sysest = frestimate(data,freqs,units)` estimates the frequency response using simulation data obtained using the Frequency Response Estimator block in offline estimation mode. Use this syntax only with data logged using that block.

Examples

Estimate Frequency Response of a Portion of a Simulink Model

Estimate the open-loop response of the plant in the `watertank` model. Open the model.

```
model = 'watertank';
open_system(model)
```



Copyright 2004-2012 The MathWorks, Inc.

To estimate the open-loop response of the plant, define a linearization I/O set that specifies this portion of the model with analysis points. Define an input analysis point at the controller output, and define an open-loop output point at the plant output.

```
io(1)=linio('watertank/PID Controller',1,'input');
io(2)=linio('watertank/Water-Tank System',1,'openoutput');
```

Find a steady-state operating point for the estimation. For this example, use a steady-state operating point derived from the model initial conditions.

```
watertank_spec = operspec(model);
opOpts = findopOptions('DisplayReport','off');
op = findop(model,watertank_spec,opOpts);
```

Create an input signal for estimation. For this example, use a sinestream signal, which sends a series of separate sinusoidal perturbations at the frequencies you specify.

```
input = frest.Sinestream('Frequency',logspace(-3,2,30));
```

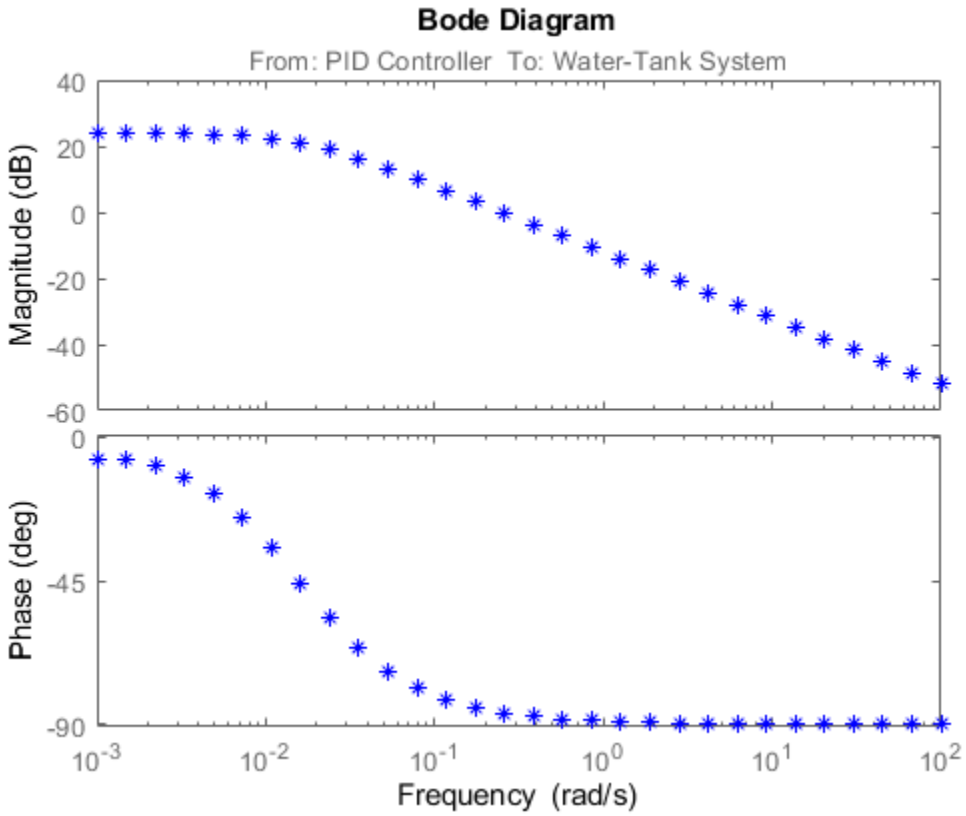
Estimate the frequency response of the specified portion of the model. The result is a frequency-response model containing responses at each of the frequencies specified in the sinestream signal.

```
sysest = frestimate(model,op,io,input);
size(sysest)
```

```
FRD model with 1 outputs, 1 inputs, and 30 frequency points.
```

Examine the measured frequency response.

```
bode(sysest,'*')
```

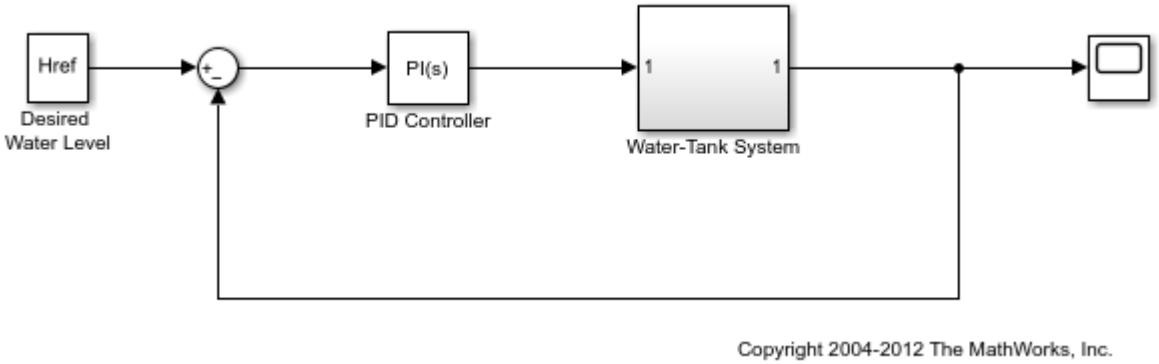


Validate Exact Linearization Results Using Estimated Frequency Response

Linearize a Simulink model and use frequency-response estimation to validate the exact linearization results.

Open the `watertank` model.

```
model = 'watertank';
open_system(model);
```



Obtain a linearization of the open-loop response of the plant. To do so, define the linearization I/O points, and find a steady-state operating point near the model initial conditions. Then, linearize the model.

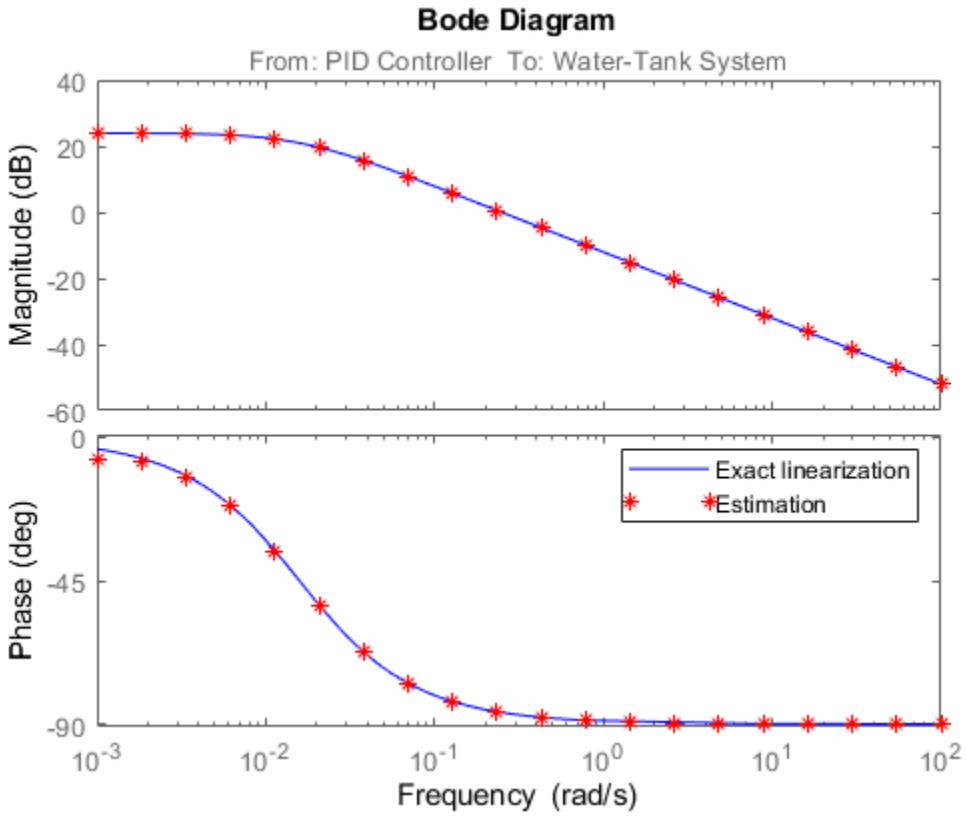
```
io(1)=linio('watertank/PID Controller',1,'input');
io(2)=linio('watertank/Water-Tank System',1,'openoutput');

watertank_spec = operspec(model);
opOpts = findopOptions('DisplayReport','off');
op = findop(model,watertank_spec,opOpts);

syslin = linearize(model,op,io);
```

To check the linearization, use the same analysis points and operating point to estimate the frequency response. For this example, use a sinestream input signal for the estimation.

```
input = frest.Sinestream('Frequency',logspace(-3,2,20));
sysest = frestimate(model,op,io,input);
```

Compare the exact linearization and the estimated response in the frequency domain using a Bode plot.

```
bode(syslin,'b-',sysest,'r*')
legend('Exact linearization','Estimation')
```

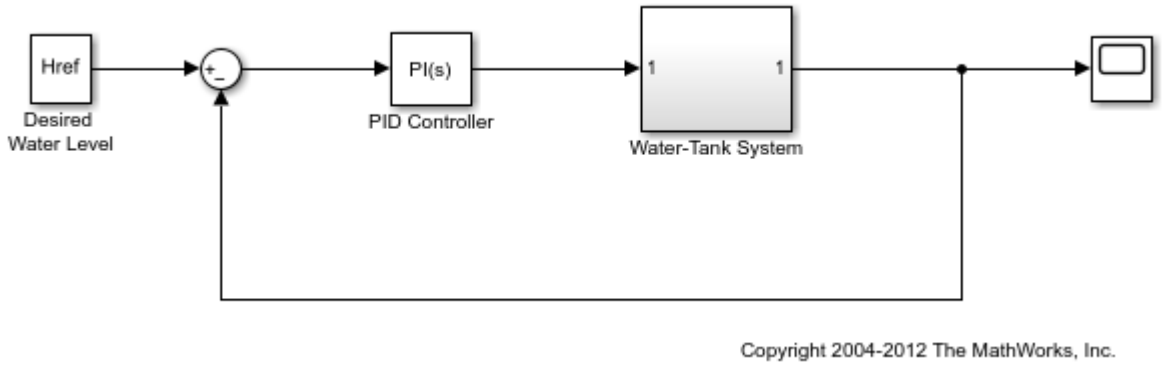


Examine Estimation Results Using Simulation Results Viewer

The Simulation Results Viewer lets you examine the results of frequency response estimation frequency by frequency. You open the viewer using the `frest.simView` command. To do so, store the simulation data using the `simout` output argument of `frestimate`.

Estimate the open-loop response of the plant in the `watertank` model. First, open the model.

```
model = 'watertank';
open_system(model)
```



Define a linearization I/O set that specifies the plant, and find a steady-state operating point for estimation.

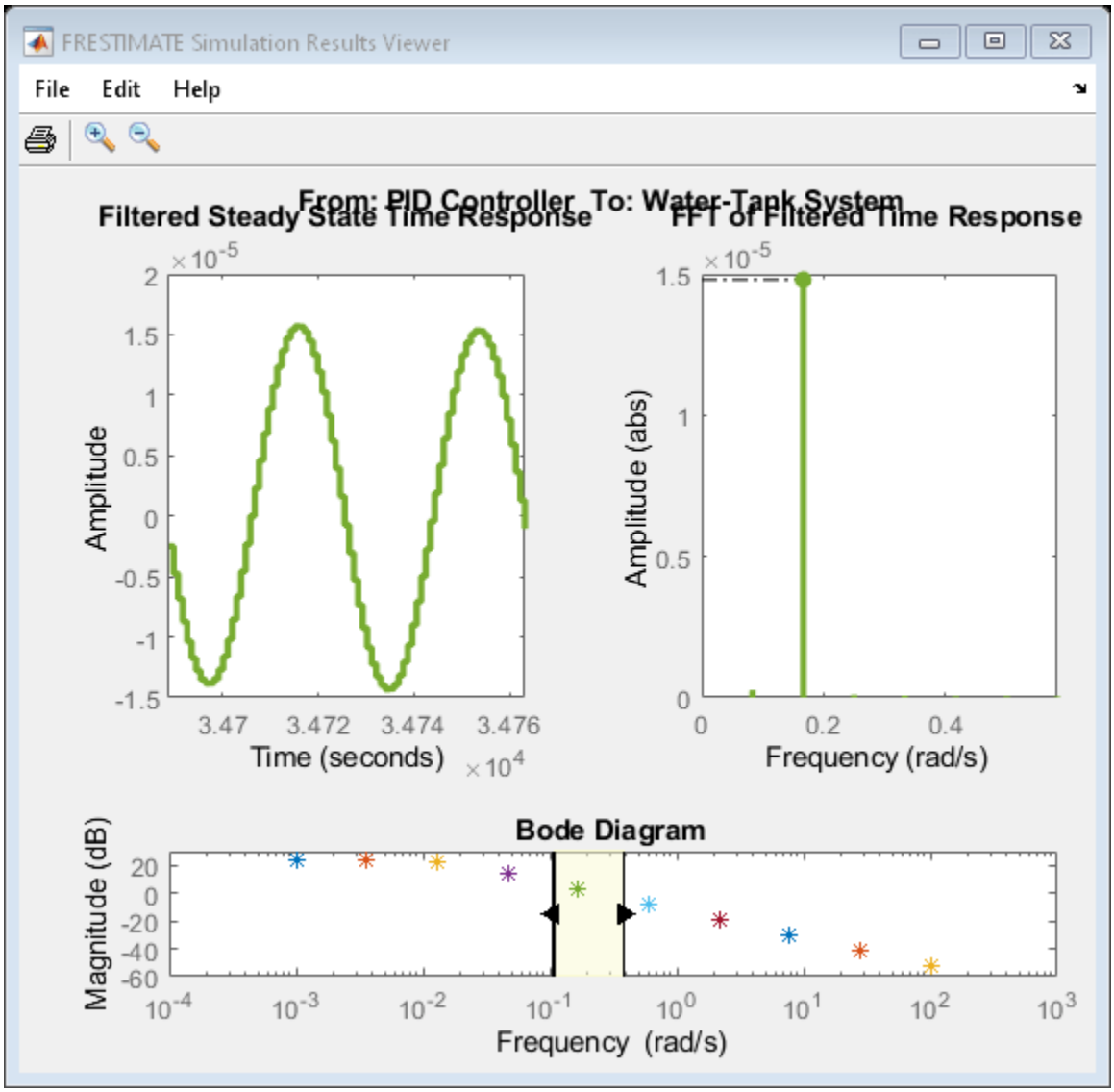
```
io(1)=linio('watertank/PID Controller',1,'input');
io(2)=linio('watertank/Water-Tank System',1,'openoutput');

watertank_spec = operspec(model);
opOpts = findopOptions('DisplayReport','off');
op = findop(model,watertank_spec,opOpts);
```

Then, create an input signal for estimation, and estimate the frequency response of the specified portion of the model. Use the `simout` output argument to store the estimation data.

```
input = frest.Sinestream('Frequency',logspace(-3,2,10));
[sysest,simout] = frestimate(model,op,io,input);
```

Open the Simulation Results Viewer.

```
frest.simView(simout,input,sysest)
```



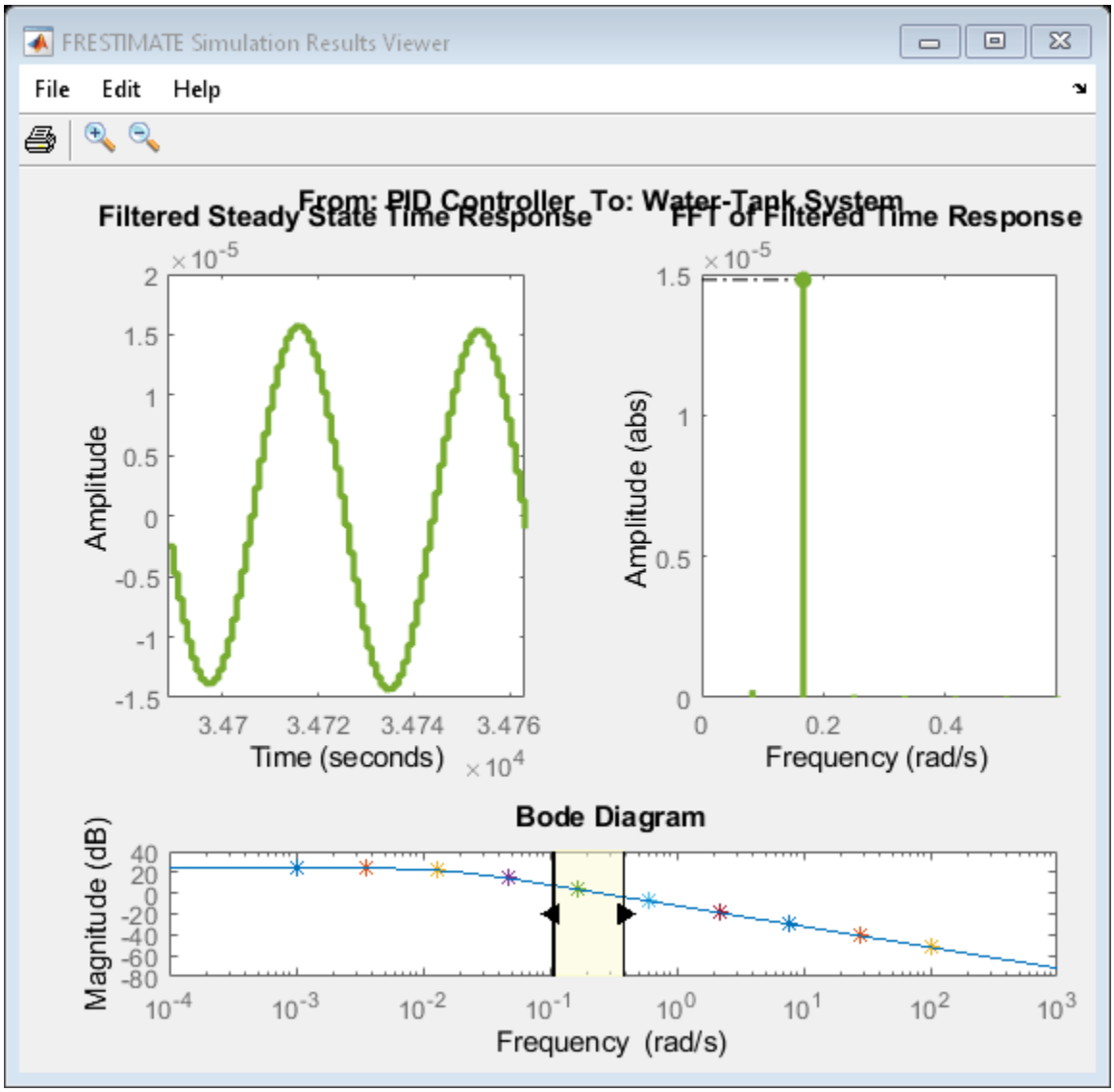
The viewer shows you the steady-state time response and the FFT of that response for all frequencies within the range you select on the Bode Diagram section of the viewer. These plots can help you identify when the response deviates from the expected response. For more information about using the Simulation Results Viewer, see “Analyze Estimated Frequency Response” on page 5-18.

If you have a linear model of the system you are estimating, you can use the model as a baseline response for comparison in the viewer. For instance, you can compare a model obtained by exact linearization to the estimated frequency response. Use the linearization I/O set and the operating point to compute an exact linearization of the `watertank` plant.

```
syslin = linearize(model,io,op);
```

Open the Simulation Results Viewer again, this time providing `syslin` as an input argument.

```
frest.simView(simout,input,sysest,syslin)
```



The Bode Diagram section of the viewer includes a line showing the exact response `syslin`. This view can be useful to identify particular frequencies where the estimated response deviates from the linearization.

Input Arguments

`model` — Simulink model

string | character vector

Simulink model, specified as a string or character vector. The model must be in the current working folder or on the MATLAB path.

`io` — Analysis points set

linearization I/O object

Analysis points set that contain inputs, outputs, and loop openings, specified as a linearization I/O object. The analysis point set defines the subset of the Simulink model whose frequency response you want to estimate. To create `io`:

- Define the inputs, outputs, and openings using `linio`.
- If the inputs, outputs, and openings are specified in the Simulink model, extract these points from the model using `getlinio`.

For frequency response estimation, I/O points cannot be on bus signals. `io` must correspond to the Simulink model `model` or a normal mode model reference in the model hierarchy. (If you use `frestimate` with an output analysis point in a model reference, the **Total number of instances allowed per top model** configuration parameter of the referenced model must be 1.)

Specifying I/O points for estimation is similar to specifying them for linearization. For more information on specifying linearization inputs, outputs, and openings, see “Specify Portion of Model to Linearize” on page 2-10.

`input` — Input signal
sinestream signal | chirp signal | random signal | time series

Input signal for perturbing the model, specified as one of the following:

- A sinestream signal, specified using `frest.Sinestream` or `frest.createFixedTsSinestream`
- A chirp signal, specified using `frest.Chirp`
- A random signal, specified using `frest.Random`
- A step signal, specified using `frest.createStep`
- An arbitrary signal, specified as a MATLAB `timeseries`

For more information about creating input signals for frequency response estimation, see “Estimation Input Signals” on page 5-24.

`op` — Operating point
`OperatingPoint` object

Operating point at which to initialize the model for estimation, specified as an `OperatingPoint` object created using one of the following functions.

- `operpoint`
- `findop` with either a single operating point specification or a single snapshot time

Generally, you use a steady-state operating point for estimation. If you do not specify an operating point, the estimation process begins at the operating point specified by the model initial conditions. This operating point consists of the initial state and input signal values stored in the model.

`options` — Estimation options
`frestimateOptions` object

Estimation options, specified as a `frestimateOptions` object. Available options include enabling parallel computing for estimation (requires Parallel Computing Toolbox).

`data` — Response data logged for offline estimation
structure | `Simulink.SimulationData.Dataset` object

Response data logged for offline estimation using the Frequency Response Estimator block, specified as one of the following:

- A structure obtained by writing the data from the **data** output port of the block to the MATLAB workspace using a To Workspace block. The **Save format** parameter of the To Workspace block must be `Timeseries`.
- A `Simulink.SimulationData.Dataset` object obtained by using Simulink data logging to write the data at the **data** port to the MATLAB workspace.

For more information, see the **data** port description on the Frequency Response Estimator block reference page or “Collect Frequency Response Experiment Data for Offline Estimation” on page 6-18.

`freqs` — Frequencies for offline estimation
vector

Frequencies for offline estimation, specified as a vector of positive values. When you collect response data using the Frequency Response Estimator block, you specify the frequencies for the estimation experiment using the **Frequencies** parameter of the block. Use the same vector of frequencies for `freqs` when you perform offline estimation with the logged data.

`units` — Units
`"rad/s"` | `"Hz"` | `'rad/s'` | `'Hz'`

Units of frequencies for offline estimation, specified as one of the strings `"rad/s"` or `"Hz"` or one of the character vectors `'rad/s'` or `'Hz'`. When you collect response data using the Frequency Response Estimator block, you specify the units of the frequencies for the estimation experiment using the frequency units block parameter. Specify the same units when you perform offline estimation with the logged data.

Output Arguments

`sysest` — Estimated frequency response
`frd` model

Estimated frequency response, returned as a frequency-response (`frd`) model object. The `frd` model has as many inputs and outputs as are specified in the linearization analysis points `io`.

The frequencies in `sysest` depend on what input signal you use for estimation, as follows:

- If you use a sinestream signal created with `frest.Sinestream`, the frequencies in `sysest` are the frequencies specified in the sinestream signal.
- If you use any other input signal, the frequencies are determined by the FFT computation that the function performs to extract the frequency response (see “Algorithms” on page 18-73).

If you use the `data` input argument to provide data collected using the Frequency Response Estimator block, then `sysest` is a SISO model. In this case, the frequencies in `sysest` are the frequencies you supply with the `freqs` input argument.

`simout` — Simulation data
cell array of `Simulink.Timeseries` objects

Simulation data collected during the estimation process, returned as a cell array of `Simulink.Timeseries` objects. The cell array has dimensions `m`-by-`n`, where `m` is the number of output points in the I/O set `io`, and `n` is the number of input points. This data can be useful for:

- Examining estimation results using `frest.simView` (see “Examine Estimation Results Using Simulation Results Viewer” on page 18-67)
- Examining time-domain responses using `frest.simCompare`

Limitations

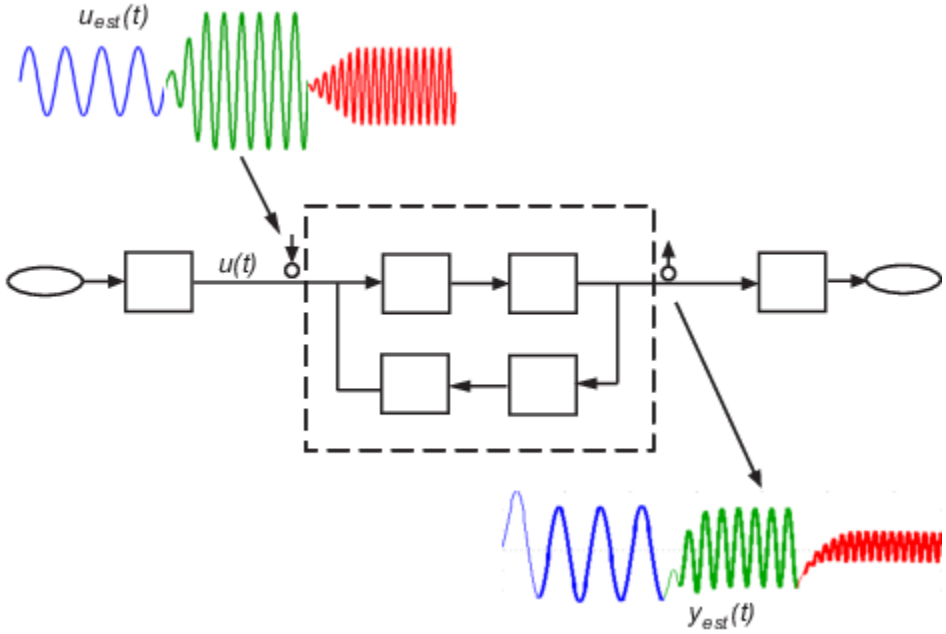
- If you use `frestimate` with an output analysis point in a model reference, the **Total number of instances allowed per top model** configuration parameter of the referenced model must be 1.

Tips

- For multiple-input multiple-output (MIMO) systems, `frestimate` 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 signals 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.

Algorithms

`frestimate` injects the input signal you specify ($u_{est}(t)$) at the input analysis points. It simulates the model and collects the response signal ($y_{est}(t)$) at the output analysis points, as illustrated below for a sinestream input.



In general, `frestimate` estimates the frequency response by computing the ratio of the fast Fourier transforms output signal and the input signal:

$$Resp = \frac{FFT(y_{est}(t))}{FFT(u_{est}(t))}.$$

- For sinestream input signals, the function discards the data collected during the specified settling periods of the signal at each frequency. (See “Sinestream Input Signals” on page 5-29.) If the filtering option of the sinestream signal is active, the function then applies a bandpass filter to the remaining signal at the corresponding frequency and discards one more period to remove any remaining transient signals. The function uses the FFT of the resulting signal to compute *Resp*. The resulting `frd` model contains all frequencies in the sinestream.
- For chirp input signals, the function discards any frequencies in the ratio *Resp* that fall outside the frequency range specified for the chirp. The resulting `frd` model contains all frequencies in the Fourier transform that fall within the chirp range.
- For other input signals, the resulting `frd` contains all the frequencies in the Fourier transform.

Estimation Using Data from Frequency Response Estimator Block

You can use the `frestimate(data,freqs,units)` syntax to perform offline estimation with data from the Frequency Response Estimator block. In this case, `frestimate` uses the `Ready` field of the data structure to determine which data points to include the FFT computation of *Resp*.

- For sinestream mode, this signal indicates which periods to discard at each frequency, determined by the **Number of settling periods** block parameter.
- For superposition mode, this signal indicates which data falls within the data-collection window determined by the **Number of periods of the lowest frequency used for estimation** parameter.

`frestimate` interpolates *Resp* to generate the resulting `frd` model, which contains the frequencies you specified in the block experiment parameters. For more information, see the Frequency Response Estimator block reference page.

Alternative Functionality

App

Model Linearizer

Simulink Block

Frequency Response Estimator

Extended Capabilities

Automatic Parallel Support

Accelerate code by automatically running computation in parallel using Parallel Computing Toolbox™.

To run in parallel, set `'UseParallel'` to `true` using `frestimateOptions` (requires Parallel Computing Toolbox).

For more information, see “Managing Estimation Speed and Memory” on page 5-68.

See Also

`frest.Sinestream` | `frest.Chirp` | `frest.Random` | `frest.simView` | `frestimateOptions` | `getSimulationTime`

Topics

“Estimate Frequency Response at the Command Line” on page 5-14
“Estimate Frequency Response Using Model Linearizer” on page 5-6
“Speeding Up Estimation Using Parallel Computing” on page 5-69

Introduced in R2009b

frestimateOptions

Options for frequency response estimation

Syntax

```
options = frestimateOptions
options = frestimateOptions('OptionName',OptionValue)
```

Description

`options = frestimateOptions` creates a frequency response estimation options object, `options`, with default settings. Pass this object to the function `frestimate` to use these options for frequency response estimation.

`options = frestimateOptions('OptionName',OptionValue)` creates a frequency response estimation options object `options` using the options specified by comma-separated name/value pairs.

Input Arguments

`'OptionName',OptionValue`

Estimation options, specified as comma-separated option name and option value pairs.

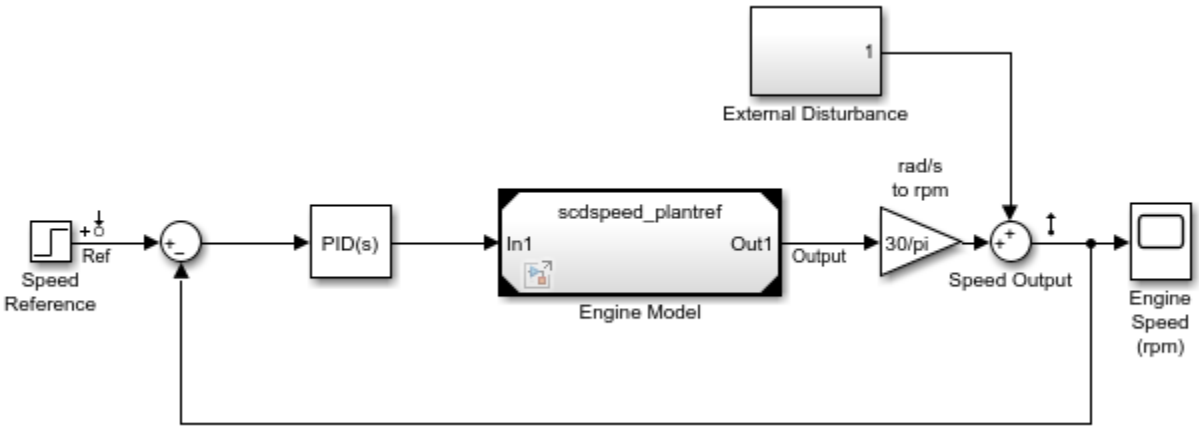
Option Name	Option Value
`'BlocksToHoldConstant'`	Block paths of time-varying source blocks to hold constant during frequency response estimation, specified as an array of `Simulink.BlockPath` objects. To identify time-varying source blocks that can interfere with frequency response estimation, use `frest.findSources`. **Default**: empty
`'UseParallel'`	Set to `'on'` to enable parallel computing for estimations with the `frestimate` command. **Default**: `'off'`
`'ParallelPathDependencies'`	A cell array of character vectors or string array that specifies the path dependencies required to execute the model to estimate. All the workers in the parallel pool must have access to the folders listed in `'ParallelPathDependencies'`. **Default**: empty

Examples

Identify and Disable Time-Varying Source Blocks

Open model.

```
mdl = 'scdspeed_ctrlloop';
open_system(mdl)
```



Convert referenced subsystem to normal mode.

```
set_param('scdspeed_ctrlloop/Engine Model','SimulationMode','Normal');
```

Obtain input/output points from the model and create sinestream input signal.

```
io = getlinio(mdl);
in = frest.Sinestream('Frequency',logspace(1,2,10),'NumPeriods',30,...
               'SettlingPeriods',25);
```

Identify time-varying sources in the model.

```
srcblks = frest.findSources(mdl)
```

```
srcblks =
  1x4 BlockPath array with properties:

    SubPath
    isLoadingModel
    isSavingModel
```

Create option set for estimation and specify the time-varying source blocks.

```
opts = frestimateOptions('BlocksToHoldConstant',srcblks);
```

Estimate the frequency response.

```
[sysest,simout] = frestimate(mdl,io,in,opts);
```

Specify Model Path Dependencies for Parallel Computing

To demonstrate a dependency on a file that is not in the current working folder, move the model files to a temporary folder and return the path to that folder. The `pathdepSetup` helper function also adds the temporary folder to the MATLAB® search path.

```
tempPath = pathdepSetup;
```

Open the Simulink® model.

```
mdl = 'scdpathdep';
open_system(mdl)
```

Obtain the model dependency path.

```
dirs = frest.findDepend(mdl)
```

```
dirs = 1×1 cell array
    {'C:/myTempFiles/tpd02d55f5_8b4c_489e_938c_ea004b9c771d'}
```

The resulting path is on the local drive `C:/`.

If you are using remote workers, 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 remote workers.

```
dirs = regexprep(dirs,'C:/','\\\\hostname\\C$\\')
```

Enable parallel computing and specify the model path dependencies.

```
options = frestimateOptions(...
    'UseParallel','on',...
    'ParallelPathDependencies',dirs);
```

You can now use these options for frequency response estimation using parallel computing.

```
io = getlinio(mdl);
in = frest.Sinestream('SimulationOrder','OneAtATime');
frd = frestimate(mdl,io,in,options);
```

After estimating the frequency response, you can close the model.

```
bdclose(mdl)
```

Return the model files to the current working folder and remove the temporary folder from the path.

```
pathdepCleanup(tempPath)
```

Alternatives

You can enable parallel computing for all models with no path dependencies. To do so, in the MATLAB preferences dialog box, click **Simulink Control Design**. Then, select the **Use the parallel pool when you use the "frestimate" command** option. This global setting persists from session to session until you change this option.

When you select this option and use the `frestimate` command, you do not need to provide an `frestimateOptions` 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`

Introduced in R2010a

fselect

Extract sinestream signal at specified frequencies

Syntax

```
input2 = fselect(input,fmin,fmax)
input2 = fselect(input,index)
```

Description

`input2 = fselect(input,fmin,fmax)` extracts a portion of the sinestream input signal `input` in the frequency range between `fmin` and `fmax`. Specify `fmin` and `fmax` in the same frequency units as the sinestream signal.

`input2 = fselect(input,index)` extracts a sinestream signal at specific frequencies, specified by the vector of indices `index`.

Examples

Extract the second frequency in a sinestream signal:

```
% Create the input signal
input = frest.Sinestream('Frequency',[1 2.5 5],...
                         'Amplitude',[1 2 1.5],...
                         'NumPeriods',[4 6 12],...
                         'RampPeriods',[0 2 6]);

% Extract a sinestream signal for the second frequency
input2 = fselect(input,2)

% Plot the extracted input signal
plot(input2)
```

See Also

`frestimate` | `frest.Sinestream` | `fdel`

Topics

“Time Response Not at Steady State” on page 5-43

Introduced in R2010a

generateTimeseries

Generate time-domain data for input signal

Syntax

```
ts = generateTimeseries(input)
```

Description

`ts = generateTimeseries(input)` creates a MATLAB `timeseries` object `ts` from the input signal `input`. `input` can be a sinestream, chirp, or random signal. For chirp and random signals, that time vector of `ts` has equally spaced time values, ranging from 0 to `Ts(NumSamples-1)`.

Examples

Create `timeseries` object for chirp signal:

```
input = frest.Chirp('Amplitude',1e-3,'FreqRange',...
                    [10 500],'NumSamples',20000);
ts = generateTimeseries(input)
```

See Also

`frestimate` | `frest.Sinestream` | `frest.Chirp` | `frest.Random`

Introduced in R2009b

get

Properties of linearization I/Os and operating points

Syntax

```
get(ob)
get(ob,'PropertyName')
```

Description

`get(ob)` displays all properties and corresponding values of the object, `ob`, which can be a linearization I/O object, an operating point object, or an operating point specification object. Create `ob` using `findop`, `getlinio`, `linio`, `operpoint`, or `operspec`.

`get(ob,'PropertyName')` returns the value of the property, `PropertyName`, within the object, `ob`. The object, `ob`, can be a linearization I/O object, an operating point object, or an operating point specification object. Create `ob` using `findop`, `getlinio`, `linio`, `operpoint`, or `operspec`.

`ob.PropertyName` is an alternative notation for displaying the value of the property, `PropertyName`, of the object, `ob`. The object, `ob`, can be a linearization I/O object, an operating point object, or an operating point specification object. Create `ob` using `findop`, `getlinio`, `linio`, `operpoint`, or `operspec`.

Examples

Create an operating point object, `op`, for the Simulink model, `magball`.

```
op=operpoint('magball');
```

Get a list of all object properties using the `get` function with the object name as the only input.

```
get(op)
```

This returns the properties of `op` and their current values.

```
     Model: 'magball'
     States: [5x1 opcond.StatePoint]
     Inputs: [0x1 double]
       Time: 0
    Version: 2
```

To view the value of a particular property of `op`, supply the property name as an argument to `get`. For example, to view the name of the model associated with the operating point object, type:

```
V=get(op,'Model')
```

which returns

```
V =
magball
```

Because `op` is a structure, you can also view any properties or fields using dot-notation, as in this example.

```
W=op.States
```

This notation returns a vector of objects containing information about the states in the operating point.

```
(1.) magball/Controller/PID Controller/Filter
      x: 0
(2.) magball/Controller/PID Controller/Integrator
      x: 14
(3.) magball/Magnetic Ball Plant/Current
      x: 7
(4.) magball/Magnetic Ball Plant/dhdt
      x: 0
(5.) magball/Magnetic Ball Plant/height
      x: 0.05
```

Use `get` to view details of `W`. For example:

```
get(W(2),'x')
```

returns

```
ans =

   14.0071
```

See Also

`findop` | `getlinio` | `linio` | `operpoint` | `operspec` | `set`

Introduced before R2006a

getBlockInfo

Package: `linearize.advisor`

Obtain diagnostic information for block linearizations

Syntax

```
blockInfo = getBlockInfo(advisor)
blockInfo = getBlockInfo(advisor,block)
blockInfo = getBlockInfo(advisor,index)
```

Description

When you linearize a Simulink model, you can create a `LinearizationAdvisor` object that contains diagnostic information about individual block linearizations. You can troubleshoot your linearization results by reviewing this diagnostic information. To access the diagnostic information, use the `getBlockInfo` function.

`blockInfo = getBlockInfo(advisor)` returns the diagnostic information for all blocks listed in the `LinearizationAdvisor` object, `advisor`.

`blockInfo = getBlockInfo(advisor,block)` returns diagnostic information for blocks with block paths specified in `block`.

`blockInfo = getBlockInfo(advisor,index)` returns diagnostic information for blocks with indices specified in `index`.

Examples

Obtain Diagnostics for Potentially Problematic Blocks

Load Simulink model.

```
mdl = 'scdpendulum';
load_system(mdl)
```

Linearize the model and obtain `LinearizationAdvisor` object.

```
io = getlinio(mdl);
opt = linearizeOptions('StoreAdvisor',true);
[linsys,~,info] = linearize(mdl,io,opt);
advisor = info.Advisor;
```

Find blocks that are potentially problematic for linearization.

```
blocks = advise(advisor);
```

Obtain diagnostics for these blocks.

```
diags = getBlockInfo(blocks)
```

```
diags =
Linearization Diagnostics for the Blocks:

Block Info:
-----------
Index   BlockPath                                          Is On Path   Contributes To Lineariza
1.      scdpendulum/pendulum/Saturation                    Yes          No
2.      scdpendulum/angle_wrap/Trigonometric Function1     Yes          No
3.      scdpendulum/pendulum/Trigonometric Function        Yes          No
```

Obtain Diagnostics Using Block Names

Load Simulink model.

```
mdl = 'scdpendulum';
load_system(mdl)
```

Linearize the model and obtain `LinearizationAdvisor` object.

```
io = getlinio(mdl);
opt = linearizeOptions('StoreAdvisor',true);
[linsys,~,info] = linearize(mdl,io,opt);
advisor = info.Advisor;
```

Obtain diagnostic information for the saturation block.

```
satDiag = getBlockInfo(advisor,'scdpendulum/pendulum/Saturation')
```

```
satDiag =
Linearization Diagnostics for scdpendulum/pendulum/Saturation with properties:

                     IsOnPath: 'Yes'
    ContributesToLinearization: 'No'
           LinearizationMethod: 'Exact'
                 Linearization: [1x1 ss]
               OperatingPoint: [1x1 linearize.advisor.BlockOperatingPoint]
```

You can also obtain diagnostic information for multiple blocks at once. Obtain diagnostics for the sin blocks in the model.

```
sinBlocks = {'scdpendulum/pendulum/Trigonometric Function';
             'scdpendulum/angle_wrap/Trigonometric Function1'};

sinDiag = getBlockInfo(advisor,sinBlocks)
```

```
sinDiag =
Linearization Diagnostics for the Blocks:

Block Info:
-----------
Index   BlockPath                                          Is On Path   Contributes To Lineariza
1.      scdpendulum/angle_wrap/Trigonometric Function1     Yes          No
2.      scdpendulum/pendulum/Trigonometric Function        Yes          No
```

Obtain Diagnostics Using Indices

Load Simulink model.

```
mdl = 'scdpendulum';
load_system(mdl)
```

Linearize the model and obtain `LinearizationAdvisor` object.

```
io = getlinio(mdl);
opt = linearizeOptions('StoreAdvisor',true);
[linsys,~,info] = linearize(mdl,io,opt);
advisor = info.Advisor;
```

Obtain diagnostic information for the first element of `advisor.BlockDiagnostics`.

```
diag = getBlockInfo(advisor,1)
```

```
diag =
Linearization Diagnostics for scdpendulum/pendulum/Saturation with properties:

                      IsOnPath: 'Yes'
    ContributesToLinearization: 'No'
           LinearizationMethod: 'Exact'
                 Linearization: [1x1 ss]
               OperatingPoint: [1x1 linearize.advisor.BlockOperatingPoint]
```

You can also obtain diagnostics for multiple blocks. For example, obtain diagnostics for the second and third blocks listed in `advisor`.

```
diags = getBlockInfo(advisor,[2 3])
```

```
diags =
Linearization Diagnostics for the Blocks:

Block Info:
-----------
Index   BlockPath                                              Is On Path   Contributes To Lineariza
1.      scdpendulum/pendulum/Integrator, Second-Order          Yes          No
2.      scdpendulum/angle_wrap/Trigonometric Function1         Yes          No
```

Obtain Diagnostics for Blocks in Subsystem

Load Simulink model.

```
mdl = 'scdpendulum';
load_system(mdl)
```

Linearize the model and obtain `LinearizationAdvisor` object.

```
io = getlinio(mdl);
opt = linearizeOptions('StoreAdvisor',true);
[linsys,~,info] = linearize(mdl,io,opt);
advisor = info.Advisor;
```

Obtain block paths of linearized blocks.

```
paths = getBlockPaths(advisor);
```

Create logical array indicating which blocks are in the `angle_wrap` subsystem.

```
index = contains(paths,'angle_wrap');
```

Obtain diagnostic information for these blocks.

```
diags = getBlockInfo(advisor,index)
```

```
diags =
Linearization Diagnostics for the Blocks:

Block Info:
-----------
Index   BlockPath                                          Is On Path   Contributes To Lineariza
1.      scdpendulum/angle_wrap/Trigonometric Function1     Yes          No
2.      scdpendulum/angle_wrap/Trigonometric Function2     Yes          No
3.      scdpendulum/angle_wrap/Trigonometric Function      Yes          No
```

Input Arguments

`advisor` — Diagnostic information for block linearizations

`LinearizationAdvisor` object | array of `LinearizationAdvisor` objects

Diagnostic information for block linearizations, specified as a `LinearizationAdvisor` object or an array of `LinearizationAdvisor` objects.

`block` — Block paths

character vector | cell array of character vectors

Block paths in Simulink model, specified as one of the following:

- Character vector — Obtain diagnostic information for a single block.
- Cell array of character vectors — Obtain diagnostic information for multiple blocks.

`index` — Block indices

positive integer | array of positive integers | boolean array

Block indices, specified as one of the following:

- Positive integer — Obtain diagnostic information for the specified element of `Advisor.BlockDiagnostics`
- Array of positive integers — Obtain diagnostic information for multiple elements of `Advisor.BlockDiagnostics`.
- Boolean array — For each element of `index` that is `true`, return the diagnostics for the corresponding element of `Advisor.BlockDiagnostics`.

Output Arguments

`blockInfo` — Diagnostic information for block linearizations

`BlockDiagnostic` object | vector of `BlockDiagnostic` objects | cell array

Diagnostic information for block linearizations indicated by `index`, returned as a `BlockDiagnostic` object or vector of `BlockDiagnostic` objects if `advisor` is a single `LinearizationAdvisor` object.

If `advisor` is an array of `LinearizationAdvisor` objects, then `blockInfo` is a cell array with the same dimensions as `advisor` in which each element is a vector of `BlockDiagnostic` objects.

See Also

Objects
`LinearizationAdvisor`

Functions
`advise` | `find` | `getBlockPaths`

Topics
“Troubleshoot Linearization Results at Command Line” on page 4-28

Introduced in R2017b

getBlockPaths

Package: `linearize.advisor`

Obtain list of blocks in `LinearizationAdvisor` object

Syntax

```
blocks = getBlockPaths(advisor)
```

Description

When you linearize a Simulink model, you can create a `LinearizationAdvisor` object that contains diagnostic information about individual block linearizations, which you can use for troubleshooting linearization results. To obtain a list of the blocks in the `LinearizationAdvisor` object, use the `getBlockPaths` function.

`blocks = getBlockPaths(advisor)` returns a list of block paths for the blocks in the `LinearizationAdvisor` object `advisor`.

Examples

Obtain List of Numerically Perturbed Blocks

Load Simulink model.

```
mdl = 'scdspeed';
load_system(mdl)
```

Linearize model and obtain `LinearizationAdvisor` object.

```
opts = linearizeOptions('StoreAdvisor',true);
io(1) = linio('scdspeed/throttle (degrees)',1,'input');
io(2) = linio('scdspeed/rad//s to rpm',1,'output');
[sys,op,info] = linearize(mdl,io,opts);
advisor = info.Advisor;
```

Find all blocks in linearization results that are numerically perturbed.

```
perturbed = find(advisor,linqueryIsNumericallyPerturbed);
```

Obtain list of numerically perturbed blocks.

```
blocks = getBlockPaths(perturbed)
```

```
blocks = 6x1 cell
    {'scdspeed/Throttle & Manifold/Intake Manifold/Convert to mass charge'}
    {'scdspeed/Combustion/Torque Gen'                                     }
    {'scdspeed/Combustion/Torque Gen2'                                    }
    {'scdspeed/Throttle & Manifold/Intake Manifold/Pumping1'              }
    {'scdspeed/Throttle & Manifold/Throttle/f(theta)'                     }
    {'scdspeed/Throttle & Manifold/Throttle/g(pratio)'                    }
```

Input Arguments

`advisor` — Diagnostic information for block linearizations

`LinearizationAdvisor` object | array of `LinearizationAdvisor` objects

Diagnostic information for block linearizations, specified as a `LinearizationAdvisor` object or an array of `LinearizationAdvisor` objects.

Output Arguments

`blocks` — Block paths

cell array of character vectors | cell array

Block paths for blocks in `advisor`, returned as a cell array of character vectors if `advisor` is a single `LinearizationAdvisor` object. If `advisor` is an array of `LinearizationAdvisor` objects, then `blocks` is a cell array with the same dimensions as `advisor` in which each element is a cell array of character vectors.

See Also

Objects
`LinearizationAdvisor`

Functions
`advise` | `getBlockInfo`

Topics
“Troubleshoot Linearization Results at Command Line” on page 4-28

Introduced in R2017b

getInputIndex

Get index of an input element of an operating point specification

Syntax

```
index = getInputIndex(op,block)
index = getInputIndex(op,block,element)
```

Description

The `Inputs` property of an operating point specification is an array that contains trimming specifications for each model input. When defining a mapping function for customized trimming of Simulink models, you can use `getInputIndex` to obtain the index of an input specification based on the corresponding block path.

When trimming Simulink models using optimization-based search, some applications require additional flexibility in defining the optimization search parameters. For such systems, you can specify custom constraints and a custom objective function. For complex models, you can define a mapping that selects a subset of the model states, inputs, and outputs to pass to the custom constraint and objective functions. For more information, see “Compute Operating Points Using Custom Constraints and Objective Functions” on page 1-57.

`index = getInputIndex(op,block)` returns the index of the input specification that corresponds to `block` in the `Inputs` property of operating point specification `op`.

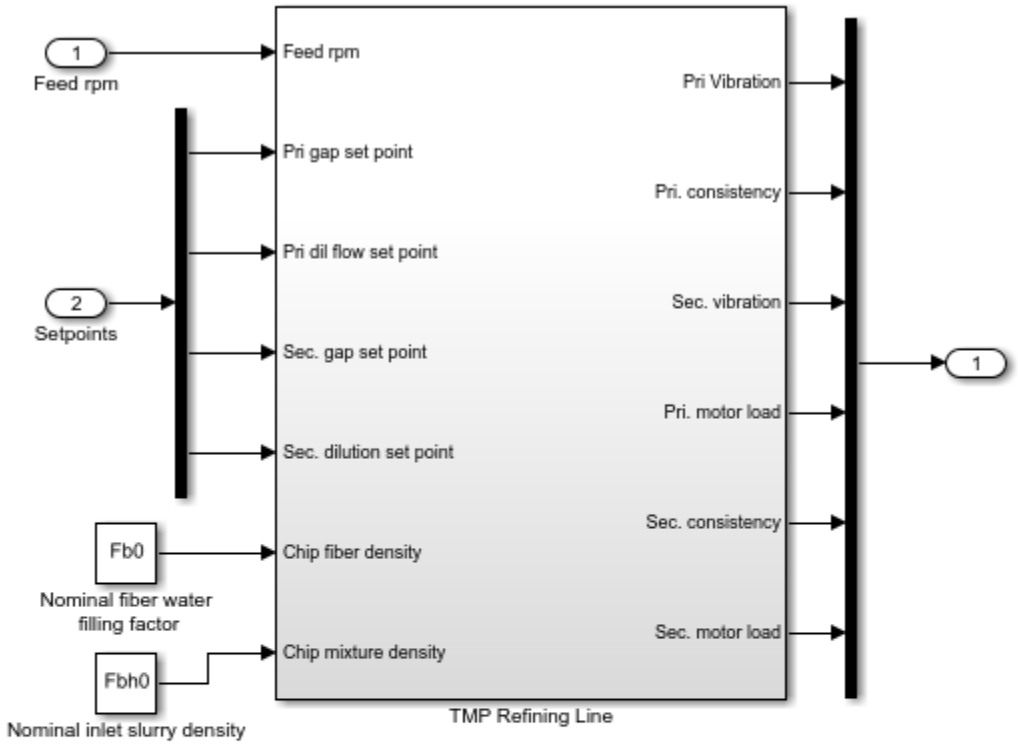
`index = getInputIndex(op,block,element)` returns the index of the specified `element` within an input specification for an input port that has a port width greater than `1`.

Examples

Get Input Index from Operating Point Specification

Open Simulink model.

```
mdl = 'scdtmpSetpoints';
open_system(mdl)
```



Copyright 2017-2021 The MathWorks, Inc.

Create an operating point specification object for the model.

```
opspec = operspec(mdl);
```

`opspec` contains specifications for the root-level input ports of the model.

```
opspec.Inputs
```

```
ans =

    <strong>u</strong>    <strong>Known</strong>    <strong>Min</strong>     <strong>Max</strong>
    <strong>_</strong>    <strong>_____</strong>    <strong>____</strong>    <strong>___</strong>

(1.) scdtmpSetpoints/Feed rpm
```

```
    0    false    -Inf    Inf
(2.) scdtmpSetpoints/Setpoints
    0    false    -Inf    Inf
    0    false    -Inf    Inf
    0    false    -Inf    Inf
    0    false    -Inf    Inf
```

Obtain the index of the specification in `opspec.Inputs` that corresponds to the Feed rpm input block.

```
index1 = getInputIndex(opspec,'scdtmpSetpoints/Feed rpm')
```

```
index1 =

     1     1
```

`index1(1)` is the index of the input specification object for the Feed rpm block in the `opspec.Inputs`. Since this input port is a scalar signal, `index1` has one row and `index1(2)` is `1`.

If an input port is a vector signal, you can obtain the indices for all of the elements in the corresponding input specification.

```
index2 = getInputIndex(opspec,'scdtmpSetpoints/Setpoints')
```

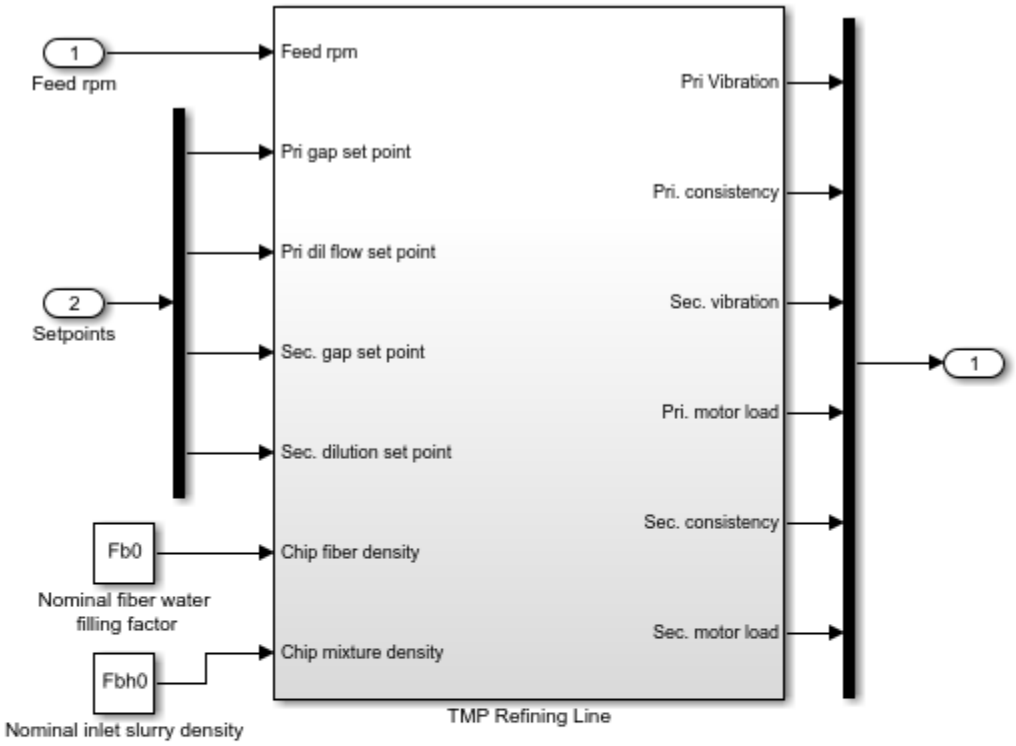
```
index2 =

     2     1
     2     2
     2     3
     2     4
```

Each row of `index2` is the index for one element of the Setpoints input vector.

Get Index of Specified Input Element of Operating Point Specification

Open Simulink model.

```
mdl = 'scdtmpSetpoints';
open_system(mdl)
```



Copyright 2017-2021 The MathWorks, Inc.

Create an operating point specification object for the model.

```
opspec = operspec(mdl);
```

`opspec` contains specifications for the root-level input ports of the model.

Obtain the index of the element that corresponds to the second signal in the Setpoints input vector.

```
index1 = getInputIndex(opspec,'scdtmpSetpoints/Setpoints',2)
```

```
index1 =

     2     2
```

You can also obtain the indices of multiple vector elements at the same time. For example, get the indices for the first and third elements of the Setpoints vector.

```
index2 = getInputIndex(opspec,'scdtmpSetpoints/Setpoints',[1 3])
```

```
index2 =

     2     1
     2     3
```

Input Arguments

op — Operating point specification or operating point

`OperatingSpec` object | `OperatingPoint` object | `OperatingReport` object

Operating point specification or operating point for a Simulink model, specified as an `OperatingSpec`, `OperatingPoint`, or `OperatingReport` object.

block — Block path

character vector | string

Block path that corresponds to an input specification in the `Inputs` property of `op`, specified as a character vector or string that contains the path of a root-level input of a Simulink model.

To see all the blocks that have input specifications, view the `Inputs` property of `op`.

```
op.Inputs
```

element — Input element index

positive integer | vector of positive integers

Input element index, specified as a positive integer less than or equal to the port width of the input specified by `block`, or as a vector of such integers. By default, if you do not specify `element`, `getInputIndex` returns the indices of all elements in the selected input specification. For an example, see “Get Index of Specified Input Element of Operating Point Specification” on page 18-93.

Output Arguments

index — Input index

2-element row vector | 2-column array

Input index, returned as a 2-element row vector when `element` is an integer, or a 2-column array when `element` is a vector. Each row of `index` contains the index for a single model input element.

The first column of `index` contains the index of the corresponding input specification in the `Inputs` property of `op`. The second column contains the element index within the input specification.

Using `index`, you can specify the input portion of a custom mapping for customized trimming of Simulink models. For more information, see the `CustomMappingFcn` property of `operspec`.

See Also

`getStateIndex` | `getOutputIndex` | `operspec` | `findop`

Topics
“Compute Operating Points Using Custom Constraints and Objective Functions” on page 1-57

Introduced in R2017a

getinputstruct

Obtain input values from operating point

Syntax

```
u = getinputstruct(op)
```

Description

`u = getinputstruct(op)` extracts a structure of input values from a specified operating point object. You can use the input structure to set initial input values for your Simulink model.

Examples

Initialize Model with Operating Point Values

Open the `scdplane` model and create an operating point. You can also compute a trimmed operating point or obtain an operating point snapshot.

```
mdl = 'scdplane';
open_system(mdl)
op = operpoint(mdl);
```

Extract the state values from the operating point.

```
xInitial = getstatestruct(op);
```

Extract the input values from the operating point.

```
uInitial = getinputstruct(op);
```

To view the values of the states or inputs within this structure, use dot notation. For example, view the input values.

```
uInitial.signals.values
```

```
ans = 0
```

Set the initial state values in the model.

```
set_param(mdl,'LoadInitialState','on','InitialState','xInitial')
```

Set the initial input values in the model.

```
set_param(mdl,'LoadExternalInput','on','ExternalInput','uInitial')
```

Input Arguments

op — Operating point

`OperatingPoint` object | `OperatingSpec` object | `OperatingReport` object | array

Operating point for a Simulink model, specified as an `OperatingPoint`, `OperatingSpec`, or `OperatingReport` object. You can also specify a homogeneous array of any of these objects.

Output Arguments

`u` — Input values
structure | structure array

Input values, returned as a structure with the following fields.

- `signals` — Input values and information
- `time` — Simulation time for input values, returned as `0`.

If `op` is an array, `u` is returned as a structure array with the same dimensions as `op`.

See Also

`getstatestruct` | `operpoint`

Introduced before R2006a

getlinio

Obtain linear analysis points from Simulink model, Linear Analysis Plots block, or Model Verification block

Syntax

```
io = getlinio(mdl)
io = getlinio(blockpath)
```

Description

`io = getlinio(mdl)` returns the analysis points defined in the Simulink model `mdl`.

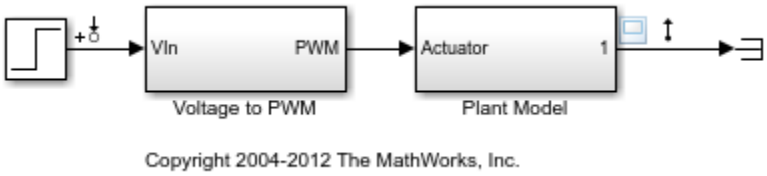
`io = getlinio(blockpath)` returns the analysis points defined for the specified Linear Analysis Plots block or Model Verification block in a Simulink model.

Examples

Obtain Analysis Points from Simulink Model

Open Simulink model.

```
mdl = 'scdpwm';
open_system(mdl)
```



This model contains the following linear analysis points:

- Input perturbation at the output of the Step block
- Output measurement at the output of the Plant Model block

Obtain the analysis points from the model.

```
io = getlinio(mdl)
```

```
2x1 vector of Linearization IOs:
--------------------------
1. Linearization input perturbation located at the following signal:
- Block: scdpwm/Step
- Port: 1
2. Linearization output measurement located at the following signal:
```

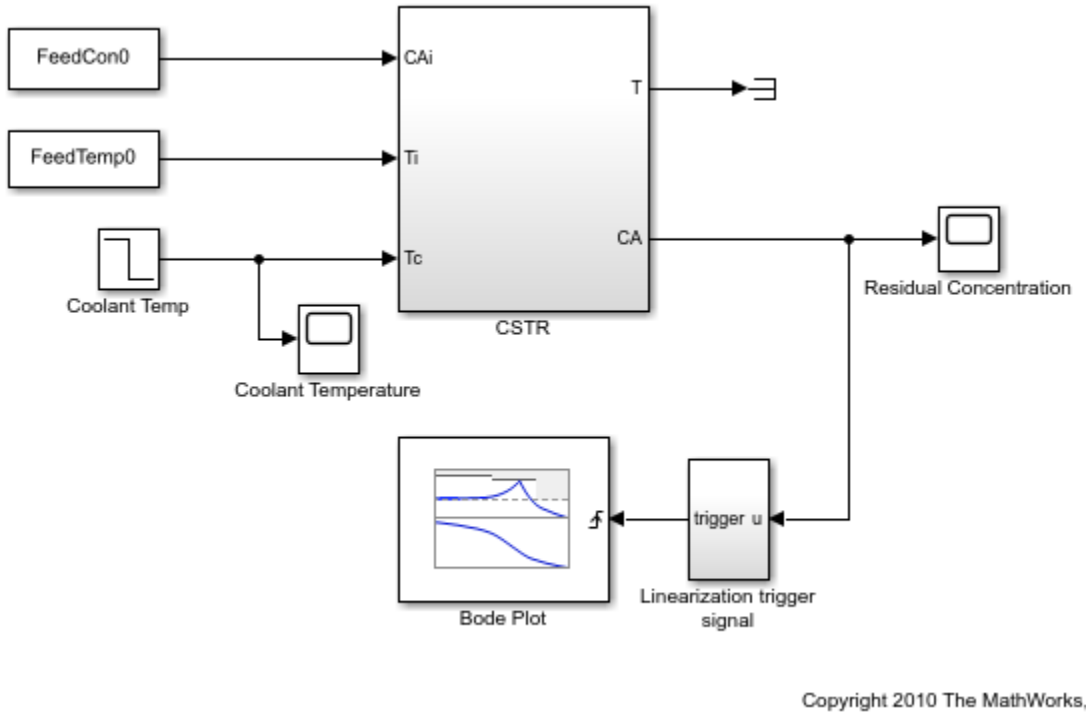
```
- Block: scdpwm/Plant Model
- Port: 1
```

You can use these analysis points for subsequent linearizations of the model using the `linearize` command or an `slLinearizer` interface.

Obtain Analysis Points from Linear Analysis Plots Block

Open Simulink model.

```
open_system('scdcstr')
```



This model contains a Bode Plot block that is configured with the following linear analysis points:

- Input perturbation at the output of the Coolant Temp block
- Output measurement at the `CA` output of the CSTR block

Obtain the analysis points from the Bode Plot block.

```
io = getlinio('scdcstr/Bode Plot')
```

```
2x1 vector of Linearization IOs:
--------------------------
1. Linearization input perturbation located at the following signal:
- Block: scdcstr/Coolant Temp
- Port: 1
2. Linearization output measurement located at the following signal:
```

```
- Block: scdcstr/CSTR
- Port: 2
```

Input Arguments

`mdl` — Simulink model name

character vector | string

Simulink model name, specified as a character vector or string. The model must be in the current working folder or on the MATLAB path.

If the model is not open or loaded into memory, `getlinio` loads the model into memory.

`blockpath` — Linear Analysis Plots block or Model Verification block

character vector | string

Linear Analysis Plots block or Model Verification block, specified as a character vector or string that contains its full block path. The model that contains the block must be in the current working folder or on the MATLAB path.

For more information on:

- Linear analysis plot blocks, see “Visualization During Simulation”.
- Model verification blocks, see “Model Verification”.

Output Arguments

`io` — Analysis point set

linearization I/O object | vector of linearization I/O objects

Analysis point set, returned as a linearization I/O object or a vector of linearization I/O objects. Use `io` to specify linearization inputs, outputs, and loop openings when using the `linearize` command. For more information, see “Specify Portion of Model to Linearize” on page 2-10.

Each analysis point has the following properties:

Property	Description
`Active`	Flag indicating whether to use the analysis point for linearization, specified as one of the following: • `'on'` — Use the analysis point for linearization. This value is the default option. • `'off'` — Do not use the analysis point for linearization. Use this option if you have an existing set of analysis points and you want to linearize a model with a subset of these points.
`Block`	Full block path of the block with which the analysis point is associated, specified as a character vector.
`PortNumber`	Output port with which the analysis point is associated, specified as an integer.

Property	Description
`Type`	Analysis point type, specified as one of the following: • `'input'` — Input perturbation • `'output'` — Output measurement • `'loopbreak'` — Loop break • `'openinput'` — Open-loop input • `'openoutput'` — Open-loop output • `'looptransfer'` — Loop transfer • `'sensitivity'` — Sensitivity • `'compsensitivity'` — Complementary sensitivity For more information on analysis point types, see “Specify Portion of Model to Linearize” on page 2-10.
`BusElement`	Bus element name with which the analysis point is associated, specified as a character vector or `''` if the analysis point is not a bus element.
`Description`	User-specified description of the analysis point, which you can set for convenience, specified as a character vector.

See Also

`linio` | `setlinio` | `linearize`

Topics

“Specify Portion of Model to Linearize” on page 2-10

Introduced before R2006a

getlinplant

Compute open-loop plant model from Simulink diagram

Syntax

```
[sysp,sysc] = getlinplant(block,op)
[sysp,sysc] = getlinplant(block,op,options)
```

Description

`[sysp,sysc] = getlinplant(block,op)` Computes the open-loop plant seen by a Simulink block labeled `block` (where `block` specifies the full path to the block). The plant model, `sysp`, and linearized block, `sysc`, are linearized at the operating point `op`.

`[sysp,sysc] = getlinplant(block,op,options)` Computes the open-loop plant seen by a Simulink block labeled `block`, using the linearization options specified in `options`.

Examples

To compute the open-loop model seen by the Controller block in the Simulink model `magball`, first create an operating point object using the function `findop`. In this case, you find the operating point from simulation of the model.

```
magball
op=findop('magball',20);
```

Next, compute the open-loop model seen by the block `magball/Controller`, with the `getlinplant` function.

```
[sysp,sysc]=getlinplant('magball/Controller',op)
```

The output variable `sysp` gives the open-loop plant model as follows:

```
a =
            Current     dhdt   height
   Current     -100        0        0
   dhdt      -2.801        0    196.2
   height         0        1        0

b =
            Controller
   Current          50
   dhdt              0
   height            0

c =
          Current     dhdt   height
   Sum2         0        0       -1

d =
          Controller
   Sum2           0
```

```
Continuous-time model.
```

See Also

findop | linearizeOptions | operpoint | operspec

Introduced before R2006a

getOffsetsForLPV

Extract LPV offsets from linearization results

Syntax

```
offsets = getOffsetsForLPV(info)
```

Description

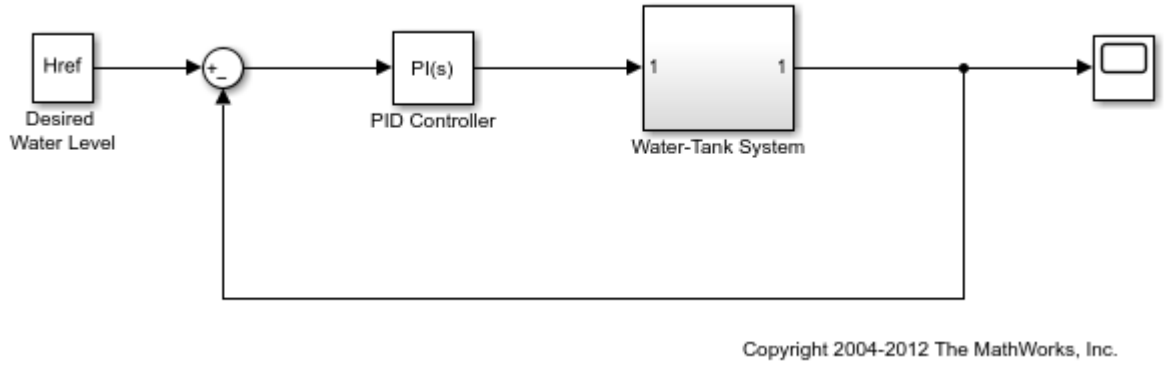
`offsets = getOffsetsForLPV(info)` extracts linearization offsets from `info` and converts them to the array format supported by the LPV System block.

Examples

Extract LPV Offsets from Linearization Results

Open the Simulink model.

```
model = 'watertank';
open_system(model)
```



Specify linearization I/Os.

```
io(1) = linio('watertank/Desired  Water Level',1,'input');
io(2) = linio('watertank/Water-Tank System',1,'output');
```

Vary plant parameters `A` and `b`, and create a 3-by-4 parameter grid.

```
[A_grid,b_grid] = ndgrid(linspace(0.9*A,1.1*A,3),linspace(0.9*b,1.1*b,4));
params(1).Name = 'A';
params(1).Value = A_grid;
params(2).Name = 'b';
params(2).Value = b_grid;
```

Create a linearization option set, setting the `StoreOffsets` option to `true`.

```
opt = linearizeOptions('StoreOffsets',true);
```

Linearize the model using the specified parameter grid, and return the linearization offsets in the `info` structure.

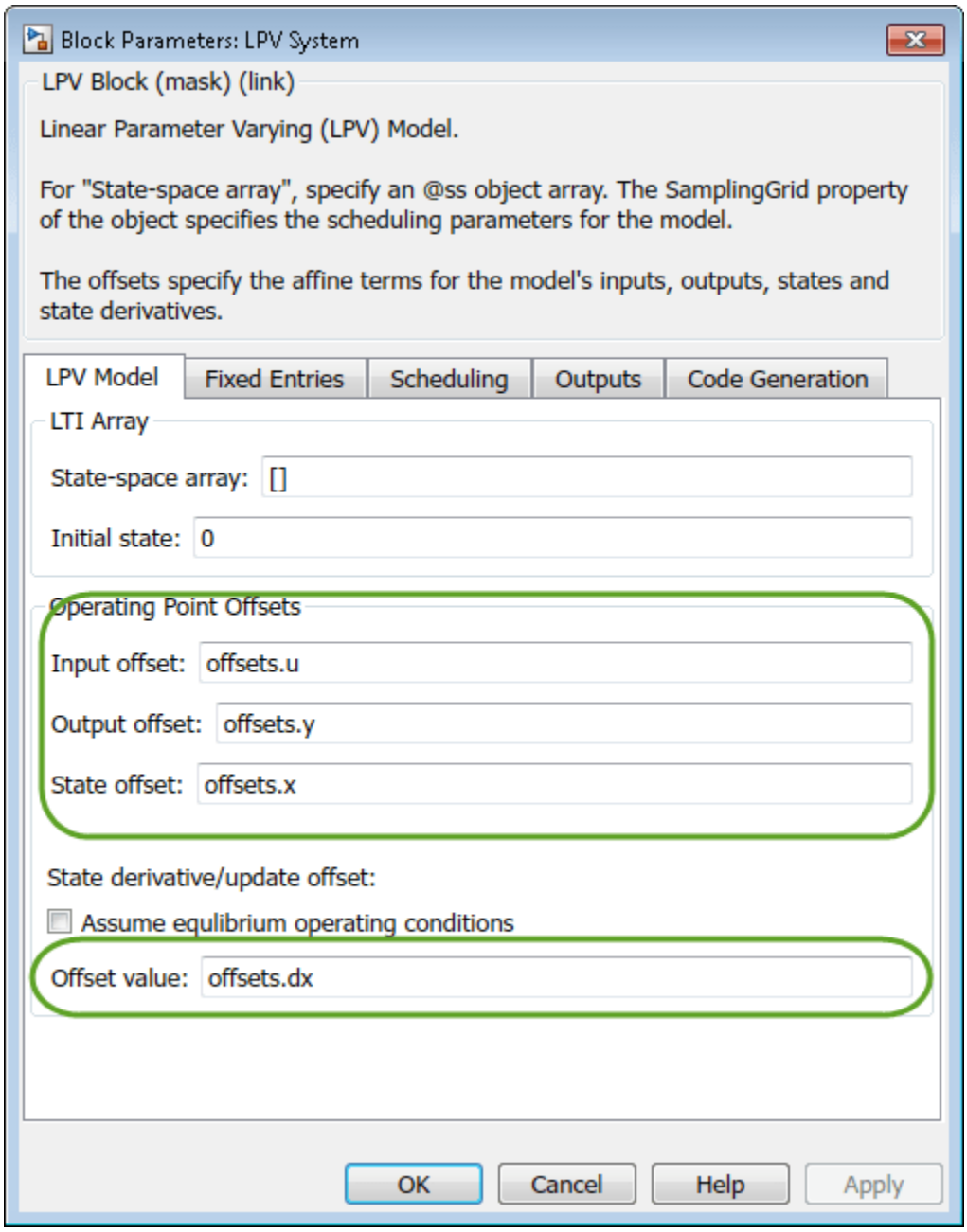
```
[sys,op,info] = linearize('watertank',io,params,opt);
```

Extract the linearization offsets.

```
offsets = getOffsetsForLPV(info)
```

```
offsets =

  struct with fields:

     x: [2x1x3x4 double]
     y: [1x1x3x4 double]
     u: [1x1x3x4 double]
    dx: [2x1x3x4 double]
```

To configure an LPV System block, use the fields from `offsets` directly.



Input Arguments

`info` — Linearization information

structure

Linearization information returned by exact linearization commands, specified as a structure. This structure has an `Offsets` field that contains an N_1-by-`...`-by-N_m array of structures, where N_1 to N_m are the dimensions of the operating point array or parameter grid used for linearization. Each structure in `info.Offsets` contains offset information that corresponds to a specific operating point.

You can store and obtain linearization offsets when you linearize your model using one of the following commands:

- `linearize`
- `getIOTransfer`
- `getLoopTransfer`
- `getSensitivity`
- `getCompSensitivity`

For example:

```
opt = linearizeOptions('StoreOffsets',true);
[sys,op,info] = linearize(mdl,io,params,opt);
```

You can then extract the offset information using `getOffsetsForLPV`.

```
offsets = getOffsetsForLPV(info);
```

Output Arguments

`offsets` — Linearization offsets

structure

Linearization offsets corresponding to the operating points at which the model was linearized, returned as a structure with the following fields:

Field	Description
`x`	State offsets used for linearization, returned as an n_x-by-1-by-N_1-by-...-by-N_m array, where n_x is the number of states in the linearized system.
`y`	Output offsets used for linearization, returned as an n_y-by-1-by-N_1-by-...-by-N_m array, where n_y is the number of outputs in the linearized system.
`u`	Input offsets used for linearization, returned as an n_u-by-1-by-N_1-by-...-by-N_m array, where n_u is the number of inputs in the linearized system.
`dx`	Derivative offsets for continuous time systems, or updated state values for discrete-time systems, returned as an n_x-by-1-by-N_1-by-...-by-N_m array.

For instance, suppose that your model has three inputs, two outputs, and four states. If you linearize your model using a 5-by-6 array of operating points, `offsets` contains arrays with the following dimensions:

- `offsets.x` — 4-by-1-by-5-by-6
- `offsets.y` — 2-by-1-by-5-by-6
- `offsets.u` — 3-by-1-by-5-by-6
- `offsets.dx` — 4-by-1-by-5-by-6

To configure an LPV System block, you can use the fields of `offsets` directly. For an example, see “Approximate Nonlinear Behavior Using Array of LTI Systems” on page 3-70.

See Also

Blocks
LPV System

Functions
`linearize` | `getIOTransfer` | `getLoopTransfer` | `getSensitivity` | `getCompSensitivity`

Topics
“Linear Parameter-Varying Models”
“Approximate Nonlinear Behavior Using Array of LTI Systems” on page 3-70

Introduced in R2016b

getOutputIndex

Get index of an output element of an operating point specification

Syntax

```
index = getOutputIndex(op,block)
index = getOutputIndex(op,block,port)
index = getOutputIndex(op,block,port,element)
```

Description

The `Outputs` property of an operating point specification is an array that contains trimming specifications for each model output. When defining a mapping function for customized trimming of Simulink models, you can use `getOutputIndex` to obtain the index of an output specification based on the corresponding block path.

When trimming Simulink models using optimization-based search, some applications require additional flexibility in defining the optimization search parameters. For such systems, you can specify custom constraints and a custom objective function. For complex models, you can define a mapping that selects a subset of the model states, inputs, and outputs to pass to the custom constraint and objective functions. For more information, see “Compute Operating Points Using Custom Constraints and Objective Functions” on page 1-57.

`index = getOutputIndex(op,block)` returns the index of the output specification that corresponds to `block` in the `Outputs` property of operating point specification `op`.

`index = getOutputIndex(op,block,port)` returns the index of the output specification that corresponds to the trim output constraint added to the specified output `port` of the specified `block`.

Use this syntax when the `Outputs` property of `op` contains trim output constraints for more than one signal originating from the same block.

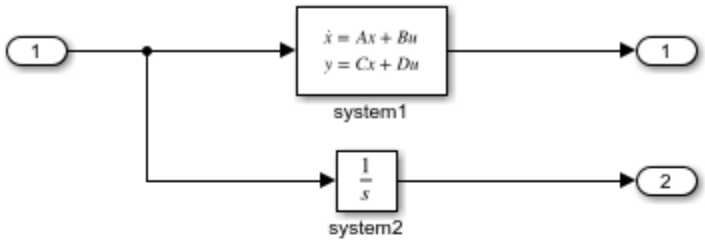
`index = getOutputIndex(op,block,port,element)` returns the index of the specified `element` within an output specification for an output with multiple elements.

Examples

Get Output Index from Operating Point Specification

Open Simulink model.

```
mdl = 'scdindex1';
open_system(mdl)
```



Copyright 2017-2021 The MathWorks, Inc.

Create an operating point specification for model.

```
opspec = operspec(mdl);
```

`opspec` contains an array of output specifications for the model.

```
opspec.Outputs
```

```
ans =

    <strong>y</strong>    <strong>Known</strong>    <strong>Min</strong>     <strong>Max</strong>
    <strong>_</strong>    <strong>_____</strong>    <strong>____</strong>    <strong>___</strong>

(1.) scdindex1/Out1
    0    false    -Inf    Inf
(2.) scdindex1/Out2
    0    false    -Inf    Inf
```

Get the index of the output specification for Out2.

```
idx = getOutputIndex(opspec,'scdindex1/Out2')
```

```
idx =
```

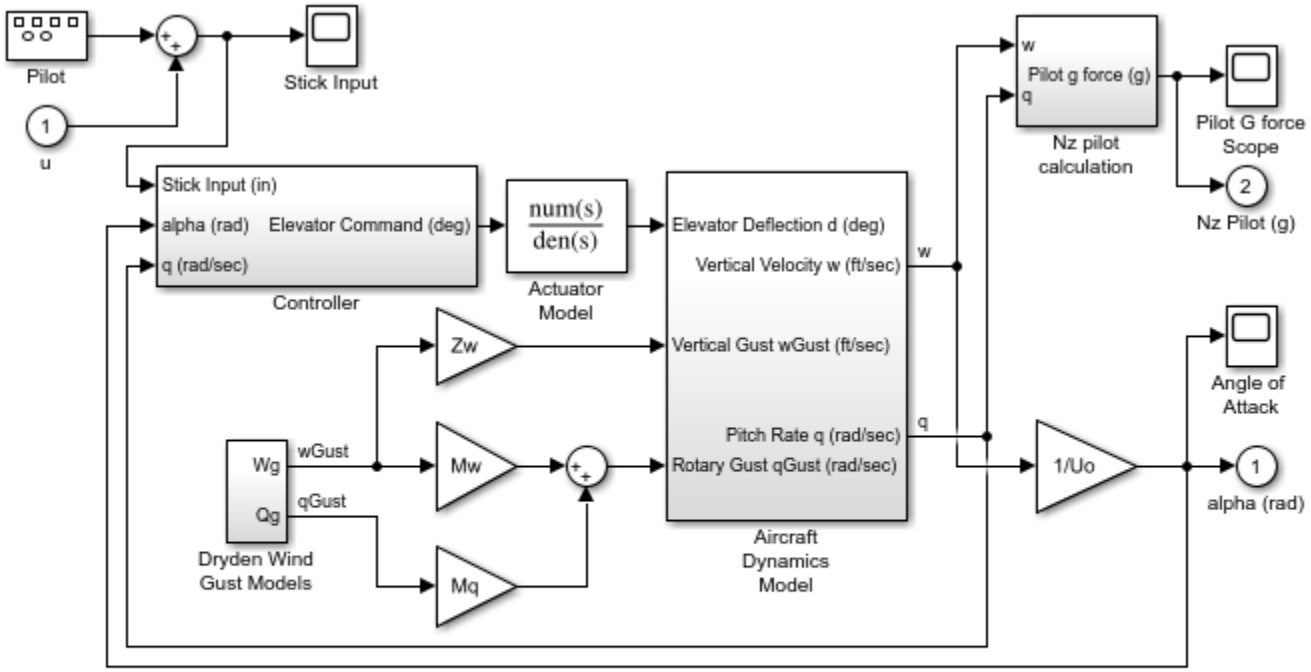
```
     2     1
```

The first column of `idx` contains the index of the output specification in `opspec.Outputs`. The second column contains the element index within the output specification. In this case, there is only one element in the output specification.

Get Index of Trim Output Specification Added To Signal

Open Simulink model.

```
mdl = 'scdplane';
open_system(mdl)
```



Copyright 1990-2012 The MathWorks, Inc.

Create an operating point specification for the model.

```
opspec = operspec(mdl);
```

In addition to root-level outputs of a model, the `opspec.Outputs` array contains specifications for trim constraints added to signals using the `addoutputspec` command.

Add an output specification to the signal originating from second output port of the Aircraft Dynamics Model block.

```
opspec = addoutputspec(opspec,'scdplane/Aircraft Dynamics Model',2);
```

View the output array of `opspec`.

```
opspec.Outputs
```

```
ans =

    <strong>y</strong>    <strong>Known</strong>    <strong>Min</strong>     <strong>Max</strong>
    <strong>_</strong>    <strong>_____</strong>    <strong>____</strong>    <strong>___</strong>

(1.) scdplane/alpha (rad)
    0    false    -Inf    Inf
(2.) scdplane/Nz Pilot (g)
    0    false    -Inf    Inf
(3.) scdplane/Aircraft Dynamics Model
    0    false    -Inf    Inf
```

Get the index of the added output specification. When there is an output specification for only one of the output ports of a given block, you do not need to specify the port number to get the output index.

```
index1 = getOutputIndex(opspec,'scdplane/Aircraft Dynamics Model')
```

```
index1 =

     3     1
```

Add an output specification to the signal originating from the first output of the same block.

```
opspec = addoutputspec(opspec,'scdplane/Aircraft Dynamics Model',1);
```

View the output array of `opspec`.

```
opspec.Outputs
```

```
ans =

    <strong>y</strong>    <strong>Known</strong>    <strong>Min</strong>     <strong>Max</strong>
    <strong>_</strong>    <strong>_____</strong>    <strong>____</strong>    <strong>___</strong>

(1.) scdplane/alpha (rad)
    0    false    -Inf    Inf
(2.) scdplane/Nz Pilot (g)
    0    false    -Inf    Inf
(3.) scdplane/Aircraft Dynamics Model
    0    false    -Inf    Inf
(4.) scdplane/Aircraft Dynamics Model
    0    false    -Inf    Inf
```

There are now two output specifications that correspond to the same block, one for each output port. Obtain the index for the output specification that corresponds with the output port 1 of the Aircraft Dynamics Model block.

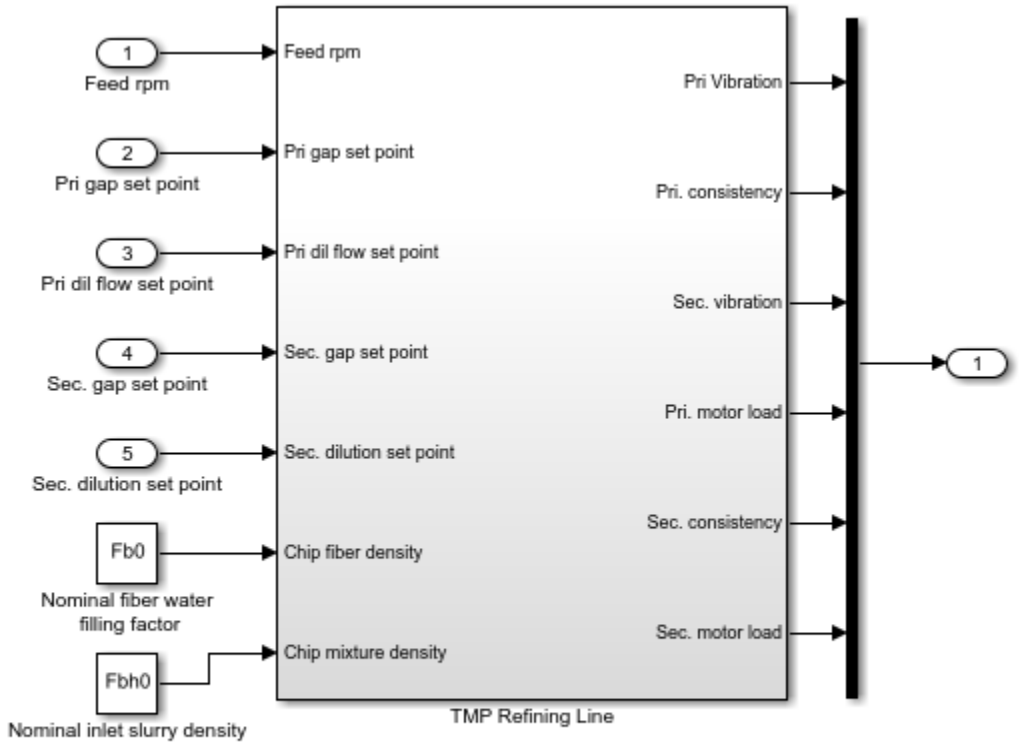
```
index2 = getOutputIndex(opspec,'scdplane/Aircraft Dynamics Model',1)
```

```
index2 =

     4     1
```

Get Output Indices for Output Specification with Multiple Elements

Open Simulink model.

```
mdl = 'scdtmp';
open_system(mdl)
```



Create an operating point specification object for the model.

```
opspec = operspec(mdl);
```

`opspec` contains an output specification for the output port Out1, which is a vector signal.

```
opspec.Outputs
```

```
ans =
```

```
    <strong>y</strong>    <strong>Known</strong>    <strong>Min</strong>     <strong>Max</strong>
    <strong>_</strong>    <strong>_____</strong>    <strong>____</strong>    <strong>___</strong>

(1.) scdtmp/Out1
    0    false    -Inf    Inf
    0    false    -Inf    Inf
    0    false    -Inf    Inf
    0    false    -Inf    Inf
    0    false    -Inf    Inf
    0    false    -Inf    Inf
```

Obtain the indices of all the elements of Out1.

```
index1 = getOutputIndex(opspec,'scdtmp/Out1')
```

```
index1 =

     1     1
     1     2
     1     3
     1     4
     1     5
     1     6
```

Each row of `index1` contains the index for one element of the vector signal in Out1. The first column is the index of the output specification object for the Out1 port in the `opsepc.Outputs`. The second column is the element index within the output specification.

You can also obtain the index for individual elements of an output specification, or a subset of elements. Get the index of element number 4 of Out1.

```
index2 = getOutputIndex(opspec,'scdtmp/Out1',[],4)
```

```
index2 =

     1     4
```

Get the indices of elements 2 and 3 of Out1.

```
index3 = getOutputIndex(opspec,'scdtmp/Out1',[],[2 3])
```

```
index3 =

     1     2
     1     3
```

Input Arguments

op — Operating point specification

`OperatingSpec` object | `OperatingReport` object

Operating point specification for a Simulink model, specified as an `OperatingSpec` or `OperatingReport`.

`block` — Block path
character vector | string

Block path that corresponds to an output specification in the `Outputs` property of `op`, specified as a character vector or string that contains the path of one of the following:

- Root-level output of the model.
- Source block for a signal in the model to which an output specification has been added. For more information on adding output specifications to a model, see `addoutputspec`.

To see all the blocks that have output specifications, view the `Outputs` property of `op`.

```
op.Outputs
```

`port` — Output port
integer in the range `[1,N]`

Output port, specified as an integer in the range `[1,N]`, where `N` is the number of output ports on the specified `block`. If `block` is a root-level output port, then `N` is `1`.

If you do not specify `port`, and there is one entry in the output array of `op` that corresponds to the specified `block`, then the default value of `port` is the port number of that entry. If there are multiple entries in the output array that correspond to the specified `block`, then the default value of `port` is the port number of the first entry. For an example, see “Get Index of Trim Output Specification Added To Signal” on page 18-112.

To view the port number of the `i`th entry in the output array of `op`, type:

```
op.Outputs(i).PortNumber
```

`element` — Output element index
`[1,M]` (default) | positive integer | vector of positive integers

Output element index, specified as a positive integer less than or equal to the port width of the output of the specified `block`, or a vector of such integers. By default, if you do not specify `element`, `getOutputIndex` returns the indices of all elements in the selected output specification. For an example, see “Get Output Indices for Output Specification with Multiple Elements” on page 18-114.

Output Arguments

`index` — Output index
2-element row vector | 2-column array

Output index, returned as a 2-element row vector when `element` is an integer, or a 2-column array when `element` is a vector. Each row of `index` contains the index for a single output element.

The first column of `index` contains the index of the corresponding output specification in the `Outputs` property of `op`. The second column contains the element index within the output specification.

Using `index`, you can specify the output portion of a custom mapping for customized trimming of Simulink models. For more information, see the `CustomMappingFcn` property of `operspec`.

See Also

`getStateIndex` | `getInputIndex` | `operspec` | `findop`

Topics

“Compute Operating Points Using Custom Constraints and Objective Functions” on page 1-57

Introduced in R2017a

getSimulationTime

Final time of simulation for frequency response estimation

Syntax

```
tfinal = getSimulationTime(input)
```

Description

`tfinal = getSimulationTime(input)` returns the final time of the Simulink simulation performed during frequency response estimation using the input signal `input`. Altering `input` to reduce the final simulation time can help reduce the time it takes to perform frequency response estimation.

Input Arguments

`input`

Input signal for frequency response estimation with the `frestimate` command.

The input signal `input` must be either:

- A sinestream input signal, created in the **Model Linearizer** or created with `frest.Sinestream`
- A chirp input signal, created in the **Model Linearizer** or created with `frest.Chirp`
- A random input signal, created in the **Model Linearizer** or created with `frest.Random`

Output Arguments

`tfinal`

Final time of simulation performed during frequency response estimation using the input signal `input`.

For example, the command `sysest = frestimate(mdl,io,input)` performs frequency response estimation on the Simulink model specified by `mdl` with the linearization I/O set `io`. The estimation uses the input signal `input`. The command `tfinal = getSimulationTime(input)` returns the simulation time at the end of the simulation performed by `frestimate`.

Examples

Retrieve Simulation Time for Frequency Response Estimation

Create a sinestream input signal.

```
input = frest.Sinestream('Amplitude',1e-3,...
                         'Frequency',logspace(1,3,50),...
                         'SamplesPerPeriod',40,'FreqUnits','Hz');
```

The sinestream signal `input` includes 50 frequencies spaced logarithmically between 10 Hz and 1000 Hz. Each frequency is sampled 40 times per period.

Calculate the final simulation time of an estimation using that signal.

```
tfinal = getSimulationTime(input)
```

```
tfinal = 4.4186
```

`tfinal` indicates that frequency response estimation of any model with this input signal would simulate the model for 4.4186 s.

See Also

`frestimate` | `frest.Sinestream` | `frest.Chirp` | `frest.Random`

Topics

“Sinestream Input Signals” on page 5-29
“Chirp Input Signals” on page 5-33
“Ways to Speed up Frequency Response Estimation” on page 5-68

Introduced in R2012a

getStateIndex

Get index of a state element of an operating point specification

Syntax

```
index = getStateIndex(op,name)
index = getStateIndex(op,name,element)
```

Description

The `States` property of an operating point specification is an array that contains trimming specifications for each model state. When defining a mapping function for customized trimming of Simulink models, you can use `getStateIndex` to obtain the index of a state specification based on the corresponding block path or state name.

When trimming Simulink models using optimization-based search, some applications require additional flexibility in defining the optimization search parameters. For such systems, you can specify custom constraints and a custom objective function. For complex models, you can define a mapping that selects a subset of the model states, inputs, and outputs to pass to the custom constraint and objective functions. For more information, see “Compute Operating Points Using Custom Constraints and Objective Functions” on page 1-57.

`index = getStateIndex(op,name)` returns the index of the state specification that corresponds to `name` in the `States` property of operating point specification `op`.

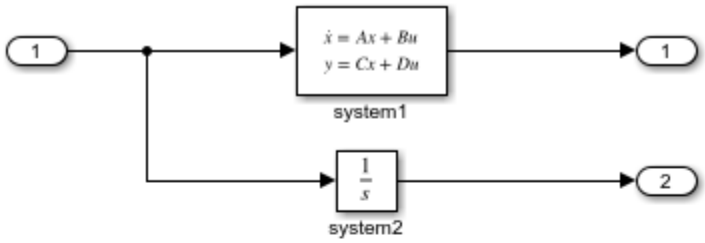
`index = getStateIndex(op,name,element)` returns the index of the specified `element` within a state specification for a block with multiple states.

Examples

Get State Index from Operating Point Specification

Open Simulink model.

```
mdl = 'scdindex1';
open_system(mdl)
```



Copyright 2017-2021 The MathWorks, Inc.

Create an operating point specification for model.

```
opspec = operspec(mdl);
```

`opspec` contains an array of state specifications for the model.

```
opspec.States
```

```
ans =

    <strong>x</strong>    <strong>Known</strong>    <strong>SteadyState</strong>    <strong>Min</
    <strong>_</strong>    <strong>_____</strong>    <strong>___________</strong>    <strong>___

(1.) scdindex1/system1
    0    false       true        -Inf    Inf    -Inf      Inf
    0    false       true        -Inf    Inf    -Inf      Inf
    0    false       true        -Inf    Inf    -Inf      Inf
(2.) scdindex1/system2
    0    false       true        -Inf    Inf    -Inf      Inf
```

Get the index of the state specification that corresponds to the system2 block.

```
index2 = getStateIndex(opspec,'scdindex1/system2')
```

```
index2 =

     2     1
```

`index2(1)` is the index of the state specification object for system2 in `opspec.States`. Since this block has a single state, `index2` has a single row and `index2(2)` is `1`.

If a block has multiple states, you can obtain the indices of all the states in the corresponding state specification.

```
index1 = getStateIndex(opspec,'scdindex1/system1')
```

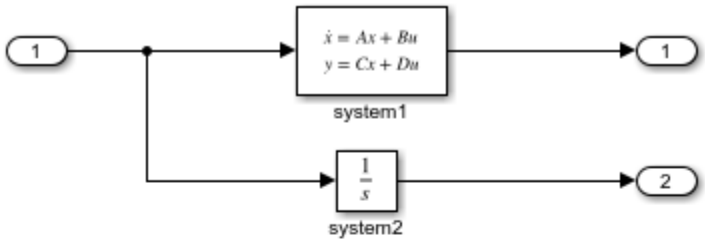
```
index1 =

     1     1
     1     2
     1     3
```

Each row of `index1` contains the index of one state in the system2 block. For each row, the first column contains the index of the state specification in `opspec.States`. The second column contains the index of each state element within the specification.

Get Index of Specified State Element of Operating Point Specification

Open Simulink model.

```
mdl = 'scdindex1';
open_system(mdl)
```



Copyright 2017-2021 The MathWorks, Inc.

Create an operating point specification for the model.

```
opspec = operspec(mdl);
```

If a block has multiple states, you can obtain the index of a specific state within the corresponding state specification by specifying the element index. For example, get the index for the second state in the specification for the system1 block.

```
index1 = getStateIndex(opspec,'scdindex1/system1',2)
```

```
index1 =

     1     2
```

You can also obtain the indices of a subset of the block states by specifying the element index as a vector. For example, get the indices for the first and third states in the specification for the system1 block.

```
index2 = getStateIndex(opspec,'scdindex1/system1',[1 3])
```

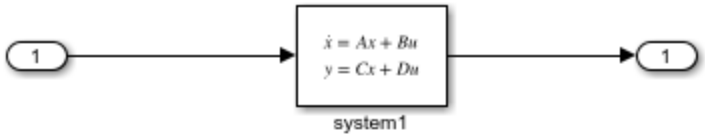
```
index2 =
```

```
     1     1
     1     3
```

Get Index of Named State from Operating Point Specification

Open Simulink model.

```
mdl = 'scdindex2';
open_system(mdl)
```



Copyright 2017-2021 The MathWorks, Inc.

The system1 block is a state-space system with three named states: `position`, `velocity`, and `acceleration`.

Create an operating point specification for the model.

```
opspec = operspec(mdl);
```

The `States` property of the operating point specification object contains one entry for each named state in system1.

```
opspec.States
```

```
ans =

    <strong>x</strong>    <strong>Known</strong>    <strong>SteadyState</strong>    <strong>Min</
    <strong>_</strong>    <strong>_____</strong>    <strong>_____________</strong>    <strong>____

(1.) position
    0    false       true        -Inf    Inf    -Inf      Inf
(2.) velocity
    0    false       true        -Inf    Inf    -Inf      Inf
(3.) acceleration
    0    false       true        -Inf    Inf    -Inf      Inf
```

To obtain the index of a state specification that corresponds to a named state within a block, specify the state name.

```
index1 = getStateIndex(opspec,'velocity')
```

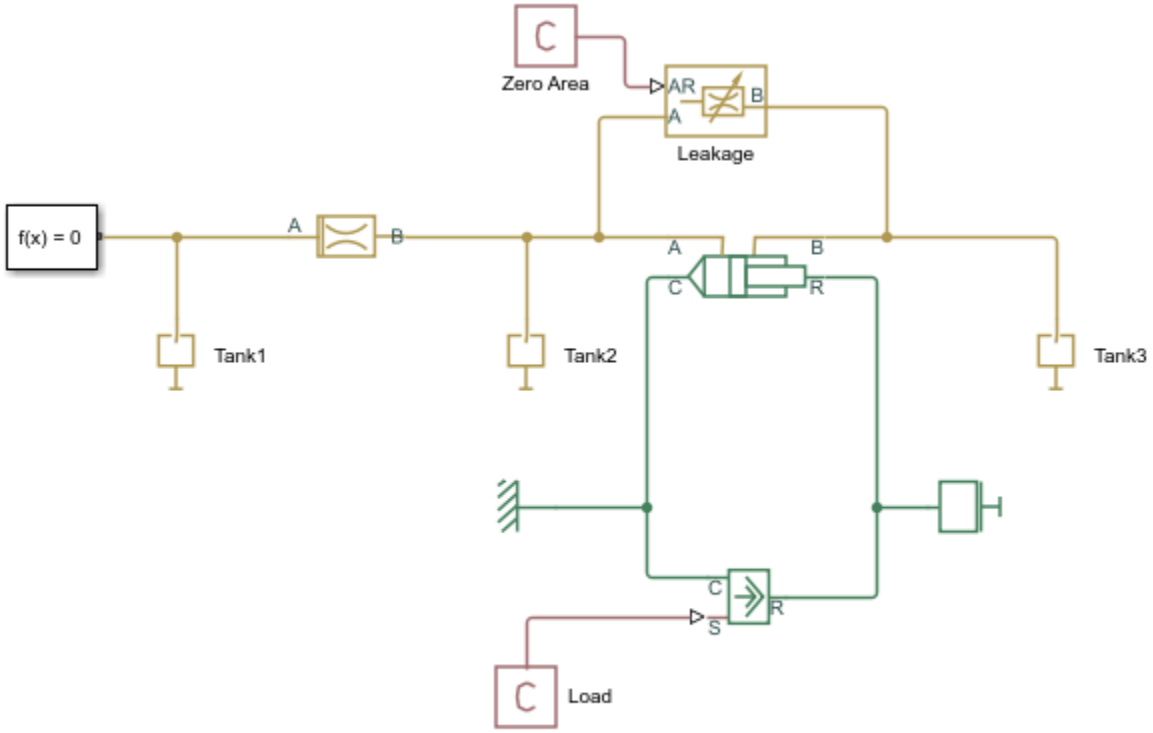
```
index1 =

     2     1
```

The first column of `index1` contains the index of the corresponding state specification in the `opspec.States` property. The second column is `1` for a named state.

Get Index of Simscape State from Operating Point Specification

Open model.

```
mdl = 'scdTanks_simscape';
open_system(mdl)
```



Copyright 2016-2021 The MathWorks, Inc.

Create an operating point specification for the model.

```
opspec = operspec(mdl);
```

The `States` property of the operating point specification object contains one state specification for each Simscape state in the model.

To obtain the index of a specification that corresponds to a Simscape state, specify the state name. For example, get the index of the pressure state of Tank3.

```
idx = getStateIndex(opspec,'scdTanks_simscape.Tank3.pressure')
```

```
idx =

    18     1
```

The first column of `idx` contains the index of the corresponding state specification in `opspec.States`. The second column is `1` for a Simscape state.

View the specification in `opspec.States` for this state.

```
opspec.States(idx(1))
```

```
ans =
```

```
    <strong>x</strong>    <strong>Known</strong>    <strong>SteadyState</strong>    <strong>Min</
    <strong>_</strong>    <strong>_____</strong>    <strong>____________</strong>    <strong>____

(1.) scdTanks_simscape.Tank3.pressure
    0    false       true        -Inf    Inf    -Inf      Inf
```

Input Arguments

op — Operating point specification or operating point

`OperatingSpec` object | `OperatingPoint` object | `OperatingReport` object

Operating point specification or operating point for a Simulink model, specified as an `OperatingSpec`, `OperatingPoint`, or `OperatingReport` object.

name — Block path or state name

character vector | string

Block path or state name that corresponds to a state specification in the `States` property of `op`, specified as a character vector or string that contains one of the following:

- Block path of a block in the Simulink model that contains unnamed states.
- Name of a named state in a Simulink or Simscape block.

To see all the states that have state specifications, view the `States` property of `op`.

```
op.States
```

element — State element index

positive integer | vector of positive integers

State element index, specified as a positive integer less than or equal to the number of state elements in the block or state specified by `name`, or a vector of such integers. By default, if you do not specify `element`, `getStateIndex` returns the indices of all elements in the selected state specification. For an example, see “Get Index of Specified State Element of Operating Point Specification” on page 18-122.

Output Arguments

index — State index

2-element row vector | 2-column array

State index, returned as a 2-element row vector when `element` is an integer, or a 2-column array when `element` is a vector. Each row of `index` contains the index for a single model state.

The first column of `index` contains the index of the corresponding state specification in the `States` property of `op`. The second column contains the element index within the state specification.

Using `index`, you can specify the state portion of a custom mapping for customized trimming of Simulink models. For more information, see the `CustomMappingFcn` property of `operspec`.

See Also

`getInputIndex` | `getOutputIndex` | `operspec` | `findop`

Topics
“Compute Operating Points Using Custom Constraints and Objective Functions” on page 1-57

Introduced in R2017a

getstatestruct

Obtain state values from operating point

Syntax

```
x = getstatestruct(op)
```

Description

`x = getstatestruct(op)` extracts a structure of state values from a specified operating point object. You can use the state structure to set initial state values for your Simulink model.

Examples

Initialize Model with Operating Point Values

Open the `scdplane` model and create an operating point. You can also compute a trimmed operating point or obtain an operating point snapshot.

```
mdl = 'scdplane';
open_system(mdl)
op = operpoint(mdl);
```

Extract the state values from the operating point.

```
xInitial = getstatestruct(op);
```

Extract the input values from the operating point.

```
uInitial = getinputstruct(op);
```

To view the values of the states or inputs within this structure, use dot notation. For example, view the input values.

```
uInitial.signals.values
```

```
ans = 0
```

Set the initial state values in the model.

```
set_param(mdl,'LoadInitialState','on','InitialState','xInitial')
```

Set the initial input values in the model.

```
set_param(mdl,'LoadExternalInput','on','ExternalInput','uInitial')
```

Input Arguments

op — Operating point

`OperatingPoint` object | `OperatingSpec` object | `OperatingReport` object | array

Operating point for a Simulink model, specified as an `OperatingPoint`, `OperatingSpec`, or `OperatingReport` object. You can also specify a homogeneous array of any of these objects.

Output Arguments

x — State values

structure | structure array

State values, returned as a structure with the following fields.

- `signals` — State values and information
- `time` — Simulation time for state values, returned as `0`.

If `op` is an array, `x` is returned as a structure array with the same dimensions as `op`.

See Also

`getinputstruct` | `operpoint`

Introduced before R2006a

getxu

States and inputs from operating points

Syntax

```
x = getxu(op)
[x,u] = getxu(op)
[x,u,xstruct] = getxu(op)
```

Description

`getxu` extracts state and input values from an operating point. Use this function for applications that require state and input values to be specified in vector format, such as when using an operating in an optimization problem using `fmincon`.

Note Do not use `getxu` when setting initial state and input values using the **Data Import/Export** pane of the Configuration Parameters dialog box. Instead, use `getstatestruct` and `getinputstruct`.

`x = getxu(op)` extracts a vector of state values, `x`, from operating point, `op`.

`[x,u] = getxu(op)` also extracts a vector of input values, `u`, from the operating point. The ordering of inputs in `u` corresponds to the root-level input port numbering in Simulink.

`[x,u,xstruct] = getxu(op)` also extracts a structure of state values, `xstruct`, from the operating point. The structure of state values, `xstruct`, has the same format as that returned from a Simulink simulation.

Examples

Extract State and Input Values from Operating Point

Create an operating point for the `magball` model.

```
op = operpoint('magball');
```

View the states in the operating point.

```
op.States
```

```
ans =
      x
    ______

(1.) magball/Controller/PID Controller/Filter/Cont. Filter/Filter
         0
(2.) magball/Controller/PID Controller/Integrator/Continuous/Integrator
    14.007
```

```
(3.) magball/Magnetic Ball Plant/Current
    7.0036
(4.) magball/Magnetic Ball Plant/dhdt
         0
(5.) magball/Magnetic Ball Plant/height
      0.05
```

Extract vectors of state and input values and a state structure from the operating point.

```
[x,u,xstruct] = getxu(op)
```

```
x = 5×1

         0
   14.0071
    7.0036
         0
    0.0500


u =

     []

xstruct = struct with fields:
       time: 0
    signals: [1x5 struct]
```

View the states within the state structure.

```
xstruct.signals
```

```
ans=1×5 struct array with fields:
    values
    dimensions
    label
    blockName
    stateName
    inReferencedModel
    sampleTime
```

The `values` field shows the state values for the operating point. The `blockName` field shows the names of the block that contain each state.

See Also

`operpoint` | `operspec`

Introduced before R2006a

highlight

Package: `linearize.advisor`

Highlight linearization path in Simulink model

Syntax

```
highlight(advisor)
```

Description

`highlight(advisor)` highlights the blocks on the linearization path for the model linearization associated with a `LinearizationAdvisor` object. The software identifies blocks that are on or off the linearization path. Also, for blocks that are on the linearization path, the software indicates whether they contribute to the linearization result.

Examples

Highlight Linearization Path

Load Simulink model.

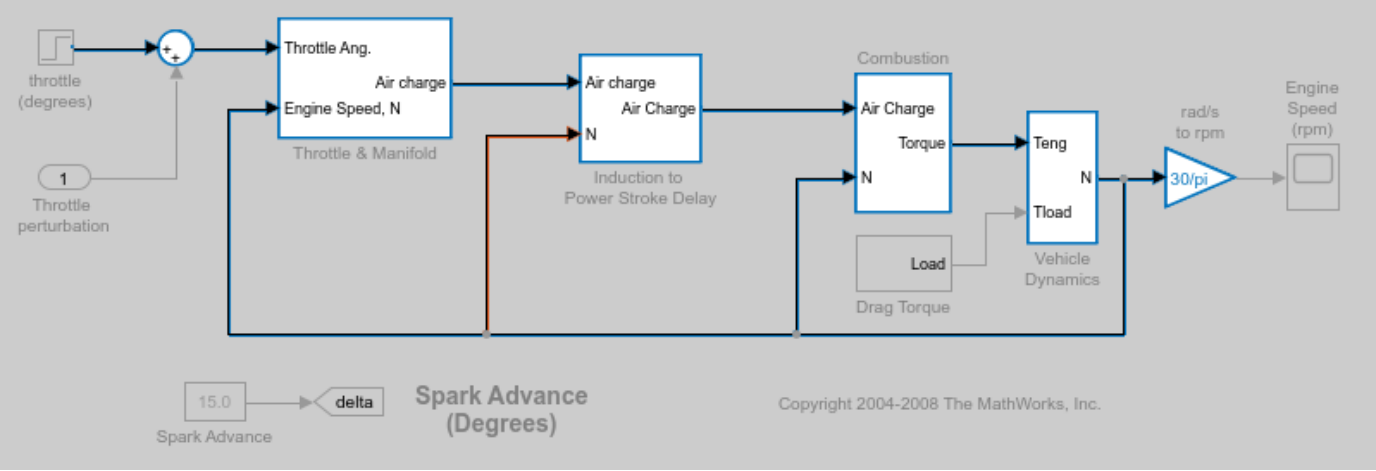
```
mdl = 'scdspeed';
load_system(mdl)
```

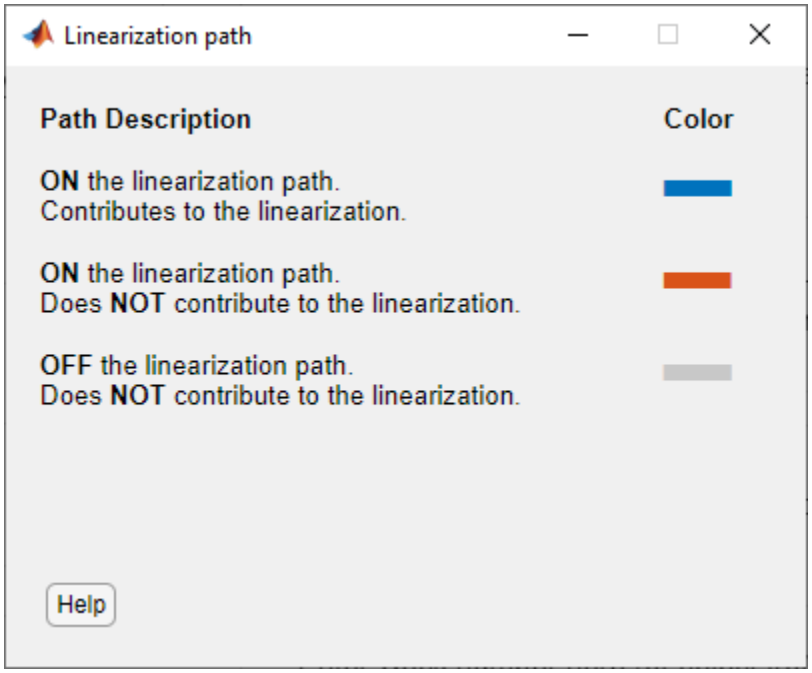
Linearize the model and obtain the `LinearizationAdvisor` object.

```
opts = linearizeOptions('StoreAdvisor',true);
io(1) = linio('scdspeed/throttle (degrees)',1,'input');
io(2) = linio('scdspeed/rad//s to rpm',1,'output');
[sys,op,info] = linearize(mdl,io,opts);
advisor = info.Advisor;
```

Highlight the linearization path.

```
highlight(advisor)
```





The Simulink model is highlighted as follows:

- Blue blocks are on the linearization path and contribute to the model linearization.
- Red blocks are on the linearization path and do not contribute to the model linearization.
- Gray blocks are not on the linearization path.

Input Arguments

`advisor` — Diagnostic information for block linearizations

`LinearizationAdvisor` object

Diagnostic information for block linearizations, specified as a `LinearizationAdvisor` object.

More About

Linearization Path

A block is on the linearization path if there is a signal path from at least one linearization input to at least one linearization output that passes through the block.

See Also

Objects

`LinearizationAdvisor`

Functions

`advise` | `find` | `getBlockInfo`

Topics
“Troubleshoot Linearization Results at Command Line” on page 4-28

Introduced in R2017b

initopspec

Initialize operating point specification values

Syntax

```
opspecNew = initopspec(opspec,op)

opspecNew = initopspec(opspec,x)
opspecNew = initopspec(opspec,x,u)

opspecNew = initopspec(opspec,xstruct)
opspecNew = initopspec(opspec,xstruct,u)
```

Description

`opspecNew = initopspec(opspec,op)` updates the operating point specification `opspec` with the values in the operating point `op` and returns a new operating point specification.

`opspecNew = initopspec(opspec,x)` updates the operating point specification using the state values in the vector `x`.

`opspecNew = initopspec(opspec,x,u)` updates the operating point specification using the state and input values in the vectors `x` and `u`, respectively.

`opspecNew = initopspec(opspec,xstruct)` updates the operating point specification using the state values in the structure `xstruct`. You can use this syntax if your operating point specification does not have input values.

`opspecNew = initopspec(opspec,xstruct,u)` updates the operating point specification using the state and input values in the structure `xstruct` and vector `u`, respectively.

Examples

Update Operating Point Specification

Find an operating point for the model using a simulation snapshot.

```
mdl = 'scdplane';
op = findop(mdl,10);
```

Create an operating point specification for your model.

```
opspec = operspec(mdl);
```

Update the operating point specification using the computed operating point values.

```
newopspec = initopspec(opspec,op);
```

Input Arguments

opspec — Operating point specification

`OperatingSpec` object

Operating point specification for a Simulink model, specified as an `OperatingSpec` object. To create an `OperatingSpec` object for your model, use the `operspec` function.

op — Operating point

`OperatingPoint` object | `OperatingSpec` object | `OperatingReport` object

Operating point for a Simulink model, specified as an `OperatingPoint`, `OperatingSpec`, or `OperatingReport` object.

x — State values

vector

State values, specified as vector. The order of values in `x` must match the state order in `opspec`.

You can create `x` using `getxu`.

u — Input values

vector

Input values, specified as a vector. The order of values in `u` must match the state order in `opspec`.

You can create `u` using `getxu`.

xstruct — State information

structure

State information, specified as a structure. Create `xstruct` using `getstatestruct`.

Output Arguments

opspecNew — Updated operating point specification

`OperatingSpec` object

Updated operating point specification, returned as an `OperatingSpec` object.

Alternatives

As an alternative to the `initopspec` function, initialize operating point specification values in the **Model Linearizer** app. For more information, see “Import and Export Specifications for Operating Point Search” on page 1-53.

See Also

`findop` | `getstatestruct` | `getxu` | `operpoint` | `operspec`

Introduced before R2006a

linearize

Linear approximation of Simulink model or subsystem

Syntax

```
linsys = linearize(mdl,io)
linsys = linearize(mdl,io,op)
linsys = linearize(mdl,io,param)
linsys = linearize(mdl,io,blocksub)
linsys = linearize(mdl,io,options)
linsys = linearize(mdl,io,op,param,blocksub,options)

linsys = linearize(mdl,blockpath)
linsys = linearize(mdl,blockpath,op)
linsys = linearize(mdl,blockpath,param)
linsys = linearize(mdl,blockpath,blocksub)
linsys = linearize(mdl,blockpath,options)
linsys = linearize(mdl,blockpath,op,param,blocksub,options)

linsys = linearize( ___ ,'StateOrder',stateorder)

[linsys,linop] = linearize( ___ )

[linsys,linop,info] = linearize( ___ )
```

Description

`linsys = linearize(mdl,io)` returns a linear approximation of the nonlinear Simulink model `mdl` at the model operating point using the analysis points specified in `io`. If you omit `io`, then `linearize` uses the root level inports and outports of the model as analysis points.

`linsys = linearize(mdl,io,op)` linearizes the model at operating point `op`.

`linsys = linearize(mdl,io,param)` linearizes the model using the parameter value variations specified in `param`. You can vary any model parameter with a value given by a variable in the model workspace, the MATLAB workspace, or a data dictionary.

`linsys = linearize(mdl,io,blocksub)` linearizes the model using the substitute block or subsystem linearizations specified in `blocksub`.

`linsys = linearize(mdl,io,options)` linearizes the model using additional linearization options.

`linsys = linearize(mdl,io,op,param,blocksub,options)` linearizes the model using any combination of `op`, `param`, `blocksub`, and `options` in any order.

`linsys = linearize(mdl,blockpath)` returns a linear approximation of a block or subsystem in model `mdl`, specified by `blockpath`, at the model operating point. The software isolates the block from the rest of the model before linearization.

`linsys = linearize(mdl,blockpath,op)` linearizes the block or subsystem at operating point `op`.

`linsys = linearize(mdl,blockpath,param)` linearizes the block or subsystem using the parameter value variations specified in `param`. You can vary any model parameter with a value given by a variable in the model workspace, the MATLAB workspace, or a data dictionary.

`linsys = linearize(mdl,blockpath,blocksub)` linearizes the block or subsystem using the substitute block or subsystem linearizations specified in `blocksub`.

`linsys = linearize(mdl,blockpath,options)` linearizes the block or subsystem using additional linearization options.

`linsys = linearize(mdl,blockpath,op,param,blocksub,options)` linearizes the block or subsystem using any combination of `op`, `param`, `blocksub`, and `options` in any order.

`linsys = linearize( ___ ,'StateOrder',stateorder)` specifies the order of the states in the linearized model for any of the previous syntaxes.

`[linsys,linop] = linearize( ___ )` returns the operating point at which the model was linearized. Use this syntax when linearizing at simulation snapshots or when varying parameters during linearization.

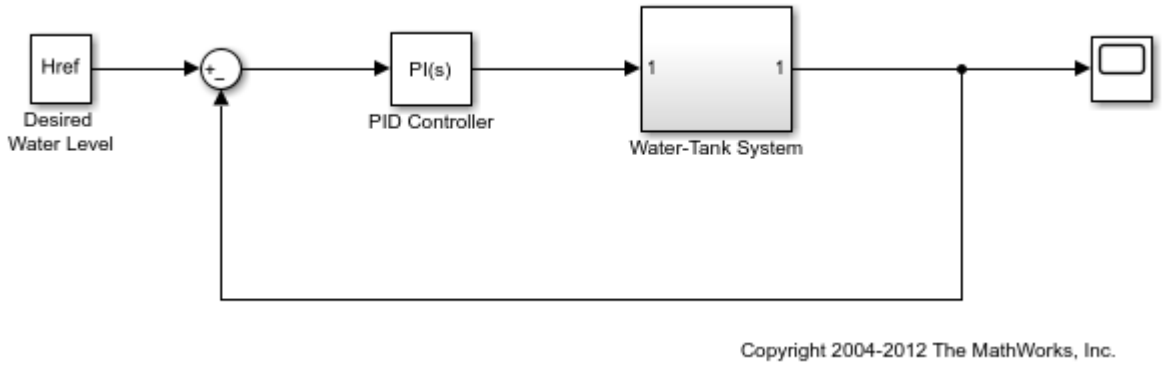
`[linsys,linop,info] = linearize( ___ )` returns additional linearization information. To select the linearization information to return in `info`, enable the corresponding option in `options`.

Examples

Linearize Model Using Specified I/O Set

Open the Simulink model.

```
mdl = 'watertank';
open_system(mdl)
```



Specify a linearization input at the output of the PID Controller block, which is the input signal for the Water-Tank System block.

```
io(1) = linio('watertank/PID Controller',1,'input');
```

Specify a linearization output point at the output of the Water-Tank System block. Specifying the output point as open-loop removes the effects of the feedback signal on the linearization without changing the model operating point.

```
io(2) = linio('watertank/Water-Tank System',1,'openoutput');
```

Linearize the model using the specified I/O set.

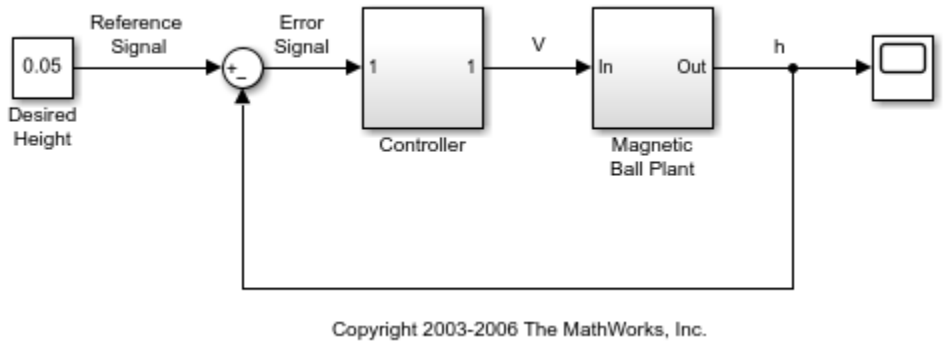
```
linsys = linearize(mdl,io);
```

`linsys` is the linear approximation of the plant at the model operating point.

Linearize Model at Specified Operating Point

Open the Simulink model.

```
mdl = 'magball';
open_system(mdl)
```



Find a steady-state operating point at which the ball height is `0.05`. Create a default operating point specification, and set the height state to a known value.

```
opspec = operspec(mdl);
opspec.States(5).Known = 1;
opspec.States(5).x = 0.05;
```

Trim the model to find the operating point.

```
options = findopOptions('DisplayReport','off');
op = findop(mdl,opspec,options);
```

Specify linearization input and output signals to compute the closed-loop transfer function.

```
io(1) = linio('magball/Desired Height',1,'input');
io(2) = linio('magball/Magnetic Ball Plant',1,'output');
```

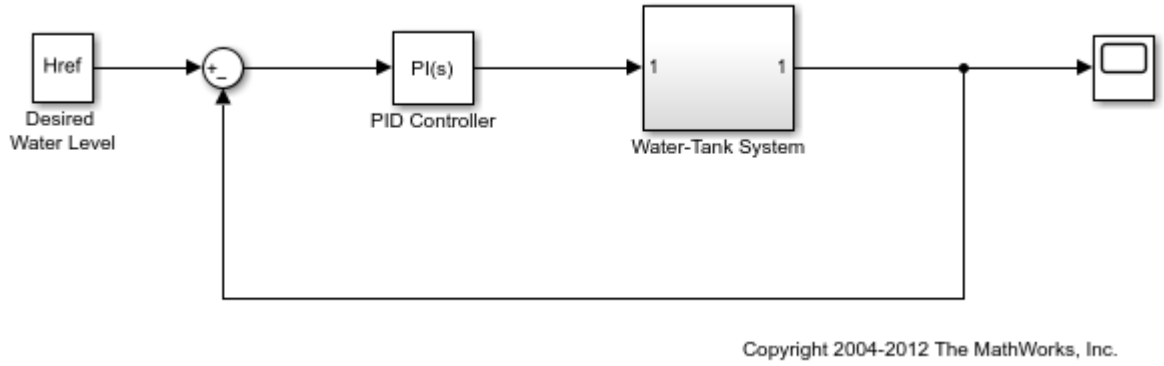
Linearize the model at the specified operating point using the specified I/O set.

```
linsys = linearize(mdl,io,op);
```

Linearize Model at Simulation Snapshot Time

Open the Simulink model.

```
mdl = 'watertank';
open_system(mdl)
```



To compute the closed-loop transfer function, first specify the linearization input and output signals.

```
io(1) = linio('watertank/PID Controller',1,'input');
io(2) = linio('watertank/Water-Tank System',1,'output');
```

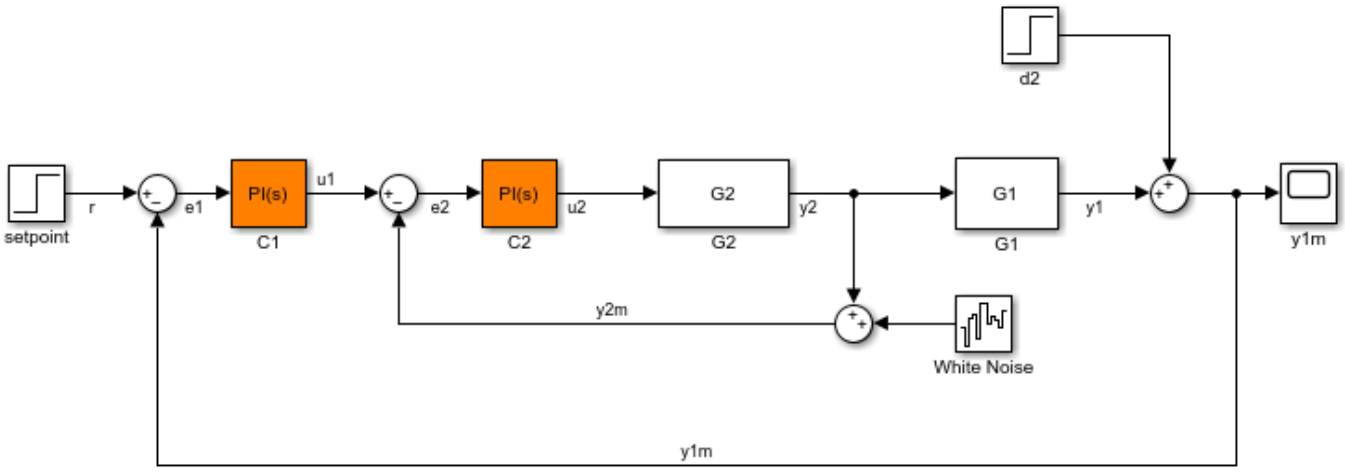
Simulate `sys` for `10` seconds and linearize the model.

```
linsys = linearize(mdl,io,10);
```

Batch Linearize Model for Parameter Variations

Open the Simulink model.

```
mdl = 'scdcascade';
open_system(mdl)
```



Specify parameter variations for the outer-loop controller gains, `Kp1` and `Ki1`. Create parameter grids for each gain value.

```
Kp1_range = linspace(Kp1*0.8,Kp1*1.2,6);
Ki1_range = linspace(Ki1*0.8,Ki1*1.2,4);
[Kp1_grid,Ki1_grid] = ndgrid(Kp1_range,Ki1_range);
```

Create a parameter value structure with fields `Name` and `Value`.

```
params(1).Name = 'Kp1';
params(1).Value = Kp1_grid;
params(2).Name = 'Ki1';
params(2).Value = Ki1_grid;
```

`params` is a 6-by-4 parameter value grid, where each grid point corresponds to a unique combination of `Kp1` and `Ki1` values.

Define linearization input and output points for computing the closed-loop response of the system.

```
io(1) = linio('scdcascade/setpoint',1,'input');
io(2) = linio('scdcascade/Sum',1,'output');
```

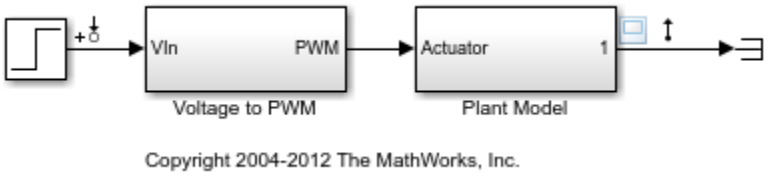
Linearize the model at the model operating point using the specified parameter values.

```
linsys = linearize(mdl,io,params);
```

Specify Substitute Block Linearization and Linearize Model

Open the Simulink model.

```
mdl = 'scdpwm';
open_system(mdl)
```



Extract linearization input and output from the model.

```
io = getlinio(mdl);
```

Linearize the model at the model operating point.

```
linsys = linearize(mdl,io)
```

```
linsys =

  D =
                 Step
   Plant Model     0
```

```
Static gain.
```

The discontinuities in the Voltage to PWM block cause the model to linearize to zero. To treat this block as a unit gain during linearization, specify a substitute linearization for this block.

```
blocksub.Name = 'scdpwm/Voltage to PWM';
blocksub.Value = 1;
```

Linearize the model using the specified block substitution.

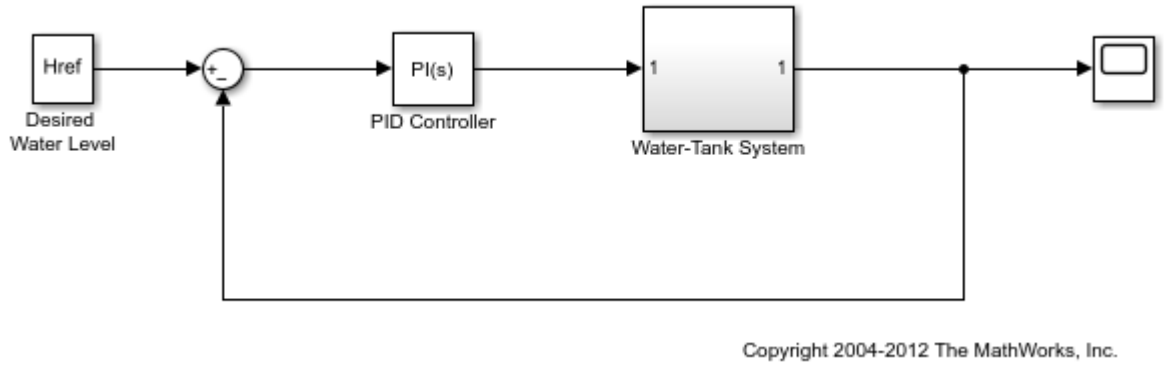
```
linsys = linearize(mdl,blocksub,io)
```

```
linsys =

  A =
                 State Space(  State Space(
   State Space(        0.9999       -0.0001
   State Space(        0.0001             1

  B =
                   Step
   State Space(  0.0001
   State Space(   5e-09

  C =
                State Space(  State Space(
   Plant Model             0             1

  D =
                Step
   Plant Model     0

Sample time: 0.0001 seconds
Discrete-time state-space model.
```

Specify Sample Time of Linearized Model

Open the Simulink model.

```
mdl = 'watertank';
open_system(mdl)
```



Copyright 2004-2012 The MathWorks, Inc.

To linearize the Water-Tank System block, specify a linearization input and output.

```
io(1) = linio('watertank/PID Controller',1,'input');
io(2) = linio('watertank/Water-Tank System',1,'openoutput');
```

Create a linearization option set, and specify the sample time for the linearized model.

```
options = linearizeOptions('SampleTime',0.1);
```

Linearize the plant using the specified options.

```
linsys = linearize(mdl,io,options)
```

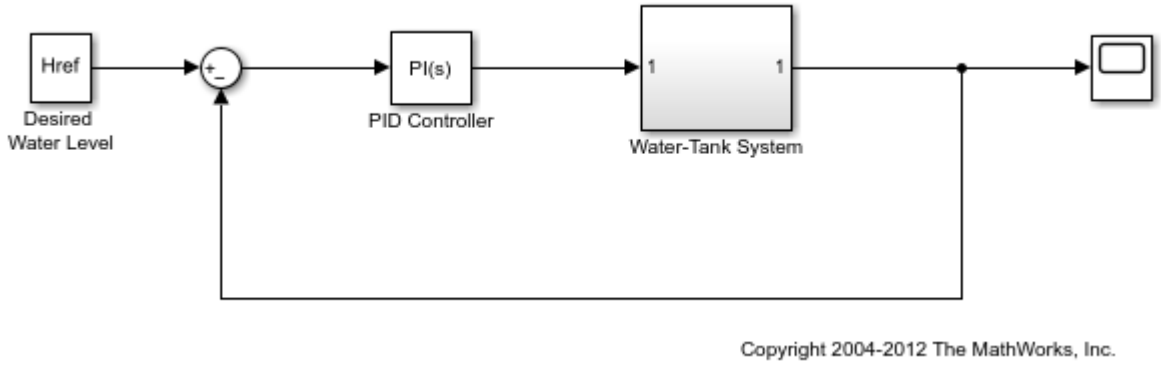
```
linsys =

  A =
          H
   H  0.995

  B =
      PID Controll
   H       0.02494

  C =
                 H
   Water-Tank S  1

  D =
                 PID Controll
   Water-Tank S             0

Sample time: 0.1 seconds
Discrete-time state-space model.
```

The linearized plant is a discrete-time state-space model with a sample time of `0.1`.

Linearize Block or Subsystem at Model Operating Point

Open the Simulink model.

```
mdl = 'watertank';
open_system(mdl)
```



Copyright 2004-2012 The MathWorks, Inc.

Specify the full block path for the block you want to linearize.

```
blockpath = 'watertank/Water-Tank System';
```

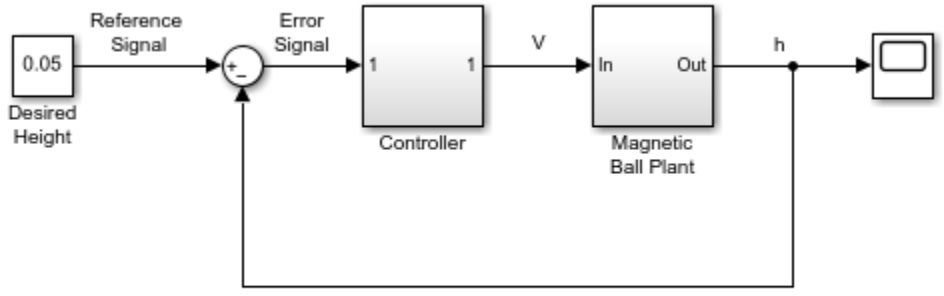
Linearize the specified block at the model operating point.

```
linsys = linearize(mdl,blockpath);
```

Linearize Block or Subsystem at Trimmed Operating Point

Open Simulink model.

```
mdl = 'magball';
open_system(mdl)
```



Copyright 2003-2006 The MathWorks, Inc.

Find a steady-state operating point at which the ball height is `0.05`. Create a default operating point specification, and set the height state to a known value.

```
opspec = operspec(mdl);
opspec.States(5).Known = 1;
opspec.States(5).x = 0.05;

options = findopOptions('DisplayReport','off');
op = findop(mdl,opspec,options);
```

Specify the block path for the block you want to linearize.

```
blockpath = 'magball/Magnetic Ball Plant';
```

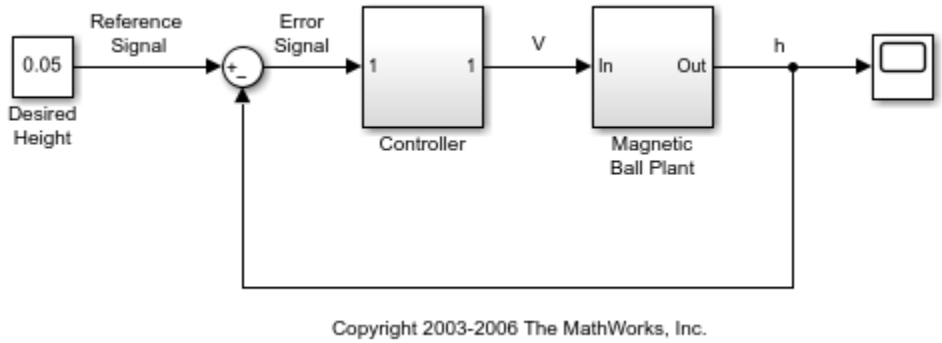
Linearize the specified block at the specified operating point.

```
linsys = linearize(mdl,blockpath,op);
```

Specify State Order in Linearized Model

Open the Simulink model.

```
mdl = 'magball';
open_system(mdl)
```



Linearize the plant at the model operating point.

```
blockpath = 'magball/Magnetic Ball Plant';
linsys = linearize(mdl,blockpath);
```

View the default state order for the linearized plant.

```
linsys.StateName
```

```
ans =

  3x1 cell array

    {'height' }
    {'Current'}
    {'dhdt'   }
```

Linearize the plant and reorder the states in the linearized model. Set the rate of change of the height as the second state.

```
stateorder = {'magball/Magnetic Ball Plant/height';...
              'magball/Magnetic Ball Plant/dhdt';...
              'magball/Magnetic Ball Plant/Current'};
linsys = linearize(mdl,blockpath,'StateOrder',stateorder);
```

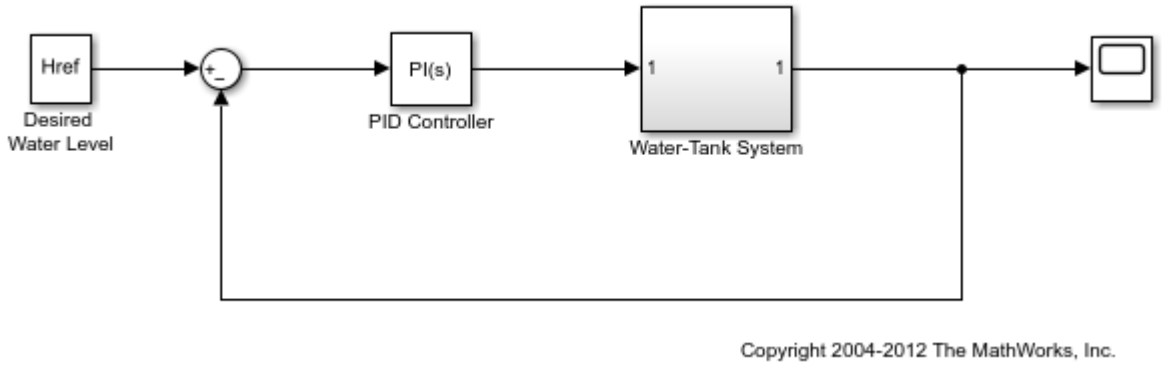
View the new state order.

```
linsys.StateName


ans =

  3x1 cell array

    {'height' }
    {'dhdt'   }
    {'Current'}
```

Linearize Model at Multiple Snapshot Times

Open the Simulink model.

```
mdl = 'watertank';
open_system(mdl)
```



To compute the closed-loop transfer function, first specify the linearization input and output signals.

```
io(1) = linio('watertank/PID Controller',1,'input');
io(2) = linio('watertank/Water-Tank System',1,'output');
```

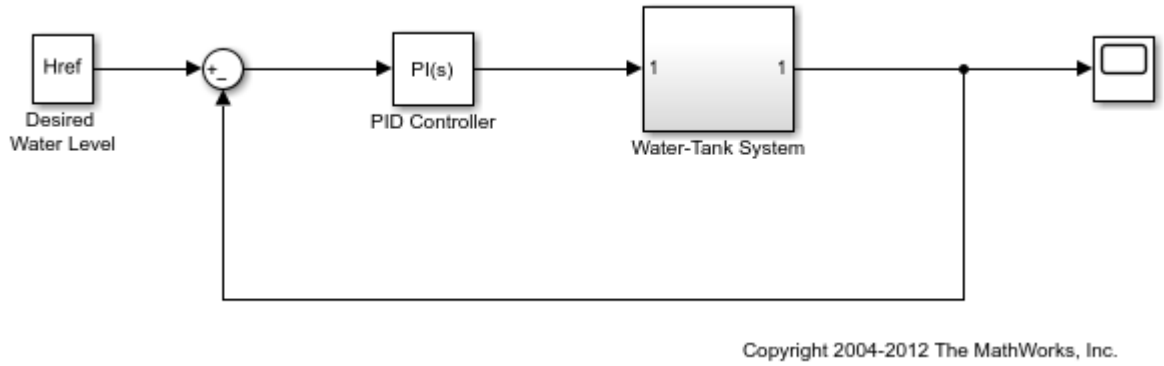
Simulate `sys` and linearize the model at `0` and `10` seconds. Return the operating points that correspond to these snapshot times; that is, the operating points at which the model was linearized.

```
[linsys,linop] = linearize(mdl,io,[0,10]);
```

Batch Linearize Plant Model and Obtain Linearization Offsets

Open the Simulink model.

```
mdl = 'watertank';
open_system(mdl)
```



Copyright 2004-2012 The MathWorks, Inc.

Vary parameters `A` and `b` within 10% of their nominal values.

```
[A_grid,b_grid] = ndgrid(linspace(0.9*A,1.1*A,3),...
                         linspace(0.9*b,1.1*b,4));
```

Create a parameter structure array, specifying the name and grid points for each parameter.

```
params(1).Name = 'A';
params(1).Value = A_grid;
params(2).Name = 'b';
params(2).Value = b_grid;
```

Create a default operating point specification for the model.

```
opspec = operspec(mdl);
```

Trim the model using the specified operating point specification, parameter grid. Suppress the display of the operating point search report.

```
opt = findopOptions('DisplayReport','off');
[op,opreport] = findop(mdl,opspec,params,opt);
```

`op` is a 3-by-4 array of operating point objects that correspond to the specified parameter grid points.

Specify the block path for the plant model.

```
blockpath = 'watertank/Desired  Water Level';
```

To store offsets during linearization, create a linearization option set and set `StoreOffsets` to `true`.

```
options = linearizeOptions('StoreOffsets',true);
```

Batch linearize the plant at the trimmed operating points, using the specified I/O points and parameter variations.

```
[linsys,linop,info] = linearize(mdl,blockpath,op,params,options);
```

You can use the offsets in `info.Offsets` when configuring an LPV System block.

```
info.Offsets


ans =
```

```
3x4 struct array with fields:

  x
  dx
  u
  y
  StateName
  InputName
  OutputName
  Ts
```

Input Arguments

`mdl` — Simulink model name

character vector | string

Simulink model name, specified as a character vector or string. The model must be in the current working folder or on the MATLAB path.

`io` — Analysis point set

linearization I/O object | vector of linearization I/O objects

Analysis point set that contains inputs, outputs, and openings, specified as a linearization I/O object or a vector of linearization I/O objects. To create `io`:

- Define the inputs, outputs, and openings using `linio`.
- If the inputs, outputs, and openings are specified in the Simulink model, extract these points from the model using `getlinio`.

Each linearization I/O object in `io` must correspond to the Simulink model `mdl` or some normal mode model reference in the model hierarchy.

If you omit `io`, then `linearize` uses the root level inports and outports of the model as analysis points.

For more information on specifying linearization inputs, outputs, and openings, see “Specify Portion of Model to Linearize” on page 2-10.

`op` — Operating point

`OperatingPoint` object | array of `OperatingPoint` objects | vector of positive scalars

Operating point for linearization, specified as one of the following:

- `OperatingPoint` object, created using:
  - `operpoint`
  - `findop` with either a single operating point specification, or a single snapshot time.
- Array of `OperatingPoint` objects, specifying multiple operating points. To create an array of `OperatingPoint` objects, you can:
  - Extract operating points at multiple snapshot times using `findop`.
  - Batch trim your model using multiple operating point specifications. For more information, see “Batch Compute Steady-State Operating Points for Multiple Specifications” on page 1-68.

    - Batch trim your model using parameter variations. For more information, see “Batch Compute Steady-State Operating Points for Parameter Variation” on page 1-72.
- Vector of positive scalars representing one or more simulation snapshot times. The software simulates `sys` and linearizes the model at the specified snapshot times.

  If you also specify parameter variations using `param`, the software simulates the model for each snapshot time and parameter grid point combination. This operation can be computationally expensive.

If you specify parameter variations using `param`, and the parameters:

- Affect the model operating point, then specify `op` as an array of operating points with the same dimensions as the parameter value grid. To obtain the operating points that correspond to the parameter value combinations, batch trim your model using `param` before linearization. For more information, see “Batch Linearize Model at Multiple Operating Points Derived from Parameter Variations” on page 3-16.
- Do not affect the model operating point, then specify `op` as a single operating point.

`blockpath` — Block or subsystem
character vector | string

Block or subsystem to linearize, specified as a character vector or string that contains its full block path.

The software treats the inports and outports of the specified block as open-loop inputs and outputs, which isolates the block from the rest of the model before linearization.

`blocksub` — Substitute linearizations for blocks and subsystems
structure | structure array

Substitute linearizations for blocks and subsystems, specified as a structure or an *n*-by-1 structure array, where *n* is the number of blocks for which you want to specify a linearization. Use `blocksub` to specify a custom linearization for a block or subsystem. For example, you can specify linearizations for blocks that do not have analytic linearizations, such as blocks with discontinuities or triggered subsystems.

To study the effects of varying the linearization of a block on the model dynamics, you can batch linearize your model by specifying multiple substitute linearizations for a block.

If you substitute a linearization with a sample time that differs from that of the original block or subsystem, it is best practice to set the overall linearization sample time (`options.SampleTime`) to a nondefault value.

Each substitute linearization structure has the following fields.

`Name` — Block path
character vector | string

Block path of the block for which you want to specify the linearization, specified as a character vector or string.

`Value` — Substitute linearization
double | double array | LTI model | model array | structure

Substitute linearization for the block, specified as one of the following:

- Double — Specify the linearization of a SISO block as a gain.
- Array of doubles — Specify the linearization of a MIMO block as an n_u-by-n_y array of gain values, where n_u is the number of inputs and n_y is the number of outputs.
- LTI model, uncertain state-space model, or uncertain real object — The I/O configuration of the specified model must match the configuration of the block specified by `Name`. Using an uncertain model requires Robust Control Toolbox software.
- Array of LTI models, uncertain state-space models, or uncertain real objects — Batch linearize the model using multiple block substitutions. The I/O configuration of each model in the array must match the configuration of the block for which you are specifying a custom linearization. If you:
  - Vary model parameters using `param` and specify `Value` as a model array, the dimensions of `Value` must match the parameter grid size.
  - Specify `op` as an array of operating points and `Value` as a model array, the dimensions of `Value` must match the size of `op`.
  - Define block substitutions for multiple blocks, and specify `Value` as an array of LTI models for one or more of these blocks, the dimensions of the arrays must match.
- Structure with the following fields.

Field	Description
`Specification`	Block linearization, specified as a character vector that contains one of the following: • MATLAB expression • Name of a “Custom Linearization Function” on page 18-156 in your current working folder or on the MATLAB path The specified expression or function must return one of the following: • Linear model in the form of a D-matrix • Control System Toolbox LTI model object • Uncertain state-space model or uncertain real object (requires Robust Control Toolbox software) The I/O configuration of the returned model must match the configuration of the block specified by `Name`.
`Type`	Specification type, specified as one of the following: • `'Expression'` • `'Function'`
`ParameterNames`	Linearization function parameter names, specified as a cell array of character vectors. Specify `ParameterNames` only when `Type = 'Function'` and your block linearization function requires input parameters. These parameters only impact the linearization of the specified block. You must also specify the corresponding `blocksub.Value.ParameterValues` field.

Field	Description
`ParameterValues`	Linearization function parameter values, specified as a vector of doubles. The order of parameter values must correspond to the order of parameter names in `blocksub.Value.ParameterNames`. Specify `ParameterValues` only when `Type = 'Function'` and your block linearization function requires input parameters.

`param` — Parameter samples
structure | structure array

Parameter samples for linearization, specified as one of the following:

- Structure — Vary the value of a single parameter by specifying `param` as a structure with the following fields:
  - `Name` — Parameter name, specified as a character vector or string. You can specify any model parameter that is a variable in the model workspace, the MATLAB workspace, or a data dictionary. If the variable used by the model is not a scalar variable, specify the parameter name as an expression that resolves to a numeric scalar value. For example, to use the first element of vector `V` as a parameter, use:

    ```
    param.Name = 'V(1)';
    ```
  - `Value` — Parameter sample values, specified as a double array.

  For example, vary the value of parameter `A` in the 10% range:

  ```
  param.Name = 'A';
  param.Value = linspace(0.9*A,1.1*A,3);
  ```
- Structure array — Vary the value of multiple parameters. For example, vary the values of parameters `A` and `b` in the 10% range:

  ```
  [A_grid,b_grid] = ndgrid(linspace(0.9*A,1.1*A,3),...
                           linspace(0.9*b,1.1*b,3));
  params(1).Name = 'A';
  params(1).Value = A_grid;
  params(2).Name = 'b';
  params(2).Value = b_grid;
  ```

For more information, see “Specify Parameter Samples for Batch Linearization” on page 3-43.

If `param` specifies tunable parameters only, the software batch linearizes the model using a single model compilation.

To compute the offsets required by the LPV System block, specify `param`, and set `options.StoreOffsets` to `true`. You can then return additional linearization information in `info`, and extract the offsets using `getOffsetsForLPV`.

`stateorder` — State order in linearization results
cell array of character vectors

State order in linearization results, specified as a cell array of block paths or state names. The order of the block paths and states in `stateorder` indicates the order of the states in `linsys`.

You can specify block paths for any blocks in `mdl` that have states, or any named states in `mdl`.

You do not have to specify every block and state from `mdl` in `stateorder`. The states you specify appear first in `linsys`, followed by the remaining states in their default order.

`options` — Linearization algorithm options
`linearizeOptions` option set

Linearization algorithm options, specified as a `linearizeOptions` option set.

Output Arguments

`linsys` — Linearization result
state-space model | array of state-space models

Linearization result, returned as a state-space model or an array of state-space models.

For most models, `linsys` is returned as an `ss` object or an array of `ss` objects. However, if `mdl` contains one of the following blocks in the linearization path defined by `io`, then `linsys` returns the specified type of state-space model.

Block	`linsys` Type
Block with a substitution specified as a `genss` object or tunable model object	`genss`
Block with a substitution specified as an uncertain model, such as `uss`	`uss`
Sparse Second Order block	`mechss`
Descriptor State-Space block configured to linearize to a sparse model	`sparss`

The dimensions of `linsys` depend on the specified parameter variations and block substitutions, and the operating points at which you linearize the model.

Note If you specify more than one of `op`, `param`, or `blocksub.Value` as an array, then their dimensions must match.

Parameter Variation	Block Substitution	Linearize At...	Resulting `linsys` Dimensions
No parameter variation	No block substitution	Model operating point	Single state-space model
		Single operating point, specified as an `OperatingPoint` object or snapshot time using `op`	
		N_1-by-`...`-by-N_m array of `OperatingPoint` objects, specified by `op`	N_1-by-`...`-by-N_m

Parameter Variation	Block Substitution	Linearize At...	Resulting `linsys` Dimensions
		N_s snapshots, specified as a vector of snapshot times using `op`	Column vector of length N_s
	N_1-by-...-by-N_m model array for at least one block, specified by `blocksub.Value`	Model operating point	N_1-by-...-by-N_m
		Single operating point, specified as an `OperatingPoint` object or snapshot time using `op`	
		N_1-by-...-by-N_m array of operating points, specified as an array of `OperatingPoint` objects using `op`	
		N_s snapshots, specified as a vector of snapshot times using `op`	N_s-by-N_1-by-...-by-N_m
N_1-by-...-by-N_m parameter grid, specified by `param`	Either no block substitution or an N_1-by-...-by-N_m model array for at least one block, specified by `blocksub.Value`	Model operating point	N_1-by-...-by-N_m
		Single operating point, specified as an `OperatingPoint` object or snapshot time using `op`	
		N_1-by-...-by-N_m array of `OperatingPoint` objects, specified by `op`	
		N_s snapshots, specified as a vector of snapshot times using `op`	N_s-by-N_1-by-...-by-N_m

For example, suppose:

- `op` is a 4-by-3 array of `OperatingPoint` objects and you do not specify parameter variations or block substitutions. In this case, `linsys` is a 4-by-3 model array.
- `op` is a single `OperatingPoint` object and `param` specifies a 3-by-4-by-2 parameter grid. In this case, `linsys` is a 3-by-4-by-2 model array.
- `op` is a row vector of positive scalars with two elements and you do not specify `param`. In this case, `linsys` is a column vector with two elements.
- `op` is a column vector of positive scalars with three elements and `param` specifies a 5-by-6 parameter grid. In this case, `linsys` is a 3-by-5-by-6 model array.
- `op` is a single operating point object, you do not specify parameter variations, and `blocksub.Value` is a 2-by-3 model array for one block in the model. In this case, `linsys` is a 2-by-3 model array.

- `op` is a column vector of positive scalars with four elements, you do not specify parameter variations, and `blocksub.Value` is a 1-by-2 model array for one block in the model. In this case, `linsys` is a 4-by-1-by-2 model array.

For more information on model arrays, see “Model Arrays”.

`linop` — Operating point
`OperatingPoint` object | array of `OperatingPoint` objects

Operating point at which the model was linearized, returned as an `OperatingPoint` object or an array of `OperatingPoint` objects with the same dimensions as `linsys`. Each element of `linop` is the operating point at which the corresponding `linsys` model was obtained.

If you specify `op` as a single operating point`OperatingPoint` object or an array of `OperatingPoint` objects, then `linop` is a copy of `op`. If you specify `op` as a single operating point object and also specify parameter variations using `param`, then `linop` is an array with the same dimensions as the parameter grid. In this case, the elements of `linop` are scalar expanded copies of `op`.

To determine whether the model was linearized at a reasonable operating point, view the states and inputs in `linop`.

`info` — Linearization information
structure

Linearization information, returned as a structure with the following fields:

`Offsets` — Linearization offsets
`[]` (default) | structure | structure array

Linearization offsets that correspond to the operating point at which the model was linearized, returned as `[]` if `options.StoreOffsets` is `false`. Otherwise, `Offsets` is returned as one of the following:

- If `linsys` is a single state-space model, then `Offsets` is a structure.
- If `linsys` is an array of state-space models, then `Offsets` is a structure array with the same dimensions as `linsys`.

Each offset structure has the following fields:

Field	Description
`x`	State offsets used for linearization, returned as a column vector of length n_x, where n_x is the number of states in `linsys`.
`y`	Output offsets used for linearization, returned as a column vector of length n_y, where n_y is the number of outputs in `linsys`.
`u`	Input offsets used for linearization, returned as a column vector of length n_u, where n_u is the number of inputs in `linsys`.
`dx`	Derivative offsets for continuous time systems or updated state values for discrete-time systems, returned as a column vector of length n_x.
`StateName`	State names, returned as a cell array that contains n_x elements that match the names in `linsys.StateName`.

Field	Description
`InputName`	Input names, returned as a cell array that contains n_u elements that match the names in `linsys.InputName`.
`OutputName`	Output names, returned as a cell array that contains n_y elements that match the names in `linsys.OutputName`.
`Ts`	Sample time of the linearized system, returned as a scalar that matches the sample time in `linsys.Ts`. For continuous-time systems, `Ts` is `0`.

If `Offsets` is a structure array, you can configure an LPV System block using the offsets. To do so, first convert them to the required format using `getOffsetsForLPV`. For an example, see “Approximate Nonlinear Behavior Using Array of LTI Systems” on page 3-70.

`Advisor` — Linearization diagnostic information

`[]` (default) | `LinearizationAdvisor` object | array of `LinearizationAdvisor` objects

Linearization diagnostic information, returned as `[]` if `options.StoreAdvisor` is `false`. Otherwise, `Advisor` is returned as one of the following:

- If `linsys` is a single state-space model, `Advisor` is a `LinearizationAdvisor` object.
- If `linsys` is an array of state-space models, `Advisor` is an array of `LinearizationAdvisor` objects with the same dimensions as `linsys`.

`LinearizationAdvisor` objects store linearization diagnostic information for individual linearized blocks. For an example of troubleshooting linearization results using a `LinearizationAdvisor` object, see “Troubleshoot Linearization Results at Command Line” on page 4-28.

More About

Custom Linearization Function

You can specify a substitute linearization for a block or subsystem in your Simulink model using a custom function on the MATLAB path.

Your custom linearization function must have one `BlockData` input argument, which is a structure that the software creates and passes to the function. `BlockData` has the following fields:

Field	Description
`BlockName`	Name of the block for which you are specifying a custom linearization.
`Parameters`	Block parameter values, specified as a structure array with `Name` and `Value` fields. `Parameters` contains the names and values of the parameters you specify in the `blocksub.Value.ParameterNames` and `blocksub.Value.ParameterValues` fields.

<table>
<tr><th>Field</th><th>Description</th></tr>
<tr><td>Inputs</td><td>Input signals to the block for which you are defining a linearization, specified as a structure array with one structure for each block input. Each structure in Inputs has the following fields:

<table><tr><th>Field</th><th>Description</th></tr><tr><td>BlockName</td><td>Full block path of the block whose output connects to the corresponding block input.</td></tr><tr><td>PortIndex</td><td>Output port of the block specified by BlockName that connects to the corresponding block input.</td></tr><tr><td>Values</td><td>Value of the signal specified by BlockName and PortIndex. If this signal is a vector signal, then Values is a vector with the same dimension.</td></tr></table></td></tr>
<tr><td>ny</td><td>Number of output channels of the block linearization.</td></tr>
<tr><td>nu</td><td>Number of input channels of the block linearization.</td></tr>
<tr><td>BlockLinearization</td><td>Current default linearization of the block, specified as a state-space model. You can specify a block linearization that depends on the default linearization using BlockLinearization.</td></tr>
</table>

Your custom function must return a model with `nu` inputs and `ny` outputs. This model must be one of the following:

- Linear model in the form of a D-matrix
- Control System Toolbox LTI model object
- Uncertain state-space model or uncertain real object (requires Robust Control Toolbox software)

For example, the following function multiplies the current default block linearization, by a delay of `Td = 0.5` seconds. The delay is represented by a Thiran filter with sample time `Ts = 0.1`. The delay and sample time are parameters stored in `BlockData`.

```
function sys = myCustomFunction(BlockData)
    Td = BlockData.Parameters(1).Value;
    Ts = BlockData.Parameters(2).Value;
    sys = BlockData.BlockLinearization*Thiran(Td,Ts);
end
```

Save this function to a location on the MATLAB path.

To use this function as a custom linearization for a block or subsystem, specify the `blocksub.Value.Specification` and `blocksub.Value.Type` fields.

```
blocksub.Value.Specification = 'myCustomFunction';
blocksub.Value.Type = 'Function';
```

To set the delay and sample time parameter values, specify the `blocksub.Value.ParameterNames` and `blocksub.Value.ParameterValues` fields.

```
blocksub.Value.ParameterNames = {'Td','Ts'};
blocksub.Value.ParameterValues = [0.5 0.1];
```

Algorithms

Model Properties for Linearization

By default, `linearize` automatically sets the following Simulink model properties:

- `BufferReuse = 'off'`
- `RTWInlineParameters = 'on'`
- `BlockReductionOpt = 'off'`
- `SaveFormat = 'StructureWithTime'`

After linearization, Simulink restores the original model properties.

Block-by-Block Linearization

Simulink Control Design software linearizes models using a block-by-block approach. The software individually linearizes each block in your Simulink model and produces the linearization of the overall system by combining the individual block linearizations.

The software determines the input and state levels for each block from the operating point, and obtains the Jacobian for each block at these levels.

For some blocks, the software cannot compute an analytical linearization in this manner. For example:

- Some nonlinearities do not have a defined Jacobian.
- Some discrete blocks, such as state charts and triggered subsystems, tend to linearize to zero.
- Some blocks do not implement a Jacobian.
- Custom blocks, such as S-Function blocks and MATLAB Function blocks, do not have analytical Jacobians.

You can specify a custom linearization for any such blocks for which you know the expected linearization. If you do not specify a custom linearization, the software linearizes the model by perturbing the block inputs and states and measuring the response to these perturbations. For each input and state, the default perturbation level is:

- $10^{-5}(1 + |x|)$ for double-precision values.
- $0.005(1 + |x|)$ for single-precision values.

Here, x is the value of the corresponding input or state at the operating point. For information on how to change perturbation levels for individual blocks, see “Change Perturbation Level of Blocks Perturbed During Linearization” on page 2-148.

For more information, see “Linearize Nonlinear Models” on page 2-3 and “Exact Linearization Algorithm” on page 2-175

Full-Model Numerical Perturbation

You can linearize your system using full-model numerical perturbation, where the software computes the linearization of the full model by perturbing the values of root-level inputs and states. To do so, create a `linearizeOptions` object and set the `LinearizationAlgorithm` property to one of the following:

- `'numericalpert'` — Perturb the inputs and states using forward differences; that is, by adding perturbations to the input and state values. This perturbation method is typically faster than the `'numericalpert2'` method.
- `'numericalpert2'` — Perturb the inputs and states using central differences; that is, by perturbing the input and state values in both positive and negative directions. This perturbation method is typically more accurate than the `'numericalpert'` method.

For each input and state, the software perturbs the model and computes a linear model based on the model response to these perturbations. You can configure the state and input perturbation levels using the `NumericalPertRel` linearization options.

Block-by-block linearization has several advantages over full-model numerical perturbation:

- Most Simulink blocks have a preprogrammed linearization that provides an exact linearization of the block.
- You can use linear analysis points to specify a portion of the model to linearize.
- You can configure blocks to use custom linearizations without affecting your model simulation.
- Structurally nonminimal states are automatically removed.
- You can specify linearizations that include uncertainty (requires Robust Control Toolbox software).
- You can obtain detailed diagnostic information.
- When linearizing multirate models, you can use different rate conversion methods. Full-model numerical perturbation can only use zero-order-hold rate conversion.

For more information, see “Linearize Nonlinear Models” on page 2-3 and “Exact Linearization Algorithm” on page 2-175.

Alternatives

As an alternative to the `linearize` function, you can linearize models using one of the following methods:

- To interactively linearize models, use the **Model Linearizer** app. For an example, see “Linearize Simulink Model at Model Operating Point” on page 2-54.
- To obtain multiple transfer functions without modifying the model or creating an analysis point set for each transfer function, use an `slLinearizer` interface. For an example, see “Vary Parameter Values and Obtain Multiple Transfer Functions” on page 3-21.

Although both Simulink Control Design software and the Simulink `linmod` function perform block-by-block linearization, Simulink Control Design linearization functionality has a more flexible user interface and uses Control System Toolbox numerical algorithms. For more information, see “Linearization Using Simulink Control Design Versus Simulink” on page 2-8.

See Also

`linearizeOptions` | `slLinearizer` | `findop` | **Model Linearizer**

Topics

“Linearize Simulink Model at Model Operating Point” on page 2-54
“Linearize at Trimmed Operating Point” on page 2-67
“Batch Linearize Model for Parameter Variations at Single Operating Point” on page 3-13

Introduced in R2006a

linearizeOptions

Set linearization options

Syntax

```
options = linearizeOptions
options = linearizeOptions(Name,Value)
```

Description

`options = linearizeOptions` returns the default linearization option set.

`options = linearizeOptions(Name,Value)` returns an option set with additional options specified by one or more `Name,Value` pair arguments.

Examples

Create Option Set for Linearization

Create a linearization option set that sets the rate conversion method to the Tustin method with prewarping at a frequency of 10 rad/s. Additionally, instruct the linearization not to omit blocks outside the linearization path.

```
options = linearizeOptions('RateConversionMethod','prewarp',...
                           'PreWarpFreq',10,...
                           'BlockReduction','off');
```

Alternatively, use dot notation to set the values of `options`.

```
options = linearizeOptions;
options.RateConversionMethod = 'prewarp';
options.PreWarpFreq = 10;
options.BlockReduction = 'off';
```

Input Arguments

Name-Value Pair Arguments

Specify optional comma-separated pairs of `Name,Value` arguments. `Name` is the argument name and `Value` is the corresponding value. `Name` must appear inside quotes. You can specify several name and value pair arguments in any order as `Name1,Value1,...,NameN,ValueN`.

Example: `'RateConversionMethod','prewarp'` sets the rate conversion method to the Tustin method with prewarping.

`LinearizationAlgorithm` — Algorithm used for linearization
`'blockbyblock'` (default) | `'numericalpert'`

Algorithm used for linearization, specified as the comma-separated pair consisting of `'LinearizationAlgorithm'` and one of the following:

- `'blockbyblock'` — Individually linearize each block in the model, and combine the results to produce the linearization of the specified system.
- `'numericalpert'` — Full-model numerical-perturbation linearization in which root-level inports and states are perturbed using forward differences; that is, by adding perturbations to the input and state values. This perturbation method is typically faster than the `'numericalpert2'` method.
- `'numericalpert2'` — Full-model numerical-perturbation linearization in which root-level inports and states are numerically perturbed using central differences; that is, by perturbing the input and state values in both positive and negative directions. This perturbation method is typically more accurate than the `'numericalpert'` method.

The numerical perturbation linearization methods ignore linear analysis points set in the model and use root-level inports and outports instead.

Block-by-block linearization has several advantages over full-model numerical perturbation:

- Many Simulink blocks have a preprogrammed exact linearization.
- You can use linear analysis points to specify a portion of the model to linearize.
- You can configure blocks to use custom linearizations without affecting your model simulation.
- Structurally nonminimal states are automatically removed.
- You can specify linearizations that include uncertainty (requires Robust Control Toolbox software).
- You can obtain detailed diagnostic information about the linearization.

`SampleTime` — Sample time of linearization result
`-1` (default) | `0` | positive scalar

Sample time of linearization result, specified as the comma-separated pair consisting of `'SampleTime'` and one of the following:

- `-1` — Set the sample time to the least common multiple of the nonzero sample times in the model.
- `0` — Create a continuous-time model.
- Positive scalar — Specify the sample time for discrete-time systems.

`UseFullBlockNameLabels` — Flag indicating whether to truncate names of I/Os and states
`'off'` (default) | `'on'`

Flag indicating whether to truncate names of I/Os and states in the linearized model, specified as the comma-separated pair consisting of `'UseFullBlockNameLabels'` and either:

- `'off'` — Use truncated names for the I/Os and states in the linearized model.
- `'on'` — Use the full block path to name the I/Os and states in the linearized model.

`UseBusSignalLabels` — Flag indicating whether to use bus signal channel numbers or names
`'off'` (default) | `'on'`

Flag indicating whether to use bus signal channel numbers or names to label the I/Os in the linearized model, specified as the comma-separated pair consisting of `'UseBusSignalLabels'` and one of the following:

- `'off'` — Use bus signal channel numbers to label I/Os on bus signals in the linearized model.
- `'on'` — Use bus signal names to label I/Os on bus signals in the linearized model. Bus signal names appear in the results when the I/O points are located at the output of the following blocks:
  - Root-level inport block containing a bus object
  - Bus creator block
  - Subsystem block whose source traces back to the output of a bus creator block
  - Subsystem block whose source traces back to a root-level inport by passing through only virtual or nonvirtual subsystem boundaries

`StoreOffsets` — Flag indicating whether to compute linearization offsets
`false` (default) | `true`

Flag indicating whether to compute linearization offsets for inputs, outputs, states, and state derivatives or updated states, specified as the comma-separated pair consisting of `'StoreOffsets'` and one of the following:

- `false` — Do not compute linearization offsets.
- `true` — Compute linearization offsets.

You can configure an LPV System block using linearization offsets. For an example, see “Approximate Nonlinear Behavior Using Array of LTI Systems” on page 3-70

`StoreAdvisor` — Flag indicating whether to store diagnostic information
`false` (default) | `true`

Flag indicating whether to store diagnostic information during linearization, specified as the comma-separated pair consisting of `'StoreAdvisor'` and one of the following:

- `false` — Do not store linearization diagnostic information.
- `true` — Store linearization diagnostic information.

Linearization commands store and return diagnostic information in a `LinearizationAdvisor` object. For an example of troubleshooting linearization results using a `LinearizationAdvisor` object, see “Troubleshoot Linearization Results at Command Line” on page 4-28.

`BlockReduction` — Flag indicating whether to omit blocks that are not on the linearization path
`'on'` (default) | `'off'`

Flag indicating whether to omit blocks that are not in the linearization path, specified as the comma-separated pair consisting of `'BlockReduction'` and one of the following:

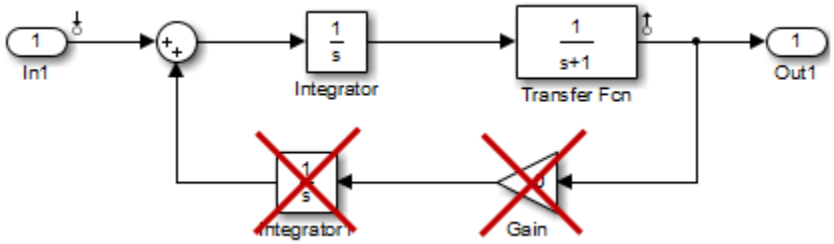
- `'on'` — Return a linearized model that does not include states from noncontributing linearization paths.
- `'off'` — Return a linearized model that includes all the states of the model.

Dead linearization paths can include:

- Blocks that linearize to zero.
- Switch blocks that are not active along the path.

- Disabled subsystems.
- Signals marked as open-loop linearization points.

For example, if this flag set to `'on'`, the linearization result of the model shown in the following figure includes only two states. It does not include states from the two blocks outside the linearization path. These states do not appear because these blocks are on a dead linearization path with a block that linearizes to zero (the zero gain block).



This option applies only when `LinearizationAlgorithm` is `'blockbyblock'`. `BlockReduction` is always treated as `'on'` when `LinearizationAlgorithm` is `'numericalpert'` or `'numericalpert2'`.

`IgnoreDiscreteStates` — Flag indicating whether to remove discrete-time states
`'off'` (default) | `'on'`

Flag indicating whether to remove discrete-time states from the linearization, specified as the comma-separated pair consisting of `'IgnoreDiscreteStates'` and one of the following:

- `'off'` — Always include discrete-time states.
- `'on'` — Remove discrete states from the linearization. Use this option when performing continuous-time linearization (`SampleTime = 0`) to accept the `D` value for all blocks with discrete-time states.

This option applies only when `LinearizationAlgorithm` is `'blockbyblock'`.

`RateConversionMethod` — Rate conversion method
`'zoh'` (default) | `'tustin'` | `'prewarp'` | `'upsampling_zoh'` | `'upsampling_tustin'` | `'upsampling_prewarp'`

Method used for rate conversion when linearizing a multirate system, specified as the comma-separated pair consisting of `'RateConversionMethod'` and one of the following:

- `'zoh'` — Zero-order hold rate conversion method
- `'tustin'` — Tustin (bilinear) method
- `'prewarp'` — Tustin method with frequency prewarp. When you use this method, set the `PreWarpFreq` option to the desired prewarp frequency.
- `'upsampling_zoh'` — Upsample discrete states when possible, and use `'zoh'` otherwise.
- `'upsampling_tustin'` — Upsample discrete states when possible, and use `'tustin'` otherwise.
- `'upsampling_prewarp'` — Upsample discrete states when possible, and use `'prewarp'` otherwise. When you use this method, set the `PreWarpFreq` option to the desired prewarp frequency.

For more information on rate conversion and linearization of multirate models, see:

- “Linearize Multirate Models” on page 2-139
- “Linearize Models Using Different Rate Conversion Methods” on page 2-145
- “Continuous-Discrete Conversion Methods”

Note If you use a rate conversion method other than `'zoh'`, the converted states no longer have the same physical meaning as the original states. As a result, the state names in the resulting LTI system change to `'?'`.

This option applies only when `LinearizationAlgorithm` is `'blockbyblock'`.

`PreWarpFreq` — Prewarp frequency
`0` (default) | positive scalar

Prewarp frequency in rad/s, specified as the comma-separated pair consisting of `'PreWarpFreq'` and a nonnegative scalar. This option applies only when `RateConversionMethod` is either `'prewarp'` or `'upsampling_prewarp'`.

`UseExactDelayModel` — Flag indicating whether to compute linearization with exact delays
`'off'` (default) | `'on'`

Flag indicating whether to compute linearization with exact delays, specified as the comma-separated pair consisting of `'UseExactDelayModel'` and one of the following:

- `'off'` — Return a linear model with approximate delays.
- `'on'` — Return a linear model with exact delays.

This option applies only when `LinearizationAlgorithm` is `'blockbyblock'`.

`AreParamsTunable` — Flag indicating whether to recompile the model when varying parameter values
`true` (default) | `false`

Flag indicating whether to recompile the model when varying parameter values for linearization, specified as the comma-separated pair consisting of `'AreParamsTunable'` and one of the following:

- `true` — Do not recompile the model when all varying parameters are tunable. If any varying parameters are not tunable, recompile the model for each parameter grid point, and issue a warning message.
- `false` — Recompile the model for each parameter grid point. Use this option when you vary the values of nontunable parameters.

For more information about model compilation when you linearize with parameter variation, see “Batch Linearization Efficiency When You Vary Parameter Values” on page 3-7.

`NumericalPertRel` — Numerical perturbation level
`1e-5` (default) | positive scalar

Numerical perturbation level, specified as the comma-separated pair consisting of `'NumericalPertRel'` and a positive scalar. This option applies only when `LinearizationAlgorithm` is `'numericalpert'` or `'numericalpert2'`.

The perturbation levels for the system states are:

$$\text{NumericalPertRel} + 10^{-3} \times \text{NumericalPertRel} \times |x|$$

The perturbation levels for the system inputs are:

$$\text{NumericalPertRel} + 10^{-3} \times \text{NumericalPertRel} \times |u|$$

You can override these values using the `NumericalXPert` or `NumericalUPert` options.

`NumericalXPert` — State perturbation levels
`[]` (default) | operating point object

State perturbation levels, specified as the comma-separated pair consisting of `'NumericalXPert'` and an operating point object. This option applies only when `LinearizationAlgorithm` is `'numericalpert'` or `'numericalpert2'`.

To set individual perturbation levels for each state:

1. Create an operating point object for the model using the `operpoint` command.

   ```
   xPert = operpoint('watertank');
   ```

2. Set the state values in the operating point object to the perturbation levels.

   ```
   xPert.States(1).x = 2e-3;
   xPert.States(2).x = 3e-3;
   ```

3. Set the value of the `NumericalXPert` option to the operating point object.

   ```
   opt = linearizeOptions('LinearizationAlgorithm','numericalpert');
   opt.NumericalXPert = xPert;
   ```

If `NumericalXPert` is empty, `[]`, the linearization algorithm derives the state perturbation levels using `NumericalPertRel`.

`NumericalUPert` — Input perturbation levels
`[]` (default) | operating point object

Input perturbation levels, specified as the comma-separated pair consisting of `'NumericalUPert'` and an operating point object. This option applies only when `LinearizationAlgorithm` is `'numericalpert'` or `'numericalpert2'`.

To set individual perturbation levels for each input:

1. Create an operating point object for the model using the `operpoint` command.

   ```
   uPert = operpoint('watertank');
   ```

2. Set the input values in the operating point object to the perturbation levels.

   ```
   uPert.Inputs(1).x = 3e-3;
   ```

3. Set the value of the `NumericalUPert` option to the operating point object.

   ```
   opt = linearizeOptions('LinearizationAlgorithm','numericalpert');
   opt.NumericalUPert = uPert;
   ```

If `NumericalUPert` is empty, `[]`, the linearization algorithm derives the input perturbation levels using `NumericalPertRel`.

Output Arguments

`options` — Linearization options
`linearizeOptions` option set

Linearization options, returned as a `linearizeOptions` option set.

See Also

`linearize` | `slLinearizer` | `ulinearize` | `linlft`

Introduced in R2013b

linio

Create linear analysis point for Simulink model, Linear Analysis Plots block, or Model Verification block

Syntax

```
io = linio(block,port)
io = linio(block,port,type)
io = linio(block,port,type,[],busElement)
```

Description

`io = linio(block,port)` creates a linearization I/O object that represents an input perturbation analysis point for the signal that originates from the specified output `port` of a Simulink `block`.

`io = linio(block,port,type)` creates an analysis point of the specified `type`.

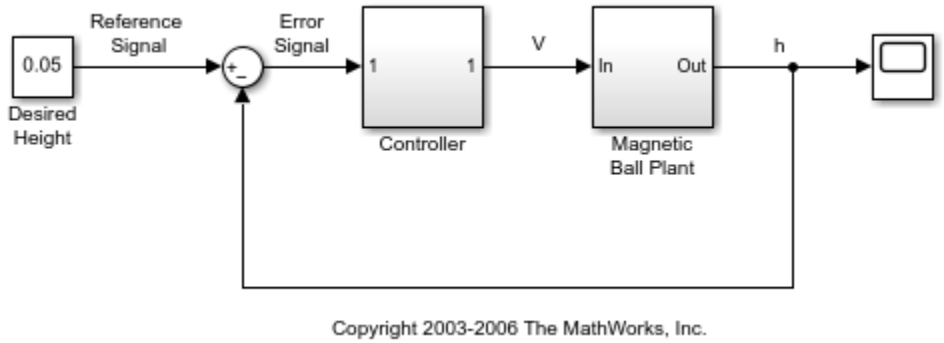
`io = linio(block,port,type,[],busElement)` creates an analysis point for an element of a bus signal.

Examples

Create Analysis Points for Simulink Model

Open Simulink model.

```
open_system('magball')
```



To specify multiple analysis points for linearization, create a vector of linearization I/O objects.

Create an input perturbation analysis point at the output port of the Controller block.

```
io(1) = linio('magball/Controller',1);
```

Create an open-loop output analysis point at the output of the Magnetic Ball Plant block. An open-loop output point is an output measurement followed by a loop opening.

```
io(2) = linio('magball/Magnetic Ball Plant',1,'openoutput');
```

View the specified analysis points.

```
io
```

```
1x2 vector of Linearization IOs:
--------------------------
1. Linearization input perturbation located at the following signal:
- Block: magball/Controller
- Port: 1
2. Linearization open-loop output located at the following signal:
- Block: magball/Magnetic Ball Plant
- Port: 1
```

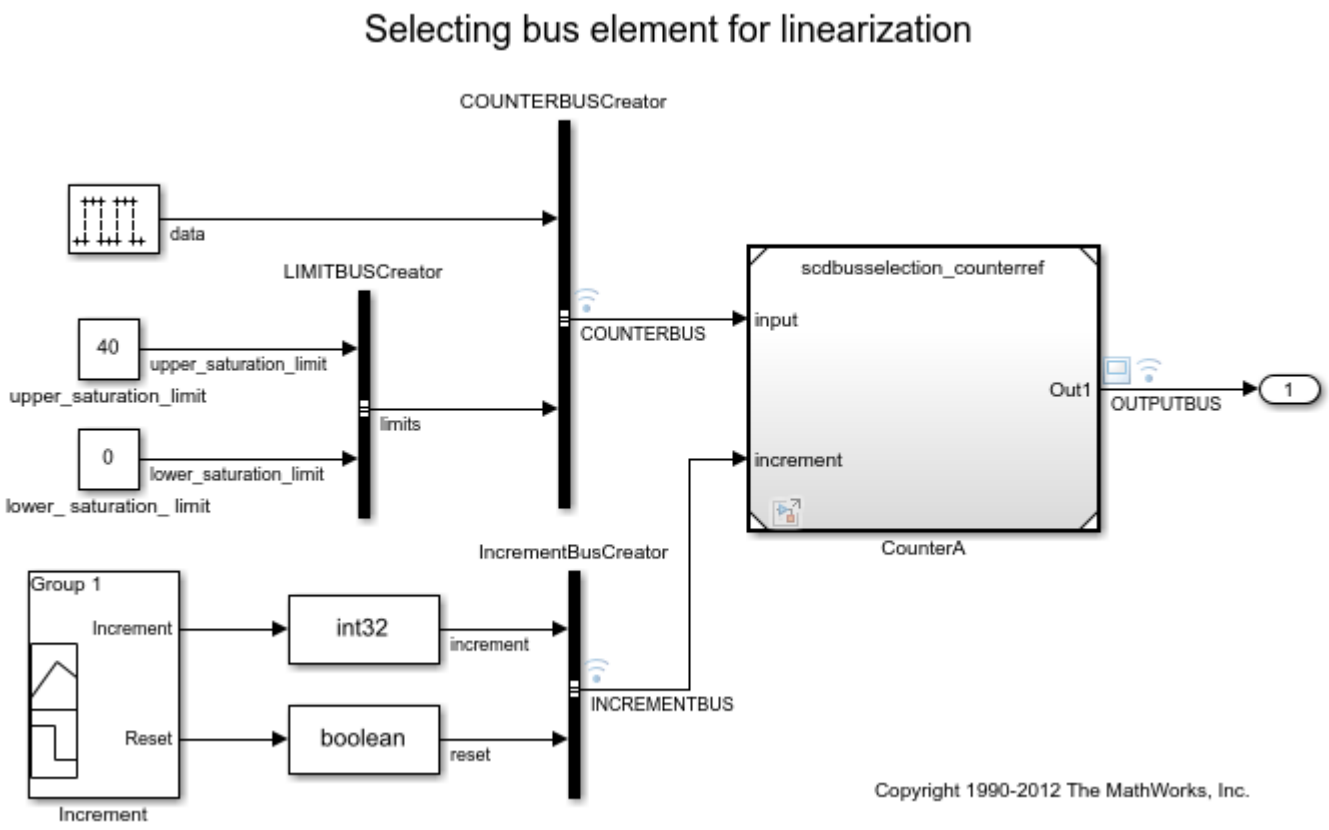
You can use these analysis points to linearize only the Magnetic Ball Plant subsystem. To do so, pass `io` to the `linearize` command or to an `slLinearizer` interface.

Unlike specifying analysis points directly in the Simulink model, when you create analysis points using `linio`, no annotations are added to the model.

Select Individual Bus Element as Analysis Point

Open Simulink model.

```
mdl = 'scdbusselection';
open_system(mdl)
```



The `COUNTERBUS` signal, which originates from the COUNTERBUSCreator block, contains multiple bus elements.

Specify the `upper_saturation_limit` bus element as a linearization input. Select this element using dot notation, since it is within the nested `limits` bus.

```
io = linio('scdbusselection/COUNTERBUSCreator',1,'input',[],...
           'limits.upper_saturation_limit');
```

Input Arguments

`block` — Simulink block

character vector | string

Simulink block from which the analysis point originates, specified as a character vector or string that contains its full block path. For example, to mark an analysis point at an output of the Controller block in the `magball` model, specify `block` as `'magball/Controller'`.

`port` — Output port

positive integer

Output port of `block` from which the analysis point originates, specified as a positive integer.

`port` must be a valid port number for the specified `block`.

`type` — Analysis point type
`'input'` (default) | `'output'` | `'loopbreak'` | ...

Analysis point type, specified as one of the following:

- `'input'` — Input perturbation
- `'output'` — Output measurement
- `'loopbreak'` — Loop break
- `'openinput'` — Open-loop input
- `'openoutput'` — Open-loop output
- `'looptransfer'` — Loop transfer
- `'sensitivity'` — Sensitivity
- `'compsensitivity'` — Complementary sensitivity

For more information on analysis point types, see “Specify Portion of Model to Linearize” on page 2-10.

`busElement` — Bus element name
character vector | string

Bus element name, specified as a character vector or string. When adding elements within a nested bus structure, use dot notation to access the elements of the nested bus. For an example, see “Select Individual Bus Element as Analysis Point” on page 18-169.

Output Arguments

`io` — Analysis point
linearization I/O object

Analysis point, returned as a linearization I/O object. Use `io` to specify a linearization input, output, or loop opening when using the `linearize` command. For more information, see “Specify Portion of Model to Linearize” on page 2-10.

Each linearization I/O object has the following properties:

Property	Description
`Active`	Flag indicating whether to use the analysis point for linearization, specified as one of the following: • `'on'` — Use the analysis point for linearization. This value is the default option. • `'off'` — Do not use the analysis point for linearization. Use this option if you have an existing set of analysis points and you want to linearize a model with a subset of these points.
`Block`	Full block path of the block with which the analysis point is associated, specified as a character vector.
`PortNumber`	Output port with which the analysis point is associated, specified as an integer.

Property	Description
`Type`	Analysis point type, specified as one of the following: • `'input'` — Input perturbation • `'output'` — Output measurement • `'loopbreak'` — Loop break • `'openinput'` — Open-loop input • `'openoutput'` — Open-loop output • `'looptransfer'` — Loop transfer • `'sensitivity'` — Sensitivity • `'compsensitivity'` — Complementary sensitivity For more information on analysis point types, see “Specify Portion of Model to Linearize” on page 2-10.
`BusElement`	Bus element name with which the analysis point is associated, specified as a character vector or `''` if the analysis point is not a bus element.
`Description`	User-specified description of the analysis point, which you can set for convenience, specified as a character vector.

Alternative Functionality

Model Linearizer

You can interactively configure analysis points using the **Model Linearizer**. For more information see, “Specify Portion of Model to Linearize in Model Linearizer” on page 2-22.

Simulink Model

You can also specify analysis points directly in a Simulink model. When you do so, the analysis points are saved within the model. For more information, see “Specify Portion of Model to Linearize in Simulink Model” on page 2-17.

`slLinearizer` and `slTuner` Interfaces

If you want to obtain multiple open-loop or closed-loop transfer functions from the linearized system without recompiling the model, you can specify linear analysis points using an `slLinearizer` interface. For more information, see “Mark Signals of Interest for Batch Linearization” on page 3-9. Similarly, if you want to tune a control system and obtain multiple open-loop or closed-loop transfer functions from the resulting system, you can specify linear analysis points using an `slTuner` interface. For more information, see “Mark Signals of Interest for Control System Analysis and Design” on page 2-38.

See Also

`getlinio` | `setlinio` | `linearize`

Topics

“Specify Portion of Model to Linearize” on page 2-10

Introduced before R2006a

linlft

Linearize model while removing contribution of specified blocks

Syntax

```
lin_fixed = linlft(sys,io,blocks)
[lin_fixed,lin_blocks] = linlft( ___ )
[ ___ ] = linlft( ___ ,opt)
```

Description

`lin_fixed = linlft(sys,io,blocks)` linearizes the Simulink model named `sys` while removing the contribution of certain blocks. Specify `sys` as a character vector or string. Specify the full block path of the blocks to ignore in the cell array of character vectors or string array called `blocks`. The linearization occurs at the operating point specified in the Simulink model, which includes the ignored blocks. You can optionally specify linearization points (linear analysis points) in the I/O object `io`. The resulting linear model `lin_fixed` has this form:

The top channels In and Out correspond to the linearization points you specify in the I/O object `io`. The remaining channels correspond to the connection to the ignored blocks.

When you use `linlft` and specify the `'block-by-block'` linearization algorithm in `linearizeOptions`, you can use all the variations of the input arguments for `linearize`.

You can linearize the ignored blocks separately using `linearize`, and then combine the linearization results using `linlftfold`.

`[lin_fixed,lin_blocks] = linlft( ___ )` returns the linearizations for each of the blocks specified in `blocks`. If `blocks` contains a single block path, `lin_blocks` is a single state-space (`ss`) model. If `blocks` is an array identifying multiple blocks, `lin_blocks` is a cell array of state-space models. The full block path for each block in `lin_blocks` is stored in the `Notes` property of the state-space model.

`[ ___ ] = linlft( ___ ,opt)` uses additional linearization options, specified as a `linearizeOptions` option set.

Examples

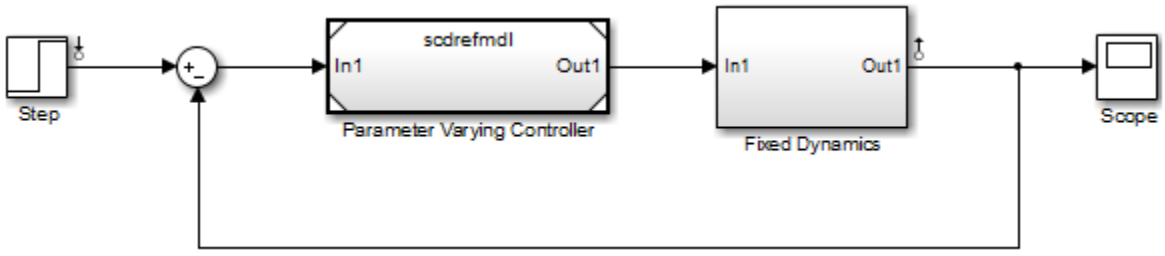
Combine Linearizations for Portions of Model

In this example, the `scdtopmdl` model contains two subsystems in the feedforward path.

- Fixed portion, which contains everything except the Parameter Varying Controller model reference
- Parameter Varying Controller model, which references the `scdrefmdl` model

Open the top-level model.

```
topmdl = 'scdtopmdl';
open_system(topmdl)
```



Linearize this model without the Parameter Varying Controller block.

```
io = getlinio(topmdl);
blocks = {'scdtopmdl/Parameter Varying Controller'};
sys_fixed = linlft(topmdl,io,blocks);
```

Linearize the controller model.

```
refmdl = 'scdrefmdl';
load_system(refmdl);
sys_pv = linearize(refmdl);
```

Combine the linearization results.

```
BlockSubs(1) = struct('Name',blocks{1},'Value',sys_pv);
```

See Also

`linlftfold` | `linearize` | `linio` | `getlinio` | `operpoint` | `linearizeOptions`

Introduced in R2009b

linlftfold

Combine linearization results from specified blocks and model

Syntax

```
lin = linlftfold(lin_fixed,blocksubs)
```

Description

`lin = linlftfold(lin_fixed,blocksubs)` combines the following linearization results into one linear model `lin`:

- Linear model `lin_fixed`, which does not include the contribution of specified blocks in your Simulink model.

  Compute `lin_fixed` using `linlft`.
- Block linearizations for the blocks excluded from `lin_fixed`

  You specify the block linearizations in a structure array `blocksubs`, which contains two fields:

  - `'Name'` is a character vector or string specifying the block path of the Simulink block to replace.
  - `'Value'` is the value of the linearization for each block.

Examples

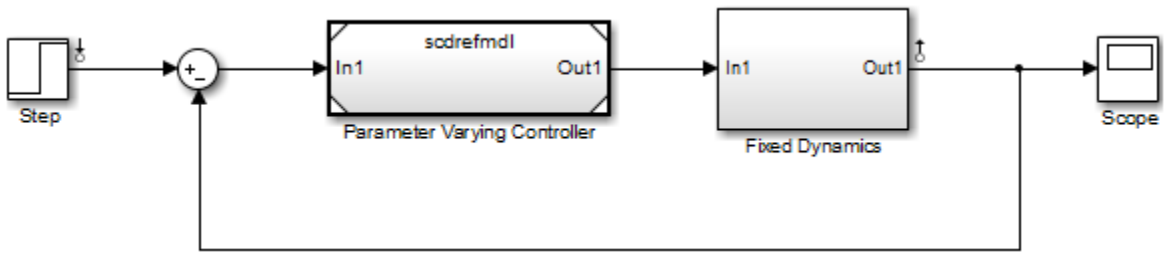
Combine Linearizations for Portions of Model

In this example, the `scdtopmdl` model contains two subsystems in the feedforward path.

- Fixed portion, which contains everything except the Parameter Varying Controller model reference
- Parameter Varying Controller model, which references the `scdrefmdl` model

Open the top-level model.

```
topmdl = 'scdtopmdl';
open_system(topmdl)
```



Linearize this model without the Parameter Varying Controller block.

```
io = getlinio(topmdl);
blocks = {'scdtopmdl/Parameter Varying Controller'};
sys_fixed = linlft(topmdl,io,blocks);
```

Linearize the controller model.

```
refmdl = 'scdrefmdl';
load_system(refmdl);
sys_pv = linearize(refmdl);
```

Combine the linearization results.

```
BlockSubs(1) = struct('Name',blocks{1},'Value',sys_pv);
```

See Also

`linlft` | `linearize` | `linio` | `getlinio` | `operpoint`

Introduced in R2009b

operpoint

Create operating point for Simulink model

Syntax

```
op = operpoint(mdl)
```

Description

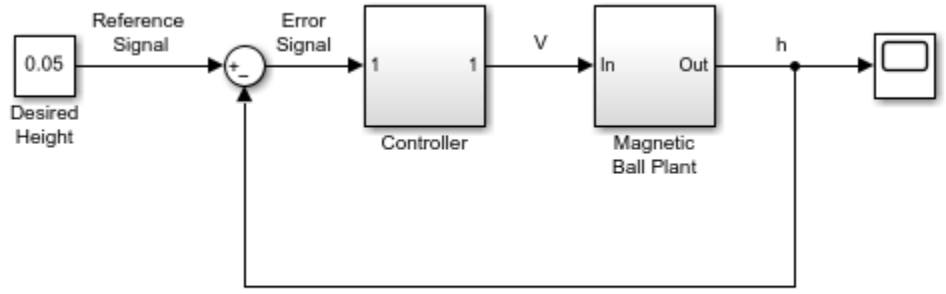
`op = operpoint(mdl)` returns the operating point of Simulink model `mdl`. You can compute a linear model of your system at this operating point using the `linearize` function.

Examples

Create Operating Point for Simulink Model

Open Simulink model.

```
open_system('magball')
```



Create operating point for the model.

```
op = operpoint('magball')
```

```
op =


 Operating point for the Model magball.
 (Time-Varying Components Evaluated at time t=0)

States:
----------
      <strong>x</strong>
    <strong>______</strong>

(1.) magball/Controller/PID Controller/Filter/Cont. Filter/Filter
```

```
         0
(2.) magball/Controller/PID Controller/Integrator/Continuous/Integrator
    14.007
(3.) magball/Magnetic Ball Plant/Current
    7.0036
(4.) magball/Magnetic Ball Plant/dhdt
         0
(5.) magball/Magnetic Ball Plant/height
      0.05

Inputs: None
----------
```

`op` lists each block in the model that has states. There are no root-level inports in this model, therefore `op` does not contain inputs.

Copy an Operating Point

You can create new operating-point variables in three ways:

- Using the `operpoint` function
- Using assignment with the equals (=) operator
- Using the `copy` function

Using the = operator results in linked variables that both point to the same underlying data. Using the `copy` function results in an independent operating-point object. In this example, create operating-point objects both ways, and examine their behavior.

```
mdl = 'watertank';
open_system(mdl)
op1 = operpoint(mdl)

op1 =
 Operating point for the Model watertank.
 (Time-Varying Components Evaluated at time t=0)

States:
----------
    x
    -

(1.) watertank/PID Controller/Integrator/Continuous/Integrator
    0
(2.) watertank/Water-Tank System/H
    1

Inputs: None
----------
```

Create a new operating-point object using assignment with the = operator.

```
op2 = op1;
```

`op2` is an operating-point object that points to the same underlying data as `op1`. Because of this link, you cannot independently change properties of the two operating-point objects. To see this, change a

property of `op2`. For instance, change the value for the first state from 0 to 2. The change shows in the `States` section of the display.

```
op2.States(1).x = 2
```

```
op2 =
 Operating point for the Model watertank.
 (Time-Varying Components Evaluated at time t=0)

States:
----------
    x
    -

(1.) watertank/PID Controller/Integrator/Continuous/Integrator
    2
(2.) watertank/Water-Tank System/H
    1

Inputs: None
----------
```

Examine the display of `op1` to see that the corresponding property value of `op1` also changes from 0 to 2.

```
op1
```

```
op1 =
 Operating point for the Model watertank.
 (Time-Varying Components Evaluated at time t=0)

States:
----------
    x
    -

(1.) watertank/PID Controller/Integrator/Continuous/Integrator
    2
(2.) watertank/Water-Tank System/H
    1

Inputs: None
----------
```

To create an independent copy of an operating-point object, use the `copy` function.

```
op3 = copy(op1);
```

Now, when you change a property of `op3`, `op1` does not change. For instance, change the value for the first state from 2 to 4.

```
op3.States(1).x = 4
```

```
op3 =
 Operating point for the Model watertank.
 (Time-Varying Components Evaluated at time t=0)

States:
----------
```

```
    x
    -

(1.) watertank/PID Controller/Integrator/Continuous/Integrator
    4
(2.) watertank/Water-Tank System/H
    1

Inputs: None
----------
```

In `op1`, the corresponding value remains 2.

```
op1.States(1).x
```

```
ans = 2
```

This copy behavior occurs because the operating-point object is a *handle object*. For more information about handle objects, see “Handle Object Behavior”.

Input Arguments

`mdl` — Simulink model name

character vector | string

Simulink model name, specified as a character vector or string. The model must be in the current working folder or on the MATLAB path.

Output Arguments

`op` — Operating point

`OperatingPoint` object

Operating point, returned as an `OperatingPoint` object with the following properties.

Property	Description
`Model`	Simulink model name, returned as a character vector.

<table>
<tr><th>Property</th><th>Description</th></tr>
<tr><td>States</td><td>State operating point, returned as a vector of state objects. Each entry in States represents the supported states of one Simulink block.

For a list of supported states for operating point objects, see “Simulink Model States Included in Operating Point Object” on page 1-3.

Note If the block has multiple named continuous states, States contains one structure for each named state.

Each state object has the following fields:

<table>
<tr><th>Field</th><th>Description</th></tr>
<tr><td>Nx (read only)</td><td>Number of states in the block</td></tr>
<tr><td>Block</td><td>Block path, returned as a character vector.</td></tr>
<tr><td>StateName</td><td>State name</td></tr>
<tr><td>x</td><td>Values of all supported block states, returned as a vector of length Nx.</td></tr>
<tr><td>Ts</td><td>Sample time and offset of each supported block state, returned as a vector. For continuous-time systems, Ts is zero.</td></tr>
<tr><td>SampleType</td><td>State time rate, returned as one of the following:
• 'CSTATE' — Continuous-time state
• 'DSTATE' — Discrete-time state</td></tr>
<tr><td>inReferencedModel</td><td>Flag indicating whether the block is inside a reference model, returned as one of the following:
• 1 — Block is inside a reference model.
• 0 — Block is in the current model file.</td></tr>
<tr><td>Description</td><td>Block state description, returned as a character vector.</td></tr>
</table>
</td></tr>
<tr><td>Inputs</td><td>Input level at the operating point, returned as a vector of input objects. Each entry in Inputs represents the input levels of one root-level inport block in the model.

Each input object has the following fields:

<table>
<tr><th>Field</th><th>Description</th></tr>
<tr><td>Nu (read only)</td><td>Number of inport block signals</td></tr>
<tr><td>Block</td><td>Inport block name</td></tr>
<tr><td>PortDimensions</td><td>Dimension of signals accepted by the inport</td></tr>
<tr><td>u</td><td>Inport block input levels at the operating point, returned as a vector of length Nu.</td></tr>
<tr><td>Description</td><td>Inport block input description, returned as a character vector.</td></tr>
</table>
</td></tr>
</table>

Property	Description
`Time`	Times at which any time-varying functions in the model are evaluated, returned as a vector.
`Version`	Object version number

Tips

- You can create new operating points of in three ways:
  - Construct a new object using the `operpoint` function.
  - Create a new variable by assignment with the equals (`=`) operator.
  - Copy an operating point object using the `copy` command.

  Using `operpoint` or `copy` creates a new, independent object. When you use assignment, there is a link between the old and new variable. For an example, see “Copy an Operating Point” on page 18-179.

Alternative Functionality

The `operpoint` function returns an operating point with the initial state and input values of the model. To create an operating point that meets your application specifications, use the `findop` function. For more information, see “Compute Steady-State Operating Points” on page 1-5.

Compatibility Considerations

`PortWidth` property of operating point inputs will be removed
Not recommended starting in R2021b

The input `PortWidth` property of operating points will be removed in a future release. Use the new `Nu` property instead.

To update your code, change instances of `PortWidth` to `Nu` as shown in the following table.

Not Recommended	Recommended
`op = operpoint('scdplane');` `numIn = op.Inputs(1).PortWidth;`	`op = operpoint('scdplane');` `numIn = op.Inputs(1).Nu;`

See Also

`linearize` | `operspec` | `update` | `copy` | `findop`

Topics
“About Operating Points” on page 1-2

Introduced before R2006a

operspec

Operating point specifications

Syntax

```
opspec = operspec(mdl)
opspec = operspec(mdl,dim)
```

Description

`opspec = operspec(mdl)` returns the default operating point specification object for the Simulink model `mdl`. Use `opspec` for steady-state operating point trimming using `findop`.

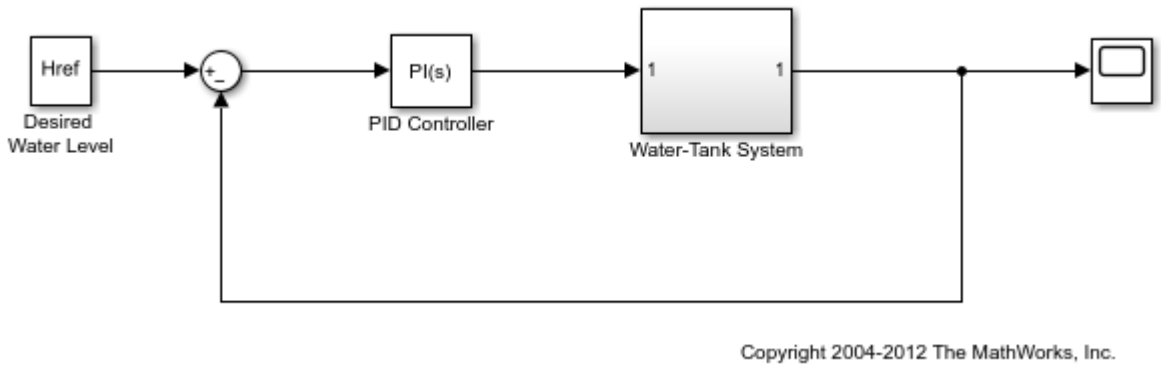
`opspec = operspec(mdl,dim)` returns an array of default operating point specification objects with the specified dimensions, `dim`.

Examples

Create Operating Point Specification Object

Open Simulink model.

```
sys = 'watertank';
open_system(sys)
```



Create the default operating point specification object for the model.

```
opspec = operspec(sys)
```

```
opspec =

 Operating point specification for the Model watertank.
 (Time-Varying Components Evaluated at time t=0)
```

```
States:
----------
    <strong>x</strong>    <strong>Known</strong>    <strong>SteadyState</strong>    <strong>Min</
    <strong>_</strong>    <strong>_____</strong>    <strong>___________</strong>    <strong>____

(1.) watertank/PID Controller/Integrator/Continuous/Integrator
    0    false       true        -Inf    Inf    -Inf      Inf
(2.) watertank/Water-Tank System/H
    1    false       true           0    Inf    -Inf      Inf

Inputs: None
----------

Outputs: None
----------
```

`opspec` contains specifications for the two states in the model. Since the model has no root level inports or outports, `opspec` does not contain input or output specifications. To add output specifications, use `addoutputspec`.

Modify the operating point specifications for each state using dot notation. For example, configure the first state to:

- Be at steady state.
- Have a lower bound of `0`.
- Have an initial value of `2` for trimming.

```
opspec.States(1).SteadyState = 1;
opspec.States(1).x = 2;
opspec.States(1).Min = 0;
```

Copy an Operating-Point Specification

You can create new `operspec` variables in three ways:

- Using the `operspec` command
- Using assignment with the equals (`=`) operator
- Using the `copy` command

Using the `=` operator results in linked variables that both point to the same underlying data. Using the `copy` command results in an independent `operspec` object. In this example, create `operspec` objects both ways, and examine their behavior.

```
mdl = 'watertank';
open_system(mdl)
opspec1 = operspec(mdl)
```

```
opspec1 =
 Operating point specification for the Model watertank.
 (Time-Varying Components Evaluated at time t=0)

States:
----------
    x    Known    SteadyState    Min     Max    dxMin    dxMax
```

```
    _    _____    ___________    ____    ___    _____    _____

(1.) watertank/PID Controller/Integrator/Continuous/Integrator
    0    false       true        -Inf    Inf    -Inf      Inf
(2.) watertank/Water-Tank System/H
    1    false       true           0    Inf    -Inf      Inf

Inputs: None
----------

Outputs: None
----------
```

Create a new operating point specification object using assignment with the = operator.

```
opspec2 = opspec1;
```

`opspec2` is an `operspec` object that points to the same underlying data as `opspec1`. Because of this link, you cannot independently change properties of the two `operspec` objects. To see this, change a property of `opspec2`. For instance, change the initial value for the first state from 0 to 2. The change shows in the `States` section of the display.

```
opspec2.States(1).x = 2
```

```
opspec2 =
 Operating point specification for the Model watertank.
 (Time-Varying Components Evaluated at time t=0)

States:
----------
    x    Known    SteadyState    Min     Max    dxMin    dxMax
    _    _____    ___________    ____    ___    _____    _____

(1.) watertank/PID Controller/Integrator/Continuous/Integrator
    2    false       true        -Inf    Inf    -Inf      Inf
(2.) watertank/Water-Tank System/H
    1    false       true           0    Inf    -Inf      Inf

Inputs: None
----------

Outputs: None
----------
```

Examine the display of `opspec1` to see that the corresponding property value of `opspec1` also changes from 0 to 2.

```
opspec1
```

```
opspec1 =
 Operating point specification for the Model watertank.
 (Time-Varying Components Evaluated at time t=0)

States:
----------
    x    Known    SteadyState    Min     Max    dxMin    dxMax
    _    _____    ___________    ____    ___    _____    _____
```

```
(1.) watertank/PID Controller/Integrator/Continuous/Integrator
    2    false       true        -Inf    Inf    -Inf      Inf
(2.) watertank/Water-Tank System/H
    1    false       true           0    Inf    -Inf      Inf

Inputs: None
----------

Outputs: None
----------
```

To create an independent copy of an operating point specification, use the `copy` command.

```
opspec3 = copy(opspec1);
```

Now, when you change a property of `opspec3`, `opspec1` does not change. For instance, change the initial value for the first state from 2 to 4.

```
opspec3.States(1).x = 4
```

```
opspec3 =
 Operating point specification for the Model watertank.
 (Time-Varying Components Evaluated at time t=0)

States:
----------
    x    Known    SteadyState    Min     Max    dxMin    dxMax
    _    _____    ___________    ____    ___    _____    _____

(1.) watertank/PID Controller/Integrator/Continuous/Integrator
    4    false       true        -Inf    Inf    -Inf      Inf
(2.) watertank/Water-Tank System/H
    1    false       true           0    Inf    -Inf      Inf

Inputs: None
----------

Outputs: None
----------
```

In `opspec1`, the corresponding value remains 2.

```
opspec1.States(1).x
```

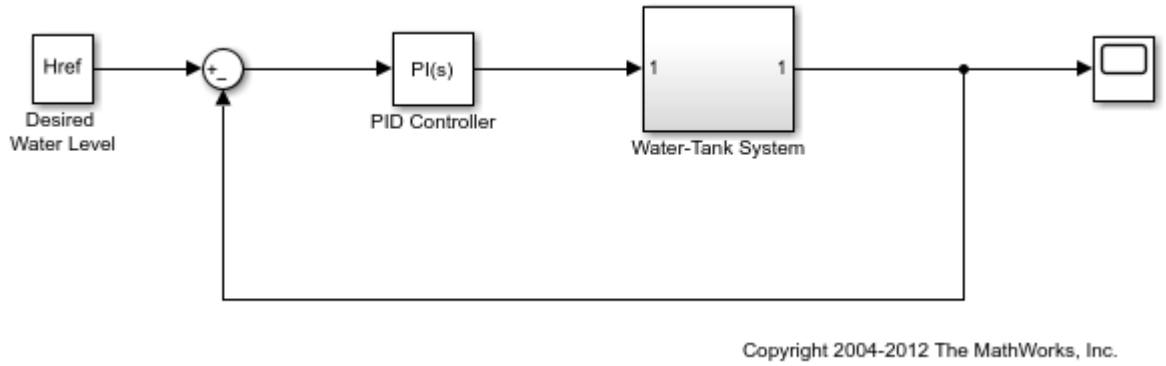
```
ans = 2
```

This copy behavior occurs because `operspec` is a *handle object*. For more information about handle objects, see “Handle Object Behavior”.

Create Array of Operating Point Specification Objects

Open Simulink model.

```
sys = 'watertank';
open_system(sys)
```



Create a 2-by-3 array of operating point specification objects. You can batch trim model at multiple operating points using such arrays.

```
opspec = operspec(sys,[2,3]);
```

Each element of `opspec` contains a default operating point specification object for the model.

Modify the operating point specification objects using dot notation. For example, configure the second state of the specification object in row `1`, column `3`.

```
opspec(1,3).States(2).SteadyState = 1;
opspec(1,3).States(1).x = 2;
```

You can also create multidimensional arrays of operating point specification objects. For example, create a 3-by-4-by-5 array.

```
opspec = operspec(sys,[3,4,5]);
```

Input Arguments

`mdl` — Simulink model

character vector | string

Simulink model name, specified as a character vector or string.

`dim` — Array dimensions

integer | row vector of integers

Array dimensions, specified as one of the following:

- Integer — Create a column vector of `dim` operating point specification objects.
- Row vector of integers — Create an array of operating point specification objects with the dimensions specified by `dim`.

  For example, to create a 4-by-5 array of operating point specification objects, use:

  ```
  opspec = operspec(mdl,[4,5]);
  ```

  To create a multidimensional array of operating point specification objects, specify additional dimensions. For example, to create a 2-by-3-by-4 array, use:

```
opspec = operspec(mdl,[2,3,4]);
```

Output Arguments

opspec — Operating point specifications

`OperatingSpec` object | array of `OperatingSpec` objects

Operating point specifications, returned as an `OperatingSpec` object or an array of such objects.

You can modify the operating point specifications using dot notation. For example, if `opspec` is a single `OperatingSpec` object, `opspec.States(1).x` accesses the state values of the first model state. If `opspec` is an array of `OperatingSpec` objects `opspec(2,3).Inputs(1).u` accesses the input level of the first inport block for the specification in row 2, column 3.

Each `OperatingSpec` object has the following properties.

Property	Description
`Model`	Simulink model name, returned as a character vector.

<table>
<tr><th>Property</th><th>Description</th></tr>
<tr><td>States</td><td>
State operating point specifications, returned as a vector of state specification objects. Each entry in States represents the supported states of one Simulink block.

For a list of supported states for operating point objects, see “Simulink Model States Included in Operating Point Object” on page 1-3. Edit the properties of this object using dot notation or the set function.

Note If the block has multiple named continuous states, States contains one structure for each named state.

Each state specification object has the following fields:

<table>
<tr><th>Field</th><th>Description</th></tr>
<tr><td>Nx (read-only)</td><td>Number of states in the block</td></tr>
<tr><td>Block</td><td>Block path, returned as a character vector.</td></tr>
<tr><td>StateName</td><td>State name</td></tr>
<tr><td>x</td><td>Values of all supported block states, specified as a vector of length Nx.

If the corresponding flag in Known field of States is 1, x contains the known state values. Otherwise, x contains initial guesses for the state values.</td></tr>
<tr><td>Ts</td><td>(Only for discrete-time states) Sample time and offset of each supported block state, returned as a vector.</td></tr>
<tr><td>SampleType</td><td>State time rate, returned as one of the following:

• 'CSTATE' — Continuous-time state
• 'DSTATE' — Discrete-time state</td></tr>
<tr><td>inReferencedModel</td><td>Flag indicating whether the block is inside a reference model, returned as one of the following:

• 1 — Block is inside a reference model.
• 0 — Block is in the current model file.</td></tr>
<tr><td>Known</td><td>Flags indicating whether state values are known during trimming, specified as a logical vector of length Nx.

• 1 — Known value that is fixed during operating point search
• 0 (default) — Unknown value found by optimization

To fix a state during an operating point search, set the corresponding Known flag to 1, and specify the value for that state using the x property of States.</td></tr>
<tr><td>SteadyState</td><td>Flags indicating whether output values are at steady state during trimming, specified as a logical vector of length Nx.

• 1 (default) — Equilibrium state</td></tr>
</table>
</td></tr>
</table>

<table>
<tr><th>Property</th><th>Description</th></tr>
<tr><td></td><td>
<table>
<tr><th>Field</th><th>Description</th></tr>
<tr><td></td><td>• <code>0</code> — Nonequilibrium state</td></tr>
<tr><td><code>Min</code></td><td>Minimum bounds on state values, specified as a vector of length <code>Nx</code>. By default, the minimum bound for each state is <code>-Inf</code>.</td></tr>
<tr><td><code>Max</code></td><td>Maximum bounds on state values, specified as a vector of length <code>Nx</code>. By default, the maximum bound for each state is <code>Inf</code>.</td></tr>
<tr><td><code>dxMin</code></td><td>Minimum bounds on state derivatives that are not at steady-state, specified as a vector of length <code>Nx</code>. By default, the minimum bound for each state derivative is <code>-Inf</code>. When you specify a derivative bound, you must also set <code>SteadyState</code> to <code>0</code>.</td></tr>
<tr><td><code>dxMax</code></td><td>Maximum bounds on state derivatives that are not at steady-state, specified as a vector of length <code>Nx</code>. By default, the maximum bound for each state derivative is <code>Inf</code>. When you specify a derivative bound, you must also set <code>SteadyState</code> to <code>0</code>.</td></tr>
<tr><td><code>Description</code></td><td>Block state description, specified as a character vector.</td></tr>
</table>
</td></tr>
</table>

<table>
<tr><th>Property</th><th>Description</th></tr>
<tr><td>Inputs</td><td>Input level specifications at the operating point, returned as a vector of input specification objects. Each entry in Inputs represents the input levels of one root-level inport block in the model.

Each input specification object has the following fields:

<table>
<tr><th>Field</th><th>Description</th></tr>
<tr><td>Nu (read only)</td><td>Number of inport block signals</td></tr>
<tr><td>Block</td><td>Inport block name</td></tr>
<tr><td>PortDimensions</td><td>Dimension of signals accepted by the inport</td></tr>
<tr><td>u</td><td>Inport block input levels at the operating point, returned as a vector of length PortWidth.

If the corresponding flag in Known field of Inputs is 1, u contains the known input values. Otherwise, u contains initial guesses for the input values.</td></tr>
<tr><td>Known</td><td>Flags indicating whether input levels are known during trimming, specified as a logical vector of length PortWidth.

• 1 — Known input level that is fixed during operating point search
• 0 (default) — Unknown input level found by optimization

To fix an input level during an operating point search, set the corresponding Known flag to 1, and specify the input value using the u property of Inputs.</td></tr>
<tr><td>Min</td><td>Minimum bounds on input levels, specified as a vector of length PortWidth. By default, the minimum bound for each input is -Inf.</td></tr>
<tr><td>Max</td><td>Maximum bounds on input levels, specified as a vector of length PortWidth. By default, the maximum bound for each input is Inf.</td></tr>
<tr><td>Description</td><td>Inport block input description, specified as a character vector.</td></tr>
</table>
</td></tr>
</table>

<table>
<tr><th>Property</th><th>Description</th></tr>
<tr><td>Outputs</td><td>Output level specifications at the operating point, returned as a vector of output specification objects. Each entry in Outputs represents the output levels of one root-level outport block of the model or one trim output constraint in the model.

You can specify additional trim output constraints using addoutputspec.

Each output specification object has the following fields:

<table>
<tr><th>Field</th><th>Description</th></tr>
<tr><td>Ny (read only)</td><td>Number of outport block signals</td></tr>
<tr><td>Block</td><td>Outport block name</td></tr>
<tr><td>PortNumber</td><td>Number of this outport in the model</td></tr>
<tr><td>y</td><td>Outport block output levels at the operating point, specified as a vector of length PortWidth.

If the corresponding flag in Known field of Outputs is 1, y contains the known output values. Otherwise, y contains initial guesses for the output values.</td></tr>
<tr><td>Known</td><td>Flags indicating whether output levels are known during trimming, specified as a logical vector of length PortWidth.

• 1 — Known output level that is fixed during operating point search
• 0 (default) — Unknown output level found by optimization

To fix an output level during an operating point search, set the corresponding Known flag to 1, and specify the output value using the y property of Outputs.</td></tr>
<tr><td>Min</td><td>Minimum bounds on output levels, specified as a vector of length PortWidth. By default, the minimum bound for each output is -Inf.</td></tr>
<tr><td>Max</td><td>Maximum bounds the output levels, specified as a vector of length PortWidth. By default, the maximum bound for each output is Inf.</td></tr>
<tr><td>Description</td><td>Outport block input description, specified as a character vector.</td></tr>
</table>
</td></tr>
<tr><td>Time</td><td>Times at which the time-varying functions in the model are evaluated, returned as a vector.</td></tr>
<tr><td>CustomObjFcn</td><td>Function providing an additional custom objective function for trimming, specified as a handle to the custom function, or a character vector or string that contains the function name. The custom function must be on the MATLAB path or in the current working folder.

You can specify a custom objective function as an algebraic combination of model states, inputs, and outputs. For more information, see “Compute Operating Points Using Custom Constraints and Objective Functions” on page 1-57.</td></tr>
</table>

Property	Description
`CustomConstrFcn`	Function providing additional custom constraints for trimming, specified as a handle to the custom function, or a character vector or string that contains the function name. The custom function must be on the MATLAB path or in the current working folder. You can specify custom equality and inequality constraints as algebraic combinations of model states, inputs, and outputs. For more information, see “Compute Operating Points Using Custom Constraints and Objective Functions” on page 1-57.
`CustomMappingFcn`	Function that maps model states, inputs, and outputs to the vectors accepted by `CustomConstrFcn` and `CustomObjFcn`, specified as a handle to the custom function, or a character vector or string that contains the function name. The custom function must be on the MATLAB path or in the current working folder. For complex models, you can pass subsets of the model inputs, outputs, and states to the custom constraint and objective functions using a custom mapping function. If you specify a custom mapping, you must use the mapping for both the custom constraint function and the custom objective function. For more information, see “Compute Operating Points Using Custom Constraints and Objective Functions” on page 1-57.

Tips

- To display the operating point specification object properties, use `get`.
- You can create new `operspec` variables of in 3 ways:
  - Construct a new object with the `operspec` command.
  - Create a new variable by assignment with the equals (=) operator.
  - Copy an `operspec` object using the `copy` command.

  Using `operspec` or `copy` creates a new, independent object. When you use assignment, there is a link between the old and new variable. For an example, see “Copy an Operating-Point Specification” on page 18-185.

Compatibility Considerations

`PortWidth` property of operating point specification inputs and outputs will be removed
Not recommended starting in R2021b

The input and output `PortWidth` properties of operating point specifications will be removed in a future release. Use the new `Nu` and `Ny` properties instead.

To update your code, change instances of `PortWidth` to either `Nu` or `Ny` as shown in the following table.

Not Recommended	Recommended

`op = operspec('scdplane');`
`numOut = op.Outputs(1).PortWidth;`
`numIn = op.Inputs(1).PortWidth;`
	`op = operspec('scdplane');`
	`numOut = op.Outputs(1).Ny;`
	`numIn = op.Inputs(1).Nu;`

See Also

`findop` | `addoutputspec` | `update` | `copy`

Introduced before R2006a

set

Set properties of linearization I/Os and operating points

Syntax

```
set(ob)
set(ob,'PropertyName',val)
```

Description

`set(ob)` displays all editable properties of the object, `ob`, which can be a linearization I/O object, an operating point object, or an operating point specification object. Create `ob` using `findop`, `getlinio`, `linio`, `operpoint`, or `operspec`.

`set(ob,'PropertyName',val)` sets the property, `PropertyName`, of the object, `ob`, to the value, `val`. The object, `ob`, can be a linearization I/O object, an operating point object, or an operating point specification object. Create `ob` using `findop`, `getlinio`, `linio`, `operpoint`, or `operspec`.

`ob.PropertyName = val` is an alternative notation for assigning the value, `val`, to the property, `PropertyName`, of the object, `ob`. The object, `ob`, can be a linearization I/O object, an operating point object, or an operating point specification object. Create `ob` using `findop`, `getlinio`, `linio`, `operpoint`, or `operspec`.

Examples

Create an operating point object for the Simulink model, `magball`:

```
op_cond=operpoint('magball');
```

Use the `set` function to get a list of all editable properties of this object:

```
set(op_cond)
```

This function returns the properties of `op_cond`.

```
ans =
     Model: {}
    States: {}
    Inputs: {}
      Time: {}
```

To set the value of a particular property of `op_cond`, provide the property name and the desired value of this property as arguments to `set`. For example, to change the name of the model associated with the operating point object from `'magball'` to `'Magnetic Ball'`, type:

```
set(op_cond,'Model','Magnetic Ball')
```

To view the property value and verify that the change was made, type:

```
op_cond.Model
```

which returns

```
ans =
Magnetic Ball
```

Because `op_cond` is a structure, you can set any properties or fields using dot-notation. First, produce a list of properties of the second `States` object within `op_cond`, as follows:

```
set(op_cond.States(2))
```

which returns

```
ans =

                  Nx: {}
               Block: {}
           StateName: {}
                   x: {}
                  Ts: {}
          SampleType: {}
    inReferencedModel: {}
         Description: {}
```

Now, use dot-notation to set the `x` property to `8`:

```
op_cond.States(2).x=8;
```

To view the property and verify that the change was made, type

```
op_cond.States(2)
```

which displays

```
(1.) magball/Magnetic Ball Plant/Current
      x: 8
```

See Also

`findop` | `get` | `linio` | `operpoint` | `operspec` | `setlinio`

Introduced before R2006a

setlinio

Save linear analysis points to Simulink model, Linear Analysis Plots block, or Model Verification block

Syntax

```
setlinio(mdl,io)
setlinio(blockpath,io)
oldio = setlinio( ___ )
```

Description

`setlinio(mdl,io)` writes the analysis points specified in `io` to the Simulink model `mdl`.

`setlinio(blockpath,io)` sets the specified analysis points to the specified Linear Analysis Plots block or Model Verification block.

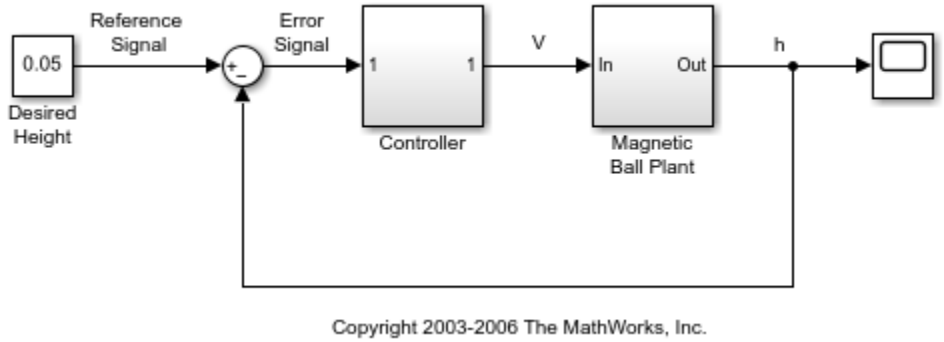
`oldio = setlinio( ___ )` returns the current set of analysis points in the model or block and replaces them with `io` using any of the previous syntaxes.

Examples

Set Analysis Points in Simulink Model

Open Simulink model.

```
model = 'magball';
open_system(model)
```



Create a vector of analysis points for linearizing the plant model:

- Input perturbation at the output of the Controller block
- Open-loop output at the output of the Magnetic Ball Plant block

```
io(1) = linio('magball/Controller',1,'input');
io(2) = linio('magball/Magnetic Ball Plant',1,'openoutput');
```

Write the analysis points to the `magball` model.

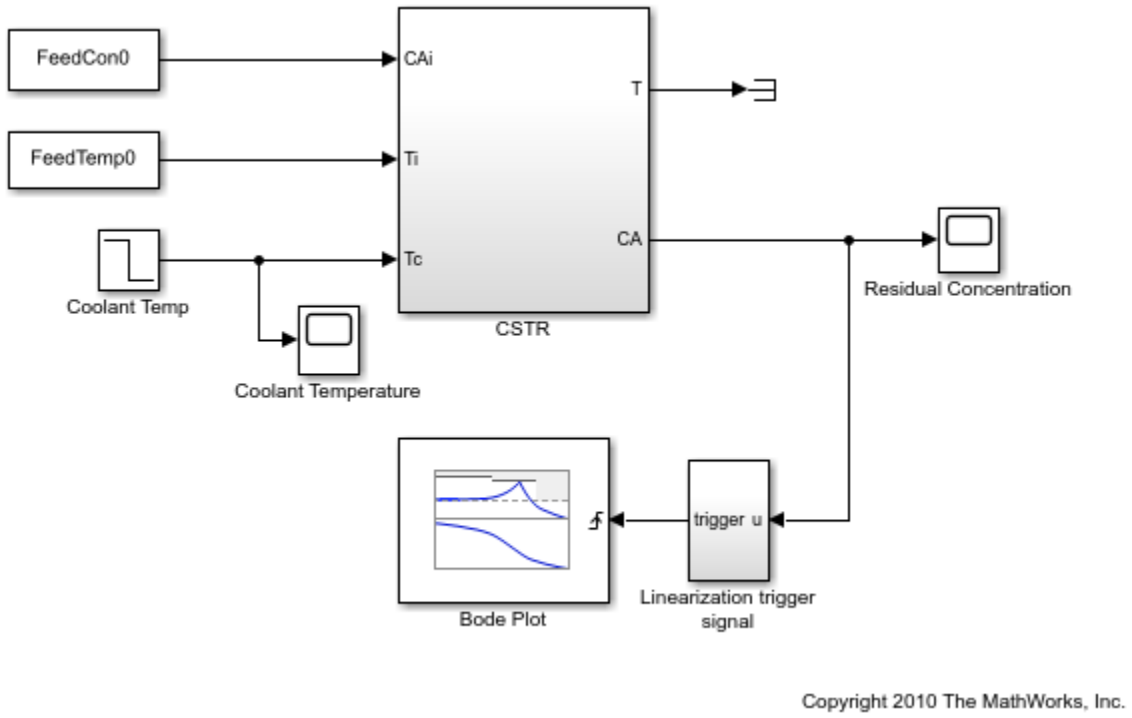
```
setlinio(model,io);
```

The analysis points in `io` are added to the model as annotations. You can then save the model to store the analysis points with the model.

Set Analysis Points in Linear Analysis Plots Block

Open Simulink model.

```
open_system('scdcstr')
```



Create analysis points for finding the transfer function between the coolant temperature and the residual concentration.

- Input perturbation at the output of the Coolant Temp block
- Output measurement at the CA output of the CSTR block

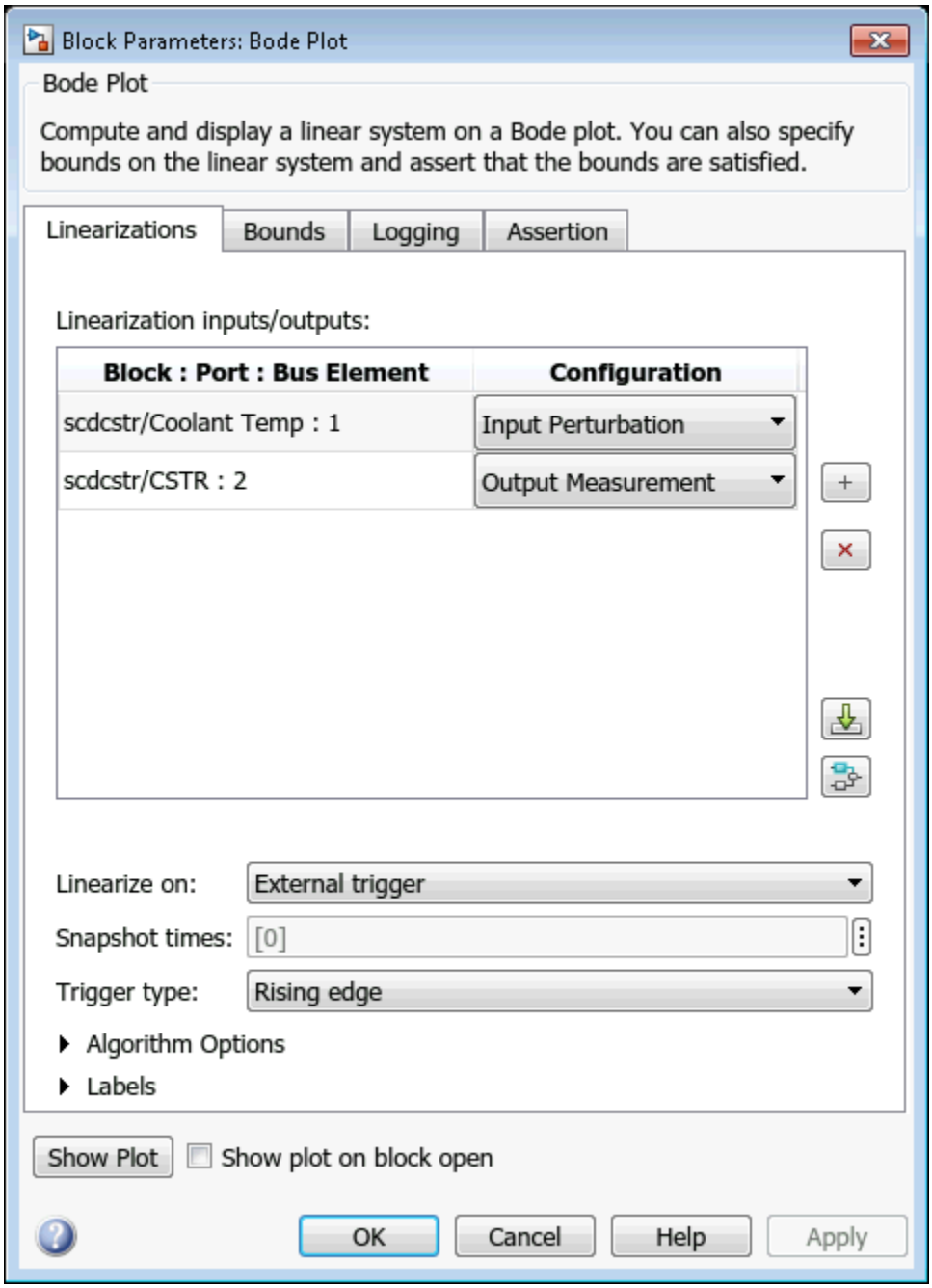
```
io(1) = linio('scdcstr/Coolant Temp',1,'input');
io(2) = linio('scdcstr/CSTR',2,'output');
```

Set the analysis points in the Bode Plot block.

```
setlinio('scdcstr/Bode Plot',io);
```

View the analysis points in the Bode Plot Block Parameters dialog box.

```
open_system('scdcstr/Bode Plot')
```

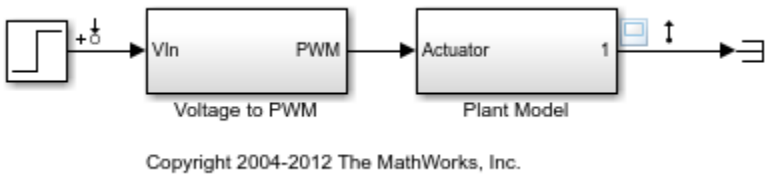


During simulation, the software linearizes the model using the specified analysis, and plots the magnitude and phase responses for the resulting linear system.

Save Old Analysis Points When Storing New Analysis Points

Open Simulink model.

```
mdl = 'scdpwm';
open_system(mdl)
```



This model is configured with analysis points for finding the combined transfer function of the PWM and plant blocks.

Create analysis points for finding the transfer function of just the plant model.

```
io(1) = linio('scdpwm/Voltage to PWM',1,'input');
io(2) = linio('scdpwm/Plant Model',1,'output');
```

Store the analysis points to the model, and save the previous analysis point configuration.

```
oldio = setlinio(mdl,io)
```

```
2x1 vector of Linearization IOs:
--------------------------
1. Linearization input perturbation located at the following signal:
- Block: scdpwm/Step
- Port: 1
2. Linearization output measurement located at the following signal:
- Block: scdpwm/Plant Model
- Port: 1
```

Input Arguments

`mdl` — Simulink model name

character vector | string

Simulink model name, specified as a character vector or string. The model must be in the current working folder or on the MATLAB path.

If the model is not open or loaded into memory, `setlinio` loads the model into memory.

`io` — Analysis point set

linearization I/O object | vector of linearization I/O objects

Analysis point set, specified as a linearization I/O object or a vector of linearization I/O objects.

Each linearization I/O object has the following properties:

Property	Description
`Active`	Flag indicating whether to use the analysis point for linearization, specified as one of the following: • `'on'` — Use the analysis point for linearization. This value is the default option. • `'off'` — Do not use the analysis point for linearization. Use this option if you have an existing set of analysis points and you want to linearize a model with a subset of these points.
`Block`	Full block path of the block with which the analysis point is associated, specified as a character vector.
`PortNumber`	Output port with which the analysis point is associated, specified as an integer.
`Type`	Analysis point type, specified as one of the following: • `'input'` — Input perturbation • `'output'` — Output measurement • `'loopbreak'` — Loop break • `'openinput'` — Open-loop input • `'openoutput'` — Open-loop output • `'looptransfer'` — Loop transfer • `'sensitivity'` — Sensitivity • `'compsensitivity'` — Complementary sensitivity For more information on analysis point types, see “Specify Portion of Model to Linearize” on page 2-10.
`BusElement`	Bus element name with which the analysis point is associated, specified as a character vector or `''` if the analysis point is not a bus element.
`Description`	User-specified description of the analysis point, which you can set for convenience, specified as a character vector.

`blockpath` — Linear Analysis Plots block or Model Verification block
character vector | string

Linear Analysis Plots block or Model Verification block, specified as a character vector or string that contains its full block path. The model that contains the block must be in the current working folder or on the MATLAB path.

For more information on:

- Linear analysis plot blocks, see “Visualization During Simulation”.
- Model verification blocks, see “Model Verification”.

Output Arguments

`oldio` — Old analysis point set
linearization I/O object | vector of linearization I/O objects

Old analysis point set, returned as a linearization I/O object or a vector of linearization I/O objects.

Alternative Functionality

Simulink Model

You can also specify analysis points directly in a Simulink model. For more information, see “Specify Portion of Model to Linearize in Simulink Model” on page 2-17.

See Also

`linio` | `getlinio` | `linearize` | `slLinearizer`

Topics

“Specify Portion of Model to Linearize” on page 2-10

Introduced before R2006a

setxu

Set states and inputs in operating points

Syntax

```
op_new = setxu(op_point,x,u)
```

Description

`setxu` set state and input values for an operating point. Use this function for applications that require state and input values to be specified in vector format, such as when using an operating in an optimization problem using `fmincon`.

`op_new = setxu(op_point,x,u)` returns a new operating point with the specified state and input values (`x` and `u`, respectively). Specify `x` as either a vector or structure with the same format as those returned from a Simulink simulation. Specifying `x` as a structure with time is not supported. Specify `u` as a vector. You can obtain both `x` and `u` 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`.

```
mdl = 'scdplane';
open_system(mdl)
```

You can save the final states of the model to the workspace after a simulation. In the Simulink editor, on the **Modeling** tab, click **Model Settings**. Then, in the Configuration Parameters dialog box, select the **Final states** parameter.

Simulate the model. After the simulation, the `xFinal` variable appears in the workspace. This variable is a vector containing the final state values.

```
sim(mdl)
```

Create an operating point object for `scdplane`.

```
op_point = operpoint(mdl)
```

```
 Operating Point for the Model scdplane.
 (Time-Varying Components Evaluated at time t=0)

States:
----------
(1.) scdplane/Actuator Model
      x: 0
```

```
(2.) scdplane/Aircraft Dynamics Model/Transfer Fcn.1
      x: 0
(3.) scdplane/Aircraft Dynamics Model/Transfer Fcn.2
      x: 0
(4.) scdplane/Controller/Alpha-sensor Low-pass Filter
      x: 0
(5.) scdplane/Controller/Pitch Rate Lead Filter
      x: 0
(6.) scdplane/Controller/Proportional plus integral compensator
      x: 0
(7.) scdplane/Controller/Stick Prefilter
      x: 0
(8.) scdplane/Dryden Wind Gust Models/Q-gust model
      x: 0
(9.) scdplane/Dryden Wind Gust Models/W-gust model
      x: 0
      x: 0

Inputs:
----------
(1.) scdplane/u
      u: 0
```

All states are initially set to `0`.

Initialize the states in the operating point object to the values in `xFinal`. Set the input to be `9`.

```
newop = setxu(op_point,xFinal,9)
```

```
 Operating Point for the Model scdplane.
 (Time-Varying Components Evaluated at time t=0)

States:
----------
(1.) scdplane/Actuator Model
      x: -0.032
(2.) scdplane/Aircraft Dynamics Model/Transfer Fcn.1
      x: 0.56
(3.) scdplane/Aircraft Dynamics Model/Transfer Fcn.2
      x: 678
(4.) scdplane/Controller/Alpha-sensor Low-pass Filter
      x: 0.392
(5.) scdplane/Controller/Pitch Rate Lead Filter
      x: 0.133
(6.) scdplane/Controller/Proportional plus integral compensator
      x: 0.166
(7.) scdplane/Controller/Stick Prefilter
      x: 0.1
(8.) scdplane/Dryden Wind Gust Models/Q-gust model
      x: 0.114
(9.) scdplane/Dryden Wind Gust Models/W-gust model
      x: 0.46
      x: -2.05

Inputs:
----------
```

```
(1.) scdplane/u
      u: 9
```

Alternatives

As an alternative to the `setxu` function, set states and inputs of operating points using the **Model Linearizer** app.

See Also

`getxu` | `initopspec` | `operpoint` | `operspec`

Introduced before R2006a

slLinearizer

Interface for batch linearization of Simulink models

Syntax

```
sllin = slLinearizer(mdl)
sllin = slLinearizer(mdl,pt)
sllin = slLinearizer(mdl,param)
sllin = slLinearizer(mdl,op)
sllin = slLinearizer(mdl,blocksub)
sllin = slLinearizer(mdl,options)
sllin = slLinearizer(mdl,pt,op,param,blocksub,options)
```

Description

`sllin = slLinearizer(mdl)` creates an `slLinearizer` interface, `sllin`, for linearizing the Simulink model, `mdl`. The interface adds the linear analysis points marked in the model as analysis points on page 18-216 of `sllin`. The interface also adds the linear analysis points that imply an opening as permanent openings on page 18-216.

`sllin = slLinearizer(mdl,pt)` adds the specified point to the list of analysis points for `sllin`, ignoring linear analysis points marked in the model.

`sllin = slLinearizer(mdl,param)` specifies the parameters whose values you want to vary when linearizing the model.

`sllin = slLinearizer(mdl,op)` specifies the operating points for linearizing the model.

`sllin = slLinearizer(mdl,blocksub)` specifies substitute linearizations of blocks and subsystems. Use this syntax, for example, to specify a custom linearization for a block. You can also use this syntax for blocks that do not linearize successfully, such as blocks with discontinuities or triggered subsystems.

`sllin = slLinearizer(mdl,options)` configures the linearization algorithm options.

`sllin = slLinearizer(mdl,pt,op,param,blocksub,options)` uses any combination of the input arguments `pt`, `op`, `param`, `blocksub`, and `options` to create `sllin`.

For example, use any of the following:

- `sllin = slLinearizer(mdl,pt,param)`
- `sllin = slLinearizer(mdl,op,param)`.

If you do not specify `pt`, the interface adds the linear analysis points marked in the model as analysis points. The interface also adds linear analysis points that imply an opening as permanent openings.

Object Description

`slLinearizer` provides an interface between a Simulink model and the linearization commands `getIOTransfer`, `getLoopTransfer`, `getSensitivity`, and `getCompSensitivity`. Use

`slLinearizer` to efficiently batch linearize a model. You can configure the `slLinearizer` interface to linearize a model at a range of operating points and specify variations for model parameter values. Use interface analysis points on page 18-216 and permanent openings on page 18-216 to obtain linearizations for any open-loop or closed-loop transfer function from a model. Analyze the stability, or time-domain or frequency-domain characteristics of the linearized models.

If you changed any interface properties since the last linearization, commands that extract linearizations from the `slLinearizer` interface recompile the Simulink model. If you made calls to specific functions since the last linearization, the commands also recompile the Simulink model. These functions include `addPoint`, `addOpening`, `removePoint`, `removeOpening`, `removeAllPoints`, and `removeAllOpenings`.

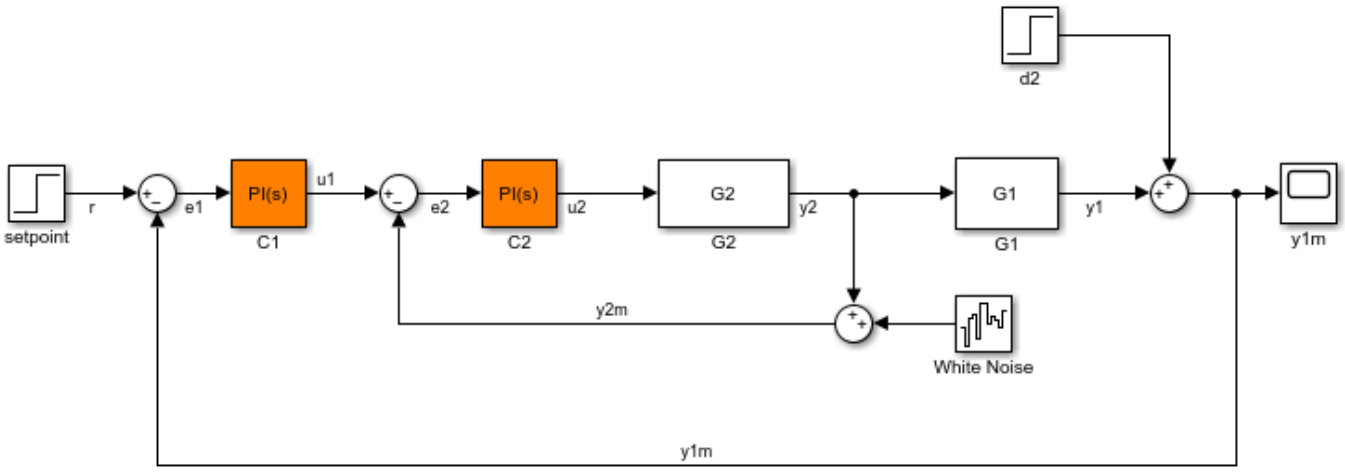
Examples

Create and Configure slLinearizer Interface for Batch Linear Analysis

Create an `slLinearizer` interface for the `scdcascade` model. Add analysis points to the interface to extract open- or closed-loop transfer functions from the model. Configure the interface to vary parameters and operating points.

Open the `scdcascade` model.

```
mdl = 'scdcascade';
open_system(mdl);
```



Create an `slLinearizer` interface for the model. Add the signals `r`, `u1`,|u2|, `y1`,|y2|, `y1m`, and `y2m` to the interface.

```
sllin = slLinearizer(mdl,{'r','u1','u2','y1','y2','y1m','y2m'});
```

`scdcascade` contains two PID Controller blocks, `C1` and `C2`. Suppose you want to vary the proportional and integral gains of `C2`, `Kp2` and `Ki2`, in the 10% range. Create a structure to specify the parameter variations.

```
Kp2_range = linspace(0.9*Kp2,1.1*Kp2,3);
Ki2_range = linspace(0.9*Ki2,1.1*Ki2,5);
```

```
[Kp2_grid,Ki2_grid]=ndgrid(Kp2_range,Ki2_range);

params(1).Name = 'Kp2';
params(1).Value = Kp2_grid;

params(2).Name = 'Ki2';
params(2).Value = Ki2_grid;
```

`params` specifies a 3x5 parameter grid. Each point in this grid corresponds to a combination of the `Kp2` and `Ki2` parameter values.

Specify `params` as the `Parameters` property of `sllin`.

```
sllin.Parameters = params;
```

Now, when you use commands such as `getIOTransfer`, `getLoopTransfer`, `getSensitivity`, and `getCompSensitivity`, the software returns a linearization for each parameter grid point specified by `sllin.Parameters`.

Suppose you want to linearize the model at multiple snapshot times, for example at `t = {0,1,2}`. To do so, configure the `OperatingPoints` property of `sllin`.

```
sllin.OperatingPoints = [0 1 2];
```

You can optionally configure the linearization options and specify substitute linearizations for blocks and subsystems in your model. After fully configuring `sllin`, use the `getIOTransfer`, `getLoopTransfer`, `getSensitivity`, and `getCompSensitivity` commands to linearize the model as required.

Input Arguments

`mdl` — Simulink model name

character vector | string

Simulink model name, specified as a character vector or string.

Example: `'scdcascade'`

`pt` — Analysis point

character vector | string | cell array of character vectors | string array | vector of linearization I/O objects

Analysis point on page 18-216 to be added to the list of analysis points for `sllin`, specified as:

- Character vector or string — Analysis point identifier that can be any of the following:
  - Signal name, for example `pt = 'torque'`
  - Block path for a block with a single output port, for example `pt = 'Motor/PID'`
  - Block path and port originating the signal, for example `pt = 'Engine Model/1'`
- Cell array of character vectors or string array — Specifies multiple analysis point identifiers. For example:

  ```
  pt = {'torque','Motor/PID','Engine Model/1'}
  ```

- Vector of linearization I/O objects — Create `pt` using `linio`. For example:

  ```
  pt(1) = linio('scdcascade/setpoint',1,'input');
  pt(2) = linio('scdcascade/Sum',1,'output');
  ```

  Here, `pt(1)` specifies an input, and `pt(2)` specifies an output.

  The interface adds all the points specified by `pt` and ignores their I/O types. The interface also adds all `'loopbreak'` type signals as permanent openings on page 18-216.

`param` — Parameter samples
structure | structure array

Parameter samples for linearizing `mdl`, specified as:

- Structure — Vary the value of a single parameter by specifying `param` as a structure with the following fields:

  - `Name` — Parameter name, specified as a character vector or string. You can specify any model parameter that is a variable in the model workspace, the MATLAB workspace, or a data dictionary. If the variable used by the model is not a scalar variable, specify the parameter name as an expression that resolves to a numeric scalar value. For example, to use the first element of vector `V` as a parameter, use:

    ```
    param.Name = 'V(1)';
    ```

  - `Value` — Parameter sample values, specified as a double array.

  For example, vary the value of parameter `A` in the 10% range:

  ```
  param.Name = 'A';
  param.Value = linspace(0.9*A,1.1*A,3);
  ```

- Structure array — Vary the value of multiple parameters. For example, vary the values of parameters `A` and `b` in the 10% range:

  ```
  [A_grid,b_grid] = ndgrid(linspace(0.9*A,1.1*A,3),...
                           linspace(0.9*b,1.1*b,3));
  params(1).Name = 'A';
  params(1).Value = A_grid;
  params(2).Name = 'b';
  params(2).Value = b_grid;
  ```

For more information, see “Specify Parameter Samples for Batch Linearization” on page 3-43.

If `param` specifies tunable parameters only, then the software batch linearizes the model using a single compilation. If you also configure `sllin.OperatingPoints` with operating point objects only, the software uses single model compilation.

For an example showing how batch linearization with parameter sampling works, see “Vary Parameter Values and Obtain Multiple Transfer Functions” on page 3-21.

To compute the offsets required by the LPV System block, specify `param`, and set `sllin.Options.StoreOffsets` to `true`. You can then return additional linearization information when calling linearization functions such as `getIOTransfer`, and extract the offsets using `getOffsetsForLPV`.

`op` — Operating point for linearizing `mdl`

`OperatingPoint` object | array of `OperatingPoint` objects | vector of positive scalars | `OperatingReport` object | array of `OperatingReport` objects

Operating point for linearizing `mdl`, specified as:

- `OperatingPoint` or `OperatingReport` object, created using `findop` with either a single operating point specification, or a single snapshot time.
- Array of `OperatingPoint` or `OperatingReport` objects, specifying multiple operating points.

  To create an array of `OperatingPoint` or `OperatingReport` objects, you can:

  - Extract operating points at multiple snapshot times using `findop`.
  - Batch trim your model using multiple operating point specifications. For more information, see “Batch Compute Steady-State Operating Points for Multiple Specifications” on page 1-68.
  - Batch trim your model using parameter variations. For more information, see “Batch Compute Steady-State Operating Points for Parameter Variation” on page 1-72.
- Vector of positive scalars, specifying simulation snapshot times.

If you configure `sllin.Parameters`, then specify `op` as one of the following:

- Single `OperatingPoint` or `OperatingReport` object.
- Array of `OperatingPoint` or `OperatingReport` objects whose size matches that of the parameter grid specified by the `Parameters` property. When you batch linearize `mdl`, the software uses only one model compilation. To obtain the operating points that correspond to the parameter value combinations, batch trim your model using `param` before linearization. For an example that uses the `linearize` command, see “Batch Linearize Model at Multiple Operating Points Derived from Parameter Variations” on page 3-16.
- Multiple snapshot times. When you batch linearize `mdl`, the software simulates the model for each snapshot time and parameter grid point combination. This operation can be computationally expensive.

`blocksub` — Substitute linearizations for blocks and model subsystems

structure | structure array

Substitute linearizations for blocks and model subsystems, specified as a structure or an *n*-by-1 structure array, where *n* is the number of blocks for which you want to specify a linearization. Use `blocksub` to specify a custom linearization for a block or subsystem. For example, you can specify linearizations for blocks that do not have analytic linearizations, such as blocks with discontinuities or triggered subsystems.

To study the effects of varying the linearization of a block on the model dynamics, you can batch linearize your model by specifying multiple substitute linearizations for a block.

If you substitute a linearization with a sample time that differs from that of the original block or subsystem, it is best practice to set the overall linearization sample time (`options.SampleTime`) to a nondefault value.

Each substitute linearization structure has the following fields.

`Name` — Block path

character vector | string

Block path of the block for which you want to specify the linearization, specified as a character vector or string.

`Value` — Substitute linearization
double | double array | LTI model | model array | structure

Substitute linearization for the block, specified as one of the following:

- Double — Specify the linearization of a SISO block as a gain.
- Array of doubles — Specify the linearization of a MIMO block as an n_u-by-n_y array of gain values, where n_u is the number of inputs and n_y is the number of outputs.
- LTI model, uncertain state-space model, or uncertain real object — The I/O configuration of the specified model must match the configuration of the block specified by `Name`. Using an uncertain model requires Robust Control Toolbox software.
- Array of LTI models, uncertain state-space models, or uncertain real objects — Batch linearize the model using multiple block substitutions. The I/O configuration of each model in the array must match the configuration of the block for which you are specifying a custom linearization. If you:
  - Vary model parameters using `param` and specify `Value` as a model array, the dimensions of `Value` must match the parameter grid size.
  - Define block substitutions for multiple blocks, and specify `Value` as an array of LTI models for more than one block, the dimensions of the arrays must match.
- Structure with the following fields:

Field	Description
`Specification`	Block linearization, specified as a character vector that contains one of the following • MATLAB expression • Name of a "Custom Linearization Function" on page 18-217 in your current working directory or on the MATLAB path. The specified expression or function must return one of the following: • Linear model in the form of a D-matrix • Control System Toolbox LTI model object • Uncertain state-space model or uncertain real object (requires Robust Control Toolbox software) The I/O configuration of the returned model must match the configuration of the block specified by `Name`.
`Type`	Specification type, specified as one of the following: • `'Expression'` • `'Function'`

Field	Description
`ParameterNames`	Linearization function parameter names, specified as a cell array of character vectors. Specify `ParameterNames` only when `Type = 'Function'` and your block linearization function requires input parameters. These parameters only impact the linearization of the specified block. You must also specify the corresponding `blocksub.Value.ParameterValues` field.
`ParameterValues`	Linearization function parameter values, specified as an vector of doubles. The order of parameter values must correspond to the order of parameter names in `blocksub.Value.ParameterNames`. Specify `ParameterValues` only when `Type = 'Function'` and your block linearization function requires input parameters.

`options` — Linearization algorithm options
`linearizeOptions` option set

Linearization algorithm options, specified as a `linearizeOptions` option set.

Properties

`Parameters`

Parameter samples for linearizing `mdl`, specified as a structure or a structure array.

Set this property using the `param` input argument or dot notation (`sllin.Parameters = param`). `param` must be one of the following:

- Structure — Vary the value of a single parameter by specifying `param` as a structure with the following fields:

  - `Name` — Parameter name, specified as a character vector or string. You can specify any model parameter that is a variable in the model workspace, the MATLAB workspace, or a data dictionary. If the variable used by the model is not a scalar variable, specify the parameter name as an expression that resolves to a numeric scalar value. For example, to use the first element of vector `V` as a parameter, use:

    ```
    param.Name = 'V(1)';
    ```

  - `Value` — Parameter sample values, specified as a double array.

  For example, vary the value of parameter `A` in the 10% range:

  ```
  param.Name = 'A';
  param.Value = linspace(0.9*A,1.1*A,3);
  ```

- Structure array — Vary the value of multiple parameters. For example, vary the values of parameters `A` and `b` in the 10% range:

  ```
  [A_grid,b_grid] = ndgrid(linspace(0.9*A,1.1*A,3),...
                           linspace(0.9*b,1.1*b,3));
  params(1).Name = 'A';
  params(1).Value = A_grid;
  ```

```
params(2).Name = 'b';
params(2).Value = b_grid;
```

If `param` specifies tunable parameters only, then the software batch linearizes the model using a single compilation. If you also configure `sllin.OperatingPoints` with operating point objects only, the software uses single model compilation.

`OperatingPoints`

Operating points for linearizing `mdl`, specified as an `OperatingPoint` object, array of `OperatingPoint` 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:

- `OperatingPoint` or `OperatingReport` object, created using `findop` with either a single operating point specification, or a single snapshot time.
- Array of `OperatingPoint` or `OperatingReport` objects, specifying multiple operating points.

  To create an array of `OperatingPoint` or `OperatingReport` objects, you can:

  - Extract operating points at multiple snapshot times using `findop`.
  - Batch trim your model using multiple operating point specifications. For more information, see “Batch Compute Steady-State Operating Points for Multiple Specifications” on page 1-68.
  - Batch trim your model using parameter variations. For more information, see “Batch Compute Steady-State Operating Points for Parameter Variation” on page 1-72.
- Vector of positive scalars, specifying simulation snapshot times.

If you configure `sllin.Parameters`, then specify `op` as one of the following:

- Single `OperatingPoint` or `OperatingReport` object.
- Array of `OperatingPoint` or `OperatingReport` objects whose size matches that of the parameter grid specified by the `Parameters` property. When you batch linearize `mdl`, the software uses only one model compilation. To obtain the operating points that correspond to the parameter value combinations, batch trim your model using `param` before linearization. For an example that uses the `linearize` command, see “Batch Linearize Model at Multiple Operating Points Derived from Parameter Variations” on page 3-16.
- Multiple snapshot times. When you batch linearize `mdl`, the software simulates the model for each snapshot time and parameter grid point combination. This operation can be computationally expensive.

`BlockSubstitutions`

Substitute linearizations for blocks and model subsystems, specified as a structure or structure array.

Use this property to specify a custom linearization for a block or subsystem. You also can use this syntax for blocks that do not have analytic linearizations, such as blocks with discontinuities or triggered subsystems.

Set this property using the `blocksub` input argument or dot notation (`sllin.BlockSubstitutions = blocksubs`). For information about the required structure, see `blocksub`.

`Options`

Linearization algorithm options, specified as an option set created using `linearizeOptions`.

Set this property using the `options` input argument or dot notation (`sllin.Options = options`).

`Model`

Name of the Simulink model to be linearized, specified as a character vector by the input argument `mdl`.

`TimeUnit`

Unit of the time variable. This property specifies the time units for linearized models returned by `getIOTransfer`, `getLoopTransfer`, `getSensitivity`, and `getCompSensitivity`. Use any of the following values:

- `'nanoseconds'`
- `'microseconds'`
- `'milliseconds'`
- `'seconds'`
- `'minutes'`
- `'hours'`
- `'days'`
- `'weeks'`
- `'months'`
- `'years'`

Default: `'seconds'`

Object Functions

addPoint	Add signal to list of analysis points for slLinearizer or slTuner interface
addOpening	Add signal to list of openings for slLinearizer or slTuner interface
addPoint	Add signal to list of analysis points for slLinearizer or slTuner interface
getPoints	Get list of analysis points for slLinearizer or slTuner interface
getOpenings	Get list of openings for slLinearizer or slTuner interface
getIOTransfer	Transfer function for specified I/O set using slLinearizer or slTuner interface
getLoopTransfer	Open-loop transfer function at specified point using slLinearizer or slTuner interface
getSensitivity	Sensitivity function at specified point using slLinearizer or slTuner interface
getCompSensitivity	Complementary sensitivity function at specified point using slLinearizer or slTuner interface
removePoint	Remove point from list of analysis points in slLinearizer or slTuner interface
removeAllPoints	Remove all points from list of analysis points in slLinearizer or slTuner interface
removeAllOpenings	Remove all openings from list of permanent openings in slLinearizer or slTuner interface
refresh	Resynchronize slLinearizer or slTuner interface with current model state

More About

Analysis Points

Analysis points, used by the `slLinearizer` and `slTuner` interfaces, identify locations within a model that are relevant for linear analysis and control system tuning. You use analysis points as inputs to the linearization commands, such as `getIOTransfer`, `getLoopTransfer`, `getSensitivity`, and `getCompSensitivity`. As inputs to the linearization commands, analysis points can specify any open-loop or closed-loop transfer function in a model. You can also use analysis points to specify design requirements when tuning control systems using commands such as `systune`.

Location refers to a specific block output port within a model or to a bus element in such an output port. For convenience, you can use the name of the signal that originates from this port to refer to an analysis point.

You can add analysis points to an `slLinearizer` or `slTuner` interface, `s`, when you create the interface. For example:

```
s = slLinearizer('scdcascade',{'u1','y1'});
```

Alternatively, you can use the `addPoint` command.

To view all the analysis points of `s`, type `s` at the command prompt to display the interface contents. For each analysis point of `s`, the display includes the block name and port number and the name of the signal that originates at this point. You can also programmatically obtain a list of all the analysis points using `getPoints`.

For more information about how you can use analysis points, see “Mark Signals of Interest for Control System Analysis and Design” on page 2-38 and “Mark Signals of Interest for Batch Linearization” on page 3-9.

Permanent Openings

Permanent openings, used by the `slLinearizer` and `slTuner` interfaces, identify locations within a model where the software breaks the signal flow. The software enforces these openings for linearization and tuning. Use permanent openings to isolate a specific model component. Suppose that you have a large-scale model capturing aircraft dynamics and you want to perform linear analysis on the airframe only. You can use permanent openings to exclude all other components of the model. Another example is when you have cascaded loops within your model and you want to analyze a specific loop.

Location refers to a specific block output port within a model. For convenience, you can use the name of the signal that originates from this port to refer to an opening.

You can add permanent openings to an `slLinearizer` or `slTuner` interface, `s`, when you create the interface or by using the `addOpening` command. To remove a location from the list of permanent openings, use the `removeOpening` command.

To view all the openings of `s`, type `s` at the command prompt to display the interface contents. For each permanent opening of `s`, the display includes the block name and port number and the name of the signal that originates at this location. You can also programmatically obtain a list of all the permanent loop openings using `getOpenings`.

Custom Linearization Function

You can specify a substitute linearization for a block or subsystem in your Simulink model using a custom function on the MATLAB path.

Your custom linearization function must have one `BlockData` input argument, which is a structure that the software creates and passes to the function. `BlockData` has the following fields:

Field	Description
`BlockName`	Name of the block for which you are specifying a custom linearization.
`Parameters`	Block parameter values, specified as a structure array with `Name` and `Value` fields. `Parameters` contains the names and values of the parameters you specify in the `blocksub.Value.ParameterNames` and `blocksub.Value.ParameterValues` fields.
`Inputs`	Input signals to the block for which you are defining a linearization, specified as a structure array with one structure for each block input. Each structure in `Inputs` has the following fields: **Field** / **Description** `BlockName` — Full block path of the block whose output connects to the corresponding block input. `PortIndex` — Output port of the block specified by `BlockName` that connects to the corresponding block input. `Values` — Value of the signal specified by `BlockName` and `PortIndex`. If this signal is a vector signal, then `Values` is a vector with the same dimension.
`ny`	Number of output channels of the block linearization.
`nu`	Number of input channels of the block linearization.
`BlockLinearization`	Current default linearization of the block, specified as a state-space model. You can specify a block linearization that depends on the default linearization using `BlockLinearization`.

Your custom function must return a model with `nu` inputs and `ny` outputs. This model must be one of the following:

- Linear model in the form of a D-matrix
- Control System Toolbox LTI model object
- Uncertain state-space model or uncertain real object (requires Robust Control Toolbox software)

For example, the following function multiplies the current default block linearization, by a delay of `Td = 0.5` seconds. The delay is represented by a Thiran filter with sample time `Ts = 0.1`. The delay and sample time are parameters stored in `BlockData`.

```
function sys = myCustomFunction(BlockData)
    Td = BlockData.Parameters(1).Value;
    Ts = BlockData.Parameters(2).Value;
    sys = BlockData.BlockLinearization*Thiran(Td,Ts);
end
```

Save this function to a location on the MATLAB path.

To use this function as a custom linearization for a block or subsystem, specify the `blocksub.Value.Specification` and `blocksub.Value.Type` fields.

```
blocksub.Value.Specification = 'myCustomFunction';
blocksub.Value.Type = 'Function';
```

To set the delay and sample time parameter values, specify the `blocksub.Value.ParameterNames` and `blocksub.Value.ParameterValues` fields.

```
blocksub.Value.ParameterNames = {'Td','Ts'};
blocksub.Value.ParameterValues = [0.5 0.1];
```

See Also

`addPoint` | `addOpening` | `getIOTransfer` | `getLoopTransfer` | `getSensitivity` | `getCompSensitivity` | `linearize`

Topics

“What Is Batch Linearization?” on page 3-2
“How the Software Treats Loop Openings” on page 2-31
“Batch Linearization Efficiency When You Vary Parameter Values” on page 3-7
“Batch Compute Steady-State Operating Points for Multiple Specifications” on page 1-68
“Specify Parameter Samples for Batch Linearization” on page 3-43
“Vary Operating Points and Obtain Multiple Transfer Functions Using slLinearizer Interface” on page 3-28
“Vary Parameter Values and Obtain Multiple Transfer Functions” on page 3-21

Introduced in R2013b

addOpening

Add signal to list of openings for `slLinearizer` or `slTuner` interface

Syntax

```
addOpening(s,pt)

addOpening(s,blk,port_num)
addOpening(s,blk,port_num,bus_elem_name)
```

Description

`addOpening(s,pt)` adds the specified point (signal) to the list of permanent openings on page 18-224 for the `slLinearizer` or `slTuner` interface, `s`.

Use permanent openings to isolate a specific model component for the purposes of linearization and tuning. Suppose you have a large-scale model capturing aircraft dynamics and you want to perform linear analysis on the airframe only. You can use permanent openings to exclude all other components of the model. Another example is when you have cascaded loops within your model and you want to analyze a specific loop.

`addOpening(s,blk,port_num)` adds the signal at the specified output port of the specified block as a permanent opening for `s`.

`addOpening(s,blk,port_num,bus_elem_name)` adds the specified bus element as a permanent opening.

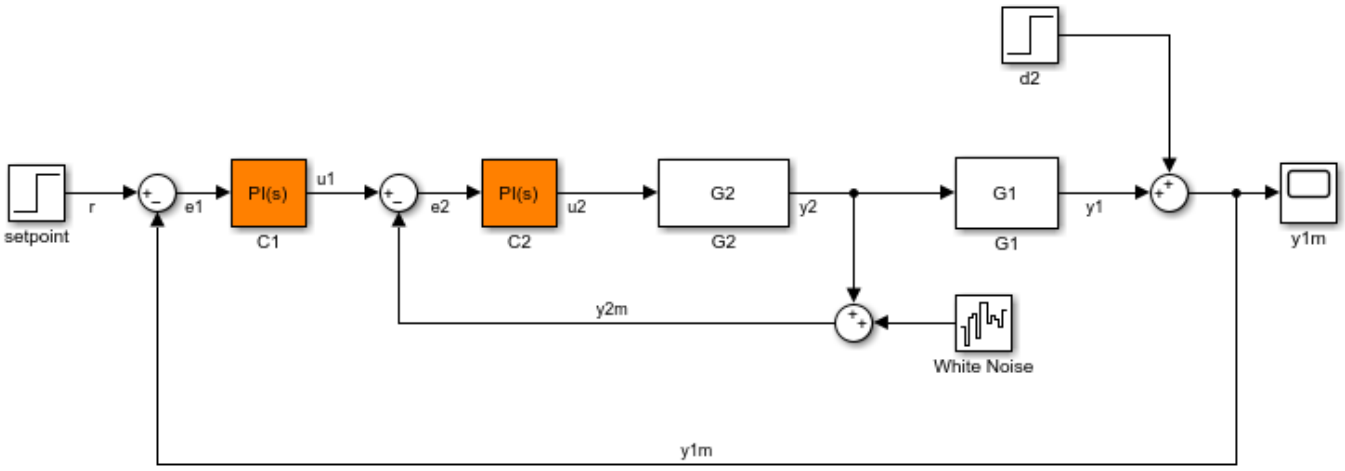
Examples

Add Opening Using Signal Name

Suppose you want to analyze only the inner-loop dynamics of the `scdcascade` model. Add the outer-loop feedback signal, `y1m`, as a permanent opening of an `slLinearizer` interface.

Open the `scdcascade` model.

```
mdl = 'scdcascade';
open_system(mdl)
```



Create an `slLinearizer` interface for the model.

```
sllin = slLinearizer(mdl);
```

Add the `y1m` signal as a permanent opening of `sllin`.

```
addOpening(sllin,'y1m');
```

View the currently defined analysis points within `sllin`.

```
sllin
```

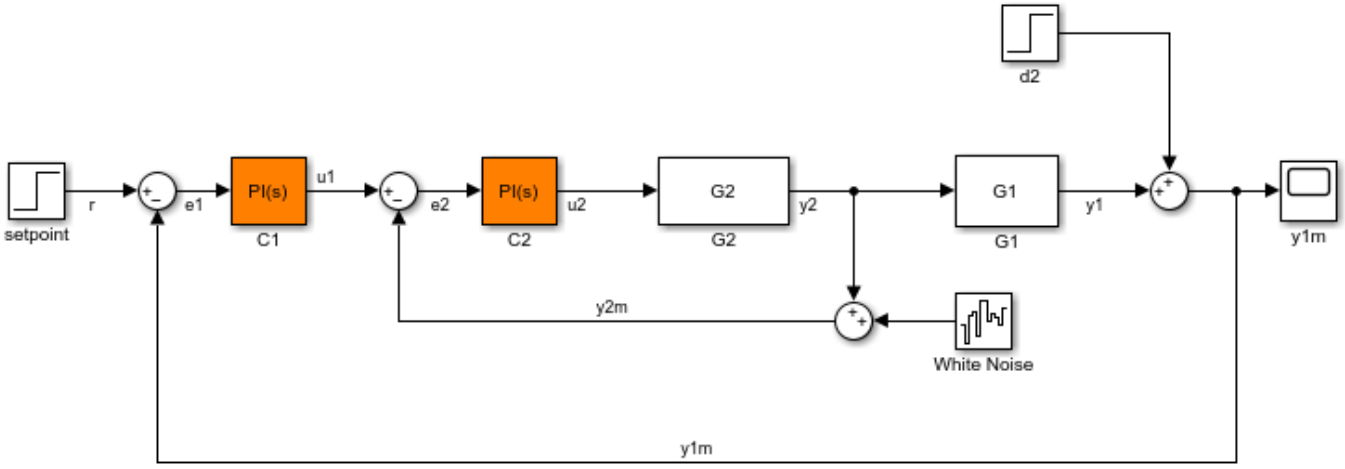
```
slLinearizer linearization interface for "scdcascade":

No analysis points. Use the addPoint command to add new points.
1 Permanent openings:
--------------------------
Opening 1:
- Block: scdcascade/Sum
- Port: 1
- Signal Name: y1m

Properties with dot notation get/set access:
      Parameters         : []
      OperatingPoints    : [] (model initial condition will be used.)
      BlockSubstitutions : []
      Options            : [1x1 linearize.LinearizeOptions]
```

Add Opening Using Block Path and Port Number

Suppose you want to analyze only the inner-loop dynamics of the `scdcascade` model. Add the outer-loop feedback signal, `y1m`, as a permanent opening of an `slLinearizer` interface.

Open the `scdcascade` model.

```
mdl = 'scdcascade';
open_system(mdl)
```



Create an `slLinearizer` interface for the model.

```
sllin = slLinearizer(mdl);
```

Add the `y1m` signal as a permanent opening of `sllin`.

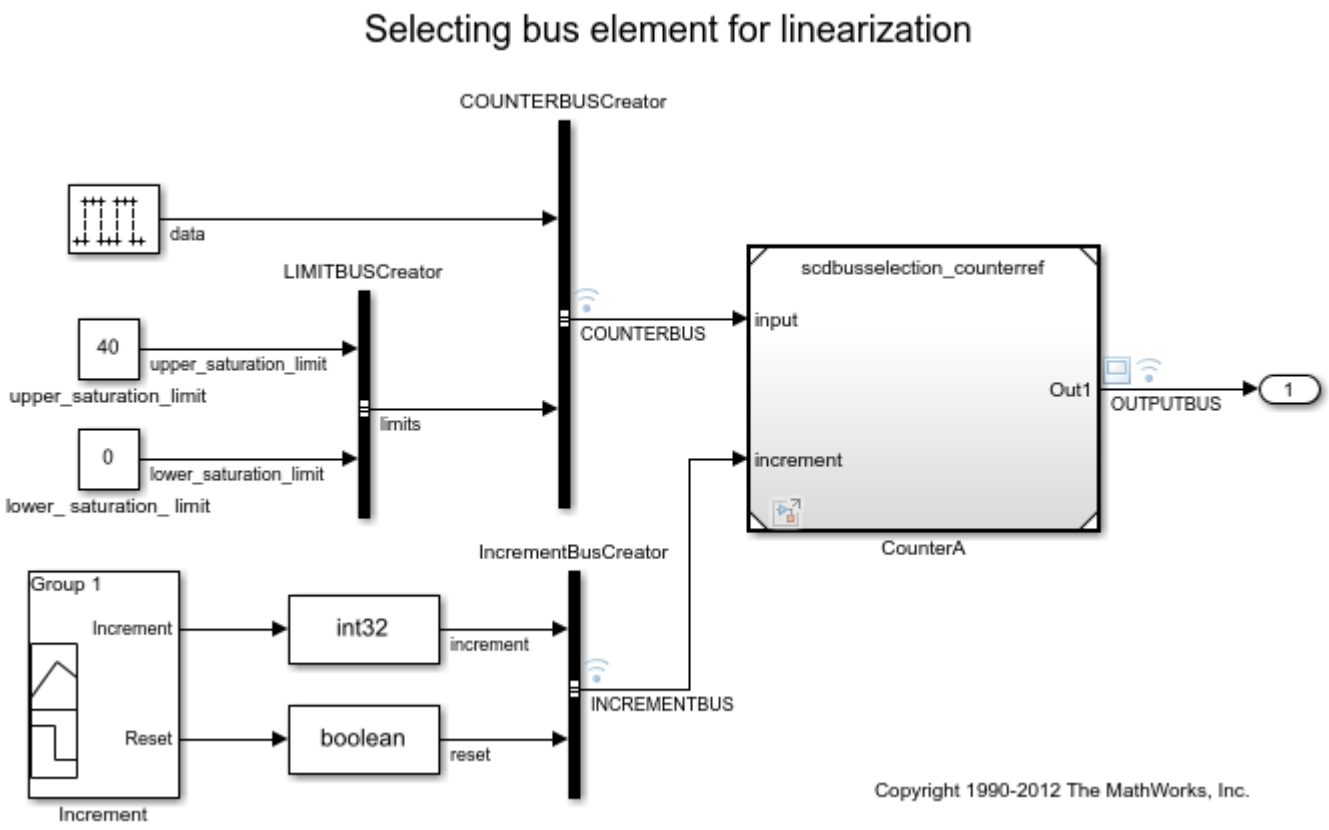
```
addOpening(sllin,'scdcascade/Sum',1);
```

The `y1m` signal originates at the first (and only) port of the `scdcascade/Sum` block.

Add Bus Elements as Openings

Open the `scdbusselection` model.

```
mdl = 'scdbusselection';
open_system(mdl);
```



Create an `slLinearizer` interface for the model.

```
sllin = slLinearizer(mdl);
```

The `COUNTERBUS` signal of `scdbusselection` contains multiple bus elements. Add the `upper_saturation_limit` and `data` bus elements as openings to `sllin`. When adding elements within a nested bus structure, use dot notation to access the elements of the nested bus, for example `limits.upper_saturation_limit`.

```
blk = {'scdbusselection/COUNTERBUSCreator','scdbusselection/COUNTERBUSCreator'};
port_num = [1 1];
bus_elem_name = {'limits.upper_saturation_limit','data'};
```

Both bus elements originate at the first (and only) port of the `scdbusselection/COUNTERBUSCreator` block. Therefore, `blk` and `port_num` repeat the same element twice.

Input Arguments

s — Interface to Simulink model

`slLinearizer` interface | `slTuner` interface

Interface to a Simulink model, specified as either an `slLinearizer` interface or an `slTuner` interface.

`pt` — Opening

character vector | string | cell array of character vectors | string array | vector of linearization I/O objects

Opening to add to the list of permanent openings on page 18-224 for `s`, specified as:

- Character vector or string — Signal identifier that can be any of the following:
  - Signal name, for example `'torque'`
  - Block path for a block with a single output port, for example `'Motor/PID'`
  - Path to block and port originating the signal, for example `'Engine Model/1'` or `'Engine Model/torque'`
- Cell array of character vectors or string array — Specifies multiple signal identifiers. For example, `pt = {'Motor/PID','Engine Model/1'}`.
- Vector of linearization I/O objects — Use `linio` to create `pt`. For example:

  ```
  pt(1) = linio('scdcascade/setpoint',1)
  pt(2) = linio('scdcascade/Sum',1,'output')
  ```

  Here, `pt(1)` specifies an input, and `pt(2)` specifies an output. However, the software ignores the I/O types and adds them both to the list of permanent openings for `s`.

`blk` — Block path identifying block where opening originates

character vector (default) | string | cell array of character vectors | string array

Block path identifying the block where the opening originates, specified as a character vector or cell array of character vectors.

Dimensions of `blk`:

- For a single opening, specify `blk` as a character vector or string.

  For example, `blk = 'scdcascade/C1'`.
- For multiple openings, specify `blk` as a cell array of character vectors or string array. `blk`, `port_num`, and `bus_elem_name` (if specified) must have the same size.

  For example, `blk = {'scdcascade/C1','scdcascade/Sum'}`.

`port_num` — Port where opening originates

positive integer (default) | vector of positive integers

Port where the opening originates, specified as a positive integer or a vector of positive integers.

Dimensions of `port_num`:

- For a single opening, specify `port_num` as a positive integer.

  For example, `port_num = 1`.
- For multiple openings, specify `port_num` as a vector of positive integers. `blk`, `port_num`, and `bus_elem_name` (if specified) must have the same size.

  For example, `port_num = [1 1]`.

bus_elem_name — Bus element name
character vector (default) | string | cell array of character vectors | string array

Bus element name, specified as a character vector or cell array of character vectors.

Dimensions of `bus_elem_name`:

- For a single opening, specify `bus_elem_name` as a character vector or string.

  For example, `bus_elem_name = 'data'`.
- For multiple openings, specify `bus_elem_name` as a cell array of character vectors or string array. `blk`, `port_num`, and `bus_elem_name` (if specified) must have the same size.

  For example, `bus_elem_name = {'limits.upper_saturation_limit','data'}`.

More About

Permanent Openings

Permanent openings, used by the `slLinearizer` and `slTuner` interfaces, identify locations within a model where the software breaks the signal flow. The software enforces these openings for linearization and tuning. Use permanent openings to isolate a specific model component. Suppose that you have a large-scale model capturing aircraft dynamics and you want to perform linear analysis on the airframe only. You can use permanent openings to exclude all other components of the model. Another example is when you have cascaded loops within your model and you want to analyze a specific loop.

Location refers to a specific block output port within a model. For convenience, you can use the name of the signal that originates from this port to refer to an opening.

You can add permanent openings to an `slLinearizer` or `slTuner` interface, `s`, when you create the interface or by using the `addOpening` command. To remove a location from the list of permanent openings, use the `removeOpening` command.

To view all the openings of `s`, type `s` at the command prompt to display the interface contents. For each permanent opening of `s`, the display includes the block name and port number and the name of the signal that originates at this location. You can also programmatically obtain a list of all the permanent loop openings using `getOpenings`.

See Also

`slLinearizer` | `slTuner` | `addPoint` | `addBlock` | `linio` | `removeOpening` | `removeAllOpenings`

Introduced in R2013b

addPoint

Add signal to list of analysis points for `slLinearizer` or `slTuner` interface

Syntax

```
addPoint(s,pt)

addPoint(s,blk,port_num)
addPoint(s,blk,port_num,bus_elem_name)
```

Description

`addPoint(s,pt)` adds the specified point to the list of analysis points on page 18-230 for the `slLinearizer` or `slTuner` interface, `s`.

Analysis points are model signals that can be used as input, output, or loop-opening locations for analysis and tuning purposes. You use analysis points as inputs to the linearization commands of `s`: `getIOTransfer`, `getLoopTransfer`, `getSensitivity`, and `getCompSensitivity`. As inputs to the linearization commands, analysis points can specify any open- or closed-loop transfer function in a model. You can also use analysis points to specify tuning goals for `systune`.

`addPoint(s,blk,port_num)` adds the point that originates at the specified output port of the specified block as an analysis point for `s`.

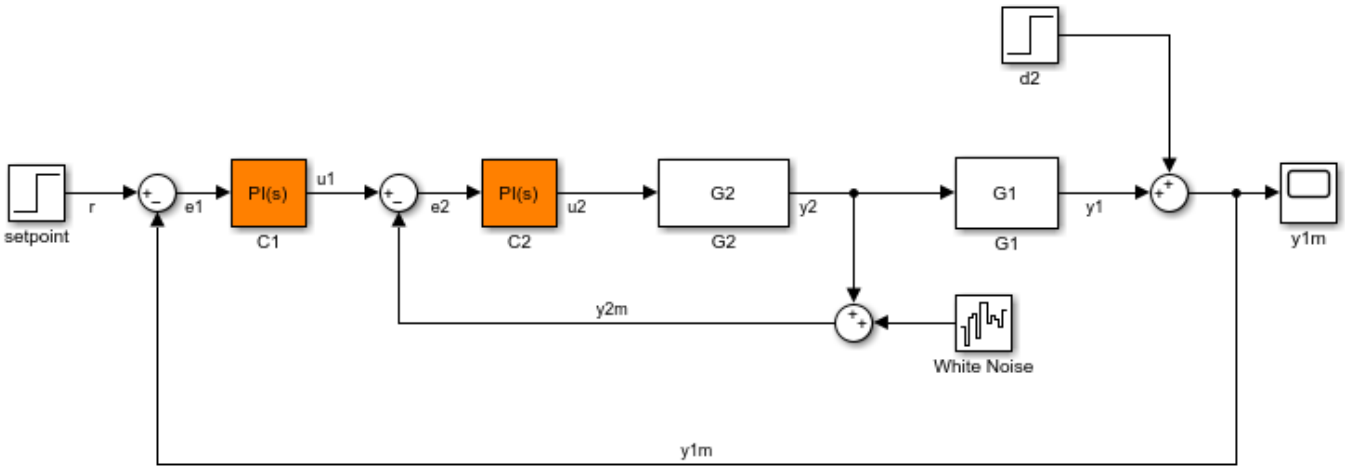
`addPoint(s,blk,port_num,bus_elem_name)` adds the specified bus element as an analysis point.

Examples

Add Analysis Point Using Signal Name

Open the `scdcascade` model.

```
mdl = 'scdcascade';
open_system(mdl);
```



Create an `slLinearizer` interface for the model.

```
sllin = slLinearizer(mdl);
```

Add `u1` and `y1` as analysis points for `sllin`.

```
addPoint(sllin,{'u1','y1'});
```

View the currently defined analysis points within `sllin`.

```
sllin
```

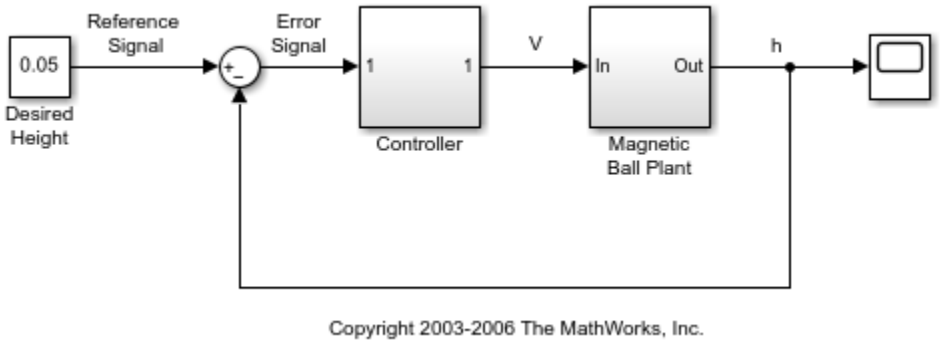
```
slLinearizer linearization interface for "scdcascade":

2 Analysis points:
--------------------------
Point 1:
- Block: scdcascade/C1
- Port: 1
- Signal Name: u1
Point 2:
- Block: scdcascade/G1
- Port: 1
- Signal Name: y1

No permanent openings. Use the addOpening command to add new permanent openings.
Properties with dot notation get/set access:
      Parameters         : []
      OperatingPoints    : [] (model initial condition will be used.)
      BlockSubstitutions : []
      Options            : [1x1 linearize.LinearizeOptions]
```

Add Analysis Points Using Block Path and Port Number

Suppose you want to linearize the magball model and obtain a transfer function from the reference input to the plant output. Add the signals originating at the `Desired Height` and `Magnetic Ball Plant` blocks as analysis points to an `slLinearizer` interface.

Open the `magball` model.

```
mdl = 'magball';
open_system(mdl);
```



Create an `slLinearizer` interface for the model.

```
sllin = slLinearizer(mdl);
```

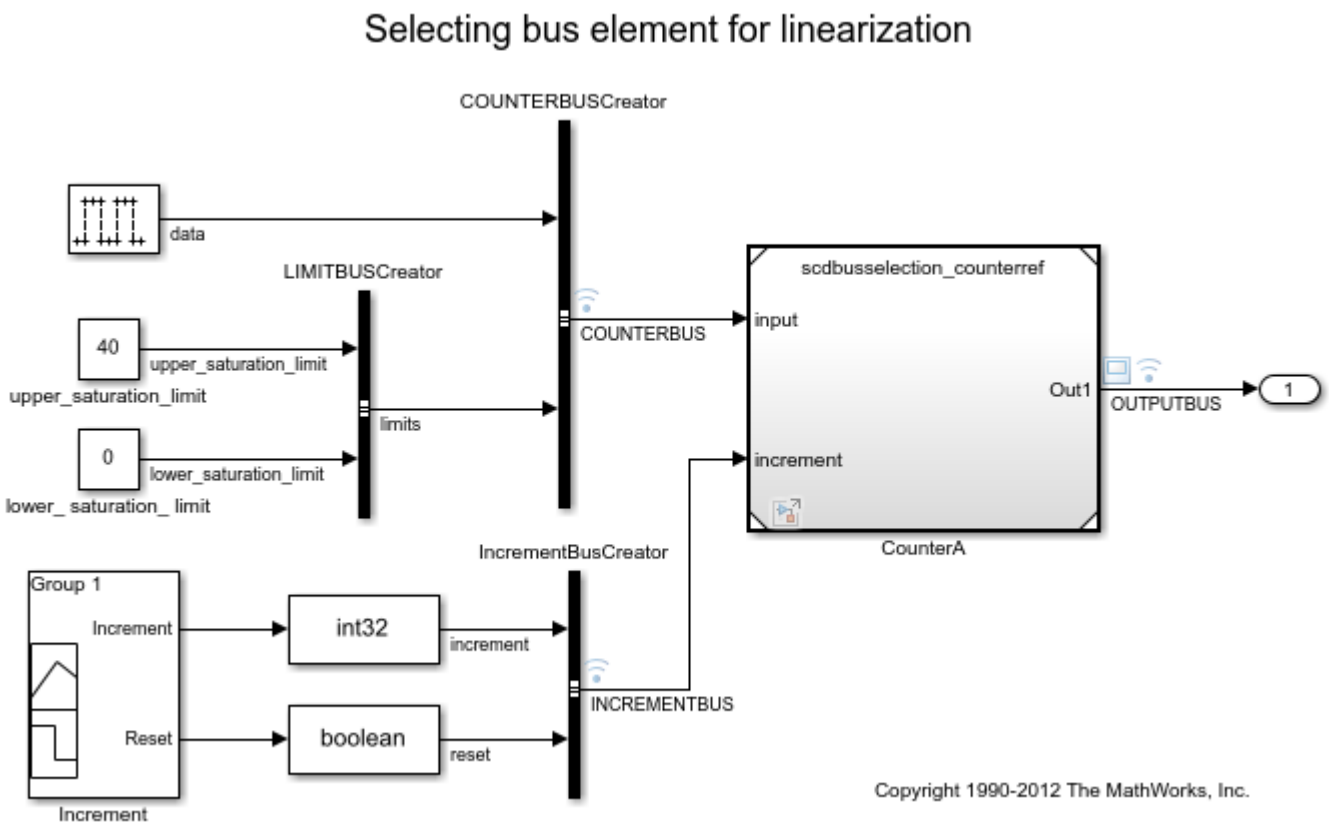
Add the signals originating at the `Design Height` and `Magnetic Ball Plant` blocks as analysis points of `sllin`. Both signals originate at the first (and only) port of the respective blocks.

```
blk = {'magball/Desired Height','magball/Magnetic Ball Plant'};
port_num = [1 1];
addPoint(sllin,blk,port_num);
```

Add Bus Elements as Analysis Points

Open the `scdbusselection` model.

```
mdl = 'scdbusselection';
open_system(mdl);
```



Create an `slLinearizer` interface model.

```
sllin = slLinearizer(mdl);
```

The `COUNTERBUS` signal of `scdbusselection` contains multiple bus elements. Add the `upper_saturation_limit` and `data` bus elements as analysis points to `sllin`. When adding elements within a nested bus structure, use dot notation to access the elements of the nested bus, for example `limits.upper_saturation_limit`.

```
blk = {'scdbusselection/COUNTERBUSCreator','scdbusselection/COUNTERBUSCreator'};
port_num = [1 1];
bus_elem_name = {'limits.upper_saturation_limit','data'};
addPoint(sllin,blk,port_num,bus_elem_name);
```

Both bus elements originate at the first (and only) port of the `scdbusselection/COUNTERBUSCreator` block. Therefore, `blk` and `port_num` repeat the same element twice.

Input Arguments

s — Interface to Simulink model

`slLinearizer` interface | `slTuner` interface

Interface to a Simulink model, specified as either an `slLinearizer` interface or an `slTuner` interface.

`pt` — Analysis point

character vector | string | cell array of character vectors | string array | vector of linearization I/O objects

Analysis point to add to the list of analysis points on page 18-230 for `s`, specified as:

- Character vector or string — Signal identifier that can be any of the following:
  - Signal name, for example `'torque'`
  - Block path for a block with a single output port, for example `'Motor/PID'`
  - Path to block and port originating the signal, for example `'Engine Model/1'` or `'Engine Model/torque'`
- Cell array of character vectors or string array — Specifies multiple signal identifiers.
- Vector of linearization I/O objects — Use `linio` to create `pt`. For example:

  ```
  pt(1) = linio('scdcascade/setpoint',1)
  pt(2) = linio('scdcascade/Sum',1,'output')
  ```

  Here, `pt(1)` specifies an input, and `pt(2)` specifies an output. The interface adds all the signals specified by `pt` and ignores the I/O types. The interface also adds all `'loopbreak'` type signals as permanent openings.

`blk` — Block path identifying block where analysis point originates

character vector (default) | string | cell array of character vectors | string array

Block path identifying the block where the analysis point originates, specified as a:

- Character vector or string to specify a single point, for example `blk = 'scdcascade/C1'`.
- Cell array of character vectors or string array to specify multiple points, for example `blk = {'scdcascade/C1','scdcascade/Sum'}`.

`blk`, `port_num`, and `bus_elem_name` (if specified) must have the same size.

`port_num` — Port where analysis point originates

positive integer (default) | vector of positive integers

Port where the analysis point originates, specified as a:

- Positive integer to specify a single point, for example `port_num = 1`.
- Vector of positive integers to specify multiple points, for example `port_num = [1 1]`.

`blk`, `port_num`, and `bus_elem_name` (if specified) must have the same size.

`bus_elem_name` — Bus element name

character vector (default) | string | cell array of character vectors | string array

Bus element name, specified as a:

- Character vector or string to specify a single point, for example `bus_elem_name = 'data'`.
- Cell array of character vectors or string array to specify multiple points, for example `bus_elem_name = {'limits.upper_saturation_limit','data'}`.

`blk`, `port_num`, and `bus_elem_name` (if specified) must have the same size.

More About

Analysis Points

Analysis points, used by the `slLinearizer` and `slTuner` interfaces, identify locations within a model that are relevant for linear analysis and control system tuning. You use analysis points as inputs to the linearization commands, such as `getIOTransfer`, `getLoopTransfer`, `getSensitivity`, and `getCompSensitivity`. As inputs to the linearization commands, analysis points can specify any open-loop or closed-loop transfer function in a model. You can also use analysis points to specify design requirements when tuning control systems using commands such as `systune`.

Location refers to a specific block output port within a model or to a bus element in such an output port. For convenience, you can use the name of the signal that originates from this port to refer to an analysis point.

You can add analysis points to an `slLinearizer` or `slTuner` interface, `s`, when you create the interface. For example:

```
s = slLinearizer('scdcascade',{'u1','y1'});
```

Alternatively, you can use the `addPoint` command.

To view all the analysis points of `s`, type `s` at the command prompt to display the interface contents. For each analysis point of `s`, the display includes the block name and port number and the name of the signal that originates at this point. You can also programmatically obtain a list of all the analysis points using `getPoints`.

For more information about how you can use analysis points, see “Mark Signals of Interest for Control System Analysis and Design” on page 2-38 and “Mark Signals of Interest for Batch Linearization” on page 3-9.

Permanent Openings

Permanent openings, used by the `slLinearizer` and `slTuner` interfaces, identify locations within a model where the software breaks the signal flow. The software enforces these openings for linearization and tuning. Use permanent openings to isolate a specific model component. Suppose that you have a large-scale model capturing aircraft dynamics and you want to perform linear analysis on the airframe only. You can use permanent openings to exclude all other components of the model. Another example is when you have cascaded loops within your model and you want to analyze a specific loop.

Location refers to a specific block output port within a model. For convenience, you can use the name of the signal that originates from this port to refer to an opening.

You can add permanent openings to an `slLinearizer` or `slTuner` interface, `s`, when you create the interface or by using the `addOpening` command. To remove a location from the list of permanent openings, use the `removeOpening` command.

To view all the openings of `s`, type `s` at the command prompt to display the interface contents. For each permanent opening of `s`, the display includes the block name and port number and the name of the signal that originates at this location. You can also programmatically obtain a list of all the permanent loop openings using `getOpenings`.

See Also

`slLinearizer` | `slTuner` | `addOpening` | `linio` | `removePoint` | `removeAllPoints`

Introduced in R2013b

getCompSensitivity

Complementary sensitivity function at specified point using `slLinearizer` or `slTuner` interface

Syntax

```
linsys = getCompSensitivity(s,pt)
linsys = getCompSensitivity(s,pt,temp_opening)
linsys = getCompSensitivity( ___ ,mdl_index)

[linsys,info] = getCompSensitivity( ___ )
```

Description

`linsys = getCompSensitivity(s,pt)` returns the complementary sensitivity function on page 18-241 at the specified analysis point for the model associated with the `slLinearizer` or `slTuner` interface, `s`.

The software enforces all the permanent openings on page 18-242 specified for `s` when it calculates `linsys`. If you configured either `s.Parameters`, or `s.OperatingPoints`, or both, `getCompSensitivity` performs multiple linearizations and returns an array of complementary sensitivity functions.

`linsys = getCompSensitivity(s,pt,temp_opening)` considers additional, temporary, openings at the point specified by `temp_opening`. Use an opening, for example, to calculate the complementary sensitivity function of an inner loop with the outer loop open.

`linsys = 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.

`[linsys,info] = getCompSensitivity( ___ )` returns additional linearization information.

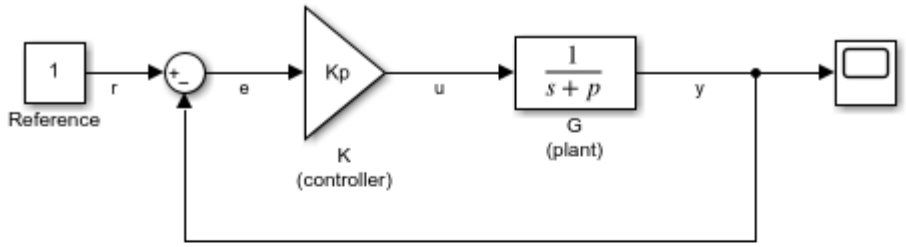
Examples

Obtain Complementary Sensitivity Function at Analysis Point

Obtain the complementary sensitivity function, calculated at the plant output, for the `ex_scd_simple_fdbk` model.

Open the `ex_scd_simple_fdbk` model.

```
mdl = 'ex_scd_simple_fdbk';
open_system(mdl);
```



Copyright 2015-2021 The MathWorks, Inc.

In this model:

$$K(s) = K_p = 3$$
$$G(s) = \frac{1}{s+5}$$

Create an `slLinearizer` interface for the model.

```
sllin = slLinearizer(mdl);
```

To calculate the complementary sensitivity function at the plant output, use the `y` signal as the analysis point. Add this point to `sllin`.

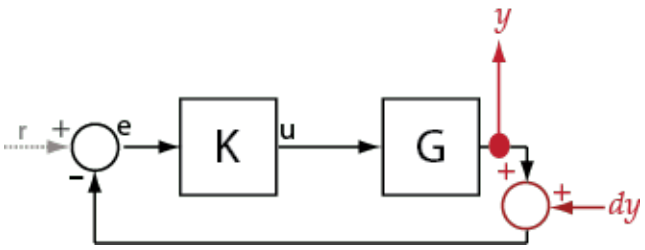
```
addPoint(sllin,'y');
```

Obtain the complementary sensitivity function at `y`.

```
sys = getCompSensitivity(sllin,'y');
tf(sys)
```

```
ans =

  From input "y" to output "y":
   -3
  -----
  s + 8

Continuous-time transfer function.
```

The software adds a linearization output at `y`, followed by a linearization input, `dy`.



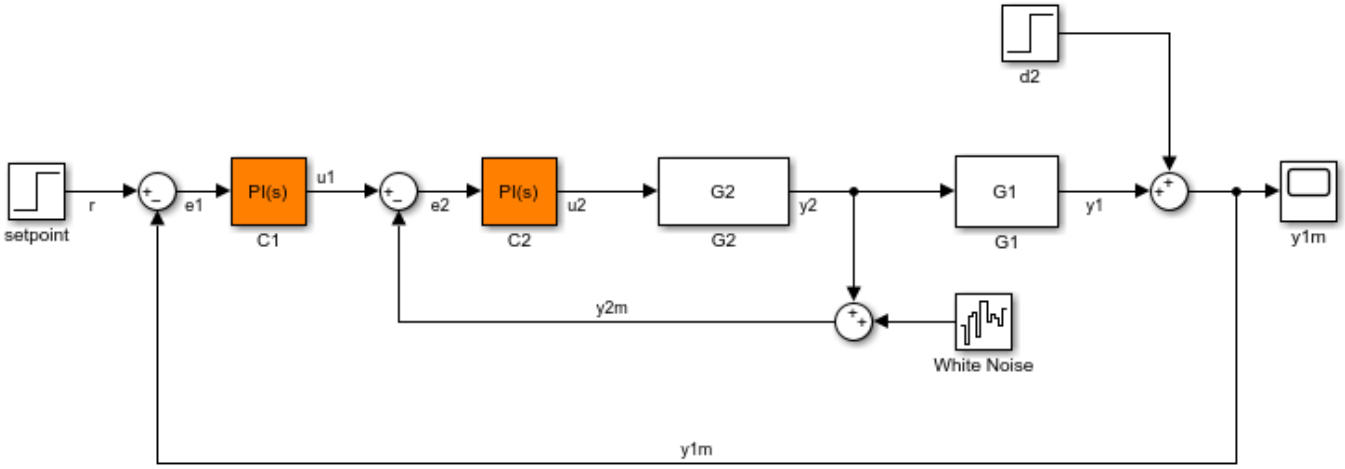
`sys` is the transfer function from `dy` to `y`, which is equal to $-(I+GK)^{-1}GK$.

Specify Temporary Loop Opening for Complementary Sensitivity Function Calculation

For the `scdcascade` model, obtain the complementary sensitivity function for the inner-loop at `y2`.

Open the `scdcascade` model.

```
mdl = 'scdcascade';
open_system(mdl)
```



Create an `slLinearizer` interface for the model.

```
sllin = slLinearizer(mdl);
```

To calculate the complementary sensitivity transfer function for the inner loop at `y2`, use the `y2` signal as the analysis point. To eliminate the effects of the outer loop, break the outer loop at `y1m`. Add both these points to `sllin`.

```
addPoint(sllin,{'y2','y1m'});
```

Obtain the complementary sensitivity function for the inner loop at `y2`.

```
sys = getCompSensitivity(sllin,'y2','y1m');
```

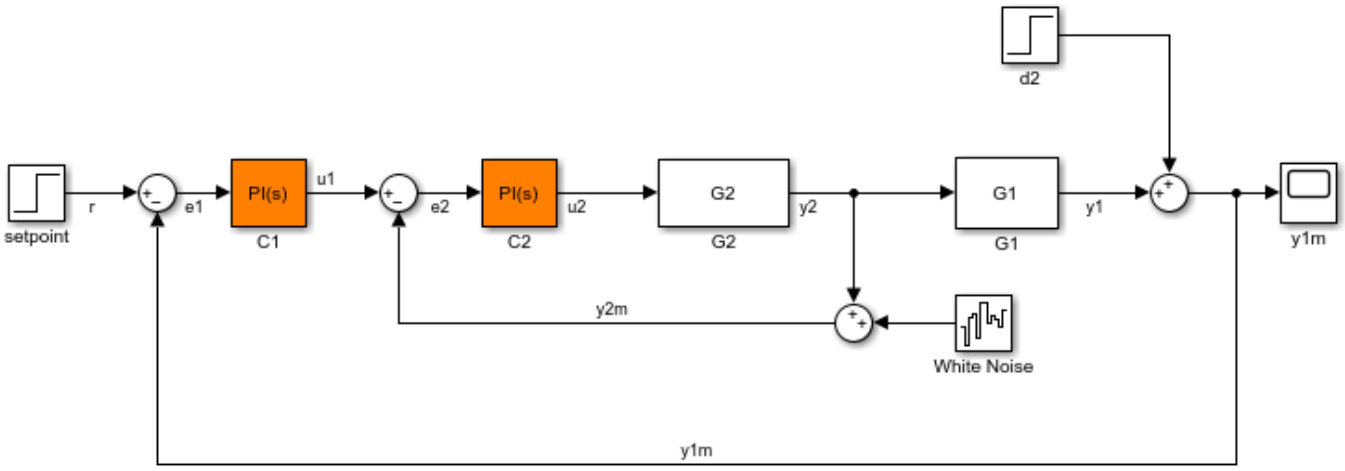
Here, `'y1m'`, the third input argument, specifies a temporary opening for the outer loop.

Obtain Complementary Sensitivity Function for Specific Parameter Combination

Suppose you batch linearize the `scdcascade` model for multiple transfer functions. For most linearizations, you vary the proportional (`Kp2`) and integral gain (`Ki2`) of the `C2` controller in the 10% range. For this example, calculate the complementary sensitivity function for the inner loop for the maximum value of `Kp2` and `Ki2`.

Open the `scdcascade` model.

```
mdl = 'scdcascade';
open_system(mdl);
```



Create an `slLinearizer` interface for the model.

```
sllin = slLinearizer(mdl);
```

Vary the proportional (`Kp2`) and integral gain (`Ki2`) of the `C2` controller in the 10% range.

```
Kp2_range = linspace(0.9*Kp2,1.1*Kp2,3);
Ki2_range = linspace(0.9*Ki2,1.1*Ki2,5);

[Kp2_grid,Ki2_grid]=ndgrid(Kp2_range,Ki2_range);

params(1).Name = 'Kp2';
params(1).Value = Kp2_grid;

params(2).Name = 'Ki2';
params(2).Value = Ki2_grid;

sllin.Parameters = params;
```

To calculate the complementary sensitivity of the inner loop, use the `y2` signal as the analysis point. To eliminate the effects of the outer loop, break the outer loop at `y1m`. Add both these points to `sllin`.

```
addPoint(sllin,{'y2','y1m'})
```

Determine the index for the maximum values of `Ki2` and `Kp2`.

```
mdl_index = params(1).Value == max(Kp2_range) & params(2).Value == max(Ki2_range);
```

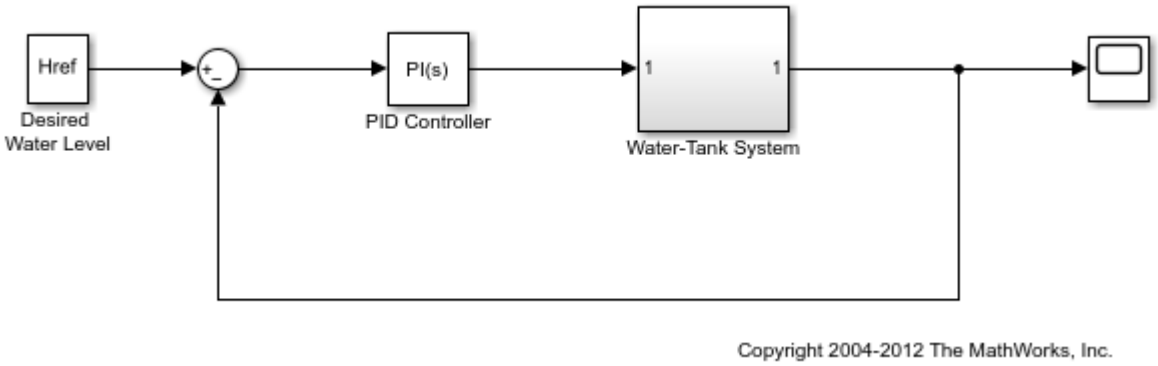
Obtain the complementary sensitivity transfer function at `y2`.

```
sys = getCompSensitivity(sllin,'y2','y1m',mdl_index);
```

Obtain Offsets from Complementary Sensitivity Function

Open Simulink model.

```
mdl = 'watertank';
open_system(mdl)
```



Create a linearization option set, and set the `StoreOffsets` option.

```
opt = linearizeOptions('StoreOffsets',true);
```

Create `slLinearizer` interface.

```
sllin = slLinearizer(mdl,opt);
```

Add an analysis point at the tank output port.

```
addPoint(sllin,'watertank/Water-Tank System');
```

Calculate the complementary sensitivity function at `y`, and obtain the corresponding linearization offsets.

```
[sys,info] = getCompSensitivity(sllin,'watertank/Water-Tank System');
```

View offsets.

```
info.Offsets
```

```
ans =

  struct with fields:

            x: [2x1 double]
           dx: [2x1 double]
            u: 1
            y: 1
    StateName: {2x1 cell}
```

```
     InputName: {'watertank/Water-Tank System'}
    OutputName: {'watertank/Water-Tank System'}
            Ts: 0
```

Input Arguments

s — Interface to Simulink model

`slLinearizer` interface | `slTuner` interface

Interface to a Simulink model, specified as either an `slLinearizer` interface or an `slTuner` interface.

pt — Analysis point signal name

character vector | string | cell array of character vectors | string array

Analysis point on page 18-242 signal name, specified as:

- Character vector or string — Analysis point signal name.

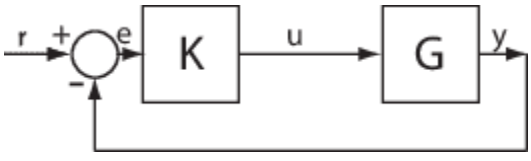
  To determine the signal name associated with an analysis point, type `s`. The software displays the contents of `s` in the MATLAB command window, including the analysis point signal names, block names, and port numbers. Suppose that an analysis point does not have a signal name, but only a block name and port number. You can specify `pt` as the block name. To use a point not in the list of analysis points for `s`, first add the point using `addPoint`.

  You can specify `pt` as a uniquely matching portion of the full signal name or block name. Suppose that the full signal name of an analysis point is `'LoadTorque'`. You can specify `pt` as `'Torque'` as long as `'Torque'` is not a portion of the signal name for any other analysis point of `s`.

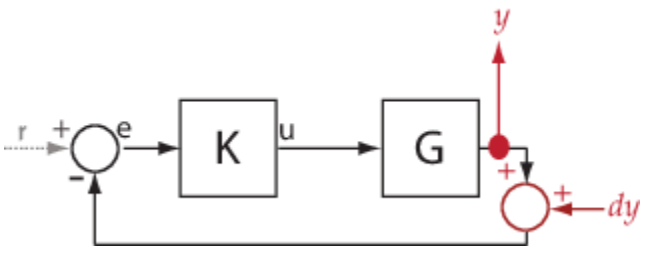
  For example, `pt = 'y1m'`.
- Cell array of character vectors or string array — Specifies multiple analysis point names. For example, `pt = {'y1m','y2m'}`.

To calculate `linsys`, the software adds a linearization output, followed by a linearization input at `pt`.

Consider the following model:

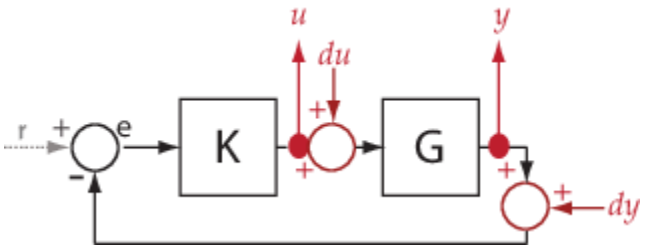


Specify `pt` as `'y'`:



The software computes `linsys` 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 `linsys` as a MIMO transfer function with a transfer function from each linearization input to each linearization output.

`temp_opening` — Temporary opening signal name
character vector | string | cell array of character vectors | string array

Temporary opening signal name, specified as:

- Character vector or string — Analysis point signal name.

  `temp_opening` must specify an analysis point that is in the list of analysis points for `s`. To determine the signal name associated with an analysis point, type `s`. The software displays the contents of `s` in the MATLAB command window, including the analysis point signal names, block names, and port numbers. Suppose that an analysis point does not have a signal name, but only a block name and port number. You can specify `temp_opening` as the block name. To use a point not in the list of analysis points for `s`, first add the point using `addPoint`.

  You can specify `temp_opening` as a uniquely matching portion of the full signal name or block name. Suppose that the full signal name of an analysis point is `'LoadTorque'`. You can specify `temp_opening` as `'Torque'` as long as `'Torque'` is not a portion of the signal name for any other analysis point of `s`.

  For example, `temp_opening = 'y1m'`.
- Cell array of character vectors or string array — Specifies multiple analysis point names. For example, `temp_opening = {'y1m','y2m'}`.

`mdl_index` — Index for linearizations of interest
array of logical values | vector of positive integers

Index for linearizations of interest, specified as:

- Array of logical values — Logical array index of linearizations of interest. Suppose that you vary two parameters, `par1` and `par2`, and want to extract the linearization for the combination of `par1 > 0.5` and `par2 <= 5`. Use:

  ```
  params = s.Parameters;
  mdl_index = params(1).Value>0.5 & params(2).Value <= 5;
  ```

  The expression `params(1).Value>0.5 & params(2).Value<5` uses logical indexing and returns a logical array. This logical array is the same size as `params(1).Value` and

`params(2).Value`. Each entry contains the logical evaluation of the expression for corresponding entries in `params(1).Value` and `params(2).Value`.

- Vector of positive integers — Linear index of linearizations of interest. Suppose that you vary two parameters, `par1` and `par2`, and want to extract the linearization for the combination of `par1 > 0.5` and `par2 <= 5`. Use:

  ```
  params = s.Parameters;
  mdl_index = find(params(1).Value>0.5 & params(2).Value <= 5);
  ```

  The expression `params(1).Value>0.5 & params(2).Value<5` returns a logical array. `find` returns the linear index of every true entry in the logical array

Output Arguments

`linsys` — Complementary sensitivity function

state-space model

Complementary sensitivity function, returned as described in the following:

- If you did not configure `s.Parameters` and `s.OperatingPoints`, the software calculates `linsys` using the default model parameter values. The software uses the model initial conditions as the linearization operating point. `linsys` is returned as a state-space model.
- If you configured `s.Parameters` only, the software computes a linearization for each parameter grid point. `linsys` is returned as a state-space model array of the same size as the parameter grid.
- If you configured `s.OperatingPoints` only, the software computes a linearization for each specified operating point. `linsys` is returned as a state-space model array of the same size as `s.OperatingPoints`.
- If you configured `s.Parameters` and specified `s.OperatingPoints` as a single operating point, the software computes a linearization for each parameter grid point. The software uses the specified operating point as the linearization operating point. `linsys` is returned as a state-space model array of the same size as the parameter grid.
- If you configured `s.Parameters` and specified `s.OperatingPoints` as multiple operating point objects, the software computes a linearization for each parameter grid point. The software requires that `s.OperatingPoints` is the same size as the parameter grid specified by `s.Parameters`. The software computes each linearization using corresponding operating points and parameter grid points. `linsys` is returned as a state-space model array of the same size as the parameter grid.
- If you configured `s.Parameters` and specified `s.OperatingPoints` as multiple simulation snapshot times, the software simulates and linearizes the model for each snapshot time and parameter grid point combination. Suppose that you specify a parameter grid of size `p` and `N` snapshot times. `linsys` is returned as a state-space model array of size `N`-by-`p`.

For most models, `linsys` is returned as an `ss` object or an array of `ss` objects. However, if your model contains one of the following blocks in the linearization path defined by `pt`, then `linsys` returns the specified type of state-space model.

Block	`linsys` Type
Block with a substitution specified as a `genss` object or tunable model object	`genss`

Block	linsys Type
Block with a substitution specified as an uncertain model, such as `uss`	`uss`
Sparse Second Order block	`mechss`
Descriptor State-Space block configured to linearize to a sparse model	`sparss`

`info` — Linearization information
structure

Linearization information, returned as a structure with the following fields:

`Offsets` — Linearization offsets
`[]` (default) | structure | structure array

Linearization offsets, returned as `[]` if `s.Options.StoreOffsets` is `false`. Otherwise, `Offsets` is returned as one of the following:

- If `linsys` is a single state-space model, then `Offsets` is a structure.
- If `linsys` is an array of state-space models, then `Offsets` is a structure array with the same dimensions as `linsys`.

Each offset structure has the following fields:

Field	Description
`x`	State offsets used for linearization, returned as a column vector of length n_x, where n_x is the number of states in `linsys`.
`y`	Output offsets used for linearization, returned as a column vector of length n_y, where n_y is the number of outputs in `linsys`.
`u`	Input offsets used for linearization, returned as a column vector of length n_u, where n_u is the number of inputs in `linsys`.
`dx`	Derivative offsets for continuous time systems or updated state values for discrete-time systems, returned as a column vector of length n_x.
`StateName`	State names, returned as a cell array that contains n_x elements that match the names in `linsys.StateName`.
`InputName`	Input names, returned as a cell array that contains n_u elements that match the names in `linsys.InputName`.
`OutputName`	Output names, returned as a cell array that contains n_y elements that match the names in `linsys.OutputName`.
`Ts`	Sample time of the linearized system, returned as a scalar that matches the sample time in `linsys.Ts`. For continuous-time systems, `Ts` is `0`.

If `Offsets` is a structure array, you can configure an LPV System block using the offsets. To do so, first convert them to the required format using `getOffsetsForLPV`. For an example, see “Approximate Nonlinear Behavior Using Array of LTI Systems” on page 3-70.

`Advisor` — Linearization diagnostic information
`[]` (default) | `LinearizationAdvisor` object | array of `LinearizationAdvisor` objects

Linearization diagnostic information, returned as `[]` if `s.Options.StoreAdvisor` is `false`. Otherwise, `Advisor` is returned as one of the following:

- If `linsys` is a single state-space model, `Advisor` is a `LinearizationAdvisor` object.
- If `linsys` is an array of state-space models, `Advisor` is an array of `LinearizationAdvisor` objects with the same dimensions as `linsys`.

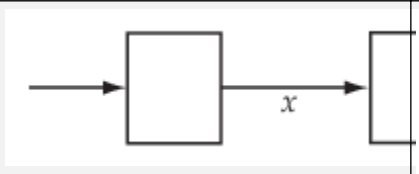
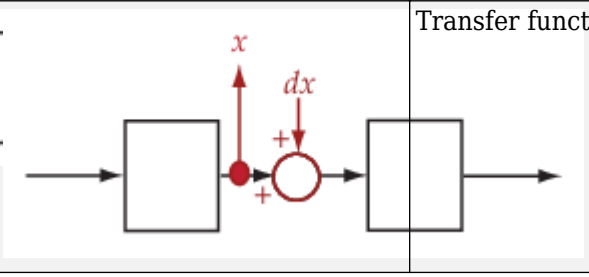
`LinearizationAdvisor` objects store linearization diagnostic information for individual linearized blocks. For an example of troubleshooting linearization results using a `LinearizationAdvisor` object, see “Troubleshoot Linearization Results at Command Line” on page 4-28.

More About

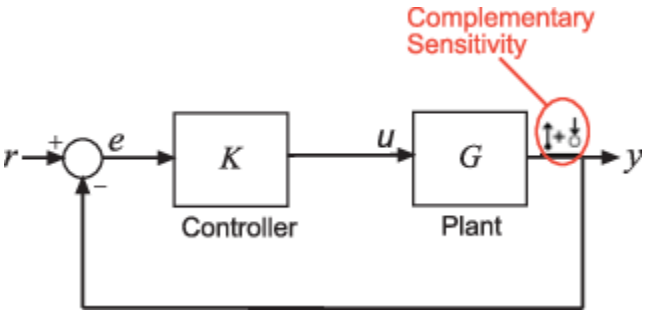
Complementary Sensitivity Function

The complementary sensitivity function at a point is the transfer function from an additive disturbance at the point to a measurement at the same point. In contrast to the sensitivity function, the disturbance is added *after* the measurement.

To compute the complementary sensitivity function at an analysis point, `x`, the software adds a linearization output at `x`, followed by a linearization input, `dx`. The complementary sensitivity function is the transfer function from `dx` to `x`.

Analysis Point in Simulink Model	How `getCompSensitivity` Interprets Analysis Point	Complementary Sensitivity Function
		Transfer function from `dx` to `x`

For example, consider the following model where you compute the complementary sensitivity function at `y`:



Here, the software adds a linearization output at `y`, followed by a linearization input, `dy`. The complementary sensitivity function at `y`, T, is the transfer function from `dy` to `y`. T is calculated as follows:

$$y = -GK(y + dy)$$
$$\rightarrow y = -GKy - GKdy$$
$$\rightarrow (I + GK)y = -GKdy$$
$$\rightarrow y = -(^{I}GKdy$$
$$\therefore T = -(^{I}GK$$

Here I is an identity matrix of the same size as GK. The complementary sensitivity transfer function at `y` is equal to `-1` times the closed-loop transfer function from `r` to `y`.

Generally, the complementary sensitivity function, T, computed from reference signals to plant outputs, is equal to $I–S$. Here S is the sensitivity function at the point, and I is the identity matrix of commensurate size. However, because `getCompSensitivity` adds the linearization output and input *at the same point*, T, as returned by `getCompSensitivity`, is equal to $S–I$.

The software does not modify the Simulink model when it computes the complementary sensitivity function.

Analysis Point

Analysis points, used by the `slLinearizer` and `slTuner` interfaces, identify locations within a model that are relevant for linear analysis and control system tuning. You use analysis points as inputs to the linearization commands, such as `getIOTransfer`, `getLoopTransfer`, `getSensitivity`, and `getCompSensitivity`. As inputs to the linearization commands, analysis points can specify any open-loop or closed-loop transfer function in a model. You can also use analysis points to specify design requirements when tuning control systems using commands such as `systune`.

Location refers to a specific block output port within a model or to a bus element in such an output port. For convenience, you can use the name of the signal that originates from this port to refer to an analysis point.

You can add analysis points to an `slLinearizer` or `slTuner` interface, `s`, when you create the interface. For example:

```
s = slLinearizer('scdcascade',{'u1','y1'});
```

Alternatively, you can use the `addPoint` command.

To view all the analysis points of `s`, type `s` at the command prompt to display the interface contents. For each analysis point of `s`, the display includes the block name and port number and the name of the signal that originates at this point. You can also programmatically obtain a list of all the analysis points using `getPoints`.

For more information about how you can use analysis points, see “Mark Signals of Interest for Control System Analysis and Design” on page 2-38 and “Mark Signals of Interest for Batch Linearization” on page 3-9.

Permanent Loop Openings

Permanent openings, used by the `slLinearizer` and `slTuner` interfaces, identify locations within a model where the software breaks the signal flow. The software enforces these openings for linearization and tuning. Use permanent openings to isolate a specific model component. Suppose that you have a large-scale model capturing aircraft dynamics and you want to perform linear

analysis on the airframe only. You can use permanent openings to exclude all other components of the model. Another example is when you have cascaded loops within your model and you want to analyze a specific loop.

Location refers to a specific block output port within a model. For convenience, you can use the name of the signal that originates from this port to refer to an opening.

You can add permanent openings to an `slLinearizer` or `slTuner` interface, `s`, when you create the interface or by using the `addOpening` command. To remove a location from the list of permanent openings, use the `removeOpening` command.

To view all the openings of `s`, type `s` at the command prompt to display the interface contents. For each permanent opening of `s`, the display includes the block name and port number and the name of the signal that originates at this location. You can also programmatically obtain a list of all the permanent loop openings using `getOpenings`.

See Also

`slLinearizer` | `slTuner` | `addPoint` | `addOpening` | `getIOTransfer` | `getLoopTransfer` | `getSensitivity`

Topics

“How the Software Treats Loop Openings” on page 2-31
“Vary Parameter Values and Obtain Multiple Transfer Functions” on page 3-21
“Vary Operating Points and Obtain Multiple Transfer Functions Using slLinearizer Interface” on page 3-28
“Analyze Command-Line Batch Linearization Results Using Response Plots” on page 3-33

Introduced in R2013b

getIOTransfer

Transfer function for specified I/O set using `slLinearizer` or `slTuner` interface

Syntax

```
linsys = getIOTransfer(s,in,out)
linsys = getIOTransfer(s,in,out,temp_opening)

linsys = getIOTransfer(s,ios)

linsys = getIOTransfer( ___ ,mdl_index)

[linsys,info] = getIOTransfer( ___ )
```

Description

`linsys = getIOTransfer(s,in,out)` returns the transfer function for the specified inputs and outputs on page 18-254 for the model associated with the `slLinearizer` or `slTuner` interface, `s`.

The software enforces all the permanent openings on page 18-258 specified for `s` when it calculates `linsys`. For information on how `getIOTransfer` treats `in` and `out`, see "Transfer Functions" on page 18-254. If you configured either `s.Parameters`, or `s.OperatingPoints`, or both, `getIOTransfer` performs multiple linearizations and returns an array of transfer functions.

`linsys = getIOTransfer(s,in,out,temp_opening)` considers additional, temporary, openings at the point specified by `temp_opening`. Use an opening, for example, to obtain the transfer function of the controller in series with the plant, with the feedback loop open.

`linsys = getIOTransfer(s,ios)` returns the transfer function for the inputs and outputs specified by `ios` for the model associated with `s`. Use the `linio` command to create `ios`. The software enforces the linearization I/O type of each signal specified in `ios` when it calculates `linsys`. The software also enforces all the permanent loop openings specified for `s`.

`linsys = 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.

`[linsys,info] = getIOTransfer( ___ )` returns additional linearization information.

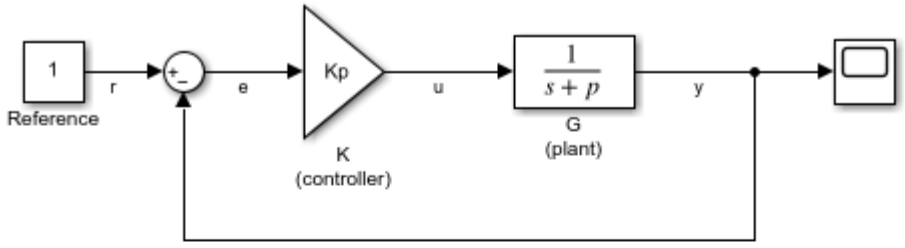
Examples

Obtain Closed-Loop Transfer Function from Reference to Plant Output

Obtain the closed-loop transfer function from the reference signal, `r`, to the plant output, `y`, for the `ex_scd_simple_fdbk` model.

Open the `ex_scd_simple_fdbk` model.

```
mdl = 'ex_scd_simple_fdbk';
open_system(mdl);
```



Copyright 2015-2021 The MathWorks, Inc.

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'});
```

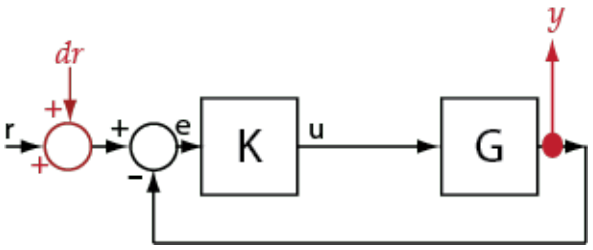
Obtain the closed-loop transfer function from `r` to `y`.

```
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`.



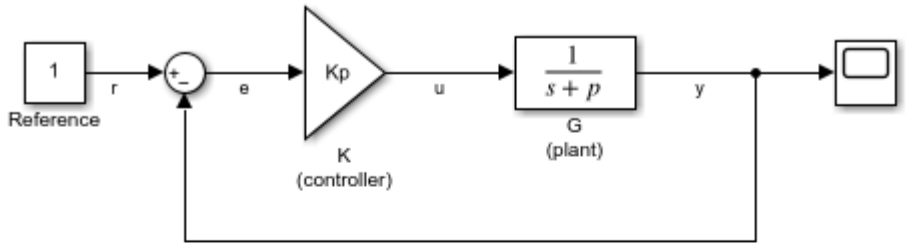
`sys` is the transfer function from `dr` to `y`, which is equal to $(I + GK)^{-1}GK$.

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);
```



Copyright 2015-2021 The MathWorks, Inc.

In this model:

$$K(s) = K_p = 3$$
$$G(s) = \frac{1}{s+5}.$$

Create an `slLinearizer` interface for the model.

```
sllin = slLinearizer(mdl);
```

To obtain the plant model transfer function, use `u` as the input point and `y` as the output point. To eliminate the effects of feedback, you must break the loop. You can break the loop at `u`, `e`, or `y`. For this example, break the loop at `u`. Add these points to `sllin`.

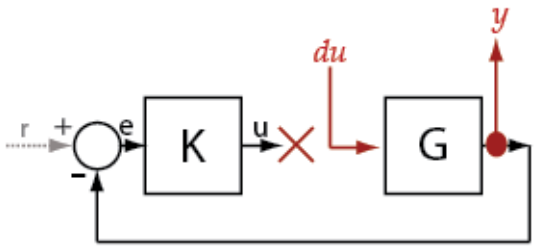
```
addPoint(sllin,{'u','y'});
```

Obtain the plant model transfer function.

```
sys = getIOTransfer(sllin,'u','y','u');
tf(sys)
```

```
ans =

  From input "u" to output "y":
    1
  -----
  s + 5

Continuous-time transfer function.
```

The second input argument specifies `u` as the input, while the fourth input argument specifies `u` as a temporary loop opening.



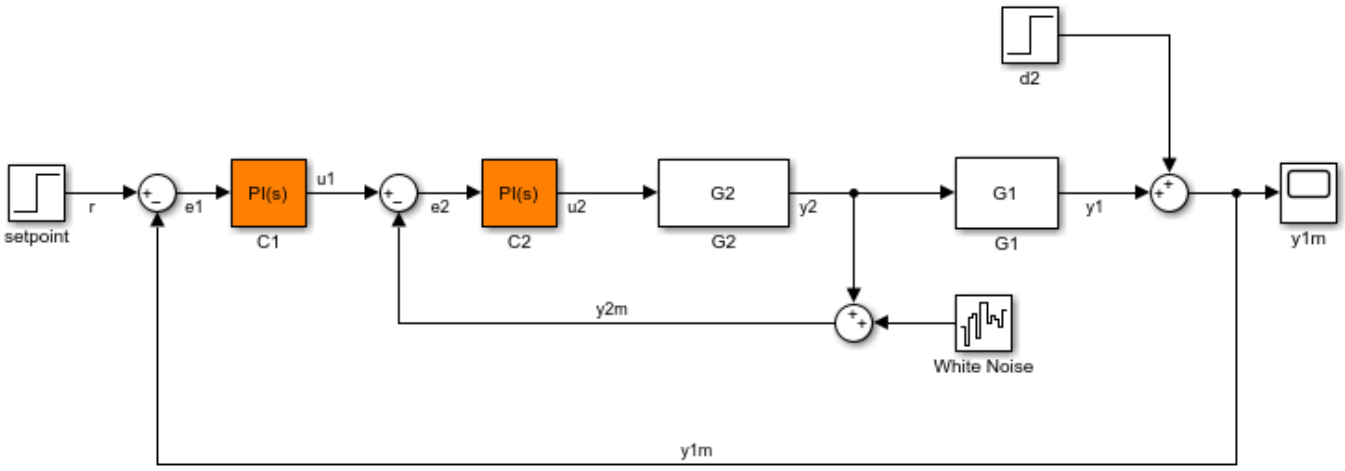
`sys` is the transfer function from `du` to `y`, which is equal to G.

Obtain Open-Loop Response Transfer Function for Specific Parameter Combination

Suppose you batch linearize the `scdcascade` model for multiple transfer functions. For most linearizations, you vary the proportional (`Kp2`) and integral gain (`Ki2`) of the `C2` controller in the 10% range. For this example, calculate the open-loop response transfer function for the inner loop, from `e2` to `y2`, for the maximum value of `Kp2` and `Ki2`.

Open the `scdcascade` model.

```
mdl = 'scdcascade';
open_system(mdl)
```



Create an `slLinearizer` interface for the model.

```
sllin = slLinearizer(mdl);
```

Vary the proportional (`Kp2`) and integral gain (`Ki2`) of the `C2` controller in the 10% range.

```
Kp2_range = linspace(0.9*Kp2,1.1*Kp2,3);
Ki2_range = linspace(0.9*Ki2,1.1*Ki2,5);

[Kp2_grid,Ki2_grid] = ndgrid(Kp2_range,Ki2_range);

params(1).Name = 'Kp2';
params(1).Value = Kp2_grid;

params(2).Name = 'Ki2';
params(2).Value = Ki2_grid;

sllin.Parameters = params;
```

To calculate the open-loop transfer function for the inner loop, use `e2` and `y2` as analysis points. To eliminate the effects of the outer loop, break the loop at `e2`. Add `e2` and `y2` to `sllin` as analysis points.

```
addPoint(sllin,{'e2','y2'})
```

Determine the index for the maximum values of `Ki2` and `Kp2`.

```
mdl_index = params(1).Value == max(Kp2_range) & params(2).Value == max(Ki2_range);
```

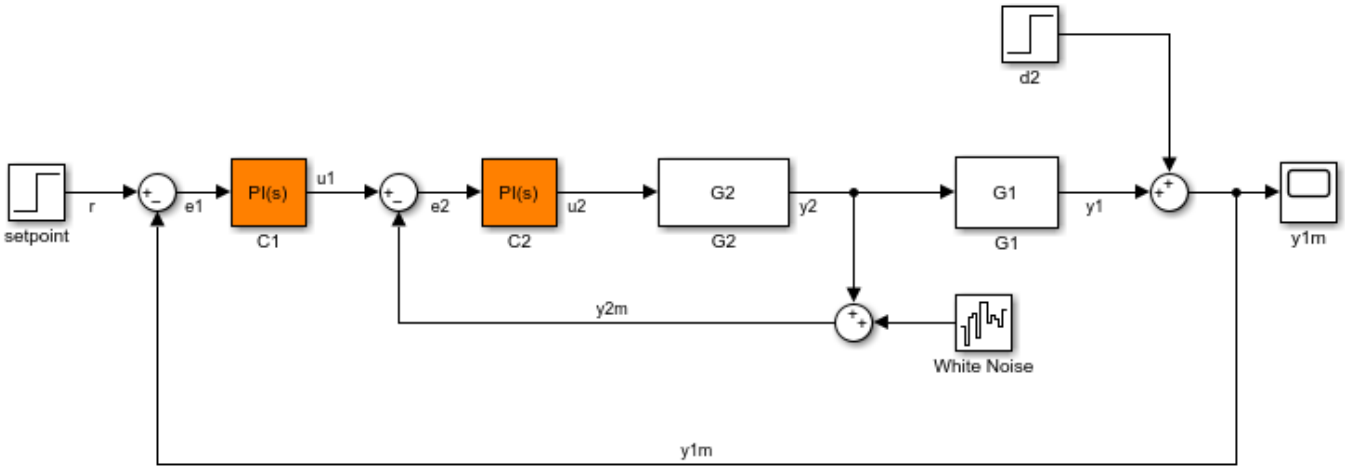
Obtain the open-loop transfer function from `e2` to `y2`.

```
sys = getIOTransfer(sllin,'e2','y2','e2',mdl_index);
```

Obtain Offsets from Input/Output Transfer Function

Open Simulink model.

```
mdl = 'scdcascade';
open_system(mdl)
```



Create a linearization option set, and set the `StoreOffsets` option.

```
opt = linearizeOptions('StoreOffsets',true);
```

Create `slLinearizer` interface.

```
sllin = slLinearizer(mdl,opt);
```

Add analysis points to calculate the closed-loop transfer function.

```
addPoint(sllin,{'r','y1m'});
```

Calculate the input/output transfer function, and obtain the corresponding linearization offsets.

```
[sys,info] = getIOTransfer(sllin,'r','y1m');
```

View offsets.

```
info.Offsets
```

```
ans =

  struct with fields:

            x: [6x1 double]
           dx: [6x1 double]
            u: 1
            y: 0
    StateName: {6x1 cell}
    InputName: {'r'}
   OutputName: {'y1m'}
```

```
Ts: 0
```

Input Arguments

s — Interface to Simulink model

`slLinearizer` interface | `slTuner` interface

Interface to a Simulink model, specified as either an `slLinearizer` interface or an `slTuner` interface.

in — Input analysis point signal name

character vector | string | cell array of character vectors | string array

Input analysis point on page 18-257 signal name, specified as:

- Character vector or string — Analysis point signal name.

  To determine the signal name associated with an analysis point, type `s`. The software displays the contents of `s` in the MATLAB command window, including the analysis point signal names, block names, and port numbers. Suppose that an analysis point does not have a signal name, but only a block name and port number. You can specify `in` as the block name. To use a point not in the list of analysis points for `s`, first add the point using `addPoint`.

  You can specify `in` as a uniquely matching portion of the full signal name or block name. Suppose that the full signal name of an analysis point is `'LoadTorque'`. You can specify `in` as `'Torque'` as long as `'Torque'` is not a portion of the signal name for any other analysis point of `s`.

  For example, `in = 'y1m'`.
- Cell array of character vectors or string array — Specifies multiple analysis point names. For example, `in = {'y1m','y2m'}`.

out — Output analysis point signal name

character vector | string | cell array of character vectors | string array

Output analysis point on page 18-257 signal name, specified as:

- Character vector or string — Analysis point signal name.

  To determine the signal name associated with an analysis point, type `s`. The software displays the contents of `s` in the MATLAB command window, including the analysis point signal names, block names, and port numbers. Suppose that an analysis point does not have a signal name, but only a block name and port number. You can specify `out` as the block name. To use a point not in the list of analysis points for `s`, first add the point using `addPoint`.

  You can specify `out` as a uniquely matching portion of the full signal name or block name. Suppose that the full signal name of an analysis point is `'LoadTorque'`. You can specify `out` as `'Torque'` as long as `'Torque'` is not a portion of the signal name for any other analysis point of `s`.

  For example, `out = 'y1m'`.
- Cell array of character vectors or string array — Specifies multiple analysis point names. For example, `out = {'y1m','y2m'}`.

`temp_opening` — Temporary opening signal name

character vector | string | cell array of character vectors | string array

Temporary opening signal name, specified as:

- Character vector or string — Analysis point signal name.

  `temp_opening` must specify an analysis point that is in the list of analysis points for `s`. To determine the signal name associated with an analysis point, type `s`. The software displays the contents of `s` in the MATLAB command window, including the analysis point signal names, block names, and port numbers. Suppose that an analysis point does not have a signal name, but only a block name and port number. You can specify `temp_opening` as the block name. To use a point not in the list of analysis points for `s`, first add the point using `addPoint`.

  You can specify `temp_opening` as a uniquely matching portion of the full signal name or block name. Suppose that the full signal name of an analysis point is `'LoadTorque'`. You can specify `temp_opening` as `'Torque'` as long as `'Torque'` is not a portion of the signal name for any other analysis point of `s`.

  For example, `temp_opening = 'y1m'`.
- Cell array of character vectors or string array — Specifies multiple analysis point names. For example, `temp_opening = {'y1m','y2m'}`.

`ios` — Linearization I/Os

linearization I/O object

Linearization I/Os, created using `linio`, specified as a linearization I/O object.

`ios` must specify signals that are in the list of analysis points for `s`. To view the list of analysis points, type `s`. To use a point that is not in the list of analysis points for `s`, you must first add the point to the list using `addPoint`.

For example:

```
ios(1) = linio('scdcascade/setpoint',1,'input');
ios(2) = linio('scdcascade/Sum',1,'output');
```

Here, `ios(1)` specifies an input, and `ios(2)` specifies an output.

`mdl_index` — Index for linearizations of interest

array of logical values | vector of positive integers

Index for linearizations of interest, specified as:

- Array of logical values — Logical array index of linearizations of interest. Suppose that you vary two parameters, `par1` and `par2`, and want to extract the linearization for the combination of `par1 > 0.5` and `par2 <= 5`. Use:

  ```
  params = s.Parameters;
  mdl_index = params(1).Value>0.5 & params(2).Value <= 5;
  ```

  The expression `params(1).Value>0.5 & params(2).Value<5` uses logical indexing and returns a logical array. This logical array is the same size as `params(1).Value` and `params(2).Value`. Each entry contains the logical evaluation of the expression for corresponding entries in `params(1).Value` and `params(2).Value`.

- Vector of positive integers — Linear index of linearizations of interest. Suppose that you vary two parameters, `par1` and `par2`, and want to extract the linearization for the combination of `par1 > 0.5` and `par2 <= 5`. Use:

  ```
  params = s.Parameters;
  mdl_index = find(params(1).Value>0.5 & params(2).Value <= 5);
  ```

  The expression `params(1).Value>0.5 & params(2).Value<5` returns a logical array. `find` returns the linear index of every true entry in the logical array

Output Arguments

`linsys` — Transfer function for specified I/Os

state-space model

Transfer function for specified I/Os, returned as described in the following:

- If you did not configure `s.Parameters` and `s.OperatingPoints`, the software calculates `linsys` using the default model parameter values. The software uses the model initial conditions as the linearization operating point. `linsys` is returned as a state-space model.
- If you configured `s.Parameters` only, the software computes a linearization for each parameter grid point. `linsys` is returned as a state-space model array of the same size as the parameter grid.
- If you configured `s.OperatingPoints` only, the software computes a linearization for each specified operating point. `linsys` is returned as a state-space model array of the same size as `s.OperatingPoints`.
- If you configured `s.Parameters` and specified `s.OperatingPoints` as a single operating point, the software computes a linearization for each parameter grid point. The software uses the specified operating point as the linearization operating point. `linsys` is returned as a state-space model array of the same size as the parameter grid.
- If you configured `s.Parameters` and specified `s.OperatingPoints` as multiple operating point objects, the software computes a linearization for each parameter grid point. The software requires that `s.OperatingPoints` is the same size as the parameter grid specified by `s.Parameters`. The software computes each linearization using corresponding operating points and parameter grid points. `linsys` is returned as a state-space model array of the same size as the parameter grid.
- If you configured `s.Parameters` and specified `s.OperatingPoints` as multiple simulation snapshot times, the software simulates and linearizes the model for each snapshot time and parameter grid point combination. Suppose that you specify a parameter grid of size `p` and `N` snapshot times. `linsys` is returned as a state-space model array of size `N`-by-`p`.

For most models, `linsys` is returned as an `ss` object or an array of `ss` objects. However, if your model contains one of the following blocks in the linearization path defined by `in` and `out`, then `linsys` returns the specified type of state-space model.

Block	`linsys` Type
Block with a substitution specified as a `genss` object or tunable model object	`genss`
Block with a substitution specified as an uncertain model, such as `uss`	`uss`

Block	linsys Type
Sparse Second Order block	mechss
Descriptor State-Space block configured to linearize to a sparse model	sparss

info — Linearization information
structure

Linearization information, returned as a structure with the following fields:

Offsets — Linearization offsets
[] (default) | structure | structure array

Linearization offsets, returned as [] if s.Options.StoreOffsets is false. Otherwise, Offsets is returned as one of the following:

- If linsys is a single state-space model, then Offsets is a structure.
- If linsys is an array of state-space models, then Offsets is a structure array with the same dimensions as linsys.

Each offset structure has the following fields:

Field	Description
x	State offsets used for linearization, returned as a column vector of length n_x, where n_x is the number of states in linsys.
y	Output offsets used for linearization, returned as a column vector of length n_y, where n_y is the number of outputs in linsys.
u	Input offsets used for linearization, returned as a column vector of length n_u, where n_u is the number of inputs in linsys.
dx	Derivative offsets for continuous time systems or updated state values for discrete-time systems, returned as a column vector of length n_x.
StateName	State names, returned as a cell array that contains n_x elements that match the names in linsys.StateName.
InputName	Input names, returned as a cell array that contains n_u elements that match the names in linsys.InputName.
OutputName	Output names, returned as a cell array that contains n_y elements that match the names in linsys.OutputName.
Ts	Sample time of the linearized system, returned as a scalar that matches the sample time in linsys.Ts. For continuous-time systems, Ts is 0.

If Offsets is a structure array, you can configure an LPV System block using the offsets. To do so, first convert them to the required format using getOffsetsForLPV. For an example, see “Approximate Nonlinear Behavior Using Array of LTI Systems” on page 3-70.

Advisor — Linearization diagnostic information
[] (default) | LinearizationAdvisor object | array of LinearizationAdvisor objects

Linearization diagnostic information, returned as [] if s.Options.StoreAdvisor is false. Otherwise, Advisor is returned as one of the following:

- If `linsys` is a single state-space model, `Advisor` is a `LinearizationAdvisor` object.
- If `linsys` is an array of state-space models, `Advisor` is an array of `LinearizationAdvisor` objects with the same dimensions as `linsys`.

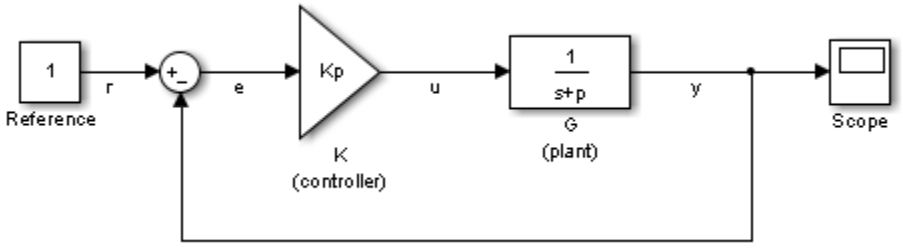
`LinearizationAdvisor` objects store linearization diagnostic information for individual linearized blocks. For an example of troubleshooting linearization results using a `LinearizationAdvisor` object, see “Troubleshoot Linearization Results at Command Line” on page 4-28.

More About

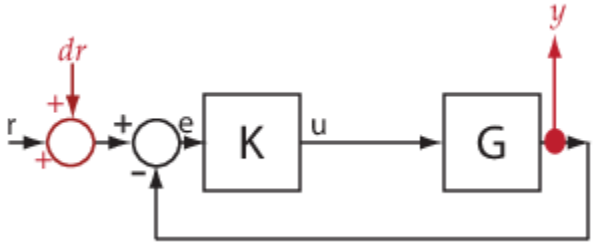
Transfer Functions

A transfer function is an LTI system response at a linearization output point to a linearization input. You perform linear analysis on transfer functions to understand the stability, time-domain characteristics, or frequency-domain characteristics of a system.

You can calculate multiple transfer functions for a given block diagram. Consider the `ex_scd_simple_fdbk` model:



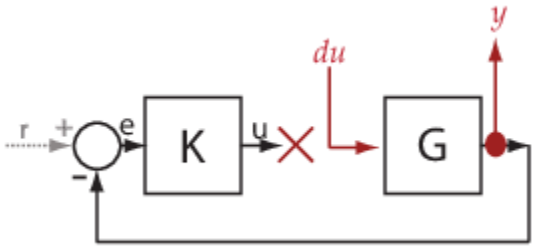
You can calculate the transfer function from the reference input signal to the plant output signal. The reference input (also referred to as setpoint), `r`, originates at the Reference block, and the plant output, `y`, originates at the G block. This transfer function is also called the overall closed-loop transfer function. To calculate this transfer function, the software adds a linearization input at `r`, `dr`, and a linearization output at `y`.



The software calculates the overall closed-loop transfer function as the transfer function from `dr` to `y`, which is equal to $(I+GK)^{-1}GK$.

Observe that the transfer function from `r` to `y` is equal to the transfer function from `dr` to `y`.

You can calculate the plant transfer function from the plant input, `u`, to the plant output, `y`. To isolate the plant dynamics from the effects of the feedback loop, introduce a loop break (or opening) at `y`, `e`, or, as shown, at `u`.



The software breaks the loop and adds a linearization input, `du`, at `u`, and a linearization output at `y`. The plant transfer function is equal to the transfer function from `du` to `y`, which is G.

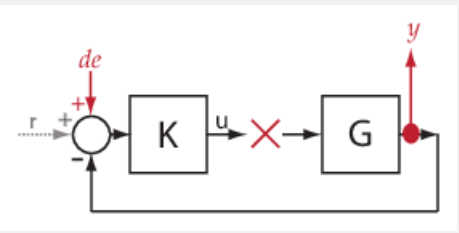
Similarly, to obtain the controller transfer function, calculate the transfer function from the controller input, `e`, to the controller output, `u`. Break the feedback loop at `y`, `e`, or `u`.

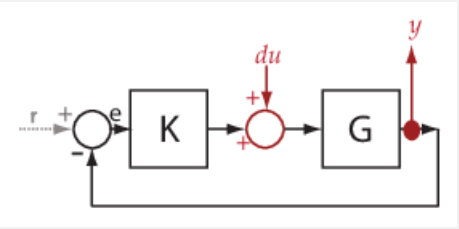
You can use `getIOTransfer` to obtain various open-loop and closed-loop transfer functions. To configure the transfer function, specify analysis points on page 18-257 as inputs, outputs, and openings (temporary or permanent on page 18-258), in any combination. The software treats each combination uniquely. Consider the following code that shows some different ways that you can use the analysis point, `u`, to obtain a transfer function:

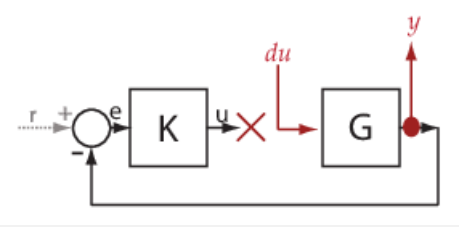
```
sllin = slLinearizer('ex_scd_simple_fdbk')

addPoint(sllin,{'u','e','y'})

T0 = getIOTransfer(sllin,'e','y','u');
T1 = getIOTransfer(sllin,'u','y');
T2 = getIOTransfer(sllin,'u','y','u');
T3 = getIOTransfer(sllin,'y','u');
T4 = getIOTransfer(sllin,'y','u','u');
T5 = getIOTransfer(sllin,'u','u');
T6 = getIOTransfer(sllin,'u','u','u');
```

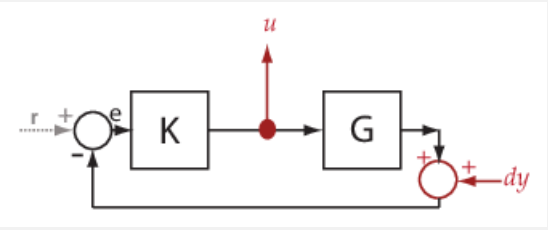
In `T0`, `u` specifies a loop break. In `T1`, `u` specifies only an input, whereas in `T2`, `u` specifies an input and an opening, also referred to as an open-loop input. In `T3`, `u` specifies only an output, whereas in `T4`, `u` specifies an output and an opening, also referred to as an open-loop output. In `T5`, `u` specifies an input and an output, also referred to as a complementary sensitivity point. In `T6`, `u` specifies an input, an output, and an opening, also referred to as a loop transfer point. The table describes how `getIOTransfer` treats the analysis points, with an emphasis on the different uses of `u`.

u Specifies...	How getIOTransfer Treats Analysis Points	Transfer Function

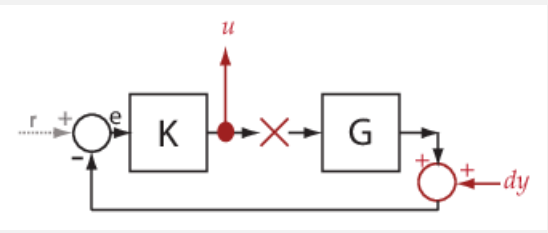
Loop break
 	
Example code:
 	
`T0 = getIOTransfer(...`
`sllin,'e','y','u')`
	
	The software stops the signal flow at u, adds a linearization input, de, at e, and a linearization output at y. Since there is not path from de to e, the transfer function is zero.
		$y = G0$
		$\rightarrow y = 0$
		$\therefore T_0 = 0$

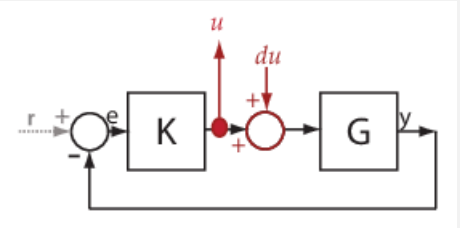
Input
 	
Example code:
 	
`T1 = getIOTransfer(...`
`sllin,'u','y')`
	
	The software adds a linearization input, du, at u, and a linearization output at y.
		$y = G(du - Ky)$
		$\rightarrow y = Gdu - GKy$
		$\rightarrow (I + GK)y = Gdu$
		$\rightarrow y = (I + GK)^{-1}Gdu$
		$\therefore T_1 = (I + GK)^{-1}G$

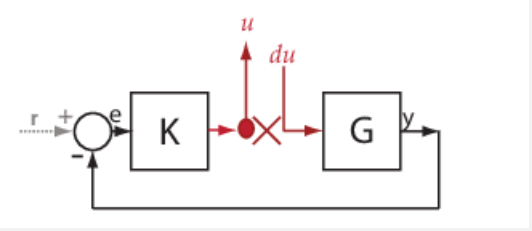
Open-loop input
 	
Example code:
 	
`T2 = getIOTransfer(...`
`sllin,'u','y','u')`
	
	The software breaks the signal flow and adds a linearization input, du, at u, and a linearization output at y.
		$y = G(du + 0)$
		$\rightarrow y = Gdu$
		$\therefore T_2 = G$

Output
 	
Example code:
 	
`T3 = getIOTransfer(...`
`sllin,'y','u')`
	
	The software adds a linearization input, dy, at y and a linearization output at u.
		$u = -K(dy + Gu)$
		$\rightarrow u = -Kdy - KGu$
		$\rightarrow (I + KG)u = -Kdy$
		$\rightarrow u = -(I + KG)^{-1}Kdy$
		$\therefore T_3 = -(I + KG)^{-1}K$

u Specifies...	How `getIOTransfer` Treats Analysis Points	Transfer Function

Open-loop output
 	
Example code:
 	
`T4 = getIOTransfer(...`
`sllin,'y','u','u')`
	
	
	The software adds a linearization input, `dy`, at `y` and adds a linearization output and breaks the signal flow at `u`.
		$u = -K(dy + G0)$
		$\rightarrow u = -Kdy$
		$\therefore T_4 = -K$

Complementary sensitivity point
 	
Example code:
 	
`T5 = getIOTransfer(...`
`sllin,'u','u')`
 	
Tip You also can obtain the complementary sensitivity function using `getCompSensitivity`.
	
	
	The software adds a linearization output and a linearization input, `du`, at `u`.
		$u = -KG(du + u)$
		$\rightarrow u = -KGdu - KGu$
		$\rightarrow (I + KG)u = -KGdu$
		$\rightarrow u = -(I + KG)^{-1}KGdu$
		$\therefore T_5 = -(I + KG)^{-1}KG$

Loop transfer function point
 	
Example code:
 	
`T6 = getIOTransfer(...`
`sllin,'u','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`.
		$u = -KG(du + 0)$
		$\rightarrow u = -KGdu$
		$\therefore T_6 = -KG$

The software does not modify the Simulink model when it computes the transfer function.

Analysis Points

Analysis points, used by the `slLinearizer` and `slTuner` interfaces, identify locations within a model that are relevant for linear analysis and control system tuning. You use analysis points as inputs to the linearization commands, such as `getIOTransfer`, `getLoopTransfer`, `getSensitivity`, and `getCompSensitivity`. As inputs to the linearization commands, analysis points can specify any open-loop or closed-loop transfer function in a model. You can also use analysis points to specify design requirements when tuning control systems using commands such as `systune`.

Location refers to a specific block output port within a model or to a bus element in such an output port. For convenience, you can use the name of the signal that originates from this port to refer to an analysis point.

You can add analysis points to an `slLinearizer` or `slTuner` interface, `s`, when you create the interface. For example:

```
s = slLinearizer('scdcascade',{'u1','y1'});
```

Alternatively, you can use the `addPoint` command.

To view all the analysis points of `s`, type `s` at the command prompt to display the interface contents. For each analysis point of `s`, the display includes the block name and port number and the name of the signal that originates at this point. You can also programmatically obtain a list of all the analysis points using `getPoints`.

For more information about how you can use analysis points, see “Mark Signals of Interest for Control System Analysis and Design” on page 2-38 and “Mark Signals of Interest for Batch Linearization” on page 3-9.

Permanent Openings

Permanent openings, used by the `slLinearizer` and `slTuner` interfaces, identify locations within a model where the software breaks the signal flow. The software enforces these openings for linearization and tuning. Use permanent openings to isolate a specific model component. Suppose that you have a large-scale model capturing aircraft dynamics and you want to perform linear analysis on the airframe only. You can use permanent openings to exclude all other components of the model. Another example is when you have cascaded loops within your model and you want to analyze a specific loop.

Location refers to a specific block output port within a model. For convenience, you can use the name of the signal that originates from this port to refer to an opening.

You can add permanent openings to an `slLinearizer` or `slTuner` interface, `s`, when you create the interface or by using the `addOpening` command. To remove a location from the list of permanent openings, use the `removeOpening` command.

To view all the openings of `s`, type `s` at the command prompt to display the interface contents. For each permanent opening of `s`, the display includes the block name and port number and the name of the signal that originates at this location. You can also programmatically obtain a list of all the permanent loop openings using `getOpenings`.

See Also

`slLinearizer` | `slTuner` | `addPoint` | `addOpening` | `getLoopTransfer` | `getSensitivity` | `getCompSensitivity`

Topics

“How the Software Treats Loop Openings” on page 2-31
“Vary Parameter Values and Obtain Multiple Transfer Functions” on page 3-21
“Vary Operating Points and Obtain Multiple Transfer Functions Using slLinearizer Interface” on page 3-28
“Analyze Command-Line Batch Linearization Results Using Response Plots” on page 3-33

Introduced in R2013b

getLoopTransfer

Open-loop transfer function at specified point using `slLinearizer` or `slTuner` interface

Syntax

```
linsys = getLoopTransfer(s,pt)
linsys = getLoopTransfer(s,pt,sign)

linsys = getLoopTransfer(s,pt,temp_opening)
linsys = getLoopTransfer(s,pt,temp_opening,sign)

linsys = getLoopTransfer( ___ ,mdl_index)

[linsys,info] = getLoopTransfer( ___ )
```

Description

`linsys = getLoopTransfer(s,pt)` returns the point-to-point open-loop transfer function on page 18-271 at the specified analysis point for the model associated with the `slLinearizer` or `slTuner` interface, `s`.

The software enforces all the permanent loop openings on page 18-273 specified for `s` when it calculates `linsys`. If you configured either `s.Parameters`, or `s.OperatingPoints`, or both, `getLoopTransfer` performs multiple linearizations and returns an array of loop transfer functions.

`linsys = getLoopTransfer(s,pt,sign)` specifies the feedback sign for computing the open-loop response. By default, `linsys` 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 `linsys`. 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,linsys,sign)`.

`linsys = getLoopTransfer(s,pt,temp_opening)` considers additional, temporary, openings at the point specified by `temp_opening`. Use an opening, for example, to calculate the loop transfer function of an inner loop, measured at the plant input, with the outer loop open.

`linsys = getLoopTransfer(s,pt,temp_opening,sign)` specifies temporary openings and the feedback sign.

`linsys = 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.

`[linsys,info] = getLoopTransfer( ___ )` returns additional linearization information.

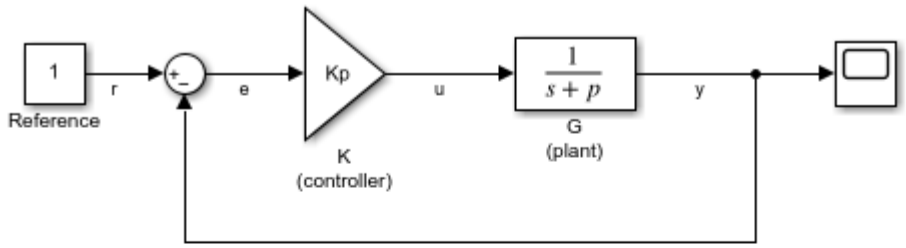
Examples

Obtain Loop Transfer Function at Analysis Point

Obtain the loop transfer function, calculated at `e`, for the `ex_scd_simple_fdbk` model.

Open the `ex_scd_simple_fdbk` model.

```
mdl = 'ex_scd_simple_fdbk';
open_system(mdl);
```



Copyright 2015-2021 The MathWorks, Inc.

In this model:

$$K(s) = K_p = 3$$
$$G(s) = \frac{1}{s+5}.$$

Create an `slLinearizer` interface for the model.

```
sllin = slLinearizer(mdl);
```

To obtain the loop transfer function at `e`, add this point to `sllin` as an analysis point.

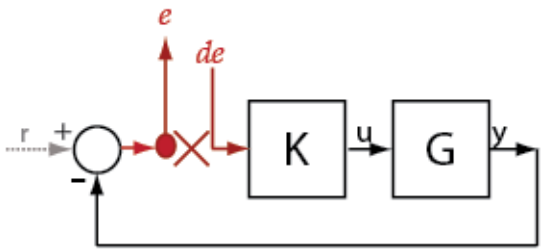
```
addPoint(sllin,'e');
```

Obtain the loop transfer function at `e`.

```
sys = getLoopTransfer(sllin,'e');
tf(sys)
```

```
ans =

  From input "e" to output "e":
   -3
  -----
  s + 5

Continuous-time transfer function.
```

The software adds a linearization output, breaks the loop, and adds a linearization input, `de`, at `e`.



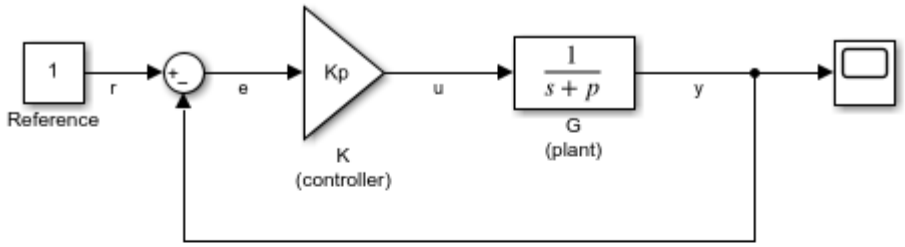
`sys` is the transfer function from `de` to `e`. Because the software assumes positive-feedback, it returns `sys` as $-GK$.

Obtain Negative-Feedback Loop Transfer Function at Analysis Point

Obtain the negative-feedback loop transfer function, calculated at `e`, for the `ex_scd_simple_fdbk` model.

Open the `ex_scd_simple_fdbk` model.

```
mdl = 'ex_scd_simple_fdbk';
open_system(mdl);
```



Copyright 2015-2021 The MathWorks, Inc.

In this model:

$K(s) = K_p = 3$
$G(s) = \frac{1}{s+5}$.

Create an `slLinearizer` interface for the model.

```
sllin = slLinearizer(mdl);
```

To obtain the loop transfer function at `e`, add this point to `sllin` as an analysis point.

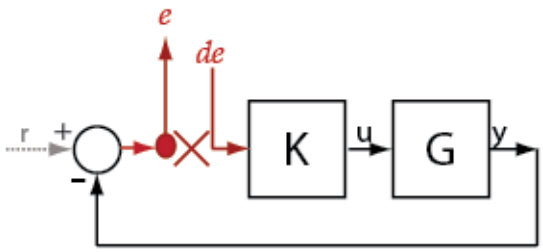
```
addPoint(sllin,'e');
```

Obtain the loop transfer function at `e`.

```
sys = getLoopTransfer(sllin,'e',-1);
tf(sys)
```

```
ans =

  From input "e" to output "e":
    3
  -----
  s + 5

Continuous-time transfer function.
```

The software adds a linearization output, breaks the loop, and adds a linearization input, `de`, at `e`.



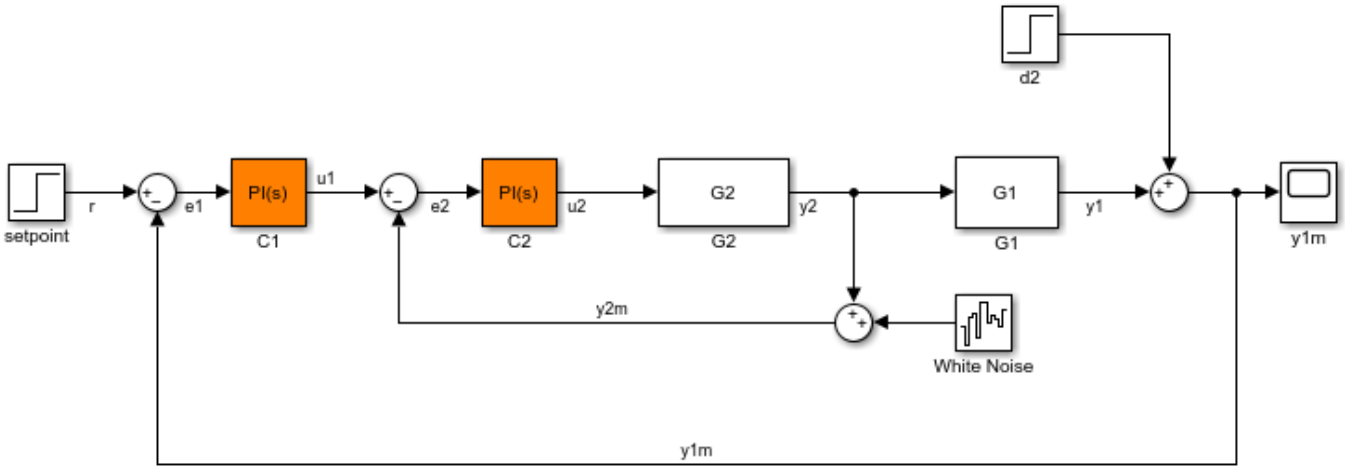
`sys` is the transfer function from `de` to `e`. Because the third input argument indicates negative-feedback, the software returns `sys` as GK.

Specify Temporary Loop Opening for Loop Transfer Function Calculation

Obtain the loop transfer function for the inner loop, calculated at `e2`, for the `scdcascade` model.

Open the `scdcascade` model.

```
mdl = 'scdcascade';
open_system(mdl)
```



Create an `slLinearizer` interface for the model.

```
sllin = slLinearizer(mdl);
```

To calculate the loop transfer function for the inner loop, use the `e2` signal as the analysis point. To eliminate the effects of the outer loop, break the outer loop at `y1m`. Add these points to `sllin`.

```
addPoint(sllin,{'e2','y1m'});
```

Obtain the inner-loop loop transfer function at `e2`.

```
sys = getLoopTransfer(sllin,'e2','y1m');
```

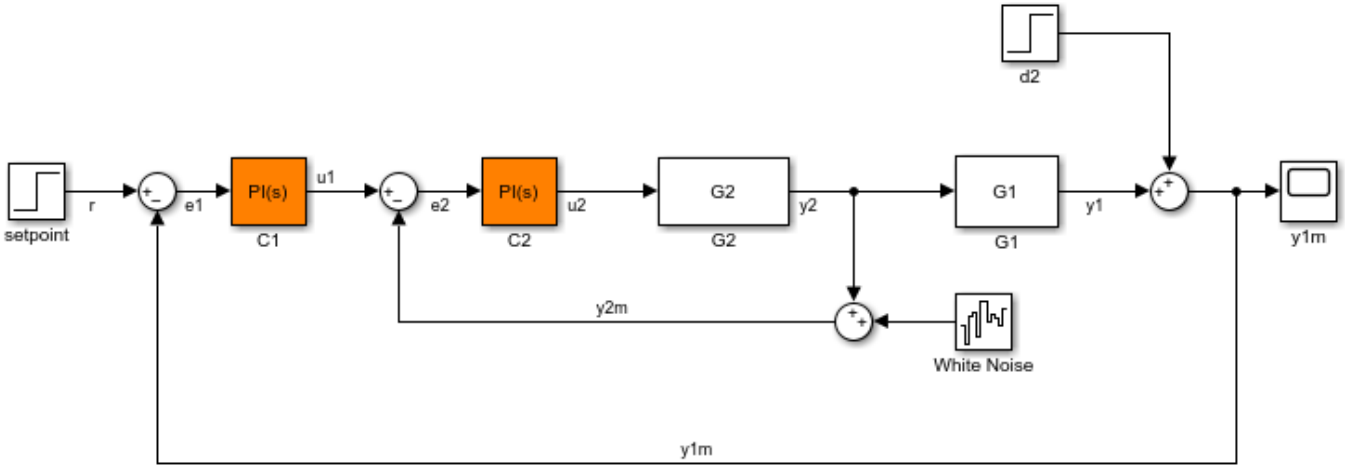
Here, `'y1m'`, the third input argument, specifies a temporary loop opening. The software assumes positive-feedback when it calculates `sys`.

Obtain Loop Transfer Function for Specific Parameter Combination

Suppose you batch linearize the `scdcascade` model for multiple transfer functions. For most linearizations, you vary the proportional (`Kp2`) and integral gain (`Ki2`) of the `C2` controller, in the 10% range. For this example, calculate the loop transfer function for the inner loop at `e2` for the maximum values of `Kp2` and `Ki2`.

Open the `scdcascade` model.

```
mdl = 'scdcascade';
open_system(mdl)
```



Create an `slLinearizer` interface for the model.

```
sllin = slLinearizer(mdl);
```

Vary the proportional (`Kp2`) and integral gain (`Ki2`) of the `C2` controller in the 10% range.

```
Kp2_range = linspace(0.9*Kp2,1.1*Kp2,3);
Ki2_range = linspace(0.9*Ki2,1.1*Ki2,5);

[Kp2_grid,Ki2_grid] = ndgrid(Kp2_range,Ki2_range);

params(1).Name = 'Kp2';
params(1).Value = Kp2_grid;

params(2).Name = 'Ki2';
params(2).Value = Ki2_grid;

sllin.Parameters = params;
```

To calculate the loop transfer function for the inner loop, use the `e2` signal as the analysis point. To eliminate the effects of the outer loop, break the outer loop at `y1m`. Add these points to `sllin`.

```
addPoint(sllin,{'e2','y1m'});
```

Determine the index for the maximum values of `Ki2` and `Kp2`.

```
mdl_index = params(1).Value == max(Kp2_range) & params(2).Value == max(Ki2_range);
```

Obtain the inner-loop loop transfer function at `e2`, with the outer loop open.

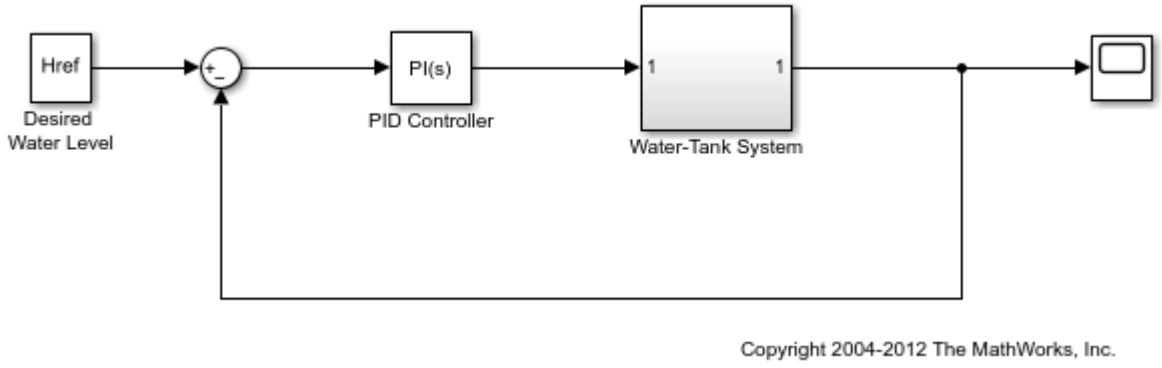
```
sys = getLoopTransfer(sllin,'e2','y1m',-1,mdl_index);
```

The fourth input argument specifies negative-feedback for the loop transfer calculation.

Obtain Offsets from Loop Transfer Function

Open Simulink model.

```
mdl = 'watertank';
open_system(mdl)
```



Create a linearization option set, and set the `StoreOffsets` option.

```
opt = linearizeOptions('StoreOffsets',true);
```

Create `slLinearizer` interface.

```
sllin = slLinearizer(mdl,opt);
```

Add an analysis point at the tank output port.

```
addPoint(sllin,'watertank/Water-Tank System');
```

Calculate the loop transfer function at the analysis point, and obtain the corresponding linearization offsets.

```
[sys,info] = getLoopTransfer(sllin,'watertank/Water-Tank System');
```

View offsets.

```
info.Offsets
```

```
ans =

  struct with fields:
```

```
          x: [2x1 double]
         dx: [2x1 double]
          u: 1
          y: 1
  StateName: {2x1 cell}
  InputName: {'watertank/Water-Tank System'}
 OutputName: {'watertank/Water-Tank System'}
         Ts: 0
```

Input Arguments

`s` — Interface to Simulink model

`slLinearizer` interface | `slTuner` interface

Interface to a Simulink model, specified as either an `slLinearizer` interface or an `slTuner` interface.

`pt` — Analysis point signal name

character vector | string | cell array of character vectors | string array

Analysis point on page 18-272 signal name, specified as:

- Character vector or string — Analysis point signal name.

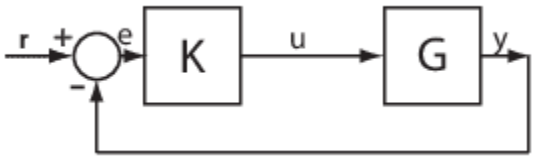
  To determine the signal name associated with an analysis point, type `s`. The software displays the contents of `s` in the MATLAB command window, including the analysis point signal names, block names, and port numbers. Suppose that an analysis point does not have a signal name, but only a block name and port number. You can specify `pt` as the block name. To use a point not in the list of analysis points for `s`, first add the point using `addPoint`.

  You can specify `pt` as a uniquely matching portion of the full signal name or block name. Suppose that the full signal name of an analysis point is `'LoadTorque'`. You can specify `pt` as `'Torque'` as long as `'Torque'` is not a portion of the signal name for any other analysis point of `s`.

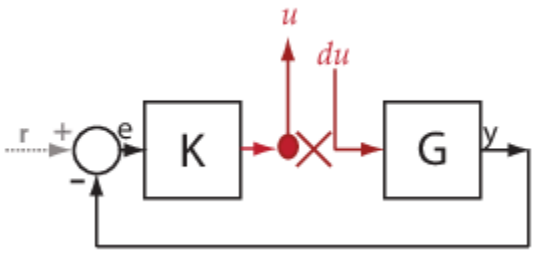
  For example, `pt = 'y1m'`.

- Cell array of character vectors or string array — Specifies multiple analysis point names. For example, `pt = {'y1m','y2m'}`.

To calculate `linsys`, the software adds a linearization output, followed by a loop break, and then a linearization input at `pt`. Consider the following model:

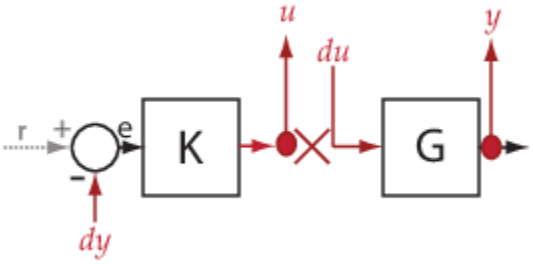


Specify `pt` as `'u'`.



The software computes `linsys` 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 `linsys` as a MIMO transfer function with a transfer function from each linearization input to each linearization output.

`sign` — Feedback sign
+1 (default) | -1

Feedback sign, specified as one of the following values:

- `+1` (default) — `getLoopTransfer` returns the positive-feedback open-loop transfer function.
- `-1` — `getLoopTransfer` returns the negative-feedback open-loop transfer function. The negative-feedback transfer function is `-1` times the positive-feedback transfer function.

`temp_opening` — Temporary opening signal name
character vector | string | cell array of character vectors | string array

Temporary opening signal name, specified as:

- Character vector or string — Analysis point signal name.

  `temp_opening` must specify an analysis point that is in the list of analysis points for `s`. To determine the signal name associated with an analysis point, type `s`. The software displays the contents of `s` in the MATLAB command window, including the analysis point signal names, block names, and port numbers. Suppose that an analysis point does not have a signal name, but only a block name and port number. You can specify `temp_opening` as the block name. To use a point not in the list of analysis points for `s`, first add the point using `addPoint`.

  You can specify `temp_opening` as a uniquely matching portion of the full signal name or block name. Suppose that the full signal name of an analysis point is `'LoadTorque'`. You can specify

`temp_opening` as `'Torque'` as long as `'Torque'` is not a portion of the signal name for any other analysis point of `s`.

For example, `temp_opening = 'y1m'`.

- Cell array of character vectors or string array — Specifies multiple analysis point names. For example, `temp_opening = {'y1m','y2m'}`.

`mdl_index` — Index for linearizations of interest
array of logical values | vector of positive integers

Index for linearizations of interest, specified as:

- Array of logical values — Logical array index of linearizations of interest. Suppose that you vary two parameters, `par1` and `par2`, and want to extract the linearization for the combination of `par1 > 0.5` and `par2 <= 5`. Use:

  ```
  params = s.Parameters;
  mdl_index = params(1).Value>0.5 & params(2).Value <= 5;
  ```

  The expression `params(1).Value>0.5 & params(2).Value<5` uses logical indexing and returns a logical array. This logical array is the same size as `params(1).Value` and `params(2).Value`. Each entry contains the logical evaluation of the expression for corresponding entries in `params(1).Value` and `params(2).Value`.
- Vector of positive integers — Linear index of linearizations of interest. Suppose that you vary two parameters, `par1` and `par2`, and want to extract the linearization for the combination of `par1 > 0.5` and `par2 <= 5`. Use:

  ```
  params = s.Parameters;
  mdl_index = find(params(1).Value>0.5 & params(2).Value <= 5);
  ```

  The expression `params(1).Value>0.5 & params(2).Value<5` returns a logical array. `find` returns the linear index of every true entry in the logical array

Output Arguments

`linsys` — Point-to-point open-loop transfer function
state-space object

Point-to-point open-loop transfer function, returned as described in the following:

- If you did not configure `s.Parameters` and `s.OperatingPoints`, the software calculates `linsys` using the default model parameter values. The software uses the model initial conditions as the linearization operating point. `linsys` is returned as a state-space model.
- If you configured `s.Parameters` only, the software computes a linearization for each parameter grid point. `linsys` is returned as a state-space model array of the same size as the parameter grid.
- If you configured `s.OperatingPoints` only, the software computes a linearization for each specified operating point. `linsys` is returned as a state-space model array of the same size as `s.OperatingPoints`.
- If you configured `s.Parameters` and specified `s.OperatingPoints` as a single operating point, the software computes a linearization for each parameter grid point. The software uses the specified operating point as the linearization operating point. `linsys` is returned as a state-space model array of the same size as the parameter grid.

- If you configured `s.Parameters` and specified `s.OperatingPoints` as multiple operating point objects, the software computes a linearization for each parameter grid point. The software requires that `s.OperatingPoints` is the same size as the parameter grid specified by `s.Parameters`. The software computes each linearization using corresponding operating points and parameter grid points. `linsys` is returned as a state-space model array of the same size as the parameter grid.
- If you configured `s.Parameters` and specified `s.OperatingPoints` as multiple simulation snapshot times, the software simulates and linearizes the model for each snapshot time and parameter grid point combination. Suppose that you specify a parameter grid of size `p` and `N` snapshot times. `linsys` is returned as a state-space model array of size `N`-by-`p`.

For most models, `linsys` is returned as an `ss` object or an array of `ss` objects. However, if your model contains one of the following blocks in the linearization path defined by `pt`, then `linsys` returns the specified type of state-space model.

Block	`linsys` Type
Block with a substitution specified as a `genss` object or tunable model object	`genss`
Block with a substitution specified as an uncertain model, such as `uss`	`uss`
Sparse Second Order block	`mechss`
Descriptor State-Space block configured to linearize to a sparse model	`sparss`

`info` — Linearization information
structure

Linearization information, returned as a structure with the following fields:

`Offsets` — Linearization offsets
`[]` (default) | structure | structure array

Linearization offsets, returned as `[]` if `s.Options.StoreOffsets` is `false`. Otherwise, `Offsets` is returned as one of the following:

- If `linsys` is a single state-space model, then `Offsets` is a structure.
- If `linsys` is an array of state-space models, then `Offsets` is a structure array with the same dimensions as `linsys`.

Each offset structure has the following fields:

Field	Description
`x`	State offsets used for linearization, returned as a column vector of length n_x, where n_x is the number of states in `linsys`.
`y`	Output offsets used for linearization, returned as a column vector of length n_y, where n_y is the number of outputs in `linsys`.
`u`	Input offsets used for linearization, returned as a column vector of length n_u, where n_u is the number of inputs in `linsys`.

Field	Description
`dx`	Derivative offsets for continuous time systems or updated state values for discrete-time systems, returned as a column vector of length n_x.
`StateName`	State names, returned as a cell array that contains n_x elements that match the names in `linsys.StateName`.
`InputName`	Input names, returned as a cell array that contains n_u elements that match the names in `linsys.InputName`.
`OutputName`	Output names, returned as a cell array that contains n_y elements that match the names in `linsys.OutputName`.
`Ts`	Sample time of the linearized system, returned as a scalar that matches the sample time in `linsys.Ts`. For continuous-time systems, `Ts` is `0`.

If `Offsets` is a structure array, you can configure an LPV System block using the offsets. To do so, first convert them to the required format using `getOffsetsForLPV`. For an example, see “Approximate Nonlinear Behavior Using Array of LTI Systems” on page 3-70.

`Advisor` — Linearization diagnostic information

`[]` (default) | `LinearizationAdvisor` object | array of `LinearizationAdvisor` objects

Linearization diagnostic information, returned as `[]` if `s.Options.StoreAdvisor` is `false`. Otherwise, `Advisor` is returned as one of the following:

- If `linsys` is a single state-space model, `Advisor` is a `LinearizationAdvisor` object.
- If `linsys` is an array of state-space models, `Advisor` is an array of `LinearizationAdvisor` objects with the same dimensions as `linsys`.

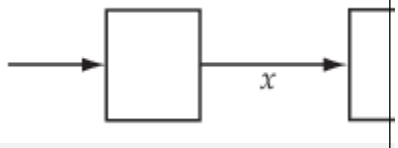
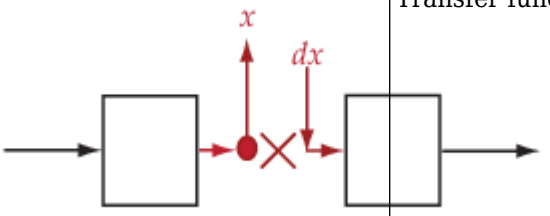
`LinearizationAdvisor` objects store linearization diagnostic information for individual linearized blocks. For an example of troubleshooting linearization results using a `LinearizationAdvisor` object, see “Troubleshoot Linearization Results at Command Line” on page 4-28.

More About

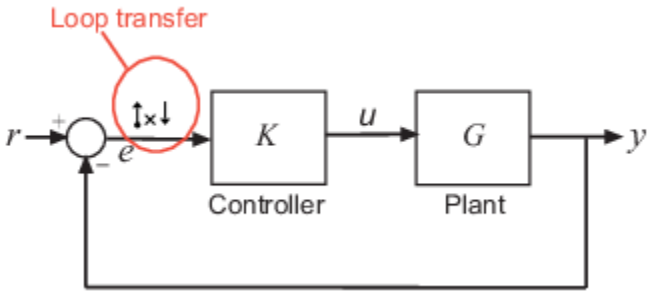
Loop Transfer Function

The loop transfer function at a point is the point-to-point open-loop transfer function from an additive disturbance at a point to a measurement at the same point.

To compute the loop transfer function at an analysis point, `x`, the software adds a linearization output, inserts a loop break, and adds a linearization input, `dx`. The software computes the transfer function from `dx` to `x`, which is equal to the loop transfer function at `x`.

Analysis Point in Simulink Model	How `getLoopTransfer` Interprets Analysis Point	Loop Transfer Function
		Transfer function from `dx` to `x`

For example, consider the following model where you compute the loop transfer function at `e`:



Here, at `e`, the software adds a linearization output, inserts a loop break, and adds a linearization input, `de`. The loop transfer function at `e`, L, is the transfer function from `de` to `e`. L is calculated as follows:

$$e = -GKde$$
$$\therefore L = -GK$$

To compute $-KG$, use `u` as the analysis point for `getLoopTransfer`.

The software does not modify the Simulink model when it computes the loop transfer function.

Analysis Points

Analysis points, used by the `slLinearizer` and `slTuner` interfaces, identify locations within a model that are relevant for linear analysis and control system tuning. You use analysis points as inputs to the linearization commands, such as `getIOTransfer`, `getLoopTransfer`, `getSensitivity`, and `getCompSensitivity`. As inputs to the linearization commands, analysis points can specify any open-loop or closed-loop transfer function in a model. You can also use analysis points to specify design requirements when tuning control systems using commands such as `systune`.

Location refers to a specific block output port within a model or to a bus element in such an output port. For convenience, you can use the name of the signal that originates from this port to refer to an analysis point.

You can add analysis points to an `slLinearizer` or `slTuner` interface, `s`, when you create the interface. For example:

```
s = slLinearizer('scdcascade',{'u1','y1'});
```

Alternatively, you can use the `addPoint` command.

To view all the analysis points of `s`, type `s` at the command prompt to display the interface contents. For each analysis point of `s`, the display includes the block name and port number and the name of the signal that originates at this point. You can also programmatically obtain a list of all the analysis points using `getPoints`.

For more information about how you can use analysis points, see “Mark Signals of Interest for Control System Analysis and Design” on page 2-38 and “Mark Signals of Interest for Batch Linearization” on page 3-9.

Permanent Openings

Permanent openings, used by the `slLinearizer` and `slTuner` interfaces, identify locations within a model where the software breaks the signal flow. The software enforces these openings for linearization and tuning. Use permanent openings to isolate a specific model component. Suppose that you have a large-scale model capturing aircraft dynamics and you want to perform linear analysis on the airframe only. You can use permanent openings to exclude all other components of the model. Another example is when you have cascaded loops within your model and you want to analyze a specific loop.

Location refers to a specific block output port within a model. For convenience, you can use the name of the signal that originates from this port to refer to an opening.

You can add permanent openings to an `slLinearizer` or `slTuner` interface, `s`, when you create the interface or by using the `addOpening` command. To remove a location from the list of permanent openings, use the `removeOpening` command.

To view all the openings of `s`, type `s` at the command prompt to display the interface contents. For each permanent opening of `s`, the display includes the block name and port number and the name of the signal that originates at this location. You can also programmatically obtain a list of all the permanent loop openings using `getOpenings`.

See Also

`slLinearizer` | `slTuner` | `addPoint` | `addOpening` | `getIOTransfer` | `getSensitivity` | `getCompSensitivity`

Topics

“How the Software Treats Loop Openings” on page 2-31
“Vary Parameter Values and Obtain Multiple Transfer Functions” on page 3-21
“Vary Operating Points and Obtain Multiple Transfer Functions Using slLinearizer Interface” on page 3-28
“Analyze Command-Line Batch Linearization Results Using Response Plots” on page 3-33

Introduced in R2013b

getOpenings

Get list of openings for `slLinearizer` or `slTuner` interface

Syntax

```
op_names = getOpenings(s)
```

Description

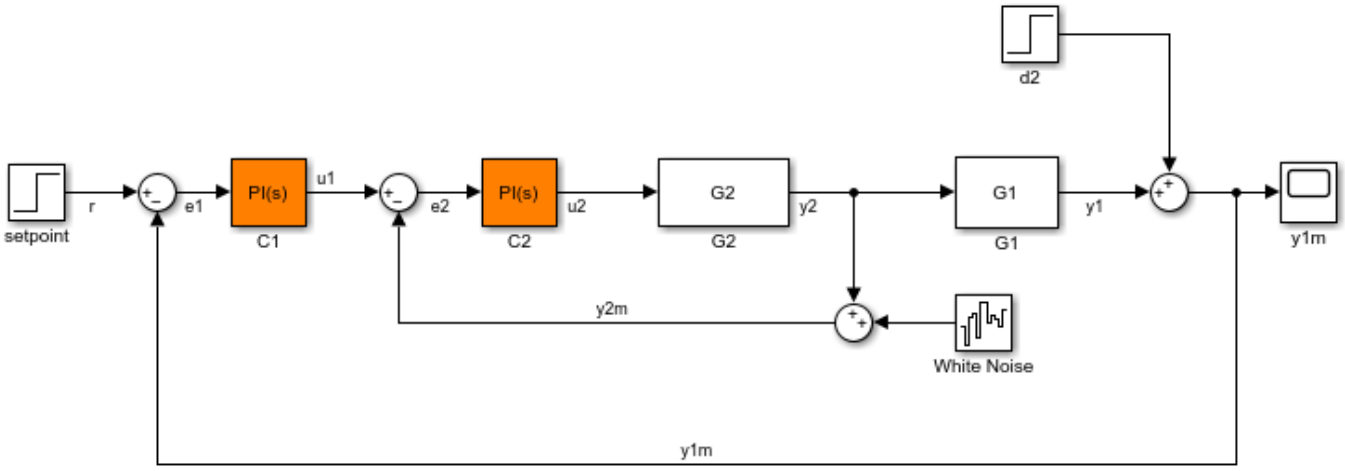
`op_names = getOpenings(s)` returns the names of the permanent openings of `s`, which can be either an `slLinearizer` interface or an `slTuner` interface.

Examples

Obtain Permanent Opening Names of slLinearizer Interface

Open the `scdcascade` model.

```
mdl = 'scdcascade';
open_system(mdl)
```



Create an `slLinearizer` interface to the model, and add some analysis points to the interface.

```
sllin = slLinearizer(mdl,{'u1','y1'});
```

Suppose you are interested in analyzing only the inner loop. To do so, add `y1m` as a permanent opening of `sllin`.

```
addOpening(sllin,'y1m');
```

In larger models, you may want to open multiple loops to isolate the system of interest.

After performing some additional steps, such as adding more points of interest and extracting transfer functions, suppose you want a list of all the openings of `sllin`.

```
op_names = getOpenings(sllin)
```

```
op_names =

  1x1 cell array

    {'scdcascade/Sum/1[y1m]'}
```

Input Arguments

s — Interface to Simulink model

`slLinearizer` interface | `slTuner` interface

Interface to a Simulink model, specified as either an `slLinearizer` interface or an `slTuner` interface.

Output Arguments

op_names — Permanent opening names

cell array of character vectors

Permanent opening names, returned as a cell array of character vectors.

Each entry of `op_names` follows the pattern, `full block path/outport number/[signal name]`.

See Also

`getIOTransfer` | `addOpening` | `removeOpening` | `slLinearizer` | `slTuner`

Introduced in R2014a

getPoints

Get list of analysis points for `slLinearizer` or `slTuner` interface

Syntax

```
pt_names = getPoints(s)
```

Description

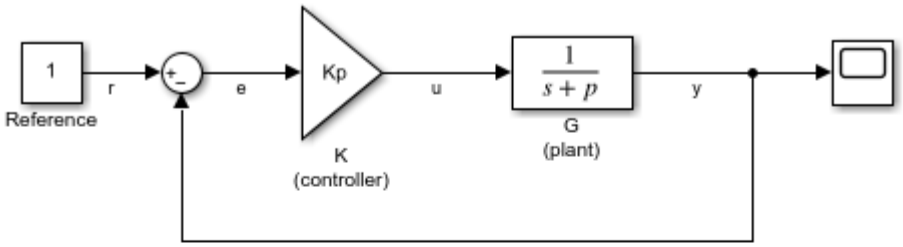
`pt_names = getPoints(s)` returns the names of the analysis points of `s`, which can be either an `slLinearizer` interface or an `slTuner` interface. Use the analysis point names to extract transfer functions using commands such as `getIOTransfer` and to specify tuning goals for an `slTuner` interface.

Examples

Obtain Analysis Point Names of slLinearizer Interface

Open Simulink model.

```
mdl = 'ex_scd_simple_fdbk';
open_system(mdl)
```



Copyright 2015-2021 The MathWorks, Inc.

Create an `slLinearizer` interface to the model, and add some analysis points to the interface.

```
sllin = slLinearizer(mdl,{'r','e','u','y'});
```

Get the names of all the analysis points associated with `sllin`.

```
pt_names = getPoints(sllin)
```

```
pt_names =

  4x1 cell array

    {'ex_scd_simple_fdbk/Reference/1[r]'     }
    {'ex_scd_simple_fdbk/Sum/1[e]'           }
    {'ex_scd_simple_fdbk/K (controller)/1[u]'}
    {'ex_scd_simple_fdbk/G (plant)/1[y]'     }
```

Input Arguments

s — Interface to Simulink model

`slLinearizer` interface | `slTuner` interface

Interface to a Simulink model, specified as either an `slLinearizer` interface or an `slTuner` interface.

Output Arguments

pt_names — Analysis point names

cell array of character vectors

Analysis point names, returned as a cell array of character vectors.

Each entry of `pt_names` follows the pattern, `full block path/outport number/[signal name]`.

See Also

`getIOTransfer` | `addPoint` | `removePoint` | `slLinearizer` | `slTuner`

Topics

“Mark Signals of Interest for Control System Analysis and Design” on page 2-38

Introduced in R2014a

getSensitivity

Sensitivity function at specified point using `slLinearizer` or `slTuner` interface

Syntax

```
linsys = getSensitivity(s,pt)
linsys = getSensitivity(s,pt,temp_opening)
linsys = getSensitivity( ___ ,mdl_index)

[linsys,info] = getSensitivity( ___ )
```

Description

`linsys = getSensitivity(s,pt)` returns the sensitivity function on page 18-287 at the specified analysis point for the model associated with the `slLinearizer` or `slTuner` interface, `s`.

The software enforces all the permanent openings on page 18-289 specified for `s` when it calculates `linsys`. If you configured either `s.Parameters`, or `s.OperatingPoints`, or both, `getSensitivity` performs multiple linearizations and returns an array of sensitivity functions.

`linsys = getSensitivity(s,pt,temp_opening)` considers additional, temporary, openings at the point specified by `temp_opening`. Use an opening, for example, to calculate the sensitivity function of an inner loop, with the outer loop open.

`linsys = 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.

`[linsys,info] = getSensitivity( ___ )` returns additional linearization information.

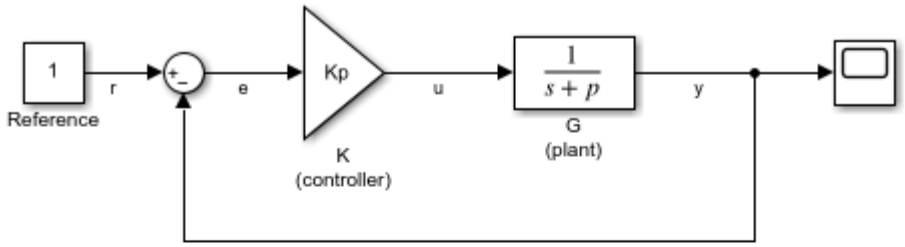
Examples

Sensitivity Function at Analysis Point

For the `ex_scd_simple_fdbk` model, obtain the sensitivity at the plant input, `u`.

Open the `ex_scd_simple_fdbk` model.

```
mdl = 'ex_scd_simple_fdbk';
open_system(mdl);
```



Copyright 2015-2021 The MathWorks, Inc.

In this model:

$$K(s) = K_p = 3$$
$$G(s) = \frac{1}{s+5}.$$

Create an `slLinearizer` interface for the model.

```
sllin = slLinearizer(mdl);
```

To obtain the sensitivity at the plant input, `u`, add `u` as an analysis point to `sllin`.

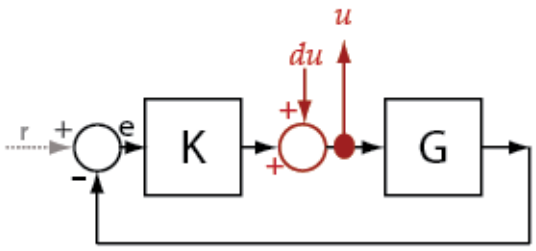
```
addPoint(sllin,'u');
```

Obtain the sensitivity at the plant input, `u`.

```
sys = getSensitivity(sllin,'u');
tf(sys)
```

```
ans =

  From input "u" to output "u":
  s + 5
  -----
  s + 8

Continuous-time transfer function.
```

The software uses a linearization input, `du`, and linearization output `u` to compute `sys`.



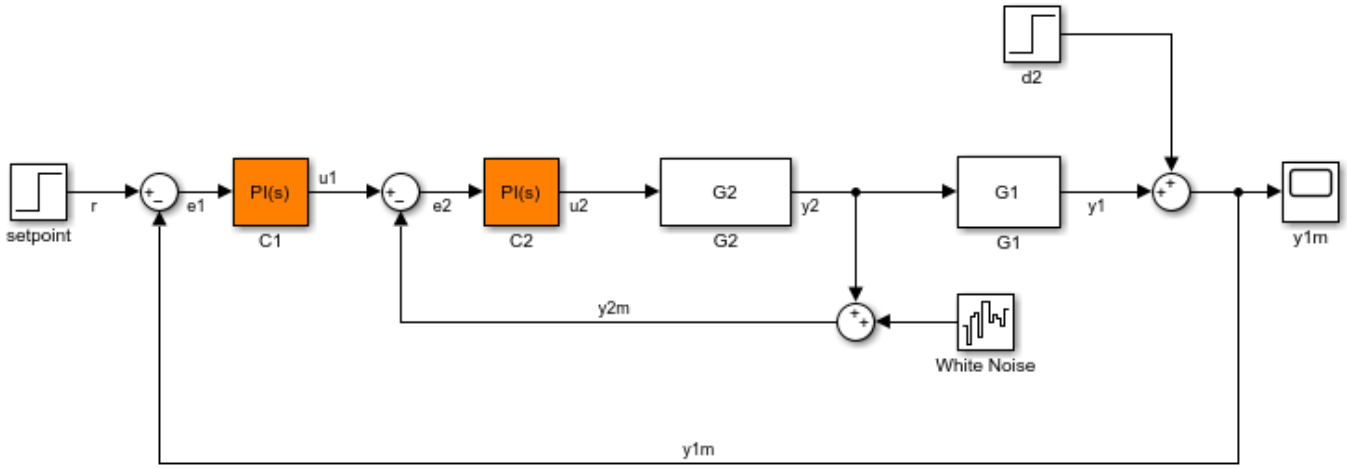
`sys` is the transfer function from `du` to `u`, which is equal to $(I + KG)^{-1}$.

Specify Temporary Loop Opening for Sensitivity Function Calculation

For the `scdcascade` model, obtain the inner-loop sensitivity at the output of `G2`, with the outer loop open.

Open the `scdcascade` model.

```
mdl = 'scdcascade';
open_system(mdl)
```



Create an `slLinearizer` interface for the model.

```
sllin = slLinearizer(mdl);
```

To calculate the sensitivity at the output of `G2`, use the `y2` signal as the analysis point. To eliminate the effects of the outer loop, break the outer loop at `y1m`. Add both these points to `sllin`.

```
addPoint(sllin,{'y2','y1m'});
```

Obtain the sensitivity at `y2` with the outer loop open.

```
sys = getSensitivity(sllin,'y2','y1m');
```

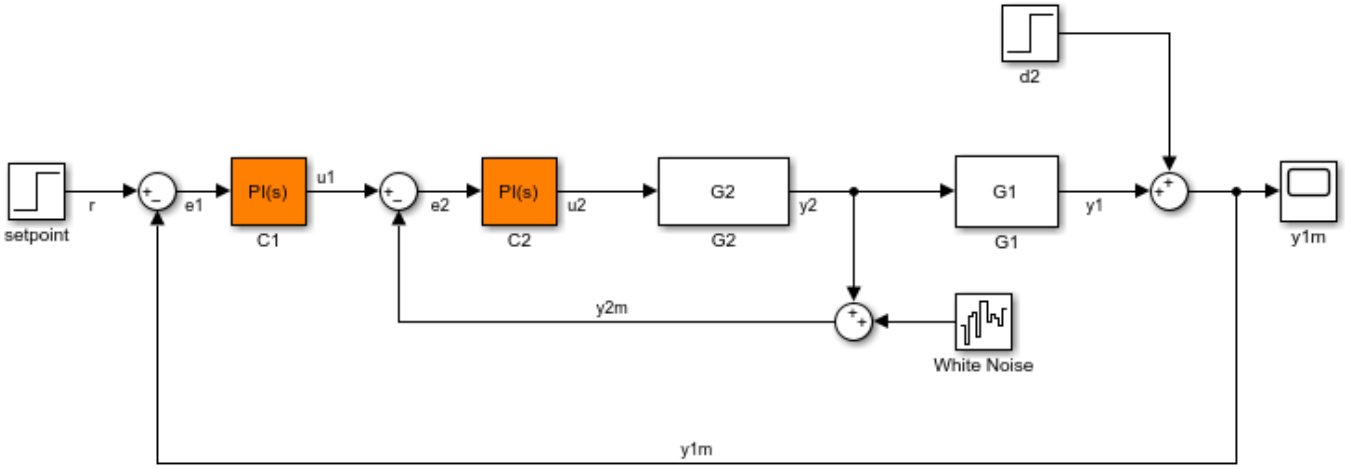
Here, `'y1m'`, the third input argument, specifies a temporary opening of the outer loop.

Obtain Sensitivity Function for Specific Parameter Combination

Suppose you batch linearize the `scdcascade` model for multiple transfer functions. For most linearizations, you vary the proportional (`Kp2`) and integral gain (`Ki2`) of the `C2` controller in the 10% range. For this example, obtain the sensitivity at the output of `G2`, with the outer loop open, for the maximum values of `Kp2` and `Ki2`.

Open the `scdcascade` model.

```
mdl = 'scdcascade';
open_system(mdl)
```



Create an `slLinearizer` interface for the model.

```
sllin = slLinearizer(mdl);
```

Vary the proportional (`Kp2`) and integral gain (`Ki2`) of the `C2` controller in the 10% range.

```
Kp2_range = linspace(0.9*Kp2,1.1*Kp2,3);
Ki2_range = linspace(0.9*Ki2,1.1*Ki2,5);

[Kp2_grid,Ki2_grid] = ndgrid(Kp2_range,Ki2_range);

params(1).Name = 'Kp2';
params(1).Value = Kp2_grid;

params(2).Name = 'Ki2';
params(2).Value = Ki2_grid;

sllin.Parameters = params;
```

To calculate the sensitivity at the output of `G2`, use the `y2` signal as the analysis point. To eliminate the effects of the outer loop, break the outer loop at `y1m`. Add both these points to `sllin` as analysis points.

```
addPoint(sllin,{'y2','y1m'});
```

Determine the index for the maximum values of `Ki2` and `Kp2`.

```
mdl_index = params(1).Value == max(Kp2_range) & params(2).Value == max(Ki2_range);
```

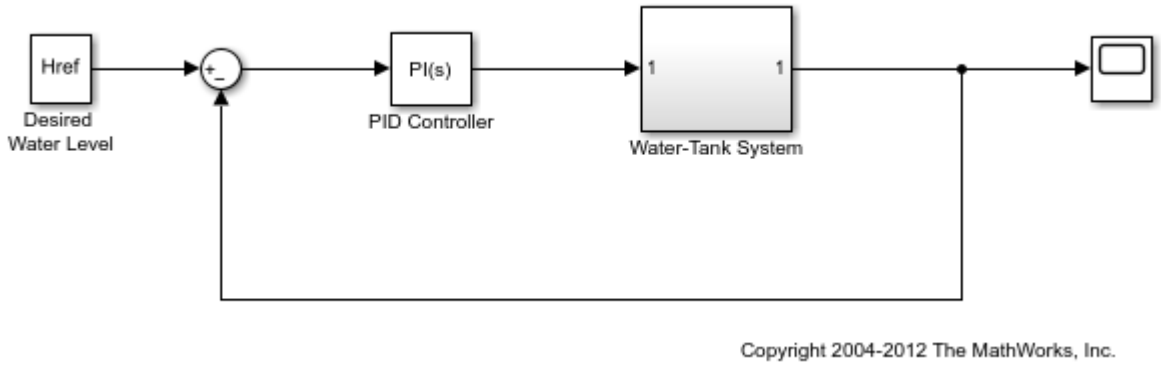
Obtain the sensitivity at the output of `G2` for the specified parameter combination.

```
sys = getSensitivity(sllin,'y2','y1m',mdl_index);
```

Obtain Offsets from Sensitivity Function

Open Simulink model.

```
mdl = 'watertank';
open_system(mdl)
```



Create a linearization option set, and set the `StoreOffsets` option.

```
opt = linearizeOptions('StoreOffsets',true);
```

Create `slLinearizer` interface.

```
sllin = slLinearizer(mdl,opt);
```

Add an analysis point at the tank output port.

```
addPoint(sllin,'watertank/Water-Tank System');
```

Calculate the sensitivity function at the analysis point, and obtain the corresponding linearization offsets.

```
[sys,info] = getSensitivity(sllin,'watertank/Water-Tank System');
```

View offsets.

```
info.Offsets
```

```
ans =

  struct with fields:

            x: [2x1 double]
```

```
          dx: [2x1 double]
           u: 1
           y: 1
   StateName: {2x1 cell}
   InputName: {'watertank/Water-Tank System'}
  OutputName: {'watertank/Water-Tank System'}
          Ts: 0
```

Input Arguments

`s` — Interface to Simulink model

`slLinearizer` interface | `slTuner` interface

Interface to a Simulink model, specified as either an `slLinearizer` interface or an `slTuner` interface.

`pt` — Analysis point signal name

character vector | string | cell array of character vectors | string array

Analysis point on page 18-288 signal name, specified as:

- Character vector or string — Analysis point signal name.

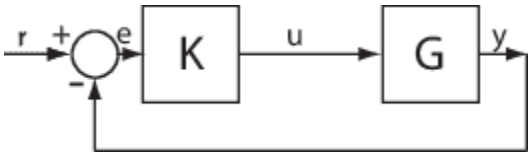
  To determine the signal name associated with an analysis point, type `s`. The software displays the contents of `s` in the MATLAB command window, including the analysis point signal names, block names, and port numbers. Suppose that an analysis point does not have a signal name, but only a block name and port number. You can specify `pt` as the block name. To use a point not in the list of analysis points for `s`, first add the point using `addPoint`.

  You can specify `pt` as a uniquely matching portion of the full signal name or block name. Suppose that the full signal name of an analysis point is `'LoadTorque'`. You can specify `pt` as `'Torque'` as long as `'Torque'` is not a portion of the signal name for any other analysis point of `s`.

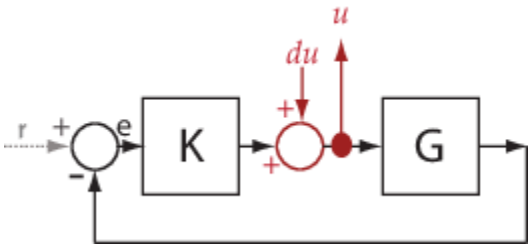
  For example, `pt = 'y1m'`.
- Cell array of character vectors or string array — Specifies multiple analysis point names. For example, `pt = {'y1m','y2m'}`.

To calculate `linsys`, the software adds a linearization input, followed by a linearization output at `pt`.

Consider the following model:

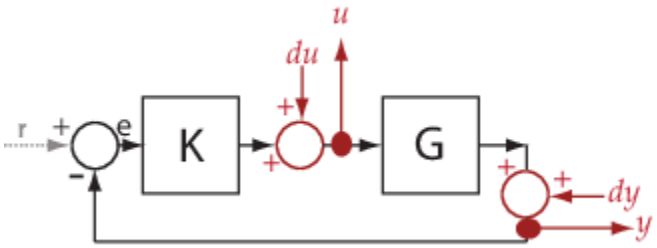


Specify `pt` as `'u'`:



The software computes `linsys` 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 `linsys` as a MIMO transfer function with a transfer function from each linearization input to each linearization output.

`temp_opening` — Temporary opening signal name

character vector | string | cell array of character vectors | string array

Temporary opening signal name, specified as:

- Character vector or string — Analysis point signal name.

  `temp_opening` must specify an analysis point that is in the list of analysis points for `s`. To determine the signal name associated with an analysis point, type `s`. The software displays the contents of `s` in the MATLAB command window, including the analysis point signal names, block names, and port numbers. Suppose that an analysis point does not have a signal name, but only a block name and port number. You can specify `temp_opening` as the block name. To use a point not in the list of analysis points for `s`, first add the point using `addPoint`.

  You can specify `temp_opening` as a uniquely matching portion of the full signal name or block name. Suppose that the full signal name of an analysis point is `'LoadTorque'`. You can specify `temp_opening` as `'Torque'` as long as `'Torque'` is not a portion of the signal name for any other analysis point of `s`.

  For example, `temp_opening = 'y1m'`.
- Cell array of character vectors or string array — Specifies multiple analysis point names. For example, `temp_opening = {'y1m','y2m'}`.

`mdl_index` — Index for linearizations of interest

array of logical values | vector of positive integers

Index for linearizations of interest, specified as:

- Array of logical values — Logical array index of linearizations of interest. Suppose that you vary two parameters, `par1` and `par2`, and want to extract the linearization for the combination of `par1 > 0.5` and `par2 <= 5`. Use:

  ```
  params = s.Parameters;
  mdl_index = params(1).Value>0.5 & params(2).Value <= 5;
  ```

  The expression `params(1).Value>0.5 & params(2).Value<5` uses logical indexing and returns a logical array. This logical array is the same size as `params(1).Value` and `params(2).Value`. Each entry contains the logical evaluation of the expression for corresponding entries in `params(1).Value` and `params(2).Value`.
- Vector of positive integers — Linear index of linearizations of interest. Suppose that you vary two parameters, `par1` and `par2`, and want to extract the linearization for the combination of `par1 > 0.5` and `par2 <= 5`. Use:

  ```
  params = s.Parameters;
  mdl_index = find(params(1).Value>0.5 & params(2).Value <= 5);
  ```

  The expression `params(1).Value>0.5 & params(2).Value<5` returns a logical array. `find` returns the linear index of every true entry in the logical array

Output Arguments

`linsys` — Sensitivity function

state-space model

Sensitivity function, returned as described in the following:

- If you did not configure `s.Parameters` and `s.OperatingPoints`, the software calculates `linsys` using the default model parameter values. The software uses the model initial conditions as the linearization operating point. `linsys` is returned as a state-space model.
- If you configured `s.Parameters` only, the software computes a linearization for each parameter grid point. `linsys` is returned as a state-space model array of the same size as the parameter grid.
- If you configured `s.OperatingPoints` only, the software computes a linearization for each specified operating point. `linsys` is returned as a state-space model array of the same size as `s.OperatingPoints`.
- If you configured `s.Parameters` and specified `s.OperatingPoints` as a single operating point, the software computes a linearization for each parameter grid point. The software uses the specified operating point as the linearization operating point. `linsys` is returned as a state-space model array of the same size as the parameter grid.
- If you configured `s.Parameters` and specified `s.OperatingPoints` as multiple operating point objects, the software computes a linearization for each parameter grid point. The software requires that `s.OperatingPoints` is the same size as the parameter grid specified by `s.Parameters`. The software computes each linearization using corresponding operating points and parameter grid points. `linsys` is returned as a state-space model array of the same size as the parameter grid.
- If you configured `s.Parameters` and specified `s.OperatingPoints` as multiple simulation snapshot times, the software simulates and linearizes the model for each snapshot time and

parameter grid point combination. Suppose that you specify a parameter grid of size `p` and `N` snapshot times. `linsys` is returned as a state-space model array of size `N`-by-`p`.

For most models, `linsys` is returned as an `ss` object or an array of `ss` objects. However, if your model contains one of the following blocks in the linearization path defined by `pt`, then `linsys` returns the specified type of state-space model.

Block	`linsys` Type
Block with a substitution specified as a `genss` object or tunable model object	`genss`
Block with a substitution specified as an uncertain model, such as `uss`	`uss`
Sparse Second Order block	`mechss`
Descriptor State-Space block configured to linearize to a sparse model	`sparss`

`info` — Linearization information
structure

Linearization information, returned as a structure with the following fields:

`Offsets` — Linearization offsets
`[]` (default) | structure | structure array

Linearization offsets, returned as `[]` if `s.Options.StoreOffsets` is `false`. Otherwise, `Offsets` is returned as one of the following:

- If `linsys` is a single state-space model, then `Offsets` is a structure.
- If `linsys` is an array of state-space models, then `Offsets` is a structure array with the same dimensions as `linsys`.

Each offset structure has the following fields:

Field	Description
`x`	State offsets used for linearization, returned as a column vector of length n_x, where n_x is the number of states in `linsys`.
`y`	Output offsets used for linearization, returned as a column vector of length n_y, where n_y is the number of outputs in `linsys`.
`u`	Input offsets used for linearization, returned as a column vector of length n_u, where n_u is the number of inputs in `linsys`.
`dx`	Derivative offsets for continuous time systems or updated state values for discrete-time systems, returned as a column vector of length n_x.
`StateName`	State names, returned as a cell array that contains n_x elements that match the names in `linsys.StateName`.
`InputName`	Input names, returned as a cell array that contains n_u elements that match the names in `linsys.InputName`.
`OutputName`	Output names, returned as a cell array that contains n_y elements that match the names in `linsys.OutputName`.

Field	Description
`Ts`	Sample time of the linearized system, returned as a scalar that matches the sample time in `linsys.Ts`. For continuous-time systems, `Ts` is `0`.

If `Offsets` is a structure array, you can configure an LPV System block using the offsets. To do so, first convert them to the required format using `getOffsetsForLPV`. For an example, see “Approximate Nonlinear Behavior Using Array of LTI Systems” on page 3-70.

`Advisor` — Linearization diagnostic information

`[]` (default) | `LinearizationAdvisor` object | array of `LinearizationAdvisor` objects

Linearization diagnostic information, returned as `[]` if `s.Options.StoreAdvisor` is `false`. Otherwise, `Advisor` is returned as one of the following:

- If `linsys` is a single state-space model, `Advisor` is a `LinearizationAdvisor` object.
- If `linsys` is an array of state-space models, `Advisor` is an array of `LinearizationAdvisor` objects with the same dimensions as `linsys`.

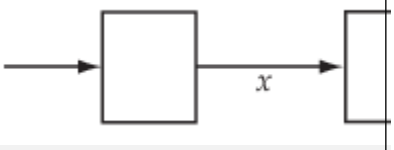
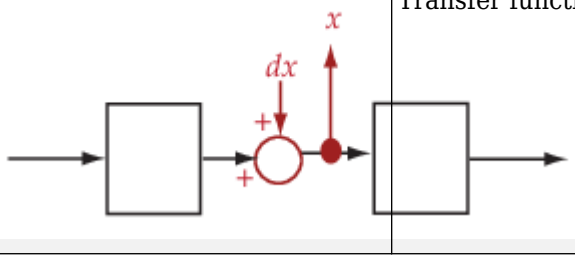
`LinearizationAdvisor` objects store linearization diagnostic information for individual linearized blocks. For an example of troubleshooting linearization results using a `LinearizationAdvisor` object, see “Troubleshoot Linearization Results at Command Line” on page 4-28.

More About

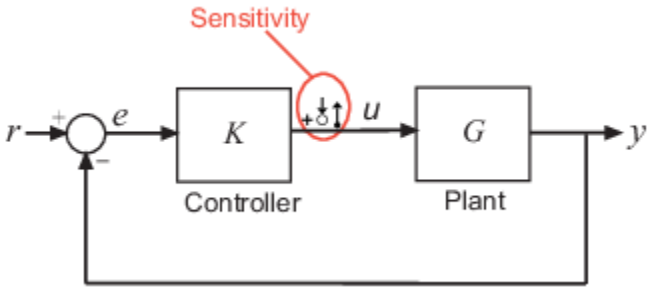
Sensitivity Function

The sensitivity function, also referred to simply as sensitivity, measures how sensitive a signal is to an added disturbance. Sensitivity is a closed-loop measure. Feedback reduces the sensitivity in the frequency band where the open-loop gain is greater than 1.

To compute the sensitivity at an analysis point, `x`, the software injects a disturbance signal, `dx`, at the point. Then, the software computes the transfer function from `dx` to `x`, which is equal to the sensitivity function at `x`.

Analysis Point in Simulink Model	How `getSensitivity` Interprets Analysis Point	Sensitivity Function
		Transfer function from `dx` to `x`

For example, consider the following model where you compute the sensitivity function at `u`:



Here, the software injects a disturbance signal (`du`) at `u`. The sensitivity at `u`, S_u, is the transfer function from `du` to `u`. The software calculates S_u as follows:

$$u = du - KGu$$
$$\rightarrow (I + KG)u = du$$
$$\rightarrow u = (^{I}du$$
$$\therefore S_u = (^{I}$$

Here, I is an identity matrix of the same size as KG.

Similarly, to compute the sensitivity at `y`, the software injects a disturbance signal (`dy`) at `y`. The software computes the sensitivity function as the transfer function from `dy` to `y`. This transfer function is equal to $(I+GK)^{-1}$, where I is an identity matrix of the same size as GK.

The software does not modify the Simulink model when it computes the sensitivity transfer function.

Analysis Point

Analysis points, used by the `slLinearizer` and `slTuner` interfaces, identify locations within a model that are relevant for linear analysis and control system tuning. You use analysis points as inputs to the linearization commands, such as `getIOTransfer`, `getLoopTransfer`, `getSensitivity`, and `getCompSensitivity`. As inputs to the linearization commands, analysis points can specify any open-loop or closed-loop transfer function in a model. You can also use analysis points to specify design requirements when tuning control systems using commands such as `systune`.

Location refers to a specific block output port within a model or to a bus element in such an output port. For convenience, you can use the name of the signal that originates from this port to refer to an analysis point.

You can add analysis points to an `slLinearizer` or `slTuner` interface, `s`, when you create the interface. For example:

```
s = slLinearizer('scdcascade',{'u1','y1'});
```

Alternatively, you can use the `addPoint` command.

To view all the analysis points of `s`, type `s` at the command prompt to display the interface contents. For each analysis point of `s`, the display includes the block name and port number and the name of the signal that originates at this point. You can also programmatically obtain a list of all the analysis points using `getPoints`.

For more information about how you can use analysis points, see “Mark Signals of Interest for Control System Analysis and Design” on page 2-38 and “Mark Signals of Interest for Batch Linearization” on page 3-9.

Permanent Openings

Permanent openings, used by the `slLinearizer` and `slTuner` interfaces, identify locations within a model where the software breaks the signal flow. The software enforces these openings for linearization and tuning. Use permanent openings to isolate a specific model component. Suppose that you have a large-scale model capturing aircraft dynamics and you want to perform linear analysis on the airframe only. You can use permanent openings to exclude all other components of the model. Another example is when you have cascaded loops within your model and you want to analyze a specific loop.

Location refers to a specific block output port within a model. For convenience, you can use the name of the signal that originates from this port to refer to an opening.

You can add permanent openings to an `slLinearizer` or `slTuner` interface, `s`, when you create the interface or by using the `addOpening` command. To remove a location from the list of permanent openings, use the `removeOpening` command.

To view all the openings of `s`, type `s` at the command prompt to display the interface contents. For each permanent opening of `s`, the display includes the block name and port number and the name of the signal that originates at this location. You can also programmatically obtain a list of all the permanent loop openings using `getOpenings`.

See Also

`slLinearizer` | `slTuner` | `addPoint` | `addOpening` | `getIOTransfer` | `getLoopTransfer` | `getCompSensitivity`

Topics

“Vary Parameter Values and Obtain Multiple Transfer Functions” on page 3-21
“Vary Operating Points and Obtain Multiple Transfer Functions Using slLinearizer Interface” on page 3-28
“Analyze Command-Line Batch Linearization Results Using Response Plots” on page 3-33
“How the Software Treats Loop Openings” on page 2-31

Introduced in R2013b

refresh

Resynchronize `slLinearizer` or `slTuner` interface with current model state

Syntax

```
refresh(s)
```

Description

`refresh(s)` resynchronizes the `slLinearizer` or `slTuner` interface, `s`, with the current state of the model. The interface recompiles the model for the next call to functions that either return transfer functions (such as `getIOTransfer` and `getLoopTransfer`) or functions that tune model parameters (such as `systune` or `looptune`). This model recompilation ensures that the interface uses the current model state when computing linearizations. Block parameterizations and values for tuned blocks are preserved. Use `setBlockParam` to sync blocks with the model.

Use this command after you make changes to the model that impact linearization. Changes that impact linearization include modifying parameter values and reconfiguring blocks and signals.

Examples

Resynchronize slLinearizer Interface with Current Model State

Create an `slLinearizer` interface.

```
sllin = slLinearizer('scdcascade');
```

Generally, you configure the interface with analysis points, openings, operating points, and parameter values. Then, you linearize the model using the `getIOTransfer`, `getLoopTransfer`, `getSensitivity`, and `getCompSensitivity` commands. The first time you call one of these commands with `sllin`, the software stores the state of the model in `sllin` and uses it to compute the linearization.

You can change the model after your first call to `getIOTransfer`, `getLoopTransfer`, `getSensitivity`, or `getCompSensitivity` with `sllin`. Some changes impact the linearization, such as changing parameter values. If your change impacts the linearization, call `refresh` to get expected linearization results. For this example, change the proportional gain of the C2 PID controller block.

```
set_param('scdcascade/C2','P','10')
```

Trigger the interface to recompile the model for the next call to `getIOTransfer`, `getLoopTransfer`, `getSensitivity`, or `getCompSensitivity`.

```
refresh(sllin);
```

Resynchronize slTuner Interface with Current Model State

Create an `slTuner` interface.

```
st = slTuner('scdcascade','C2');
```

Generally, you configure the interface with analysis points, openings, operating points, and parameter values. Then, you tune the model block parameters using the `systune` and `looptune` commands. You can also analyze various transfer functions in the model using commands such as `getIOTransfer` and `getLoopTransfer`. The first time you call one of these commands with `st`, the software stores the state of the model in `st` and uses it to compute the linearization.

You can change the model after your first call to one of these commands. Some changes impact the linearization, such as changing parameter values. If your change impacts the linearization, call `refresh` to get expected linearization results. For this example, change the proportional gain of the C1 PID controller block.

```
set_param('scdcascade/C1','P','10')
```

Trigger the interface to recompile the model for the next call to commands such as `getIOTransfer`, `getLoopTransfer`, or `systune`.

```
refresh(st);
```

Input Arguments

s — Interface to Simulink model

`slLinearizer` interface | `slTuner` interface

Interface to a Simulink model, specified as either an `slLinearizer` interface or an `slTuner` interface.

See Also

`slLinearizer` | `slTuner` | `systune` | `looptune` | `getIOTransfer` | `getLoopTransfer` | `getSensitivity` | `getCompSensitivity`

Introduced in R2013b

removeAllOpenings

Remove all openings from list of permanent openings in `slLinearizer` or `slTuner` interface

Syntax

```
removeAllOpenings(s)
```

Description

`removeAllOpenings(s)` removes all openings from the list of permanent openings on page 18-293 in the `slLinearizer` or `slTuner` interface, `s`. This function does not modify the Simulink model associated with `s`.

Examples

Remove All Openings from slLinearizer Interface

Create an `slLinearizer` interface for the `scdcascade` model.

```
sllin = slLinearizer('scdcascade');
```

Generally, you configure the interface with analysis points, openings, operating points, and parameter values. For this example, add two openings to the interface.

```
addOpening(sllin,{'y2m','y1m'});
```

`'y2m'` and `'y1m'` are the names of two feedback signals in the `scdcascade` model. The `addOpening` command adds these signals to the list of openings for `sllin`.

Remove all the openings from `sllin`.

```
removeAllOpenings(sllin);
```

To verify that all openings have been removed, display the contents of `sllin`, and examine the information about the interface openings.

```
sllin
```

```
slLinearizer linearization interface for "scdcascade":

No analysis points. Use the addPoint command to add new points.
No permanent openings. Use the addOpening command to add new permanent openings.
Properties with dot notation get/set access:
      Parameters         : []
      OperatingPoints    : [] (model initial condition will be used.)
      BlockSubstitutions : []
      Options            : [1x1 linearize.LinearizeOptions]
```

Input Arguments

s — Interface to Simulink model

`slLinearizer` interface | `slTuner` interface

Interface to a Simulink model, specified as either an `slLinearizer` interface or an `slTuner` interface.

More About

Permanent Openings

Permanent openings, used by the `slLinearizer` and `slTuner` interfaces, identify locations within a model where the software breaks the signal flow. The software enforces these openings for linearization and tuning. Use permanent openings to isolate a specific model component. Suppose that you have a large-scale model capturing aircraft dynamics and you want to perform linear analysis on the airframe only. You can use permanent openings to exclude all other components of the model. Another example is when you have cascaded loops within your model and you want to analyze a specific loop.

Location refers to a specific block output port within a model. For convenience, you can use the name of the signal that originates from this port to refer to an opening.

You can add permanent openings to an `slLinearizer` or `slTuner` interface, `s`, when you create the interface or by using the `addOpening` command. To remove a location from the list of permanent openings, use the `removeOpening` command.

To view all the openings of `s`, type `s` at the command prompt to display the interface contents. For each permanent opening of `s`, the display includes the block name and port number and the name of the signal that originates at this location. You can also programmatically obtain a list of all the permanent loop openings using `getOpenings`.

See Also

`slLinearizer` | `slTuner` | `addOpening` | `removeOpening`

Introduced in R2013b

removeAllPoints

Remove all points from list of analysis points in `slLinearizer` or `slTuner` interface

Syntax

```
removeAllPoints(s)
```

Description

`removeAllPoints(s)` removes all points from the list of analysis points on page 18-295 for the `slLinearizer` or `slTuner` interface, `s`. This function does not modify the model associated with `s`.

Examples

Remove All Analysis Points

Create an `slLinearizer` interface for the `scdcascade` model. Add analysis points for the `r`, `e1`, and `y1m` signals.

```
sllin = slLinearizer('scdcascade',{'r','e1','y1m'});
```

Remove all signals from the list of interface analysis points.

```
removeAllPoints(sllin);
```

To verify that all analysis points have been removed, display the contents of `sllin`, and examine the information about the interface analysis points.

```
sllin
```

```
slLinearizer linearization interface for "scdcascade":

No analysis points. Use the addPoint command to add new points.
No permanent openings. Use the addOpening command to add new permanent openings.
Properties with dot notation get/set access:
      Parameters         : []
      OperatingPoints    : [] (model initial condition will be used.)
      BlockSubstitutions : []
      Options            : [1x1 linearize.LinearizeOptions]
```

Input Arguments

s — Interface to Simulink model

`slLinearizer` interface | `slTuner` interface

Interface to a Simulink model, specified as either an `slLinearizer` interface or an `slTuner` interface.

More About

Analysis Points

Analysis points, used by the `slLinearizer` and `slTuner` interfaces, identify locations within a model that are relevant for linear analysis and control system tuning. You use analysis points as inputs to the linearization commands, such as `getIOTransfer`, `getLoopTransfer`, `getSensitivity`, and `getCompSensitivity`. As inputs to the linearization commands, analysis points can specify any open-loop or closed-loop transfer function in a model. You can also use analysis points to specify design requirements when tuning control systems using commands such as `systune`.

Location refers to a specific block output port within a model or to a bus element in such an output port. For convenience, you can use the name of the signal that originates from this port to refer to an analysis point.

You can add analysis points to an `slLinearizer` or `slTuner` interface, `s`, when you create the interface. For example:

```
s = slLinearizer('scdcascade',{'u1','y1'});
```

Alternatively, you can use the `addPoint` command.

To view all the analysis points of `s`, type `s` at the command prompt to display the interface contents. For each analysis point of `s`, the display includes the block name and port number and the name of the signal that originates at this point. You can also programmatically obtain a list of all the analysis points using `getPoints`.

For more information about how you can use analysis points, see “Mark Signals of Interest for Control System Analysis and Design” on page 2-38 and “Mark Signals of Interest for Batch Linearization” on page 3-9.

See Also

`slTuner` | `removePoint` | `slLinearizer` | `addPoint`

Introduced in R2013b

removeOpening

Remove opening from list of permanent loop openings in `slLinearizer` or `slTuner` interface

Syntax

```
removeOpening(s,op)
```

Description

`removeOpening(s,op)` removes the specified opening, `op`, from the list of permanent openings on page 18-299 for the `slLinearizer` or `slTuner` interface, `s`. You can specify `op` to remove either a single or multiple openings.

`removeOpening` does not modify the model associated with `s`.

Examples

Remove Opening Using Signal Name

Create an `slLinearizer` interface for the `scdcascade` model.

```
sllin = slLinearizer('scdcascade');
```

Generally, you configure the interface with analysis points, openings, operating points, and parameter values. For this example, add only openings to the interface.

```
addOpening(sllin,{'y2m','y1m','u1'});
```

`'y2m'`, `'y1m'`, and `'u1'` are the names of signals in the `scdcascade` model. The `addOpening` command adds these signals to the list of permanent openings for `sllin`.

Remove the `'y1m'` opening from `sllin`.

```
removeOpening(sllin,'y1m');
```

Remove Multiple Openings Using Signal Names

Create an `slLinearizer` interface for the `scdcascade` model.

```
sllin = slLinearizer('scdcascade');
```

Generally, you configure the interface with analysis points, openings, operating points, and parameter values. For this example, add only openings to the interface.

```
addOpening(sllin,{'y2m','y1m','u1'});
```

`'y2m'`, `'y1m'`, and `'u1'` are the names of signals in the `scdcascade` model. The `addOpening` command adds these signals to the list of permanent openings for `sllin`.

Remove the `'y1m'` and `'y2m'` openings from `sllin`.

```
removeOpening(sllin,{'y1m','y2m'});
```

Remove Opening Using Index

Create an `slLinearizer` interface for the `scdcascade` model.

```
sllin = slLinearizer('scdcascade');
```

Generally, you configure the interface with analysis points, loop openings, operating points, and parameter values. For this example, add only openings to the interface.

```
addOpening(sllin,{'y2m','y1m','u1'});
```

`'y2m'`, `'y1m'`, and `'u1'` are the names of signals in the `scdcascade` model. The `addOpening` command adds these signals to the list of permanent openings for `sllin`.

Determine the index number of the opening you want to remove. To do this, display the contents of the interface, which includes opening index numbers, in the Command Window.

For this example, remove the `'y1m'` opening from `sllin`.

```
sllin
```

```
slLinearizer linearization interface for "scdcascade":

No analysis points. Use the addPoint command to add new points.
3 Permanent openings:
-------------------------
Opening 1:
- Block: scdcascade/Sum3
- Port: 1
- Signal Name: y2m
Opening 2:
- Block: scdcascade/Sum
- Port: 1
- Signal Name: y1m
Opening 3:
- Block: scdcascade/C1
- Port: 1
- Signal Name: u1

Properties with dot notation get/set access:
      Parameters         : []
      OperatingPoints    : [] (model initial condition will be used.)
      BlockSubstitutions : []
      Options            : [1x1 linearize.LinearizeOptions]
```

The displays shows that `'y1m'` is the second opening of `sllin` .

Remove the opening from the interface.

```
removeOpening(sllin,2);
```

Remove Multiple Openings Using Index

Create an `slLinearizer` interface for the `scdcascade` model.

```
sllin = slLinearizer('scdcascade');
```

Generally, you configure the interface with analysis points, openings, operating points, and parameter values. For this example, add only openings to the interface.

```
addOpening(sllin,{'y2m','y1m','u1'});
```

`'y2m'`, `'y1m'`, and `'u1'` are the names of signals in the `scdcascade` model. The `addOpening` command adds these signals to the list of permanent openings for `sllin`.

Determine the index numbers of the openings you want to remove. To do this, display the contents of the interface, which includes opening index numbers, in the Command Window.

For this example, remove the `'y2m'` and `'y1m'` openings from `sllin`.

```
sllin
```

```
slLinearizer linearization interface for "scdcascade":

No analysis points. Use the addPoint command to add new points.
3 Permanent openings:
--------------------------
Opening 1:
- Block: scdcascade/Sum3
- Port: 1
- Signal Name: y2m
Opening 2:
- Block: scdcascade/Sum
- Port: 1
- Signal Name: y1m
Opening 3:
- Block: scdcascade/C1
- Port: 1
- Signal Name: u1

Properties with dot notation get/set access:
      Parameters         : []
      OperatingPoints    : [] (model initial condition will be used.)
      BlockSubstitutions : []
      Options            : [1x1 linearize.LinearizeOptions]
```

The displays shows that `'y2m'` and `'y1m'` are the first and second openings of `sllin` .

Remove the openings from the interface.

```
removeOpening(sllin,[1 2]);
```

Input Arguments

s — Interface to Simulink model

`slLinearizer` interface | `slTuner` interface

Interface to a Simulink model, specified as either an `slLinearizer` interface or an `slTuner` interface.

op — Opening

character vector | string | cell array of character vectors | string array | positive integer | vector of positive integers

Opening on page 18-299 to remove from the list of permanent openings for `s`, specified as:

- Character vector or string — Opening signal name.

  To determine the signal name associated with an opening, type `s`. The software displays the contents of `s` in the MATLAB command window, including the opening signal names, block names, and port numbers. Suppose an opening does not have a signal name, but only a block name and port number. You can specify `op` as the block name.

  You can specify `op` as a uniquely matching portion of the full signal name or block name. Suppose the full signal name of an opening is `'LoadTorque'`. You can specify `op` as `'Torque'` as long as `'Torque'` is not a portion of the signal name for any other opening of `s`.

  For example, `op = 'y1m'`.
- Cell array of character vectors or string array — Specifies multiple opening names. For example, `op = {'y1m','y2m'}`.
- Positive integer — Opening index.

  To determine the index of an opening, type `s`. The software displays the contents of `s` in the MATLAB command window, including the opening indices. For example, `op = 1`.
- Vector of positive integers — Specifies multiple opening indices. For example, `op = [1 2]`.

More About

Permanent Openings

Permanent openings, used by the `slLinearizer` and `slTuner` interfaces, identify locations within a model where the software breaks the signal flow. The software enforces these openings for linearization and tuning. Use permanent openings to isolate a specific model component. Suppose that you have a large-scale model capturing aircraft dynamics and you want to perform linear analysis on the airframe only. You can use permanent openings to exclude all other components of the model. Another example is when you have cascaded loops within your model and you want to analyze a specific loop.

Location refers to a specific block output port within a model. For convenience, you can use the name of the signal that originates from this port to refer to an opening.

You can add permanent openings to an `slLinearizer` or `slTuner` interface, `s`, when you create the interface or by using the `addOpening` command. To remove a location from the list of permanent openings, use the `removeOpening` command.

To view all the openings of `s`, type `s` at the command prompt to display the interface contents. For each permanent opening of `s`, the display includes the block name and port number and the name of the signal that originates at this location. You can also programmatically obtain a list of all the permanent loop openings using `getOpenings`.

See Also

`slTuner` | `removeAllOpenings` | `addOpening` | `slLinearizer` | `removePoint`

Introduced in R2013b

removePoint

Remove point from list of analysis points in `slLinearizer` or `slTuner` interface

Syntax

```
removePoint(s,pt)
```

Description

`removePoint(s,pt)` removes the specified point, `pt`, from the list of analysis points on page 18-304 for the `slLinearizer` or `slTuner` interface, `s`. You can specify `pt` to remove either a single or multiple points.

`removePoint` does not modify the model associated with `s`.

Examples

Remove Analysis Point Using Signal Name

Create an `slLinearizer` interface for the `scdcascade` model. Add analysis points for the `r`, `e1`, and `y1m` signals.

```
sllin = slLinearizer('scdcascade',{'r','e1','y1m'});
```

Remove the `y1m` point from the interface.

```
removePoint(sllin,'y1m');
```

Remove Multiple Analysis Points Using Signal Name

Create an `slLinearizer` interface for the `scdcascade` model. Add analysis points for the `r`, `e1`, and `y1m` signals.

```
sllin = slLinearizer('scdcascade',{'r','e1','y1m'});
```

Remove the `y1m` and `e1` points from the interface.

```
removePoint(sllin,{'y1m','e1'});
```

Remove Analysis Point Using Index

Create an `slLinearizer` interface for the `scdcascade` model. Add analysis points for the `r`, `e1`, and `y1m` signals.

```
sllin = slLinearizer('scdcascade',{'r','e1','y1m'});
```

Determine the index number of the point you want to remove. To do this, display the contents of the interface, which includes analysis point index numbers, in the Command Window.

For this example, remove the `y1m` point from `sllin`.

```
sllin
```

```
slLinearizer linearization interface for "scdcascade":

3 Analysis points:
--------------------------
Point 1:
- Block: scdcascade/setpoint
- Port: 1
- Signal Name: r
Point 2:
- Block: scdcascade/Sum1
- Port: 1
- Signal Name: e1
Point 3:
- Block: scdcascade/Sum
- Port: 1
- Signal Name: y1m

No permanent openings. Use the addOpening command to add new permanent openings.
Properties with dot notation get/set access:
      Parameters         : []
      OperatingPoints    : [] (model initial condition will be used.)
      BlockSubstitutions : []
      Options            : [1x1 linearize.LinearizeOptions]
```

The displays shows that `y1m` is the third analysis point of `sllin` .

Remove the point from the interface.

```
removePoint(sllin,3);
```

Remove Multiple Analysis Points Using Index

Create an `slLinearizer` interface for the `scdcascade` model. Add analysis points for the `r`, `e1`, and `y1m` signals.

```
sllin = slLinearizer('scdcascade',{'r','e1','y1m'});
```

Determine the index numbers of the points you want to remove. To do this, display the contents of the interface, which includes analysis point index numbers, in the Command Window.

For this example, remove the `e1` and `y1m` points from `sllin`.

```
sllin
```

```
slLinearizer linearization interface for "scdcascade":

3 Analysis points:
```

```
--------------------------
Point 1:
- Block: scdcascade/setpoint
- Port: 1
- Signal Name: r
Point 2:
- Block: scdcascade/Sum1
- Port: 1
- Signal Name: e1
Point 3:
- Block: scdcascade/Sum
- Port: 1
- Signal Name: y1m

No permanent openings. Use the addOpening command to add new permanent openings.
Properties with dot notation get/set access:
      Parameters         : []
      OperatingPoints    : [] (model initial condition will be used.)
      BlockSubstitutions : []
      Options            : [1x1 linearize.LinearizeOptions]
```

The displays shows that `e1` and `y1m` are the second and third analysis points of `sllin` .

Remove the points from the interface.

```
removePoint(sllin,[2 3]);
```

Input Arguments

`s` — Interface to Simulink model

`slLinearizer` interface | `slTuner` interface

Interface to a Simulink model, specified as either an `slLinearizer` interface or an `slTuner` interface.

`pt` — Analysis point

character vector | string | cell array of character vectors | string array | positive integer | vector of positive integers

Analysis point on page 18-304 to remove from the list of analysis points for `s`, specified as:

- Character vector or string — Analysis point signal name.

  To determine the signal name associated with an analysis point, type `s`. The software displays the contents of `s` in the MATLAB command window, including the analysis point signal names, block names, and port numbers. Suppose an analysis point does not have a signal name, but only a block name and port number. You can specify `pt` as the block name.

  You can specify `pt` as a uniquely matching portion of the full signal name or block name. Suppose the full signal name of an analysis point is `'LoadTorque'`. You can specify `pt` as `'Torque'` as long as `'Torque'` is not a portion of the signal name for any other analysis point of `s`.

  For example, `pt = 'y1m'`.
- Cell array of character vectors or string array — Specifies multiple analysis point names. For example, `pt = {'y1m','y2m'}`.

- Positive integer or — Analysis point index.

  To determine the index of an analysis point, type `s`. The software displays the contents of `s` in the MATLAB command window, including the analysis points indices.

  For example, `pt = 1`.
- Vector of positive integers — Specifies multiple analysis point indices. For example, `pt = [1 2]`.

More About

Analysis Points

Analysis points, used by the `slLinearizer` and `slTuner` interfaces, identify locations within a model that are relevant for linear analysis and control system tuning. You use analysis points as inputs to the linearization commands, such as `getIOTransfer`, `getLoopTransfer`, `getSensitivity`, and `getCompSensitivity`. As inputs to the linearization commands, analysis points can specify any open-loop or closed-loop transfer function in a model. You can also use analysis points to specify design requirements when tuning control systems using commands such as `systune`.

Location refers to a specific block output port within a model or to a bus element in such an output port. For convenience, you can use the name of the signal that originates from this port to refer to an analysis point.

You can add analysis points to an `slLinearizer` or `slTuner` interface, `s`, when you create the interface. For example:

```
s = slLinearizer('scdcascade',{'u1','y1'});
```

Alternatively, you can use the `addPoint` command.

To view all the analysis points of `s`, type `s` at the command prompt to display the interface contents. For each analysis point of `s`, the display includes the block name and port number and the name of the signal that originates at this point. You can also programmatically obtain a list of all the analysis points using `getPoints`.

For more information about how you can use analysis points, see “Mark Signals of Interest for Control System Analysis and Design” on page 2-38 and “Mark Signals of Interest for Batch Linearization” on page 3-9.

See Also

`slTuner` | `removeAllPoints` | `addPoint` | `slLinearizer` | `removeOpening`

Introduced in R2013b

addBlock

Add block to list of tuned blocks for `slTuner` interface

Syntax

```
addBlock(st,blk)
```

Description

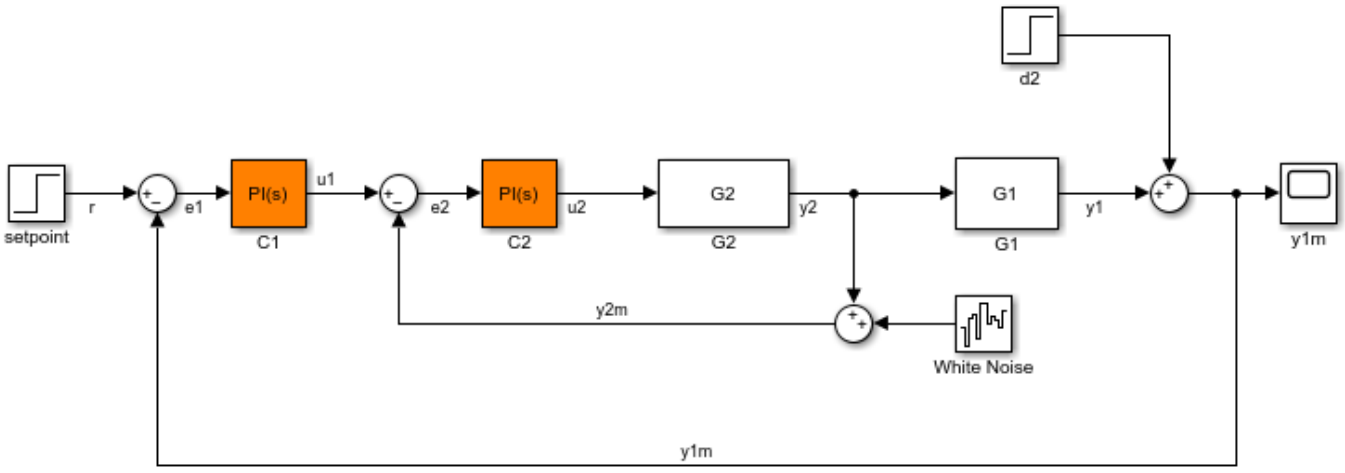
`addBlock(st,blk)` adds the block referenced by `blk` to the list of tuned blocks on page 18-306 of the `slTuner` interface, `st`.

Examples

Add Block to slTuner Interface

Open the Simulink model.

```
mdl = 'scdcascade';
open_system(mdl)
```



Create an `slTuner` interface for the Simulink model, and add a block to the list of tuned blocks of the interface.

```
st = slTuner(mdl,'C1');
addBlock(st,'C2');
```

Input Arguments

`st` — Interface for tuning control systems modeled in Simulink

`slTuner` interface

Interface for tuning control systems modeled in Simulink, specified as an `slTuner` interface.

`blk` — Block
character vector | string | cell array of character vectors | string array

Block to add to the list of tuned blocks on page 18-306 for `st`, specified as:

- Character vector or string — Block path. You can specify the full block path or a partial path. The partial path must match the end of the full block path and unambiguously identify the block to add. For example, you can refer to a block by its name, provided the block name appears only once in the Simulink model.

  For example, `blk = 'scdcascade/C1'`.
- Cell array of character vectors or string array — Multiple block paths.

  For example, `blk = {'scdcascade/C1','scdcascade/C2'}`.

More About

Tuned Block

Tuned blocks, used by the `slTuner` interface, identify blocks in a Simulink model whose parameters are to be tuned to satisfy tuning goals. You can tune most Simulink blocks that represent linear elements such as gains, transfer functions, or state-space models. (For the complete list of blocks that support tuning, see “How Tuned Simulink Blocks Are Parameterized” on page 10-27). You can also tune more complex blocks such as SubSystem or S-Function blocks by specifying an equivalent tunable linear model.

Use tuning commands such as `systune` to tune the parameters of tuned blocks.

You must specify tuned blocks (for example, `C1` and `C2`) when you create an `slTuner` interface.

```
st = slTuner('scdcascade',{'C1','C2'})
```

You can modify the list of tuned blocks using `addBlock` and `removeBlock`.

To interact with the tuned blocks use:

- `getBlockParam`, `getBlockValue`, and `getTunedValue` to access the tuned block parameterizations and their current values.
- `setBlockParam`, `setBlockValue`, and `setTunedValue` to modify the tuned block parameterizations and their values.
- `writeBlockValue` to update the blocks in a Simulink model with the current values of the tuned block parameterizations.

See Also
`slTuner` | `removeBlock` | `addPoint` | `addOpening`

Introduced in R2014a

getBlockParam

Get parameterization of tuned block in `slTuner` interface

Syntax

```
blk_param = getBlockParam(st,blk)
[blk_param1,...,blk_paramN] = getBlockParam(st,blk1,...,blkN)

S = getBlockParam(st)
```

Description

`getBlockParam` lets you retrieve the current parameterization of a tuned block on page 18-310 in an `slTuner` interface.

An `slTuner` interface parameterizes each tuned Simulink block as a Control Design Block, or a generalized parametric model of type `genmat` or `genss`. This parameterization specifies the tuned variables"Tuned Variables" on page 18-311 for commands such as `systune`.

`blk_param = getBlockParam(st,blk)` returns the parameterization used to tune the Simulink block, `blk`.

`[blk_param1,...,blk_paramN] = getBlockParam(st,blk1,...,blkN)` returns the parameterizations of one or more specified blocks.

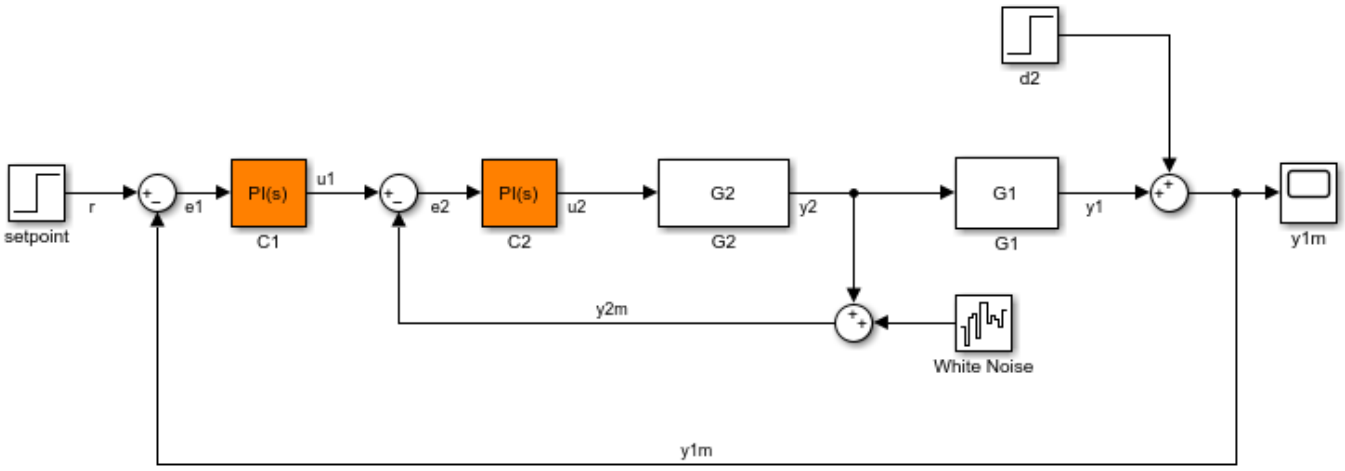
`S = getBlockParam(st)` returns a structure containing the parameterizations of all the tuned blocks of `st`.

Examples

Get Parameterization of Tuned Block

Create an `slTuner` interface for the `scdcascade` model.

```
open_system('scdcascade');
st = slTuner('scdcascade',{'C1','C2'});
```



Examine the block parameterization of one of the tuned blocks.

```
blk_param = getBlockParam(st,'C1')
```

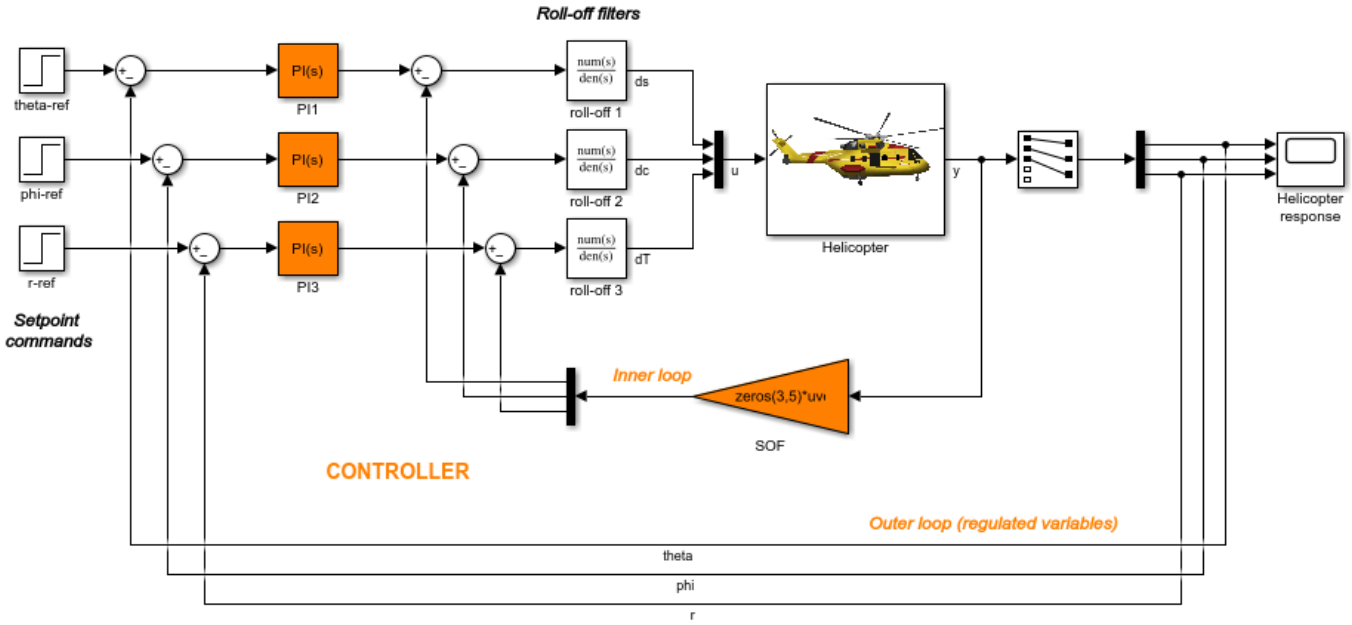
```
blk_param =

  Tunable continuous-time PID controller "C1" with formula:

             1
  Kp + Ki * ---
             s

  and tunable parameters Kp, Ki.

Type "pid(blk_param)" to see the current value and "get(blk_param)" to see all properties.
```

The block `C1` is a PID Controller block. Therefore, its parameterization in `st` is a `tunablePID` Control Design Block.

Get Parameterizations of Multiple Tuned blocks

Create an `slTuner` interface for the `scdhelicopter` model.

```
open_system('scdhelicopter')
st = slTuner('scdhelicopter',{'PI1','PI2','PI3','SOF'});
```



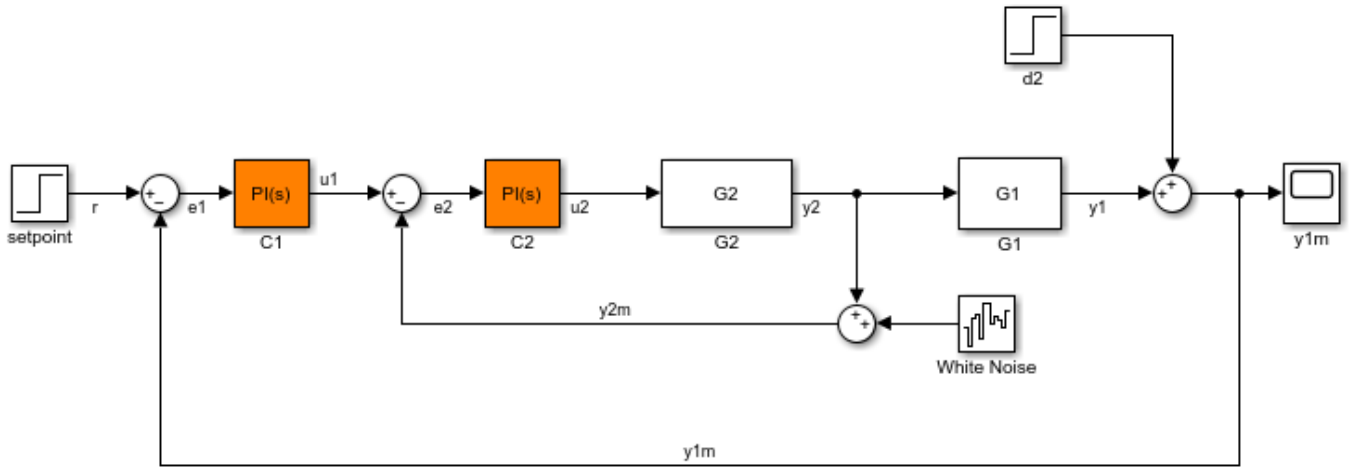
Retrieve the parameterizations for the PI controllers in the model.

```
[parPI1,parPI2,parPI3] = getBlockParam(st,'PI1','PI2','PI3');
```

Get Parameterizations of All Tuned Blocks

Create an `slTuner` interface for the `scdcascade` model.

```
open_system('scdcascade')
st = slTuner('scdcascade',{'C1','C2'});
```



Retrieve the parameterizations for both tuned blocks in `st`.

```
blockParams = getBlockParam(st)
```

```
blockParams =

  struct with fields:

    C1: [1x1 tunablePID]
    C2: [1x1 tunablePID]
```

`blockParams` is a structure with field names corresponding to the names of the tunable blocks in `st`. The field values of `blockParams` are `tunablePID` models, because `C1` and `C2` are both PID Controller blocks.

Input Arguments

`st` — Interface for tuning control systems modeled in Simulink

`slTuner` interface

Interface for tuning control systems modeled in Simulink, specified as an `slTuner` interface.

`blk` — Block

character vector | string

Block in the list of tuned blocks for `st`, specified as a character vector or string. You can specify the full block path or any portion of the block path that uniquely identifies the block among the other tuned blocks of `st`.

Example: `blk = 'scdcascade/C1'`, `blk = "C1"`

Output Arguments

`blk_param` — Parameterization of tuned block

control design block | generalized model | tunable surface | `[]`

Parameterization of the specified tuned block, returned as one of the following:

- A tunable Control Design Block.
- A tunable `genss` model, tunable `genmat` matrix, or `tunableSurface`, if you specified such a parameterization for `blk` using `setBlockParam`.
- An empty array (`[]`), if `slTuner` cannot parameterize `blk`. You can use `setBlockParam` to specify a parameterization for such blocks.

`S` — Parameterizations of all tuned blocks

structure

Parameterization of all tuned blocks in `st`, returned as a structure. The field names in `S` are the names of the tuned blocks in `st`, and the corresponding field values are block parameterizations as described in `blk_param`.

More About

Tuned Blocks

Tuned blocks, used by the `slTuner` interface, identify blocks in a Simulink model whose parameters are to be tuned to satisfy tuning goals. You can tune most Simulink blocks that represent linear

elements such as gains, transfer functions, or state-space models. (For the complete list of blocks that support tuning, see “How Tuned Simulink Blocks Are Parameterized” on page 10-27). You can also tune more complex blocks such as SubSystem or S-Function blocks by specifying an equivalent tunable linear model.

Use tuning commands such as `systune` to tune the parameters of tuned blocks.

You must specify tuned blocks (for example, `C1` and `C2`) when you create an `slTuner` interface.

```
st = slTuner('scdcascade',{'C1','C2'})
```

You can modify the list of tuned blocks using `addBlock` and `removeBlock`.

To interact with the tuned blocks use:

- `getBlockParam`, `getBlockValue`, and `getTunedValue` to access the tuned block parameterizations and their current values.
- `setBlockParam`, `setBlockValue`, and `setTunedValue` to modify the tuned block parameterizations and their values.
- `writeBlockValue` to update the blocks in a Simulink model with the current values of the tuned block parameterizations.

Tuned Variables

Within an `slTuner` interface, tuned variables are any Control Design Blocks involved in the parameterization of a tuned Simulink block, either directly or through a generalized parametric model. Tuned variables are the parameters manipulated by tuning commands such as `systune`.

For Simulink blocks parameterized by a generalized model or a tunable surface:

- `getBlockValue` provides access to the overall value of the block parameterization. To access the values of the tuned variables within the block parameterization, use `getTunedValue`.
- `setBlockValue` cannot be used to modify the block value. To modify the values of tuned variables within the block parameterization, use `setTunedValue`.

For Simulink blocks parameterized by a Control Design Block, the block itself is the tuned variable. To modify the block value, you can use either `setBlockValue` or `setTunedValue`. Similarly, you can retrieve the block value using either `getBlockValue` or `getTunedValue`.

See Also

`slTuner` | `setBlockParam` | `getBlockValue` | `getTunedValue` | `genss` | `tunablePID`

Topics

“How Tuned Simulink Blocks Are Parameterized”

Introduced in R2011b

getBlockRateConversion

Get rate conversion settings for tuned block in `slTuner` interface

Syntax

```
method = getBlockRateConversion(st,blk)
[method,pwf] = getBlockRateConversion(st,blk)

[IF,DF] = getBlockRateConversion(st,blk)
```

Description

When you use `systune` with Simulink, tuning is performed at the sampling rate specified by the `Ts` property of the `slTuner` interface. When you use `writeBlockValue` to write tuned parameters back to the Simulink model, each tuned block value is automatically converted from the sample time used for tuning, to the sample time of the Simulink block. The rate conversion method associated with each tuned block specifies how this resampling operation should be performed. Use `getBlockRateConversion` to query the block conversion rate and use `setBlockRateConversion` to modify it.

`method = getBlockRateConversion(st,blk)` returns the rate conversion method associated with the tuned block on page 18-314, `blk`.

`[method,pwf] = getBlockRateConversion(st,blk)` also returns the prewarp frequency. When `method` is not `'tustin'`, the prewarp frequency is always 0.

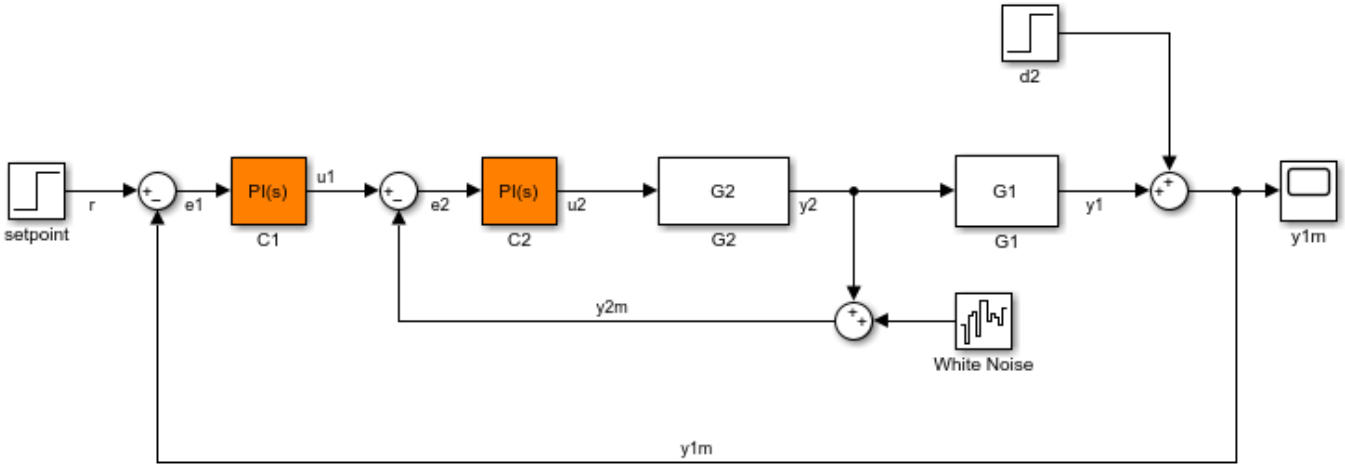
`[IF,DF] = getBlockRateConversion(st,blk)` returns the discretization methods for the integrator and derivative filter terms when `blk` is a PID Controller block.

Examples

Get Rate Conversion Settings of Tuned PID Block

Create an `slTuner` interface for the Simulink model `scdcascade`. Examine the block rate conversion settings of one of the tuned blocks.

```
open_system('scdcascade')
st = slTuner('scdcascade',{'C1','C2'});
```




```
[IF,DF] = getBlockRateConversion(st,'C1')
```

```
IF =

    'Trapezoidal'

DF =

    'Trapezoidal'
```

`C1` is a PID block. Therefore, its rate-conversion settings are expressed in terms of integrator and derivative filter methods. For a continuous-time PID block, the rate-conversion methods are set to `Trapezoidal` by default. To override this setting, use `setBlockRateConversion`.

Input Arguments

`st` — Interface for tuning control systems modeled in Simulink

`slTuner` interface

Interface for tuning control systems modeled in Simulink, specified as an `slTuner` interface.

`blk` — Block

character vector | string

Block in the list of tuned blocks for `st`, specified as a character vector or string. You can specify the full block path or any portion of the block path that uniquely identifies the block among the other tuned blocks of `st`.

Example: `blk = 'scdcascade/C1'`, `blk = "C1"`

Output Arguments

`method` — Rate conversion method

`'zoh'` | `'foh'` | `'tustin'` | `'matched'`

Rate conversion method associated with `blk`, returned as one of the following:

- `'zoh'` — Zero-order hold on the inputs
- `'foh` — Linear interpolation of inputs
- `'tustin'` — Bilinear (Tustin) approximation
- `'matched'` — Matched pole-zero method (for SISO blocks only)

`pwf` — Prewarp frequency for Tustin method
positive scalar

Prewarp frequency for the Tustin method, returned as a positive scalar.

If the rate conversion method associated with `blk` is zero-order hold or Tustin without prewarp, then `pwf` is 0.

`IF,DF` — Integrator and filter methods
`'ForwardEuler'` | `'BackwardEuler'` | `'Trapezoidal'`

Integrator and filter methods for rate conversion of PID Controller block, each returned as `'ForwardEuler'`, `'BackwardEuler'`, or `'Trapezoidal'`. For continuous-time PID blocks, the default methods are `'Trapezoidal'` for both integrator and derivative filter. For discrete-time PID blocks, `IF` and `DF` are determined by the **Integrator method** and **Filter method** settings in the Simulink block. See the Discrete PID Controller and `pid` reference pages for more details about integrator and filter methods.

More About

Tuned Blocks

Tuned blocks, used by the `slTuner` interface, identify blocks in a Simulink model whose parameters are to be tuned to satisfy tuning goals. You can tune most Simulink blocks that represent linear elements such as gains, transfer functions, or state-space models. (For the complete list of blocks that support tuning, see “How Tuned Simulink Blocks Are Parameterized” on page 10-27). You can also tune more complex blocks such as SubSystem or S-Function blocks by specifying an equivalent tunable linear model.

Use tuning commands such as `systune` to tune the parameters of tuned blocks.

You must specify tuned blocks (for example, `C1` and `C2`) when you create an `slTuner` interface.

```
st = slTuner('scdcascade',{'C1','C2'})
```

You can modify the list of tuned blocks using `addBlock` and `removeBlock`.

To interact with the tuned blocks use:

- `getBlockParam`, `getBlockValue`, and `getTunedValue` to access the tuned block parameterizations and their current values.
- `setBlockParam`, `setBlockValue`, and `setTunedValue` to modify the tuned block parameterizations and their values.
- `writeBlockValue` to update the blocks in a Simulink model with the current values of the tuned block parameterizations.

See Also

`slTuner` | `setBlockRateConversion` | `writeBlockValue`

Topics

“Tuning of a Digital Motion Control System”
“Continuous-Discrete Conversion Methods”

Introduced in R2014a

getBlockValue

Get current value of tuned block parameterization in `slTuner` interface

Syntax

```
value = getBlockValue(st,blk)
[val1,val2,...] = getBlockValue(st,blk1,blk2,...)

S = getBlockValue(st)
```

Description

`getBlockValue` lets you access the current value of the parameterization of a tuned block on page 18-320 in an `slTuner` interface.

An `slTuner` interface parameterizes each tuned Simulink block as a Control Design Block, or a generalized parametric model of type `genmat` or `genss`. This parameterization specifies the tuned variables on page 18-320 for commands such as `systune`.

`value = getBlockValue(st,blk)` returns the current value of the parameterization of a tunable block, `blk`, in an `slTuner` interface.

`[val1,val2,...] = getBlockValue(st,blk1,blk2,...)` returns the current values of the parameterizations of one or more tuned blocks of `st`.

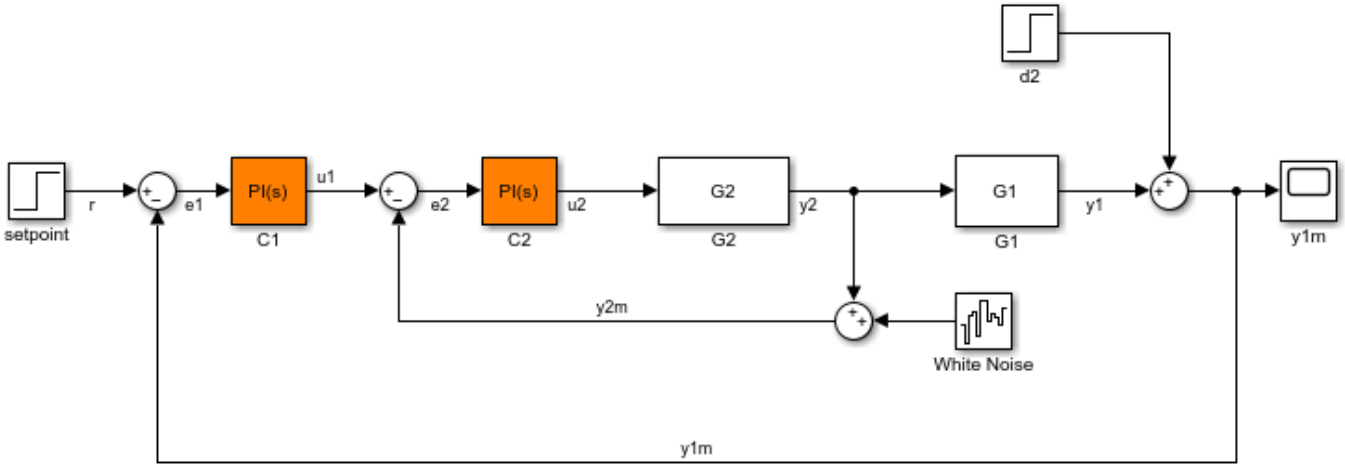
`S = getBlockValue(st)` returns a structure containing the current values of the parameterizations of all tuned blocks of `st`.

Examples

Get Current Value of Tuned Block Parameterization

Create an `slTuner` interface for the `scdcascade` model.

```
open_system('scdcascade')
st = slTuner('scdcascade',{'C1','C2'});
```



Examine the current parameterization value of one of the tuned blocks.

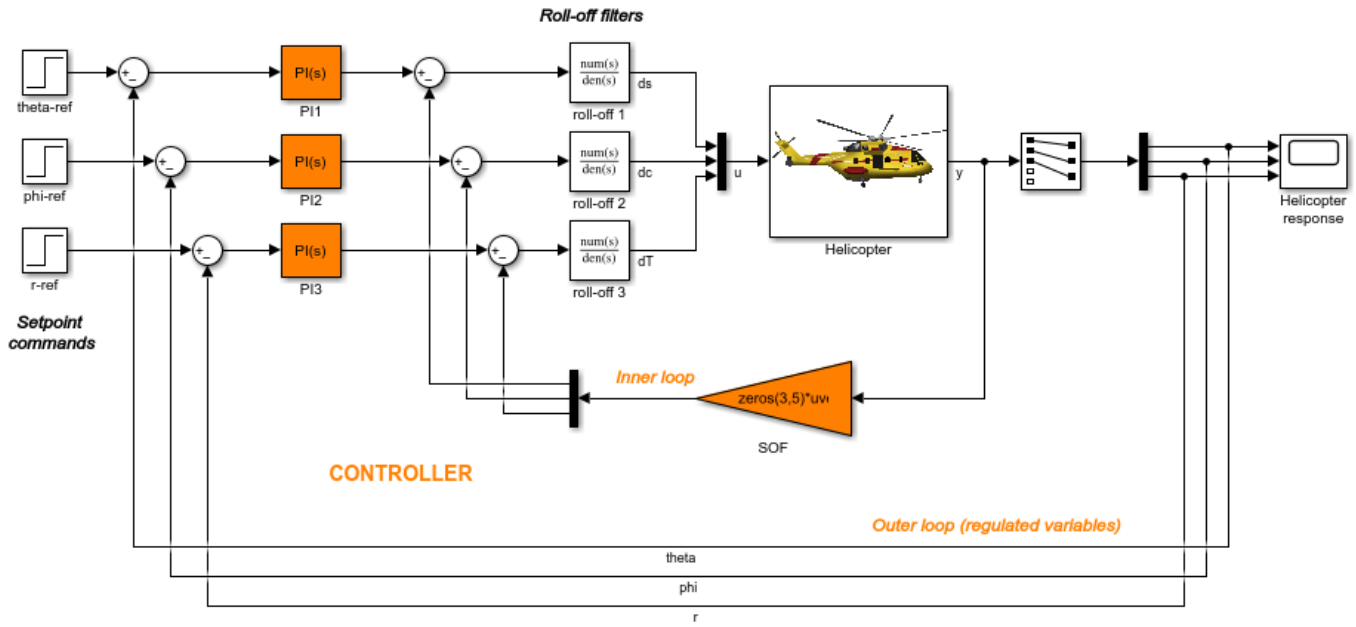
```
val = getBlockValue(st,'C1')
```

```
val =

             1
  Kp + Ki * ---
             s

  with Kp = 0.158, Ki = 0.042

Name: C1
Continuous-time PI controller in parallel form.
```

Get Current Values of Multiple Tuned Block Parameterizations

Create an `slTuner` interface for the `scdhelicopter` model.

```
open_system('scdhelicopter')
st = slTuner('scdhelicopter',{'PI1','PI2','PI3','SOF'});
```



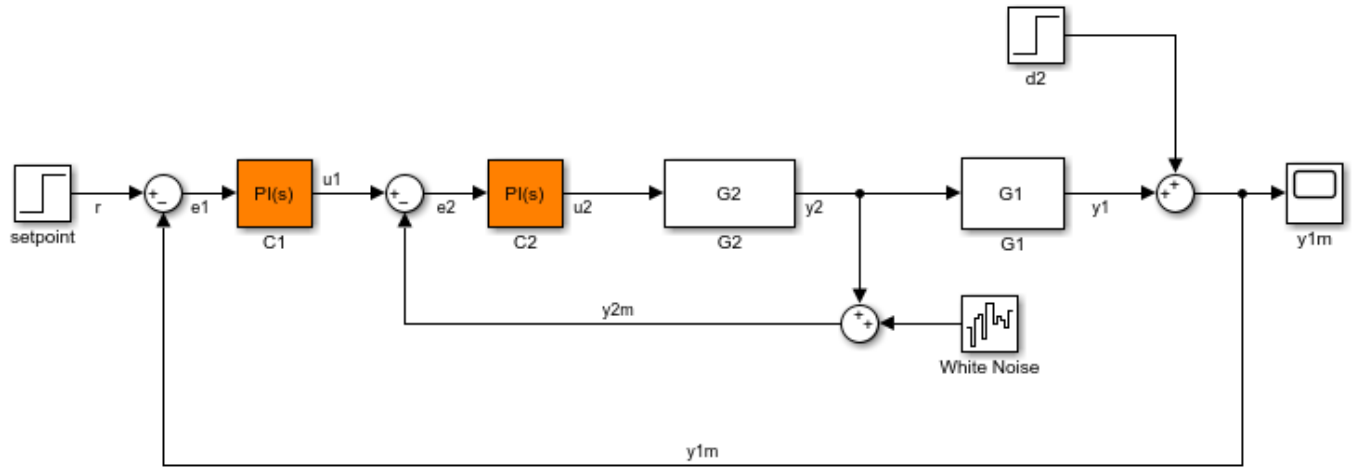
Retrieve the values of parameterizations for the PI controller blocks in the model.

```
[valPI1,valPI2,valPI3] = getBlockParam(st,'PI1','PI2','PI3');
```

Get Current Values of All Tuned Block Parameterizations

Create an `slTuner` interface for the `scdcascade` model.

```
open_system('scdcascade')
st = slTuner('scdcascade',{'C1','C2'});
```



Retrieve the parameterization values for both tuned blocks in `st`.

```
blockValues = getBlockValue(st)
```

```
blockValues =

  struct with fields:

    C1: [1x1 pid]
    C2: [1x1 pid]
```

`blockValues` is a structure with field names corresponding to the names of the tunable blocks in `st`. The field values of `blockValues` are `pid` models, because `C1` and `C2` are both PID Controller blocks.

Input Arguments

`st` — Interface for tuning control systems modeled in Simulink

`slTuner` interface

Interface for tuning control systems modeled in Simulink, specified as an `slTuner` interface.

`blk` — Block

character vector | string

Block in the list of tuned blocks for `st`, specified as a character vector or string. You can specify the full block path or any portion of the block path that uniquely identifies the block among the other tuned blocks of `st`.

Example: `blk = 'scdcascade/C1'`, `blk = "C1"`

Output Arguments

`value` — Current value of block parameterization

numeric LTI model

Current value of block parameterization, returned as a numeric LTI model, such as `pid`, `ss`, or `tf`.

When the tuning results have not been applied to the Simulink model using `writeBlockValue`, the value returned by `getBlockValue` can differ from the actual Simulink block value.

Note Use `writeBlockValue` to align the block parameterization values with the actual block values in the Simulink model.

`S` — Current values of all block parameterizations

structure

Current values of all block parameterizations in `st`, returned as a structure. The names of the fields in `S` are the names of the tuned blocks in `st`, and the field values are the corresponding numeric LTI models.

You can use this structure to transfer the tuned values from one `slTuner` interface to another `slTuner` interface with the same tuned block parameterizations.

```
S = getBlockValue(st1);
setBlockValue(st2,S);
```

More About

Tuned Blocks

Tuned blocks, used by the `slTuner` interface, identify blocks in a Simulink model whose parameters are to be tuned to satisfy tuning goals. You can tune most Simulink blocks that represent linear elements such as gains, transfer functions, or state-space models. (For the complete list of blocks that support tuning, see “How Tuned Simulink Blocks Are Parameterized” on page 10-27). You can also tune more complex blocks such as SubSystem or S-Function blocks by specifying an equivalent tunable linear model.

Use tuning commands such as `systune` to tune the parameters of tuned blocks.

You must specify tuned blocks (for example, `C1` and `C2`) when you create an `slTuner` interface.

```
st = slTuner('scdcascade',{'C1','C2'})
```

You can modify the list of tuned blocks using `addBlock` and `removeBlock`.

To interact with the tuned blocks use:

- `getBlockParam`, `getBlockValue`, and `getTunedValue` to access the tuned block parameterizations and their current values.
- `setBlockParam`, `setBlockValue`, and `setTunedValue` to modify the tuned block parameterizations and their values.
- `writeBlockValue` to update the blocks in a Simulink model with the current values of the tuned block parameterizations.

Tuned Variables

Within an `slTuner` interface, tuned variables are any Control Design Blocks involved in the parameterization of a tuned Simulink block, either directly or through a generalized parametric model. Tuned variables are the parameters manipulated by tuning commands such as `systune`.

For Simulink blocks parameterized by a generalized model or a tunable surface:

- `getBlockValue` provides access to the overall value of the block parameterization. To access the values of the tuned variables within the block parameterization, use `getTunedValue`.
- `setBlockValue` cannot be used to modify the block value. To modify the values of tuned variables within the block parameterization, use `setTunedValue`.

For Simulink blocks parameterized by a Control Design Block, the block itself is the tuned variable. To modify the block value, you can use either `setBlockValue` or `setTunedValue`. Similarly, you can retrieve the block value using either `getBlockValue` or `getTunedValue`.

See Also

`slTuner` | `setBlockValue` | `getBlockParam` | `getTunedValue`

Topics

“How Tuned Simulink Blocks Are Parameterized”

Introduced in R2011b

getTunedValue

Get current value of tuned variable in `slTuner` interface

Syntax

```
value = getTunedValue(st,var)
[value1,value2,...] = getTunedValue(st,var1,var2,...)

S = getTunedValue(st)
```

Description

`getTunedValue` lets you access the current value of a tuned variable on page 18-326 within an `slTuner` interface.

An `slTuner` interface parameterizes each tuned block on page 18-325 as a Control Design Block, or a generalized parametric model of type `genmat` or `genss`. This parameterization specifies the tuned variables for commands such as `systune`.

`value = getTunedValue(st,var)` returns the current value of the tuned variable, `var`, in the `slTuner` interface, `st`.

`[value1,value2,...] = getTunedValue(st,var1,var2,...)` returns the current values of multiple tuned variables.

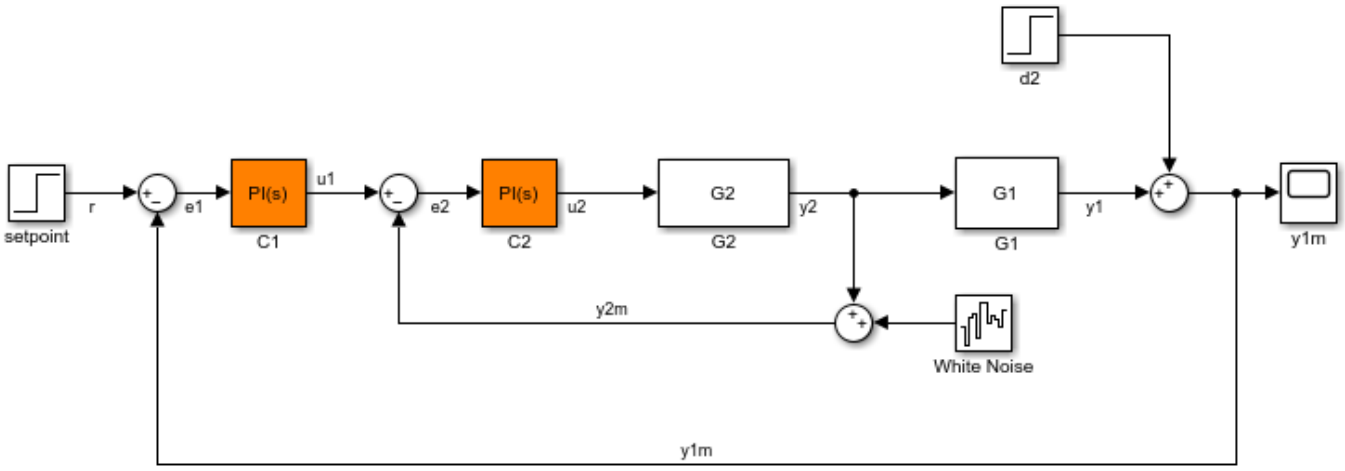
`S = getTunedValue(st)` returns a structure containing the current values of all tuned variables in `st`.

Examples

Query Value of Single Tunable Element within Custom Parameterization

Create an `slTuner` interface for the `scdcascade` model.

```
open_system('scdcascade')
st = slTuner('scdcascade',{'C1','C2'});
```



Set a custom parameterization for one of the tunable blocks.

```
C1CustParam = realp('Kp',1) + tf(1,[1 0]) * realp('Ki',1);
setBlockParam(st,'C1',C1CustParam);
```

These commands set the parameterization of the `C1` controller block to a generalized state-space (`genss`) model containing two tunable parameters, `Ki` and `Kp`.

Typically, you would use a tuning command such as `systune` to tune the values of the parameters in the custom parameterization.

After tuning, use `getTunedValue` to query the tuned value of `Ki`.

```
KiTuned = getTunedValue(st,'Ki')
```

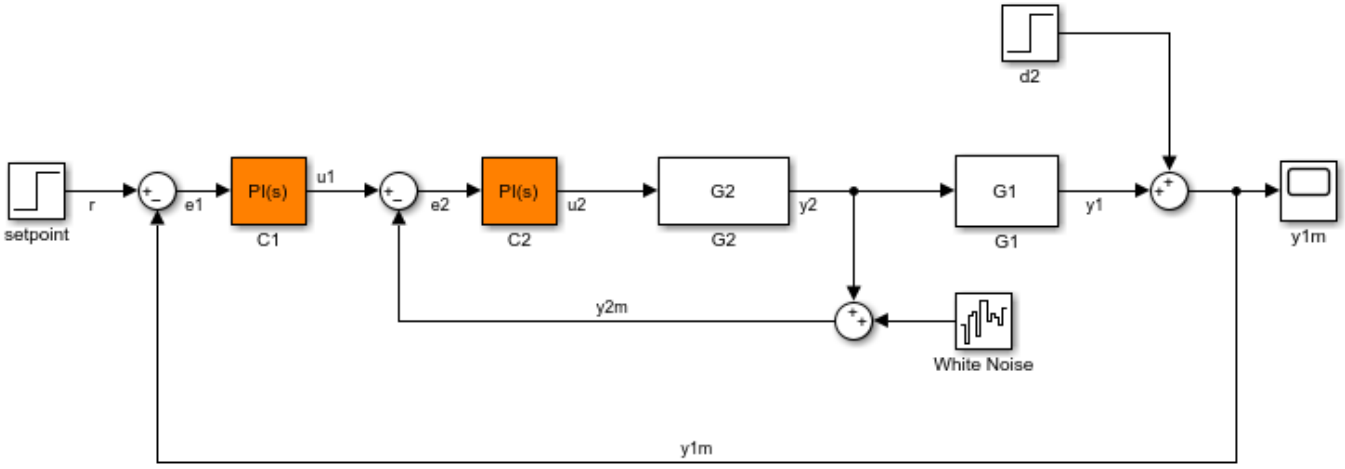
```
KiTuned =

     1
```

To query the value of the tuned block as a whole, `C1`, use `getBlockValue`.

Query Value of Multiple Tunable Elements within Custom Parameterization

Create an `slTuner` interface for the `scdcascade` model.

```
open_system('scdcascade')
st = slTuner('scdcascade',{'C1','C2'});
```



Set a custom parameterization for one of the tunable blocks.

```
C1CustParam = realp('Kp',1) + tf(1,[1 0]) * realp('Ki',1);
setBlockParam(st,'C1',C1CustParam);
```

These commands set the parameterization of the `C1` controller block to a generalized state-space (`genss`) model containing tunable parameters `Kp` and `Ki`.

Typically, you would use a tuning command such as `systune` to tune the values of the parameters in the custom parameterization.

After tuning, use `getTunedValue` to query the tuned values of both `Kp` and `Ki`.

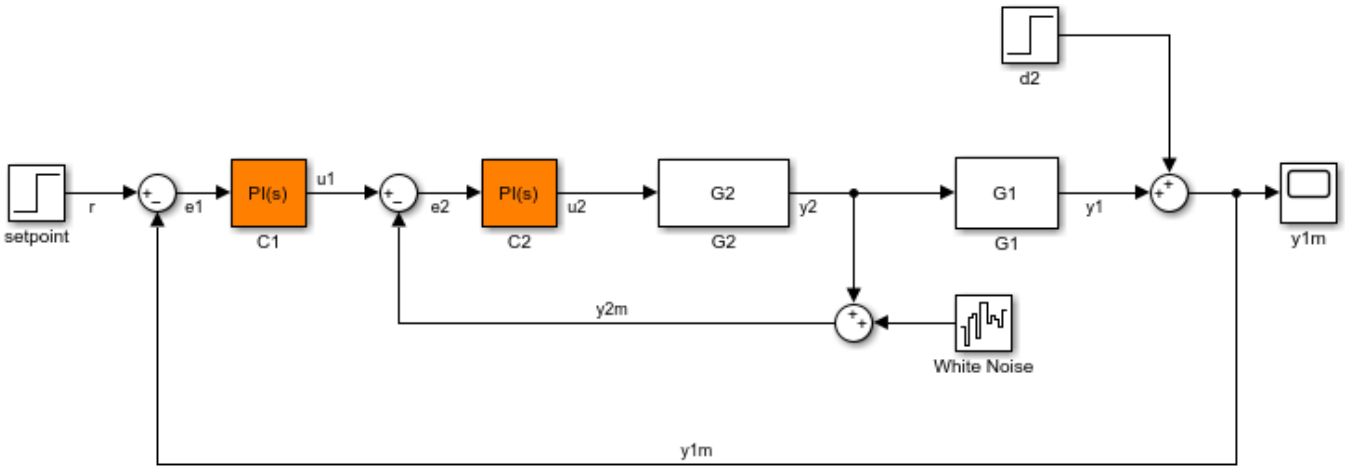
```
[KiTuned,KpTuned] = getTunedValue(st,'Ki','Kp')
```

```
KiTuned =

     1


KpTuned =

     1
```

Query Value of All Tuned Elements in slTuner Interface with Custom Parameterizations

Create an `slTuner` interface for the `scdcascade` model.

```
open_system('scdcascade')
st = slTuner('scdcascade',{'C1','C2'});
```



Set a custom parameterization for tuned block `C1`.

```
C1CustParam = realp('Kp',1) + tf(1,[1 0]) * realp('Ki',1);
setBlockParam(st,'C1',C1CustParam);
```

Typically, you would use a tuning command such as `systune` to tune the values of the parameters in the custom parameterization.

After tuning, use `getTunedValue` to query the tuned values of the parameterizations of all the tuned blocks in `st`.

```
S = getTunedValue(st)
```

```
S =

  struct with fields:

    C2: [1x1 pid]
    Ki: 1
    Kp: 1
```

The tuned values are returned in a structure that contains fields for:

- The tuned block, `C2`, which is parameterized as a Control Design Block.
- The tunable elements, `Kp` and `Ki`, within block `C2`, which is parameterized as a custom `genss` model.

Input Arguments

`st` — Interface for tuning control systems modeled in Simulink

`slTuner` interface

Interface for tuning control systems modeled in Simulink, specified as an `slTuner` interface.

`var` — Tuned variable

character vector | string

Tuned variable within `st`, specified as a character vector or string. A tuned variable is any Control Design Block, such `realp`, `tunableSS`, or `tunableGain`, involved in the parameterization of a tuned Simulink block, either directly or through a generalized parametric model. To get a list of all tuned variables within `st`, use `getTunedValue(st)`.

`var` can refer to the following:

- For a block parameterized by a Control Design Block, the name of the block. For example, if the parameterization of the block is

  ```
  C = tunableSS('C')
  ```

  then set `var = 'C'`.
- For a block parameterized by a `genmat`/`genss` model, `M`, the name of any Control Design Block listed in `M.Blocks`. For example, if the parameterization of the block is

  ```
  a = realp('a',1);
  C = tf(a,[1 a]);
  ```

  then set `var = 'a'`.

Output Arguments

`value` — Current value of tuned variable

numeric scalar | numeric array | state-space model

Current value of tuned variable in `st`, returned as a numeric scalar or array or a state-space model. When the tuning results have not been applied to the Simulink model using `writeBlockValue`, the value returned by `getTunedValue` can differ from the Simulink block value.

Note Use `writeBlockValue` to align the block parameterization values with the actual block values in the Simulink model.

`S` — Current values of all tuned variables

structure

Current values of all tuned variables in `st`, returned as a structure. The names of the fields in `S` are the names of the tuned variables in `st`, and the field values are the corresponding numeric scalars or arrays.

You can use this structure to transfer the tuned variable values from one `slTuner` interface to another `slTuner` interface with the same tuned variables, as follows:

```
S = getTunedValue(st1);
setTunedValue(st2,S);
```

More About

Tuned Blocks

Tuned blocks, used by the `slTuner` interface, identify blocks in a Simulink model whose parameters are to be tuned to satisfy tuning goals. You can tune most Simulink blocks that represent linear

elements such as gains, transfer functions, or state-space models. (For the complete list of blocks that support tuning, see “How Tuned Simulink Blocks Are Parameterized” on page 10-27). You can also tune more complex blocks such as SubSystem or S-Function blocks by specifying an equivalent tunable linear model.

Use tuning commands such as `systune` to tune the parameters of tuned blocks.

You must specify tuned blocks (for example, `C1` and `C2`) when you create an `slTuner` interface.

```
st = slTuner('scdcascade',{'C1','C2'})
```

You can modify the list of tuned blocks using `addBlock` and `removeBlock`.

To interact with the tuned blocks use:

- `getBlockParam`, `getBlockValue`, and `getTunedValue` to access the tuned block parameterizations and their current values.
- `setBlockParam`, `setBlockValue`, and `setTunedValue` to modify the tuned block parameterizations and their values.
- `writeBlockValue` to update the blocks in a Simulink model with the current values of the tuned block parameterizations.

Tuned Variables

Within an `slTuner` interface, tuned variables are any Control Design Blocks involved in the parameterization of a tuned Simulink block, either directly or through a generalized parametric model. Tuned variables are the parameters manipulated by tuning commands such as `systune`.

For Simulink blocks parameterized by a generalized model or a tunable surface:

- `getBlockValue` provides access to the overall value of the block parameterization. To access the values of the tuned variables within the block parameterization, use `getTunedValue`.
- `setBlockValue` cannot be used to modify the block value. To modify the values of tuned variables within the block parameterization, use `setTunedValue`.

For Simulink blocks parameterized by a Control Design Block, the block itself is the tuned variable. To modify the block value, you can use either `setBlockValue` or `setTunedValue`. Similarly, you can retrieve the block value using either `getBlockValue` or `getTunedValue`.

See Also

`slTuner` | `setTunedValue` | `getBlockParam` | `getBlockValue` | `tunableSurface`

Topics

“How Tuned Simulink Blocks Are Parameterized”

Introduced in R2015b

looptune

Tune MIMO feedback loops in Simulink using `slTuner` interface

Syntax

```
[st,gam,info] = looptune(st0,controls,measurements,wc)
[st,gam,info] = looptune(st0,controls,measurements,wc,req1,...,reqN)
[st,gam,info] = looptune( ___ ,opt)
```

Description

`[st,gam,info] = looptune(st0,controls,measurements,wc)` tunes the free parameters on page 18-332 of the control system of the Simulink model associated with the `slTuner` interface, `st0`, to achieve the following goals:

- Bandwidth — Gain crossover for each loop falls in the frequency interval `wc`
- Performance — Integral action at frequencies below `wc`
- Robustness — Adequate stability margins and gain roll-off at frequencies above `wc`

`controls` and `measurements` specify the controller output signals and measurement signals that are subject to the goals, respectively. `st` is the updated `slTuner` interface, `gam` indicates the measure of success in satisfying the goals, and `info` gives details regarding the optimization run.

Tuning is performed at the sample time specified by the `Ts` property of `st0`. For tuning algorithm details, see “Algorithms” on page 18-332.

`[st,gam,info] = looptune(st0,controls,measurements,wc,req1,...,reqN)` tunes the feedback loop to meet additional goals specified in one or more tuning goal objects, `req`. Omit `wc` to drop the default loop shaping goal associated with `wc`. Note that the stability margin goals remain in force.

`[st,gam,info] = looptune( ___ ,opt)` specifies further options, including target gain and phase margins, number of runs, and computation options for the tuning algorithm. Use `looptuneOptions` to create `opt`.

If you specify multiple runs using the `RandomStarts` property of `opt`, `looptune` performs only as many runs required to achieve the target objective value of 1. Note that all tuning goals should be normalized so that a maximum value of 1 means that all design goals are met.

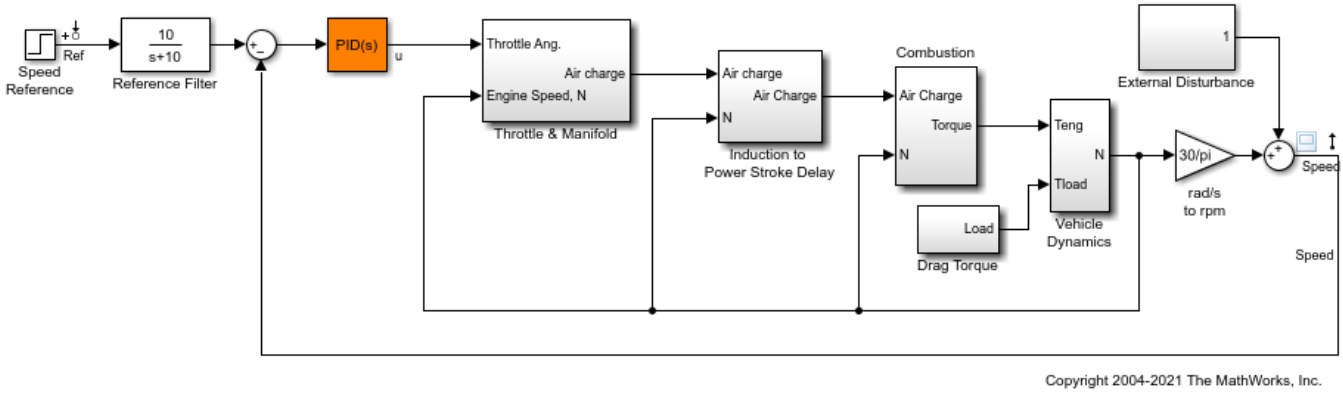
Examples

Tune Controller to Achieve Specified Bandwidth

Tune the PID Controller in the `rct_engine_speed` model to achieve the specified bandwidth.

Open the Simulink model.

```
mdl = 'rct_engine_speed';
open_system(mdl);
```



Create an `slTuner` interface for the model.

```
st0 = slTuner(mdl,'PID Controller');
```

Add the PID Controller output, `u`, as an analysis point to `st0`.

```
addPoint(st0,'u');
```

Based on first-order characteristics, the crossover frequency should exceed 1 rad/s for the closed-loop response to settle in less than 5 seconds. So, tune the PID loop using 1 rad/s as the target 0 dB crossover frequency.

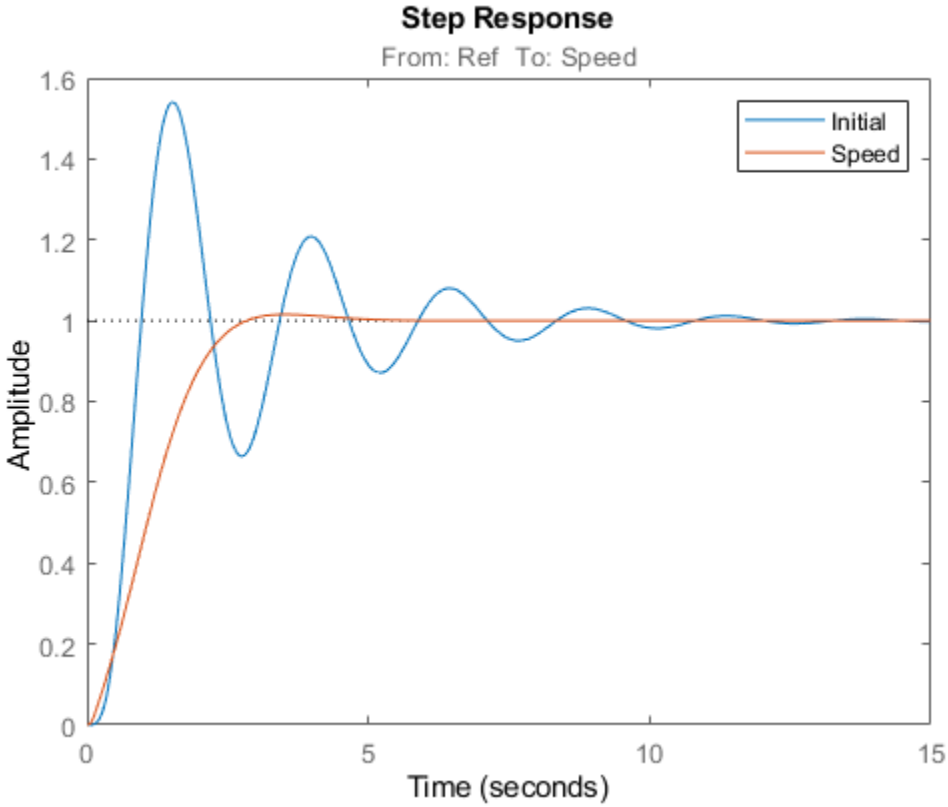
```
wc = 1;
st = looptune(st0,'u','Speed',wc);
```

```
Final: Peak gain = 0.979, Iterations = 4
Achieved target gain value TargetGain=1.
```

In the call to `looptune`, `'u'` specifies the control signal, and `'Speed'` specifies the measured signal.

Compare the tuned and initial response.

```
stepplot(getIOTransfer(st0,'Ref','Speed'),getIOTransfer(st,'Ref','Speed'));
legend('Initial','Speed');
```



View the tuned block value.

```
showTunable(st)
```

```
Block 1: rct_engine_speed/PID Controller =

             1             s
  Kp + Ki * --- + Kd * --------
             s          Tf*s+1

  with Kp = 0.00062, Ki = 0.00303, Kd = 0.000168, Tf = 0.01

Name: PID_Controller
Continuous-time PIDF controller in parallel form.
```

To write the tuned values back to the Simulink model, use `writeBlockValue`.

Input Arguments

`st0` — Interface for tuning control systems modeled in Simulink

`slTuner` interface

Interface for tuning control systems modeled in Simulink, specified as an `slTuner` interface.

`controls` — Controller output
character vector | cell array of character vectors

Controller output name, specified as one of the following:

- Character vector — Name of an analysis point of `st0`.

  You can specify the full name or any portion of the name that uniquely identifies the analysis point among the other analysis points of `st0`.

  For example, `'u'`.
- Cell array of character vectors — Multiple analysis point names.

  For example, `{'u','y'}`.

`measurements` — Measurement
character vector | cell array of character vectors

Measurement signal name, specified as one of the following:

- Character vector — Name of an analysis point of `st0`.

  You can specify the full name or any portion of the name that uniquely identifies the analysis point among the other analysis points of `st0`.

  For example, `'u'`.
- Cell array of character vector — Multiple analysis point names.

  For example, `{'u','y'}`.

`wc` — Target crossover region
`[wcmin,wcmax]` | positive scalar

Target crossover region, specified as one of the following:

- `[wcmin,wcmax]` — `looptune` attempts to tune all loops in the control system so that the open-loop gain crosses 0 dB within the target crossover region.
- Positive scalar — Specifies the target crossover region as `[wc/10^0.1,wc*10^0.1]` or `wc` +/- 0.1 decades.

Specify `wc` in the working time units, that is, the time units of the model.

`req1,...,reqN` — Design goals
`TuningGoal` objects

Design goals, specified as one or more `TuningGoal` objects.

For a complete list of the design goals you can specify, see “Tuning Goals”.

`opt` — Tuning algorithm options
options set created using `looptuneOptions`

Tuning algorithm options, specified as an options set created using `looptuneOptions`.

Available options include:

- Number of additional optimizations to run starting from random initial values of the free parameters
- Tolerance for terminating the optimization
- Flag for using parallel processing
- Specification of target gain and phase margin

Output Arguments

`st` — Tuned interface
`slTuner` interface

Tuned interface, returned as an `slTuner` interface.

`gam` — Parameter indicating degree of success at meeting all tuning constraints
scalar

Parameter indicating degree of success at meeting all tuning constraints, returned as a scalar.

A value of `gam <= 1` indicates that all goals are satisfied. A value of `gam >> 1` indicates failure to meet at least one requirement. Use `loopview` to visualize the tuned result and identify the unsatisfied requirement.

For best results, use the `RandomStart` option in `looptuneOptions` to obtain several minimization runs. Setting `RandomStart` to an integer `N > 0` causes `looptune` to run the optimization `N` additional times, beginning from parameter values it chooses randomly. You can examine `gam` for each run to help identify an optimization result that meets your design goals.

`info` — Detailed information about each optimization run
structure

Detailed information about each optimization run, returned as a structure with the following fields:

`Di,Do` — Optimal input and output scalings
state-space models

Optimal input and output scalings, return as state-space models.

The scaled plant is given by `Do\G*Di`.

`Specs` — Design goals used for tuning
vector of `TuningGoal` requirement objects

Design goals used for tuning, returned as a vector of `TuningGoal` requirement objects.

`Runs` — Detailed information about each optimization run
structure

Detailed information about each optimization run, returned as a structure. For details, see “Algorithms” on page 18-332.

The contents of `Runs` are the `info` output of the call to `systune` performed by `looptune`. For information about the fields of `Runs`, see the `info` output argument description on the `systune` reference page.

More About

Tuned Blocks

Tuned blocks, used by the `slTuner` interface, identify blocks in a Simulink model whose parameters are to be tuned to satisfy tuning goals. You can tune most Simulink blocks that represent linear elements such as gains, transfer functions, or state-space models. (For the complete list of blocks that support tuning, see “How Tuned Simulink Blocks Are Parameterized” on page 10-27). You can also tune more complex blocks such as SubSystem or S-Function blocks by specifying an equivalent tunable linear model.

Use tuning commands such as `systune` to tune the parameters of tuned blocks.

You must specify tuned blocks (for example, `C1` and `C2`) when you create an `slTuner` interface.

```
st = slTuner('scdcascade',{'C1','C2'})
```

You can modify the list of tuned blocks using `addBlock` and `removeBlock`.

To interact with the tuned blocks use:

- `getBlockParam`, `getBlockValue`, and `getTunedValue` to access the tuned block parameterizations and their current values.
- `setBlockParam`, `setBlockValue`, and `setTunedValue` to modify the tuned block parameterizations and their values.
- `writeBlockValue` to update the blocks in a Simulink model with the current values of the tuned block parameterizations.

Algorithms

`looptune` automatically converts target bandwidth, performance goals, and additional design goals into weighting functions that express the goals as an H_∞ optimization problem. `looptune` then uses `systune` to optimize tunable parameters to minimize the H_∞ norm.

For information about the optimization algorithms, see [1].

`looptune` computes the H_∞ norm using the algorithm of [2] and structure-preserving eigensolvers from the SLICOT library. For more information about the SLICOT library, see http://slicot.org.

References

[1] P. Apkarian and D. Noll, "Nonsmooth H-infinity Synthesis." *IEEE Transactions on Automatic Control*, Vol. 51, Number 1, 2006, pp. 71–86.

[2] Bruinsma, N.A., and M. Steinbuch. "A Fast Algorithm to Compute the H_∞ Norm of a Transfer Function Matrix." *Systems & Control Letters*, 14, no.4 (April 1990): 287–93.

Extended Capabilities

Automatic Parallel Support

Accelerate code by automatically running computation in parallel using Parallel Computing Toolbox™.

To run in parallel, set `'UseParallel'` to `true` using `looptuneOptions`.

See Also

`looptune (for genss)` | `looptuneOptions` | `TuningGoal.Tracking` | `TuningGoal.Gain` | `TuningGoal.Margins` | `slTuner` | `addPoint` | `getIOTransfer` | `getLoopTransfer` | `writeBlockValue` | `systune` | `hinfstruct`

Topics

“Tune Control Systems in Simulink”
“Decoupling Controller for a Distillation Column”
“Tuning of a Digital Motion Control System”
“Tuning of a Two-Loop Autopilot”
“Structure of Control System for Tuning With looptune”
“Set Up Your Control System for Tuning with looptune”

Introduced in R2014a

looptuneSetup

Construct tuning setup for `looptune` to tuning setup for `systune` using `slTuner` interface

Syntax

```
[st0,SoftReqs,HardReqs,sysopt] = looptuneSetup(looptuneInputs)
```

Description

`[st0,SoftReqs,HardReqs,sysopt] = looptuneSetup(looptuneInputs)` converts a tuning setup for `looptune` into an equivalent tuning setup for `systune`. The argument `looptuneInputs` is a sequence of input arguments for `looptune` that specifies the tuning setup. For example,

```
[st0,SoftReqs,HardReqs,sysopt] = looptuneSetup(st0,wc,Req1,Req2,loopopt)
```

generates a set of arguments such that `looptune(st0,wc,Req1,Req2,loopopt)` and `systune(st0,SoftReqs,HardReqs,sysopt)` produce the same results.

Use this command to take advantage of additional flexibility that `systune` offers relative to `looptune`. For example, `looptune` requires that you tune all channels of a MIMO feedback loop to the same target bandwidth. Converting to `systune` allows you to specify different crossover frequencies and loop shapes for each loop in your control system. Also, `looptune` treats all tuning requirements as soft requirements, optimizing them, but not requiring that any constraint be exactly met. Converting to `systune` allows you to enforce some tuning requirements as hard constraints, while treating others as soft requirements.

You can also use this command to probe into the tuning requirements enforced by `looptune`.

Examples

Convert looptune Problem into systune Problem

Convert a set of `looptune` inputs for tuning a Simulink model into an equivalent set of inputs for `systune`.

Suppose you have created and configured an `slTuner` interface, `st0`, for tuning with `looptune`. Suppose also that you used `looptune` to tune the feedback loop defined in `st0` to within a bandwidth of `wc = [wmin,wmax]`. Convert these variables into a form that allows you to use `systune` for further tuning.

```
[st0,SoftReqs,HardReqs,sysopt] = looptuneSetup(st0,wc,controls,measurements);
```

The command returns the closed-loop system and tuning requirements for the equivalent `systune` command, `systune(st0,SoftReqs,HardReqs,sysopt)`. The arrays `SoftReqs` and `HardReqs` contain the tuning requirements implicitly imposed by `looptune`. These requirements enforce the target bandwidth and default stability margins of `looptune`.

If you used additional tuning requirements when tuning the system with `looptune`, add them to the input list of `looptuneSetup`. For example, suppose you used a `TuningGoal.Tracking`

requirement, `Req1`, and a `TuningGoal.Rejection` requirement, `Req2`. Suppose also that you set algorithm options for `looptune` using `looptuneOptions`. Incorporate these requirements and options into the equivalent `systune` command.

```
[st0,SoftReqs,HardReqs,sysopt] = looptuneSetup(st0,wc,Req1,Req2,loopopt);
```

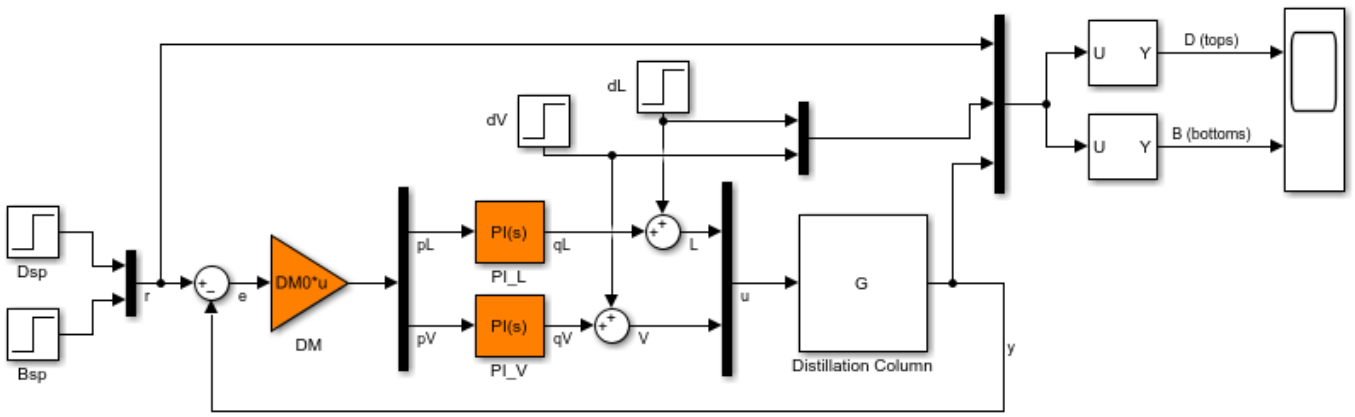
The resulting arguments allow you to construct the equivalent tuning problem for `systune`.

Convert Distillation Column Problem for Tuning With systune

Set up the control system of the Simulink® model `rct_distillation` for tuning with `looptune`. Then, convert the setup to a `systune` problem, and examine the resulting arguments. The results reflect the tuning requirements implicitly enforced when tuning with `looptune`.

Create an `slTuner` interface to the Simulink model, and specify the blocks to be tuned. Configure the interface for tuning with `looptune` by adding analysis points that define the separation between the plant and the controller. Also add the analysis points needed for imposing tuning requirements.

```
open_system('rct_distillation')

tuned_blocks = {'PI_L','PI_V','DM'};
st0 = slTuner('rct_distillation',tuned_blocks);

addPoint(st0,{'L','V','y','r','dL','dV'});
```



Decoupling controller for a distillation column

Copyright 2016-2021 The MathWorks, Inc.

This system is now ready for tuning with `looptune`, using tuning goals that you specify. For example, specify a target bandwidth range. Create a tuning requirement that imposes reference tracking in both channels of the system, and a disturbance rejection requirement.

```
wc = [0.1,0.5];
req1 = TuningGoal.Tracking('r','y',15,0.001,1);
max_disturbance_gain = frd([0.05 5 5],[0.001 0.1 10],'TimeUnit','min');
req2 = TuningGoal.Gain({'dL','dV'},'y',max_disturbance_gain);

controls = {'L','V'};
measurement = 'y';

[st,gam,info] = looptune(st0,controls,measurement,wc,req1,req2);
```

```
Final: Peak gain = 1.04, Iterations = 73
```

`looptune` successfully tunes the system to these requirements. However, you might want to switch to `systune` to take advantage of additional flexibility in configuring your problem. For example, instead of tuning both channels to a loop bandwidth inside `wc`, you might want to specify different crossover frequencies for each loop. Or, you might want to enforce the tuning requirements, `req1` and `req2`, as hard constraints, and add other requirements as soft requirements.

Convert the `looptune` input arguments to a set of input arguments for `systune`.

```
[st0,SoftReqs,HardReqs,sysopt] = looptuneSetup(st0,controls,measurement,wc,req1,req2);
```

This command returns a set of arguments you can feed to `systune` for equivalent results to tuning with `looptune`. In other words, the following command is equivalent to the `looptune` command.

```
[st,fsoft,ghard,info] = systune(st0,SoftReqs,HardReqs,sysopt);
```

```
Final: Peak gain = 1.04, Iterations = 73
```

Examine the tuning requirements returned by `looptuneSetup`. When tuning this control system with `looptune`, all requirements are treated as soft requirements. Therefore, `HardReqs` is empty. `SoftReqs` is an array of `TuningGoal` requirements. These requirements together enforce the bandwidth and margins of the `looptune` command, plus the additional requirements that you specified.

```
SoftReqs
```

```
SoftReqs =

  5x1 heterogeneous SystemLevel (LoopShape, Tracking, Gain, ...) array with properties:

    Models
    Openings
    Name
```

For example, examine the first entry in `SoftReqs`.

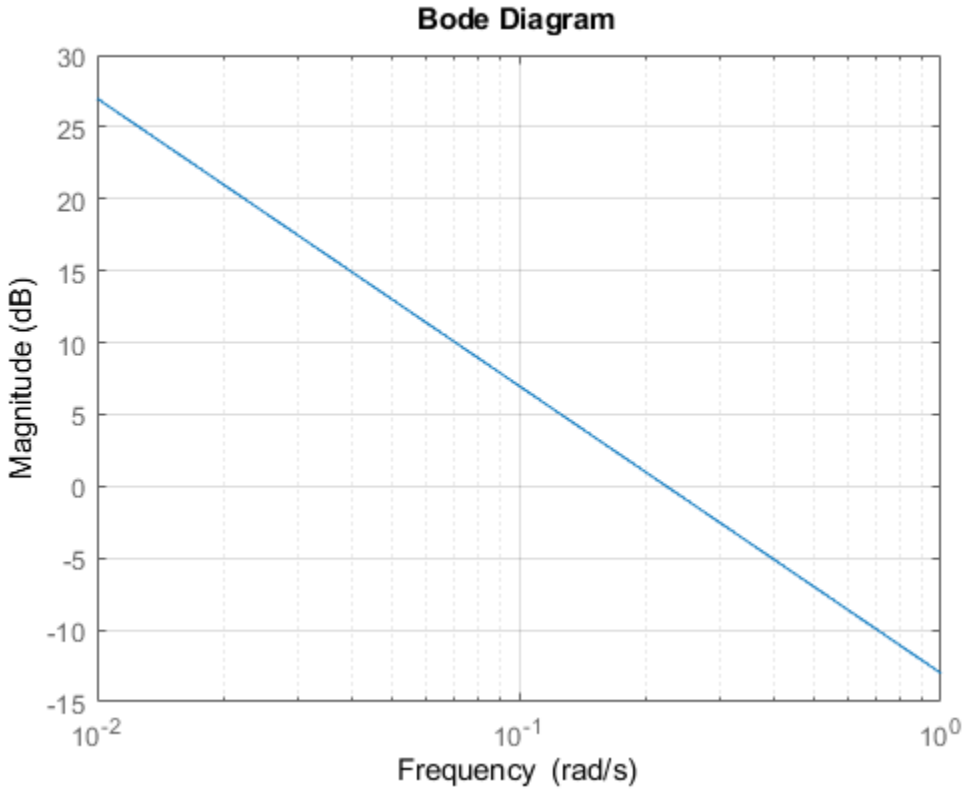
```
SoftReqs(1)
```

```
ans =
```

```
  LoopShape with properties:

       LoopGain: [1x1 zpk]
       CrossTol: 0.3495
          Focus: [0 Inf]
      Stabilize: 1
    LoopScaling: 'on'
       Location: {'y'}
         Models: NaN
       Openings: {0x1 cell}
           Name: 'Open loop GC'
```

`looptuneSetup` expresses the target crossover frequency range `wc` as a `TuningGoal.LoopShape` requirement. This requirement constrains the open-loop gain profile to the loop shape stored in the `LoopGain` property, with a crossover frequency and crossover tolerance (`CrossTol`) determined by `wc`. Examine this loop shape.

```
bodemag(SoftReqs(1).LoopGain,logspace(-2,0)),grid
```



The target crossover is expressed as an integrator gain profile with a crossover between 0.1 and 0.5 rad/s, as specified by `wc`. If you want to specify a different loop shape, you can alter this `TuningGoal.LoopShape` requirement before providing it to `systune`.

`looptune` also tunes to default stability margins that you can change using `looptuneOptions`. For `systune`, stability margins are specified using `TuningGoal.Margins` requirements. Here,

`looptuneSetup` has expressed the default stability margins as soft `TuningGoal.Margins` requirements. For example, examine the fourth entry in `SoftReqs`.

```
SoftReqs(4)
```

```
ans =

  Margins with properties:

      GainMargin: 7.6000
     PhaseMargin: 45
    ScalingOrder: 0
           Focus: [0 Inf]
        Location: {2x1 cell}
          Models: NaN
        Openings: {0x1 cell}
            Name: 'Margins at plant inputs'
```

The last entry in `SoftReqs` is a similar `TuningGoal.Margins` requirement constraining the margins at the plant outputs. `looptune` enforces these margins as soft requirements. If you want to convert them to hard constraints, pass them to `systune` in the input vector `HardReqs` instead of the input vector `SoftReqs`.

Input Arguments

`looptuneInputs` — Control system and requirements configured for tuning with `looptune`
valid `looptune` input sequence

Control system and requirements configured for tuning with `looptune`, specified as a valid `looptune` input sequence. For more information about the arguments in a valid `looptune` input sequence, see the `looptune` reference page.

Output Arguments

`st0` — Interface for tuning control systems modeled in Simulink
`slTuner` interface

Interface for tuning control systems modeled in Simulink, returned as an `slTuner` interface. `st0` is identical to the `slTuner` interface you use as input to `looptuneSetup`.

`SoftReqs` — Soft tuning requirements
vector of `TuningGoal` requirement objects

Soft tuning requirements for tuning with `systune`, returned as a vector of `TuningGoal` requirement objects.

`looptune` expresses most of its implicit tuning requirements as soft tuning requirements. For example, a specified target loop bandwidth is expressed as a `TuningGoal.LoopShape` requirement with integral gain profile and crossover at the target frequency. Additionally, `looptune` treats all of the explicit requirements you specify (`Req1,...ReqN`) as soft requirements. `SoftReqs` contains all of these tuning requirements.

`HardReqs` — Hard tuning requirements
vector of `TuningGoal` requirement objects

Hard tuning requirements (constraints) for tuning with `systune`, returned as a vector of `TuningGoal` requirement objects.

Because `looptune` treats most tuning requirements as soft requirements, `HardReqs` is usually empty. However, if you change the default `MaxFrequency` option of the `looptuneOptions` set, `loopopt`, then this requirement appears as a hard `TuningGoal.Poles` constraint.

`sysopt` — Algorithm options for `systune` tuning
`systuneOptions` options set

Algorithm options for `systune` tuning, returned as a `systuneOptions` options set.

Some of the options in the `looptuneOptions` set, `loopopt`, are converted into hard or soft requirements that are returned in `HardReqs` and `SoftReqs`. Other options correspond to options in the `systuneOptions` set.

See Also

`slTuner` | `looptune` | `systune` | `looptuneOptions` | `systuneOptions` | `looptuneSetup (for genss)`

Introduced in R2014a

loopview

Graphically analyze results of control system tuning using `slTuner` interface

Syntax

```
loopview(st,controls,measurements)

loopview(st,info)
```

Description

`loopview(st,controls,measurements)` plots characteristics of the control system described by the `slTuner` interface `st`. Use `loopview` to analyze the performance of a tuned control system you obtain using `looptune`.

`loopview` plots:

- The gains of the open-loop frequency response measured at the plant inputs (`controls` analysis points) and at plant outputs (`measurements` analysis points)
- The (largest) gain of the sensitivity and complementary sensitivity functions at the plant inputs or outputs

`loopview(st,info)` uses the `info` structure returned by `looptune` and also plots the target and tuned values of tuning constraints imposed on the system. Use this syntax to assist in troubleshooting when tuning fails to meet all requirements.

Additional plots with this syntax include:

- Normalized multi-loop disk margins at the plant inputs and outputs. For more information about disk margins, see “Stability Analysis Using Disk Margins” (Robust Control Toolbox).
- Target vs. achieved response for any additional tuning goal you used with `looptune`.

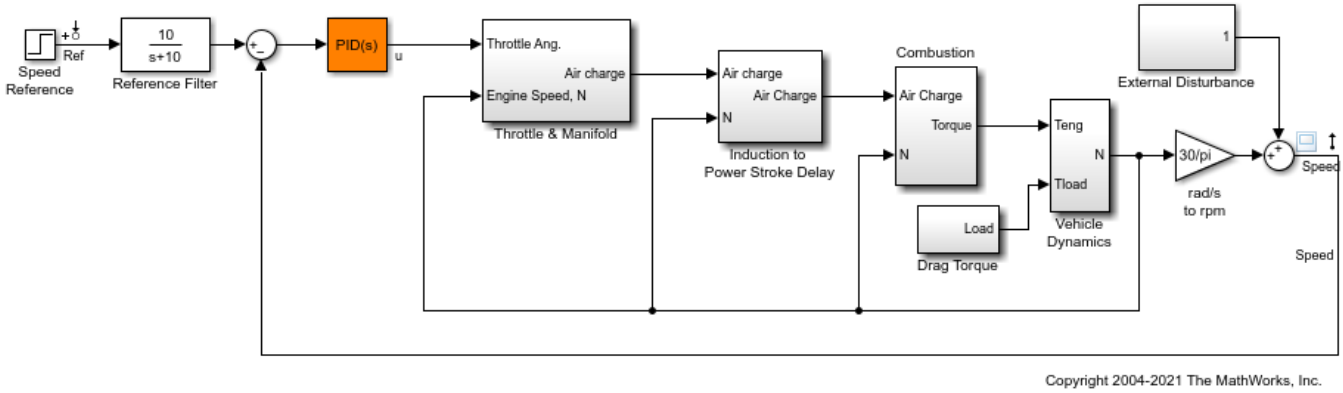
Examples

Graphically Analyze Results of Control System Tuning

Tune the Simulink® model, `rct_engine_speed`, to achieve a specified settling time. Use `loopview` to graphically analyze the tuning results.

Open the model.

```
mdl = 'rct_engine_speed';
open_system(mdl);
```



Create an `slTuner` interface for the model and specify the PID Controller block to be tuned.

```
st0 = slTuner(mdl,'PID Controller');
```

Specify a requirement to achieve a 2 second settling time for the `Speed` signal when tracking the reference signal.

```
req = TuningGoal.Tracking('Ref','Speed',2);
```

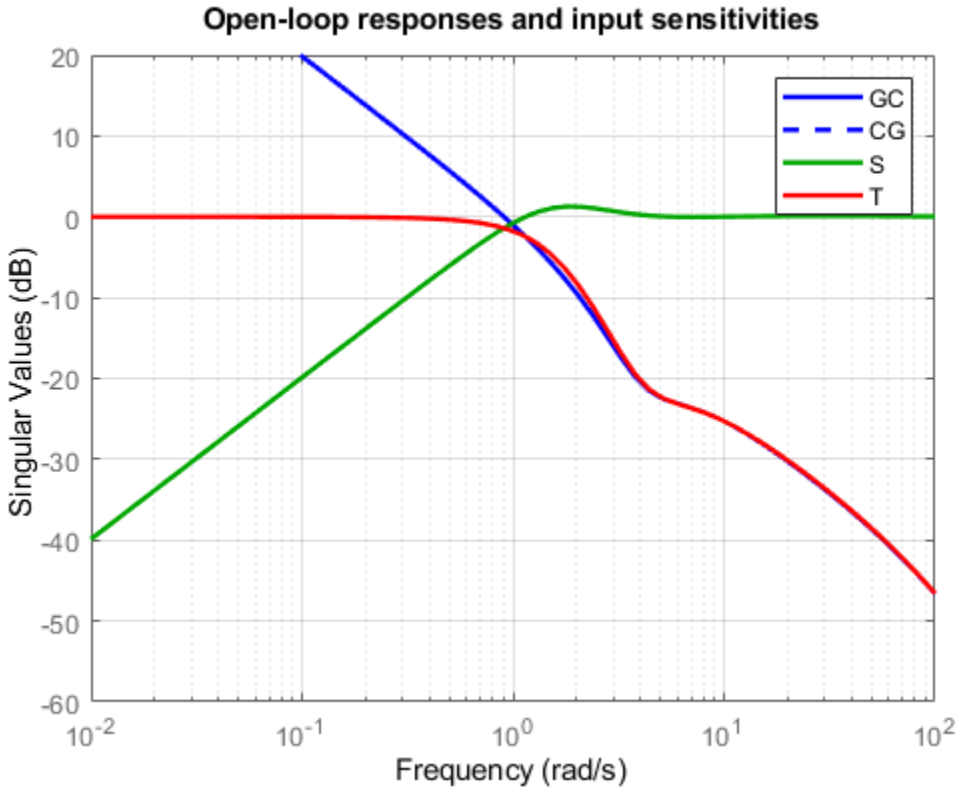
Tune the PID Controller block.

```
addPoint(st0,'u')

control = 'u';
measurement = 'Speed';

wc = 1;

[st1,gam,info] = looptune(st0,control,measurement,wc);
```

```
Final: Peak gain = 0.979, Iterations = 4
Achieved target gain value TargetGain=1.
```

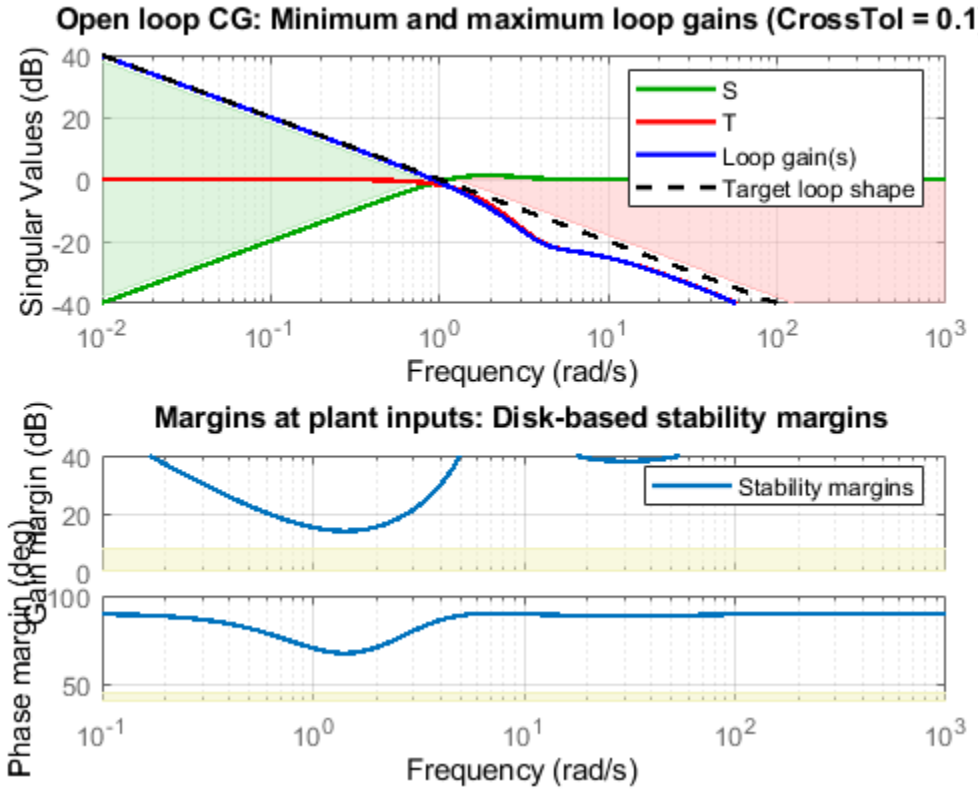
View the response of the model for the tuned block values.

```
loopview(st1,control,measurement);
```



Compare the performance of the tuned block against the tuning goals.

```
figure
loopview(st1,info);
```



Input Arguments

`st` — Interface for tuning control systems modeled in Simulink

`slTuner` interface

Interface for tuning control systems modeled in Simulink, specified as an `slTuner` interface.

`controls` — Controller output

character vector | cell array of character vectors

Controller output name, specified as one of the following:

- Character vector — Name of an analysis point of `st`.

  You can specify the full name or any portion of the name that uniquely identifies the analysis point among the other analysis points of `st`.

  For example, `'u'`.
- Cell array of character vectors — Multiple analysis point names.

  For example, `{'u','y'}`.

`measurements` — Measurement

character vector | cell array of character vectors

Measurement signal name, specified as one of the following:

- Character vector — Name of an analysis point of `st`.

  You can specify the full name or any portion of the name that uniquely identifies the analysis point among the other analysis points of `st`.

  For example, `'u'`.
- Cell array of character vector — Multiple analysis point names.

  For example, `{'u','y'}`.

`info` — Detailed information about each optimization run
structure

Detailed information about each optimization run, specified as the structure returned by `looptune`.

Alternative Functionality

For analyzing Control System Toolbox models tuned with `looptune`, use `loopview`.

See Also

`loopview` | `looptune` | `slTuner`

Topics
“Decoupling Controller for a Distillation Column”
“Tuning of a Two-Loop Autopilot”
“Mark Signals of Interest for Control System Analysis and Design” on page 2-38

Introduced in R2014a

removeBlock

Remove block from list of tuned blocks in `slTuner` interface

Syntax

```
removeBlock(st,blk)
```

Description

`removeBlock(st,blk)` removes the specified block from the list of tuned blocks on page 18-346 for the `slTuner` interface, `st`. You can specify `blk` to remove either a single or multiple blocks.

`removeBlock` does not modify the Simulink model associated with `st`.

Examples

Remove Block From List of Tuned Blocks of slTuner Interface

Create an `slTuner` interface for the `scdcascade` model. Add `C1` and `C2` as tuned blocks to the interface.

```
st = slTuner('scdcascade',{'C1','C2'});
```

Remove `C1` from the list of tuned blocks of `st`.

```
removeBlock(st,'C1');
```

Input Arguments

`st` — Interface for tuning control systems modeled in Simulink

`slTuner` interface

Interface for tuning control systems modeled in Simulink, specified as an `slTuner` interface.

`blk` — Block

character vector | string | cell array of character vectors | string array | positive integer | vector of positive integers

Block to remove from the list of tuned blocks on page 18-346 for `st`, specified as one of the following:

- Character vector or string — Full block path or any portion of the block path that uniquely identifies the block among the other tuned blocks of `st`. For example, `blk = 'scdcascade/C1'`.
- Cell array of character vectors or string array — Specifies multiple blocks. For example, `blk = {'C1','C2'}`.
- Positive integer — Block index. For example, `blk = 1`.
- Vector of positive integers — Specifies multiple block indices. For example, `blk = [1 2]`.

To determine the name or index associated with a tuned block, type `st`. The software displays the contents of `st` in the MATLAB command window, including the tuned block names.

More About

Tuned Blocks

Tuned blocks, used by the `slTuner` interface, identify blocks in a Simulink model whose parameters are to be tuned to satisfy tuning goals. You can tune most Simulink blocks that represent linear elements such as gains, transfer functions, or state-space models. (For the complete list of blocks that support tuning, see “How Tuned Simulink Blocks Are Parameterized” on page 10-27). You can also tune more complex blocks such as SubSystem or S-Function blocks by specifying an equivalent tunable linear model.

Use tuning commands such as `systune` to tune the parameters of tuned blocks.

You must specify tuned blocks (for example, `C1` and `C2`) when you create an `slTuner` interface.

```
st = slTuner('scdcascade',{'C1','C2'})
```

You can modify the list of tuned blocks using `addBlock` and `removeBlock`.

To interact with the tuned blocks use:

- `getBlockParam`, `getBlockValue`, and `getTunedValue` to access the tuned block parameterizations and their current values.
- `setBlockParam`, `setBlockValue`, and `setTunedValue` to modify the tuned block parameterizations and their values.
- `writeBlockValue` to update the blocks in a Simulink model with the current values of the tuned block parameterizations.

See Also

`addBlock` | `slTuner` | `addPoint` | `addOpening`

Introduced in R2014a

setBlockParam

Set parameterization of tuned block in `slTuner` interface

Syntax

```
setBlockParam(st,blk,tunable_mdl)
setBlockParam(st,blk1,tunable_mdl1,...,blkN,tunable_mdlN)

setBlockParam(st,blk)
setBlockParam(st)
```

Description

`setBlockParam` lets you override the default parameterization for a tuned block on page 18-350 in an `slTuner` interface. You can also specify the parameterization for non-atomic components such as Subsystem or S-Function blocks.

An `slTuner` interface parameterizes each tuned Simulink block as a Control Design Block, or a generalized parametric model of type `genmat` or `genss`. This parameterization specifies the tuned variables on page 18-351 for commands such as `systune`.

`setBlockParam(st,blk,tunable_mdl)` assigns a tunable model as the parameterization of the specified block of an `slTuner` interface.

`setBlockParam(st,blk1,tunable_mdl1,...,blkN,tunable_mdlN)` assigns parameterizations to multiple blocks at once.

`setBlockParam(st,blk)` reverts to the default parameterization for the block referenced by `blk` and initializes the block with the current block value in Simulink.

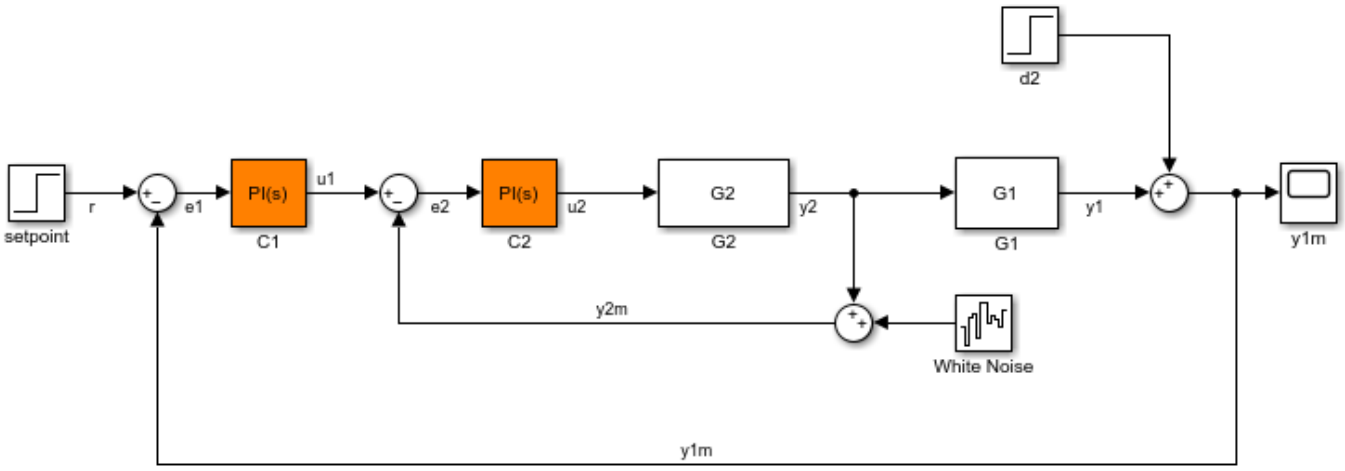
`setBlockParam(st)` reverts all the tuned blocks of `st` to their default parameterizations.

Examples

Set Parameterization of Tuned Block

Create an `slTuner` interface for the `scdcascade` model.

```
open_system('scdcascade')
st = slTuner('scdcascade',{'C1','C2'});
```



Both `C1` and `C2` are PI controllers. Examine the default parameterization of `C1`.

```
getBlockParam(st,'C1')
```

```
ans =

  Tunable continuous-time PID controller "C1" with formula:

             1
  Kp + Ki * ---
             s

  and tunable parameters Kp, Ki.

Type "pid(ans)" to see the current value and "get(ans)" to see all properties.
```

The default parameterization is a tunable PI controller (`tunablePID`).

Reparameterize `C1` as a proportional controller. Initialize the proportional gain to 4.2, and assign the parameterization to the block.

```
G = tunableGain('C1',4.2);
setBlockParam(st,'C1',G);
```

Tuning commands, such as `systune`, now use this proportional controller parameterization of the `C1` block of `st`. The custom parameterization is compatible with the default parameterization of the Simulink® block. Therefore, you can use `writeBlockValue` to write the tuned values back to the block.

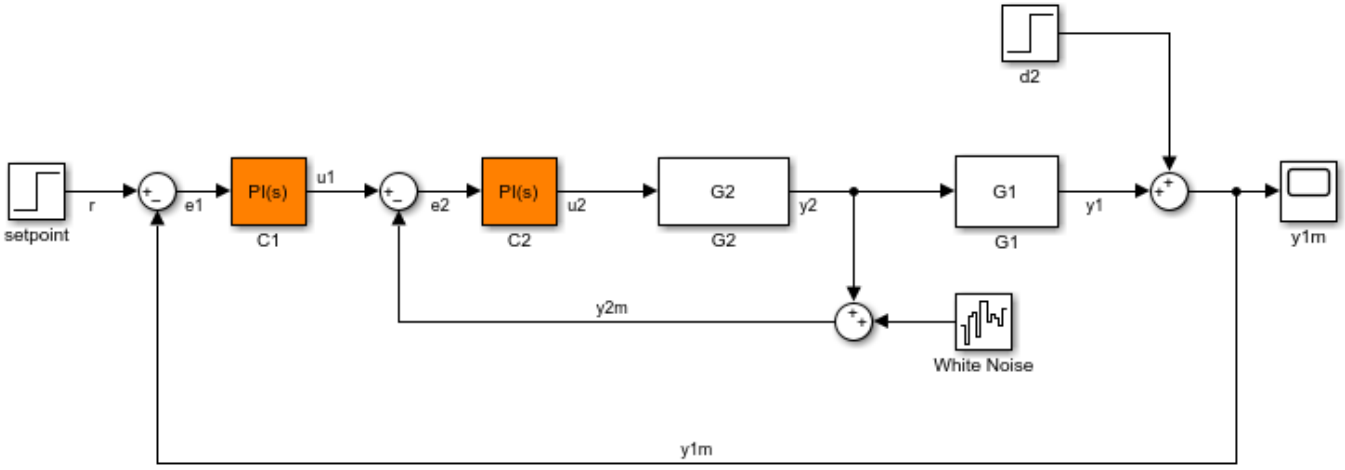
You can also use `setBlockParam` to set multiple block parameterizations at once, without requiring multiple recompilations of the model. For example, reparameterize both `C1` and `C2` as PID controllers.

```
C1PID = tunablePID('C1PID','PID');
C2PID = tunablePID('C2PID','PID');
setBlockParam(st,'C1',C1PID,'C2',C2PID);
```

Revert Parameterization of Tuned Block to Default

Create an `slTuner` interface for the `scdcascade` model.

```
open_system('scdcascade')
st = slTuner('scdcascade',{'C1','C2'});
```



Modify the parameterization of `C2` to be a tunable gain and examine the result.

```
G = tunableGain('C2',5);
setBlockParam(st,'C2',G);
getBlockParam(st,'C2')
```

```
ans =

  Tunable gain "C2" with 1 outputs, 1 inputs, and 1 tunable parameters.

Type "ss(ans)" to see the current value and "get(ans)" to see all properties.
```

Revert the parameterization of `C2` back to the default PI controller and examine the result.

```
setBlockParam(st,'C2');
getBlockParam(st,'C2')
```

```
ans =

  Tunable continuous-time PID controller "C2" with formula:

            1
  Kp + Ki * ---
            s

  and tunable parameters Kp, Ki.
```

```
Type "pid(ans)" to see the current value and "get(ans)" to see all properties.
```

Input Arguments

`st` — Interface for tuning control systems modeled in Simulink

`slTuner` interface

Interface for tuning control systems modeled in Simulink, specified as an `slTuner` interface.

`blk` — Block

character vector | string | cell array of character vectors | string array

Block in the list of tuned blocks for `st`, specified as a character vector or string. You can specify the full block path or any portion of the block path that uniquely identifies the block among the other tuned blocks of `st`.

Example: `blk = 'scdcascade/C1'`, `blk = "C1"`

When reverting to the default block parameterization using `setBlockParam(st,blk)`, you can specify `blk` as a cell array of character vectors or string array to revert multiple blocks.

Example: `{'C1','C2'}`

`tunable_mdl` — Block parameterization

control design block | generalized state-space model | generalized matrix | tunable gain surface

Block parameterization, specified as one of the following:

- Control Design Block
- Generalized state-space (`genss`) model
- Generalized matrix (`genmat`)
- Tunable gain surface, modeled by `tunableSurface`

More About

Tuned Blocks

Tuned blocks, used by the `slTuner` interface, identify blocks in a Simulink model whose parameters are to be tuned to satisfy tuning goals. You can tune most Simulink blocks that represent linear elements such as gains, transfer functions, or state-space models. (For the complete list of blocks that support tuning, see “How Tuned Simulink Blocks Are Parameterized” on page 10-27). You can also tune more complex blocks such as SubSystem or S-Function blocks by specifying an equivalent tunable linear model.

Use tuning commands such as `systune` to tune the parameters of tuned blocks.

You must specify tuned blocks (for example, `C1` and `C2`) when you create an `slTuner` interface.

```
st = slTuner('scdcascade',{'C1','C2'})
```

You can modify the list of tuned blocks using `addBlock` and `removeBlock`.

To interact with the tuned blocks use:

- `getBlockParam`, `getBlockValue`, and `getTunedValue` to access the tuned block parameterizations and their current values.
- `setBlockParam`, `setBlockValue`, and `setTunedValue` to modify the tuned block parameterizations and their values.
- `writeBlockValue` to update the blocks in a Simulink model with the current values of the tuned block parameterizations.

Tuned Variables

Within an `slTuner` interface, tuned variables are any Control Design Blocks involved in the parameterization of a tuned Simulink block, either directly or through a generalized parametric model. Tuned variables are the parameters manipulated by tuning commands such as `systune`.

For Simulink blocks parameterized by a generalized model or a tunable surface:

- `getBlockValue` provides access to the overall value of the block parameterization. To access the values of the tuned variables within the block parameterization, use `getTunedValue`.
- `setBlockValue` cannot be used to modify the block value. To modify the values of tuned variables within the block parameterization, use `setTunedValue`.

For Simulink blocks parameterized by a Control Design Block, the block itself is the tuned variable. To modify the block value, you can use either `setBlockValue` or `setTunedValue`. Similarly, you can retrieve the block value using either `getBlockValue` or `getTunedValue`.

See Also

`slTuner` | `getBlockParam` | `setBlockValue` | `setTunedValue` | `writeBlockValue` | `systune` | `genss`

Topics

“How Tuned Simulink Blocks Are Parameterized” on page 10-27

Introduced in R2011b

setBlockRateConversion

Set rate conversion settings for tuned block in `slTuner` interface

Syntax

```
setBlockRateConversion(st,blk,method)
setBlockRateConversion(st,blk,'tustin',pwf)

setBlockRateConversion(st,blk,IF,DF)
```

Description

When you use `systune` with Simulink, tuning is performed at the sampling rate specified by the `Ts` property of the `slTuner` interface. When you use `writeBlockValue` to write tuned parameters back to the Simulink model, each tuned block value is automatically converted from the sample time used for tuning, to the sample time of the Simulink block. The rate conversion method associated with each tuned block specifies how this resampling operation should be performed. Use `getBlockRateConversion` to query the block conversion rate and use `setBlockRateConversion` to modify it.

`setBlockRateConversion(st,blk,method)` sets the rate conversion method of a tuned block on page 18-354 in the `slTuner` interface, `st`.

`setBlockRateConversion(st,blk,'tustin',pwf)` sets the Tustin method as the rate conversion method for `blk`, with `pwf` as the prewarp frequency.

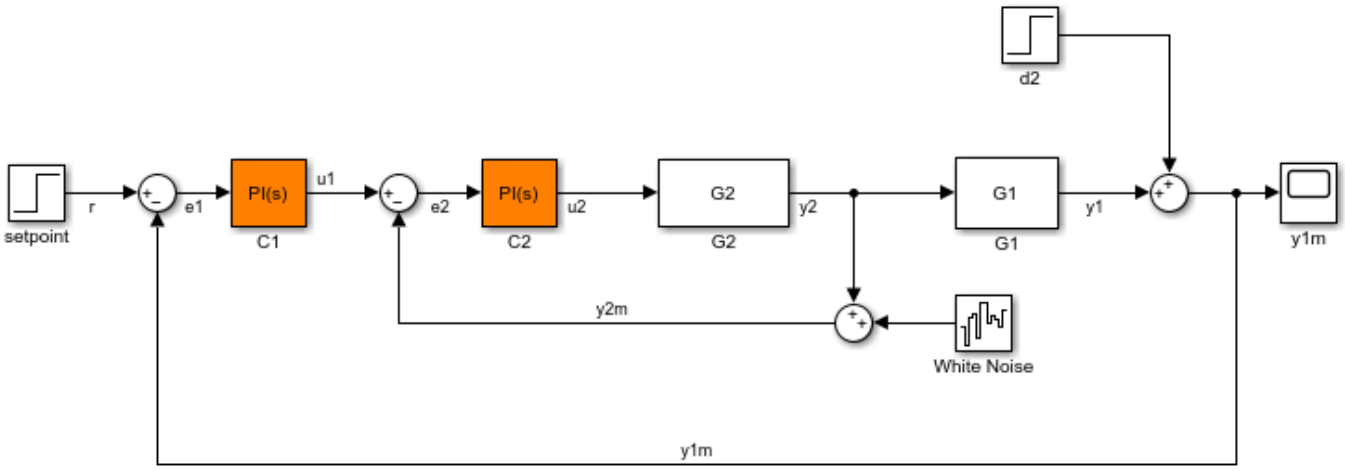
`setBlockRateConversion(st,blk,IF,DF)` sets the discretization methods for the integrator and derivative filter terms when `blk` is a continuous-time PID Controller block. For discrete-time PID blocks, these methods are specified in the Simulink block and cannot be modified in the `slTuner` interface.

Examples

Set Rate Conversion Settings of Tuned PID Block

Create an `slTuner` interface for the Simulink model `scdcascade`. Set the block rate conversion settings of one of the tuned blocks.

```
open_system('scdcascade')
st = slTuner('scdcascade',{'C1','C2'});
```



Examine the default block rate conversion for the PID Controller block `C1`.

```
[IF,DF] = getBlockRateConversion(st,'C1')
```

```
IF =

    'Trapezoidal'


DF =

    'Trapezoidal'
```

By default, both the integrator and derivative filter controller methods are `Trapezoidal`. Set the integrator to `BackwardEuler` and the derivative to `ForwardEuler`.

```
IF = 'BackwardEuler';
DF = 'ForwardEuler';
setBlockRateConversion(st,'C1',IF,DF);
```

Input Arguments

`st` — Interface for tuning control systems modeled in Simulink

`slTuner` interface

Interface for tuning control systems modeled in Simulink, specified as an `slTuner` interface.

`blk` — Block

character vector | string

Block in the list of tuned blocks for `st`, specified as a character vector or string. You can specify the full block path or any portion of the block path that uniquely identifies the block among the other tuned blocks of `st`.

Example: `blk = 'scdcascade/C1'`, `blk = "C1"`

`method` — Rate conversion method
`'zoh'` | `'foh'` | `'tustin'` | `'matched'`

Rate conversion method associated with `blk`, specified as one of the following:

- `'zoh'` — Zero-order hold on the inputs. This method is the default rate-conversion method for most dynamic blocks.
- `'foh'` — Linear interpolation of inputs.
- `'tustin'` — Bilinear (Tustin) approximation. Optionally, specify a prewarp frequency with the `pwf` argument for better frequency-domain matching between the original and rate-converted dynamics near the prewarp frequency.
- `'matched'` — Matched pole-zero method. This method is available for SISO blocks only.

For more detailed information about these rate-conversion methods, see “Continuous-Discrete Conversion Methods”.

`pwf` — Prewarp frequency for Tustin method
positive scalar

Prewarp frequency for the Tustin method, specified as a positive scalar.

`IF,DF` — Integrator and filter methods
`'ForwardEuler'` | `'BackwardEuler'` | `'Trapezoidal'`

Integrator and filter methods for rate conversion of PID Controller block, each specified as one of the following:

- `'ForwardEuler'` — Integrator or derivative-filter state discretized as `Ts/(z-1)`
- `'BackwardEuler'` — `Ts*z/(z-1)`
- `'Trapezoidal'` — `(Ts/2)*(z+1)/(z-1)`

For continuous-time PID blocks, the default methods are `'Trapezoidal'` for both integrator and derivative filter. This method is the same as the Tustin method.

For discrete-time PID blocks, `IF` and `DF` are determined by the **Integrator method** and **Filter method** settings in the Simulink block and cannot be changed with `setBlockRateConversion`.

See the Discrete PID Controller and `pid` reference pages for more details about integrator and filter methods.

More About

Tuned Block

Tuned blocks, used by the `slTuner` interface, identify blocks in a Simulink model whose parameters are to be tuned to satisfy tuning goals. You can tune most Simulink blocks that represent linear elements such as gains, transfer functions, or state-space models. (For the complete list of blocks that support tuning, see “How Tuned Simulink Blocks Are Parameterized” on page 10-27). You can also tune more complex blocks such as SubSystem or S-Function blocks by specifying an equivalent tunable linear model.

Use tuning commands such as `systune` to tune the parameters of tuned blocks.

You must specify tuned blocks (for example, `C1` and `C2`) when you create an `slTuner` interface.

```
st = slTuner('scdcascade',{'C1','C2'})
```

You can modify the list of tuned blocks using `addBlock` and `removeBlock`.

To interact with the tuned blocks use:

- `getBlockParam`, `getBlockValue`, and `getTunedValue` to access the tuned block parameterizations and their current values.
- `setBlockParam`, `setBlockValue`, and `setTunedValue` to modify the tuned block parameterizations and their values.
- `writeBlockValue` to update the blocks in a Simulink model with the current values of the tuned block parameterizations.

Tips

- For Model Discretizer blocks, the rate conversion method is specified in the Simulink block and cannot be modified with `setBlockRateConversion`.
- For static blocks such as Gain or Lookup Table blocks, the block rate conversion method is ignored.

See Also

`slTuner` | `writeBlockValue` | `getBlockRateConversion`

Topics

“Tuning of a Digital Motion Control System”
“Continuous-Discrete Conversion Methods”

Introduced in R2014a

setBlockValue

Set value of tuned block parameterization in `slTuner` interface

Syntax

```
setBlockValue(st,blk,value)
```

```
setBlockValue(st,blkValues)
```

Description

`setBlockValue` lets you initialize or modify the current value of the parameterization of a tuned block on page 18-360 in an `slTuner` interface.

An `slTuner` interface parameterizes each tuned Simulink block as a Control Design Block, or a generalized parametric model of type `genmat` or `genss`. This parameterization specifies the tuned variables on page 18-360 for commands such as `systune`.

`setBlockValue(st,blk,value)` sets the current value of the parameterization of a block in the `slTuner` interface, `st`.

`setBlockValue(st,blkValues)` updates the values of the parameterizations of multiple blocks using the structure, `blkValues`.

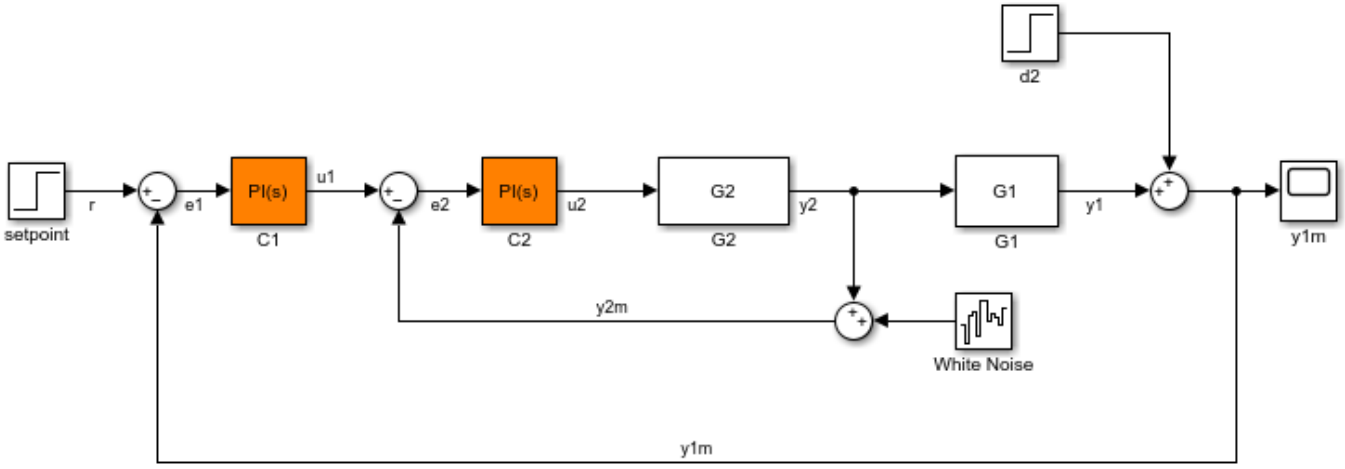
Examples

Set Value of Tuned Block Parameterization

Create an `slTuner` interface for the `scdcascade` model, and set the value of the parametrization of one of the tuned blocks.

Create an `slTuner` interface.

```
open_system('scdcascade')
st = slTuner('scdcascade',{'C1','C2'});
```



Both `C1` and `C2` are PI controllers. Examine the default parameterization of `C1`.

```
getBlockParam(st,'C1')
```

```
ans =

  Tunable continuous-time PID controller "C1" with formula:

             1
  Kp + Ki * ---
             s

  and tunable parameters Kp, Ki.

Type "pid(ans)" to see the current value and "get(ans)" to see all properties.
```

The default parameterization is a PI controller with two tunable parameters, `Kp` and `Ki`.

Set the value of the parameterization of `C1`.

```
C = pid(4.2);
setBlockValue(st,'C1',C);
```

Examine the value of the parameterization of `C1`.

```
getBlockValue(st,'C1')
```

```
ans =

  Kp = 4.2

Name: C1
P-only controller.
```

Examine the parameterization of C1.

```
getBlockParam(st,'C1')
```

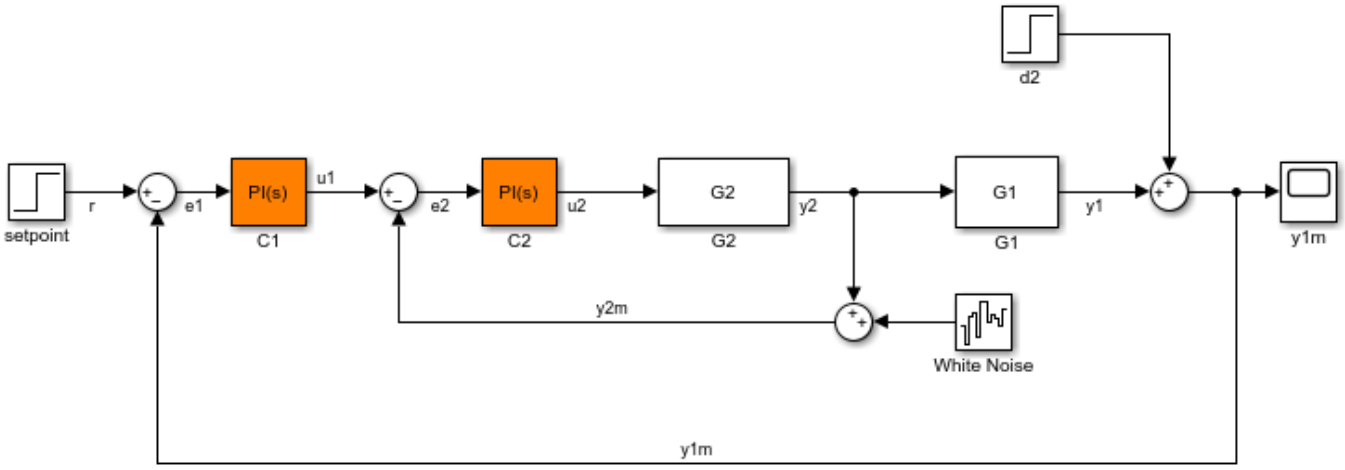
```
ans =

  Tunable continuous-time PID controller "C1" with formula:

             1
  Kp + Ki * ---
             s

  and tunable parameters Kp, Ki.

Type "pid(ans)" to see the current value and "get(ans)" to see all properties.
```

Observe that although the current block value is a P-only controller, the block parameterization continues to be a PI-controller.

Set Value of Multiple Tuned Block Parameterizations

Create an `slTuner` interface.

```
open_system('scdcascade')
st = slTuner('scdcascade',{'C1','C2'});
```



Create a block value structure with field names that correspond to the tunable blocks in `st`.

```
blockValues = getBlockValue(st);
blockValues.C1 = pid(0.2,0.1);
blockValues.C2 = pid(2.3);
```

Set the values of the parameterizations of the tunable blocks in `st` using the defined structure.

```
setBlockValue(st,blockValues);
```

Input Arguments

`st` — Interface for tuning control systems modeled in Simulink

`slTuner` interface

Interface for tuning control systems modeled in Simulink, specified as an `slTuner` interface.

`blk` — Block

character vector | string

Block in the list of tuned blocks for `st`, specified as a character vector or string. You can specify the full block path or any portion of the path that uniquely identifies the block among the other tuned blocks of `st`.

Example: `blk = 'scdcascade/C1'`, `blk = "C1"`

Note `setBlockValue` allows you to modify only the overall value of the parameterization of `blk`. To modify the values of elements within custom block parameterizations, such as generalized state-space models, use `setTunedValue`.

`value` — Value of block parameterization

numeric LTI model | control design block

Value of block parameterization, specified as a numeric LTI model or a Control Design Block, such `tunableGain` or `tunablePID`. The value of `value` must be compatible with the parameterization of `blk`. For example, if `blk` is parameterized as a PID controller, then `value` must be an `tunablePID` block, a numeric `pid` model, or a numeric `tf` model that represents a PID controller.

`setBlockValue` updates the value of the parameters of the tuned block based on the parameters of `value`. Using `setBlockValue` does not change the structure of the parameterization of the tuned block. To change the parameterization of `blk`, use `setBlockParam`. For example, you can use `setBlockParam` to change a block parameterization from `tunablePID` to a three-pole `tunableTF` model.

`blkValues` — Values of multiple block parameterizations

structure

Values of multiple block parameterizations, specified as a structure with fields specified as numeric LTI models or Control Design Blocks. The field names are the names of blocks in `st`. Only blocks common to `st` and `blkValues` are updated, while all other blocks in `st` remain unchanged.

To specify `blkValues`, you can retrieve and modify the block parameterization value structure from `st`.

```
blkValues = getblockValue(st);
blkValues.C1 = pid(0.1,0.2);
```

Note For Simulink blocks whose names are not valid field names, specify the corresponding field name in `blkValues` as it appears in the block parameterization.

```
blockParam = getBlockParam(st,'B-1');
fieldName = blockParam.Name;
blockValues = struct(fieldName,newB1);
```

More About

Tuned Blocks

Tuned blocks, used by the `slTuner` interface, identify blocks in a Simulink model whose parameters are to be tuned to satisfy tuning goals. You can tune most Simulink blocks that represent linear elements such as gains, transfer functions, or state-space models. (For the complete list of blocks that support tuning, see “How Tuned Simulink Blocks Are Parameterized” on page 10-27). You can also tune more complex blocks such as SubSystem or S-Function blocks by specifying an equivalent tunable linear model.

Use tuning commands such as `systune` to tune the parameters of tuned blocks.

You must specify tuned blocks (for example, `C1` and `C2`) when you create an `slTuner` interface.

```
st = slTuner('scdcascade',{'C1','C2'})
```

You can modify the list of tuned blocks using `addBlock` and `removeBlock`.

To interact with the tuned blocks use:

- `getBlockParam`, `getBlockValue`, and `getTunedValue` to access the tuned block parameterizations and their current values.
- `setBlockParam`, `setBlockValue`, and `setTunedValue` to modify the tuned block parameterizations and their values.
- `writeBlockValue` to update the blocks in a Simulink model with the current values of the tuned block parameterizations.

Tuned Variables

Within an `slTuner` interface, tuned variables are any Control Design Blocks involved in the parameterization of a tuned Simulink block, either directly or through a generalized parametric model. Tuned variables are the parameters manipulated by tuning commands such as `systune`.

For Simulink blocks parameterized by a generalized model or a tunable surface:

- `getBlockValue` provides access to the overall value of the block parameterization. To access the values of the tuned variables within the block parameterization, use `getTunedValue`.
- `setBlockValue` cannot be used to modify the block value. To modify the values of tuned variables within the block parameterization, use `setTunedValue`.

For Simulink blocks parameterized by a Control Design Block, the block itself is the tuned variable. To modify the block value, you can use either `setBlockValue` or `setTunedValue`. Similarly, you can retrieve the block value using either `getBlockValue` or `getTunedValue`.

See Also

`slTuner` | `getBlockValue` | `setTunedValue` | `setBlockParam` | `writeBlockValue`

Topics
“Fixed-Structure Autopilot for a Passenger Jet”
“How Tuned Simulink Blocks Are Parameterized”

Introduced in R2011b

setTunedValue

Set current value of tuned variable in `slTuner` interface

Syntax

```
setTunedValue(st,var,value)
setTunedValue(st,varValues)
setTunedValue(st,model)
```

Description

`setTunedValue` lets you initialize or modify the current value of a tuned variable on page 18-367 within an `slTuner` interface.

An `slTuner` interface parameterizes each tuned block on page 18-367 as a Control Design Block, or a generalized parametric model of type `genmat` or `genss`. This parameterization specifies the tuned variables for commands such as `systune`.

`setTunedValue(st,var,value)` sets the current value of the tuned variable, `var`, in the `slTuner` interface, `st`.

`setTunedValue(st,varValues)` sets the values of multiple tuned variables in `st` using the structure, `varValues`.

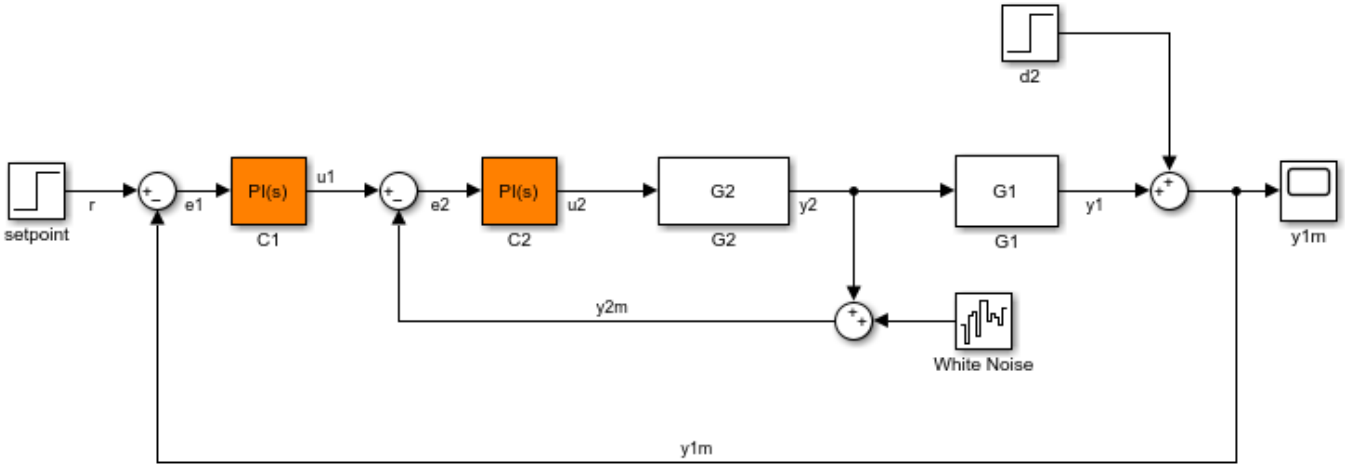
`setTunedValue(st,model)` updates the values of the tuned variables in `st` to match their values in the generalized model `model`. To propagate tuned values from one model to another, use this syntax.

Examples

Set Value of Single Tunable Element within Custom Parameterization

Create an `slTuner` interface for the `scdcascade` model.

```
open_system('scdcascade')
st = slTuner('scdcascade',{'C1','C2'});
```



Set a custom parameterization for one of the tunable blocks.

```
C1CustParam = realp('Kp',1) + tf(1,[1 0]) * realp('Ki',1);
setBlockParam(st,'C1',C1CustParam);
```

These commands set the parameterization of the `C1` controller block to a generalized state-space (`genss`) model containing two tunable parameters, `Ki` and `Kp`.

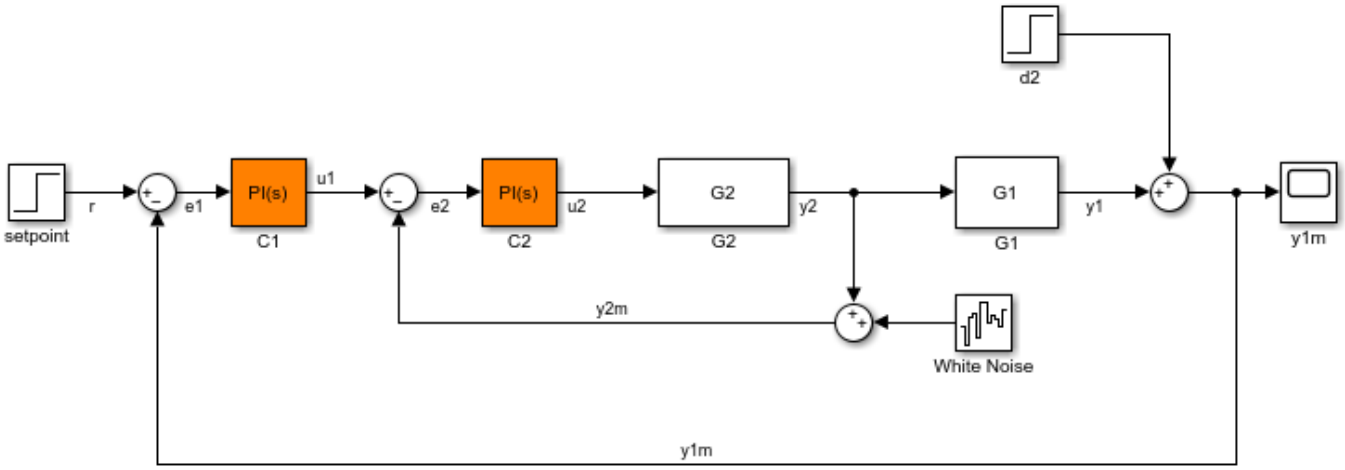
Initialize the value of `Ki` to `10` without changing the value of Kp.

```
setTunedValue(st,'Ki',10);
```

Set Value of Multiple Tunable Elements within Custom Parameterization

Create an `slTuner` interface for the `scdcascade` model.

```
open_system('scdcascade')
st = slTuner('scdcascade',{'C1','C2'});
```



Set a custom parameterization for one of the tunable blocks.

```
C1CustParam = realp('Kp',1) + tf(1,[1 0]) * realp('Ki',1);
setBlockParam(st,'C1',C1CustParam);
```

These commands set the parameterization of the `C1` controller block to a generalized state-space (`genss`) model containing two tunable parameters, `Ki` and `Kp`.

Create a structure of tunable element values, setting `Kp` to `5` and `Ki` to `10`.

```
S = struct('Kp',5,'Ki',10);
```

Set the values of the tunable elements in `st`.

```
setTunedValue(st,S);
```

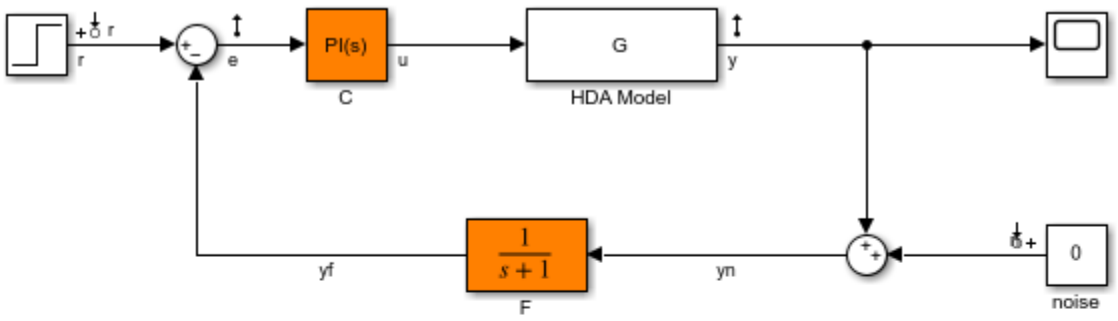
Set Value of Tuned Block Parameterization Using Generalized State-Space Model

Convert an `slTuner` interface for the Simulink® model `rct_diskdrive` to a `genss` model to tune the model blocks using `hinfstruct`. After tuning, update the `slTuner` interface with the tuned parameters and write the parameter values to the Simulink model for validation.

Use of `hinfstruct` requires a Robust Control Toolbox license.

Create an `slTuner` interface for `rct_diskdrive`. Add `C` and `F` as tuned blocks of the interface.

```
open_system('rct_diskdrive');
st = slTuner('rct_diskdrive',{'C','F'});
```



See hinfstruct_demo to see how you can tune the PI gains and the filter coefficient with the HINFSTRUCT command.

Copyright 2004-2010 The MathWorks, Inc.

Copyright 2015-2021 The MathWorks, Inc.

The default parameterization of the transfer function block, `F`, is a transfer function with two free parameters. Because `F` is a low-pass filter, you must constrain its coefficients. To do so, specify a custom parameterization of `F` with filter coefficient `a`.

```
a = realp('a',1);
setBlockParam(st,'F',tf(a,[1 a]));
```

Convert `st` to a `genss` model.

```
m = getIOTransfer(st,{'r','n'},{'y','e'});
```

Typically, for tuning with `hinfstruct`, you append weighting functions to the `genss` model that depend on your design requirements. You then tune the augmented model. For more information, see “Fixed-Structure H-infinity Synthesis with hinfstruct” (Robust Control Toolbox).

For this example, instead of tuning the model, manually adjust the tuned variable values.

```
m.Blocks.C.Kp.Value = 0.00085;
m.Blocks.C.Ki.Value = 0.01;
m.Blocks.a.Value = 5500;
```

After tuning, update the block parameterization values in `st`.

```
setTunedValue(st,m);
```

This is equivalent to `setBlockValue(st,m.Blocks)`.

To validate the tuning result in Simulink, first update the Simulink model with the tuned values.

```
writeBlockValue(st);
```

Input Arguments

`st` — Interface for tuning control systems modeled in Simulink

`slTuner` interface

Interface for tuning control systems modeled in Simulink, specified as an `slTuner` interface.

`var` — Tuned variable

character vector | string

Tuned variable within `st`, specified as a character vector or string. A tuned variable is any Control Design Block, such `realp`, `tunableSS`, or `tunableGain`, involved in the parameterization of a tuned Simulink block, either directly or through a generalized parametric model. To get a list of all tuned variables within `st`, use `getTunedValue(st)`.

`var` can refer to the following:

- For a block parameterized by a Control Design Block, the name of the block. For example, if the parameterization of the block is

  ```
  C = tunableSS('C')
  ```

  then set `var = 'C'`.
- For a block parameterized by a `genmat`/`genss` model, `M`, the name of any Control Design Block listed in `M.Blocks`. For example, if the parameterization of the block is

  ```
  a = realp('a',1);
  C = tf(a,[1 a]);
  ```

  then set `var = 'a'`.

`value` — Value of tuned variable

numeric scalar | numeric array | state-space model

Value of tuned variable in `st`, specified as a numeric scalar, a numeric array or a state-space model that is compatible with the tuned variable. For example, if `var` is a scalar element such as a PID gain, `value` must be a scalar. If `var` is a 2–by–2 `tunableGain`, then `value` must be a 2–by–2 scalar array.

`varValues` — Values of multiple tuned variables

structure

Values of multiple tuned variables in `st`, specified as a structure with fields specified as numeric scalars, numeric arrays, or state-space models. The field names are the names of tuned variables in `st`. Only blocks common to `st` and `varValues` are updated, while all other blocks in `st` remain unchanged.

To specify `varValues`, you can retrieve and modify the tuned variable structure from `st`.

```
varValues = getTunedValue(st);
varValues.Ki = 10;
```

`model` — Tuned model
generalized LTI model

Tuned model that has some parameters in common with `st`, specified as a Generalized LTI Model. Only variables common to `st` and `model` are updated, while all other variables in `st` remain unchanged.

More About

Tuned Blocks

Tuned blocks, used by the `slTuner` interface, identify blocks in a Simulink model whose parameters are to be tuned to satisfy tuning goals. You can tune most Simulink blocks that represent linear elements such as gains, transfer functions, or state-space models. (For the complete list of blocks that support tuning, see “How Tuned Simulink Blocks Are Parameterized” on page 10-27). You can also tune more complex blocks such as SubSystem or S-Function blocks by specifying an equivalent tunable linear model.

Use tuning commands such as `systune` to tune the parameters of tuned blocks.

You must specify tuned blocks (for example, `C1` and `C2`) when you create an `slTuner` interface.

```
st = slTuner('scdcascade',{'C1','C2'})
```

You can modify the list of tuned blocks using `addBlock` and `removeBlock`.

To interact with the tuned blocks use:

- `getBlockParam`, `getBlockValue`, and `getTunedValue` to access the tuned block parameterizations and their current values.
- `setBlockParam`, `setBlockValue`, and `setTunedValue` to modify the tuned block parameterizations and their values.
- `writeBlockValue` to update the blocks in a Simulink model with the current values of the tuned block parameterizations.

Tuned Variables

Within an `slTuner` interface, tuned variables are any Control Design Blocks involved in the parameterization of a tuned Simulink block, either directly or through a generalized parametric model. Tuned variables are the parameters manipulated by tuning commands such as `systune`.

For Simulink blocks parameterized by a generalized model or a tunable surface:

- `getBlockValue` provides access to the overall value of the block parameterization. To access the values of the tuned variables within the block parameterization, use `getTunedValue`.
- `setBlockValue` cannot be used to modify the block value. To modify the values of tuned variables within the block parameterization, use `setTunedValue`.

For Simulink blocks parameterized by a Control Design Block, the block itself is the tuned variable. To modify the block value, you can use either `setBlockValue` or `setTunedValue`. Similarly, you can retrieve the block value using either `getBlockValue` or `getTunedValue`.

See Also

`slTuner` | `getTunedValue` | `setBlockParam` | `setBlockValue` | `writeBlockValue` | `tunableSurface`

Topics

"How Tuned Simulink Blocks Are Parameterized"

Introduced in R2015b

showTunable

Show value of parameterizations of tunable blocks of `slTuner` interface

Syntax

```
showTunable(st)
```

Description

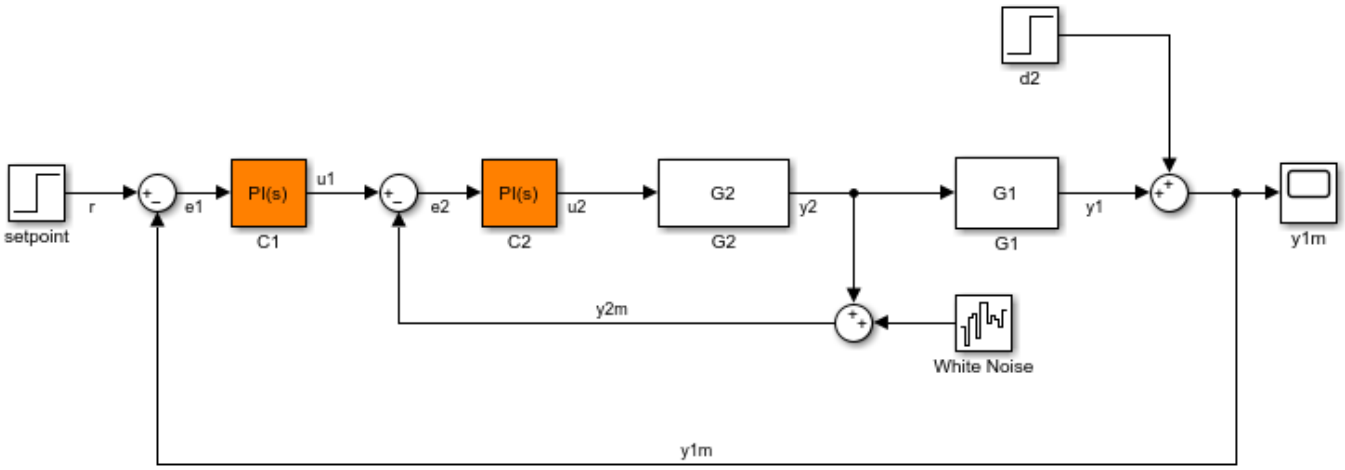
`showTunable(st)` displays the values of the parametric models associated with each tunable block on page 18-370 in the `slTuner` interface, `st`.

Examples

Display Tunable Block Values

Open the Simulink model.

```
mdl = 'scdcascade';
open_system(mdl)
```



Create an `slTuner` interface for the model, and add `C1` and `C2` as tuned blocks of the interface.

```
st = slTuner(mdl,{'C1','C2'});
```

Display the default values of the tuned blocks.

```
showTunable(st);
```

```
Block 1: scdcascade/C1 =

             1
```

```
  Kp + Ki * ---
             s

  with Kp = 0.158, Ki = 0.042

Name: C1
Continuous-time PI controller in parallel form.

----------------------------------

Block 2: scdcascade/C2 =

             1
  Kp + Ki * ---
             s

  with Kp = 1.48, Ki = 4.76

Name: C2
Continuous-time PI controller in parallel form.
```

Input Arguments

`st` — Interface for tuning control systems modeled in Simulink

`slTuner` interface

Interface for tuning control systems modeled in Simulink, specified as an `slTuner` interface.

More About

Tuned Blocks

Tuned blocks, used by the `slTuner` interface, identify blocks in a Simulink model whose parameters are to be tuned to satisfy tuning goals. You can tune most Simulink blocks that represent linear elements such as gains, transfer functions, or state-space models. (For the complete list of blocks that support tuning, see “How Tuned Simulink Blocks Are Parameterized” on page 10-27). You can also tune more complex blocks such as SubSystem or S-Function blocks by specifying an equivalent tunable linear model.

Use tuning commands such as `systune` to tune the parameters of tuned blocks.

You must specify tuned blocks (for example, `C1` and `C2`) when you create an `slTuner` interface.

```
st = slTuner('scdcascade',{'C1','C2'})
```

You can modify the list of tuned blocks using `addBlock` and `removeBlock`.

To interact with the tuned blocks use:

- `getBlockParam`, `getBlockValue`, and `getTunedValue` to access the tuned block parameterizations and their current values.
- `setBlockParam`, `setBlockValue`, and `setTunedValue` to modify the tuned block parameterizations and their values.

- `writeBlockValue` to update the blocks in a Simulink model with the current values of the tuned block parameterizations.

See Also

`slTuner` | `writeBlockValue` | `getBlockValue` | `setBlockValue`

Introduced in R2014a

systune

Tune control system parameters in Simulink using `slTuner` interface

Syntax

```
[st,fSoft] = systune(st0,SoftGoals)
[st,fSoft,gHard] = systune(st0,SoftGoals,HardGoals)
[st,fSoft,gHard] = systune( ___ ,opt)
[st,fSoft,gHard,info] = systune( ___ )
```

Description

`systune` tunes fixed-structure control systems subject to both soft and hard design goals. `systune` can tune multiple fixed-order, fixed-structure control elements distributed over one or more feedback loops. For an overview of the tuning workflow, see “Automated Tuning Workflow” on page 10-6.

This command tunes control systems modeled in Simulink. For tuning control systems represented in MATLAB, use `systune` for `genss` models.

`[st,fSoft] = systune(st0,SoftGoals)` tunes the free parameters of the control system in Simulink. The Simulink model, tuned blocks on page 18-379, and analysis points on page 18-379 of interest are specified by the `slTuner` interface, `st0`. `systune` tunes the control system parameters to best meet the performance goals, `SoftGoals`. The command returns a tuned version of `st0` as `st`. The best achieved soft constraint values are returned as `fSoft`.

If the `st0` contains real parameter uncertainty, `systune` automatically performs robust tuning to optimize the constraint values for worst-case parameter values. `systune` also performs robust tuning against a set of plant models obtained at different operating points or parameter values. See “Input Arguments” on page 18-374.

Tuning is performed at the sample time specified by the `Ts` property of `st0`.

`[st,fSoft,gHard] = systune(st0,SoftGoals,HardGoals)` tunes the control system to best meet the soft goals, subject to satisfying the hard goals. It returns the best achieved values, `fSoft` and `gHard`, for the soft and hard goals. A goal is met when its achieved value is less than 1.

`[st,fSoft,gHard] = systune( ___ ,opt)` specifies options for the optimization for any of the input argument combinations in previous syntaxes.

`[st,fSoft,gHard,info] = systune( ___ )` also returns detailed information about each optimization run for any of the input argument combinations in previous syntaxes.

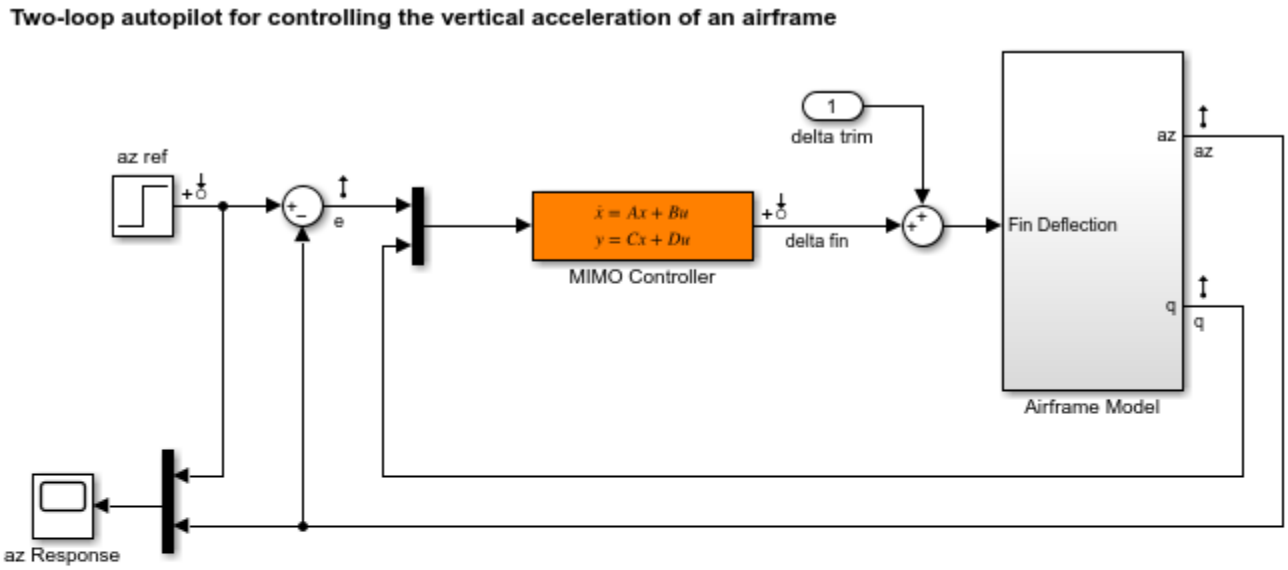
Examples

Tune Control System to Soft Constraints

Tune the control system in the `rct_airframe2` model to soft goals for tracking, roll off, stability margin, and disturbance rejection.

Open the Simulink model.

```
mdl = 'rct_airframe2';
open_system(mdl);
```



Create and configure an `slTuner` interface to the model.

```
st0 = slTuner(mdl,'MIMO Controller');
```

`st0` is an `slTuner` interface to the `rct_aircraft2` model with the `MIMO Controller` block specified as the tunable portion of the control system.

The model already has linearization input points on the signals `az ref`, `delta fin`, `az`, `q`, and `e`. These signals are therefore available as analysis points for tuning goals and linearization.

Specify the tracking requirement, roll-off requirement, stability margins, and disturbance rejection requirement.

```
req1 = TuningGoal.Tracking('az ref','az',1);
req2 = TuningGoal.Gain('delta fin','delta fin',tf(25,[1 0]));
req3 = TuningGoal.Margins('delta fin',7,45);
```

```
max_gain = frd([2 200 200],[0.02 2 200]);
req4 = TuningGoal.Gain('delta fin','az',max_gain);
```

`req1` constrains `az` to track `az ref`. The next requirement, `req2`, imposes a roll-off requirement by specifying a gain profile for the open-loop, point-to-point transfer function measured at `delta fin`. The next requirement, `req3`, imposes open-loop gain and phase margins on that same point-to-point transfer function. Finally, `req4` rejects disturbances to `az` injected at `delta fin`, by specifying a maximum gain profile between those two points.

Tune the model using these tuning goals.

```
opt = systuneOptions('RandomStart',3);
rng(0);
[st,fSoft,~,info] = systune(st0,[req1,req2,req3,req4],opt);
```

```
Final: Soft = 1.13, Hard = -Inf, Iterations = 88
Final: Soft = 1.13, Hard = -Inf, Iterations = 78
Final: Soft = 1.13, Hard = -Inf, Iterations = 67
Final: Soft = 40, Hard = -Inf, Iterations = 85
```

`st` is a tuned version of `st0`.

The `RandomStart` option specifies that `systune` must perform three independent optimization runs that use different (random) initial values of the tunable parameters. These three runs are in addition to the default optimization run that uses the current value of the tunable parameters as the initial value. The call to `rng` seeds the random number generator to produce a repeatable sequence of numbers.

`systune` displays the final result for each run. The displayed value, `Soft`, is the maximum of the values achieved for each of the four performance goals. The software chooses the best run overall, which is the run yielding the lowest value of `Soft`. The last run fails to achieve closed-loop stability, which corresponds to `Soft = Inf`.

Examine the best achieved values of the soft constraints.

```
fSoft
```

```
fSoft =

    1.1327    1.1327    0.5140    1.1327
```

Only `req3`, the stability margin requirement, is met for all frequencies. The other values are close to, but exceed, 1, indicating violations of the goals for at least some frequencies.

Use `viewGoal` to visualize the tuned control system performance against the goals and to determine whether the violations are acceptable. To evaluate specific open-loop or closed-loop transfer functions for the tuned parameter values, you can use linearization commands such as `getIOTransfer` and `getLoopTransfer`. After validating the tuned parameter values, if you want to apply these values to the Simulink® model, you can use `writeBlockValue`.

Input Arguments

`st0` — Interface for tuning control systems modeled in Simulink

`slTuner` interface

Interface for tuning control systems modeled in Simulink, specified as an `slTuner` interface.

If you specify parameter variation or linearization at multiple operating points when you create `st0`, then `systune` performs robust tuning against all the plant models. If you specify an uncertain (`uss`) model as a block substitution when you create `st0`, then `systune` performs robust tuning, optimizing the parameters against the worst-case parameter values. For more information about robust tuning approaches, see “Robust Tuning Approaches” (Robust Control Toolbox). (Using uncertain models requires a Robust Control Toolbox license.)

`SoftGoals` — Soft goals (objectives)
vector of `TuningGoal` objects

Soft goals (objectives) for tuning the control system described by `st0`, specified as a vector of `TuningGoal` objects. For a complete list, see “Tuning Goals”.

`systune` tunes the tunable parameters of the control system to minimize the maximum value of the soft tuning goals, subject to satisfying the hard tuning goals (if any).

`HardGoals` — Hard goals (constraints)
vector of `TuningGoal` objects

Hard goals (constraints) for tuning the control system described by `st0`, specified as a vector of `TuningGoal` objects. For a complete list, see “Tuning Goals”.

A hard goal is satisfied when its value is less than 1. `systune` tunes the tunable parameters of the control system to minimize the maximum value of the soft tuning goals, subject to satisfying all the hard tuning goals.

`opt` — Tuning algorithm options
options set created using `systuneOptions`

Tuning algorithm options, specified as an options set created using `systuneOptions`.

Available options include:

- Number of additional optimizations to run starting from random initial values of the free parameters
- Tolerance for terminating the optimization
- Flag for using parallel processing

See the `systuneOptions` reference page for more details about all available options.

Output Arguments

`st` — Tuned interface
`slTuner` interface

Tuned interface, returned as an `slTuner` interface.

`fSoft` — Best achieved values of soft goals
vector

Best achieved values of soft goals, returned as a vector.

Each tuning goal evaluates to a scalar value, and `systune` minimizes the maximum value of the soft goals, subject to satisfying all the hard goals.

`fSoft` contains the value of each soft goal for the best overall run. The best overall run is the run that achieved the smallest value for `max(fSoft)`, subject to `max(gHard)<1`.

`gHard` — Achieved values of hard goals
vector

Achieved values of hard goals, returned as a vector.

`gHard` contains the value of each hard goal for the best overall run (the run that achieved the smallest value for `max(fSoft)`, subject to `max(gHard)<1`. All entries of `gHard` are less than 1 when all hard goals are satisfied. Entries greater than 1 indicate that `systune` could not satisfy one or more design constraints.

`info` — Detailed information about optimization runs
structure

Detailed information about each optimization run, returned as a data structure. The fields of `info` are summarized in the following table.

Field	Value
`Run`	Run number, returned as a scalar. If you use the `RandomStart` option of `systuneOptions` to perform multiple optimization runs, `info` is a structure array, and `info.Run` is the index.
`Iterations`	Total number of iterations performed during the run, returned as a scalar. If you use `RandomStart`, `info.Iterations(j)` is the number of iterations performed in the *j*th run before termination.
`f`	Best overall soft constraint value, returned as a scalar. `systune` converts the soft tuning goals to a function of the free parameters of the control system. The command then tunes the parameters to minimize that function subject to the hard goals. (See “Algorithms” on page 18-380.) `info.f` is the maximum soft goal value at the final iteration. This value is meaningful only when the hard goals are satisfied. If the value is less than 1, then the soft goals are also attained.
`g`	Best overall hard constraint value, returned as a scalar. `systune` converts the hard tuning goals to a function of the free parameters of the control system. The command then tunes the parameters to drive those values below 1. (See “Algorithms” on page 18-380.) `info.g` is the largest hard goal value at the final iteration. If this value is less than 1, then the hard goals are satisfied.
`x`	Tuned parameter values, returned as a vector. This vector contains the values of the tunable parameters at the end of the run. `info.x` can also include the values of additional variables such as loop scalings, if `systune` uses them (see `info.LoopScaling`).

Field	Value
`MinDecay`	Minimum decay rate of tuned system dynamics, returned as a two-element row vector. `info.MinDecay(1)` is the minimum decay rate of the closed-loop poles. `info.MinDecay(2)` is the minimum decay rate of the dynamics of tuned blocks with stability constraints. For more information about stabilized dynamics and decay rates, see the `MinDecay` option of `systuneOptions`.
`fSoft`	Individual soft constraint values, returned as a vector. `systune` converts each soft tuning goal 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 goals. (See “Algorithms” on page 18-380.) `info.fSoft` contains the individual values of the soft goals at the end of each run. These values appear in `fSoft` in the same order in which you specify goals in the `SoftReqs` input argument to `systune`.
`gHard`	Individual hard constraint values, returned as a vector. `systune` converts each hard tuning goal 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 goal is satisfied if its value is less than 1. (See “Algorithms” on page 18-380.) `info.gHard` contains the individual values of the hard goals at the end of each run. These values appear in `gHard` in the same order in which you specify goals in the `HardReqs` input argument to `systune`.
`Blocks`	Tuned values of tunable blocks and parameters in the tuned control system, returned as a structure whose fields are the names of tunable elements and whose values are the corresponding tuned values. When you perform multiple runs by setting the `RandomStart` option to a positive value, you can use this field to examine control system performance with the results from other runs. For instance, use the following code to apply the tuned values from the *j*th run. `stj = setBlockValue(st0,info(j).Blocks)`

Field	Value
`LoopScaling`	Optimal diagonal scaling for evaluating MIMO tuning requirements, returned as a state-space model. When applied to multiloop control systems, tuning goals that involve an open-loop response can be sensitive to the scaling of the loop transfer functions to which they apply. This sensitivity can lead to poor optimization results. `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`. Tuning goals affected by such loop scaling include: • `TuningGoal.LoopShape` • `TuningGoal.MinLoopGain` and `TuningGoal.MaxLoopGain` • `TuningGoal.Sensitivity` • `TuningGoal.Rejection` • `TuningGoal.Margins`

`info` also contains the following fields, whose entries are meaningful when you use `systune` for robust tuning of control systems with uncertainty.

Field	Value
`wcPert`	Worst combinations of uncertain parameters, returned as a structure array. Each structure contains one set of uncertain parameter values. The perturbations with the worst performance are listed first.
`wcf`	Worst soft-goal value, returned as a scalar. This value is the largest soft goal value (`f`) over the uncertainty range when using the tuned controller.
`wcg`	Worst hard-goal value, returned as a scalar. This value is the largest hard goal value (`g`) over the uncertainty range when using the tuned controller.
`wcDecay`	Smallest closed-loop decay rate over the uncertainty range when using the tuned controller, returned as a scalar. A positive value indicates robust stability. For more information about stabilized dynamics and decay rates, see the `MinDecay` option of `systuneOptions`.

More About

Tuned Blocks

Tuned blocks, used by the `slTuner` interface, identify blocks in a Simulink model whose parameters are to be tuned to satisfy tuning goals. You can tune most Simulink blocks that represent linear elements such as gains, transfer functions, or state-space models. (For the complete list of blocks that support tuning, see “How Tuned Simulink Blocks Are Parameterized” on page 10-27). You can also tune more complex blocks such as SubSystem or S-Function blocks by specifying an equivalent tunable linear model.

Use tuning commands such as `systune` to tune the parameters of tuned blocks.

You must specify tuned blocks (for example, `C1` and `C2`) when you create an `slTuner` interface.

```
st = slTuner('scdcascade',{'C1','C2'})
```

You can modify the list of tuned blocks using `addBlock` and `removeBlock`.

To interact with the tuned blocks use:

- `getBlockParam`, `getBlockValue`, and `getTunedValue` to access the tuned block parameterizations and their current values.
- `setBlockParam`, `setBlockValue`, and `setTunedValue` to modify the tuned block parameterizations and their values.
- `writeBlockValue` to update the blocks in a Simulink model with the current values of the tuned block parameterizations.

Analysis Points

Analysis points, used by the `slLinearizer` and `slTuner` interfaces, identify locations within a model that are relevant for linear analysis and control system tuning. You use analysis points as inputs to the linearization commands, such as `getIOTransfer`, `getLoopTransfer`, `getSensitivity`, and `getCompSensitivity`. As inputs to the linearization commands, analysis points can specify any open-loop or closed-loop transfer function in a model. You can also use analysis points to specify design requirements when tuning control systems using commands such as `systune`.

Location refers to a specific block output port within a model or to a bus element in such an output port. For convenience, you can use the name of the signal that originates from this port to refer to an analysis point.

You can add analysis points to an `slLinearizer` or `slTuner` interface, `s`, when you create the interface. For example:

```
s = slLinearizer('scdcascade',{'u1','y1'});
```

Alternatively, you can use the `addPoint` command.

To view all the analysis points of `s`, type `s` at the command prompt to display the interface contents. For each analysis point of `s`, the display includes the block name and port number and the name of the signal that originates at this point. You can also programmatically obtain a list of all the analysis points using `getPoints`.

For more information about how you can use analysis points, see “Mark Signals of Interest for Control System Analysis and Design” on page 2-38 and “Mark Signals of Interest for Batch Linearization” on page 3-9.

Algorithms

x is the vector of tunable parameters in the control system to tune. `systune` converts each soft and hard tuning requirement `SoftReqs(i)` and `HardReqs(j)` into normalized values $f_i(x)$ and $g_j(x)$, respectively. `systune` then solves the constrained minimization problem:

Minimize $\max_i f_i(x)$ subject to $\max_j g_j(x) < 1$, for $x_{\min} < x < x_{\max}$.

x_{min} and x_{max} are the minimum and maximum values of the free parameters of the control system.

When you use both soft and hard tuning goals, the software approaches this optimization problem by solving a sequence of unconstrained subproblems of the form:

$$\min_x \max(\alpha f(x), g(x)) .$$

The software adjusts the multiplier α so that the solution of the subproblems converges to the solution of the original constrained optimization problem.

`systune` returns the `slTuner` interface with parameters tuned to the values that best solve the minimization problem. `systune` also returns the best achieved values of $f_i(x)$ and $g_j(x)$, as `fSoft` and `gHard` respectively.

For information about the functions $f_i(x)$ and $g_j(x)$ for each type of constraint, see the reference pages for each `TuningGoal` requirement object.

`systune` uses the nonsmooth optimization algorithms described in [1],[2],[3],[4]

`systune` computes the H_∞ norm using the algorithm of [5] and structure-preserving eigensolvers from the SLICOT library. For information about the SLICOT library, see http://slicot.org.

Alternative Functionality

Tune interactively using **Control System Tuner**.

References

[1] P. Apkarian and D. Noll, "Nonsmooth H-infinity Synthesis," *IEEE Transactions on Automatic Control*, Vol. 51, Number 1, 2006, pp. 71–86.

[2] Apkarian, P. and D. Noll, "Nonsmooth Optimization for Multiband Frequency-Domain Control Design," *Automatica*, 43 (2007), pp. 724–731.

[3] Apkarian, P., P. Gahinet, and C. Buhr, "Multi-model, multi-objective tuning of fixed-structure controllers," *Proceedings ECC* (2014), pp. 856–861.

[4] Apkarian, P., M.-N. Dao, and D. Noll, "Parametric Robust Structured Control Design," *IEEE Transactions on Automatic Control*, 2015.

[5] Bruinsma, N.A., and M. Steinbuch. "A Fast Algorithm to Compute the H_∞ Norm of a Transfer Function Matrix." *Systems & Control Letters*, 14, no.4 (April 1990): 287–93.

Extended Capabilities

Automatic Parallel Support

Accelerate code by automatically running computation in parallel using Parallel Computing Toolbox™.

To run in parallel, set `'UseParallel'` to `true` using `systuneOptions`.

See Also

`systune (for genss)` | `systuneOptions` | `slTuner` | `addPoint` | `getIOTransfer` | `getLoopTransfer` | `writeBlockValue` | `looptune` | `hinfstruct`

Topics

“Tune Control Systems in Simulink”
“Control of a Linear Electric Actuator”
“Interpret Numeric Tuning Results” on page 10-139
“Tuning Goals”
“Robust Tuning Approaches” (Robust Control Toolbox)

Introduced in R2014a

writeBlockValue

Update block values in Simulink model

Syntax

```
writeBlockValue(st)
writeBlockValue(st,blockid)
writeBlockValue(st,m)
```

Description

`writeBlockValue(st)` writes tuned parameter values from the `slTuner` interface, `st`, to the Simulink model that `st` describes. Use this command, for example, to validate parameters of a control system that you tuned using `systune` or `looptune`.

`writeBlockValue` skips blocks that cannot represent their tuned value in a straightforward and lossless manner. For example, suppose you tune an user defined Subsystem or S-Function block. `writeBlockValue` will skip this block because there is no clear way to map the tuned value to a Subsystem or S-Function block. Similarly, if you parameterize a Gain block as a second-order transfer function, `writeBlockValue` will skip this block, unless the transfer function value is a static gain.

`writeBlockValue(st,blockid)` only updates the block or blocks referenced by `blockid`.

`writeBlockValue(st,m)` writes tuned parameter values from a generalized model, `m`, to the Simulink model described by the `slTuner` interface, `st`.

Examples

Update Simulink Model with All Tuned Parameters

Create an `slTuner` interface for the model.

```
st = slTuner('scdcascade',{'C1','C2'});
```

Specify the tuning goals and necessary analysis points.

```
tg1 = TuningGoal.StepTracking('r','y1m',5);

addPoint(st,{'r','y1m'});

tg2 = TuningGoal.Poles();
tg2.MaxFrequency = 10;
```

Tune the controller.

```
[sttuned,fSoft] = systune(st,[tg1 tg2]);
```

```
Final: Soft = 1.28, Hard = -Inf, Iterations = 37
```

After validating the tuning results, update the model to use the tuned controller values.

```
writeBlockValue(sttuned);
```

Input Arguments

`st` — Interface for tuning control systems modeled in Simulink

`slTuner` interface

Interface for tuning control systems modeled in Simulink, specified as an `slTuner` interface.

`blockid` — Blocks to update

character vector | string | cell array of character vectors | string array

Blocks to update with tuned values, specified as a:

- Character vector or string, to update one block.
- Cell array of character vectors or string array, to update multiple blocks.

The blocks in `blockid` must be in the `TunedBlocks` property of the `slTuner` interface `st`. You can specify a full block path, or any portion of the block path that uniquely identifies the block among the other tuned blocks of `st`.

Example: `blk = {'scdcascade/C1','scdcascade/C2'}`

Example: `"C1"`

`m` — Tuned control system

generalized state-space

Tuned control system, specified as a generalized state-space model (`genss`).

Typically, `m` is the output of a tuning function like `systune`, `looptune`, or `hinfstruct`. The model `m` must have some tunable parameters in common with `st`. For example, `m` can be a generalized model that you obtained by linearizing your Simulink model, and then tuned to meet some design requirements.

See Also

`slTuner` | `getBlockValue` | `setBlockValue` | `showTunable` | `writeLookupTableData`

Topics

“Tuning of a Digital Motion Control System”
“Control of a Linear Electric Actuator”
“How Tuned Simulink Blocks Are Parameterized”

Introduced in R2014a

slTuner

Interface for control system tuning of Simulink models

Syntax

```
st = slTuner(mdl,tuned_blocks)
st = slTuner(mdl,tuned_blocks,pt)
st = slTuner(mdl,tuned_blocks,param)
st = slTuner(mdl,tuned_blocks,op)
st = slTuner(mdl,tuned_blocks,blocksub)
st = slTuner(mdl,tuned_blocks,options)
st = slTuner(mdl,tuned_blocks,pt,op,param,blocksub,options)
```

Description

`st = slTuner(mdl,tuned_blocks)` creates an `slTuner` interface, `st`, for tuning the control system blocks of the Simulink model, `mdl`. The interface adds the linear analysis points marked in the model as analysis points on page 18-394 of `st`. The interface also adds the linear analysis points that imply an opening as permanent openings on page 18-395. When the interface performs linearization, for example, to tune the blocks, it uses the model initial condition as the operating point.

`st = slTuner(mdl,tuned_blocks,pt)` adds the specified point to the list of analysis points for `st`, ignoring linear analysis points marked in the model.

`st = slTuner(mdl,tuned_blocks,param)` specifies the parameters whose values you want to vary when tuning the model blocks.

`st = slTuner(mdl,tuned_blocks,op)` specifies the operating points for tuning the model blocks.

`st = slTuner(mdl,tuned_blocks,blocksub)` specifies substitute linearizations of blocks and subsystems. Use this syntax, for example, to specify a custom linearization for a block. You can also use this syntax for blocks that do not linearize successfully, such as blocks with discontinuities or triggered subsystems.

`st = slTuner(mdl,tuned_blocks,options)` configures the linearization algorithm options.

`st = slTuner(mdl,tuned_blocks,pt,op,param,blocksub,options)` uses any combination of the input arguments `pt`, `op`, `param`, `blocksub`, and `options` to create `st`. For example, you can use:

- `st = slTuner(mdl,tuned_blocks,pt,param)`
- `st = slTuner(mdl,tuned_blocks,op,param)`.

Object Description

`slTuner` provides an interface between a Simulink model and the tuning commands `systune` and `looptune`. `slTuner` allows you to:

- Specify the control architecture.
- Designate and parameterize blocks to be tuned.
- Tune the control system.
- Validate design by computing (linearized) open-loop and closed-loop responses.
- Write tuned values back to the model.

Because tuning commands such as `systune` operate on linear models, the `slTuner` interface automatically computes and stores a linearization of your Simulink model. This linearization is automatically updated when you change any properties of the `slTuner` interface. The update occurs when you call commands that query the linearization stored in the interface. Such commands include `systune`, `looptune`, `getIOTransfer`, and `getLoopTransfer`. For more information about linearization, see “What Is Linearization?” on page 2-3

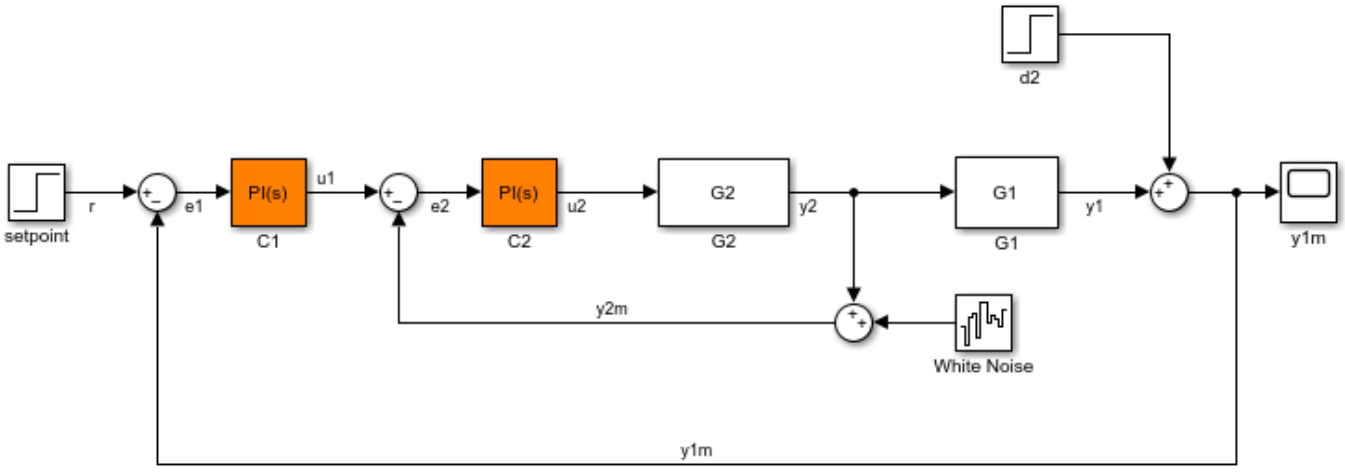
Examples

Create and Configure slTuner Interface for Control System Tuning

Create and configure an `slTuner` interface for a Simulink® model that specifies which blocks to tune with `systune` or `looptune`.

Open the Simulink model.

```
mdl = 'scdcascade';
open_system(mdl)
```



The control system consists of two feedback loops, an inner loop with PI controller `C2`, and an outer loop with PI controller `C1`. Suppose you want to tune this model to meet the following control objectives:

- Track setpoint changes to `r` at the system output `y1m` with zero steady-state error and specified rise time.
- Reject the disturbance represented by `d2`.

The `systune` command can jointly tune the controller blocks to meet these design requirements, which you specify using `TuningGoal` objects. The `slTuner` interface sets up this tuning task.

Create an `slTuner` interface for the model.

```
st = slTuner(mdl,{'C1','C2'});
```

This command initializes the `slTuner` interface and designates the two PI controller blocks as tunable. Each tunable block is automatically parameterized according to its type and initialized with its value in the Simulink model. A linearization of the remaining, nontunable portion of the model is computed and stored in the `slTuner` interface.

To configure the `slTuner` interface, designate as analysis points any signal locations of relevance to your design requirements. Add the output and reference input for the tracking requirement. Also, add the disturbance-rejection location.

```
addPoint(st,{'r','y1m','d2'});
```

These locations in your model are now available for referencing in `TuningGoal` objects that capture your design goals.

Display a summary of the `slTuner` interface configuration in the command window.

```
st
```

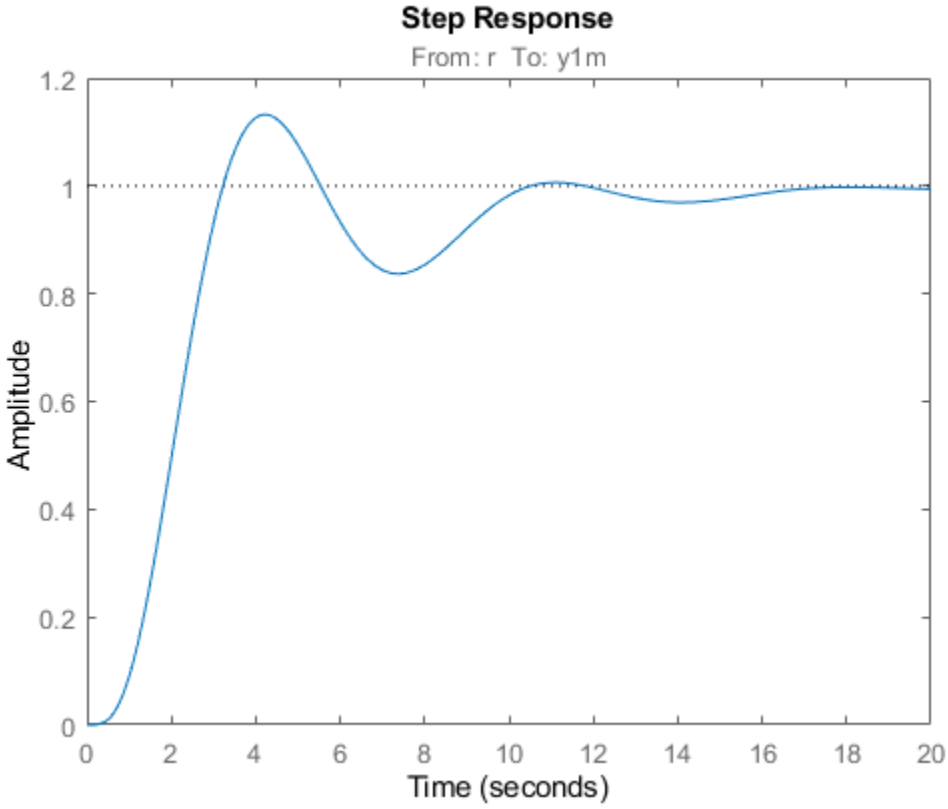
```
slTuner tuning interface for "scdcascade":

2 Tuned blocks: (Read-only TunedBlocks property)
-------------------------
Block 1: scdcascade/C1
Block 2: scdcascade/C2

3 Analysis points:
-------------------------
Point 1: Signal "r", located at 'Output Port 1' of scdcascade/setpoint
Point 2: Signal "y1m", located at 'Output Port 1' of scdcascade/Sum
Point 3: 'Output Port 1' of scdcascade/d2

No permanent openings. Use the addOpening command to add new permanent openings.
Properties with dot notation get/set access:
      Parameters         : []
      OperatingPoints    : [] (model initial condition will be used.)
      BlockSubstitutions : []
      Options            : [1x1 linearize.SlTunerOptions]
      Ts                 : 0
```

The display lists the designated tunable blocks, analysis points, and other information about the interface. In the command window, click on any highlighted signal to see its location in the Simulink model. Note that specifying the block name `'d2'` in the `addPoint` command is equivalent to designating that block's single output signal as the analysis point.

You can now capture your design goals with `TuningGoal` objects and use `systune` or `looptune` to tune the control system to meet those design goals.

In addition to specifying design goals, you can use analysis points for extracting system responses. For example, extract and plot the step response between the reference signal `'r'` and the output `'y1m'`.

```
T = getIOTransfer(st,'r','y1m');
stepplot(T)
```



Input Arguments

`mdl` — Simulink model name

character vector | string

Simulink model name, specified as a character vector or string.

Example: `'scdcascade'`

`tuned_blocks` — Blocks to be tuned

character vector | string | cell array of character vectors | string array

Blocks to be added to the list of tuned blocks of `st`, specified as:

- Character vector or string — Block path. You can specify the full block path or a partial path. The partial path must match the end of the full block path and unambiguously identify the block to add. For example, you can refer to a block by its name, provided the block name appears only once in the Simulink model.

  For example, `blk = 'scdcascade/C1'`.

- Cell array of character vectors or string array — Multiple block paths.

  For example, `blk = {'scdcascade/C1','scdcascade/C2'}`.

`pt` — Analysis point
character vector | string | cell array of character vectors | string array | vector of linearization I/O objects

Analysis point on page 18-394 to be added to the list of analysis points for `st`, specified as:

- Character vector or string — Analysis point identifier that can be any of the following:
  - Signal name, for example `pt = 'torque'`
  - Block path for a block with a single output port, for example `pt = 'Motor/PID'`
  - Block path and port originating the signal, for example `pt = 'Engine Model/1'`
- Cell array of character vectors or string array — Specifies multiple analysis point identifiers. For example:

  ```
  pt = {'torque','Motor/PID','Engine Model/1'}
  ```
- Vector of linearization I/O objects — Create `pt` using `linio`. For example:

  ```
  pt(1) = linio('scdcascade/setpoint',1,'input');
  pt(2) = linio('scdcascade/Sum',1,'output');
  ```

  Here, `pt(1)` specifies an input, and `pt(2)` specifies an output.

  The interface adds all the points specified by `pt` and ignores their I/O types. The interface also adds all `'loopbreak'` type signals as permanent openings on page 18-395.

`param` — Parameter samples
structure | structure array

Parameter samples for linearizing `mdl`, specified as:

- Structure — Vary the value of a single parameter by specifying `param` as a structure with the following fields:
  - `Name` — Parameter name, specified as a character vector or string. You can specify any model parameter that is a variable in the model workspace, the MATLAB workspace, or a data dictionary. If the variable used by the model is not a scalar variable, specify the parameter name as an expression that resolves to a numeric scalar value. For example, to use the first element of vector `V` as a parameter, use:

    ```
    param.Name = 'V(1)';
    ```
  - `Value` — Parameter sample values, specified as a double array.

  For example, vary the value of parameter `A` in the 10% range:

  ```
  param.Name = 'A';
  param.Value = linspace(0.9*A,1.1*A,3);
  ```
- Structure array — Vary the value of multiple parameters. For example, vary the values of parameters `A` and `b` in the 10% range:

  ```
  [A_grid,b_grid] = ndgrid(linspace(0.9*A,1.1*A,3),...
                           linspace(0.9*b,1.1*b,3));
  ```

```
params(1).Name = 'A';
params(1).Value = A_grid;
params(2).Name = 'b';
params(2).Value = b_grid;
```

For more in information, see “Specify Parameter Samples for Batch Linearization” on page 3-43.

If `param` specifies tunable parameters only, then the software batch linearizes the model using a single compilation. If you also configure `st.OperatingPoints` with operating point objects only, the software uses single model compilation.

For an example showing how batch linearization with parameter sampling works, see “Vary Parameter Values and Obtain Multiple Transfer Functions” on page 3-21. That example uses `slLinearizer`, but the process is the same for `slTuner`.

To compute the offsets required by the LPV System block, specify `param`, and set `st.Options.StoreOffsets` to `true`. You can then return additional linearization information when calling linearization functions such as `getIOTransfer`, and extract the offsets using `getOffsetsForLPV`.

`op` — Operating point for linearizing `mdl`

`OperatingPoint` object | array of `OperatingPoint` objects | vector of positive scalars | `OperatingReport` object | array of `OperatingReport` objects

Operating point for linearizing `mdl`, specified as:

- `OperatingPoint` or `OperatingReport` object, created using `findop` with either a single operating point specification, or a single snapshot time.
- Array of `OperatingPoint` or `OperatingReport` objects, specifying multiple operating points.

  To create an array of `OperatingPoint` or `OperatingReport` objects, you can:

  - Extract operating points at multiple snapshot times using `findop`.
  - Batch trim your model using multiple operating point specifications. For more information, see “Batch Compute Steady-State Operating Points for Multiple Specifications” on page 1-68.
  - Batch trim your model using parameter variations. For more information, see “Batch Compute Steady-State Operating Points for Parameter Variation” on page 1-72.
- Vector of positive scalars, specifying simulation snapshot times.

If you configure `st.Parameters`, then specify `op` as one of the following:

- Single `OperatingPoint` or `OperatingReport` object.
- Array of `OperatingPoint` or `OperatingReport` objects whose size matches that of the parameter grid specified by the `Parameters` property. When you batch linearize `mdl`, the software uses only one model compilation. To obtain the operating points that correspond to the parameter value combinations, batch trim your model using `param` before linearization. For an example that uses the `linearize` command, see “Batch Linearize Model at Multiple Operating Points Derived from Parameter Variations” on page 3-16.
- Multiple snapshot times. When you batch linearize `mdl`, the software simulates the model for each snapshot time and parameter grid point combination. This operation can be computationally expensive.

`blocksub` — Substitute linearizations for blocks and model subsystems

structure | structure array

Substitute linearizations for blocks and model subsystems, specified as a structure or an *n*-by-1 structure array, where *n* is the number of blocks for which you want to specify a linearization. Use `blocksub` to specify a custom linearization for a block or subsystem. For example, you can specify linearizations for blocks that do not have analytic linearizations, such as blocks with discontinuities or triggered subsystems.

To study the effects of varying the linearization of a block on the model dynamics, you can batch linearize your model by specifying multiple substitute linearizations for a block.

If you substitute a linearization with a sample time that differs from that of the original block or subsystem, it is best practice to set the overall linearization sample time (`options.SampleTime`) to a nondefault value.

Each substitute linearization structure has the following fields.

`Name` — Block path

character vector | string

Block path of the block for which you want to specify the linearization, specified as a character vector or string.

`Value` — Substitute linearization

double | double array | LTI model | model array | structure

Substitute linearization for the block, specified as one of the following:

- Double — Specify the linearization of a SISO block as a gain.
- Array of doubles — Specify the linearization of a MIMO block as an n_u-by-n_y array of gain values, where n_u is the number of inputs and n_y is the number of outputs.
- LTI model, uncertain state-space model, or uncertain real object — The I/O configuration of the specified model must match the configuration of the block specified by `Name`. Using an uncertain model requires Robust Control Toolbox software.
- Array of LTI models, uncertain state-space models, or uncertain real objects — Batch linearize the model using multiple block substitutions. The I/O configuration of each model in the array must match the configuration of the block for which you are specifying a custom linearization. If you:
  - Vary model parameters using `param` and specify `Value` as a model array, the dimensions of `Value` must match the parameter grid size.
  - Define block substitutions for multiple blocks, and specify `Value` as an array of LTI models for more than one block, the dimensions of the arrays must match.
- Structure with the following fields:

Field	Description
`Specification`	Block linearization, specified as a character vector that contains one of the following • MATLAB expression • Name of a "Custom Linearization Function" on page 18-395 in your current working folder or on the MATLAB path. The specified expression or function must return one of the following: • Linear model in the form of a D-matrix • Control System Toolbox LTI model object • Uncertain state-space model or uncertain real object (requires Robust Control Toolbox software) The I/O configuration of the returned model must match the configuration of the block specified by `Name`.
`Type`	Specification type, specified as one of the following: • `'Expression'` • `'Function'`
`ParameterNames`	Linearization function parameter names, specified as a cell array of character vectors. Specify `ParameterNames` only when `Type = 'Function'` and your block linearization function requires input parameters. These parameters only impact the linearization of the specified block. You must also specify the corresponding `blocksub.Value.ParameterValues` field.
`ParameterValues`	Linearization function parameter values, specified as a vector of doubles. The order of parameter values must correspond to the order of parameter names in `blocksub.Value.ParameterNames`. Specify `ParameterValues` only when `Type = 'Function'` and your block linearization function requires input parameters.

`options` — `slTuner` options
`slTunerOptions` option set

`slTuner` options, specified as an `slTunerOptions` option set.

Example: `options = slTunerOptions('IgnoreDiscreteStates','on')`

Properties

`slTuner` object properties include:

`TunedBlocks`

Blocks to be tuned in `mdl`, specified as a cell array of character vectors.

When you create an `slTuner` interface, the `TunedBlocks` property is automatically populated with the blocks you specify in the `tuned_blocks` input argument. To specify additional tunable blocks in an existing `slTuner` interface, use `addBlock`.

`Ts`

Sampling time for analyzing and tuning `mdl`, specified as nonnegative scalar.

Set this property using dot notation (`st.Ts = Ts`).

Default: 0 (implies continuous-time)

`Parameters`

Parameter samples for linearizing `mdl`, specified as a structure or a structure array.

Set this property using the `param` input argument or dot notation (`st.Parameters = param`). `param` must be one of the following:

If `param` specifies tunable parameters only, then the software batch linearizes the model using a single compilation. If you also configure `st.OperatingPoints` with operating point objects only, the software uses single model compilation.

`OperatingPoints`

Operating points for linearizing `mdl`, specified as an `OperatingPoint` object, array of `OperatingPoint` objects, or array of positive scalars.

Set this property using the `op` input argument or dot notation (`st.OperatingPoints = op`). `op` must be one of the following:

- `OperatingPoint` or `OperatingReport` object, created using `findop` with either a single operating point specification, or a single snapshot time.
- Array of `OperatingPoint` or `OperatingReport` objects, specifying multiple operating points.

  To create an array of `OperatingPoint` or `OperatingReport` objects, you can:

  - Extract operating points at multiple snapshot times using `findop`.
  - Batch trim your model using multiple operating point specifications. For more information, see “Batch Compute Steady-State Operating Points for Multiple Specifications” on page 1-68.
  - Batch trim your model using parameter variations. For more information, see “Batch Compute Steady-State Operating Points for Parameter Variation” on page 1-72.
- Vector of positive scalars, specifying simulation snapshot times.

If you configure `st.Parameters`, then specify `op` as one of the following:

- Single `OperatingPoint` or `OperatingReport` object.
- Array of `OperatingPoint` or `OperatingReport` objects whose size matches that of the parameter grid specified by the `Parameters` property. When you batch linearize `mdl`, the software uses only one model compilation. To obtain the operating points that correspond to the parameter value combinations, batch trim your model using `param` before linearization. For an example that uses the `linearize` command, see “Batch Linearize Model at Multiple Operating Points Derived from Parameter Variations” on page 3-16.

- Multiple snapshot times. When you batch linearize `mdl`, the software simulates the model for each snapshot time and parameter grid point combination. This operation can be computationally expensive.

`BlockSubstitutions`

Substitute linearizations for blocks and model subsystems, specified as a structure or structure array.

Use this property to specify a custom linearization for a block or subsystem. You also can use this syntax for blocks that do not have analytic linearizations, such as blocks with discontinuities or triggered subsystems.

Set this property using the `blocksub` input argument or dot notation (`st.BlockSubstitutions = blocksubs`). For information about the required structure, see `blocksub`.

`Options`

Linearization algorithm options, specified as an option set created using `slTunerOptions`.

Set this property using the `opt` input argument or dot notation (`st.Options = opt`).

`Model`

Name of the Simulink model to be linearized, specified as a character vector by the input argument `mdl`.

`TimeUnit`

Unit of the time variable. This property specifies the time units for linearized models returned by `getIOTransfer`, `getLoopTransfer`, `getSensitivity`, and `getCompSensitivity`. Use any of the following values:

- `'nanoseconds'`
- `'microseconds'`
- `'milliseconds'`
- `'seconds'`
- `'minutes'`
- `'hours'`
- `'days'`
- `'weeks'`
- `'months'`
- `'years'`

Default: `'seconds'`

Object Functions

addBlock	Add block to list of tuned blocks for slTuner interface
addOpening	Add signal to list of openings for slLinearizer or slTuner interface
addPoint	Add signal to list of analysis points for slLinearizer or slTuner interface
getPoints	Get list of analysis points for slLinearizer or slTuner interface

getOpenings	Get list of openings for slLinearizer or slTuner interface
getBlockParam	Get parameterization of tuned block in slTuner interface
getBlockValue	Get current value of tuned block parameterization in slTuner interface
getTunedValue	Get current value of tuned variable in slTuner interface
getBlockRateConversion	Get rate conversion settings for tuned block in slTuner interface
setBlockParam	Set parameterization of tuned block in slTuner interface
setBlockValue	Set value of tuned block parameterization in slTuner interface
setBlockRateConversion	Set rate conversion settings for tuned block in slTuner interface
systune	Tune control system parameters in Simulink using slTuner interface
looptune	Tune MIMO feedback loops in Simulink using slTuner interface
loopview	Graphically analyze results of control system tuning using slTuner interface
looptuneSetup	Construct tuning setup for looptune to tuning setup for systune using slTuner interface
showTunable	Show value of parameterizations of tunable blocks of slTuner interface
getIOTransfer	Transfer function for specified I/O set using slLinearizer or slTuner interface
getLoopTransfer	Open-loop transfer function at specified point using slLinearizer or slTuner interface
getSensitivity	Sensitivity function at specified point using slLinearizer or slTuner interface
getCompSensitivity	Complementary sensitivity function at specified point using slLinearizer or slTuner interface
writeBlockValue	Update block values in Simulink model
writeLookupTableData	Update portion of tuned lookup table
removePoint	Remove point from list of analysis points in slLinearizer or slTuner interface
removeAllPoints	Remove all points from list of analysis points in slLinearizer or slTuner interface
removeAllOpenings	Remove all openings from list of permanent openings in slLinearizer or slTuner interface
refresh	Resynchronize slLinearizer or slTuner interface with current model state

More About

Analysis Points

Analysis points, used by the `slLinearizer` and `slTuner` interfaces, identify locations within a model that are relevant for linear analysis and control system tuning. You use analysis points as inputs to the linearization commands, such as `getIOTransfer`, `getLoopTransfer`, `getSensitivity`, and `getCompSensitivity`. As inputs to the linearization commands, analysis points can specify any open-loop or closed-loop transfer function in a model. You can also use analysis points to specify design requirements when tuning control systems using commands such as `systune`.

Location refers to a specific block output port within a model or to a bus element in such an output port. For convenience, you can use the name of the signal that originates from this port to refer to an analysis point.

You can add analysis points to an `slLinearizer` or `slTuner` interface, `s`, when you create the interface. For example:

```
s = slLinearizer('scdcascade',{'u1','y1'});
```

Alternatively, you can use the `addPoint` command.

To view all the analysis points of `s`, type `s` at the command prompt to display the interface contents. For each analysis point of `s`, the display includes the block name and port number and the name of the signal that originates at this point. You can also programmatically obtain a list of all the analysis points using `getPoints`.

For more information about how you can use analysis points, see “Mark Signals of Interest for Control System Analysis and Design” on page 2-38 and “Mark Signals of Interest for Batch Linearization” on page 3-9.

Permanent Openings

Permanent openings, used by the `slLinearizer` and `slTuner` interfaces, identify locations within a model where the software breaks the signal flow. The software enforces these openings for linearization and tuning. Use permanent openings to isolate a specific model component. Suppose that you have a large-scale model capturing aircraft dynamics and you want to perform linear analysis on the airframe only. You can use permanent openings to exclude all other components of the model. Another example is when you have cascaded loops within your model and you want to analyze a specific loop.

Location refers to a specific block output port within a model. For convenience, you can use the name of the signal that originates from this port to refer to an opening.

You can add permanent openings to an `slLinearizer` or `slTuner` interface, `s`, when you create the interface or by using the `addOpening` command. To remove a location from the list of permanent openings, use the `removeOpening` command.

To view all the openings of `s`, type `s` at the command prompt to display the interface contents. For each permanent opening of `s`, the display includes the block name and port number and the name of the signal that originates at this location. You can also programmatically obtain a list of all the permanent loop openings using `getOpenings`.

Custom Linearization Function

You can specify a substitute linearization for a block or subsystem in your Simulink model using a custom function on the MATLAB path.

Your custom linearization function must have one `BlockData` input argument, which is a structure that the software creates and passes to the function. `BlockData` has the following fields:

Field	Description
`BlockName`	Name of the block for which you are specifying a custom linearization.
`Parameters`	Block parameter values, specified as a structure array with `Name` and `Value` fields. `Parameters` contains the names and values of the parameters you specify in the `blocksub.Value.ParameterNames` and `blocksub.Value.ParameterValues` fields.

Field	Description
`Inputs`	Input signals to the block for which you are defining a linearization, specified as a structure array with one structure for each block input. Each structure in `Inputs` has the following fields: **Field** / **Description** `BlockName` — Full block path of the block whose output connects to the corresponding block input. `PortIndex` — Output port of the block specified by `BlockName` that connects to the corresponding block input. `Values` — Value of the signal specified by `BlockName` and `PortIndex`. If this signal is a vector signal, then `Values` is a vector with the same dimension.
`ny`	Number of output channels of the block linearization.
`nu`	Number of input channels of the block linearization.
`BlockLinearization`	Current default linearization of the block, specified as a state-space model. You can specify a block linearization that depends on the default linearization using `BlockLinearization`.

Your custom function must return a model with `nu` inputs and `ny` outputs. This model must be one of the following:

- Linear model in the form of a D-matrix
- Control System Toolbox LTI model object
- Uncertain state-space model or uncertain real object (requires Robust Control Toolbox software)

For example, the following function multiplies the current default block linearization, by a delay of `Td = 0.5` seconds. The delay is represented by a Thiran filter with sample time `Ts = 0.1`. The delay and sample time are parameters stored in `BlockData`.

```
function sys = myCustomFunction(BlockData)
    Td = BlockData.Parameters(1).Value;
    Ts = BlockData.Parameters(2).Value;
    sys = BlockData.BlockLinearization*Thiran(Td,Ts);
end
```

Save this function to a location on the MATLAB path.

To use this function as a custom linearization for a block or subsystem, specify the `blocksub.Value.Specification` and `blocksub.Value.Type` fields.

```
blocksub.Value.Specification = 'myCustomFunction';
blocksub.Value.Type = 'Function';
```

To set the delay and sample time parameter values, specify the `blocksub.Value.ParameterNames` and `blocksub.Value.ParameterValues` fields.

```
blocksub.Value.ParameterNames = {'Td','Ts'};
blocksub.Value.ParameterValues = [0.5 0.1];
```

Algorithms

`slTuner` linearizes your Simulink model using the algorithms described in “Exact Linearization Algorithm” on page 2-175.

See Also

`systune` | `looptune` | `addPoint` | `addOpening` | `getIOTransfer` | `getLoopTransfer` | `getSensitivity` | `getCompSensitivity` | `linearize`

Topics

“Mark Signals of Interest for Control System Analysis and Design” on page 2-38
“How the Software Treats Loop Openings” on page 2-31
“Create and Configure slTuner Interface to Simulink Model”
“Vary Parameter Values and Obtain Multiple Transfer Functions” on page 3-21
“Tune Control Systems in Simulink”
“Fault-Tolerant Control of a Passenger Jet”
“Multi-Loop PI Control of a Robotic Arm”

Introduced in R2014a

slTunerOptions

Set `slTuner` interface options

Syntax

```
options = slTunerOptions
options = slTunerOptions(Name,Value)
```

Description

`options = slTunerOptions` returns the default `slTuner` interface option set.

`options = slTunerOptions(Name,Value)` returns an option set with additional options specified by one or more `Name,Value` pair arguments.

Examples

Create Option Set for `slTuner` Interface

Create an option set for an `slTuner` interface that sets the rate conversion method to the Tustin method with prewarping at a frequency of 10 rad/s.

```
options = slTunerOptions('RateConversionMethod','prewarp',...
                         'PreWarpFreq',10);
```

Alternatively, use dot notation to set the values of `options`.

```
options = slTunerOptions;
options.RateConversionMethod = 'prewarp';
options.PreWarpFreq = 10;
```

Input Arguments

Name-Value Pair Arguments

Specify optional comma-separated pairs of `Name,Value` arguments. `Name` is the argument name and `Value` is the corresponding value. `Name` must appear inside quotes. You can specify several name and value pair arguments in any order as `Name1,Value1,...,NameN,ValueN`.

Example: `'RateConversionMethod','prewarp'` sets the rate conversion method to the Tustin method with prewarping.

`SampleTime` — Linearization sample time
`0` (default) | positive scalar

Linearization sample time, specified as the comma-separated pair consisting of `'SampleTime'` and one of the following:

- `0` — Create a continuous-time model.
- Positive scalar — Specify the sample time for discrete-time systems.

`UseFullBlockNameLabels` — Flag indicating whether to truncate names of I/Os and states
`'off'` (default) | `'on'`

Flag indicating whether to truncate names of I/Os and states in the linearized model, specified as the comma-separated pair consisting of `'UseFullBlockNameLabels'` and either:

- `'off'` — Use truncated names for the I/Os and states in the linearized model.
- `'on'` — Use the full block path to name the I/Os and states in the linearized model.

`UseBusSignalLabels` — Flag indicating whether to use bus signal channel numbers or names
`'off'` (default) | `'on'`

Flag indicating whether to use bus signal channel numbers or names to label the I/Os in the linearized model, specified as the comma-separated pair consisting of `'UseBusSignalLabels'` and one of the following:

- `'off'` — Use bus signal channel numbers to label I/Os on bus signals in the linearized model.
- `'on'` — Use bus signal names to label I/Os on bus signals in the linearized model. Bus signal names appear in the results when the I/O points are located at the output of the following blocks:
  - Root-level inport block containing a bus object
  - Bus creator block
  - Subsystem block whose source traces back to the output of a bus creator block
  - Subsystem block whose source traces back to a root-level inport by passing through only virtual or nonvirtual subsystem boundaries

`StoreOffsets` — Flag indicating whether to compute linearization offsets
`false` (default) | `true`

Flag indicating whether to compute linearization offsets for inputs, outputs, states, and state derivatives or updated states, specified as the comma-separated pair consisting of `'StoreOffsets'` and one of the following:

- `false` — Do not compute linearization offsets.
- `true` — Compute linearization offsets.

You can configure an LPV System block using linearization offsets. For an example, see “Approximate Nonlinear Behavior Using Array of LTI Systems” on page 3-70

`StoreAdvisor` — Flag indicating whether to store diagnostic information
`false` (default) | `true`

Flag indicating whether to store diagnostic information during linearization, specified as the comma-separated pair consisting of `'StoreAdvisor'` and one of the following:

- `false` — Do not store linearization diagnostic information.
- `true` — Store linearization diagnostic information.

Linearization commands store and return diagnostic information in a `LinearizationAdvisor` object. For an example of troubleshooting linearization results using a `LinearizationAdvisor` object, see “Troubleshoot Linearization Results at Command Line” on page 4-28.

`RateConversionMethod` — Rate conversion method
`'zoh'` (default) | `'tustin'` | `'prewarp'` | `'upsampling_zoh'` | `'upsampling_tustin'` | `'upsampling_prewarp'`

Method used for rate conversion when linearizing a multirate system, specified as the comma-separated pair consisting of `'RateConversionMethod'` and one of the following:

- `'zoh'` — Zero-order hold rate conversion method
- `'tustin'` — Tustin (bilinear) method
- `'prewarp'` — Tustin method with frequency prewarp. When you use this method, set the `PreWarpFreq` option to the desired prewarp frequency.
- `'upsampling_zoh'` — Upsample discrete states when possible, and use `'zoh'` otherwise.
- `'upsampling_tustin'` — Upsample discrete states when possible, and use `'tustin'` otherwise.
- `'upsampling_prewarp'` — Upsample discrete states when possible, and use `'prewarp'` otherwise. When you use this method, set the `PreWarpFreq` option to the desired prewarp frequency.

For more information on rate conversion and linearization of multirate models, see:

- “Linearize Multirate Models” on page 2-139
- “Linearize Models Using Different Rate Conversion Methods” on page 2-145
- “Continuous-Discrete Conversion Methods”

Note If you use a rate conversion method other than `'zoh'`, the converted states no longer have the same physical meaning as the original states. As a result, the state names in the resulting LTI system change to `'?'`.

`PreWarpFreq` — Prewarp frequency
`0` (default) | positive scalar

Prewarp frequency in rad/s, specified as the comma-separated pair consisting of `'PreWarpFreq'` and a nonnegative scalar. This option applies only when `RateConversionMethod` is either `'prewarp'` or `'upsampling_prewarp'`.

`AreParamsTunable` — Flag indicating whether to recompile the model when varying parameter values
`true` (default) | `false`

Flag indicating whether to recompile the model when varying parameter values for linearization, specified as the comma-separated pair consisting of `'AreParamsTunable'` and one of the following:

- `true` — Do not recompile the model when all varying parameters are tunable. If any varying parameters are not tunable, recompile the model for each parameter grid point, and issue a warning message.

- `false` — Recompile the model for each parameter grid point. Use this option when you vary the values of nontunable parameters.

For more information about model compilation when you linearize with parameter variation, see “Batch Linearization Efficiency When You Vary Parameter Values” on page 3-7.

Output Arguments

`options` — `slTuner` interface options
`slTunerOptions` option set

`slTuner` interface options, returned as an `slTunerOptions` option set.

See Also

`slTuner`

Introduced in R2014a

update

Update operating point object with structural changes in model

Syntax

```
update(op)
```

Description

`update(op)` updates an operating point object, `op`, to reflect any changes in the associated Simulink model, such as states being added or removed.

Examples

Open the magball model:

```
magball
```

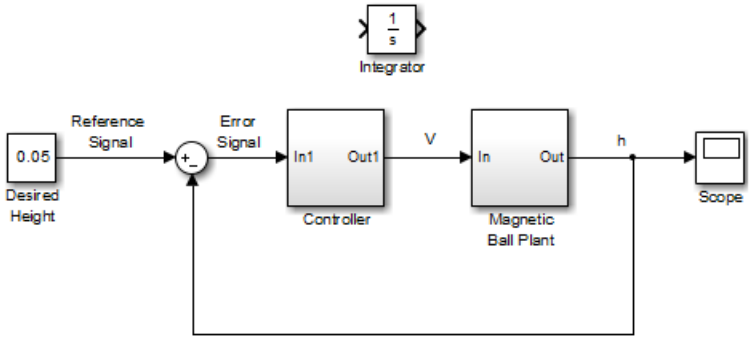
Create an operating point object for the model:

```
op=operpoint('magball')
```

This syntax returns:

```
 Operating Point for the Model magball.
 (Time-Varying Components Evaluated at time t=0)

States:
----------
(1.) magball/Controller/PID Controller/Filter
      x: 0
(2.) magball/Controller/PID Controller/Integrator
      x: 14
(3.) magball/Magnetic Ball Plant/Current
      x: 7
(4.) magball/Magnetic Ball Plant/dhdt
      x: 0
(5.) magball/Magnetic Ball Plant/height
      x: 0.05

Inputs: None
----------
```

Add an Integrator block to the model, as shown in the following figure.



Update the operating point to include this new state:

```
update(op)
```

The new operating point appears:

```
 Operating Point for the Model magball.
 (Time-Varying Components Evaluated at time t=0)

States:
----------
(1.) magball/Controller/PID Controller/Filter
      x: 0
(2.) magball/Controller/PID Controller/Integrator
      x: 14
(3.) magball/Magnetic Ball Plant/Current
      x: 7
(4.) magball/Magnetic Ball Plant/dhdt
      x: 0
(5.) magball/Magnetic Ball Plant/height
      x: 0.05
(6.) magball/Integrator
      x: 0

Inputs: None
----------
```

Alternatives

As an alternative to the `update` function, update operating point objects using the **Sync with Model** button in the **Model Linearizer**.

See Also

`operpoint` | `operspec`

Introduced before R2006a

writeLookupTableData

Update portion of tuned lookup table

Syntax

```
writeLookupTableData(st,blockid,breakpoints)
writeLookupTableData(st,blockid,ix1,…,ixN)
```

Description

When tuning lookup table blocks with `systune`, use this function to update only a portion of the table data in the Simulink model. This function is useful when retuning a single point or a portion of the lookup table. To update the entire lookup table, use `writeBlockValue`.

`writeLookupTableData(st,blockid,breakpoints)` writes tuned gain values from an `slTuner` interface to a portion of a lookup table in the associated Simulink model. Each row of `breakpoints` identifies an entry in the lookup table to update. `blockid` must identify a single block in the `TunedBlocks` property of the `slTuner` interface.

`writeLookupTableData(st,blockid,ix1,…,ixN)` updates a rectangular portion of the table data. The index vectors `ix1,…,ixN` select specific breakpoints along each table dimension.

Examples

Update Specific Entries in Lookup Table

Suppose you have an `slTuner` interface `st` to a Simulink model that contains a 2-D Lookup Table block `Kp Lookup`. The block is listed in `slTuner.TunedBlocks`. Suppose further that you have retuned for design points corresponding to the (3,5) and (4,6) breakpoints in the lookup table. Update the lookup table with the new values.

```
breakpoints = [3 5;4 6];
writeLookupTableData(st,'Kp Lookup',breakpoints)
```

Update Rectangular Portion of Lookup Table

Suppose you have retuned design points between the third and fifth values of the first scheduling variable, and the seventh and tenth values of the second scheduling variables. Update the lookup table with the new values.

```
ix1 = 3:5;
ix2 = 7:10;
writeLookupTableData(st,'Kp Lookup',ix1,ix2)
```

Input Arguments

`st` — Interface for tuning control systems modeled in Simulink

`slTuner` interface

Interface for tuning control systems modeled in Simulink, specified as an `slTuner` interface.

`blockid` — Lookup table

character vector | string

Lookup table to update with tuned values, specified as a character vector or string. The block identified by `blockid` must be a lookup-table block in the `TunedBlocks` property of the `slTuner` interface `st`. You can specify a full block path, or any portion of the block path that uniquely identifies the block among the other tuned blocks of `st`.

Example: `'scdcascade/Kp Lookup'`

Example: `"Kp Lookup"`

`breakpoints` — Lookup-table entries

integer array

Lookup-table entries to update, specified as an integer array. Each row of `breakpoints` specifies a table entry by its `(i1,i2,…,iN)` subscripts. For instance:

- To update the data associated with the first and third breakpoints in a 1-D Lookup Table block, use `breakpoints = [1;3]`.
- To update the data associated with the (3,5) and (4,6) entries in a 2-D Lookup Table block, use `breakpoints = [3 5;4 6]`.

`ix1,…,ixN` — Portion of lookup table

vectors

Portion of lookup table to update, specified as index vectors that select specific breakpoints along each table dimension. For instance, to update a 2-D Lookup Table block, specify two index vectors that identify the rows and columns to update. If you want to update the portion of the table blocked out by entries 3 through 5 in the first dimension and 7 through 10 in the second dimension, use `ix1 = 3:5` and `ix2 = 7:10`.

Tips

- If you use `writeBlockValue` to update other retuned blocks in your model, exclude the lookup table `blockid` from the list of blocks to update with that function.

See Also

`writeBlockValue`

Topics

“Validate Gain-Scheduled Control Systems” on page 11-36

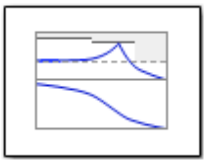
Introduced in R2017b

Blocks

Bode Plot, Check Bode Characteristics

Bode plot of linear system computed from nonlinear Simulink model

Library: Simulink Control Design / Linear Analysis Plots
Simulink Control Design / Model Verification



Description

The Bode Plot and Check Bode Characteristics blocks compute a linear system from a nonlinear Simulink model and plot the linear system on a Bode plot during simulation. These blocks are identical except for the default settings on the **Bounds** tab.

- The Bode Plot does not define default bounds.
- The Check Bode Characteristics block defines default bounds and enables these bounds for assertion.

For more information on frequency domain analysis of linear systems, see “Frequency-Domain Responses”.

During simulation, the software linearizes the portion of the model between specified linearization inputs and outputs and then plots the magnitude and phase of the linear system. You also can save the linear system as a variable in the MATLAB workspace.

The Simulink model can be continuous-time, 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 block displays plots for all input/output combinations.

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 in the MATLAB Command Window. 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 any bound is not satisfied, the signal is false (`0`).

To compute and plot the magnitude and phase of various portions of your model, you can add multiple Bode Plot and Check Bode Characteristics blocks.

These blocks do not support code generation and can be used only in `Normal` simulation mode.

Ports

Input

`Trigger` — External trigger signal

scalar

Use this input port (indicated by) to connect an external trigger signal for computing the model linearization. To specify the type of trigger signal to detect, use the **Trigger type** parameter.

Dependencies

To enable this port, set the **Linearize on** parameter to `External trigger`.

Output

z^{-1} — Assertion signal

`1` | `0`

Output the value of the assertion signal as a logical value. If any bound specified on the **Bounds** tab is violated, the assertion signal is false (`0`). Otherwise, this signal is true (`1`).

By default, the data type of the output signal is double. To set the output data type as Boolean, in the Simulink model, in the Configuration Parameters dialog box, select the **Implement logic signals as Boolean data** parameter. This setting applies to all blocks in the model that generate logic signals.

You can use the assertion signal to design complex assertion logic. For an example, see “Verify Model Using Simulink Control Design and Simulink Verification Blocks” on page 17-20.

Dependencies

To enable this port, select the **Output assertion signal** parameter.

Parameters

`Show Plot` — Open plot

button

To view Bode plots computed during a simulation, click this button before starting the simulation. If you specify bounds on the **Bounds** tab, they are also shown on the plot.

To show the plot when opening the block, select the **Show plot on block open** parameter.

For more information on using the plot, see “Using the Plot” on page 19-18.

`Show plot on block open` — Open plot when opening block

`off` (default) | `on`

Select this parameter to open the plot when opening the block. You can then perform tasks, such as adding or modifying bounds, in the plot window instead of using the block parameters. To access the block parameters from the plot window, select **Edit** or click .

For more information on using the plot, see “Using the Plot” on page 19-18.

Programmatic Use
Block Parameter: `LaunchViewOnOpen`
Type: character vector
Value: `'off'` | `'on'`
Default: `'off'`

`Response Optimization` — Open Response Optimizer

button

Open the **Response Optimizer** app to optimize the model response to meet the design requirements specified on the **Bounds** tab.

This button is available only if you have Simulink Design Optimization software installed.

For more information on response optimization, see “Design Optimization to Meet Step Response Requirements (GUI)” (Simulink Design Optimization) and “Design Optimization to Meet Time-Domain and Frequency-Domain Requirements (GUI)” (Simulink Design Optimization).

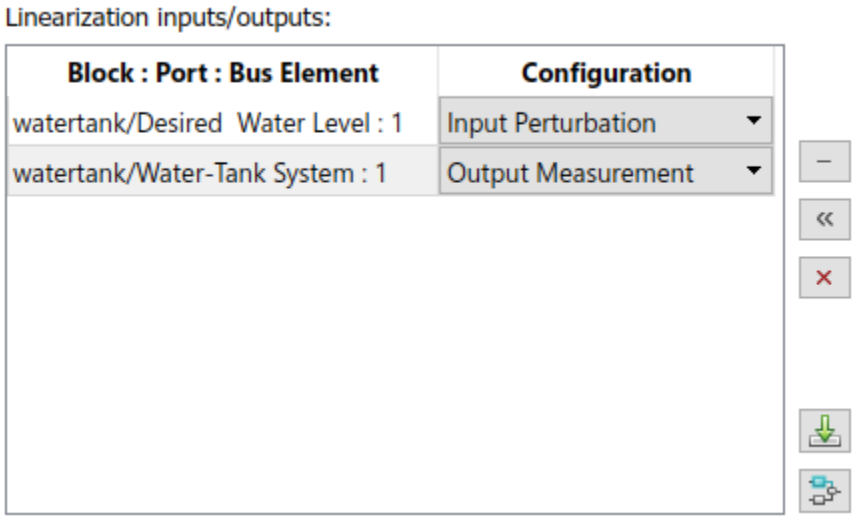
Linearizations

To specify the portion of the model to linearize and other linearization settings, use the parameters on the **Linearizations** tab. The default settings on this tab are the same for the Bode Plot and Check Bode Characteristics blocks.

`Linearization inputs/outputs` — Specify portion of model to linearize

linear analysis points

To specify the portion of the model to linearize, select signals from the Simulink model and add them as linearization inputs or outputs.



In the table, the **Block:Port:Bus Element** column shows the following information for each signal.

- Source block
- Output port of the source block to which the signal is connected
- Bus element name (if the signal is in a bus)

In the **Configuration** column, select the type of linear analysis point from the following types. For more information on linear analysis points, see “Specify Portion of Model to Linearize” on page 2-10.

- `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 a 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 a linearization input or output, the software generates a warning in the MATLAB Command Window and does not compute a linear system.

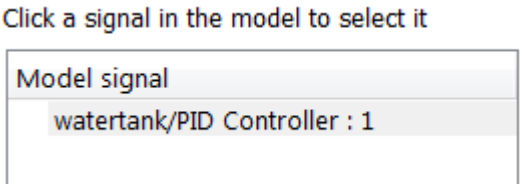
Edit Linearization Inputs and Outputs

To add linearization inputs and outputs:

1. To expand the signal selection area, click .

   The dialog box expands to display a **Click a signal in the model to select it** area.
2. In the Simulink model, select one or more signals.

   The selected signals appear in the **Model signal** table.

3. (Optional) For bus signals, expand the bus to select individual elements.

   Tip For large buses or other large lists of signals, you can filter the signal names. In the **Filter by name** box, enter search text. The name match is case-sensitive.

   To modify the filtering options, click . For more information on filtering options, see the **Enable regular expression** and **Show filtered results as a flat list** parameters.
4. To add the selected signal to the **Linearization inputs/outputs** table, click .
5. In the **Configuration** column, specify the signal type.

Alternatively, if you have linearization inputs and outputs defined in your model, you can add them to the **Linearization inputs/outputs** table by clicking .

To remove a signal from the **Linearization inputs/outputs** table, select the signal and click .

To highlight the source block of a signal in the Simulink model, select the signal in the **Linearization inputs/outputs** table and click .

`Enable regular expression` — Enable signal searching using regular expressions
`on` (default) | `off`

Select this option to enable the use of MATLAB regular expressions for filtering signal names. For example, entering `t$` in the **Filter by name** text box displays all signals whose names end with a lowercase `t` (and their immediate parents). For more information, see “Regular Expressions”.

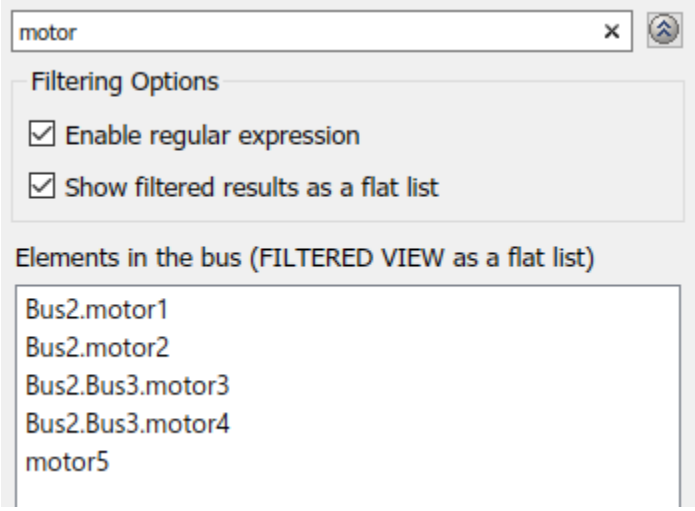
Dependencies

To enable this parameter, click next to the **Filter by name** text box.

`Show filtered results as a flat list` — Display filtered bus signal hierarchy using flat list
`off` (default) | `on`

Select this option to display the list of filtered signals in a flat list format. The flat list format uses dot notation to reflect the hierarchy of bus signals. The signals are filtered based on the text in the **Filter by name** text box.

The following figure shows an example of the flat list format for a filtered set of nested bus signals.



Dependencies

To enable this parameter, click next to the **Filter by name** text box.

`Linearize on` — When to compute linear model
`Simulation snapshots` (default) | `External trigger`

Use this parameter to specify when you want to compute a linear model.

To compute linear models at specified simulation snapshot times, set this parameter to `Simulation snapshots`. Specify snapshot times using the **Snapshot times** parameter.

Use simulation snapshots when you:

- Know one or more times when the model is at a steady-state operating point
- Want to compute linear systems at specific times

To compute linear models at trigger-based simulation events, set this parameter to `External trigger`. Selecting this option adds a trigger input port to the block, to which you connect your external trigger signal. To specify the type of trigger to detect, use the **Trigger type** parameter.

Use an external trigger when a signal generated during simulation indicates that the model is at a steady-state condition of interest. For example, for an aircraft model, you might want to compute the linear system whenever the fuel mass is a given fraction of the maximum fuel mass.

Programmatic Use

Block Parameter: `LinearizeAt`
Type: character vector
Value: `'SnapshotTimes'` | `'ExternalTrigger'`
Default: `'SnapshotTimes'`

`Snapshot times` — Simulation times at which to compute linear model

`0` (default) | positive real value | vector of positive real values

To compute a linear system at specific simulation times, such as a time that you know the model reaches a steady state operating point, specify one or more snapshot times. To specify multiple snapshot times, specify this parameter as a vector of positive values.

Snapshot times must be less than or equal to the simulation time specified in the Simulink model.

For examples of linearizing a model at simulation snapshot times, see:

- “Visualize Linear System at Multiple Simulation Snapshots” on page 2-85
- “Verify Model at Default Simulation Snapshot Time” on page 17-5
- “Verify Model at Multiple Simulation Snapshots” on page 17-13

Dependencies

To enable this parameter, set the **Linearize on** parameter to `Simulation snapshots`

Programmatic Use

Block Parameter: `SnapshotTimes`
Type: character vector
Value: `'0'` | positive real value | vector of positive real values
Default: `'0'`

`Trigger type` — Type of external trigger to detect

`Rising edge` (default) | `Falling edge`

Specify the trigger to detect in the external trigger signal as one of the following types.

- `Rising edge` — Use the rising edge of the trigger signal; that is, when the signal changes from `0` to `1`.
- `Falling edge` — Use the falling edge of the trigger signal; that is, when the signal changes from `1` to `0`.

Dependencies

To enable this parameter, set the **Linearize on** parameter to `External trigger`.

Programmatic Use
Block Parameter: `TriggerType`
Type: character vector
Value: `'rising'` | `'falling'`
Default: `'rising'`

`Enable zero-crossing detection` — Enable zero-crossing detection

`on` (default) | `off`

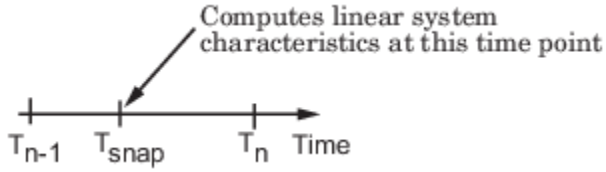
Select this option to enable zero-crossing detection.

When you set the **Linearize on** parameter to `Simulation snapshots`, enabling zero-crossing detection ensures that the software computes the linear model at the exact snapshot times you specify in the **Snapshot times** parameter.

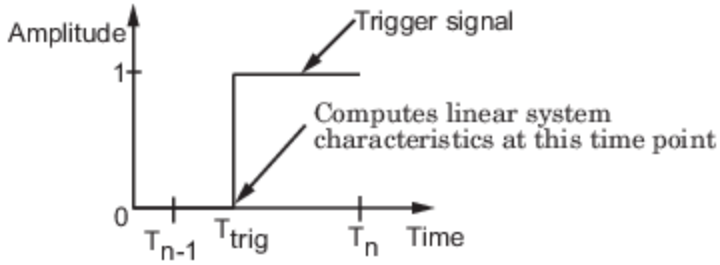
When you set the **Linearize on** parameter to `External trigger`, enabling zero-crossing detection ensures that the software computes the linear model at the exact time that the external trigger is detected. To specify the type of trigger, use the **Trigger type** parameter.

If you clear this option, the software computes the linear system at simulation times selected by the variable-step Simulink solver, which might not correspond to an exact snapshot time or the exact time when a trigger signal is detected.

For example, consider the case where the variable-step solver selects simulation times T_{n-1} and T_n. As shown in the following figure, the specified snapshot time T_{snap} can be between the selected simulation times. If you enable zero-crossing detection, the solver also simulates the model at time T_{snap} and computes the linear model at this point.



Similarly, the external trigger can be detected at a time T_{trig} that is between the selected simulation times. If you enable zero-crossing detection, the solver also simulates the model at time T_{trig} and computes the linear model at this point.



In both cases, if you do not enable zero-crossing detection, the software computes the linear model at either T_{n-1} or T_n.

For more information on zero-crossing detection, see “Zero-Crossing Detection”.

Dependencies

This parameter is ignored when you use a fixed-step Simulink solver.

Programmatic Use

Block Parameter: `ZeroCross`
Type: character vector
Value: `'on'` | `'off'`
Default: `'on'`

`Use exact delays` — Use exact delays in linear model

`off` (default) | `on`

Select this option to compute a linear model with exact delays. If you clear this option, the linear model uses Padé approximations of any delays.

For more information on linearizing models with delays, see “Linearize Models with Delays” on page 2-78.

Programmatic Use

Block Parameter: `UseExactDelayModel`
Type: character vector
Value: `'off'` | `'on'`
Default: `'off'`

`Linear system sample time` — Sample time of linear system

`'auto'` (default) | positive finite value | `0`

To compute a linear system with the specified sample time, the software coverts sample times in the model using the method you specify in the **Sample time rate conversion method** parameter.

You can set the sample time to one of the following values.

- `auto` — If all blocks in the model are continuous-time, use a sample time of `0`. Otherwise, set the sample time to the least common multiple of the nonzero sample times in the model.
- Positive finite value — Create a discrete-time model with the specified sample time
- `0` — Create a continuous-time model

Programmatic Use

Block Parameter: `SampleTime`
Type: character vector
Value: `'auto'` | positive finite value | `'0'`
Default: `'auto'`

`Sample time rate conversion method` — Rate conversion method

`Zero-Order Hold` (default) | `Tustin (bilinear)` | `Tustin with Prewarping` | ...

Method for converting sample times during linearization, specified as one of the following values.

- `Zero-Order Hold` — Zero-order hold, where the control inputs are assumed piecewise constant over the sample time `Ts`. This method usually performs better in the 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. This method usually performs better in the frequency domain.

- `Tustin with Prewarping` — Bilinear (Tustin) approximation with frequency prewarping. Specify the prewarp frequency using the **Prewarp frequency** parameter. This method usually performs better in the frequency domain. Use this method to ensure matching in a frequency region of interest.
- `Upsampling when possible, Zero-Order Hold otherwise` — Upsample a discrete-time system when possible; otherwise, use a zero-order hold.
- `Upsampling when possible, Tustin otherwise` — Upsample a discrete-time system when possible; otherwise, use a Tustin approximation.
- `Upsampling when possible, Tustin with Prewarping otherwise` — Upsample a discrete-time system when possible; otherwise, use a Tustin approximation with frequency prewarping.

You can upsample only when you convert a discrete-time system to a new faster sample time that is an integer multiple of the sample time of the original system.

For more information on rate conversion and linearization of multirate models, see:

- “Multirate Linearization Algorithm” on page 2-140
- “Linearize Models Using Different Rate Conversion Methods” on page 2-145
- “Continuous-Discrete Conversion Methods”

Note If you use a rate conversion method other than `Zero-Order Hold`, the converted states no longer have the same physical meaning as the original states. As a result, the state names in the resulting LTI system change to `'?'`.

Dependencies

To enable this parameter, set the **Linear system sample time** parameter to a value other than `auto`.

Programmatic Use

Block Parameter: `RateConversionMethod`
Type: character vector
Value: `'zoh'` | `'tustin'` | `'prewarp'` | `'upsampling_zoh'` | `'upsampling_tustin'` | `'upsampling_prewarp'`
Default: `'zoh'`

`Prewarp frequency` — Prewarp frequency for Tustin rate conversion

`'10'` (default) | positive scalar

Prewarp frequency for Tustin rate conversion in radians per second, specified as a scalar value less than the Nyquist frequency before and after resampling.

Dependencies

To enable this parameter, set the **Sample time rate conversion method** parameter to one of the following values.

- `Tustin with Prewarping`
- `Upsampling when possible, Tustin with Prewarping otherwise`

Programmatic Use
Block Parameter: `PreWarpFreq`
Type: character vector
Value: positive scalar
Default: `'10'`

`Use full block names` — Use full block path in state, input, and output names

`off` (default) | `on`

To show the state, input, and output names of the computed linear system using their full block path, select this parameter. For example, in the `scdcstr` model, a state in the `Integrator1` block of the `CSTR` subsystem appears with its full block path as `scdcstr/CSTR/Integrator1`.

If you clear this parameter, only the names of the states, inputs, and outputs are used, which is useful when the signal names are unique and you know their locations in your Simulink model. In the preceding example, the state name of the integrator block appears as `Integrator1`.

The computed linear system is a state-space object (`ss`). The state, input, and output names for the system appear in the following state-space object properties.

Input, Output, or State Name	State-Space Object Property
Linearization input names	`InputName`
Linearization output names	`OutputName`
State names	`StateName`

Programmatic Use
Block Parameter: `UseFullBlockNameLabels`
Type: character vector
Value: `'off'` | `'on'`
Default: `'off'`

`Use bus signal names` — Use bus signal names in linear system

`off` (default) | `on`

When you select an entire bus as a linearization input or output, select this parameter to use the signal names of the individual bus elements in the computed linear system. If you do not enable this option, the bus channel numbers are used instead.

Note Selecting an entire bus signal is not recommended. Instead, select individual bus elements.

Bus signal names appear when the linearization input or output is from one 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

Dependencies

Using this parameter is not supported when your model contains mux/bus mixtures.

Programmatic Use

Block Parameter: `UseBusSignalLabels`
Type: character vector
Value: `'off'` | `'on'`
Default: `'off'`

Bounds

To define magnitude bounds for your Bode plot and specify whether to check for violations of these bounds, use the parameters on the **Bounds** tab. The default settings on this tab are different for the Bode Plot and Check Bode Characteristics blocks.

`Include upper magnitude bound in assertion` — Check whether Bode magnitude violates upper bounds

`on` | `off`

Select this parameter to check whether the magnitude of the Bode plot violates the lower bounds specified in the corresponding **Magnitudes** and **Frequencies** parameters.

By default, this parameter is cleared for the Bode Plot block and selected for the Check Bode Characteristics block.

Dependencies

This parameter is used for assertion only if you select the **Enable assertion** parameter.

Programmatic Use

Block Parameter: `EnableUpperBound`
Type: character vector
Value: `'on'` | `'off'`
Default: `'off'` for Bode Plot block, `'on'` for Check Bode Characteristics block

`Frequencies` — Frequencies for upper magnitude bound segments

vector | matrix | cell array

To define upper bounds for your Bode plot, specify the start and end frequencies for each bound segment in radians per second. To specify no upper bounds, set this parameter to `[]`.

By default, the frequencies are `[]` for the Bode Plot block and `[10 100]` for the Check Bode Characteristics block.

To specify:

- A single bound with one edge, specify a two-element vector of positive finite values.
- A single bound with multiple edges, specify an *N*-by-2 array, where *N* is the number of edges. For example, enter `[0.1 1;1 10]` for two edges at frequencies `[0.1 1]` and `[1 10]`.
- Multiple bounds, specify an *M*-element cell array of matrices, where *M* is the number of bounds.

Set the corresponding magnitude values for the bounds using the **Magnitudes** parameter. The dimensions of the **Frequencies** and **Magnitudes** parameters must match.

You can also add bound segments in the plot window. For more information, see “Using the Plot” on page 19-18.

Dependencies

To check whether the magnitude bounds are violated during simulation, select both the **Include upper magnitude bound in assertion** and **Enable assertion** parameters.

Programmatic Use

Block Parameter: `UpperBoundFrequencies`
Type: character vector
Value: vector of positive finite numbers | matrix of positive finite numbers | cell array of matrices with positive finite numbers
Default: `'[]'` for Bode Plot block, `'[10 100]'` for Check Bode Characteristics block

`Magnitudes` — Magnitudes for upper bound segments

vector | matrix | cell array

To define upper bounds for your Bode plot, specify the start and end frequencies for each bound segment in decibels. To specify no upper bounds, set this parameter to `[]`.

By default, the magnitudes are `[]` for the Bode Plot block and `[-20 -20]` for the Check Bode Characteristics block.

To specify:

- A single bound with one edge, specify a two-element vector of finite magnitude values.
- A single bound with multiple edges, specify an *N*-by-2 array, where *N* is the number of edges. For example, enter `[-10 -10;-20 -20]` for two edges with magnitudes `[-10 -10]` and `[-20 -20]`.
- Multiple bounds, specify an *M*-element cell array of matrices, where *M* is the number of bounds.

Set the corresponding frequency values for the bounds using the **Frequencies** parameter. The dimensions of the **Magnitude** and **Frequencies** parameters must match.

You can also add bound segments in the plot window. For more information, see “Using the Plot” on page 19-18.

Dependencies

To check whether the magnitude bounds are violated during simulation, select both the **Include upper magnitude bound in assertion** and **Enable assertion** parameters.

Programmatic Use

Block Parameter: `UpperBoundFrequencies`
Type: character vector
Value: vector of finite numbers | matrix of finite numbers | cell array of matrices with finite numbers
Default: `'[]'` for Bode Plot block, `'[-20 -20]'` for Check Bode Characteristics block

`Include lower magnitude bound in assertion` — Check whether Bode magnitude violates lower bounds

on | off

Select this parameter to check whether the magnitude of the Bode plot violates the lower bounds specified in the corresponding **Magnitudes** and **Frequencies** parameters.

By default, this parameter is cleared for the Bode Plot block and selected for the Check Bode Characteristics block.

Dependencies

This parameter is used for assertion only if you select the **Enable assertion** parameter.

Programmatic Use

Block Parameter: `EnableLowerBound`
Type: character vector
Value: `'on'` | `'off'`
Default: `'off'` for Bode Plot block, `'on'` for Check Bode Characteristics block

`Frequencies` — Frequencies for lower magnitude bound segments

vector | matrix | cell array

To define lower bounds for your Bode plot, specify the start and end frequencies for each bound segment in radians per second. To specify no lower bounds, set this parameter to `[]`.

By default, the frequencies are `[]` for the Bode Plot block and `[0.1 1]` for the Check Bode Characteristics block.

To specify:

- A single bound with one edge, specify a two-element vector of positive finite values.
- A single bound with multiple edges, specify an *N*-by-2 array, where *N* is the number of edges. For example, enter `[0.1 1;1 10]` for two edges at frequencies `[0.1 1]` and `[1 10]`.
- Multiple bounds, specify an *M*-element cell array of matrices, where *M* is the number of bounds.

Set the corresponding magnitude values for the bounds using the **Magnitudes** parameter. The dimensions of the **Frequencies** and **Magnitudes** parameters must match.

You can also add bound segments in the plot window. For more information, see “Using the Plot” on page 19-18.

Dependencies

To check whether the magnitude bounds are violated during simulation, select both the **Include lower magnitude bound in assertion** and **Enable assertion** parameters.

Programmatic Use

Block Parameter: `LowerBoundFrequencies`
Type: character vector
Value: vector of positive finite numbers | matrix of positive finite numbers | cell array of matrices with positive finite numbers
Default: `'[]'` for Bode Plot block, `'[0.1 1]'` for Check Bode Characteristics block

`Magnitudes` — Magnitudes for lower bound segments

vector | matrix | cell array

To define lower bounds for your Bode plot, specify the magnitudes at each corresponding frequency point in decibels. To specify no lower bounds, set this parameter to `[]`.

By default, the magnitudes are `[]` for the Bode Plot block and `[20 20]` for the Check Bode Characteristics block.

To specify:

- A single bound with one edge, specify a two-element vector of finite magnitude values.
- A single bound with multiple edges, specify an *N*-by-2 array, where *N* is the number of edges. For example, enter `[20 20; 40 40]` for two edges with magnitudes `[20 20]` and `[40 40]`.
- Multiple bounds, specify an *M*-element cell array of matrices, where *M* is the number of bounds.

Set the corresponding frequency values for the bounds using the **Frequencies** parameter. The dimensions of the **Magnitude** and **Frequencies** parameters must match.

You can also add bound segments in the plot window. For more information, see “Using the Plot” on page 19-18.

Dependencies

To check whether the magnitude bounds are violated during simulation, select both the **Include lower magnitude bound in assertion** and **Enable assertion** parameters.

Programmatic Use

Block Parameter: `LowerBoundFrequencies`
Type: character vector
Value: vector of finite numbers | matrix of finite numbers | cell array of matrices with finite numbers
Default: `'[]'` for Bode Plot block, `'[20 20]'` for Check Bode Characteristics block

Logging

To control whether linearization results computed during the simulation are saved, use the parameters on the **Logging** tab. The default settings on this tab are the same for the Bode Plot and Check Bode Characteristics blocks.

`Save data to workspace` — Save linear systems for further analysis

`off` (default) | `on`

Select this parameter to save the computed linear systems for further analysis or control design. The data is saved in a structure with the following fields.

- `time` — Simulation times at which the linear systems are computed.
- `values` — State-space model representing the linear system. If the linear system is computed at multiple simulation times, `values` is an array of state-space models.
- `operatingPoints` — Operating points corresponding to each linear system in `values`. To enable this field, select the **Save operating points for each linearization** parameter.

To specify the name of the saved data structure, use the **Variable name** property.

The location of the saved data structure depends upon the configuration of the Simulink model.

- If the model is not configured to save simulation output as a single object, the data structure is a variable in the MATLAB workspace.
- If the 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 Configuration Parameters dialog box, select the **Single simulation output** parameter.

For more information about data logging in Simulink, see “Export Simulation Data” and the `Simulink.SimulationOutput` reference page.

Programmatic Use

Block Parameter: `SaveToWorkspace`
Type: character vector
Value: `'off'` | `'on'`
Default: `'off'`

`Variable name` — Name of data structure for saving linear systems

`sys` (default) | character vector

Specify the name of the data structure that stores linear systems computed during simulation.

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.

Dependencies

To enable this parameter, select the **Save data to workspace** parameter.

Programmatic Use

Block Parameter: `SaveName`
Type: character vector
Default: `'sys'`

`Save operating points for each linearization` — Save operating points with linearization

`off` (default) | `on`

Select this parameter to save the operating point at which each linearization is computed. Selecting this parameter adds the `operatingPoints` field to the saved data structure.

Dependencies

To enable this parameter, select the **Save data to workspace** parameter.

Programmatic Use

Block Parameter: `SaveOperatingPoints`
Type: character vector
Value: `'off'` | `'on'`
Default: `'off'`

Assertion

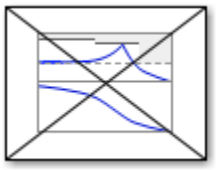
To control the assertion behavior of the block when bounds defined on the **Bounds** tab are violated, use the parameters on the **Assertion** tab. The default settings on this tab are the same for the Bode Plot and Check Bode Characteristics blocks.

`Enable assertion` — Enable bound checking

`on` (default) | `off`

To check whether the bounds defined on the **Bounds** tab are satisfied during the simulation, select this parameter. When a bound is not satisfied, the assertion fails and a warning is generated.

Clearing this parameter disables assertion; that is, the block no longer checks that the specified bounds are satisfied. The block icon also updates to indicate that assertion is disabled.



By default, on the **Bounds** tab:

- The Bode Plot block does not have defined bounds.
- The Check Bode Characteristics block has defined bounds.

You can configure your Simulink model to enable or disable all model verification blocks and override the **Enable assertion** parameter. To do so, in the Simulink model, in the Configuration Parameters dialog box, specify the **Model Verification block enabling** parameter.

Programmatic Use

Block Parameter: `enabled`
Type: character vector
Value: `'on'` | `'off'`
Default: `'on'`

`Simulation callback when assertion fails` — Expression to evaluate when bounds are violated

`''` (default) | MATLAB expression

Specify a MATLAB expression to evaluate when the bounds specified on the **Bounds** tab are violated. All variables used in the expression must be in the MATLAB workspace.

Dependencies

To enable this parameter, select the **Enable assertion** parameter.

Programmatic Use

Block Parameter: `callback`
Type: character vector
Value: MATLAB expression
Default: `''`

`Stop simulation when assertion fails` — Stop simulation when bounds are violated

`off` (default) | `on`

To stop the simulation when the bounds specified on the **Bounds** tab are violated, select this parameter. If you do not select this option, the bound violation is reported as a warning in the MATLAB Command Window and the simulation continues.

If you run the simulation from the Simulink model, when the assertion fails, the block where the bound violation occurs is highlighted and an error message is displayed in the Simulation Diagnostics window.

Note Since selecting this option stops the simulation as soon as the assertion fails, bound violations that might occur later during the simulation are not reported. If you want all bound violations to be reported, do not select this option.

Dependencies

To enable this parameter, select the **Enable assertion** parameter.

Programmatic Use

Block Parameter: `stopWhenAssertionFail`
Type: character vector
Value: `'off'` | `'on'`
Default: `'off'`

`Output assertion signal` — Add assertion output port

`off` (default) | `on`

Add the $\mathbf{z}^{-1}$ assertion signal output port to the block. This port outputs the value of the assertion as a Boolean signal. When the bounds defined on the **Bounds** tab are violated, the assertion fails and the assertion signal is `0`. Otherwise, the assertion signal is `1`.

You can use the assertion signal to design complex assertion logic. For an example, see “Verify Model Using Simulink Control Design and Simulink Verification Blocks” on page 17-20.

Programmatic Use

Block Parameter: `export`
Type: character vector
Value: `'off'` | `'on'`
Default: `'off'`

More About

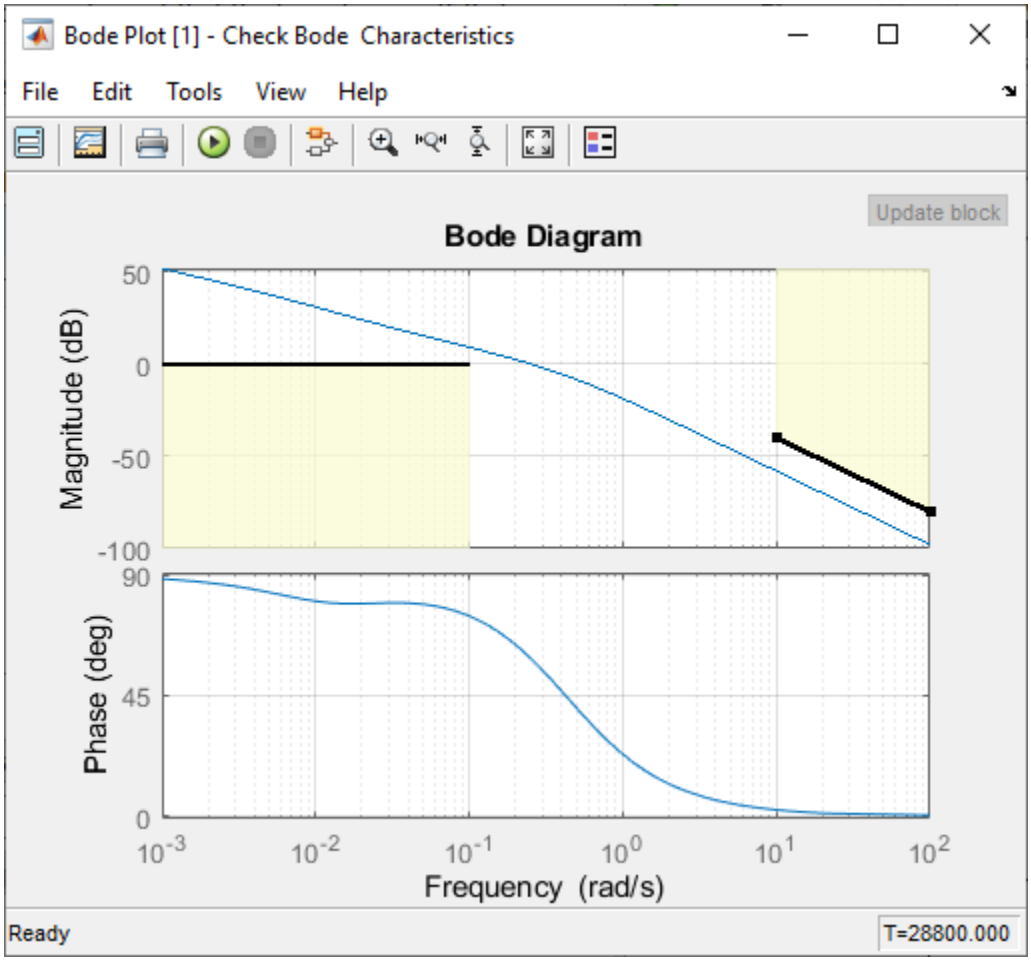
Using the Plot

In the plot window, you can:

- View the block parameters by clicking or selecting **Edit**.
- Highlight the block in the model by clicking or selecting **Highlight Simulink Block** in the **View** menu.
- Simulate the model by clicking .
- Add a legend to the plot by clicking .

To display response characteristics, such as the peak response or stability margins, right-click the plot. Then, under **Characteristics**, select the characteristics to show.

Any bounds you specify for the block appear on the plot.

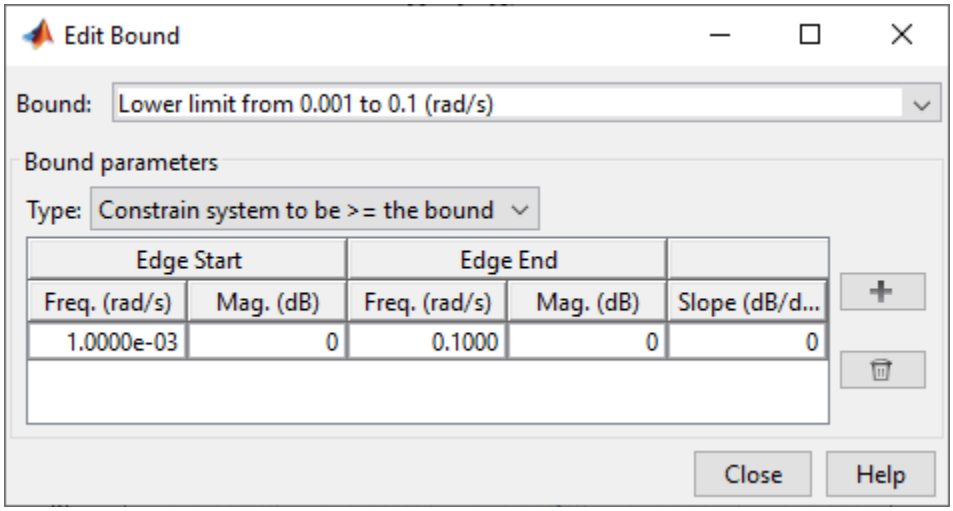


You can specify bounds on the **Bounds** tab.

Alternatively, to add a new bound from the plot, right-click the plot and select **Bounds > New Bound**.

To modify a bound, you can drag the bound in the plot. You can also:

- Right-click the plot and select **Bounds > Edit Bound**.
- Right-click the bound and select **Edit**.



In the Edit Bound dialog box, in the **Bound** drop-down, select the bound to edit. Then, specify in the bound parameters and click **Close**.

After adding or editing bounds from the plot window, update the bound value in the block by clicking **Update Block**.

See Also

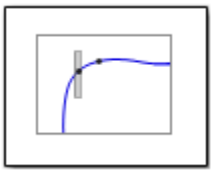
Topics

“Visualize Bode Response of Simulink Model During Simulation” on page 2-61
“Verify Model Using Simulink Control Design and Simulink Verification Blocks” on page 17-20
“Visualize Linear System of a Continuous-Time Model Discretized During Simulation” on page 2-91

Introduced in R2010b

Check Nichols Characteristics

Check that gain and phase bounds on Nichols response are satisfied during simulation



Library

Simulink Control Design

Description

This block is the 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 in the MATLAB Command Window. 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 any 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 19-123 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** • **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

- “Verify Model at Default Simulation Snapshot Time” on page 17-5
- “Verify Model at Multiple Simulation Snapshots” on page 17-13
- “Verify Model Using Simulink Control Design and Simulink Verification Blocks” on page 17-20
- “Verify Frequency-Domain Characteristics of an Aircraft” on page 17-27

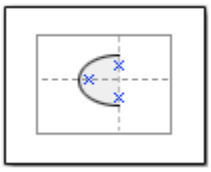
How To

“Monitor Linear System Characteristics in Simulink Models” on page 17-2

Introduced in R2010b

Check Pole-Zero Characteristics

Check that bounds on pole locations are satisfied during simulation



Library

Simulink Control Design

Description

This block is the 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 in the MATLAB Command Window. 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 any 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 19-167 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**
	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

- “Verify Model at Default Simulation Snapshot Time” on page 17-5
- “Verify Model at Multiple Simulation Snapshots” on page 17-13
- “Verify Model Using Simulink Control Design and Simulink Verification Blocks” on page 17-20
- “Verify Frequency-Domain Characteristics of an Aircraft” on page 17-27

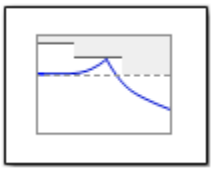
How To

“Monitor Linear System Characteristics in Simulink Models” on page 17-2

Introduced in R2010b

Check Singular Value Characteristics

Check that singular value bounds are satisfied during simulation



Library

Simulink Control Design

Description

This block is the 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 multirate 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 in the MATLAB Command Window. 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 any 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 19-197 in the Singular Value Plot block reference page.

Task		Parameters
Configure linearization.	Specify inputs and outputs (I/Os).	In **Linearizations** tab: • **Linearization inputs/outputs** • **Click a model signal to add it as a linearization I/O**
	Specify settings.	In **Linearizations** tab: • **Linearize on** • **Snapshot times** • **Trigger type**
	Specify algorithm options.	In **Linearizations** tab: • **Enable zero-crossing detection** • **Use exact delays** • **Linear system sample time** • **Sample time rate conversion method** • **Prewarp frequency (rad/s)**
	Specify labels for linear system I/Os and state names.	In **Linearizations** tab: • **Use full block names** • **Use bus signal names**
Specify bounds on the linear system for assertion.		In **Bounds** tab: • **Include upper singular value bound in assertion** • **Include lower singular value bound in assertion**

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

Singular Value Plot

Tutorials

- “Verify Model at Default Simulation Snapshot Time” on page 17-5
- “Verify Model at Multiple Simulation Snapshots” on page 17-13
- “Verify Model Using Simulink Control Design and Simulink Verification Blocks” on page 17-20
- “Verify Frequency-Domain Characteristics of an Aircraft” on page 17-27

How To

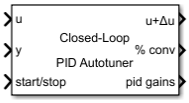
“Monitor Linear System Characteristics in Simulink Models” on page 17-2

Introduced in R2010b

Closed-Loop PID Autotuner

Automatically tune PID gains based on plant frequency responses estimated from closed-loop experiment in real time

Library: Simulink Control Design



Description

The Closed-Loop PID Autotuner block lets you tune a PID controller in real time against a physical plant for which you have an initial PID controller that yields a stable loop. The plant remains under closed-loop control of the initial PID controller during the entire autotuning process. The block can tune the PID controller to achieve a specified bandwidth and phase margin without a parametric plant model. If you have a code-generation product such as Simulink Coder, you can generate code that implements the tuning algorithm on hardware, letting you tune in real time with or without using Simulink to manage the autotuning process.

If you have a plant modeled in Simulink and an initial PID controller, you can perform closed-loop PID autotuning against the modeled plant. Doing so lets you preview plant response and adjust the settings for PID autotuning before tuning the controller in real time.

To achieve model-free tuning, the Closed-Loop PID Autotuner block:

1. Injects a test signal into the plant to collect plant input-output data and estimate frequency response in real time. The test signal is combination of sinusoidal perturbation signals added on top of the plant input.
2. At the end of the experiment, tunes PID controller parameters based on estimated plant frequency responses near the target bandwidth.
3. Updates a PID Controller block or a custom PID controller with the tuned parameters, allowing you to validate closed-loop performance in real time.

Unlike with the Open-Loop PID Autotuner block, the loop remains closed throughout the experiment. Keeping the loop closed helps to maintain safe operation of the plant during the estimation experiment.

You can use the Closed-Loop PID Autotuner block to tune PID controllers for:

- Any stable plant
- Any continuous-time plant with one or more integrators (poles at $s = 0$) or one or more pairs of complex poles on the imaginary axis
- Any discrete-time plant with one or more integrators (poles at $z = -1$) or pairs of complex poles on the unit circle $|z| = 1$

If you do not have an initial PID controller, you can use the Open-Loop PID Autotuner block to obtain one. You can then switch to closed-loop PID autotuning for refinement or retuning.

The block supports code generation with Simulink Coder, Embedded Coder®, and Simulink PLC Coder™. It does not support code generation with HDL Coder™.

For more information about using the Closed-Loop PID Autotuner block, see:

- “PID Autotuning for a Plant Modeled in Simulink” on page 8-7
- “PID Autotuning in Real Time” on page 8-13

For more general information about PID autotuning and a comparison of the closed-loop and open-loop approaches, see “When to Use PID Autotuning” on page 8-2.

Ports

Input

u — Signal from controller

scalar

Insert the block into your system such that this port accepts a control signal from a source. Typically, this port accepts the signal from the PID controller in your system.

Data Types: `single` | `double`

y — Plant output

scalar

Connect this port to the plant output.

Data Types: `single` | `double`

`start/stop` — Start and stop the autotuning experiment

scalar

To start and stop the autotuning process, provide a signal at the `start/stop` port. When the value of the signal changes from:

- Negative or zero to positive, the experiment starts
- Positive to negative or zero, the experiment stops

When the experiment is not running, the block passes signals unchanged from **u** to **u+Δu**. In this state, the block has no impact on plant or controller behavior.

Typically, you can use a signal that changes from 0 to 1 to start the experiment, and from 1 to 0 to stop it. Some points to consider when configuring the **start/stop** signal include:

- Start the experiment when the plant is at the desired equilibrium operating point. Use the initial controller to drive the plant to the operating point. If you have no initial controller (open-loop tuning only) you can use a source block connected to **u** to drive the plant to the operating point.
- Avoid any load disturbance to the plant during the experiment. Load disturbance can distort the plant output and reduce the accuracy of the frequency-response estimation.
- Let the experiment run long enough for the algorithm to collect sufficient data for a good estimate at all frequencies it probes. There are two ways to determine when to stop the experiment:
  - Determine the experiment duration in advance. A conservative estimate for the experiment duration is $200/\omega_c$ in superposition experiment mode or $550/\omega_c$ in sinestream experiment mode, where ω_c is your target bandwidth.

  - Observe the signal at the `% conv` output, and stop the experiment when the signal stabilizes near 100%.
- When you stop the experiment, the block computes tuned PID gains and updates the signal at the `pid gains` port.

You can configure any logic appropriate for your application to control the start and stop times of the experiment.

Data Types: `single` | `double`

`bandwidth` — Target bandwidth for tuning

scalar

Supply a value for the `Target bandwidth (rad/sec)` parameter. See that parameter for details.

Dependencies

To enable this port, in the Tuning tab, next to `Target bandwidth (rad/sec)`, select **Use external source**.

Data Types: `single` | `double`

`target PM` — Target phase margin for tuning

scalar

Supply a value for the `Target phase margin (degrees)` parameter. See that parameter for details.

Dependencies

To enable this port, in the Tuning tab, next to `Target phase margin (degrees)`, select **Use external source**.

Data Types: `single` | `double`

`sine Amp` — Amplitudes of injected sinusoidal signals

scalar | vector

Supply a value for the `Sine Amplitudes` parameter. See that parameter for details.

Dependencies

To enable this port, in the Experiment tab, next to `Sine Amplitudes`, select **Use external source**.

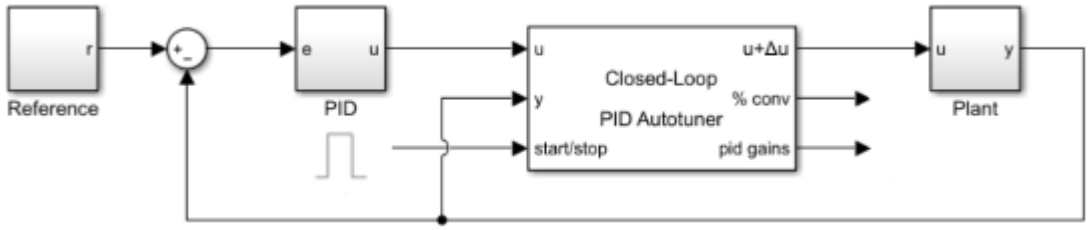
Data Types: `single` | `double`

Output

`u+Δu` — Signal for plant input

scalar

Insert the block into your system such that this port feeds the input signal to your plant.



- When the experiment is running (`start/stop` positive), the block injects test signals into the plant at this port. If you have any saturation or rate limit protecting the plant, feed the signal from **u+Δu** into it.
- When the experiment is not running (`start/stop` zero or negative), the block passes signals unchanged from **u** to **u+Δu**.

Dependencies

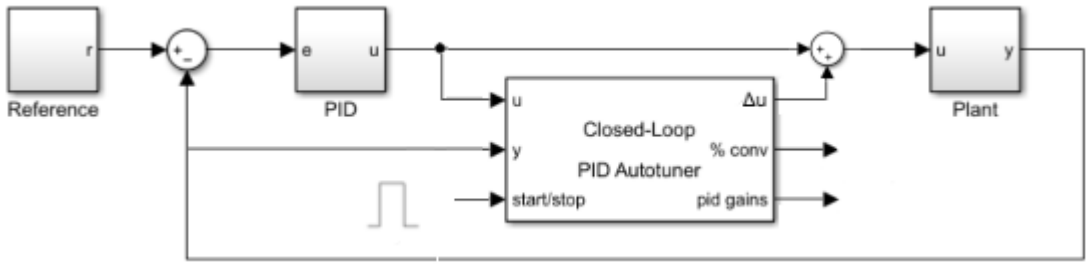
To enable this port, in **Output Signal Configuration**, select **control + perturbation**.

Data Types: `single` | `double`

Δu — Plant input perturbation

scalar

The block generates a perturbation signal at this port. Typically, you inject the perturbation from this port via a sum block, as shown in the following diagram.



- When the experiment is running (**start/stop** positive), the block generates perturbation signals at this port.
- When the experiment is not running (**start/stop** zero or negative), the signal at this port is zero. In this state, the block has no effect on the plant.

Dependencies

To enable this port, in **Output Signal Configuration**, select **perturbation only**.

Data Types: `single` | `double`

% conv — Convergence of FRD estimation during experiment

scalar

When the experiment is running (`start/stop` positive), the block injects test signals into the plant and measures the plant response at `y`. It uses these signals to estimate the frequency response of the plant at several frequencies around the target bandwidth for tuning. `% conv` indicates how close to completion the estimation of the plant frequency response is. Typically, this value quickly rises to

about 90% after the experiment begins, and then gradually converges to a higher value. Stop the experiment when it levels off near 100%.

Data Types: `single` | `double`

`pid gains` — Tuned PID coefficients

bus

This 4-element bus signal contains the tuned PID gains P, I, D, and the filter coefficient N. These values correspond to the `P`, `I`, `D`, and `N` parameters in the expressions given in the `Form` parameter. Initially, the values are 0, 0, 0, and 100, respectively. The block updates the values when the experiment ends. This bus signal always has four elements, even if you are not tuning a PIDF controller.

If you have a PID controller associated with the block, you can update that controller with these values after the experiment ends. To do so, in the Block tab, click **Update PID Block**.

Data Types: `single` | `double`

`estimated PM` — Estimated phase margin with tuned controller

scalar

This port outputs the estimated phase margin achieved by the tuned controller, in degrees. The block updates this value when the tuning experiment ends. The estimated phase margin is calculated from the angle of $G(j\omega_c)C(j\omega_c)$, where G is the estimated plant, C is the tuned controller, and ω_c is the crossover frequency (bandwidth). The estimated phase margin might differ from the target phase margin specified by the `Target phase margin (degrees)` parameter. It is an indicator of the robustness and stability achieved by the tuned system.

- Typically, the estimated phase margin is near the target phase margin. In general, the larger the value, the more robust is the tuned system, and the less overshoot there is.
- A negative phase margin indicates that the closed-loop system might be unstable.

Dependencies

To enable this port, in the Tuning tab, select **Output estimated phase margin achieved by tuned controller**.

`frd` — Estimated frequency response

vector

This port outputs the frequency-response data estimated by the experiment. Initially, the value at `frd` is [0, 0, 0, 0, 0]. During the experiment, the block injects signals at frequencies [1/10, 1/3, 1, 3, 10]ω_c, where ω_c is the target bandwidth. At each sample time during the experiment, the block updates `frd` with a vector containing the complex frequency response at each of these frequencies, respectively. You can use the progress of the response as an alternative to `% conv` to examine the convergence of the estimation. When the experiment stops, the block updates `frd` with the final estimated frequency response used for computing the PID gains.

Dependencies

To enable this port, in the Experiment tab, select **Plant frequency responses near bandwidth**.

`nominal` — Plant input and output at nominal operating point

vector

This port outputs a vector containing the plant input (**u+Δu**) and plant output (**y**) when the experiment begins. These values are the plant input and output at the nominal operating point at which the block performs the experiment.

Dependencies

To enable this port, in the Experiment tab, select **Plant nominal input and output**.

Parameters

Tuning Tab

Type — PID controller actions

`PI` (default) | `PID` | `PIDF` | ...

Specify the type of the PID controller in your system. The controller type indicates what actions are present in the controller. The following controller types are available for PID autotuning:

- `P` — Proportional only
- `I` — Integral only
- `PI` — Proportional and integral
- `PD` — Proportional and derivative
- `PDF` — Proportional and derivative with derivative filter
- `PID` — Proportional, integral, and derivative
- `PIDF` — Proportional, integral, and derivative with derivative filter

When you update a PID Controller block or custom PID controller with tuned parameter values, make sure the controller type matches.

Tunable: Yes

Programmatic Use

Block Parameter: `PIDType`
Type: character vector
Values: `'P'` | `'I'` | `'PI'` | `'PD'` | `'PDF'` | `'PID'` | `'PIDF'`
Default: `'PI'`

Form — PID controller form

`Parallel` (default) | `Ideal`

Specify the controller form. The controller form determines the interpretation of the PID coefficients P, I, D, and N.

- `Parallel` — In `Parallel` form, the transfer function of a discrete-time PIDF controller is:

  $$C = P + IF_i(z) + D\left[\frac{N}{1 + NF_d(z)}\right],$$

  where $F_i(z)$ and $F_d(z)$ are the integrator and filter formulas (see `Integrator method` and `Filter method`). The transfer function of a continuous-time parallel-form PIDF controller is:

  $$C = P + I\left(\frac{1}{s}\right) + D\left(\frac{Ns}{s + N}\right).$$

Other controller actions amount to setting P, I, or D to zero.

- `Ideal` — In `Ideal` form, the transfer function of a discrete-time PIDF controller is:

  $$C = P\left[1 + IF_i(z) + D\left(\frac{N}{1 + NF_d(z)}\right)\right].$$

  The transfer function of a continuous-time ideal-form PIDF controller is:

  $$C = P\left[1 + I\left(\frac{1}{s}\right) + D\left(\frac{Ns}{s + N}\right)\right].$$

  Other controller actions amount to setting D to zero or setting, I to `Inf`. (In ideal form, the controller must have proportional action.)

When you update a PID Controller block or custom PID controller with tuned parameter values, make sure the controller form matches.

Tunable: Yes

Programmatic Use
Block Parameter: `PIDForm`
Type: character vector
Values: `'Parallel'` | `'Ideal'`
Default: `'Parallel'`

`Time Domain` — PID controller time domain

discrete-time (default) | continuous-time

Specify whether your PID controller is a discrete-time or continuous-time controller.

- For discrete time, you must specify the sample time of your PID controller using the **Controller sample time (sec)** parameter.
- For continuous time, you must also specify a sample time for the PID autotuning experiment using the **Experiment sample time (sec)** parameter.

Programmatic Use
Block Parameter: `TimeDomain`
Type: character vector
Values: `'discrete-time'` | `'continuous-time'`
Default: `'discrete-time'`

`Controller sample time (sec)` — Sample time of PID controller

0.1 (default) | positive scalar | –1

Specify the sample time of your PID controller in seconds. This value also sets the sample time for the experiment performed by the block.

To perform PID tuning, the block measures frequency-response information up to a frequency of 10 times the target bandwidth. To ensure that this frequency is less than the Nyquist frequency, the target bandwidth, ω_c, must satisfy $\omega_c T_s \leq 0.3$, where T_s ω_c is the controller sample time that you specify with the `Controller sample time (sec)` parameter.

When you update a PID Controller block or custom PID controller with tuned parameter values, make sure the controller sample time matches.

Tips

If you want to run the deployed block with different sample times in your application, set this parameter to –1 and put the block in a Triggered Subsystem. Then, trigger the subsystem at the desired sample time. If you do not plan to change the sample time after deployment, specify a fixed and finite sample time.

Dependencies

To enable this parameter, set **Time Domain** to `discrete-time`.

Programmatic Use

Block Parameter: `DiscreteTs`
Type: scalar
Value positive scalar | –1
Default: 0.1

`Tune at different sample time` — Enable tuning at different sample time from PID controller and experiment

`off` (default) | `on`

Enable this parameter to run tuning at a sample rate that is different from the sample rate of the PID controller you are tuning and the frequency response estimation experiment performed by the block. The PID gain tuning algorithm is computationally intensive, and when you want to deploy the block to hardware and tune a controller with a fast sample time, some hardware might not complete the PID gain calculation in a single time step. To reduce the hardware throughput requirements, specify a tuning sample time slower than the controller sample time using the **Tuning sample time (sec)** parameter.

Dependencies

To enable this parameter, set **Time Domain** to `discrete-time`.

Programmatic Use

Block Parameter: `UseTuningTs`
Type: character vector
Value `'off'` | `'on'`
Default: `'off'`

`Tuning sample time (sec)` — Sample time of tuning algorithm

0.2 (default) | positive scalar

Specify the sample time of the tuning algorithm in seconds.

If you intend to deploy the block on hardware with limited processing power and want to tune a controller with a fast sample time, specify a sample time such that the tuning algorithm runs at a slower rate than the PID controller you are tuning.

Dependencies

To enable this parameter, set **Time Domain** to `discrete-time` and select **Tune at different sample time**.

Programmatic Use

Block Parameter: `TuningTs`
Type: scalar

Value positive scalar
Default: 0.2

`Experiment sample time (sec)` — Sample time for experiment

0.02 (default) | positive scalar

Even when you tune a continuous-time controller, you must specify a sample time for the experiment performed by the block. In general, continuous-time controller tuning is not recommended for PID autotuning against a physical plant. If you want to tune in continuous time against a Simulink model of the plant, use a fast experiment sample time, such as $0.02/\omega_c$.

Dependencies

This parameter is enabled when the **Time Domain** is `continuous-time`.

Programmatic Use

Block Parameter: `ContinuousTs`
Type: positive scalar
Default: 0.02

`Integrator method` — Discrete integration formula for integrator term

`Forward Euler` (default) | `Backward Euler` | `Trapezoidal`

Specify the discrete integration formula for the integrator term in your controller. In discrete time, the PID controller transfer function assumed by the block is:

$$C = P + IF_i(z) + D\left[\frac{N}{1 + NF_d(z)}\right],$$

in parallel form, or in ideal form,

$$C = P\left[1 + IF_i(z) + D\left(\frac{N}{1 + NF_d(z)}\right)\right].$$

For a controller sample time T_s, the `Integrator method` parameter determines the formula F_i as follows:

Integrator method	F_i
`Forward Euler`	$\frac{T_s}{z-1}$
`Backward Euler`	$\frac{T_s z}{z-1}$
`Trapezoidal`	$\frac{T_s}{2}\frac{z+1}{z-1}$

For more information about the relative advantages of each method, see the Discrete PID Controller block reference page.

When you update a PID Controller block or custom PID controller with tuned parameter values, make sure the integrator method matches.

Tunable: Yes

Dependencies

This parameter is enabled when the **Time Domain** is `discrete-time` and the controller includes integral action.

Programmatic Use

Block Parameter: `IntegratorFormula`
Type: character vector
Values: `'Forward Euler'` | `'Backward Euler'` | `'Trapezoidal'`
Default: `'Forward Euler'`

`Filter method` — Discrete integration formula for derivative filter term

`Forward Euler` (default) | `Backward Euler` | `Trapezoidal`

Specify the discrete integration formula for the derivative filter term in your controller. In discrete time, the PID controller transfer function assumed by the block is:

$$C = P + IF_i(z) + D\left[\frac{N}{1 + NF_d(z)}\right],$$

in parallel form, or in ideal form,

$$C = P\left[1 + IF_i(z) + D\left(\frac{N}{1 + NF_d(z)}\right)\right].$$

For a controller sample time T_s, the `Filter method` parameter determines the formula F_d as follows:

Filter method	F_d
`Forward Euler`	$\frac{T_s}{z-1}$
`Backward Euler`	$\frac{T_s z}{z-1}$
`Trapezoidal`	$\frac{T_s}{2}\frac{z+1}{z-1}$

For more information about the relative advantages of each method, see the Discrete PID Controller block reference page.

When you update a PID Controller block or custom PID controller with tuned parameter values, make sure the filter method matches.

Tunable: Yes

Dependencies

This parameter is enabled when the **Time Domain** is `discrete-time` and the controller includes a derivative filter term.

Programmatic Use

Block Parameter: `FilterFormula`
Type: character vector
Values: `'Forward Euler'` | `'Backward Euler'` | `'Trapezoidal'`

Default: `'Forward Euler'`

`Target bandwidth (rad/sec)` — Target crossover frequency of tuned response

1 (default) | positive scalar

The target bandwidth, specified in rad/sec, is the target value for the 0-dB gain crossover frequency of the tuned open-loop response CP, where P is the plant response, and C is the controller response. This crossover frequency roughly sets the control bandwidth. For a rise-time τ seconds, a good guess for the target bandwidth is $2/\tau$ rad/sec.

To perform PID tuning, the autotuner block measures frequency-response information up to a frequency of 10 times the target bandwidth. To ensure that this frequency is less than the Nyquist frequency, the target bandwidth, ω_c, must satisfy $\omega_c T_s \leq 0.3$, where T_s is the controller sample time that you specify with the **Controller sample time (sec)** parameter. Because of this condition, the fastest rise time you can enforce for tuning is about $6.67T_s$. If this rise time does not meet your design goals, consider reducing T_s.

For best results with closed-loop tuning, use a target bandwidth that is within about a factor of 10 of the bandwidth with the initial PID controller. To tune a controller for a larger change in bandwidth, tune incrementally using smaller changes.

To provide the target bandwidth via an input port, select **Use external source**.

Programmatic Use

Block Parameter: `Bandwidth`
Type: positive scalar
Default: 1

`Target phase margin (degrees)` — Target minimum phase margin of open-loop response

60 (default) | scalar in range 0–90

Specify a target minimum phase margin for the tuned open-loop response at the crossover frequency. The target phase margin reflects desired robustness of the tuned system. Typically, choose a value in the range of about 45°–60°. In general, higher phase margin improves overshoot, but can limit response speed. The default value, 60°, tends to balance performance and robustness, yielding about 5–10% overshoot, depending on the characteristics of your plant.

To provide the target phase margin via an input port, select **Use external source**.

Tunable: Yes

Programmatic Use

Block Parameter: `TargetPM`
Type: scalar
Values: 0–90
Default: 60

Experiment Tab

`Experiment Mode` — Sinusoidal perturbation signal type

`Superposition` (default) | `Sinestream`

Specify whether the perturbation at each frequency is applied sequentially (**Sinestream**) or simultaneously (**Superposition**).

- **Sinestream** — In this mode, the block applies perturbation at each frequency separately. For more information about sinestream signals for estimation, see “Sinestream Input Signals” on page 5-29.
- **Superposition** — In this mode, the perturbation signal includes all specified frequencies at once. For frequency response estimation at a vector of frequencies $\omega = [\omega_1, \ldots, \omega_N]$ at amplitudes $A = [A_1, \ldots, A_N]$, the perturbation signal is:

$$\Delta u = \sum_i A_i \sin(\omega_i t).$$

Sinestream mode can be more accurate and can also be less intrusive, because the total size of the perturbation is never bigger than the values specified by the **Sine Amplitudes** parameter. However, due to the sequential nature of the sinestream perturbation, each frequency point you add increases the recommended experiment time (see the **start/stop** input port for details). Thus, the estimation experiment is typically much faster in **Superposition** mode with satisfactory results.

Sinestream signals reduce the execution time compared to superposition input signals, but also take longer to estimate the frequency response. Frequency response estimation using sinestream signals is useful when you have limited processing power and you want to reduce the execution time.

Programmatic Use

Block Parameter: `ExperimentMode`
Type: character vector
Values: `'Superposition'` | `'Sinestream'`
Default: `'Superposition'`

`Plant Type` — Stability of plant

`Stable` (default) | `Integrating`

Specify whether the plant is stable or integrating. If the plant has one or more integrators, select `Integrating`.

Programmatic Use

Block Parameter: `PlantType`
Type: character vector
Values: `'Stable'` | `'Integrating'`
Default: `'Stable'`

`Plant Sign` — Sign of plant

`Positive` (default) | `Negative`

Specify whether the plant is positive or negative. If a positive change in the plant input at the nominal operating point results in a positive change in the plant output, specify `Positive`. Otherwise, specify negative. For stable plants, the sign of the plant is the sign of the plant DC gain.

Programmatic Use

Block Parameter: `PlantSign`
Type: character vector
Values: `'Positive'` | `'Negative'`
Default: `'Positive'`

`Sine Amplitudes` — Amplitude of sinusoidal perturbations

1 (default) | scalar | vector of length 5

During the experiment, the block injects a sinusoidal signal into the plant at the frequencies [1/10, 1/3, 1, 3, 10]ω_c , where ω_c is the target bandwidth for tuning. Use **Sine Amplitudes** to specify the amplitude of each of these injected signals. Specify a:

- Scalar value to inject the same amplitude at each frequency
- Vector of length 5 to specify a different amplitude at each of [1/10, 1/3, 1, 3, 10]ω_c

In a typical plant with typical target bandwidth, the magnitudes of the plant responses at the experiment frequencies do not vary widely. In such cases, you can use a scalar value to apply the same magnitude perturbation at all frequencies. However, if you know that the response decays sharply over the frequency range, consider decreasing the amplitude of the lower-frequency inputs and increasing the amplitude of the higher-frequency inputs. It is numerically better for the estimation experiment when all the plant responses have comparable magnitudes.

The perturbation amplitudes must be:

- Large enough that the perturbation overcomes any deadband in the plant actuator and generates a response above the noise level
- Small enough to keep the plant running within the approximately linear region near the nominal operating point, and to avoid saturating the plant input or output

When **Experiment mode** is **Superposition**, the sinusoidal signals are superimposed. Thus, the perturbation can be at least as large as the sum of all amplitudes. Make sure that the largest possible perturbation is within the range of your plant actuator. Saturating the actuator can introduce errors into the estimated frequency response.

To provide the sine amplitudes via an input port, select **Use external source**.

Tunable: Yes

Programmatic Use

Block Parameter: `AmpSine`
Type: scalar, vector of length 5
Default: 1

Block Tab

`Reduce memory and avoid task overrun (external mode only)` — Deploy tuning algorithm only

`off` (default) | `on`

The block contains two modules, one that performs the real-time frequency-response estimation, and one that uses the resulting estimated response to tune the PID gains. When you run a Simulink model containing the block in the external simulation mode, by default both modules are deployed. You can save memory on the target hardware by deploying the estimation module only (see “Control Real-Time PID Autotuning in Simulink” on page 8-20). In this case, the tuning algorithm runs on the Simulink host computer instead of the target hardware. When this option is selected, the deployed algorithm uses about a third as much memory as when the option is cleared.

The PID gain calculation demands more computational load than the frequency-response estimation. For fast controller sample times, some hardware might not finish the gain calculation within one execution cycle. Therefore, when using hardware with limited computing power, selecting this option lets you tune a PID controller with a fast sample time.

Additionally, when you enable this option, there can be a delay of several sampling periods between when the tuning experiment ends and when the new PID gains arrive at the **pid gains** output port. Before pushing gains to the controller, first confirm the change at the **pid gains** output port instead of using **start/stop** signal as the trigger for the update.

If you intend to deploy the block and perform PID tuning without using external simulation mode, do not select this option.

Caution When you use this option, the model must be configured such that numeric block parameters are tunable in generated code, not inlined. To specify tunable parameters:

- In the model editor: In **Configuration Parameters**, in **Code Generation** > **Optimization**, set **Default parameter behavior** to `Tunable`.
- At the command line: Use `set_param(mdl,'DefaultParameterBehavior','Tunable')`.

Programmatic Use

Block Parameter: `DeployTuningModule`
Type: character vector
Values: `'off'` | `'on'`
Default: `'off'`

`Configure block for PLC Coder` — Configure block for code generation with Simulink PLC Coder

off (default) | on

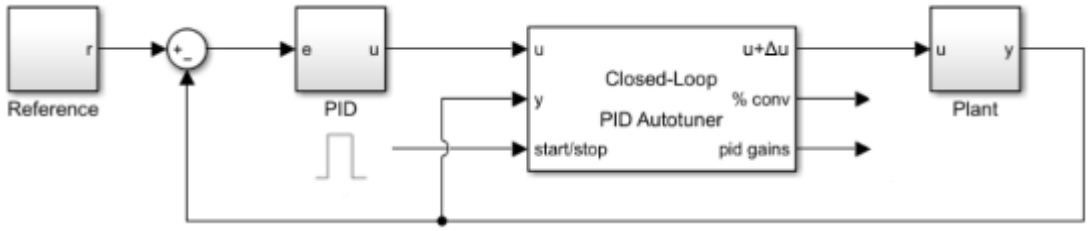
Select this parameter if you are using Simulink PLC Coder to generate code for the autotuner block. Clear the parameter for code generation with any other MathWorks code-generation product.

Selecting this parameter affects internal block configuration only, for compatibility with Simulink PLC Coder. The parameter has no operative effect on generated code.

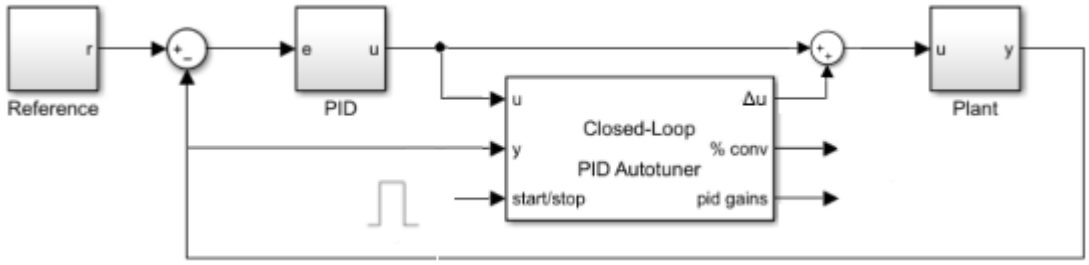
`Output Signal Configuration` — Provide control signal plus perturbation or perturbation only

control action + perturbation (default) | **perturbation only**

By default, the block takes a control signal as input and provides the control signal plus the experiment perturbation at the port **u+Δu**. You then feed this signal into the plant input, as shown in the following diagram.



This default configuration requires inserting the block between the controller and the plant. If you want to add the perturbation signal to the control signal yourself, select **perturbation only**. In this configuration, the block output contains the perturbation signal only, at the port **Δu**. You inject this perturbation signal into the plant using, for example, a sum block, as in the following diagram.



In this configuration, you can optionally comment out the Closed-Loop PID Autotuner block without disrupting the model.

`Data Type` — Floating point precision
`double` (default) | `single`

Specify the floating-point precision based on simulation environment or hardware requirements.

Programmatic Use
Block Parameter: `BlockDataType`
Type: character vector
Values: `'double'` | `'single'`
Default: `'double'`

`Clicking "Update PID Block" writes tuned gains to the PID block connected to "u" port` — Automatically detect target for writing tuned PID coefficients
`on` (default) | `off`

Under some conditions, the autotuner block can write tuned gains to a standard or custom PID controller block. To indicate that the target PID controller is the block connected to the **u** port of the autotuner block, select this option. To specify a PID controller that is not connected to **u**, clear this option.

To write tuned gains from the autotuner block to a PID controller anywhere in the model, the target block must be either:

- A PID Controller or Discrete PID Controller block.
- A masked subsystem in which the PID coefficients are mask parameters named `P`, `I`, `D`, and `N`, or whatever subset of these parameters exist in the your controller. For example, if you use a custom PI controller, then you only need mask parameters `P` and `I`.

`Specify PID block path` — Target PID controller block for writing tuned coefficients
`[ ]` (default) | block path

Under some conditions, the autotuner block can write tuned gains to a standard or custom PID controller block. Use this parameter to specify the path of the target PID controller.

To write tuned gains from the autotuner block to a PID controller anywhere in the model, the target block must be either:

- A PID Controller or Discrete PID Controller block.
- A masked subsystem in which the PID coefficients are mask parameters named `P`, `I`, `D`, and `N`, or whatever subset of these parameters exist in your controller

Dependencies

This parameter is enabled when **Clicking "Update PID Block" writes tuned gains to the PID block connected to "u" port** is selected.

`Update PID Block` — Write tuned PID gains to target controller block

button

The block does not automatically push the tuned gains to the target PID block. If your PID controller block meets the criteria described in the `Specify PID block path` parameter description, after tuning, click this button to transfer the tuned gains to the block.

You can update the PID block while the simulation is running, including when running in external mode. Doing so is useful for immediately validating tuned PID gains. At any time during simulation, you can change parameters, start the experiment again, and push the new tuned gains to the PID block. You can then continue to run the model and observe the behavior of your plant.

`Export to MATLAB` — Send experiment and tuning results to MATLAB workspace

button

When you click this button, the block creates a structure in the MATLAB workspace containing the experiment and tuning results. This structure, `OnlinePIDTuningResult`, contains the following fields:

- `P`, `I`, `D`, `N` — Tuned PID gains. The structure contains whichever of these fields are necessary for the controller type you are tuning. For instance, if you are tuning a PI controller, the structure contains `P` and `I`, but not `D` and `N`.
- `TargetBandwidth` — The value you specified in the **Target bandwidth (rad/sec)** parameter of the block.
- `TargetPhaseMargin` — The value you specified in the **Target phase margin (degrees)** parameter of the block.
- `EstimatedPhaseMargin` — Estimated phase margin achieved by the tuned system.
- `Controller` — The tuned PID controller, returned as a `pid` (for parallel form) or `pidstd` (for ideal form) model object.
- `Plant` — The estimated plant, returned as an `frd` model object. This `frd` contains the response data obtained at the experiment frequencies [1/10, 1/3, 1, 3, 10]ω_c.
- `PlantNominal` — The plant input and output at the nominal operating point when the experiment begins, specified as a structure having fields `u` (input) and `y` (output).

You can export to the MATLAB workspace while the simulation is running, including when running in external mode.

Extended Capabilities

C/C++ Code Generation

Generate C and C++ code using Simulink® Coder™.

PLC Code Generation

Generate Structured Text code using Simulink® PLC Coder™.

See Also

Open-Loop PID Autotuner

Topics

“PID Autotuning for a Plant Modeled in Simulink” on page 8-7
“PID Autotuning in Real Time” on page 8-13
“When to Use PID Autotuning” on page 8-2
“How PID Autotuning Works” on page 8-5

Introduced in R2018a

Constraint Enforcement

Modify control actions to satisfy constraints and action bounds
Library: Simulink Control Design

Description

The Constraint Enforcement block computes the modified control actions that are closest to specified control actions subject to constraints and action bounds.

The block uses a quadratic programming (QP) solver to find the control action u that minimizes the function $|u - u_0|^2$. Here, u_0 is the unmodified control action.

The solver applies the following constraints to the optimization problem.

$$f_x + g_x u \le c$$
$$u_{\min} \le u \le u_{\max}$$

Here:

- f_x and g_x are coefficients of the constraint function.
- c is a bound for the constraint function.
- $u_{\min}$ is a lower bound for the control action.
- $u_{\max}$ is an upper bound for the control action.

The Constraint Enforcement block requires Optimization Toolbox software.

For more information on constraint enforcement, see “Constraint Enforcement for Control Design” on page 16-2.

Ports

Input

`u0` — Control actions

scalar | vector

Unmodified control actions, specified as a scalar or a vector.

If the **Number of actions** parameter is `1`, connect **u0** to a scalar signal. Otherwise, connect **u0** to a vector signal with length equal to **Number of actions**.

`fx` — Constraint function offset coefficient

scalar | vector

Offset coefficient f_x in the following constraint equation.

$$f_x + g_x u \le c$$

If the **Number of constraints** parameter is `1`, connect **fx** to a scalar signal. Otherwise, connect **fx** to a vector signal with length equal to **Number of constraints**.

gx — Constraint function linear coefficient

scalar | vector | matrix

Linear coefficient g_x in the following constraint equation.

$$f_x + g_x u \le c$$

Connect **gx** to an N_c-by-N_u signal, where N_c is equal to the **Number of constraints** parameter and N_u is equal to the **Number of actions** parameter.

c — Constraint bounds

scalar | vector

Run-time constraint bound c in the following constraint function.

$$f_x + g_x u \le c$$

If the **Number of constraints** parameter is `1`, connect **c** to a scalar signal. Otherwise, connect **c** to a vector signal with length equal to **Number of constraints**.

If this port is disabled, the block uses the constant constraint bounds specified using the **Constraint bound** parameter.

Dependencies

To enable this input port, select the **Use external source** parameter.

umax — Action signal upper bounds

scalar | vector

To specify run-time upper bounds to the action signals, enable this input port. If this port is disabled, the block does not apply any upper bounds to the control actions.

If the **Number of actions** parameter is `1`, connect **umax** to a scalar signal. Otherwise, connect **umax** to a vector signal with length equal to **Number of actions**.

Dependencies

To enable this input port, select the **Use external source for upper bound** parameter.

umin — Action signal lower bounds

scalar | vector

To specify run-time lower bounds to the action signals, enable this input port. If this port is disabled, the block does not apply any lower bounds to the control actions.

If the **Number of actions** parameter is `1`, connect **umin** to a scalar signal. Otherwise, connect **umin** to a vector signal with length equal to **Number of actions**.

Dependencies

To enable this input port, select the **Use external source for lower bound** parameter.

Output

u* — Modified control action

scalar | vector

Modified control action returned by the QP solver.

If the solver finds a solution before reaching the maximum number of iterations, **u*** outputs this optimal solution.

If the solver reaches the maximum number of iterations, optimization stops and **u*** outputs a suboptimal solution.

If the initial optimization problem is infeasible, the returned control action depends on the whether the block is configured to ignore constraint or action bounds. For more information, see the **exitflag** parameter.

If the **Number of actions** parameter is `1`, **u*** outputs a scalar signal. Otherwise, **u*** outputs a vector signal with length equal to **Number of actions**.

exitflag — Optimization status

`1` | `2` | `3` | `4` | `0` | negative integer

Optimization status of the QP solver. The following table shows the possible status values.

Exit Flag	Description
`1`	The solver converged to an optimal solution with all constraints and bounds active. In this case, **u*** outputs the optimal control actions.
`2`	The initial optimization problem was infeasible and the block is configured to ignore all constraints and bounds. In this case, **u*** outputs the unmodified control action **u0**.
`3`	The initial optimization problem was infeasible. The block reran the optimization ignoring the action bounds and found a feasible solution, which the block outputs in **u***.
`4`	The initial optimization problem was infeasible. The block reran the optimization ignoring the constraint bounds and found a feasible solution, which the block outputs in **u***.
`0`	The solver reached the maximum number of iterations. The control actions output in **u*** might be suboptimal.

Exit Flag	Description
negative integer	The initial optimization problem was infeasible and one of the following scenarios applies. • Rerunning the optimization without action bounds did not produce a feasible solution. • Rerunning the optimization without constraint bounds did not produce a feasible solution. • The block is configured to not ignore constraint and action bounds. In this case, the control actions output in **u*** are zero.

Dependencies

To enable this output port, select the **Optimization status** parameter.

Parameters

Parameters Tab

`Number of constraints` — Number of constraints

`1` (default) | positive integer

Specify the number of constraints to enforce.

Programmatic Use

Block Parameter: `nc`
Type: character vector
Default: `'1'`

`Number of actions` — Number of actions

`1` (default) | positive integer

Specify the number of actions to apply bounds to and optimize.

Programmatic Use

Block Parameter: `nu`
Type: character vector
Default: `'1'`

`Constraint bound` — Constraint bounds

`0` (default) | finite scalar | vector

Specify constant bounds for constraints. If the **Number of constraints** parameter is `1`, specify **Constraint bound** as a finite scalar. Otherwise, specify **Constraint bound** as a vector of finite value with length equal to **Number of constraints**.

If your constraints vary at run time, select the **Use external source** parameter and connect the run-time constraint signal to the **c** input port.

Dependencies

To enable this parameter, clear the **Use external source** parameter.

Programmatic Use
Block Parameter: `c`
Type: character vector
Default: `'0'`

`Use external source` — Add external constraint bound input port
`off` (default) | `on`

Select this parameter to add the **c** input port for external constraint bounds.

Programmatic Use
Block Parameter: `external_c`
Type: character vector
Values: `'off'|'on'`
Default: `'off'`

`Use external source for upper bound` — Add upper action bound input port
`off` (default) | `on`

Select this parameter to add the **umax** input port for external upper action bounds.

Programmatic Use
Block Parameter: `external_umax`
Type: character vector
Values: `'off'|'on'`
Default: `'off'`

`Use external source for lower bound` — Add lower action bound input port
off (default) | on

Select this parameter to add the **umin** input port for external lower action bounds.

Programmatic Use
Block Parameter: `external_umin`
Type: character vector
Values: `'off'|'on'`
Default: `'off'`

Block Tab

`Sample time` — Optimization sample time
`0.1` (default) | positive scalar

Specify the sample time for running the optimization.

Programmatic Use
Block Parameter: `Ts`
Type: character vector
Default: `'0.1'`

`Maximum iterations` — Maximum optimization iterations
`200` (default) | positive integer

Specify the maximum number of optimization iterations.

Programmatic Use
Block Parameter: `maxiter`
Type: character vector
Default: `'200'`

`Constraint tolerance` — Tolerance for constraint violations
`1e-6` (default) | nonnegative scalar

Specify a tolerance value for constraint violations.

Programmatic Use
Block Parameter: `tol`
Type: character vector
Default: `'1e-6'`

`Optimization status` — Add exit flag output port
`off` (default) | `on`

Select this parameter to add the **exitflag** output port for the optimization status of the QP solver.

Programmatic Use
Block Parameter: `exitflag`
Type: character vector
Values: `'off'|'on'`
Default: `'off'`

`Ignore constraints when QP is infeasible` — Disable constraints when optimization is infeasible
`off` (default) | `on`

When you select this parameter, if the initial QP problem is infeasible, the block reruns the optimization with the constraints disabled.

When you select both this parameter and **Ignore action bounds when QP is infeasible**, if the initial QP problem is infeasible, the block outputs the unmodified action signal.

Programmatic Use
Block Parameter: `relax_c`
Type: character vector
Values: `'off'|'on'`
Default: `'off'`

`Ignore action bounds when QP is infeasible` — Disable action bounds when optimization is infeasible
`off` (default) | `on`

When you select this parameter, if the initial QP problem is infeasible, the block reruns the optimization with the action bounds disabled. The block ignores this parameter if both the **umax** and **umin** input ports are disabled.

When you select both this parameter and **Ignore constraints when QP is infeasible**, if the initial QP problem is infeasible, the block outputs the unmodified action signal.

Programmatic Use
Block Parameter: `relax_u`

Type: character vector
Values: `'off'|'on'`
Default: `'off'`

Extended Capabilities

C/C++ Code Generation

Generate C and C++ code using Simulink® Coder™.

The Constraint Enforcement block supports code generation for double-precision signals only.

See Also

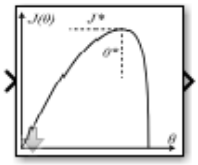
Topics

“Constraint Enforcement for Control Design” on page 16-2

Introduced in R2021a

Extremum Seeking Control

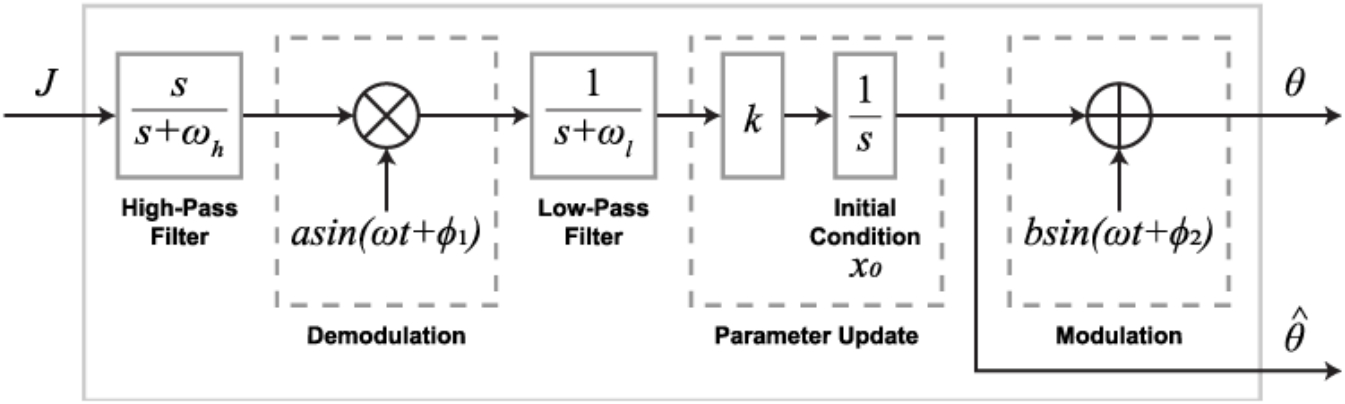
Compute controller parameters in real time by maximizing objective function
Library: Simulink Control Design



Description

The Extremum Seeking Control block tunes controller parameters to maximize an objective function. Extremum seeking controllers are model-free adaptive controllers that are useful for adapting to unknown system dynamics and unknown mappings from control parameters to an objective function. When seeking multiple parameters, the Extremum Seeking Control block uses a separate tuning loop for each parameter.

The Extremum Seeking Control block searches for optimal control parameters by modulating (perturbing) the parameters with sinusoidal signals and demodulating the resulting perturbed objective function.



To configure the extremum seeking algorithm, set appropriate forcing frequencies, demodulation and modulation signals, learning rates, and parameter initial conditions. When configuring the block, ensure that the system dynamics are on the fastest time scale, the perturbation forcing frequencies are on the medium time scale, and the filter cutoff frequencies are on the slowest time scale.

You can implement both continuous-time and discrete-time extremum seeking controllers. Changing the time-domain of the controller affects the time domain of the high-pass filters, low-pass filters, and integrators used in the tuning loops. To generate hardware-deployable code for the Extremum Seeking Control block, use a discrete-time controller.

For more information, see “Extremum Seeking Control” on page 15-2.

Ports

Input

`J` — Objective function
scalar

The objective function from the control system is perturbed as a result of the system response to the perturbed parameters **theta**. To compute parameter updates, the block filters and demodulates **J**.

Output

`theta` — Perturbed parameters
scalar | vector

Apply these perturbed parameters (θ) to your control system. The block uses the resulting perturbation of the objective function **J** to compute parameter updates.

If **Number of parameters** is `1`, **theta** outputs a scalar signal. Otherwise, **theta** outputs a vector signal with length equal to **Number of parameters**.

`theta_hat` — Estimated parameters
scalar | vector

Use this output port to obtain the estimated parameter values ($\widehat{\theta}$) before they are perturbed by the modulation signal.

If **Number of parameters** is `1`, **theta_hat** outputs a scalar signal. Otherwise, **theta_hat** outputs a vector signal with length equal to **Number of parameters**.

Dependencies

To enable this output port, select **Output estimated parameters**.

Parameters

`Time Domain` — Filter and integrator time domain
Continuous time (default) | **Discrete time**

Specify the time domain for the high-pass filters, low-pass filters, and integrators.

- **Continuous time** — Use continuous-time filters and integrators.
- **Discrete time** — Use discrete-time filters and integrators. Specify the sample time using the **Sample time** parameter, and specify the integration method using the **Integration method** parameter.

Programmatic Use
Block Parameter: `'timeDomainStatus'`
Type: character vector
Values: `'Continuous Time'` | `'Discrete Time'`
Default: `'Continuous Time'`

`Sample time` — Discrete-time filter and integrator sample time
`0.1` (default) | finite positive scalar

Specify the sample time for the discrete-time high-pass filters, low-pass filters, and integrators.

When tuning multiple parameters, all tuning loops use the same sample time.

Dependencies

To enable this parameter, set the **Time Domain** property to **Discrete time**.

Programmatic Use
Block Parameter: `'SampleTime'`
Type: character vector
Values: finite positive scalar
Default: `'0.1'`

`Integrator method` — Discrete-time integration method

`Forward Euler` (default) | `Backward Euler` | `Trapezoidal`

You can select one of the following integration methods for the discrete-time integrators.

- `Forward Euler`:

  $y(n) = y(n-1) + T_s \cdot u(n-1)$

- `Backward Euler`:

  $y(n) = y(n-1) + T_s \cdot u(n)$

- `Trapezoidal`:

  $y(n) = y(n-1) + T_s \cdot [u(n) + u(n-1)]/2$

Here:

- y is the integrator output
- u is the input
- n is the current sample time
- T_s is the sample time

When tuning multiple parameters, the integrators in all tuning loops use the same integration method.

Dependencies

To enable this parameter, set the **Time Domain** property to **Discrete time**.

Programmatic Use
Block Parameter: `'IntegratorMethods'`
Type: character vector
Values: `'Forward Euler'` | `'Backward Euler'` | `'Trapezoidal'`
Default: `'Forward Euler'`

`Number of parameters` — Number of parameters

1 (default) | positive integer less than or equal to 5

You can simultaneously tune up to five parameters. The block uses a separate tuning loop for each parameter.

Programmatic Use

Block Parameter: `'paramCount'`
Type: character vector
Values: positive integer less than 5
Default: `'1'`

`Initial condition x0` — Initial parameter values

`0` (default) | finite scalar | vector

Initial parameter values, which correspond to the initial conditions of the parameter update integrators.

If **Number of parameters** is `1`, specify **Initial condition** as a finite scalar.

To specify different initial conditions for multiple parameters, specify **Initial condition** as a vector of finite values with length equal to **Number of parameters**. Otherwise, to specify the same initial condition for all parameters, specify **Initial condition** as a finite scalar.

Programmatic Use

Block Parameter: `'initialVal'`
Type: character vector
Values: finite scalar | vector
Default: `'0'`

`Forcing frequency omega` — Forcing frequency

`1` (default) | positive finite scalar | vector

Specify the frequency of the modulation and demodulation signals in radians per second. For a given parameter tuning loop, specify a forcing frequency that is lower than the frequencies of important system dynamics and higher than the high-pass and low-pass filter cutoff frequencies.

When tuning a single parameter, specify **Forcing frequency** as a positive finite scalar.

When tuning a multiple parameters, specify **Forcing frequency** as a vector of positive finite values with length equal to **Number of parameters**. Each forcing frequency must be unique, which allows convergence of the extremum-seeking algorithm.

Programmatic Use

Block Parameter: `'omega'`
Type: character vector
Values: positive finite scalar | vector
Default: `'1'`

`Learning rate k` — Parameter update rate

`1` (default) | positive finite scalar | vector

The learning rate is a gain factor that controls the rate at which the block updates a parameter.

When tuning a single parameter, specify **Learning rate** as a positive finite scalar.

When tuning a multiple parameters, you can specify a different learning rate for each parameter tuning loop. To do so, specify **Learning rate** as a vector of positive finite values with length equal to **Number of parameters**. Otherwise, to specify the same learning rate for all tuning loops, specify **Learning rate** as a positive finite scalar.

Programmatic Use
Block Parameter: `'gain'`
Type: character vector
Values: positive finite scalar | vector
Default: `'1'`

`Demodulation amplitude a` — Demodulation signal amplitude

`1` (default) | positive finite scalar | vector

Specify the amplitude of the signal used to demodulate the objective function. For most applications, specify **Demodulation amplitude** ≫ **Modulation amplitude**. The product of these amplitudes, along with the learning rate, controls the convergence speed of the algorithm.

When tuning a single parameter, specify **Demodulation amplitude** as a positive finite scalar.

When tuning multiple parameters, you can specify a different demodulation amplitude for each parameter tuning loop. To do so, specify **Demodulation amplitude** as a vector of positive finite values with length equal to **Number of parameters**. Otherwise, to specify the same amplitude for all tuning loops, specify **Demodulation amplitude** as a positive finite scalar.

Programmatic Use
Block Parameter: `'demodAmp'`
Type: character vector
Default: `'1'`

`Demodulation phase phi_1` — Demodulation signal phase

`0` (default) | positive finite scalar | vector

Specify the phase ϕ_1 of the signal used to demodulate the objective function in radians.

When tuning a single parameter, specify **Demodulation phase** as a positive finite scalar.

When tuning multiple parameters, you can specify a different demodulation phase for each parameter tuning loop. To do so, specify **Demodulation phase** as a vector of positive finite values with length equal to **Number of parameters**. Otherwise, to specify the same phase for all tuning loops, specify **Demodulation phase** as a positive finite scalar.

The demodulation and modulation phases must satisfy the condition $\cos(\phi_1–\phi_2) > 0$.

Programmatic Use
Block Parameter: `'demodPhase'`
Type: character vector
Values: positive finite scalar | vector
Default: `'0'`

`Modulation amplitude b` — Modulation signal amplitude

`0.1` (default) | positive finite scalar | vector

Amplitude of the perturbation signal added to the estimated parameters. For most applications, specify **Modulation amplitude** ≪ **Demodulation amplitude**. The product of these amplitudes, along with the learning rate, controls the convergence speed of the algorithm.

When tuning a single parameter, specify **Modulation amplitude** as a positive finite scalar.

When tuning multiple parameters, you can specify a different modulation amplitude for each parameter tuning loop. To do so, specify **Modulation amplitude** as a vector of positive finite values

with length equal to **Number of parameters**. Otherwise, to specify the same amplitude for all tuning loops, specify **Modulation amplitude** as a positive finite scalar.

Programmatic Use
Block Parameter: `'modAmp'`
Type: character vector
Values: positive finite scalar | vector
Default: `'0.1'`

`Modulation phase phi_2` — Modulation signal phase

`0` (default) | positive finite scalar | vector

Phase ϕ_2 of the perturbation signal added to the estimated parameters, specified in radians. You must select the demodulation phase ϕ_1 and modulation phase ϕ_2 such that $\cos(\phi_1 - \phi_2) > 0$.

When tuning a single parameter, specify **Modulation phase** as a positive finite scalar.

When tuning multiple parameters, you can specify a different modulation phase for each parameter tuning loop. To do so, specify **Modulation phase** as a vector of positive finite values with length equal to **Number of parameters**. Otherwise, to specify the same phase for all tuning loops, specify **Modulation phase** as a positive finite scalar.

Programmatic Use
Block Parameter: `'modPhase'`
Type: character vector
Values: positive finite scalar | vector
Default: `'0'`

`Enable HPF` — Enable high-pass filtering of objective function signal

`off` (default) | `on`

Select this parameter to enable a high-pass filter that removes any signal bias from the objective function signal before the demodulation stage. To specify the filter cutoff frequency, use the **HPF frequency** parameter.

Programmatic Use
Block Parameter: `'highPassEnable'`
Type: character vector
Values: `'off'` | `'on'`
Default: `'off'`

`HPF frequency omega_h` — High-pass filter cutoff frequency

`1` (default) | positive finite scalar | vector

Cutoff frequency ω_h for high-pass filtering the objective function signal, specified in radians per second. For a given parameter tuning loop, specify a cutoff frequency such that $\omega_h < \omega/(2\pi)$, where ω is the corresponding forcing frequency.

When tuning a single parameter, specify **HPF frequency** as a positive finite scalar.

When tuning multiple parameters, you can specify a different frequency for each parameter tuning loop. To do so, specify **HPF frequency** as a vector of positive finite values with length equal to **Number of parameters**. Otherwise, to specify the same frequency for all tuning loops, specify **HPF frequency** as a positive finite scalar.

Dependencies

To enable this parameter, select the **Enable HPF** parameter.

Programmatic Use

Block Parameter: `'highPassCutoff'`
Type: character vector
Values: positive finite scalar | vector
Default: `'1'`

`Enable LPF` — Enable low-pass filtering of demodulated signal

`off` (default) | `on`

Select this parameter to enable a low-pass filter that removes high-frequency components from the demodulated signal before the parameter update stage. To specify the filter cutoff frequency, use the **LPF frequency** parameter.

Dependencies

To enable this parameter, select the **Enable LPF** parameter.

Programmatic Use

Block Parameter: `'lowPassEnable'`
Type: character vector
Values: `'off'` | `'on'`
Default: `'off'`

`LPF frequency omega_l` — Low-pass filter cutoff frequency ω_l

1 (default) | positive finite scalar | vector

Cutoff frequency ω_l for low-pass filtering the demodulated signal, specified in radians per second. For a given parameter tuning loop, specify a cutoff frequency such that $\omega_l > \omega/(2\pi)$, where ω is the corresponding forcing frequency.

When tuning a single parameter, specify **LPF frequency** as a positive finite scalar.

When tuning multiple parameters, you can specify a different frequency for each parameter tuning loop. To do so, specify **LPF frequency** as a vector of positive finite values with length equal to **Number of parameters**. Otherwise, to specify the same frequency for all tuning loops, specify **LPF frequency** as a positive finite scalar.

Dependencies

To enable this parameter, select the **Enable LPF** parameter.

Programmatic Use

Block Parameter: `'lowPassCutoff'`
Type: character vector
Values: positive finite scalar | vector
Default: `'1'`

`Output estimated parameters` — Add estimated parameters output port

`off` (default) | `on`

Select this parameter to add the **theta_hat** output port.

Programmatic Use
Block Parameter: `estimatedVarOn`
Type: character vector
Values: `'off'` | `'on'`
Default: `'off'`

Extended Capabilities

C/C++ Code Generation

Generate C and C++ code using Simulink® Coder™.

- The Extremum Seeking Control block supports code generation for double-precision signals only.
- When running in Rapid Accelerator mode, the Extremum Seeking Control block does not support data logging.
- To generate hardware-deployable code for the Extremum Seeking Control block, use a discrete-time controller. To do so, set the **Time Domain** parameter to **Discrete time**.

See Also

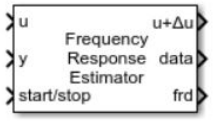
Topics

“Extremum Seeking Control” on page 15-2

Introduced in R2021a

Frequency Response Estimator

Estimate plant frequency responses during simulation or in real time
Library: Simulink Control Design



Description

Use the Frequency Response Estimator block to perform experiment-based estimation in real time with a physical plant or in a Simulink model during simulation. To obtain an estimated frequency response, the block simultaneously:

- Injects sinusoidal test signals into the plant at the nominal operating point
- Collects response data from the plant output
- Computes the estimated frequency response

You specify the frequencies at which to perturb the plant and measure system response. You trigger the estimation process via a start/stop signal. This signal lets you start estimation at any time, typically when the plant is at the nominal operating point. You stop the estimation after the frequency responses converge.

You can use online frequency response estimation with any stable SISO plant. For an unstable plant, online estimation works in a closed-loop configuration, provided that the closed loop is internally stable. A closed-loop system is internally stable if and only if the roots of the nominal closed-loop characteristic equation all lie in the open left half-plane. For a plant with transfer function $G = N_G/D_G$ and controller $C = N_C/D_C$, the characteristic equation is:

$$D_G D_C + N_G N_C = 0.$$

In practice, this condition means that no unstable poles in G are stabilized by pole-zero cancellation in GC. Do not use online estimation with an unstable plant that does not meet this condition.

You can generate code and deploy the Frequency Response Estimator block on hardware to perform the estimation in real time. The block supports code generation with Simulink Coder, Embedded Coder, and Simulink PLC Coder. It does not support code generation with HDL Coder.

For more information about using the Frequency Response Estimator block, see:

- “Online Estimation Using Plant Modeled in Simulink” on page 6-5
- “Deploy Frequency Response Estimation Algorithm for Real-Time Use” on page 6-9

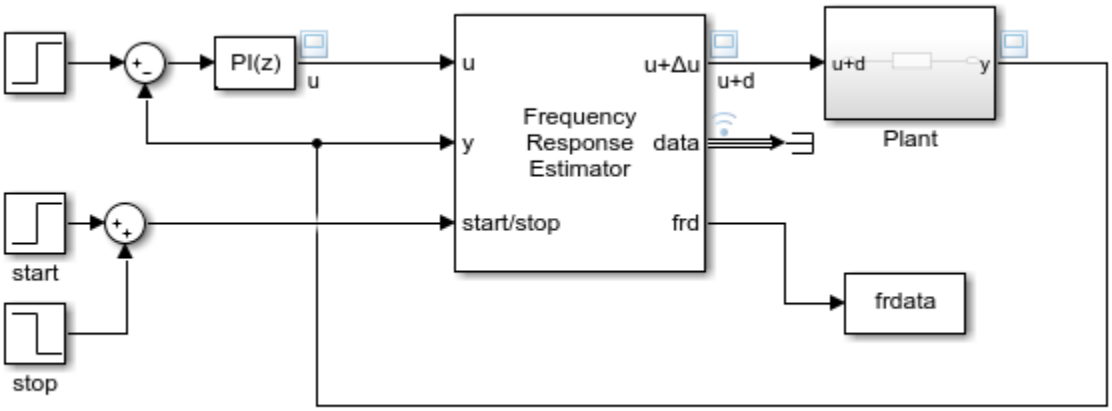
For more general information about online frequency response estimation, see “Online Frequency Response Estimation Basics” on page 6-2.

Ports

Input

u — Plant input before perturbation
scalar

Insert the block into your system such that this port accepts a control signal or other plant input signal. For instance, in a closed-loop configuration, you can connect this port as shown in the following diagram.



In an open-loop configuration, you can connect this input port to a source that drives your plant to the desired operating point for estimation. For instance, you can use a Constant block set to an appropriate value.

Data Types: `single` | `double`

y — Plant output
scalar

Connect this port to the plant output.

Data Types: `single` | `double`

`start/stop` — Start and stop the estimation experiment
scalar

To start and stop the estimation process, provide a signal at the `start/stop` port. When the value of the signal changes from:

- Negative or zero to positive, the experiment starts
- Positive to negative or zero, the experiment stops

Typically, you can use a signal that changes from 0 to 1 to start the experiment, and from 1 to 0 to stop it. When the experiment is not running, the block adds no perturbation at the **u + Δu** or **Δu** port. In this state, the block has no impact on plant behavior.

Start the experiment when the plant is at the desired equilibrium operating point. In a closed-loop configuration, use the controller to drive the plant to the operating point. In an open-loop configuration, you can use a source block connected to **u** to drive the plant to the operating point.

Let the experiment run long enough for the algorithm to collect sufficient data for a good estimate at all frequencies it probes. The block displays a recommended experiment length in the **Experiment Length** section of the block parameters. This value is based on the experiment mode and the frequencies you specify for the experiment.

- When **Experiment mode** is **Sinestream**, the recommended experiment length is:

  $$\sum_i \frac{2\pi}{\omega_i}\left(N_{set,i} + N_{estim,i}\right) + 2T_s N_{freq},$$

  where:

  - ω_i is the ith frequency specified in the **Frequencies** parameter (in rad/s).
  - N_{freq} is the number of frequencies in **Frequencies**.
  - $N_{set,i}$ is the corresponding value of the **Number of settling periods** parameter.
  - $N_{estim,i}$ is the corresponding value of the **Number of estimation periods** parameter.
  - T_s is the experiment sampling time, specified by the **Sample time (Ts)** parameter.
- When **Experiment mode** is **Superposition**, the recommended experiment length is six times the longest period. If your system does not require much time for the decay of transients or for the averaging away of noise, then you can use a shorter experiment length. For more information about how to determine experiment length in superposition mode, see “Experiment Length and Data-Collection Window in Superposition Mode” on page 19-75.

Avoid any load disturbance to the plant during the experiment. Load disturbance can distort the plant output and reduce the accuracy of the frequency-response estimation.

Data Types: `single` | `double`

`w` — Frequencies for estimation experiment

vector

Supply a value for the **Frequencies** parameter. See that parameter for information about how to choose frequencies.

When you supply frequencies via this port, specify the number of frequencies with the **Number of frequencies in the excitation signal** parameter.

Dependencies

To enable this port, in **Excitation Signal Source**, select **External ports**.

Data Types: `single` | `double`

`amp` — Perturbation amplitudes

scalar | vector

Supply a value for the **Amplitudes** parameter. See that parameter for details.

Dependencies

To enable this port, in **Excitation Signal Source**, select **External Ports**.

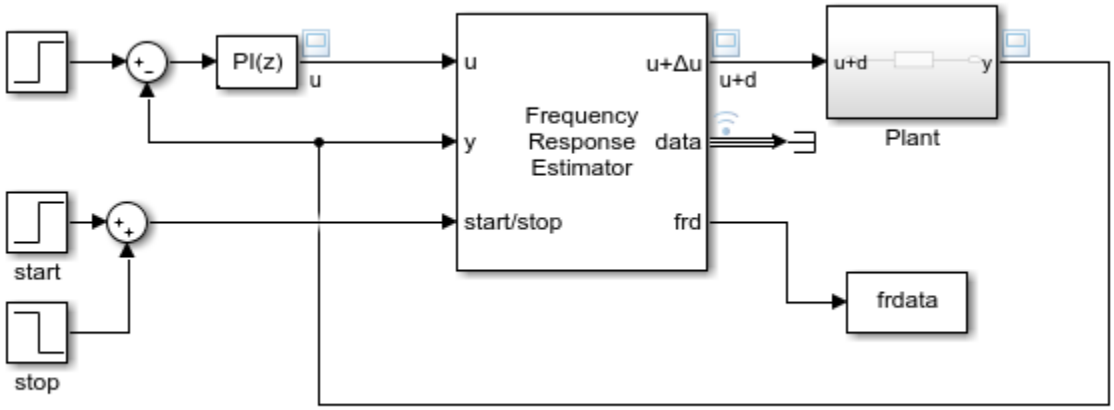
Data Types: `single` | `double`

Output

u + Δu — Perturbed plant input

scalar

Insert the block into your system such that this port feeds the input signal to your plant, such as in the following diagram.



- When the experiment is running (**start/stop** positive), the block injects test signals into the plant at this port. If you have any saturation or rate limit protecting the plant, feed the signal from **u + Δu** into it.
- When the experiment is not running (**start/stop** zero or negative), the block passes signals unchanged from **u** to **u + Δu**. In this state, the block has no effect on the plant.

Dependencies

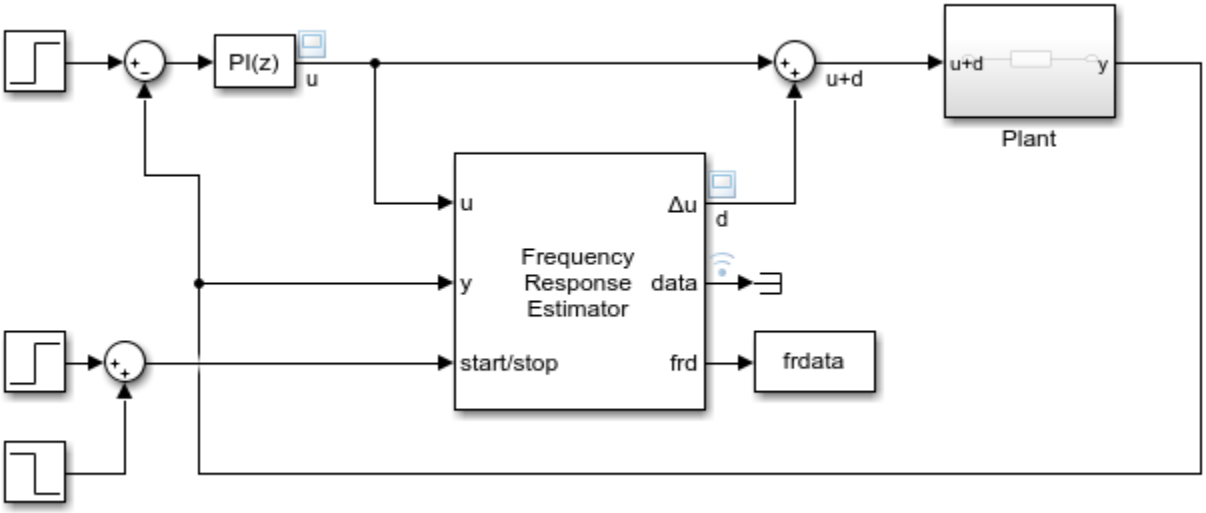
To enable this port, in **Output Signal Configuration**, select **control action + perturbation**.

Data Types: `single` | `double`

Δu — Plant input perturbation

scalar

The block generates a perturbation signal at this port. Typically, you inject the perturbation from this port via a sum block, as shown in the following diagram.



- When the experiment is running (**start/stop** positive), the block generates perturbation signals at this port.
- When the experiment is not running (**start/stop** zero or negative), the signal at this port is zero. In this state, the block has no effect on the plant.

Dependencies

To enable this port, in **Output Signal Configuration**, select **perturbation only**.

Data Types: `single` | `double`

`data` — Experiment data

bus

The signal at this port contains the data that the block collects during the frequency-response estimation experiment, including the perturbation signal applied to the plant and the measured plant response. Use this port when you want to log experiment data for later use. For instance, you can conserve resources in a deployed environment by logging the data and performing the estimation offline (see **Estimation Mode**). There are two ways to access the frequency response experiment data.

- Use a To Workspace block to write the data to the MATLAB workspace as a structure containing timeseries data. The **Save format** parameter of the To Workspace block must be `Timeseries`. The structure has the following fields:
  - `Ready` — Logical signal indicating which time steps are included in the estimation computation (1) and which are excluded (0). For instance, for sinestream mode, this signal is 1 only for data that falls within the periods determined by the **Number of settling periods** and **Number of estimation periods** parameters. In superposition mode, the signal is 1 only for data that falls within the window described in “Experiment Length and Data-Collection Window in Superposition Mode” on page 19-75.
  - `Perturbation` — Sinusoidal perturbations `Δu` applied to the plant

  - `PlantInput` — Plant input signal `u + Δu`, where `u` is the signal collected at the block input port **y**
  - `PlantOutput` — Plant output signal collected at block input port **y**
- Use Simulink data logging to write the data to the workspace as a `Simulink.SimulationData.Dataset` object. In this case, the structure containing the four timeseries signals is stored in the `Values` field of the resulting dataset. For instance, suppose that the model is configured to save the logged data to a variable `logsout`, and **data** is the only logged port. In that case, the structure is contained in `logsout{1}.Values`.

You can use the data from this port to perform the frequency-response estimation offline. For instance, you can compute the estimated frequency response in MATLAB by passing the structure to the `frestimate` command. For more information about accessing and using the experiment data, see “Collect Frequency Response Experiment Data for Offline Estimation” on page 6-18.

Data Types: `single` | `double`

`frd` — Estimated frequency responses

vector

The signal at this port contains the estimated frequency responses of the plant, in a vector with one entry for each frequency specified in the **Frequencies** parameter. You can write this signal to the MATLAB workspace using a To Workspace block, or use Simulink data logging to write the data to the workspace as a `Simulink.SimulationData.Dataset` object.

Typically, the best estimation is achieved at the end of the experiment. For that reason, you might not need to log all historical data at this port. Instead, you can discard the values for every time step except the last. For instance, in a To Workspace block, you can set the **Limit data points to last** parameter to 1. Then, when the experiment ends, the resulting workspace variable contains a vector of complex values, one for each frequency specified in the **Frequencies** parameter.

Dependencies

To enable this port, set **Estimation Mode** to **Online**.

Data Types: `single` | `double`

Parameters

`Sample time (Ts)` — Experiment sample time

0.1 (default) | positive scalar | –1

The block is a discrete-time block that runs at a fixed sample time, specified with this parameter. The largest frequency that you can estimate is the Nyquist frequency, π/T_s rad/s. Best practice is to use a sampling time at least five times faster than the Nyquist frequency,

$$T_s = \pi/(5\omega_{max}) \cong 0.6/\omega_{max} \text{ or } 0.1/f_{max},$$

where , where ω_{max} is the highest frequency in **Frequencies** in rad/s, and f_{max} is the highest frequency in Hz. The sample time must be small enough to estimate the fastest desired frequency, but not so small as to introduce unnecessary computational burden.

If you set the sample time to –1, then the software determines the sample time on compilation, based on the sources outside the block. Setting sample time to –1 disables the internal checks in the block that ensure your estimation frequencies are below the Nyquist frequency.

Tip

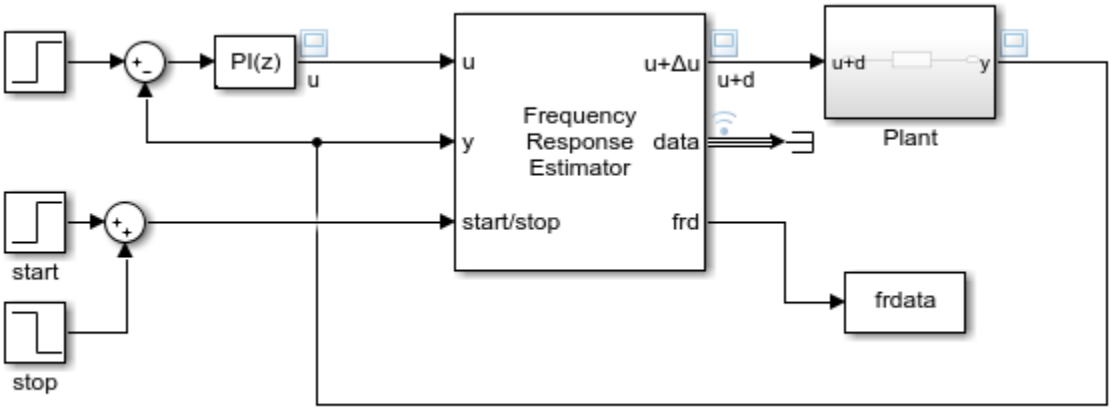
If you want to run the deployed block with different sample times in your application, set this parameter to –1 and put the block in a Triggered Subsystem. Then, trigger the subsystem at the desired sample time. If you do not plan to change the sample time after deployment, specify a fixed and finite sample time.

Programmatic Use
Block Parameter: `DiscreteTs`
Type: scalar
Value positive scalar | –1
Default: 0.1

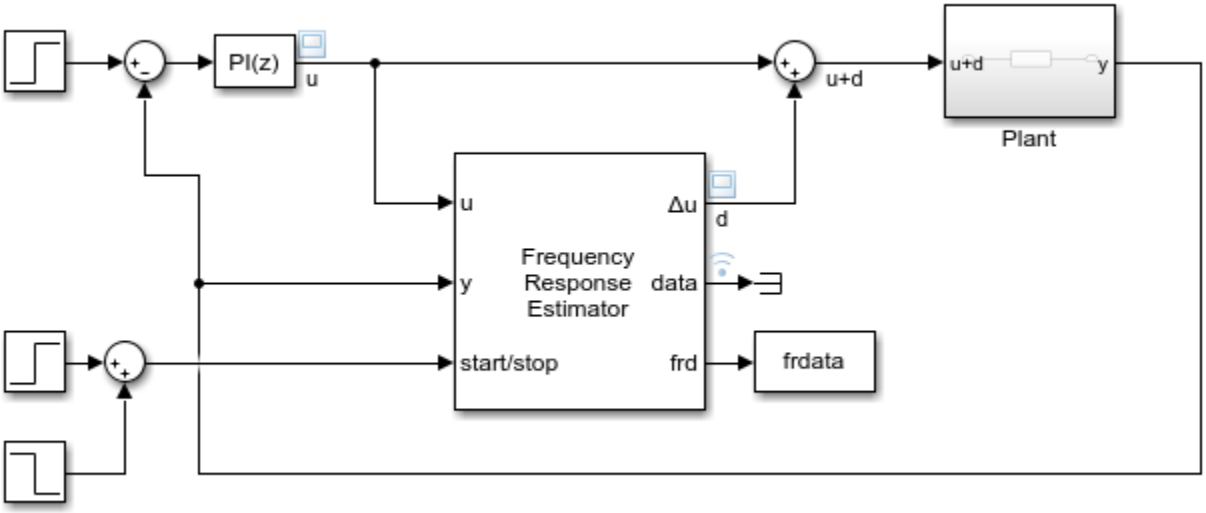
`Output Signal Configuration` — Provide control signal plus perturbation or perturbation only

control action + perturbation (default) | **perturbation only**

By default, the block takes a control signal as input and provides the control signal plus the experiment perturbation at the port **u+Δu**. You then feed this signal into the plant input, as shown in the following diagram.



This default configuration requires inserting the block between the controller and the plant. If you want to add the perturbation signal to the control signal yourself, select **perturbation only**. In this configuration, the block output contains the perturbation signal only, at the port **Δu**. You inject this perturbation signal into the plant using, for example, a sum block, as in the following diagram.



In this configuration, because the Frequency Response Estimator is not part of the closed loop, you can optionally comment it out without disrupting the loop configuration.

`Data Type` — Floating point precision
`double` (default) | `single`

Specify the floating-point precision based on simulation environment or hardware requirements.

Programmatic Use
Block Parameter: `BlockDataType`
Type: character vector
Values: `'double'` | `'single'`
Default: `'double'`

`Excitation signal source` — Excitation signal source
Block parameters (default) | **External ports**

Specify whether to supply the frequencies and amplitudes of the experiment perturbation signal via block parameters or via external ports.

- **Block parameters** — Select to enable the **Frequencies** and **Amplitudes** parameters.
- **External ports** — Select to enable the **w** and **amp** input ports. Use this option if you want to change the frequencies and amplitudes of the perturbation signal after deployment.

Programmatic Use
Block Parameter: `SineSource`
Type: character vector, string
Values: `'Block parameters'` | `'External ports'`
Default: `'Block parameters'`

`Frequencies` — Frequencies for estimation
[0.5 1 2] (default) | vector

Frequencies at which to estimate the frequency response of the plant. The block injects a perturbation at each of these frequencies, either simultaneously (when **Experiment mode** is

Superposition) or sequentially (**Experiment mode** is **Sinestream**). The highest frequency you can estimate is limited by the Nyquist frequency, π/T_s rad/s, where T_s is the value you set for the **Sample time (Ts)** parameter.

When **Experiment mode** is **Superposition**:

- To maintain reasonable convergence speed and estimation accuracy, it is typical to use about 20–30 frequencies for estimation. The best practice is to specify no more than about 50 frequencies.
- The best practice is to limit the range between the lowest and highest frequency to no more than about two decades. This limit reduces the chance that the responses of some frequencies are so dominant that they hurt the estimation of responses at other frequencies.
- Attempting to linearize a model containing a Frequency Response Estimator block using superposition mode and more than 50 frequencies can generate an error. The error states "The model contains too many elements for linearization. Please reduce the model size." To complete linearization, you must either comment out the frequency-response estimator block or reduce the number of frequencies.

When **Experiment mode** is **Sinestream**, there is no recommended limit on the number or range of frequencies. However, due to the sequential nature of the sinestream perturbation, each frequency point you add increases the required experiment time (see the **start/stop** input port for details). Further, a too-wide range of frequencies requires you to use a fast sample time for high frequencies that is inefficient for the lower frequencies.

In either mode, when you use the block in a closed-loop configuration, frequencies much higher than the open-loop bandwidth might result in less accurate estimation.

Tips

This parameter is not tunable. To provide frequencies after deployment, set **Excitation Signal Source** to **External ports** and use the **w** input port. For more information, see “Deploy Frequency Response Estimation Algorithm for Real-Time Use” on page 6-9.

Dependencies

To enable this parameter, set **Excitation Signal Source** to **Block parameters**.

Programmatic Use

Block Parameter: `Frequencies`
Type: vector
Values: positive real values
Default: `'[0.5 1 2]'`

`Frequency units` — Frequency units

`rad/s` (default) | `Hz`

Indicate whether the values of the **Frequencies** parameter are in radians per second or Hertz.

To enable this parameter, set **Excitation Signal Source** to **Block parameters**.

Programmatic Use

Block Parameter: `FreqUnits`
Type: string, character vector
Values: `'rad/s','Hz'`
Default: `'rad/s'`

`Amplitudes` — Amplitudes of injected perturbations

1 (default) | scalar | vector

Specify the amplitudes of the perturbation signals injected into the plant. To use the same amplitude for all frequencies, specify a scalar value. If you know that the response changes significantly over range of frequencies to estimate, then you can use a vector to specify a different amplitude for each frequency. For instance, you can use a smaller value around known resonant frequencies and a larger value above the rolloff frequency. The vector must be the same length as the vector you provide for **Frequencies**.

The amplitudes must be:

- Large enough that the perturbation overcomes any deadband in the plant actuator and generates a response above the noise level
- Small enough to keep the plant running within the approximately linear region near the nominal operating point, and to avoid saturating the plant input or output

When **Experiment mode** is **Superposition**, the sinusoidal signals are superimposed with no phase shift. Thus, the maximum perturbation can exceed the amplitude of any individual component, up to the sum of all amplitudes. Make sure that the largest possible perturbation is within the range of your plant actuator. Saturating the actuator can introduce errors into the estimated frequency response.

Tip

This parameter is not tunable. To provide amplitudes after deployment, set **Excitation Signal Source** to **External ports** and use the **amp** input port. For more information, see “Deploy Frequency Response Estimation Algorithm for Real-Time Use” on page 6-9.

Dependencies

To enable this parameter, set **Excitation Signal Source** to **Block parameters**.

Programmatic Use

Block Parameter: `Amplitudes`
Type: scalar, vector
Default: `'1'`

`Number of frequencies in the excitation signal` — Number of externally supplied frequencies

3 (default) | positive integer

When you provide the experiment frequencies via the external port **w**, specify the number of frequencies (the length of the vector signal at **w**) with this parameter.

Dependencies

To enable this parameter, set **Excitation Signal Source** to **External ports**.

Programmatic Use

Block Parameter: `NumOfFreq`
Type: scalar
Default: `'3'`

`Experiment mode` — Experiment mode

Sinestream (default) | **Superposition**

Specify whether the perturbation at each frequency is applied sequentially (**Sinestream**) or simultaneously (**Superposition**).

- **Sinestream** — In this mode, a perturbation is applied at each frequency separately. You specify how many periods at each frequency to allow the system to settle using the **Number of settling periods** parameter. Specify how many periods to measure the response using the **Number of estimation periods** parameter. For more information about sinestream signals for estimation, see “Sinestream Input Signals” on page 5-29.
- **Superposition** — In this mode, the perturbation signal includes all specified frequencies at once. For frequency response estimation at a vector of frequencies $\omega = [\omega_1, \ldots, \omega_N]$ at amplitudes $A = [A_1, \ldots, A_N]$, the perturbation signal is:

  $$\Delta u = \sum_i A_i \sin(\omega_i t) .$$

  Best practice is to use no more than about 50 frequencies in a superposition signal.

Sinestream mode can be more accurate and can accommodate a wider range of frequencies than **Superposition** mode (see the **Frequencies** parameter). **Sinestream** mode can also be less intrusive, because the total size of the perturbation is never bigger than the values specified by the **Amplitudes** parameter. However, due to the sequential nature of the sinestream perturbation, each frequency point you add increases the recommended experiment time (see the **start/stop** input port for details). Thus, the estimation experiment is typically much faster in **Superposition** mode with satisfactory results.

Tip

Attempting to linearize a model containing a Frequency Response Estimator block using superposition mode and more than 50 frequencies can generate an error. The error states "The model contains too many elements for linearization. Please reduce the model size." To complete linearization, you must either comment out the frequency-response estimator block or reduce the number of frequencies.

Programmatic Use

Block Parameter: `ExperimentMode`
Type: character vector, string
Values: `'Sinestream'` | `'Superposition'`
Default: `'Sinestream'`

`Number of settling periods` — Number of periods to wait for settling of transients

2 (default) | positive integer | vector of positive integers

In the sinestream experiment mode, the block injects separate perturbations at each frequency you specify in **Frequencies**. Use **Number of settling periods** to specify how long to wait at each frequency before beginning estimation at that frequency. Waiting allows any transients in the plant response to decay away, improving the accuracy of the estimated frequency response. Waiting for more periods can improve the accuracy of the estimation, but also increases the experiment time.

To use the same number of settling periods for all frequencies, specify a positive scalar value. If you know that the transients settle at different rates over range of frequencies to estimate, then you can use a vector to specify a different number of settling periods for each frequency.

For more information about sinestream signals for estimation, see “Sinestream Input Signals” on page 5-29.

Tunable: Yes

Dependencies

To enable this parameter, in **Experiment Mode**, select **Sinestream**.

Programmatic Use

Block Parameter: `NumOfSetPeriod`
Type: integer, vector of integers
Default: `'2'`

`Number of estimation periods` — Number of periods after settling to use for estimation

4 (default) | integer ≥ 2 | vector of integers

In the sinestream experiment mode, the block injects separate perturbations at each frequency you specify in **Frequencies**. Use **Number of estimation periods** to specify how many periods of injected signal to use for the estimation at each frequency. Using more periods can improve the accuracy of the estimation, but also increases the experiment time.

To use the same number of estimation periods for all frequencies, specify a scalar value greater than or equal to 2. You can use a vector to specify a different number of settling periods for each frequency. This approach is useful when you know that your system is less noisy at some frequencies, or you are less concerned about accuracy at some frequencies.

For more information about sinestream signals for estimation, see “Sinestream Input Signals” on page 5-29.

Tunable: Yes

Dependencies

To enable this parameter, in **Experiment Mode**, select **Sinestream**.

Programmatic Use

Block Parameter: `NumOfEstPeriod`
Type: integer, vector of integers
Default: `'4'`

`Number of periods of the lowest frequency used for estimation` — Duration of data-collection window

3 (default) | integer between 1 and 5

In the superposition experiment mode, the block applies perturbations at all frequencies simultaneously while the experiment is running. The block uses this parameter to determine how long a data-collection window to use for estimation. For more information about the data-collection window, see “Experiment Length and Data-Collection Window in Superposition Mode” on page 19-75.

Dependencies

To enable this parameter, in **Experiment Mode**, select **Superposition**.

Programmatic Use

Block Parameter: `NumOfSlowestPeriod`
Type: integer
Default: `'3'`

`Estimation mode` — Whether block performs estimation or only collects response data

Online (default) | **Offline**

Specify whether to perform the frequency response estimation computation online or to collect frequency-response data only, for later offline estimation.

- **Online** — The block collects experiment data and computes the estimated frequency response while the experiment is running. You can get the resulting estimated frequency response at the **frd** port (see that port description for more information).
- **Offline** — The block collects experiment data only and does not compute the estimated frequency response. You can get the experiment data at the **data** port (see that port description for more information). You can then perform the frequency-response estimation offline. For instance, you can use the data in MATLAB to compute the estimated frequency response with the `frestimate` command. For more information, see “Collect Frequency Response Experiment Data for Offline Estimation” on page 6-18.

Programmatic Use

Block Parameter: `EstimationMode`
Type: character vector, string
Values: `'Online'` | `'Offline'`
Default: `'Online'`

`Display Bode plot` — Plot estimated frequency response

`off` (default) | `on`

Select to generate a Bode plot showing the estimated frequency response. The plot updates periodically during the estimation experiment. If you have an LTI model representing the expected plant response or other relevant baseline, include it on the plot for reference using the **Baseline plant model** parameter.

Tips

- To speed up trimming or linearization of a model containing a Frequency Response Estimator block, clear this parameter.

Programmatic Use

Block Parameter: `UseBodePlot`
Type: character vector, string
Values: `'off'` | `'on'`
Default: `'off'`

`Baseline plant model` — Baseline model for Bode plot

`[]` (default) | LTI model

Specify the baseline model to plot with the estimated frequency response. Use an LTI model such as a `tf`, `ss`, or `frd` model.

Example: `tf(10,[1 10 1000])`

Dependencies

To enable this parameter, select **Display Bode plot**.

Programmatic Use

Block Parameter: `BaselinePlant`
Type: LTI model

Default: `'[]'`

`Refresh plot every N*Ts seconds where N is` — How often to update Bode plot
100 (default) | scalar

During the frequency response estimation experiment, the block updates the Bode plot with the estimated frequency responses as often as you specify with this parameter. Increase the value if refreshing the Bode plot takes too much time.

Dependencies

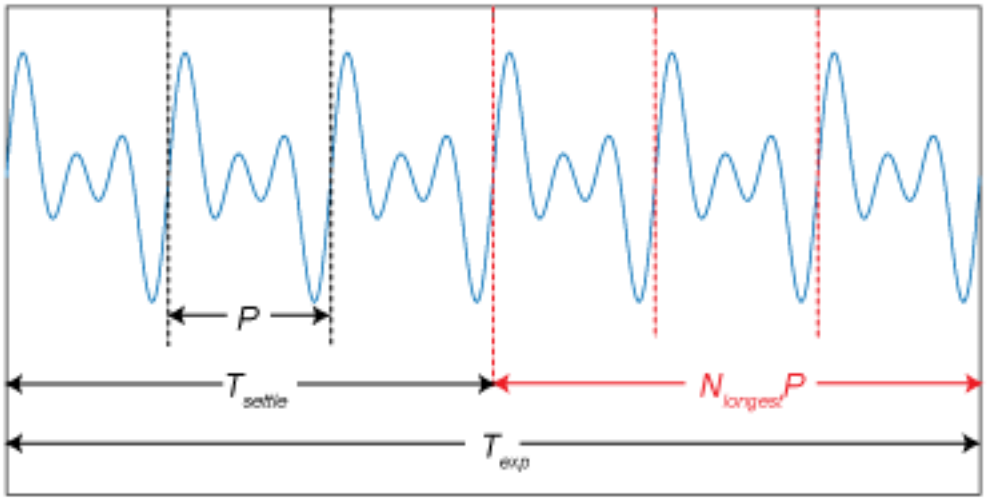
To enable this parameter, select **Display Bode plot**.

Programmatic Use
Block Parameter: `PlotRefreshFactor`
Type: integer
Default: `'100'`

More About

Experiment Length and Data-Collection Window in Superposition Mode

The block supplies the perturbation Δu for the duration of the experiment (while the **start/stop** signal is positive). The block determines how long to wait for system transients to die away and how many cycles to use for estimation as shown in the following illustration.



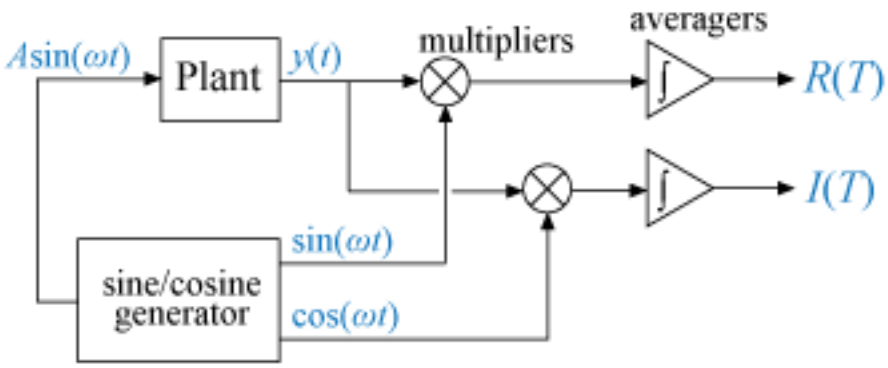
T_{exp} is the experiment duration that you specify with your configuration of the start/stop signal (See the **start/stop** port description on the block reference page for more information). For the estimation computation, the block uses only the data collected in a window of $N_{longest}P$. Here, P is the period of the slowest frequency in the frequency vector ω, and $N_{longest}$ is the value of the **Number of periods of the lowest frequency used for estimation** block parameter. Any cycles before this window are discarded. Thus, the settling time $T_{settle} = T_{exp} - N_{longest}P$. If you know that your system settles quickly, you can shorten T_{exp} without changing $N_{longest}$ to effectively shorten T_{settle}. If your system is noisy, you can increase $N_{longest}$ to get more averaging in the data-collection window. Either way, always choose

T_{exp} long enough for sufficient settling and sufficient data-collection. The recommended T_{exp} = $2N_{longest}P$.

Algorithms

Sinestream Mode

When **Experiment mode** is **Sinestream**, the block uses a correlation analysis method. In this method, the measured plant output $y(t)$ is mixed with a sine signal and a cosine signal at the test frequency ω. The resulting signal is then integrated and averaged for a time $T = N(2\pi/\omega)$, where N is the integer value of the **Number of estimation periods** parameter. These operations are shown in the following diagram.



As the averaging time T increases, the contribution of components in $y(t)$ at frequencies other than ω go to zero. $R(T)$ and $I(T)$ become constant and can be used to calculate the frequency response of the plant at ω. For further details, see [1].

Superposition Mode

When **Experiment mode** is **Superposition**, the block uses a recursive least squares (RLS) algorithm to compute the estimated frequency response. Assume that the plant frequency response is $G(j\omega) = \gamma\angle j\theta$. When a signal $u(t) = A\sin(\omega t)$ excites the plant, the steady-state plant output is $y(t) = A\gamma\sin(\omega t + \theta)$, which is equivalent to:

$$y(t) = (\gamma\cos\theta)A\sin(\omega t) + (\gamma\sin\theta)A\cos(\omega t)\,.$$

At any given time, $A\sin(\omega t)$ and $A\cos(\omega t)$ are known. Therefore, they can be used as regressors in an RLS algorithm to estimate $\gamma\cos(\theta)$ and $\gamma\sin(\theta)$ from the measured plant output $y(t)$ at run time.

When the excitation signal contains a superposition of multiple signals, then:

$$u(t) = A_1\sin(\omega_1 t) + A_2\sin(\omega_2 t) + \ldots\,.$$

In this case, the plant output becomes:

$$\begin{aligned} y(t) = {} & (\gamma_1\cos\theta_1)A_1\sin(\omega_1 t) + (\gamma_1\sin\theta_1)A_1\cos(\omega_1 t) + \\ & (\gamma_2\cos\theta_2)A_2\sin(\omega_2 t) + (\gamma_2\sin\theta_2)A_2\cos(\omega_2 t) + \ldots\,. \end{aligned}$$

The estimation algorithm uses $A_i\sin(\omega_i t)$ and $A_i\cos(\omega_i t)$ as regressors to estimate $\gamma_i\cos(\theta_i)$ and $\gamma_i\sin(\theta_i)$. For N frequencies, the algorithm uses $2N$ regressors.

The computation assumes that the perturbation signal $u(t)$ is applied to a plant with zero nominal input and output. To achieve this condition, the block subtracts from the measured plant input and output signals their values measured at the start of the experiment.

References

[1] Wellstead, P. E. *Technical Report 10: Frequency Response Analysis*. Farnborough, Hampshire, UK: Solartron Instruments, 1997.

Extended Capabilities

C/C++ Code Generation

Generate C and C++ code using Simulink® Coder™.

PLC Code Generation

Generate Structured Text code using Simulink® PLC Coder™.

See Also

Closed-Loop PID Autotuner

Topics

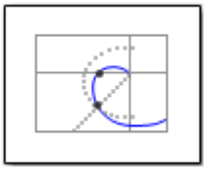
“Online Frequency Response Estimation Basics” on page 6-2
“Deploy Frequency Response Estimation Algorithm for Real-Time Use” on page 6-9
“Online Estimation Using Plant Modeled in Simulink” on page 6-5

Introduced in R2019a

Gain and Phase Margin Plot, Check Gain and Phase Margins

Gain and phase margins of linear system approximated from nonlinear Simulink model

Library: Simulink Control Design / Linear Analysis Plots
Simulink Control Design / Model Verification



Description

The Gain and Phase Margin Plot and Check Gain and Phase Margins blocks compute a linear system from a nonlinear Simulink model and display the gain and phase margins during simulation. These blocks are identical except for the default settings on the **Bounds** tab.

- The Gain and Phase Margin Plot does not define default bounds.
- The Check Gain and Phase Margins block defines default bounds and enables these bounds for assertion.

You can view the margins in a table or on a Bode, Nichols, or Nyquist plot.

For more information on frequency domain analysis of linear systems, see “Frequency-Domain Responses”.

During simulation, the software linearizes the portion of the model between specified linearization inputs and outputs and then plots the response of the linear system. You also can save the linear system as a variable in the MATLAB workspace.

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 one minimum bound each for the gain and phase margin and view them on the selected plot or table. 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 in the MATLAB Command Window. 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 any bound is not satisfied, the signal is false (`0`).

To compute and plot the gain and phase margins of various portions of your model, you can add multiple Gain and Phase Margin Plot and Check Gain and Phase Margins blocks.

These blocks do not support code generation and can be used only in `Normal` simulation mode.

Ports

Input

`Trigger` — External trigger signal

scalar

Use this input port (indicated by) to connect an external trigger signal for computing the model linearization. To specify the type of trigger signal to detect, use the **Trigger type** parameter.

Dependencies

To enable this port, set the **Linearize on** parameter to `External trigger`.

Output

z^{-1} — Assertion signal

`1` | `0`

Output the value of the assertion signal as a logical value. If any bound specified on the **Bounds** tab is violated, the assertion signal is false (`0`). Otherwise, this signal is true (`1`).

By default, the data type of the output signal is double. To set the output data type as Boolean, in the Simulink model, in the Configuration Parameters dialog box, select the **Implement logic signals as Boolean data** parameter. This setting applies to all blocks in the model that generate logic signals.

You can use the assertion signal to design complex assertion logic. For an example, see “Verify Model Using Simulink Control Design and Simulink Verification Blocks” on page 17-20.

Dependencies

To enable this port, select the **Output assertion signal** parameter.

Parameters

`Plot type` — Select plot type

`Bode` (default) | `Nichols` | `Nyquist` | `Tabular`

Select one of the following methods for displaying the computed gain and phase margins.

- `Bode` — Bode plot
- `Nichols` — Nichols plot
- `Nyquist` — Nyquist plot
- `Tabular` — Table

For more information on using the plot, see “Using the Plot” on page 19-93.

Programmatic Use
Block Parameter: `PlotType`
Type: character vector
Value: `'bode'` | `'nichols'` | `'nyquist'` | `'table'`
Default: `'bode'`

`Show Plot` — Open plot
button

To view gain and phase margins computed during a simulation, click this button before starting the simulation. If you specify bounds on the **Bounds** tab, they are also shown on the plot.

To show the plot when opening the block, select the **Show plot on block open** parameter.

For more information on using the plot, see “Using the Plot” on page 19-93.

`Show plot on block open` — Open plot when opening block
`off` (default) | `on`

Select this parameter to open the plot when opening the block. You can then perform tasks, such as adding or modifying bounds, in the plot window instead of using the block parameters. To access the block parameters from the plot window, select **Edit** or click .

For more information on using the plot, see “Using the Plot” on page 19-93.

Programmatic Use
Block Parameter: `LaunchViewOnOpen`
Type: character vector
Value: `'off'` | `'on'`
Default: `'off'`

`Response Optimization` — Open Response Optimizer
button

Open the **Response Optimizer** app to optimize the model response to meet the design requirements specified on the **Bounds** tab.

This button is available only if you have Simulink Design Optimization software installed.

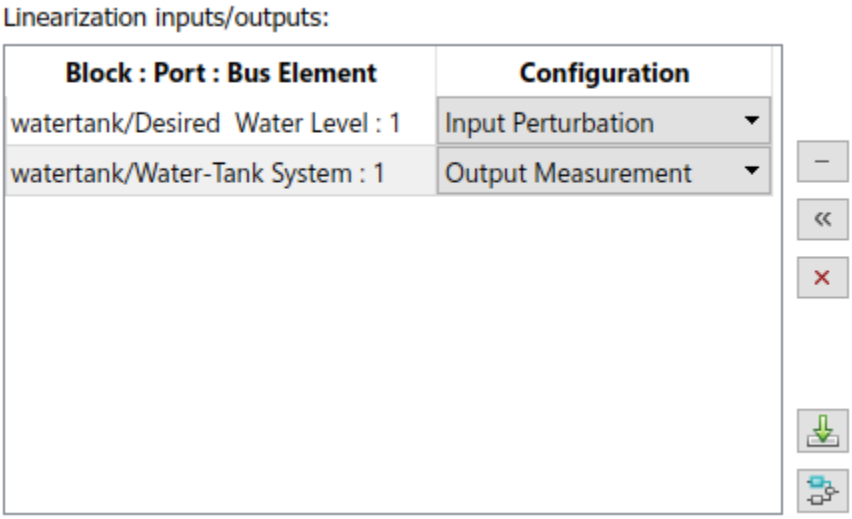
For more information on response optimization, see “Design Optimization to Meet Step Response Requirements (GUI)” (Simulink Design Optimization) and “Design Optimization to Meet Time-Domain and Frequency-Domain Requirements (GUI)” (Simulink Design Optimization).

Linearizations

To specify the portion of the model to linearize and other linearization settings, use the parameters on the **Linearizations** tab. The default settings on this tab are the same for the Gaind and Phase Margin Plot and Check Gain and Phase Margins blocks.

`Linearization inputs/outputs` — Specify portion of model to linearize
linear analysis points

To specify the portion of the model to linearize, select signals from the Simulink model and add them as linearization inputs or outputs.



In the table, the **Block:Port:Bus Element** column shows the following information for each signal.

- Source block
- Output port of the source block to which the signal is connected
- Bus element name (if the signal is in a bus)

In the **Configuration** column, select the type of linear analysis point from the following types. For more information on linear analysis points, see “Specify Portion of Model to Linearize” on page 2-10.

- `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 a 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 a linearization input or output, the software generates a warning in the MATLAB Command Window and does not compute a linear system.

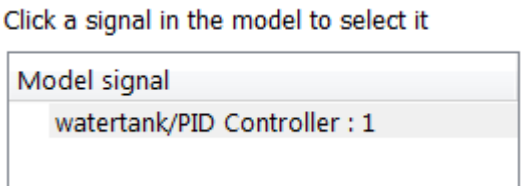
Edit Linearization Inputs and Outputs

To add linearization inputs and outputs:

1. To expand the signal selection area, click +.

   The dialog box expands to display a **Click a signal in the model to select it** area.
2. In the Simulink model, select one or more signals.

   The selected signals appear in the **Model signal** table.



3 (Optional) For bus signals, expand the bus to select individual elements.

Tip For large buses or other large lists of signals, you can filter the signal names. In the **Filter by name** box, enter search text. The name match is case-sensitive.

To modify the filtering options, click . For more information on filtering options, see the **Enable regular expression** and **Show filtered results as a flat list** parameters.

4 To add the selected signal to the **Linearization inputs/outputs** table, click .

5 In the **Configuration** column, specify the signal type.

Alternatively, if you have linearization inputs and outputs defined in your model, you can add them to the **Linearization inputs/outputs** table by clicking .

To remove a signal from the **Linearization inputs/outputs** table, select the signal and click .

To highlight the source block of a signal in the Simulink model, select the signal in the **Linearization inputs/outputs** table and click .

`Enable regular expression` — Enable signal searching using regular expressions
`on` (default) | `off`

Select this option to enable the use of MATLAB regular expressions for filtering signal names. For example, entering `t$` in the **Filter by name** text box displays all signals whose names end with a lowercase `t` (and their immediate parents). For more information, see “Regular Expressions”.

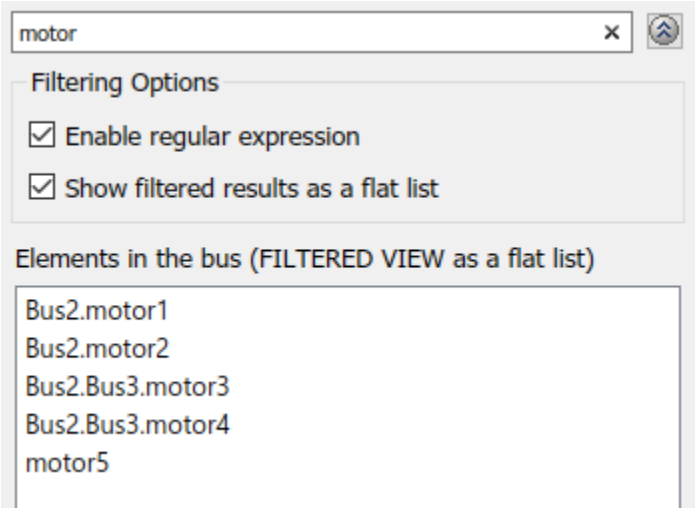
Dependencies

To enable this parameter, click next to the **Filter by name** text box.

`Show filtered results as a flat list` — Display filtered bus signal hierarchy using flat list
`off` (default) | `on`

Select this option to display the list of filtered signals in a flat list format. The flat list format uses dot notation to reflect the hierarchy of bus signals. The signals are filtered based on the text in the **Filter by name** text box.

The following figure shows an example of the flat list format for a filtered set of nested bus signals.



Dependencies

To enable this parameter, click next to the **Filter by name** text box.

`Linearize on` — When to compute linear model

`Simulation snapshots` (default) | `External trigger`

Use this parameter to specify when you want to compute a linear model.

To compute linear models at specified simulation snapshot times, set this parameter to `Simulation snapshots`. Specify snapshot times using the **Snapshot times** parameter.

Use simulation snapshots when you:

- Know one or more times when the model is at a steady-state operating point
- Want to compute linear systems at specific times

To compute linear models at trigger-based simulation events, set this parameter to `External trigger`. Selecting this option adds a trigger input port to the block, to which you connect your external trigger signal. To specify the type of trigger to detect, use the **Trigger type** parameter.

Use an external trigger when a signal generated during simulation indicates that the model is at a steady-state condition of interest. For example, for an aircraft model, you might want to compute the linear system whenever the fuel mass is a given fraction of the maximum fuel mass.

Programmatic Use
Block Parameter: `LinearizeAt`
Type: character vector
Value: `'SnapshotTimes'` | `'ExternalTrigger'`
Default: `'SnapshotTimes'`

`Snapshot times` — Simulation times at which to compute linear model

`0` (default) | positive real value | vector of positive real values

To compute a linear system at specific simulation times, such as a time that you know the model reaches a steady state operating point, specify one or more snapshot times. To specify multiple snapshot times, specify this parameter as a vector of positive values.

Snapshot times must be less than or equal to the simulation time specified in the Simulink model.

For examples of linearizing a model at simulation snapshot times, see:

- “Visualize Linear System at Multiple Simulation Snapshots” on page 2-85
- “Verify Model at Default Simulation Snapshot Time” on page 17-5
- “Verify Model at Multiple Simulation Snapshots” on page 17-13

Dependencies

To enable this parameter, set the **Linearize on** parameter to `Simulation snapshots`

Programmatic Use

Block Parameter: `SnapshotTimes`
Type: character vector
Value: `'0'` | positive real value | vector of positive real values
Default: `'0'`

`Trigger type` — Type of external trigger to detect

`Rising edge` (default) | `Falling edge`

Specify the trigger to detect in the external trigger signal as one of the following types.

- `Rising edge` — Use the rising edge of the trigger signal; that is, when the signal changes from `0` to `1`.
- `Falling edge` — Use the falling edge of the trigger signal; that is, when the signal changes from `1` to `0`.

Dependencies

To enable this parameter, set the **Linearize on** parameter to `External trigger`.

Programmatic Use

Block Parameter: `TriggerType`
Type: character vector
Value: `'rising'` | `'falling'`
Default: `'rising'`

`Enable zero-crossing detection` — Enable zero-crossing detection

`on` (default) | `off`

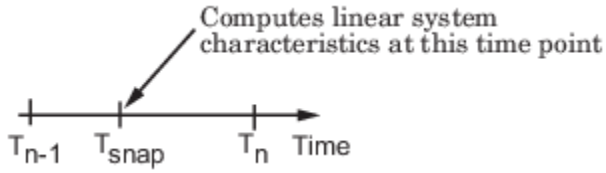
Select this option to enable zero-crossing detection.

When you set the **Linearize on** parameter to `Simulation snapshots`, enabling zero-crossing detection ensures that the software computes the linear model at the exact snapshot times you specify in the **Snapshot times** parameter.

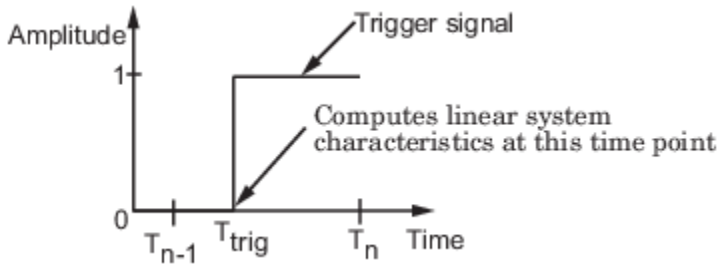
When you set the **Linearize on** parameter to `External trigger`, enabling zero-crossing detection ensures that the software computes the linear model at the exact time that the external trigger is detected. To specify the type of trigger, use the **Trigger type** parameter.

If you clear this option, the software computes the linear system at simulation times selected by the variable-step Simulink solver, which might not correspond to an exact snapshot time or the exact time when a trigger signal is detected.

For example, consider the case where the variable-step solver selects simulation times T_{n-1} and T_n. As shown in the following figure, the specified snapshot time T_{snap} can be between the selected simulation times. If you enable zero-crossing detection, the solver also simulates the model at time T_{snap} and computes the linear model at this point.



Similarly, the external trigger can be detected at a time T_{trig} that is between the selected simulation times. If you enable zero-crossing detection, the solver also simulates the model at time T_{trig} and computes the linear model at this point.



In both cases, if you do not enable zero-crossing detection, the software computes the linear model at either T_{n-1} or T_n.

For more information on zero-crossing detection, see “Zero-Crossing Detection”.

Dependencies

This parameter is ignored when you use a fixed-step Simulink solver.

Programmatic Use

Block Parameter: `ZeroCross`
Type: character vector
Value: `'on'` | `'off'`
Default: `'on'`

`Use exact delays` — Use exact delays in linear model

`off` (default) | `on`

Select this option to compute a linear model with exact delays. If you clear this option, the linear model uses Padé approximations of any delays.

For more information on linearizing models with delays, see “Linearize Models with Delays” on page 2-78.

Programmatic Use

Block Parameter: `UseExactDelayModel`
Type: character vector
Value: `'off'` | `'on'`
Default: `'off'`

`Linear system sample time` — Sample time of linear system
`'auto'` (default) | positive finite value | `0`

To compute a linear system with the specified sample time, the software coverts sample times in the model using the method you specify in the **Sample time rate conversion method** parameter.

You can set the sample time to one of the following values.

- `auto` — If all blocks in the model are continuous-time, use a sample time of `0`. Otherwise, set the sample time to the least common multiple of the nonzero sample times in the model.
- Positive finite value — Create a discrete-time model with the specified sample time
- `0` — Create a continuous-time model

Programmatic Use
Block Parameter: `SampleTime`
Type: character vector
Value: `'auto'` | positive finite value | `'0'`
Default: `'auto'`

`Sample time rate conversion method` — Rate conversion method
`Zero-Order Hold` (default) | `Tustin (bilinear)` | `Tustin with Prewarping` | ...

Method for converting sample times during linearization, specified as one of the following values.

- `Zero-Order Hold` — Zero-order hold, where the control inputs are assumed piecewise constant over the sample time `Ts`. This method usually performs better in the 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. This method usually performs better in the frequency domain.
- `Tustin with Prewarping` — Bilinear (Tustin) approximation with frequency prewarping. Specify the prewarp frequency using the **Prewarp frequency** parameter. This method usually performs better in the frequency domain. Use this method to ensure matching in a frequency region of interest.
- `Upsampling when possible, Zero-Order Hold otherwise` — Upsample a discrete-time system when possible; otherwise, use a zero-order hold.
- `Upsampling when possible, Tustin otherwise` — Upsample a discrete-time system when possible; otherwise, use a Tustin approximation.
- `Upsampling when possible, Tustin with Prewarping otherwise` — Upsample a discrete-time system when possible; otherwise, use a Tustin approximation with frequency prewarping.

You can upsample only when you convert a discrete-time system to a new faster sample time that is an integer multiple of the sample time of the original system.

For more information on rate conversion and linearization of multirate models, see:

- “Multirate Linearization Algorithm” on page 2-140
- “Linearize Models Using Different Rate Conversion Methods” on page 2-145
- “Continuous-Discrete Conversion Methods”

Note If you use a rate conversion method other than `Zero-Order Hold`, the converted states no longer have the same physical meaning as the original states. As a result, the state names in the resulting LTI system change to `'?'`.

Dependencies

To enable this parameter, set the **Linear system sample time** parameter to a value other than `auto`.

Programmatic Use

Block Parameter: `RateConversionMethod`
Type: character vector
Value: `'zoh'` | `'tustin'` | `'prewarp'` | `'upsampling_zoh'` | `'upsampling_tustin'` | `'upsampling_prewarp'`
Default: `'zoh'`

`Prewarp frequency` — Prewarp frequency for Tustin rate conversion

`'10'` (default) | positive scalar

Prewarp frequency for Tustin rate conversion in radians per second, specified as a scalar value less than the Nyquist frequency before and after resampling.

Dependencies

To enable this parameter, set the **Sample time rate conversion method** parameter to one of the following values.

- `Tustin with Prewarping`
- `Upsampling when possible, Tustin with Prewarping otherwise`

Programmatic Use

Block Parameter: `PreWarpFreq`
Type: character vector
Value: positive scalar
Default: `'10'`

`Use full block names` — Use full block path in state, input, and output names

`off` (default) | `on`

To show the state, input, and output names of the computed linear system using their full block path, select this parameter. For example, in the `scdcstr` model, a state in the `Integrator1` block of the `CSTR` subsystem appears with its full block path as `scdcstr/CSTR/Integrator1`.

If you clear this parameter, only the names of the states, inputs, and outputs are used, which is useful when the signal names are unique and you know their locations in your Simulink model. In the preceding example, the state name of the integrator block appears as `Integrator1`.

The computed linear system is a state-space object (`ss`). The state, input, and output names for the system appear in the following state-space object properties.

Input, Output, or State Name	State-Space Object Property
Linearization input names	`InputName`
Linearization output names	`OutputName`

Input, Output, or State Name	State-Space Object Property
State names	`StateName`

Programmatic Use

Block Parameter: `UseFullBlockNameLabels`
Type: character vector
Value: `'off'` | `'on'`
Default: `'off'`

`Use bus signal names` — Use bus signal names in linear system

`off` (default) | `on`

When you select an entire bus as a linearization input or output, select this parameter to use the signal names of the individual bus elements in the computed linear system. If you do not enable this option, the bus channel numbers are used instead.

Note Selecting an entire bus signal is not recommended. Instead, select individual bus elements.

Bus signal names appear when the linearization input or output is from one 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

Dependencies

Using this parameter is not supported when your model contains mux/bus mixtures.

Programmatic Use

Block Parameter: `UseBusSignalLabels`
Type: character vector
Value: `'off'` | `'on'`
Default: `'off'`

Bounds

To define gain and phase margin bounds whether to check for violations of these bounds, use the parameters on the **Bounds** tab. The default settings on this tab are different for the Gain and Phase Margin Plot and Check Gain and Phase Margins blocks.

`Include gain and phase margins in assertion` — Check whether gain and phase margins violate bounds

`on` | `off`

Select this parameter to check whether the gain and phase margins violate the specified bounds.

By default, this parameter is cleared for the Gain and Phase Margin Plot block and selected for the Check Gain and Phase Margins block.

Dependencies

This parameter is used for assertion only if you select the **Enable assertion** parameter.

Programmatic Use

Block Parameter: `EnableMargins`
Type: character vector
Value: `'on'` | `'off'`
Default: `'off'` for Gain and Phase Margin Plot block, `'on'` for Check Gain and Phase Margins block

`Gain margin` — Minimum gain margin bound

positive finite value

You can specify a single lower bound for the gain margin in decibels. To specify no gain margin bound, set this parameter to `[]`.

By default, the gain margin bound is `[]` for the Gain and Phase Margin Plot block and `20` for the Check Gain and Phase Margins block.

You can also edit the gain margin bound in the plot window. For more information, see “Using the Plot” on page 19-93.

Dependencies

To check whether the gain margin bound is violated during simulation, select both the **Include gain and phase margins in assertion** and **Enable assertion** parameters.

Programmatic Use

Block Parameter: `GainMargin`
Type: character vector
Value: positive finite number
Default: `'[]'` for Gain and Phase Margin Plot block, `'20'` for Check Bode Characteristics block

`Phase margin` — Minimum phase minimum bound

positive finite value

You can specify a single lower bound for the phase margin in degrees. To specify no phase margin bound, set this parameter to `[]`.

By default, the phase margin bound is `[]` for the Gain and Phase Margin Plot block and `30` for the Check Gain and Phase Margins block.

You can also edit the phase margin bound in the plot window. For more information, see “Using the Plot” on page 19-93.

Dependencies

To check whether the phase margin bound is violated during simulation, select both the **Include gain and phase margins in assertion** and **Enable assertion** parameters.

Programmatic Use

Block Parameter: `PhaseMargin`
Type: character vector
Value: positive finite number
Default: `'[]'` for Gain and Phase Margin Plot block, `'30'` for Check Bode Characteristics block

Feedback sign — Feedback sign for computing margins

`negative feedback` (default) | `positive feedback`

Specify the feedback sign for determining the gain and phase margins using one of the following options.

- `negative feedback` — Use negative feedback
- `positive feedback` — Use positive feedback

To determine the feedback sign, check if the path defined by the linearization inputs and outputs includes 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 "Verify Frequency-Domain Characteristics of an Aircraft" on page 17-27, the Check Gain and Phase Margins block includes the negative sign in the summation block. Therefore, the feedback sign is positive.

Programmatic Use

Block Parameter: `FeedbackSign`
Type: character vector
Value: `'-1'` | `'+1'`
Default: `'-1'`

Logging

To control whether linearization results computed during the simulation are saved, use the parameters on the **Logging** tab. The default settings on this tab are the same for the Gain and Phase Margin Plot and Check Gain and Phase Margins blocks.

Save data to workspace — Save linear systems for further analysis

`off` (default) | `on`

Select this parameter to save the computed linear systems for further analysis or control design. The data is saved in a structure with the following fields.

- `time` — Simulation times at which the linear systems are computed.
- `values` — State-space model representing the linear system. If the linear system is computed at multiple simulation times, `values` is an array of state-space models.
- `operatingPoints` — Operating points corresponding to each linear system in `values`. To enable this field, select the **Save operating points for each linearization** parameter.

To specify the name of the saved data structure, use the **Variable name** property.

The location of the saved data structure depends upon the configuration of the Simulink model.

- If the model is not configured to save simulation output as a single object, the data structure is a variable in the MATLAB workspace.
- If the 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 Configuration Parameters dialog box, select the **Single simulation output** parameter.

For more information about data logging in Simulink, see “Export Simulation Data” and the `Simulink.SimulationOutput` reference page.

Programmatic Use
Block Parameter: `SaveToWorkspace`
Type: character vector
Value: `'off'` | `'on'`
Default: `'off'`

`Variable name` — Name of data structure for saving linear systems

`sys` (default) | character vector

Specify the name of the data structure that stores linear systems computed during simulation.

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.

Dependencies

To enable this parameter, select the **Save data to workspace** parameter.

Programmatic Use
Block Parameter: `SaveName`
Type: character vector
Default: `'sys'`

`Save operating points for each linearization` — Save operating points with linearization

`off` (default) | `on`

Select this parameter to save the operating point at which each linearization is computed. Selecting this parameter adds the `operatingPoints` field to the saved data structure.

Dependencies

To enable this parameter, select the **Save data to workspace** parameter.

Programmatic Use
Block Parameter: `SaveOperatingPoints`
Type: character vector
Value: `'off'` | `'on'`
Default: `'off'`

Assertion

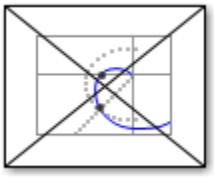
To control the assertion behavior of the block when bounds defined on the **Bounds** tab are violated, use the parameters on the **Assertion** tab. The default settings on this tab are the same for the Gain and Phase Margin Plot and Check Gain and Phase Margins blocks.

`Enable assertion` — Enable bound checking

`on` (default) | `off`

To check whether the bounds defined on the **Bounds** tab are satisfied during the simulation, select this parameter. When a bound is not satisfied, the assertion fails and a warning is generated.

Clearing this parameter disables assertion; that is, the block no longer checks that the specified bounds are satisfied. The block icon also updates to indicate that assertion is disabled.



By default, on the **Bounds** tab:

- The Gain and Phase Margin Plot block does not have defined bounds.
- The Check Gain and Phase Margins block has defined bounds.

You can configure your Simulink model to enable or disable all model verification blocks and override the **Enable assertion** parameter. To do so, in the Simulink model, in the Configuration Parameters dialog box, specify the **Model Verification block enabling** parameter.

Programmatic Use

Block Parameter: `enabled`
Type: character vector
Value: `'on'` | `'off'`
Default: `'on'`

`Simulation callback when assertion fails` — Expression to evaluate when bounds are violated

`''` (default) | MATLAB expression

Specify a MATLAB expression to evaluate when the bounds specified on the **Bounds** tab are violated. All variables used in the expression must be in the MATLAB workspace.

Dependencies

To enable this parameter, select the **Enable assertion** parameter.

Programmatic Use

Block Parameter: `callback`
Type: character vector
Value: MATLAB expression
Default: `''`

`Stop simulation when assertion fails` — Stop simulation when bounds are violated

`off` (default) | `on`

To stop the simulation when the bounds specified on the **Bounds** tab are violated, select this parameter. If you do not select this option, the bound violation is reported as a warning in the MATLAB Command Window and the simulation continues.

If you run the simulation from the Simulink model, when the assertion fails, the block where the bound violation occurs is highlighted and an error message is displayed in the Simulation Diagnostics window.

Note Since selecting this option stops the simulation as soon as the assertion fails, bound violations that might occur later during the simulation are not reported. If you want all bound violations to be reported, do not select this option.

Dependencies

To enable this parameter, select the **Enable assertion** parameter.

Programmatic Use

Block Parameter: `stopWhenAssertionFail`
Type: character vector
Value: `'off'` | `'on'`
Default: `'off'`

`Output assertion signal` — Add assertion output port

`off` (default) | `on`

Add the $\mathbf{z^{-1}}$ assertion signal output port to the block. This port outputs the value of the assertion as a Boolean signal. When the bounds defined on the **Bounds** tab are violated, the assertion fails and the assertion signal is `0`. Otherwise, the assertion signal is `1`.

You can use the assertion signal to design complex assertion logic. For an example, see “Verify Model Using Simulink Control Design and Simulink Verification Blocks” on page 17-20.

Programmatic Use

Block Parameter: `export`
Type: character vector
Value: `'off'` | `'on'`
Default: `'off'`

More About

Using the Plot

You can view the gain and phase margins in one of the following plots. To specify the plot type, use the **Plot type** parameter.

- Bode plot
- Nichols plot
- Nyquist plot
- Table

By default:

- The Bode and Nyquist plots display the specified minimum margins.
- The Nichols plot shows the specified margins as bounds in the plot area.
- The table shows the minimum margins and highlights any computed margins that violate the bounds.

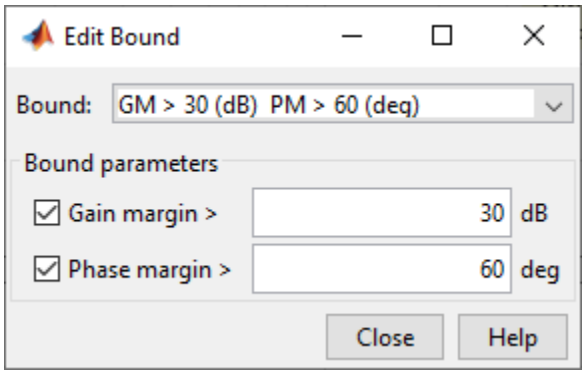
To view the computed gain and phase margins on the Bode, Nichols, or Nyquist plot, right-click the plot and select **Characteristics > Minimum Stability Margins**. The table displays the computed margins automatically.

In the plot window, you can:

- View the block parameters by clicking or selecting **Edit**.
- Highlight the block in the model by clicking or selecting **Highlight Simulink Block** in the **View** menu.
- Simulate the model by clicking .
- Add a legend to the plot by clicking .

The margin bounds you specify for the block appear on the plot. You can specify bounds on the **Bounds** tab.

To modify the margin bounds from the Bode, Nyquist, or Nichols plot window, right-click the plot, and select **Bounds > Edit Bound**.



In the Edit Bound dialog box, in the **Bound** drop-down list, select the margin bound. Then, specify the gain and phase margins and click **Close**.

After editing the margins from the plot window, update the bound values in the block by clicking **Update Block**.

See Also

Topics

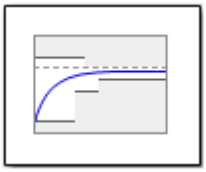
“Visualize Bode Response of Simulink Model During Simulation” on page 2-61
“Verify Model Using Simulink Control Design and Simulink Verification Blocks” on page 17-20
“Visualize Linear System of a Continuous-Time Model Discretized During Simulation” on page 2-91

Introduced in R2010b

Linear Step Response Plot, Check Linear Step Response Characteristics

Step response of linear system computed from nonlinear Simulink model

Library: Simulink Control Design / Linear Analysis Plots
Simulink Control Design / Model Verification



Description

The Linear Step Response Plot and Check Linear Step Response Characteristics blocks compute a linear system from a nonlinear Simulink model and plot the step response of the linear system during simulation. These blocks are identical except for the default settings on the **Bounds** tab.

- The Linear Step Response Plot does not define default bounds.
- The Check Linear Step Response Characteristics block defines default bounds and enables these bounds for assertion.

For more information on time domain analysis of linear systems, see “Time-Domain Responses”.

During simulation, the software linearizes the portion of the model between specified linearization inputs and outputs and then plots the step response of the linear system. You also can save the linear system as a variable in the MATLAB workspace.

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 in the MATLAB Command Window. 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 any bound is not satisfied, the signal is false (`0`).

To compute and plot the step response of various portions of your model, you can add multiple Linear Step Response Plot and Check Linear Step Response Characteristics blocks.

These blocks do not support code generation and can be used only in `Normal` simulation mode.

Ports

Input

`Trigger` — External trigger signal

scalar

Use this input port (indicated by) to connect an external trigger signal for computing the model linearization. To specify the type of trigger signal to detect, use the **Trigger type** parameter.

Dependencies

To enable this port, set the **Linearize on** parameter to `External trigger`.

Output

z^{-1} — Assertion signal

`1` | `0`

Output the value of the assertion signal as a logical value. If any bound specified on the **Bounds** tab is violated, the assertion signal is false (`0`). Otherwise, this signal is true (`1`).

By default, the data type of the output signal is double. To set the output data type as Boolean, in the Simulink model, in the Configuration Parameters dialog box, select the **Implement logic signals as Boolean data** parameter. This setting applies to all blocks in the model that generate logic signals.

You can use the assertion signal to design complex assertion logic. For an example, see “Verify Model Using Simulink Control Design and Simulink Verification Blocks” on page 17-20.

Dependencies

To enable this port, select the **Output assertion signal** parameter.

Parameters

`Show Plot` — Open plot

button

To view gain and phase margins computed during a simulation, click this button before starting the simulation. If you specify bounds on the **Bounds** tab, they are also shown on the plot.

To show the plot when opening the block, select the **Show plot on block open** parameter.

For more information on using the plot, see “Using the Plot” on page 19-111.

`Show plot on block open` — Open plot when opening block

`off` (default) | `on`

Select this parameter to open the plot when opening the block. You can then perform tasks, such as adding or modifying bounds, in the plot window instead of using the block parameters. To access the block parameters from the plot window, select **Edit** or click .

For more information on using the plot, see “Using the Plot” on page 19-111.

Programmatic Use
Block Parameter: `LaunchViewOnOpen`
Type: character vector
Value: `'off'` | `'on'`
Default: `'off'`

`Response Optimization` — Open Response Optimizer

button

Open the **Response Optimizer** app to optimize the model response to meet the design requirements specified on the **Bounds** tab.

This button is available only if you have Simulink Design Optimization software installed.

For more information on response optimization, see “Design Optimization to Meet Step Response Requirements (GUI)” (Simulink Design Optimization) and “Design Optimization to Meet Time-Domain and Frequency-Domain Requirements (GUI)” (Simulink Design Optimization).

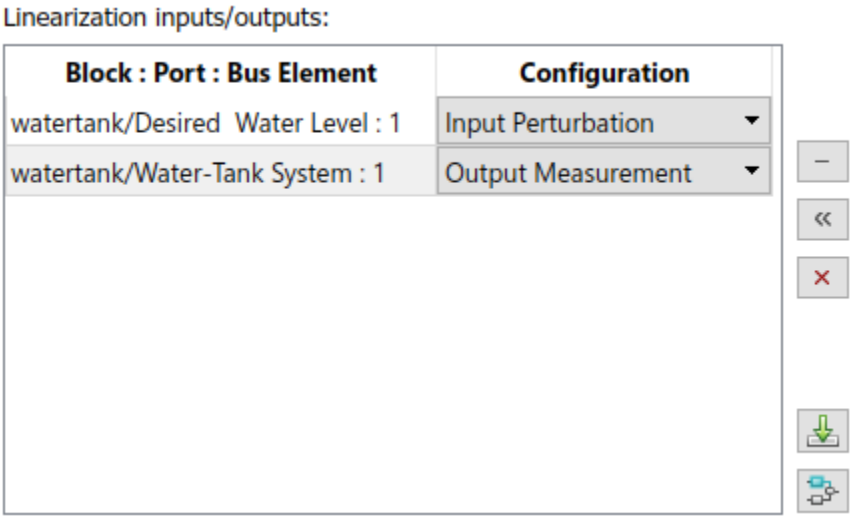
Linearizations

To specify the portion of the model to linearize and other linearization settings, use the parameters on the **Linearizations** tab. The default settings on this tab are the same for the Linear Step Response Plot and Check Linear Step Response Characteristics blocks.

`Linearization inputs/outputs` — Specify portion of model to linearize

linear analysis points

To specify the portion of the model to linearize, select signals from the Simulink model and add them as linearization inputs or outputs.



In the table, the **Block:Port:Bus Element** column shows the following information for each signal.

- Source block
- Output port of the source block to which the signal is connected

- Bus element name (if the signal is in a bus)

In the **Configuration** column, select the type of linear analysis point from the following types. For more information on linear analysis points, see “Specify Portion of Model to Linearize” on page 2-10.

- `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 a 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 a linearization input or output, the software generates a warning in the MATLAB Command Window and does not compute a linear system.

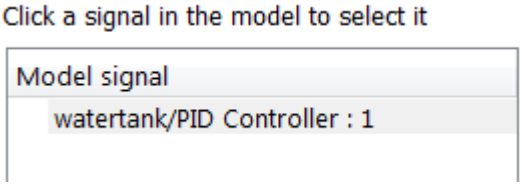
Edit Linearization Inputs and Outputs

To add linearization inputs and outputs:

1. To expand the signal selection area, click .

   The dialog box expands to display a **Click a signal in the model to select it** area.
2. In the Simulink model, select one or more signals.

   The selected signals appear in the **Model signal** table.

3. (Optional) For bus signals, expand the bus to select individual elements.

   Tip For large buses or other large lists of signals, you can filter the signal names. In the **Filter by name** box, enter search text. The name match is case-sensitive.

   To modify the filtering options, click . For more information on filtering options, see the **Enable regular expression** and **Show filtered results as a flat list** parameters.
4. To add the selected signal to the **Linearization inputs/outputs** table, click .
5. In the **Configuration** column, specify the signal type.

Alternatively, if you have linearization inputs and outputs defined in your model, you can add them to the **Linearization inputs/outputs** table by clicking .

To remove a signal from the **Linearization inputs/outputs** table, select the signal and click .

To highlight the source block of a signal in the Simulink model, select the signal in the **Linearization inputs/outputs** table and click .

`Enable regular expression` — Enable signal searching using regular expressions
`on` (default) | `off`

Select this option to enable the use of MATLAB regular expressions for filtering signal names. For example, entering `t$` in the **Filter by name** text box displays all signals whose names end with a lowercase `t` (and their immediate parents). For more information, see “Regular Expressions”.

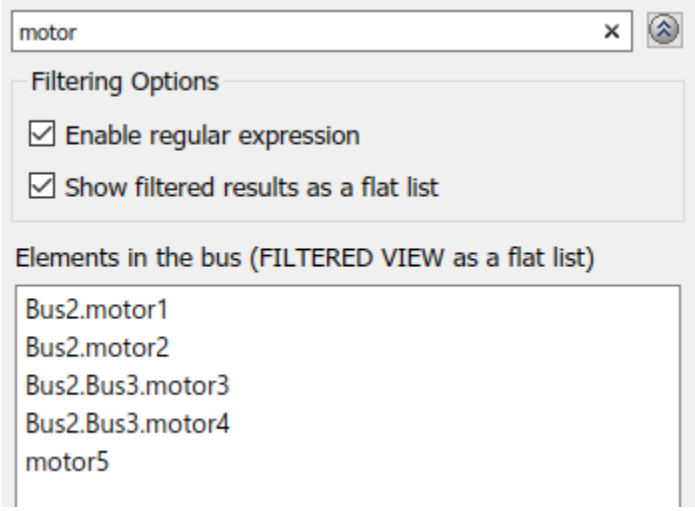
Dependencies

To enable this parameter, click next to the **Filter by name** text box.

`Show filtered results as a flat list` — Display filtered bus signal hierarchy using flat list
`off` (default) | `on`

Select this option to display the list of filtered signals in a flat list format. The flat list format uses dot notation to reflect the hierarchy of bus signals. The signals are filtered based on the text in the **Filter by name** text box.

The following figure shows an example of the flat list format for a filtered set of nested bus signals.



Dependencies

To enable this parameter, click next to the **Filter by name** text box.

`Linearize on` — When to compute linear model
`Simulation snapshots` (default) | `External trigger`

Use this parameter to specify when you want to compute a linear model.

To compute linear models at specified simulation snapshot times, set this parameter to `Simulation snapshots`. Specify snapshot times using the **Snapshot times** parameter.

Use simulation snapshots when you:

- Know one or more times when the model is at a steady-state operating point
- Want to compute linear systems at specific times

To compute linear models at trigger-based simulation events, set this parameter to `External trigger`. Selecting this option adds a trigger input port to the block, to which you connect your external trigger signal. To specify the type of trigger to detect, use the **Trigger type** parameter.

Use an external trigger when a signal generated during simulation indicates that the model is at a steady-state condition of interest. For example, for an aircraft model, you might want to compute the linear system whenever the fuel mass is a given fraction of the maximum fuel mass.

Programmatic Use

Block Parameter: `LinearizeAt`
Type: character vector
Value: `'SnapshotTimes'` | `'ExternalTrigger'`
Default: `'SnapshotTimes'`

`Snapshot times` — Simulation times at which to compute linear model

`0` (default) | positive real value | vector of positive real values

To compute a linear system at specific simulation times, such as a time that you know the model reaches a steady state operating point, specify one or more snapshot times. To specify multiple snapshot times, specify this parameter as a vector of positive values.

Snapshot times must be less than or equal to the simulation time specified in the Simulink model.

For examples of linearizing a model at simulation snapshot times, see:

- “Visualize Linear System at Multiple Simulation Snapshots” on page 2-85
- “Verify Model at Default Simulation Snapshot Time” on page 17-5
- “Verify Model at Multiple Simulation Snapshots” on page 17-13

Dependencies

To enable this parameter, set the **Linearize on** parameter to `Simulation snapshots`

Programmatic Use

Block Parameter: `SnapshotTimes`
Type: character vector
Value: `'0'` | positive real value | vector of positive real values
Default: `'0'`

`Trigger type` — Type of external trigger to detect

`Rising edge` (default) | `Falling edge`

Specify the trigger to detect in the external trigger signal as one of the following types.

- `Rising edge` — Use the rising edge of the trigger signal; that is, when the signal changes from `0` to `1`.
- `Falling edge` — Use the falling edge of the trigger signal; that is, when the signal changes from `1` to `0`.

Dependencies

To enable this parameter, set the **Linearize on** parameter to `External trigger`.

Programmatic Use
Block Parameter: `TriggerType`
Type: character vector
Value: `'rising'` | `'falling'`
Default: `'rising'`

`Enable zero-crossing detection` — Enable zero-crossing detection
`on` (default) | `off`

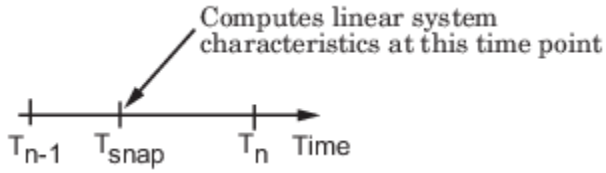
Select this option to enable zero-crossing detection.

When you set the **Linearize on** parameter to `Simulation snapshots`, enabling zero-crossing detection ensures that the software computes the linear model at the exact snapshot times you specify in the **Snapshot times** parameter.

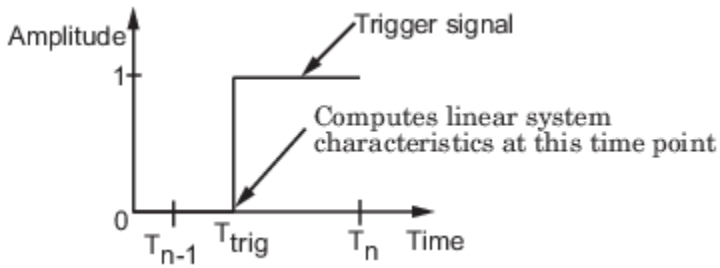
When you set the **Linearize on** parameter to `External trigger`, enabling zero-crossing detection ensures that the software computes the linear model at the exact time that the external trigger is detected. To specify the type of trigger, use the **Trigger type** parameter.

If you clear this option, the software computes the linear system at simulation times selected by the variable-step Simulink solver, which might not correspond to an exact snapshot time or the exact time when a trigger signal is detected.

For example, consider the case where the variable-step solver selects simulation times T_{n-1} and T_n. As shown in the following figure, the specified snapshot time T_{snap} can be between the selected simulation times. If you enable zero-crossing detection, the solver also simulates the model at time T_{snap} and computes the linear model at this point.



Similarly, the external trigger can be detected at a time T_{trig} that is between the selected simulation times. If you enable zero-crossing detection, the solver also simulates the model at time T_{trig} and computes the linear model at this point.



In both cases, if you do not enable zero-crossing detection, the software computes the linear model at either T_{n-1} or T_n.

For more information on zero-crossing detection, see “Zero-Crossing Detection”.

Dependencies

This parameter is ignored when you use a fixed-step Simulink solver.

Programmatic Use

Block Parameter: `ZeroCross`
Type: character vector
Value: `'on'` | `'off'`
Default: `'on'`

`Use exact delays` — Use exact delays in linear model

`off` (default) | `on`

Select this option to compute a linear model with exact delays. If you clear this option, the linear model uses Padé approximations of any delays.

For more information on linearizing models with delays, see “Linearize Models with Delays” on page 2-78.

Programmatic Use

Block Parameter: `UseExactDelayModel`
Type: character vector
Value: `'off'` | `'on'`
Default: `'off'`

`Linear system sample time` — Sample time of linear system

`'auto'` (default) | positive finite value | `0`

To compute a linear system with the specified sample time, the software coverts sample times in the model using the method you specify in the **Sample time rate conversion method** parameter.

You can set the sample time to one of the following values.

- `auto` — If all blocks in the model are continuous-time, use a sample time of `0`. Otherwise, set the sample time to the least common multiple of the nonzero sample times in the model.
- Positive finite value — Create a discrete-time model with the specified sample time
- `0` — Create a continuous-time model

Programmatic Use

Block Parameter: `SampleTime`
Type: character vector
Value: `'auto'` | positive finite value | `'0'`
Default: `'auto'`

`Sample time rate conversion method` — Rate conversion method

`Zero-Order Hold` (default) | `Tustin (bilinear)` | `Tustin with Prewarping` | ...

Method for converting sample times during linearization, specified as one of the following values.

- `Zero-Order Hold` — Zero-order hold, where the control inputs are assumed piecewise constant over the sample time `Ts`. This method usually performs better in the 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. This method usually performs better in the frequency domain.

- `Tustin with Prewarping` — Bilinear (Tustin) approximation with frequency prewarping. Specify the prewarp frequency using the **Prewarp frequency** parameter. This method usually performs better in the frequency domain. Use this method to ensure matching in a frequency region of interest.
- `Upsampling when possible, Zero-Order Hold otherwise` — Upsample a discrete-time system when possible; otherwise, use a zero-order hold.
- `Upsampling when possible, Tustin otherwise` — Upsample a discrete-time system when possible; otherwise, use a Tustin approximation.
- `Upsampling when possible, Tustin with Prewarping otherwise` — Upsample a discrete-time system when possible; otherwise, use a Tustin approximation with frequency prewarping.

You can upsample only when you convert a discrete-time system to a new faster sample time that is an integer multiple of the sample time of the original system.

For more information on rate conversion and linearization of multirate models, see:

- “Multirate Linearization Algorithm” on page 2-140
- “Linearize Models Using Different Rate Conversion Methods” on page 2-145
- “Continuous-Discrete Conversion Methods”

Note If you use a rate conversion method other than `Zero-Order Hold`, the converted states no longer have the same physical meaning as the original states. As a result, the state names in the resulting LTI system change to `'?'`.

Dependencies

To enable this parameter, set the **Linear system sample time** parameter to a value other than `auto`.

Programmatic Use

Block Parameter: `RateConversionMethod`
Type: character vector
Value: `'zoh'` | `'tustin'` | `'prewarp'` | `'upsampling_zoh'` | `'upsampling_tustin'` | `'upsampling_prewarp'`
Default: `'zoh'`

`Prewarp frequency` — Prewarp frequency for Tustin rate conversion

`'10'` (default) | positive scalar

Prewarp frequency for Tustin rate conversion in radians per second, specified as a scalar value less than the Nyquist frequency before and after resampling.

Dependencies

To enable this parameter, set the **Sample time rate conversion method** parameter to one of the following values.

- `Tustin with Prewarping`
- `Upsampling when possible, Tustin with Prewarping otherwise`

Programmatic Use
Block Parameter: `PreWarpFreq`
Type: character vector
Value: positive scalar
Default: `'10'`

`Use full block names` — Use full block path in state, input, and output names

`off` (default) | `on`

To show the state, input, and output names of the computed linear system using their full block path, select this parameter. For example, in the `scdcstr` model, a state in the `Integrator1` block of the `CSTR` subsystem appears with its full block path as `scdcstr/CSTR/Integrator1`.

If you clear this parameter, only the names of the states, inputs, and outputs are used, which is useful when the signal names are unique and you know their locations in your Simulink model. In the preceding example, the state name of the integrator block appears as `Integrator1`.

The computed linear system is a state-space object (`ss`). The state, input, and output names for the system appear in the following state-space object properties.

Input, Output, or State Name	State-Space Object Property
Linearization input names	`InputName`
Linearization output names	`OutputName`
State names	`StateName`

Programmatic Use
Block Parameter: `UseFullBlockNameLabels`
Type: character vector
Value: `'off'` | `'on'`
Default: `'off'`

`Use bus signal names` — Use bus signal names in linear system

`off` (default) | `on`

When you select an entire bus as a linearization input or output, select this parameter to use the signal names of the individual bus elements in the computed linear system. If you do not enable this option, the bus channel numbers are used instead.

Note Selecting an entire bus signal is not recommended. Instead, select individual bus elements.

Bus signal names appear when the linearization input or output is from one 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

Dependencies

Using this parameter is not supported when your model contains mux/bus mixtures.

Programmatic Use

Block Parameter: `UseBusSignalLabels`
Type: character vector
Value: `'off'` | `'on'`
Default: `'off'`

Bounds

To define bounds on step response characteristics and specify whether to check for violations of these bounds, use the parameters on the **Bounds** tab. The default settings on this tab are different for the Linear Step Response Plot and Check Linear Step Response Characteristics blocks.

`Include step response bound in assertion` — Check whether step response violates bounds on characteristics

`on` | `off`

Select this parameter to check whether the step response characteristics violate the bounds specified on the **Bounds** tab.

By default, this parameter is cleared for the Linear Step Response Plot block and selected for the Check Linear Step Response Characteristics block.

Dependencies

This parameter is used for assertion only if you select the **Enable assertion** parameter.

Programmatic Use

Block Parameter: `EnableStepResponseBound`
Type: character vector
Value: `'on'` | `'off'`
Default: `'off'` for Linear Step Response Plot block, `'on'` for Check Linear Step Response Characteristics block

`Final value` — Final value of output signal in response to step input

finite real scalar

To compute the other step response characteristics on the **Bounds** tab, the block uses the expected final value of the output signal in response to a step input.

By default, this parameter is `[]` for the Linear Step Response Plot block and `1` for the Check Linear Step Response Characteristics block.

Programmatic Use

Block Parameter: `FinalValue`
Type: character vector
Value: finite real scalar
Default: `'[]'` for Linear Step Response Plot block, `'1'` for Check Linear Step Response Characteristics block

`Rise time` — Upper bound on rise time

finite positive scalar

Upper bound on the time in seconds for the step response to reach a percentage of the final value.

- To specify the final value, use the **Final value** parameter.

- To specify the percentage of the final value, use the **% Rise** parameter.

By default, this parameter is `[]` for the Linear Step Response Plot block and `5` for the Check Linear Step Response Characteristics block.

Programmatic Use
Block Parameter: `RiseTime`
Type: character vector
Value: finite positive scalar
Default: `'[]'` for Linear Step Response Plot block, `'5'` for Check Linear Step Response Characteristics block

`% Rise` — Percentage of final value used for computing rise time

positive scalar less than (100 - % Settling)

Percentage of the final value used to define the rise time bound.

- To specify the final value, use the **Final value** parameter.
- To specify the rise time bound, use the **Rise time** parameter.

By default, this parameter is `[]` for the Linear Step Response Plot block and `80` for the Check Linear Step Response Characteristics block.

Programmatic Use
Block Parameter: `PercentRise`
Type: character vector
Value: positive scalar less than (100 - % Settling)
Default: `'[]'` for Linear Step Response Plot block, `'80'` for Check Linear Step Response Characteristics block

`Settling time` — Upper bound on settling time

finite scalar greater than rise time

Upper bound on the time in seconds 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 specified settling percentage of the final value.

- To specify the final value, use the **Final value** parameter.
- To specify the settling percentage, use the **% Settling** parameter.

By default, this parameter is `[]` for the Linear Step Response Plot block and `7` for the Check Linear Step Response Characteristics block.

Programmatic Use
Block Parameter: `SettlingTime`
Type: character vector
Value: finite scalar greater than rise time
Default: `'[]'` for Linear Step Response Plot block, `'7'` for Check Linear Step Response Characteristics block

`% Settling` — Percentage of final value used for computing settling range

positive scalar less than (100 - % Rise)

Percentage of the final value used to define the settling range bounds. The settling range is defined as the final value plus or minus the specified settling percentage of the final value.

- To specify the final value, use the **Final value** parameter.
- To specify the settling time bound, use the **Settling time** parameter.

By default, this parameter is `[]` for the Linear Step Response Plot block and `1` for the Check Linear Step Response Characteristics block.

Programmatic Use
Block Parameter: `PercentSettling`
Type: character vector
Value: positive scalar less than (100 - % Settling)
Default: `'[]'` for Linear Step Response Plot block, `'1'` for Check Linear Step Response Characteristics block

`% Overshoot` — Upper bound on overshoot

scalar between 0 and 100

The amount by which the step response can exceed the final value, defined as a percentage of the final value. To specify the final value, use the **Final value** parameter.

By default, this parameter is `[]` for the Linear Step Response Plot block and `10` for the Check Linear Step Response Characteristics block.

Programmatic Use
Block Parameter: `PercentOvershoot`
Type: character vector
Value: scalar between 0 and 100
Default: `'[]'` for Linear Step Response Plot block, `'10'` for Check Linear Step Response Characteristics block

`% Undershoot` — Upper bound on overshoot

scalar between 0 and 100

The amount by which the step response can undershoot the initial value, defined as a percentage of the final value. To specify the final value, use the **Final value** parameter.

By default, this parameter is `[]` for the Linear Step Response Plot block and `1` for the Check Linear Step Response Characteristics block.

Programmatic Use
Block Parameter: `PercentUndershoot`
Type: character vector
Value: scalar between 0 and 100
Default: `'[]'` for Linear Step Response Plot block, `'1'` for Check Linear Step Response Characteristics block

Logging

To control whether linearization results computed during the simulation are saved, use the parameters on the **Logging** tab. The default settings on this tab are the same for the Linear Step Response Plot and Check Linear Step Response Characteristics blocks.

`Save data to workspace` — Save linear systems for further analysis

`off` (default) | `on`

Select this parameter to save the computed linear systems for further analysis or control design. The data is saved in a structure with the following fields.

- `time` — Simulation times at which the linear systems are computed.
- `values` — State-space model representing the linear system. If the linear system is computed at multiple simulation times, `values` is an array of state-space models.
- `operatingPoints` — Operating points corresponding to each linear system in `values`. To enable this field, select the **Save operating points for each linearization** parameter.

To specify the name of the saved data structure, use the **Variable name** property.

The location of the saved data structure depends upon the configuration of the Simulink model.

- If the model is not configured to save simulation output as a single object, the data structure is a variable in the MATLAB workspace.
- If the 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 Configuration Parameters dialog box, select the **Single simulation output** parameter.

For more information about data logging in Simulink, see “Export Simulation Data” and the `Simulink.SimulationOutput` reference page.

Programmatic Use

Block Parameter: `SaveToWorkspace`
Type: character vector
Value: `'off'` | `'on'`
Default: `'off'`

`Variable name` — Name of data structure for saving linear systems

`sys` (default) | character vector

Specify the name of the data structure that stores linear systems computed during simulation.

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.

Dependencies

To enable this parameter, select the **Save data to workspace** parameter.

Programmatic Use

Block Parameter: `SaveName`
Type: character vector
Default: `'sys'`

`Save operating points for each linearization` — Save operating points with linearization

`off` (default) | `on`

Select this parameter to save the operating point at which each linearization is computed. Selecting this parameter adds the `operatingPoints` field to the saved data structure.

Dependencies

To enable this parameter, select the **Save data to workspace** parameter.

Programmatic Use

Block Parameter: `SaveOperatingPoints`
Type: character vector
Value: `'off'` | `'on'`
Default: `'off'`

Assertion

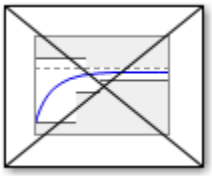
To control the assertion behavior of the block when bounds defined on the **Bounds** tab are violated, use the parameters on the **Assertion** tab. The default settings on this tab are the same for the Linear Step Response Plot and Check Linear Step Response Characteristics blocks.

`Enable assertion` — Enable bound checking

`on` (default) | `off`

To check whether the bounds defined on the **Bounds** tab are satisfied during the simulation, select this parameter. When a bound is not satisfied, the assertion fails and a warning is generated.

Clearing this parameter disables assertion; that is, the block no longer checks that the specified bounds are satisfied. The block icon also updates to indicate that assertion is disabled.



By default, on the **Bounds** tab:

- The Linear Step Response Plot block does not have defined bounds.
- The Check Linear Step Response Characteristics block has defined bounds.

You can configure your Simulink model to enable or disable all model verification blocks and override the **Enable assertion** parameter. To do so, in the Simulink model, in the Configuration Parameters dialog box, specify the **Model Verification block enabling** parameter.

Programmatic Use

Block Parameter: `enabled`
Type: character vector
Value: `'on'` | `'off'`
Default: `'on'`

`Simulation callback when assertion fails` — Expression to evaluate when bounds are violated

`''` (default) | MATLAB expression

Specify a MATLAB expression to evaluate when the bounds specified on the **Bounds** tab are violated. All variables used in the expression must be in the MATLAB workspace.

Dependencies

To enable this parameter, select the **Enable assertion** parameter.

Programmatic Use

Block Parameter: `callback`
Type: character vector
Value: MATLAB expression
Default: `''`

`Stop simulation when assertion fails` — Stop simulation when bounds are violated

`off` (default) | `on`

To stop the simulation when the bounds specified on the **Bounds** tab are violated, select this parameter. If you do not select this option, the bound violation is reported as a warning in the MATLAB Command Window and the simulation continues.

If you run the simulation from the Simulink model, when the assertion fails, the block where the bound violation occurs is highlighted and an error message is displayed in the Simulation Diagnostics window.

Note Since selecting this option stops the simulation as soon as the assertion fails, bound violations that might occur later during the simulation are not reported. If you want all bound violations to be reported, do not select this option.

Dependencies

To enable this parameter, select the **Enable assertion** parameter.

Programmatic Use

Block Parameter: `stopWhenAssertionFail`
Type: character vector
Value: `'off'` | `'on'`
Default: `'off'`

`Output assertion signal` — Add assertion output port

`off` (default) | `on`

Add the $\mathbf{z^{-1}}$ assertion signal output port to the block. This port outputs the value of the assertion as a Boolean signal. When the bounds defined on the **Bounds** tab are violated, the assertion fails and the assertion signal is `0`. Otherwise, the assertion signal is `1`.

You can use the assertion signal to design complex assertion logic. For an example, see “Verify Model Using Simulink Control Design and Simulink Verification Blocks” on page 17-20.

Programmatic Use

Block Parameter: `export`
Type: character vector
Value: `'off'` | `'on'`
Default: `'off'`

More About

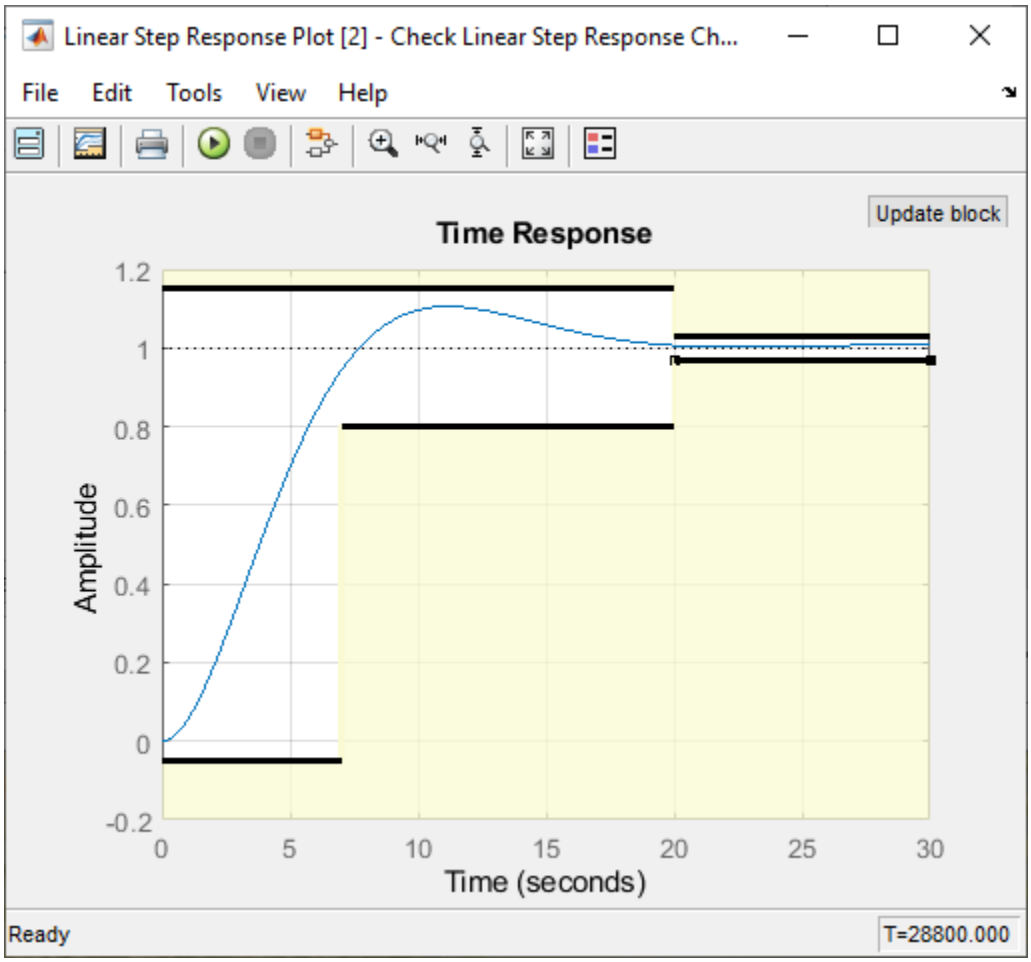
Using the Plot

In the plot window, you can:

- View the block parameters by clicking or selecting **Edit**.
- Highlight the block in the model by clicking or selecting **Highlight Simulink Block** in the **View** menu.
- Simulate the model by clicking .
- Add a legend to the plot by clicking .

To display response characteristics, such as the peak response or settling time, right-click the plot. Then, under **Characteristics**, select the characteristics to show.

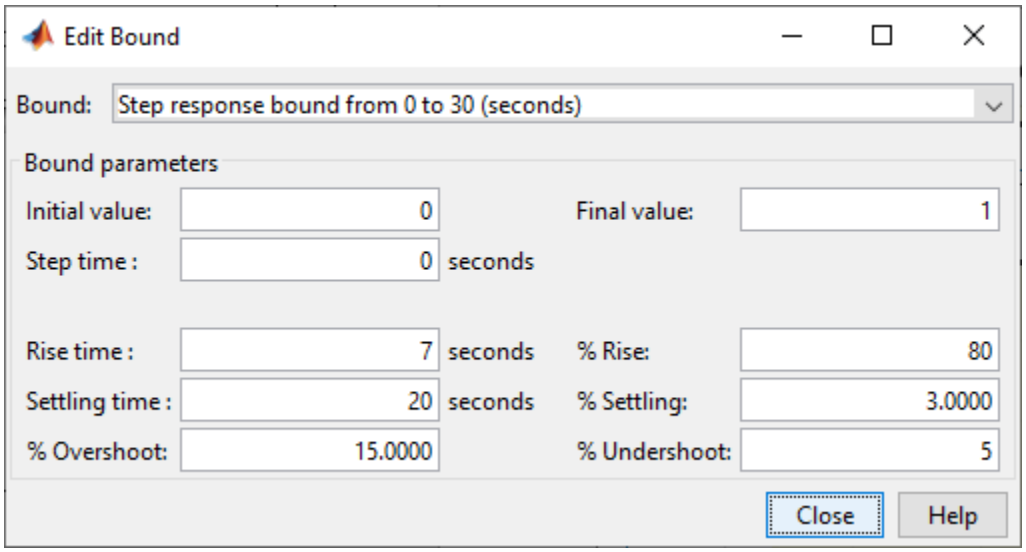
The bounds you specify for the block appear on the plot.



You can specify bounds on the **Bounds** tab.

To modify a bound, you can drag the bound in the plot. You can also:

- Right-click the plot and select **Bounds > Edit Bound**.
- Right-click the bound and select **Edit**.



Then, in the Edit Bound dialog box, specify the parameter values and click **Close**.

After editing bounds from the plot window, update the bound value in the block by clicking **Update Block**.

See Also

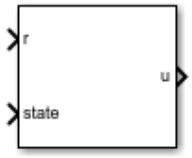
Topics

“Visualize Bode Response of Simulink Model During Simulation” on page 2-61
“Verify Model Using Simulink Control Design and Simulink Verification Blocks” on page 17-20
“Visualize Linear System of a Continuous-Time Model Discretized During Simulation” on page 2-91

Introduced in R2010b

Model Reference Adaptive Control

Compute control actions to make a controlled system track a reference model
Library: Simulink Control Design



Description

The Model Reference Adaptive Control block computes control actions to make the states of an uncertain controlled system track the states of an ideal reference model.

The MRAC controller computes control actions using:

- A model of the external disturbances and uncertainty in the system being controlled.
- Feedback gains that relate the state of the controlled system to the control signal.
- Feedforward gains that relate the reference signal to the control signal.

Based on the tracking error, the controller can adapt the parameters of the disturbance model and gains in real time.

For more information, see “Model Reference Adaptive Control” on page 15-28.

Ports

Input

`r` — Reference signal
scalar signal | vector signal

Provide the reference signal for the controlled system to follow.

`state` — Controlled system state
vector signal

Provide the current state vector from the controlled system.

`phi` — Disturbance model features
vector signal | matrix signal

You can provide custom disturbance model features using the **phi** input port.

Dependencies

To enable this input port, set the **Feature type** parameter to `Use External Source for Feature`.

Output

u — Control input
scalar signal | matrix signal

Connect the control input signal to the input of the controlled system. The control input is a vector signal with length equal to the number of control inputs in the controlled system.

u_ad — Disturbance and uncertainty estimate
scalar signal | matrix signal

The disturbance and uncertainty estimate is the product of the of the disturbance model features and the disturbance model weight vector. You can use this signal to compare the estimated disturbance model with the actual disturbances in your system.

Dependencies

To enable this output port, select the **Output disturbance/uncertainty estimate** parameter.

Parameters

System

Specify nominal and reference model dynamics.

A — Nominal model state matrix
0 (default) | square matrix

Specify the state matrix for the nominal model as an *N*-by-*N* matrix, where *N* is the number of states in the controlled system.

Programmatic Use
Block Parameter: 'A'
Type: character vector
Values: square matrix
Default: '0'

B — Nominal model control effective matrix
1 (default) | vector | matrix

Specify the control effective matrix of the nominal model as a nonzero *N*-by-*M* matrix, where *N* is the number of states in the controlled system and *M* is the number of control inputs in the controlled system.

Programmatic Use
Block Parameter: 'B'
Type: character vector
Values: vector | matrix
Default: '1'

Am — Reference model state matrix
-1 (default) | matrix

Specify the state matrix for the reference model a matrix with the same dimensions as parameter **A**. For a stable reference model, **Am** must be a Hurwitz matrix for which every eigenvalue must have a strictly negative real part.

Programmatic Use
Block Parameter: `'Am'`
Type: character vector
Values: square matrix
Default: `'-1'`

`Bm` — Reference model control effective matrix

1 (default) | vector | matrix

Specify the control effective matrix of the reference model as a nonzero matrix with same dimensions as parameter **B**.

Programmatic Use
Block Parameter: `'Bm'`
Type: character vector
Values: vector | matrix
Default: `'[0;4]'`

Control Gains

Define initial feedback and feedforward gains for model matching. You can configure the block to update these control gains and adjust the corresponding learning rates.

`Feedback gains` — Feedback gains for model matching

`0` (default) | finite scalar | matrix of finite values

Initial feedback gain values. If you do not select the **Adapt feedback gains** parameter, then the controller holds the specified feedback gains at these initial values.

Programmatic Use
Block Parameter: `'kx'`
Type: character vector
Values: finite scalar | matrix of finite values
Default: `'0'`

`Adapt feedback gains` — Option to adapt feedback gains

`on` (default) | `off`

When you select this parameter, the controller adapts the feedback gains based on the difference between the states of the controlled system and the reference model.

Programmatic Use
Block Parameter: `'FBLrEnable'`
Type: character vector
Values: `'on'` | `'off'`
Default: `'on'`

`Learning rate (gamma_x)` — Learning rate for adapting feedback gains

1 (default) | finite positive scalar

Use this parameter to control the rate at which the controller adapts the feedback gains. A larger value increases the size of the gain updates.

Programmatic Use
Block Parameter: `'gamma_kx'`

Type: character vector
Values: finite positive scalar
Default: `'2'`

`Use learning modification` — Option to enable learning modification for updating the feedback gain learning rate

`on` (default) | `off`

To add robustness at higher gains, select this option to add a momentum term to the feedback gain updates. Configure the learning modification on the **Learning Modification** tab.

Programmatic Use

Block Parameter: `'FBLrEnableMod'`
Type: character vector
Values: `'on'` | `'off'`
Default: `'on'`

`Feedforward gains` — Feedforward gains for model matching

`0` (default) | finite scalar | matrix of finite values

Initial feedforward gain values. If you do not select the **Adapt feedforward gains** parameter, then the controller holds the specified feedforward gains at these initial values.

Programmatic Use

Block Parameter: `'kr'`
Type: character vector
Values: finite scalar | matrix of finite values
Default: `'0'`

`Adapt feedforward gains` — Option to adapt feedforward gains

`on` (default) | `off`

When you select this parameter, the controller adapts the feedforward gains based on the difference between the states of the controlled system and the reference model.

Programmatic Use

Block Parameter: `'FFLrEnable'`
Type: character vector
Values: `'on'` | `'off'`
Default: `'on'`

`Learning rate (gamma_r)` — Learning rate for adapting feedforward gains

`1` (default) | finite positive scalar

Use this parameter to control the rate at which the controller adapts the feedforward gains. A larger value increases the size of the gain updates.

Programmatic Use

Block Parameter: `'gamma_kr'`
Type: character vector
Values: finite positive scalar
Default: `'2'`

`Use learning modification` — Option to enable learning modification for updating the feedforward gain learning rate

`on` (default) | `off`

To add robustness at higher gains, select this option to add a momentum term to the feedforward gain updates. Configure the learning modification on the **Learning Modification** tab.

Programmatic Use

Block Parameter: `'FFLrEnableMod'`
Type: character vector
Values: `'on'` | `'off'`
Default: `'on'`

Disturbance Model

Configure the disturbance and uncertainty model used by the block. During operation, the block adapts the disturbance model parameters.

`Enable disturbance adaptation` — Option to enable adaptation to system disturbances and uncertainties

`on` (default) | `off`

When you select this parameter, the controller uses a disturbance model to estimate the uncertainty and external disturbances in the controlled system. The controller adapts the parameters of the disturbance model based on the error between the states of the controlled system and reference model.

The disturbance model has the form $w^T\phi(x)$.

- $\phi(x)$ is the disturbance model feature vector. To configure the feature vector, use the **Feature type** parameter.
- w^T is a weighting matrix that contains disturbance model parameters. The controller adjusts its disturbance and uncertainty model by adapting these parameters.

Programmatic Use

Block Parameter: `'AdaptEnable'`
Type: character vector
Values: `'on'` | `'off'`
Default: `'on'`

`Learning rate (gamma_w)` — Learning rate for adapting disturbance model parameters

1 (default) | finite positive scalar

Use this parameter to control the rate at which the controller adapts the disturbance model parameters. A larger value increases the size of the parameter updates.

Dependencies

To enable this parameter, select the **Enable disturbance adaptation** parameter.

Programmatic Use

Block Parameter: `'gamma'`
Type: character vector
Values: finite positive scalar
Default: `'100'`

`Use learning modification` — Option to enable learning modification for updating disturbance model parameters

`on` (default) | `off`

To add robustness at higher gains, select this option to add a momentum term to the disturbance parameter updates. Configure the learning modification on the **Learning Modification** tab.

Dependencies

To enable this parameter, select the **Enable disturbance adaptation** parameter.

Programmatic Use
Block Parameter: `'WLrEnableMod'`
Type: character vector
Values: `'on'` | `'off'`
Default: `'on'`

`Channel learning rate (Q)` — Channel learning rate used for the Lyapunov solution

`1` (default) | finite positive scalar

The channel learning rate Q is a weighting matrix for state tracking errors in the Lyapunov function for the error dynamics. The larger the value of Q, the faster the tracking error goes to zero. However, a larger value of Q also creates larger transients and a less robust system.

Dependencies

To enable this parameter, select the **Enable disturbance adaptation** parameter.

Programmatic Use
Block Parameter: `'Q'`
Type: character vector
Values: finite positive scalar
Default: `'1'`

`Feature type` — Disturbance model feature type

`State` (default) | `Radial Basis Function` | `Use External Source for Feature`

Select one of the following types for the disturbance model feature vector.

- `State` — Use the states from the controlled plant as the disturbance model. This option can under represent the uncertainty and therefore perform poorly.
- `Radial Basis Function` — Use Gaussian radial basis functions to create the feature vector.
- `Use External Source for Feature` — Add the **phi** input port to the block. Use this port to provide your own custom feature vector.

For more information on when to use each type of feature vector, see “Disturbance and Uncertainty Model Features” on page 15-31.

Dependencies

To enable this parameter, select the **Enable disturbance adaptation** parameter.

Programmatic Use
Block Parameter: `'FeatureTypeOptions'`
Type: character vector
Values: `Radial Basis Function` | `State` | `Use External Source for Feature`
Default: `Radial Basis Function`

`Number of RBF centers` — Number of radial basis function centers for disturbance model

`20` (default) | positive integer

Number of radial basis function (RBF) centers to use in the disturbance model. The RBF centers are evenly spaced across the span defined by the **Centers min** and **Centers max** parameters.

Dependencies

To enable this parameter, select the **Enable disturbance adaptation** parameter and set the **Feature type** parameter to `Radial Basis Function`.

Programmatic Use

Block Parameter: `'nCen'`
Type: character vector
Values: positive integer
Default: `'20'`

`Centers min` — Lower limit for radial basis function centers

`-1` (default) | finite scalar | vector

Specify the lower limits for the radial basis function centers. If you specify a scalar value, the same minimum is used for all basis functions. Otherwise, you must specify a vector with length equal to the **Number of RBF centers** parameter.

The **Centers min** parameter must be less than the **Centers max** parameter.

Dependencies

To enable this parameter, select the **Enable disturbance adaptation** parameter and set the **Feature type** parameter to `Radial Basis Function`.

Programmatic Use

Block Parameter: `'cSpanMin`
Type: character vector
Values: finite scalar | vector
Default: `'-1'`

`Centers max` — Upper limit for radial basis function centers

1 (default) | finite scalar | vector

Specify the upper limits for the radial basis function centers. If you specify a scalar value, the same maximum is used for all basis functions. Otherwise, you must specify a vector with length equal to the **Number of RBF centers** parameter.

The **Centers max** parameter must be greater than the **Centers min** parameter.

Dependencies

To enable this parameter, select the **Enable disturbance adaptation** parameter and set the **Feature type** parameter to `Radial Basis Function`.

Programmatic Use

Block Parameter: `'cSpanMax`
Type: character vector
Values: finite scalar | vector
Default: `'1'`

`Bandwidth` — Radial basis function standard deviation

5 (default) | positive scalar | vector

Specify the standard deviation for the Gaussian basis function kernel. If you specify a scalar value, the same standard deviation is used for all basis functions. Otherwise, you must specify a vector with length equal to the **Number of RBF centers** parameter.

Dependencies

To enable this parameter, select the **Enable disturbance adaptation** parameter and set the **Feature type** parameter to `Radial Basis Function`.

Programmatic Use

Block Parameter: `'cSig'`
Type: character vector
Values: positive scalar | vector
Default: `'5'`

`Output disturbance/uncertancy estimate` — Option to output disturbance and uncertainty estimate

`on` (default) | `off`

Select this parameter to add the **u_ad** output port.

Dependencies

To enable this parameter, select the **Enable disturbance adaptation** parameter and set the **Feature type** parameter to `Radial Basis Function`.

Programmatic Use

Block Parameter: `'adaptiveCntrlOutport'`
Type: character vector
Values: `'on'` | `'off'`
Default: `'on'`

Learning Modification

To add robustness at higher gains, you can modify the parameter updates to include a momentum term. The equations in the **Parameter Update Formula** section show the update formulas for the current block configuration.

`Modification Methods` — Modification method for updating parameters

`Sigma Modification` (default) | `e-Modification` | `None`

Select one of the following options for computing the momentum term that is added to the parameter updates.

- `Sigma Modification` — The momentum term is the product of the momentum weight parameter σ and the current parameter values.
- `e-Modification` — Scale the sigma-modification momentum term by the magnitude of the error value
- `None` — Do not use learning modification.

To specify σ, use the **Sigma** parameter.

Programmatic Use

Block Parameter: `'modChoice'`
Type: character vector

Values: `'Sigma Modification'` | `'e-Modification'` | `'None'`
Default: `'Sigma Modification'`

`Sigma` — Momentum weight for parameter updates

`0.1` (default) | scalar

Specify the value for the momentum weight term. A larger value increases the size of the parameter updates gain and parameter updates.

Dependencies

To enable this parameter, set the **Modification Methods** parameter to either `Sigma Modification` or `e-Modification`.

Programmatic Use

Block Parameter: `'sigma_val'`
Type: character vector
Values: finite positive scalar
Default: `'0.1'`

Extended Capabilities

C/C++ Code Generation

Generate C and C++ code using Simulink® Coder™.

See Also

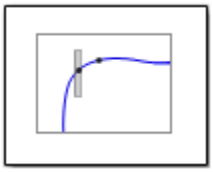
Topics

“Model Reference Adaptive Control” on page 15-28

Introduced in R2021b

Nichols Plot

Nichols plot of linear system approximated from nonlinear Simulink model



Library

Simulink Control Design

Description

This block is the 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 in the MATLAB Command Window. 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 any 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 19-124. • “Click a signal in the model to select it” on page 19-126.
	Specify settings.	In **Linearizations** tab: • “Linearize on” on page 19-128. • “Snapshot times” on page 19-129. • “Trigger type” on page 19-130.
	Specify algorithm options.	In **Algorithm Options** of **Linearizations** tab: • “Enable zero-crossing detection” on page 19-130. • “Use exact delays” on page 19-131. • “Linear system sample time” on page 19-132. • “Sample time rate conversion method” on page 19-133. • “Prewarp frequency (rad/s)” on page 19-134.
	Specify labels for linear system I/Os and state names.	In **Labels** of **Linearizations** tab: • “Use full block names” on page 19-134. • “Use bus signal names” on page 19-135.
Plot the linear system.		**Show Plot** on page 19-149
Specify the feedback sign for closed-loop gain and phase margins.		“Feedback sign” on page 19-143 in **Bounds** tab.

Task	Parameters
(Optional) Specify bounds on gains and phases of the linear system for assertion.	In **Bounds** tab: • Include gain and phase margins in assertion on page 19-136. • Include closed-loop peak gain in assertion on page 19-138. • Include open-loop gain-phase bound in assertion on page 19-140.
Specify assertion options (only when you specify bounds on the linear system).	In **Assertion** tab: • “Enable assertion” on page 19-146. • “Simulation callback when assertion fails (optional)” on page 19-147. • “Stop simulation when assertion fails” on page 19-147. • “Output assertion signal” on page 19-148.
Save linear system to MATLAB workspace.	“Save data to workspace” on page 19-143 in **Logging** tab.
Display plot window instead of block parameters dialog box on double-clicking the block.	“Show plot on block open” on page 19-149.

Linearization inputs/outputs

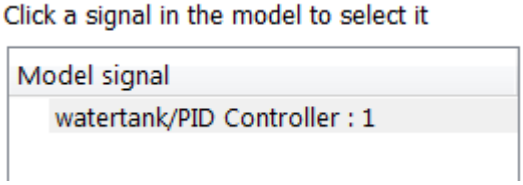
Linearization inputs and outputs that define the portion of a nonlinear Simulink model to linearize.

If you have defined the linearization input and output in the Simulink model, the block automatically detects these points and displays them in the **Linearization inputs/outputs** area. Click at any time to update the **Linearization inputs/outputs** table with I/Os from the model. To add other analysis points:

1. Click .

   The dialog box expands to display a **Click a signal in the model to select it** on page 19-126 area and a new button.
2. Select one or more signals in the Simulink Editor.

   The selected signals appear under **Model signal** in the **Click a signal in the model to select it** area.



3 (Optional) For buses, expand the bus signal to select individual elements.

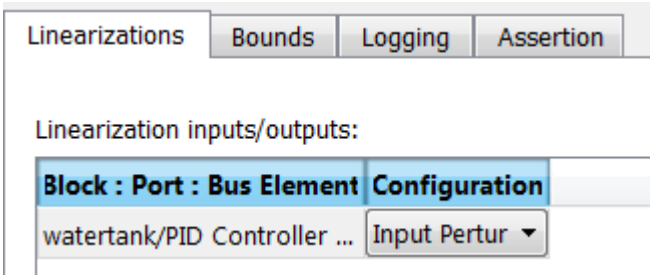
Tip For large buses or other large lists of signals, you can enter search text for filtering element names in the **Filter by name** edit box. The name match is case-sensitive. Additionally, you can enter a MATLAB regular expression.

To modify the filtering options, click . To hide the filtering options, click .

Filtering Options

- “Enable regular expression” on page 19-127
- “Show filtered results as a flat list” on page 19-127

4 Click « to add the selected signals to the **Linearization inputs/outputs** table.



To remove a signal from the **Linearization inputs/outputs** table, select the signal and click ×.

Tip To find the location in the Simulink model corresponding to a signal in the **Linearization inputs/outputs** table, select the signal in the table and click .

The table displays the following information about the selected signal:

Block : Port : Bus Element — Name of the block associated with the input/output. The number adjacent to the block name is the port number where the selected bus signal is located. The last entry is the selected bus element name.

Configuration

Type of linearization point:

- `Open-loop Input` — Specifies a linearization input point after a loop opening.
- `Open-loop Output` — Specifies a linearization output point before a loop opening.
- `Loop Transfer` — Specifies an output point before a loop opening followed by an input.
- `Input Perturbation` — Specifies an additive input to a signal.
- `Output Measurement` — Takes measurement at a signal.
- `Loop Break` — Specifies a loop opening.
- `Sensitivity` — Specifies an additive input followed by an output measurement.
- `Complementary Sensitivity` — Specifies an output followed by an additive input.

Note If you simulate the model without specifying an input or output, the software does not compute a linear system. Instead, you see a warning message at the MATLAB prompt.

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 on page 2-61

"Verify Model at Default Simulation Snapshot Time" on page 17-5

Click a signal in the model to select it

Enables signal selection in the Simulink model. Appears only when you click .

When this option appears, you also see the following changes:

- A new button.

  Use to add a selected signal as a linearization input or output in the **Linearization inputs/outputs** table. For more information, see **Linearization inputs/outputs** on page 19-124.

- 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 on page 2-61

"Verify Model at Default Simulation Snapshot Time" on page 17-5

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

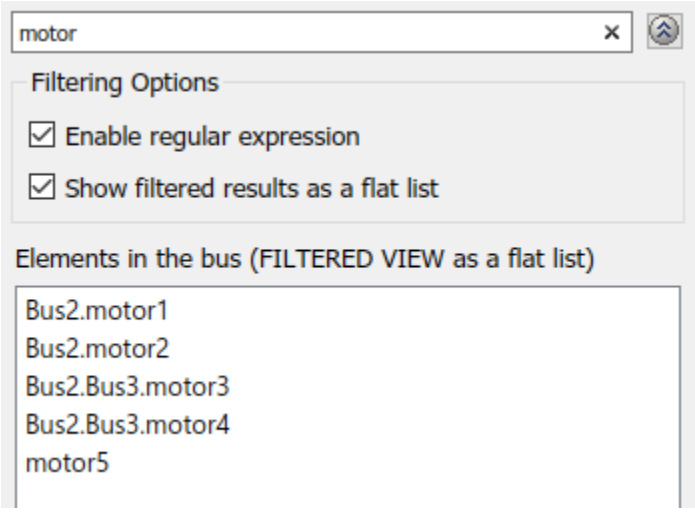
Disable use of MATLAB regular expressions for filtering signal names. Filtering treats the text you enter in the **Filter by name** edit box as a literal character vector.

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** on page 19-129.

Use when you:

- Know one or more times when the model is at a steady-state operating point
- Want to compute the linear systems at specific times

`External trigger`

Trigger-based simulation event. Specify the trigger type in **Trigger type** on page 19-130.

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: character vector
Value: `'SnapshotTimes'` | `'ExternalTrigger'`
Default: `'SnapshotTimes'`

See Also

Plot Linear Characteristics of Simulink Models During Simulation on page 2-61

"Verify Model at Default Simulation Snapshot Time" on page 17-5

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** on page 19-128 enables this parameter.

Command-Line Information

Parameter: `SnapshotTimes`
Type: character vector
Value: `0` | positive real number | vector of positive real numbers
Default: `0`

See Also

Plot Linear Characteristics of Simulink Models During Simulation on page 2-61

"Verify Model at Default Simulation Snapshot Time" on page 17-5

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** on page 19-128 enables this parameter.

Command-Line Information

Parameter: `TriggerType`
Type: character vector
Value: `'rising'` | `'falling'`
Default: `'rising'`

See Also

Plot Linear Characteristics of Simulink Models During Simulation on page 2-61

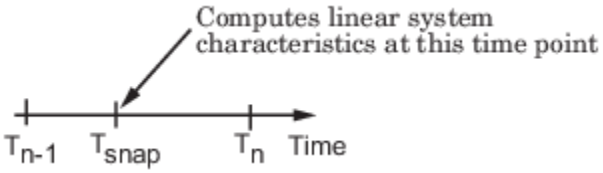
"Verify Model at Default Simulation Snapshot Time" on page 17-5

Enable zero-crossing detection

Enable zero-crossing detection to ensure that the software computes the linear system characteristics at the following simulation times:

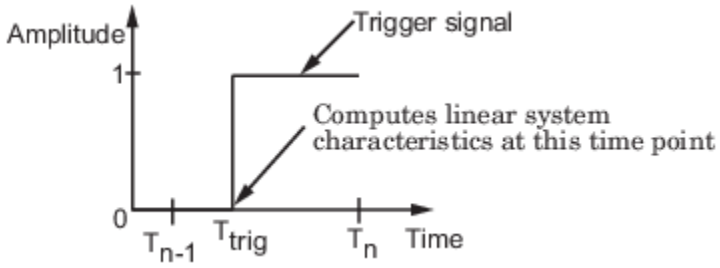
- The exact snapshot times, specified in **Snapshot times** on page 19-129.

  As shown in the following figure, when zero-crossing detection is enabled, the variable-step Simulink solver simulates the model at the snapshot time T_{snap}. T_{snap} may lie between the simulation time steps T_{n-1} and T_n which are automatically chosen by the solver.



- The exact times when an external trigger is detected, specified in **Trigger type** on page 19-130.

  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: character vector
Value: `'on'` | `'off'`
Default: `'on'`

See Also

Plot Linear Characteristics of Simulink Models During Simulation on page 2-61

“Verify Model at Default Simulation Snapshot Time” on page 17-5

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: character vector
Value: `'on'` | `'off'`
Default: `'off'`

See Also

Plot Linear Characteristics of Simulink Models During Simulation on page 2-61

"Verify Model at Default Simulation Snapshot Time" on page 17-5

Linear system sample time

Sample time of the linear system computed during simulation.

Use this parameter to:

- Compute a discrete-time system with a specific sample time from a continuous-time system
- Resample a discrete-time system with a different sample time
- Compute a continuous-time system from a discrete-time system

When computing discrete-time systems from continuous-time systems and vice-versa, the software uses the conversion method specified in **Sample time rate conversion method** on page 19-133.

Settings

Default: `auto`

`auto`. Computes the sample time as:

- 0, for continuous-time models.
- For models that have blocks with different sample times (multirate 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: character vector
Value: `'auto'` | Positive finite value | `'0'`
Default: `'auto'`

See Also

Plot Linear Characteristics of Simulink Models During Simulation on page 2-61

“Verify Model at Default Simulation Snapshot Time” on page 17-5

Sample time rate conversion method

Method for converting the sample time of single-rate or multirate models.

This parameter is used only when the value of **Linear system sample time** on page 19-132 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”.

This method usually performs better in the 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”.

This method usually performs 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”.

This method usually performs 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 a discrete-time system to a new faster sample time that is an integer multiple of the sample 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 a discrete-time system to a new faster sample time that is an integer multiple of the sample 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 a discrete-time system to a new faster sample time that is an integer multiple of the sample time of the original system.

Dependencies

Selecting either:

- `Tustin with Prewarping`
- `Upsampling when possible, Tustin with Prewarping otherwise`

enables **Prewarp frequency (rad/s)** on page 19-134.

Command-Line Information

Parameter: `RateConversionMethod`
Type: character vector
Value: `'zoh'` | `'tustin'` | `'prewarp'` | `'upsampling_zoh'` | `'upsampling_tustin'` | `'upsampling_prewarp'`
Default: `'zoh'`

See Also

Plot Linear Characteristics of Simulink Models During Simulation on page 2-61

"Verify Model at Default Simulation Snapshot Time" on page 17-5

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** on page 19-133 enables this parameter.

Command-Line Information

Parameter: `PreWarpFreq`
Type: character vector
Value: `10` | positive scalar value
Default: `10`

See Also

Plot Linear Characteristics of Simulink Models During Simulation on page 2-61

"Verify Model at Default Simulation Snapshot Time" on page 17-5

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: character vector
Value: `'on'` | `'off'`
Default: `'off'`

See Also

Plot Linear Characteristics of Simulink Models During Simulation on page 2-61

“Verify Model at Default Simulation Snapshot Time” on page 17-5

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: character vector
Value: `'on'` | `'off'`
Default: `'off'`

See Also

Plot Linear Characteristics of Simulink Models During Simulation on page 2-61

"Verify Model at Default Simulation Snapshot Time" on page 17-5

Include gain and phase margins in assertion

Check that the gain and phase margins are greater than the values specified in **Gain margin (dB) >** on page 19-137 and **Phase margin (deg) >** on page 19-138, 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** on page 19-146 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: character vector
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 on page 2-61

“Verify Model at Default Simulation Snapshot Time” on page 17-5

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** on page 19-136 and **Enable assertion** on page 19-146.
- 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: character vector
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 on page 2-61

“Verify Model at Default Simulation Snapshot Time” on page 17-5

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** on page 19-136 and **Enable assertion** on page 19-146.
- 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: character vector
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 on page 2-61

“Verify Model at Default Simulation Snapshot Time” on page 17-5

Include closed-loop peak gain in assertion

Check that the closed-loop peak gain is less than the value specified in **Closed-loop peak gain (dB) <** on page 19-139, 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** on page 19-146 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

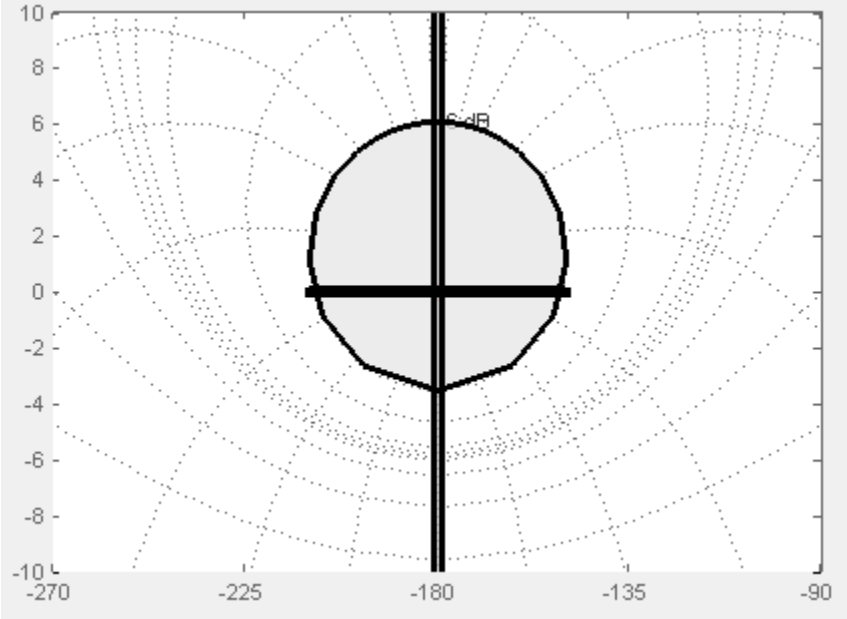
Check that the closed-loop peak gain satisfies the specified value, during simulation.

☐ 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: character vector
Value: `'on'` | `'off'`
Default: `'off'`

See Also

Plot Linear Characteristics of Simulink Models During Simulation on page 2-61

“Verify Model at Default Simulation Snapshot Time” on page 17-5

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** on page 19-138 and **Enable assertion** on page 19-146.
- 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: character vector
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 on page 2-61

"Verify Model at Default Simulation Snapshot Time" on page 17-5

Include open-loop gain-phase bound in assertion

Check that the Nichols response satisfies open-loop gain and phase bounds, specified in **Open-loop phases (deg)** on page 19-141 and **Open-loop gains (dB)** on page 19-142, during simulation. The software displays a warning if the Nichols response violates the bounds.

This parameter is used for assertion only if **Enable assertion** on page 19-146 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

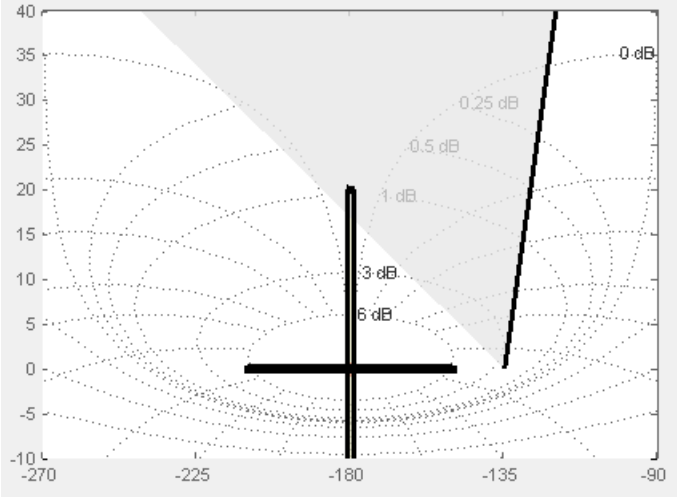
Check if the Nichols response satisfies the specified open-loop gain and phase bounds, during simulation.

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: character vector
Value: `'on'` | `'off'`
Default: `'off'`

Open-loop phases (deg)

Open-loop phases, in degrees.

Specify the corresponding open-loop gains in **Open-loop gains (dB)** on page 19-142.

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** on page 19-140 and **Enable assertion** on page 19-146.

- 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: character vector
Value: `[]` | positive or negative finite numbers | matrix of positive or negative finite numbers | cell array of matrices with finite numbers. Must be specified inside single quotes (`''`).
Default: `'[]'`

See Also

Plot Linear Characteristics of Simulink Models During Simulation on page 2-61

"Verify Model at Default Simulation Snapshot Time" on page 17-5

Open-loop gains (dB)

Open-loop gains, in decibels.

Specify the corresponding open-loop phases in **Open-loop phases (deg)** on page 19-141.

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** on page 19-140 and **Enable assertion** on page 19-146.
- 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: character vector
Value: `[]` | positive or negative number | matrix of positive or negative finite numbers | cell array of matrices with finite numbers. Must be specified inside single quotes (`''`).
Default: `'[]'`

See Also

Plot Linear Characteristics of Simulink Models During Simulation on page 2-61

"Verify Model at Default Simulation Snapshot Time" on page 17-5

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: character vector
Value: `'-1'` | `'+1'`
Default: `'-1'`

See Also

Plot Linear Characteristics of Simulink Models During Simulation on page 2-61

"Verify Model at Default Simulation Snapshot Time" on page 17-5

Save data to workspace

Save one or more linear systems to perform further linear analysis or control design.

The saved data is in a structure whose fields include:

- `time` — Simulation times at which the linear systems are computed.
- `values` — State-space model representing the linear system. If the linear system is computed at multiple simulation times, `values` is an array of state-space models.
- `operatingPoints` — Operating points corresponding to each linear system in `values`. This field exists only if **Save operating points for each linearization** is checked.

The location of the saved data structure depends upon the configuration of the Simulink model:

- If the Simulink model is not configured to save simulation output as a single object, the data structure is a variable in the MATLAB workspace.
- If the Simulink model is configured to save simulation output as a single object, the data structure is a field in the `Simulink.SimulationOutput` object that contains the logged simulation data.

  To configure your model to save simulation output in a single object, in the Simulink editor, on the **Modeling** tab, click **Model Settings**. Then, in the Configuration Parameters dialog box, select the **Single simulation output** parameter.

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** on page 19-144.

Command-Line Information

Parameter: `SaveToWorkspace`
Type: character vector
Value: `'on'` | `'off'`
Default: `'off'`

See Also

Plot Linear Characteristics of Simulink Models During Simulation on page 2-61

“Verify Model at Default Simulation Snapshot Time” on page 17-5

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 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

Character vector.

Dependencies

Save data to workspace on page 19-143 enables this parameter.

Command-Line Information

Parameter: `SaveName`
Type: character vector
Value: `sys` | any character vector. Must be specified inside single quotes (`''`).
Default: `'sys'`

See Also

Plot Linear Characteristics of Simulink Models During Simulation on page 2-61

“Verify Model at Default Simulation Snapshot Time” on page 17-5

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 on page 19-143 enables this parameter.

Command-Line Information

Parameter: `SaveOperatingPoint`
Type: character vector
Value: `'on'` | `'off'`

Default: `'off'`

See Also

Plot Linear Characteristics of Simulink Models During Simulation on page 2-61

“Verify Model at Default Simulation Snapshot Time” on page 17-5

Enable assertion

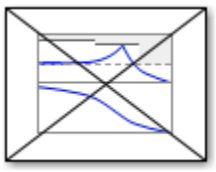
Enable the block to check that bounds specified and included for assertion in the **Bounds** tab are satisfied during simulation. Assertion fails if a bound is not satisfied. A warning, reporting the assertion failure, appears at the MATLAB prompt.

If assertion fails, you can optionally specify that the block:

- Execute a MATLAB expression, specified in **Simulation callback when assertion fails (optional)** on page 19-147.
- Stop the simulation and bring that block into focus, by selecting **Stop simulation when assertion fails** on page 19-147.

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; that is, the block no longer checks that specified bounds are satisfied. The block icon also updates to indicate that assertion is disabled.



In the Simulink model, in the Configuration Parameters dialog box, the **Model Verification block enabling** parameter lets you enable or disable all model verification blocks in a model, regardless of the setting of this option in the block.

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: character vector
Value: `'on'` | `'off'`
Default: `'on'`

See Also

Plot Linear Characteristics of Simulink Models During Simulation on page 2-61

“Verify Model at Default Simulation Snapshot Time” on page 17-5

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 on page 19-146 enables this parameter.

Command-Line Information

Parameter: `callback`
Type: character vector
Value: `''` | MATLAB expression
Default: `''`

See Also

Plot Linear Characteristics of Simulink Models During Simulation on page 2-61

“Verify Model at Default Simulation Snapshot Time” on page 17-5

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 on page 19-146 enables this parameter.

Command-Line Information

Parameter: `stopWhenAssertionFail`
Type: character vector
Value: `'on'` | `'off'`
Default: `'off'`

See Also

Plot Linear Characteristics of Simulink Models During Simulation on page 2-61

"Verify Model at Default Simulation Snapshot Time" on page 17-5

Output assertion signal

Output a Boolean signal that, at each time step, is:

- True (1) if assertion succeeds; that is, all bounds are satisfied
- False (1) if assertion fails; that is, a bound is violated.

The output signal data type is Boolean only if, in the Simulink model, in the Configuration Parameters dialog box, the **Implement logic signals as Boolean data** parameter is selected. Otherwise, the data type of the output signal is double.

Selecting this parameter adds an output port to the block that you can connect to any block in the model.

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 “Verify Model Using Simulink Control Design and Simulink Verification Blocks” on page 17-20.

Command-Line Information

Parameter: `export`
Type: character vector
Value: `'on'` | `'off'`
Default: `'off'`

See Also

Plot Linear Characteristics of Simulink Models During Simulation on page 2-61

“Verify Model at Default Simulation Snapshot Time” on page 17-5

Show plot on block open

Open the plot window instead of the Block Parameters dialog box when you double-click the block in the Simulink model.

Use this parameter if you prefer to open and perform tasks, such as adding or modifying bounds, in the plot window instead of the Block Parameters dialog box. If you want to access the block parameters from the plot window, select **Edit** or click .

For more information on the plot, see “Show Plot” on page 19-0 .

Settings

Default: Off

On

Open the plot window when you double-click the block.

Off

Open the Block Parameters dialog box when you double-click the block.

Command-Line Information

Parameter: `LaunchViewOnOpen`
Type: character vector
Value: `'on'` | `'off'`
Default: `'off'`

See Also

Plot Linear Characteristics of Simulink Models During Simulation on page 2-61

“Verify Model at Default Simulation Snapshot Time” on page 17-5

Show Plot

Open the plot window.

Use the plot to view:

- System characteristics and signals computed during simulation

  You must click this button before you simulate the model to view the system characteristics or signal.

  You can display additional characteristics, such as the peak response time, by right-clicking the plot and selecting **Characteristics**.
- Bounds

  You can specify bounds in the **Bounds** tab of the Block Parameters dialog box or right-click the plot and select **Bounds > New Bound**. For more information on the types of bounds you can specify, see the individual reference pages.

  You can modify bounds by dragging the bound segment or by right-clicking the plot and selecting **Bounds > Edit Bound**. Before you simulate the model, click **Update Block** to update the bound value in the block parameters.

Typical tasks that you perform in the plot window include:

- Opening the Block Parameters dialog box by clicking or selecting **Edit**.
- Finding the block that the plot window corresponds to by clicking or selecting **View > Highlight Simulink Block**. This action makes the model window active and highlights the block.
- Simulating the model by clicking . This action also linearizes the portion of the model between the specified linearization input and output.
- Adding a legend on the linear system characteristic plot by clicking .

Note To optimize the model response to meet design requirements specified in the **Bounds** tab, open the **Response Optimizer** by selecting **Tools > Response Optimization** in the plot window. This option is only available if you have Simulink Design Optimization software installed.

Response Optimization

Open the **Response Optimizer** to optimize the model response to meet design requirements specified in the **Bounds** tab.

This button is available only if you have Simulink Design Optimization software installed.

See Also

- “Design Optimization to Meet Step Response Requirements (GUI)” (Simulink Design Optimization)
- “Design Optimization to Meet Time-Domain and Frequency-Domain Requirements (GUI)” (Simulink Design Optimization)

See Also

Check Nichols Characteristics

Tutorials

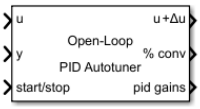
- “Visualize Bode Response of Simulink Model During Simulation” on page 2-61
- “Visualize Linear System at Multiple Simulation Snapshots” on page 2-85
- “Visualize Linear System of a Continuous-Time Model Discretized During Simulation” on page 2-91
- “Plot Linear System Characteristics of a Chemical Reactor” on page 2-95

Introduced in R2010b

Open-Loop PID Autotuner

Automatically tune PID gains based on plant frequency responses estimated from open-loop experiment in real time

Library: Simulink Control Design



Description

The Open-Loop PID Autotuner block lets you tune a PID controller in real time against a physical plant. The block can tune a PID controller to achieve a specified bandwidth and phase margin without a parametric plant model or an initial controller design. If you have a code-generation product such as Simulink Coder, you can generate code that implements the tuning algorithm on hardware, letting you tune in real time with or without using Simulink to manage the autotuning process.

If you have a plant model in Simulink, you can also use the block to obtain an initial PID design. Doing so lets you preview plant response and adjust the settings for PID autotuning before tuning the controller in real time.

To achieve model-free tuning, the Open-Loop PID Autotuner block:

1. Injects a test signal into the plant at the nominal operating point to collect plant input-output data and estimate frequency response in real time. The test signal is a combination of sine and step perturbation signals added on top of the nominal plant input measured when the experiment starts. If the plant is part of a feedback loop, the block opens the loop during the experiment.
2. At the end of the experiment, tunes PID controller parameters based on estimated plant frequency responses near the open-loop bandwidth.
3. Updates a PID Controller block or a custom PID controller with the tuned parameters, allowing you to validate closed-loop performance in real time.

Because the block performs an open-loop estimation experiment, do not use this block with an unstable plant or a plant with multiple integrators.

To use the algorithm, you do not need an initial PID controller design. However, you must have some way to get the plant to a nominal operating point for the frequency-response estimation experiment. If you have an initial controller design, you can use the Closed-Loop PID Autotuner. For a comparison of closed-loop and open-loop PID autotuning, see “When to Use PID Autotuning” on page 8-2.

The block supports code generation with Simulink Coder, Embedded Coder, and Simulink PLC Coder. It does not support code generation with HDL Coder.

For more information about using the Open-Loop PID Autotuner block, see:

- “PID Autotuning for a Plant Modeled in Simulink” on page 8-7
- “PID Autotuning in Real Time” on page 8-13

For more general information about PID autotuning and a comparison of the closed-loop and open-loop approaches, see “When to Use PID Autotuning” on page 8-2.

Ports

Input

u — Signal from controller

scalar

Insert the block into your system such that this port accepts a control signal from a source. Typically, this port accepts the signal from the PID controller in your system.

Data Types: `single` | `double`

y — Plant output

scalar

Connect this port to the plant output.

Data Types: `single` | `double`

`start/stop` — Start and stop the autotuning experiment

scalar

To start and stop the autotuning process, provide a signal at the `start/stop` port. When the value of the signal changes from:

- Negative or zero to positive, the experiment starts
- Positive to negative or zero, the experiment stops

When the experiment is not running, the block passes signals unchanged from **u** to **u+Δu**. In this state, the block has no impact on plant or controller behavior.

Typically, you can use a signal that changes from 0 to 1 to start the experiment, and from 1 to 0 to stop it. Some points to consider when configuring the **start/stop** signal include:

- Start the experiment when the plant is at the desired equilibrium operating point. Use the initial controller to drive the plant to the operating point. If you have no initial controller (open-loop tuning only) you can use a source block connected to **u** to drive the plant to the operating point.
- Avoid any load disturbance to the plant during the experiment. Load disturbance can distort the plant output and reduce the accuracy of the frequency-response estimation.
- Let the experiment run long enough for the algorithm to collect sufficient data for a good estimate at all frequencies it probes. There are two ways to determine when to stop the experiment:
  - Determine the experiment duration in advance. A conservative estimate for the experiment duration is $200/\omega_c$ for closed-loop tuning, or $100/\omega_c$ for open-loop tuning, where ω_c is your target bandwidth.
  - Observe the signal at the `% conv` output, and stop the experiment when the signal stabilizes near 100%.
- When you stop the experiment, the block computes tuned PID gains and updates the signal at the `pid gains` port.

You can configure any logic appropriate for your application to control the start and stop times of the experiment.

Data Types: `single` | `double`

`bandwidth` — Target bandwidth for tuning

scalar

Supply a value for the `Target bandwidth (rad/sec)` parameter. See that parameter for details.

Dependencies

To enable this port, in the Tuning tab, next to `Target bandwidth (rad/sec)`, select **Use external source**.

Data Types: `single` | `double`

`target PM` — Target phase margin for tuning

scalar

Supply a value for the `Target phase margin (degrees)` parameter. See that parameter for details.

Dependencies

To enable this port, in the Tuning tab, next to `Target phase margin (degrees)`, select **Use external source**.

Data Types: `single` | `double`

`sine Amp` — Amplitudes of injected sinusoidal signals

scalar | vector

Supply a value for the `Sine Amplitudes` parameter. See that parameter for details.

Dependencies

To enable this port, in the Experiment tab, next to `Sine Amplitudes`, select **Use external source**.

Data Types: `single` | `double`

`step Amp` — Amplitude of injected step signal

scalar

Supply a value for the `Step Amplitude` parameter. See that parameter for details.

Dependencies

To enable this port, in the Experiment tab, next to `Step Amplitudes`, select **Use external source**.

Data Types: `single` | `double`

Output

u+Δu — Signal for plant input

scalar

Insert the block into your system such that this port feeds the input signal to your plant.

- When the experiment is running (`start/stop` positive), the block injects test signals into the plant at this port. The test signal is the value at u when the experiment begins plus the experiment perturbation. If you have any saturation or rate limit protecting the plant, feed the signal from **u +Δu** into it.

- When the experiment is not running (`start/stop` zero or negative), the block passes signals unchanged from **u** to **u+Δu**.

Data Types: `single` | `double`

`% conv` — Convergence of FRD estimation during experiment

scalar

When the experiment is running (`start/stop` positive), the block injects test signals into the plant and measures the plant response at `y`. It uses these signals to estimate the frequency response of the plant at several frequencies around the target bandwidth for tuning. `% conv` indicates how close to completion the estimation of the plant frequency response is. Typically, this value quickly rises to about 90% after the experiment begins, and then gradually converges to a higher value. Stop the experiment when it levels off near 100%.

Data Types: `single` | `double`

`pid gains` — Tuned PID coefficients

bus

This 4-element bus signal contains the tuned PID gains P, I, D, and the filter coefficient N. These values correspond to the `P`, `I`, `D`, and `N` parameters in the expressions given in the `Form` parameter. Initially, the values are 0, 0, 0, and 100, respectively. The block updates the values when the experiment ends. This bus signal always has four elements, even if you are not tuning a PIDF controller.

If you have a PID controller associated with the block, you can update that controller with these values after the experiment ends. To do so, in the Block tab, click **Update PID Block**.

Data Types: `single` | `double`

`estimated PM` — Estimated phase margin with tuned controller

scalar

This port outputs the estimated phase margin achieved by the tuned controller, in degrees. The block updates this value when the tuning experiment ends. The estimated phase margin is calculated from the angle of $G(j\omega_c)C(j\omega_c)$, where G is the estimated plant, C is the tuned controller, and ω_c is the crossover frequency (bandwidth). The estimated phase margin might differ from the target phase margin specified by the `Target phase margin (degrees)` parameter. It is an indicator of the robustness and stability achieved by the tuned system.

- Typically, the estimated phase margin is near the target phase margin. In general, the larger the value, the more robust is the tuned system, and the less overshoot there is.
- A negative phase margin indicates that the closed-loop system might be unstable.

Dependencies

To enable this port, in the Tuning tab, select **Output estimated phase margin achieved by tuned controller**.

`frd` — Estimated frequency response

vector

This port outputs the frequency-response data estimated by the experiment. Initially, the value at `frd` is [0, 0, 0, 0]. During the experiment, the block injects signals at frequencies [1/3, 1, 3, 10]ω_c, where

ω_c is the target bandwidth. At each sample time during the experiment, the block updates `frd` with a vector containing the complex frequency response at each of these frequencies, respectively. You can use the progress of the response as an alternative to `% conv` to examine the convergence of the estimation. When the experiment stops, the block updates `frd` with the final estimated frequency response used for computing the PID gains.

Dependencies

To enable this port, in the Experiment tab, select **Plant frequency responses near bandwidth**.

`dcgain` — Estimated DC gain of plant

scalar

If you select **Estimate DC gain with step signal** in the Experiment tab, the block estimates the DC gain of the plant by including a step signal in the injected perturbation. When the experiment stops, the block updates this port with the estimated DC gain value.

Dependencies

To enable this port, in the Experiment tab, select **Plant DC Gain**.

`nominal` — Plant input and output at nominal operating point

vector

This port outputs a vector containing the plant input (**u+Δu**) and plant output (**y**) when the experiment begins. These values are the plant input and output at the nominal operating point at which the block performs the experiment.

Dependencies

To enable this port, in the Experiment tab, select **Plant nominal input and output**.

Parameters

Tuning Tab

Type — PID controller actions

`PI` (default) | `PID` | `PIDF` | ...

Specify the type of the PID controller in your system. The controller type indicates what actions are present in the controller. The following controller types are available for PID autotuning:

- `P` — Proportional only
- `I` — Integral only
- `PI` — Proportional and integral
- `PD` — Proportional and derivative
- `PDF` — Proportional and derivative with derivative filter
- `PID` — Proportional, integral, and derivative
- `PIDF` — Proportional, integral, and derivative with derivative filter

When you update a PID Controller block or custom PID controller with tuned parameter values, make sure the controller type matches.

Tunable: Yes

Programmatic Use
Block Parameter: `PIDType`
Type: character vector
Values: `'P'` | `'I'` | `'PI'` | `'PD'` | `'PDF'` | `'PID'` | `'PIDF'`
Default: `'PI'`

`Form` — PID controller form

`Parallel` (default) | `Ideal`

Specify the controller form. The controller form determines the interpretation of the PID coefficients P, I, D, and N.

- `Parallel` — In `Parallel` form, the transfer function of a discrete-time PIDF controller is:

  $$C = P + IF_i(z) + D\left[\frac{N}{1 + NF_d(z)}\right],$$

  where $F_i(z)$ and $F_d(z)$ are the integrator and filter formulas (see `Integrator method` and `Filter method`). The transfer function of a continuous-time parallel-form PIDF controller is:

  $$C = P + I\left(\frac{1}{s}\right) + D\left(\frac{Ns}{s + N}\right).$$

  Other controller actions amount to setting P, I, or D to zero.
- `Ideal` — In `Ideal` form, the transfer function of a discrete-time PIDF controller is:

  $$C = P\left[1 + IF_i(z) + D\left(\frac{N}{1 + NF_d(z)}\right)\right].$$

  The transfer function of a continuous-time ideal-form PIDF controller is:

  $$C = P\left[1 + I\left(\frac{1}{s}\right) + D\left(\frac{Ns}{s + N}\right)\right].$$

  Other controller actions amount to setting D to zero or setting, I to `Inf`. (In ideal form, the controller must have proportional action.)

When you update a PID Controller block or custom PID controller with tuned parameter values, make sure the controller form matches.

Tunable: Yes

Programmatic Use
Block Parameter: `PIDForm`
Type: character vector
Values: `'Parallel'` | `'Ideal'`
Default: `'Parallel'`

`Time Domain` — PID controller time domain

discrete-time (default) | continuous-time

Specify whether your PID controller is a discrete-time or continuous-time controller.

- For discrete time, you must specify the sample time of your PID controller using the **Controller sample time (sec)** parameter.

- For continuous time, you must also specify a sample time for the PID autotuning experiment using the **Experiment sample time (sec)** parameter.

Programmatic Use

Block Parameter: `TimeDomain`
Type: character vector
Values: `'discrete-time'` | `'continuous-time'`
Default: `'discrete-time'`

`Controller sample time (sec)` — Sample time of PID controller

0.1 (default) | positive scalar | –1

Specify the sample time of your PID controller in seconds. This value also sets the sample time for the experiment performed by the block.

To perform PID tuning, the block measures frequency-response information up to a frequency of 10 times the target bandwidth. To ensure that this frequency is less than the Nyquist frequency, the target bandwidth, ω_c, must satisfy $\omega_c T_s \leq 0.3$, where T_s ω_c is the controller sample time that you specify with the `Controller sample time (sec)` parameter.

When you update a PID Controller block or custom PID controller with tuned parameter values, make sure the controller sample time matches.

Tips

If you want to run the deployed block with different sample times in your application, set this parameter to –1 and put the block in a Triggered Subsystem. Then, trigger the subsystem at the desired sample time. If you do not plan to change the sample time after deployment, specify a fixed and finite sample time.

Dependencies

To enable this parameter, set **Time Domain** to `discrete-time`.

Programmatic Use

Block Parameter: `DiscreteTs`
Type: scalar
Value positive scalar | –1
Default: 0.1

`Experiment sample time (sec)` — Sample time for experiment

0.02 (default) | positive scalar

Even when you tune a continuous-time controller, you must specify a sample time for the experiment performed by the block. In general, continuous-time controller tuning is not recommended for PID autotuning against a physical plant. If you want to tune in continuous time against a Simulink model of the plant, use a fast experiment sample time, such as $0.02/\omega_c$.

Dependencies

This parameter is enabled when the **Time Domain** is `continuous-time`.

Programmatic Use

Block Parameter: `ContinuousTs`
Type: positive scalar
Default: 0.02

`Integrator method` — Discrete integration formula for integrator term

`Forward Euler` (default) | `Backward Euler` | `Trapezoidal`

Specify the discrete integration formula for the integrator term in your controller. In discrete time, the PID controller transfer function assumed by the block is:

$$C = P + IF_i(z) + D\left[\frac{N}{1 + NF_d(z)}\right],$$

in parallel form, or in ideal form,

$$C = P\left[1 + IF_i(z) + D\left(\frac{N}{1 + NF_d(z)}\right)\right].$$

For a controller sample time T_s, the `Integrator method` parameter determines the formula F_i as follows:

Integrator method	F_i
`Forward Euler`	$\frac{T_s}{z-1}$
`Backward Euler`	$\frac{T_s z}{z-1}$
`Trapezoidal`	$\frac{T_s}{2}\frac{z+1}{z-1}$

For more information about the relative advantages of each method, see the Discrete PID Controller block reference page.

When you update a PID Controller block or custom PID controller with tuned parameter values, make sure the integrator method matches.

Tunable: Yes

Dependencies

This parameter is enabled when the **Time Domain** is `discrete-time` and the controller includes integral action.

Programmatic Use

Block Parameter: `IntegratorFormula`
Type: character vector
Values: `'Forward Euler'` | `'Backward Euler'` | `'Trapezoidal'`
Default: `'Forward Euler'`

`Filter method` — Discrete integration formula for derivative filter term

`Forward Euler` (default) | `Backward Euler` | `Trapezoidal`

Specify the discrete integration formula for the derivative filter term in your controller. In discrete time, the PID controller transfer function assumed by the block is:

$$C = P + IF_i(z) + D\left[\frac{N}{1 + NF_d(z)}\right],$$

in parallel form, or in ideal form,

$$C = P\left[1 + IF_i(z) + D\left(\frac{N}{1 + NF_d(z)}\right)\right].$$

For a controller sample time T_s, the `Filter method` parameter determines the formula F_d as follows:

Filter method	F_d
`Forward Euler`	$\frac{T_s}{z-1}$
`Backward Euler`	$\frac{T_s z}{z-1}$
`Trapezoidal`	$\frac{T_s}{2}\frac{z+1}{z-1}$

For more information about the relative advantages of each method, see the Discrete PID Controller block reference page.

When you update a PID Controller block or custom PID controller with tuned parameter values, make sure the filter method matches.

Tunable: Yes

Dependencies

This parameter is enabled when the **Time Domain** is `discrete-time` and the controller includes a derivative filter term.

Programmatic Use

Block Parameter: `FilterFormula`
Type: character vector
Values: `'Forward Euler'` | `'Backward Euler'` | `'Trapezoidal'`
Default: `'Forward Euler'`

`Target bandwidth (rad/sec)` — Target crossover frequency of tuned response

1 (default) | positive scalar

The target bandwidth, specified in rad/sec, is the target value for the 0-dB gain crossover frequency of the tuned open-loop response CP, where P is the plant response, and C is the controller response. This crossover frequency roughly sets the control bandwidth. For a rise-time τ seconds, a good guess for the target bandwidth is $2/\tau$ rad/sec.

To perform PID tuning, the autotuner block measures frequency-response information up to a frequency of 10 times the target bandwidth. To ensure that this frequency is less than the Nyquist frequency, the target bandwidth, ω_c, must satisfy $\omega_c T_s \leq 0.3$, where T_s is the controller sample time that you specify with the **Controller sample time (sec)** parameter. Because of this condition, the fastest rise time you can enforce for tuning is about $6.67T_s$. If this rise time does not meet your design goals, consider reducing T_s.

To provide the target bandwidth via an input port, select **Use external source**.

Programmatic Use

Block Parameter: `Bandwidth`
Type: positive scalar

Default: 1

`Target phase margin (degrees)` — Target minimum phase margin of open-loop response

60 (default) | scalar in range 0–90

Specify a target minimum phase margin for the tuned open-loop response at the crossover frequency. The target phase margin reflects desired robustness of the tuned system. Typically, choose a value in the range of about 45°–60°. In general, higher phase margin improves overshoot, but can limit response speed. The default value, 60°, tends to balance performance and robustness, yielding about 5–10% overshoot, depending on the characteristics of your plant.

To provide the target phase margin via an input port, select **Use external source**.

Tunable: Yes

Programmatic Use

Block Parameter: `TargetPM`
Type: scalar
Values: 0–90
Default: 60

Experiment Tab

`Sine Amplitudes` — Amplitude of sinusoidal perturbations

1 (default) | scalar | vector of length 4

During the tuning experiment, the block injects a sinusoidal signal into the plant at the frequencies [1/3, 1, 3, 10]ω_c , where ω_c is the target bandwidth for tuning. Use **Sine Amplitudes** to specify the amplitude of each of these injected signals. Specify a:

- Scalar value to inject the same amplitude at each frequency
- Vector of length 4 to specify a different amplitude at each of [1/3, 1, 3, 10]ω_c

In a typical plant with typical target bandwidth, the magnitudes of the plant responses at the experiment frequencies do not vary widely. In such cases, you can use a scalar value to apply the same magnitude perturbation at all frequencies. However, if you know that the response decays sharply over the frequency range, consider decreasing the amplitude of the lower-frequency inputs and increasing the amplitude of the higher-frequency inputs. It is numerically better for the estimation experiment when all the plant responses have comparable magnitudes.

The perturbation amplitudes must be:

- Large enough that the perturbation overcomes any deadband in the plant actuator and generates a response above the noise level
- Small enough to keep the plant running within the approximately linear region near the nominal operating point, and to avoid saturating the plant input or output

In the experiment, the sinusoidal signals are superimposed (with the step perturbation, if any, in the case of open-loop tuning). Thus, the perturbation can be at least as large as the sum of all amplitudes. Therefore, to obtain appropriate values for the amplitudes, consider:

- Actuator limits. Make sure that the largest possible perturbation is within the range of your plant actuator. Saturating the actuator can introduce errors into the estimated frequency response.
- How much the plant response changes in response to a given actuator input at the nominal operating point for tuning. For instance, suppose that you are tuning a PID controller used in

engine-speed control. You have determined that at frequencies around the target bandwidth, a 1° change in throttle angle causes a change of about 200 rpm in the engine speed. Suppose further that to preserve linear performance the speed must not deviate by more than 100 rpm from the nominal operating point. In this case, choose amplitudes to ensure that the perturbation signal is no greater than 0.5 (assuming that value is within actuator limits).

To provide the sine amplitudes via an input port, select **Use external source**.

Tunable: Yes

Programmatic Use

Block Parameter: `AmpSine`
Type: scalar, vector of length 4
Default: 1

`Estimate DC gain with step signal` — Inject step signal into plant

`on` (default) | `off`

When this option is selected, the experiment includes an estimation of the plant DC gain. The block performs this estimation by injecting a step signal into the plant.

Caution If your plant has a single integrator, clear this option. For plants with multiple integrators or unstable poles, do not use the Open-Loop PID Autotuner block.

Tunable: Yes

Programmatic Use

Block Parameter: `EstimateDCGain`
Type: character vector
Values: `'off'` | `'on'`
Default: `'on'`

`Step Amplitude` — Amplitude of step perturbation

1 (default) | scalar

If **Estimate DC gain with step signal** is selected, the block estimates the DC gain by injecting a step signal into the plant. Use this parameter to set the amplitude of the signal. The considerations for choosing a step amplitude are the same as the considerations for specifying **Sine Amplitudes**.

To provide the step amplitude via an input port, select **Use external source**.

Tunable: Yes

Dependencies

This parameter is enabled when **Estimate DC gain with step signal** is selected.

Programmatic Use

Block Parameter: `AmpStep`
Type: scalar
Default: 1

Block Tab

`Reduce memory and avoid task overrun (external mode only)` — Deploy tuning algorithm only
`off` (default) | `on`

The block contains two modules, one that performs the real-time frequency-response estimation, and one that uses the resulting estimated response to tune the PID gains. When you run a Simulink model containing the block in the external simulation mode, by default both modules are deployed. You can save memory on the target hardware by deploying the estimation module only (see “Control Real-Time PID Autotuning in Simulink” on page 8-20). In this case, the tuning algorithm runs on the Simulink host computer instead of the target hardware. When this option is selected, the deployed algorithm uses about a third as much memory as when the option is cleared.

The PID gain calculation demands more computational load than the frequency-response estimation. For fast controller sample times, some hardware might not finish the gain calculation within one execution cycle. Therefore, when using hardware with limited computing power, selecting this option lets you tune a PID controller with a fast sample time.

Additionally, when you enable this option, there can be a delay of several sampling periods between when the tuning experiment ends and when the new PID gains arrive at the **pid gains** output port. Before pushing gains to the controller, first confirm the change at the **pid gains** output port instead of using **start/stop** signal as the trigger for the update.

If you intend to deploy the block and perform PID tuning without using external simulation mode, do not select this option.

Caution When you use this option, the model must be configured such that numeric block parameters are tunable in generated code, not inlined. To specify tunable parameters:

- In the model editor: In **Configuration Parameters**, in **Code Generation** > **Optimization**, set **Default parameter behavior** to `Tunable`.
- At the command line: Use `set_param(mdl,'DefaultParameterBehavior','Tunable')`.

Programmatic Use

Block Parameter: `DeployTuningModule`
Type: character vector
Values: `'off'` | `'on'`
Default: `'off'`

`Configure block for PLC Coder` — Configure block for code generation with Simulink PLC Coder
off (default) | on

Select this parameter if you are using Simulink PLC Coder to generate code for the autotuner block. Clear the parameter for code generation with any other MathWorks code-generation product.

Selecting this parameter affects internal block configuration only, for compatibility with Simulink PLC Coder. The parameter has no operative effect on generated code.

`Data Type` — Floating point precision
`double` (default) | `single`

Specify the floating-point precision based on simulation environment or hardware requirements.

Programmatic Use
Block Parameter: `BlockDataType`
Type: character vector
Values: `'double'` | `'single'`
Default: `'double'`

`Clicking "Update PID Block" writes tuned gains to the PID block connected to "u" port` — Automatically detect target for writing tuned PID coefficients

`on` (default) | `off`

Under some conditions, the autotuner block can write tuned gains to a standard or custom PID controller block. To indicate that the target PID controller is the block connected to the **u** port of the autotuner block, select this option. To specify a PID controller that is not connected to **u**, clear this option.

To write tuned gains from the autotuner block to a PID controller anywhere in the model, the target block must be either:

- A PID Controller or Discrete PID Controller block.
- A masked subsystem in which the PID coefficients are mask parameters named `P`, `I`, `D`, and `N`, or whatever subset of these parameters exist in the your controller. For example, if you use a custom PI controller, then you only need mask parameters `P` and `I`.

`Specify PID block path` — Target PID controller block for writing tuned coefficients

`[ ]` (default) | block path

Under some conditions, the autotuner block can write tuned gains to a standard or custom PID controller block. Use this parameter to specify the path of the target PID controller.

To write tuned gains from the autotuner block to a PID controller anywhere in the model, the target block must be either:

- A PID Controller or Discrete PID Controller block.
- A masked subsystem in which the PID coefficients are mask parameters named `P`, `I`, `D`, and `N`, or whatever subset of these parameters exist in your controller

Dependencies

This parameter is enabled when **Clicking "Update PID Block" writes tuned gains to the PID block connected to "u" port** is selected.

`Update PID Block` — Write tuned PID gains to target controller block

button

The block does not automatically push the tuned gains to the target PID block. If your PID controller block meets the criteria described in the `Specify PID block path` parameter description, after tuning, click this button to transfer the tuned gains to the block.

You can update the PID block while the simulation is running, including when running in external mode. Doing so is useful for immediately validating tuned PID gains. At any time during simulation, you can change parameters, start the experiment again, and push the new tuned gains to the PID block. You can then continue to run the model and observe the behavior of your plant.

`Export to MATLAB` — Send experiment and tuning results to MATLAB workspace
button

When you click this button, the block creates a structure in the MATLAB workspace containing the experiment and tuning results. This structure, `OnlinePIDTuningResult`, contains the following fields:

- `P`, `I`, `D`, `N` — Tuned PID gains. The structure contains whichever of these fields are necessary for the controller type you are tuning. For instance, if you are tuning a PI controller, the structure contains `P` and `I`, but not `D` and `N`.
- `TargetBandwidth` — The value you specified in the **Target bandwidth (rad/sec)** parameter of the block.
- `TargetPhaseMargin` — The value you specified in the **Target phase margin (degrees)** parameter of the block.
- `EstimatedPhaseMargin` — Estimated phase margin achieved by the tuned system.
- `Controller` — The tuned PID controller, returned as a `pid` (for parallel form) or `pidstd` (for ideal form) model object.
- `Plant` — The estimated plant, returned as an `frd` model object. This `frd` contains the response data obtained at the experiment frequencies [1/3, 1, 3, 10]ω_c.
- `PlantNominal` — The plant input and output at the nominal operating point when the experiment begins, specified as a structure having fields `u` (input) and `y` (output).
- `PlantDCGain` — The estimated DC gain of the system in absolute units, if **Estimate DC gain with step signal** is selected during tuning.

You can export to the MATLAB workspace while the simulation is running, including when running in external mode.

Extended Capabilities

C/C++ Code Generation
Generate C and C++ code using Simulink® Coder™.

PLC Code Generation
Generate Structured Text code using Simulink® PLC Coder™.

See Also

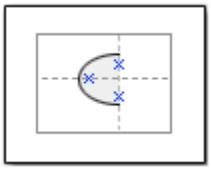
Closed-Loop PID Autotuner

Topics
“PID Autotuning for a Plant Modeled in Simulink” on page 8-7
“PID Autotuning in Real Time” on page 8-13
“When to Use PID Autotuning” on page 8-2
“How PID Autotuning Works” on page 8-5

Introduced in R2017b

Pole-Zero Plot

Pole-zero plot of linear system approximated from nonlinear Simulink model



Library

Simulink Control Design

Description

This block is the 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 in the MATLAB Command Window. 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 any 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 19-168. • "Click a signal in the model to select it" on page 19-170.
	Specify settings.	In **Linearizations** tab: • "Linearize on" on page 19-172. • "Snapshot times" on page 19-173. • "Trigger type" on page 19-174.
	Specify algorithm options.	In **Algorithm Options** of **Linearizations** tab: • "Enable zero-crossing detection" on page 19-174. • "Use exact delays" on page 19-175. • "Linear system sample time" on page 19-176. • "Sample time rate conversion method" on page 19-177. • "Prewarp frequency (rad/s)" on page 19-178.
	Specify labels for linear system I/Os and state names.	In **Labels** of **Linearizations** tab: • "Use full block names" on page 19-178. • "Use bus signal names" on page 19-179.
Plot the linear system.		**Show Plot** on page 19-194

Task	Parameters
(Optional) Specify bounds on pole-zero for assertion.	In **Bounds** tab: • Include settling time bound in assertion on page 19-180. • Include percent overshoot bound in assertion on page 19-182. • Include damping ratio bound in assertion on page 19-184. • Include natural frequency bound in assertion on page 19-186.
Specify assertion options (only when you specify bounds on the linear system).	In **Assertion** tab: • “Enable assertion” on page 19-190. • “Simulation callback when assertion fails (optional)” on page 19-191. • “Stop simulation when assertion fails” on page 19-192. • “Output assertion signal” on page 19-193.
Save linear system to MATLAB workspace.	“Save data to workspace” on page 19-188 in **Logging** tab.
Display plot window instead of block parameters dialog box on double-clicking the block.	“Show plot on block open” on page 19-193.

Linearization inputs/outputs

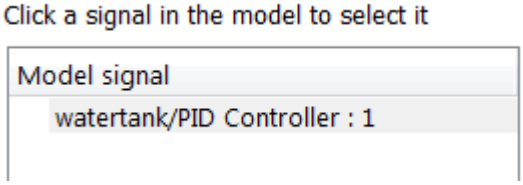
Linearization inputs and outputs that define the portion of a nonlinear Simulink model to linearize.

If you have defined the linearization input and output in the Simulink model, the block automatically detects these points and displays them in the **Linearization inputs/outputs** area. Click at any time to update the **Linearization inputs/outputs** table with I/Os from the model. To add other analysis points:

1. Click .

   The dialog box expands to display a **Click a signal in the model to select it** on page 19-170 area and a new button.
2. Select one or more signals in the Simulink Editor.

   The selected signals appear under **Model signal** in the **Click a signal in the model to select it** area.



3 (Optional) For buses, expand the bus signal to select individual elements.

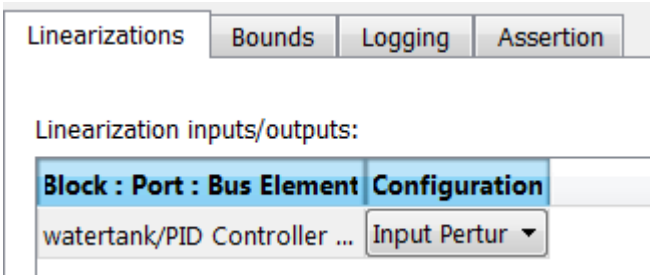
Tip For large buses or other large lists of signals, you can enter search text for filtering element names in the **Filter by name** edit box. The name match is case-sensitive. Additionally, you can enter a MATLAB regular expression.

To modify the filtering options, click . To hide the filtering options, click .

Filtering Options

- “Enable regular expression” on page 19-171
- “Show filtered results as a flat list” on page 19-171

4 Click « to add the selected signals to the **Linearization inputs/outputs** table.



To remove a signal from the **Linearization inputs/outputs** table, select the signal and click ×.

Tip To find the location in the Simulink model corresponding to a signal in the **Linearization inputs/outputs** table, select the signal in the table and click .

The table displays the following information about the selected signal:

Block : Port : Bus Element — Name of the block associated with the input/output. The number adjacent to the block name is the port number where the selected bus signal is located. The last entry is the selected bus element name.

Configuration

Type of linearization point:

- `Open-loop Input` — Specifies a linearization input point after a loop opening.
- `Open-loop Output` — Specifies a linearization output point before a loop opening.
- `Loop Transfer` — Specifies an output point before a loop opening followed by an input.
- `Input Perturbation` — Specifies an additive input to a signal.
- `Output Measurement` — Takes measurement at a signal.
- `Loop Break` — Specifies a loop opening.
- `Sensitivity` — Specifies an additive input followed by an output measurement.
- `Complementary Sensitivity` — Specifies an output followed by an additive input.

Note If you simulate the model without specifying an input or output, the software does not compute a linear system. Instead, you see a warning message at the MATLAB prompt.

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 on page 2-61

"Verify Model at Default Simulation Snapshot Time" on page 17-5

Click a signal in the model to select it

Enables signal selection in the Simulink model. Appears only when you click .

When this option appears, you also see the following changes:

- A new button.

  Use to add a selected signal as a linearization input or output in the **Linearization inputs/outputs** table. For more information, see **Linearization inputs/outputs** on page 19-168.

- 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 on page 2-61

"Verify Model at Default Simulation Snapshot Time" on page 17-5

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

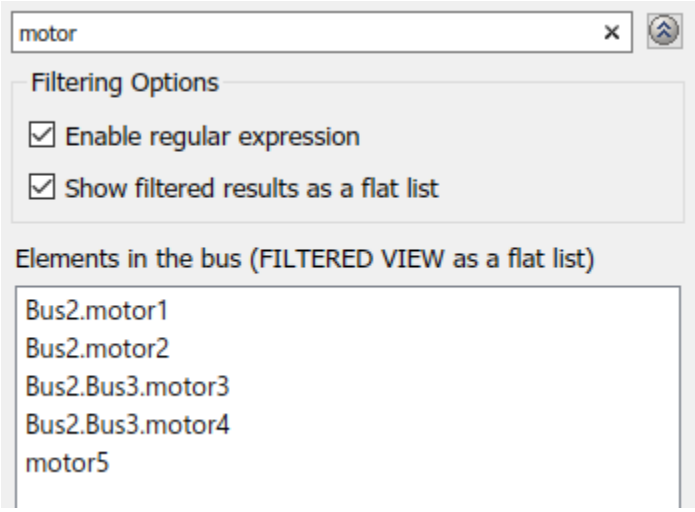
Disable use of MATLAB regular expressions for filtering signal names. Filtering treats the text you enter in the **Filter by name** edit box as a literal character vector.

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** on page 19-173.

Use when you:

- Know one or more times when the model is at a steady-state operating point
- Want to compute the linear systems at specific times

`External trigger`

Trigger-based simulation event. Specify the trigger type in **Trigger type** on page 19-174.

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: character vector
Value: `'SnapshotTimes'` | `'ExternalTrigger'`
Default: `'SnapshotTimes'`

See Also

Plot Linear Characteristics of Simulink Models During Simulation on page 2-61

"Verify Model at Default Simulation Snapshot Time" on page 17-5

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** on page 19-172 enables this parameter.

Command-Line Information

Parameter: `SnapshotTimes`
Type: character vector
Value: `0` | positive real number | vector of positive real numbers
Default: `0`

See Also

Plot Linear Characteristics of Simulink Models During Simulation on page 2-61

"Verify Model at Default Simulation Snapshot Time" on page 17-5

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** on page 19-172 enables this parameter.

Command-Line Information

Parameter: `TriggerType`
Type: character vector
Value: `'rising'` | `'falling'`
Default: `'rising'`

See Also

Plot Linear Characteristics of Simulink Models During Simulation on page 2-61

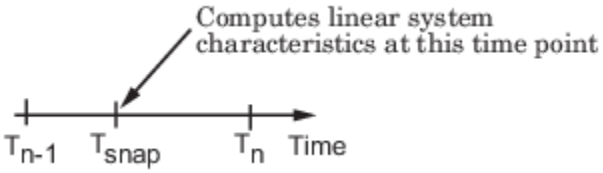
"Verify Model at Default Simulation Snapshot Time" on page 17-5

Enable zero-crossing detection

Enable zero-crossing detection to ensure that the software computes the linear system characteristics at the following simulation times:

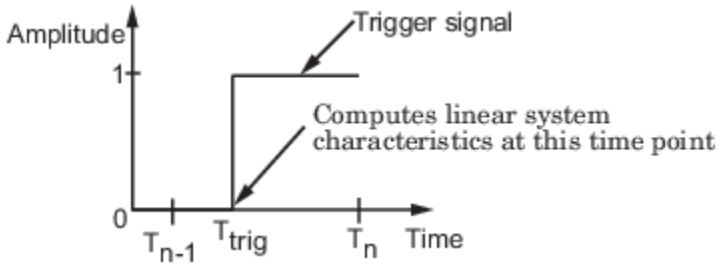
- The exact snapshot times, specified in **Snapshot times** on page 19-173.

  As shown in the following figure, when zero-crossing detection is enabled, the variable-step Simulink solver simulates the model at the snapshot time T_{snap}. T_{snap} may lie between the simulation time steps T_{n-1} and T_n which are automatically chosen by the solver.



- The exact times when an external trigger is detected, specified in **Trigger type** on page 19-174.

  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: character vector
Value: `'on'` | `'off'`
Default: `'on'`

See Also

Plot Linear Characteristics of Simulink Models During Simulation on page 2-61

“Verify Model at Default Simulation Snapshot Time” on page 17-5

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: character vector
Value: `'on'` | `'off'`
Default: `'off'`

See Also

Plot Linear Characteristics of Simulink Models During Simulation on page 2-61

"Verify Model at Default Simulation Snapshot Time" on page 17-5

Linear system sample time

Sample time of the linear system computed during simulation.

Use this parameter to:

- Compute a discrete-time system with a specific sample time from a continuous-time system
- Resample a discrete-time system with a different sample time
- Compute a continuous-time system from a discrete-time system

When computing discrete-time systems from continuous-time systems and vice-versa, the software uses the conversion method specified in **Sample time rate conversion method** on page 19-177.

Settings

Default: `auto`

`auto`. Computes the sample time as:

- 0, for continuous-time models.
- For models that have blocks with different sample times (multirate 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: character vector
Value: `'auto'` | Positive finite value | `'0'`
Default: `'auto'`

See Also

Plot Linear Characteristics of Simulink Models During Simulation on page 2-61

"Verify Model at Default Simulation Snapshot Time" on page 17-5

Sample time rate conversion method

Method for converting the sample time of single-rate or multirate models.

This parameter is used only when the value of **Linear system sample time** on page 19-176 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".

This method usually performs better in the 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".

This method usually performs 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".

This method usually performs 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 a discrete-time system to a new faster sample time that is an integer multiple of the sample 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 a discrete-time system to a new faster sample time that is an integer multiple of the sample 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 a discrete-time system to a new faster sample time that is an integer multiple of the sample time of the original system.

Dependencies

Selecting either:

- `Tustin with Prewarping`
- `Upsampling when possible, Tustin with Prewarping otherwise`

enables **Prewarp frequency (rad/s)** on page 19-178.

Command-Line Information

Parameter: `RateConversionMethod`
Type: character vector
Value: `'zoh'` | `'tustin'` | `'prewarp'` | `'upsampling_zoh'` | `'upsampling_tustin'` | `'upsampling_prewarp'`
Default: `'zoh'`

See Also

Plot Linear Characteristics of Simulink Models During Simulation on page 2-61

"Verify Model at Default Simulation Snapshot Time" on page 17-5

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** on page 19-177 enables this parameter.

Command-Line Information

Parameter: `PreWarpFreq`
Type: character vector
Value: `10` | positive scalar value
Default: `10`

See Also

Plot Linear Characteristics of Simulink Models During Simulation on page 2-61

"Verify Model at Default Simulation Snapshot Time" on page 17-5

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: character vector
Value: `'on'` | `'off'`
Default: `'off'`

See Also

Plot Linear Characteristics of Simulink Models During Simulation on page 2-61

“Verify Model at Default Simulation Snapshot Time” on page 17-5

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: character vector
Value: `'on'` | `'off'`
Default: `'off'`

See Also

Plot Linear Characteristics of Simulink Models During Simulation on page 2-61

"Verify Model at Default Simulation Snapshot Time" on page 17-5

Include settling time bound in assertion

Check that the pole locations satisfy approximate second-order bounds on the settling time, specified in **Settling time (sec) <=** on page 19-181. The software displays a warning if the poles lie outside the region defined by the settling time bound.

This parameter is used for assertion only if **Enable assertion** on page 19-190 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

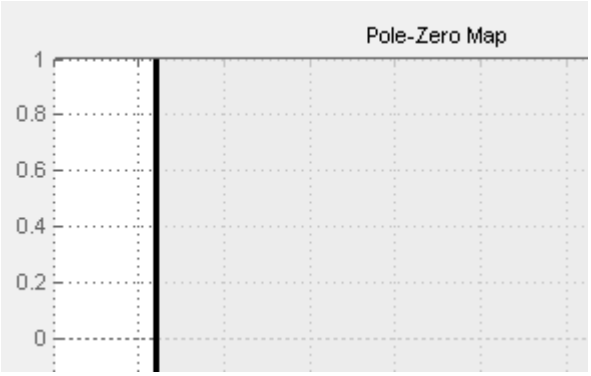
Check that each pole lies in the region defined by the settling time bound, during simulation.

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 on page 19-182, damping ratio on page 19-184 or natural frequency on page 19-186, 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: character vector
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 on page 2-61

"Verify Model at Default Simulation Snapshot Time" on page 17-5

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** on page 19-180 and **Enable assertion** on page 19-190.
- 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: character vector
Value: `[]` | `1` | `finite positive real scalar` | `cell array of finite positive real scalars`. Must be specified inside single quotes (`''`).
Default: `'[]'` for Pole-Zero Plot block, `'1'` for Check Pole-Zero Characteristics block.

See Also

Plot Linear Characteristics of Simulink Models During Simulation on page 2-61

“Verify Model at Default Simulation Snapshot Time” on page 17-5

Include percent overshoot bound in assertion

Check that the pole locations satisfy approximate second-order bounds on the percent overshoot, specified in **Percent overshoot <=** on page 19-181. The software displays a warning if the poles lie outside the region defined by the percent overshoot bound.

This parameter is used for assertion only if **Enable assertion** on page 19-190 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

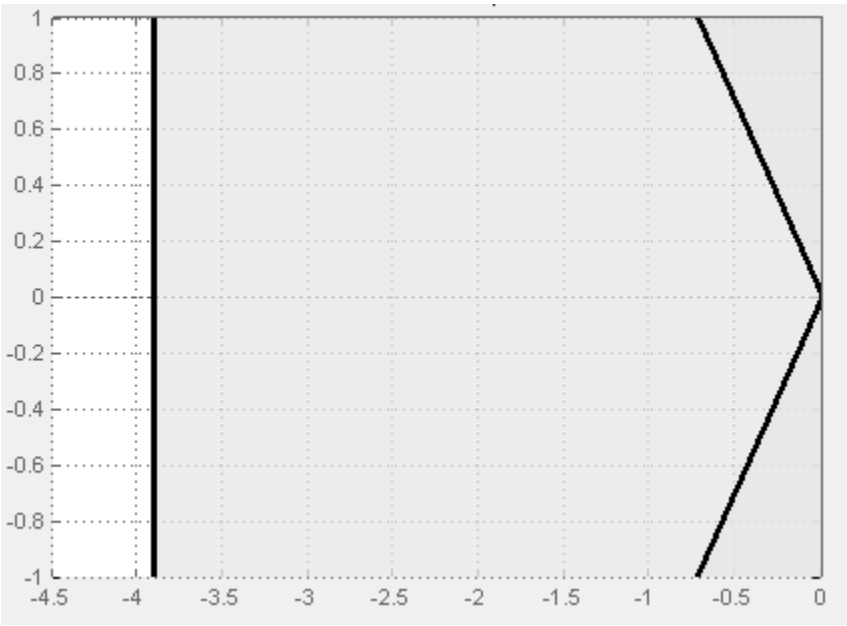
Check that each pole lies in the region defined by the percent overshoot bound, during simulation.

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 on page 19-180, damping ratio on page 19-184 or natural frequency on page 19-186, 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: character vector
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 on page 2-61

“Verify Model at Default Simulation Snapshot Time” on page 17-5

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 on page 19-185 ζ, as:

  $p.o. = 100e^{-\pi\zeta/\sqrt{1-\zeta^2}}$.
- To assert that the percent overshoot bounds are satisfied, select both **Include percent overshoot bound in assertion** on page 19-180 and **Enable assertion** on page 19-190.
- 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: character vector
Value: `[]` | `10` | `real scalar between 0 and 100` | `cell array of real scalars between 0 and 100`. Must be specified inside single quotes (`''`).
Default: `'[]'` for Pole-Zero Plot block, `'10'` for Check Pole-Zero Characteristics block.

See Also

Plot Linear Characteristics of Simulink Models During Simulation on page 2-61

"Verify Model at Default Simulation Snapshot Time" on page 17-5

Include damping ratio bound in assertion

Check that the pole locations satisfy approximate second-order bounds on the damping ratio, specified in **Damping ratio >=** on page 19-184. The software displays a warning if the poles lie outside the region defined by the damping ratio bound.

This parameter is used for assertion only if **Enable assertion** on page 19-190 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

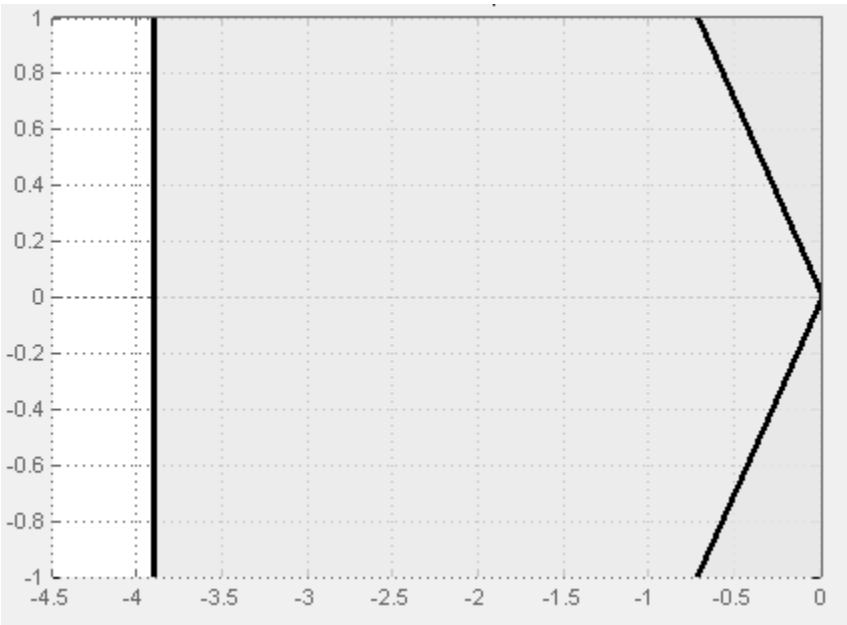
Check that each pole lies in the region defined by the damping ratio bound, during simulation.

☐ 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 on page 19-180, percent overshoot on page 19-182 or natural frequency on page 19-186, 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: character vector
Value: `'on'` | `'off'`
Default: `'off'`

See Also

Plot Linear Characteristics of Simulink Models During Simulation on page 2-61

“Verify Model at Default Simulation Snapshot Time” on page 17-5

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** on page 19-184 and **Enable assertion** on page 19-190.
- 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: character vector
Value: `[]|finite positive real scalar between 0 and 1|cell array of finite positive real scalars between 0 and 1` . Must be specified inside single quotes (`''`).
Default: `'[]'`

See Also

Plot Linear Characteristics of Simulink Models During Simulation on page 2-61

"Verify Model at Default Simulation Snapshot Time" on page 17-5

Include natural frequency bound in assertion

Check that the pole locations satisfy approximate second-order bounds on the natural frequency, specified in **Natural frequency (rad/sec)** on page 19-187. The natural frequency bound can be greater than, less than or equal one or more specific values. The software displays a warning if the pole locations do not satisfy the region defined by the natural frequency bound.

This parameter is used for assertion only if **Enable assertion** on page 19-190 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

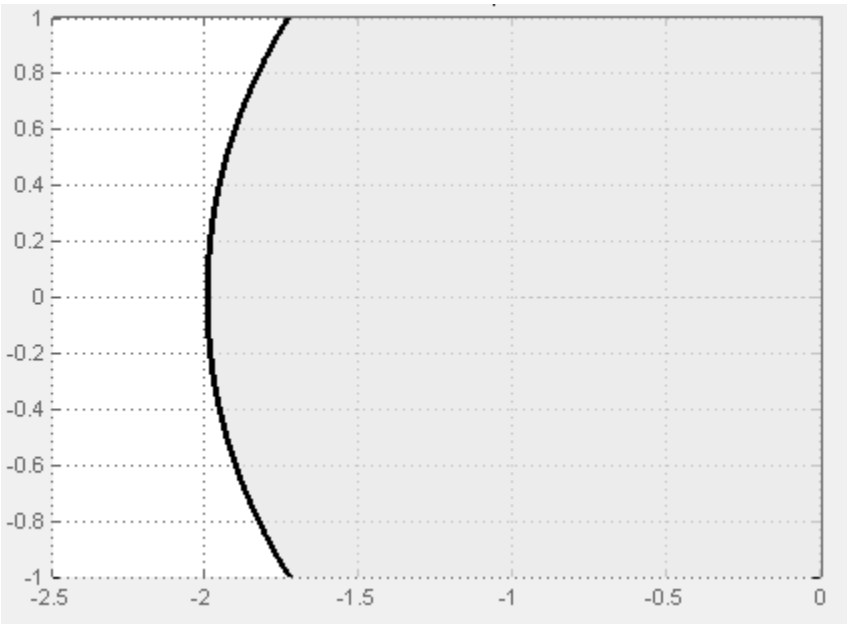
Check that each pole lies in the region defined by the natural frequency bound, during simulation.

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 on page 19-180, percent overshoot on page 19-182 or damping ratio on page 19-184 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: character vector
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** on page 19-184 and **Enable assertion** on page 19-190.
- 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: character vector
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 on page 2-61

"Verify Model at Default Simulation Snapshot Time" on page 17-5

Save data to workspace

Save one or more linear systems to perform further linear analysis or control design.

The saved data is in a structure whose fields include:

- `time` — Simulation times at which the linear systems are computed.
- `values` — State-space model representing the linear system. If the linear system is computed at multiple simulation times, `values` is an array of state-space models.
- `operatingPoints` — Operating points corresponding to each linear system in `values`. This field exists only if **Save operating points for each linearization** is checked.

The location of the saved data structure depends upon the configuration of the Simulink model:

- If the Simulink model is not configured to save simulation output as a single object, the data structure is a variable in the MATLAB workspace.
- If the Simulink model is configured to save simulation output as a single object, the data structure is a field in the `Simulink.SimulationOutput` object that contains the logged simulation data.

  To configure your model to save simulation output in a single object, in the Simulink editor, on the **Modeling** tab, click **Model Settings**. Then, in the Configuration Parameters dialog box, select the **Single simulation output** parameter.

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** on page 19-189.

Command-Line Information

Parameter: `SaveToWorkspace`
Type: character vector
Value: `'on'` | `'off'`
Default: `'off'`

See Also

Plot Linear Characteristics of Simulink Models During Simulation on page 2-61

"Verify Model at Default Simulation Snapshot Time" on page 17-5

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 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

Character vector.

Dependencies

Save data to workspace on page 19-188 enables this parameter.

Command-Line Information

Parameter: `SaveName`

Type: character vector
Value: `sys` | any character vector. Must be specified inside single quotes (`''`).
Default: `'sys'`

See Also

Plot Linear Characteristics of Simulink Models During Simulation on page 2-61

"Verify Model at Default Simulation Snapshot Time" on page 17-5

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 on page 19-188 enables this parameter.

Command-Line Information

Parameter: `SaveOperatingPoint`
Type: character vector
Value: `'on'` | `'off'`
Default: `'off'`

See Also

Plot Linear Characteristics of Simulink Models During Simulation on page 2-61

"Verify Model at Default Simulation Snapshot Time" on page 17-5

Enable assertion

Enable the block to check that bounds specified and included for assertion in the **Bounds** tab are satisfied during simulation. Assertion fails if a bound is not satisfied. A warning, reporting the assertion failure, appears at the MATLAB prompt.

If assertion fails, you can optionally specify that the block:

- Execute a MATLAB expression, specified in **Simulation callback when assertion fails (optional)** on page 19-191.
- Stop the simulation and bring that block into focus, by selecting **Stop simulation when assertion fails** on page 19-192.

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; that is, the block no longer checks that specified bounds are satisfied. The block icon also updates to indicate that assertion is disabled.

In the Simulink model, in the Configuration Parameters dialog box, the **Model Verification block enabling** parameter lets you enable or disable all model verification blocks in a model, regardless of the setting of this option in the block.

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: character vector
Value: `'on'` | `'off'`
Default: `'on'`

See Also

Plot Linear Characteristics of Simulink Models During Simulation on page 2-61

“Verify Model at Default Simulation Snapshot Time” on page 17-5

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 on page 19-190 enables this parameter.

Command-Line Information

Parameter: `callback`
Type: character vector
Value: `''` | MATLAB expression
Default: `''`

See Also

Plot Linear Characteristics of Simulink Models During Simulation on page 2-61

"Verify Model at Default Simulation Snapshot Time" on page 17-5

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 on page 19-190 enables this parameter.

Command-Line Information

Parameter: `stopWhenAssertionFail`
Type: character vector
Value: `'on'` | `'off'`
Default: `'off'`

See Also

Plot Linear Characteristics of Simulink Models During Simulation on page 2-61

“Verify Model at Default Simulation Snapshot Time” on page 17-5

Output assertion signal

Output a Boolean signal that, at each time step, is:

- True (1) if assertion succeeds; that is, all bounds are satisfied
- False (1) if assertion fails; that is, a bound is violated.

The output signal data type is Boolean only if, in the Simulink model, in the Configuration Parameters dialog box, the **Implement logic signals as Boolean data** parameter is selected. Otherwise, the data type of the output signal is double.

Selecting this parameter adds an output port to the block that you can connect to any block in the model.

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 “Verify Model Using Simulink Control Design and Simulink Verification Blocks” on page 17-20.

Command-Line Information

Parameter: `export`
Type: character vector
Value: `'on'` | `'off'`
Default: `'off'`

See Also

Plot Linear Characteristics of Simulink Models During Simulation on page 2-61

“Verify Model at Default Simulation Snapshot Time” on page 17-5

Show plot on block open

Open the plot window instead of the Block Parameters dialog box when you double-click the block in the Simulink model.

Use this parameter if you prefer to open and perform tasks, such as adding or modifying bounds, in the plot window instead of the Block Parameters dialog box. If you want to access the block parameters from the plot window, select **Edit** or click .

For more information on the plot, see “Show Plot” on page 19-0 .

Settings

Default: Off

On

Open the plot window when you double-click the block.

Off

Open the Block Parameters dialog box when you double-click the block.

Command-Line Information

Parameter: `LaunchViewOnOpen`
Type: character vector
Value: `'on' | 'off'`
Default: `'off'`

See Also

Plot Linear Characteristics of Simulink Models During Simulation on page 2-61

“Verify Model at Default Simulation Snapshot Time” on page 17-5

Show Plot

Open the plot window.

Use the plot to view:

- System characteristics and signals computed during simulation

  You must click this button before you simulate the model to view the system characteristics or signal.

  You can display additional characteristics, such as the peak response time, by right-clicking the plot and selecting **Characteristics**.
- Bounds

  You can specify bounds in the **Bounds** tab of the Block Parameters dialog box or right-click the plot and select **Bounds > New Bound**. For more information on the types of bounds you can specify, see the individual reference pages.

  You can modify bounds by dragging the bound segment or by right-clicking the plot and selecting **Bounds > Edit Bound**. Before you simulate the model, click **Update Block** to update the bound value in the block parameters.

Typical tasks that you perform in the plot window include:

- Opening the Block Parameters dialog box by clicking or selecting **Edit**.
- Finding the block that the plot window corresponds to by clicking or selecting **View > Highlight Simulink Block**. This action makes the model window active and highlights the block.

- Simulating the model by clicking . This action also linearizes the portion of the model between the specified linearization input and output.
- Adding a legend on the linear system characteristic plot by clicking .

Note To optimize the model response to meet design requirements specified in the **Bounds** tab, open the **Response Optimizer** by selecting **Tools > Response Optimization** in the plot window. This option is only available if you have Simulink Design Optimization software installed.

Response Optimization

Open the **Response Optimizer** to optimize the model response to meet design requirements specified in the **Bounds** tab.

This button is available only if you have Simulink Design Optimization software installed.

See Also

- “Design Optimization to Meet Step Response Requirements (GUI)” (Simulink Design Optimization)
- “Design Optimization to Meet Time-Domain and Frequency-Domain Requirements (GUI)” (Simulink Design Optimization)

See Also

Check Pole-Zero Characteristics

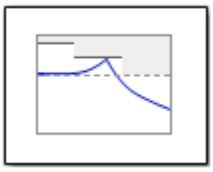
Tutorials

- “Visualize Bode Response of Simulink Model During Simulation” on page 2-61
- “Visualize Linear System at Multiple Simulation Snapshots” on page 2-85
- “Visualize Linear System of a Continuous-Time Model Discretized During Simulation” on page 2-91
- “Plot Linear System Characteristics of a Chemical Reactor” on page 2-95

Introduced in R2010b

Singular Value Plot

Singular value plot of linear system approximated from nonlinear Simulink model



Library

Simulink Control Design

Description

This block is the 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-time, 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 block displays plots for all input/output combinations.

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 in the MATLAB Command Window. 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 any 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 19-198. • "Click a signal in the model to select it" on page 19-200.
	Specify settings.	In **Linearizations** tab: • "Linearize on" on page 19-202. • "Snapshot times" on page 19-203. • "Trigger type" on page 19-204.
	Specify algorithm options.	In **Algorithm Options** of **Linearizations** tab: • "Enable zero-crossing detection" on page 19-204. • "Use exact delays" on page 19-205. • "Linear system sample time" on page 19-206. • "Sample time rate conversion method" on page 19-207. • "Prewarp frequency (rad/s)" on page 19-208.
	Specify labels for linear system I/Os and state names.	In **Labels** of **Linearizations** tab: • "Use full block names" on page 19-208. • "Use bus signal names" on page 19-209.
Plot the linear system.		**Show Plot** on page 19-222

Task	Parameters
(Optional) Specify bounds on singular values for assertion.	In **Bounds** tab: • Include upper singular value bound in assertion on page 19-210. • Include lower singular value bound in assertion on page 19-213.
Specify assertion options (only when you specify bounds on the linear system).	In **Assertion** tab: • “Enable assertion” on page 19-218. • “Simulation callback when assertion fails (optional)” on page 19-219. • “Stop simulation when assertion fails” on page 19-220. • “Output assertion signal” on page 19-221.
Save linear system to MATLAB workspace.	“Save data to workspace” on page 19-216 in **Logging** tab.
Display plot window instead of block parameters dialog box on double-clicking the block.	“Show plot on block open” on page 19-221.

Linearization inputs/outputs

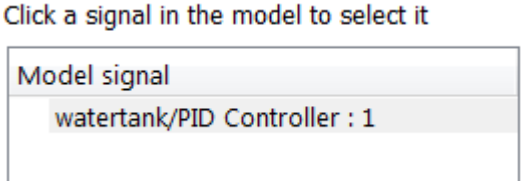
Linearization inputs and outputs that define the portion of a nonlinear Simulink model to linearize.

If you have defined the linearization input and output in the Simulink model, the block automatically detects these points and displays them in the **Linearization inputs/outputs** area. Click at any time to update the **Linearization inputs/outputs** table with I/Os from the model. To add other analysis points:

1. Click .

   The dialog box expands to display a **Click a signal in the model to select it** on page 19-200 area and a new button.
2. Select one or more signals in the Simulink Editor.

   The selected signals appear under **Model signal** in the **Click a signal in the model to select it** area.



3 (Optional) For buses, expand the bus signal to select individual elements.

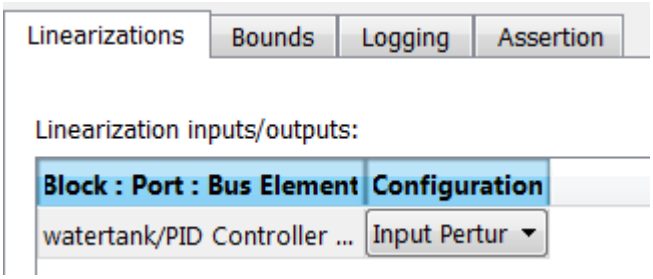
Tip For large buses or other large lists of signals, you can enter search text for filtering element names in the **Filter by name** edit box. The name match is case-sensitive. Additionally, you can enter a MATLAB regular expression.

To modify the filtering options, click . To hide the filtering options, click .

Filtering Options

- “Enable regular expression” on page 19-201
- “Show filtered results as a flat list” on page 19-201

4 Click « to add the selected signals to the **Linearization inputs/outputs** table.



To remove a signal from the **Linearization inputs/outputs** table, select the signal and click ×.

Tip To find the location in the Simulink model corresponding to a signal in the **Linearization inputs/outputs** table, select the signal in the table and click .

The table displays the following information about the selected signal:

Block : Port : Bus Element — Name of the block associated with the input/output. The number adjacent to the block name is the port number where the selected bus signal is located. The last entry is the selected bus element name.

Configuration

Type of linearization point:

- `Open-loop Input` — Specifies a linearization input point after a loop opening.
- `Open-loop Output` — Specifies a linearization output point before a loop opening.
- `Loop Transfer` — Specifies an output point before a loop opening followed by an input.
- `Input Perturbation` — Specifies an additive input to a signal.
- `Output Measurement` — Takes measurement at a signal.
- `Loop Break` — Specifies a loop opening.
- `Sensitivity` — Specifies an additive input followed by an output measurement.
- `Complementary Sensitivity` — Specifies an output followed by an additive input.

Note If you simulate the model without specifying an input or output, the software does not compute a linear system. Instead, you see a warning message at the MATLAB prompt.

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 on page 2-61

"Verify Model at Default Simulation Snapshot Time" on page 17-5

Click a signal in the model to select it

Enables signal selection in the Simulink model. Appears only when you click .

When this option appears, you also see the following changes:

- A new button.

  Use to add a selected signal as a linearization input or output in the **Linearization inputs/outputs** table. For more information, see **Linearization inputs/outputs** on page 19-198.

- 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 on page 2-61

"Verify Model at Default Simulation Snapshot Time" on page 17-5

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

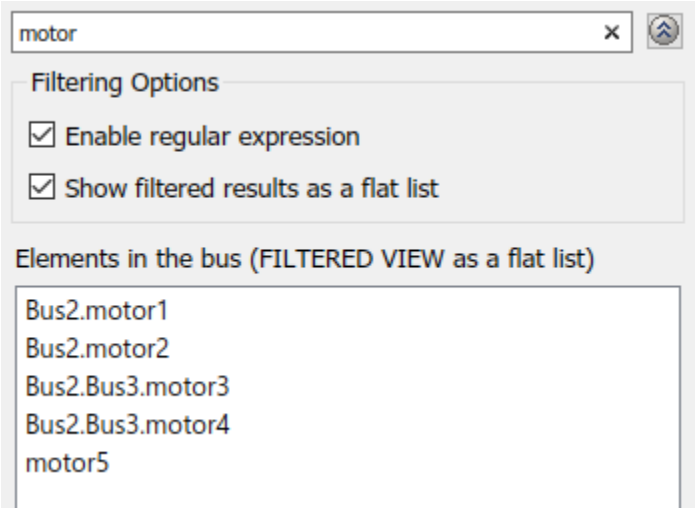
Disable use of MATLAB regular expressions for filtering signal names. Filtering treats the text you enter in the **Filter by name** edit box as a literal character vector.

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** on page 19-203.

Use when you:

- Know one or more times when the model is at a steady-state operating point
- Want to compute the linear systems at specific times

`External trigger`

Trigger-based simulation event. Specify the trigger type in **Trigger type** on page 19-204.

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: character vector
Value: `'SnapshotTimes'` | `'ExternalTrigger'`
Default: `'SnapshotTimes'`

See Also

Plot Linear Characteristics of Simulink Models During Simulation on page 2-61

"Verify Model at Default Simulation Snapshot Time" on page 17-5

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** on page 19-202 enables this parameter.

Command-Line Information

Parameter: `SnapshotTimes`
Type: character vector
Value: `0` | positive real number | vector of positive real numbers
Default: `0`

See Also

Plot Linear Characteristics of Simulink Models During Simulation on page 2-61

"Verify Model at Default Simulation Snapshot Time" on page 17-5

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** on page 19-202 enables this parameter.

Command-Line Information

Parameter: `TriggerType`
Type: character vector
Value: `'rising'` | `'falling'`
Default: `'rising'`

See Also

Plot Linear Characteristics of Simulink Models During Simulation on page 2-61

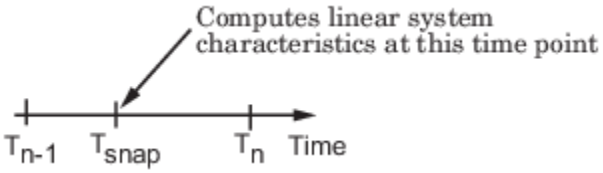
"Verify Model at Default Simulation Snapshot Time" on page 17-5

Enable zero-crossing detection

Enable zero-crossing detection to ensure that the software computes the linear system characteristics at the following simulation times:

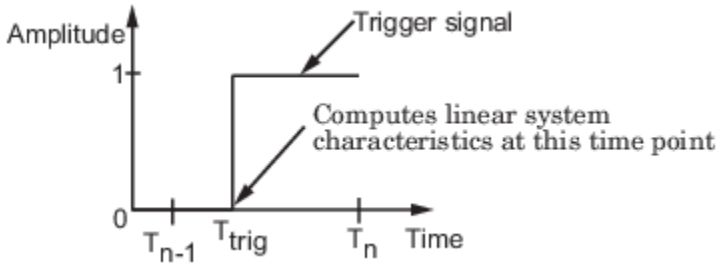
- The exact snapshot times, specified in **Snapshot times** on page 19-203.

  As shown in the following figure, when zero-crossing detection is enabled, the variable-step Simulink solver simulates the model at the snapshot time T_{snap}. T_{snap} may lie between the simulation time steps T_{n-1} and T_n which are automatically chosen by the solver.



- The exact times when an external trigger is detected, specified in **Trigger type** on page 19-204.

  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: character vector
Value: `'on'` | `'off'`
Default: `'on'`

See Also

Plot Linear Characteristics of Simulink Models During Simulation on page 2-61

“Verify Model at Default Simulation Snapshot Time” on page 17-5

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: character vector
Value: `'on'` | `'off'`
Default: `'off'`

See Also

Plot Linear Characteristics of Simulink Models During Simulation on page 2-61

"Verify Model at Default Simulation Snapshot Time" on page 17-5

Linear system sample time

Sample time of the linear system computed during simulation.

Use this parameter to:

- Compute a discrete-time system with a specific sample time from a continuous-time system
- Resample a discrete-time system with a different sample time
- Compute a continuous-time system from a discrete-time system

When computing discrete-time systems from continuous-time systems and vice-versa, the software uses the conversion method specified in **Sample time rate conversion method** on page 19-207.

Settings

Default: `auto`

`auto`. Computes the sample time as:

- 0, for continuous-time models.
- For models that have blocks with different sample times (multirate 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: character vector
Value: `'auto'` | Positive finite value | `'0'`
Default: `'auto'`

See Also

Plot Linear Characteristics of Simulink Models During Simulation on page 2-61

“Verify Model at Default Simulation Snapshot Time” on page 17-5

Sample time rate conversion method

Method for converting the sample time of single-rate or multirate models.

This parameter is used only when the value of **Linear system sample time** on page 19-206 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”.

This method usually performs better in the 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”.

This method usually performs 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”.

This method usually performs 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 a discrete-time system to a new faster sample time that is an integer multiple of the sample 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 a discrete-time system to a new faster sample time that is an integer multiple of the sample 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 a discrete-time system to a new faster sample time that is an integer multiple of the sample time of the original system.

Dependencies

Selecting either:

- `Tustin with Prewarping`
- `Upsampling when possible, Tustin with Prewarping otherwise`

enables **Prewarp frequency (rad/s)** on page 19-208.

Command-Line Information

Parameter: `RateConversionMethod`
Type: character vector
Value: `'zoh'` | `'tustin'` | `'prewarp'` | `'upsampling_zoh'` | `'upsampling_tustin'` | `'upsampling_prewarp'`
Default: `'zoh'`

See Also

Plot Linear Characteristics of Simulink Models During Simulation on page 2-61

"Verify Model at Default Simulation Snapshot Time" on page 17-5

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** on page 19-207 enables this parameter.

Command-Line Information

Parameter: `PreWarpFreq`
Type: character vector
Value: `10` | positive scalar value
Default: `10`

See Also

Plot Linear Characteristics of Simulink Models During Simulation on page 2-61

"Verify Model at Default Simulation Snapshot Time" on page 17-5

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: character vector
Value: `'on'` | `'off'`
Default: `'off'`

See Also

Plot Linear Characteristics of Simulink Models During Simulation on page 2-61

“Verify Model at Default Simulation Snapshot Time” on page 17-5

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: character vector
Value: `'on'` | `'off'`
Default: `'off'`

See Also

Plot Linear Characteristics of Simulink Models During Simulation on page 2-61

"Verify Model at Default Simulation Snapshot Time" on page 17-5

Include upper singular value bound in assertion

Check that the singular values satisfy upper bounds, specified in **Frequencies (rad/sec)** on page 19-211 and **Magnitude (dB)** on page 19-212, during simulation. The software displays a warning during simulation if the singular values violate the upper bound.

This parameter is used for assertion only if **Enable assertion** on page 19-218 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

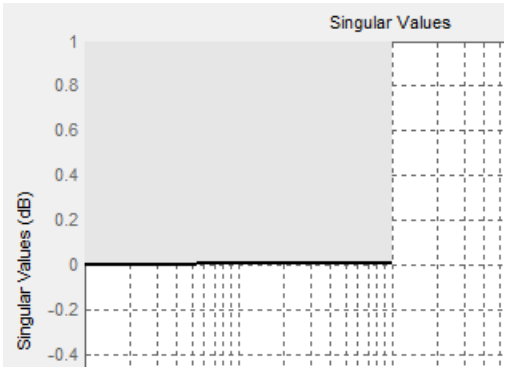
Check that the singular value satisfies the specified upper bounds, during simulation.

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 on page 19-213 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: character vector
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 on page 2-61

“Verify Model at Default Simulation Snapshot Time” on page 17-5

Frequencies (rad/sec)

Frequencies for one or more upper singular value bound segments, specified in radians/sec.

Specify the corresponding magnitudes in **Magnitude (dB)** on page 19-212.

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** on page 19-210 and **Enable assertion** on page 19-218.
- 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: character vector
Value: `[]` | `[0.1 100]` | positive finite numbers | matrix of positive finite numbers | cell array of matrices with positive finite numbers. Must be specified inside single quotes (`''`).
Default: `'[]'` for Singular Value Plot block, `'[0.1 100]'` for Check Singular Value Characteristics block.

See Also

Plot Linear Characteristics of Simulink Models During Simulation on page 2-61

"Verify Model at Default Simulation Snapshot Time" on page 17-5

Magnitudes (dB)

Magnitude values for one or more upper singular value bound segments, specified in decibels.

Specify the corresponding frequencies in **Frequencies (rad/sec)** on page 19-211.

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** on page 19-210 and **Enable assertion** on page 19-218.
- 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: character vector
Value: `[]` | `[0 0]` | finite number | matrix of finite numbers | cell array of matrices with finite numbers. Must be specified inside single quotes (`''`).
Default: `'[]'` for Singular Value Plot block, `'[0 0]'` for Check Singular Value Characteristics block.

See Also

Plot Linear Characteristics of Simulink Models During Simulation on page 2-61

“Verify Model at Default Simulation Snapshot Time” on page 17-5

Include lower singular value bound in assertion

Check that the singular values satisfy lower bounds, specified in **Frequencies (rad/sec)** on page 19-214 and **Magnitude (dB)** on page 19-215, during simulation. The software displays a warning if the singular values violate the lower bound.

This parameter is used for assertion only if **Enable assertion** on page 19-218 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

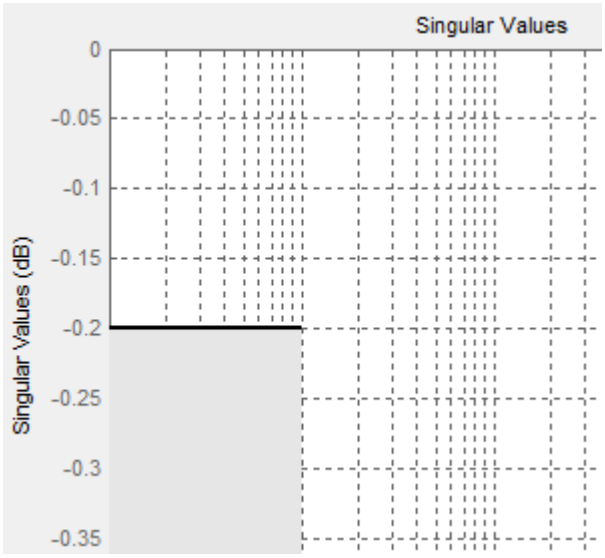
Check that the singular value satisfies the specified lower bounds, during simulation.

☐ 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 on page 19-210 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: character vector
Value: `'on'` | `'off'`
Default: `'off'`

See Also

Plot Linear Characteristics of Simulink Models During Simulation on page 2-61

"Verify Model at Default Simulation Snapshot Time" on page 17-5

Frequencies (rad/sec)

Frequencies for one or more lower singular value bound segments, specified in radians/sec.

Specify the corresponding magnitudes in **Magnitude (dB)** on page 19-215.

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** on page 19-213 and **Enable assertion** on page 19-218.
- 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: character vector
Value: `[]` | positive finite number | matrix of positive finite numbers | cell array of matrices with positive finite numbers. Must be specified inside single quotes (`' '`).
Default: `'[]'`

See Also

Plot Linear Characteristics of Simulink Models During Simulation on page 2-61

"Verify Model at Default Simulation Snapshot Time" on page 17-5

Magnitudes (dB)

Magnitude values for one or more lower singular value bound segments, specified in decibels.

Specify the corresponding frequencies in **Frequencies (rad/sec)** on page 19-214.

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** on page 19-213 and **Enable assertion** on page 19-218.
- 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: character vector
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 on page 2-61

"Verify Model at Default Simulation Snapshot Time" on page 17-5

Save data to workspace

Save one or more linear systems to perform further linear analysis or control design.

The saved data is in a structure whose fields include:

- `time` — Simulation times at which the linear systems are computed.
- `values` — State-space model representing the linear system. If the linear system is computed at multiple simulation times, `values` is an array of state-space models.
- `operatingPoints` — Operating points corresponding to each linear system in `values`. This field exists only if **Save operating points for each linearization** is checked.

The location of the saved data structure depends upon the configuration of the Simulink model:

- If the Simulink model is not configured to save simulation output as a single object, the data structure is a variable in the MATLAB workspace.
- If the Simulink model is configured to save simulation output as a single object, the data structure is a field in the `Simulink.SimulationOutput` object that contains the logged simulation data.

  To configure your model to save simulation output in a single object, in the Simulink editor, on the **Modeling** tab, click **Model Settings**. Then, in the Configuration Parameters dialog box, select the **Single simulation output** parameter.

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** on page 19-217.

Command-Line Information

Parameter: `SaveToWorkspace`
Type: character vector
Value: `'on'` | `'off'`
Default: `'off'`

See Also

Plot Linear Characteristics of Simulink Models During Simulation on page 2-61

"Verify Model at Default Simulation Snapshot Time" on page 17-5

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 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

Character vector.

Dependencies

Save data to workspace on page 19-216 enables this parameter.

Command-Line Information

Parameter: `SaveName`
Type: character vector
Value: `sys` | any character vector. Must be specified inside single quotes (`''`).
Default: `'sys'`

See Also

Plot Linear Characteristics of Simulink Models During Simulation on page 2-61

"Verify Model at Default Simulation Snapshot Time" on page 17-5

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 on page 19-216 enables this parameter.

Command-Line Information

Parameter: `SaveOperatingPoint`
Type: character vector
Value: `'on'` | `'off'`
Default: `'off'`

See Also

Plot Linear Characteristics of Simulink Models During Simulation on page 2-61

"Verify Model at Default Simulation Snapshot Time" on page 17-5

Enable assertion

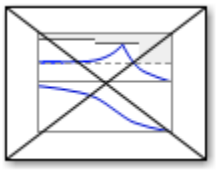
Enable the block to check that bounds specified and included for assertion in the **Bounds** tab are satisfied during simulation. Assertion fails if a bound is not satisfied. A warning, reporting the assertion failure, appears at the MATLAB prompt.

If assertion fails, you can optionally specify that the block:

- Execute a MATLAB expression, specified in **Simulation callback when assertion fails (optional)** on page 19-219.
- Stop the simulation and bring that block into focus, by selecting **Stop simulation when assertion fails** on page 19-220.

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; that is, the block no longer checks that specified bounds are satisfied. The block icon also updates to indicate that assertion is disabled.



In the Simulink model, in the Configuration Parameters dialog box, the **Model Verification block enabling** parameter lets you enable or disable all model verification blocks in a model, regardless of the setting of this option in the block.

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: character vector
Value: `'on'` | `'off'`
Default: `'on'`

See Also

Plot Linear Characteristics of Simulink Models During Simulation on page 2-61

“Verify Model at Default Simulation Snapshot Time” on page 17-5

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 on page 19-218 enables this parameter.

Command-Line Information

Parameter: `callback`
Type: character vector
Value: `''` | MATLAB expression
Default: `''`

See Also

Plot Linear Characteristics of Simulink Models During Simulation on page 2-61

"Verify Model at Default Simulation Snapshot Time" on page 17-5

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 on page 19-218 enables this parameter.

Command-Line Information

Parameter: `stopWhenAssertionFail`
Type: character vector
Value: `'on'` | `'off'`
Default: `'off'`

See Also

Plot Linear Characteristics of Simulink Models During Simulation on page 2-61

"Verify Model at Default Simulation Snapshot Time" on page 17-5

Output assertion signal

Output a Boolean signal that, at each time step, is:

- True (1) if assertion succeeds; that is, all bounds are satisfied
- False (1) if assertion fails; that is, a bound is violated.

The output signal data type is Boolean only if, in the Simulink model, in the Configuration Parameters dialog box, the **Implement logic signals as Boolean data** parameter is selected. Otherwise, the data type of the output signal is double.

Selecting this parameter adds an output port to the block that you can connect to any block in the model.

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 “Verify Model Using Simulink Control Design and Simulink Verification Blocks” on page 17-20.

Command-Line Information

Parameter: `export`
Type: character vector
Value: `'on'` | `'off'`
Default: `'off'`

See Also

Plot Linear Characteristics of Simulink Models During Simulation on page 2-61

“Verify Model at Default Simulation Snapshot Time” on page 17-5

Show plot on block open

Open the plot window instead of the Block Parameters dialog box when you double-click the block in the Simulink model.

Use this parameter if you prefer to open and perform tasks, such as adding or modifying bounds, in the plot window instead of the Block Parameters dialog box. If you want to access the block parameters from the plot window, select **Edit** or click .

For more information on the plot, see “Show Plot” on page 19-0 .

Settings

Default: Off

On

Open the plot window when you double-click the block.

Off

Open the Block Parameters dialog box when you double-click the block.

Command-Line Information
Parameter: `LaunchViewOnOpen`
Type: character vector
Value: `'on' | 'off'`
Default: `'off'`

See Also

Plot Linear Characteristics of Simulink Models During Simulation on page 2-61

“Verify Model at Default Simulation Snapshot Time” on page 17-5

Show Plot

Open the plot window.

Use the plot to view:

- System characteristics and signals computed during simulation

  You must click this button before you simulate the model to view the system characteristics or signal.

  You can display additional characteristics, such as the peak response time, by right-clicking the plot and selecting **Characteristics**.
- Bounds

  You can specify bounds in the **Bounds** tab of the Block Parameters dialog box or right-click the plot and select **Bounds > New Bound**. For more information on the types of bounds you can specify, see the individual reference pages.

  You can modify bounds by dragging the bound segment or by right-clicking the plot and selecting **Bounds > Edit Bound**. Before you simulate the model, click **Update Block** to update the bound value in the block parameters.

Typical tasks that you perform in the plot window include:

- Opening the Block Parameters dialog box by clicking or selecting **Edit**.
- Finding the block that the plot window corresponds to by clicking or selecting **View > Highlight Simulink Block**. This action makes the model window active and highlights the block.
- Simulating the model by clicking . This action also linearizes the portion of the model between the specified linearization input and output.
- Adding a legend on the linear system characteristic plot by clicking .

Note To optimize the model response to meet design requirements specified in the **Bounds** tab, open the **Response Optimizer** by selecting **Tools > Response Optimization** in the plot window. This option is only available if you have Simulink Design Optimization software installed.

Response Optimization

Open the **Response Optimizer** to optimize the model response to meet design requirements specified in the **Bounds** tab.

This button is available only if you have Simulink Design Optimization software installed.

See Also

- “Design Optimization to Meet Step Response Requirements (GUI)” (Simulink Design Optimization)
- “Design Optimization to Meet Time-Domain and Frequency-Domain Requirements (GUI)” (Simulink Design Optimization)

See Also

Check Singular Value Characteristics

Tutorials

- “Visualize Bode Response of Simulink Model During Simulation” on page 2-61
- “Visualize Linear System at Multiple Simulation Snapshots” on page 2-85
- “Visualize Linear System of a Continuous-Time Model Discretized During Simulation” on page 2-91
- “Plot Linear System Characteristics of a Chemical Reactor” on page 2-95

Introduced in R2010b

Trigger-Based Operating Point Snapshot

Generate operating points at triggered events
Library: Simulink Control Design

Description

The Trigger-Based Operating Point Snapshot block takes snapshot operating points of a Simulink model at triggered events indicated by the input trigger signal. For example, you can configure the block to take a snapshot whenever the trigger signal crosses zero while increasing.

You can then linearize your model at the operating points using the `linearize` function or the **Model Linearizer** app.

Ports

Input

`Trigger` — Trigger control signal
scalar

Trigger control signal, specified as a scalar signal.

Parameters

`Trigger type` — Type of control signal
`rising` (default) | `falling` | `either` | `function-call`

Type of control signal that triggers the operating point snapshot.

- `rising` — Trigger when the control signal crosses zero while increasing.
- `falling` — Trigger when the control signal crosses zero while decreasing.
- `either` — Trigger when the control signal crosses zero while either increasing or decreasing.
- `function-call`: Trigger when the control signal receives a function-call event from a Stateflow chart, a Function-Call Generator block, an S-Function block, or a Hit Crossing block.

Programmatic Use
Block Parameter: `'TriggerType'`
Type: `character vector`
Values: `'rising''falling''either''function-call'`
Default: `'rising'`

See Also
`findop` | `linearize`

Topics
“About Operating Points” on page 1-2
“Linearize at Triggered Simulation Events” on page 2-75

Introduced before R2006a

Objects

BlockDiagnostic

Diagnostic information for individual block linearization

Description

When you linearize a Simulink model, you can create a `LinearizationAdvisor` object that contains `BlockDiagnostic` objects. Each `BlockDiagnostic` object contains diagnostic information regarding the linearization of the corresponding Simulink block. You can troubleshoot the block linearization by examining the `BlockDiagnostic` object properties.

Creation

To access block diagnostic information in a `LinearizationAdvisor` object, use the `getBlockInfo` function. Using this function, you can obtain either a single `BlockDiagnostic` object or multiple `BlockDiagnostic` objects. For example, see:

- “Obtain Diagnostics for Potentially Problematic Blocks” on page 20-4
- “Obtain Diagnostics Using Block Names” on page 20-4

Properties

`IsOnPath` — Flag indicating whether the block is on the linearization path
`'Yes'` | `'No'`

Flag indicating whether the block is on the linearization path, specified as one of the following:

- `'Yes'` — Block is on linearization path
- `'No'` — Block is not on linearization path

The linearization path connects the linearization inputs to the linearization outputs. To view the linearization path in the Simulink model, use the `highlight` function.

`ContributesToLinearization` — Flag indicating whether the block numerically influences the model linearization
`'Yes'` | `'No'`

Flag indicating whether the block numerically influences the model linearization, specified as one of the following:

- `'Yes'` — Block contributes to the linearization result
- `'No'` — Block does not contribute to the linearization result

If a block is not on the linearization path; that is, if `IsOnPath` is `'No'`, then `ContributesToLinearization` is `'No'`.

`DiagnosticMessages` — Diagnostic messages
cell array of character vectors

Diagnostic message regarding the block linearization, specified as a cell array of character vectors. These messages indicate possible issues that can affect the block linearization.

If `HasDiagnostics` is `'No'`, then `DiagnosticMessages` is an empty cell array.

`BlockPath` — Block path
character vector

Block path in Simulink model, specified as a character vector.

`HasDiagnostics` — Flag indicating whether the block has diagnostic messages
`'Yes'` | `'No'`

Flag indicating whether the block has diagnostic messages regarding its linearization, specified as one of the following:

- `'Yes'` — Block has diagnostic messages
- `'No'` — Block does not have diagnostic messages

If `HasDiagnostics` is `'Yes'`, then `DiagnosticMessages` is a cell array of character vectors that contains the messages.

`Linearization` — Block linearization
state-space model

Block linearization, specified as a state-space model.

`LinearizationMethod` — Linearization method
`'Exact'` | `'Perturbation'` | `'Block Substituted'` | `'Simscape Network'` | `'Not Supported'`

Linearization method, specified as one of the following:

- `'Exact'` — Block linearized using its defined exact linearization
- `'Perturbation'` — Block linearized using numerical perturbation
- `'Block Substituted'` — Block linearized using a specified custom linearization
- `'Simscape Network'` — Simscape network linearized using the exact linearization defined in the Simscape engine. A `LinearizationAdvisor` object does not provide diagnostic information on a component-level basis for Simscape networks. Instead, it groups diagnostic information together for multiple Simscape components connected to a single Solver Configuration block.
- `'Not Supported'` — Block in its current configuration does not support linearization. For example, a Discrete Transfer Fcn block with an external reset does not support linearization.

  In this case, the block `Linearization` is zero. For more troubleshooting information, check the `DiagnosticMessages` property.

`OperatingPoint` — Operating point
`BlockOperatingPoint` object

Operating point at which the block is linearized, specified as a `BlockOperatingPoint` object.

Usage

You can troubleshoot the linearization of a Simulink model by examining the diagnostics for individual block linearizations. To do so, examine the properties of `BlockDiagnostic` objects returned from `getBlockInfo`. For more information, see “Troubleshoot Linearization Results at Command Line” on page 4-28.

Examples

Obtain Diagnostics for Potentially Problematic Blocks

Load Simulink model.

```
mdl = 'scdpendulum';
load_system(mdl)
```

Linearize the model and obtain `LinearizationAdvisor` object.

```
io = getlinio(mdl);
opt = linearizeOptions('StoreAdvisor',true);
[linsys,~,info] = linearize(mdl,io,opt);
advisor = info.Advisor;
```

Find blocks that are potentially problematic for linearization.

```
blocks = advise(advisor);
```

Obtain diagnostics for these blocks.

```
diags = getBlockInfo(blocks)
```

```
diags =
Linearization Diagnostics for the Blocks:

Block Info:
-----------
Index   BlockPath                                          Is On Path   Contributes To Linearizа
1.      scdpendulum/pendulum/Saturation                    Yes          No
2.      scdpendulum/angle_wrap/Trigonometric Function1     Yes          No
3.      scdpendulum/pendulum/Trigonometric Function        Yes          No
```

Obtain Diagnostics Using Block Names

Load Simulink model.

```
mdl = 'scdpendulum';
load_system(mdl)
```

Linearize the model and obtain `LinearizationAdvisor` object.

```
io = getlinio(mdl);
opt = linearizeOptions('StoreAdvisor',true);
[linsys,~,info] = linearize(mdl,io,opt);
advisor = info.Advisor;
```

Obtain diagnostic information for the saturation block.

```
satDiag = getBlockInfo(advisor,'scdpendulum/pendulum/Saturation')
```

```
satDiag =
Linearization Diagnostics for scdpendulum/pendulum/Saturation with properties:

                     IsOnPath: 'Yes'
    ContributesToLinearization: 'No'
           LinearizationMethod: 'Exact'
                 Linearization: [1x1 ss]
               OperatingPoint: [1x1 linearize.advisor.BlockOperatingPoint]
```

You can also obtain diagnostic information for multiple blocks at once. Obtain diagnostics for the sin blocks in the model.

```
sinBlocks = {'scdpendulum/pendulum/Trigonometric Function';
             'scdpendulum/angle_wrap/Trigonometric Function1'};
```

```
sinDiag = getBlockInfo(advisor,sinBlocks)
```

```
sinDiag =
Linearization Diagnostics for the Blocks:

Block Info:
-----------
Index   BlockPath                                          Is On Path   Contributes To Lineariza
1.      scdpendulum/angle_wrap/Trigonometric Function1     Yes          No
2.      scdpendulum/pendulum/Trigonometric Function        Yes          No
```

Compatibility Considerations

Simscape states and inputs now combined into single block diagnostic per Solver Configuration block

Behavior changed in R2018a

When troubleshooting linearization issues using the Linearization Advisor, diagnostic information for Simscape states and inputs is now combined into a single block diagnostic object. Previously, states and inputs were returned in separate diagnostic objects.

Update Code

To view diagnostic information for Simscape networks at the command line, you first query the Linearization Advisor object, `advisor`, for blocks of type `'simscape'`.

```
qSS = linqueryIsBlockType('simscape');
advSS = find(advisor,qSS);
```

In R2017b, to view state or input information for the Simscape network, you searched the block paths of the diagnostics in `advSS` for the text `EVAL_KEY/STATE` or `EVAL_KEY/INPUT`, respectively. For example, to find diagnostics with state information, you used:

```
paths = getBlockPaths(advisor);
index = contains(paths,'EVAL_KEY/STATE');
diag = getBlockInfo(advSS,index);
```

You then viewed the state or input information in the associated operating point object.

```
diag(i).OperatingPoint
```

Starting in R2018a, you access both the state and input information directly from the operating point of the Simscape block diagnostic object without searching the block paths.

```
advSS.BlockDiagnostics(i).OperatingPoint
```

See Also

Objects
`LinearizationAdvisor` | `BlockOperatingPoint`

Functions
`highlight` | `getBlockInfo` | `getBlockPaths`

Topics
“Troubleshoot Linearization Results at Command Line” on page 4-28

Introduced in R2017b

BlockOperatingPoint

Operating point at which block is linearized

Description

When you linearize a Simulink model, you can create a `LinearizationAdvisor` object that contains `BlockDiagnostic` objects. Each `BlockDiagnostic` object contains diagnostic information regarding the linearization of the corresponding Simulink block. Each `BlockDiagnostic` object contains a `BlockOperatingPoint` with the input and state values for the operating point at which the block was linearized.

Creation

To obtain the operating point at which a block was linearized, use the `OperatingPoint` property of a `BlockDiagnostic` object. For example, see “Obtain Block Operating Point” on page 20-8.

Properties

`States` — Block state values
structure | structure array

State values at operating point, specified as a structure if the block has a single state, or a structure array if the block has multiple states. Each state structure has the following fields:

- `Name` — State name
- `x` — State value

`Inputs` — Block input values
structure | structure array

Input values at operating point, specified as a structure if the block has a single input, or a structure array if the block has multiple inputs. Each input structure has the following fields:

- `Port` — Input port number
- `u` — Input value

`BlockPath` — Block path
character vector

Block path in Simulink model, specified as a character vector.

Usage

When troubleshooting a block linearization, you can check the input and state values for the operating point at which the block was linearized using the `OperatingPoint` property of a `BlockDiagnostic` object.

Examples

Obtain Block Operating Point

Load Simulink model.

```
mdl = 'scdpendulum';
load_system(mdl)
```

Linearize the model and obtain a `LinearizationAdvisor` object.

```
io = getlinio(mdl);
opt = linearizeOptions('StoreAdvisor',true);
[linsys,~,info] = linearize(mdl,io,opt);
advisor = info.Advisor;
```

Obtain block diagnostics for the second block in the list. This block is a second-order integrator.

```
diags = getBlockInfo(advisor,2);
```

Obtain the operating point at which this block was linearized.

```
blockOP = diags.OperatingPoint
```

```
blockOP =
Block Operating Point for scdpendulum/pendulum/Integrator, Second-Order

States:
-------
Name        x
theta       1.5708
theta_dot   0

Inputs:
-------
Port   u
1      0.0090909
```

The block has two states and one input.

See Also

Objects
`BlockDiagnostic`

Topics
“Troubleshoot Linearization Results at Command Line” on page 4-28

Introduced in R2017b

CompoundQuery

Complex query object for finding specific blocks in linearization results

Description

`CompoundQuery` query object for finding all the blocks in a `LinearizationAdvisor` object that have a specified number of inputs.

When you linearize a Simulink model, you can create a `LinearizationAdvisor` object that contains diagnostic information about individual block linearizations. To find block linearizations that satisfy specific criteria, you can use the `find` function with custom query objects. Alternatively, you can analyze linearization diagnostics using the Linearization Advisor in the **Model Linearizer**. For more information on finding specific blocks in linearization results, see “Find Blocks in Linearization Results Matching Specific Criteria” on page 4-37.

Creation

To create a `CompoundQuery` object, combine other query objects using AND (&), OR (|), and NOT (~) logical operations. For example, see:

- “Find All SISO Blocks” on page 20-10
- “Create Complex Query Object” on page 20-10

Properties

`QueryType` — Query type
character vector

Query type, specified as a character vector. By default, `QueryType` is constructed using logical operators and the `QueryType` properties of the queries used to create the compound query. For example, suppose that you create a compound query for finding all SISO blocks:

```
qIn = linqueryHasInputs(1);
qOut = linqueryHasOutputs(1);
qSISO = qIn & qOut;
```

Then, `QueryType` is `'(Has 1 Inputs & Has 1 Outputs)'`.

You can modify `QueryType` for your application. For example:

```
qSISO.QueryType = 'SISO Blocks';
```

`Description` — Query description
`''` (default) | character vector

Query description, specified as `''` by default. You can add your own description to the query object using this property.

Object Functions

find Find blocks in linearization results that match specific criteria

Examples

Create Complex Query Object

Create a `CompundQuery` object for finding any blocks that linearize to zero or any non-SISO blocks that are on the linearization path.

Create a query object for finding all non-SISO blocks.

```
qNotSISO = ~(linqueryHasOutputs(1) & linqueryHasInputs(1));
```

Create a query object for finding all blocks on the linearization path.

```
qOnPath = linqueryIsOnPath;
```

Create a query object for finding all blocks that linearize to zero.

```
qZero = linqueryIsZero;
```

To create a query for finding any blocks that linearize to zero or any non-SISO blocks that are on the linearization path, combine the other query objects.

```
query = (qNotSISO & qOnPath) | qZero
```

```
query =
  CompoundQuery with properties:

      QueryType: '((~((Has 1 Outputs & Has 1 Inputs)) & On Linearization Path) | Linearized to Ze
    Description: ''
```

Find All SISO Blocks

Load the Simulink model.

```
mdl = 'scdspeed';
load_system(mdl)
```

Linearize the model and obtain the `LinearizationAdvisor` object.

```
opts = linearizeOptions('StoreAdvisor',true);
io(1) = linio('scdspeed/throttle (degrees)',1,'input');
io(2) = linio('scdspeed/rad//s to rpm',1,'output');
[sys,op,info] = linearize(mdl,io,opts);
advisor = info.Advisor;
```

Create compound query object for finding all blocks with one input and one output.

```
qSISO = linqueryHasInputs(1) & linqueryHasOutputs(1);
```

Find all SISO blocks using compound query object.

```
advSISO = find(advisor,qSISO)
```

```
advSISO =
  LinearizationAdvisor with properties:

               Model: 'scdspeed'
      OperatingPoint: [1x1 opcond.OperatingPoint]
    BlockDiagnostics: [1x10 linearize.advisor.BlockDiagnostic]
           QueryType: '(Has 1 Inputs & Has 1 Outputs)'
```

Alternative Functionality

App

You can also create custom queries for finding specific blocks in linearization results using the Linearization Advisor in the **Model Linearizer**. For more information, see “Find Blocks in Linearization Results Matching Specific Criteria” on page 4-37.

See Also

Objects
`LinearizationAdvisor`

Functions
`find`

Topics
“Find Blocks in Linearization Results Matching Specific Criteria” on page 4-37
“Troubleshoot Linearization Results at Command Line” on page 4-28

Introduced in R2017b

frest.Chirp

Swept-frequency cosine input signal

Description

Use a `frest.Chirp` object to represent a swept-frequency cosine input signal for frequency response estimation. A swept-frequency cosine input signal, or *chirp* signal, excites your system at a range of frequencies, such that the input frequency changes instantaneously.

Chirp signals are useful when your system is nearly linear in the simulation range. They are also useful when you want to obtain a response quickly for a lot of frequency points. The frequency-response model that results when you use a chirp input contains only frequencies that fall within the range of the chirp.

You can use a chirp input signal for estimation at the command line, in the **Model Linearizer**, or with the Frequency Response Estimator block. The estimation algorithm injects the signal at the input point you specify for estimation, and measures the response at the output point. For more information, see “Chirp Input Signals” on page 5-33.

To view a plot of your input signal, type `plot(input)`. To create a `timeseries` object for your input signal, use the `generateTimeseries` command.

Creation

Syntax

```
input = frest.Chirp(sys)
input = frest.Chirp(Name,Value)
```

Description

`input = frest.Chirp(sys)` creates a swept-frequency cosine input signal with properties based on the dynamics of the linear system `sys`. For instance, if you have an exact linearization of your system, you can use it to initialize the parameters.

`input = frest.Chirp(Name,Value)` creates a swept-frequency cosine input signal with properties on page 20-13 specified using one or more name-value pairs. Enclose each property name in quotes.

Input Arguments

`sys` — Linear dynamics system
`ss` object | `tf` object | `zpk` object

Linear dynamic system, specified as a SISO `ss`, `tf`, or `zpk` object. You can specify known dynamics or obtain the linear model 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.

The remaining properties use default values.

Properties

`Amplitude` — Signal amplitude
`1e-5` (default) | positive scalar

Signal amplitude at each frequency, specified as a positive scalar.

`FreqRange` — Signal frequency range
`[1,1000]` (default) | two-element vector | two-element cell array

Signal frequency range, specified as one of the following:

- Two-element vector, for example `[w1 w2]`
- Two-element cell array, for example `{w1 w2}`

Here, `w1` is the lower bound of the frequency range, and `w2` is the upper bound.

`FreqUnits` — Frequency units
`'rad/s'` (default) | `'Hz'`

Frequency units, specified as one of the following:

- `'rad/s'` — Radians per second
- `'Hz'` — Hertz

Changing frequency units does not impact frequency response estimation.

`SampleTime` — Sample time
positive scalar

Sample time of the chirp signal in seconds, specified as a positive scalar. The default sample time, which avoids aliasing, is:

$$\frac{2\pi}{5*\max(FreqRange)}$$

Here, *FreqRange* is specified in rad/s.

`NumSamples` — Number of samples
positive integer

Number of samples in the chirp signal, specified as a positive integer. The default number of samples, which ensures that the estimation includes the lower end of the frequency range, is:

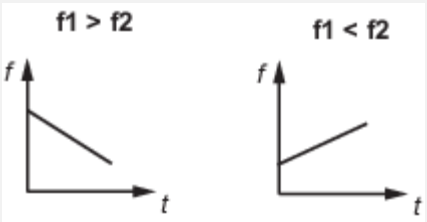
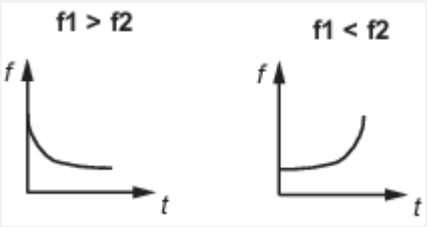
$$\frac{4\pi}{Ts*\min(FreqRange)}$$

Here, *FreqRange* is specified in rad/s.

This property does not determine number of frequency points in the final estimation result. The `frestimate` function only includes frequency points with positive values. The function also discards any frequencies that fall outside the frequency range specified for the chirp.

`SweepMethod` — Method for evolution of instantaneous frequency
`'linear'` (default) | `'logarithmic'` | `'quadratic'`

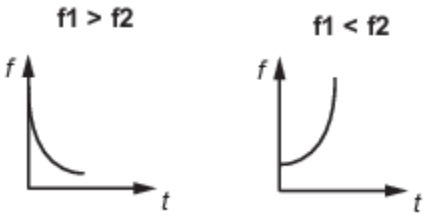
Method for evolution of instantaneous frequency, specified as one of the following.

Method	Description
`'linear'`	Specifies the instantaneous frequency sweep $f_i(t)$: $f_i(t) = f_0 + \beta t \ \ where \ \ \beta = (f_1 - f_0)/t_f$ β ensures that the signal maintains the desired frequency breakpoint f_1 at final time t_f. 
`'logarithmic'`	Specifies the instantaneous frequency sweep $f_i(t)$: $f_i(t) = f_0 \times \beta^t \ \ where \ \ \beta = \left(\frac{f_1}{f_0}\right)^{\frac{1}{t_f}}$ 
`'quadratic'`	Specifies the instantaneous frequency sweep $f_i(t)$: $f_i(t) = f_0 + \beta t^2 \ \ where \ \ \beta = (f_1 - f_0)/t_i^2$ Specify the shape of the quadratic using the `Shape` option.

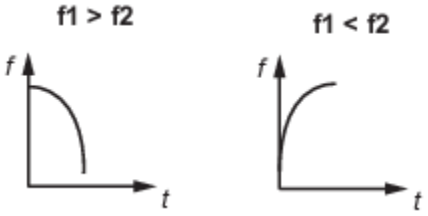
`Shape` — Quadratic sweep parabola shape
`'concave'` (default) | `convex`

Quadratic sweep parabola shape, specified as one of the following:

- `'concave'` — Concave quadratic sweeping shape.



- `'convex'` — Convex quadratic sweeping shape.



This property is available on when `SweepMethod` is `'quadratic'`.

`InitialPhase` — Initial phase of the chirp signal
`270` (default) | `scalar`

Initial phase of the Chirp signal in degrees, specified as a scalar.

Object Functions

frestimate	Frequency response estimation of Simulink models
generateTimeseries	Generate time-domain data for input signal
frest.simCompare	Plot time-domain simulation of nonlinear and linear models
frest.simView	Plot frequency response model in time- and frequency-domain
getSimulationTime	Final time of simulation for frequency response estimation

Examples

Create a Chirp Input Signal with Specified Frequency Range

Create a chirp input signal with frequencies ranging from 10 to 500 rad/s. Specify the amplitude and the number of samples as well.

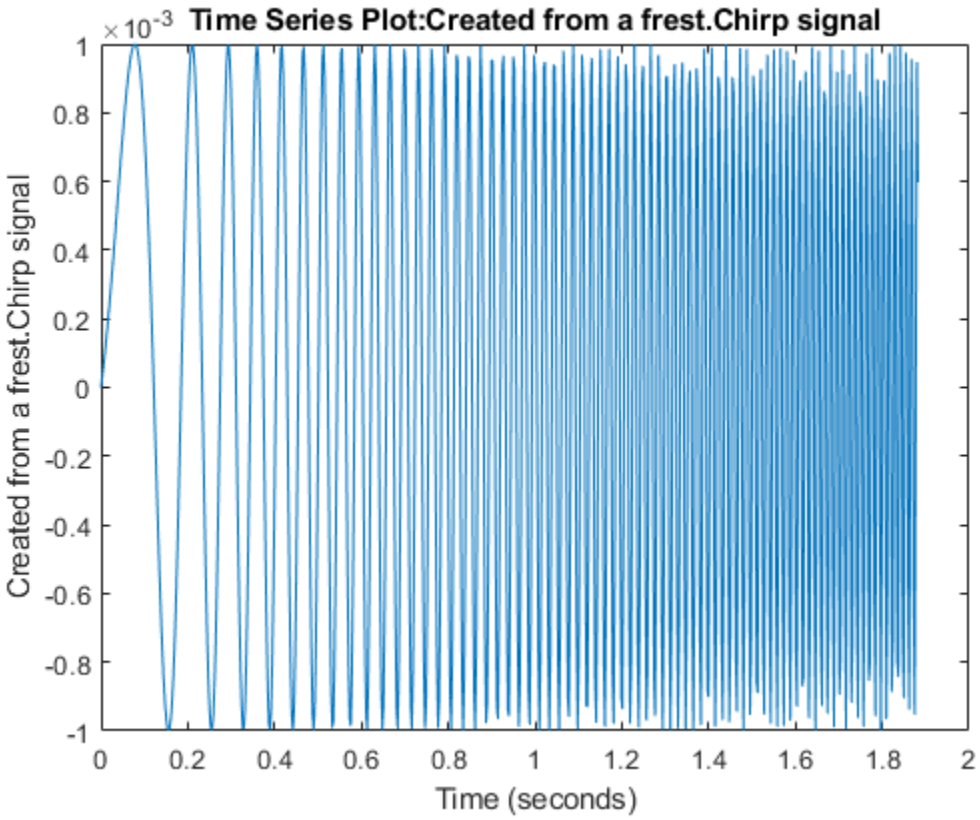
```
input = frest.Chirp('Amplitude',1e-3,...
                    'FreqRange',[10 500],...
                    'NumSamples',750)
```

```
The chirp input signal:

       FreqRange              : [10 500] (rad/s)
       Amplitude              : 0.001
       Ts                     : 0.00251327412287183 (sec)
       NumSamples             : 750
       InitialPhase           : 270 (deg)
       FreqUnits (rad/s or Hz): rad/s
       SweepMethod(linear/    : linear
```

```
                quadratic/
                logarithmic)
```

Plot the chirp signal.

```
plot(input)
```

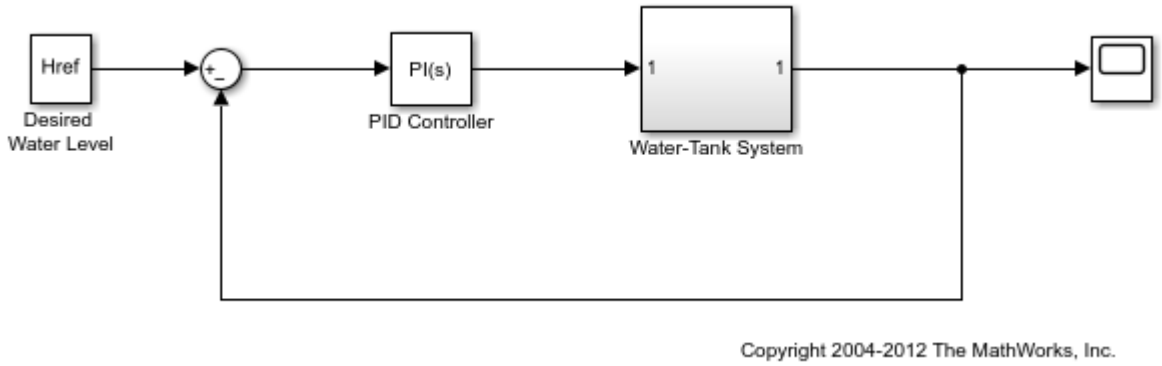


Chirp Signal Based on Linearized Dynamics

Create a chirp input signal based on the dynamics of a linear system. This approach is useful when you are using frequency response estimation to validate the linearization of your model.

Open a Simulink model.

```
model = 'watertank';
open_system(model)
```



Copyright 2004-2012 The MathWorks, Inc.

For this example, linearize the model at a steady-state operating point to obtain a state-space model you can use to initialize the chirp signal.

```
io(1)=linio('watertank/PID Controller',1,'input');
io(2)=linio('watertank/Water-Tank System',1,'openoutput');

watertank_spec = operspec(model);
opOpts = findopOptions('DisplayReport','off');
op = findop(model,watertank_spec,opOpts);

sys = linearize(model,op,io);
```

Create the chirp signal.

```
input = frest.Chirp(sys);
```

`frest.Chirp` chooses a frequency range based on the system dynamics. It also automatically initializes other parameters of the chirp signal.

```
input
```

```
The chirp input signal:

      FreqRange              : [0.001581138830107 0.1581138830107] (rad/s)
      Amplitude              : 1e-05
      Ts                     : 7.94767061252222 (sec)
      NumSamples             : 1000
      InitialPhase           : 270 (deg)
      FreqUnits (rad/s or Hz): rad/s
      SweepMethod(linear/    : linear
                  quadratic/
                  logarithmic)
```

You can change properties of the signal using dot notation. For instance, increase the signal amplitude.

```
input.Amplitude = 3e-5
```

```
The chirp input signal:
```

```
FreqRange              : [0.001581138830107 0.1581138830107] (rad/s)
Amplitude              : 3e-05
Ts                     : 7.94767061252222 (sec)
NumSamples             : 1000
InitialPhase           : 270 (deg)
FreqUnits (rad/s or Hz): rad/s
SweepMethod(linear/    : linear
            quadratic/
            logarithmic)
```

Alternative Functionality

Model Linearizer

In the **Model Linearizer**, to use a chirp input signal for estimation, on the **Estimation** tab, select **Input Signal > Chirp**.

See Also

`frest.Sinestream` | `frest.Random` | `frestimate`

Topics

“Estimation Input Signals” on page 5-24
“Sinestream Input Signals” on page 5-29
“Estimate Frequency Response at the Command Line” on page 5-14
“Speeding Up Estimation Using Parallel Computing” on page 5-69

Introduced in R2009b

frest.Random

Random input signal

Description

Use a `frest.Random` object to represent a random input signal for frequency response estimation. The random signal contains uniformly distributed random numbers in the interval `[0 Amplitude]` or `[Amplitude 0]` for positive and negative amplitudes, respectively.

Random signals are useful because they can excite the system uniformly at all frequencies up to the Nyquist frequency.

You can use a random input signal for estimation at the command line, in the **Model Linearizer**, or with the Frequency Response Estimator block. The estimation algorithm injects the signal at the input point you specify for estimation, and measures the response at the output point.

When you use a random input signal for estimation, the frequencies returned in the estimated `frd` model depend on the length and sampling time of the signal. They are the frequencies obtained in the fast Fourier transform of the input signal. For more information, see the Algorithm section of `frestimate`.

To view a plot of your input signal, type `plot(input)`. To create a `timeseries` object for your input signal, use the `generateTimeseries` command.

Creation

Syntax

```
input = frest.Random(sys)
input = frest.Random(Name,Value)
```

Description

`input = frest.Random(sys)` creates a random signal with properties based on the dynamics of the linear system `sys`. For instance, if you have an exact linearization of your system, you can use it to initialize the parameters.

`input = frest.Random(Name,Value)` creates random signal with properties on page 20-20 specified using one or more name-value pairs. Enclose each property name in quotes.

Input Arguments

`sys` — Linear dynamics system
`ss` object | `tf` object | `zpk` object

Linear dynamic system, specified as a SISO `ss`, `tf`, or `zpk` object. You can specify known dynamics or obtain the linear model by linearizing a nonlinear system.

The resulting `frest.Random` object automatically sets the following properties 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.

The remaining properties use default values.

Properties

`Amplitude` — Signal amplitude

`1e-5` (default) | nonzero scalar

Signal amplitude, specified as a scalar. If `Amplitude` is:

- Positive, the random signal values are uniformly distributed in the range `[0 Amplitude]`
- Negative, the random signal values are uniformly distributed in the range `[Amplitude 0]`

`Ts` — Sample time

`1e-3` (default) | positive scalar

Sample time of the random signal in seconds, specified as a positive scalar.

`NumSamples` — Number of samples

`1e4` (default) | positive integer

Number of samples in the random signal, specified as a positive integer.

This property does not determine number of frequency points in the final estimation result. The `frestimate` function discards any frequency points with negative values for this signal.

`Stream` — Random number stream

`RandStream` object

Random number stream, specified as a `RandStream` object. The state of the stream you specify is stored 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.

By default, `Stream` is the default stream of the current MATLAB session.

Object Functions

frestimate	Frequency response estimation of Simulink models
generateTimeseries	Generate time-domain data for input signal
frest.simCompare	Plot time-domain simulation of nonlinear and linear models
frest.simView	Plot frequency response model in time- and frequency-domain
getSimulationTime	Final time of simulation for frequency response estimation

Examples

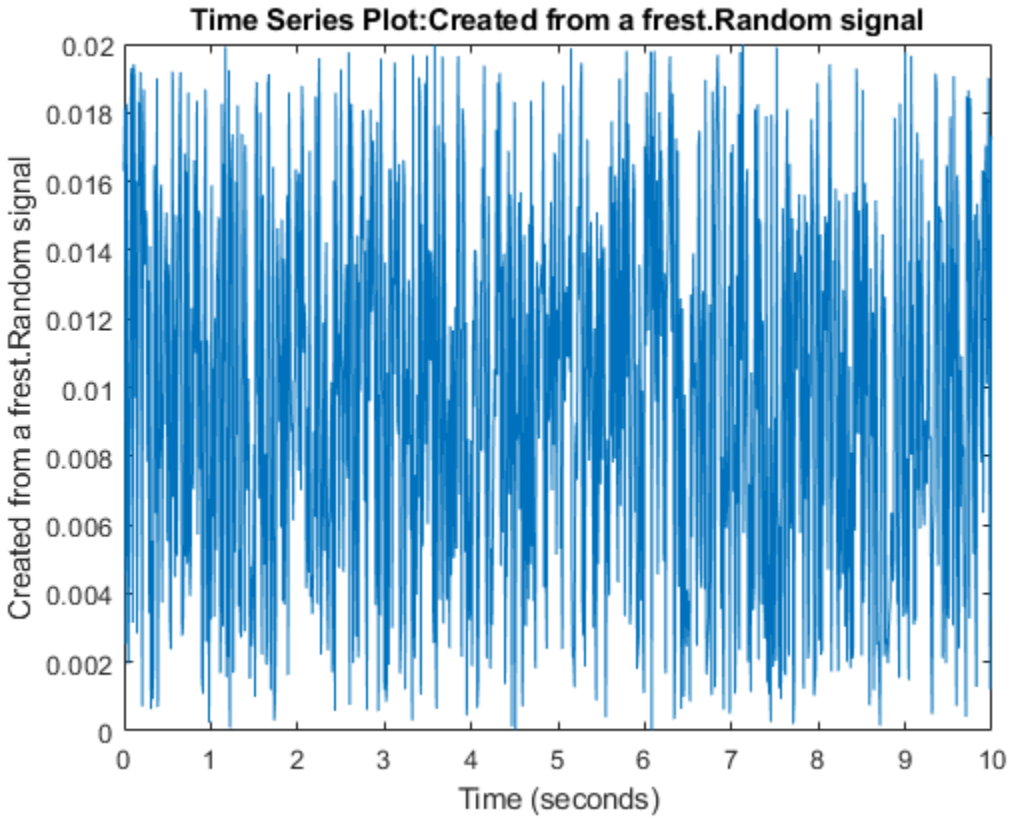
Create Random Input Signal

Create a Random input signal with `1000` samples taken at `100` Hz and an amplitude of `0.02`.

```
input = frest.Random('Amplitude',0.02,...
                     'Ts',1/100,...
                     'NumSamples',1000);
```

Plot the random signal.

```
plot(input)
```



Create Random Signal Using Specified Random Stream

Create a multiplicative lagged Fibonacci generator random stream.

```
stream = RandStream('mlfg6331_64','Seed',0);
```

Create a random input signal using this stream.

```
input = frest.Random('Stream',stream);
```

Alternative Functionality

Model Linearizer

In the **Model Linearizer**, to use a random input signal for estimation, on the **Estimation** tab, select **Input Signal** > **Random**.

See Also

`frest.Sinestream` | `frest.Chirp` | `frestimate`

Topics

“Estimation Input Signals” on page 5-24
“Sinestream Input Signals” on page 5-29
“Estimate Frequency Response at the Command Line” on page 5-14
“Estimate Frequency Response Using Model Linearizer” on page 5-6
“Speeding Up Estimation Using Parallel Computing” on page 5-69

Introduced in R2009b

frest.Sinestream

Input signal containing series of sine waves

Description

Use a `frest.Sinestream` object to represent a sinestream input signal for frequency response estimation. Such a signal consists of sine waves of varying frequencies applied one after another. Each frequency excites the system for a period of time.

Sinestream signals are recommended for most situations. They are especially useful when your system contains strong nonlinearities or you require highly accurate frequency response models. The frequency-response model that results when you use a sinestream input contains all the frequencies in the sinestream signal

You can use a sinestream input signal for estimation at the command line, in the **Model Linearizer**, or with the Frequency Response Estimator block. The estimation algorithm injects the sinestream signal at the input point you specify for estimation, and measures the response at the output point. For more information, see “Sinestream Input Signals” on page 5-29.

To view a plot of your input signal, type `plot(input)`. To create a `timeseries` object for your input signal, use the `generateTimeseries` command.

Creation

You can create a sinestream signal in one of the following ways:

- Using the `frest.Sinestream` function for continuous-time signals
- Using the `frest.createFixedTsSinestream` function for discrete-time signals

For more information, see “Sinestream Input Signals” on page 5-29.

Syntax

```
input = frest.Sinestream(sys)
input = frest.Sinestream(Name,Value)
```

Description

`input = frest.Sinestream(sys)` creates a signal with a series of sinusoids with properties based on the dynamics of the linear system `sys`. For instance, if you have an exact linearization of your system, you can use it to initialize the parameters.

`input = frest.Sinestream(Name,Value)` creates a signal with a series of sinusoids with properties on page 20-24 specified using one or more name-value pairs. Enclose each property name in quotes.

Input Arguments

sys — Linear dynamics system

`ss` object | `tf` object | `zpk` object

Linear dynamic system, specified as a SISO `ss`, `tf`, or `zpk` object. You can specify known dynamics or obtain the linear model by linearizing a nonlinear system.

The resulting `frest.Sinestream` object automatically sets the following properties based on the linear system:

- `Frequency` contains 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 the maximum 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.

The remaining properties use default values.

Properties

Frequency — Signal frequencies

`logspace(1,3,50)` (default) | vector | scalar

Signal frequencies, specified as a vector of frequency values in units specified by `FreqUnits`.

Amplitude — Signal amplitude

`1e-5` (default) | scalar | vector

Signal amplitude at each frequency, specified as one of the following:

- Scalar — Set all frequencies to same amplitude.
- Vector with length equal to the length of `Frequency` — Set amplitude for each frequency to a different value.

SamplesPerPeriod — Number of samples per period

`40` (default) | scalar | vector

Number of samples per period for each frequency, specified as one of the following:

- Scalar — Use the same number of samples per period for all frequencies.
- Vector with length equal to the length of `Frequency` — Use a different number of samples for each frequency.

FreqUnits — Frequency units

`'rad/s'` (default) | `'Hz'`

Frequency units, specified as one of the following:

- `'rad/s'` — Radians per second

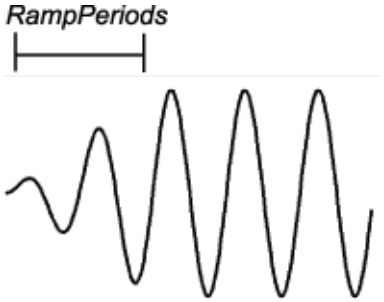
- `'Hz'` — Hertz

`RampPeriods` — Number of periods for ramping up the amplitude of each sine wave to its maximum value
`0` (default) | scalar | vector

Number of periods for ramping up the amplitude of each sine wave to its maximum value, specified as one of the following:

- Scalar — Use the same number of ramping up periods for all frequencies.
- Vector with length equal to the length of `Frequency` — Use a different number of ramping up periods for each frequency.

Use `RampUpPeriods` to specify the number of periods over which to linearly increase the amplitude of each sine wave to its maximum value. Specifying this option ensures a smooth response when your input amplitude changes.



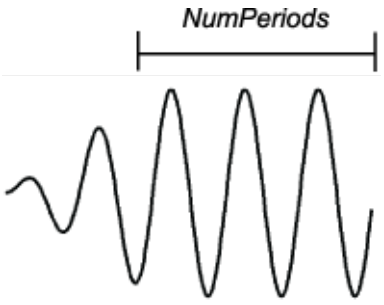
`frestimate` discards response data collected during the ramping up periods.

`NumPeriods` — Number of periods each sine wave is at maximum amplitude
`max(3 - RampPeriods + SettlingPeriods,2)` (default) | scalar | vector

Number of periods each sine wave is at maximum amplitude, specified as one of the following:

- Scalar — Use the same number of periods for all frequencies.
- Vector with length equal to the length of `Frequency` — Use a different number of periods for each frequency.

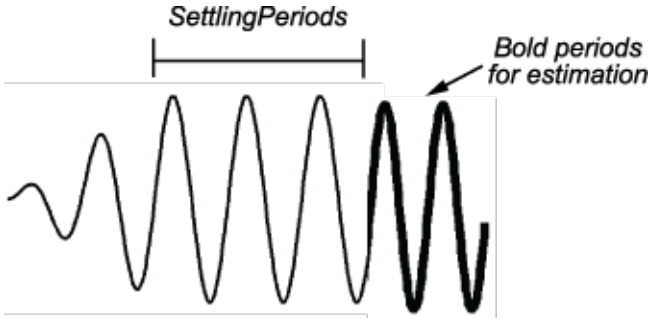
The specified number of periods includes the settling periods (`SettlingPeriods`) and the periods used for estimation.



`SettlingPeriods` — Number of periods before the system reaches steady state
`1` (default) | scalar | vector

Number of periods before the system reaches steady state, specified as one of the following:

- Scalar — Use the same number of settling periods for all frequencies.
- Vector with length equal to the length of `Frequency` — Use a different number of settling periods for each frequency.



`frestimate` discards response data collected during the settling periods.

`ApplyFilteringInFRESTIMATE` — Flag to apply frequency-selective FIR filtering of the input signal
`'on'` (default) | `'off'`

Flag to apply frequency-selective FIR filtering of the input signal before estimating it using `frestimate`, specified as one of the following:

- `'on'` — Filter the input signal. When you use filtering, `frestimate` discards response data for one additional period after the settling periods before estimation.
- `'off'` — Do not filter the input signal.

`SimulationOrder` — Order in which individual input signal frequencies are injected
`'Sequential'` (default) | `'OneAtATime'`

Order in which individual input signal frequencies are injected into your Simulink model during simulation, specified as one of the following:

- `'Sequential'` — `frestimate` injects one frequency after the next into your model in a single Simulink simulation using variable sample time. To use this option, your 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 model to the operating point specified for estimation. If you have Parallel Computing Toolbox software, you can run these simulations in parallel to speed up estimation. For more information, see “Speeding Up Estimation Using Parallel Computing” on page 5-69.

Object Functions

frestimate	Frequency response estimation of Simulink models
generateTimeseries	Generate time-domain data for input signal
frest.simCompare	Plot time-domain simulation of nonlinear and linear models
frest.simView	Plot frequency response model in time- and frequency-domain
getSimulationTime	Final time of simulation for frequency response estimation

Examples

Create Sinestream Signal by Specifying Frequencies

Create a sinestream input signal for estimation by specifying the frequencies for the signal. Also, specify the amplitude, the number of ramp-up periods, the number of settling periods, and the total number of periods after the ramp-up.

To specify the frequencies, use a vector of frequencies.

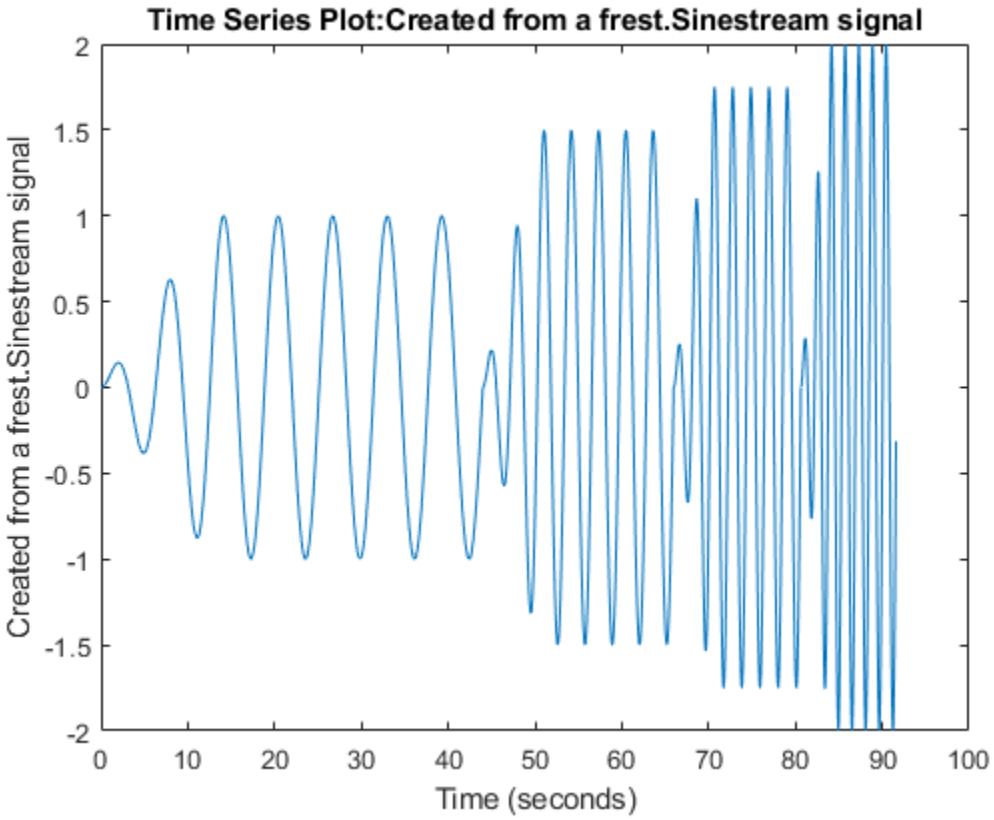
```
freqs = linspace(1,4,4);
```

To specify the other parameters, use a scalar to use the same parameter value for every frequency. To use different values for each frequency, use a vector of the same length as `freqs`. For this example, use increasing amplitudes at each frequency, but keep the number of ramp-up periods, number of settling periods, and the number of periods after ramp-up constant.

```
amps = [1 1.5 1.75 2];
ramp = 2;
settle = 3;
pds = 5;

input = frest.Sinestream('Frequency',freqs,...
                         'Amplitude',amps,...
                         'RampPeriods',ramp,...
                         'SettlingPeriods',settle,...
                         'NumPeriods',pds);
```

Examine the resulting sinestream signal.

```
plot(input)
```



Sinestream Input with Specified Number of Samples

When your sinestream signal covers a wide range of frequencies, it can be inefficient to use the same sample time across all frequencies. For that reason, `frest.Sinestream` by default uses a fixed number of samples at each frequency. You can specify that number with a scalar value, or use a vector to provide a different number of samples at each frequency. (To create a sinestream signal with a fixed sample time across the entire signal, use `frest.createFixedTsSinestream`. This option is useful when the input linearization point for estimation is on a discrete-time signal.)

Create a sinusoidal input signal with the following characteristics:

- 50 frequencies spaced logarithmically between 10 Hz and 1000 Hz
- Amplitude of 1e-3 at all frequencies
- Sampled with a frequency 10 times the frequency of the signal (meaning ten samples per period)

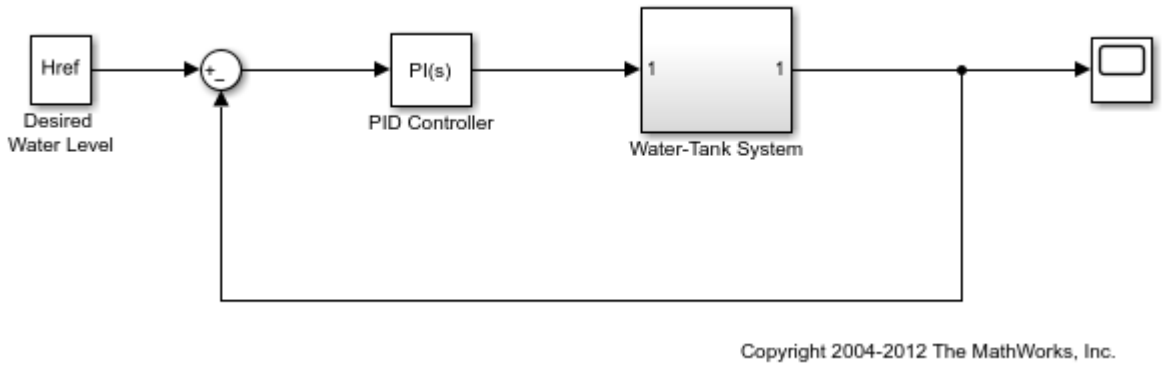
```
input = frest.Sinestream('Amplitude',1e-3,...
                         'Frequency',logspace(1,3,50),...
                         'SamplesPerPeriod',10,...
                         'FreqUnits','Hz');
```

Sinestream Signal Based on Linearized Dynamics

Create a sinestream input signal based on the dynamics of a linear system. This approach is useful when you are using frequency response estimation to validate the linearization of your model.

Open a Simulink model.

```
model = 'watertank';
open_system(model)
```



For this example, linearize the model at a steady-state operating point to obtain a state-space model you can use to initialize the sinestream signal.

```
io(1)=linio('watertank/PID Controller',1,'input');
io(2)=linio('watertank/Water-Tank System',1,'openoutput');

watertank_spec = operspec(model);
opOpts = findopOptions('DisplayReport','off');
op = findop(model,watertank_spec,opOpts);

sys = linearize(model,op,io);
```

Create the sinestream signal.

```
input = frest.Sinestream(sys);
```

`frest.Sinestream` chooses frequencies based on the system dynamics. It also automatically initializes other parameters of the sinestream signal.

```
input
```

```
The sinestream input signal:

      Frequency           : [0.0015811;0.0026375;0.0043996;0.007339 ...] (rad/s)
      Amplitude           : 1e-05
      SamplesPerPeriod    : 40
      NumPeriods          : [4;4;4;4 ...]
      RampPeriods         : 0
      FreqUnits (rad/s,Hz): rad/s
      SettlingPeriods     : [1;1;1;1 ...]
      ApplyFilteringInFRESTIMATE (on/off)    : on
      SimulationOrder (Sequential/OneAtATime): Sequential
```

You can change properties of the signal using dot notation. For instance, increase the signal amplitude.

```
input.Amplitude = 3e-5
```

```
The sinestream input signal:

      Frequency           : [0.0015811;0.0026375;0.0043996;0.007339 ...] (rad/s)
      Amplitude           : 3e-05
      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
```

Alternative Functionality

Model Linearizer

In the **Model Linearizer**, to use a sinestream input signal for estimation, on the **Estimation** tab, select:

- **Input Signal** > **Sinestream** when the sample time of the I/Os is continuous.
- **Input Signal** > **Fixed Sample Time Sinestream** when the sample time of the I/Os is discrete.

See Also

`frest.Chirp` | `frest.Random` | `frestimate`

Topics

“Estimation Input Signals” on page 5-24
“Sinestream Input Signals” on page 5-29
“Estimate Frequency Response at the Command Line” on page 5-14
“Speeding Up Estimation Using Parallel Computing” on page 5-69

Introduced in R2009b

frest.PRBS

Pseudorandom binary sequence input signal

Description

Use a `frest.PRBS` object to represent a pseudorandom binary sequence (PRBS) input signal for frequency response estimation. A PRBS signal is a deterministic signal that shifts between two values and has white-noise-like properties. A PRBS signal is inherently periodic with a maximum period length of 2^n–1, where n is the PRBS order.

PRBS signals reduce total estimation time compared to using sinestream input signals, while producing comparable estimation results. PRBS signals are useful for estimating frequency responses for communications and power electronics systems with high-frequency switching components, such as pulse-width modulation (PWM) generators.

You can use a PRBS input signal for estimation at the command line or in the **Model Linearizer** app. The estimation algorithm injects the PRBS signal at the input point you specify for estimation and measures the response at the output point. For more information, see “PRBS Input Signals” on page 5-36.

To view a plot of your input signal, type `plot(input)`. To create a `timeseries` object for your input signal, use the `generateTimeseries` command.

Creation

Syntax

```
input = frest.PRBS(sys)
input = frest.PRBS(Name,Value)
```

Description

`input = frest.PRBS(sys)` creates a PRBS signal with parameters based on the dynamics of the linear system `sys`. For instance, if you have an exact linearization of your system, you can use it to initialize the parameters.

`input = frest.PRBS(Name,Value)` creates a PRBS signal with properties on page 20-24 specified using one or more name-value pairs. Enclose each property name in quotes.

Input Arguments

`sys` — Linear dynamic system
`ss` object | `tf` object | `zpk` object

Linear dynamic system, specified as a SISO `ss`, `tf`, or `zpk` object. You can specify known dynamics or obtain the linear model by linearizing a nonlinear system.

The resulting `frest.PRBS` object automatically sets the `Order` and `Ts` properties based on the linear system. The `Amplitude` and `NumPeriods` properties remain at their default values. For more information, see “PRBS Input Signals” on page 5-36.

Properties

`Amplitude` — Signal amplitude

`1e-5` (default) | positive scalar

Signal amplitude, specified as a positive scalar. You must set the amplitude such that the system is properly excited for your application. If the input amplitude is too large, the signal can deviate too far from the model operating point. If the input amplitude is too small, the PRBS signal is indistinguishable from noise and ripples in your model.

`Ts` — Signal sample time

`0.001` (default) | positive scalar

Signal sample time in seconds, specified as a positive scalar. As a starting point, specify the PRBS sample time to match the sample time of your model.

For some systems, using a larger sample time than in the original model can produce a higher resolution frequency response estimation result over the low-frequency range. In this case, you must ensure that the frequency of your model at the input and output linear analysis points matches the value you specify for `Ts`. For an example, see “Frequency Response Estimation for Power Electronics Model Using Pseudorandom Binary Signal” on page 5-94.

`Order` — Signal order

`10` (default) | positive integer

Signal order, specified as a positive integer. The maximum length of the PRBS signal is 2^n–1, where n is the signal order. To obtain an accurate frequency response estimation, the length of the PRBS must be sufficiently large.

For a given sample time, to obtain a higher frequency resolution, specify a larger signal order.

`NumPeriods` — Number of periods

`1` (default) | positive integer

Number of periods in the PRBS signal, specified as a positive integer. For most frequency response estimation applications, use the default value of `1`. Using a single period produces a flat frequency profile across the input signal frequency range.

`OneSamplePerClockPeriod` — Option to keep signal constant for either one or multiple samples per clock period

`'on'` (default) | `'off'`

Option to keep the input signal constant for either one sample or multiple samples per clock period when you have `NumPeriods` > 1, specified as one of the following:

- `'on'` — Keep the signal constant for one sample.
- `'off'` — Keep the signal constant for a number of samples equal to the value specified in `NumPeriods`.

`UseWindow` — Option to apply window-based filtering of input signal

`'on'` (default) | `'off'`

Option to apply Hann window-based filtering for single-period signals or conduct down-sampling for multi-period signals, specified as `'on'` or `'off'`.

For a single-period signal, use this option to apply Hann window-based filtering of the logged input and output signals, specified as one of the following:

- `'on'` — Filter the input and output signals. When you use filtering, `frestimate` produces smoother frequency response estimation results.
- `'off'` — Do not filter the input signal.

For a multi-period signal, use this option to conduct down-sampling of the frequency response estimation result, specified as one of the following:

- `'on'` — Resample the frequency response estimation result. When you conduct down-sampling, `frestimate` produces smoother frequency response estimation results.
- `'off'` — Do not down-sample the frequency response estimation result.

Object Functions

frestimate	Frequency response estimation of Simulink models
generateTimeseries	Generate time-domain data for input signal
frest.simCompare	Plot time-domain simulation of nonlinear and linear models
frest.simView	Plot frequency response model in time- and frequency-domain
getSimulationTime	Final time of simulation for frequency response estimation

Examples

Create PRBS Signal by Specifying Parameters

Create a PRBS with the following configuration.

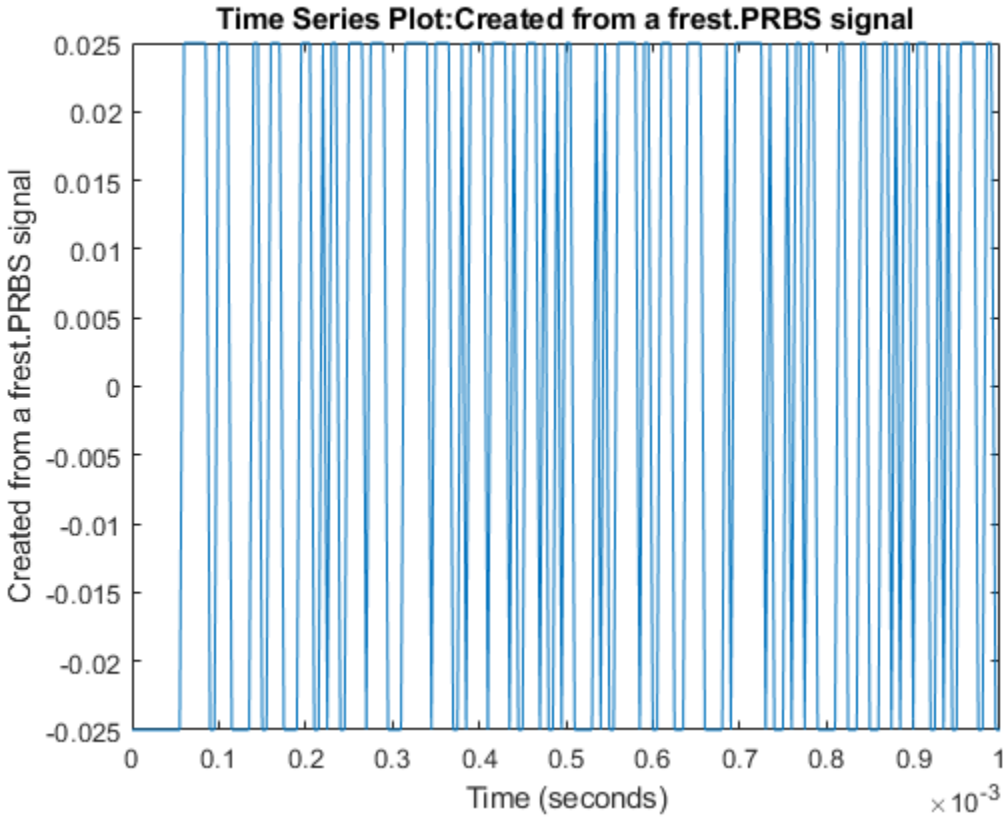
- To use a nonperiodic PRBS set the number of periods to `1`.
- Use a PRBS order of `12`, producing a signal of length `4095`.
- Set the sample time of the signal to `5e-6` seconds.
- Set the perturbation amplitude to `0.05`.

```
input = frest.PRBS('Order',12,'NumPeriods',1,'Amplitude',0.05,'Ts',5e-6)
```

```
The PRBS input signal:

      Amplitude  : 0.05
      Ts         : 5e-06 (secs)
      Order      : 12
      NumPeriods : 1
      OneSamplePerClockPeriod (on/off)    : on
      UseWindow (on/off) : on
```

Examine a subset of the resulting PRBS signal.

```
plot(input)
xlim([0 0.001])
```



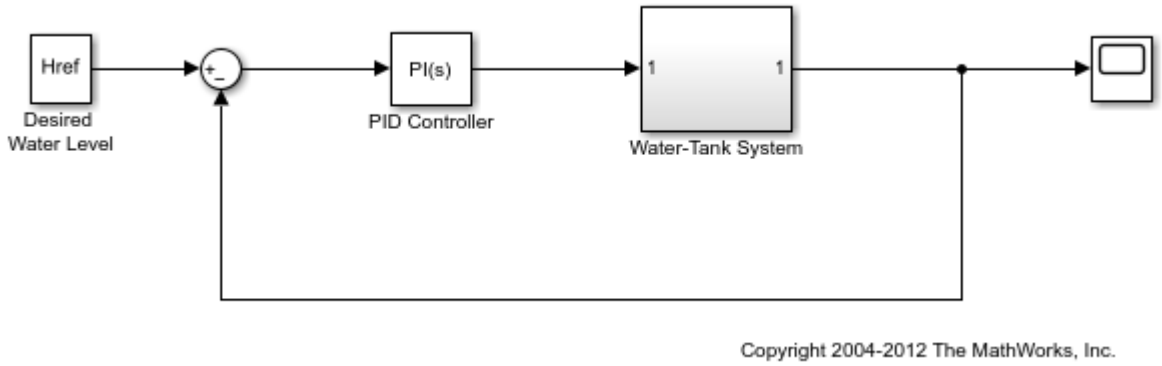
The signal switches between 0.025 and -0.025 in a deterministic pseudorandom manner.

Create PRBS Signal Based on Linearized Dynamics

Create a PRBS input signal based on the dynamics of a linear system. This approach is useful when you are using frequency response estimation to validate the linearization of your model.

Open a Simulink model.

```
model = 'watertank';
open_system(model)
```



Copyright 2004-2012 The MathWorks, Inc.

For this example, linearize the model at a steady-state operating point to obtain a state-space model you can use to initialize the PRBS signal.

```
io(1) = linio('watertank/PID Controller',1,'input');
io(2) = linio('watertank/Water-Tank System',1,'openoutput');

watertank_spec = operspec(model);
opOpts = findopOptions('DisplayReport','off');
op = findop(model,watertank_spec,opOpts);

sys = linearize(model,op,io);
```

Create the PRBS signal.

```
input = frest.PRBS(sys);
```

`frest.PRBS` configures the order and sample time of the input signal based on the system dynamics. The amplitude and number of periods remain at their default values.

```
input
```

```
The PRBS input signal:

      Amplitude  : 1e-05
      Ts         : 7.94767061252222 (secs)
      Order      : 8
      NumPeriods : 1
      OneSamplePerClockPeriod (on/off)    : on
      UseWindow (on/off) : on
```

You can change properties of the signal using dot notation. For instance, increase the signal amplitude.

```
input.Amplitude = 3e-5
```

```
The PRBS input signal:

      Amplitude  : 3e-05
      Ts         : 7.94767061252222 (secs)
      Order      : 8
```

```
NumPeriods : 1
OneSamplePerClockPeriod (on/off)    : on
UseWindow (on/off) : on
```

Alternative Functionality

Model Linearizer

In the **Model Linearizer**, to use a PRBS input signal for estimation, on the **Estimation** tab, select **Input Signal** > **PRBS Pseudorandom Binary Sequence**

Compatibility Considerations

Additional properties `OneSamplePerClockPeriod` and `UseWindow`

Behavior changed in R2021a

Starting in R2021a, the `frest.PRBS` object has two additional properties, `OneSamplePerClockPeriod` and `UseWindow`.

The `OneSamplePerClockPeriod` property specifies whether the signal remains constant for one sample per clock period or multiple samples per clock period when you have `NumPeriods` > 1. By default, this property is set to `'on'` and the signal remains constant for one sample.

`UseWindow` property provides an option to apply filtering to the input and output signals to produce a smoother frequency response estimation result. By default, this property is set to `'on'` and signals are filtered.

If you have existing code, the frequency response estimation results might differ. To use same settings as the previous releases, set these properties to `'off'`.

See Also

`frest.Chirp` | `frest.Random` | `frest.Sinestream` | `frestimate`

Topics

“PRBS Input Signals” on page 5-36
“Estimation Input Signals” on page 5-24
“Estimate Frequency Response at the Command Line” on page 5-14
“Frequency Response Estimation Using Simulation-Based Techniques” on page 5-74

Introduced in R2020a

LinearizationAdvisor

Diagnostic information for troubleshooting linearization results

Description

When you linearize a Simulink model, you can create a `LinearizationAdvisor` object that contains diagnostic information about individual block linearizations. You can troubleshoot your linearization results by reviewing this diagnostic information.

To access the diagnostic information, use the `getBlockInfo` function.

Creation

There are several ways to create a `LinearizationAdvisor` object when linearizing a Simulink model. When you linearize a model using:

- The `linearize` function, first create a `linearizeOptions` option set, setting the `StoreAdvisor` option to `true`. Then, linearize the model using `linearize`, returning the `info` argument.
- An `slLinearizer` interface, first create a `linearizeOptions` option set, setting the `StoreAdvisor` option to `true`. Then, create the `slLinearizer` interface. When you obtain a linear model from the interface using a linearization function, such as `getIOTransfer`, return the `info` argument.
- An `slTuner` interface, first create a `slTunerOptions` option set, setting the `StoreAdvisor` option to `true`. Then, create the `slTuner` interface. When you obtain a linear model from the interface using a linearization function, such as `getIOTransfer`, return the `info` argument.

You can then access the `LinearizationAdvisor` object using `info.Advisor`. If you linearize the model at multiple operating points or using parameter variation, `info.Advisor` is an array of `LinearizationAdvisor` objects.

Also, the `advise` and `find` functions return a `LinearizationAdvisor` object that contains diagnostic information for blocks that satisfy the relevant search criteria.

Properties

`Model` — Simulink model
character vector

Simulink model associated with the linearization diagnostic information, returned as a character vector.

`Model` is a read-only property.

`AnalysisPoints` — Linear analysis points
linearization I/O object | vector of linearization I/O objects

Linear analysis points, including inputs, outputs, and openings, returned as a linearization I/O object or a vector of linearization I/O objects.

`AnalysisPoints` corresponds to the:

- `io` input argument of the `linearize` command.
- Analysis points and loop openings of an `slLinearizer` or `slTuner` interface.

For more information on analysis points, see “Specify Portion of Model to Linearize” on page 2-10.

`AnalysisPoints` is a read-only property.

`OperatingPoint` — Operating point
operating point object

Operating point at which the model was linearized, specified as an operating point object.

`OperatingPoint` is a read-only property.

`Parameters` — Parameter samples
`[]` (default) | structure | structure array

Parameter samples for linearization, specified as one of the following:

- `[]` — Linearization result has no associated parameter values.
- Structure — Value for a single parameter, specified as a structure with the following fields:
  - `Name` — Parameter name
  - `Value` — Parameter value
- Structure array — Values for multiple parameters.

For more information on parameter variation, see “Specify Parameter Samples for Batch Linearization” on page 3-43.

`Parameters` is a read-only property.

`LinearizationOptions` — Linearization algorithm options
`linearizeOptions` option set

Linearization algorithm options, specified as a `linearizeOptions` object.

`LinearizationOptions` corresponds to the `options` input argument of `linearize`, `slLinearizer`, or `slTuner`.

`LinearizationOptions` is a read-only property.

`BlockDiagnostics` — Diagnostic information
`BlockDiagnostic` object | vector of `BlockDiagnostic` objects

Diagnostic information for each block that matches the search criteria used to create the `LinearizationAdvisor` object, specified as a `BlockDiagnostic` object or a vector of `BlockDiagnostic` objects.

You can access these block diagnostics using the `getBlockInfo` command. To obtain a list of the blocks, use the `getBlockPaths` command.

`BlockDiagnostics` is a read-only property.

`QueryType` — Query type
character vector

Query type used to obtain the linearization diagnostics, specified as one of the following:

- `'All Blocks'` when you initially create a `LinearizationAdvisor` object using a linearization function such as `linearize` or `getIOTransfer`.
- `'Linearization Advice'` when you create a `LinearizationAdvisor` object using the `advise` command.
- A character vector matching the `QueryType` property of the corresponding custom query object when you create a `LinearizationAdvisor` object using the `find` command.

`QueryType` is a read-only property.

`Description` — Query description
character vector

Description of the query used to obtain the linearization diagnostics, specified as one of the following:

- `'All Linearized Blocks'` when you initially create a `LinearizationAdvisor` object using a linearization function such as `linearize` or `getIOTransfer`.
- `'Blocks that are Potentially Problematic for Linearization'` when you create a `LinearizationAdvisor` object using the `advise` command.
- A character vector matching the `Description` property of the corresponding custom query object when you create a `LinearizationAdvisor` object using the `find` command.

`Description` is a read-only property.

Object Functions

advise	Find blocks that are potentially problematic for linearization
highlight	Highlight linearization path in Simulink model
find	Find blocks in linearization results that match specific criteria
getBlockInfo	Obtain diagnostic information for block linearizations
getBlockPaths	Obtain list of blocks in LinearizationAdvisor object

Examples

Create `LinearizationAdvisor` Using `linearize`

Load Simulink model.

```
mdl = 'scdpendulum';
load_system(mdl)
```

Create a linearization option set, enabling the `StoreAdvisor` option.

```
opt = linearizeOptions('StoreAdvisor',true);
```

Linearize the model using this option set, returning the `info` argument.

```
io = getlinio(mdl);
[linsys,~,info] = linearize(mdl,io,opt);
```

Extract the `LinearizationAdvisor` object from `info`.

```
advisor = info.Advisor
```

```
advisor =
  LinearizationAdvisor with properties:

               Model: 'scdpendulum'
      OperatingPoint: [1x1 opcond.OperatingPoint]
    BlockDiagnostics: [1x11 linearize.advisor.BlockDiagnostic]
           QueryType: 'All Blocks'
```

Create `LinearizationAdvisor` Using `slLinearizer` Interface

Load Simulink model.

```
mdl = 'scdspeed';
load_system(mdl)
```

Create a linearization option set, enabling the `StoreAdvisor` option.

```
opt = linearizeOptions('StoreAdvisor',true);
```

Define input and output analysis points, and create an `slLinearizer` interface using this option set.

```
io(1) = linio('scdspeed/throttle (degrees)',1,'input');
io(2) = linio('scdspeed/rad//s to rpm',1,'output');
SL = slLinearizer(mdl,io,opt);
```

Find the transfer function from the input to the output, returning the `info` argument.

```
[linsys,info] = getIOTransfer(SL,'scdspeed/throttle (degrees)','scdspeed/rad//s to rpm');
```

Extract the `LinearizationAdvisor` object from `info`.

```
advisor = info.Advisor
```

```
advisor =
  LinearizationAdvisor with properties:

               Model: 'scdspeed'
      OperatingPoint: [1x1 opcond.OperatingPoint]
    BlockDiagnostics: [1x27 linearize.advisor.BlockDiagnostic]
           QueryType: 'All Blocks'
```

Create `LinearizationAdvisor` Using `slTuner` Interface

Load Simulink model.

```
mdl = 'scdspeed';
load_system(mdl)
```

Create a `slTunerOptions` option set, enabling the `StoreAdvisor` option.

```
opt = slTunerOptions('StoreAdvisor',true);
```

Define input and output analysis points, and create an `slTuner` interface using this option set.

```
io(1) = linio('scdspeed/throttle (degrees)',1,'input');
io(2) = linio('scdspeed/rad//s to rpm',1,'output');
ST = slTuner(mdl,io,opt);
```

Typically, you would tune your control system using the `systune` function. Then, you can find the transfer function from the input to the output, returning the `info` argument.

```
[linsys,info] = getIOTransfer(ST,'scdspeed/throttle (degrees)','scdspeed/rad//s to rpm');
```

Extract the `LinearizationAdvisor` object from `info`.

```
advisor = info.Advisor
```

```
advisor =
  LinearizationAdvisor with properties:

               Model: 'scdspeed'
      OperatingPoint: [1x1 opcond.OperatingPoint]
    BlockDiagnostics: [1x27 linearize.advisor.BlockDiagnostic]
           QueryType: 'All Blocks'
```

Find Potentially Problematic Blocks for Linearization

Load Simulink model.

```
mdl = 'scdpendulum';
load_system(mdl)
```

Linearize model and obtain `LinearizationAdvisor` object.

```
io = getlinio(mdl);
opt = linearizeOptions('StoreAdvisor',true);
[linsys,~,info] = linearize(mdl,io,opt);
advisor = info.Advisor;
```

Find potentially problematic blocks for linearization.

```
result = advise(advisor)
```

```
result =
  LinearizationAdvisor with properties:

               Model: 'scdpendulum'
      OperatingPoint: [1x1 opcond.OperatingPoint]
    BlockDiagnostics: [1x3 linearize.advisor.BlockDiagnostic]
           QueryType: 'Linearization Advice'
```

Find All SISO Blocks

Load the Simulink model.

```
mdl = 'scdspeed';
load_system(mdl)
```

Linearize the model and obtain the `LinearizationAdvisor` object.

```
opts = linearizeOptions('StoreAdvisor',true);
io(1) = linio('scdspeed/throttle (degrees)',1,'input');
io(2) = linio('scdspeed/rad//s to rpm',1,'output');
[sys,op,info] = linearize(mdl,io,opts);
advisor = info.Advisor;
```

Create compound query object for finding all blocks with one input and one output.

```
qSISO = linqueryHasInputs(1) & linqueryHasOutputs(1);
```

Find all SISO blocks using compound query object.

```
advSISO = find(advisor,qSISO)
```

```
advSISO =
  LinearizationAdvisor with properties:

               Model: 'scdspeed'
      OperatingPoint: [1x1 opcond.OperatingPoint]
    BlockDiagnostics: [1x10 linearize.advisor.BlockDiagnostic]
           QueryType: '(Has 1 Inputs & Has 1 Outputs)'
```

Obtain Diagnostics for Potentially Problematic Blocks

Load Simulink model.

```
mdl = 'scdpendulum';
load_system(mdl)
```

Linearize the model and obtain `LinearizationAdvisor` object.

```
io = getlinio(mdl);
opt = linearizeOptions('StoreAdvisor',true);
[linsys,~,info] = linearize(mdl,io,opt);
advisor = info.Advisor;
```

Find blocks that are potentially problematic for linearization.

```
blocks = advise(advisor);
```

Obtain diagnostics for these blocks.

```
diags = getBlockInfo(blocks)
```

```
diags =
Linearization Diagnostics for the Blocks:

Block Info:
-----------
Index   BlockPath                                          Is On Path   Contributes To Lineariza
1.      scdpendulum/pendulum/Saturation                    Yes          No
2.      scdpendulum/angle_wrap/Trigonometric Function1     Yes          No
3.      scdpendulum/pendulum/Trigonometric Function        Yes          No
```

Alternative Functionality

App

You can interactively troubleshoot linearization results using the Linearization Advisor in the **Model Linearizer**. For an example, see “Troubleshoot Linearization Results in Model Linearizer” on page 4-16.

See Also

Objects
`BlockDiagnostic`

Functions
`linearize` | `getIOTransfer` | `getLoopTransfer` | `getCompSensitivity` | `getSensitivity` | `advise` | `find`

Topics
“Identify and Fix Common Linearization Issues” on page 4-6
“Troubleshoot Linearization Results at Command Line” on page 4-28
“Find Blocks in Linearization Results Matching Specific Criteria” on page 4-37

Introduced in R2017b

linqueryAdvise

Query object for finding blocks that are potentially problematic for linearization

Description

`linqueryAdvise` creates a custom query object for finding the blocks in a linearization result that are potentially problematic for linearization.

When you linearize a Simulink model, you can create a `LinearizationAdvisor` object that contains diagnostic information about individual block linearizations. To find block linearizations that satisfy specific criteria, you can use the `find` function with custom query objects. Alternatively, you can analyze linearization diagnostics using the Linearization Advisor in the **Model Linearizer**. For more information on finding specific blocks in linearization results, see “Find Blocks in Linearization Results Matching Specific Criteria” on page 4-37.

Using the `find` function with a `linqueryAdvise` object is equivalent to using the `advise` function.

Creation

Syntax

```
query = linqueryAdvise
```

Description

`query = linqueryAdvise` creates a query object for finding all the blocks in a `LinearizationAdvisor` object that are potentially problematic for linearization.

Properties

`QueryType` — Query type
`'Linearization Advice'` (default) | character vector

Query type, specified as `'Linearization Advice'`.

`Description` — Query description
`'Blocks that are Potentially Problematic for Linearization'` (default) | character vector

Query description, specified as `'Blocks that are Potentially Problematic for Linearization'`. You can add your own description to the query object using this property.

Usage

After creating a `linqueryAdvise` query object, you can:

- Find all the blocks in a `LinearizationAdvisor` object that are potentially problematic for linearization by using the `linqueryAdvise` query directly with the `find` command.
- Create a `CompoundQuery` object by logically combining the `linqueryAdvise` query with other query objects.

Object Functions

find Find blocks in linearization results that match specific criteria

Examples

Find Blocks with Potentially Problematic Linearizations

Load the Simulink model.

```
mdl = 'scdpendulum';
load_system(mdl)
```

Linearize the model and obtain the `LinearizationAdvisor` object.

```
opts = linearizeOptions('StoreAdvisor',true);
io = getlinio(mdl);
[sys,op,info] = linearize(mdl,io,opts);
advisor = info.Advisor;
```

Create query object, and find all the linearized blocks that have potentially problematic linearizations.

```
qAdvise = linqueryAdvise;
advAdvise = find(advisor,qAdvise)
```

```
advAdvise =
  LinearizationAdvisor with properties:

               Model: 'scdpendulum'
      OperatingPoint: [1x1 opcond.OperatingPoint]
    BlockDiagnostics: [1x3 linearize.advisor.BlockDiagnostic]
           QueryType: 'Linearization Advice'
```

Algorithms

Creating a `linqueryAdvise` object is equivalent to creating the following custom query:

```
qPath     = linqueryIsOnPath;
qZero     = linqueryIsZero;
qBlkRep   = linqueryIsBlockSubstituted;
qDiags    = linqueryHasDiagnostics;

q = qPath & (qZero | qDiags | qBlkRep);

advisor_new = find(advisor,q);
```

Alternative Functionality

App

You can also create custom queries for finding specific blocks in linearization results using the Linearization Advisor in the **Model Linearizer**. For more information, see “Find Blocks in Linearization Results Matching Specific Criteria” on page 4-37.

See Also

Objects
`LinearizationAdvisor` | `CompoundQuery`

Functions
`find`

Topics
“Find Blocks in Linearization Results Matching Specific Criteria” on page 4-37
“Troubleshoot Linearization Results at Command Line” on page 4-28

Introduced in R2017b

linqueryAllBlocks

Query object for finding all linearized blocks

Description

`linqueryAllBlocks` creates a custom query object for finding all the linearized blocks listed in a `LinearizationAdvisor` object.

When you linearize a Simulink model, you can create a `LinearizationAdvisor` object that contains diagnostic information about individual block linearizations. To find block linearizations that satisfy specific criteria, you can use the `find` function with custom query objects. Alternatively, you can analyze linearization diagnostics using the Linearization Advisor in the **Model Linearizer**. For more information on finding specific blocks in linearization results, see “Find Blocks in Linearization Results Matching Specific Criteria” on page 4-37.

When you use this query object with the `find` command, the `LinearizationAdvisor` object returned by `find` contains the same blocks as the input `LinearizationAdvisor` object. Therefore, it is not necessary to use `linqueryAllBlocks`. This command is a utility function used by the Linearization Advisor in the **Model Linearizer**.

Creation

Syntax

```
query = linqueryAllBlocks
```

Description

`query = linqueryAllBlocks` creates a query object for finding all the linearized blocks listed in a `LinearizationAdvisor` object.

Properties

`QueryType` — Query type
`'All Blocks'` (default) | character vector

Query type, specified as `'All Blocks'`.

`Description` — Query description
`'All Linearized Blocks'` (default) | character vector

Query description, specified as `'All Linearized Blocks'`.

Object Functions

find Find blocks in linearization results that match specific criteria

Examples

Find All Linearized Blocks

Load the Simulink model.

```
mdl = 'scdpwm';
load_system(mdl)
```

Linearize the model and obtain the `LinearizationAdvisor` object.

```
opts = linearizeOptions('StoreAdvisor',true);
[sys,op,info] = linearize(mdl,getlinio(mdl),opts);
advisor = info.Advisor;
```

Create query object, and find all the linearized blocks.

```
qAll = linqueryAllBlocks;
advAll = find(advisor,qAll)
```

```
advAll =
  LinearizationAdvisor with properties:

              Model: 'scdpwm'
     OperatingPoint: [1x1 opcond.OperatingPoint]
    BlockDiagnostics: [1x10 linearize.advisor.BlockDiagnostic]
           QueryType: 'All Blocks'
```

Alternative Functionality

App

You can also create custom queries for finding specific blocks in linearization results using the Linearization Advisor in the **Model Linearizer**. For more information, see “Find Blocks in Linearization Results Matching Specific Criteria” on page 4-37.

See Also

Objects
`LinearizationAdvisor` | `CompoundQuery`

Functions
`find`

Topics
“Find Blocks in Linearization Results Matching Specific Criteria” on page 4-37
“Troubleshoot Linearization Results at Command Line” on page 4-28

Introduced in R2017b

linqueryContributesToLinearization

Query object for finding blocks that contribute to the model linearization result

Description

`linqueryContributesToLinearization` creates a custom query object for finding all the blocks that numerically contribute to the model linearization result.

When you linearize a Simulink model, you can create a `LinearizationAdvisor` object that contains diagnostic information about individual block linearizations. To find block linearizations that satisfy specific criteria, you can use the `find` function with custom query objects. Alternatively, you can analyze linearization diagnostics using the Linearization Advisor in the **Model Linearizer**. For more information on finding specific blocks in linearization results, see “Find Blocks in Linearization Results Matching Specific Criteria” on page 4-37.

Creation

Syntax

```
query = linqueryContributesToLinearization
```

Description

`query = linqueryContributesToLinearization` creates a query object for finding all the blocks in a `LinearizationAdvisor` object that numerically contribute to the model linearization result.

Properties

`QueryType` — Query type
`'Contributes to Linearization'` (default) | character vector

Query type, specified as `'Contributes to Linearization'`.

`Description` — Query description
`'Blocks that Contribute to the Model Linearization'` (default) | character vector

Query description, specified as `'Blocks that Contribute to the Model Linearization'`. You can add your own description to the query object using this property.

Usage

After creating a `linqueryContributesToLinearization` query object, you can:

- Find all the blocks in a `LinearizationAdvisor` object that numerically contribute to the model linearization result by using the `linqueryContributesToLinearization` query directly with the `find` command.

- Create a `CompoundQuery` object by logically combining the `linqueryContributesToLinearization` query with other query objects.

Object Functions

find Find blocks in linearization results that match specific criteria

Examples

Find Blocks That Contribute to Linearization Result

Load the Simulink model.

```
mdl = 'scdspeed';
load_system(mdl)
```

Linearize the model, and obtain the `LinearizationAdvisor` object.

```
opts = linearizeOptions('StoreAdvisor',true);
io(1) = linio('scdspeed/throttle (degrees)',1,'input');
io(2) = linio('scdspeed/rad//s to rpm',1,'output');
[sys,op,info] = linearize(mdl,io,opts);
advisor = info.Advisor;
```

Create query object, and find all blocks that numerically contribute to the model linearization result.

```
qContribute = linqueryContributesToLinearization;
advContribute = find(advisor,qContribute)
```

```
advContribute =
  LinearizationAdvisor with properties:

               Model: 'scdspeed'
      OperatingPoint: [1x1 opcond.OperatingPoint]
    BlockDiagnostics: [1x22 linearize.advisor.BlockDiagnostic]
           QueryType: 'Contributes to Linearization'
```

To find blocks that do not contribute to the linearization result, use the same query object with a NOT (~) logical operator.

```
advNoContribute = find(advisor,~qContribute)
```

```
advNoContribute =
  LinearizationAdvisor with properties:

               Model: 'scdspeed'
      OperatingPoint: [1x1 opcond.OperatingPoint]
    BlockDiagnostics: [1x5 linearize.advisor.BlockDiagnostic]
           QueryType: '~(Contributes to Linearization)'
```

Alternative Functionality

App

You can also create custom queries for finding specific blocks in linearization results using the Linearization Advisor in the **Model Linearizer**. For more information, see “Find Blocks in Linearization Results Matching Specific Criteria” on page 4-37.

See Also

Objects

`LinearizationAdvisor` | `CompoundQuery`

Functions

`find` | `highlight`

Topics

“Find Blocks in Linearization Results Matching Specific Criteria” on page 4-37
“Troubleshoot Linearization Results at Command Line” on page 4-28

Introduced in R2017b

linqueryHasDiagnostics

Query object for finding blocks that have diagnostic messages regarding their linearization

Description

`linqueryHasDiagnostics` creates a custom query object for finding all the blocks in a linearization result that have diagnostic messages regarding their linearization.

When you linearize a Simulink model, you can create a `LinearizationAdvisor` object that contains diagnostic information about individual block linearizations. To find block linearizations that satisfy specific criteria, you can use the `find` function with custom query objects. Alternatively, you can analyze linearization diagnostics using the Linearization Advisor in the **Model Linearizer**. For more information on finding specific blocks in linearization results, see “Find Blocks in Linearization Results Matching Specific Criteria” on page 4-37.

Creation

Syntax

```
query = linqueryHasDiagnostics
```

Description

`query = linqueryHasDiagnostics` creates a query object for finding all the blocks in a `LinearizationAdvisor` object that have diagnostic messages regarding their linearization.

Properties

`QueryType` — Query type
`'Has Diagnostics'` (default) | character vector

Query type, specified as `'Has Diagnostics'`.

`Description` — Query description
`'Blocks that have Linearization Diagnostic Messages'` (default) | character vector

Query description, specified as `'Blocks that have Linearization Diagnostic Messages'`. You can add your own description to the query object using this property.

Usage

After creating a `linqueryHasDiagnostics` query object, you can:

- Find all the blocks in a `LinearizationAdvisor` object that have diagnostic messages regarding their linearization by using the `linqueryHasDiagnostics` query directly with the `find` command.

- Create a `CompoundQuery` object by logically combining the `linqueryHasDiagnostics` query with other query objects.

Object Functions

find Find blocks in linearization results that match specific criteria

Examples

Find All Blocks with Linearization Diagnostic Messages

Load the Simulink model.

```
mdl = 'scdpendulum';
load_system(mdl)
```

Linearize the model and obtain the `LinearizationAdvisor` object.

```
opts = linearizeOptions('StoreAdvisor',true);
io = getlinio(mdl);
[sys,op,info] = linearize(mdl,io,opts);
advisor = info.Advisor;
```

Create query object, and find all blocks with diagnostic messages regarding their linearization.

```
qDiag = linqueryHasDiagnostics;
advDiag = find(advisor,qDiag)
```

```
advDiag =
  LinearizationAdvisor with properties:

               Model: 'scdpendulum'
      OperatingPoint: [1x1 opcond.OperatingPoint]
    BlockDiagnostics: [1x1 linearize.advisor.BlockDiagnostic]
           QueryType: 'Has Diagnostics'
```

Alternative Functionality

App

You can also create custom queries for finding specific blocks in linearization results using the Linearization Advisor in the **Model Linearizer**. For more information, see “Find Blocks in Linearization Results Matching Specific Criteria” on page 4-37.

See Also

Objects

`LinearizationAdvisor` | `CompoundQuery`

Functions

`find`

Topics
“Find Blocks in Linearization Results Matching Specific Criteria” on page 4-37
“Troubleshoot Linearization Results at Command Line” on page 4-28

Introduced in R2017b

linqueryHasInputs

Query object for finding blocks with specified number of inputs

Description

`linqueryHasInputs` creates a custom query object for finding all the blocks in a linearization result that have a specified number of inputs.

When you linearize a Simulink model, you can create a `LinearizationAdvisor` object that contains diagnostic information about individual block linearizations. To find block linearizations that satisfy specific criteria, you can use the `find` function with custom query objects. Alternatively, you can analyze linearization diagnostics using the Linearization Advisor in the **Model Linearizer**. For more information on finding specific blocks in linearization results, see “Find Blocks in Linearization Results Matching Specific Criteria” on page 4-37.

Creation

Syntax

```
query = linqueryHasInputs(numInputs)
```

Description

`query = linqueryHasInputs(numInputs)` creates a query object for finding all the blocks in a `LinearizationAdvisor` object that have the specified number of inputs. This syntax sets the `NumInputs` property of the query object.

Properties

`NumInputs` — Number of block inputs
nonnegative integer

Number of block inputs, specified as a nonnegative integer.

`QueryType` — Query type
character vector

Query type, specified as a character vector of the form `'Has <N> Inputs'`, where `<N>` is equal to `NumInputs`.

`Description` — Query description
character vector

Query description, specified as a character vector of the form `'Blocks with <N> Inputs'`, where `<N>` is equal to `NumInputs`. You can add your own description to the query object using this property.

Usage

After creating a `linqueryHasInputs` query object, you can:

- Find all the blocks in a `LinearizationAdvisor` object that have the specified number of inputs by using the `linqueryHasInputs` query directly with the `find` command.
- Create a `CompoundQuery` object by logically combining the `linqueryHasInputs` query with other query objects.

Object Functions

find Find blocks in linearization results that match specific criteria

Examples

Find All Blocks with Two Inputs

Load the Simulink model.

```
mdl = 'scdspeed';
load_system(mdl)
```

Linearize the model and obtain the `LinearizationAdvisor` object.

```
opts = linearizeOptions('StoreAdvisor',true);
io(1) = linio('scdspeed/throttle (degrees)',1,'input');
io(2) = linio('scdspeed/rad//s to rpm',1,'output');
[sys,op,info] = linearize(mdl,io,opts);
advisor = info.Advisor;
```

Create query object, and find all the linearized blocks with two inputs.

```
qIn = linqueryHasInputs(2);
advIn = find(advisor,qIn)
```

```
advIn =
  LinearizationAdvisor with properties:

               Model: 'scdspeed'
      OperatingPoint: [1x1 opcond.OperatingPoint]
    BlockDiagnostics: [1x13 linearize.advisor.BlockDiagnostic]
           QueryType: 'Has 2 Inputs'
```

Find All SISO Blocks

Load the Simulink model.

```
mdl = 'scdspeed';
load_system(mdl)
```

Linearize the model and obtain the `LinearizationAdvisor` object.

```
opts = linearizeOptions('StoreAdvisor',true);
io(1) = linio('scdspeed/throttle (degrees)',1,'input');
io(2) = linio('scdspeed/rad//s to rpm',1,'output');
[sys,op,info] = linearize(mdl,io,opts);
advisor = info.Advisor;
```

Create compound query object for finding all blocks with one input and one output.

```
qSISO = linqueryHasInputs(1) & linqueryHasOutputs(1);
```

Find all SISO blocks using compound query object.

```
advSISO = find(advisor,qSISO)
```

```
advSISO =
  LinearizationAdvisor with properties:

               Model: 'scdspeed'
      OperatingPoint: [1x1 opcond.OperatingPoint]
    BlockDiagnostics: [1x10 linearize.advisor.BlockDiagnostic]
           QueryType: '(Has 1 Inputs & Has 1 Outputs)'
```

Alternative Functionality

App

You can also create custom queries for finding specific blocks in linearization results using the Linearization Advisor in the **Model Linearizer**. For more information, see “Find Blocks in Linearization Results Matching Specific Criteria” on page 4-37.

See Also

Objects

`LinearizationAdvisor` | `CompoundQuery`

Functions

`find`

Topics

“Find Blocks in Linearization Results Matching Specific Criteria” on page 4-37
“Troubleshoot Linearization Results at Command Line” on page 4-28

Introduced in R2017b

linqueryHasOrder

Query object for finding blocks with specified number of states

Description

`linqueryHasOrder` creates a custom query object for finding all the blocks in a linearization result that have a specified number of states.

When you linearize a Simulink model, you can create a `LinearizationAdvisor` object that contains diagnostic information about individual block linearizations. To find block linearizations that satisfy specific criteria, you can use the `find` function with custom query objects. Alternatively, you can analyze linearization diagnostics using the Linearization Advisor in the **Model Linearizer**. For more information on finding specific blocks in linearization results, see “Find Blocks in Linearization Results Matching Specific Criteria” on page 4-37.

Creation

Syntax

```
query = linqueryHasOrder(numStates)
```

Description

`query = linqueryHasOrder(numStates)` creates a query object for finding all the blocks in a `LinearizationAdvisor` object that have the specified number of states. This syntax sets the `NumStates` property of the query object.

Properties

`NumStates` — Number of block states
nonnegative integer

Number of block states, specified as a nonnegative integer.

`QueryType` — Query type
character vector

Query type, specified as a character vector of the form `'Has <N> States'`, where `<N>` is equal to `NumStates`.

`Description` — Query description
character vector

Query description, specified as a character vector of the form `'Blocks with <N> States`, where `<N>` is equal to `NumStates`. You can add your own description to the query object using this property.

Usage

After creating a `linqueryHasOrder` query object, you can:

- Find all the blocks in a `LinearizationAdvisor` object that have the specified number of states by using the `linqueryHasOrder` query directly with the `find` command.
- Create a `CompoundQuery` object by logically combining the `linqueryHasOrder` query with other query objects.

Object Functions

find Find blocks in linearization results that match specific criteria

Examples

Find All Blocks with Two States

Load the Simulink model.

```
mdl = 'scdspeed';
load_system(mdl)
```

Linearize the model and obtain the `LinearizationAdvisor` object.

```
opts = linearizeOptions('StoreAdvisor',true);
io(1) = linio('scdspeed/throttle (degrees)',1,'input');
io(2) = linio('scdspeed/rad//s to rpm',1,'output');
[sys,op,info] = linearize(mdl,io,opts);
advisor = info.Advisor;
```

Create query object, and find all the linearized blocks with two states.

```
qOrder = linqueryHasOrder(2);
advOrder = find(advisor,qOrder)
```

```
advOrder =
  LinearizationAdvisor with properties:

               Model: 'scdspeed'
      OperatingPoint: [1x1 opcond.OperatingPoint]
    BlockDiagnostics: [1x1 linearize.advisor.BlockDiagnostic]
           QueryType: 'Has 2 States'
```

Alternative Functionality

App

You can also create custom queries for finding specific blocks in linearization results using the Linearization Advisor in the **Model Linearizer**. For more information, see “Find Blocks in Linearization Results Matching Specific Criteria” on page 4-37.

See Also

Objects
`LinearizationAdvisor` | `CompoundQuery`

Functions
`find`

Topics
“Find Blocks in Linearization Results Matching Specific Criteria” on page 4-37
“Troubleshoot Linearization Results at Command Line” on page 4-28

Introduced in R2017b

linqueryHasOutputs

Query object for finding blocks with specified number of outputs

Description

`linqueryHasOutputs` creates a custom query object for finding all the blocks in a linearization result that have a specified number of outputs.

When you linearize a Simulink model, you can create a `LinearizationAdvisor` object that contains diagnostic information about individual block linearizations. To find block linearizations that satisfy specific criteria, you can use the `find` function with custom query objects. Alternatively, you can analyze linearization diagnostics using the Linearization Advisor in the **Model Linearizer**. For more information on finding specific blocks in linearization results, see “Find Blocks in Linearization Results Matching Specific Criteria” on page 4-37.

Creation

Syntax

```
query = linqueryHasOutputs(numOutputs)
```

Description

`query = linqueryHasOutputs(numOutputs)` creates a query object for finding all the blocks in a `LinearizationAdvisor` object that have the specified number of outputs. This syntax sets the `NumOutputs` property of the query object.

Properties

`NumOutputs` — Number of block outputs
nonnegative integer

Number of block outputs, specified as a nonnegative integer.

`QueryType` — Query type
character vector

Query type, specified as a character vector of the form `'Has <N> Outputs'`, where `<N>` is equal to `NumOutputs`.

`Description` — Query description
character vector

Query description, specified as a character vector of the form `'Blocks with <N> Outputs'`, where `<N>` is equal to `NumOutputs`. You can add your own description to the query object using this property.

Usage

After creating a `linqueryHasOutputs` query object, you can:

- Find all the blocks in a `LinearizationAdvisor` object that have the specified number of outputs by using the `linqueryHasOutputs` query directly with the `find` command.
- Create a `CompoundQuery` object by logically combining the `linqueryHasOutputs` query with other query objects.

Object Functions

find Find blocks in linearization results that match specific criteria

Examples

Find All Blocks with Two Outputs

Load the Simulink model.

```
mdl = 'scdpendulum';
load_system(mdl)
```

Linearize the model and obtain the `LinearizationAdvisor` object.

```
opts = linearizeOptions('StoreAdvisor',true);
io = getlinio(mdl);
[sys,op,info] = linearize(mdl,io,opts);
advisor = info.Advisor;
```

Create query object, and find all the linearized blocks with two outputs.

```
qOut = linqueryHasOutputs(2);
advOut = find(advisor,qOut)
```

```
advOut =
  LinearizationAdvisor with properties:

               Model: 'scdpendulum'
      OperatingPoint: [1x1 opcond.OperatingPoint]
    BlockDiagnostics: [1x1 linearize.advisor.BlockDiagnostic]
           QueryType: 'Has 2 Outputs'
```

Find All SISO Blocks

Load the Simulink model.

```
mdl = 'scdspeed';
load_system(mdl)
```

Linearize the model and obtain the `LinearizationAdvisor` object.

```
opts = linearizeOptions('StoreAdvisor',true);
io(1) = linio('scdspeed/throttle (degrees)',1,'input');
```

```
io(2) = linio('scdspeed/rad//s to rpm',1,'output');
[sys,op,info] = linearize(mdl,io,opts);
advisor = info.Advisor;
```

Create compound query object for finding all blocks with one input and one output.

```
qSISO = linqueryHasInputs(1) & linqueryHasOutputs(1);
```

Find all SISO blocks using compound query object.

```
advSISO = find(advisor,qSISO)
```

```
advSISO =
  LinearizationAdvisor with properties:

               Model: 'scdspeed'
      OperatingPoint: [1x1 opcond.OperatingPoint]
    BlockDiagnostics: [1x10 linearize.advisor.BlockDiagnostic]
           QueryType: '(Has 1 Inputs & Has 1 Outputs)'
```

Alternative Functionality

App

You can also create custom queries for finding specific blocks in linearization results using the Linearization Advisor in the **Model Linearizer**. For more information, see “Find Blocks in Linearization Results Matching Specific Criteria” on page 4-37.

See Also

Objects

`LinearizationAdvisor` | `CompoundQuery`

Functions

`find`

Topics

“Find Blocks in Linearization Results Matching Specific Criteria” on page 4-37
“Troubleshoot Linearization Results at Command Line” on page 4-28

Introduced in R2017b

linqueryHasSampleTime

Query object for finding blocks with specified sample time

Description

`linqueryHasSampleTime` creates a custom query object for finding all the blocks in a linearization result that have a specified sample time.

When you linearize a Simulink model, you can create a `LinearizationAdvisor` object that contains diagnostic information about individual block linearizations. To find block linearizations that satisfy specific criteria, you can use the `find` function with custom query objects. Alternatively, you can analyze linearization diagnostics using the Linearization Advisor in the **Model Linearizer**. For more information on finding specific blocks in linearization results, see “Find Blocks in Linearization Results Matching Specific Criteria” on page 4-37.

Creation

Syntax

```
query = linqueryHasSampleTime(ts)
```

Description

`query = linqueryHasSampleTime(ts)` creates a query object for finding all the blocks in a `LinearizationAdvisor` object that have sample time `ts`. This syntax sets the `Ts` property of the query object.

Properties

`Ts` — Sample
nonnegative scalar

Block sample time, specified as a nonnegative scalar. Specify `Ts` in the time units of the linearized model.

To find continuous-time blocks, specify `Ts` as `0`.

`QueryType` — Query type
character vector

Query type, specified as a character vector of the form `'Has <T> Sample Time'`, where `<T>` is equal to `Ts`.

`Description` — Query description
character vector

Query description, specified as a character vector of the form `'Blocks with <T> Sample Time'`, where `<T>` is equal to `Ts`. You can add your own description to the query object using this property.

Usage

After creating a `linqueryHasSampleTime` query object, you can:

- Find all the blocks in a `LinearizationAdvisor` object that have a specified sample time by using the `linqueryHasSampleTime` query directly with the `find` command.
- Create a `CompoundQuery` object by logically combining the `linqueryHasSampleTime` query with other query objects.

Object Functions

find Find blocks in linearization results that match specific criteria

Examples

Find Blocks with Specified Sample Time

Load the Simulink model.

```
mdl = 'scdmrate';
load_system(mdl)
```

Linearize the model and obtain the `LinearizationAdvisor` object.

```
opts = linearizeOptions('StoreAdvisor',true);
io(1) = linio('scdmrate/Constant',1,'input');
io(2) = linio('scdmrate/sysTs2',1,'openoutput');
[linsys,linop,info] = linearize(mdl,io,opts);
advisor = info.Advisor;
```

Create query object and find all the linearized blocks with a sample time of `0.1` seconds.

```
qTs = linqueryHasSampleTime(0.01);
advTs = find(advisor,qTs)
```

```
advTs =
  LinearizationAdvisor with properties:

               Model: 'scdmrate'
      OperatingPoint: [1x1 opcond.OperatingPoint]
    BlockDiagnostics: [1x1 linearize.advisor.BlockDiagnostic]
           QueryType: 'Has 0.01 Sample Time'
```

Find All Continuous-Time Blocks

Load the Simulink model.

```
mdl = 'scdmrate';
load_system(mdl)
```

Linearize the model and obtain the `LinearizationAdvisor` object.

```
opts = linearizeOptions('StoreAdvisor',true);
io = getlinio(mdl);
[sys,op,info] = linearize(mdl,io,opts);
advisor = info.Advisor;
```

Create query object, and find all linearized blocks with continuous-time linearizations.

```
qCont = linqueryHasSampleTime(0);
advCont = find(advisor,qCont)
```

```
advCont =
  LinearizationAdvisor with properties:

               Model: 'scdmrate'
      OperatingPoint: [1x1 opcond.OperatingPoint]
    BlockDiagnostics: [1x5 linearize.advisor.BlockDiagnostic]
           QueryType: 'Has 0 Sample Time'
```

Alternative Functionality

App

You can also create custom queries for finding specific blocks in linearization results using the Linearization Advisor in the **Model Linearizer**. For more information, see “Find Blocks in Linearization Results Matching Specific Criteria” on page 4-37.

See Also

Objects

`LinearizationAdvisor` | `CompoundQuery`

Functions

`find`

Topics

“Find Blocks in Linearization Results Matching Specific Criteria” on page 4-37
“Troubleshoot Linearization Results at Command Line” on page 4-28

Introduced in R2017b

linqueryHasZeroIOPair

Query object for finding blocks with at least one input/output pair that linearizes to zero

Description

`linqueryHasZeroIOPair` creates a custom query object for finding all the blocks in a linearization result that have at least one input/output pair that linearizes to zero. For a zero input/output pair, a change in the input value has no effect on the output value.

When you linearize a Simulink model, you can create a `LinearizationAdvisor` object that contains diagnostic information about individual block linearizations. To find block linearizations that satisfy specific criteria, you can use the `find` function with custom query objects. Alternatively, you can analyze linearization diagnostics using the Linearization Advisor in the **Model Linearizer**. For more information on finding specific blocks in linearization results, see “Find Blocks in Linearization Results Matching Specific Criteria” on page 4-37.

Creation

Syntax

```
query = linqueryHasZeroIOPair
```

Description

`query = linqueryHasZeroIOPair` creates a query object for finding all the blocks in a `LinearizationAdvisor` object that have at least one input/output path that linearizes to zero.

Properties

`QueryType` — Query type
`'Has Zero I/O Pair'` (default) | character vector

Query type, specified as `'Has Zero I/O Pair'`.

`Description` — Query description
`'Blocks with a Zero IO Pair'` (default) | character vector

Query description, specified as `'Blocks with a Zero IO Pair'`. You can add your own description to the query object using this property.

Usage

After creating a `linqueryHasZeroIOPair` query object, you can:

- Find all the blocks in a `LinearizationAdvisor` object that have at least one input/output path that linearizes to zero by using the `linqueryHasZeroIOPair` query directly with the `find` command.

- Create a `CompoundQuery` object by logically combining the `linqueryHasZeroIOPair` query with other query objects.

Object Functions

find Find blocks in linearization results that match specific criteria

Examples

Find Blocks with Zero Input/Output Paths

Load the Simulink model.

```
mdl = 'scdspeed';
load_system(mdl)
```

Linearize the model and obtain the `LinearizationAdvisor` object.

```
opts = linearizeOptions('StoreAdvisor',true);
io(1) = linio('scdspeed/throttle (degrees)',1,'input');
io(2) = linio('scdspeed/rad//s to rpm',1,'output');
[sys,op,info] = linearize(mdl,io,opts);
advisor = info.Advisor;
```

Create query object, and find all blocks with at least one input/output path that linearizes to zero.

```
qZeroPair = linqueryHasZeroIOPair;
advZeroPair = find(advisor,qZeroPair)
```

```
advZeroPair =
  LinearizationAdvisor with properties:

              Model: 'scdspeed'
     OperatingPoint: [1x1 opcond.OperatingPoint]
    BlockDiagnostics: [1x6 linearize.advisor.BlockDiagnostic]
           QueryType: 'Has Zero I/O Pair'
```

Alternative Functionality

App

You can also create custom queries for finding specific blocks in linearization results using the Linearization Advisor in the **Model Linearizer**. For more information, see “Find Blocks in Linearization Results Matching Specific Criteria” on page 4-37.

See Also

Objects

`LinearizationAdvisor` | `CompoundQuery`

Functions

`find`

Topics
“Find Blocks in Linearization Results Matching Specific Criteria” on page 4-37
“Troubleshoot Linearization Results at Command Line” on page 4-28

Introduced in R2017b

linqueryIsBlockSubstituted

Query object for finding blocks that have custom block linearizations specified

Description

`linqueryIsBlockSubstituted` creates a custom query object for finding all the blocks in a linearization result that have custom block linearizations specified.

When you linearize a Simulink model, you can create a `LinearizationAdvisor` object that contains diagnostic information about individual block linearizations. To find block linearizations that satisfy specific criteria, you can use the `find` function with custom query objects. Alternatively, you can analyze linearization diagnostics using the Linearization Advisor in the **Model Linearizer**. For more information on finding specific blocks in linearization results, see “Find Blocks in Linearization Results Matching Specific Criteria” on page 4-37.

Creation

Syntax

```
query = linqueryIsBlockSubstituted
```

Description

`query = linqueryIsBlockSubstituted` creates a query object for finding all the blocks in a `LinearizationAdvisor` object that have custom block linearization specified.

Properties

`QueryType` — Query type
`'Block Substituted'` (default) | character vector

Query type, specified as `'Block Substituted'`.

`Description` — Query description
`'Blocks Linearized with Block Substitution'` (default) | character vector

Query description, specified as `'Blocks Linearized with Block Substitution'`. You can add your own description to the query object using this property.

Usage

After creating a `linqueryIsBlockSubstituted` query object, you can:

- Find all the blocks in a `LinearizationAdvisor` object that have a custom linearization specified by using the `linqueryIsBlockSubstituted` query directly with the `find` command.
- Create a `CompoundQuery` object by logically combining the `linqueryIsBlockSubstituted` query with other query objects.

Object Functions

find Find blocks in linearization results that match specific criteria

Examples

Find Blocks with Substitute Linearizations

Load the Simulink model.

```
mdl = 'scdpwmCustom';
load_system(mdl)
```

Linearize the model and obtain the `LinearizationAdvisor` object.

```
opts = linearizeOptions('StoreAdvisor',true);
[sys,op,info] = linearize(mdl,getlinio(mdl),opts);
advisor = info.Advisor;
```

Create query object, and find all blocks with substitute linearizations.

```
qSub = linqueryIsBlockSubstituted;
advSub = find(advisor,qSub)
```

```
advSub =
  LinearizationAdvisor with properties:

               Model: 'scdpwmCustom'
      OperatingPoint: [1x1 opcond.OperatingPoint]
    BlockDiagnostics: [1x1 linearize.advisor.BlockDiagnostic]
           QueryType: 'Block Substituted'
```

Alternative Functionality

App

You can also create custom queries for finding specific blocks in linearization results using the Linearization Advisor in the **Model Linearizer**. For more information, see “Find Blocks in Linearization Results Matching Specific Criteria” on page 4-37.

See Also

Objects

`LinearizationAdvisor` | `CompoundQuery`

Functions

`find`

Topics

“Find Blocks in Linearization Results Matching Specific Criteria” on page 4-37
“Troubleshoot Linearization Results at Command Line” on page 4-28

Introduced in R2017b

linqueryIsBlockType

Query object for finding blocks of the specified type

Description

`linqueryIsBlockType` creates a custom query object for finding all the blocks of a specified type in a linearization result.

When you linearize a Simulink model, you can create a `LinearizationAdvisor` object that contains diagnostic information about individual block linearizations. To find block linearizations that satisfy specific criteria, you can use the `find` function with custom query objects. Alternatively, you can analyze linearization diagnostics using the Linearization Advisor in the **Model Linearizer**. For more information on finding specific blocks in linearization results, see “Find Blocks in Linearization Results Matching Specific Criteria” on page 4-37.

Creation

Syntax

```
query = linqueryIsBlockType(Type)
```

Description

`query = linqueryIsBlockType(Type)` creates a query object for finding all the blocks in a `LinearizationAdvisor` object that are of type `Type`.

Input Arguments

Type — Block type
character vector | string

Block type, specified as a character vector or string. To specify a block type, use the corresponding `blocktype` parameter of the block. To get the `blocktype` parameter for the currently selected block in the Simulink model, at the MATLAB command line, type:

```
get_param(gcb,'blocktype')
```

Also, to find:

- MATLAB Function blocks, specify `Type` as `'matlab function'`.
- Stateflow charts, specify `Type` as `'chart'`.
- Simscape networks, specify `Type` as `'simscape'`. A `LinearizationAdvisor` object does not provide diagnostic information on a component-level basis for Simscape networks. Instead, it groups diagnostic information together for multiple Simscape components connected to a single Solver Configuration block.

Properties

`QueryType` — Query type
character vector

Query type, specified as a character vector of the form `'<type> Blocks'`, where `<type>` is equal the block type specified in `Type`.

`Description` — Query description
character vector

Query description, specified as a character vector of the form `'Blocks with <type> Block types'`, where `<type>` is equal to `Type`. You can add your own description to the query object using this property.

Usage

After creating a `linqueryIsBlockType` query object, you can:

- Find all the blocks of a specified type in a `LinearizationAdvisor` object by using the `linqueryIsBlockType` query directly with the `find` command.
- Create a `CompoundQuery` object by logically combining the `linqueryIsBlockType` query with other query objects.

Object Functions

find Find blocks in linearization results that match specific criteria

Examples

Find All Integrator Blocks in Linearization Result

Load the Simulink model.

```
mdl = 'scdspeed';
load_system(mdl)
```

Linearize the model and obtain the `LinearizationAdvisor` object.

```
opts = linearizeOptions('StoreAdvisor',true);
io(1) = linio('scdspeed/throttle (degrees)',1,'input');
io(2) = linio('scdspeed/rad//s to rpm',1,'output');
[sys,op,info] = linearize(mdl,io,opts);
advisor = info.Advisor;
```

Create query object, and find all the integrator blocks.

```
qInteg = linqueryIsBlockType('Integrator');
advInteg = find(advisor,qInteg)
```

```
advInteg =
  LinearizationAdvisor with properties:

               Model: 'scdspeed'
```

```
    OperatingPoint: [1x1 opcond.OperatingPoint]
BlockDiagnostics: [1x2 linearize.advisor.BlockDiagnostic]
       QueryType: 'Integrator Blocks'
```

Alternative Functionality

App

You can also create custom queries for finding specific blocks in linearization results using the Linearization Advisor in the **Model Linearizer**. For more information, see “Find Blocks in Linearization Results Matching Specific Criteria” on page 4-37.

See Also

Objects

`LinearizationAdvisor` | `CompoundQuery`

Functions

`find`

Topics

“Find Blocks in Linearization Results Matching Specific Criteria” on page 4-37
“Troubleshoot Linearization Results at Command Line” on page 4-28

Introduced in R2017b

linqueryIsExact

Query object for finding blocks linearized using their defined exact linearization

Description

`linqueryIsExact` creates a custom query object for finding all the blocks in a linearization result that are linearized using their defined exact linearization.

When you linearize a Simulink model, you can create a `LinearizationAdvisor` object that contains diagnostic information about individual block linearizations. To find block linearizations that satisfy specific criteria, you can use the `find` function with custom query objects. Alternatively, you can analyze linearization diagnostics using the Linearization Advisor in the **Model Linearizer**. For more information on finding specific blocks in linearization results, see “Find Blocks in Linearization Results Matching Specific Criteria” on page 4-37.

Creation

Syntax

```
query = linqueryIsExact
```

Description

`query = linqueryIsExact` creates a query object for finding all the blocks in a `LinearizationAdvisor` object that are linearized using their defined exact linearization.

Properties

`QueryType` — Query type
`'Exact'` (default) | character vector

Query type, specified as `'Exact'`.

`Description` — Query description
`'Blocks that are Analytically Linearized'` (default) | character vector

Query description, specified as `'Blocks that are Analytically Linearized'`. You can add your own description to the query object using this property.

Usage

After creating a `linqueryIsExact` query object, you can:

- Find all the blocks in a `LinearizationAdvisor` object that are linearized using their defined exact linearization by using the `linqueryIsExact` query directly with the `find` command.
- Create a `CompoundQuery` object by logically combining the `linqueryIsExact` query with other query objects.

Object Functions

find Find blocks in linearization results that match specific criteria

Examples

Find All Blocks Linearized Using Exact Linearization

Load the Simulink model.

```
mdl = 'scdspeed';
load_system(mdl)
```

Linearize the model and obtain the `LinearizationAdvisor` object.

```
opts = linearizeOptions('StoreAdvisor',true);
io(1) = linio('scdspeed/throttle (degrees)',1,'input');
io(2) = linio('scdspeed/rad//s to rpm',1,'output');
[sys,op,info] = linearize(mdl,io,opts);
advisor = info.Advisor;
```

Create query object, and find all blocks linearized using their defined exact linearization.

```
qExact = linqueryIsExact;
advExact = find(advisor,qExact)
```

```
advExact =
  LinearizationAdvisor with properties:

               Model: 'scdspeed'
      OperatingPoint: [1x1 opcond.OperatingPoint]
    BlockDiagnostics: [1x21 linearize.advisor.BlockDiagnostic]
           QueryType: 'Exact'
```

Alternative Functionality

App

You can also create custom queries for finding specific blocks in linearization results using the Linearization Advisor in the **Model Linearizer**. For more information, see “Find Blocks in Linearization Results Matching Specific Criteria” on page 4-37.

See Also

Objects

`LinearizationAdvisor` | `CompoundQuery`

Functions

`find`

Topics

“Find Blocks in Linearization Results Matching Specific Criteria” on page 4-37
“Troubleshoot Linearization Results at Command Line” on page 4-28

Introduced in R2017b

linqueryIsNumericallyPerturbed

Query object for finding blocks linearized using numerical perturbation

Description

`linqueryIsNumericallyPerturbed` creates a custom query object for finding all the blocks in a linearization result that are linearized using numerical perturbation.

When you linearize a Simulink model, you can create a `LinearizationAdvisor` object that contains diagnostic information about individual block linearizations. To find block linearizations that satisfy specific criteria, you can use the `find` function with custom query objects. Alternatively, you can analyze linearization diagnostics using the Linearization Advisor in the **Model Linearizer**. For more information on finding specific blocks in linearization results, see “Find Blocks in Linearization Results Matching Specific Criteria” on page 4-37.

Creation

Syntax

```
query = linqueryIsNumericallyPerturbed
```

Description

`query = linqueryIsNumericallyPerturbed` creates a query object for finding all the blocks in a `LinearizationAdvisor` object that are linearized using numerical perturbation.

Properties

`QueryType` — Query type
`'Perturbation'` (default) | character vector

Query type, specified as `'Perturbation'`.

`Description` — Query description
`'Blocks that are Numerically Perturbed'` (default) | character vector

Query description, specified as `'Blocks that are Numerically Perturbed'`. You can add your own description to the query object using this property.

Usage

After creating a `linqueryIsNumericallyPerturbed` query object, you can:

- Find all the blocks in a `LinearizationAdvisor` object that are linearized using numerical perturbation by using the `linqueryIsNumericallyPerturbed` query directly with the `find` command.

- Create a `CompoundQuery` object by logically combining the `linqueryIsNumericallyPerturbed` query with other query objects.

Object Functions

find Find blocks in linearization results that match specific criteria

Examples

Find All Numerically Perturbed Blocks

Load the Simulink model.

```
mdl = 'scdpendulum';
load_system(mdl)
```

Linearize the model and obtain the `LinearizationAdvisor` object.

```
opts = linearizeOptions('StoreAdvisor',true);
io = getlinio(mdl);
[sys,op,info] = linearize(mdl,io,opts);
advisor = info.Advisor;
```

Create query object, and find all numerically perturbed blocks.

```
qPert = linqueryIsNumericallyPerturbed;
advPert = find(advisor,qPert)
```

```
advPert =
  LinearizationAdvisor with properties:

               Model: 'scdpendulum'
      OperatingPoint: [1x1 opcond.OperatingPoint]
    BlockDiagnostics: [1x4 linearize.advisor.BlockDiagnostic]
           QueryType: 'Perturbation'
```

Alternative Functionality

App

You can also create custom queries for finding specific blocks in linearization results using the Linearization Advisor in the **Model Linearizer**. For more information, see “Find Blocks in Linearization Results Matching Specific Criteria” on page 4-37.

See Also

Objects

`LinearizationAdvisor` | `CompoundQuery`

Functions

`find`

Topics
“Find Blocks in Linearization Results Matching Specific Criteria” on page 4-37
“Troubleshoot Linearization Results at Command Line” on page 4-28

Introduced in R2017b

linqueryIsOnPath

Query object for finding blocks that are on the linearization path

Description

`linqueryIsOnPath` creates a custom query object for finding all the blocks in a linearization result that are on the linearization path.

When you linearize a Simulink model, you can create a `LinearizationAdvisor` object that contains diagnostic information about individual block linearizations. To find block linearizations that satisfy specific criteria, you can use the `find` function with custom query objects. Alternatively, you can analyze linearization diagnostics using the Linearization Advisor in the **Model Linearizer**. For more information on finding specific blocks in linearization results, see “Find Blocks in Linearization Results Matching Specific Criteria” on page 4-37.

Creation

Syntax

```
query = linqueryIsOnPath
```

Description

`query = linqueryIsOnPath` creates a query object for finding all the blocks in a `LinearizationAdvisor` object that are on the linearization path.

Properties

`QueryType` — Query type
`'On Linearization Path'` (default) | character vector

Query type, specified as `'On Linearization Path'`.

`Description` — Query description
`'Blocks on the Linearization Path'` (default) | character vector

Query description, specified as `'Blocks on the Linearization Path'`. You can add your own description to the query object using this property.

Usage

After creating a `linqueryIsOnPath` query object, you can:

- Find all the blocks in a `LinearizationAdvisor` object that are on the linearization path by using the `linqueryIsOnPath` query directly with the `find` command.
- Create a `CompoundQuery` object by logically combining the `linqueryIsOnPath` query with other query objects.

Object Functions

find Find blocks in linearization results that match specific criteria

Examples

Find All Blocks On Linearization Path

Load the Simulink model.

```
mdl = 'scdspeed';
load_system(mdl)
```

Linearize the model and obtain the `LinearizationAdvisor` object.

```
opts = linearizeOptions('StoreAdvisor',true);
io(1) = linio('scdspeed/throttle (degrees)',1,'input');
io(2) = linio('scdspeed/rad//s to rpm',1,'output');
[sys,op,info] = linearize(mdl,io,opts);
advisor = info.Advisor;
```

Create query object, and find all the linearized blocks on the linearization path.

```
qPath = linqueryIsOnPath;
advPath = find(advisor,qPath)
```

```
advPath =
  LinearizationAdvisor with properties:

               Model: 'scdspeed'
      OperatingPoint: [1x1 opcond.OperatingPoint]
    BlockDiagnostics: [1x26 linearize.advisor.BlockDiagnostic]
           QueryType: 'On Linearization Path'
```

Alternative Functionality

App

You can also create custom queries for finding specific blocks in linearization results using the Linearization Advisor in the **Model Linearizer**. For more information, see “Find Blocks in Linearization Results Matching Specific Criteria” on page 4-37.

See Also

Objects

`LinearizationAdvisor` | `CompoundQuery`

Functions

`find` | `highlight`

Topics

“Find Blocks in Linearization Results Matching Specific Criteria” on page 4-37
“Troubleshoot Linearization Results at Command Line” on page 4-28

Introduced in R2017b

linqueryIsZero

Query object for finding blocks that linearize to zero

Description

`linqueryIsZero` creates a custom query object for finding all the blocks in a linearization result that linearize to zero.

When you linearize a Simulink model, you can create a `LinearizationAdvisor` object that contains diagnostic information about individual block linearizations. To find block linearizations that satisfy specific criteria, you can use the `find` function with custom query objects. Alternatively, you can analyze linearization diagnostics using the Linearization Advisor in the **Model Linearizer**. For more information on finding specific blocks in linearization results, see “Find Blocks in Linearization Results Matching Specific Criteria” on page 4-37.

Creation

Syntax

```
query = linqueryIsZero
```

Description

`query = linqueryIsZero` creates a query object for finding all the blocks in a `LinearizationAdvisor` object that linearize to zero.

Properties

`QueryType` — Query type
`'Linearized to Zero'` (default) | character vector

Query type, specified as `'Linearized to Zero'`.

`Description` — Query description
`'Blocks Linearized to Zero'` (default) | character vector

Query description, specified as `'Blocks Linearized to Zero'`. You can add your own description to the query object using this property.

Usage

After creating a `linqueryIsZero` query object, you can:

- Find all the blocks in a `LinearizationAdvisor` object that linearize to zero by using the `linqueryIsZero` query directly with the `find` command.
- Create a `CompoundQuery` object by logically combining the `linqueryIsZero` query with other query objects.

Object Functions

find Find blocks in linearization results that match specific criteria

Examples

Find All Blocks That Linearize to Zero

Load the Simulink model.

```
mdl = 'scdpendulum';
load_system(mdl)
```

Linearize the model and obtain the `LinearizationAdvisor` object.

```
opts = linearizeOptions('StoreAdvisor',true);
io = getlinio(mdl);
[sys,op,info] = linearize(mdl,io,opts);
advisor = info.Advisor;
```

Create query object, and find all blocks that linearize to zero.

```
qZero = linqueryIsZero;
advZero = find(advisor,qZero)
```

```
advZero =
  LinearizationAdvisor with properties:

               Model: 'scdpendulum'
      OperatingPoint: [1x1 opcond.OperatingPoint]
    BlockDiagnostics: [1x3 linearize.advisor.BlockDiagnostic]
           QueryType: 'Linearized to Zero'
```

Alternative Functionality

App

You can also create custom queries for finding specific blocks in linearization results using the Linearization Advisor in the **Model Linearizer**. For more information, see “Find Blocks in Linearization Results Matching Specific Criteria” on page 4-37.

See Also

Objects

`LinearizationAdvisor` | `CompoundQuery`

Functions

`find`

Topics

“Find Blocks in Linearization Results Matching Specific Criteria” on page 4-37
“Troubleshoot Linearization Results at Command Line” on page 4-28

Introduced in R2017b

Model Advisor Checks

Simulink Control Design Checks

Identify time-varying source blocks interfering with frequency response estimation

Identify all time-varying source blocks in the signal path of any output linearization point marked in the Simulink model.

Description

Frequency response estimation uses the steady-state response of a Simulink model to a specified input signal. Time-varying source blocks in the signal path prevent the response from reaching steady-state. In addition, when such blocks appear in the signal path, the resulting response is not purely a response to the specified input signal. Thus, time-varying source blocks can interfere with accurate frequency response estimation.

This check finds and reports all the time-varying source blocks which appear in the signal path of any output linearization output points currently marked on the Simulink model. The report:

- Includes blocks in subsystems and in referenced models that are in normal simulation mode
- Excludes any blocks specified as `BlocksToHoldConstant` in the `frestimateOptions` object you enter as the input parameter

For more information about the algorithm that identifies time-varying source blocks, see the `frest.findSources` reference page.

Available with Simulink Control Design.

Input Parameters

FRESTIMATE options object to compare results against

Provide the paths of any blocks to exclude from the check. Specify the block paths as an array of `Simulink.BlockPath` objects. This array is stored in the `BlocksToHoldConstant` field of an option set you create with `frestimateOptions`. See the `frestimateOptions` reference page for more information.

Results and Recommended Actions

Condition	Recommended Action

Source blocks exist whose output reaches linearization output points currently marked on the model.	Consider holding these source blocks constant during frequency response estimation.
	
	Use the `frest.findSources` command to identify time-varying source blocks at the command line. Then use the `BlocksToHoldConstant` option of `frestimateOptions` to pass these blocks to the `frestimate` command. For example,
	
	`% Get linearization I/Os from the model.`
	`mdl = 'scdengine';`
	`io = getlinio(mdl);`
	`% Find time-varying source blocks.`
	`blks = frest.findSources(mdl,io);`
	`% Create options set with blocks to hold constant.`
	`opts = frestimateOptions;`
	`opts.BlocksToHoldConstant = blks;`
	`% Run estimation with the options.`
	`in = frest.Sinestream;`
	`sysest = frestimate(mdl,io,in,opts);`
	
	For more information and examples, see the `frest.findSources` and `frestimateOptions` reference pages.

Tip

Sometimes, the model includes referenced models containing source blocks in the signal path of an output linearization point. In such cases, set the referenced models to normal simulation mode to ensure that this check locates them. Use the `set_param` command to set `SimulationMode` of any referenced models to `Normal` before running the check.

See Also

- “Estimate Frequency Response Using Model Linearizer” on page 5-6
- “Effects of Time-Varying Source Blocks on Frequency Response Estimation” on page 5-53
- `frest.findSources` reference page
- `frestimateOptions` reference page
- `frestimate` reference page

Apps

Control Design Onramp with Simulink

Self-paced, interactive tutorial included with Simulink Control Design license

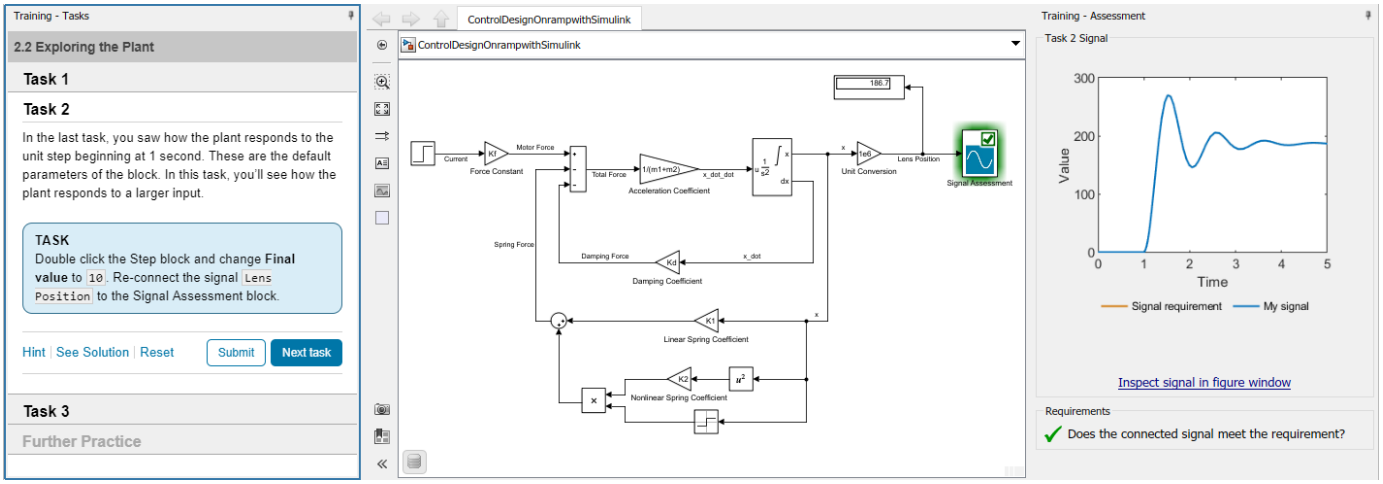
Description

Control Design Onramp with Simulink is a self-paced, interactive tutorial that helps you get started with control design basics in Simulink. It is included with a Simulink Control Design license.

To teach concepts incrementally, Control Design Onramp with Simulink uses tasks. You receive automated assessments and feedback after submitting tasks. Your progress is saved when you exit the training, so you can complete the training in multiple sessions.

Using Control Design Onramp with Simulink, you can:

- Follow basic control design workflows in Simulink
- Explore classical control theories using Simulink Control Design and Control System Toolbox
- Linearize a control system plant with **Model Linearizer**
- Tune a PID controller with **PID Tuner**
- Practice what you learn with a walking robot project



Open the Control Design Onramp with Simulink

- On the Simulink Start Page, click **Control Design Onramp with Simulink**.
- In the MATLAB Command Window, enter `learning.simulink.launchOnramp('controls')`.

See Also

Model Linearizer | PID Tuner

Topics

“Compute Operating Points from Specifications Using Model Linearizer” on page 1-31

“Specify Portion of Model to Linearize” on page 2-10
“Linearize Simulink Model at Model Operating Point” on page 2-54
“PID Controller Tuning in Simulink”

Introduced in R2020b

Model Linearizer

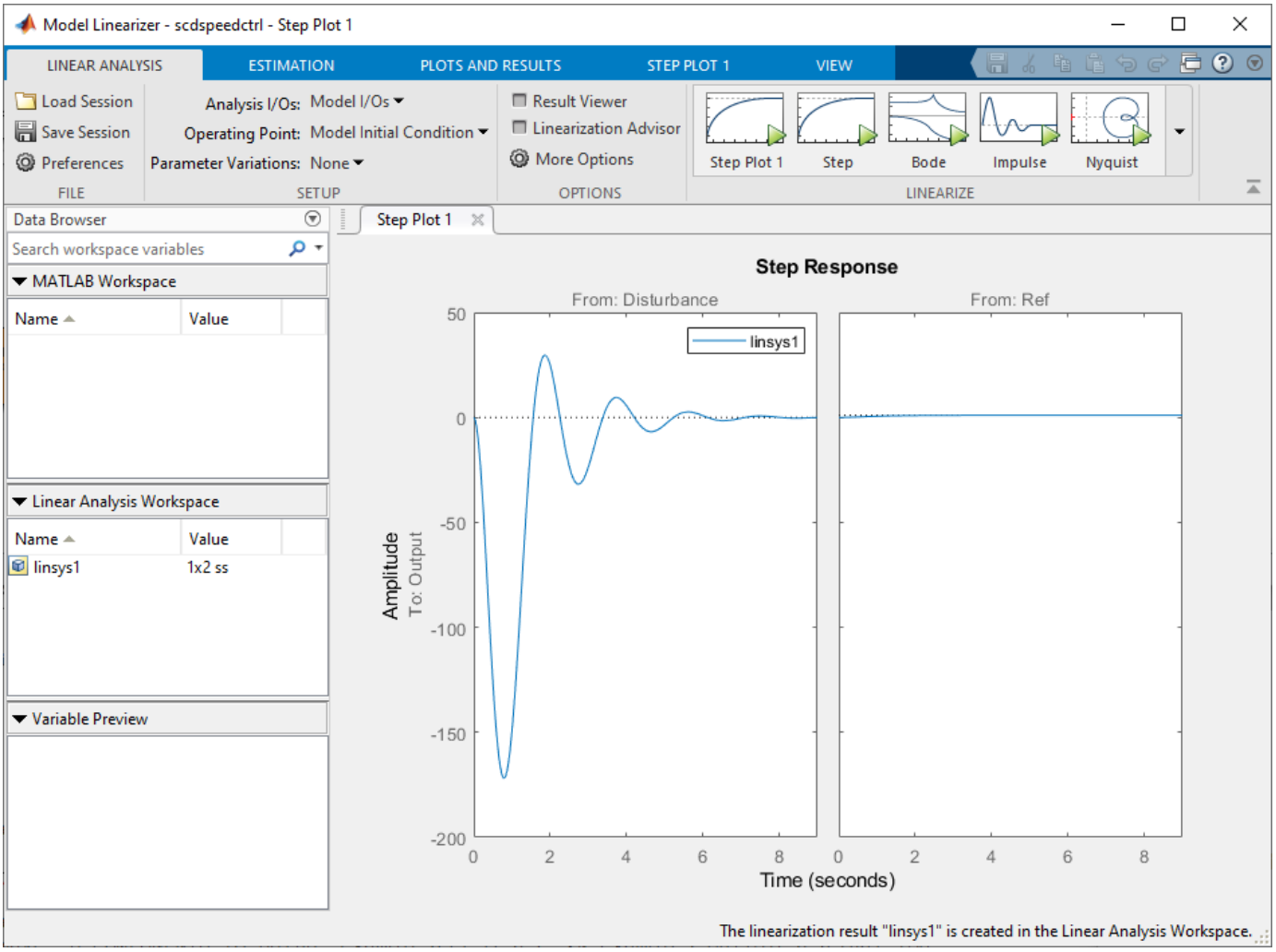
Linearize Simulink models

Description

The **Model Linearizer** lets you perform linear analysis of nonlinear Simulink models.

Using this app you can:

- Interactively linearize models at different operating points
- Interactively obtain operating points by trimming or simulating models
- Perform exact linearization of nonlinear models
- Perform frequency response estimation of nonlinear models
- Batch linearize models for varying parameter values
- Generate MATLAB code for performing linearization tasks
- Generate MATLAB code for computing operating points



Open the Model Linearizer App

- Simulink Toolstrip: On the **Apps** tab, under **Control Systems**, click **Model Linearizer**.
- Simulink Toolstrip: On the **Apps** tab, under **Control Systems**, click **Frequency Response Estimator**.
- Simulink Toolstrip: On the **Linearization** tab, click **Model Linearizer**.
- Simulink Toolstrip: On the **Linearization** tab, click **Frequency Response Estimator**.
- Simulink Toolstrip: On the **Linearization** tab, click **Linearize Block**.

Examples

- “Linearize Simulink Model at Model Operating Point” on page 2-54
- “Linearize at Trimmed Operating Point” on page 2-67
- “Linearize at Simulation Snapshot” on page 2-72

- “Estimate Frequency Response Using Model Linearizer” on page 5-6
- “Specify Portion of Model to Linearize in Model Linearizer” on page 2-22
- “Analyze Results Using Model Linearizer Response Plots” on page 2-113
- “Batch Linearize Model for Parameter Value Variations Using Model Linearizer” on page 3-53

Parameters

Linear Analysis Tab

`Analysis I/Os` — Linearization inputs, outputs, and loop openings

`Model I/Os` (default) | linearization I/O set

Linearization inputs, outputs, and loop openings. The currently active I/O set is displayed. To change the I/O set, select one of the following:

- `Model I/Os` — Use the inputs, outputs, and loop openings specified in the Simulink model. For more information on specifying analysis points in your model, see “Specify Portion of Model to Linearize in Simulink Model” on page 2-17.
- `Root Level Inports and Outports` — Use the root level inputs and outputs of the Simulink model.
- `Linearize the Currently Selected Block` — Use the input and output ports of the currently selected block in the Simulink model.
- `Create New Linearization I/Os` — Specify inputs, outputs, and loop openings. For more information, see “Specify Portion of Model to Linearize in Model Linearizer” on page 2-22.
- `Existing I/Os` — Select a previously created I/O set.
- `View/Edit` — View or edit the currently selected operating point. For more information, see “Edit Analysis Points” on page 2-25.

`Operating Point` — Linearization operating point

`Model Initial Condition` (default) | operating point

Linearization operating point. The current operating point is displayed. To change the operating point, select one of the following:

- `Model Initial Condition` — Use the initial conditions defined in the Simulink.
- `Linearize At` — Simulate the model using the model initial conditions, and use the simulation snapshot at the specified time as the operating point. For more information, see “Linearize at Simulation Snapshot” on page 2-72.
- `Linearize at Multiple Points` — Select multiple previously created operating points.
- `Existing Operating points` — Select a previously created operating point.
- `Trim Model` — Compute a steady-state operating point. For more information, see “Compute Steady-State Operating Points” on page 1-5.
- `Take Simulation Snapshot` — Simulate the model using the model initial conditions, and compute an operating point at the specified simulation snapshot times. For more information, see “Find Operating Points at Simulation Snapshots” on page 1-83.
- `View/Edit` — View or edit the currently selected operating point.

`Parameter Variations` — Parameters to vary for batch linearization
`None` (default) | parameters to vary

To vary parameters for batch linearization, in the drop-down list, click `Select parameters to vary`. On the **Parameter Variations** tab, specify the parameters to vary.

For more information, see “Specify Parameter Samples for Batch Linearization” on page 3-43.

`Result Viewer` — Open linearization result viewer
`off` (default) | `on`

Select to display result details after linearization. For more information, see “View Linearized Model Equations Using Model Linearizer” on page 2-111.

`Linearization Advisor` — Collect diagnostic information and open Linearization Advisor
`off` (default) | `on`

Select to collect diagnostic information during linearization and open an **Advisor** tab for interactive troubleshooting of linearization problems. For more information, see “Troubleshoot Linearization Results in Model Linearizer” on page 4-16.

Note The **Model Linearizer** only collects diagnostic information when **Linearization Advisor** is checked before performing a linearization task.

Estimation Tab

`Input Signal` — Estimation input signal
`Sinestream` | `Fixed Sample Time Sinestream` | `Chirp` | `Random` | `PRBS Pseudorandom Binary Sequence`

Estimation input signal. The current input signal is displayed. To change the input signal, select one of the following:

- `Sinestream` — Create an input signal that consists of adjacent sine waves of varying frequencies. For more information, see “Sinestream Input Signals” on page 5-29.
- `Fixed Sample Time Sinestream` — Create a discrete-time sinestream input with a specified sample time.
- `Chirp` — Create a swept-frequency cosine input signal. For more information, see “Chirp Input Signals” on page 5-33.
- `Random` — Create a random input signal.
- `PRBS Pseudorandom Binary Sequence` — Create a pseudorandom binary sequence (PRBS) input signal. For more information, see “PRBS Input Signals” on page 5-36.

`Analysis I/Os` — Linearization inputs, outputs, and loop openings
`Model I/Os` (default) | linearization I/O set

Linearization inputs, outputs, and loop openings. The currently active I/O set is displayed. To change the I/O set, select one of the following:

- `Model I/Os` — Use the inputs, outputs, and loop openings specified in the Simulink model. For more information on specifying analysis points in your model, see “Specify Portion of Model to Linearize in Simulink Model” on page 2-17.

- `Root Level Inports and Outports` — Use the root level inputs and outputs of the Simulink model.
- `Linearize the Currently Selected Block` — Use the input and output ports of the currently selected block in the Simulink model.
- `Create New Linearization I/Os` — Specify inputs, outputs, and loop openings. For more information, see “Specify Portion of Model to Linearize in Model Linearizer” on page 2-22.
- `Existing I/Os` — Select a previously created I/O set.
- `View/Edit` — View or edit the currently selected operating point. For more information, see “Edit Analysis Points” on page 2-25.

`Operating Point` — Linearization operating point
`Model Initial Condition` (default) | operating point

Linearization operating point. The current operating point is displayed. To change the operating point, select one of the following:

- `Model Initial Condition` — Use the initial conditions defined in the Simulink.
- `Linearize At` — Simulate the model using the model initial conditions, and use the simulation snapshot at the specified time as the operating point. For more information, see “Linearize at Simulation Snapshot” on page 2-72.
- `Linearize at Multiple Points` — Select multiple previously created operating points.
- `Existing Operating points` — Select a previously created operating point.
- `Trim Model` — Compute a steady-state operating point. For more information, see “Compute Steady-State Operating Points” on page 1-5.
- `Take Simulation Snapshot` — Simulate the model using the model initial conditions, and compute an operating point at the specified simulation snapshot times. For more information, see “Find Operating Points at Simulation Snapshots” on page 1-83.
- `View/Edit` — View or edit the currently selected operating point.

`Result Viewer` — Open estimation result viewer
`off` (default) | `on`

Select to display result details about the estimation configuration and input signal used for estimation.

`Diagnostic Viewer` — Collect diagnostic information and open diagnostic viewer
`off` (default) | `on`

Select to collect diagnostic information that displays after estimation. You can use the diagnostic information to analyze the estimation result and troubleshoot estimation problems. For more information, see “Analyze Estimated Frequency Response” on page 5-18.

Note The **Model Linearizer** only collects diagnostic information when **Diagnostic Viewer** is selected before performing an estimation task.

See Also

Apps
Steady State Manager

Functions
`linearize` | `frestimate` | `findop`

Topics
“Linearize Simulink Model at Model Operating Point” on page 2-54
“Linearize at Trimmed Operating Point” on page 2-67
“Linearize at Simulation Snapshot” on page 2-72
“Estimate Frequency Response Using Model Linearizer” on page 5-6
“Specify Portion of Model to Linearize in Model Linearizer” on page 2-22
“Analyze Results Using Model Linearizer Response Plots” on page 2-113
“Batch Linearize Model for Parameter Value Variations Using Model Linearizer” on page 3-53

Introduced in R2011b

Steady State Manager

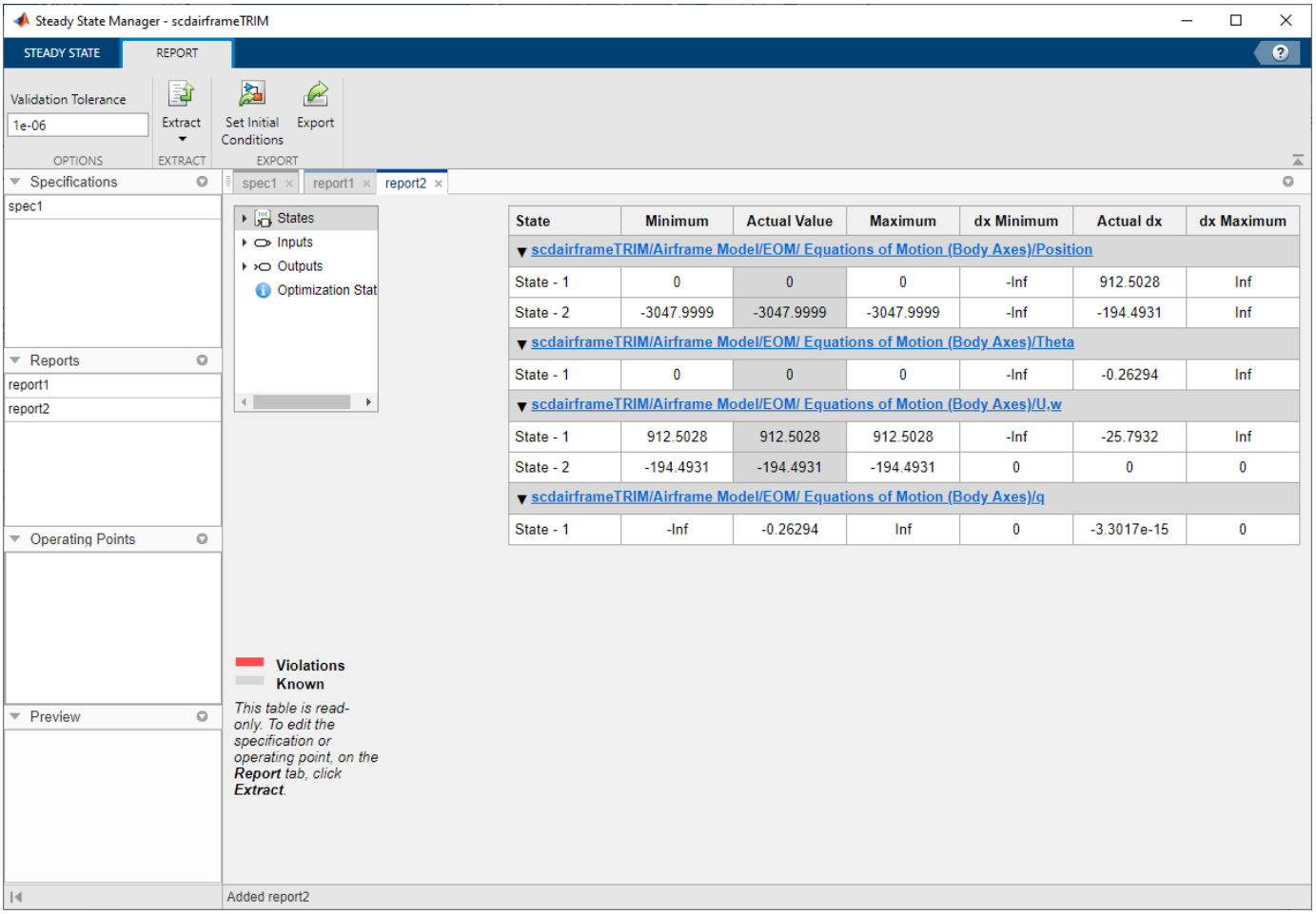
Find operating points for Simulink models

Description

The **Steady State Manager** lets you compute steady-state operating points for Simulink models.

Using this tool you can:

- Interactively obtain operating points from state, input, and output specifications
- Validate operating points against specifications
- Interactively obtain operating points from simulation snapshots
- Generate MATLAB code for computing operating points



State	Minimum	Actual Value	Maximum	dx Minimum	Actual dx	dx Maximum
▼ scdairframeTRIM/Airframe Model/EOM/ Equations of Motion (Body Axes)/Position						
State - 1	0	0	0	-Inf	912.5028	Inf
State - 2	-3047.9999	-3047.9999	-3047.9999	-Inf	-194.4931	Inf
▼ scdairframeTRIM/Airframe Model/EOM/ Equations of Motion (Body Axes)/Theta						
State - 1	0	0	0	-Inf	-0.26294	Inf
▼ scdairframeTRIM/Airframe Model/EOM/ Equations of Motion (Body Axes)/U,w						
State - 1	912.5028	912.5028	912.5028	-Inf	-25.7932	Inf
State - 2	-194.4931	-194.4931	-194.4931	0	0	0
▼ scdairframeTRIM/Airframe Model/EOM/ Equations of Motion (Body Axes)/q						
State - 1	-Inf	-0.26294	Inf	0	-3.3017e-15	0

Open the Steady State Manager App

- Simulink Toolstrip: On the **Apps** tab, under **Control Systems**, click the app icon.
- Simulink Toolstrip: On the **Linearization** tab, click **Steady State Manager**.

Examples

- “Compute Operating Points from Specifications Using Steady State Manager” on page 1-20
- “Validate Operating Point Against Specifications” on page 1-38
- “Find Operating Points at Simulation Snapshots” on page 1-83
- “Simulate Simulink Model at Specific Operating Point” on page 1-93
- “Generate MATLAB Code for Operating Point Configuration” on page 1-110
- “Batch Compute Steady-State Operating Points Reusing Generated MATLAB Code” on page 1-80

See Also

Apps
Model Linearizer

Functions
`findop` | `findopOptions`

Topics
“Compute Operating Points from Specifications Using Steady State Manager” on page 1-20
“Validate Operating Point Against Specifications” on page 1-38
“Find Operating Points at Simulation Snapshots” on page 1-83
“Simulate Simulink Model at Specific Operating Point” on page 1-93
“Generate MATLAB Code for Operating Point Configuration” on page 1-110
“Batch Compute Steady-State Operating Points Reusing Generated MATLAB Code” on page 1-80

Introduced in R2018b