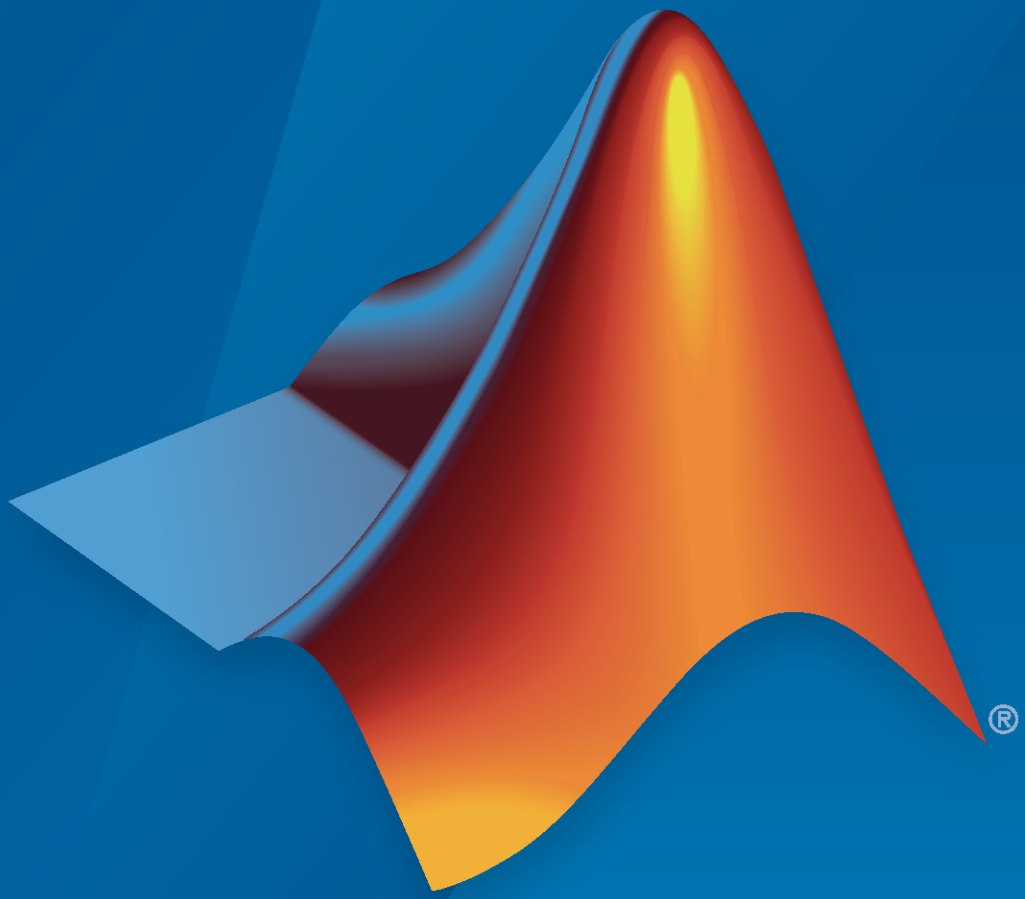
Control System Toolbox™

User's Guide



MATLAB®

R2023a

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

Control System Toolbox™ User's Guide

Revision History

June 2001	Online only	New for Version 5.1 (Release 12.1)
July 2002	Online only	Revised for Version 5.2 (Release 13)
June 2004	Online only	Revised for Version 6.0 (Release 14)
March 2005	Online only	Revised for Version 6.2 (Release 14SP2)
September 2005	Online only	Revised for Version 6.2.1 (Release 14SP3)
March 2006	Online only	Revised for Version 7.0 (Release 2006a)
September 2006	Online only	Revised for Version 7.1 (Release 2006b)
March 2007	Online only	Revised for Version 8.0 (Release 2007a)
September 2007	Online only	Revised for Version 8.0.1 (Release 2007b)
March 2008	Online only	Revised for Version 8.1 (Release 2008a)
October 2008	Online only	Revised for Version 8.2 (Release 2008b)
March 2009	Online only	Revised for Version 8.3 (Release 2009a)
September 2009	Online only	Revised for Version 8.4 (Release 2009b)
March 2010	Online only	Revised for Version 8.5 (Release 2010a)
September 2010	Online only	Revised for Version 9.0 (Release 2010b)
April 2011	Online only	Revised for Version 9.1 (Release 2011a)
September 2011	Online only	Revised for Version 9.2 (Release 2011b)
March 2012	Online only	Revised for Version 9.3 (Release 2012a)
September 2012	Online only	Revised for Version 9.4 (Release 2012b)
March 2013	Online only	Revised for Version 9.5 (Release 2013a)
September 2013	Online only	Revised for Version 9.6 (Release 2013b)
March 2014	Online only	Revised for Version 9.7 (Release 2014a)
October 2014	Online only	Revised for Version 9.8 (Release 2014b)
March 2015	Online only	Revised for Version 9.9 (Release 2015a)
September 2015	Online only	Revised for Version 9.10 (Release 2015b)
March 2016	Online only	Revised for Version 10.0 (Release 2016a)
September 2016	Online only	Revised for Version 10.1 (Release 2016b)
March 2017	Online only	Revised for Version 10.2 (Release 2017a)
September 2017	Online only	Revised for Version 10.3 (Release 2017b)
March 2018	Online only	Revised for Version 10.4 (Release 2018a)
September 2018	Online only	Revised for Version 10.5 (Release 2018b)
March 2019	Online only	Revised for Version 10.6 (Release 2019a)
September 2019	Online only	Revised for Version 10.7 (Release 2019b)
March 2020	Online only	Revised for Version 10.8 (Release 2020a)
September 2020	Online only	Revised for Version 10.9 (Release 2020b)
March 2021	Online only	Revised for Version 10.10 (Release 2021a)
September 2021	Online only	Revised for Version 10.11 (Release 2021b)
March 2022	Online only	Revised for Version 10.11.1 (Release 2022a)
September 2022	Online only	Revised for Version 10.12 (Release 2022b)
March 2023	Online only	Revised for Version 10.13 (Release 2023a)

Contents

Linear System Modeling

1 Linear System Model Objects

What Are Model Objects? **1-2**
Model Objects Represent Linear Systems **1-2**
About Model Data **1-2**

Control System Modeling with Model Objects **1-3**

Types of Model Objects **1-5**

Dynamic System Models **1-7**

Static Models **1-9**

Numeric Models **1-10**
Numeric Linear Time Invariant (LTI) Models **1-10**
Identified LTI Models **1-10**
Identified Nonlinear Models **1-11**

Generalized Models **1-12**
Generalized and Uncertain LTI Models **1-12**
Control Design Blocks **1-12**
Generalized Matrices **1-13**

Models with Tunable Coefficients **1-15**
Tunable Generalized LTI Models **1-15**
Modeling Tunable Components **1-15**
Modeling Control Systems with Tunable Components **1-15**
Internal Structure of Generalized Models **1-16**

Sparse Model Basics **1-18**
Model Objects **1-18**
Combining Sparse Models **1-18**
Time-Domain Analysis **1-20**
Frequency-Domain Analysis **1-21**
Continuous and Discrete Conversions **1-21**
Sparse Linearization **1-22**
Other Supported Functionality **1-24**
Limitations **1-24**

LTV and LPV Modeling **1-26**
Types of LTV and LPV Models **1-26**
Limitations of LPV Models **1-27**
Offsets and Initial Conditions **1-27**
Incremental Form of LTV and LPV Models **1-28**
State Consistency and State Transformation **1-29**
Gridded Models and Choice of Sampling Grid **1-29**
Optimizing LPV Models for Fast Simulation and Code Generation . . **1-30**
Other Considerations **1-31**

Using LTV and LPV Models in MATLAB and Simulink **1-32**
Model Objects **1-32**
Gridded LPV Models **1-33**
Sampling and Interpolation **1-33**
Model Interconnection **1-34**
Continuous and Discrete Conversions **1-34**
Time Response Simulation **1-34**
Gain-Scheduled Controller Design **1-35**
LPV System Block **1-35**
Other Supported Functionality **1-35**
Applications of Linear Parameter-Varying Models **1-35**

Linearize Simulink Model to a Sparse Second-Order Model Object **1-37**

Rigid Assembly of Model Components **1-40**

Linear Analysis of Cantilever Beam **1-44**

Linear Analysis of Tuning Fork **1-52**

Using Model Objects **1-60**

References **1-61**

Using the Right Model Representation **1-62**

Approximate Nonlinear Aircraft Pitch Dynamics Using LPV Model **1-68**

LPV Approximation of Boost Converter Model **1-78**

Control Design for Spinning Disks **1-85**

LTV Model of Two-Link Robot **1-93**

LPV Model of Bouncing Ball **1-103**

Gain-Scheduled LQG Controller **1-106**

Analysis of Gain-Scheduled PI Controller **1-113**

Hidden Couplings in Gain-Scheduled Control **1-118**

LPV Model of Magnetic Levitation System **1-127**

LPV Model of Magnetic Levitation Model from Batch Linearization Results **1-133**

Control Design for Wind Turbine **1-140**

LPV Model of Engine Throttle **1-157**

2 Model Creation

Transfer Functions **2-2**
Transfer Function Representations **2-2**
Commands for Creating Transfer Functions **2-2**
Create Transfer Function Using Numerator and Denominator Coefficients **2-3**
Create Transfer Function Model Using Zeros, Poles, and Gain **2-3**

State-Space Models **2-5**
State-Space Model Representations **2-5**
Explicit State-Space Models **2-5**
Descriptor (Implicit) State-Space Models **2-5**
Commands for Creating State-Space Models **2-5**
Create State-Space Model From Matrices **2-6**

Frequency Response Data (FRD) Models **2-8**
Frequency Response Data **2-8**
Commands for Creating FRD Models **2-8**
Create Frequency Response Model from Data **2-8**

Proportional-Integral-Derivative (PID) Controllers **2-11**
Continuous-Time PID Controller Representations **2-11**
Create Continuous-Time Parallel-Form PID Controller **2-11**
Create Continuous-Time Standard-Form PID Controller **2-12**

Two-Degree-of-Freedom PID Controllers **2-13**
Continuous-Time 2-DOF PID Controller Representations **2-13**
2-DOF Control Architectures **2-14**

Discrete-Time Numeric Models **2-18**
Create Discrete-Time Transfer Function Model **2-18**
Other Model Types in Discrete Time Representations **2-18**

Discrete-Time Proportional-Integral-Derivative (PID) Controllers **2-19**
Discrete-Time PID Controller Representations **2-19**
Create Discrete-Time Standard-Form PID Controller **2-20**
Discrete-Time 2-DOF PI Controller in Standard Form **2-20**

MIMO Transfer Functions **2-22**
Concatenation of SISO Models **2-22**

Using the tf Function with Cell Arrays 2-22

MIMO State-Space Models 2-24
MIMO Explicit State-Space Models 2-24
MIMO Descriptor State-Space Models 2-25
State-Space Model of Jet Transport Aircraft 2-26

MIMO Frequency Response Data Models 2-28

Select Input/Output Pairs in MIMO Models 2-30

Time Delays in Linear Systems 2-31
First Order Plus Dead Time Model 2-31
Input and Output Delay in State-Space Model 2-32
Transport Delay in MIMO Transfer Function 2-33
Discrete-Time Transfer Function with Time Delay 2-33

Closing Feedback Loops with Time Delays 2-35

Time-Delay Approximation 2-37
Time-Delay Approximation in Discrete-Time Models 2-37

Time-Delay Approximation in Continuous-Time Open-Loop Model 2-39

Time-Delay Approximation in Continuous-Time Closed-Loop Model 2-43

Approximate Different Delays with Different Approximation Orders 2-47

Convert Time Delay in Discrete-Time Model to Factors of 1/z 2-50

Frequency Response Data (FRD) Model with Time Delay 2-53

Internal Delays 2-55
Why Internal Delays Are Necessary 2-55
Behavior of Models With Internal Delays 2-56
Inside Time Delay Models 2-56
Functions That Support Internal Time Delays 2-57
Functions That Do Not Support Internal Time Delays 2-57
References 2-57

Tunable Low-Pass Filter 2-59

Create Tunable Second-Order Filter 2-60

Create State-Space Model with Both Fixed and Tunable Parameters 2-62

Control System with Tunable Components 2-63

Control System with Multichannel Analysis Points 2-65

Mark Signals of Interest for Control System Analysis and Design **2-68**
Analysis Points **2-68**
Specify Analysis Points for MATLAB Models **2-69**
Specify Analysis Points for Simulink Models **2-69**
Refer to Analysis Points for Analysis and Tuning **2-72**

Model Arrays **2-76**
What Are Model Arrays? **2-76**
Uses of Model Arrays **2-76**
Visualizing Model Arrays **2-76**
Visualizing Selection of Models From Model Arrays **2-77**

Select Models from Array **2-79**

Query Array Size and Characteristics **2-81**

Linear Parameter-Varying Models **2-83**
What are Linear Parameter-Varying Models? **2-83**
Regular vs. Irregular Grids **2-85**
Use Model Arrays to Create Linear Parameter-Varying Models **2-86**
Approximate Nonlinear Systems using LPV Models **2-86**
Applications of Linear Parameter-Varying Models **2-87**

Using LTI Arrays for Simulating Multi-Mode Dynamics **2-89**

Creating Discrete-Time Models **2-94**

Creating Continuous-Time Models **2-97**

Specifying Time Delays **2-103**

Working with Linear Models

3 Data Manipulation

Store and Retrieve Model Data **3-2**
Model Properties **3-2**
Specify Model Properties at Model Creation **3-2**
Examine and Change Properties of an Existing Model **3-2**

Extract Model Coefficients **3-5**
Functions for Extracting Model Coefficients **3-5**
Extracting Coefficients of Different Model Type **3-5**
Extract Numeric Model Data and Time Delay **3-5**
Extract PID Gains from Transfer Function **3-6**

Attach Metadata to Models **3-8**
Specify Model Time Units **3-8**
Interconnect Models with Different Time Units **3-8**
Specify Frequency Units of Frequency-Response Data Model **3-8**
Extract Subsystems of Multi-Input, Multi-Output (MIMO) Models . . . **3-9**
Specify and Select Input and Output Groups **3-10**

Query Model Characteristics **3-12**

Customize Model Display **3-14**
Configure Transfer Function Display Variable **3-14**
Configure Display Format of Transfer Function in Factorized Form **3-15**

Accessing and Modifying the Model Data **3-17**

4 Model Interconnections

Why Interconnect Models? **4-2**

Catalog of Model Interconnections **4-3**
Model Interconnection Commands **4-3**
Arithmetic Operations **4-4**

Numeric Model of SISO Feedback Loop **4-5**

Control System Model with Both Numeric and Tunable Components **4-7**

Multi-Loop Control System **4-9**

Mark Analysis Points in Closed-Loop Models **4-11**

MIMO Control System **4-15**

MIMO Feedback Loop **4-17**

How the Software Determines Properties of Connected Models . . . **4-20**

Rules That Determine Model Type **4-21**

Recommended Model Type for Building Block Diagrams **4-22**

Using FEEDBACK to Close Feedback Loops **4-24**

Preventing State Duplication in System Interconnections **4-28**

5 Model Transformation

Conversion Between Model Types 5-2
Explicit Conversion Between Model Types 5-2
Automatic Conversion Between Model Types 5-2
Recommended Working Representation 5-2

Convert from One Model Type to Another 5-4

Get Current Value of Generalized Model by Model Conversion 5-5

Decompose a 2-DOF PID Controller into SISO Components 5-7

Discretize a Compensator 5-10

Improve Accuracy of Discretized System with Time Delay 5-15

Convert Discrete-Time System to Continuous Time 5-18

Continuous-Discrete Conversion Methods 5-20
Zero-Order Hold 5-20
First-Order Hold 5-22
Impulse-Invariant Mapping 5-22
Tustin Approximation 5-23
Zero-Pole Matching Equivalents 5-26
Least Squares 5-26

Upsample Discrete-Time System 5-28

Choosing a Resampling Command 5-31

Switching Model Representation 5-32

Connecting Models 5-36

Discretizing and Resampling Models 5-47

Discretizing a Notch Filter 5-52

Scaling State-Space Models to Maximize Accuracy 5-59

Sensitivity of Multiple Roots 5-67

6 Model Simplification

Model Reduction Basics 6-2
When to Reduce Model Order 6-2
Model Reduction Tools 6-3
Choosing a Model Reduction Method 6-3

Reduce Model Order Using the Model Reducer App **6-5**

Balanced Truncation Model Reduction **6-13**
Balanced Truncation in the Model Reducer App **6-13**
Balanced Truncation in Other Environments **6-18**

Approximate Model by Balanced Truncation at the Command Line **6-20**

Compare Truncated and DC Matched Low-Order Model Approximations **6-23**

Approximate Model with Unstable or Near-Unstable Pole **6-27**

Frequency-Limited Balanced Truncation **6-31**

Model Reduction in the Live Editor **6-36**

Pole-Zero Simplification **6-43**
Pole-Zero Simplification in the Model Reducer App **6-43**
Pole-Zero Cancellation at the Command Line **6-47**

Mode-Selection Model Reduction **6-50**
Mode Selection in the Model Reducer App **6-50**
Mode Selection at the Command Line **6-54**

Visualize Reduced-Order Models in the Model Reducer App **6-58**
Error Plots **6-58**
Response Plots **6-59**
Plot Characteristics **6-60**
Plot Tools **6-61**

Linear Analysis

7 Time Domain Analysis

Plotting System Responses **7-2**

Time-Domain Responses **7-19**

Time-Domain Response Data and Plots **7-20**

Time-Domain Characteristics on Response Plots **7-22**

Numeric Values of Time-Domain System Characteristics **7-25**

Time-Domain Responses of Discrete-Time Model **7-26**

Time-Domain Responses of MIMO Model **7-28**

Time-Domain Responses of Multiple Models **7-30**

Joint Time-Domain and Frequency-Domain Analysis **7-32**

Response from Initial Conditions **7-36**

Import LTI Model Objects into Simulink **7-39**
Simulate LTI Model in Simulink **7-39**
Import MIMO LTI Model into Simulink **7-40**

Analysis of Systems with Time Delays **7-43**
Considerations to Keep in Mind when Analyzing Systems with Internal Time Delays **7-45**

8 Frequency Domain Analysis

Frequency-Domain Responses **8-2**

Frequency Response of a SISO System **8-3**

Frequency Response of a MIMO System **8-5**

Frequency-Domain Characteristics on Response Plots **8-8**

Numeric Values of Frequency-Domain Characteristics of SISO Model **8-11**

Pole and Zero Locations **8-13**

Assessing Gain and Phase Margins **8-15**

Analyzing Control Systems with Delays **8-26**

Analyzing the Response of an RLC Circuit **8-41**

9 Sensitivity Analysis

Model Array with Single Parameter Variation **9-2**

Model Array with Variations in Two Parameters **9-5**

Study Parameter Variation by Sampling Tunable Model **9-7**

Sensitivity of Control System to Time Delays **9-9**

Absolute Stability for Quantized System **9-11**

10 Passivity and Conic Sectors

About Passivity and Passivity Indices **10-2**

About Sector Bounds and Sector Indices **10-7**

Passivity Indices **10-14**

Parallel Interconnection of Passive Systems **10-18**

Series Interconnection of Passive Systems **10-20**

Feedback Interconnection of Passive Systems **10-23**

Control Design

11 PID Controller Design

PID Controller Design at the Command Line **11-2**

Designing Cascade Control System with PI Controllers **11-7**

Tune 2-DOF PID Controller (Command Line) **11-11**

Tune 2-DOF PID Controller (PID Tuner) **11-16**

PID Controller Types for Tuning **11-26**
Specifying PID Controller Type **11-26**
1-DOF Controllers **11-28**
2-DOF Controllers **11-29**
2-DOF Controllers with Fixed Setpoint Weights **11-30**

PID Controller Tuning in Simulink **11-33**

Design PID Controller Using Estimated Frequency Response **11-42**

Design Family of PID Controllers for Multiple Operating Points . **11-50**

Design PID Controller Using Simulated I/O Data **11-57**

PID Controller Design in the Live Editor **11-73**

Tune PID Controller from Measured Plant Data in the Live Editor **11-80**

Design PID Controller for Disturbance Rejection Using PID Tuner **11-90**

Temperature Control in a Heat Exchanger **11-101**

Control of Processes with Long Dead Time: The Smith Predictor **11-115**

12 Classical Control Design

Choosing a Control Design Approach **12-3**

Control System Designer Tuning Methods **12-5**
Graphical Tuning Methods **12-5**
Automated Tuning Methods **12-5**
Effective Plant for Tuning **12-6**
Select a Tuning Method **12-7**

Design Requirements **12-9**
Add Design Requirements **12-9**
Edit Design Requirements **12-13**
Root Locus and Pole-Zero Plot Requirements **12-14**
Open-Loop and Closed-Loop Bode Diagram Requirements **12-15**
Open-Loop Nichols Plot Requirements **12-17**
Step and Impulse Response Requirements **12-18**

Feedback Control Architectures **12-21**

Design Multiloop Control System **12-23**

Multimodel Control Design **12-32**
Control Design Overview **12-32**
Model Arrays **12-32**
Nominal Model **12-33**
Frequency Grid **12-35**
Design Controller for Multiple Plant Models **12-35**

Bode Diagram Design **12-42**
Tune Compensator For DC Motor Using Bode Diagram Graphical Tuning **12-42**

Root Locus Design **12-55**
Tune Electrohydraulic Servomechanism Using Root Locus Graphical Tuning **12-55**

Nichols Plot Design **12-67**
Tune Compensator for DC Motor Using Nichols Plot Graphical Design **12-67**

Edit Compensator Dynamics **12-78**
Compensator Editor **12-78**
Graphical Compensator Editing **12-80**
Poles and Zeros **12-80**
Lead and Lag Networks **12-81**
Notch Filters **12-81**

Design Compensator Using Automated Tuning Methods **12-83**
Select Tuning Method **12-83**
Select Compensator and Loop to Tune **12-84**
PID Tuning **12-84**
Optimization-Based Tuning **12-89**
LQG Design **12-90**
Loop Shaping **12-91**
Internal Model Control Tuning **12-92**

Analyze Designs Using Response Plots **12-95**
Analysis Plots **12-95**
Editor Plots **12-96**
Plot Characteristics **12-97**
Plot Tools **12-98**
Design Requirements **12-99**

Compare Performance of Multiple Designs **12-101**

Design Hard-Disk Read/Write Head Controller **12-105**

Design Compensator for Plant Model with Time Delays **12-116**

Design Compensator for Systems Represented by Frequency Response Data **12-122**

Design Internal Model Controller for Chemical Reactor Plant . . **12-126**

Design LQG Tracker Using Control System Designer **12-140**

Export Design to MATLAB Workspace **12-149**

Generate Simulink Model for Control Architecture **12-151**

Tune Simulink Blocks Using Compensator Editor **12-153**

Single Loop Feedback/Prefilter Compensator Design **12-158**

Cascaded Multiloop Feedback Design **12-164**

Reference Tracking of DC Motor with Parameter Variations . . . **12-173**

Getting Started with the Control System Designer **12-178**

Compensator Design for a Set of Plant Models **12-186**

Programmatically Initializing the Control System Designer **12-191**

DC Motor Control **12-195**

Feedback Amplifier Design **12-206**

Digital Servo Control of a Hard-Disk Drive **12-223**

Yaw Damper Design for a 747 Jet Aircraft **12-238**

Thickness Control for a Steel Beam **12-251**

Kalman Filtering **12-265**

State Estimation Using Time-Varying Kalman Filter **12-273**

Nonlinear State Estimation of a Degrading Battery System **12-285**

Parameter and State Estimation in Simulink Using Particle Filter Block **12-298**

13 State-Space Control Design

Extended and Unscented Kalman Filter Algorithms for Online State Estimation **13-2**
Extended Kalman Filter Algorithm **13-2**
Unscented Kalman Filter Algorithm **13-4**

Generate Code for Online State Estimation in MATLAB **13-9**
Tunable and Nontunable Object Properties **13-10**

Validate Online State Estimation at the Command Line **13-12**
Examine Output Estimation Error **13-12**
Examine State Estimation Error for Simulated Data **13-13**

Validate Online State Estimation in Simulink **13-14**
Examine Residuals **13-14**
Examine State Estimation Error for Simulated Data **13-14**
Compute Residuals and State Estimation Errors **13-15**

Troubleshoot Online State Estimation **13-17**

Nonlinear State Estimation Using Unscented Kalman Filter and Particle Filter **13-18**

Estimate States of Nonlinear System with Multiple, Multirate Sensors **13-33**

Regulate Pressure in Drum Boiler **13-43**

State Estimation with Wrapped Measurements Using Extended Kalman Filter **13-50**

Detect Replay Attacks in DC Microgrids Using Distributed Watermarking **13-57**

Detect Attack in Cyber-Physical Systems Using Dynamic Watermarking **13-65**

Control System Tuning

14 Control System Tuning

Automated Tuning Overview **14-3**

Choosing an Automated Tuning Approach **14-4**

Automated Tuning Workflow **14-6**

Specify Control Architecture in Control System Tuner **14-7**
About Control Architecture **14-7**
Predefined Feedback Architecture **14-7**
Arbitrary Feedback Control Architecture **14-8**
Control System Architecture in Simulink **14-9**

Open Control System Tuner for Tuning Simulink Model **14-10**
Command-Line Equivalents **14-10**

Specify Operating Points for Tuning in Control System Tuner . . . **14-11**
About Operating Points in Control System Tuner **14-11**
Linearize at Simulation Snapshot Times **14-11**
Compute Operating Points at Simulation Snapshot Times **14-12**
Compute Steady-State Operating Points **14-14**

Specify Blocks to Tune in Control System Tuner **14-17**

View and Change Block Parameterization in Control System Tuner **14-19**
View Block Parameterization **14-19**
Fix Parameter Values or Limit Tuning Range **14-20**
Custom Parameterization **14-21**
Block Rate Conversion **14-22**

Setup for Tuning Control System Modeled in MATLAB **14-25**

How Tuned Simulink Blocks Are Parameterized **14-26**
Blocks With Predefined Parameterization **14-26**
Blocks Without Predefined Parameterization **14-27**
View and Change Block Parameterization **14-27**

Specify Goals for Interactive Tuning **14-28**

Quick Loop Tuning of Feedback Loops in Control System Tuner . **14-33**

Quick Loop Tuning **14-41**
Purpose **14-41**
Description **14-41**
Feedback Loop Selection **14-41**
Desired Goals **14-42**
Options **14-43**
Algorithms **14-43**

Step Tracking Goal **14-44**
Purpose **14-44**
Description **14-44**
Step Response Selection **14-45**
Desired Response **14-45**
Options **14-46**
Algorithms **14-47**

Step Rejection Goal **14-49**
Purpose **14-49**
Description **14-49**
Step Disturbance Response Selection **14-50**
Desired Response to Step Disturbance **14-50**
Options **14-51**
Algorithms **14-51**

Transient Goal **14-53**
Purpose **14-53**
Description **14-53**
Response Selection **14-54**
Initial Signal Selection **14-54**
Desired Transient Response **14-55**
Options **14-55**
Tips **14-56**
Algorithms **14-56**

LQR/LQG Goal **14-58**
Purpose **14-58**
Description **14-58**
Signal Selection **14-58**
LQG Objective **14-59**
Options **14-60**
Tips **14-60**
Algorithms **14-60**

Gain Goal **14-62**
Purpose **14-62**
Description **14-62**
I/O Transfer Selection **14-63**
Options **14-63**
Algorithms **14-64**

Variance Goal **14-66**
Purpose **14-66**
Description **14-66**

I/O Transfer Selection 14-66
Options 14-67
Tips 14-68
Algorithms 14-68

Reference Tracking Goal 14-70
Purpose 14-70
Description 14-70
Response Selection 14-71
Tracking Performance 14-71
Options 14-72
Algorithms 14-73

Overshoot Goal 14-75
Purpose 14-75
Description 14-75
Response Selection 14-76
Options 14-76
Algorithms 14-77

Disturbance Rejection Goal 14-79
Purpose 14-79
Description 14-79
Disturbance Scenario 14-80
Rejection Performance 14-81
Options 14-81
Algorithms 14-82

Sensitivity Goal 14-84
Purpose 14-84
Description 14-84
Sensitivity Evaluation 14-85
Sensitivity Bound 14-85
Options 14-85
Algorithms 14-86

Weighted Gain Goal 14-88
Purpose 14-88
Description 14-88
I/O Transfer Selection 14-88
Weights 14-89
Options 14-89
Algorithms 14-90

Weighted Variance Goal 14-91
Purpose 14-91
Description 14-91
I/O Transfer Selection 14-91
Weights 14-92
Options 14-92
Tips 14-93
Algorithms 14-93

Minimum Loop Gain Goal 14-95
Purpose 14-95

Description 14-95
Open-Loop Response Selection 14-96
Desired Loop Gain 14-96
Options 14-97
Algorithms 14-98

Maximum Loop Gain Goal 14-100
Purpose 14-100
Description 14-100
Open-Loop Response Selection 14-101
Desired Loop Gain 14-101
Options 14-102
Algorithms 14-103

Loop Shape Goal 14-105
Purpose 14-105
Description 14-105
Open-Loop Response Selection 14-106
Desired Loop Shape 14-107
Options 14-107
Algorithms 14-108

Margins Goal 14-110
Purpose 14-110
Description 14-110
Feedback Loop Selection 14-111
Desired Margins 14-111
Options 14-112
Algorithms 14-113

Passivity Goal 14-114
Purpose 14-114
Description 14-114
I/O Transfer Selection 14-115
Options 14-115
Algorithms 14-116

Conic Sector Goal 14-118
Purpose 14-118
Description 14-118
I/O Transfer Selection 14-119
Options 14-119
Tips 14-120
Algorithms 14-121

Weighted Passivity Goal 14-123
Purpose 14-123
Description 14-123
I/O Transfer Selection 14-124
Weights 14-124
Options 14-125
Algorithms 14-126

Poles Goal 14-127
Purpose 14-127

Description 14-127
Feedback Configuration 14-128
Pole Location 14-128
Options 14-129
Algorithms 14-129

Controller Poles Goal 14-131
Purpose 14-131
Description 14-131
Constrain Dynamics of Tuned Block 14-132
Keep Poles Inside the Following Region 14-132
Algorithms 14-132

Manage Tuning Goals 14-134

Generate MATLAB Code from Control System Tuner for Command-Line Tuning 14-135

Interpret Numeric Tuning Results 14-138
Tuning-Goal Scalar Values 14-138
Tuning Results at the Command Line 14-138
Tuning Results in Control System Tuner 14-139
Improve Tuning Results 14-140

Visualize Tuning Goals 14-141
Tuning-Goal Plots 14-141
Difference Between Dashed Line and Shaded Region 14-142
Improve Tuning Results 14-146

Create Response Plots in Control System Tuner 14-147

Examine Tuned Controller Parameters in Control System Tuner 14-152

Compare Performance of Multiple Tuned Controllers 14-154

Create and Configure slTuner Interface to Simulink Model 14-157

Stability Margins in Control System Tuning 14-161
Gain and Phase Margins 14-161
Interpret Gain and Phase Margin Plots 14-161
Simultaneous Gain and Phase Variations 14-162
Algorithm 14-163

Tune Control System at the Command Line 14-166

Speed Up Tuning with Parallel Computing Toolbox Software . . . 14-167

Validate Tuned Control System 14-168
Extract and Plot System Responses 14-168
Validate Design in Simulink Model 14-169

Extract Responses from Tuned MATLAB Model at the Command Line 14-171

15 Loop-Shaping Design

Structure of Control System for Tuning With looptune **15-2**

Set Up Your Control System for Tuning with looptune **15-3**
Set Up Your Control System for looptunein MATLAB **15-3**
Set Up Your Control System for looptune in Simulink **15-3**

Tune MIMO Control System for Specified Bandwidth **15-4**

Tune Feedback Loops Using looptune **15-10**

Decoupling Controller for a Distillation Column **15-15**

Tuning of a Digital Motion Control System **15-26**

16 Gain-Scheduled Controllers

Gain Scheduling Basics **16-2**
Gain Scheduling in Simulink **16-2**
Tune Gain Schedules **16-2**

Model Gain-Scheduled Control Systems in Simulink **16-4**
Model Scheduled Gains **16-4**
Gain-Scheduled Equivalents for Commonly Used Control Elements **16-6**
Custom Gain-Scheduled Control Structures **16-9**
Tunability of Gain Schedules **16-10**

Tune Gain Schedules in Simulink **16-12**
Workflow for Tuning Gain Schedules **16-12**

Plant Models for Gain-Scheduled Controller Tuning **16-14**
Obtaining the Family of Linear Models **16-15**
Set Up for Gain Scheduling by Linearizing at Design Points **16-15**
Sample System at Simulation Snapshots **16-18**
Sample System at Varying Parameter Values **16-18**
Eliminate Samples at Unneeded Design Points **16-19**
LPV Plants in MATLAB **16-19**

Multiple Design Points in slTuner Interface **16-20**
Block Substitution for Plant **16-20**
Multiple Block Substitutions **16-20**
Substituting Blocks that Depend on the Scheduling Variables **16-21**
Resolving Mismatches Between a Block and its Substitution **16-22**
Block Substitution for LPV Blocks **16-23**

Parameterize Gain Schedules **16-24**
Basis Function Parameterization **16-24**

Tunable Gain Surfaces 16-26
Tunable Gain with Two Independent Scheduling Variables 16-27
Tunable Surfaces in Simulink 16-29
Tunable Surfaces in MATLAB 16-31

Change Requirements with Operating Condition 16-33
Define Variable Tuning Goal 16-33
Enforce Tuning Goal at Subset of Design Points 16-35
Exclude Design Points from systune Run 16-35

Validate Gain-Scheduled Control Systems 16-36
Examine Tuned Gain Surfaces 16-36
Visualize Tuning Goals 16-36
Check Linear Performance 16-39
Validate Gain Schedules in Nonlinear System 16-39

Gain-Scheduled Control of a Chemical Reactor 16-41

Tuning of Gain-Scheduled Three-Loop Autopilot 16-55

Trimming and Linearization of the HL-20 Airframe 16-68

Angular Rate Control in the HL-20 Autopilot 16-75

Attitude Control in the HL-20 Autopilot - SISO Design 16-81

Attitude Control in the HL-20 Autopilot - MIMO Design 16-91

MATLAB Workflow for Tuning the HL-20 Autopilot 16-99

17 Control System Tuning Examples - Generalized LTI Models

Tune Control Systems Using systune 17-2

Building Tunable Models 17-9

Active Vibration Control in Three-Story Building 17-15

Vibration Control in Flexible Beam 17-25

Passive Control with Communication Delays 17-34

Tune Phase-Locked Loop Using Loop-Shaping Design 17-41

Feedback Amplifier Design for Voltage-Mode Boost Converter . . . 17-56

18 Control System Tuning Examples

Tuning Multiloop Control Systems 18-2

PID Tuning for Setpoint Tracking vs. Disturbance Rejection 18-11

Time-Domain Specifications 18-20

Frequency-Domain Specifications 18-26

Loop Shape and Stability Margin Specifications 18-34

System Dynamics Specifications 18-39

Configuring Design Requirements 18-41

Validating Results 18-42

Tune Control Systems in Simulink 18-50

Tune a Control System Using Control System Tuner 18-58

Using Parallel Computing to Accelerate Tuning 18-72

Control of a Linear Electric Actuator 18-76

Control of a Linear Electric Actuator Using Control System Tuner 18-85

Multi-Loop PI Control of a Robotic Arm 18-110

Control of an Inverted Pendulum on a Cart 18-125

Digital Control of Power Stage Voltage 18-132

MIMO Control of Diesel Engine 18-141

Tuning of a Two-Loop Autopilot 18-154

Multiloop Control of a Helicopter 18-169

Fixed-Structure Autopilot for a Passenger Jet 18-176

Fault-Tolerant Control of a Passenger Jet 18-187

Passive Control of Water Tank Level 18-196

Tuning for Multiple Values of Plant Parameters 18-206

Customization

19 Preliminaries

Terminology **19-2**

Property and Preferences Hierarchy **19-3**

Ways to Customize Plots **19-4**

20 Setting Toolbox Preferences

Toolbox Preferences Editor **20-2**
- Overview of the Toolbox Preferences Editor **20-2**
- Opening the Toolbox Preferences Editor **20-2**
- Units Pane **20-2**
- Style Pane **20-4**
- Options Pane **20-5**
- SISO Tool Pane **20-5**

21 Setting Tool Preferences

Linear System Analyzer Preferences Editor **21-2**
- Opening the Linear System Analyzer Preference Editor **21-2**
- Units Pane **21-2**
- Style Pane **21-4**
- Options Pane **21-4**
- Parameters Pane **21-5**

22 Customizing Response Plot Properties

Customize Response Plots Using the Response Plots Property Editor **22-2**
- Opening the Property Editor **22-2**
- Overview of Response Plots Property Editor **22-3**
- Labels Pane **22-4**
- Limits Pane **22-4**
- Units Pane **22-5**

Style Pane 22-13
Options Pane 22-14
Editing Subplots Using the Property Editor 22-17

Customizing Response Plots Using Plot Tools 22-18
Properties You Can Customize Using Plot Tools 22-18
Opening and Working with Plot Tools 22-18
Example of Changing Line Color Using Plot Tools 22-18

Customizing Response Plots from the Command Line 22-20
Overview of Customizing Plots from the Command Line 22-20
Obtaining Plot Handles 22-22
Obtaining Plot Options Handles 22-23
Examples of Customizing Plots from the Command Line 22-24
Properties and Values Reference 22-27

Build GUI With Interactive Response-Plot Updates 22-37

23 Design Case Studies

Design Yaw Damper for Jet Transport 23-2
Overview of this Case Study 23-2
Creating the Jet Model 23-2
Computing Open-Loop Poles 23-3
Open-Loop Analysis 23-4
Root Locus Design 23-7
Washout Filter Design 23-10

LQG Regulation: Rolling Mill Case Study 23-14
Overview of this Case Study 23-14
Process and Disturbance Models 23-14
LQG Design for the x-Axis 23-16
LQG Design for the y-Axis 23-20
Cross-Coupling Between Axes 23-21
MIMO LQG Design 23-24

24 Canonical State-Space Realizations

State-Space Realizations 24-2
Modal Form 24-2
Controllable Companion Form 24-2
Observable Companion Form 24-3
Controllable Canonical Form 24-4
Observable Canonical Form 24-4

25 Reliable Computations

Scaling State-Space Models **25-2**
Why Scaling Is Important **25-2**
When to Scale Your Model **25-2**
Manually Scale Your Model **25-2**

26 Linear System Analyzer

Linear System Analyzer Overview **26-2**

Using the Right-Click Menu in the Linear System Analyzer **26-4**
Overview of the Right-Click Menu **26-4**
Setting Characteristics of Response Plots **26-4**

Importing, Exporting, and Deleting Models in the Linear System Analyzer **26-8**
Importing Models **26-8**
Exporting Models **26-8**
Deleting Models **26-9**

Selecting Response Types **26-11**
Methods for Selecting Response Types **26-11**
Right Click Menu: Plot Type **26-11**
Plot Configurations Window **26-11**
Line Styles Editor **26-12**

Analyzing MIMO Models **26-15**
Overview of Analyzing MIMO Models **26-15**
Array Selector **26-15**
I/O Grouping for MIMO Models **26-16**
Selecting I/O Pairs **26-17**

Customizing the Linear System Analyzer **26-19**
Overview of Customizing the Linear System Analyzer **26-19**
Linear System Analyzer Preferences Editor **26-19**

Linear System Modeling

Linear System Model Objects

- “What Are Model Objects?” on page 1-2
- “Control System Modeling with Model Objects” on page 1-3
- “Types of Model Objects” on page 1-5
- “Dynamic System Models” on page 1-7
- “Static Models” on page 1-9
- “Numeric Models” on page 1-10
- “Generalized Models” on page 1-12
- “Models with Tunable Coefficients” on page 1-15
- “Sparse Model Basics” on page 1-18
- “LTV and LPV Modeling” on page 1-26
- “Using LTV and LPV Models in MATLAB and Simulink” on page 1-32
- “Linearize Simulink Model to a Sparse Second-Order Model Object” on page 1-37
- “Rigid Assembly of Model Components” on page 1-40
- “Linear Analysis of Cantilever Beam” on page 1-44
- “Linear Analysis of Tuning Fork” on page 1-52
- “Using Model Objects” on page 1-60
- “References” on page 1-61
- “Using the Right Model Representation” on page 1-62
- “Approximate Nonlinear Aircraft Pitch Dynamics Using LPV Model” on page 1-68
- “LPV Approximation of Boost Converter Model” on page 1-78
- “Control Design for Spinning Disks” on page 1-85
- “LTV Model of Two-Link Robot” on page 1-93
- “LPV Model of Bouncing Ball” on page 1-103
- “Gain-Scheduled LQG Controller” on page 1-106
- “Analysis of Gain-Scheduled PI Controller” on page 1-113
- “Hidden Couplings in Gain-Scheduled Control” on page 1-118
- “LPV Model of Magnetic Levitation System” on page 1-127
- “LPV Model of Magnetic Levitation Model from Batch Linearization Results” on page 1-133
- “Control Design for Wind Turbine” on page 1-140
- “LPV Model of Engine Throttle” on page 1-157

What Are Model Objects?

Model Objects Represent Linear Systems

In Control System Toolbox, System Identification Toolbox™, and Robust Control Toolbox™ software, you represent linear systems as model objects. In System Identification Toolbox, you also represent nonlinear models as model objects. Model objects are specialized data containers that encapsulate model data and other attributes in a structured way. Model objects allow you to manipulate linear systems as single entities rather than keeping track of multiple data vectors, matrices, or cell arrays.

Model objects can represent single-input, single-output (SISO) systems or multiple-input, multiple-output (MIMO) systems. You can represent both continuous- and discrete-time linear systems.

The main families of model objects are:

- **Numeric Models** — Basic representation of linear systems with fixed numerical coefficients. This family also includes identified models that have coefficients estimated with System Identification Toolbox software.
- **Generalized Models** — Representations that combine numeric coefficients with tunable or uncertain coefficients. Generalized models support tasks such as parameter studies or compensator tuning.

About Model Data

The data encapsulated in your model object depends on the model type you use. For example:

- Transfer functions store the numerator and denominator coefficients
- State-space models store the A, B, C, and D matrices that describe the dynamics of the system
- PID controller models store the proportional, integral, and derivative gains

Other model attributes stored as model data include time units, names for the model inputs or outputs, and time delays. For more information about setting and retrieving model attributes, see “Model Attributes”.

Note All model objects are MATLAB® objects, but working with them does not require a background in object-oriented programming. To learn more about objects and object syntax, see “Role of Classes in MATLAB”.

See Also

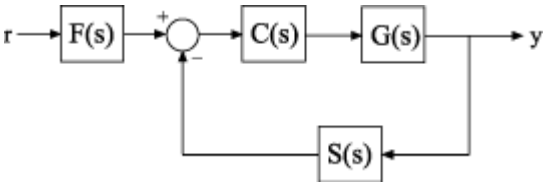
More About

- “Control System Modeling with Model Objects” on page 1-3
- “Types of Model Objects” on page 1-5

Control System Modeling with Model Objects

Model objects can represent individual components of a control architecture, such as the plant, actuators, sensors, or controllers. You can connect model objects to build aggregate models of block diagrams that represent the combined response of multiple elements.

For example, the following control system contains a prefilter *F*, a plant *G*, and a controller *C*, arranged in a single-loop configuration. The model also includes a representation of sensor dynamics, *S*.



You can represent each of the components as a model object. You do not need to use the same type of model object for each component. For example, represent the plant *G* as a zero-pole-gain (`zpk`) model with a double pole at `s` = -1; *C* as a PID controller, and *F* and *S* as transfer functions:

```
G = zpk([],[-1,-1],1);
C = pid(2,1.3,0.3,0.5);
S = tf(5,[1 4]);
F = tf(1,[1 1]);
```

You can then combine these elements build models that represent your control system or the control system as a whole. For example, create the open-loop response *SGC*:

```
open_loop = S*G*C;
```

To build a model of the unfiltered closed-loop response, use the `feedback` command:

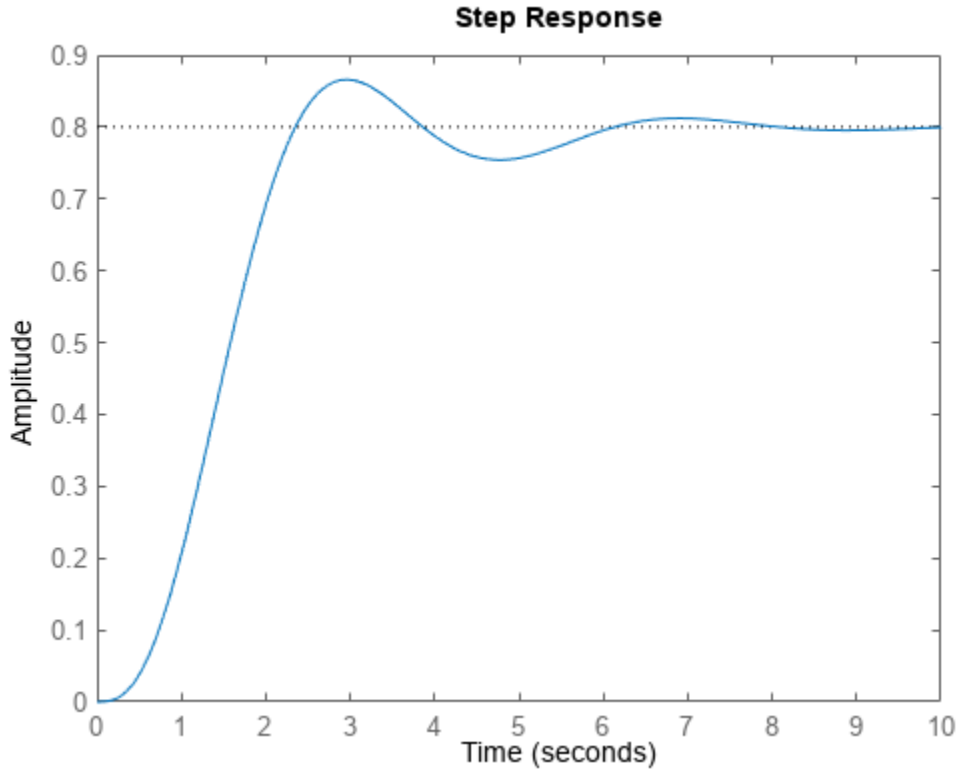
```
T = feedback(G*C,S);
```

To model the entire closed-loop system response from *r* to *y*, combine *T* with the filter transfer function:

```
Try = T*F;
```

The results `open_loop`, `T`, and `Try` are also linear model objects. You can operate on them with Control System Toolbox™ control design and analysis commands. For example, plot the step response of the entire system:

```
stepplot(Try)
```



When you combine Numeric LTI models, the resulting Numeric LTI model represents the aggregate system. The resulting model does not retain the original data from the combined components. For example, `T` does not separately keep track of the dynamics of the components `G`, `C`, and `S` that are combined to create `T`.

See Also

`feedback`

Related Examples

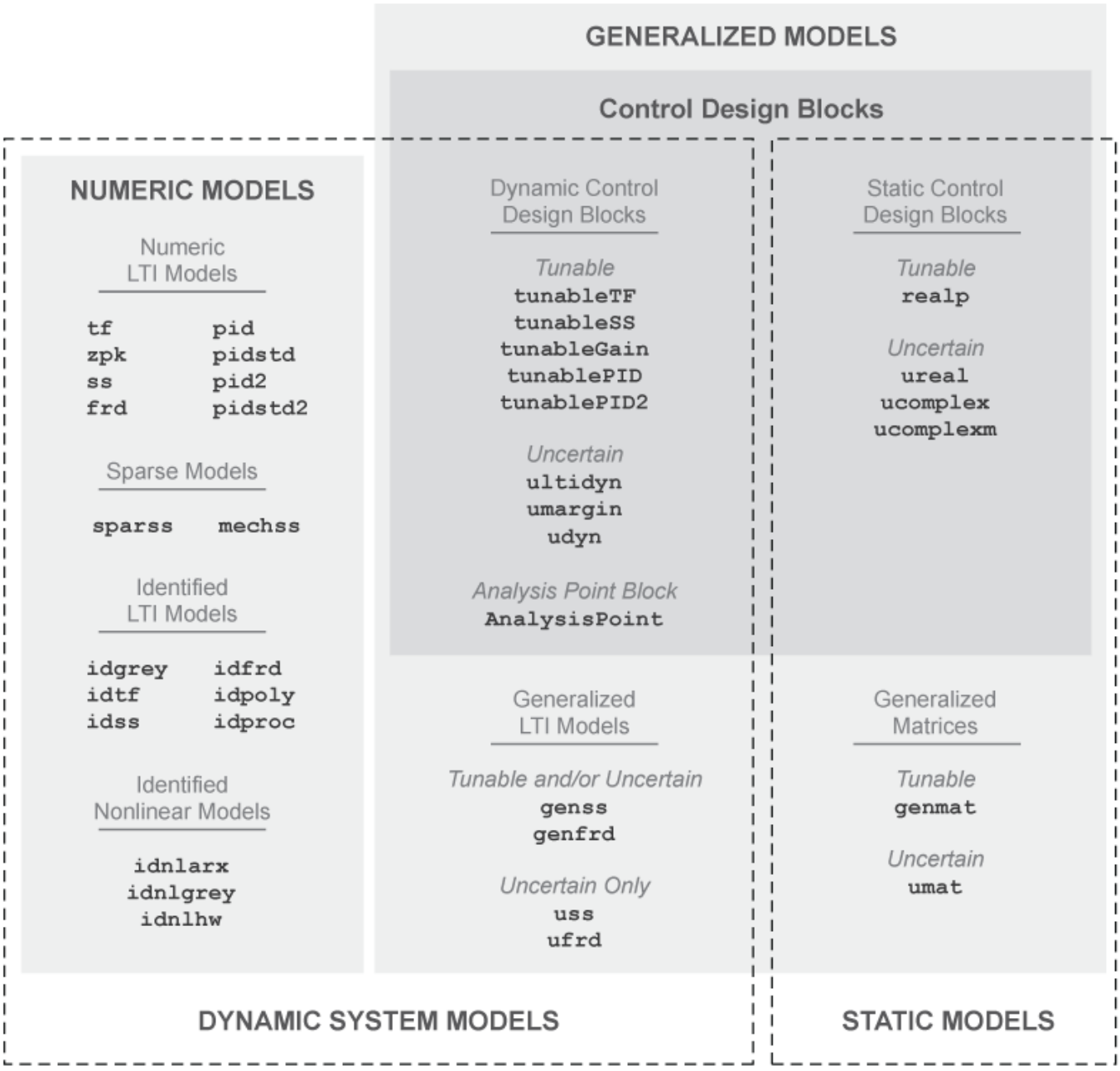
- “Numeric Model of SISO Feedback Loop” on page 4-5
- “Multi-Loop Control System” on page 4-9
- “MIMO Control System” on page 4-15

More About

- “Types of Model Objects” on page 1-5

Types of Model Objects

The following diagram illustrates the relationships between the types of model objects in Control System Toolbox, Robust Control Toolbox, and System Identification Toolbox software. Model types that begin with `id` require System Identification Toolbox software. Model types that begin with `u` require Robust Control Toolbox software. All other model types are available with Control System Toolbox software.



The diagram illustrates the following two overlapping broad classifications of model object types:

- **Dynamic System Models vs. Static Models** — In general, Dynamic System Models represent systems that have internal dynamics, while Static Models represent static input/output relationships.
- **Numeric Models vs. Generalized Models** — Numeric Models are the basic numeric representation of linear systems with fixed coefficients. Generalized Models represent systems with tunable or uncertain components.

See Also

More About

- “What Are Model Objects?” on page 1-2
- “Dynamic System Models” on page 1-7
- “Static Models” on page 1-9
- “Numeric Models” on page 1-10
- “Generalized Models” on page 1-12

Dynamic System Models

Dynamic System Models generally represent systems that have internal dynamics or memory of past states such as integrators, delays, transfer functions, and state-space models.

Most commands for analyzing linear systems, such as `bode`, `margin`, and `linearSystemAnalyzer`, work on most Dynamic System Model objects. For Generalized Models, analysis commands use the current value of tunable parameters and the nominal value of uncertain parameters. Commands that generate response plots display random samples of uncertain models.

The following table lists the Dynamic System Models.

Model Family	Model Types
Numeric LTI models — Basic numeric representation of linear systems	`tf`
	`zpk`
	`ss`
	`frd`
	`pid`
	`pidstd`
	`pid2`
	`pidstd2`
Sparse State-Space Models — Represent large sparse state-space models	`mechss`
	`sparss`
LTV and LPV Models — Represent models with varying coefficients	`ltvss`
	`lpvss`
Identified LTI models — Representations of linear systems with tunable coefficients, whose values can be identified using measured input/output data.	`idtf`
	`idss`
	`idfrd`
	`idgrey`
	`idpoly`
	`idproc`
Identified nonlinear models — Representations of nonlinear systems with tunable coefficients, whose values can be identified using input/output data. Limited support for commands that analyze linear systems.	`idnlarx`
	`idnlhw`
	`idnlgrey`
Generalized LTI models — Representations of systems that include tunable or uncertain coefficients	`genss`
	`genfrd`
	`uss`
	`ufrd`
Dynamic Control Design Blocks — Tunable, uncertain, or switch analysis points for constructing models of control systems	`tunableGain`
	`tunableTF`

Model Family	Model Types
	`tunableSS`
	`tunablePID`
	`tunablePID2`
	`ultidyn`
	`udyn`
	`AnalysisPoint`

See Also

More About

- “Numeric Linear Time Invariant (LTI) Models” on page 1-10
- “Identified LTI Models” on page 1-10
- “Identified Nonlinear Models” on page 1-11
- “Generalized and Uncertain LTI Models” on page 1-12
- “Control Design Blocks” on page 1-12
- “LTV and LPV Modeling” on page 1-26

Static Models

Static Models represent static input/output relationships and generalize the notions of matrix and numeric array to parametric or uncertain arrays. You can use static models to create parametric or uncertain expressions, and to construct Generalized LTI models whose coefficients are parametric or uncertain expressions. The Static Models family includes:

- Tunable parameters (`realp` objects)
- Generalized matrices (`genmat` objects)
- Uncertain parameters and matrices (`ureal`, `ucomplex`, `ucomplexm`) (requires Robust Control Toolbox software)
- Uncertain matrices (`umat`) objects (requires Robust Control Toolbox software)

For more information about using these objects to create parametric models, see “Models with Tunable Coefficients” on page 1-15. For information about creating uncertain static models, see “Uncertain Real Parameters” (Robust Control Toolbox) and “Uncertain Matrices” (Robust Control Toolbox).

Numeric Models

Numeric Linear Time Invariant (LTI) Models

Numeric LTI models are the basic numeric representation of linear systems or components of linear systems. Use numeric LTI models for modeling dynamic components, such as transfer functions or state-space models, whose coefficients are fixed, numeric values. You can use numeric LTI models for linear analysis or control design tasks.

The following table summarizes the available types of numeric LTI models.

Model Type	Description
`tf`	Transfer function model in polynomial form
`zpk`	Transfer function model in zero-pole-gain (factorized) form
`ss`	State-space model
`frd`	Frequency response data model
`pid`	Parallel-form PID controller
`pidstd`	Standard-form PID controller
`pid2`	Parallel-form two-degree-of-freedom (2-DOF) PID controller
`pidstd2`	Standard-form 2-DOF PID controller

Creating Numeric LTI Models

For information about creating numeric LTI models, see:

- “Transfer Functions” on page 2-2
- “State-Space Models” on page 2-5
- “Frequency Response Data (FRD) Models” on page 2-8
- “Proportional-Integral-Derivative (PID) Controllers” on page 2-11

Applications of Numeric LTI Models

You can use Numeric LTI models to represent block diagram components such as plant or sensor dynamics. By connecting Numeric LTI models together, you can derive Numeric LTI models of block diagrams. Use Numeric LTI models for most modeling, analysis, and control design tasks, including:

- Analyzing linear system dynamics using analysis commands such as `bode`, `step`, or `impulse`.
- Designing controllers for linear systems using the **Control System Designer** app or the PID Tuner GUI.
- Designing controllers using control design commands such as `pidtune`, `rlocus`, or `lqr`/`lqg`.

Identified LTI Models

Identified LTI Models represent linear systems with coefficients that are identified using measured input/output data (requires System Identification Toolbox software). You can specify initial values and constraints for the estimation of the coefficients.

The following table summarizes the available types of identified LTI models.

Model Type	Description
`idtf`	Transfer function model in polynomial form, with identifiable parameters
`idss`	State-space model, with identifiable parameters
`idpoly`	Polynomial input-output model, with identifiable parameters
`idproc`	Continuous-time process model, with identifiable parameters
`idfrd`	Frequency-response model, with identifiable parameters
`idgrey`	Linear ODE (grey-box) model, with identifiable parameters

Identified Nonlinear Models

Identified Nonlinear Models represent nonlinear systems with coefficients that are identified using measured input/output data (requires System Identification Toolbox software). You can specify initial values and constraints for the estimation of the coefficients.

The following table summarizes the available types of identified nonlinear models.

Model Type	Description
`idnlarx`	Nonlinear ARX model, with identifiable parameters
`idnlgrey`	Nonlinear ODE (grey-box) model, with identifiable parameters
`idnlhw`	Hammerstein-Wiener model, with identifiable parameters

Generalized Models

Generalized and Uncertain LTI Models

Generalized LTI Models represent systems having a mixture of fixed coefficients and tunable or uncertain coefficients. Generalized LTI models arise from combining numeric LTI models with Control Design Blocks. For more information about tunable Generalized LTI models and their applications, see “Models with Tunable Coefficients” on page 1-15.

Uncertain LTI Models are a special type of Generalized LTI model that include uncertain coefficients but not tunable coefficients. For more information about using uncertain models, see “Uncertain State-Space Models” (Robust Control Toolbox) and “Create Uncertain Frequency Response Data Models” (Robust Control Toolbox).

Family	Model Type	Description
Generalized LTI Models	`genss`	Generalized LTI model arising from combination of Numeric LTI models (except `frd` models) with Control Design Blocks
	`genfrd`	Generalized LTI model arising from combination `frd` models with Control Design Blocks
Uncertain LTI Models (requires Robust Control Toolbox software)	`uss`	Generalized LTI model arising from combination of Numeric LTI models (except `frd` models) with uncertain Control Design Blocks
	`ufrd`	Generalized LTI model arising from combination `frd` models with uncertain Control Design Blocks

Control Design Blocks

Control Design Blocks are building blocks for constructing tunable or uncertain models of control systems. Combine tunable Control Design Blocks with numeric arrays or Numeric LTI models to create Generalized Matrices or Generalized LTI models that include both fixed and tunable components.

Tunable Control Design Blocks include tunable parameter objects as well as tunable linear models with predefined structure. For more information about using tunable Control Design Blocks, see “Models with Tunable Coefficients” on page 1-15.

If you have Robust Control Toolbox software, you can use uncertain Control Design Blocks to model uncertain parameters or uncertain system dynamics. For more information about using uncertain blocks, see “Uncertain LTI Dynamics Elements” (Robust Control Toolbox), “Uncertain Real Parameters” (Robust Control Toolbox), and “Uncertain Complex Parameters and Matrices” (Robust Control Toolbox).

The following tables summarize the available types of Control Design Blocks.

Dynamic System Model Control Design Blocks

Family	Model Type	Description
Tunable Linear Components	`tunableGain`	Tunable gain block
	`tunableTF`	SISO fixed-order transfer function with tunable coefficients
	`tunableSS`	Fixed-order state-space model with tunable coefficients
	`tunablePID`	One-degree-of-freedom PID controller with tunable coefficients
	`tunablePID2`	Two-degree-of-freedom PID controller with tunable coefficients
Uncertain Dynamics (requires Robust Control Toolbox software)	`ultidyn`	Uncertain linear time-invariant dynamics
	`umargin`	Uncertain gain and phase
	`udyn`	Unmodeled dynamics
Analysis Point Block	`AnalysisPoint`	Points of interest for linear analysis or control system tuning

Static Model Control Design Blocks

Family	Model Type	Description
Tunable Parameter	`realp`	Tunable scalar parameter or matrix
Uncertain Parameters (requires Robust Control Toolbox software)	`ureal`	Uncertain real scalar
	`ucomplex`	Uncertain complex scalar
	`ucomplexm`	Uncertain complex matrix

Generalized Matrices

Generalized Matrices extend the notion of numeric matrices to matrices that include tunable or uncertain values.

Create tunable generalized matrices by building rational expressions involving `realp` parameters. You can use generalized matrices as inputs to `tf` or `ss` to create tunable linear models with structures other than the predefined structures of the Control Design Blocks. Use such models for parameter studies or some compensator tuning tasks.

If you have Robust Control Toolbox software, you can create uncertain matrices by building rational expressions involving uncertain parameters such as `ureal` or `ucomplex`.

Model Type	Description
`genmat`	Generalized matrix that includes parametric or tunable entries

Model Type	Description
`umat` (requires Robust Control Toolbox software)	Generalized matrix that includes uncertain entries

For more information about generalized matrices and their applications, see “Models with Tunable Coefficients” on page 1-15.

Models with Tunable Coefficients

Tunable Generalized LTI Models

Tunable Generalized LTI models represent systems having both fixed and tunable (or parametric) coefficients.

You can use tunable Generalized LTI models to:

- Model a tunable (or parametric) component of a control system, such as a tunable low-pass filter.
- Model a control system that contains both:
  - Fixed components, such as plant dynamics and sensor dynamics
  - Tunable components, such as filters and compensators

You can use tunable Generalized LTI models for parameter studies. For an example, see “Study Parameter Variation by Sampling Tunable Model” on page 9-7. You can also use tunable Generalized LTI models for tuning fixed control structures using tuning commands such as `systune` or the Control System Tuner app. See “Multiloop, Multiobjective Tuning”.

Modeling Tunable Components

Control System Toolbox includes tunable components with predefined structure called “Control Design Blocks” on page 1-12. You can use tunable Control Design Blocks to model any tunable component that fits one of the predefined structures.

To create tunable components with a specific custom structure that is not covered by the Control Design Blocks:

1. Use the tunable real parameter `realp` or the generalized matrix `genmat` to represent the tunable coefficients of your component.
2. Use the resulting `realp` or `genmat` objects as inputs to `tf` or `ss` to model the component. The result is a generalized state-space (`genss`) model of the component.

For examples of creating such custom tunable components, see:

- “Tunable Low-Pass Filter” on page 2-59
- “Create Tunable Second-Order Filter” on page 2-60
- “Create State-Space Model with Both Fixed and Tunable Parameters” on page 2-62

Modeling Control Systems with Tunable Components

To construct a tunable Generalized LTI model representing a control system with both fixed and tunable components:

1. Model the nontunable components of your system using numeric LTI models on page 1-10.
2. Model each tunable component using Control Design Blocks or expressions involving such blocks. See “Modeling Tunable Components” on page 1-15.
3. Use model interconnection commands such as `series`, `parallel` or `connect`, or the arithmetic operators `+`, `-`, `*`, `/`, `\`, and `^`, to combine all the components of your system.

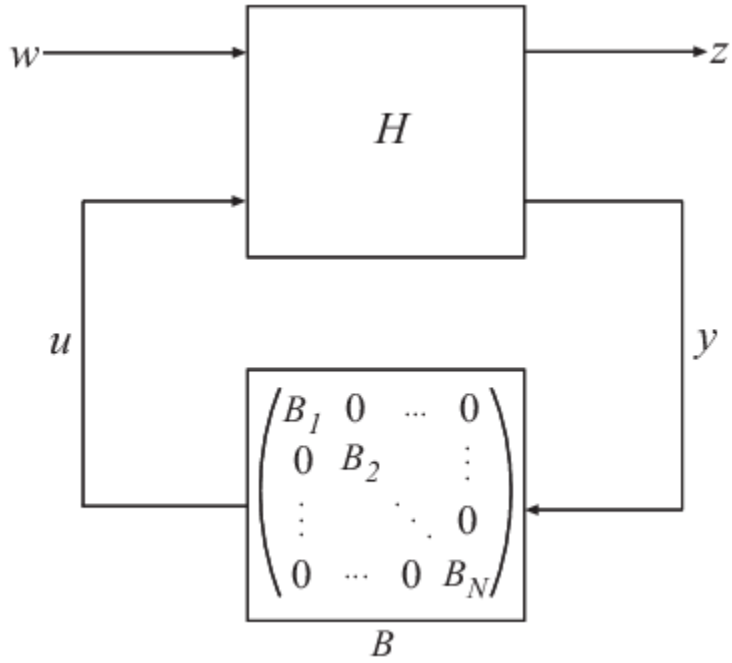
The resulting model is:

- A `genss` model, if none of the nontunable components is a frequency response data model (for example, `frd`)
- A `genfrd` model, if the nontunable component is a `frd` model

For an example of constructing a `genss` model of a control system with both fixed and tunable components, see “Control System with Tunable Components” on page 2-63.

Internal Structure of Generalized Models

A Generalized model separately stores the numeric and parametric portions of the model by structuring the model in Standard Form, as shown in the following illustration.



w and *z* represent the inputs and outputs of the Generalized model.

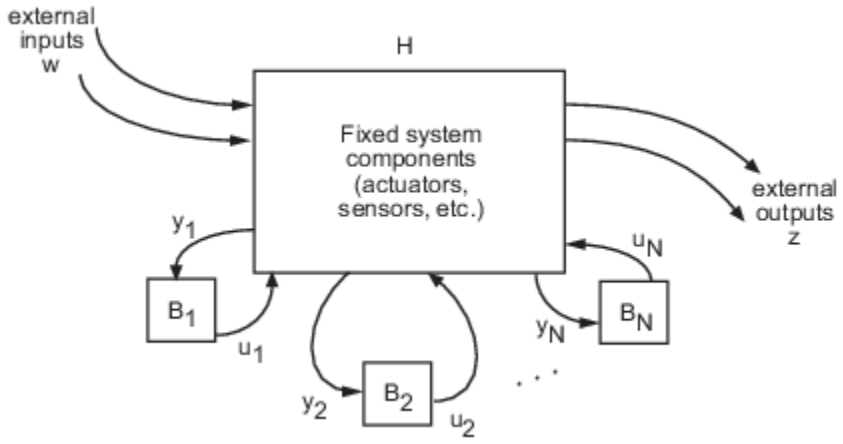
H represents all portions of the Generalized model that have fixed (non-parametric) coefficients. *H* is:

- A state-space (`ss`) model, for `genss` models
- A frequency response data (`frd`) model, for `genfrd` models
- A matrix, for `genmat` models

B represents the parametric components of the Generalized model, which are the Control Design Blocks $B_1, \ldots, B_N$. The `Blocks` property of the Generalized model stores a list of the names of these blocks. If the Generalized model has blocks that occur multiple times in $B_1, \ldots, B_N$, these are only listed once in the `Blocks` property.

To access the internal representation of a Generalized model, including `H` and `B`, use the `getLFTModel` command.

This Standard Form can represent any control structure. To understand why, consider the control structure as an aggregation of fixed-coefficient elements interacting with the parametric elements:



To rewrite this in Standard Form, define

$$u := [u_1, \ldots, u_N]$$
$$y := [y_1, \ldots, y_N],$$

and group the tunable control elements $B_1, \ldots, B_N$ into the block-diagonal configuration C. P includes all the fixed components of the control architecture—actuators, sensors, and other nontunable elements—and their interconnections.

Sparse Model Basics

Model Objects

Use the `sparss` and `mechss` objects to represent sparse first-order and second-order systems, respectively. Such sparse models arise from finite element analysis (FEA) and are useful in fields like structural analysis, fluid flow, heat transfer and electromagnetics. FEA involves analyzing a problem using finite element method (FEM) where a large system is subdivided into numerous, smaller components or finite elements (FE) which are then analyzed separately. These numerous components when combined together result in large sparse models which are computationally expensive and inefficient to be represented by traditional dense model objects like `ss`.

Sparse matrices provide efficient storage of double or logical data that has a large percentage of zeros. While full (or dense) matrices store every single element in memory regardless of value, sparse matrices store only the nonzero elements and their row indices. For this reason, using sparse matrices can significantly reduce the amount of memory required for data storage. For more information, see “Computational Advantages of Sparse Matrices”.

The following table illustrates the types of sparse models that can be represented:

Model Type	Mathematical Representation	Model Object
Continuous-time sparse first-order model	$E\frac{dx}{dt} = A\,x(t) + B\,u(t)$ $y(t) = C\,x(t) + D\,u(t)$	`sparss`
Discrete-time sparse first-order model	$E\,x[k+1] = A\,x[k] + B\,u[k]$ $y[k] = C\,x[k] + D\,u[k]$	`sparss`
Continuous-time sparse second-order model	$\mathrm{M}\,\ddot{q}(t) + \mathrm{C}\,\dot{q}(t) + \mathrm{K}\,q(t) = \mathrm{B}\,u(t)$ $y(t) = \mathrm{F}\,q(t) + \mathrm{G}\,\dot{q}(t) + \mathrm{D}\,u(t)$	`mechss`
Discrete-time sparse second-order model	$\mathrm{M}\,q[k+2] + \mathrm{C}\,q[k+1] + \mathrm{K}\,q[k] = \mathrm{B}\,u[k]$ $y[k] = \mathrm{F}\,q[k] + \mathrm{G}\,q[k+1] + \mathrm{D}\,u[k]$	`mechss`

You can use `sparssdata` and `mechssdata` to access the model matrices in sparse form. You can also use the `spy` command to visualize the sparsity of both first-order and second-order model matrices.

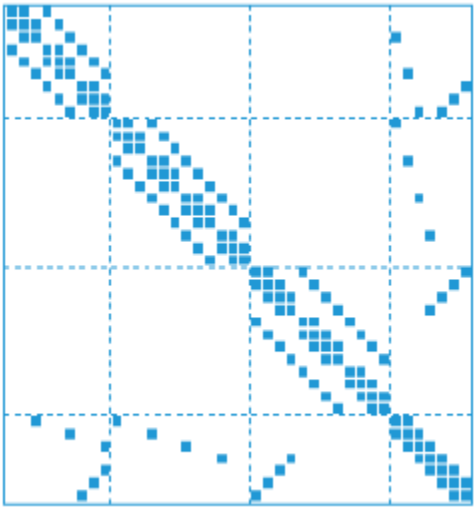
You can also convert any non-FRD model into a `sparss` or a `mechss` object respectively. Conversely, you can use the `full` command to convert sparse models to dense storage `ss`. Converting to dense storage is not recommended for large sparse models as it result in high memory usage and poor performance.

Combining Sparse Models

Signal-Based Connections

All standard signal-based connections listed under “Model Interconnection” are supported for sparse model objects. Interconnecting models using signals allows you to construct models for control systems. You can conceptualize your control system as a block diagram containing multiple interconnected components, such as a plant and a controller connected in a feedback configuration. Using model arithmetic or interconnection commands, you combine models of each of these components into a single model representing the entire block diagram.

In series and feedback connections of `sparss` models, eliminating the internal signals can lead to undesirable fill-in in the `A` matrix. To prevent this, the software eliminates only those signals that do not affect the sparsity of `A` and adds the remaining signals to the state vector. In general, this produces a differential algebraic equation (DAE) model of the interconnection where the `A` matrix has a block arrow structure as depicted in the following figure:



Here, each diagonal block is a sub-component of the sparse model. The last row and column combines the `Interface` and `Signal` groups to capture all couplings and connections between components.

The same rules apply to second-order `mechss` model objects. You can use `showStateInfo` to print a summary of the state vector partition into components, interfaces, and signals

Use the `xsort` command to sort states based on state partition. In the sorted model, all components appear first, followed by the interfaces, and then followed by a single group of all internal signals.

For examples, see the `sparss` and `mechss` reference pages.

Physical Coupling

You can use the `interface` command to specify physical couplings between the components of a `mechss` model. `interface` uses *dual assembly* formulation where the global set of degrees of freedom (DoFs) is retained and an assembly is made by describing coupling constraints at the interface. For rigid couplings between DOFs N_1 of substructure S_1 and DOFs N_2 of substructure S_2, these include:

- Displacement matching: $q(N_1) = q(N_2)$
- Action/reaction principle: $g(N_1) + g(N_2) = 0$ where $g(N_1)$ are the forces exerted by S_2 on S_1, and $g(N_2)$ the forces exerted by S_1 on S_2.

These constraints can be summarized by the equations:

$$M\,\ddot{q} + C\,\dot{q} + K\,q = B\,u + g, \qquad H\,q = 0, \qquad g = -H^T\lambda,$$
$$y = Fq + G\dot{q} + Du,$$

where g is the vector of the interface forces and H is a suitable DOF-selecting matrix.

You can also specify non-rigid couplings between the DOFs using the `interface` command.

For more information, see `interface` and “Rigid Assembly of Model Components” on page 1-40.

Combining Models of Different Types

The following precedence rules apply when combining models of different types:

- Combining sparse models with FRD models yields an `frd` model object
- Combining sparse models with any non-FRD model like `tf`, `ss`, and `zpk` yields a sparse model object
- Combining `sparss` and `mechss` models yields a `mechss` model object.
- Currently, sparse models cannot be combined with generalized or uncertain models, `genss` and `uss`.

For examples, see the `mechss` reference page.

Time-Domain Analysis

All standard time-domain analysis functions listed under “Time and Frequency Domain Analysis” are supported for `sparss` and `mechss` model objects.

You must specify a final time or time vector when using time-domain response functions for sparse models. For example:

```
tf = 10;
step(sys,tf)

t = 0:0.001:1;
initial(sys,x0,t)
```

The time response functions rely on custom fixed-step DAE solvers that have been developed especially for large-scale sparse models. You can configure the DAE solver type and enable parallel computing by using the `SolverOptions` property of the `sparss` and `mechss` model objects. Parallel computing can be used to accelerate `step` or `impulse` response simulation for multi-input models as the response for each input channel is simulated in parallel. Enabling parallel computing requires a Parallel Computing Toolbox™ license.

The available solver options are outlined in the table below:

DAE Solver	Description	Usage
`'trbdf2'`[2]	Fixed step solver with accuracy of `o(h^2)`, where `h` is the step size. `'trbdf2'` is 50% more computationally expensive than `'hht'`. This is the default DAE solver option for both `sparss` and `mechss` models.	Available for both `sparss` and `mechss` models.
`'trbdf3'`	Fixed step solver with accuracy of `o(h^3)`. `'trbdf3'` is 50% more computationally expensive than `'trbdf2'`.	Available for both `sparss` and `mechss` models.

DAE Solver	Description	Usage
`'hht'`[1]	Fixed step solver with accuracy of `o(h^2)`, where `h` is the step size. `'hht'` is the fastest but can run into difficulties with high initial acceleration like the impulse response of a system.	Available for `mechss` models only.

To enable parallel computing and solver selection, use the following syntax:

```
sys.SolverOptions.UseParallel = true;
sys.SolverOptions.DAESolver = 'trbdf3';
```

For an example, see “Linear Analysis of Cantilever Beam” on page 1-44.

Frequency-Domain Analysis

All standard frequency-domain analysis functions listed under “Time and Frequency Domain Analysis” are supported for `sparss` and `mechss` model objects. For frequency response computations, enabling parallel computing speeds the response computation by distributing the set of frequencies across available workers. Enabling parallel computing requires a Parallel Computing Toolbox license.

You must specify a frequency vector when using frequency response functions for sparse models. For example:

```
w = logspace(0,8,1000);
bode(sys,w)
sigma(sys,w)
```

For an example, see “Linear Analysis of Cantilever Beam” on page 1-44.

Continuous and Discrete Conversions

You can convert between continuous-time and discrete-time, and resample `sparss` models using `c2d`, `d2c` and `d2d`. The following table outlines the available methods for `sparss` models:

Method	Description	Usage
`'tustin'`	Bilinear Tustin approximation	Available with `c2d`, `d2c` and `d2d` functions.
`'damped'`	Damped Tustin approximation based on *TRBDF2*[2] formula. This method provides damping at infinity in contrast to the `'tustin'` method, that is, the high frequency dynamics are filtered out and do not contribute to an accumulation of modes near `z = -1` in the discrete model (a source of numerical instability).	Available with `c2d` only.

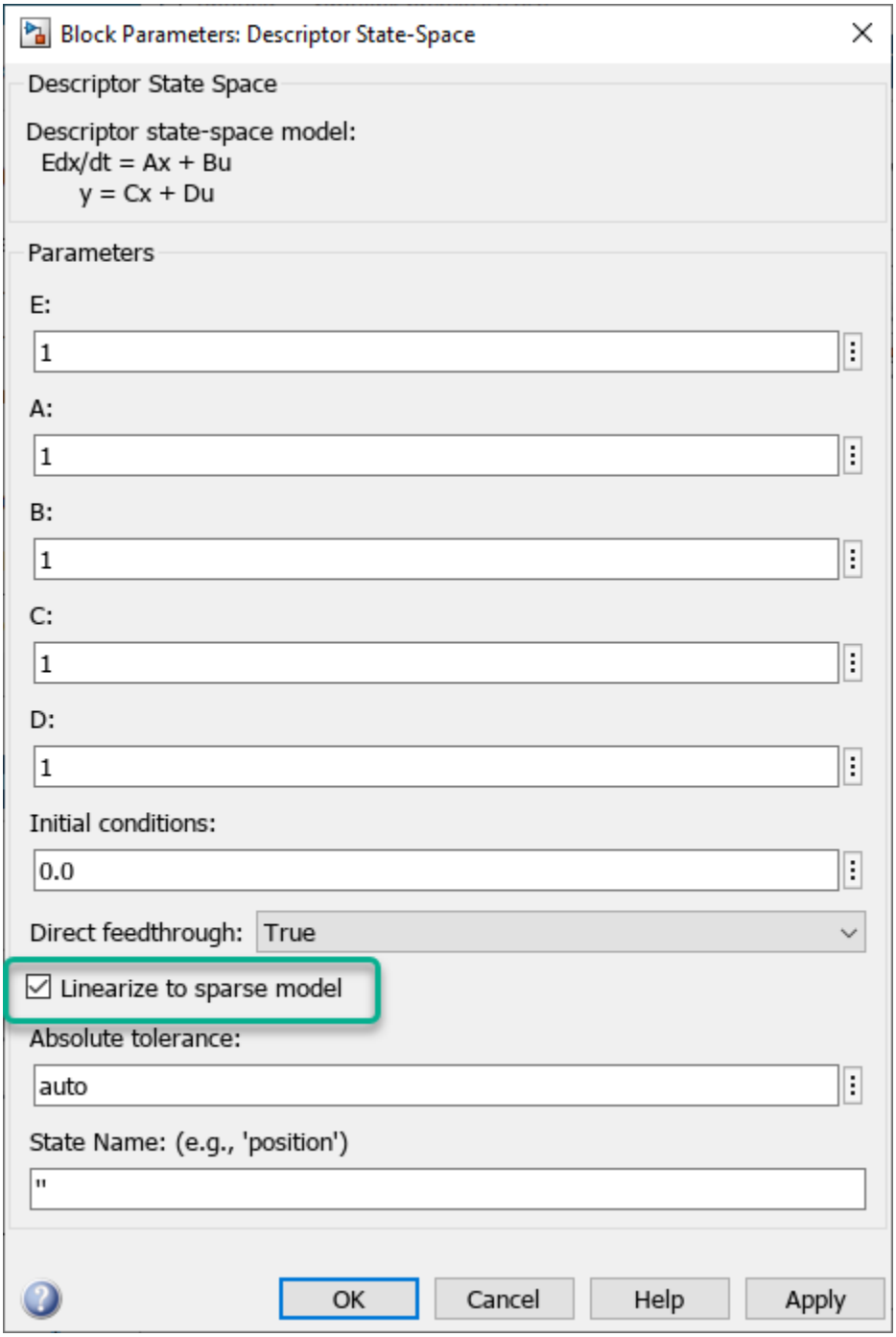
Note Currently, `mechss` models are not supported for continuous and discrete conversions. Convert to `sparss` model form for discretization.

Sparse Linearization

Linearize Simulink Model

You can obtain a sparse linearized model from a Simulink® model when a Descriptor State-Space or Sparse Second Order block is present.

- The Sparse Second Order block is configured to always linearize to a `mechss` model. As a result, the overall linearized model is a second-order sparse model when this block is present.
- Check the **Linearize to sparse model** option in the Descriptor State-Space block to linearize to a `sparss` model. You can also achieve this by setting the `LinearizerToSparse` parameter to `'on'` in the Descriptor State-Space. By default, the **Linearize to sparse model** box is unchecked and the `LinearizerToSparse` is `'off'`. The block-level `LinearizeToSparse` setting is ignored when you specify a replacement for the block, either with the `SCDBlockLinearizationSpecification` block parameter, or the `blocksub` input to `linearize`.



This linearization workflow requires Simulink Control Design™ software.

For an example, see “Linearize Simulink Model to a Sparse Second-Order Model Object” on page 1-37.

Linearize Structural or Thermal PDE Model

You can obtain a sparse linearized model from a structural or a thermal PDE model by using the `linearize` function.

- For a structural analysis model, `linearize` extracts a `mechss` model.
- For a thermal analysis model, `linearize` extracts a `sparss` model.

Use `linearizeInput` to specify the inputs of the linearized model in terms of boundary conditions, loads, or internal heat sources in the PDE model. Use `linearizeOutput` to specify the outputs of the linearized model in terms of regions of the geometry, such as cells (for 3-D geometries only), faces, edges, or vertices.

This linearization workflow requires Partial Differential Equation Toolbox™ software.

For examples, see “Linear Analysis of Cantilever Beam” on page 1-44 and “Linear Analysis of Tuning Fork” on page 1-52.

Other Supported Functionality

Additionally, the following functionality is currently supported for sparse models:

- Low-frequency gain computation using `dcgain`
- Frequency-response evaluation using `evalfr` and `freqresp`
- Inverting models using `inv`
- Implicit-explicit relation conversions using `imp2exp`
- Padé approximations using `pade`
- Input, output and internal delay specification
- I/O selection and indexing

Limitations

The following operations are currently not supported for sparse models:

- Pole/zero and stability margin computation
- Compensator design and tuning
- Model order reduction

You can use the `full` command to convert small sparse models to dense storage `ss` to perform the above operations. Converting to dense storage is not recommended for large sparse models as it may saturate available memory and cause severe performance degradation.

The following limitations exist for sparse linearization:

- When you set the `linearize` option `BlockReduction` to `'off'`, the model is always linearized to a dense `ss` model. This is because block substitution is disabled in this case.
- Sparse linearization is incompatible with block substitutions involving tunable or uncertain models. The **Linearize to sparse model** option should be unchecked when trying to linearize to `genss` or `uss` models.

- Snapshot linearization may not work when the Descriptor State-Space or Sparse Second Order block is present. Since snapshot linearization requires simulation, the simulation capabilities are currently limited for the Descriptor State-Space and Sparse Second Order blocks.

References

[1] H. Hilber, T. Hughes & R. Taylor. " Improved numerical dissipation for time integration algorithms in structural dynamics." *Earthquake Engineering and Structural Dynamics*, vol. 5, no. 3, pp. 283-292, 1977.

[2] M. Hosea and L. Shampine. "Analysis and implementation of TR-BDF2." *Applied Numerical Mathematics*, vol. 20, no. 1-2, pp. 21-37, 1996.

See Also

`sparss` | `mechss` | `showStateInfo` | `getx0` | `full` | `sparssdata` | `mechssdata` | `xsort` | `interface` | `spy` | Descriptor State-Space | Sparse Second Order

More About

- “Computational Advantages of Sparse Matrices”
- “Linearize Simulink Model to a Sparse Second-Order Model Object” on page 1-37
- “Rigid Assembly of Model Components” on page 1-40
- “Linear Analysis of Cantilever Beam” on page 1-44

LTV and LPV Modeling

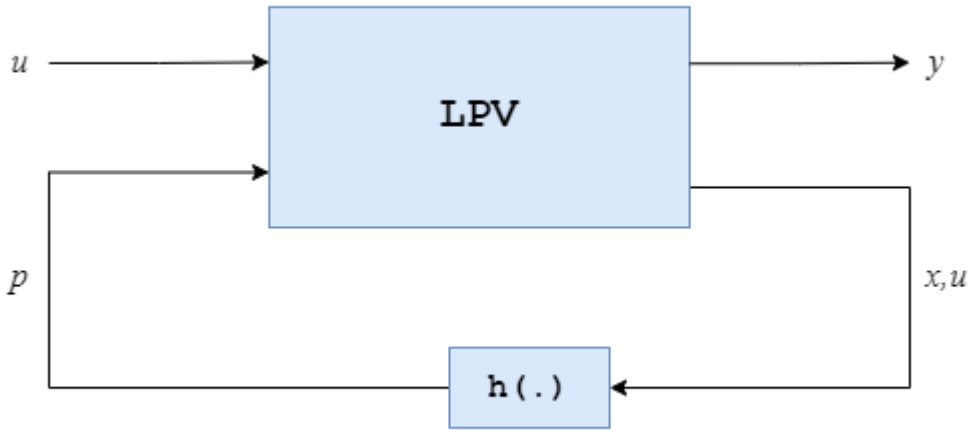
Types of LTV and LPV Models

There are two main categories of linear time-varying (LTV) and linear parameter-varying (LPV) models:

- Models specified by their data function (a MATLAB function), also called *analytic* models. These are often physical models whose state-space equations are mostly linear except for a few time-varying or parameter-dependent terms. Examples of such models include the models in “LPV Model of Bouncing Ball” on page 1-103, “LPV Model of Engine Throttle” on page 1-157, “Analysis of Gain-Scheduled PI Controller” on page 1-113, and “Control Design for Spinning Disks” on page 1-85. They can also come from hand linearization of nonlinear models, as demonstrated in “LPV Model of Magnetic Levitation System” on page 1-127 and “Hidden Couplings in Gain-Scheduled Control” on page 1-118.
- Models interpolating linearization results either along a trajectory or over a grid of operating conditions, also called *data-driven* models. Each linearization captures the local linear dynamics at a given time or around a given operating point, and the interpolation provides smooth transition between these operating regimes. Examples of such models are given in “Approximate Nonlinear Aircraft Pitch Dynamics Using LPV Model” on page 1-68, “LPV Approximation of Boost Converter Model” on page 1-78, “LPV Model of Magnetic Levitation Model from Batch Linearization Results” on page 1-133, “LTV Model of Two-Link Robot” on page 1-93, and “Control Design for Wind Turbine” on page 1-140.

Technically, *gridded* models are a particular type of analytic models where the data function is data driven rather than formula driven. Conceptually, however, the two types of model have different origins and correspond to different modeling workflows and approximation strategies.

Quasi-LPV models are LPV models where the scheduling map or parameter trajectory $p(t)$ is endogenous rather than exogenous, that is, depends on the state x and input u of the model. This creates a feedback loop between the LPV model and the scheduling map.



The `lpvss` object cannot represent an entire quasi-LPV model with its scheduling map. However, you can simulate the response of quasi-LPV models by specifying the scheduling function $p(t) = h(t, x, u)$

as the parameter trajectory. Quasi-LPV models can represent virtually any nonlinear system but do not turn nonlinear systems into linear ones. Moving nonlinearities into the scheduling map can create instabilities and hide difficulties when not done carefully (see “Hidden Couplings in Gain-Scheduled Control” on page 1-118). These difficulties generally do not arise, however, when $p(t)$ changes slowly compared to the dominant system dynamics (see the “Gain-Scheduled LQG Controller” on page 1-106 example for an illustration). Examples involving quasi-LPV simulations include: “LPV Model of Magnetic Levitation System” on page 1-127, “LPV Model of Bouncing Ball” on page 1-103, “LPV Model of Engine Throttle” on page 1-157, “Approximate Nonlinear Aircraft Pitch Dynamics Using LPV Model” on page 1-68, and “Control Design for Wind Turbine” on page 1-140.

Limitations of LPV Models

LPV approximations are most useful for systems whose behavior is nearly linear on a short time scale but nonlinear or time-varying on a longer time horizon. They tend to be less useful for systems with fast, highly nonlinear dynamics. Quasi-LPV models are most useful when the dynamics of the scheduling feedback loop are slow or benign, and least useful when these dynamics dominate those of the LPV model itself (or of any gain-scheduled control loop based on the LPV model alone).

Gridded LPV models based on linearized dynamics cannot represent hard nonlinearities such as saturations and dead zones (see “Hidden Couplings in Gain-Scheduled Control” on page 1-118 for an example). They can nevertheless be coupled with static nonlinearities to recover such behaviors.

Offsets and Initial Conditions

Offsets are an important part of LTV and LPV models. The linearization of the nonlinear model

$$\dot{x} = f(x,u),\ y = g(x,u)$$

around an operating point (x_0,u_0) is

$$\dot{x} = \underbrace{f(x_0,u_0)}_{\delta_0} + A(x-x_0) + B(u-u_0)$$

$$y = \underbrace{g(x_0,u_0)}_{y_0} + C(x-x_0) + D(u-u_0),$$

where

$$A = \frac{\partial f}{\partial x}(x_0,u_0),\ B = \frac{\partial f}{\partial u}(x_0,u_0),\ C = \frac{\partial g}{\partial x}(x_0,u_0),\ D = \frac{\partial g}{\partial u}(x_0,u_0)\,.$$

For the linearization to be a good approximation of the nonlinear maps, it must include the offsets δ_0, x_0, u_0, and y_0. The `linearize` command returns both A, B, C, D and the offsets when using the `StoreOffset` option. This is the basis for constructing most gridded LTV and LPV models. The `ltvss` and `lpvss` objects automatically manage and propagate offset through interconnection operations (`feedback`, `connect`, `seriesparallel`, `lft`) and transformations such as `c2d`, `d2c`, and `d2d`.

Offsets and initial conditions matter for time response simulation. To obtain correct results and good approximations of the nonlinear behavior, it is important to:

1. Consider whether the input signal or step change is absolute or relative to the offset u_0.
2. Correctly initialize the state vector. When the operating conditions (x_0,u_0) are equilibrium conditions, initializing the state vector to x_0 and applying a step change relative to u_0 ensures the

simulation starts from the steady state (x_0,u_0) and approximates the nonlinear behavior around this condition. Use `RespConfig` to specify these values in step. See “Approximate Nonlinear Aircraft Pitch Dynamics Using LPV Model” on page 1-68, “LPV Model of Bouncing Ball” on page 1-103, “LPV Approximation of Boost Converter Model” on page 1-78, “LTV Model of Two-Link Robot” on page 1-93, and “LPV Model of Engine Throttle” on page 1-157 for examples.

3. Specify the parameter trajectory, either explicitly for exogenous parameters (see “LPV Approximation of Boost Converter Model” on page 1-78, “Control Design for Spinning Disks” on page 1-85, “Analysis of Gain-Scheduled PI Controller” on page 1-113, and “Gain-Scheduled LQG Controller” on page 1-106), or implicitly as a function of t, x, u for quasi-LPV simulations (see “LPV Model of Bouncing Ball” on page 1-103, “LPV Model of Engine Throttle” on page 1-157, “Approximate Nonlinear Aircraft Pitch Dynamics Using LPV Model” on page 1-68, and “Control Design for Wind Turbine” on page 1-140). For a critical comparison of these two approaches, see “LPV Model of Magnetic Levitation System” on page 1-127 and “Hidden Couplings in Gain-Scheduled Control” on page 1-118.

Incremental Form of LTV and LPV Models

When linearizing along a trajectory $(x_0(t),u_0(t),y_0(t))$ satisfying the nonlinear equations

$$\dot{x}_0(t) = f(x_0(t), u_0(t)),\ y_0(t) = g(x_0(t), u_0(t)),$$

the linearized model can be expressed as

$$\dot{\delta}_x = A(t)\delta_x + B(t)\delta_u,\ \delta_y = C(t)\delta_x + D(t)\delta_u,$$

where

$$\delta_x = x - x_0,\ \delta_u = u - u_0,\ \delta_y = y - y_0$$

are the deviations from the reference trajectory. See the “LTV Model of Two-Link Robot” on page 1-93 example for an illustration.

This *incremental* or *delta* form looks simpler than the linearized form used for `ltvss` and `lpvss`. However, it is not a valid representation of LPV models in general. Consider, for example, an LPV model constructed from a family of steady-state operating conditions $(x_0(p),u_0(p),y_0(p))$ satisfying for all p:

$$0 = f(x_0(p), u_0(p)),\ y_0(p) = g(x_0(p), u_0(p))\,.$$

For fixed p, you can write the linearized dynamics around $(x_0(p),u_0(p),y_0(p))$ as

$$\dot{\delta}_x = A(p)\delta_x + B(p)\delta_u,\ \delta_y = C(p)\delta_x + D(p)\delta_u,$$

with

$$\delta_x = x - x_0(p),\ \delta_u = u - u_0(p),\ \delta_y = y - y_0(p)\,.$$

This is no longer correct when p varies with time, that is, for an LPV trajectory that takes the system from one steady-state condition to another. In this case, you have:

$$\dot{\delta}_x = \dot{x} - \frac{d}{dt}x_0(p(t))$$

$$\approx \underbrace{f(x_0(p), u_0(p))}_{=\,0} + A(p)\delta_x + B(p)\delta_u - \frac{d}{dt}x_0(p(t))$$

$$= A(p)\delta_x + B(p)\delta_u - \frac{\partial x_0}{\partial p}\dot{p}\,.$$

The incremental form now has an extra term involving the time derivative of $x_0(p(t))$, a quantity that is not readily available. This is why the linearized form is preferred.

State Consistency and State Transformation

When constructing LPV models from arrays of state-space models, ensure that the models are expressed in consistent state coordinates. For example, reordering the states for some models but not others creates state inconsistencies that invalidate the interpolated LPV model. In general, fixed state transformations preserve state consistency, while independent state transformations such as modal decompositions of individual models do not.

This does not mean that time-varying or parameter-varying transformations are disallowed. Given the LTV model

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

the time-varying state transformation $x = T(t)z$ produces the equivalent model

$$\dot{z} = T(^t(A(t)T(t) - \frac{dT}{dx})z + T(^tB(t)u$$
$$y = C(t)T(t)z + D(t)u\,.$$

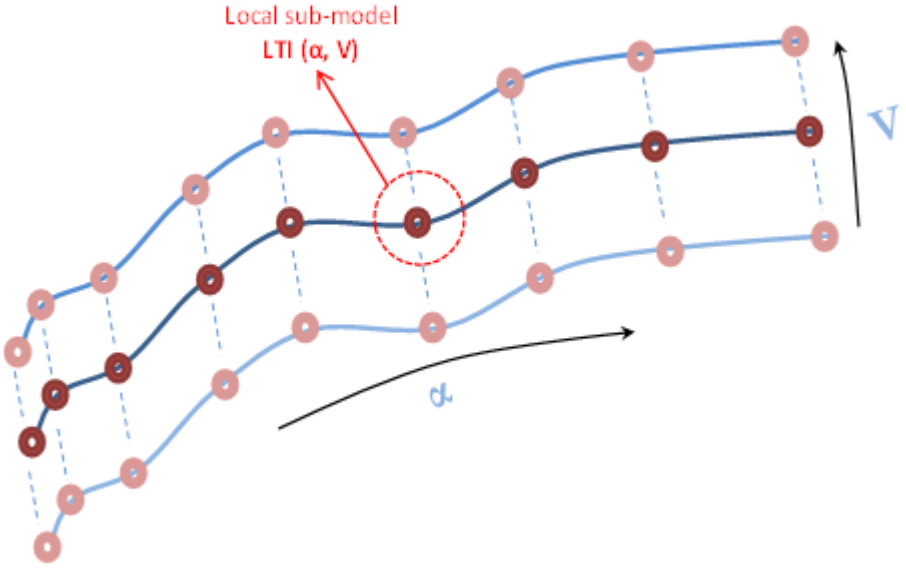
However, the extra term $\frac{dT}{dx}$ explains why independent state transformations are problematic. If $T(t)$ changes abruptly between time samples t_k, the transformed model has an additional term $T(t)^{-1}\frac{dT}{dx}z$ that can be large and is not accounted for when transforming each model individually using $T_k = T(t_k)$:

$$\dot{z} = T_k{}^{-1}A_kT_kz + T_kB_ku,\ y = C_kT_k + D_ku\,.$$

In general, a varying state transformation is fine only when it varies slowly with time or parameters and the term $T(t)^{-1}\frac{dT}{dx}z$ can be neglected.

Gridded Models and Choice of Sampling Grid

A common way of representing LPV models is as an interpolated array of linear state-space models. For example, the aerodynamic behavior of an aircraft (see “Approximate Nonlinear Aircraft Pitch Dynamics Using LPV Model” on page 1-68) is often scheduled over a grid of incidence angle (α) and wind speed (V) values. For each scheduling parameter, a range of values is chosen, such as α = -20:5:20 degrees, V = 700:100:1400 m/s. For each combination of (α,V) values, a linear approximation of the aircraft behavior is obtained. The local models are connected as shown in this figure.



Each point represents a local LTI model, and the connecting curves represent the interpolation rules. The abscissa and ordinate of the surface are the scheduling parameters (α,V).

When using gridded LTV or LPV models to approximate nonlinear dynamics, you must determine what grid density to use and how to best distribute the grid points. The denser the grid, the more accurate the approximation, but also the more memory needed to store the matrices and offsets at all grid points. To reduce memory usage, you can prune the grid while keeping an eye on the accuracy of the LPV approximation for representative use cases (simulations for specific parameter trajectories).

You can start with a dense grid for which the LPV approximation has the desired accuracy. Using `sample`, you can then sample the LPV dynamics and look for parameter ranges where the local LTI dynamics do not change much. Using `ssInterpolant`, you can also try various grid densities and observe how the LPV model fidelity deteriorates. Finally, structural information can guide this process. For example, if the matrices and offsets depend linearly on some parameters, the two extreme values of this parameter are enough to capture parameter variations over an entire interval (assuming linear interpolation). By contrast, the more nonlinear the parameter dependence is, the more grid points are needed.

For gain-scheduled control design, it is customary to use a coarser grid for the controller than for the LPV plant model. Mismatched grids are seamlessly handled by the software. See “Approximate Nonlinear Aircraft Pitch Dynamics Using LPV Model” on page 1-68, “LPV Model of Magnetic Levitation Model from Batch Linearization Results” on page 1-133, and “Control Design for Wind Turbine” on page 1-140 for examples of this practice.

Optimizing LPV Models for Fast Simulation and Code Generation

LPV models can provide low-complexity approximations of high-fidelity models that support fast simulation and code generation. When the original model is a high-fidelity nonlinear model, batch linearization over a dense grid of operating conditions provides the raw material for building a gridded LPV approximation. When the original model is an analytic LPV model, it can be turned into a gridded LPV model using `ssInterpolant`. Using `ssInterpolant`, you can then resample the

gridded LPV model to reduce memory usage as indicated above. Finally, the gridded LPV model can be discretized with `c2d`, since discrete-time models tend to simulate faster and be more amenable to code generation and deployment. See “LTV Model of Two-Link Robot” on page 1-93 and “LPV Approximation of Boost Converter Model” on page 1-78 for examples.

Other Considerations

LTV and LPV models do not commute even in the SISO case. In a gain-scheduled PI controller, for example, placing the gain-scheduled integral gain before or after the integrator is not the same, as illustrated in the “Analysis of Gain-Scheduled PI Controller” on page 1-113 example.

See Also

`lpvss` | `ltvss` | `sample` | `ssInterpolant` | LPV System

Related Examples

- “Using LTV and LPV Models in MATLAB and Simulink” on page 1-32

Using LTV and LPV Models in MATLAB and Simulink

Model Objects

Use `lpvss` and `ltvss` to represent linear parameter-varying (LPV) and linear time-varying (LTV) systems, respectively. For more information about these models, see “LTV and LPV Modeling” on page 1-26.

The following table illustrates the types of varying models that you can represent:

Model Type	Mathematical Representation	Model Object
Continuous-time LPV model	$E(t,p)\dot{x} = \delta_0(t,p) + A(t,p)(x - x_0(t,p)) + B(t,p)(u - u_0(t,p))$ $y(t) = y_0(t,p) + C(t,p)(x - x_0(t,p)) + D(t,p)(u - u_0(t,p))$	`lpvss`
Discrete-time LPV model	$E(k,p_k)x_{k+1} = \delta_0(k,p_k) + A(k,p_k)(x_k - x_0(k,p_k)) + B(k,p_k)(u_k - u_0(k,p_k))$ $y_k = y_0(k,p_k) + C(k,p_k)(x_k - x_0(k,p_k)) + D(k,p_k)(u_k - u_0(k,p_k))$	`lpvss`
Continuous-time LTV model	$E(t)\dot{x} = \delta_0(t) + A(t)(x - x_0(t)) + B(t)(u - u_0(t))$ $y(t) = y_0(t) + C(t)(x - x_0(t)) + D(t)(u - u_0(t))\,.$	`ltvss`
Discrete-time LTV model	$E_k x_{k+1} = \delta_{0k} + A_k(x_k - x_{0k}) + B_k(u_k - u_{0k})$ $y_k = y_{0k} + C_k(x_k - x_{0k}) + D_k(u_k - u_{0k})$	`ltvss`

Data Function

The `ltvss` and `lpvss` objects require you to specify a user-defined MATLAB function for calculating matrices and offsets. This is called the *data function* and must be of the following form.

Model Type	Data Function
Continuous-time `ltvss`	$[A,B,C,D,E,\delta_0,x_0,u_0,y_0,Delays] = f(t)$
Discrete-time `ltvss`	$[A,B,C,D,E,\delta_0,x_0,u_0,y_0,Delays] = f(k)$
Continuous-time `lpvss`	$[A,B,C,D,E,\delta_0,x_0,u_0,y_0,Delays] = f(t,p)$
Discrete-time `lpvss`	$[A,B,C,D,E,\delta_0,x_0,u_0,y_0,Delays] = f(k,p)$

To understand the anatomy of a data function, consider `dataFcnMaglev.m` which is provided with the “LPV Model of Magnetic Levitation System” on page 1-127 example.

```
function [A,B,C,D,E,dx0,x0,u0,y0,Delays] = dataFcnMaglev(~,p)
% MAGLEV example:
% x = [h ; dh/dt]
% p=hbar (equilibrium height)
mb = 0.02; % kg
g = 9.81;
alpha = 2.4832e-5;
A = [0 1;2*g/p 0];
B = [0 ; -2*sqrt(g*alpha/mb)/p];
C = [1 0];  % h
D = 0;
E = [];
```

```
dx0 = [];
x0 = [p;0];
u0 = sqrt(mb*g/alpha)*p;  % ibar
y0 = p;                   % y = h = hbar + (h-hbar)
Delays = [];
```

This data function performs the following tasks:

1. Specifies the inputs of the data function.

   A data function must have time and parameter values as inputs. If your parameters do not explicitly depend on time, you can omit the time input as shown in `dataFcnMaglev.m`. In discrete time, the time input k is the integer index that counts the number of sampling periods T_s , where the absolute time is given by $t = kT_s$. And, p is the parameter value at the time t or sample k, that is, $p(t)$ or $p[k]$.

   You can specify additional inputs to the data function by using anonymous functions. For more information , see “Anonymous Functions”. This helps reduce code redundancy and computation cost. For example, if you wrote a function that also requires parameters `m` and `n` and an `options` structure, you can write the data function as follows.

   ```
   DF = @(t,p) myFunction(t,p(1),..,p(n),m,n,options)
   ```

2. Defines the matrices and offsets.

   The time or parameter matrices and offsets of the LTV or LPV models. You typically obtain these from linearizing nonlinear models around operating conditions.

3. Specifies the values of matrices and offsets as outputs of the data function.

   The data function must return valid values for the outputs `A`, `B`, `C`, `D`, `E`, `dx0`, `x0`, `u0`, `y0`, and `Delays`. You can set all output arguments except `A`, `B`, `C`, `D` to `[]` when absent for (`t`,`p`) values.

   Note For R2023a, set *Delays* to `[]`. In future releases, this argument will allow you to specify model delays.

Gridded LPV Models

A common way of representing LPV models is as an interpolated array of linear state-space models. A certain number of points in the scheduling space are selected. An LTI system is assigned to each point, representing the dynamics in the local vicinity of that point. You obtain the dynamics at scheduling locations in between the grid points by interpolating LTI systems at neighboring points. For meaningful interpolations of system matrices, all the local models must use the same state basis.

This form is called the *grid-based LPV representation*. For more information, see “Gridded Models and Choice of Sampling Grid” on page 1-29.

Sampling and Interpolation

Use `sample` to sample the LTV or LPV dynamics over a grid of t or (t,p) values (k or (k,p) in discrete time). This gives the local linear (affine) dynamics for a given time or parameter value. The result consists of an array of `ss` objects and a `struct` array of offsets.

Conversely, `ssInterpolant` takes an `ss` array and a `struct` array of offsets and creates an LTV or LPV model that interpolates these values between grid points (the grid is defined by the `SamplingGrid` property of the `ss` array). The resulting model is called a gridded LTV or LPV model.

Model Interconnection

All standard signal-based connections listed under “Model Interconnection” are supported for LTV and LPV models. These operations automatically manage offsets and do not involve any approximation. Interconnecting models using signals allow you to construct models for control systems. You can conceptualize your control system as a block diagram containing multiple interconnected components, such as a plant and a controller connected in a feedback configuration. For instance, you can build and LPV model of a plant, design a gain-scheduled controller on a t or (t,p) grid, and simulate the closed-loop control system. See “Analysis of Gain-Scheduled PI Controller” on page 1-113 and “Control Design for Spinning Disks” on page 1-85 for examples.

Additionally:

- Combining two `ltvss` models yields an `ltvss` model.
- Combining two `lpvss` models yields an `lpvss` model, depending on the union of parameters.
- Combining an `ltvss` model and an `lpvss` model yields an `lpvss` model.
- Combining an LTI model with an `ltvss` or `lpvss` model yields an `ltvss` or `lpvss` model, respectively.

Before you combine models with shared parameters that each use a different test value `p0` for validation, you must first reconcile these values using `setTestValue`. For `p0` = 0, the interconnection functions ignore the test values.

If each model has different parameters, the combined model depends on the union of the parameters.

Continuous and Discrete Conversions

You can convert between continuous-time and discrete-time and resample `ltvss` and `lpvss` models using `c2d`, `d2c`, and `d2d`.

- For analytic `ltvss` and `lpvss` models, the software only supports conversion using the `'tustin'` method.
- For gridded LTV and LPV models (see `ssInterpolant`), the functions convert or resample the LTI model at each grid point and interpolate the resulting data. The software only supports conversion for gridded models using `'zoh'`, `'impulse'`, and `'tustin'` methods.

Time Response Simulation

The time-response functions such as `step`, `impulse`, `lsim`, and `initial` support LTV and LPV model simulation along with LTI models for easy comparison. You must specify an equisampled time vector to set the integration step size of the fixed-step LTV and LPV solvers. There are two key differences with LTI simulation:

- For LPV models, you must specify an additional input/output `p` depending on the syntax. As input, you can specify the parameter trajectory $p(t)$ as a vector or matrix or implicitly as a function $p = h(t,x,u)$. As output, p is the actual parameter trajectory.

For examples that compare the two approaches, see “LPV Model of Magnetic Levitation System” on page 1-127 and “Hidden Couplings in Gain-Scheduled Control” on page 1-118.

- Input offsets and initial conditions matter for LTV and LPV model. You can use the `RespConfig` object to manage these settings. The software now supports the input offsets and initial states for LTI models for uniformity. This allows for side-by-side comparisons with LTV and LPV models. You must specify correct model offsets along with proper initial conditions for accurate time-response simulation.

Gain-Scheduled Controller Design

You can use functions such as `pidtune` and `systune` to tune gain-scheduled controllers. A gain-scheduled controller is an LPV model parameterized by the scheduling variables. For examples that implement a gain-scheduled controller as an analytic LPV model, see “Gain-Scheduled LQG Controller” on page 1-106 and “Analysis of Gain-Scheduled PI Controller” on page 1-113. For examples that implement a gain-scheduled controller as a gridded LPV model, see “LPV Model of Magnetic Levitation Model from Batch Linearization Results” on page 1-133, “Control Design for Wind Turbine” on page 1-140, and “Approximate Nonlinear Aircraft Pitch Dynamics Using LPV Model” on page 1-68.

LPV System Block

You can use the LPV System block to represent and simulate gridded LPV models in Simulink. You can use this block to represent an array of state-space models and its associated offsets as an LPV model. For an example, see “Using LTI Arrays for Simulating Multi-Mode Dynamics” on page 2-89.

You can also use this block to implement a controller designed on a grid of trim points, and then test the closed-loop response of the controller with the nonlinear Simulink model. For examples, see “LPV Model of Magnetic Levitation Model from Batch Linearization Results” on page 1-133 and “Approximate Nonlinear Aircraft Pitch Dynamics Using LPV Model” on page 1-68.

Other Supported Functionality

Additionally, the following functionality is currently supported.

- Use `setTestValue` and `getTestValue` to manage the test values used to validate the data function.
- Use `sminreal` to eliminate structurally nonminimal states. This is only possible in gridded models and when it is done in a uniform way across models in the grid.
- Use `xperm` to reorder the states.
- Use `order` to find the number of states.

Applications of Linear Parameter-Varying Models

Modeling Multimode Dynamics

You can use LPV models to represent systems that exhibit multiple modes (regimes) of operation. Examples of such systems include colliding bodies, systems controlled by operator switches, and approximations of systems affected by dry friction and hysteresis effects. For an example, see “LPV Model of Bouncing Ball” on page 1-103.

Proxy Modeling for Faster Simulations

This approach is useful for generating surrogate models that you can use in place of the original system, which enable faster simulations as well as hardware-in-loop (HIL) simulations and reduce the memory footprint of target hardware code. You can also use surrogate models of this type for designing gain-scheduled controllers and for initializing parameter estimation tasks in Simulink. For an example of approximating a general nonlinear system behavior by an LPV model, see “Approximate Nonlinear Aircraft Pitch Dynamics Using LPV Model” on page 1-68.

LPV models can help speed up the simulation of physical component based systems, such as those built using Simscape™ Multibody™ and Simscape Electrical™ Power Systems software. For an example of this approach, see “LPV Approximation of Boost Converter Model” on page 1-78.

See Also

`lpvss` | `ltvss` | `sample` | `ssInterpolant` | LPV System

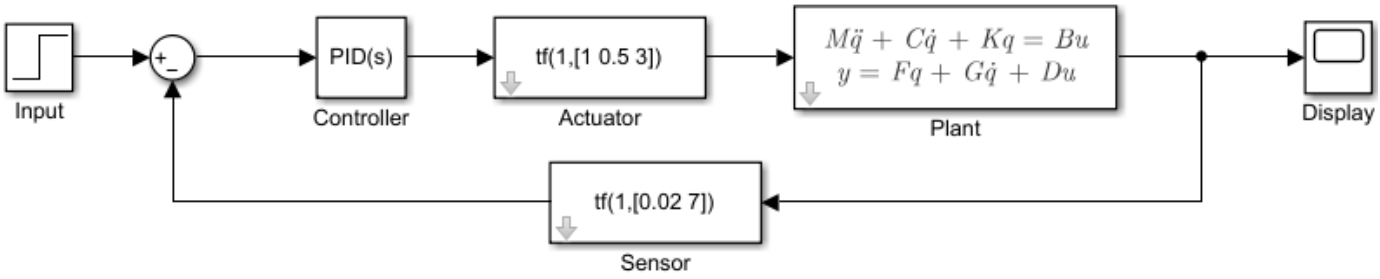
Related Examples

- “LTV and LPV Modeling” on page 1-26
- “LPV Model of Bouncing Ball” on page 1-103
- “LPV Model of Magnetic Levitation System” on page 1-127
- “LPV Approximation of Boost Converter Model” on page 1-78
- “Approximate Nonlinear Aircraft Pitch Dynamics Using LPV Model” on page 1-68
- “Control Design for Spinning Disks” on page 1-85
- “LTV Model of Two-Link Robot” on page 1-93

Linearize Simulink Model to a Sparse Second-Order Model Object

This example shows how to linearize a Simulink® Model to obtain a sparse second-order model object (`mechss`). The Simulink model `sparseSimulinkModel.slx` contains a plant modeled using a Sparse Second Order block connected in a feedback loop with a PID controller and an actuator and sensor. The sensor and actuator are modeled using transfer functions. You need a Simulink Control Design™ license to perform linearization.

For more information on sparse models, see “Sparse Model Basics” on page 1-18.



Load the model data contained in `linData.mat` and open the Simulink model.

```
load('linData.mat')
mdl = 'sparseSimulinkModel';
open_system(mdl);
```

Next, construct the linearization I/O settings using `linio` (Simulink Control Design) and linearize the Simulink model.

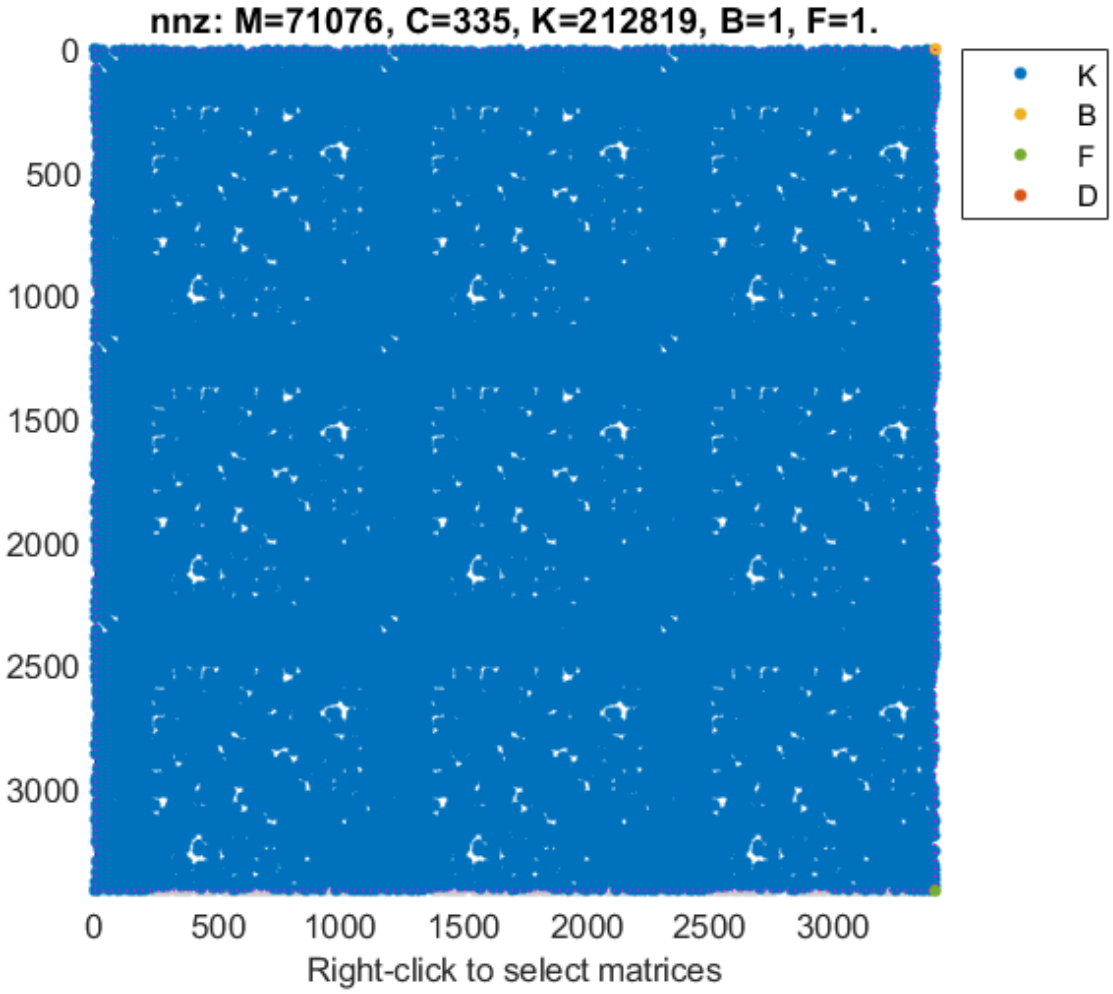
```
sys_io(1) = linio('sparseSimulinkModel/Controller',1,'input');
sys_io(2) = linio('sparseSimulinkModel/Plant',1,'output');
sys = linearize(mdl,sys_io)
```

```
Sparse continuous-time second-order model with 1 outputs, 1 inputs, and 3415 degrees of freedom.

Use "spy" and "showStateInfo" to inspect model structure.
Type "properties('mechss')" for a list of model properties.
Type "help mechssOptions" for available solver options for this model.
```

The resultant linearized model `sys` is a `mechss` model object with 3415 degrees of freedom, 1 input and 1 output.

You can use `spy` to visualize the sparsity of the sparse model. Right-click on the plot to switch to select the matrices to be displayed.

```
spy(sys)
```



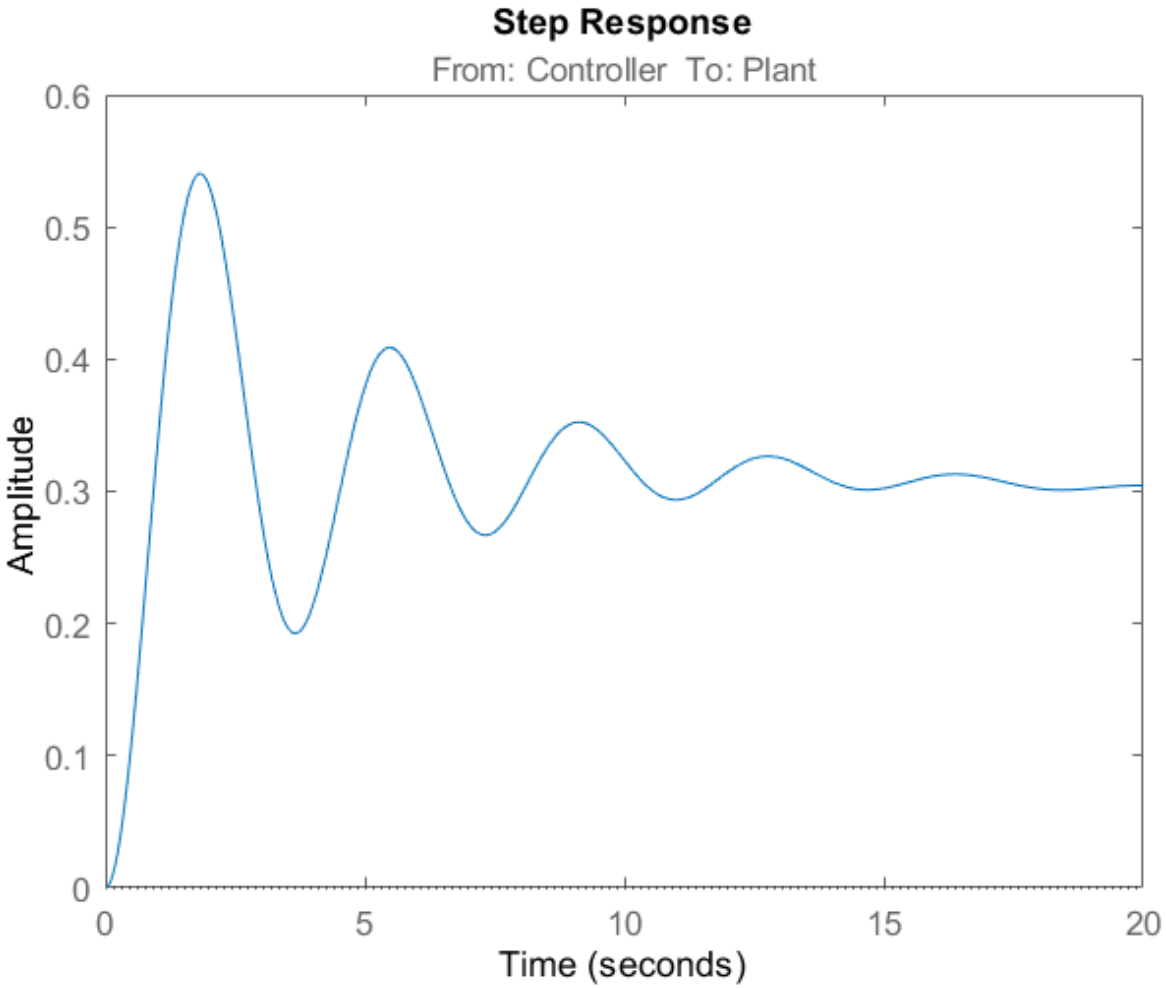
Use `showStateInfo` to view the partition information of the sparse second-order model `sys`.

```
showStateInfo(sys)
```

```
The state groups are:

    Type                 Name               Size
  ------------------------------------------------
  Component                                    5
  Component    sparseSimulinkModel/Plant    3408
  Signal                                       2
```

Examine the step response of the sparse second-order model. You must specify the final time or the time vector for sparse models.

```
t = 0:0.01:20;
step(sys,t)
```



See Also

`linearize` | `linio` | `mechss` | `showStateInfo` | `xsort` | `spy` | Sparse Second Order

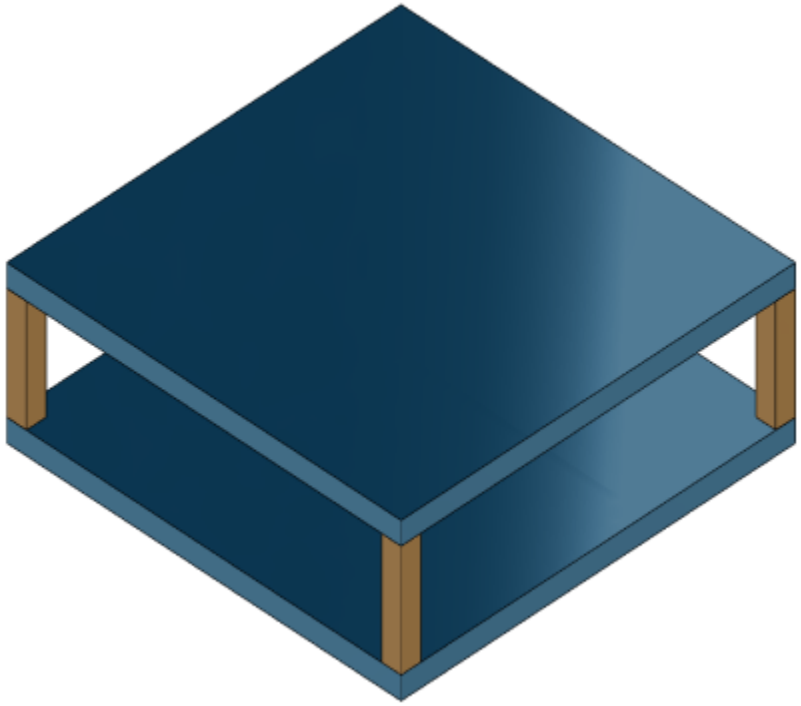
More About

- “Sparse Model Basics” on page 1-18

Rigid Assembly of Model Components

This example shows how to specify rigid physical coupling between the components of a structural model. Consider a structure that consists of two square plates connected with pillars at each vertex as depicted in the figure below. The lower plate is connected rigidly to the ground while the pillars are connected rigidly to each vertex of the square plates.

For more information on sparse models, see “Sparse Model Basics” on page 1-18.



`platePillarModel.mat` contains the sparse matrices for pillar and plate model. Load the finite element model matrices contained in `platePillarModel.mat` and create the sparse second-order state-space model representing the above system.

```
load('platePillarModel.mat')
sys = ...
   mechss(M1,[],K1,B1,F1,'Name','Plate1') + ...
   mechss(M2,[],K2,B2,F2,'Name','Plate2') + ...
   mechss(Mp,[],Kp,Bp,Fp,'Name','Pillar3') + ...
   mechss(Mp,[],Kp,Bp,Fp,'Name','Pillar4') + ...
   mechss(Mp,[],Kp,Bp,Fp,'Name','Pillar5') + ...
   mechss(Mp,[],Kp,Bp,Fp,'Name','Pillar6')
```

```
Sparse continuous-time second-order model with 1 outputs, 1 inputs, and 5820 degrees of freedom.

Use "spy" and "showStateInfo" to inspect model structure.
Type "help mechssOptions" for available solver options for this model.
```

The resultant model `sys` has 5820 degrees of freedom, one input and one output.

Use `showStateInfo` to examine the components of the `mechss` model object.

```
showStateInfo(sys)
```

```
The state groups are:

    Type        Name      Size
  ---------------------------
  Component    Plate1     2646
  Component    Plate2     2646
  Component    Pillar3     132
  Component    Pillar4     132
  Component    Pillar5     132
  Component    Pillar6     132
```

The named components are listed in the command window with their respective sizes.

Now, load the interfaced DOF index data from `dofData.mat` and use `interface` to create the physical connections between the two plates and the four pillars. `dofs` is a `6x7` cell array where the first two rows contain DOF index data for the first and second plate while the remaining four rows contain index data for the four pillars.

```
load('dofData.mat','dofs')
```

Now, specify rigid couplings between the plates and the pillars.

```
for i=3:6
   sys = interface(sys,"Plate1",dofs{1,i},"Pillar"+i,dofs{i,1});
   sys = interface(sys,"Plate2",dofs{2,i},"Pillar"+i,dofs{i,2});
end
```

Specify rigid connection between the bottom plate and the ground.

```
sysCon = interface(sys,"Plate2",dofs{2,7})
```

```
Sparse continuous-time second-order model with 1 outputs, 1 inputs, and 5922 degrees of freedom.

Use "spy" and "showStateInfo" to inspect model structure.
Type "help mechssOptions" for available solver options for this model.
```

Notice that the model now contains 5922 degrees of freedom. The extra DOFs are a result of the specific rigid interfaces.

`interface` uses 'dual assembly' formulation to connect the components. In the concept of dual assembly, the global set of degrees of freedom (DoFs) q is retained and the physical coupling is expressed as consistency and equilibrium constraints at the interface. For rigid connections, these constraints are of the form:

$$Bq = 0, \qquad g = -B^T\lambda$$

where g is the vector of internal forces at the interface, and the matrix B is permutable to $[I \ -I]$. For a pair of matching DOFs with indices i_i, i_2 where i_i selects a DOF in the first component while i_2 selects the matching DOF in the second component, $Bq = 0$ enforces consistency of displacements:

$$q(i_1) = q(i_2)$$

while $g = -B^T\lambda$ enforces equilibrium of the internal forces g at the interface:

$g(i_1) + g(i_2) = 0$.

Combining these constrains with the uncoupled equations $M\ddot{q} + C\dot{q} + Kq = f + g$ leads to the following dual assembly model for the coupled system:

$$\begin{bmatrix} M & 0 \\ 0 & 0 \end{bmatrix}\begin{bmatrix} \ddot{q} \\ \lambda \end{bmatrix} + \begin{bmatrix} C & 0 \\ 0 & 0 \end{bmatrix}\begin{bmatrix} \dot{q} \\ \lambda \end{bmatrix} + \begin{bmatrix} K & B^T \\ B & 0 \end{bmatrix}\begin{bmatrix} q \\ \lambda \end{bmatrix} = \begin{bmatrix} f \\ 0 \end{bmatrix}$$

For more information, see `interface`.

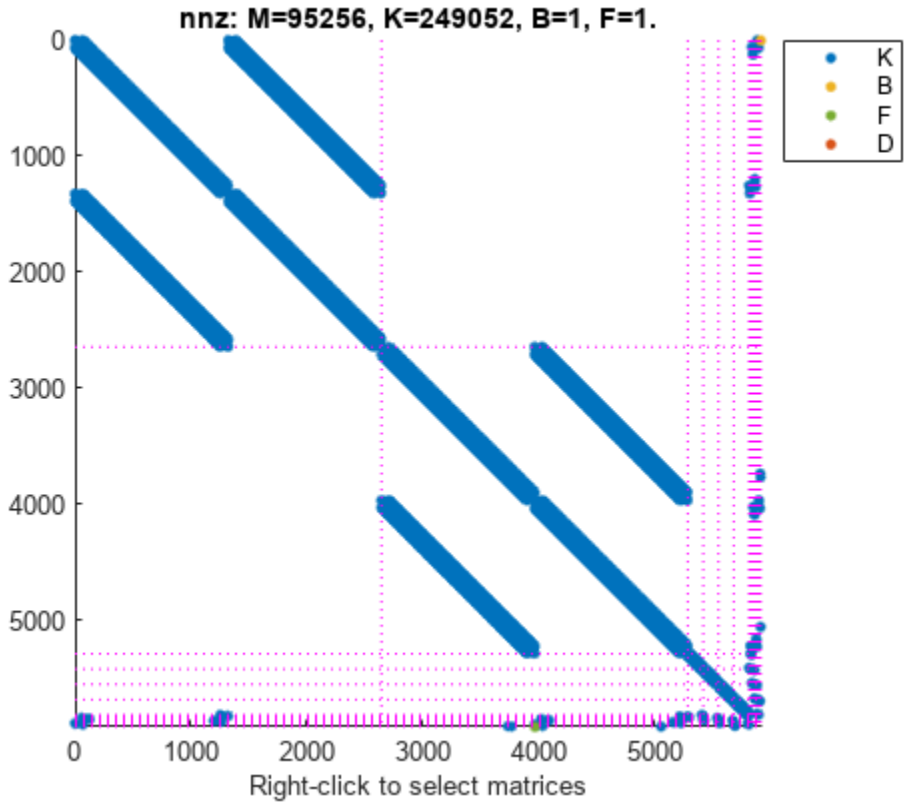
Use `showStateInfo` to confirm the physical connections.

```
showStateInfo(sysCon)
```

```
The state groups are:

    Type            Name          Size
  ------------------------------------
  Component        Plate1        2646
  Component        Plate2        2646
  Component       Pillar3         132
  Component       Pillar4         132
  Component       Pillar5         132
  Component       Pillar6         132
  Interface    Plate1-Pillar3      12
  Interface    Plate2-Pillar3      12
  Interface    Plate1-Pillar4      12
  Interface    Plate2-Pillar4      12
  Interface    Plate1-Pillar5      12
  Interface    Plate2-Pillar5      12
  Interface    Plate1-Pillar6      12
  Interface    Plate2-Pillar6      12
  Interface    Plate2-Ground        6
```

You can use `spy` to visualize the sparse matrices in the final model. Choose between the matrices to be displayed using the display menu that can be accessed by right-clicking on the plot.

```
spy(sysCon)
```



Acknowledgements

The data set for this example was provided by Victor Dolk from ASML.

See Also

`interface` | `xsort` | `showStateInfo` | `spy` | `mechss`

More About

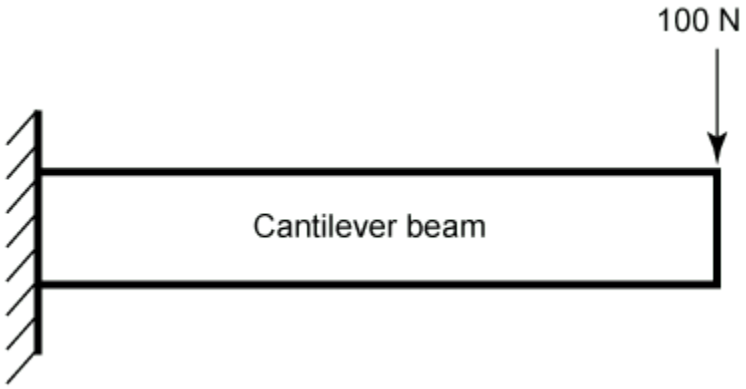
- “Sparse Model Basics” on page 1-18

Linear Analysis of Cantilever Beam

This example shows how to obtain a structural model of a beam and calculate the time and frequency response when it is subjected to an impulse force. For this example, consider a steel cantilever beam subject to a point load at the tip. Building the structural model requires Partial Differential Equation Toolbox™.

The steel beam is deformed by applying an external load of 100 N at the tip of the beam and then released. This example does not use any additional loading, so the displacement of the tip oscillates with an amplitude equal to the initial displacement from the applied force. This example follows the three-step workflow:

1. Construct a structural model of the beam.
2. Linearize the structural model to obtain a sparse linear model of the beam.
3. Analyze the time and frequency response of the linearized model.



Structural Model of Beam

Using Partial Differential Equation Toolbox, build a structural model and compute the impulse response. For an example of modeling a cantilever beam, see “Dynamics of Damped Cantilever Beam” (Partial Differential Equation Toolbox).

First construct the beam and specify the Young's modulus, Poisson's ratio, and mass density of steel. Specify the tip of the beam using the `addVertex` function.

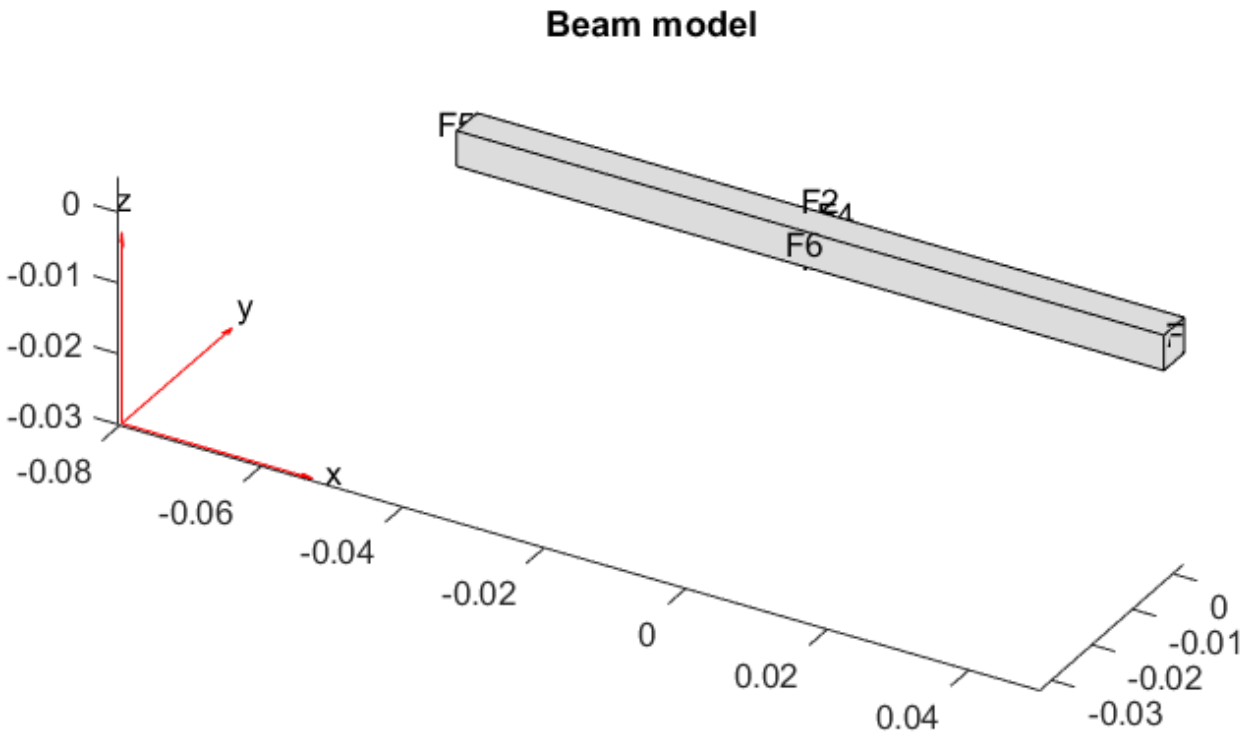
```
gm = multicuboid(0.1,0.005,0.005);
E = 210E9;
nu = 0.3;
rho = 7800;
TipVertex = addVertex(gm,'Coordinates',[0.05,0,0.005]);
```

Use `createpde` (Partial Differential Equation Toolbox) to construct the structural model and generate the mesh using the `generateMesh` (Partial Differential Equation Toolbox) command.

```
sModel = createpde('structural','transient-solid');
sModel.Geometry = gm;
msh = generateMesh(sModel);
```

Visualize the beam geometry using `pdegplot` (Partial Differential Equation Toolbox).

```
figure
pdegplot(sModel,'FaceLabels','on');
title('Beam model')
```



Assign structural properties for the steel beam with the `structuralProperties` (Partial Differential Equation Toolbox) command and fix one end using `structuralBC` (Partial Differential Equation Toolbox).

```
structuralProperties(sModel,'YoungsModulus',E,'PoissonsRatio',nu,'MassDensity',rho);
structuralBC(sModel,'Face',5,'Constraint','fixed');
```

You can calculate the vibration modes of the beam by solving the modal analysis model in a specified frequency range using `solve` (Partial Differential Equation Toolbox). For this beam, the first vibration mode is at 2639 rad/s as confirmed by the Bode plot in the Linear Analysis on page 1-49 section of this example.

```
firstNF = 2639;
Tfundamental = 2*pi/firstNF;
```

To model an impulse (knock) on the tip of the beam, apply force for 2% of the fundamental period of oscillation (impulse) using `structuralBoundaryLoad` (Partial Differential Equation Toolbox). Specify the label `force` to use this force as a linearization input.

```
Te = 0.02*Tfundamental;
structuralBoundaryLoad(sModel,'Vertex',TipVertex,...
   'Force',[0;0;-100],'EndTime',Te,'Label','force');
```

Set initial conditions for the beam model using `structuralIC` (Partial Differential Equation Toolbox).

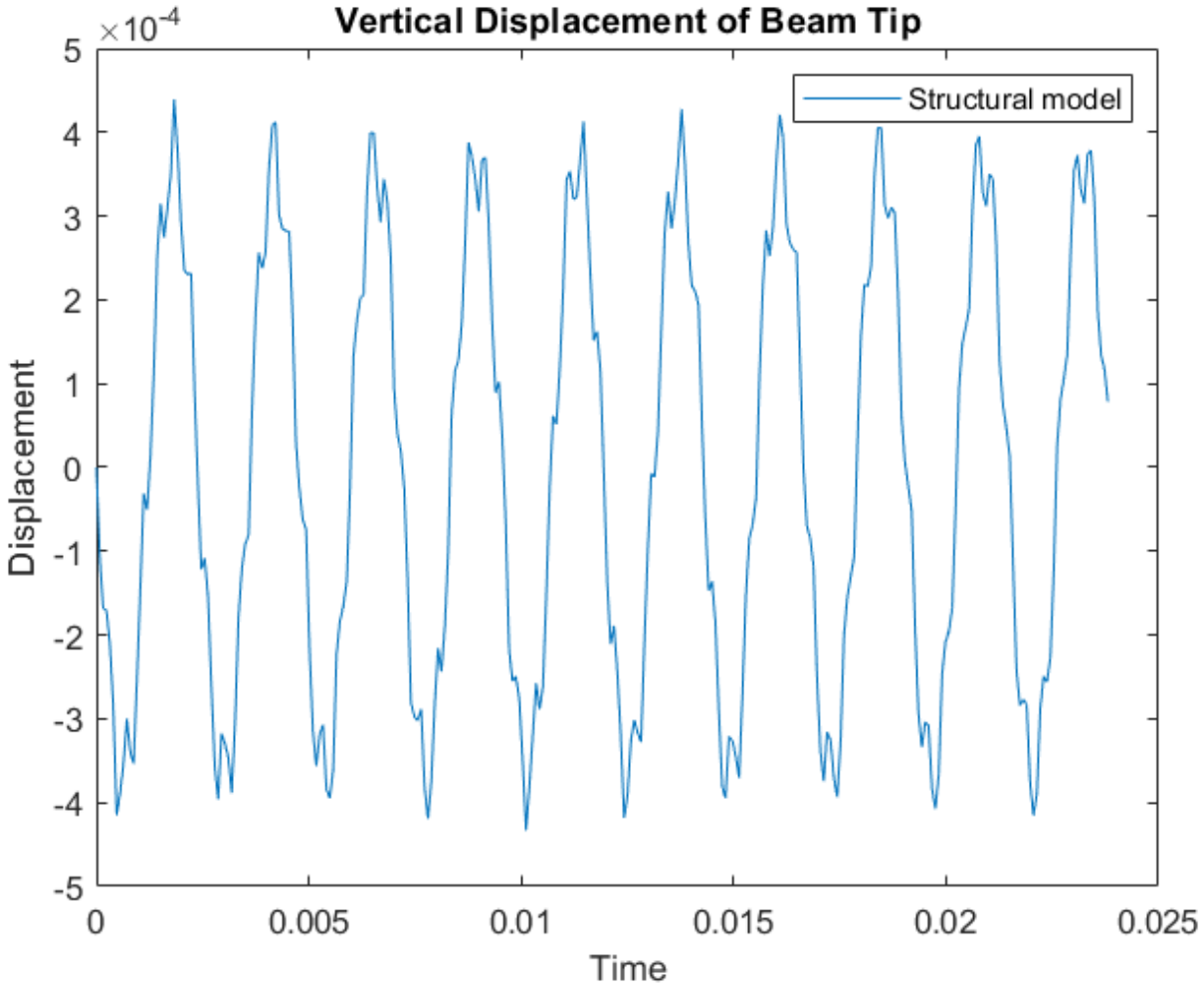
```
structuralIC(sModel,'Velocity',[0;0;0]);
```

Compute the impulse response by solving the structural beam model.

```
ncycles = 10;
tsim = linspace(0,ncycles*Tfundamental,30*ncycles);
R1 = solve(sModel,tsim);
```

Visualize the oscillations at the tip of the beam.

```
figure
plot(tsim,R1.Displacement.uz(TipVertex,:))
title('Vertical Displacement of Beam Tip')
legend('Structural model')
xlabel('Time')
ylabel('Displacement')
```



Structural Model Linearization

For this beam model, you want to obtain a linear model (transfer function) from the force applied at the tip to the z-displacement of the tip.

To do so, first specify the inputs and outputs of the linearized model in terms of the structural model. Here, the input is the force specified with `structuralBoundaryLoad` (Partial Differential Equation Toolbox) and the output is the z degree of freedom of the tip vertex.

```
linearizeInput(sModel,'force');
linearizeOutput(sModel,'Vertex',TipVertex,'Component','z');
```

Then, use the `linearize` (Partial Differential Equation Toolbox) command to extract the `mechss` model.

```
sys = linearize(sModel)
```

```
Sparse continuous-time second-order model with 1 outputs, 3 inputs, and 3486 degrees of freedom.

Use "spy" and "showStateInfo" to inspect model structure.
```

```
Type "properties('mechss')" for a list of model properties.
Type "help mechssOptions" for available solver options for this model.
```

The linearized model has three inputs corresponding to the x, y, and z components of the force applied to the tip vertex.

```
sys.InputName
```

```
ans = 3×1 cell
    {'force_x'}
    {'force_y'}
    {'force_z'}
```

In the linearized model, select the z component of the force.

```
sys = sys(:,3)
```

```
Sparse continuous-time second-order model with 1 outputs, 1 inputs, and 3486 degrees of freedom.

Use "spy" and "showStateInfo" to inspect model structure.
Type "properties('mechss')" for a list of model properties.
Type "help mechssOptions" for available solver options for this model.
```

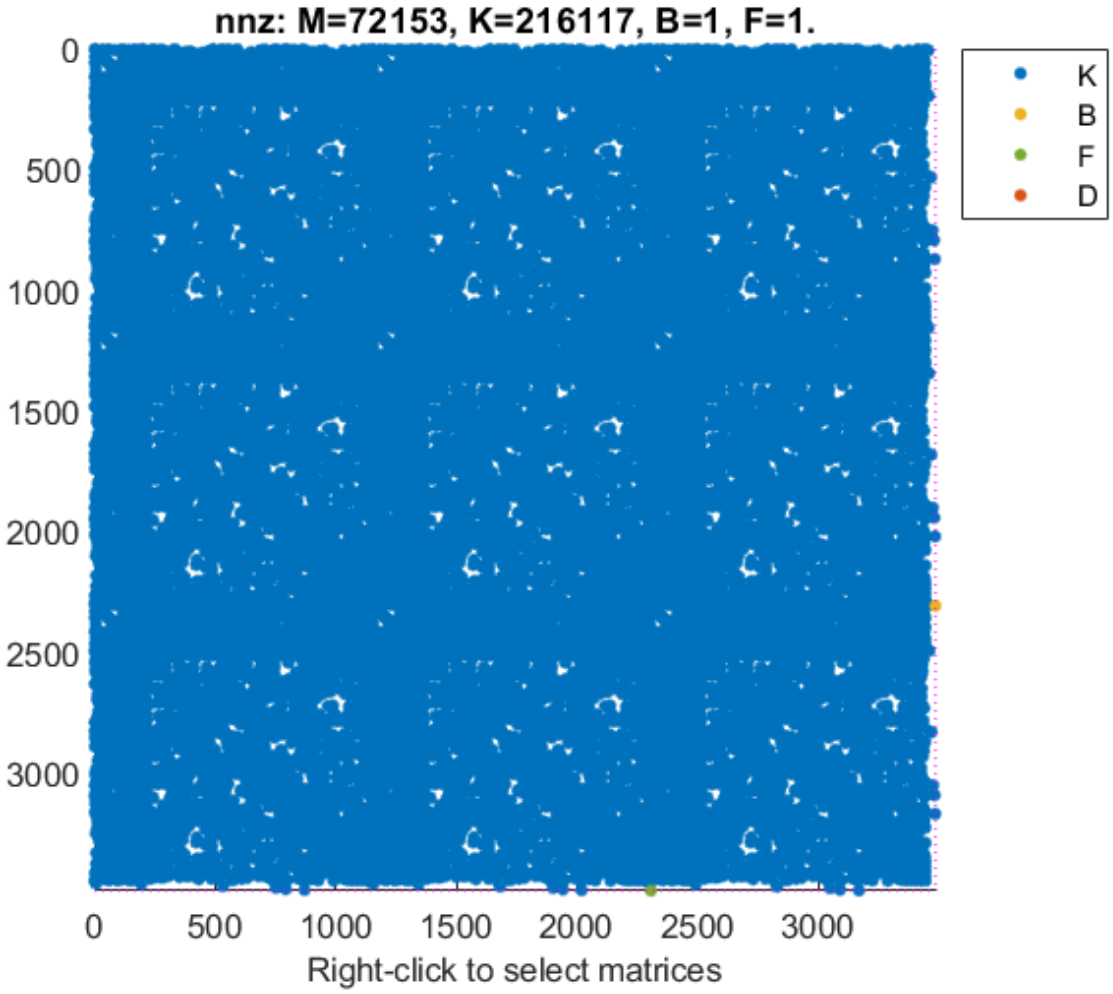
The resultant model is defined by the following set of equations:

$$M\ddot{q} + Kq = B \times force$$

$$y = Fq$$

Use `spy` to visualize the sparsity of the `mechss` model `sys`.

```
figure
spy(sys)
```



Linear Analysis

Enable parallel computing and choose `'tfbdf3'` as the differential algebraic equation (DAE) solver.

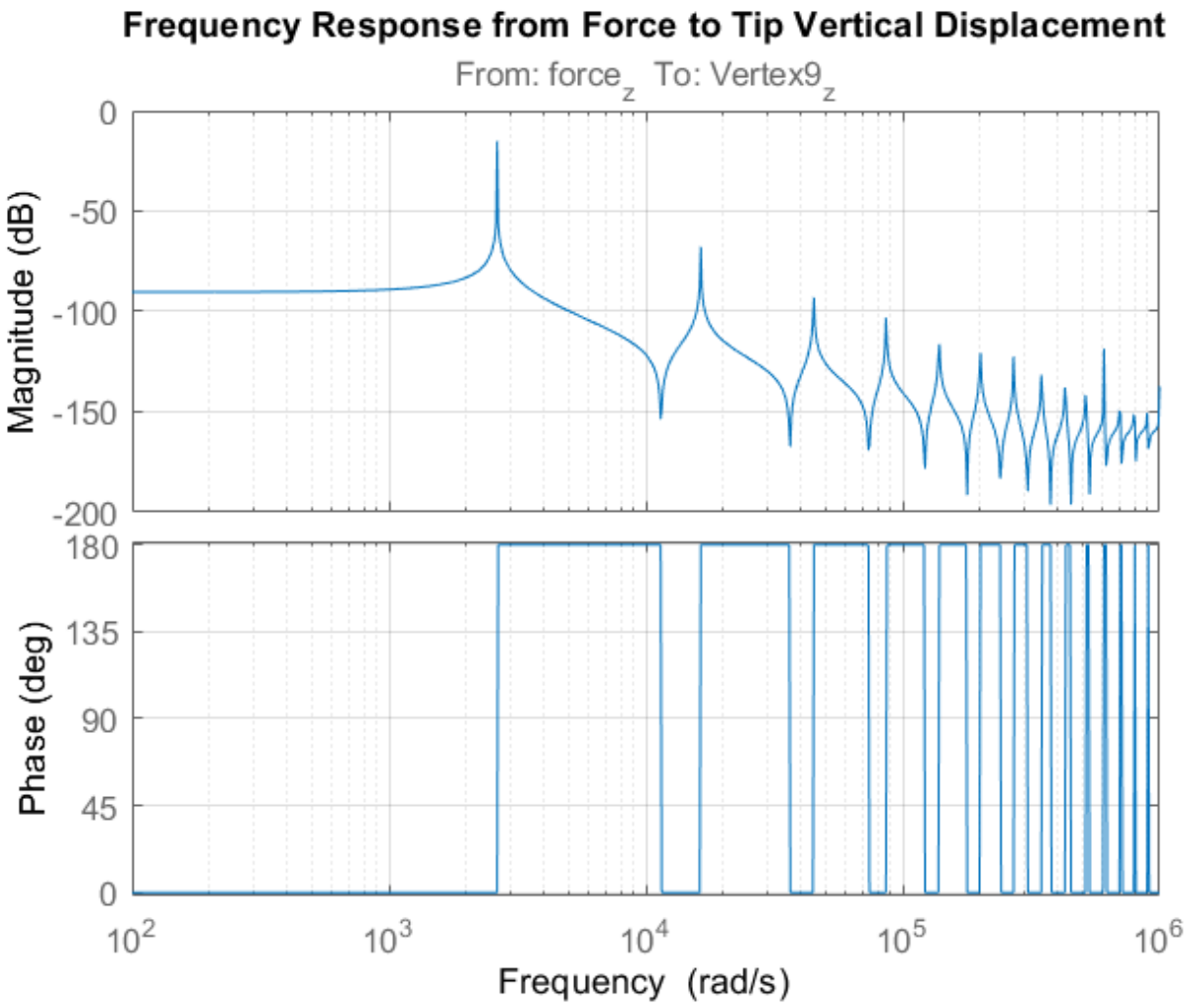
```
sys.SolverOptions.UseParallel = true;
sys.SolverOptions.DAESolver = 'trbdf3';
```

Use `bode` to compute the frequency response of the linearized model `sys`.

```
w = logspace(2,6,1000);
figure
bode(sys,w)
```

```
Starting parallel pool (parpool) using the 'local' profile ...
Connected to the parallel pool (number of workers: 6).
```

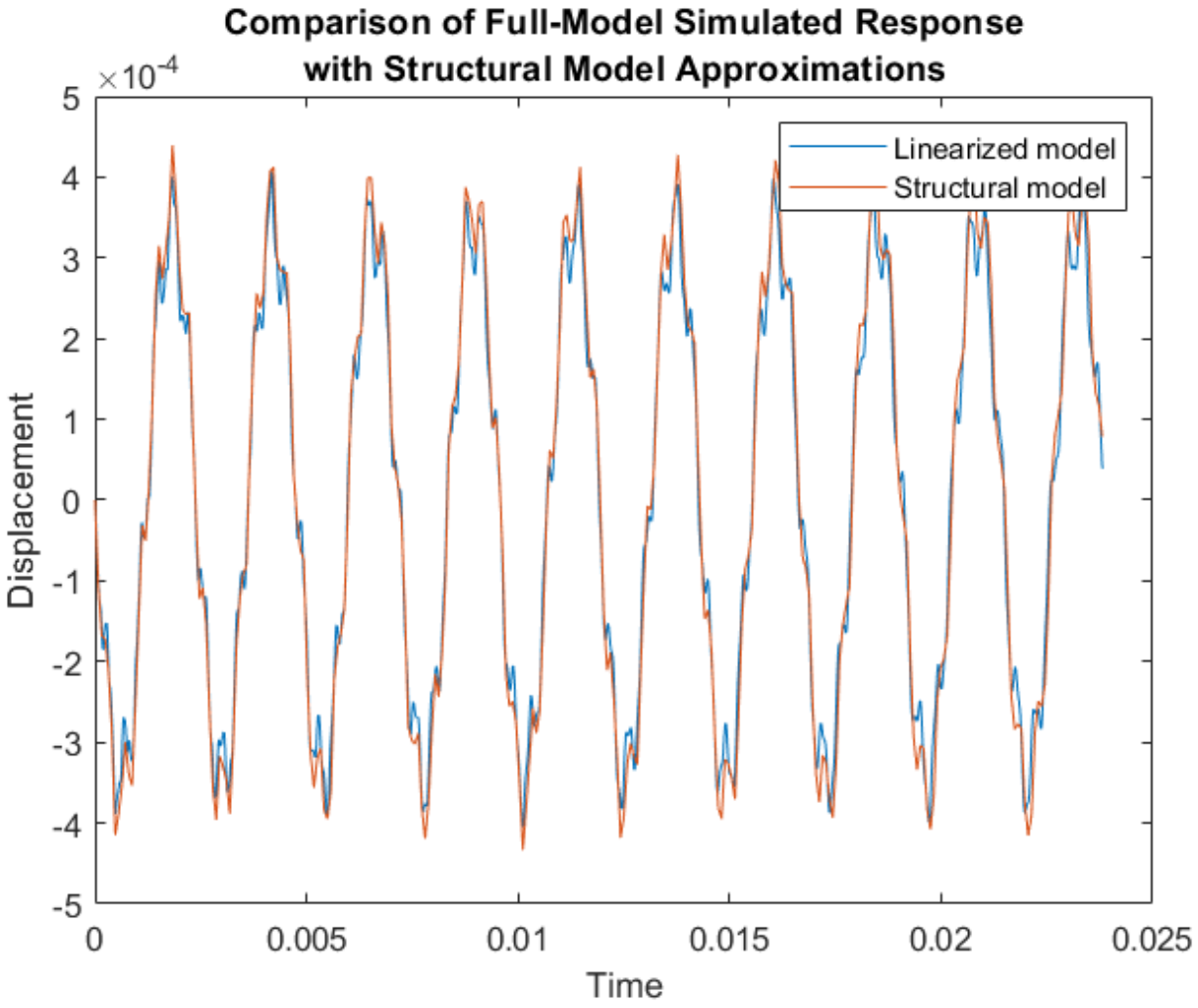
```
grid
title('Frequency Response from Force to Tip Vertical Displacement')
```



The Bode plot shows that the first vibration mode is at approximately `firstNF` = 2639 rad/s.

Next use `lsim` to compare the impulse response with the approximations obtained from the structural beam model. To limit the error due to linear interpolation of the force between samples, use a step size of `Te/5`. Recall that the force is applied at the beam tip for the short time interval [0 `Te`].

```
h = Te/5;
t = 0:h:ncycles*Tfundamental;
u = zeros(size(t));
u(t<=Te) = -100;

y = lsim(sys,u,t);
figure
plot(t,y,tsim,R1.Displacement.uz(TipVertex,:))
title({'Comparison of Full-Model Simulated Response';...
        'with Structural Model Approximations'})
legend('Linearized model','Structural model')
xlabel('Time')
ylabel('Displacement')
```



The linear response from `lsim` closely matches the transient simulation of the structural model obtained in the first step.

See Also

`showStateInfo` | `spy` | `mechss` | `solve` | `structuralBC` | `structuralIC` | `createpde` | `linearize` | `linearizeInput` | `linearizeOutput`

More About

- “Sparse Model Basics” on page 1-18

Linear Analysis of Tuning Fork

This example shows how to linearize the structural model of a tuning fork and calculate the time and frequency response when it is subjected to an impulse on one of the tines. This load results in transverse vibration of the tines and axial vibration of the end handle at same frequency. For more details about the structural model, see “Structural Dynamics of Tuning Fork” (Partial Differential Equation Toolbox).

This example requires Partial Differential Equation Toolbox™ software.

Tuning Fork Structural Model

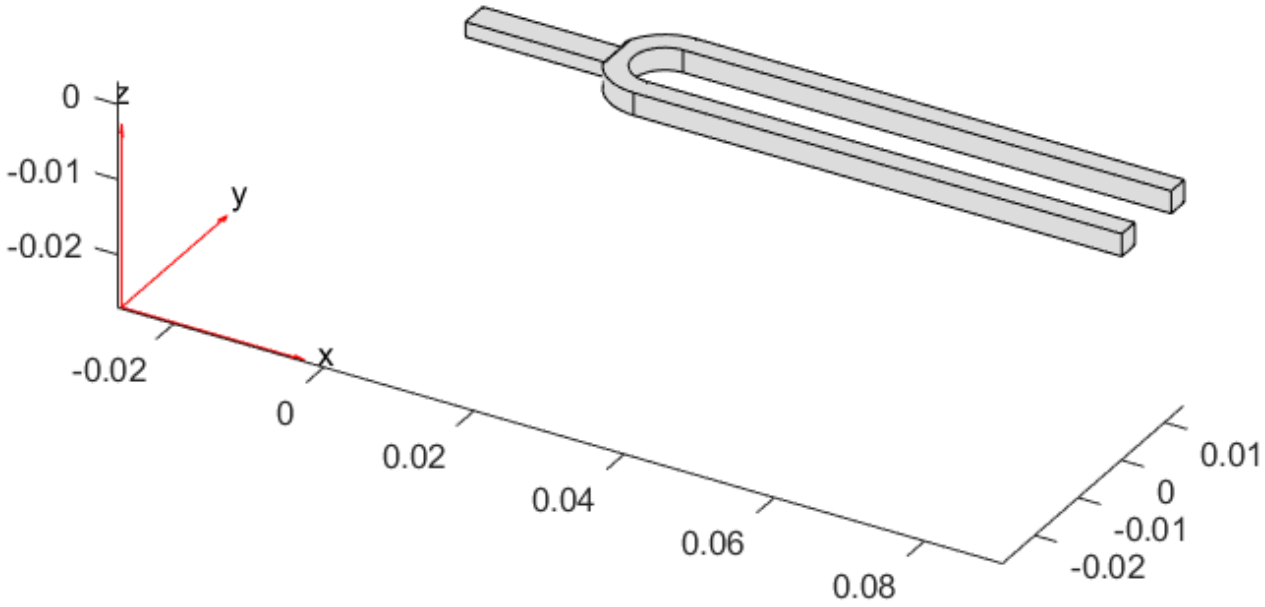
Using Partial Differential Equation Toolbox, create a structural model of the tuning fork.

Use `createpde` (Partial Differential Equation Toolbox) to construct a structural model.

```
model = createpde('structural','transient-solid');
```

Import and plot the tuning fork geometry.

```
importGeometry(model,'TuningFork.stl');
figure
pdegplot(model)
```

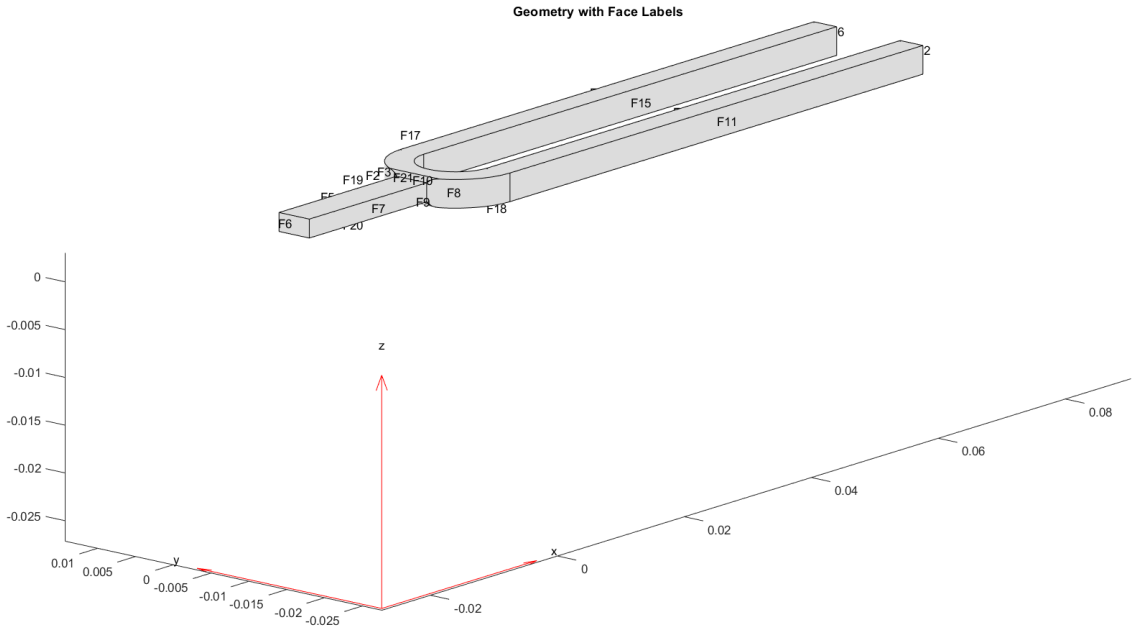


Specify the Young's modulus, Poisson's ratio, and mass density to model linear elastic material behavior. Specify all physical properties in consistent units.

```
E = 210E9;
nu = 0.3;
rho = 8000;
structuralProperties(model,'YoungsModulus',E, ...
                           'PoissonsRatio',nu, ...
                           'MassDensity',rho);
generateMesh(model,'Hmax',0.005);
```

Identify faces for applying boundary constraints and loads by plotting the geometry with face labels.

```
figure('units','normalized','outerposition',[0 0 1 1])
pdegplot(model,'FaceLabels','on')
view(-50,15)
title('Geometry with Face Labels')
```



The first mode of vibration of the tines is about 2926 rad/s. You can determine this value by modal analysis (see “Structural Dynamics of Tuning Fork” (Partial Differential Equation Toolbox)) or from the Bode plot in the Linear Analysis on page 1-56 section of this example.

Calculate the corresponding period of oscillations.

```
T = 2*pi/2926;
```

Add boundary conditions to prevent rigid body motion when applying an impulse to the tine. Typically, you hold a tuning fork by hand or mount it on a table. A simplified approximation to this boundary condition is fixing a region near the intersection of tines and the handle (faces 21 and 22).

```
structuralBC(model,'Face',[21,22],'Constraint','fixed');
```

To model an impulse load on the tine, apply a pressure load for 2% of the fundamental period of oscillation `T` using `structuralBoundaryLoad` (Partial Differential Equation Toolbox). By using this very short pressure pulse, you ensure that only the fundamental mode of the tuning fork is excited. Specify the label `Pressure` to use this load as a linearization input.

```
Te = 0.02*T;
structuralBoundaryLoad(model,'Face',11,'Pressure',5e6,'EndTime',Te,'Label','Pressure');
```

Set initial conditions for the model using `structuralIC` (Partial Differential Equation Toolbox).

```
structuralIC(model,'Displacement',[0;0;0],'Velocity',[0;0;0]);
```

Structural Model Linearization

For this tuning fork model, you want to obtain a linear model from the pressure load on the tine to the y-displacement of the tine tip (face 12) and x-displacement of the end handle (face 6).

To do so, first specify the inputs and the outputs of the linearized model in terms of the structural model. Here, the input is the pressure specified with `structuralBoundaryLoad` (Partial Differential Equation Toolbox) and the outputs are the y and x degrees of freedom of faces 12 and 6, respectively.

```
linearizeInput(model,'Pressure');
linearizeOutput(model,'Face',12,'Component','y');
linearizeOutput(model,'Face',6,'Component','x');
```

Then, use the `linearize` (Partial Differential Equation Toolbox) command to extract the `mechss` model.

```
sys = linearize(model)
```

```
Sparse continuous-time second-order model with 26 outputs, 1 inputs, and 3240 degrees of freedom

Use "spy" and "showStateInfo" to inspect model structure.
Type "properties('mechss')" for a list of model properties.
Type "help mechssOptions" for available solver options for this model.
```

In the linearized model, use `sys.OutputGroup` to locate the outputs associated with each face.

```
sys.OutputGroup
```

```
ans = struct with fields:
    Face12_y: [1 2 3 4 5 6 7 8 9 10 11 12 13]
     Face6_x: [14 15 16 17 18 19 20 21 22 23 24 25 26]
```

Select one node from each output group to create a model with one input and two outputs.

```
sys = sys([1,14],:)
```

```
Sparse continuous-time second-order model with 2 outputs, 1 inputs, and 3240 degrees of freedom.

Use "spy" and "showStateInfo" to inspect model structure.
Type "properties('mechss')" for a list of model properties.
Type "help mechssOptions" for available solver options for this model.
```

```
sys.OutputName = {'y tip','x base'};
```

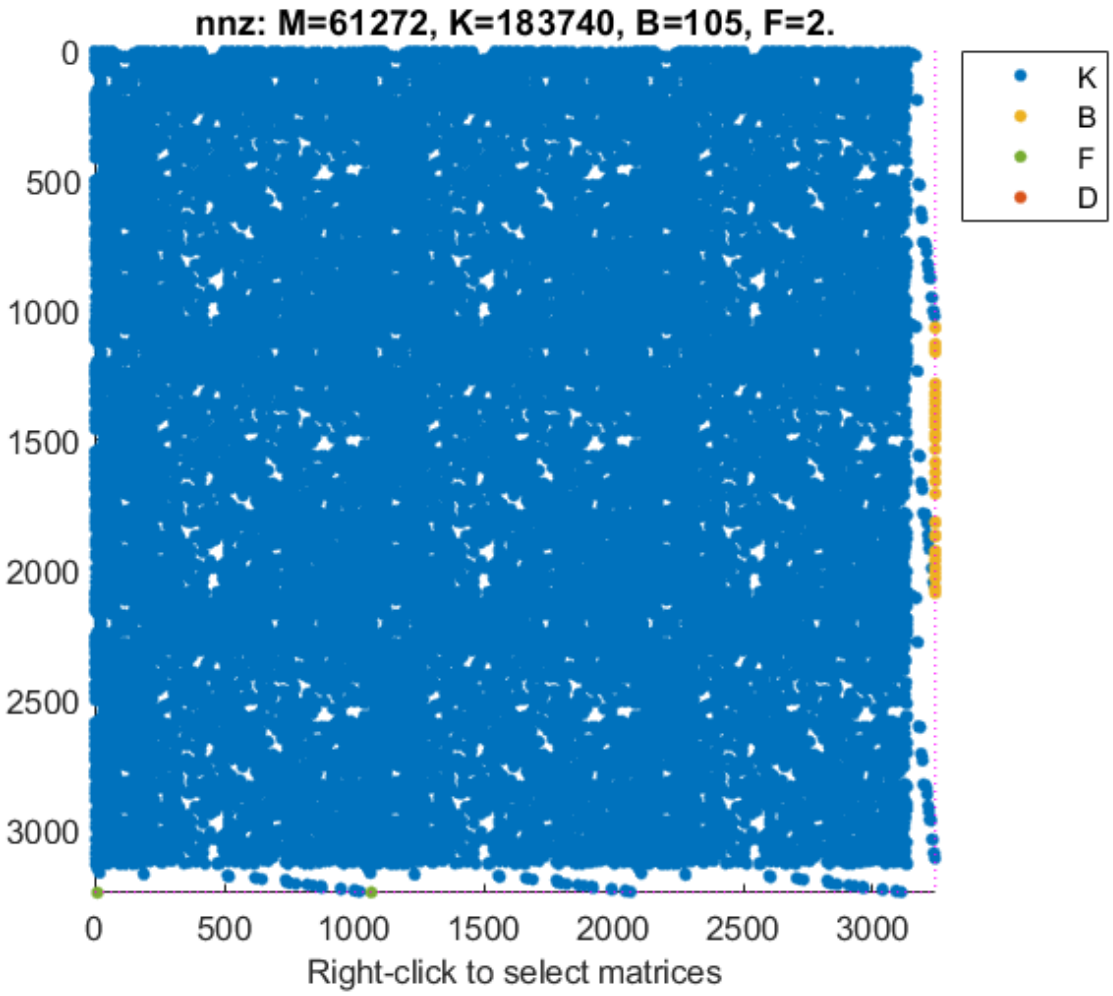
The resultant model is defined by the following set of equations:

$$M\ddot{q} + K\,q = B \times \text{Pressure}$$

$$y = F\,q$$

Use `spy` to visualize the sparsity of the `mechss` model `sys`.

```
figure
spy(sys)
```



Linear Analysis

Enable parallel computing and choose `'tfbdf3'` as the differential algebraic equation (DAE) solver.

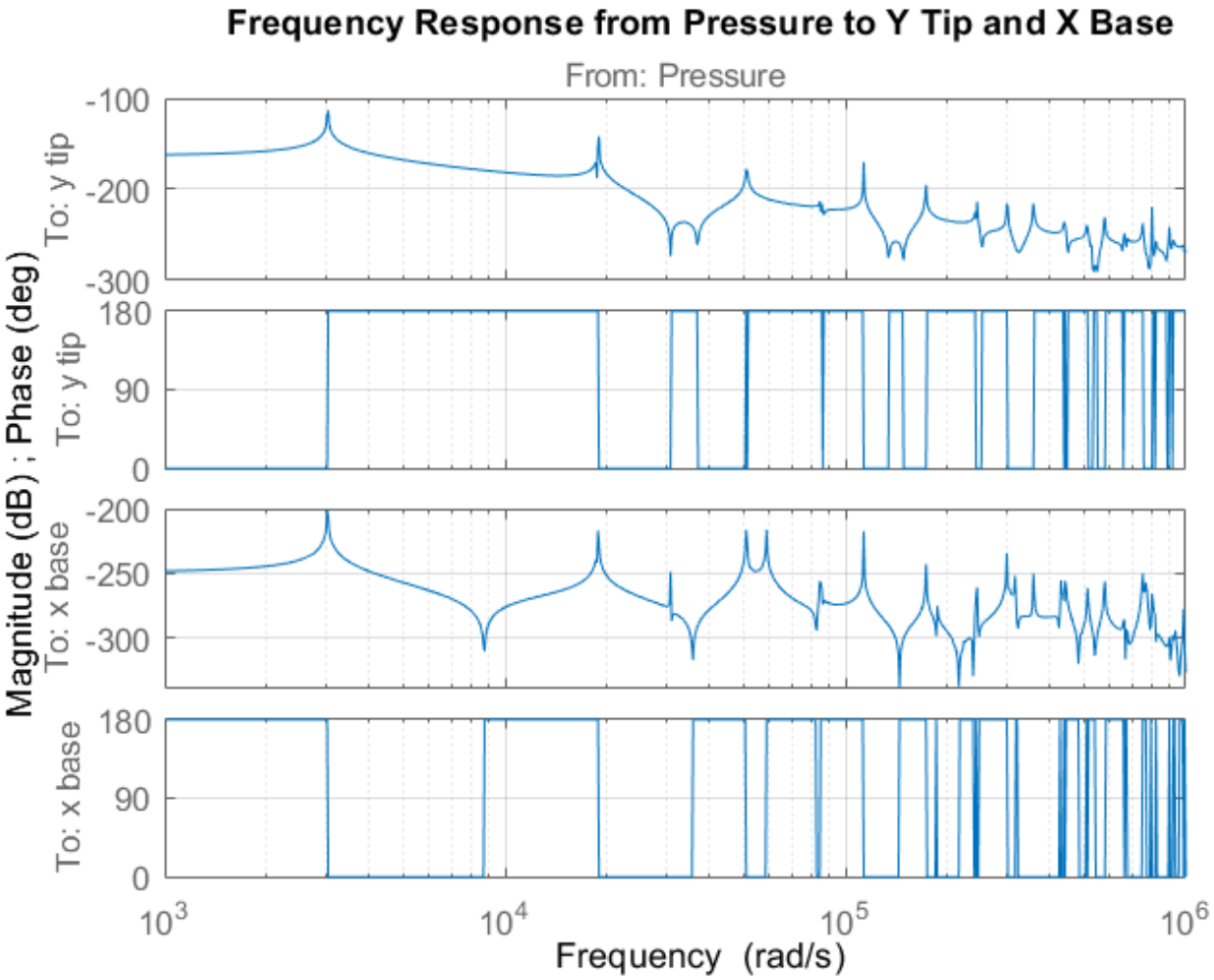
```
sys.SolverOptions.UseParallel = true;
sys.SolverOptions.DAESolver = 'trbdf3';
```

Use `bode` to compute the frequency response of this model.

```
w = logspace(3,6,1000);
figure
bode(sys,w)
```

```
Starting parallel pool (parpool) using the 'local' profile ...
Connected to the parallel pool (number of workers: 6).
```

```
grid
title('Frequency Response from Pressure to Y Tip and X Base')
```



The plot clearly shows the tine and the end handle vibrate at the same frequency. The first mode is at approximately 2926 rad/s and second resonance is at a frequency approximately six times higher.

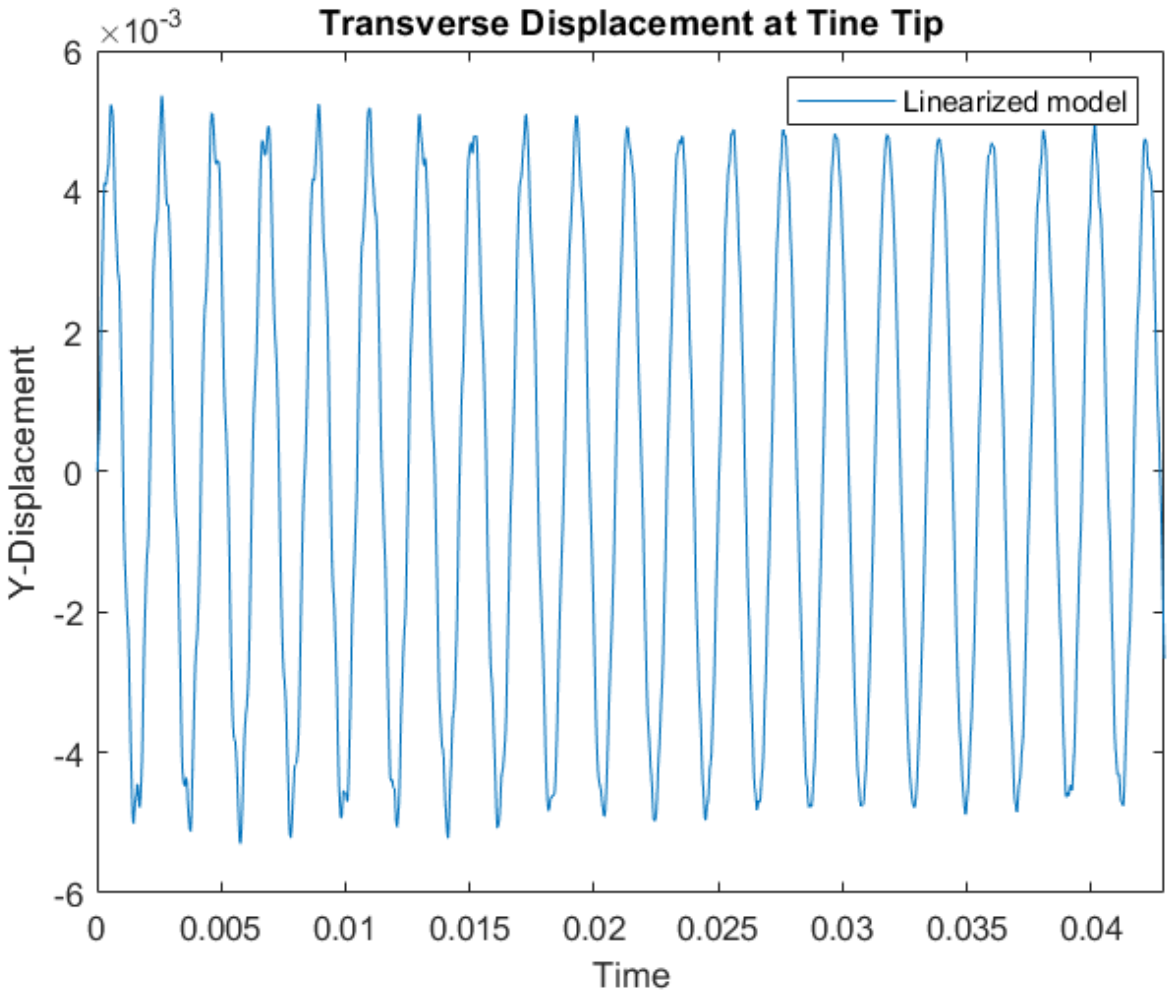
Next use `lsim` to obtain the impulse response of the linearized tuning fork model for 20 periods of the fundamental mode. To limit error due to linear interpolation of pressure between samples, use a step size of `Te/10`. The pressure is applied for the time interval [0 `Te`].

```
ncycle = 20;
Tf = ncycle*T;
h = Te/10;
t = 0:h:Tf;
u = zeros(size(t));
u(t<=Te) = 5e6;
y = lsim(sys,u,t);
```

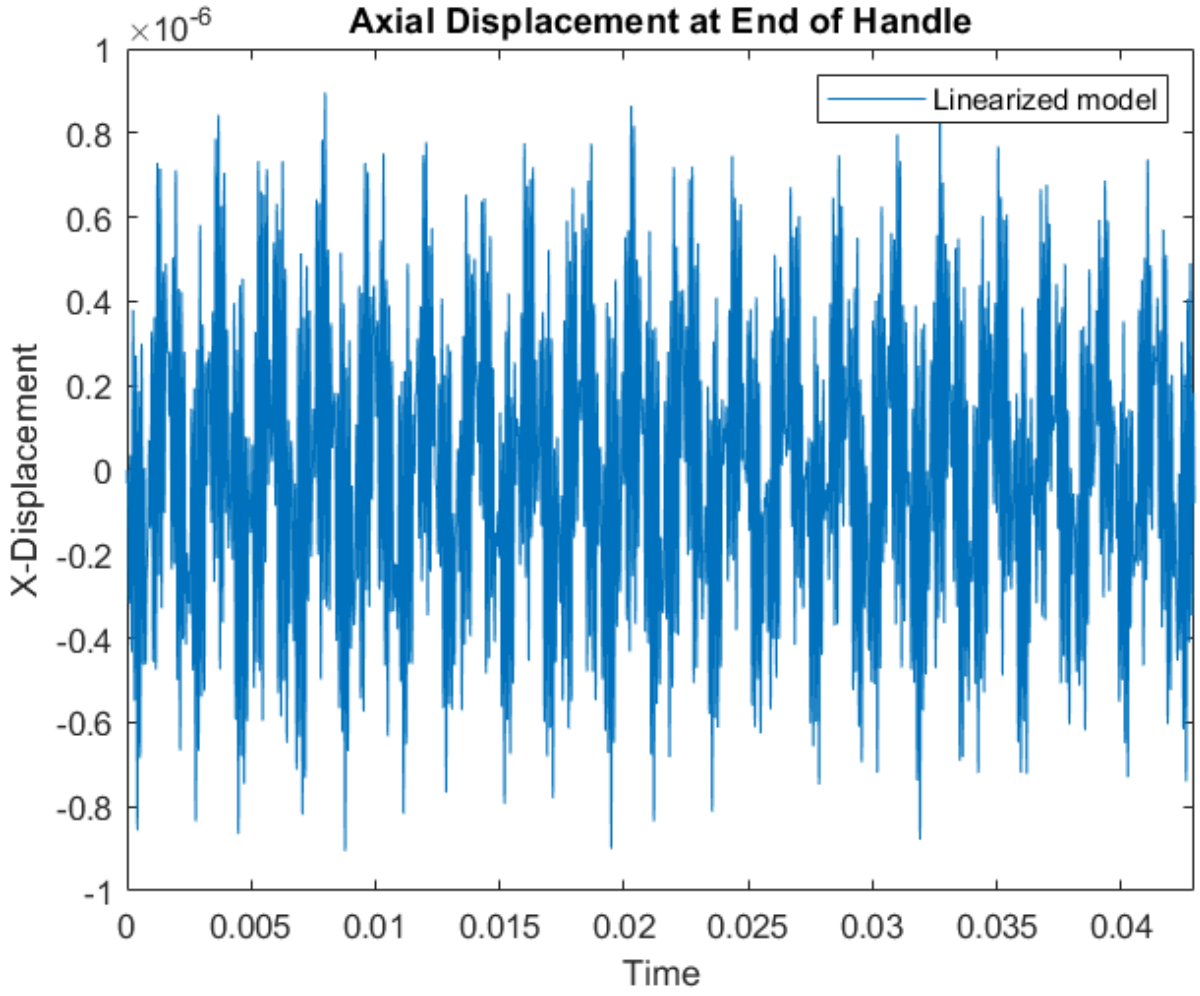
Plot the transverse displacement at the tine tip and axial displacement of the end handle.

```
figure
plot(t,y(:,1))
title('Transverse Displacement at Tine Tip')
xlim([0,Tf])
```

```
xlabel('Time')
ylabel('Y-Displacement')
legend('Linearized model')
```




```
figure
plot(t,y(:,2))
title('Axial Displacement at End of Handle')
xlim([0,Tf])
ylabel('X-Displacement')
xlabel('Time')
legend('Linearized model')
```



See Also

`showStateInfo` | `spy` | `mechss` | `solve` | `structuralBC` | `structuralIC` | `createpde` | `linearize` | `linearizeInput` | `linearizeOutput`

More About

- “Sparse Model Basics” on page 1-18

Using Model Objects

After you represent your dynamic system as a model object, you can:

- Attach additional information to the model using model attributes (properties). See “Model Attributes”.
- Manipulate the model using arithmetic and model interconnection operations. See “Model Interconnection”.
- Analyze the model response using commands such as `bode` and `step`. See “Linear Analysis”.
- Perform parameter studies using model arrays. See “Model Arrays”.
- Design compensators. You can:
  - Design compensators for systems specified as numeric LTI models. Available compensator design techniques include PID tuning, root locus analysis, pole placement, LQG optimal control, and frequency domain loop-shaping. See “PID Controller Tuning”, “Classical Control Design”, or “State-Space Control Design”.
  - Manually tune many control architectures using **Control System Designer**. See “Classical Control Design”.
  - Use tuning commands such as `systune` or **Control System Tuner** to automatically tune a control system that you represent as a `genss` model with tunable blocks. See “Multiloop, Multiobjective Tuning”.

References

[1] Dorf, R.C. and R.H. Bishop, *Modern Control Systems*, Addison-Wesley, Menlo Park, CA, 1998.

Using the Right Model Representation

This example shows some best practices for working with LTI models.

Which Representation is Best Suited for Computations?

Using the Control System Toolbox™ software, you can represent LTI systems in four different ways:

- Transfer function (TF)
- Zero-pole-gain (ZPK)
- State space (SS)
- Frequency response data (FRD)

While the TF and ZPK representations are compact and convenient for display purposes, they are not ideal for system manipulation and analysis for several reasons:

- Working with TF and ZPK models often results in high-order polynomials whose evaluation can be plagued by inaccuracies.
- The TF and ZPK representations are inefficient for manipulating MIMO systems and tend to inflate the model order.

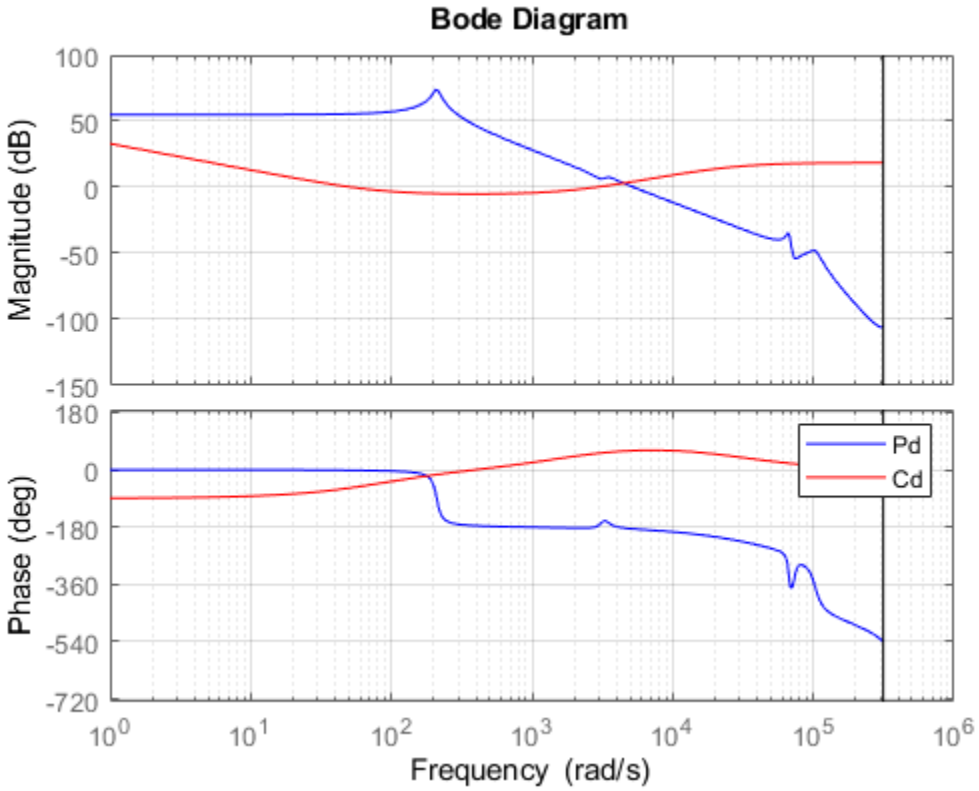
Some of these limitations are illustrated below. Because of these limitations, you should use the SS or FRD representations for most computations involving LTI models.

Pitfalls of High-Order Transfer Functions

Computations involving high-order transfer functions can suffer from severe loss of accuracy and even overflow. Even a simple product of two transfer functions can give surprising results, as shown below.

Load and plot two discrete-time transfer functions `Pd` and `Cd` of order 9 and 2, respectively:

```
% Load Pd,Cd models
load numdemo Pd Cd

% Plot their frequency response
bode(Pd,'b',Cd,'r'), grid
legend('Pd','Cd')
```

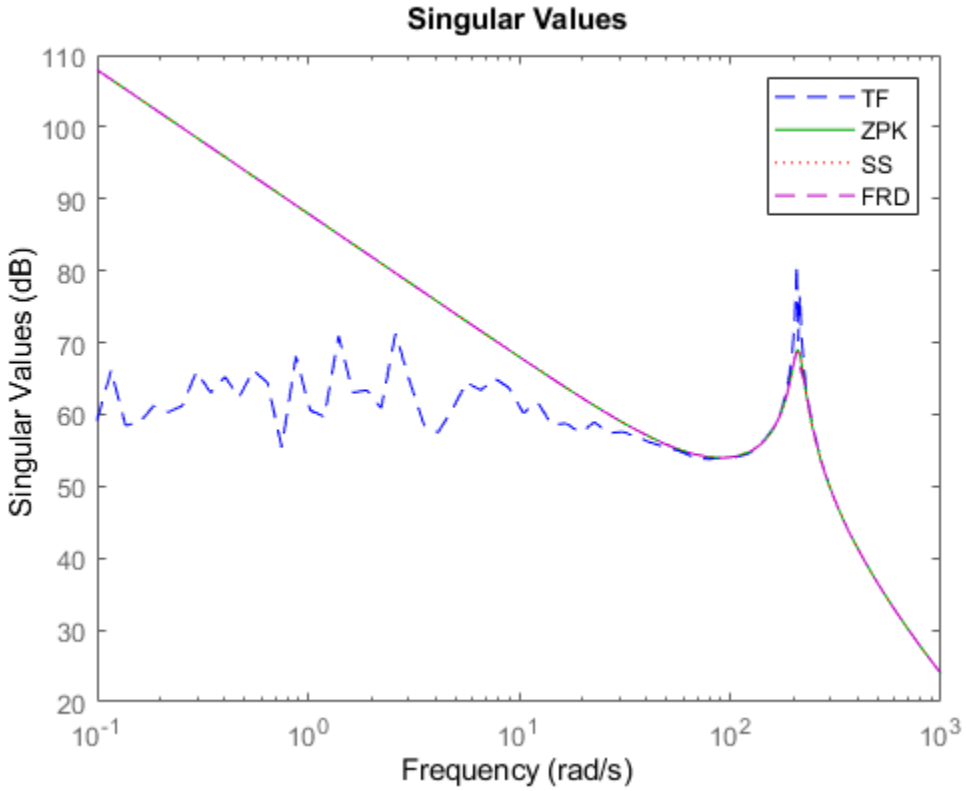


Next, compute the open-loop transfer function L = Pd*Cd using the TF, ZPK, SS, and FRD representations:

```
Ltf = Pd * Cd;                  % TF
Lzp = zpk(Pd) * Cd;             % ZPK
Lss = ss(Pd) * Cd;              % SS
w = logspace(-1,3,100);
Lfrd = frd(Pd,w) * Cd;          % FRD
```

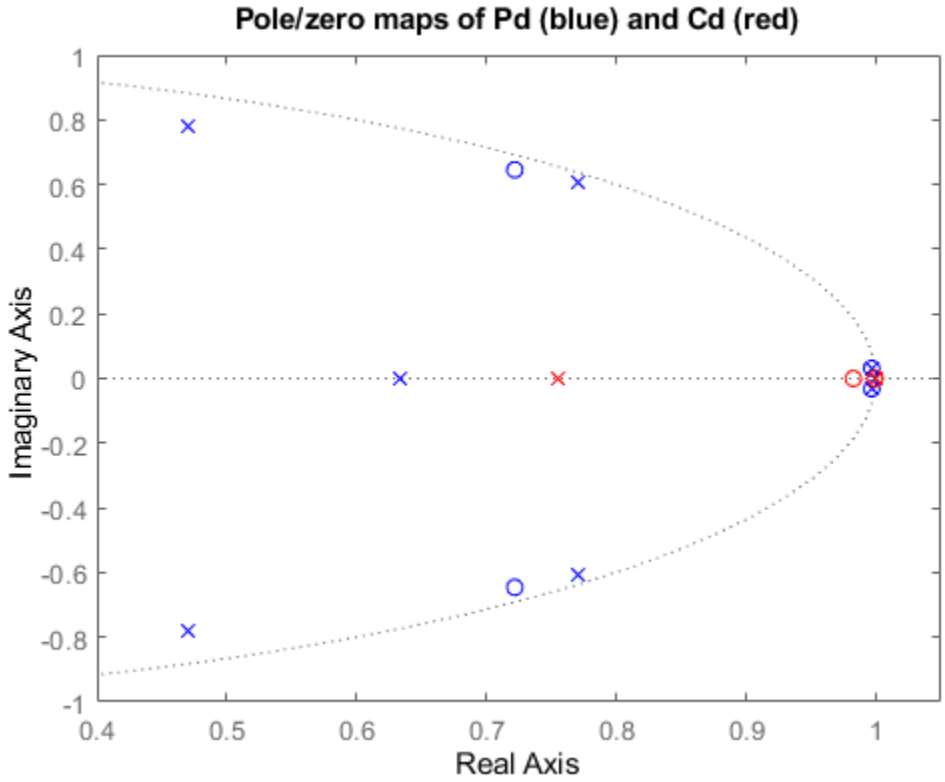
Finally, compare the frequency response magnitude for the resulting four models:

```
sigma(Ltf,'b--',Lzp,'g',Lss,'r:',Lfrd,'m--',{1e-1,1e3});
legend('TF','ZPK','SS','FRD')
```



The responses from the ZPK, SS, and FRD representations closely match, but the response from the TF representation is choppy and erratic below 100 rad/sec. To understand the loss of accuracy with the transfer function form, compare the pole/zero maps of Pd and Cd near z=1:

```
pzplot(Pd,'b',Cd,'r');
title('Pole/zero maps of Pd (blue) and Cd (red)');
axis([0.4 1.05 -1 1])
```



Note that there are multiple roots near z=1. Because the relative accuracy of polynomial values drops near roots, the relative error on the transfer function value near z=1 exceeds 100%. The frequencies below 100 rad/s map to `|z-1|`<1e-3, which explains the erratic results below 100 rad/s.

Pitfalls of Back-and-Forth Conversions Between Representations

You can easily convert any LTI model to transfer function, zero-pole-gain, or state-space form using the commands `tf`, `zpk`, and `ss`, respectively. For example, given a two-input, two-output random state-space model HSS1 created using

```
HSS1 = rss(3,2,2);
```

you can obtain its transfer function using

```
HTF = tf(HSS1);
```

and convert it back to state-space using

```
HSS2 = ss(HTF);
```

However, beware that such back-and-forth conversions are expensive, can incur some loss of accuracy, and artificially inflate the model order for MIMO systems. For example, the order of `HSS2` is double that of `HSS1` because 6 is the generic order of a 2x2 transfer matrix with denominators of degree 3:

```
order(HSS1)
```

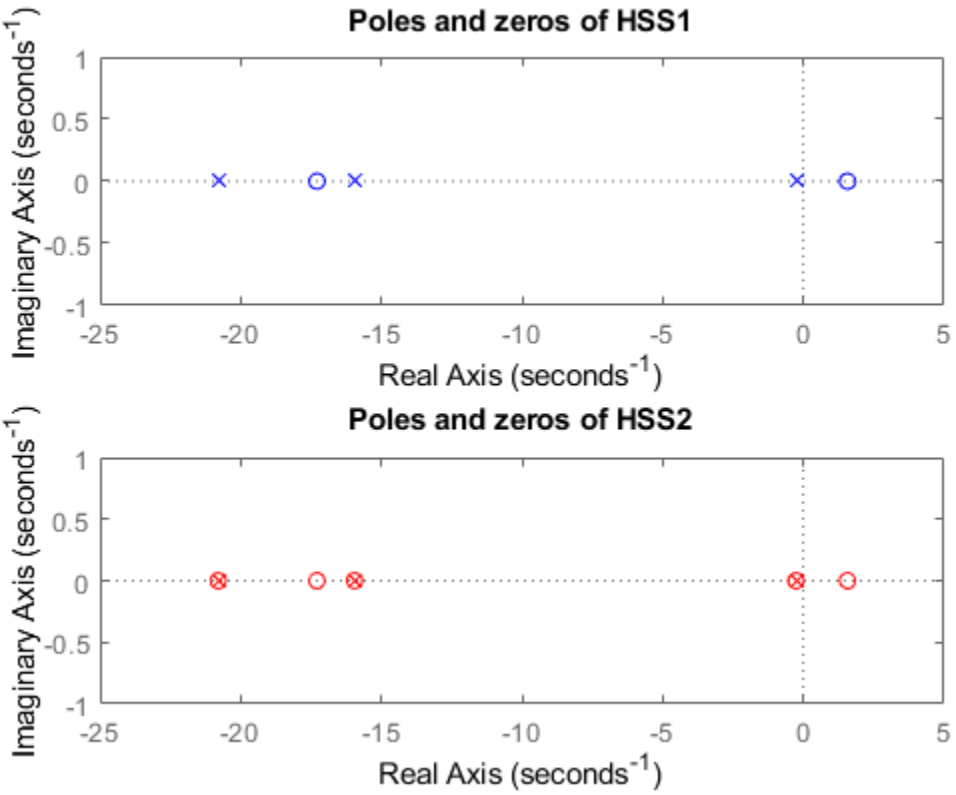
```
ans = 3
```

```
order(HSS2)
```

```
ans = 6
```

To understand the difference in model order, compare the pole/zero maps of the two models:

```
subplot(211)
pzmap(HSS1,'b')
title('Poles and zeros of HSS1');
subplot(212)
pzmap(HSS2,'r')
title('Poles and zeros of HSS2');
```



Notice the cancelling pole/zero pairs in HSS2 depicted by x's inside o's in the pole/zero map. You can use the command `minreal` to eliminate cancelling pole/zero pairs and recover a 3rd-order, minimal state-space model from HSS2:

```
HSS2_min = minreal(HSS2);
```

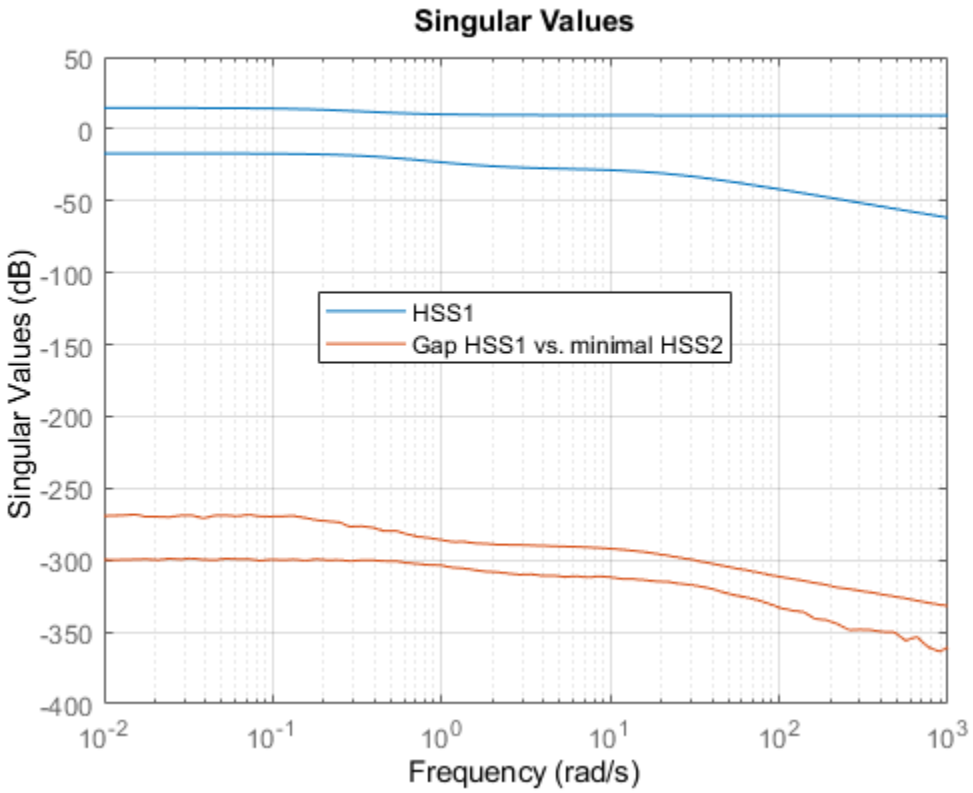
```
3 states removed.
```

```
order(HSS2_min)
```

```
ans = 3
```

Check that `HSS1` and `HSS2_min` coincide by plotting the relative gap between these two models:

```
clf
Gap = HSS1-HSS2_min;
sigma(HSS1,Gap), grid

Warning: The frequency response has poor relative accuracy. This may be because the response is n

legend('HSS1','Gap HSS1 vs. minimal HSS2','Location','Best')
```



The gap (green curve) is very small at all frequencies. Note that `sigma` warns that the `Gap` plot is "noisy" because the difference is so small that it essentially consists of rounding errors.

Because extracting minimal realizations is numerically tricky, you should avoid creating nonminimal models. See also “Preventing State Duplication in System Interconnections” on page 4-28 for related insights.

Approximate Nonlinear Aircraft Pitch Dynamics Using LPV Model

This example shows how to approximate the nonlinear behavior of a system using a linear parameter-varying state-space model. This example is based on the “Approximate Nonlinear Behavior Using Array of LTI Systems” (Simulink Control Design)example, and shows how to approximate the nonlinear behavior at the command line.

In this example, you:

1. Linearize a nonlinear Simulink® model on a grid of trim points.
2. Construct an LPV model from the gridded array of LTI models and trim offsets.
3. Design a controller on a grid of trim points. This grid may be different from the grid used for the linearization.
4. Test the controller using command-line LTI simulations (at frozen operating points) and LPV simulations (along a specified parameter trajectory).
5. Test the controller on the nonlinear plant in Simulink.

Aircraft Model

The airframe dynamics are nonlinear and the aerodynamic forces and moments depend on speed V and incidence α. The `scdairframeOpenLoop` model describes these dynamics.

```
open_system('scdairframeOpenLoop')
```

Range Of Operating Conditions

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);
```

Batch Trimming

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 ct = 1:nA*nV
   alpha_ini = alpha(ct);      % Incidence [rad]
   v_ini = V(ct);              % Speed [m/s]

   % Specify trim condition
   opspec(ct) = operspec('scdairframeOpenLoop');

   % 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('scdairframeOpenLoop',opspec,opt);
```

Batch Linearization

To linearize the model, first specify linearization input and output points.

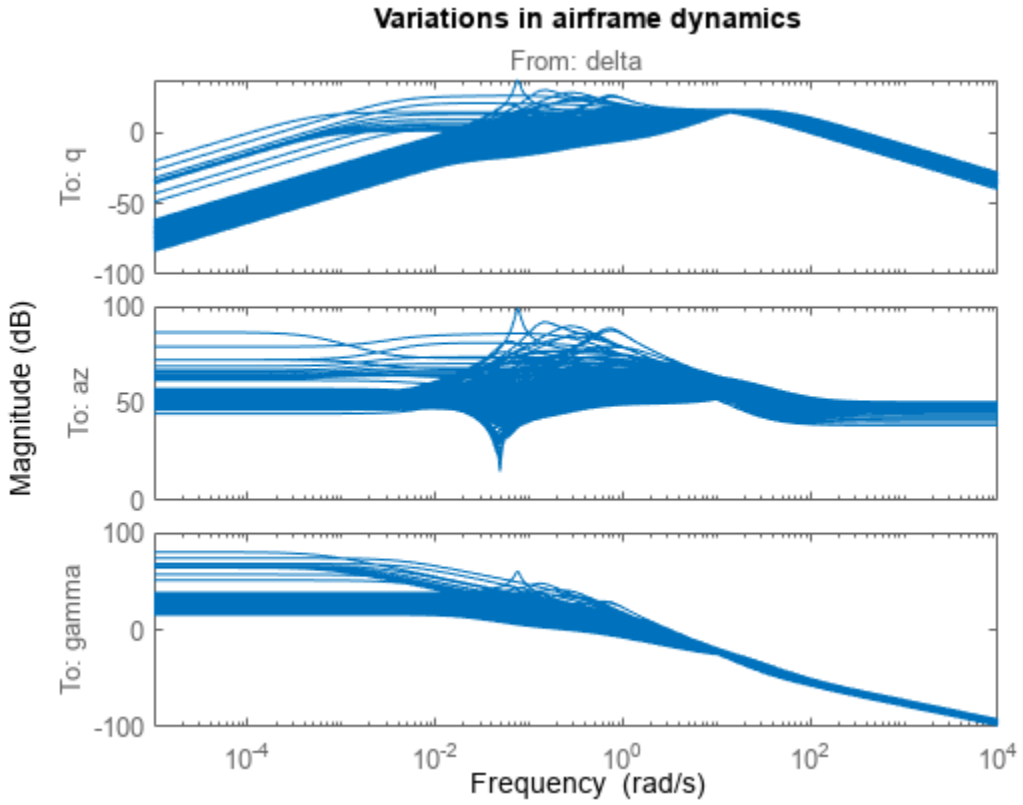
```
io = [linio('scdairframeOpenLoop/delta',1,'in');            % delta
      linio('scdairframeOpenLoop/Airframe Model',1,'out');  % alpha
      linio('scdairframeOpenLoop/Airframe Model',2,'out');  % V
      linio('scdairframeOpenLoop/Airframe Model',3,'out');  % q
      linio('scdairframeOpenLoop/Airframe Model',4,'out');  % az
      linio('scdairframeOpenLoop/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('scdairframeOpenLoop',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).

```
bodemag(G(3:5,:,:,:))
title('Variations in airframe dynamics')
```



Use `ssInterpolant` to construct an LPV model that interpolates these linearized models between grid points.

```
offsets = info.Offsets;
Glpv = ssInterpolant(G,offsets);
```

Control Design: Pitch-Axis Stability Augmentation

The (α,V) grid for control design is coarser and on a smaller range than for the plant model.

```
nAcd = 5;   % number of alpha values
nVcd = 3;   % number of V values
alphacd = linspace(-10,10,nAcd)*pi/180;
Vcd = linspace(900,1200,nVcd);
[alphacd,Vcd] = ndgrid(alphacd,Vcd);
Ncd = nAcd*nVcd;
```

Use a 2-DOF gain-scheduled architecture for the pitch rate loop. This combines an I-only term on the error signal $e = q_{\text{ref}} - q$ with a proportional term on q.

$$\delta = \frac{K_i}{s}(q_{\text{ref}} - q) + K_p q$$

The tunable gains K_p and K_i are defined as gain surfaces with a linear dependence on (α,V).

```
s = tf('s');
Domain = struct('alpha',alphacd,'V',Vcd);
```

```
ShapeFcn = @(x,y) [x y];
Ki = tunableSurface("Ki",-1,Domain,ShapeFcn);
Kp = tunableSurface("Kp",1,Domain,ShapeFcn);
K = [Ki/s Kp] * [1 -1;0 1];
K.InputName = {'qref';'q'};
K.OutputName = {'delta'};
```

Sample the LPV model of the airframe over the tuning grid `(alphacd,Vcd)` and build the closed-loop model.

```
[Ga,Goffsets] = sample(Glpv,[],alphacd,Vcd);
T0 = connect(Ga,K,'qref','q','delta');
```

```
Warning: The following block outputs are not used: alpha,V,az,gamma.
```

Tune the gain surfaces subject to step response and loop-shaping requirements.

```
R1 = TuningGoal.StepTracking('qref','q',1/2);
R2 = TuningGoal.MaxLoopGain('delta',50,1);
R3 = TuningGoal.MaxGain('qref','q',tf(15,[1 0.01]));
T = systune(T0,[R1 R2],R3);
```

```
Final: Soft = 4.94, Hard = 0.19413, Iterations = 30
```

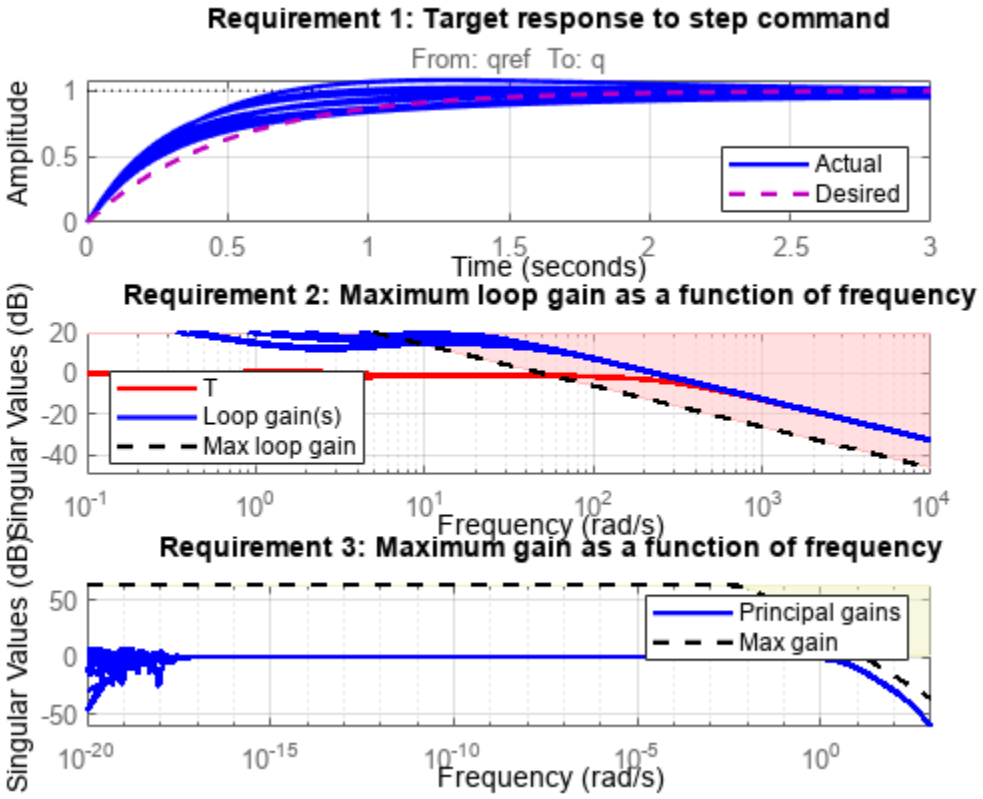
```
Warning: StepTracking goal: Feedback configuration has fixed integrators that cannot be stabilize
```

```
showTunable(T)
```

```
Ki =
   -4.4209   -0.0712    1.2825
-----------------------------------
Kp =
    1.1167   -0.0000   -0.2609
```

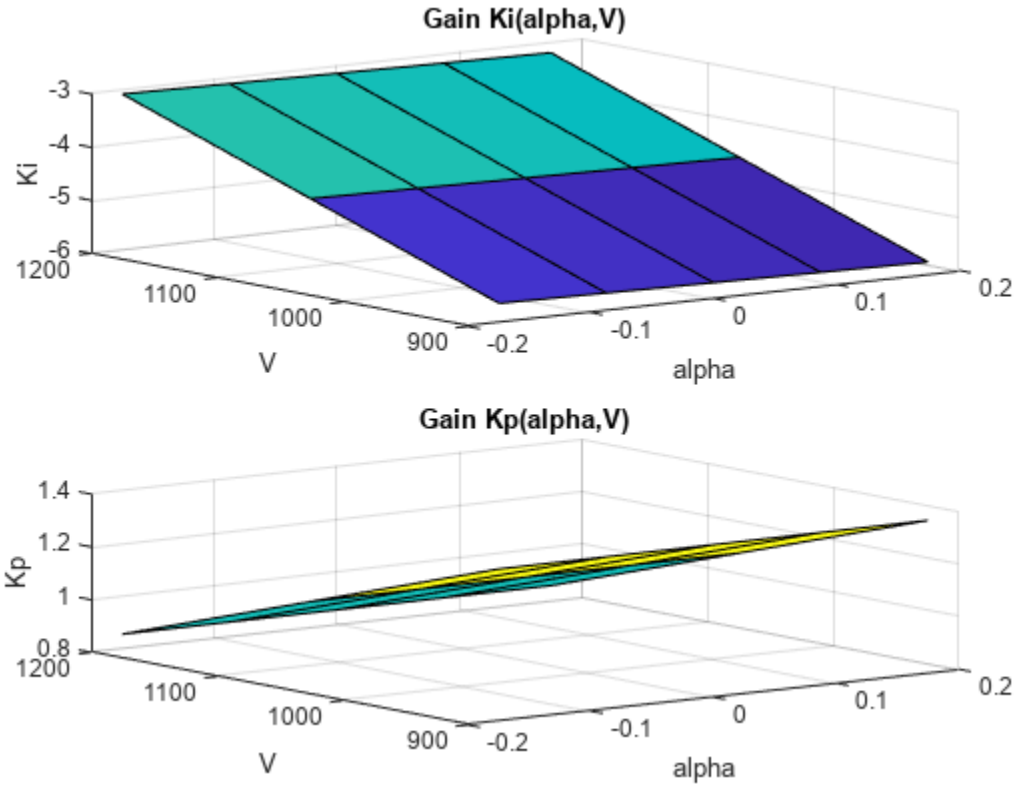
View the tuning results.

```
viewGoal([R1 R2 R3],T)
```



The tuned gain surfaces show a weak dependence on α and a stronger dependence on V.

```
Ki = setBlockValue(Ki,T);
clf
subplot(211)
viewSurf(Ki)
Kp = setBlockValue(Kp,T);
subplot(212)
viewSurf(Kp)
```



Build the LPV controller. Include the trim deflection as output offset so that the controller performs an incremental correction around trim.

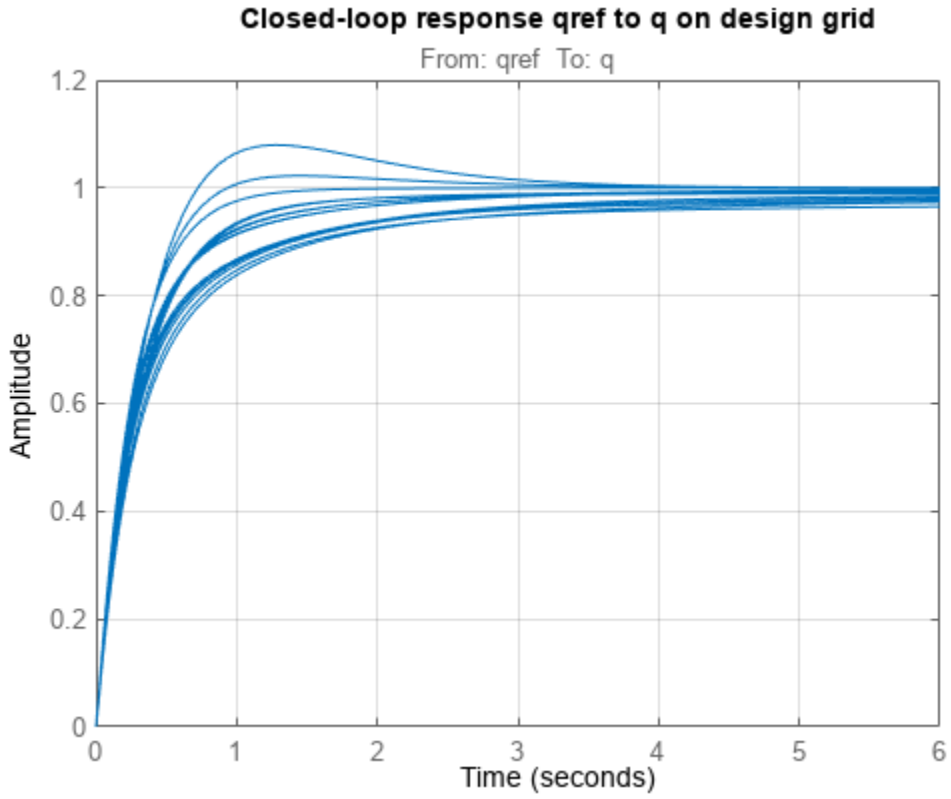
```
Ka = ss(setBlockValue(K,T));
Koffsets = reshape(struct('y',{Goffsets.u}),size(Goffsets));
Klpv = ssInterpolant(Ka,Koffsets);
```

Build the closed-loop LPV model. Note that the plant and controller are sampled and interpolated on two different (α,V) grids.

```
Tlpv = feedback(Glpv*Klpv,1,2,3,+1);
Tlpv = Tlpv(:,1); % from qref to plant outputs
```

Plot the closed-loop responses from q_{ref} to q over the design grid.

```
clf
step(sample(Tlpv(3,:),[],alphacd,Vcd),6);
grid
title('Closed-loop response qref to q on design grid')
```



Nonlinear Simulation

To compare the nonlinear and LPV simulation, simulate the step response of the pitch rate loop initialized from one of the trim operating points. This trim condition is chosen on the denser grid and does not belong to the design grid.

```
aidx = 9;
Vidx = 6;
alpha0 = alphaRange(aidx);
V0 = VRange(Vidx);
q0 = offsets(aidx,Vidx).x(4);
```

Pick the initial state x_{K0} of the LPV controller to deliver the trim deflection δ_0 for the selected operating point. Since the LPV controller output is

$\delta = \delta_0 + C_K x_K + D_{K,2} q,$

x_{K0} must satisfy

$C_K x_{K0} + D_{K,2} q_0 = 0.$

```
K0 = sample(Klpv,[],alpha0,V0);
xK0 = -K0.C\K0.D(2)*q0;
```

Starting from the trim condition for `(alpha0,V0)`, apply a step change at t=0 from q0 to q0+0.05:

```
tstep = 0;
rstep0 = q0;
```

```
rStepAmp = 0.05;
rstepf = rstep0+rStepAmp;
Tf = 5;
```

Open the closed-loop Simulink model and initialize the nonlinear simulation.

```
open_system('scdairframeClosedLoop')

alpha_ini = alpha0;
v_ini = V0;
q_ini = q0;
yK0 = reshape([Koffsets.y],[1 1 nAcd nVcd]);
```

Acquire the nonlinear simulation results.

```
sim('scdairframeClosedLoop',[0 tstep+Tf]);
tsim = y.Time;
ysim = y.Data;
```

For comparison, first compute the LTI response at this operating condition.

```
t = linspace(0,Tf,250);
y = step(sample(Tlpv,[],alpha0,V0),t);
ylin = [alpha0,V0,q0] + rStepAmp * y(:,1:3);
```

LPV Simulation with Ideal Parameter Trajectory

Compute the response with the LPV approximation `Glpv` of the nonlinear airframe model. To do this you need the $p = (\alpha, V)$ trajectory. A first option is to use the approximate trajectory from the LTI simulation above (recall that α and V are the first two outputs of `Tlpv`).

```
p_ideal = ylin(:,1:2);
```

The initial state of the plant is available from the trim analysis. The overall initial state `xinit` is obtained by appending the initial state `xK0` of `Klpv`:

```
xinit = [offsets(aidx,Vidx).x ; xK0];
```

Simulate the LPV response with the approximate parameter trajectory `p_ideal`.

```
qref = rstepf*ones(1,numel(t));
y1 = lsim(Tlpv,qref,t,xinit,p_ideal);
```

The true parameter trajectory is endogenous since α,V depend on the states u,w according to

$$\tan(\alpha) = \frac{w}{u},\ V = \sqrt{u^2 + w^2}$$

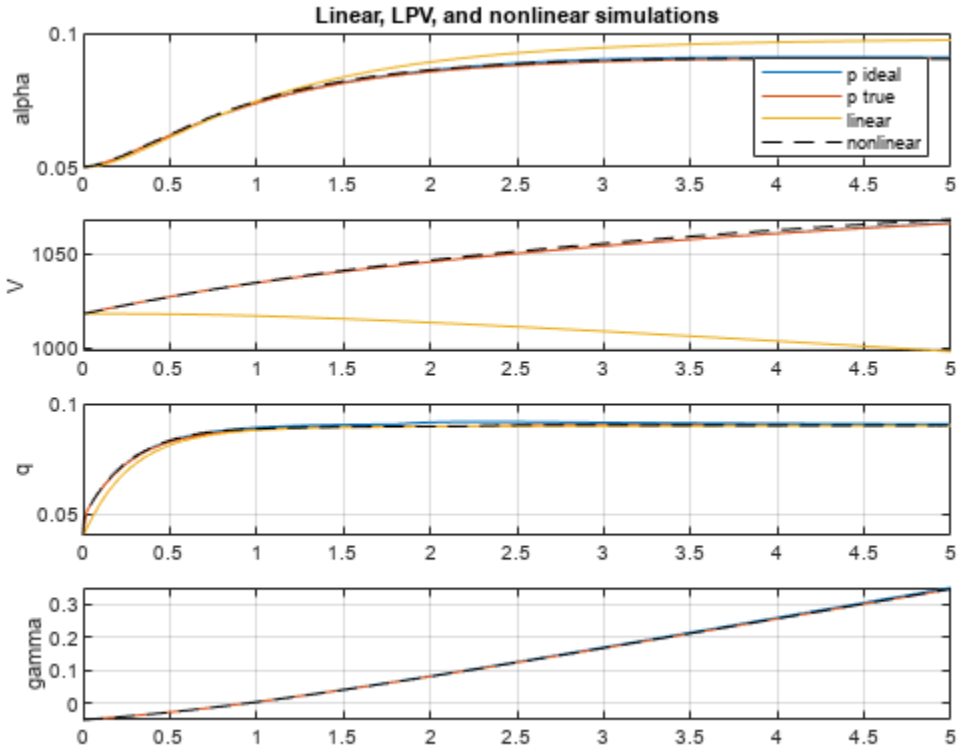
For more accurate results, you can describe this dependency using a function $p = F(t, x, u)$.

```
pFcn = @(t,x,u) [atan2(x(3),x(2)) ; sqrt(x(2)^2+x(3)^2)];
[y2,~,~,p2] = lsim(Tlpv,qref,t,xinit,pFcn);
```

Compare the simulation results.

```
figure(1)
clf
subplot(411)
plot(t,y1(:,1),t,y2(:,1),t,ylin(:,1),tsim,ysim(:,1),'k--')
```

```
ylabel('alpha')
grid on
title('Linear, LPV, and nonlinear simulations')
legend('p ideal','p true','linear','nonlinear','location','southeast')
subplot(412)
plot(t,y1(:,2),t,y2(:,2),t,ylin(:,2),tsim,ysim(:,2),'k--')
ylabel('V')
grid on
subplot(413)
plot(t,y1(:,3),t,y2(:,3),t,ylin(:,3),tsim,ysim(:,3),'k--')
ylabel('q')
grid on
subplot(414)
plot(t,y1(:,5),t,y2(:,5),tsim,ysim(:,5),'k--')
ylabel('gamma')
grid on
```



The LPV simulations are very close to the nonlinear simulation, confirming that the LPV model of the airframe is an effective surrogate and that the gain-scheduled LPV controller is performing well. As expected, the LPV simulation using the true parameter trajectory is slightly more accurate than its surrogate using `p_ideal`.

Close the models.

```
bdclose('scdairframeOpenLoop')
bdclose('scdairframeClosedLoop')
```

See Also

`ssInterpolant` | `sample` | `linearize` | `systune` | `ndgrid`

Related Examples

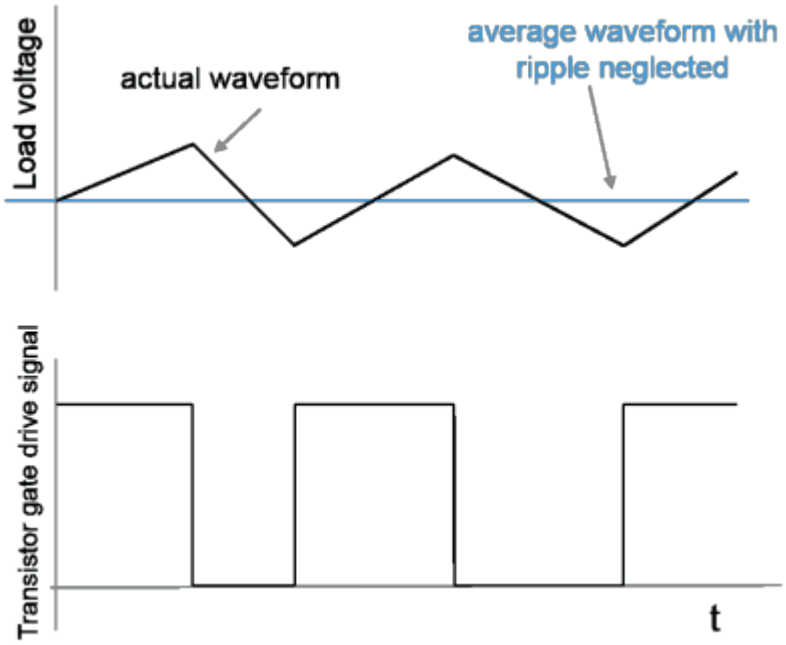
- “LTV and LPV Modeling” on page 1-26
- “Using LTV and LPV Models in MATLAB and Simulink” on page 1-32
- “LPV Approximation of Boost Converter Model” on page 1-78
- “LPV Model of Magnetic Levitation Model from Batch Linearization Results” on page 1-133
- “Control Design for Wind Turbine” on page 1-140

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 using the `lpvss` object. This example uses the model from the "LPV Approximation of Boost Converter Model" (Simulink Control Design) example to construct an LPV approximation at the command line. The LPV approximation allows quick analysis of average behavior at various operating conditions.

Boost Converter Model

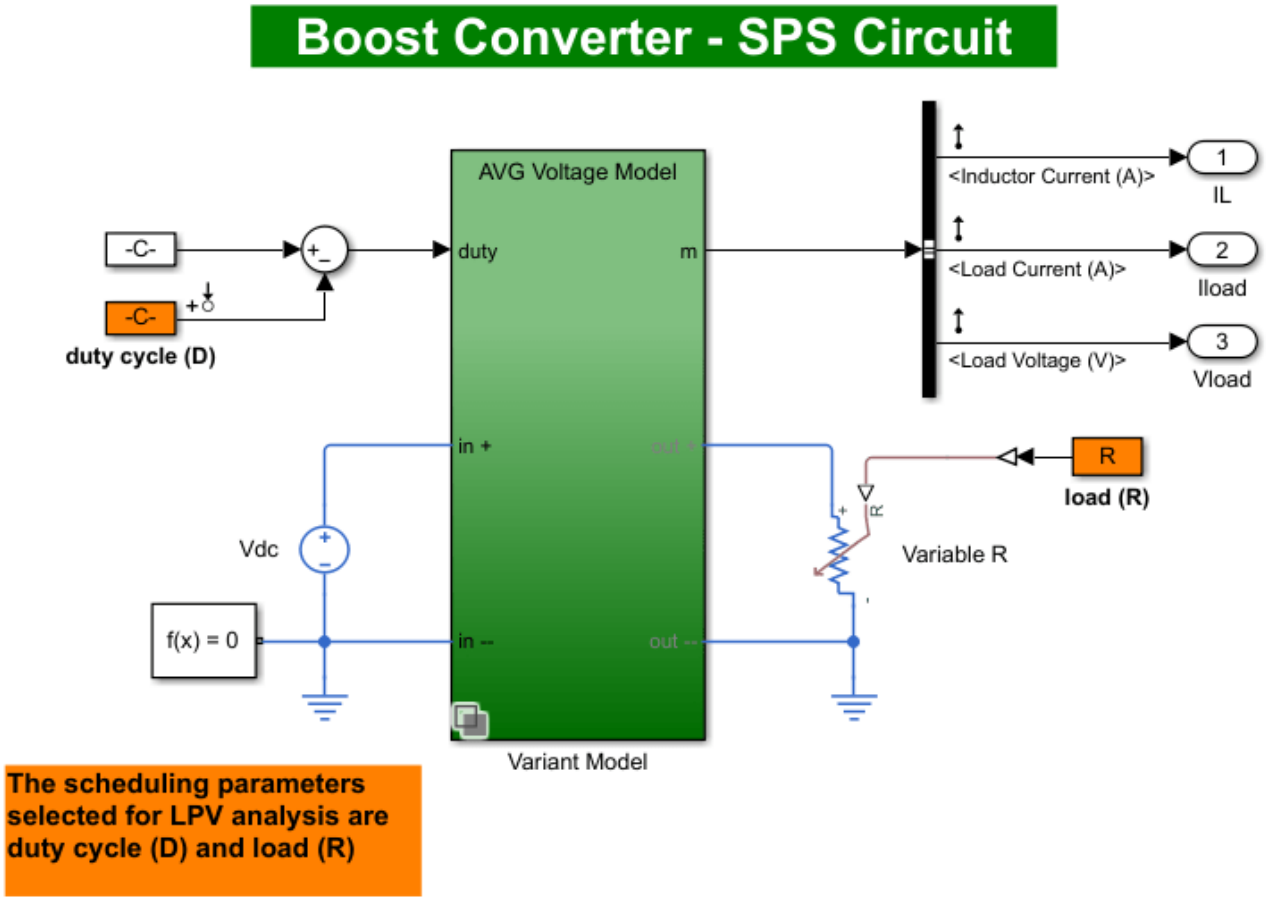
A Boost Converter circuit converts a DC voltage to another DC voltage by controlled chopping or switching of the source voltage. The request for a certain load voltage is translated into a corresponding requirement for the transistor duty cycle. The duty cycle modulation is typically several orders of magnitude slower than the switching frequency, which produces an average voltage with relatively small ripples.



In practice, there are also disturbances in the source voltage and the resistive load affecting the actual load voltage.

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.

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;
```

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.

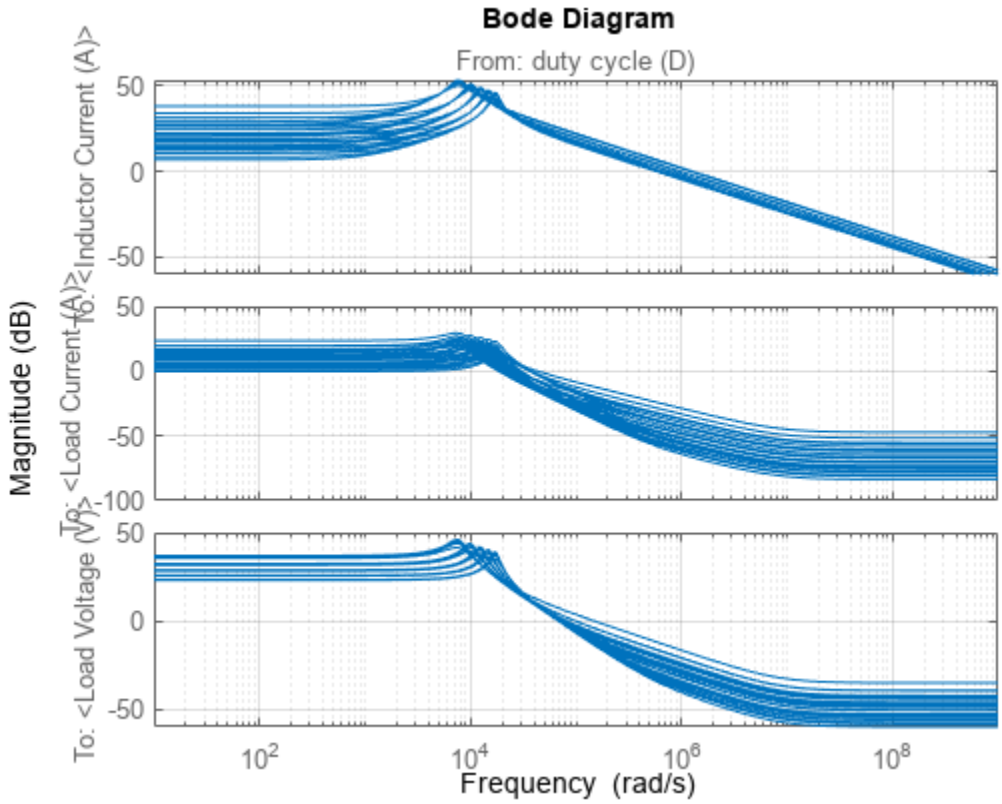
```
opt = linearizeOptions('StoreOffsets', true);
```

Linearize the model at the operating points in array `op`.

```
[linsys,~,info] = linearize(mdl,op,io,params,opt);
```

Plot the linear system array.

```
bodemag(linsys)
grid on
```



LPV Model

Use `ssInterpolant` to create an LPV model that interpolates the linearized models and offsets over the (*d*,*R*) operating range.

```
lpvsys = ssInterpolant(linsys,info.Offsets);
```

LPV Simulation

To simulate 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;
```

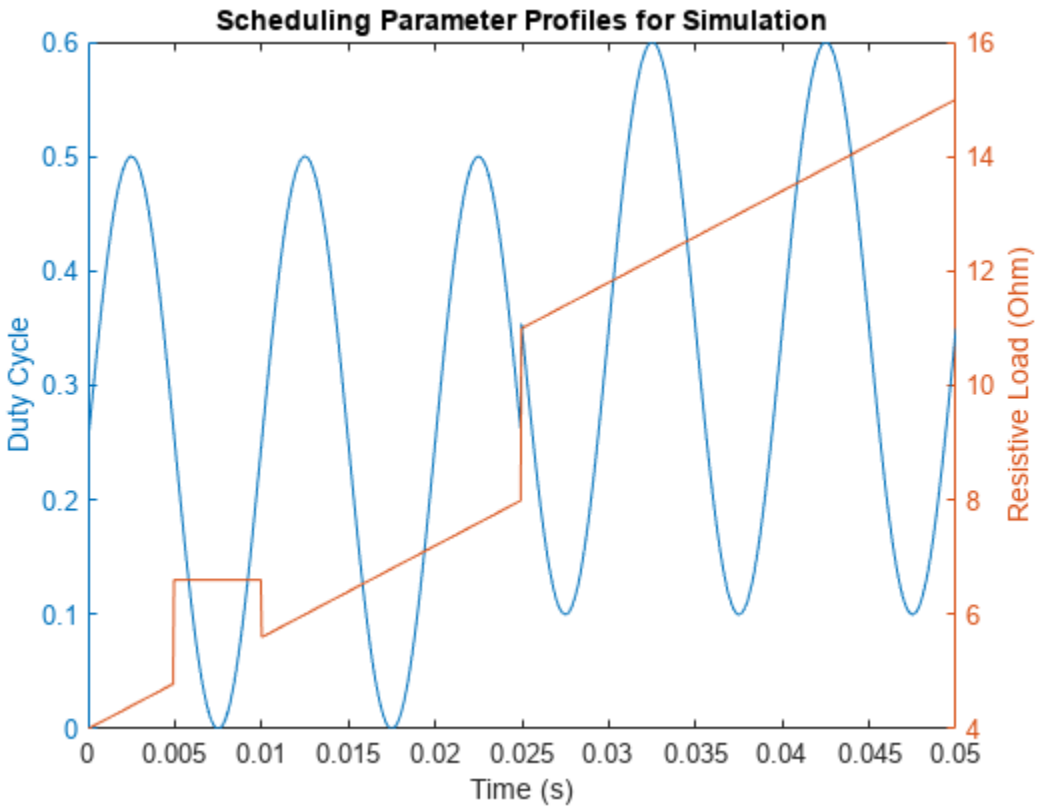
Generate the resistive load profile `rin`.

```
rin = linspace(4,12,length(t))';
rin(500:end) = rin(500:end)+3;
rin(100:200) = 6.6;
```

Plot the scheduling parameter profiles.

```
clf
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')
```



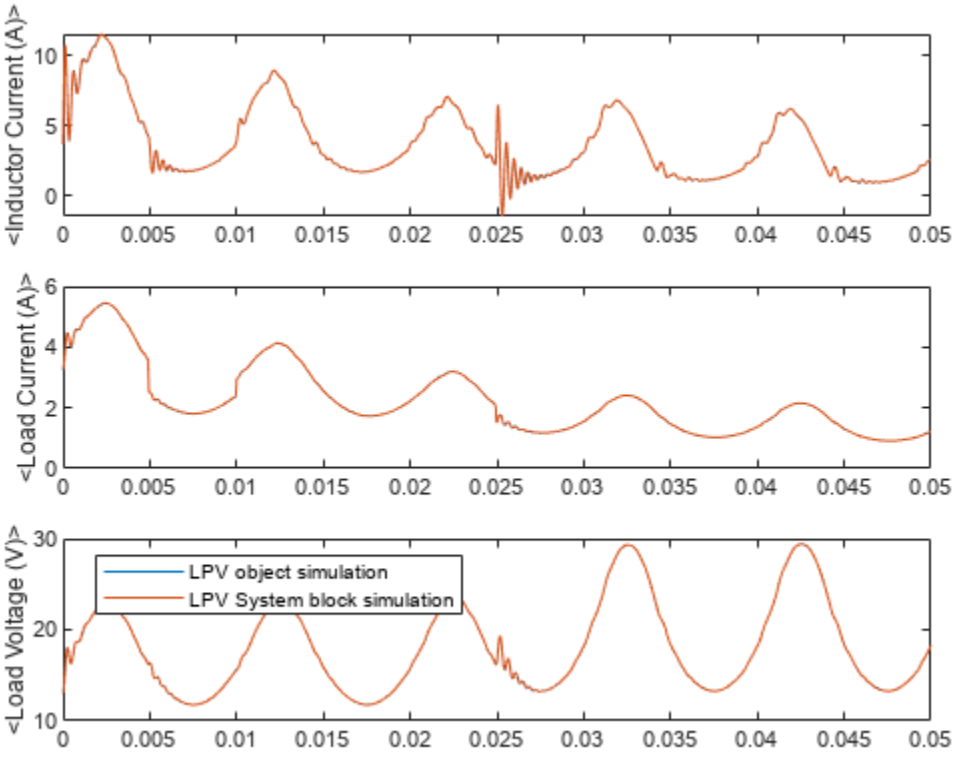
Use `lsim` to simulate the response of the LPV approximation to this stimulus.

```
p = [din rin];  % scheduling variables
xinit = info.Offsets(1).x;
y = lsim(lpvsys,din,t,xinit,p);
```

Simulate the boost converter model LPV model implemented using the LPV System block.

```
% Offset data for LPV block
offsets = getOffsetsForLPV(info);
yoff = offsets.y;
xoff = offsets.x;
uoff = offsets.u;
simOut = sim('BoostConverterLPVModel','StopTime','0.05');
lpvBlockSim = simOut.logsout.getElement('ysim');
tsim = lpvBlockSim.Values.Time;
ysim = lpvBlockSim.Values.Data(:,:);
clf
subplot(311)
plot(t,y(:,1),tsim,ysim(:,1))
ylabel(lpvsys.OutputName{1});
subplot(312)
```

```
plot(t,y(:,2),tsim,ysim(:,2))
ylabel(lpvsys.OutputName{2});
subplot(313)
plot(t,y(:,3),tsim,ysim(:,3))
ylabel(lpvsys.OutputName{3});
legend('LPV object simulation','LPV System block simulation','location','best')
```

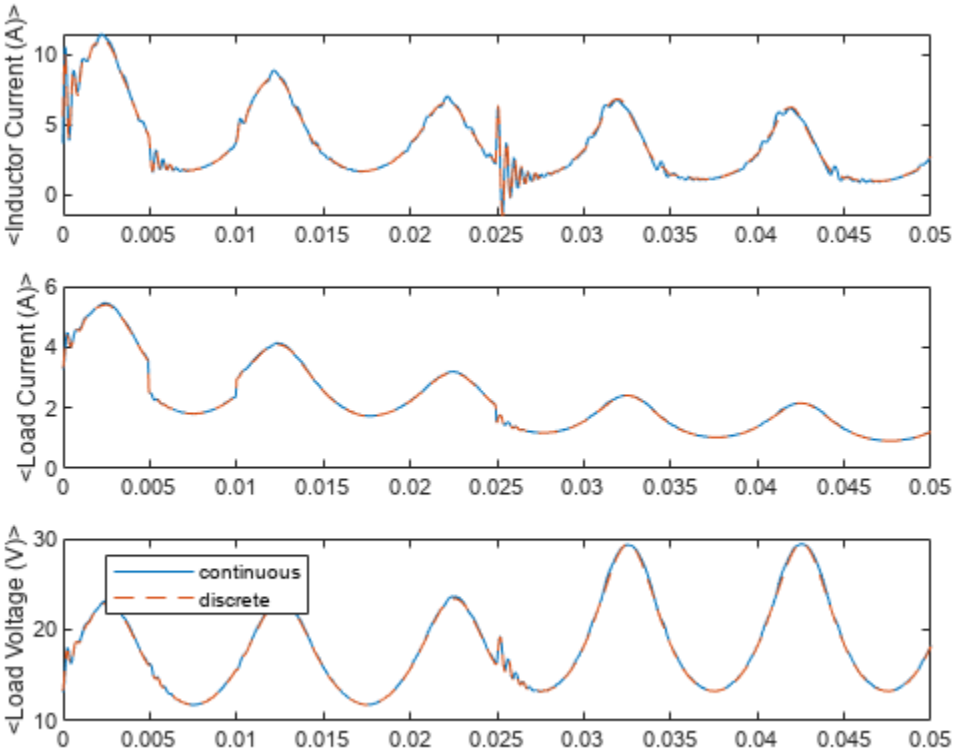


These results match the simulation results obtained with the LPV System block.

Discretization of LPV Model

Compute an equivalent discretized LPV model of the boost converter. This facilitates fixed-step simulation and code generation for this model.

```
dlpvsys = c2d(lpvsys,t(2)-t(1),'tustin');
yd = lsim(dlpvsys,din,t,xinit,p);
clf
subplot(311)
plot(t,y(:,1),t,yd(:,1),'--')
ylabel(lpvsys.OutputName{1});
subplot(312)
plot(t,y(:,2),t,yd(:,2),'--')
ylabel(lpvsys.OutputName{2});
subplot(313)
plot(t,y(:,3),t,yd(:,3),'--')
ylabel(lpvsys.OutputName{3});
legend('continuous','discrete','location','best')
```



Close the model

```
close_system(mdl,0)
```

See Also

`ssInterpolant` | `sample` | `linearize` | `ndgrid`

Related Examples

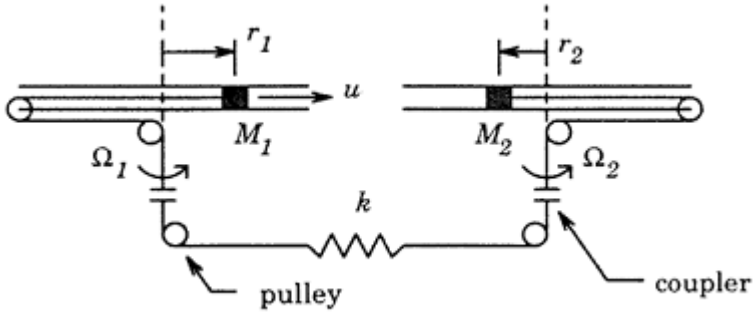
- “LTV and LPV Modeling” on page 1-26
- “Using LTV and LPV Models in MATLAB and Simulink” on page 1-32
- “Approximate Nonlinear Aircraft Pitch Dynamics Using LPV Model” on page 1-68
- “LPV Model of Magnetic Levitation Model from Batch Linearization Results” on page 1-133
- “Control Design for Wind Turbine” on page 1-140

Control Design for Spinning Disks

This example designs a gain-scheduled H_2 controller for a mechanical system consisting of coupled spinning disks described in [1] on page 1-91.

Coupled Spinning Disks

A pair of rotating disks is shown in this figure. The slotted disks rotate at rates Ω_1 rad/s and Ω_2 rad/s. The disks contain masses M_1 and M_2, which can move in the horizontal plane and slide radially. The two masses are connected by a wire, coupler, and pulley system that can transmit force in both compression and tension. This coupling system acts as a spring with spring constant k and friction due to the sliding motion of each mass is modeled by a damping coefficient b.



The goal is to control the position of the two masses: r_1 for mass M_1 and r_2 for M_2. The control input is a radial force f_u acting on mass M_1 and there are radial disturbance forces f_1 and f_2 acting on each mass. The equations of motions are as follows.

$$M_1\left(\ddot{r}_1 - \Omega_1^2 r_1\right) + b\,\dot{r}_1 + k(r_1 + r_2) = f_u + f_1$$

$$M_2\left(\ddot{r}_2 - \Omega_2^2 r_2\right) + b\,\dot{r}_2 + k(r_1 + r_2) = f_2$$

Here:

- M_1 = 1 kg and is the mass of the sliding mass in disk 1.
- M_1 = 2 kg and is the mass of the sliding mass in disk 2.
- b = 1 kg/s and is the damping coefficient due to friction.
- k = 200 N/m is the spring constant of mass coupling system.
- $r_1(t)$ is the position of the sliding mass M_1 relative to the center of disk 1 (m).
- $r_2(t)$ is the position of the sliding mass M_2 relative to the center of disk 2 (m).
- $\Omega_1(t)$ is the rotational rate of disk 1 (rad/s).
- $\Omega_2(t)$ is the rotational rate of disk 2 (rad/s).
- f_u is the control force acting radially on mass M_1(N).
- f_1 and f_2 are the disturbance forces acting radially on M_1 and M_2, respectively (N).

The rotational rates of the spinning disks are allowed to vary in the ranges $\Omega_1 \in [0, 3]$ and $\Omega_2 \in [0, 5]$ (in rad/s). These rates are not known in advance but are measured and available for control design. The objective of the control design is to command the radial position of mass M_2. Note that the control input is applied to mass M_1 and is transmitted to mass M_2 through the disk coupling system.

LPV Model

The equations of motions are linear except for the rates Ω_1 and Ω_2. Choosing $p_1 = \Omega_1^2$, $p_2 = \Omega_2^2$, $p_1 \in [0, 9]$, and $p_2 \in [0, 25]$ as scheduling parameters, you obtain the following LPV model of the overall system.

$$\dot{x}(t) = \begin{bmatrix} 0 & 0 & 1 & 0 \\ 0 & 0 & 0 & 1 \\ p_1 - \frac{k}{M_1} & -\frac{k}{M_1} & -\frac{b}{M_1} & 0 \\ -\frac{k}{M_2} & p_2 - \frac{k}{M_2} & 0 & -\frac{b}{M_2} \end{bmatrix} x(t) + \begin{bmatrix} 0 & 0 & 0 \\ 0 & 0 & 0 \\ \frac{1}{M_1} & \frac{0.1}{M_1} & 0 \\ 0 & 0 & \frac{0.1}{M_2} \end{bmatrix} \begin{bmatrix} f_u(t) \\ f_1(t) \\ f_2(t) \end{bmatrix}$$

$$y(t) = [0\ 1\ 0\ 0]x(t)$$

Here, the state vector $x(t) = \begin{bmatrix} r_1(t) \\ r_2(t) \\ \dot{r}_1(t) \\ \dot{r}_2(t) \end{bmatrix}$.

To create this LPV model, write a function `disksGFCN` that computes the A, B, C matrices as a function of the scheduling parameters $p = \begin{bmatrix} p_1 \\ p_2 \end{bmatrix}$. To see the code for this function, see the `disksGFCN.m` file or enter `type disksGFCN` at the command line.

Define system parameters

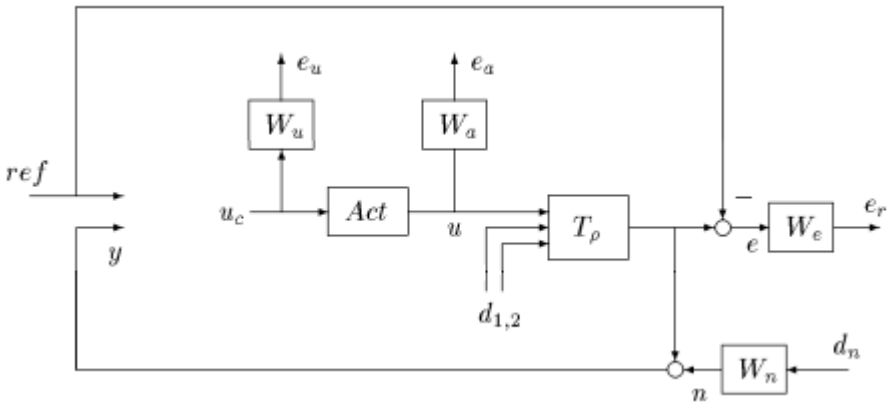
```
m1 = 1;
m2 = 0.5;
k = 200;
b = 1;
```

Define the LPV system.

```
fh = @(t,p) disksGFCN(t,p,m1,m2,k,b);
G = lpvss(["p1","p2"],fh);
```

Gain-Scheduled H2 Control Design

Design a gain-scheduled H_2-optimal controller. First, design an H_2-optimal controller at each point on a grid of (Ω_1^2, Ω_2^2) values. For fixed rotational rates, such controllers minimize the closed-loop H_2 norm from disturbance $d = \begin{bmatrix} f_1(t) \\ f_2(t) \end{bmatrix}$ to error e for the weighted interconnection. For appropriately selected weights, this interconnection defines the desired H_2 performance of the control law. The controllers takes in the reference r and measurement y and generates the command u_c.



Define the weights and build an LPV model of this interconnection.

```
% Add input/output names to plant
G.InputName = {'u'; 'f1'; 'f2'};
G.OutputName = 'Gy';

% Actuator model
act = ss(-100,100,1,0);
act.InputName = 'uc';
act.OutputName = 'u';

% Weight: Control Command
Wu = ss(1/50);
Wu.InputName = 'uc';
Wu.OutputName = 'eu';

% Weight: Actuator Output
Wa = ss(0.00001);
Wa.InputName = 'u';
Wa.OutputName = 'ea';

% Weight: Tracking Error
We = tf(2,[1 0.04]);
We.InputName = 'e';
We.OutputName = 'er';

% Weight: Noise
Wn = tf([1,0.4],[0.01 400]);
Wn.InputName = 'dn';
Wn.OutputName = 'n';

% Form synthesis interconnection
sum1 = sumblk('y = Gy + n');
sum2 = sumblk('e = Gy - r');
IC = connect(G,act,Wu,Wa,We,Wn,sum1,sum2,{'r','f1','f2','dn','uc'},...
     {'eu','ea','er','r','y'});
```

Construct a grid of (Ω_1^2, Ω_2^2) values and compute the H_2-optimal controller at each grid point.

```
p1 = linspace(0,9,3);
p2 = linspace(0,25,3);
[p1,p2] = ndgrid(p1,p2);
```

Sample IC at grid points and design H_2-optimal controllers.

```
nmeas = 2;   % # of measurements
ncon = 1;    % # of controls
ICs = sample(IC,[],p1,p2);
Ks = ss( zeros([ncon nmeas size(p1)]) );
for i=1:numel(p1)
   Ks(:,:,i) = h2syn(ICs(:,:,i),nmeas,ncon);
end
```

Use `ssInterpolant` to create a gain-scheduled controller that linearly interpolates these H_2 controllers between grid points. This approach is valid when the rotational rates vary slowly. Alternative LPV methods for rapid changes are described in [1].

```
Ks.SamplingGrid = struct('p1',p1,'p2',p2);
Klpv = ssInterpolant(Ks,[]);
Klpv.InputName = {'r','Gy'};
Klpv.OutputName = {'uc'};
```

LTI Performance Evaluation

Evaluate the gain-scheduled controller for fixed rotation rates using a finer grid than for the controller design.

Form closed-loop sytem from `[r;f1;f2]` to `Gy`.

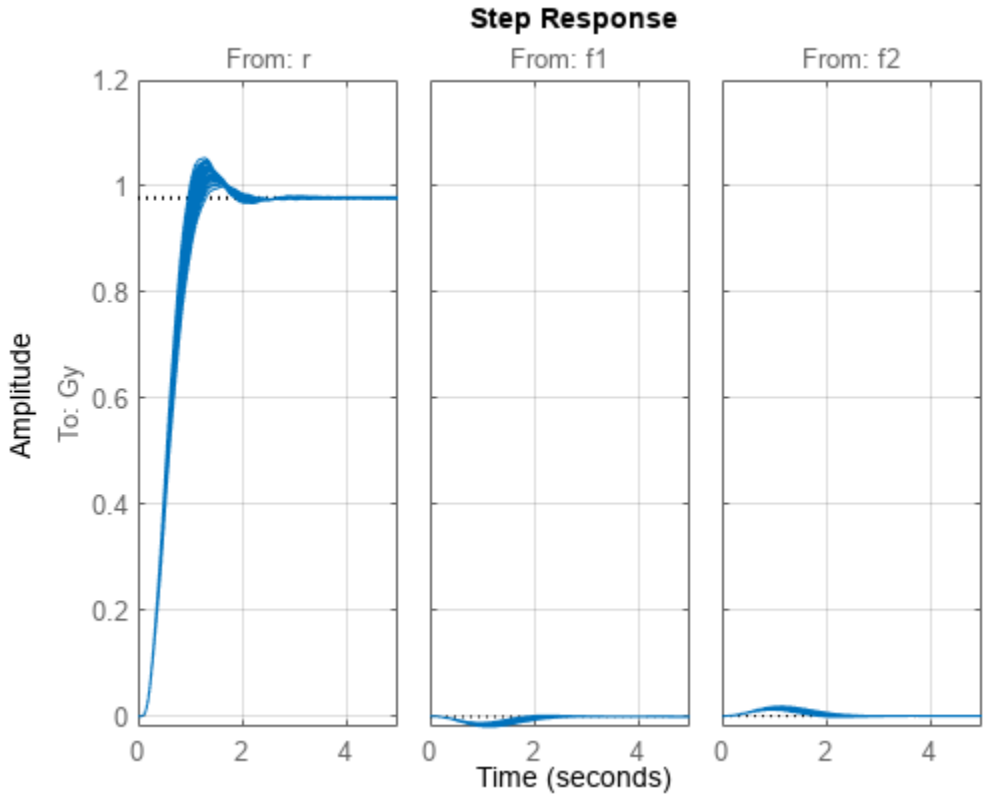
```
CL = connect(G,act,Klpv,{'r','f1','f2'},{'Gy'});
```

Evaluate open and closed-loop on a finer grid of points.

```
p1 = linspace(0,9,5);
p2 = linspace(0,25,5);
[p1,p2] = ndgrid(p1,p2);
Gs = sample(G,[],p1,p2);
CLs = sample(CL,[],p1,p2);
```

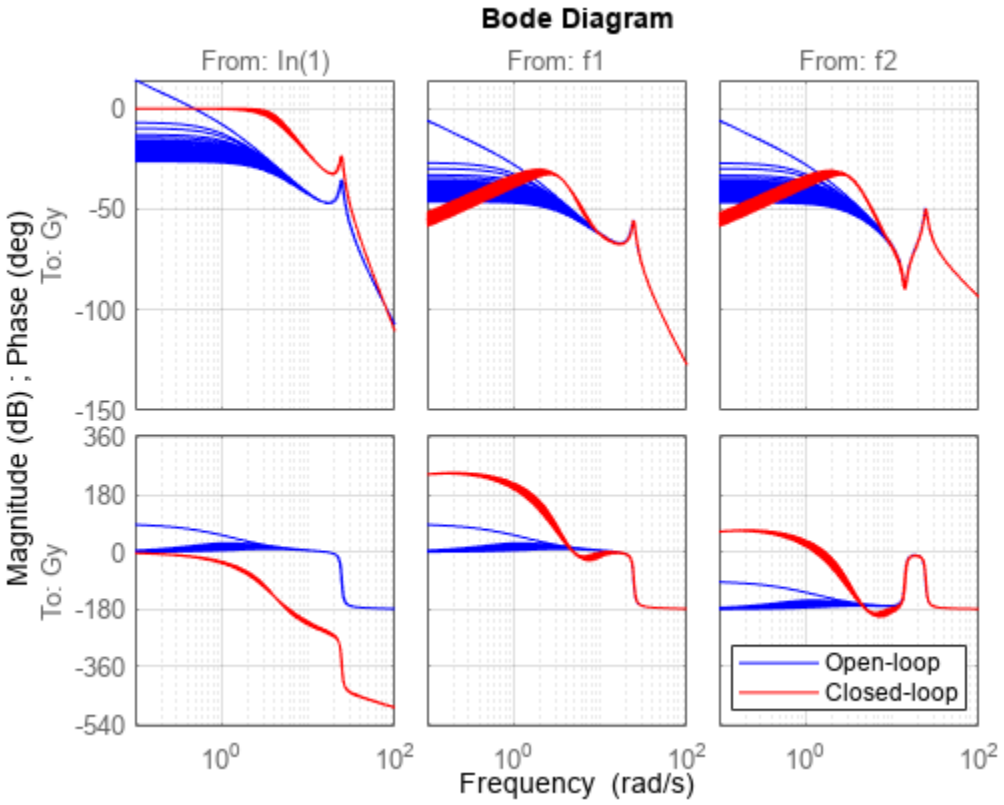
The controller achieves good tracking with minimal overshoot, approximately 2.5 sec settling time, and less than 3% steady-state tracking error. Furthermore, the disturbances have minimal effect.

```
step(CLs,5)
grid on
```



The closed-loop Bode response from `r` to `Gy` has a DC gain near one with a bandwidth near 3 rad/sec. It shows good disturbance attenuation below 1 rad/sec compared to the open loop.

```
bode(Gs,'b',CLs,'r',{0.1,100})
legend('Open-loop','Closed-loop','location','best')
grid minor
```



LPV Performance Evaluation

Finally, evaluate the gain-scheduled controller with a time-domain simulation with time-varying rotation rates. The parameters vary in time as follows:

$p_1(t) = \Omega_1^2(t) = \sin(t) + 1.5$, $p_2(t) = \Omega_2^2(t) = 0.5\cos(5t) + 3$.

The reference signal r is a unit step and the disturbances are set to

$f_1(t) = \cos(3t) + \sin(5t) + \cos(8t)$, $f_2(t) = \cos(t) + \sin(2t) + \cos(4t)$.

Define the inputs to the system.

```
t = 0:0.01:15;
u = ones(size(t));
d1 = cos(3*t)+sin(5*t)+cos(8*t);
d2 = cos(t)+sin(2*t)+cos(4*t);
```

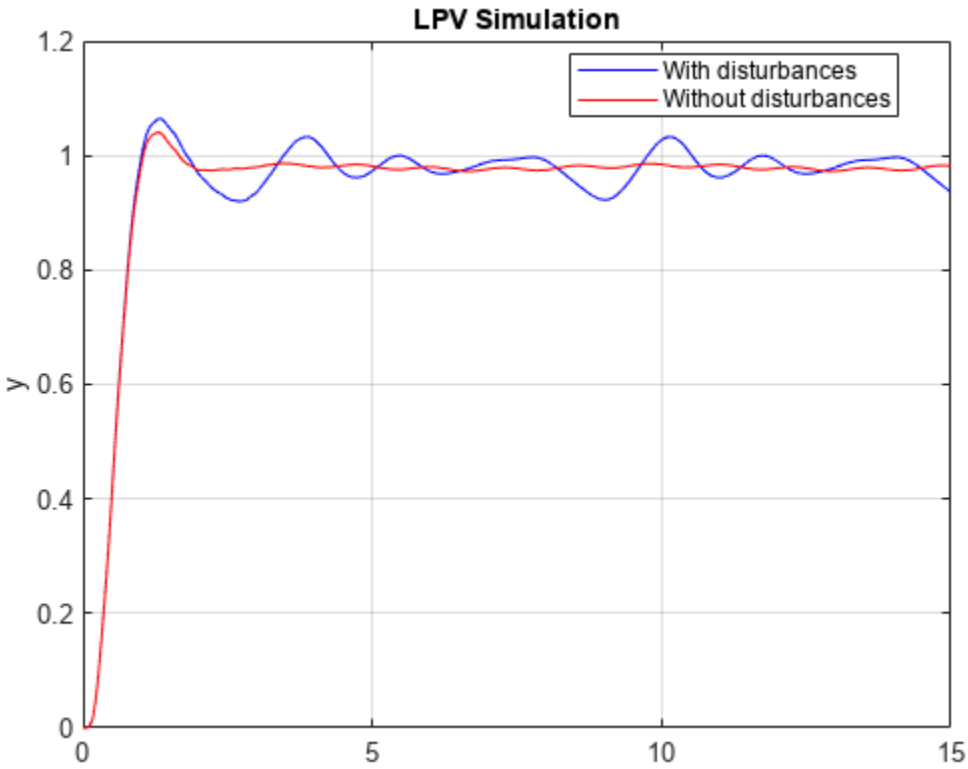
Define the trajectories of the parameters.

```
p1 = sin(t)+1.5;
p2 = 0.5*cos(5*t)+3;
```

Simulate the LPV model with and without disturbances.

```
[y1,t1,x1] = lsim(CL,[u;d1;d2],t,[],[p1;p2]);
[y2,t2,x2] = lsim(CL,[u;0*d1;0*d2],t,[],[p1;p2]);
```

```
figure
plot(t1,y1,'b',t2,y2,'r');
grid on;
title('LPV Simulation')
ylabel('y');
legend('With disturbances','Without disturbances','Location','Best')
```



The error due to the disturbances is on the order of 5% and is consists of 1-2 Hz oscillation about the disturbance-free response. This frequency range is incidentally where the LTI frequency response analysis indicated that disturbances would have the greatest impact on the output signal.

References

[1] F. Wu, "Control of Linear Parameter Varying Systems," Ph.D. dissertation, Department of Mechanical Engineering, University of California at Berkeley, CA, May 1995.

See Also

`lpvss` | `sample` | `ndgrid`

Related Examples

- “LTV and LPV Modeling” on page 1-26

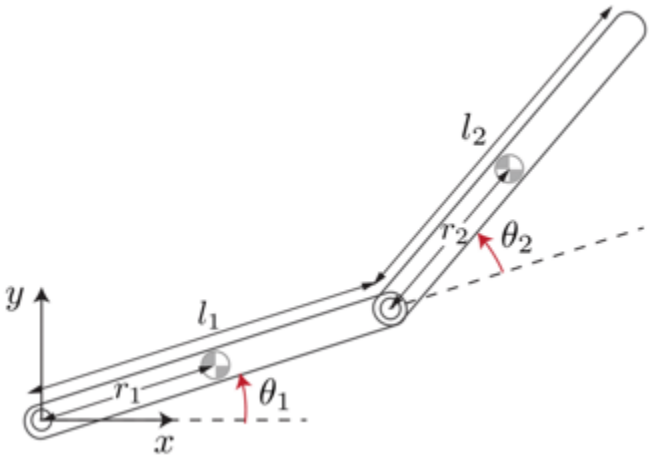
- “Using LTV and LPV Models in MATLAB and Simulink” on page 1-32
- “Approximate Nonlinear Aircraft Pitch Dynamics Using LPV Model” on page 1-68
- “LPV Model of Magnetic Levitation Model from Batch Linearization Results” on page 1-133
- “Control Design for Wind Turbine” on page 1-140

LTV Model of Two-Link Robot

This example shows how to obtain an LTV model of a two-link robot along a prescribed trajectory. The robot model and its corresponding dynamics are based on [1] on page 1-101 and [2] on page 1-101.

Nonlinear Model

This figure gives the schematics of the two-link planar robot.



The robot configuration is described by the joint angles θ_1, θ_2 and torques τ_1 and τ_2 applied at the joints to control the position of the tip. The state vector is

$$x(t) = \begin{bmatrix} \theta_1 \\ \theta_2 \\ \dot{\theta}_1(t) \\ \dot{\theta}_2(t) \end{bmatrix}.$$

The following equations describe the robot dynamics.

$$\begin{bmatrix} \alpha + 2\beta\cos(\theta_2) & \delta + \beta\cos(\theta_2) \\ \delta + \beta\cos(\theta_2) & \delta \end{bmatrix} \begin{bmatrix} \ddot{\theta}_1 \\ \ddot{\theta}_2 \end{bmatrix} + \begin{bmatrix} -\beta\sin(\theta_2)\dot{\theta}_2 & \beta\sin(\theta_2)\left(\dot{\theta}_1 + \dot{\theta}_2\right) \\ \beta\sin(\theta_2)\dot{\theta}_1 & 0 \end{bmatrix} \begin{bmatrix} \dot{\theta}_1 \\ \dot{\theta}_2 \end{bmatrix} = \begin{bmatrix} \tau_1 \\ \tau_2 \end{bmatrix}$$

Here:

$$\alpha = I_{z1} + I_{z2} + m_1 r_1^2 + m_2\left(l_1^2 + r_2^2\right)$$

$$\beta = m_2 l_1 r_2$$

$$\delta = I_{z2} + m_2 r_2^2$$

This example uses the `twoLinkInverseKinematics.m` script to solve this inverse kinematics problem.

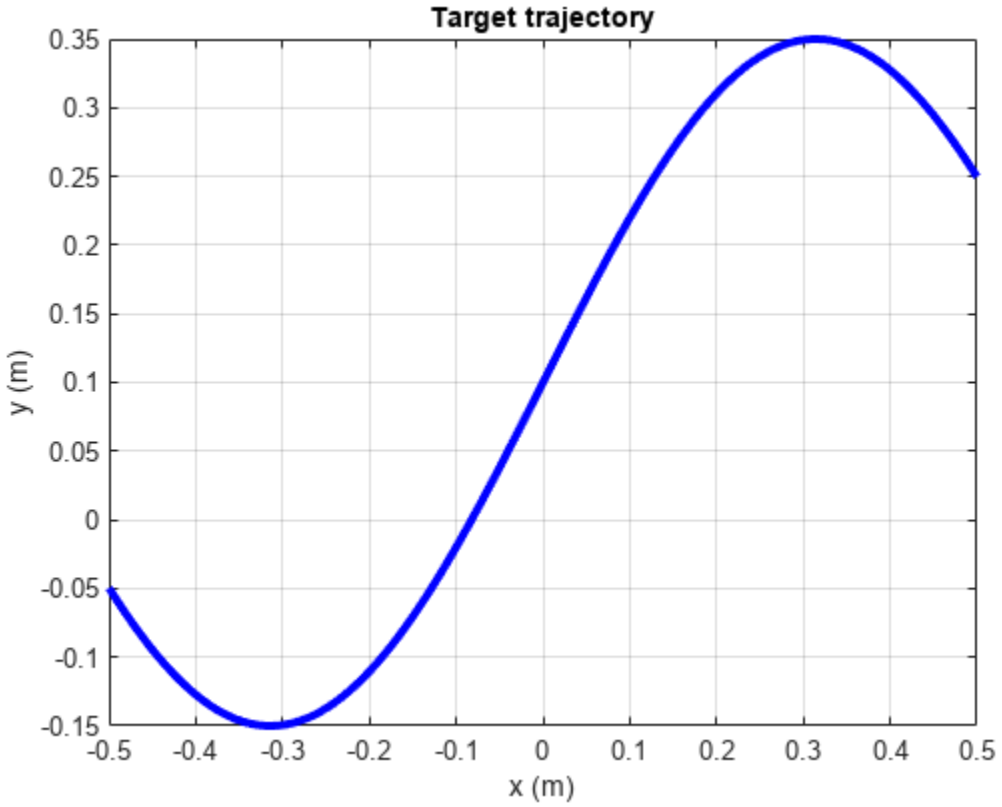
Define the physical parameters.

```
m1 = 3;         % Mass of link 1, kg
m2 = 2;         % Mass of link 2, kg
L1 = 0.3;       % Length of link 1, meters
L2 = 0.3;       % Length of link 2, meters
```

Desired Trajectory and Inverse Kinematics

The tip of the arm must follow a prescribed planar trajectory $X(t),Y(t)$.

```
T0 = 0;
Tf = 5;
tref = linspace(T0,Tf,400)';
load XYTrajectory x2ref y2ref
plot( x2ref, y2ref, 'b', 'LineWidth',3,'MarkerSize',10)
xlabel('x (m)');
ylabel('y (m)');
grid on
title('Target trajectory')
```

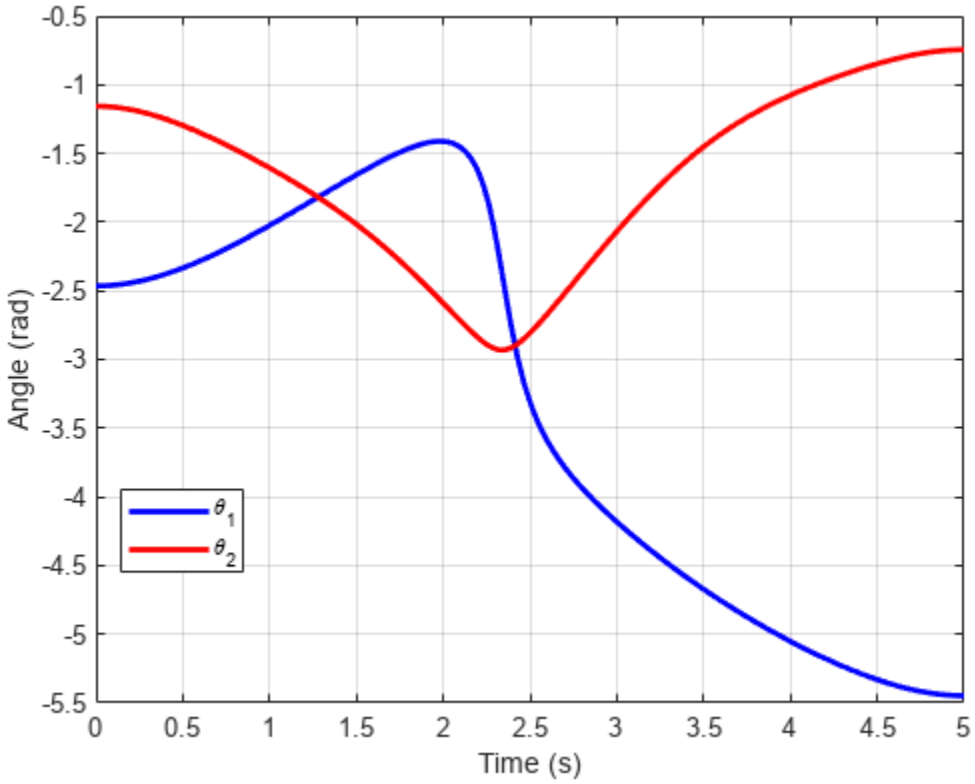


Solve the inverse kinematics problem for joint angles, joint velocities, and torque inputs corresponding to the motion of link 2 end point. Use joint angles to compute the trajectory of the tip of link 1.

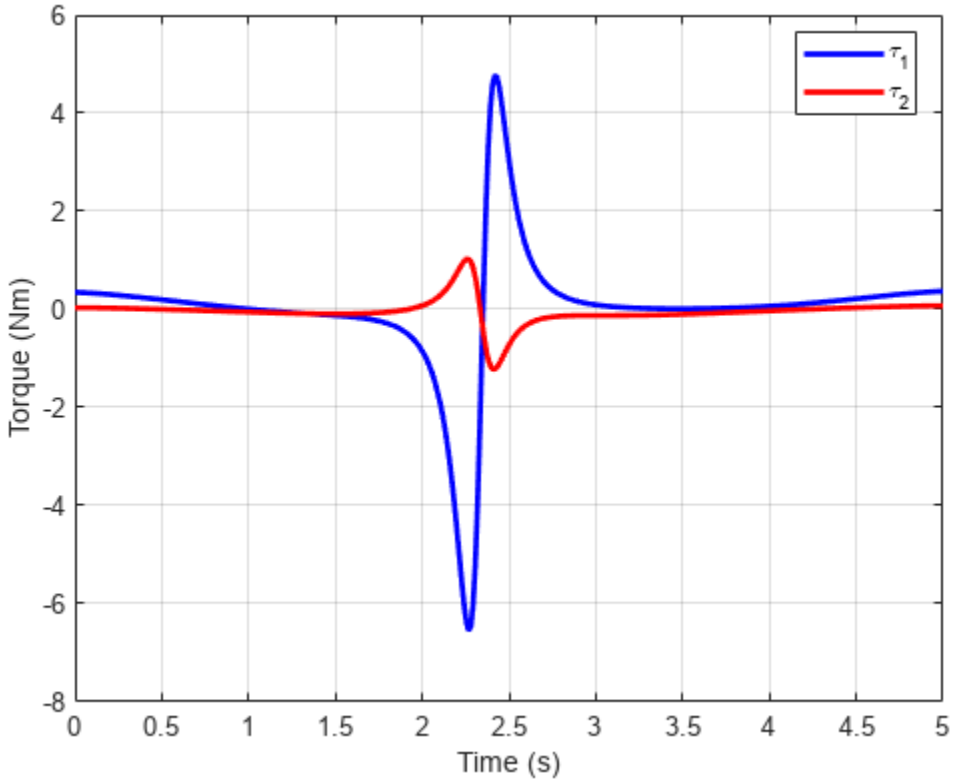
```
[x,tau] = twoLinkInverseKinematics(tref,x2ref,y2ref,m1,m2,L1,L2);
x1ref = L1*cos(x(:,1));
y1ref = L1*sin(x(:,1));
```

Plot the joint angles and torques.

```
plot(tref,x(:,1),'b',tref,x(:,2),'r','LineWidth',2);
legend('\theta_1','\theta_2','location','Best')
ylabel('Angle (rad)')
xlabel('Time (s)')
grid on;
```

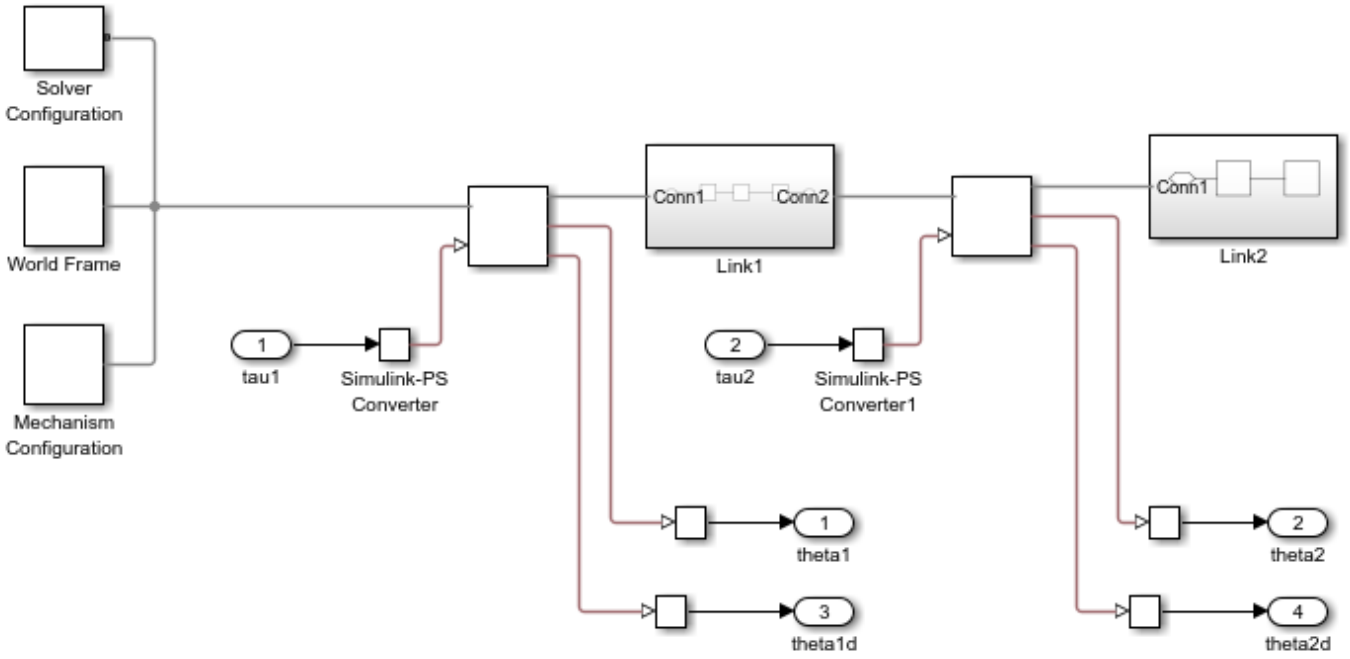



```
plot(tref,tau(:,1),'b',tref,tau(:,2),'r','LineWidth',2);
legend('\tau_1','\tau_2','location','Best')
ylabel('Torque (Nm)')
xlabel('Time (s)')
grid on;
```



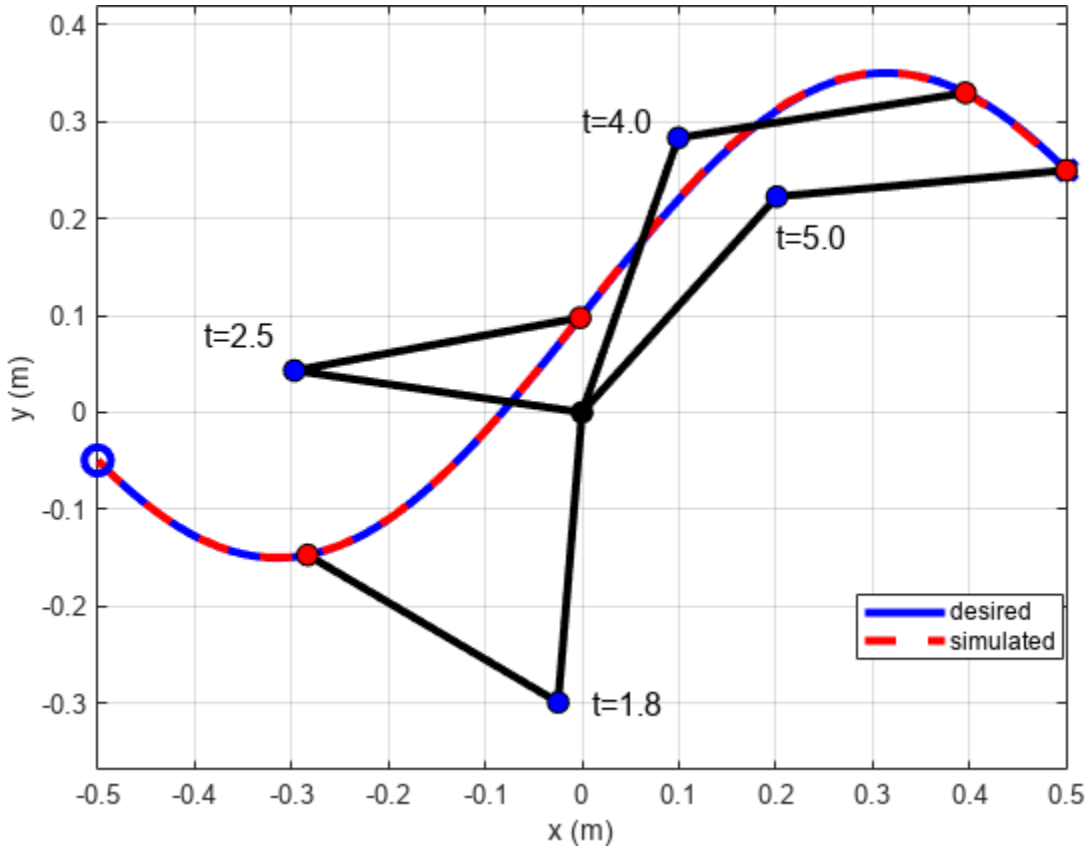
Use a Simscape Multibody model of the two-link robot to simulate the robot response to the computed torques.

```
tau1 = tau(:,1);
tau2 = tau(:,2);
xinit = x(1,:)';  % initial state

open_system('TwoLinkRobotSM')
```



Unpack the simulation results and verify that the tip follows the desired trajectory.

```
out = sim('TwoLinkRobotSM','ReturnWorkspaceOutputs', 'on');
tsim = out.xsim.time;
xsim = out.xsim.signals.values;

plotTrajectory(xsim,tref,x1ref,y1ref,x2ref,y2ref,L1,L2)
```



LTV Model

To build an LTV model of two-link robot along the desired trajectory, take linearization snapshots at 50 evenly distributed times between the start and end of the simulation. Use `linearize` to obtain linearized models at the corresponding operating conditions.

Specify linearization I/Os.

```
io = [...
   linio('TwoLinkRobotSM/tau1',1,'input');...
   linio('TwoLinkRobotSM/tau2',1,'input');...
   linio('TwoLinkRobotSM/Mux',1,'output')];
```

Take snapshot operating points.

```
tlin = linspace(T0,Tf,50);
op = findop('TwoLinkRobotSM',tlin);
```

Compute linearizations with offsets.

```
linOpt = linearizeOptions('StoreOffsets',true);
[G,~,info] = linearize('TwoLinkRobotSM',io,op,linOpt);
G.u = 'tau';
G.y = 'x';
G.SamplingGrid = struct('Time',tlin);
```

Use `ssInterpolant` to construct an LTV object from the array of linearized models and associated offsets. The resulting LTV model interpolates the linearized dynamics between the snapshot times `tlin`.

```
Gltv = ssInterpolant(G,info.Offsets);
Gltv.StateName
```

```
ans = 4x1 cell
    {'TwoLinkRobotSM.Subsystem.Revolute_Joint1.Rz.q'}
    {'TwoLinkRobotSM.Subsystem.Revolute_Joint1.Rz.w'}
    {'TwoLinkRobotSM.Subsystem.Revolute_Joint2.Rz.q'}
    {'TwoLinkRobotSM.Subsystem.Revolute_Joint2.Rz.w'}
```

The states are ordered differently in the linearization. Use `xperm` to align the state order with the convention θ_1, θ_2, $\dot{\theta}_1$, $\dot{\theta}_2$.

```
Gltv = xperm(Gltv,[1 3 2 4]);
Gltv.StateName
```

```
ans = 4x1 cell
    {'TwoLinkRobotSM.Subsystem.Revolute_Joint1.Rz.q'}
    {'TwoLinkRobotSM.Subsystem.Revolute_Joint2.Rz.q'}
    {'TwoLinkRobotSM.Subsystem.Revolute_Joint1.Rz.w'}
    {'TwoLinkRobotSM.Subsystem.Revolute_Joint2.Rz.w'}
```

LTV Model Validation

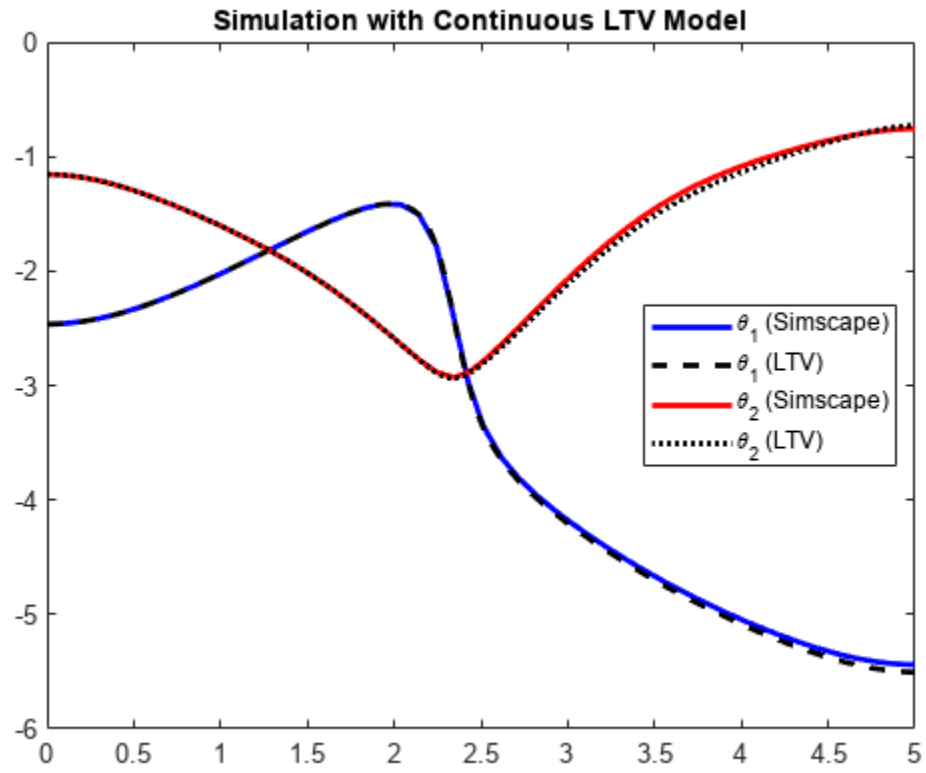
Simulate the response of this LTV model for the same torque inputs and compare with the results from Simscape Multibody .

Simulate LTV response to torque profiles.

```
xltv = lsim(Gltv,[tau1 tau2],tref,xinit);
```

Compare with Simscape results.

```
clf
plot(tsim,xsim(:,1),'b',tref,xltv(:,1),'k--',...
   tsim,xsim(:,2),'r',tref,xltv(:,2),'k:','LineWidth',2);
xlim([0 5])
legend('\theta_1 (Simscape)','\theta_1 (LTV)',...
   '\theta_2 (Simscape)','\theta_2 (LTV)','location','Best')
title('Simulation with Continuous LTV Model')
```



Discretization

In embedded applications, it is often desirable to have a discrete model that can be easily simulated. Using `c2d`, you can compute a Tustin discretization of `Gltv` and turn it into a discrete gridded LTV model.

```
Ts = 0.05;
Gd = c2d(Gltv,Ts,'tustin');
```

Turn Gd into a gridded LTV model with 50 grid points.

```
k = round(linspace(0,Tf/Ts,50));
Gd = ssInterpolant(Gd,struct('Time',k));
```

The initial condition for the Tustin discretization is

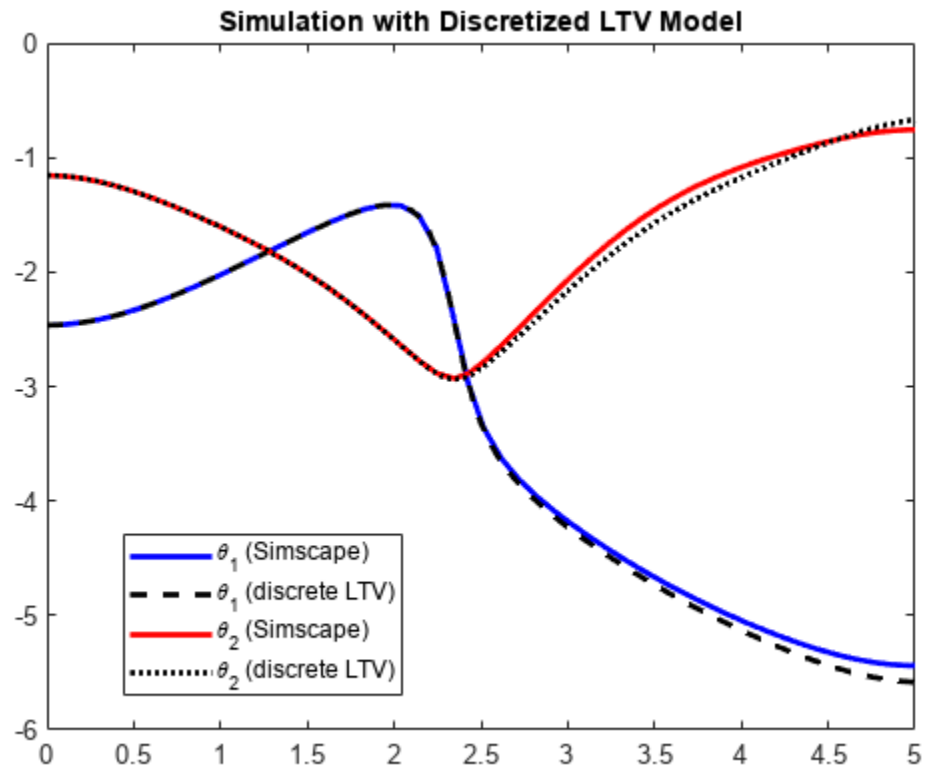
$z_0 = x(0) - \frac{T_s}{2}\dot{x}(0).$

```
[A,B,~,~,~,dx0,x0,u0] = Gltv.DataFunction(0);
dxinit = dx0+A*(xinit-x0)+B*(tau(1,:)'-u0);
zinit = xinit-Ts/2*dxinit;
```

Simulate and compare with Simscape results.

```
t = T0:Ts:Tf;
taud = interp1(tref,[tau1 tau2],t);
xd = lsim(Gd,taud,t,zinit);
```

```
clf
plot(tsim,xsim(:,1),'b',t,xd(:,1),'k--',...
   tsim,xsim(:,2),'r',t,xd(:,2),'k:','LineWidth',2);
xlim([0 5])
legend('\theta_1 (Simscape)','\theta_1 (discrete LTV)',...
   '\theta_2 (Simscape)','\theta_2 (discrete LTV)','location','Best')
title('Simulation with Discretized LTV Model')
```



References

[1] Murray, Richard M., Zexiang Li, and Shankar Sastry. *A Mathematical Introduction to Robotic Manipulation*. Boca Raton: CRC Press, 1994.

[2] Seiler, Peter, Robert M. Moore, Chris Meissen, Murat Arcak, and Andrew Packard. “Finite Horizon Robustness Analysis of LTV Systems Using Integral Quadratic Constraints.” *Automatica* 100 (February 2019): 135–43. https://doi.org/10.1016/j.automatica.2018.11.009.

See Also

`ltvss` | `ssInterpolant` | `sample` | `ndgrid` | `c2d` | `linearize`

Related Examples

- “LTV and LPV Modeling” on page 1-26
- “Using LTV and LPV Models in MATLAB and Simulink” on page 1-32
- “Approximate Nonlinear Aircraft Pitch Dynamics Using LPV Model” on page 1-68
- “LPV Model of Magnetic Levitation Model from Batch Linearization Results” on page 1-133
- “Control Design for Wind Turbine” on page 1-140

LPV Model of Bouncing Ball

This example shows how to construct a Linear Parameter Varying (LPV) representation of a system that exhibits multi-mode dynamics using `lpvss`.

The function `BouncingBallDF` defines the dynamics of this system. For more information on this model, see “Using LTI Arrays for Simulating Multi-Mode Dynamics” on page 2-89.

Define values of initial mass heights and lengths springs when uncompressed.

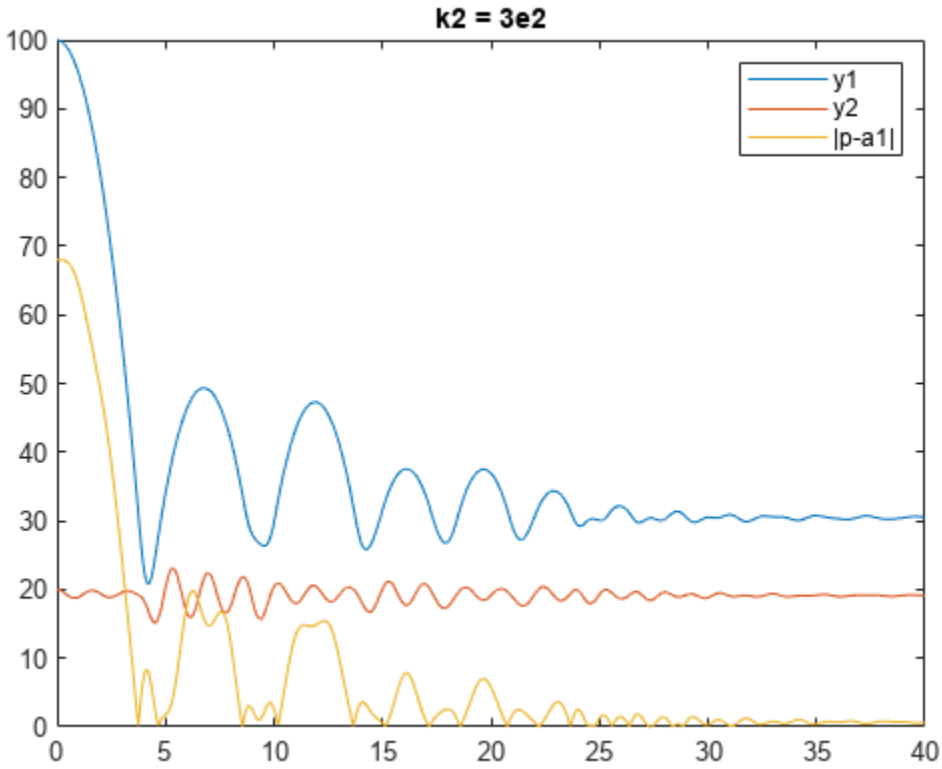
```
a1 = 12;    % uncompressed lengths of spring 1 (mm)
a2 = 20;    % uncompressed lengths of spring 2 (mm)
h1 = 100;   % initial height of mass m1 (mm)
h2 = a2;    % initial height of mass m2 (mm)
```

Moderate Floor Stiffness

First simulate the response for a soft surface. The state is $x = [y_1, \dot{y}_1, y_2, \dot{y}_2]$.

The input is the constant $u = 1$ (the forcing term is $F = mgu$).

```
k2 = 3e2;   % spring constant for second mass (g/s^2)
sys = lpvss("ygap",@(t,p) BouncingBallDF(t,p,k2));

t = 0:0.05:40;
pFcn = @(t,x,u) x(1)-x(3);
xinit = [h1;0;h2;0];
[y,~,~,~,p] = step(sys,t,pFcn,RespConfig('InitialState',xinit));

figure(1), plot(t,y(:,1),t,y(:,2),t,abs(p-a1))
legend('y1','y2','|p-a1|')
title('k2 = 3e2')
```

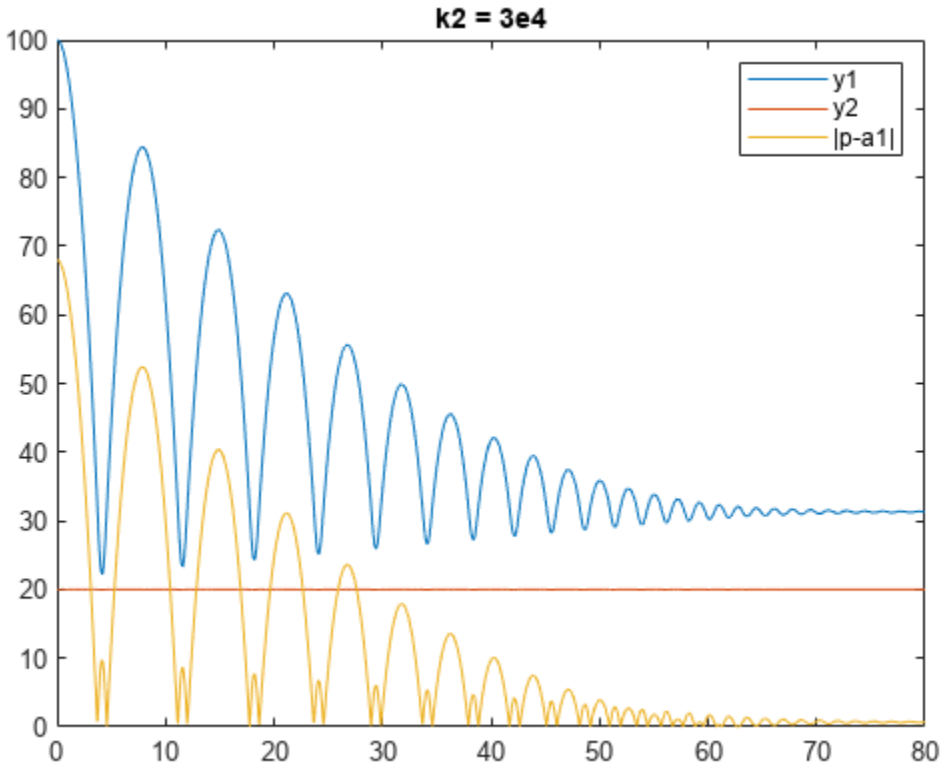


The curves show the position of the masses. At the start of simulation, mass 1 undergoes free fall until it hits mass 2. The collision causes the mass 2 to be displaced but it recoils quickly and bounces mass 1 back. The two masses are in contact for the time duration where $y_{gap} < a1$. When the masses settle down, their equilibrium values are determined by the static settling due to gravity.

High Floor Stiffness

Now simulate the response for a hard surface.

```
k2 = 3e4;   % spring constant for second mass (g/s^2)
sys = lpvss("ygap",@(t,p) BouncingBallDF(t,p,k2));

t = 0:0.05:80;
[y,~,~,~,p] = step(sys,t,pFcn,RespConfig('InitialState',xinit));

figure(2), plot(t,y(:,1),t,y(:,2),t,abs(p-a1))
legend('y1','y2','|p-a1|')
title('k2 = 3e4')
```



For high floor stiffness, the collision does not cause the Mass 2 to be displaced. It bounces the mass 1 back to a higher position and it takes more bounces for mass 1 to settle.

See Also

`lpvss` | `sample`

Related Examples

- “LTV and LPV Modeling” on page 1-26
- “Using LTV and LPV Models in MATLAB and Simulink” on page 1-32
- “LPV Model of Engine Throttle” on page 1-157
- “Analysis of Gain-Scheduled PI Controller” on page 1-113
- “Gain-Scheduled LQG Controller” on page 1-106
- “Hidden Couplings in Gain-Scheduled Control” on page 1-118
- “LPV Model of Magnetic Levitation System” on page 1-127

Gain-Scheduled LQG Controller

This example illustrates the loss of stablity that can arise in gain-scheduled control when the operating point changes too quickly. This corresponds to the example on page 102 of Reference [1] below. Additional details are given in Example 5.2.1. of Reference [2].

LPV Plant

The LPV plant is a second-order model that depends on the parameter p. The system has two lightly damped poles for frozen (constant) values of p. The plant equations are:

$$\begin{pmatrix}\dot{x}_1\\ \dot{x}_2\end{pmatrix} = \begin{pmatrix}0 & 1\\ -1-\frac{p}{2} & -0.2\end{pmatrix}\begin{pmatrix}x_1\\ x_2\end{pmatrix} + \begin{pmatrix}0\\ 1\end{pmatrix}u$$

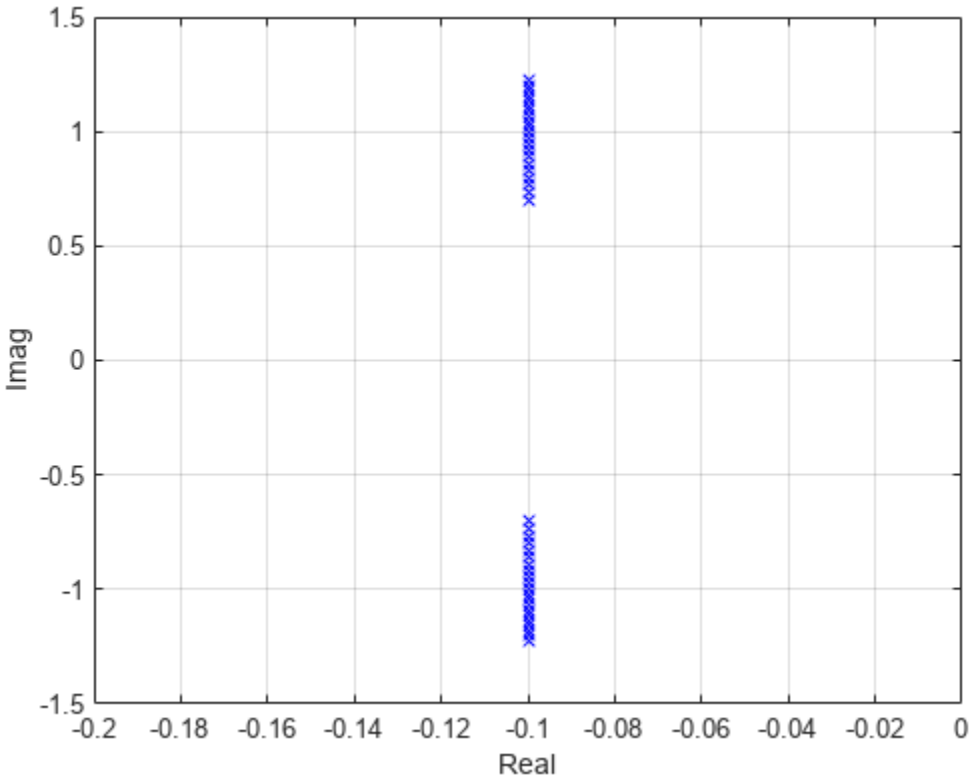
$$y = (1\ \ 0)\begin{pmatrix}x_1\\ x_2\end{pmatrix},$$

where the parameter p satisfies $|p| \le 1$.

```
G = lpvss('p',@GFCN);
```

Poles at "frozen" parameter values.

```
pval = -1:0.1:1;
Gs = sample(G,0,pval);
Gpole = pole(Gs);
Gpole = Gpole(:);

plot(real(Gpole),imag(Gpole),'bx');
axis([-0.2 0 -1.5 1.5]);
xlabel('Real');
ylabel('Imag')
grid on;
```



Gain-Scheduled LQG Controller with Integral Action

The gain-scheduled controller combines state feedback with a state observer and includes integral action. The observer and state feedback gains are designed using loop-transfer recovery. The scheduled controller (including integral action) corresponds to Ka in Figure 1 of [1].

Augment the plant with integrator at the input as shown in Figure 1 of [1]. The integrator will be implemented with the controller. The integral action ensures good low frequency tracking and perfect rejection of constant disturbances.

```
Gaug = G*tf(1,[1 0]);
```

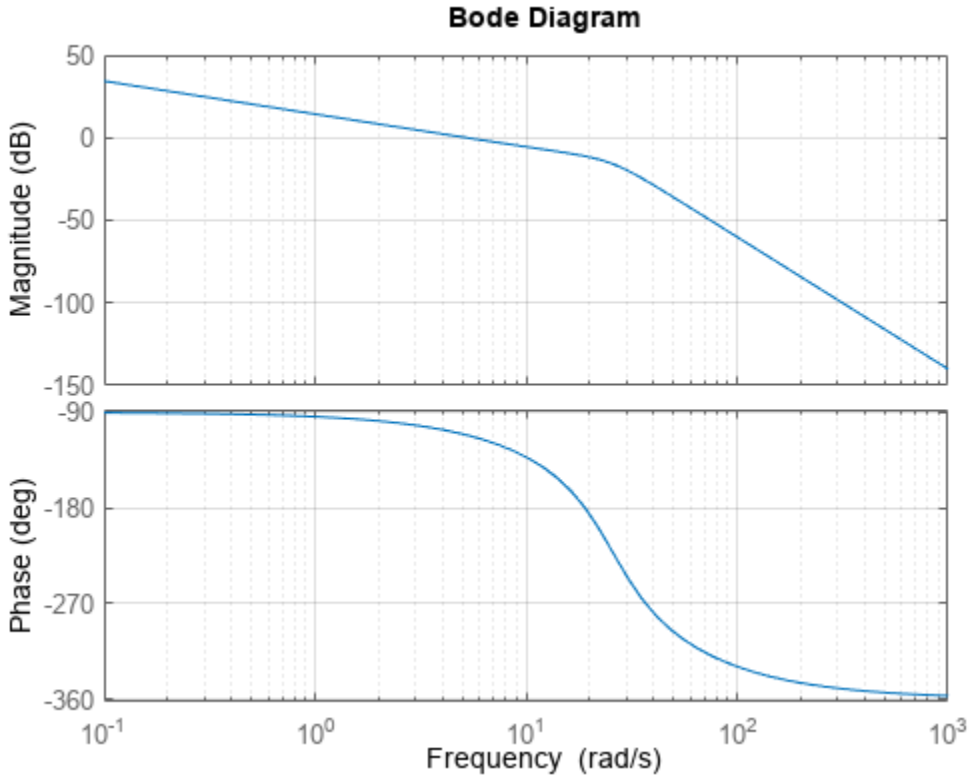
Construct the LPV controller using the function `KFCN`. For any parameter value , this function computes LQR and Kalman gains for the corresponding plant matrices.

```
Ka = lpvss('p', @(t,p) KFCN(t,p,Gaug));
```

LTI Analysis

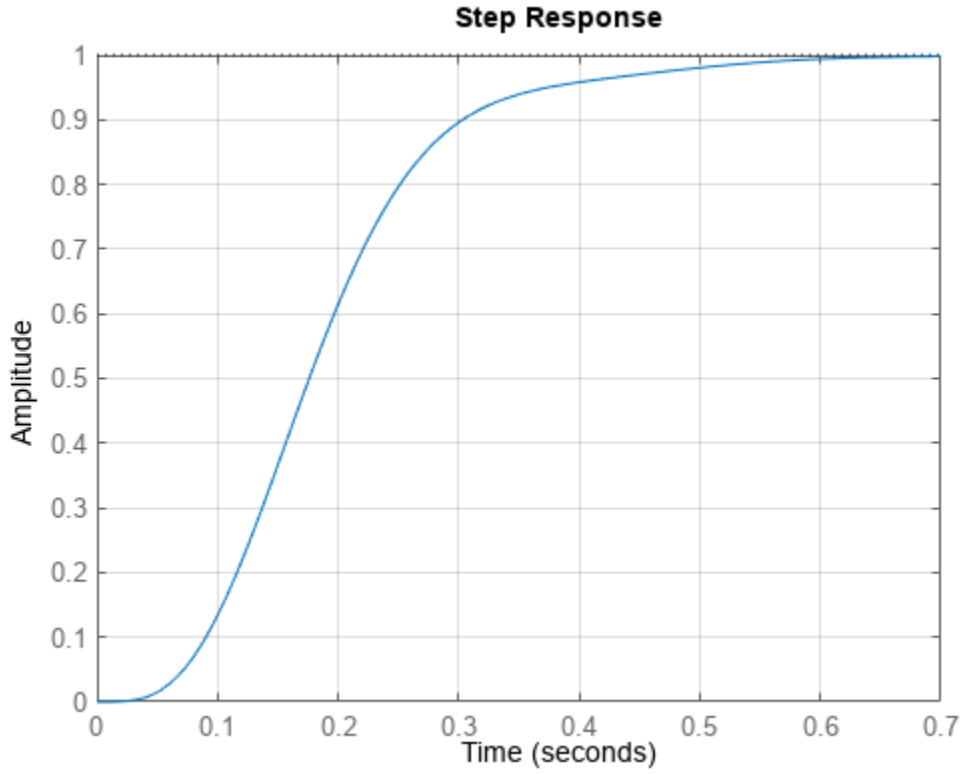
Plot the open-loop response for frozen parameter values. Note that the loops are the same at all parameter values. The scheduled controller cancels the lightly damped poles at frozen parameter values.

```
L = G*Ka;
Ls = sample(L,[],pval);

bode(Ls)
grid on;
```



The step responses for frozen parameter values show zero steady-state error, no overshoot, and settling time less than 1 second.

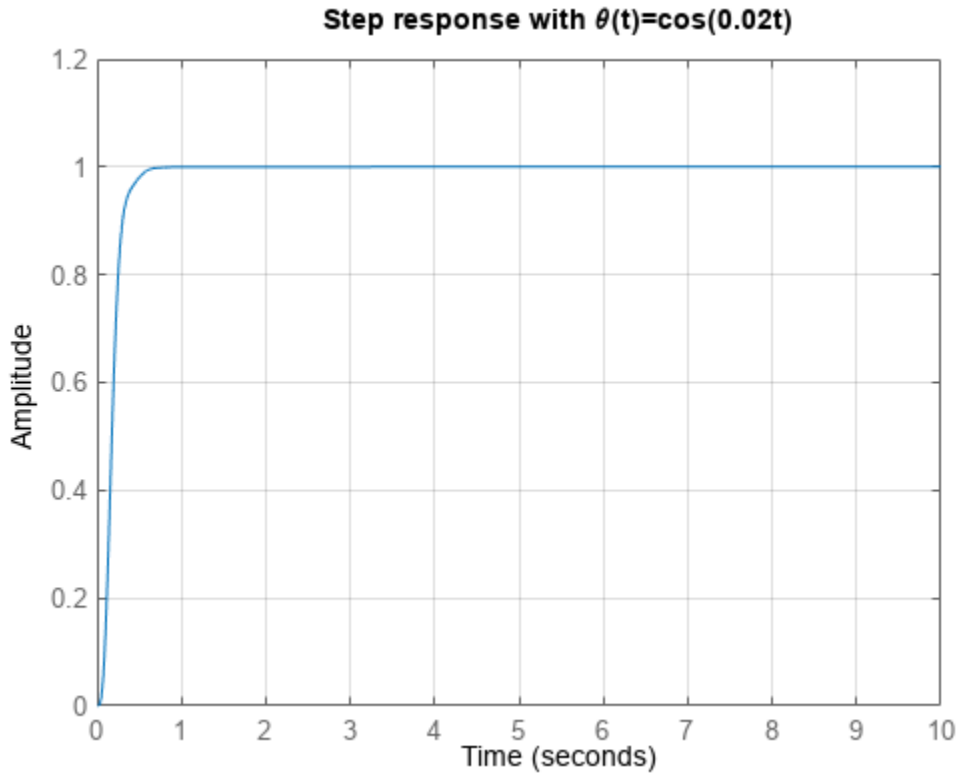
```
Ts = feedback(Ls,1);
step(Ts)
grid on;
```



LPV Analysis

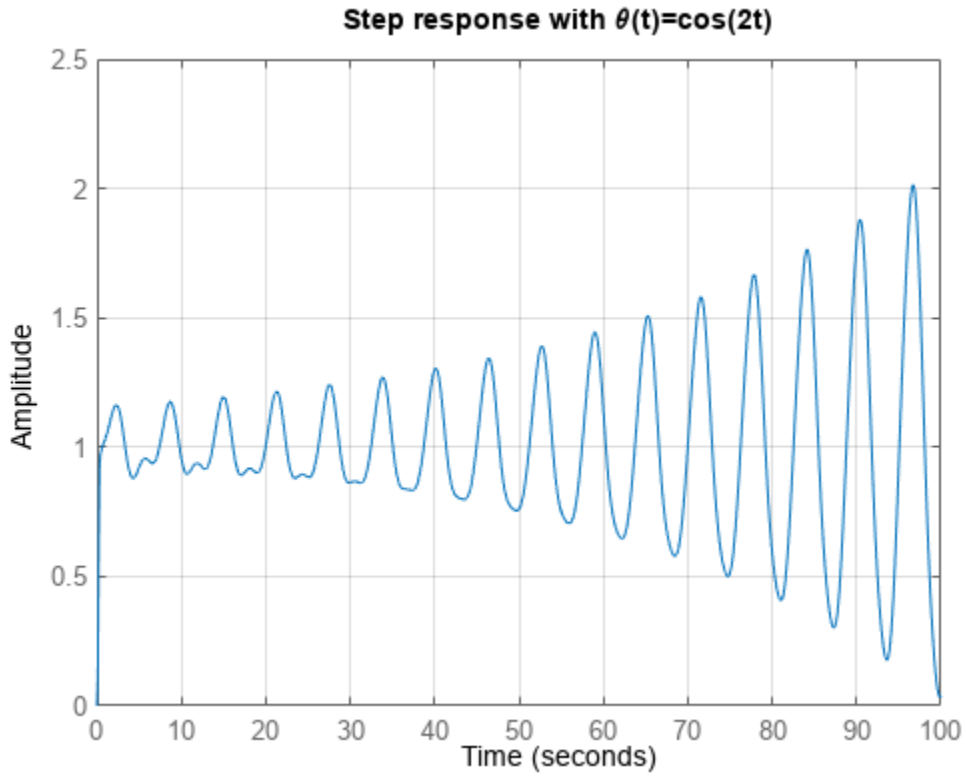
Use `feedback` to construct a closed-loop LPV model and plot the LPV step response for slowly-varying parameter $p(t)$. The response is similar to the LTI responses for frozen p.

```
T = feedback(L,1);

% Slow parameter variation
t = linspace(0,10,1000);
pt = cos(0.02*t);
step(T,t,pt)
title('Step response with \theta(t)=cos(0.02t)')
grid on
```



Next consider fast parameter variations $p(t) = \cos(2t)$. The step response now shows instability.

```
% Fast parameter variation
t = linspace(0,100,1000);
pt = cos(2*t);
step(T,t,pt);
title('Step response with \theta(t)=cos(2t)')
grid on
```



This plot corresponds to Figure 2 in reference [1]. Shamma and Athans make the comment in [1]: "It is emphasized that the resulting instability is **not** due to poor stability margins in the frozen parameter designs. Rather, the frozen parameter plant models do not reflect accurately the actual time-varying dynamics. Hence degradation in performance and even loss of stability is not surprising."

References

[1] Shamma and Athans, "Gain Scheduling: Potential Hazards and Possible Remedies," Control Systems Magazine, p. 101-107, 1992.

[2] Shamma, "Analysis and Design of Gain Scheduled Control Systems," Ph.D. Thesis, MIT, 1988.

See Also

`lpvss` | `sample`

Related Examples

- “LTV and LPV Modeling” on page 1-26
- “Using LTV and LPV Models in MATLAB and Simulink” on page 1-32
- “LPV Model of Engine Throttle” on page 1-157
- “Analysis of Gain-Scheduled PI Controller” on page 1-113
- “Hidden Couplings in Gain-Scheduled Control” on page 1-118

- “LPV Model of Magnetic Levitation System” on page 1-127

Analysis of Gain-Scheduled PI Controller

This example analyzes gain-scheduled PI control of a linear parameter-varying system. This example is based on [1] and [2].

The plant $G(\rho)$ is a first-order system with dynamics depending on the parameter ρ.

$$\dot{x} = -\frac{1}{\tau(\rho)}x + \frac{1}{\tau(\rho)}u$$

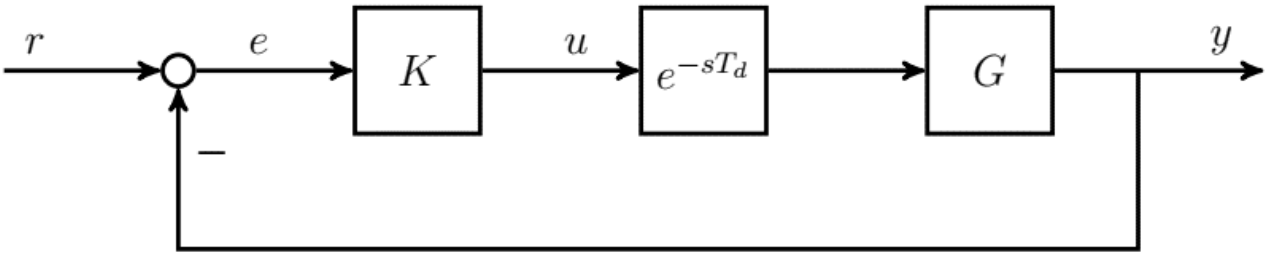
$$y = c(\rho)x$$

Here:

$$\tau(\rho) = \sqrt{133.6 - 16.8\rho}$$

$c(\rho) = \sqrt{4.8\rho - 8.6}$.

The parameter ρ is restricted to the interval [2,7]. The closed-loop system consists of the plant $G(\rho)$, gain-scheduled PI controller $K(\rho)$, and a time delay $T_d = 0.5$ sec, as shown in this interconnection figure.



LPV Plant

Use `lpvss` to construct a model of the LPV plant. The function `plantFcnGSPI` returns the state-space matrices and offsets as a function of time `t` and parameter `rho`. To see the code for this function, open the file `plantFcnGSPI.m` or enter `type plantFcnGSPI` at the command line.

```
G = lpvss('rho',@plantFcnGSPI);
```

Gain-Scheduled PI Controller

The gain-scheduled Proportional-Integral (PI) controller $K(\rho)$ is designed to achieve a closed loop damping ratio $\zeta_d = 0.7$ and natural frequency $\omega_d = 0.25$ in the domain $\rho \in [2, 7]$. You can construct this controller in two different forms.

Controller K_1, with the intergral gain at the integrator input:

$$\dot{x}_c = K_i(\rho)e$$

$u = x_c + K_p(\rho)e$,

Controller K_2, with the intergral gain at the integrator output:

$$\dot{x}_c = e$$

$u = K_i(\rho)x_c + K_p(\rho)e.$

These forms are equivalent when ρ and K_i are constant. However, the two forms have different input/output response when the parameter varies in time. This example compares these two options in terms of closed-loop performance.

This example provides the code for these two controllers in the files `k1FcnGSPI.m` and `k2FcnGSPI.m`. Use `lpvss` to construct the two controllers.

```
K1 = lpvss('rho',@k1FcnGSPI);
K2 = lpvss('rho',@k2FcnGSPI);
```

Closed-Loop Models

Construct the closed-loop LPV models as shown in the interconnection figure with controllers K_1 and K_2.

Use a second-order Pade approximation of the delay.

```
Td = 0.5;
[n,d] = pade(Td,2);
H = tf(n,d);
```

Construct the closed-loop models.

```
CL1 = feedback(G*H*K1,1);
CL2 = feedback(G*H*K2,1);
```

LTI Analysis

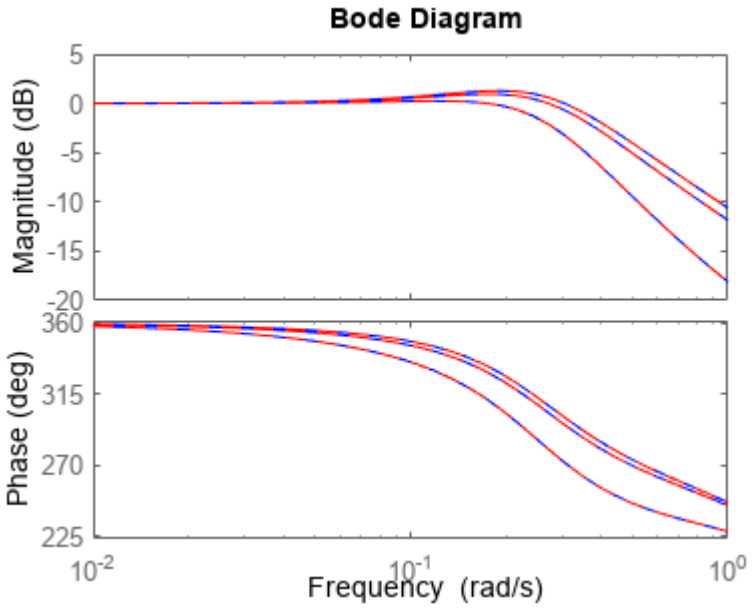
Evaluate this closed-loop response for several frozen values of ρ.

Sample LPV system for three values of rho.

```
Nvals = 3;
pvals = linspace(2,7,Nvals);
CL1vals = sample(CL1,[],pvals);
CL2vals = sample(CL2,[],pvals);
```

Plot Bode response of complementary sensitivity at fixed values of ρ.

```
bode(CL1vals,'b',CL2vals,'r--',{1e-2,1});
```



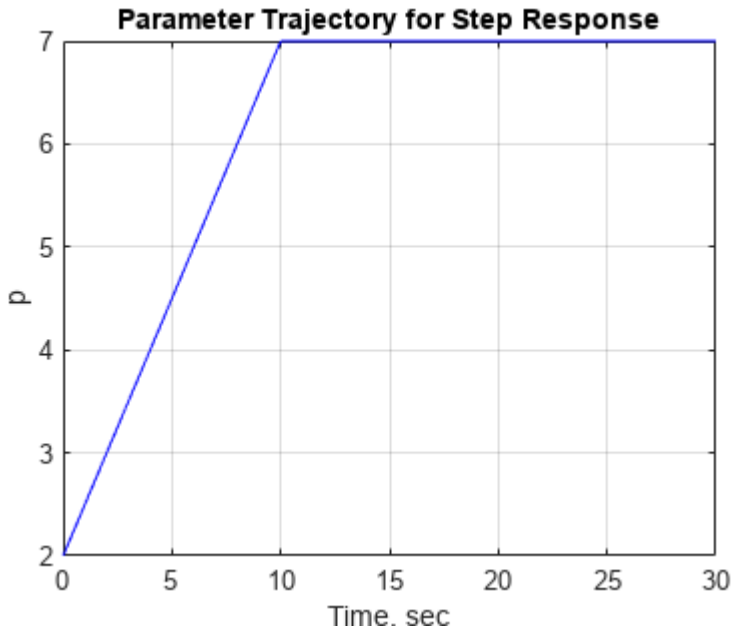
As expected, the location of the integral gain does not affect the controller when ρ is constant. Hence, the two closed-loop responses are identical in the LTI cases.

LPV Analysis

Generate closed-loop step responses from r to y along one specific parameter trajectory.

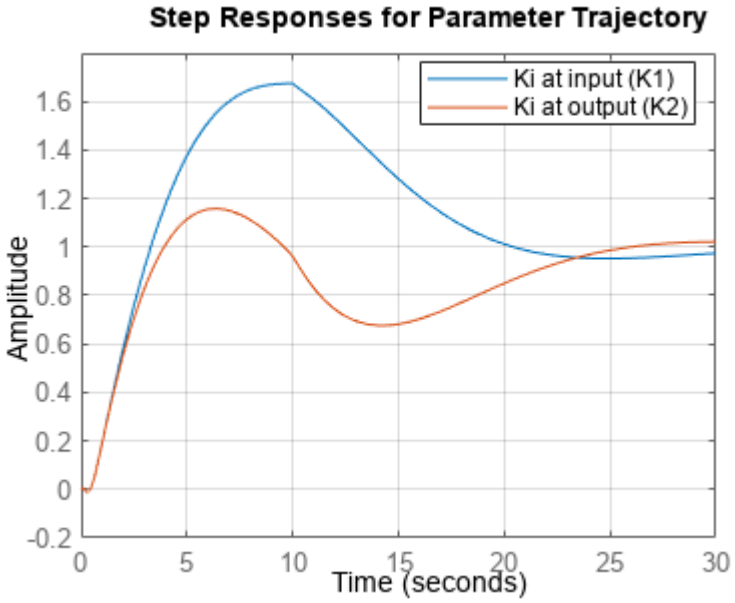
Specify the parameter trajectory.

```
t = linspace(0,30,1e3);
rho = interp1([0 10 30],[2 7 7],t);
plot(t,rho,'b');
grid on
xlabel('Time, sec');
ylabel('p');
title('Parameter Trajectory for Step Response');
```



Plot the step responses.

```
step(CL1,CL2,t,rho)
grid on
title('Step Responses for Parameter Trajectory');
legend('Ki at input (K1)','Ki at output (K2)','Location','Best');
```



The two closed-loop simulations are different. This indicates that the placement of the integrator in a gain-scheduled control makes a difference due to the parameter variations.

References

1 Tan, Shaohua, Chang-Chieh Hang, and Joo-Siong Chai. “Gain Scheduling: From Conventional to Neuro-Fuzzy.” *Automatica* 33, no. 3 (March 1997): 411–19. https://doi.org/10.1016/S0005-1098(96)00162-8.

2 Pfifer, Harald, and Peter Seiler. “Robustness Analysis of Linear Parameter Varying Systems Using Integral Quadratic Constraints.” *International Journal of Robust and Nonlinear Control* 25, no. 15 (October 2015): 2843–64. https://doi.org/10.1002/rnc.3240.

See Also

`lpvss` | `sample`

Related Examples

- “LTV and LPV Modeling” on page 1-26
- “Using LTV and LPV Models in MATLAB and Simulink” on page 1-32
- “LPV Model of Engine Throttle” on page 1-157
- “Gain-Scheduled LQG Controller” on page 1-106
- “Hidden Couplings in Gain-Scheduled Control” on page 1-118
- “LPV Model of Magnetic Levitation System” on page 1-127

Hidden Couplings in Gain-Scheduled Control

This example illustrates the *hidden couplings* that can arise in gain-scheduled control depending on controller configuration. This corresponds to examples 2, 8, 9, and 11 in the reference [1].

LPV Plant

The following equations define the nonlinear plant.

$$\frac{dx}{dt} = -x + u$$

$y = \tanh(x)$.

There is a collection of equilibrium points parameterized by $p = y \in [-1, 1]$.

$$(x_0(p), u_0(p), y_0(p)) = \left(\tanh^{-1}(p), \tanh^{-1}(p), p\right)$$

Linearization around these equilibrium points yields the LPV model.

$$\dot{x} = -(x - x_0(p)) + (u - u_0(p))$$

$$y = y_0(p) + \left(1 - p^2\right)(x - x_0(p))$$

Use `lpvss` to construct this LPV plant. The data function is defined in the file `lpvHCModel.m`. Since $\tanh^{-1}(p)$ is infinite for $|p| = 1$, clip p to the range $[-p_{\max}, p_{\max}]$ to stay away from singularity.

```
pmax = 0.99;
G = lpvss('p',@(t,p) lpvHCModel(t,p,pmax),'StateName','x');
```

Gain-Scheduled PI Controller

The gain-scheduled Proportional-Integral (PI) controller $K(p)$ is designed to achieve a closed loop damping ratio $\zeta_d = 2/3$ and natural frequency $\omega_d = 1$ at each point in the domain. You can construct this controller in two different forms.

Controller K_{in}, with the intergral gain at the integrator input:

$$\dot{x}_c = K_i(p)e$$

$u = x_c + K_p(p)e$,

Controller K_{out}, with the intergral gain at the integrator output:

$$\dot{x}_c = e$$

$u = K_i(p)x_c + K_p(p)e$.

These forms are equivalent when p is constant. However, the two forms have different input/output response when the parameter varies in time. This example compares these two options in terms of closed-loop performance.

This example provides the code for these two controllers in the files `hcKinFCN.m` and `hcKoutFCN.m`. Use `lpvss` to construct the two controllers.

```
Kin = lpvss('p',@(t,p) hcKinFCN(t,p,pmax));
Kout = lpvss('p',@(t,p) hcKoutFCN(t,p,pmax));
```

Use `feedback` to construct LPV models of the closed-loop systems for each controller.

```
CLin = feedback(G*Kin,1);
CLout = feedback(G*Kout,1);
```

LTI Analysis

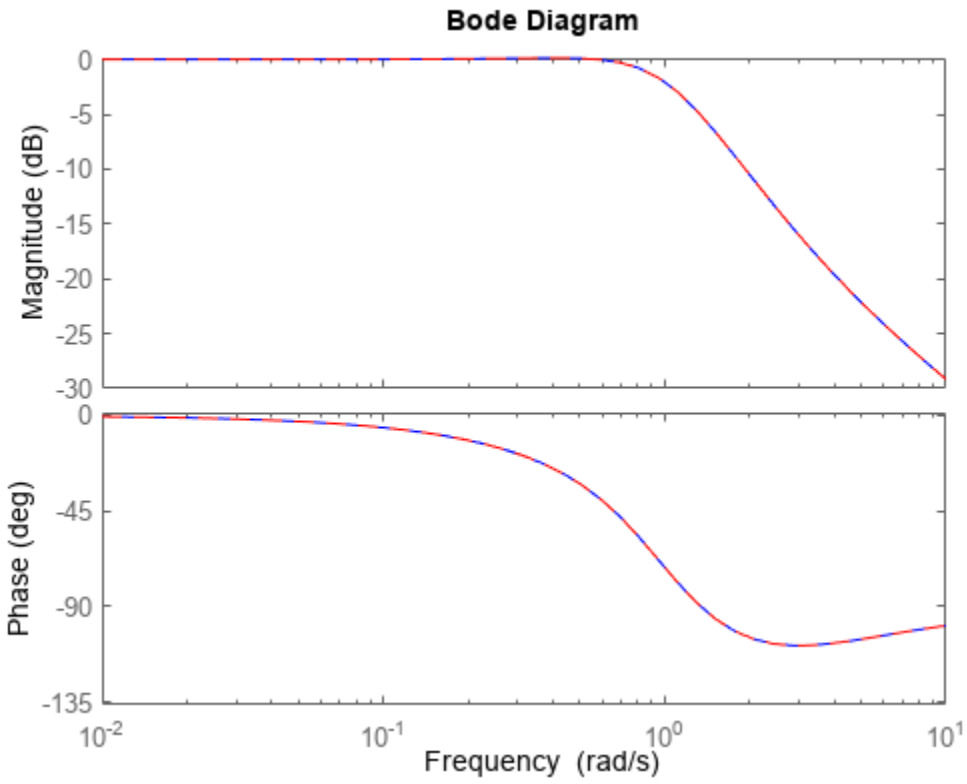
Evaluate this closed-loop response for several fixed values of *p*.

Sample LPV system for three values of rho.

```
Nvals = 5;
pvals = linspace(-0.9,0.9,Nvals);
CLinvals = sample(CLin,[],pvals);
CLoutvals = sample(CLout,[],pvals);
```

Plot the Bode response for fixed values of *p*.

```
bode(CLinvals,'b',CLoutvals,'r--',{1e-2,1e1});
```



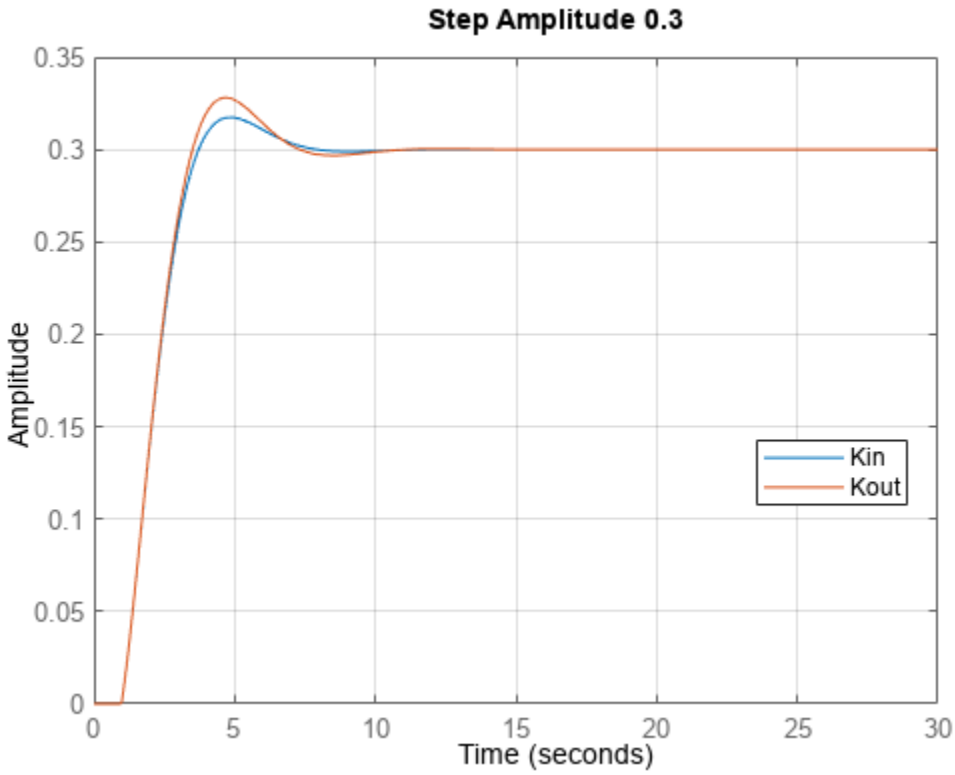
The location of the integral gain does not affect the controller when *p* is constant. Hence, the two feedback loops have identical responses.

LPV Simulation

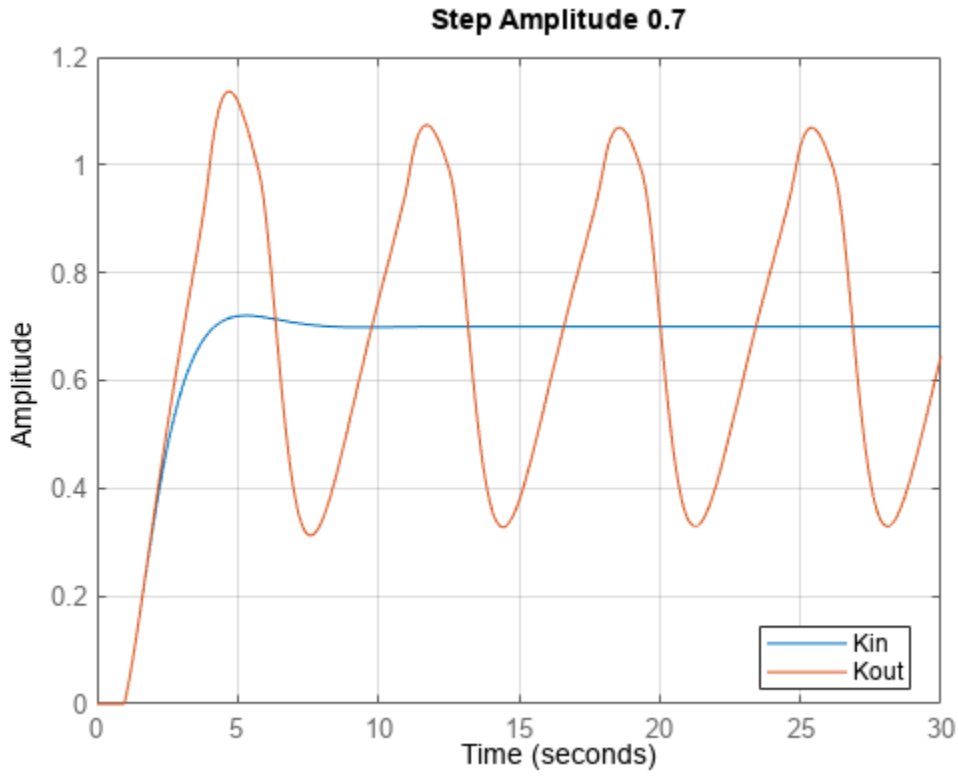
Compute the step response of the closed-loop LPV models for the two step amplitudes `rstep` = 0.3 and `rstep` = 0.7. This approximates the tracking performance of each gain-scheduled controller against the nonlinear plant. Both controllers use $p = y$ as the scheduling variable.

The article [1] indicates that the feedback loop is locally unstable $p > 0.725$ with `Kout` while it remains locally stable for all $p < 1$ with `Kin`. Try reproducing these results with an LPV simulation.

```
Tf = 30;
tsim = linspace(0,Tf,1000);
xinit = [0;0];
rstep  = 0.3;
Config = RespConfig('Amplitude',rstep,'Delay',1,'InitialState',xinit);
pFcn = @(t,x,u) tanh(x(1));
step(CLin,CLout,tsim,pFcn,Config), grid
title('Step Amplitude 0.3')
legend('Kin','Kout','location','Best')
```




```
rstep  = 0.7;
Config = RespConfig('Amplitude',rstep,'Delay',1,'InitialState',xinit);
step(CLin,CLout,tsim,pFcn,Config), grid
title('Step Amplitude 0.7')
legend('Kin','Kout','location','Best')
```

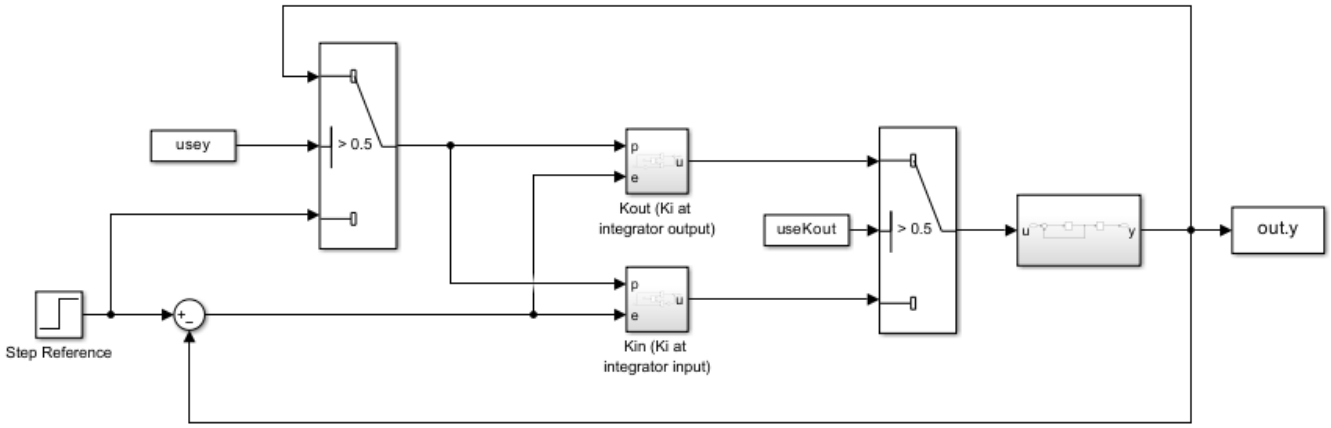


The LPV simulation confirms that the feedback loop remains stable with `Kin` (integral gain at the integrator input) but exhibits a limit cycle with `Kout` for large setpoint changes. This is only an approximation of the true nonlinear behavior since the output *y* should never exceed 1. The LPV plant model is locally linear and cannot capture the saturating behavior of the function tanh.

To further assess the fidelity of the LPV model, compare with a nonlinear simulation using the true nonlinear plant.

Open Simulink model of closed-loop system.

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

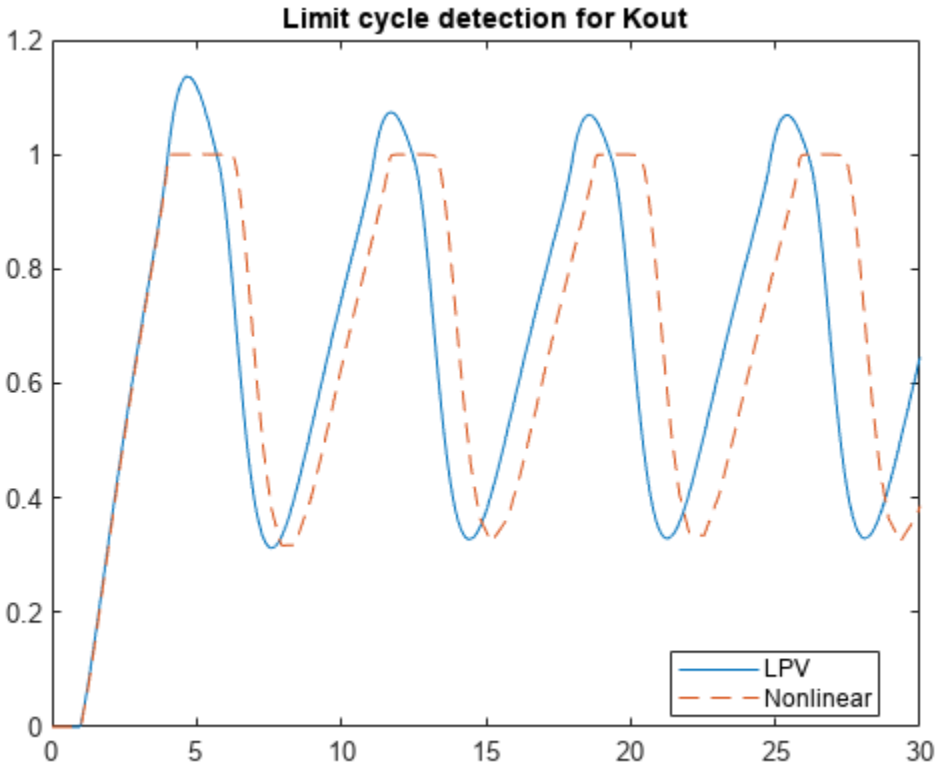


Simulate response with `Ki` at integrator output.

```
usey = 1; useKout = 1;
simout = sim('HiddenCouplings',Tf);
```

Compare the LPV and nonlinear responses with the controller `Kout`.

```
y = step(CLout,tsim,pFcn,Config);
plot(tsim,y,simout.tout,simout.y,'--')
title('Limit cycle detection for Kout')
legend('LPV','Nonlinear','location','Best')
```

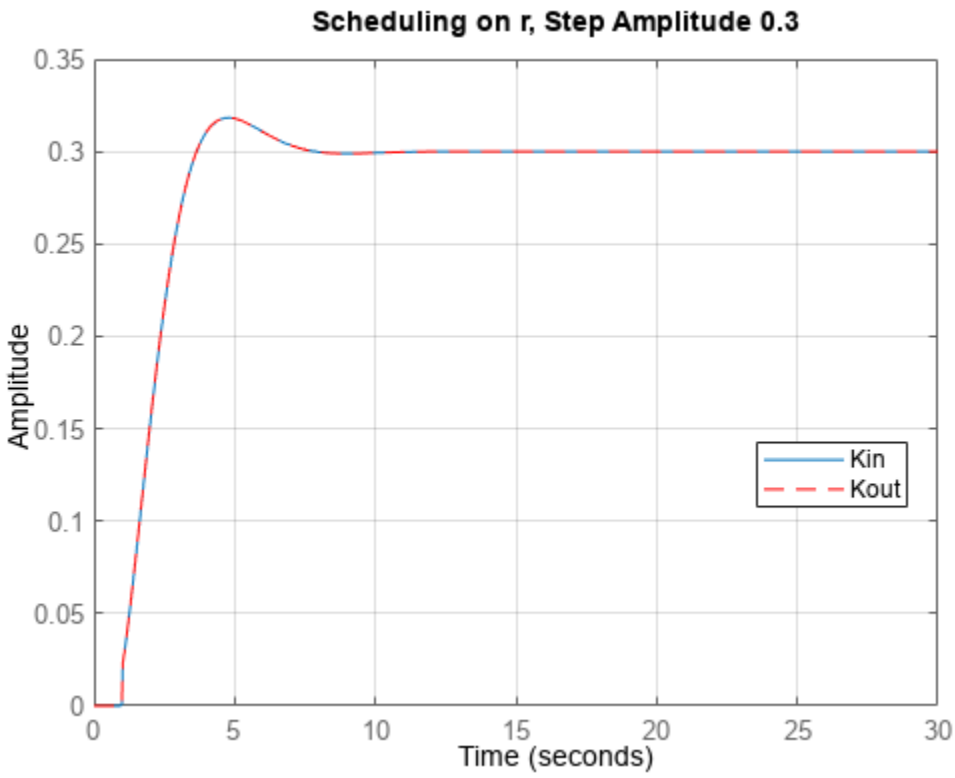


As expected, results don't match exactly due to the strong nonlinearity of tanh. Still, the LPV simulation correctly predicts the limit cycle with approximately the right period.

Scheduling on Reference Signal

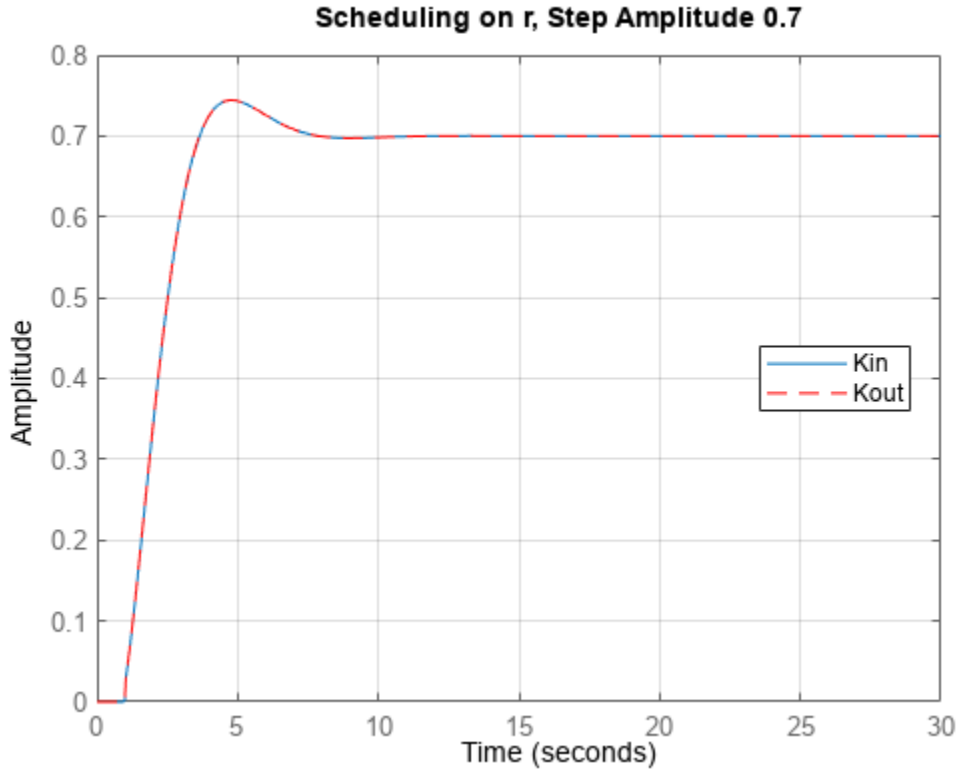
Scheduling on $y = \tanh(x)$ creates an additional feedback loop between the plant and controller via the scheduling function. In other words, the closed-loop model simulated by `step` is quasi-LPV and the state-dependent scheduling causes instability. Since the output y is meant to track the reference signal r, an alternative strategy is to schedule the controller on , which is a truly exogenous signal. You can easily compare the two strategies by replace the parameter trajectory $p = \tanh(x)$ with $p = r$ in the `step` command.

```
rstep  = 0.3;
Config = RespConfig('Amplitude',rstep,'Delay',1,'InitialState',xinit);
r = rstep*(tsim>=1);
step(CLin,CLout,'r--',tsim,r,Config), grid
title('Scheduling on r, Step Amplitude 0.3')
legend('Kin','Kout','location','Best')
```



Simulate with a larger step amplitude.

```
rstep  = 0.7;
Config = RespConfig('Amplitude',rstep,'Delay',1,'InitialState',xinit);
step(CLin,CLout,'r--',tsim,r,Config), grid
title('Scheduling on r, Step Amplitude 0.7')
legend('Kin','Kout','location','Best')
```



Both controllers now remain stable and have identical performance. This is because p is piecewise constant, which is close enough to the LTI case when both controllers are interchangeable. Save the results with `Kout` for comparison with the hybrid scheme discussed next.

```
y = step(CLout,tsim,r,Config);
```

Hybrid approach

You can also use a hybrid approach where the LPV plant depends on $p = y$ and the PI controller is scheduled in r. To try this strategy, construct the LPV controller with a different parameter called `pc`:

```
Kout = lpvss('pc',@(t,p) hcKoutFCN(t,p,pmax));
CLout = feedback(G*Kout,1);
```

The closed-loop model now depends on two parameters `p` and `pc`. Set `p` to y and `pc` to r and simulate the response.

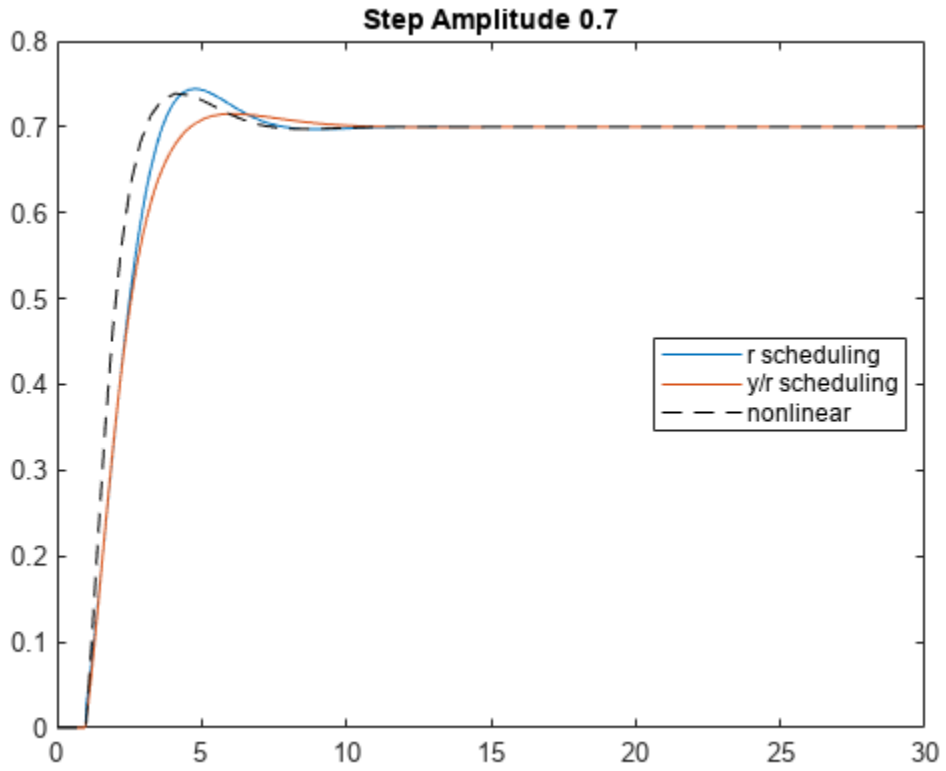
```
pFcn = @(t,x,u) [tanh(x(1)) ; interp1(tsim,r,t)];
yh = step(CLout,tsim,pFcn,Config);
```

Finally, run a nonlinear simulation with the true plant and a PI controller scheduled on r with K_i at the integrator output.

```
usey = 0; useKout = 1;
simout = sim('HiddenCouplings',Tf);
```

Compare the r-scheduling and hybrid scheduling schemes with the true nonlinear response.

```
plot(tsim,y,tsim,yh,simout.tout,simout.y,'k--')
title('Step Amplitude 0.7')
legend('r scheduling','y/r scheduling','nonlinear','location','Best')
```



Here scheduling both the plant and controller on *r* yields a better approximation of the nonlinear response.

References

1 Rugh, Wilson J., and Jeff S. Shamma. “Research on Gain Scheduling.” *Automatica* 36, no. 10 (October 2000): 1401–25. https://doi.org/10.1016/S0005-1098(00)00058-3.

See Also

`lpvss` | `sample`

Related Examples

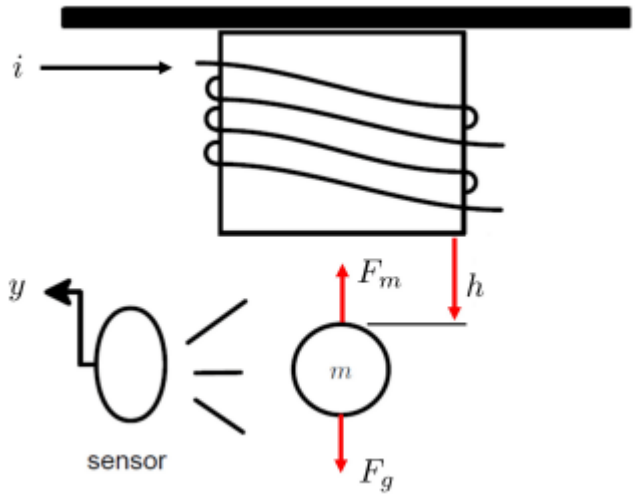
- “LTV and LPV Modeling” on page 1-26
- “Using LTV and LPV Models in MATLAB and Simulink” on page 1-32
- “LPV Model of Engine Throttle” on page 1-157
- “Analysis of Gain-Scheduled PI Controller” on page 1-113
- “Gain-Scheduled LQG Controller” on page 1-106
- “LPV Model of Magnetic Levitation System” on page 1-127

LPV Model of Magnetic Levitation System

This example uses an analytic LPV model of a magnetic levitation system to control the height of the ball. In this example, you build the LPV plant model directly from the linearized equations of motion.

Nonlinear System

This figure shows the magnetic ball levitation device and its key components. A current i (A) is supplied to a coil which creates a magnetic force on the ball. The position of the ball is denoted by h (m). An infrared sensor measures the position of the ball y (V). The objective is to make the ball levitate at a desired position $\overline{h}$. There are two key forces on the ball: the gravity pull F_g and a magnetic force F_m.



This nonlinear equation defines the dynamics of the model.

$$m_b\ddot{h} = m_b g - \alpha\frac{i(t)^2}{h(t)^2}$$

Here:

- m_b is the mass of the ball (kg).
- g is the gravitational acceleration (m/s^2).
- α is the magnetic force constant (N*m/(A^2)).

Define the model parameters.

```
mb = 0.02;
g = 9.81;
alpha = 2.4832e-5;
```

Linearized Equations of Motion

Linearize the nonlinear equation around equilibrium heights $p = \overline{h}$ to obtain the LPV model

$$\dot{x} = A(p)(x - x_0(p)) + B(p)(u - u_0(p))$$

$$y = y_0(p) + C(x - x_0(p)),$$

where:

$$x = \begin{pmatrix} h \\ \dot{h} \end{pmatrix},\ A = \begin{pmatrix} 0 & 1 \\ a_0(p) & 0 \end{pmatrix},\ B = \begin{pmatrix} 0 \\ b_0(p) \end{pmatrix},\ C = (1\ \ 0),$$

and

$$x_0(p) = \begin{pmatrix} p \\ 0 \end{pmatrix},\ u_0(p) = \bar{i} = \sqrt{\frac{m_b g}{\alpha}}p,\ y_0(p) = p,\ a_0(p) = \frac{2g}{p},\ b_0(p) = -2\frac{\sqrt{g\alpha/m_b}}{p}.$$

The matrices and offsets of this system are defined in the function `dataFcnMaglev.m`.

Use `lpvss` to create the model.

```
Glpv = lpvss('h',@dataFcnMaglev);
```

Gain-Scheduled PID Controller

For PID tuning, pick a few values of h and sample the LPV dynamics at these heights to obtain local LTI models.

Pick three height values between 0.05 and 0.25.

```
hmin = 0.05;
hmax = 0.25;
hcd = linspace(hmin,hmax,3);
[Ga,Goffsets] = sample(Glpv,[],hcd);
size(Ga)
```

```
1x3 array of state-space models.
Each model has 1 outputs, 1 inputs, and 2 states.
```

Use `pidtune` to tune a PID controller for each value of h. The target crossover frequency is set to 50 rad/s.

```
wc = 50;
Ka = pidtune(Ga,'pid',wc);
Ka.Tf = 0.001;
```

Now use `ssInterpolant` to construct a gain-scheduled PID controller that linearly interpolates the tuned gains between the selected values of h. This controller also uses the trim current $u_0(p)$ as feedforward control so that the PID regulates the current around its equilibrium value.

```
Ka.SamplingGrid.h = hcd;
Koffsets = struct('y',{Goffsets.u});
Klpv = ssInterpolant(ss(Ka),Koffsets);
```

LTI Analysis

For evaluation purpose, model the ideal response as a second-order systems `Tideal`.

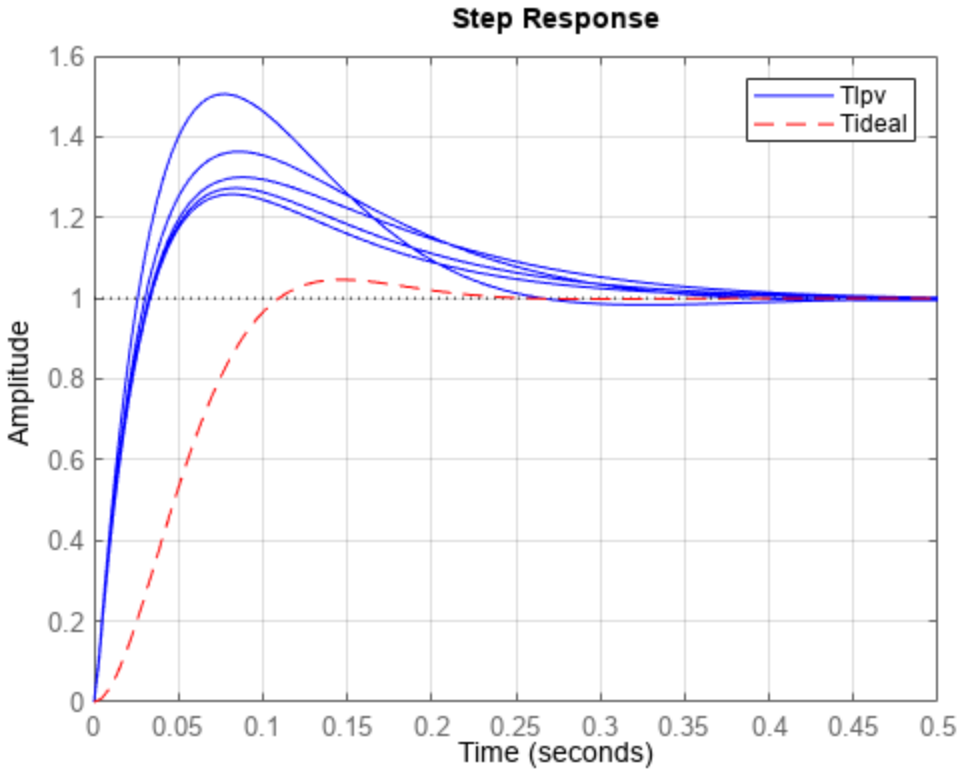
```
wn = 30;
zeta = 0.7;
Tideal = tf(wn^2,[1 2*zeta*wn wn^2]);
```

Sample the closed-loop LPV dynamics over a finer *h* grid.

```
hs = linspace(hmin,hmax,5);
```

Simulate the step response of the closed-loop model and compare with `Tideal`.

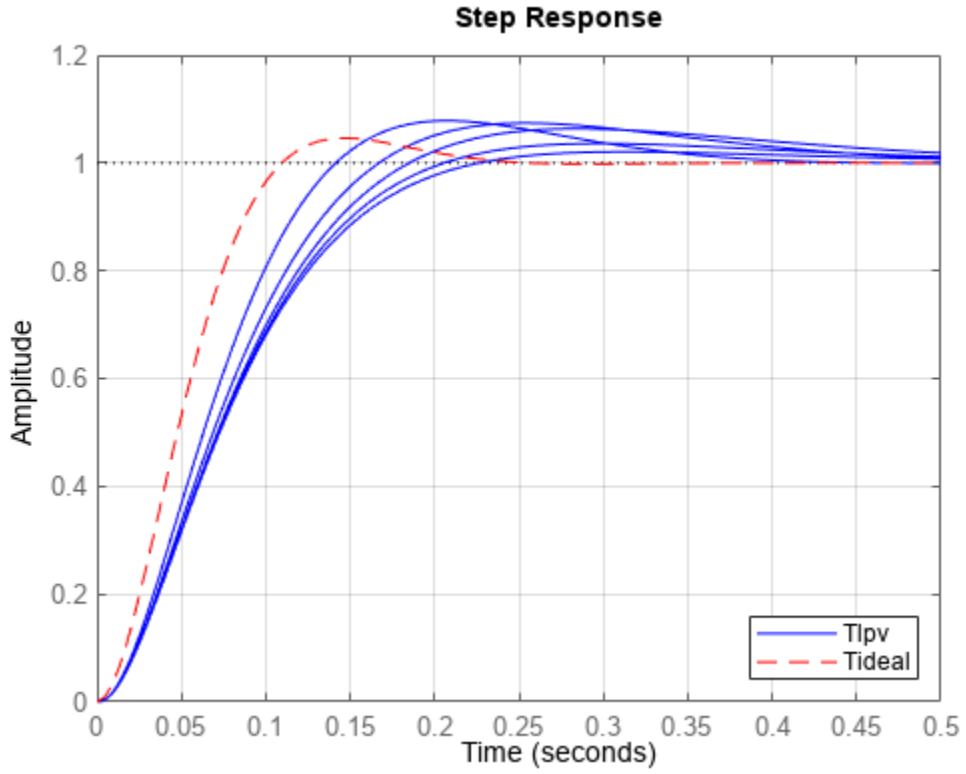
```
Tlpv = feedback(Glpv*Klpv,1);
step(sample(Tlpv,[],hs),'b',Tideal,'r--',0.5);
grid on
legend('Tlpv','Tideal')
```



The closed-loop system has additional zeros not in the ideal response `Tideal`. These zeros arise from the PID controller and cause a shorter rise time and larger overshoot.

You can use a two degree-of-freedom architecture with a reference prefilter `Fref` to reduce the overshoot. Model the prefilter as an LTI system for simplicity.

```
Fref = ss(-10,10,1,0);
step(sample(Tlpv*Fref,[],hs),'b',Tideal,'r--',0.5);
grid on;
legend('Tlpv','Tideal','Location','Southeast');
```



LPV Simulations

To approximate the nonlinear closed-loop response, simulate the transient response of the closed-loop LPV model with pre-filter. This requires you to specify the parameter trajectory $p(t) = h(t)$, which is not known a priori. As a proxy, you can set to the *ideal* response produced by `Tideal*Fref`.

```
CL = Tlpv*Fref;
```

Set the initial height and step signal parameters.

```
h0 = (hmin+hmax)/2;
tstep = 0.2;  Tf = 1.2;  hStepAmp = 0.25*h0;
```

Set $p(t)$ to ideal trajectory and simulate the response.

```
t = linspace(0,tstep+Tf,100);
StepConfig = RespConfig('InputOffset',h0,'Delay',tstep,'Amplitude',hStepAmp);
p_ideal = step(Tideal*Fref,t,StepConfig);
y1 = step(CL,t,p_ideal,StepConfig);
```

Alternatively, to use the *true* trajectory, you can specify $p(t)$ implicitly as a function $F(t, x, u)$ of time t, input u, and state x. Here $p(t)$ is just the first state.

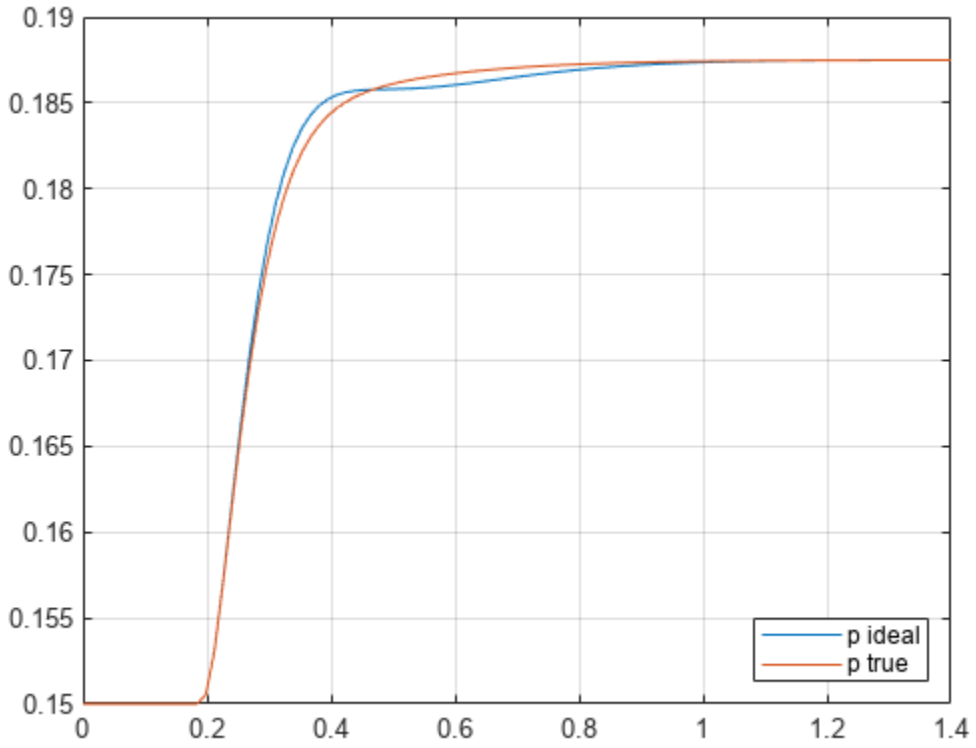
Set $p(t) = h(t)$ and simulate the step response.

```
pFcn = @(t,x,u) x(1);
StepConfig = RespConfig('InputOffset',h0,'Delay',tstep,...
```

```
   'Amplitude',hStepAmp,'InitialParameter',h0);
y2 = step(CL,t,pFcn,StepConfig);
```

Plot and compare the two responses.

```
plot(t,y1,t,y2);
legend('p ideal','p true','Location','Southeast');
grid
```



The two responses are close to each other. The Simulink-based counterpart of this example shows that the nonlinear response is well approximated by both LPV simulations with a small edge for the simulation with *true* trajectory.

See Also

`lpvss` | `sample` | `step` | `RespConfig`

Related Examples

- “LTV and LPV Modeling” on page 1-26
- “Using LTV and LPV Models in MATLAB and Simulink” on page 1-32
- “LPV Model of Engine Throttle” on page 1-157
- “Analysis of Gain-Scheduled PI Controller” on page 1-113
- “Gain-Scheduled LQG Controller” on page 1-106

- “Hidden Couplings in Gain-Scheduled Control” on page 1-118
- “LPV Model of Magnetic Levitation Model from Batch Linearization Results” on page 1-133

LPV Model of Magnetic Levitation Model from Batch Linearization Results

This example shows how to obtain a linear parameter-varying model of a nonlinear model for a magnetic levitation system from batch linearization results. This nonlinear model is described in the "LPV Model of Magnetic Levitation System" on page 1-127 example.

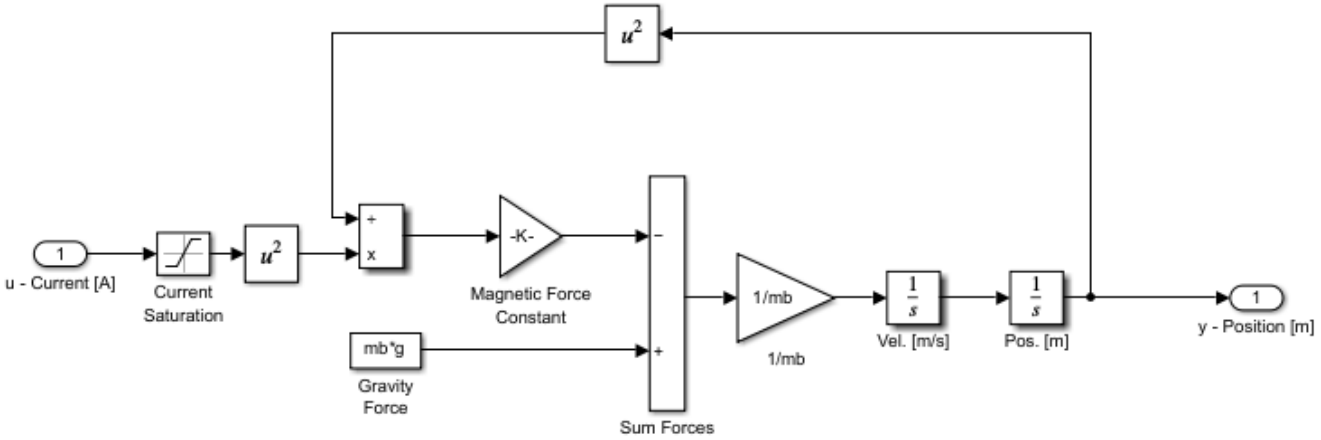
In this example, you:

1. Linearize a nonlinear Simulink® model on a grid of trim points.
2. Construct an LPV model from the gridded array of LTI models and trim offsets.
3. Design a controller on a grid of trim points. This grid may be different from the grid used for the linearization.
4. Test the controller using command-line LTI simulations (at frozen operating points) and LPV simulations (along a specified parameter trajectory).
5. Test the controller on the nonlinear plant in Simulink.

Model Parameters

Open the Simulink model.

```
open_system('MaglevOpenLoop.slx')
```



Nonlinear Magnetic Levitation Model

Specify the model parameters.

```
mb = 0.02;
g = 9.81;
alpha = 2.4832e-5;
umin = -inf;
umax = inf;
```

Batch Trim and Linearization

Specify collection of trim conditions: $(h,\dot{h}) = (h_s(i),0)$.

```
Ns = 5;
hmin = 0.05; hmax = 0.25;
hs = linspace(hmin,hmax,Ns);
```

Create operating point specification.

```
clear opspec;
for i=1:Ns
    % Set states to desired equilibrium values
    h0 = hs(i);
    hdot0 = 0;

    % Initialize operating point specification
    opspec(i) = operspec('MaglevOpenLoop');

    % Position: h=hs(i)
    opspec(i).States(1).Known = 1;
    opspec(i).States(1).SteadyState = 1;

    % Velocity: hdot=0
    opspec(i).States(2).Known = 1;
    opspec(i).States(2).SteadyState = 1;

    % Input current: Restrict u>=0
    opspec(i).Inputs.Min=0;
end
```

Find trim points.

```
opt = findopOptions('DisplayReport','off', 'OptimizerType','graddescent-proj');
[op,report] = findop('MaglevOpenLoop',opspec,opt);
```

Compute linearizations.

```
io = [linio('MaglevOpenLoop/u - Current [A]',1,'in');                    % u
    linio('MaglevOpenLoop/Magnetic Levitation Plant Model',1,'out')]; % y
linOpt = linearizeOptions('StoreOffsets',true);
[G,~,info] = linearize('MaglevOpenLoop',op,io,linOpt);
G.u = 'u';
G.y = 'y';
G.SamplingGrid = struct('h',hs);
```

Construct the LPV model using `ssInterpolant`.

```
offsets = info.Offsets;
Glpv = ssInterpolant(G,offsets);
```

Batch PID Tuning

The grid for the controller is coarser than the one used for the model to illustrate that you can do so. Moreover, the controller gridding is often coarse to simplify the controller design (fewer grid points) while allowing higher-fidelity LPV model with finer gridding for analysis and simulation.

```
Ncd = 3;
hcd = linspace(hmin,hmax,Ncd);
```

Tideal is the ideal, second-order response corresponding to (ω_n,ζ).

```
wn = 30;
zeta = 0.7;
Tideal = tf(wn^2,[1 2*zeta*wn wn^2]);
```

Use `pidtune` to tune a gain-scheduled PID controller.

```
wc = 50;
[Ga,Goffsets] = sample(Glpv,[],hcd);
Ka = pidtune(Ga,'pid',wc);
Ka.Tf = 0.001;
Ka.SamplingGrid.h = hcd;
```

Build LPV controller with equations

$$\dot{\zeta} = A_K(p)\zeta + B_K(p)e$$

$$u = u_0(p) + C_K(p)\zeta + D_K(p)e,$$

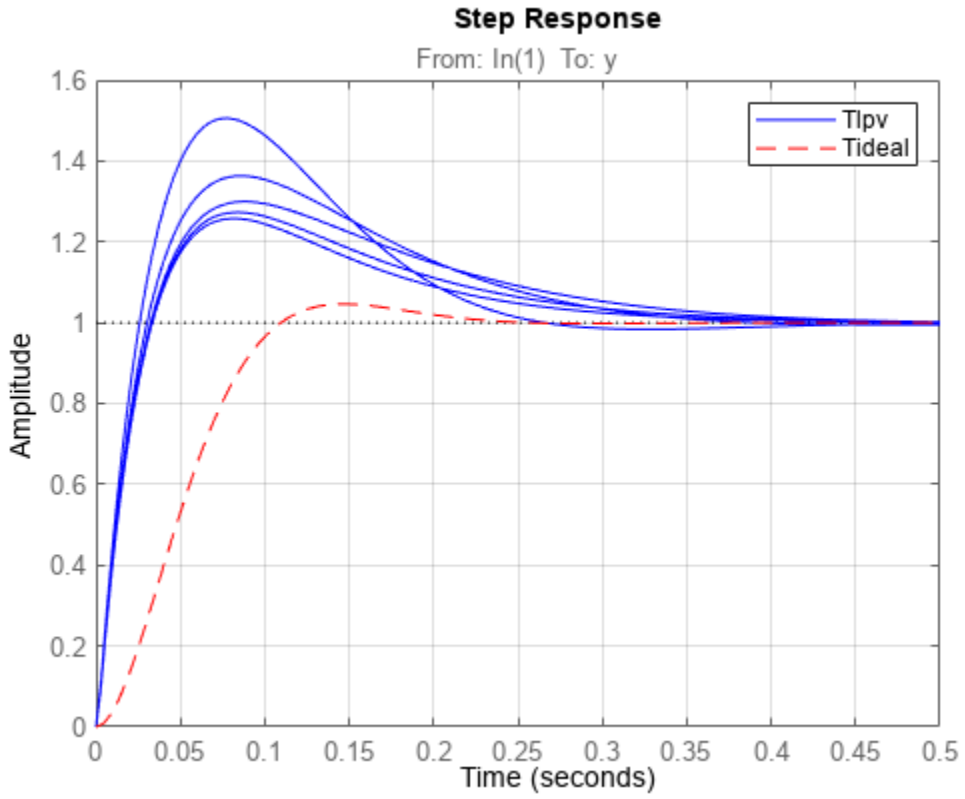
where $u_0(p)$is the trim current for the height p. This controller will regulate the current around this equilibrium value.

```
Koffsets = struct('u',0,'y',{Goffsets.u});
Ka = ss(Ka);
Klpv = ssInterpolant(Ka,Koffsets);
```

LTI Analysis

Simulate the step response of the closed-loop model and compare with `Tideal`.

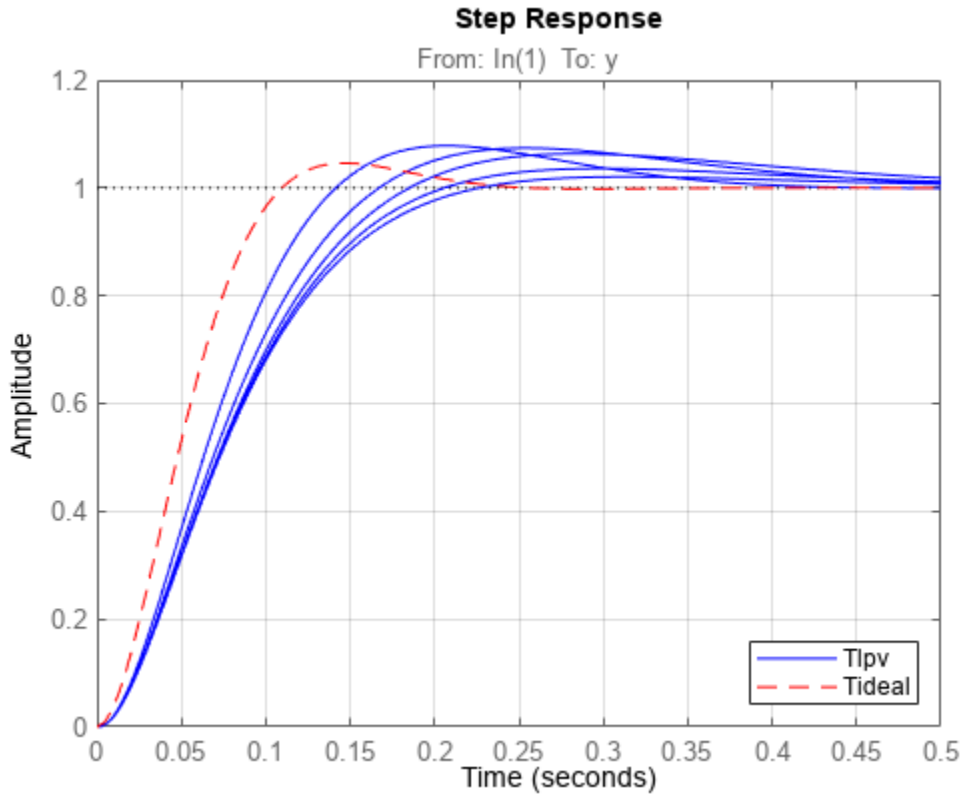
```
figure(1);
Tf = 0.5;
Tlpv = feedback(Glpv*Klpv,1);
step(sample(Tlpv,[],hs),'b',Tideal,'r--',Tf);
grid on;
legend('Tlpv','Tideal');
```



The closed-loop system has additional zeros not in the ideal response Tideal. These zeros arise from the PID controller and cause a shorter rise time and larger overshoot.

You can use a two degree-of-freedom architecture with a reference prefilter `Fref` to reduce the overshoot. Model the prefilter as an LTI system for simplicity.

```
Fref = ss(-10,10,1,0);
figure(2);
Tf = 0.5;
step(sample(Tlpv*Fref,[],hs),'b',Tideal,'r--',Tf);
grid on;
legend('Tlpv','Tideal','Location','Southeast');
```



Compare LPV and Nonlinear Simulations

Simulate the nonlinear response with the gain-scheduled PID. The `MaglevClosedLoop.slx` model uses the LPV System block for the controller.

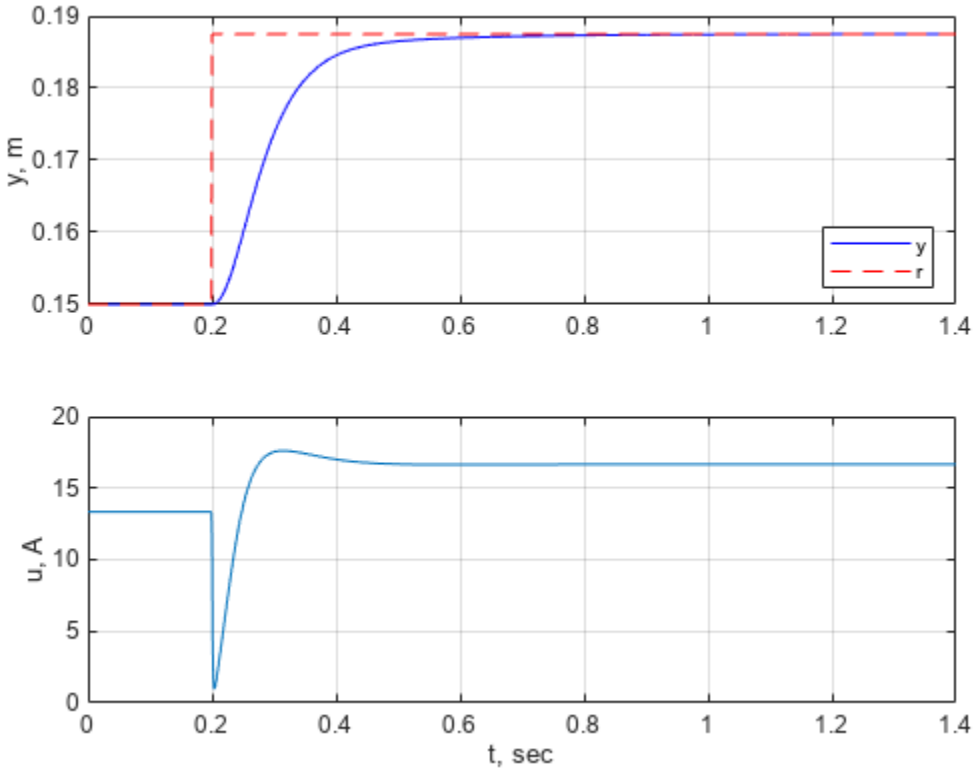
Specify initial conditions and simulate the model.

```
h0 = (hmin+hmax)/2;
hdot0 = 0;
tstep = 0.2;
Tf = 1.2;
hstep0 = h0;
hStepAmp = 0.25*h0;
hstepf = h0+hStepAmp;
sim('MaglevClosedLoop',[0 tstep+Tf]);
```

Plot the responses.

```
figure(3);
subplot(2,1,1);
plot(y.Time,y.Data,'b',r.Time,r.Data,'r--');
ylabel('y, m');
legend('y','r','Location','Southeast');
grid on;
subplot(2,1,2);
plot(u.Time,u.Data);
ylabel('u, A');
```

```
xlabel('t, sec');
grid on;
```



Compare results with the LPV simulation.

```
CL = Tlpv*Fref;
```

Set $p(t)$ to *ideal* trajectory and simulate the step response.

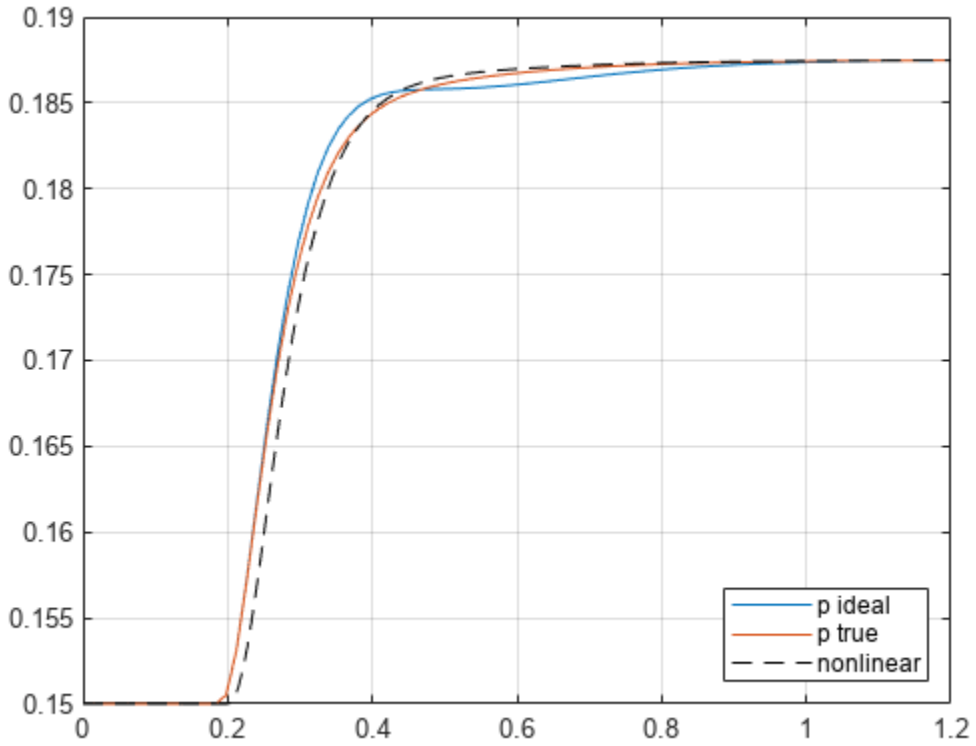
```
t = linspace(0,tstep+Tf,100);
StepConfig = RespConfig('InputOffset',h0,'Delay',tstep,'Amplitude',hStepAmp);
p_ideal = step(Tideal*Fref,t,StepConfig);
y1 = step(CL,t,p_ideal,StepConfig);
```

Set $p(t) = h(t)$ and simulate the step response.

```
pFcn = @(t,x,u) x(1);
StepConfig = RespConfig('InputOffset',h0,'Delay',tstep,...
   'Amplitude',hStepAmp,'InitialParameter',h0);
y2 = step(CL,t,pFcn,StepConfig);
```

Plot the response comparison.

```
figure(4)
tsim = y.Time;
ysim = y.Data;
plot(t,y1,t,y2,tsim,ysim,'k--');
legend('p ideal','p true','nonlinear','Location','Southeast');
xlim([0 Tf]), grid
```



Both LPV simulations approximate the nonlinear response well. The simulation with *true* trajectory provides a slightly better approximation.

Close the models.

```
bdclose('MaglevOpenLoop')
bdclose('MaglevClosedLoop')
```

See Also

`ssInterpolant` | `sample` | `ndgrid` | `linearize` | `step` | `RespConfig`

Related Examples

- “LTV and LPV Modeling” on page 1-26
- “Using LTV and LPV Models in MATLAB and Simulink” on page 1-32
- “LPV Model of Magnetic Levitation System” on page 1-127
- “Approximate Nonlinear Aircraft Pitch Dynamics Using LPV Model” on page 1-68
- “Control Design for Wind Turbine” on page 1-140

Control Design for Wind Turbine

This example discusses the control system for a 1.5 MW wind turbine. This example models the rotor dynamics as a simple first-order system, which neglects the flexible modes in the drivetrain, blades, and tower. The focus is on the design and validation of a gain-scheduled controller for the blade pitch in high-wind regime. For more details, see [1] on page 1-155 and [2] on page 1-155.

Turbine Model and Data

The rigid-body dynamics for the low-speed shaft are:

$$J\dot{\omega} = T_a - T_g,$$

where ω is the rotor speed, T_a is the aerodynamic torque, and T_g is the reaction torque from the generator connected to the high-speed shaft.

The aerodynamic torque depends on wind speed and blade pitch. Its calculation involves power coefficient data consisting of:

1. A grid `TSRgrid` of tip speed ratios (unitless). The tip speed ratio is TSR = $\frac{R\omega}{V}$, where R is the rotor radius (m), ω is the rotor speed (rad/sec), and V is the wind speed (m/s).
2. A grid `Bgrid` of blade pitch angles in degrees.
3. The table `CpData` of power coefficients C_p (unitless) over the grid of tip speed ratios and pitch blade angles. The power generated by the turbine is $0.5\rho AV^3C_p$, where ρ is the air density (kg/m^3) and $A = \pi R^2$ is the rotor area (m).

This data is generated over a wide range of blade pitch and tip speed ratio values. The normal operating points correspond to $C_p > 0$. Some entries of the table have $C_p < 0$ (the turbine actually requires energy to operate at these points). These points where $C_p < 0$ should be considered with some caution.

Load the data.

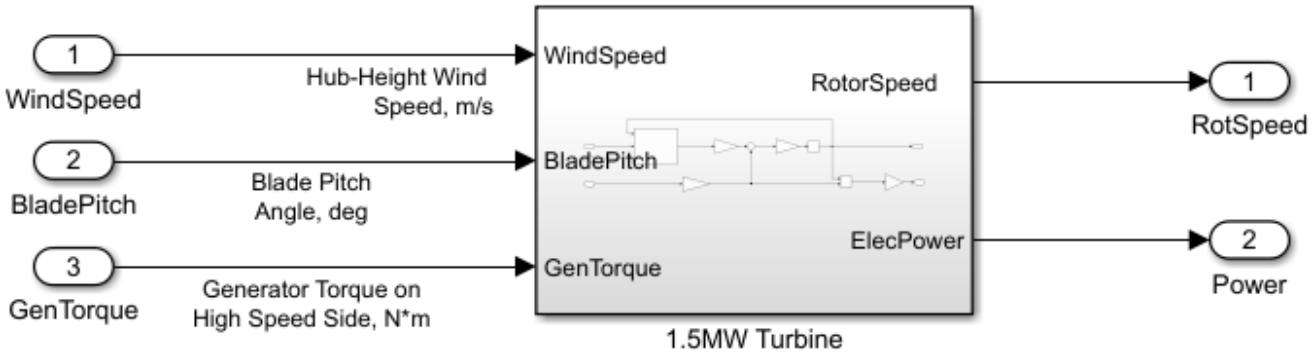
```
load WindCpData Bgrid TSRgrid CpData
```

This example uses the following values for the turbine physical parameters:

```
GBRatio = 88;       % Gearbox ratio, unitless
GBEff = 1.0;        % Gearbox efficiency, unitless (=1 for perfect eff.)
GenEff = 0.95;      % Generator efficiency, unitless (=1 for perfect eff.)
Jgen = 53;          % Generator Inertia about high-speed shaft, kg*m^2
Jrot = 3e6;         % Rotor inertia about low-speed shaft, kg*m^2
Jeq = Jrot+Jgen*GBRatio^2; % Equiv. inertia about low-speed shaft, kg*m^2
R = 35;             % Rotor radius, m
rho = 1.225;        % Air density, kg/m^3
wPA = 8.6;          % Pitch actuator natural frequency, rad/sec
zetaPA = 0.8;       % Pitch actuator damping ratio, unitless
Bmax = 90;          % Maximum blade pitch angle, degs
Kaw = 0.5;          % Anti-windup gain, rad/s/deg
```

The Simulink® model `WindTurbineOpenLoop` implements the simplified model of the rotor dynamics. Open the model.

```
open_system('WindTurbineOpenLoop')
```



Rated Operating Point and Operating Regions

The power electronics on a wind turbine are sized to only produce a certain maximum power. This is called the rated power of the turbine (1.5 MW for this turbine). The rated torque, wind speed, and rotor speed correspond to the operating conditions where the turbine achvies the rated power.

Specify the rated power as 1.5 MW.

```
PRated = 1.5e6;
```

The rated rotor speed is 1800 rpm on high-speed shaft (HSS). Converted this to rad/sec on the low-speed shaft (LSS).

```
wRatedHSS = 1800*(2*pi/60);
wRatedLSS = wRatedHSS/GBRatio;
```

The rated power is the product of the rated rotor speed, generator torque, and generator efficiency. Deduce the rate generator torque on high speed shaft, N*m.

```
GenTRated = PRated/(wRatedHSS*GenEff);
```

Specify the rated wind speed (m/s). Value of rated wind speed that you choose to achieve the rated rotor speed and rated generator torque form an equilibrium point (although with $C_p < C_{p,\max}$).

```
WindRated = 11.2;
```

The controller switches between two operating regions delimited by the rated operating point:

- Region 2 (torque control): For wind speeds below rated, the blade pitch is set equal to its optimal (most efficient) value and the generator torque is set to a value proportional to ω^2.
- Region 3 (blade pitch control): For wind speeds above rated, the generator torque is set to its rated value while the blade pitch is adjusted to maintain the rated rotor speed and deliver the rated power.

The generator torque in Region 2 is set to

$$\text{GeneratorTorque} = \text{Kreg2} \times \text{RotorSpeed}^2,$$

where you choose Kreg2 such that there is a smooth transition with Region 3. Rotor speed is `wRatedLSS` and generator torque is `GenTRated`.

```
Kreg2 = GenTRated / wRatedLSS^2
```

```
Kreg2 = 1.8257e+03
```

Finally, compute the maximal power coefficient and corresponding optimal tip speed ratio and blade pitch angle.

```
CpMax = max(CpData,[],'all');
[i,j] = find(CpData==CpMax);
TSRopt = TSRgrid(i);
Bopt = Bgrid(j);
```

Optimal Operating Conditions as Function of Wind Speed

Compute the optimal steady-state operating conditions for wind speeds ranging from 4 to 24 m/s.

Specify the wind speed data for computing equilibrium points and initialize the arrays to store the rotor speeds on LSS, generator torques, blade pitch angles, and power delivered.

```
WindData = sort([4:0.5:24 WindRated]);

nW = numel(WindData);
wLSSeq = zeros(nW,1);
GenTeq = zeros(nW,1);
BladePitcheq = zeros(nW,1);
Peq = zeros(nW,1);
```

In Region 2 (torque control), the rotor speed is proportional to the wind speed and the blade pitch is set to `Bopt`.

```
for i=1:nW
    Wind = WindData(i);
    if Wind<=WindRated
        % Region 2: Torque Control
        wLSSeq(i) = Wind/WindRated*wRatedLSS;
        GenTeq(i) = Kreg2*wLSSeq(i)^2;
        wHSS = wLSSeq(i)*GBRatio;
        Peq(i) = GenTeq(i)*wHSS*GenEff; % wRatedHSS*GenEff;
        BladePitcheq(i) = Bopt;
        % Populate operating point
        op(i) = operpoint('WindTurbineOpenLoop');
        op(i).States.x = wLSSeq(i);
        op(i).Inputs(1).u = Wind;
        op(i).Inputs(2).u = BladePitcheq(i);
```

```
        op(i).Inputs(3).u = GenTeq(i);
    end
end
```

In Region 3 (blade pitch control), the rotor speed and generator torque max out to their rated values `wRatedLSS` and `GenTRated`. Use `findop` to compute the blade pitch angle that maintains these steady-state values.

Specify trim options.

```
opt = findopOptions('DisplayReport','off', 'OptimizerType','lsqnonlin');
opt.OptimizationOptions.Algorithm = 'trust-region-reflective';
```

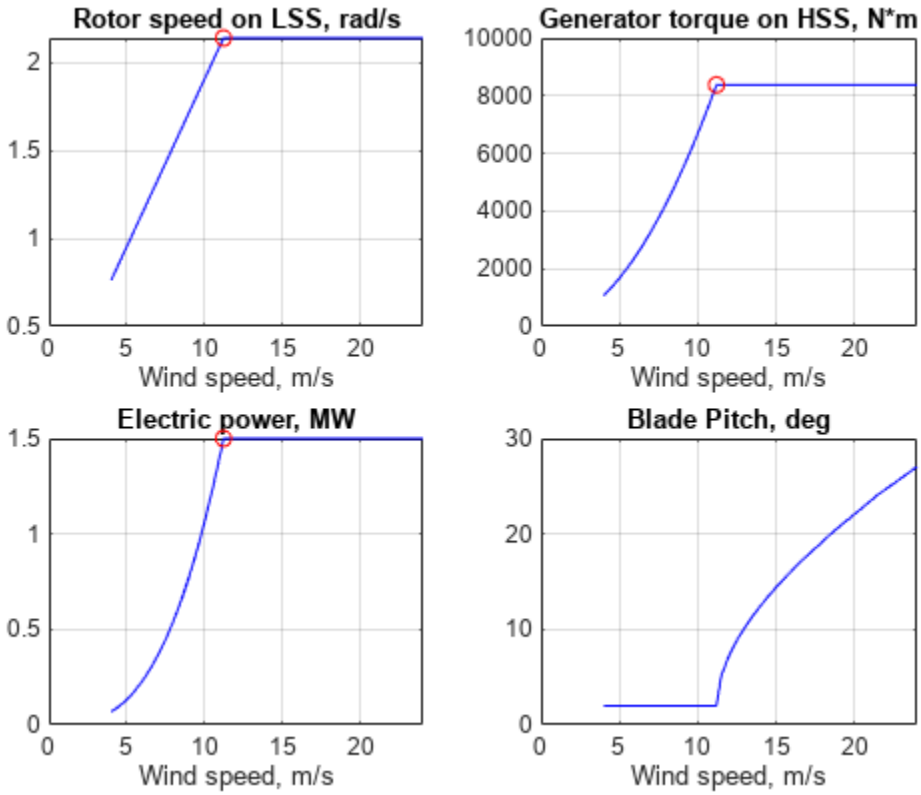
Perform batch trimming.

```
opspec = operspec('WindTurbineOpenLoop');
for i=1:nW
    Wind = WindData(i);
    if Wind>WindRated
        % Region 3: Blade Pitch Control
        wLSSeq(i) = wRatedLSS;
        GenTeq(i) = GenTRated;
        Peq(i) = PRated;
        % Trim condition
        opspec.States.Known = 1;
        opspec.States.SteadyState = 1;
        opspec.Inputs(1).Known=1;
        opspec.Inputs(1).u = Wind;
        opspec.Inputs(2).min = BladePitcheq(i-1);
        opspec.Inputs(3).Known=1;
        opspec.Inputs(3).u = GenTeq(i);
        % Compute corresponding operating point
        op(i) = findop('WindTurbineOpenLoop',opspec,opt);
        % Log steady-state blade pitch angle
        BladePitcheq(i) = op(i).Inputs(2).u;
    end
end
```

Plot the optimal settings of rotor speed, generator torque, electric power, and blade pitch angle as a function fo wind speed. The red dot marks the rated operating point and transition between Regions 2 and 3.

```
clf
subplot(2,2,1)
plot(WindData,wLSSeq,'b',WindRated,wRatedLSS,'ro');
grid on;
xlabel('Wind speed, m/s');
title('Rotor speed on LSS, rad/s');

subplot(2,2,2)
plot(WindData,GenTeq,'b',WindRated,GenTRated,'ro');
grid on;
xlabel('Wind speed, m/s');
title('Generator torque on HSS, N*m');

subplot(2,2,3)
plot(WindData,Peq/1e6,'b',WindRated,PRated/1e6,'ro');
grid on;
```

```
xlabel('Wind speed, m/s');
title('Electric power, MW');

subplot(2,2,4)
plot(WindData,BladePitcheq,'b');
grid on;
xlabel('Wind speed, m/s');
title('Blade Pitch, deg');
```



Batch Linearization and LPV Model

Obtain a linearized model with offsets for the operating points computed previously.

Specify linearization input and output points.

```
io = [linio('WindTurbineOpenLoop/WindSpeed',1,'in');
    linio('WindTurbineOpenLoop/BladePitch',1,'in');
    linio('WindTurbineOpenLoop/GenTorque',1,'in');
    linio('WindTurbineOpenLoop/1.5MW Turbine',1,'out'); % RotorSpeed
    linio('WindTurbineOpenLoop/1.5MW Turbine',2,'out')]; % Power
```

Linearize the model for each of the trim conditions. Store linearization offset information in the |info| structure.

```
linOpt = linearizeOptions('StoreOffsets',true);
[G,~,info] = linearize('WindTurbineOpenLoop',op,io,linOpt);
G.u = {'WindSpeed'; 'BladePitch'; 'GenTorque'};
```

```
G.y = {'RotorSpeed','Power'};
G.SamplingGrid = struct('WindSpeed',WindData);
```

Use `ssInterpolant` to construct an LPV model that interpolates these linearized models between grid points (wind speeds).

```
offsets = info.Offsets;
Glpv = ssInterpolant(G,offsets);
```

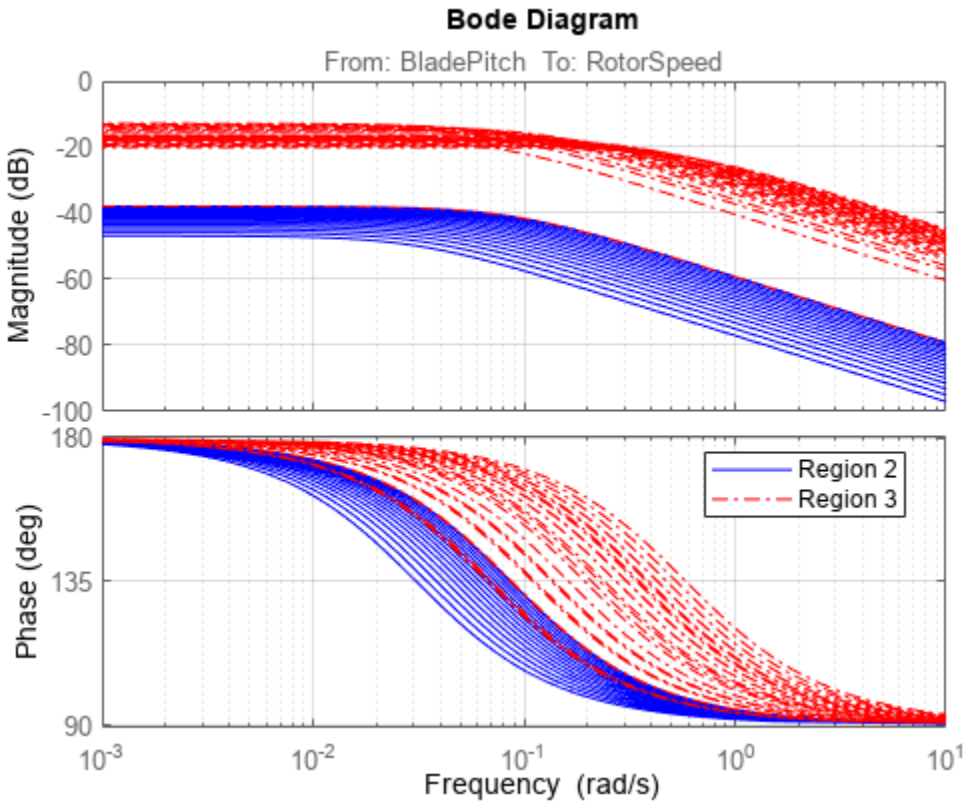
Linear Analysis for Fixed Wind Speeds

Split wind speeds into below rated (Region 2) and above rated (Region 3). Sample the LPV model of the turbine at these wind speeds.

```
[GR2,GR2offsets] = sample(Glpv,[],WindData( WindData<=WindRated ));
[GR3,GR3offsets] = sample(Glpv,[],WindData( WindData>=WindRated ));
```

Plot the Bode responses of linearization from blade pitch to rotor speed.

```
clf
bode(GR2(1,2,:),'b',GR3(1,2,:),'r-.')
legend('Region 2','Region 3','Location','Best');
grid on
```



Gain-Scheduled Blade Pitch Controller

Design a gain-scheduled PI controller to adjust blade pitch in Region 3. The gains are scheduled on blade pitch angle which you cam measure more reliably than wind speed. Recall that blade pitch

must be adjusted to keep rotor speed, generator torque, and electric power from exceeding their rated values.

Sample the LPV plant for 20 values of wind speed between 11.5 and 24.

```
WindGS = linspace(11.5,24,20)';
BladePitchGS = interp1(WindData,BladePitcheq,WindGS);
GGS = sample(Glpv,[],WindGS);
```

For each wind speed, tune the PI gains to achieve a bandwidth of 0.66 rad/s.

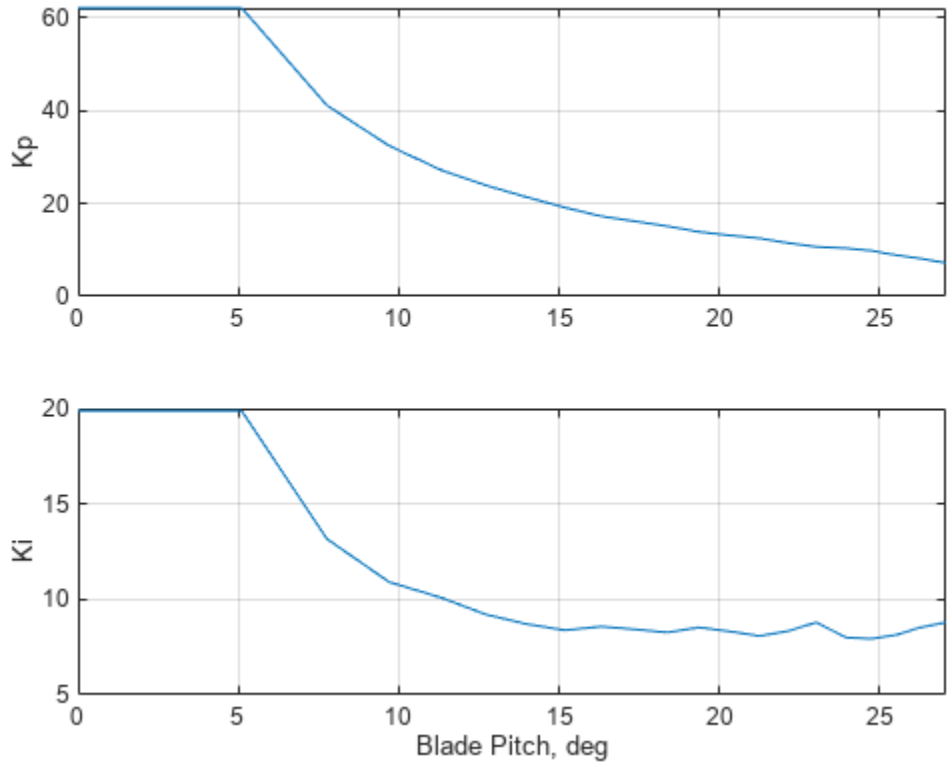
```
wL = 0.66;
opt = pidtuneOptions('PhaseMargin',70);
CPI = pidtune(-GGS(1,2),'pi',wL,opt);
Kp = CPI.Kp(:);
Ki = CPI.Ki(:);
```

Extend gain schedule to the entire range of blade pitch angles.

```
KpGS = [Kp(1); Kp; Kp(end)];
KiGS = [Ki(1); Ki; Ki(end)];
BladePitchGS = [0; BladePitchGS; Bmax];
```

Plot the gain schedules as a function of blade pitch angle.

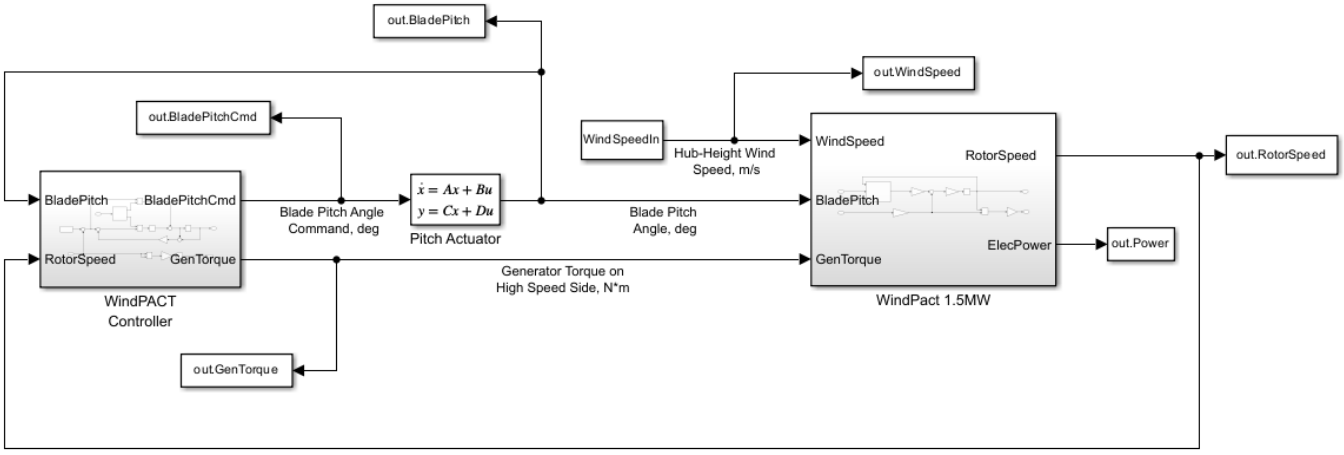
```
clf
subplot(2,1,1)
plot(BladePitchGS,KpGS);
ylabel('Kp');
grid on;
xlim(BladePitchGS([1 end-1]));
subplot(2,1,2)
plot(BladePitchGS,KiGS);
ylabel('Ki');
grid on;
xlim(BladePitchGS([1 end-1]));
xlabel('Blade Pitch, deg');
```



Nonlinear Closed-Loop Simulation

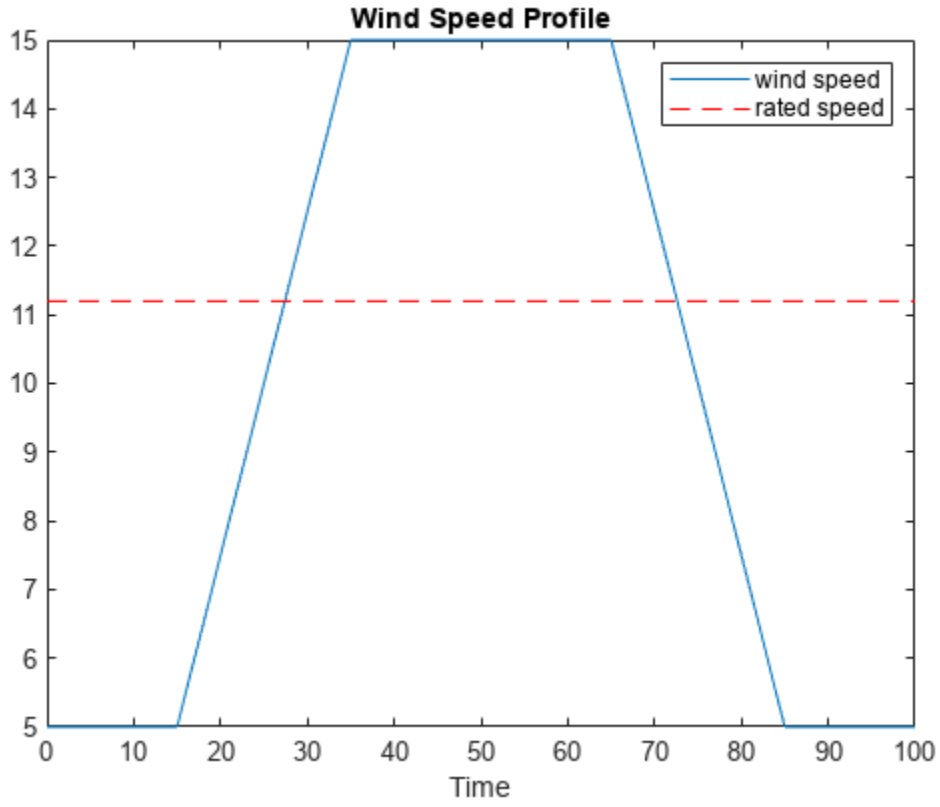
The Simulink model `WindTurbineClosedLoop` combines the turbine model, torque control for Region 2, and gain-scheduled blade pitch PI controller for Region 3.

```
open_system('WindTurbineClosedLoop')
```



Use the following wind speed profile as input to the simulation.

```
V0 = 5;
Vf = 15;
T1 = 15;
T2 = 20;
T3 = 30;
Tf = 2*T1+2*T2+T3;
WindSpeedIn = [0 V0; T1 V0; T1+T2 Vf; T1+T2+T3 Vf; T1+2*T2+T3 V0; Tf V0];

t = (0:0.1:Tf)';
Wind = interp1(WindSpeedIn(:,1),WindSpeedIn(:,2),t);
clf
plot(t,Wind,[0 Tf],WindRated*[1 1],'r--')
xlabel('Time')
title('Wind Speed Profile')
legend('wind speed','rated speed')
```



Simulate the closed-loop response for this wind profile with proper initial conditions.

Specify the initial conditions for rotor speed, actuator state, and rotor speed error. This assumes `V0 < WindRated`, that is, Region 2.

```
w0 = V0/WindRated*wRatedLSS;
xPA0= [Bopt; 0];
e0 = wRatedLSS-w0;
```

Specify the condition for controller integrator to be in steady-state:

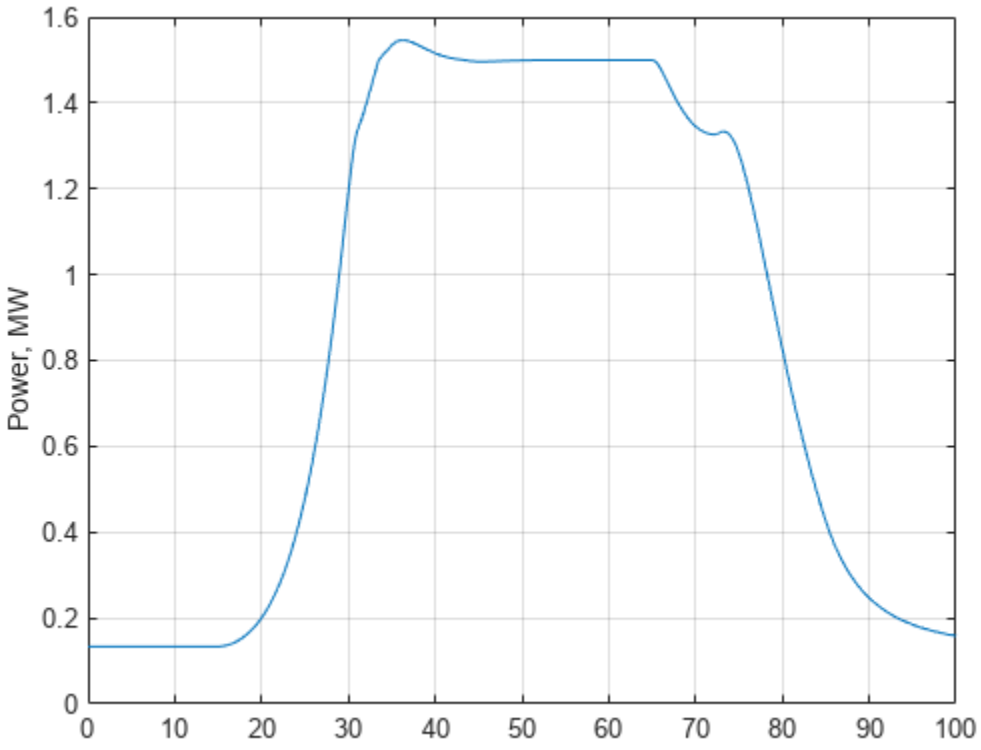
```
0 = e0 + Kaw*(-KpGS(1)*e0-xK0-Bopt)

xK0 = e0*(1/Kaw-KpGS(1))-Bopt;
```

Simulate the model.

```
out = sim('WindTurbineClosedLoop',Tf);
```

Plot the power output.

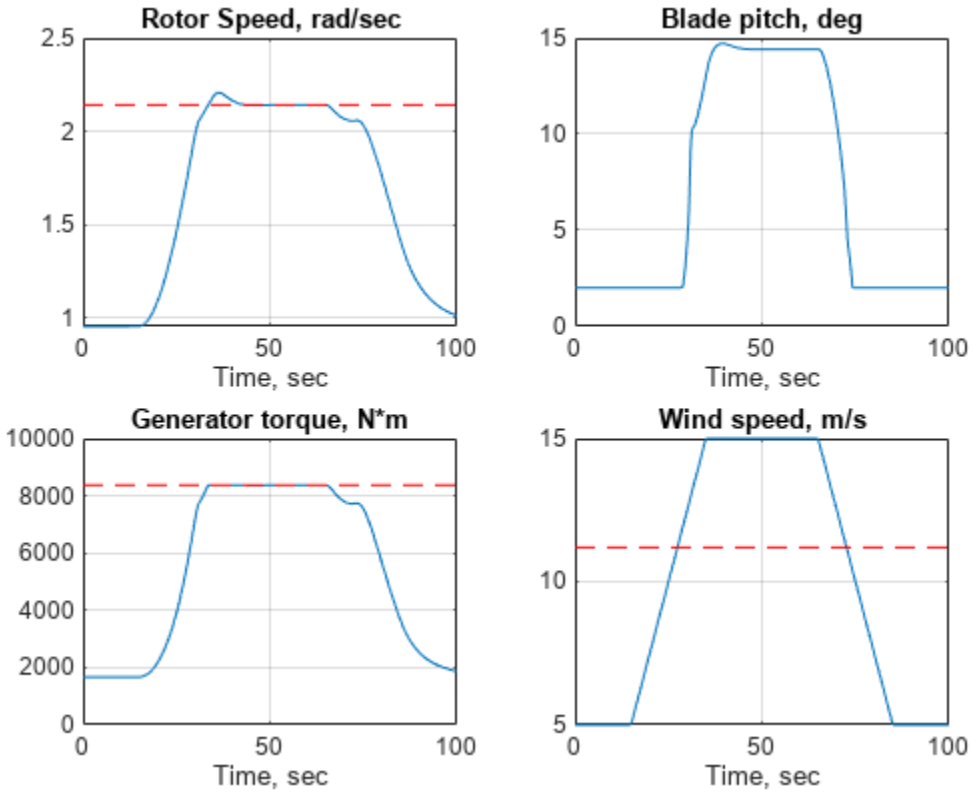
```
Power = out.Power;
clf
plot(Power.Time,Power.Data/1e6);
ylabel('Power, MW');
grid on;
```



Plot the other variables.

```
clf
subplot(2,2,1)
RotorSpeed = out.RotorSpeed;
plot(RotorSpeed.Time,RotorSpeed.Data,[0 Tf],wRatedLSS*[1 1],'r--');
title('Rotor Speed, rad/sec');
xlabel('Time, sec');grid on;

subplot(2,2,2)
BladePitch = out.BladePitch;
plot(BladePitch.Time,BladePitch.Data);
```

```
title('Blade pitch, deg');
xlabel('Time, sec'); grid on;

subplot(2,2,3)
GenTorque = out.GenTorque;
plot(GenTorque.Time,GenTorque.Data,[0 Tf],GenTRated*[1 1],'r--');
title('Generator torque, N*m');
xlabel('Time, sec'); grid on;

subplot(2,2,4)
WindSpeed = out.WindSpeed;
plot(WindSpeed.Time,WindSpeed.Data,[0 Tf],WindRated*[1 1],'r--');
title('Wind speed, m/s');
xlabel('Time, sec');  grid on;
```



Closed-Loop LPV Simulation

From the LPV model `Glpv` of the turbine and the PI gain schedule, you can also construct a closed-loop LPV model and use it to validate the gain-scheduled controller in Region 3. First use `ssInterpolant` to create an LPV model of the gain-scheduled controller.

```
CPI = ss(0,permute(KiGS,[2 3 1]),1,permute(KpGS,[2 3 1]));
CPI.SamplingGrid = struct('BladePitch',BladePitchGS);

Clpv = ssInterpolant(CPI);
Clpv.InputName = 'RotSpeedErr';
Clpv.OutputName = 'BladePitchCmd';
Clpv.StateName = 'Controller';
```

The blade pitch actuator is modeled as a second-order system.

```
Actuator = ss([0 1; -wPA^2 -2*zetaPA*wPA],[0; wPA^2],[1 0],0);
Actuator.InputName = 'BladePitchCmd';
Actuator.OutputName = 'BladePitch';
Actuator.StateName = {'BladePitch','Act2'};
```

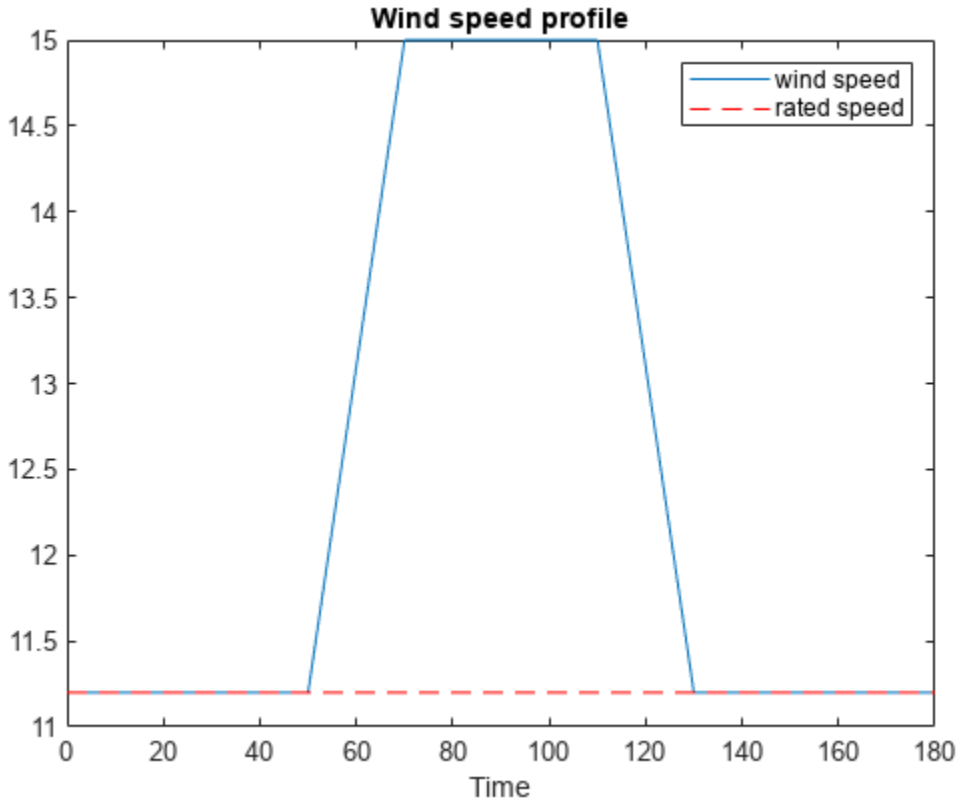
Use `connect` to build a closed-loop LPV model `Tlpv` including the LPV plant and the gain-scheduled PI controller. This closed-loop model depends on two parameters: wind speed and blade pitch angle.

```
SumBlk = sumblk('RotSpeedErr = RotorSpeed - RotSpeedCmd');

Tlpv = connect(Glpv,Clpv,Actuator,SumBlk,...
   {'WindSpeed','RotSpeedCmd','GenTorque'},...
   {'RotorSpeed','Power','BladePitch'});

Tlpv.ParameterName
```

```
ans = 2x1 cell
    {'WindSpeed' }
    {'BladePitch'}
```

To validate the blade pitch controller, use a wind profile that lies entirely in Region 3.

```
V0 = WindRated;
Vf = 15;
T1 = 50;
T2 = 20;
T3 = 40;
Tf = 2*T1+2*T2+T3;
WindSpeedIn = [0 V0; T1 V0; T1+T2 Vf; T1+T2+T3 Vf; T1+2*T2+T3 V0; Tf V0];

t = (0:0.1:Tf)';
Wind = interp1(WindSpeedIn(:,1),WindSpeedIn(:,2),t);
clf
plot(t,Wind,[0 Tf],WindRated*[1 1],'r--')
xlabel('Time')
title('Wind speed profile')
legend('wind speed','rated speed')
```



Use `lsim` to simulate the closed-loop response. Define the parameter trajectory implicitly (wind speed is the first input and blade pitch angle is the first state of the actuator model).

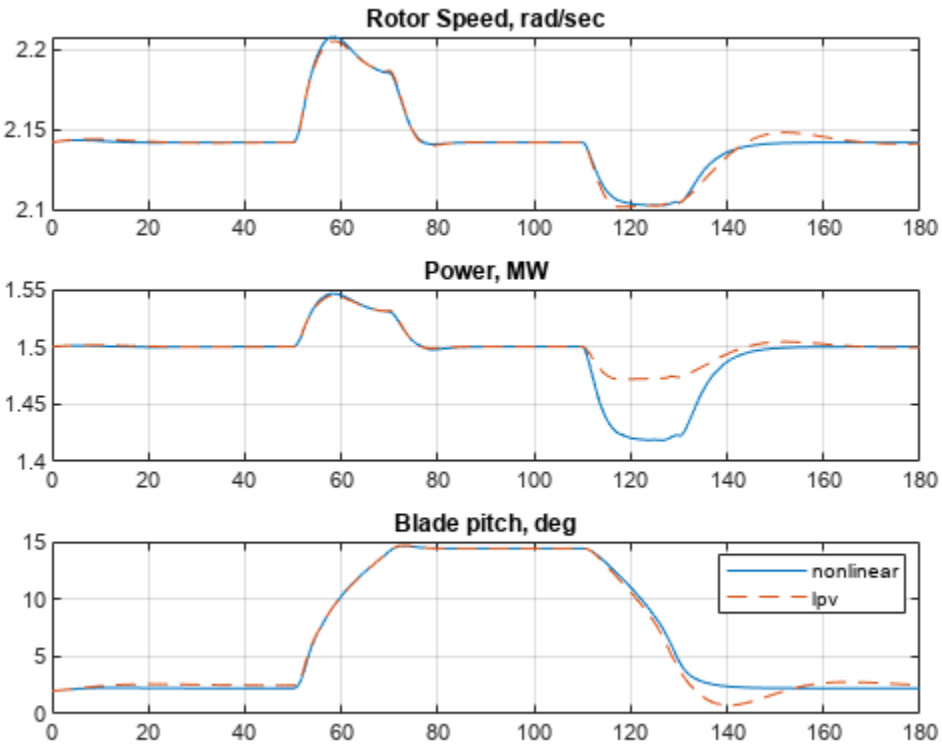
```
% Inputs
u = zeros(numel(t),3);
u(:,1) = Wind;       % Wind profile
u(:,2) = wRatedLSS;  % Rotor speed
u(:,3) = GenTRated;  % Generator torque

% Initial condition
xinit = [wRatedLSS;Bopt;Bopt;0];

% Parameter trajectory
pFcn = @(t,x,u) [u(1);x(3)];  % x(3) = BladePitch

% LPV simulation
ylpv = lsim(Tlpv,u,t,xinit,pFcn);
```

Run the nonlinear simulation for the same wind profile.

```
w0 = wRatedLSS;  % Initial rotor speed, rad/sec
xPA0 = [Bopt; 0];    % Initial actuator state
xK0 = -Bopt;        % integrator output
out = sim('WindTurbineClosedLoop',Tf);
```

Compare the LPV and nonlinear simulations.

```
RotorSpeed = out.RotorSpeed;
clf, subplot(311)
plot(RotorSpeed.Time,RotorSpeed.Data,t,ylpv(:,1),'--')
title('Rotor Speed, rad/sec');
grid on;
Power = out.Power;
subplot(312)
plot(Power.Time,Power.Data/1e6,t,ylpv(:,2)/1e6,'--')
title('Power, MW');
grid on;
BladePitch = out.BladePitch;
subplot(313)
plot(BladePitch.Time,BladePitch.Data,t,ylpv(:,3),'--')
title('Blade pitch, deg');
grid on;
legend('nonlinear','lpv','location','Best')
```



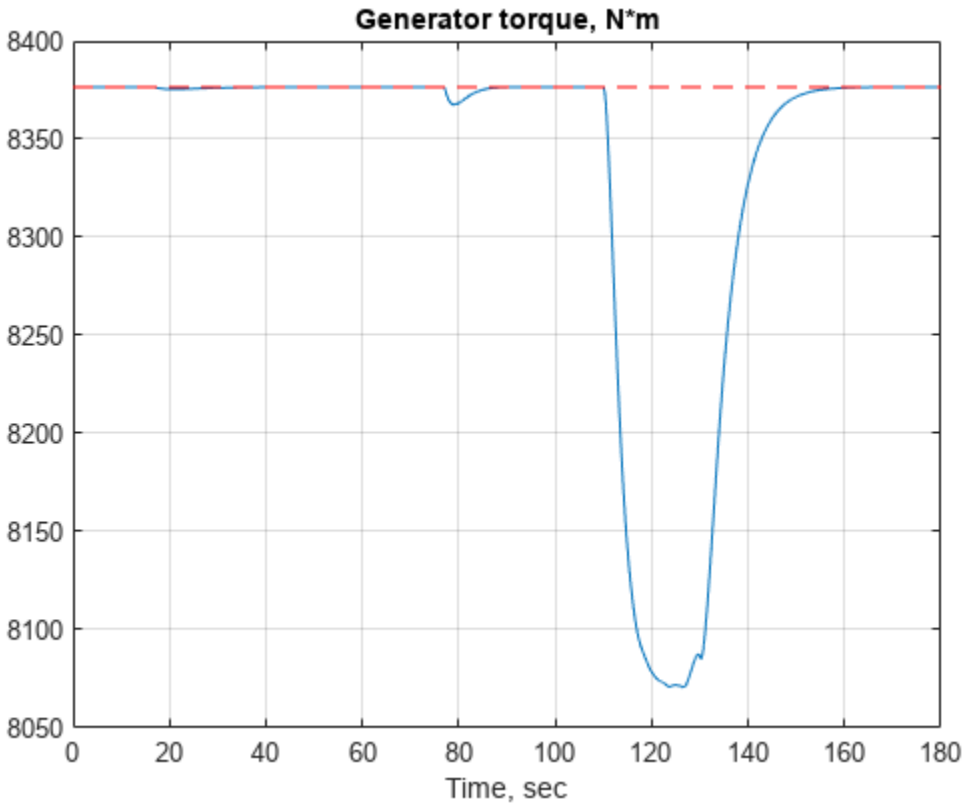
The LPV simulation accurately models rotor speed and blade pitch but underestimates the drop in power when wind speed decreases. This is because the LPV simulation fails to account for the drop in generator torque from the relation

GeneratorTorque = Kreg2 × RotorSpeed^2.

Instead set `u(:,3) = GenTRated` which fixed the generator torque to its rated value.

```
GenTorque = out.GenTorque;
clf, plot(GenTorque.Time,GenTorque.Data,[0 Tf],GenTRated*[1 1],'r--');
```

```
title('Generator torque, N*m');
xlabel('Time, sec');
grid on;
```

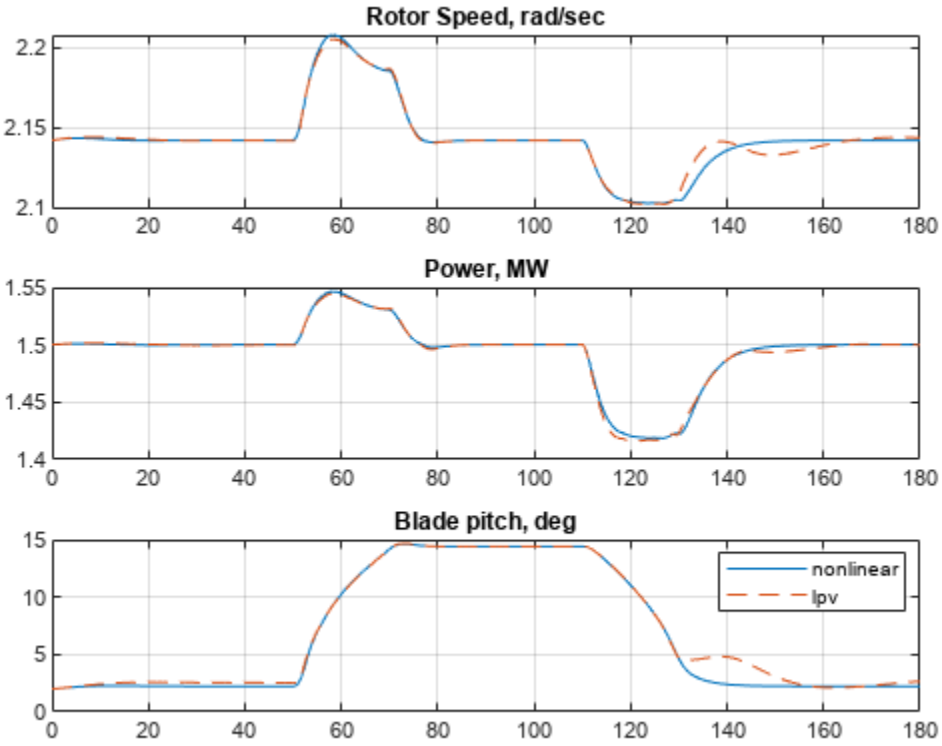


To improve accuracy, use the rotor speed data and relation above to estimate the generator torque data.

```
GenTorque = Kreg2 * ylpv(:,1).^2;
u(:,3) = min(GenTorque,GenTRated);
ylpv = lsim(Tlpv,u,t,xinit,pFcn);
```

Plot the responses.

```
clf
RotorSpeed = out.RotorSpeed;
subplot(311), plot(RotorSpeed.Time,RotorSpeed.Data,t,ylpv(:,1),'--')
title('Rotor Speed, rad/sec');
grid on;
Power = out.Power;
subplot(312), plot(Power.Time,Power.Data/1e6,t,ylpv(:,2)/1e6,'--')
title('Power, MW');
grid on;
BladePitch = out.BladePitch;
subplot(313), plot(BladePitch.Time,BladePitch.Data,t,ylpv(:,3),'--')
title('Blade pitch, deg');
grid on;
legend('nonlinear','lpv','location','Best')
```



The LPV simulation now closely matches its nonlinear counterpart.

Close the models.

```
bdclose('WindTurbineClosedLoop')
bdclose('WindTurbineOpenLoop')
```

References

[1] Malcolm, D.J. and A.C. Hansen. “WindPACT Turbine Rotor Design Study: June 2000–June 2002 (Revised).” National Renewable Energy Laboratory, 2006 NREL/SR-500-32495. https://www.nrel.gov/docs/fy06osti/32495.pdf.

[2] Rinker, Jennifer and Dykes, Katherine. "WindPACT Reference Wind Turbines." National Renewable Energy Laboratory, 2018 NREL/TP-5000-67667. https://www.nrel.gov/docs/fy18osti/67667.pdf.

See Also

`ssInterpolant` | `sample` | `linearize` | `ndgrid`

Related Examples

- “LTV and LPV Modeling” on page 1-26

- “Using LTV and LPV Models in MATLAB and Simulink” on page 1-32
- “Approximate Nonlinear Aircraft Pitch Dynamics Using LPV Model” on page 1-68
- “LPV Model of Magnetic Levitation Model from Batch Linearization Results” on page 1-133
- “Control Design for Spinning Disks” on page 1-85
- “LTV Model of Two-Link Robot” on page 1-93

LPV Model of Engine Throttle

This example shows how to model an engine throttle behavior as an LPV model with state offsets $\dot{x}_0(p)$ to account for the nonlinearity. You also simulate the results of the LPV model and compare the results with nonlinear simulation.

The throttle controls the air mass flow into the intake manifold of an engine. The throttle body contains a butterfly valve that opens when the driver presses down on the accelerator pedal. This lets more air enter the cylinders and causes the engine to produce more torque. The butterfly valve is modeled as a mass-spring-damper system. The amount of rotation of the valve is limited between 15 degrees and 90 degrees.

For more information about the throttle model, see “Estimate Model Parameter Values (GUI)” (Simulink Design Optimization).

LPV Model

The throttle dynamics are defined in the function `throttleLPV.m` provided with this example.

Use `lpvss` to create the model. This model is parameterized by the throttle angle, which is the first state of the model.

```
c0 = 50;
k0 = 120;
K0 = 1e6;
b0 = 4e4;
Ts = 0;
x0 = [15;0];
DF = @(t,p) throttleLPV(p,c0,k0,b0,K0);
sysp = lpvss('p',DF,Ts,0,15);
```

LPV Simulation

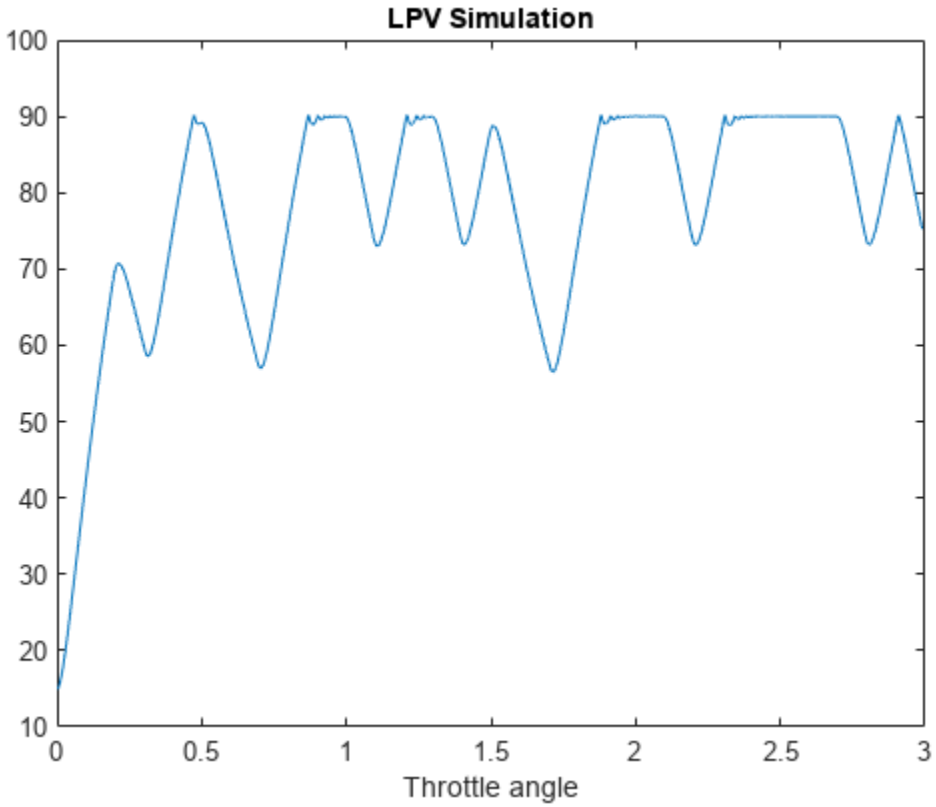
Generate open-loop response to an input along a parameter trajectory.

For this model, specify $p(t)$ implicitly as a function $F(t, x, u)$ of time t, input u, and state x. Here $p(t)$ is the throttle angle (first state).

```
load ThrottleInputData.mat
t2 = linspace(t(1),t(end),2000);
u2 = interp1(t,u,t2);
p = @(t,x,u)x(1);
yLPV = lsim(sysp,u2,t2,x0,p);
```

Plot the response.

```
plot(t2,yLPV)
title('LPV Simulation')
xlabel('Throttle angle')
```



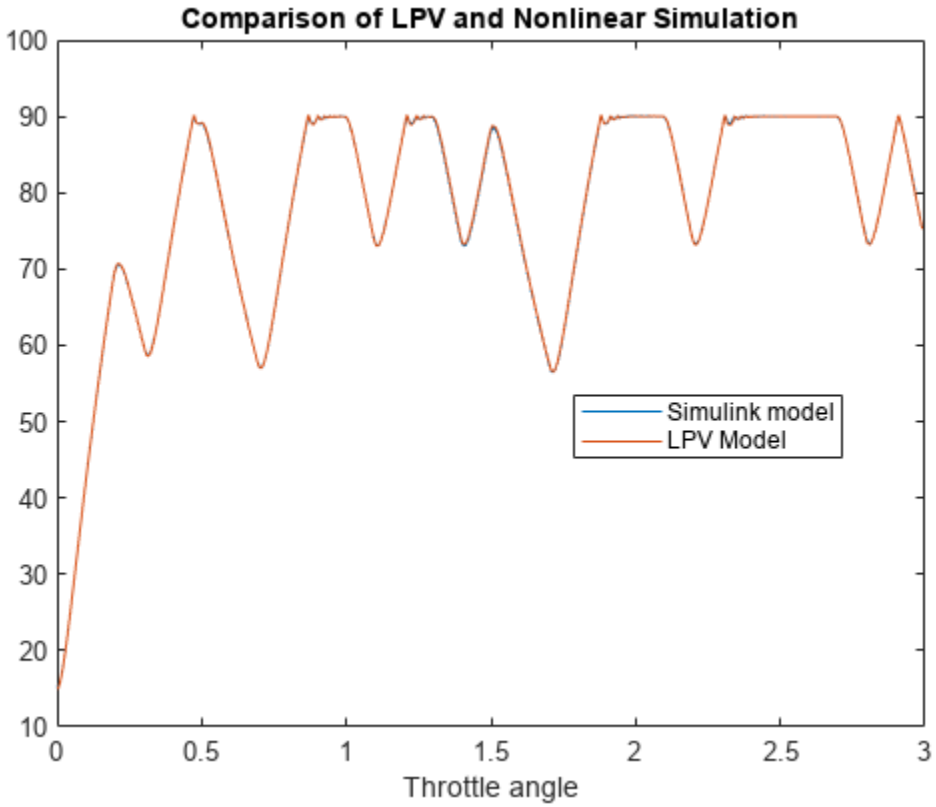
Nonlinear Simulation

Simulate the nonlinear Simulink® model.

```
open_system('throttleNLModel')
ysl = sim('throttleNLModel');
ysl = interp1(ysl.ScopeData(:,1),ysl.ScopeData(:,2),t);
```

Compare the simulation results for the LPV and nonlinear models.

```
figure(1)
clf
plot(t,ysl,t2,yLPV)
legend('Simulink model','LPV Model','location','Best')
title('Comparison of LPV and Nonlinear Simulation')
xlabel('Throttle angle')
```



The LPV model simulation the nonlinear response well. This is because the model uses the actual trajectory for scheduling.

Close the model.

```
bdclose('throttleNLModel')
```

See Also

`lpvss` | `sample` | `step` | `RespConfig`

Related Examples

- “LTV and LPV Modeling” on page 1-26
- “Using LTV and LPV Models in MATLAB and Simulink” on page 1-32
- “LPV Model of Bouncing Ball” on page 1-103

Model Creation

- “Transfer Functions” on page 2-2
- “State-Space Models” on page 2-5
- “Frequency Response Data (FRD) Models” on page 2-8
- “Proportional-Integral-Derivative (PID) Controllers” on page 2-11
- “Two-Degree-of-Freedom PID Controllers” on page 2-13
- “Discrete-Time Numeric Models” on page 2-18
- “Discrete-Time Proportional-Integral-Derivative (PID) Controllers” on page 2-19
- “MIMO Transfer Functions” on page 2-22
- “MIMO State-Space Models” on page 2-24
- “MIMO Frequency Response Data Models” on page 2-28
- “Select Input/Output Pairs in MIMO Models” on page 2-30
- “Time Delays in Linear Systems” on page 2-31
- “Closing Feedback Loops with Time Delays” on page 2-35
- “Time-Delay Approximation” on page 2-37
- “Time-Delay Approximation in Continuous-Time Open-Loop Model” on page 2-39
- “Time-Delay Approximation in Continuous-Time Closed-Loop Model” on page 2-43
- “Approximate Different Delays with Different Approximation Orders” on page 2-47
- “Convert Time Delay in Discrete-Time Model to Factors of 1/z” on page 2-50
- “Frequency Response Data (FRD) Model with Time Delay” on page 2-53
- “Internal Delays” on page 2-55
- “Tunable Low-Pass Filter” on page 2-59
- “Create Tunable Second-Order Filter” on page 2-60
- “Create State-Space Model with Both Fixed and Tunable Parameters” on page 2-62
- “Control System with Tunable Components” on page 2-63
- “Control System with Multichannel Analysis Points” on page 2-65
- “Mark Signals of Interest for Control System Analysis and Design” on page 2-68
- “Model Arrays” on page 2-76
- “Select Models from Array” on page 2-79
- “Query Array Size and Characteristics” on page 2-81
- “Linear Parameter-Varying Models” on page 2-83
- “Using LTI Arrays for Simulating Multi-Mode Dynamics” on page 2-89
- “Creating Discrete-Time Models” on page 2-94
- “Creating Continuous-Time Models” on page 2-97
- “Specifying Time Delays” on page 2-103

Transfer Functions

Transfer Function Representations

Control System Toolbox software supports transfer functions that are continuous-time or discrete-time, and SISO or MIMO. You can also have time delays in your transfer function representation.

A SISO continuous-time transfer function is expressed as the ratio:

$$G(s) = \frac{N(s)}{D(s)},$$

of polynomials $N(s)$ and $D(s)$, called the numerator and denominator polynomials, respectively.

You can represent linear systems as transfer functions in polynomial or factorized (zero-pole-gain) form. For example, the polynomial-form transfer function:

$$G(s) = \frac{s^2 - 3s - 4}{s^2 + 5s + 6}$$

can be rewritten in factorized form as:

$$G(s) = \frac{(s+1)(s-4)}{(s+2)(s+3)}.$$

The `tf` model object represents transfer functions in polynomial form. The `zpk` model object represents transfer functions in factorized form.

MIMO transfer functions are arrays of SISO transfer functions. For example:

$$G(s) = \begin{bmatrix} \frac{s-3}{s+4} \\ \frac{s+1}{s+2} \end{bmatrix}$$

is a one-input, two output transfer function.

Commands for Creating Transfer Functions

Use the commands described in the following table to create transfer functions.

Command	Description
`tf`	Create `tf` objects representing continuous-time or discrete-time transfer functions in polynomial form.
`zpk`	Create `zpk` objects representing continuous-time or discrete-time transfer functions in zero-pole-gain (factorized) form.
`filt`	Create `tf` objects representing discrete-time transfer functions using digital signal processing (DSP) convention.

Create Transfer Function Using Numerator and Denominator Coefficients

This example shows how to create continuous-time single-input, single-output (SISO) transfer functions from their numerator and denominator coefficients using `tf`.

Create the transfer function $G(s) = \frac{s}{s^2 + 3s + 2}$:

```
num = [1 0];
den = [1 3 2];
G = tf(num,den);
```

`num` and `den` are the numerator and denominator polynomial coefficients in descending powers of s. For example, `den = [1 3 2]` represents the denominator polynomial $s^2 + 3s + 2$.

`G` is a `tf` model object, which is a data container for representing transfer functions in polynomial form.

Tip Alternatively, you can specify the transfer function $G(s)$ as an expression in s:

1. Create a transfer function model for the variable s.

   ```
   s = tf('s');
   ```

2. Specify $G(s)$ as a ratio of polynomials in s.

   ```
   G = s/(s^2 + 3*s + 2);
   ```

Create Transfer Function Model Using Zeros, Poles, and Gain

This example shows how to create single-input, single-output (SISO) transfer functions in factored form using `zpk`.

Create the factored transfer function $G(s) = 5\frac{s}{(s+1+i)(s+1-i)(s+2)}$:

```
Z = [0];
P = [-1-1i -1+1i -2];
K = 5;
G = zpk(Z,P,K);
```

`Z` and `P` are the zeros and poles (the roots of the numerator and denominator, respectively). `K` is the gain of the factored form. For example, $G(s)$ has a real pole at $s = -2$ and a pair of complex poles at $s = -1 \pm i$. The vector `P = [-1-1i -1+1i -2]` specifies these pole locations.

`G` is a `zpk` model object, which is a data container for representing transfer functions in zero-pole-gain (factorized) form.

See Also

`tf` | `zpk` | `filt`

Related Examples

- “MIMO Transfer Functions” on page 2-22
- “State-Space Models” on page 2-5
- “Discrete-Time Numeric Models” on page 2-18

More About

- “What Are Model Objects?” on page 1-2
- “Store and Retrieve Model Data” on page 3-2

External Websites

- Transfer Function Analysis of Dynamic Systems (MathWorks Teaching Resources)

State-Space Models

State-Space Model Representations

State-space models rely on linear differential equations or difference equations to describe system dynamics. Control System Toolbox software supports SISO or MIMO state-space models in continuous or discrete time. State-space models can include time delays. You can represent state-space models in either explicit or descriptor (implicit) form.

State-space models can result from:

- Linearizing a set of ordinary differential equations that represent a physical model of the system.
- State-space model identification using System Identification Toolbox software.
- State-space realization of transfer functions. (See “Conversion Between Model Types” on page 5-2 for more information.)

Use `ss` model objects to represent state-space models.

Explicit State-Space Models

Explicit continuous-time state-space models have the following form:

$$\frac{dx}{dt} = Ax + Bu$$
$$y = Cx + Du$$

where x is the state vector. u is the input vector, and y is the output vector. A, B, C, and D are the state-space matrices that express the system dynamics.

A discrete-time explicit state-space model takes the following form:

$$x[n+1] = Ax[n] + Bu[n]$$
$$y[n] = Cx[n] + Du[n]$$

where the vectors $x[n]$, $u[n]$, and $y[n]$ are the state, input, and output vectors for the nth sample.

Descriptor (Implicit) State-Space Models

A descriptor state-space model is a generalized form of state-space model. In continuous time, a descriptor state-space model takes the following form:

$$E\frac{dx}{dt} = Ax + Bu$$
$$y = Cx + Du$$

where x is the state vector. u is the input vector, and y is the output vector. A, B, C, D, and E are the state-space matrices.

Commands for Creating State-Space Models

Use the commands described in the following table to create state-space models.

Command	Description
`ss`	Create explicit state-space model.
`dss`	Create descriptor (implicit) state-space model.
`delayss`	Create state-space models with specified time delays.

Create State-Space Model From Matrices

This example shows how to create a continuous-time single-input, single-output (SISO) state-space model from state-space matrices using `ss`.

Create a model of an electric motor where the state-space equations are:

$$\frac{dx}{dt} = Ax + Bu$$
$$y = Cx + Du$$

where the state variables are the angular position θ and angular velocity $d\theta/dt$:

$$x = \begin{bmatrix} \theta \\ \frac{d\theta}{dt} \end{bmatrix},$$

u is the electric current, the output y is the angular velocity, and the state-space matrices are:

$$A = \begin{bmatrix} 0 & 1 \\ -5 & -2 \end{bmatrix}, \quad B = \begin{bmatrix} 0 \\ 3 \end{bmatrix}, \quad C = [0\ 1], \quad D = [0].$$

To create this model, enter:

```
A = [0 1;-5 -2];
B = [0;3];
C = [0 1];
D = 0;
sys = ss(A,B,C,D);
```

`sys` is an `ss` model object, which is a data container for representing state-space models.

Tip To represent a system of the form:

$$E\frac{dx}{dt} = Ax + Bu$$
$$y = Cx + Du$$

use `dss`. This command creates a `ss` model with a nonempty `E` matrix, also called a descriptor state-space model. See “MIMO Descriptor State-Space Models” on page 2-25 for an example.

See Also

`ss` | `dss` | `delayss`

Related Examples

- “MIMO State-Space Models” on page 2-24
- “Transfer Functions” on page 2-2
- “Discrete-Time Numeric Models” on page 2-18

More About

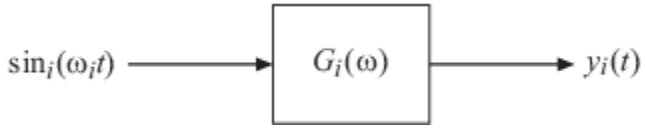
- “What Are Model Objects?” on page 1-2
- “Store and Retrieve Model Data” on page 3-2

Frequency Response Data (FRD) Models

Frequency Response Data

In the Control System Toolbox software, you can use `frd` models to store, manipulate, and analyze frequency response data. An `frd` model stores a vector of frequency points with the corresponding complex frequency response data you obtain either through simulations or experimentally.

For example, suppose you measure frequency response data for the SISO system you want to model. You can measure such data by driving the system with a sine wave at a set of frequencies $\omega_1, \omega_2, ,...,\omega_n$, as shown:



At steady state, the measured response $y_i(t)$ to the driving signal at each frequency ω_i takes the following form:

$$y_i(t) = a\sin(\omega_i t + b), \quad i = 1, \ldots, n\,.$$

The measurement yields the complex frequency response G at each input frequency:

$$G(j\omega_i) = ae^{jb}, \quad i = 1, \ldots, n\,.$$

You can do most frequency-domain analysis tasks on `frd` models, but you cannot perform time-domain simulations with them. For information on frequency response analysis of linear systems, see Chapter 8 of [1].

Commands for Creating FRD Models

Use the following commands to create FRD models.

Command	Description
`frd`	Create `frd` objects from frequency response data.
`frestimate`	Create `frd` objects by estimating the frequency response of a Simulink model. This approach requires Simulink Control Design software. See “Offline Frequency Response Estimation” (Simulink Control Design) for more information.

Create Frequency Response Model from Data

This example shows how to create a single-input, single-output (SISO) frequency-response model using `frd`.

A frequency-response model stores a vector of frequency points with corresponding complex frequency response data you obtain either through simulations or experimentally. Therefore, if you measure the frequency response of your system at a set of test frequencies, you can use the data to create a frequency response model.

Load the frequency response data in `AnalyzerData.mat`.

```
load AnalyzerData
```

This command loads the data into the MATLAB® workspace as the column vectors `freq` and `resp`. The variables `freq` and `resp` contain 256 test frequencies and the corresponding complex-valued frequency response points, respectively.

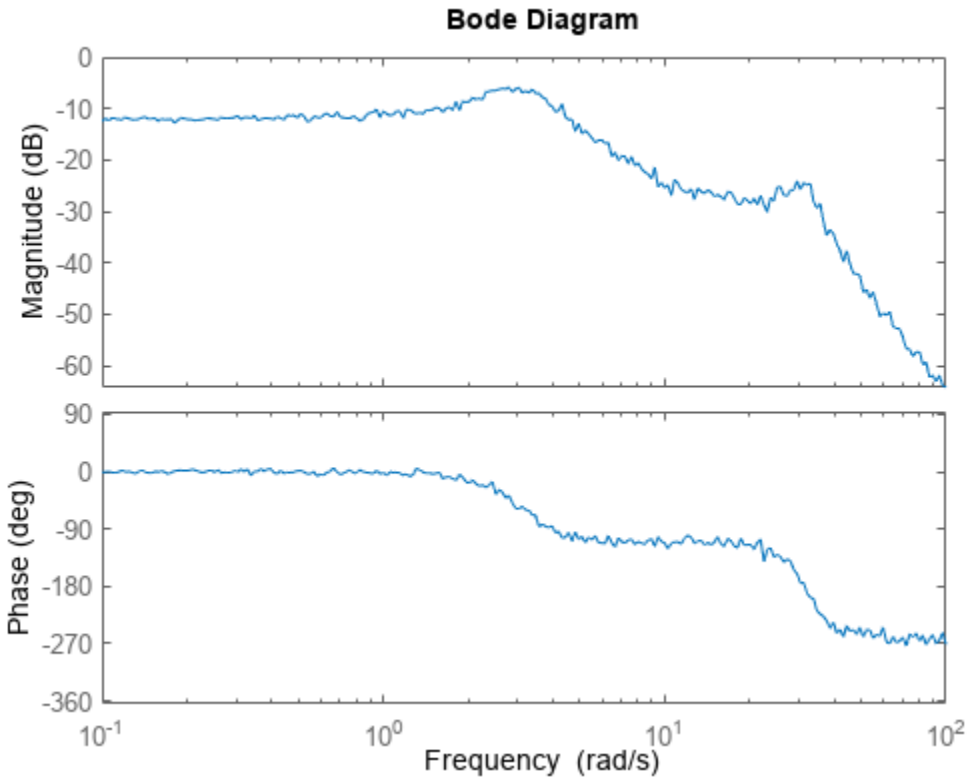
Create a frequency response model.

```
sys = frd(resp,freq);
```

`sys` is an `frd` model object that represents frequency response data.

You can use `frd` models with many frequency-domain analysis commands. For example, visualize the frequency response data using `bode`.

```
bode(sys)
```



By default, the `frd` function assumes that the frequencies are in radians/second. To specify different frequency units, use the `TimeUnit` and `FrequencyUnit` properties of the `frd` model object. For example, set the frequency units to radians/minute.

```
sys = frd(resp,freq,...
    "TimeUnit","min","FrequencyUnit","rad/TimeUnit");
```

See Also

`frd` | `frestimate`

Related Examples

- “MIMO Frequency Response Data Models” on page 2-28
- “Discrete-Time Numeric Models” on page 2-18

More About

- “What Are Model Objects?” on page 1-2
- “Store and Retrieve Model Data” on page 3-2

Proportional-Integral-Derivative (PID) Controllers

You can represent PID controllers using the specialized model objects `pid` and `pidstd`. This topic describes the representation of PID controllers in MATLAB. For information about automatic PID controller tuning, see “PID Controller Tuning”.

Continuous-Time PID Controller Representations

You can represent continuous-time Proportional-Integral-Derivative (PID) controllers in either parallel or standard form. The two forms differ in the parameters used to express the proportional, integral, and derivative actions and the filter on the derivative term, as shown in the following table.

Form	Formula
Parallel (`pid` object)	$$C = K_p + \frac{K_i}{s} + \frac{K_d s}{T_f s + 1},$$ where: • K_p = proportional gain • K_i = integrator gain • K_d = derivative gain • T_f = derivative filter time
Standard (`pidstd` object)	$$C = K_p\left(1 + \frac{1}{T_i s} + \frac{T_d s}{\frac{T_d}{N}s + 1}\right),$$ where: • K_p = proportional gain • T_i = integrator time • T_d = derivative time • N = derivative filter divisor

Use a controller form that is convenient for your application. For example, if you want to express the integrator and derivative actions in terms of time constants, use standard form.

For information on representing PID Controllers in discrete time, see “Discrete-Time Proportional-Integral-Derivative (PID) Controllers” on page 2-19

Create Continuous-Time Parallel-Form PID Controller

This example shows how to create a continuous-time Proportional-Integral-Derivative (PID) controller in parallel form using `pid`.

Create the following parallel-form PID controller: $C = 29.5 + \frac{26.2}{s} - \frac{4.3s}{0.06s + 1}$.

```
Kp = 29.5;
Ki = 26.2;
```

```
Kd = 4.3;
Tf = 0.06;
C = pid(Kp,Ki,Kd,Tf)
```

C is a `pid` model object, which is a data container for representing parallel-form PID controllers. For more examples of how to create PID controllers, see the `pid` reference page.

Create Continuous-Time Standard-Form PID Controller

This example shows how to create a continuous-time Proportional-Integral-Derivative (PID) controller in standard form using `pidstd`.

Create the following standard-form PID controller: $C = 29.5\left(1 + \frac{1}{1.13s} + \frac{0.15s}{\frac{0.15}{2.3}s + 1}\right)$.

```
Kp = 29.5;
Ti = 1.13;
Td = 0.15;
N = 2.3;
C = pidstd(Kp,Ti,Td,N)
```

C is a `pidstd` model object, which is a data container for representing standard-form PID controllers. For more examples of how to create standard-form PID controllers, see the `pidstd` reference page.

See Also

`pid` | `pidstd` | `pidtune` | `pidTuner`

Related Examples

- “Transfer Functions” on page 2-2
- “Discrete-Time Proportional-Integral-Derivative (PID) Controllers” on page 2-19
- “Two-Degree-of-Freedom PID Controllers” on page 2-13

More About

- “What Are Model Objects?” on page 1-2
- “Store and Retrieve Model Data” on page 3-2

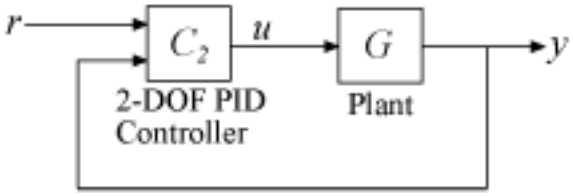
Two-Degree-of-Freedom PID Controllers

Two-degree-of-freedom (2-DOF) PID controllers include setpoint weighting on the proportional and derivative terms. A 2-DOF PID controller is capable of fast disturbance rejection without significant increase of overshoot in setpoint tracking. 2-DOF PID controllers are also useful to mitigate the influence of changes in the reference signal on the control signal.

You can represent PID controllers using the specialized model objects `pid2` and `pidstd2`. This topic describes the representation of 2-DOF PID controllers in MATLAB. For information about automatic PID controller tuning, see “PID Controller Tuning”.

Continuous-Time 2-DOF PID Controller Representations

This illustration shows a typical control architecture using a 2-DOF PID controller.



The relationship between the 2-DOF controller’s output (u) and its two inputs (r and y) can be represented in either parallel or standard form. The two forms differ in the parameters used to express the proportional, integral, and derivative actions of the controller, as expressed in the following table.

Form	Formula
Parallel (`pid2` object)	$$u = K_p(br - y) + \frac{K_i}{s}(r - y) + \frac{K_d s}{T_f s + 1}(cr - y)\,.$$ In this representation: • K_p = proportional gain • K_i = integrator gain • K_d = derivative gain • T_f = derivative filter time • b = setpoint weight on proportional term • c = setpoint weight on derivative term

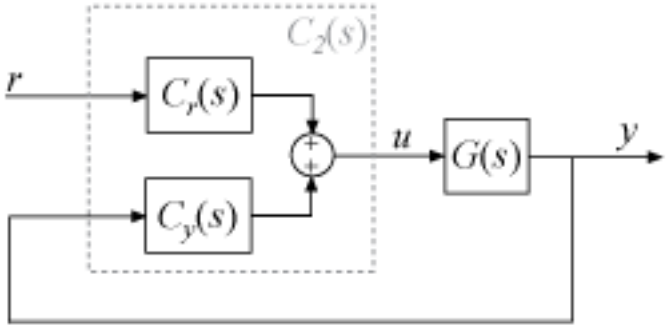
Form	Formula
Standard (`pidstd2` object)	$$u = K_p\left[(br - y) + \frac{1}{T_i s}(r - y) + \frac{T_d s}{\frac{T_d}{N}s + 1}(cr - y)\right].$$ In this representation: • K_p = proportional gain • T_i = integrator time • T_d = derivative time • N = derivative filter divisor • b = setpoint weight on proportional term • c = setpoint weight on derivative term

Use a controller form that is convenient for your application. For instance, if you want to express the integrator and derivative actions in terms of time constants, use standard form. For examples showing how to create parallel-form and standard-form controllers, see the `pid2` and `pidstd2` reference pages, respectively.

For information on representing PID Controllers in discrete time, see “Discrete-Time Proportional-Integral-Derivative (PID) Controllers” on page 2-19.

2-DOF Control Architectures

The 2-DOF PID controller is a two-input, one output controller of the form $C_2(s)$, as shown in the following figure. The transfer function from each input to the output is itself a PID controller.



Each of the components $C_r(s)$ and $C_y(s)$ is a PID controller, with different weights on the proportional and derivative terms. For example, in continuous time, these components are given by:

$$C_r(s) = bK_p + \frac{K_i}{s} + \frac{cK_d s}{T_f s + 1},$$

$$C_y(s) = -\left[K_p + \frac{K_i}{s} + \frac{K_d s}{T_f s + 1}\right].$$

You can access these components by converting the PID controller into a two-input, one-output transfer function. For example, suppose that `C2` is a 2-DOF PID controller, stored as a `pid2` object.

```
C2tf = tf(C2);
Cr = C2tf(1);
Cy = C2tf(2);
```

$C_r(s)$ is the transfer function from the first input of `C2` to the output. Similarly, $C_y(s)$ is the transfer function from the second input of `C2` to the output.

Suppose that `G` is a dynamic system model, such as a `zpk` model, representing the plant. Build the closed-loop transfer function from r to y. Note that the $C_y(s)$ loop has positive feedback, by the definition of $C_y(s)$.

```
T = Cr*feedback(G,Cy,+1)
```

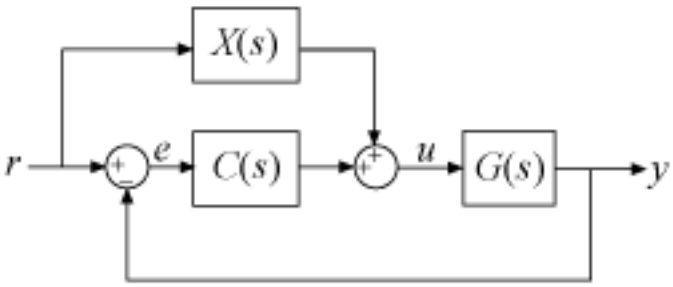
Alternatively, use the `connect` command to build an equivalent closed-loop system directly with the 2-DOF controller `C2`. To do so, set the `InputName` and `OutputName` properties of `G` and `C2`.

```
G.InputName = 'u';
G.OutputName = 'y';
C2.Inputname = {'r','y'};
C2.OutputName = 'u';
T = connect(G,C2,'r','y');
```

There are other configurations in which you can decompose a 2-DOF PID controller into SISO components. For particular choices of $C(s)$ and $X(s)$, each of the following configurations is equivalent to the 2-DOF architecture with $C_2(s)$. You can obtain $C(s)$ and $X(s)$ for each of these configurations using the `getComponents` command.

Feedforward

In the feedforward configuration, the 2-DOF PID controller is decomposed into a conventional SISO PID controller that takes the error signal as its input, and a feedforward controller.



For a continuous-time, parallel-form 2-DOF PID controller, the components are given by:

$$C(s) = K_p + \frac{K_i}{s} + \frac{K_d s}{T_f s + 1},$$

$$X(s) = (b-1)K_p + \frac{(c-1)K_d s}{T_f s + 1}.$$

Access these components using `getComponents`.

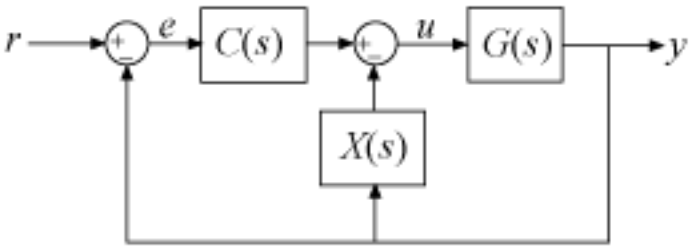
```
[C,X] = getComponents(C2,'feedforward');
```

The following command constructs the closed-loop system from r to y for the feedforward configuration.

```
T = G*(C+X)*feedback(1,G*C);
```

Feedback

In the feedback configuration, the 2-DOF PID controller is decomposed into a conventional SISO PID controller and a feedback controller.



For a continuous-time, parallel-form 2-DOF PID controller, the components are given by:

$$C(s) = bK_p + \frac{K_i}{s} + \frac{cK_d s}{T_f s + 1},$$

$$X(s) = (1 - b)K_p + \frac{(1 - c)K_d s}{T_f s + 1}.$$

Access these components using `getComponents`.

```
[C,X] = getComponents(C2,'feedback');
```

The following command constructs the closed-loop system from r to y for the feedback configuration.

```
T = G*C*feedback(1,G*(C+X));
```

Filter

In the filter configuration, the 2-DOF PID controller is decomposed into a conventional SISO PID controller and a prefilter on the reference signal.

For a continuous-time, parallel-form 2-DOF PID controller, the components are given by:

$$C(s) = K_p + \frac{K_i}{s} + \frac{K_d s}{T_f s + 1},$$

$$X(s) = \frac{\left(bK_pT_f + cK_d\right)s^2 + \left(bK_p + K_iT_f\right)s + K_i}{\left(K_pT_f + K_d\right)s^2 + \left(K_p + K_iT_f\right)s + K_i}.$$

The filter $X(s)$ can also be expressed as the ratio: $-[C_r(s)/C_y(s)]$.

The following command constructs the closed-loop system from r to y for the filter configuration.

```
T = X*feedback(G*C,1);
```

For an example illustrating the decomposition of a 2-DOF PID controller into these configurations, see “Decompose a 2-DOF PID Controller into SISO Components” on page 5-7.

The formulas shown above pertain to continuous-time, parallel-form controllers. Standard-form controllers and controllers in discrete time can be decomposed into analogous configurations. The `getComponents` command works on all 2-DOF PID controller objects.

See Also

`getComponents` | `pid2` | `pidstd2` | `pidtune` | `pidTuner`

Related Examples

- “Discrete-Time Proportional-Integral-Derivative (PID) Controllers” on page 2-19
- “Proportional-Integral-Derivative (PID) Controllers” on page 2-11

More About

- “What Are Model Objects?” on page 1-2
- “Store and Retrieve Model Data” on page 3-2

Discrete-Time Numeric Models

Create Discrete-Time Transfer Function Model

This example shows how to create a discrete-time transfer function model using `tf`.

Create the transfer function $G(z) = \frac{z}{z^2 - 2z - 6}$ with a sample time of 0.1 s.

```
num = [1 0];
den = [1 -2 -6];
Ts = 0.1;
G = tf(num,den,Ts)
```

`num` and `den` are the numerator and denominator polynomial coefficients in descending powers of z. `G` is a `tf` model object.

The sample time is stored in the `Ts` property of `G`. Access the sample time `Ts`, using dot notation:

```
G.Ts
```

Other Model Types in Discrete Time Representations

Create discrete-time `zpk`, `ss`, and `frd` models in a similar way to discrete-time transfer functions, by appending a sample time to the input arguments. For examples, see the reference pages for those commands.

See Also

`tf` | `zpk` | `ss` | `frd`

More About

- “What Are Model Objects?” on page 1-2

Discrete-Time Proportional-Integral-Derivative (PID) Controllers

All the PID controller object types, `pid`, `pidstd`, `pid2`, and `pidstd2`, can represent PID controllers in discrete time.

Discrete-Time PID Controller Representations

Discrete-time PID controllers are expressed by the following formulas.

Form	Formula
Parallel (`pid`)	$$C = K_p + K_i IF(z) + \frac{K_d}{T_f + DF(z)},$$ where: • K_p = proportional gain • K_i = integrator gain • K_d = derivative gain • T_f = derivative filter time
Standard (`pidstd`)	$$C = K_p\left(1 + \frac{1}{T_i}IF(z) + \frac{T_d}{\frac{T_d}{N} + DF(z)}\right),$$ where: • K_p = proportional gain • T_i = integrator time • T_d = derivative time • N = derivative filter divisor
2-DOF Parallel (`pid2`)	The relationship between the 2-DOF controller's output (u) and its two inputs (r and y) is: $$u = K_p(br - y) + K_i IF(z)(r - y) + \frac{K_d}{T_f + DF(z)}(cr - y).$$ In this representation: • K_p = proportional gain • K_i = integrator gain • K_d = derivative gain • T_f = derivative filter time • b = setpoint weight on proportional term • c = setpoint weight on derivative term

Form	Formula
2-DOF Standard (`pidstd2` object)	$$u = K_p\left[(br - y) + \frac{1}{T_i}IF(z)(r - y) + \frac{T_d}{\frac{T_d}{N} + DF(z)}(cr - y)\right].$$ In this representation: • K_p = proportional gain • T_i = integrator time • T_d = derivative time • N = derivative filter divisor • b = setpoint weight on proportional term • c = setpoint weight on derivative term

In all of these expressions, $IF(z)$ and $DF(z)$ are the discrete integrator formulas for the integrator and derivative filter, respectively. Use the `IFormula` and `DFormula` properties of the controller objects to set the $IF(z)$ and $DF(z)$ formulas. The next table shows available formulas for $IF(z)$ and $DF(z)$. T_s is the sample time.

`IFormula` or `DFormula`	IF(z) or DF(z)
`ForwardEuler` (default)	$\frac{T_s}{z-1}$
`BackwardEuler`	$\frac{T_s z}{z-1}$
`Trapezoidal`	$\frac{T_s}{2}\frac{z+1}{z-1}$

If you do not specify a value for `IFormula`, `DFormula`, or both when you create the controller object, `ForwardEuler` is used by default. For more information about setting and changing the discrete integrator formulas, see the reference pages for the controller objects, `pid`, `pidstd`, `pid2`, and `pidstd2`.

Create Discrete-Time Standard-Form PID Controller

This example shows how to create a standard-form discrete-time Proportional-Integral-Derivative (PID) controller that has K_p = 29.5, T_i = 1.13, T_d = 0.15 N = 2.3, and sample time T_s 0.1 :

```
C = pidstd(29.5,1.13,0.15,2.3,0.1,...
             'IFormula','Trapezoidal','DFormula','BackwardEuler')
```

This command creates a `pidstd` model with $IF(z) = \frac{T_s}{2}\frac{z+1}{z-1}$ and $DF(z) = \frac{T_s z}{z-1}$.

You can set the discrete integrator formulas for a parallel-form controller in the same way, using `pid`.

Discrete-Time 2-DOF PI Controller in Standard Form

Create a discrete-time 2-DOF PI controller in standard form, using the trapezoidal discretization formula. Specify the formula using `Name,Value` syntax.

```
Kp = 1;
Ti = 2.4;
Td = 0;
N = Inf;
b = 0.5;
c = 0;
Ts = 0.1;
C2 = pidstd2(Kp,Ti,Td,N,b,c,Ts,'IFormula','Trapezoidal')
```

```
C2 =

                       1     Ts*(z+1)
  u = Kp * [(b*r-y) + ---- * -------- * (r-y)]
                       Ti    2*(z-1)

  with Kp = 1, Ti = 2.4, b = 0.5, Ts = 0.1

Sample time: 0.1 seconds
Discrete-time 2-DOF PI controller in standard form
```

Setting `Td = 0` specifies a PI controller with no derivative term. As the display shows, the values of `N` and `c` are not used in this controller. The display also shows that the trapezoidal formula is used for the integrator.

See Also

`pid` | `pidstd` | `pid2` | `pidstd2`

Related Examples

- “Proportional-Integral-Derivative (PID) Controllers” on page 2-11
- “Two-Degree-of-Freedom PID Controllers” on page 2-13

More About

- “What Are Model Objects?” on page 1-2
- “Store and Retrieve Model Data” on page 3-2

MIMO Transfer Functions

MIMO transfer functions are two-dimensional arrays of elementary SISO transfer functions. There are two ways to specify MIMO transfer function models:

- Concatenation of SISO transfer function models
- Using `tf` with cell array arguments

Concatenation of SISO Models

Consider the following single-input, two-output transfer function.

$$H(s) = \begin{bmatrix} \dfrac{s-1}{s+1} \\ \dfrac{s+2}{s^2+4s+5} \end{bmatrix}.$$

You can specify $H(s)$ by concatenation of its SISO entries. For instance,

```
h11 = tf([1 -1],[1 1]);
h21 = tf([1 2],[1 4 5]);
```

or, equivalently,

```
s = tf('s')
h11 = (s-1)/(s+1);
h21 = (s+2)/(s^2+4*s+5);
```

can be concatenated to form $H(s)$.

```
H = [h11; h21]
```

This syntax mimics standard matrix concatenation and tends to be easier and more readable for MIMO systems with many inputs and/or outputs.

Tip Use `zpk` instead of `tf` to create MIMO transfer functions in factorized form on page 2-3.

Using the tf Function with Cell Arrays

Alternatively, to define MIMO transfer functions using `tf`, you need two cell arrays (say, `N` and `D`) to represent the sets of numerator and denominator polynomials, respectively. See “What Is a Cell Array?” for more details on cell arrays.

For example, for the rational transfer matrix $H(s)$, the two cell arrays `N` and `D` should contain the row-vector representations of the polynomial entries of

$$N(s) = \left[\frac{s-1}{s+2}\right], \quad D(s) = \left[\frac{s+1}{s^2+4s+5}\right].$$

You can specify this MIMO transfer matrix $H(s)$ by typing

```
N = {[1 -1];[1 2]};   % Cell array for N(s)
D = {[1 1];[1 4 5]}; % Cell array for D(s)
H = tf(N,D)

Transfer function from input to output...
      s - 1
 #1:  -----
      s + 1

          s + 2
 #2:  -------------
      s^2 + 4 s + 5
```

Notice that both `N` and `D` have the same dimensions as H. For a general MIMO transfer matrix $H(s)$, the cell array entries `N{i,j}` and `D{i,j}` should be row-vector representations of the numerator and denominator of $H_{ij}(s)$, the ijth entry of the transfer matrix $H(s)$.

See Also

`tf` | `zpk`

Related Examples

- “Transfer Functions” on page 2-2

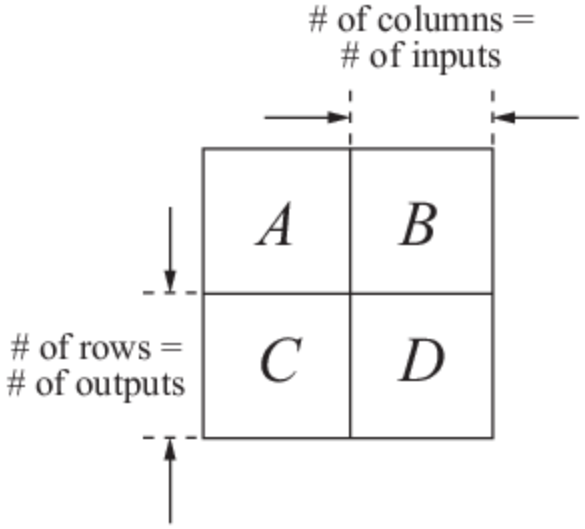
More About

- “What Are Model Objects?” on page 1-2
- “Store and Retrieve Model Data” on page 3-2

MIMO State-Space Models

MIMO Explicit State-Space Models

You create a MIMO state-space model in the same way as you create a SISO state-space model on page 2-6. The only difference between the SISO and MIMO cases is the dimensions of the state-space matrices. The dimensions of the B, C, and D matrices increase with the numbers of inputs and outputs as shown in the following illustration.



In this example, you create a state-space model for a rotating body with inertia tensor J, damping force F, and three axes of rotation, related as:

$$J\frac{d\omega}{dt} + F\omega = T$$
$$y = \omega .$$

The system input T is the driving torque. The output y is the vector of angular velocities of the rotating body.

To express this system in state-space form:

$$\frac{dx}{dt} = Ax + Bu$$
$$y = Cx + Du$$

rewrite it as:

$$\frac{d\omega}{dt} = -J^{-1}F\omega + J^{-1}T$$
$$y = \omega .$$

Then the state-space matrices are:

$$A = -J^{-1}F, \quad B = J^{-1}, \quad C = I, \quad D = 0.$$

To create this model, enter the following commands:

```
J = [8 -3 -3; -3 8 -3; -3 -3 8];
F = 0.2*eye(3);
A = -J\F;
B = inv(J);
C = eye(3);
D = 0;
sys_mimo = ss(A,B,C,D);
```

These commands assume that J is the inertia tensor of a cube rotating about its corner, and the damping force has magnitude 0.2.

`sys_mimo` is an `ss` model.

MIMO Descriptor State-Space Models

This example shows how to create a continuous-time descriptor (implicit) state-space model using `dss`.

This example uses the same rotating-body system shown in “MIMO Explicit State-Space Models” on page 2-24, where you inverted the inertia matrix J to obtain the value of the B matrix. If J is poorly-conditioned for inversion, you can instead use a descriptor (implicit) state-space model. A descriptor (implicit) state-space model is of the form:

$$E\frac{dx}{dt} = Ax + Bu$$
$$y = Cx + Du$$

Create a state-space model for a rotating body with inertia tensor J, damping force F, and three axes of rotation, related as:

$$J\frac{d\omega}{dt} + F\omega = T$$
$$y = \omega\,.$$

The system input T is the driving torque. The output y is the vector of angular velocities of the rotating body. You can write this system as a descriptor state-space model having the following state-space matrices:

$$A = -F, \quad B = I, \quad C = I, \quad D = 0, \quad E = J\,.$$

To create this system, enter:

```
J = [8 -3 -3; -3 8 -3; -3 -3 8];
F = 0.2*eye(3);
A = -F;
B = eye(3);
C = eye(3);
D = 0;
E = J;
sys_mimo = dss(A,B,C,D,E)
```

These commands assume that J is the inertia tensor of a cube rotating about its corner, and the damping force has magnitude 0.2.

`sys` is an `ss` model with a nonempty *E* matrix.

State-Space Model of Jet Transport Aircraft

This example shows how to build a MIMO model of a jet transport. Because the development of a physical model for a jet aircraft is lengthy, only the state-space equations are presented here. See any standard text in aviation for a more complete discussion of the physics behind aircraft flight.

The jet model during cruise flight at MACH = 0.8 and H = 40,000 ft. is

```
A = [-0.0558   -0.9968    0.0802    0.0415
      0.5980   -0.1150   -0.0318         0
     -3.0500    0.3880   -0.4650         0
           0    0.0805    1.0000         0];

B = [ 0.0073         0
     -0.4750    0.0077
      0.1530    0.1430
           0         0];

C = [0     1     0     0
     0     0     0     1];

D = [0     0
     0     0];
```

Use the following commands to specify this state-space model as an LTI object and attach names to the states, inputs, and outputs.

```
states = {'beta' 'yaw' 'roll' 'phi'};
inputs = {'rudder' 'aileron'};
outputs = {'yaw rate' 'bank angle'};

sys_mimo = ss(A,B,C,D,'statename',states,...
'inputname',inputs,...
'outputname',outputs);
```

You can display the LTI model by typing `sys_mimo`.

```
sys_mimo
```

```
a =
                beta          yaw         roll          phi
      beta   -0.0558      -0.9968       0.0802       0.0415
       yaw    0.598       -0.115      -0.0318            0
      roll    -3.05        0.388       -0.465            0
       phi        0       0.0805            1            0


b =
               rudder      aileron
      beta     0.0073            0
       yaw     -0.475       0.0077
      roll      0.153        0.143
       phi          0            0
```

```
c =
                      beta          yaw         roll          phi
     yaw rate            0            1            0            0
   bank angle            0            0            0            1


d =
                    rudder      aileron
     yaw rate            0            0
   bank angle            0            0

Continuous-time model.
```

The model has two inputs and two outputs. The units are radians for `beta` (sideslip angle) and `phi` (bank angle) and radians/sec for `yaw` (yaw rate) and `roll` (roll rate). The rudder and aileron deflections are in degrees.

As in the SISO case, use `tf` to derive the transfer function representation.

```
tf(sys_mimo)
```

```
Transfer function from input "rudder" to output...
               -0.475 s^3 - 0.2479 s^2 - 0.1187 s - 0.05633
 yaw rate:  ---------------------------------------------------
            s^4 + 0.6358 s^3 + 0.9389 s^2 + 0.5116 s + 0.003674

                         0.1148 s^2 - 0.2004 s - 1.373
 bank angle:  ---------------------------------------------------
              s^4 + 0.6358 s^3 + 0.9389 s^2 + 0.5116 s + 0.003674

Transfer function from input "aileron" to output...
            0.0077 s^3 - 0.0005372 s^2 + 0.008688 s + 0.004523
 yaw rate:  ---------------------------------------------------
            s^4 + 0.6358 s^3 + 0.9389 s^2 + 0.5116 s + 0.003674

                        0.1436 s^2 + 0.02737 s + 0.1104
 bank angle:  ---------------------------------------------------
              s^4 + 0.6358 s^3 + 0.9389 s^2 + 0.5116 s + 0.003674
```

See Also

`ss`

Related Examples

- “State-Space Models” on page 2-5

More About

- “What Are Model Objects?” on page 1-2
- “Store and Retrieve Model Data” on page 3-2

MIMO Frequency Response Data Models

This example shows how to create a MIMO frequency-response model using an `frd` object.

Frequency response data for a MIMO system includes a vector of complex response data for each of the input/output (I/O) pair of the system. Thus, if you measure the frequency response of each I/O pair of your system at a set of test frequencies, you can use the data to create a frequency response model.

For this example, load the frequency response data in `AnalyzerDataMIMO.mat`.

```
load AnalyzerDataMIMO H11 H12 H21 H22 freq
```

This command loads the data into the MATLAB® workspace as five column vectors `H11`, `H12`, `H21`, `H22`, and `freq`. The vector `freq` contains 100 test frequencies. The other four vectors contain the corresponding complex-valued frequency response of each I/O pair of a two-input, two-output system.

Organize the data into a three-dimensional array.

```
Hresp = zeros(2,2,length(freq));
Hresp(1,1,:) = H11;
Hresp(1,2,:) = H12;
Hresp(2,1,:) = H21;
Hresp(2,2,:) = H22;
```

The dimensions of `Hresp` are the number of outputs, number of inputs, and the number of frequencies for which there is response data. `Hresp(i,j,:)` contains the frequency response from input `j` to output `i`.

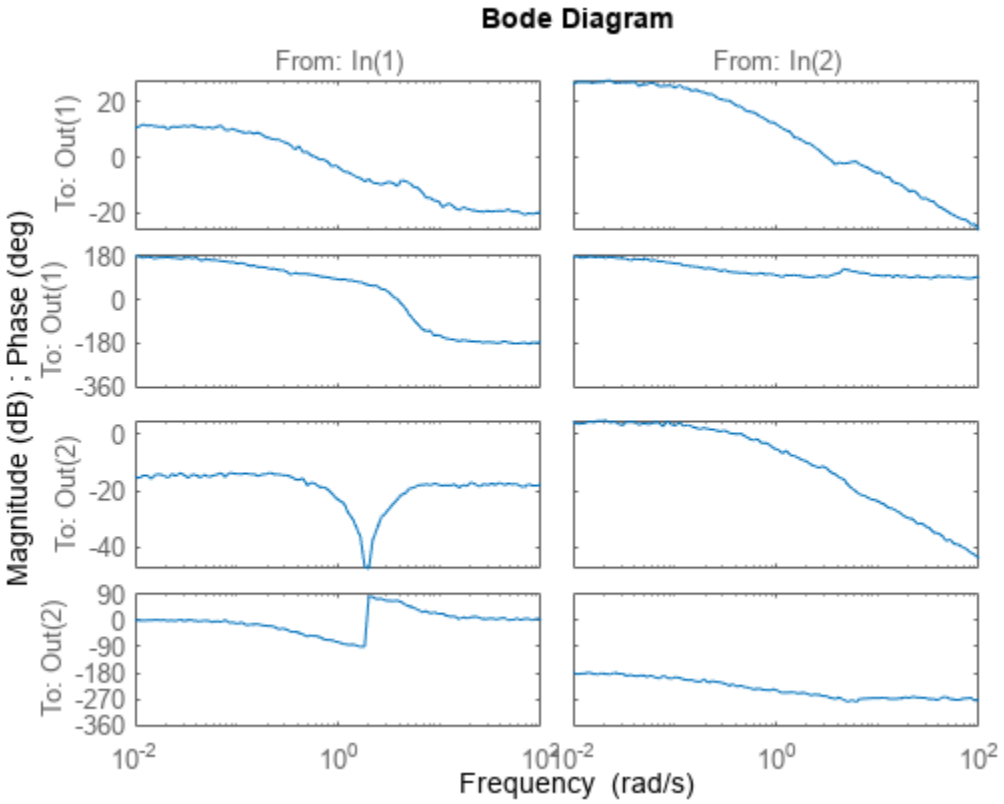
Create a frequency-response model.

```
H = frd(Hresp,freq);
```

`H` is an `frd` model object that represents frequency response data.

You can use `frd` models with many frequency-domain analysis commands. For example, visualize the response of this two-input, two-output system using `bode`.

```
bode(H)
```



By default, the `frd` command assumes that the frequencies are in radians/second. To specify different frequency units, use the `TimeUnit` and `FrequencyUnit` properties of the `frd` model object. For example, set the frequency units to radians/minute.

```
H = frd(Hresp,freq,"TimeUnit","min","FrequencyUnit","rad/TimeUnit");
```

See Also

`frd`

Related Examples

- “Frequency Response Data (FRD) Models” on page 2-8

More About

- “What Are Model Objects?” on page 1-2
- “Store and Retrieve Model Data” on page 3-2

Select Input/Output Pairs in MIMO Models

This example shows how to select the response from the first input to the second output of a MIMO model.

1. Create a two-input, one-output transfer function.

```
N = {[1 -1],[1];[1 2],[3 1 4]};
D = [1 1 10];
H = tf(N,D)
```

Note For more information about using cell arrays to create MIMO transfer functions, see the `tf` reference page.

2. Select the response from the second input to the output of `H`.

   To do this, use MATLAB array indexing.

```
H12 = H(1,2)
```

   For any MIMO system `H`, the index notation `H(i,j)` selects the response from the *j*th input to the *i*th output.

See Also

Related Examples

- “MIMO Transfer Functions” on page 2-22
- “MIMO State-Space Models” on page 2-24

More About

- “Store and Retrieve Model Data” on page 3-2

Time Delays in Linear Systems

Use the following model properties to represent time delays in linear systems.

- `InputDelay`, `OutputDelay` — Time delays at system inputs or outputs
- `ioDelay`, `InternalDelay` — Time delays that are internal to the system

In discrete-time models, these properties are constrained to integer values that represent delays expressed as integer multiples of the sample time. To approximate discrete-time models with delays that are a fractional multiple of the sample time, use `thiran`.

First Order Plus Dead Time Model

This example shows how to create a first order plus dead time model using the `InputDelay` or `OutputDelay` properties of `tf`.

To create the following first-order transfer function with a 2.1 s time delay:

$$G(s) = e^{-2.1s}\frac{1}{s+10},$$

enter:

```
G = tf(1,[1 10],'InputDelay',2.1)
```

where `InputDelay` specifies the delay at the input of the transfer function.

Tip You can use `InputDelay` with `zpk` the same way as with `tf`:

```
G = zpk([],-10,1,'InputDelay',2.1)
```

For SISO transfer functions, a delay at the input is equivalent to a delay at the output. Therefore, the following command creates the same transfer function:

```
G = tf(1,[1 10],'OutputDelay',2.1)
```

Use dot notation to examine or change the value of a time delay. For example, change the time delay to 3.2 as follows:

```
 G.OutputDelay = 3.2;
```

To see the current value, enter:

```
G.OutputDelay

ans =

    3.2000
```

Tip An alternative way to create a model with a time delay is to specify the transfer function with the delay as an expression in *s*:

1. Create a transfer function model for the variable s.

```
s = tf('s');
```

2. Specify $G(s)$ as an expression in s.

```
G = exp(-2.1*s)/(s+10);
```

Input and Output Delay in State-Space Model

This example shows how to create state-space models with delays at the inputs and outputs, using the `InputDelay` or `OutputDelay` properties of `ss`.

Create a state-space model describing the following one-input, two-output system:

$$\frac{dx(t)}{dt} = -2x(t) + 3u(t - 1.5)$$
$$y(t) = \begin{bmatrix} x(t - 0.7) \\ -x(t) \end{bmatrix}.$$

This system has an input delay of 1.5. The first output has an output delay of 0.7, and the second output is not delayed.

Note In contrast to SISO transfer functions, input delays are not equivalent to output delays for state-space models. Shifting a delay from input to output in a state-space model requires introducing a time shift in the model states. For example, in the model of this example, defining $T = t$ – 1.5 and $X(T) = x(T + 1.5)$ results in the following equivalent system:

$$\frac{dX(T)}{dT} = -2X(T) + 3u(T)$$
$$y(T) = \begin{bmatrix} X(T - 2.2) \\ -X(T - 1.5) \end{bmatrix}.$$

All of the time delays are on the outputs, but the new state variable X is time-shifted relative to the original state variable x. Therefore, if your states have physical meaning, or if you have known state initial conditions, consider carefully before shifting time delays between inputs and outputs.

To create this system:

1. Define the state-space matrices.

```
A = -2;
B = 3;
C = [1;-1];
D = 0;
```

2. Create the model.

```
G = ss(A,B,C,D,'InputDelay',1.5,'OutputDelay',[0.7;0])
```

`G` is a `ss` model.

Tip Use `delayss` to create state-space models with more general combinations of input, output, and state delays, of the form:

$$\frac{dx}{dt} = Ax(t) + Bu(t) + \sum_{j=1}^{N} (Ajx(t - tj) + Bju(t - tj))$$

$$y(t) = Cx(t) + Du(t) + \sum_{j=1}^{N} (Cjx(t - tj) + Dju(t - tj))$$

Transport Delay in MIMO Transfer Function

This example shows how to create a MIMO transfer function with different transport delays for each input-output (I/O) pair.

Create the MIMO transfer function:

$$H(s) = \begin{bmatrix} e^{-0.1}\frac{2}{s} & e^{-0.3}\frac{s+1}{s+10} \\ 10 & e^{-0.2}\frac{s-1}{s+5} \end{bmatrix}.$$

Time delays in MIMO systems can be specific to each I/O pair, as in this example. You cannot use `InputDelay` and `OutputDelay` to model I/O-specific transport delays. Instead, use `ioDelay` to specify the transport delay across each I/O pair.

To create this MIMO transfer function:

1. Create a transfer function model for the variable `s`.

   ```
   s = tf('s');
   ```

2. Use the variable `s` to specify the transfer functions of `H` without the time delays.

   ```
   H = [2/s (s+1)/(s+10); 10 (s-1)/(s+5)];
   ```

3. Specify the `ioDelay` property of `H` as an array of values corresponding to the transport delay for each I/O pair.

   ```
   H.IODelay = [0.1 0.3; 0 0.2];
   ```

`H` is a two-input, two-output `tf` model. Each I/O pair in `H` has the time delay specified by the corresponding entry in `tau`.

Discrete-Time Transfer Function with Time Delay

This example shows how to create a discrete-time transfer function with a time delay.

In discrete-time models, a delay of one sampling period corresponds to a factor of z^{-1} in the transfer function. For example, the following transfer function represents a discrete-time SISO system with a delay of 25 sampling periods.

$$H(z) = z^{-25}\frac{2}{z - 0.95}.$$

To represent integer delays in discrete-time systems in MATLAB, set the `'InputDelay'` property of the model object to an integer value. For example, the following command creates a `tf` model representing $H(z)$ with a sampling time of 0.1 s.

```
H = tf(2,[1 -0.95],0.1,'InputDelay',25)
```

```
H =

                2
  z^(-25) * --------
            z - 0.95

Sample time: 0.1 seconds
Discrete-time transfer function.
```

If system has a time delay that is not an integer multiple of the sampling time, you can use the `thiran` command to approximate the fractional portion of the time delay with an all-pass filter. See “Time-Delay Approximation” on page 2-37.

See Also

Related Examples

- “Closing Feedback Loops with Time Delays” on page 2-35
- “Convert Time Delay in Discrete-Time Model to Factors of 1/z” on page 2-50

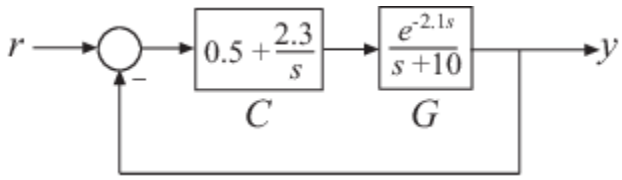
More About

- “Time-Delay Approximation” on page 2-37

Closing Feedback Loops with Time Delays

This example shows how internal delays arise when you interconnect models that have input, output, or transport time delays.

Create a model of the following control architecture:



`G` is the plant model, which has an input delay. `C` is a proportional-integral (PI) controller.

To create a model representing the closed-loop response of this system:

1. Create the plant `G` and the controller `C`.

   ```
   G = tf(1,[1 10],'InputDelay',2.1);
   C = pid(0.5,2.3);
   ```

   `C` has a proportional gain of 0.5 and an integral gain of 2.3.

2. Use `feedback` to compute the closed-loop response from *r* to *y*.

   ```
   T = feedback(C*G,1);
   ```

The time delay in `T` is not an input delay as it is in `G`. Because the time delay is internal to the closed-loop system, the software returns `T` as an `ss` model with an internal time delay of 2.1 seconds.

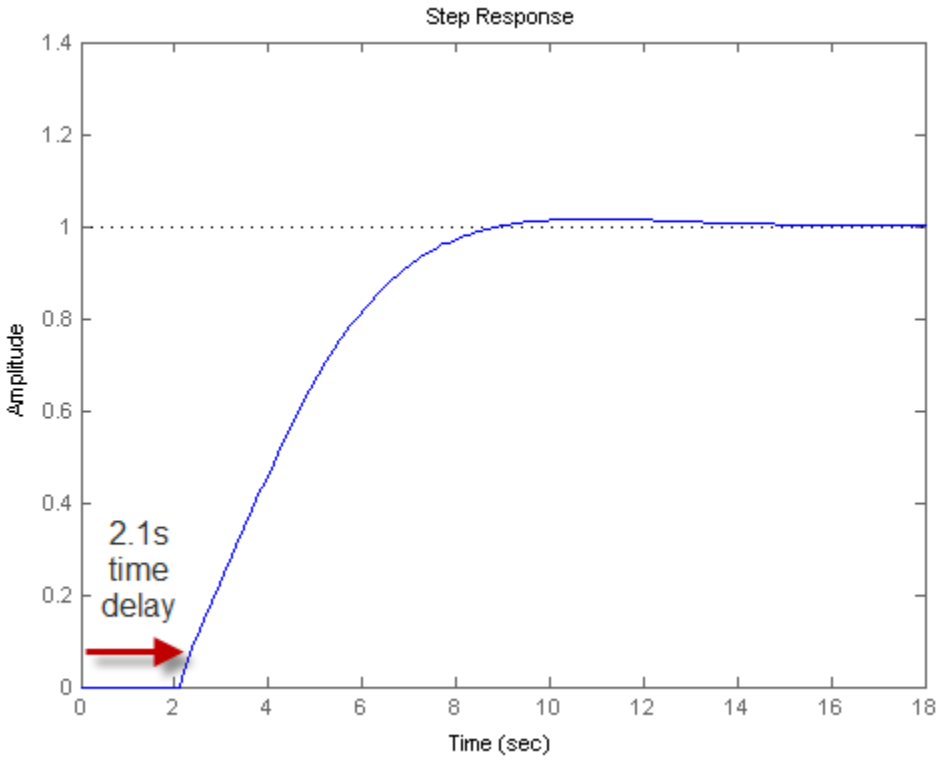
Note In addition to `feedback`, any system interconnection function (including `parallel` and `series`) can give rise to internal delays.

T is an exact representation of the closed-loop response, not an approximation. To access the internal delay value, enter:

```
T.InternalDelay
```

A step plot of *T* confirms the presence of the time delay:

```
step(T)
```



Note Most analysis commands, such as `step`, `bode` and `margin`, support models with internal delays.

The internal time delay is stored in the `InternalDelay` property of `T`. Use dot notation to access `InternalDelay`. For example, to change the internal delay to 3.5 seconds, enter:

```
T.InternalDelay = 3.5
```

You cannot modify the number of internal delays because they are structural properties of the model.

See Also

Related Examples

- “Convert Time Delay in Discrete-Time Model to Factors of 1/z” on page 2-50

More About

- “Internal Delays” on page 2-55

Time-Delay Approximation

Many control design algorithms cannot handle time delays directly. For example, techniques such as root locus, LQG, and pole placement do not work properly if time delays are present. A common technique is to replace delays with all-pass filters that approximate the delays.

To approximate time delays in continuous-time LTI models, use the `pade` command to compute a Padé approximation. The Padé approximation is valid only at low frequencies, and provides better frequency-domain approximation than time-domain approximation. It is therefore important to compare the true and approximate responses to choose the right approximation order and check the approximation validity.

Time-Delay Approximation in Discrete-Time Models

For discrete-time models, use `absorbDelay` to convert a time delay to factors of $1/z$ where the time delay is an integer multiple of the sample time.

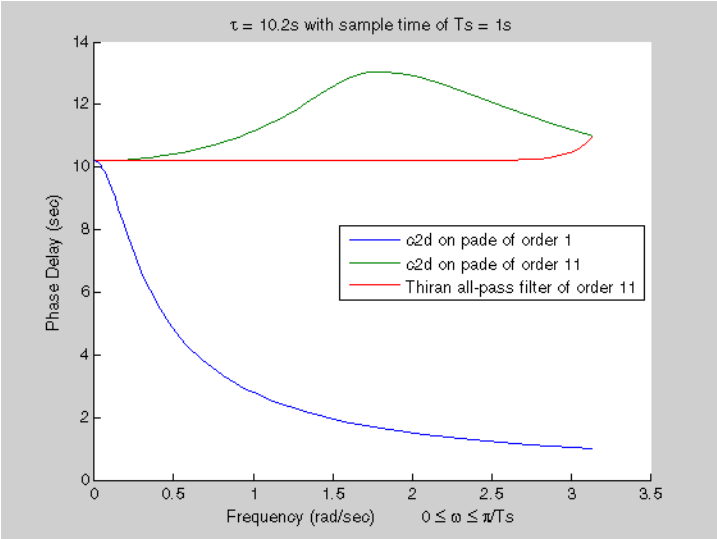
Use the `thiran` command to approximate a time delay that is a fractional multiple of the sample time as a Thiran all-pass filter.

For a time delay of `tau` and a sample time of `Ts`, the syntax `thiran(tau,Ts)` creates a discrete-time transfer function that is the product of two terms:

- A term representing the integer portion of the time delay as a pure line delay, $(1/z)^N$, where `N = ceil(tau/Ts)`.
- A term approximating the fractional portion of the time delay (`tau - NTs`) as a Thiran all-pass filter.

Discretizing a Padé approximation does not guarantee good phase matching between the continuous-time delay and its discrete approximation. Using `thiran` to generate a discrete-time approximation of a continuous-time delay can yield much better phase matching. For example, the following figure shows the phase delay of a 10.2-second time delay discretized with a sample time of 1 s, approximated in three ways:

- a first-order Padé approximation, discretized using the `tustin` method of `c2d`
- an 11th-order Padé approximation, discretized using the `tustin` method of `c2d`
- an 11th-order Thiran filter



The Thiran filter yields the closest approximation of the 10.2-second delay.

See the `thiran` reference page for more information about Thiran filters.

See Also

`pade` | `absorbDelay` | `thiran`

Related Examples

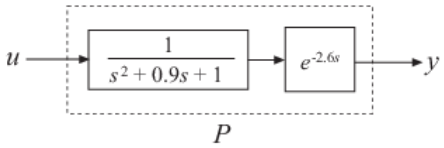
- “Time-Delay Approximation in Continuous-Time Open-Loop Model” on page 2-39
- “Convert Time Delay in Discrete-Time Model to Factors of 1/z” on page 2-50
- “Approximate Different Delays with Different Approximation Orders” on page 2-47

Time-Delay Approximation in Continuous-Time Open-Loop Model

This example shows how to approximate delays in a continuous-time open-loop system using `pade`.

Padé approximation is helpful when using analysis or design tools that do not support time delays.

1 Create sample open-loop system with an output delay.




```
s = tf('s');
P = exp(-2.6*s)/(s^2+0.9*s+1);
```

`P` is a second-order transfer function (`tf`) object with a time delay.

2 Compute the first-order Padé approximation of `P`.

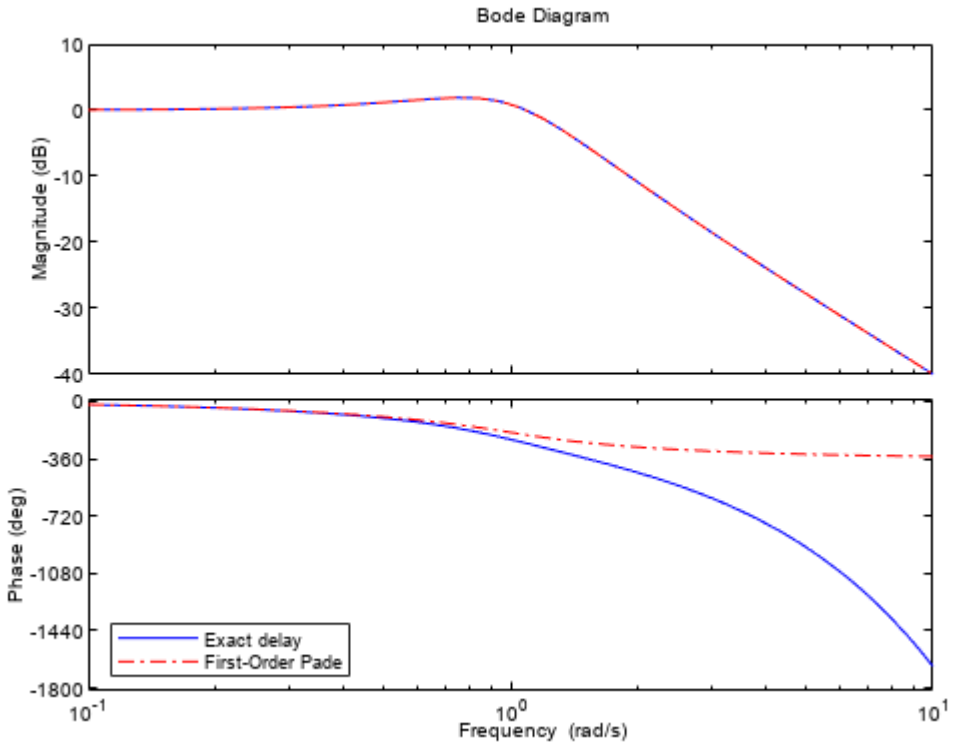
```
Pnd1 = pade(P,1)
```

```
Pnd1 =

             -s + 0.7692
  ---------------------------------
  s^3 + 1.669 s^2 + 1.692 s + 0.7692

Continuous-time transfer function.
```

This command replaces all time delays in `P` with a first-order approximation. Therefore, `Pnd1` is a third-order transfer function with no delays.

3 Compare the frequency response of the original and approximate models using `bodeplot`.

```
h = bodeoptions;
h.PhaseMatching = 'on';
bodeplot(P,'-b',Pnd1,'-.r',{0.1,10},h)
legend('Exact delay','First-Order Pade','Location','SouthWest')
```



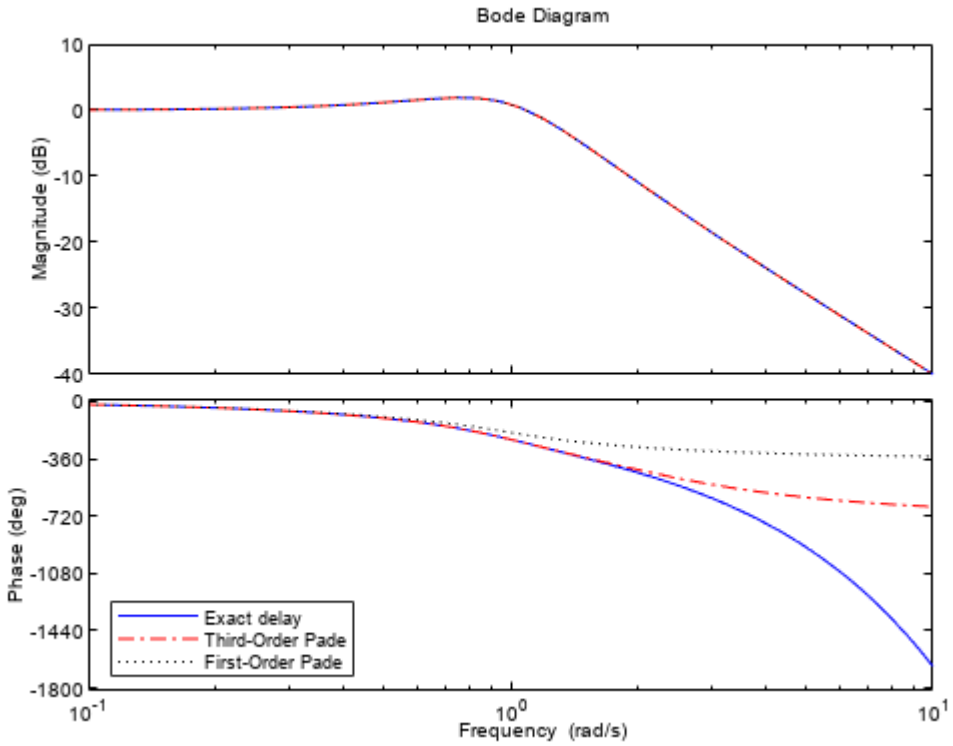
The magnitude of `P` and `Pnd1` match exactly. However, the phase of `Pnd1` deviates from the phase of `P` beyond approximately 1 rad/s.

4 Increase the Padé approximation order to extend the frequency band in which the phase approximation is good.

```
Pnd3 = pade(P,3);
```

5 Compare the frequency response of `P`, `Pnd1` and `Pnd3`.

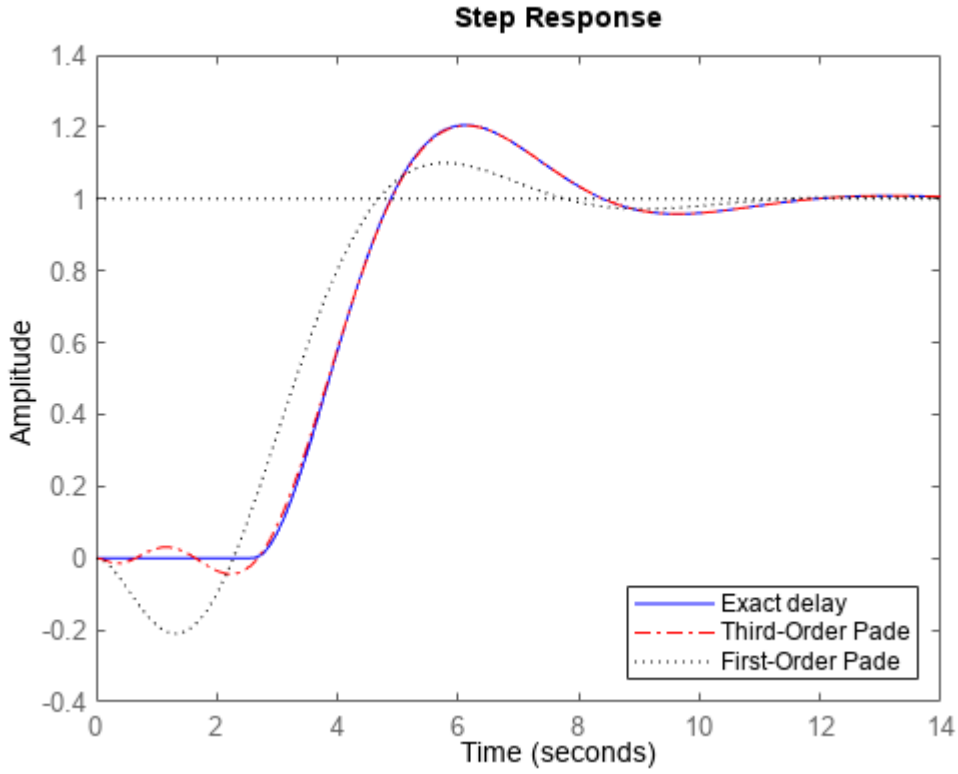
```
bodeplot(P,'-b',Pnd3,'-.r',Pnd1,':k',{0.1 10},h)
legend('Exact delay','Third-Order Pade','First-Order Pade',...
       'Location','SouthWest')
```



The phase approximation error is reduced by using a third-order Padé approximation.

6 Compare the time domain responses of the original and approximated systems using `stepplot`.

```
stepplot(P,'-b',Pnd3,'-.r',Pnd1,':k')
legend('Exact delay','Third-Order Pade','First-Order Pade',...
       'Location','Southeast')
```



Using the Padé approximation introduces a nonminimum phase artifact ("wrong way" effect) in the initial transient response. The effect is quite pronounced in the first-order approximation, which dips significantly below zero before changing direction. The effect is reduced in the higher-order approximation, which far more closely matches the exact system's response.

Note Using too high an approximation order may result in numerical issues and possibly unstable poles. Therefore, avoid Padé approximations with order N>10.

See Also

pade

Related Examples

- "Time-Delay Approximation in Continuous-Time Closed-Loop Model" on page 2-43

More About

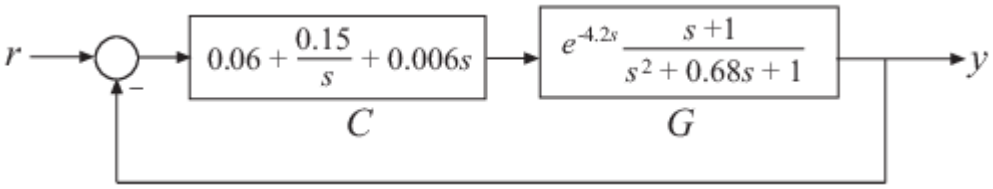
- "Time-Delay Approximation" on page 2-37
- "Internal Delays" on page 2-55

Time-Delay Approximation in Continuous-Time Closed-Loop Model

This example shows how to approximate delays in a continuous-time closed-loop system with internal delays, using `pade`.

Padé approximation is helpful when using analysis or design tools that do not support time delays.

1 Create sample continuous-time closed-loop system with an internal delay.



Construct a model `Tcl` of the closed-loop transfer function from `r` to `y`.

```
s = tf('s');
G = (s+1)/(s^2+.68*s+1)*exp(-4.2*s);
C = pid(0.06,0.15,0.006);
Tcl = feedback(G*C,1);
```

Examine the internal delay of `Tcl`.

```
Tcl.InternalDelay
```

```
ans = 4.2000
```

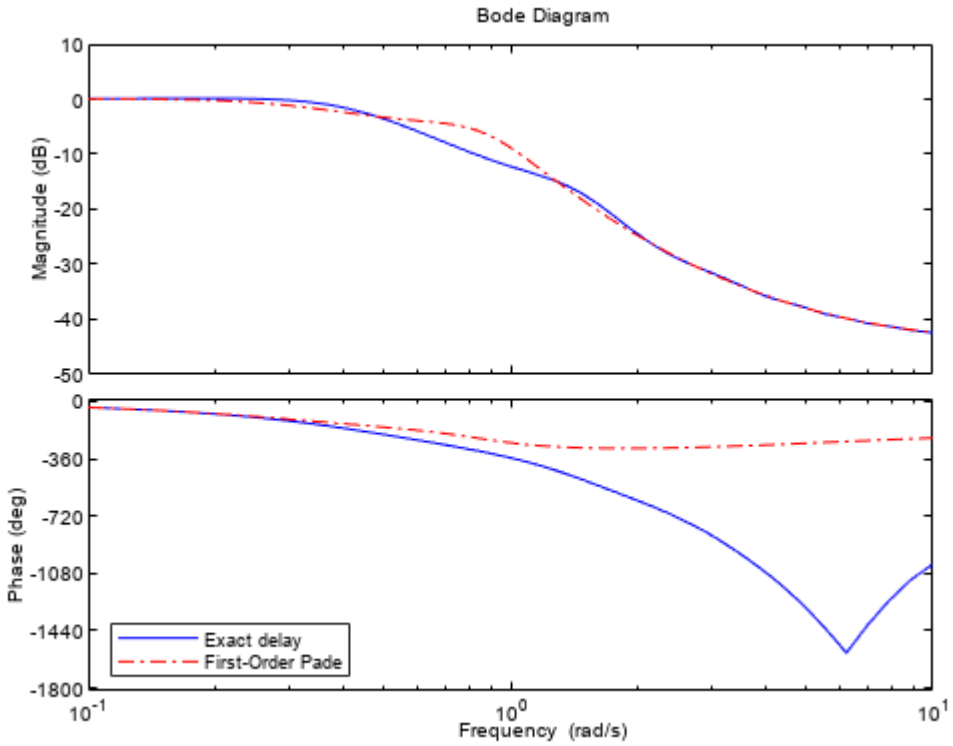
2 Compute the first-order Padé approximation of `Tcl`.

```
Tnd1 = pade(Tcl,1);
```

`Tnd1` is a state-space (`ss`) model with no delays.

3 Compare the frequency response of the original and approximate models.

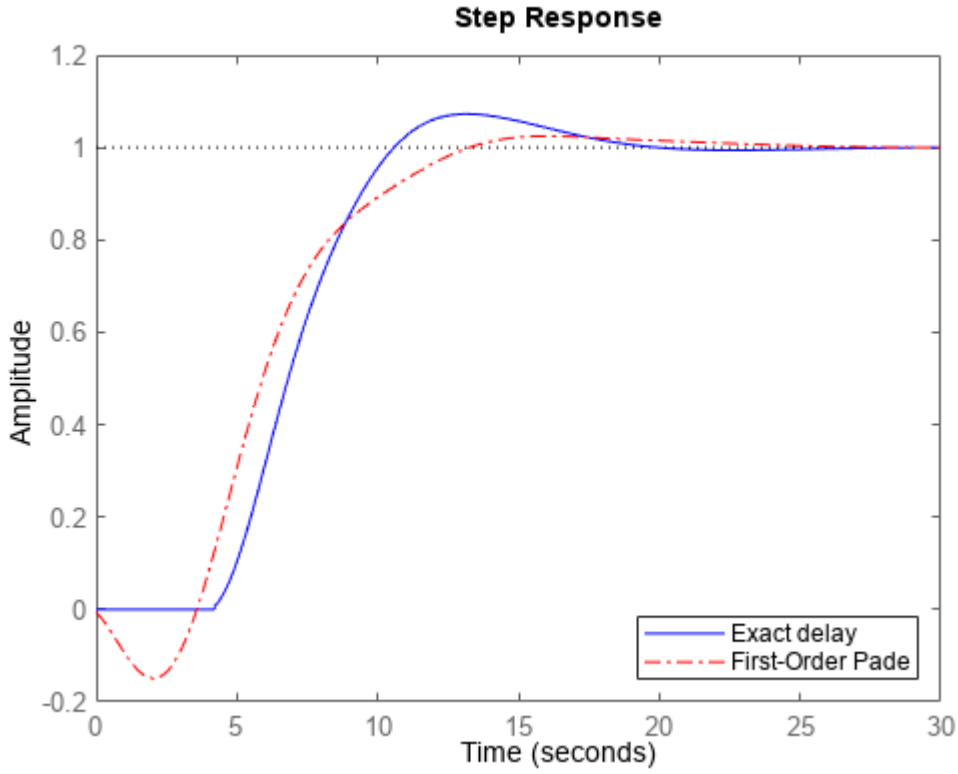
```
h = bodeoptions;
h.PhaseMatching = 'on';
bodeplot(Tcl,'-b',Tnd1,'-.r',{.1,10},h);
legend('Exact delay','First-Order Pade','Location','SouthWest');
```



The magnitude and phase approximation errors are significant beyond 1 rad/s.

4 Compare the time domain response of `Tcl` and `Tnd1` using `stepplot`.

```
stepplot(Tcl,'-b',Tnd1,'-.r');
legend('Exact delay','First-Order Pade','Location','SouthEast');
```



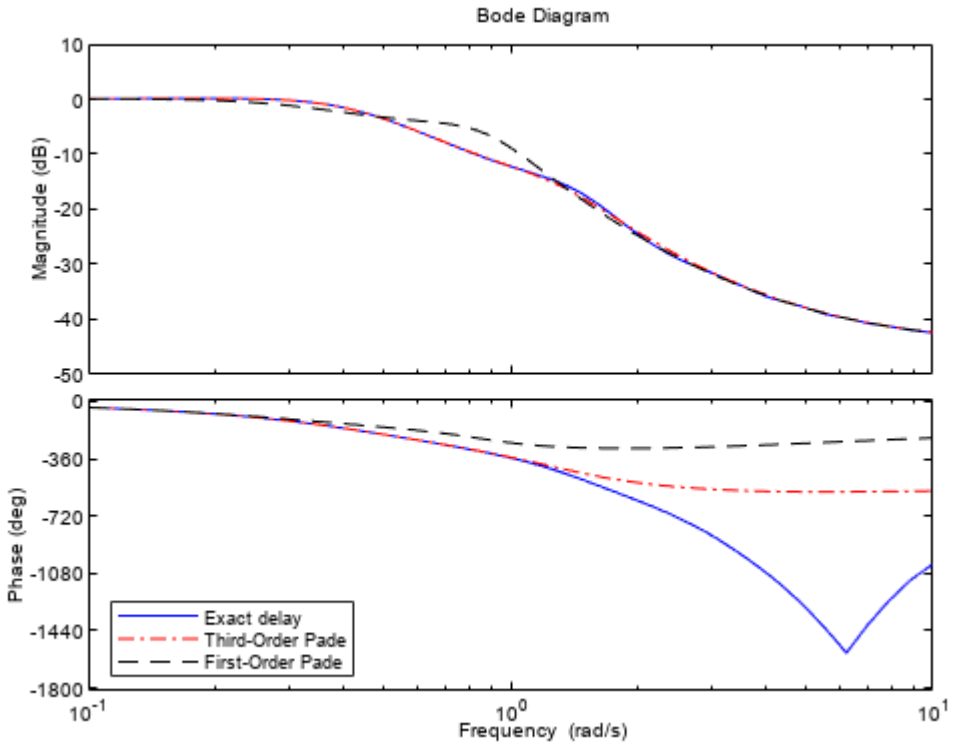
Using the Padé approximation introduces a nonminimum phase artifact (“wrong way” effect) in the initial transient response.

5 Increase the Padé approximation order to see if this will extend the frequency with good phase and magnitude approximation.

```
Tnd3 = pade(Tcl,3);
```

6 Observe the behavior of the third-order Padé approximation of `Tcl`. Compare the frequency response of `Tcl` and `Tnd3`.

```
bodeplot(Tcl,'-b',Tnd3,'-.r',Tnd1,'--k',{.1,10},h);
legend('Exact delay','Third-Order Pade','First-Order Pade',...
       'Location','SouthWest');
```



The magnitude and phase approximation errors are reduced when a third-order Padé approximation is used.

Increasing the Padé approximation order extends the frequency band where the approximation is good. However, too high an approximation order may result in numerical issues and possibly unstable poles. Therefore, avoid Padé approximations with order N>10.

See Also

pade

Related Examples

- “Approximate Different Delays with Different Approximation Orders” on page 2-47

More About

- “Time-Delay Approximation” on page 2-37
- “Internal Delays” on page 2-55

Approximate Different Delays with Different Approximation Orders

This example shows how to specify different Padé approximation orders to approximate internal and output delays in a continuous-time open-loop system.

Load a sample continuous-time open-loop system that contains internal and output time delays.

```
load('PadeApproximation1.mat','sys')
sys
```

```
sys =

  A =
         x1    x2
   x1  -1.5  -0.1
   x2     1     0

  B =
       u1
   x1   1
   x2   0

  C =
        x1   x2
   y1  0.5  0.1

  D =
       u1
   y1   0

  (values computed with all internal delays set to zero)

  Output delays (seconds): 1.5
  Internal delays (seconds): 3.4

Continuous-time state-space model.
```

`sys` is a second-order continuous-time `ss` model with internal delay 3.4 s and output delay 1.5 s.

Use the `pade` function to compute a third-order approximation of the internal delay and a first-order approximation of the output delay.

```
P13 = pade(sys,inf,1,3);
size(P13)
```

```
State-space model with 1 outputs, 1 inputs, and 6 states.
```

The three input arguments following `sys` specify the approximation orders of any input, output, and internal delays of `sys`, respectively. `inf` specifies that a delay is not to be approximated. The approximation orders for the output and internal delays are one and three respectively.

Approximating the time delays with `pade` absorbs delays into the dynamics, adding as many states to the model as orders in the approximation. Thus, `P13` is a sixth-order model with no delays.

For comparison, approximate only the internal delay of `sys`, leaving the output delay intact.

```
P3 = pade(sys,inf,inf,3);
size(P3)
```

```
State-space model with 1 outputs, 1 inputs, and 5 states.
```

```
P3.OutputDelay
```

```
ans = 1.5000
```

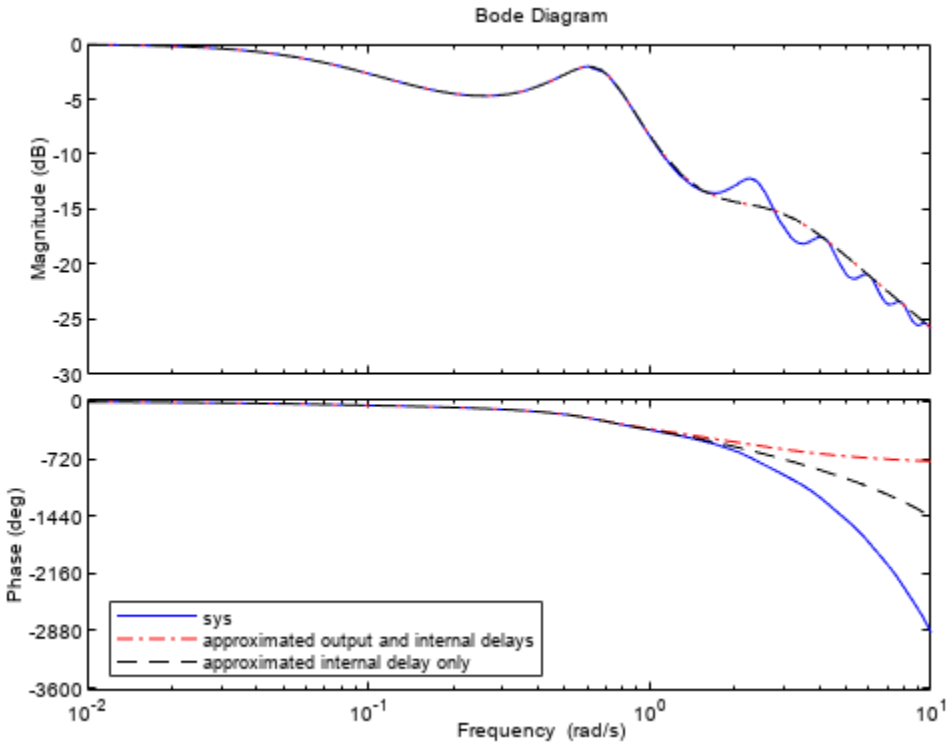
```
P3.InternalDelay
```

```
ans =

  0x1 empty double column vector
```

P3 retains the output delay, but the internal delay is approximated and absorbed into the state-space matrices, resulting in a fifth-order model without internal delays.

Compare the frequency response of the exact and approximated systems `sys`, `P13`, `P3`.

```
h = bodeoptions;
h.PhaseMatching = 'on';
bode(sys,'b-',P13,'r-.',P3,'k--',h,{.01,10});
legend('sys','approximated output and internal delays','approximated internal delay only',...
    'location','SouthWest')
```



Notice that approximating the internal delay loses the gain ripple displayed in the exact system.

See Also

pade

Related Examples

- “Time-Delay Approximation in Continuous-Time Open-Loop Model” on page 2-39

More About

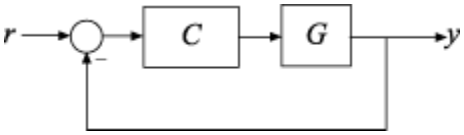
- “Time-Delay Approximation” on page 2-37
- “Internal Delays” on page 2-55

Convert Time Delay in Discrete-Time Model to Factors of 1/z

This example shows how to convert a time delay in a discrete-time model to factors of 1/_z_.

In a discrete-time model, a time delay of one sampling interval is equivalent to a factor of 1/_z_ (a pole at $z = 0$) in the model. Therefore, time delays stored in the `InputDelay`, `OutputDelay`, or `IODelay` properties of a discrete-time model can be rewritten in the model dynamics by rewriting them as poles at $z = 0$. However, the additional poles increase the order of the system. Particularly for large time delays, this can yield systems of very high order, leading to long computation times or numerical inaccuracies.

To illustrate how to eliminate time delays in a discrete-time closed-loop model, and to observe the effects of doing so, create the following closed-loop system:



G is a first-order discrete-time system with an input delay, and *C* is a PI controller.

```
G = ss(0.9,0.125,0.08,0,'Ts',0.01,'InputDelay',7);
C = pid(6,90,0,0,'Ts',0.01);
T = feedback(C*G,1);
```

Closing the feedback loop on a plant with input delays gives rise to internal delays in the closed-loop system. Examine the order and internal delay of `T`.

```
order(T)
```

```
ans = 2
```

```
T.InternalDelay
```

```
ans = 7
```

`T` is a second-order state-space model. One state is contributed by the first-order plant, and the other by the one pole of the PI controller. The delays do not increase the order of `T`. Instead, they are represented as an internal delay of seven time steps.

Replace the internal delay by z^{-7}.

```
Tnd = absorbDelay(T);
```

This command converts the internal delay to seven poles at $z = 0$. To confirm this, examine the order and internal delay of `Tnd`.

```
order(Tnd)
```

```
ans = 9
```

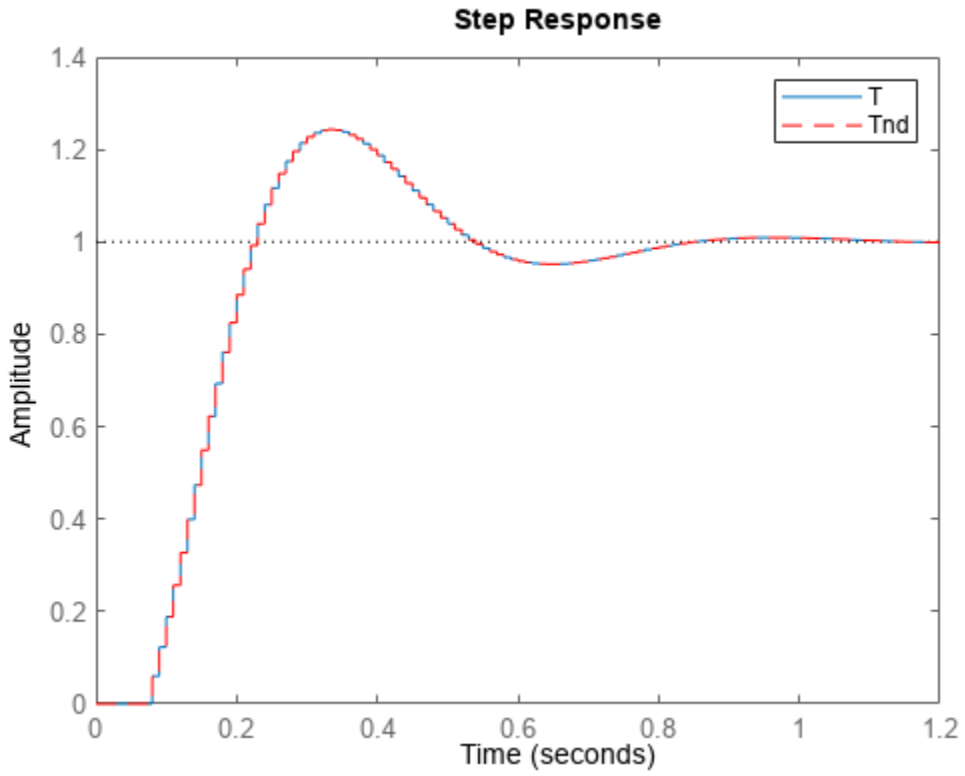
```
Tnd.InternalDelay
```

```
ans =

  0x1 empty double column vector
```

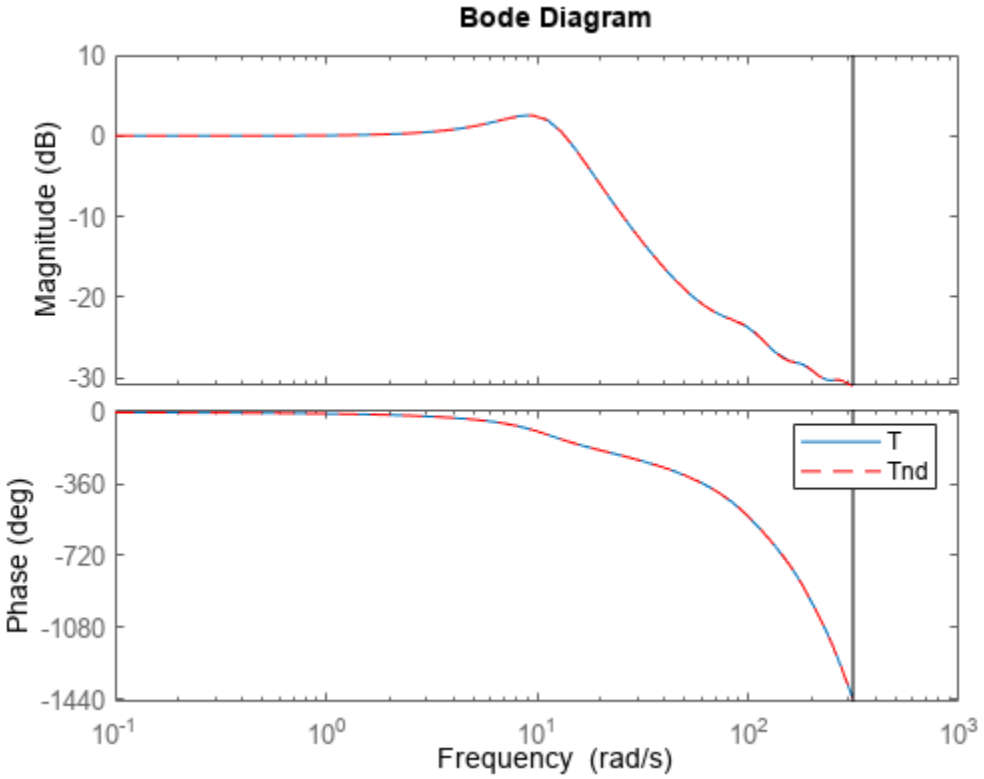
Tnd has no internal delay, but it is a ninth-order model, due to the seven extra poles introduced by absorbing the seven-unit delay into the model dynamics.

Despite this difference in representation, the responses of Tnd exactly match those of T.

```
stepplot(T,Tnd,'r--')
legend('T','Tnd')
```




```
bodeplot(T,Tnd,'r--')
legend('T','Tnd')
```



See Also

pade

Related Examples

- “Time-Delay Approximation in Continuous-Time Open-Loop Model” on page 2-39

More About

- “Time-Delay Approximation” on page 2-37
- “Internal Delays” on page 2-55

Frequency Response Data (FRD) Model with Time Delay

This example shows that absorbing time delays into frequency response data can cause undesirable phase wrapping at high frequencies.

When you collect frequency response data for a system that includes time delays, you can absorb the time delay into the frequency response as a phase shift. Alternatively, if you are able to separate time delays from your measured frequency response, you can represent the delays using the `InputDelay`, `OutputDelay`, or `ioDelay` properties of the `frd` model object. The latter approach can give better numerical results, as this example illustrates.

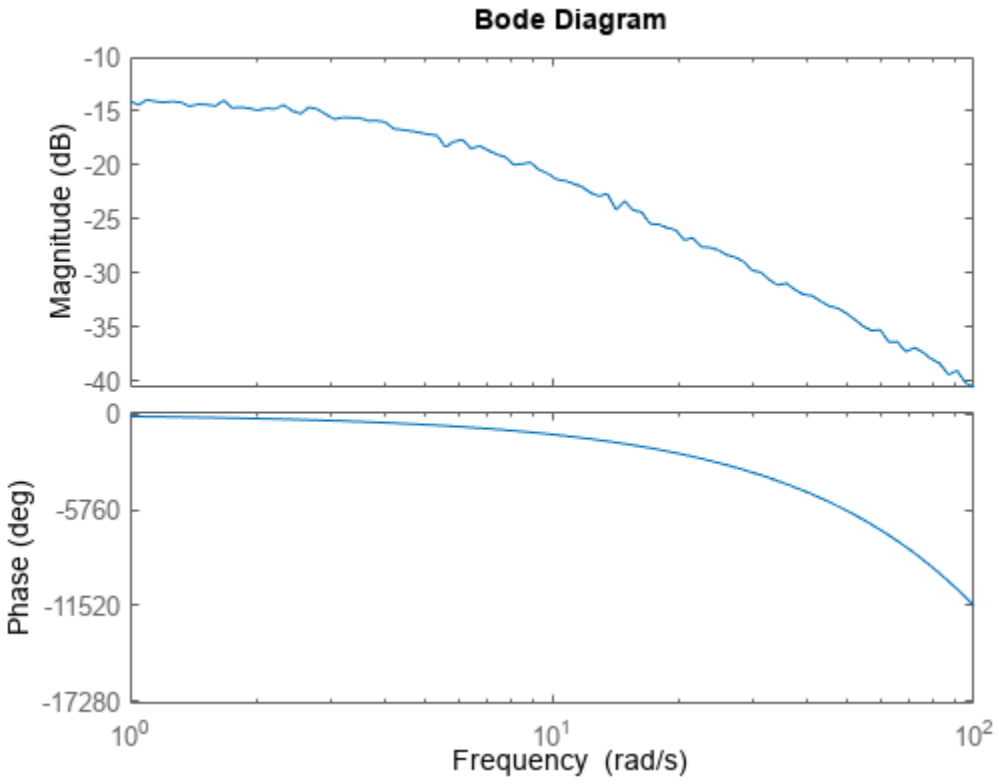
The `frd` model `fsys` includes a transport delay of 2 s. Load the model into the MATLAB® workspace and inspect the time delay.

```
load('frddelayexample.mat','fsys')
fsys.IODelay
```

```
ans = 2
```

A Bode plot of `fsys` shows the effect of the transport delay, causing the accumulation of phase as frequency increases.

```
bodeplot(fsys)
```



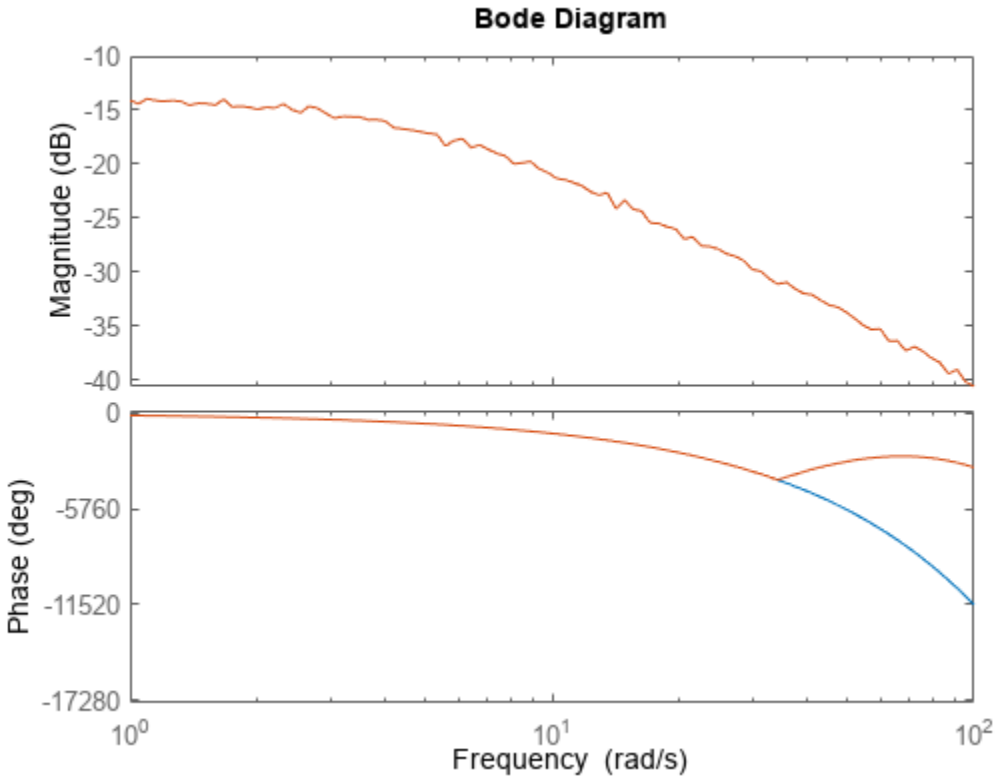
The `absorbDelay` command absorbs all time delays directly into the frequency response, resulting in an `frd` model with `IODelay = 0`.

```
fsys2 = absorbDelay(fsys);
fsys2.IODelay
```

```
ans = 0
```

Comparing the two ways of representing the delay shows that absorbing the delay into the frequency response causes phase-wrapping.

```
bode(fsys,fsys2)
```



Phase wrapping can introduce numerical inaccuracy at high frequencies or where the frequency grid is sparse. For that reason, if your system takes the form $e^{-\tau s}G(s)$, you might get better results by measuring frequency response data for $G(s)$ and using `InputDelay`, `OutputDelay`, or `ioDelay` to model the time delay τ.

See Also

`absorbDelay`

More About

- “Time-Delay Approximation” on page 2-37

Internal Delays

Using the `InputDelay`, `OutputDelay`, and `ioDelay` properties, you can model simple processes with transport delays. However, these properties cannot model more complex situations, such as feedback loops with delays. In addition to the `InputDelay` and `OutputDelay` properties, state-space (`ss`) models have an `InternalDelay` property. This property lets you model the interconnection of systems with input, output, or transport delays, including feedback loops with delays. You can use `InternalDelay` property to accurately model and analyze arbitrary linear systems with delays. Internal delays can arise from the following:

- Concatenating state-space models with input and output delays.
- Feeding back a delayed signal.
- Converting MIMO `tf` or `zpk` models with transport delays to state-space form.

Using internal time delays, you can do the following:

- In continuous time, generate approximate-free time and frequency simulations, because delays do not have to be replaced by a Padé approximation. In continuous time, this allows for more accurate analysis of systems with long delays.
- In discrete time, keep delays separate from other system dynamics, because delays are not replaced with poles at $z = 0$, which boosts efficiency of time and frequency simulations for discrete-time systems with long delays.
- Use most Control System Toolbox functions.
- Test advanced control strategies for delayed systems. For example, you can implement and test an accurate model of a Smith predictor. See the example “Control of Processes with Long Dead Time: The Smith Predictor” on page 11-115.

Why Internal Delays Are Necessary

This example illustrates why input, output, and transport delays not enough to model all types of delays that can arise in dynamic systems. Consider the simple feedback loop with a 2 s. delay:

The closed-loop transfer function is

$$\frac{e^{-2s}}{s+2+e^{-2s}}$$

The delay term in the numerator can be represented as an output delay. However, the delay term in the denominator cannot. In order to model the effect of the delay on the feedback loop, the `InternalDelay` property is needed to keep track of internal coupling between delays and ordinary dynamics.

Typically, you do not create state-space models with internal delays directly, by specifying the A, B, C, and D matrices together with a set of internal delays. Rather, such models arise when you interconnect models having delays. There is no limitation on how many delays are involved and how the models are connected. For an example of creating an internal delay by closing a feedback loop, see “Closing Feedback Loops with Time Delays” on page 2-35.

Behavior of Models With Internal Delays

When you work with models having internal delays, be aware of the following behavior:

- When a model interconnection gives rise to internal delays, the software returns an `ss` model regardless of the interconnected model types. This occurs because only `ss` supports internal delays.
- The software fully supports feedback loops. You can wrap a feedback loop around any system with delays.
- When displaying the `A`, `B`, `C`, and `D` matrices, the software sets all delays to zero (creating a zero-order Padé approximation). This approximation occurs for the display only, and not for calculations using the model.

  For some systems, setting delays to zero creates singular algebraic loops, which result in either improper or ill-defined, zero-delay approximations. For these systems:

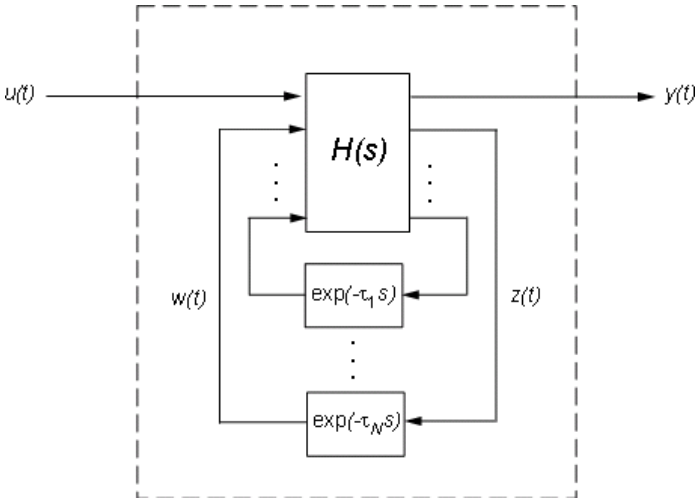
  - Entering `sys` returns only sizes for the matrices of a system named `sys`.
  - Entering `sys.A` produces an error.

    The limited display and the error do not imply a problem with the model `sys` itself.

Inside Time Delay Models

State-space objects use generalized state-space equations to keep track of internal delays. Conceptually, such models consist of two interconnected parts:

- An ordinary state-space model *H(s)* with an augmented I/O set
- A bank of internal delays.



The corresponding state-space equations are:

$$\dot{x} = Ax(t) + B_1u(t) + B_2w(t)$$
$$y(t) = C_1x(t) + D_{11}u(t) + D_{12}w(t)$$
$$z(t) = C_2x(t) + D_{21}u(t) + D_{22}w(t)$$
$$w_j(t) = z(t - \tau_j), \quad j = 1, \ldots, N$$

You need not bother with this internal representation to use the tools. If, however, you want to extract H or the matrices A, B1, B2, ... , you can use getDelayModel, For the example:

```
P = 5*exp(-3.4*s)/(s+1);
C = 0.1 * (1 + 1/(5*s));
T = feedback(ss(P*C),1);
[H,tau] = getDelayModel(T,'lft');
size(H)
```

Note that H is a two-input, two-output model whereas T is SISO. The inverse operation (combining H and tau to construct T) is performed by setDelayModel.

See [1], [2] for details.

Functions That Support Internal Time Delays

The following commands support internal delays for both continuous- and discrete-time systems:

- All interconnection functions
- Time domain response functions—except for impulse and initial
- Frequency domain functions—except for norm

Limitations on Functions that Support Internal Time Delays

The following commands support internal delays for both continuous- and discrete-time systems and have certain limitations:

- allmargin, margin—Uses interpolation, therefore these commands are only as precise as the fineness of the specified grid.
- pole, zero—Returns poles and zeros of the system with all delays set to zero.
- ssdata, get—If an SS model has internal delays, these commands return the A, B, C, and D matrices of the system with all internal delays set to zero. Use getDelayModel to access the internal state-space representation of models with internal delays.

Functions That Do Not Support Internal Time Delays

The following commands do not support internal time delays:

- System dynamics—norm and isstable
- Time-domain analysis—initial and initialplot
- Model simplification—balreal, balred, and modred
- Compensator design—rlocus, lqg, lqry, lqrd, kalman, kalmd, lqgreg, lqgtrack, lqi, and augstate.

  To use these functions on a system with internal delays, use pade to approximate the internal delays. See “Time-Delay Approximation” on page 2-37.

References

[1] P. Gahinet and L.F. Shampine, "Software for Modeling and Analysis of Linear Systems with Delays," *Proc. American Control Conf.*, Boston, 2004, pp. 5600-5605

[2] L.F. Shampine and P. Gahinet, Delay-differential-algebraic Equations in Control Theory, *Applied Numerical Mathematics*, 56 (2006), pp. 574-588

See Also

Related Examples

- “Closing Feedback Loops with Time Delays” on page 2-35

Tunable Low-Pass Filter

In this example, you will create a low-pass filter with one tunable parameter a:

$$F = \frac{a}{s + a}$$

Since the numerator and denominator coefficients of a `tunableTF` block are independent, you cannot use `tunableTF` to represent `F`. Instead, construct `F` using the tunable real parameter object `realp`.

Create a real tunable parameter with an initial value of `10`.

```
a = realp('a',10)
```

```
a =
       Name: 'a'
      Value: 10
    Minimum: -Inf
    Maximum: Inf
       Free: 1

Real scalar parameter.
```

Use `tf` to create the tunable low-pass filter `F`.

```
numerator = a;
denominator = [1,a];
F = tf(numerator,denominator)
```

```
Generalized continuous-time state-space model with 1 outputs, 1 inputs, 1 states, and the follow
  a: Scalar parameter, 2 occurrences.

Type "ss(F)" to see the current value and "F.Blocks" to interact with the blocks.
```

`F` is a `genss` object which has the tunable parameter `a` in its `Blocks` property. You can connect `F` with other tunable or numeric models to create more complex control system models. For an example, see “Control System with Tunable Components” on page 2-63.

See Also

`tunableTF` | `realp` | `genss`

More About

- “Models with Tunable Coefficients” on page 1-15
- “Create Tunable Second-Order Filter” on page 2-60
- “Create State-Space Model with Both Fixed and Tunable Parameters” on page 2-62
- “Control System with Tunable Components” on page 2-63

Create Tunable Second-Order Filter

This example shows how to create a parametric model of the second-order filter:

$$F(s) = \frac{\omega_n^2}{s^2 + 2\zeta\omega_n s + \omega_n^2},$$

where the damping ζ and the natural frequency ω_n are tunable parameters.

Define the tunable parameters using `realp`.

```
wn = realp('wn',3);
zeta = realp('zeta',0.8);
```

`wn` and `zeta` are `realp` parameter objects, with initial values `3` and `0.8`, respectively.

Create a model of the filter using the tunable parameters.

```
F = tf(wn^2,[1 2*zeta*wn wn^2]);
```

The inputs to `tf` are the vectors of numerator and denominator coefficients expressed in terms of `wn` and `zeta`.

`F` is a `genss` model. The property `F.Blocks` lists the two tunable parameters `wn` and `zeta`.

```
F.Blocks
```

```
ans = struct with fields:
      wn: [1x1 realp]
    zeta: [1x1 realp]
```

You can examine the number of tunable blocks in a generalized model using `nblocks`.

```
nblocks(F)
```

```
ans = 6
```

`F` has two tunable parameters, but the parameter `wn` appears five times - twice in the numerator and three times in the denominator.

To reduce the number of tunable blocks, you can rewrite `F` as:

$$F(s) = \frac{1}{\left(\frac{s}{\omega_n}\right)^2 + 2\zeta\left(\frac{s}{\omega_n}\right) + 1}.$$

Create the alternative filter.

```
F = tf(1,[(1/wn)^2 2*zeta*(1/wn) 1]);
```

Examine the number of tunable blocks in the new model.

```
nblocks(F)
```

```
ans = 4
```

In the new formulation, there are only three occurrences of the tunable parameter `wn`. Reducing the number of occurrences of a block in a model can improve the performance of calculations involving the model. However, the number of occurrences does not affect the results of tuning the model or sampling it for parameter studies.

See Also

`realp` | `genss` | `nblocks`

More About

- “Models with Tunable Coefficients” on page 1-15
- “Tunable Low-Pass Filter” on page 2-59
- “Create State-Space Model with Both Fixed and Tunable Parameters” on page 2-62
- “Control System with Tunable Components” on page 2-63

Create State-Space Model with Both Fixed and Tunable Parameters

This example shows how to create a state-space `genss` model having both fixed and tunable parameters.

$$A = \begin{bmatrix} 1 & a+b \\ 0 & ab \end{bmatrix}, \quad B = \begin{bmatrix} -3.0 \\ 1.5 \end{bmatrix}, \quad C = [0.3 \ 0], \quad D = 0,$$

where a and b are tunable parameters, whose initial values are `-1` and `3`, respectively.

Create the tunable parameters using `realp`.

```
a = realp('a',-1);
b = realp('b',3);
```

Define a generalized matrix using algebraic expressions of `a` and `b`.

```
A = [1 a+b;0 a*b];
```

`A` is a generalized matrix whose `Blocks` property contains `a` and `b`. The initial value of `A` is `[1 2;0 -3]`, from the initial values of `a` and `b`.

Create the fixed-value state-space matrices.

```
B = [-3.0;1.5];
C = [0.3 0];
D = 0;
```

Use `ss` to create the state-space model.

```
sys = ss(A,B,C,D)
```

```
Generalized continuous-time state-space model with 1 outputs, 1 inputs, 2 states, and the follow
  a: Scalar parameter, 2 occurrences.
  b: Scalar parameter, 2 occurrences.

Type "ss(sys)" to see the current value and "sys.Blocks" to interact with the blocks.
```

`sys` is a generalized LTI model (`genss`) with tunable parameters `a` and `b`.

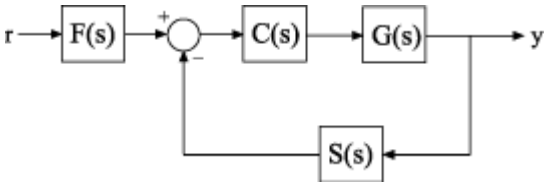
See Also

More About

- “Models with Tunable Coefficients” on page 1-15
- “Tunable Low-Pass Filter” on page 2-59
- “Create Tunable Second-Order Filter” on page 2-60
- “Control System with Tunable Components” on page 2-63

Control System with Tunable Components

This example shows how to create a tunable model of the control system in the following illustration.



The plant response is $G(s) = 1/(s+1)^2$. The model of sensor dynamics is $S(s) = 5/(s+4)$. The controller C is a tunable PID controller, and the prefilter $F = a/(s+a)$ is a low-pass filter with one tunable parameter, a.

Create models representing the plant and sensor dynamics. Since the plant and sensor dynamics are fixed, represent them using numeric LTI models `zpk` and `tf`.

```
G = zpk([],[-1,-1],1);
S = tf(5,[1 4]);
```

Create a tunable representation of the controller C.

```
C = tunablePID('C','PID');
```

`C` is a `tunablePID` object, which is a Control Design Block with a predefined proportional-integral-derivative (PID) structure.

Create a model of the filter $F = a/(s+a)$ with one tunable parameter.

```
a = realp('a',10);
F = tf(a,[1 a]);
```

`a` is a `realp` (real tunable parameter) object with initial value 10. Using `a` as a coefficient in `tf` creates the tunable `genss` model object `F`.

Connect the models together to construct a model of the closed-loop response from r to y.

```
T = feedback(G*C,S)*F
```

```
Generalized continuous-time state-space model with 1 outputs, 1 inputs, 5 states, and the follow
  C: Tunable PID controller, 1 occurrences.
  a: Scalar parameter, 2 occurrences.

Type "ss(T)" to see the current value and "T.Blocks" to interact with the blocks.
```

`T` is a `genss` model object. In contrast to an aggregate model formed by connecting only Numeric LTI models, `T` keeps track of the tunable elements of the control system. The tunable elements are stored in the `Blocks` property of the `genss` model object.

Display the tunable elements of `T`.

```
T.Blocks
```

```
ans = struct with fields:
    C: [1x1 tunablePID]
```

```
a: [1x1 realp]
```

You can use tuning commands such as `systune` to tune the free parameters of `T` to meet design requirements you specify.

See Also

Related Examples

- “Tunable Low-Pass Filter” on page 2-59
- “Create Tunable Second-Order Filter” on page 2-60
- “Create State-Space Model with Both Fixed and Tunable Parameters” on page 2-62

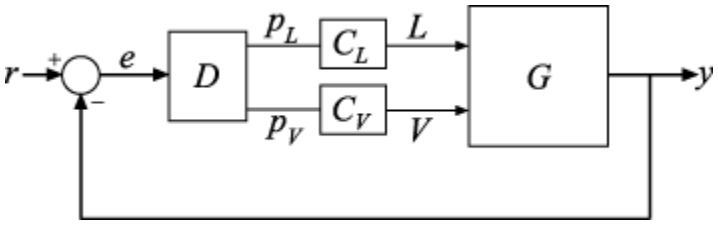
More About

- “Models with Tunable Coefficients” on page 1-15

Control System with Multichannel Analysis Points

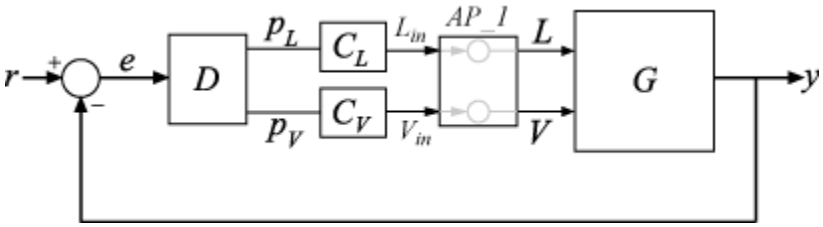
This example shows how to insert multichannel analysis points in a generalized state-space model of a MIMO control system.

Consider the following two-input, two-output control system.



The plant `G` has two inputs and two outputs. Therefore, the line marked `y` in the block diagram represents two signals, `y(1)` and `y(2)`. Similarly, `r` and `e` each represent two signals.

Suppose you want to create tuning requirements or extract responses that require injecting or measuring signals at the locations `L` and `V`. To do so, create an `AnalysisPoint` block and include it in the closed-loop model of the control system as shown in the following illustration.



To create a model of this system, first create the numeric LTI models and control design blocks that represent the plant and controller elements. `D` is a tunable gain block, and `C_L` and `C_V` are tunable PI controllers. Suppose the plant model is the following:

$$G(s) = \frac{1}{75s+1}\begin{bmatrix} 87.8 & -86.4 \\ 108.2 & -109.6 \end{bmatrix}.$$

```
s = tf('s');
G = [87.8 -86.4 ; 108.2 -109.6]/(75*s+1);

D = tunableGain('Decoupler',eye(2));
C_L = tunablePID('C_L','pi');
C_V = tunablePID('C_V','pi');
```

Create an `AnalysisPoint` block that bundles together the `L` and `V` channels.

```
AP_1 = AnalysisPoint('AP_1',2)
```

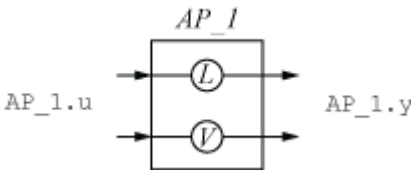
```
Multi-channel analysis point at locations:
   AP_1(1)
   AP_1(2)

Type "ss(AP_1)" to see the current value.
```

For convenience, rename the channels to match the corresponding signals.

```
AP_1.Location = {'L';'V'}
```

```
Multi-channel analysis point at locations:
   L
   V

Type "ss(AP_1)" to see the current value.
```

The following diagram illustrates the input names, output names, and channel names (locations) in the block `AP_1`.



The input and output names of the `AnalysisPoint` block are distinct from the channel names. Use the channel names to refer to the analysis-point locations when extracting responses or defining design goals for tuning. You can use the input and output names `AP_1.u` and `AP_1.y`, for example, when interconnecting blocks using the `connect` command.

You can now build the closed-loop model of the control system. First, join all the plant and controller blocks along with the first `AnalysisPoint` block.

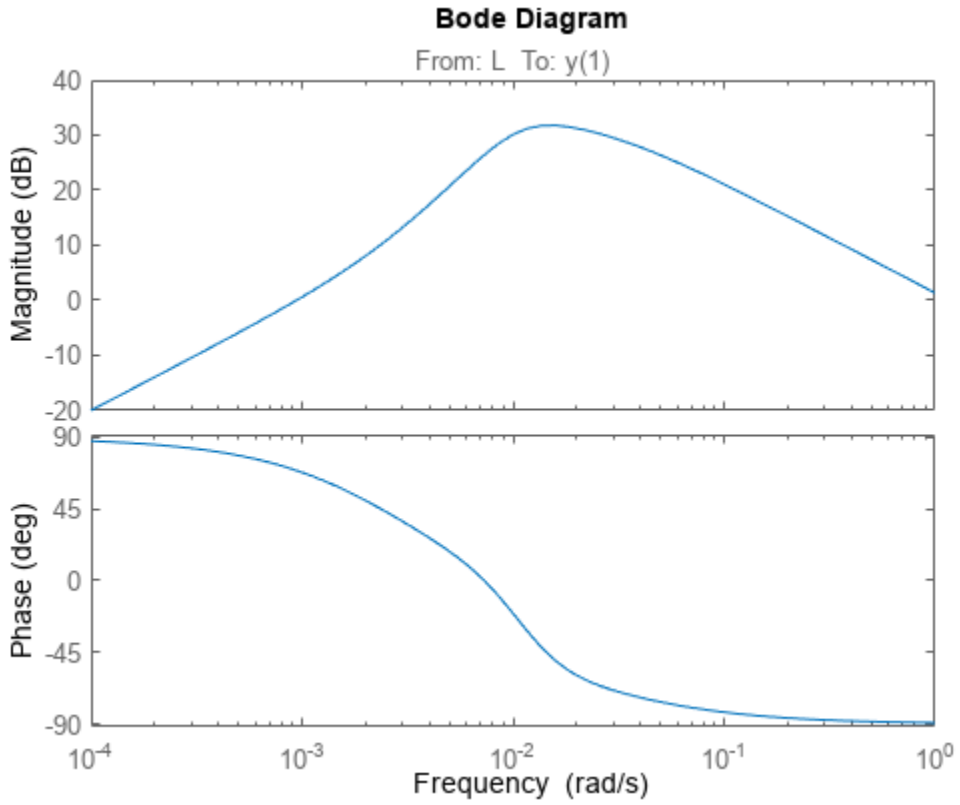
```
GC = G*AP_1*append(C_L,C_V)*D;
```

Then, close the feedback loop. Recall that `GC` has two inputs and outputs.

```
CL = feedback(GC,eye(2));
```

You can now use the analysis points for analysis or tuning. For example, extract the SISO closed-loop transfer function from `'L'` to the first output. Assign a name to the output so you can reference it in analysis functions. The software automatically expands the assigned name `'y'` to the vector-valued output signals `{y(1),y(2)}`.

```
CL.OutputName = 'y';
TLy1 = getIOTransfer(CL,'L','y(1)');
bodeplot(TLy1);
```



See Also

`AnalysisPoint`

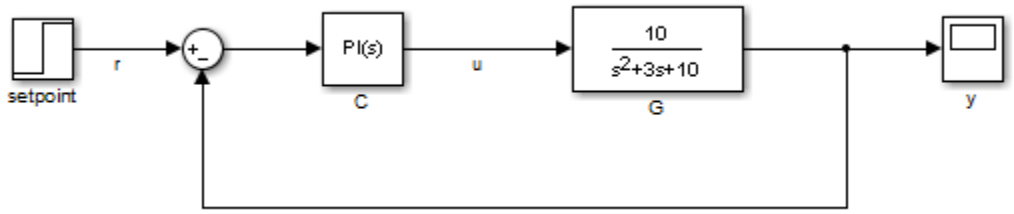
More About

- “Mark Signals of Interest for Control System Analysis and Design” on page 2-68

Mark Signals of Interest for Control System Analysis and Design

Analysis Points

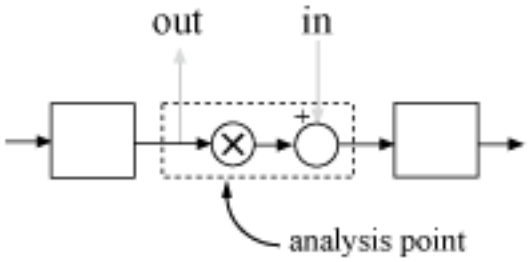
Whether you model your control system in MATLAB or Simulink, use analysis points to mark points of interest in the model. Analysis points allow you to access internal signals, perform open-loop analysis, or specify requirements for controller tuning. In the block diagram representation, an analysis point 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:

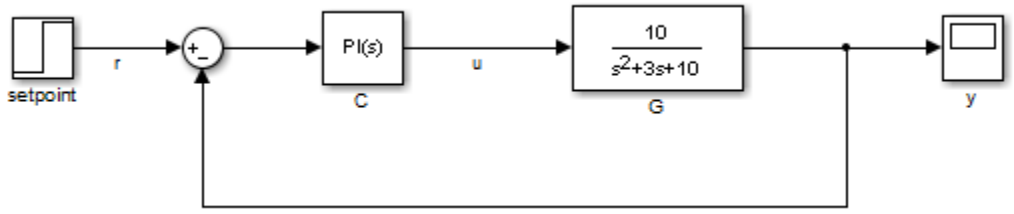
```
Tuy = getIOTransfer(T,'u','y');
stepplot(Tuy)
```

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**

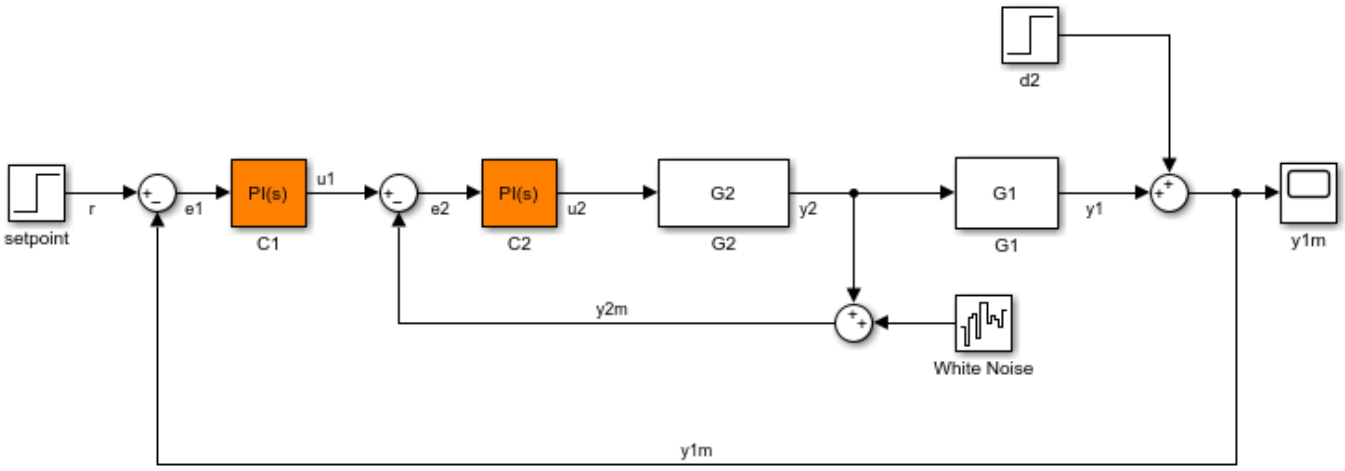
When you define a signal as an open-loop point, analysis functions such as `getIOTransfer` always enforce a loop break at that signal during linearization. All open-loop analysis point types are equivalent within an `slLinearizer` or `slTuner` interface. For more information on how the software treats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

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” (Simulink Control Design).

```
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.

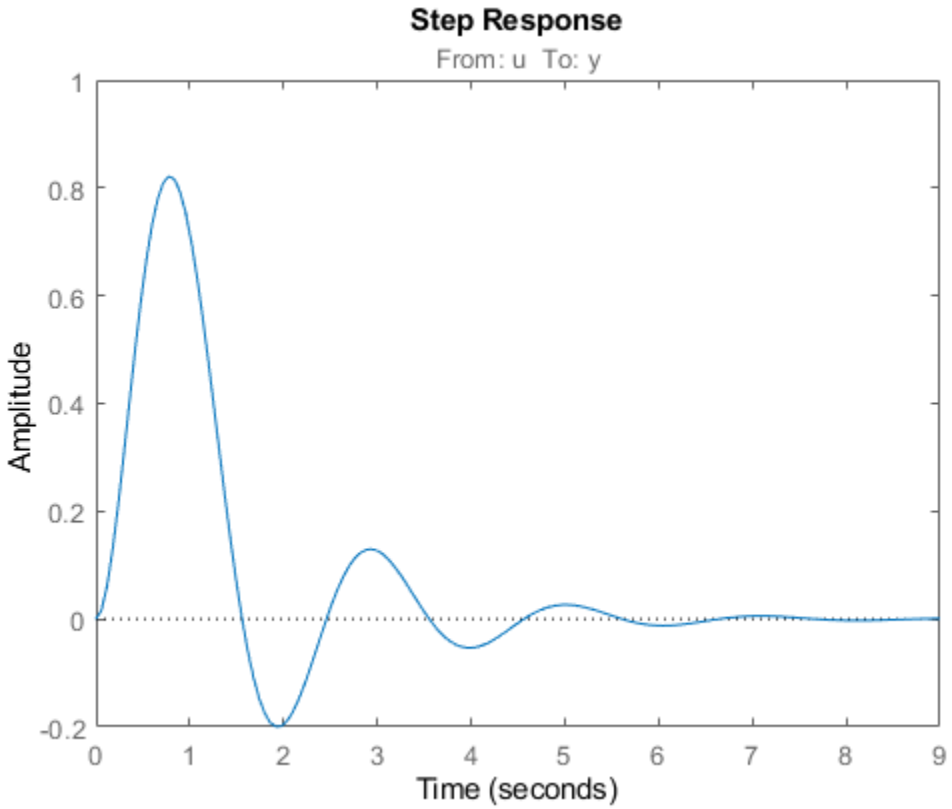
To view the available analysis points, use the `getPoints` function. You can view the analysis for models created:

- 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:

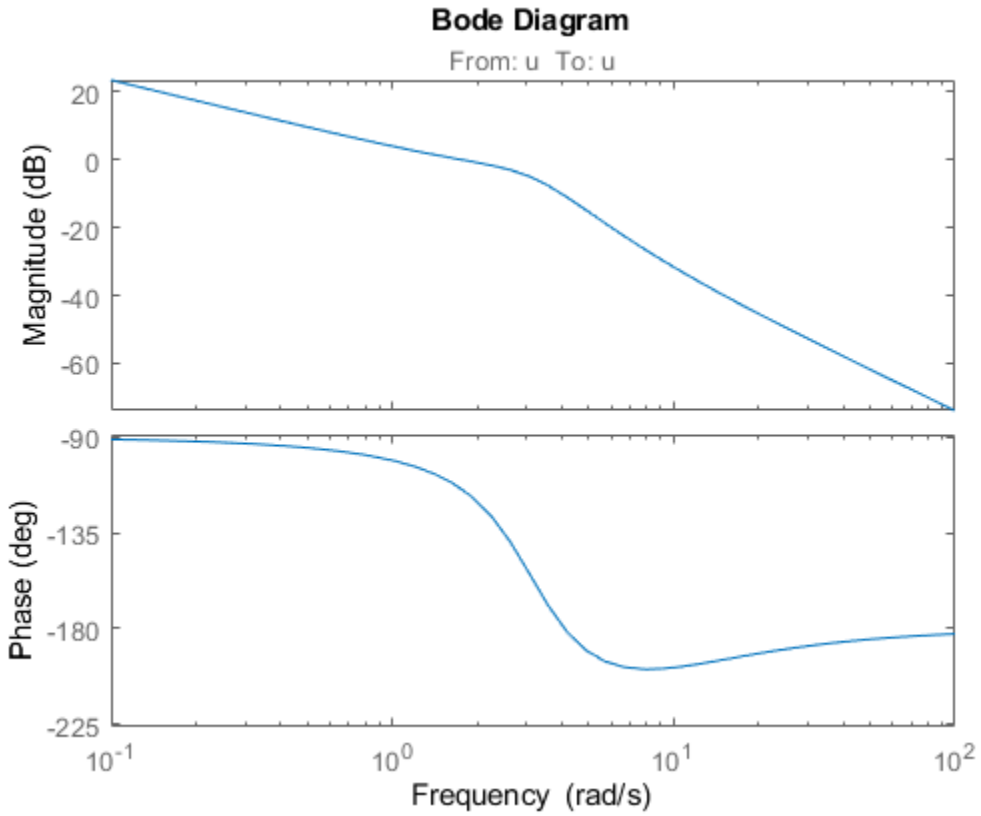
- Computing a closed-loop response.

  ```
  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`.

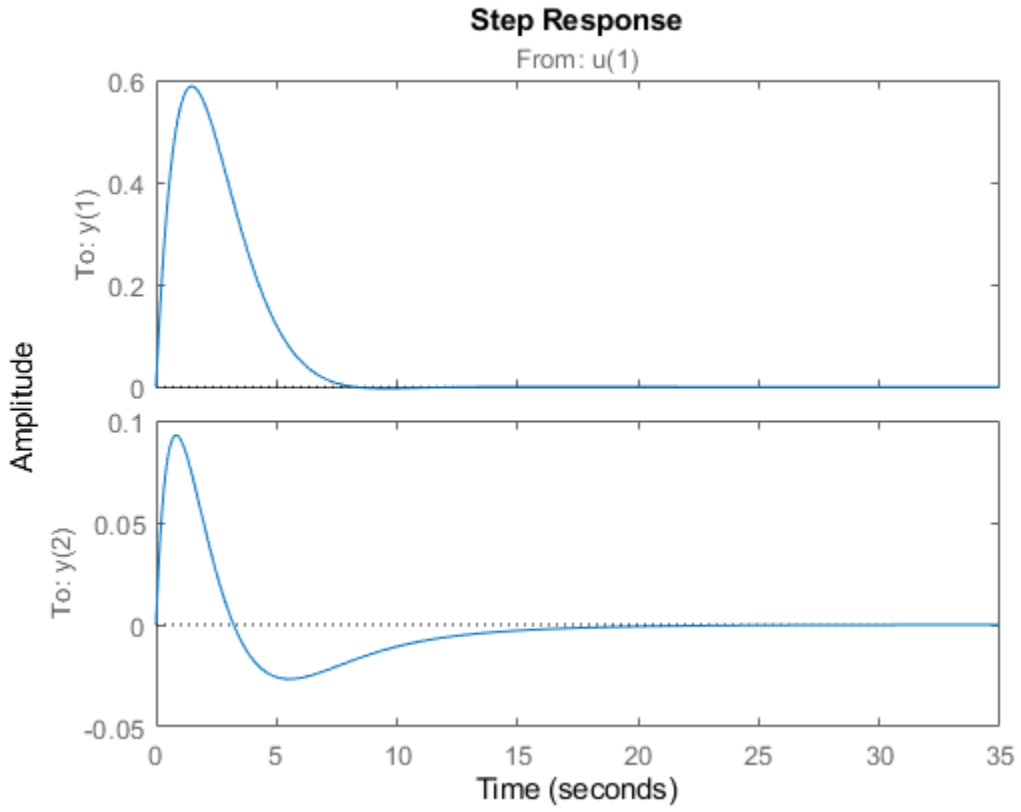
```
ioSys = getIOTransfer(ST,'u1','y1');
sensG2 = getSensitivity(ST,'G2');
R = TuningGoal.Margins('u1',10,60);
```

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

`AnalysisPoint` | `getPoints` | `getIOTransfer`

More About

- “Control System with Multichannel Analysis Points” on page 2-65
- “Mark Analysis Points in Closed-Loop Models” on page 4-11

Model Arrays

What Are Model Arrays?

In many applications, it is useful to consider collections of multiple model objects. For example, you may want to consider a model with a parameter that varies across a range of values, such as

```
sys1 = tf(1, [1 1 1]);
sys2 = tf(1, [1 1 2]);
sys3 = tf(1, [1 1 3]);
```

and so on. Model arrays are a convenient way to store and analyze such a collection. Model arrays are collections of multiple linear models, stored as elements in a single MATLAB array.

For all models collected in a single model array, the following attributes must be the same:

- The number of inputs and outputs
- The sample time `Ts`
- The time unit `TimeUnit`

Uses of Model Arrays

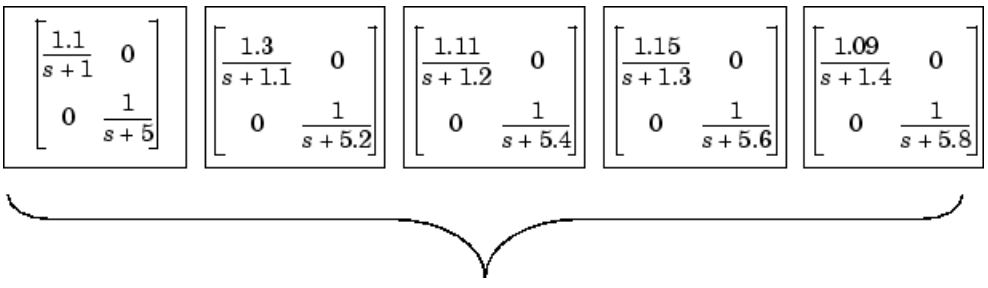
Uses of model arrays include:

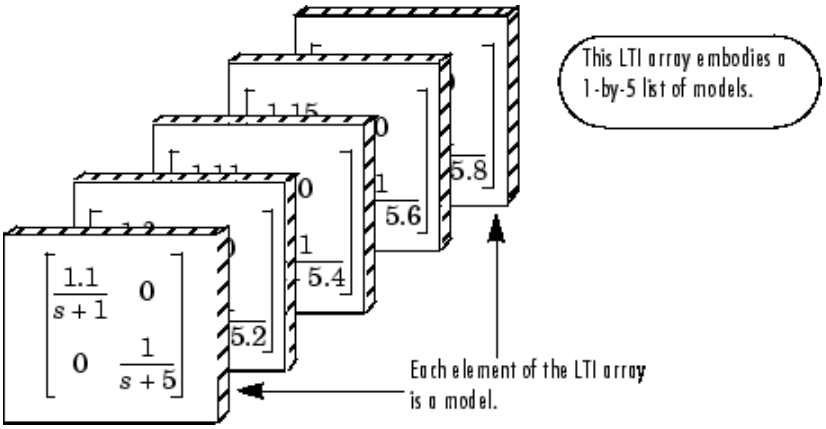
- Representing and analyzing sensitivity to parameter variations
- Validating a controller design against several plant models
- Representing linear models arising from the linearization of a nonlinear system at several operating points
- Storing models obtained from several system identification experiments applied to one plant

Using model arrays, you can apply almost all of the basic model operations that work on single model objects to entire sets of models at once. Functions operate on arrays model by model, allowing you to manipulate an entire collection of models in a vectorized fashion. You can also use analysis functions such as `bode`, `nyquist`, and `step` to model arrays to analyze multiple models simultaneously. You can access the individual models in the collection through MATLAB array indexing.

Visualizing Model Arrays

To visualize the concept of a model array, consider the set of five transfer function models shown below. In this example, each model has two inputs and two outputs. They differ by parameter variations in the individual model components.

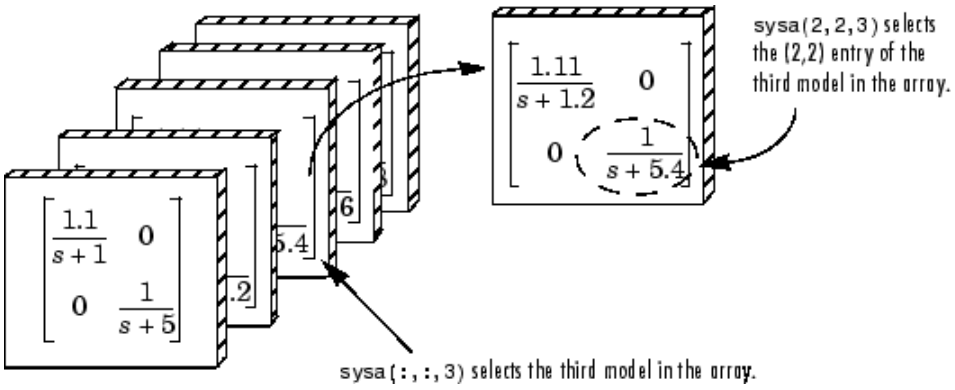




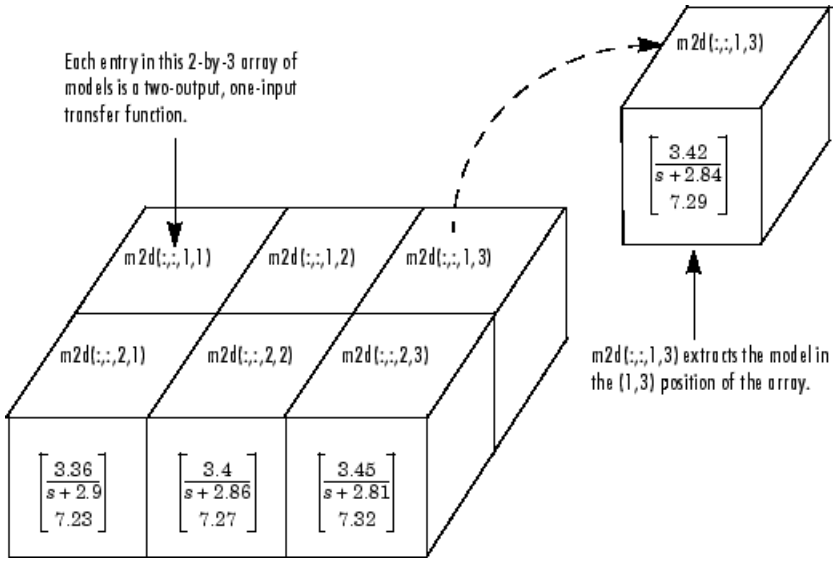
Just as you might collect a set of two-by-two matrices in a multidimensional array, you can collect this set of five transfer function models as a list in a model array under one variable name, say, `sys`. Each element of the model array is a single model object.

Visualizing Selection of Models From Model Arrays

The following illustration shows how indexing selects models from a one-dimensional model array. The illustration shows a 1-by-5 array `sysa` of 2-input, 2-output transfer functions.



The following illustration shows selection of models from the two-dimensional model array `m2d`.



See Also

Related Examples

- “Query Array Size and Characteristics” on page 2-81
- “Select Models from Array” on page 2-79
- “Model Array with Variations in Two Parameters” on page 9-5

Select Models from Array

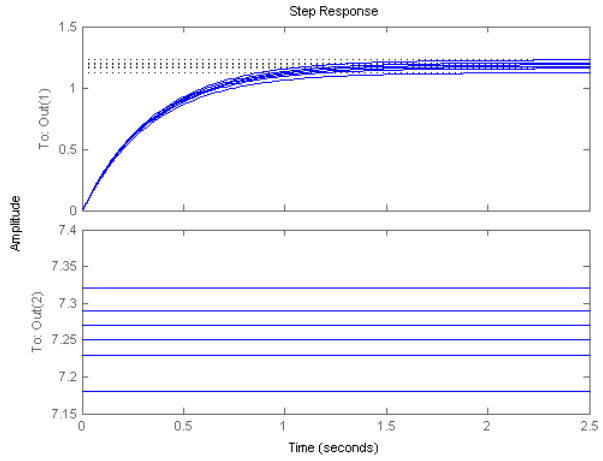
This example shows how to select individual models or sets of models from a model array using array indexing.

1 Load the transfer function array `m2d` into the MATLAB workspace.

```
load LTIexamples m2d
```

2 (Optional) Plot the step response of `m2d`.

```
step(m2d)
```



The step response shows that `m2d` contains six one-input, two-output models. The `step` command plots all of the models in an array on a single plot.

3 (Optional) Examine the dimensions of `m2d`.

```
arraydim = size(m2d)
```

This command produces the result:

```
arraydim =

     2     1     2     3
```

- The first entries of `arraydim`, 2 and 1, show that `m2d` is an array of two-output, one-input transfer functions.
- The remaining entries in `arraydim` give the array dimensions of `m2d`, 2-by-3.

In general, the dimensions of a model array are `[Ny,Nu,S1,...,Sk]`. `Ny` and `Nu` are the numbers of outputs and inputs of each model in the array. `S1,...,Sk` are the array dimensions. Thus, `Si` is the number of models along the *i*th array dimension.

4 Select the transfer function in the second row, first column of `m2d`.

To do so, use MATLAB array indexing.

```
sys = m2d(:,:,2,1)
```

Tip You can also access models using single index referencing of the array dimensions. For example,

```
sys = m2d(:,:,4)
```

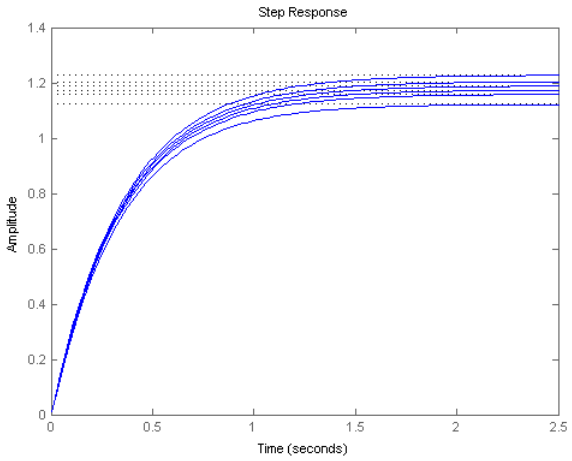
selects the same model as `m2d(:,:,2,1)`.

5 Select the array of subsystems from the first input to the first output of each model in `m2d`.

```
m11 = m2d(1,1,:,:)
```

6 (Optional) Plot the step response of `m11`.

```
step(m11)
```



The step response shows that `m11` is an array of six single-input, single-output (SISO) models.

Note For frequency response data (FRD) models, the array indices can be followed by the keyword `'frequency'` and some expression selecting a subset of the frequency points, as in:

```
sys(outputs,inputs,n1,...,nk,'frequency',SelectedFreqs)
```

See Also

More About

- “Model Arrays” on page 2-76
- “Query Array Size and Characteristics” on page 2-81

Query Array Size and Characteristics

This example shows how to query the size of model arrays, including the number of inputs and outputs of the models in the array, and the array dimensions. It also shows how to query characteristics of the models in the array, such as stability.

Array Size

Model arrays have two different sets of dimensions, the I/O dimensions and the array dimensions. The I/O dimensions are the numbers of inputs and outputs of the models in the array. (Each model in an array must have the same I/O dimensions.) The array dimensions are the dimensions of the array itself. Load a saved model array and query its dimensions.

```
load('queryexample.mat','sysarr')
size(sysarr)
```

```
2x4 array of state-space models.
Each model has 3 outputs, 1 inputs, and 3 states.
```

When you use the `size` command on a model array with no output argument, the display shows the two sets of dimensions.

To obtain the array dimensions as a numeric array, use `size` with an output argument.

```
dims = size(sysarr)
```

```
dims = 1×4

     3     1     2     4
```

The first two entries in `dims` are the I/O dimensions of the models in `sysarr`, which each have three outputs and one input. The remaining entries in `dims` are the dimensions of the array itself. Thus, `sysarr` is a 2-by-4 array of models.

To query the number of dimensions in the array, rather than the values of those dimensions, use `ndims`.

```
dims = ndims(sysarr)
```

```
dims = 4
```

In this case, `sysarr` has 4 = 2 + 2 dimensions: The I/O dimensions (outputs and inputs), and the array dimensions. Query the I/O dimensions alone using the `iosize` command.

```
ios = iosize(sysarr)
```

```
ios = 1×2

     3     1
```

Query the total number of models in the array.

```
N = nmodels(sysarr)
```

```
N = 8
```

Because `sysarr` is a 2-by-4 array of models, this command returns a value of 2 × 4 = 8.

Characteristics of Models in the Array

Query commands such as `isproper` and `isstable` work on model arrays. For example, query whether the models in `sysarr` are stable.

```
Bsiso = isstable(sysarr)
```

```
Bsiso = logical
   1
```

By default, `isstable` returns 1 (`true`) if all of the models in the array are stable. The commands returns 0 (`false`) if one or more of the models is not stable. To perform an element-by-element query of a model array, use the `'elem'` option.

```
Bsiso = isstable(sysarr,'elem')
```

```
Bsiso = 2x4 logical array

   1   1   1   1
   1   1   1   1
```

Now `isstable` returns an array of Boolean values. The dimensions of this array match the array dimensions of `sysarr`. Each entry in the array `Bsiso` indicates whether the corresponding model of `sysarr` is stable. The `'elem'` option works similarly for many query commands.

See Also

More About

- “Model Arrays” on page 2-76
- “Select Models from Array” on page 2-79

Linear Parameter-Varying Models

What are Linear Parameter-Varying Models?

A linear parameter-varying (LPV) system is a linear state-space model whose dynamics vary as a function of certain time-varying parameters called scheduling parameters. In MATLAB, an LPV model is represented in a state-space form using coefficients that are parameter dependent.

Mathematically, an LPV system is represented as:

$$\begin{aligned} dx(t) &= A(p)x(t) + B(p)u(t) \\ y(t) &= C(p)x(t) + D(p)u(t) \\ x(0) &= x_0 \end{aligned} \tag{2-1}$$

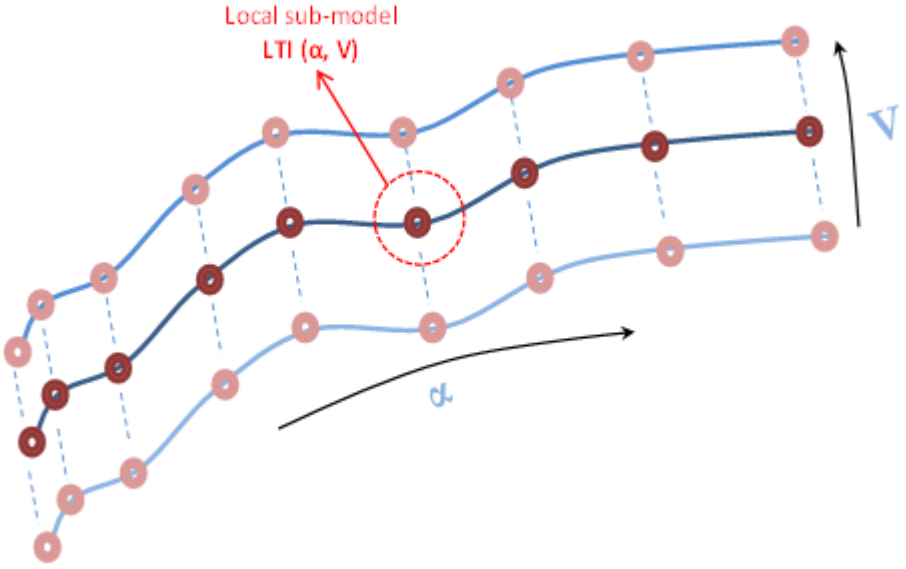
where

- `u(t)` are the inputs
- `y(t)` the outputs
- `x(t)` are the model states with initial value `x0`
- $dx(t)$ is the state derivative vector $\dot{x}$ for continuous-time systems and the state update vector $x(t + \Delta T)$ for discrete-time systems. ΔT is the sample time.
- `A(p)`, `B(p)`, `C(p)` and `D(p)` are the state-space matrices parameterized by the scheduling parameter vector `p`.
- The parameters `p = p(t)` are measurable functions of the inputs and the states of the model. They can be a scalar quantity or a vector of several parameters. The set of scheduling parameters define the *scheduling space* over which the LPV model is defined.

Grid-Based LPV Model

A common way of representing LPV models is as an interpolated array of linear state-space models. A certain number of points in the scheduling space are selected, usually forming a regular grid on page 2-85. An LTI system is assigned to each point, representing the dynamics in the local vicinity of that point. The dynamics at scheduling locations in between the grid points is obtained by interpolation of LTI systems at neighboring points.

For example, the aerodynamic behavior of an aircraft is often scheduled over a grid of incidence angle (α) and wind speed (V) values. For each scheduling parameter, a range of values is chosen, such as α = 0:5:20 degrees, V = 700:100:1400 m/s. For each combination of (α,V) values, a linear approximation of the aircraft behavior is obtained. The local models are connected as shown in the following figure:



Each donut represents a local LTI model, and the connecting curves represent the interpolation rules. The abscissa and ordinate of the surface are the scheduling parameters (α, V).

This form is sometimes called the *grid-based LPV representation*. This is the form used by the LPV System block. For meaningful interpolations of system matrices, all the local models must use the same state basis.

Affine Form of LPV Model

The LPV system representation can be extended to allow offsets in `dx`, `x`, `u` and `y` variables. This form is known as affine form of the LPV model. Mathematically, the following represents an LPV system:

$$\begin{aligned} dx(t) &= A(p)x(t) + B(p)u(t) + \left(\overline{dx}(p) - A(p)\bar{x}(p) - B(p)\bar{u}(p)\right) \\ y(t) &= C(p)x(t) + D(p)u(t) + \left(\bar{y}(p) - C(p)\bar{x}(p) - D(p)\bar{u}(p)\right) \\ x(0) &= x_0 \end{aligned} \tag{2-2}$$

$\overline{dx}(p)$, $\bar{x}(p)$, $\bar{u}(p)$, $\bar{y}(p)$ are the offsets in the values of `dx(t)`, `x(t)`, `u(t)` and `y(t)` at a given parameter value `p = p(t)`.

To obtain such representations of the linear system array, linearize a Simulink model over a batch of operating points (see “Batch Linearization” (Simulink Control Design).) The offsets correspond to the operating points at which you linearized the model.

You can obtain the offsets by returning additional linearization information when calling functions such as `linearize` or `getIOTransfer`. You can then extract the offsets using `getOffsetsForLPV`. For an example, see “LPV Approximation of Boost Converter Model” (Simulink Control Design).

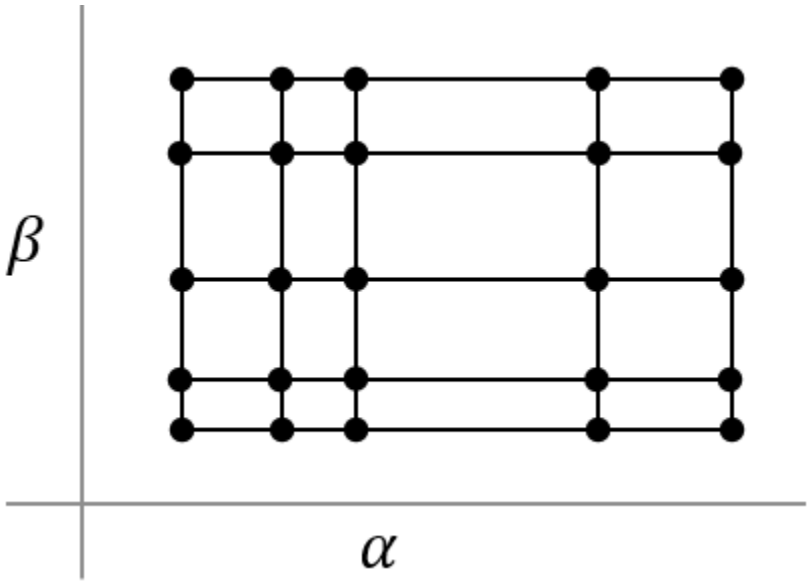
In the affine representation, the linear model at a given point `p = p*` in the scheduling space is:

$$\begin{aligned} d\Delta x(t, p^*) &= A(p^*)\Delta x(t, p^*) + B(p^*)\Delta u(t, p^*) \\ \Delta y(t, p^*) &= C(p^*)\Delta x(t, p^*) + D(p^*)\Delta u(t, p^*) \end{aligned}$$

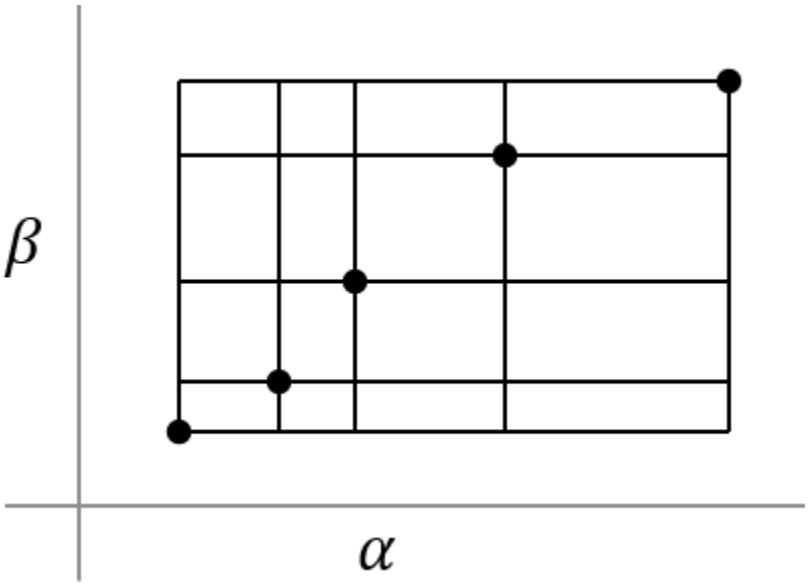
The states of this linear model are related to the states of the overall LPV model (“Equation 2-2”) by $\Delta x(t, p^*) = x(t) - \bar{x}(p^*)$. Similarly, $\Delta y(t, p^*) = y(t) - \bar{y}(p^*)$ and $\Delta u(t, p^*) = u(t) - \bar{u}(p^*)$.

Regular vs. Irregular Grids

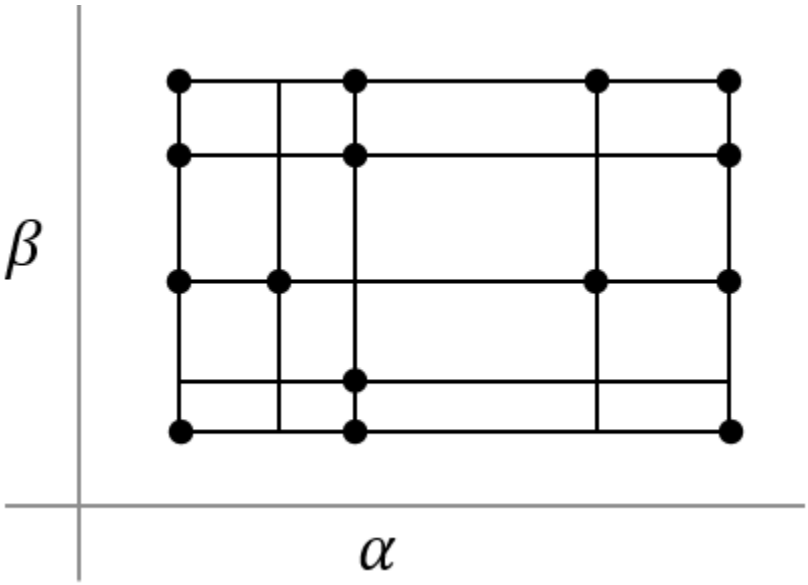
Consider a system that uses two scheduling parameters, α and β. When α and β vary monotonically, a regular grid is formed, as shown in the next figure. The state space array contains a value at every combination of α and β values. Regular grid does not imply uniform spacing between values.



When parameters co-vary, that is, α and β increase together, an irregular grid is formed. The system array parameters are available only along the diagonal in the parameter plane.



If certain samples are missing from an otherwise regular grid, the grid is considered to be irregular.



Use Model Arrays to Create Linear Parameter-Varying Models

The array of state-consistent linear models that define an LPV model are represented by an array of state-space model objects. For more information on model arrays, see “Model Arrays”.

The system array size is equal to the grid size in scheduling space. In the aircraft example, α takes 5 values in the 0–20 degrees range and V takes 8 values in the 700–1400 m/s range. If you define a linear model at every combination of (α,V) values (i.e., the grid is regular), the grid size is 5-by-8. Therefore, the model array size must be 5-by-8.

The information about scheduling parameters is attached to the linear model array using its `SamplingGrid` property. The value of the `SamplingGrid` property must be a structure with as many fields as there are scheduling parameters. For each field, the value must be set to all the values assumed by the corresponding variable in the scheduling space.

For the aircraft example, you can define the `SamplingGrid` property as:

```
Alpha = 0:5:20;
V = 700:100:1400;
[Alpha_Grid,V_Grid] = ndgrid(Alpha, V);
linsysArray.SamplingGrid = struct('Alpha',Alpha_Grid,'V',V_Grid);
```

Approximate Nonlinear Systems using LPV Models

In the same way as a linear model provides the approximation of system behavior at a given operating condition, an LPV model provides the approximation of the behavior over a span on operating conditions. A common approach for constructing the LPV model is by batch trimming and linearization, followed by stacking the local models in a state-space model array.

Note When obtaining linear models by linearization, do not reduce or alter the state variables used by the models.

The operating region is usually of a high dimension because it consists of all the input and state variables. Generating or interpolating local models in such high-dimensional spaces is usually infeasible. A simpler approach is to use a small set of scheduling parameters as a proxy for the operating space variables. The scheduling parameters are derived from the inputs and state variables of the original system. You must choose the values carefully so that for a fixed value of the scheduling parameters, the system behavior is approximately linear. This approach is not always possible.

Consider a nonlinear system described by the following equations:

$$\dot{x}_1 = x_1^2 + x_2^2$$
$$\dot{x}_2 = -2x_1 - 3x_2 + 2u$$
$$y = x_1 + 2$$

Suppose you use $p(t) = \dot{x}_1$ as a scheduling variable. At a given time instant $t = t_0$, you have:

$$\dot{x}_1 \approx 2x_1(t_0)x_1 + 2x_2(t_0)x_2 - \dot{x}_1(t_0)$$
$$\dot{x}_2 = -2x_1 - 3x_2 + 2u$$
$$y = x_1 + 2$$

Thus, the dynamics are linear (affine) in the neighborhood of a given value of $\dot{x}$. The approximation holds for all time spans and values of input u as long as of $\dot{x}$ does not deviate much from its nominal value at sampling point t_0. Note that scheduling on input u or states x_1 or x_2 does not help locally linearize the system. Therefore, they are not good candidates for scheduling parameters.

For an example of this approach, see “Approximate Nonlinear Behavior Using Array of LTI Systems” (Simulink Control Design).

Applications of Linear Parameter-Varying Models

Modeling Multimode Dynamics

You can use LPV models to represent systems that exhibit multiple modes (regimes) of operation. Examples of such systems include colliding bodies, systems controlled by operator switches, and approximations of systems affected by dry friction and hysteresis effects. For an example, see “Using LTI Arrays for Simulating Multi-Mode Dynamics” on page 2-89.

Proxy Modeling for Faster Simulations

This approach is useful for generating surrogate models that you can use in place of the original system for enabling faster simulations, reducing memory footprint of target hardware code, and hardware-in-loop (HIL) simulations. You can also use surrogate models of this type for designing gain-scheduled controllers and for initializing the parameter estimation tasks in Simulink. For an example of approximating a general nonlinear system behavior by an LPV model, see “Approximate Nonlinear Behavior Using Array of LTI Systems” (Simulink Control Design).

LPV models can help speed up the simulation of physical component based systems, such as those built using Simscape Multibody and Simscape Electrical Power Systems software. For an example of this approach, see “LPV Approximation of Boost Converter Model” (Simulink Control Design).

See Also

LPV System | `getOffsetsForLPV`

More About

- “Using LTI Arrays for Simulating Multi-Mode Dynamics” on page 2-89
- “Approximate Nonlinear Behavior Using Array of LTI Systems” (Simulink Control Design)
- “LPV Approximation of Boost Converter Model” (Simulink Control Design)

Using LTI Arrays for Simulating Multi-Mode Dynamics

This example shows how to construct a Linear Parameter Varying (LPV) representation of a system that exhibits multi-mode dynamics.

Introduction

We often encounter situations where an elastic body collides with, or presses against, a possibly elastic surface. Examples of such situations are:

- An elastic ball bouncing on a hard surface.
- An engine throttle valve that is constrained to close to no more than 90^o using a hard spring.
- A passenger sitting on a car seat made of polyurethane foam, a viscoelastic material.

In these situations, the motion of the moving body exhibits different dynamics when it is moving freely than when it is in contact with a surface. In the case of a bouncing ball, the motion of the mass can be described by rigid body dynamics when it is falling freely. When the ball collides and deforms while in contact with the surface, the dynamics have to take into account the elastic properties of the ball and of the surface. A simple way of modeling the impact dynamics is to use lumped mass spring-damper descriptions of the colliding bodies. By adjusting the relative stiffness and damping coefficients of the two bodies, we can model the various situations described above.

Modeling Bounce Dynamics

Figure 1 shows a mass-spring-damper model of the system. Mass 1 is falling freely under the influence of gravity. Its elastic properties are described by stiffness constant k_1 and damping coefficient c_1. When this mass hits the fixed surface, the impact causes Mass 1 and Mass 2 to move downwards together. After a certain "residence time" during which the Mass 1 deforms and recovers, it loses contact with Mass 2 completely to follow a projectile motion. The overall dynamics are thus broken into two distinct modes - when the masses are not in contact and when they are moving jointly.

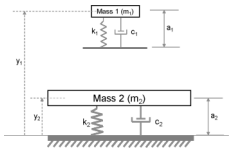


Figure 1: Elastic body bouncing on a fixed elastic surface.

The unstretched (load-free) length of spring attached to Mass 1 is a_1, while that of Mass 2 is a_2. The variables $y_1(t)$ and $y_2(t)$ denote the positions of the two masses. When the masses are not in contact ("Mode 1"), their motions are governed by the following equations:

$$\ddot{y}_1 = -g$$

$$m_2\ddot{y}_2 + c_2\dot{y}_2 + k_2(y_2 - a_2) = -m_2 g$$

with initial conditions $y_1(0) = h_1$, $\dot{y}_1(0) = 0$, $y_2(0) = h_2$, $\dot{y}_2(0) = 0$. h_1 is the height from which Mass 1 is originally dropped. $h_2 = a_2$ is the initial location of Mass 2 which corresponds to an unstretched state of its spring.

When Mass 1 touches Mass 2 ("Mode 2"), their displacements and velocities get interlinked. The governing equations in this mode are:

$$m_1\ddot{y}_1 + c_1(\dot{y}_1 - \dot{y}_2) + k_1(y_1 - y_2 - a_1) = -m_1 g$$

$$m_2\ddot{y}_2 + c_2\dot{y}_2 + k_2(y_2 - a_2) - c_1(\dot{y}_1 - \dot{y}_2) - k_1(y_1 - y_2 - a_1) = -m_2 g$$

with $y_1(t_c) = y_2(t_c)$, where t_c is the time at which Mass 1 first touches Mass 2.

LPV Representation

The governing equations are linear and time invariant. However, there are two distinct behavioral modes corresponding to different equations of motion. Both modes are governed by sets of second order equations. If we pick the positions and velocities of the masses as state variables, we can represent each mode by a 4th order state-space equation.

In the state-space view, it becomes possible to treat the two modes as a single system whose coefficients change as a function of a certain condition which determines which mode is active. The condition is, of course, whether the two masses are moving freely or jointly. Such a representation, where the coefficients of a linear system are parameterized by an external but measurable parameter is called a Linear Parameter Varying (LPV) model. A common representation of an LPV model is by means of an array of linear state-space models and a set of scheduling parameters that dictate the rules for choosing the correct model under a given condition. The array of linear models must all be defined using the same state variables.

For our example, we need two state-space models, one for each mode of operation. We also need to define a scheduling variable to switch between them. We begin by writing the above equations of motion in state-space form.

Define values of masses and their spring constants.

```
m1 = 7;     % first mass (g)
k1 = 100;   % spring constant for first mass (g/s^2)
c1 = 2;     % damping coefficient associated with first mass (g/s)

m2 = 20;    % second mass (g)
k2 = 300;   % spring constant for second mass (g/s^2)
c2 = 5;     % damping coefficient associated with second mass (g/s)

g = 9.81;   % gravitational acceleration (m/s^2)

a1 = 12;    % uncompressed lengths of spring 1 (mm)
a2 = 20;    % uncompressed lengths of spring 2 (mm)

h1 = 100;   % initial height of mass m1 (mm)
h2 = a2;    % initial height of mass m2 (mm)
```

First mode: state-space representation of dynamics when the masses are not in contact.

```
A11 = [0 1; 0 0];
B11 = [0; -g];
C11 = [1 0];
D11 = 0;

A12 = [0 1; -k2/m2, -c2/m2];
B12 = [0; -g+(k2*a2/m2)];
```

```
C12 = [1 0];
D12 = 0;

A1 = blkdiag(A11, A12);
B1 = [B11; B12];
C1 = blkdiag(C11, C12);
D1 = [D11; D12];

sys1 = ss(A1,B1,C1,D1);
```

Second mode: state-space representation of dynamics when the masses are in contact.

```
A2 = [ 0        1,         0,             0; ...
      -k1/m1,   -c1/m1,    k1/m1,         c1/m1;...
      0,        0,         0,             1; ...
      k1/m2,    c1/m2,     -(k1+k2)/m2,   -(c1+c2)/m2];

B2 = [0; -g+k1*a1/m1; 0; -g+(k2/m2*a2)-(k1/m2*a1)];
C2 = [1 0 0 0; 0 0 1 0];
D2 = [0;0];

sys2 = ss(A2,B2,C2,D2);
```

Now we stack the two models `sys1` and `sys2` together to create a state-space array.

```
sys = stack(1,sys1,sys2);
```

Use the information on whether the masses are moving freely or jointly for scheduling. Let us call this parameter "FreeMove" which takes the value of 1 when masses are moving freely and 0 when they are in contact and moving jointly. The scheduling parameter information is incorporated into the state-space array object (`sys`) by using its "SamplingGrid" property:

```
sys.SamplingGrid = struct('FreeMove',[1; 0]);
```

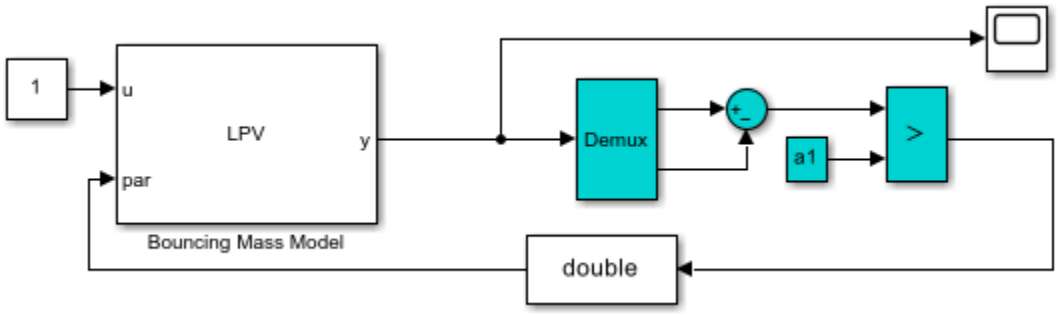
Whether the masses are in contact or not is decided by the relative positions of the two masses; when $y_1 - y_2 > a_1$, the masses are not in contact.

Simulation of LPV Model in Simulink

The state-space array `sys` has the necessary information to represent an LPV model. We can simulate this model in Simulink using the "LPV System" block from the Control System Toolbox™'s block library.

Open the preconfigured Simulink model `LPVBouncingMass.slx`

```
open_system('LPVBouncingMass')
open_system('LPVBouncingMass/Bouncing Mass Model','mask')
```



The block called "Bouncing Mass Model" is an LPV System block. Its parameters are specified as follows:

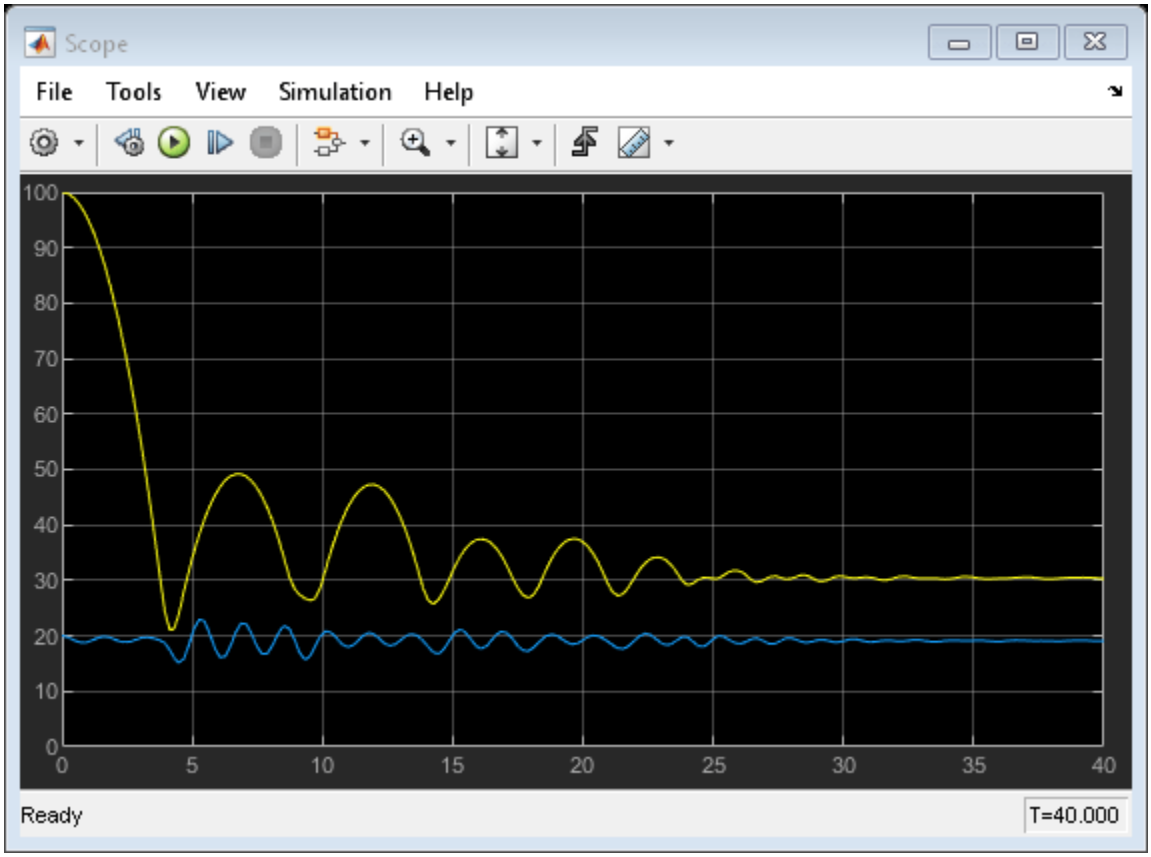
- For "State-space array" field, specify the state-space model array `sys` that was created above.
- For "Initial state" field, specify the initial positions and velocities of the two masses. Note that the state vector is: $[y_1, \dot{y}_1, y_2, \dot{y}_2]$. Specify its value as [h1 0 h2 0]'.
- Under the "Scheduling" tab, set the "Interpolation method" to "Nearest". This choice causes only one of the two models in the array to be active at any time. In our example, the behavior modes are mutually exclusive.
- Under the "Outputs" tab, uncheck all the checkboxes for optional output ports. We will be observing only the positions of the two masses.

The constant block outputs a unit value. This serves as the input to the model and is supplied from the first input port of the LPV block. The block has only one output port which outputs the positions of the two masses as a 2-by-1 vector.

The second input port of the LPV block is for specifying the scheduling signal. As discussed before, this signal represents the scheduling parameter "FreeMove" and takes discrete values 0 (masses in contact) or 1 (masses not in contact). The value of this parameter is computed as a function of the block's output signal. This computation is performed by the blocks with cyan background color. We take the difference between the two outputs (after demuxing) and compare the result to the unstretched length of spring attached to Mass 1. The resulting Boolean result is converted into a double signal which serves as the scheduling parameter value.

We are now ready to perform the simulation.

```
open_system('LPVBouncingMass/Scope')
sim('LPVBouncingMass')
```



The yellow curve shows the position of Mass 1 while the magenta curve shows the position of Mass 2. At the start of simulation, Mass 1 undergoes free fall until it hits Mass 2. The collision causes the Mass 2 to be displaced but it recoils quickly and bounces Mass 1 back. The two masses are in contact for the time duration where $y_1 - y_2 < a_1$. When the masses settle down, their equilibrium values are determined by the static settling due to gravity. For example, the absolute location of Mass 1 is $a_1 + a_2 - m1 * g/k1 - (m2 + m1) * g/k2 = 30.43mm.$

Conclusions

This example shows how a Linear Parameter Varying model can be constructed by using an array of state-space models and suitable scheduling variables. The example describes the case of mutually exclusive modes, although a similar approach can be used in cases where the dynamics behavior at a given value of scheduling parameters is influenced by several linear models.

The LPV System block facilitates the simulation of parameter varying systems. The block also supports code generation for various hardware targets.

Creating Discrete-Time Models

This example shows how to create discrete-time linear models using the `tf`, `zpk`, `ss`, and `frd` commands.

Specifying Discrete-Time Models

Control System Toolbox™ lets you create both continuous-time and discrete-time models. The syntax for creating discrete-time models is similar to that for continuous-time models, except that you must also provide a sample time (sampling interval in seconds).

For example, to specify the discrete-time transfer function:

$$H(z) = \frac{z - 1}{z^2 - 1.85z + 0.9}$$

with sampling period `Ts = 0.1 s`, type:

```
num = [ 1  -1 ];
den = [ 1  -1.85  0.9 ];
H = tf(num,den,0.1)
```

```
H =

        z - 1
  ------------------
  z^2 - 1.85 z + 0.9

Sample time: 0.1 seconds
Discrete-time transfer function.
```

or equivalently:

```
z = tf('z',0.1);
H = (z - 1) / (z^2 - 1.85*z + 0.9);
```

Similarly, to specify the discrete-time state-space model:

$$x[k+1] = 0.5x[k] + u[k]$$

$$y[k] = 0.2x[k].$$

with sampling period `Ts = 0.1 s`, type:

```
sys = ss(.5,1,.2,0,0.1);
```

Recognizing Discrete-Time Systems

There are several ways to determine if your LTI model is discrete:

- The display shows a nonzero sample time value
- `sys.Ts` or `get(sys,'Ts')` return a nonzero sample time value.
- `isdt(sys)` returns true.

For example, for the transfer function `H` specified above,

```
H.Ts
```

```
ans = 0.1000
```

```
isdt(H)
```

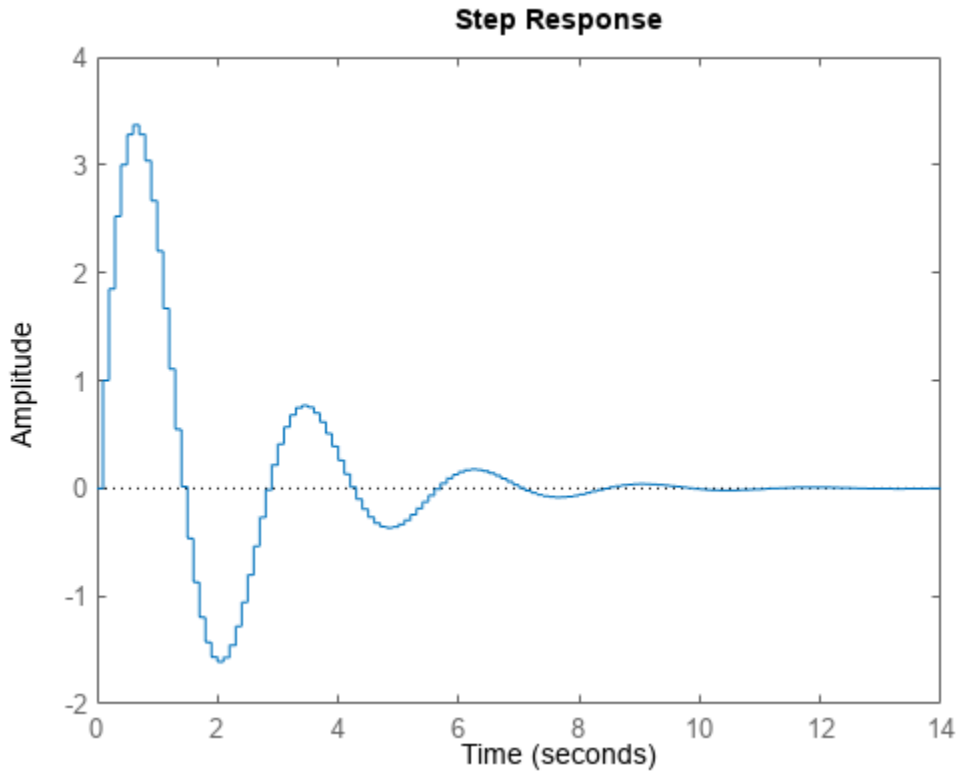
```
ans = logical
   1
```

You can also spot discrete-time systems by looking for the following traits:

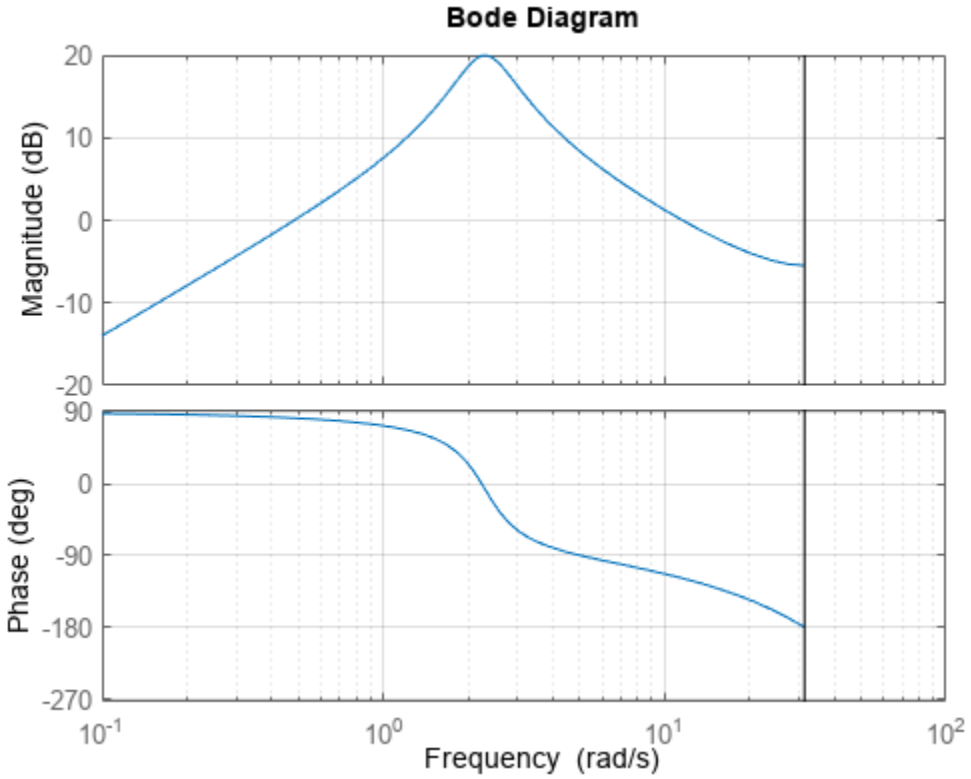
- Time response plots - Response curve has a staircase look owing to its sampled-data nature
- Bode plots - There is a vertical bar marking the Nyquist frequency (pi divided by the sample time).

The following plots show these characteristic traits:

```
step(H)
```




```
bode(H), grid
```


Bode Diagram
Magnitude (dB)
20
10
0
-10
-20
Phase (deg)
90
0
-90
-180
-270
10^{-1}
10^{0}
10^{1}
10^{2}
Frequency (rad/s)

Creating Continuous-Time Models

This example shows how to create continuous-time linear models using the `tf`, `zpk`, `ss`, and `frd` commands.

LTI Model Types

Control System Toolbox™ provides functions for creating four basic representations of linear time-invariant (LTI) models:

- Transfer function (TF) models
- Zero-pole-gain (ZPK) models
- State-space (SS) models
- Frequency response data (FRD) models

These functions take model data as input and create objects that embody this data in a single MATLAB® variable.

Creating Transfer Function Models

Transfer functions (TF) are frequency-domain representations of LTI systems. A SISO transfer function is a ratio of polynomials:

$$H(s) = \frac{A(s)}{B(s)} = \frac{a_1 s^n + a_2 s^{n-1} + \ldots + a_{n+1}}{b_1 s^m + b_2 s^{m-1} + \ldots + b_{m+1}}$$

Transfer functions are specified by their numerator and denominator polynomials `A(s)` and `B(s)`. In MATLAB, a polynomial is represented by the vector of its coefficients, for example, the polynomial

$$s^2 + 2s + 10$$

is specified as `[1 2 10]`.

To create a TF object representing the transfer function:

$$H(s) = \frac{s}{s^2 + 2s + 10}$$

specify the numerator and denominator polynomials and use `tf` to construct the TF object:

```
num = [ 1  0 ];       % Numerator: s
den = [ 1  2  10 ];   % Denominator: s^2 + 2 s + 10
H = tf(num,den)
```

```
H =

        s
  --------------
  s^2 + 2 s + 10

Continuous-time transfer function.
```

Alternatively, you can specify this model as a rational expression of the Laplace variable `s`:

```
s = tf('s');        % Create Laplace variable
H = s / (s^2 + 2*s + 10)
```

```
H =

        s
  --------------
  s^2 + 2 s + 10

Continuous-time transfer function.
```

Creating Zero-Pole-Gain Models

Zero-pole-gain (ZPK) models are the factored form of transfer functions:

$$H(s) = k\frac{(s-z_1)...(s-z_n)}{(s-p_1)...(s-p_m)}$$

Such models expose the roots `z` of the numerator (the zeros) and the roots `p` of the denominator (the poles). The scalar coefficient `k` is called the gain.

To create the ZPK model:

$$H(s) = \frac{-2s}{(s-2)(s^2-2s+2)}$$

specify the vectors of poles and zeros and the gain `k`:

```
z = 0;                  % Zeros
p = [ 2  1+i  1-i ];    % Poles
k = -2;                 % Gain
H = zpk(z,p,k)
```

```
H =

          -2 s
  --------------------
  (s-2) (s^2 - 2s + 2)

Continuous-time zero/pole/gain model.
```

As for TF models, you can also specify this model as a rational expression of `s`:

```
s = zpk('s');
H = -2*s / (s - 2) / (s^2 - 2*s + 2)
```

```
H =

          -2 s
  --------------------
  (s-2) (s^2 - 2s + 2)

Continuous-time zero/pole/gain model.
```

Creating State-Space Models

State-space (SS) models are time-domain representations of LTI systems:

$$\frac{dx}{dt} = Ax(t) + Bu(t)$$

$$y(t) = Cx(t) + Du(t)$$

where `x(t)` is the state vector, `u(t)` is input vector, and `y(t)` is the output trajectory.

State-space models are derived from the differential equations describing the system dynamics. For example, consider the second-order ODE for a simple electric motor:

$$\frac{d^2\theta}{dt^2} + 2\frac{d\theta}{dt} + 5\theta = 3I$$

where `I` is the driving current (input) and `theta` is the angular displacement of the rotor (output). This ODE can be rewritten in state-space form as:

$$\frac{dx}{dt} = Ax + BI \quad A = \begin{bmatrix} 0 & 1 \\ -5 & -2 \end{bmatrix} \quad B = \begin{bmatrix} 0 \\ 3 \end{bmatrix} \quad x = \begin{bmatrix} \theta \\ \frac{d\theta}{dt} \end{bmatrix}$$

$$\theta = Cx + DI \quad C = [1\ 0] \quad D = [0]$$

To create this model, specify the state-space matrices `A, B, C, D` and use `ss` to construct the SS object:

```
A = [ 0  1 ; -5  -2 ];
B = [ 0 ; 3 ];
C = [ 1  0 ];
D = 0;
H = ss(A,B,C,D)
```

```
H =

  A =
       x1  x2
   x1   0   1
   x2  -5  -2

  B =
       u1
   x1   0
   x2   3

  C =
       x1  x2
   y1   1   0

  D =
       u1
   y1   0

Continuous-time state-space model.
```

Creating Frequency Response Data Models

Frequency response data (FRD) models let you store the measured or simulated complex frequency response of a system in an LTI object. You can then use this data as a surrogate model for frequency-domain analysis and design purposes.

For example, suppose you get the following data out of a frequency analyzer:

- Frequency (Hz): 10, 30, 50, 100, 500
- Response: 0.0021+0.0009i, 0.0027+0.0029i, 0.0044+0.0052i, 0.0200-0.0040i, 0.0001-0.0021i

You can create an FRD object containing this data using:

```
freq = [10, 30, 50, 100, 500];
resp = [0.0021+0.0009i, 0.0027+0.0029i, 0.0044+0.0052i, 0.0200-0.0040i, 0.0001-0.0021i];
H = frd(resp,freq,'Units','Hz')
```

```
H =

    Frequency(Hz)           Response
    -------------           --------
          10          2.100e-03 + 9.000e-04i
          30          2.700e-03 + 2.900e-03i
          50          4.400e-03 + 5.200e-03i
         100          2.000e-02 - 4.000e-03i
         500          1.000e-04 - 2.100e-03i

Continuous-time frequency response.
```

Note that frequency values are assumed to be in rad/s unless you specify the `Units` to be Hertz.

Creating MIMO Models

The `tf`, `zpk`, `ss`, and `frd` commands let you construct both SISO and MIMO models. For TF or ZPK models, it is often convenient to construct MIMO models by concatenating simpler SISO models. For example, you can create the 2x2 MIMO transfer function:

$$H(s) = \begin{bmatrix} \frac{1}{s+1} & 0 \\ \frac{s+1}{s^2+s+3} & \frac{-4s}{s+2} \end{bmatrix}$$

using:

```
s = tf('s');
H = [ 1/(s+1) , 0 ; (s+1)/(s^2+s+3) , -4*s/(s+2) ]
```

```
H =

  From input 1 to output...
         1
   1:  -----
       s + 1

          s + 1
   2:  -----------
```

```
       s^2 + s + 3

  From input 2 to output...
   1:  0

       -4 s
   2:  -----
       s + 2

Continuous-time transfer function.
```

Analyzing LTI Models

Control System Toolbox provides an extensive set of functions for analyzing LTI models. These functions range from simple queries about I/O size and order to sophisticated time and frequency response analysis.

For example, you can obtain size information for the MIMO transfer function `H` specified above by typing:

```
size(H)
```

```
Transfer function with 2 outputs and 2 inputs.
```

You can compute the poles using:

```
pole(H)
```

```
ans = 4×1 complex

  -1.0000 + 0.0000i
  -0.5000 + 1.6583i
  -0.5000 - 1.6583i
  -2.0000 + 0.0000i
```

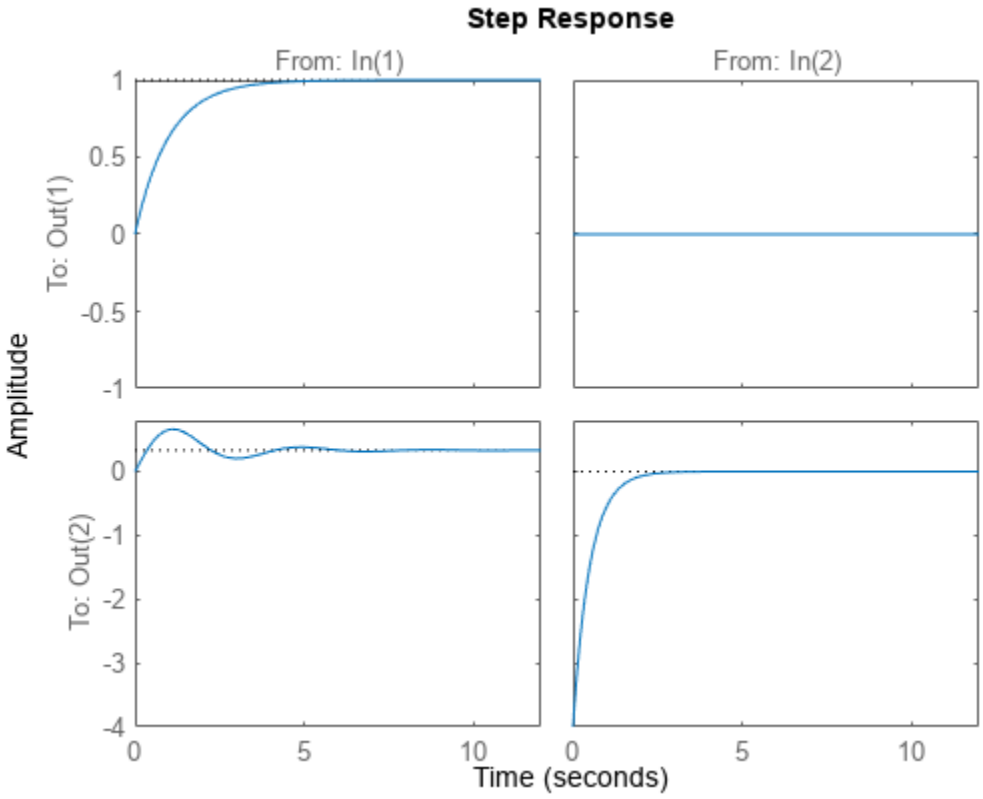
You can ask whether this system is stable using:

```
isstable(H)
```

```
ans = logical
   1
```

Finally, you can plot the step response by typing:

```
step(H)
```



See Also

`tf` | `ss` | `zpk` | `step` | `bode`

Related Examples

- “Creating Discrete-Time Models” on page 2-94
- “Plotting System Responses” on page 7-2

Specifying Time Delays

This example shows how the Control System Toolbox™ lets you represent, manipulate, and analyze any LTI model with a finite number of delays. The delays can be at the system inputs or outputs, between specific I/O pairs, or internal to the model (for example, inside a feedback loop).

Time Delays in LTI Models

Transfer function (TF), zero-pole-gain (ZPK), and frequency response data (FRD) objects offer three properties for modeling delays:

- InputDelay, to specify delays at the inputs
- OutputDelay, to specify delays at the outputs
- IODelay, to specify independent transport delays for each I/O pair.

The state-space (SS) object has three delay-related properties as well:

- InputDelay, to specify delays at the inputs
- OutputDelay, to specify delays at the outputs
- InternalDelay, to keep track of delays when combining models or closing feedback loops.

The ability to keep track of internal delays makes the state-space representation best suited to modeling and analyzing delay effects in control systems. This tutorial shows how to construct and manipulate systems with delays. For more information on how to analyze delay effects, see “Analyzing Control Systems with Delays” on page 8-26.

First-Order Plus Dead Time Models

First-order plus dead time models are commonly used in process control applications. One such example is:

$$P(s) = \frac{5e^{-3.4s}}{s+1}$$

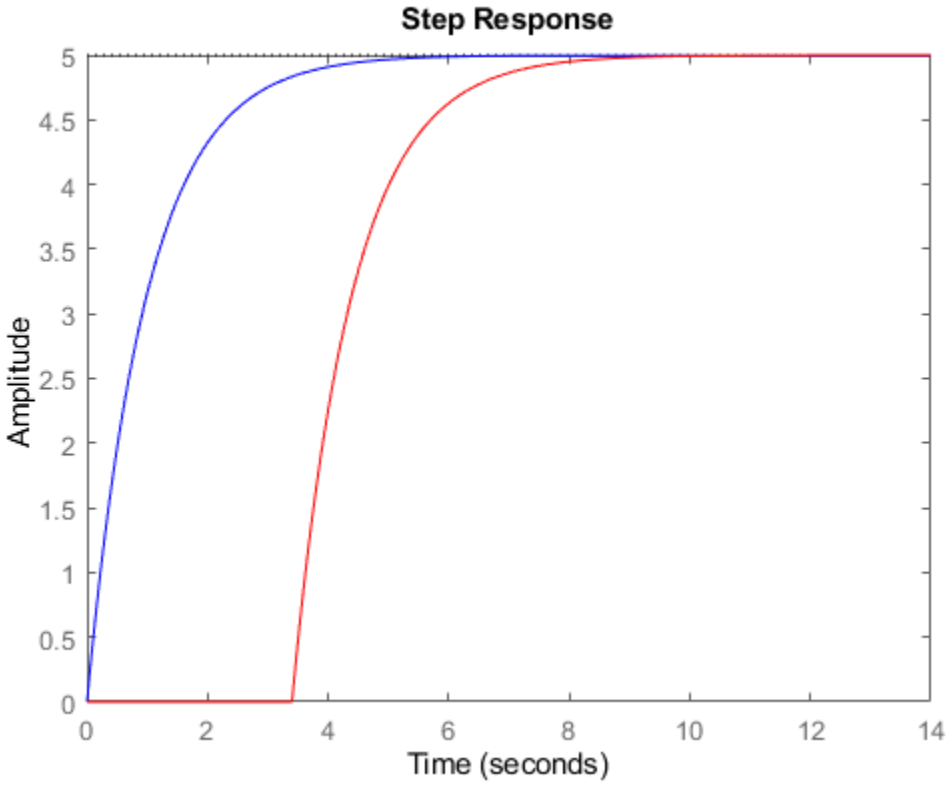
To specify this transfer function, use

```
num = 5;
den = [1 1];
P = tf(num,den,'InputDelay',3.4)
```

```
P =

                5
  exp(-3.4*s) * -----
              s + 1

Continuous-time transfer function.
```

As expected, the step response of P is a shifted version of the delay-free response:

```
P0 = tf(num,den);
step(P0,'b',P,'r')
```



If the process model has multiple outputs, for example:

$$P(s) = \begin{bmatrix} \frac{5e^{-3.4s}}{s+1} \\ \frac{-2e^{-2.7s}}{s+3} \end{bmatrix},$$

you can use the OutputDelay property to specify a different delay for each output channel:

```
num = {5 ; -2};
den = {[1 1] ; [1 3]};
P = tf(num,den,'OutputDelay',[3.4 ; 2.7])
```

```
P =

  From input to output...
                    5
   1:  exp(-3.4*s) * -----
                    s + 1

                     -2
   2:  exp(-2.7*s) * -----
                    s + 3

Continuous-time transfer function.
```

Next consider a multi-input, multi-output model, e.g.,

$$P(s) = \begin{bmatrix} \frac{5e^{-3.4s}}{s+1} & 1 \\ \frac{-2e^{-2.7s}}{s+3} & \frac{e^{-0.7s}}{s} \end{bmatrix}.$$

Here the delays are different for each I/O pair, so you must use the IODelay property:

```
num = {5 , 1; -2 1};
den = {[1 1] , 1; [1 3], [1 0]};
P = tf(num,den,'IODelay',[3.4  0;2.7 0.7]);
```

A more direct and literal way to specify this model is to introduce the Laplace variable "s" and use transfer function arithmetic:

```
s = tf('s');
P = [ 5*exp(-3.4*s)/(s+1) , 1 ; -2*exp(-2.7*s)/(s+3) , exp(-0.7*s)/s ]
```

```
P =

  From input 1 to output...
                      5
   1:  exp(-3.4*s) * -----
                     s + 1

                      -2
   2:  exp(-2.7*s) * -----
                     s + 3

  From input 2 to output...
   1:  1

                     1
   2:  exp(-0.7*s) * -
                     s

Continuous-time transfer function.
```

Note that in this case, MATLAB® automatically decides how to distribute the delays between the InputDelay, OutputDelay, and IODelay properties.

```
P.InputDelay
P.OutputDelay
P.IODelay
```

```
ans =

     0
     0


ans =

         0
    0.7000


ans =
```

```
    3.4000         0
    2.0000         0
```

The function `totaldelay` sums up the input, output, and I/O delay values to give back the values we entered:

```
totaldelay(P)
```

```
ans =

    3.4000         0
    2.7000    0.7000
```

State-Space Models with Input and Output Delays

Consider the state-space model:

$$\frac{dx}{dt} = -x(t) + u(t-2.5), \quad y(t) = 12x(t).$$

Note that the input signal u(t) is delayed by 2.5 seconds. To specify this model, enter:

```
sys = ss(-1,1,12,0,'InputDelay',2.5)
```

```
sys =

  A =
       x1
   x1  -1

  B =
       u1
   x1   1

  C =
       x1
   y1  12

  D =
       u1
   y1   0

  Input delays (seconds): 2.5

Continuous-time state-space model.
```

A related model is

$$\frac{dx_1}{dt} = -x_1(t) + u(t), \quad y(t) = 12x_1(t-2.5).$$

Here the 2.5 second delay is at the output, as seen by rewriting these state equations as:

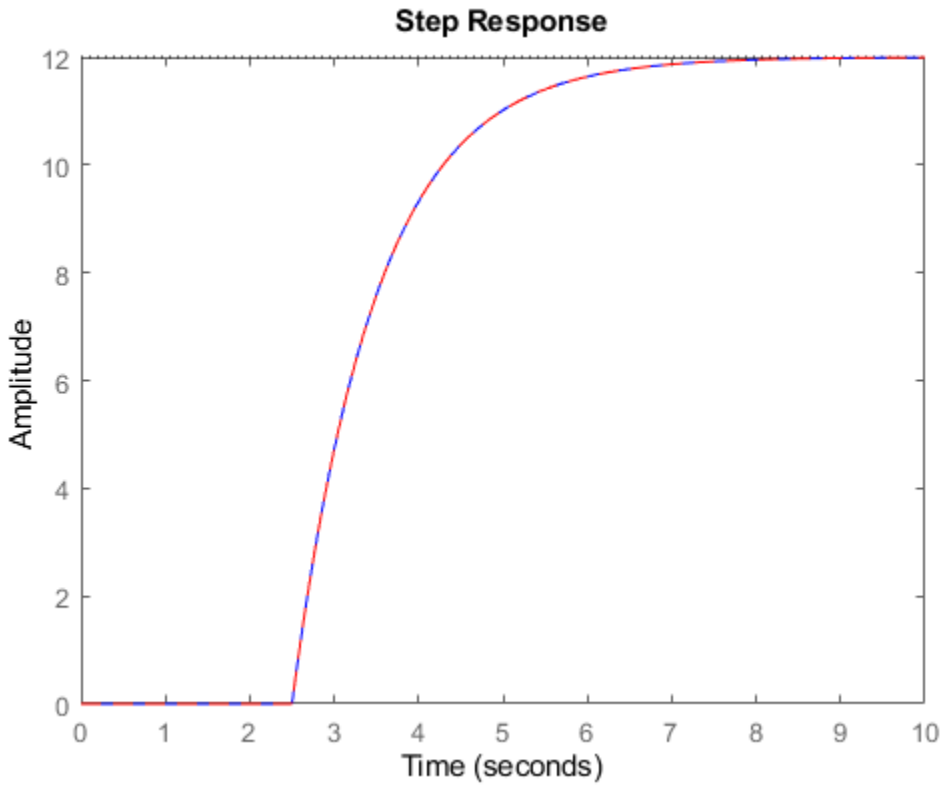
$$\frac{dx_1}{dt} = -x_1(t) + u(t), \quad y_1(t) = 12x_1(t), \quad y(t) = y_1(t - 2.5).$$

You can therefore specify this model as:

```
sys1 = ss(-1,1,12,0,'OutputDelay',2.5);
```

Note that both models have the same I/O response as confirmed by

```
step(sys,'b',sys1,'r--')
```



However, their state trajectories are not the same because the states `x` and `x1` are related by

$$x(t) = x_1(t - 2.5)$$

Combining Models with I/O Delays

So far we have only considered LTI models with transport delays between specific I/O pairs. While this is enough to model many processes, this class of models is not general enough to analyze most control systems with delays, including simple feedback loops with delays. For example, consider the parallel connection:

$$H(s) = H_1(s) + H_2(s) = \frac{1}{s+2} + \frac{5e^{-3.4s}}{s+1}$$

The resulting transfer function

$$H(s) = \frac{s + 1 + (5s + 10)e^{-3.4s}}{(s + 1)(s + 2)}$$

cannot be represented as an ordinary transfer function with a delay at the input or output. To represent $H(s)$, we must switch to the state-space representation and use the notion of "internal delay". State-space (SS) models have the ability to keep track of delays when connecting systems together. Structural information on the delay location and their coupling with the remaining dynamics is encoded in an efficient and fully general manner. Adding the transfer functions $H_1(s)$ and $H_2(s)$ together automatically computes a state-space representation of $H(s)$:

```
H1 = 1/(s+2);
H2 = 5*exp(-3.4*s)/(s+1);
H = H1 + H2
```

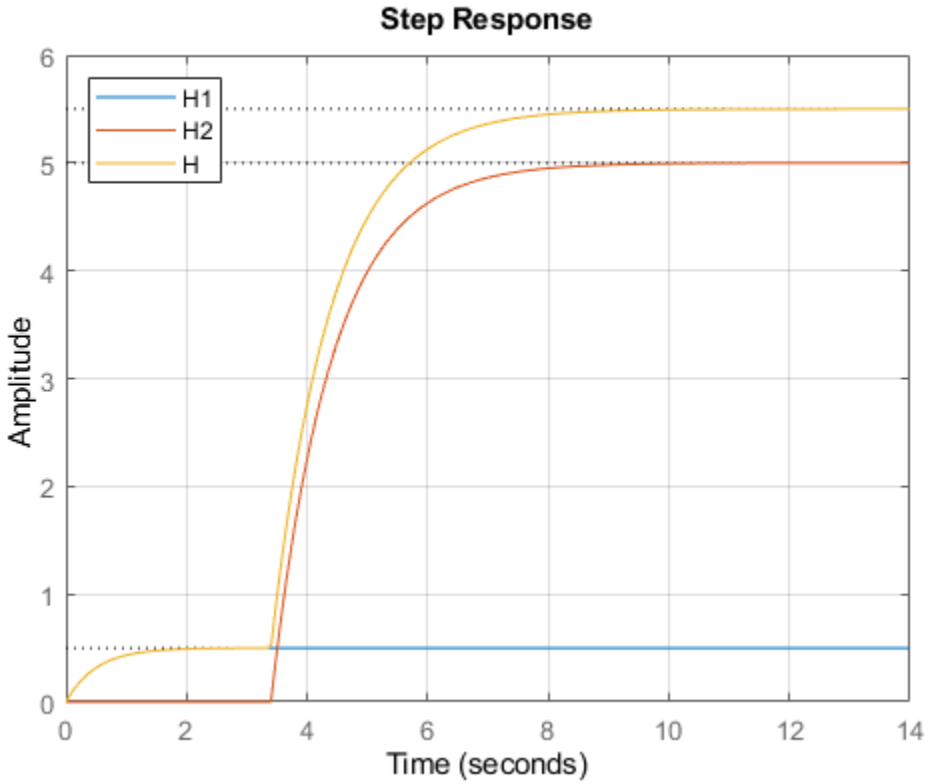
```
H =

  A =
       x1  x2
   x1  -2   0
   x2   0  -1

  B =
       u1
   x1   1
   x2   2

  C =
        x1   x2
   y1    1  2.5

  D =
       u1
   y1   0

  (values computed with all internal delays set to zero)

  Internal delays (seconds): 3.4

Continuous-time state-space model.
```

Note that

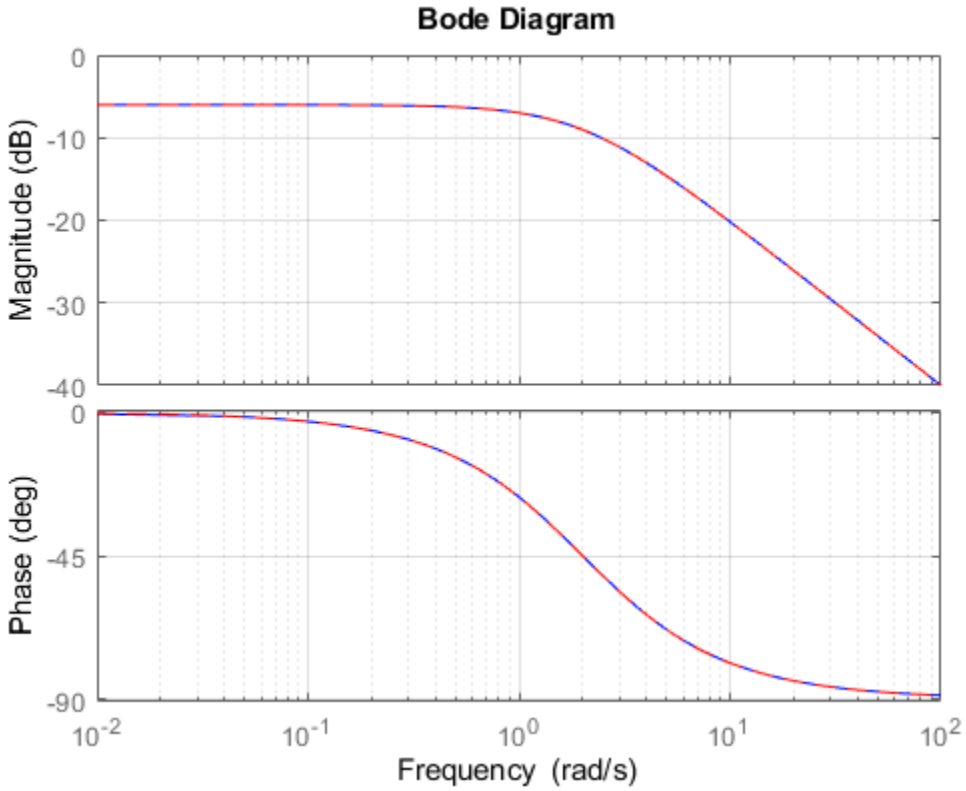
- The delay value of 3.4 is listed as "internal"
- The A,B,C,D data corresponds to the dynamics when all delays are set to zero (zero-order Pade approximation)

It is neither possible nor advisable to look at the transfer function of models with internal delays. Instead, use time and frequency plots to compare and validate models:

```
step(H1,H2,H)
legend('H1','H2','H','Location','NorthWest'), grid
```




```
bode(H1,'b',H-H2,'r--')  % verify that H-H2 = H1
grid
```



Building Models with Internal Delays

Typically, state-space models with internal delays are not created by specifying A,B,C,D data together with a set of internal delays. Rather, you build such models by connecting simpler LTI models (some with I/O delays) in series, parallel, or feedback. There is no limitation on how many delays are involved and how the LTI models are connected together.

For example, consider the control loop shown below, where the plant is modeled as a first-order plus dead time.

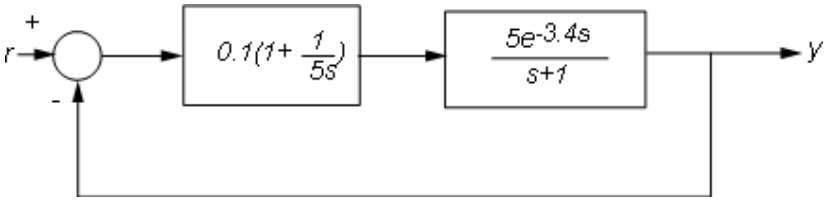
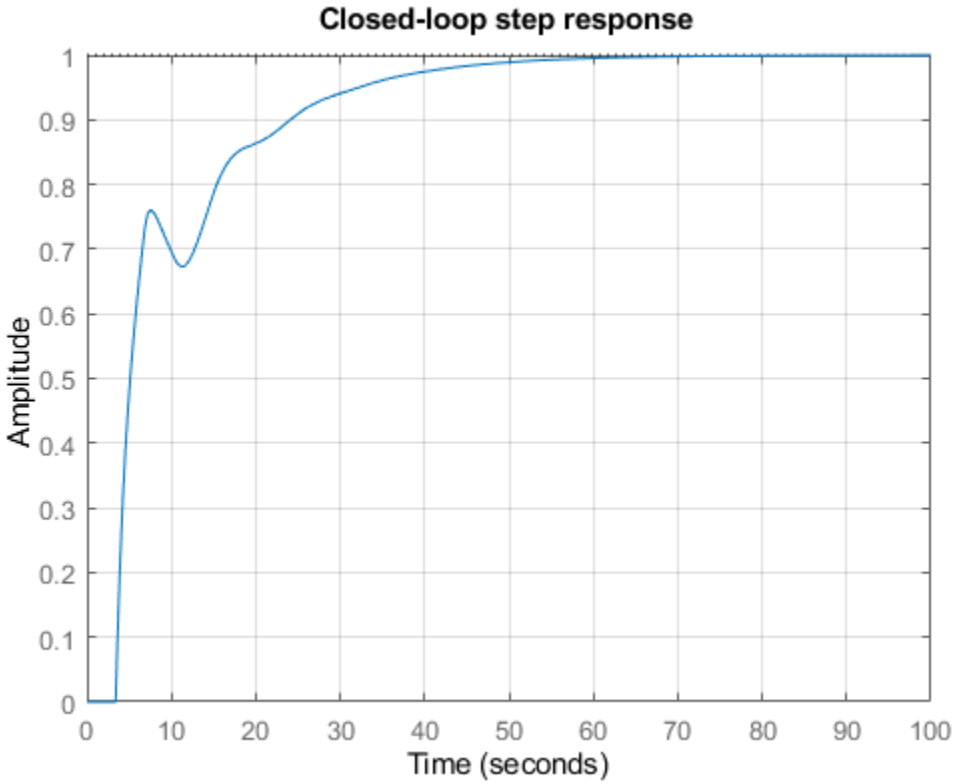


Figure 1: Feedback Loop with Delay.

Using the state-space representation, you can derive a model T for the closed-loop response from r to y and simulate it by

```
P = 5*exp(-3.4*s)/(s+1);
C = 0.1 * (1 + 1/(5*s));
T = feedback(P*C,1);
```

```
step(T,100)
grid, title('Closed-loop step response')
```



For more complicated interconnections, you can name the input and output signals of each block and use `connect` to automatically take care of the wiring. Suppose, for example, that you want to add feedforward to the control loop of Figure 1:

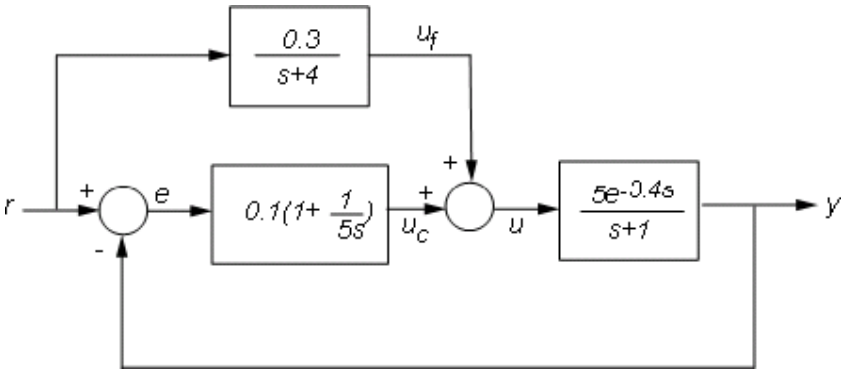


Figure 2: Feedforward and Feedback Control.

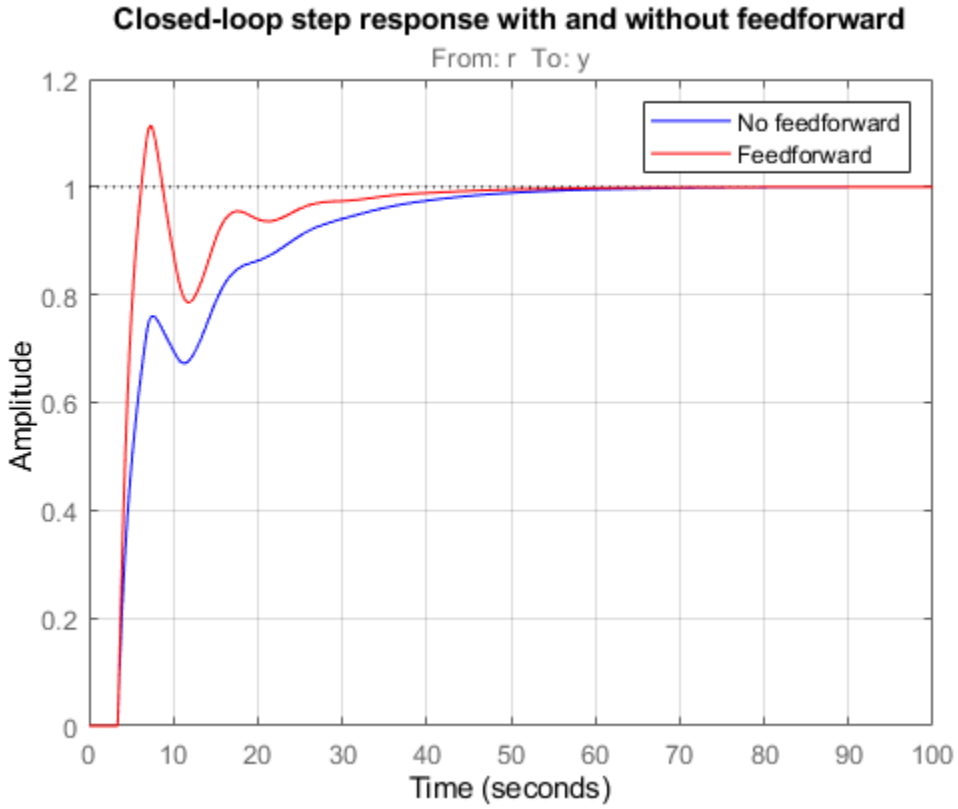
You can derive the corresponding closed-loop model T by

```
F = 0.3/(s+4);
P.u = 'u';  P.y = 'y';
```

```
C.u = 'e';  C.y = 'uc';
F.u = 'r';  F.y = 'uf';
Sum1 = sumblk('e = r-y');
Sum2 = sumblk('u = uf+uc');
Tff = connect(P,C,F,Sum1,Sum2,'r','y');
```

and compare its response with the feedback only design:

```
step(T,'b',Tff,'r',100)
legend('No feedforward','Feedforward')
grid, title('Closed-loop step response with and without feedforward')
```



State-Space Equations with Delayed Terms

A special class of LTI models with delays are state-space equations with delayed terms. The general form is

$$\frac{dx}{dt} = Ax(t) + Bu(t) + \sum_j (A_j x(t-\tau_j) + B_j u(t-\tau_j))$$

$$y(t) = Cx(t) + Du(t) + \sum_j (C_j x(t-\tau_j) + B_j u(t-\tau_j))$$

The function `delayss` helps you specify such models. For example, consider

$$\frac{dx}{dt} = -x(t) - x(t-1.2) + 2u(t-0.5), \quad y(t) = x(t-0.5) + u(t)$$

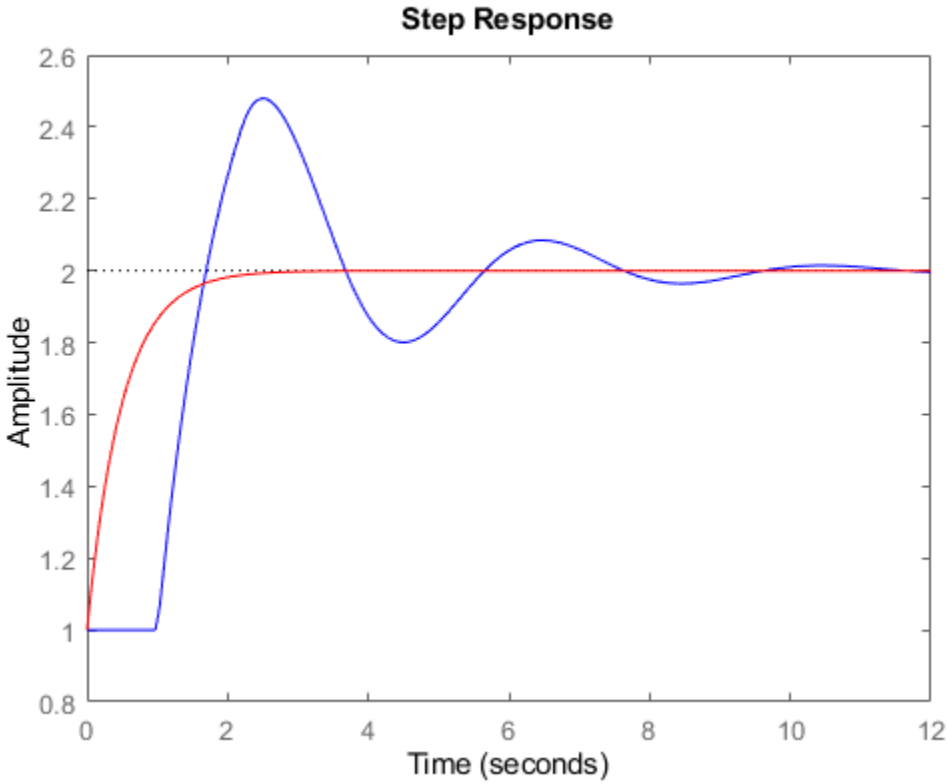
To create this model, specify `Aj,Bj,Cj,Dj` for each delay and use `delayss` to assemble the model:

```
DelayT(1) = struct('delay',0.5,'a',0,'b',2,'c',1,'d',0);   % tau1=0.5
DelayT(2) = struct('delay',1.2,'a',-1,'b',0,'c',0,'d',0);  % tau2=1.2
sys = delayss(-1,0,0,1,DelayT)
```

```
sys =

  A =
       x1
   x1  -2

  B =
       u1
   x1   2

  C =
       x1
   y1   1

  D =
       u1
   y1   1

  (values computed with all internal delays set to zero)

  Internal delays (seconds): 0.5  0.5  1.2

Continuous-time state-space model.
```

Note that the `A,B,C,D` values are for all delays set to zero. The response for these values need not be close to the actual response with delays:

```
step(sys,'b',pade(sys,0),'r')
```



Discrete-Time Models with Delays

Discrete-time delays are handled in a similar way with some minor differences:

- Discrete-time delays are always integer multiples of the sampling period
- Discrete-time delays are equivalent to poles at z=0, so it is always possible to absorb delays into the model dynamics. However, keeping delays separate is better for performance, especially for systems with long delays compared to the sampling period.

To specify the first-order model

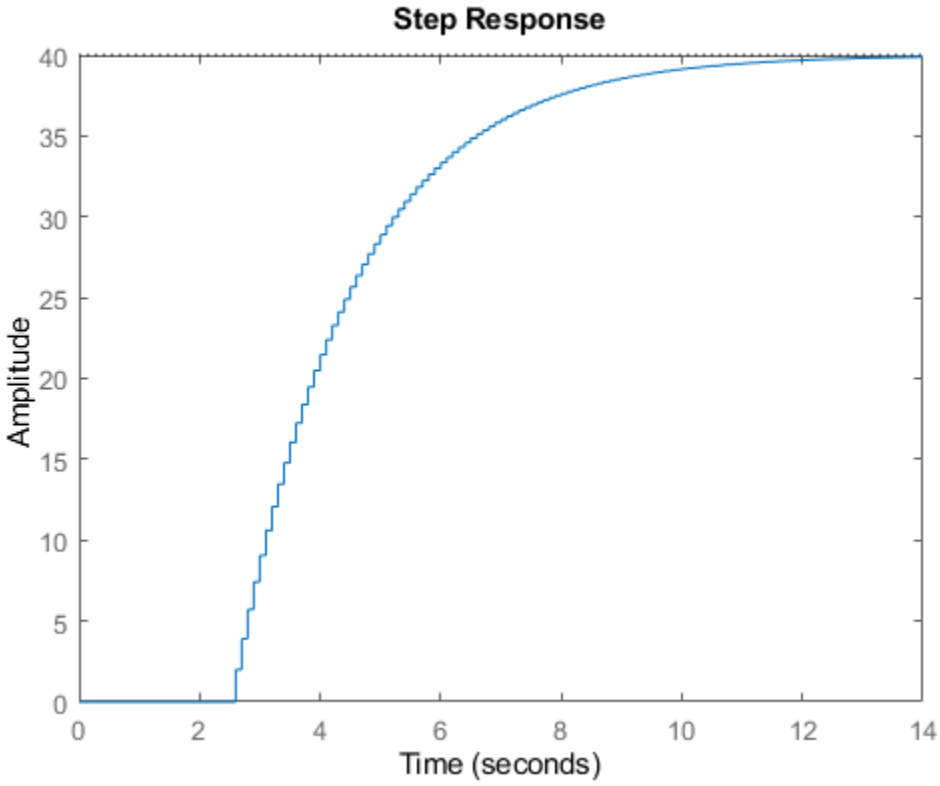
$$H(z) = z^{-25}\frac{2}{z-0.95}$$

with sampling period Ts=0.1, use

```
H = tf(2,[1 -0.95],0.1,'inputdelay',25)
step(H)
```

```
H =

              2
  z^(-25) * --------
            z - 0.95
```

```
Sample time: 0.1 seconds
Discrete-time transfer function.
```



The equivalent state-space representation is

```
H = ss(H)
```

```
H =

  A =
        x1
   x1  0.95

  B =
       u1
   x1   2

  C =
       x1
   y1   1

  D =
       u1
   y1   0

  Input delays (sampling periods): 25
```

```
Sample time: 0.1 seconds
Discrete-time state-space model.
```

Note that the delays are kept separate from the poles. Next, consider the feedback loop below where g is a pure gain.

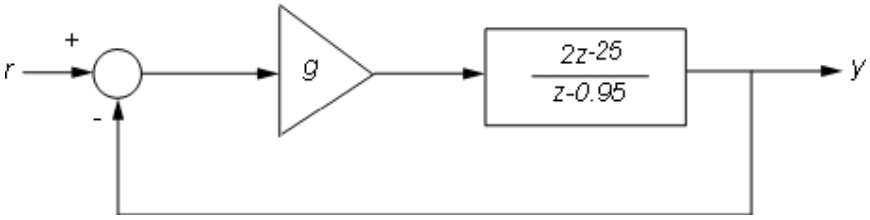
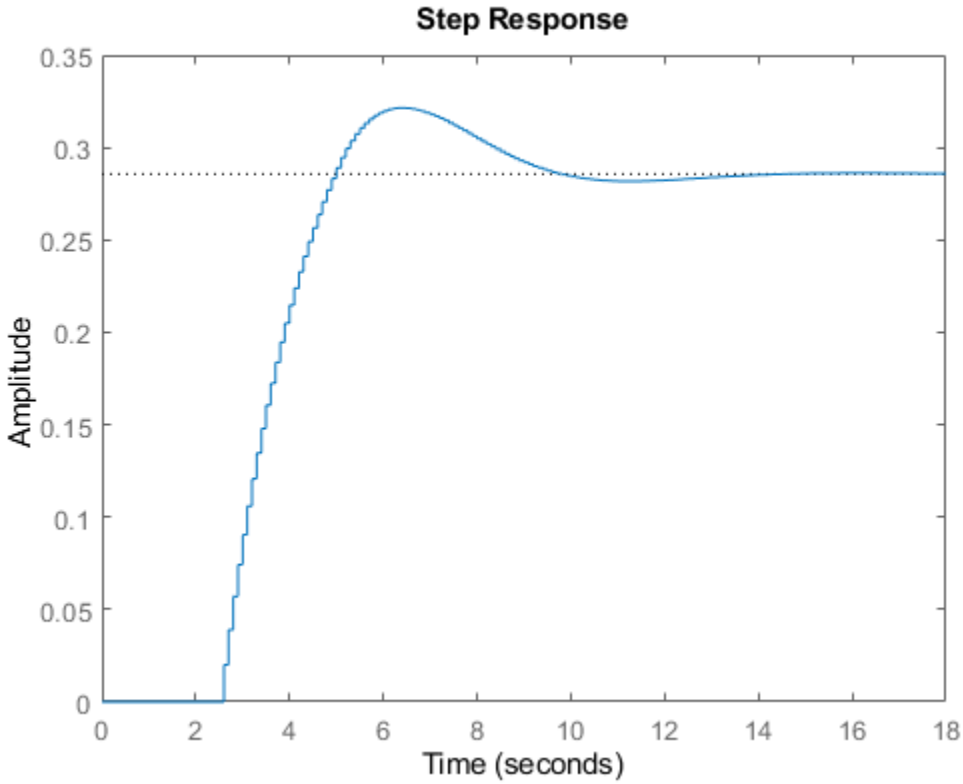


Figure 3: Discrete-Time Feedback Loop.

To compute the closed-loop response for `g=0.01`, type

```
g = .01;
T = feedback(g*H,1)
step(T)
```

```
T =

  A =
        x1
   x1  0.93

  B =
      u1
   x1   2

  C =
        x1
   y1  0.01

  D =
      u1
   y1   0

  (values computed with all internal delays set to zero)

  Internal delays (sampling periods): 25

Sample time: 0.1 seconds
Discrete-time state-space model.
```



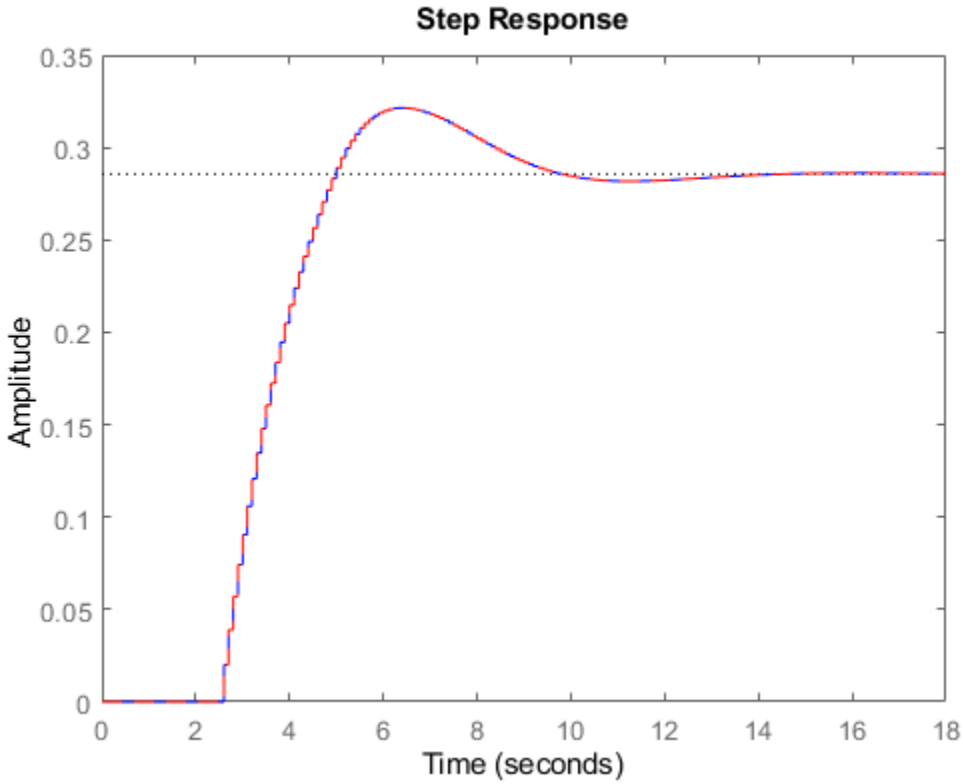
Note that T is still a first-order model with an internal delay of 25 samples. For comparison, map all delays to poles at z=0 using `absorbDelay`:

```
T1 = absorbDelay(T);
order(T1)
```

```
ans =

    26
```

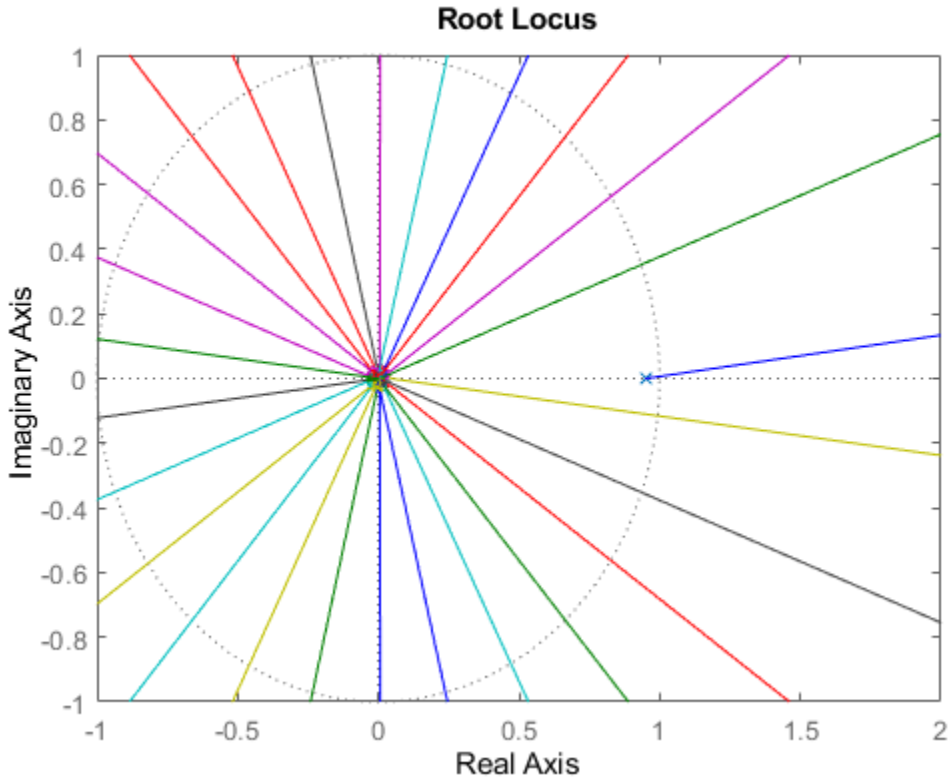
The resulting model has 26 states and is therefore less efficient to simulate. Note that the step responses of T and T1 exactly match as expected:

```
step(T,'b',T1,'r--')
```



In general, it is recommended to keep delays separate except when analyzing the closed-loop dynamics of models with internal delays:

```
rlocus(H)
axis([-1 2 -1 1])
```



Inside State-Space Models with Internal Delays

State-space objects use generalized state-space equations to keep track of internal delays. Conceptually, such models consist of two interconnected parts:

- An ordinary state-space model `H(s)` with augmented I/O set
- A bank of internal delays.

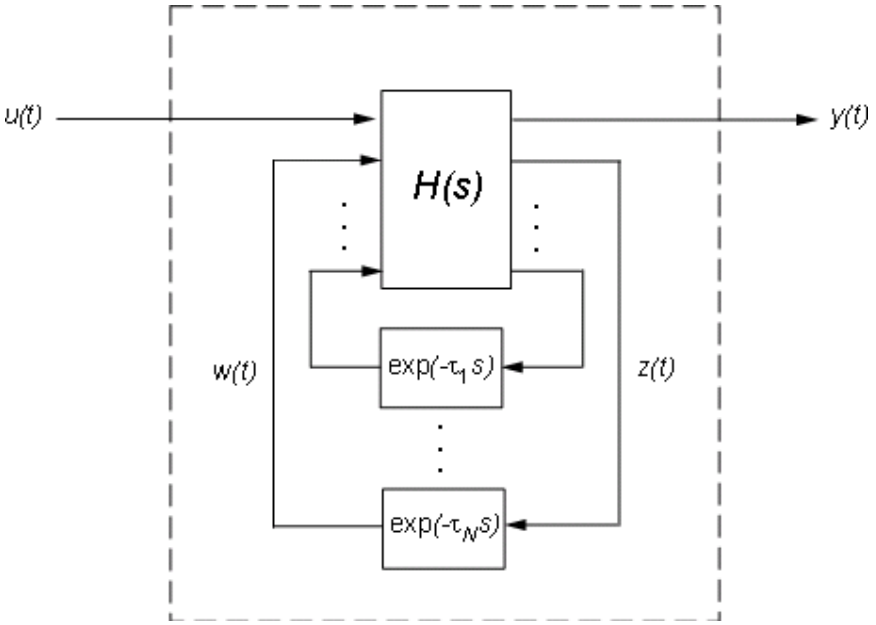


Figure 4: Internal Representation of State-Space Models with Internal Delays.

The corresponding state-space equations are

$$\begin{aligned}\dot{x}(t) &= Ax(t) + B_1u(t) + B_2w(t)\\ y(t) &= C_1x(t) + D_{11}u(t) + D_{12}w(t)\\ z(t) &= C_2x(t) + D_{21}u(t) + D_{22}w(t)\\ w_j(t) &= z_j(t-\tau_j),\ \ j = 1, ..., N\end{aligned}$$

You need not bother with this internal representation to use the tools. However, if for some reason you want to extract `H` or the matrices `A,B1,B2,...`, you can do this with `getDelayModel`. For the example

```
P = 5*exp(-3.4*s)/(s+1);
C = 0.1 * (1 + 1/(5*s));
T = feedback(P*C,1);

[H,tau] = getDelayModel(T,'lft');
size(H)
```

```
State-space model with 2 outputs, 2 inputs, and 2 states.
```

Note that `H` is a two-input, two-output model whereas `T` is SISO. The inverse operation (combining `H` and `tau` to construct `T`) is performed by `setDelayModel`.

Working with Linear Models

Data Manipulation

- “Store and Retrieve Model Data” on page 3-2
- “Extract Model Coefficients” on page 3-5
- “Attach Metadata to Models” on page 3-8
- “Query Model Characteristics” on page 3-12
- “Customize Model Display” on page 3-14
- “Accessing and Modifying the Model Data” on page 3-17

Store and Retrieve Model Data

Model Properties

Model properties are the data fields that store all data about a dynamic system model. Data stored in model properties includes model dynamics, such as transfer-function coefficients, state-space matrices, and time delays. Model properties also let you specify other model attributes such as sample time, channel names, and state names.

For information about the properties associated with each model type, see the corresponding reference page, such as `tf`, `pid`, or `ss`.

Specify Model Properties at Model Creation

When you create a dynamic system model, the software sets all property values. Properties that contain model dynamics are automatically set with the appropriate values. Other properties are set to default values. (See model reference pages for information about default property values.)

You can specify other values for model properties at model creation using the `Name,Value` pair syntax of the model-creation command. In this syntax, you specify the name of the property you want to set, followed by the value. You can set multiple property values in one command. For example, assign a transport delay and input and output names to a new transfer function model.

```
H = tf(1,[1 10],'IODelay',6.5,'InputName','torque','OutputName','velocity')
```

```
H =

  From input "torque" to output "velocity":
                  1
  exp(-6.5*s) * ------
                s + 10

Continuous-time transfer function.
```

Some property values are reflected in the model display, such as the input and output names. You can use `Name,Value` pair syntax when creating any type of model.

Examine and Change Properties of an Existing Model

Load an existing state-space (`ss`) model.

```
load('PadeApproximation1.mat','sys')
sys
```

```
sys =

  A =
         x1    x2
   x1  -1.5  -0.1
   x2     1     0
```

```
  B =
       u1
   x1   1
   x2   0

  C =
        x1   x2
   y1  0.5  0.1

  D =
       u1
   y1   0

  (values computed with all internal delays set to zero)

  Output delays (seconds): 1.5
  Internal delays (seconds): 3.4

Continuous-time state-space model.
```

The display shows that `sys` is a state-space model, and includes some of the property values of `sys`. To see all properties of `sys`, use the `get` command.

```
get(sys)
```

```
                A: [2x2 double]
                B: [2x1 double]
                C: [0.5000 0.1000]
                D: 0
                E: []
           Scaled: 0
        StateName: {2x1 cell}
        StatePath: {2x1 cell}
        StateUnit: {2x1 cell}
    InternalDelay: 3.4000
       InputDelay: 0
      OutputDelay: 1.5000
        InputName: {''}
        InputUnit: {''}
       InputGroup: [1x1 struct]
       OutputName: {''}
       OutputUnit: {''}
      OutputGroup: [1x1 struct]
            Notes: [0x1 string]
         UserData: []
             Name: ''
               Ts: 0
         TimeUnit: 'seconds'
     SamplingGrid: [1x1 struct]
```

Use dot notation to access the values of particular properties. For example, display the A matrix of `sys`.

```
Amat = sys.A
```

```
Amat = 2×2

   -1.5000   -0.1000
```

```
    1.0000         0
```

Dot notation also lets you change the value of individual model properties.

```
sys.InputDelay = 4.2;
sys.InputName = 'thrust';
sys.OutputName = 'velocity';
```

When you must change multiple property values at the same time to preserve the validity of the model, such as changing the dimensions of the state-space matrices, you can use the `set` command. For example, create a 1-state state-space model, and then replace the matrices with new values representing a 2-state model.

```
sys2 = rss(1);
Anew = [-2, 1; 0.5 0];
Bnew = [1; -1];
Cnew = [0, -0.4];
set(sys2,'A',Anew,'B',Bnew,'C',Cnew)
sys2
```

```
sys2 =

  A =
        x1   x2
   x1   -2    1
   x2  0.5    0

  B =
       u1
   x1   1
   x2  -1

  C =
         x1    x2
   y1     0  -0.4

  D =
          u1
   y1  0.3426

Continuous-time state-space model.
```

Changing certain properties, such as `Ts` or `TimeUnit`, can cause undesirable changes in system behavior. See the property descriptions in the model reference pages for more information.

See Also

Related Examples

- “Attach Metadata to Models” on page 3-8
- “Extract Model Coefficients” on page 3-5

Extract Model Coefficients

Functions for Extracting Model Coefficients

Control System Toolbox software includes several commands for extracting model coefficients such as transfer function numerator and denominator coefficients, state-space matrices, and proportional-integral-derivative (PID) gains.

The following commands are available for data extraction.

Command	Result
`tfdata`	Extract transfer function coefficients
`zpkdata`	Extract zero and pole locations and system gain
`ssdata`	Extract state-space matrices
`dssdata`	Extract descriptor state-space matrices
`frdata`	Extract frequency response data from `frd` model
`piddata`	Extract parallel-form PID data
`pidstddata`	Extract standard-form PID data
`get`	Access all model property values

Extracting Coefficients of Different Model Type

When you use a data extraction command on a model of a different type, the software computes the coefficients of the target model type. For example, if you use `zpkdata` on a `ss` model, the software converts the model to `zpk` form and returns the zero and pole locations and system gain.

Extract Numeric Model Data and Time Delay

This example shows how to extract transfer function numerator and denominator coefficients using `tfdata`.

1. Create a first-order plus dead time transfer function model.

   ```
   s = tf('s');
   H = exp(-2.5*s)/(s+12);
   ```

2. Extract the numerator and denominator coefficients.

   ```
   [num,den] = tfdata(H,'v')
   ```

   The variables `num` and `den` are numerical arrays. Without the `'v'` flag, `tfdata` returns cell arrays.

Note For SISO transfer function models, you can also extract coefficients using:

```
num = H.Numerator{1};
den = H.Denominator{1};
```

---

3. Extract the time delay.

   a. Determine which property of `H` contains the time delay.

      In a SISO `tf` model, you can express a time delay as an input delay, an output delay, or a transport delay (I/O delay).

      ```
      get(H)
      ```

      ```
             Numerator: {[0 1]}
           Denominator: {[1 12]}
              Variable: 's'
               IODelay: 0
            InputDelay: 0
           OutputDelay: 2.5000
                    Ts: 0
              TimeUnit: 'seconds'
             InputName: {''}
             InputUnit: {''}
            InputGroup: [1×1 struct]
            OutputName: {''}
            OutputUnit: {''}
           OutputGroup: [1×1 struct]
                 Notes: [0×1 string]
              UserData: []
                  Name: ''
        SamplingGrid: [1×1 struct]
      ```

      The time delay is stored in the `OutputDelay` property.

   b. Extract the output delay.

      ```
      delay = H.OutputDelay;
      ```

Extract PID Gains from Transfer Function

This example shows how to extract PID (proportional-integral-derivative) gains from a transfer function using `piddata`. You can use the same steps to extract PID gains from a model of any type that represents a PID controller, using `piddata` or `pidstddata`.

1. Create a transfer function that represents a PID controller with a first-order filter on the derivative term.

   ```
   Czpk = zpk([-6.6,-0.7],[0,-2],0.2)
   ```

2. Obtain the PID gains and filter constant.

   ```
   [Kp,Ki,Kd,Tf] = piddata(Czpk)
   ```

   This command returns the proportional gain `Kp`, integral gain `Ki`, derivative gain `Kd`, and derivative filter time constant `Tf`. Because `piddata` automatically computes the PID controller parameters, you can extract the PID coefficients without creating a `pid` model.

See Also

Related Examples

- “Attach Metadata to Models” on page 3-8

More About

- “Store and Retrieve Model Data” on page 3-2

Attach Metadata to Models

Specify Model Time Units

This example shows how to specify time units of a transfer function model.

The `TimeUnit` property of the `tf` model object specifies units of the time variable, time delays (for continuous-time models), and the sample time Ts (for discrete-time models). The default time units is `seconds`.

Create a SISO transfer function model $sys = \frac{4s+2}{s^2+3s+10}$ with time units in milliseconds:

```
num = [4 2];
den = [1 3 10];
sys = tf(num,den,'TimeUnit','milliseconds');
```

You can specify the time units of any dynamic system on page 1-7 in a similar way.

The system time units appear on the time- and frequency-domain plots. For multiple systems with different time units, the units of the first system are used if the time and frequency units in the "Toolbox Preferences Editor" on page 20-2 are `auto`.

Note Changing the `TimeUnit` property changes the system behavior. If you want to use different time units without modifying system behavior, use `chgTimeUnit`.

Interconnect Models with Different Time Units

This example shows how to interconnect transfer function models with different time units.

To interconnect models using arithmetic operations or interconnection commands, the time units of all models must match.

1. Create two transfer function models with time units of milliseconds and seconds, respectively.

   ```
   sys1 = tf([1 2],[1 2 3],'TimeUnit','milliseconds');
   sys2 = tf([4 2],[1 3 10]);
   ```

2. Change the time units of `sys2` to milliseconds.

   ```
   sys2 = chgTimeUnit(sys2,'milliseconds');
   ```

3. Connect the systems in parallel.

   ```
   sys = sys1+sys2;
   ```

Specify Frequency Units of Frequency-Response Data Model

This example shows how to specify units of the frequency points of a frequency-response data model.

The `FrequencyUnit` property specifies units of the frequency vector in the `Frequency` property of the `frd` model object. The default frequency units are `rad/TimeUnit`, where `TimeUnit` is the time unit specified in the `TimeUnit` property.

Create a random SISO frequency-response data model with frequency data in GHz.

```
resp = randn(7,1) + i*randn(7,1);
freq = logspace(-2,2,7);
sys = frd(resp,freq,'FrequencyUnit','GHz');
```

You can independently specify the units in which you measure the frequency points and sample time in the `FrequencyUnit` and `TimeUnit` properties, respectively. You can also specify the frequency units of a `genfrd` in a similar way.

The frequency units appear on the frequency-domain plots. For multiple systems with different frequency units, the units of the first system are used if the frequency units in the “Toolbox Preferences Editor” on page 20-2 is `auto`.

Note Changing the `FrequencyUnit` property changes the system behavior. If you want to use different frequency units without modifying system behavior, use `chgFreqUnit`.

Extract Subsystems of Multi-Input, Multi-Output (MIMO) Models

This example shows how to extract subsystems of a MIMO model using MATLAB indexing and using channel names.

Extracting subsystems is useful when, for example, you want to analyze a portion of a complex system.

Create a MIMO transfer function.

```
G1 = tf(3,[1 10]);
G2 = tf([1 2],[1 0]);
G = [G1,G2];
```

Extract the subsystem of `G` from the first input to all outputs.

```
Gsub = G(:,1);
```

This command uses MATLAB indexing to specify a subsystem as `G(out,in)`, where `out` specifies the output indices and `in` specifies the input indices.

Using channel names, you can use MATLAB indexing to extract all the dynamics relating to a particular channel. By using this approach, you can avoid having to keep track of channel order in a complex MIMO model.

Assign names to the model inputs.

```
G.InputName = {'temperature';'pressure'};
```

Because `G` has two inputs, use a cell array to specify the two channel names.

Extract the subsystem of `G` that contains all dynamics from the `'temperature'` input to all outputs.

```
Gt = G(:,'temperature');
```

`Gt` is the same subsystem as `Gsub`.

Note When you extract a subsystem from a state-space (`ss`) model, the resulting state-space model may not be minimal. Use `sminreal` to eliminate unnecessary states in the subsystem.

Specify and Select Input and Output Groups

This example shows how to specify groups of input and output channels in a model object and extract subsystems using the groups.

Input and output groups are useful for keeping track of inputs and outputs in complex MIMO models.

1. Create a state-space model with three inputs and four outputs.

   ```
   H = rss(3,4,3);
   ```

2. Group the inputs as follows:

   - Inputs 1 and 2 in a group named `controls`
   - Outputs 1 and 3 to a group named `temperature`
   - Outputs 1, 3, and 4 to a group named `measurements`

   ```
   H.InputGroup.controls = [1 2];
   H.OutputGroup.temperature = [1 3];
   H.OutputGroup.measurements = [1 3 4];
   ```

   `InputGroup` and `OutputGroup` are structures. The name of each field in the structure is the name of the input or output group. The value of each field is a vector that identifies the channels in that group.

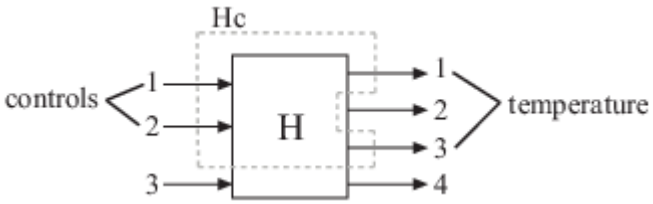
3. Extract the subsystem corresponding to the `controls` inputs and the `temperature` outputs.

   You can use group names to index into subsystems.

   ```
   Hc = H('temperature','controls')
   ```

   `Hc` is a two-input, two-output `ss` model containing the I/O channels from the `'controls'` input to the `'temperature'` outputs.

   You can see the relationship between `H` and the subsystem `Hc` in this illustration.



See Also

Related Examples

- “Store and Retrieve Model Data” on page 3-2
- “Extract Model Coefficients” on page 3-5

- “Query Model Characteristics” on page 3-12

Query Model Characteristics

This example shows how to query model characteristics such as stability, time domain, and number of inputs and outputs. You can use the techniques of this example on any type of dynamic system model.

Load a saved state-space (`ss`) model.

```
load('queryexample.mat','T')
```

Query whether `T` has stable dynamics.

```
Bstab = isstable(T)
```

```
Bstab = logical
   1
```

The `isstable` command returns 1 (`true`) if all system poles are in the open left-half plane (for continuous-time models) or inside the open unit disk (for discrete-time models). Otherwise, `isstable` command returns 0 (`false`). Here, the result shows that the model is stable.

Query whether `T` has time delays.

```
Bdel = hasdelay(T)
```

```
Bdel = logical
   1
```

The returned value, 1, indicates that `T` has a time delay. For a state-space model, time delay can be stored as input delay, output delay, internal delay, or a combination. Use `get(T)` to determine which properties of `T` hold the time delay, and use dot notation to access the delay values. The `hasInternalDelay` command tells you whether there is any internal delay.

Query whether `T` is proper.

```
Bprop = isproper(T)
```

```
Bprop = logical
   1
```

The returned value indicates that the system has relative degree less than or equal to 0. This is true of a SISO system when it can be represented as a transfer function in which the degree of the numerator does not exceed the degree of the denominator.

Query the order of `T`.

```
N = order(T)
```

```
N = 5
```

For a state-space model, `order` returns the number of states, which is 5 in this case. For a `tf` or `zpk` model, the order is the number of states required for a state-space realization of the system.

Query whether `T` is a discrete-time system.

```
Bdisc = isdt(T)

Bdisc = logical
   1
```

The returned value indicates that `T` is a discrete-time model. Similarly, use `isct` to query whether `T` is a continuous-time model.

Load a MIMO model and query the input/output dimensions.

```
load('queryexample.mat','Tmimo')
ios = iosize(Tmimo)

ios = 1×2

     7     4
```

In the resulting array, the number of outputs is first. Therefore, `Tmimo` has 4 inputs and 7 outputs.

See Also

`isstable` | `isproper` | `size`

Related Examples

- “Select Models from Array” on page 2-79

More About

- “Store and Retrieve Model Data” on page 3-2

Customize Model Display

Configure Transfer Function Display Variable

This example shows how to configure the MATLAB command-window display of transfer function (`tf`) models.

You can use the same steps to configure the display variable of transfer function models in factorized form (`zpk` models).

By default, `tf` and `zpk` models are displayed in terms of `s` in continuous time and `z` in discrete time. Use the `Variable` property change the display variable to `'p'` (equivalent to `'s'`), `'q'` (equivalent to `'z'`), `'z^-1'`, or `'q^-1'`.

1 Create the discrete-time transfer function $H(z) = \frac{z-1}{z^2-3z+2}$

with a sample time of 1 s.

```
H = tf([1 -1],[1 -3 2],0.1)
```

```
H =

      z - 1
  -------------
  z^2 - 3 z + 2

Sample time: 0.1 seconds
Discrete-time transfer function.
```

The default display variable is `z`.

2 Change the display variable to `q^-1`.

```
H.Variable = 'q^-1'
```

```
H =

      q^-1 - q^-2
  --------------------
  1 - 3 q^-1 + 2 q^-2

Sample time: 0.1 seconds
Discrete-time transfer function.
```

When you change the `Variable` property, the software computes new coefficients and displays the transfer function in terms of the new variable. The `num` and `den` properties are automatically updated with the new coefficients.

Tip Alternatively, you can directly create the same transfer function expressed in terms of `'q^-1'`.

```
H = tf([0 1 -1],[1 -3 2],0.1,'Variable','q^-1');
```

For the inverse variables `'z^-1'` and `'q^-1'`, `tf` interprets the numerator and denominator arrays as coefficients of ascending powers of `'z^-1'` or `'q^-1'`.

Configure Display Format of Transfer Function in Factorized Form

This example shows how to configure the display of transfer function models in factorized form (`zpk` models).

You can configure the display of the factorized numerator and denominator polynomials to highlight:

- The numerator and denominator roots
- The natural frequencies and damping ratios of each root
- The time constants and damping ratios of each root

See the `DisplayFormat` property on the `zpk` reference page for more information about these quantities.

1 Create a `zpk` model having a zero at $s = 5$, a pole at $s = -10$, and a pair of complex poles at $s = -3 \pm 5i$.

```
H = zpk(5,[-10,-3-5*i,-3+5*i],10)
```

```
H =

        10 (s-5)
  ----------------------
  (s+10) (s^2 + 6s + 34)

Continuous-time zero/pole/gain model.
```

The default display format, `'roots'`, displays the standard factorization of the numerator and denominator polynomials.

2 Configure the display format to display the natural frequency of each polynomial root.

```
 H.DisplayFormat = 'frequency'
```

```
H =

               -0.14706 (1-s/5)
  ------------------------------------------
  (1+s/10) (1 + 1.029(s/5.831) + (s/5.831)^2)

Continuous-time zero/pole/gain model.
```

You can read the natural frequencies and damping ratios for each pole and zero from the display as follows:

- Factors corresponding to real roots are displayed as $(1 - s/\omega_0)$. The variable ω_0 is the natural frequency of the root. For example, the natural frequency of the zero of `H` is 5.
- Factors corresponding to complex pairs of roots are displayed as $1 - 2\zeta(s/\omega_0) + (s/\omega_0)^2$. The variable ω_0 is the natural frequency, and ζ is the damping ratio of the root. For example, the natural frequency of the complex pole pair is 5.831, and the damping ratio is 1.029/2.

3 Configure the display format to display the time constant of each pole and zero.

```
H.DisplayFormat = 'time constant'
```

```
H =
```

```
              -0.14706 (1-0.2s)
  ------------------------------------------
  (1+0.1s) (1 + 1.029(0.1715s) + (0.1715s)^2)

Continuous-time zero/pole/gain model.
```

You can read the time constants and damping ratios from the display as follows:

- Factors corresponding to real roots are displayed as (τs). The variable τ is the time constant of the root. For example, the time constant of the zero of H is 0.2.
- Factors corresponding to complex pairs of roots are displayed as $1 - 2\zeta(\tau s) + (\tau s)^2$. The variable τ is the time constant, and ζ is the damping ratio of the root. For example, the time constant of the complex pole pair is 0.1715, and the damping ratio is 1.029/2.

See Also

`tf` | `zpk`

Related Examples

- “Transfer Functions” on page 2-2

Accessing and Modifying the Model Data

This example shows how to access or edit parameter values and metadata in LTI objects.

Accessing Data

The `tf`, `zpk`, `ss`, and `frd` commands create LTI objects that store model data in a single MATLAB® variable. This data includes model-specific parameters (e.g., A,B,C,D matrices for state-space models) as well as generic metadata such as input and output names. The data is arranged into a fixed set of data fields called **properties**.

You can access model data in the following ways:

- The `get` command
- Structure-like dot notation
- Data retrieval commands

For illustration purposes, create the SISO transfer function (TF):

```
G = tf([1 2],[1 3 10],'inputdelay',3)
```

```
G =

                 s + 2
  exp(-3*s) * --------------
              s^2 + 3 s + 10

Continuous-time transfer function.
```

To see all properties of the TF object `G`, type

```
get(G)
```

```
       Numerator: {[0 1 2]}
     Denominator: {[1 3 10]}
        Variable: 's'
         IODelay: 0
      InputDelay: 3
     OutputDelay: 0
       InputName: {''}
       InputUnit: {''}
      InputGroup: [1x1 struct]
      OutputName: {''}
      OutputUnit: {''}
     OutputGroup: [1x1 struct]
           Notes: [0x1 string]
        UserData: []
            Name: ''
              Ts: 0
        TimeUnit: 'seconds'
    SamplingGrid: [1x1 struct]
```

The first four properties `Numerator`, `Denominator`, `IODelay`, and `Variable` are specific to the TF representation. The remaining properties are common to all LTI representations. You can use `help tf.Numerator` to get more information on the `Numerator` property and similarly for the other properties.

To retrieve the value of a particular property, use

```
G.InputDelay    % get input delay value

ans = 3
```

You can use abbreviations for property names as long as they are unambiguous, for example:

```
G.iod    % get transport delay value

ans = 0

G.var    % get variable

ans =
's'
```

Quick Data Retrieval

You can also retrieve all model parameters at once using `tfdata`, `zpkdata`, `ssdata`, or `frdata`. For example:

```
[Numerator,Denominator,Ts] = tfdata(G)

Numerator = 1x1 cell array
    {[0 1 2]}


Denominator = 1x1 cell array
    {[1 3 10]}


Ts = 0
```

Note that the numerator and denominator are returned as cell arrays. This is consistent with the MIMO case where `Numerator` and `Denominator` contain cell arrays of numerator and denominator polynomials (with one entry per I/O pair). For SISO transfer functions, you can return the numerator and denominator data as vectors by using a flag, for example:

```
[Numerator,Denominator] = tfdata(G,'v')

Numerator = 1×3

     0     1     2


Denominator = 1×3

     1     3    10
```

Editing Data

You can modify the data stored in LTI objects by editing the corresponding property values with `set` or dot notation. For example, for the transfer function `G` created above,

```
G.Ts = 1;
```

changes the sample time from 0 to 1, which redefines the model as discrete:

```
G
```

```
G =

               z + 2
  z^(-3) * --------------
           z^2 + 3 z + 10

Sample time: 1 seconds
Discrete-time transfer function.
```

The `set` command is equivalent to dot assignment, but also lets you set multiple properties at once:

```
G.Ts = 0.1;
G.Variable = 'q';
G
```

```
G =

               q + 2
  q^(-3) * --------------
           q^2 + 3 q + 10

Sample time: 0.1 seconds
Discrete-time transfer function.
```

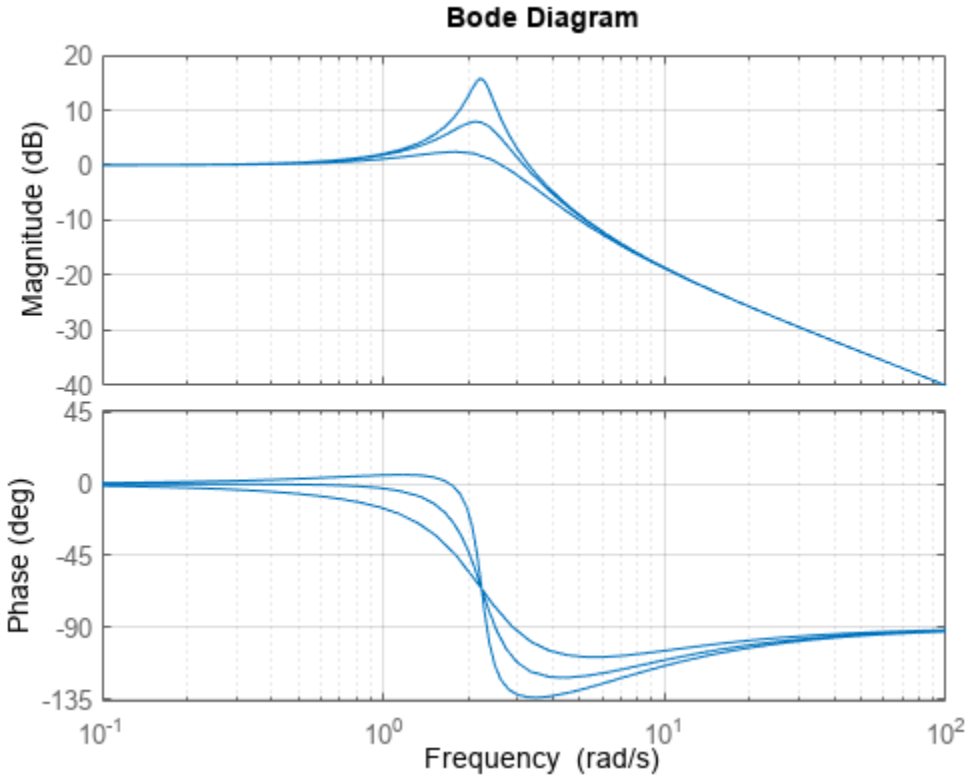
Sensitivity Analysis Example

Using model editing together with LTI array support, you can easily investigate sensitivity to parameter variations. For example, consider the second-order transfer function

$$H(s) = \frac{s+5}{s^2 + 2\zeta s + 5}$$

You can investigate the effect of the damping parameter `zeta` on the frequency response by creating three models with different `zeta` values and comparing their Bode responses:

```
s = tf('s');

% Create 3 transfer functions with Numerator = s+5 and Denominator = 1
H = repsys(s+5,[1 1 3]);

% Specify denominators using 3 different zeta values
zeta = [1 .5 .2];
for k = 1:3
  H(:,:,k).Denominator = [1 2*zeta(k) 5];  % zeta(k) -> k-th model
end

% Plot Bode response
bode(H)
grid
```


Bode Diagram
Magnitude (dB)
20
10
0
-10
-20
-30
-40
Phase (deg)
45
0
-45
-90
-135
10^-1
10^0
10^1
10^2
Frequency (rad/s)

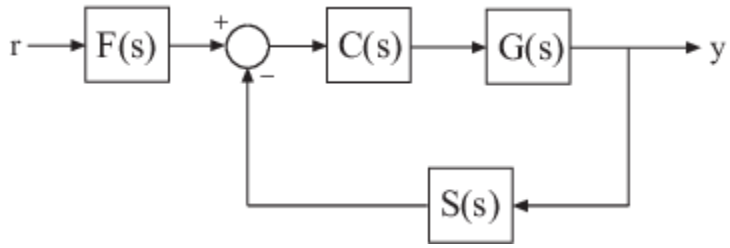
Model Interconnections

- “Why Interconnect Models?” on page 4-2
- “Catalog of Model Interconnections” on page 4-3
- “Numeric Model of SISO Feedback Loop” on page 4-5
- “Control System Model with Both Numeric and Tunable Components” on page 4-7
- “Multi-Loop Control System” on page 4-9
- “Mark Analysis Points in Closed-Loop Models” on page 4-11
- “MIMO Control System” on page 4-15
- “MIMO Feedback Loop” on page 4-17
- “How the Software Determines Properties of Connected Models” on page 4-20
- “Rules That Determine Model Type” on page 4-21
- “Recommended Model Type for Building Block Diagrams” on page 4-22
- “Using FEEDBACK to Close Feedback Loops” on page 4-24
- “Preventing State Duplication in System Interconnections” on page 4-28

Why Interconnect Models?

Interconnecting models of components allows you to construct models of control systems. You can conceptualize your control system as a block diagram containing multiple interconnected components, such as a plant or a controller. Using model arithmetic or interconnection commands, you combine models of each of these components into a single model representing the entire block diagram.

For example, you can interconnect dynamic system models of a plant $G(s)$, a controller $C(s)$, sensor dynamics $S(s)$, and a filter $F(s)$ to construct a single model that represents the entire closed-loop control system in the following illustration:



See Also

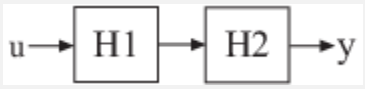
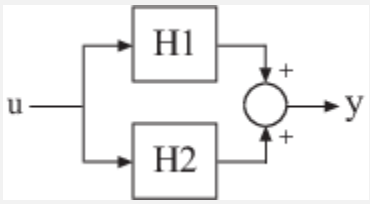
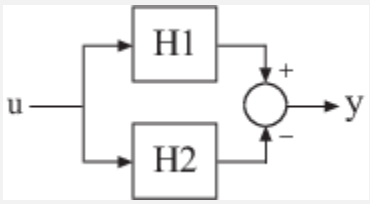
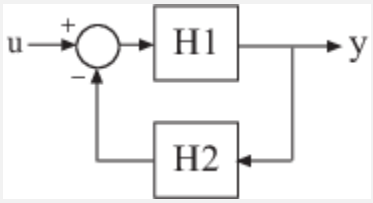
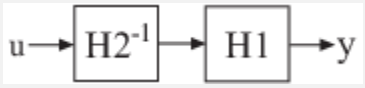
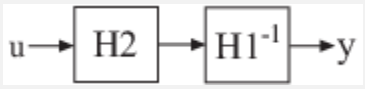
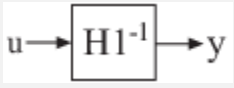
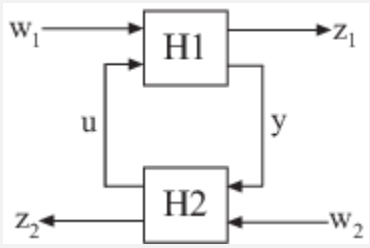
More About

- “Catalog of Model Interconnections” on page 4-3

Catalog of Model Interconnections

Each type of block diagram connection corresponds to a model interconnection command or arithmetic expression. The following tables summarize the block diagram connections with the corresponding interconnection command and arithmetic expression.

Model Interconnection Commands

Block Diagram Connection	Command	Arithmetic Expression
	`series(H1,H2)`	`H2*H1`
	`parallel(H1,H2)`	`H1+H2`
	`parallel(H1,-H2)`	`H1-H2`
	`feedback(H1,H2)`	`H1/(1+H2*H1)` (not recommended)
	N/A	`H1/H2` (division)
	N/A	`H1\H2` (left division)
	`inv(H1)`	N/A
	`lft(H1,H2,nu,ny)`	N/A

Arithmetic Operations

You can apply almost all arithmetic operations to dynamic system models, including those shown below.

Operation	Description
`+`	Addition
`-`	Subtraction
`*`	Multiplication
`.*`	Element-by-element multiplication
`/`	Right matrix divide
`\`	Left matrix divide
`inv`	Matrix inversion
`'`	Conjugate transposition. See `ctranspose`.
`.'`	Transposition

`^`	Powers of a dynamic system model, as in the following syntax for creating transfer functions:
	
	`s = tf('s');`
	`G = 25/(s^2 + 10*s + 25);`

In some cases, you might obtain better results using model interconnection commands, such as `feedback` or `connect`, instead of model arithmetic. For example, the command `T = feedback(H1,H2)` returns better results than the algebraic expression `T = H1/(1+H2*H1)`. The latter expression duplicates the poles of `H1`, which inflates the model order and might lead to computational inaccuracy.

See Also

`connect` | `feedback` | `series` | `parallel`

Related Examples

- “Numeric Model of SISO Feedback Loop” on page 4-5
- “Multi-Loop Control System” on page 4-9
- “MIMO Control System” on page 4-15

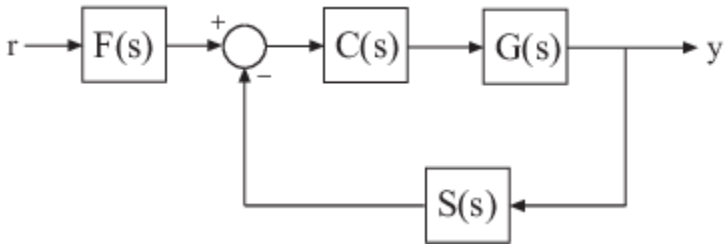
More About

- “How the Software Determines Properties of Connected Models” on page 4-20
- “Recommended Model Type for Building Block Diagrams” on page 4-22

Numeric Model of SISO Feedback Loop

This example shows how to interconnect numeric LTI models on page 1-10 representing multiple system components to build a single numeric model of a closed-loop system, using model arithmetic and interconnection commands.

Construct a model of the following single-loop control system.



The feedback loop includes a plant $G(s)$, a controller $C(s)$, and a representation of sensor dynamics, $S(s)$. The system also includes a prefilter $F(s)$.

1 Create model objects representing each of the components.

```
G = zpk([],[-1,-1],1);
C = pid(2,1.3,0.3,0.5);
S = tf(5,[1 4]);
F = tf(1,[1 1]);
```

The plant G is a zero-pole-gain (`zpk`) model with a double pole at $s = -1$. Model object C is a PID controller. The models F and S are transfer functions.

2 Connect the controller and plant models.

```
H = G*C;
```

To combine models using the multiplication operator *, enter the models in reverse order compared to the block diagram.

Tip Alternatively, construct $H(s)$ using the `series` command:

```
H = series(C,G);
```

---

3 Construct the unfiltered closed-loop response $T(s) = \frac{H}{1 + HS}$.

```
T = feedback(H,S);
```

Caution Do not use model arithmetic to construct `T` algebraically:

```
T = H/(1+H*S)
```

This computation duplicates the poles of `H`, which inflates the model order and might lead to computational inaccuracy.

---

4 Construct the entire closed-loop system response from r to y.

```
T_ry = T*F;
```

`T_ry` is a Numeric LTI Model representing the aggregate closed-loop system. `T_ry` does not keep track of the coefficients of the components `G`, `C`, `F`, and `S`.

You can operate on `T_ry` with any Control System Toolbox control design or analysis commands.

See Also

`connect` | `feedback` | `series` | `parallel`

Related Examples

- “Control System Model with Both Numeric and Tunable Components” on page 4-7
- “Multi-Loop Control System” on page 4-9
- “MIMO Control System” on page 4-15

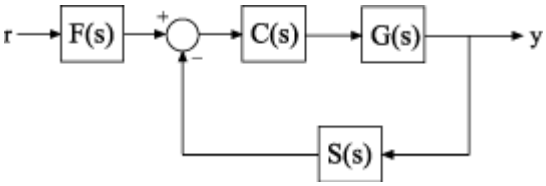
More About

- “Catalog of Model Interconnections” on page 4-3

Control System Model with Both Numeric and Tunable Components

This example shows how to create a tunable model of a control system that has both fixed plant and sensor dynamics and tunable control components.

Consider the control system of the following illustration.



Suppose that the plant response is $G(s) = 1/(s + 1)^2$, and that the model of the sensor dynamics is $S(s) = 5/(s + 4)$. The controller C is a tunable PID controller, and the prefilter $F = a/(s + a)$ is a low-pass filter with one tunable parameter, a.

Create models representing the plant and sensor dynamics. Because the plant and sensor dynamics are fixed, represent them using numeric LTI models.

```
G = zpk([],[-1,-1],1);
S = tf(5,[1 4]);
```

To model the tunable components, use Control Design Blocks. Create a tunable representation of the controller C.

```
C = tunablePID('C','PID');
```

`C` is a `tunablePID` object, which is a Control Design Block with a predefined proportional-integral-derivative (PID) structure.

Create a model of the filter $F = a/(s + a)$ with one tunable parameter.

```
a = realp('a',10);
F = tf(a,[1 a]);
```

`a` is a `realp` (real tunable parameter) object with initial value 10. Using `a` as a coefficient in `tf` creates the tunable `genss` model object `F`.

Interconnect the models to construct a model of the complete closed-loop response from r to y.

```
T = feedback(G*C,S)*F
```

```
Generalized continuous-time state-space model with 1 outputs, 1 inputs, 5 states, and the follow
  C: Tunable PID controller, 1 occurrences.
  a: Scalar parameter, 2 occurrences.

Type "ss(T)" to see the current value and "T.Blocks" to interact with the blocks.
```

`T` is a `genss` model object. In contrast to an aggregate model formed by connecting only numeric LTI models, `T` keeps track of the tunable elements of the control system. The tunable elements are stored in the `Blocks` property of the `genss` model object. Examine the tunable elements of `T`.

```
T.Blocks

ans = struct with fields:
    C: [1x1 tunablePID]
    a: [1x1 realp]
```

When you create a `genss` model of a control system that has tunable components, you can use tuning commands such as `systune` to tune the free parameters to meet design requirements you specify.

See Also

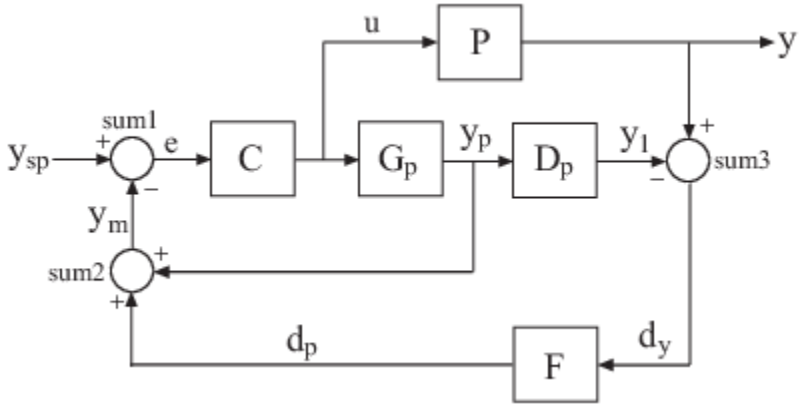
`feedback` | `tunablePID`

More About

- “Control Design Blocks” on page 1-12
- “Dynamic System Models” on page 1-7

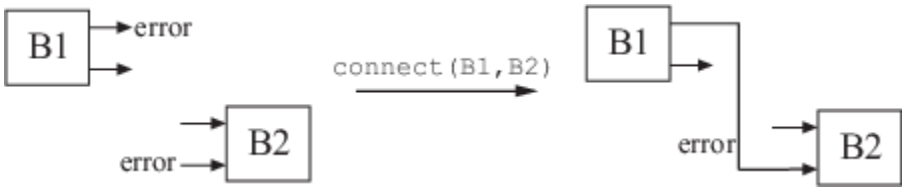
Multi-Loop Control System

This example shows how to build an arbitrary block diagram by connecting models using `connect`. The system is a Smith Predictor, the single-input, single-output (SISO) multi-loop control system shown in the following block diagram.



For more information about the Smith Predictor, see “Control of Processes with Long Dead Time: The Smith Predictor” on page 11-115.

The `connect` command lets you construct the overall transfer function from y_{sp} to y. To use `connect`, specify the input and output channel names of the components of the block diagram. `connect` automatically joins ports that have the same name, as shown in the following figure.



To build the closed loop model of the Smith Predictor system from y_{sp} to y:

1 Create the components of the block diagram: the process model `P`, the predictor model `Gp`, the delay model `Dp`, the filter `F`, and the PI controller `C`. Specify names for the input and output channels of each model so that `connect` can automatically join them to build the block diagram.

```
s = tf('s');

P = exp(-93.9*s) * 5.6/(40.2*s+1);
P.InputName = 'u'; P.OutputName = 'y';

Gp = 5.6/(40.2*s+1);
Gp.InputName = 'u'; Gp.OutputName = 'yp';

Dp = exp(-93.9*s);
Dp.InputName = 'yp'; Dp.OutputName = 'y1';

F = 1/(20*s+1);
F.InputName = 'dy'; F.OutputName = 'dp';
```

```
C = pidstd(0.574,40.1);
C.Inputname = 'e'; C.OutputName = 'u';
```

2 Create the summing junctions needed to complete the block diagram.

```
sum1 = sumblk('e = ysp - ym');
sum2 = sumblk('ym = yp + dp');
sum3 = sumblk('dy = y - y1');
```

The argument to `sumblk` is a formula that relates the input and output signals of the summing junction. `sumblk` creates a summing junction with the input and output signal names specified in the formula. For example, in `sum1`, the formula `'e = ysp - ym'` specifies an output signal named `e`, which is the difference between input signals named `ysp` and `ym`.

3 Assemble the complete model from y_{sp} to y.

```
T = connect(P,Gp,Dp,C,F,sum1,sum2,sum3,'ysp','y');
```

You can list the models and summing junctions in any order because `connect` automatically interconnects them using their input and output channel names.

The last two arguments specify the input and output signals of the multi-loop control structure. Thus, `T` is a `ss` model with input `ysp` and output `y`.

See Also

`connect` | `sumblk`

Related Examples

- “Control System Model with Both Numeric and Tunable Components” on page 4-7
- “MIMO Control System” on page 4-15
- “Mark Analysis Points in Closed-Loop Models” on page 4-11

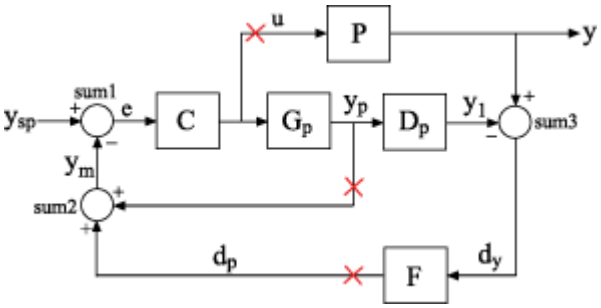
More About

- “How the Software Determines Properties of Connected Models” on page 4-20

Mark Analysis Points in Closed-Loop Models

This example shows how to build a block diagram and insert analysis points at locations of interest using the `connect` command. You can then use the analysis points to extract various system responses from the model.

For this example, create a model of a Smith predictor, the SISO multiloop control system shown in the following block diagram.



Points marked by x are analysis points to mark for this example. For instance, if you want to calculate the step response of the closed-loop system to a disturbance at the plant input, you can use an analysis point at *u*. If you want to calculate the response of the system with one or both of the control loops open, you can use the analysis points at *yp* or *dp*.

To build this system, first create the dynamic components of the block diagram. Specify names for the input and output channels of each model so that `connect` can automatically join them to build the block diagram.

```
s = tf('s');

% Process model
P = exp(-93.9*s) * 5.6/(40.2*s+1);
P.InputName = 'u';
P.OutputName = 'y';

% Predictor model
Gp = 5.6/(40.2*s+1);
Gp.InputName = 'u';
Gp.OutputName = 'yp';

% Delay model
Dp = exp(-93.9*s);
Dp.InputName = 'yp';
Dp.OutputName = 'y1';

% Filter
F = 1/(20*s+1);
F.InputName = 'dy';
F.OutputName = 'dp';

% PI controller
C = pidstd(0.574,40.1);
C.Inputname = 'e';
C.OutputName = 'u';
```

Create the summing junctions needed to complete the block diagram. (For more information about creating summing junctions, see `sumblk`).

```
sum1 = sumblk('e = ysp - ym');
sum2 = sumblk('ym = yp + dp');
sum3 = sumblk('dy = y - y1');
```

Assemble the complete model.

```
input = 'ysp';
output = 'y';
APs = {'u','dp','yp'};
T = connect(P,Gp,Dp,C,F,sum1,sum2,sum3,input,output,APs)
```

```
Generalized continuous-time state-space model with 1 outputs, 1 inputs, 4 states, and the follow
  AnalysisPoints_: Analysis point, 3 channels, 1 occurrences.

Type "ss(T)" to see the current value and "T.Blocks" to interact with the blocks.
```

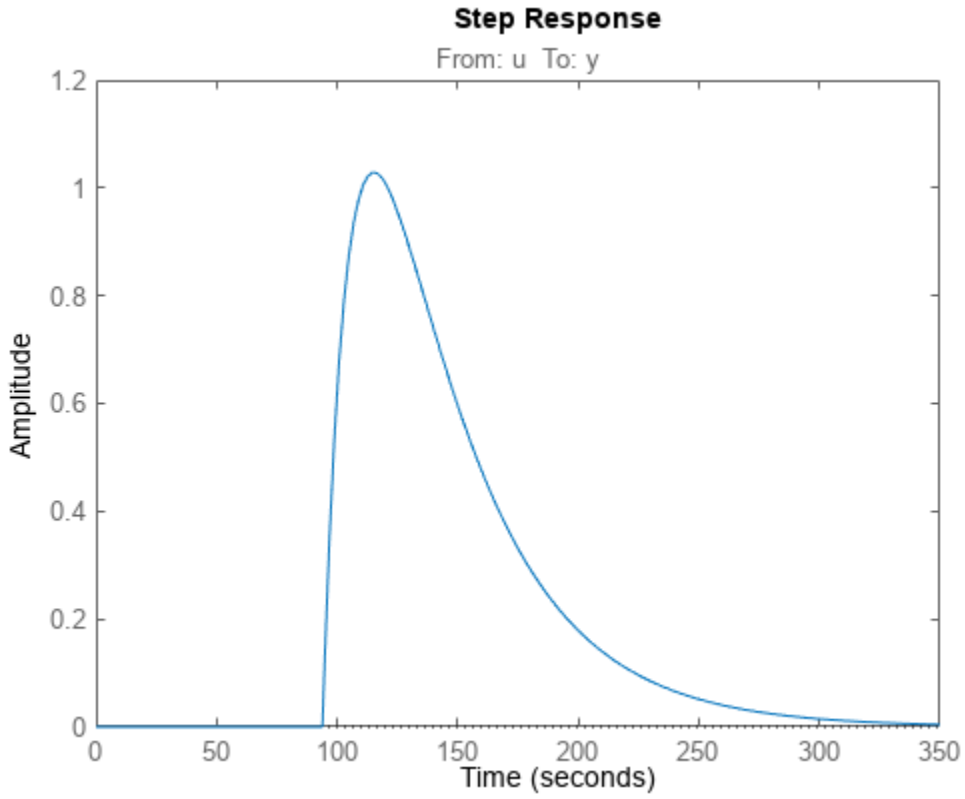
When you use the `APs` input argument, the `connect` command automatically inserts an `AnalysisPoint` block into the generalized state-space (`genss`) model, `T`. The automatically generated block is named `AnalysisPoints_`. The three channels of `AnalysisPoints_` correspond to the three locations specified in the `APs` argument to the `connect` command. Use `getPoints` to see a list of the available analysis points in the model.

```
getPoints(T)
```

```
ans = 3x1 cell
    {'dp'}
    {'u' }
    {'yp'}
```

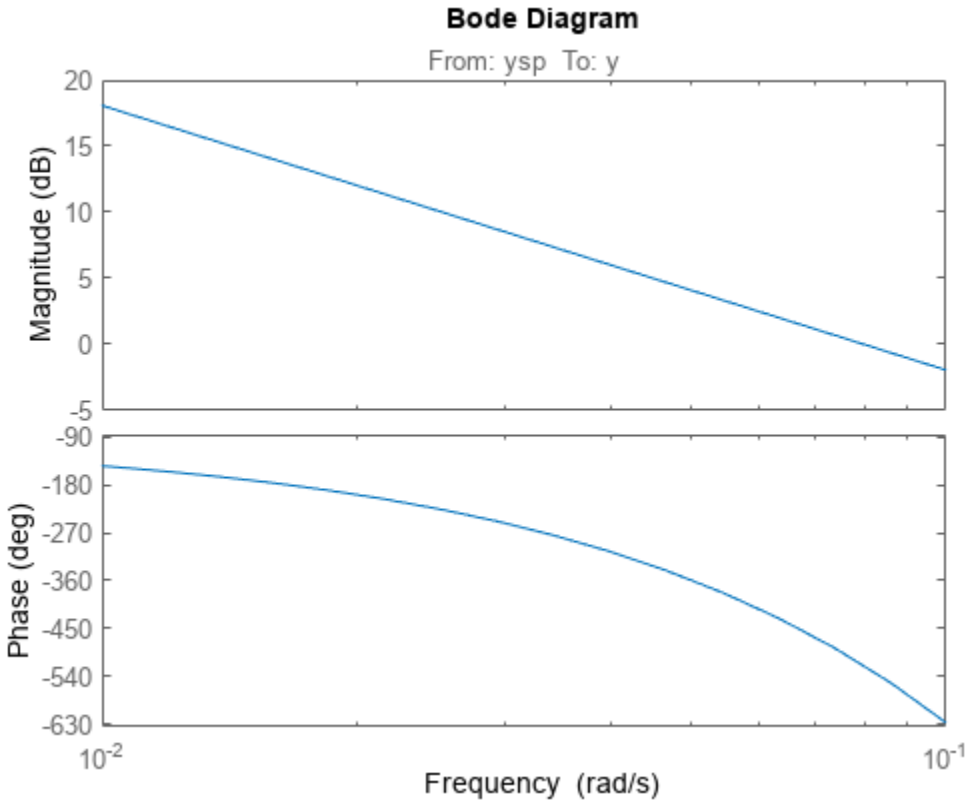
Use these locations as inputs, outputs, or loop openings when you extract responses from the model. For example, extract and plot the response at the system output to a step disturbance at the plant input, *u*.

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



Similarly, calculate the open-loop response of the plant and controller by opening both feedback loops.

```
openings = {'dp','yp'};
L = getIOTransfer(T,'ysp','y',openings);
bodeplot(L)
```



When you create a control system model, you can create an `AnalysisPoint` block explicitly and assign input and output names to it. You can then include it in the input arguments to `connect` as one of the blocks to combine. However, using the `APs` argument to `connect` as illustrated in this example is a simpler way to mark points of interest when building control system models.

See Also

`connect` | `AnalysisPoint` | `sumblk`

Related Examples

- “Control System with Multichannel Analysis Points” on page 2-65

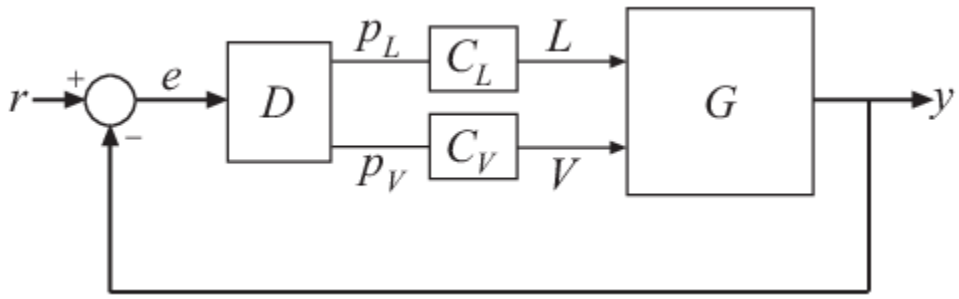
More About

- “Mark Signals of Interest for Control System Analysis and Design” on page 2-68

MIMO Control System

This example shows how to build a MIMO control system using `connect` to interconnect Numeric LTI models on page 1-10 and tunable Control Design Blocks on page 1-12.

Consider the following two-input, two-output control system.



The plant G is a distillation column with two inputs and two outputs. The two inputs are the reflux L and boilup V. The two outputs are the concentrations of two chemicals, represented by the vector signal $y = [y_1,y_2]$. You can represent this plant model as:

$$G(s) = \frac{1}{75s+1}\begin{bmatrix} 87.8 & -86.4 \\ 108.2 & -109.6 \end{bmatrix}.$$

The vector setpoint signal $r = [r_1,r_2]$ specifies the desired concentrations of the two chemicals. The vector error signal e represents the input to D, a static 2-by-2 decoupling matrix. C_L and C_V represent independent PI controllers that control the two inputs of G.

To create a two-input, two-output model representing this closed-loop control system:

1 Create a Numeric LTI model representing the 2-by-2 plant G.

```
s = tf('s','TimeUnit','minutes');
G = [87.8 -86.4 ; 108.2 -109.6]/(75*s+1);
G.InputName = {'L','V'};
G.OutputName = 'y';
```

When you construct the closed-loop model, `connect` uses the input and output names to form connections between the block diagram components. Therefore, you must assign names to the inputs and outputs of the transfer function `G` in either of the following ways: .

- You can assign input and output names to individual signals by specifying signal names in a cell array, as in `G.InputName = {'L','V'}`
- Alternatively, you can use vector signal naming, which the software automatically expands. For example, the command `G.OutputName = 'y'` assigns the names `'y(1)'` and `'y(2)'` to the outputs of `G`.

2 Create tunable Control Design Blocks representing the decoupling matrix D and the PI controllers C_L and C_V.

```
D = tunableGain('Decoupler',eye(2));
D.u = 'e';
D.y = {'pL','pV'};
```

```
C_L = tunablePID('C_L','pi');  C_L.TimeUnit = 'minutes';
C_L.u = 'pL'; C_L.y = 'L';

C_V = tunablePID('C_V','pi');  C_V.TimeUnit = 'minutes';
C_V.u = 'pV'; C_V.y = 'V';
```

Note `u` and `y` are shorthand notations for the `InputName` and `OutputName` properties, respectively. Thus, for example, entering:

```
D.u = 'e';
D.y = {'pL','pV'};
```

is equivalent to entering:

```
D.InputName = 'e';
D.OutputName = {'pL','pV'};
```

---

3 Create the summing junction.

The summing junction produces the error signals *e* by taking the difference between *r* and *y*.

```
Sum = sumblk('e = r - y',2);
```

`Sum` represents the transfer function for the summing junction described by the formula `'e = r - y'`. The second argument to `sumblk` specifies that the inputs and outputs of `Sum` are each vector signals of length 2. The software therefore automatically assigns the signal names `{'r(1)','r(2)','y(1)','y(2)'}` to `Sum.InputName` and `{'e(1)','e(2)'}` to `Sum.OutputName`.

4 Join all components to build the closed-loop system from *r* to *y*.

```
CLry = connect(G,D,C_L,C_V,Sum,'r','y');
```

The arguments to the `connect` function include all the components of the closed-loop system, in any order. `connect` automatically combines the components using the input and output names to join signals.

The last two arguments to `connect` specify the output and input signals of the closed-loop model, respectively. The resulting `genss` model `CLry` has two-inputs and two outputs.

See Also

`connect` | `sumblk`

Related Examples

- “Control System Model with Both Numeric and Tunable Components” on page 4-7
- “Multi-Loop Control System” on page 4-9

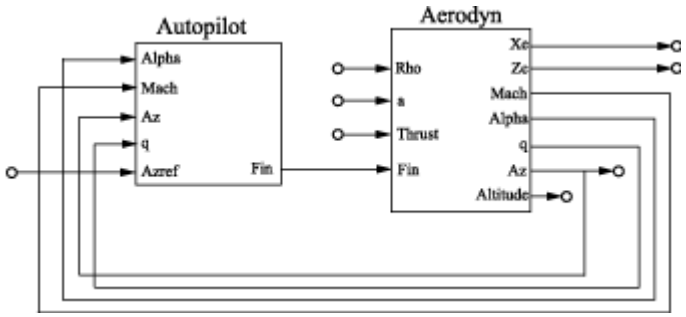
More About

- “Catalog of Model Interconnections” on page 4-3

MIMO Feedback Loop

This example shows how to obtain the closed-loop response of a MIMO feedback loop in three different ways.

In this example, you obtain the response from `Azref` to `Az` of the MIMO feedback loop of the following block diagram.



You can compute the closed-loop response using one of the following three approaches:

- Name-based interconnection with `connect`
- Name-based interconnection with `feedback`
- Index-based interconnection with `feedback`

You can use whichever of these approaches is most convenient for your application.

Load the plant `Aerodyn` and the controller `Autopilot` into the MATLAB® workspace. These models are stored in the datafile `MIMOfeedback.mat`.

```
load('MIMOfeedback.mat')
```

`Aerodyn` is a 4-input, 7-output state-space (`ss`) model. `Autopilot` is a 5-input, 1-output `ss` model. The inputs and outputs of both models names appear as shown in the block diagram.

Compute the closed-loop response from `Azref` to `Az` using `connect`.

```
T1 = connect(Autopilot,Aerodyn,'Azref','Az');
```

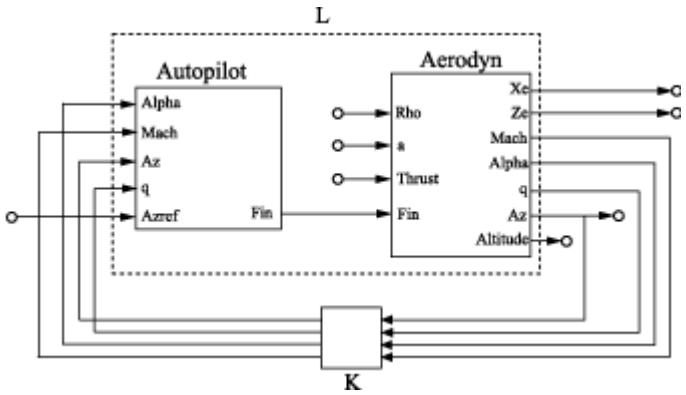
```
Warning: The following block inputs are not used: Rho,a,Thrust.
```

```
Warning: The following block outputs are not used: Xe,Ze,Altitude.
```

The `connect` function combines the models by joining the inputs and outputs that have matching names. The last two arguments to `connect` specify the input and output signals of the resulting model. Therefore, `T1` is a state-space model with input `Azref` and output `Az`. The `connect` function ignores the other inputs and outputs in `Autopilot` and `Aerodyn`.

Compute the closed-loop response from `Azref` to `Az` using name-based interconnection with the `feedback` command. Use the model input and output names to specify the interconnections between `Aerodyn` and `Autopilot`.

When you use the `feedback` function, think of the closed-loop system as a feedback interconnection between an open-loop plant-controller combination `L` and a diagonal unity-gain feedback element `K`. The following block diagram shows this interconnection.




```
L = series(Autopilot,Aerodyn,'Fin');

FeedbackChannels = {'Alpha','Mach','Az','q'};
K = ss(eye(4),'InputName',FeedbackChannels,...
               'OutputName',FeedbackChannels);

T2 = feedback(L,K,'name',+1);
```

The closed-loop model `T2` represents the positive feedback interconnection of `L` and `K`. The `'name'` option causes `feedback` to connect `L` and `K` by matching their input and output names.

`T2` is a 5-input, 7-output state-space model. The closed-loop response from `Azref` to `Az` is `T2('Az','Azref')`.

Compute the closed-loop response from `Azref` to `Az` using `feedback`, using indices to specify the interconnections between `Aerodyn` and `Autopilot`.

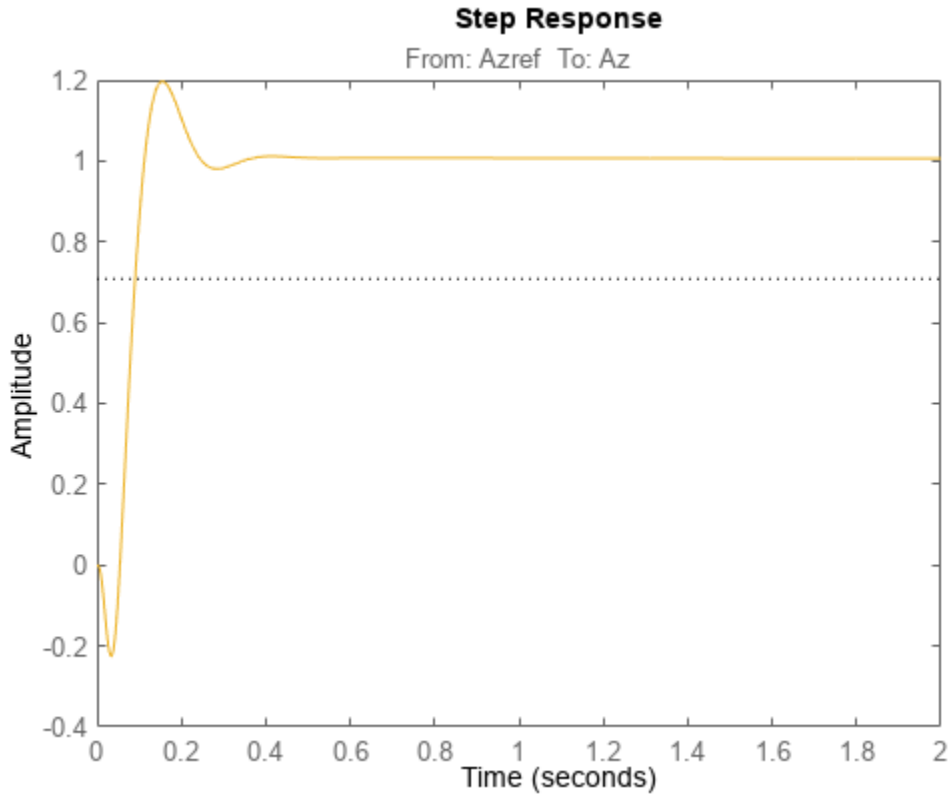
```
L = series(Autopilot,Aerodyn,1,4);
K = ss(eye(4));
T3 = feedback(L,K,[1 2 3 4],[4 3 6 5],+1);
```

The vectors `[1 2 3 4]` and `[4 3 6 5]` specify which inputs and outputs, respectively, complete the feedback interconnection. For example, `feedback` uses output 4 and input 1 of `L` to create the first feedback interconnection. The function uses output 3 and input 2 to create the second interconnection, and so on.

`T3` is a 5-input, 7-output state-space model. The closed-loop response from `Azref` to `Az` is `T3(6,5)`.

Compare the step response from `Azref` to `Az` to confirm that the three approaches yield the same results.

```
step(T1,T2('Az','Azref'),T3(6,5),2)
```



See Also

`connect` | `feedback`

Related Examples

- “Multi-Loop Control System” on page 4-9
- “MIMO Control System” on page 4-15

More About

- “How the Software Determines Properties of Connected Models” on page 4-20

How the Software Determines Properties of Connected Models

When you interconnect models, the operation and the properties of the models you are connecting determine the resulting model's properties. The following table summarizes some general rules governing how resulting model property values are determined.

Property	Expected Behavior
`Ts`	When connecting discrete-time models, all models must have identical or unspecified (`sys.Ts = -1`) sample time. The resulting model inherits the sample time from the connected models.

`InputName`
`OutputName`
`InputGroup`
`InputGroup`
	In general, the resulting model inherits I/O names and I/O groups from connected models. However, conflicting I/O names or I/O groups are not inherited. For example, the `InputName` property for `sys1 + sys2` is left unspecified if `sys1` and `sys2` have different `InputName` property values.

`TimeUnit`	All connected models must have identical `TimeUnit` properties. The resulting model inherits its `TimeUnit` from the connected models.
`Variable`	A model resulting from operations on `tf` or `zpk` models inherits its `Variable` property value from the operands. Conflicts are resolved according the following rules: • For continuous-time models, `'p'` has precedence over `'s'`. • For discrete-time models, `'q^-1'` and `'z^-1'` have precedence over `'q'` and `'z'`, while `'q'` has precedence over `'z'`.
`Notes` `UserData`	Most operations ignore the `Notes` and `UserData` properties. These properties of the resulting model are empty.

See Also

More About

- “Rules That Determine Model Type” on page 4-21

Rules That Determine Model Type

When you combine numeric LTI models on page 1-10 other than `frd` models using `connect`, the resulting model is a state-space (`ss`) model. For other interconnection commands, the resulting model is determined by the following order of precedence:

`ss` > `zpk` > `tf` > `pid` > `pidstd`

For example, connect an `ss` model with a `pid` model.

```
P = ss([-0.8,0.4;0.4,-1.0],[-3.0;1.4],[0.3,0],0);
C = pid(-0.13,-0.61);
CL = feedback(P*C,1)
```

The `ss` model has the highest precedence among Numeric LTI models. Therefore, combining `P` and `C` with any model interconnection command returns an `ss` model.

Combining Numeric LTI models with Generalized LTI models on page 1-12 or with Control Design Blocks on page 1-12 results in Generalized LTI models.

For example, connect the `ss` model `CL` with a Control Design Block.

```
F = tunableTF('F',0,1);
CLF = F*CL
```

`CLF` is a `genss` model.

Any connection that includes a frequency-response model (`frd` or `genfrd`) results in a frequency-response model.

Note The software automatically converts all models to the resulting model type before performing the connection operation.

See Also

`connect` | `feedback` | `series` | `parallel`

Related Examples

- “Numeric Model of SISO Feedback Loop” on page 4-5
- “Multi-Loop Control System” on page 4-9

More About

- “How the Software Determines Properties of Connected Models” on page 4-20
- “Recommended Model Type for Building Block Diagrams” on page 4-22

Recommended Model Type for Building Block Diagrams

This example shows how choice of model type can affect numerical accuracy when interconnecting models.

You can represent block diagram components with any model type. However, certain connection operations yield better numerical accuracy for models in `ss` form.

For example, interconnect two models in series using different model types to see how different representations introduce numerical inaccuracies.

Load the models `Pd` and `Cd`. These models are ninth-order and second-order discrete-time transfer functions, respectively.

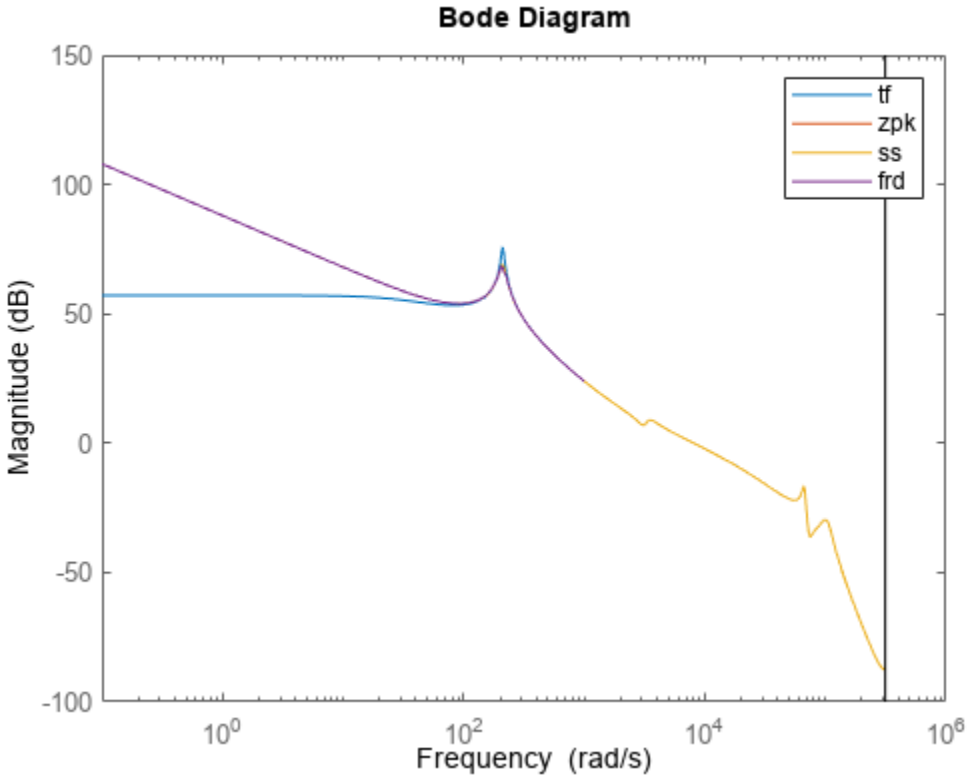
```
load numdemo Pd Cd
```

Compute the open-loop transfer function `L = Pd*Cd` using the `tf`, `zpk`, `ss`, and `frd` representations.

```
Ltf = Pd*Cd;
Lzp = zpk(Pd)*Cd;
Lss = ss(Pd)*Cd;

w = logspace(-1,3,100);
Lfrd = frd(Pd,w)*Cd;
```

Plot the magnitude of the frequency response to compare the four representations.

```
bodemag(Ltf,Lzp,Lss,Lfrd)
legend('tf','zpk','ss','frd')
```



The `tf` representation has lost low-frequency dynamics that other representations preserve.

See Also

More About

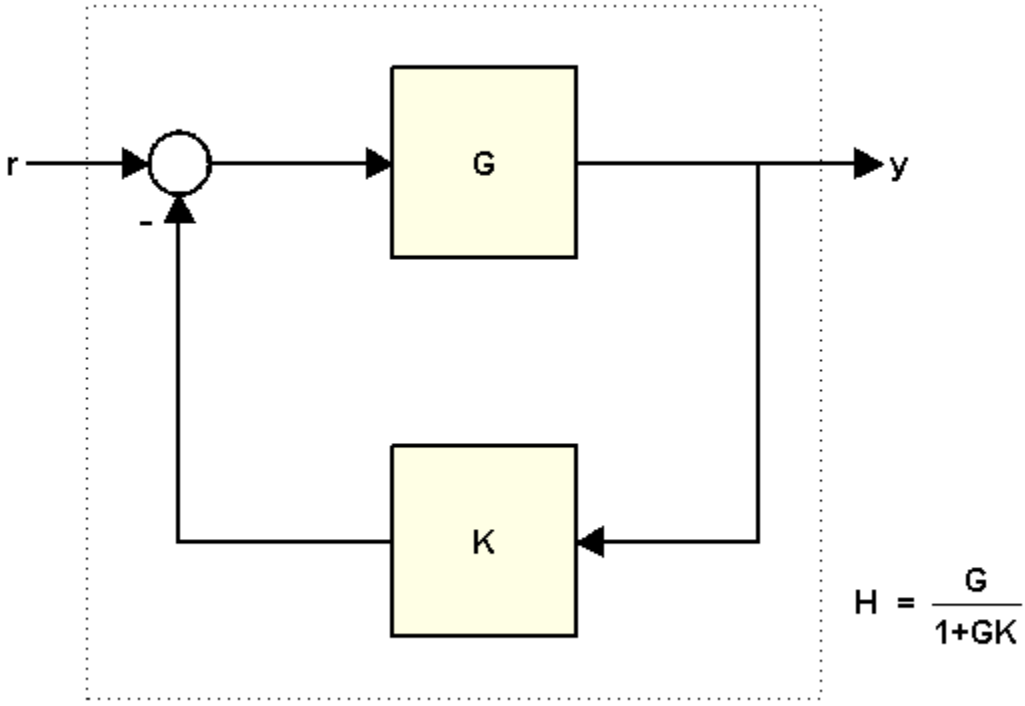
- “Rules That Determine Model Type” on page 4-21

Using FEEDBACK to Close Feedback Loops

This example shows why you should always use FEEDBACK to close feedback loops.

Two Ways of Closing Feedback Loops

Consider the following feedback loop



where

```
K = 2;
G = tf([1 2],[1 .5 3])
```

```
G =

       s + 2
  --------------
  s^2 + 0.5 s + 3

Continuous-time transfer function.
```

You can compute the closed-loop transfer function H from r to y in at least two ways:

- Using the `feedback` command
- Using the formula

$$H = \frac{G}{1 + GK}$$

To compute H using `feedback`, type

```
H = feedback(G,K)
```

```
H =

       s + 2
  ---------------
  s^2 + 2.5 s + 7

Continuous-time transfer function.
```

To compute H from the formula, type

```
H2 = G/(1+G*K)
```

```
H2 =

         s^3 + 2.5 s^2 + 4 s + 6
  -----------------------------------
  s^4 + 3 s^3 + 11.25 s^2 + 11 s + 21

Continuous-time transfer function.
```

Why Using FEEDBACK is Better

A major issue with computing H from the formula is that it inflates the order of the closed-loop transfer function. In the example above, H2 has double the order of H. This is because the expression `G/(1+G*K)` is evaluated as a ratio of the two transfer functions G and `1+G*K`. If

$$G(s) = \frac{N(s)}{D(s)}$$

then `G/(1+G*K)` is evaluated as:

$$\frac{N}{D}\left(\frac{D + KN}{D}\right)^{-1} = \frac{ND}{D(D + KN)}.$$

As a result, the poles of G are added to both the numerator and denominator of H. You can confirm this by looking at the ZPK representation:

```
zpk(H2)
```

```
ans =

       (s+2) (s^2 + 0.5s + 3)
  ---------------------------------
  (s^2 + 0.5s + 3) (s^2 + 2.5s + 7)

Continuous-time zero/pole/gain model.
```

This excess of poles and zeros can negatively impact the accuracy of your results when dealing with high-order transfer functions, as shown in the next example. This example involves a 17th-order transfer function G. As you did before, use both approaches to compute the closed-loop transfer function for K=1:

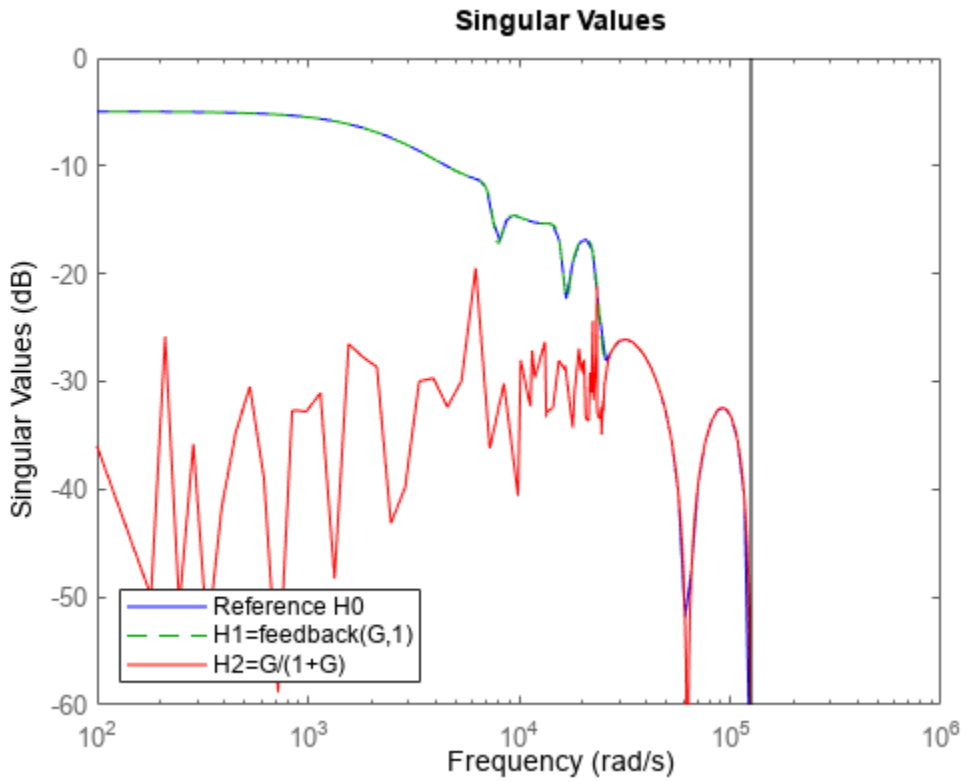
```
load numdemo G
H1 = feedback(G,1);          % good
H2 = G/(1+G);                % bad
```

To have a point of reference, also compute an FRD model containing the frequency response of G and apply `feedback` to the frequency response data directly:

```
w = logspace(2,5.1,100);
H0 = feedback(frd(G,w),1);
```

Then compare the magnitudes of the closed-loop responses:

```
h = sigmaplot(H0,'b',H1,'g--',H2,'r');
legend('Reference H0','H1=feedback(G,1)','H2=G/(1+G)','location','southwest')
setoptions(h,'YlimMode','manual','Ylim',{[-60 0]})
```



The frequency response of `H2` is inaccurate for frequencies below 2e4 rad/s. This inaccuracy can be traced to the additional (cancelling) dynamics introduced near z=1. Specifically, `H2` has about twice as many poles and zeros near z=1 as `H1`. As a result, `H2(z)` has much poorer accuracy near z=1, which distorts the response at low frequencies. See the example “Using the Right Model Representation” on page 1-62 for more details.

See Also

`feedback`

More About

- “MIMO Feedback Loop” on page 4-17
- “Connecting Models” on page 5-36
- “Recommended Model Type for Building Block Diagrams” on page 4-22

Preventing State Duplication in System Interconnections

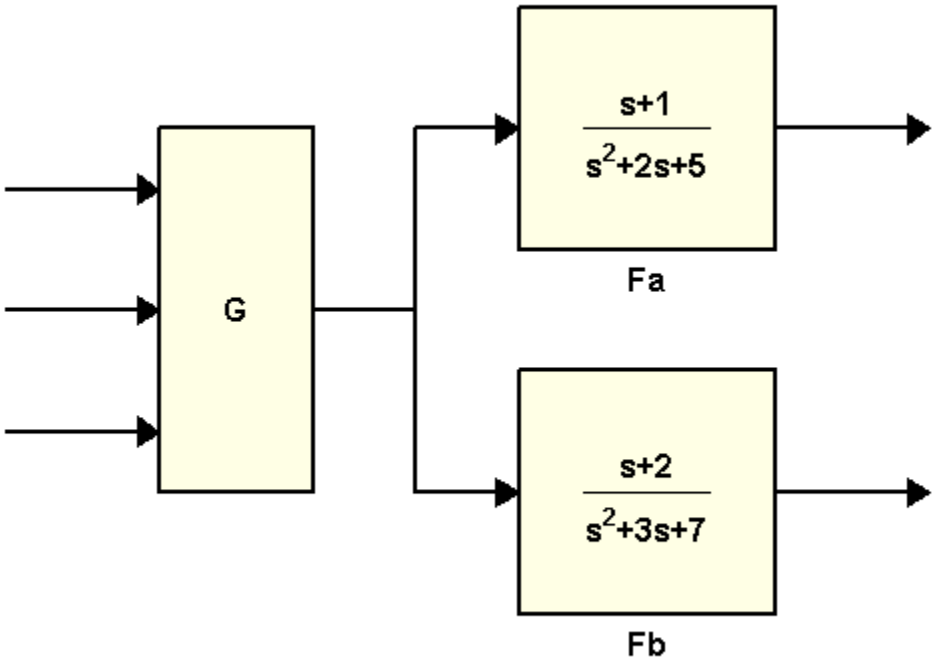
This example shows guidelines for building minimum-order models of LTI system interconnections.

Model Interconnections

You can connect LTI models using the operators +, *, `[,]`, `[;]` and the commands `series`, `parallel`, `feedback`, and `lft`. To prevent duplication of some of the dynamics and ensure that the resulting model has minimal order, it is important that you follow some simple rules:

- Convert all models to the state-space representation before connecting them
- Respect the block diagram structure
- Avoid closed-form expressions and transfer function algebra.

As an illustration, this example compares two ways to compute a state-space model for the following block diagram



where

```
G = [1 , tf(1,[1 0]) , 5];
Fa = tf([1 1] , [1 2 5]);
Fb = tf([1 2] , [1 3 7]);
```

Recommended Method

The best way to connect these three blocks is to convert them to state space and treat the block diagram as a series connection of `G` with `[Fa;Fb]`:

```
H1 = [ss(Fa) ; Fb] * G;
```

To find the order of `H1`, type

```
order(H1)
```

```
ans = 5
```

The order 5 is minimal. Note that because SS has higher precedence than TF, it is enough to convert one of the blocks to state-space (the remaining conversions take place automatically).

Order-Inflating Method

Observe that the overall transfer function is

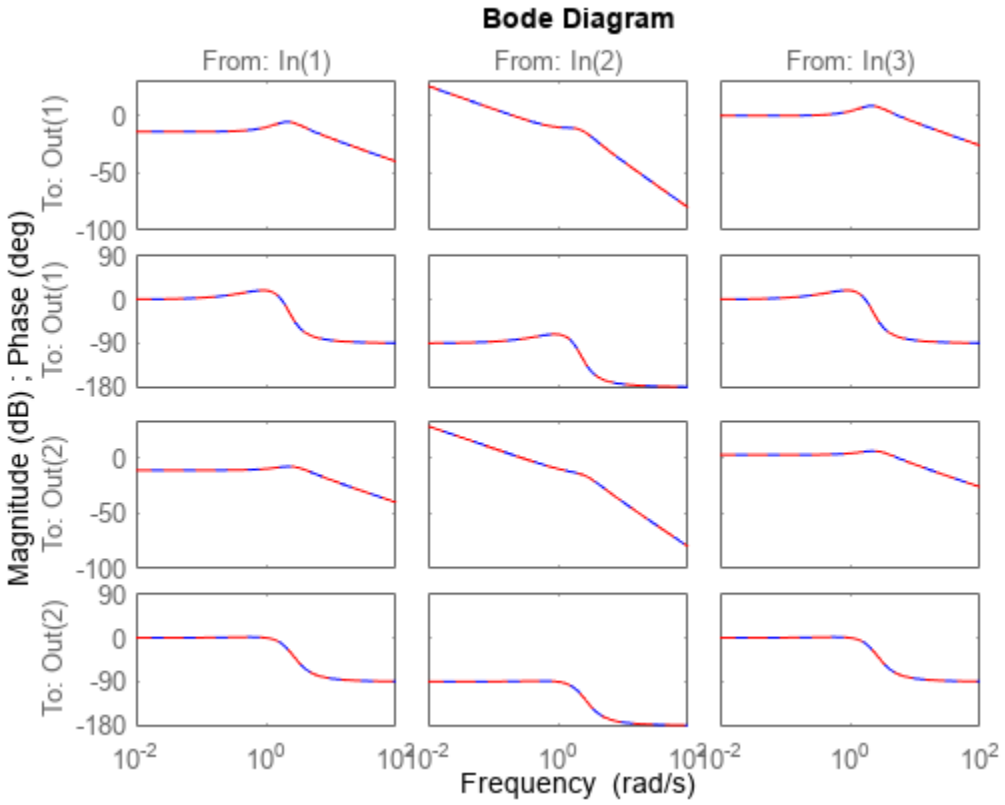
$$H(s) = \begin{pmatrix} F_a(s)G(s) \\ F_b(s)G(s) \end{pmatrix}$$

Therefore, you can also connect the three blocks and compute `H` by typing

```
H2 = ss([Fa * G ; Fb * G]);
```

Verify that the frequency responses of `H1` and `H2` match:

```
bode(H1,'b',H2,'r--')
```



While `H2` is a valid model, its order is 14, almost three times higher than that of `H1`:

```
order(H2)
```

```
ans = 14
```

`H2` has higher order because:

- `G` appears twice in this expression
- The dynamics of `Fa` and `Fb` get replicated three time when evaluating `Fa*G` and `Fb*G`
- The state-space conversion is performed on a 2x3 MIMO transfer matrix with four entries of order 2 and two entries of order 3, yielding a total order of 14.

Using a closed-form expression for the overall transfer function is a bad idea in general as it will typically inflate the order and introduce lots of cancelling pole/zero dynamics.

Conclusion

When connecting LTI models, avoid introducing duplicate dynamics by staying away from closed-form expressions, working with the state-space representation, and breaking block diagrams down to elementary series, parallel, and feedback connections. When in doubt, use the function `connect` which automatically converts all models to state space and is guaranteed to produce minimal realizations of block diagrams.

Model Transformation

- “Conversion Between Model Types” on page 5-2
- “Convert from One Model Type to Another” on page 5-4
- “Get Current Value of Generalized Model by Model Conversion” on page 5-5
- “Decompose a 2-DOF PID Controller into SISO Components” on page 5-7
- “Discretize a Compensator” on page 5-10
- “Improve Accuracy of Discretized System with Time Delay” on page 5-15
- “Convert Discrete-Time System to Continuous Time” on page 5-18
- “Continuous-Discrete Conversion Methods” on page 5-20
- “Upsample Discrete-Time System” on page 5-28
- “Choosing a Resampling Command” on page 5-31
- “Switching Model Representation” on page 5-32
- “Connecting Models” on page 5-36
- “Discretizing and Resampling Models” on page 5-47
- “Discretizing a Notch Filter” on page 5-52
- “Scaling State-Space Models to Maximize Accuracy” on page 5-59
- “Sensitivity of Multiple Roots” on page 5-67

Conversion Between Model Types

Explicit Conversion Between Model Types

You can explicitly convert a model from one representation to another using the model-creation command for the target model type. For example, convert to state-space representation using `ss`, and convert to parallel-form PID using `pid`. For information about converting to a particular model type, see the reference page for that model type.

In general, you can convert from any model type to any other. However, there are a few limitations. For example, you cannot convert:

- `frd` models to analytic model types such as `ss`, `tf`, or `zpk` (unless you perform system identification with System Identification Toolbox software).
- `ss` models with internal delays to `tf` or `zpk`.

You can convert between Numeric LTI models and Generalized LTI models.

- Converting a Generalized LTI model to a Numeric LTI model evaluates any Control Design Blocks at their current (nominal) value.
- Converting a Numeric LTI model to a Generalized LTI model creates a Generalized LTI model with an empty `Blocks` property.

Automatic Conversion Between Model Types

Some algorithms operate only on one type of model object. For example, the algorithm for zero-order-hold discretization with `c2d` can only be performed on state-space models. Similarly, commands such as `tfdata` or `piddata` expect a particular type of model (`tf` or `pid`, respectively). For convenience, such commands automatically convert input models to the appropriate or required model type. For example:

```
sys = ss(0,1,1,0)
[num,den] = tfdata(sys)
```

`tfdata` automatically converts the state-space model `sys` to transfer function form to return numerator and denominator data.

Conversions to state-space form are not uniquely defined. For this reason, automatic conversions to state space do not occur when the result depends on the choice of state coordinates. For example, the `initial` and `kalman` commands require state-space models.

Recommended Working Representation

You can represent numeric system components using any model type. However, Numeric LTI model types are not equally well-suited for numerical computations. In general, it is recommended that you work with state-space (`ss`) or frequency response data (`frd`) models, for the following reasons:

- The accuracy of computations using high-order transfer functions (`tf` or `zpk` models) is sometimes poor, particularly for MIMO or high-order systems. Conversions to a transfer function representation can incur a loss of accuracy.
- When you convert `tf` or `zpk` models to state space using `ss`, the software automatically performs balancing and scaling operations. Balancing and scaling improves the numeric accuracy of

computations involving the model. For more information about balancing and scaling state-space models, see “Scaling State-Space Models” on page 25-2.

In addition, converting back and forth between model types can introduce additional states or orders, or introduce numeric inaccuracies. For example, conversions to state space are not uniquely defined, and are not guaranteed to produce a minimal realization for MIMO models. For a given state-space model `sys`,

```
ss(tf(sys))
```

can return a model with different state-space matrices, or even a different number of states in the MIMO case.

See Also

`tf` | `pid` | `zpk` | `ss` | `frd`

Related Examples

- “Convert from One Model Type to Another” on page 5-4

Convert from One Model Type to Another

This example shows how to convert a numeric LTI model from one type (`pid`) to another type (`tf`).

In general, you can convert a model from one type to another type using the model-creation command for the target type. For example, you can use the `tf` command to convert an `ss` model to transfer function form, or use the `ss` command to convert a `zpk` model to state-space form.

Create a PID controller.

```
pid_sys = pid(1,1.5,3)
```

```
pid_sys =

             1
  Kp + Ki * --- + Kd * s
             s

  with Kp = 1, Ki = 1.5, Kd = 3

Continuous-time PID controller in parallel form.
```

Convert `pid_sys` to a transfer function model.

```
C = tf(pid_sys)
```

```
C =

  3 s^2 + s + 1.5
  ---------------
         s

Continuous-time transfer function.
```

`C` is a `tf` representation of `pid_sys`. `C` has the same dynamics as `pid_sys`, but stores the dynamic parameters as transfer-function numerator and denominator coefficients instead of proportional, integral, and derivative gains.

You can similarly convert transfer function models to `pid` models, provided the `tf` model object represents a parallel-form PID controller with $T_f \geq 0$.

In general, you can use the technique of this example to convert any type of model to another type of model. For more specific information about converting to a particular model type, see the reference page for that model type.

See Also

`tf` | `pid` | `zpk` | `ss` | `frd`

More About

- “Conversion Between Model Types” on page 5-2

Get Current Value of Generalized Model by Model Conversion

This example shows how to get the current value of a generalized model by converting it to a numeric model. This conversion is useful, for example, when you have tuned the parameters of the generalized model using a tuning command such as `systune`.

Create a Generalized Model

Represent the transfer function

$$F = \frac{a}{s + a}$$

containing a real, tunable parameter, `a`, which is initialized to 10.

```
a = realp('a',10);
F = tf(a,[1 a]);
```

`F` is a `genss` model parameterized by `a`.

Tune the Model

Typically, once of you have a generalized model, you tune the parameters of the model using a tuning command such as `systune`. For this example, instead of tuning the model, manually change the value of the tunable component of `F`.

```
F.Blocks.a.Value = 5;
```

Get the Current Value of the Generalized Model

Get the current value of the generalized model by converting it to a numeric model.

```
F_cur_val = tf(F)
```

```
F_cur_val =

    5
  -----
  s + 5

Continuous-time transfer function.
```

`tf(F)` converts the generalized model, `F`, to a numeric transfer function, `F_cur_val`.

To view the state-space representation of the current value of `F`, type `ss(F)`.

To examine the current values of the individual tunable components in a generalized model, use `showBlockValue`.

See Also

`realp` | `tf` | `showBlockValue`

More About

- “Models with Tunable Coefficients” on page 1-15
- “Conversion Between Model Types” on page 5-2
- “Convert from One Model Type to Another” on page 5-4

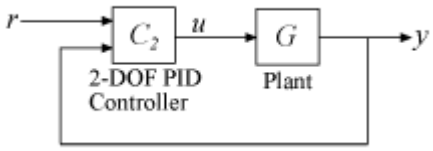
Decompose a 2-DOF PID Controller into SISO Components

This example shows how to extract SISO control components from a 2-DOF PID controller in each of the feedforward, feedback, and filter configurations. The example compares the closed-loop systems in all configurations to confirm that they are all equivalent.

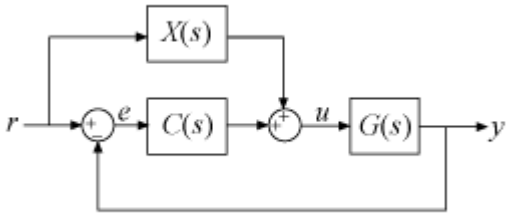
Obtain a 2-DOF PID controller. For this example, create a plant model, and tune a 2-DOF PID controller for it.

```
G = tf(1,[1 0.5 0.1]);
C2 = pidtune(G,'pidf2',1.5);
```

`C2` is a `pid2` controller object. The control architecture for `C2` is as shown in the following illustration.



This control system can be equivalently represented in several other architectures that use only SISO components. In the feedforward configuration, the 2-DOF controller is represented as a SISO PID controller and a feedforward compensator.



Decompose `C2` into SISO control components using the feedforward configuration.

```
[Cff,Xff] = getComponents(C2,'feedforward')
```

```
Cff =

             1            s
  Kp + Ki * --- + Kd * --------
             s          Tf*s+1

  with Kp = 1.12, Ki = 0.23, Kd = 1.3, Tf = 0.122

Continuous-time PIDF controller in parallel form.

Xff =

  -10.898 (s+0.2838)
  ------------------
```

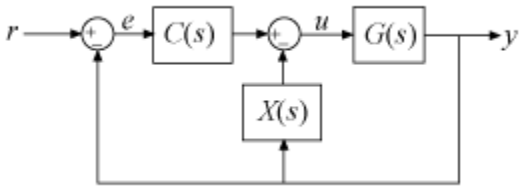
```
      (s+8.181)

Continuous-time zero/pole/gain model.
```

This command returns the SISO PID controller `Cff` as a `pid` object. The feedforward compensator `X` is returned as a `zpk` object.

Construct the closed-loop system for the feedforward configuration.

```
Tff = G*(Cff+Xff)*feedback(1,G*Cff);
```

In the feedback configuration, the 2-DOF controller is represented as a SISO PID controller and an additional feedback compensator.



Decompose `C2` using the feedback configuration and construct that closed-loop system.

```
[Cfb, Xfb] = getComponents(C2,'feedback');
Tfb = G*Cfb*feedback(1,G*(Cfb+Xfb));
```

In the filter configuration, the 2-DOF controller is represented as a SISO PID controller and prefilter on the reference signal.

Decompose `C2` using the filter configuration. Construct that closed-loop system as well.

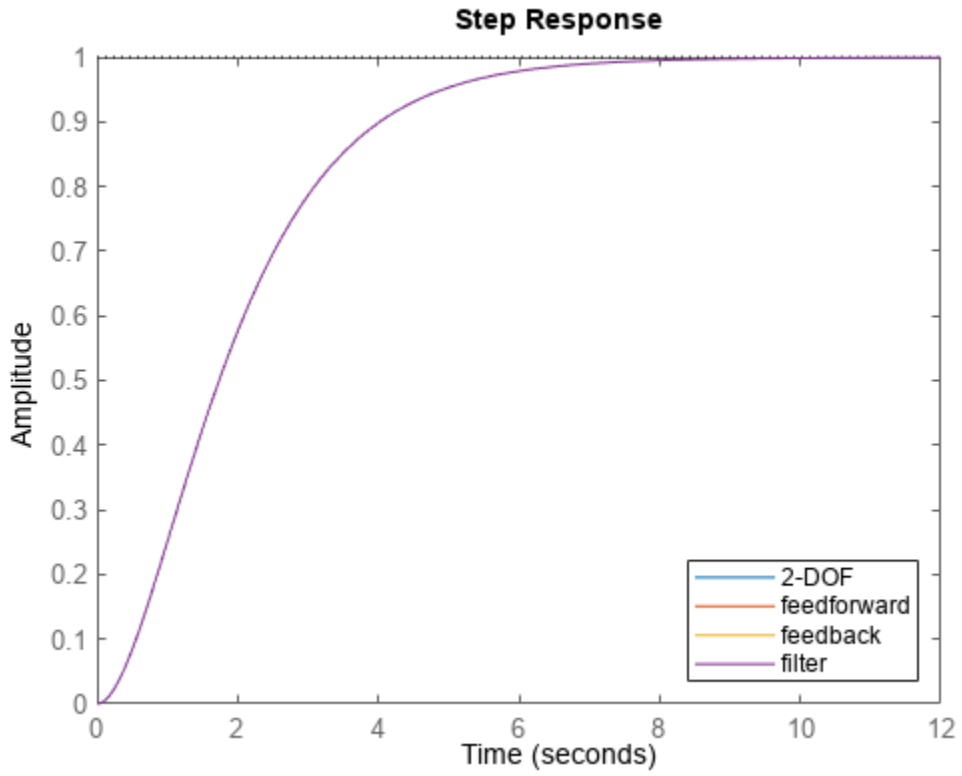
```
[Cfr, Xfr] = getComponents(C2,'filter');
Tfr = Xfr*feedback(G*Cfr,1);
```

Construct the closed-loop system for the original 2-DOF controller, `C2`. To do so, convert `C2` to a two-input, one-output transfer function, and use array indexing to access the channels.

```
Ctf = tf(C2);
Cr = Ctf(1);
Cy = Ctf(2);
T = Cr*feedback(G,Cy,+1);
```

Compare the step responses of all the closed-loop systems.

```
stepplot(T,Tff,Tfb,Tfr)
legend('2-DOF','feedforward','feedback','filter','Location','Southeast')
```



The plots coincide, demonstrating that all the systems are equivalent.

Using a 2-DOF PID controller can yield improved performance compared to a 1-DOF controller. For more information, see “Tune 2-DOF PID Controller (Command Line)” on page 11-11.

See Also

`pid2` | `pidstd2` | `getComponents`

Related Examples

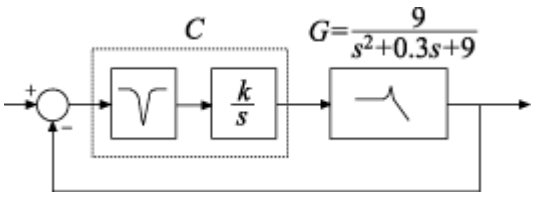
- “Two-Degree-of-Freedom PID Controllers” on page 2-13

Discretize a Compensator

This example shows how to convert a compensator from continuous to discrete time using several discretization methods, to identify a method that yields a good match in the frequency domain.

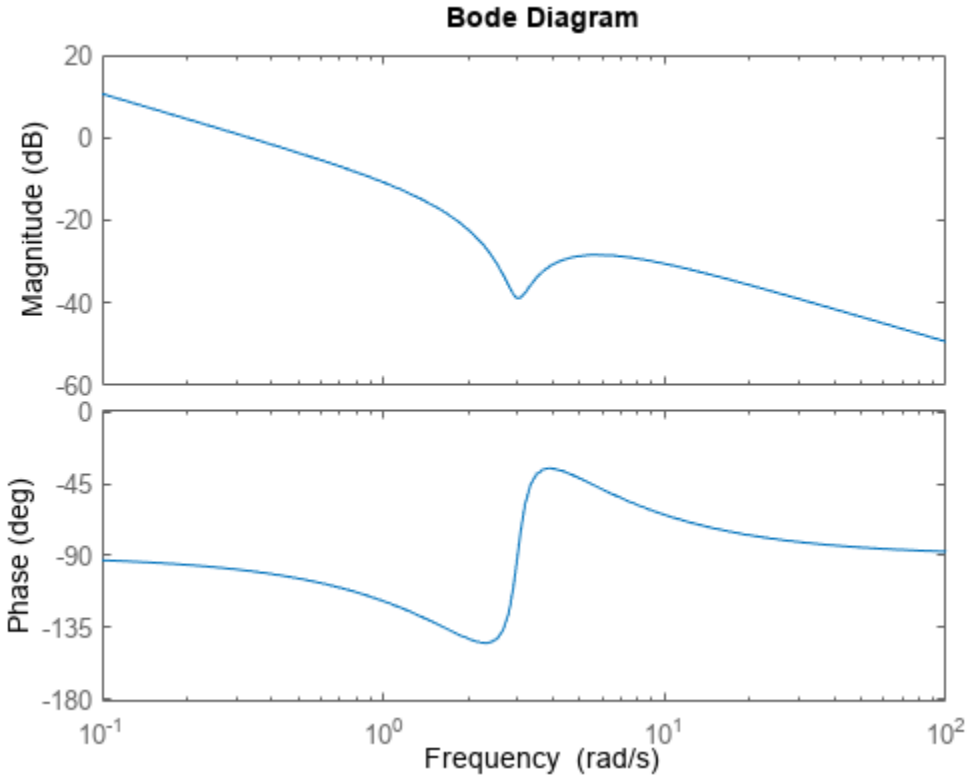
You might design a compensator in continuous time, and then need to convert it to discrete time for a digital implementation. When you do so, you want the discretization to preserve frequency-domain characteristics that are essential to your performance and stability requirements.

In the following control system, `G` is a continuous-time second-order system with a sharp resonance around 3 rad/s.



One valid controller for this system includes a notch filter in series with an integrator. Create a model of this controller.

```
notch = tf([1,0.5,9],[1,5,9]);
integ = pid(0,0.34);
C = integ*notch;
bodeplot(C)
```



The notch filter centered at 3 rad/s counteracts the effect of the resonance in `G`. This configuration allows higher loop gain for a faster overall response.

Discretize the compensator.

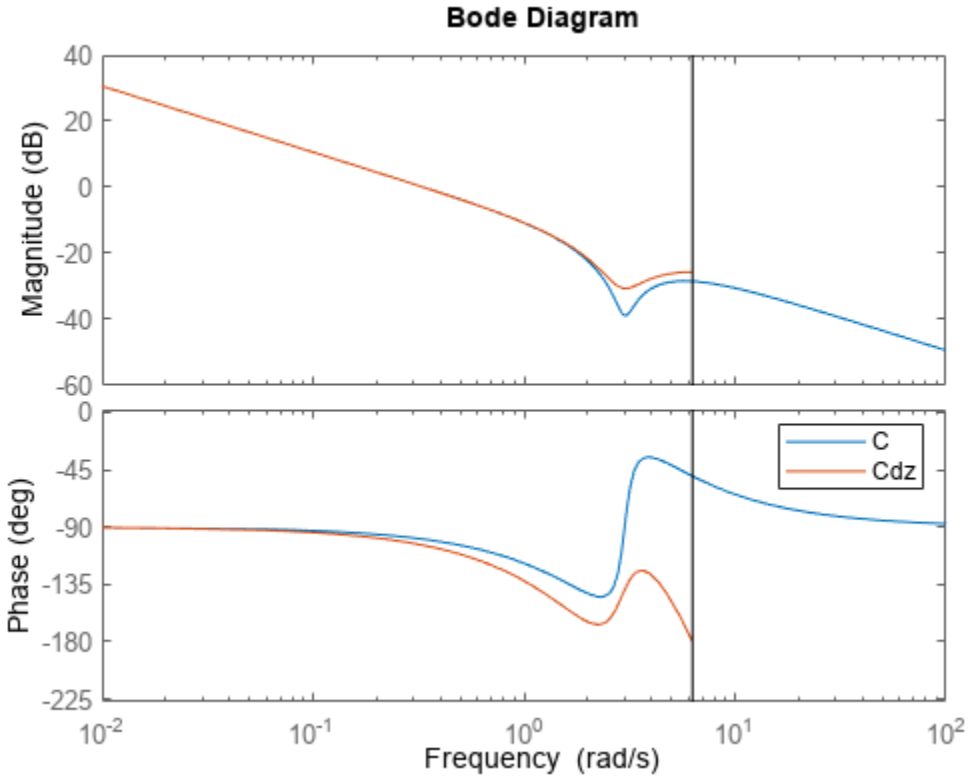
```
Cdz = c2d(C,0.5);
```

The `c2d` command supports several different discretization methods. Since this command does not specify a method, `c2d` uses the default method, Zero-Order Hold (ZOH). In the ZOH method, the time-domain response of the discretized compensator matches the continuous-time response at each time step.

The discretized controller `Cdz` has a sample time of 0.5 s. In practice, the sample time you choose might be constrained by the system in which you implement your controller, or by the bandwidth of your control system.

Compare the frequency-domain response of `C` and `Cdz`.

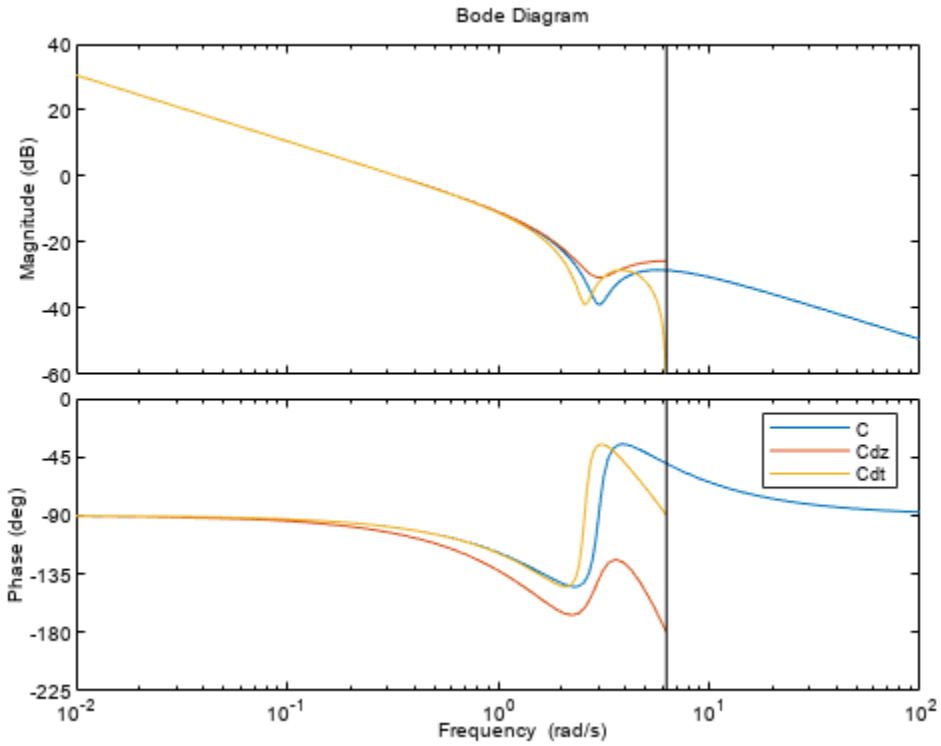
```
bodeplot(C,Cdz)
legend('C','Cdz');
```



The vertical line marks the Nyquist frequency, π/T_s, where T_s is the sample time. Near the Nyquist frequency, the response of the discretized compensator is distorted relative to the continuous-time response. As a result, the discretized notched filter may not properly counteract the plant resonance.

To fix this, try discretizing the compensator using the Tustin method and compare to the ZOH result. The Tustin discretization method often yields a better match in the frequency domain than the ZOH method.

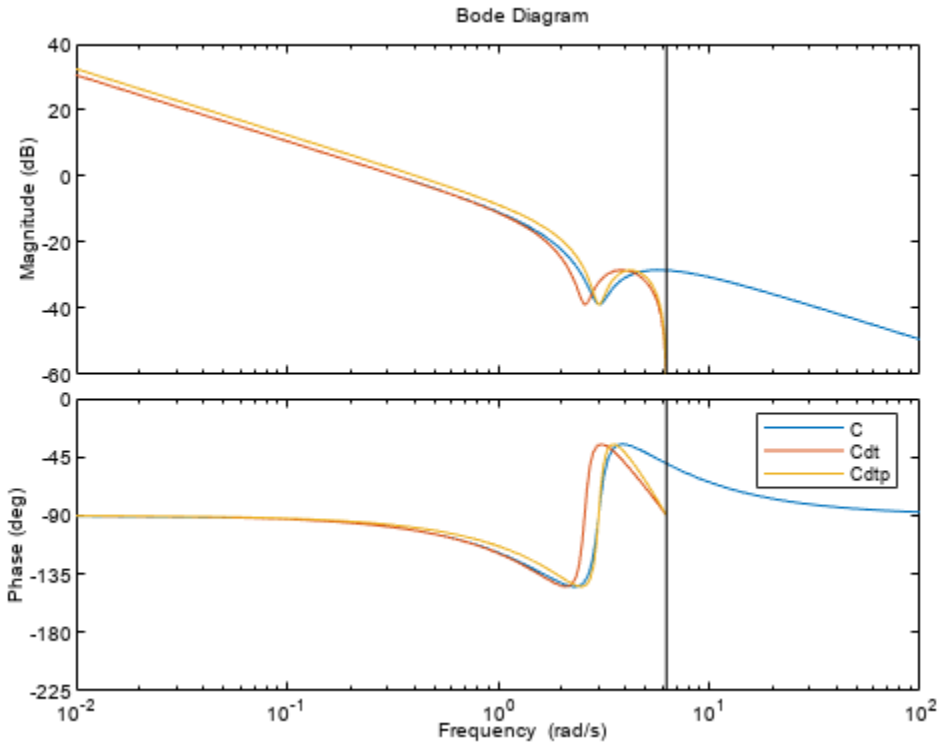
```
Cdt = c2d(C,0.5,'tustin');
plotopts = bodeoptions;
plotopts.Ylim = {[-60,40],[-225,0]};
bodeplot(C,Cdz,Cdt,plotopts)
legend('C','Cdz','Cdt')
```



The Tustin method preserves the depth of the notch. However, the method introduces a frequency shift that is unacceptable for many applications. You can remedy the frequency shift by specifying the notch frequency as the prewarping frequency in the Tustin transform.

Discretize the compensator using the Tustin method with frequency prewarping, and compare the results.

```
discopts = c2dOptions('Method','tustin','PrewarpFrequency',3.0);
Cdtp = c2d(C,0.5,discopts);
bodeplot(C,Cdt,Cdtp,plotopts)
legend('C','Cdt','Cdtp')
```



To specify additional discretization options beyond the discretization method, use `c2dOptions`. Here, the discretization options set `discopts` specifies both the Tustin method and the prewarp frequency. The prewarp frequency is 3.0 rad/s, the frequency of the notch in the compensator response.

Using the Tustin method with frequency prewarping yields a better-matching frequency response than Tustin without prewarping.

See Also

Functions
`c2d` | `c2dOptions`

Live Editor Tasks
Convert Model Rate

More About

- “Continuous-Discrete Conversion Methods” on page 5-20
- “Improve Accuracy of Discretized System with Time Delay” on page 5-15

Improve Accuracy of Discretized System with Time Delay

This example shows how to improve the frequency-domain accuracy of a system with a time delay that is a fractional multiple of the sample time.

For systems with time delays that are not integer multiples of the sample time, the `Tustin` and `Matched` methods by default round the time delays to the nearest multiple of the sample time. To improve the accuracy of these methods for such systems, `c2d` can optionally approximate the fractional portion of the time delay by a discrete-time all-pass filter (a Thiran filter). In this example, discretize the system both without and with an approximation of the fractional portion of the delay and compare the results.

Create a continuous-time transfer function with a transport delay of 2.5 s.

```
G = tf(1,[1,0.2,4],'ioDelay',2.5);
```

Discretize `G` using a sample time of 1 s. `G` has a sharp resonance at 2 rad/s. At a sample time of 1 s, that peak is close to the Nyquist frequency. For a frequency-domain match that preserves dynamics near the peak, use the Tustin method with prewarp frequency 2 rad/s.

```
discopts = c2dOptions('Method','tustin','PrewarpFrequency',2);
Gt = c2d(G,1,discopts)
```

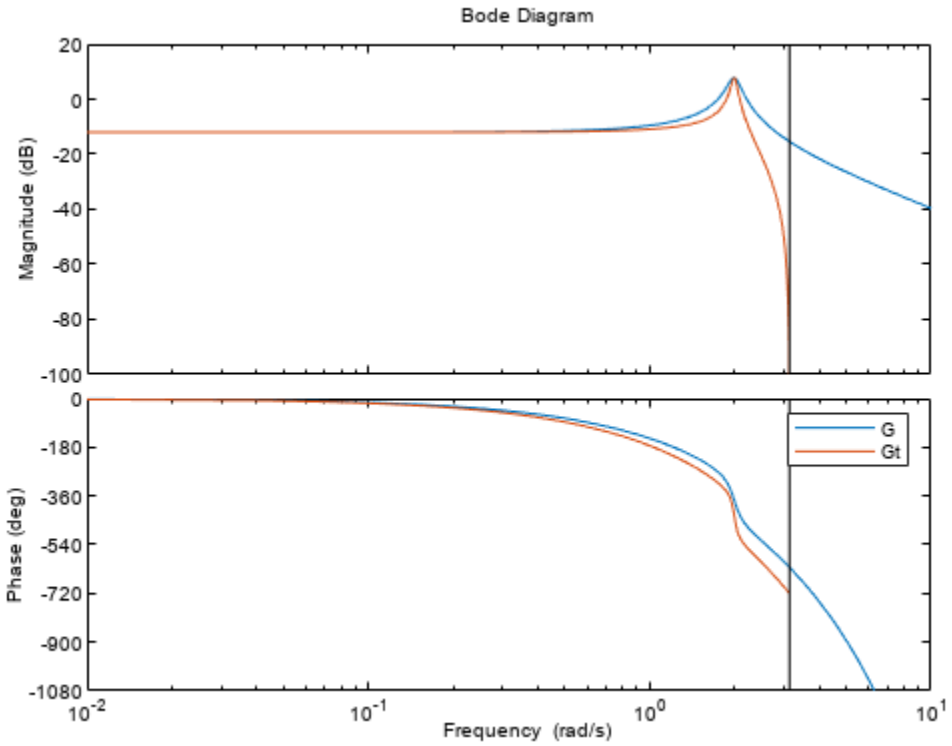
```
Warning: Rounding delays to the nearest multiple of the sampling period. For more accuracy in the
```

```
Gt =

            0.1693 z^2 + 0.3386 z + 0.1693
  z^(-3) * ------------------------------
               z^2 + 0.7961 z + 0.913

Sample time: 1 seconds
Discrete-time transfer function.
```

The software warns you that it rounds the fractional time delay to the nearest multiple of the sample time. In this example, the time delay of 2.5 times the sample time (2.5 s) converts to an additional factor of `z^(-3)` in `Gt`.

Compare `Gt` to the continuous-time system `G`.

```
plotopts = bodeoptions;
plotopts.Ylim = {[-100,20],[-1080,0]};
bodeplot(G,Gt,plotopts);
legend('G','Gt')
```



There is a phase lag between the discretized system `Gt` and the continuous-time system `G`, which grows as the frequency approaches the Nyquist frequency. This phase lag is largely due to the rounding of the fractional time delay. In this example, the fractional time delay is half the sample time.

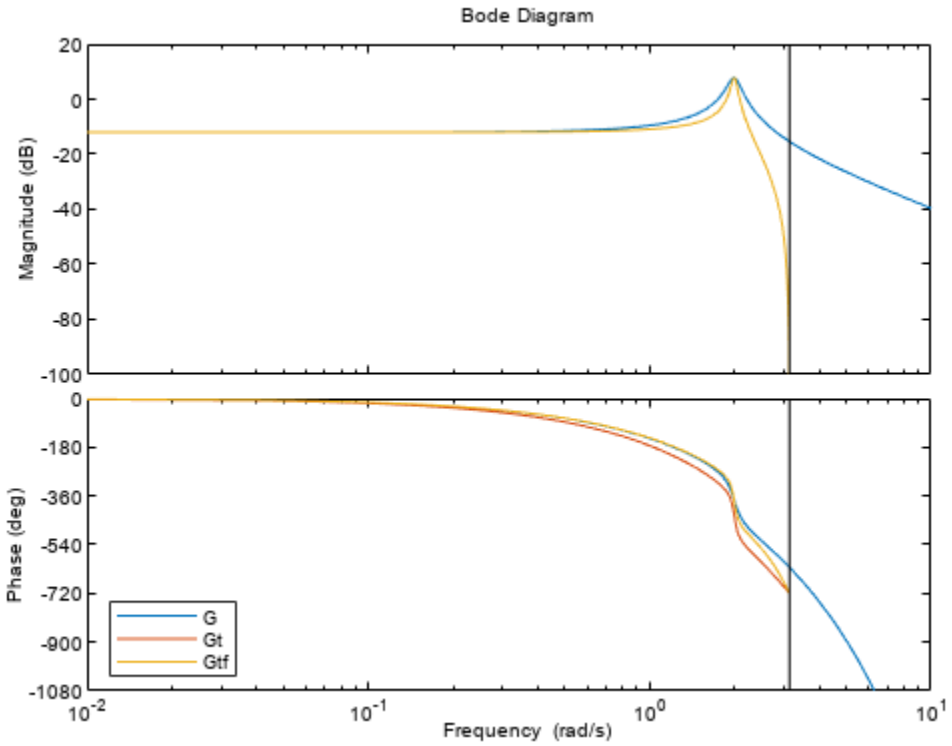
Discretize `G` again using a third-order discrete-time all-pass filter (Thiran filter) to approximate the half-period portion of the delay.

```
discopts.FractDelayApproxOrder = 3;
Gtf = c2d(G,1,discopts);
```

The `FractDelayApproxOrder` option specifies the order of the Thiran filter that approximates the fractional portion of the delay. The other options in `discopts` are unchanged. Thus `Gtf` is a Tustin discretization of `G` with prewarp at 2 rad/s.

Compare `Gtf` to `G` and `Gt`.

```
plotopts.PhaseMatching = 'on';
bodeplot(G,Gt,Gtf,plotopts);
legend('G','Gt','Gtf','Location','SouthWest')
```



The magnitudes of `Gt` and `Gtf` are identical. However, the phase of `Gtf` provides a better match to the phase of the continuous-time system through the resonance. As the frequency approaches the Nyquist frequency, this phase match deteriorates. A higher-order approximation of the fractional delay would improve the phase matching closer to the Nyquist frequencies. However, each additional order of approximation adds an additional order (or state) to the discretized system.

If your application requires accurate frequency-matching near the Nyquist frequency, use `c2dOptions` to make `c2d` approximate the fractional portion of the time delay as a Thiran filter.

See Also

Functions
`c2d` | `c2dOptions` | `thiran`

Live Editor Tasks
Convert Model Rate

More About

- “Continuous-Discrete Conversion Methods” on page 5-20
- “Discretize a Compensator” on page 5-10

Convert Discrete-Time System to Continuous Time

This example shows how to convert a discrete-time system to continuous time using `d2c`, and compare the results using two different interpolation methods.

Convert the following second-order discrete-time system to continuous time using the zero-order hold (ZOH) method:

$$G(z) = \frac{z + 0.5}{(z + 2)(z - 5)}.$$

```
G = zpk(-0.5,[-2,5],1,0.1);
Gcz = d2c(G)
```

```
Warning: The model order was increased to handle real negative poles.
```

```
Gcz =

   2.6663 (s^2 + 14.28s + 780.9)
  -------------------------------
  (s-16.09) (s^2 - 13.86s + 1035)

Continuous-time zero/pole/gain model.
```

When you call `d2c` without specifying a method, the function uses ZOH by default. The ZOH interpolation method increases the model order for systems that have real negative poles. This order increase occurs because the interpolation algorithm maps real negative poles in the z domain to pairs of complex conjugate poles in the s domain.

Convert G to continuous time using the Tustin method.

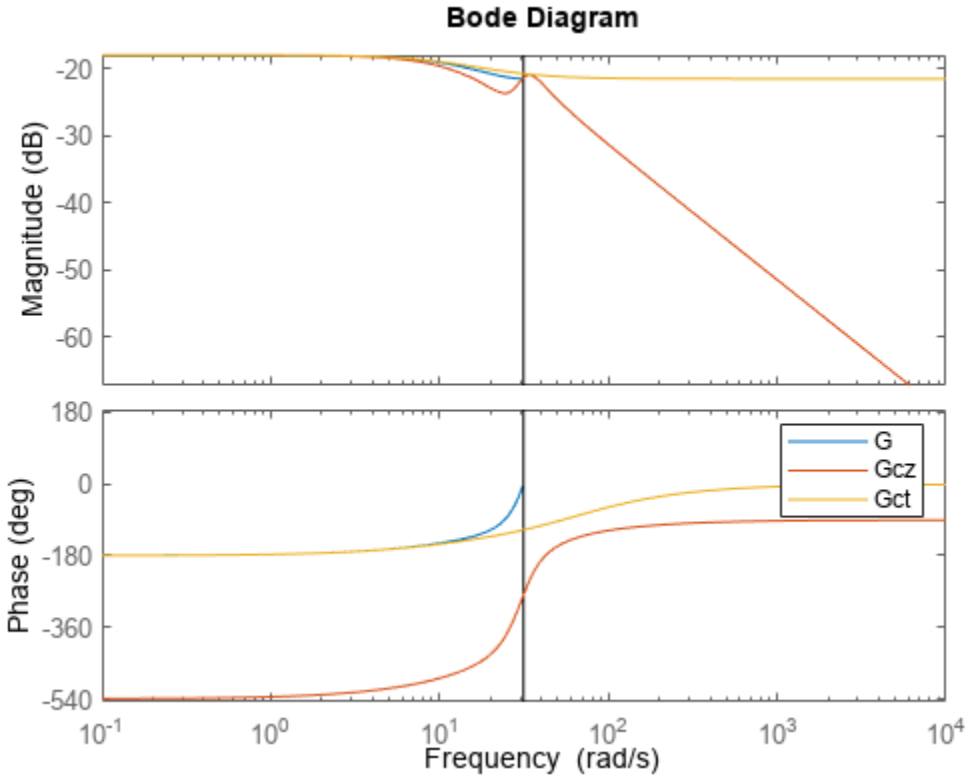
```
Gct = d2c(G,'tustin')
```

```
Gct =

  0.083333 (s+60) (s-20)
  ----------------------
     (s-60) (s-13.33)

Continuous-time zero/pole/gain model.
```

In this case, there is no order increase.

Compare frequency responses of the interpolated systems with that of `G`.

```
bode(G,Gcz,Gct)
legend('G','Gcz','Gct')
```



In this case, the Tustin method provides a better frequency-domain match between the discrete system and the interpolation. However, the Tustin interpolation method is undefined for systems with poles at $z = -1$ (integrators), and is ill-conditioned for systems with poles near $z = 1$.

See Also

Functions
`d2c` | `d2cOptions`

Live Editor Tasks
Convert Model Rate

More About

- “Continuous-Discrete Conversion Methods” on page 5-20
- “Discretize a Compensator” on page 5-10

Continuous-Discrete Conversion Methods

Control System Toolbox offers several discretization and interpolation methods for converting dynamic system models between continuous time and discrete time and for resampling discrete-time models. Some methods tend to provide a better frequency-domain match between the original and converted systems, while others provide a better match in the time domain. Use the following table to help select the method that is best for your application.

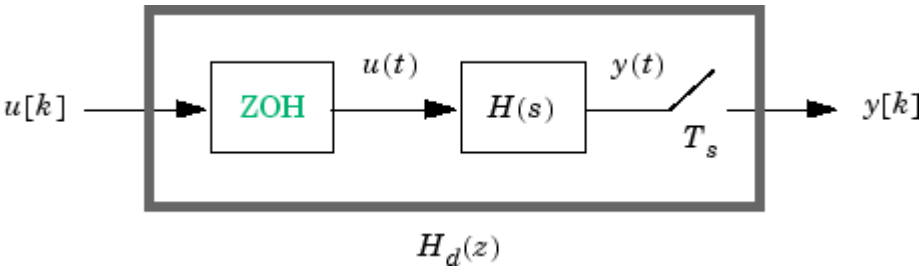
Discretization Method	Use When
"Zero-Order Hold" on page 5-20	You want an exact discretization in the time domain for staircase inputs.
"First-Order Hold" on page 5-22	You want an exact discretization in the time domain for piecewise linear inputs.
"Impulse-Invariant Mapping" on page 5-22 (continuous-to-discrete conversion only)	You want an exact discretization in the time domain for impulse train inputs.
"Tustin Approximation" on page 5-23	• You want good matching in the frequency domain between the continuous- and discrete-time models. • Your model has important dynamics at some particular frequency.
"Zero-Pole Matching Equivalents" on page 5-26	• You have a SISO model. • You want good matching in the frequency domain between the continuous- and discrete-time models.
"Least Squares" on page 5-26 (continuous-to-discrete conversion only)	• You have a SISO model. • You want good matching in the frequency domain between the continuous- and discrete-time models. • You want to capture fast system dynamics but must use a larger sample time.

For information about how to specify a conversion method at the command line, see `c2d`, `d2c`, and `d2d`. You can experiment interactively with different discretization methods in the Live Editor using the **Convert Model Rate** task.

Zero-Order Hold

The Zero-Order Hold (ZOH) method provides an exact match between the continuous- and discrete-time systems in the time domain for staircase inputs.

The following block diagram illustrates the zero-order-hold discretization $H_d(z)$ of a continuous-time linear model $H(s)$.



The ZOH block generates the continuous-time input signal $u(t)$ by holding each sample value $u(k)$ constant over one sample period:

$$u(t) = u[k], \qquad kT_s \le t \le (k+1)T_s$$

The signal $u(t)$ is the input to the continuous system $H(s)$. The output $y[k]$ results from sampling $y(t)$ every T_s seconds.

Conversely, given a discrete system $H_d(z)$, `d2c` produces a continuous system $H(s)$. The ZOH discretization of $H(s)$ coincides with $H_d(z)$.

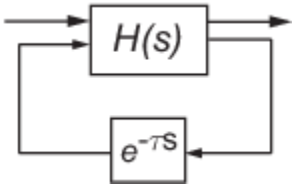
The ZOH discrete-to-continuous conversion has the following limitations:

- `d2c` cannot convert LTI models with poles at $z = 0$.
- For discrete-time LTI models having negative real poles, ZOH `d2c` conversion produces a continuous system with higher order. The model order increases because a negative real pole in the z domain maps to a pure imaginary value in the s domain. Such mapping results in a continuous-time model with complex data. To avoid this issue, the software instead introduces a conjugate pair of complex poles in the s domain. See “Convert Discrete-Time System to Continuous Time” on page 5-18 for an example.

ZOH Method for Systems with Time Delays

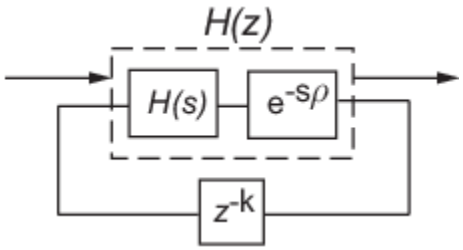
You can use the ZOH method to discretize SISO or MIMO continuous-time models with time delays. The ZOH method yields an exact discretization for systems with input delays, output delays, or transport delays.

For systems with internal delays (delays in feedback loops), the ZOH method results in approximate discretizations. The following figure illustrates a system with an internal delay.



For such systems, `c2d` performs the following actions to compute an approximate ZOH discretization:

1. Decomposes the delay τ as $\tau = kT_s + \rho$ with $0 \le \rho < T_s$.
2. Absorbs the fractional delay ρ into $H(s)$.
3. Discretizes $H(s)$ to $H(z)$.
4. Represents the integer portion of the delay kT_s as an internal discrete-time delay z^{-k}. The final discretized model appears in the following figure:



First-Order Hold

The First-Order Hold (FOH) method provides an exact match between the continuous- and discrete-time systems in the time domain for piecewise linear inputs.

FOH differs from ZOH by the underlying hold mechanism. To turn the input samples $u[k]$ into a continuous input $u(t)$, FOH uses linear interpolation between samples:

$$u(t) = u[k] + \frac{t - kT_s}{T_s}(u[k+1] - u[k]), \quad kT_s \le t \le (k+1)T_s$$

In general, this method is more accurate than ZOH for systems driven by smooth inputs.

This FOH method differs from standard causal FOH and is more appropriately called triangle approximation (see [2], p. 228). The method is also known as ramp-invariant approximation.

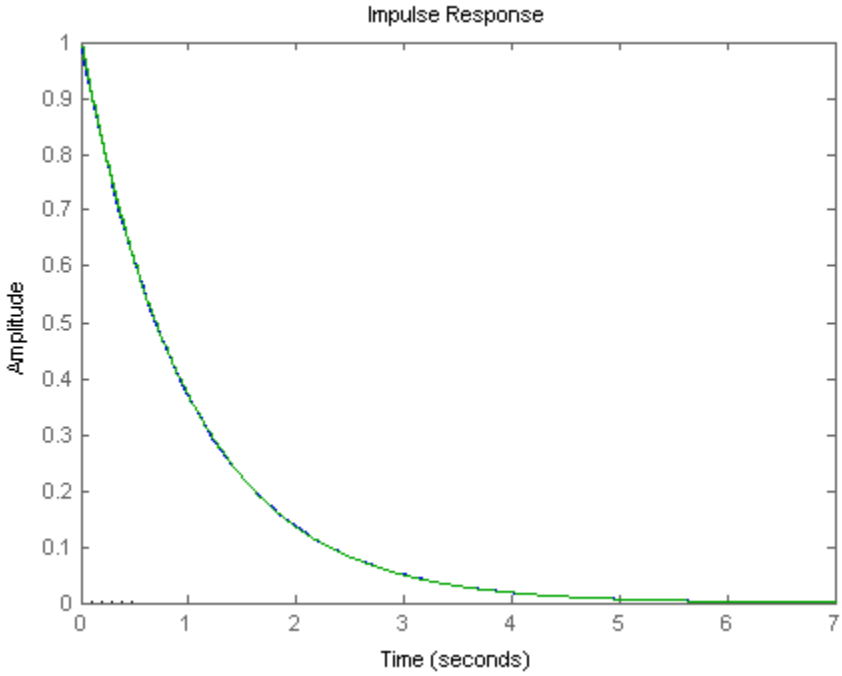
FOH Method for Systems with Time Delays

You can use the FOH method to discretize SISO or MIMO continuous-time models with time delays. The FOH method handles time delays in the same way as the ZOH method. See “ZOH Method for Systems with Time Delays” on page 5-21.

Impulse-Invariant Mapping

The impulse-invariant mapping produces a discrete-time model with the same impulse response as the continuous time system. For example, compare the impulse response of a first-order continuous system with the impulse-invariant discretization:

```
G = tf(1,[1,1]);
Gd1 = c2d(G,0.01,'impulse');
impulse(G,Gd1)
```



The impulse response plot shows that the impulse responses of the continuous and discretized systems match.

Impulse-Invariant Mapping for Systems with Time Delays

You can use impulse-invariant mapping to discretize SISO or MIMO continuous-time models with time delays, except that the method does not support `ss` models with internal delays. For supported models, impulse-invariant mapping yields an exact discretization of the time delay.

Tustin Approximation

The Tustin or bilinear approximation yields the best frequency-domain match between the continuous-time and discretized systems. This method relates the s-domain and z-domain transfer functions using the approximation:

$$z = e^{sT_s} \approx \frac{1 + sT_s/2}{1 - sT_s/2}.$$

In `c2d` conversions, the discretization $H_d(z)$ of a continuous transfer function $H(s)$ is:

$$H_d(z) = H(s'), \qquad s' = \frac{2}{T_s}\frac{z-1}{z+1}$$

Similarly, the `d2c` conversion relies on the inverse correspondence

$$H(s) = H_d(z'), \qquad z' = \frac{1 + sT_s/2}{1 - sT_s/2}$$

When you convert a state-space model using the Tustin method, the states are not preserved. The state transformation depends upon the state-space matrices and whether the system has time delays. For example, for an explicit ($E = I$) continuous-time model with no time delays, the state vector $w[k]$ of the discretized model is related to the continuous-time state vector $x(t)$ by:

$$w[kT_s] = \left(I - A\frac{T_s}{2}\right)x(kT_s) - \frac{T_s}{2}Bu(kT_s) = x(kT_s) - \frac{T_s}{2}(Ax(kT_s) + Bu(kT_s)).$$

T_s is the sample time of the discrete-time model. A and B are state-space matrices of the continuous-time model.

The Tustin approximation is not defined for systems with poles at $z = -1$ and is ill-conditioned for systems with poles near $z = -1$.

Tustin Approximation with Frequency Prewarping

If your system has important dynamics at a particular frequency that you want the transformation to preserve, you can use the Tustin method with frequency prewarping. This method ensures a match between the continuous- and discrete-time responses at the prewarp frequency.

The Tustin approximation with frequency prewarping uses the following transformation of variables:

$$H_d(z) = H(s'), \qquad s' = \frac{\omega}{\tan(\omega T_s/2)}\frac{z-1}{z+1}$$

This change of variable ensures the matching of the continuous- and discrete-time frequency responses at the prewarp frequency ω, because of the following correspondence:

$$H(j\omega) = H_d\left(e^{j\omega T_s}\right)$$

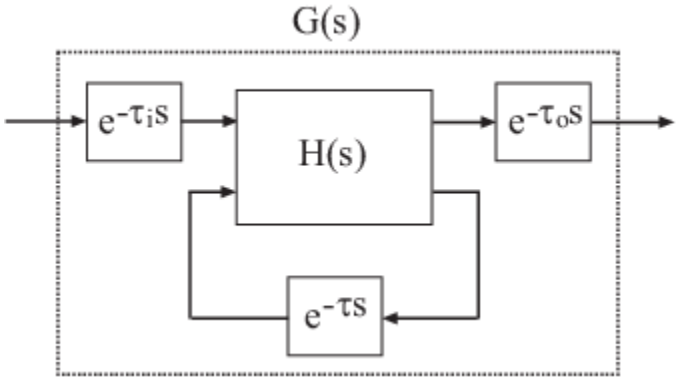
Tustin Approximation for Systems with Time Delays

You can use the Tustin approximation to discretize SISO or MIMO continuous-time models with time delays.

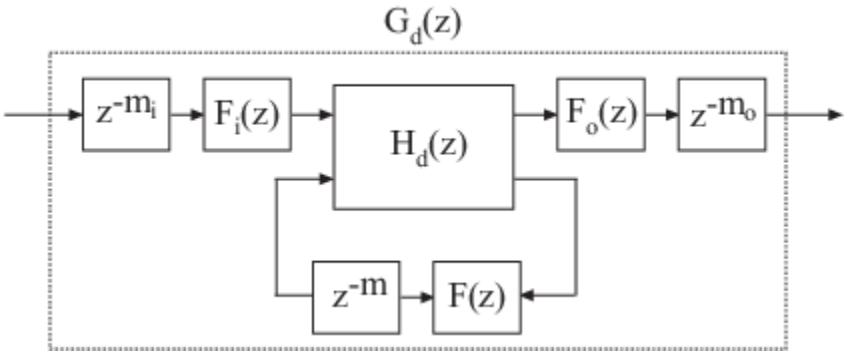
By default, the Tustin method rounds any time delay to the nearest multiple of the sample time. Therefore, for any time delay `tau`, the integer portion of the delay, `k*Ts`, maps to a delay of `k` sampling periods in the discretized model. This approach ignores the residual fractional delay, `tau - k*Ts`.

You can to approximate the fractional portion of the delay by a discrete all-pass filter (Thiran filter) of specified order. To do so, use the `FractDelayApproxOrder` option of `c2dOptions`. See “Improve Accuracy of Discretized System with Time Delay” on page 5-15 for an example.

To understand how the Tustin method handles systems with time delays, consider the following SISO state-space model $G(s)$. The model has input delay τ_i, output delay τ_o, and internal delay τ.



The following figure shows the general result of discretizing $G(s)$ using the Tustin method.



By default, `c2d` converts the time delays to pure integer time delays. The `c2d` command computes the integer delays by rounding each time delay to the nearest multiple of the sample time T_s. Thus, in the default case, m_i = round(τ_i/T_s), m_o = `round`(τ_o/T_s), and m = `round`(τ/T_s).. Also in this case, $F_i(z) = F_o(z) = F(z) = 1$.

If you set `FractDelayApproxOrder` to a non-zero value, `c2d` approximates the fractional portion of the time delays by Thiran filters $F_i(z)$, $F_o(z)$, and $F(z)$.

The Thiran filters add additional states to the model. The maximum number of additional states for each delay is `FractDelayApproxOrder`.

For example, for the input delay τ_i, the order of the Thiran filter $F_i(z)$ is:

`order`($F_i(z)$) = `max`(`ceil`(τ_i/T_s), `FractDelayApproxOrder`).

If `ceil`(τ_i/T_s) < `FractDelayApproxOrder`, the Thiran filter $F_i(z)$ approximates the entire input delay τ_i. If `ceil`(τ_i/T_s) > `FractDelayApproxOrder`, the Thiran filter only approximates a portion of the input delay. In that case, `c2d` represents the remainder of the input delay as a chain of unit delays z^{-m_i}, where

m_i = `ceil`(τ_i/T_s) – `FractDelayApproxOrder`

`c2d` uses Thiran filters and `FractDelayApproxOrder` in a similar way to approximate the output delay τ_o and the internal delay τ.

When you discretize`tf` and `zpk` models using the Tustin method, `c2d` first aggregates all input, output, and transport delays into a single transport delay τ_{TOT} for each channel. `c2d` then

approximates τ_{TOT} as a Thiran filter and a chain of unit delays in the same way as described for each of the time delays in `ss` models.

For more information about Thiran filters, see the `thiran` reference page and [4].

Zero-Pole Matching Equivalents

This method of conversion, which computes zero-pole matching equivalents, applies only to SISO systems. The continuous and discretized systems have matching DC gains. Their poles and zeros are related by the transformation:

$$z_i = e^{s_i T_s}$$

where:

- z_i is the ith pole or zero of the discrete-time system.
- s_i is the ith pole or zero of the continuous-time system.
- T_s is the sample time.

See [2] for more information.

Zero-Pole Matching for Systems with Time Delays

You can use zero-pole matching to discretize SISO continuous-time models with time delay, except that the method does not support `ss` models with internal delays. The zero-pole matching method handles time delays in the same way as the Tustin approximation. See “Tustin Approximation for Systems with Time Delays” on page 5-24.

Least Squares

The least squares method minimizes the error between the frequency responses of the continuous-time and discrete-time systems up to the Nyquist frequency using a vector-fitting optimization approach. This method is useful when you want to capture fast system dynamics but must use a larger sample time, for example, when computational resources are limited.

This method is supported only by the `c2d` function and only for SISO systems.

As with Tustin approximation and zero-pole matching, the least squares method provides a good match between the frequency responses of the original continuous-time system and the converted discrete-time system. However, when using the least squares method with:

- The same sample time as Tustin approximation or zero-pole matching, you get a smaller difference between the continuous-time and discrete-time frequency responses.
- A lower sample time than what you would use with Tustin approximation or zero-pole matching, you can still get a result that meets your requirements. Doing so is useful if computational resources are limited, since the slower sample time means that the processor must do less work.

References

[1] Åström, K.J. and B. Wittenmark, *Computer-Controlled Systems: Theory and Design*, Prentice-Hall, 1990, pp. 48-52.

[2] Franklin, G.F., Powell, D.J., and Workman, M.L., *Digital Control of Dynamic Systems* (3rd Edition), Prentice Hall, 1997.

[3] Smith, J.O. III, "Impulse Invariant Method", *Physical Audio Signal Processing*, August 2007. `https://www.dsprelated.com/dspbooks/pasp/Impulse_Invariant_Method.html`.

[4] T. Laakso, V. Valimaki, "Splitting the Unit Delay", *IEEE Signal Processing Magazine*, Vol. 13, No. 1, p.30-60, 1996.

See Also

Functions
`c2d` | `d2c` | `c2dOptions` | `d2cOptions` | `d2d` | `d2dOptions` | `thiran`

Live Editor Tasks
Convert Model Rate

Related Examples

- “Discretize a Compensator” on page 5-10
- “Improve Accuracy of Discretized System with Time Delay” on page 5-15
- “Convert Discrete-Time System to Continuous Time” on page 5-18

Upsample Discrete-Time System

This example shows how to upsample a system using both the `d2d` and `upsample` commands and compare the results of both to the original system.

Upsampling a system can be useful, for example, when you need to implement a digital controller at a faster rate than you originally designed it for.

Create the discrete-time system

$$G(z) = \frac{z + 0.4}{z - 0.7}$$

with a sample time of 0.3 s.

```
G = tf([1,0.4],[1,-0.7],0.3);
```

Resample the system at 0.1 s using `d2d`.

```
G_d2d = d2d(G,0.1)
```

```
G_d2d =

  z - 0.4769
  ----------
  z - 0.8879

Sample time: 0.1 seconds
Discrete-time transfer function.
```

By default, `d2d` uses the zero-order-hold (ZOH) method to resample the system. The resampled system has the same order as `G`.

Resample the system again at 0.1 s, using `upsample`.

```
G_up = upsample(G,3)
```

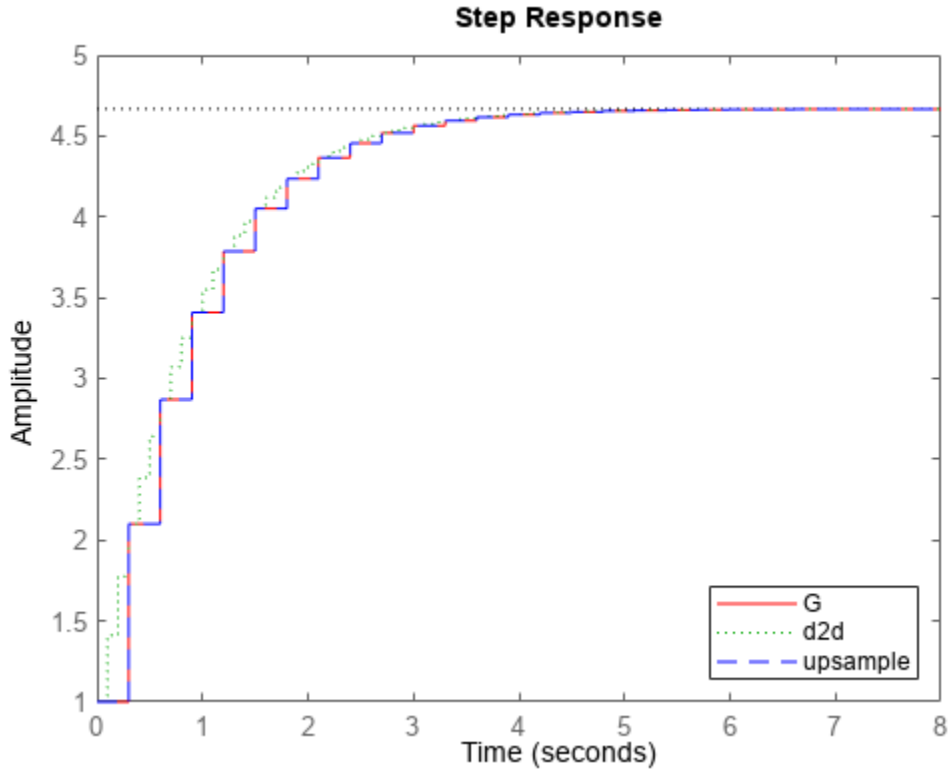
```
G_up =

  z^3 + 0.4
  ---------
  z^3 - 0.7

Sample time: 0.1 seconds
Discrete-time transfer function.
```

The second input, `3`, tells `upsample` to resample `G` at a sample time three times faster than the sample time of `G`. This input to `upsample` must be an integer.

`G_up` has three times as many poles and zeroes as `G`.

Compare the step responses of the original model `G` with the resampled models `G_d2d` and `G_up`.

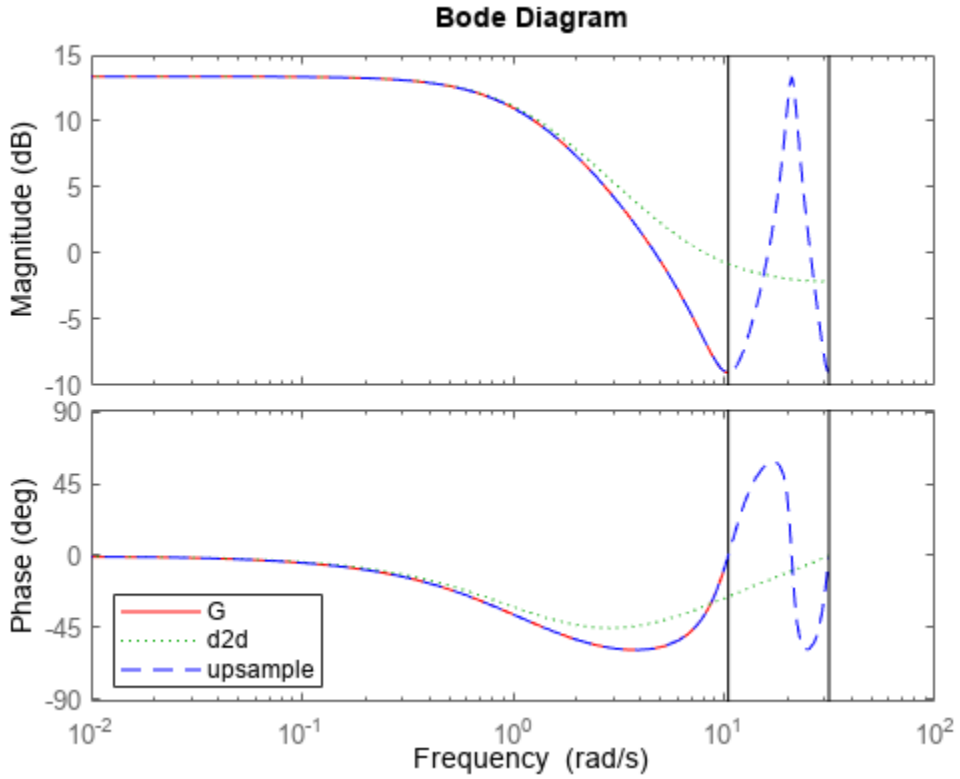
```
step(G,'-r',G_d2d,':g',G_up,'--b')
legend('G','d2d','upsample','Location','SouthEast')
```



The step response of the upsampled model `G_up` matches exactly the step response of the original model `G`. The response of the resampled model `G_d2d` matches only at every third sample.

Compare the frequency response of the original model with the resampled models.

```
bode(G,'-r',G_d2d,':g',G_up,'--b')
legend('G','d2d','upsample','Location','SouthWest')
```



In the frequency domain as well, the model `G_up` created with the `upsample` command matches the original model exactly up to the Nyquist frequency of the original model.

Using `upsample` provides a better match than `d2d` in both the time and frequency domains. However, `upsample` increases the model order, which can be undesirable. Additionally, `upsample` is only available where the original sample time is an integer multiple of the new sample time.

See Also

Functions
`d2d` | `d2dOptions` | `upsample`

Live Editor Tasks
Convert Model Rate

More About

- “Choosing a Resampling Command” on page 5-31

Choosing a Resampling Command

You can resample a discrete-time model using the commands described in the following table.

To...	Use the command...
• Downsample a system. • Upsample a system without any restriction on the new sample time.	`d2d`
Upsample a system with the highest accuracy when: • The new sample time is integer-value-times faster than the sample time of the original model. • Your new model can have more states than the original model.	`upsample`

See Also

Functions
`d2d` | `d2dOptions` | `upsample`

Live Editor Tasks
Convert Model Rate

Related Examples

- “Upsample Discrete-Time System” on page 5-28

Switching Model Representation

This example shows how to switch between the transfer function (TF), zero-pole-gain (ZPK), state-space (SS), and frequency response data (FRD) representations of LTI systems.

Model Type Conversions

You can convert models from one representation to another using the same commands that you use for constructing LTI models (`tf`, `zpk`, `ss`, and `frd`). For example, you can convert the state-space model:

```
sys = ss(-2,1,1,3);
```

to a zero-pole-gain model by typing:

```
zpksys = zpk(sys)
```

```
zpksys =

  3 (s+2.333)
  -----------
     (s+2)

Continuous-time zero/pole/gain model.
```

Similarly, you can calculate the transfer function of `sys` by typing:

```
tf(sys)
```

```
ans =

  3 s + 7
  -------
   s + 2

Continuous-time transfer function.
```

Conversions to FRD require a frequency vector:

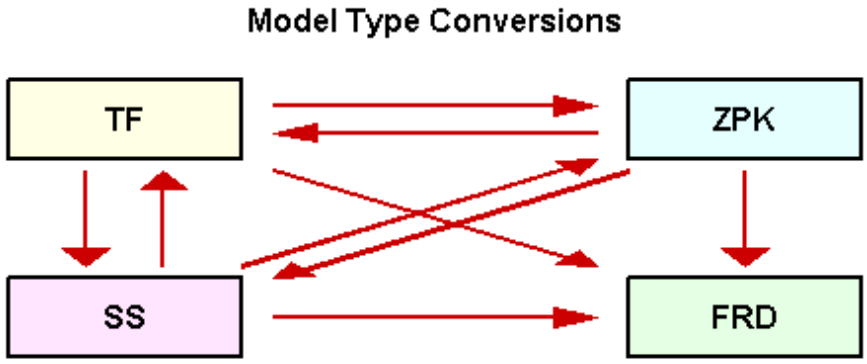
```
f = logspace(-2,2,10);
frdsys = frd(sys,f)
```

```
frdsys =

    Frequency(rad/s)        Response
    ----------------        --------
          0.0100        3.5000 - 0.0025i
          0.0278        3.4999 - 0.0070i
          0.0774        3.4993 - 0.0193i
          0.2154        3.4943 - 0.0532i
          0.5995        3.4588 - 0.1375i
          1.6681        3.2949 - 0.2459i
          4.6416        3.0783 - 0.1817i
         12.9155        3.0117 - 0.0756i
         35.9381        3.0015 - 0.0277i
        100.0000        3.0002 - 0.0100i
```

```
Continuous-time frequency response.
```

Note that FRD models cannot be converted back to the TF, ZPK, or SS representations (such conversion requires the frequency-domain identification tools available in System Identification).

All model type conversion paths are summarized in the diagram below.



Implicit Type Casting

Some commands expect a specific type of LTI model. For convenience, such commands automatically convert incoming LTI models to the appropriate representation. For example, in the sample code:

```
sys = ss(0,1,1,0);
[num,den] = tfdata(sys,'v')
```

```
num = 1×2

     0     1


den = 1×2

     1     0
```

the function `tfdata` automatically converts the state-space model `sys` to an equivalent transfer function to obtain its numerator and denominator data.

Caution About Switching Back and Forth Between Representations

Conversions between the TF, ZPK, and SS representations involve numerical computations and can incur loss of accuracy when overused. Because the SS and FRD representations are best suited for numerical computations, it is good practice to convert all models to SS or FRD and only use the TF and ZPK representations for construction or display purposes.

For example, convert the ZPK model

```
G = zpk([],ones(10,1),1,0.1)
```

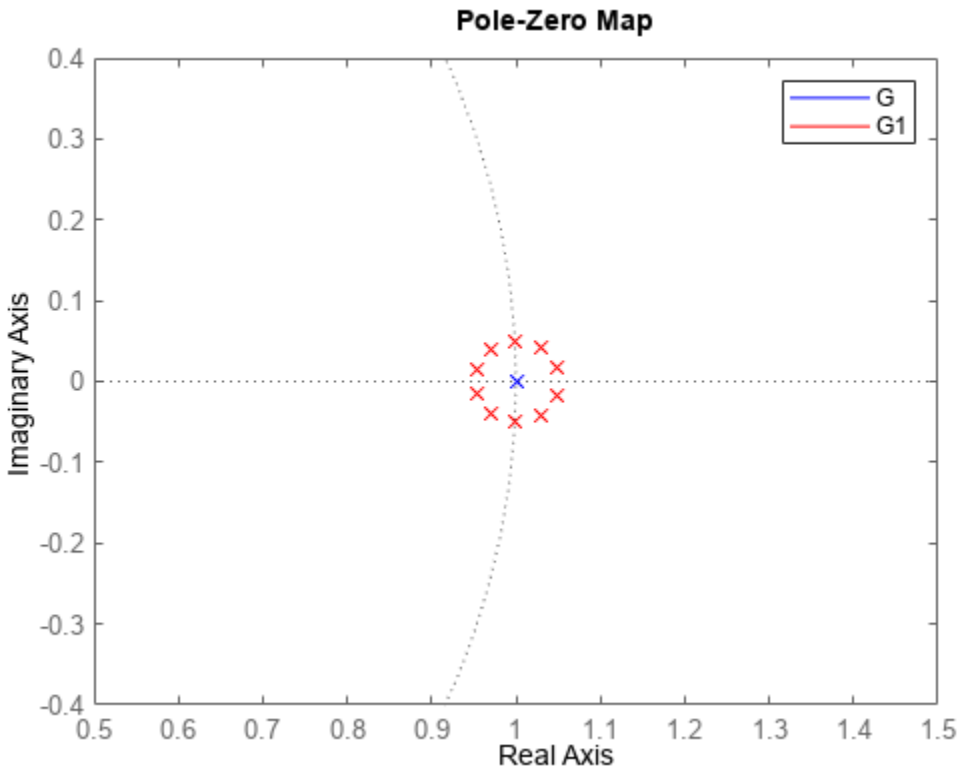
```
G =

     1
  --------
  (z-1)^10

Sample time: 0.1 seconds
Discrete-time zero/pole/gain model.
```

to TF and then back to ZPK:

```
G1 = zpk(tf(G));
```

Now compare the pole locations for `G` and `G1`:

```
G1 = zpk(tf(G));
pzmap(G,'b',G1,'r')
axis([0.5 1.5 -0.4 0.4])
legend('G','G1')
```



Observe how the pole of multiplicity 10 at `z=1` in `G` is replaced by a cluster of poles in `G1`. This occurs because the poles of `G1` are computed as the roots of the polynomial

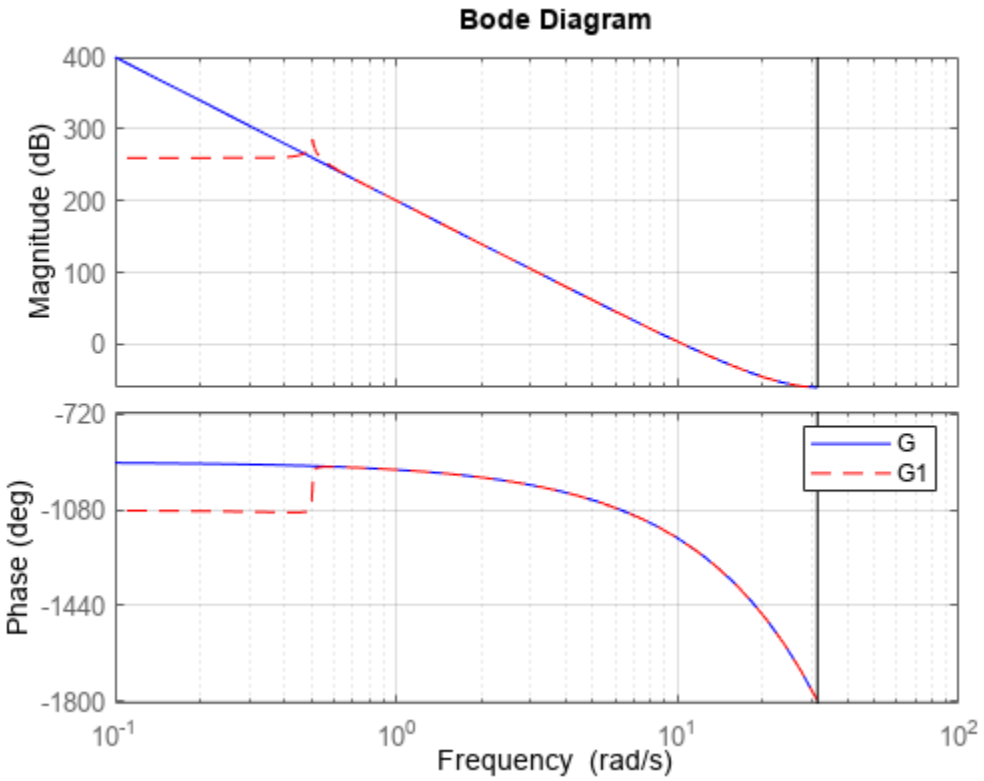
$$(z-1)^{10} = z^{10} - 10z^9 + 45z^8 - 120z^7 + 210z^6 - 252z^5 + 210z^4 - 120z^3 + 45z^2 - 10z + 1$$

and an `o(eps)` error on the last coefficient of this polynomial is enough to move the roots by

$$o(\epsilon^{1/10}) = o(3 \times 10^{-2}).$$

In other words, the transfer function representation is not accurate enough to capture the system behavior near z=1, which is also visible in the Bode plot of `G` vs. `G1`:

```
bode(G,'b',G1,'r--'), grid
legend('G','G1')
```



This illustrates why you should avoid unnecessary model conversions.

Connecting Models

This example shows how to model interconnections of LTI systems, from simple series and parallel connections to complex block diagrams.

Overview

Control System Toolbox™ provides a number of functions to help you build networks of LTI models. These include functions to perform

- Series and parallel connections (`series` and `parallel`)
- Feedback connections (`feedback` and `lft`)
- Input and output concatenations (`[ , ]`, `[ ; ]`, and `append`)
- General block-diagram building (`connect`).

These functions can handle any combination of model representations. For illustration purposes, create the following two SISO transfer function models:

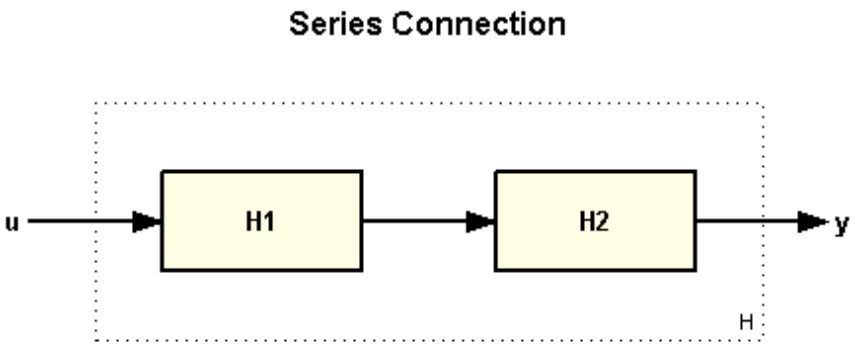
```
H1 = tf(2,[1 3 0])
```

```
H1 =

      2
  ---------
  s^2 + 3 s

Continuous-time transfer function.
```

```
H2 = zpk([],-5,5)
```

```
H2 =

    5
  -----
  (s+5)

Continuous-time zero/pole/gain model.
```

Series Connection



Use the * operator or the `series` function to connect LTI models in series, for example:

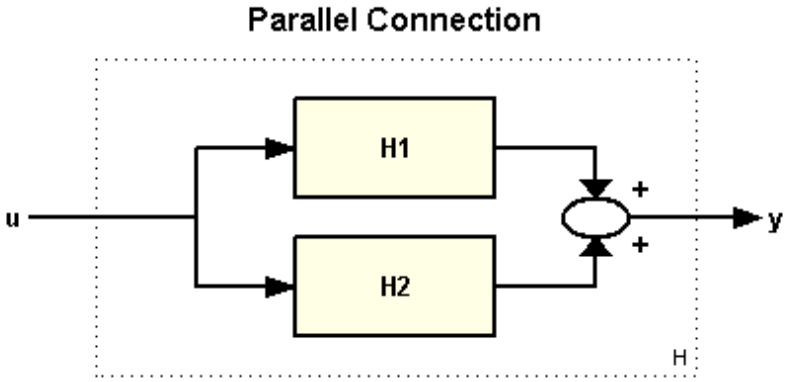
```
H = H2 * H1
```

```
H =

       10
  -------------
  s (s+5) (s+3)

Continuous-time zero/pole/gain model.
```

or equivalently

```
H = series(H1,H2);
```

Parallel Connection



Use the + operator or the `parallel` function to connect LTI models in parallel, for example:

```
H = H1 + H2
```

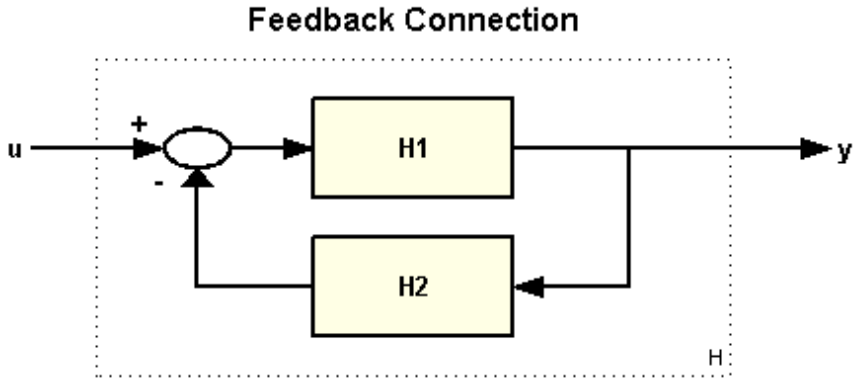
```
H =

  5 (s+2.643) (s+0.7566)
  ----------------------
      s (s+3) (s+5)

Continuous-time zero/pole/gain model.
```

or equivalently

```
H = parallel(H1,H2);
```

Feedback Connections

The standard feedback configuration is shown below:



To build a model of the closed-loop transfer from `u` to `y`, type

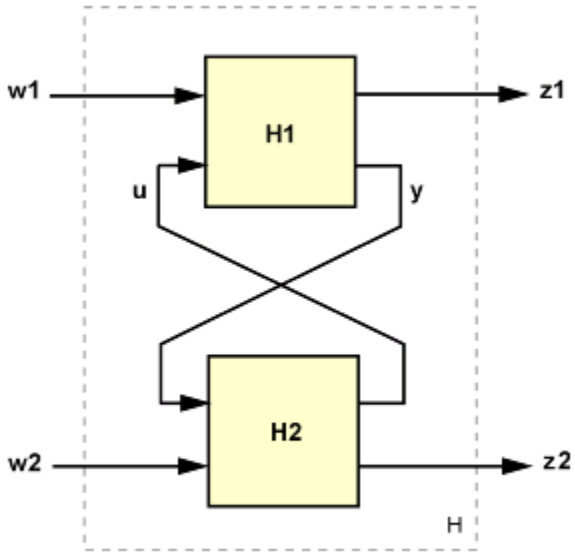
```
H = feedback(H1,H2)
```

```
H =

              2 (s+5)
  --------------------------------
  (s+5.663) (s^2 + 2.337s + 1.766)

Continuous-time zero/pole/gain model.
```

Note that `feedback` assumes negative feedback by default. To apply positive feedback, use the following syntax:

```
H = feedback(H1,H2,+1);
```

You can also use the `lft` function to build the more general feedback interconnection sketched below.



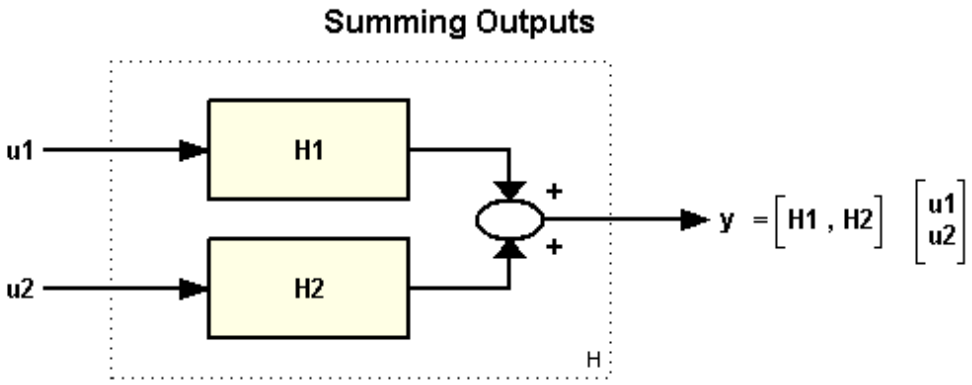
Concatenating Inputs and Outputs

You can concatenate the inputs of the two models `H1` and `H2` by typing

```
H = [ H1 , H2 ]
```

```
H =

  From input 1 to output:
     2
  -------
  s (s+3)

  From input 2 to output:
    5
  -----
  (s+5)

Continuous-time zero/pole/gain model.
```

The resulting model has two inputs and corresponds to the interconnection:

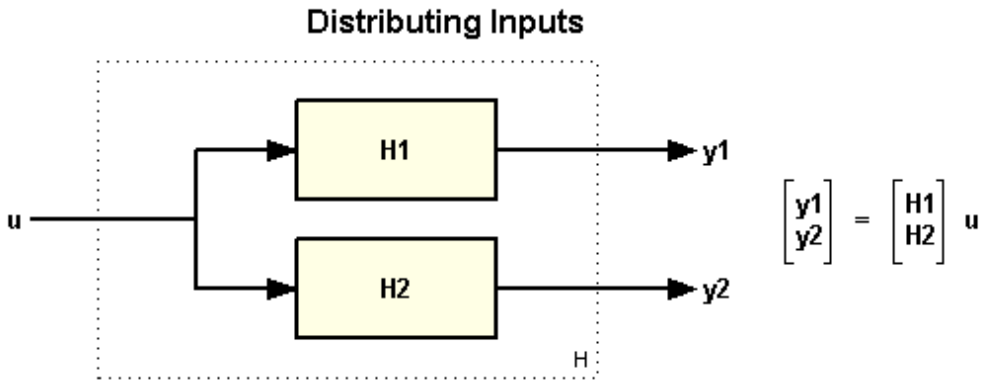


Similarly, you can concatenate the outputs of `H1` and `H2` by typing

```
H = [ H1 ; H2 ]
```

```
H =

  From input to output...
          2
   1:  -------
       s (s+3)

         5
   2:  -----
       (s+5)

Continuous-time zero/pole/gain model.
```

The resulting model `H` has two outputs and one input and corresponds to the following block diagram:

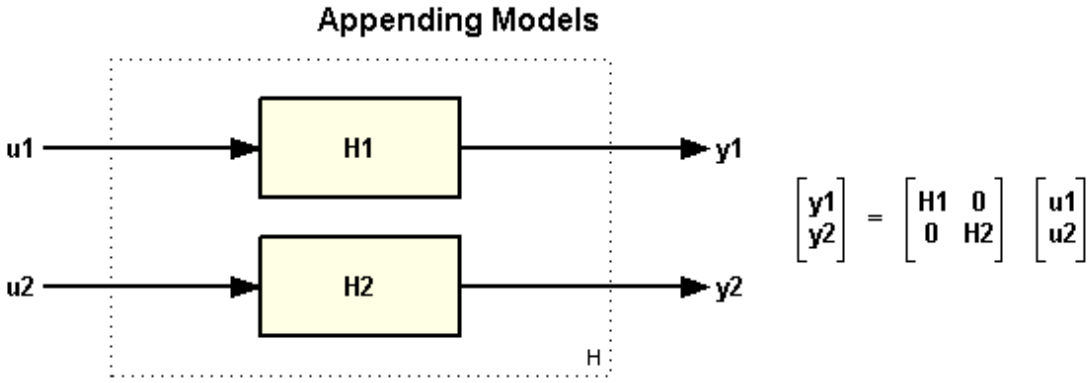


Finally, you can append the inputs and outputs of two models using:

```
H = append(H1,H2)
```

```
H =

  From input 1 to output...
          2
   1:  -------
       s (s+3)

   2:  0

  From input 2 to output...
   1:  0

         5
   2:  -----
       (s+5)

Continuous-time zero/pole/gain model.
```

The resulting model `H` has two inputs and two outputs and corresponds to the block diagram:



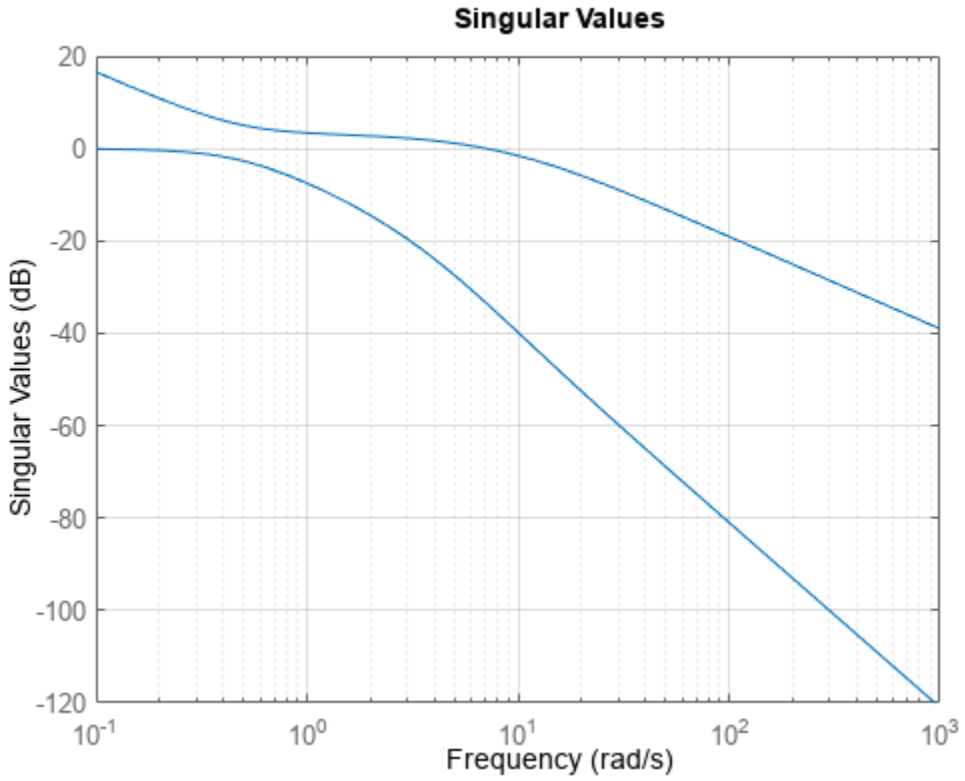
You can use concatenation to build MIMO models from elementary SISO models, for example:

```
H = [H1 , -tf(10,[1 10]) ; 0 , H2 ]
```

```
H =

  From input 1 to output...
          2
   1:  -------
       s (s+3)

   2:  0

  From input 2 to output...
        -10
```

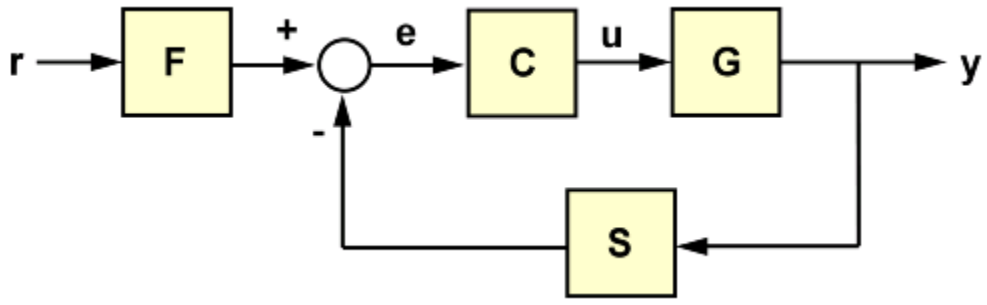
```
  1:  ------
      (s+10)

        5
  2:  -----
      (s+5)

Continuous-time zero/pole/gain model.
```

```
sigma(H), grid
```



Building Models from Block Diagrams

You can use combinations of the functions and operations introduced so far to construct models of simple block diagrams. For example, consider the following block diagram:

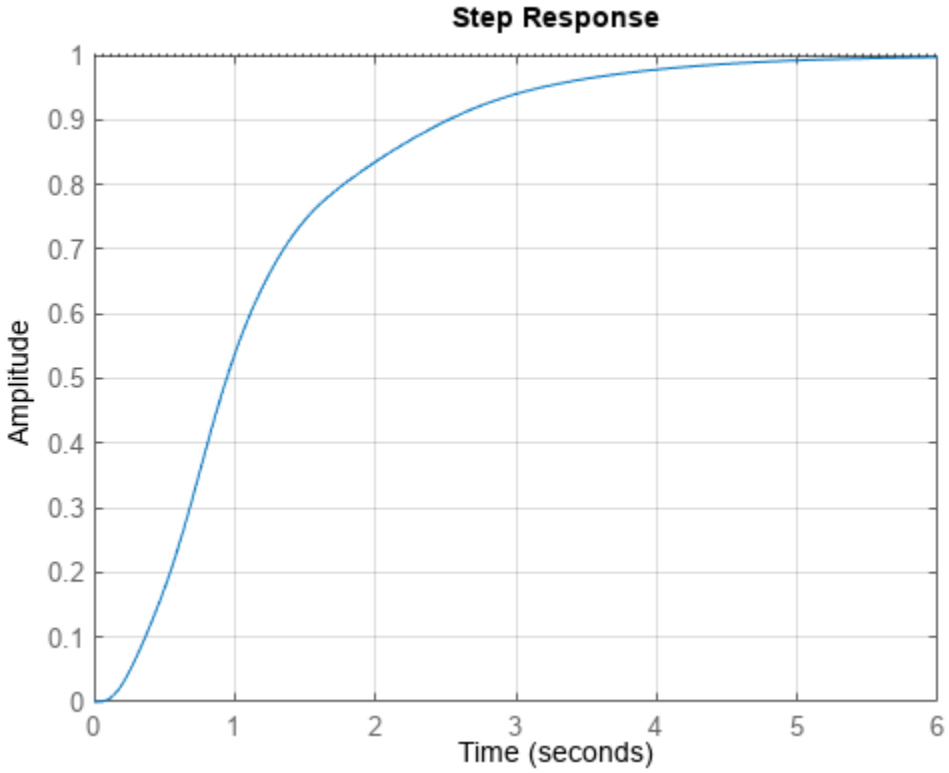


with the following data for the blocks `F`, `C`, `G`, `S`:

```
s = tf('s');
F = 1/(s+1);
G = 100/(s^2+5*s+100);
C = 20*(s^2+s+60)/s/(s^2+40*s+400);
S = 10/(s+10);
```

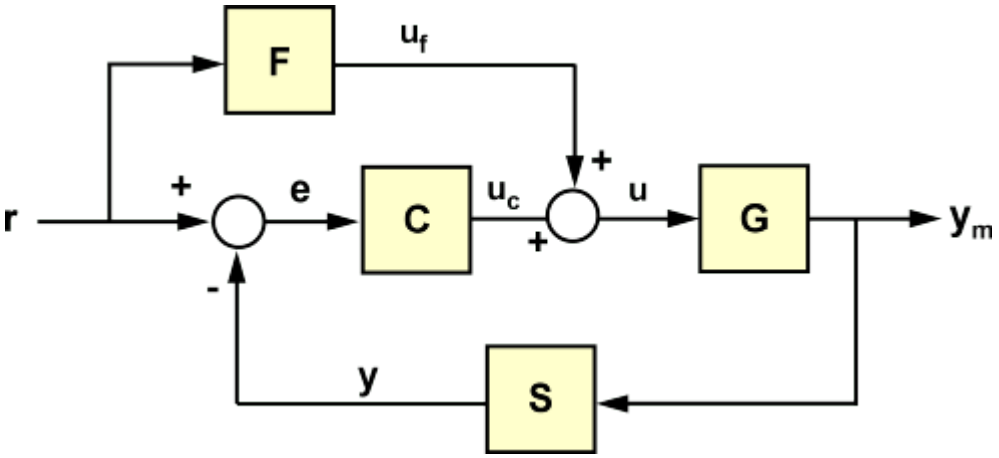
You can compute the closed-loop transfer `T` from `r` to `y` as

```
T = F * feedback(G*C,S);
step(T), grid
```



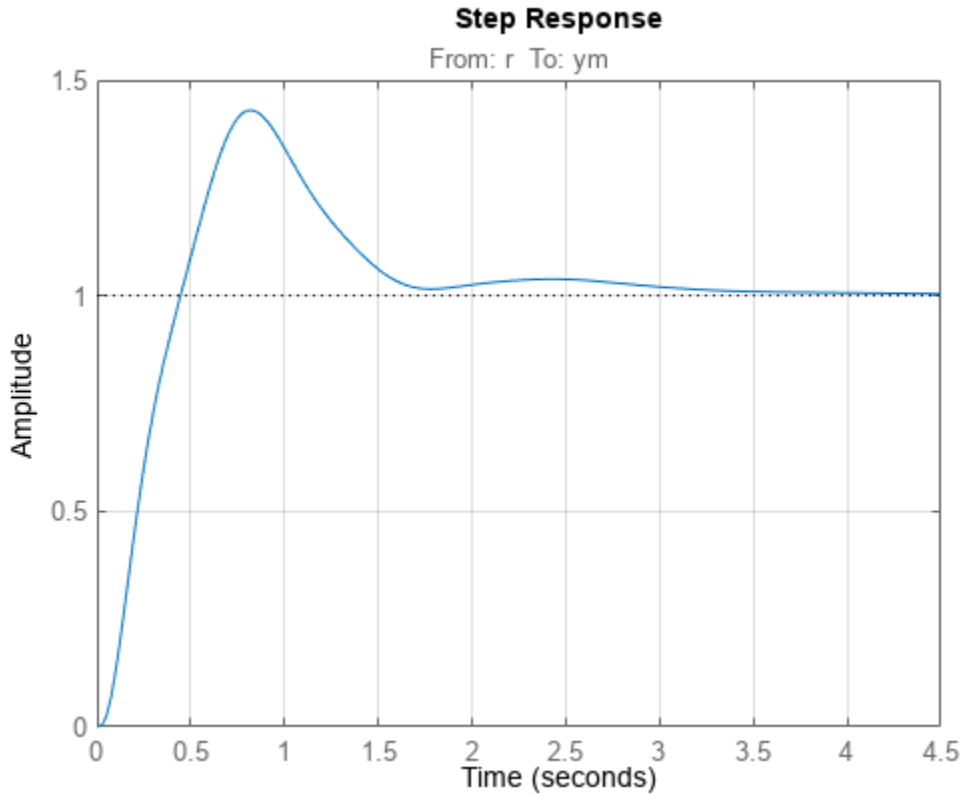
For more complicated block diagrams, the `connect` function provides a systematic and simple way to wire blocks together. To use `connect`, follow these steps:

- Define all blocks in the diagram, including summation blocks
- Name all block input and output channels
- Select the block diagram I/Os from the list of block I/Os.



For the block diagram above, these steps amount to:

```
Sum1 = sumblk('e = r - y');
Sum2 = sumblk('u = uC + uF');

% Define block I/Os ("u" and "y" are shorthand for "InputName" and "OutputName")
F.u = 'r';   F.y = 'uF';
C.u = 'e';   C.y = 'uC';
G.u = 'u';   G.y = 'ym';
S.u = 'ym';  S.y = 'y';

% Compute transfer r -> ym
T = connect(F,C,G,S,Sum1,Sum2,'r','ym');
step(T), grid
```



Precedence Rules

When connecting models of different types, the resulting model type is determined by the precedence rule

```
FRD > SS > ZPK > TF > PID
```

This rule states that FRD has highest precedence, followed by SS, ZPK, TF, and PID has the lowest precedence. For example, in the series connection:

```
H1 = ss(-1,2,3,0);
H2 = tf(1,[1 0]);
H = H2 * H1;
```

`H2` is automatically converted to the state-space representation and the result `H` is a state-space model:

```
class(H)

ans =
'ss'
```

Because the SS and FRD representations are best suited for system interconnections, it is recommended that you convert at least one of the models to SS or FRD to ensure that all computations are performed using one of these two representations. One exception is when using

`connect` which automatically performs such conversion and always returns a state-space or FRD model of the block diagram.

Discretizing and Resampling Models

This example shows how to use the commands for continuous/discrete, discrete/continuous, and discrete/discrete conversions.

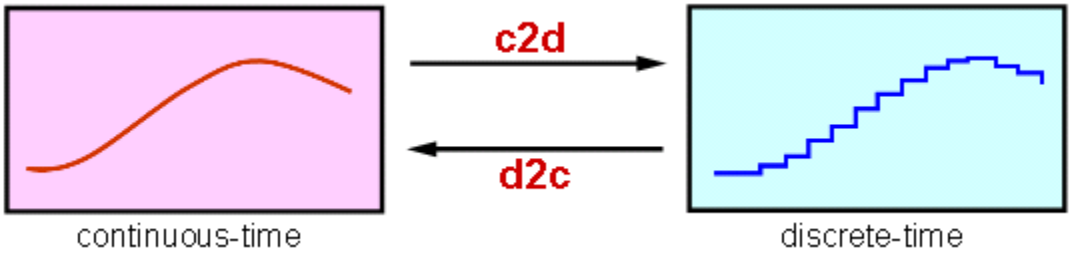
Related Commands

Control System Toolbox™ offers extensive support for discretization and resampling of linear systems including:

- `c2d` discretizes continuous-time models
- `d2c` compute continuous-time extensions of discrete-time models
- `d2d` resamples discrete-time models.

Several algorithms are available to perform these operations, including:

- Zero-order hold
- First-order hold
- Impulse invariant
- Tustin
- Matched poles/zeros.



Continuous/Discrete Conversion

For example, consider the second-order system with delay:

$$G(s) = e^{-s}\frac{s-2}{s^2+3s+20}$$

To compute its zero-order hold (ZOH) discretization with sampling rate of 10 Hz, type

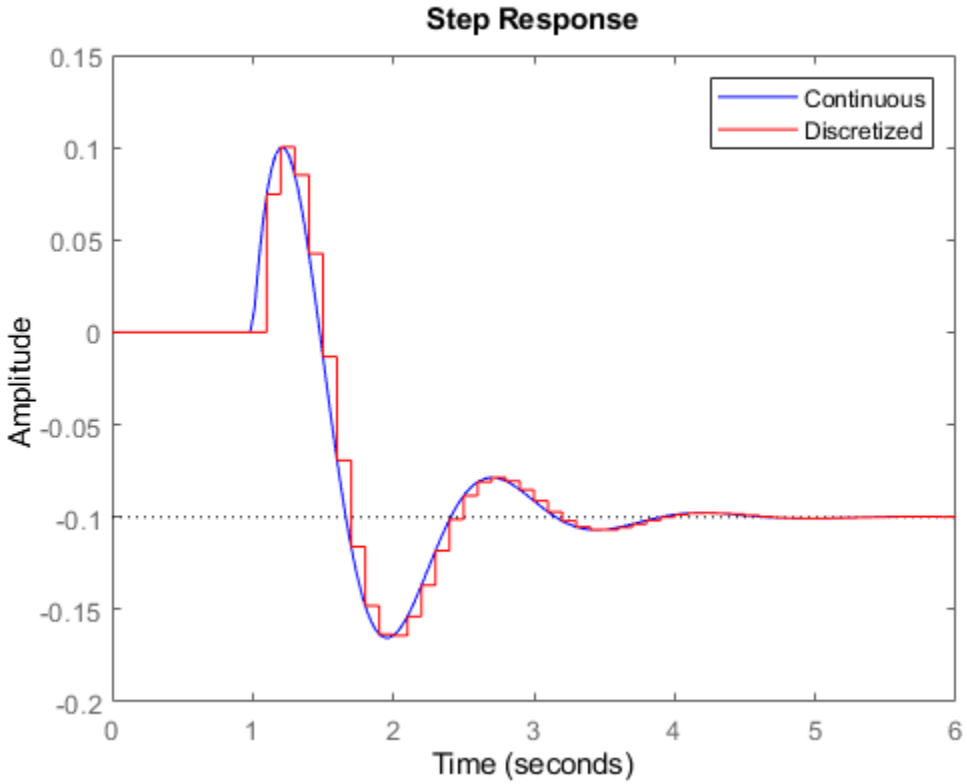
```
G = tf([1 -2],[1 3 20],'inputdelay',1);
Ts = 0.1;   % sampling interval
Gd = c2d(G,Ts)
```

```
Gd =

              0.07462 z - 0.09162
  z^(-10) * ----------------------
            z^2 - 1.571 z + 0.7408
```

```
Sample time: 0.1 seconds
Discrete-time transfer function.
```

Compare the continuous and discrete step responses:

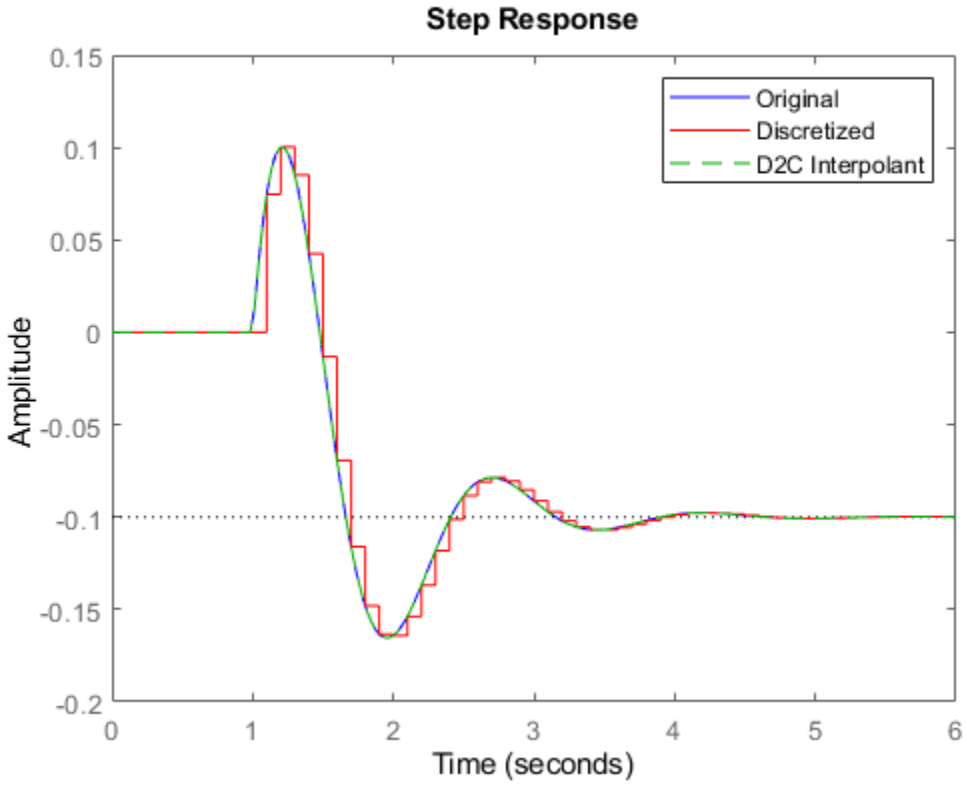
```
step(G,'b',Gd,'r')
legend('Continuous','Discretized')
```



Discrete/Continuous Conversion

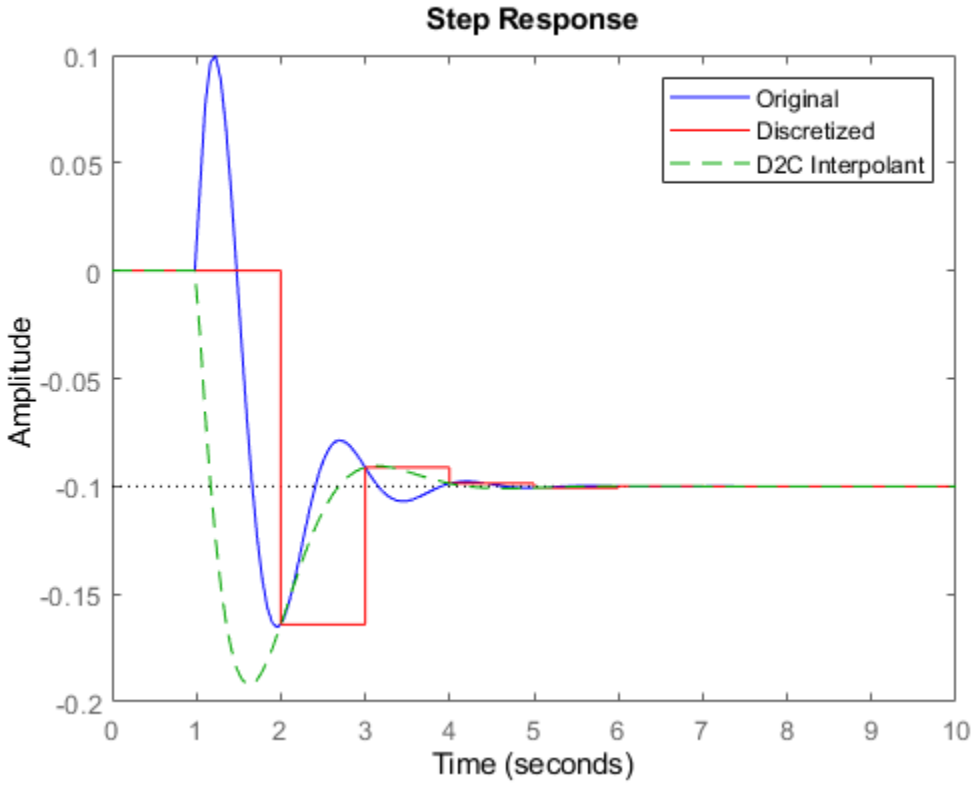
Conversely, you can use `d2c` to compute a continuous-time "interpolant" for a given discrete-time system. Starting with the discretization `Gd` computed above, convert it back to continuous and compare with the original model `G`:

```
Gc = d2c(Gd);
step(G,'b',Gd,'r',Gc,'g--')
legend('Original','Discretized','D2C Interpolant')
```



The two continuous-time responses match perfectly. You may not always obtain a perfect match especially when your sampling interval `Ts` is too large and aliasing occurs during discretization:

```
Ts = 1;  % 10 times larger than previously
Hd = c2d(G,Ts);
Hc = d2c(Hd);
step(G,'b',Hd,'r',Hc,'g--',10)
legend('Original','Discretized','D2C Interpolant')
```



Resampling of Discrete-Time Systems

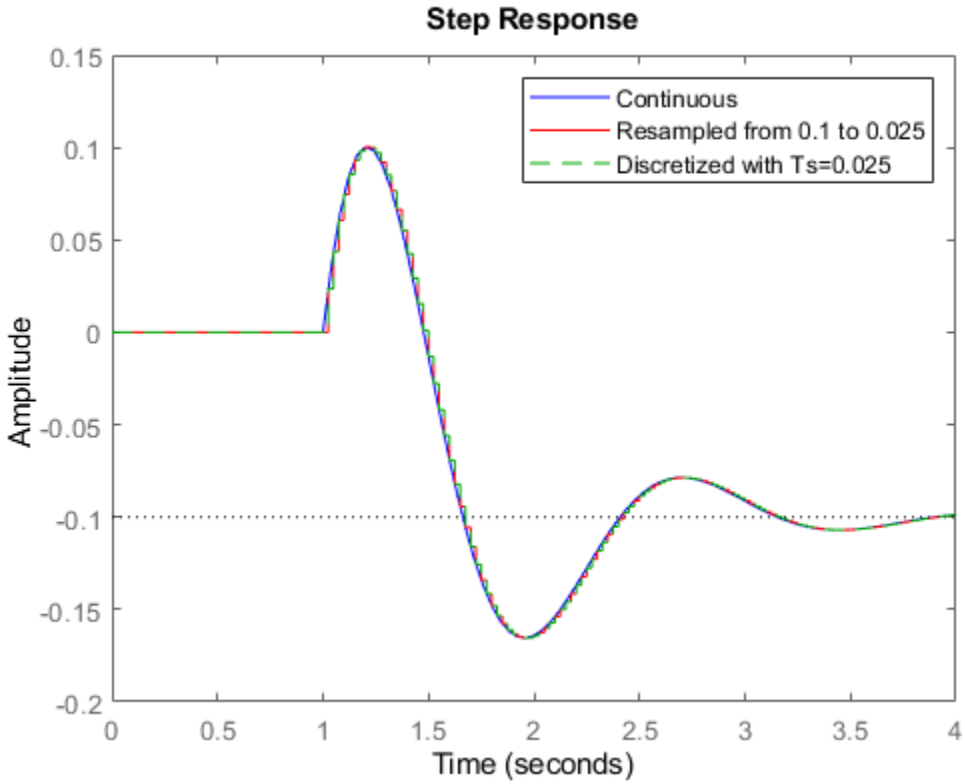
Resampling consists of changing the sampling interval of a discrete-time system. This operation is performed with `d2d`. For example, consider the 10 Hz discretization `Gd` of our original continuous-time model `G`. You can resample it at 40 Hz using:

```
Gr = d2d(Gd,0.025)
```

```
Gr =

              0.02343 z - 0.02463
  z^(-40) * ----------------------
            z^2 - 1.916 z + 0.9277

Sample time: 0.025 seconds
Discrete-time transfer function.
```

Compare this with a direct discretization at 40 Hz:

```
step(G,'b',Gr,'r',c2d(G,0.025),'g--',4)
legend('Continuous','Resampled from 0.1 to 0.025','Discretized with Ts=0.025')
```



Notice that both approaches lead to the same answer.

Which Algorithm and Sampling Rate to Choose?

See the example entitled “Discretizing a Notch Filter” on page 5-52 for more details on how the choice of algorithm and sampling rate affect the discretization accuracy.

Discretizing a Notch Filter

This example shows the comparison of several techniques for discretizing a notch filter. While control system components are often designed in continuous time, they must generally be discretized for implementation on digital computers and embedded processors.

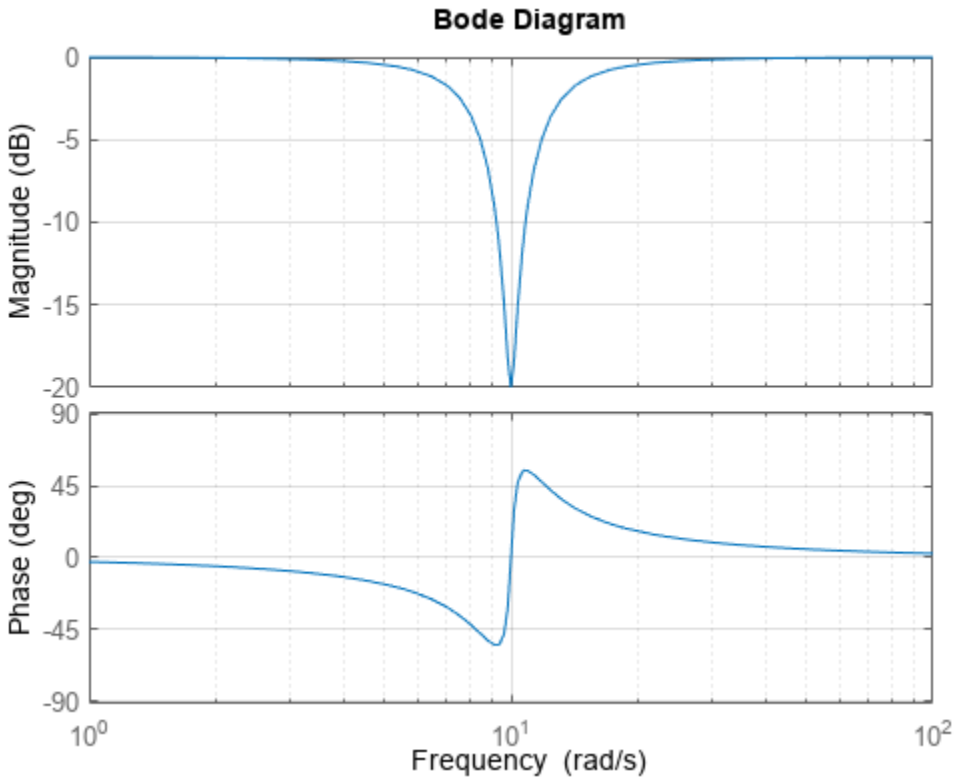
Continuous-Time Notch Filter

Notch filters are designed to reject signal content at a specific frequency by sharply attenuating the gain at that frequency. For this example we consider the following notch filter:

$$H(s) = \frac{s^2 + 0.5s + 100}{s^2 + 5s + 100}$$

You can plot the frequency response of this filter with the `bode` command:

```
H = tf([1 0.5 100],[1 5 100]);
bode(H), grid
```



This notch filter provides a 20dB attenuation at the frequency w = 10 rad/s.

Choosing the Discretization Method

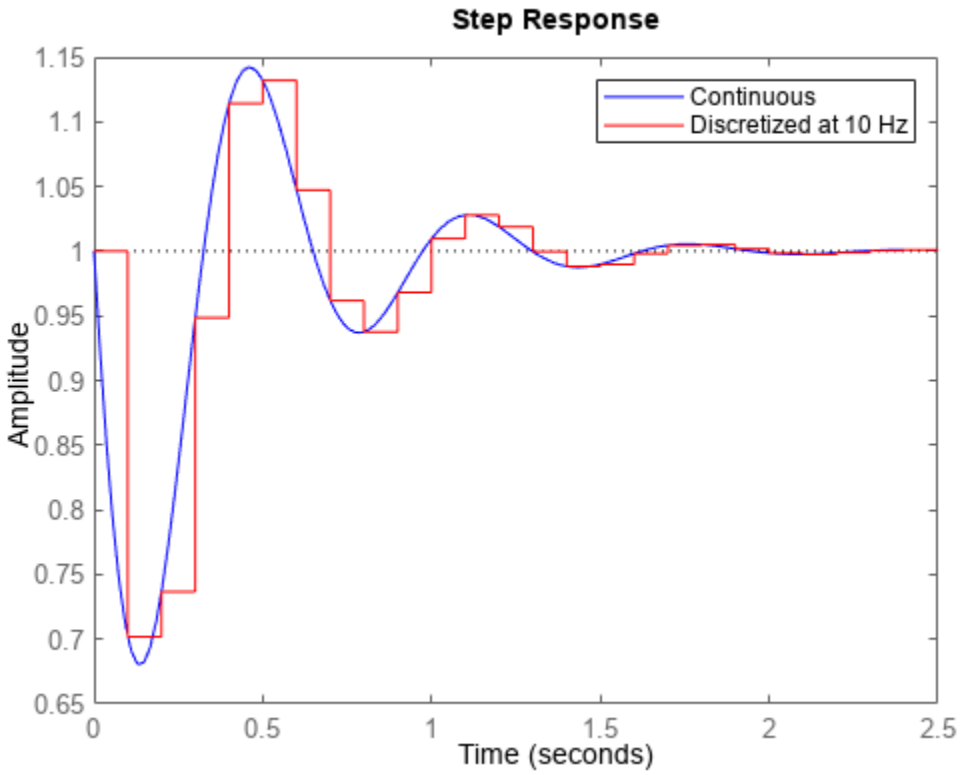
You can discretize a continuous-time system with the `c2d` command. Several discretization algorithms are supported by Control System Toolbox™, including:

- Zero-order hold
- First-order hold
- Impulse invariant
- Tustin (bilinear approximation)
- Tustin with frequency prewarping
- Matched poles and zeros

Which method to choose depends on the application and requirements.

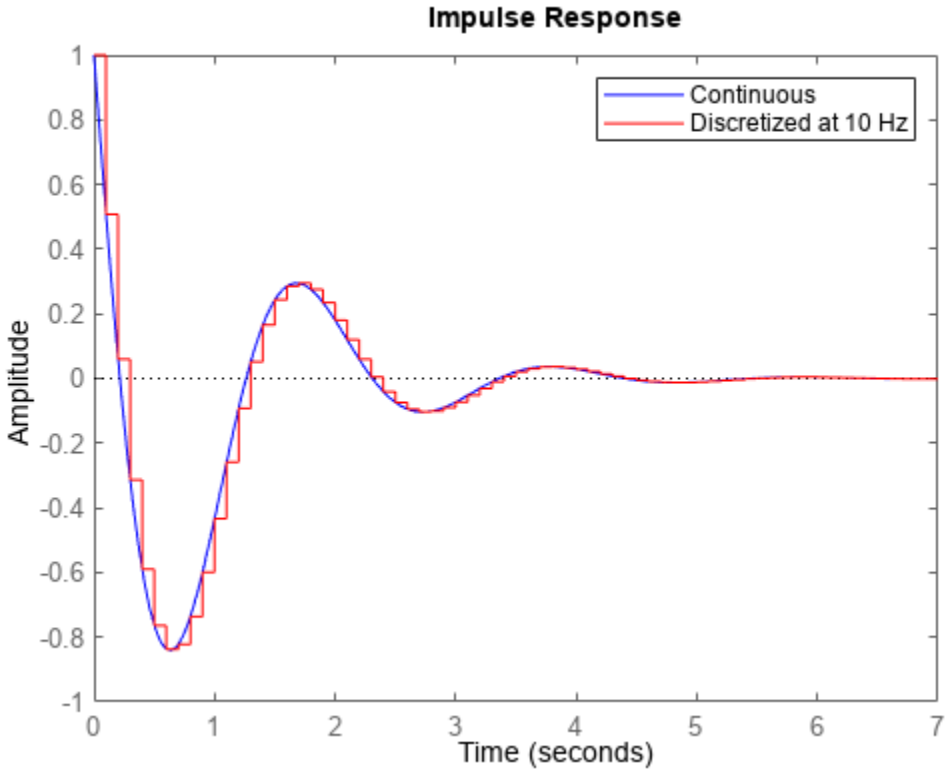
The zero- and first-order hold methods and the impulse-invariant method are well-suited for discrete approximations in the time domain. For example, the step response of the ZOH discretization matches the continuous-time step response at each time step (independently of the sampling rate):

```
Ts = 0.1;
Hdz = c2d(H,Ts,'zoh');
step(H,'b',Hdz,'r'),
legend('Continuous','Discretized at 10 Hz')
```



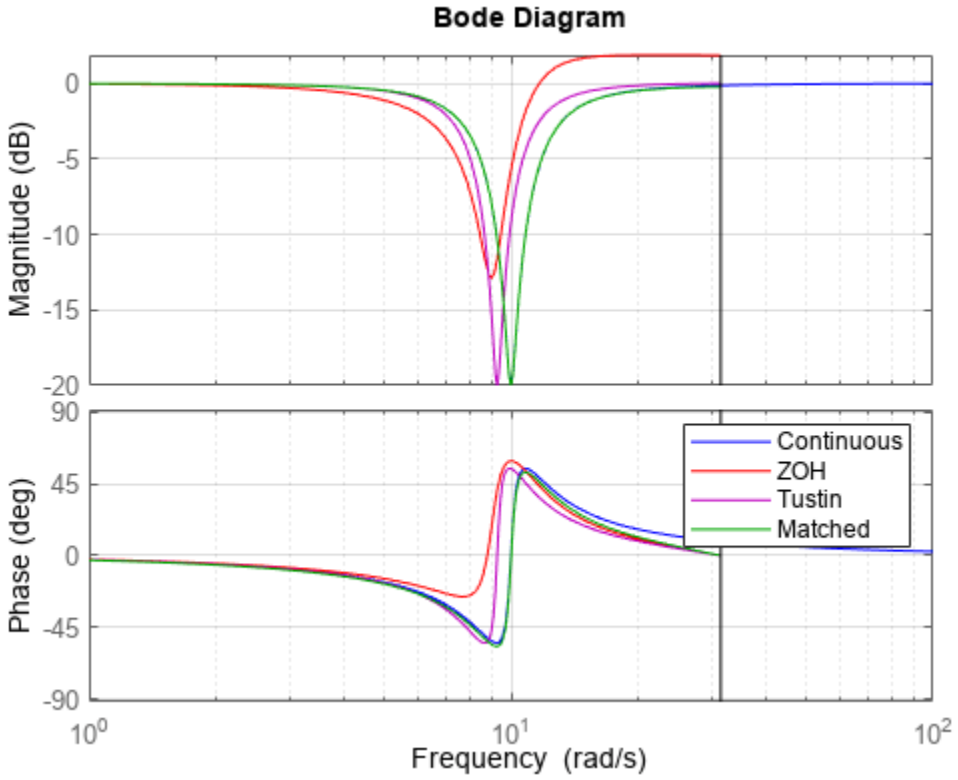
Similarly, the impulse-invariant discretization has the same impulse response as the original system:

```
G = tf([1 -3],[1 2 10]);
Gd = c2d(G,Ts,'imp');
impulse(G,'b',Gd,'r')
legend('Continuous','Discretized at 10 Hz')
```



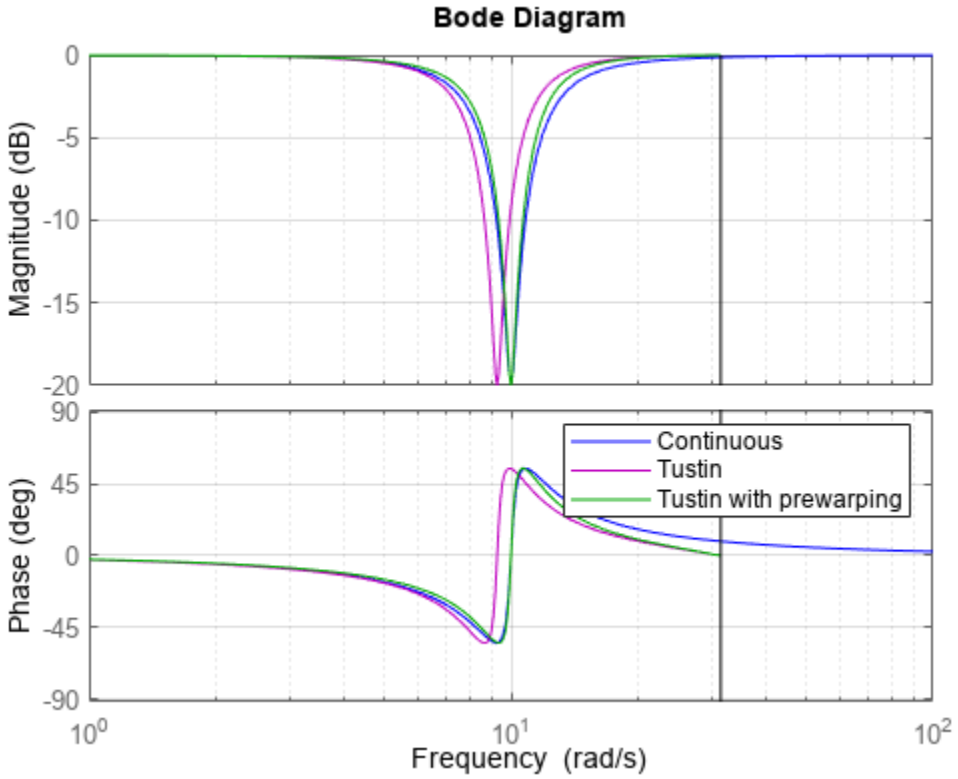
By contrast, the Tustin and Matched methods tend to perform better in the frequency domain because they introduce less gain and phase distortion near the Nyquist frequency. For example, compare the Bode responses of the continuous-time notch filter and its discretizations using the ZOH, Tustin, and Matched algorithms:

```
Hdt = c2d(H,Ts,'tustin');
Hdm = c2d(H,Ts,'match');
bode(H,'b',Hdz,'r',Hdt,'m',Hdm,'g',{1 100}), grid
legend('Continuous','ZOH','Tustin','Matched')
```



This comparison indicates that the Matched method provides the most accurate frequency-domain approximation of the notch filter. However, you can further improve the accuracy of the Tustin algorithm by specifying a prewarping frequency equal to the notch frequency. This ensures accurate match near w = 10 rad/s:

```
Hdp = c2d(H,Ts,'prewarp',10);
bode(H,'b',Hdt,'m',Hdp,'g',{1 100}), grid
legend('Continuous','Tustin','Tustin with prewarping')
```

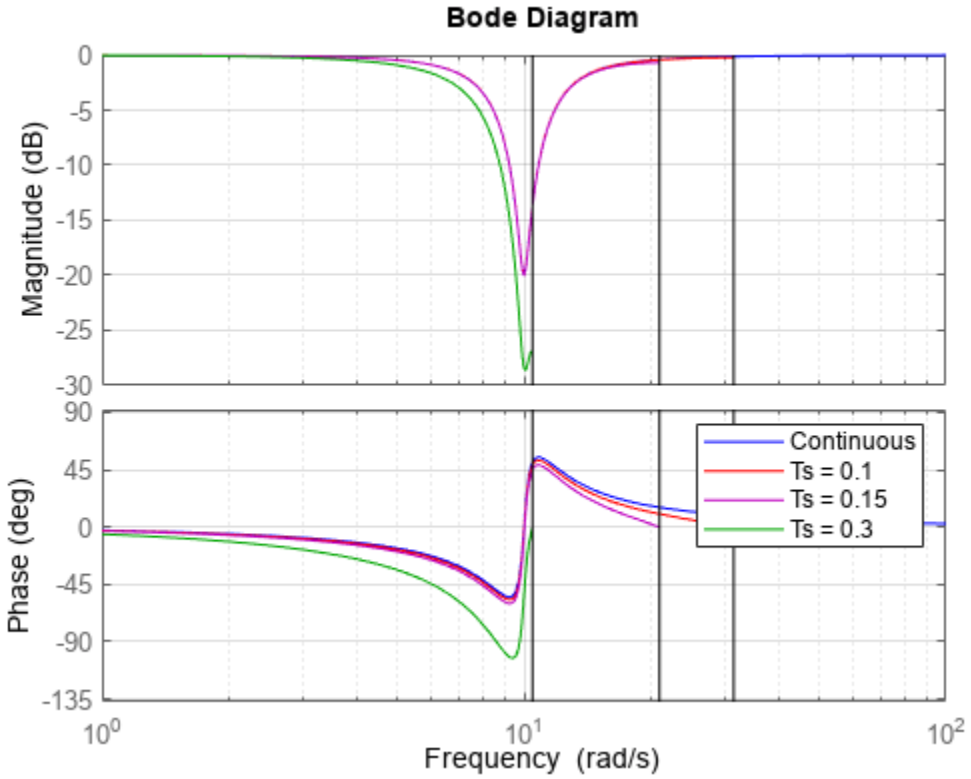


Choosing the Sampling Rate

The higher the sampling rate, the closer the match between the continuous and discretized responses. But how small can the sampling rate be, or equivalently, how large can the sampling interval be? As a rule of thumb, if you want the continuous and discretized models to match closely up to some frequency `wm`, make sure that the Nyquist frequency (sampling rate times Pi) is at least twice `wm`. For the notch filter, you need to preserve the shape near 10 rad/s, so the Nyquist frequency should be beyond 20 rad/s, which gives a sampling period of at most pi/20 = 0.16 s.

To confirm this choice, compare the matched discretizations with sampling period 0.1, 0.15, and 0.3:

```
Hd1 = c2d(H,0.1,'m');
Hd2 = c2d(H,0.15,'m');
Hd3 = c2d(H,0.3,'m');
bode(H,'b',Hd1,'r',Hd2,'m',Hd3,'g',{1 100}), grid
legend('Continuous','Ts = 0.1','Ts = 0.15','Ts = 0.3')
```



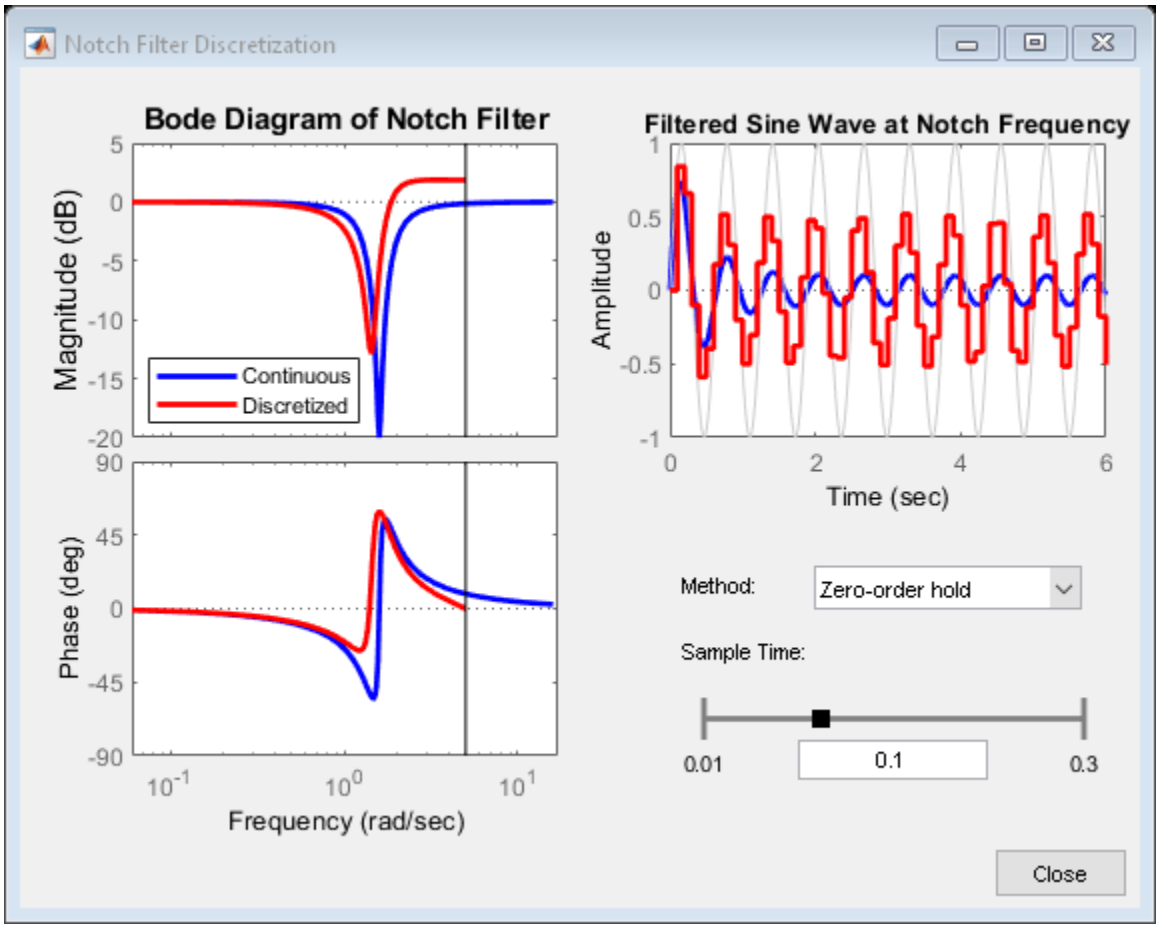
As predicted, the discretization remains fairly accurate for `Ts < 0.16` but starts breaking down for larger sampling intervals.

Interactive GUI

Click on the link below to launch an interactive GUI that further shows how the discretized notch filter is affected by the choice of discretization algorithm and sampling rate.

Open the Notch Discretization GUI

```
notch_gui
```


Notch Filter Discretization
Bode Diagram of Notch Filter
Magnitude (dB)
Phase (deg)
Frequency (rad/sec)
Continuous
Discretized
Filtered Sine Wave at Notch Frequency
Amplitude
Time (sec)
Method:
Zero-order hold
Sample Time:
0.01
0.1
0.3
Close

Scaling State-Space Models to Maximize Accuracy

This example shows that proper scaling of state-space models can be critical for accuracy and provides an overview of automatic and manual rescaling tools.

Why Scaling Matters

A state-space model is well scaled when the entries of the A,B,C matrices are homogeneous in magnitude and the model characteristics are insensitive to small perturbations of A,B,C (in comparison to their norms). By contrast, a model is poorly scaled when A,B,C have both small and large entries and the model characteristics are sensitive to the small entries.

Mixing disparate time scales or unit scales can give rise to badly scaled models. Working with such models can lead to severe loss of accuracy and puzzling results. To prevent these problems, it is often necessary to rescale the state vector, that is, multiply each state by some scaling factor to reduce the numerical range and sensitivity of the model.

Poor scaling can impact the accuracy of most frequency-domain computations. State-of-the-art algorithms heavily rely on orthogonal state coordinate transformations, and such transformations introduce errors of order `eps` (the machine precision) times the norms of A,B,C. While such errors are usually negligible, they can become dominant when A,B,C are poorly scaled. To see this phenomenon, load the following example:

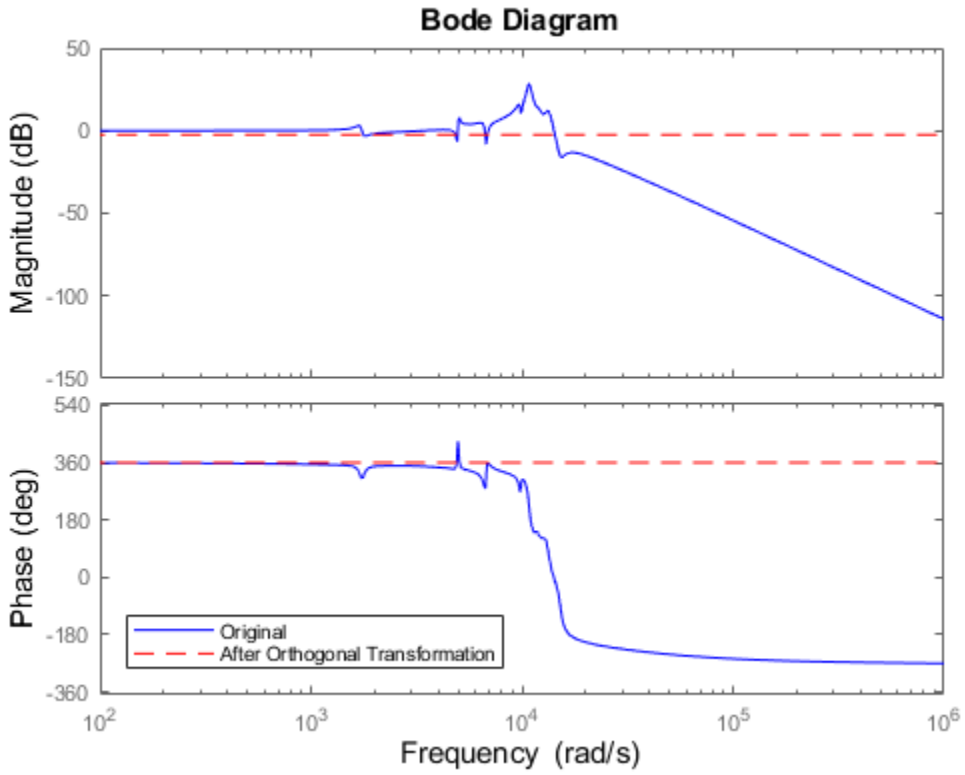
```
load numdemo anil
```

The `anil` model is a state-space realization of a 20-th order transfer function with most of its dynamics between 1e3 and 1e5 rad/s. The coefficients of the A matrix range between 1 and 1e80 in magnitude. To simulate the effect of orthogonal transformations applied to this model, generate a random orthogonal matrix `U` and use `ss2ss` to perform the corresponding change of state coordinates:

```
[U,junk] = qr(randn(20));
anil2 = ss2ss(anil,U);   % perform state coordinate transformation U
```

Mathematically, `anil` and `anil2` have the same frequency response. Numerically, however, the Bode responses of `anil` and `anil2` are very different:

```
bode(anil,'b',anil2,'r--')
legend('Original','After Orthogonal Transformation','Location','southwest')
```



This example shows that for poorly scaled models, even orthogonal transformations can be unsafe and destroy accuracy.

Sensitivity-Minimizing Scaling

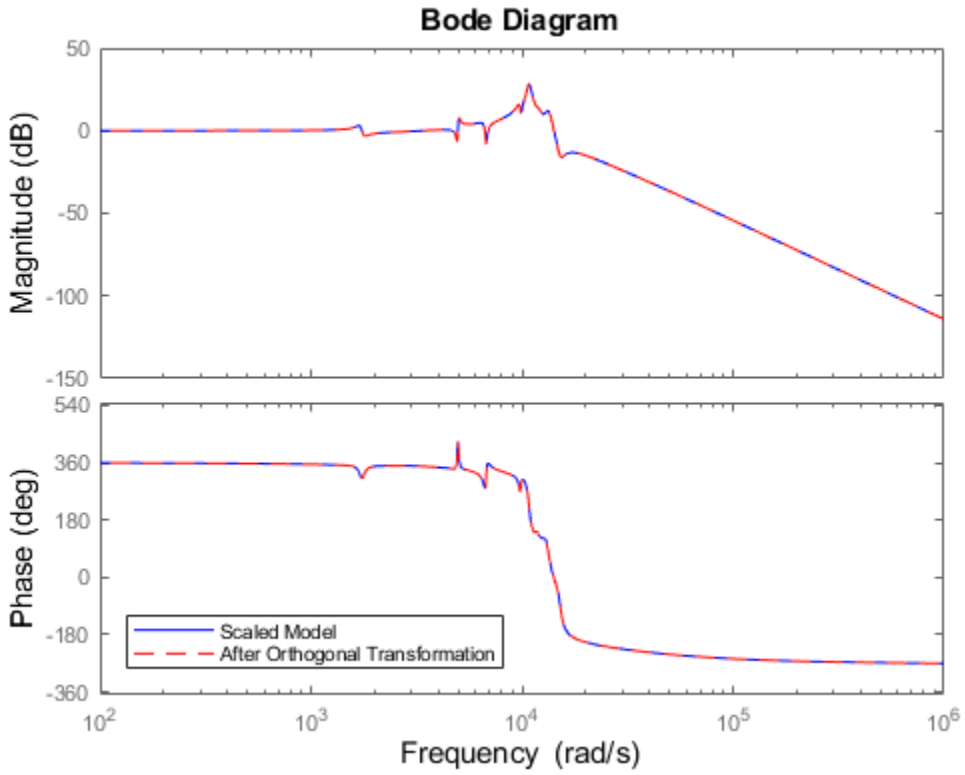
For simple models derived from physics, you can avoid scaling issues by carefully selecting units to reduce the spread between small and large coefficients. For more complex models, however, finding a proper scaling is challenging. Ad hoc schemes such as balancing of the A matrix (see `balance`) are often useful but sometimes harmful.

The Control System Toolbox™ software provides advanced scaling algorithms that minimize the model sensitivity to small perturbations of A,B,C proportional to their norms. This helps maximize accuracy of the computed frequency response, ZPK representation, etc. The `prescale` command is the gateway to these scaling algorithms. For example, you can use `prescale` to scale the `anil` model used above:

```
Scaled_anil = prescale(anil);
```

The coefficients of the A matrix now range from 1e3 to 3e7 instead of 1 to 1e80. Apply the orthogonal transformation `U` to the scaled model and compare the Bode responses:

```
Scaled_anil2 = ss2ss(Scaled_anil,U);
bode(Scaled_anil,'b',Scaled_anil2,'r--')
legend('Scaled Model','After Orthogonal Transformation','Location','southwest')
```



The Bode responses match closely now. Scaling has made orthogonal transformations safe again and you can expect good accuracy from computations involving this scaled model.

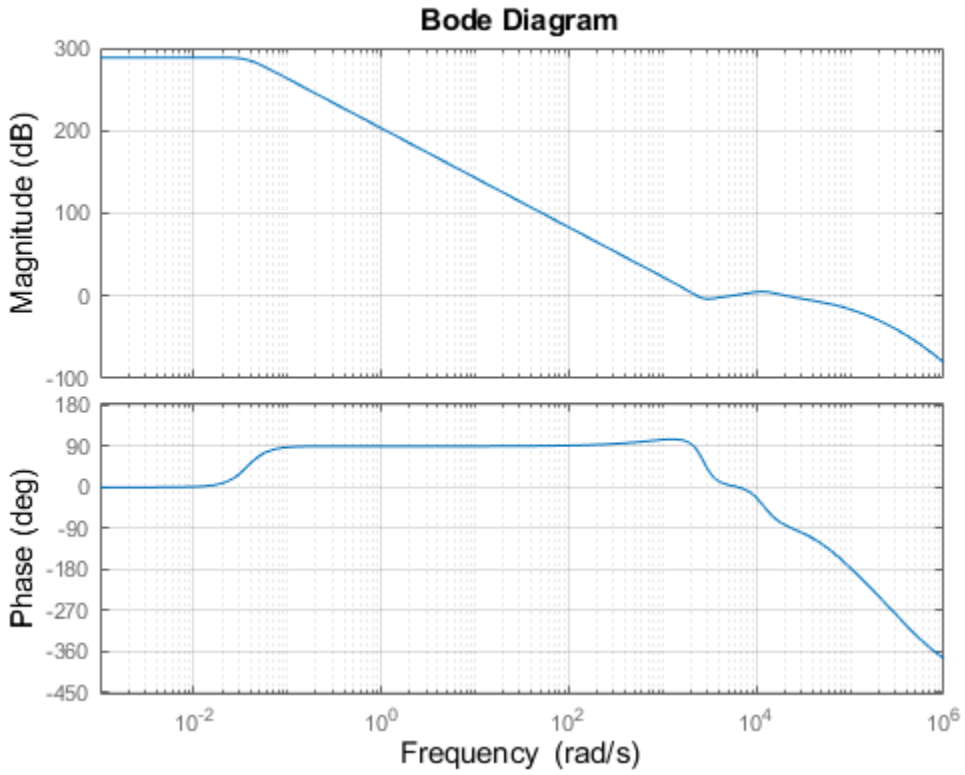
Automatic Scaling

Most algorithms in the Control System Toolbox software automatically rescale state-space models to prevent catastrophic loss of accuracy. As a result, you are mostly insulated from scaling issues. For example, the `bode` command automatically scales incoming models so that it can safely perform orthogonal transformations to speed up the frequency response computation. Therefore, there is no need to use `prescale` before `bode` unless you want detailed information about the relative accuracy of the computed Bode response.

Manual Scaling

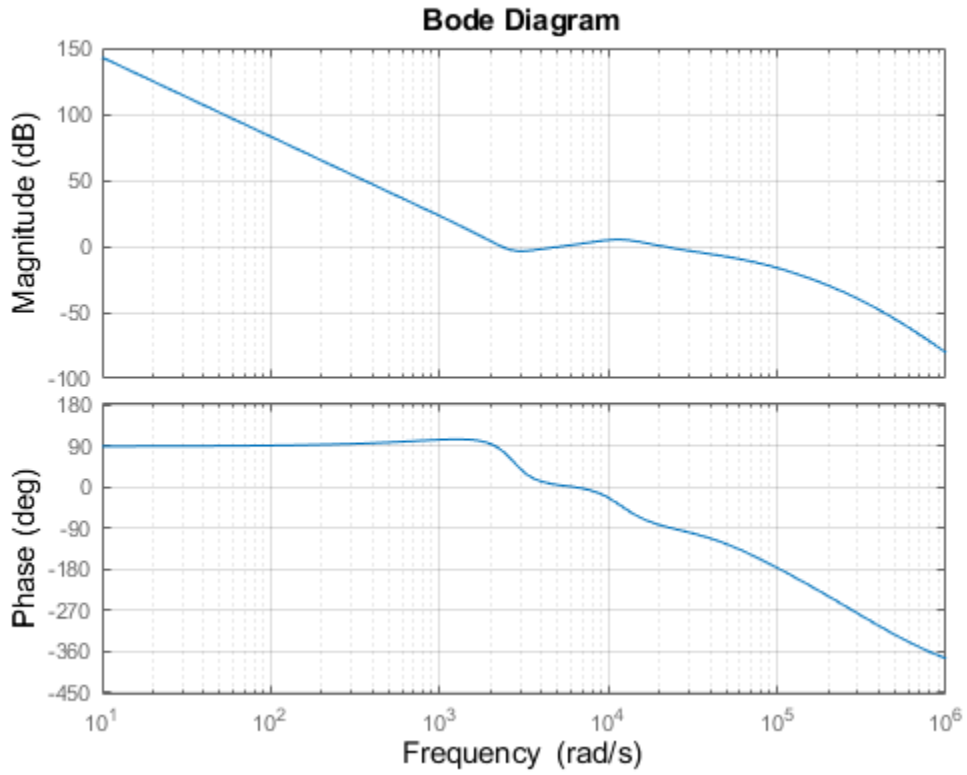
While the Control System Toolbox scaling algorithm handles most models without user intervention, there are rare cases where good accuracy can't be achieved over the entire frequency range and the algorithm must trade good accuracy in one frequency band for bad accuracy in another. In such case, a warning is issued to alert you of potential inaccuracies. To illustrate this behavior, load the next example and plot its Bode response:

```
load numdemo warnsys
bode(warnsys,{1e-3,1e6}), grid on
```



Note the warning issued by the `bode` command. This 17-th order model has dynamics near 0.01 rad/s and between 1e3 and 1e6 rad/s, separated by a 300dB gain drop. You can eliminate the warning by narrowing down the frequency range of interest, for example, to [10,1e6] rad/s:

```
bode(warnsys,{10,1e6}), grid on
```



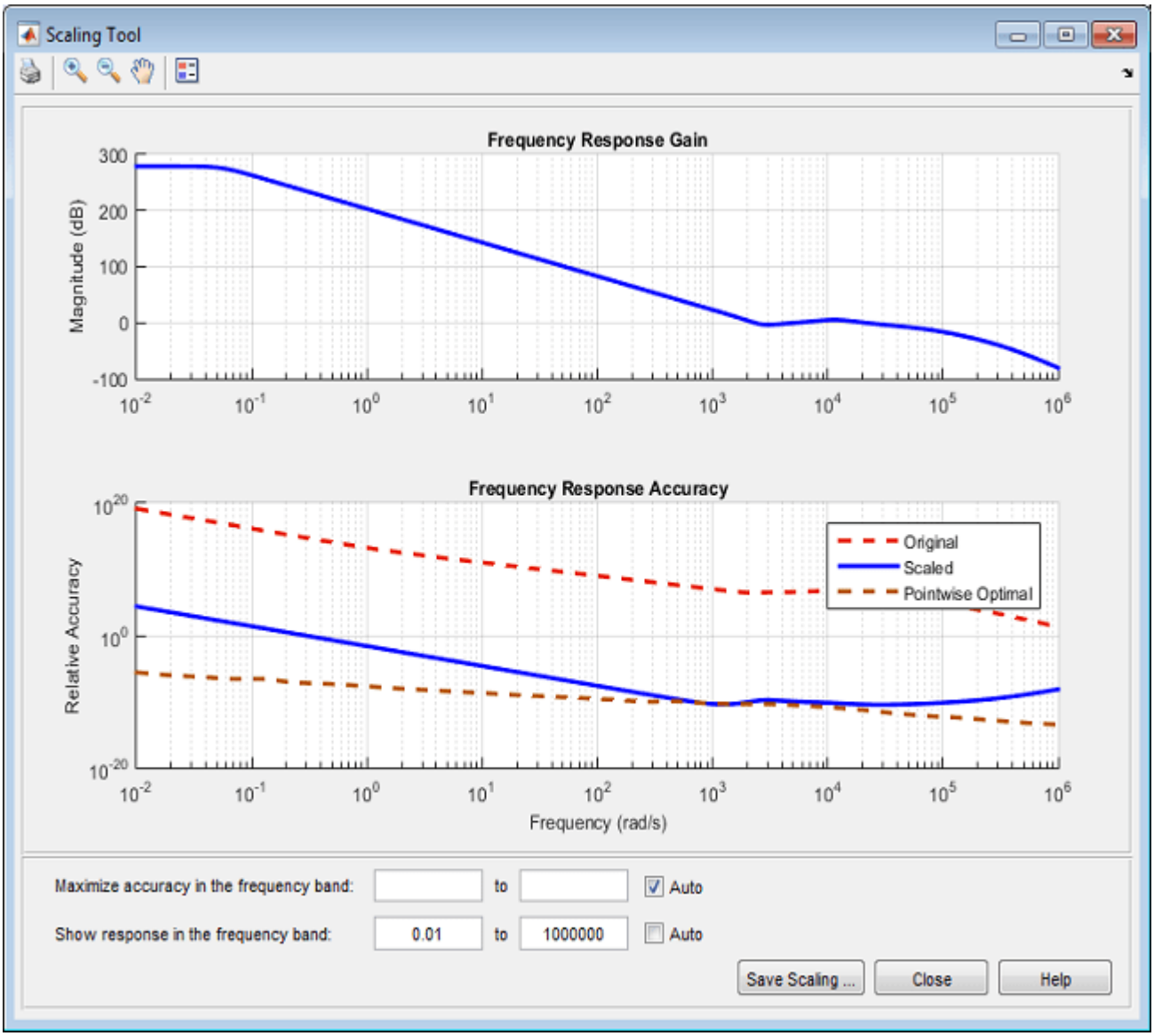
This instructs the algorithm to maximizes accuracy in the interval [10,1e6]. You can also investigate the underlying accuracy tradeoff by typing:

```
>> prescale(warnsys)
```

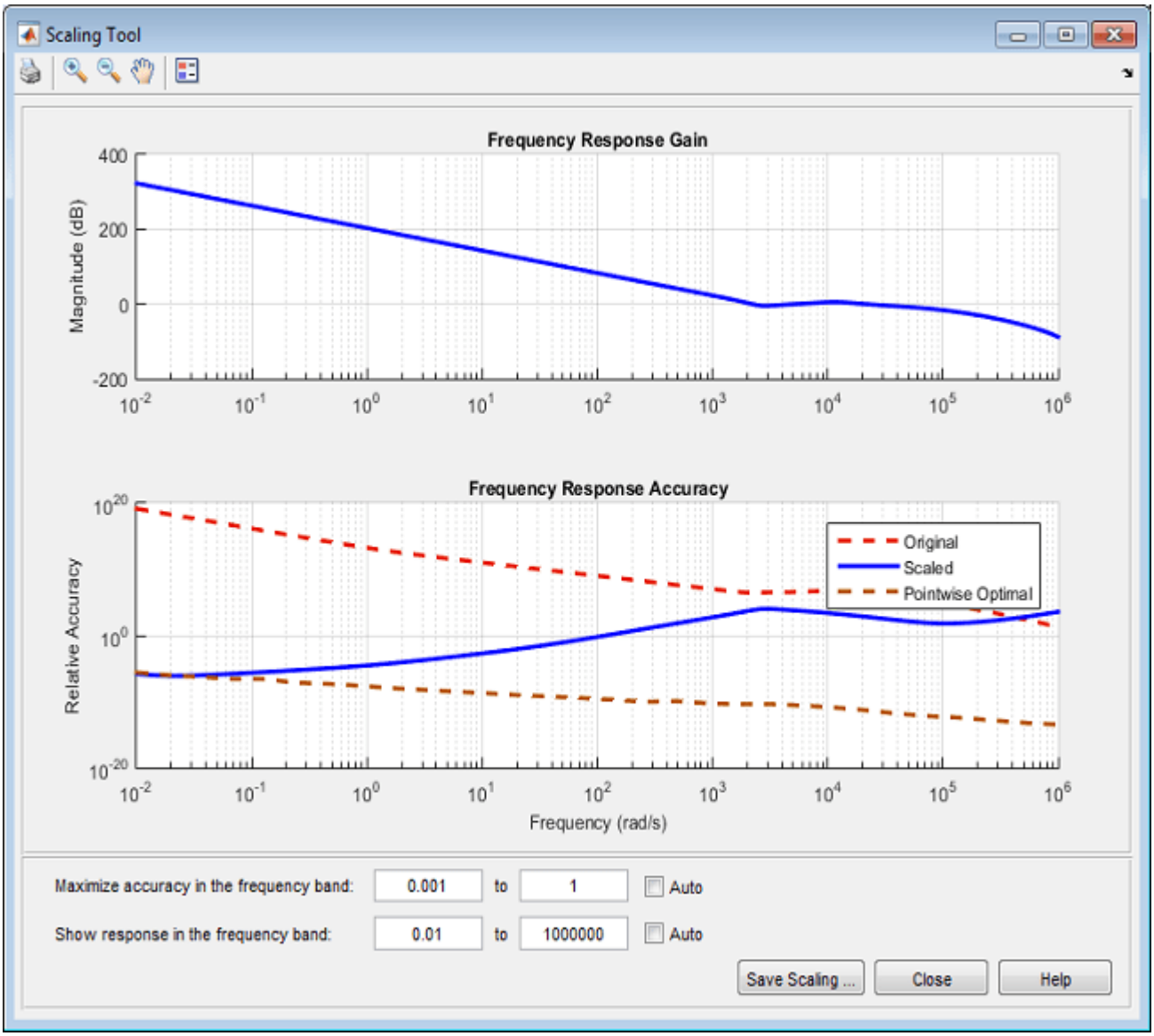
This command opens the interactive Scaling Tool shown below. Set the frequency axis limits to 1e-2 to 1e6. The top plot shows the frequency response magnitude, and the bottom plot shows:

- The relative accuracy of the computed response without scaling (red)
- The relative accuracy of the computed response with scaling (blue)
- The best achievable accuracy when using independent scaling at each frequency (brown)

Any relative accuracy value greater than one signals poor accuracy.



In this example, the Relative Accuracy plot shows that the scaling algorithm achieved good accuracy in the [1e3,1e6] frequency band at the expense of accuracy at low frequencies. If you only care about the frequency band [1e-3,1], you can override this default range selection and manually specify the frequency band where you want maximum accuracy. For example, enter [1e-3,1] in the edit boxes next to **Maximize accuracy in the frequency band**:



This action updates the bottom plot and the relative accuracy of the scaled model (blue curve) is now best near 1e-2 rad/s, but is significantly worse in the [1e3,1e6] band.

The Scaled Property of State-Space Models

The State-Space (@ss) object has a `Scaled` property to indicate when a model is already scaled. Its default value is `false`. The `prescale` command sets this property to `true`:

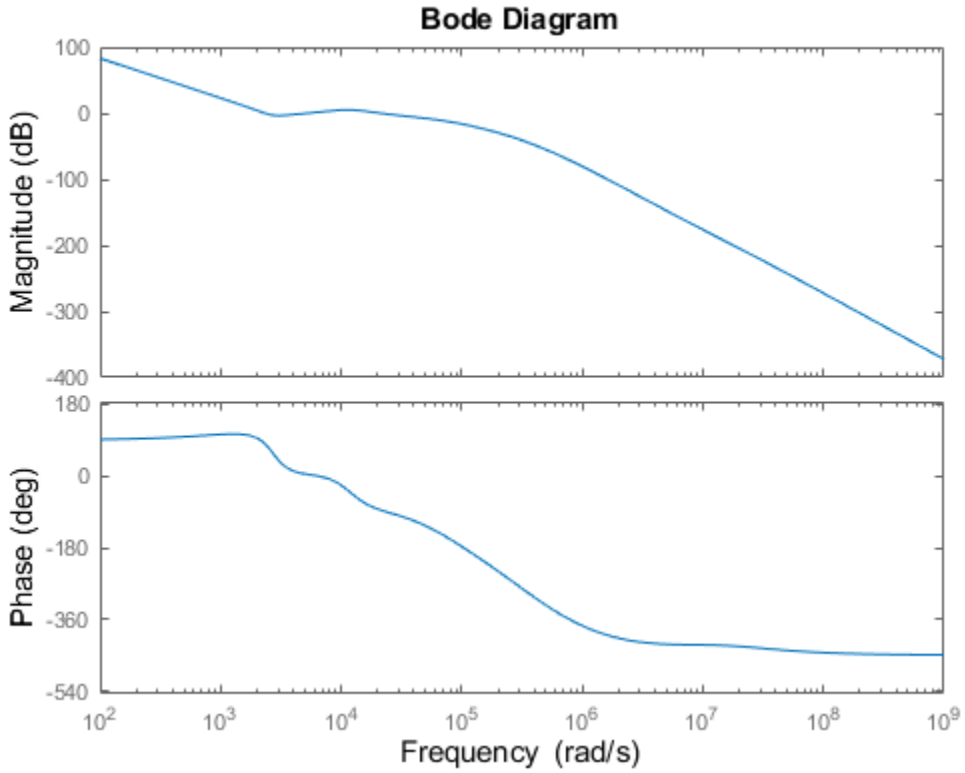
```
sys = prescale(anil);
sys.Scaled
```

```
ans =

  logical
```

```
1
```

Because the scaling algorithm skips models with `Scaled=true`, you can manually set the `Scaled` property to `true` when your model is well scaled and you want to eliminate the small overhead associated with scaling. If you want more control over where accuracy is maximized, you can also explicitly scale your model before using it:

```
sys = prescale(warnsys,{10,1e6});
bode(sys)
```



Here `warnsys` is scaled with emphasis on the frequency range [10,1e6] and `bode` does not attempt to rescale the resulting model `sys` (no more warning).

Conclusion

Proper scaling of state-space models is important to achieve accurate results. Most Control System Toolbox commands take care of scaling automatically. You are alerted when accuracy may be compromised and you can easily correct the problem by specifying the frequency band of interest.

Sensitivity of Multiple Roots

This example shows that high-multiplicity poles have high numerical sensitivity and can shift by significant amounts when switching model representation.

Example

Poles with high multiplicity and clusters of nearby poles can be very sensitive to rounding errors, which can sometimes have dramatic consequences. This example uses a 15th-order discrete-time state-space model `Hss` with a cluster of stable poles near `z=1`:

```
load numdemo Hss
```

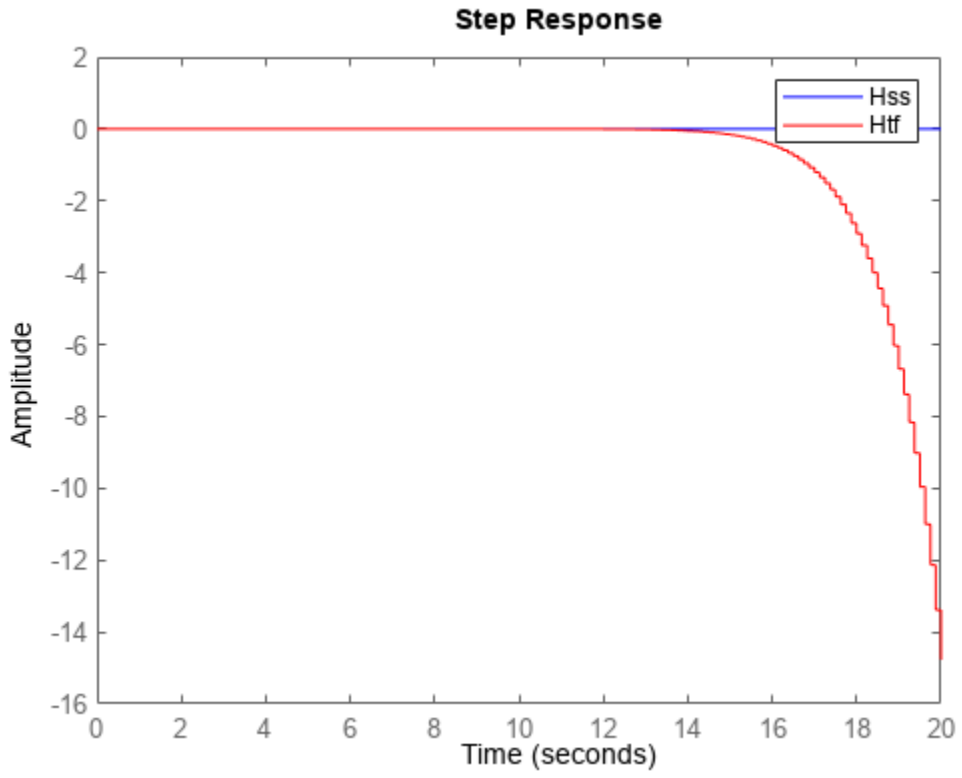
Convert the model to transfer function using `tf`:

```
Htf = tf(Hss);
```

Response Comparison

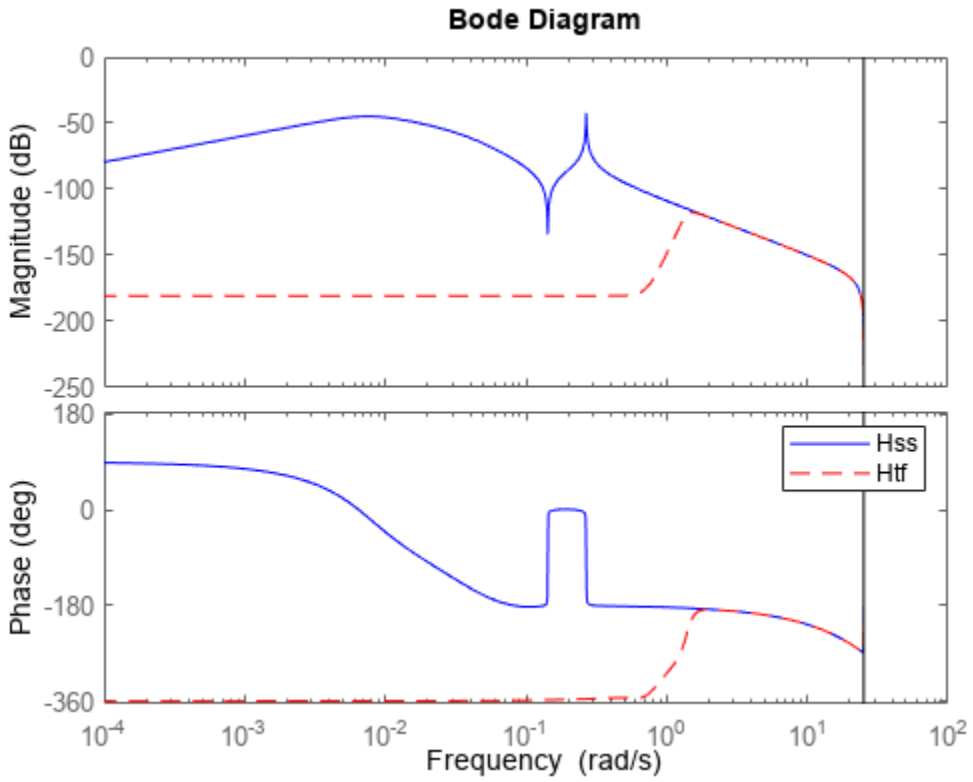
Compare the step responses of `Hss` and `Htf` to see how pole sensitivity can affect the stability of the model and cause large variations in the computed time and frequency responses:

```
step(Hss,'b',Htf,'r',20)
legend('Hss','Htf')
```



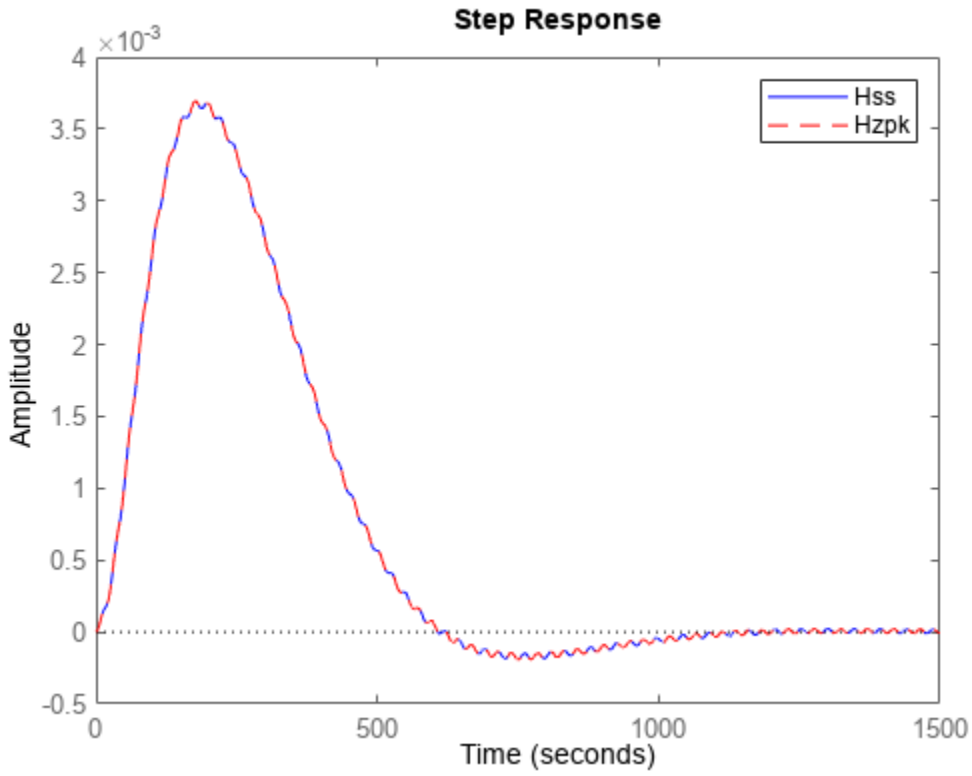
The step response of `Htf` diverges even though the state-space model `Hss` is stable (all its poles lie in the unit circle). The Bode plot also shows a large discrepancy between the state-space and transfer function models:

```
bode(Hss,'b',Htf,'r--')
legend('Hss','Htf')
```

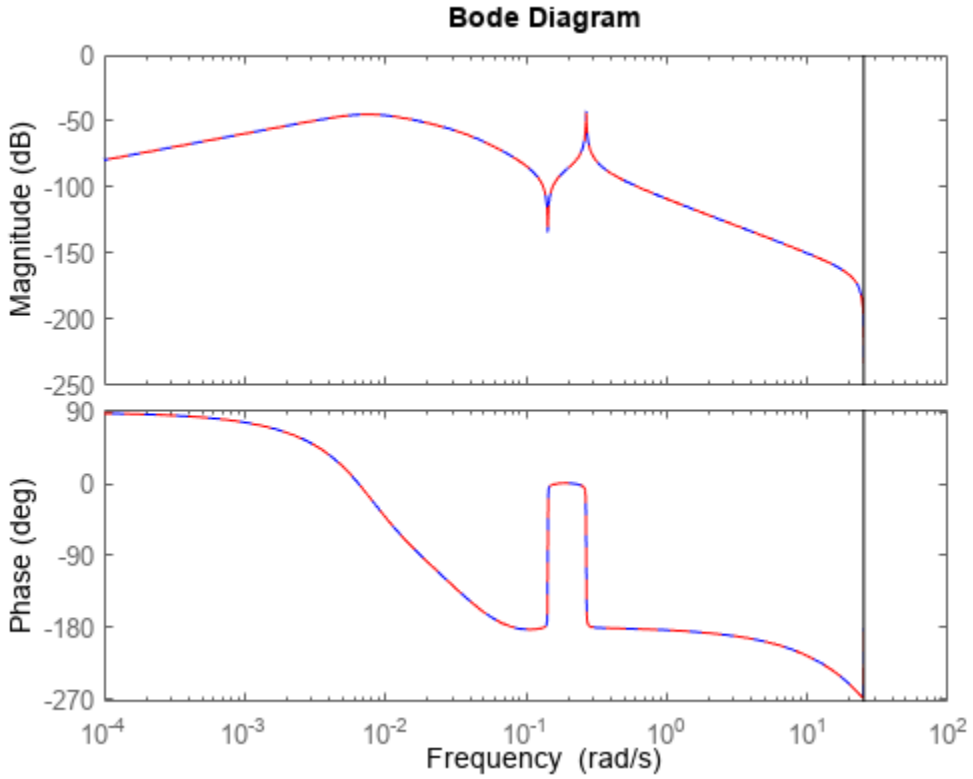


The algorithm used to convert from state space to transfer function is not causing this discrepancy. If you convert from state space to zero-pole-gain, the first step in any SS to TF conversion, the discrepancies disappear:

```
Hzpk = zpk(Hss);

step(Hss,'b',Hzpk,'r--')
legend('Hss','Hzpk')
```




```
bode(Hss,'b',Hzpk,'r--')
```

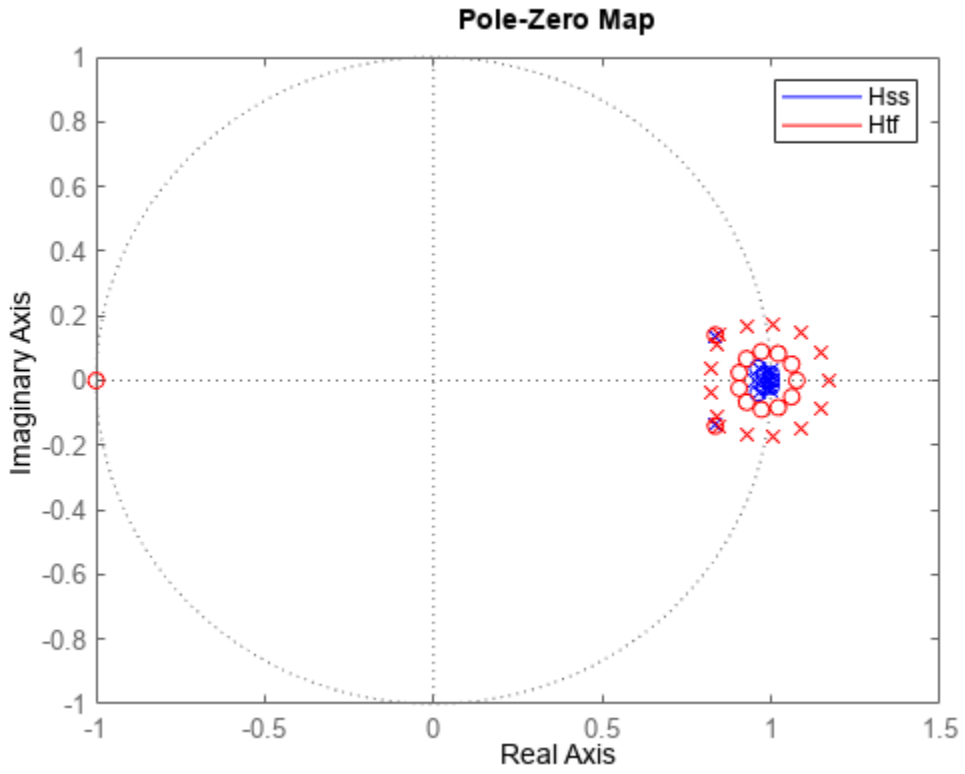


This analysis shows that discrepancies arise in the ZPK to TF conversion, which merely involves computing a polynomial from its roots.

Cause of Discrepancy

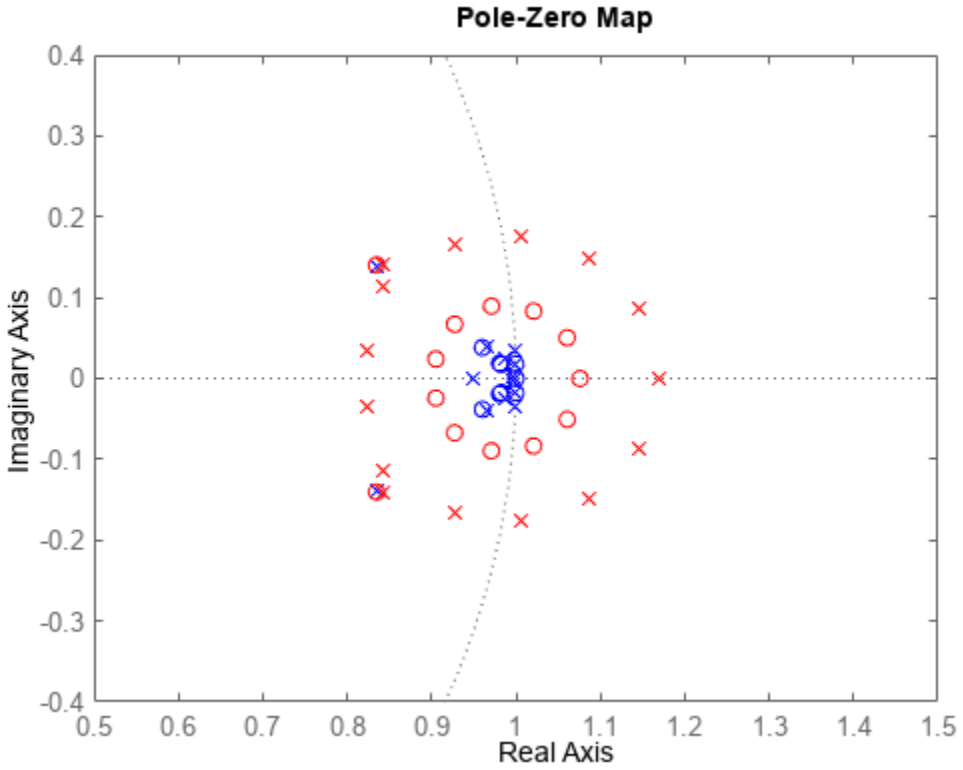
To understand the cause of these large discrepancies, compare the pole/zero maps of the state-space model and its transfer function:

```
pzplot(Hss,'b',Htf,'r')
legend('Hss','Htf')
```



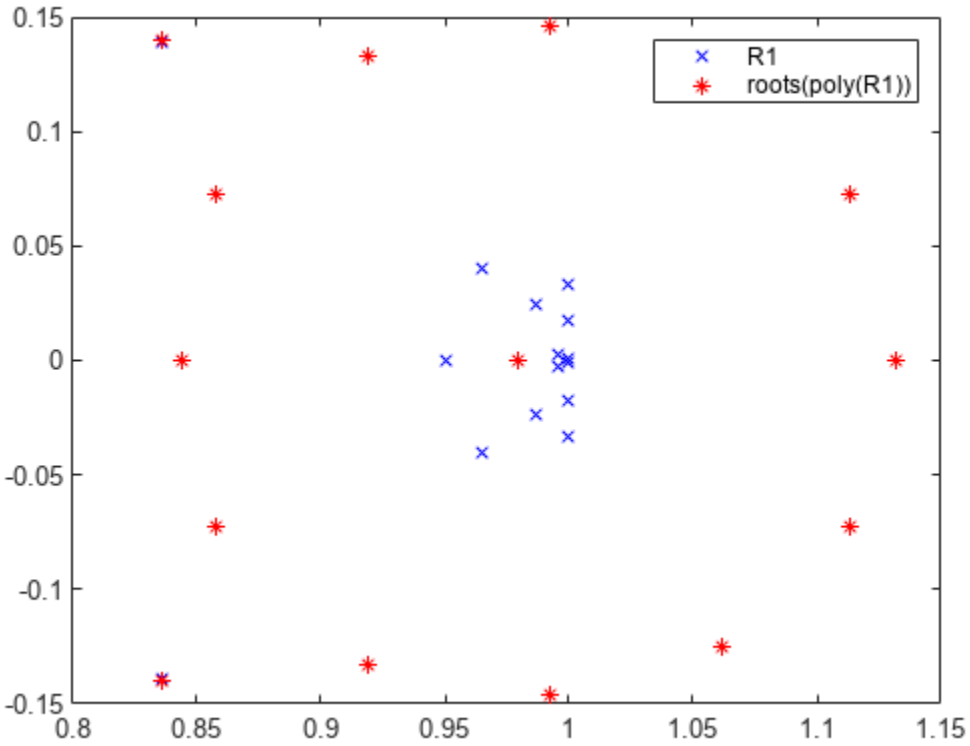
Note the tightly packed cluster of poles near z=1 in `Hss`. When these poles are recombined into the transfer function denominator, roundoff errors perturb the pole cluster into an evenly-distributed ring of poles around z=1 (a typical pattern for perturbed multiple roots). Unfortunately, some perturbed poles cross the unit circle and make the transfer function unstable. Zoom in on the plot to see these poles:

```
pzplot(Hss,'b',Htf,'r');
axis([0.5 1.5 -.4 .4])
```



You can confirm this explanation with a simple experiment. Construct a polynomial whose roots are the poles `R1` of `Hss`, compute the roots of this polynomial, and compare these roots with `R1`:

```
R1 = pole(Hss);                  % poles of Hss
Den = poly(R1);                  % polynomial with roots R1
R2 = roots(Den);                 % roots of this polynomial
plot(real(R1),imag(R1),'bx',real(R2),imag(R2),'r*')
legend('R1','roots(poly(R1))');
```



This plot shows that `ROOTS(POLY(R1))` is quite different from `R1` because of the clustered roots. As a result, the roots of the transfer function denominator differ significantly from the poles of the original state-space model `Hss`.

In conclusion, you should avoid converting state-space or zero-pole-gain models to transfer function form because this process can incur significant loss of accuracy.

Model Simplification

- “Model Reduction Basics” on page 6-2
- “Reduce Model Order Using the Model Reducer App” on page 6-5
- “Balanced Truncation Model Reduction” on page 6-13
- “Approximate Model by Balanced Truncation at the Command Line” on page 6-20
- “Compare Truncated and DC Matched Low-Order Model Approximations” on page 6-23
- “Approximate Model with Unstable or Near-Unstable Pole” on page 6-27
- “Frequency-Limited Balanced Truncation” on page 6-31
- “Model Reduction in the Live Editor” on page 6-36
- “Pole-Zero Simplification” on page 6-43
- “Mode-Selection Model Reduction” on page 6-50
- “Visualize Reduced-Order Models in the Model Reducer App” on page 6-58

Model Reduction Basics

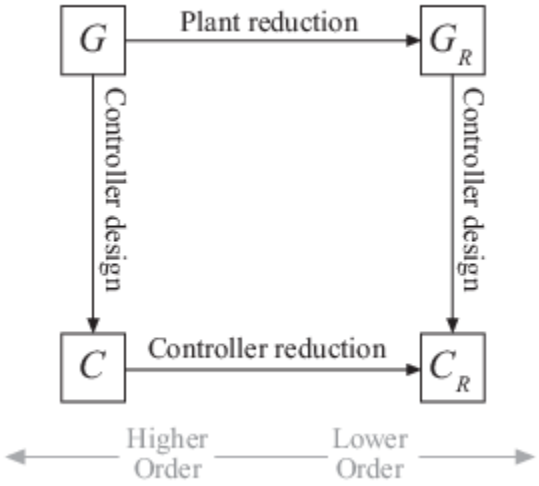
Working with lower-order models can simplify analysis and control design, relative to higher-order models. Simpler models are also easier to understand and manipulate. High-order models obtained by linearizing complex Simulink models or from other sources can contain states that do not contribute much to the dynamics of particular interest to your application. Therefore, it can be useful to reduce model order while preserving model characteristics that are important for your application.

When to Reduce Model Order

Cases where you might want to reduce model order include these situations:

- You are working with a relatively high-order model obtained from linearizing a Simulink model, performing a finite-element calculation, interconnecting model elements, or other source.
- You want to improve the simulation speed of a Simulink model at a certain operating point. In that case, you can linearize a portion of the model at that operating point and compute a reduced-order simplification or approximation of the linearized model. You can then replace the portion of the model with an LTI Block containing the reduced-order model.
- You design a high-order controller that you want to implement as a lower-order controller, such as a PID controller. For example, controller design using Linear-Quadratic-Gaussian methods or H_∞ synthesis techniques can yield a high-order result. In this case, you can try reducing the plant order before synthesis, reducing the controller order after synthesis, or both.
- You want to simplify a model obtained by identification with System Identification Toolbox software.

The following diagram illustrates the relationship between model reduction and control design.



In general, when designing a controller for a system represented by a high-order model, G, it is useful to start by simplifying the plant model. Then, design a relatively low-order controller, C_R, for the lower-order plant model G_R. After you design a controller for either the original or the reduced plant model, you can try to reduce the controller further.

Reducing the plant or controller can include:

- Discarding states that do not contribute to the system dynamics, such as structurally disconnected states or canceling pole-zero pairs.
- Discarding low-energy states that contribute relatively little to system dynamics.
- Focusing on a particular frequency region and discarding dynamics outside that region. For example, if your control bandwidth is limited by actuator dynamics, discard higher-frequency dynamics.

In any case, when you reduce model order, you want to preserve model characteristics that are important for your application. Whenever you compute a reduced-order model, verify that the reduced model preserves time-domain or frequency-domain behavior that you care about. For example, for control design, it is useful to verify that the reduced closed-loop system is stable. It is also useful to check that the reduced open-loop transfer function $C_R G_R$ adequately matches the original models where the open-loop gain GC is close to 1 (in the gain crossover region).

Model Reduction Tools

Control System Toolbox offers tools for model reduction in several environments. These include:

- Functions for performing model reduction at the MATLAB command prompt, in scripts, or in your own functions.
- **Reduce Model Order** task for generating code in the Live Editor. When you are working in a live script, use this task to interactively experiment with model-reduction methods and parameters and generate code for your live script.
- **Model Reducer** app, a standalone app that lets you import models from the MATLAB workspace, and interactively generate reduced-order models using different methods and parameters. The app can also generate code for use in a MATLAB script or function.

Choosing a Model Reduction Method

To reduce the order of a model, you can either simplify your model, or compute a lower-order approximation. The following table summarizes the differences among several model-reduction approaches.

Approach	Command Line	Model Reducer App and Reduce Model Order Live Editor Task
Simplification — Reduce model order exactly by canceling pole-zero pairs or eliminating states that have no effect on the overall model response	• `sminreal` — Eliminate states that are structurally disconnected from the inputs or outputs. • `minreal` — Eliminate canceling or near-canceling pole-zero pairs from transfer functions. Eliminate unobservable or uncontrollable states from state-space models.	"Pole-Zero Simplification" on page 6-43 method — Eliminate: • Structurally disconnected states • Unobservable or uncontrollable states from state-space models • Canceling or near-canceling pole-zero pairs from transfer functions

Approach	Command Line	Model Reducer App and Reduce Model Order Live Editor Task
Approximation — Compute a lower-order approximation of your model.	`balred` — Discard states that have relatively low effect on the overall model response.	Balanced Truncation on page 6-13 method — Discard states that have relatively low effect on the overall model response.
Mode Selection — Eliminate poles and zeros that fall outside a specific frequency range of interest.	`freqsep` — Separate model into slow and fast dynamics around a specified cutoff frequency.	Mode Selection on page 6-50 method — Select frequency range of interest and discard dynamics outside that range.

Sometimes, approximation can yield better results, even if the model looks like a good candidate for simplification. For example, models with near pole-zero cancellations are sometimes better reduced by approximation than simplification. Similarly, using `balred` to reduce state-space models can yield more accurate results than `minreal`.

When you use a reduced-order model, always verify that the simplification or approximation preserves model characteristics that are important for your application. For example, compare the frequency responses of the original and reduced models using `bodeplot` or `sigmaplot`. Or, compare the open-loop responses for the original and reduced plant and controller models.

See Also

Apps
Model Reducer

Live Editor Tasks
Reduce Model Order

Functions
`balred` | `freqsep` | `minreal` | `sminreal`

Related Examples

- “Balanced Truncation Model Reduction” on page 6-13
- “Mode-Selection Model Reduction” on page 6-50
- “Pole-Zero Simplification” on page 6-43

Reduce Model Order Using the Model Reducer App

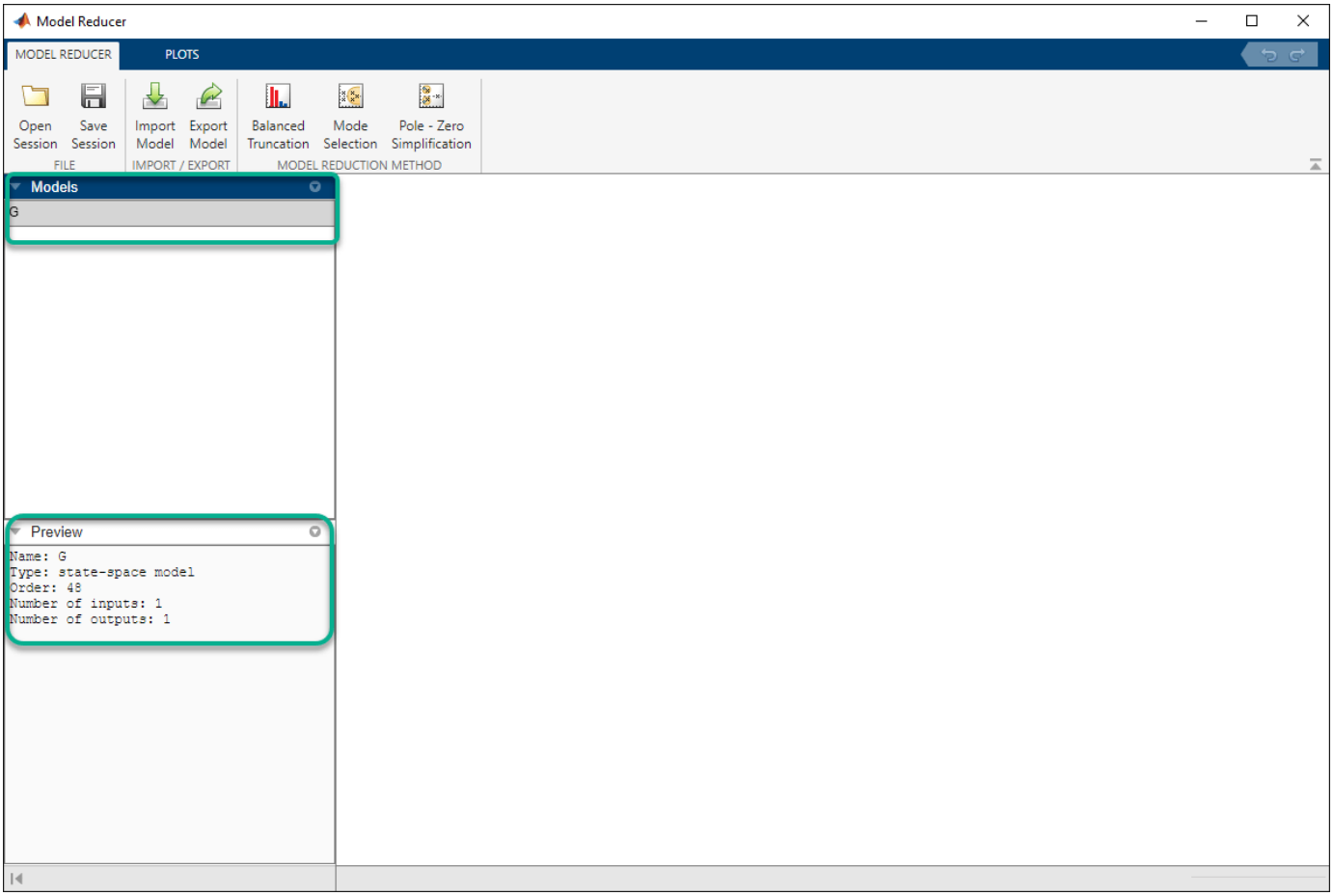
This example shows how to reduce model order while preserving important dynamics using the **Model Reducer** app. This example illustrates the Balanced Truncation method, which eliminates states based on their energy contributions to the system response.

Open Model Reducer with a Building Model

This example uses a model of the Los Angeles University Hospital building. The building has eight floors, each with three degrees of freedom: two displacements and one rotation. The input-output relationship for any one of these displacements is represented as a 48-state model, where each state represents a displacement or its rate of change (velocity). Load the building model and open **Model Reducer** with that model.

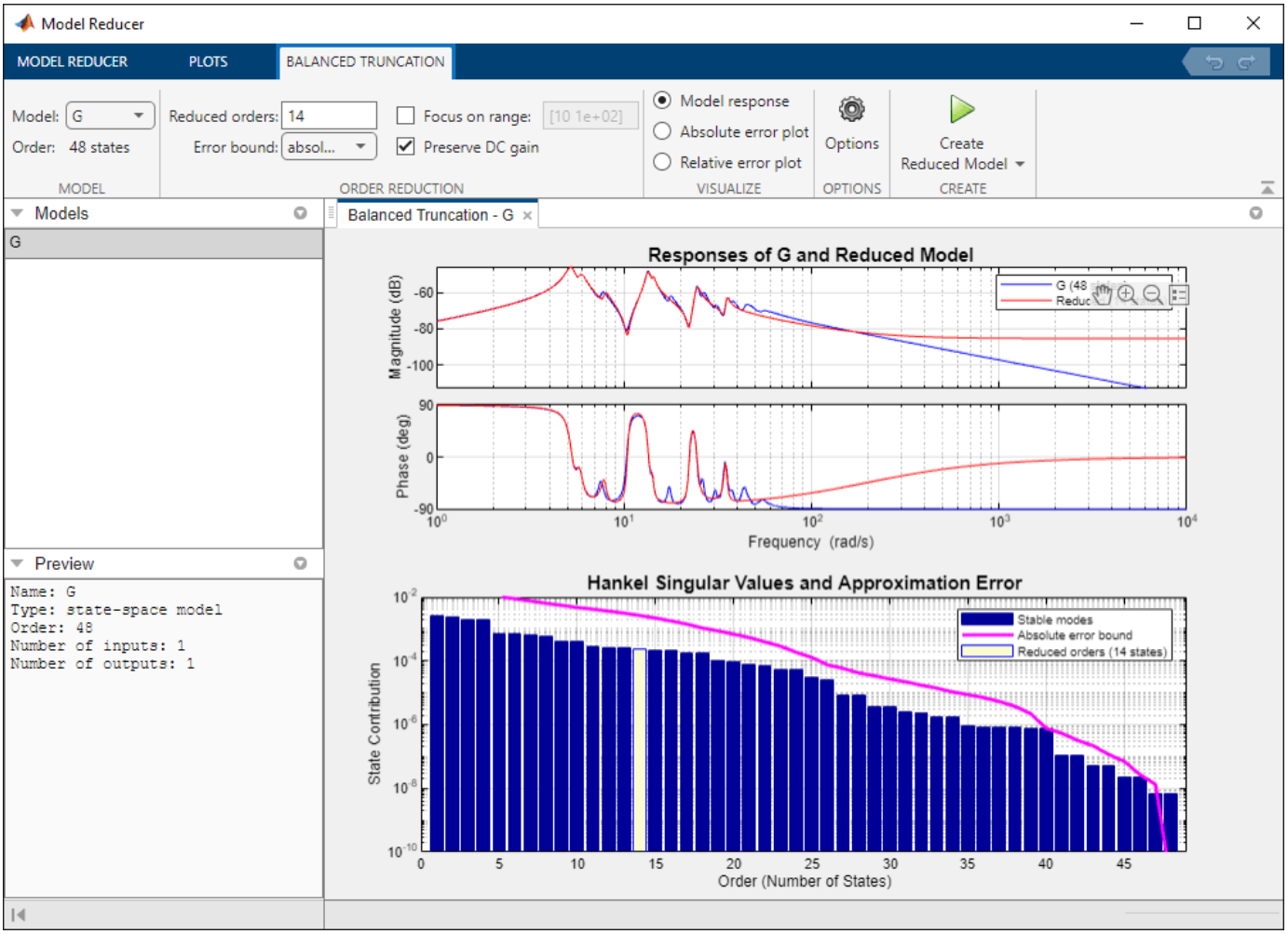
```
load building.mat
modelReducer(G)
```

Select the model in the data browser to display some information about the model in the Preview section. Double-click the model to see more detailed information.



Open the Balanced Truncation Tab

Model Reducer has three model reduction methods: Balanced Truncation, Mode Selection, and Pole/Zero Simplification. For this example, click **Balanced Truncation**.



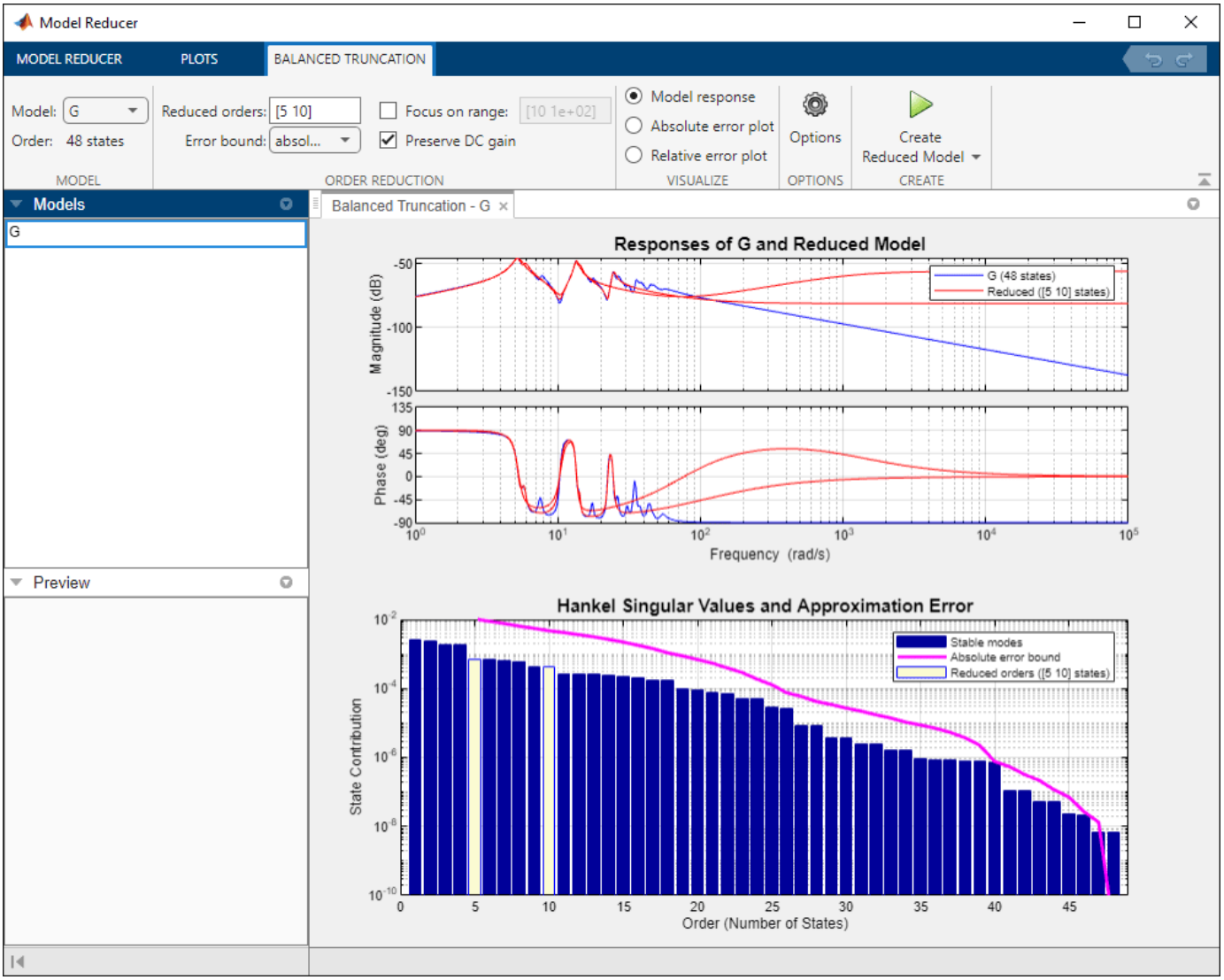
Model Reducer opens the **Balanced Truncation** tab and automatically generates a reduced-order model. The top plot compares the original and reduced model in the frequency domain. The bottom plot shows the energy contribution of each state, where the states are sorted from high energy to low energy. The order of the reduced model, 14, is highlighted in the bar chart. In the reduced model, all states with lower energy contribution than this one are discarded.

Compute Multiple Approximations

Suppose that you want to preserve the first, second, and third peaks of the model response, around 5.2 rad/s, 13 rad/s, and 25 rad/s. Try other model orders to see whether you can achieve this goal with a lower model order. Compute a 5th-order and a 10th-order approximation in one of the following ways:

- In the **Reduced model orders** text box, enter `[5 10]`.
- In the state-contribution plot, ctrl-click the bars for state 5 and state 10.

Model Reducer computes two new reduced-order models and displays them on the response plot with the original model G. To examine the three peaks more closely, Zoom in on the relevant frequency range. The 10th-order model captures the three peaks successfully, while the 5th-order model only approximates the first two peaks. (For information about zooming and other interactions with the analysis plots, see “Visualize Reduced-Order Models in the Model Reducer App” on page 6-58.)



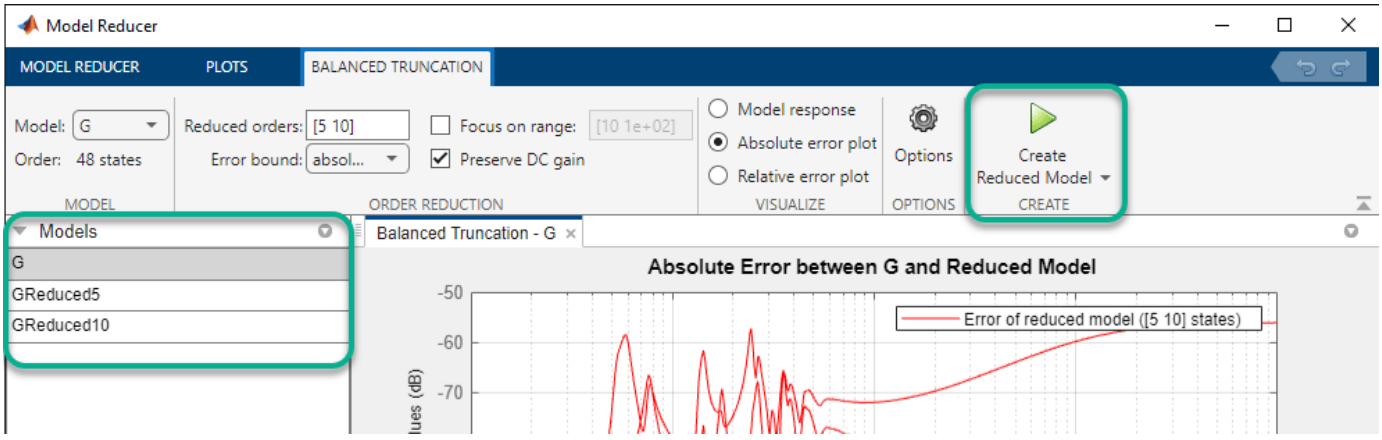
Compare Reduced Models with Different Visualizations

In addition to the frequency response plot of all three models, **Model Reducer** lets you examine the absolute and relative error between the original and reduced models. Select **Absolute error plot** to see the difference between the building and reduced models.

The 5th-order reduced model has at most -60dB error in the frequency region of the first two peaks, below about 30 rad/s. The error increases at higher frequencies. The 10th-order reduced model has smaller error over all frequencies.

Create Reduced Models in Data Browser

Store the reduced models in the data browser by clicking **Create Reduced Model**. The 5th-order and 10th-order reduced models appear in the data browser with names `GReduced5` and `Greduced10`.

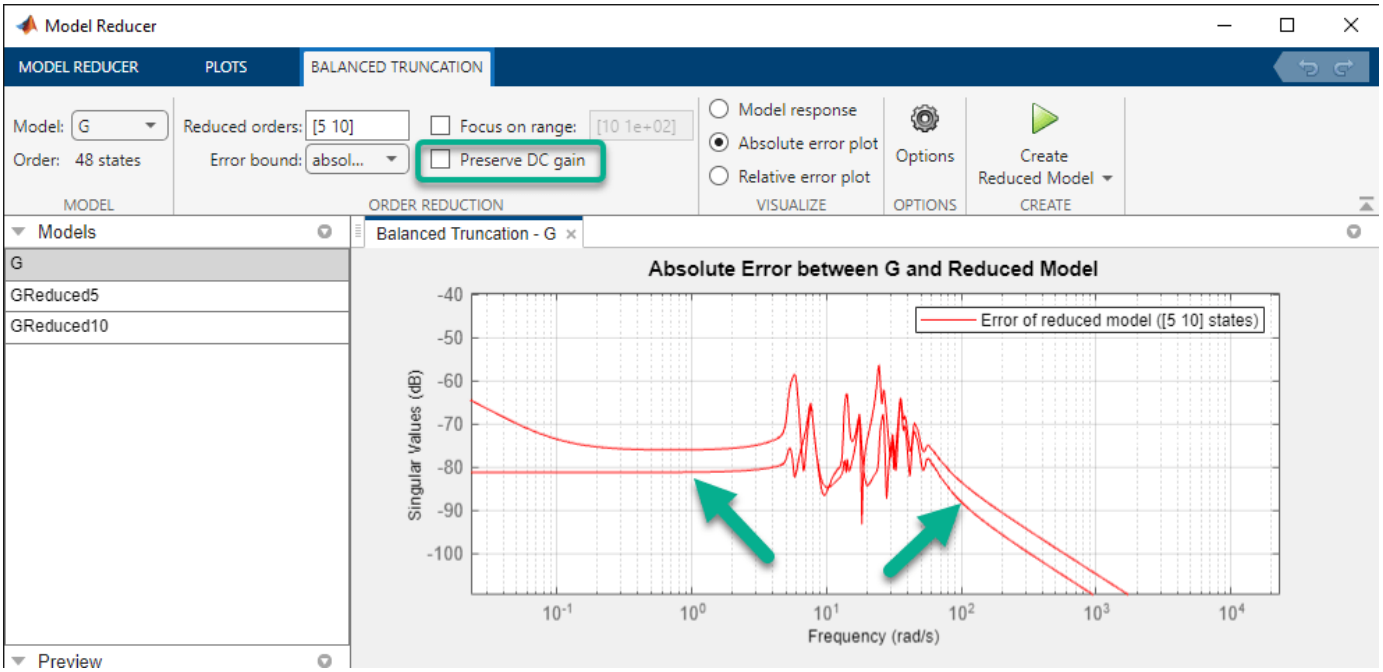


You can continue to change the model-reduction parameters and generate additional reduced models. As you do so, `GReduced5` and `Greduced10` remain unchanged in the data browser.

Focus on Dynamics at Particular Frequencies

By default, balanced truncation in **Model Reducer** preserves DC gain, matching the steady-state response of the original and reduced models. Clear the **Preserve DC Gain** checkbox to better

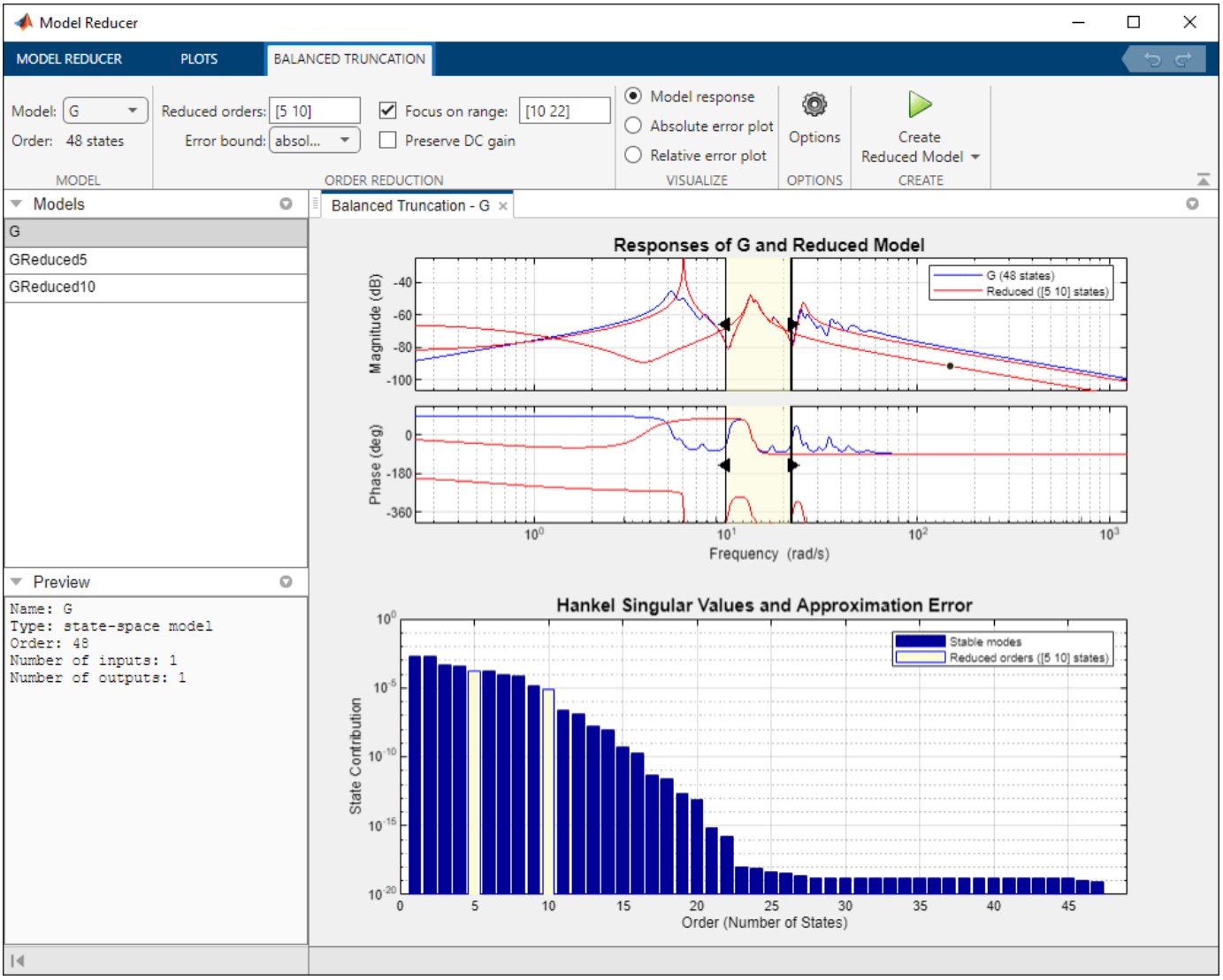
approximate high-frequency dynamics. **Model Reducer** computes new reduced models. The error in the high-frequency region is decreased at the cost of a slight increase in error at low frequencies.



You can also focus the balanced truncation on the model dynamics in a particular frequency interval. For example, approximate only the second peak of the building model around 13 rad/s. First, select the **Model response** plot to see the Bode plots of models. Then check **Focus on range** checkbox. **Model Reducer** analyzes state contributions in the highlighted frequency interval only.

You can drag the boundaries to change the frequency range interactively. As you change the frequency interval, the Hankel Singular Value plot reflects the changes in the energy contributions of the states.

Enter the frequency limits `[10 22]` into the text box next to **Focus on range**. The 5th-order reduced model captures the essential dynamics. The 10th-order model has almost the same dynamics as the original building model within this frequency range.

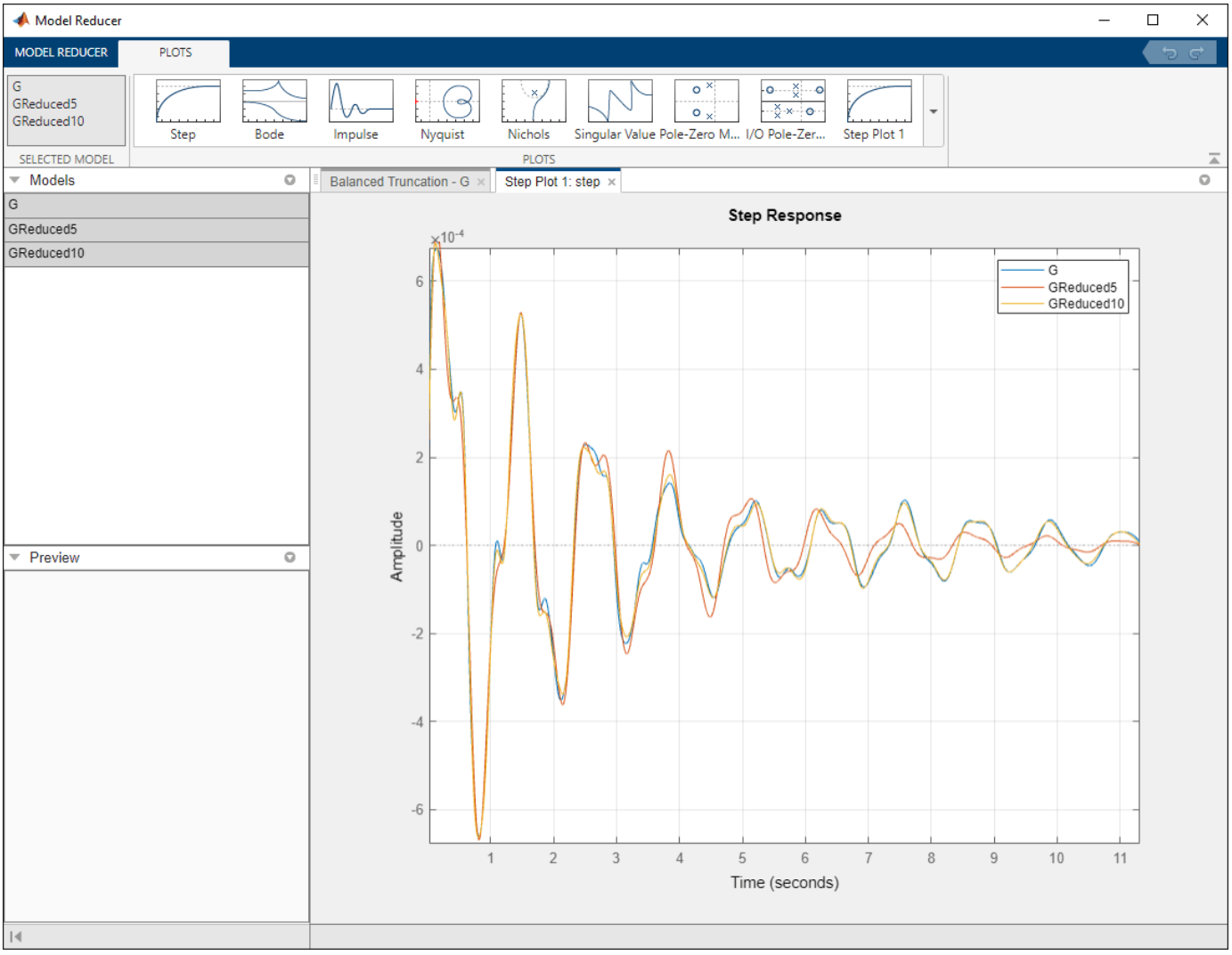


Optionally, store these additional models in the data browser by clicking **Create Reduced Model**.

Compare Models in Time Domain

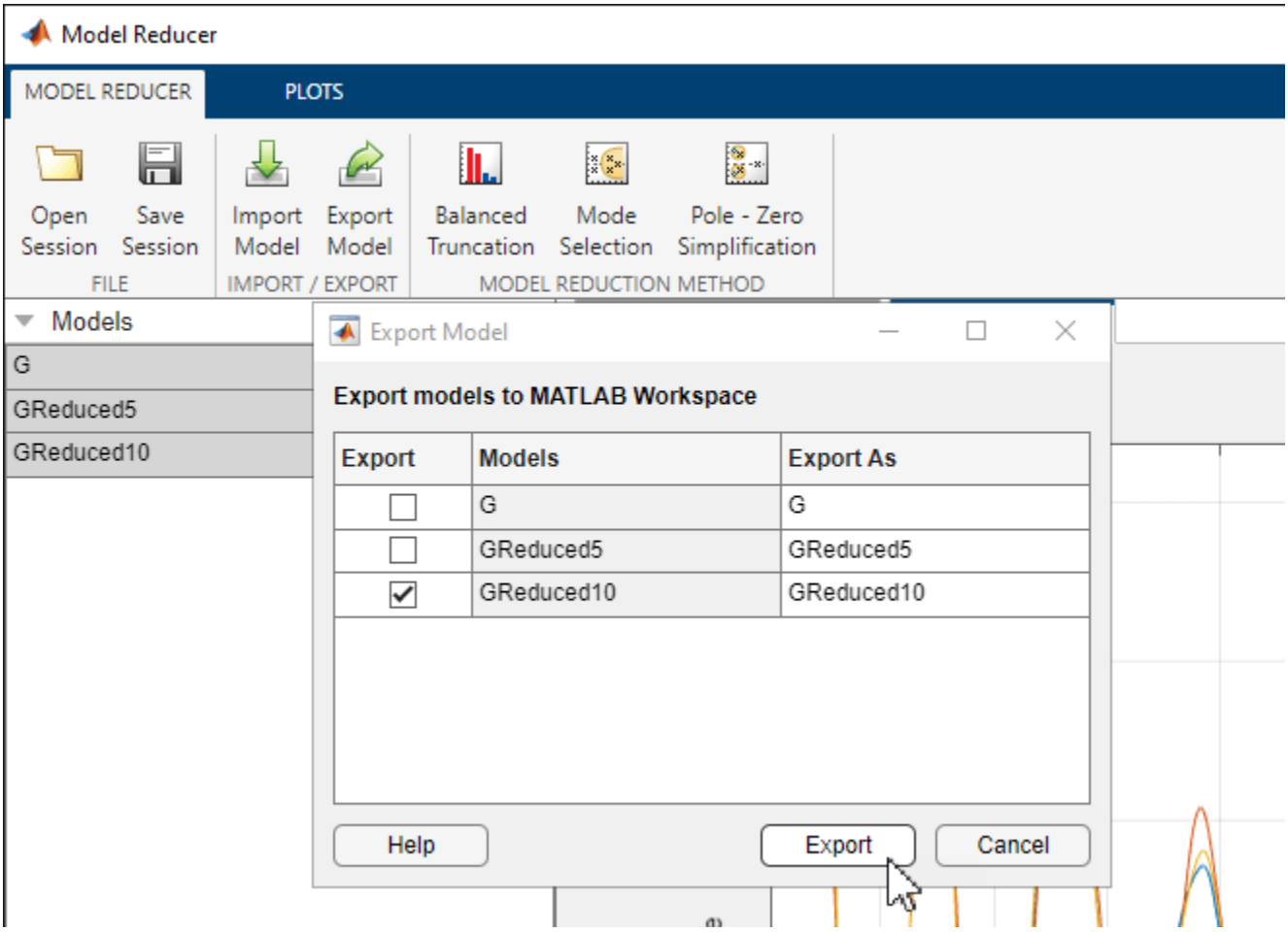
You can compare time-domain responses of the stored reduced models and the original in the **Plots** tab. In the data browser, control-click to select the models you want to compare, `G`, `GReduced5`, and `GReduced10`. Then, click `Step`. **Model Reducer** creates a step plot with all three models.

Zooming on the transient behavior of this plot shows that `GReduced10` captures the time domain behavior of the original model well. However, the response of `GReduced5` deviates from the original model after about 3 seconds.



Export Model for Further Analysis

Comparison of the reduced and original models in the time and frequency domains shows that `GReduced10` adequately captures the dynamics of interest. Export that model to the MATLAB® workspace for further analysis and design. In the **Model Reducer** tab, click **Export Model**. Clear the check boxes for `G` and `Greduced5`, and click **Export** to export `Greduced10`.



`Greduced10` appears in the MATLAB workspace as a state-space (`ss`) model.

See Also

Apps
Model Reducer

Live Editor Tasks
Reduce Model Order

Related Examples

- “Model Reduction Basics” on page 6-2
- “Balanced Truncation Model Reduction” on page 6-13
- “Pole-Zero Simplification” on page 6-43
- “Mode-Selection Model Reduction” on page 6-50
- “Model Reduction in the Live Editor” on page 6-36

Balanced Truncation Model Reduction

Balanced truncation computes a lower-order approximation of your model by neglecting states that have relatively low effect on the overall model response. Using a lower-order approximation that preserves the dynamics of interest can simplify analysis and control design. In the balanced truncation method of model reduction, the software measures state contributions by Hankel singular values (see `hsvd`) and discards states with smaller values. You can compute a reduced-order model by balanced truncation:

- At the command line, using the `balred` command.
- In the **Model Reducer** app, using the **Balanced Truncation** method.
- In the **Reduce Model Order** task in the Live Editor, using the `Balanced Truncation` method.

For more general information about model reduction, see “Model Reduction Basics” on page 6-2.

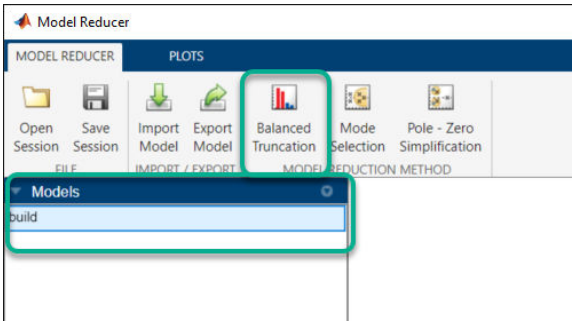
Balanced Truncation in the Model Reducer App

Model Reducer provides an interactive tool for performing model reduction and examining and comparing the responses of the original and reduced-order models. To approximate a model by balanced truncation in **Model Reducer**:

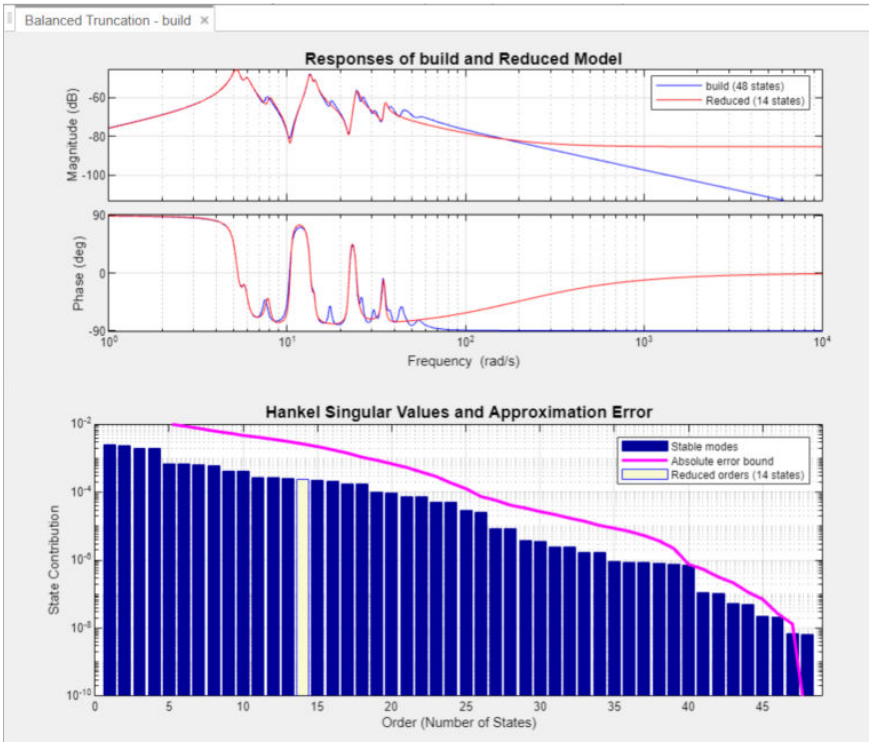
1. Open the app, and import an LTI model to reduce. For instance, suppose that there is a model named `build` in the MATLAB workspace. The following command opens **Model Reducer** and imports the model.

   ```
   modelReducer(build)
   ```

2. In the data browser, select the model to reduce. Click  **Balanced Truncation**.



   In the **Balanced Truncation** tab, **Model Reducer** displays a plot of the frequency response of the original model and a reduced version of the model. The frequency response is a Bode plot for SISO models, and a singular-value plot for MIMO models. The app also displays a Hankel singular-value and approximation error plot of the original model.



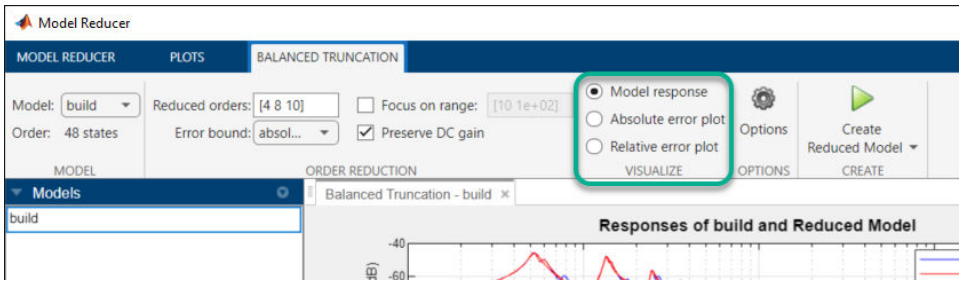
The Hankel singular-value plot shows the relative energy contributions of each state in the system. **Model Reducer** computes an initial reduced-order model based on these values. The highlighted bar is the lowest-energy state in the initial reduced-order model. **Model Reducer** discards states that have lower Hankel singular values than the highlighted bar.

3 Try different reduced-model orders to find the lowest-order model that preserves the dynamics that are important for your application. To specify different orders, either:

- Enter model orders in the **Reduced orders** field. You can enter a single integer or an array of integers, such as `10:14` or `[8,11,12]`.
- Click a bar on the Hankel singular-value plot to specify the lowest-energy state of the reduced-order model. Ctrl-click to specify multiple values.

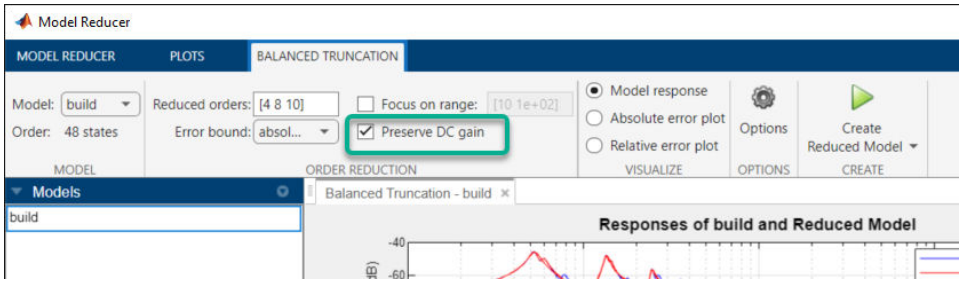
When you change the specified reduced model order, **Model Reducer** automatically computes a new reduced-order model. If you specify multiple model orders, **Model Reducer** computes multiple reduced-order models and displays their responses on the plot.

4 Optionally, examine the absolute or relative error between the original and reduced-order model, in addition to the frequency response. Select the error-plot type using the buttons on the **Balanced Truncation** tab.



For more information about using the analysis plots, see “Visualize Reduced-Order Models in the Model Reducer App” on page 6-58.

5 If low-frequency dynamics are not important to your application, you can clear the **Preserve DC Gain** checkbox. Doing so sometimes yields a better match at higher frequencies between the original and reduced-order models.

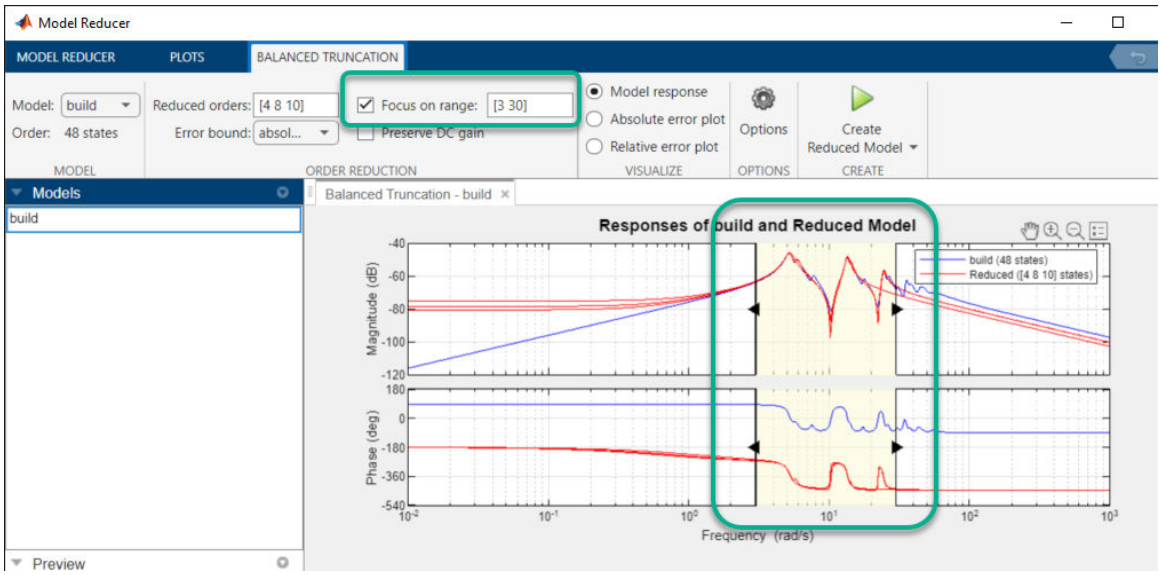


When you check or clear the **Preserve DC Gain** checkbox, **Model Reducer** automatically computes new reduced-order models. For more information about this option, see “Compare Truncated and DC Matched Low-Order Model Approximations” on page 6-23.

6 Optionally, limit the Hankel singular-value computation to a specific frequency range. Such a limit is useful when the model has modes outside the region of interest to your particular application. When you apply a frequency limit, **Model Reducer** determines which states to truncate based on their energy contribution within the specified frequency range only. Neglecting energy contributions outside that range can yield an even lower-order approximation that is still adequate for your application.

To limit the singular-value computation, check **Focus on range**. Then, specify the frequency range by:

- In the text box, entering a vector of the form `[fmin,fmax]`. Units are `rad/TimeUnit`, where `TimeUnit` is the `TimeUnit` property of the model you are reducing.
- On the response plot or error plot, dragging the boundaries of the shaded region or the shaded region itself. **Model Reducer** analyzes the state contributions within the shaded region only.

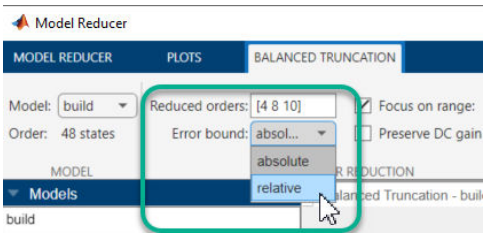


When you check or clear the **Focus on range** checkbox or change the selected range, **Model Reducer** automatically computes new reduced-order models.

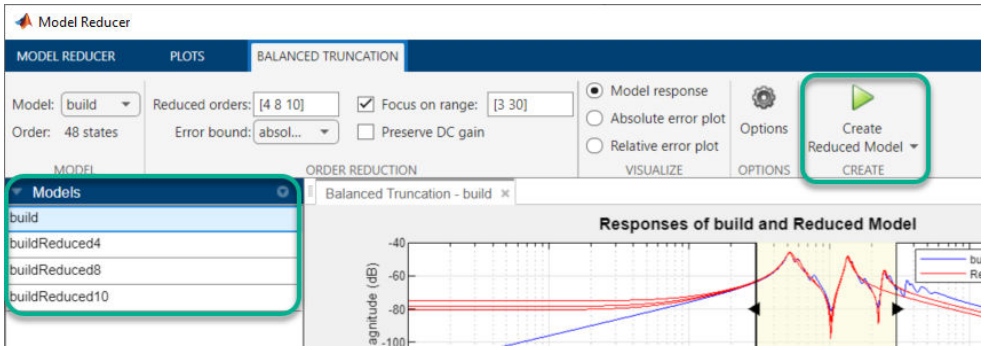
Note Checking **Focus on range** automatically clears **Preserve DC Gain**. To enforce a DC match even when using frequency limits, recheck **Preserve DC Gain**. Note that restricting the frequency range is not supported with relative error control.

7 You can choose between absolute and relative errors by selecting the appropriate option in **Error Bound**. Setting it to **absolute** controls the absolute error $\|G - G_r\|_\infty$ while setting it to **relative** controls the relative error $\left\|G^{-1}(G - G_r)\right\|_\infty$. Relative error gives a better match across frequency while absolute error emphasizes areas with most gain.

Note Switching between **Error Bound** options automatically clears **Preserve DC Gain** and **Focus on Range**. To enforce a DC match, recheck **Preserve DC Gain**. Note that restricting the frequency range is not supported with relative error control.



8 When you have one or more reduced models that you want to store and analyze further, click  . The new models appear in the data browser. If you have specified multiple orders, each reduced model appears separately. Model names reflect the reduced model order.



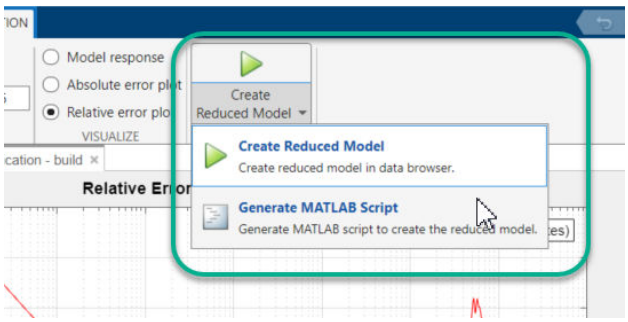
After creating reduced models in the data browser, you can continue changing the reduction parameters and create reduced models with different orders for analysis and comparison.

You can now perform further analysis with the reduced model. For example:

- Examine other responses of the reduced system, such as the step response or Nichols plot. To do so, use the tools on the **Plots** tab. See “Visualize Reduced-Order Models in the Model Reducer App” on page 6-58 for more information.
- Export reduced models to the MATLAB workspace for further analysis or control design. On the **Model Reducer** tab, click **Export**.

Generate MATLAB Code for Balanced Truncation

To create a MATLAB script you can use for further model-reduction tasks at the command line, click **Create Reduced Model**, and select **Generate MATLAB Script**.



Model Reducer creates a script that uses the `balred` command to perform model reduction with the parameters and options you have set on the **Balanced Truncation** tab. The script opens in the MATLAB editor.

Balanced Truncation in Other Environments

Instead of working in the **Model Reducer** app, you can perform balanced truncation in other environments.

- In the Live Editor, use the **Reduce Model Order** task to interactively perform balanced-truncation model reduction and generate code in your live script. For an example, see "Model Reduction in the Live Editor" on page 6-36.
- At the MATLAB command prompt or in scripts and functions, use the `balred` command. For examples, see:
  - "Approximate Model by Balanced Truncation at the Command Line" on page 6-20
  - "Compare Truncated and DC Matched Low-Order Model Approximations" on page 6-23
  - "Approximate Model with Unstable or Near-Unstable Pole" on page 6-27
  - "Frequency-Limited Balanced Truncation" on page 6-31

See Also

Apps
Model Reducer

Live Editor Tasks
Reduce Model Order

Functions
`balred` | `hsvplot`

Related Examples

- "Mode-Selection Model Reduction" on page 6-50
- "Pole-Zero Simplification" on page 6-43

- “Model Reduction Basics” on page 6-2

Approximate Model by Balanced Truncation at the Command Line

This example shows how to use `balred` to compute a reduced-order approximation of a model at the MATLAB® command line.

`balred` removes the states with the lowest energy contribution to overall model behavior. Therefore, to use `balred`, you can begin by examining the energy contribution of the model states. You choose the approximation order based on the number of states that make a significant contribution to the overall model behavior.

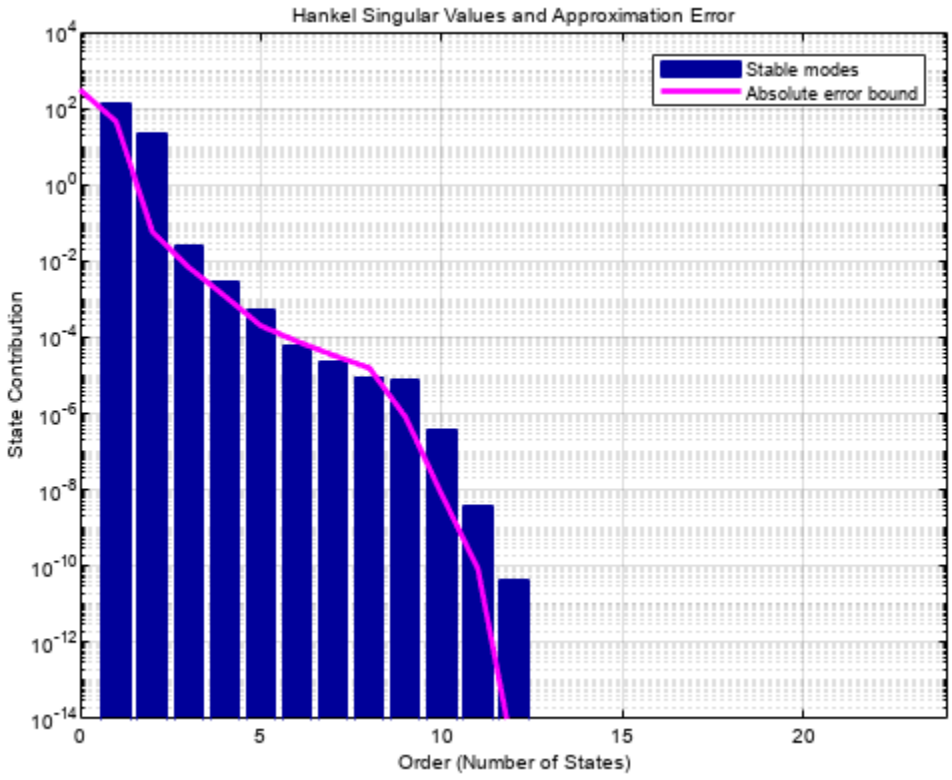
For this example, load a high-order model. `hplant` is a 23rd-order SISO model.

```
load ltiexamples hplant
order(hplant)
```

```
ans = 23
```

Examine the relative amount of energy per state in `hplant` using a Hankel singular-value (HSV) plot.

```
hsvplot(hplant)
```



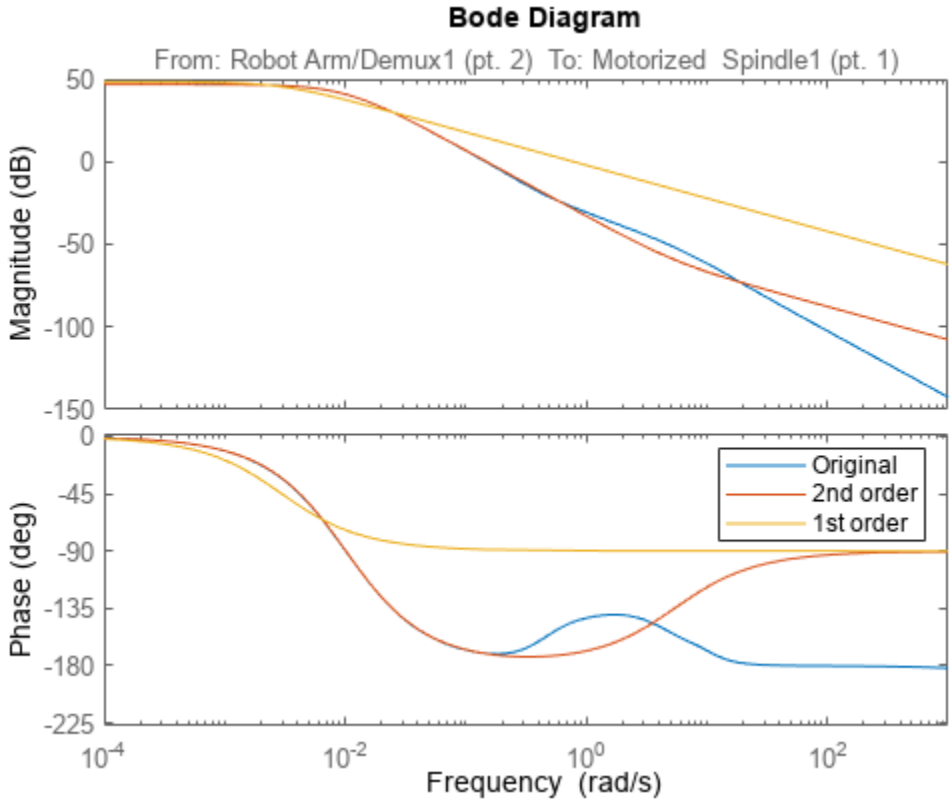
Small Hankel singular values indicate that the associated states contribute little to the behavior of the system. The plot shows that two states account for most of the energy in the system. Therefore, try simplifying the model to just first or second order.

```
opts = balredOptions('StateElimMethod','Truncate');
hplant1 = balred(hplant,1,opts);
hplant2 = balred(hplant,2,opts);
```

The second argument to `balred` specifies the target approximation order, so that `hplant1` is a first-order approximation and `hplant2` is a second-order approximation of `hplant`. By default, `balred` discards the states with the smallest Hankel singular values, and alters the remaining states to preserve the DC gain of the system. Setting the `StateElimMethod` option to `Truncate` causes `balred` to discard low-energy states without altering the remaining states.

When working with reduced-order models, it is important to verify that the approximation does not introduce inaccuracies at frequencies that are important for your application. Therefore, compare the frequency responses of the original and approximated systems. For MIMO systems, use the `sigmaplot` command. For this SISO system, examine a Bode plot.

```
bodeplot(hplant,hplant2,hplant1)
legend('Original','2nd order','1st order')
```

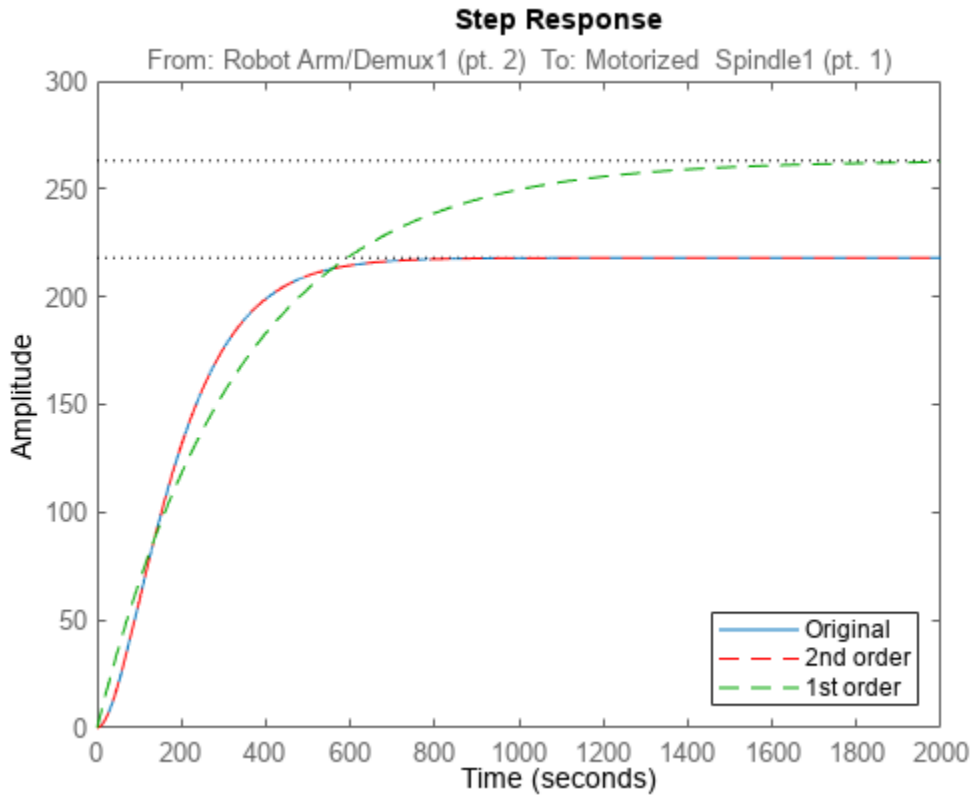


The second-order approximation `hplant2` matches the original 23rd-order system very well, especially at lower frequencies. The first-order system does not match as well.

In general, as you decrease the order of the approximated model, the frequency response of the approximated model begins to differ from the original model. Choose an approximation that is sufficiently accurate in the bands that are important to you. For example, in a control system you might want good accuracy inside the control bandwidth. Accuracy at frequencies far above the control bandwidth, where the gain rapidly rolls off, might be less important.

You can also validate the approximation in the time domain. For instance, examine the step responses of the original and reduced-order systems.

```
stepplot(hplant,hplant2,'r--',hplant1,'g--')
legend('Original','2nd order','1st order','Location','SouthEast')
```



This result confirms that the second-order approximation is a good match to the original 23rd-order system.

See Also

Functions
`balred` | `hsvplot`

Live Editor Tasks
Reduce Model Order

Related Examples

- “Balanced Truncation Model Reduction” on page 6-13
- “Compare Truncated and DC Matched Low-Order Model Approximations” on page 6-23
- “Approximate Model with Unstable or Near-Unstable Pole” on page 6-27
- “Frequency-Limited Balanced Truncation” on page 6-31

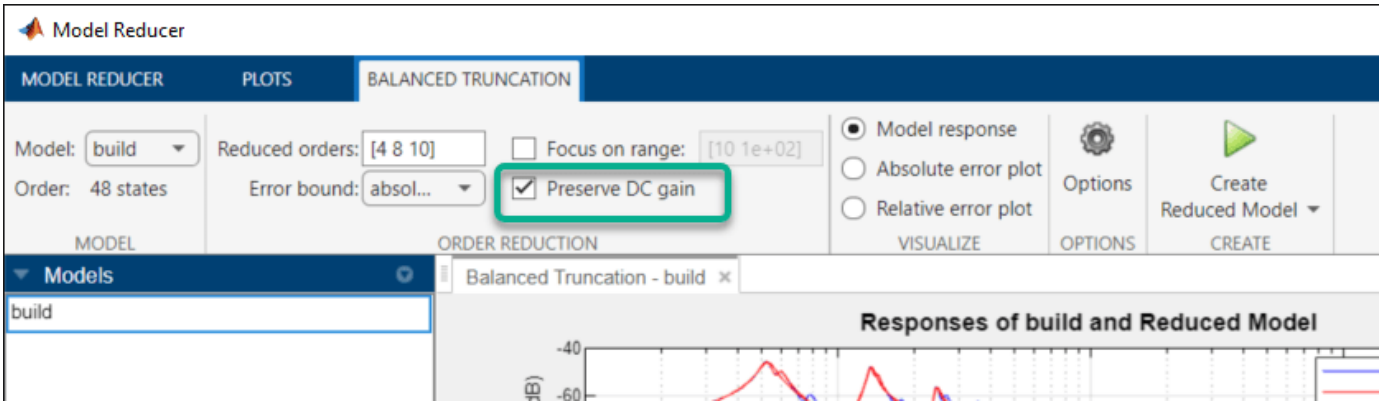
Compare Truncated and DC Matched Low-Order Model Approximations

This example shows how to compute a low-order approximation in two ways and compares the results. When you compute a low-order approximation by the balanced truncation method, you can either:

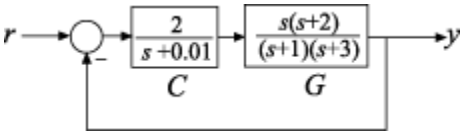
- Discard the states that make the smallest contribution to system behavior, altering the remaining states to preserve the DC gain of the system.
- Discard the low-energy states without altering the remaining states.

Which method you choose depends on what dynamics are most important to your application. In general, preserving DC gain comes at the expense of accuracy in higher-frequency dynamics. Conversely, state truncation can yield more accuracy in fast transients, at the expense of low-frequency accuracy.

This example compares the state-elimination methods of the `balred` command, `Truncate` and `MatchDC`. You can similarly control the state-elimination method in the **Model Reducer** app, on the **Balanced Truncation** tab, using the **Preserve DC Gain** check box, as shown.



Consider the following system.



Create a closed-loop model of this system from *r* to *y*.

```
G = zpk([0 -2],[-1 -3],1);
C = tf(2,[1 1e-2]);
T = feedback(G*C,1)
```

```
T =

              2 s (s+2)
  --------------------------------
  (s+0.004277) (s+1.588) (s+4.418)
```

```
Continuous-time zero/pole/gain model.
```

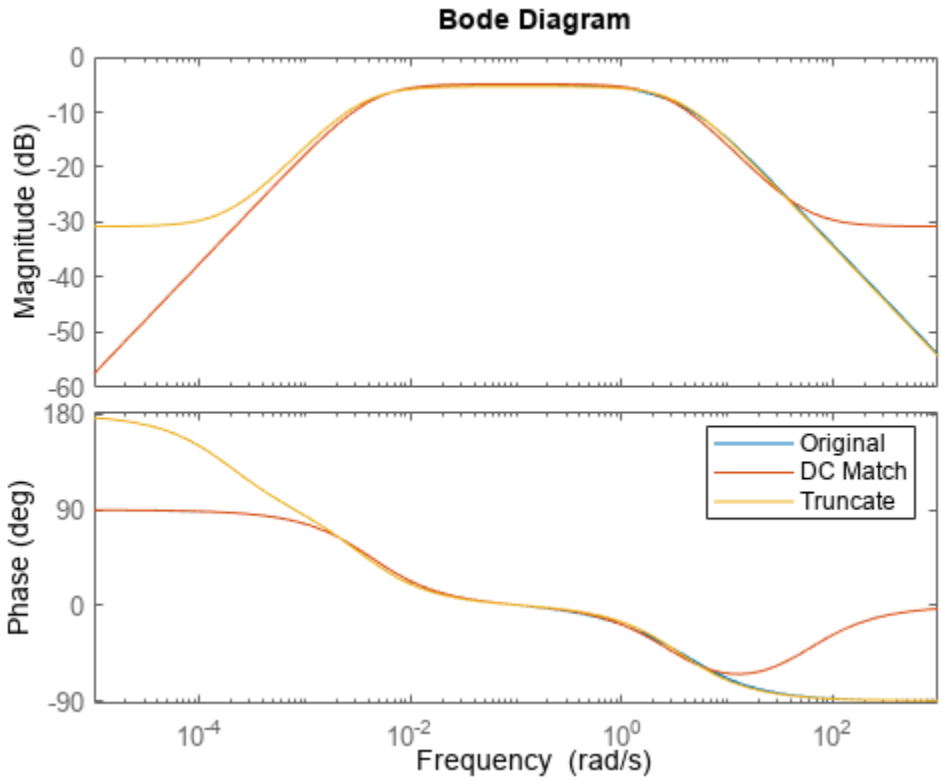
T is a third-order system that has a pole-zero near-cancellation close to $s = 0$. Therefore, it is a good candidate for order reduction by approximation.

Compute two second-order approximations to T, one that preserves the DC gain and one that truncates the lowest-energy state without changing the other states. Use `balredOptions` to specify the approximation methods, `MatchDC` and `Truncate`, respectively.

```
matchopt = balredOptions('StateProjection','MatchDC');
truncopt = balredOptions('StateProjection','Truncate');
Tmatch = balred(T,2,matchopt);
Ttrunc = balred(T,2,truncopt);
```

Compare the frequency responses of the approximated models.

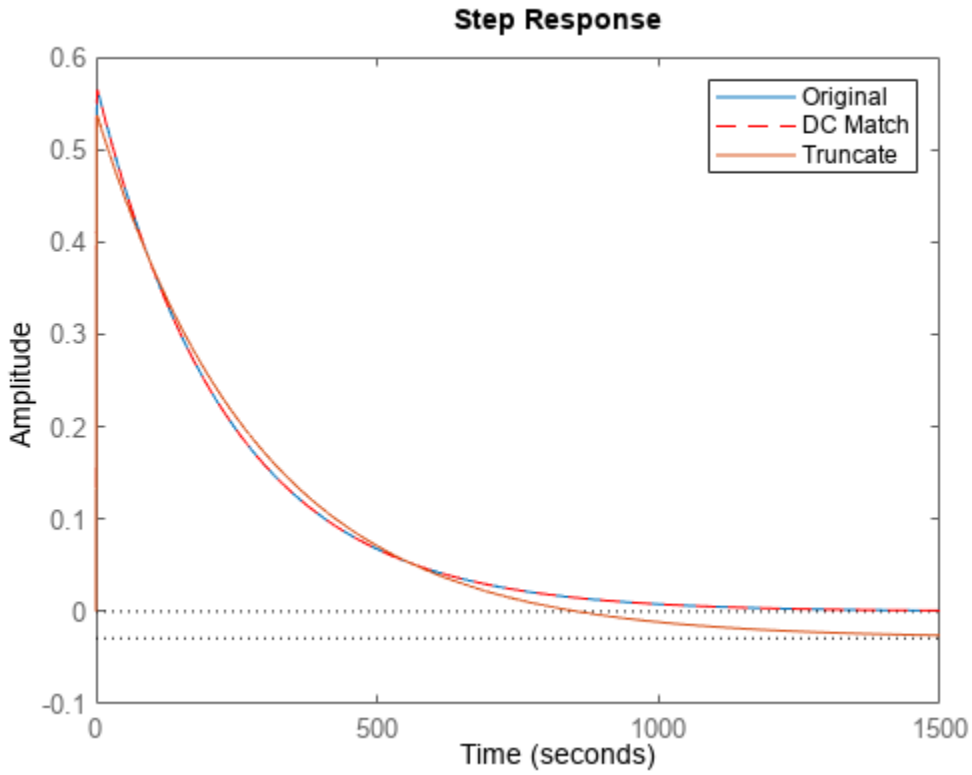
```
bodeplot(T,Tmatch,Ttrunc)
legend('Original','DC Match','Truncate')
```



The truncated model `Ttrunc` matches the original model well at high frequencies, but differs considerably at low frequency. Conversely, `Tmatch` yields a good match at low frequencies as expected, at the expense of high-frequency accuracy.

You can also see the differences between the two methods by examining the time-domain response in different regimes. Compare the slow dynamics by looking at the step response of all three models with a long time horizon.

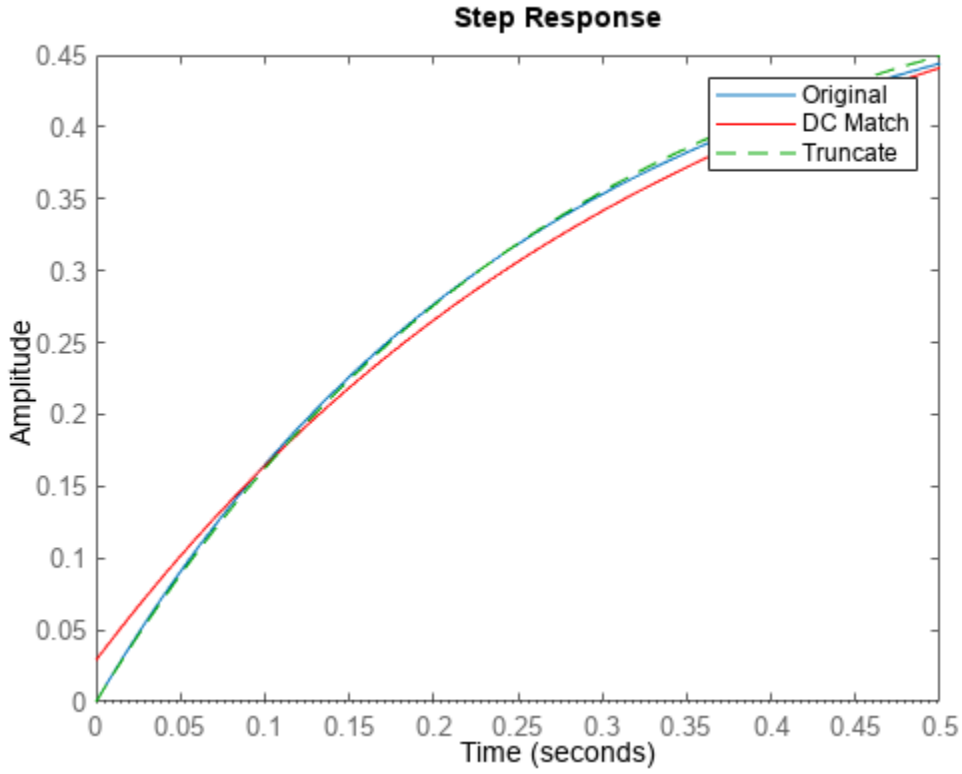
```
stepplot(T,Tmatch,'r--',Ttrunc,1500)
legend('Original','DC Match','Truncate')
```



As expected, on long time scales the DC-matched approximation `Tmatch` has a very similar response to the original model.

Compare the fast transients in the step response.

```
stepplot(T,Tmatch,'r',Ttrunc,'g--',0.5)
legend('Original','DC Match','Truncate')
```



On short time scales, the truncated approximation `Ttrunc` provides a better match to the original model. Which approximation method you should use depends on which regime is most important for your application.

See Also

Functions
`balred` | `hsvplot`

Live Editor Tasks
Reduce Model Order

Related Examples

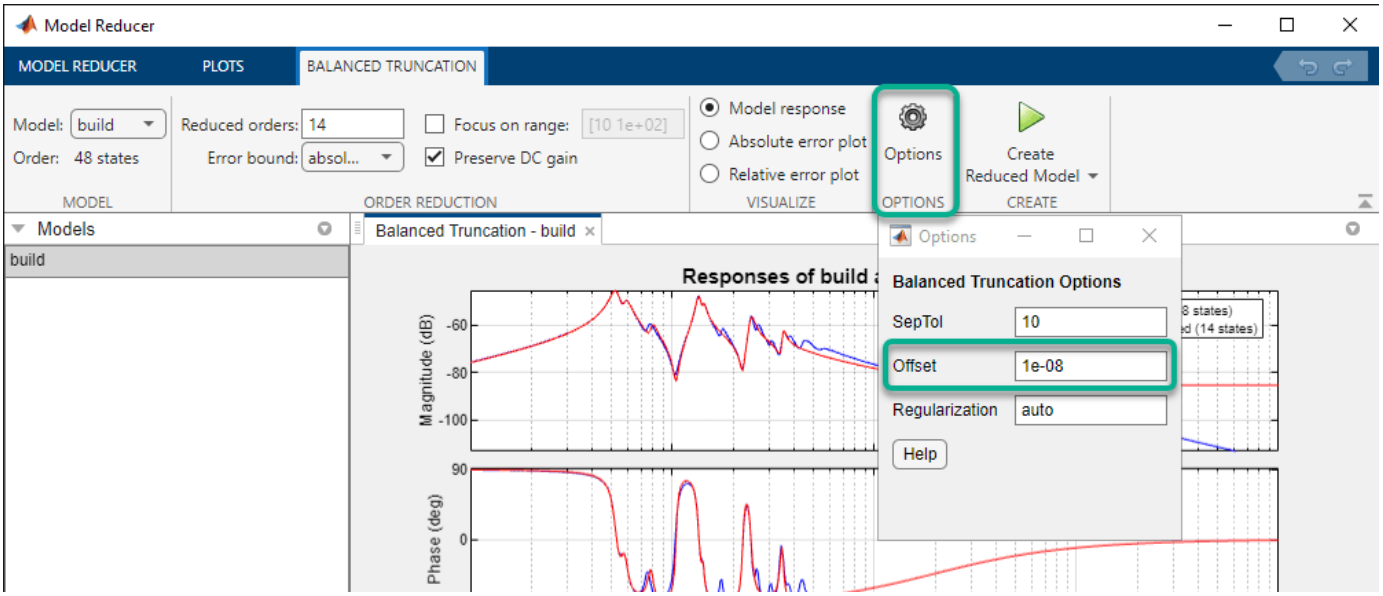
- “Balanced Truncation Model Reduction” on page 6-13
- “Approximate Model with Unstable or Near-Unstable Pole” on page 6-27
- “Frequency-Limited Balanced Truncation” on page 6-31

Approximate Model with Unstable or Near-Unstable Pole

This example shows how to compute a reduced-order approximation of a system when the system has unstable or near-unstable poles.

When computing a reduced-order approximation, the `balred` command (or the **Model Reducer** app) does not eliminate unstable poles because doing so would fundamentally change the system dynamics. Instead, the software decomposes the model into stable and unstable parts and reduces the stable part of the model.

If your model has near-unstable poles, you might want to ensure that the reduced-order approximation preserves these dynamics. This example shows how to use the `Offset` option of `balred` to preserve poles that are close to the stable-unstable boundary. You can achieve the same result in the **Model Reducer** app, on the **Balanced Truncation** tab, under **Options**, using the **Offset** field, as shown:

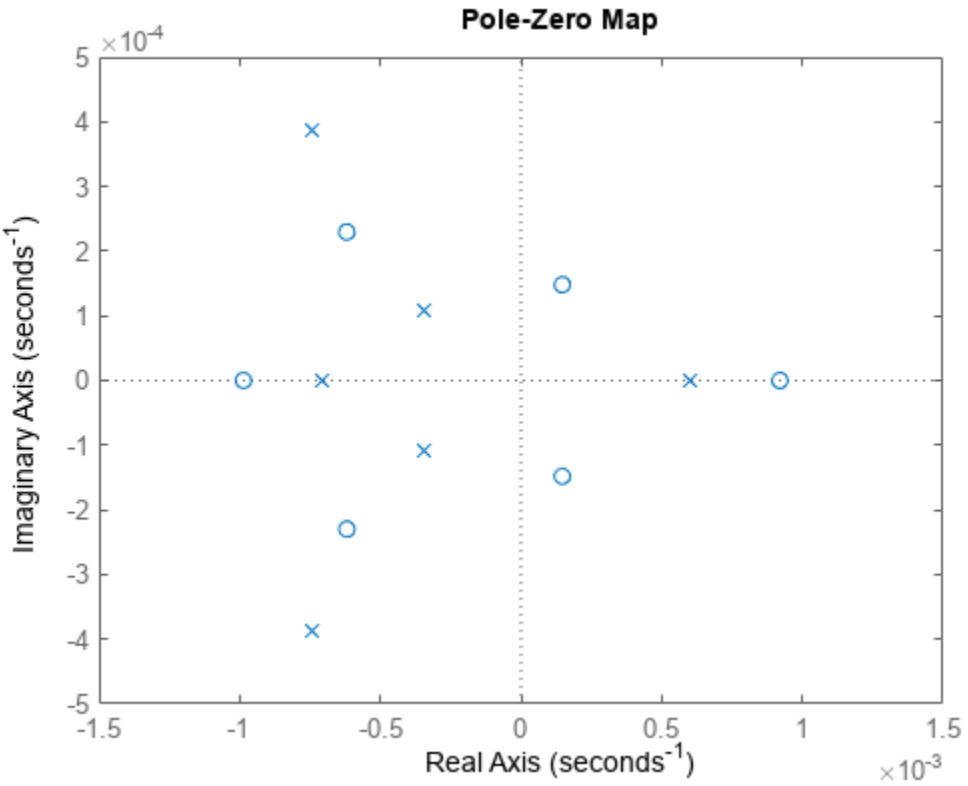


Load a model with unstable and near-unstable poles.

```
load('reduce.mat','gasf35unst')
```

`gasf35unst` is a 25-state SISO model with two unstable poles (Re(s) > 0). Examine the system poles to find the near-unstable poles.

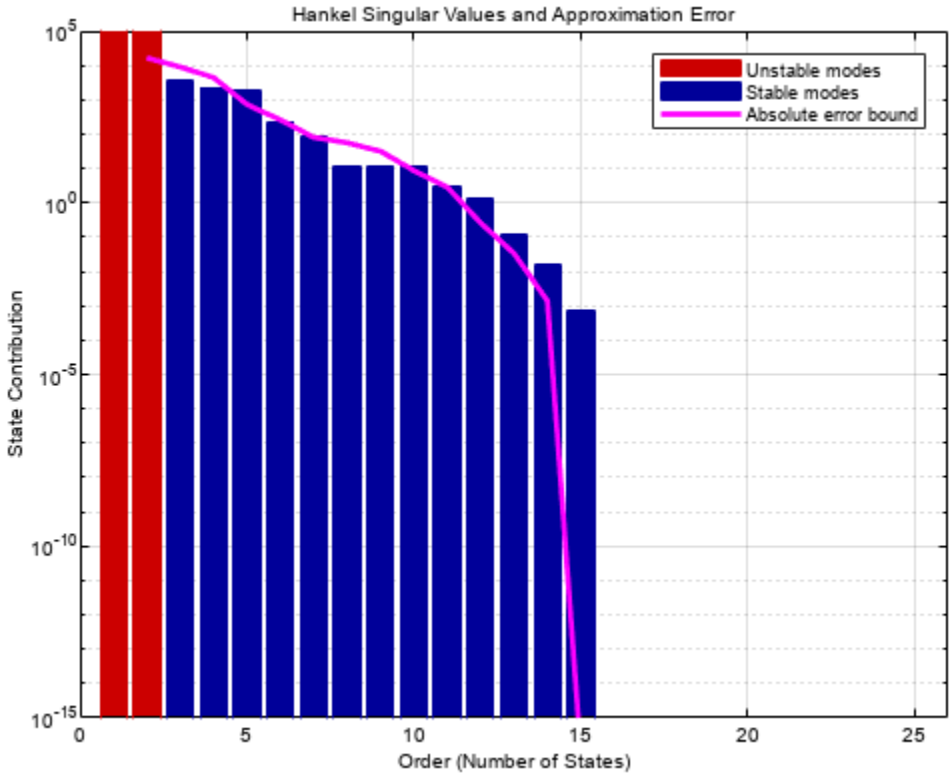
```
pzplot(gasf35unst)
axis([-0.0015 0.0015 -0.0005 0.0005])
```



The pole-zero plot shows several poles (marked by x) that fall in the left half-plane, but relatively close to the imaginary axis. These are the near-unstable poles. Two of these fall within 0.0005 of instability. Three more fall within 0.001 of instability.

Examine a Hankel singular-value plot of the model.

```
hsvplot(gasf35unst)
```



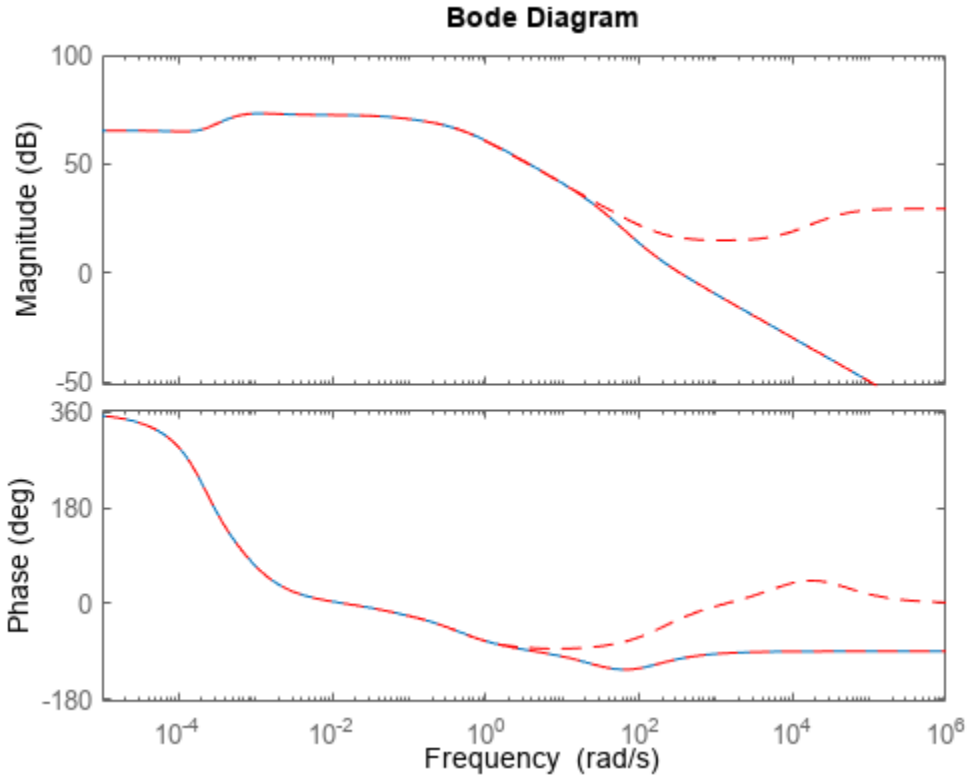
The plot shows the two unstable modes, but you cannot easily determine the energy contribution of the near-unstable poles. In your application, you might want to reduce the model without discarding those poles nearest to instability, even if they are of relatively low energy. Use the `Offset` option of `balred` to calculate a reduced-order system that preserves the two stable poles that are closest to the imaginary axis. The `Offset` option sets the boundary between poles that `balred` can discard, and poles that `balred` must preserve (treat as unstable).

```
opts = balredOptions('Offset',0.0005);
gasf_arr = balred(gasf35unst,[10 15],opts);
```

Providing `balred` an array of target approximation orders `[10 15]` causes `balred` to return an array of approximated models. The array `gasf_arr` contains two models, a 10th-order and a 15th-order approximation of `gasf35unst`. In both approximations, `balred` does not discard the two unstable poles or the two nearly-unstable poles.

Compare the reduced-order approximations to the original model.

```
bodeplot(gasf35unst,gasf_arr,'r--')
```



The 15th order approximation is a good frequency-domain match to the original model. However, the 10th-order approximation shows changes in high-frequency dynamics, which might be too large to be acceptable. The 15th-order approximation is likely a better choice.

See Also

Functions
`balred` | `hsvplot`

Live Editor Tasks
Reduce Model Order

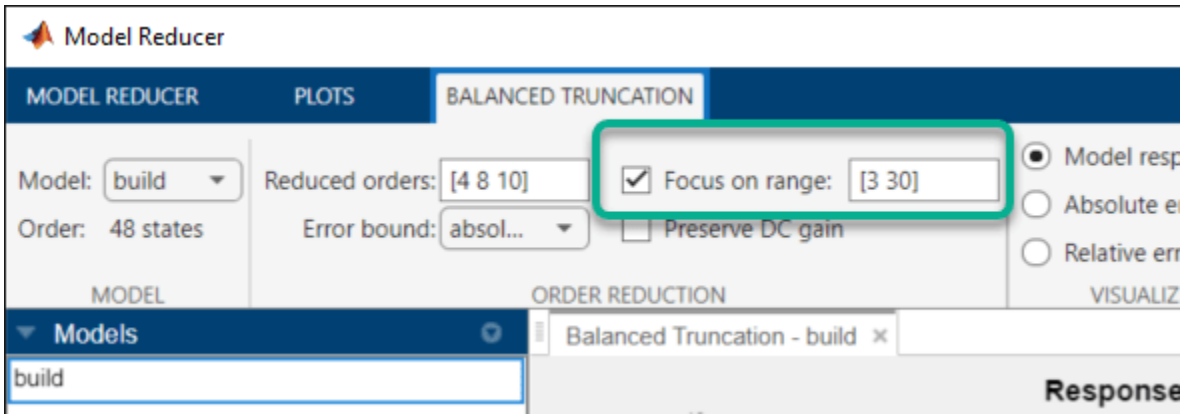
Related Examples

- “Balanced Truncation Model Reduction” on page 6-13
- “Approximate Model by Balanced Truncation at the Command Line” on page 6-20
- “Frequency-Limited Balanced Truncation” on page 6-31

Frequency-Limited Balanced Truncation

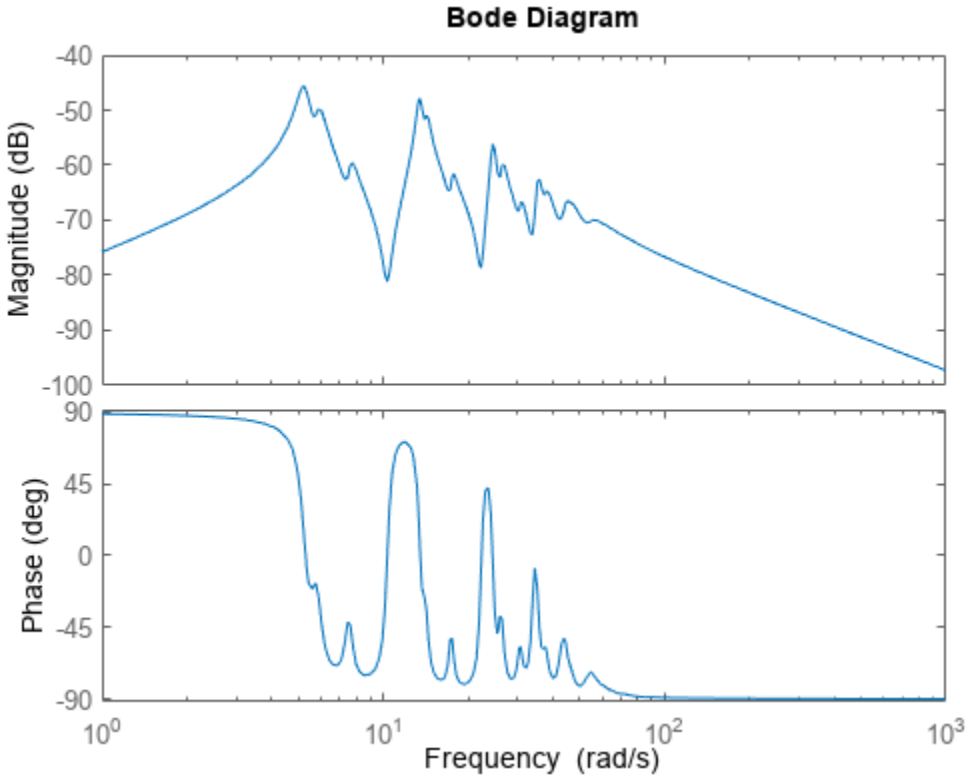
This example shows how to reduce a high-order model by removing states of relatively low energy within a particular frequency interval. Focusing the energy-contribution calculation on a particular frequency region sometimes yields a good approximation to the dynamics of interest at a lower order than a reduction that takes all frequencies into account.

This example demonstrates frequency-limited balanced truncation at the command line, using options for the `balred` command. You can also perform frequency-limited balanced truncation in the **Model Reducer** app, on the **Balanced Truncation** tab, using the **Focus on range** check box, as shown.



Load a model and examine its frequency response.

```
load('building.mat','G')
bodeplot(G)
```



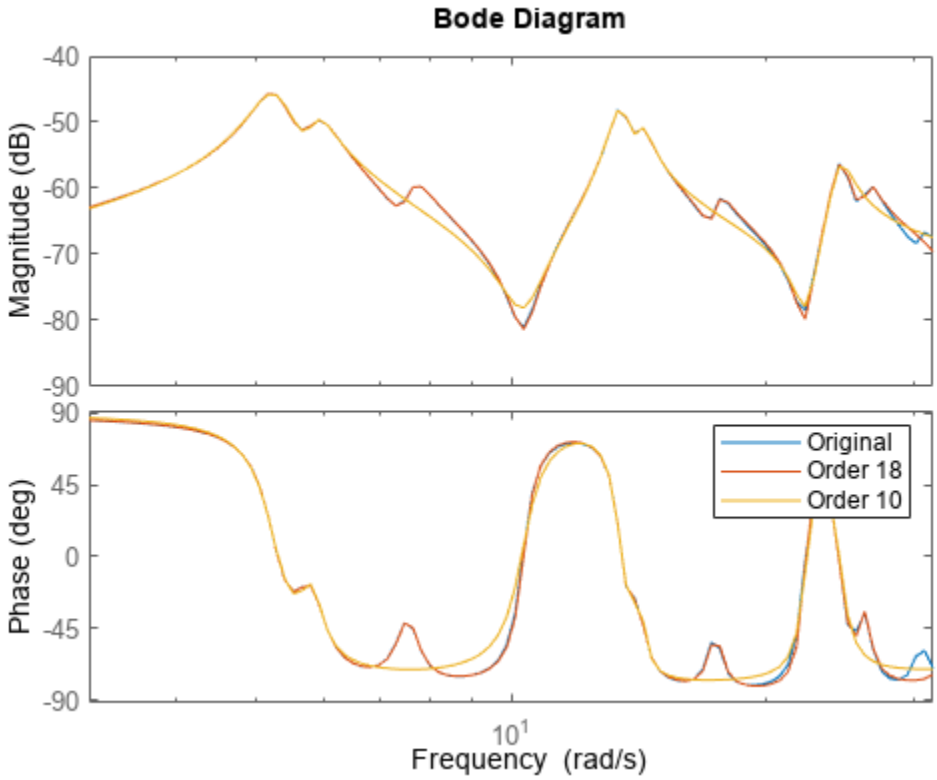
`G` is a 48th-order model with several large peak regions around 5.2 rad/s, 13.5 rad/s, and 24.5 rad/s, and smaller peaks scattered across many frequencies. Examine the Hankel singular-value plot to see the energy contributions of the model's 48 states.

```
hsvd(G)
```

```
ans = 48×1

    0.0025
    0.0024
    0.0019
    0.0019
    0.0007
    0.0007
    0.0006
    0.0006
    0.0004
    0.0004
      ⋮
```

The singular-value plot suggests that you can discard at least 20 states without significant impact on the overall system response. Suppose that for your application you are only interested in the dynamics near the second large peak, between 10 rad/s and 22 rad/s. Try a few reduced-model orders based on the Hankel singular value plot. Compare their frequency responses to the original model, especially in the region of that peak.

```
G18 = balred(G,18);
G10 = balred(G,10);

bodeplot(G,G18,G10,logspace(0.5,1.5,100));
legend('Original','Order 18','Order 10');
```



The 18th-order model is a good match to the dynamics in the region of interest. In the 10th order model, however, there is some degradation of the match.

Focus the model reduction on the region of interest to obtain a good match with a lower-order approximation. First, examine the state energy contributions in that frequency region only. Use `hsvdOptions` to specify the frequency interval for `hsvd`.

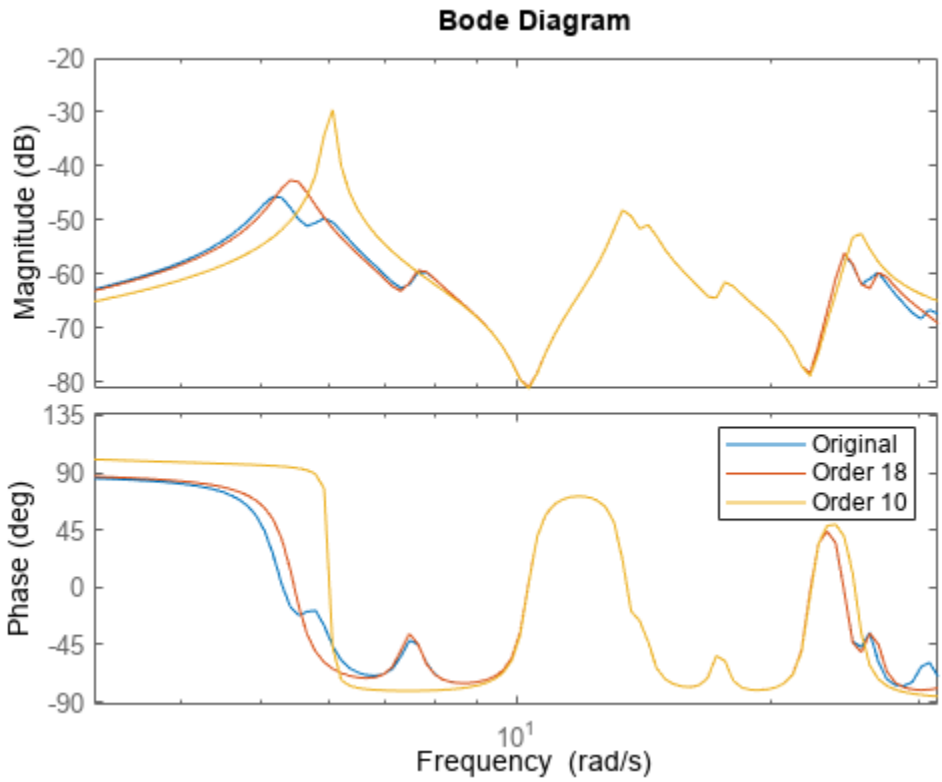
```
hopt = hsvdOptions('FreqIntervals',[10,22]);
hsvd(G,hopt)
```

```
ans = 48×1

    0.0018
    0.0018
    0.0004
    0.0004
    0.0002
    0.0002
    0.0001
    0.0001
    0.0000
    0.0000
```

Comparing this plot to the previous Hankel singular-value plot shows that in this frequency region, many fewer states contribute significantly to the dynamics than contribute to the overall dynamics.

Try the same reduced-model orders again, this time choosing states to eliminate based only on their contribution to the frequency interval. Use `balredOptions` to specify the frequency interval for `balred`.

```
bopt = balredOptions('StateProjection','Truncate','FreqIntervals',[10,22]);
GLim18 = balred(G,18,bopt);
GLim10 = balred(G,10,bopt);

bodeplot(G,GLim18,GLim10,logspace(0.5,1.5,100));
legend('Original','Order 18','Order 10');
```



With the frequency-limited energy computation, a 10th-order approximation is as good in the region of interest as the 18th-order approximation computed without frequency limits.

See Also

Functions
`balred` | `hsvplot`

Live Editor Tasks
Reduce Model Order

Related Examples

- “Balanced Truncation Model Reduction” on page 6-13
- “Approximate Model by Balanced Truncation at the Command Line” on page 6-20

Model Reduction in the Live Editor

This example shows how to use the Reduce Model Order task in the Live Editor to generate code for performing model reduction by balanced truncation, mode selection, and pole-zero simplification. The **Reduce Model Order** task lets you interactively compute reduced-order approximations of high-order models while preserving model characteristics that are important to your application.

Open this example to see a preconfigured script containing the **Reduce Model Order** task. For more information about Live Editor tasks generally, see “Add Interactive Tasks to a Live Script”.

In the Live Editor, load the model you want to reduce into the MATLAB® workspace.

```
load build G
size(G)
```

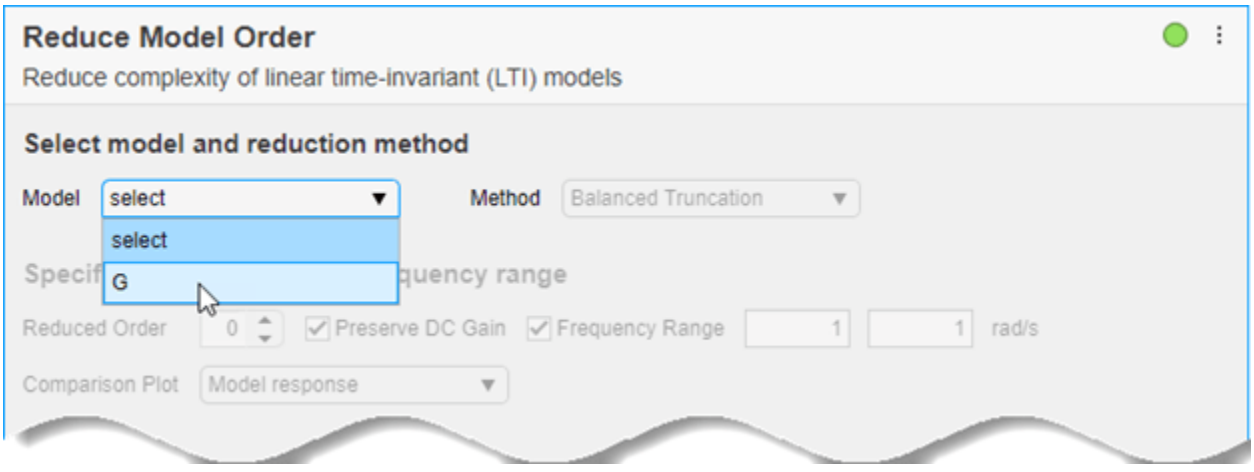
```
State-space model with 1 outputs, 1 inputs, and 48 states.
```

The LTI model `G` is a state-space model with 48 states. Some of these states can be discarded while preserving relevant dynamics. To experiment with reducing the model, open the **Reduce Model Order** Live Editor task. On the **Live Editor** tab, select **Task** > **Reduce Model Order**.

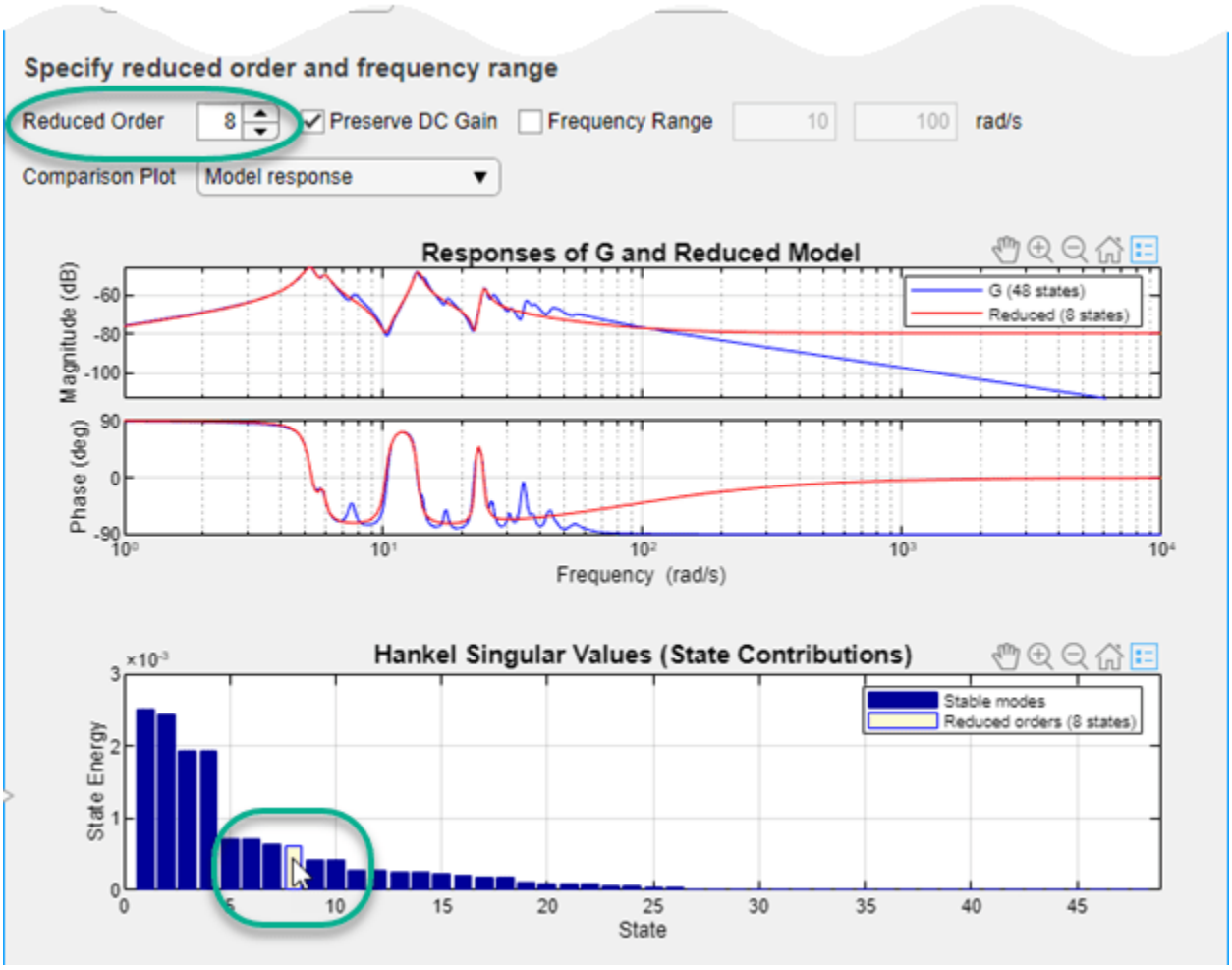
Balanced Truncation

Balanced truncation computes a lower-order approximation of your model by removing states with relatively small energy contributions. For more information about balanced truncation, see “Balanced Truncation Model Reduction” on page 6-13.

To generate code for balanced truncation in the **Reduce Model Order** task, select `G` as the model to reduce. Specify `Balanced Truncation` as the **Method**.

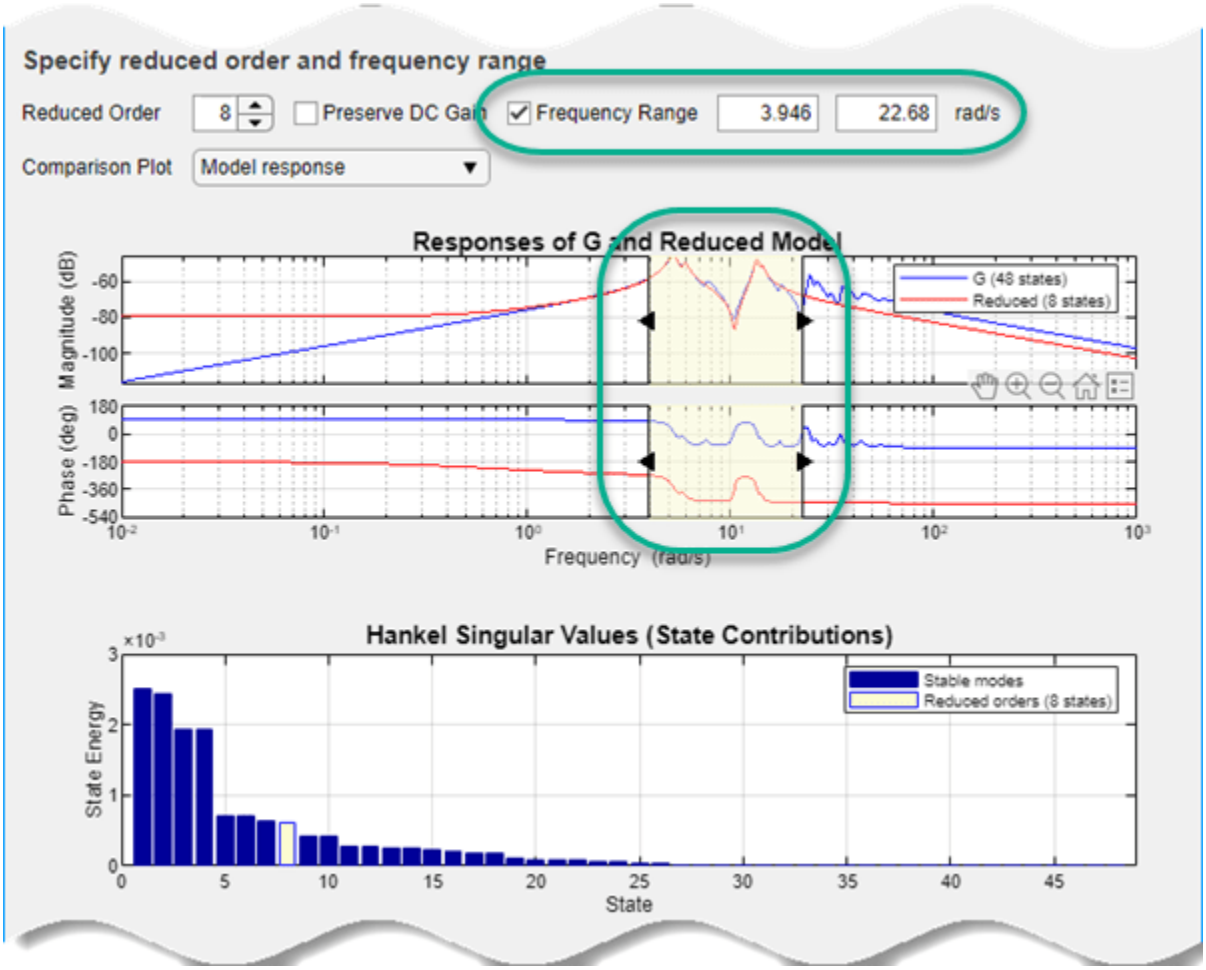


Specify the target **Reduced Order** of the reduced model. You can use the *Hankel singular-value plot* to help select a target order. This plot visualizes the relative energy contribution of each state in the original model. The task discards states with lower energy than the state you select in this plot.

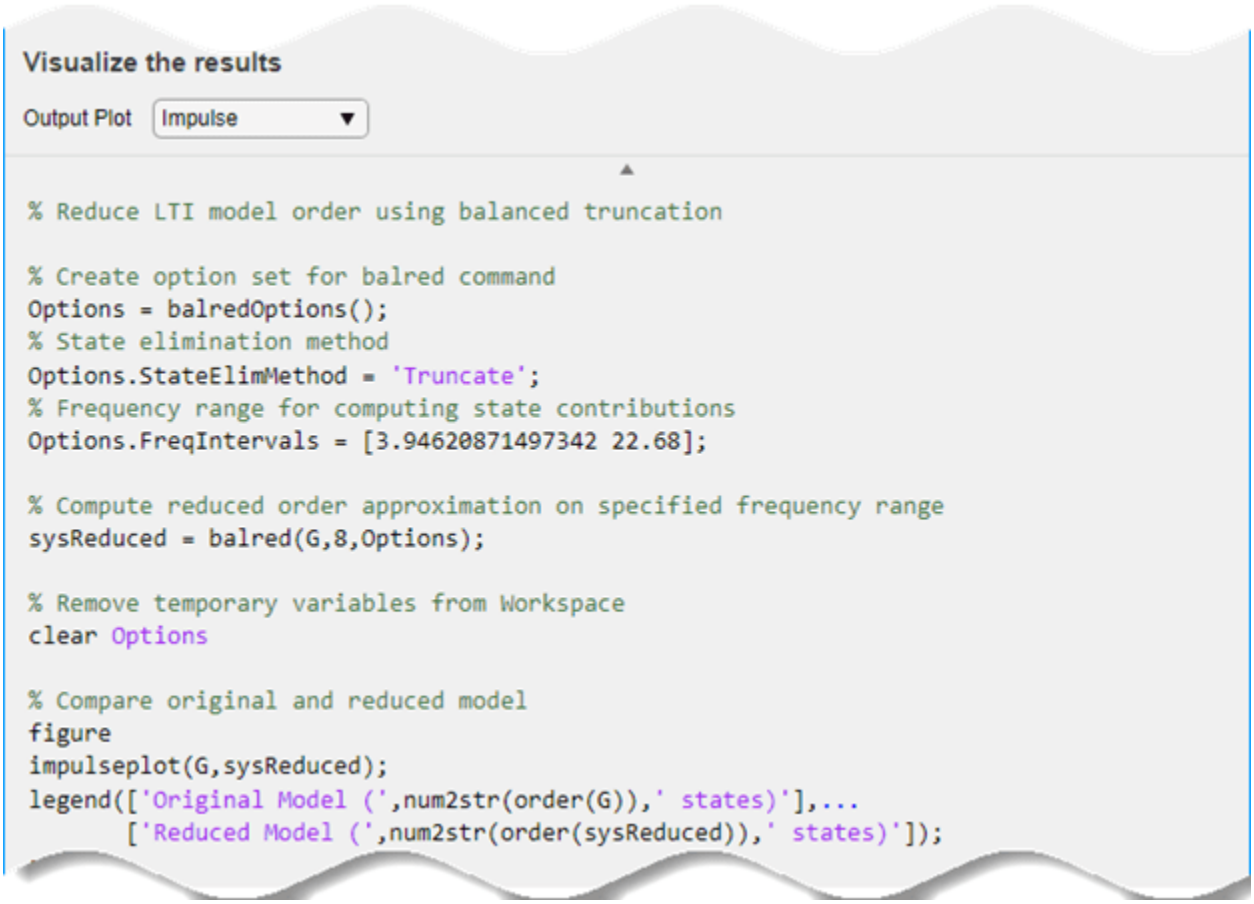


The response plot shows the Bode plot of the original model and the reduced-order model. Experiment with different reduced-order models by selecting different orders on the Hankel singular-value plot, using the response plot to observe how the reduced-order model changes. (To see the absolute or relative error between the original and reduced model, use the **Model Response** menu.)

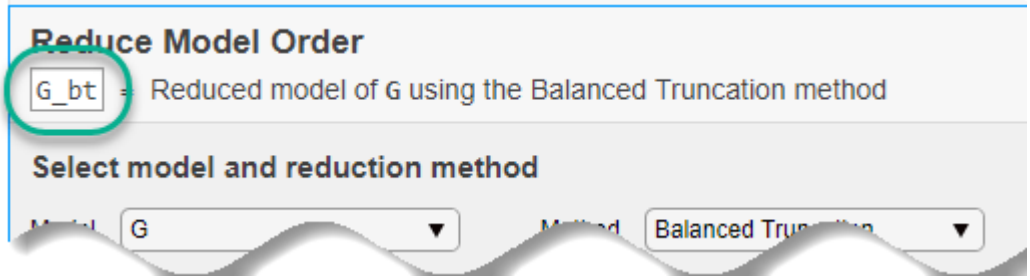
Select the smallest model order that adequately preserves the dynamics that are important to your application. If you are only interested in dynamics in a specific frequency range, you can restrict the computation of energy contributions to that range. To do, select **Frequency Range**. Then, enter the minimum and maximum frequencies or specify the range using the vertical sliders on the response plot. (Selecting **Frequency Range** clears the **Preserve DC Gain** option, because 0 rad/s is not within the default frequency range.)



The task generates the code to compute the reduced model that you have specified. To see the generated code, click ▼ at the bottom of the task. The task expands to show the generated code. For balanced truncation, **Reduce Model Order** uses `balred` with options specified by `balredOptions`. (To include code that produces a response plot, in the **Output Plot** menu, select the response you want.)



By default, the generated code uses `sysReduced` as the name of the output variable. To specify a different output variable name, enter a new name in the summary line at the top of the task.

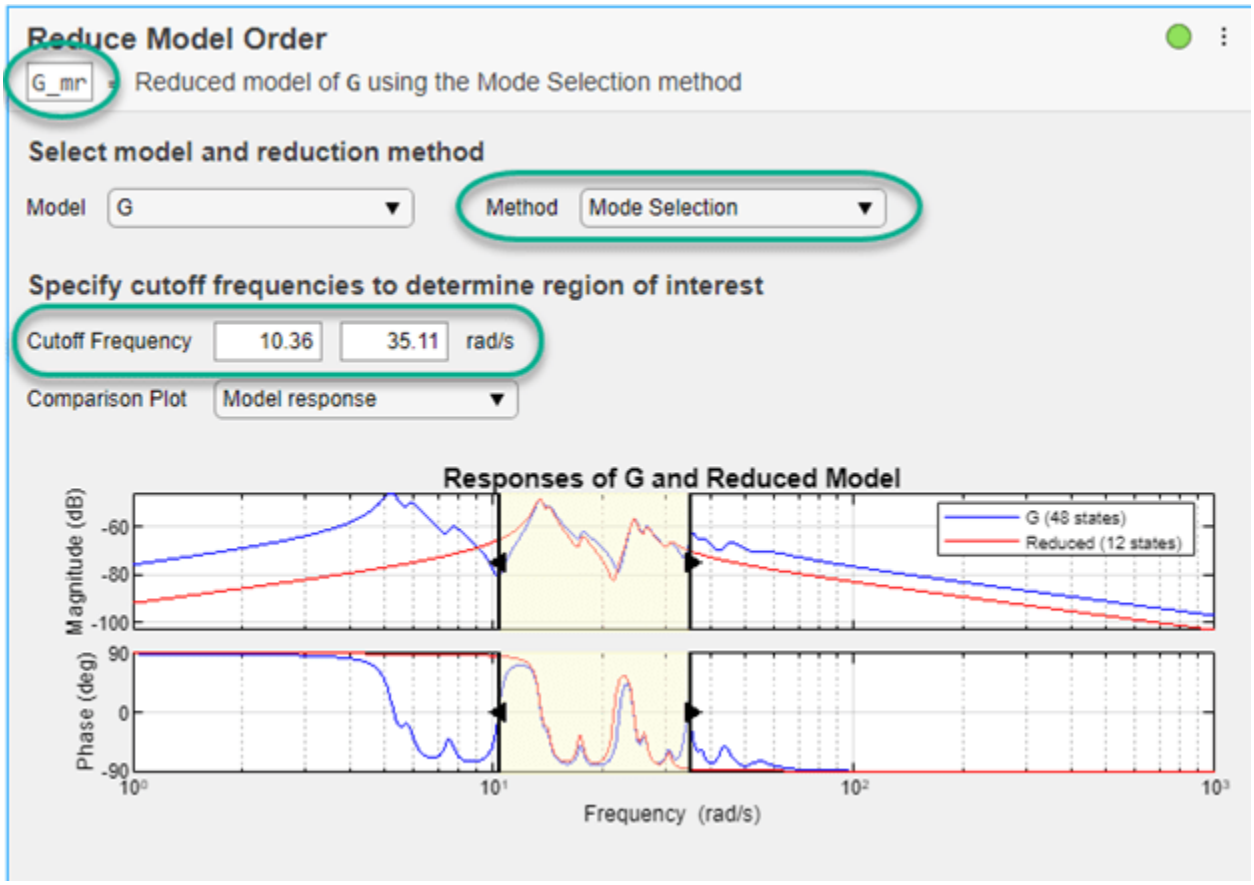


The task updates the generated code to reflect the new variable, and a reduced-order state-space model appears in the MATLAB® workspace with the new name.

Mode Selection

You can use the reduced-order model in the MATLAB workspace as you would use any other LTI model for analysis and control design. For this example, compare the reduced model responses to a reduced-order model created using a different model-reduction method, *mode selection*. Mode selection reduces the model by discarding dynamics that fall outside a frequency region you specify.

Change the name of the output variable to `G_mr`, so you do not overwrite the model you created using balanced truncation. Then, set the reduction method to `Mode Selection`. Use the vertical sliders on the plot to choose the frequency range within which to preserve dynamics, or enter the frequency range in **Cutoff Frequency**.



For mode selection, Reduce Model Order generates code that uses `freqsep`. It can be useful to add a pole-zero plot to observe which poles and zeroes are eliminated from the reduced-order model. To do so, in the **Output Plot** menu, select `Pole-Zero`. This selection generates a plot that shows the poles and zeros of both the original and reduced model. It also adds the code for creating that plot to the generated code.

Pole-Zero Simplification

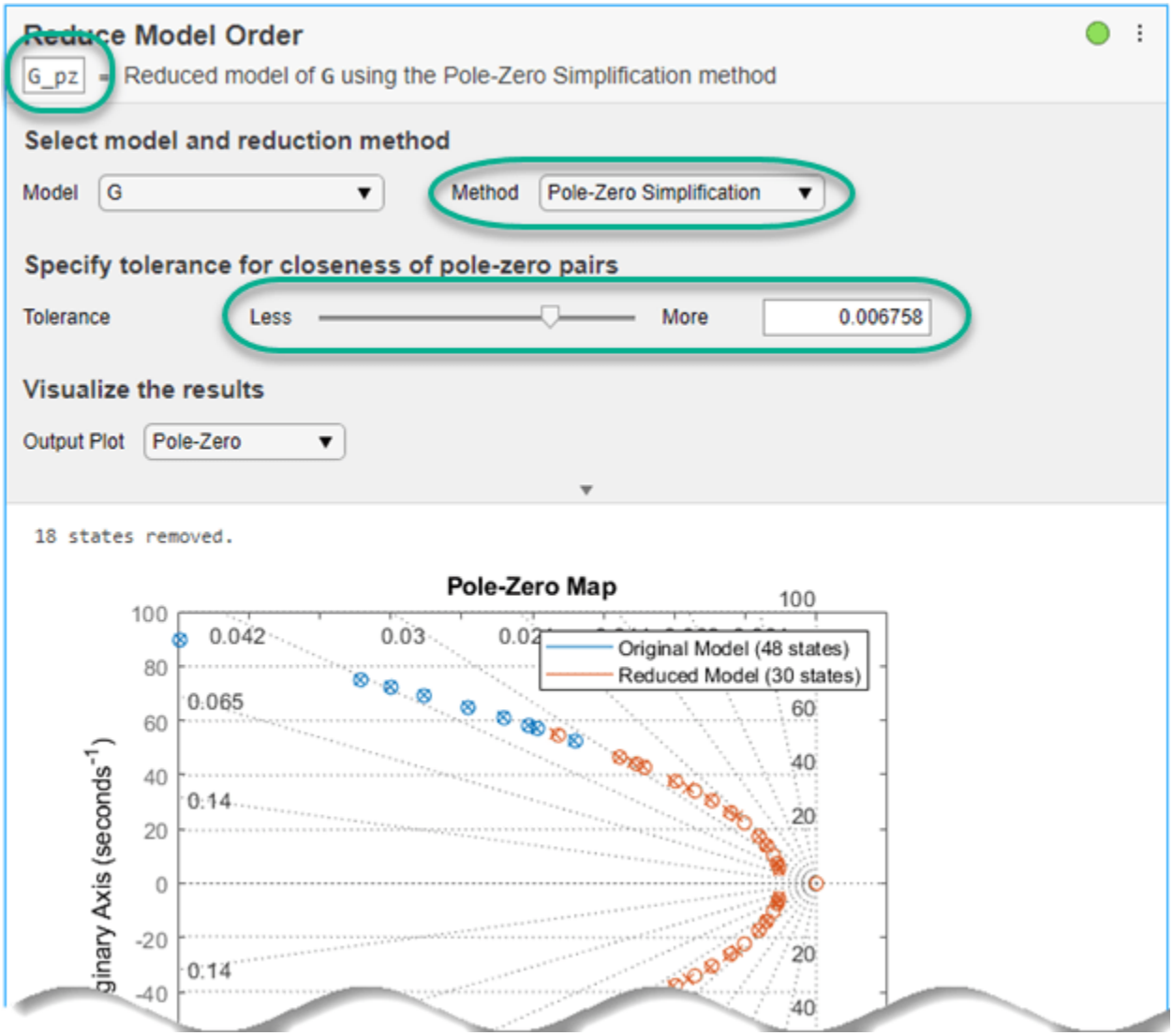
Pole-zero simplification reduces the order of your model exactly by canceling pole-zero pairs or eliminating states that have no effect on the overall model response. Pole-zero pairs can be introduced, for example, when you construct closed-loop architectures. Normal small errors associated with numerical computation can convert such canceling pairs to near-canceling pairs. Removing these states preserves the model response characteristics while simplifying analysis and control design.

The Pole-Zero Simplification method of **Reduce Model Order** automatically eliminates:

- Canceling or near-canceling pole-zero pairs from transfer functions
- Unobservable or uncontrollable states from state-space models

- States that are structurally disconnected from the inputs or outputs.

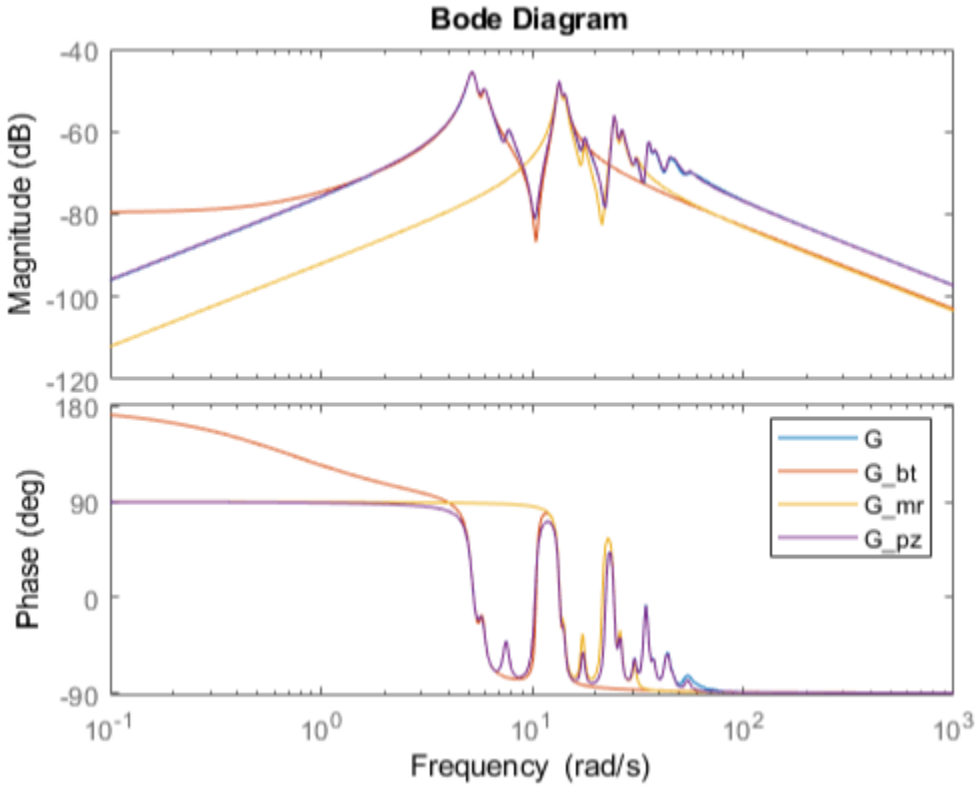
Change the name of the output variable to `G_pz`, so you do not overwrite the model you created using the other methods. Then, set the reduction method to `Pole-Zero Cancellation`. Use the **Tolerance** parameter to adjust how close to canceling pole-zero pairs must be to be eliminated. Move the slider toward **More** to cancel more pole-zero pairs, reducing the model to a smaller order. In **Output Plot**, select `Pole-Zero` and use the plot to observe which poles and zeros are eliminated from the reduced model.



Use Reduced-Order Models

The **Reduce Model Order** task creates the reduced-order models in the MATLAB workspace automatically. You can use those models for further analysis or control design as you would use any other LTI model. For instance, compare the frequency responses of the original and all reduced-order models on one plot.

```
bode(G,G_bt,G_mr,G_pz,{0.1,1000})
legend
```



See Also

Apps
Model Reducer

Live Editor Tasks
Reduce Model Order

Functions
`balred`

More About

- “Balanced Truncation Model Reduction” on page 6-13

Pole-Zero Simplification

Pole-zero simplification reduces the order of your model exactly by canceling pole-zero pairs or eliminating states that have no effect on the overall model response. Pole-zero pairs can be introduced, for example, when you construct closed-loop architectures. Normal small errors associated with numerical computation can convert such canceling pairs to near-canceling pairs. Removing these states preserves the model response characteristics while simplifying analysis and control design. Types of pole-zero simplification include:

- Structural elimination — Eliminate states that are structurally disconnected from the inputs or outputs. Eliminating structurally disconnected states is a good first step in model reduction because the process does not involve any numerical computation. It also preserves the state structure of the remaining states. Such structurally nonminimal states can arise, for example, when you linearize a Simulink model that includes some unconnected state-space or transfer function blocks. At the command line, perform structural elimination with `sminreal`.
- Pole-zero cancellation or minimal realization — Eliminate canceling or near-canceling pole-zero pairs from transfer functions. Eliminate unobservable or uncontrollable states from state-space models. At the command line, perform this kind of simplification with `minreal`.

In the **Model Reducer** app and the **Reduce Model Order** Live Editor task, the **Pole-Zero Simplification** method automatically eliminates structurally disconnected states and also performs pole-zero cancellation or minimal realization.

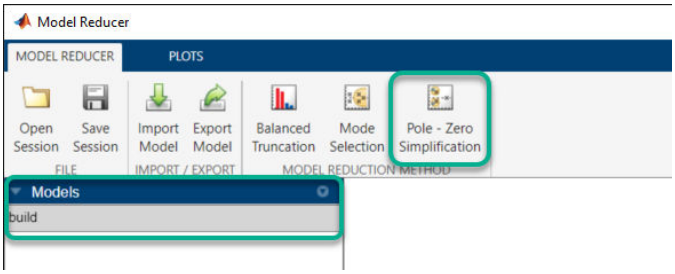
Pole-Zero Simplification in the Model Reducer App

Model Reducer provides an interactive tool for performing model reduction and examining and comparing the responses of the original and reduced-order models. To reduce a model by pole-zero simplification in **Model Reducer**:

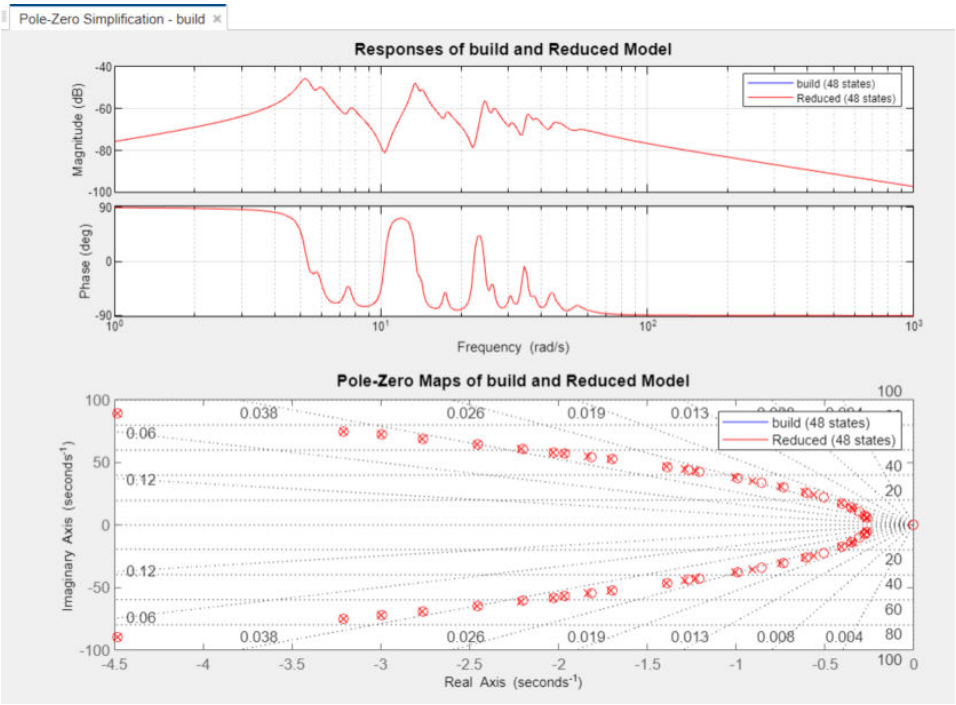
1. Open the app and import a model to reduce. For instance, suppose that there is a model named `build` in the MATLAB workspace. The following command opens **Model Reducer** and imports the LTI model `build`.

   ```
   modelReducer(build)
   ```

2. In the data browser, select the model to reduce. Click **Pole-Zero Simplification**.



   In the **Pole-Zero Simplification** tab, **Model Reducer** displays a plot of the frequency response of the original model and a reduced version of the model. The app also displays a pole-zero map of both models.

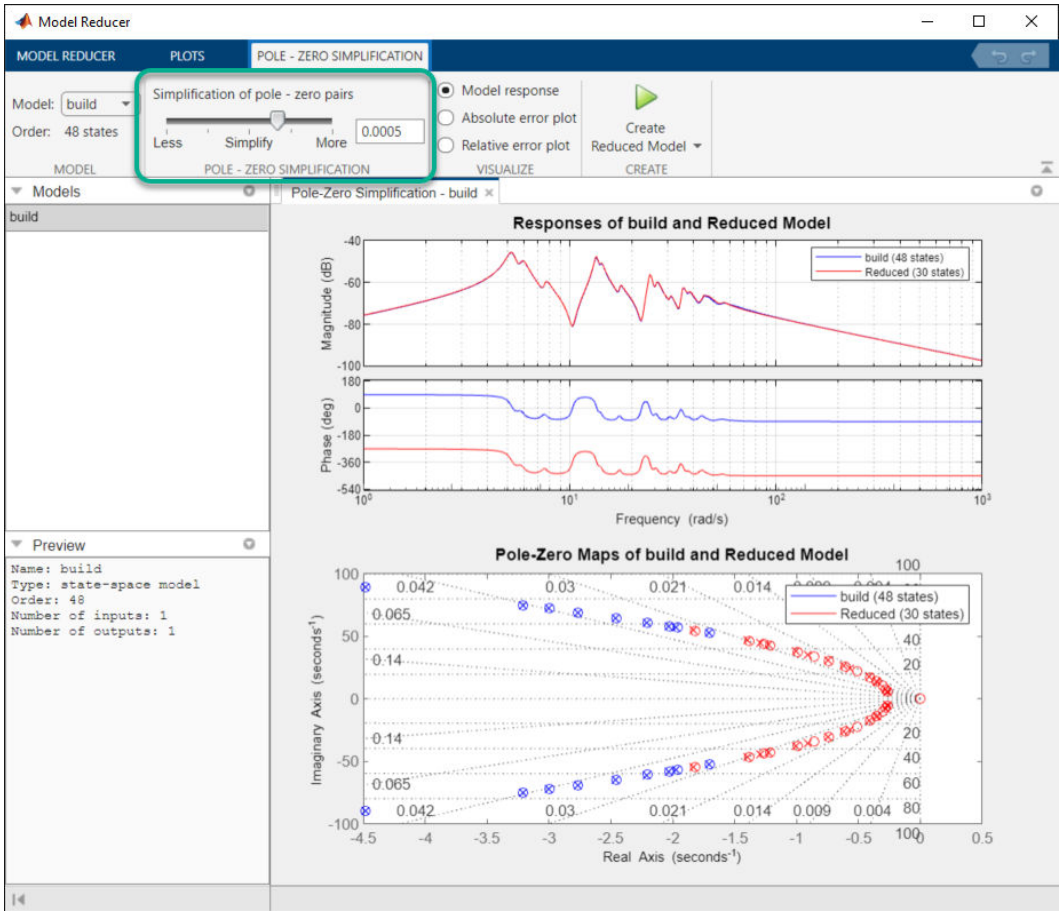


The pole-zero map marks pole locations with `x` and zero locations with `o`.

Note The frequency response is a Bode plot for SISO models, and a singular-value plot for MIMO models.

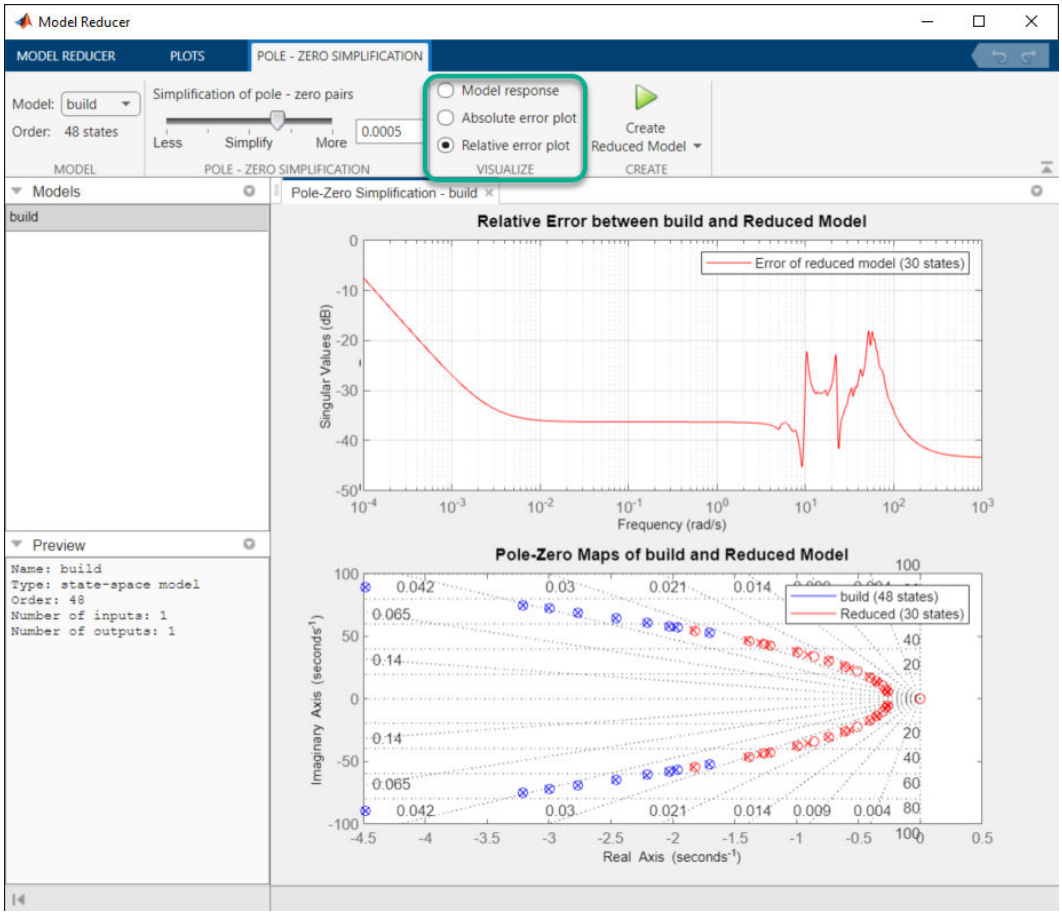
---

3 Optionally, change the tolerance with which **Model Reducer** identifies canceling pole-zero pairs. **Model Reducer** cancels pole-zero pairs that fall within the tolerance specified by the **Simplification of pole-zero pairs** value. In this case, no pole-zero pairs are close enough together for **Model Reducer** to cancel them at the default tolerance of `1e-05`. To cancel pairs that are a little further apart, move the slider to the right or enter a larger value in the text box.



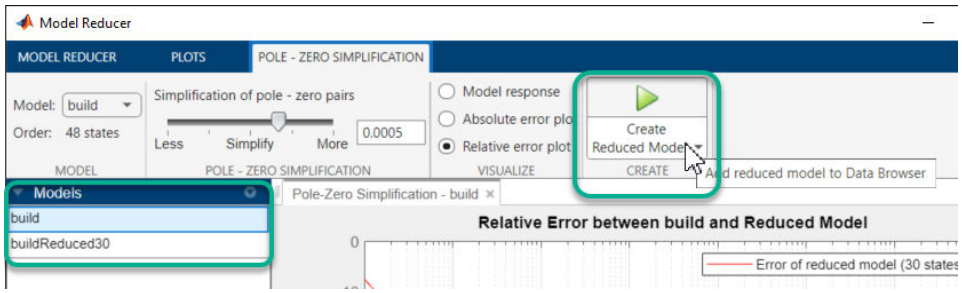
The blue x and o marks on the pole-zero map show the near-canceling pole-zero pairs in the original model that are eliminated from the simplified model. Poles and zeros remaining in the simplified model are marked with red x and o.

4 Try different simplification tolerances while observing the frequency response of the original and simplified model. Remove as many poles and zeros as you can while preserving the system behavior in the frequency region that is important for your application. Optionally, examine absolute or relative error between the original and simplified model. Select the error-plot type using the buttons on the **Pole-Zero Simplification** tab.



For more information about using the analysis plots, see “Visualize Reduced-Order Models in the Model Reducer App” on page 6-58.

5 When you have a simplified model that you want to store and analyze further, click  . The new model appears in the data browser with a name that reflects the reduced model order.



After creating a reduced model in the data browser, you can continue changing the simplification parameters and create reduced models with different orders for analysis and comparison.

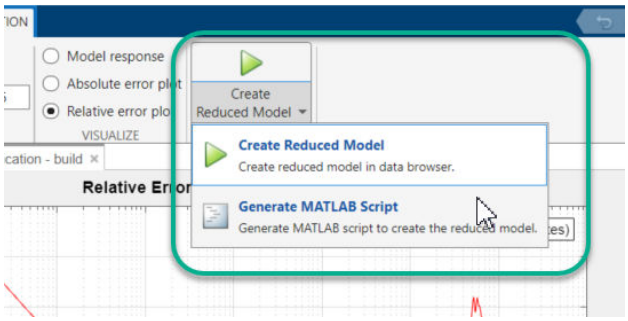
You can now perform further analysis with the reduced model. For example:

- Examine other responses of the reduced system, such as the step response or Nichols plot. To do so, use the tools on the **Plots** tab. See “Visualize Reduced-Order Models in the Model Reducer App” on page 6-58 for more information.

- Export reduced models to the MATLAB workspace for further analysis or control design. On the **Model Reducer** tab, click **Export**.

Generate MATLAB Code for Pole-Zero Simplification

To create a MATLAB script you can use for further model-reduction tasks at the command line, click **Create Reduced Model**, and select **Generate MATLAB Script**.

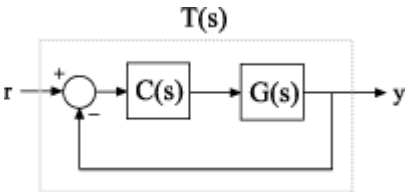


Model Reducer creates a script that uses the `minreal` command to perform model reduction with the parameters you have set on the **Pole-Zero Simplification** tab. The script opens in the MATLAB editor.

Pole-Zero Cancellation at the Command Line

To reduce the order of a model by pole-zero cancellation at the command line, use `minreal`.

Create a model of the following system, where `C` is a PI controller, and `G` has a zero at 3×10^{-8} rad/s. Such a low-frequency zero can arise from derivative action somewhere in the plant dynamics. For example, the plant may include a component that computes speed from position measurements.




```
G = zpk(3e-8,[-1,-3],1);
C = pid(1,0.3);
T = feedback(G*C,1)
```

```
T =

    (s+0.3) (s-3e-08)
  ----------------------
  s (s+4.218) (s+0.7824)

Continuous-time zero/pole/gain model.
```

In the closed-loop model `T`, the integrator ($1/s$) from `C` very nearly cancels the low-frequency zero of `G`.

Force a cancellation of the integrator with the zero near the origin.

```
Tred = minreal(T,1e-7)
```

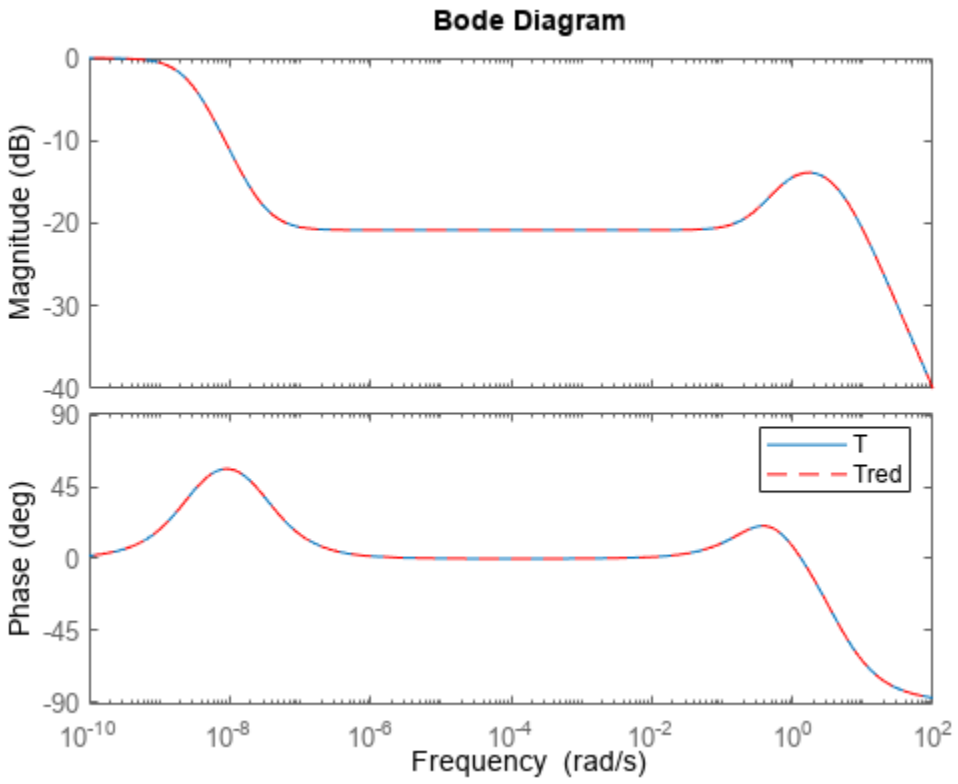
```
Tred =

    (s+0.3) (s-3e-08)
  ----------------------
  s (s+4.218) (s+0.7824)

Continuous-time zero/pole/gain model.
```

By default, `minreal` reduces transfer function order by canceling exact pole-zero pairs or near pole-zero pairs within `sqrt(eps)`. Specifying `1e-7` as the second input causes `minreal` to eliminate pole-zero pairs within 10^{-7} rad/s of each other.

The reduced model `Tred` includes all the dynamics of the original closed-loop model `T`, except for the near-canceling zero-pole pair.

Compare the frequency responses of the original and reduced systems.

```
bode(T,Tred,'r--')
legend('T','Tred')
```



Because the canceled pole and zero do not match exactly, some extreme low-frequency dynamics evident in the original model are missing from `Tred`. In many applications, you can neglect such

extreme low-frequency dynamics. When you increase the matching tolerance of `minreal`, make sure that you do not eliminate dynamic features that are relevant to your application.

See Also

Apps
Model Reducer

Functions
`minreal` | `sminreal`

Live Editor Tasks
Reduce Model Order

Related Examples

- “Balanced Truncation Model Reduction” on page 6-13
- “Mode-Selection Model Reduction” on page 6-50
- “Model Reduction Basics” on page 6-2

Mode-Selection Model Reduction

Model selection eliminates poles that fall outside a specific frequency range of interest. This method is useful when you want to focus your analysis on a particular subset of system dynamics. For instance, if you are working with a control system with bandwidth limited by actuator dynamics, you might discard higher-frequency dynamics in the plant. Eliminating dynamics outside the frequency range of interest reduces the numerical complexity of calculations with the model. There are two ways to compute a reduced-order model by mode selection:

- At the command line, using the `freqsep` command.
- In the **Model Reducer**, using the **Mode Selection** method.
- In the **Reduce Model Order** task in the Live Editor, using the `Mode Selection` method.

For more general information about model reduction, see “Model Reduction Basics” on page 6-2.

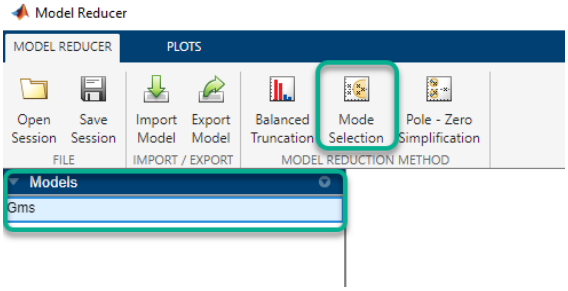
Mode Selection in the Model Reducer App

Model Reducer provides an interactive tool for performing model reduction and examining and comparing the responses of the original and reduced-order models. To approximate a model by mode selection in **Model Reducer**:

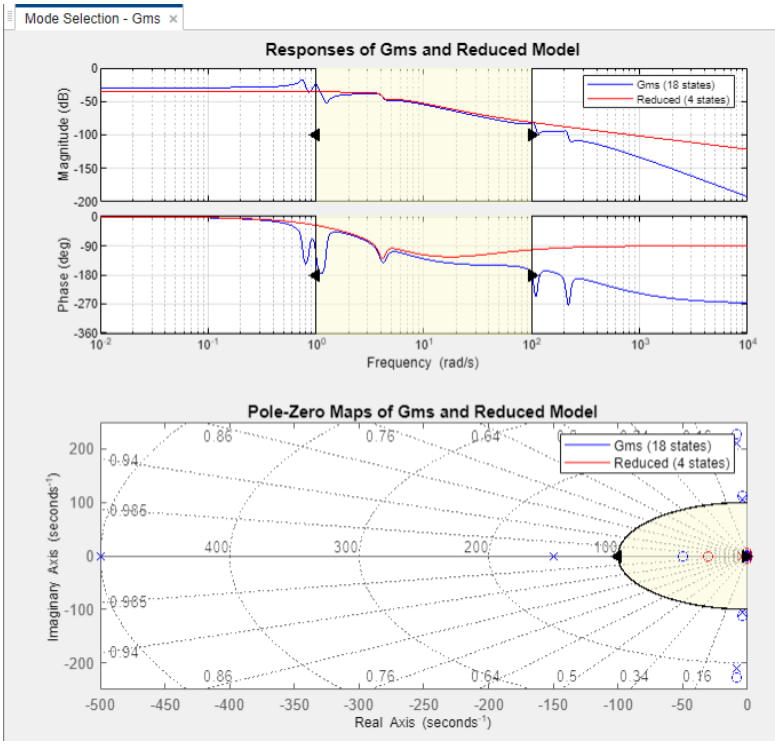
1. Open the app and import an LTI model to reduce. For instance, suppose that there is a model named `Gms` in the MATLAB workspace. The following command opens **Model Reducer** and imports the model.

   ```
   modelReducer(Gms)
   ```

2. In the data browser, select the model to reduce. Click  **Mode Selection**.



   In the **Mode Selection** tab, **Model Reducer** displays a plot of the frequency response of the original model and a reduced version of the model. The app also displays a pole-zero map of both models.

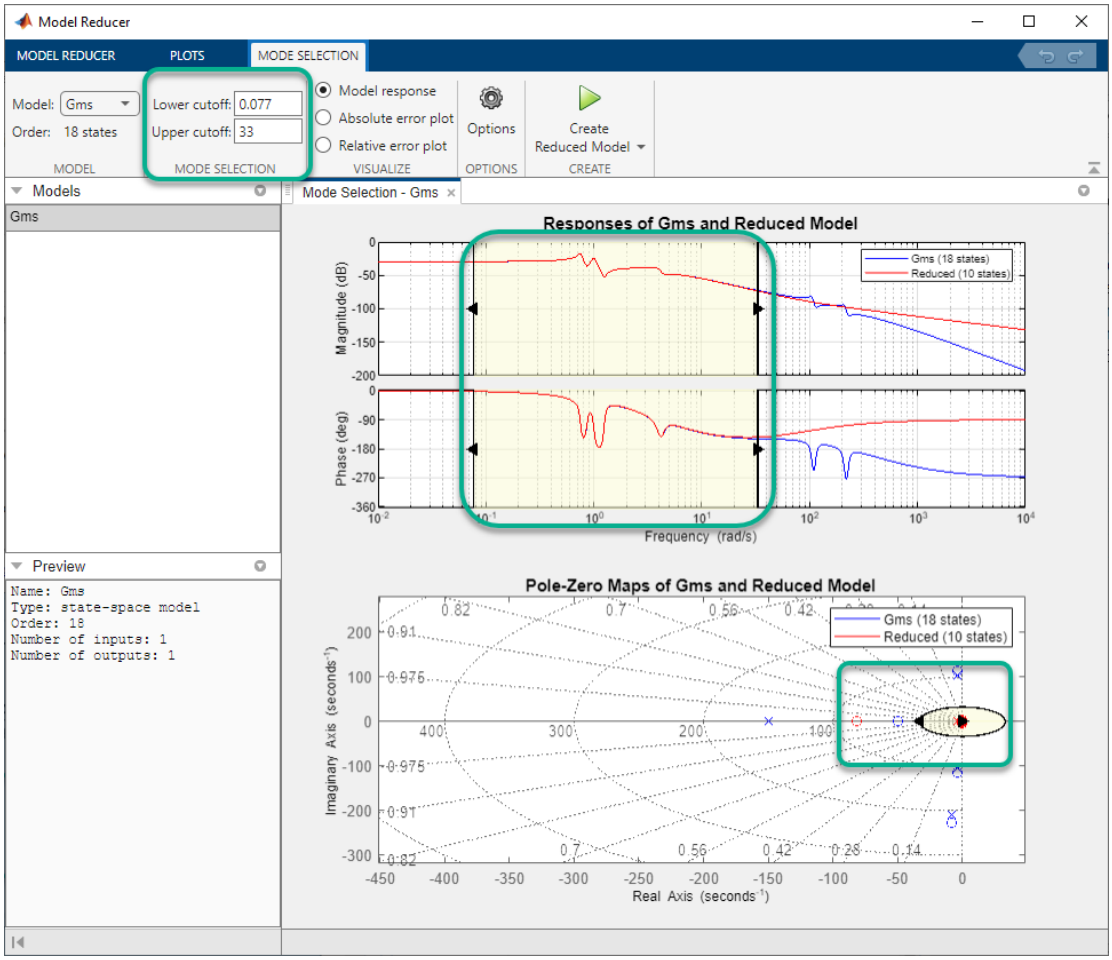


The pole-zero map marks pole locations with x and zero locations with o.

Note The frequency response is a Bode plot for SISO models, and a singular-value plot for MIMO models.

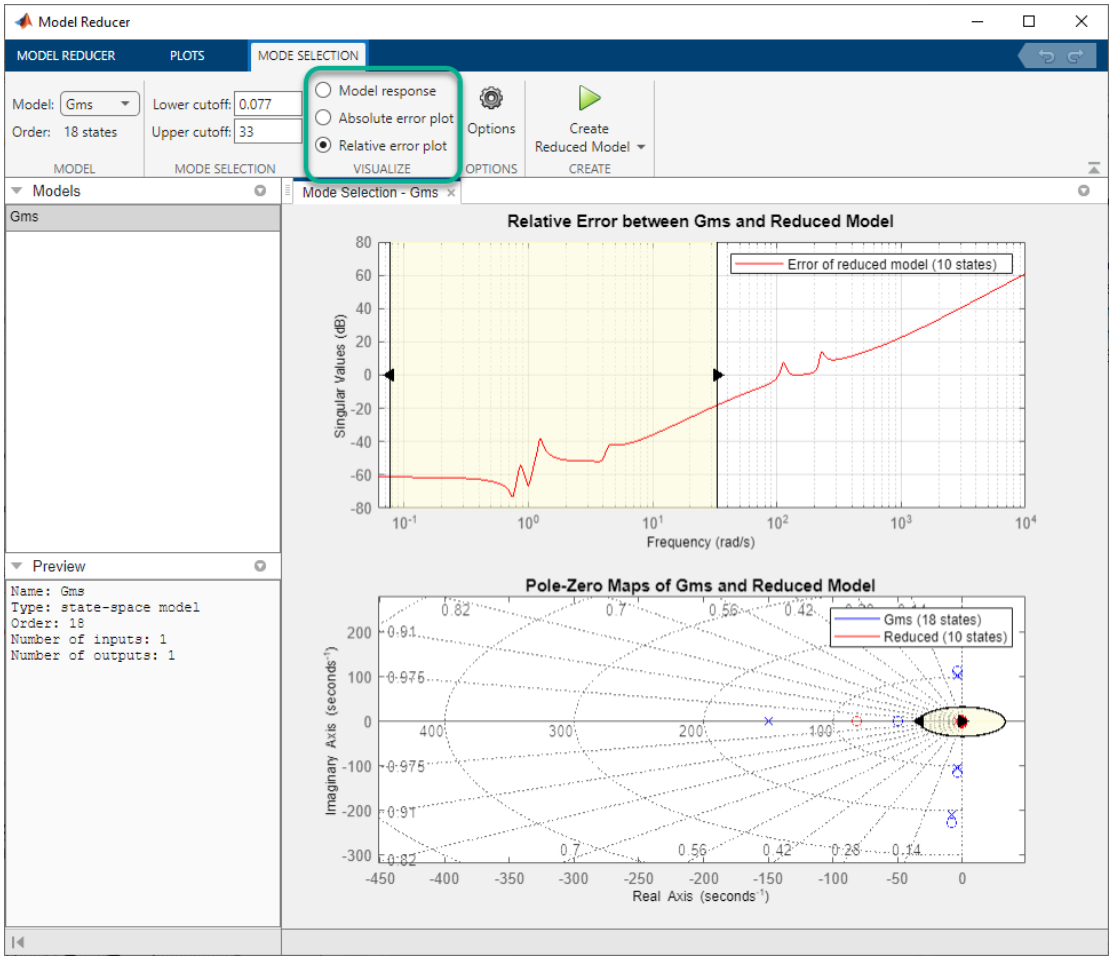
3 **Model Reducer** eliminates poles that lie outside the shaded region. Change the shaded region to capture only the dynamics you want to preserve in the reduced model. There are two ways to do so.

- On either the response plot or the pole-zero map, drag the boundaries of the shaded region or the shaded region itself.
- On the **Mode Selection** tab, enter lower and upper cutoff frequencies.



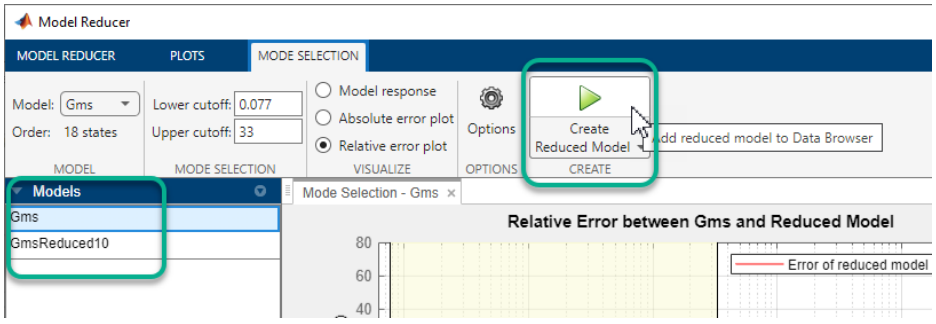
When you change the shaded regions or cutoff frequencies, **Model Reducer** automatically computes a new reduced-order model. All poles retained in the reduced model fall within the shaded region on the pole-zero map. The reduced model might contain zeros that fall outside the shaded region.

4 Optionally, examine absolute or relative error between the original and simplified model. Select the error-plot type using the buttons on the **Mode Selection** tab.



For more information about using the analysis plots, see “Visualize Reduced-Order Models in the Model Reducer App” on page 6-58.

5 When you have one or more reduced models that you want to store and analyze further, click  . The new model appears in the data browser.



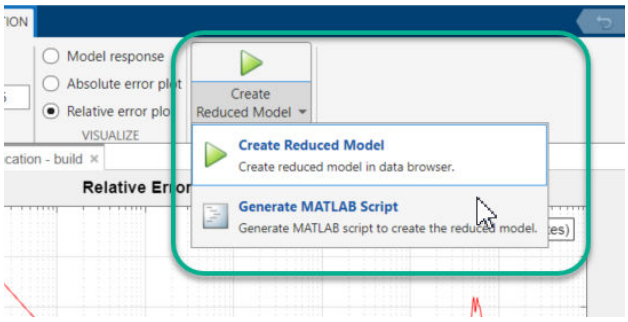
After creating a reduced model in the data browser, you can continue adjusting the mode-selection region to create reduced models with different orders for analysis and comparison.

You can now perform further analysis with the reduced model. For example:

- Examine other responses of the reduced system, such as the step response or Nichols plot. To do so, use the tools on the **Plots** tab. See “Visualize Reduced-Order Models in the Model Reducer App” on page 6-58 for more information.
- Export reduced models to the MATLAB workspace for further analysis or control design. On the **Model Reducer** tab, click **Export**.

Generate MATLAB Code for Mode Selection

To create a MATLAB script you can use for further model-reduction tasks at the command line, click **Create Reduced Model**, and select **Generate MATLAB Script**.



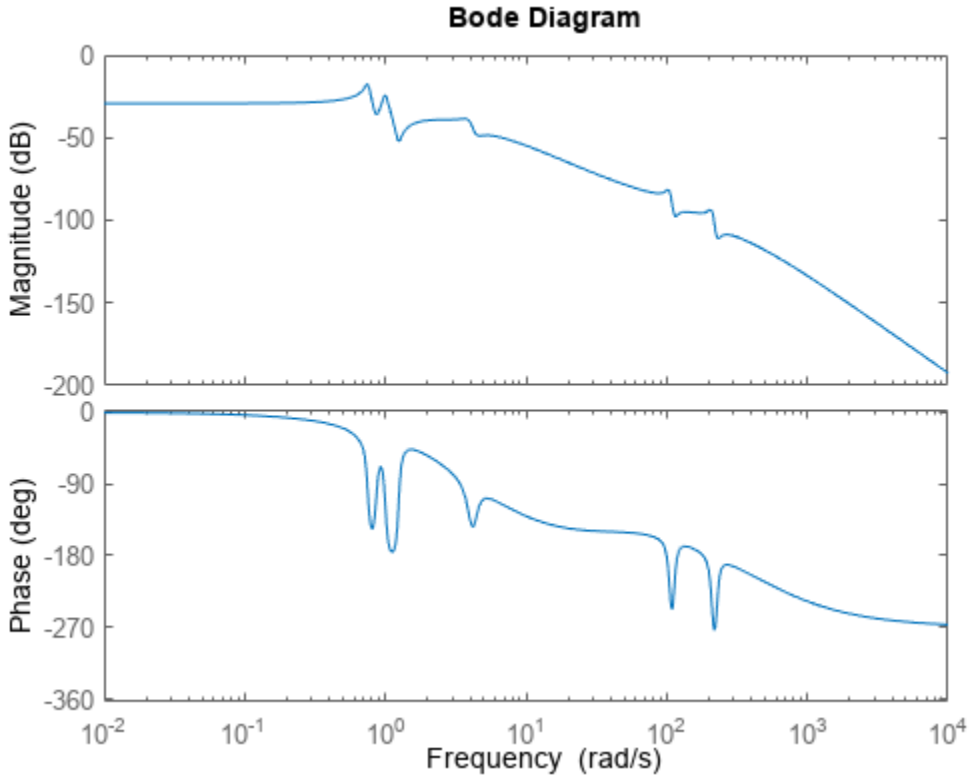
Model Reducer creates a script that uses the `freqsep` command to perform model reduction with the parameters you have set on the **Mode Selection** tab. The script opens in the MATLAB editor.

Mode Selection at the Command Line

To reduce the order of a model by mode selection at the command line, use `freqsep`. This command separates a dynamic system model into slow and fast components around a specified frequency.

For this example, load the model `Gms` and examine its frequency response.

```
load modeselect Gms
bodeplot(Gms)
```

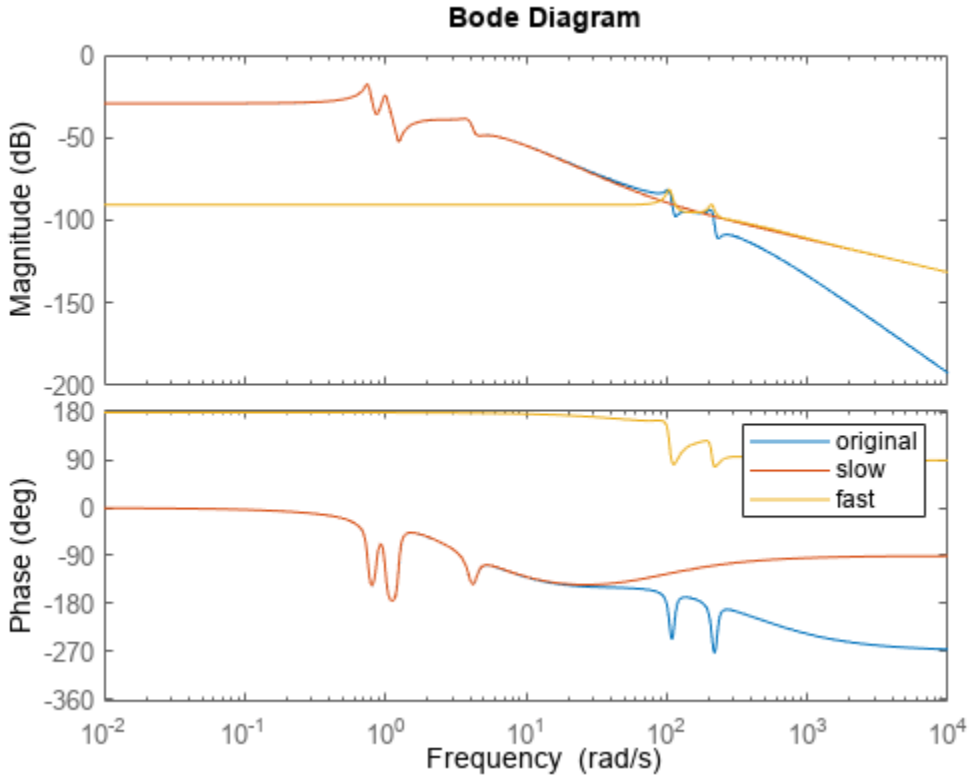


`Gms` has two sets of resonances, one at relatively low frequency and the other at relatively high frequency. Suppose that you want to tune a controller for `Gms`, but the actuator in your system is limited to a bandwidth of about 3 rad/s, in between the two groups of resonances. To simplify calculation and tuning using `Gms`, you can use mode selection to eliminate the high-frequency dynamics.

```
[Gms_s,Gms_f] = freqsep(Gms,30);
```

`freqsep` decomposes `Gms` into slow and fast components such that `Gms = Gms_s + Gms_f`. All modes (poles) with natural frequency less than 30 are in `Gms_s`, and the higher-frequency poles are in `Gms_f`.

```
bodeplot(Gms,Gms_s,Gms_f)
legend('original','slow','fast')
```



The slow component, `Gms_s`, contains only the lower-frequency resonances and matches the DC gain of the original model. Examine the orders of both models.

```
order(Gms)
```

```
ans = 18
```

```
order(Gms_s)
```

```
ans = 10
```

When the high-frequency dynamics are unimportant for your application, you can use the 10th-order `Gms_s` instead of the original 18th-order model. If neglecting low-frequency dynamics is appropriate for your application, you can use `Gms_f`. To select modes that fall between a low-frequency and a high-frequency cutoff, use additional calls to `freqsep`.

See Also

Apps
Model Reducer

Functions
`freqsep`

Live Editor Tasks
Reduce Model Order

Related Examples

- “Balanced Truncation Model Reduction” on page 6-13
- “Pole-Zero Simplification” on page 6-43
- “Model Reduction Basics” on page 6-2

Visualize Reduced-Order Models in the Model Reducer App

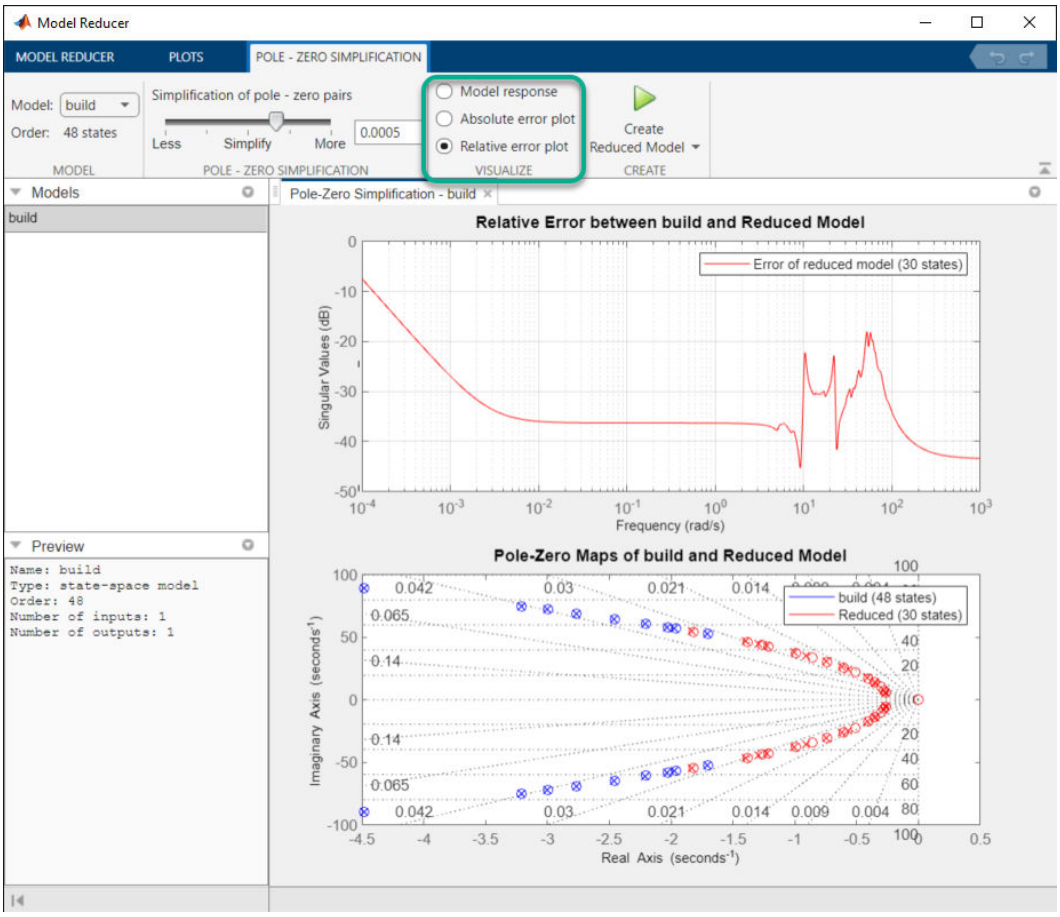
The plotting tools in the **Model Reducer** app let you examine and compare time-domain and frequency-domain responses of the original model and the reduced models you create in the app. Use these tools to help verify that the reduced-order model you choose to work with preserves the system behaviors that are important for your application.

For more general information about model reduction, see “Model Reduction Basics” on page 6-2.

Error Plots

By default, for any model reduction method, **Model Reducer** shows a frequency-response plot of both the original and reduced models. This plot is a Bode plot for SISO models, and a singular-value plot for MIMO models.

To more closely examine the differences between an original model and a reduced model, you can use absolute error or relative error plots. On any model reduction tab, click **Absolute error plot** or **Relative error plot** to view these plots.



- **Absolute error plot** — Shows the singular values of `G-Gr`, where `G` is the original model and `Gr` is the current reduced model.

- **Relative error plot** — Shows the singular values of `(G-Gr)/G`. This plot is useful when the model has very high or very low gain in the region that is important to your application. In such regions, absolute error can be misleading.

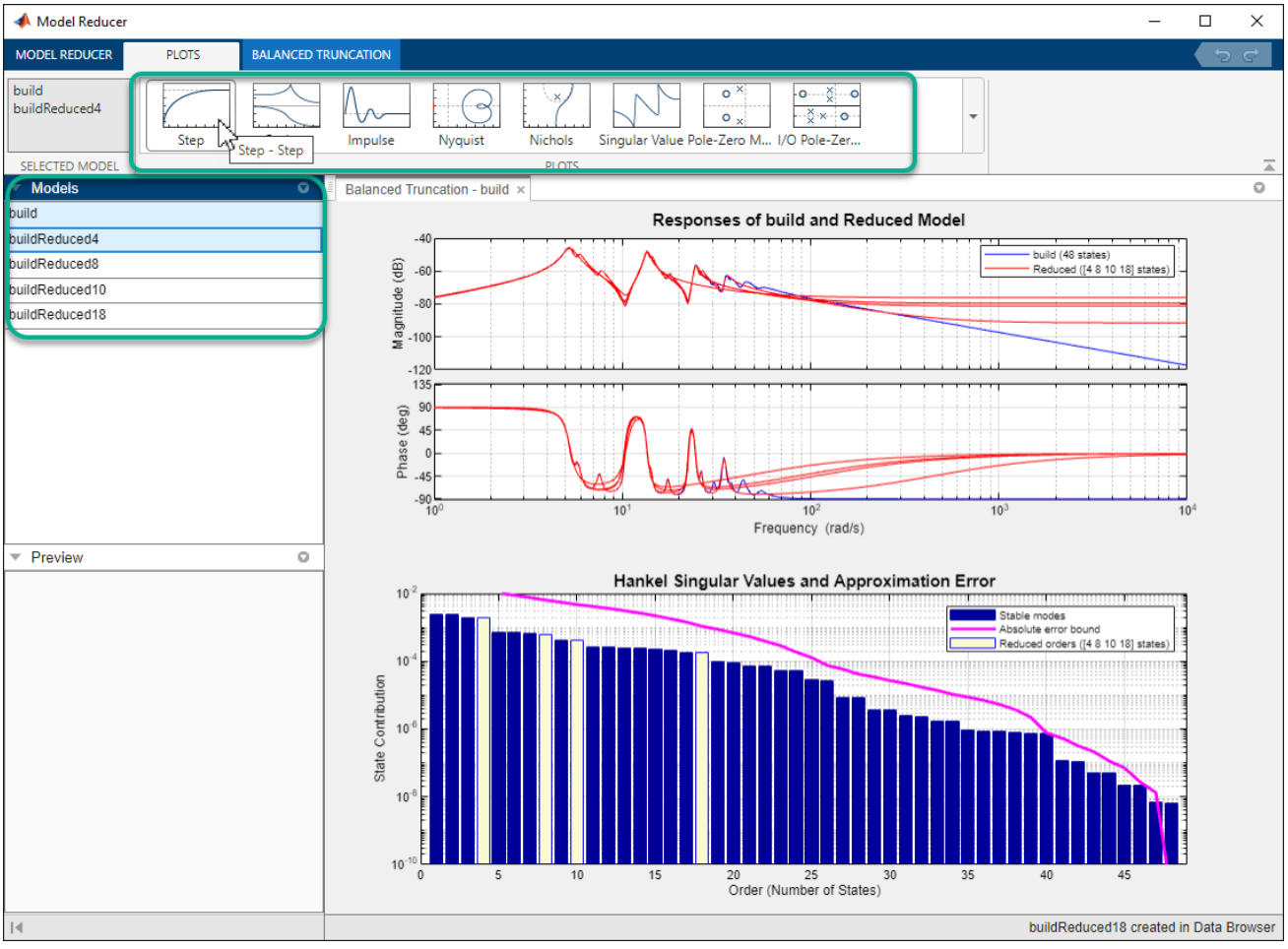
For SISO models, the singular-value plot is the magnitude of the frequency response.

Response Plots

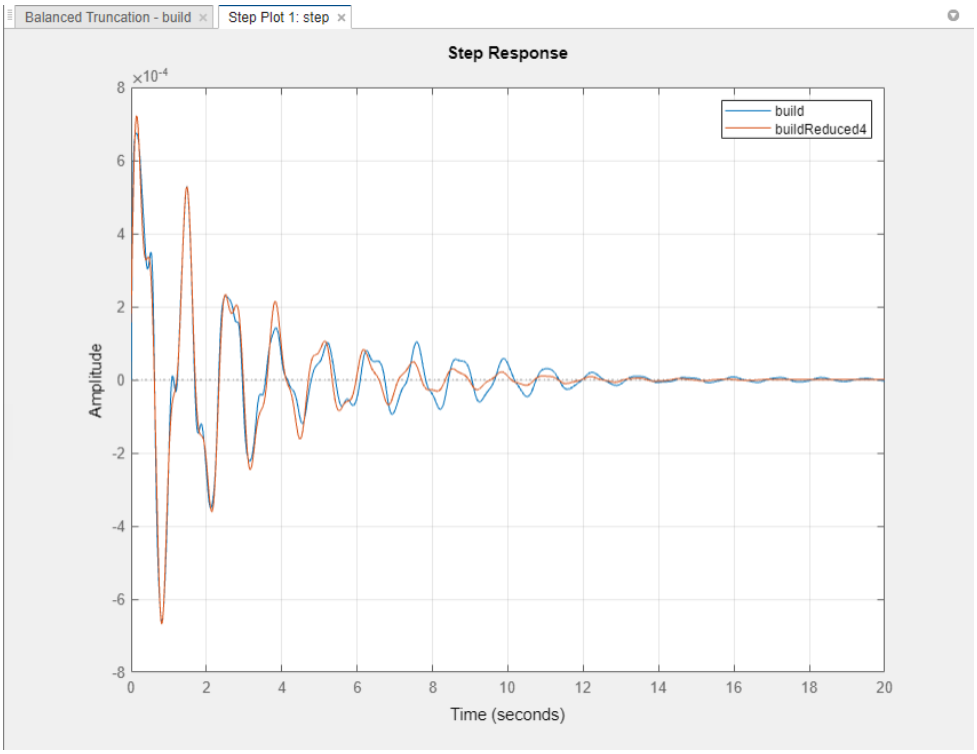
After you click  to add one or more reduced models to the data browser, compare additional responses of the original and reduced models using the **Plots** tab.

Create New Response Plot

In the data browser, select one or more models to plot. (Ctrl-click to select multiple models.) Then, on the **Plots** tab, click the type of plot you want to create.

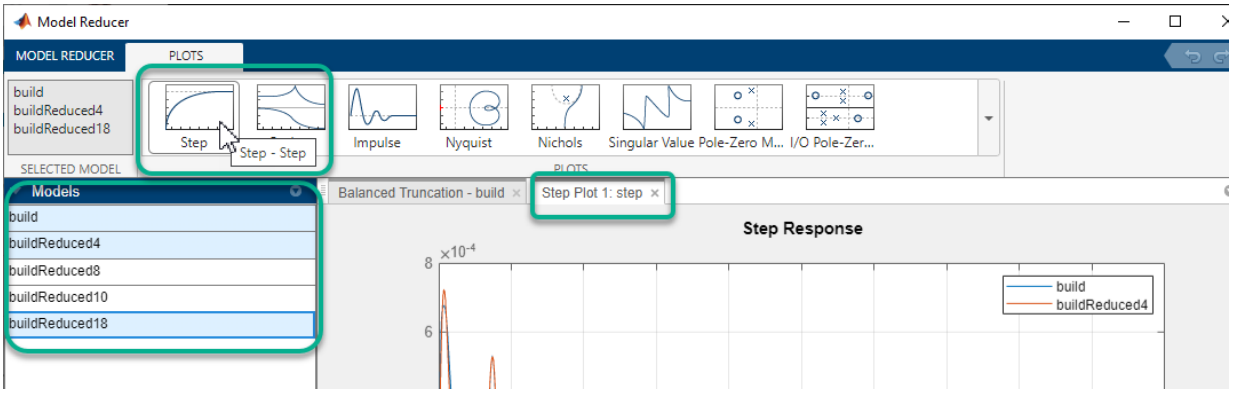


Model Reducer creates the plot.



Add Model to Existing Plot

In the data browser, select the model to add. Then, on the **Plots** tab, click the icon corresponding to the plot you want to update. Plots you have created appear on the left side of the plot gallery.



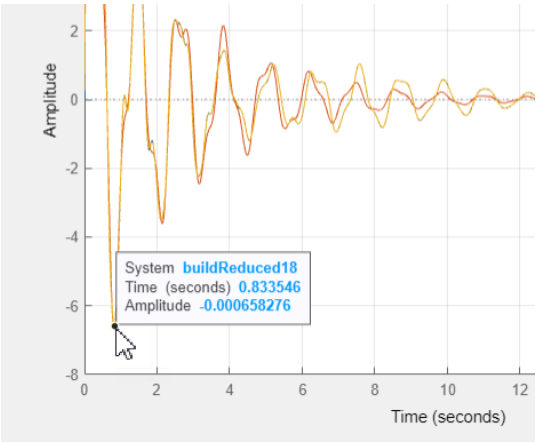
Model Reducer updates the plot with the new model.

Tip To expand the gallery view, click ▼.

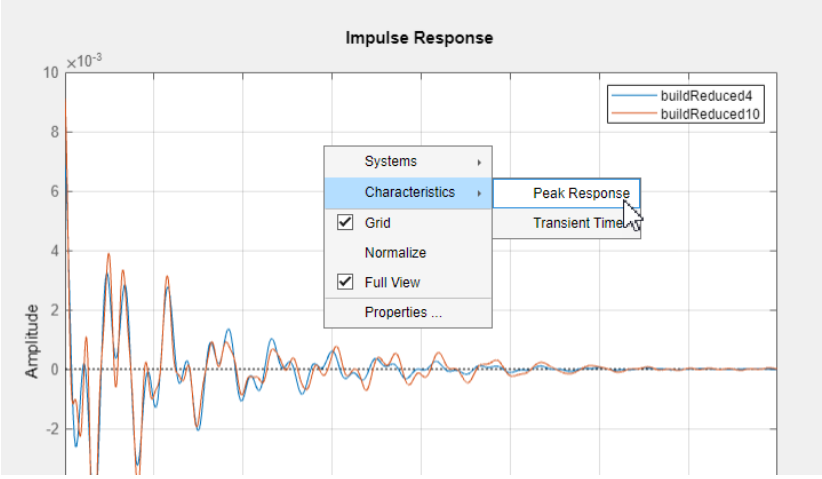
Plot Characteristics

On any plot in **Model Reducer**:

- To see response information and data values, click a line on the plot.

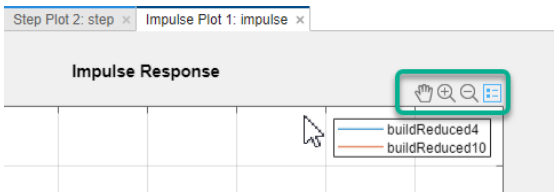


- To view system characteristics, right-click anywhere on the plot, as described in “Frequency-Domain Characteristics on Response Plots” on page 8-8.

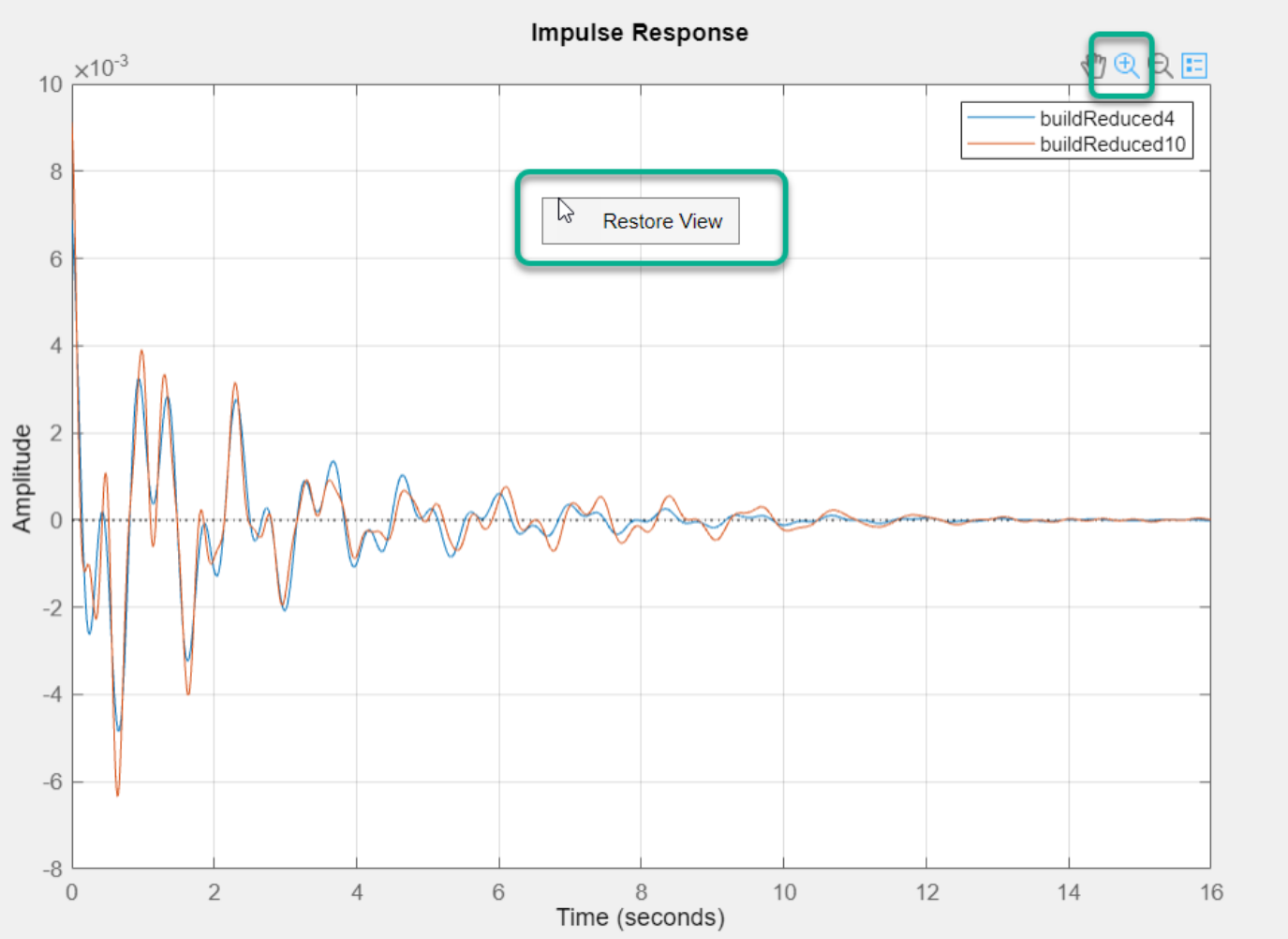


Plot Tools

Mouse over any plot to access plot tools at the upper right corner of the plot.

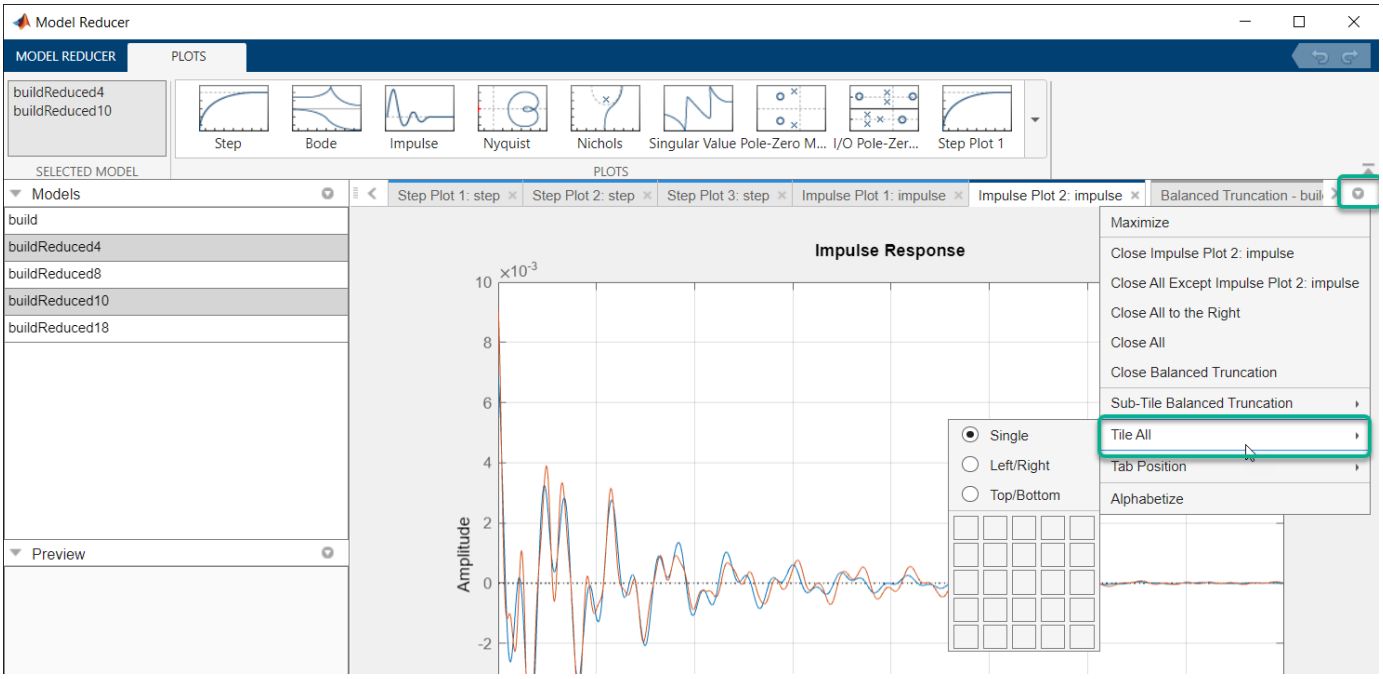


- and — Zoom in and zoom out. Click to activate, and drag the cursor over the region to zoom. The zoom icon turns dark when zoom is active. Right-click while zoom is active select **Restore View**. Click the icon again to deactivate.



- — 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 active. To toggle the legend off and on, 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 on the arrow icon at the end of the plot tabs and select one of the options from **Tile All**.



See Also

Model Reducer

Related Examples

- “Balanced Truncation Model Reduction” on page 6-13
- “Mode-Selection Model Reduction” on page 6-50
- “Pole-Zero Simplification” on page 6-43

Linear Analysis

Time Domain Analysis

- “Plotting System Responses” on page 7-2
- “Time-Domain Responses” on page 7-19
- “Time-Domain Response Data and Plots” on page 7-20
- “Time-Domain Characteristics on Response Plots” on page 7-22
- “Numeric Values of Time-Domain System Characteristics” on page 7-25
- “Time-Domain Responses of Discrete-Time Model” on page 7-26
- “Time-Domain Responses of MIMO Model” on page 7-28
- “Time-Domain Responses of Multiple Models” on page 7-30
- “Joint Time-Domain and Frequency-Domain Analysis” on page 7-32
- “Response from Initial Conditions” on page 7-36
- “Import LTI Model Objects into Simulink” on page 7-39
- “Analysis of Systems with Time Delays” on page 7-43

Plotting System Responses

This example shows how to plot the time and frequency responses of SISO and MIMO linear systems.

Time Responses

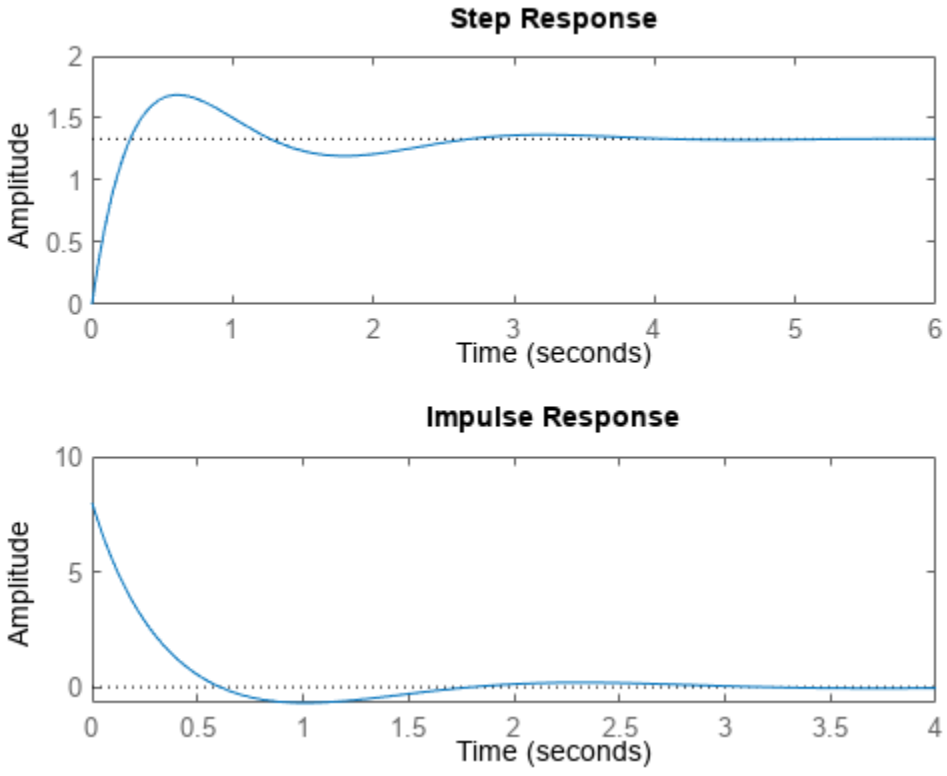
Create a linear system. For this example, create a third-order transfer function.

```
sys = tf([8 18 32],[1 6 14 24])
```

```
sys =

     8 s^2 + 18 s + 32
  -----------------------
  s^3 + 6 s^2 + 14 s + 24

Continuous-time transfer function.
```

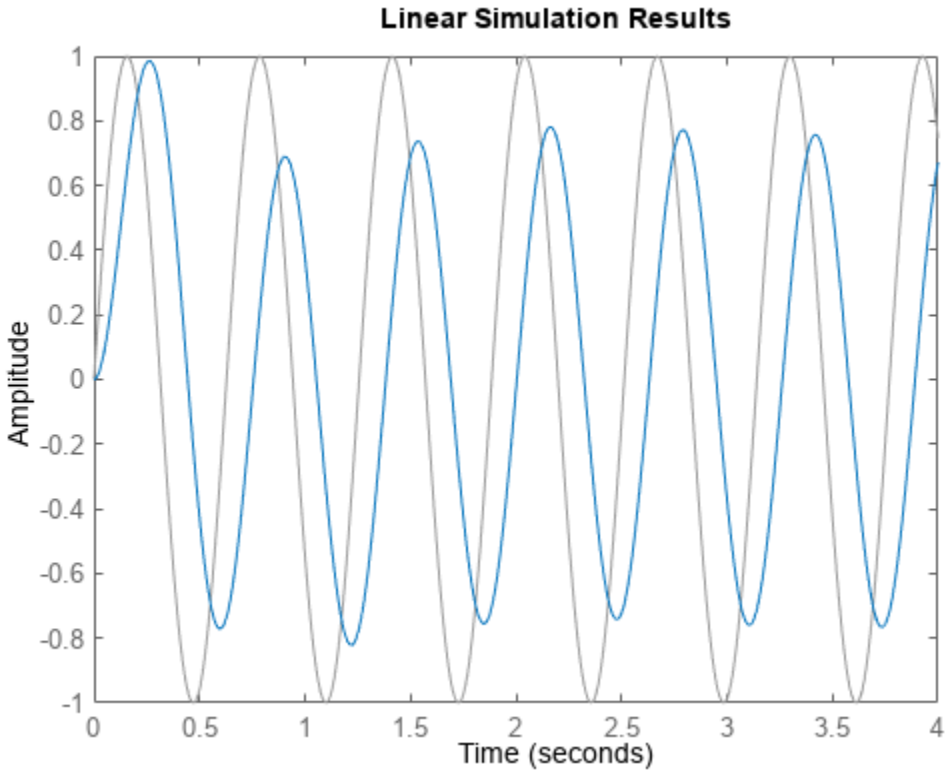
You can plot the step and impulse responses of this system using the `step` and `impulse` commands.

```
subplot(2,1,1)
step(sys)
subplot(2,1,2)
impulse(sys)
```



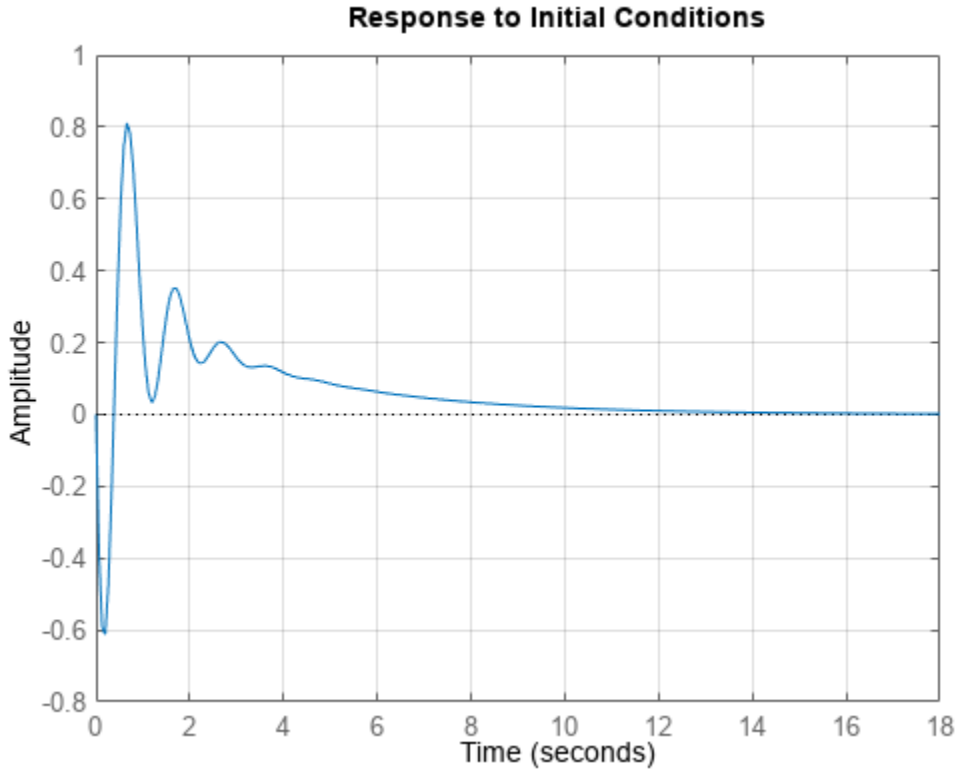
You can also simulate the response to an arbitrary signal, such as a sine wave, using the `lsim` command. The input signal appears in gray and the system response in blue.

```
clf
t = 0:0.01:4;
u = sin(10*t);
lsim(sys,u,t)   % u,t define the input signal
```



You can use the plotting commands with continuous or discrete `tf`, `ss`, or `zpk` models. For state-space models, you can also plot the unforced response from some given initial state. For example:

```
A = [-0.8 3.6 -2.1;-3 -1.2 4.8;3 -4.3 -1.1];
B = [0; -1.1; -0.2];
C = [1.2 0 0.6];
D = -0.6;
G = ss(A,B,C,D);
x0 = [-1;0;2];  % initial state
initial(G,x0)
grid
```



Frequency Responses

Frequency-domain analysis is key to understanding stability and performance properties of control systems. Bode plots, Nyquist plots, and Nichols charts are three standard ways to plot and analyze the frequency response of a linear system. You can create these plots using the `bode`, `nichols`, and `nyquist` commands.

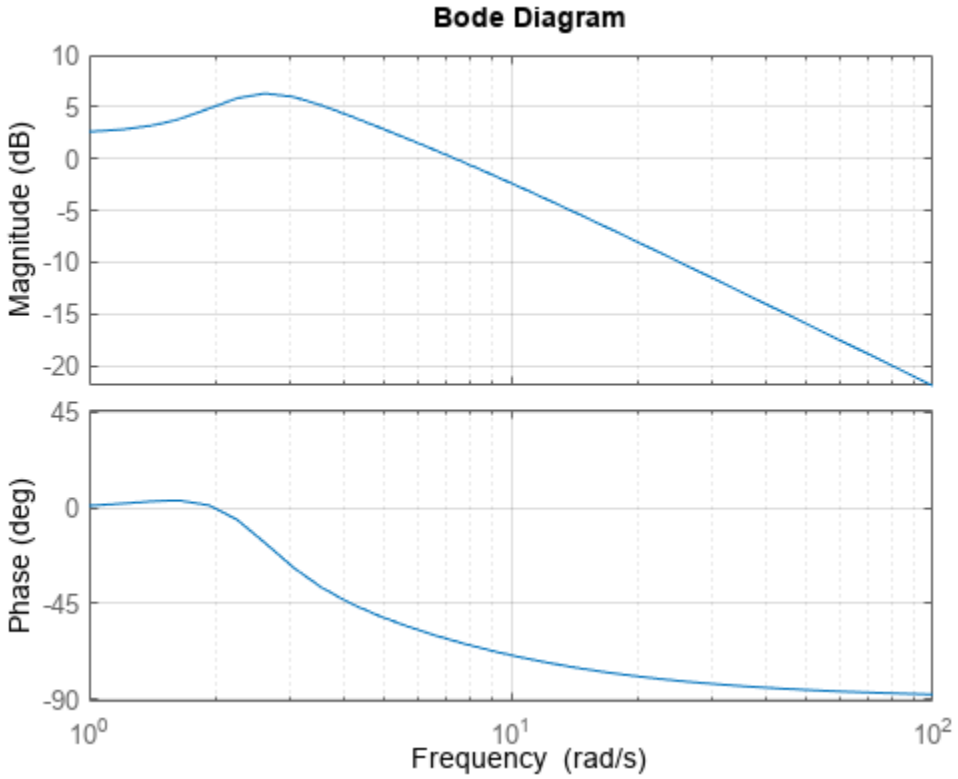
Create a linear system.

```
sys = tf([8 18 32],[1 6 14 24])
```

```
sys =

     8 s^2 + 18 s + 32
  -----------------------
  s^3 + 6 s^2 + 14 s + 24

Continuous-time transfer function.
```

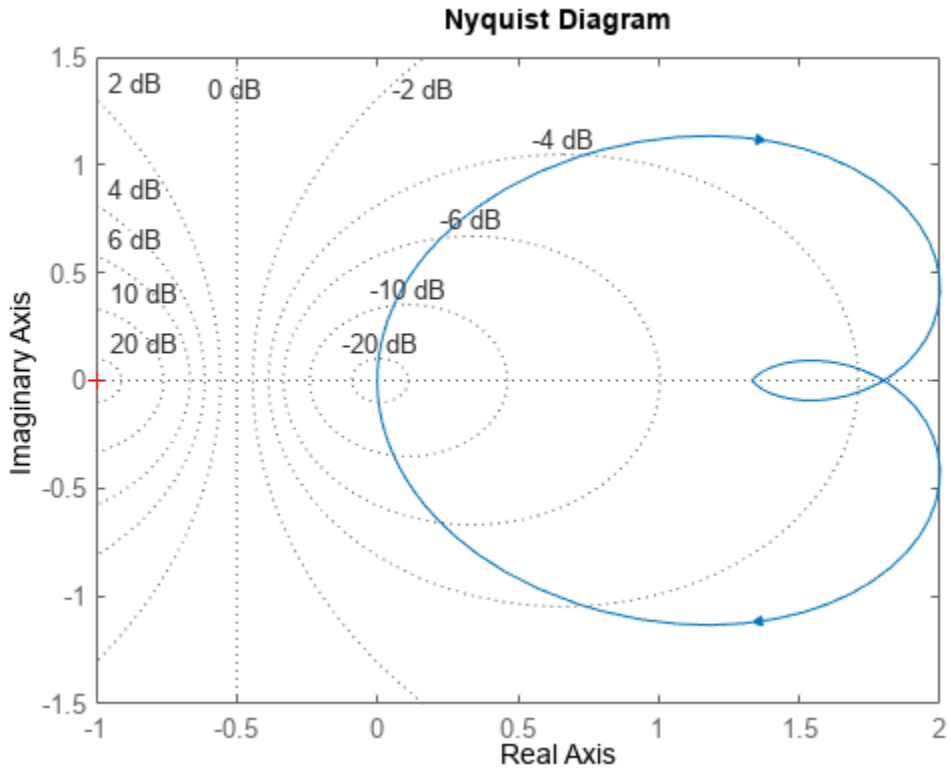
Create a Bode plot for this system.

```
bode(sys)
grid
```



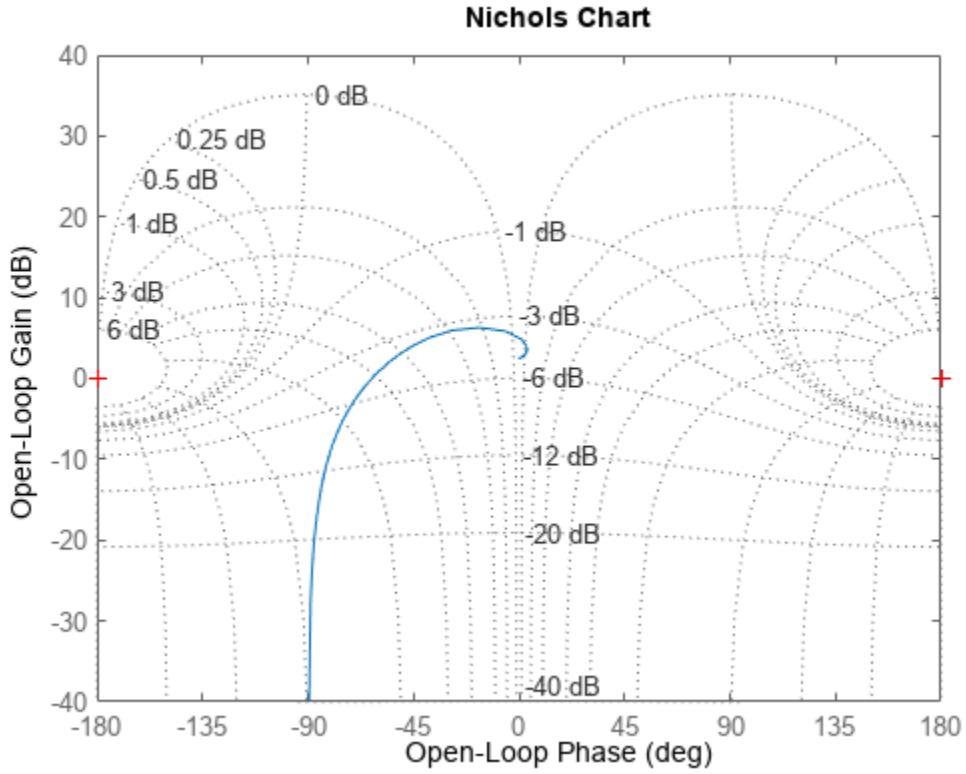
Create a Nyquist plot for this system.

```
nyquist(sys)
grid
```



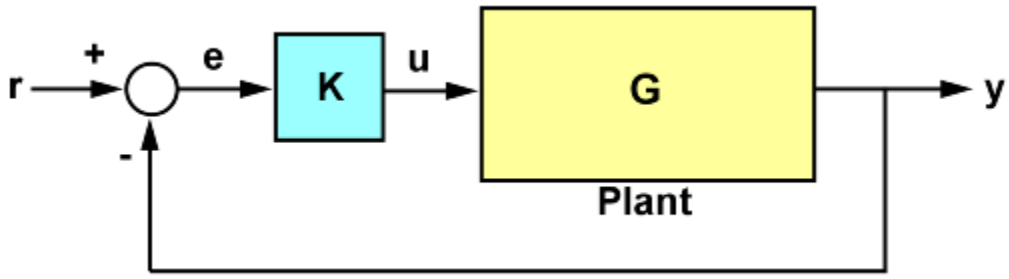
Create a Nichols chart for this system.

```
nichols(sys)
grid
```



Pole/Zero Maps and Root Locus

The poles and zeros of a system contain valuable information about its dynamics, stability, and limits of performance. For example, consider the feedback loop in the following SISO control loop.



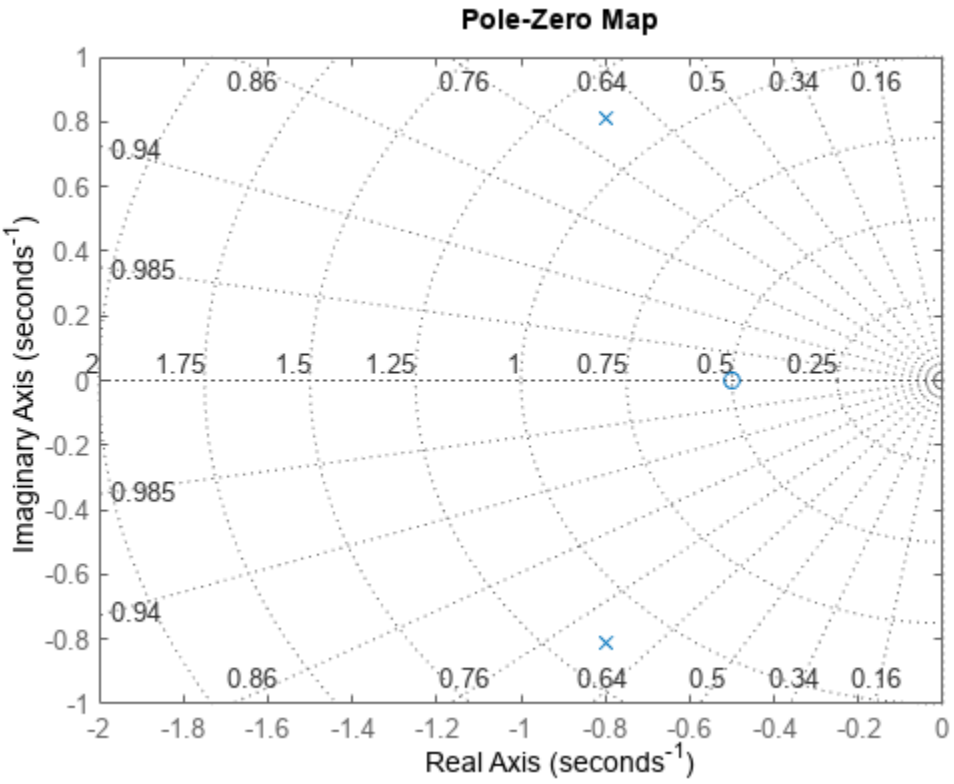
Here:

$$G = \frac{-(2s+1)}{s^2+3s+2}$$

For the gain value `k = 0.7`, you can plot the closed-loop poles and zeros using `pzmap`.

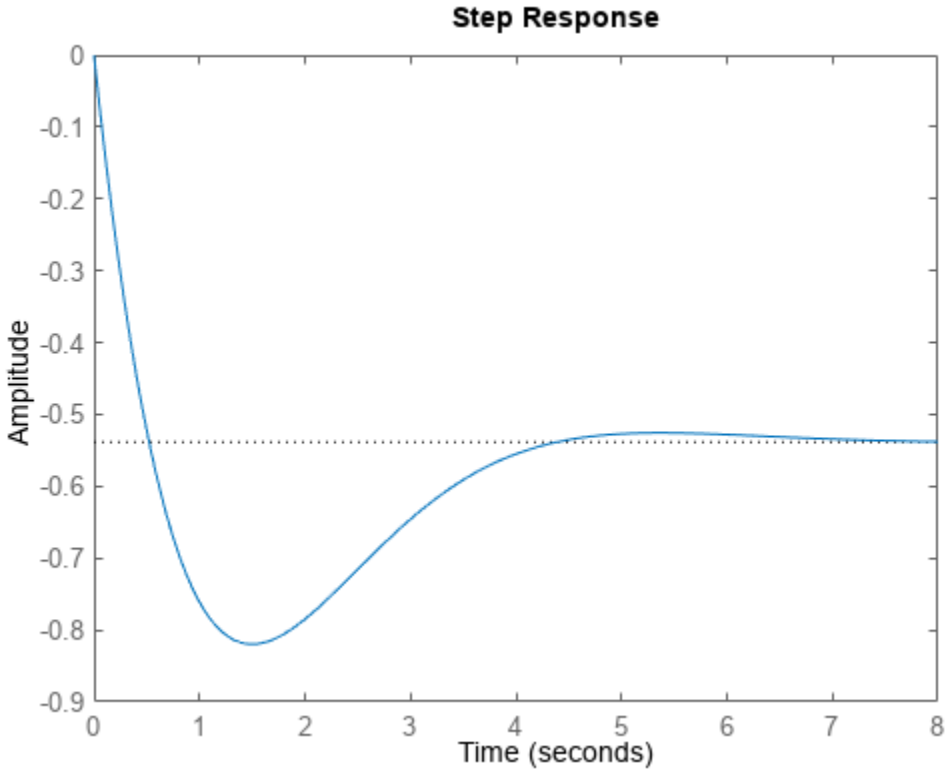
```
s = tf('s');
G = -(2*s+1)/(s^2+3*s+2);
k = 0.7;
T = feedback(G*k,1);
```

```
pzmap(T)
grid, axis([-2 0 -1 1])
```



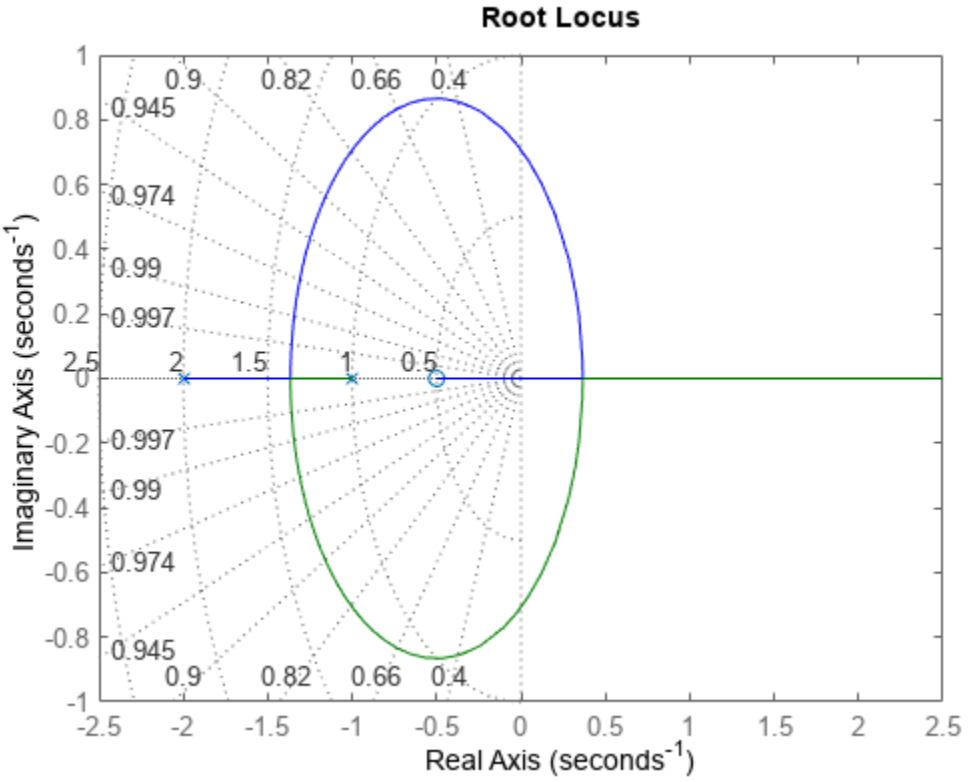
The closed-loop poles (marked by blue x's) lie in the left half-plane so the feedback loop is stable for this choice of gain k. You can read the damping ratio of the closed-loop poles from this chart (see labels on the radial lines). Here the damping ratio is about 0.7, suggesting a well-damped closed-loop response as confirmed by:

```
clf
step(T)
```

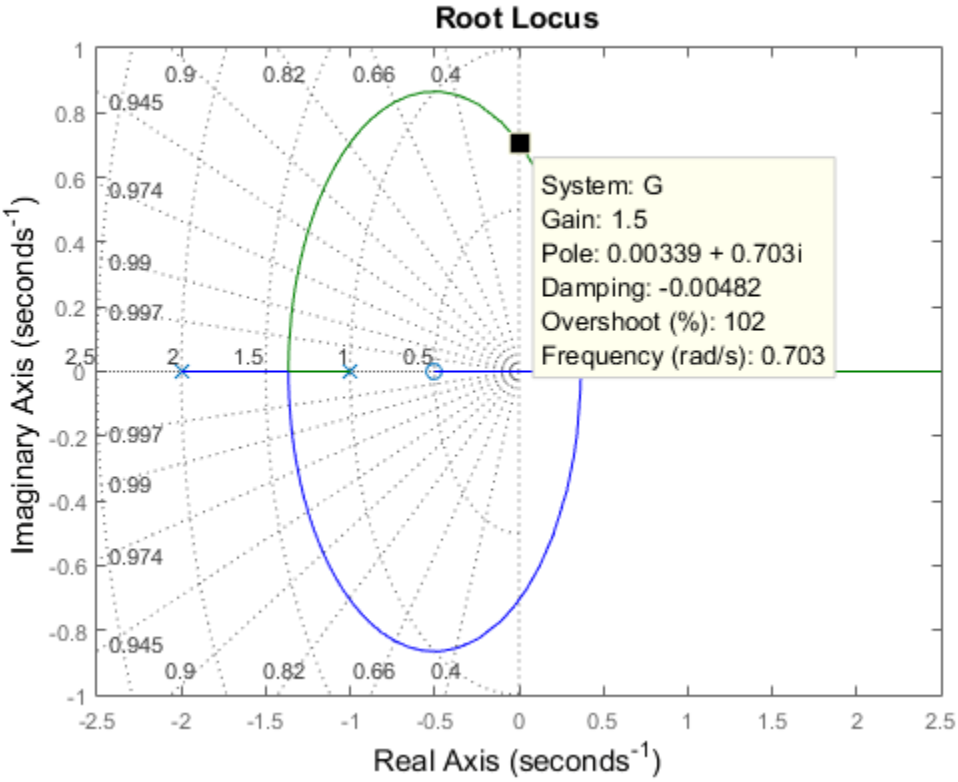


To further understand how the loop gain k affects closed-loop stability, you can plot the locus of the closed-loop poles as a function of k.

```
rlocus(G)
grid
```



Clicking where the locus intersects the y axis reveals that the closed-loop poles become unstable for `k = 1.51`. So the loop gain should remain smaller than 1.5 for closed-loop stability.

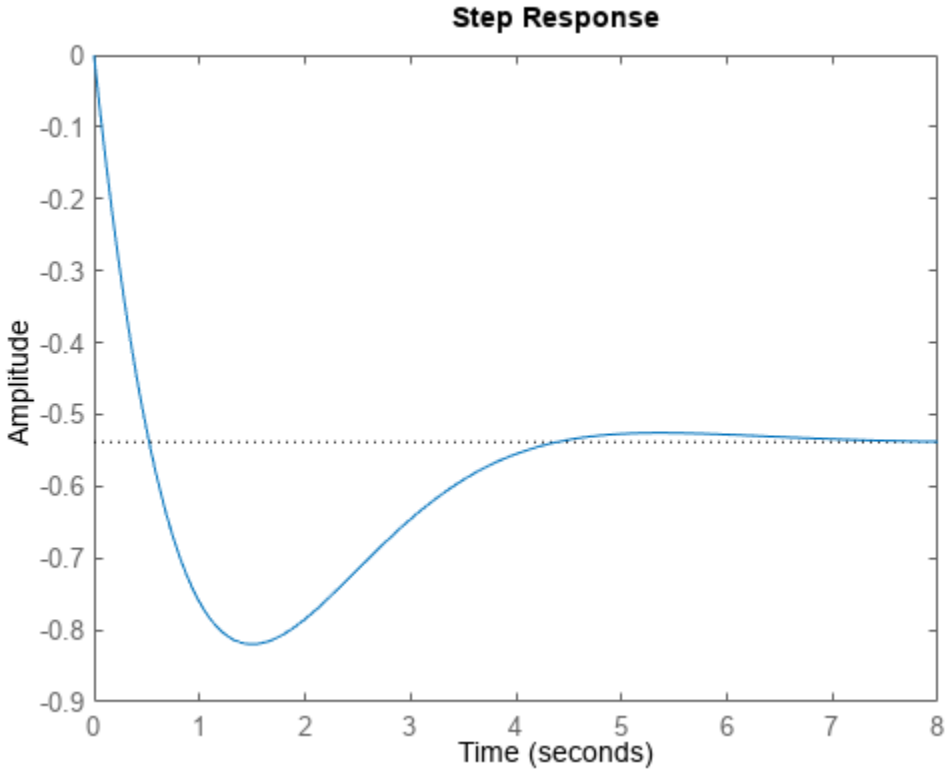


Response Characteristics

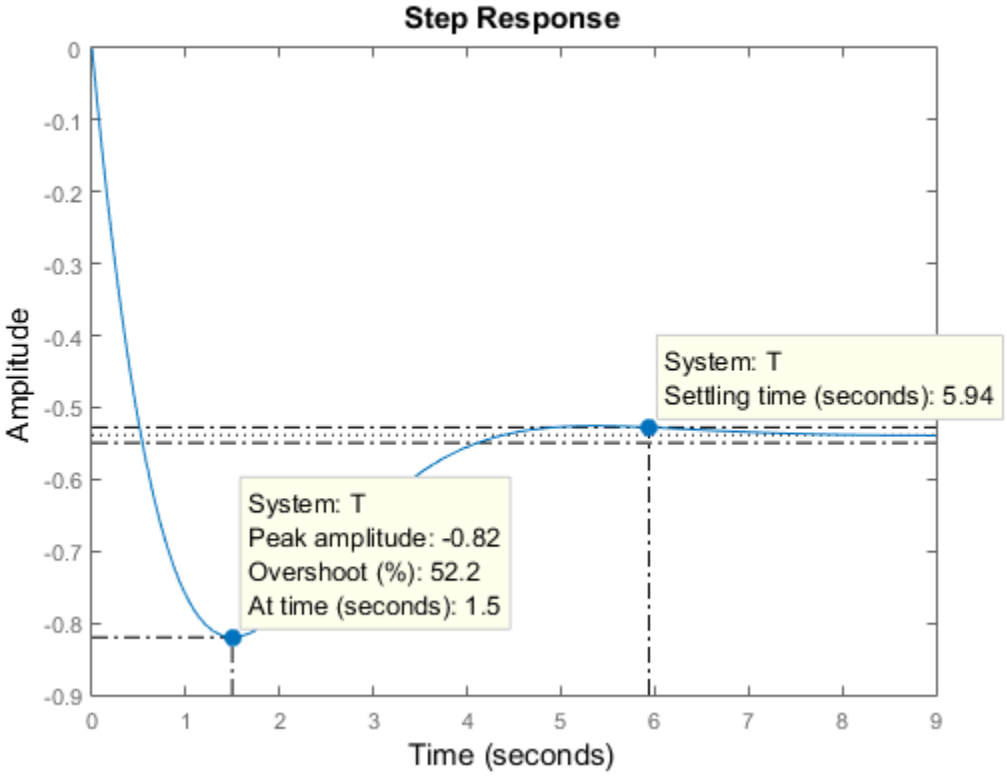
Right-clicking on response plots gives access to a variety of options and annotations. In particular, the **Characteristics** menu lets you display standard metrics such as rise time and settling time for step responses, or peak gain and stability margins for frequency response plots.

Using the example from the previous section, plot the closed-loop step response:

```
step(T)
```



Now, right-click on the plot to display the Peak Response and Settling Time Characteristics, and click on the blue dots to read the corresponding overshoot and settling time values:



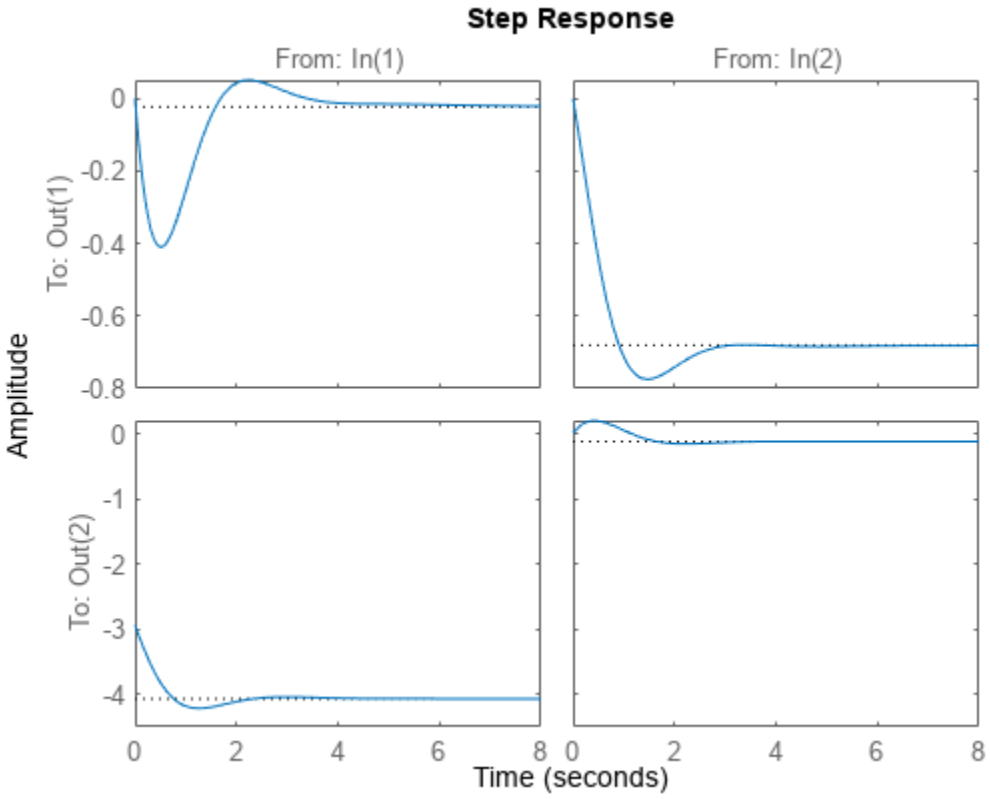
Analyzing MIMO Systems

All commands mentioned so far fully support multi-input multi-output (MIMO) systems. In the MIMO case, these commands produce arrays of plots. For example, consider the following two-input, two-output system.

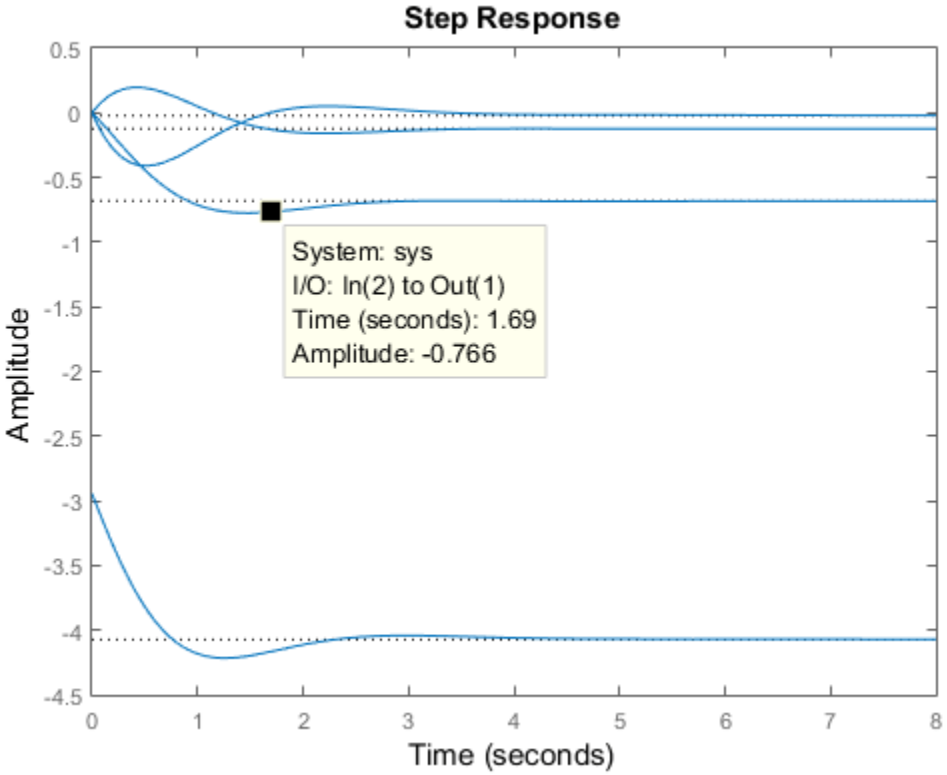
```
sys = rss(3,2,2);
sys.A = [-0.5 -0.3 -0.2 ; 0 -1.3  -1.7; 0.4 1.7 -1.3];
```

The step response is a 2-by-2 array of plots where each column shows the step response of a particular input channel.

```
step(sys)
```



You can group all four responses on a single plot by right-clicking on the plot and selecting the **I/O Grouping -> All** submenu.

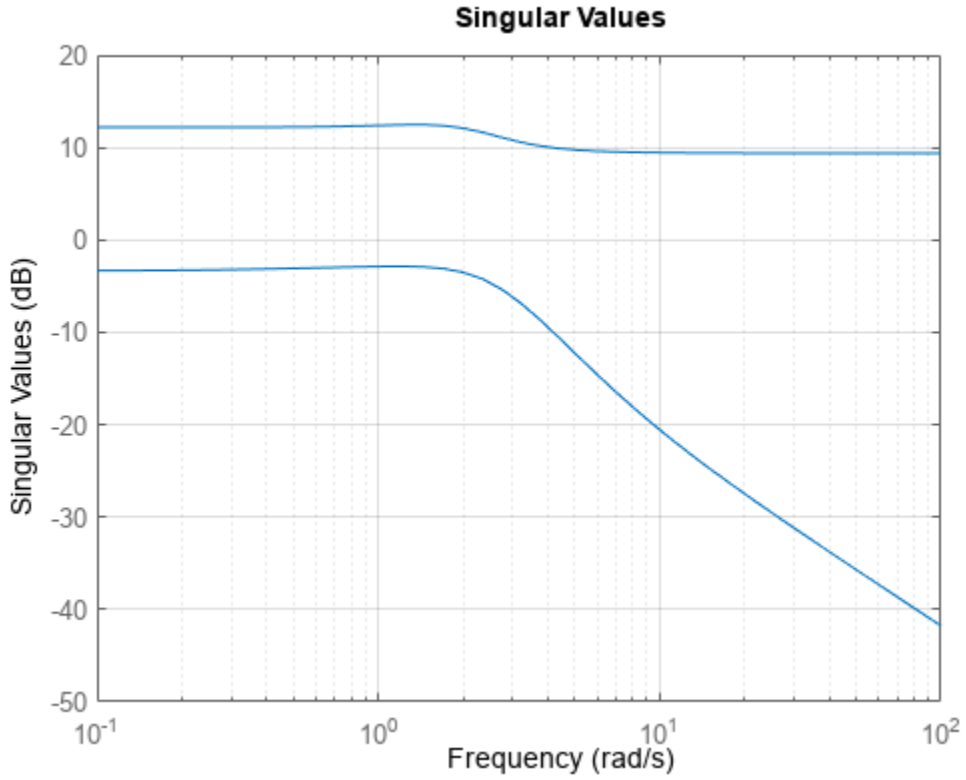


The following additional plots are useful for analyzing MIMO systems:

- Singular value plot (`sigma`), which shows the principal gains of the frequency response
- Pole/zero map for each I/O pair (`iopzplot`)

For example, plot the peak gain of `sys` as a function of frequency:

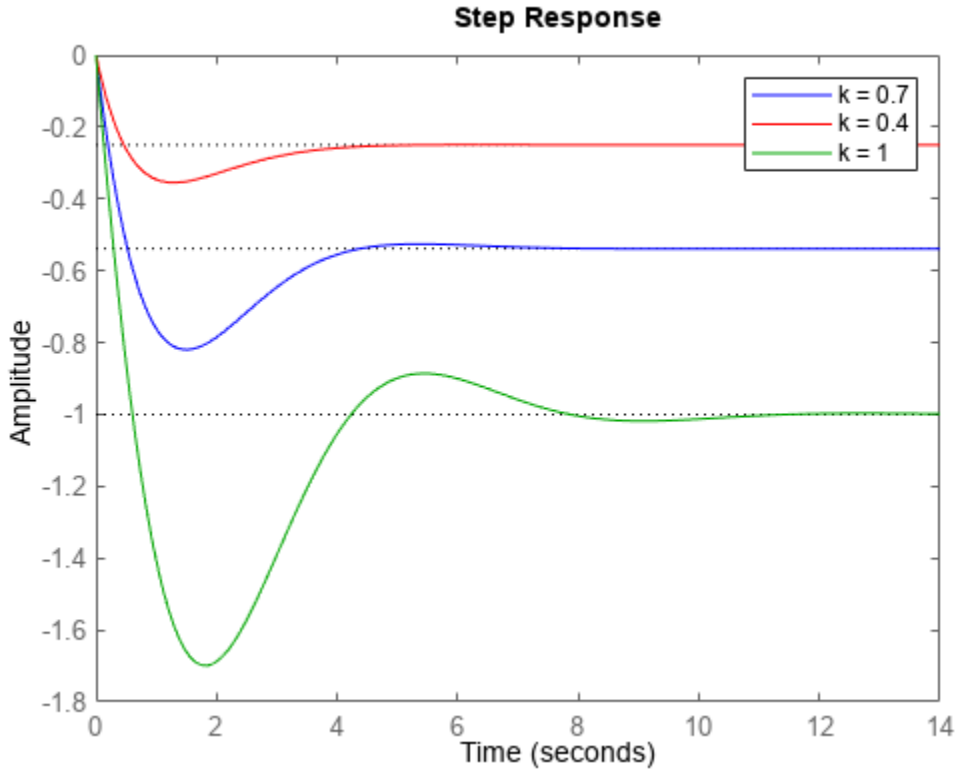
```
sigma(sys)
grid
```



Comparing Systems

You can plot multiple systems at once using any of the response plot commands. You can assign a specific color, marker, or line style to each system for easy comparison. Using the feedback example above, plot the closed-loop step response for three values of the loop gain k in three different colors:

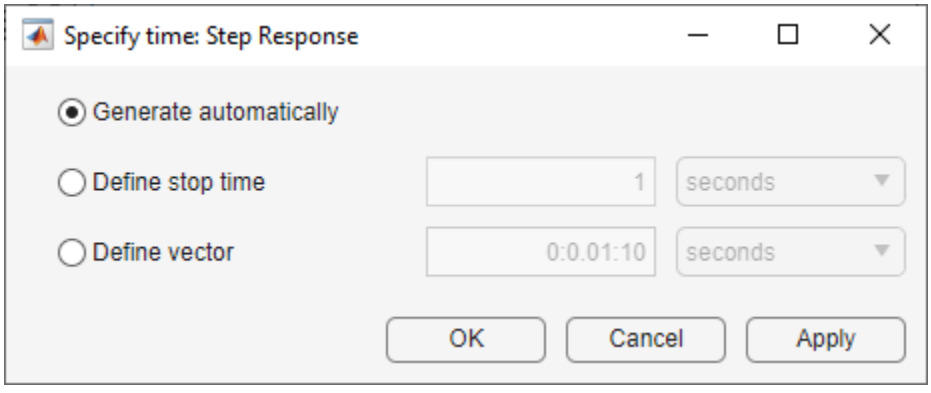
```
k1 = 0.4;
T1 = feedback(G*k1,1);
k2 = 1;
T2 = feedback(G*k2,1);
step(T,'b',T1,'r',T2,'g')
legend('k = 0.7','k = 0.4','k = 1')
```



Modify Time or Frequency Axis Values

You can modify the time and frequency vectors for existing linear analysis plots.

For `step` and `impulse` plots, you can specify the time vector by right-clicking the plot area and selecting **Specify time**.

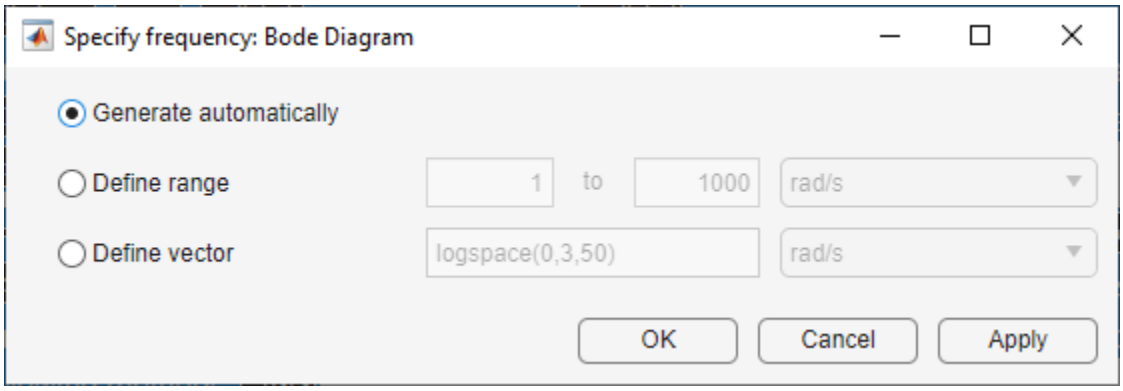


In the Specify time dialog box, you can define time values and units using one of the following methods.

- **Generate automatically** — Automatically generate the time vector based on the system dynamics. This option is not supported for sparse systems.

- **Define stop time** — Specify the stop time, which creates a time vector starting at time 0. The step size for the time vector is determined based on the system dynamics.
- **Define vector** — Specify the times to plot as a vector of monotonically increasing evenly spaced time values.

For `bode`, `nyquist`, `nichols`, and `sigma` plots, you can specify the frequency vector by right-clicking the plot area and selecting **Specify frequency**.



In the Specify frequency dialog box, you can define frequency values and units using one of the following methods.

- **Generate automatically** — Automatically generate the frequency vector based on the system dynamics. This method is not supported for sparse systems.
- **Define range** — Specify the frequency range. This method is not supported for sparse systems.
- **Define vector** — Specify the frequencies to plot as a vector.

If your system is an `frd` object, the plot interpolates the response between frequency values.

Changing the time and frequency units specifies the units for the input and does not change the units in the plot.

See Also

`bode` | `step`

More About

- “Time-Domain Responses” on page 7-19
- “Frequency-Domain Responses” on page 8-2

Time-Domain Responses

When you perform time-domain analysis of a dynamic system model, you may want one or more of the following:

- A plot of the system response as a function of time.
- Numerical values of the system response in a data array.
- Numerical values of characteristics of the system response such as peak response or settling time.

Control System Toolbox time-domain analysis commands can obtain these results for any kind of dynamic system model (for example, continuous or discrete, SISO or MIMO, or arrays of models) except for frequency response data models.

To obtain numerical data, use:

- `step`,`impulse`,`initial`,`lsim` — System response data at a vector of time points.
- `stepinfo`,`lsiminfo` — Numerical values of system response characteristics such as settling time and overshoot.

To obtain response plots, use:

- `step`,`impulse`,`initial`,`lsim` — Plot system response data, visualize response characteristics on plots, compare responses of multiple systems on a single plot.
- `stepplot`,`impulseplot`,`initialplot`,`lsimplot` — Create system response plots with more plot-customization options. For details about plot customization, see “Plot Customization”.
- **Linear System Analyzer** — App for plotting many types of system responses simultaneously, including both time-domain and frequency-domain responses

See Also

Related Examples

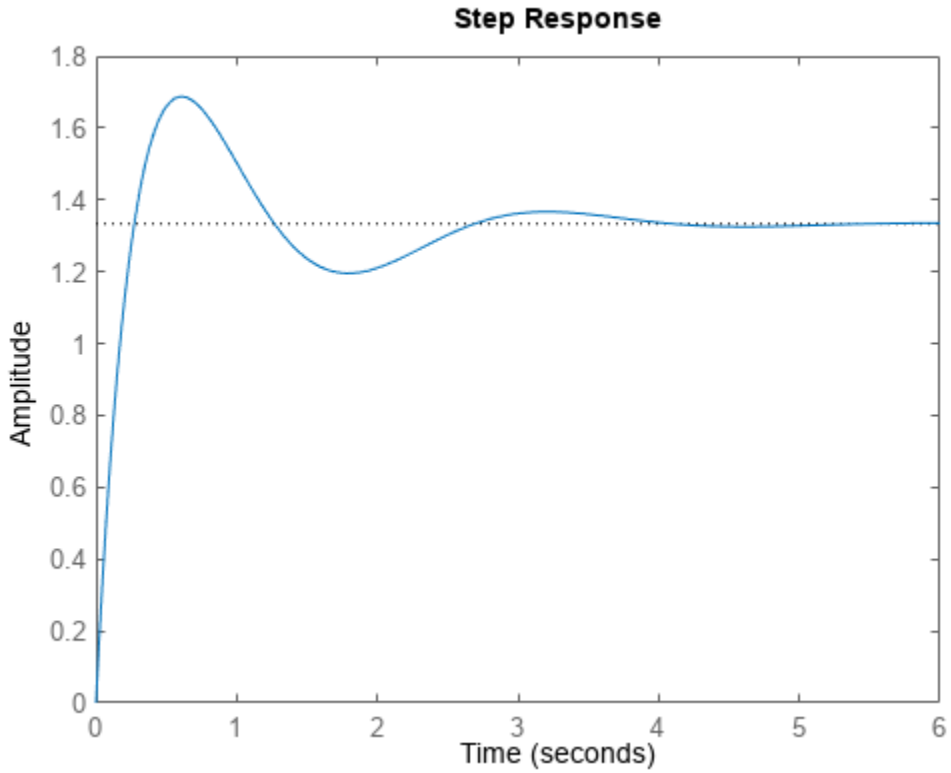
- “Time-Domain Response Data and Plots” on page 7-20
- “Joint Time-Domain and Frequency-Domain Analysis” on page 7-32

Time-Domain Response Data and Plots

This example shows how to obtain step and impulse response data, as well as step and impulse response plots, from a dynamic system model.

Create a transfer function model and plot its response to a step input at $t = 0$.

```
H = tf([8 18 32],[1 6 14 24]);
step(H);
```



When call `step` without output arguments, it plots the step response on the screen. Unless you specify a time range to plot, `step` automatically chooses a time range that illustrates the system dynamics.

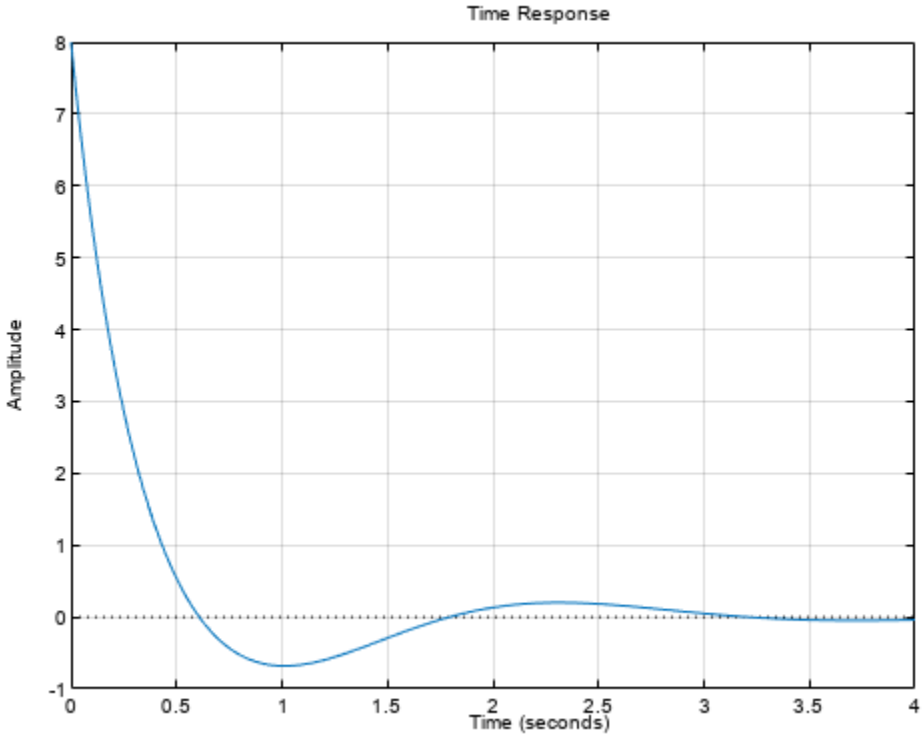
Calculate the step response data from $t = 0$ (application of the step input) to $t = 8$ s.

```
[y,t] = step(H,8);
```

When you call `step` with output arguments, the command returns the step response data `y`. The vector `t` contains corresponding time values.

Plot the response of `H` to an impulse input applied at $t = 0$. Plot the response with a grid.

```
opts = timeoptions;
opts.Grid = 'on';
impulseplot(H,opts)
```



Use the `timeoptions` command to define options sets for customizing time-domain plots with commands like `impulseplot` and `stepplot`.

Calculate 200 points of impulse response data from $t = 1$ (one second after application of the impulse input) to $t = 3$s.

```
[y,t] = impulse(H,linspace(1,3,200));
```

As for `step`, you can omit the time vector to allow `impulse` to automatically select a time range.

See Also

`step` | `impulse` | `stepplot` | `impulseplot` | `timeoptions`

Related Examples

- “Time-Domain Characteristics on Response Plots” on page 7-22
- “Time-Domain Responses of Multiple Models” on page 7-30
- “Joint Time-Domain and Frequency-Domain Analysis” on page 7-32

More About

- “Time-Domain Responses” on page 7-19

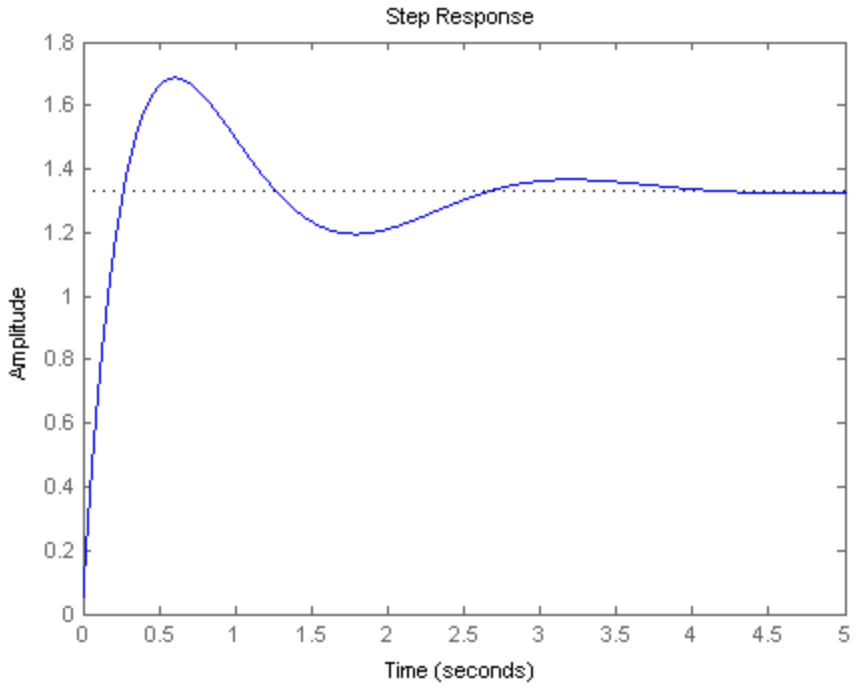
Time-Domain Characteristics on Response Plots

This example shows how to display system characteristics such as settling time and overshoot on step response plots.

You can use similar procedures to display system characteristics on impulse response plots or initial value response plots, such as peak response or settling time.

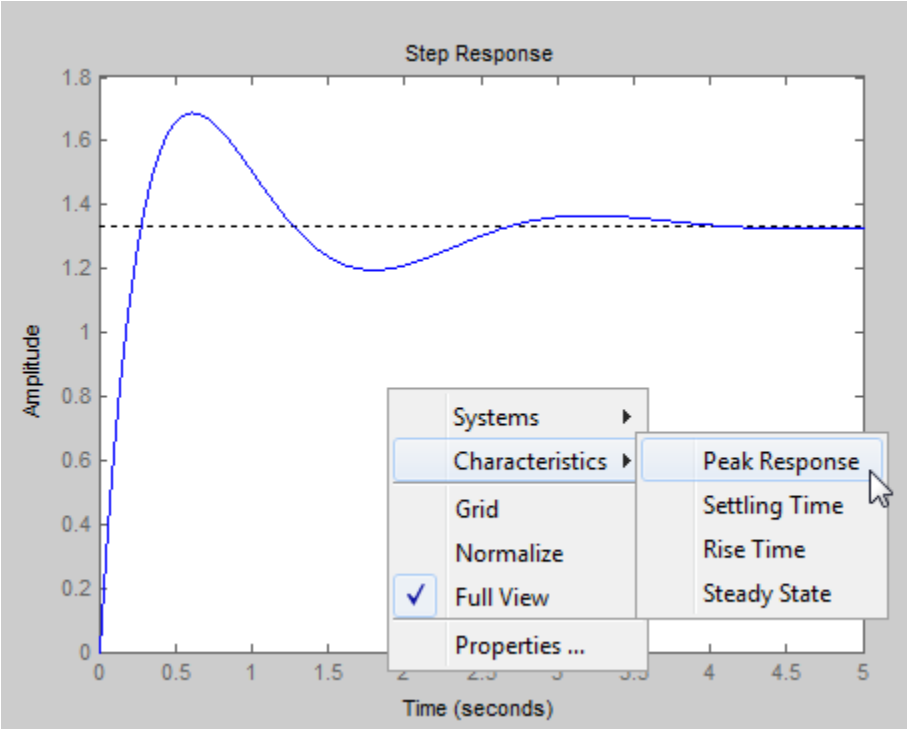
Create a transfer function model and plot its response to a step input at $t = 0$.

```
H = tf([8 18 32],[1 6 14 24]);
stepplot(H)
```

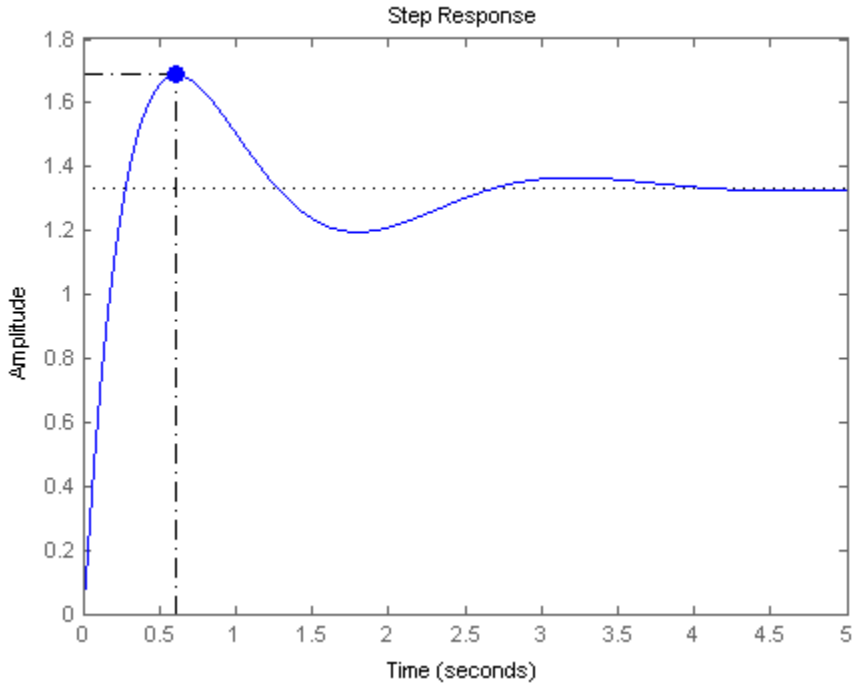


Display the peak response on the plot.

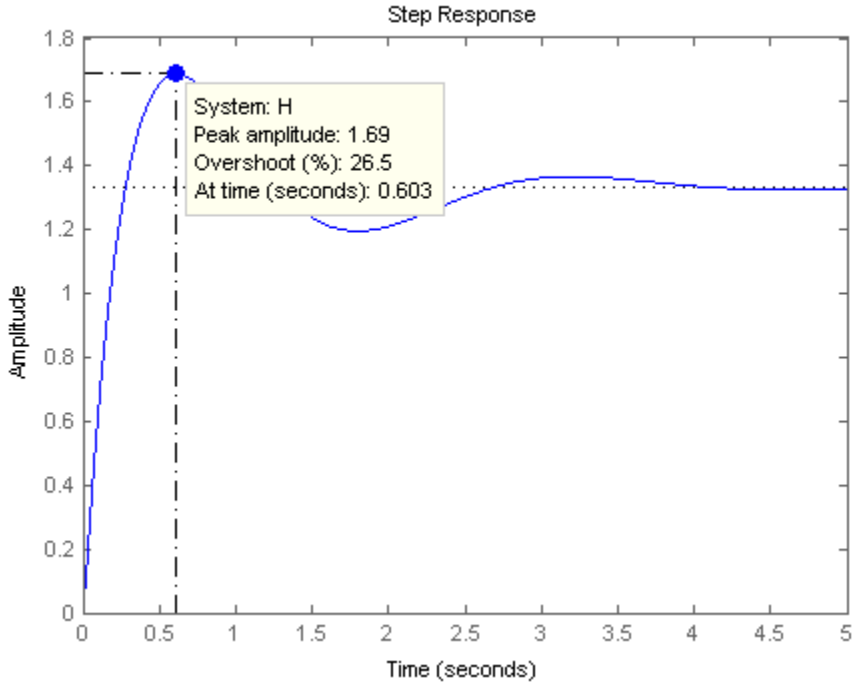
Right-click anywhere in the figure and select **Characteristics > Peak Response** from the menu.



A marker appears on the plot indicating the peak response. Horizontal and vertical dotted lines indicate the time and amplitude of that response.



Click the marker to view the value of the peak response and the overshoot in a datatip.



You can use a similar procedure to select other characteristics such as settling time and rise time from the **Characteristics** menu and view the values.

See Also

`step` | `impulse` | `stepinfo` | `lsiminfo`

Related Examples

- “Numeric Values of Time-Domain System Characteristics” on page 7-25
- “Joint Time-Domain and Frequency-Domain Analysis” on page 7-32

More About

- “Time-Domain Responses” on page 7-19

Numeric Values of Time-Domain System Characteristics

This example shows how to obtain numeric values of step response characteristics such as rise time, settling time, and overshoot using `stepinfo`. You can use similar techniques with `lsiminfo` to obtain characteristics of the system response to an arbitrary input or initial conditions.

Create a dynamic system model and get numeric values of the system's step response characteristics.

```
H = tf([8 18 32],[1 6 14 24]);
data = stepinfo(H)
```

```
data = struct with fields:
         RiseTime: 0.2087
    TransientTime: 3.4972
     SettlingTime: 3.4972
      SettlingMin: 1.1956
      SettlingMax: 1.6871
        Overshoot: 26.5302
       Undershoot: 0
             Peak: 1.6871
         PeakTime: 0.5987
```

The output is a structure that contains values for several step response characteristics. To access these values or refer to them in other calculations, use dot notation. For example, `data.Overshoot` is the overshoot value.

Calculate the time it takes the step response of `H` to settle within 0.5% of its final value.

```
data = stepinfo(H,'SettlingTimeThreshold',0.005);
t05 = data.SettlingTime
```

```
t05 = 4.8896
```

By default, `stepinfo` defines the settling time as the time it takes for the output to settle within 0.02 (2%) of its final value. Specifying a more stringent `'SettlingTimeThreshold'` of 0.005 results in a longer settling time.

For more information about the options and the characteristics, see the `stepinfo` reference page.

See Also

`stepinfo` | `lsiminfo`

Related Examples

- "Time-Domain Characteristics on Response Plots" on page 7-22
- "Joint Time-Domain and Frequency-Domain Analysis" on page 7-32

More About

- "Time-Domain Responses" on page 7-19

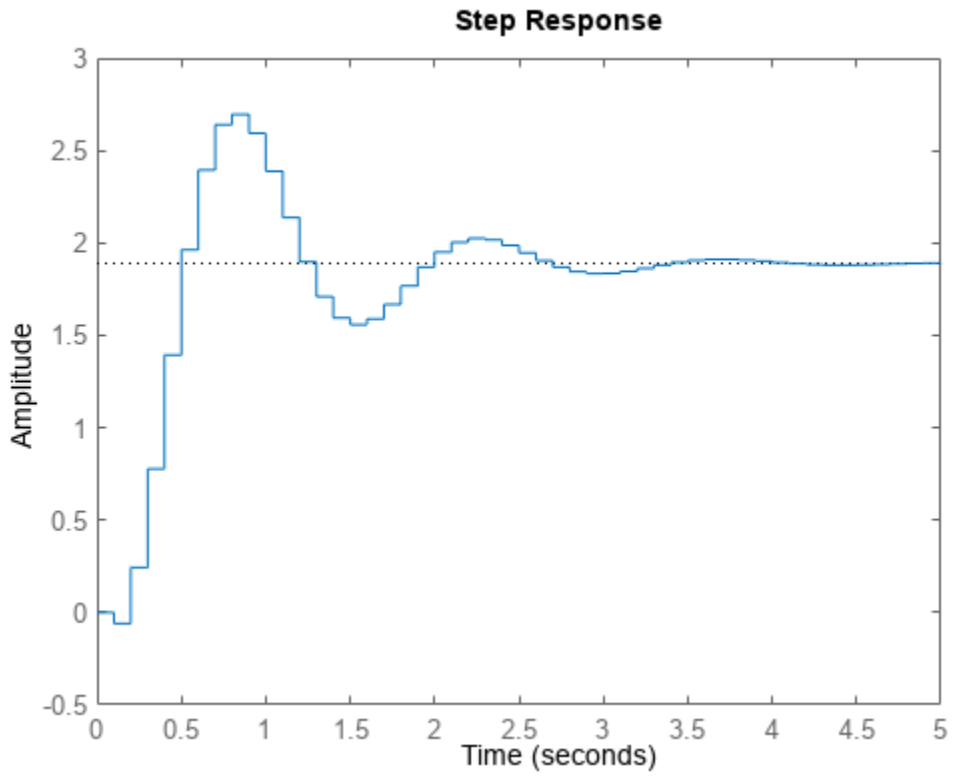
Time-Domain Responses of Discrete-Time Model

This example shows how to obtain a step-response plot and step-response data for a discrete-time dynamic system model. Obtaining time-domain responses of discrete-time models is the same as for continuous-time models, except that the time sample points are limited by the sample time `Ts` of the model.

You can use the techniques of this example with commands such as `impulse`, `initial`, `impulseplot`, and `initialpot` to obtain time-domain responses of discrete-time models.

Create a discrete-time transfer function model and plot its response to a step input at $t = 0$.

```
H = tf([-0.06,0.4],[1,-1.6,0.78],0.1);
step(H)
```



For discrete-time models, `step` plots the response at multiples of the sample time, assuming a hold between samples.

Compute the step response of `H` between 0.5 and 2.5 seconds.

```
[y,t] = step(H,0.5:0.1:2.5);
```

When you specify a time vector for the response of a discrete-time model, the time step must match the sample time `Ts` of the discrete-time model. The vector `t` contains the time points between 0.5 and 2.5 seconds, at multiples of the sample time of `H`, 0.1 s. The vector `y` contains the corresponding step response values.

See Also

`step` | `impulse` | `stepplot` | `initial` | `impulseplot` | `initialplot`

Related Examples

- “Time-Domain Responses of MIMO Model” on page 7-28
- “Time-Domain Responses of Multiple Models” on page 7-30
- “Joint Time-Domain and Frequency-Domain Analysis” on page 7-32

More About

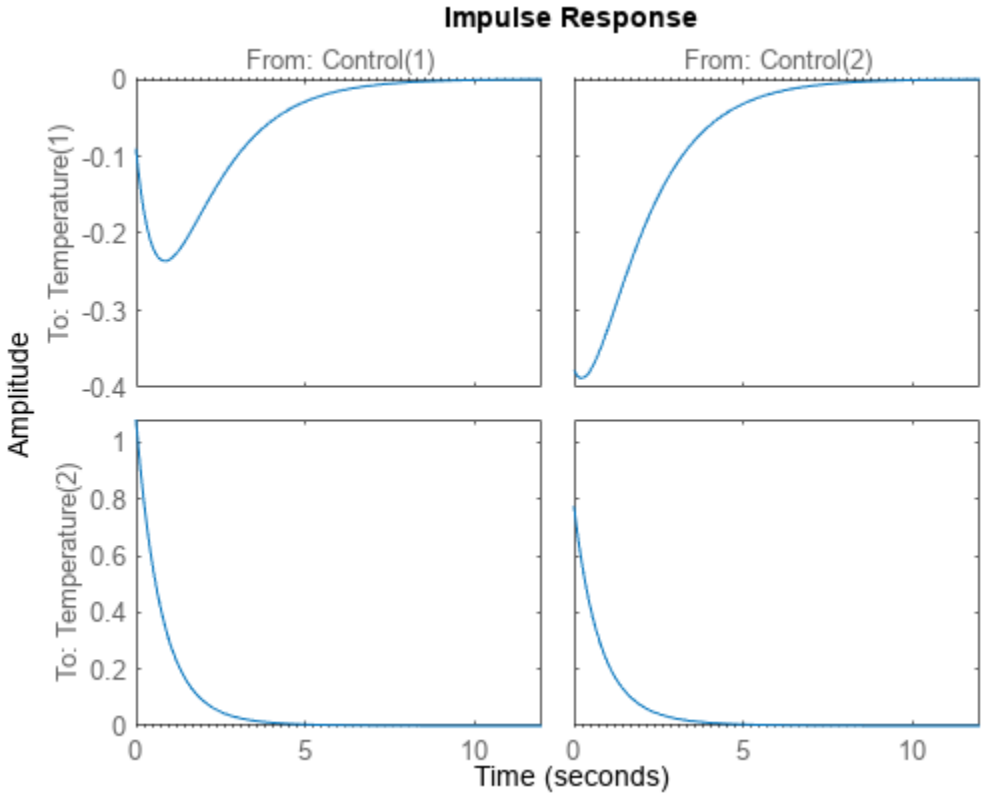
- “Time-Domain Responses” on page 7-19

Time-Domain Responses of MIMO Model

This example shows how to obtain impulse response data and plots for a multi-input, multi-output (MIMO) model using `impulse`. You can use the same techniques to obtain other types of time-domain responses of MIMO models.

Create a MIMO model and plot its response to a `t = 0` impulse at all inputs.

```
H = rss(2,2,2);
H.InputName = 'Control';
H.OutputName = 'Temperature';
impulse(H)
```



`impulse` plots the response of each output to an impulse applied at each input. (Because `rss` generates a random state-space model, you might see different responses from those pictured.) The first column of plots shows the response of each output to an impulse applied at the first input, `Control(1)`. The second column shows the response of each output to an impulse applied at the second input, `Control(2)`.

Calculate the impulse responses of all channels of `H`, and examine the size of the output.

```
[y,t] = impulse(H);
size(y)
```

```
ans = 1×3
```

```
   162     2     2
```

The first dimension of the data array `y` is the number of samples in the time vector `t`. The `impulse` command determines this number automatically if you do not supply a time vector. The remaining dimensions of `y` are the numbers of outputs and inputs in `H`. Thus, `y(:,i,j)` is the response at the `i` th output of `H` to an impulse applied at the `j` th input.

See Also

`step` | `initial` | `stepplot` | `initialplot` | `impulse` | `impulseplot`

Related Examples

- “Time-Domain Responses of Multiple Models” on page 7-30
- “Joint Time-Domain and Frequency-Domain Analysis” on page 7-32

More About

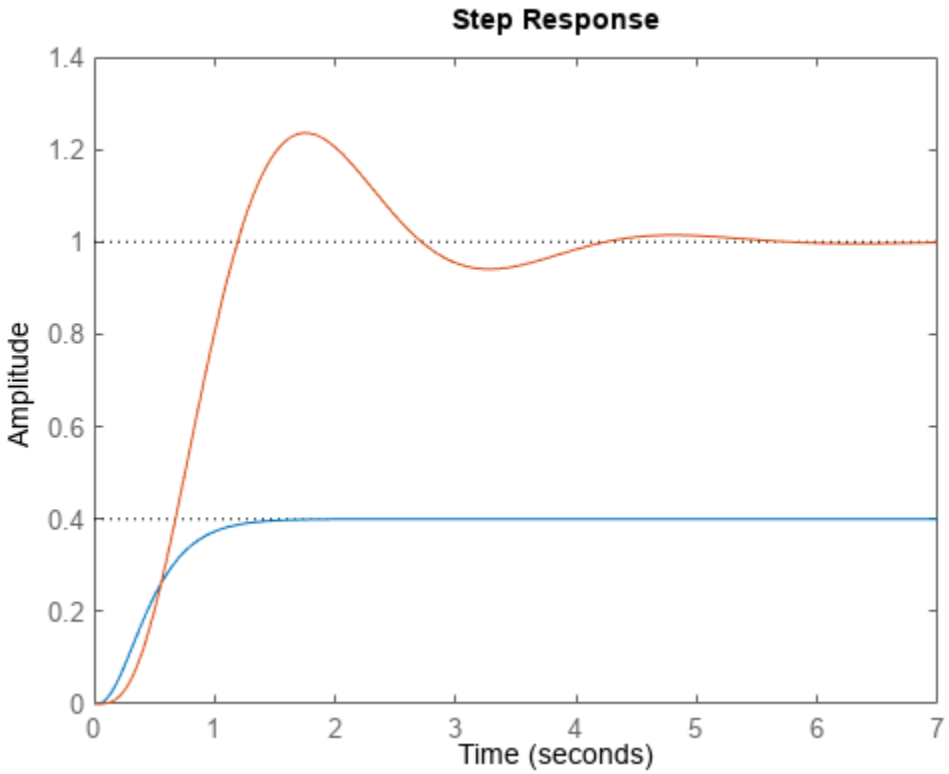
- “Time-Domain Responses” on page 7-19

Time-Domain Responses of Multiple Models

This example shows how to compare the step responses of multiple models on a single plot using `step`. This example compares the step response of an uncontrolled plant to the closed-loop step response of the plant with two different PI controllers. You can use similar techniques with other response commands, such as `impulse` or `initial`, to obtain plots of responses of multiple models.

For this example, obtain two models whose time responses you want to compare, and plot them on a single step plot. For instance, you can compare a third-order plant `G`, and the closed-loop response of `G` with a controller `C1` having integral action.

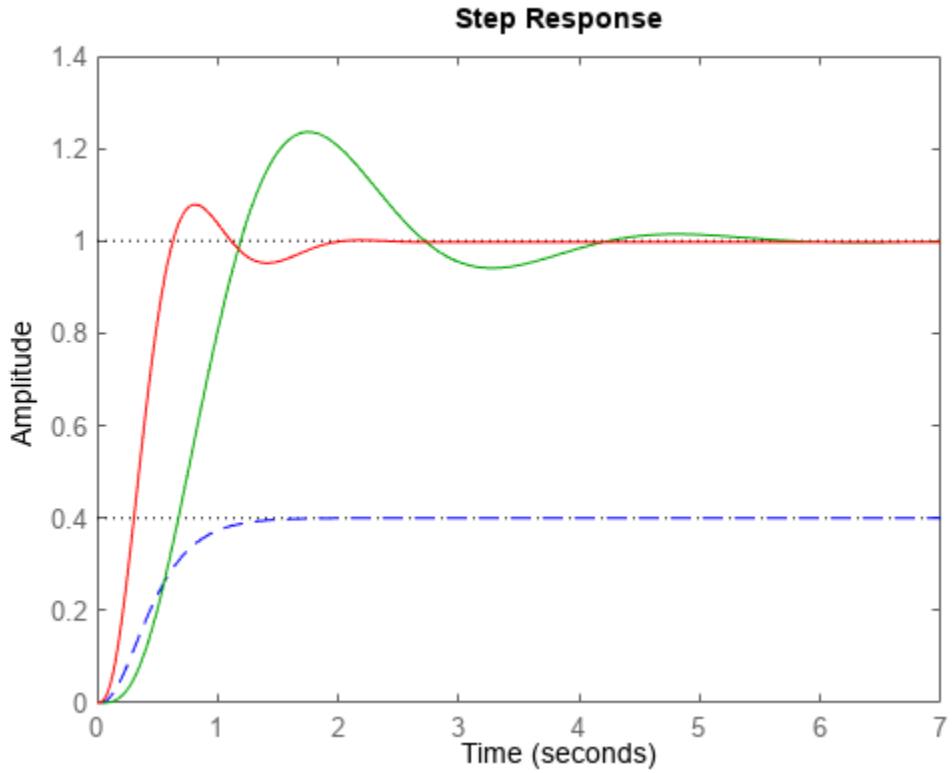
```
G = zpk([],[-5 -5 -10],100);

C1 = pid(0,4.4);
CL1 = feedback(G*C1,1);

step(G,CL1);
```



When you provide multiple models to `step` as input arguments, the command displays the responses of both models on the same plot. If you do not specify a time range to plot, `step` attempts to choose a time range that illustrates the dynamics of all the models.

Compare the step response of the closed-loop model with another controller. Specify plot colors and styles for each response.

```
C2 = pid(2.9,7.1);
CL2 = feedback(G*C2,1);

step(G,'b--',CL1,'g-',CL2,'r-')
```



You can specify custom plot color and style for each response in the plot. For example, `'g-'` specifies a solid green line for response `CL2`. For additional plot customization options, use `stepplot`.

See Also

`step` | `initial` | `stepplot` | `initialplot` | `impulse` | `impulseplot` | **Linear System Analyzer**

Related Examples

- “Time-Domain Responses of MIMO Model” on page 7-28
- “Joint Time-Domain and Frequency-Domain Analysis” on page 7-32

More About

- “Time-Domain Responses” on page 7-19

Joint Time-Domain and Frequency-Domain Analysis

This example shows how to compare multiple types of responses side by side, including both time-domain and frequency-domain responses, using the interactive Linear System Analyzer app.

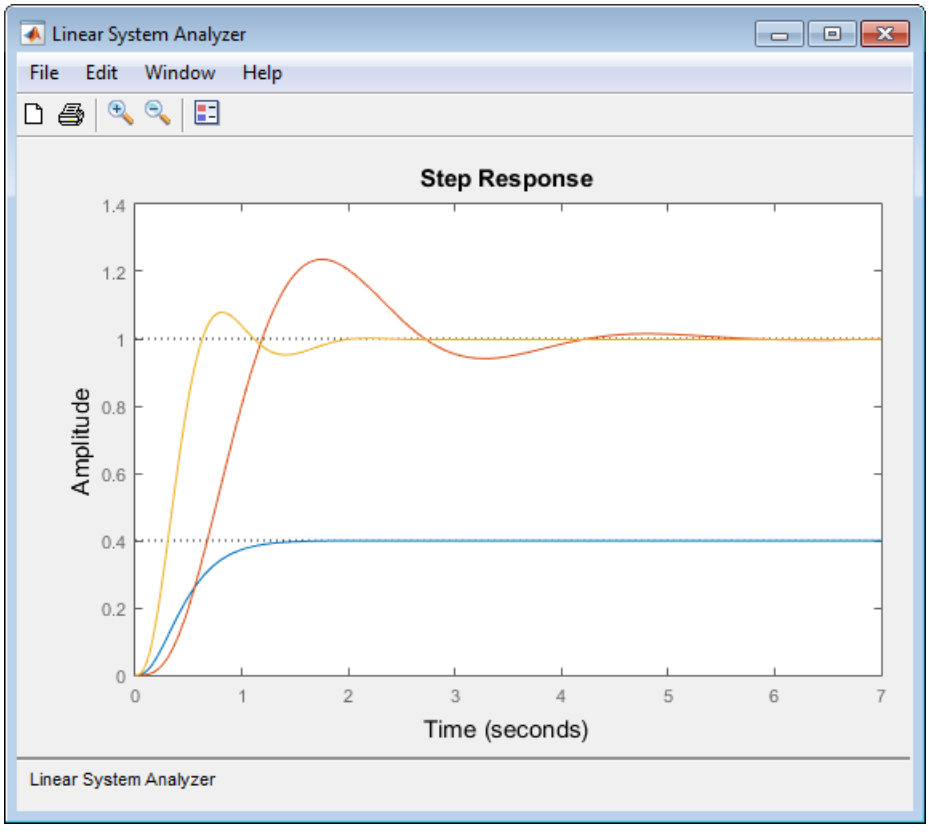
Obtain models whose responses you want to compare.

For example, compare a third-order plant `G`, and the closed-loop responses of `G` with two different controllers, `C1` and `C2`.

```
G = zpk([],[-5 -5 -10],100);
C1 = pid(0,4.4);
T1 = feedback(G*C1,1);
C2 = pid(2.9,7.1);
T2 = feedback(G*C2,1);
```

Open the Linear System Analyzer tool to examine the responses of the plant and the closed-loop systems.

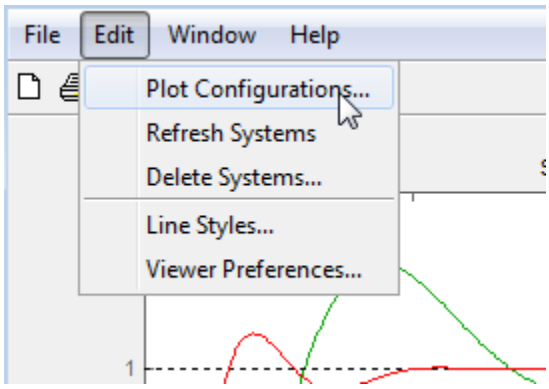
```
linearSystemAnalyzer(G,T1,T2)
```



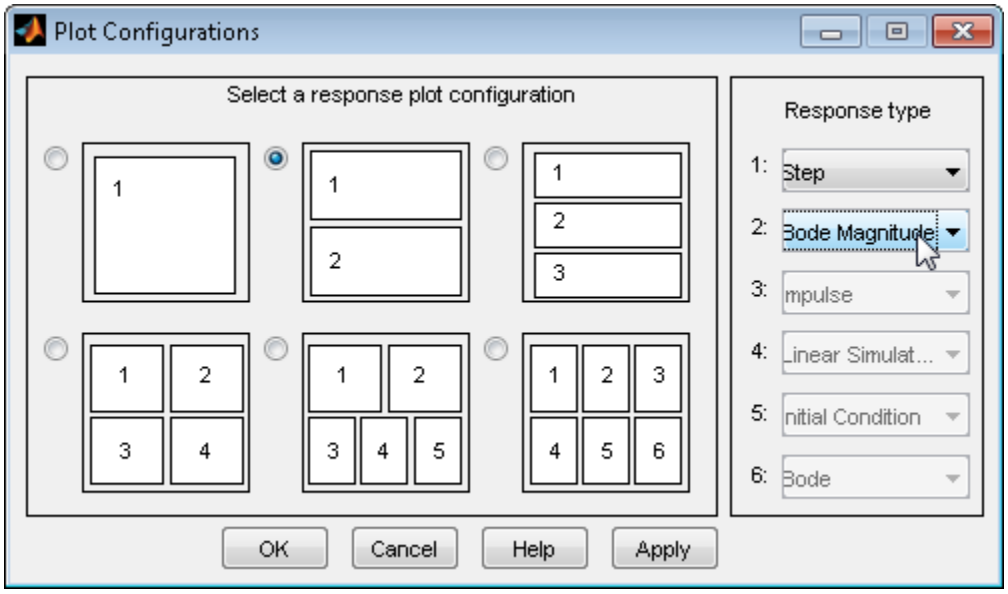
By default, the Linear System Analyzer launches with a plot of the step response of the three systems. Click  to add a legend to the plot.

Add plots of the impulse responses to the Linear System Analyzer display.

In the Linear System Analyzer, select **Edit** > **Plot Configurations** to open the Plot Configurations dialog box.



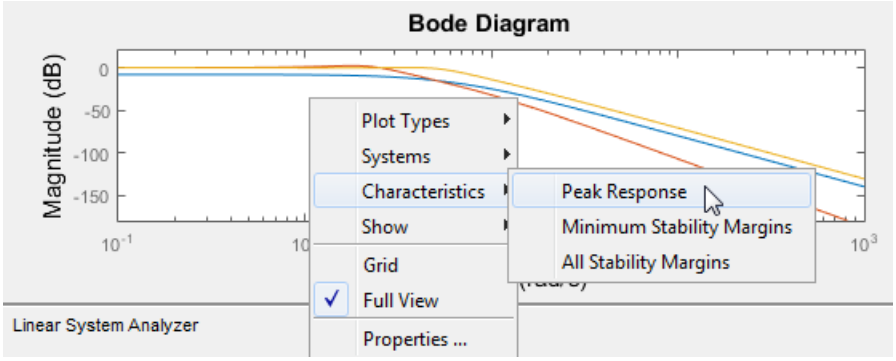
Select the two-plot configuration. In the Response Type area, select `Bode Magnitude` for the second plot type.



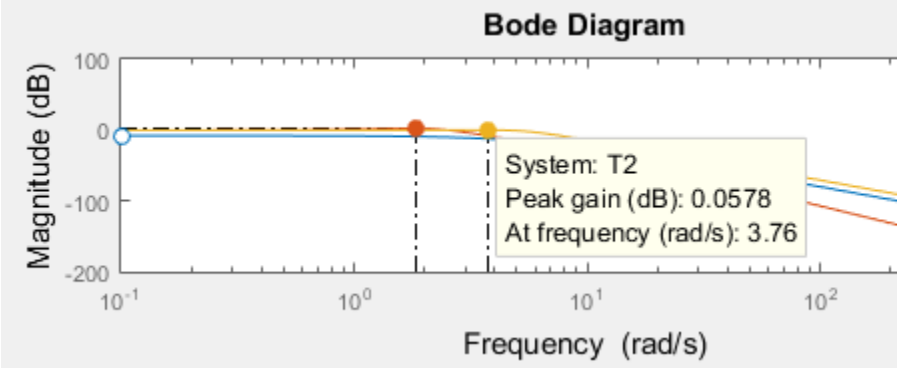
Click **OK** to add the Bode plots to the Linear System Analyzer display.

Display the peak values of the Bode responses on the plot.

Right-click anywhere in the Bode Magnitude plot and select **Characteristics** > **Peak Response** from the menu.

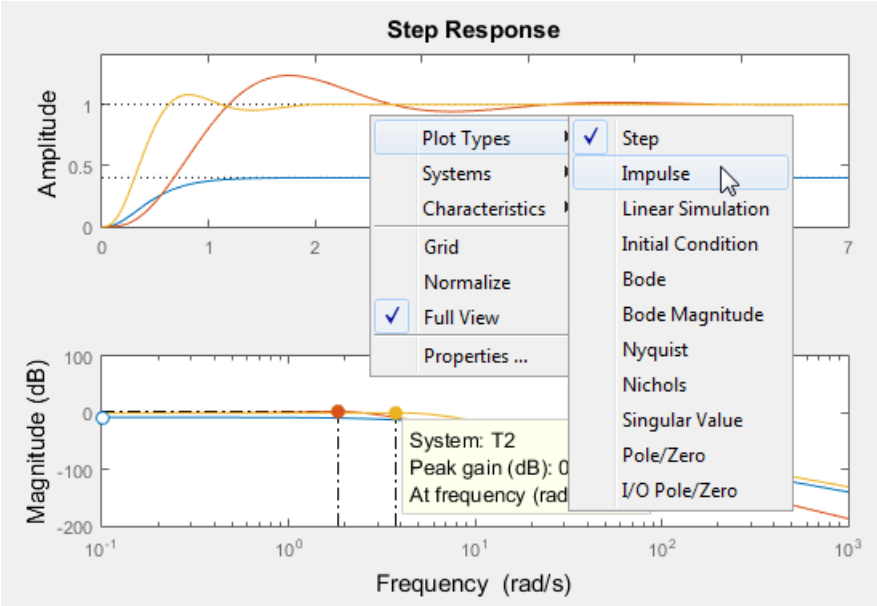


Markers appear on the plot indicating the peak response values. Horizontal and vertical dotted lines indicate the frequency and amplitude of those responses. Click on a marker to view the value of the peak response in a datatip.



You can use a similar procedure to select other characteristics such as settling time and rise time from the **Characteristics** menu and view the values.

You can also change the type of plot displayed in the Linear System Analyzer. For example, to change the first plot type to a plot of the impulse response, right-click anywhere in the plot. Select **Plot Types > Impulse**



The displayed plot changes to show the impulse of the three systems.

See Also

Linear System Analyzer | `step` | `initial` | `stepplot` | `initialplot` | `impulse` | `impulseplot`

Related Examples

- “Time-Domain Responses of Multiple Models” on page 7-30

More About

- “Time-Domain Responses” on page 7-19

Response from Initial Conditions

This example shows how to compute and plot the response of a state-space (`ss`) model to specified initial state values using `initial`.

Load a state-space model.

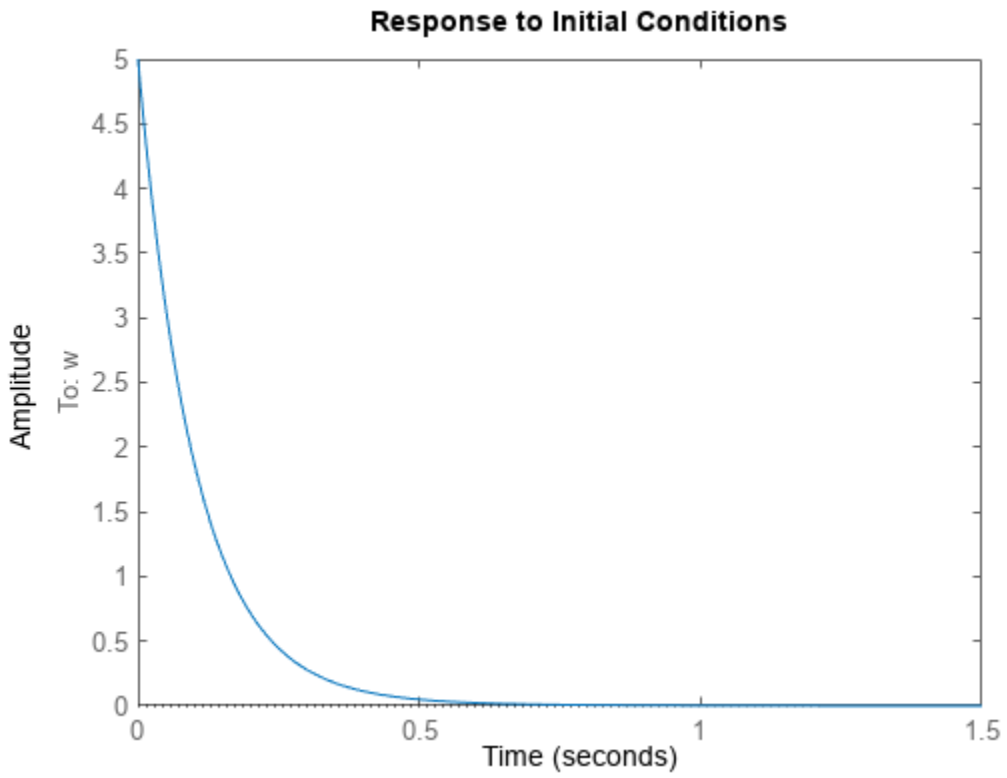
```
load ltiexamples sys_dc
sys_dc.InputName = 'Volts';
sys_dc.OutputName = 'w';
sys_dc.StateName = {'Current','w'};
sys_dc
```

```
sys_dc =

  A =
            Current         w
   Current       -4    -0.03
   w           0.75      -10

  B =
            Volts
   Current      2
   w            0

  C =
       Current        w
   w         0        1

  D =
       Volts
   w       0

Continuous-time state-space model.
```

This example uses the SISO, 2-state model `sys_dc`. This model represents a DC motor. The input is an applied voltage, and the output is the angular rate of the motor ω. The states of the model are the induced current *i* (`x1`), and ω (`x2`). The model display in the command window shows the labeled input, output, and states.

Plot the undriven evolution of the motor's angular rate from an initial state in which the induced current is 1.0 amp and the initial rotation rate is 5.0 rad/s.

```
x0 = [1.0 5.0];
initial(sys_dc,x0)
```



`initial` plots the time evolution from the specified initial condition on the screen. Unless you specify a time range to plot, `initial` automatically chooses a time range that illustrates the system dynamics.

Calculate the time evolution of the output and the states of `sys_dc` from $t = 0$ (application of the step input) to $t = 1$ s.

```
t = 0:0.01:1;
[y,t,x] = initial(sys_dc,x0,t);
```

The vector `y` contains the output at each time step in `t`. The array `x` contains the state values at each time step. Therefore, in this example `x` is a 2-by-101 array. Each row of `x` contains the values of the two states of `sys_dc` at the corresponding time step.

See Also

`step` | `impulse` | `initial` | `initialplot`

Related Examples

- “Time-Domain Response Data and Plots” on page 7-20
- “Numeric Values of Time-Domain System Characteristics” on page 7-25

More About

- “Time-Domain Responses” on page 7-19

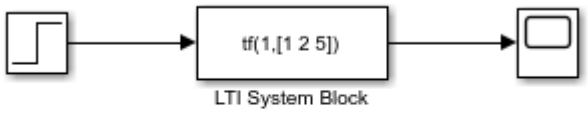
Import LTI Model Objects into Simulink

Use the LTI System block to import linear system model objects into Simulink. You can simulate linear systems represented as LTI model objects, and incorporate such systems as elements of Simulink models of more complex systems.

In the block parameters, set the **LTI system variable** parameter to the LTI model to import. For state-space models, set the **Initial states** parameter to a vector to specify non-zero initial states.

Simulate LTI Model in Simulink

The `LTISystemBlockSimulation` model shows how to use an LTI System block to simulate the response of a SISO transfer function to a step input.



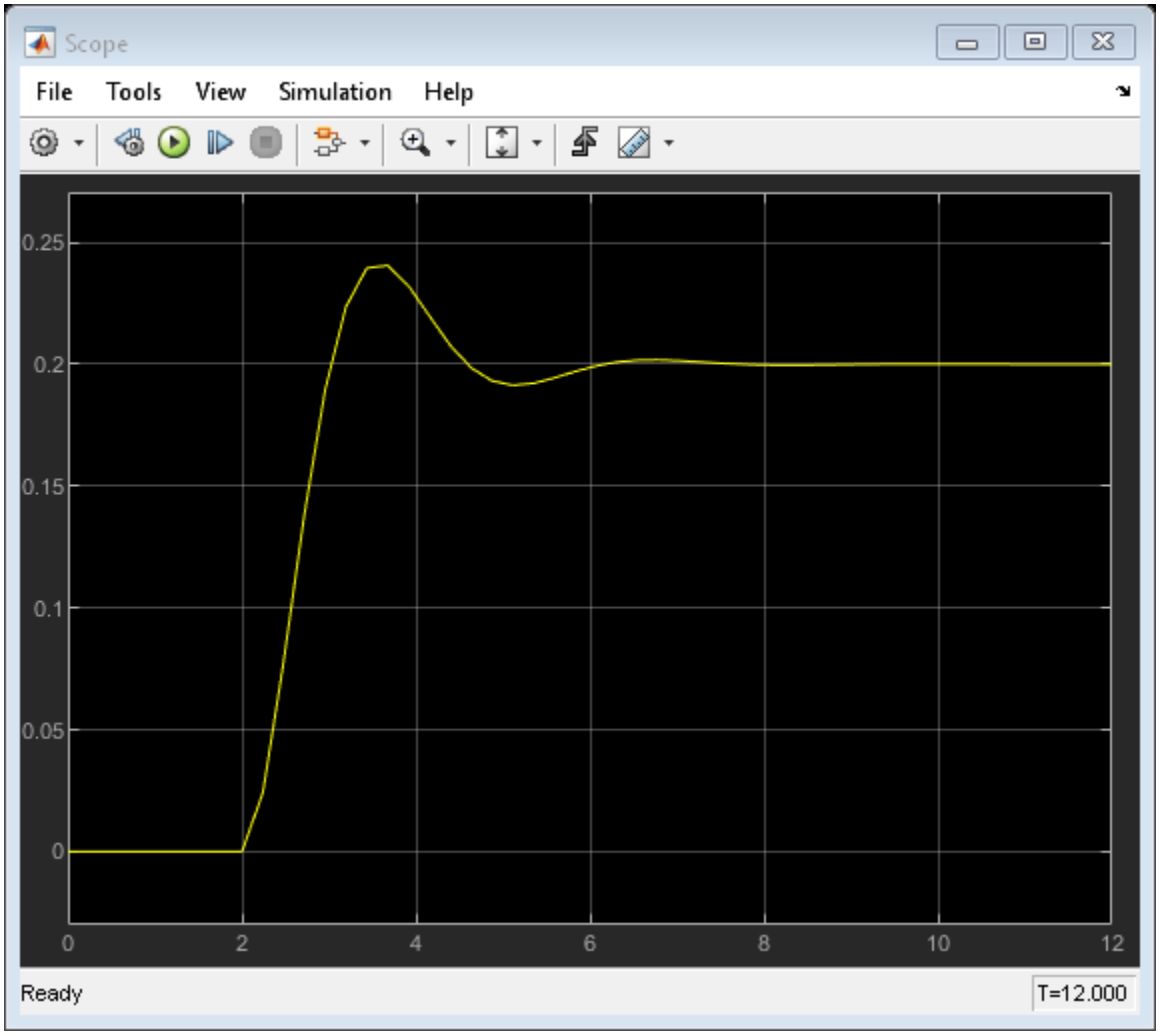
To specify a model for the LTI System block, set the **LTI system variable** block parameter to either:

- The variable name of an LTI model in the MATLAB® workspace or model workspace, such as `sys`.
- A MATLAB expression that evaluates to an LTI model, such as `tf(1,[1 1])`.

For example, you can specify a state-space (`ss`), zero-pole-gain (`zpk`), or transfer function (`tf`) model. You can simulate SISO models or MIMO models, and continuous-time or discrete-time models.

In `LTISystemBlockSimulation` model, the **LTI system variable** parameter is a MATLAB expression, `tf(1,[1 2 5])`, which creates a continuous-time SISO transfer function. If the specified system is a state-space (`ss`) model, then you can specify initial state values by setting the **Initial states** parameter.

Simulate the model, and examine the result in the scope.

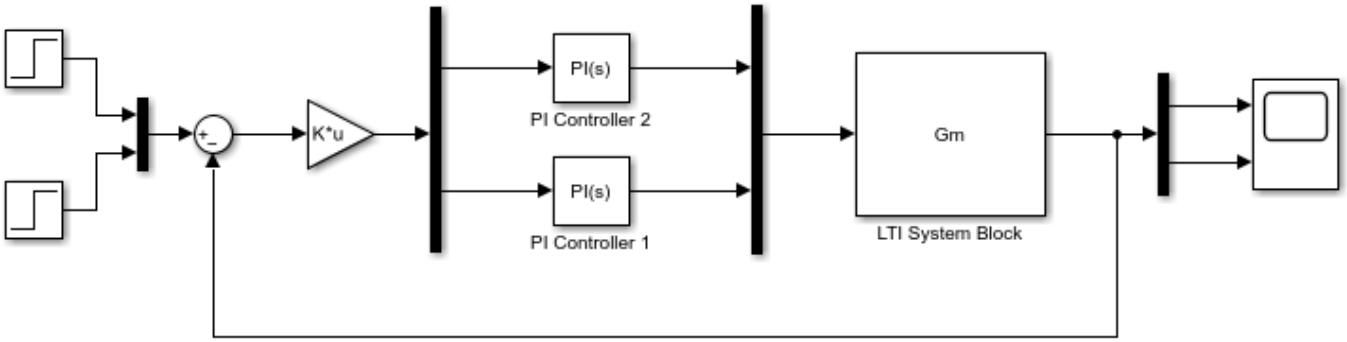


This example simulates the system response to a step input at $t = 2$ s. Use the LTI System block to import an LTI model object anywhere in your Simulink model to simulate the linear system response to any input.

Import MIMO LTI Model into Simulink

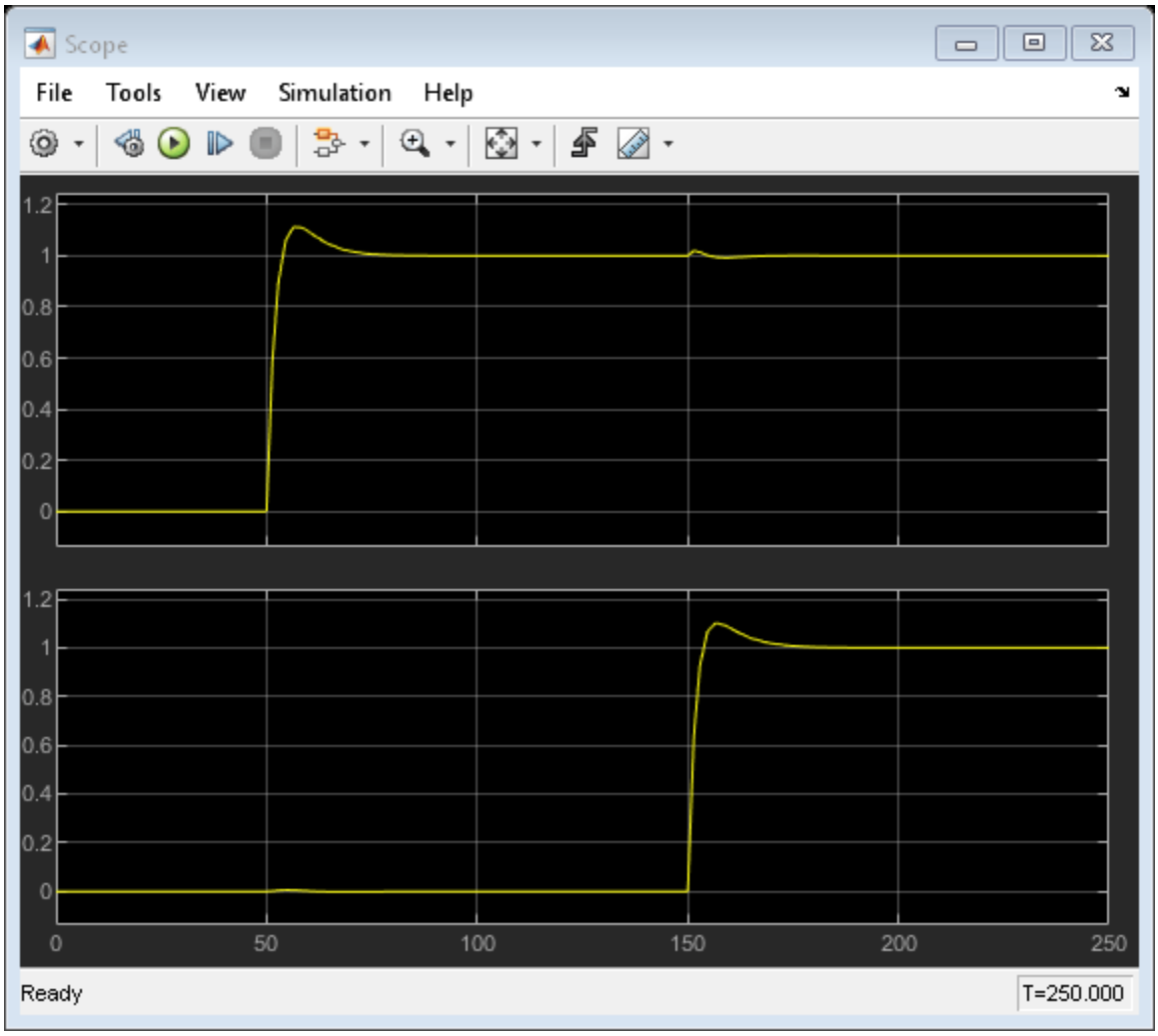
This model shows how to use an LTI System block to represent a MIMO linear system in Simulink®.

The LTI System block has one input and one output, even when you specify a MIMO model for the block. In that case, the block input and output become vector signals. For instance, the model `LTISystemBlockMIMO` uses an LTI system block to represent a MIMO plant in a control system.



In this model, the LTI System specified in the block is `Gm`, a 2-output, 2-input transfer function model stored in the model workspace. A Mux block combines the two controller outputs into a vector signal for the LTI System block input. Similarly, a Demux block separates the vector output of the LTI System block into two scalar signals.

Simulate the model, and examine the result in the scope.



This example simulates a closed-loop system response to a t = 50 s step at the first input and a t = 150 s step at the second input. You can use the LTI system block anywhere you want to insert an LTI system into a Simulink model.

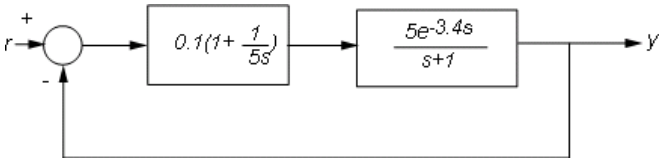
See Also

LTI System

Analysis of Systems with Time Delays

You can use analysis commands such as `step`, `bode`, or `margin` to analyze systems with time delays. The software makes no approximations when performing such analysis.

For example, consider the following control loop, where the plant is modeled as first-order plus dead time:



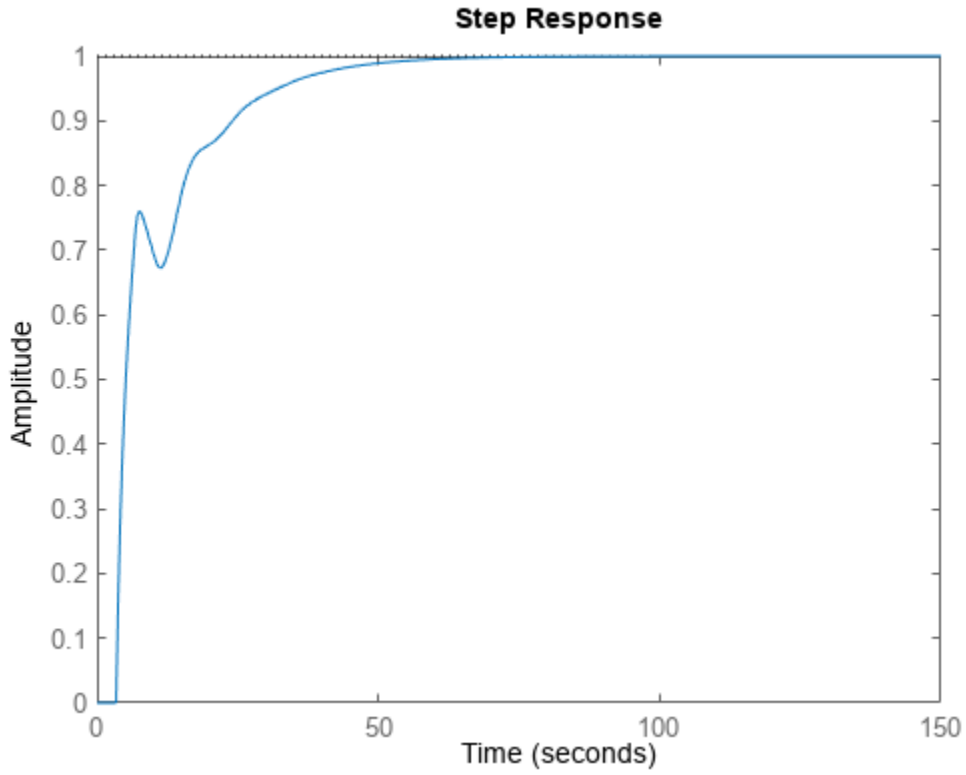
You can model the closed-loop system from `r` to `y` with the following commands:

```
s = tf('s');
P = 5*exp(-3.4*s)/(s+1);
C = 0.1 * (1 + 1/(5*s));
T = feedback(P*C,1);
```

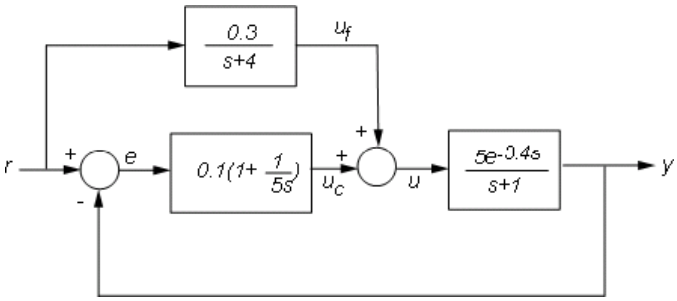
`T` is a state-space model with an internal delay. For more information about models with internal delays, see “Closing Feedback Loops with Time Delays” on page 2-35.

Plot the step response of `T`:

```
stepplot(T)
```



For more complicated interconnections, you can name the input and output signals of each block and use `connect` to automatically take care of the wiring. Suppose, for example, that you want to add feedforward to the control loop of the previous model.

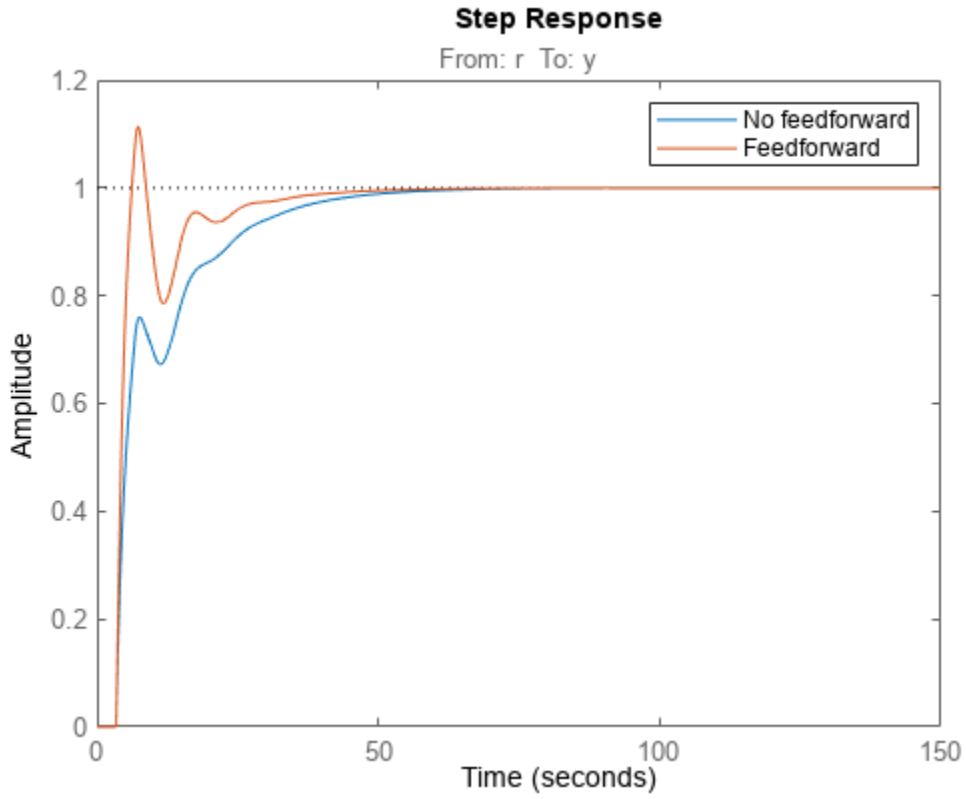


You can derive the corresponding closed-loop model `Tff` by

```
F = 0.3/(s+4);
P.InputName = 'u';
P.OutputName = 'y';
C.InputName = 'e';
C.OutputName = 'uc';
F.InputName = 'r';
F.OutputName = 'uf';
Sum1 = sumblk('e','r','y','+-');    % e = r-y
Sum2 = sumblk('u','uf','uc','++');  % u = uf+uc
Tff = connect(P,C,F,Sum1,Sum2,'r','y');
```

and compare its response with the feedback only design.

```
stepplot(T,Tff)
legend('No feedforward','Feedforward')
```



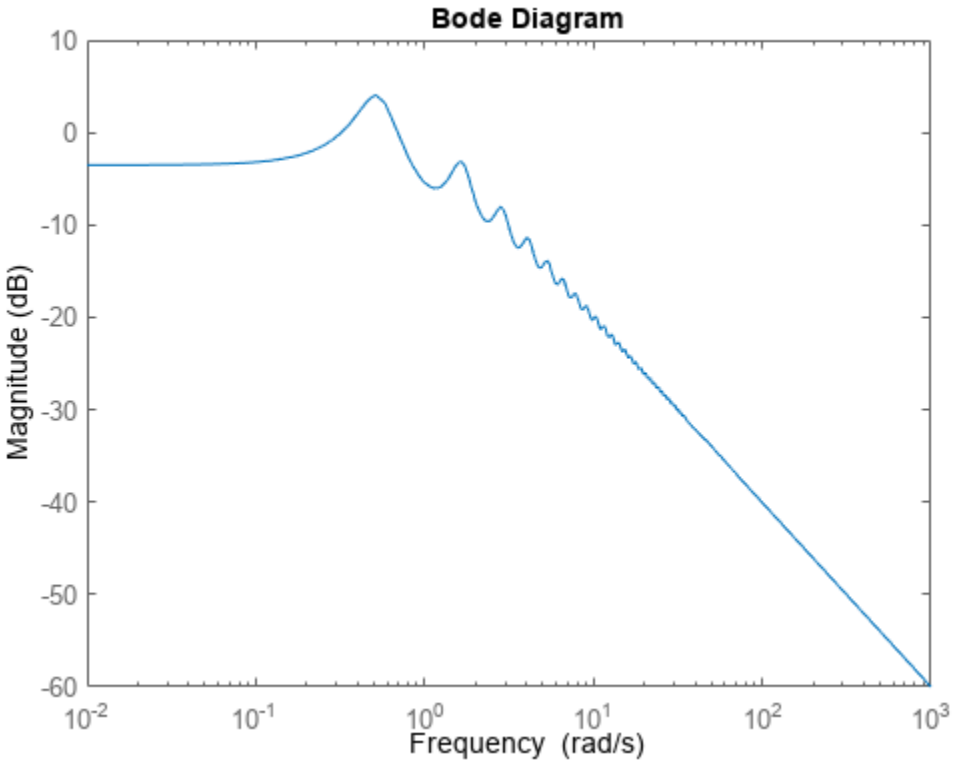
The state-space representation keeps track of the internal delays in both models.

Considerations to Keep in Mind when Analyzing Systems with Internal Time Delays

The time and frequency responses of delay systems can look odd and suspicious to those only familiar with delay-free LTI analysis. Time responses can behave chaotically, Bode plots can exhibit gain oscillations, etc. These are not software or numerical quirks but real features of such systems. Below are a few illustrations of these phenomena.

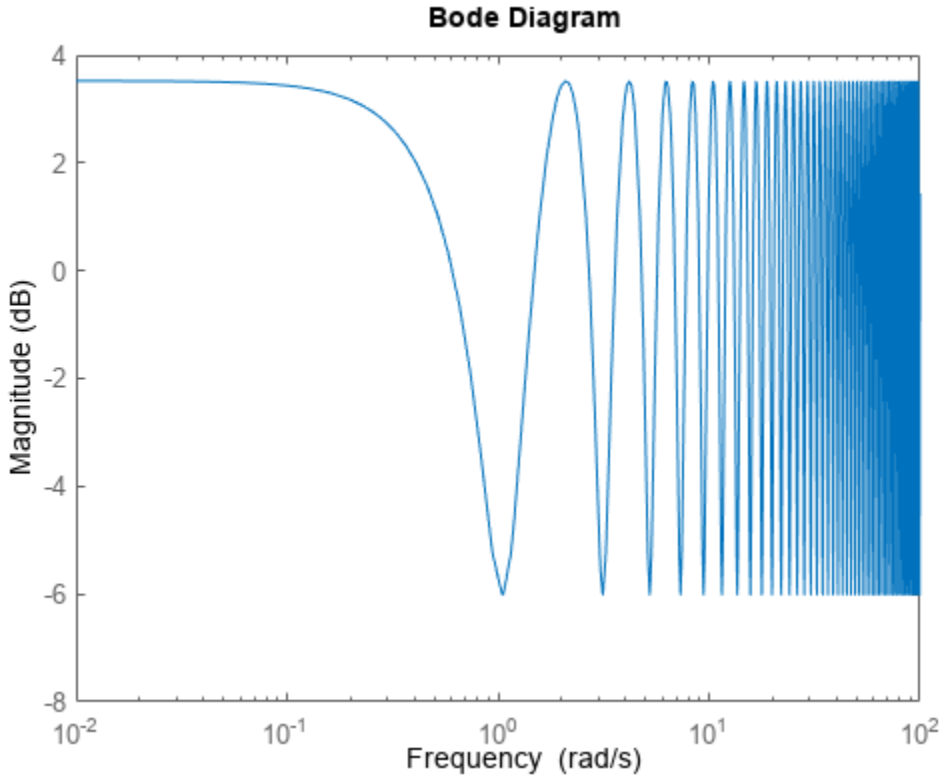
Gain ripple:

```
s = tf('s');
G = exp(-5*s)/(s+1);
T = feedback(G,.5);
bodemag(T)
```



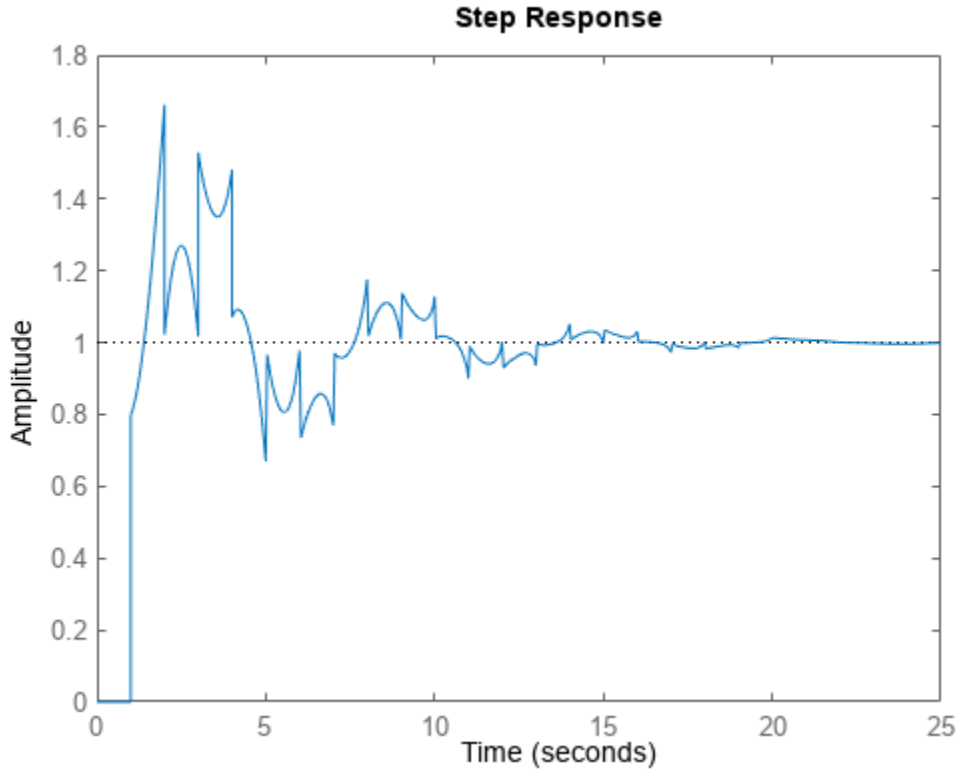
Gain oscillations:

```
G = 1 + 0.5 * exp(-3*s);
bodemag(G)
```



Jagged step response:

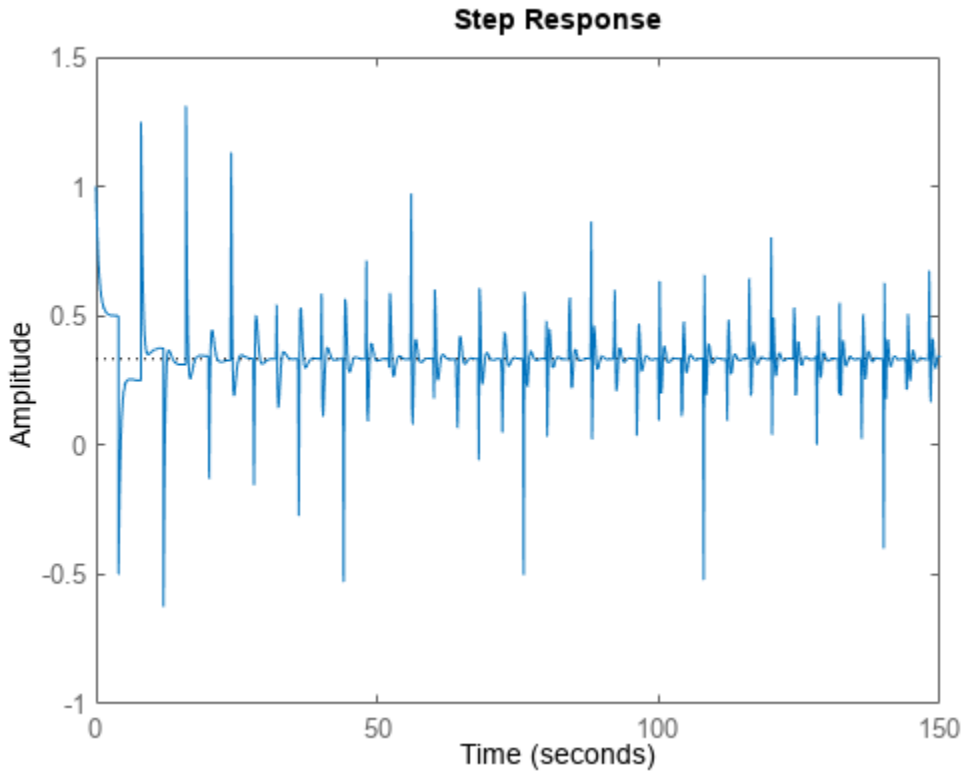
```
G = exp(-s) * (0.8*s^2+s+2)/(s^2+s);
T = feedback(G,1);
stepplot(T)
```



Note the rearrivals (echoes) of the initial step function.

Chaotic response:

```
G = 1/(s+1) + exp(-4*s);
T = feedback(1,G);
stepplot(T,150)
```



You can use Control System Toolbox tools to model and analyze these and other strange-appearing artifacts of internal delays.

See Also

Related Examples

- “Closing Feedback Loops with Time Delays” on page 2-35

More About

- “Time Delays in Linear Systems” on page 2-31
- “Internal Delays” on page 2-55

Frequency Domain Analysis

- “Frequency-Domain Responses” on page 8-2
- “Frequency Response of a SISO System” on page 8-3
- “Frequency Response of a MIMO System” on page 8-5
- “Frequency-Domain Characteristics on Response Plots” on page 8-8
- “Numeric Values of Frequency-Domain Characteristics of SISO Model” on page 8-11
- “Pole and Zero Locations” on page 8-13
- “Assessing Gain and Phase Margins” on page 8-15
- “Analyzing Control Systems with Delays” on page 8-26
- “Analyzing the Response of an RLC Circuit” on page 8-41

Frequency-Domain Responses

When you perform frequency-domain analysis of a dynamic system model, you may want one or more of the following:

- A plot of the system response as a function of frequency, or plots of pole and zero locations.
- Numerical values of the system response in a data array.
- Numerical values of characteristics of the system response such as stability margins, peak gains, or singular values.

Control System Toolbox frequency-domain analysis commands can obtain these results for any kind of dynamic system model (for example, continuous or discrete, SISO or MIMO, or arrays of models).

To obtain numerical data, use:

- `bode`,`freqresp`,`nichols`,`nyquist` — System response data at a vector of frequency points.
- `margin`,`getPeakGain`,`getGainCrossover`,`sigma` — Numerical values of system response characteristics such as gain margins, phase margins, and singular values.

To obtain response plots, use:

- `bode`,`bodemag`,`nichols`,`nyquist` — Plot system response data, visualize response characteristics on plots, compare responses of multiple systems on a single plot.
- `bodeplot`,`nicholsplot`,`nyquistplot`,`sigmaplot` — Create system response plots with more plot-customization options. For details about plot customization, see “Plot Customization”.
- **Linear System Analyzer** — App for plotting many types of system responses simultaneously, including both time-domain and frequency-domain responses

To obtain pole-zero maps, use:

- `pzplot`, `iopzplot` — Plot pole and zero locations in the complex plane.

If you have a generalized state-space (`genss`) model of a control system, you can extract various transfer functions from it for analysis using frequency-domain and time-domain analysis commands. Extract responses from such models using `getIOTransfer`, `getLoopTransfer`, `getSensitivity`, and `getCompSensitivity`.

See Also

Related Examples

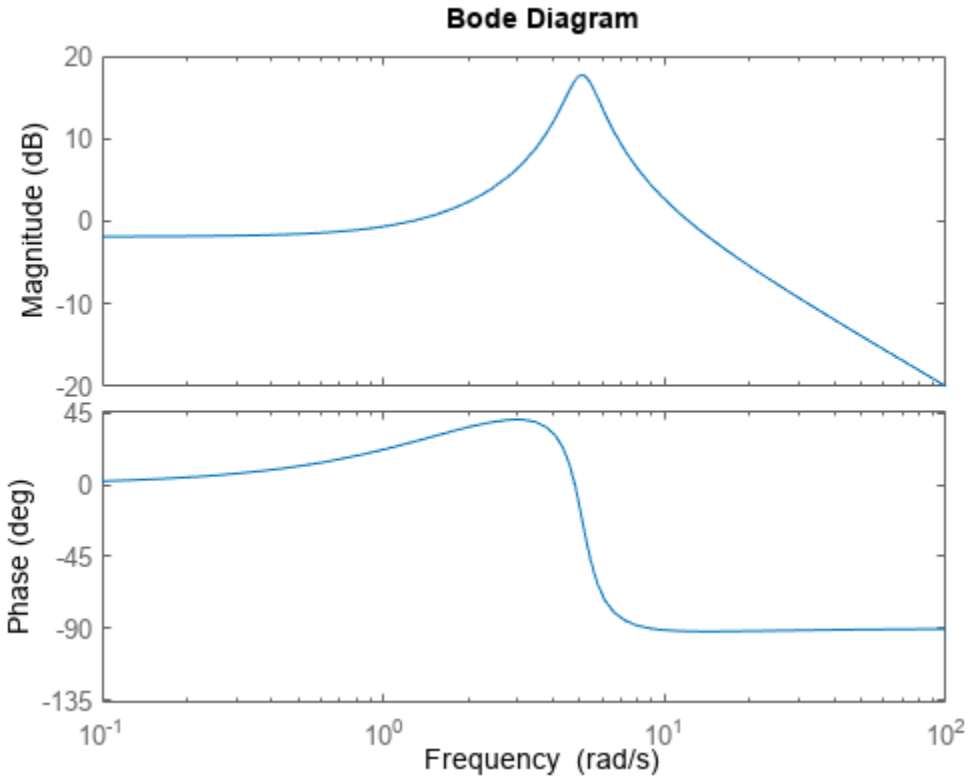
- “Frequency Response of a SISO System” on page 8-3
- “Frequency Response of a MIMO System” on page 8-5
- “Joint Time-Domain and Frequency-Domain Analysis” on page 7-32

Frequency Response of a SISO System

This example shows how to plot the frequency response and obtain frequency response data for a single-input, single-output (SISO) dynamic system model.

Create a transfer function model and plot its frequency response.

```
H = tf([10,21],[1,1.4,26]);
bode(H)
```



When you can `bode` without output arguments, it plots the frequency response on the screen. Unless you specify a frequency range to plot, `bode` automatically chooses a frequency range based on the system dynamics.

Calculate the frequency response between 1 and 13 rad/s.

```
[mag,phase,w] = bode(H,{1,13});
```

When you call `bode` with output arguments, the command returns vectors `mag` and `phase` containing the magnitude and phase of the frequency response. The cell array input `{1,13}` tells `bode` to calculate the response at a grid of frequencies between 1 and 13 rad/s. `bode` returns the frequency points in the vector `w`.

See Also

`bode` | `bodeplot` | `bodeoptions`

Related Examples

- “Frequency Response of a MIMO System” on page 8-5
- “Numeric Values of Frequency-Domain Characteristics of SISO Model” on page 8-11

More About

- “Frequency-Domain Responses” on page 8-2

Frequency Response of a MIMO System

This example shows how to examine the frequency response of a multi-input, multi-output (MIMO) system in two ways: by computing the frequency response, and by computing the singular values.

Calculate the frequency response of a MIMO model and examine the size of the output.

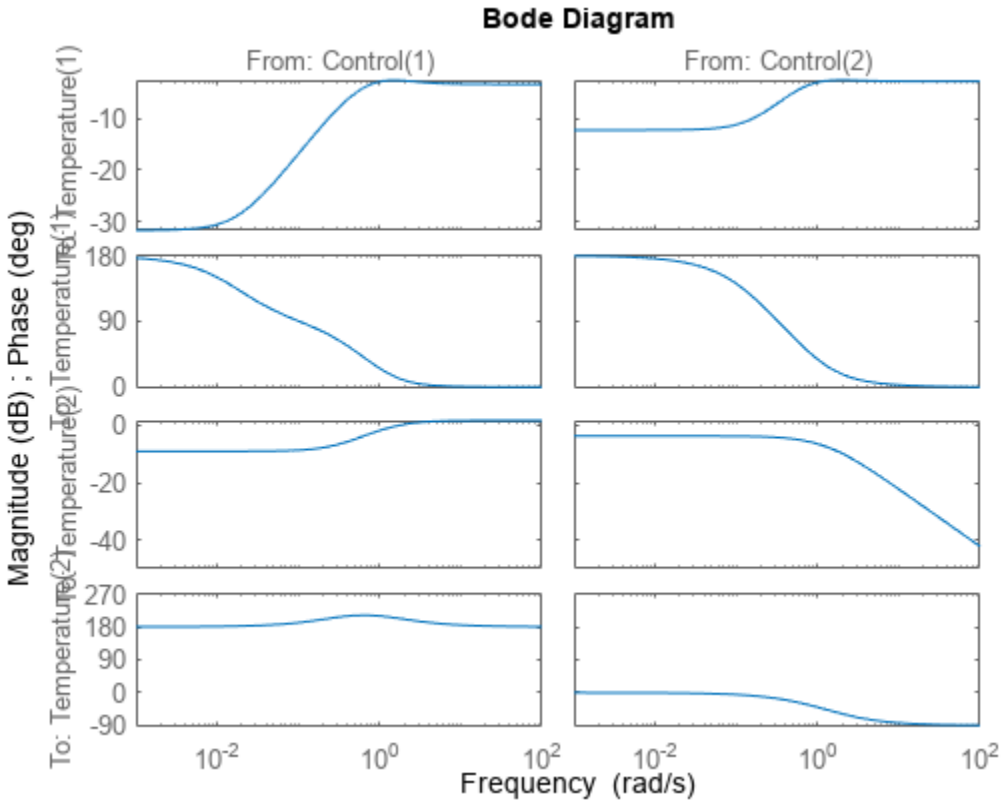
```
H = rss(2,2,2);
H.InputName = 'Control';
H.OutputName = 'Temperature';
[mag,phase,w] = bode(H);
size(mag)
```

```
ans = 1×3

     2     2    89
```

The first and second dimension of the data array `mag` are the number of outputs and inputs of `H`. The third dimension is the number of points in the frequency vector `w`. (The `bode` command determines this number automatically if you do not supply a frequency vector.) Thus, `mag(i,j,:)` is the frequency response from the `j` th input of `H` to the `i` th output, in absolute units. The phase data array `phase` takes the same form as `mag`.

Plot the frequency response of each input/output pair in `H`.

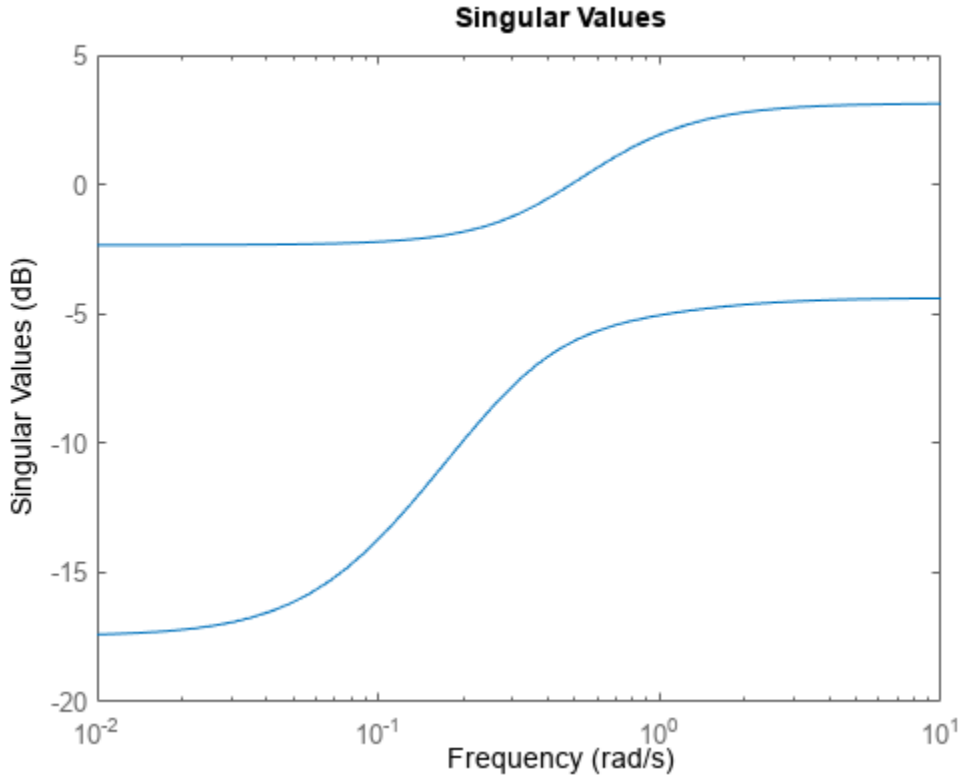
```
bode(H)
```



`bode` plots the magnitude and the phase of the frequency response of each input/output pair in `H`. (Because `rss` generates a random state-space model, you might see different responses from those pictured.) The first column of plots shows the response from the first input, `Control(1)`, to each output. The second column shows the response from the second input, `Control(2)`, to each output.

Plot the singular values of `H` as a function of frequency.

```
sigma(H)
```



`sigma` plots the singular values of the MIMO system `H` as a function of frequency. The maximum singular value at a particular frequency is the maximum gain of the system over all linear combinations of inputs at that frequency. Singular values can provide a better indication of the overall response, stability, and conditioning of a MIMO system than a channel-by-channel Bode plot.

Calculate the singular values of `H` between 0.1 and 10 rad/s.

```
[sv,w] = sigma(H,{0.1,10});
```

When you call `sigma` with output arguments, the command returns the singular values in the data array `sv`. The cell array input `{0.1,10}` tells `sigma` to calculate the singular values at a grid of frequencies between 0.1 and 10 rad/s. `sigma` returns these frequencies in the vector `w`. Each row of `sv` contains the singular values of `H` at the frequencies of `w`.

See Also

`bode` | `bodeplot` | `sigma` | `sigmaplot`

Related Examples

- “Numeric Values of Frequency-Domain Characteristics of SISO Model” on page 8-11
- “Joint Time-Domain and Frequency-Domain Analysis” on page 7-32

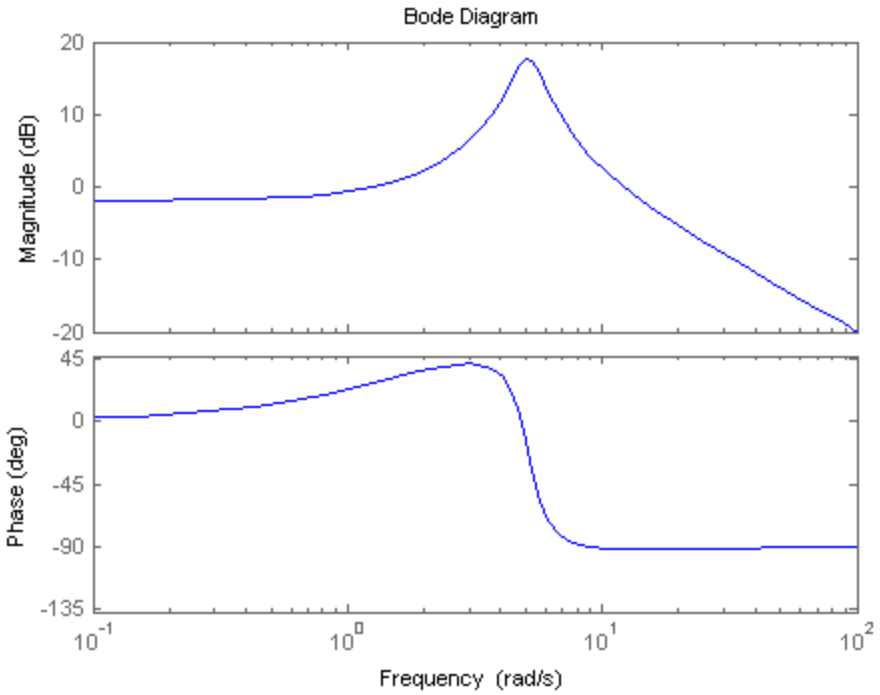
Frequency-Domain Characteristics on Response Plots

This example shows how to display system characteristics such as peak response on Bode response plots.

You can use similar procedures to display system characteristics on other types of response plots.

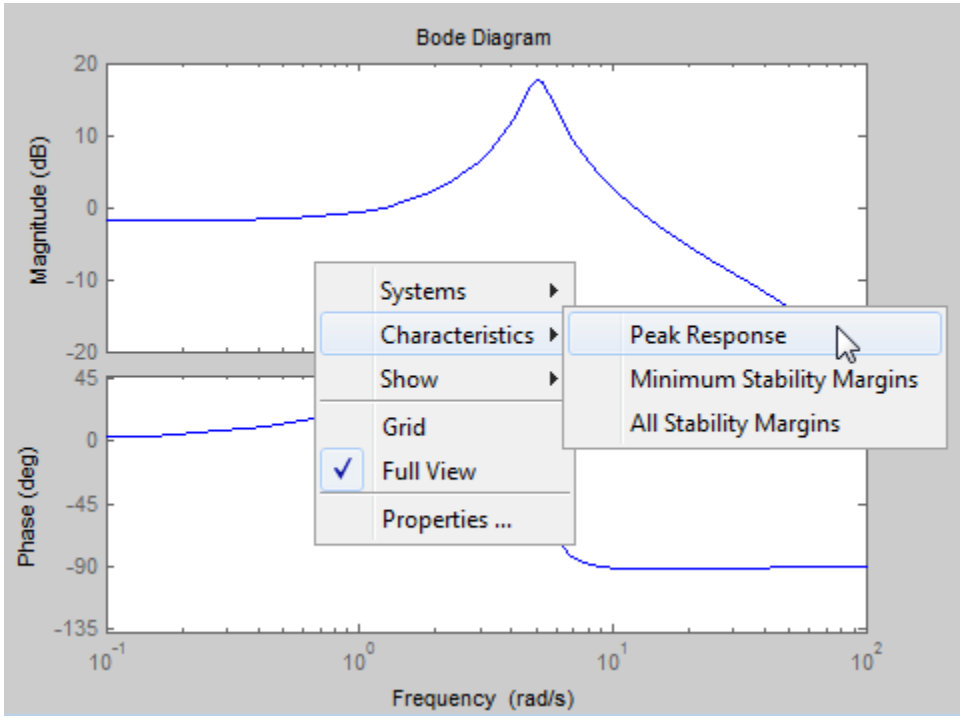
Create a transfer function model and plot its frequency response.

```
H = tf([10,21],[1,1.4,26]);
bodeplot(H)
```

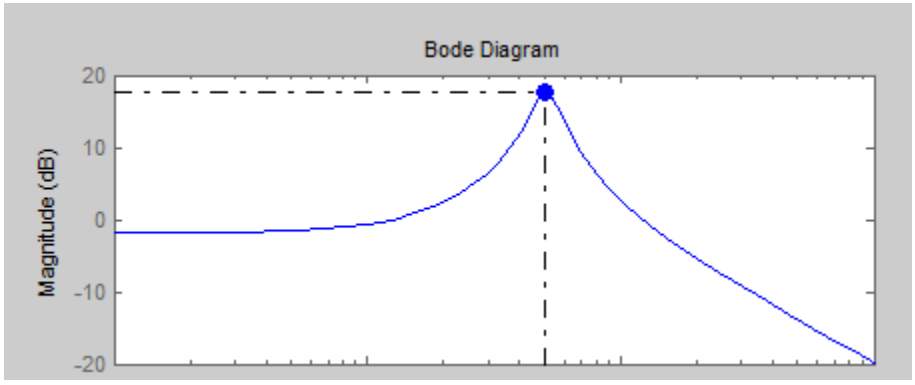


Display the peak response on the plot.

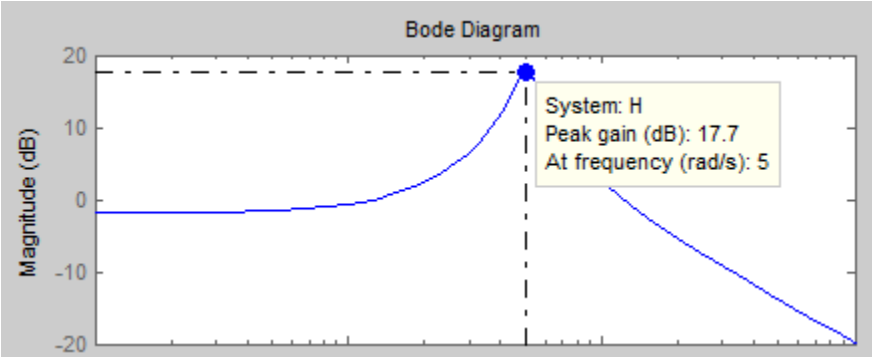
Right-click anywhere in the figure and select **Characteristics > Peak Response** from the menu.



A marker appears on the plot indicating the peak response. Horizontal and vertical dotted lines indicate the frequency and magnitude of that response. The other menu options add other system characteristics to the plot.



Click the marker to view the magnitude and frequency of the peak response in a datatip.



See Also

Related Examples

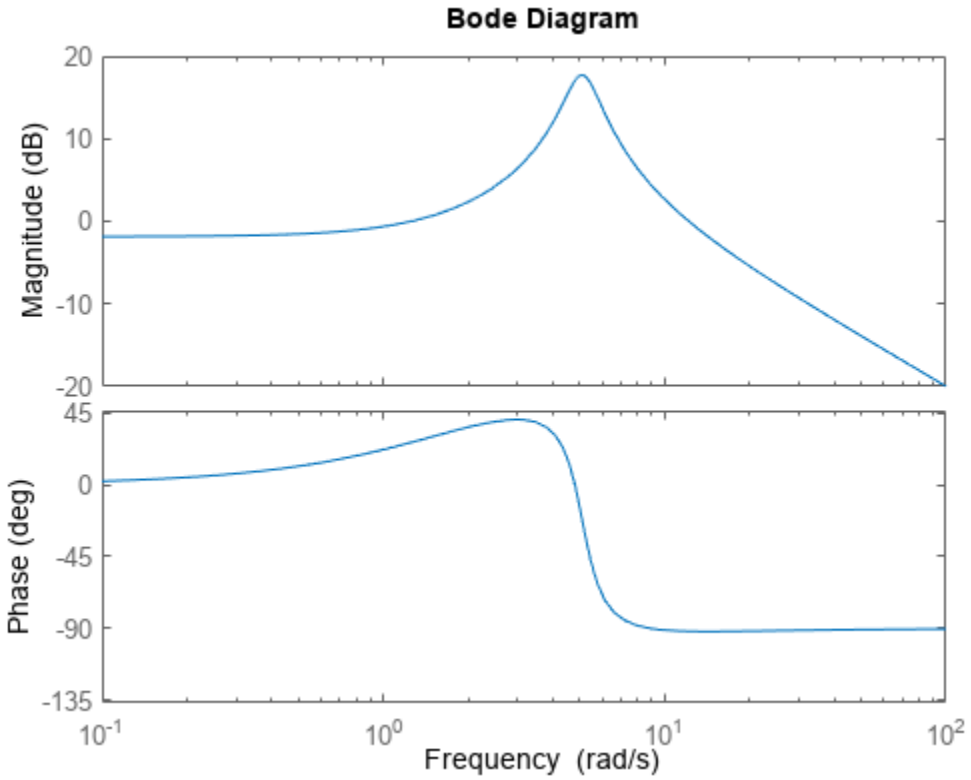
- “Numeric Values of Frequency-Domain Characteristics of SISO Model” on page 8-11
- “Joint Time-Domain and Frequency-Domain Analysis” on page 7-32
- “Pole and Zero Locations” on page 8-13

Numeric Values of Frequency-Domain Characteristics of SISO Model

This example shows how to obtain numeric values of several frequency-domain characteristics of a SISO dynamic system model, including the peak gain, dc gain, system bandwidth, and the frequencies at which the system gain crosses a specified frequency.

Create a transfer function model and plot its frequency response.

```
H = tf([10,21],[1,1.4,26]);
bodeplot(H)
```



Plotting the frequency response gives a rough idea of the frequency-domain characteristics of the system. `H` includes a pronounced resonant peak, and rolls off at 20 dB/decade at high frequency. It is often desirable to obtain specific numeric values for such characteristics.

Calculate the peak gain and the frequency of the resonance.

```
[gpeak,fpeak] = getPeakGain(H);
gpeak_dB = mag2db(gpeak)
```

```
gpeak_dB = 17.7596
```

`getPeakGain` returns both the peak location `fpeak` and the peak gain `gpeak` in absolute units. Using `mag2db` to convert `gpeak` to decibels shows that the gain peaks at almost 18 dB.

Find the band within which the system gain exceeds 0 dB, or 1 in absolute units.

```
wc = getGainCrossover(H,1)
```

```
wc = 2×1

    1.2582
   12.1843
```

`getGainCrossover` returns a vector of frequencies at which the system response crosses the specified gain. The resulting `wc` vector shows that the system gain exceeds 0 dB between about 1.3 and 12.2 rad/s.

Find the dc gain of `H`.

The Bode response plot shows that the gain of `H` tends toward a finite value as the frequency approaches zero. The `dcgain` command finds this value in absolute units.

```
k = dcgain(H);
```

Find the frequency at which the response of `H` rolls off to –10 dB relative to its dc value.

```
fb = bandwidth(H,-10);
```

`bandwidth` returns the first frequency at which the system response drops below the dc gain by the specified value in dB.

See Also

`getPeakGain` | `getGainCrossover` | `bandwidth`

Related Examples

- “Pole and Zero Locations” on page 8-13

More About

- “Frequency-Domain Responses” on page 8-2

Pole and Zero Locations

This example shows how to examine the pole and zero locations of dynamic systems both graphically using `pzplot` and numerically using `pole` and `zero`.

Examining the pole and zero locations can be useful for tasks such as stability analysis or identifying near-canceling pole-zero pairs for model simplification. This example compares two closed-loop systems that have the same plant and different controllers.

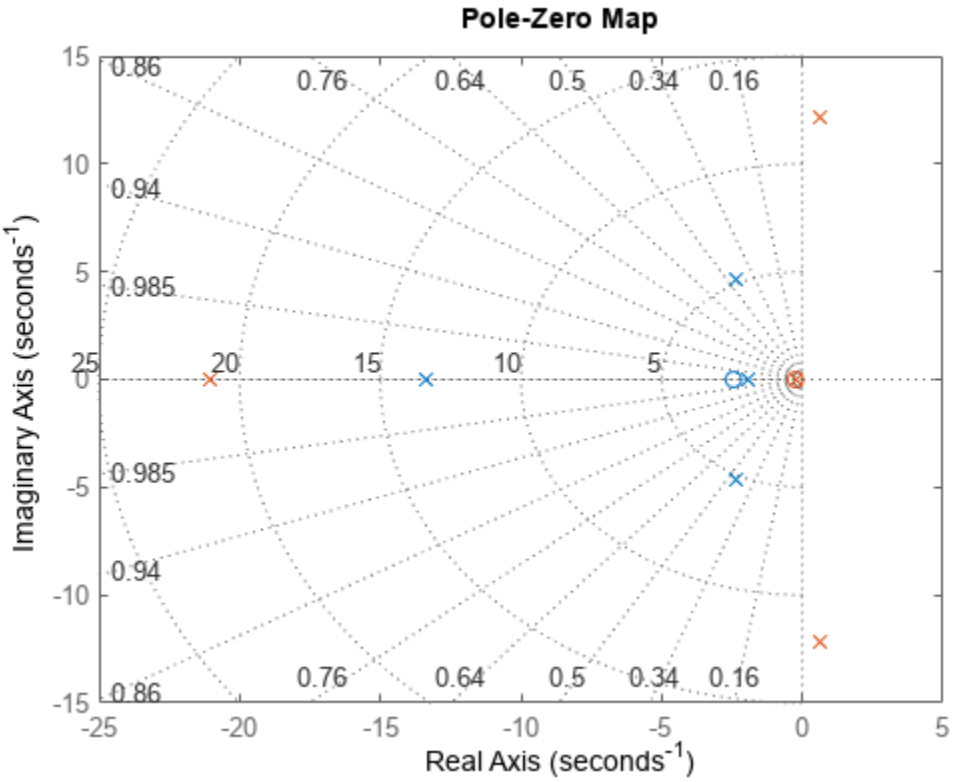
Create dynamic system models representing the two closed-loop systems.

```
G = zpk([],[-5 -5 -10],100);
C1 = pid(2.9,7.1);
CL1 = feedback(G*C1,1);
C2 = pid(29,7.1);
CL2 = feedback(G*C2,1);
```

The controller `C2` has a much higher proportional gain. Otherwise, the two closed-loop systems `CL1` and `CL2` are the same.

Graphically examine the pole and zero locations of `CL1` and `CL2`.

```
pzplot(CL1,CL2)
grid
```



`pzplot` plots pole and zero locations on the complex plane as `x` and `o` marks, respectively. When you provide multiple models, `pzplot` plots the poles and zeros of each model in a different color. Here, there poles and zeros of `CL1` are blue, and those of `CL2` are green.

The plot shows that all poles of `CL1` are in the left half-plane, and therefore `CL1` is stable. From the radial grid markings on the plot, you can read that the damping of the oscillating (complex) poles is approximately 0.45. The plot also shows that `CL2` contains poles in the right half-plane and is therefore unstable.

Compute numerical values of the pole and zero locations of `CL2`.

```
z = zero(CL2);
p = pole(CL2);
```

`zero` and `pole` return column vectors containing the zero and pole locations of the system.

See Also

`zero` | `pole` | `pzplot`

Related Examples

- “Numeric Values of Frequency-Domain Characteristics of SISO Model” on page 8-11

More About

- “Frequency-Domain Responses” on page 8-2

Assessing Gain and Phase Margins

This example shows how to examine the effect of stability margins on closed-loop response characteristics of a control system.

Stability of a Feedback Loop

Stability generally means that all internal signals remain bounded. This is a standard requirement for control systems to avoid loss of control and damage to equipment. For linear feedback systems, stability can be assessed by looking at the poles of the closed-loop transfer function. Consider for example the SISO feedback loop:

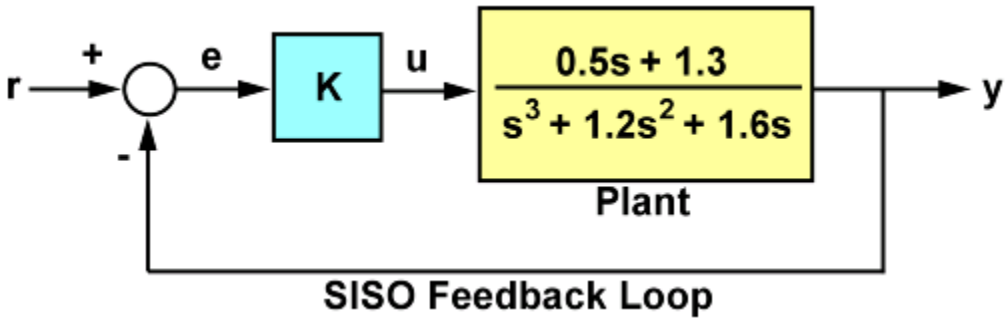


Figure 1: SISO Feedback Loop.

For a unit loop gain `k`, you can compute the closed-loop transfer function `T` using:

```
G = tf([.5 1.3],[1 1.2  1.6 0]);
T = feedback(G,1);
```

To obtain the poles of `T`, type

```
pole(T)
```

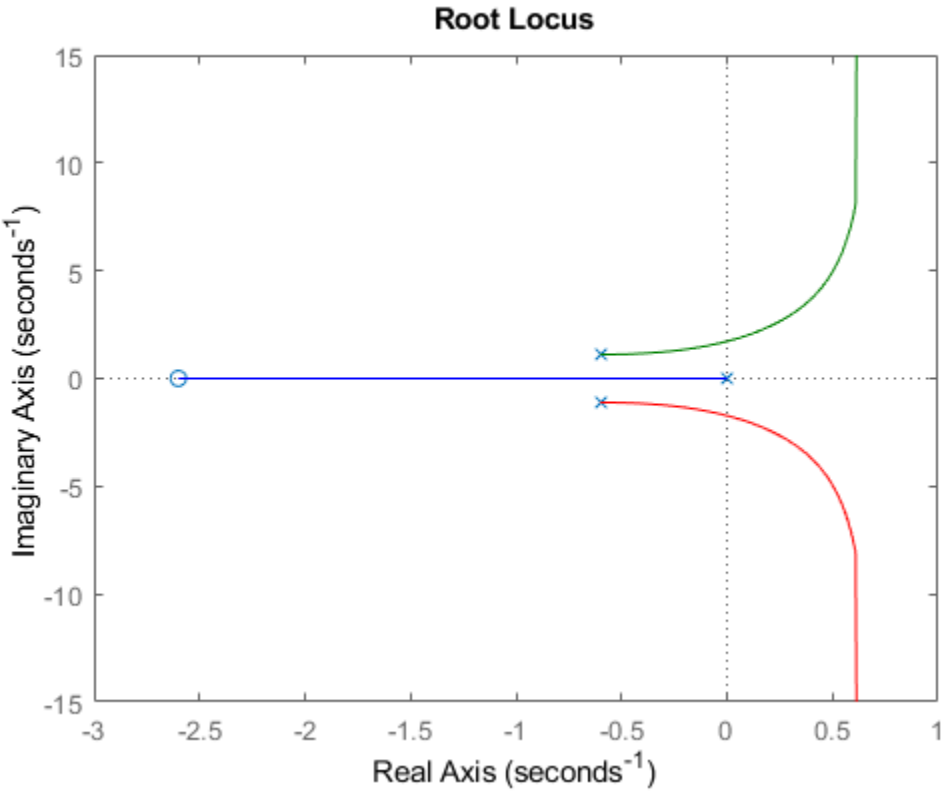
```
ans =

  -0.2305 + 1.3062i
  -0.2305 - 1.3062i
  -0.7389 + 0.0000i
```

The feedback loop for `k=1` is stable since all poles have negative real parts.

How Stable is Stable?

Checking the closed-loop poles gives us a binary assessment of stability. In practice, it is more useful to know how robust (or fragile) stability is. One indication of robustness is how much the loop gain can change before stability is lost. You can use the root locus plot to estimate the range of `k` values for which the loop is stable:

```
rlocus(G)
```



Clicking on the point where the locus intersects the y axis reveals that this feedback loop is stable for

$$0 < k < 2.7$$

This range shows that with k=1, the loop gain can increase 270% before you lose stability.

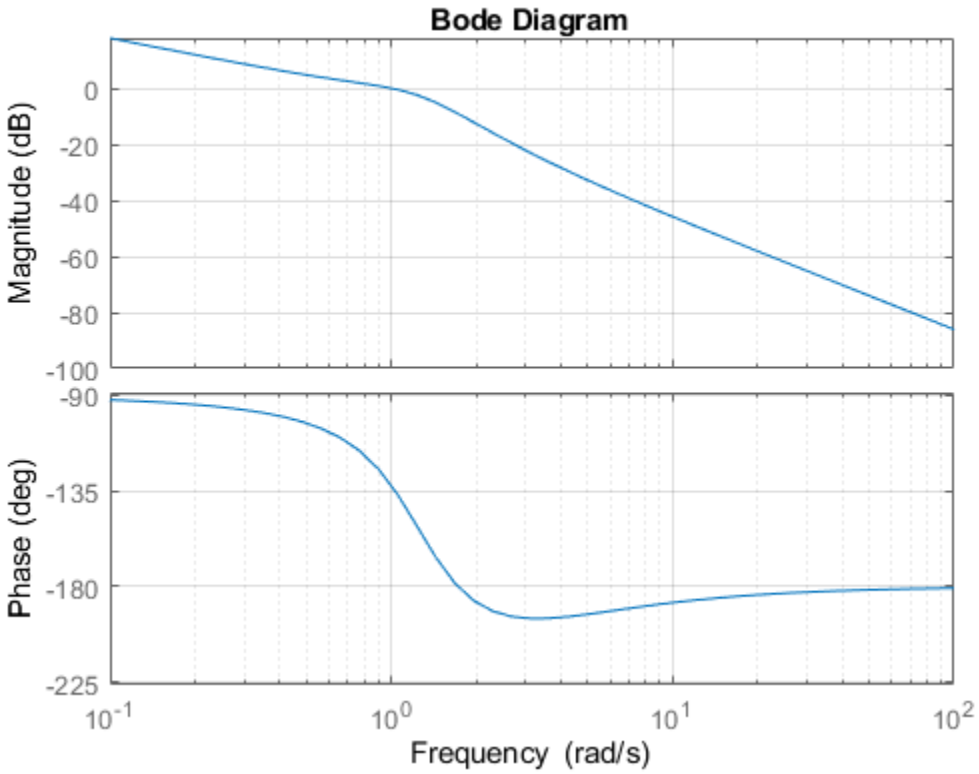
Gain and Phase Margins

Changes in the loop gain are only one aspect of robust stability. In general, imperfect plant modeling means that both gain and phase are not known exactly. Because modeling errors are most damaging near the gain crossover frequency (frequency where open-loop gain is 0dB), it also matters how much phase variation can be tolerated at this frequency.

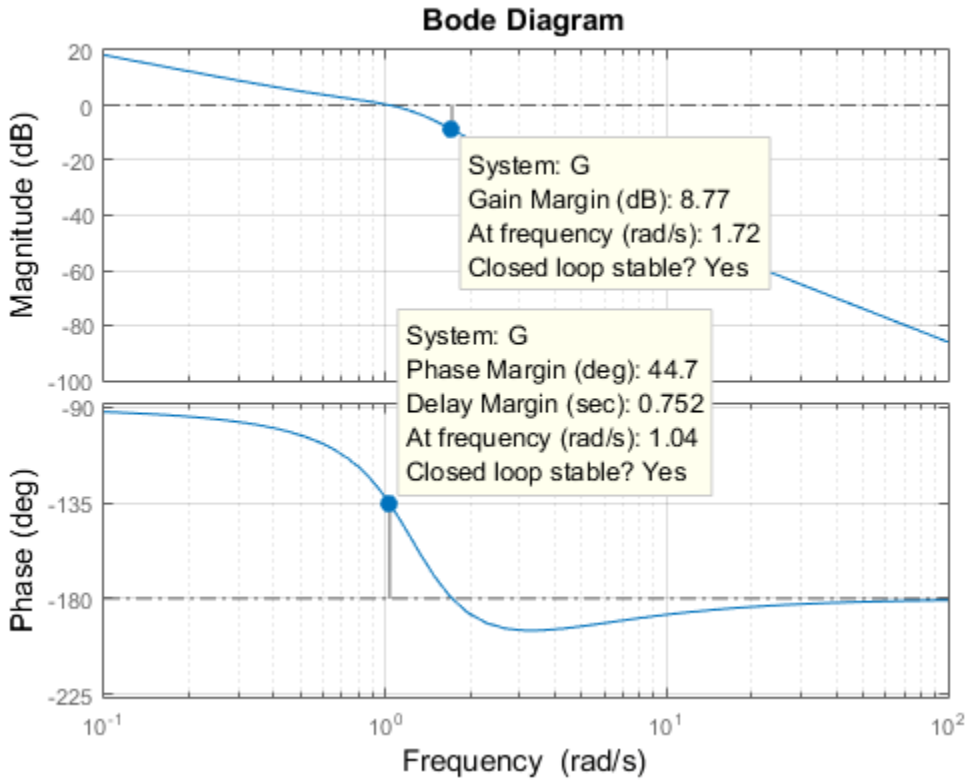
The phase margin measures how much phase variation is needed at the gain crossover frequency to lose stability. Similarly, the gain margin measures what relative gain variation is needed at the gain crossover frequency to lose stability. Together, these two numbers give an estimate of the "safety margin" for closed-loop stability. The smaller the stability margins, the more fragile stability is.

You can display the gain and phase margins on a Bode plot as follows. First create the plot:

```
bode(G), grid
```

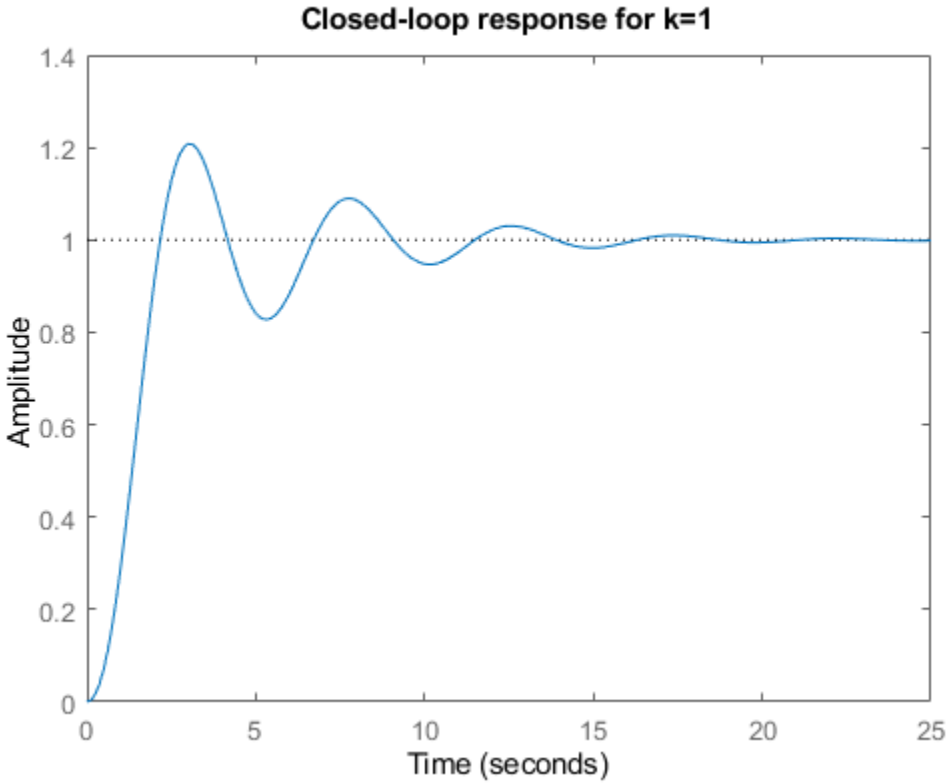


Then, right-click on the plot and select the **Characteristics -> Minimum Stability Margins** submenu. Finally, click on the blue dot markers. The resulting plot is shown below:



This indicates a gain margin of about 9 dB and a phase margin of about 45 degrees. The corresponding closed-loop step response exhibits about 20% overshoot and some oscillations.

```
step(T), title('Closed-loop response for k=1')
```



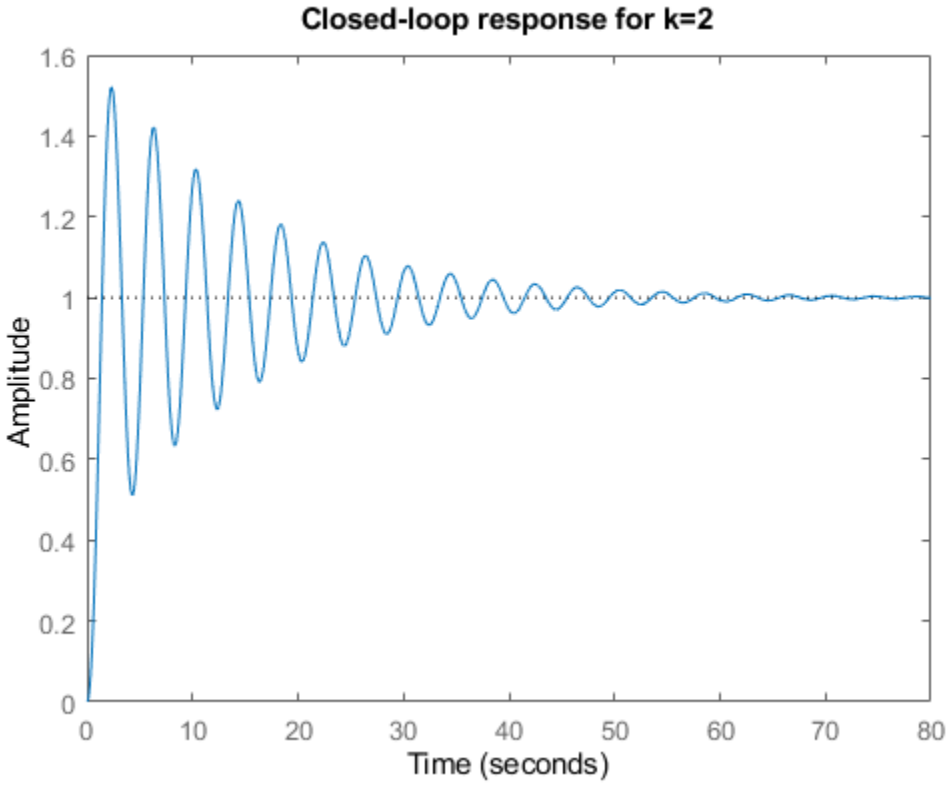
If we increase the gain to k=2, the stability margins are reduced to

```
[Gm,Pm] = margin(2*G);
GmdB = 20*log10(Gm)   % gain margin in dB
Pm  % phase margin in degrees
```

```
GmdB =

    2.7435

Pm =

    8.6328
```

and the closed-loop response has poorly damped oscillations, a sign of near instability.

```
step(feedback(2*G,1)), title('Closed-loop response for k=2')
```



Systems with Multiple Gain or Phase Crossings

Some systems have multiple gain crossover or phase crossover frequencies, which leads to multiple gain or phase margin values. For example, consider the feedback loop

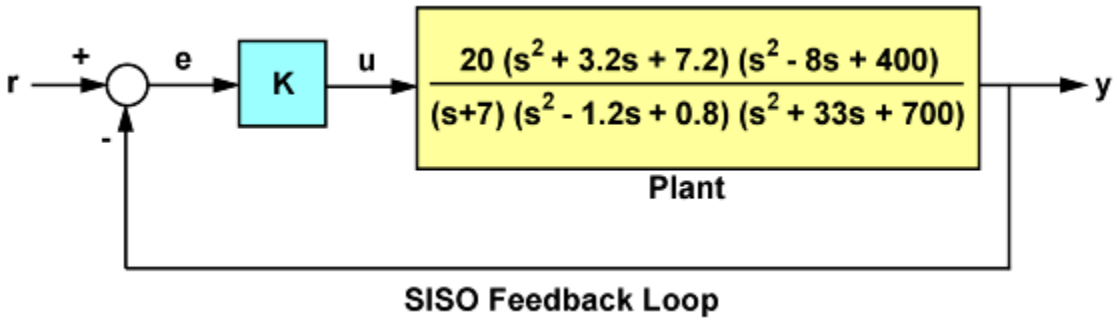
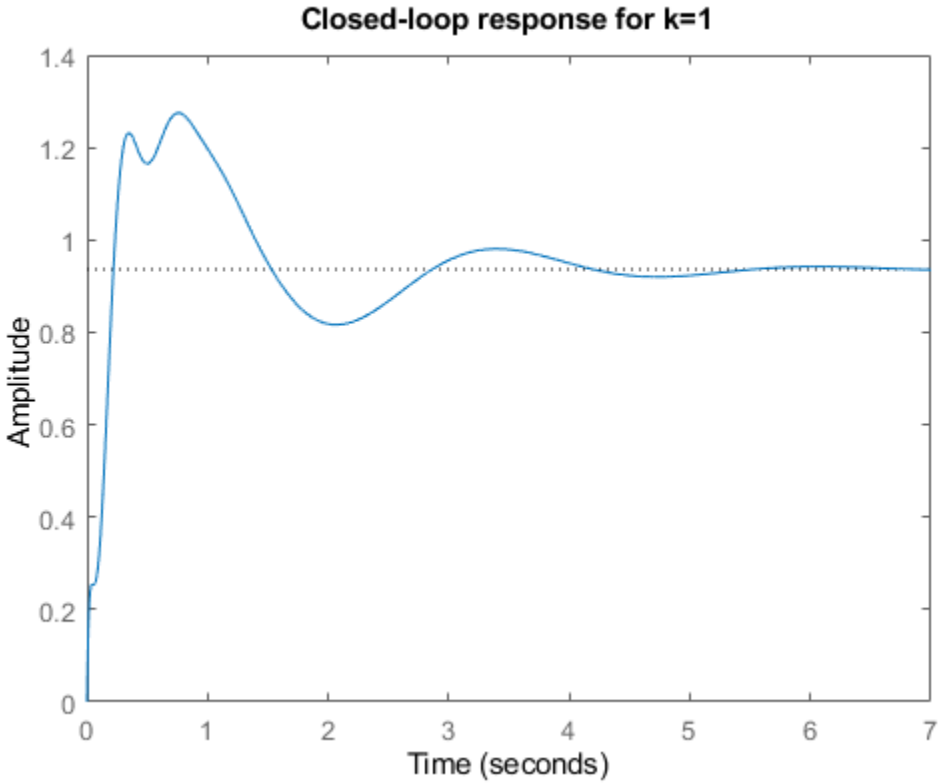


Figure 2: Feedback Loop with Multiple Phase Crossovers

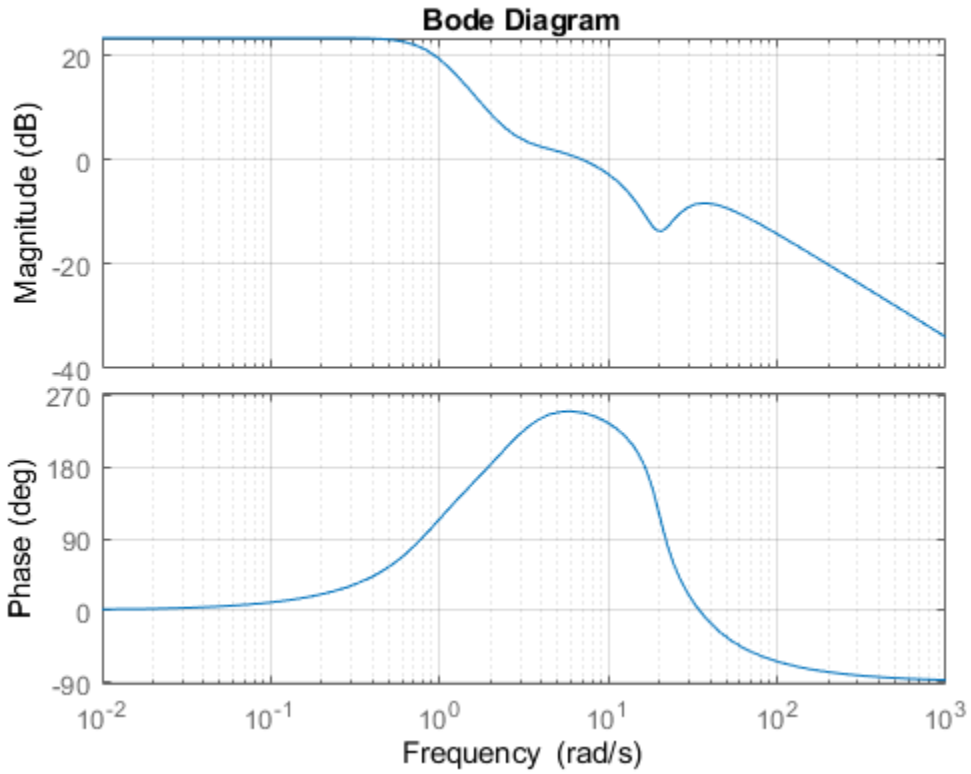
The closed-loop response for `k=1` is stable:

```
G = tf(20,[1 7]) * tf([1 3.2 7.2],[1 -1.2 0.8]) * tf([1 -8 400],[1 33 700]);
T = feedback(G,1);
step(T), title('Closed-loop response for k=1')
```

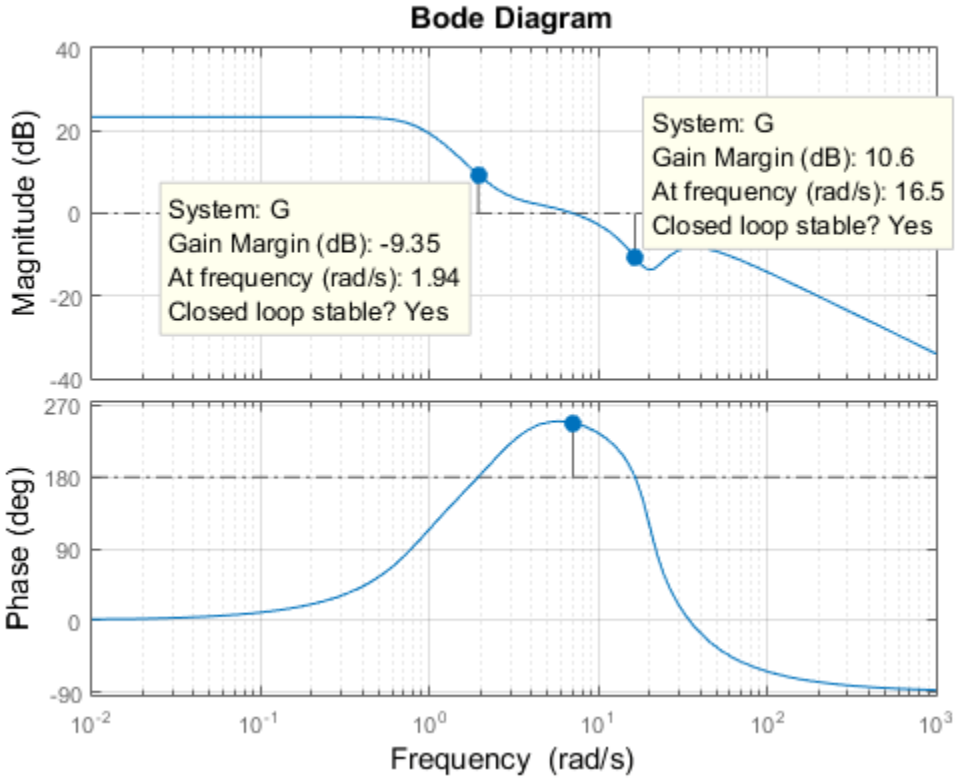


To assess how robustly stable this loop is, plot its Bode response:

```
bode(G), grid
```

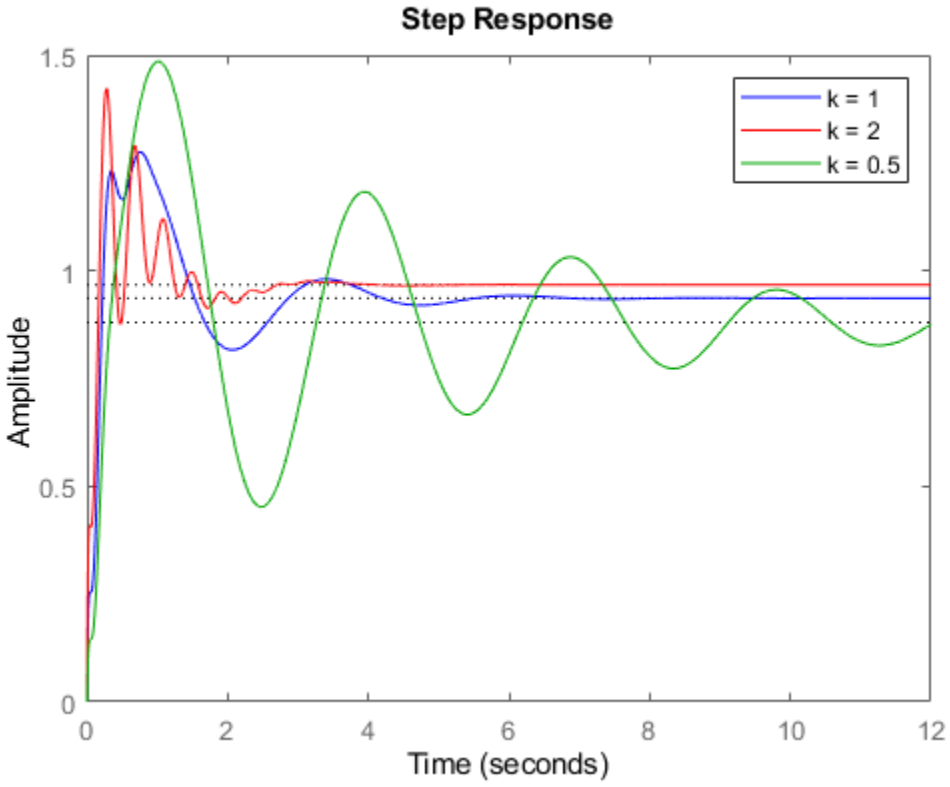


Then, right-click on the plot and select the **Characteristics -> All Stability Margins** submenu to show all the crossover frequencies and associated stability margins. The resulting plot is shown below.



Note that there are two 180 deg phase crossings with corresponding gain margins of -9.35dB and +10.6dB. Negative gain margins indicate that stability is lost by decreasing the gain, while positive gain margins indicate that stability is lost by increasing the gain. This is confirmed by plotting the closed-loop step response for a plus/minus 6dB gain variation about k=1:

```
k1 = 2;     T1 = feedback(G*k1,1);
k2 = 1/2;   T2 = feedback(G*k2,1);
step(T,'b',T1,'r',T2,'g',12),
legend('k = 1','k = 2','k = 0.5')
```



The plot shows increased oscillations for both smaller and larger gain values.

You can use the command `allmargin` to compute all stability margins. Note that gain margins are expressed as gain ratios, not dB. Use `mag2db` to convert the values to dB.

```
m = allmargin(G)

GainMargins_dB = mag2db(m.GainMargin)
```

```
m =

  struct with fields:

     GainMargin: [0.3408 3.3920]
    GMFrequency: [1.9421 16.4807]
    PhaseMargin: 68.1140
    PMFrequency: 7.0776
    DelayMargin: 0.1680
    DMFrequency: 7.0776
         Stable: 1


GainMargins_dB =
```

```
   -9.3510   10.6091
```

See Also

`margin` | `pole` | `diskmargin`

Related Examples

- “Pole and Zero Locations” on page 8-13

Analyzing Control Systems with Delays

This example shows how to use Control System Toolbox™ to analyze and design control systems with delays.

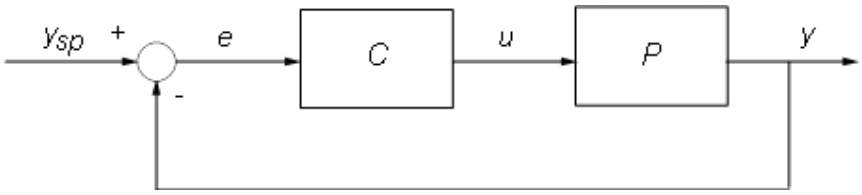
Control of Processes with Delays

Many processes involve dead times, also referred to as transport delays or time lags. Controlling such processes is challenging because delays cause linear phase shifts that limit the control bandwidth and affect closed-loop stability.

Using the state-space representation, you can create accurate open- or closed-loop models of control systems with delays and analyze their stability and performance without approximation. The state-space (SS) object automatically keeps track of "internal" delays when combining models, see the "Specifying Time Delays" tutorial for more details.

Example: PI Control Loop with Dead Time

Consider the standard setpoint tracking loop:



where the process model P has a 2.6 second dead time and the compensator C is a PI controller:

$$P(s) = \frac{e^{-2.6s}(s+3)}{s^2+0.3s+1}, \quad C(s) = 0.06(1+\frac{1}{s})$$

You can specify these two transfer functions as

```
s = tf('s');
P = exp(-2.6*s)*(s+3)/(s^2+0.3*s+1);
C = 0.06 * (1 + 1/s);
```

To analyze the closed-loop response, construct a model T of the closed-loop transfer from ysp to y. Because there is a delay in this feedback loop, you must convert P and C to state space and use the state-space representation for analysis:

```
T = feedback(P*C,1)
```

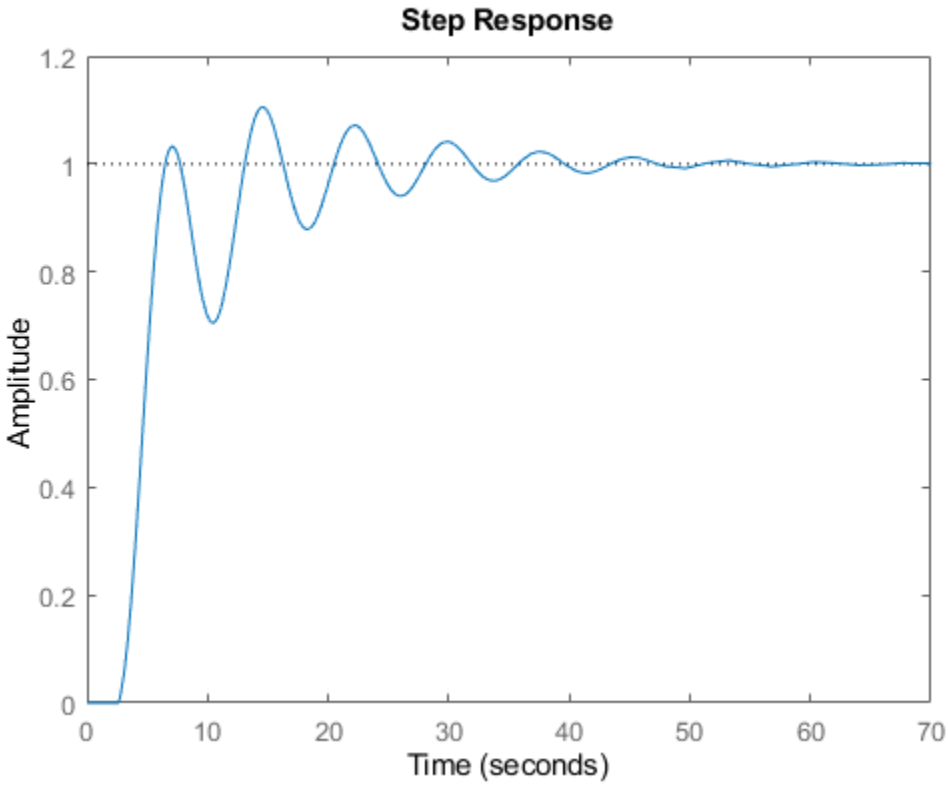
```
T =

  A =
         x1     x2     x3
   x1  -0.36  -1.24  -0.18
   x2      1      0      0
   x3      0      1      0

  B =
```

```
      u1
 x1  0.5
 x2    0
 x3    0

C =
       x1    x2    x3
 y1  0.12  0.48  0.36

D =
     u1
 y1   0

 (values computed with all internal delays set to zero)

 Internal delays (seconds): 2.6

Continuous-time state-space model.
```

The result is a third-order model with an internal delay of 2.6 seconds. Internally, the state-space object `T` tracks how the delay is coupled with the remaining dynamics. This structural information is not visible to users, and the display above only gives the A,B,C,D values when the delay is set to zero.

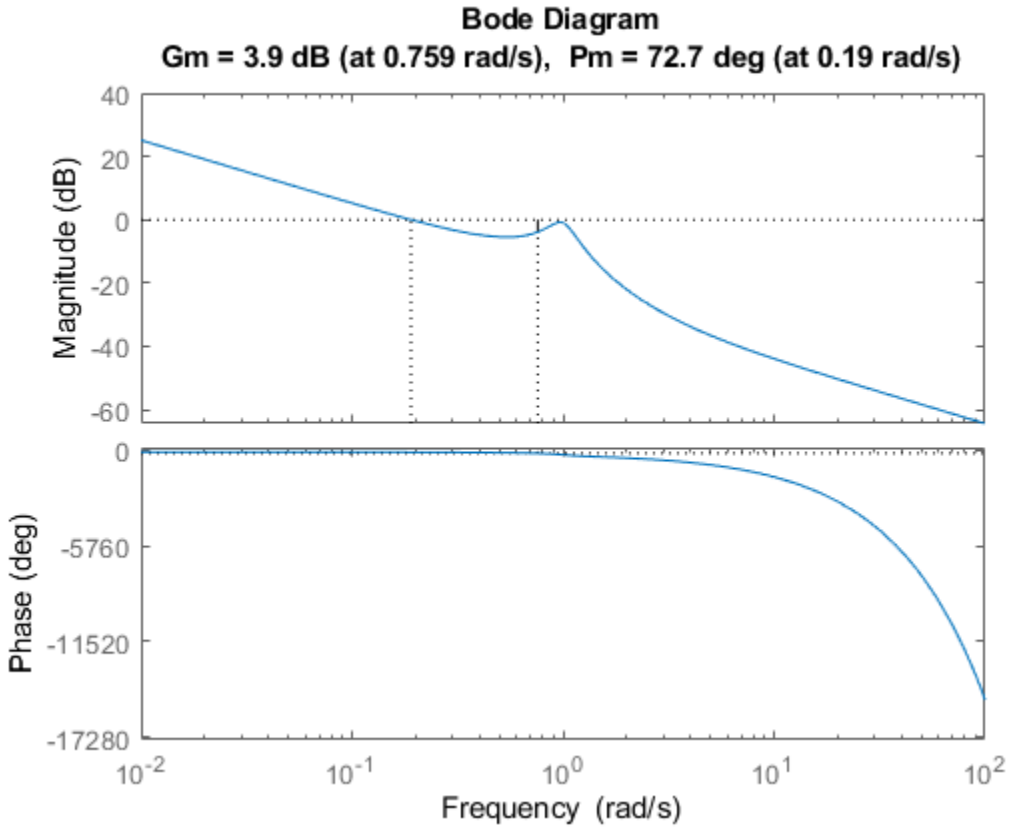
Use the `STEP` command to plot the closed-loop step response from `ysp` to `y`:

```
step(T)
```



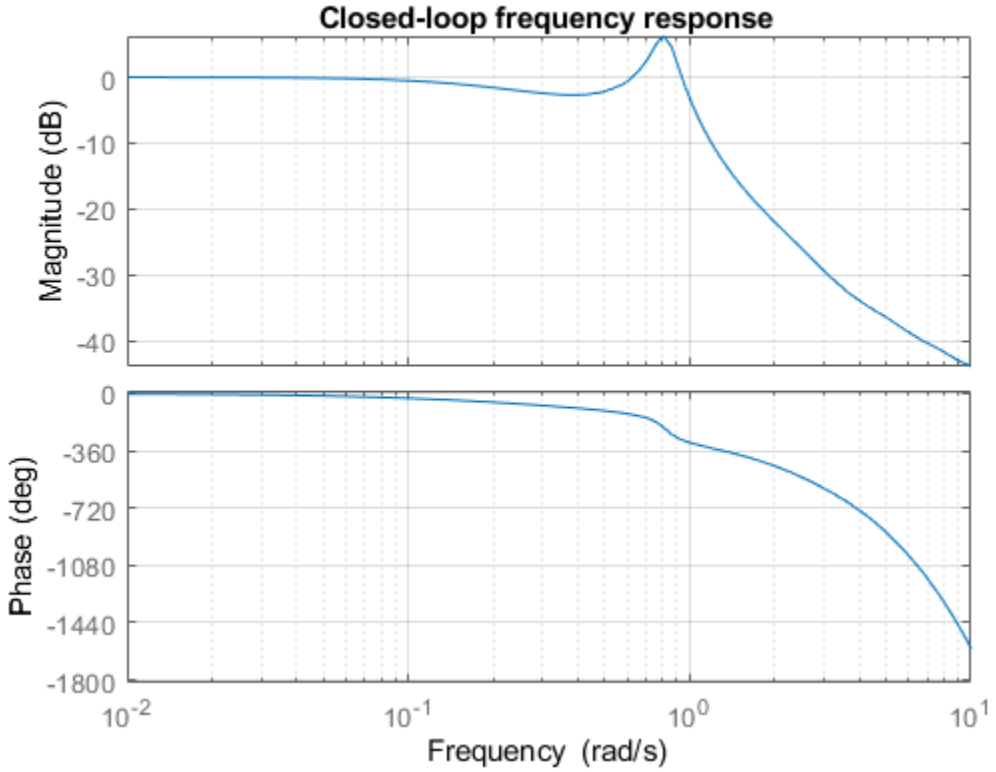
The closed-loop oscillations are due to a weak gain margin as seen from the open-loop response `P*C`:

```
margin(P*C)
```



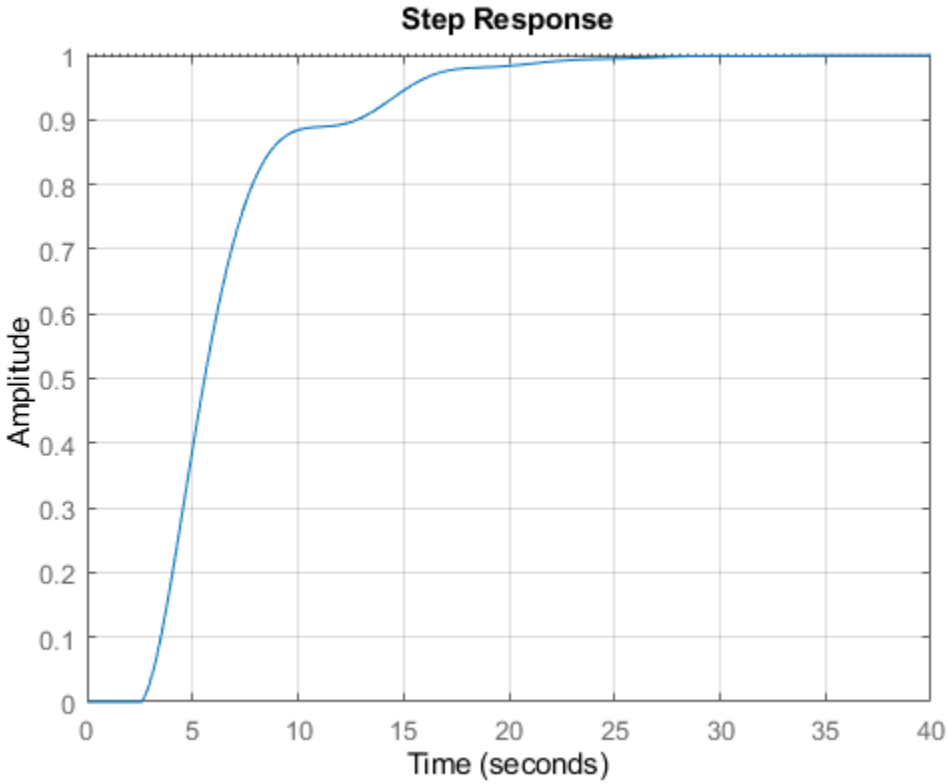
There is also a resonance in the closed-loop frequency response:

```
bode(T)
grid, title('Closed-loop frequency response')
```



To improve the design, you can try to notch out the resonance near 1 rad/s:

```
notch = tf([1 0.2 1],[1 .8 1]);
C = 0.05 * (1 + 1/s);
Tnotch = feedback(P*C*notch,1);

step(Tnotch), grid
```

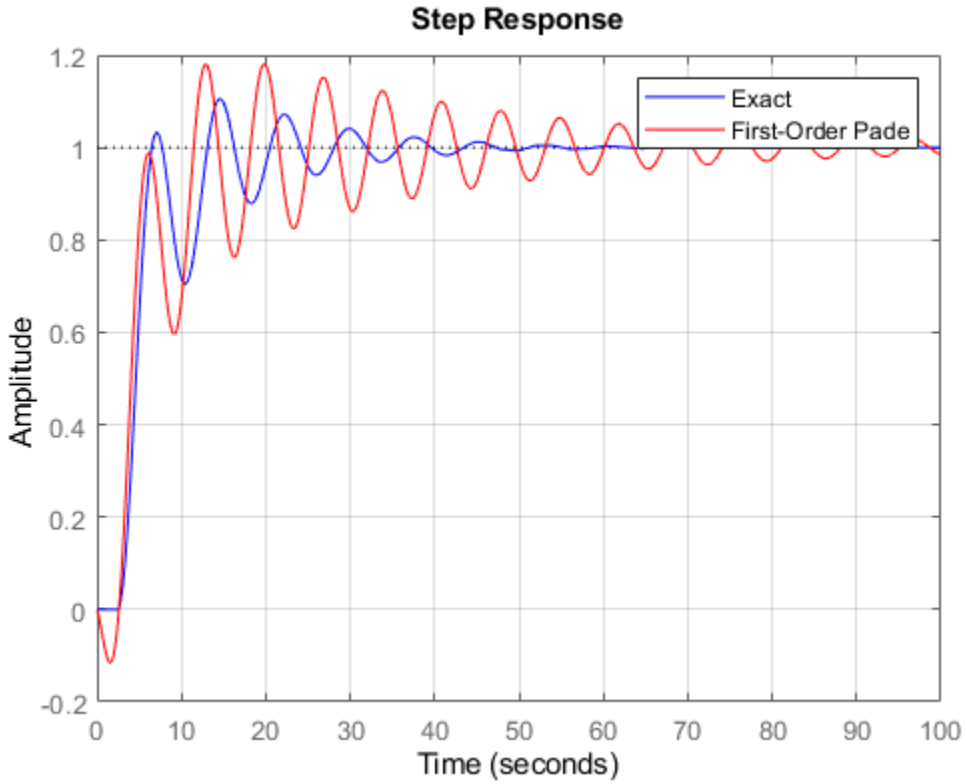


Pade Approximation of Time Delays

Many control design algorithms cannot handle time delays directly. A common workaround consists of replacing delays by their Pade approximations (all-pass filters). Because this approximation is only valid at low frequencies, it is important to compare the true and approximate responses to choose the right approximation order and check the approximation validity.

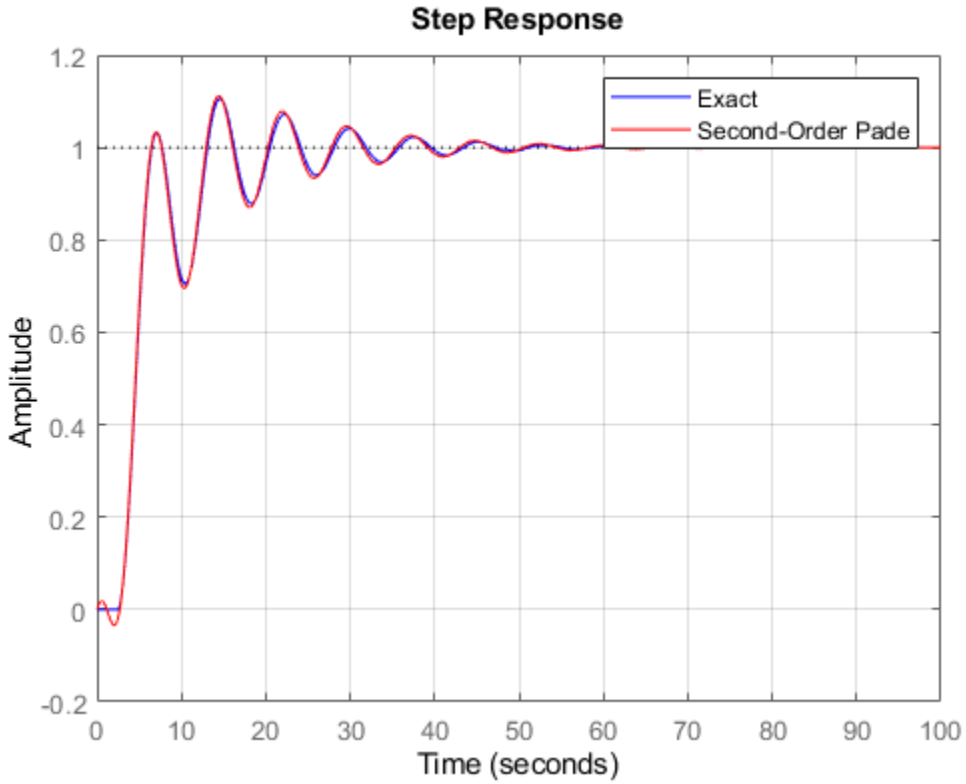
Use the PADE command to compute Pade approximations of LTI models with delays. For the PI control example above, you can compare the exact closed-loop response T with the response obtained for a first-order Pade approximation of the delay:

```
T1 = pade(T,1);
step(T,'b',T1,'r',100)
grid, legend('Exact','First-Order Pade')
```



The approximation error is fairly large. To get a better approximation, try a second-order Pade approximation of the delay:

```
T2 = pade(T,2);
step(T,'b',T2,'r',100)
grid, legend('Exact','Second-Order Pade')
```



The responses now match closely except for the non-minimum phase artifact introduced by the Pade approximation.

Sensitivity Analysis

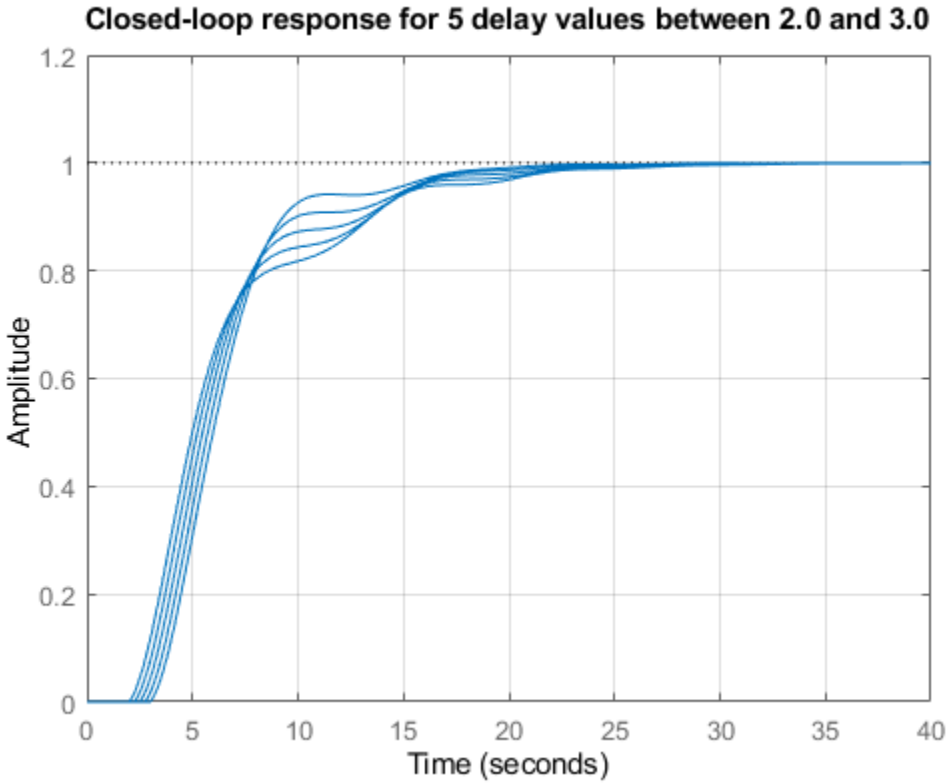
Delays are rarely known accurately, so it is often important to understand how sensitive a control system is to the delay value. Such sensitivity analysis is easily performed using LTI arrays and the InternalDelay property.

For example, to analyze the sensitivity of the notched PI control above, create 5 models with delay values ranging from 2.0 to 3.0:

```
tau = linspace(2,3,5);                  % 5 delay values
Tsens = repsys(Tnotch,[1 1 5]);         % 5 copies of Tnotch
for j=1:5
  Tsens(:,:,j).InternalDelay = tau(j);  % jth delay value -> jth model
end
```

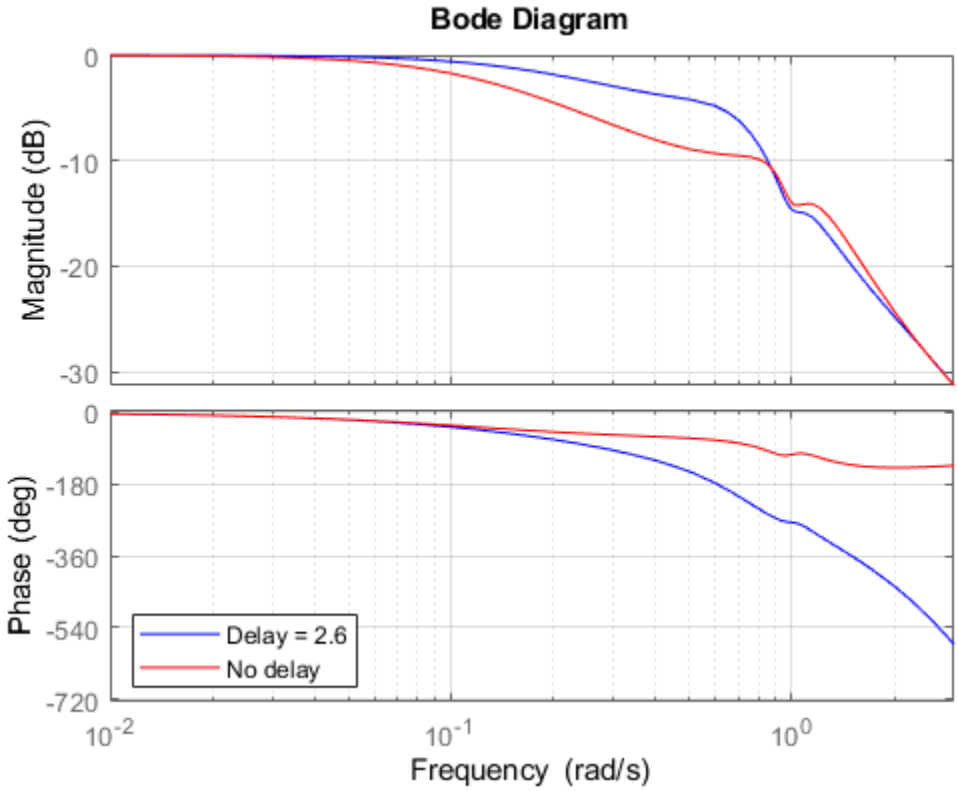
Then use STEP to create an envelope plot:

```
step(Tsens)
grid, title('Closed-loop response for 5 delay values between 2.0 and 3.0')
```



This plot shows that uncertainty on the delay value has little effect on closed-loop characteristics. Note that while you can change the values of internal delays, you cannot change how many there are because this is part of the model structure. To eliminate some internal delays, set their value to zero or use PADE with order zero:

```
Tnotch0 = Tnotch;
Tnotch0.InternalDelay = 0;
bode(Tnotch,'b',Tnotch0,'r',{1e-2,3})
grid, legend('Delay = 2.6','No delay','Location','SouthWest')
```

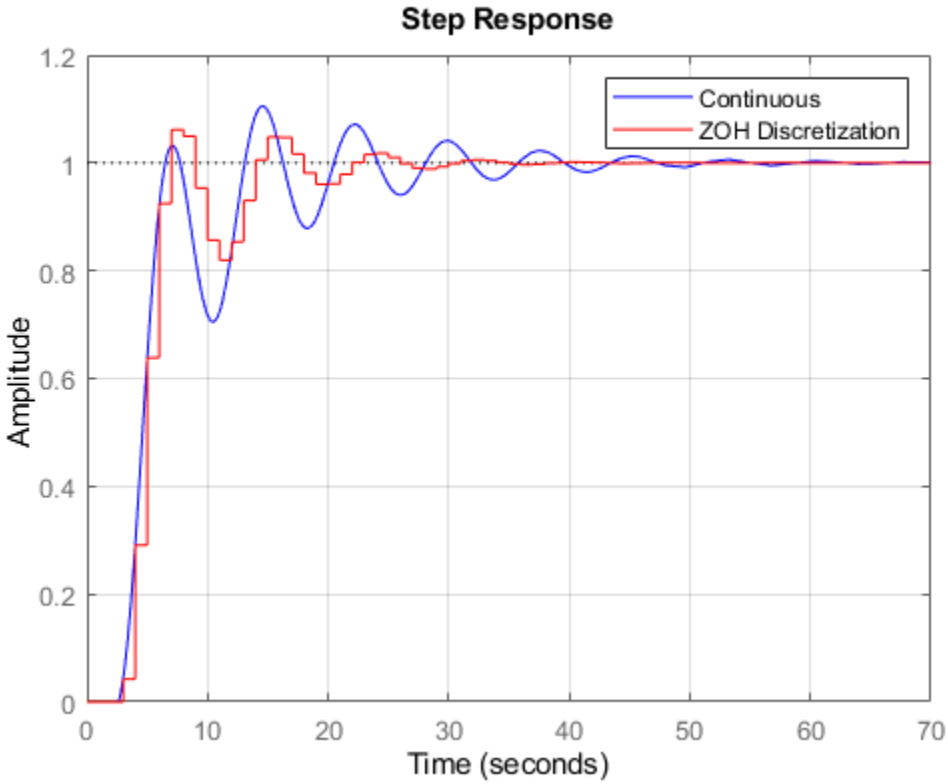


Discretization

You can use `C2D` to discretize continuous-time delay systems. Available methods include zero-order hold (ZOH), first-order hold (FOH), and Tustin. For models with internal delays, the ZOH discretization is not always "exact," i.e., the continuous and discretized step responses may not match:

```
Td = c2d(T,1);
step(T,'b',Td,'r')
grid, legend('Continuous','ZOH Discretization')
```

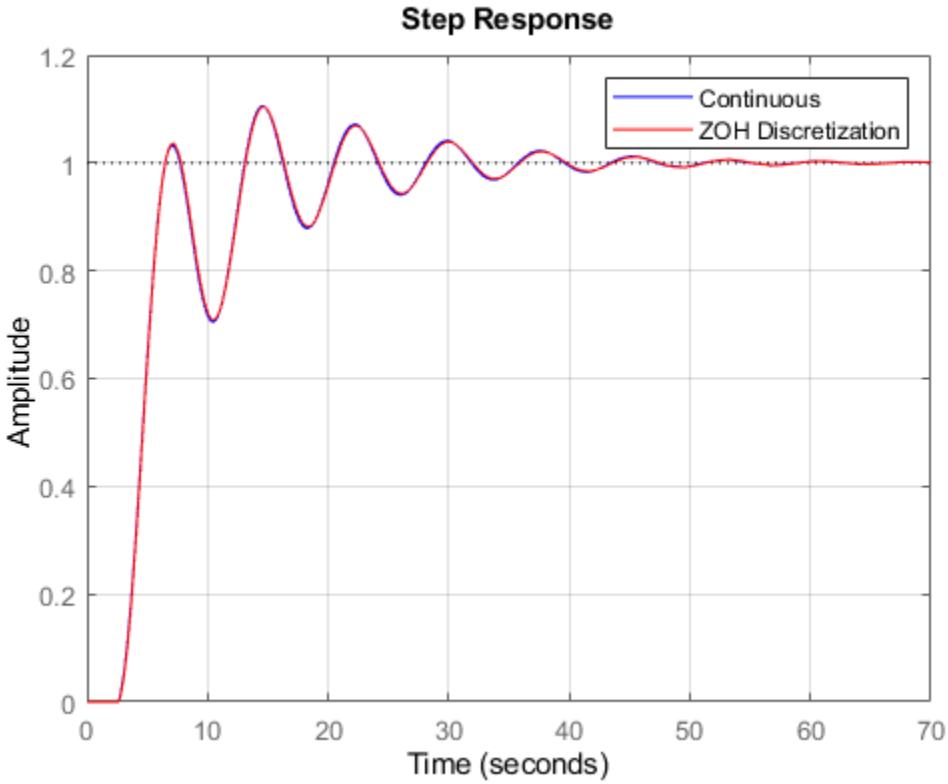
```
Warning: Discretization is only approximate due to internal delays. Use faster
sampling rate if discretization error is large.
```



To correct such discretization gaps, reduce the sampling period until the continuous and discrete responses match closely:

```
Td = c2d(T,0.05);
step(T,'b',Td,'r')
grid, legend('Continuous','ZOH Discretization')
```

```
Warning: Discretization is only approximate due to internal delays. Use faster
sampling rate if discretization error is large.
```



Note that internal delays remain internal in the discretized model and do not inflate the model order:

```
order(Td)
Td.InternalDelay
```

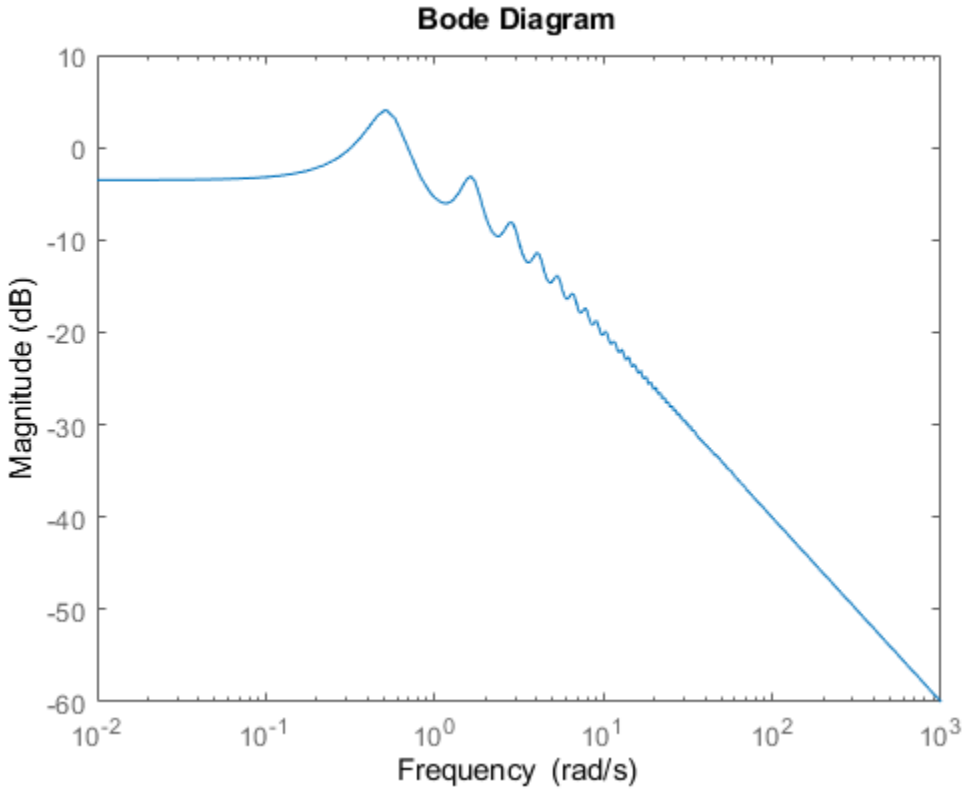
```
ans =

     3


ans =

    52
```

Some Unique Features of Delay Systems

The time and frequency responses of delay systems can look bizarre and suspicious to those only familiar with delay-free LTI analysis. Time responses can behave chaotically, Bode plots can exhibit gain oscillations, etc. These are not software quirks but real features of such systems. Below are a few illustrations of these phenomena

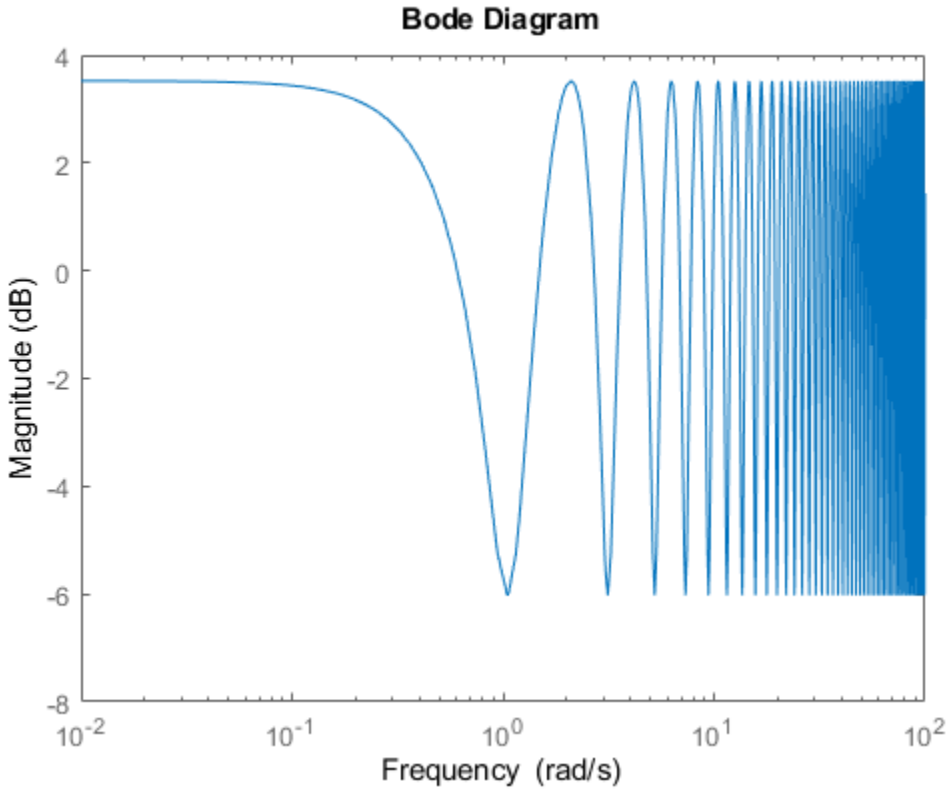
Gain ripples:

```
G = exp(-5*s)/(s+1);
T = feedback(G,.5);
bodemag(T)
```



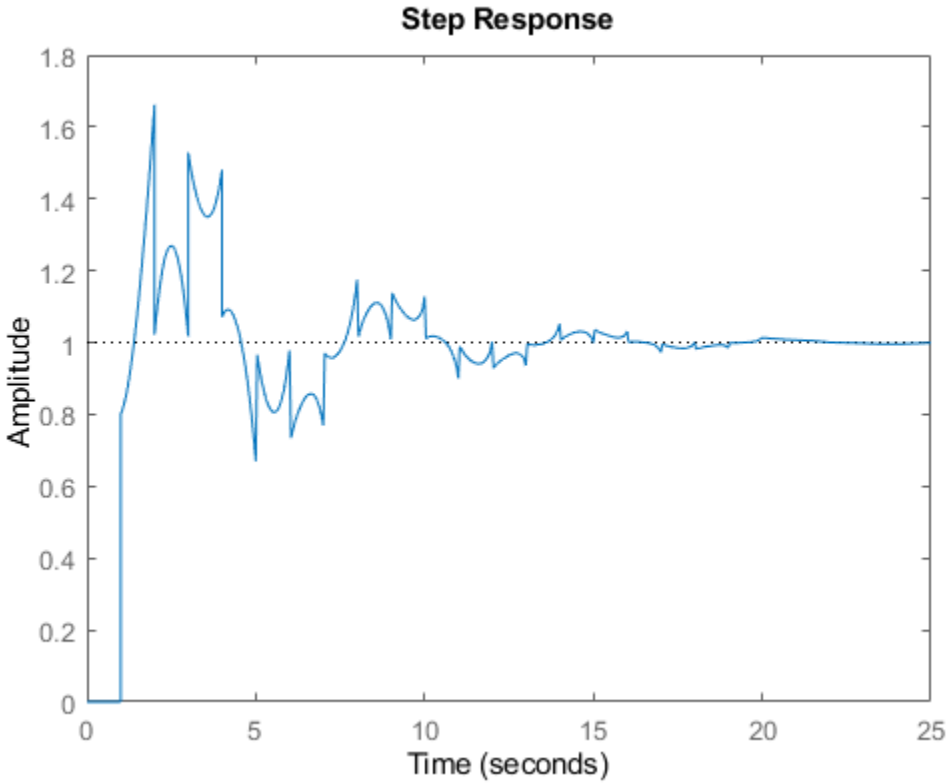
Gain oscillations:

```
G = 1 + 0.5 * exp(-3*s);
bodemag(G)
```



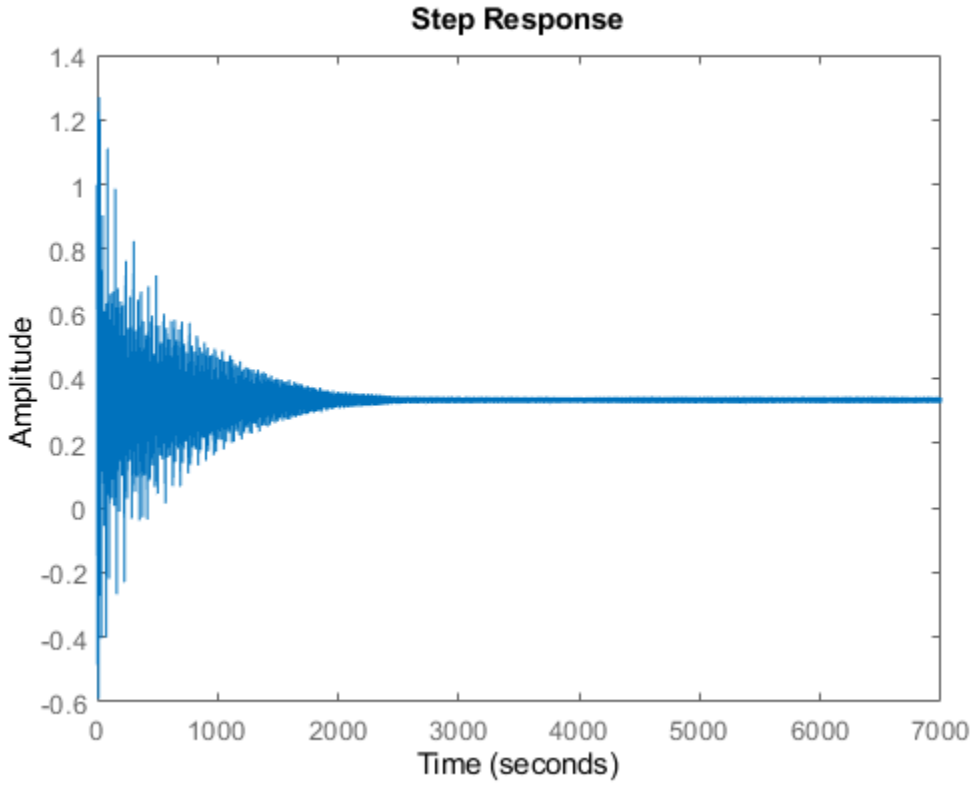
Jagged step response (note the "echoes" of the initial step):

```
G = exp(-s) * (0.8*s^2+s+2)/(s^2+s);
T = feedback(G,1);
step(T)
```



Chaotic response:

```
G = 1/(s+1) + exp(-4*s);
T = feedback(1,G);

step(T)
```



See Also

`pade` | `margin`

Related Examples

- “Analyzing the Response of an RLC Circuit” on page 8-41

More About

- “Time Delays in Linear Systems” on page 2-31
- “Time-Delay Approximation” on page 2-37

Analyzing the Response of an RLC Circuit

This example shows how to analyze the time and frequency responses of common RLC circuits as a function of their physical parameters using Control System Toolbox™ functions.

Bandpass RLC Network

The following figure shows the parallel form of a bandpass RLC circuit:

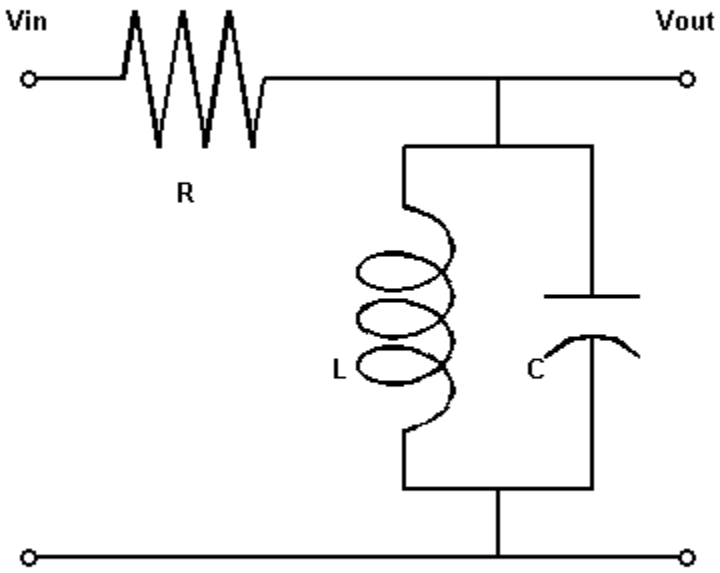


Figure 1: Bandpass RLC Network.

The transfer function from input to output voltage is:

$$G(s) = \frac{s/(RC)}{s^2 + s/(RC) + 1/(LC)}$$

The product `LC` controls the bandpass frequency while `RC` controls how narrow the passing band is. To build a bandpass filter tuned to the frequency 1 rad/s, set `L=C=1` and use `R` to tune the filter band.

Analyzing the Frequency Response of the Circuit

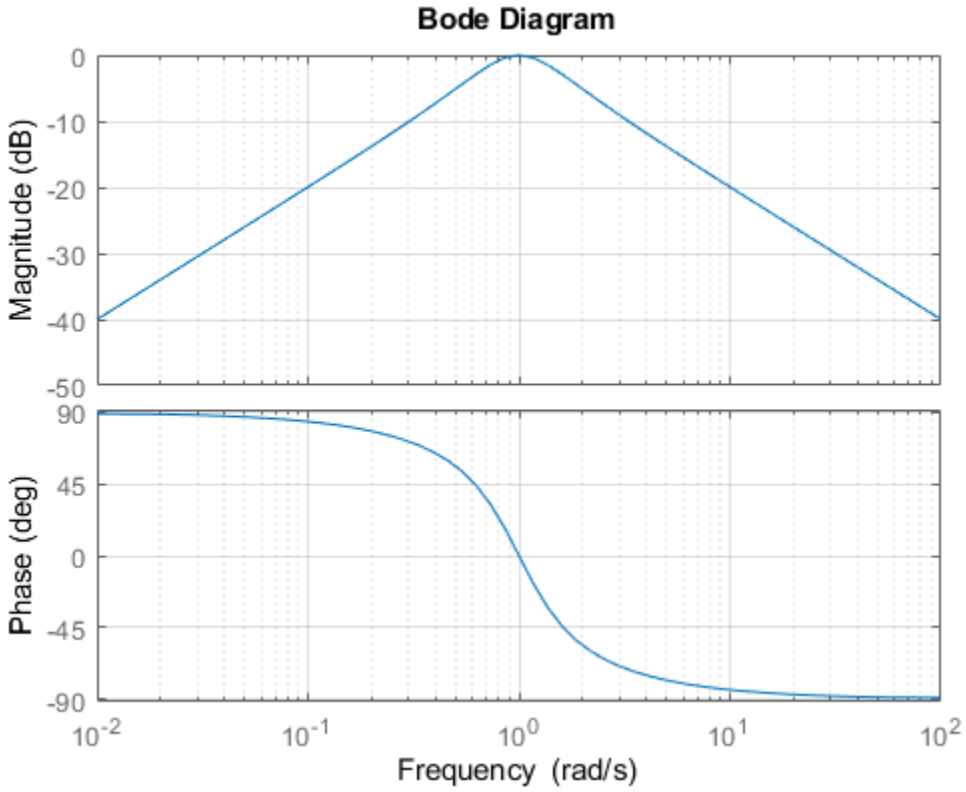
The Bode plot is a convenient tool for investigating the bandpass characteristics of the RLC network. Use `tf` to specify the circuit's transfer function for the values

```
%|R=L=C=1|:
R = 1; L = 1; C = 1;
G = tf([1/(R*C) 0],[1 1/(R*C) 1/(L*C)])
```

```
G =

       s
  -----------
  s^2 + s + 1

Continuous-time transfer function.
```

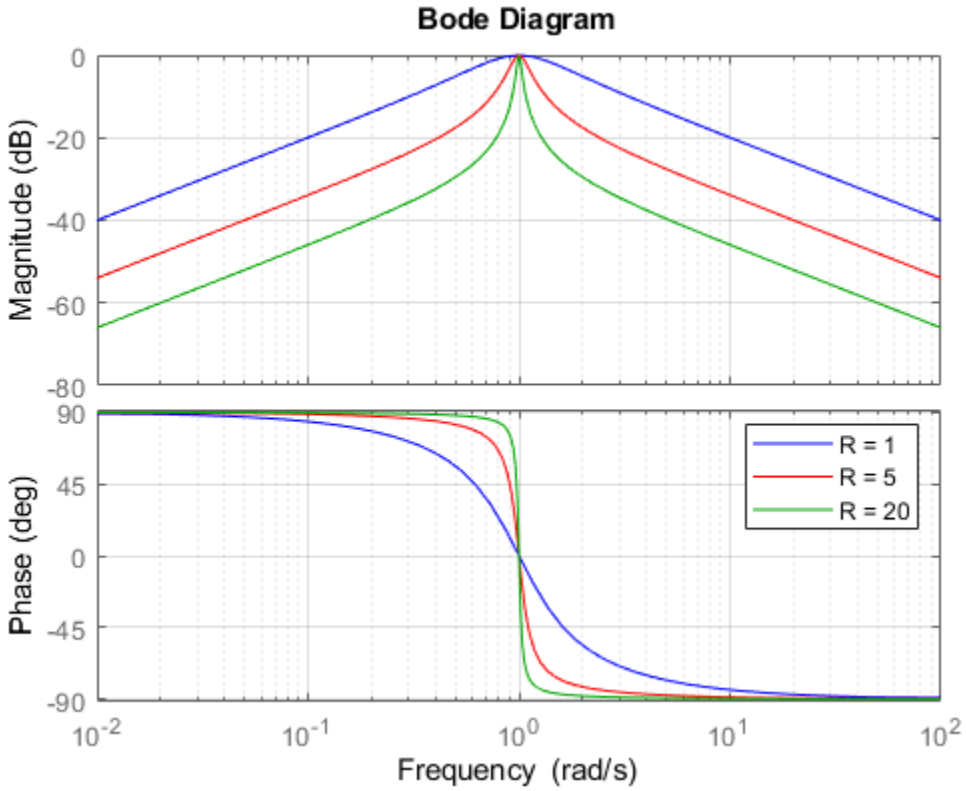
Next, use `bode` to plot the frequency response of the circuit:

```
bode(G), grid
```



As expected, the RLC filter has maximum gain at the frequency 1 rad/s. However, the attenuation is only -10dB half a decade away from this frequency. To get a narrower passing band, try increasing values of R as follows:

```
R1 = 5;   G1 = tf([1/(R1*C) 0],[1 1/(R1*C) 1/(L*C)]);
R2 = 20;  G2 = tf([1/(R2*C) 0],[1 1/(R2*C) 1/(L*C)]);
bode(G,'b',G1,'r',G2,'g'), grid
legend('R = 1','R = 5','R = 20')
```

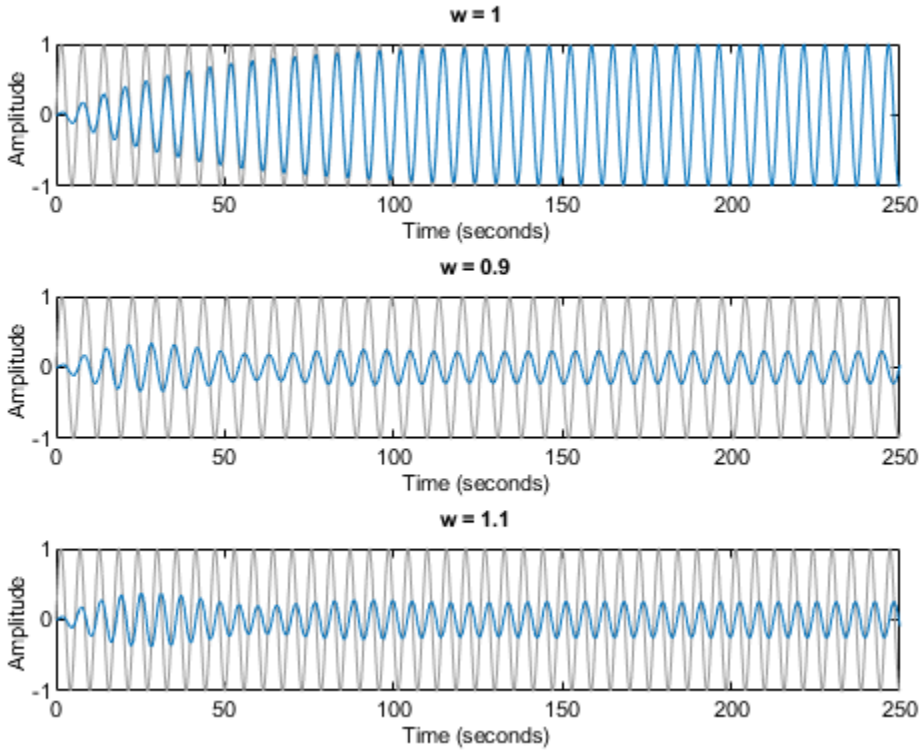


The resistor value `R=20` gives a filter narrowly tuned around the target frequency of 1 rad/s.

Analyzing the Time Response of the Circuit

We can confirm the attenuation properties of the circuit `G2` (`R=20`) by simulating how this filter transforms sine waves with frequency 0.9, 1, and 1.1 rad/s:

```
t = 0:0.05:250;
opt = timeoptions;
opt.Title.FontWeight = 'Bold';
subplot(311), lsim(G2,sin(t),t,opt), title('w = 1')
subplot(312), lsim(G2,sin(0.9*t),t,opt), title('w = 0.9')
subplot(313), lsim(G2,sin(1.1*t),t,opt), title('w = 1.1')
```



The waves at 0.9 and 1.1 rad/s are considerably attenuated. The wave at 1 rad/s comes out unchanged once the transients have died off. The long transient results from the poorly damped poles of the filters, which unfortunately are required for a narrow passing band:

```
damp(pole(G2))
```

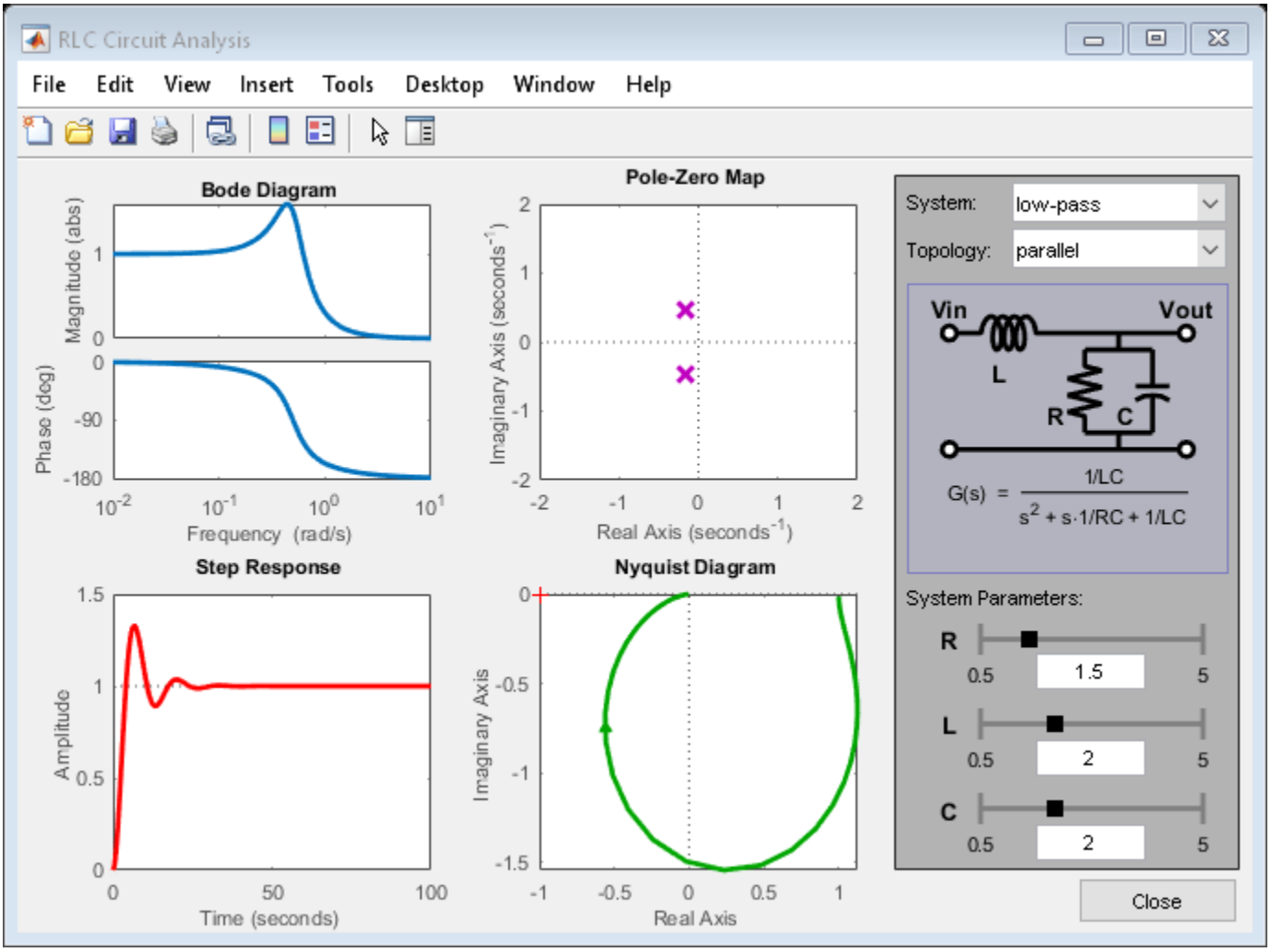
```
        Pole              Damping       Frequency       Time Constant
                                     (rad/TimeUnit)     (TimeUnit)

 -2.50e-02 + 1.00e+00i     2.50e-02       1.00e+00          4.00e+01
 -2.50e-02 - 1.00e+00i     2.50e-02       1.00e+00          4.00e+01
```

Interactive GUI

To analyze other standard circuit configurations such as low-pass and high-pass RLC networks, click on the link below to launch an interactive GUI. In this GUI, you can change the R,L,C parameters and see the effect on the time and frequency responses in real time.

Open the RLC Circuit GUI

```
rlc_gui
```



See Also

`lsim` | `bodeplot` | `stepplot`

Related Examples

- “Joint Time-Domain and Frequency-Domain Analysis” on page 7-32

Sensitivity Analysis

- “Model Array with Single Parameter Variation” on page 9-2
- “Model Array with Variations in Two Parameters” on page 9-5
- “Study Parameter Variation by Sampling Tunable Model” on page 9-7
- “Sensitivity of Control System to Time Delays” on page 9-9
- “Absolute Stability for Quantized System” on page 9-11

Model Array with Single Parameter Variation

This example shows how to create a one-dimensional array of transfer functions using the `stack` command. One parameter of the transfer function varies from model to model in the array. You can use such an array to investigate the effect of parameter variation on your model, such as for sensitivity analysis.

Create an array of transfer functions representing the following low-pass filter at three values of the roll-off frequency, a.

$$F(s) = \frac{a}{s+a}.$$

Create transfer function models representing the filter with roll-off frequency at a = 3, 5, and 7.

```
F1 = tf(3,[1 3]);
F2 = tf(5,[1 5]);
F3 = tf(7,[1 7]);
```

Use the `stack` command to build an array.

```
Farray = stack(1,F1,F2,F3);
```

The first argument to `stack` specifies the array dimension along which `stack` builds an array. The remaining arguments specify the models to arrange along that dimension. Thus, `Farray` is a 3-by-1 array of transfer functions.

Concatenating models with MATLAB® array concatenation commands, instead of with `stack`, creates multi-input, multi-output (MIMO) models rather than model arrays. For example:

```
G = [F1;F2;F3];
```

creates a one-input, three-output transfer function model, not a 3-by-1 array.

When working with a model array that represents parameter variations, You can associate the corresponding parameter value with each entry in the array. Set the `SamplingGrid` property to a data structure that contains the name of the parameter and the sampled parameter values corresponding with each model in the array. This assignment helps you keep track of which model corresponds to which parameter value.

```
Farray.SamplingGrid = struct('alpha',[3 5 7]);
Farray
```

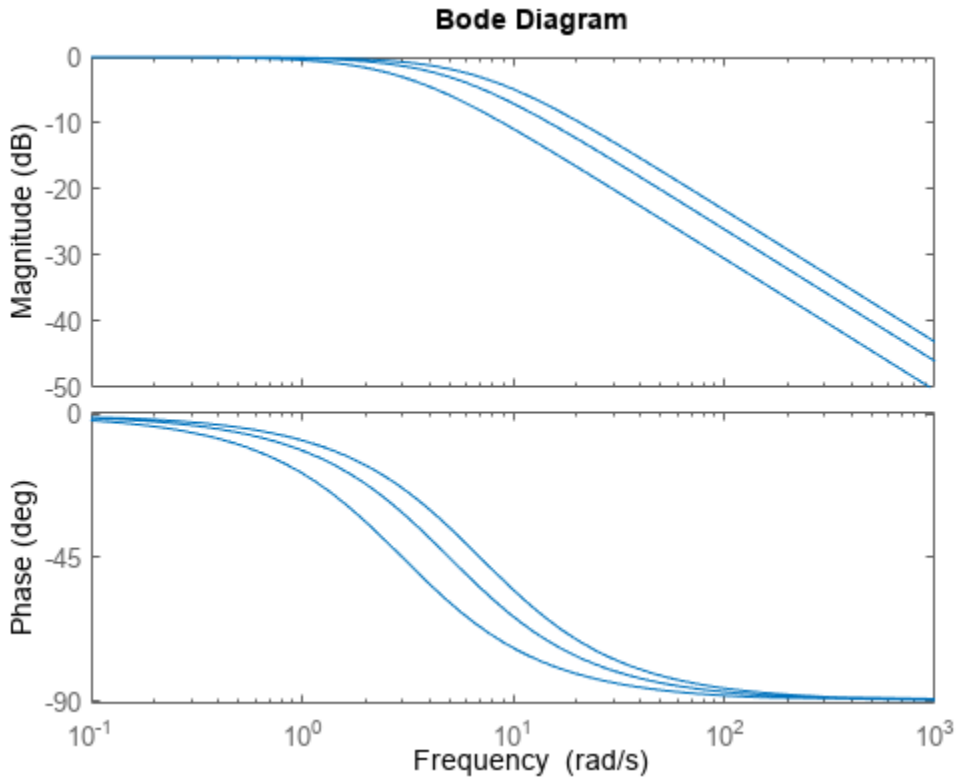
```
Farray(:,:,1,1) [alpha=3] =

    3
  -----
  s + 3


Farray(:,:,2,1) [alpha=5] =

    5
  -----
  s + 5
```

```
Farray(:,:,3,1) [alpha=7] =

    7
  -----
  s + 7

3x1 array of continuous-time transfer functions.
```

The parameter values in Farray.SamplingGrid are displayed along with the each transfer function in the array.

Plot the frequency response of the array to examine the effect of parameter variation on the filter behavior.

```
bodeplot(Farray)
```



When you use analysis commands such as `bodeplot` on a model array, the resulting plot shows the response of each model in the array. Therefore, you can see the range of responses that results from the parameter variation.

See Also

`stack`

More About

- “Model Arrays” on page 2-76
- “Select Models from Array” on page 2-79

Model Array with Variations in Two Parameters

This example shows how to create a two-dimensional (2-D) array of transfer functions using `for` loops. One parameter of the transfer function varies in each dimension of the array.

You can use the technique of this example to create higher-dimensional arrays with variations of more parameters. Such arrays are useful for studying the effects of multiple-parameter variations on system response.

The second-order single-input, single-output (SISO) transfer function

$$H(s) = \frac{\omega^2}{s^2 + 2\zeta\omega s + \omega^2}.$$

depends on two parameters: the damping ratio, ζ, and the natural frequency, ω. If both ζ and ω vary, you obtain multiple transfer functions of the form:

$$H_{ij}(s) = \frac{\omega_j^2}{s^2 + 2\zeta_i\omega_j s + \omega_j^2},$$

where ζ_i and ω_j represent different measurements or sampled values of the variable parameters. You can collect all of these transfer functions in a single variable to create a two-dimensional model array.

Preallocate memory for the model array. Preallocating memory is an optional step that can enhance computation efficiency. To preallocate, create a model array of the required size and initialize its entries to zero.

```
H = tf(zeros(1,1,3,3));
```

In this example, there are three values for each parameter in the transfer function H. Therefore, this command creates a 3-by-3 array of single-input, single-output (SISO) zero transfer functions.

Create arrays containing the parameter values.

```
zeta = [0.66,0.71,0.75];
w = [1.0,1.2,1.5];
```

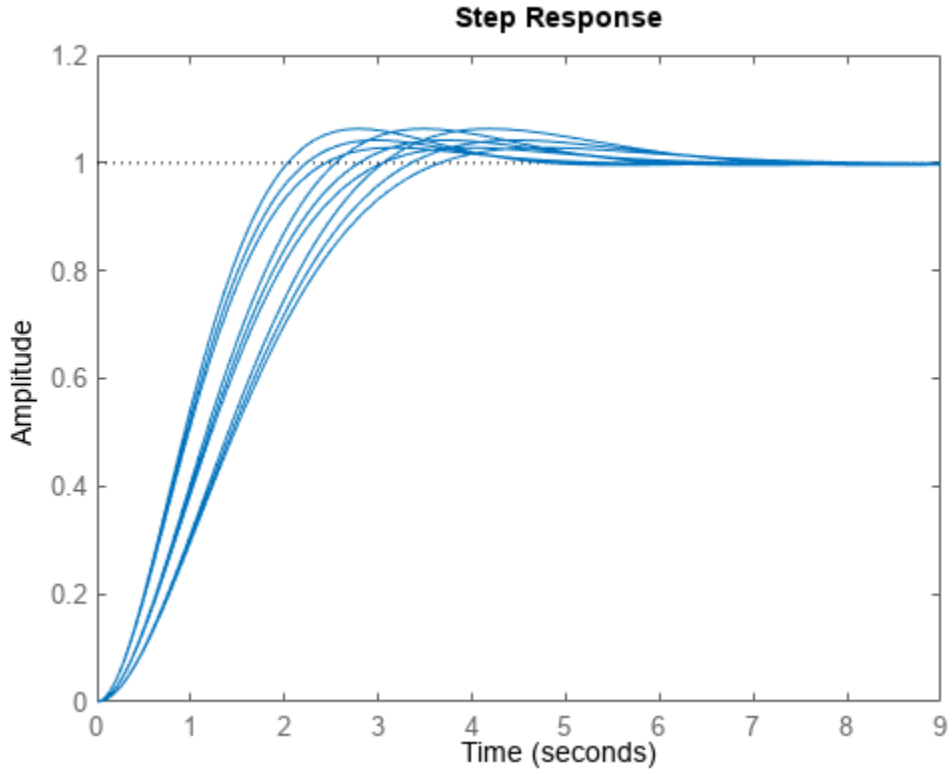
Build the array by looping through all combinations of parameter values.

```
for i = 1:length(zeta)
  for j = 1:length(w)
    H(:,:,i,j) = tf(w(j)^2,[1 2*zeta(i)*w(j) w(j)^2]);
  end
end
```

`H` is a 3-by-3 array of transfer functions. ζ varies as you move from model to model along a single column of `H`. The parameter ω varies as you move along a single row.

Plot the step response of `H` to see how the parameter variation affects the step response.

```
stepplot(H)
```



You can set the `SamplingGrid` property of the model array to help keep track of which set of parameter values corresponds to which entry in the array. To do so, create a grid of parameter values that matches the dimensions of the array. Then, assign these values to `H.SamplingGrid` with the parameter names.

```
[zetagrid,wgrid] = ndgrid(zeta,w);
H.SamplingGrid = struct('zeta',zetagrid,'w',wgrid);
```

When you display `H`, the parameter values in `H.SamplingGrid` are displayed along with the each transfer function in the array.

See Also

`ndgrid`

More About

- “Model Arrays” on page 2-76
- “Study Parameter Variation by Sampling Tunable Model” on page 9-7

Study Parameter Variation by Sampling Tunable Model

This example shows how to sample a parametric model of a second-order filter across a grid of parameter values using `sampleBlock`.

Consider the second-order filter represented by:

$$F(s) = \frac{\omega_n^2}{s^2 + 2\zeta\omega_n s + \omega_n^2}.$$

Sample this filter at varying values of the damping constant ζ and the natural frequency ω_n. Create a parametric model of the filter by using tunable elements for ζ and ω_n.

```
wn = realp('wn',3);
zeta = realp('zeta',0.8);
F = tf(wn^2,[1 2*zeta*wn wn^2])
```

```
Generalized continuous-time state-space model with 1 outputs, 1 inputs, 2 states, and the follow
  wn: Scalar parameter, 5 occurrences.
  zeta: Scalar parameter, 1 occurrences.

Type "ss(F)" to see the current value and "F.Blocks" to interact with the blocks.
```

`F` is a `genss` model with two tunable Control Design Blocks, the `realp` blocks `wn` and `zeta`. The blocks `wn` and `zeta` have initial values of 3 and 0.8, respectively.

Sample `F` over a 2-by-3 grid of (`wn`, `zeta`) values.

```
wnvals = [3;5];
zetavals = [0.6 0.8 1.0];
Fsample = sampleBlock(F,'wn',wnvals,'zeta',zetavals);
```

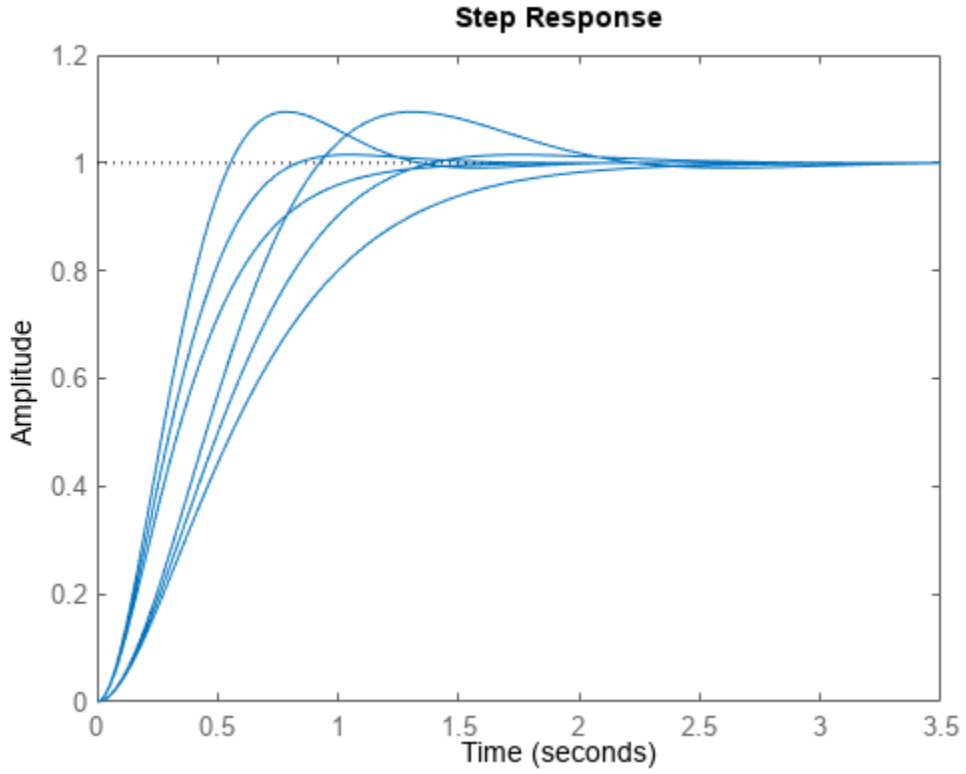
Here, `sampleBlock` samples the model independently over the two ω_n values and three ζ values. Thus, `Fsample` is a 2-by-3 array of state-space models. Each entry in the array is a state-space model that represents `F` evaluated at the corresponding (`wn`, `zeta`) pair. For example, `Fsample(:,:,2,3)` has `wn` = 5 and `zeta` = 1.0.

Set the `SamplingGrid` property of the model array to help keep track of which set of parameter values corresponds to which entry in the array. To do so, create a grid of parameter values that matches the dimensions of the array. Then, assign these values to `Fsample.SamplingGrid` in a structure with the parameter names.

```
[wngrid,zetagrid] = ndgrid(wnvals,zetavals);
Fsample.SamplingGrid = struct('wn',wngrid,'zeta',zetagrid);
```

The `ndgrid` command produces the full 2-by-3 grid of (`wn`, `zeta`) combinations. When you display `Fsample` in the command window, the parameter values in `Fsample.SamplingGrid` are displayed along with the each transfer function in the array. The parameter information is also available in response plots. For instance, examine the step response of `Fsample`.

```
stepplot(Fsample)
```



The step response plots show the variation in the natural frequency and damping constant across the six models in the array. When you click on one of the responses in the plot, the datatip includes the corresponding `wn` and `zeta` values as specified in `Fsample.SamplingGrid`.

See Also

`sampleBlock`

More About

- “Models with Tunable Coefficients” on page 1-15

Sensitivity of Control System to Time Delays

This example shows how to examine the sensitivity of a closed-loop control system to time delays within the system.

Time delays are rarely known accurately, so it is often important to understand how sensitive a control system is to the delay value. Such sensitivity analysis is easily performed using LTI arrays and the `InternalDelay` property. For example, consider the notched PI control system developed in "PI Control Loop with Dead Time" from the example "Analyzing Control Systems with Delays." The following commands create an LTI model of that closed-loop system, a third-order plant with an input delay, a PI controller and a notch filter.

```
s = tf('s');
G = exp(-2.6*s)*(s+3)/(s^2+0.3*s+1);
C = 0.06 * (1 + 1/s);
T = feedback(ss(G*C),1);
notch = tf([1 0.2 1],[1 .8 1]);
C = 0.05 * (1 + 1/s);
Tnotch = feedback(ss(G*C*notch),1);
```

Examine the internal delay of the closed-loop system Tnotch.

```
Tnotch.InternalDelay
```

```
ans = 2.6000
```

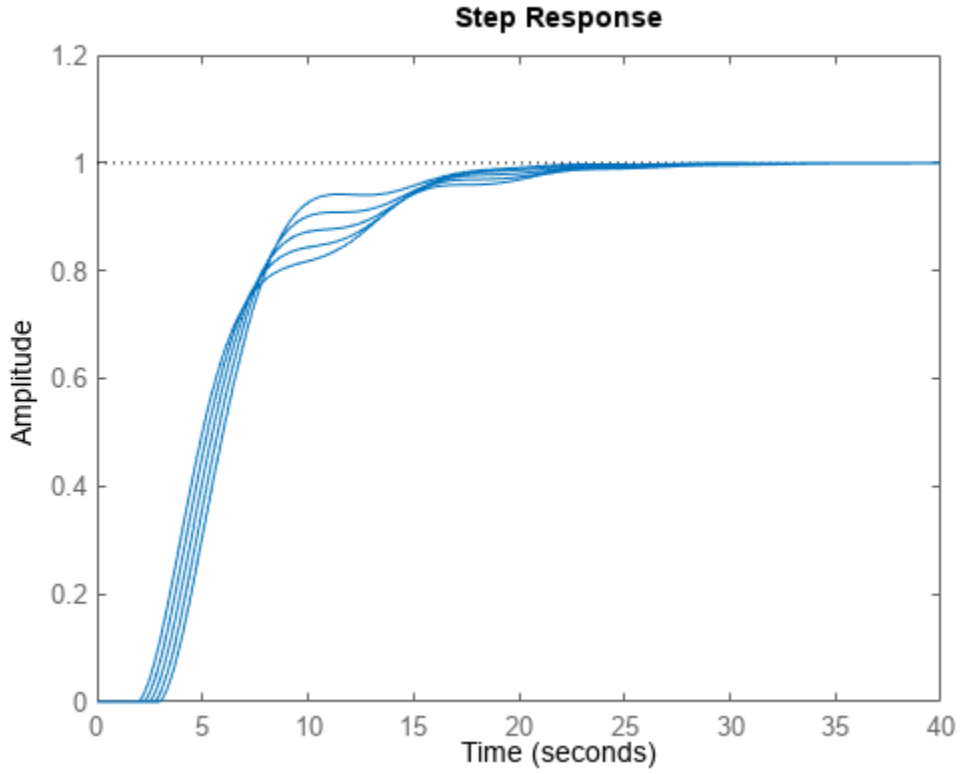
The 2.6-second input delay of the plant `G` becomes an internal delay of 2.6 s in the closed-loop system. To examine the sensitivity of the responses of Tnotch to variations in this delay, create an array of copies of Tnotch. Then, vary the internal delay across the array.

```
Tsens = repsys(Tnotch,[1 1 5]);
tau = linspace(2,3,5);
for j = 1:5;
    Tsens(:,:,j).InternalDelay = tau(j);
end
```

The array `Tsens` contains five models with internal delays that range from 2.0 to 3.0.

Examine the step responses of these models.

```
stepplot(Tsens)
```



The plot shows that uncertainty on the delay value has a small effect on closed-loop characteristics.

See Also

More About

- “Time Delays in Linear Systems” on page 2-31

Absolute Stability for Quantized System

This example shows how to enforce absolute stability when a linear time-invariant system is in feedback interconnection with a static nonlinearity that belongs to a conic sector.

Feedback Connection

Consider the feedback connection as shown in Figure 1.

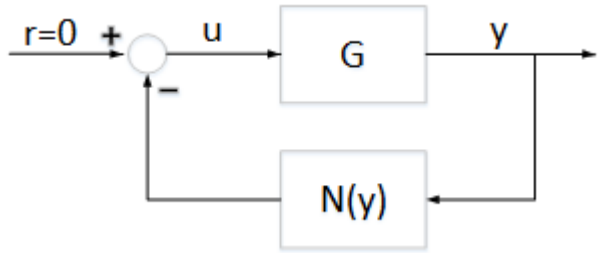


Figure 1: Feedback connection

G is a linear time invariant system, and $N(y)$ is a static nonlinearity that belongs to a conic sector $[\alpha, \beta]$ (where $\alpha < \beta$); that is,

$$\alpha y^2 < \ yN(y) < \ \beta y^2$$

For this example, G is the following discrete-time system.

```
A = [0.9995, 0.0100, 0.0001;
    -0.0020, 0.9995, 0.0106;
          0,      0, 0.9978];
B = [0, 0.002, 0.04]';
C = [2.3948, 0.3303, 2.2726];
D = 0;
G = ss(A,B,C,D,0.01);
```

Sector Bounded Nonlinearity

In this example, the nonlinearity $N(y)$ is the logarithmic quantizer, which is defined as follows:

$$N(y) = \begin{cases} \rho^j, & \text{if } \frac{1+\rho}{2}\rho^j < y \le \frac{1+\rho}{2\rho}\rho^j; \\ 0, & \text{if } y = 0; \\ -N(-y), & \text{if } y < 0 \end{cases}$$

where, $j \in \{0, \pm 1, \pm 2, \ldots\}$. This quantizer belongs to a sector bound $[\frac{2\rho}{1+\rho}, \frac{2}{1+\rho}]$. For example, if $\rho = 0.1$, then the quantizer belongs to the conic sector [0.1818,1.8182].

```
% Quantizer parameter
rho = 0.1;
% Lower bound
alpha = 2*rho/(1+rho)
```

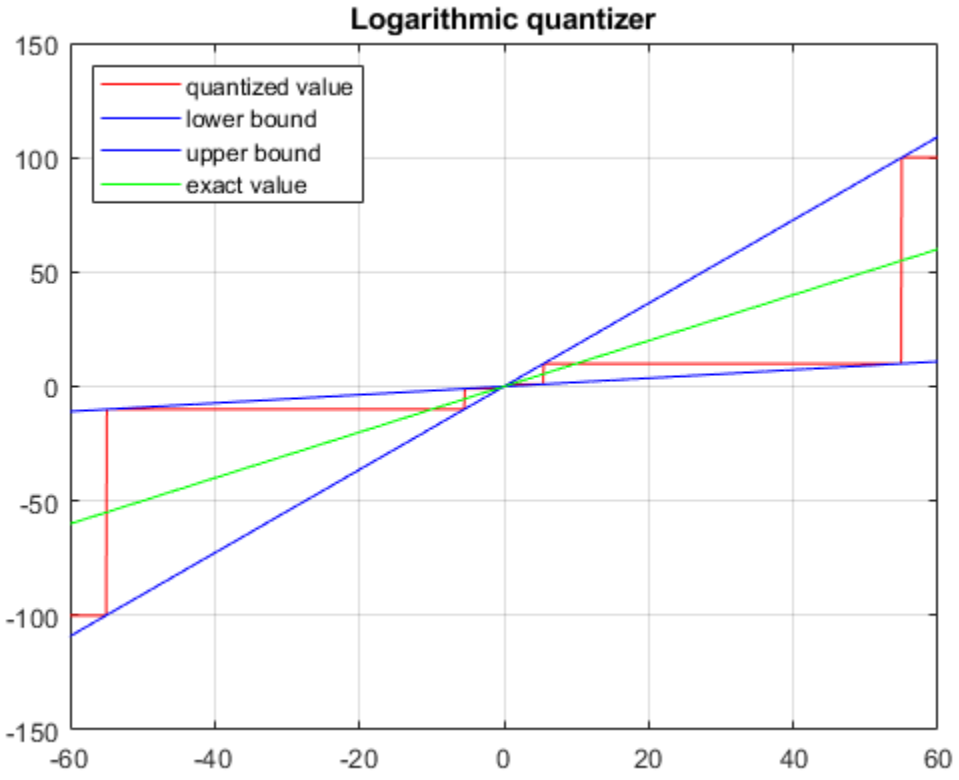
```
% Upper bound
beta = 2/(1+rho)
```

```
alpha =

    0.1818
```

```
beta =

    1.8182
```

Plot the sector bounds for the quantizer.

```
PlotSectorBound(rho)
```



ρ represents the quantization density, where $0 < \rho < 1$. If ρ is larger, then the quantized value is more accurate. For more details about this quantizer, see [1].

Conic Sector Condition for Absolute Stability

The conic sector matrix for the quantizer is given by

$$Q = \begin{pmatrix} 1 & -\frac{\alpha+\beta}{2} \\ -\frac{\alpha+\beta}{2} & \alpha\beta \end{pmatrix}.$$

To guarantee stability of the feedback connection in Figure 1, the linear system G needs to satisfy

$$\int_0^T \begin{pmatrix} u(t) \\ -y(t) \end{pmatrix}^T Q \begin{pmatrix} u(t) \\ -y(t) \end{pmatrix} > 0$$

where, u and y are the input and output of G, respectively.

This condition can be verified by checking if the sector index, R, is less than 1.

Define the conic sector matrix for a quantizer with $\rho = 0.1$.

```
Q = [1,-(alpha+beta)/2;-(alpha+beta)/2,alpha*beta];
```

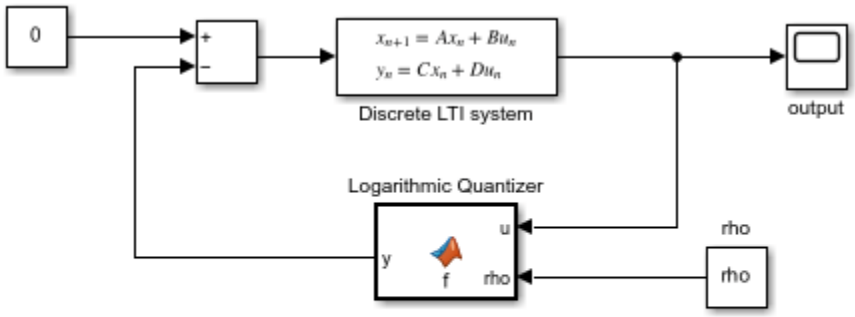
Get the sector index for `Q` and `G`.

```
R = getSectorIndex([1;-G],-Q)
```

```
R =

    1.8247
```

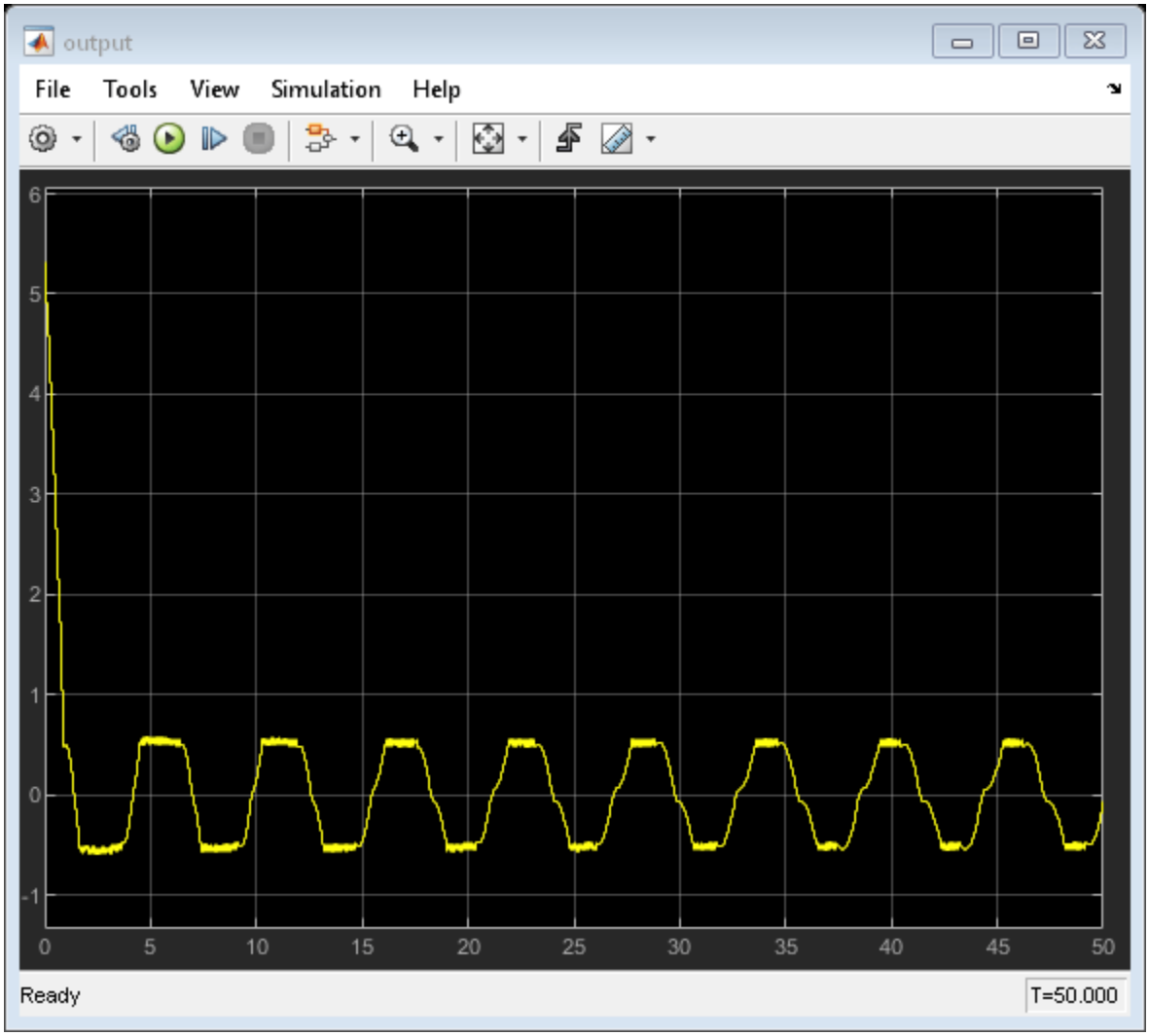
Since $R > 1$, the closed-loop system is not stable. To see this instability, use the following Simulink model.

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



Run the Simulink model.

```
sim(mdl)
open_system('DTQuantization/output')
```



From the output trajectory, it can be seen that the closed-loop system is not stable. This is because the quantizer with $\rho = 0.1$ is too coarse.

Increase the quantization density by letting $\rho = 0.25$. The quantizer belongs to the conic sector [0.4,1.6].

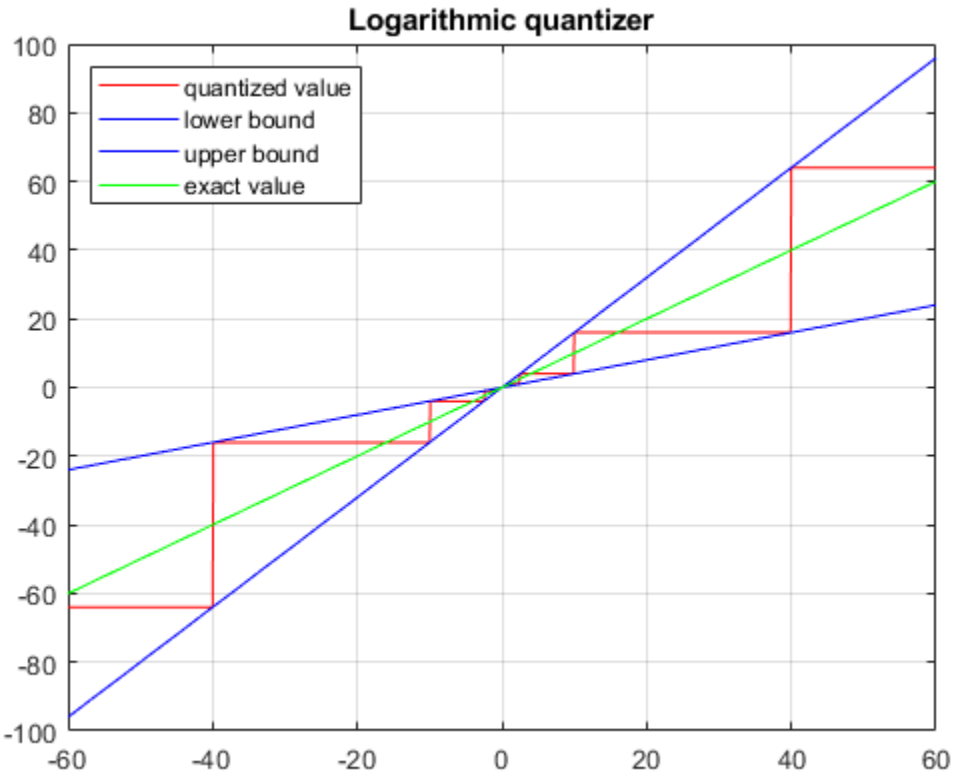
```
% Quantizer parameter
rho = 0.25;
% Lower bound
alpha = 2*rho/(1+rho)
% Upper bound
beta = 2/(1+rho)
```

```
alpha =

    0.4000
```

```
beta =

    1.6000
```

Plot the sector bounds for the quantizer.

```
PlotSectorBound(rho)
```



Define the conic sector matrix for a quantizer with $\rho = 0.25$.

```
Q = [1,-(alpha+beta)/2;-(alpha+beta)/2,alpha*beta];
```

Get the sector index for `Q` and `G`.

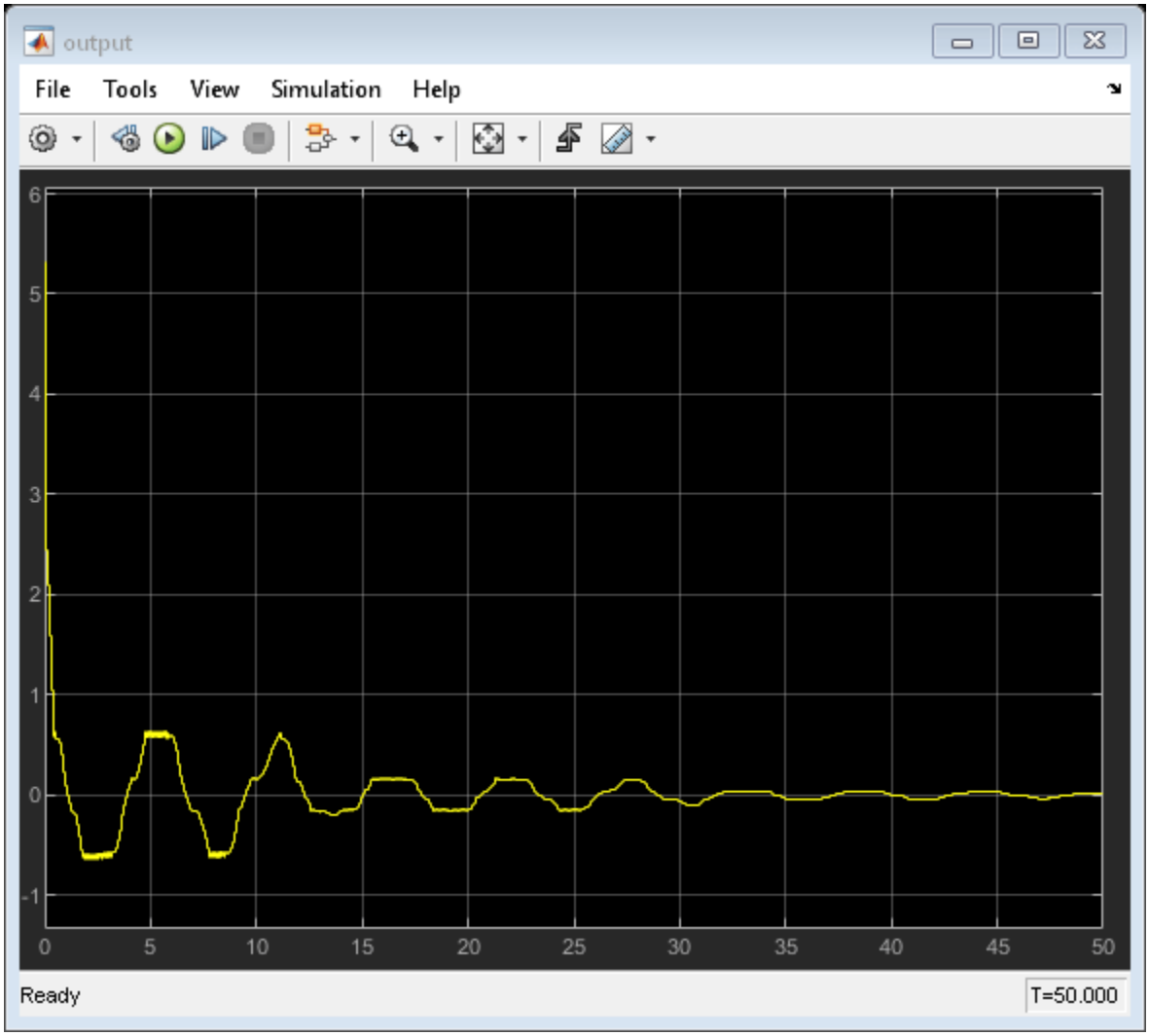
```
R = getSectorIndex([1;-G],-Q)
```

```
R =

    0.9702
```

The quantizer with $\rho = 0.25$ satisfies the conic sector condition for stability of the feedback connection since $R < 1$.

Run the Simulink model with $\rho = 0.25$.

```
sim(mdl)
open_system('DTQuantization/output')
```



As indicated by the sector index, the closed-loop system is stable.

Reference

[1] M. Fu and L. Xie,"The sector bound approach to quantized feedback control," *IEEE Transactions on Automatic Control* 50(11), 2005, 1698-1711.

Passivity and Conic Sectors

- “About Passivity and Passivity Indices” on page 10-2
- “About Sector Bounds and Sector Indices” on page 10-7
- “Passivity Indices” on page 10-14
- “Parallel Interconnection of Passive Systems” on page 10-18
- “Series Interconnection of Passive Systems” on page 10-20
- “Feedback Interconnection of Passive Systems” on page 10-23

About Passivity and Passivity Indices

Passive control is often part of the safety requirements in applications such as process control, tele-operation, human-machine interfaces, and system networks. A system is *passive* if it cannot produce energy on its own, and can only dissipate the energy that is stored in it initially. More generally, an I/O map is passive if, on average, increasing the output y requires increasing the input u.

For example, a PID controller is passive because the control signal (the output) moves in the same direction as the error signal (the input). But a PID controller with delay is not passive, because the control signal can move in the opposite direction from the error, a potential cause of instability.

Most physical systems are passive. The Passivity Theorem holds that the negative-feedback interconnection of two strictly passive systems is passive and stable. As a result, it can be desirable to enforce passivity of the controller for a passive system, or to *passivate* the operator of a passive system, such as the driver of a car.

In practice, passivity can easily be destroyed by the phase lags introduced by sensors, actuators, and communication delays. These problems have led to extension of the Passivity Theorem that consider excesses or shortages of passivity, frequency-dependent measures of passivity, and a mix of passivity and small-gain properties.

Passive Systems

A linear system $G(s)$ is passive if all input/output trajectories $y(t) = Gu(t)$ satisfy:

$$\int_0^T y^T(t)u(t)dt > 0, \quad \forall T > 0,$$

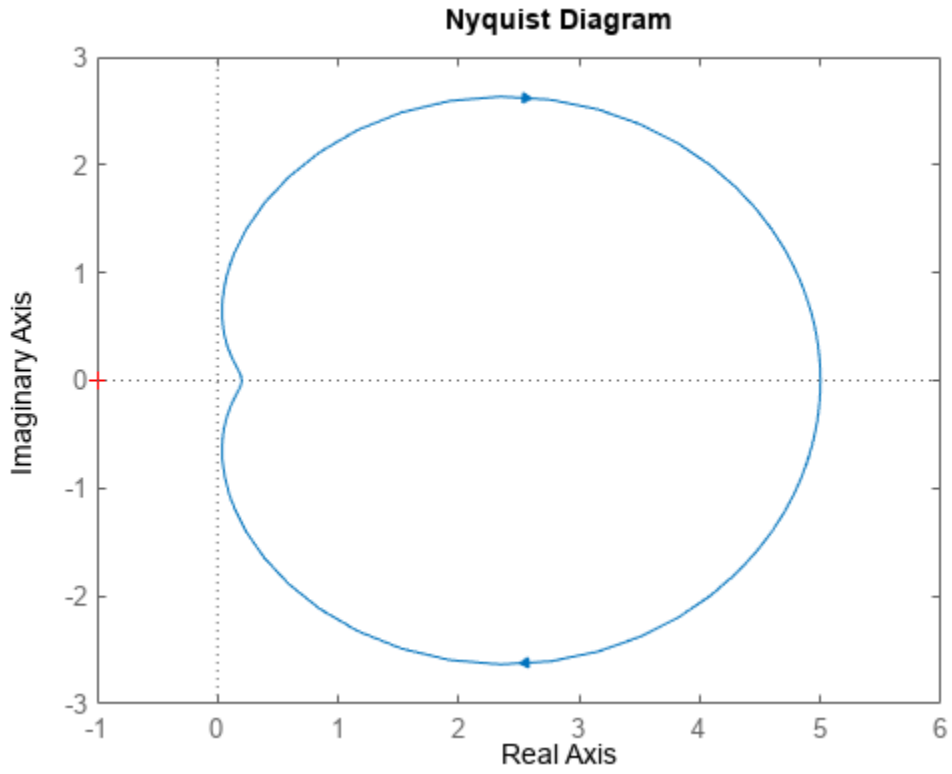
where $y^T(t)$ denotes the transpose of $y(t)$. For physical systems, the integral typically represents the energy going into the system. Thus passive systems are systems that only consume or dissipate energy. As a result, passive systems are intrinsically stable.

In the frequency domain, passivity is equivalent to the "positive real" condition:

$$G(j\omega) + G^H(j\omega) > 0, \quad \forall \omega \in \mathbf{R}.$$

For SISO systems, this is saying that $\mathrm{Re}(G(j\omega)) > 0$ at all frequencies, so the entire Nyquist plot lies in the right-half plane.

```
nyquist(tf([1 3 5],[5 6 1]))
```



Nyquist plot of passive system

Passive systems have the following important properties for control purposes:

- The inverse of a passive system is passive.
- The parallel interconnection of passive systems is passive (see “Parallel Interconnection of Passive Systems” on page 10-18).
- The feedback interconnection of passive systems is passive (see “Feedback Interconnection of Passive Systems” on page 10-23).

When controlling a passive system with unknown or variable characteristics, it is therefore desirable to use a passive feedback law to guarantee closed-loop stability. This task can be rendered difficult given that delays and significant phase lag destroy passivity.

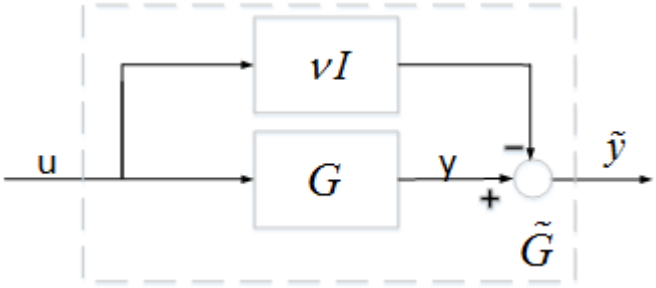
Directional Passivity Indices

For stability, knowing whether a system is passive or not does not tell the full story. It is often desirable to know by how much it is passive or fails to be passive. In addition, a shortage of passivity in the plant can be compensated by an excess of passivity in the controller, and vice versa. It is therefore important to measure the excess or shortage of passivity, and this is where passivity indices come into play.

There are different types of indices with different applications. One class of indices measure the excess or shortage of passivity in a particular direction of the input/output space. For example, the input passivity index is defined as the largest ν such that:

$$\int_0^T y^T(t)u(t)dt > \nu\int_0^T u^T(t)u(t)dt,$$

for all trajectories $y(t) = Gu(t)$ and $T > 0$. The system G is *input strictly passive* (ISP) when $\nu > 0$, and has a shortage of passivity when $\nu < 0$. The input passivity index is also called the input feedforward passivity (IFP) index because it corresponds to the minimum static feedforward action needed to make the system passive.



In the frequency domain, the input passivity index is characterized by:

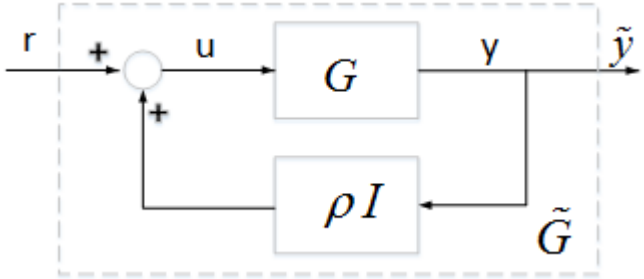
$$\nu = \frac{1}{2}\min_{\omega}\lambda_{\min}\left(G(j\omega) + G^H(j\omega)\right),$$

where $\lambda_{\min}$ denotes the smallest eigenvalue. In the SISO case, ν is the abscissa of the leftmost point on the Nyquist curve.

Similarly, the output passivity index is defined as the largest ρ such that:

$$\int_0^T y^T(t)u(t)dt > \rho\int_0^T y^T(t)y(t)dt,$$

for all trajectories $y(t) = Gu(t)$ and $T > 0$. The system G is *output strictly passive* (OSP) when $\rho > 0$, and has a shortage of passivity when $\rho < 0$. The output passivity index is also called the output feedback passivity (OFP) index because it corresponds to the minimum static feedback action needed to make the system passive.



In the frequency domain, the output passivity index of a *minimum-phase* system $G(s)$ is given by:

$$\rho = \frac{1}{2}\min_{\omega}\lambda_{\min}\left(G^{-1}(j\omega) + G^{-H}(j\omega)\right).$$

In the SISO case, ρ is the abscissa of the leftmost point on the Nyquist curve of $G^{-1}(s)$.

Combining these two notions leads to the I/O passivity index, which is the largest τ such that:

$$\int_0^T y^T(t)u(t)dt > \tau \int_0^T \left(u^T(t)u(t) + y^T(t)y(t)\right)dt\,.$$

A system with $\tau > 0$ is *very strictly passive*. More generally, we can define the index in the direction δQ as the largest τ such that:

$$\int_0^T y^T(t)u(t)dt > \tau \int_0^T \begin{pmatrix} y(t) \\ u(t) \end{pmatrix}^T \delta Q \begin{pmatrix} y(t) \\ u(t) \end{pmatrix} dt\,.$$

The input, output, and I/O passivity indices all correspond to special choices of δQ and are collectively referred to as *directional passivity indices*. You can use `getPassiveIndex` to compute any of these indices for linear systems in either parametric or FRD form. You can also use `passiveplot` to plot the input, output, or I/O passivity indices as a function of frequency. This plot provides insight into which frequency bands have weaker or stronger passivity.

There are many results quantifying how the input and output passivity indices propagate through parallel, series, or feedback interconnections. There are also results quantifying the excess of input or output passivity needed to compensate a given shortage of passivity in a feedback loop. For details, see:

- “Parallel Interconnection of Passive Systems” on page 10-18
- “Series Interconnection of Passive Systems” on page 10-20
- “Feedback Interconnection of Passive Systems” on page 10-23

Relative Passivity Index

The *positive real* condition for passivity:

$$G(j\omega) + G^H(j\omega) > 0 \quad \forall \omega \in \mathbf{R},$$

is equivalent to the small gain condition:

$$||(I - G(j\omega))(I + G(j\omega))^{-1}|| < 1 \quad \forall \omega \in \mathbf{R}\,.$$

We can therefore use the peak gain of $(I - G)(I + G)^{-1}$ as a measure of passivity. Specifically, let

$$R := \left\|(I - G)(I + G)^{-1}\right\|_\infty\,.$$

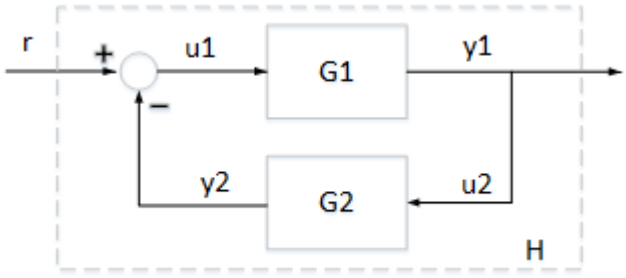
Then G is passive if and only if $R < 1$, and $R > 1$ indicates a shortage of passivity. Note that R is finite if and only if $I + G$ is minimum phase. We refer to R as the *relative passivity index*, or R-index. In the time domain, the R-index is the smallest $r > 0$ such that:

$$\int_0^T ||y - u||^2 dt < r^2 \int_0^T ||y + u||^2 dt,$$

for all trajectories $y(t) = Gu(t)$ and $T > 0$. When $I + G$ is minimum phase, you can use `passiveplot` to plot the principal gains of $(I - G(j\omega))(I + G(j\omega))^{-1}$. This plot is entirely analogous to the singular value plot (see `sigma`), and shows how the degree of passivity changes with frequency and direction.

The following result is analogous to the Small Gain Theorem for feedback loops. It gives a simple condition on R-indices for compensating a shortage of passivity in one system by an excess of passivity in the other.

Small-R Theorem: Let $G_1(s)$ and $G_2(s)$ be two linear systems with passivity R-indices R_1 and R_2, respectively. If $R_1R_2 < 1$, then the negative feedback interconnection of G_1 and G_2 is stable.



References

[1] Xia, M., P. Gahinet, N. Abroug, C. Buhr, and E. Laroche. “Sector Bounds in Stability Analysis and Control Design.” *International Journal of Robust and Nonlinear Control* 30, no. 18 (December 2020): 7857–82. https://doi.org/10.1002/rnc.5236.

See Also

`isPassive` | `getPassiveIndex` | `passiveplot`

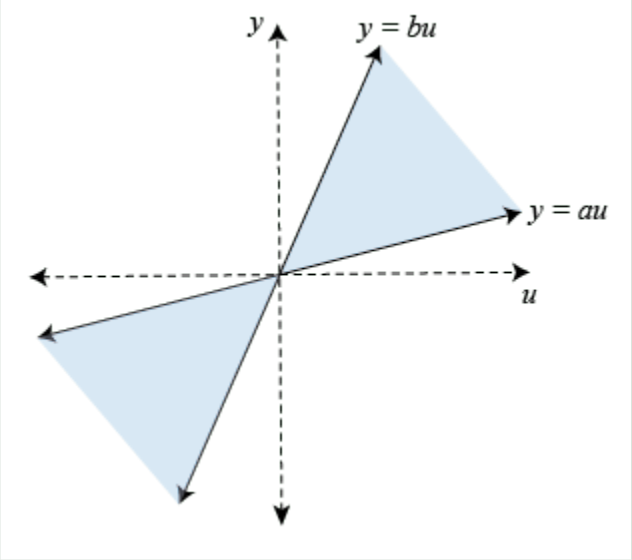
Related Examples

- “Passivity Indices” on page 10-14
- “Parallel Interconnection of Passive Systems” on page 10-18
- “Series Interconnection of Passive Systems” on page 10-20
- “Feedback Interconnection of Passive Systems” on page 10-23
- “About Sector Bounds and Sector Indices” on page 10-7

About Sector Bounds and Sector Indices

Conic Sectors

In its simplest form, a conic sector is the 2-D region delimited by two lines, $y = au$ and $y = bu$.



The shaded region is characterized by the inequality $(y - au)(y - bu) < 0$. More generally, any such sector can be parameterized as:

$$\begin{pmatrix} y \\ u \end{pmatrix}^T Q \begin{pmatrix} y \\ u \end{pmatrix} < 0,$$

where Q is a 2x2 symmetric indefinite matrix (Q has one positive and one negative eigenvalue). We call Q the *sector matrix*. This concept generalizes to higher dimensions. In an N-dimensional space, a conic sector is a set:

$$\mathbf{S} = \left\{ z \in \mathbf{R}^N : \ z^T Q z < 0 \right\},$$

where Q is again a symmetric indefinite matrix.

Sector Bounds

Sector bounds are constraints on the behavior of a system. Gain constraints and passivity constraints are special cases of sector bounds. If for all nonzero input trajectories $u(t)$, the output trajectory $z(t) = (Hu)(t)$ of a linear system $H(s)$ satisfies:

$$\int_0^T z^T(t) Q z(t) dt < 0, \quad \forall T > 0,$$

then the output trajectories of H lie in the conic sector with matrix Q. Selecting different Q matrices imposes different conditions on the system's response. For example, consider trajectories $y(t) = (Gu)(t)$ and the following values:

$$H(s) = \begin{pmatrix} G(s) \\ I \end{pmatrix}, \quad Q = \begin{pmatrix} 0 & -I \\ -I & 0 \end{pmatrix}.$$

These values correspond to the sector bound:

$$\int_0^T \begin{pmatrix} y(t) \\ u(t) \end{pmatrix}^T \begin{pmatrix} 0 & -I \\ -I & 0 \end{pmatrix} \begin{pmatrix} y(t) \\ u(t) \end{pmatrix} dt < 0, \quad \forall T > 0 .$$

This sector bound is equivalent to the passivity condition for $G(s)$:

$$\int_0^T y^T(t)u(t)dt > 0, \quad \forall T > 0 .$$

In other words, passivity is a particular sector bound on the system defined by:

$$H = \begin{pmatrix} G \\ I \end{pmatrix}.$$

Frequency-Domain Condition

Because the time-domain condition must hold for all $T > 0$, deriving an equivalent frequency-domain bound takes a little care and is not always possible. Let the following:

$$Q = W_1^T W_1 - W_2^T W_2$$

be (any) decomposition of the indefinite matrix Q into its positive and negative parts. When $W_2^T H(s)$ is square and minimum phase (has no unstable zeros), the time-domain condition:

$$\int_0^T (Hu)(t)^T \, Q \, (Hu)(t)dt < 0, \quad \forall T > 0$$

is equivalent to the frequency-domain condition:

$$H(j\omega)^H Q H(j\omega) < 0 \quad \forall \omega \in \mathbf{R} .$$

It is therefore enough to check the sector inequality for real frequencies. Using the decomposition of Q, this is also equivalent to:

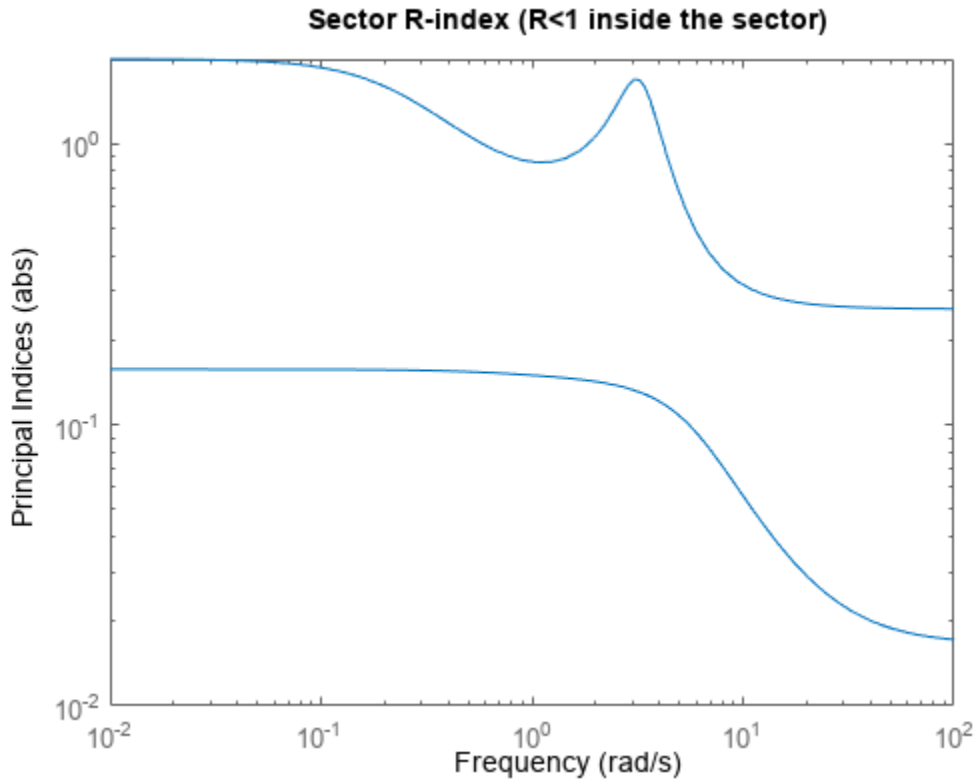
$$\left\| (W_1^T H)(W_2^T H)^{-1} \right\|_\infty < 1 .$$

Note that $W_2^T H$ is square when Q has as many negative eigenvalues as input channels in $H(s)$. If this condition is not met, it is no longer enough (in general) to just look at real frequencies. Note also that if $W_2^T H(s)$ is square, then it must be minimum phase for the sector bound to hold.

This frequency-domain characterization is the basis for `sectorplot`. Specifically, `sectorplot` plots the singular values of $(W_1^T H(j\omega))(W_2^T H(j\omega))^{-1}$ as a function of frequency. The sector bound is satisfied if and only if the largest singular value stays below 1. Moreover, the plot contains useful information about the frequency bands where the sector bound is satisfied or violated, and the degree to which it is satisfied or violated.

For instance, examine the sector plot of a 2-output, 2-input system for a particular sector.

```
load("sectorExampleSystem.mat","H1")
Q = [-5.12   2.16  -2.04   2.17
      2.16  -1.22  -0.28  -1.11
     -2.04  -0.28  -3.35   0.00
      2.17  -1.11   0.00   0.18];
sectorplot(H1,Q)
```



The plot shows that the largest singular value of $(W_1^T H(j\omega))(W_2^T H(j\omega))^{-1}$ exceeds 1 below about 0.5 rad/s and in a narrow band around 3 rad/s. Therefore, `H` does not satisfy the sector bound represented by `Q`.

Relative Sector Index

We can extend the notion of relative passivity index to arbitrary sectors. Let $H(s)$ be an LTI system, and let:

$$Q = W_1^T W_1 - W_2^T W_2, \quad W_1^T W_2 = 0$$

be an orthogonal decomposition of Q into its positive and negative parts, as is readily obtained from the Schur decomposition of Q. The *relative sector index* R, or R-index, is defined as the smallest $r > 0$ such that for all output trajectories $z(t) = (Hu)(t)$:

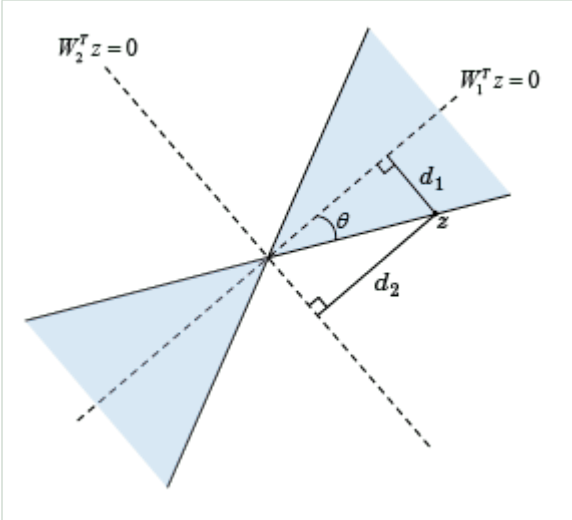
$$\int_0^T z^T(t)\,(W_1^T W_1 - r^2 W_2^T W_2)\,z(t)dt < 0, \quad \forall T > 0\,.$$

Because increasing r makes $W_1^T W_1 - r^2 W_2^T W_2$ more negative, the inequality is usually satisfied for r large enough. However, there are cases when it can never be satisfied, in which case the R-index is $R = +\infty$. Clearly, the original sector bound is satisfied if and only of $R \le 1$.

To understand the geometrical interpretation of the R-index, consider the family of cones with matrix $Q(r) = W_1^T W_1 - r^2 W_2^T W_2$. In 2D, the cone slant angle θ is related to r by

$$\tan(\theta) = r\frac{\|W_2\|}{\|W_1\|}$$

(see diagram below). More generally, $\tan(\theta)$ is proportional to R. Thus, given a conic sector with matrix Q, an R-index value $R < 1$ means that we can reduce $\tan(\theta)$ (narrow the cone) by a factor R before some output trajectory of H leaves the conic sector. Similarly, a value $R > 1$ means that we must increase $\tan(\theta)$ (widen the cone) by a factor R to include all output trajectories of H. This clearly makes the R-index a relative measure of how well the response of H fits in a particular conic sector.



In the diagram,

$$d_1\frac{\left|W_1^T z\right|}{\|W_1\|}, \quad d_2\frac{\left|W_2^T z\right|}{\|W_2\|}, \quad R = \frac{\left|W_1^T z\right|}{\left|W_2^T z\right|},$$

and

$$\tan(\theta) = \frac{d_1}{d_2} = R\frac{\|W_2\|}{\|W_1\|}.$$

When $W_2^T H(s)$ is square and minimum phase, the R-index can also be characterized in the frequency domain as the smallest $r > 0$ such that:

$$H(j\omega)^H \left(W_1^T W_1 - r^2 W_2^T W_2\right) H(j\omega) < 0 \quad \forall \omega \in \mathbf{R}.$$

Using elementary algebra, this leads to:

$$R = \max_{\omega} \left\| (W_1^T H(j\omega))(W_2^T H(j\omega))^{-1} \right\|.$$

In other words, the R-index is the peak gain of the (stable) transfer function $\Phi(s) := (W_1^T H(s))(W_2^T H(s))^{-1}$, and the singular values of $\Phi(jw)$ can be seen as the "principal" R-indices at each frequency. This also explains why plotting the R-index vs. frequency looks like a singular value plot (see `sectorplot`). There is a complete analogy between relative sector index and system gain. Note, however, that this analogy only holds when $W_2^T H(s)$ is square and minimum phase.

Directional Sector Index

Similarly, we can extend the notion of directional passivity index to arbitrary sectors. Given a conic sector with matrix Q, and a direction δQ, the directional sector index is the largest τ such that for all output trajectories $z(t) = (Hu)(t)$:

$$\int_0^T z^T(t)\,(Q + \tau\delta Q)\,z(t)dt < 0, \quad \forall T > 0\,.$$

The directional passivity index for a system $G(s)$ corresponds to:

$$H(s) = \begin{pmatrix} G(s) \\ I \end{pmatrix}, \quad Q = \begin{pmatrix} 0 & -I \\ -I & 0 \end{pmatrix}.$$

The directional sector index measures by how much we need to deform the sector in the direction δQ to make it fit tightly around the output trajectories of H. The sector bound is satisfied if and only if the directional index is positive.

Common Sectors

There are many ways to specify sector bounds. Next we review commonly encountered expressions and give the corresponding system H and sector matrix Q for the standard form used by `getSectorIndex` and `sectorplot`:

$$\int_0^T (Hu)(t)^T Q (Hu)(t)dt < 0, \quad \forall T > 0\,.$$

For simplicity, these descriptions use the notation:

$$\|x\|_T = \int_0^T \|x(t)\|^2 dt,$$

and omit the $\forall T > 0$ requirement.

Passivity

Passivity is a sector bound with:

$$H(s) = \begin{pmatrix} G(s) \\ I \end{pmatrix}, \quad Q = \begin{pmatrix} 0 & -I \\ -I & 0 \end{pmatrix}.$$

Gain constraint

The gain constraint $\|G\|_\infty < \gamma$ is a sector bound with:

$$H(s) = \begin{pmatrix} G(s) \\ I \end{pmatrix}, \quad Q = \begin{pmatrix} I & 0 \\ 0 & -\gamma^2 I \end{pmatrix}.$$

Ratio of distances

Consider the "interior" constraint,

$$\|y - cu\|_T < r\|u\|_T$$

where c, r are scalars and $y(t) = (Gu)(t)$. This is a sector bound with:

$$H(s) = \begin{pmatrix} G(s) \\ I \end{pmatrix}, \quad Q = \begin{pmatrix} I & -cI \\ -cI & (c^2 - r^2)I \end{pmatrix}.$$

The underlying conic sector is symmetric with respect to $y = cu$. Similarly, the "exterior" constraint,

$$\|y - cu\|_T > r\|u\|_T$$

is a sector bound with:

$$H(s) = \begin{pmatrix} G(s) \\ I \end{pmatrix}, \quad Q = \begin{pmatrix} -I & cI \\ cI & (r^2 - c^2)I \end{pmatrix}.$$

Double inequality

When dealing with static nonlinearities, it is common to consider conic sectors of the form

$$au^2 < yu < bu^2,$$

where $y = \phi(u)$ is the nonlinearity output. While this relationship is not a sector bound per se, it clearly implies:

$$a\int_0^T u(t)^2 dt < \int_0^T y(t)u(t)dt < b\int_0^T u(t)^2 dt$$

along all I/O trajectories and for all $T > 0$. This condition in turn is equivalent to a sector bound with:

$$H(s) = \begin{pmatrix} \phi(\,.\,) \\ 1 \end{pmatrix}, \quad Q = \begin{pmatrix} 1 & -(a+b)/2 \\ -(a+b)/2 & ab \end{pmatrix}.$$

Product form

Generalized sector bounds of the form:

$$\int_0^T (y(t) - K_1 u(t))^T (y(t) - K_2 u(t))dt < 0$$

correspond to:

$$H(s) = \begin{pmatrix} G(s) \\ I \end{pmatrix}, \quad Q = \begin{pmatrix} 2I & -(K_2 + K_1^T) \\ -(K_1 + K_2^T) & K_1^T K_2 + K_2^T K_1 \end{pmatrix}.$$

As before, the static sector bound:

$$(y - K_1 u)^T (y - K_2 u) < 0$$

implies the integral sector bound above.

QSR dissipative

A system $y = Gu$ is QSR-dissipative if it satisfies:

$$\int_0^T \begin{pmatrix} y(t) \\ u(t) \end{pmatrix}^T \begin{pmatrix} \mathbf{Q} & \mathbf{S} \\ \mathbf{S^T} & \mathbf{R} \end{pmatrix} \begin{pmatrix} y(t) \\ u(t) \end{pmatrix} dt > 0, \quad \forall T > 0 \,.$$

This is a sector bound with:

$$H(s) = \begin{pmatrix} G(s) \\ I \end{pmatrix}, \quad Q = -\begin{pmatrix} \mathbf{Q} & \mathbf{S} \\ \mathbf{S^T} & \mathbf{R} \end{pmatrix}.$$

References

[1] Xia, M., P. Gahinet, N. Abroug, C. Buhr, and E. Laroche. “Sector Bounds in Stability Analysis and Control Design.” *International Journal of Robust and Nonlinear Control* 30, no. 18 (December 2020): 7857–82. https://doi.org/10.1002/rnc.5236.

See Also

`getSectorIndex` | `sectorplot` | `getSectorCrossover`

Related Examples

• “About Passivity and Passivity Indices” on page 10-2

Passivity Indices

This example shows how to compute various measures of passivity for linear time-invariant systems.

Passive Systems

A linear system G(s) is passive when all I/O trajectories $(u(t), y(t))$ satisfy

$$\int_0^T y^T(t)u(t)dt > 0, \quad \forall T > 0$$

where $y^T(t)$ denotes the transpose of $y(t)$.

To measure "how passive" a system is, we use passivity indices.

- The input passivity index is defined as the largest ν such that

$$\int_0^T y^T(t)u(t)dt > \nu \int_0^T u^T(t)u(t)dt$$

The system G is "input strictly passive" (ISP) when $\nu > 0$. ν is also called the "input feedforward passivity" (IFP) index and corresponds to the minimum feedforward action needed to make the system passive.

- The output passivity index is defined as the largest ρ such that

$$\int_0^T y^T(t)u(t)dt > \rho \int_0^T y^T(t)y(t)dt$$

The system G is "output strictly passive" (OSP) when $\rho > 0$. ρ is also called the "output feedback passivity" (OFP) index and corresponds to the minimum feedback action needed to make the system passive.

- The I/O passivity index is defined as the largest τ such that

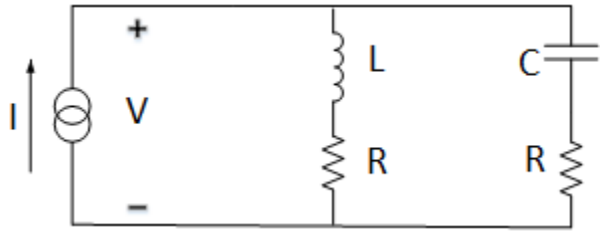
$$\int_0^T y^T(t)u(t)dt > \tau \int_0^T \left(u^T(t)u(t) + y^T(t)y(t)\right)dt$$

The system is "very strictly passive" (VSP) if $\tau > 0$.

Circuit Example

Consider the following example. We take the current I as the input and the voltage V as the output. Based on Kirchhoff's current and voltage law, we obtain the transfer function for $G(s)$,

$$G(s) = \frac{V(s)}{I(s)} = \frac{(Ls + R)(Rs + \frac{1}{C})}{Ls^2 + 2Rs + \frac{1}{C}}.$$



Let $R = 2$, $L = 1$ and $C = 0.1$.

```
R = 2; L = 1; C = 0.1;
s = tf('s');
G = (L*s+R)*(R*s+1/C)/(L*s^2 + 2*R*s+1/C);
```

Use `isPassive` to check whether $G(s)$ is passive.

```
PF = isPassive(G)
```

```
PF = logical
   1
```

Since PF = true, $G(s)$ is passive. Use `getPassiveIndex` to compute the passivity indices of $G(s)$.

```
% Input passivity index
nu = getPassiveIndex(G,'in')
```

```
nu = 2
```

```
% Output passivity index
rho = getPassiveIndex(G,'out')
```

```
rho = 0.2857
```

```
% I/O passivity index
tau = getPassiveIndex(G,'io')
```

```
tau = 0.2642
```

Since $\tau > 0$, the system $G(s)$ is very strictly passive.

Frequency-Domain Characterization

A linear system is passive if and only if it is "positive real":

$$G(j\omega) + G^H(j\omega) > 0 \quad \forall \omega \in \mathbf{R}.$$

The smallest eigenvalue of the left-hand-side is related to the input passivity index ν:

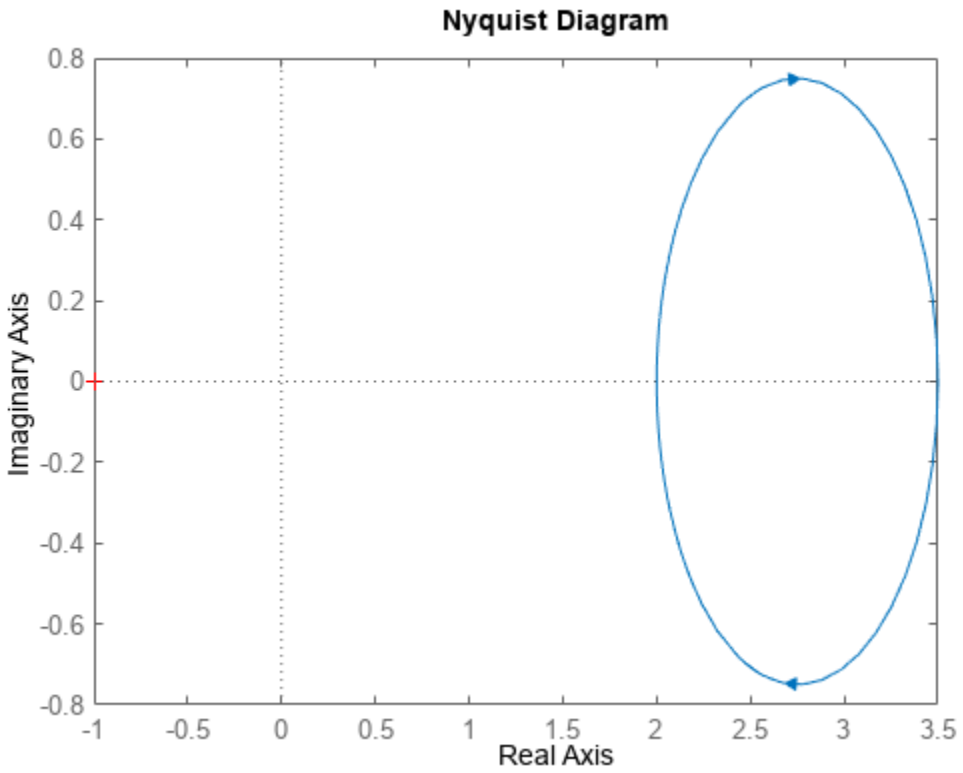
$$\nu = \frac{1}{2}\min_{\omega}\lambda_{\min}\left(G(j\omega) + G^H(j\omega)\right)$$

where $\lambda_{\min}$ denotes the smallest eigenvalue. Similarly, when $G(s)$ is minimum-phase, the output passivity index is given by:

$$\rho = \frac{1}{2}\min_{\omega}\lambda_{\min}\left(G^{-1}(j\omega) + G^{-H}(j\omega)\right).$$

Verify this for the circuit example. Plot the Nyquist plot of the circuit transfer function.

```
nyquist(G)
```



The entire Nyquist plot lies in the right-half plane so $G(s)$ is positive real. The leftmost point on the Nyquist curve is $(x, y) = (2, 0)$ so the input passivity index is $\nu = 2$, the same value we obtained earlier. Similarly, the leftmost point on the Nyquist curve for $G^{-1}(s)$ gives the output passivity index value $\rho = 0.286$.

Relative Passivity Index

It can be shown that the "positive real" condition

$$G(j\omega) + G^H(j\omega) > 0 \quad \forall \omega \in \mathbf{R}$$

is equivalent to the small gain condition

$$||(I - G(j\omega))(I + G(j\omega))^{-1}|| < 1 \quad \forall \omega \in \mathbf{R}.$$

The relative passivity index (R-index) is the peak gain over frequency of $(I - G)(I + G)^{-1}$ when $I + G$ is minimum phase, and $+\infty$ otherwise:

$$R = \left\|(I - G)(I + G)^{-1}\right\|_\infty.$$

In the time domain, the R-index is the smallest $r > 0$ such that

$$\int_0^T ||y-u||^2 dt < r^2 \int_0^T ||y+u||^2 dt$$

The system $G(s)$ is passive if and only if $R < 1$, and the smaller R is, the more passive the system is. Use `getPassiveIndex` to compute the R-index for the circuit example.

```
R = getPassiveIndex(G)
```

```
R = 0.5556
```

The resulting R value indicates that the circuit is a very passive system.

References

[1] Xia, M., P. Gahinet, N. Abroug, C. Buhr, and E. Laroche. “Sector Bounds in Stability Analysis and Control Design.” *International Journal of Robust and Nonlinear Control* 30, no. 18 (December 2020): 7857–82. https://doi.org/10.1002/rnc.5236.

See Also

`getPassiveIndex` | `isPassive`

Related Examples

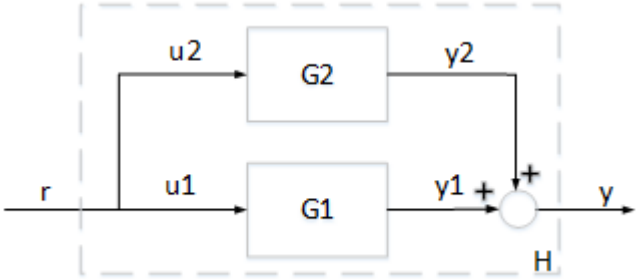
- “About Passivity and Passivity Indices” on page 10-2
- “Parallel Interconnection of Passive Systems” on page 10-18
- “Series Interconnection of Passive Systems” on page 10-20
- “Feedback Interconnection of Passive Systems” on page 10-23

Parallel Interconnection of Passive Systems

This example illustrates the properties of a parallel interconnection of passive systems.

Parallel Interconnection of Passive Systems

Consider an interconnection of two subsystems G_1 and G_2 in parallel. The interconnected system H maps the input r to the output y.



If both systems G_1 and G_2 are passive, then the interconnected system H is guaranteed to be passive. Take for example

$$G_1(s) = \frac{0.1s+1}{s+2}; \quad G_2(s) = \frac{s^2+2s+1}{s^2+3s+10}$$

Both systems are passive.

```
G1 = tf([0.1,1],[1,2]);
isPassive(G1)
```

```
ans = logical
   1
```

```
G2 = tf([1,2,1],[1,3,10]);
isPassive(G2)
```

```
ans = logical
   1
```

We can therefore expect their parallel interconnection H to be passive, as confirmed by

```
H = parallel(G1,G2);
isPassive(H)
```

```
ans = logical
   1
```

Passivity Indices for Parallel Interconnection

There is a relationship between the passivity indices of G_1 and G_2 and the passivity indices of the interconnected system H. Let ν_1 and ν_2 denote the input passivity indices for G_1 and G_2, and let ρ_1

and ρ_2 denote the output passivity indices. If all these indices are nonnegative, then the input passivity index ν and the output passivity index ρ for the parallel interconnection H satisfy

$$\nu \geq \nu_1 + \nu_2, \quad \rho \geq \frac{\rho_1\rho_2}{\rho_1 + \rho_2}.$$

In other words, we can infer some minimum level of input and output passivity for the parallel connection H from the input and output passivity indices of G_1 and G_2. For details, see the paper by Yu, H., "Passivity and dissipativity as design and analysis tools for networked control systems," *Chapter 2*, PhD Thesis, University of Notre Dame, 2012. Verify the lower bound for the input passivity index ν.

```
% Input passivity index for G1
nu1 = getPassiveIndex(G1,'input');
% Input passivity index for G2
nu2 = getPassiveIndex(G2,'input');
% Input passivity index for H
nu = getPassiveIndex(H,'input')
```

```
nu = 0.3777
```

```
% Lower bound
nu1+nu2
```

```
ans = 0.1474
```

Similarly, verify the lower bound for the output passivity index of H.

```
% Output passivity index for G1
rho1 = getPassiveIndex(G1,'output');
% Output passivity index for G2
rho2 = getPassiveIndex(G2,'output');
% Output passivity index for H
rho = getPassiveIndex(H,'output')
```

```
rho = 0.6443
```

```
% Lower bound
rho1*rho2/(rho1+rho2)
```

```
ans = 0.2098
```

See Also

`getPassiveIndex` | `isPassive`

Related Examples

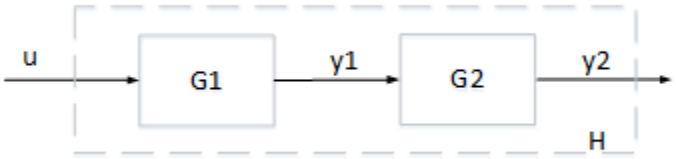
- “About Passivity and Passivity Indices” on page 10-2
- “Series Interconnection of Passive Systems” on page 10-20
- “Feedback Interconnection of Passive Systems” on page 10-23

Series Interconnection of Passive Systems

This example illustrates the properties of a series interconnection of passive systems.

Series Interconnection of Passive Systems

Consider an interconnection of two subsystems G_1 and G_2 in series. The interconnected system H is given by the mapping from input u to output y_2.



In contrast with parallel and feedback interconnections, passivity of the subsystems G_1 and G_2 does not guarantee passivity for the interconnected system H. Take for example

$$G_1(s) = \frac{5s^2 + 3s + 1}{s^2 + 2s + 1}, \quad G_2(s) = \frac{s^2 + s + 5s + 0.1}{s^3 + 2s^2 + 3s + 4}.$$

Both systems are passive as confirmed by

```
G1 = tf([5 3 1],[1,2,1]);
isPassive(G1)
```

```
ans = logical
   1
```

```
G2 = tf([1,1,5,.1],[1,2,3,4]);
isPassive(G2)
```

```
ans = logical
   1
```

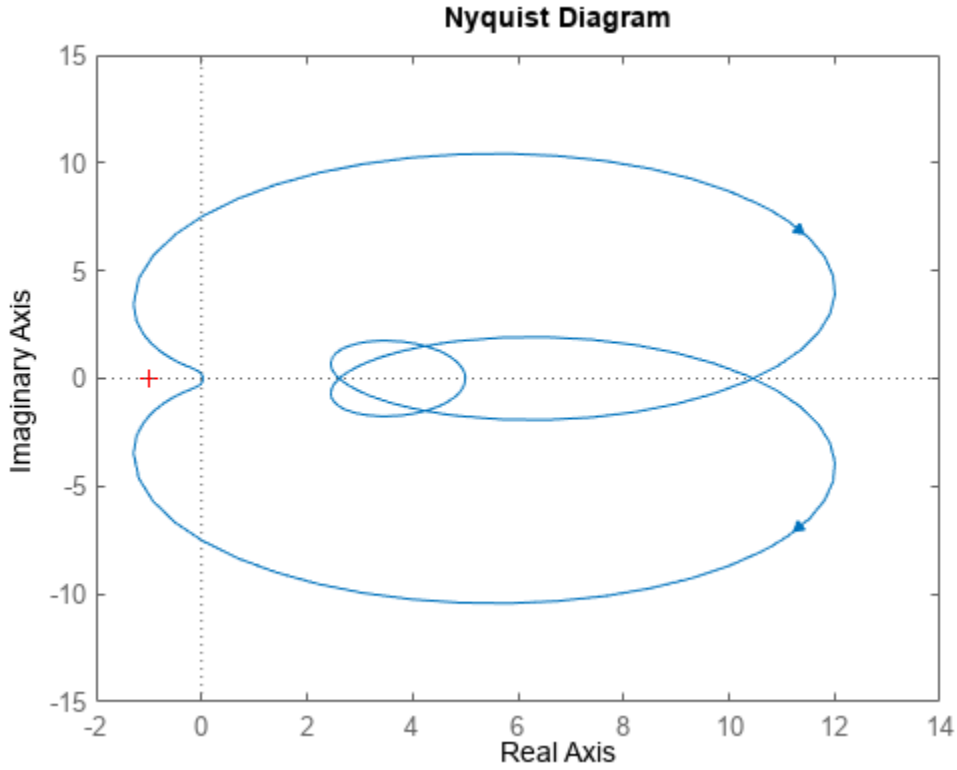
However the series interconnection of G_1 and G_2 is not passive:

```
H = G2*G1;
isPassive(H)
```

```
ans = logical
   0
```

This is confirmed by verifying that the Nyquist plot of G_2G_1 is not positive real.

```
nyquist(H)
```



Passivity Indices for Series Interconnection

While the series interconnection of passive systems is not passive in general, there is a relationship between the passivity indices of G_1 and G_2 and the passivity indices of $H = G_2G_1$. Let ν_1 and ν_2 denote the input passivity indices for G_1 and G_2, and let ρ_1 and ρ_2 denote the output passivity indices. If all these indices are positive, then the input passivity index ν and the output passivity index ρ for the series interconnection H satisfy

$$\nu \geq -\frac{0.125}{\rho_1\rho_2}, \quad \rho \geq -\frac{0.125}{\nu_1\nu_2}.$$

In other words, the shortage of passivity at the inputs or outputs of H is no worse than the right-hand-side expressions. For details, see the paper by Arcak, M. and Sontag, E.D., "Diagonal stability of a class of cyclic systems and its connection with the secant criterion," *Automatica*, Vol 42, No. 9, 2006, pp. 1531-1537. Verify these lower bounds for the example above.

```
% Output passivity index for G1
rho1 = getPassiveIndex(G1,'output');
% Output passivity index for G2
rho2 = getPassiveIndex(G2,'output');
% Input passivity index for H=G2*G1
nu = getPassiveIndex(H,'input')
```

```
nu = -1.2886
```

```
% Lower bound
-0.125/(rho1*rho2)
```

```
ans = -2.4194
```

Similarly, verify the lower bound for the output passivity index of H.

```
% Input passivity index for G1
nu1 = getPassiveIndex(G1,'input');
% Input passivity index for G2
nu2 = getPassiveIndex(G2,'input');
% Output passivity index for H=G2*G1
rho = getPassiveIndex(H,'output')
```

```
rho = -0.6966
```

```
% Lower bound
-0.125/(nu1*nu2)
```

```
ans = -6.0000
```

See Also

`getPassiveIndex` | `isPassive`

Related Examples

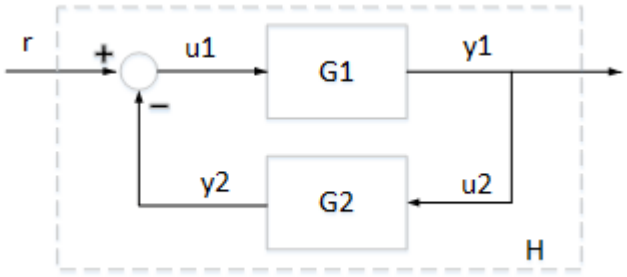
- “About Passivity and Passivity Indices” on page 10-2
- “Parallel Interconnection of Passive Systems” on page 10-18
- “Feedback Interconnection of Passive Systems” on page 10-23

Feedback Interconnection of Passive Systems

This example illustrates the properties of a feedback interconnection of passive systems.

Feedback Interconnection of Passive Systems

Consider an interconnection of two subsystems G_1 and G_2 in feedback. The interconnected system H maps the input r to the output y_1.



If both systems G_1 and G_2 are passive, then the interconnected system H is guaranteed to be passive. Take for example

$$G_1(s) = \frac{s^2 + s + 1}{s^2 + s + 4}, \quad G_2(s) = \frac{s + 2}{s + 5}.$$

Both systems are passive as confirmed by

```
G1 = tf([1,1,1],[1,1,4]);
isPassive(G1)
```

```
ans = logical
   1
```

```
G2 = tf([1,2],[1,5]);
isPassive(G2)
```

```
ans = logical
   1
```

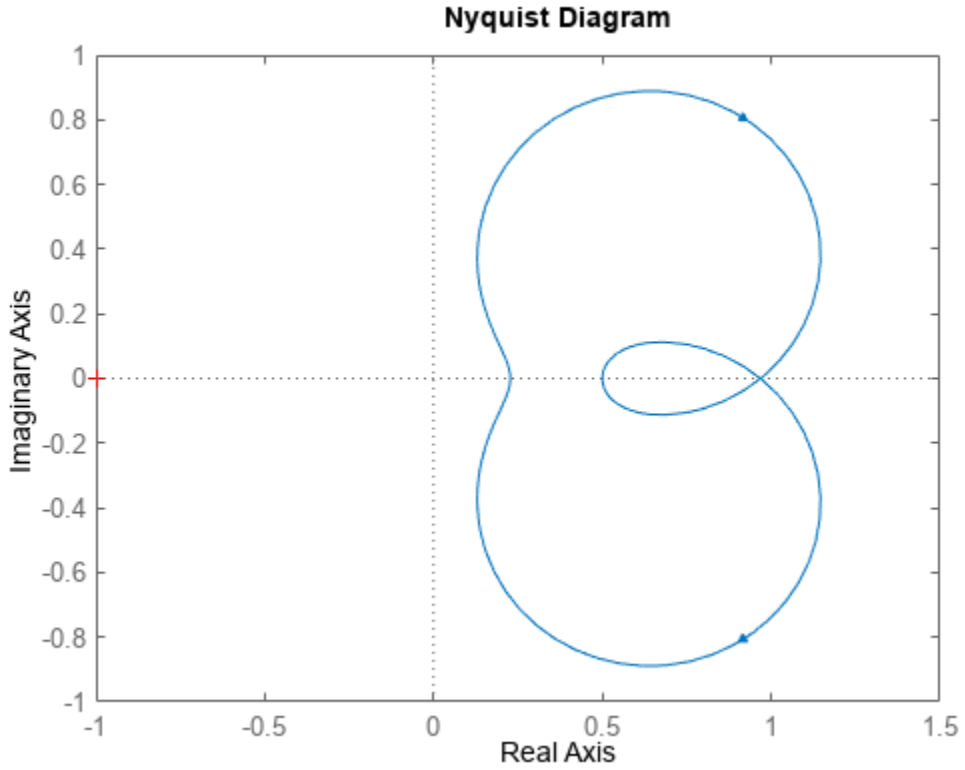
The interconnected system is therefore passive.

```
H = feedback(G1,G2);
isPassive(H)
```

```
ans = logical
   1
```

This is confirmed by verifying that the Nyquist plot of H is positive real.

```
nyquist(H)
```



Passivity Indices for Feedback Interconnection

There is a relationship between the passivity indices of G_1 and G_2 and the passivity indices of the interconnected system H. Let ν_1 and ν_2 denote the input passivity indices for G_1 and G_2, and let ρ_1 and ρ_2 denote the output passivity indices. If all these indices are positive, then the input passivity index ν and the output passivity index ρ for the feedback interconnection H satisfy

$$\nu \geq \frac{\nu_1\rho_2}{\nu_1+\rho_2}, \quad \rho \geq \rho_1+\nu_2\,.$$

In other words, we can infer some minimum level of input and output passivity for the closed-loop system H from the input and output passivity indices of G_1 and G_2. For details, see the paper by Zhu, F. and Xia, M and Antsaklis, P.J., "Passivity analysis and passivation of feedback systems using passivity indices," *American Control Conference* , 2014, pp. 1833-1838. Verify the lower bound for the input passivity index ν.

```
% Input passivity index for G1
nu1 = getPassiveIndex(G1,'input');
% Output passivity index for G2
rho2 = getPassiveIndex(G2,'output');
% Input passivity index for H
nu = getPassiveIndex(H,'input')
```

```
nu = 0.1293
```

```
% Lower bound
nu1*rho2/(nu1+rho2)
```

```
ans = 7.1402e-11
```

Similarly, verify the lower bound for the output passivity index of H.

```
% Output passivity index for G1
rho1 = getPassiveIndex(G1,'output');
% Input passivity index for G2
nu2 = getPassiveIndex(G2,'input');
% Output passivity index for H
rho = getPassiveIndex(H,'output')
```

```
rho = 0.4441
```

```
% Lower bound
rho1+nu2
```

```
ans = 0.4000
```

See Also

`getPassiveIndex` | `isPassive`

Related Examples

- “About Passivity and Passivity Indices” on page 10-2
- “Parallel Interconnection of Passive Systems” on page 10-18
- “Series Interconnection of Passive Systems” on page 10-20
- “Passive Control with Communication Delays” on page 17-34

Control Design

PID Controller Design

- “PID Controller Design at the Command Line” on page 11-2
- “Designing Cascade Control System with PI Controllers” on page 11-7
- “Tune 2-DOF PID Controller (Command Line)” on page 11-11
- “Tune 2-DOF PID Controller (PID Tuner)” on page 11-16
- “PID Controller Types for Tuning” on page 11-26
- “PID Controller Tuning in Simulink” on page 11-33
- “Design PID Controller Using Estimated Frequency Response” on page 11-42
- “Design Family of PID Controllers for Multiple Operating Points” on page 11-50
- “Design PID Controller Using Simulated I/O Data” on page 11-57
- “PID Controller Design in the Live Editor” on page 11-73
- “Tune PID Controller from Measured Plant Data in the Live Editor” on page 11-80
- “Design PID Controller for Disturbance Rejection Using PID Tuner” on page 11-90
- “Temperature Control in a Heat Exchanger” on page 11-101
- “Control of Processes with Long Dead Time: The Smith Predictor” on page 11-115

PID Controller Design at the Command Line

This example shows how to design a PID controller for the plant given by:

$$sys = \frac{1}{(s+1)^3}.$$

As a first pass, create a model of the plant and design a simple PI controller for it.

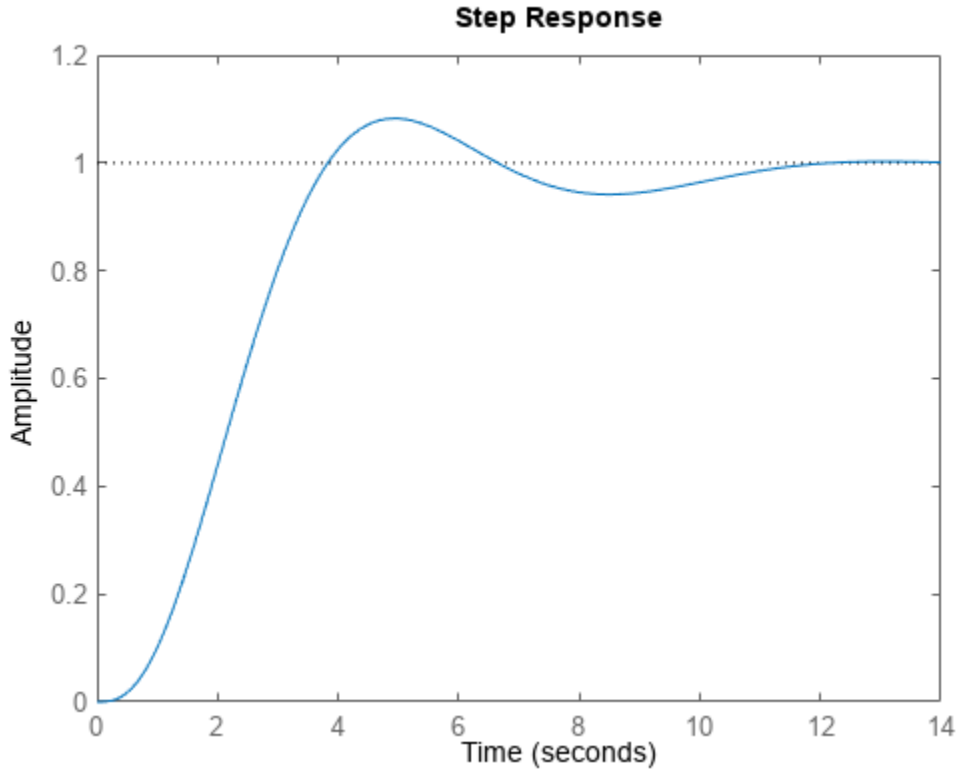
```
sys = zpk([],[-1 -1 -1],1);
[C_pi,info] = pidtune(sys,'PI')
```

```
C_pi =

             1
  Kp + Ki * ---
             s

  with Kp = 1.14, Ki = 0.454

Continuous-time PI controller in parallel form.

info = struct with fields:
                  Stable: 1
    CrossoverFrequency: 0.5205
           PhaseMargin: 60.0000
```

`C_pi` is a `pid` controller object that represents a PI controller. The fields of `info` show that the tuning algorithm chooses an open-loop crossover frequency of about 0.52 rad/s.

Examine the closed-loop step response (reference tracking) of the controlled system.

```
T_pi = feedback(C_pi*sys, 1);
step(T_pi)
```



To improve the response time, you can set a higher target crossover frequency than the result that `pidtune` automatically selects, 0.52. Increase the crossover frequency to 1.0.

```
[C_pi_fast,info] = pidtune(sys,'PI',1.0)
```

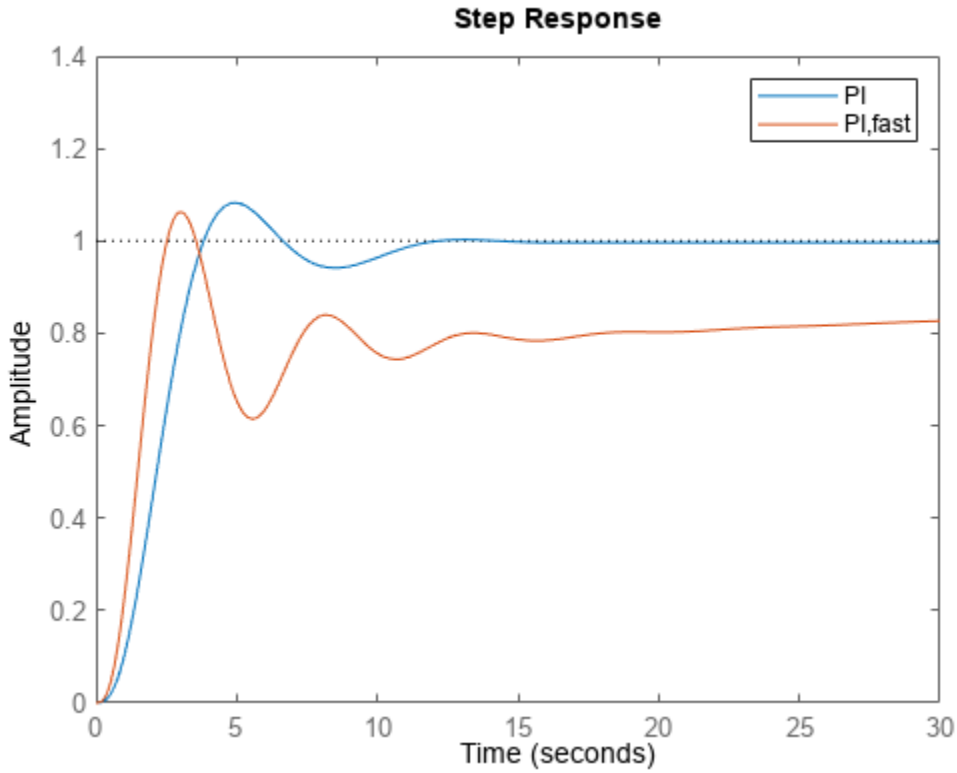
```
C_pi_fast =

             1
  Kp + Ki * ---
             s

  with Kp = 2.83, Ki = 0.0495

Continuous-time PI controller in parallel form.

info = struct with fields:
                 Stable: 1
    CrossoverFrequency: 1
           PhaseMargin: 43.9973
```

The new controller achieves the higher crossover frequency, but at the cost of a reduced phase margin.

Compare the closed-loop step response with the two controllers.

```
T_pi_fast = feedback(C_pi_fast*sys,1);
step(T_pi,T_pi_fast)
```

```
axis([0 30 0 1.4])
legend('PI','PI,fast')
```



This reduction in performance results because the PI controller does not have enough degrees of freedom to achieve a good phase margin at a crossover frequency of 1.0 rad/s. Adding a derivative action improves the response.

Design a PIDF controller for `Gc` with the target crossover frequency of 1.0 rad/s.

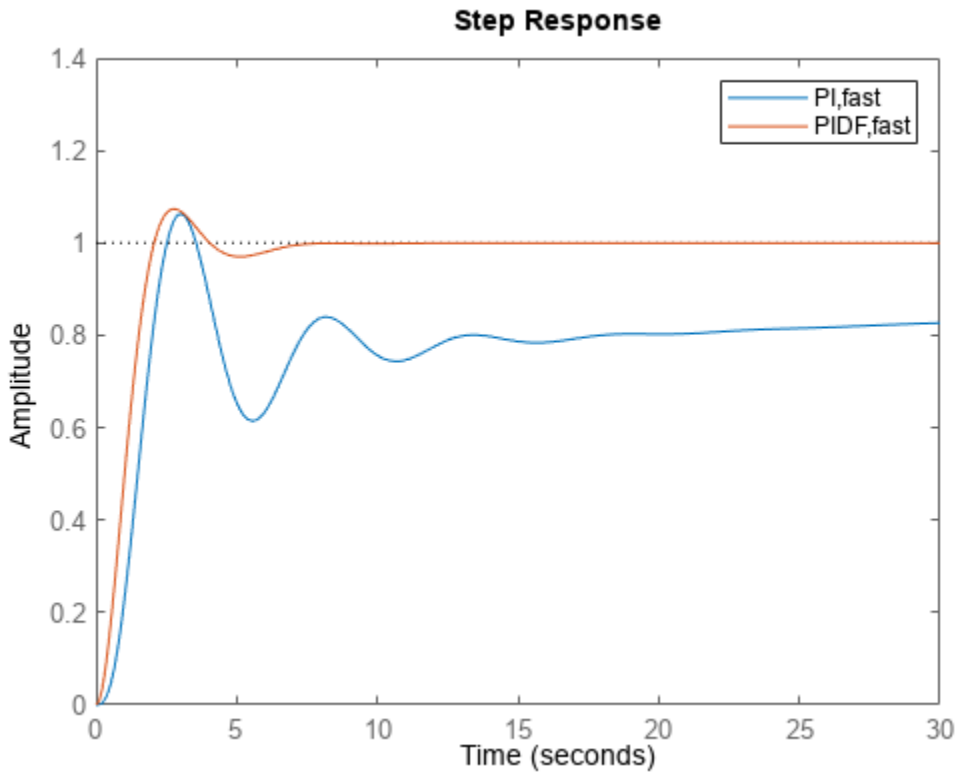
```
[C_pidf_fast,info] = pidtune(sys,'PIDF',1.0)
```

```
C_pidf_fast =

             1             s
  Kp + Ki * --- + Kd * --------
             s          Tf*s+1

  with Kp = 2.72, Ki = 0.985, Kd = 1.72, Tf = 0.00875

Continuous-time PIDF controller in parallel form.

info = struct with fields:
                 Stable: 1
    CrossoverFrequency: 1
           PhaseMargin: 60.0000
```

The fields of info show that the derivative action in the controller allows the tuning algorithm to design a more aggressive controller that achieves the target crossover frequency with a good phase margin.

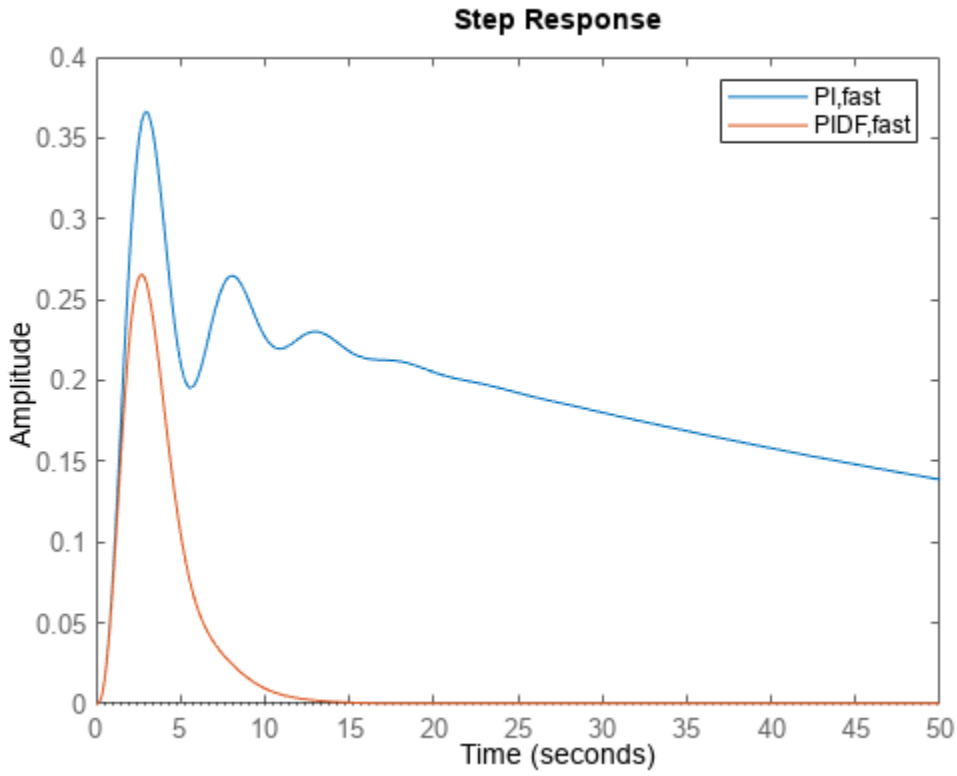
Compare the closed-loop step response and disturbance rejection for the fast PI and PIDF controllers.

```
T_pidf_fast =  feedback(C_pidf_fast*sys,1);
step(T_pi_fast, T_pidf_fast);
axis([0 30 0 1.4]);
legend('PI,fast','PIDF,fast');
```



You can compare the input (load) disturbance rejection of the controlled system with the fast PI and PIDF controllers. To do so, plot the response of the closed-loop transfer function from the plant input to the plant output.

```
S_pi_fast = feedback(sys,C_pi_fast);
S_pidf_fast = feedback(sys,C_pidf_fast);
step(S_pi_fast,S_pidf_fast);
axis([0 50 0 0.4]);
legend('PI,fast','PIDF,fast');
```



This plot shows that the PIDF controller also provides faster disturbance rejection.

See Also

`pidtune` | `pid`

More About

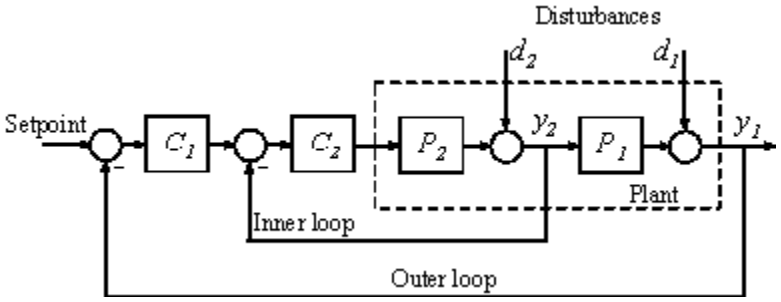
- “Choosing a PID Controller Design Tool”
- “Designing Cascade Control System with PI Controllers” on page 11-7
- “PID Controller Design for Fast Reference Tracking”

Designing Cascade Control System with PI Controllers

This example shows how to design a cascade control loop with two PI controllers using the pidtune command.

Introduction to Cascade Control

Cascade control is mainly used to achieve fast rejection of disturbance before it propagates to the other parts of the plant. The simplest cascade control system involves two control loops (inner and outer) as shown in the block diagram below.



Controller **C1** in the outer loop is the primary controller that regulates the primary controlled variable **y1** by setting the set-point of the inner loop. Controller **C2** in the inner loop is the secondary controller that rejects disturbance **d2** locally before it propagates to **P1**. For a cascade control system to function properly, the inner loop must respond much faster than the outer loop.

In this example, you will design a single loop control system with a PI controller and a cascade control system with two PI controllers. The responses of the two control systems are compared for both reference tracking and disturbance rejection.

Plant

In this example, the inner loop plant P2 is

$$P2(s) = \frac{3}{s+2}$$

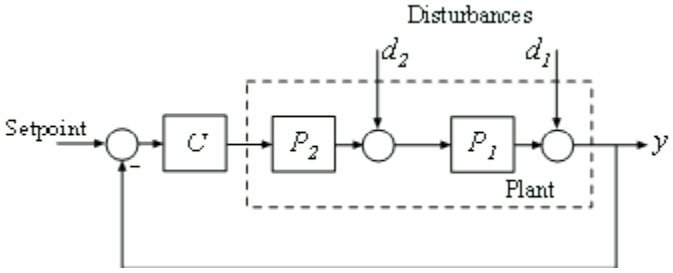
The outer loop plant P1 is

$$P1(s) = \frac{10}{(s+1)^3}$$

```
P2 = zpk([],-2,3);
P1 = zpk([],[-1 -1 -1],10);
```

Designing a Single Loop Control System with a PI Controller

Use **pidtune** command to design a PI controller in standard form for the whole plant model P = P1 * P2.



The desired open loop bandwidth is 0.2 rad/s, which roughly corresponds to the response time of 10 seconds.

```
% The plant model is P = P1*P2
P = P1*P2;
% Use a PID or PIDSTD object to define the desired controller structure
C = pidstd(1,1);
% Tune PI controller for target bandwidth is 0.2 rad/s
C = pidtune(P,C,0.2);
C
```

```
C =

             1      1
  Kp * (1 + ---- * ---)
             Ti     s

  with Kp = 0.0119, Ti = 0.849

Continuous-time PI controller in standard form
```

Designing a Cascade Control System with Two PI Controllers

The best practice is to design the inner loop controller **C2** first and then design the outer loop controller **C1** with the inner loop closed. In this example, the inner loop bandwidth is selected as 2 rad/s, which is ten times higher than the desired outer loop bandwidth. In order to have an effective cascade control system, it is essential that the inner loop responds much faster than the outer loop.

Tune inner-loop controller C2 with open-loop bandwidth at 2 rad/s.

```
C2 = pidtune(P2,pidstd(1,1),2);
C2
```

```
C2 =

             1      1
  Kp * (1 + ---- * ---)
             Ti     s

  with Kp = 0.244, Ti = 0.134

Continuous-time PI controller in standard form
```

Tune outer-loop controller C1 with the same bandwidth as the single loop system.

```
% Inner loop system when the control loop is closed first
clsys = feedback(P2*C2,1);
% Plant seen by the outer loop controller C1 is clsys*P1
```

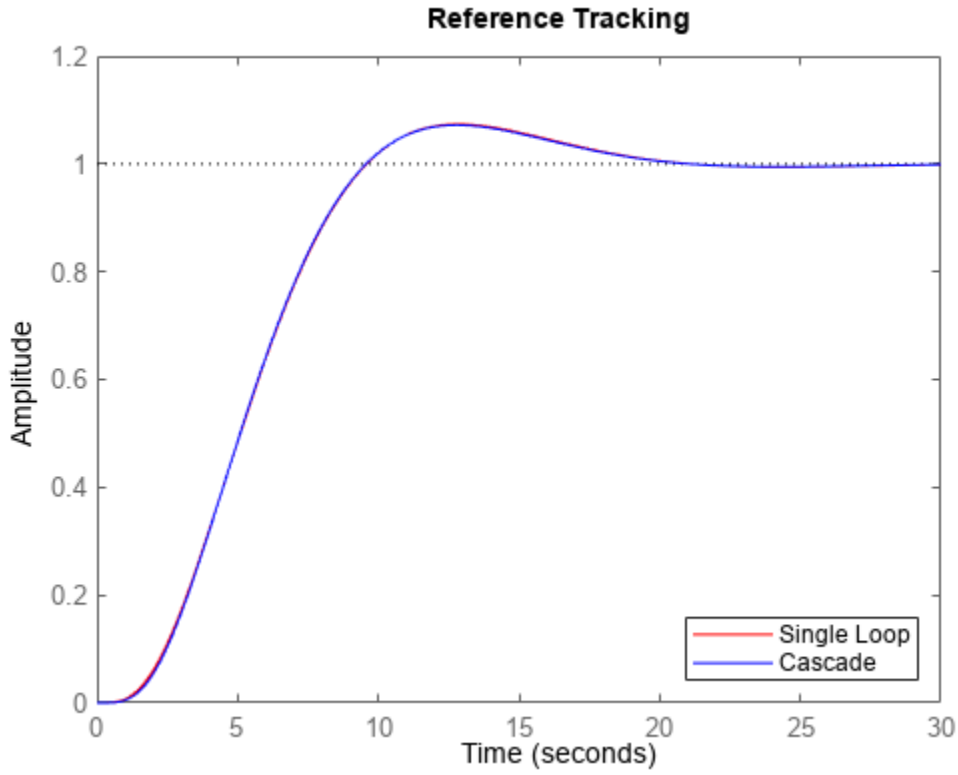
```
C1 = pidtune(clsys*P1,pidstd(1,1),0.2);
C1
```

```
C1 =

             1      1
  Kp * (1 + ---- * ---)
             Ti     s

  with Kp = 0.015, Ti = 0.716

Continuous-time PI controller in standard form
```

Performance Comparison

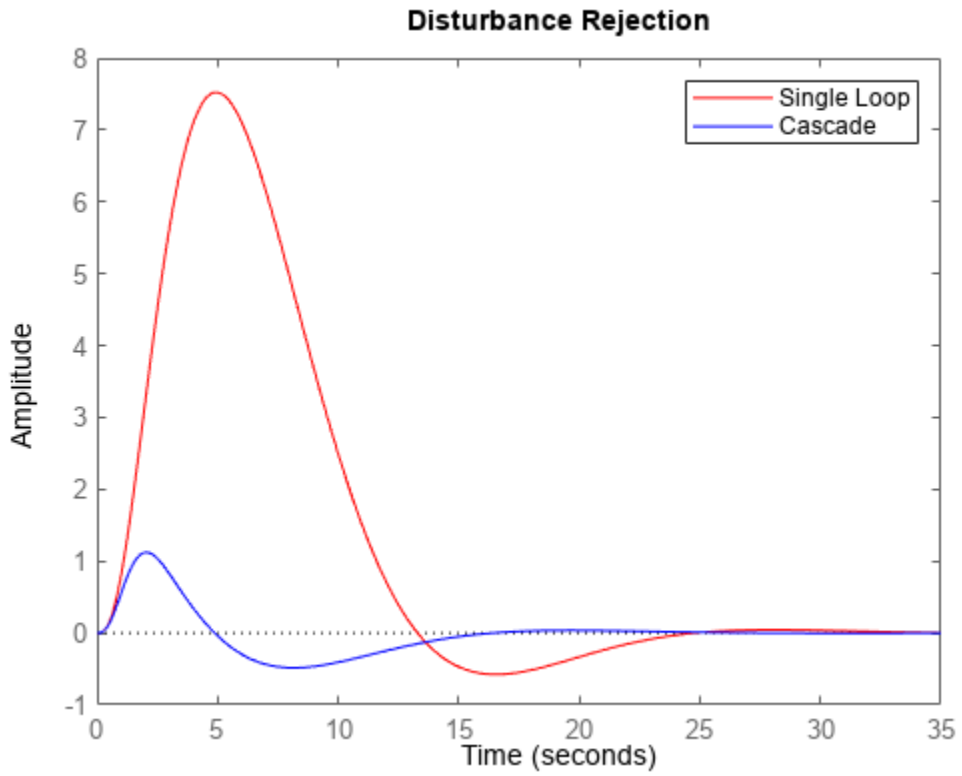
First, plot the step reference tracking responses for both control systems.

```
% single loop system for reference tracking
sys1 = feedback(P*C,1);
sys1.Name = 'Single Loop';
% cascade system for reference tracking
sys2 = feedback(clsys*P1*C1,1);
sys2.Name = 'Cascade';
% plot step response
figure;
step(sys1,'r',sys2,'b')
legend('show','location','southeast')
title('Reference Tracking')
```



Secondly, plot the step disturbance rejection responses of d2 for both control systems.

```
% single loop system for rejecting d2
sysd1 = feedback(P1,P2*C);
sysd1.Name = 'Single Loop';
% cascade system for rejecting d2
sysd2 = P1/(1+P2*C2+P2*P1*C1*C2);
sysd2.Name = 'Cascade';
% plot step response
figure;
step(sysd1,'r',sysd2,'b')
legend('show')
title('Disturbance Rejection')
```



From the two response plots you can conclude that the cascade control system performs much better in rejecting disturbance d2 while the set-point tracking performances are almost identical.

See Also

`pidtune` | `pidstd`

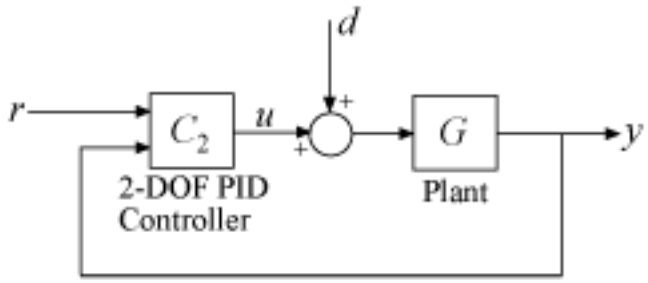
More About

- “Choosing a PID Controller Design Tool”
- “PID Controller Design at the Command Line” on page 11-2
- “Tune PID Controller to Favor Reference Tracking or Disturbance Rejection (Command Line)”

Tune 2-DOF PID Controller (Command Line)

This example shows how to design a two-degree-of-freedom (2-DOF) PID controller at the command line. The example also compares the 2-DOF controller performance to the performance achieved with a 1-DOF PID controller.

2-DOF PID controllers include setpoint weighting on the proportional and derivative terms. Compared to a 1-DOF PID controller, a 2-DOF PID controller can achieve better disturbance rejection without significant increase of overshoot in setpoint tracking. A typical control architecture using a 2-DOF PID controller is shown in the following diagram.



For this example, design a 2-DOF controller for the plant given by:

$$G(s) = \frac{1}{s^2 + 0.5s + 0.1}.$$

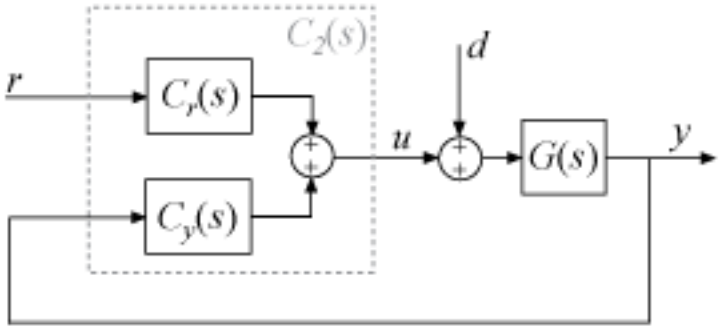
Suppose that your target bandwidth for the system is 1.5 rad/s.

```
wc = 1.5;
G = tf(1,[1 0.5 0.1]);
C2 = pidtune(G,'PID2',wc)
```

```
C2 =

                      1
  u = Kp (b*r-y) + Ki --- (r-y) + Kd*s (c*r-y)
                      s

  with Kp = 1.26, Ki = 0.255, Kd = 1.38, b = 0.665, c = 0

Continuous-time 2-DOF PID controller in parallel form.
```

Using the type `'PID2'` causes `pidtune` to generate a 2-DOF controller, represented as a `pid2` object. The display confirms this result. `pidtune` tunes all controller coefficients, including the setpoint weights `b` and `c`, to balance performance and robustness.

To compute the closed-loop response, note that a 2-DOF PID controller is a 2-input, 1-output dynamic system. You can resolve the controller into two channels, one for the reference signal and one for the feedback signal, as shown in the diagram. (See “Continuous-Time 2-DOF PID Controller Representations” on page 2-13 for more information.)



Decompose the controller into the components `Cr` and `Cy`, and use them to compute the closed-loop response from *r* to *y*.

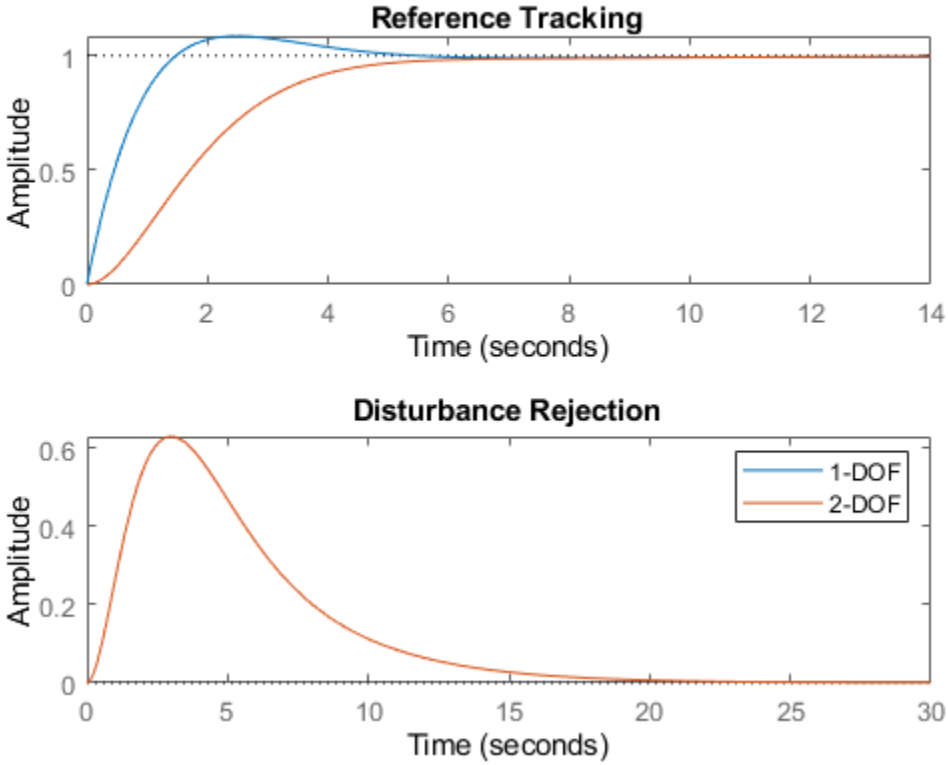
```
C2tf = tf(C2);
Cr = C2tf(1);
Cy = C2tf(2);
T2 = Cr*feedback(G,Cy,+1);
```

To examine the disturbance-rejection performance, compute the transfer function from *d* to *y*.

```
S2 = feedback(G,Cy,+1);
```

For comparison, design a 1-DOF PID controller with the same bandwidth and compute the corresponding transfer functions. Then compare the step responses.

```
C1 = pidtune(G,'PID',wc);
T1 = feedback(G*C1,1);
S1 = feedback(G,C1);

subplot(2,1,1)
stepplot(T1,T2)
title('Reference Tracking')
subplot(2,1,2)
stepplot(S1,S2)
title('Disturbance Rejection')
legend('1-DOF','2-DOF')
```



The plots show that adding the second degree of freedom eliminates the overshoot in the reference-tracking response without any cost to disturbance rejection. You can improve disturbance rejection too using the `DesignFocus` option. This option causes `pidtune` to favor disturbance rejection over setpoint tracking.

```
opt = pidtuneOptions('DesignFocus','disturbance-rejection');
C2dr = pidtune(G,'PID2',wc,opt)
```

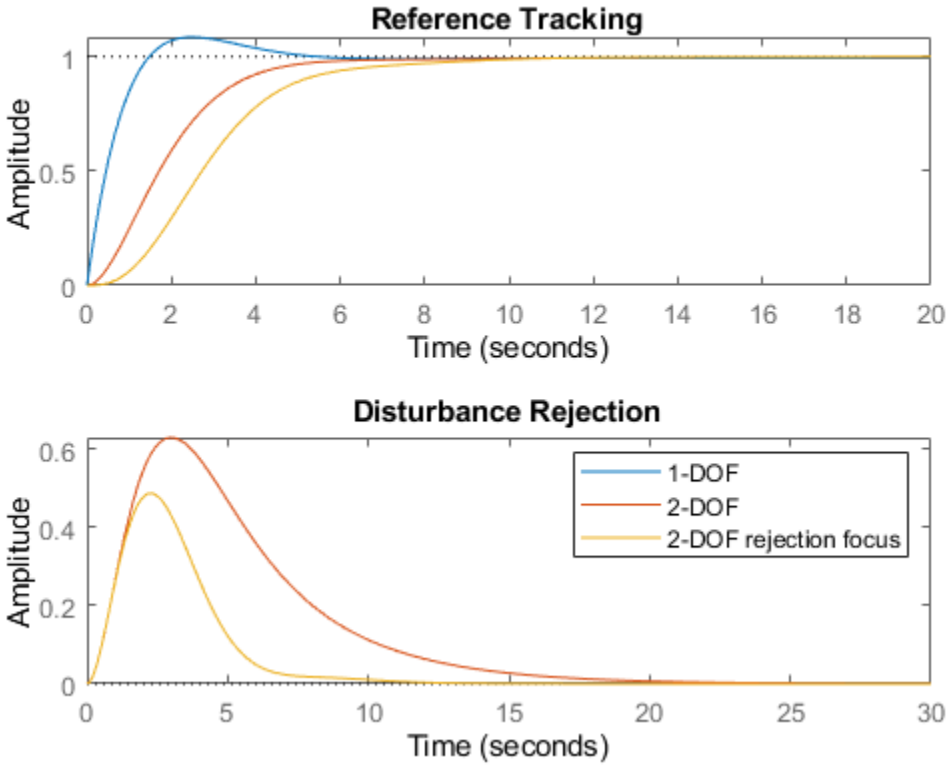
```
C2dr =

                    1
  u = Kp (b*r-y) + Ki --- (r-y) + Kd*s (c*r-y)
                    s

  with Kp = 1.72, Ki = 0.593, Kd = 1.25, b = 0, c = 0

Continuous-time 2-DOF PID controller in parallel form.
```

With the default balanced design focus, `pidtune` selects a `b` value between 0 and 1. For this plant, when you change design focus to favor disturbance rejection, `pidtune` sets `b` = 0 and `c` = 0. Thus, `pidtune` automatically generates an I-PD controller to optimize for disturbance rejection. (Explicitly specifying an I-PD controller without setting the design focus yields a similar controller.)

Compare the closed-loop responses using all three controllers.

```
C2dr_tf = tf(C2dr);
Cdr_r = C2dr_tf(1);
```

```
Cdr_y = C2dr_tf(2);
T2dr = Cdr_r*feedback(G,Cdr_y,+1);
S2dr = feedback(G,Cdr_y,+1);

subplot(2,1,1)
stepplot(T1,T2,T2dr)
title('Reference Tracking')
subplot(2,1,2)
stepplot(S1,S2,S2dr);
title('Disturbance Rejection')
legend('1-DOF','2-DOF','2-DOF rejection focus')
```



The plots show that the disturbance rejection is further improved compared to the balanced 2-DOF controller. This improvement comes with some sacrifice of reference-tracking performance, which is slightly slower. However, the reference-tracking response still has no overshoot.

Thus, using 2-DOF control can improve disturbance rejection without sacrificing as much reference tracking performance as 1-DOF control. These effects on system performance depend strongly on the properties of your plant. For some plants and some control bandwidths, using 2-DOF control or changing the design focus has less or no impact on the tuned result.

See Also

`pidtune` | `pid2`

More About

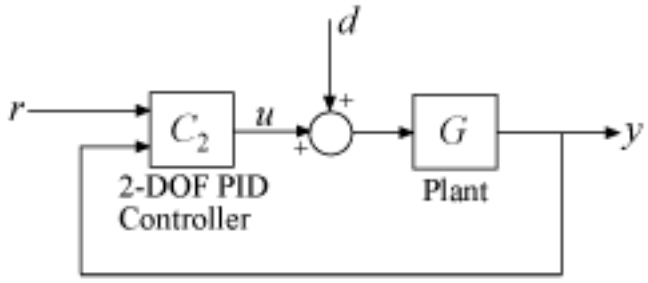
- “Designing PID Controllers with PID Tuner”
- “Two-Degree-of-Freedom PID Controllers” on page 2-13
- “Tune 2-DOF PID Controller (PID Tuner)” on page 11-16
- “Analyze Design in PID Tuner”

Tune 2-DOF PID Controller (PID Tuner)

This example shows how to design a two-degree-of-freedom (2-DOF) PID controller using **PID Tuner**. The example also compares the 2-DOF controller performance to the performance achieved with a 1-DOF PID controller.

In this example, you represent the plant as an LTI model on page 1-10. For information about using **PID Tuner** to tune a PID Controller (2DOF) block in a Simulink model, see “Design Two-Degree-of-Freedom PID Controllers” (Simulink Control Design).

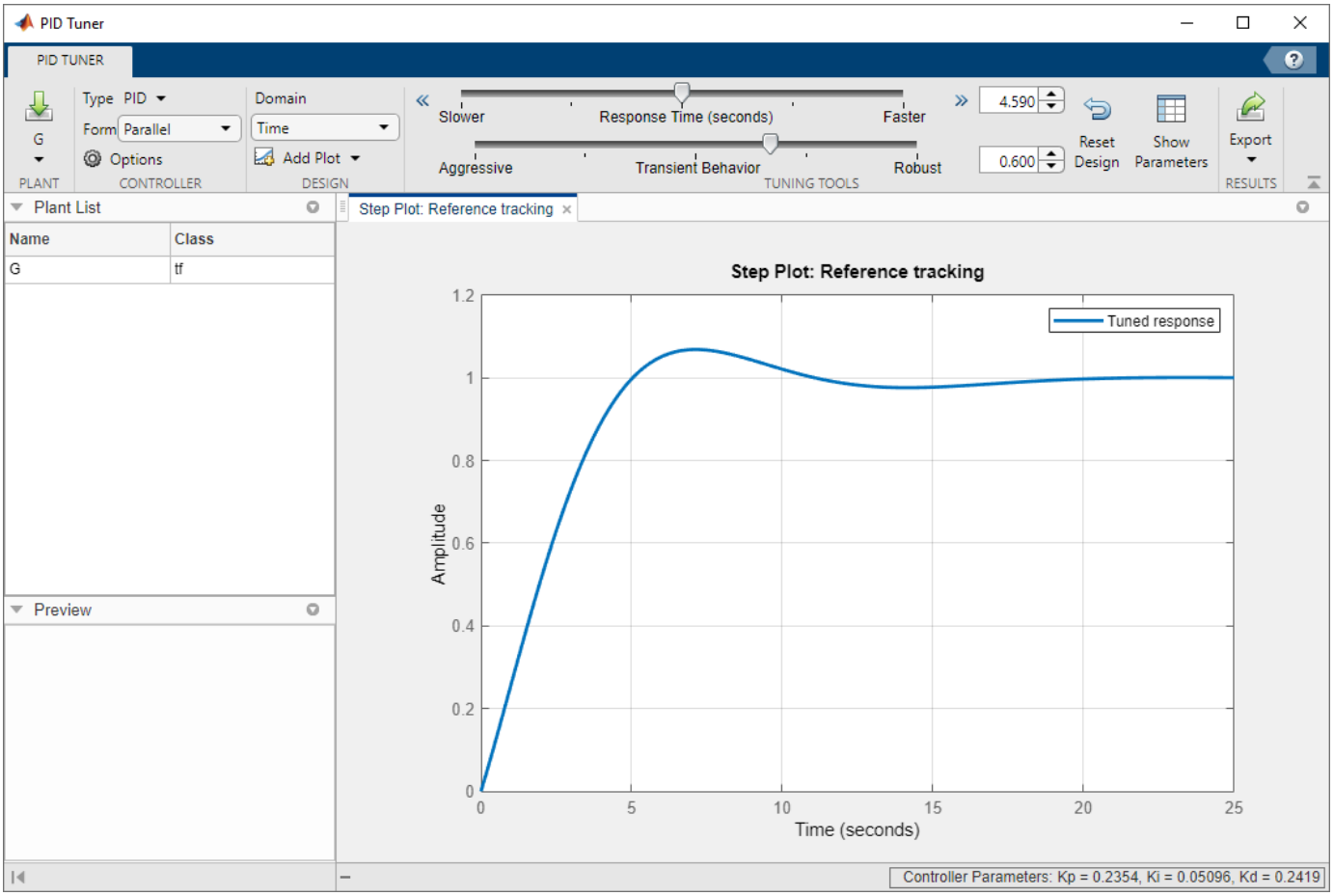
2-DOF PID controllers include setpoint weighting on the proportional and derivative terms. Compared to a 1-DOF PID controller, a 2-DOF PID controller can achieve better disturbance rejection without significant increase of overshoot in setpoint tracking. A typical control architecture using a 2-DOF PID controller is shown in the following diagram.



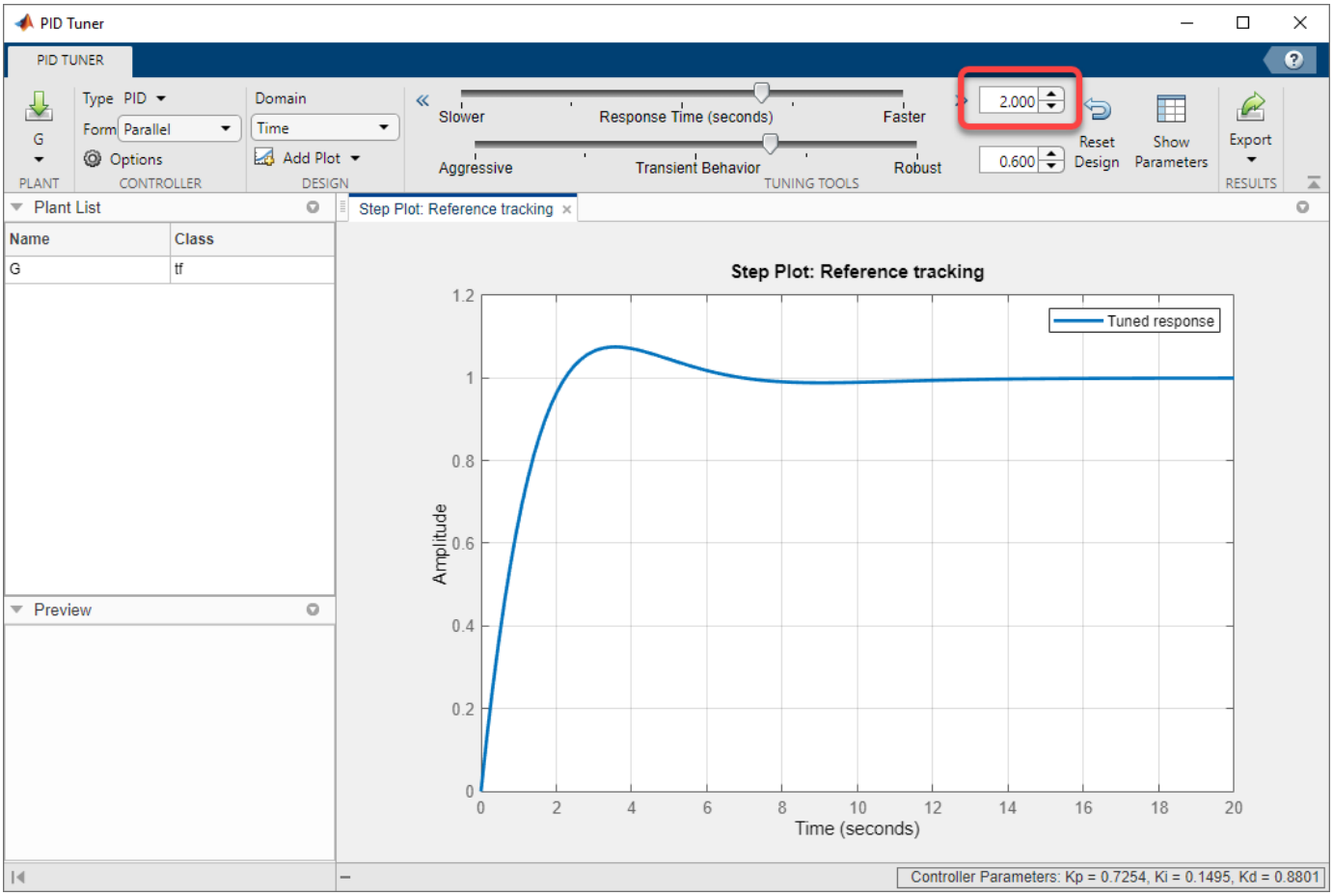
For this example, first design a 1-DOF controller for the plant given by:

$$G(s) = \frac{1}{s^2 + 0.5s + 0.1}.$$

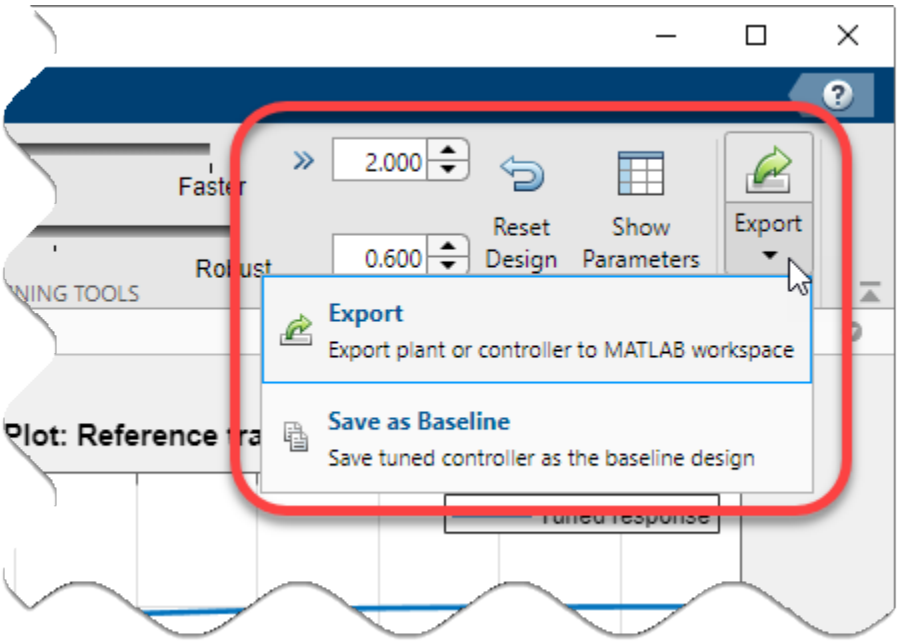
```
G = tf(1,[1 0.5 0.1]);
pidTuner(G,'PID')
```



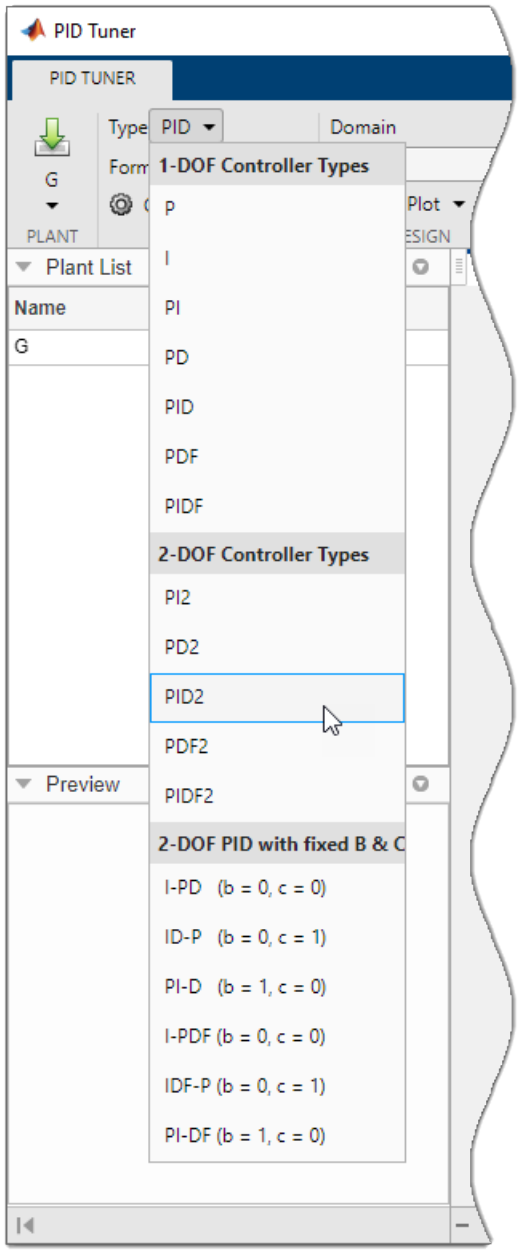
Suppose for this example that your application requires a faster response than the **PID Tuner** initial design. In the text box next to the **Response Time** slider, enter 2.



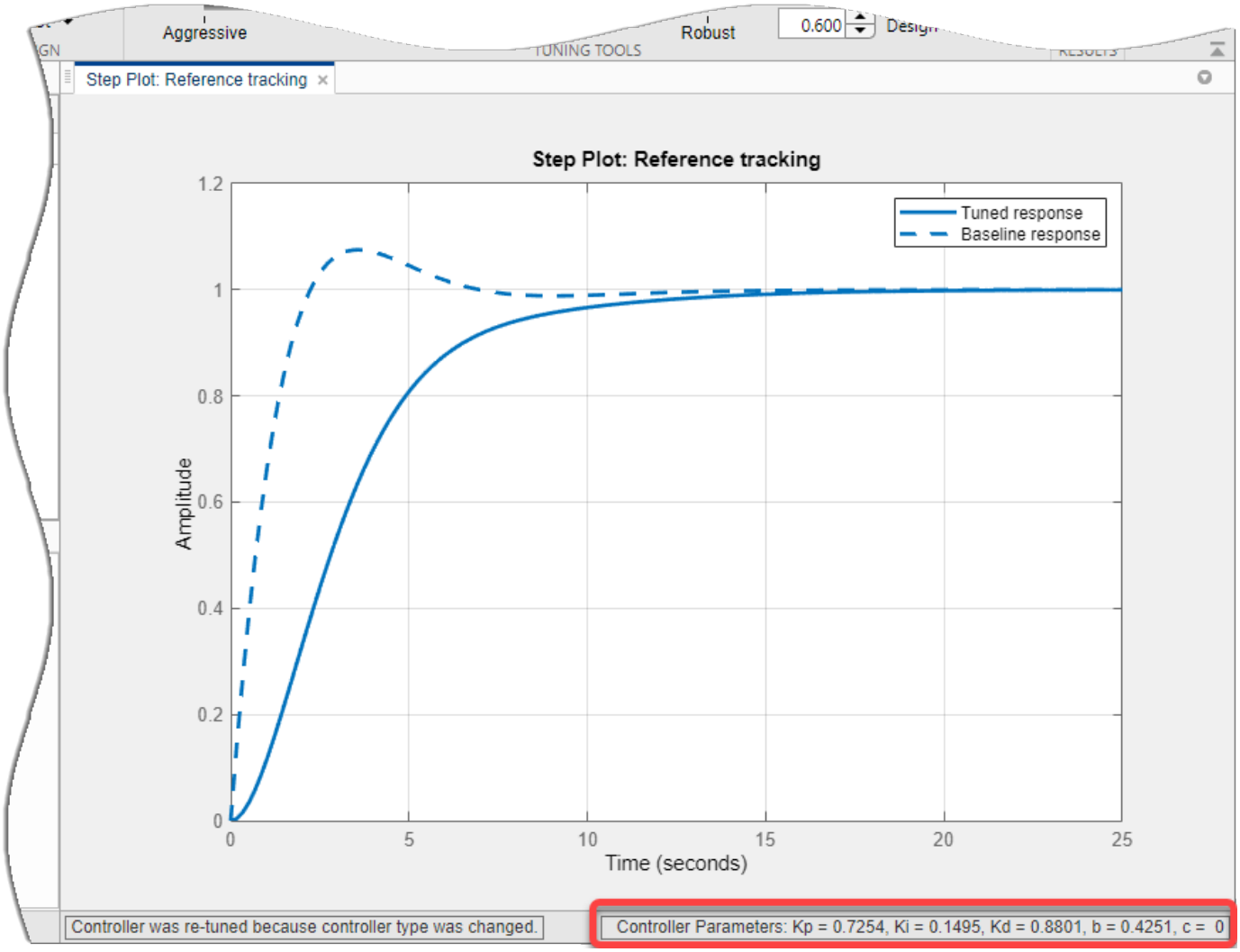
The resulting response is fast, but has a considerable amount of overshoot. Design a 2-DOF controller to improve the overshoot. First, set the 1-DOF controller as the baseline controller for comparison. Click the **Export** arrow and select `Save as Baseline`.



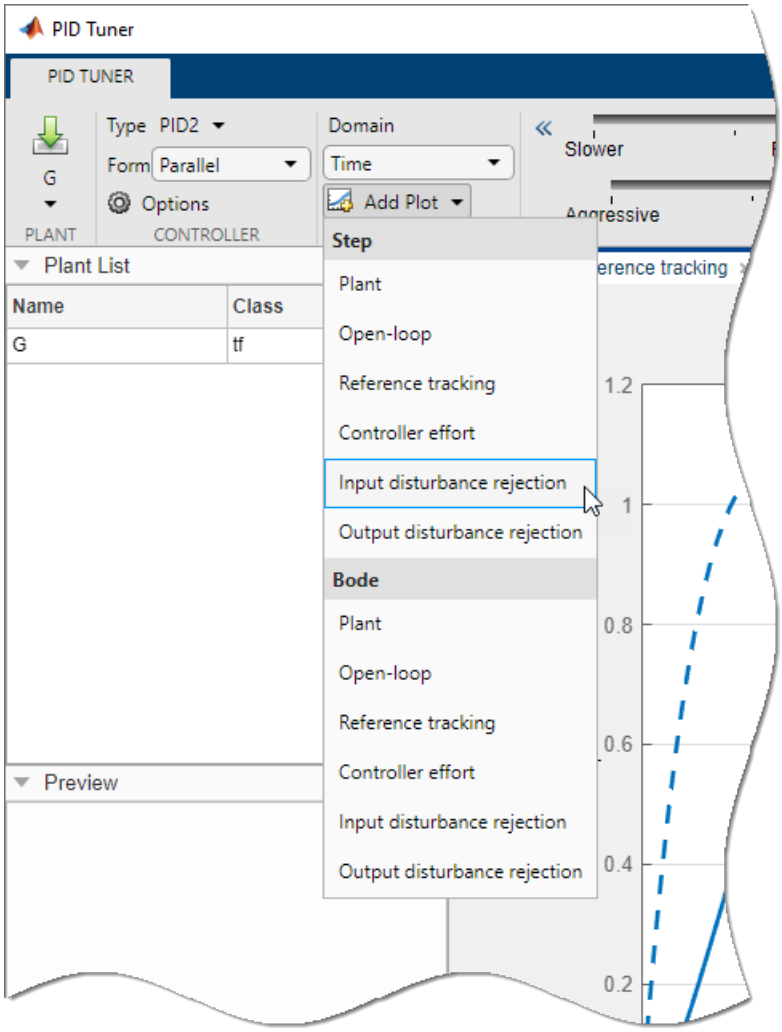
Design the 2-DOF controller. In the **Type** menu, select `PID2`.



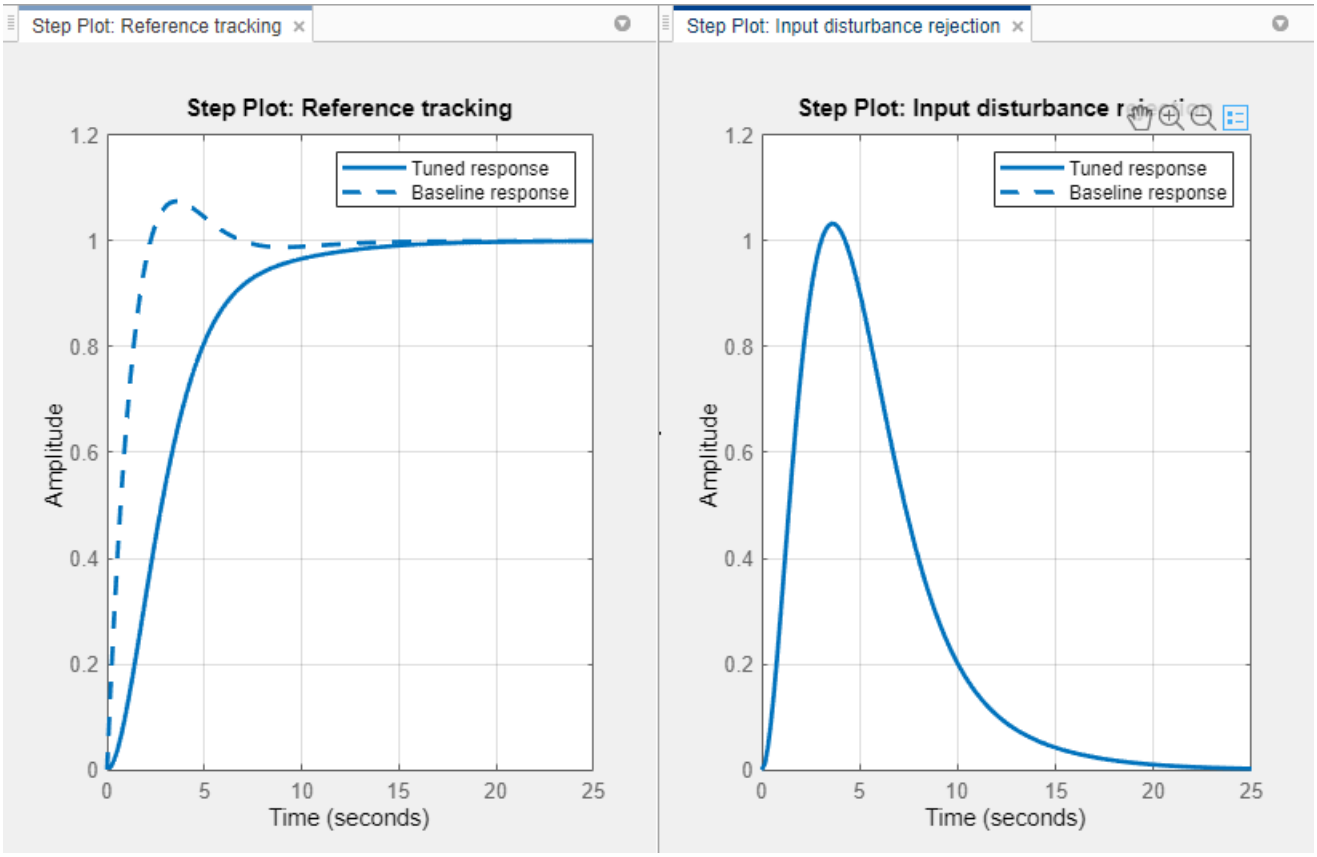
PID Tuner generates a 2-DOF controller with the same target response time. The controller parameters displayed at the bottom right show that **PID Tuner** tunes all controller coefficients, including the setpoint weights `b` and `c`, to balance performance and robustness. Compare the 2-DOF controller performance (solid line) with the performance of the 1-DOF controller that you stored as the baseline (dotted line).



Adding the second degree of freedom eliminates the overshoot in the reference tracking response. Next, add a step response plot to compare the disturbance rejection performance of the two controllers. Select **Add Plot > Input Disturbance Rejection**.



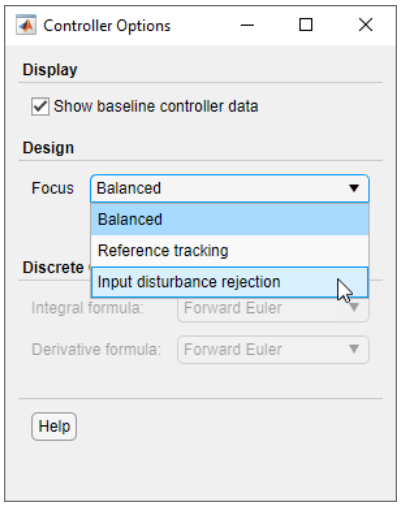
You can move the plots in the **PID Tuner** such that the disturbance-rejection plot side by side with the reference-tracking plot.



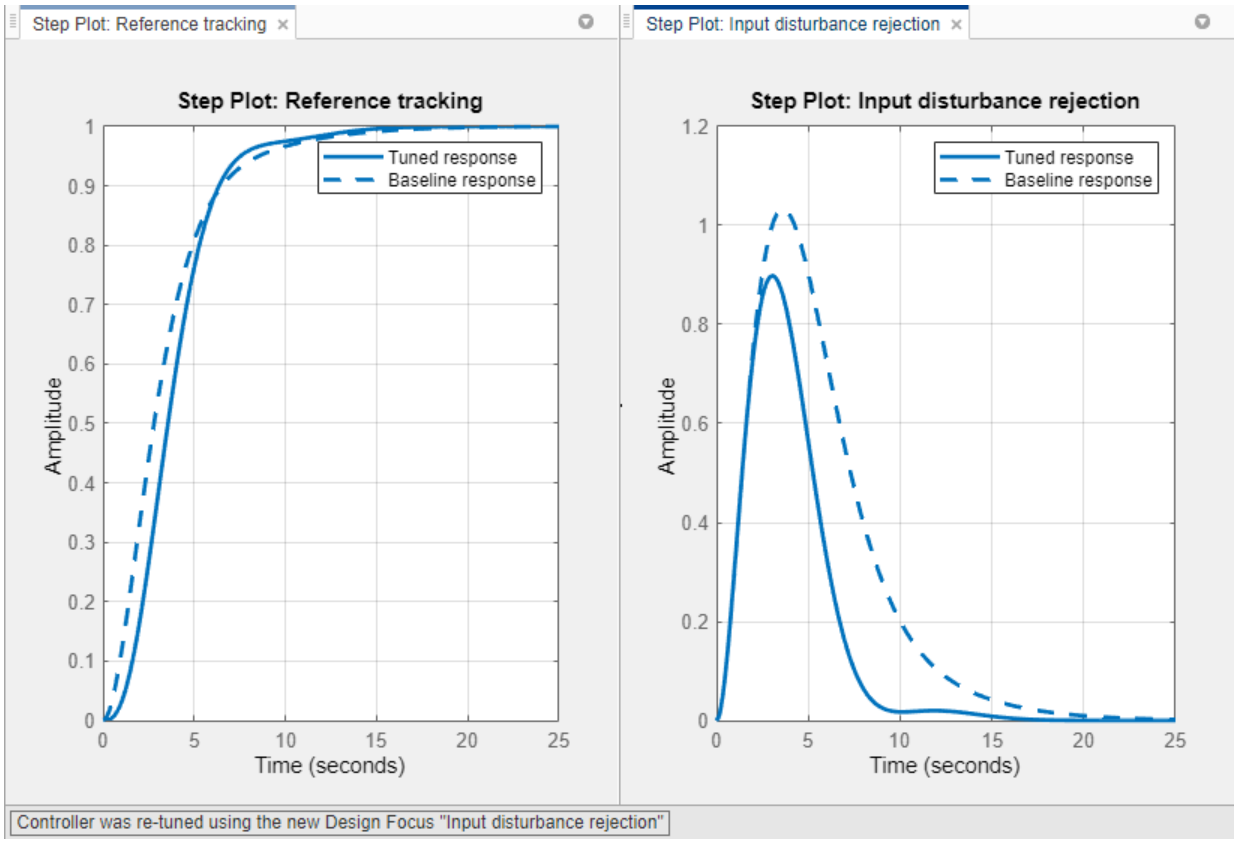
The disturbance-rejection performance is identical with both controllers. Thus, using a 2-DOF controller eliminates reference-tracking overshoot without any cost to disturbance rejection.

You can improve disturbance rejection too by changing the **PID Tuner** design focus. First, click the **Export** arrow and select `Save as Baseline` again to set the 2-DOF controller as the baseline for comparison.

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 `Input disturbance rejection`.



PID Tuner automatically retunes the controller coefficients with a focus on disturbance-rejection performance.



With the default balanced design focus, **PID Tuner** selects a `b` value between 0 and 1. For this plant, when you change design focus to favor disturbance rejection, **PID Tuner** sets `b = 0` and `c = 0`. Thus, **PID Tuner** automatically generates an I-PD controller to optimize for disturbance rejection. (Explicitly specifying an I-PD controller without setting the design focus yields a similar controller.)

The response plots show that with the change in design focus, the disturbance rejection is further improved compared to the balanced 2-DOF controller. This improvement comes with some sacrifice of reference-tracking performance, which is slightly slower. However, the reference-tracking response still has no overshoot.

Thus, using 2-DOF control can improve disturbance rejection without sacrificing as much reference tracking performance as 1-DOF control. These effects on system performance depend strongly on the properties of your plant and the speed of your controller. For some plants and some control bandwidths, using 2-DOF control or changing the design focus has less or no impact on the tuned result.

See Also

`pidTuner`

More About

- “Designing PID Controllers with PID Tuner”
- “Two-Degree-of-Freedom PID Controllers” on page 2-13
- “Tune 2-DOF PID Controller (Command Line)” on page 11-11
- “Analyze Design in PID Tuner”

PID Controller Types for Tuning

Control System Toolbox PID tuning tools can tune many PID and 2-DOF PID controller types. The term controller type refers to which terms are present in the controller action. For example, a PI controller has only a proportional and an integral term, while a PIDF controller contains proportional, integrator, and filtered derivative terms. This topic summarizes the types of PID controllers available for tuning in the following tools:

- **PID Tuner** app
- **Tune PID Controller** task in the Live Editor
- `pidtune` command

Specifying PID Controller Type

The PID tuning tools let you design numerous controller types. How you specify controller type depends on which tool you are using.

Command-Line Tuning

For command-line tuning, provide the `type` argument to the `pidtune` command. For example, `C = pidtune(G,'PI')` tunes a PI controller for plant `G`.

Alternatively, if you provide an existing controller object as the input argument `C0`, `pidtune` tunes a new controller of the same type and form. For example, suppose `C0` is a `pid` controller object that has proportional and derivative action only (PD controller). Then, `pidtune(G,C0)` generates a new `pid` controller object that also has only proportional and derivative action. See `pidtune`.

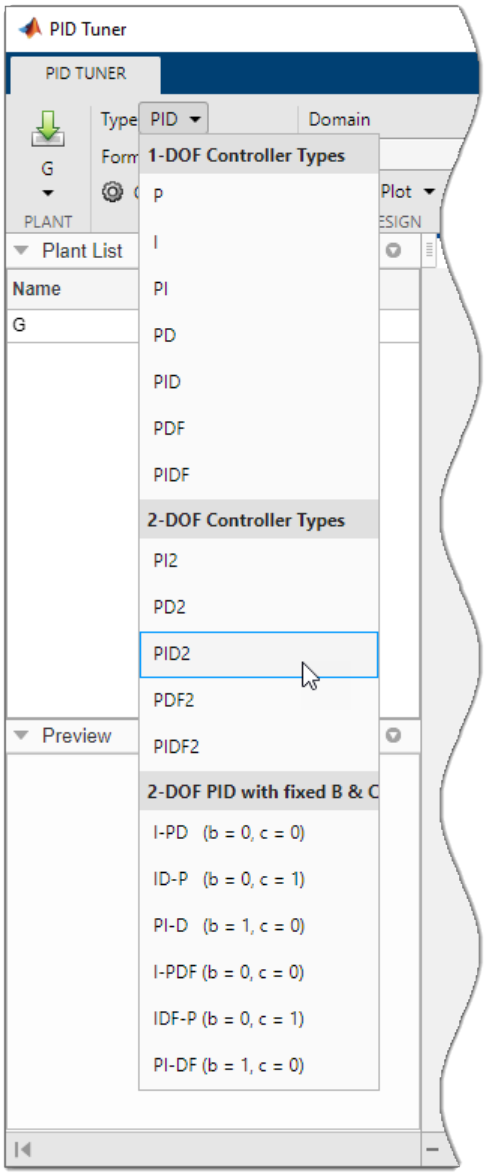
For more about the specific controller types available with command-line tuning, see:

- “1-DOF Controllers” on page 11-28
- “2-DOF Controllers” on page 11-29
- “2-DOF Controllers with Fixed Setpoint Weights” on page 11-30

PID Tuner App

In the **PID Tuner** app, you can specify a controller type when you open the app or change controller type within the app.

- **Specify type when opening the app** — Provide the `type` argument to the `pidTuner` command when you open **PID Tuner**. For example, `pidTuner(G,'PIDF2')` opens **PID Tuner** with an initial design that is a 2-DOF PID controller with a filter on the derivative term.
- **Specify type with an existing controller object** — Provide the baseline-controller `Cbase` argument to the `pidTuner` command when you open **PID Tuner**. **PID Tuner** designs a controller of the same type as `Cbase`. For example, suppose `C0` is a `pid` controller object that has proportional and derivative action only (PD controller). Then, `pidTuner(G,C0)` opens **PID Tuner** with an initial design that is a PD controller.
- **Specify controller type within the app** — In **PID Tuner**, use the **Type** menu to change controller types.

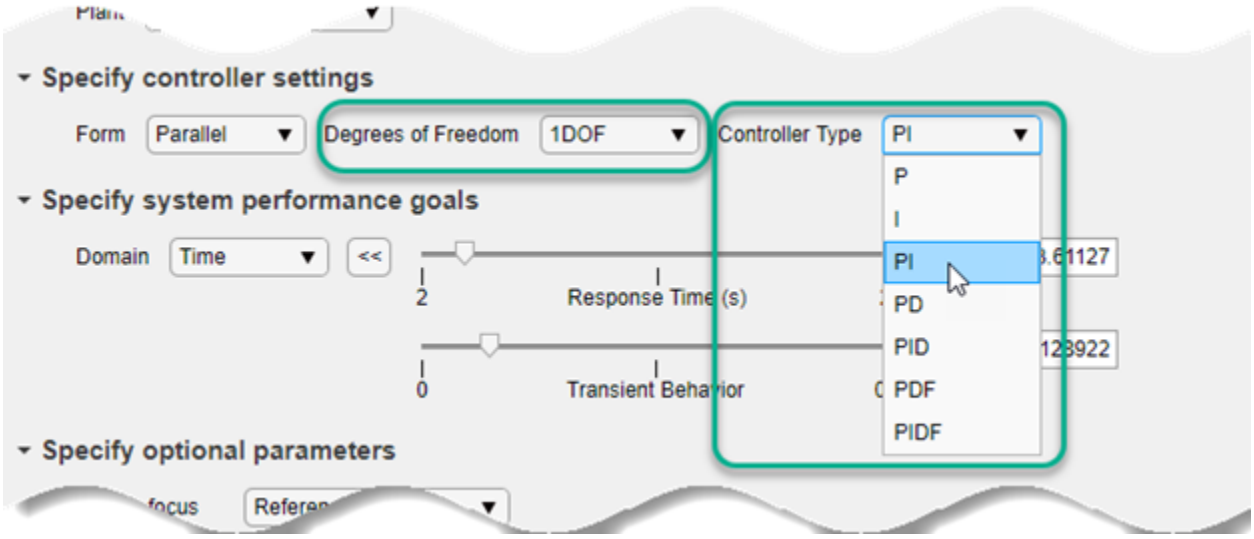


For more about the specific controller types available in the **PID Tuner** app:

- “1-DOF Controllers” on page 11-28
- “2-DOF Controllers” on page 11-29
- “2-DOF Controllers with Fixed Setpoint Weights” on page 11-30

Tune PID Controller Live Editor Task

In the **Tune PID Controller** task in the Live Editor, you specify controller type using the **Degrees of Freedom** and **Controller Type** menus.



For more about the specific controller types available in the **Tune PID Controller** task, see:

- “1-DOF Controllers” on page 11-28
- “2-DOF Controllers” on page 11-29
- “2-DOF Controllers with Fixed Setpoint Weights” on page 11-30

1-DOF Controllers

The following table summarizes the 1-DOF PID controller types available with all tools and provides representative controller formulas for parallel form. The standard-form and discrete-time formulas are analogous.

Type	Controller Actions	Continuous-Time Controller Formula (parallel form)	Discrete-Time Controller Formula (parallel form, ForwardEuler integration method)
`P`	Proportional only	K_p	K_p
`I`	Integral only	$\frac{K_i}{s}$	$K_i\frac{T_s}{z-1}$
`PI`	Proportional and integral	$K_p+\frac{K_i}{s}$	$K_p+K_i\frac{T_s}{z-1}$
`PD`	Proportional and derivative	K_p+K_ds	$K_p+K_d\frac{z-1}{T_s}$
`PDF`	Proportional and derivative with first-order filter on derivative term	$K_p+\frac{K_ds}{T_fs+1}$	$K_p+K_d\frac{1}{T_f+\frac{T_s}{z-1}}$
`PID`	Proportional, integral, and derivative	$K_p+\frac{K_i}{s}+K_ds$	$K_p+K_i\frac{T_s}{z-1}+K_d\frac{z-1}{T_s}$

Type	Controller Actions	Continuous-Time Controller Formula (parallel form)	Discrete-Time Controller Formula (parallel form, ForwardEuler integration method)
`PIDF`	Proportional, integral, and derivative with first-order filter on derivative term	$K_p + \frac{K_i}{s} + \frac{K_d s}{T_f s + 1}$	$K_p + K_i \frac{T_s}{z-1} + K_d \frac{1}{T_f + \frac{T_s}{z-1}}$

2-DOF Controllers

The tuning tools can automatically design 2-DOF PID controller types with free setpoint weights. The following table summarizes the 2-DOF controller types available in all tools and provides representative controller formulas for parallel form. The standard-form formulas are analogous. For more information about 2-DOF PID controllers generally, see “Two-Degree-of-Freedom PID Controllers” on page 2-13.

Type	Controller Actions	Continuous-Time Controller Formula (parallel form)	Discrete-Time Controller Formula (parallel form, ForwardEuler integration method)
`PI2`	2-DOF proportional and integral	$u = K_p(br - y) + \frac{K_i}{s}(r - y)$	$u = K_p(br - y) + K_i \frac{T_s}{z-1}(r - y)$
`PD2`	2-DOF proportional and derivative	$u = K_p(br - y) + K_d s(cr - y)$	$u = K_p(br - y) + K_d \frac{z-1}{T_s}(cr - y)$
`PDF2`	2-DOF proportional and derivative with first-order filter on derivative term	$u = K_p(br - y) + K_d \frac{s}{T_f s + 1}(cr - y)$	$u = K_p(br - y) + K_d \frac{1}{T_f + \frac{T_s}{z-1}}(cr - y)$
`PID2`	2-DOF proportional, integral, and derivative	$u = K_p(br - y) + \frac{K_i}{s}(r - y) + K_d s(cr - y)$	$u = K_p(br - y) + K_i \frac{T_s}{z-1}(r - y) + K_d \frac{z-1}{T_s}(cr - y)$

Type	Controller Actions	Continuous-Time Controller Formula (parallel form)	Discrete-Time Controller Formula (parallel form, ForwardEuler integration method)
`PIDF2`	2-DOF proportional, integral, and derivative with first-order filter on derivative term	$u = K_p(br - y) + \frac{K_i}{s}(r - y) + K_d\frac{s}{T_f s + 1}(cr - y)$	$u = K_p(br - y) + K_i\frac{T_s}{z - 1}(r - y) + K_d\frac{1}{T_f + \frac{T_s}{z - 1}}(cr - y)$

2-DOF Controllers with Fixed Setpoint Weights

With PID control, step changes in the reference signal can cause spikes in the control signal contributed by the proportional and derivative terms. By fixing the setpoint weights of a 2-DOF controller, you can mitigate the influence on the control signal exerted by changes in the reference signal. For example, consider the relationship between the inputs r (setpoint) and y (feedback) and the output u (control signal) of a continuous-time 2-DOF PID controller.

$$u = K_p(br - y) + \frac{K_i}{s}(r - y) + K_d s(cr - y)$$

If you set $b = 0$ and $c = 0$, then changes in the setpoint r do not feed through directly to either the proportional or the derivative terms in u. The $b = 0$, $c = 0$ controller is called an I-PD type controller. I-PD controllers are also useful for improving disturbance rejection.

PID Tuner and `pidtune` can design the fixed-setpoint-weight controller types summarized in the following table. The standard-form and discrete-time formulas are analogous.

Type	Controller Actions	Continuous-Time Controller Formula (parallel form)	Discrete-Time Controller Formula (parallel form, ForwardEuler integration method)
`I-PD`	2-DOF PID with $b = 0$, $c = 0$	$u = -K_p y + \frac{K_i}{s}(r - y) - K_d s y$	$u = -K_p y + K_i\frac{T_s}{z - 1}(r - y) - K_d\frac{z - 1}{T_s}y$

Type	Controller Actions	Continuous-Time Controller Formula (parallel form)	Discrete-Time Controller Formula (parallel form, ForwardEuler integration method)
`I-PDF`	2-DOF PIDF with $b = 0$, $c = 0$	$u = -K_p y + \frac{K_i}{s}(r-y) - K_d \frac{s}{T_f s + 1} y$	$u = -K_p y + K_i \frac{T_s}{z-1}(r-y) - K_d \frac{1}{T_f + \frac{T_s}{z-1}} y$
`ID-P`	2-DOF PID with $b = 0$, $c = 1$	$u = -K_p y + \frac{K_i}{s}(r-y) + K_d s(r-y)$	$u = -K_p y + K_i \frac{T_s}{z-1}(r-y) + K_d \frac{z-1}{T_s}(r-y)$
`IDF-P`	2-DOF PIDF with $b = 0$, $c = 1$	$u = -K_p y + \frac{K_i}{s}(r-y) + K_d \frac{s}{T_f s + 1}(r-y)$	$u = -K_p y + K_i \frac{T_s}{z-1}(r-y) + K_d \frac{1}{T_f + \frac{T_s}{z-1}} (r-y)$
`PI-D`	2-DOF PID with $b = 1$, $c = 0$	$u = K_p(r-y) + \frac{K_i}{s}(r-y) - K_d s y$	$u = K_p(r-y) + K_i \frac{T_s}{z-1}(r-y) - K_d \frac{z-1}{T_s} y$
`PI-DF`	2-DOF PIDF with $b = 1$, $c = 0$	$u = K_p(r-y) + \frac{K_i}{s}(r-y) - K_d \frac{s}{T_f s + 1} y$	$u = K_p(r-y) + K_i \frac{T_s}{z-1}(r-y) - K_d \frac{1}{T_f + \frac{T_s}{z-1}} y$

See Also

Functions
`pidtune` | `pidTuner`

Live Editor Tasks
Tune PID Controller

More About

- “Designing PID Controllers with PID Tuner”
- “Proportional-Integral-Derivative (PID) Controllers” on page 2-11
- “Two-Degree-of-Freedom PID Controllers” on page 2-13
- “PID Controller Design at the Command Line” on page 11-2
- “PID Controller Design for Fast Reference Tracking”
- “Tune 2-DOF PID Controller (Command Line)” on page 11-11
- “Tune 2-DOF PID Controller (PID Tuner)” on page 11-16

PID Controller Tuning in Simulink

This example shows how to automatically tune a PID Controller block using **PID Tuner**.

Introduction of the PID Tuner

PID Tuner provides a fast and widely applicable single-loop PID tuning method for the Simulink® PID Controller blocks. With this method, you can tune PID controller parameters to achieve a robust design with the desired response time.

A typical design workflow with the **PID Tuner** involves the following tasks:

(1) Launch the **PID Tuner**. When launching, the software automatically computes a linear plant model from the Simulink model and designs an initial controller.

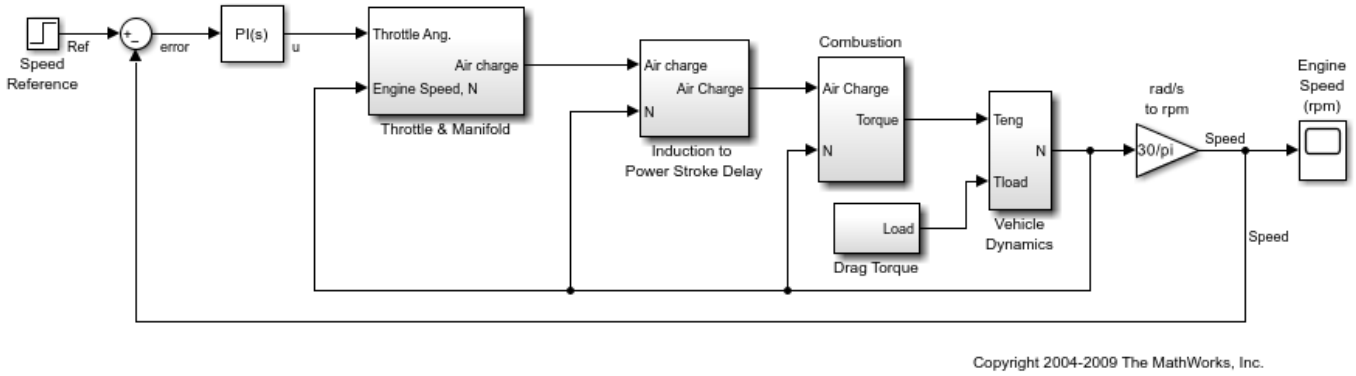
(2) Tune the controller in the **PID Tuner** by manually adjusting design criteria in two design modes. The tuner computes PID parameters that robustly stabilize the system.

(3) Export the parameters of the designed controller back to the PID Controller block and verify controller performance in Simulink.

Open the Model

Open the engine speed control model with PID Controller block and take a few moments to explore it.

```
open_system('scdspeedctrlpidblock')
```



Design Overview

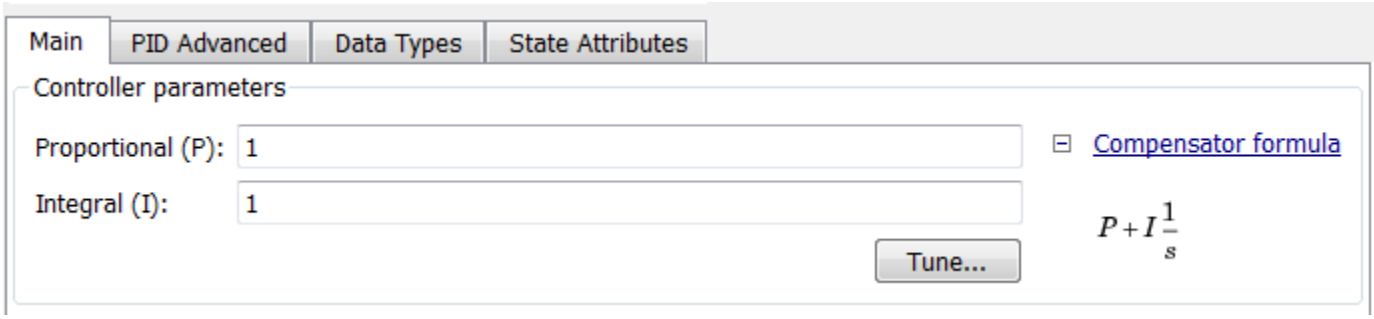
In this example, you design a PI controller in an engine speed control loop. The goal of the design is to track the reference signal from a Simulink step block `scdspeedctrlpidblock/Speed Reference`. The design requirement are:

- Settling time under 5 seconds
- Zero steady-state error to the step reference input.

In this example, you stabilize the feedback loop and achieve good reference tracking performance by designing the PI controller `scdspeedctrl/PID Controller` in the **PID Tuner**.

Open PID Tuner

To launch the **PID Tuner**, double-click the PID Controller block to open its block dialog. In the **Main** tab, click **Tune**.

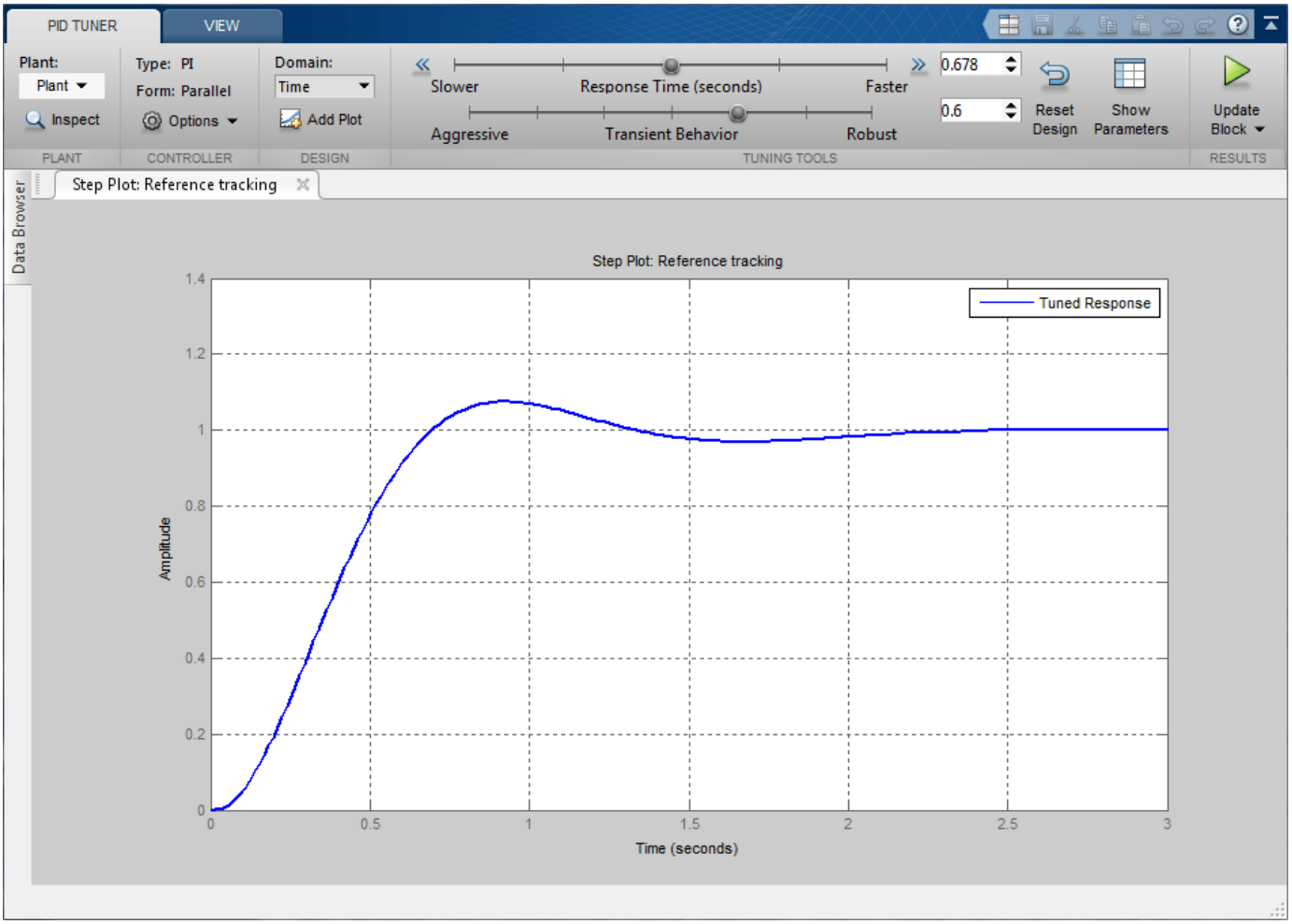


Initial PID Design

When the **PID Tuner** launches, the software computes a linearized plant model seen by the controller. The software automatically identifies the plant input and output, and uses the current operating point for the linearization. The plant can have any order and can have time delays.

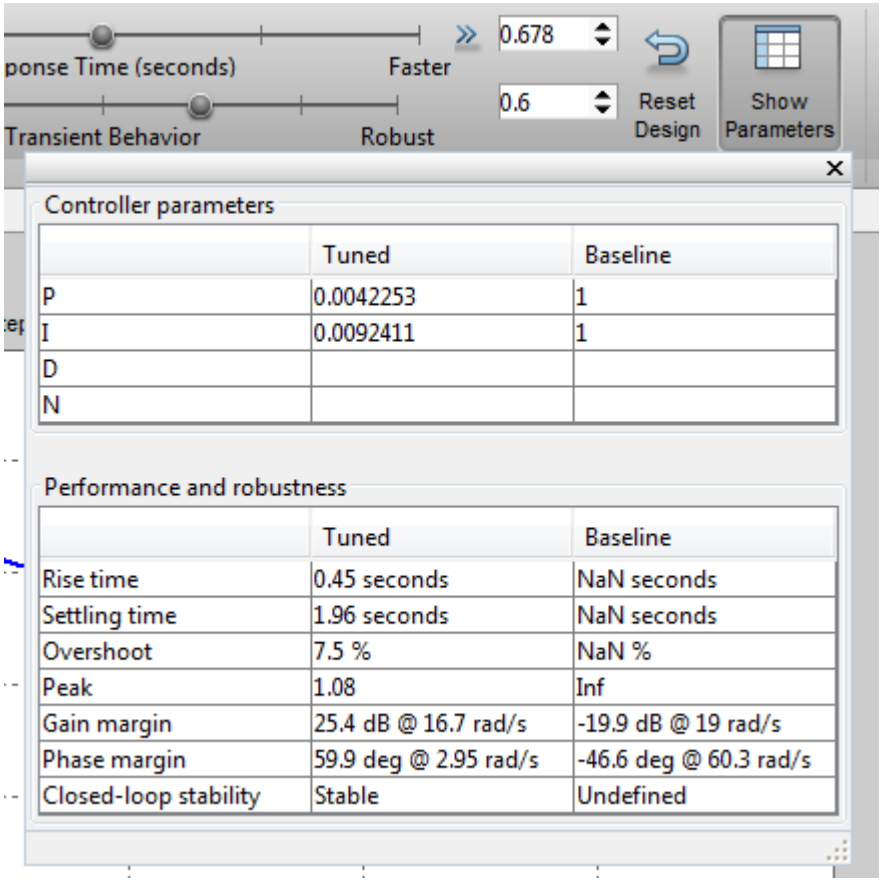
The **PID Tuner** computes an initial PI controller to achieve a reasonable tradeoff between performance and robustness. By default, step reference tracking performance displays in the plot.

The following figure shows the **PID Tuner** dialog with the initial design:



Display PID Parameters

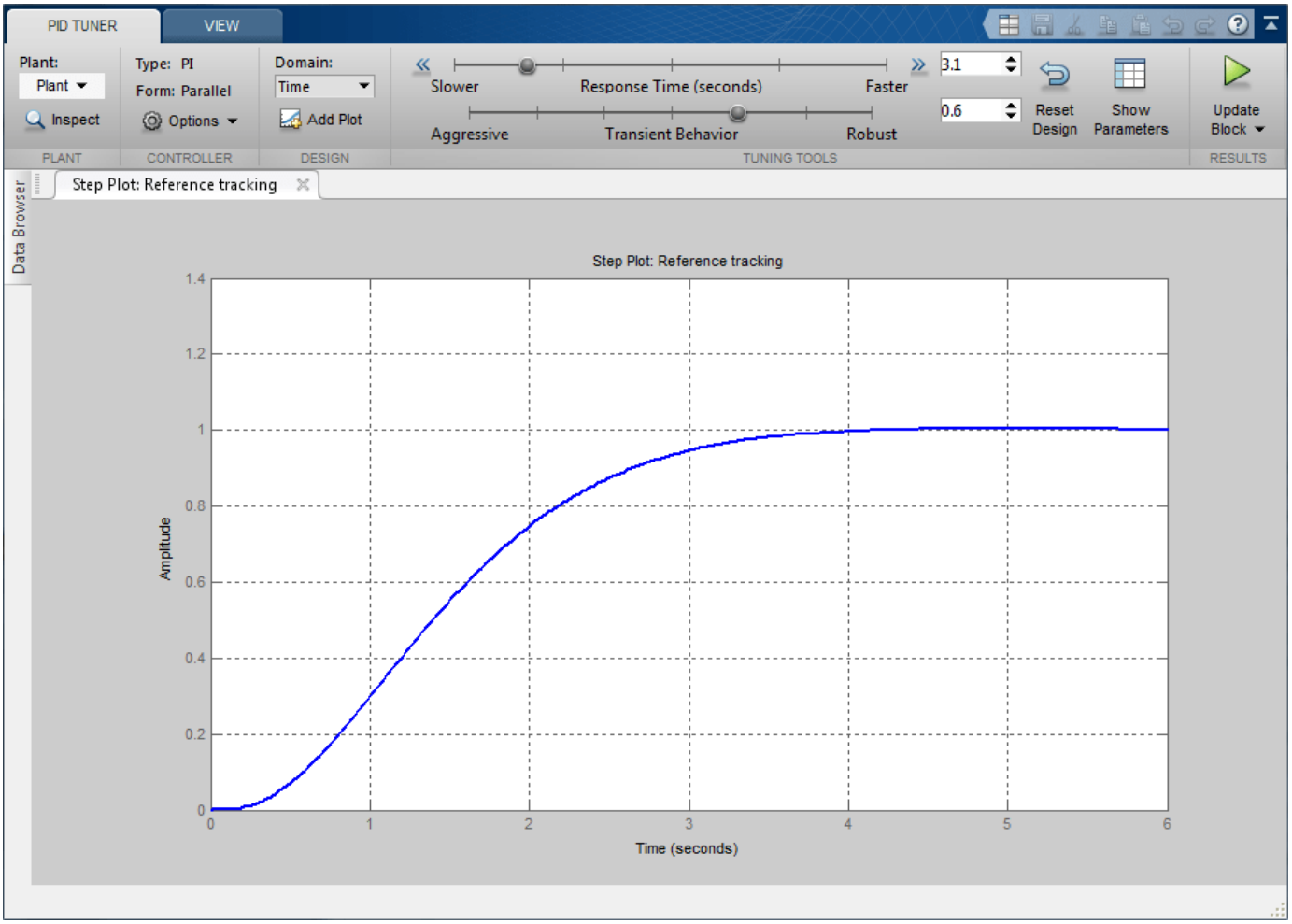
Click **Show parameters** to view controller parameters P and I, and a set of performance and robustness measurements. In this example, the initial PI controller design gives a settling time of 2 seconds, which meets the requirement.

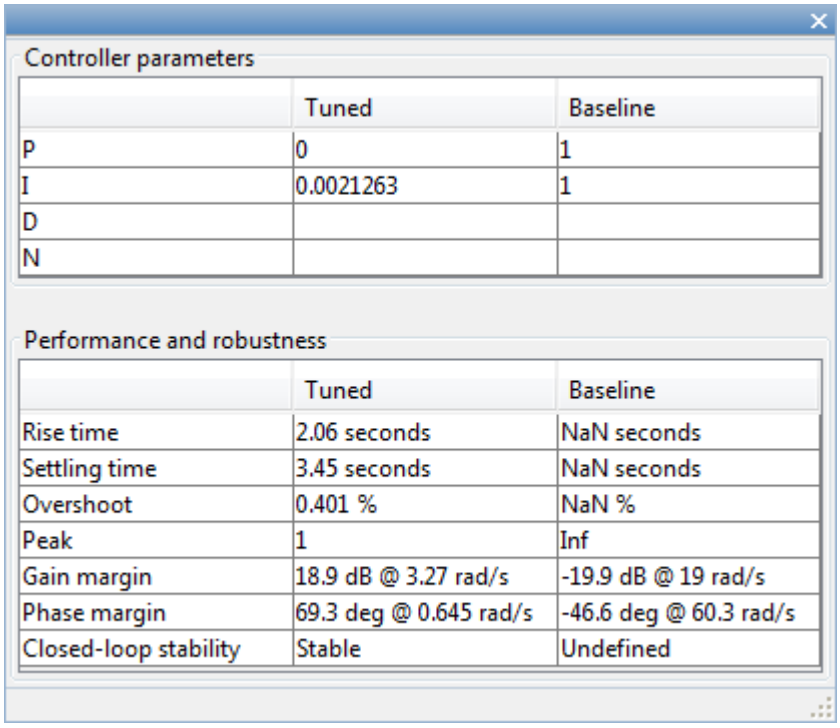


Adjust PID Design in PID Tuner

The overshoot of the reference tracking response is about 7.5 percent. Since we still have some room before reaching the settling time limit, you could reduce the overshoot by increasing the response time. Move the response time slider to the left to increase the closed loop response time. Notice that when you adjust response time, the response plot and the controller parameters and performance measurements update.

The following figure shows an adjusted PID design with an overshoot of zero and a settling time of 4 seconds. The designed controller effectively becomes an integral-only controller.


PID TUNER
VIEW
Plant:
Plant
Inspect
Type: PI
Form: Parallel
Options
Domain:
Time
Add Plot
Slower
Response Time (seconds)
Faster
Aggressive
Transient Behavior
Robust
3.1
0.6
Reset Design
Show Parameters
Update Block
PLANT
CONTROLLER
DESIGN
TUNING TOOLS
RESULTS
Data Browser
Step Plot: Reference tracking
Step Plot: Reference tracking
Amplitude
Time (seconds)
0
0.2
0.4
0.6
0.8
1
1.2
1.4
0
1
2
3
4
5
6



Controller parameters

	Tuned	Baseline
P	0	1
I	0.0021263	1
D		
N		

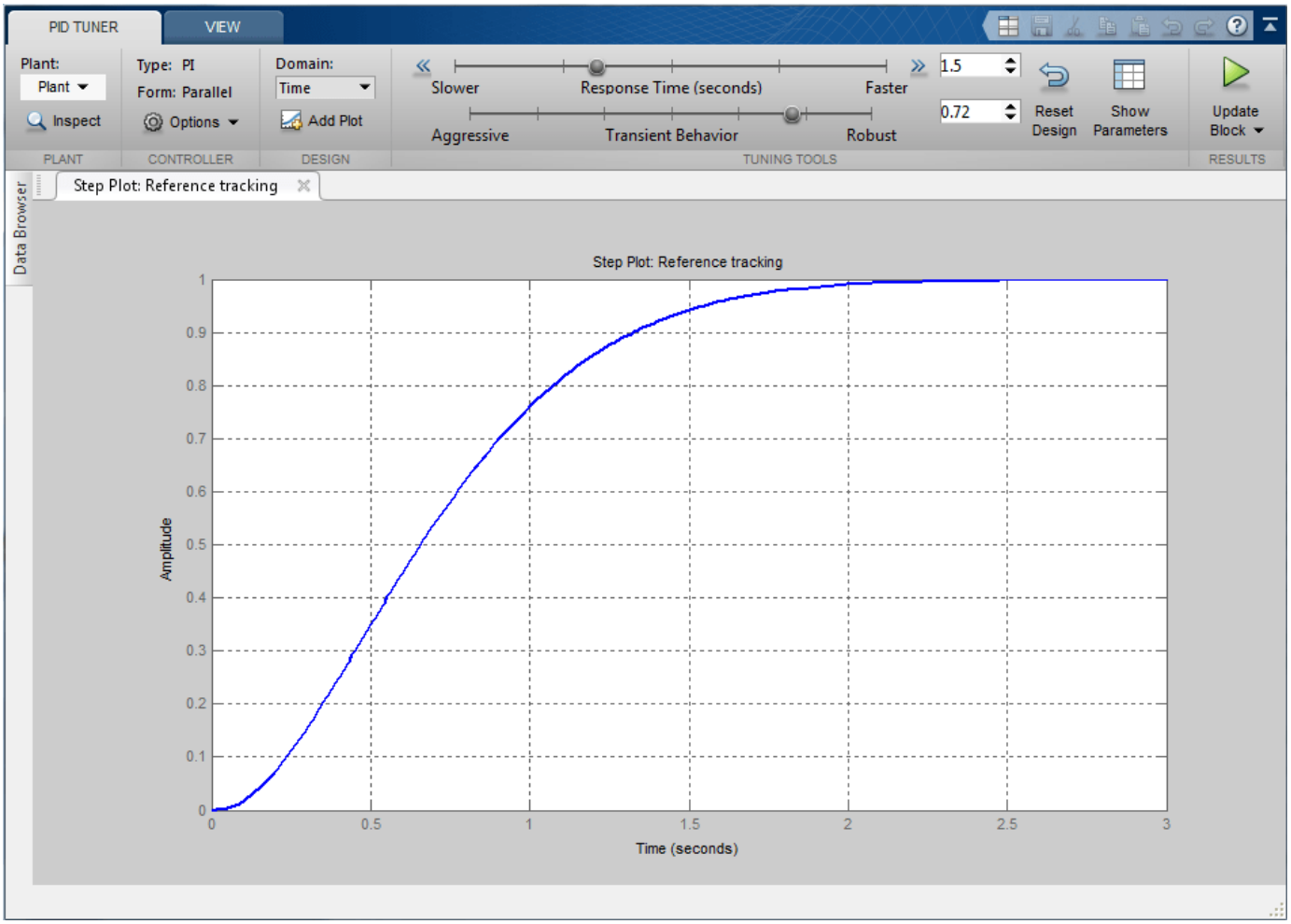
Performance and robustness

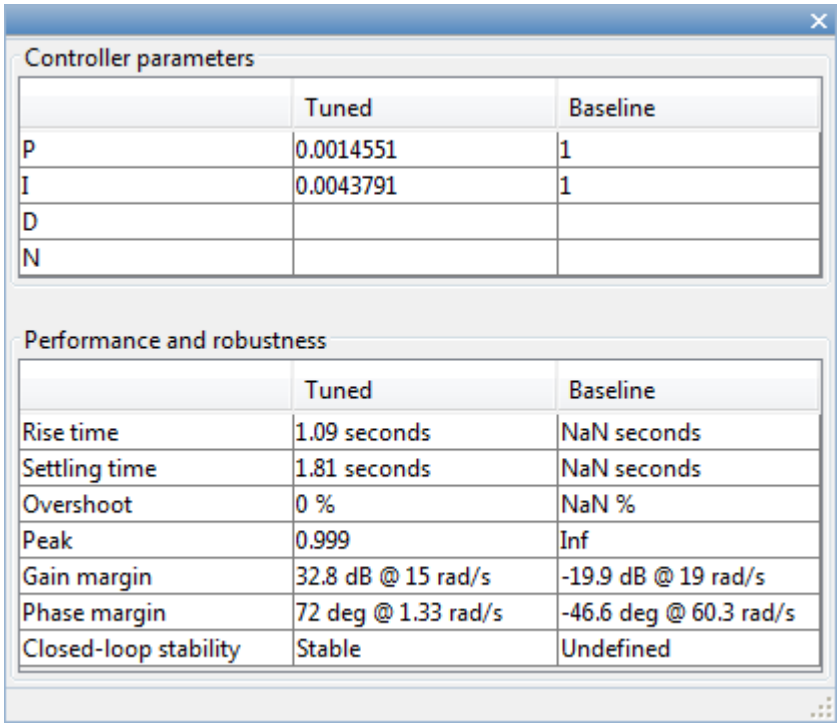
	Tuned	Baseline
Rise time	2.06 seconds	NaN seconds
Settling time	3.45 seconds	NaN seconds
Overshoot	0.401 %	NaN %
Peak	1	Inf
Gain margin	18.9 dB @ 3.27 rad/s	-19.9 dB @ 19 rad/s
Phase margin	69.3 deg @ 0.645 rad/s	-46.6 deg @ 60.3 rad/s
Closed-loop stability	Stable	Undefined

Complete PID Design with Performance Trade-Off

In order to achieve zero overshoot while reducing the settling time below 2 seconds, you need to take advantage of both sliders. You need to make control response faster to reduce the settling time and increase the robustness to reduce the overshoot. For example, you can reduce the response time from 3.4 to 1.5 seconds and increase robustness from 0.6 to 0.72.

The following figure shows the closed-loop response with these settings:


PID TUNER
VIEW
Plant:
Plant
Inspect
Type: PI
Form: Parallel
Options
Domain:
Time
Add Plot
Slower
Response Time (seconds)
Faster
1.5
Aggressive
Transient Behavior
Robust
0.72
Reset Design
Show Parameters
Update Block
PLANT
CONTROLLER
DESIGN
TUNING TOOLS
RESULTS
Data Browser
Step Plot: Reference tracking
Step Plot: Reference tracking
Amplitude
Time (seconds)
0
0.1
0.2
0.3
0.4
0.5
0.6
0.7
0.8
0.9
1
0.5
1.5
2
2.5
3



Controller parameters

	Tuned	Baseline
P	0.0014551	1
I	0.0043791	1
D		
N		

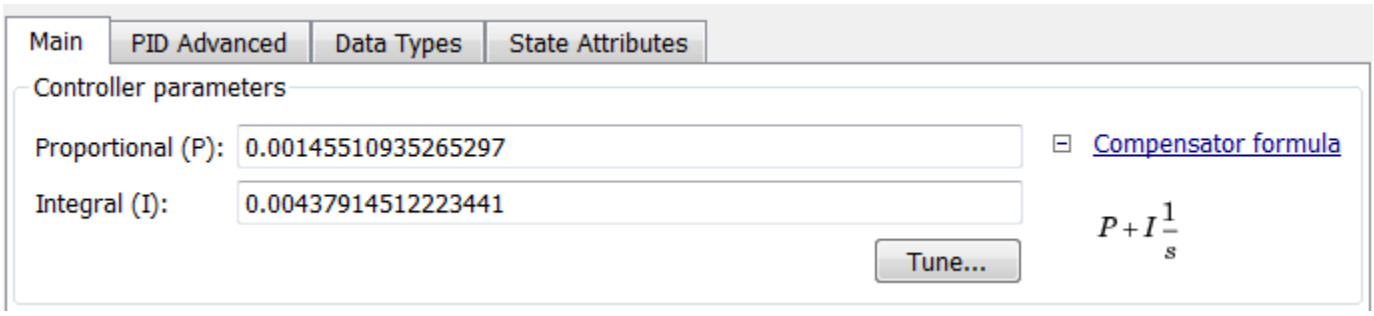
Performance and robustness

	Tuned	Baseline
Rise time	1.09 seconds	NaN seconds
Settling time	1.81 seconds	NaN seconds
Overshoot	0 %	NaN %
Peak	0.999	Inf
Gain margin	32.8 dB @ 15 rad/s	-19.9 dB @ 19 rad/s
Phase margin	72 deg @ 1.33 rad/s	-46.6 deg @ 60.3 rad/s
Closed-loop stability	Stable	Undefined

Write Tuned Parameters to PID Controller Block

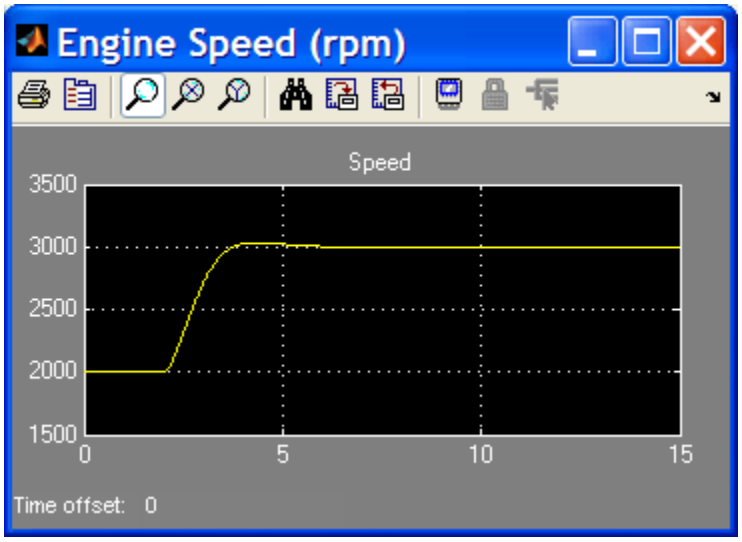
After you are happy with the controller performance on the linear plant model, you can test the design on the nonlinear model. To do this, click **Update Block** in the **PID Tuner**. This action writes the parameters back to the PID Controller block in the Simulink model.

The following figure shows the updated PID Controller block dialog:



Completed Design

The following figure shows the response of the closed-loop system:



The response shows that the new controller meets all the design requirements.

You can also use the **Control System Designer** to design the PID Controller block, when the PID Controller block belongs to a multi-loop design task. See the example “Single Loop Feedback/Prefilter Compensator Design” (Simulink Control Design).

```
bdclose('scdspeedctrlpidblock')
```

See Also

PID Tuner

More About

- “Tune PID Controller to Favor Reference Tracking or Disturbance Rejection” (Simulink Control Design)
- “Analyze Design in PID Tuner” (Simulink Control Design)

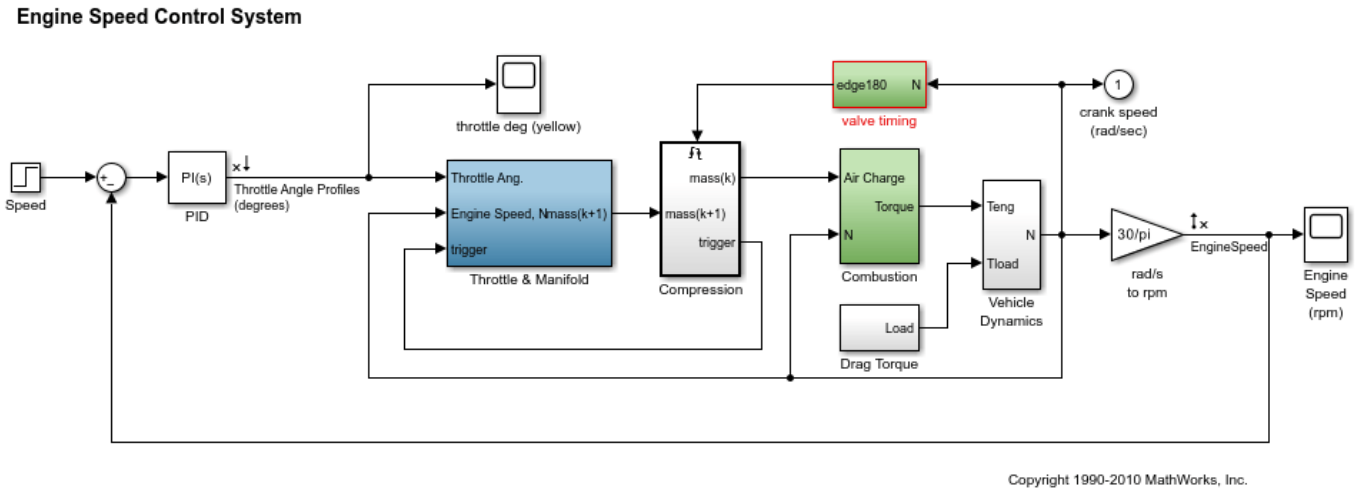
Design PID Controller Using Estimated Frequency Response

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

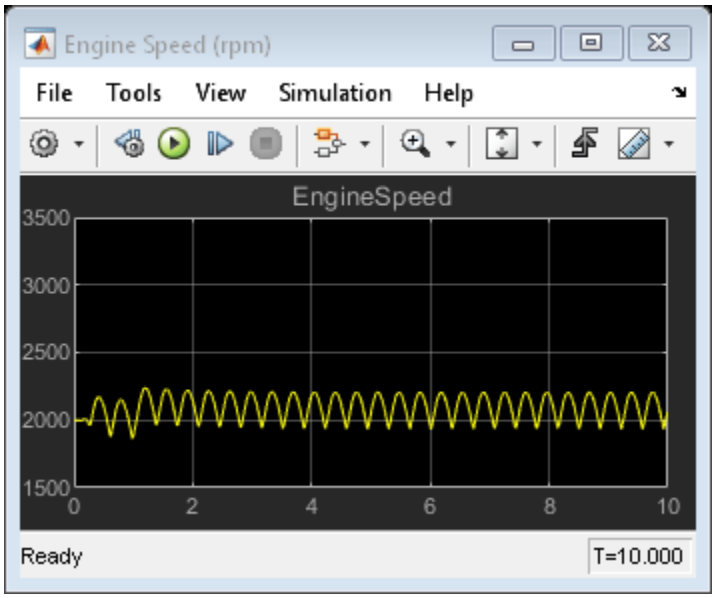
Open the engine control model and take a few moments to explore it.

```
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);
```

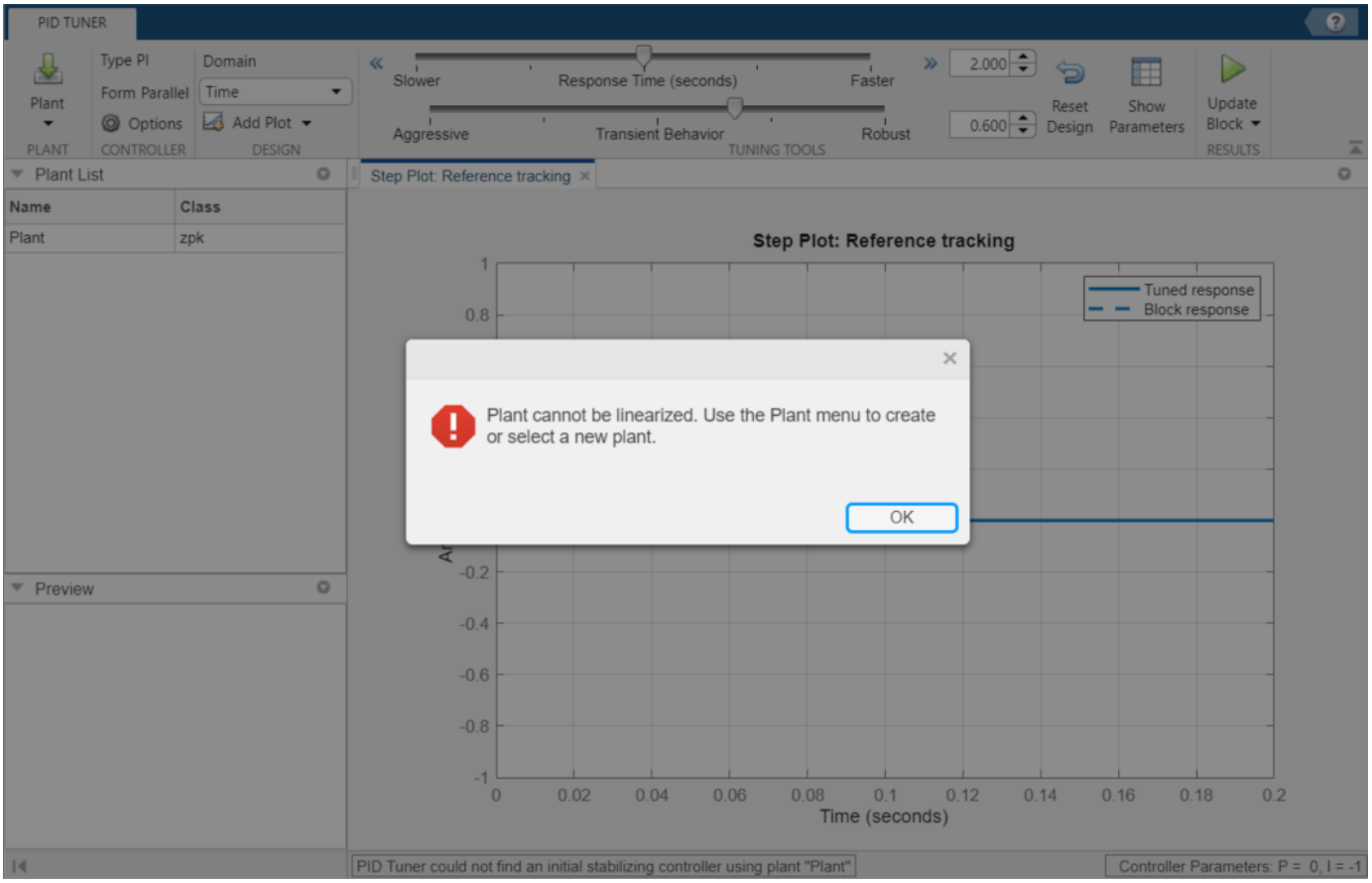
Then, linearize the plant at its initial operating point.

```
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” (Simulink Control Design).

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.

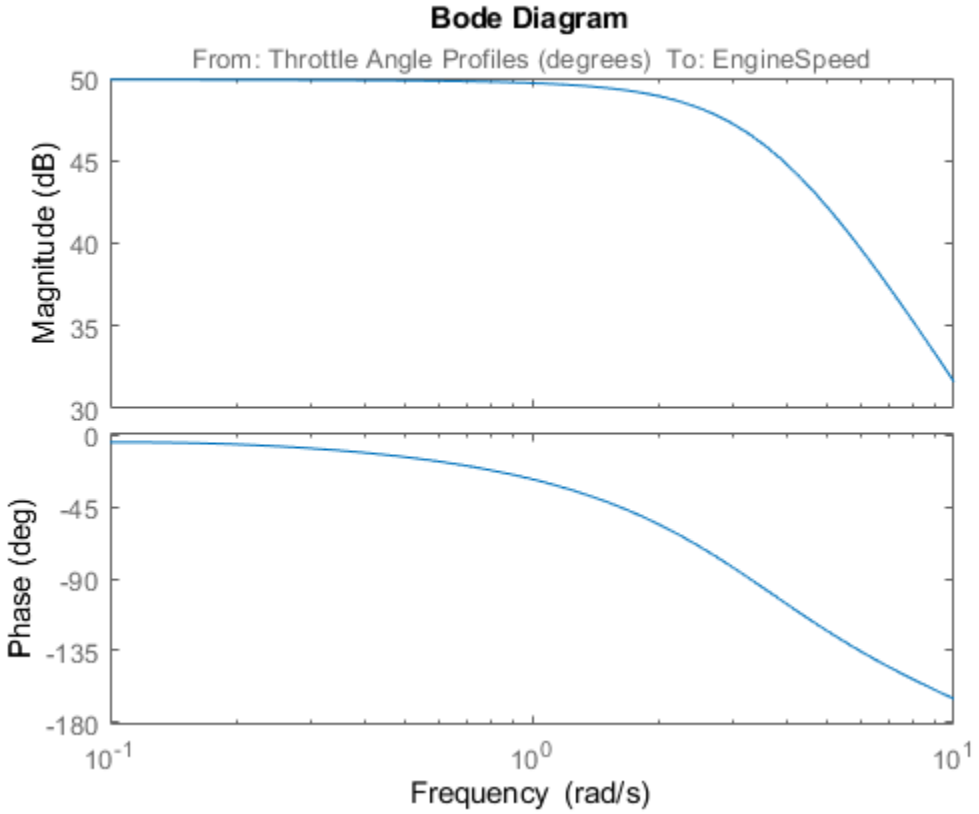
```
in = frest.Sinestream('Frequency',logspace(-1,1,50),'Amplitude',1e-3);
```

Estimate the frequency response. This process can take a few minutes.

```
sys = frestimate(mdl,io,in);
```

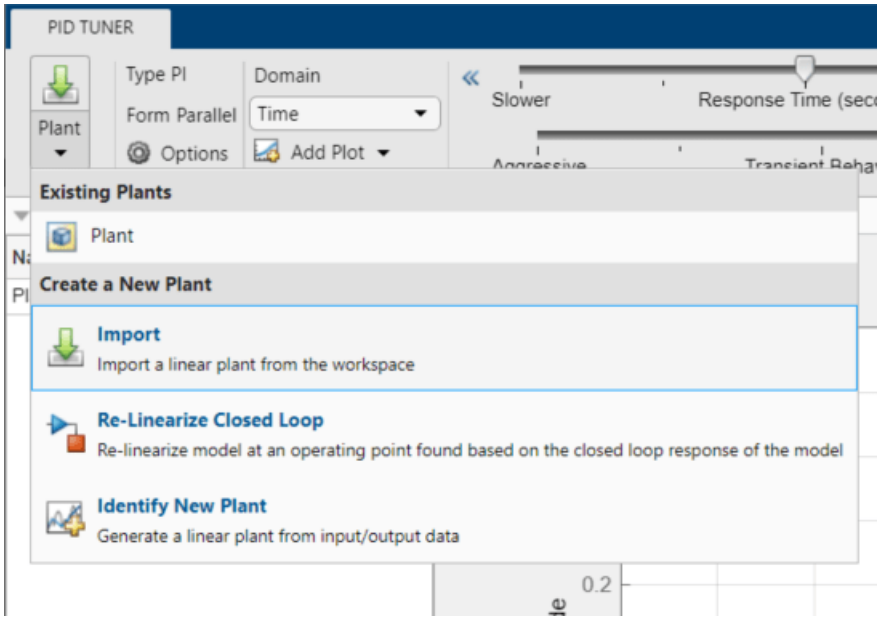
Display the estimated frequency response.

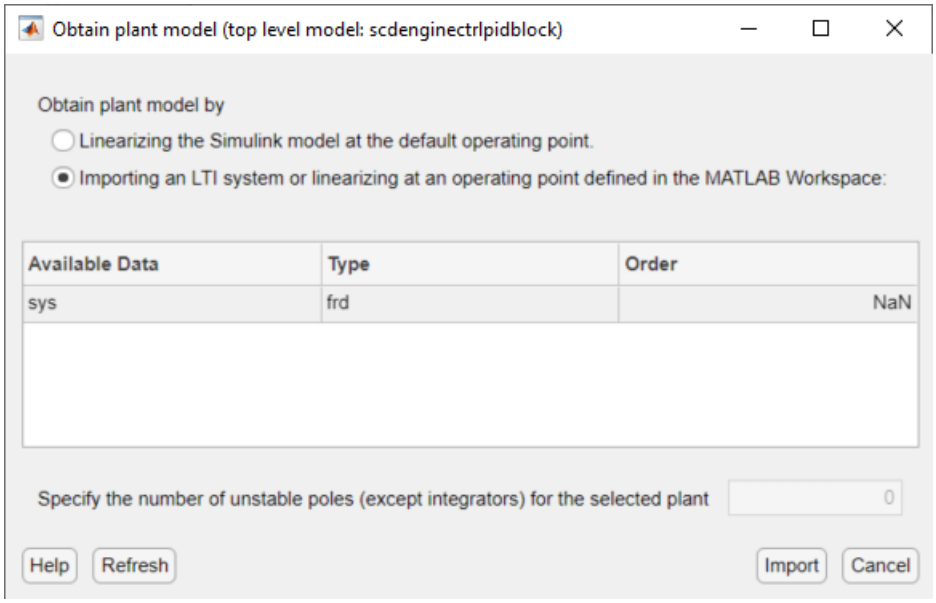
```
bode(sys)
```



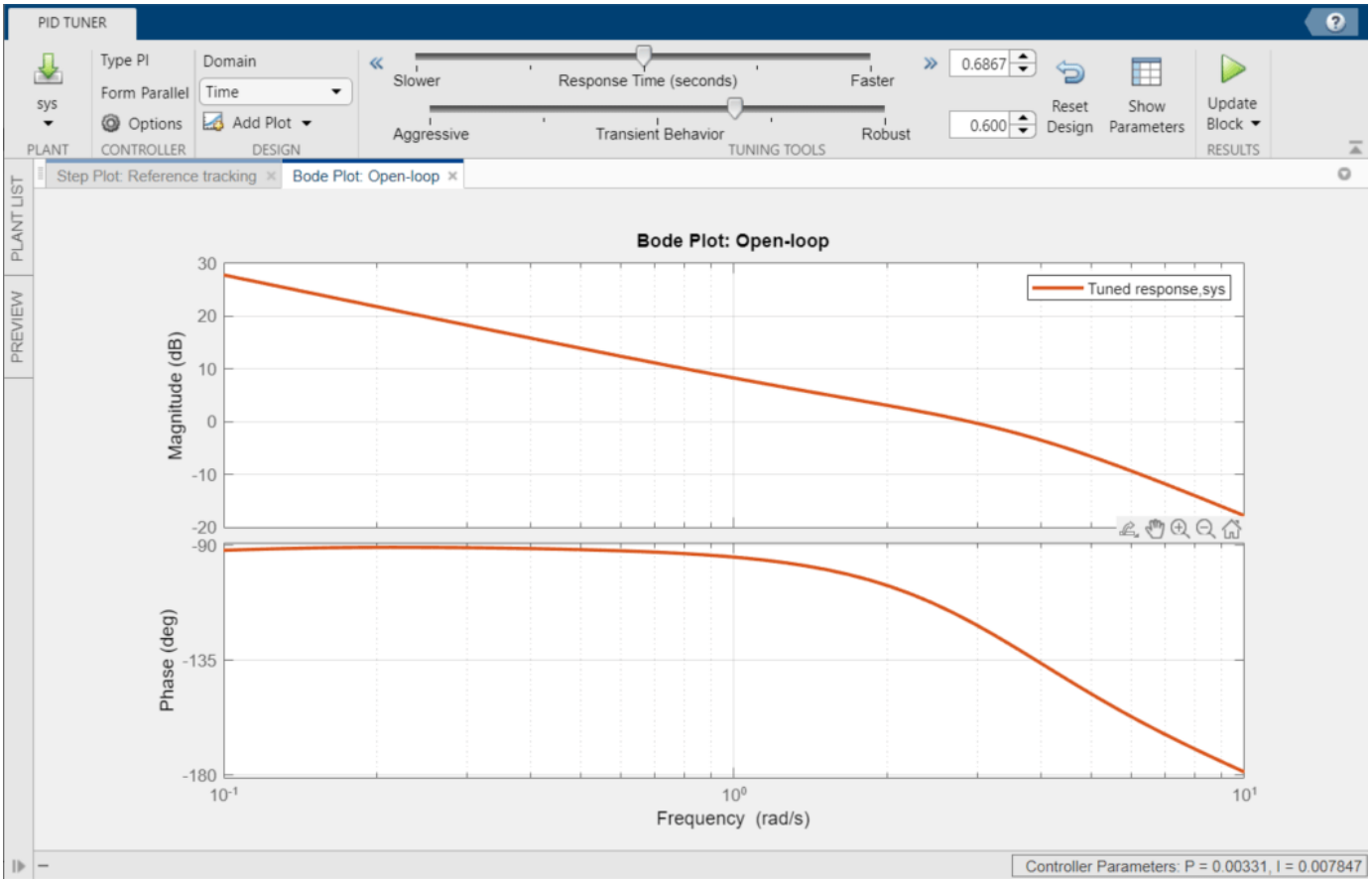
Design PI Controller

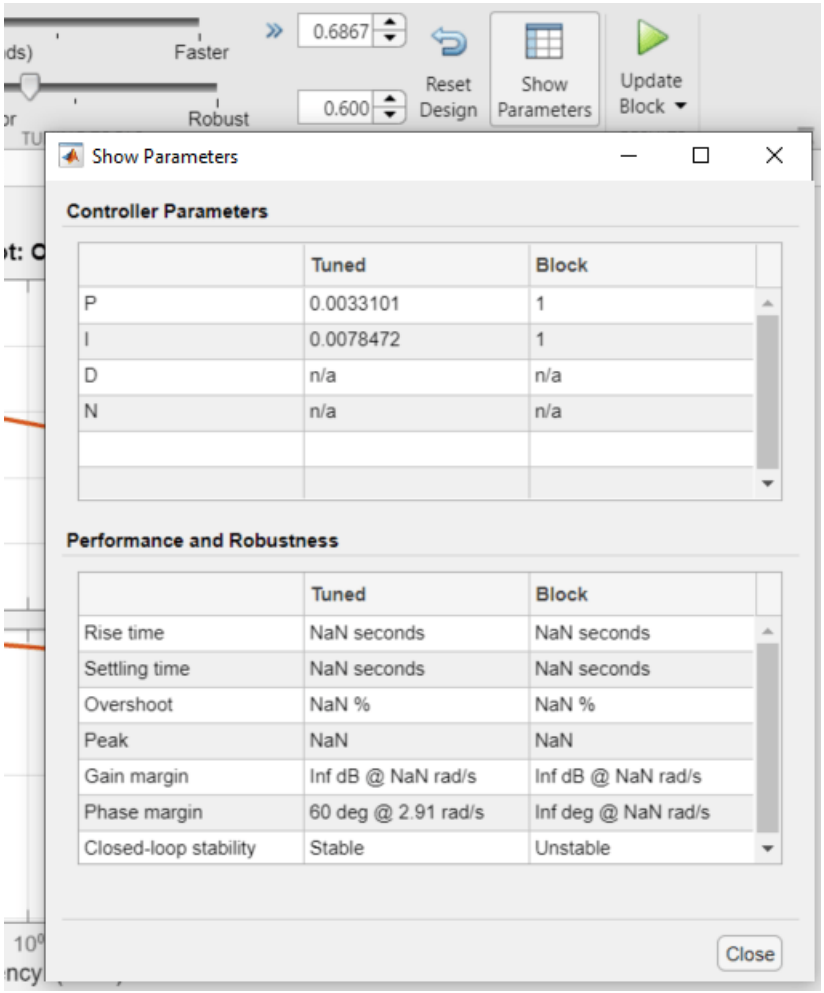
`sys` is an `frd` system that represents the plant frequency response at the initial operating point. To use it in **PID Tuner**, we need to import it after **PID Tuner** is launched. Click **Plant**, and select **Import**. The sampling rate of the imported `frd` plant must match the sampling rate of 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.

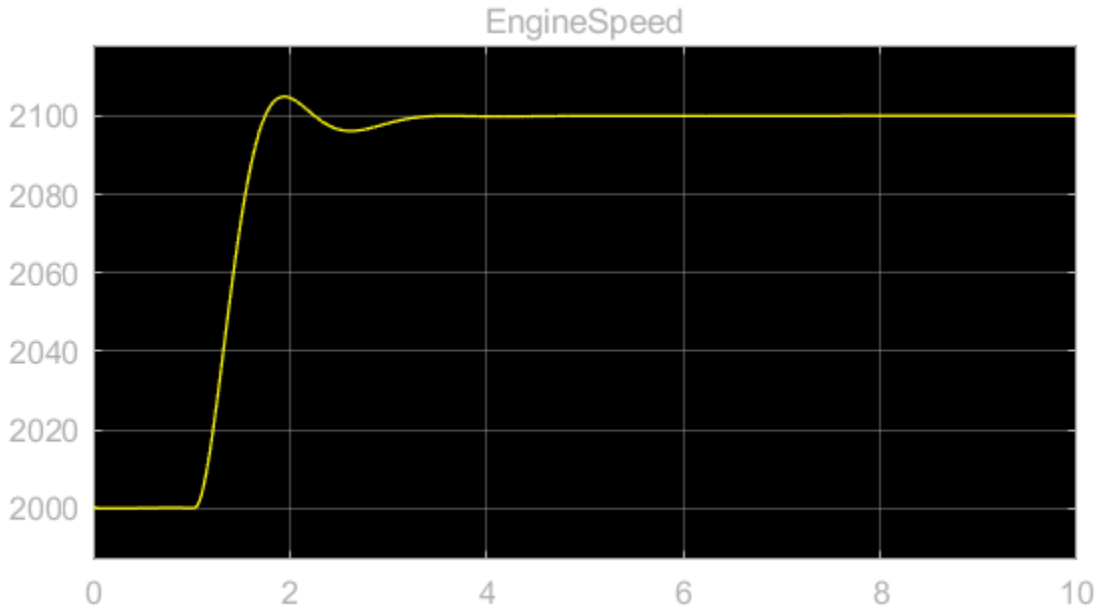

PID TUNER
Type PI
Form Parallel
Options
sys
PLANT
CONTROLLER
Domain
Time
Add Plot
DESIGN
Slower
Response Time (seconds)
Faster
Aggressive
Transient Behavior
Robust
TUNING TOOLS
0.6867
0.600
Reset Design
Show Parameters
Update Block
RESULTS
PLANT LIST
PREVIEW
Step Plot: Reference tracking
Bode Plot: Open-loop
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” (Simulink Control Design)

Design Family of PID Controllers for Multiple Operating Points

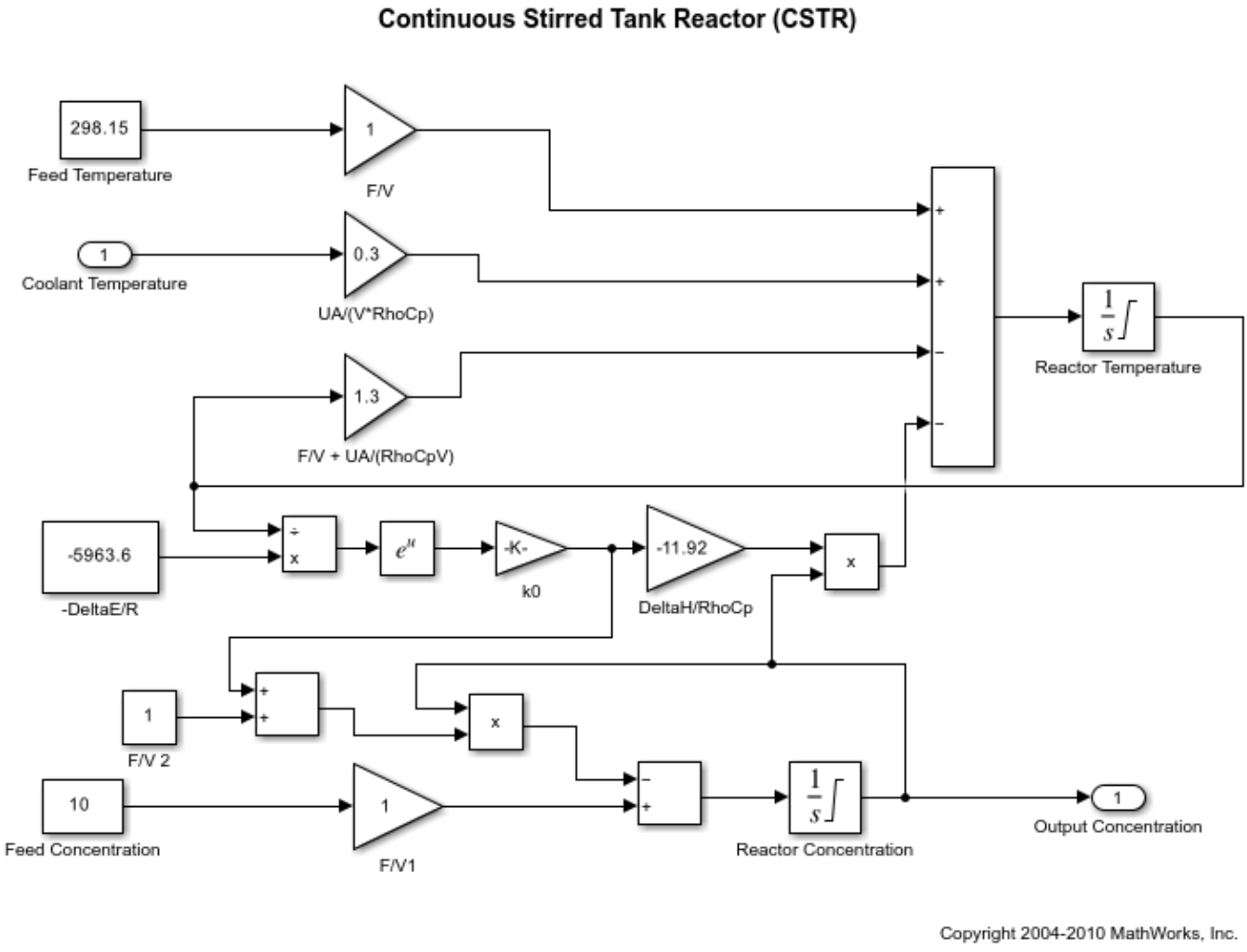
This example shows how to design an array of PID controllers for a nonlinear plant in Simulink® that operates over a wide range of operating points.

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)
```

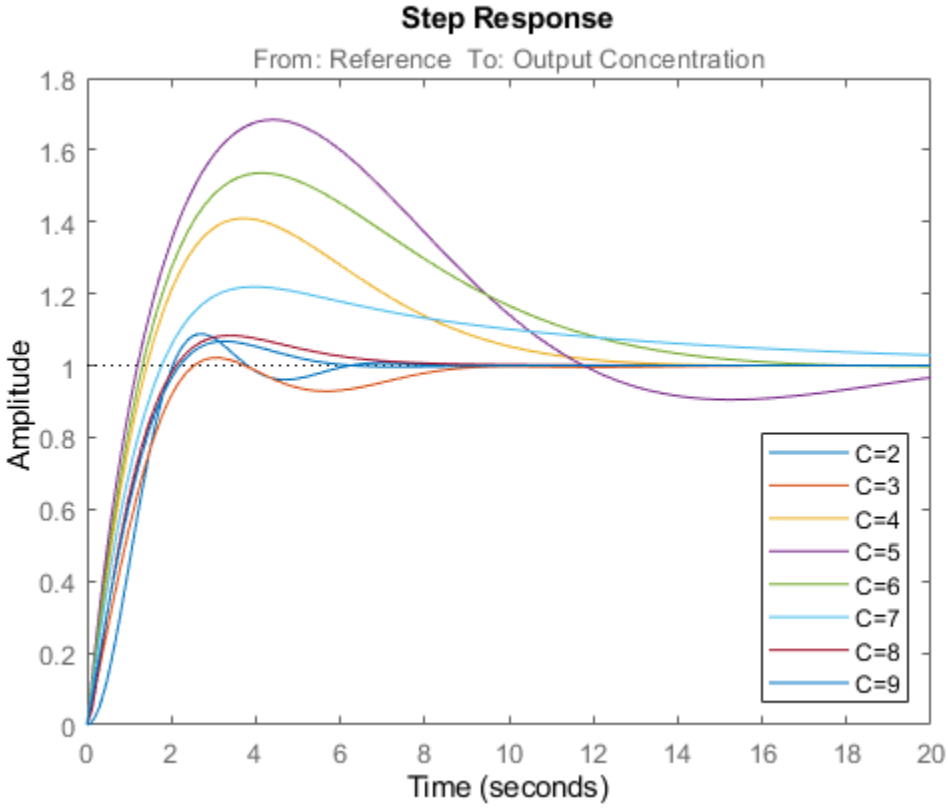
```
ans =

             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`.

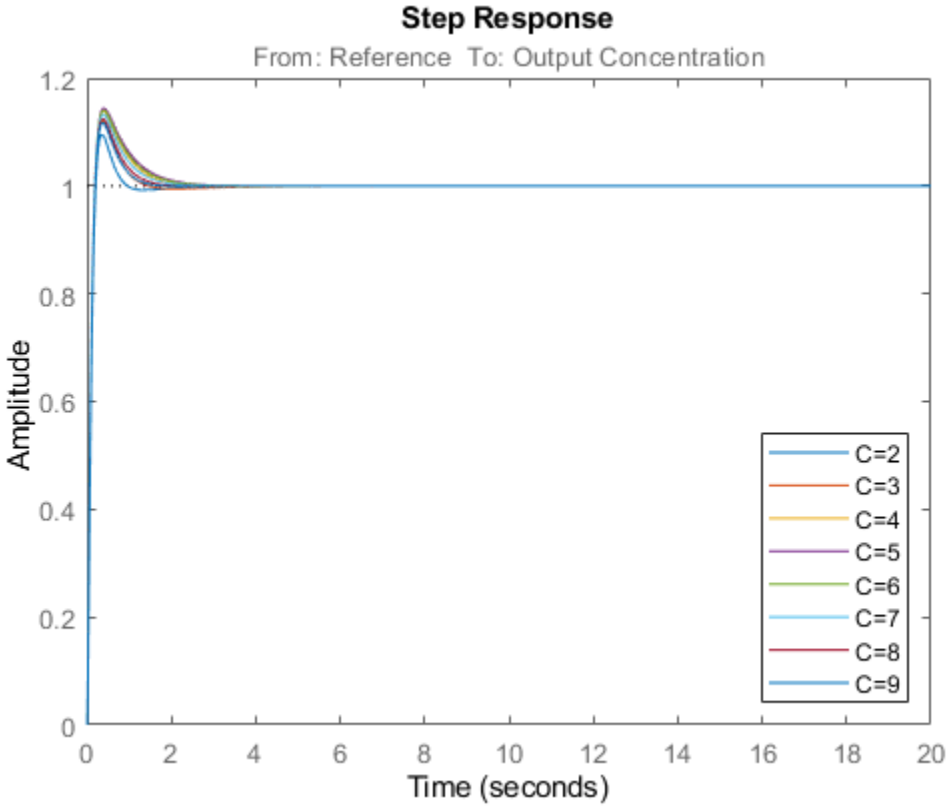
```
Controllers(:,:,4)
```

```
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.

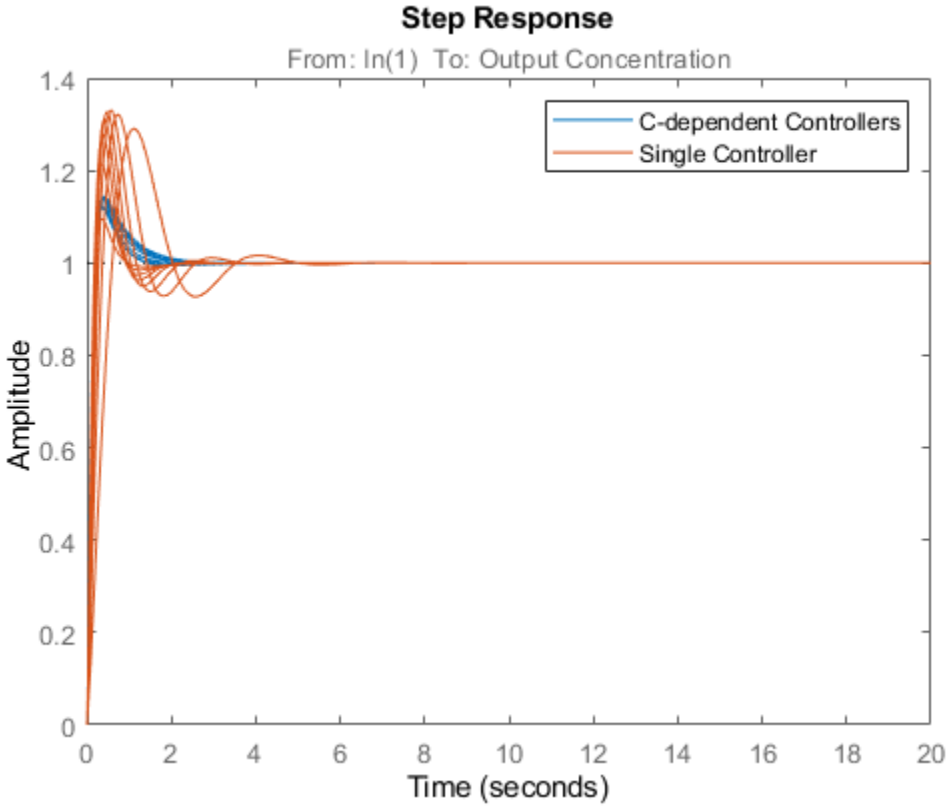
```
clsys = feedback(Plants*Controllers,1);
figure
hold on
for ct = 1:length(C)
    % Select a system from the LTI array.
```

```
    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” (Simulink Control Design).

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” (Simulink Control Design)

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.

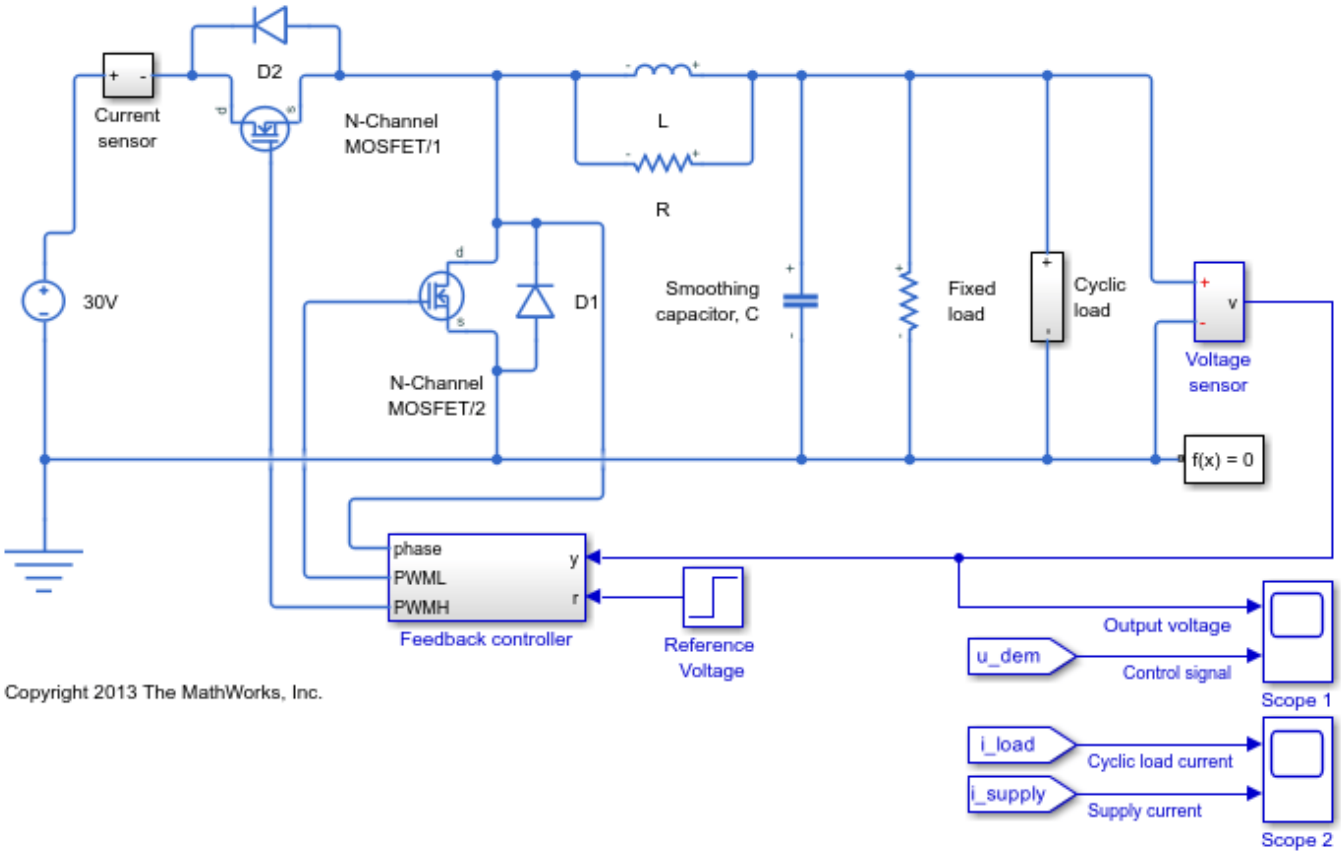
This example uses a buck converter model that requires Simscape™ Electrical™ software.

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.

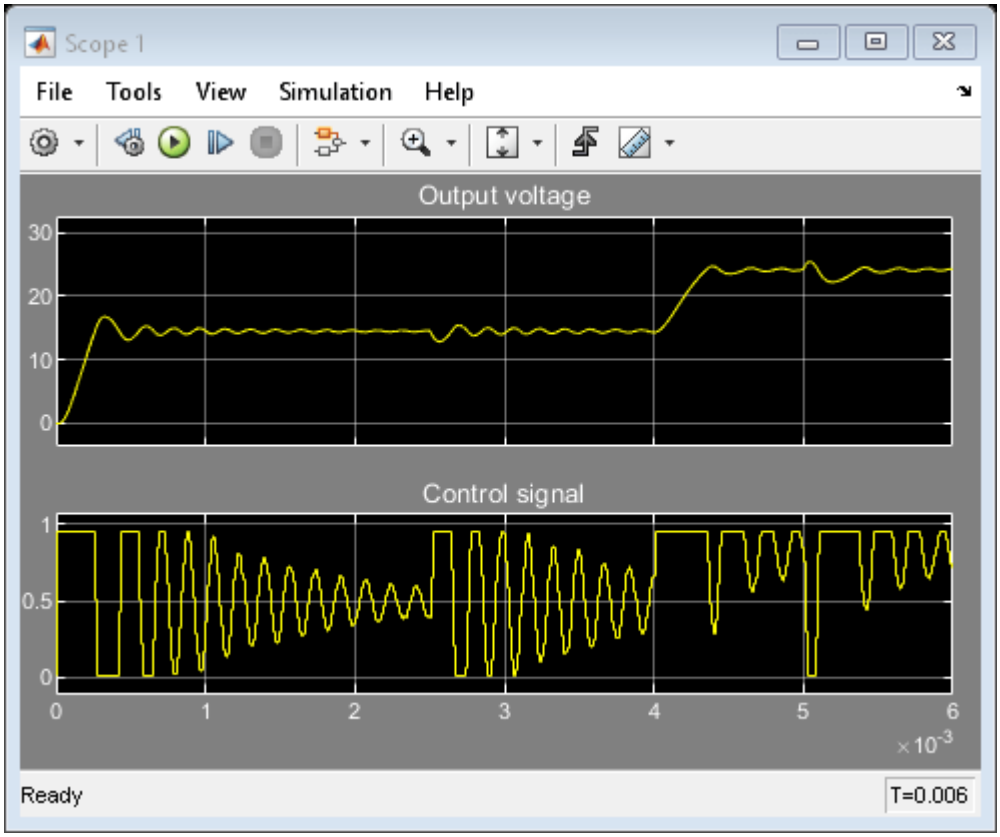
For more information on creating a buck converter model, see “Buck Converter” (Simscape Electrical).

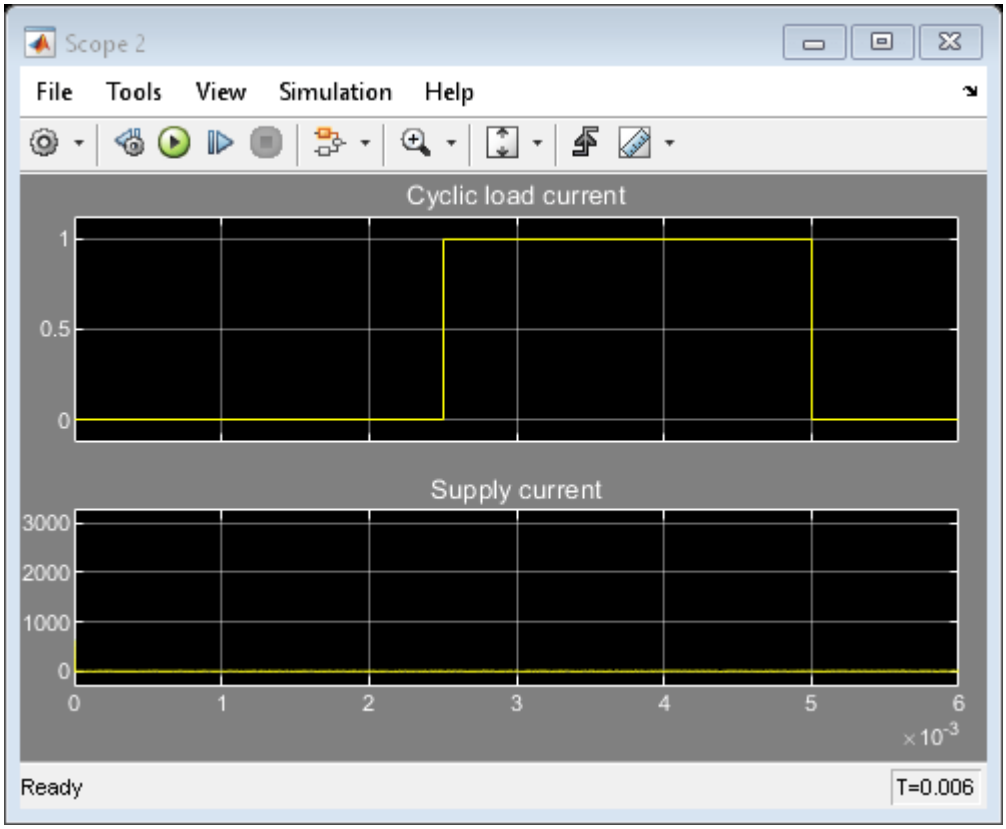
```
open_system('scdbuckconverter')
sim('scdbuckconverter')
```



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.

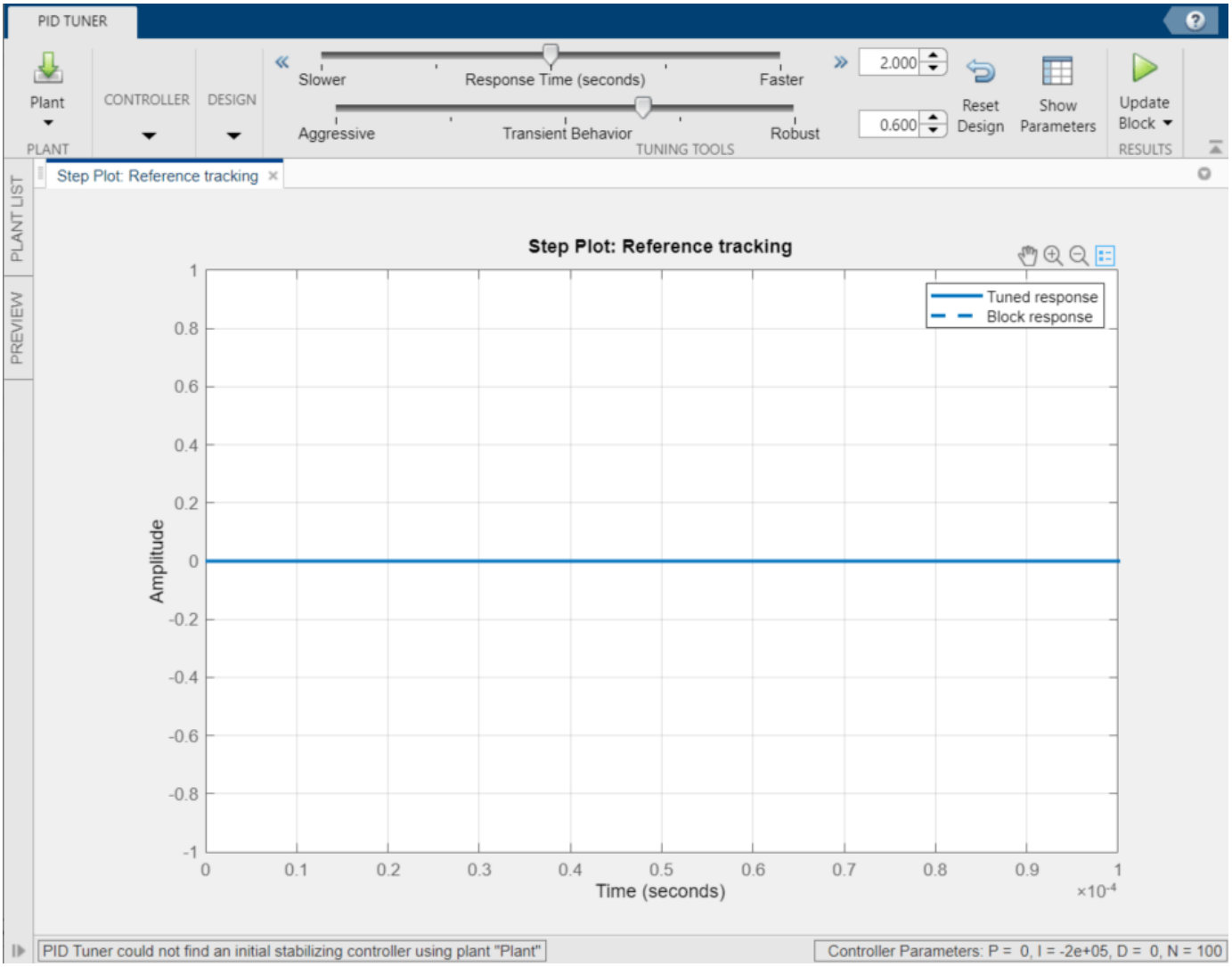
```
open_system('scdbuckconverter/Scope 1')
open_system('scdbuckconverter/Scope 2')
```





Simulate Model to Generate I/O Data

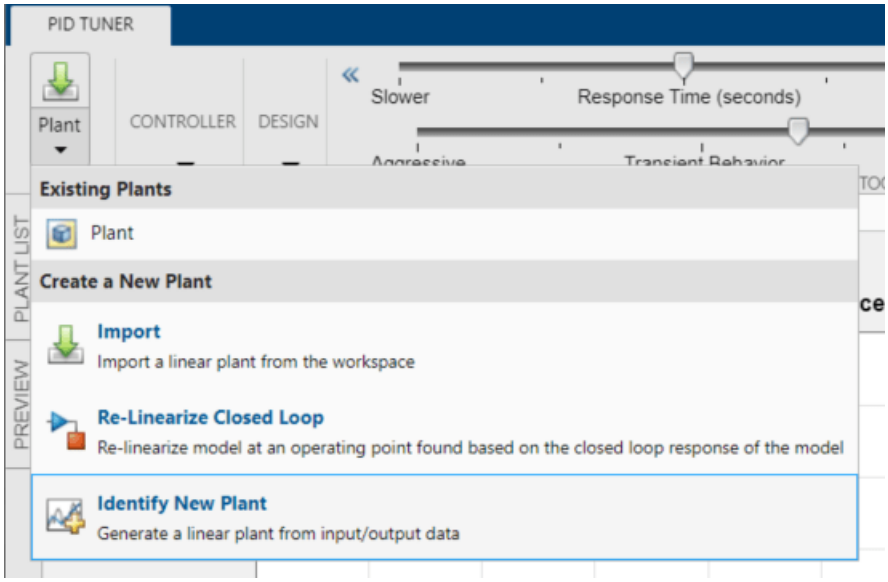
To open the PID Tuner, in the **Feedback controller** subsystem, open the **PID Controller** block dialog, and click **Tune**. **PID Tuner** indicates that the model cannot be linearized and returned a zero system.



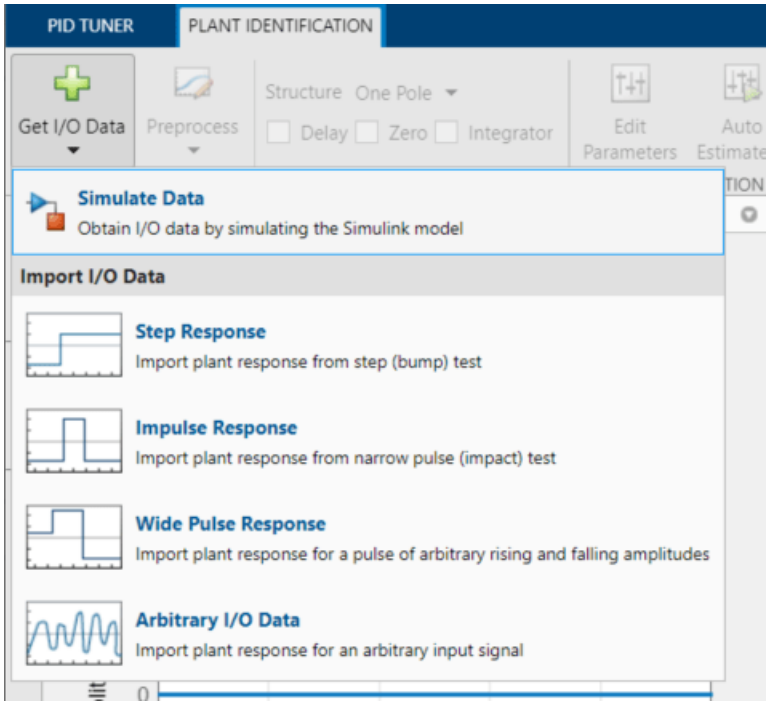
PID Tuner provides several alternatives when linearization fails. In the **Plant** drop-down list, you can select one of the following methods:

- **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**.



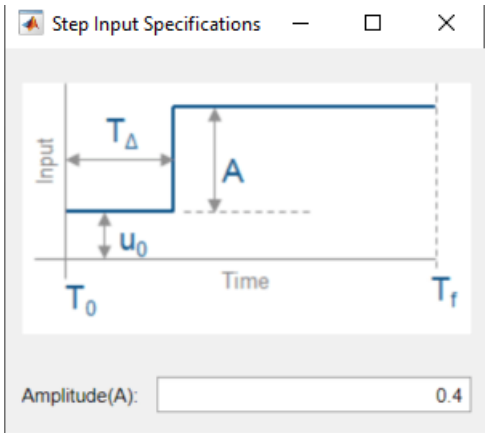
On the **Simulate I/O Data** tab, you simulate the plant seen by the controller. The software temporarily:

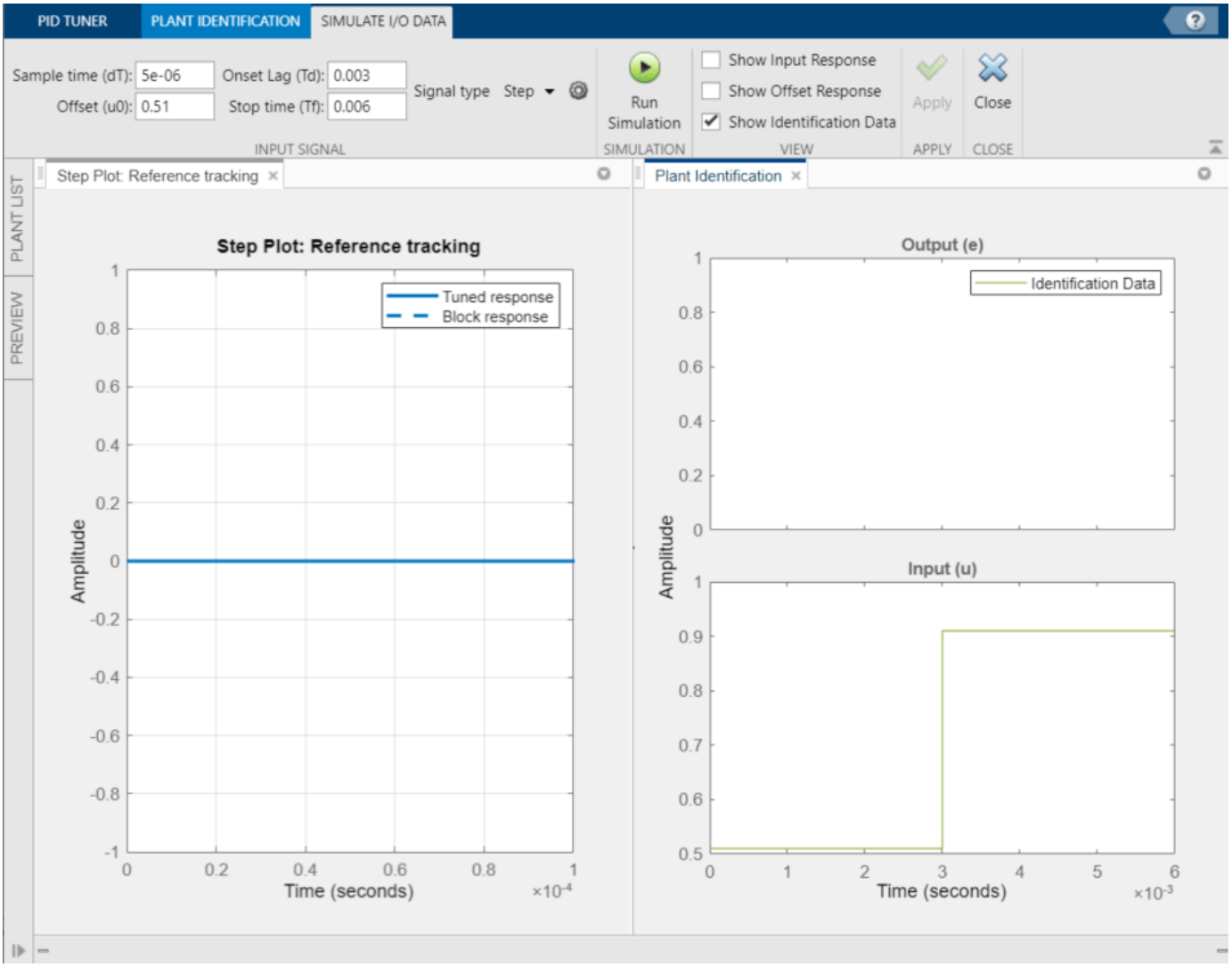
- Removes the PID Controller block from the model.
- Injects a signal where the output of the PID block used to be.
- Measures the resulting signal where the input to the PID block used to be.

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.

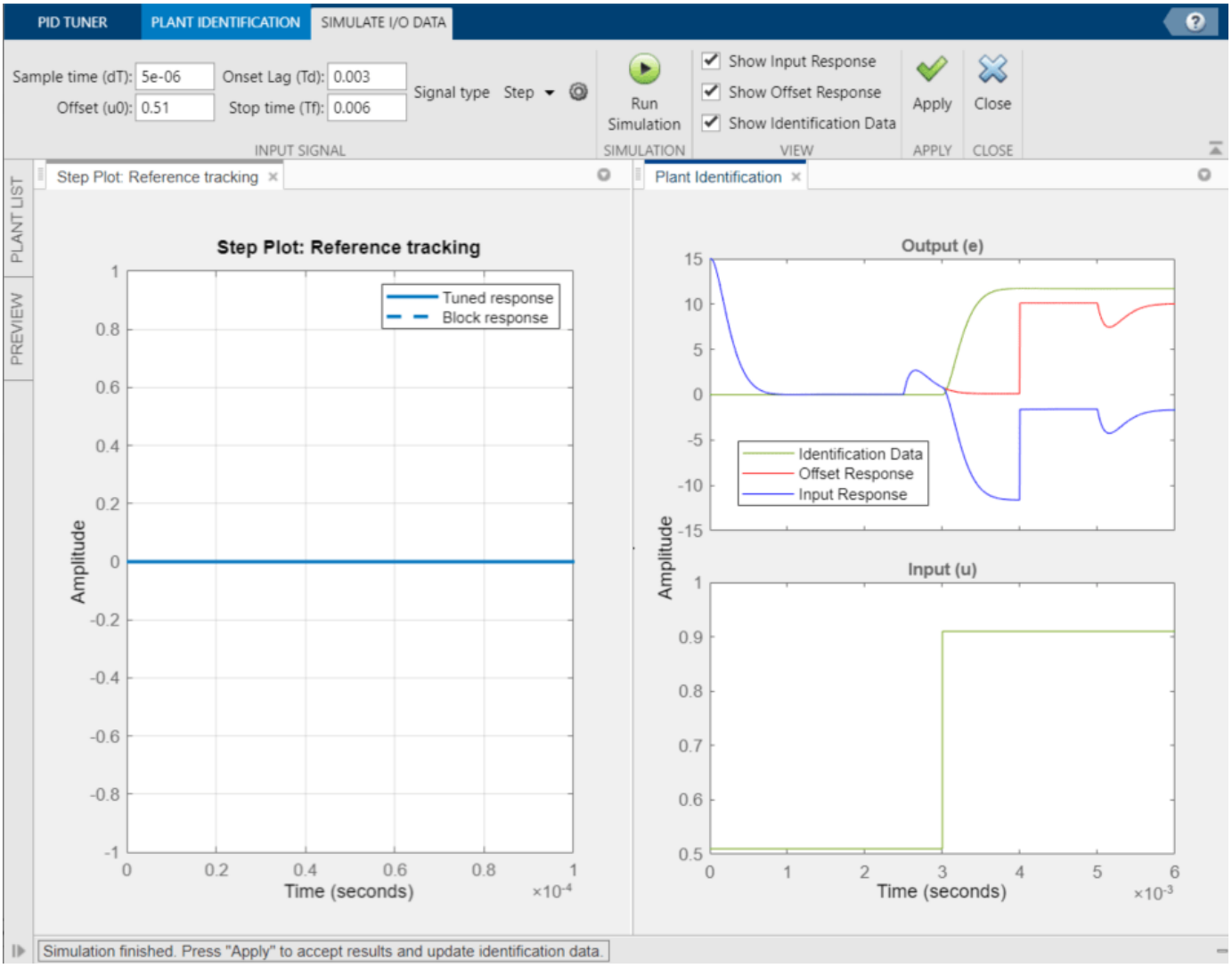
Configure the input signal as a step input with the following properties:

- **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.

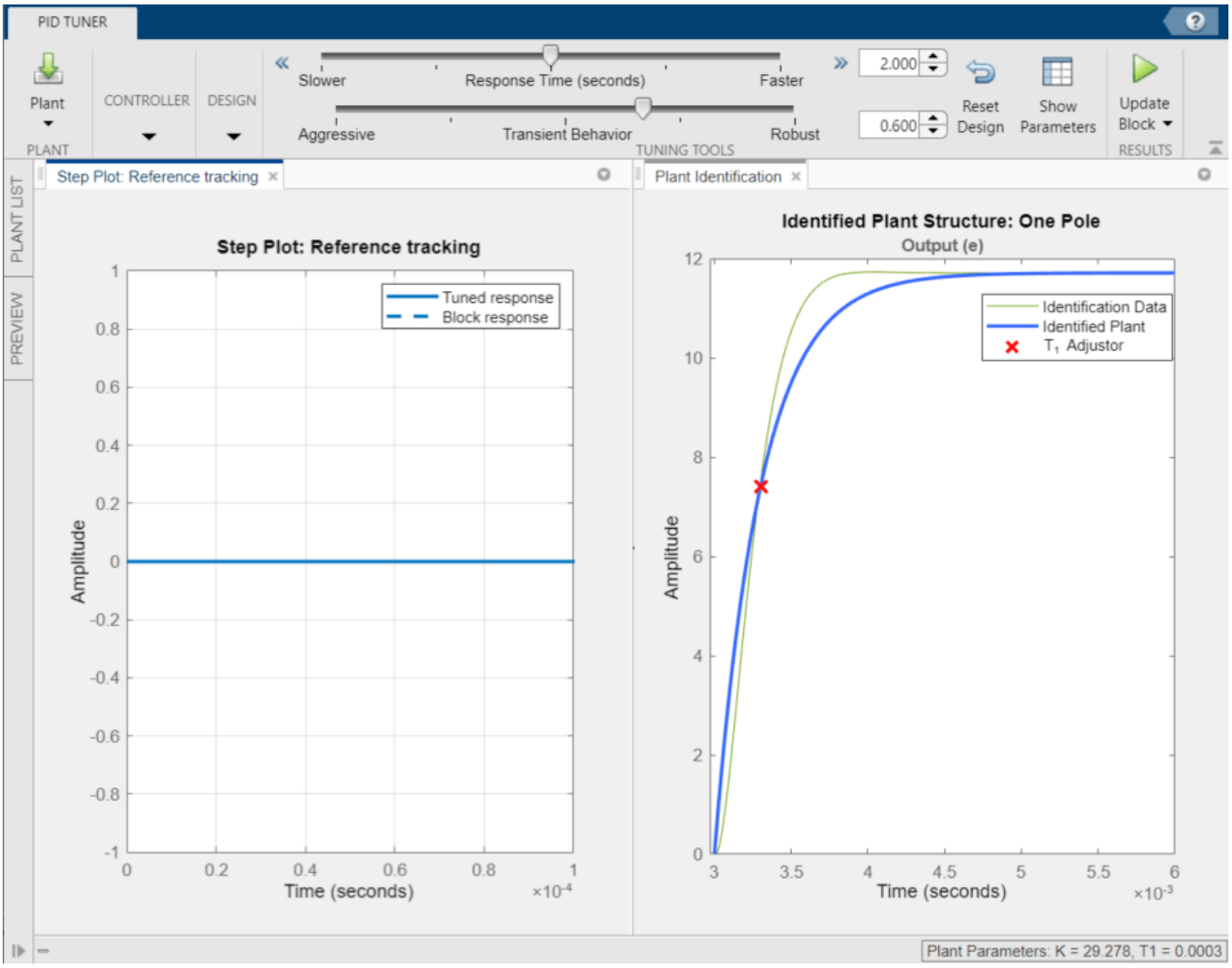
The blue curve shows the complete plant response that contains the contributions from the initial transients (significant for times < 0.001 seconds), the response to the cyclic current load (time durations 0.0025 to 0.005 seconds), reference voltage change (at 0.004 seconds), and response to the step 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.

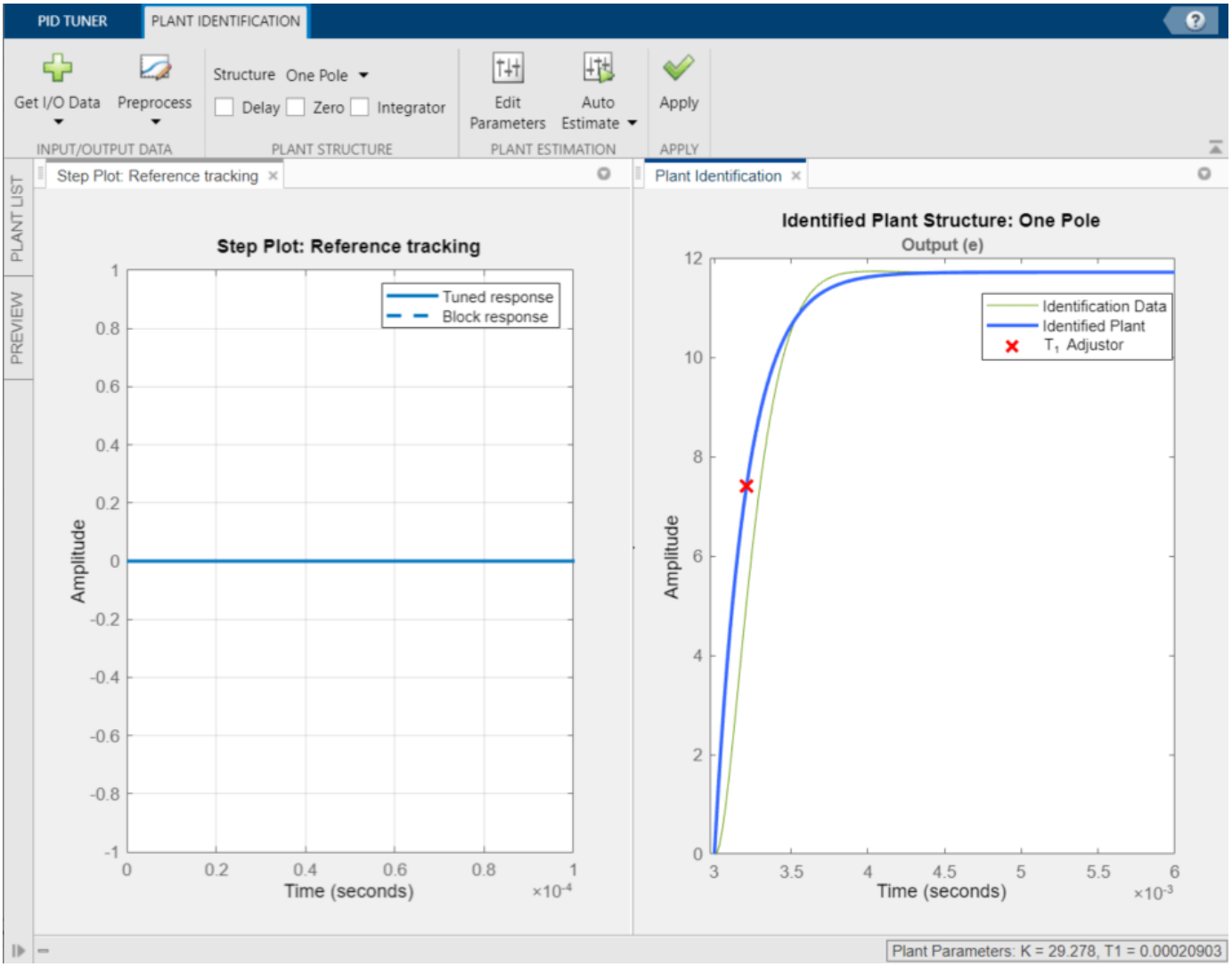
To use the measured data to identify a plant model, click **Apply**. Then, to return to plant identification, click **Close**.

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.

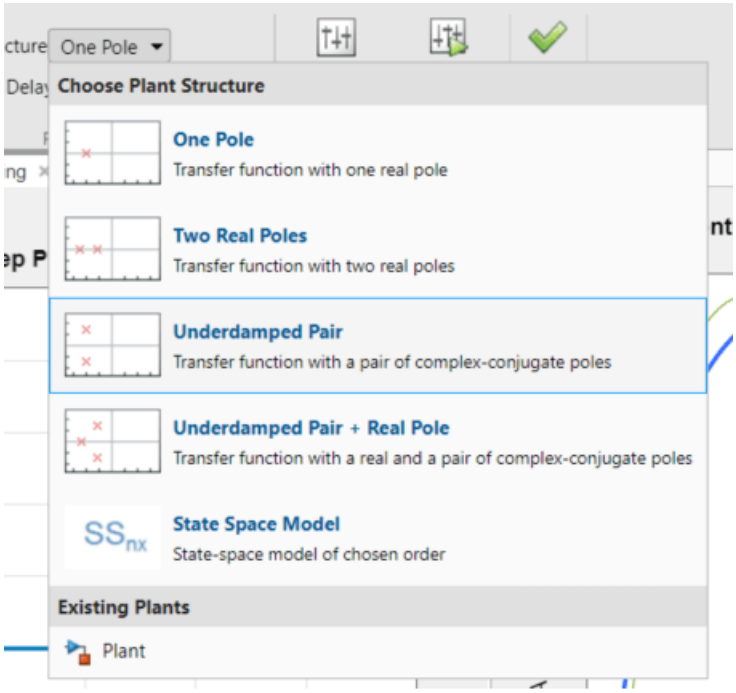


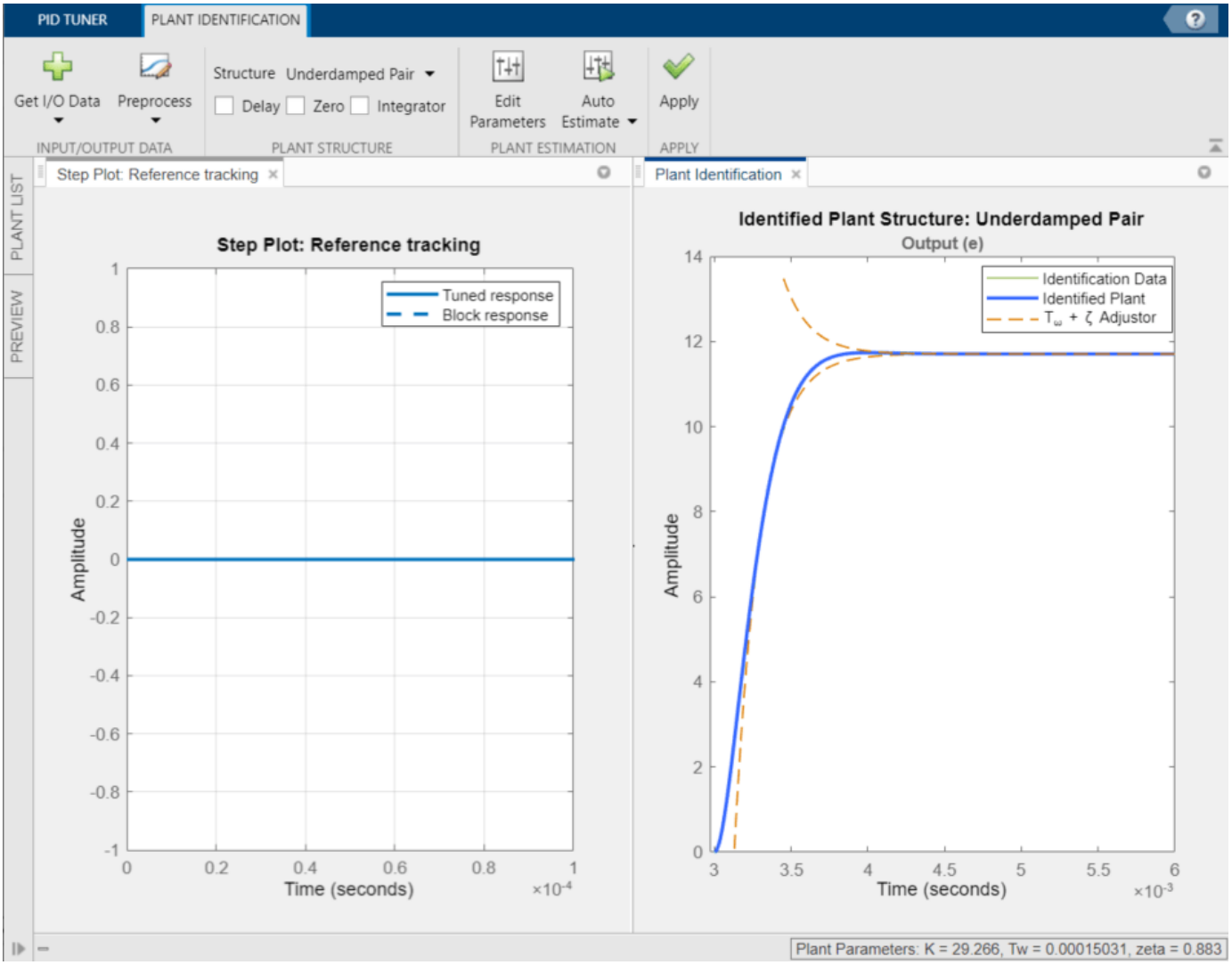
You can manually adjust the estimated model. Click and drag the plant curve and pole location (X) to adjust the identified plant response so that it matches the identification data as closely as possible.



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.

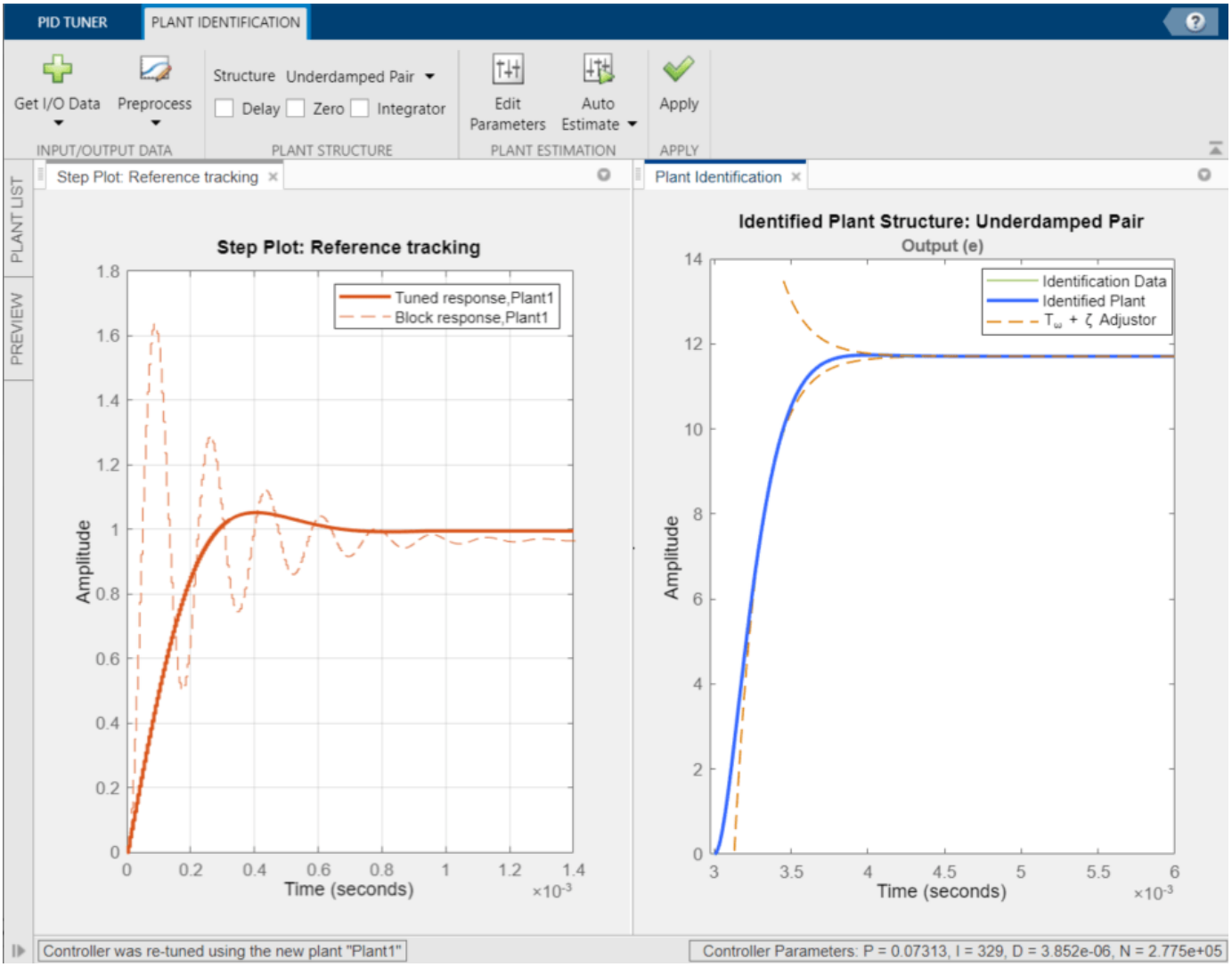

One Pole
Choose Plant Structure
One Pole
Transfer function with one real pole
Two Real Poles
Transfer function with two real poles
Underdamped Pair
Transfer function with a pair of complex-conjugate poles
Underdamped Pair + Real Pole
Transfer function with a real and a pair of complex-conjugate poles
SSnx
State Space Model
State-space model of chosen order
Existing Plants
Plant



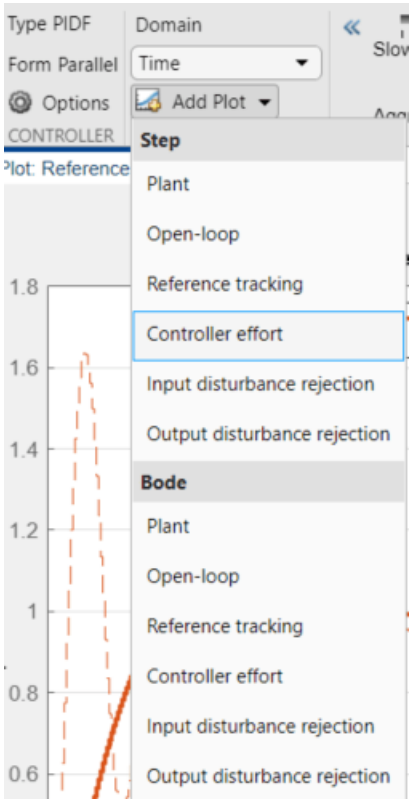
To designate the identified model as the current plant for controller tuning, Click **Apply**. **PID Tuner** then automatically tunes a controller for the identified plant and updates the **Reference Tracking** step plot.

Controller Tuning

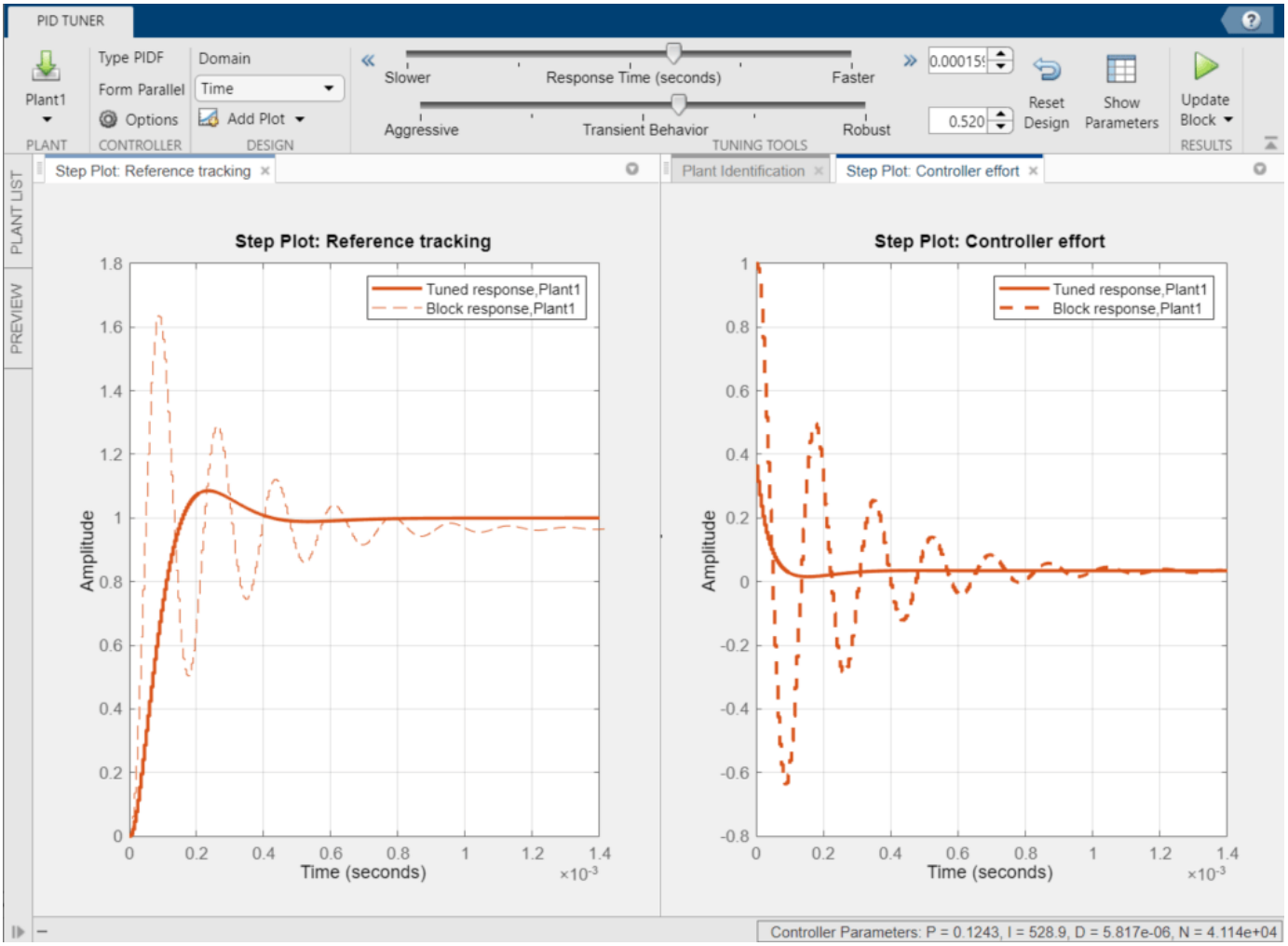
The PID Tuner automatically tunes a PID controller for the identified plant. The tuned controller response has about 5% overshoot and a settling time of around 0.0006 seconds. Click the **Reference Tracking** step plot to make it the current figure.



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.

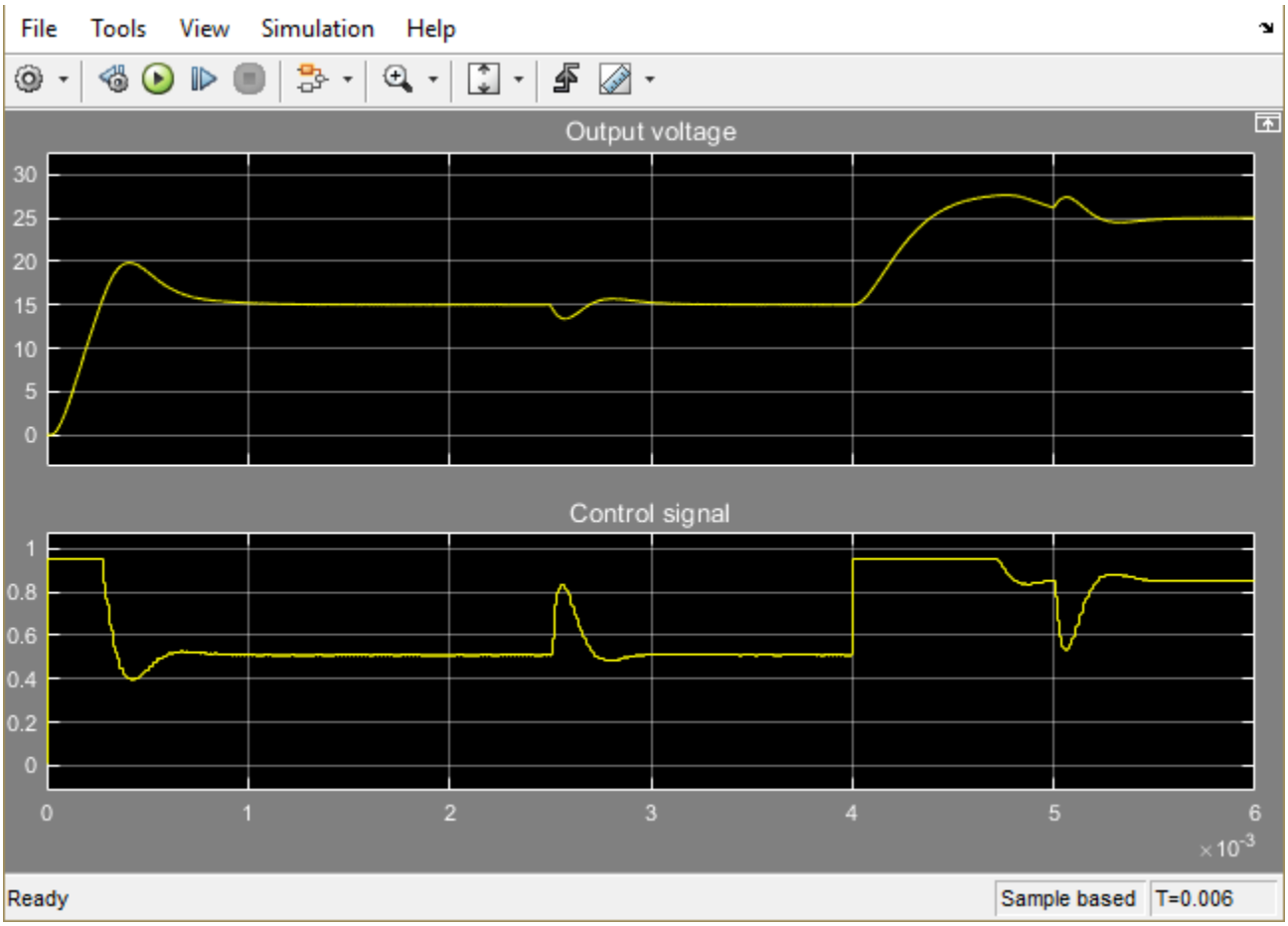


In the **Controller effort** plot, the tuned response (solid line) shows a large control effort required at the start of the simulation. To achieve a settling time of about 0.0004 seconds and overshoot of 9%, adjust the **Response Time** and **Transient Behavior** sliders. These adjustments reduce the 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” (Simulink Control Design)
- “Input/Output Data for Identification” (Simulink Control Design)
- “Interactively Estimate Plant from Measured or Simulated Response Data” (Simulink Control Design)
- “Choosing Identified Plant Structure” (Simulink Control Design)

PID Controller Design in the Live Editor

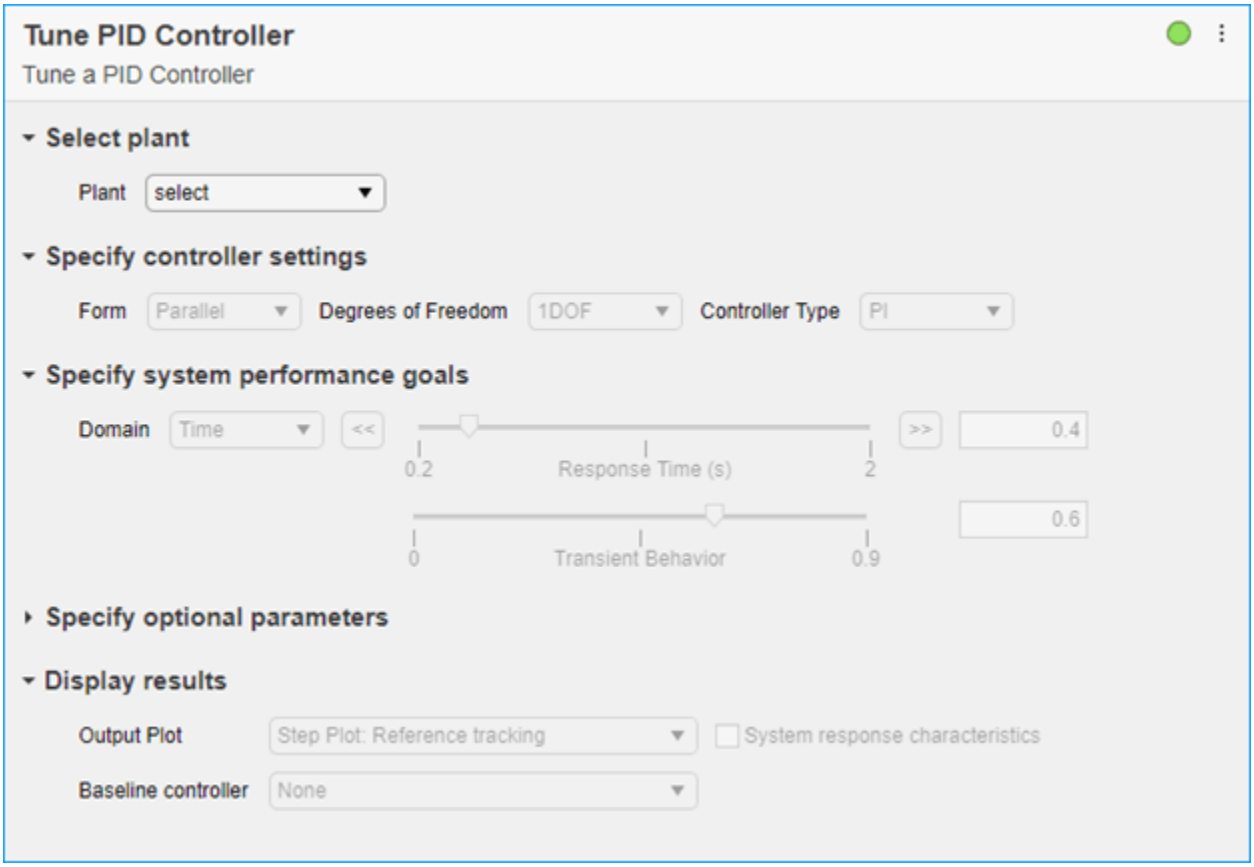
This example shows how to use the Tune PID Controller task in the Live Editor to generate code for designing a PID controller for a linear plant model. The **Tune PID Controller** task lets you interactively refine the performance of the controller to adjust loop bandwidth and phase margin, or to favor setpoint tracking or disturbance. The task generates a response plot that lets you monitor controller performance while you adjust the tuning parameters.

Open this example to see a preconfigured script containing the **Tune PID Controller** task. For more information about Live Editor tasks generally, see “Add Interactive Tasks to a Live Script”.

In the Live Editor, create an LTI model for your plant.

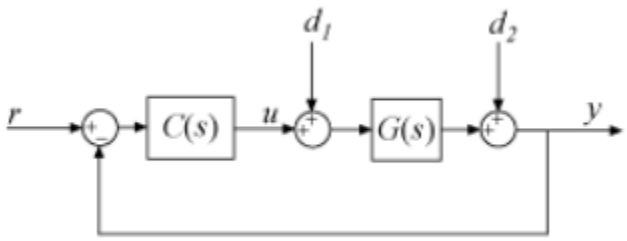
```
G = zpk(-5,[-1 -2 -3 -4],6);
```

To design a PID controller for this plant, open the **Tune PID Controller** Live Editor task. On the **Live Editor** tab, select **Task > Tune PID Controller**. This action inserts the task into your script.

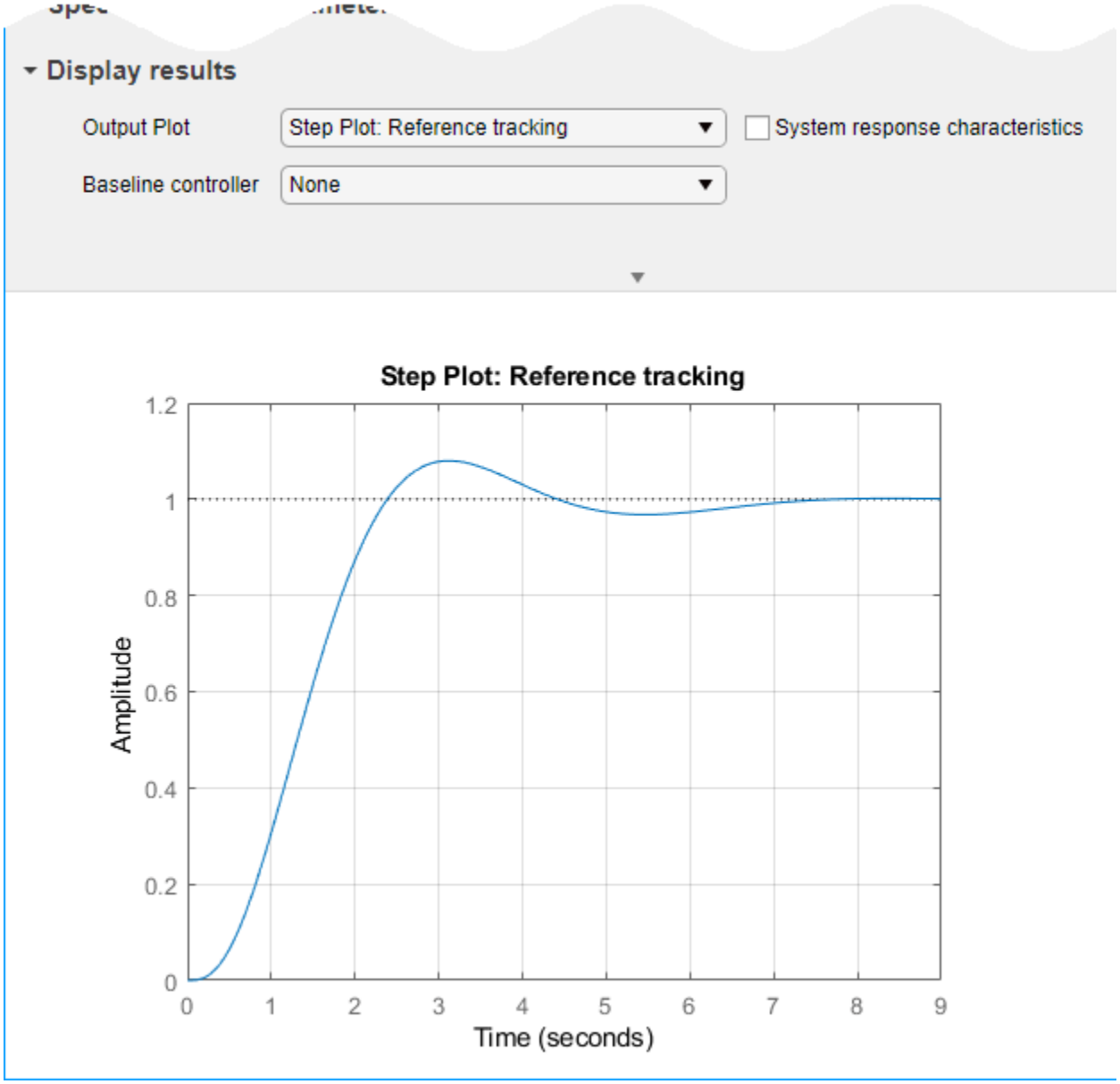


Initial Controller Design

To generate an initial PID controller design, in the **Plant** menu, select the plant you created, `G`. **Tune PID Controller** automatically generates a PI controller that balances performance and robustness, assuming the standard unit-feedback control configuration of the following diagram.



The task also generates a step response plot showing the closed-loop step response from r to y using the initial controller design.



Refine Controller Design

Select **System response characteristics** to display numeric values of some time-domain characteristics of this response.

```
    RiseTime: 1.4691
SettlingTime: 6.3562
 SettlingMin: 0.9049
 SettlingMax: 1.0799
   Overshoot: 7.9931
  Undershoot: 0
        Peak: 1.0799
    PeakTime: 3.1305
```

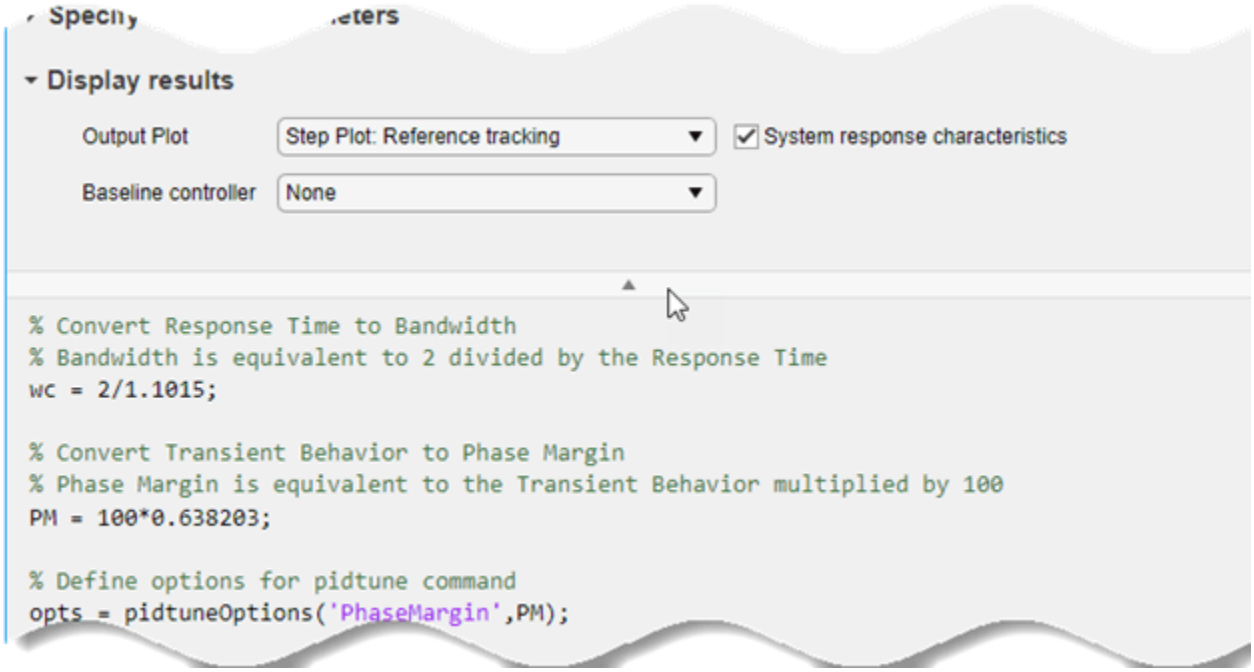
The initial controller design has a rise time of about 1.5 seconds, with about 8% overshoot. Experiment with the **Response Time** and **Transient Behavior** sliders to change the design targets and see their effect on the step response.

With a PI controller and this plant, it is difficult to decrease the response time without introducing instability or otherwise degrading system response. Try switching to a PID controller to see if you can achieve a better response time. In the **Controller Type** drop-down menu, select `PID`.

You can now reduce the response time of the controller. Experiment with the sliders again, observing the effect on the step response. For an example that shows in more detail how the **Response Time** and **Transient Behavior** sliders affect controller performance, see “Tune PID Controller to Favor Reference Tracking or Disturbance Rejection (PID Tuner)”. That example uses the **PID Tuner** app instead of the **Tune PID Controller** task in the Live Editor, but the behavior and effect of the sliders is the same in both tools.

Examine Generated Code

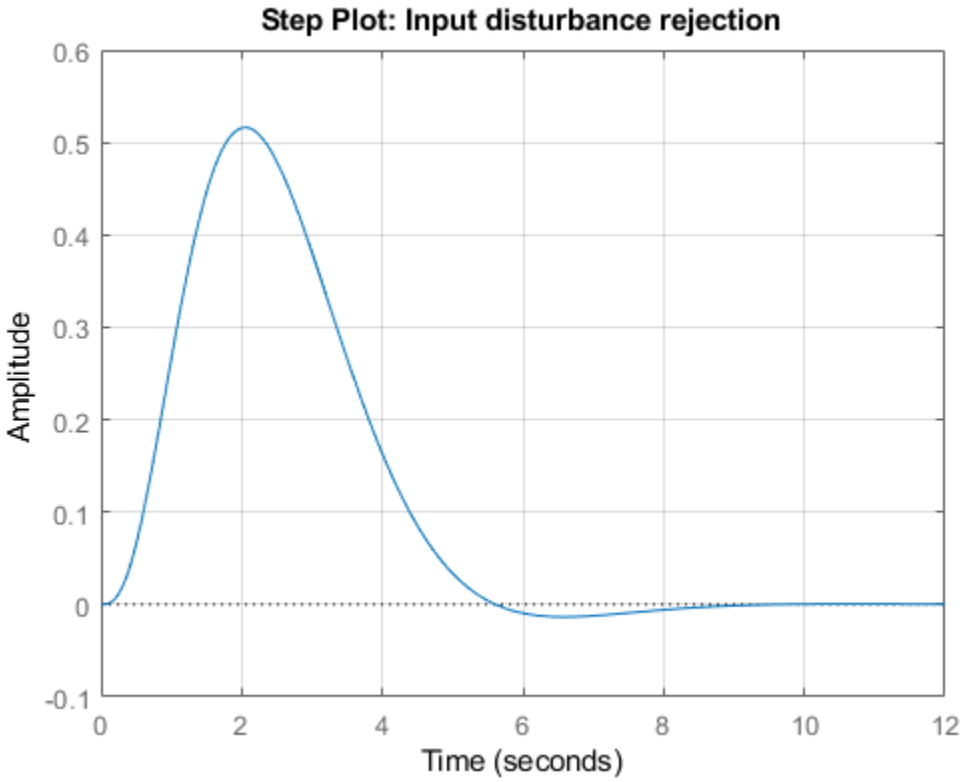
The task automatically generates code to to tune a PID controller for the plant with the specified design goals. To see the generated code, click ▼ at the bottom of the task. The task expands to show the generated code.



As you change parameters such as controller structure, performance goals, and response-plot type, the generate code updates automatically to reflect the new settings.

Examine Disturbance Rejection Performance

Suppose that you are interested in the closed-loop system response to a disturbance at the plant input. To generate a plot of the step response from d_1 to y, in the **Output Plot** drop-down menu, select `Step Plot: Input disturbance rejection`. The plot updates to show the new response. Depending on how you set the performance goals when you change the response plot, you might see a response that looks like the following.



You can now experiment again with controller parameters and observe their effect on disturbance rejection. For an example that shows in more detail how you can use the sliders and other design parameters to improve disturbance rejection performance, see “Tune PID Controller to Favor Reference Tracking or Disturbance Rejection (PID Tuner)”. That example uses the **PID Tuner** app instead of the **Tune PID Controller** task in the Live Editor, but the behavior and effect of design parameters is the same in both tools.

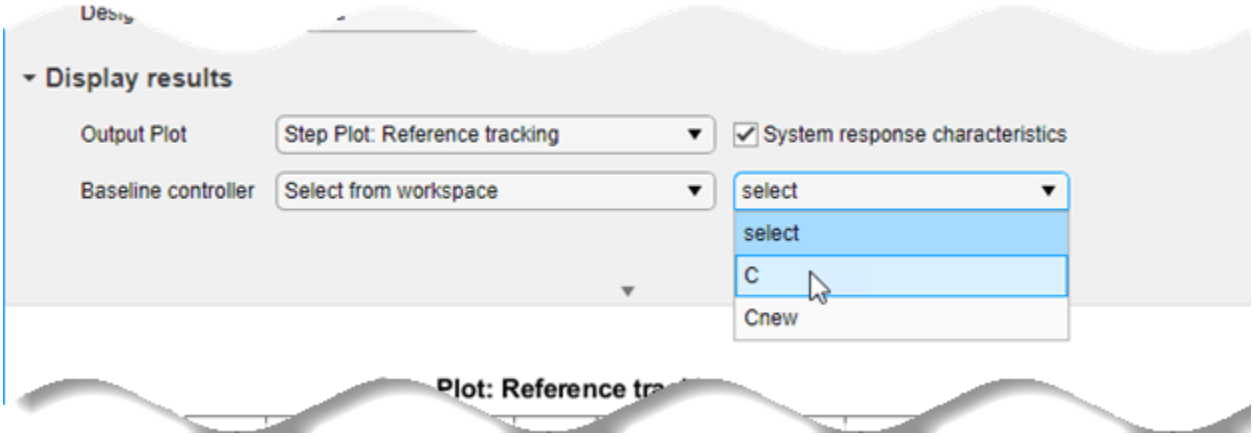
Compare Two Controller Designs

Tune PID Controller automatically writes the tuned controller to the MATLAB® workspace as a `pid`, `pidstd`, `pid2`, or `pidstd2` model object, whichever is appropriate for your controller settings. The task stores the controller using the variable name specified in the task summary line. By default, that variable name is `C`. When you change controller settings, performance goals, or other tuning parameters, by default the task writes over the variable `C`.

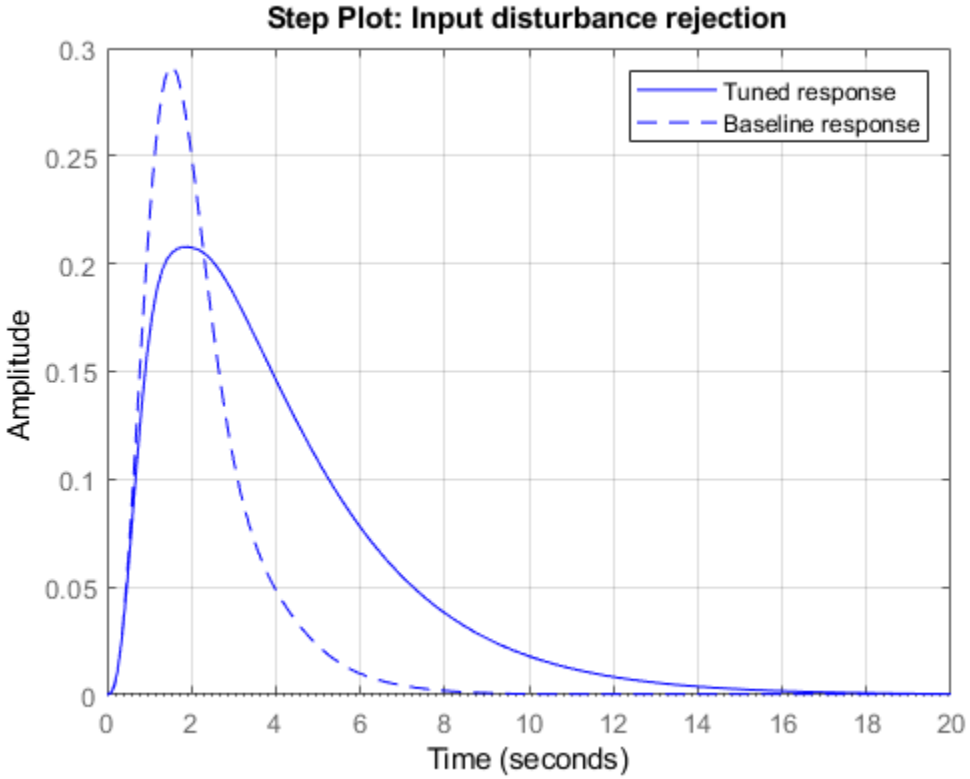
You can save a controller design to use as a baseline for comparison while you experiment further with controller types, performance goals, and other settings. To do so, type a new variable name in the task summary line. For instance, change the output controller name to `Cnew`.

Now, the current design is stored in the MATLAB workspace as `C`. Any further changes to the design are stored as `Cnew`.

To use `C` as a baseline for comparison, in the **Baseline controller** menu, choose `Select from workspace`. Then, select `C` in the menu that appears.



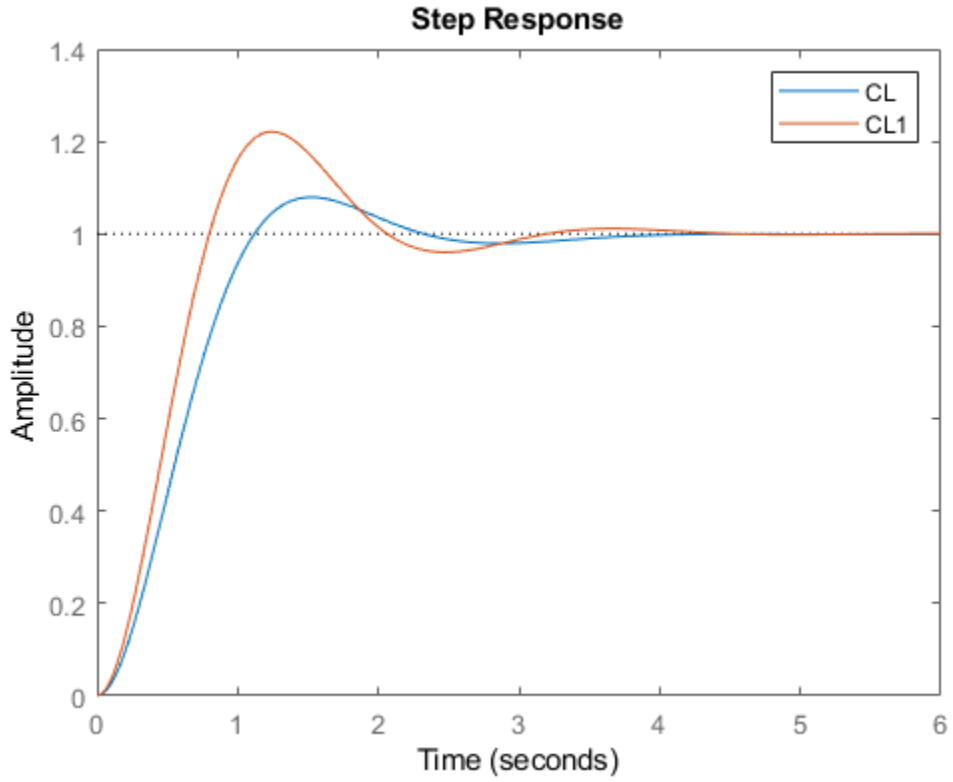
Now, as you experiment further with the controller design, the plot displays both the system response with controller `C` (dotted line) and with controller `Cnew` (solid line).



Use the Controller

Because the **Tune PID Controller** task saves the controller to the MATLAB workspace, you can use the controller as you would use any other PID model object for control design and analysis. For instance, examine the controller performance against a slightly different plant model, to get a sense of the robustness of the closed-loop system against parameter variation.

```
G1 = zpk(-5,[-0.75 -2 -3 -4],8);
CL1 = getPIDLoopResponse(C,G1,'closed-loop');
CL = getPIDLoopResponse(C,G,'closed-loop');
step(CL,CL1)
```



See Also

Live Editor Tasks
Tune PID Controller

More About

- “Tune PID Controller from Measured Plant Data in the Live Editor” on page 11-80

Tune PID Controller from Measured Plant Data in the Live Editor

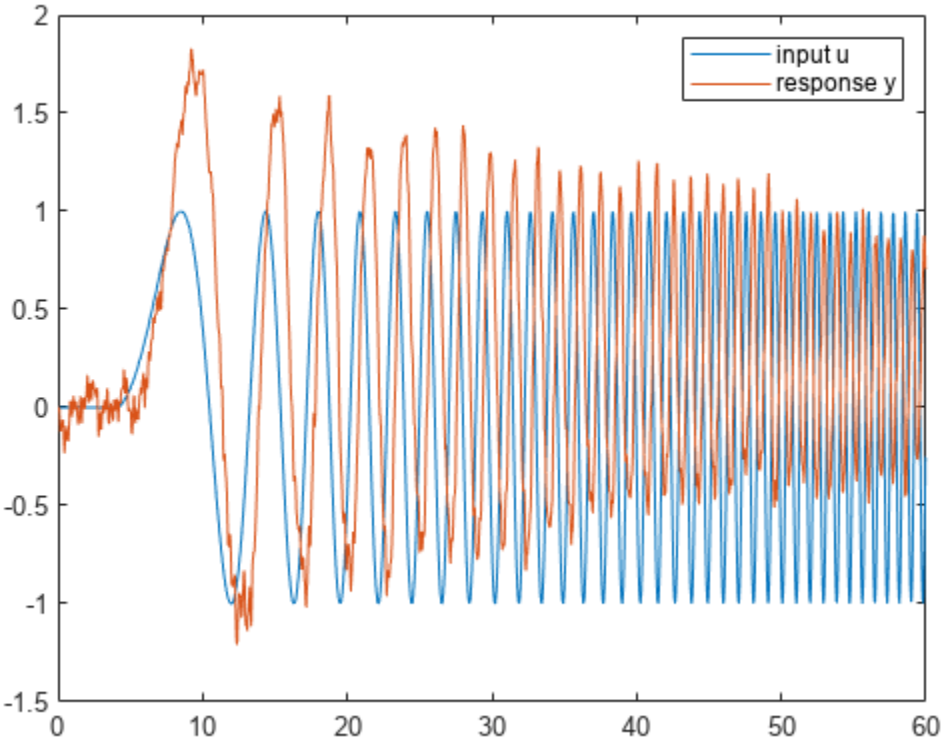
This example shows how to use Live Editor tasks to tune a PID controller for a plant, starting from the measured plant response to a known input signal. In this example, you use the **Estimate State-Space Model** task to generate code for estimating a parametric plant model. Then, you use the **Convert Model Rate** task to discretize the continuous-time identified model. Finally, you use the **Tune PID Controller** task to design a PID controller to achieve a closed-loop response that meets your design requirements. (Using **Estimate State-Space Model** requires a System Identification Toolbox™ license.)

Live Editor tasks let you interactively iterate on parameters and settings while observing their effects on the result of your computation. The tasks then automatically generate MATLAB® code that achieves the displayed results. To experiment with the Live Editor tasks in this script, open this example. For more information about Live Editor Tasks generally, see “Add Interactive Tasks to a Live Script”.

Load Plant Data

Load the measured input-output data. In this example, the data consists of the response of an engine to a chirp input. The input `u` is a vector containing the input signal sampled every 0.04 seconds. The output vector `y` contains the corresponding measured response.

```
load IdentPlantPIDExample u y
t = 0.04*(0:length(u)-1);
plot(t,u,t,y)
legend('input u','response y')
```

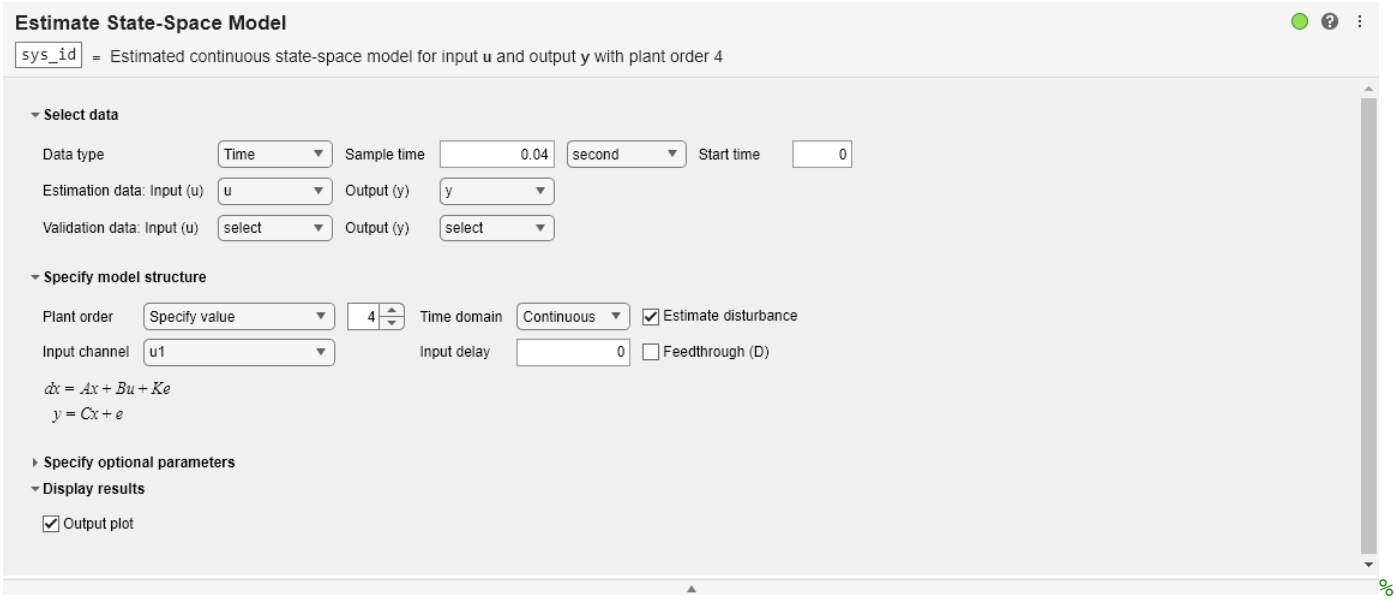


Estimate State-Space Model

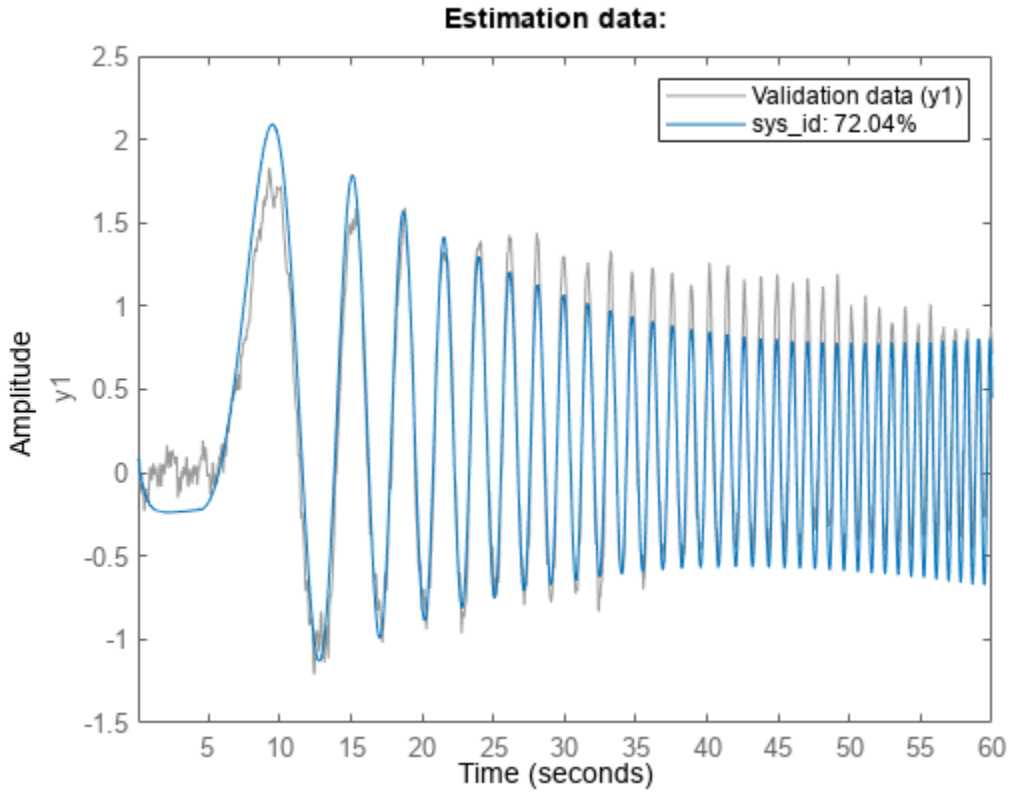
To estimate a state-space model from this data, use the Estimate State-Space Model (System Identification Toolbox) Live Editor task. You can insert a task into your script using the **Task** menu in the Live Editor. In this script, **Estimate State-Space Model** is already inserted. Open the example to experiment with the task.

To perform the estimation, in the task, specify the input and output signals you loaded, `u` and `y`, and the sample time, 0.04 seconds. (For this example, you do not have validation data.) You also need to specify a plant order. Typically, you can guess the plant order based upon your knowledge of your system. In general, you want to use the lowest plant order that gives a reasonably good estimation fit. In the **Estimate State-Space Model** task, experiment with different plant order values and observe the fit result, displayed in the output plot. For details about the available options and parameters, see the Estimate State-Space Model (System Identification Toolbox) task reference page.

As you vary parameters in the task, it automatically updates the generated code for performing the estimation and creating the plot. (To see the generated code, click ▼ at the bottom of the task.)

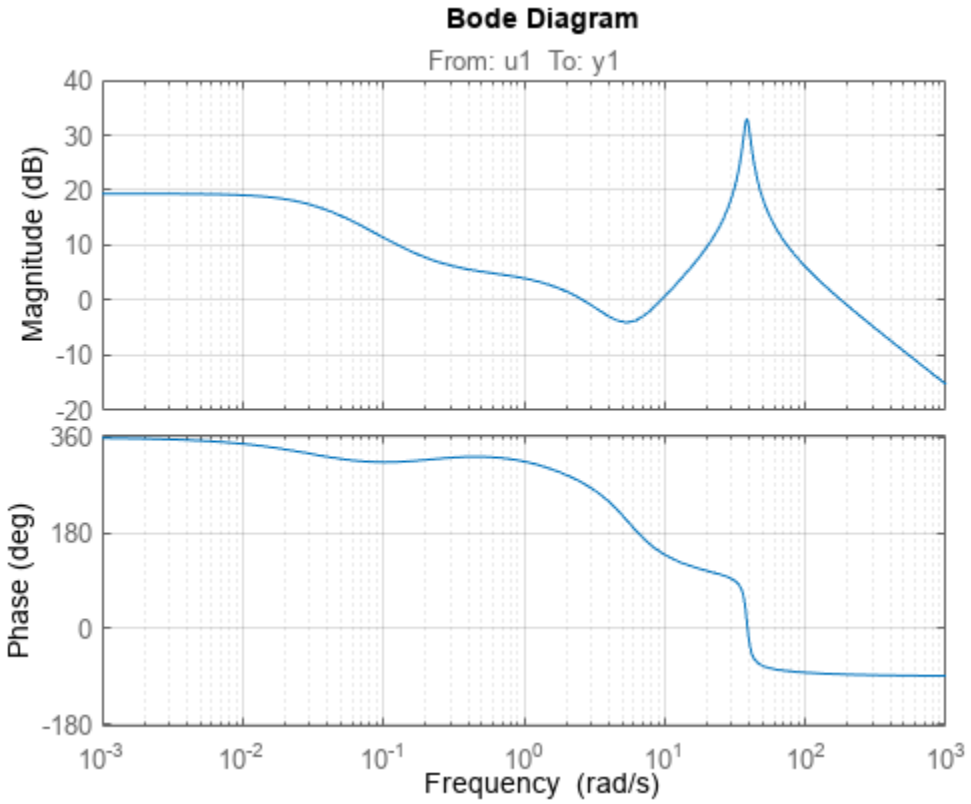



```
% Create o
estimationData = iddata(y,u,0.04);

% Estimate state-space model
sys_id = ssest(estimationData,4);

% Display results
compare(estimationData,sys_id);
clear estimationData;
title('Estimation data:');
```



For this example, at plant order 4, the estimation fit is about 72%. Try increasing the plant order to see that doing so does not improve the fit much. Therefore, use the fourth-order plant. The code produces an identified state-space model with the variable name that you type into the summary line of the **Estimate State-Space Model** task. For this example, use `sys_id`. After you finish experimenting with the task, the identified state-space model `sys_id` is in the MATLAB® workspace, and you can use it for additional design and analysis in the same way you use any other LTI model object. For instance, examine the frequency response of the identified state-space model `sys_id`.

```
bode(sys_id)
grid on
```

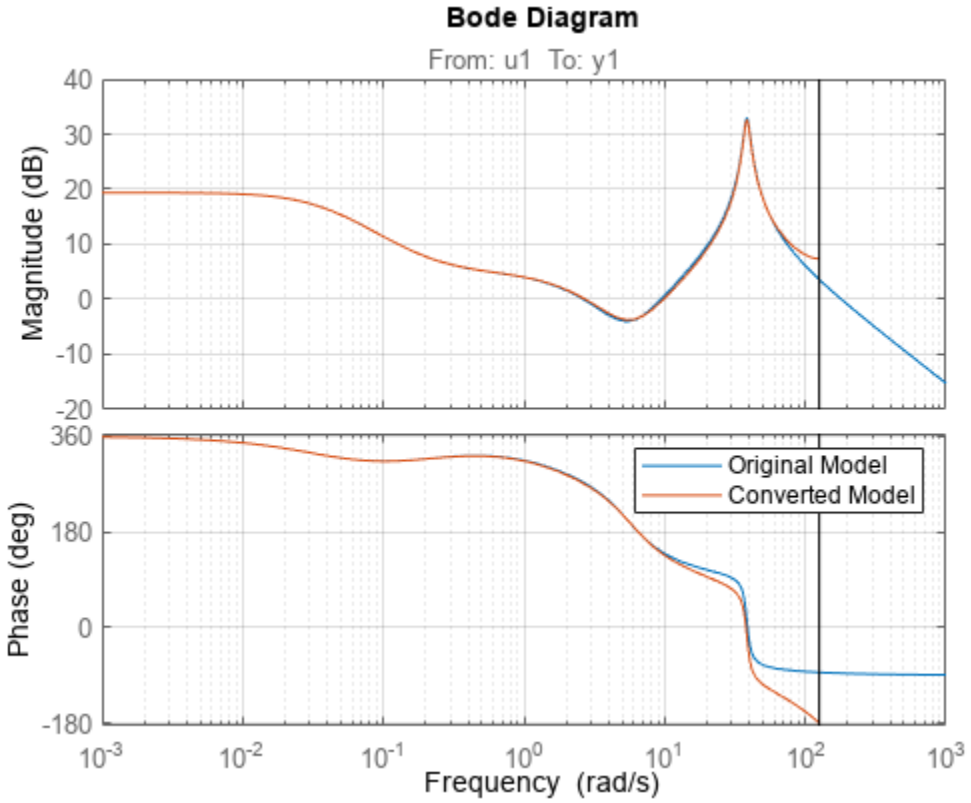


Discretize Model

Suppose that you want to discretize this model before you design a PID controller for it. To do so, use the Convert Model Rate task. In the task, select the identified model `sys_id`. Specify a sample time fast enough to accommodate the resonance in the identified model response, such as 0.025 s. You can also choose a different conversion method to better match the frequency response in the vicinity of the resonance. For instance, try setting **Method** to `Bilinear (Tustin) approximation` with a prewarp frequency of 38.4 rad/s, the location of the peak response. As you experiment with settings in the task, compare the original and converted models in a Bode plot to make sure you are satisfied with the match. (For more information about the parameters and options, see the Convert Model Rate task reference page.)

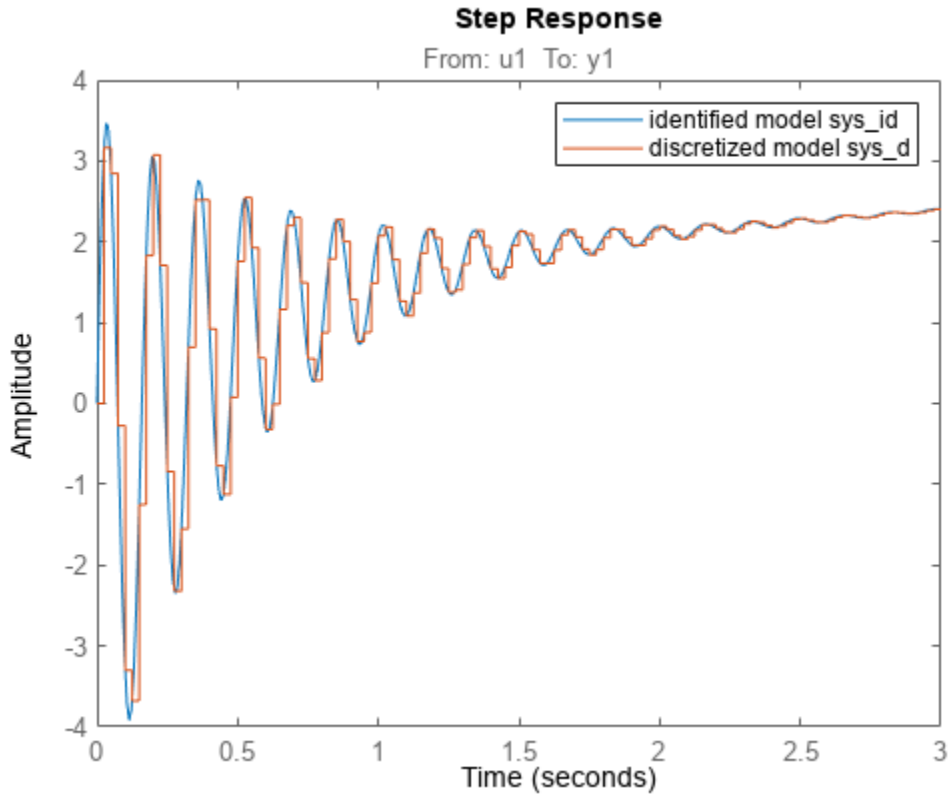
Convert Model Rate generates code that produces the discretized model with the variable name that you type into the summary line of the task. For this example, use `sys_d`.


```
% Convert
sys_d = c2d(sys_id,0.025);

% Visualize the results
bodeplot(sys_id,sys_d);
legend('Original Model','Converted Model');
grid on;
```



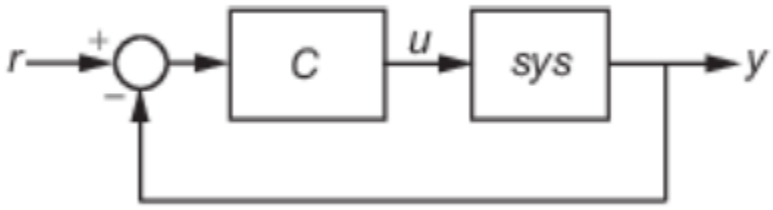
To confirm that the discretized model captures the transient response due to the resonance, compare the first few seconds of the step responses of the original identified model `sys_id` and the discretized model `sys_d`.

```
step(sys_id,sys_d,3)
legend('identified model sys_id','discretized model sys_d')
```

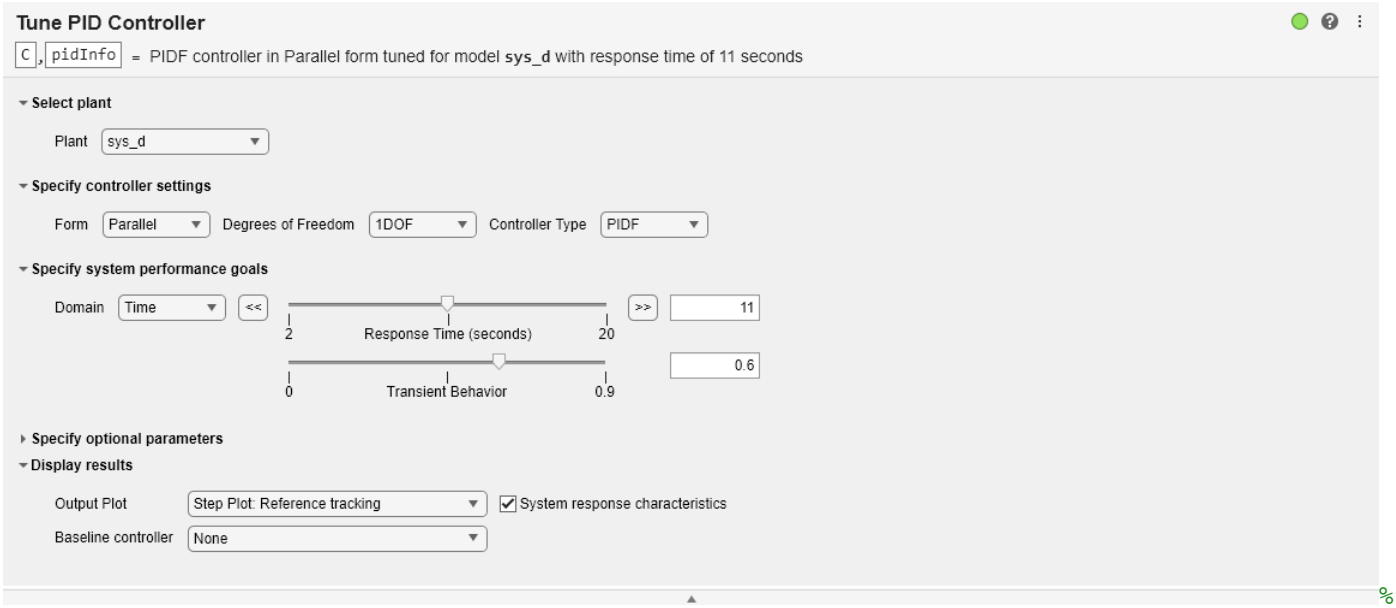


Tune Controller for Discretized Plant Model

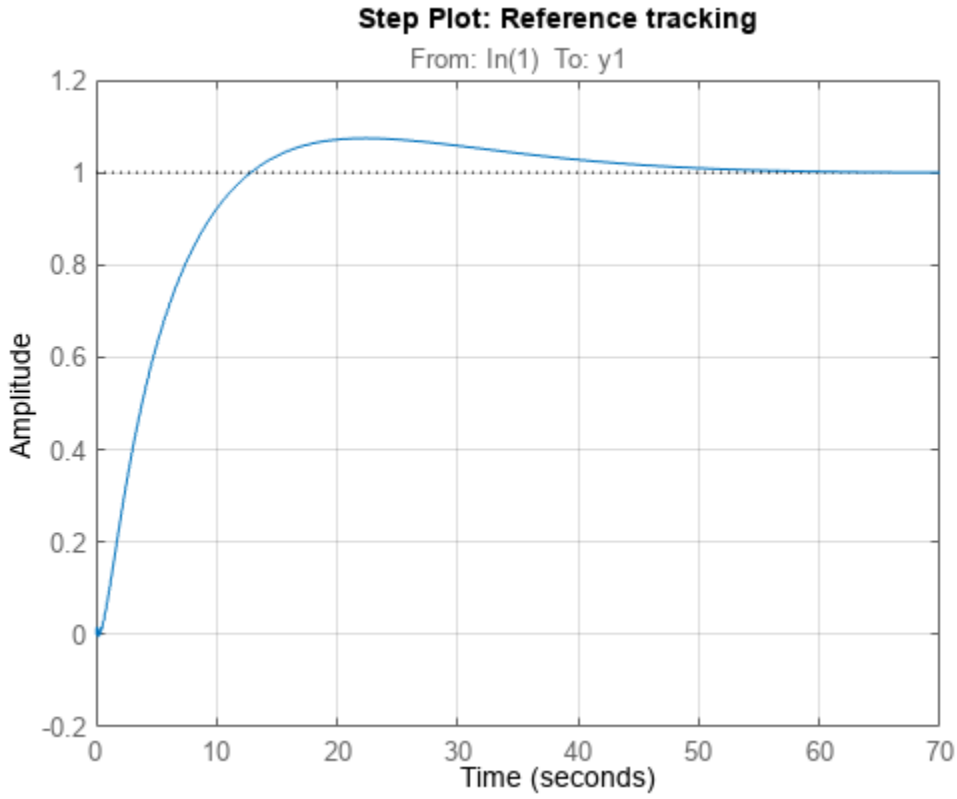
Finally, use the **Tune PID Controller** task to generate code for tuning a PI or PID controller for the discretized plant `sys_d`. The task designs a PID controller for a specified plant assuming the standard unit-feedback control configuration of the following diagram.



In the task, select `sys_d` as the plant and experiment with settings such as controller type and response time. As you change settings, select output plots on which to observe the closed-loop response generated by the task. Check **System response characteristics** to generate a numerical display of closed-loop step-response characteristics such as rise time and overshoot.




```
% Convert
% Bandwidth is equivalent to 2 divided by the Response Time
wc = 2/11;

% PID tuning algorithm for linear plant model
[C,pidInfo] = pidtune(sys_d,'PIDF',wc);

% Clear Temporary Variables
clear wc

% Get desired loop response
Response = getPIDLoopResponse(C,sys_d,'closed-loop');

% Plot the result
stepplot(Response)
title('Step Plot: Reference tracking')
grid on
```




```
% Display system response characteristics
disp(stepinfo(Response))
```

```
          RiseTime: 8.3250
    TransientTime: 43.2750
     SettlingTime: 43.3500
      SettlingMin: 0.9008
      SettlingMax: 1.0741
        Overshoot: 7.4059
       Undershoot: 0.5440
             Peak: 1.0741
         PeakTime: 22.3250
```

```
% Clear Temporary Variables
clear Response
```

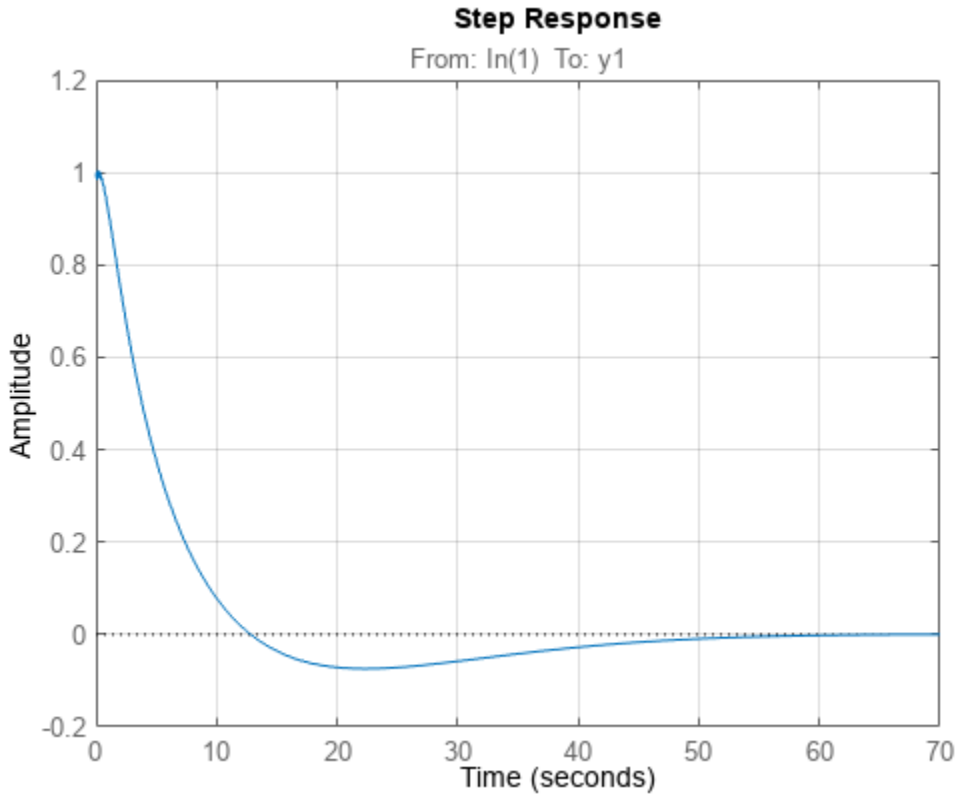
For this example, suppose that you want the closed-loop system to settle within 50 seconds, and that the system can tolerate overshoot of no more than 10%. Adjust controller settings such as **Controller Type** and **Response Time** to achieve that target. For more information about the available parameters and options, see the Tune PID Controller task reference page.

Further Analysis of Design

Like the other Live Editor tasks, **Tune PID Controller** generates code that produces a tuned controller with the variable name that you type into the summary line of the task. For this example,

use `C`. The tuned controller `C` is a `pid` model object in the MATLAB workspace that you can use for further analysis. For example, compute the closed-loop response to a disturbance at the output of the plant `sys_d`, using this controller. Examine the response and its characteristics.

```
CLdist = getPIDLoopResponse(C,sys_d,"output-disturbance");
step(CLdist)
grid on
```



You can use the models `sys_id`, `sys_d`, and `C` for any other control design or analysis tasks.

See Also

Live Editor Tasks
Estimate State-Space Model | Convert Model Rate | Tune PID Controller

More About

- “PID Controller Design in the Live Editor” on page 11-73

Design PID Controller for Disturbance Rejection Using PID Tuner

This example shows how to design a PI controller with good disturbance rejection performance using the PID Tuner app. The example also shows how to design an ISA-PID controller for both good disturbance rejection and good reference tracking.

Launching the PID Tuner with Initial PID Design

The plant model is

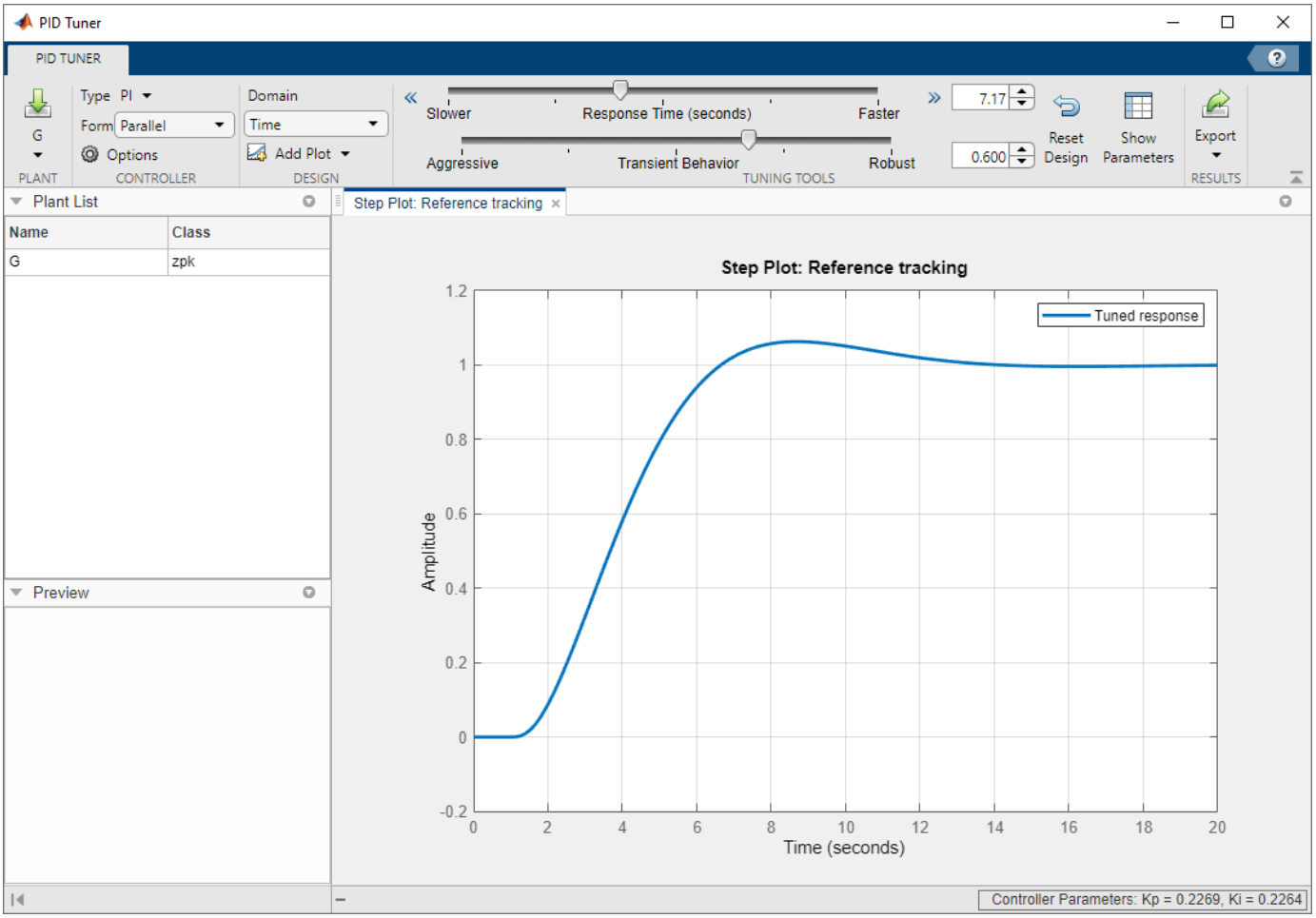
$$G(s) = \frac{6(s+5)e^{-s}}{(s+1)(s+2)(s+3)(s+4)}$$

```
G = zpk(-5,[-1 -2 -3 -4],6,'OutputDelay',1);
G.InputName = 'u';
G.OutputName = 'y';
```

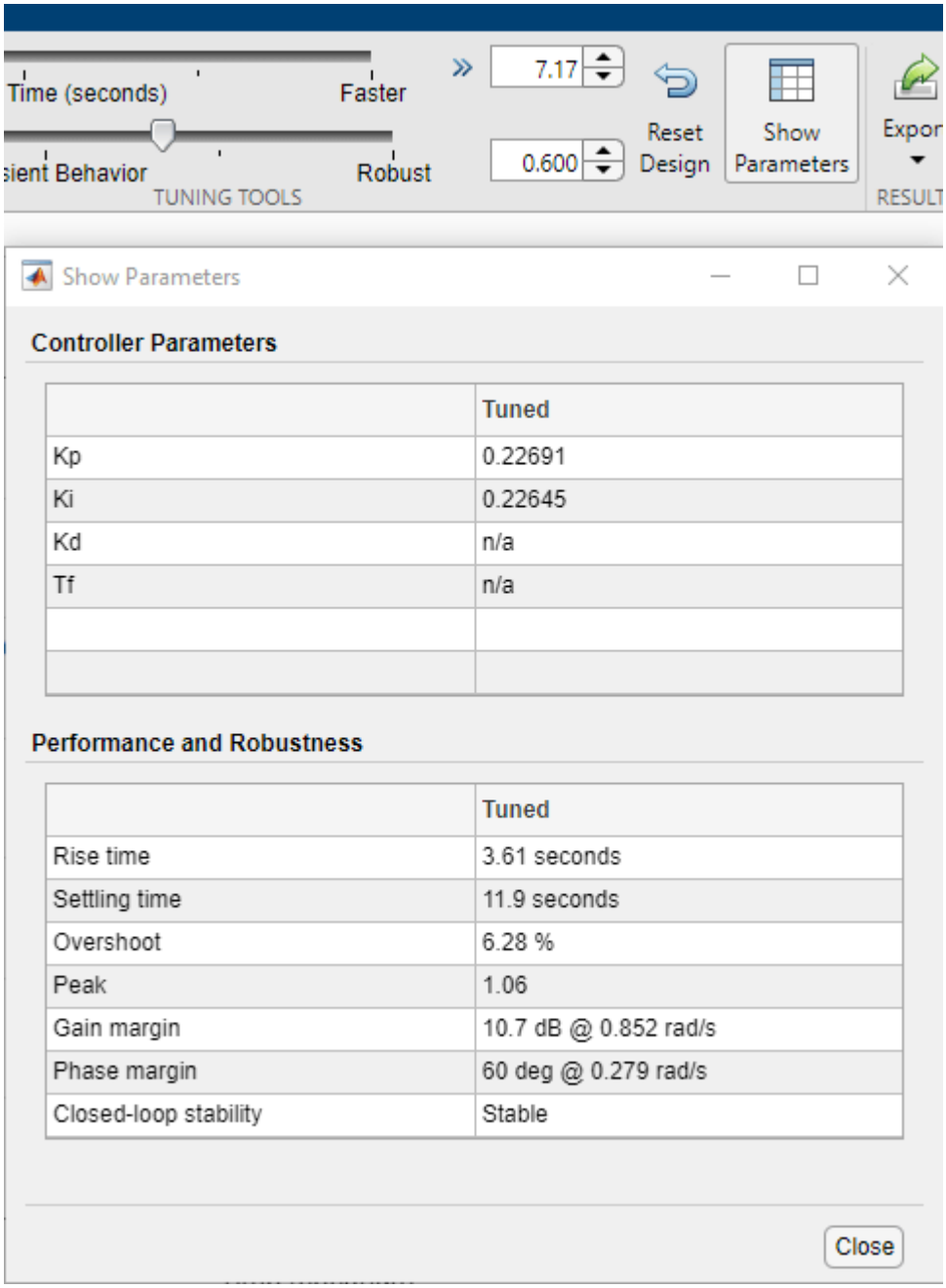
Use the following command to launch the PID Tuner app to design a PI controller in parallel form for plant G.

```
pidTuner(G,'pi')
```

The PID Tuner app automatically designs an initial PI controller.



To display the controller gains and performance metrics, click **Show Parameters**.



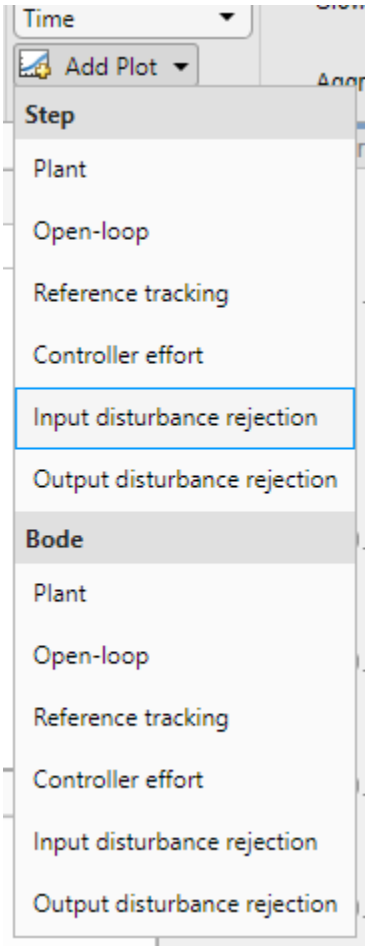
For step reference tracking, the settling time is about 12 seconds and the overshoot is about 6.3 percent, which is acceptable for this example.

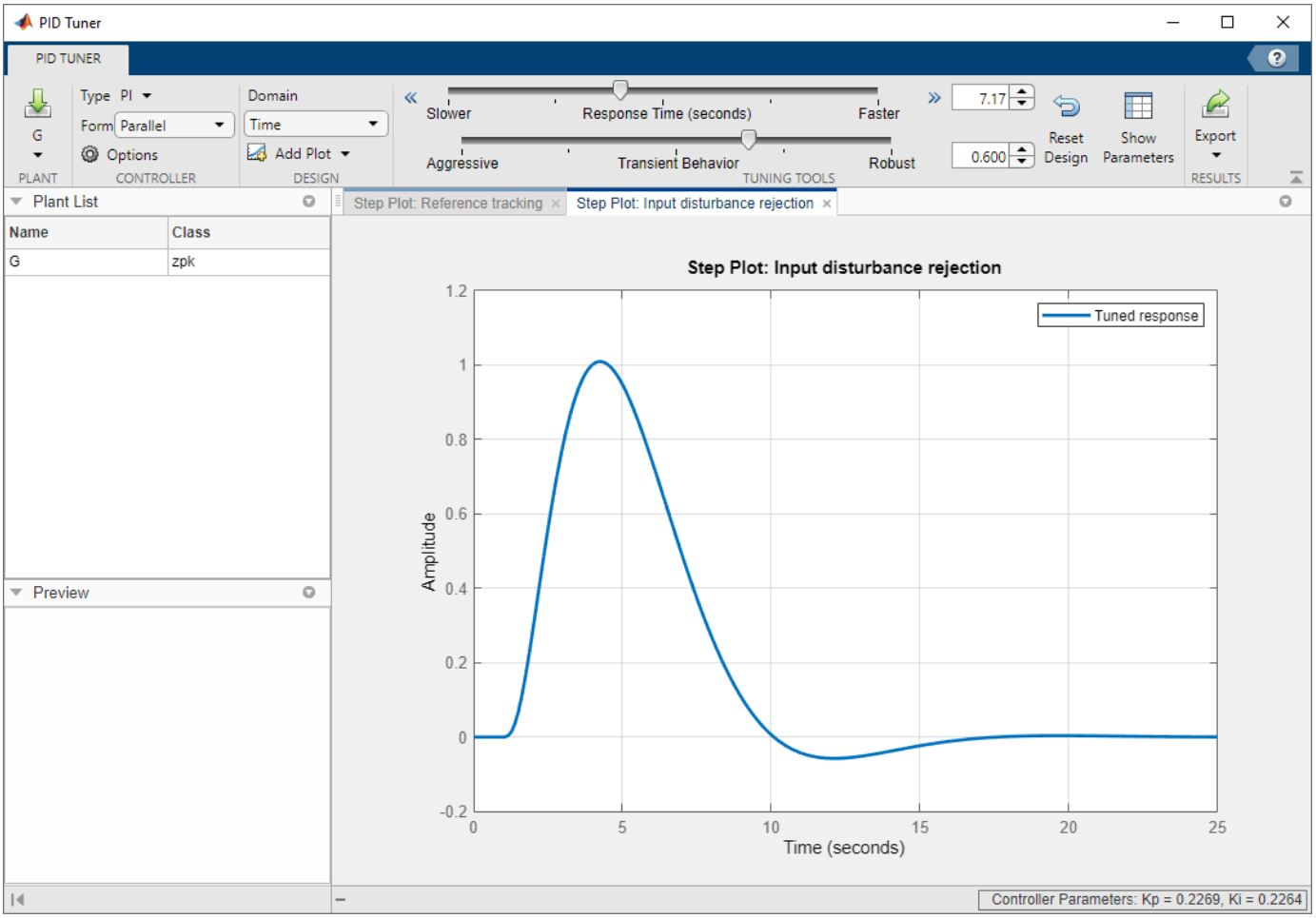
Tuning PID for Disturbance Rejection

Assume that a step disturbance occurs at the plant input and the main purpose of the PI controller is to reject this disturbance quickly. In the rest of this section, we will show how to design the PI controller for better disturbance rejection in the PID Tuner. We also expect that the reference tracking performance is degraded as disturbance rejection performance improves.

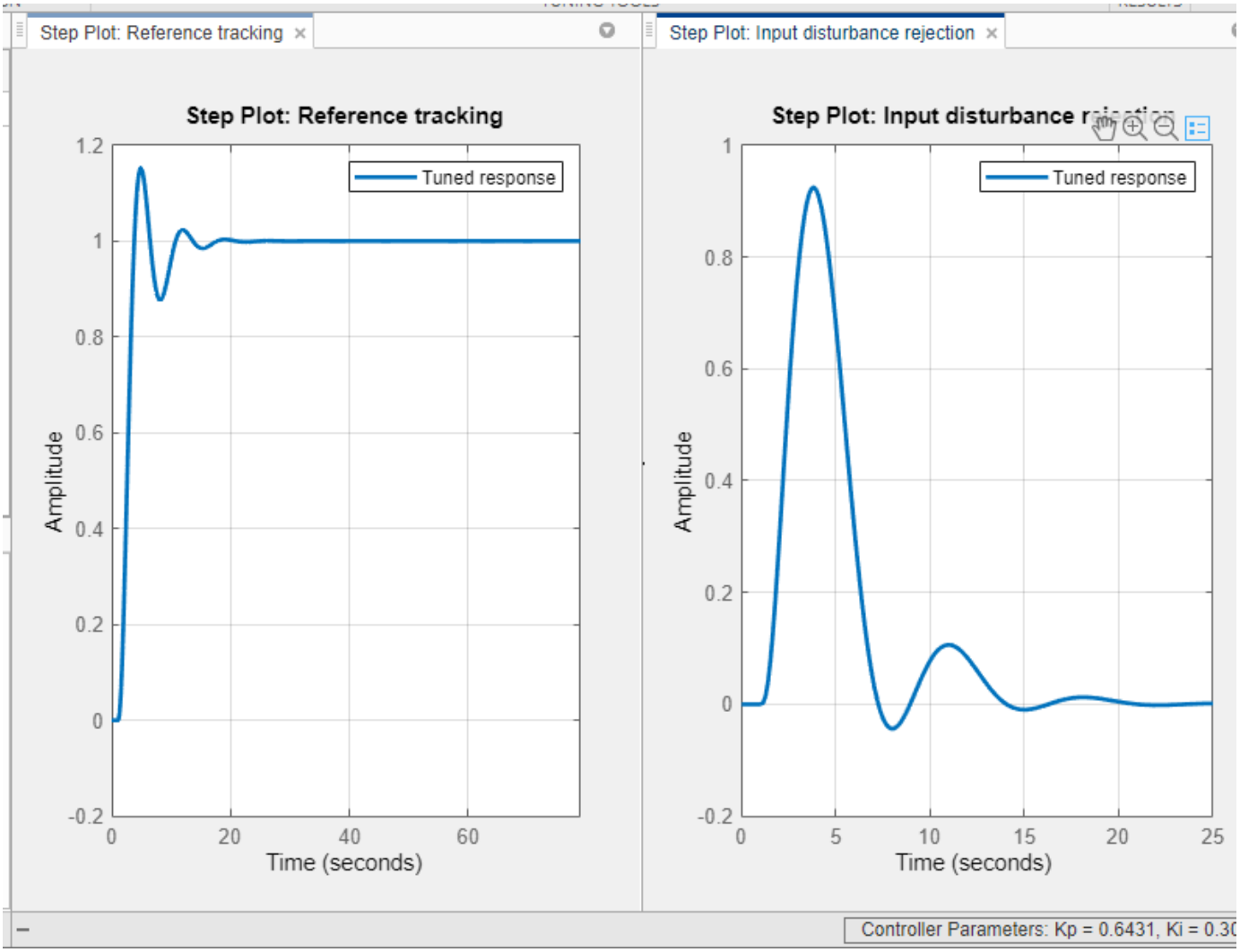
Because the attenuation of low frequency disturbance is inversely proportional to integral gain Ki, maximizing the integral gain is a useful heuristic to obtain a PI controller with good disturbance rejection. For background, see Karl Astrom et al., "Advanced PID Control", Chapter 4 "Controller Design", 2006, The ISA Society.

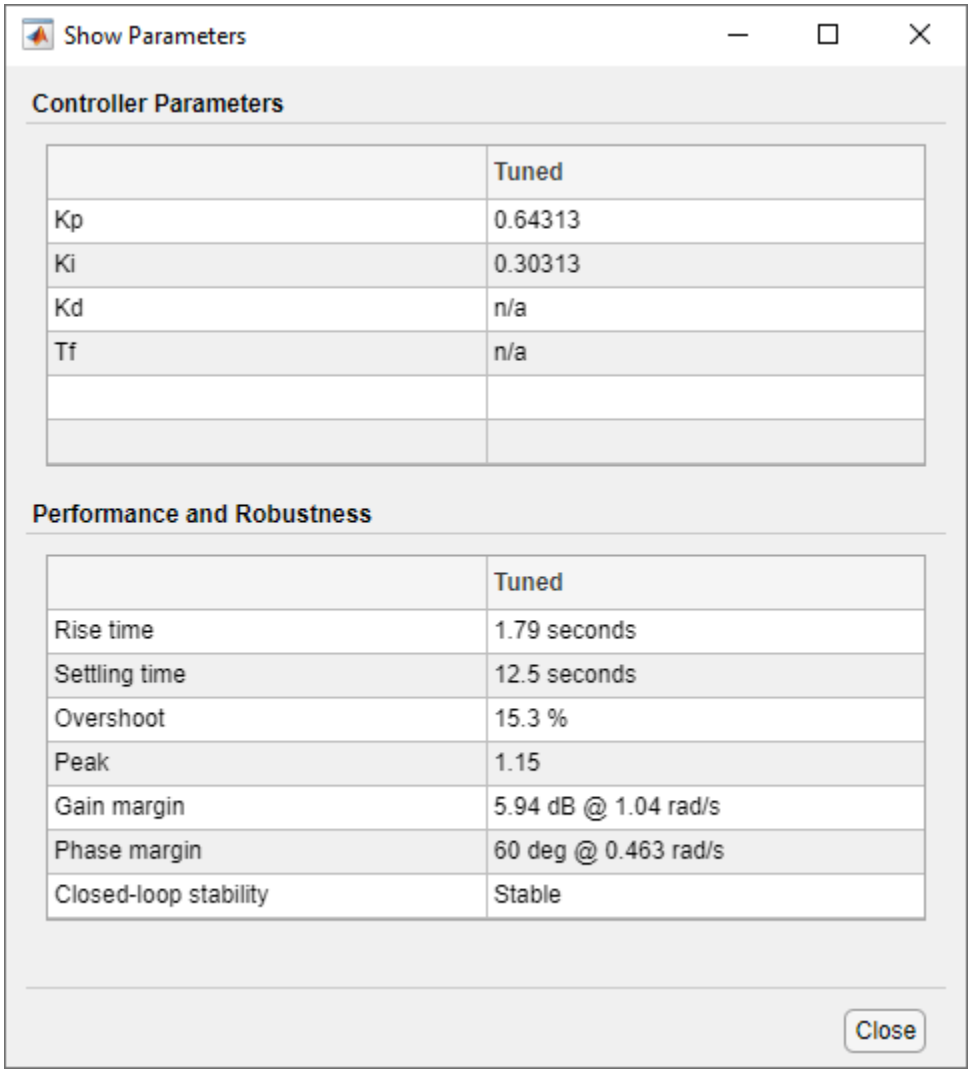
Click **Add Plot and** select **Input disturbance rejection** to plot the input disturbance step response. The peak deviation is about 1 and it settles to less than 0.1 in about 9 seconds.





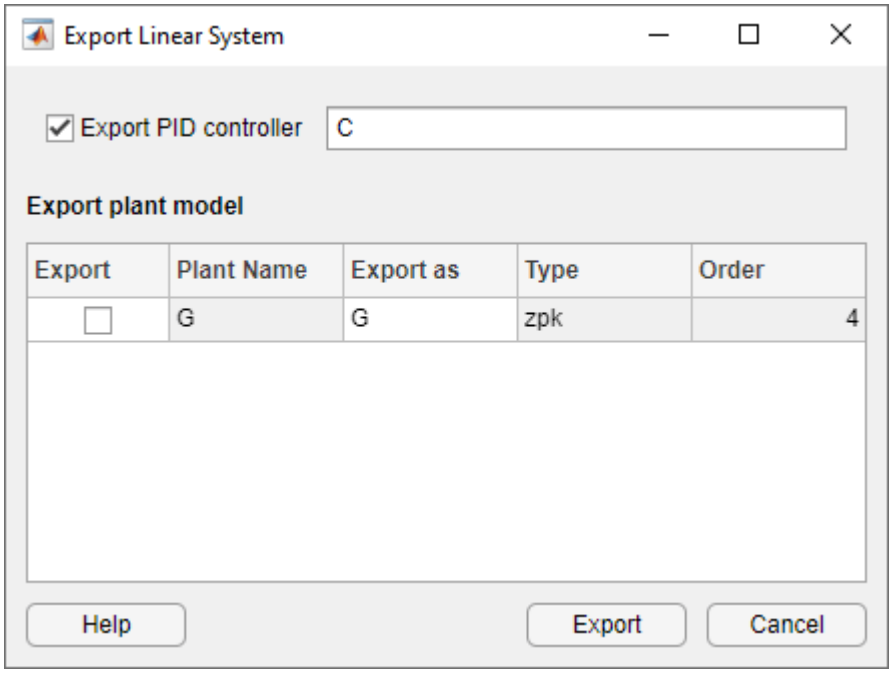
Tile the plots to show both the reference tracking and input disturbance responses. Move the response time slider to the right to increase the response speed (open loop bandwidth). The Ki gain in the **Controller parameters** table first increases and then decreases, with the maximum value occurring at 0.3. When Ki is 0.3, the peak deviation is reduced to 0.9 (about 10% improvement) and it settles to less than 0.1 in about 6.7 seconds (about 25% improvement).


Step Plot: Reference tracking
Step Plot: Input disturbance rejection
Step Plot: Reference tracking
Tuned response
Amplitude
Time (seconds)
Step Plot: Input disturbance rejection
Tuned response
Amplitude
Time (seconds)
Controller Parameters: Kp = 0.6431, Ki = 0.30



Because we increased the bandwidth, the step reference tracking response becomes more oscillatory. Additionally the overshoot exceeds 15 percent, which is usually unacceptable. This type of performance trade off between reference tracking and disturbance rejection often exists because a single PID controller is not able to satisfy both design goals at the same time.

Click **Export** to export the designed PI controller to the MATLAB Workspace. The controller is represented by a PID object and you need it to create an ISA-PID controller in the next section.



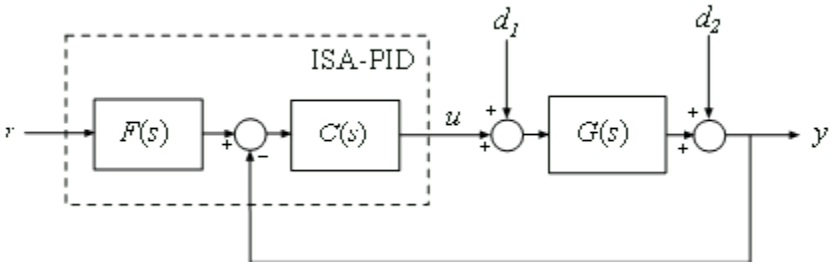
You can also manually create the same PI controller in MATLAB Workspace by using the **pid** command. In this command you can directly specify the Kp and Ki gains obtained from the parameter table of the PID Tuner.

```
C = pid(0.64362,0.30314);
C.InputName = 'e';
C.OutputName = 'u';
C
```

```
C =

             1
  Kp + Ki * ---
             s

  with Kp = 0.644, Ki = 0.303

Continuous-time PI controller in parallel form.
```

Extending PID controller to ISA-PID Controller

A simple solution to make a PI controller perform well for both reference tracking and disturbance rejection is to upgrade it to an ISA-PID controller. It improves reference tracking response by providing an additional tuning parameters **b** that allows independent control of the impact of the reference signal on the proportional action.



In the above ISA-PID structure, there is a feedback controller C and a feed-forward filter F. In this example, C is a regular PI controller in parallel form that can be represented by a PID object:

$$C(s) = pid(K_p, K_i) = K_p + \frac{K_i}{s}$$

F is a pre-filter that involves Kp and Ki gains from C plus the setpoint weight **b**:

$$F(s) = \frac{bK_p s + K_i}{K_p s + K_i}$$

Therefore the ISA-PID controller has two inputs (r and y) and one output (u).

Set-point weight **b** is a real number between 0 and 1. When it decreases, the overshoot in the reference tracking response is reduced. In this example, **b** is chosen to be 0.7.

```
b = 0.7;
% The following code constructs an ISA-PID from F and C
F = tf([b*C.Kp C.Ki],[C.Kp C.Ki]);
F.InputName = 'r';
F.OutputName = 'uf';
Sum = sumblk('e','uf','y','+-');
ISAPID = connect(C,F,Sum,{'r','y'},'u');
tf(ISAPID)
```

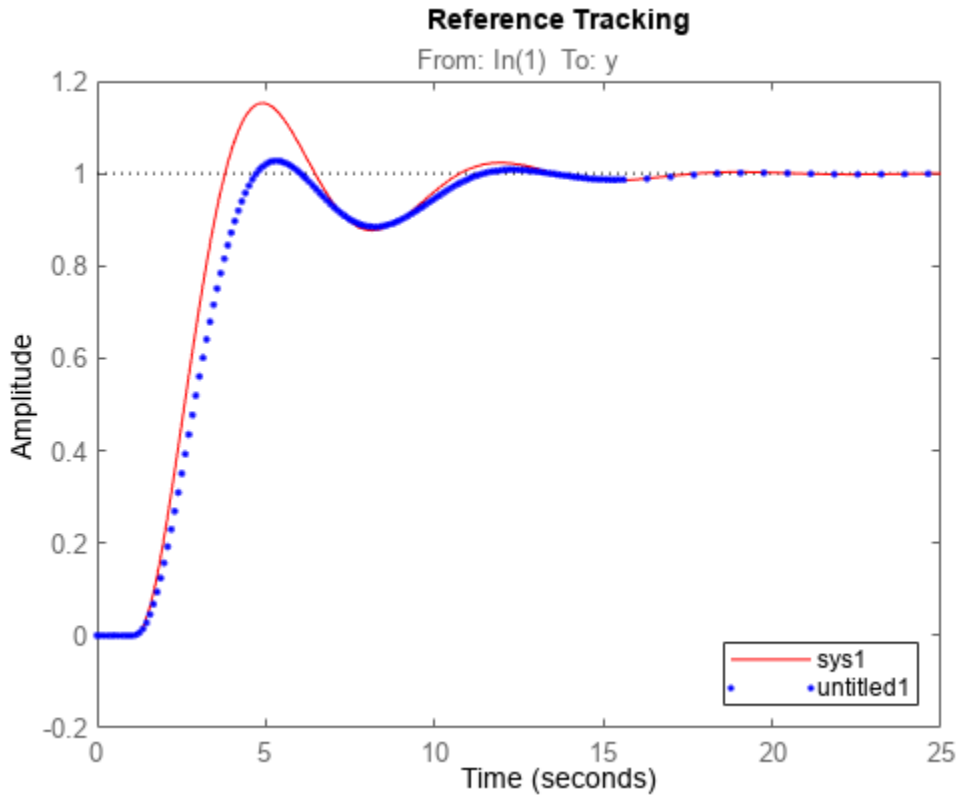
```
ans =

  From input "r" to output "u":
  0.4505 s^2 + 0.5153 s + 0.1428
  ------------------------------
          s^2 + 0.471 s

  From input "y" to output "u":
  -0.6436 s - 0.3031
  ------------------
          s

Continuous-time transfer function.
```

Compare Performance

The reference tracking response with ISA-PID controller has much less overshoot because setpoint weight **b** reduces overshoot.

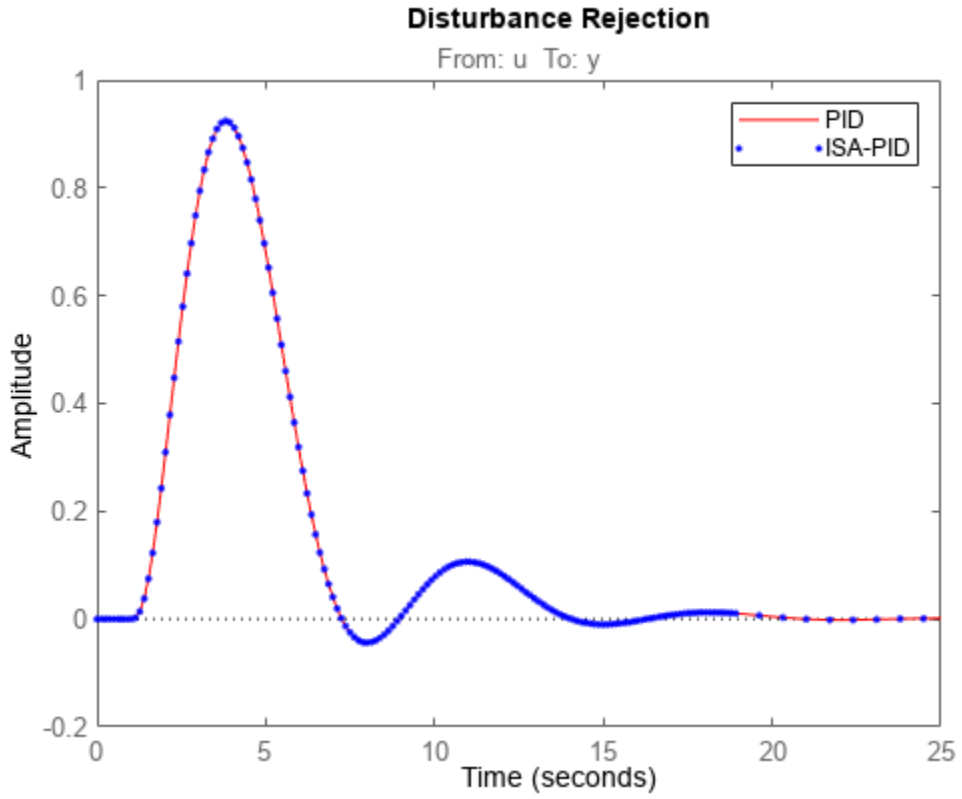
```
% Closed-loop system with PI controller for reference tracking
sys1 = feedback(G*C,1);
% Closed-loop system with ISA-PID controller
```

```
sys2 = connect(ISAPID,G,{'r','u'},'y');
% Compare responses
step(sys1,'r-',sys2(1),'b.');
legend('show','location','southeast')
title('Reference Tracking')
```



The disturbance rejection responses are the same because setpoint weight **b** only affects reference tracking.

```
% Closed-loop system with PI controller for disturbance rejection
sys1 = feedback(G,C);
% Compare responses
step(sys1,'r-',sys2(2),'b.');
legend('PID','ISA-PID');
title('Disturbance Rejection')
```



See Also

PID Tuner

Related Examples

- “Designing PID Controllers with PID Tuner”
- “PID Controller Design for Fast Reference Tracking”
- “Tune PID Controller to Favor Reference Tracking or Disturbance Rejection (PID Tuner)”

Temperature Control in a Heat Exchanger

This example shows how to design feedback and feedforward compensators to regulate the temperature of a chemical reactor through a heat exchanger.

Heat Exchanger Process

A chemical reactor called "stirring tank" is depicted below. The top inlet delivers liquid to be mixed in the tank. The tank liquid must be maintained at a constant temperature by varying the amount of steam supplied to the heat exchanger (bottom pipe) via its control valve. Variations in the temperature of the inlet flow are the main source of disturbances in this process.

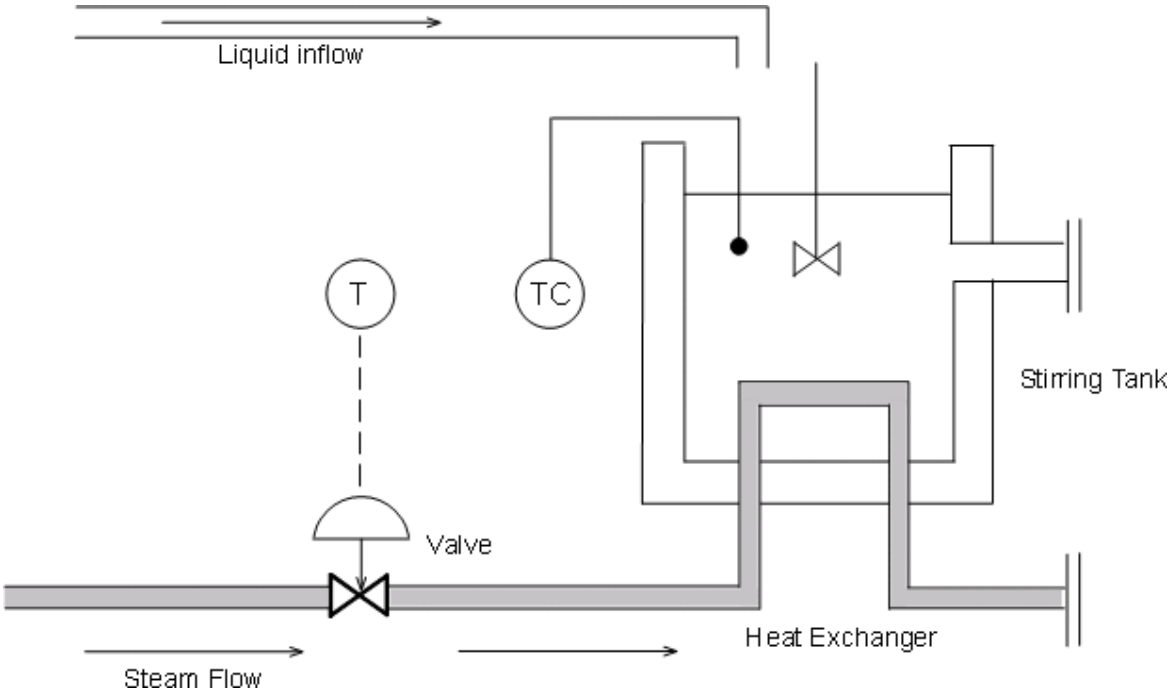
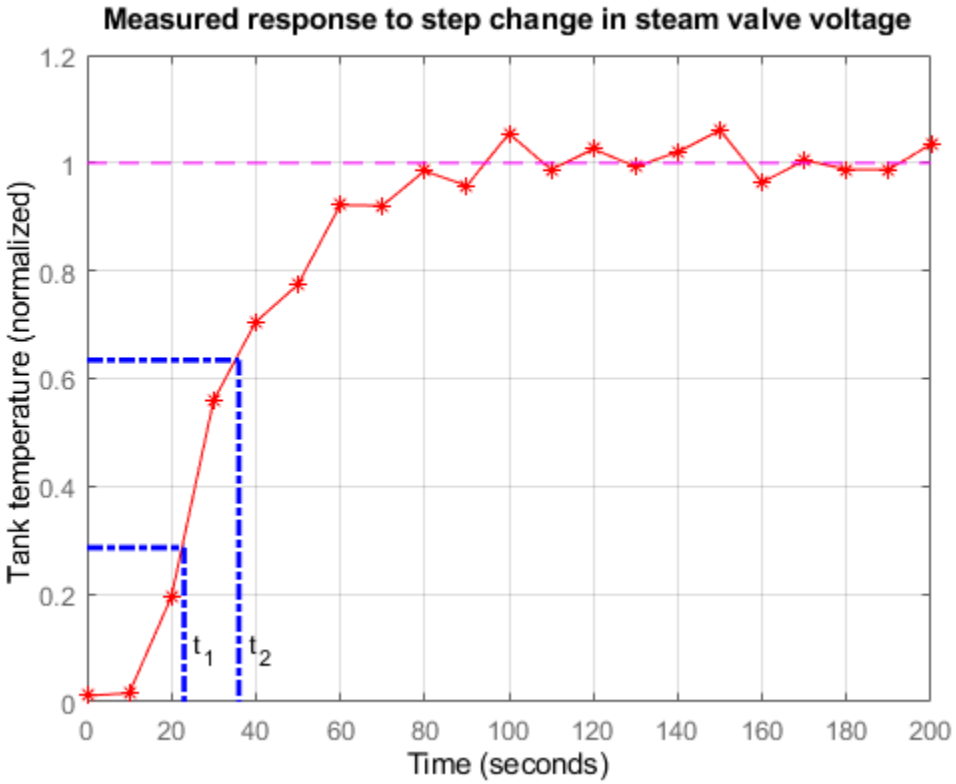


Figure 1: Stirring Reactor with Heat Exchanger.

Using Measured Data to Model The Heat Exchanger Dynamics

To derive a first-order-plus-deadtime model of the heat exchanger characteristics, inject a step disturbance in valve voltage V and record the effect on the tank temperature T over time. The measured response in normalized units is shown below:

```
heatex_plotdata
title('Measured response to step change in steam valve voltage');
```



The values t1 and t2 are the times where the response attains 28.3% and 63.2% of its final value. You can use these values to estimate the time constant `tau` and dead time `theta` for the heat exchanger:

```
t1 = 21.8; t2 = 36.0;
tau = 3/2 * ( t2 - t1 )
theta = t2 - tau
```

```
tau =

   21.3000


theta =

   14.7000
```

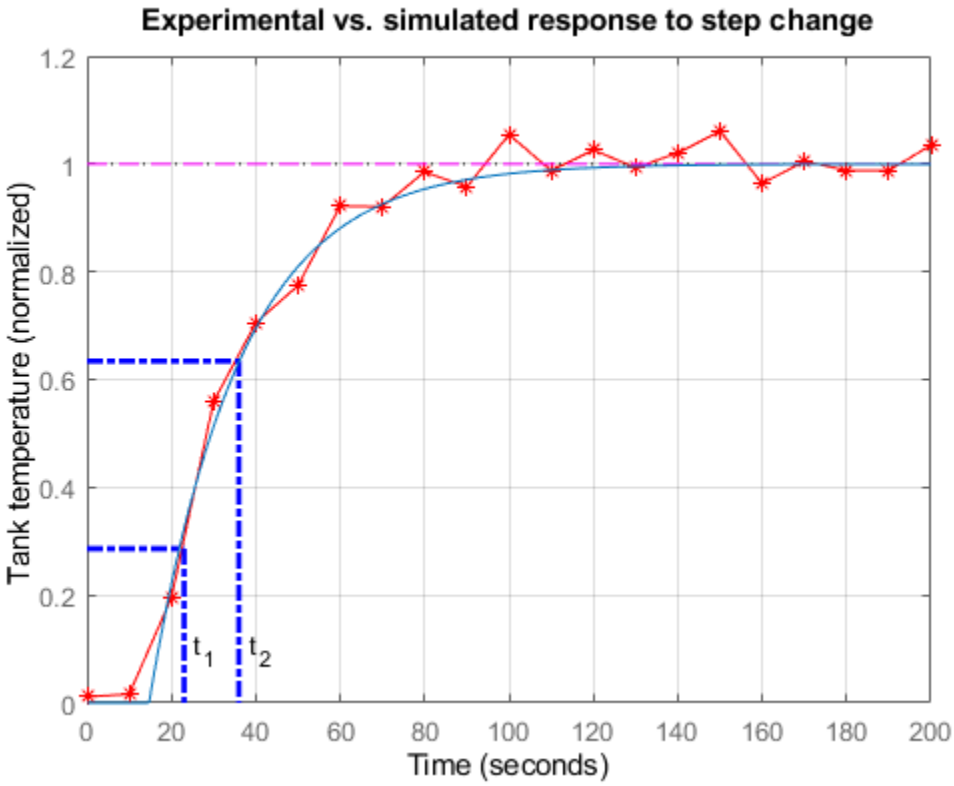
Verify these calculations by comparing the first-order-plus-deadtime response with the measured response:

```
s = tf('s');
Gp = exp(-theta*s)/(1+tau*s)
```

```
Gp =

                    1
```

```
  exp(-14.7*s) * ----------
                 21.3 s + 1

Continuous-time transfer function.
```

```
hold on, step(Gp), hold off
title('Experimental vs. simulated response to step change');
```



The model response and the experimental data are in good agreement. A similar bump test experiment could be conducted to estimate the first-order response to a step disturbance in inflow temperature. Equipped with models for the heat exchanger and inflow disturbance, we are ready to design the control algorithm.

Feedback Control

A block diagram representation of the open-loop process is shown below.

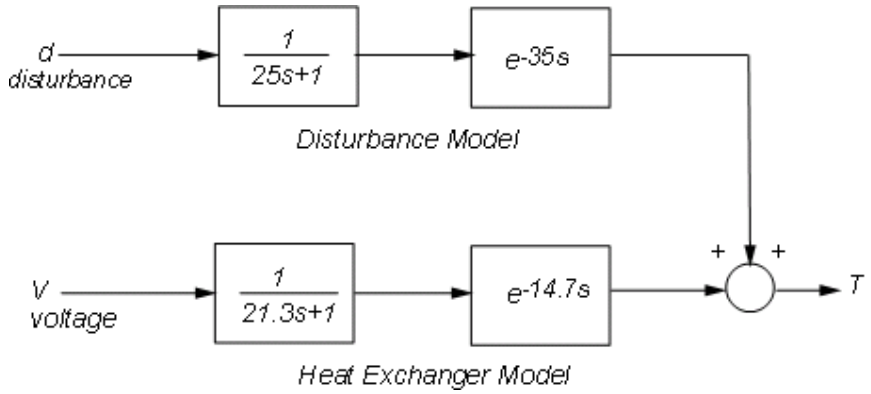


Figure 2: Open-Loop Process.

The transfer function

$$G_p(s) = \frac{e^{-14.7s}}{21.3s + 1}$$

models how a change in the voltage `V` driving the steam valve opening affects the tank temperature `T`, while the transfer function

$$G_d(s) = \frac{e^{-35s}}{25s + 1}$$

models how a change `d` in inflow temperature affects `T`. To regulate the tank temperature `T` around a given setpoint `Tsp`, we can use the following feedback architecture to control the valve opening (voltage `V`):

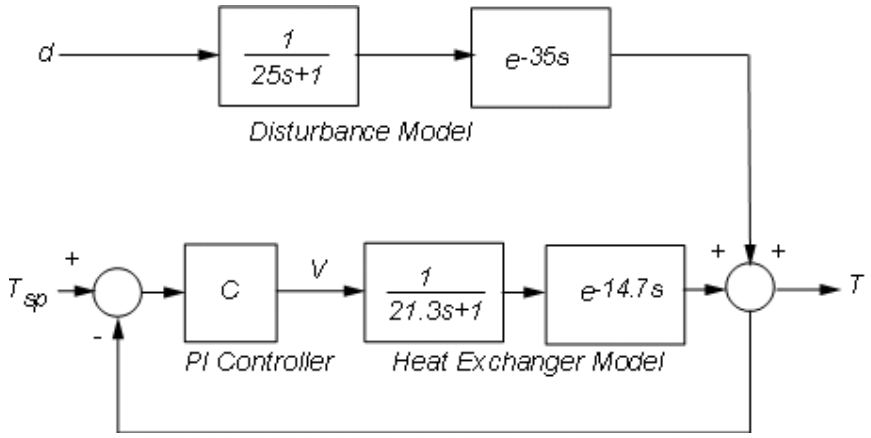


Figure 3: Feedback Control.

In this configuration, the proportional-integral (PI) controller

$$C(s) = K_c(1 + \frac{1}{\tau_c s})$$

calculates the voltage `V` based on the gap `Tsp-T` between the desired and measured temperatures. You can use the ITAE formulas to pick adequate values for the controller parameters:

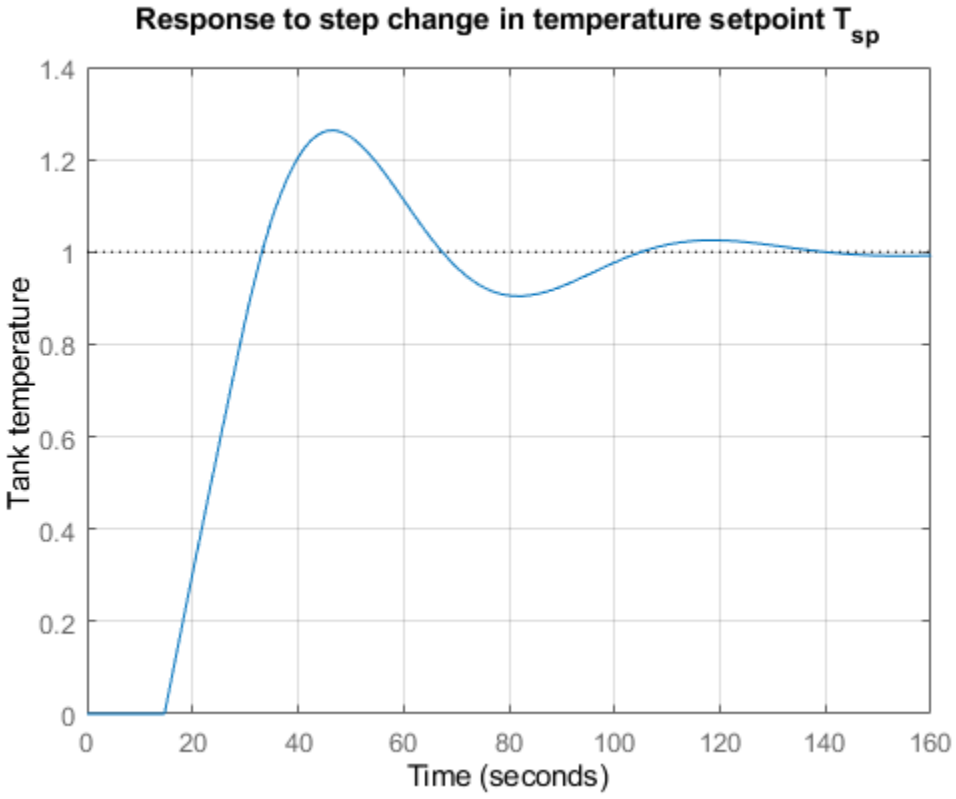
$$K_c = 0.859(\theta/\tau)^{-0.977}, \quad \tau_c = (\theta/\tau)^{0.680}\tau/0.674$$

```
Kc = 0.859 * (theta / tau)^(-0.977)
tauc = ( tau / 0.674 ) * ( theta / tau )^0.680
C = Kc * (1 + 1/(tauc*s));
```

```
Kc =

    1.2341


tauc =

   24.5582
```

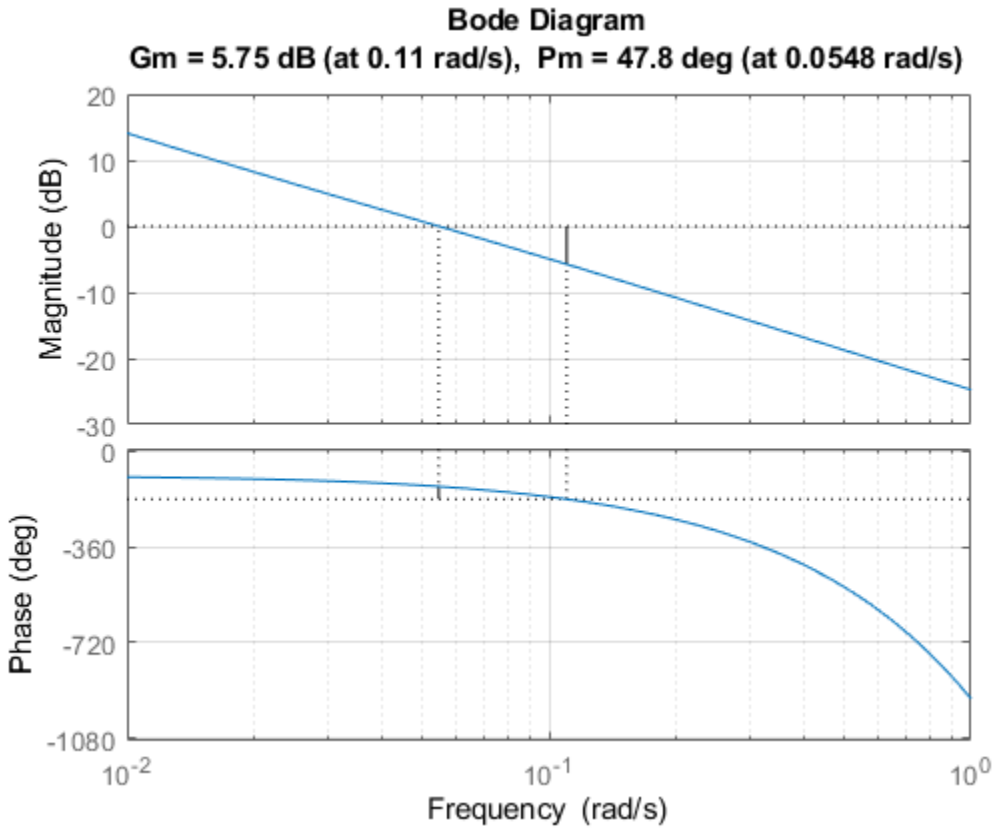
To see how well the ITAE controller performs, close the feedback loop and simulate the response to a set point change:

```
Tfb = feedback(ss(Gp*C),1);
step(Tfb), grid on
title('Response to step change in temperature setpoint T_{sp}')
ylabel('Tank temperature')
```



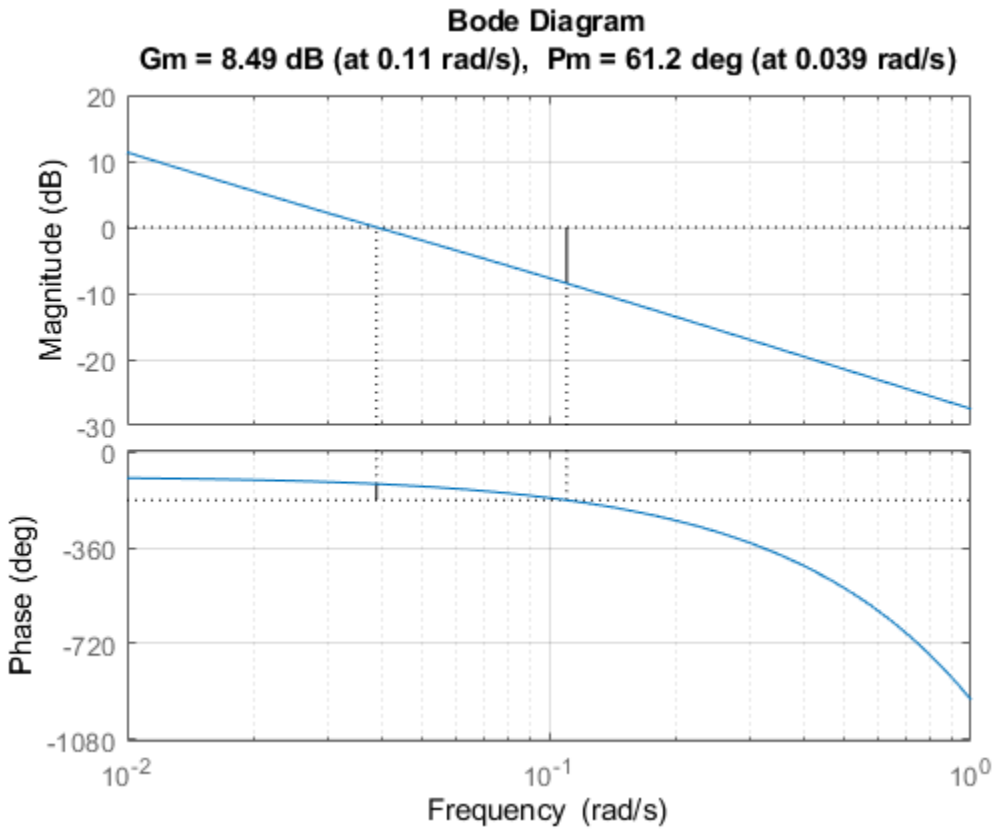
The response is fairly fast with some overshoot. Looking at the stability margins confirms that the gain margin is weak:

```
margin(Gp*C), grid
```

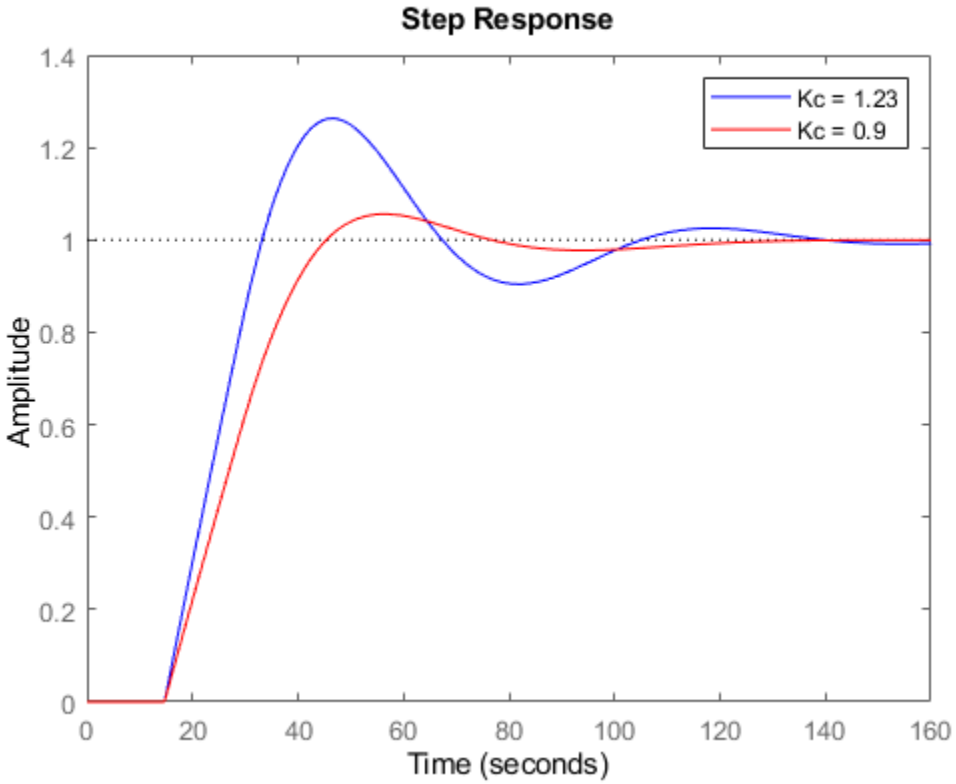


Reducing the proportional gain `Kc` strengthens stability at the expense of performance:

```
C1 = 0.9 * (1 + 1/(tauc*s));  % reduce Kc from 1.23 to 0.9
margin(Gp*C1), grid
```




```
step(Tfb,'b', feedback(ss(Gp*C1),1),'r')
legend('Kc = 1.23','Kc = 0.9')
```



Feedforward Control

Recall that changes in inflow temperature are the main source of temperature fluctuations in the tank. To reject such disturbances, an alternative to feedback control is the feedforward architecture shown below:

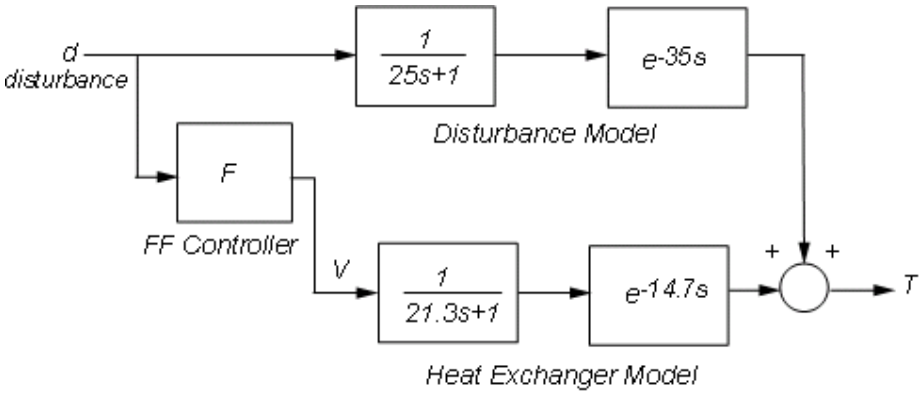


Figure 4: Feedforward Control.

In this configuration, the feedforward controller F uses measurements of the inflow temperature to adjust the steam valve opening (voltage V). Feedforward control thus anticipates and preempts the effect of inflow temperature changes.

Straightforward calculation shows that the overall transfer from temperature disturbance d to tank temperature T is

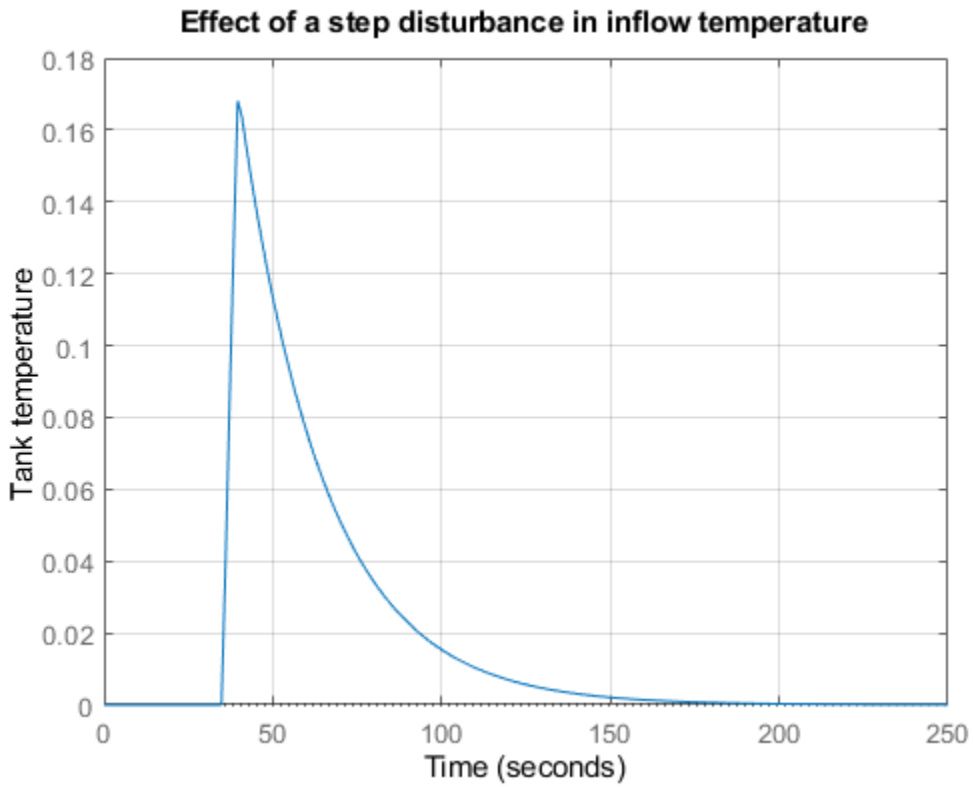
$$T = (G_p F + G_d)d$$

Perfect disturbance rejection requires

$$G_p F + G_d = 0 \rightarrow F = -\frac{G_d}{G_p} = -\frac{21.3s+1}{25s+1}e^{-20.3s}$$

In reality, modeling inaccuracies prevent exact disturbance rejection, but feedforward control will help minimize temperature fluctuations due to inflow disturbances. To get a better sense of how the feedforward scheme would perform, increase the ideal feedforward delay by 5 seconds and simulate the response to a step change in inflow temperature:

```
Gd = exp(-35*s)/(25*s+1);
F = -(21.3*s+1)/(25*s+1) * exp(-25*s);
Tff = Gp * ss(F) + Gd;   % d->T transfer with feedforward control

step(Tff), grid
title('Effect of a step disturbance in inflow temperature')
ylabel('Tank temperature')
```



Combined Feedforward-Feedback Control

Feedback control is good for setpoint tracking in general, while feedforward control can help with rejection of measured disturbances. Next we look at the benefits of combining both schemes. The corresponding control architecture is shown below:

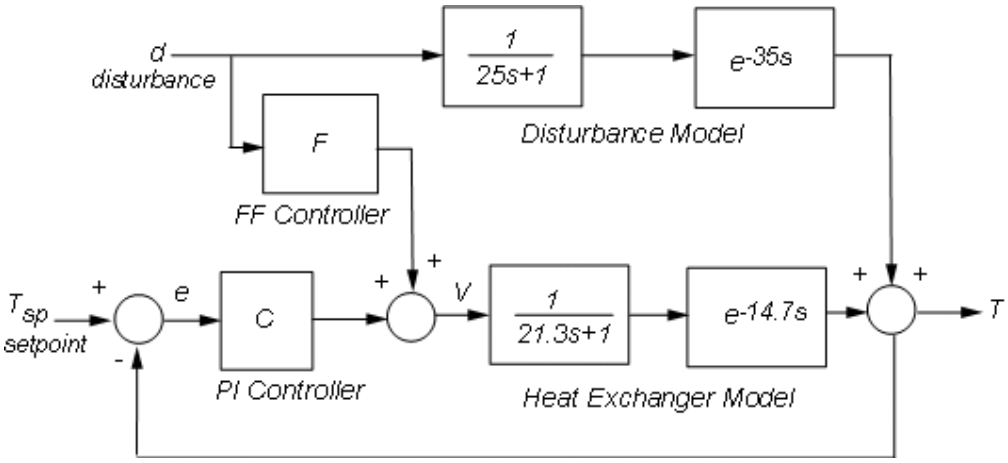


Figure 5: Feedforward-Feedback Control.

Use `connect` to build the corresponding closed-loop model from `Tsp,d` to `T`. First name the input and output channels of each block, then let `connect` automatically wire the diagram:

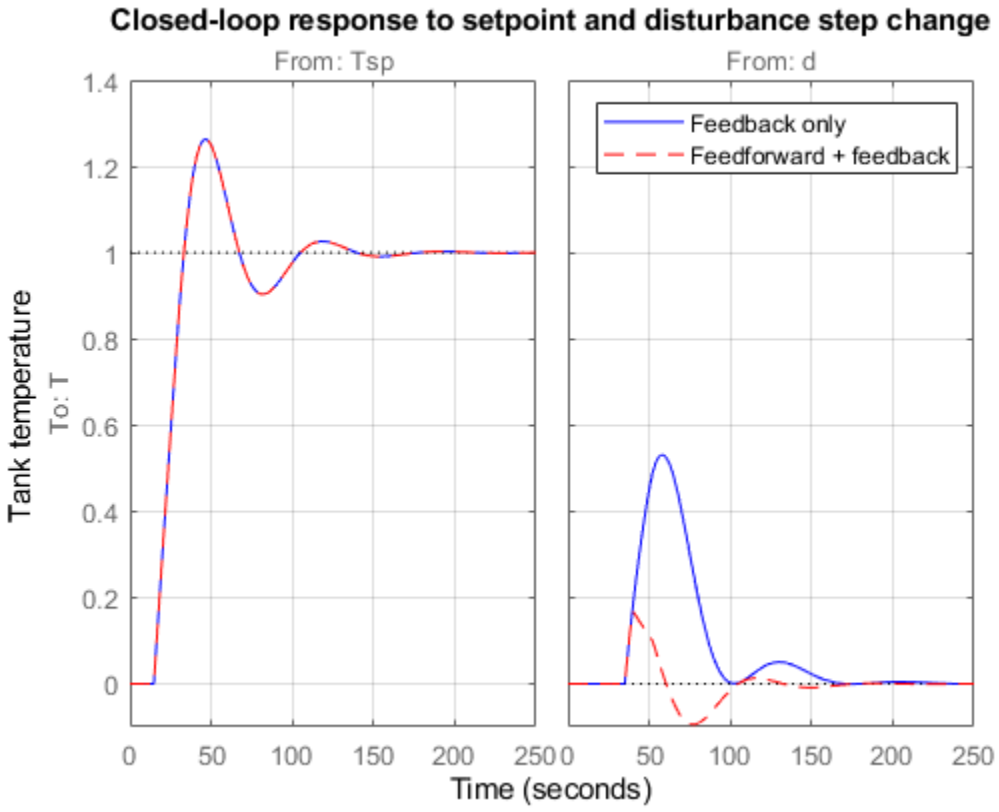
```
Gd.u = 'd';  Gd.y = 'Td';
Gp.u = 'V';  Gp.y = 'Tp';
F.u = 'd';  F.y = 'Vf';
C.u = 'e';  C.y = 'Vc';
Sum1 = sumblk('e = Tsp - T');
Sum2 = sumblk('V = Vf + Vc');
Sum3 = sumblk('T = Tp + Td');
Tffb = connect(Gp,Gd,C,F,Sum1,Sum2,Sum3,{'Tsp','d'},'T');
```

To compare the closed-loop responses with and without feedforward control, calculate the corresponding closed-loop transfer function for the feedback-only configuration:

```
C.u = 'e';  C.y = 'V';
Tfb = connect(Gp,Gd,C,Sum1,Sum3,{'Tsp','d'},'T');
```

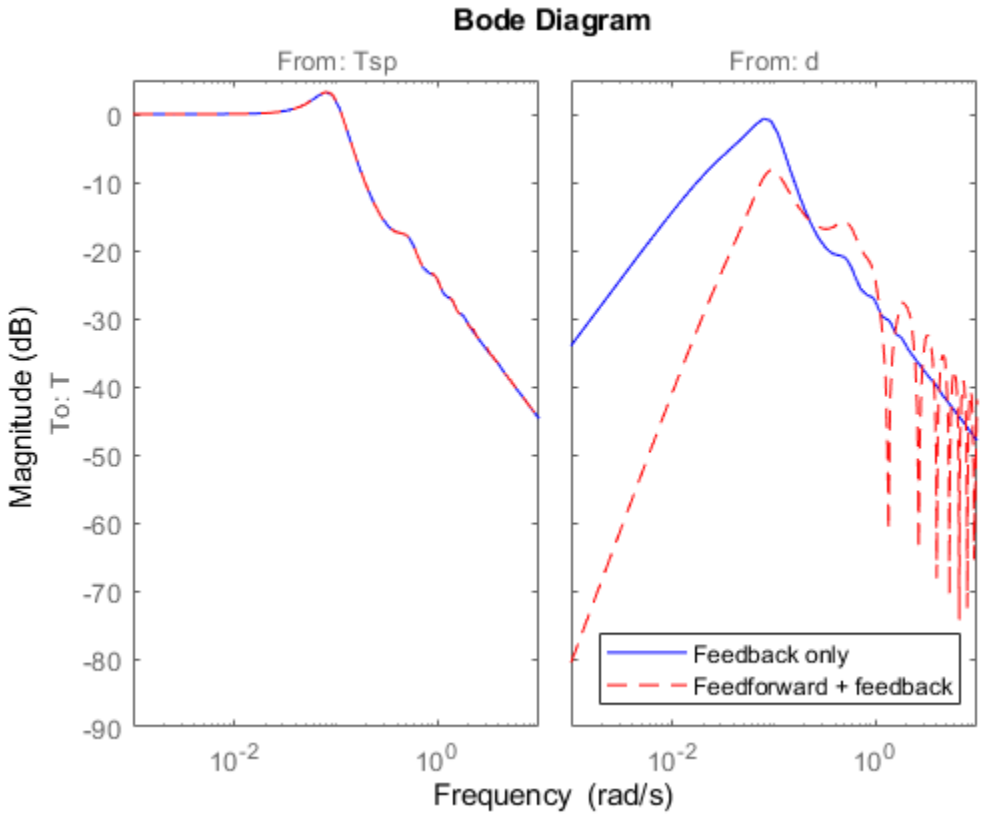
Now compare the two designs:

```
step(Tfb,'b',Tffb,'r--'), grid
title('Closed-loop response to setpoint and disturbance step change')
ylabel('Tank temperature')
legend('Feedback only','Feedforward + feedback')
```



The two designs have identical performance for setpoint tracking, but the addition of feedforward control is clearly beneficial for disturbance rejection. This is also visible on the closed-loop Bode plot

```
bodemag(Tfb,'b',Tffb,'r--',{1e-3,1e1})
legend('Feedback only','Feedforward + feedback','Location','SouthEast')
```

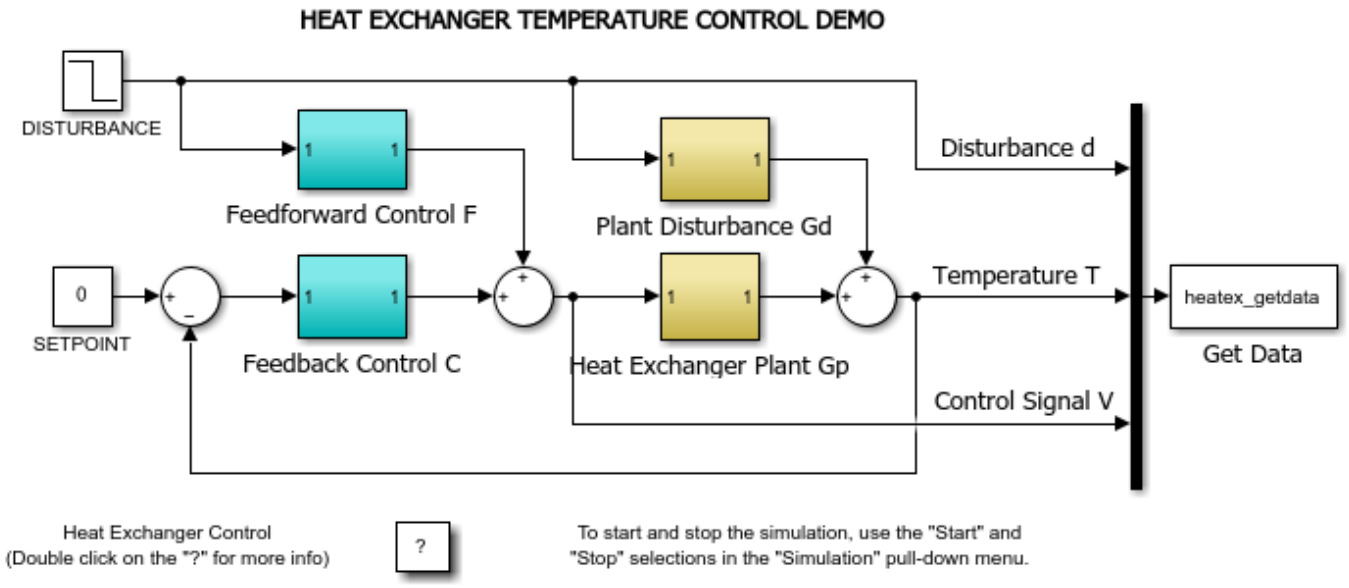


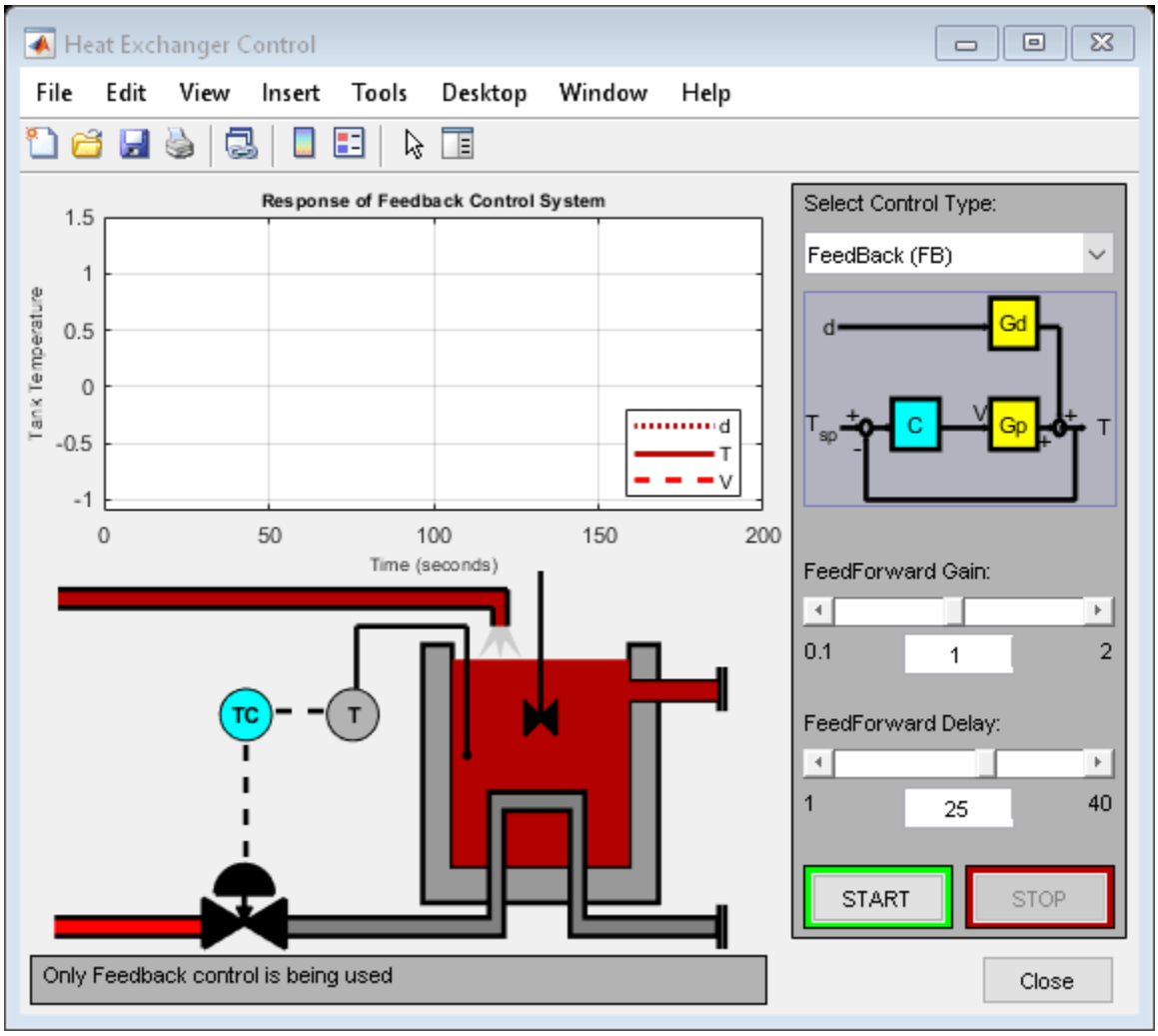
Interactive Simulation

To gain additional insight and interactively tune the feedforward and feedback gains, use the companion GUI and Simulink® model. Click on the link below to launch the GUI.

Open the Heat Exchanger model and GUI

```
heatex
```


HEAT EXCHANGER TEMPERATURE CONTROL DEMO
DISTURBANCE
Feedforward Control F
Plant Disturbance Gd
Disturbance d
0
SETPOINT
Feedback Control C
Heat Exchanger Plant Gp
Temperature T
heatex_getdata
Get Data
Control Signal V
Heat Exchanger Control
(Double click on the "?" for more info)
?
To start and stop the simulation, use the "Start" and
"Stop" selections in the "Simulation" pull-down menu.


Heat Exchanger Control
File Edit View Insert Tools Desktop Window Help
Response of Feedback Control System
Tank Temperature
Time (seconds)
d
T
V
Select Control Type:
FeedBack (FB)
Gd
C
Gp
FeedForward Gain:
0.1
1
2
FeedForward Delay:
1
25
40
START
STOP
TC
T
Only Feedback control is being used
Close

Control of Processes with Long Dead Time: The Smith Predictor

This example shows the limitations of PI control for processes with long dead time and illustrates the benefits of a control strategy called "Smith Predictor."

The example is inspired by:

```
A. Ingimundarson and T. Hagglund, "Robust Tuning Procedures of
Dead-Time Compensating Controllers," Control Engineering Practice,
9, 2001, pp. 1195-1208.
```

Process Model

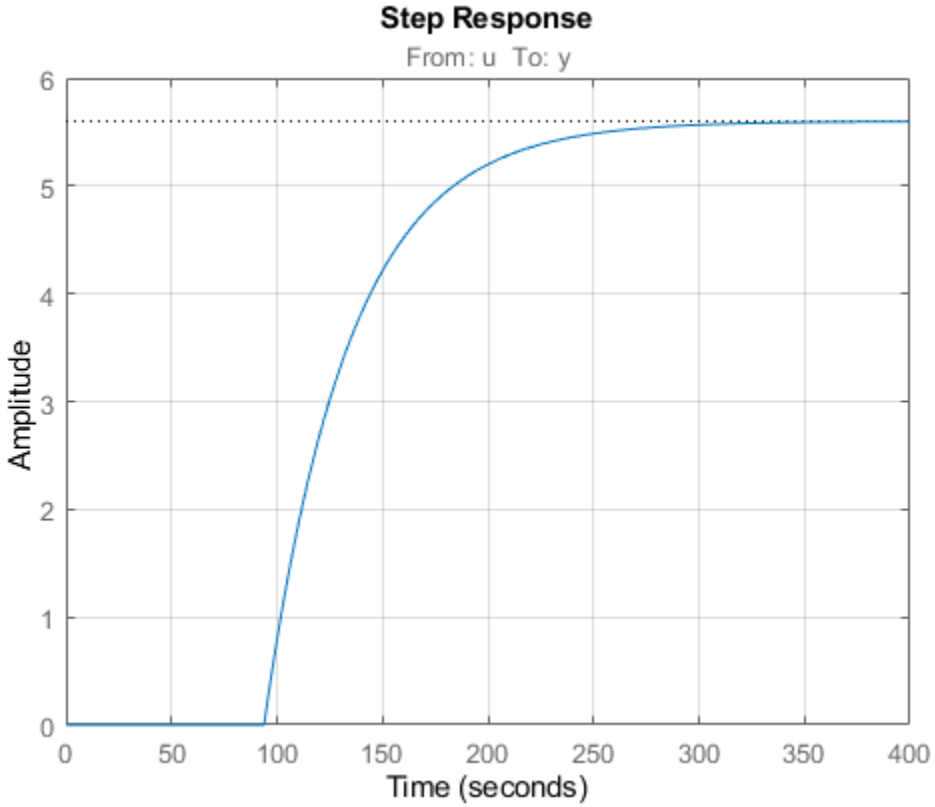
The process open-loop response is modeled as a first-order plus dead time with a 40.2 second time constant and 93.9 second time delay:

```
s = tf('s');
P = exp(-93.9*s) * 5.6/(40.2*s+1);
P.InputName = 'u';
P.OutputName = 'y';
P
```

```
P =

  From input "u" to output "y":
                    5.6
  exp(-93.9*s) * ----------
                 40.2 s + 1

Continuous-time transfer function.
```

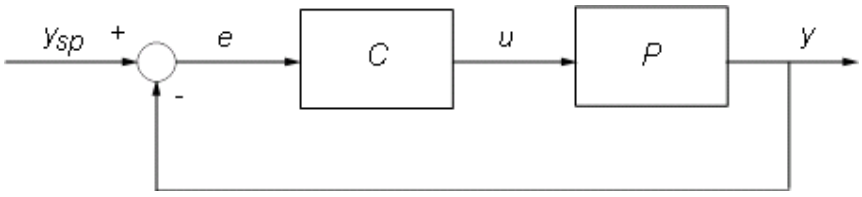
Note that the delay is more than twice the time constant. This model is representative of many chemical processes. Its step response is shown below.

```
step(P), grid on
```



PI Controller

Proportional-Integral (PI) control is a commonly used technique in Process Control. The corresponding control architecture is shown below.



Compensator C is a PI controller in standard form with two tuning parameters: proportional gain `Kp` and an integral time `Ti`. We use the `PIDTUNE` command to design a PI controller with the open loop bandwidth at 0.006 rad/s:

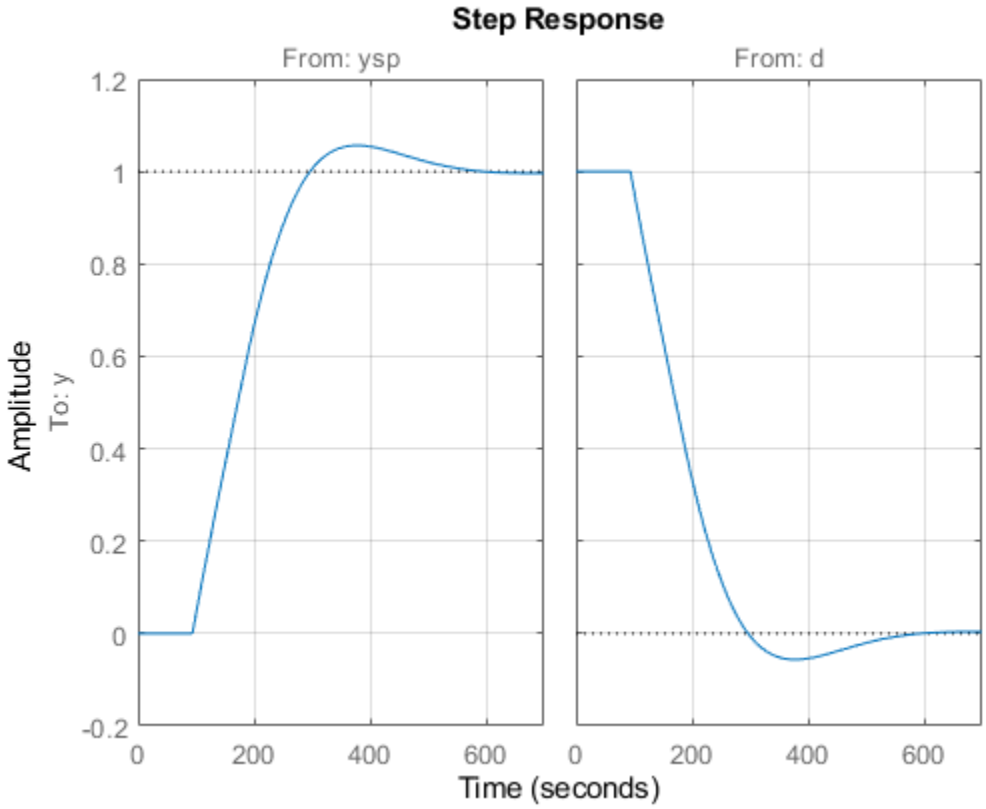
```
Cpi = pidtune(P,pidstd(1,1),0.006);
Cpi
```

```
Cpi =

             1      1
  Kp * (1 + ---- * ---)
             Ti     s

  with Kp = 0.0501, Ti = 47.3
```

```
Continuous-time PI controller in standard form
```

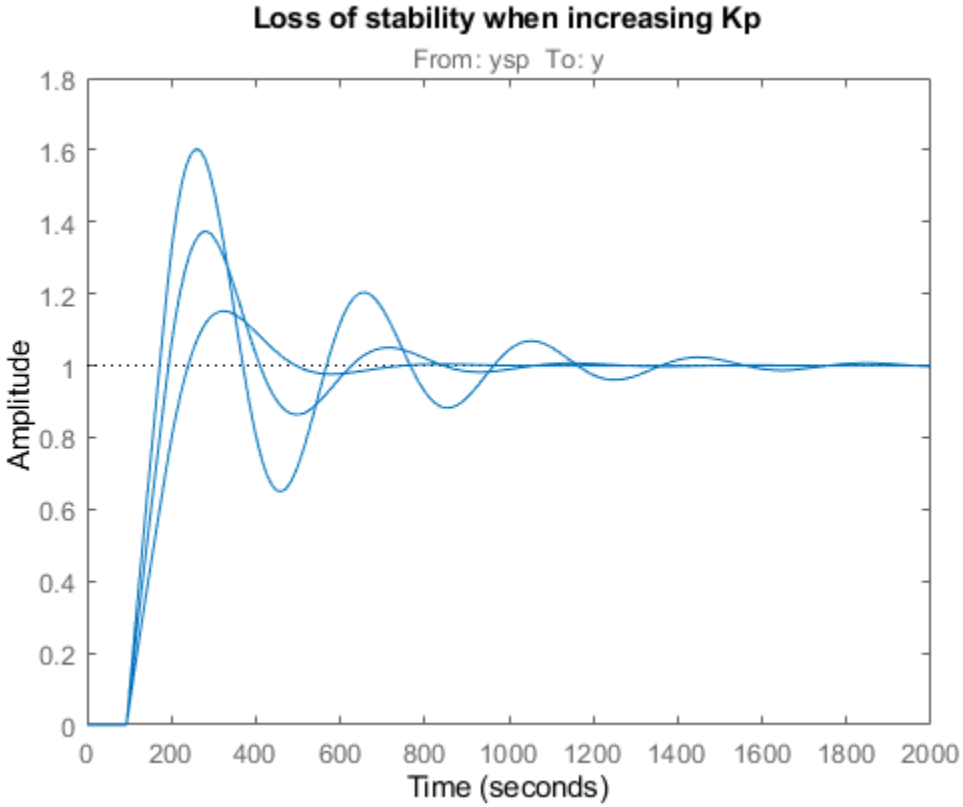
To evaluate the performance of the PI controller, close the feedback loop and simulate the responses to step changes in the reference signal `ysp` and output disturbance signal `d`. Because of the delay in the feedback path, it is necessary to convert `P` or `Cpi` to the state-space representation using the `SS` command:

```
Tpi = feedback([P*Cpi,1],1,1,1);  % closed-loop model [ysp;d]->y
Tpi.InputName = {'ysp' 'd'};

step(Tpi), grid on
```



The closed-loop response has acceptable overshoot but is somewhat sluggish (it settles in about 600 seconds). Increasing the proportional gain Kp speeds up the response but also significantly increases overshoot and quickly leads to instability:

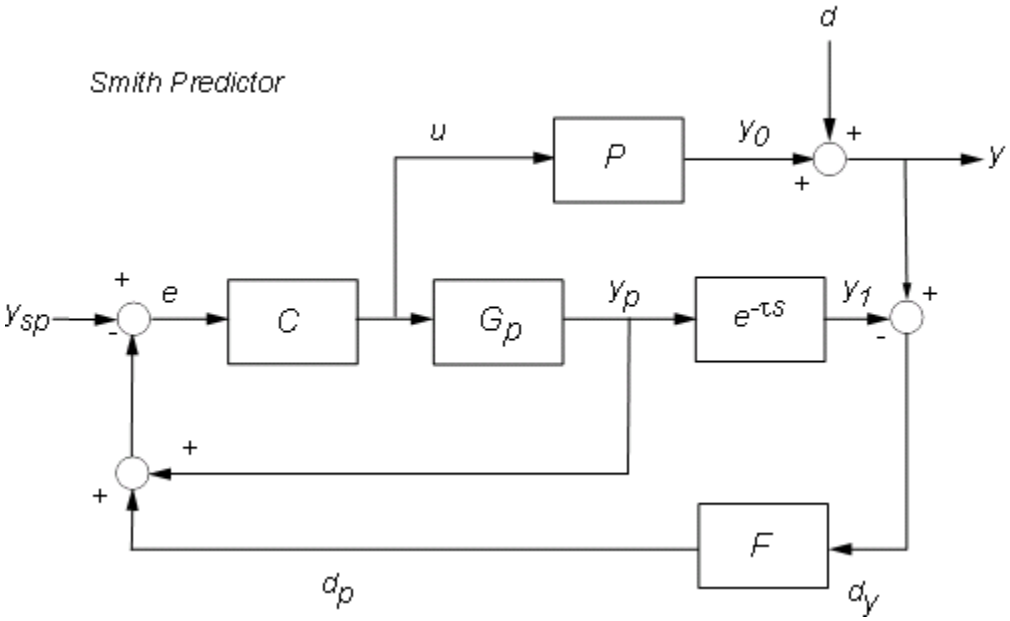
```
Kp3 = [0.06;0.08;0.1];      % try three increasing values of Kp
Ti3 = repmat(Cpi.Ti,3,1);   % Ti remains the same
C3 = pidstd(Kp3,Ti3);       % corresponding three PI controllers
T3 = feedback(P*C3,1);
T3.InputName = 'ysp';

step(T3)
title('Loss of stability when increasing Kp')
```



The performance of the PI controller is severely limited by the long dead time. This is because the PI controller has no knowledge of the dead time and reacts too "impatiently" when the actual output `y` does not match the desired setpoint `ysp`. Everyone has experienced a similar phenomenon in showers where the water temperature takes a long time to adjust. There, impatience typically leads to alternate scolding by burning hot and freezing cold water. A better strategy consists of waiting for a change in temperature setting to take effect before making further adjustments. And once we have learned what knob setting delivers our favorite temperature, we can get the right temperature in just the time it takes the shower to react. This "optimal" control strategy is the basic idea behind the Smith Predictor scheme.

Smith Predictor

The Smith Predictor control structure is sketched below.



The Smith Predictor uses an internal model `Gp` to predict the delay-free response yp of the process (e.g., what water temperature a given knob setting will deliver). It then compares this prediction yp with the desired setpoint ysp to decide what adjustments are needed (control u). To prevent drifting and reject external disturbances, the Smith predictor also compares the actual process output with a prediction `y1` that takes the dead time into account. The gap `dy=y-y1` is fed back through a filter F and contributes to the overall error signal `e`. Note that `dy` amounts to the perceived temperature mismatch *after* waiting long enough for the shower to react.

Deploying the Smith Predictor scheme requires

- A model `Gp` of the process dynamics and an estimate `tau` of the process dead time
- Adequate settings for the compensator and filter dynamics (`C` and `F`)

Based on the process model, we use:

$$G_p(s) = \frac{5.6}{1+40.2s}, \tau = 93.9$$

For `F`, use a first-order filter with a 20 second time constant to capture low-frequency disturbances.

```
F = 1/(20*s+1);
F.InputName = 'dy';
F.OutputName = 'dp';
```

For `C`, we re-design the PI controller with the overall plant seen by the PI controller, which includes dynamics from `P`, `Gp`, `F` and dead time. With the help of the Smith Predictor control structure we are able to increase the open-loop bandwidth to achieve faster response and increase the phase margin to reduce the overshoot.

```
% Process
P = exp(-93.9*s) * 5.6/(40.2*s+1);
P.InputName = 'u';
P.OutputName = 'y0';
```

```
% Prediction model
Gp = 5.6/(40.2*s+1);
Gp.InputName = 'u';
Gp.OutputName = 'yp';

Dp = exp(-93.9*s);
Dp.InputName = 'yp'; Dp.OutputName = 'y1';

% Overall plant
S1 = sumblk('ym = yp + dp');
S2 = sumblk('dy = y0 - y1');
Plant = connect(P,Gp,Dp,F,S1,S2,'u','ym');

% Design PI controller with 0.08 rad/s bandwidth and 90 degrees phase margin
Options = pidtuneOptions('PhaseMargin',90);
C = pidtune(Plant,pidstd(1,1),0.08,Options);
C.InputName = 'e';
C.OutputName = 'u';
C
```

```
C =

             1      1
  Kp * (1 + ---- * ---)
             Ti     s

  with Kp = 0.574, Ti = 40.2

Continuous-time PI controller in standard form
```

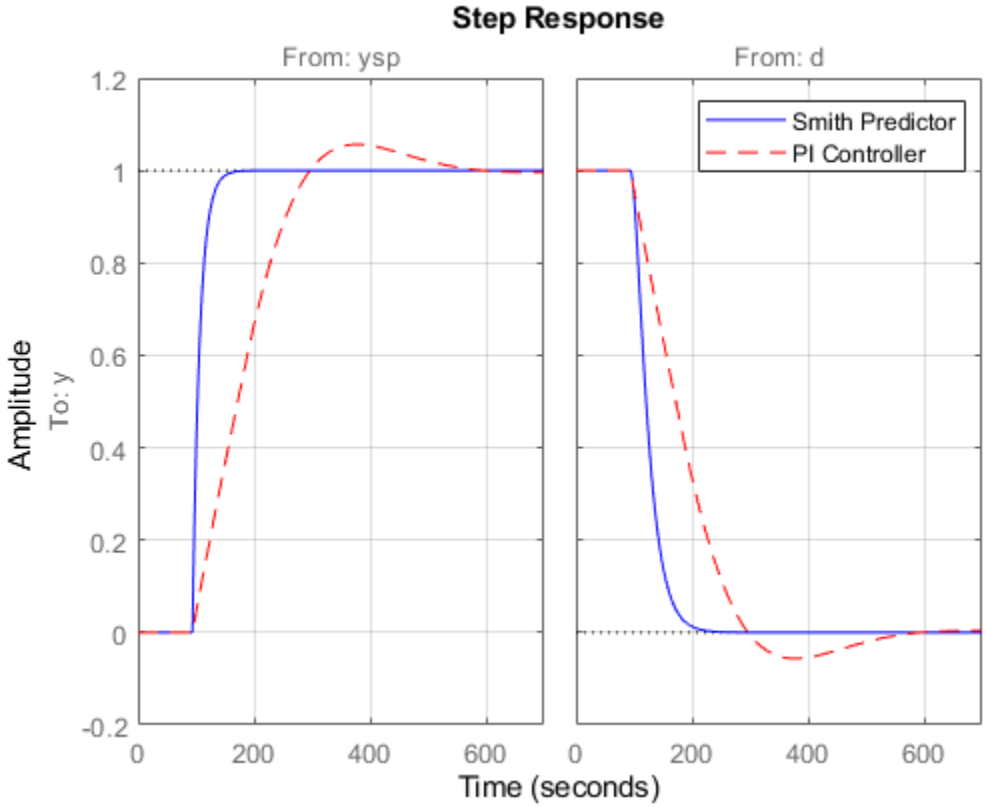
Comparison of PI Controller vs. Smith Predictor

To compare the performance of the two designs, first derive the closed-loop transfer function from `ysp,d` to `y` for the Smith Predictor architecture. To facilitate the task of connecting all the blocks involved, name all their input and output channels and let `CONNECT` do the wiring:

```
% Assemble closed-loop model from [y_sp,d] to y
Sum1 = sumblk('e = ysp - yp - dp');
Sum2 = sumblk('y = y0 + d');
Sum3 = sumblk('dy = y - y1');
T = connect(P,Gp,Dp,C,F,Sum1,Sum2,Sum3,{'ysp','d'},'y');
```

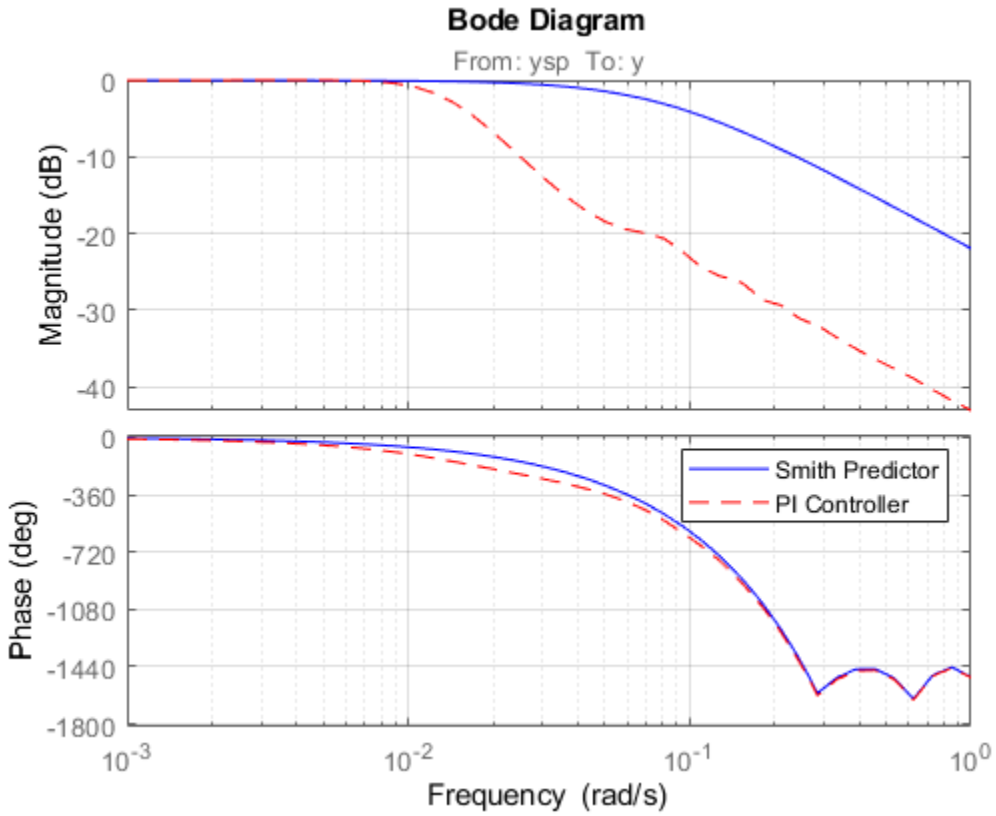
Use `STEP` to compare the Smith Predictor (blue) with the PI controller (red):

```
step(T,'b',Tpi,'r--')
grid on
legend('Smith Predictor','PI Controller')
```



The Smith Predictor provides much faster response with no overshoot. The difference is also visible in the frequency domain by plotting the closed-loop Bode response from `ysp` to `y`. Note the higher bandwidth for the Smith Predictor.

```
bode(T(1,1),'b',Tpi(1,1),'r--',{1e-3,1})
grid on
legend('Smith Predictor','PI Controller')
```



Robustness to Model Mismatch

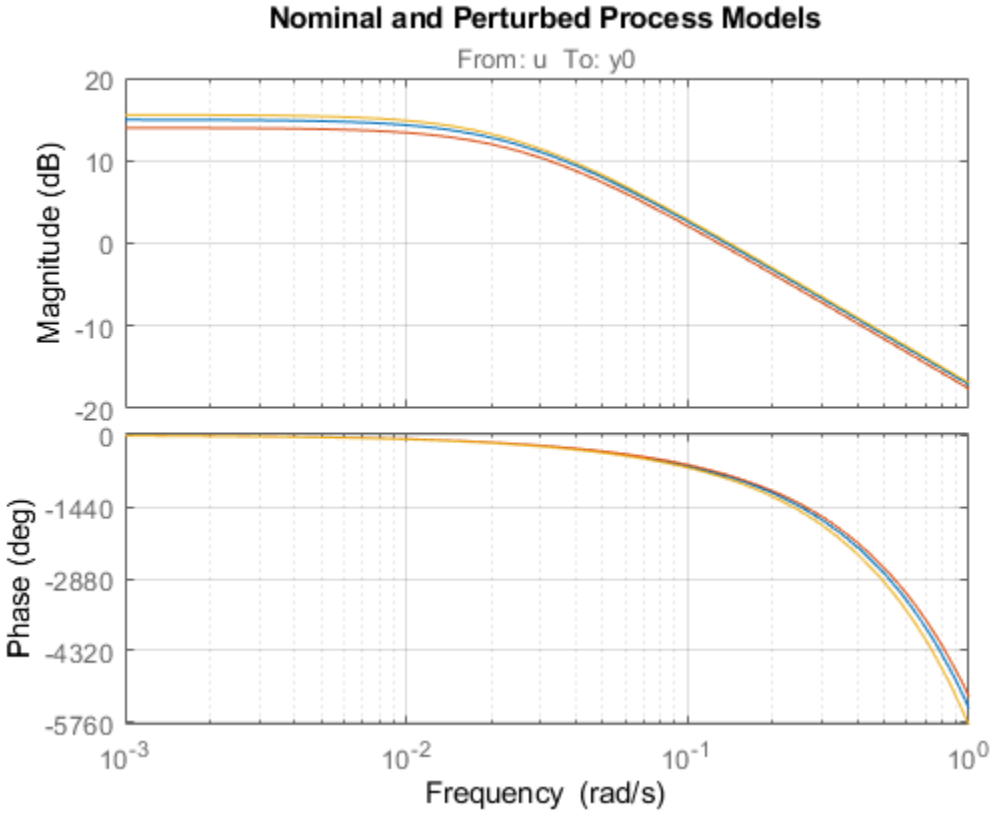
In the previous analysis, the internal model

$$G_p(s)e^{-\tau s}$$

matched the process model P exactly. In practical situations, the internal model is only an approximation of the true process dynamics, so it is important to understand how robust the Smith Predictor is to uncertainty on the process dynamics and dead time.

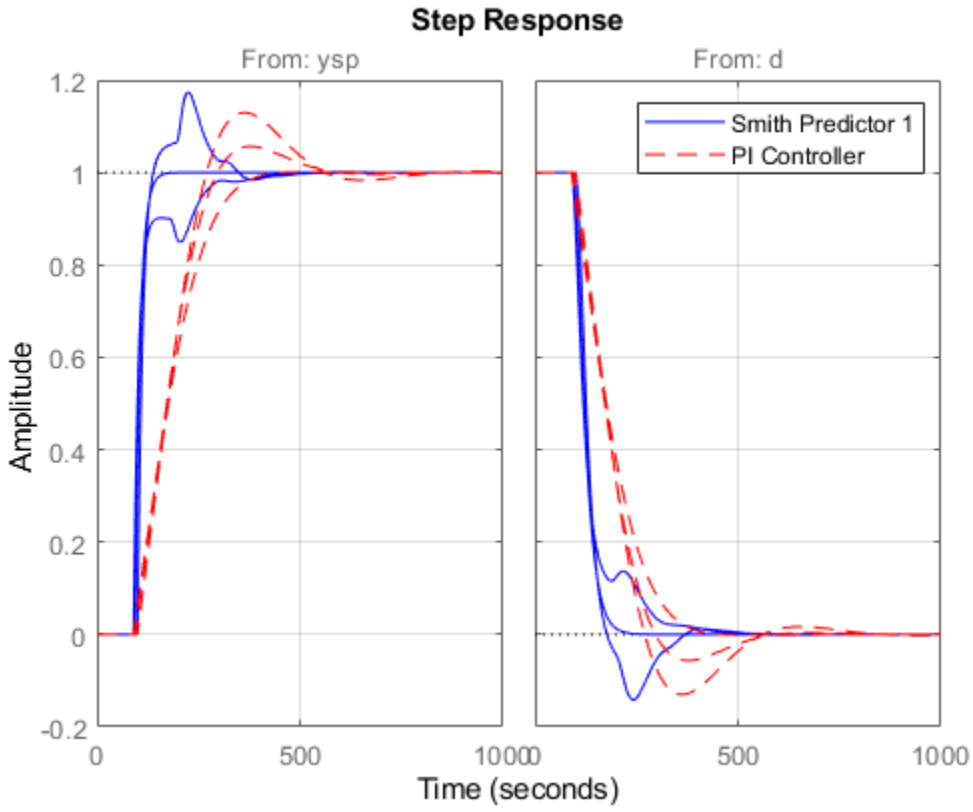
Consider two perturbed plant models representative of the range of uncertainty on the process parameters:

```
P1 = exp(-90*s) * 5/(38*s+1);
P2 = exp(-100*s) * 6/(42*s+1);

bode(P,P1,P2), grid on
title('Nominal and Perturbed Process Models')
```



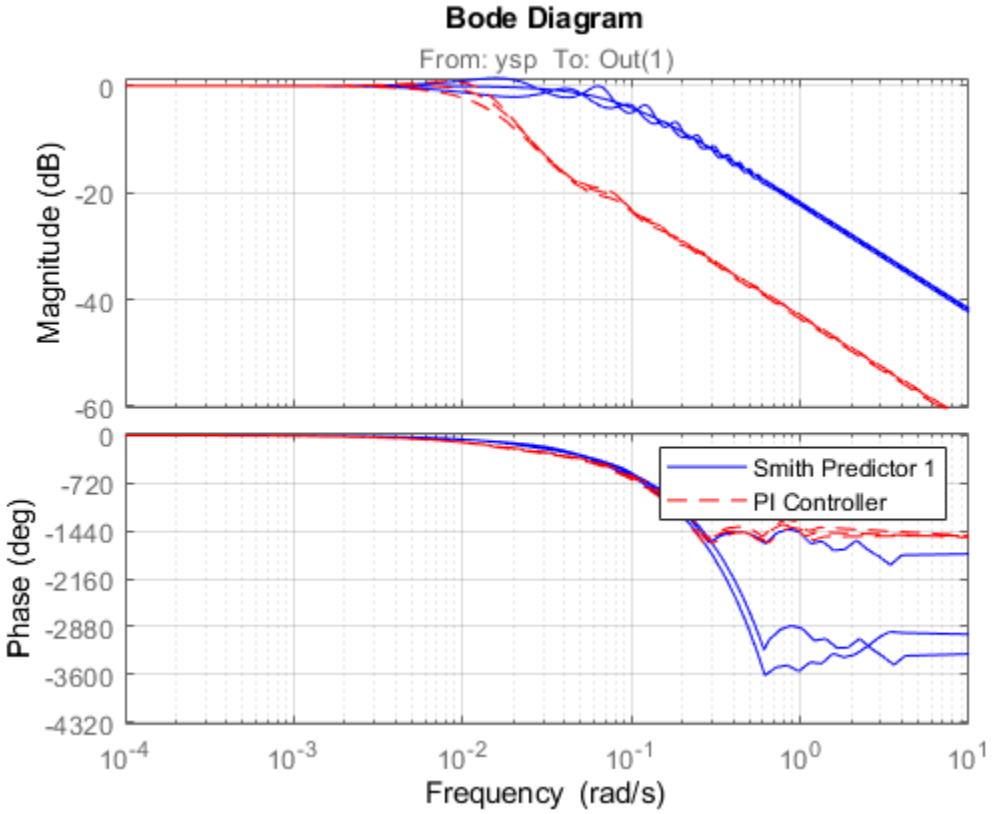
To analyze robustness, collect the nominal and perturbed models into an array of process models, rebuild the closed-loop transfer functions for the PI and Smith Predictor designs, and simulate the closed-loop responses:

```
Plants = stack(1,P,P1,P2);  % array of process models
T1 = connect(Plants,Gp,Dp,C,F,Sum1,Sum2,Sum3,{'ysp','d'},'y'); % Smith
Tpi = feedback([Plants*Cpi,1],1,1,1);   % PI

step(T1,'b',Tpi,'r--')
grid on
legend('Smith Predictor 1','PI Controller')
```



Both designs are sensitive to model mismatch, as confirmed by the closed-loop Bode plots:

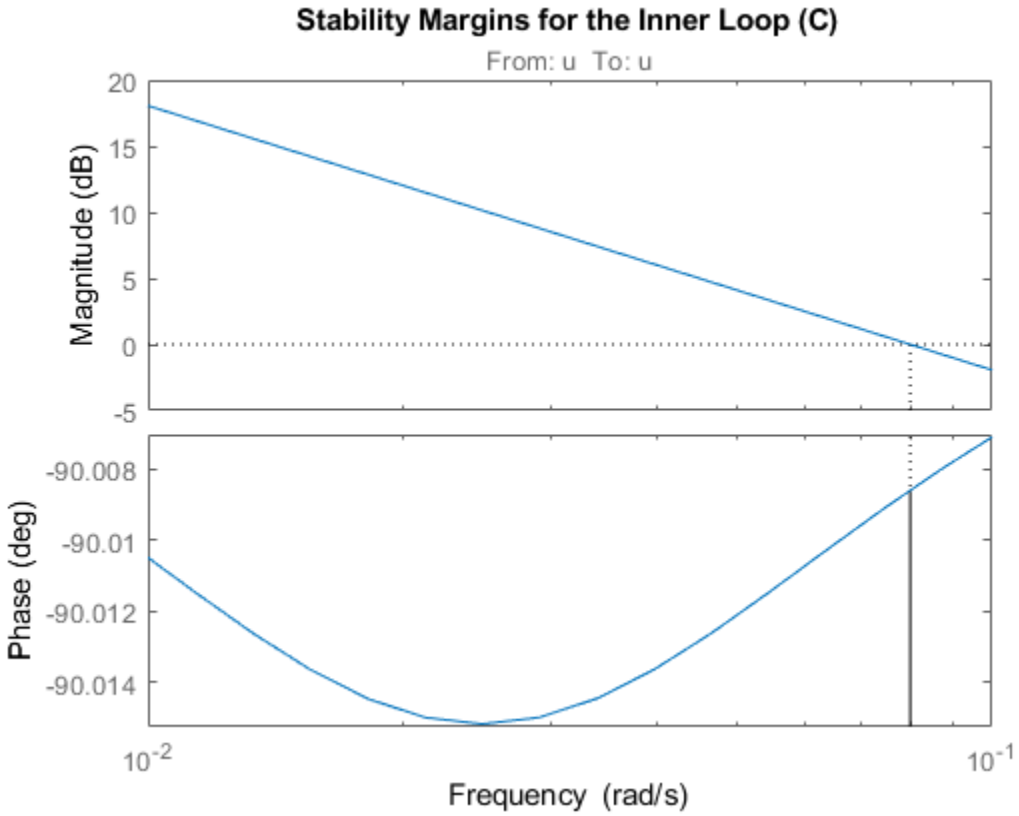
```
bode(T1(1,1),'b',Tpi(1,1),'r--')
grid on
legend('Smith Predictor 1','PI Controller')
```



Improving Robustness

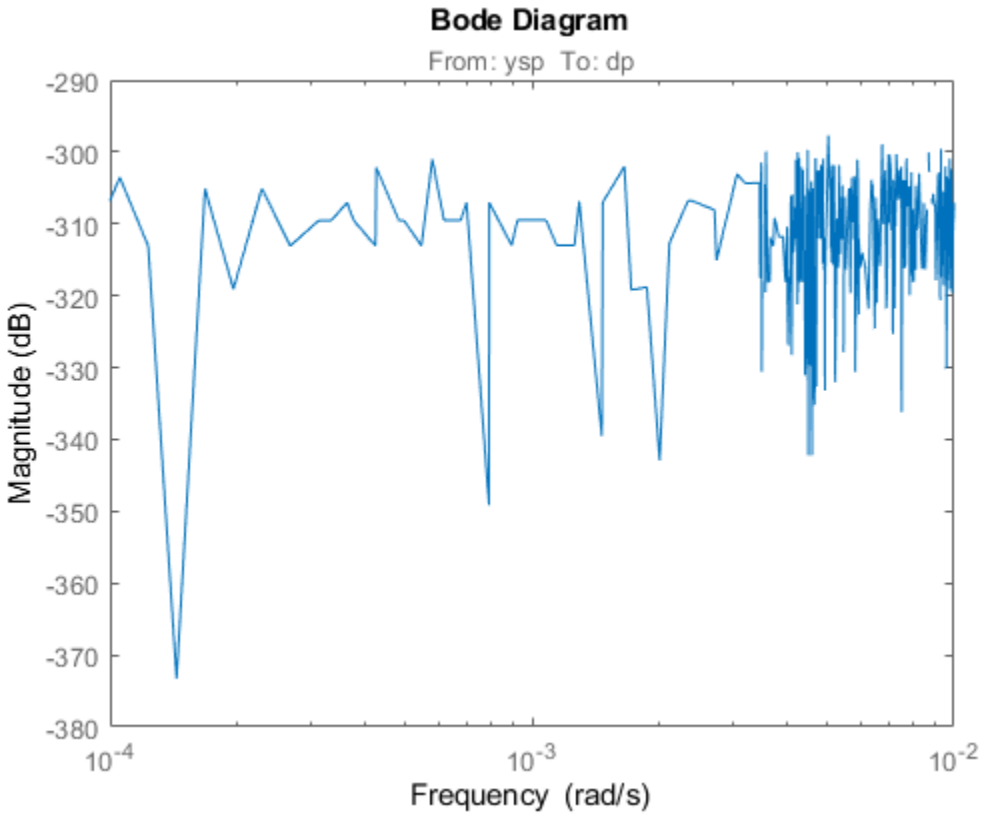
To reduce the Smith Predictor's sensitivity to modeling errors, check the stability margins for the inner and outer loops. The inner loop `C` has open-loop transfer `C*Gp` so the stability margin are obtained by

```
margin(C * Gp)
title('Stability Margins for the Inner Loop (C)')
```



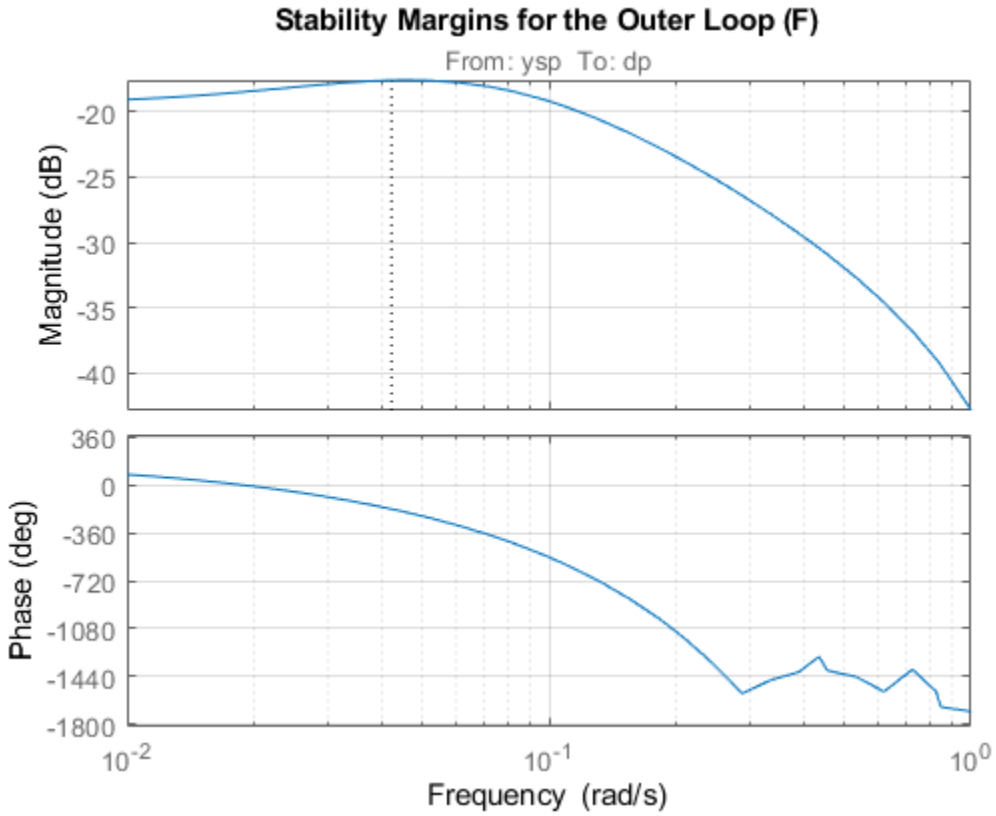
The inner loop has comfortable gain and phase margins so focus on the outer loop next. Use `CONNECT` to derive the open-loop transfer function `L` from `ysp` to `dp` with the inner loop closed:

```
Sum1o = sumblk('e = ysp - yp');  % open the loop at dp
L = connect(P,Gp,Dp,C,F,Sum1o,Sum2,Sum3,{'ysp','d'},'dp');

bodemag(L(1,1))
```



This transfer function is essentially zero, which is to be expected when the process and prediction models match exactly. To get insight into the stability margins for the outer loop, we need to work with one of the perturbed process models, e.g., `P1`:

```
H = connect(Plants(:,:,2),Gp,Dp,C,Sum1o,Sum2,Sum3,{'ysp','d'},'dy');
H = H(1,1);  % open-loop transfer ysp -> dy
L = F * H;

margin(L)
title('Stability Margins for the Outer Loop (F)')
grid on;
xlim([1e-2 1]);
```

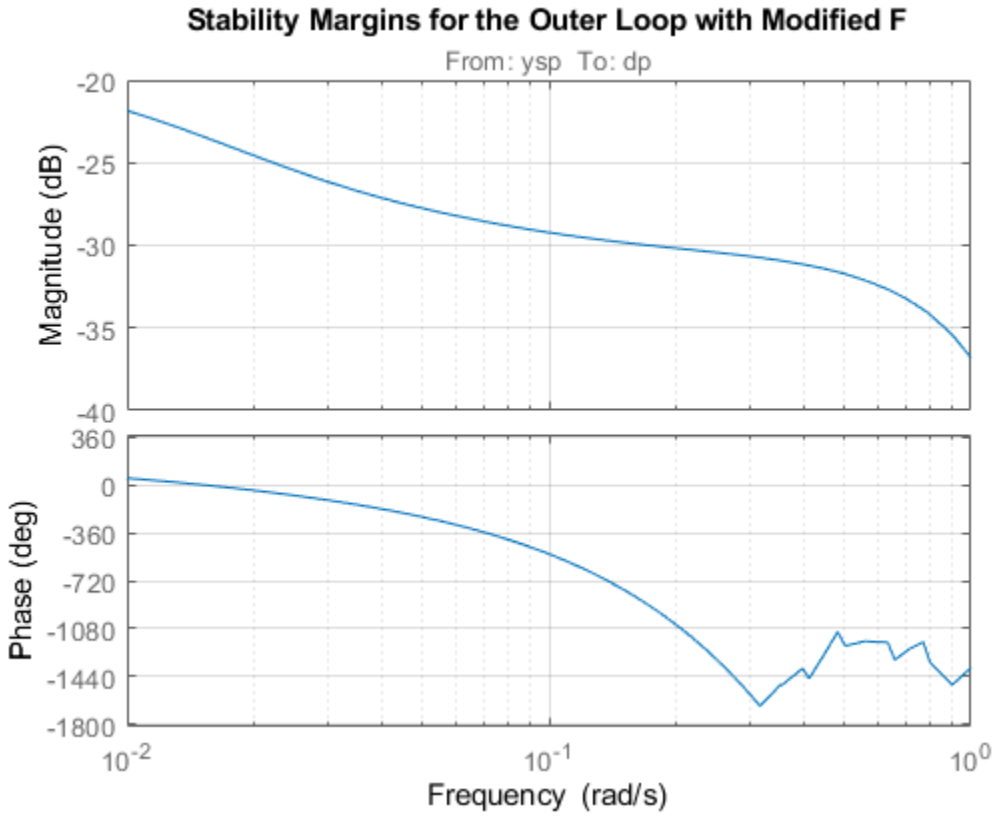


This gain curve has a hump near 0.04 rad/s that lowers the gain margin and increases the hump in the closed-loop step response. To fix this issue, pick a filter `F` that rolls off earlier and more quickly:

```
F = (1+10*s)/(1+100*s);
F.InputName = 'dy';
F.OutputName = 'dp';
```

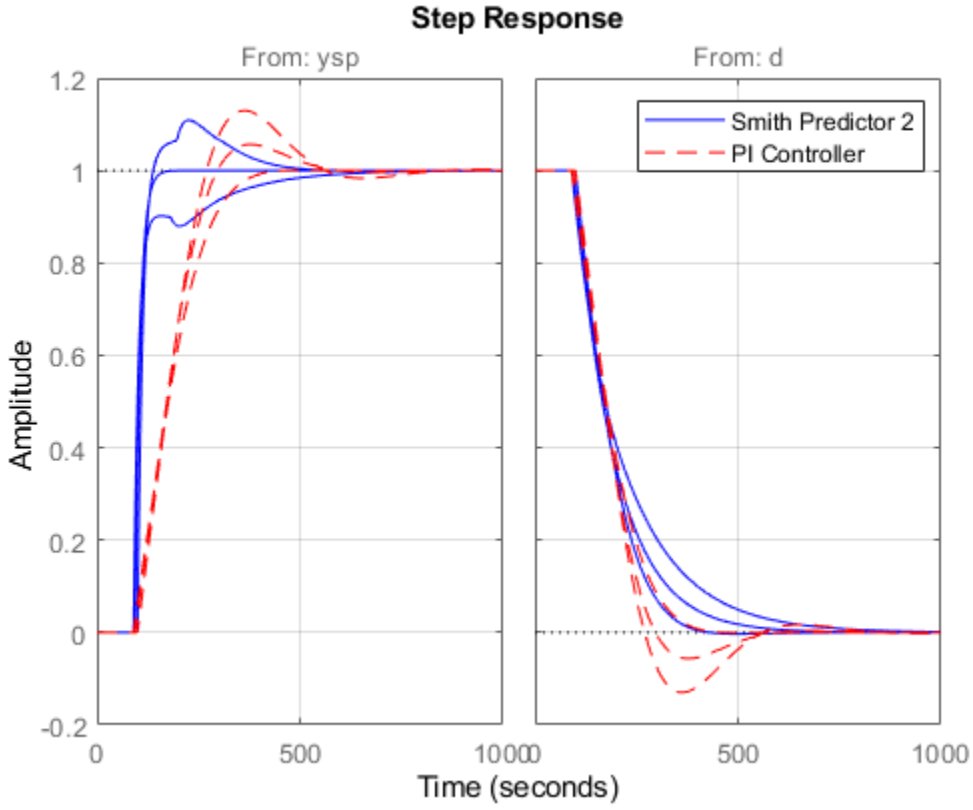
Verify that the gain margin has improved near the 0.04 rad/s phase crossing:

```
L = F * H;
margin(L)
title('Stability Margins for the Outer Loop with Modified F')
grid on;
xlim([1e-2 1]);
```



Finally, simulate the closed-loop responses with the modified filter:

```
T2 = connect(Plants,Gp,Dp,C,F,Sum1,Sum2,Sum3,{'ysp','d'},'y');

step(T2,'b',Tpi,'r--')
grid on
legend('Smith Predictor 2','PI Controller')
```



The modified design provides more consistent performance at the expense of a slightly slower nominal response.

Improving Disturbance Rejection

Formulas for the closed-loop transfer function from `d` to `y` show that the optimal choice for `F` is

$$F(s) = e^{\tau s}$$

where `tau` is the internal model's dead time. This choice achieves perfect disturbance rejection regardless of the mismatch between `P` and `Gp`. Unfortunately, such "negative delay" is not causal and cannot be implemented. In the paper:

```
Huang, H.-P., et al., "A Modified Smith Predictor with an Approximate
Inverse of Dead Time," AiChE Journal, 36 (1990), pp. 1025-1031
```

the authors suggest using the phase lead approximation:

$$e^{\tau s} \approx \frac{1 + B(s)}{1 + B(s)e^{-\tau s}}$$

where `B` is a low-pass filter with the same time constant as the internal model `Gp`. You can test this scheme as follows:

Define B(s) and F(s)

```
B = 0.05/(40*s+1);
tau = totaldelay(Dp);
F = (1+B)/(1+B*exp(-tau*s));
F.InputName = 'dy';
F.OutputName = 'dp';
```

Re-design PI controller with reduced bandwidth

```
Plant = connect(P,Gp,Dp,F,S1,S2,'u','ym');
C = pidtune(Plant,pidstd(1,1),0.02,pidtuneOptions('PhaseMargin',90));
C.InputName = 'e';
C.OutputName = 'u';
C
```

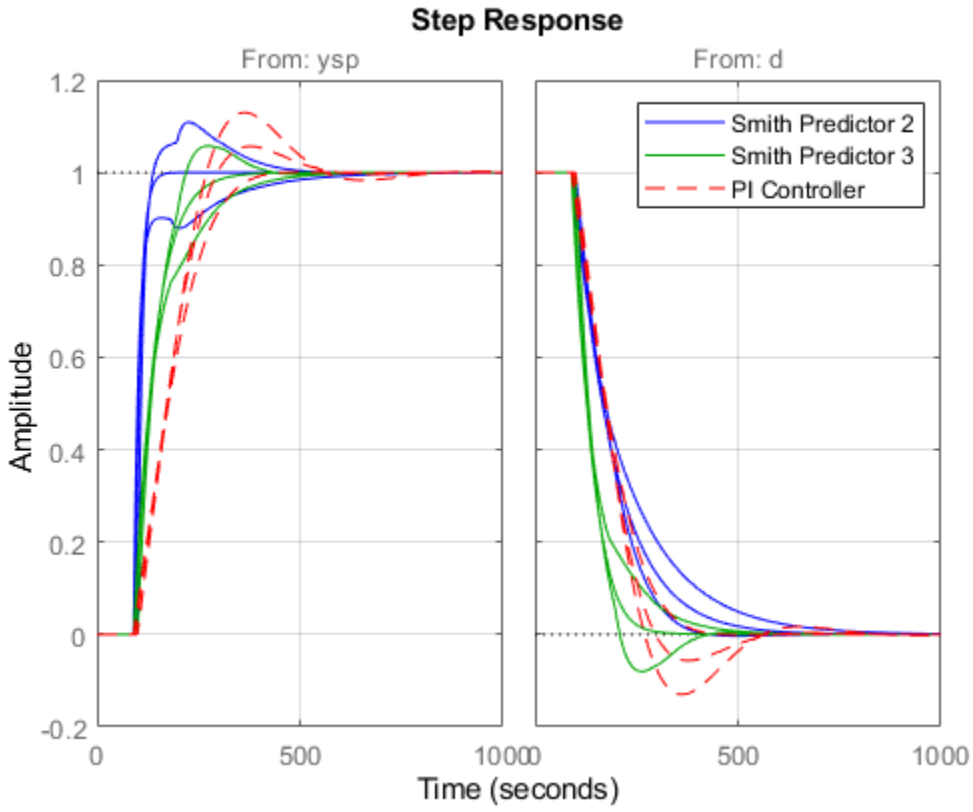
```
C =

             1      1
  Kp * (1 + ---- * ---)
             Ti     s

  with Kp = 0.144, Ti = 40.1

Continuous-time PI controller in standard form
```

Computed closed-loop model T3

```
T3 = connect(Plants,Gp,Dp,C,F,Sum1,Sum2,Sum3,{'ysp','d'},'y');
```

Compare T3 with T2 and Tpi

```
step(T2,'b',T3,'g',Tpi,'r--')
grid on
legend('Smith Predictor 2','Smith Predictor 3','PI Controller')
```



This comparison shows that our last design speeds up disturbance rejection at the expense of slower setpoint tracking.

Classical Control Design

- “Choosing a Control Design Approach” on page 12-3
- “Control System Designer Tuning Methods” on page 12-5
- “Design Requirements” on page 12-9
- “Feedback Control Architectures” on page 12-21
- “Design Multiloop Control System” on page 12-23
- “Multimodel Control Design” on page 12-32
- “Bode Diagram Design” on page 12-42
- “Root Locus Design” on page 12-55
- “Nichols Plot Design” on page 12-67
- “Edit Compensator Dynamics” on page 12-78
- “Design Compensator Using Automated Tuning Methods” on page 12-83
- “Analyze Designs Using Response Plots” on page 12-95
- “Compare Performance of Multiple Designs” on page 12-101
- “Design Hard-Disk Read/Write Head Controller” on page 12-105
- “Design Compensator for Plant Model with Time Delays” on page 12-116
- “Design Compensator for Systems Represented by Frequency Response Data” on page 12-122
- “Design Internal Model Controller for Chemical Reactor Plant” on page 12-126
- “Design LQG Tracker Using Control System Designer” on page 12-140
- “Export Design to MATLAB Workspace” on page 12-149
- “Generate Simulink Model for Control Architecture” on page 12-151
- “Tune Simulink Blocks Using Compensator Editor” on page 12-153
- “Single Loop Feedback/Prefilter Compensator Design” on page 12-158
- “Cascaded Multiloop Feedback Design” on page 12-164
- “Reference Tracking of DC Motor with Parameter Variations” on page 12-173
- “Getting Started with the Control System Designer” on page 12-178
- “Compensator Design for a Set of Plant Models” on page 12-186
- “Programmatically Initializing the Control System Designer” on page 12-191
- “DC Motor Control” on page 12-195
- “Feedback Amplifier Design” on page 12-206
- “Digital Servo Control of a Hard-Disk Drive” on page 12-223
- “Yaw Damper Design for a 747 Jet Aircraft” on page 12-238
- “Thickness Control for a Steel Beam” on page 12-251
- “Kalman Filtering” on page 12-265
- “State Estimation Using Time-Varying Kalman Filter” on page 12-273
- “Nonlinear State Estimation of a Degrading Battery System” on page 12-285

• “Parameter and State Estimation in Simulink Using Particle Filter Block” on page 12-298

Choosing a Control Design Approach

Control System Toolbox provides several approaches to tuning control systems. Use the following table to determine which approach best supports what you want to do.

	PID Tuning	Classical Control Design	Multiloop, Multiobjective Tuning
Architecture	PID loops with unit feedback (1-DOF and 2-DOF)	Control systems having SISO controllers in common single-loop or multiloop configurations (see "Feedback Control Architectures" on page 12-21)	Any architecture, 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
Interface	• Graphical tuning using **PID Tuner** (see "Designing PID Controllers with PID Tuner") • Programmatic tuning using `pidtune` (see "PID Controller Design at the Command Line" on page 11-2)	Graphical tuning using **Control System Designer**	• Graphical tuning using **Control System Tuner** • Programmatic tuning using `systune` (see "Programmatic Tuning")

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” on page 12-78

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” on page 12-42
- “Root Locus Design” on page 12-55
- “Nichols Plot Design” on page 12-67

Automated Tuning Methods

Use automated tuning methods to automatically tune compensators based on your design specifications.

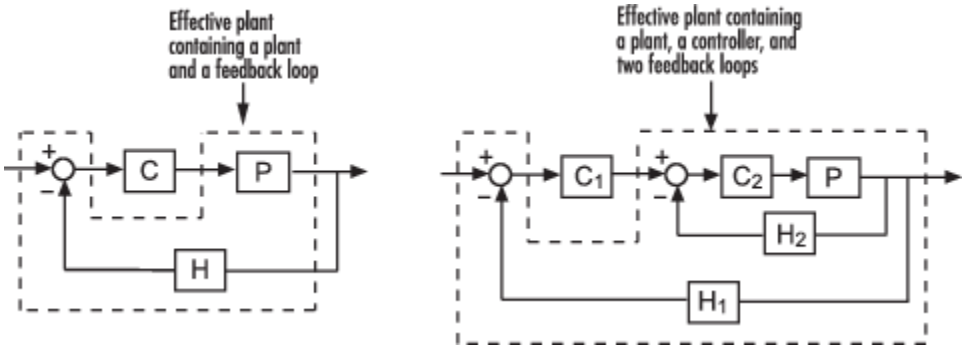
Tuning Method	Description	Requirements and Limitations
PID Tuning	Automatically tune PID gains to balance performance and robustness or tune controllers using classical PID tuning formulas.	Classical PID tuning formulas require a stable or integrating effective plant.
Optimization Based Tuning	Optimize 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” on page 12-83.

Effective Plant for Tuning

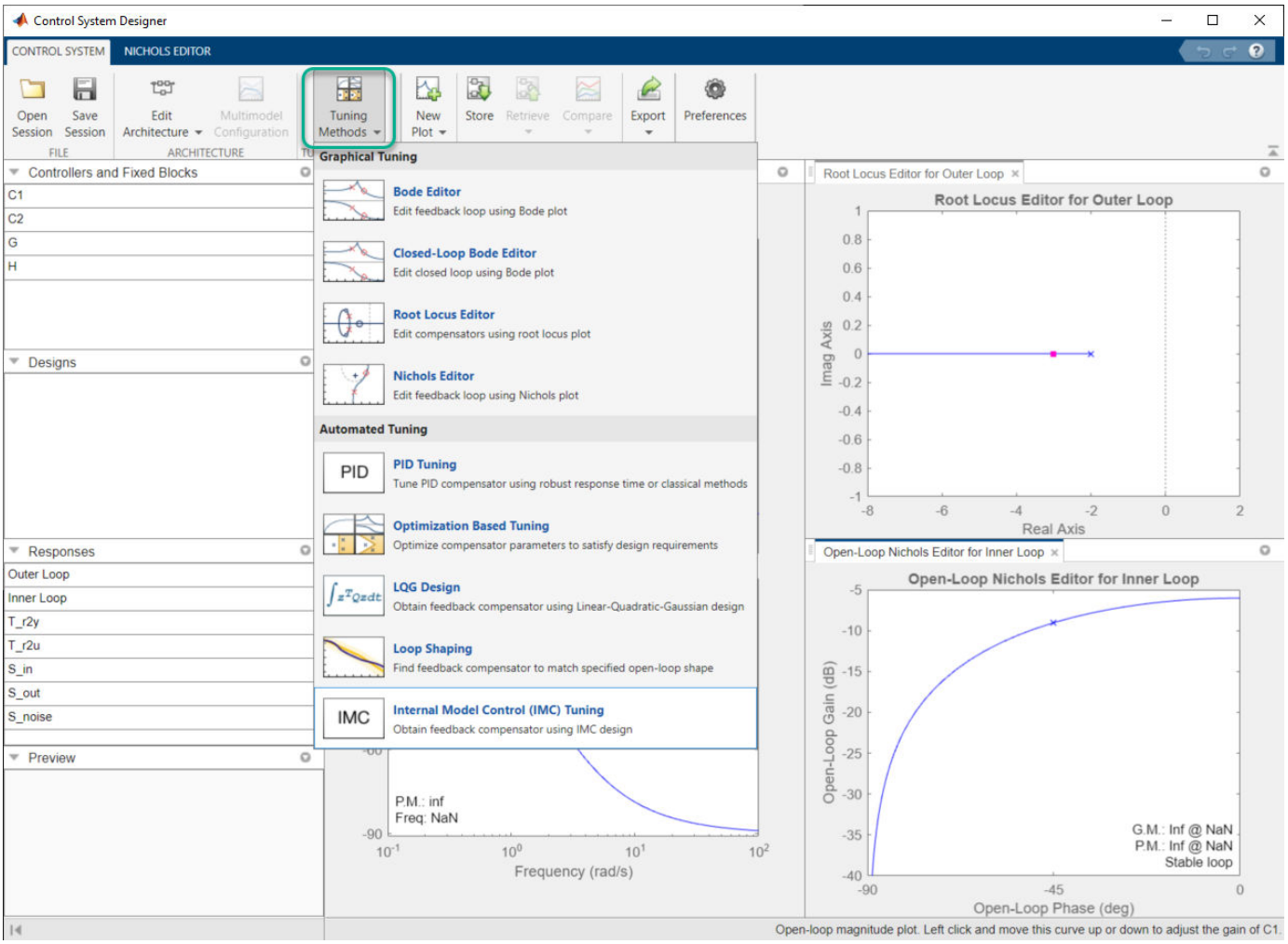
An *effective plant* is the system controlled by a compensator that contains all elements of the open loop in your model other than the compensator you are tuning. The following diagrams show examples of effective plants:



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.

Select a Tuning Method

To select a tuning method, in **Control System Designer**, click **Tuning Methods**.



See Also

Control System Designer

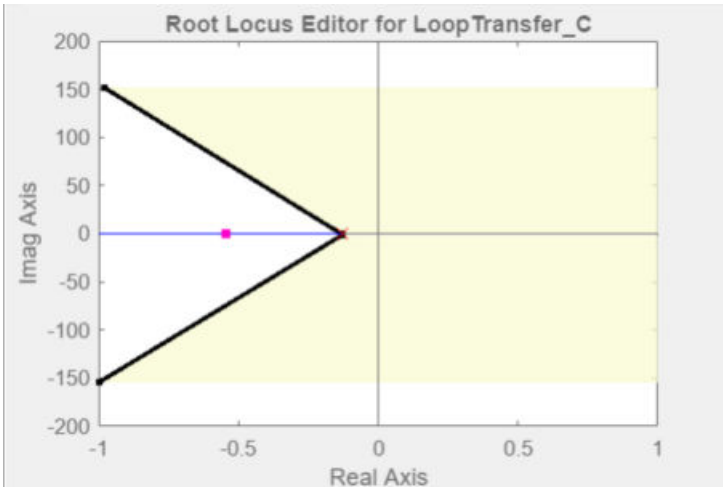
Related Examples

- “Bode Diagram Design” on page 12-42
- “Root Locus Design” on page 12-55

- “Nichols Plot Design” on page 12-67
- “Design Compensator Using Automated Tuning Methods” on page 12-83

Design Requirements

This topic describes time-domain and frequency-domain design requirements available in **Control System Designer**. Each requirement defines an exclusion region, indicated by a yellow shaded area. To satisfy a requirement, a response plot must remain outside of the associated exclusion region.



If you have Simulink Design Optimization software installed, you can use response optimization techniques to find a compensator that meets your specified design requirements. For examples of optimization-based control design using design requirements, see “Optimize LTI System to Meet Frequency-Domain Requirements” (Simulink Design Optimization) and “Design Optimization-Based PID Controller for Linearized Simulink Model (GUI)” (Simulink Design Optimization).

For other **Control System Designer** tuning methods, you can use the specified design requirements as visual guidelines during the tuning process.

Add Design Requirements

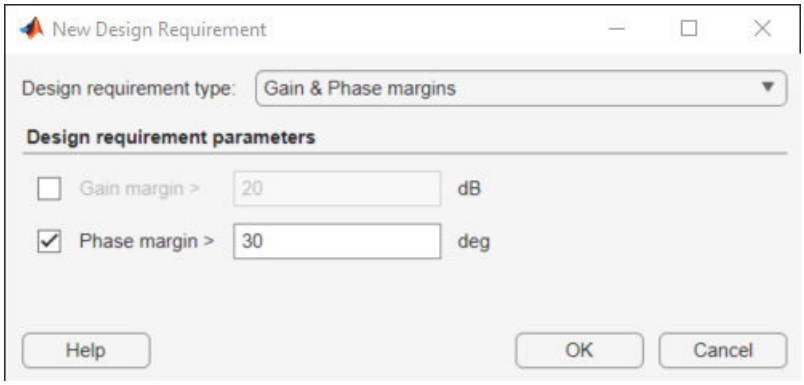
You can add design requirements either directly to existing plots or, when using optimization-based tuning, from the Response Optimization dialog box.

Add Requirements to Existing Plots

You can add design requirements directly to existing:

- Bode, root locus, and Nichols editor plots.
- Analysis plots:
  - Root locus plots and pole/zero maps
  - Bode diagrams
  - Nichols plots
  - Step and impulse responses

To add a design requirement to a plot, in **Control System Designer**, right-click the plot, and select **Design Requirements** > **New**.



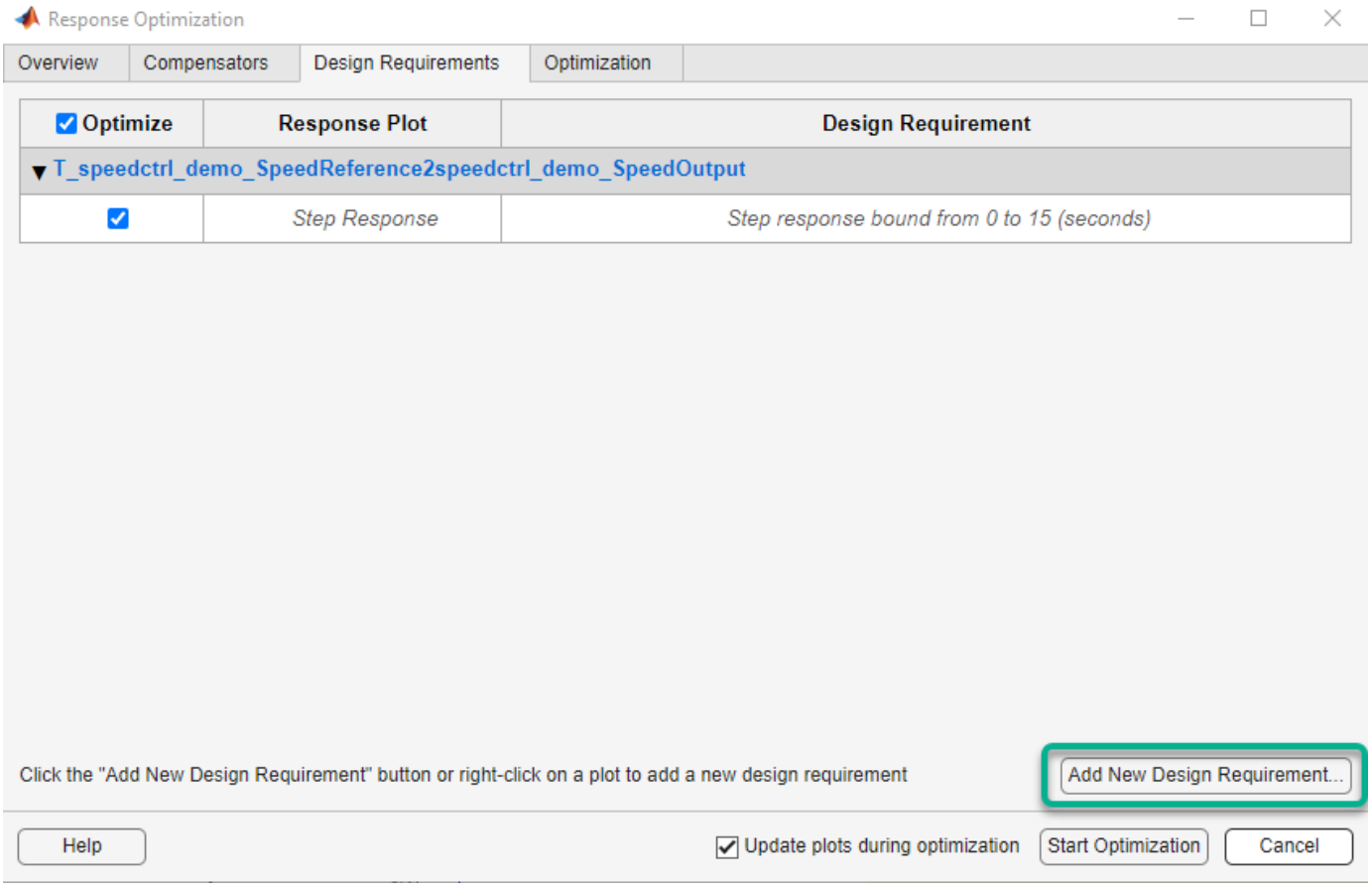
In the New Design Requirement dialog box, in the **Design requirement type** drop-down list, select the type of requirement to add. You can select any valid requirement for the associated plot type.

In the **Design requirement parameters** section, configure the requirement properties. Parameters are dependent on the type of requirement you select.

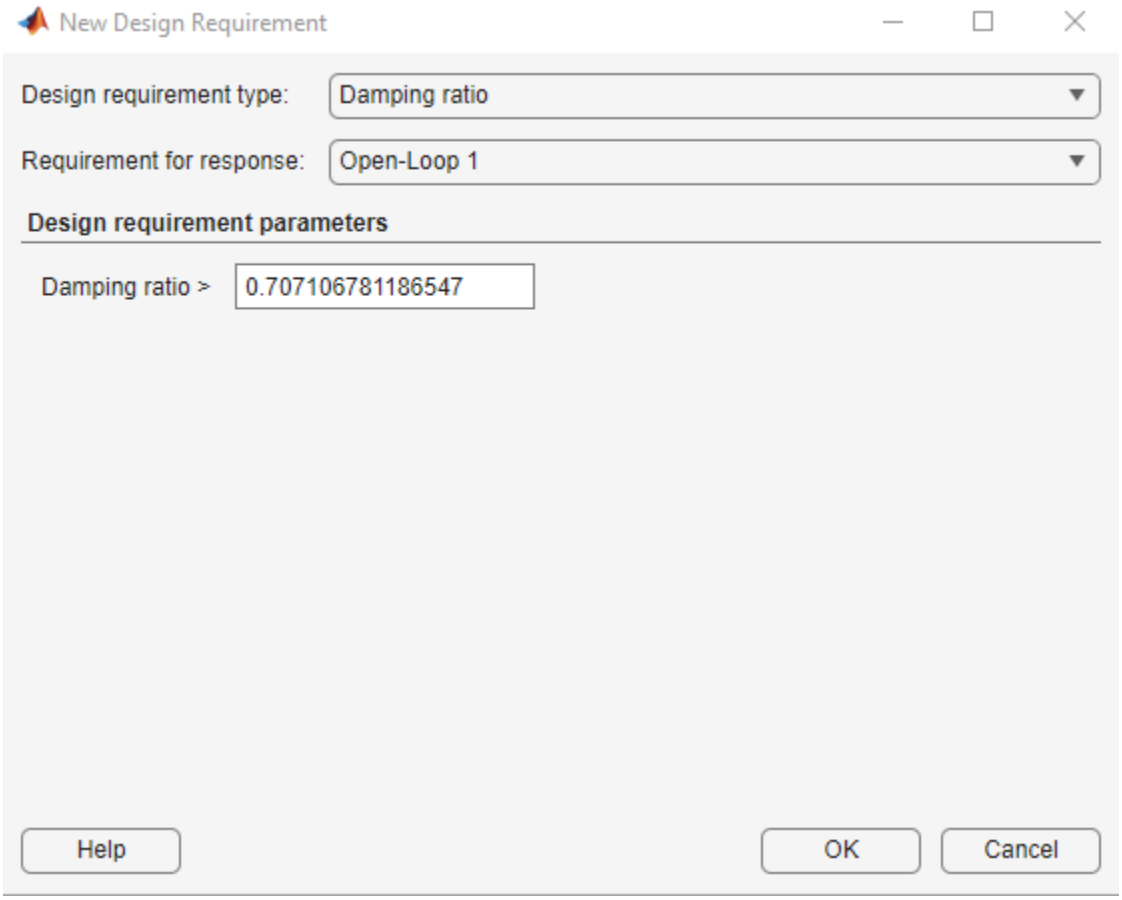
To create the specified requirement and add it to the plot, click **OK**.

Add Requirements from Response Optimization Dialog Box

When using optimization-based tuning, you can add design requirements from the Response Optimization dialog box.



To do so, on the **Design Requirements** tab, click **Add New Design Requirement**.



In the New Design Requirement dialog box, select a **Design requirement type** from the drop-down list.

In the **Requirement for response** drop-down list, specify the response to which to apply the design requirement. You can select any response in **Data Browser**.

In the **Design requirement parameters** section, configure the requirement properties. Parameters are dependent on the type of requirement you select.

To create the specified design requirement, click **OK**. In the Response Optimization dialog box, on the **Design Requirements** tab, the new requirement is added to the table.

The app also adds the design requirement to a corresponding editor or analysis plot. The plot type used depends on the selected design requirement type.

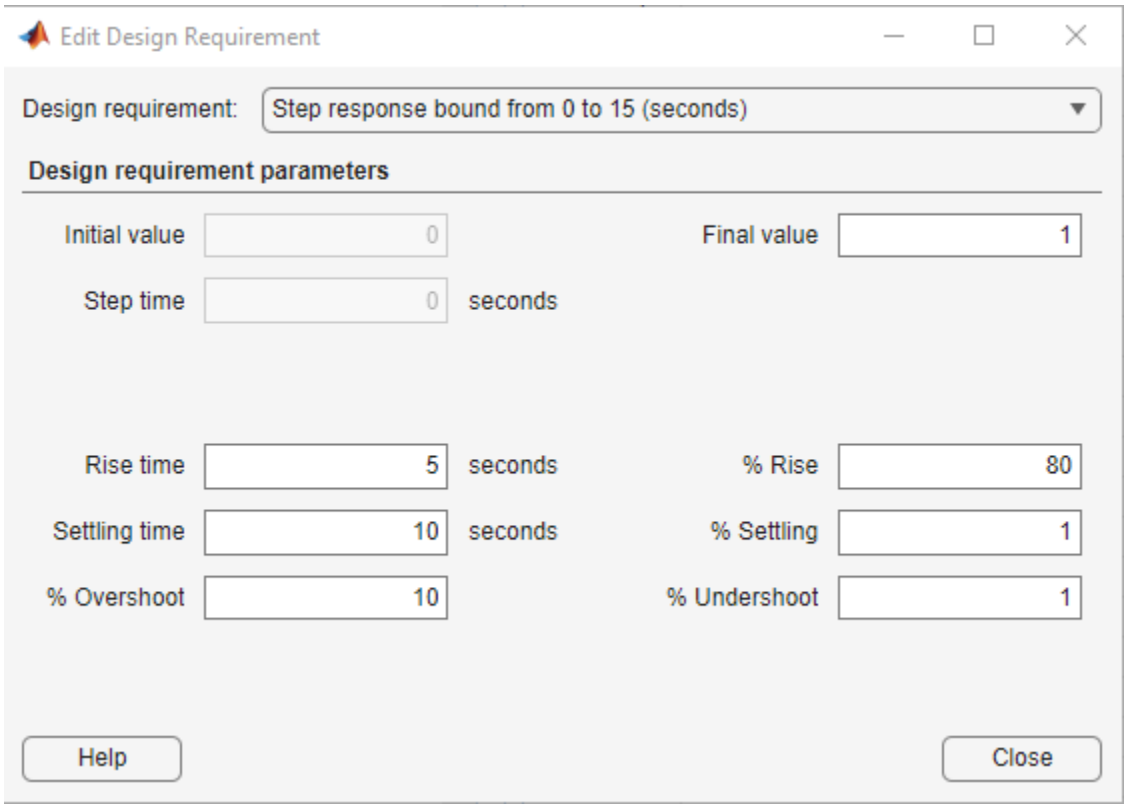
If the requirement is for a Bode, root locus, or Nichols plot and:

- A corresponding editor plot is open, the requirement is added to that plot.
- Only a corresponding analysis plot is open, the requirement is added to that plot.
- No corresponding plot is open, the requirement is added to a new Editor plot.

Otherwise, if the requirement is for a different plot type, the requirement is added to an appropriate analysis plot. For example, a `Step requirement bound` is added to a new step analysis plot.

Edit Design Requirements

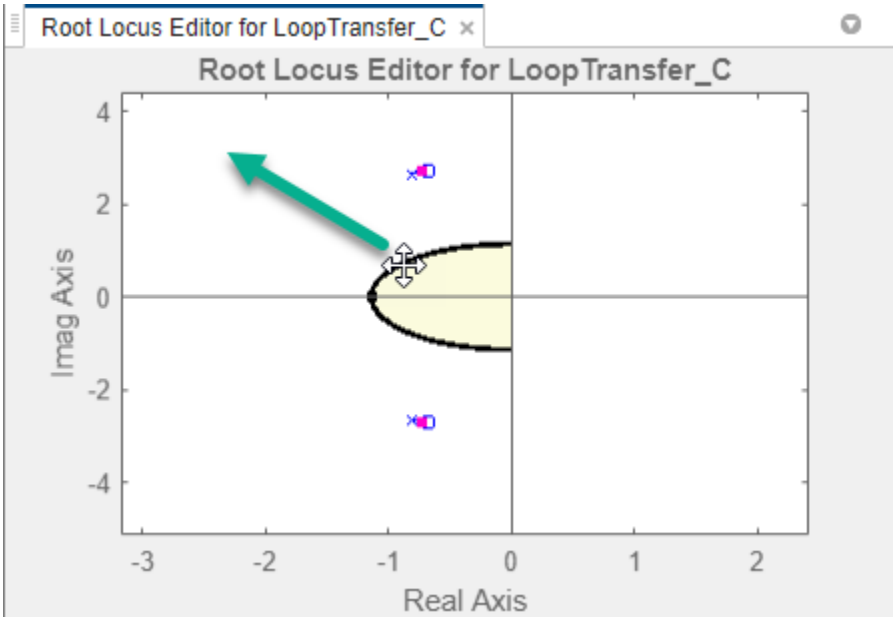
To edit an existing requirement, in **Control System Designer**, right-click the corresponding plot, and select **Design Requirements > Edit**.



In the Edit Design Requirement dialog box, in the **Design requirement** drop-down list, select a design requirement to edit. You can select any existing design requirement from the current plot.

In the **Design requirement parameters** section, specify the requirement properties. Parameters are dependent on the type of requirement you select. When you change a parameter, the app automatically updates the requirement display in the associated plot.

You can also interactively adjust design requirements by dragging the edges or vertices of the shaded exclusion region in the associated plot.



Root Locus and Pole-Zero Plot Requirements

Settling Time

Specifying a settling time for a continuous-time system adds a vertical boundary line to the root locus or pole-zero plot. This line represents pole locations associated with the specified settling time. This boundary is exact for a second-order system with no zeros. For higher order systems, the boundary is an approximation based on second-order dominant systems.

To satisfy this requirement, your system poles must be to the left of the boundary line.

For a discrete-time system, the design requirement boundary is a curved line centered on the origin. In this case, your system poles must be within the boundary line to satisfy the requirement.

Percent Overshoot

Specifying percent overshoot for a continuous-time system adds two rays to the plot that start at the origin. These rays are the locus of poles associated with the specified overshoot value. In the discrete-time case, the design requirement adds two curves originating at (1,0) and meeting on the real axis in the left-hand plane.

Note The percent overshoot (p.o.) design requirement can be expressed in terms of the damping ratio, ζ:

$$p.o. = 100\exp\left(-\frac{\pi\zeta}{\sqrt{1-\zeta^2}}\right)$$

Damping Ratio

Specifying a damping ratio for a continuous-time system adds two rays to the plot that start at the origin. These rays are the locus of poles associated with the specified overshoot value. This boundary

is exact for a second-order system and, for higher order systems, is an approximation based on second-order dominant systems.

To meet this requirement, your system poles must be to the left of the boundary lines.

For discrete-time systems, the design requirement adds two curves originating at (1,0) and meeting on the real axis in the left-hand plane. In this case, your system poles must be within the boundary curves.

Natural Frequency

Specifying a natural frequency bound adds a semicircle to the plot that is centered around the origin. The radius of the semicircle equals the natural frequency.

If you specify a natural frequency lower bound, the system poles must remain outside this semicircle. If you specify a natural frequency upper bound, the system poles must remain within this semicircle.

Region Constraint

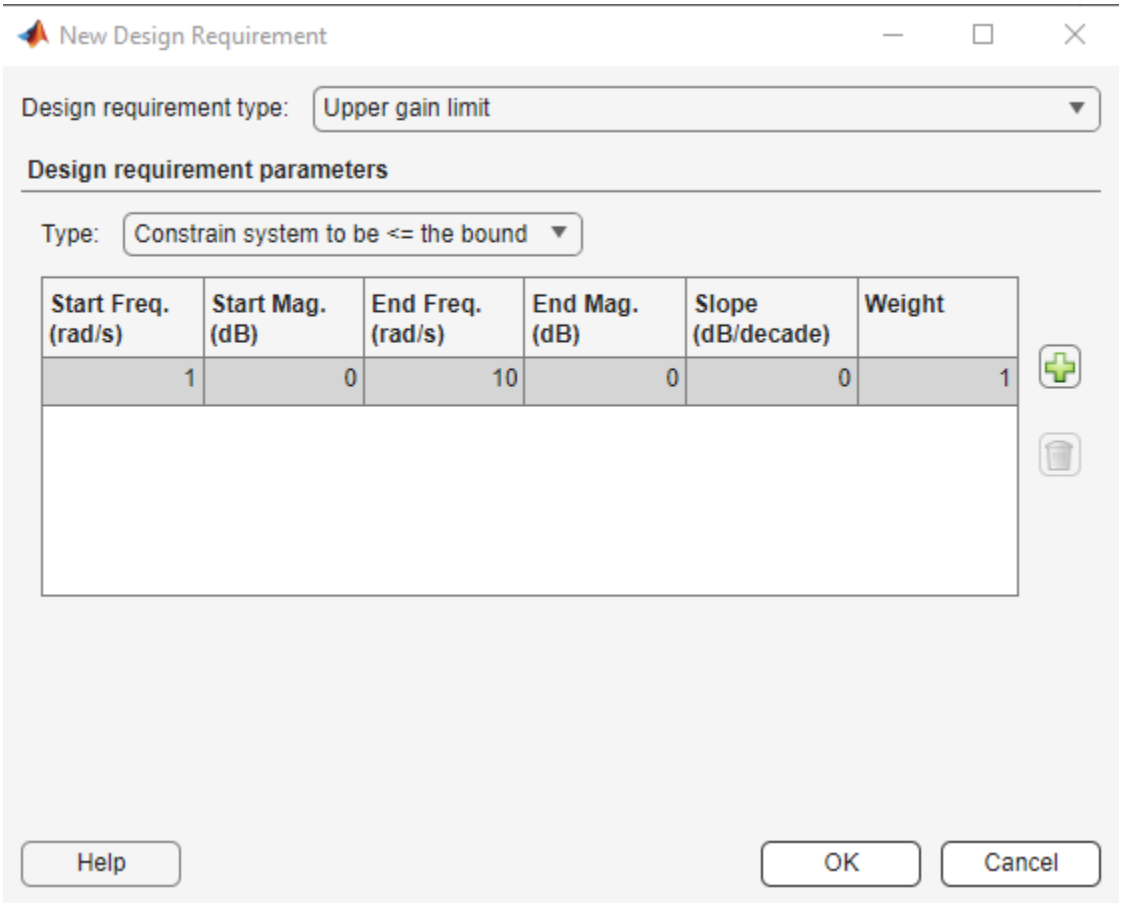
To specify a region constraint, define two or more vertices of a piece-wise linear boundary line. For each vertex, specify **Real** and **Imaginary** components. This requirement adds a shaded exclusion region on one side of the boundary line. To switch the exclusion region to the opposite side of the boundary, in the response plot, right-click the requirement, and select **Flip**.

To satisfy this requirement, your system poles must be outside of the exclusion region.

Open-Loop and Closed-Loop Bode Diagram Requirements

Upper Gain Limit

You can specify upper gain limits for both open-loop and closed-loop Bode responses.



A gain limit consists of one or more line segments. For the start and end points of each segment, specify a frequency, **Freq**, and magnitude, **Mag**. You can also specify the slope of the line segment in dB/decade. When you change the slope, the magnitude for the end point updates.

If you are using optimization-based tuning, you can assign a tuning **Weight** to each segment to indicate their relative importance.

In the **Type** drop-down list you can select whether to constrain the magnitude to be above or below the specified boundary.

Lower Gain Limit

You can specify lower gain limits in the same way as upper gain limits.

Gain and Phase Margin

You can specify a lower bound for the gain margin, the phase margin, or both. The specified bounds appear in text on the Bode magnitude plot.

Note Gain and phase margin requirements are only applicable to open-loop Bode diagrams.

Open-Loop Nichols Plot Requirements

Phase Margin

Specify a minimum phase margin as a positive value. Graphically, **Control System Designer** displays this requirement as a region of exclusion along the 0 dB open-loop gain axis.

Gain Margin

Specify a minimum gain margin value. Graphically, **Control System Designer** displays this requirement as a region of exclusion along the -180 degree open-loop phase axis.

Closed-Loop Peak Gain

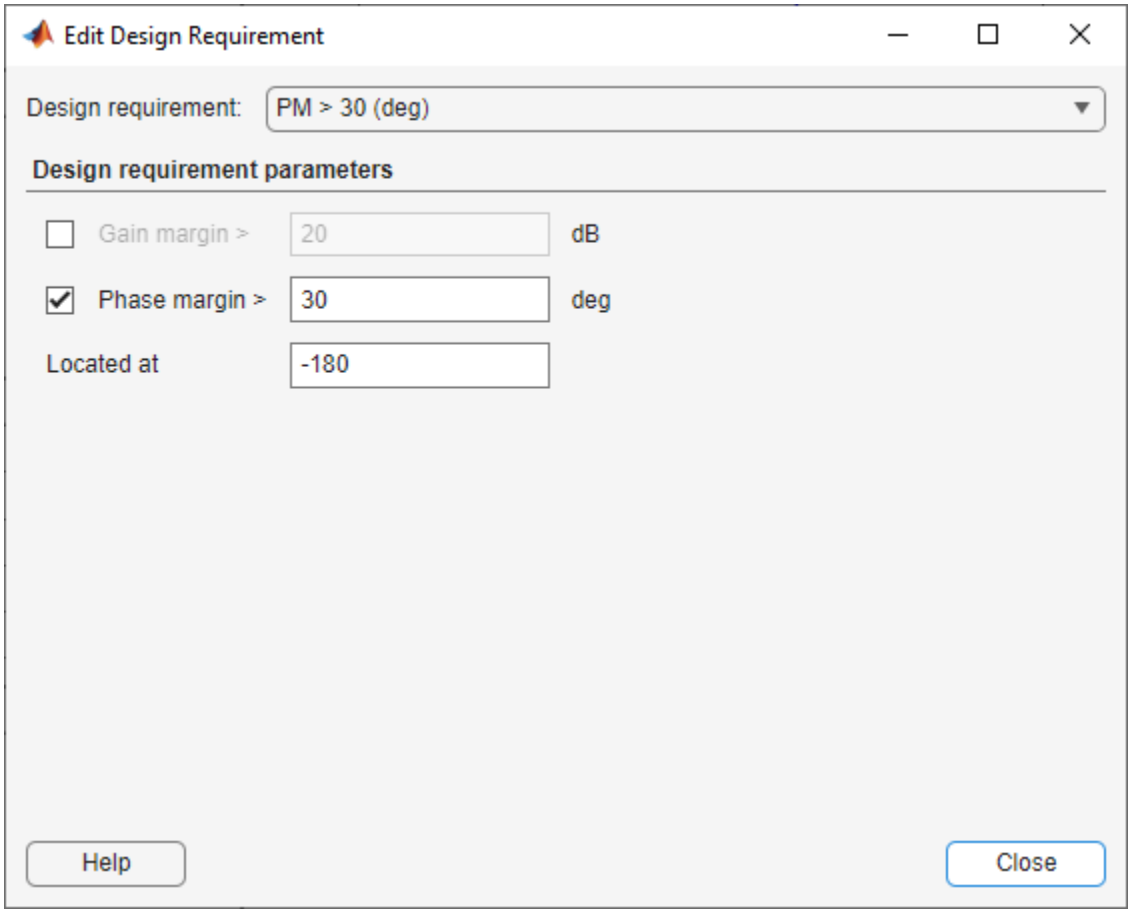
Specify a minimum closed-loop peak gain value. The specified dB value can be positive or negative. The design requirement follows the curves of the Nichols plot grid. As a best practice, have the grid on when using a closed-loop peak gain requirement.

Gain-Phase Design Requirement

To specify a gain-phase design requirement, define two or more vertices of a piece-wise linear boundary line. For each vertex, specify **Open-Loop phase** and **Open-Loop gain** values. This requirement adds a shaded exclusion region on one side of the boundary line. To switch the exclusion region to the opposite side of the boundary, in the Nichols plot, right-click the requirement, and select **Flip**.

Display Location

When editing a phase margin, gain margin, or closed-loop peak gain requirement, you can specify the display location as $-180 \pm k360$ degrees, where k is an integer value.

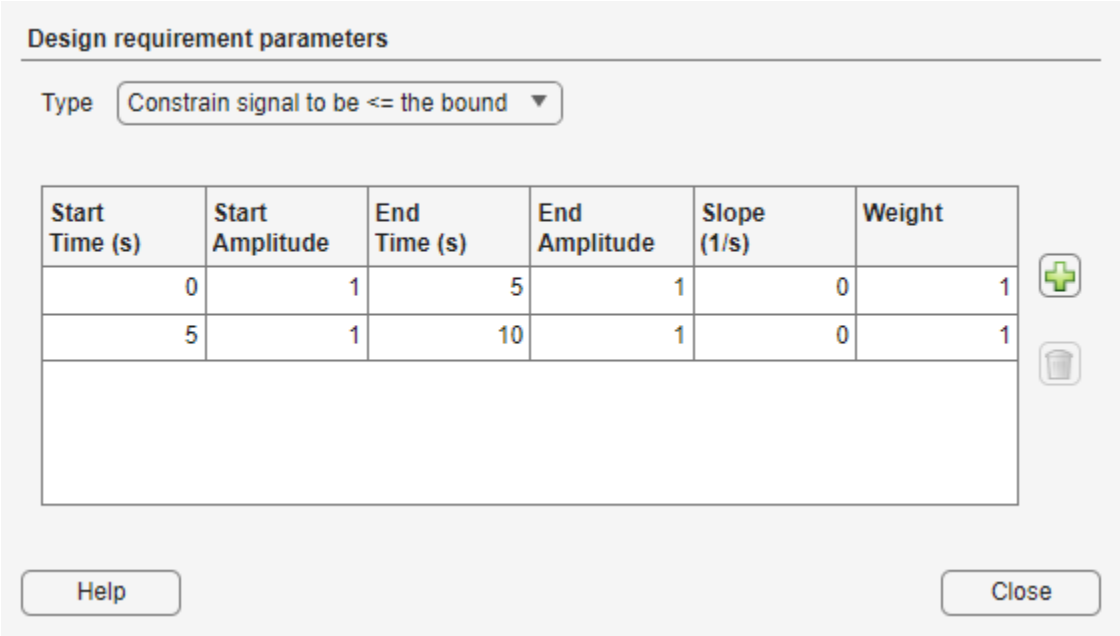


If you enter an invalid location, the closest valid location is selected. While displayed graphically at only one location, these requirements apply regardless of actual phase; that is, they are applied for all values of k.

Step and Impulse Response Requirements

Upper Time Response Bound

You can specify upper time response bounds for both step and impulse responses.



A time-response bound consists of one or more line segments. For the start and end points of each segment, specify a **Time** and **Amplitude** value. You can also specify the slope of the line segment. When you change the slope, the amplitude for the end point updates.

If you are using optimization-based tuning, you can assign a tuning **Weight** to each segment to indicate its relative importance.

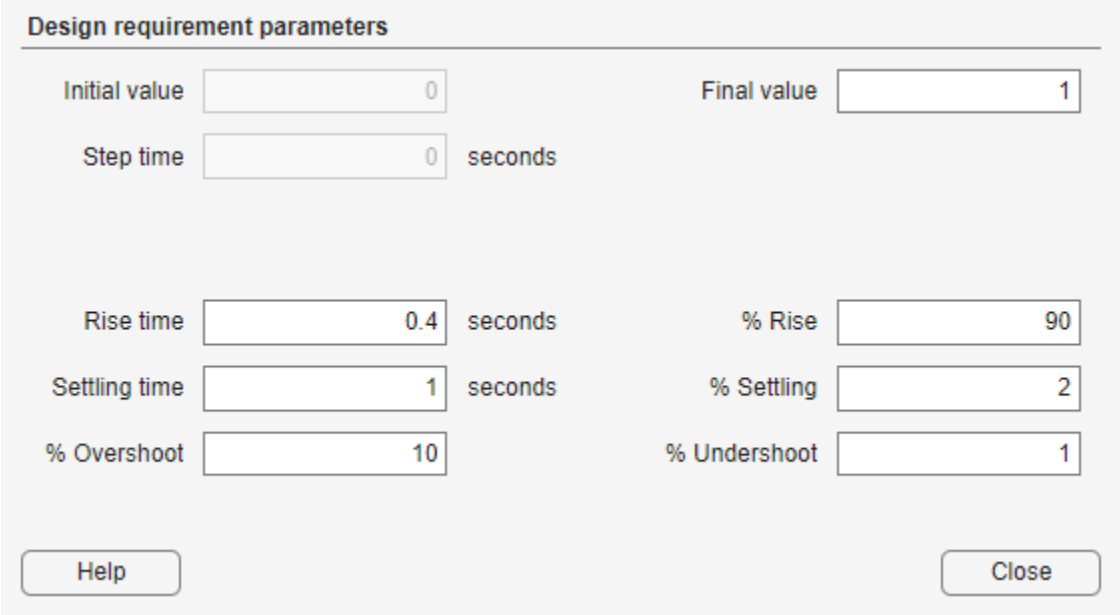
In the **Type** drop-down list, you can select whether to constrain the response to be above or below the specified boundary.

Lower Time Response Bound

You can specify lower time response bounds for both step and impulse responses in the same way as upper gain limits.

Step Response Bound

For a step response plot, you can also specify a step response bound design requirement.



To define a step response bound requirement, specify the following step response parameters:

- **Final value** — Final steady-state value
- **Rise time** — Time required to reach the specified percentage, **% Rise**, of the **Final value**
- **Settling time** — Time at which the response enters and stays within the settling percentage, **% Settling**, of the **Final value**
- **% Overshoot** — Maximum percentage overshoot above the **Final value**
- **% Undershoot** — Maximum percentage undershoot below the **Initial value**

In **Control System Designer**, step response plots always use an **Initial value** and a **Step time** of `0`

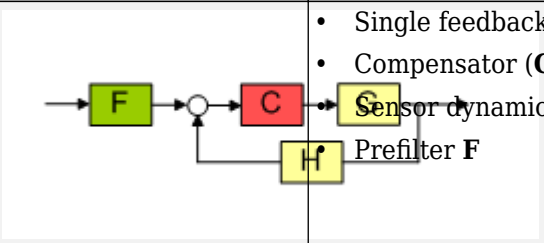
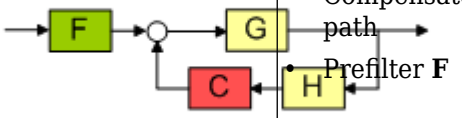
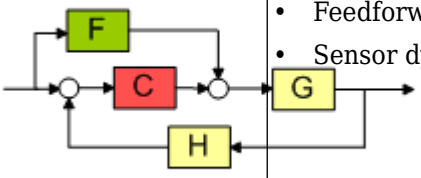
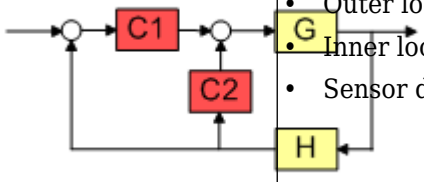
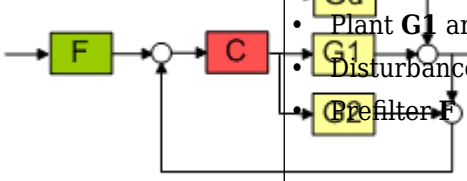
See Also

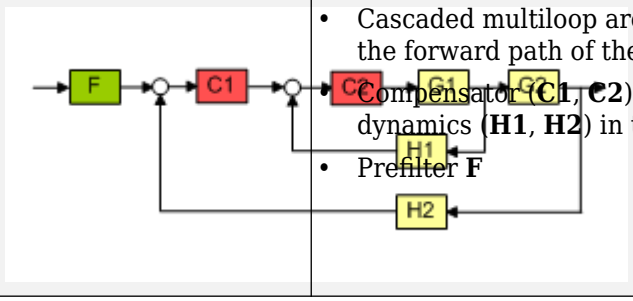
More About

- “Optimize LTI System to Meet Frequency-Domain Requirements” (Simulink Design Optimization)
- “Design Optimization-Based PID Controller for Linearized Simulink Model (GUI)” (Simulink Design Optimization)

Feedback Control Architectures

When you open **Control System Designer** from MATLAB, you can select one of six possible control architecture configurations.

Configuration	Block Diagram	Features
1		• Single feedback loop • Compensator (**C**) and plant (**G**) in forward path • Sensor dynamics (**H**) in feedback path • Prefilter **F**
2		• Single feedback loop • Compensator (**C**) and sensor dynamics (**H**) in feedback path • Prefilter **F**
3		• Compensator (**C**) and plant (**G**) in forward path • Feedforward prefilter **F** for input disturbance attenuation • Sensor dynamics (**H**) in feedback path
4		• Nested multiloop architecture • Outer loop with compensator (**C1**) in forward path • Inner loop with compensator (**C2**) in feedback path • Sensor dynamics (**H**) in feedback path
5		• Standard Internal Model Control (IMC) architecture • Compensator (**C**) in forward path • Plant **G1** and plant predictive model **G2** • Disturbance model **Gd** • Prefilter **F**

Configuration	Block Diagram	Features
6		• Cascaded multiloop architecture with the inner loop in the forward path of the outer loop. • Compensator (**C1**, **C2**) in the forward path and sensor dynamics (**H1**, **H2**) in the feedback path (both loops) • Prefilter **F**

If your control application does not match one of the supported control architectures, you can use block diagram algebra to convert your system to match an architecture. For an example of such an application, see “Design Multiloop Control System” on page 12-23.

Note If you are unable to match your application to one of the supported control architectures, consider using the **Control System Tuner** app to design your control system.

See Also

Control System Designer | `sisoinit`

More About

- “Design Multiloop Control System” on page 12-23

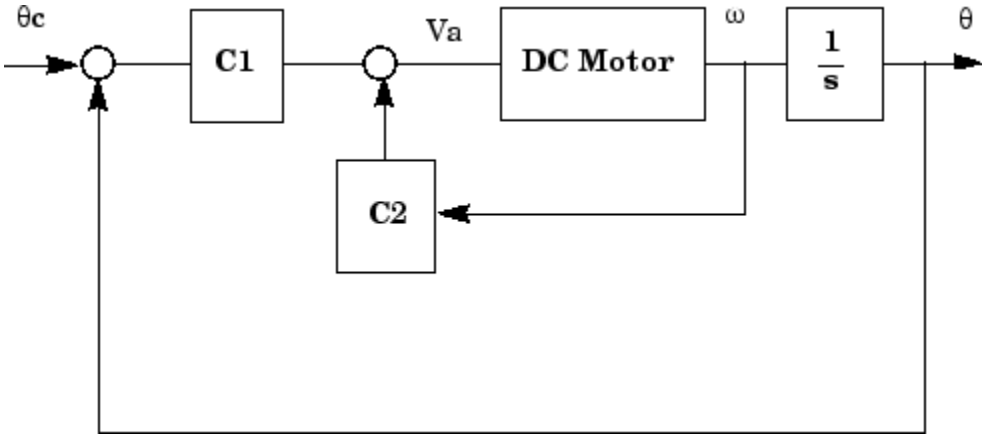
Design Multiloop Control System

In many applications, a single-loop control system is not feasible due to your plant design or design requirements. If you have a design with an inner and outer loop, you can use **Control System Designer** to design compensators for both loops.

The typical workflow is to tune the compensator for the inner loop first, by isolating the inner loop from the rest of the control system. Once the inner loop is satisfactorily tuned, tune the outer loop to achieve your desired closed-loop response.

System Model

For this example develop a position control system for a DC motor. A single-loop angular velocity controller is designed in “Bode Diagram Design” on page 12-42. To design an angular position controller, add an outer loop that contains an integrator.



Define a state-space plant model, as described in “SISO Example: The DC Motor”.

```
% Define the motor parameters
R = 2.0
L = 0.5
Km = .015
Kb = .015
Kf = 0.2
J = 0.02
% Create the state-space model
A = [-R/L -Kb/L; Km/J -Kf/J]
B = [1/L; 0];
C = [0 1];
D = [0];
sys_dc = ss(A,B,C,D);
```

Design Objectives

The design objective is to minimize the closed-loop step response settling time, while maintaining an inner-loop phase margin of at least 65 degrees with maximum bandwidth:

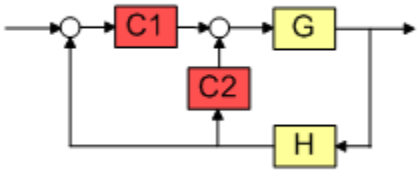
- Minimal closed-loop step response settling time.
- Inner-loop phase margin of at least 65 degrees.

- Maximum inner-loop bandwidth.

Match System to Control Architecture

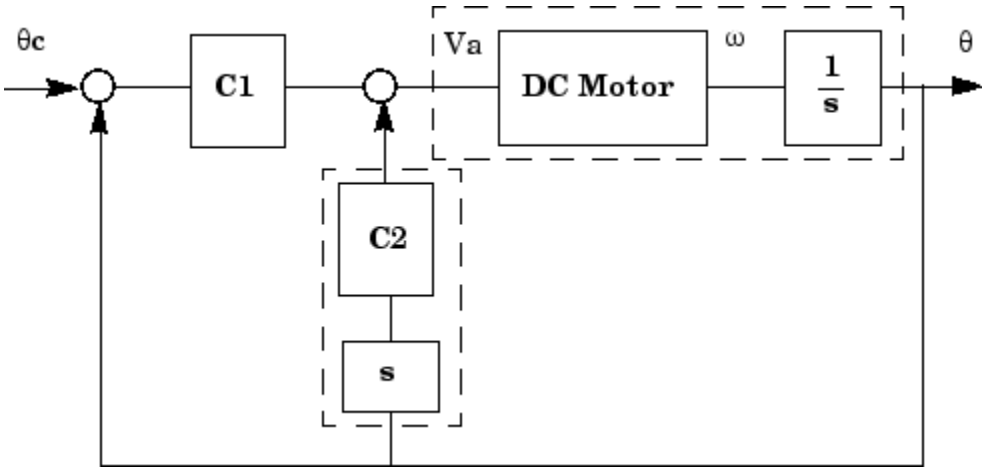
Control System Designer has six possible control architectures from which you can choose. For more information on these architectures, see “Feedback Control Architectures” on page 12-21.

For this example use Configuration 4, which has an inner and outer control loop.



Currently, the control system structure does not match Configuration 4. However, using block diagram algebra, you can modify the system model by adding:

- An integrator to the motor output to get the angular displacement.
- A differentiator to the inner-loop feedback path.



At the MATLAB command line, add the integrator to the motor plant model.

```
plant = sys_dc*tf(1,[1,0]);
```

Create an initial model of the inner-loop compensator that contains the feedback differentiator.

```
Cdiff = tf('s');
```

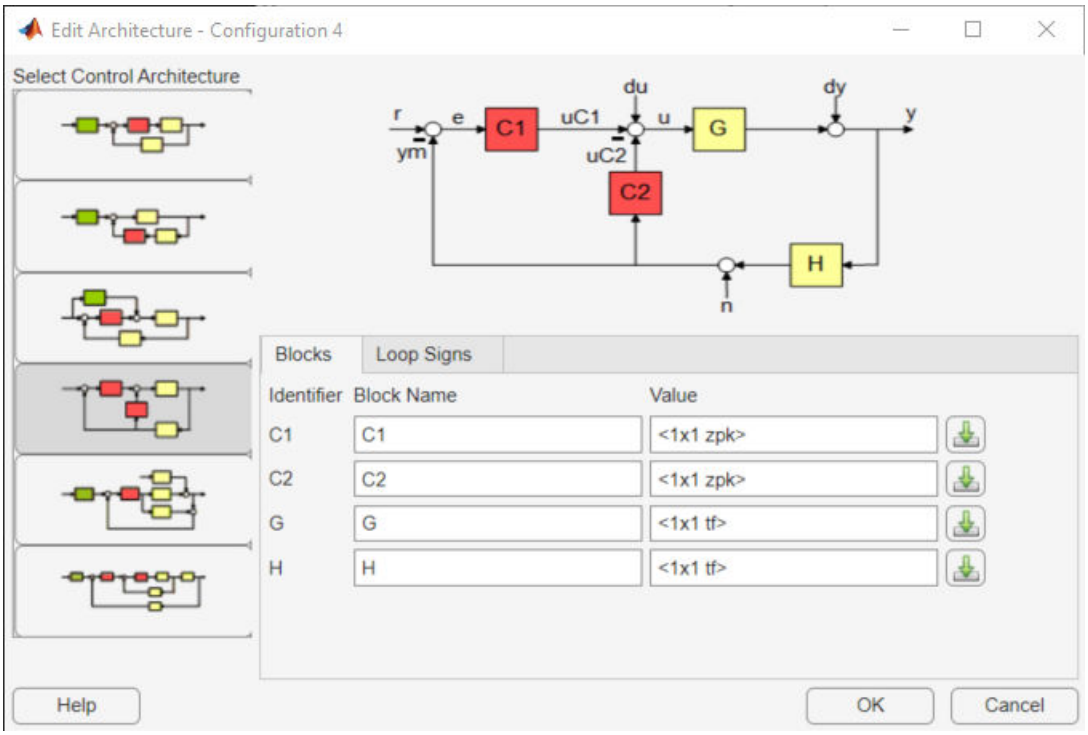
Define Control Architecture

Open **Control System Designer**.

```
controlSystemDesigner
```

In **Control System Designer**, on the **Control System** tab, click **Edit Architecture**.

In the Edit Architecture dialog box, under **Select Control Architecture**, click the fourth architecture.



Import the plant and controller models from the MATLAB workspace.

In the **Blocks** tab, for:

- Controller **C2**, specify a **Value** of `Cdiff`.
- Plant **G**, specify a **Value** of `plant`.

Click **OK**.

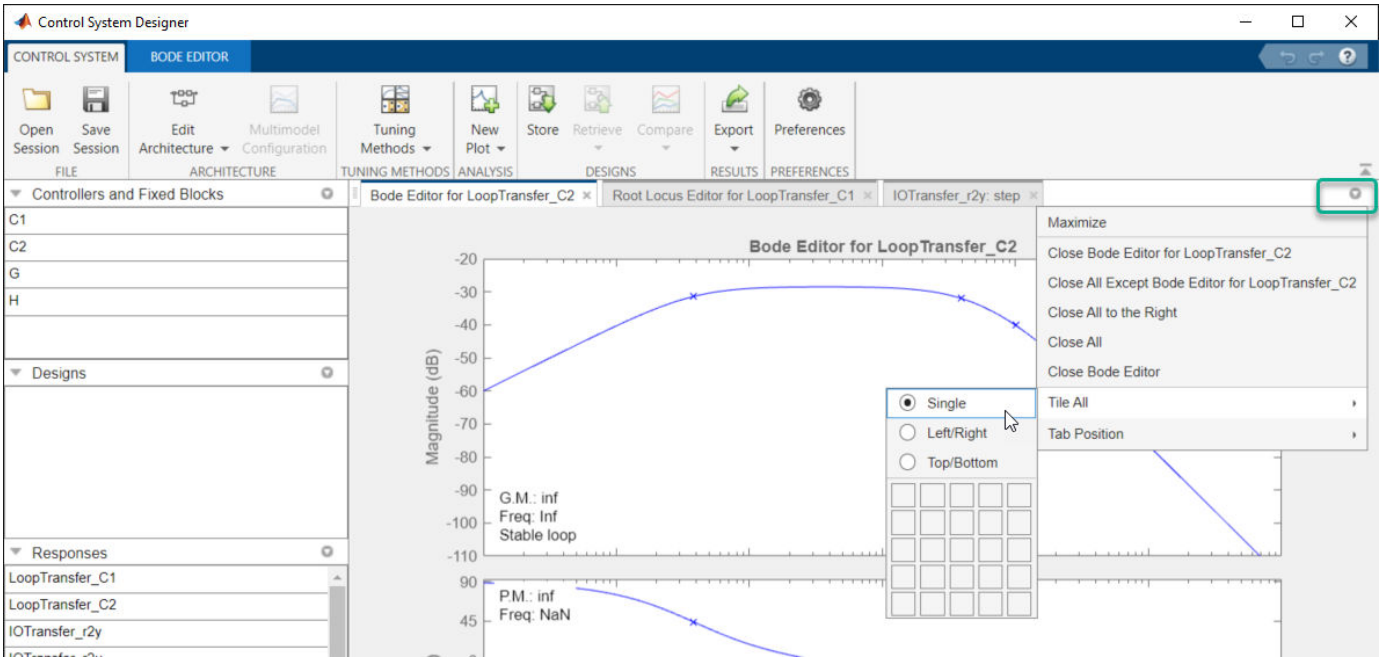
The app updates the control architecture and imports the specified models for the motor plant and the inner-loop controller.

In **Control System Designer**, the following plots open:

- **Bode Editor for LoopTransfer_C1** — Open-loop Bode Editor for the outer loop
- **Root Locus Editor for LoopTransfer_C1** — Open-loop Root Locus Editor for the outer loop
- **Bode Editor for LoopTransfer_C2** — Open-loop Bode Editor for the inner loop
- **Root Locus Editor for LoopTransfer_C2** — Open-loop root Locus Editor for the inner loop
- **IOTransfer_r2y: step** — Overall closed-loop step response from input *r* to output *y*

For this example, close the **Bode Editor for LoopTransfer_C1** and **Root Locus Editor for LoopTransfer_C2** plots.

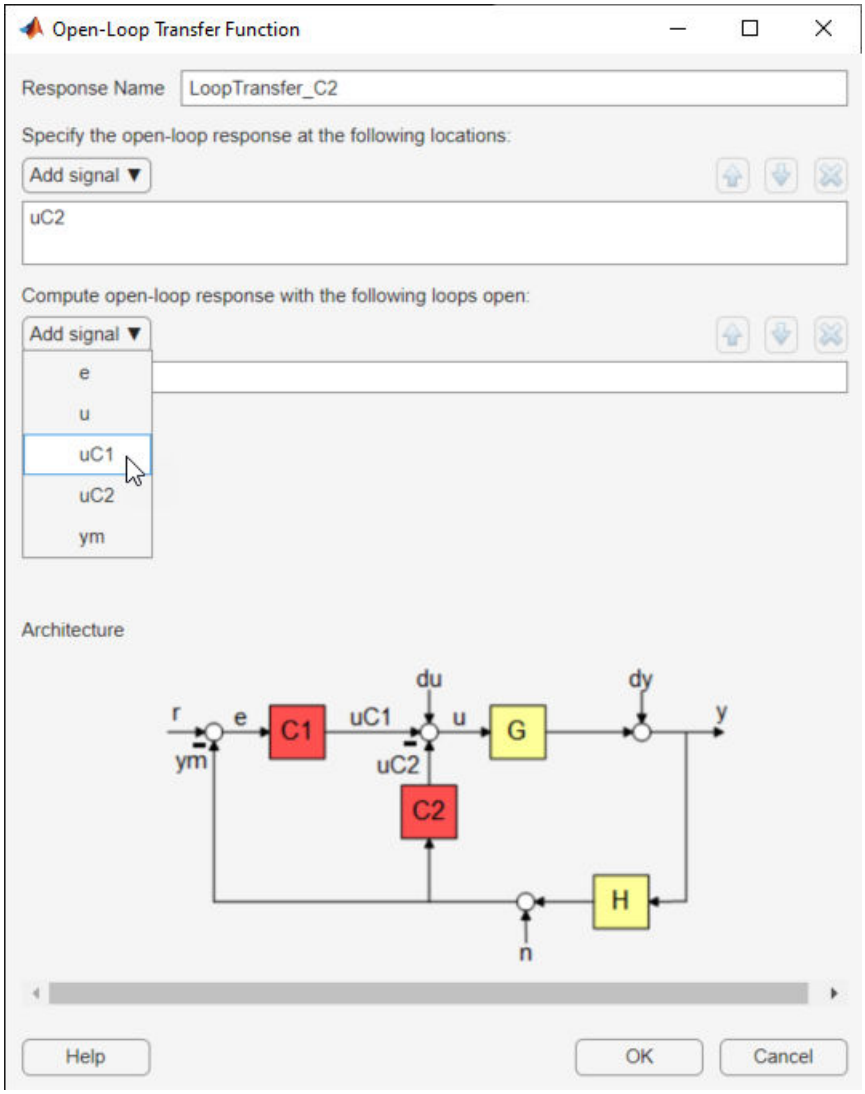
Since the inner loop is tuned first, configure the plots to view just the inner-loop Bode editor plot. To the side of the plot tab, click arrow and select **Tile All**, then click **Single**.



Isolate Inner Loop

To isolate the inner loop from the rest of the control system architecture, add a loop opening to the open-loop response of the inner loop. In the data browser, right-click `LoopTransfer_C2`, and select **Open Selection**.

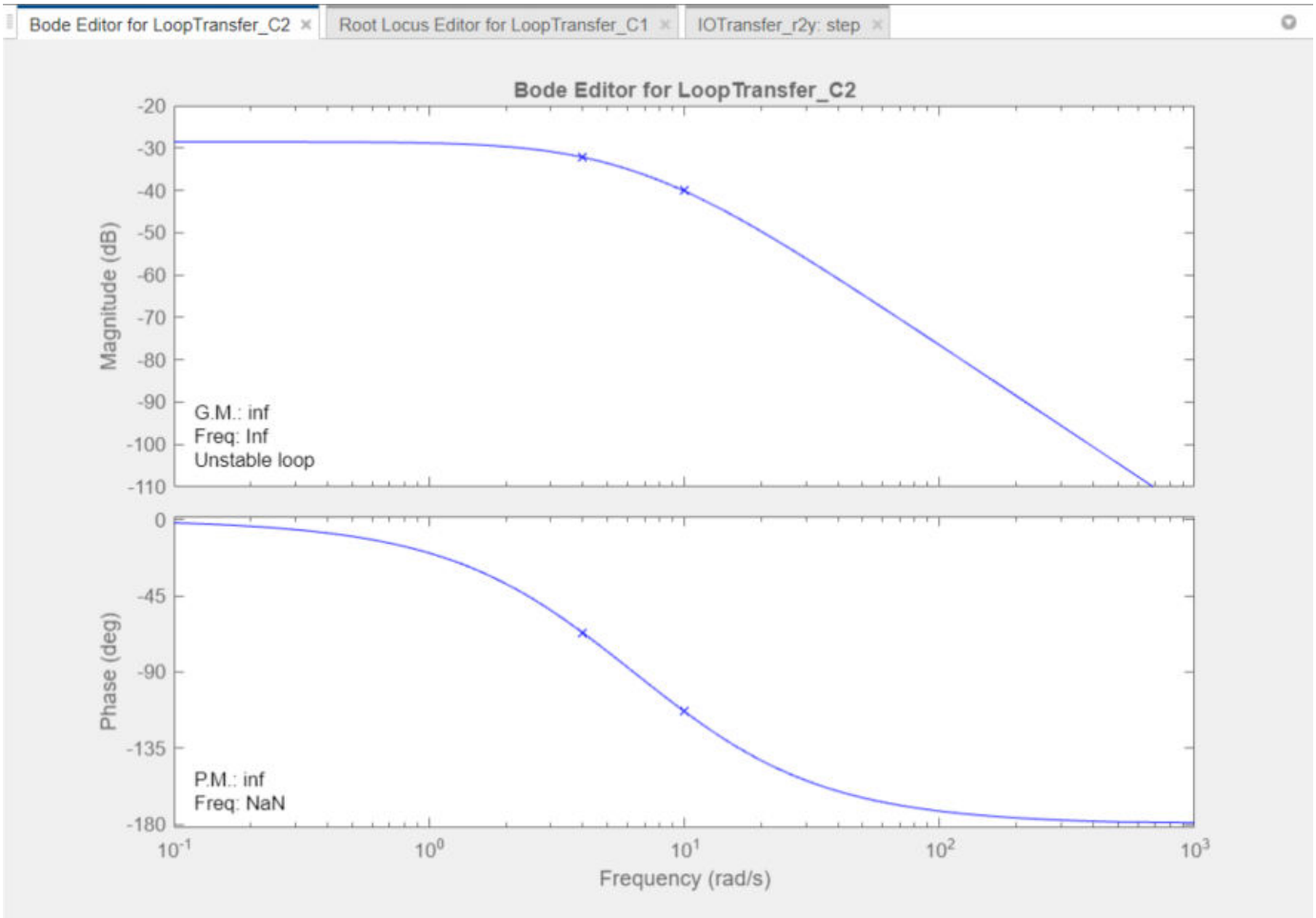
To add a loop opening at the output of outer-loop compensator, **C1**, in the Open-Loop Transfer Function dialog box, click ➕ **Add loop opening location to list**. Then, select **uC1**.



Click **OK**.

The app adds a loop opening at the selected location. This opening removes the effect of the outer control loop on the open-loop transfer function of the inner loop.

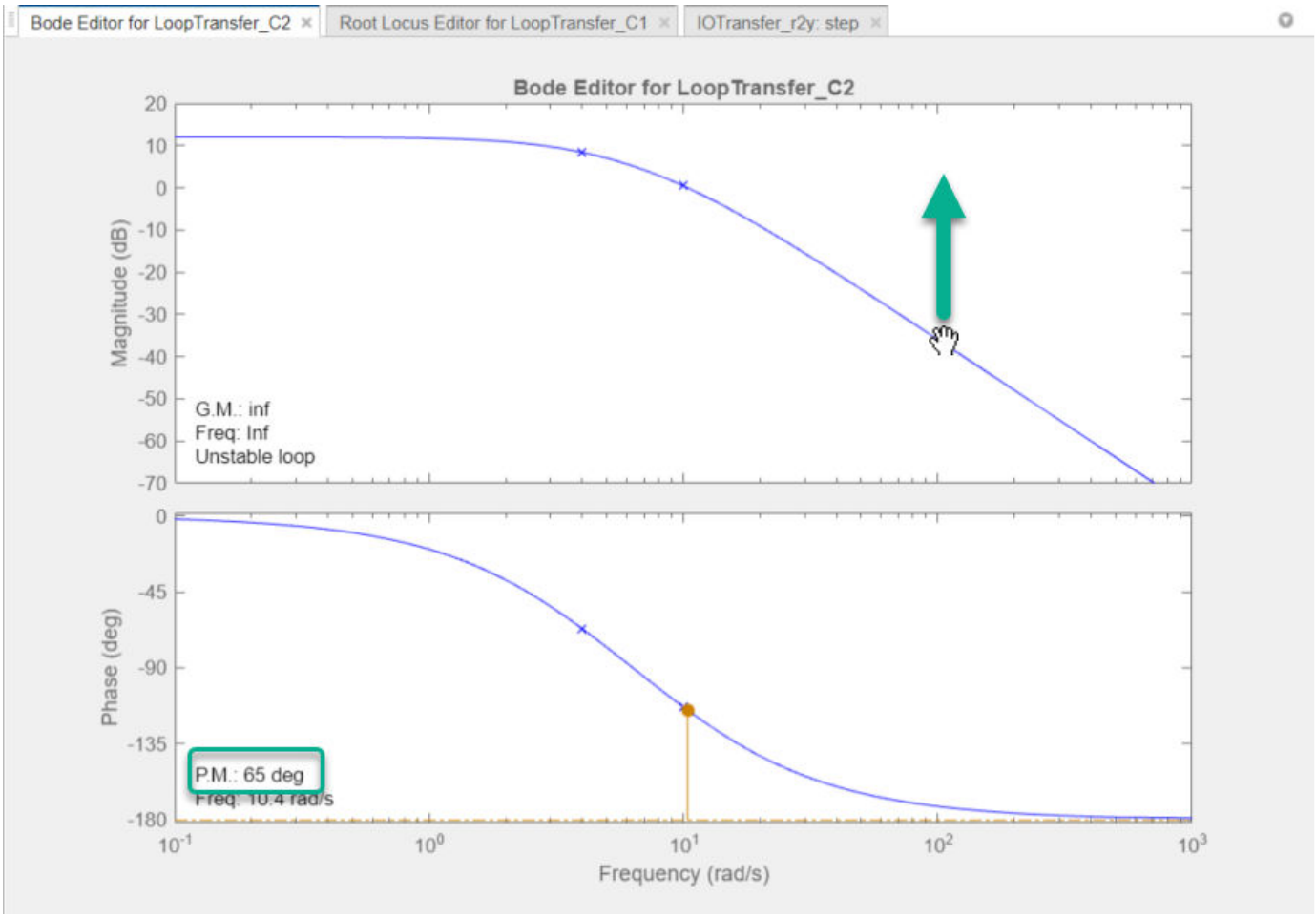
The **Bode Editor** response plot updates to reflect the new open-loop transfer function.



Tune Inner Loop

To increase the bandwidth of the inner loop, increase the gain of compensator **C2**.

In the **Bode Editor** plot, drag the magnitude response upward until the phase margin is 65 degrees. This corresponds to a compensator gain of `107`. Increasing the gain further reduces the phase margin below 65 degrees.



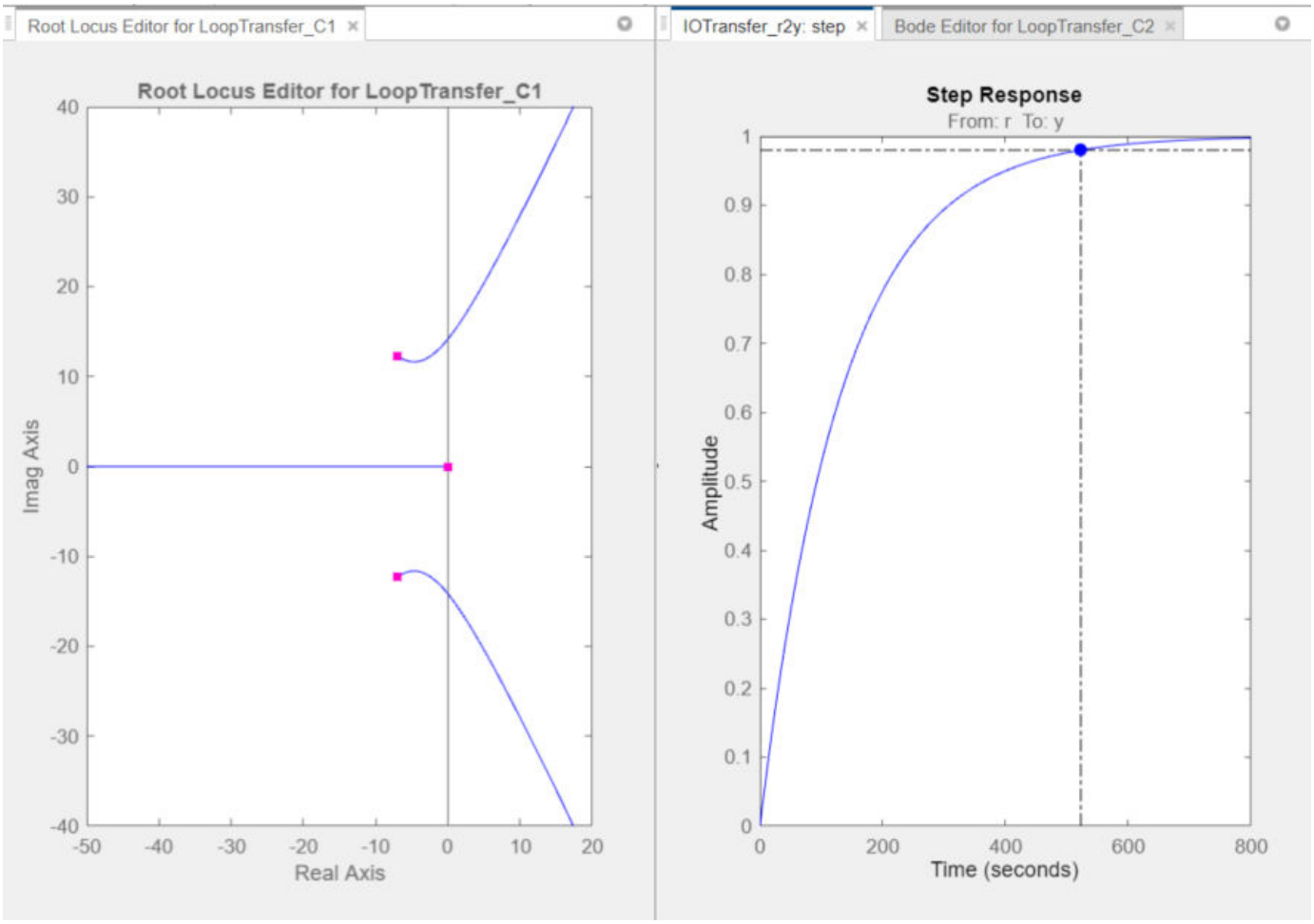
Alternatively, you can adjust the gain value using the compensator editor. For more information, see “Edit Compensator Dynamics” on page 12-78.

Tune Outer Loop

With the inner loop tuned, you can now tune the outer loop to reduce the closed-loop settling time.

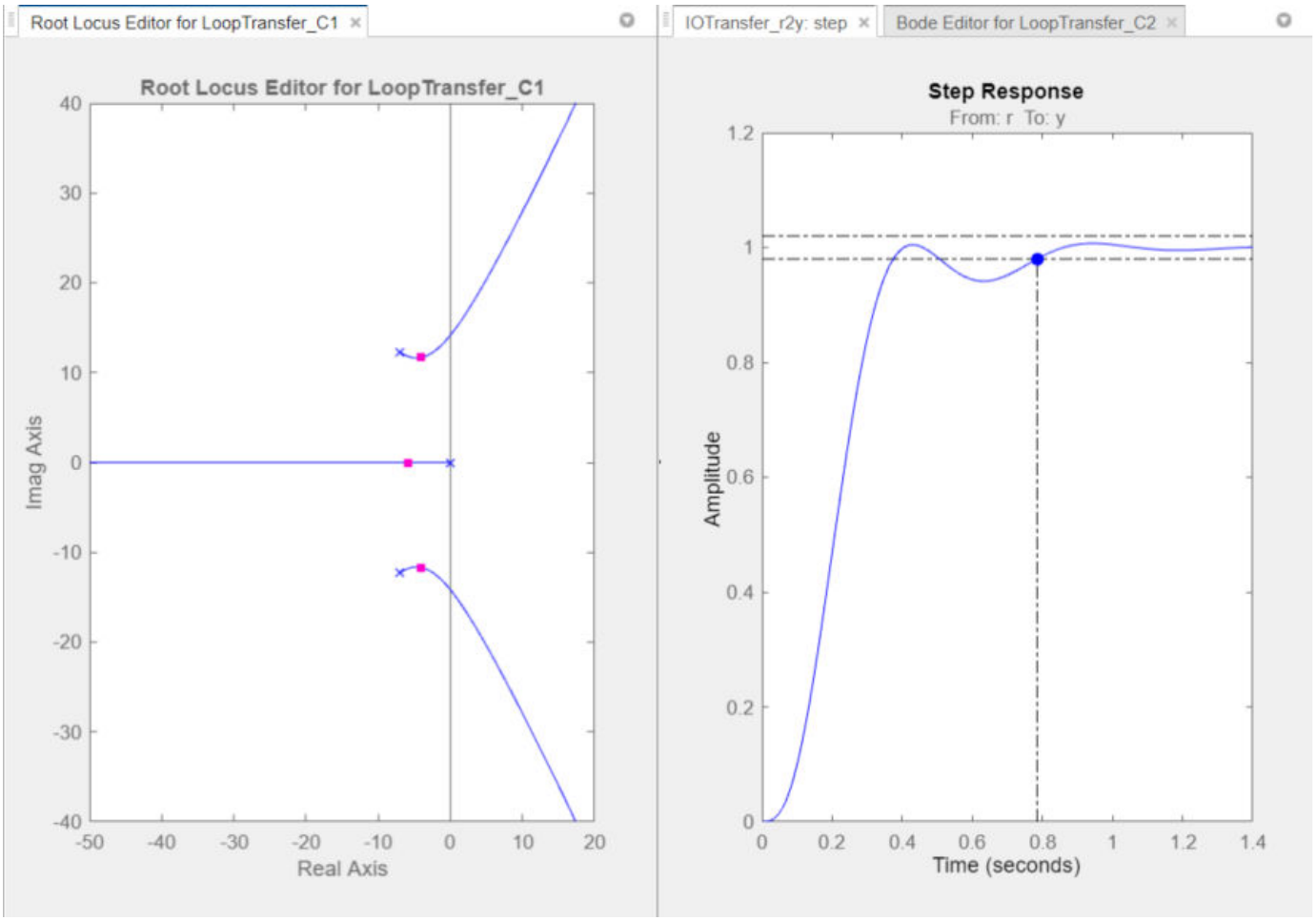
In **Control System Designer**, from **Tile All** tab, select **Left/Right**. Arrange the plots to display the **Root Locus for LoopTransfer_C1** and **IOTransfer_r2y_step** plots simultaneously.

To view the current settling time, right-click in the step response plot and select **Characteristics** > **Settling Time**.



The current closed-loop settling time is greater than 500 seconds.

In the **Root Locus Editor**, increase the gain of compensator **C1**. As the gain increases, the complex pole pair moves toward a slower time constant and the real pole moves toward a faster time constant. A gain of `600` produces a good compromise between rise time and settling time.



With a closed-loop settling time below 0.8 seconds and an inner-loop phase margin of 65 degrees, the design satisfies the design requirements.

See Also

Control System Designer

More About

- “Feedback Control Architectures” on page 12-21

Multimodel Control Design

Typically, the dynamics of a system are not known exactly and may vary. For example, system dynamics can vary because of:

- Parameter value variations caused by manufacturing tolerances — For example, the resistance value of a resistor is typically within a range about the nominal value, 5 Ω +/– 5%.
- Operating conditions — For example, aircraft dynamics change based on altitude and speed.

Any controller you design for such a system must satisfy the design requirements for all potential system dynamics.

Control Design Overview

To design a controller for a system with varying dynamics:

1. Sample the variations.
2. Create an LTI model for each sample.
3. Create an array of sampled LTI models.
4. Design a controller for a nominal representative model from the array.
5. Analyze the controller design for all models in the array.
6. If the controller design does not satisfy the requirements for all the models, specify a different nominal model and redesign the controller.

Model Arrays

In **Control System Designer**, you can specify multiple models for any plant or sensor in the current control architecture using an array of LTI models (see “Model Arrays” on page 2-76). If you specify model arrays for more than one plant or sensor, the lengths of the arrays must match.

Create Model Arrays

To create arrays for multimodel control design, you can:

- Create multiple LTI models using the `tf`, `ss`, `zpk`, or `frd` commands.

```
% Specify model parameters.
m = 3;
b = 0.5;
k = 8:1:10;
T = 0.1:.05:.2;
% Create an array of LTI models.
for ct = 1:length(k);
    G(:,:,ct) = tf(1,[m,b,k(ct)]);
end
```

- Create an array of LTI models using the `stack` command.

```
% Create individual LTI models.
G1 = tf(1, [1 1 8]);
G2 = tf(1, [1 1 9]);
G3 = tf(1, [1 1 10]);
```

```
% Combine models in an array.
G = stack(1,G1,G2,G3);
```

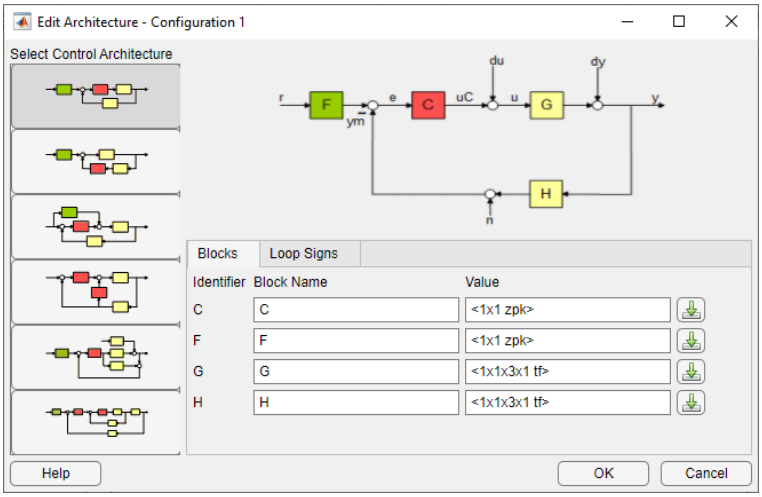
- Perform batch linearizations at multiple operating points. Then export the computed LTI models to create an array of LTI models. See the example “Reference Tracking of DC Motor with Parameter Variations” (Simulink Control Design).
- Sample an uncertain state-space (`uss`) model using `usample`.
- Compute a `uss` model from a Simulink model. Then use `usubs` or `usample` to create an array of LTI models. See “Obtain Uncertain State-Space Model from Simulink Model” (Robust Control Toolbox).
- Specify a core Simulink block to linearize to a `uss` or `ufrd` model. See “Specify Uncertain Linearization for Core or Custom Simulink Blocks” (Robust Control Toolbox).

Import Model Arrays to Control System Designer

To import models as arrays, you can pass them as input arguments when opening **Control System Designer** from the MATLAB command line. For more information, see **Control System Designer**.

You can also import model arrays into **Control System Designer** when configuring the control architecture. In the Edit Architecture dialog box:

- In the **Value** text box, specify the name of an LTI model from the MATLAB workspace.
- To import block data from the MATLAB workspace or from a MAT-file in your current working directory, click  .



Nominal Model

What Is a Nominal Model?

The nominal model is a representative model in the array of LTI models that you use to design the controller in **Control System Designer**. Use the editor and analysis plots to visualize and analyze the effect of the controller on the remaining plants in the array.

You can select any model in the array as your nominal model. For example, you can choose a model that:

- Represents the expected nominal operating point of your system.
- Is an average of the models in the array.
- Represents a worst-case plant.
- Lies closest to the stability point.

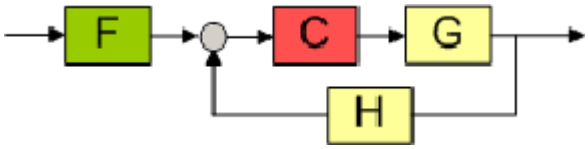
Tip You can plot and analyze the open-loop dynamics of the system on a Bode plot to determine which model to choose as nominal.

Specify Nominal Model

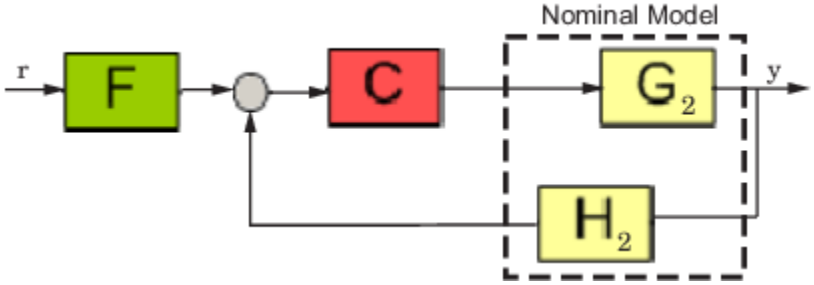
To select a nominal model from the array of LTI models, in **Control System Designer**, click **Multimodel Configuration**. Then, in the Multimodel Configuration dialog box, select a **Nominal model index**. The default index is 1.

For each plant or sensor that is defined as a model array, the app selects the model at the specified index as the nominal model. Otherwise, the app uses scalar expansion to apply the single LTI model for all model indices.

For example, for the following control architecture:



if G and H are both three-element arrays and the nominal model index is 2, the software uses the second element in both the arrays to compute the nominal model:

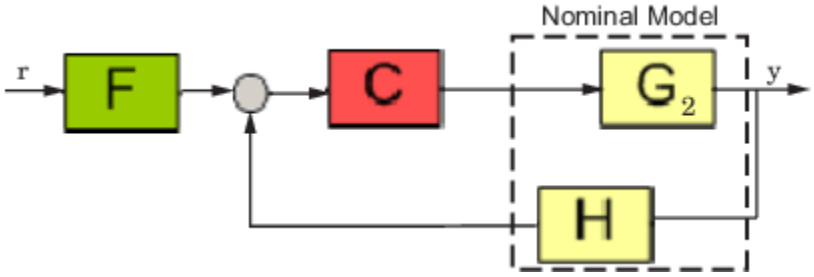


The nominal response from r to y is:

$$T = \frac{CG_2}{1 + CG_2H_2}$$

The app also computes and plots the responses showing the effect of C on the remaining pairs of plant and sensor models — G_1H_1 and G_3H_3.

If only G is an array of LTI models, and the specified nominal model is 2, then the control architecture for nominal response is:



In this case, the nominal response from r to y is:

$$T = \frac{CG_2}{1 + CG_2H}$$

The app also computes and plots the responses showing the effect of C on the remaining pairs of plant and sensor model — G_1H and G_3H.

Frequency Grid

The frequency response of a system is computed at a series of frequency values, called a *frequency grid*. By default, **Control System Designer** computes a logarithmically equally spaced grid based on the dynamic range of each model in the array.

Specify a custom frequency grid when:

- The automatic grid has more points than you require. To improve computational efficiency, specify a less dense grid spacing.
- The automatic grid is not sufficiently dense within a particular frequency range. For example, if the response does not capture the resonant peak dynamics of an underdamped system, specify a more dense grid around the corner frequency.
- You are only interested in the response within specific frequency ranges. To improve computational efficiency, specify a grid that covers only the frequency ranges of interest.

For more information on specifying logarithmically spaced vectors, see `logspace`.

Note Modifying the frequency grid does not affect the frequency response computation for the nominal model. The app always uses the **Auto select** option to compute the nominal model frequency response.

Design Controller for Multiple Plant Models

This example shows how to design a compensator for a set of plant models using **Control System Designer**.

1 Create Array of Plant Models

Create an array of LTI plant models using the `stack` command.

```
% Create an array of LTI models to model plant (G) variations.
G1 = tf(1,[1 1 8]);
G2 = tf(1,[1 1 9]);
```

```
G3 = tf(1,[1 1 10]);
G = stack(1,G1,G2,G3);
```

2 Create Array of Sensor Models

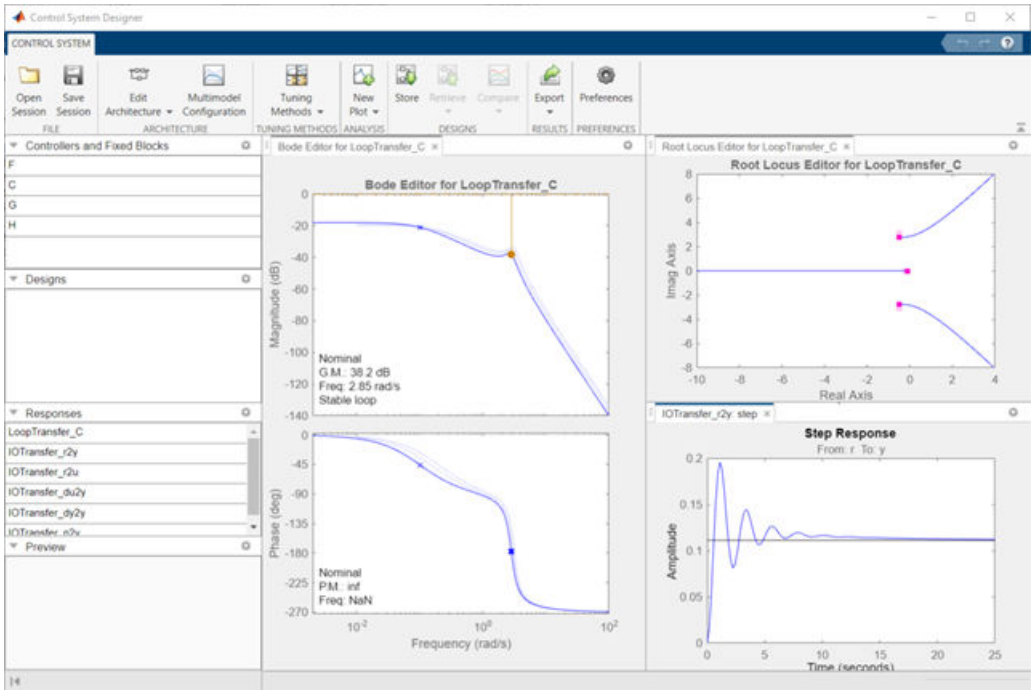
Similarly, create an array of sensor models.

```
H1 = tf(1,[1/0.1,1]);
H2 = tf(1,[1/0.15,1]);
H3 = tf(1,[1/0.2,1]);
H = stack(1,H1,H2,H3);
```

3 Open Control System Designer

Open **Control System Designer**, and import the plant and sensor model arrays.

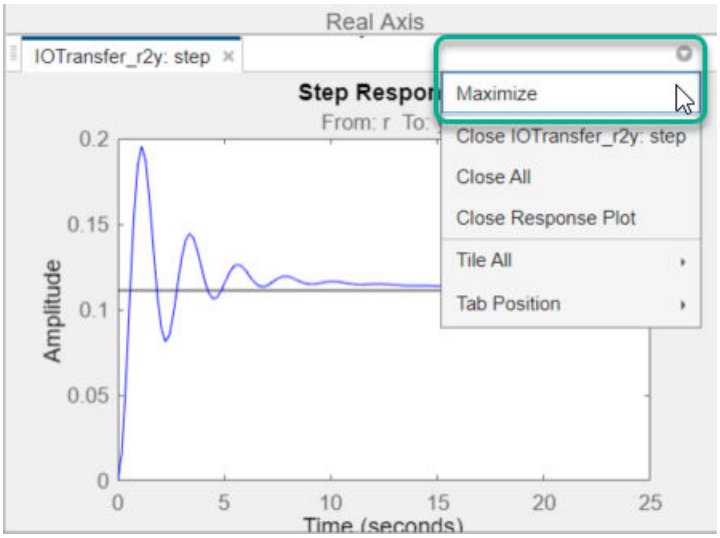
```
controlSystemDesigner(G,1,H)
```



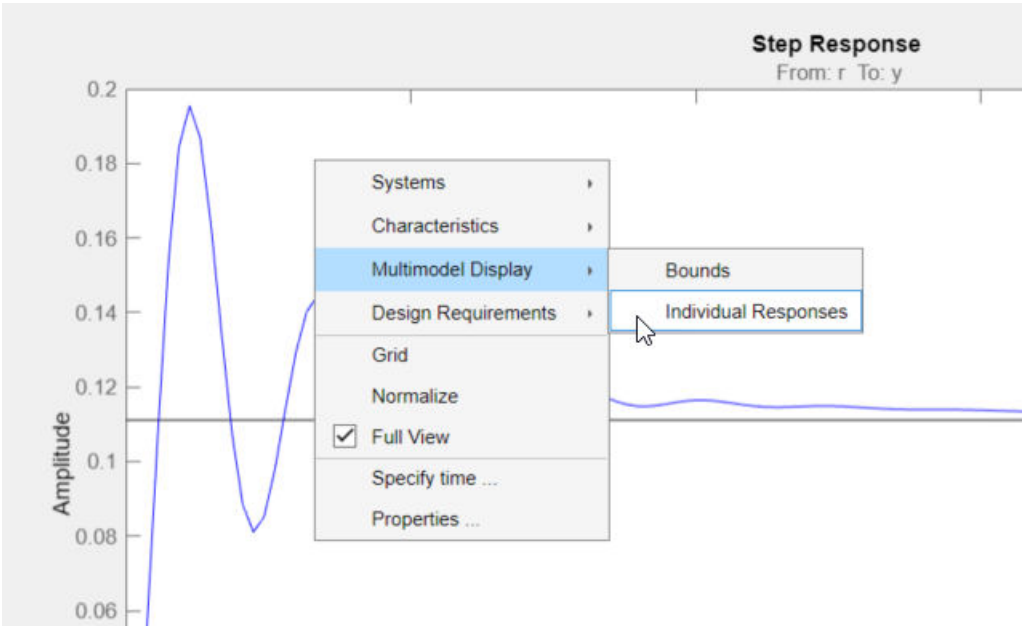
The app opens and imports the plant and sensor model arrays.

4 Configure Analysis Plot

To view the closed-loop step response in a larger plot, in **Control System Designer**, click on the small dropdown arrow on the **IOTransfer_r2y: step** plot and then select `Maximize`.

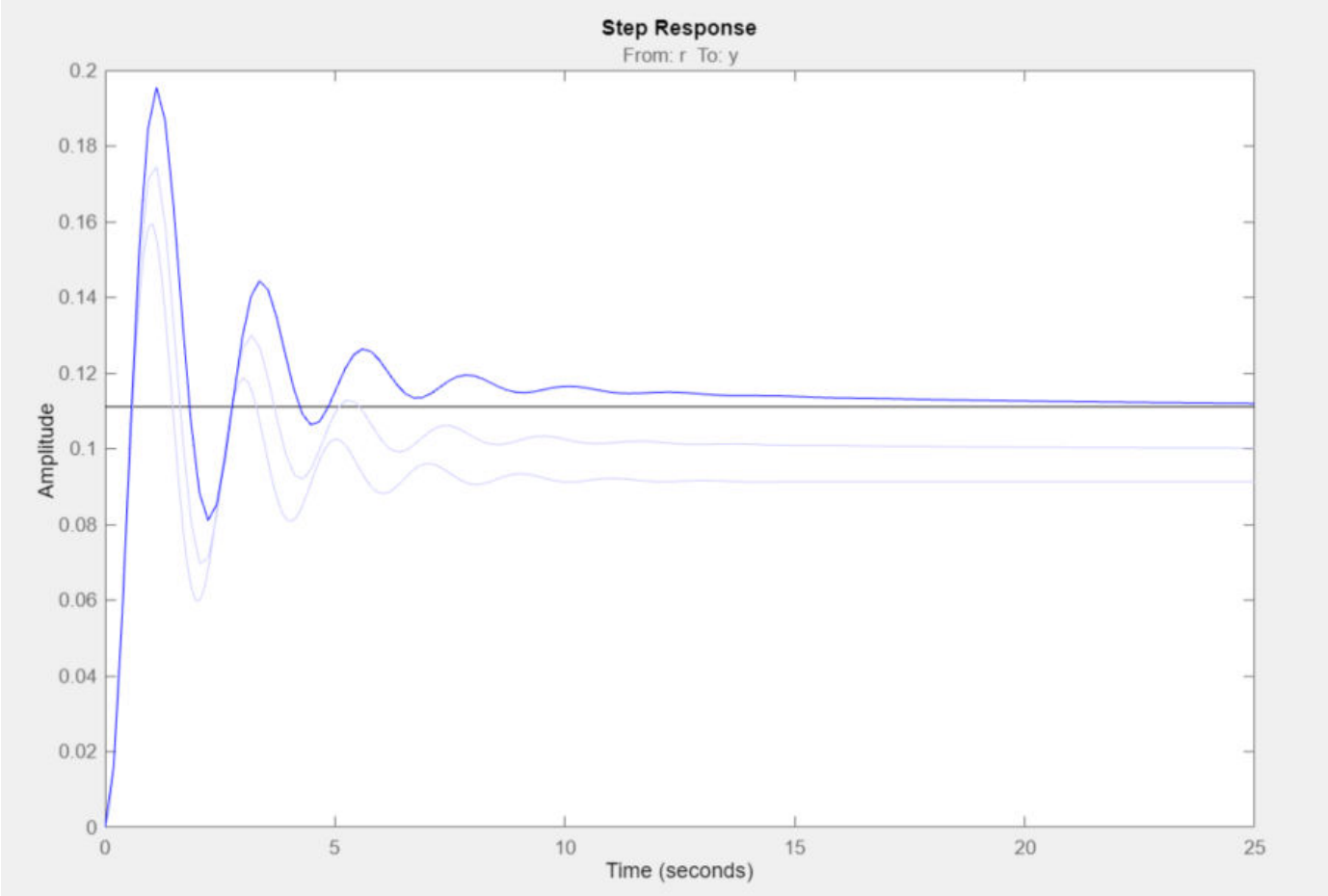


By default the step response shows only the nominal response. To display the individual responses for the other model indices, right-click the plot area, and select **Multimodel Display > Individual Responses**.



Note To view an envelope of all model responses, right-click the plot area, and select **Multimodel Display > Bounds**

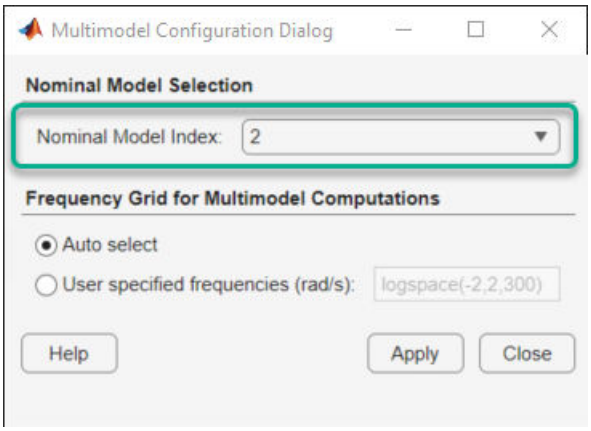
The plot updates to display the responses for the other models.



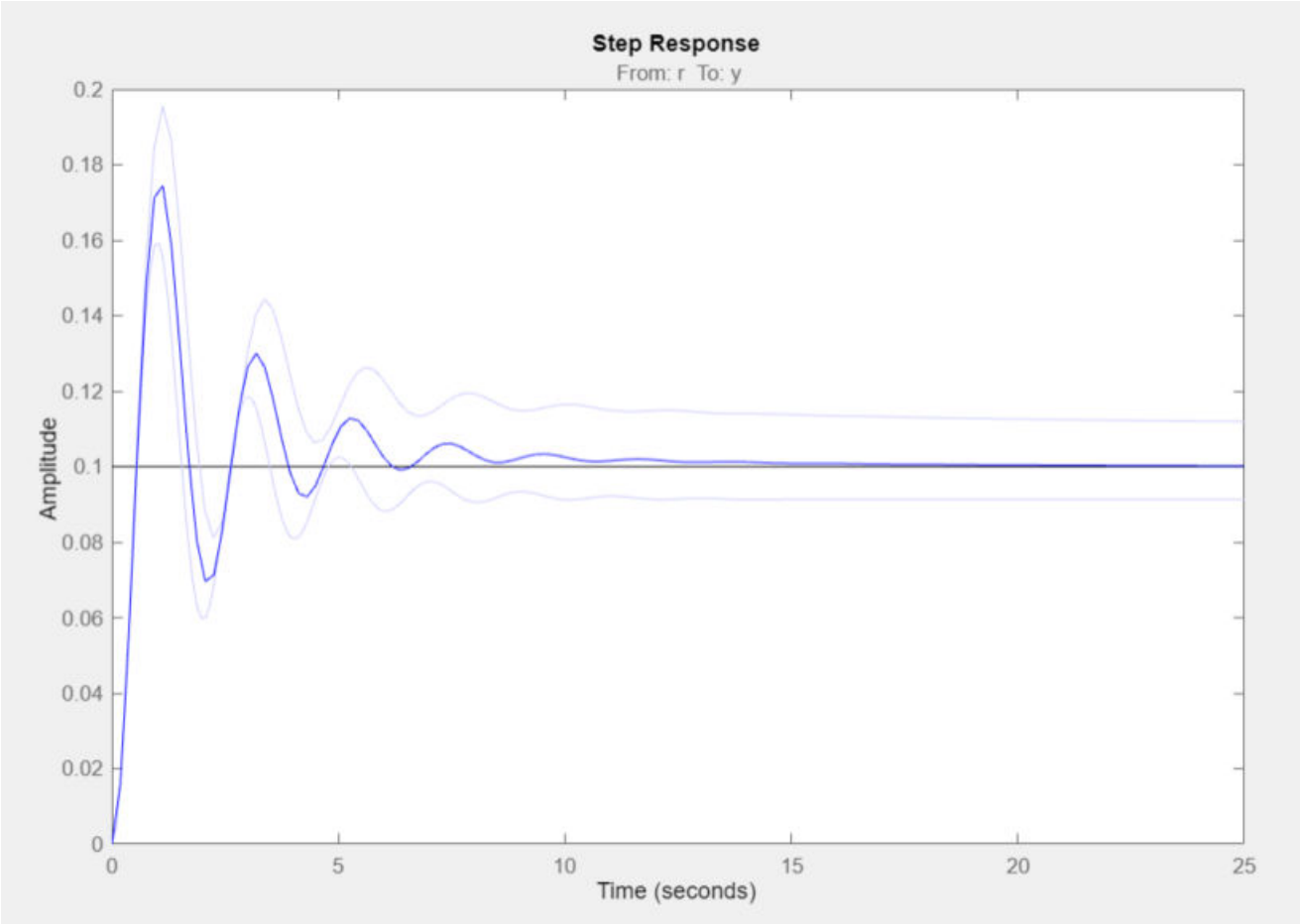
5 Select Nominal Model

On the **Control System** tab, click **Multimodel Configuration**.

In the Multimodel Configuration dialog box, specify a **Nominal Model Index** of 2.



Click **Apply**, then **Close**.

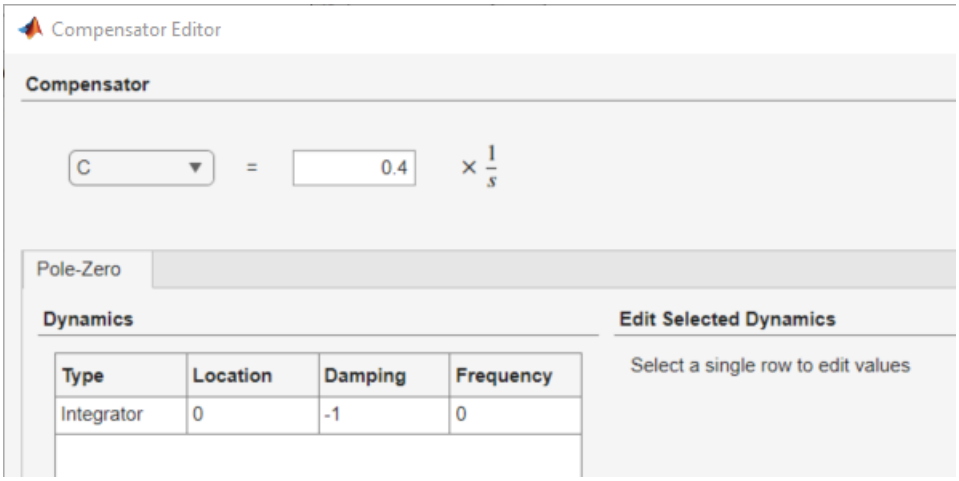


The selected nominal model corresponds to the average system response.

6 Design Compensator

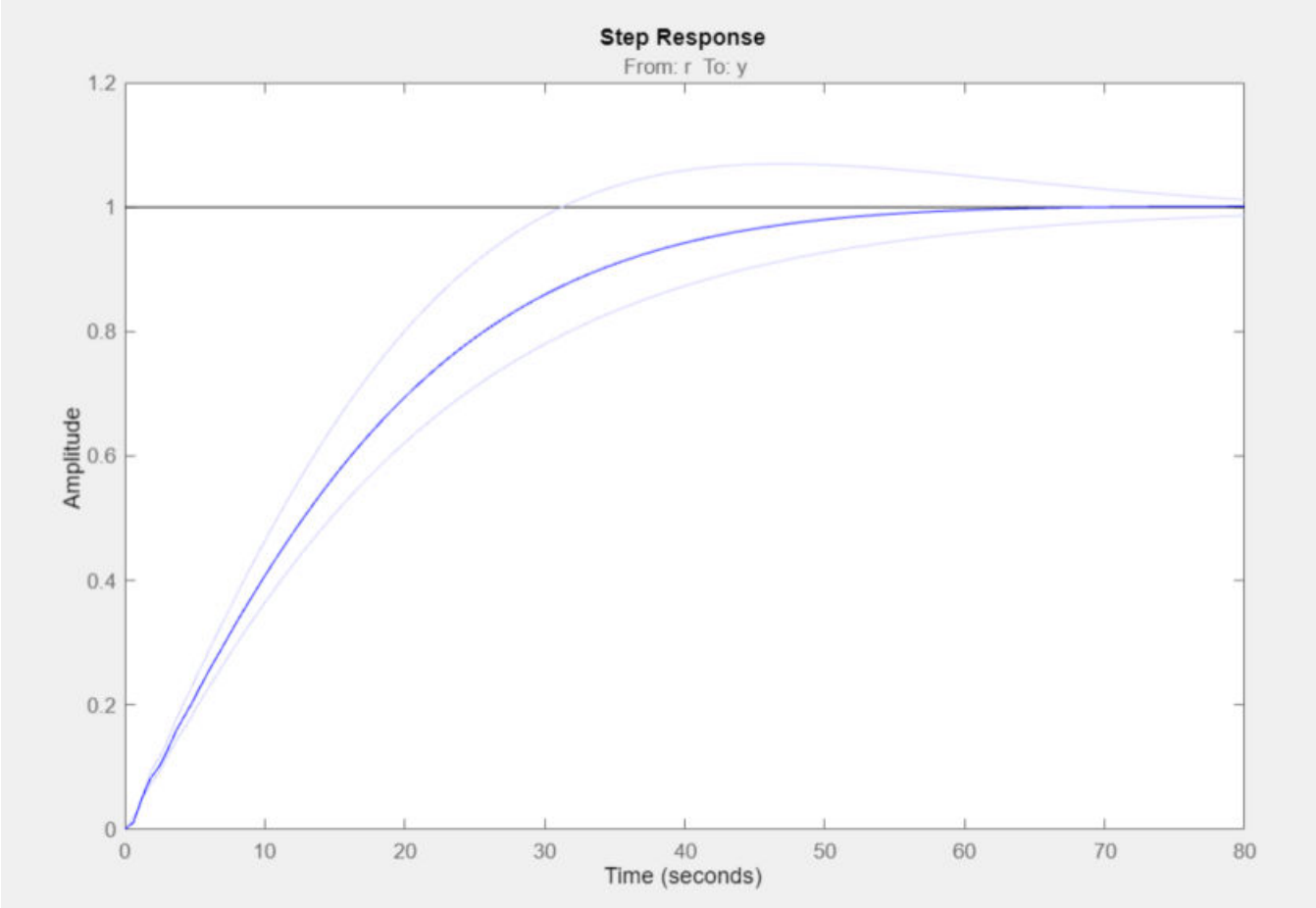
To design a compensator using the nominal model, you can use any of the supported “Control System Designer Tuning Methods” on page 12-5.

For this example, use the Compensator Editor to manually specify the compensator dynamics. Add an integrator to the compensator and set the compensator gain to `0.4`. For more information, see “Edit Compensator Dynamics” on page 12-78.



7 Analyze Results

The tuned controller produces a step response with minimal overshoot for the nominal models and a worst-case overshoot less than 10%.



See Also

Control System Designer

Related Examples

- “Model Arrays” on page 2-76
- “Control System Designer Tuning Methods” on page 12-5

Bode Diagram Design

Bode diagram design is an interactive graphical method of modifying a compensator to achieve a specific open-loop response (loop shaping). To interactively shape the open-loop response using **Control System Designer**, use the **Bode Editor**. In the editor, you can adjust the open-loop bandwidth and design to gain and phase margin specifications.

To adjust the loop shape, you can add poles and zeros to your compensator and adjust their values directly in the **Bode Editor**, or you can use the Compensator Editor. For more information, see “Edit Compensator Dynamics” on page 12-78.

For information on all of the tuning methods available in **Control System Designer**, see “Control System Designer Tuning Methods” on page 12-5.

Tune Compensator For DC Motor Using Bode Diagram Graphical Tuning

This example shows how to design a compensator for a DC motor using Bode diagram graphical tuning techniques.

Plant Model and Requirements

The transfer function of the DC motor plant, as described in “SISO Example: The DC Motor”, is:

$$G = \frac{1.5}{s^2 + 14s + 40.02}$$

For this example, the design requirements are:

- Rise time of less than 0.5 seconds
- Steady-state error of less than 5%
- Overshoot of less than 10%
- Gain margin greater than 20 dB
- Phase margin greater than 40 degrees

Open Control System Designer

At the MATLAB command line, create a transfer function model of the plant, and open **Control System Designer** in the Bode Editor configuration.

```
G = tf(1.5,[1 14 40.02]);
controlSystemDesigner('bode',G);
```

The app opens and imports `G` as the plant model for the default control architecture, **Configuration 1**.

In the app, the following response plots open:

- Open-loop **Bode Editor** for the `LoopTransfer_C` response. This response is the open-loop transfer function *GC*, where *C* is the compensator and *G* is the plant.
- **Step Response** for the `IOTransfer_r2y` response. This response is the input-output transfer function for the overall closed-loop system.

Tip To open the open-loop **Bode Editor** when **Control System Designer** is already open, on the **Control System** tab, in the **Tuning Methods** drop-down list, select **Bode Editor**. In the Select Response to Edit dialog box, select an existing response to plot, or create a `New Open-Loop Response`.

To view the open-loop frequency response and closed-loop step response simultaneously, click and drag the plots to the desired location.

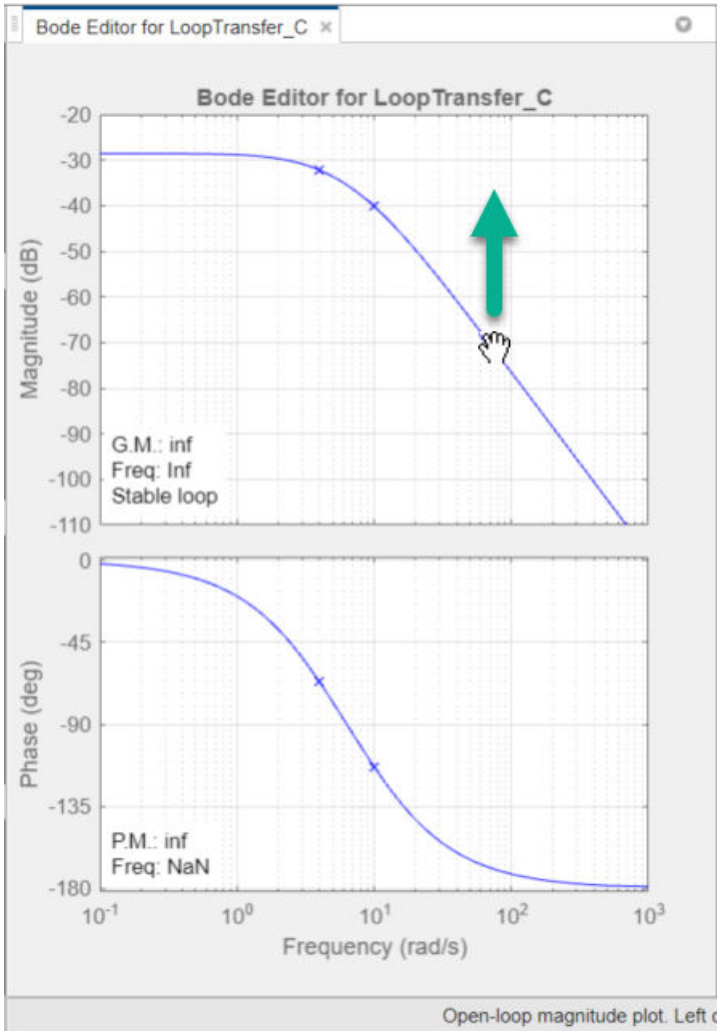
The app displays the **Bode Editor** and **Step Response** plots side-by-side.

Adjust Bandwidth

Since the design requires a rise time less than 0.5 seconds, set the open-loop DC crossover frequency to about 3 rad/s. To a first-order approximation, this crossover frequency corresponds to a time constant of 0.33 seconds.

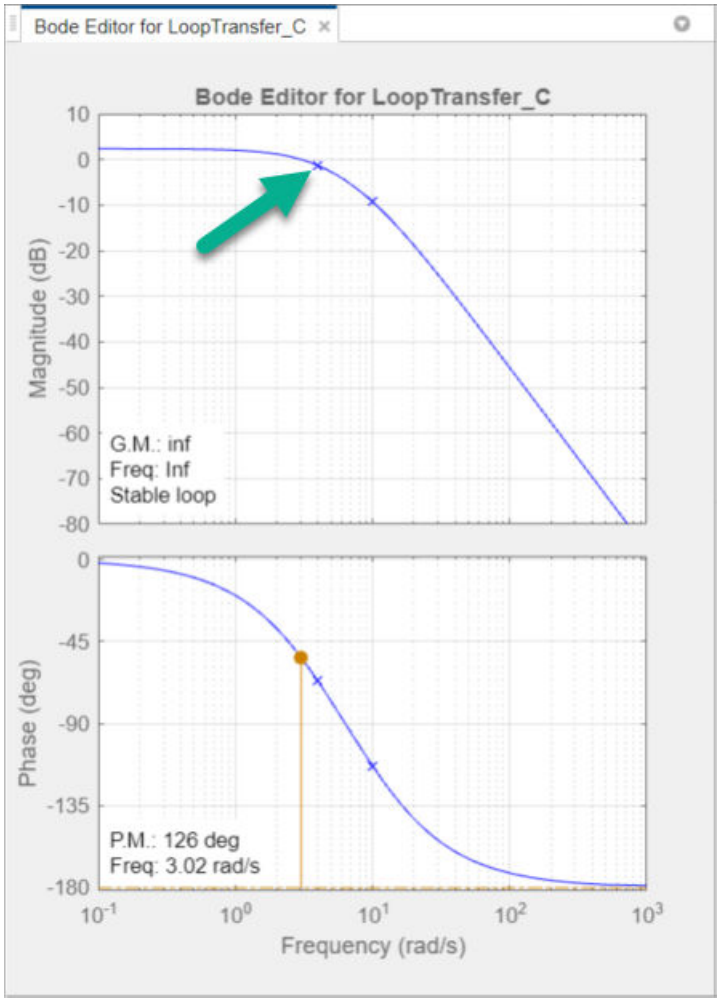
To make the crossover easier to see, turn on the plot grid. Right-click the **Bode Editor** plot area, and select **Grid**. The app adds a grid to the Bode response plots.

To adjust the crossover frequency increase the compensator gain. In the **Bode Editor** plot, in the **Magnitude** response plot, drag the response upward. Doing so increases the gain of the compensator.



As you drag the magnitude plot, the app computes the compensator gain and updates the response plots.

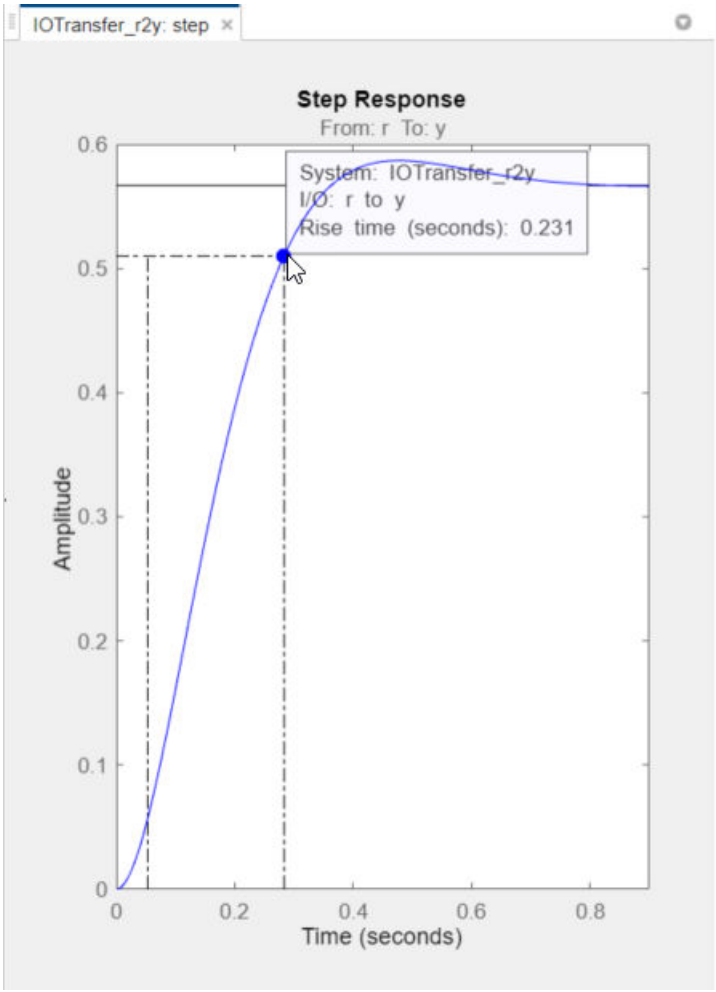
Drag the magnitude response upward until the crossover frequency is about 3 rad/s.



View Step Response Characteristics

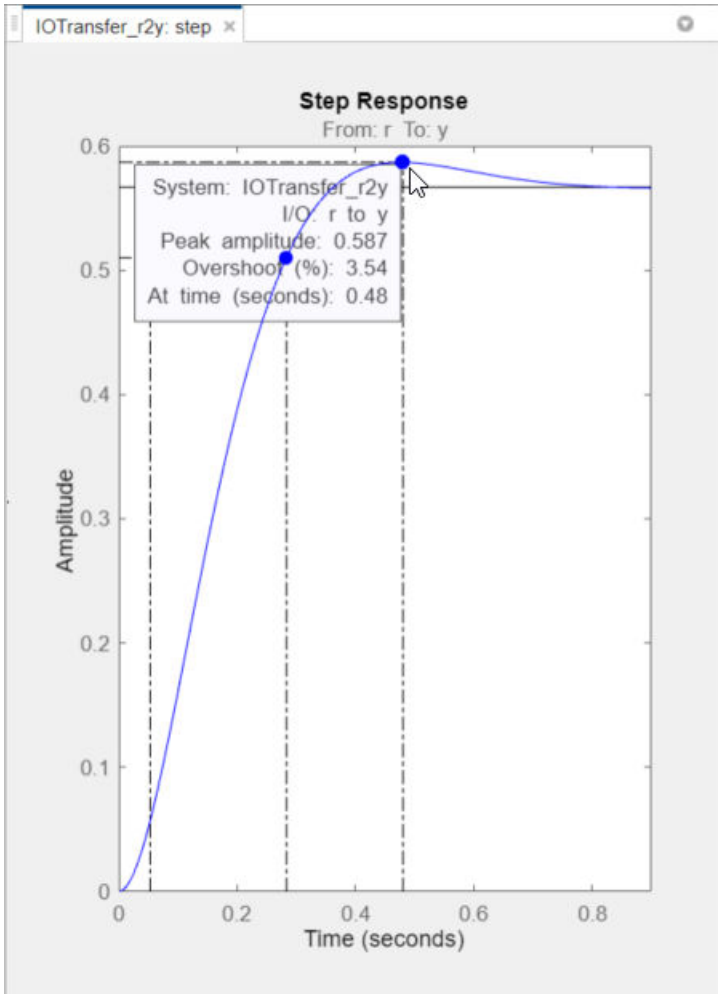
To add the rise time to the **Step Response** plot, right-click the plot area, and select **Characteristics > Rise Time**.

To view the rise time, move the cursor over the rise time indicator.



The rise time is around 0.23 seconds, which satisfies the design requirements.

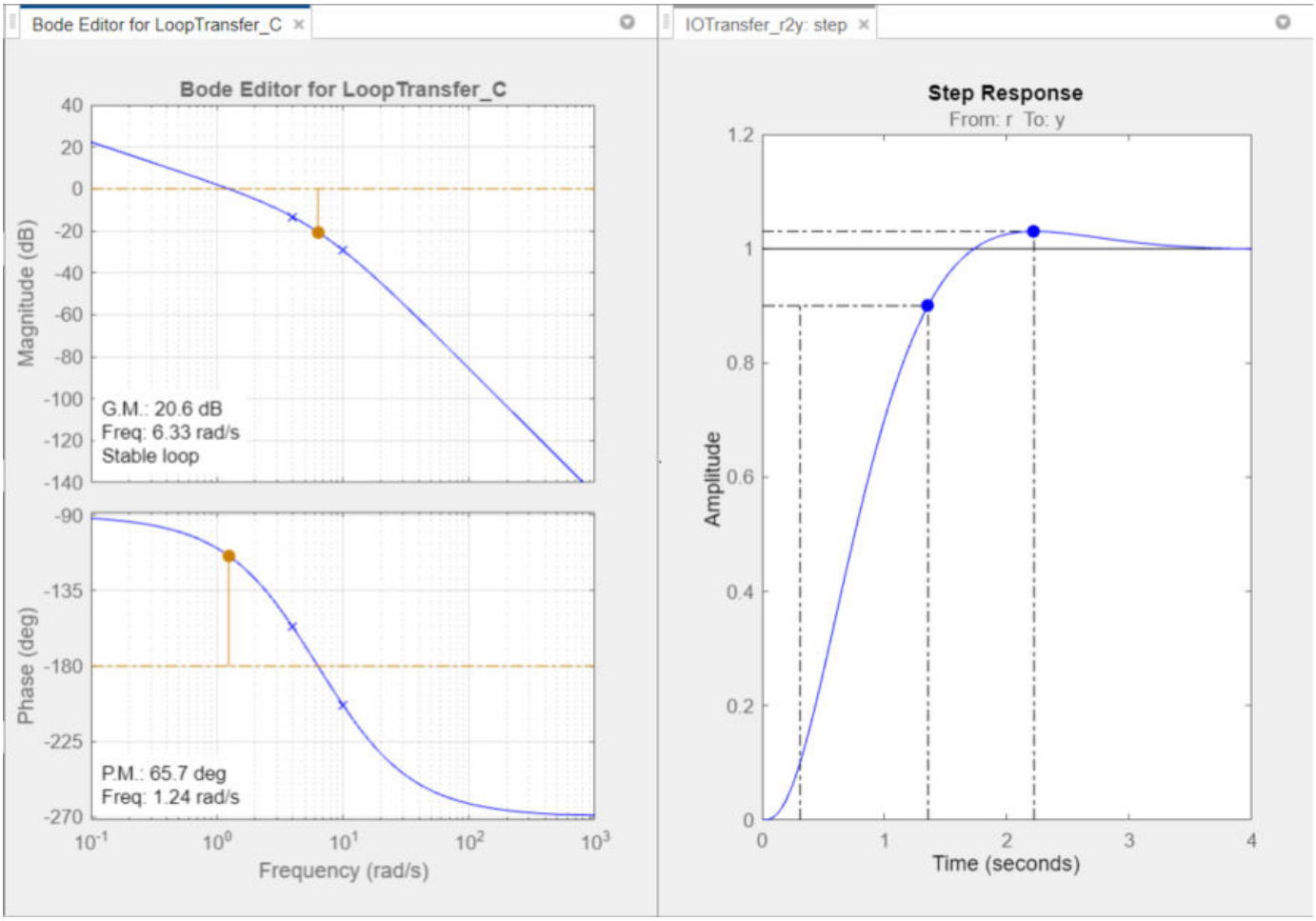
Similarly, to add the peak response to the **Step Response** plot, right-click the plot area, and select **Characteristics > Peak Response**.



The peak overshoot is around 3.5%.

Add Integrator to Compensator

To meet the 5% steady-state error requirement, eliminate steady-state error from the closed-loop step response by adding an integrator to your compensator. In the **Bode Editor** right-click in the plot area, and select **Add Pole or Zero > Integrator**.



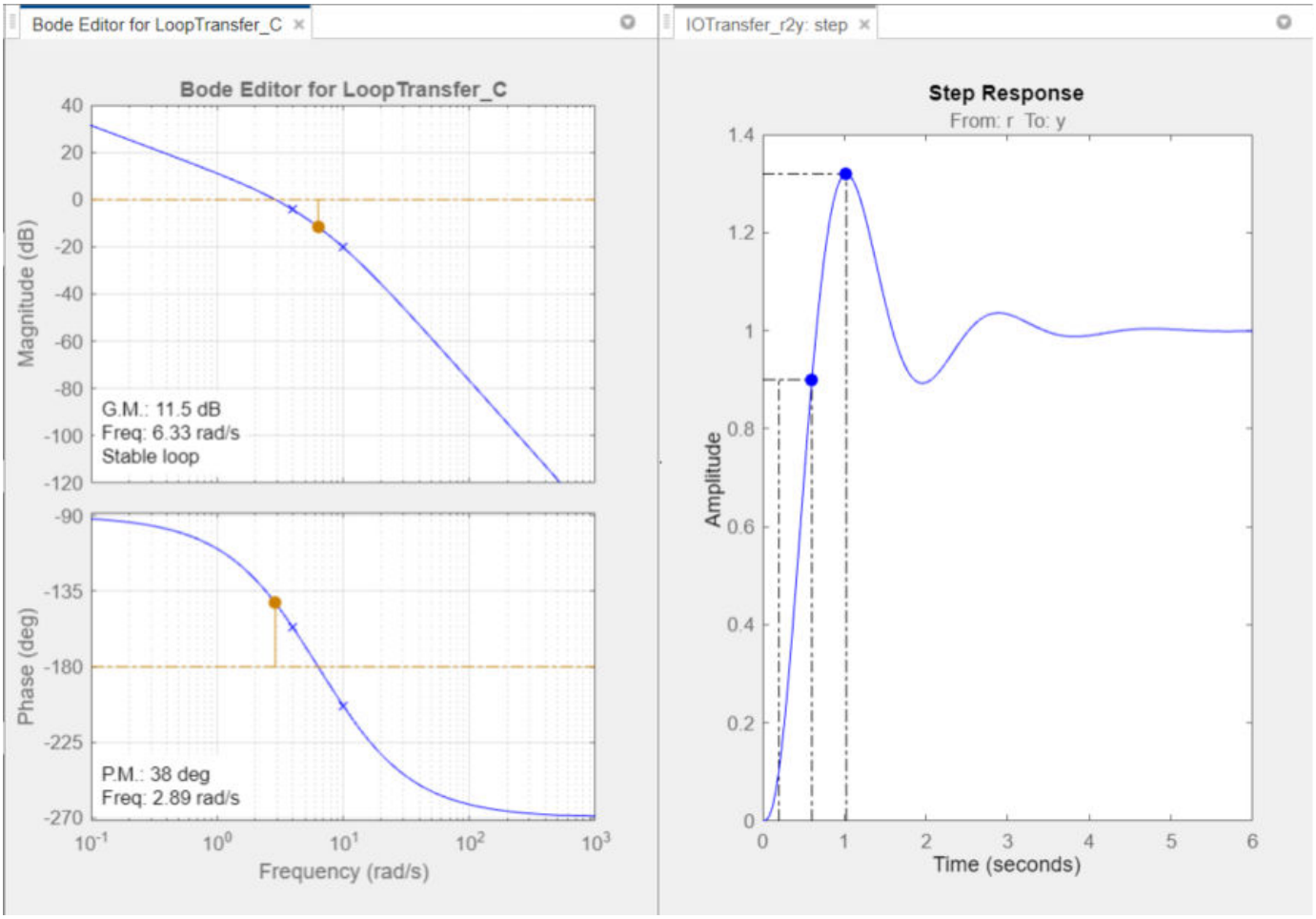
Adding an integrator produces zero steady-state error. However, changing the compensator dynamics also changes the crossover frequency, increasing the rise time. To reduce the rise time, increase the crossover frequency to around 3 rad/s.

Adjust Compensator Gain

To return the crossover frequency to around 3 rad/s, increase the compensator gain further. Right-click the **Bode Editor** plot area, and select **Edit Compensator**.

In the Compensator Editor dialog box, in the **Compensator** section, specify a gain of 99, and press **Enter**.

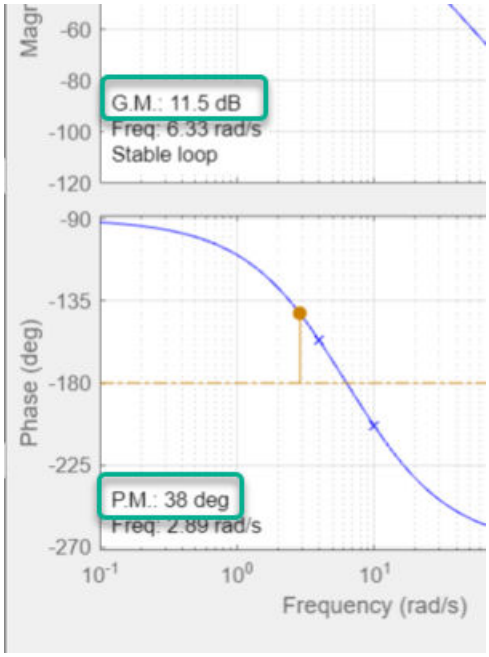
The response plots update automatically.



The rise time is around 0.4 seconds, which satisfies the design requirements. However, the peak overshoot is around 32%. A compensator consisting of a gain and an integrator is not sufficient to meet the design requirements. Therefore, the compensator requires additional dynamics.

Add Lead Network to Compensator

In the **Bode Editor**, review the gain margin and phase margin for the current compensator design. The design requires a gain margin greater than 20 dB and phase margin greater than 40 degrees. The current design does not meet either of these requirements.



To increase the stability margins, add a lead network to the compensator.

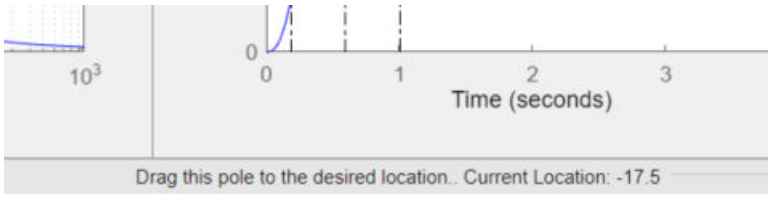
In the **Bode Editor**, right-click and select **Add Pole or Zero > Lead**.

To specify the location of the lead network pole, click on the magnitude response. The app adds a real pole (red X) and real zero (red 0) to the compensator and to the **Bode Editor** plot.

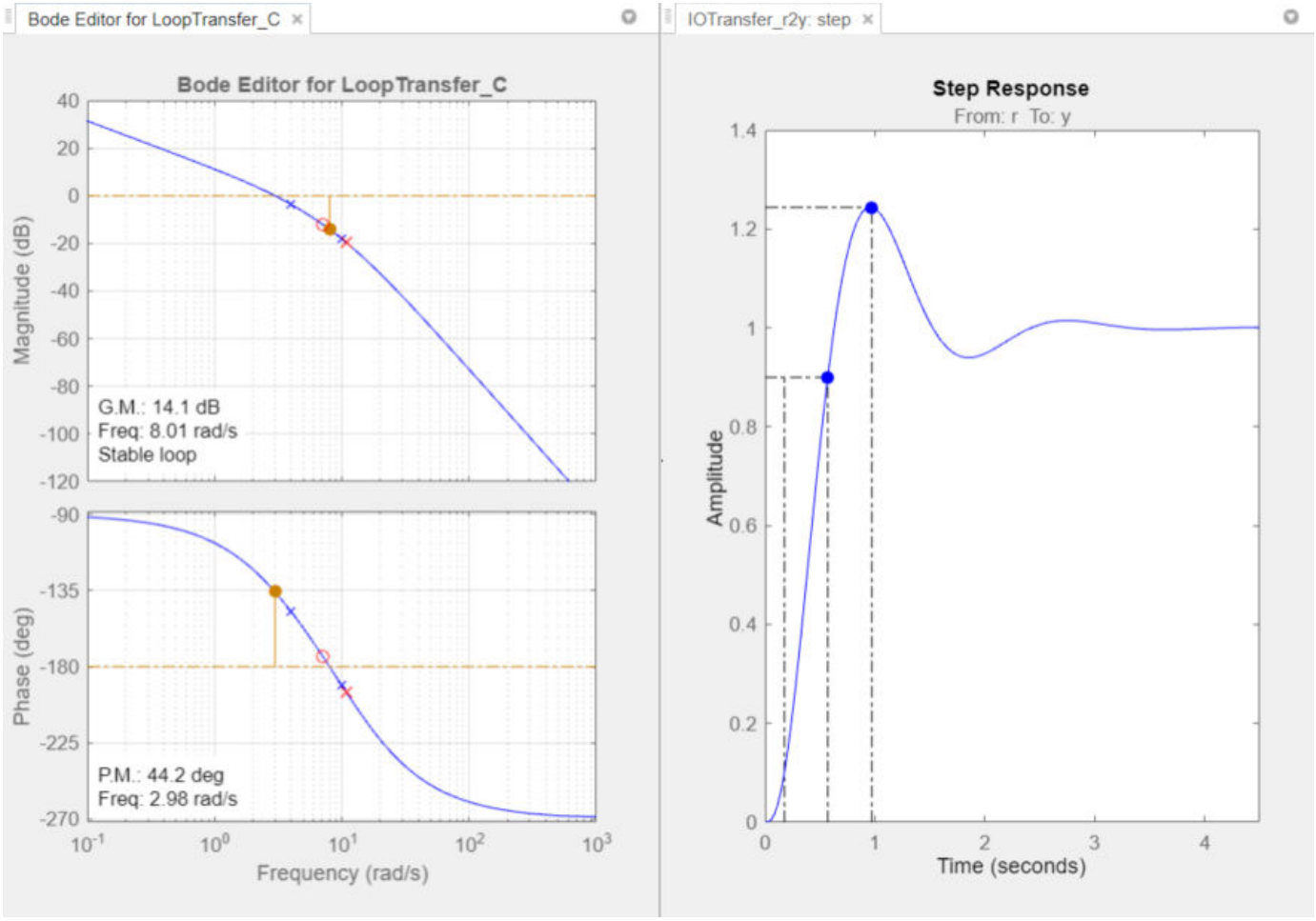
In the **Bode Editor**, drag the pole and zero to change their locations. As you drag them, the app updates the pole/zero values and updates the response plots.

To decrease the magnitude of a pole or zero, drag it towards the left. Since the pole and zero are on the negative real axis, dragging them to the left moves them closer to the origin in the complex plane.

Tip As you drag a pole or zero, the app displays the new value in the status bar, on the right side.



As an initial estimate, drag the zero to a location around `-7` and the pole to a location around `-11`.



The phase margin meets the design requirements; however, the gain margin is still too low.

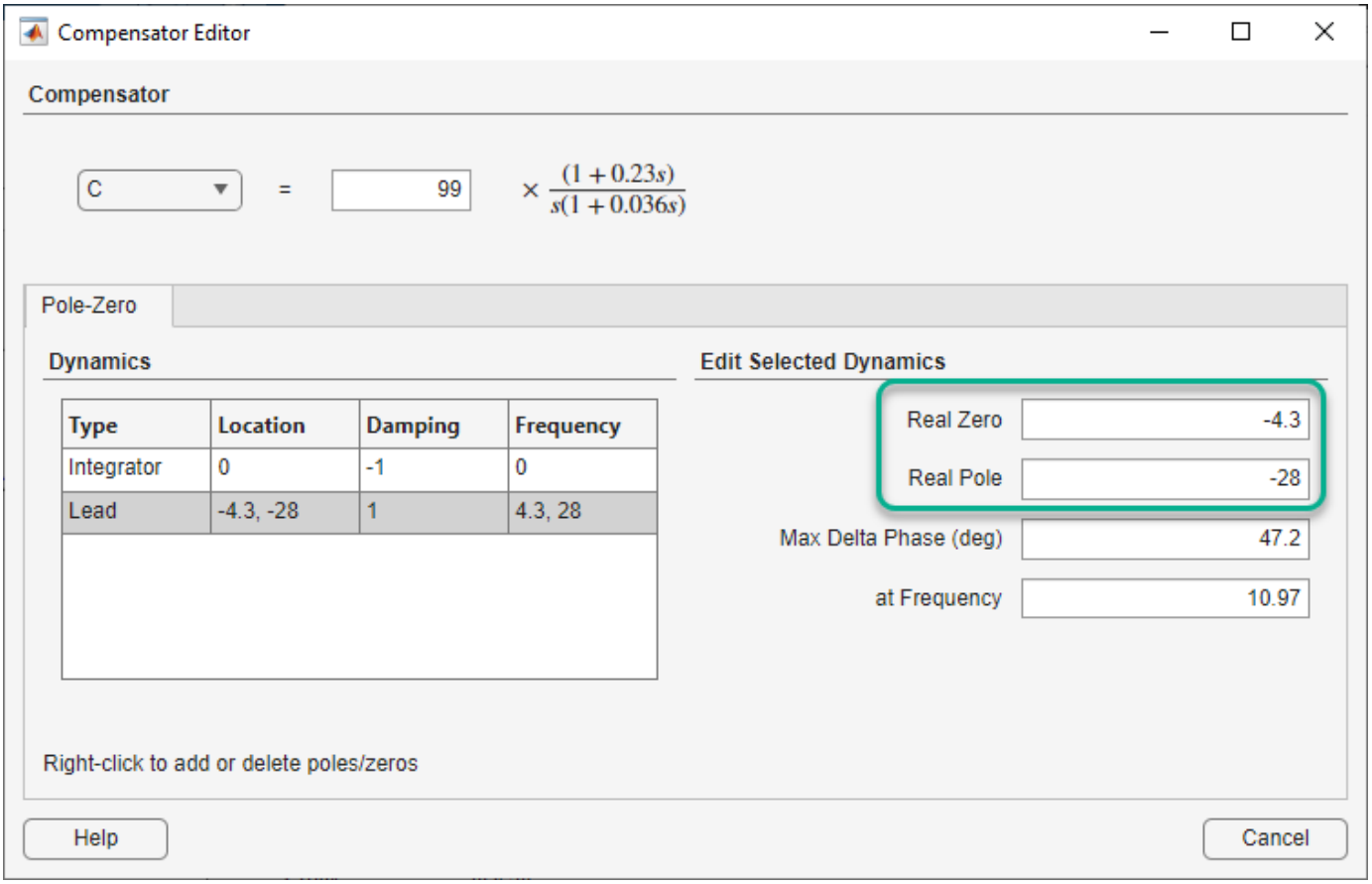
Edit Lead Network Pole and Zero

To improve the controller performance, tune the lead network parameters.

In the Compensator Editor dialog box, in the **Dynamics** section, click the **Lead** row.

In the **Edit Selected Dynamics** section, in the **Real Zero** text box, specify a location of `-4.3`, and press **Enter**. This value is near the slowest (left-most) pole of the DC motor plant.

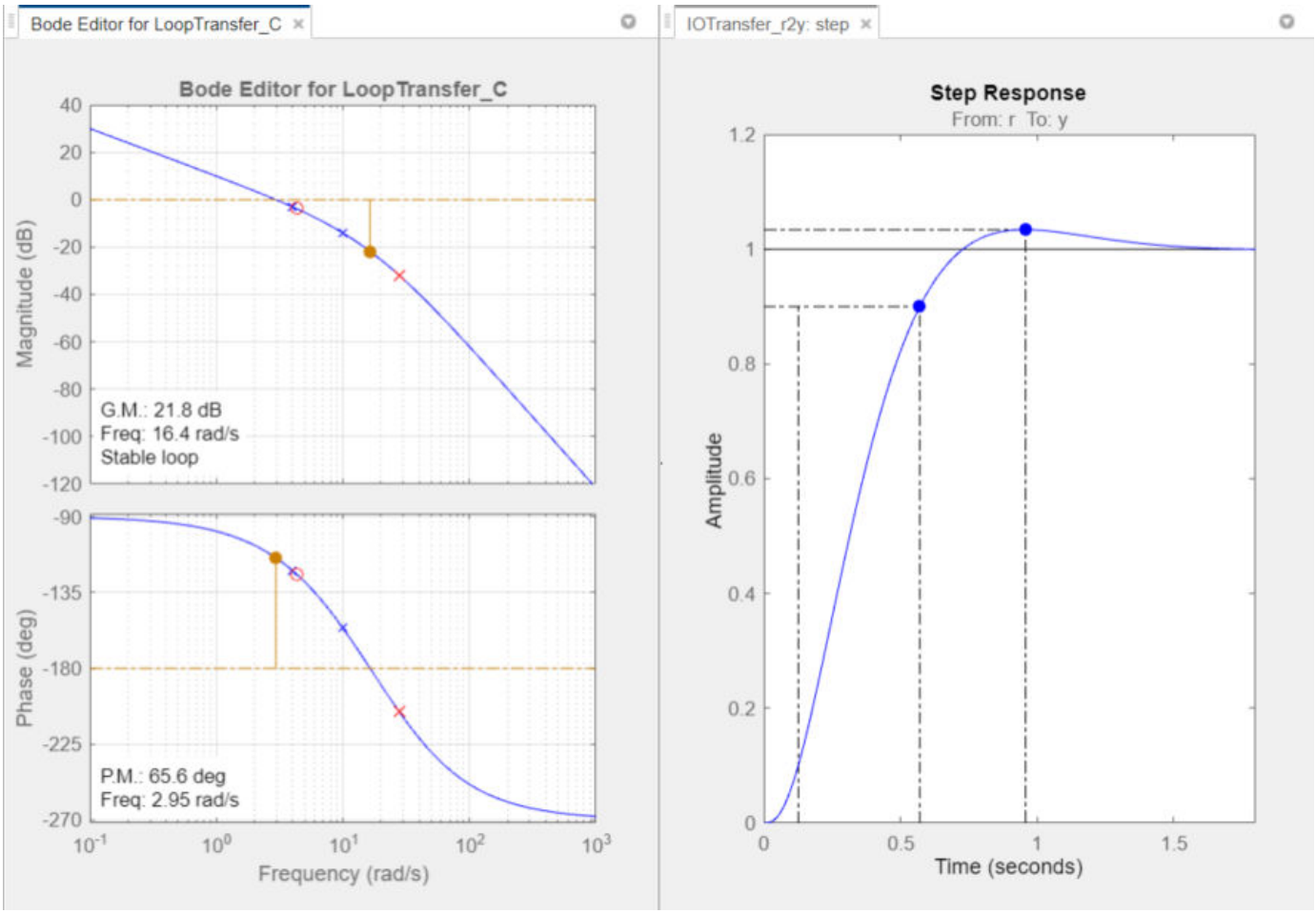
In the **Real Pole** text box, specify a value of `-28`, and press **Enter**.



When you modify a lead network parameters, the **Compensator** and response plots update automatically.

In the app, in the **Bode Editor**, the gain margin of `20.5` just meets the design requirement.

To add robustness to the system, in the Compensator Editor dialog box, decrease the compensator gain to `84.5`, and press **Enter**. The gain margin increases to `21.8`, and the response plots update.



In **Control System Designer**, in the response plots, compare the system performance to the design requirements. The system performance characteristics are:

- Rise time is 0.445 seconds.
- Steady-state error is zero.
- Overshoot is 3.39%.
- Gain margin is 21.8 dB.
- Phase margin is 65.6 degrees.

The system response meets all of the design requirements.

See Also

Control System Designer | `bodeplot`

More About

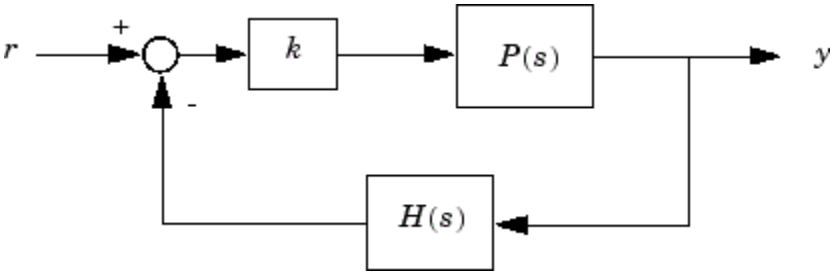
- “Edit Compensator Dynamics” on page 12-78
- “Control System Designer Tuning Methods” on page 12-5
- “Root Locus Design” on page 12-55

- “Nichols Plot Design” on page 12-67

Root Locus Design

Root locus design is a common control system design technique in which you edit the compensator gain, poles, and zeros in the root locus diagram.

As the open-loop gain, k, of a control system varies over a continuous range of values, the root locus diagram shows the trajectories of the closed-loop poles of the feedback system. For example, in the following tracking system:



$P(s)$ is the plant, $H(s)$ is the sensor dynamics, and k is an adjustable scalar gain The closed-loop poles are the roots of

$$q(s) = 1 + kP(s)H(s)$$

The root locus technique consists of plotting the closed-loop pole trajectories in the complex plane as k varies. You can use this plot to identify the gain value associated with a desired set of closed-loop poles.

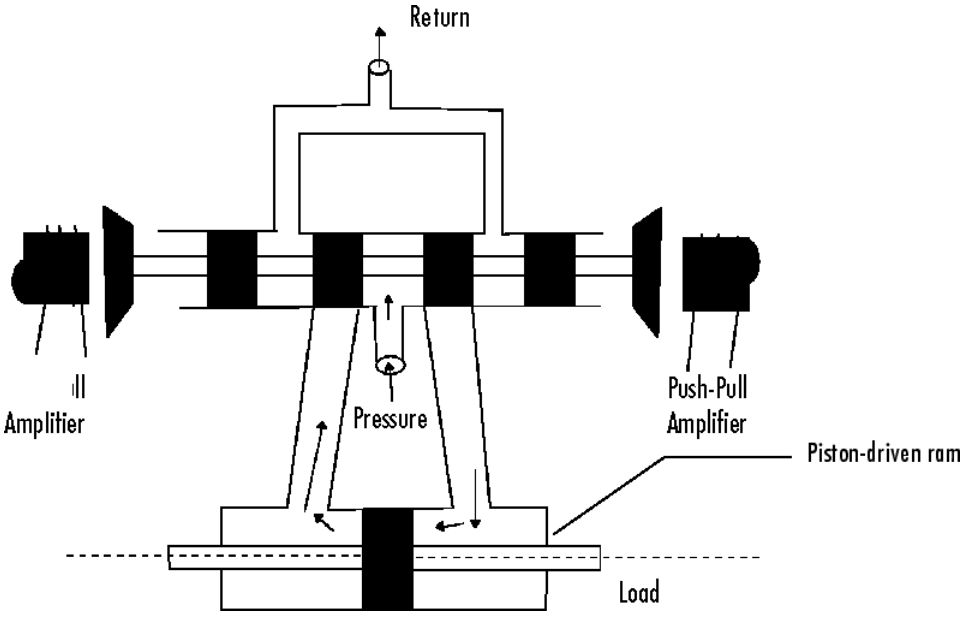
Tune Electrohydraulic Servomechanism Using Root Locus Graphical Tuning

This example shows how to design a compensator for an electrohydraulic servomechanism using root locus graphical tuning techniques.

Plant Model

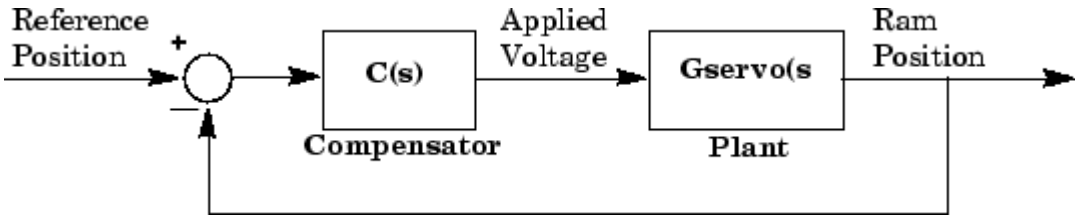
A simple version of an electrohydraulic servomechanism model consists of

- A push-pull amplifier (a pair of electromagnets)
- A sliding spool in a vessel of high-pressure hydraulic fluid
- Valve openings in the vessel to allow for fluid to flow
- A central chamber with a piston-driven ram to deliver force to a load
- A symmetrical fluid return vessel



The force on the spool is proportional to the current in the electromagnet coil. As the spool moves, the valve opens, allowing the high-pressure hydraulic fluid to flow through the chamber. The moving fluid forces the piston to move in the opposite direction of the spool. For more information on this model, including the derivation of a linearized model, see [1].

You can use the input voltage to the electromagnet to control the ram position. When measurements of the ram position are available, you can use feedback for the ram position control, as shown in the following, where *Gservo* represents the servomechanism:



Design Requirements

For this example, tune the compensator, $C(s)$ to meet the following closed-loop step response requirements:

- The 2% settling time is less than 0.05 seconds.
- The maximum overshoot is less than 5%.

Open Control System Designer

At the MATLAB command line, load a linearized model of the servomechanism, and open **Control System Designer** in the root locus editor configuration.

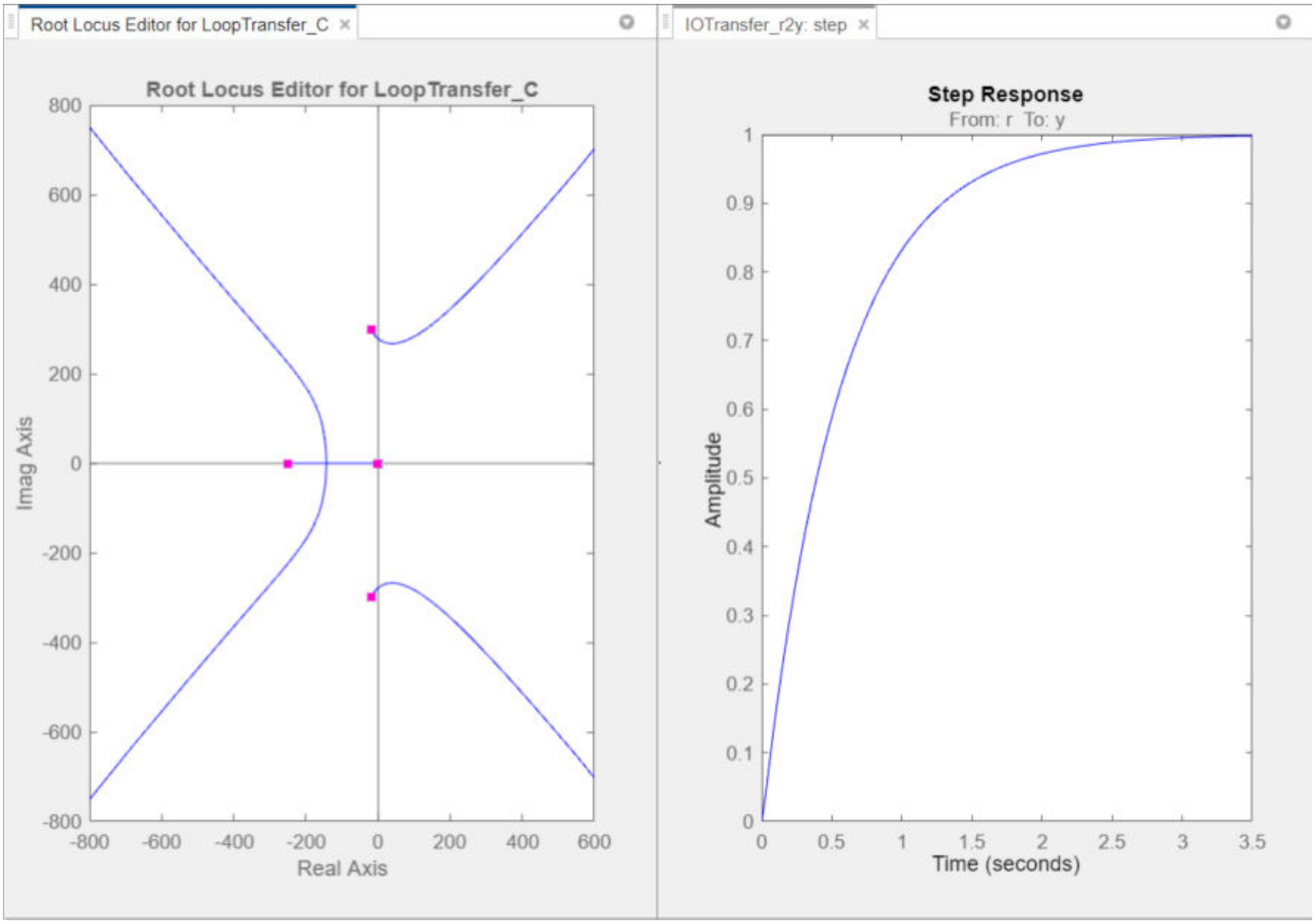
```
load ltiexamples Gservo
controlSystemDesigner('rlocus',Gservo);
```

The app opens and imports `Gservo` as the plant model for the default control architecture, **Configuration 1**.

In **Control System Designer**, a **Root Locus Editor** plot and input-output **Step Response** open.

To view the open-loop frequency response and closed-loop step response simultaneously, click and drag the plots to the desired location.

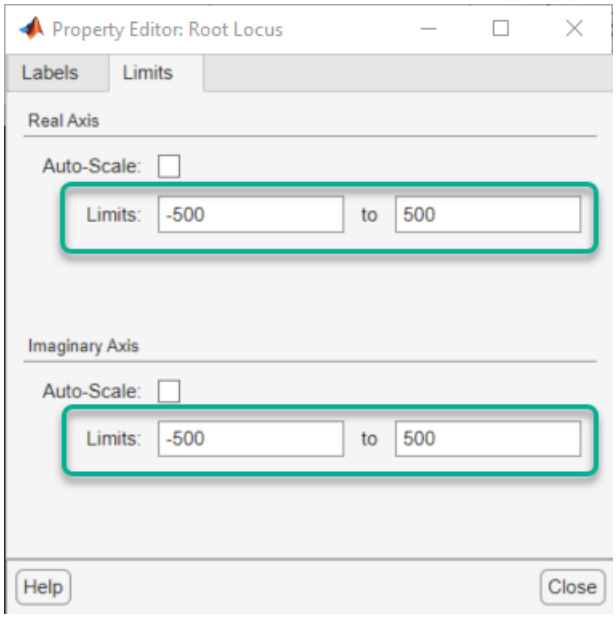
The app displays **Bode Editor** and **Step Response** plots side-by-side.



In the closed-loop step response plot, the rise time is around two seconds, which does not satisfy the design requirements.

To make the root locus diagram easier to read, zoom in. In the **Root Locus Editor**, right-click the plot area and select **Properties**.

In the Property Editor dialog box, on the **Limits** tab, specify **Real Axis** and **Imaginary Axis** limits from `-500` to `500`.

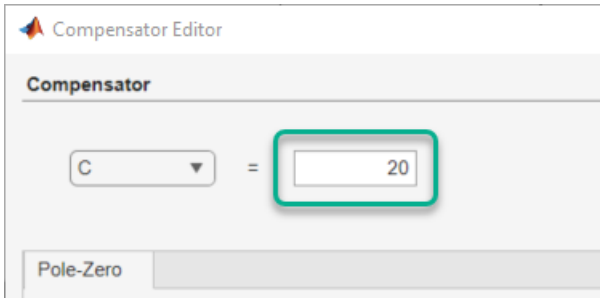


Click **Close**.

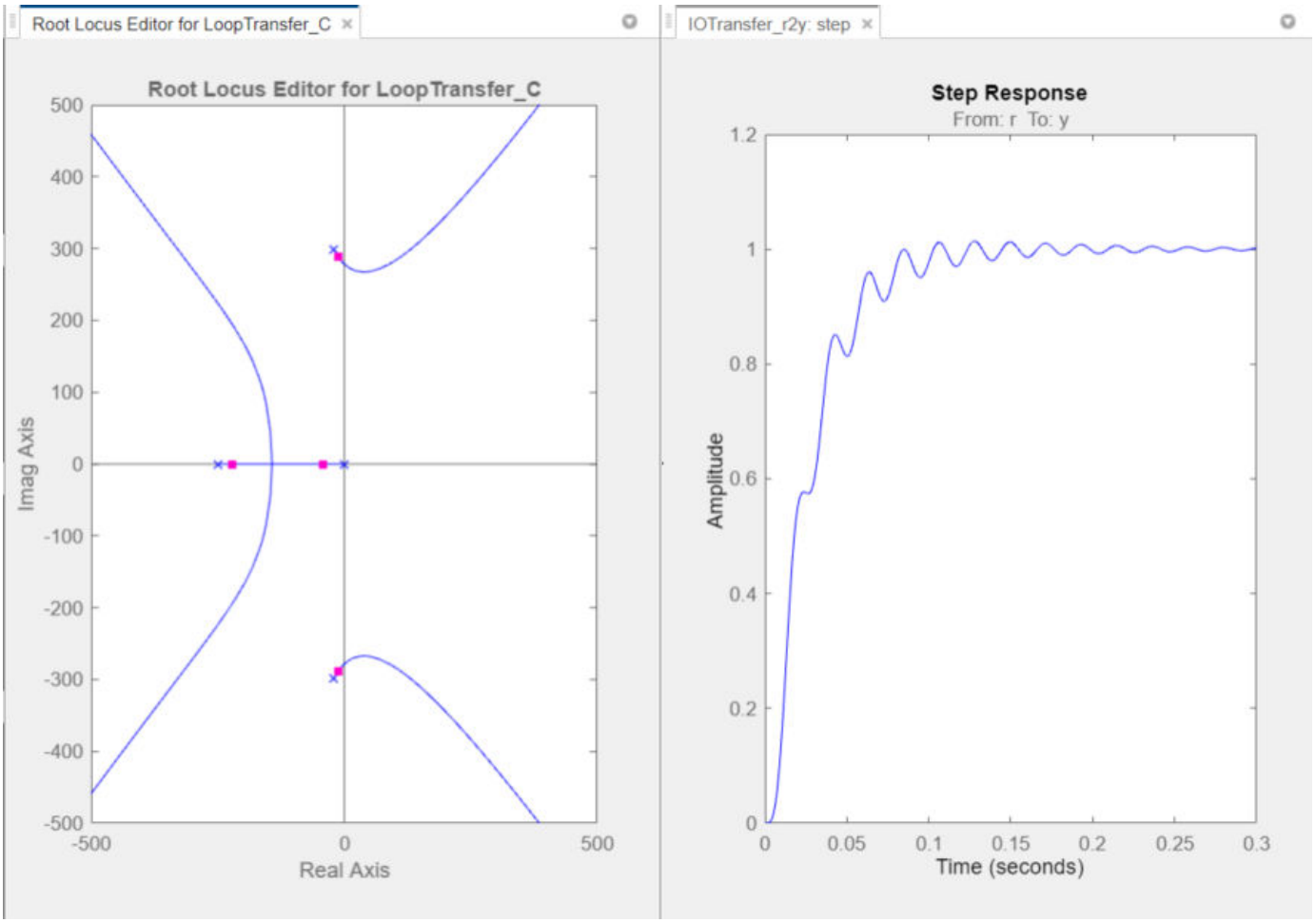
Increase Compensator Gain

To create a faster response, increase the compensator gain. In the **Root Locus Editor**, right-click the plot area and select **Edit Compensator**.

In the Compensator Editor dialog box, specify a gain of `20`.



In the **Root Locus Editor** plot, the closed-loop pole locations move to reflect the new gain value. Also, the **Step Response** plot updates.

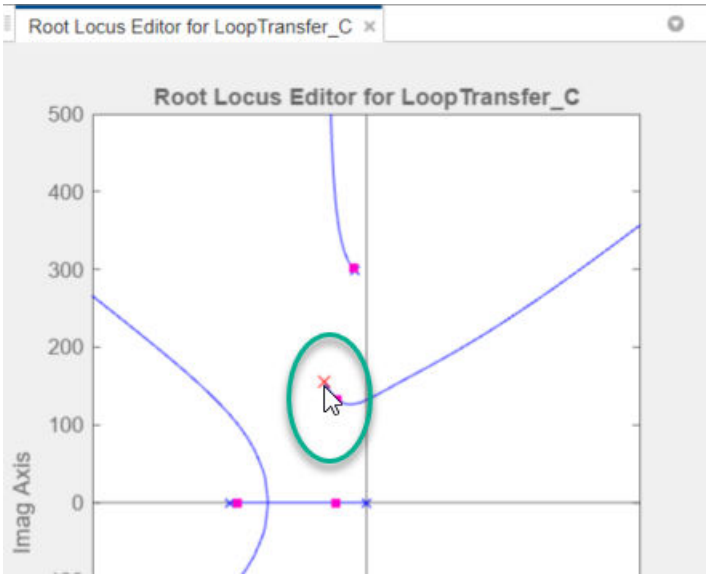


The closed-loop response does not satisfy the settling time requirement and exhibits unwanted ringing.

Increasing the gain makes the system underdamped and further increases lead to instability. Therefore, to meet the design requirements, you must specify additional compensator dynamics. For more information on adding and editing compensator dynamics, see “Edit Compensator Dynamics” on page 12-78.

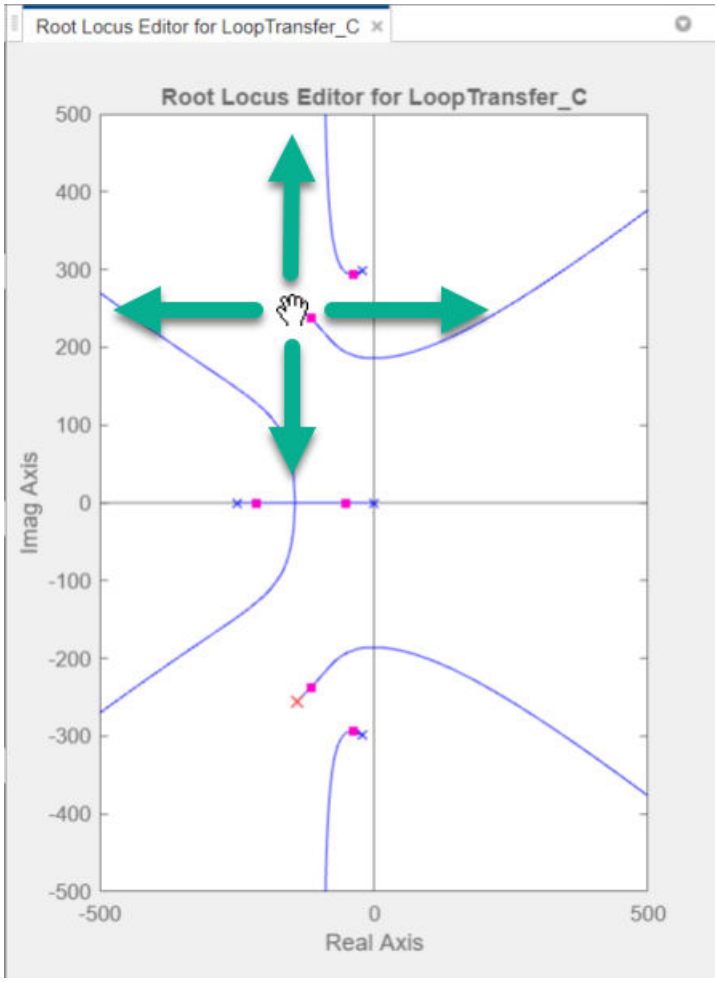
Add Poles to Compensator

To add a complex pole pair to the compensator, in the **Root Locus Editor**, right-click the plot area and select **Add Pole or Zero** > **Complex Pole**. Click the plot area where you want to add one of the complex poles.



The app adds the complex pole pair to the root locus plot as red X's, and updates the step response plot.

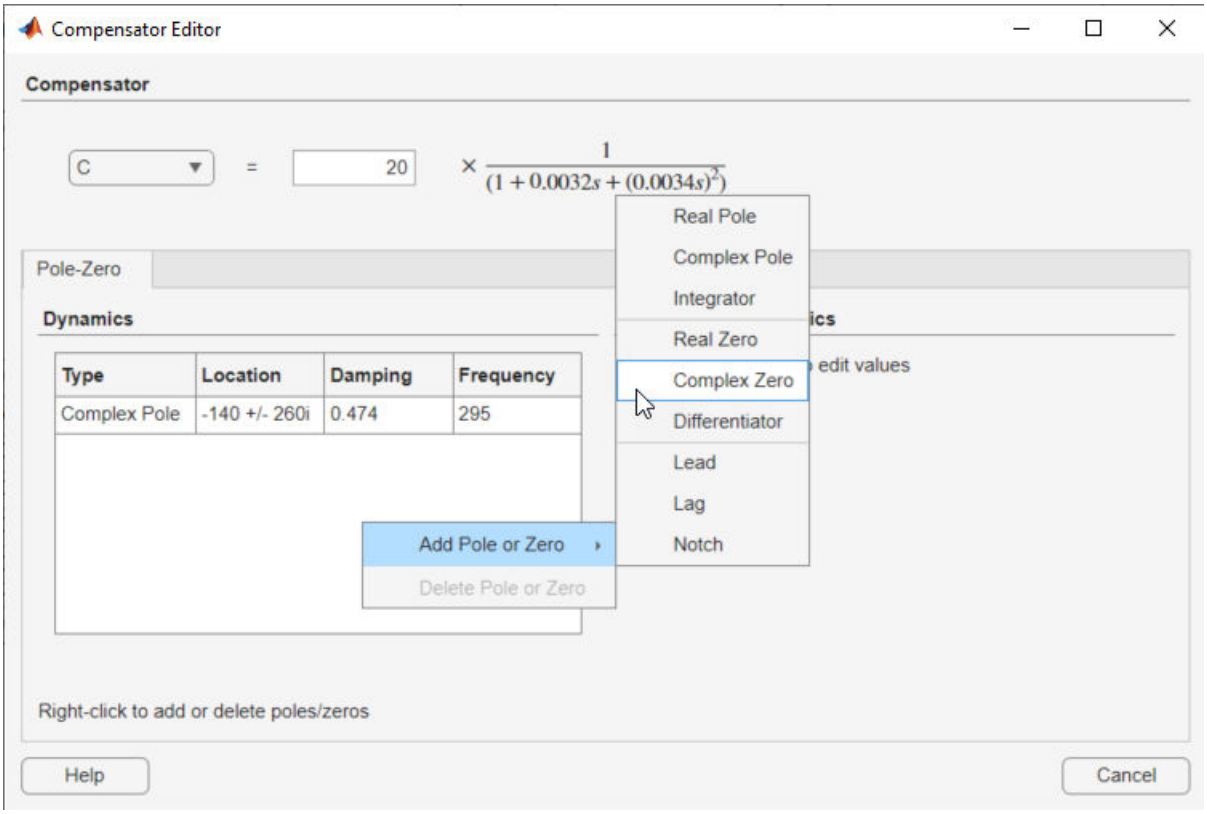
In the **Root Locus Editor**, drag the new poles to locations near –140 ± 260*i*. As you drag one pole, the other pole updates automatically.



Tip As you drag a pole or zero, the app displays the new value in the status bar, on the right side.

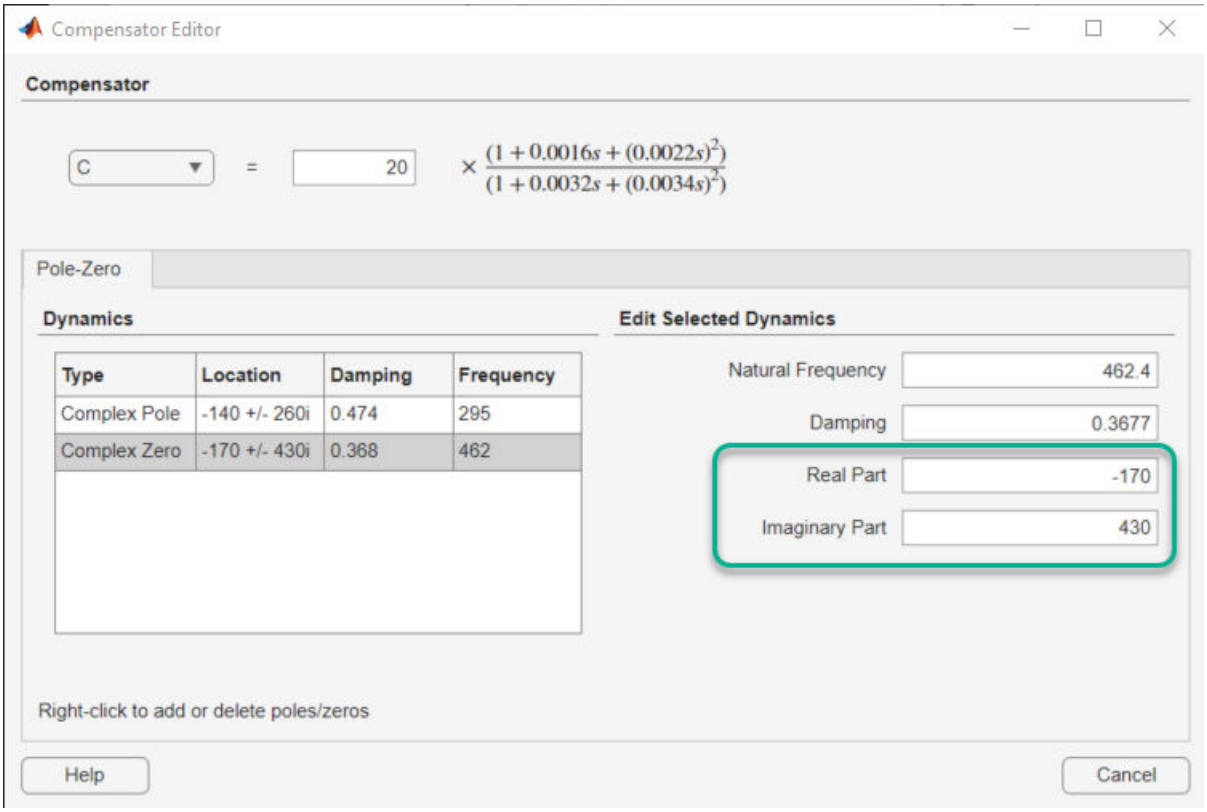
Add Zeros to Compensator

To add a complex zero pair to your compensator, in the Compensator Editor dialog box, right-click the **Dynamics** table, and select **Add Pole or Zero > Complex Zero**

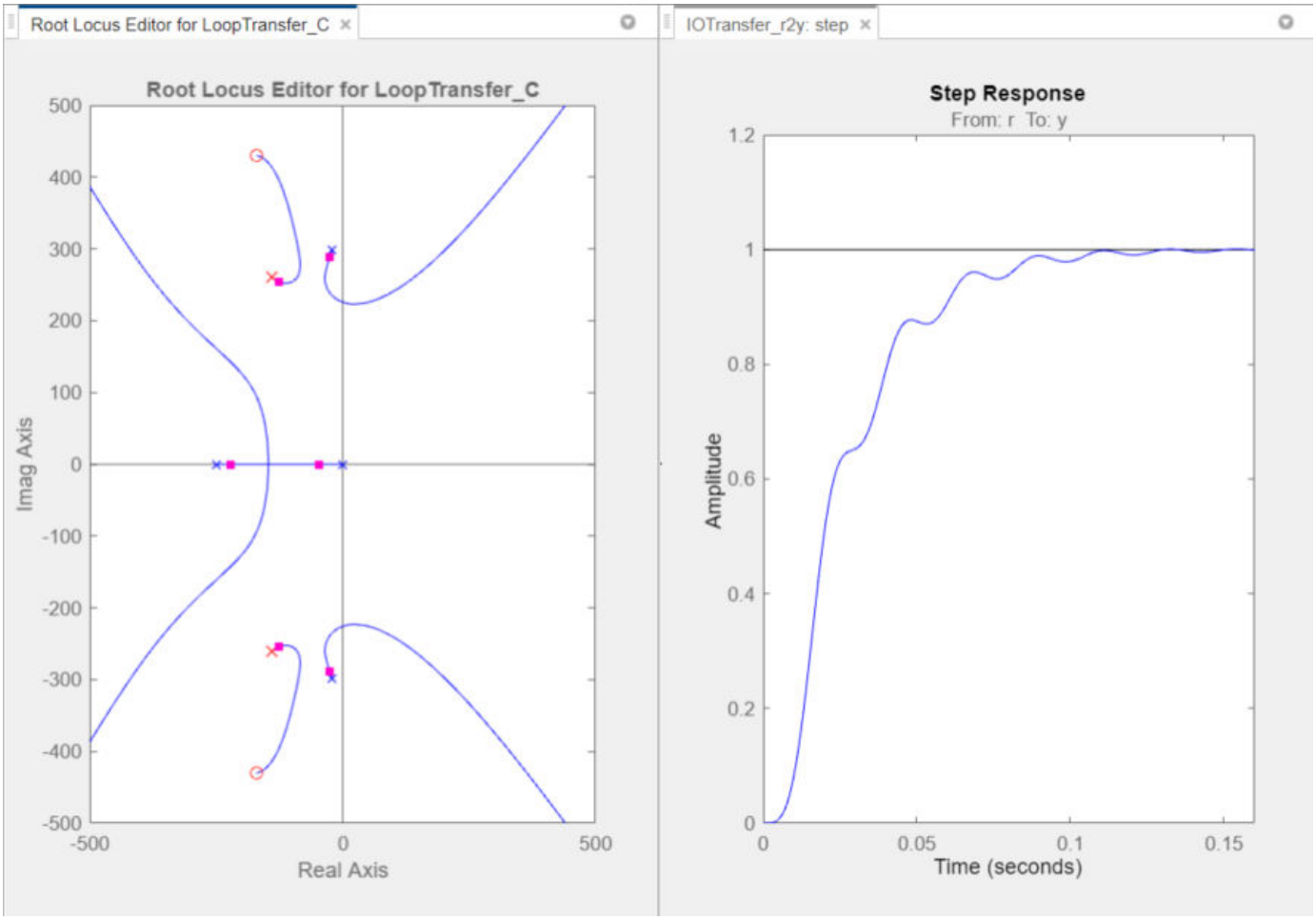


The app adds a pair of complex zeros at –1 ± *i* to your compensator

In the **Dynamics** table, click the **Complex Zero** row. Then in the **Edit Selected Dynamics** section, specify a **Real Part** of `-170` and an **Imaginary Part** of `430`.



The compensator and response plots automatically update to reflect the new zero locations.



In the **Step Response** plot, the settling time is around 0.1 seconds, which does not satisfy the design requirements.

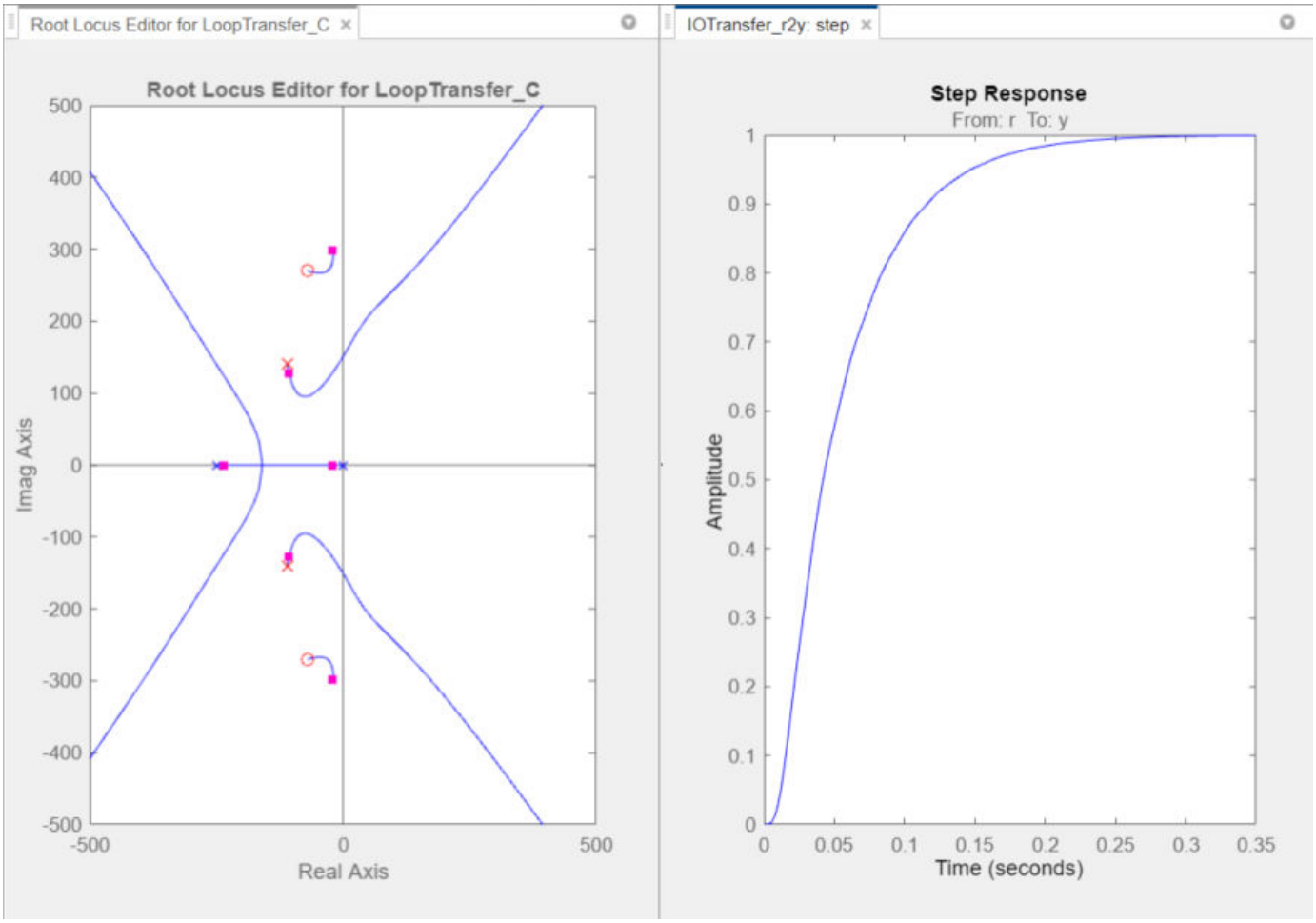
Adjust Pole and Zero Locations

The compensator design process can involve some trial and error. Adjust the compensator gain, pole locations, and zero locations until you meet the design criteria.

One possible compensator design that satisfies the design requirements is:

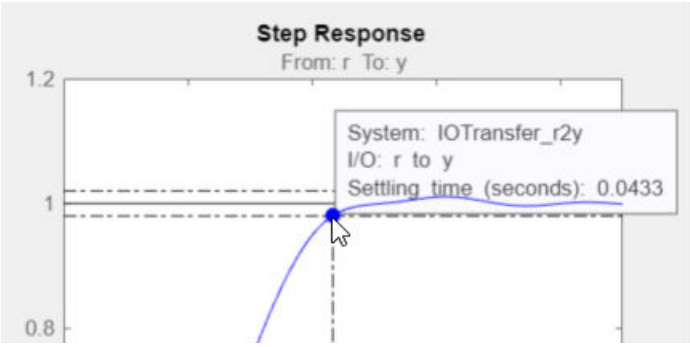
- Compensator gain of `10`
- Complex poles at –110 ± 140*i*
- Complex zeros at –70 ± 270*i*

In the Compensator Editor dialog box, configure your compensator using these values. In the **Step Response** plot, the settling time is around 0.05 seconds.



To verify the exact settling time, right-click the **Step Response** plot area and select **Characteristics > Settling Time**. A settling time indicator appears on the response plot.

To view the settling time, move the cursor over the settling time indicator.



The settling time is about 0.043 seconds, which satisfies the design requirements.

References

[1] Clark, R. N. *Control System Dynamics*, Cambridge University Press, 1996.

See Also

Control System Designer | `rlocusplot`

More About

- “Edit Compensator Dynamics” on page 12-78
- “Control System Designer Tuning Methods” on page 12-5
- “Bode Diagram Design” on page 12-42
- “Nichols Plot Design” on page 12-67

Nichols Plot Design

Nichols plot design is an interactive graphical method of modifying a compensator to achieve a specific open-loop response (loop shaping). Unlike “Bode Diagram Design” on page 12-42, Nichols plot design uses Nichols plots to view the open-loop frequency response. Nichols plots combine gain and phase information into a single plot, which is useful when you are designing to gain and phase margin specifications. You can also use the Nichols plot grid lines to estimate the closed-loop response (see `ngrid`). For more information on Nichols plots, see `nicholsplot`.

Tune Compensator for DC Motor Using Nichols Plot Graphical Design

This example shows how to design a compensator for a DC motor using Nichols plot graphical tuning techniques.

Plant Model and Requirements

The transfer function of the DC motor plant, as described in “SISO Example: The DC Motor”, is:

$$G = \frac{1.5}{s^2 + 14s + 40.02}$$

For this example, the design requirements are:

- Rise time of less than 0.5 seconds
- Steady-state error of less than 5%
- Overshoot of less than 10%
- Gain margin greater than 20 dB
- Phase margin greater than 40 degrees

Open Control System Designer

At the MATLAB command line, create a transfer function model of the plant, and open **Control System Designer** in the **Nichols Editor** configuration.

```
G = tf(1.5,[1 14 40.02]);
controlSystemDesigner('nichols',G);
```

The app opens and imports `G` as the plant model for the default control architecture, **Configuration 1**.

In the app, the following response plots open:

- Open-loop **Nichols Editor** for the `LoopTransfer_C` response. This response is the open-loop transfer function *GC*, where *C* is the compensator and *G* is the plant.
- **Step Response** for the `IOTransfer_r2y` response. This response is the input-output transfer function for the overall closed-loop system.

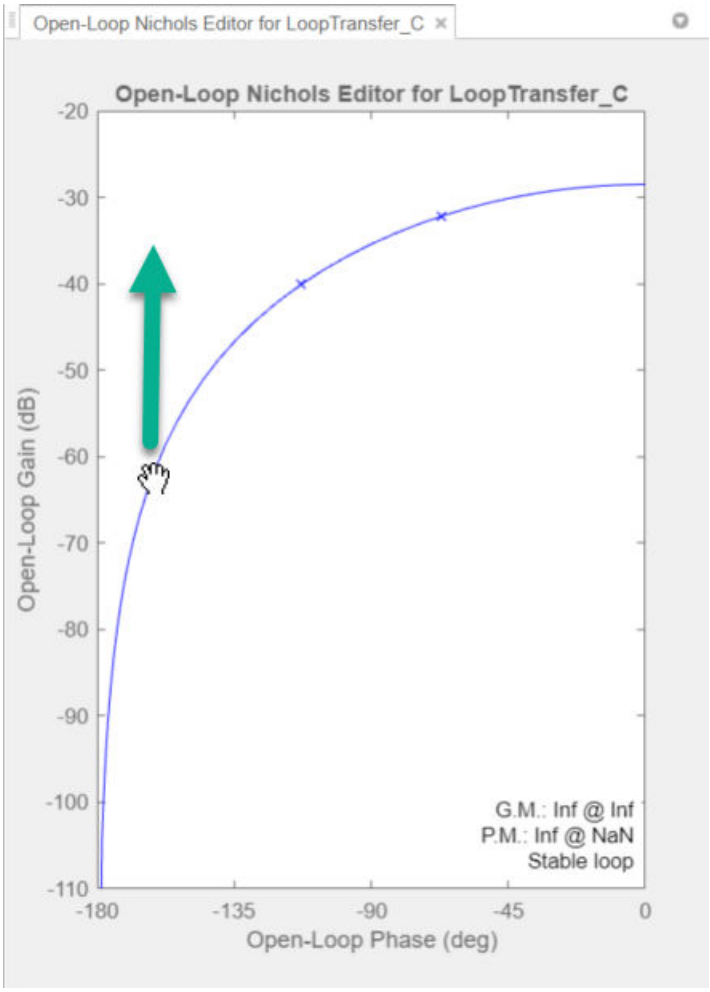
Tip To open the open-loop **Nichols Editor** when **Control System Designer** is already open, on the **Control System** tab, in the **Tuning Methods** drop-down list, select **Nichols Editor**. In the Select Response to Edit dialog box, select an existing response to plot, or create a `New Open-Loop Response`.

The app displays the **Nichols Editor** and **Step Response** plots side-by-side.

Adjust Bandwidth

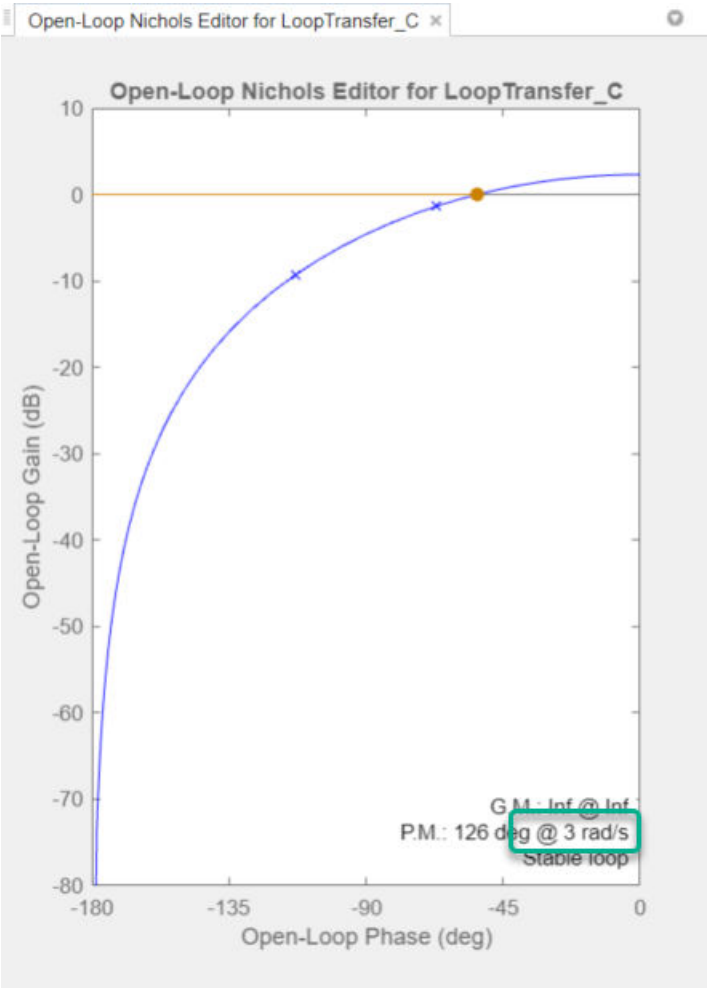
Since the design requires a rise time less than 0.5 seconds, set the open-loop DC crossover frequency to about 3 rad/s. To a first-order approximation, this crossover frequency corresponds to a time constant of 0.33 seconds.

To adjust the crossover frequency increase the compensator gain. In the **Nichols Editor**, drag the response upward. Doing so increases the gain of the compensator.



As you drag the Nichols plot, the app computes the compensator gain and updates the response plots.

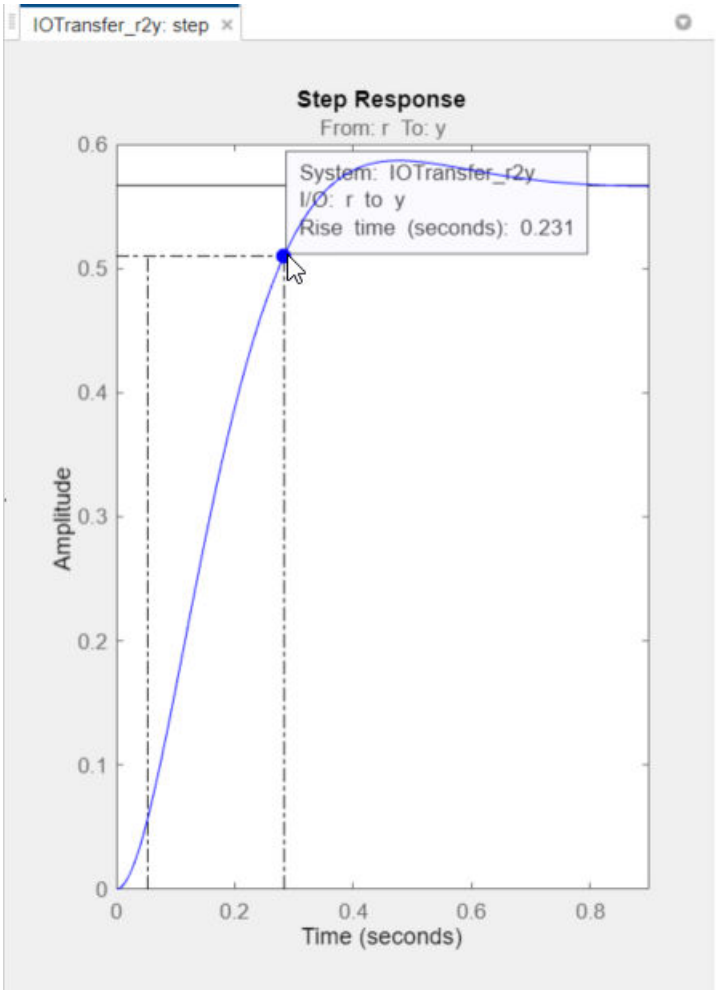
Drag the Nichols plot upward until the crossover frequency is about 3 rad/s.



View Step Response Characteristics

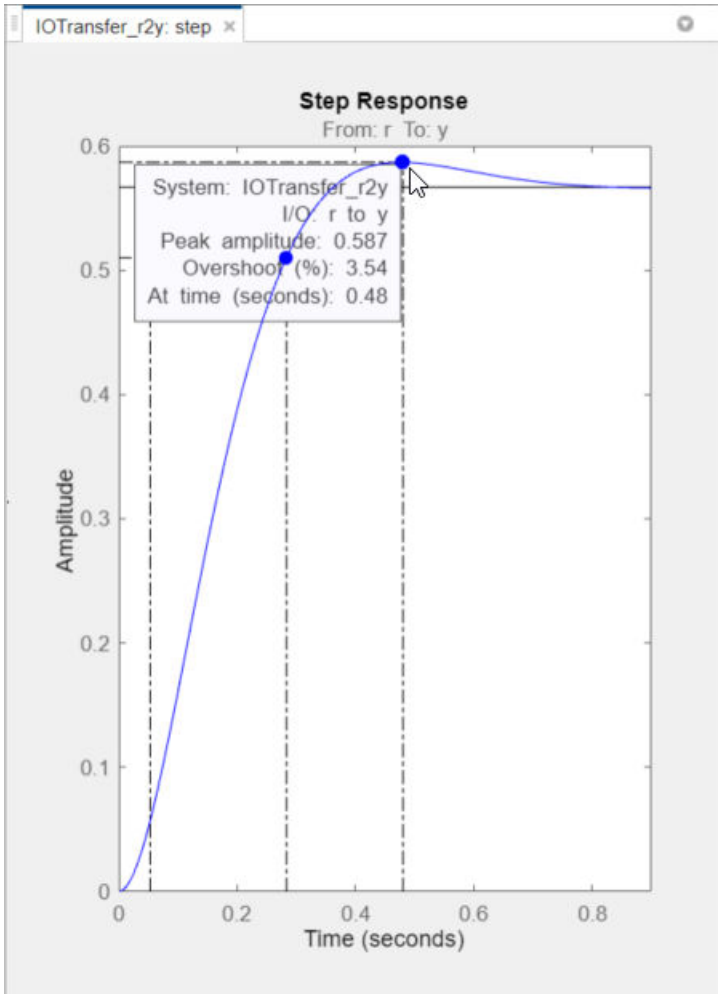
To add the rise time to the **Step Response** plot, right-click the plot area, and select **Characteristics > Rise Time**.

To view the rise time, move the cursor over the rise time indicator.



The rise time is around 0.23 seconds, which satisfies the design requirements.

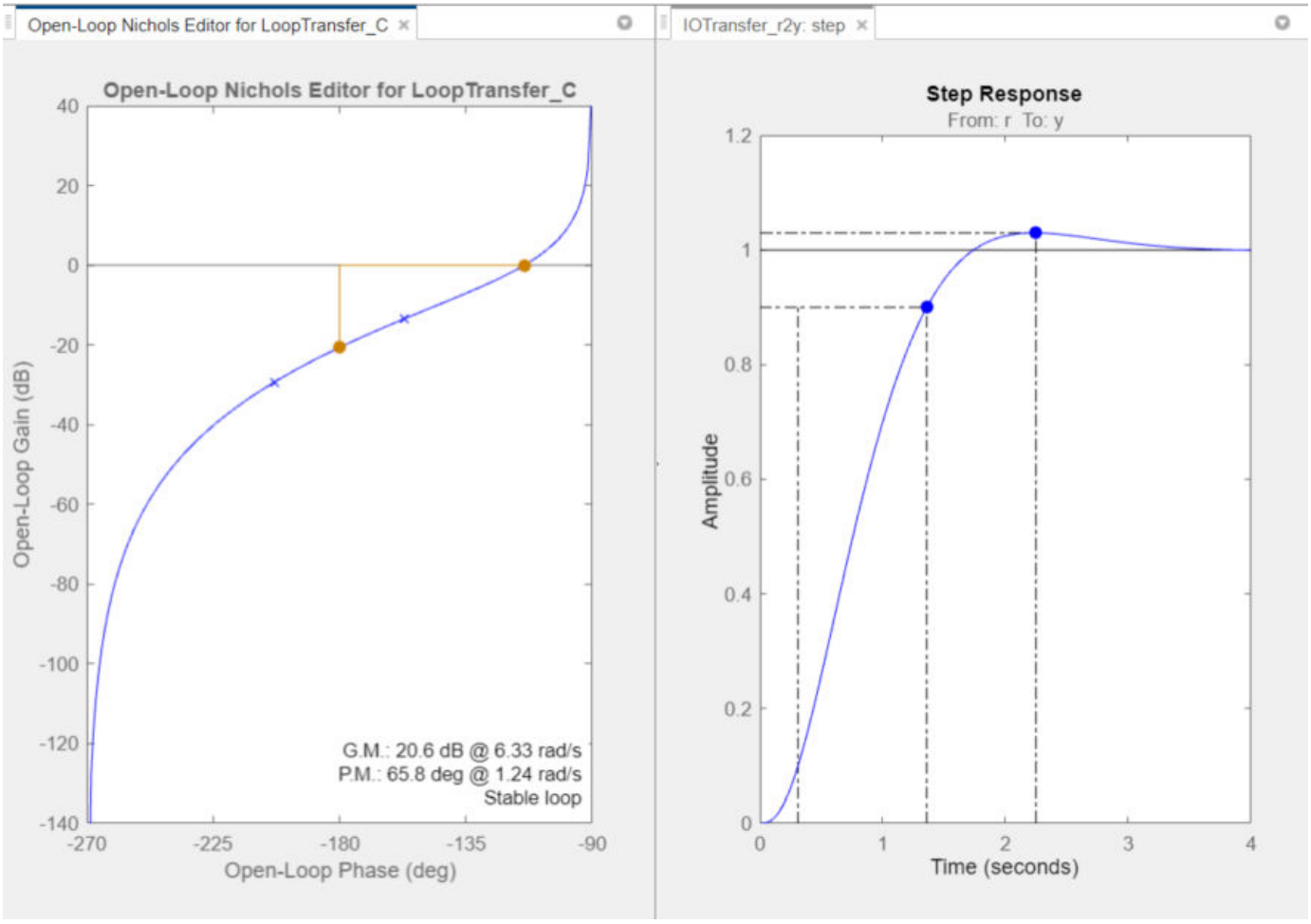
Similarly, to add the peak response to the **Step Response** plot, right-click the plot area, and select **Characteristics > Peak Response**.



The peak overshoot is around 3.5%.

Add Integrator to Compensator

To meet the 5% steady-state error requirement, eliminate steady-state error from the closed-loop step response by adding an integrator to your compensator. In the **Nichols Editor** right-click in the plot area, and select **Add Pole or Zero > Integrator**.



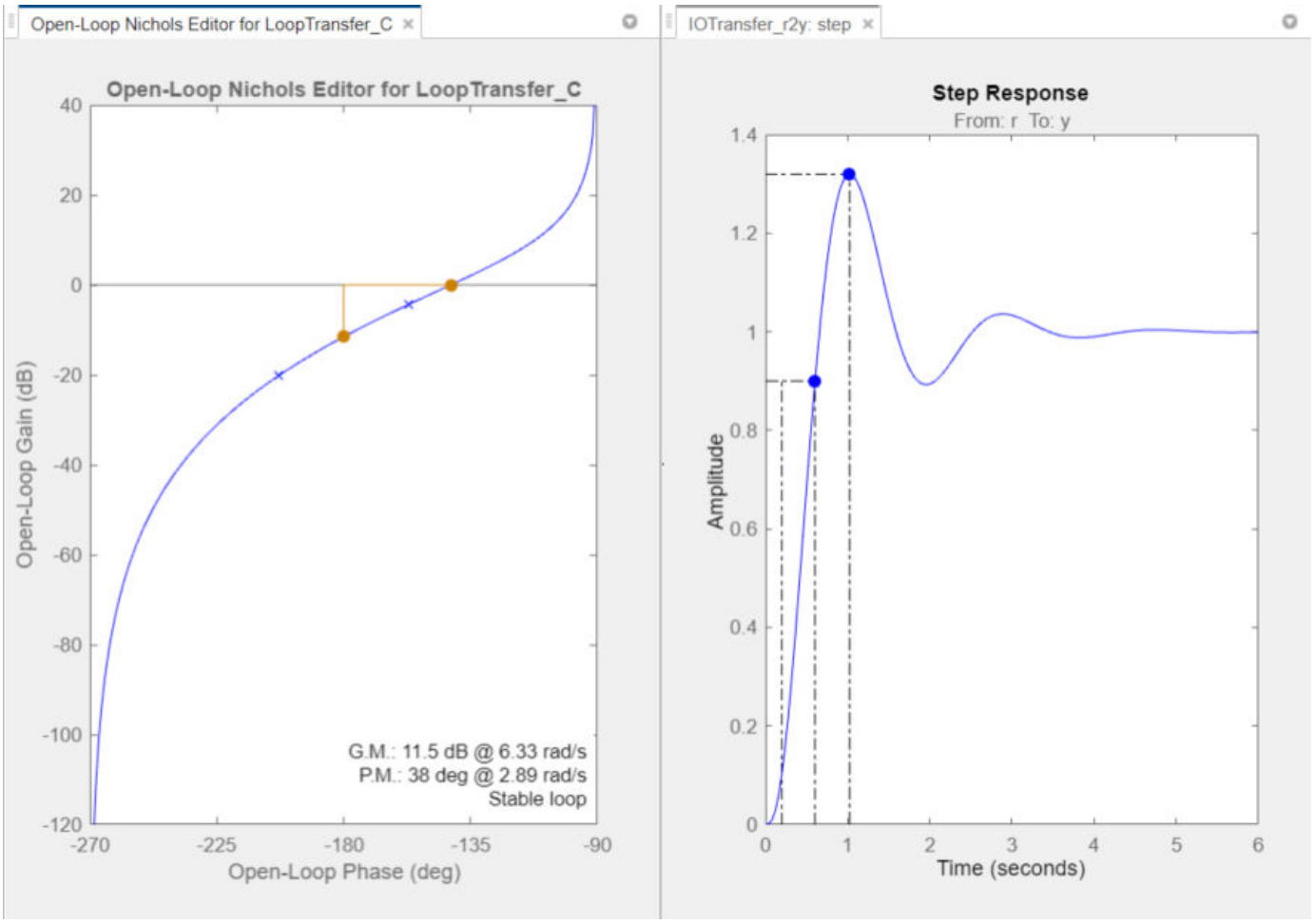
Adding an integrator produces zero steady-state error. However, changing the compensator dynamics also changes the crossover frequency, increasing the rise time. To reduce the rise time, increase the crossover frequency to around 3 rad/s.

Adjust Compensator Gain

To return the crossover frequency to around 3 rad/s, increase the compensator gain further. Right-click the **Nichols Editor** plot area, and select **Edit Compensator**.

In the Compensator Editor dialog box, in the **Compensator** section, specify a gain of 99, and press **Enter**.

The response plots update automatically.

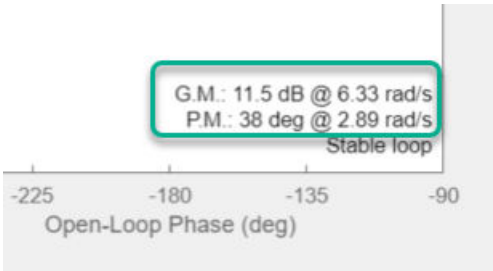


The rise time is around 0.4 seconds, which satisfies the design requirements. However, the peak overshoot is around 32%. A compensator consisting of a gain and an integrator is not sufficient to meet the design requirements. Therefore, the compensator requires additional dynamics.

Add Lead Network to Compensator

In the **Nichols Editor**, review the gain margin and phase margin for the current compensator design. The design requires a gain margin greater than 20 dB and phase margin greater than 40 degrees. The current design does not meet either of these requirements.

To increase the stability margins, add a lead network to the compensator.



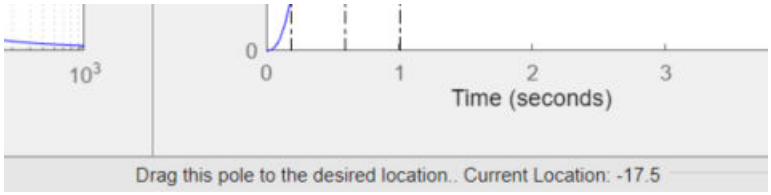
In the **Nichols Editor**, right-click and select **Add Pole or Zero > Lead**.

To specify the location of the lead network pole, click on the magnitude response. The app adds a real pole (red X) and real zero (red 0) to the compensator and to the **Nichols Editor** plot.

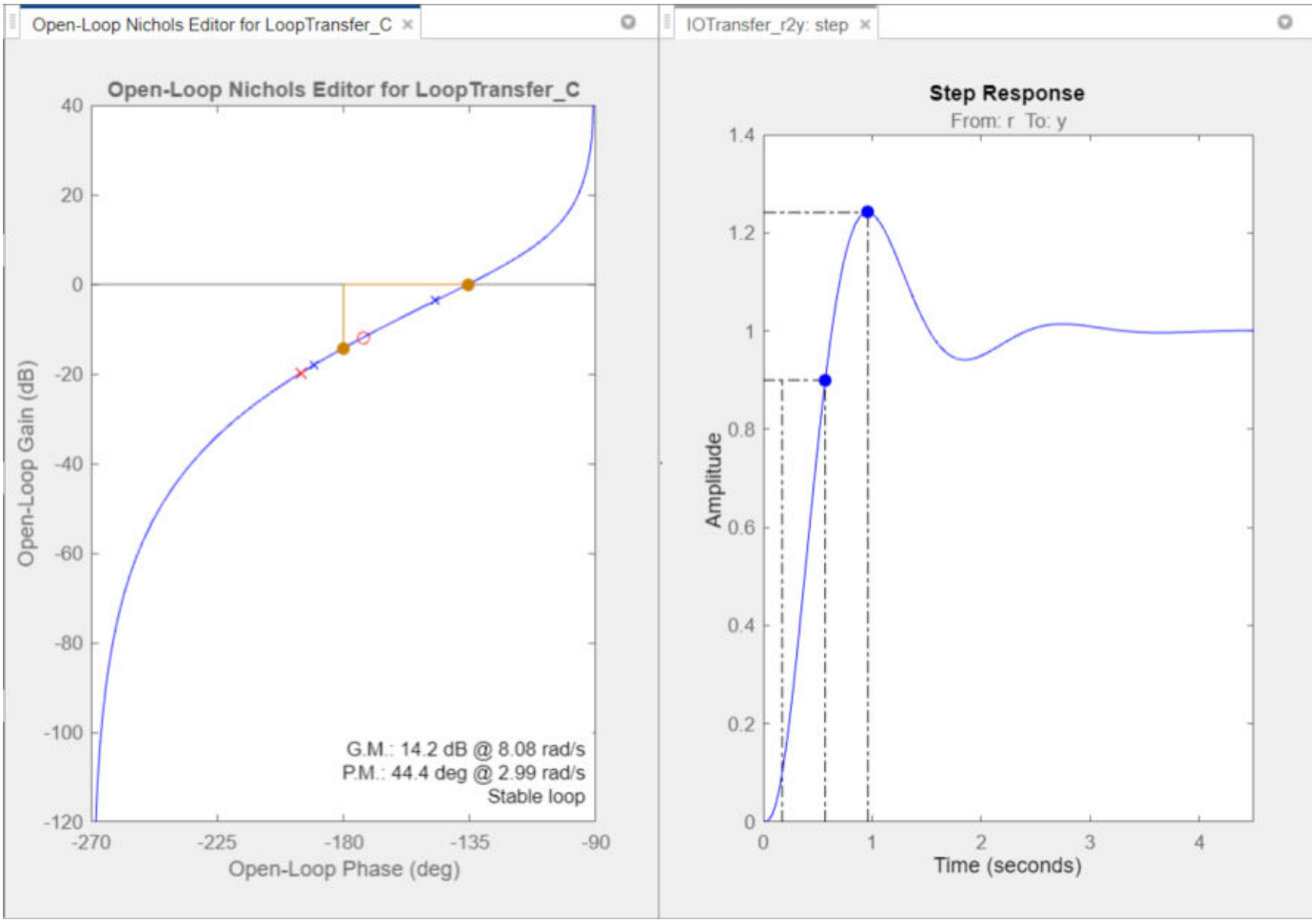
In the **Nichols Editor**, drag the pole and zero to change their locations. As you drag them, the app updates the pole/zero values and updates the response plots.

To decrease the magnitude of a pole or zero, drag it towards the left. Since the pole and zero are on the negative real axis, dragging them to the left moves them closer to the origin in the complex plane.

Tip As you drag a pole or zero, the app displays the new value in the status bar, on the right side.



As an initial estimate, drag the zero to a location around -7 and the pole to a location around -11.



The phase margin meets the design requirements; however, the gain margin is still too low.

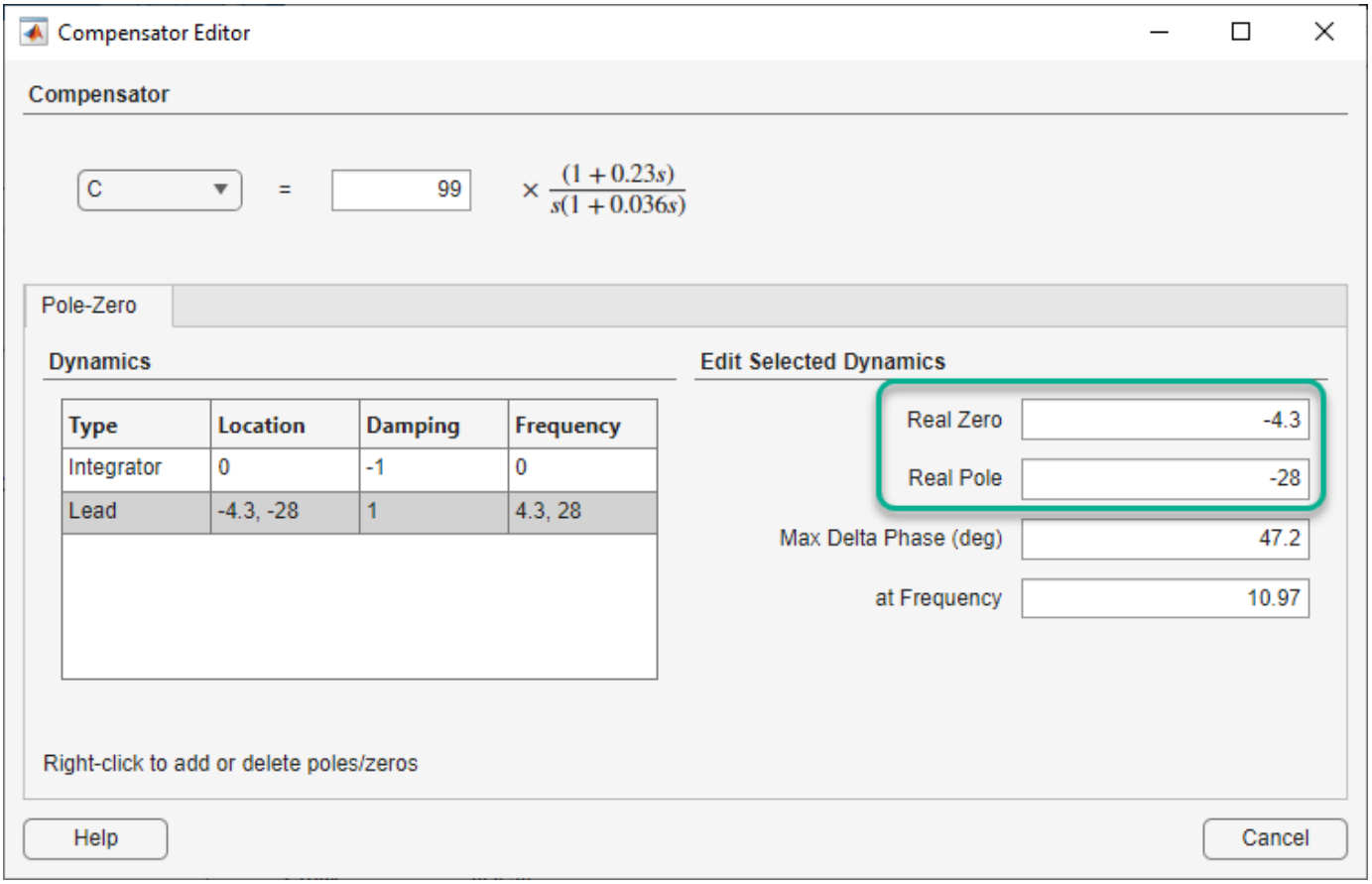
Edit Lead Network Pole and Zero

To improve the controller performance, tune the lead network parameters.

In the Compensator Editor dialog box, in the **Dynamics** section, click the **Lead** row.

In the **Edit Selected Dynamics** section, in the **Real Zero** text box, specify a location of `-4.3`, and press **Enter**. This value is near the slowest (left-most) pole of the DC motor plant.

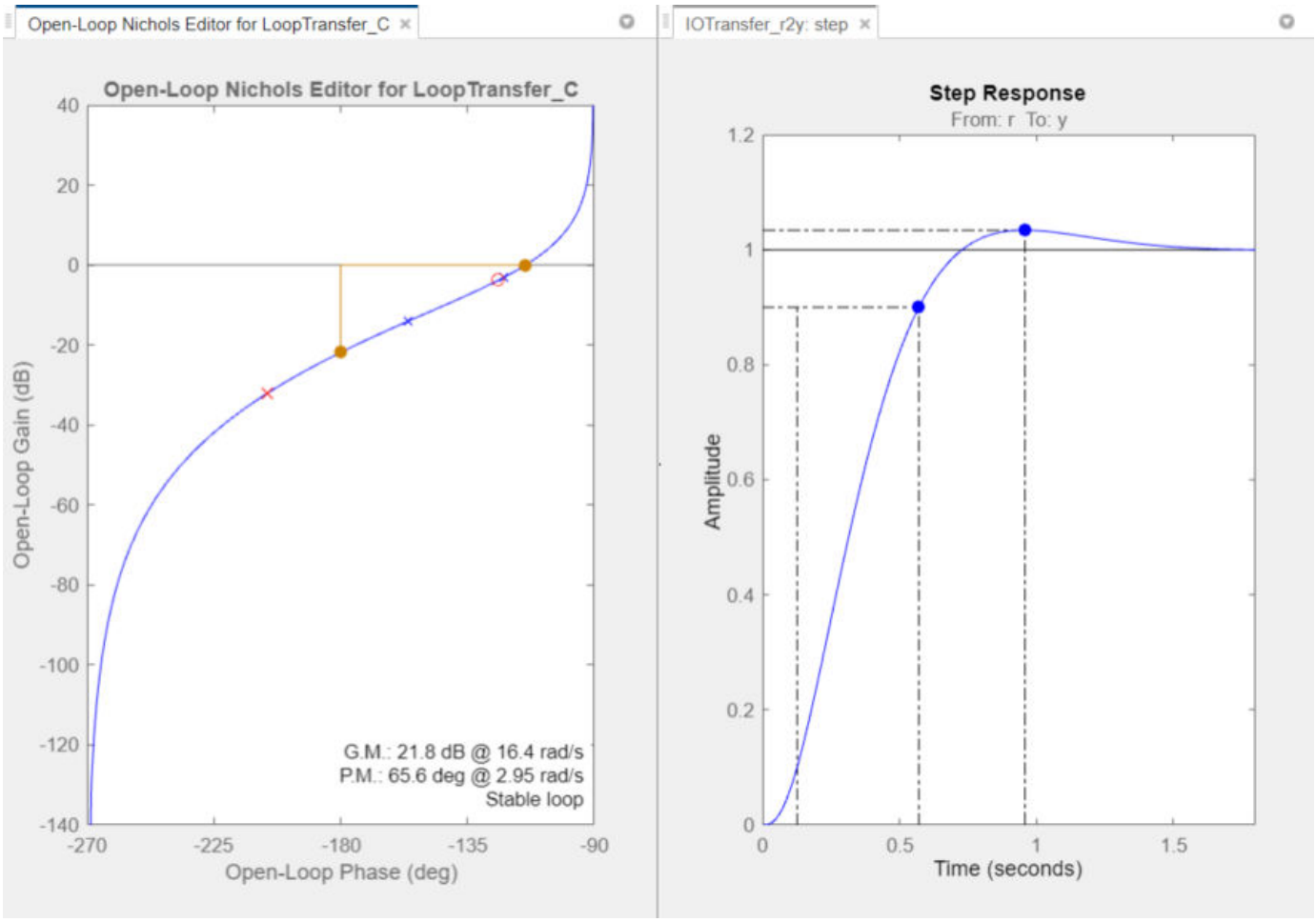
In the **Real Pole** text box, specify a value of `-28`, and press **Enter**.



When you modify a lead network parameters, the **Compensator** and response plots update automatically.

In the app, in the **Nichols Editor**, the gain margin of `20.5` just meets the design requirement.

To add robustness to the system, in the Compensator Editor dialog box, decrease the compensator gain to `84.5`, and press **Enter**. The gain margin increases to `21.8`, and the response plots update.



In **Control System Designer**, in the response plots, compare the system performance to the design requirements. The system performance characteristics are:

- Rise time is 0.445 seconds.
- Steady-state error is zero.
- Overshoot is 3.39%.
- Gain margin is 21.8 dB.
- Phase margin is 65.6 degrees.

The system response meets all of the design requirements.

See Also

Control System Designer | `nicholsplot`

More About

- “Edit Compensator Dynamics” on page 12-78
- “Control System Designer Tuning Methods” on page 12-5

- “Bode Diagram Design” on page 12-42
- “Root Locus Design” on page 12-55

Edit Compensator Dynamics

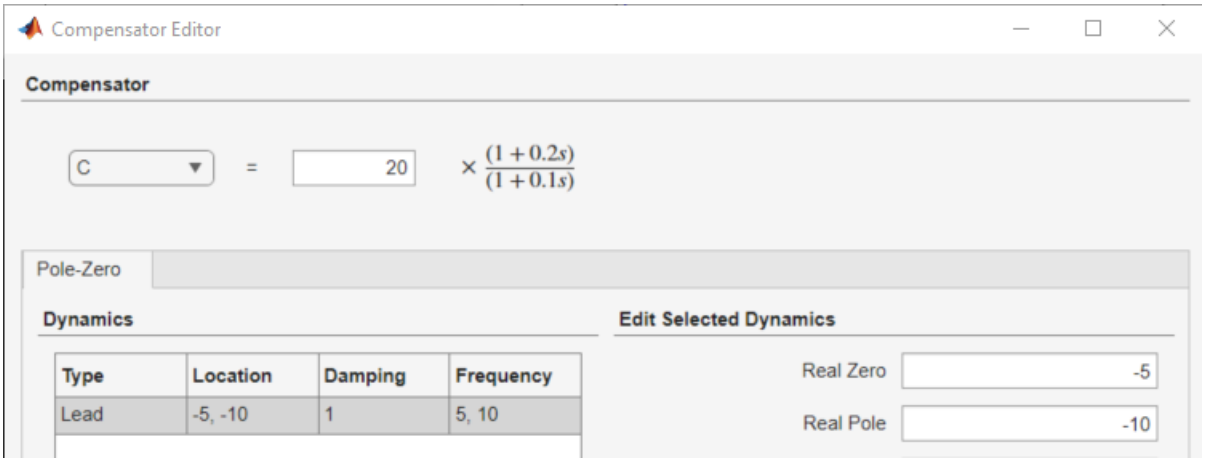
Using **Control System Designer**, you can manually edit compensator dynamics to achieve your design goals. In particular, you can adjust the compensator gain, and you can add the following compensator dynamics:

- Real and complex poles, including integrators
- Real and complex zeros, including differentiators
- Lead and lag networks
- Notch filters

You can add dynamics and modify compensator parameters using the Compensator Editor or using the graphical **Bode Editor**, **Root Locus Editor**, or **Nichols Editor** plots.

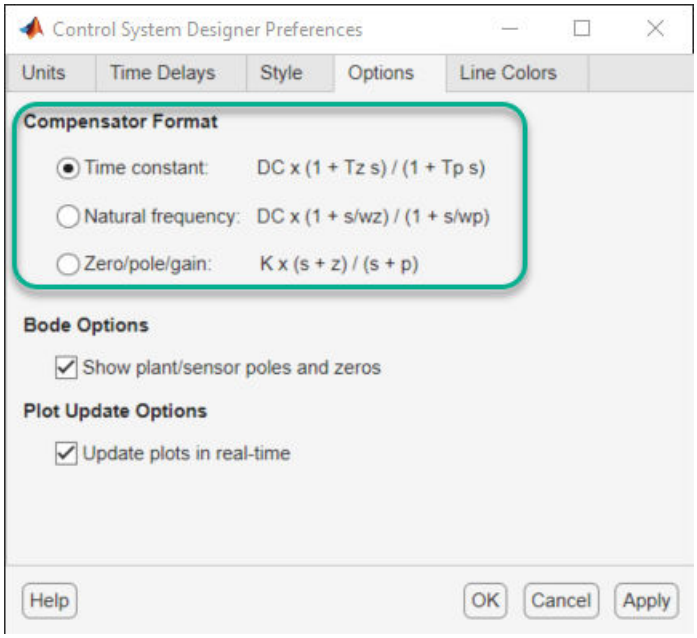
Compensator Editor

To open the Compensator Editor dialog box, in **Control System Designer**, in an editor plot area, right-click and select **Edit Compensator**. Alternatively, in the **Data Browser**, in the **Controllers** section, right-click the compensator you want to edit and click **Open Selection**.

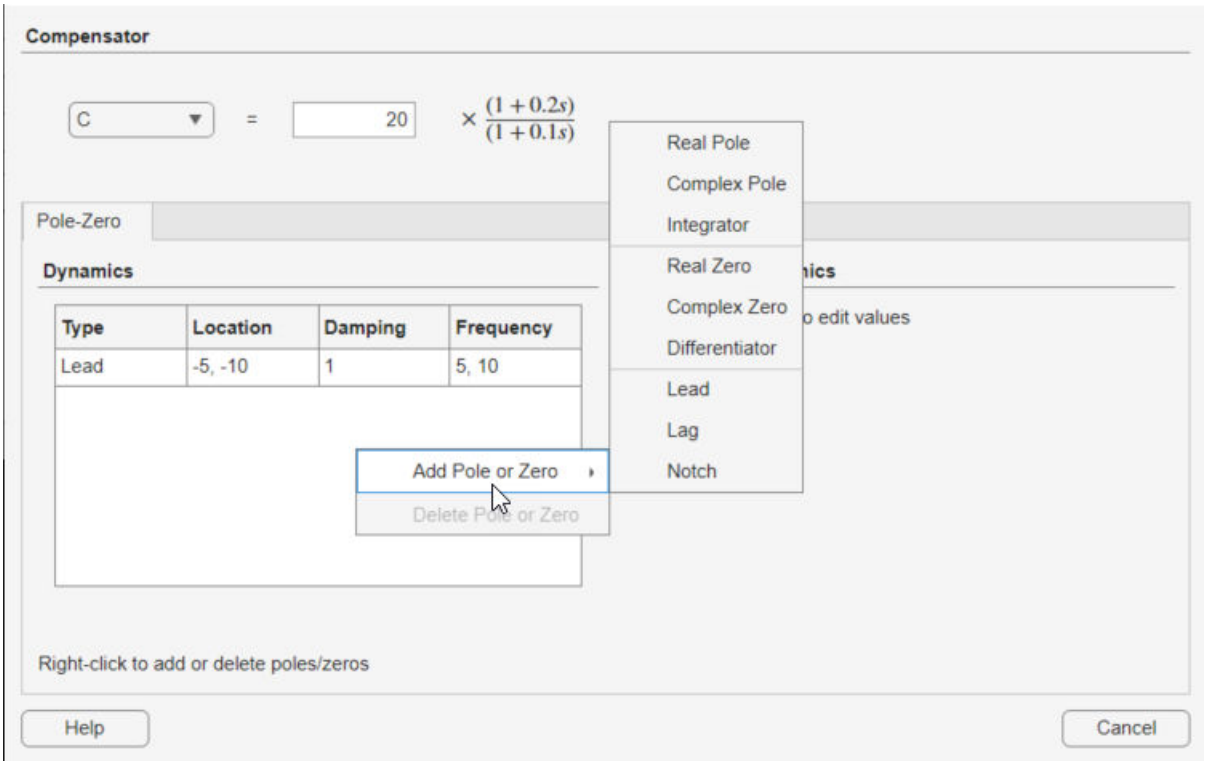


The Compensator Editor displays the transfer function for the currently selected compensator. You can select a different compensator to edit using the drop-down list. By default, the compensator transfer function displays in the time constant format. You can select a different format by changing the corresponding **Control System Designer** preference.

In **Control System Designer**, on the **Control System** tab, click **Preferences**. In the Control System Designer Preferences dialog box, on the **Options** tab, select a **Compensator Format**.



To add poles and zeros to your compensator, in the Compensator Editor, right-click in the **Dynamics** table and, under **Add Pole or Zero**, select the type of pole/zero you want to add.



The app adds a pole or zero of the selected type with default parameters.

To edit a pole or zero, in the **Dynamics** table, click on the pole/zero type you want to edit. Then, in the **Edit Selected Dynamics** section, in the text boxes, specify the pole and zero locations.

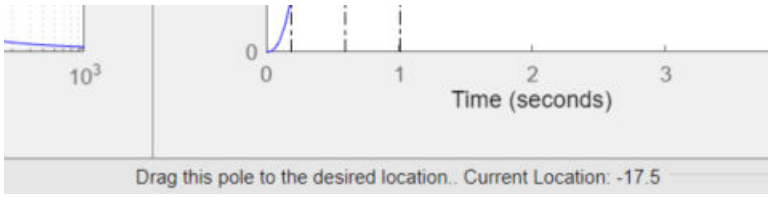
To delete poles and zeros, in the **Dynamics** table, click on the pole/zero type you want to delete. Then, right-click and select **Delete Pole or Zero**.

Graphical Compensator Editing

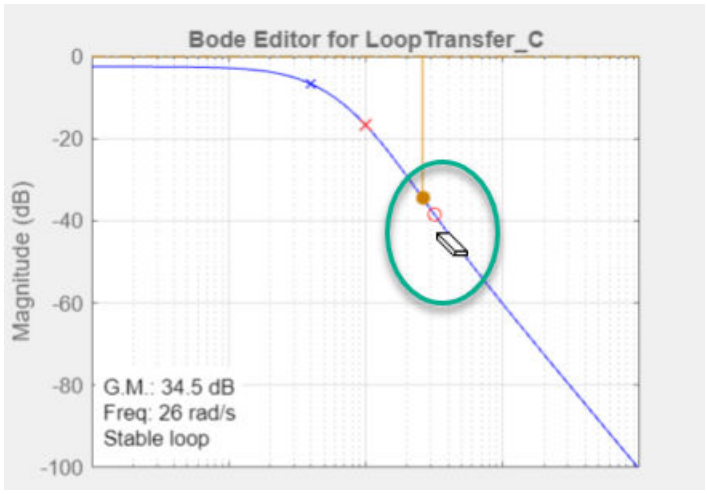
You can also add and adjust poles and zeros directly from Bode Editor, Root Locus Editor, or Nichols Editor plots. Use this method to roughly place poles and zeros in the correct area before fine-tuning their locations using the Compensator Editor.

To add poles and zeros directly from an editor plot, right-click the plot area and, under **Add Pole or Zero**, select the type of pole/zero you want to add. In the editor plot, the app displays the editable compensator poles and zeros as red X's and 0's respectively.

In the editor plots, you can drag poles and zeros to adjust their locations. As you drag a pole or zero, the app displays the new value in the status bar, on the right side.



To delete a pole or zero, right-click the plot area and select **Delete Pole or Zero**. Then, in the editor plot, click the pole or zero you want to delete.



Poles and Zeros

You can add the following poles and zeros to your compensator:

- Real pole/zero — Specify the pole/zero location on the real axis
- Complex poles/zeros — Specify complex conjugate pairs by:
  - Setting the real and imaginary parts directly.
  - Setting the natural frequency, ω_n, and damping ratio, ξ.

- Integrator — Add a pole at the origin to eliminate steady-state error for step inputs and DC inputs.
- Differentiator — Add a zero at the origin.

Lead and Lag Networks

You can add lead networks, lag networks, and combination lead-lag networks to your compensator.

Network Type	Description	Use This To
Lead	One pole and one zero on the negative real axis, with the zero having a lower natural frequency	• Increase stability margins • Increase system bandwidth • Reduce rise time
Lag	One pole and one zero on the negative real axis, with the pole having a lower natural frequency	• Reduce high-frequency gain • Increase phase margin • Improve steady-state accuracy
Lead-Lag	A combination of a lead network and a lag network	Combine the effects of lead and lag networks

To add a lead-lag network, add separate lead and lag networks.

To configure a lead or lag network for your compensator, use one of the following options:

- Specify the pole and zero locations. Placing the pole and zero further apart increases the amount of phase angle change.
- Specify the maximum amount of phase angle change and the frequency at which this change occurs. The app automatically computes the pole and zero locations.

When graphically changing pole and zero locations for a lead or lag compensator, in the editor plot, you can drag the pole and zeros independently.

Notch Filters

If you know that your system has disturbances at a particular frequency, you can add a notch filter to attenuate the gain of the system at that frequency. The notch filter transfer function is:

$$\frac{s^2 + 2\xi_1\omega_n s + \omega_n^2}{s^2 + 2\xi_2\omega_n s + \omega_n^2}$$

where

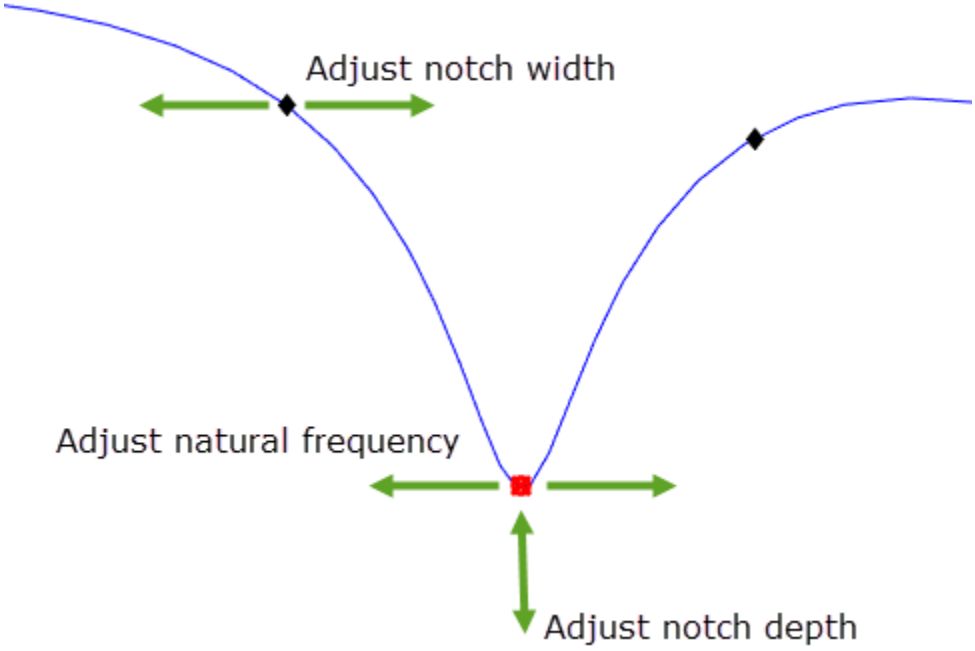
- ω_n is the natural frequency of the notch.
- The ratio ξ_2/ξ_1 sets the depth of the notch.

To configure a notch filter for your compensator, in the Compensator Editor dialog box, you can specify the:

- **Natural Frequency** — Attenuated frequency
- **Notch Depth** and **Notch Width**

- **Damping** for the complex poles and zeros of the transfer function.

When graphically editing a notch filter, in the Bode Editor, you can drag the bottom of the notch to adjust ω_n and the notch depth. To adjust the width of the notch without changing ω_n or the notch depth, you can drag the edges of the notch.



See Also

Control System Designer

More About

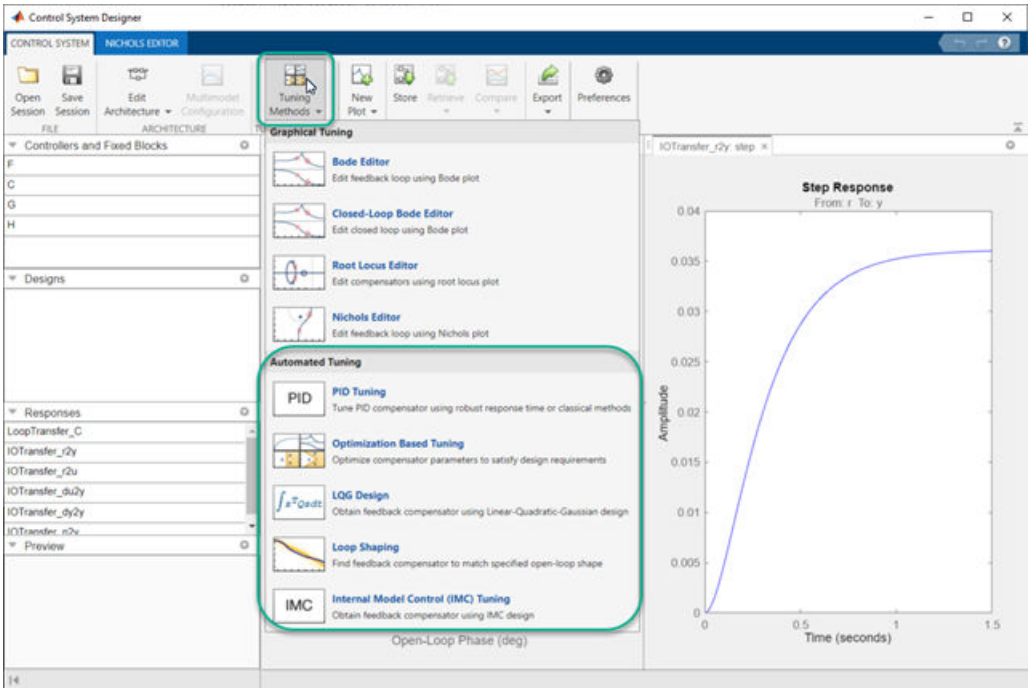
- “Bode Diagram Design” on page 12-42
- “Root Locus Design” on page 12-55
- “Nichols Plot Design” on page 12-67

Design Compensator Using Automated Tuning Methods

This example shows how to tune a compensator using automated tuning methods in **Control System Designer**.

Select Tuning Method

To select an automated tuning method, in **Control System Designer**, click **Tuning Methods**.



Select one of the following tuning methods:

- `PID Tuning` — Tune PID gains to balance performance and robustness or use classical tuning formulas.
- `Optimization Based Tuning` — Optimize compensator parameters using design requirements implemented in graphical tuning and analysis plots (requires Simulink Design Optimization software).
- `LQG Design` — Design a full-order stabilizing feedback controller as a linear-quadratic-Gaussian (LQG) tracker.
- `Loop Shaping`
  - `Free-form structure` — Find a full-order stabilizing feedback controller with a specified open-loop bandwidth or shape (requires Robust Control Toolbox software).
  - `Fixed structure` — Tune a user-specified stabilizing feedback controller with a specified open-loop bandwidth or shape.
- `Internal Model Control (IMC) Tuning` — Obtain a full-order stabilizing feedback controller using the IMC design method.

Select Compensator and Loop to Tune

In the dialog box for your selected tuning method, in the **Compensator** section, select the compensator and loop to tune. You can use the Compensator Editor to specify your compensator structure. For more information, see “Edit Compensator Dynamics” on page 12-78.

- **Compensator** — Select a compensator to tune from the drop-down list. The app displays the current compensator transfer function.
- **Select Loop to Tune** — Select an existing open-loop transfer function to tune from the drop-down list. You can select any open-loop transfer function from the **Data Browser** that includes the selected compensator in series.
- **Add New Loop** — Create a new loop to tune. In the Open-Loop Transfer Function dialog box, select signals and loop openings to configure the loop transfer function.

Note

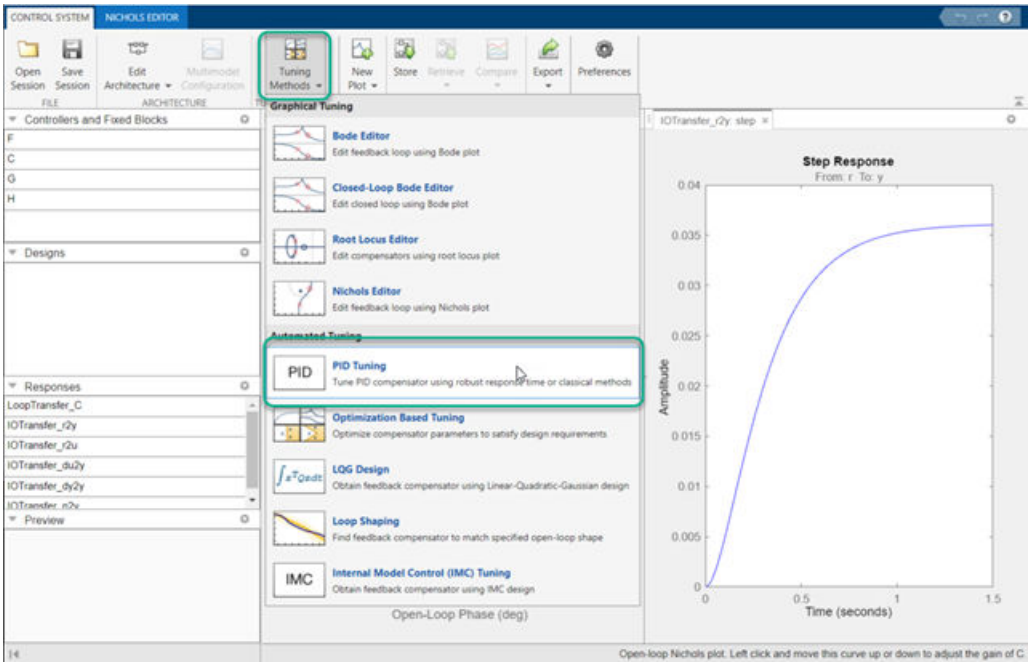
- For optimization-based tuning, you do not specify the compensator and loop to tune in this way. Instead, you define the compensator structure and select compensator and prefilter parameters to optimize. For more information, see “Select Tunable Compensator Elements” (Simulink Design Optimization).
- The structure of the compensator is maintained as poles and zeros after tuning except when performing optimization-based tuning.

PID Tuning

Using **Control System Designer**, you can automatically tune any of the following PID controller types:

- P — Proportional-only control
- I — Integral-only control
- PI — Proportional-integral control
- PD — Proportional-derivative control
- PDF — Proportional-derivative control with a low-pass filter on the derivative term
- PID — Proportional-integral-derivative control
- PIDF — Proportional-integral-derivative control with a low-pass filter on the derivative term

To open the PID Tuning dialog box, in **Control System Designer**, click **Tuning Methods**, and select **PID Tuning**.



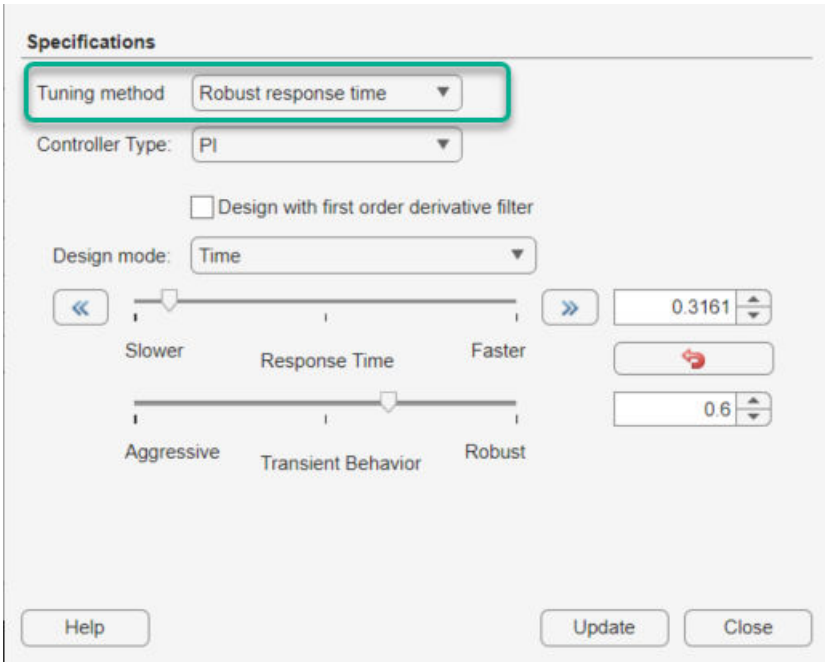
Robust Response Time

The robust response time algorithm automatically tunes PID parameters to balance performance and robustness. Using the robust response time method, you can:

- Tune all parameters for any type of PID controller.
- Design for plants that are stable, unstable, or integrating.

To tune your compensator using this method:

1. In the PID Tuning dialog box, in the **Specifications** section, in the **Tuning method** drop-down list, select `Robust response time`.

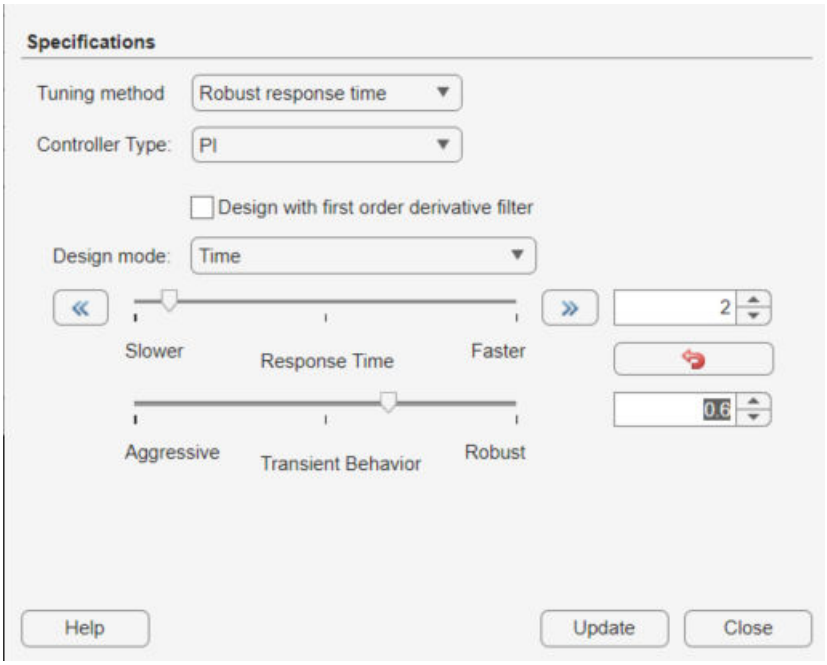


2 Select a **Controller type**. If you choose **PD** or **PID**, check **Design with first order derivative filter** to design a PDF or PIDF controller, respectively.

Tip Adding derivative action to the controller gives the algorithm more freedom to achieve both adequate phase margin and faster response time.

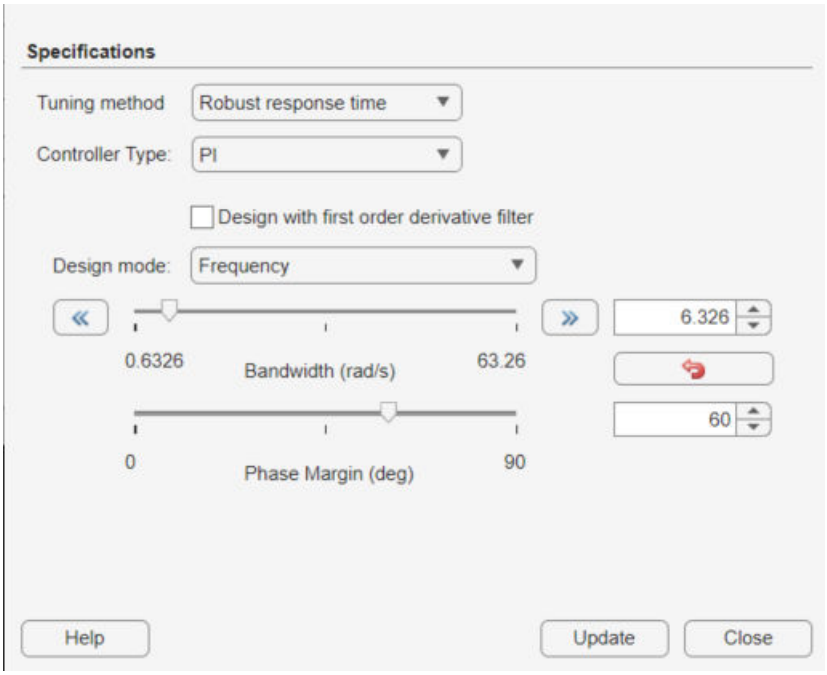
3 In the **Design mode** drop-down list, select one of the following:

- `Time` — Specify controller performance using time-domain parameters.



- **Response Time** — Specify a faster or slower controller response time. To modify the response time by a factor of ten, use the left or right arrows.
- **Transient Behavior** — Specify the controller transient behavior. You can make the controller more aggressive at disturbance rejection or more robust against plant uncertainty.

- `Frequency` — Specify controller performance using frequency-domain parameters.



- **Bandwidth** — Specify the closed-loop bandwidth of the control system. To produce a faster response time, increase the bandwidth. To modify the bandwidth by a factor of ten, use the left or right arrows.
- **Phase Margin** — Specify a target phase margin for the system. To reduce overshoot and create a more robust controller, increase the phase margin.

4 To apply the specified controller design to the selected compensator, click **Update Compensator**.

Note If you previously specified the controller structure manually or using a different automated tuning method, that structure is lost when you click **Update Compensator**.

5 By default, the app automatically computes controller parameters for balanced performance and robustness. To revert to these default parameters at any time, click **Reset Parameters**.

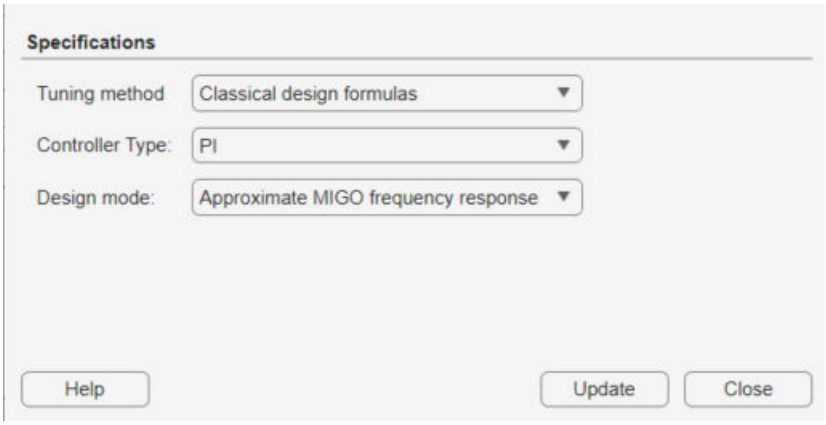
Classical Design Formulas

You can use classical PID design formulas to tune P, PI, PID, and PIDF controllers. These design formulas:

- Require a stable or integrating plant. For more information about the effective plant seen by the compensator, see “Effective Plant for Tuning” on page 12-6.
- Cannot tune the derivative filter. If you select a PIDF controller, the classical design methods set the filter time constant to *Td*/10, where *Td* is the tuned derivative time.

To tune your compensator using a classical method:

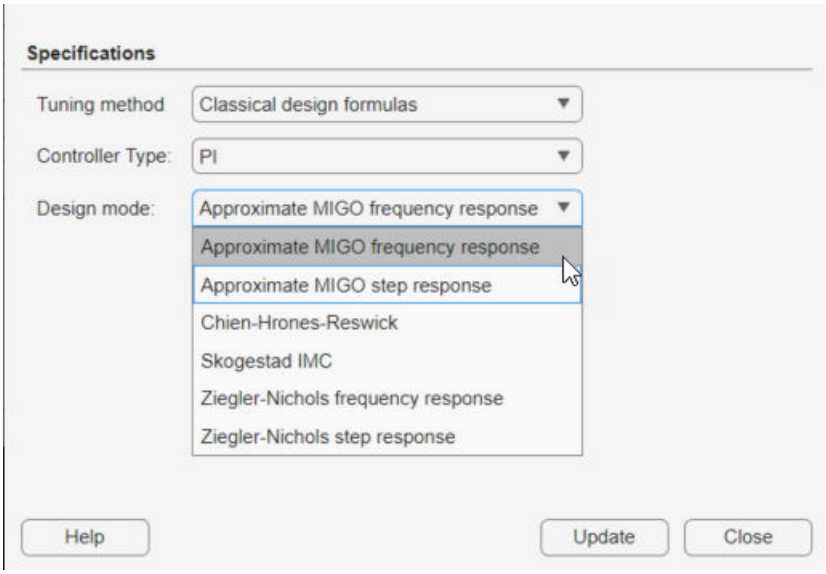
1. In the PID Tuning dialog box, in the **Specifications** section, in the **Tuning method** drop-down list, select `Classical design formulas`.



2. Select a **Controller type**.

Tip Adding derivative action to the compensator gives the algorithm more freedom to achieve both adequate phase margin and faster response time.

3. In the **Design mode** drop-down list, select a classical design formula.



- `Approximate MIGO frequency response` — Compute controller parameters using closed-loop, frequency-domain, approximate M-constrained integral gain optimization (see [1]).
- `Approximate MIGO step response` — Compute controller parameters using open-loop, time-domain, approximate M-constrained integral gain optimization (see [1]).
- `Chien-Hrones-Reswick` — Approximate the plant as a first-order model with a time delay, and compute PID parameters using a Chien-Hrones-Reswick lookup table for 0% overshoot and disturbance rejection (see [2]).

- `Skogestad IMC` — Approximate the plant as a first-order model with a time delay, and compute PID parameters using Skogestad design rules (see [3]).

  Note This method is different from selecting “Internal Model Control Tuning” on page 12-92 as the full-order compensator tuning method.

- `Ziegler-Nichols frequency response` — Compute controller parameters from a Ziegler-Nichols lookup table, based on the ultimate gain and frequency of the system (see [2]).
- `Ziegler-Nichols step response` — Approximate the plant as a first-order model with a time delay, and compute PID parameters using the Ziegler-Nichols design method (see [2]).

4 Apply the specified controller design to the selected compensator. Click **Update Compensator**.

Note If you previously specified the controller structure manually or using a different automated tuning method, that structure is lost when you click **Update Compensator**.

Optimization-Based Tuning

Optimization-based tuning is available only if you have Simulink Design Optimization software installed. You can use this method to design control systems for LTI models by optimizing controller parameters.

Note Optimization-based tuning only changes the values of controller parameters and not the controller structure itself. For information on adding or removing compensator elements, see “Edit Compensator Dynamics” on page 12-78.

To design a controller using optimization-based tuning:

1 Define the structure of the compensators you want to tune. Typically, you design an initial controller either manually or using a different automated tuning method.

2 Open the Response Optimization dialog box. In **Control System Designer**, click **Tuning Methods**, and select **Optimization-Based Tuning**.

3 Select compensator parameters to optimize. On the **Compensators** tab, in the **Optimize** column, select the compensator elements to tune.

  You can optimize elements for any compensator listed in the **Data Browser**.

  Any elements that you do not select in the **Optimize** column remain at their current values during optimization.

4 For each compensator element, specify:

- **Initial guess** — Starting point for the optimization algorithm. To use the current element **Value** as the **Initial guess**, click a row in the table, and click **Use Value as Initial guess**.
- **Minimum** and **Maximum** bounds on the element value. The optimization constrains the search results to the specified range.
- **Typical value** scaling factor for normalizing the compensator elements.

5 On the **Design Requirements** tab, in the **Optimize** column, select the design requirements to satisfy during optimization.

Each design requirement is associated with a plot of a specific response in the **Data Browser**.

For information on adding and editing design requirements, see “Design Requirements” on page 12-9.

6 (optional) Configure optimization options. On the **Optimization** tab, click **Optimization options**.

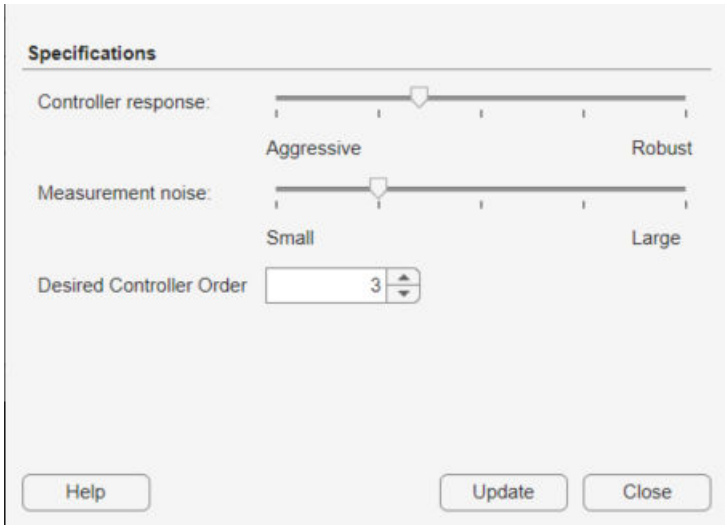
7 Click **Start Optimization**.

For examples of optimization-based tuning, see “Optimize LTI System to Meet Frequency-Domain Requirements” (Simulink Design Optimization) and “Design Optimization-Based PID Controller for Linearized Simulink Model (GUI)” (Simulink Design Optimization).

LQG Design

Linear-quadratic-Gaussian (LQG) control is a technique for designing optimal dynamic regulators and setpoint trackers. This technique allows you to trade off performance and control effort, and to take into account process disturbances and measurement noise.

LQG synthesis generates a full-order feedback controller that guarantees closed-loop stability. The designed controller contains an integrator, which guarantees zero steady-state error for plants without a free differentiator.



To design an LQG controller:

1 Open the LQG Synthesis dialog box. In **Control System Designer**, click **Tuning Methods**, and select **LQG Synthesis**.

2 Specify the transient behavior of the controller using the **Controller response** slider. You can make the controller more aggressive at disturbance rejection or more robust against plant uncertainty. If you believe your model is accurate and that the manipulated variable has a large enough range, an aggressive controller is preferable.

3 Specify an estimate of the level of output measurement noise for your application using the **Measurement noise** slider. To produce a more robust controller, specify a larger noise estimate.

4 Specify your controller order preference using the **Desired controller order** slider. The maximum controller order is dependent on the effective plant dynamics.

5 Apply the specified controller design to the selected compensator. Click **Update Compensator**.

Note If you previously specified the controller structure manually or using a different automated tuning method, that structure is lost when you click **Update Compensator**.

For an example of LQG synthesis using **Control System Designer**, see “Design LQG Tracker Using Control System Designer” on page 12-140.

Loop Shaping

You can use loop shaping to design SISO compensators in **Control System Designer** for free-form or fixed structure compensators. Loop shaping to design free-form compensators requires a Robust Control Toolbox license. Loop shaping generates a stabilizing feedback controller to match, as closely as possible, a target loop shape. You can specify this loop shape as a bandwidth or an open-loop frequency response.

To design a controller using loop shaping:

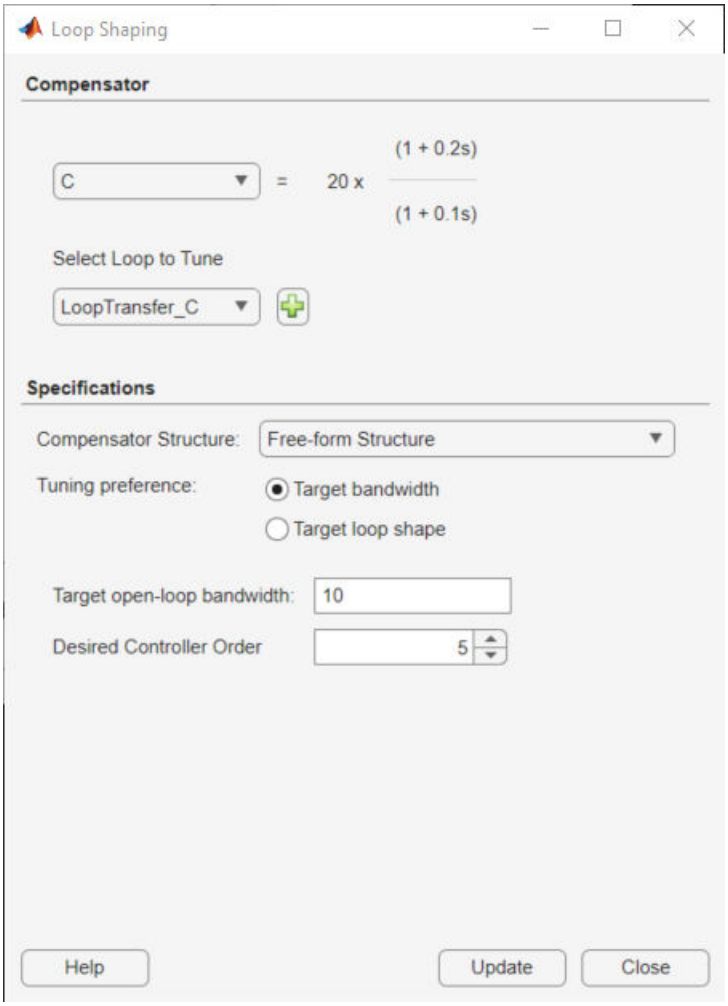
1 Open the Loop Shaping dialog box. In **Control System Designer**, click **Tuning Methods**, and select **Loop Shaping**.

2 Select one of the following tuning preferences:

- **Compensator Structure** — Choose one of the following compensator structures:
  - `Free-form structure` — If you have Robust Control Toolbox software installed, you can use a free-form compensator structure. Use the **Desired controller order** slider to specify your controller order preference. You can use the Compensator Editor to specify your compensator structure. For more information about the Compensator Editor, see “Edit Compensator Dynamics” on page 12-78.
  - `Fixed structure` — When you select `Fixed Structure`, **Control System Designer** will use the supplied controller order to perform the automated loop shaping.
- **Target bandwidth** — Specify a **Target open-loop bandwidth**, ω_b, to produce a loop shape of the specified bandwidth over an integrator, $\frac{\omega_b}{s}$.
- **Target loop shape** — Specify the **Target open-loop shape** as a `tf`, `ss`, or `zpk` object. To limit the frequencies over which to match the target loop shape, specify the **Frequency range for loop shaping** as a two-element row vector.

3 Specify your controller order preference using the **Desired controller order** slider when using a free-form compensator structure. The maximum controller order is dependent on the effective plant dynamics. When you use a fixed compensator structure, **Control System Designer** will use the supplied controller order to perform the automated loop shaping.

4 Apply the specified controller design to the selected compensator. Click **Update Compensator**.



Note If you previously specified the controller structure manually or using a different automated tuning method, that structure is lost when you click **Update Compensator**.

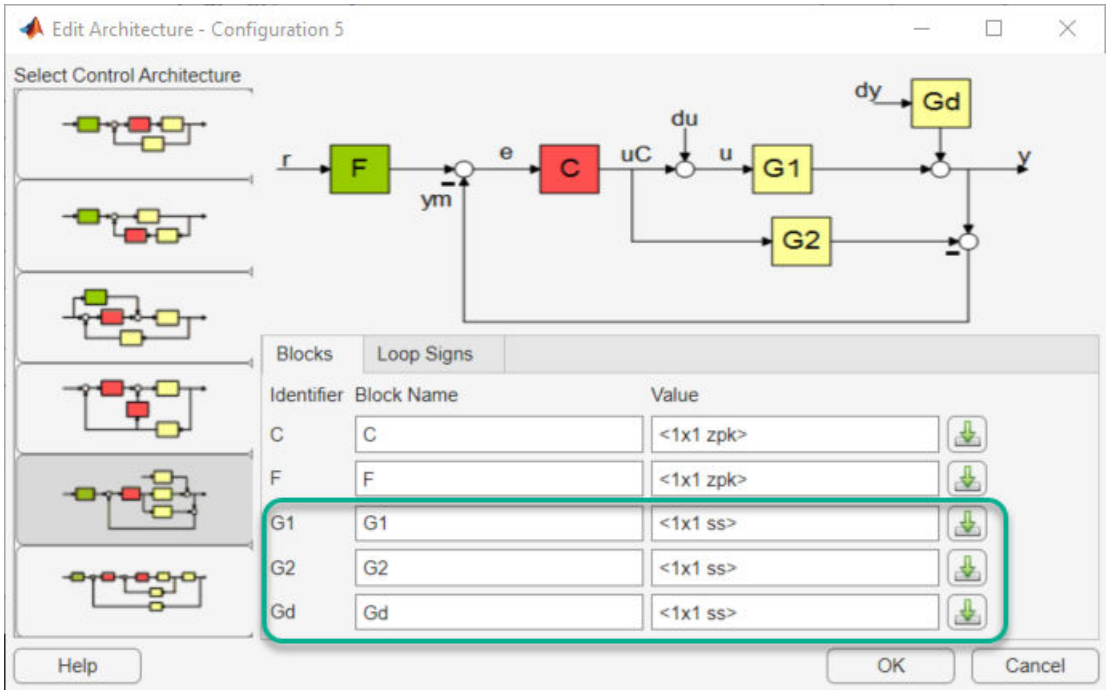
Internal Model Control Tuning

Internal model control (IMC) uses a predictive model of the plant dynamics to compute control actions. IMC design generates a full-order feedback controller that guarantees closed-loop stability when there is no model error. The tuned compensator also contains an integrator, which guarantees zero steady-state offset for plants without a free differentiator. You can use this tuning method for both stable and unstable plants.

To design an IMC controller:

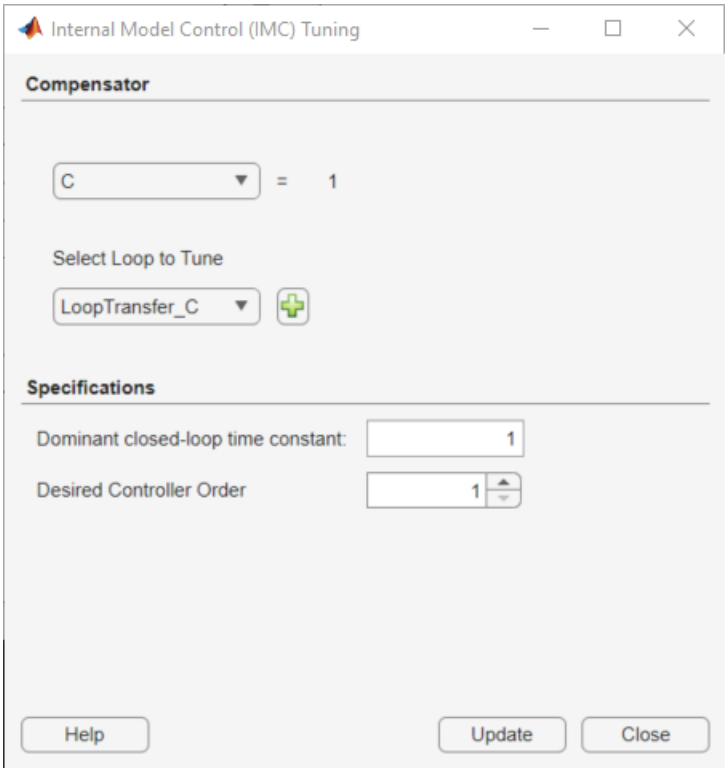
1 Select and configure the IMC control architecture. In **Control System Designer**, click **Edit Architecture**.

In the Edit Architecture dialog box, select the fifth control architecture, and import the plant model, **G1**, predictive model, **G2**, and disturbance model **Gd**.



Click **OK**.

2 Open the Internal Model Control (IMC) Tuning dialog box. In **Control System Designer**, click **Tuning Methods**, and select **Internal Model Control (IMC) Tuning**.



3. Specify a **Dominant closed-loop time constant**. The default value is 5% of the open-loop settling time. In general, increasing this value slows down the closed-loop system and makes it more robust.
4. Specify your controller order preference using the **Desired controller order** slider. The maximum controller order is dependent on the effective plant dynamics.
5. Apply the specified controller design to the selected compensator. Click **Update Compensator**.

Note If you previously specified the controller structure manually or using a different automated tuning method, that structure is lost when you click **Update Compensator**.

For an example of IMC tuning, see “Design Internal Model Controller for Chemical Reactor Plant” on page 12-126.

References

[1] Åström, K. J. and Hägglund, T. “Replacing the Ziegler-Nichols Tuning Rules.” Chapter 7 in *Advanced PID Control*, Research Triangle Park, NC: Instrumentation, Systems, and Automation Society, 2006, pp. 233–270.

[2] Åström, K. J. and Hägglund, T. “Ziegler-Nichols' and Related Methods.” Section 6.2 in *Advanced PID Control*, Research Triangle Park, NC: Instrumentation, Systems, and Automation Society, 2006, pp. 167–176.

[3] Skogestad, S., “Simple analytic rules for model reduction and PID controller tuning.” *Journal of Process Control*, Vol. 13, No. 4, 2003, pp. 291–309.

See Also

Control System Designer

More About

- “Control System Designer Tuning Methods” on page 12-5
- “Design Internal Model Controller for Chemical Reactor Plant” on page 12-126
- “Design LQG Tracker Using Control System Designer” on page 12-140

Analyze Designs Using Response Plots

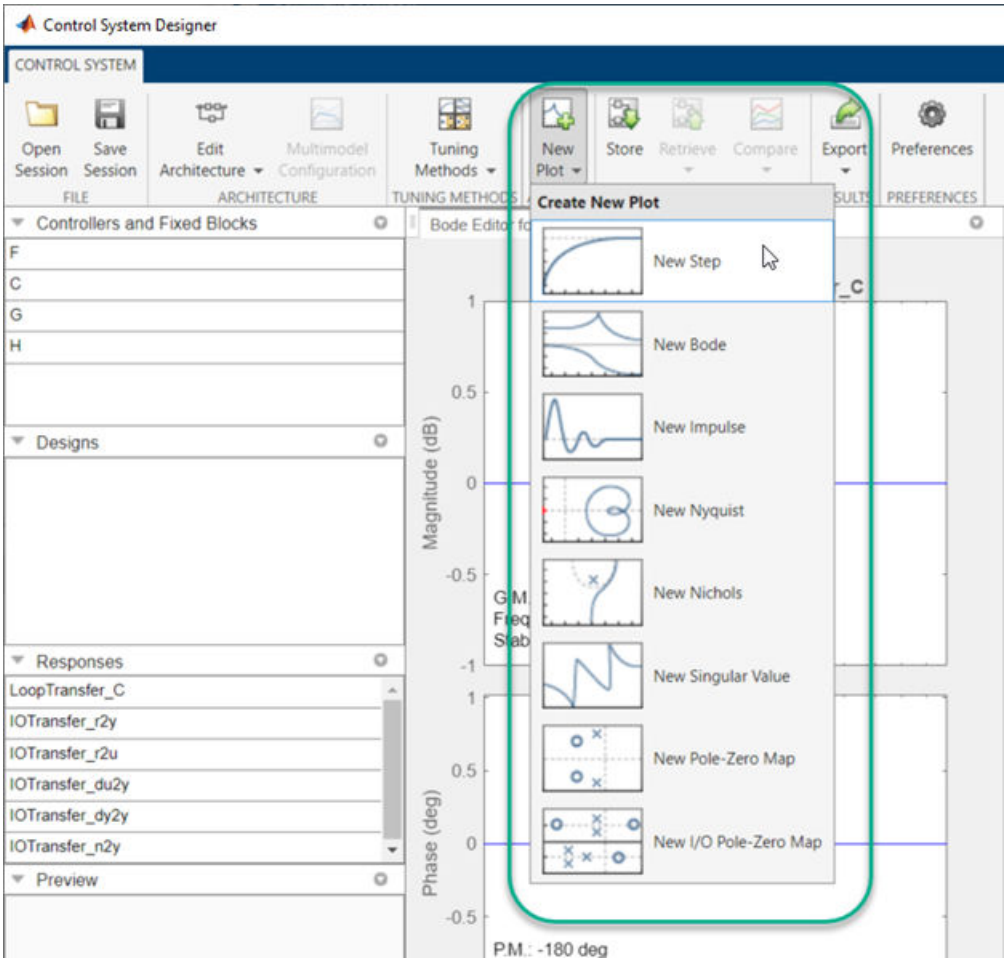
This example shows how to analyze your control system designs using the plotting tools in **Control System Designer**. There are two types of **Control System Designer** plots:

- 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 12-101.

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 12-101.

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 using the **Add signal** drop-down lists. 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.

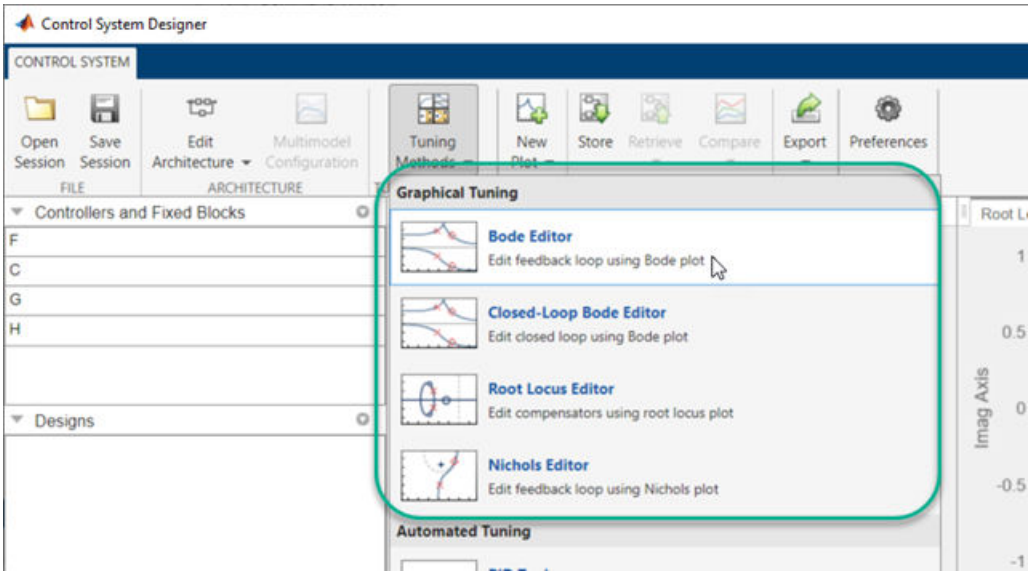
  Use  ,  , and  to reorder and delete signals.

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:

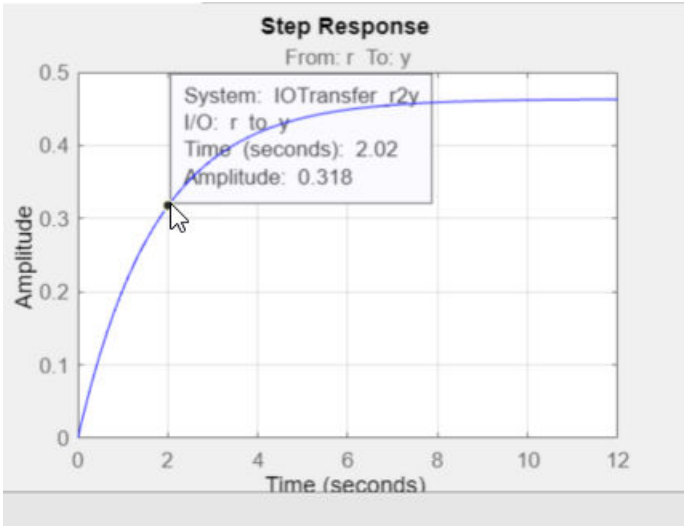
- “Bode Diagram Design” on page 12-42
- “Root Locus Design” on page 12-55
- “Nichols Plot Design” on page 12-67

For more information on interactively editing compensator dynamics, see “Edit Compensator Dynamics” on page 12-78.

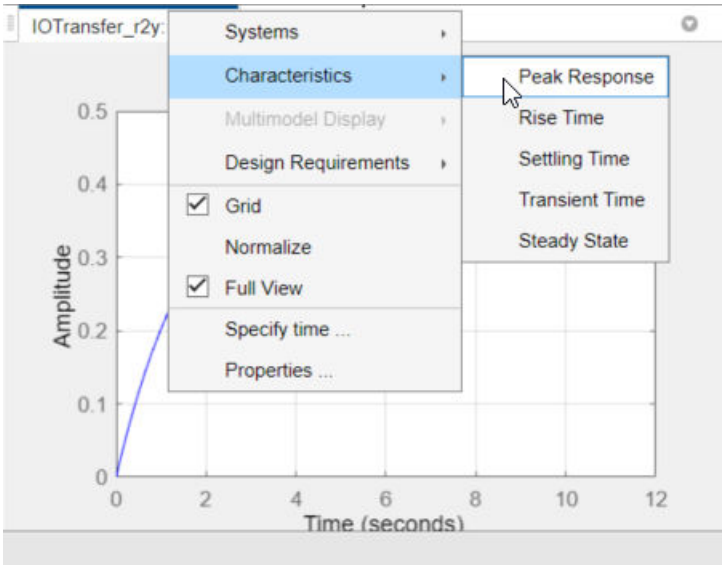
Plot Characteristics

On any analysis plot in **Control System Designer**:

- To see response information and data values, click a line on the plot.

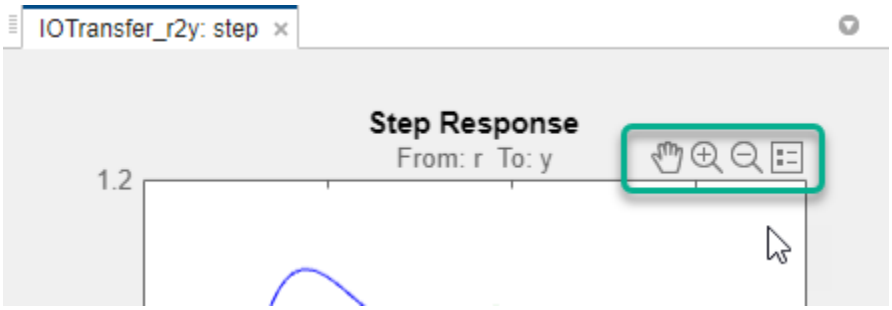


- To view system characteristics, right-click anywhere on the plot, as described in “Frequency-Domain Characteristics on Response Plots” on page 8-8.

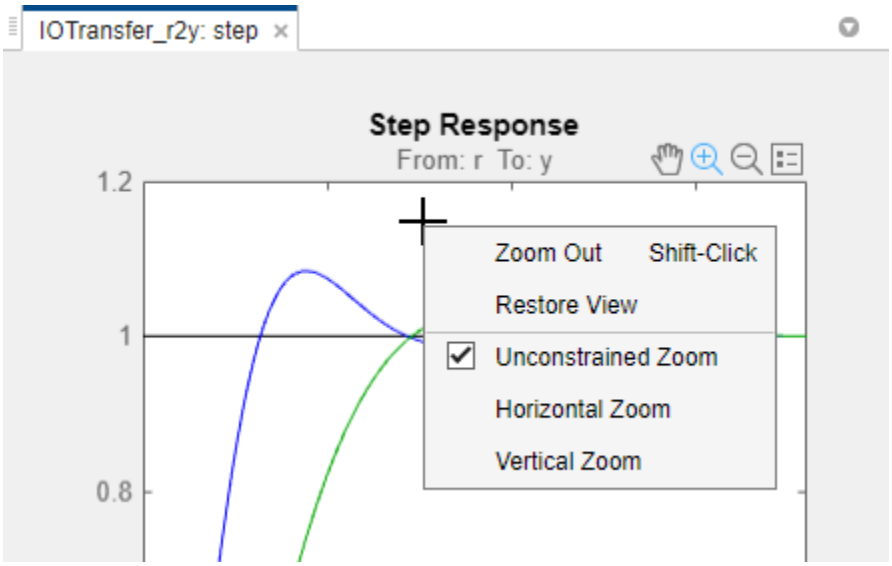


Plot Tools

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.

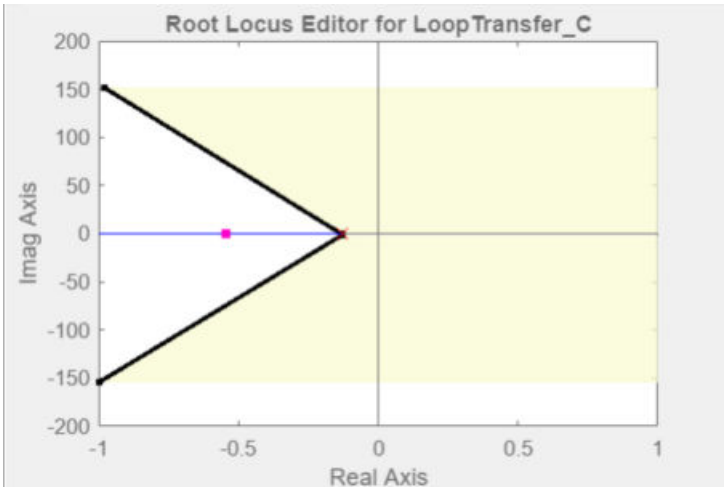


- — Pan. Click to activate, and drag the cursor across the plot area to pan. The pan icon turns dark when pan is active. Right-click while pan is active to access additional pan options. Click the icon again to deactivate.
- — Legend. By default, the plot legend is inactive. To toggle the legend on and 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.

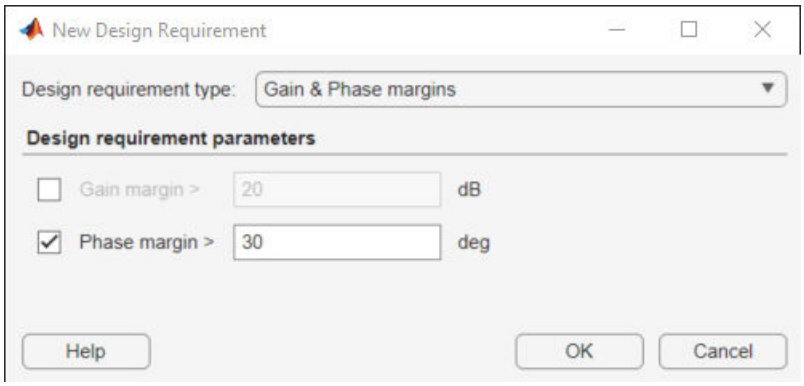
Design Requirements

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” on page 12-9.

See Also

More About

- “Control System Designer Tuning Methods” on page 12-5
- “Compare Performance of Multiple Designs” on page 12-101
- “Design Requirements” on page 12-9

Compare Performance of Multiple Designs

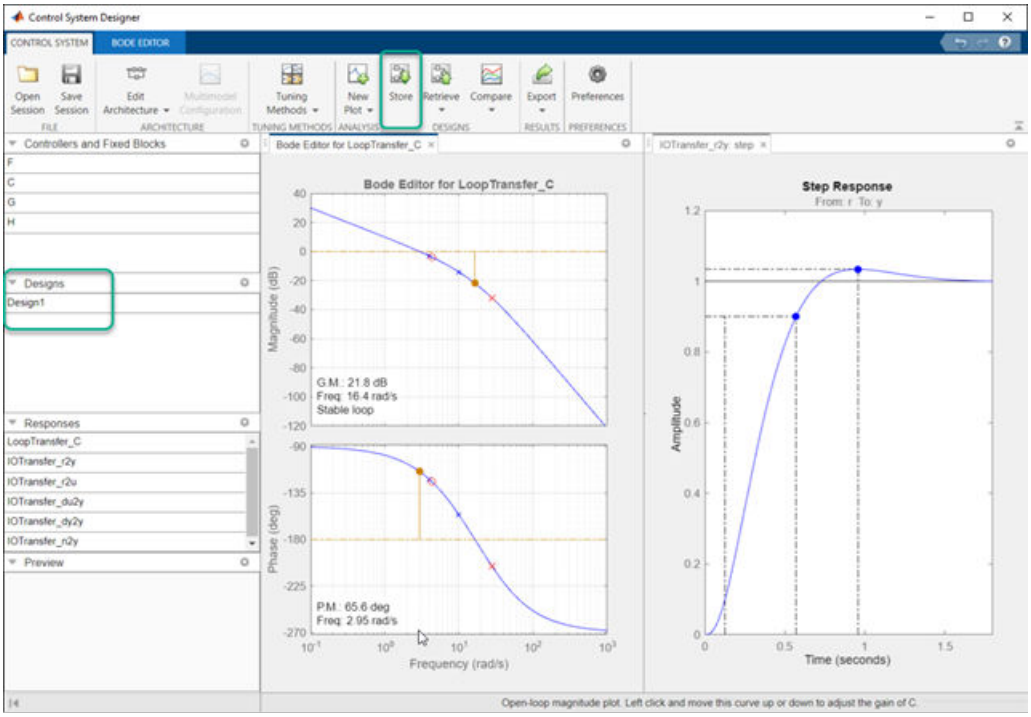
This example shows how to compare the performance of two different control system designs. Such comparison is useful, for example, to see the effects of different tuning methods or compensator structures.

Store First Design

In this example, the first design is the compensator tuned graphically in “Bode Diagram Design” on page 12-42.

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, click the design, and specify a new name.

Compute New Design

On the **Control System** tab, tune the compensator using a different tuning method.

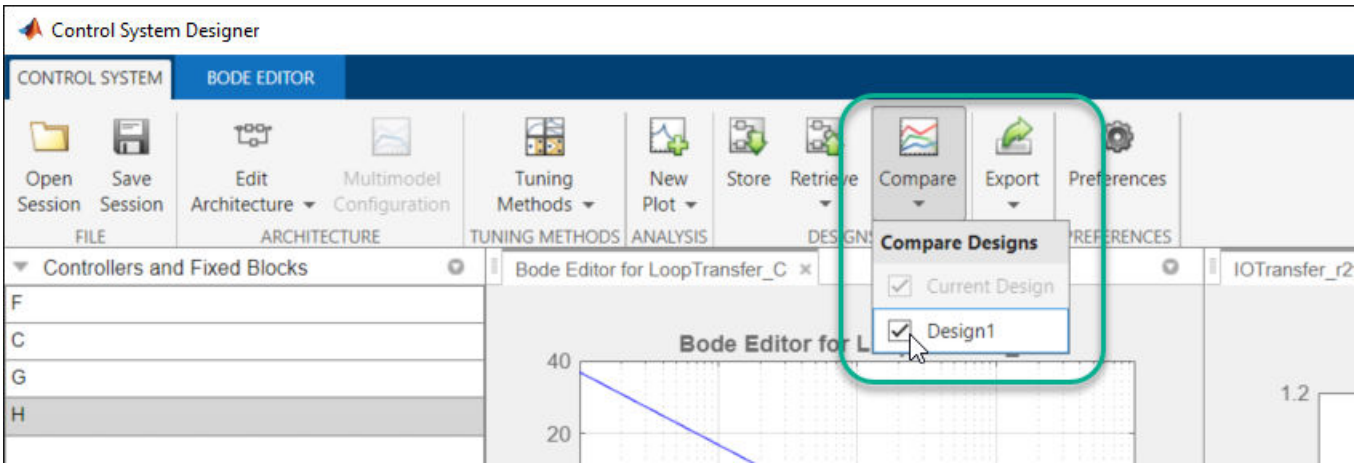
Under **Tuning Methods**, select `PID Tuning`.

To design a controller with the default `Robust response time` specifications, in the PID Tuning dialog box, click **Update Compensator**.

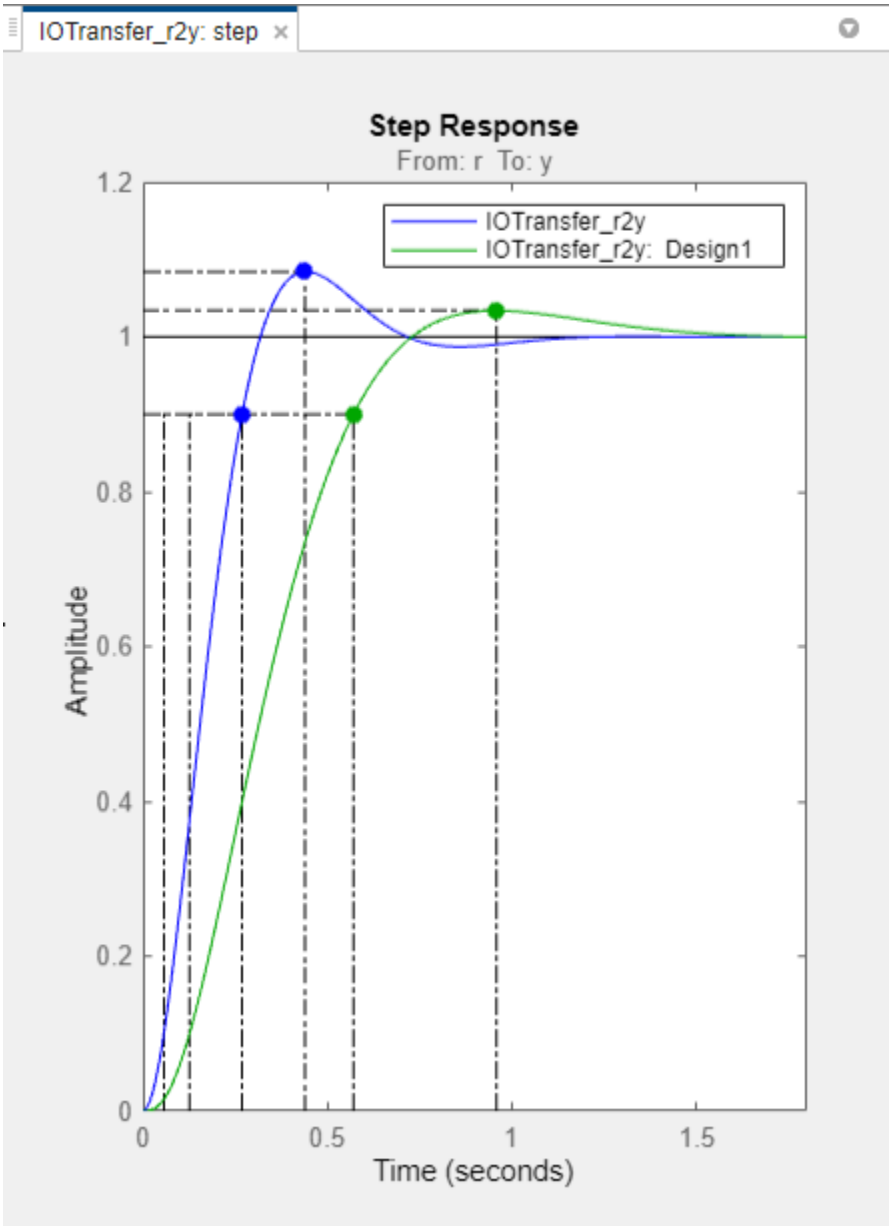
Compare New Design with Stored Design

Update all plots to reflect both the new design and the stored design.

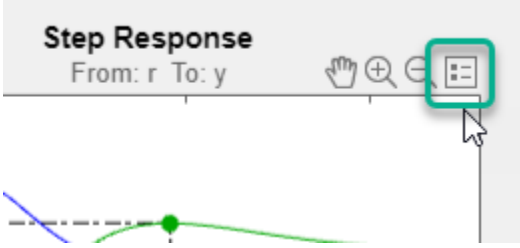
On the **Control System** tab, click **Compare**.



In the **Compare Designs** drop-down list, 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.



Tip To add the legend to the plot, hover over the plot and click the legend icon.

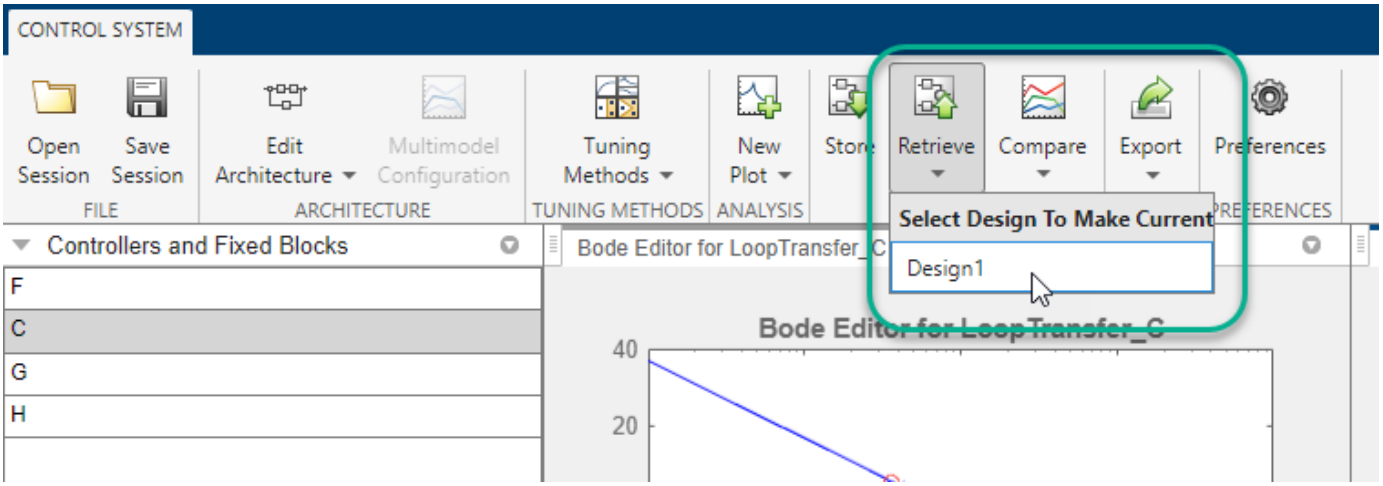


To compare a stored design with the current design, the sample times of the current design and stored design must be the same. To modify the sample time of the current design to match that of a stored design, on the **Control System** tab, select **Edit Architecture > Sample Time Conversion**. Then, in the Sample Time Conversion dialog box, specify the sample time and a rate conversion method for each block in the architecture.

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” (Simulink Control Design)). 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.

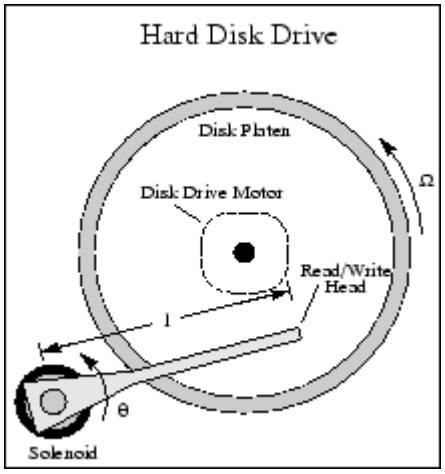
See Also

More About

- “Analyze Designs Using Response Plots” on page 12-95
- “Control System Designer Tuning Methods” on page 12-5

Design Hard-Disk Read/Write Head Controller

This example shows how to design a computer hard-disk read/write head position controller using classical control design methods.



Create Read/Write Head Model

Using Newton's laws, model the read/write head using the following differential equation:

$$J\frac{d^2\theta}{dt^2} + C\frac{d\theta}{dt} + K\theta = K_i i$$

Here,

- J is the inertia of the head assembly.
- C is the viscous damping coefficient of the bearings.
- K is the return spring constant.
- K_i is the motor torque constant.
- θ is the angular position of the head.
- i is the input current.

Taking the Laplace transform, the transfer function from i to θ is

$$H(s) = \frac{K_i}{Js^2 + Cs + K}.$$

Specify the physical constants of the model, such that:

- $J = 0.01\ kgm^2$
- $C = 0.004\ Nm/(rad/sec)$
- $K = 10\ Nm/rad$
- $K_i = 0.05\ Nm/rad$

```
J = 0.01;
C = 0.004;
K = 10;
Ki = 0.05;
```

Define the transfer function using these constants.

```
num = Ki;
den = [J C K];
H = tf(num,den)
```

```
H =

           0.05
  -----------------------
  0.01 s^2 + 0.004 s + 10

Continuous-time transfer function.
```

Discretize the Model

To design a digital controller that provides accurate positioning of the read/write head, first discretize the continuous-time plant.

Specify the sample time.

```
Ts = 0.005;
```

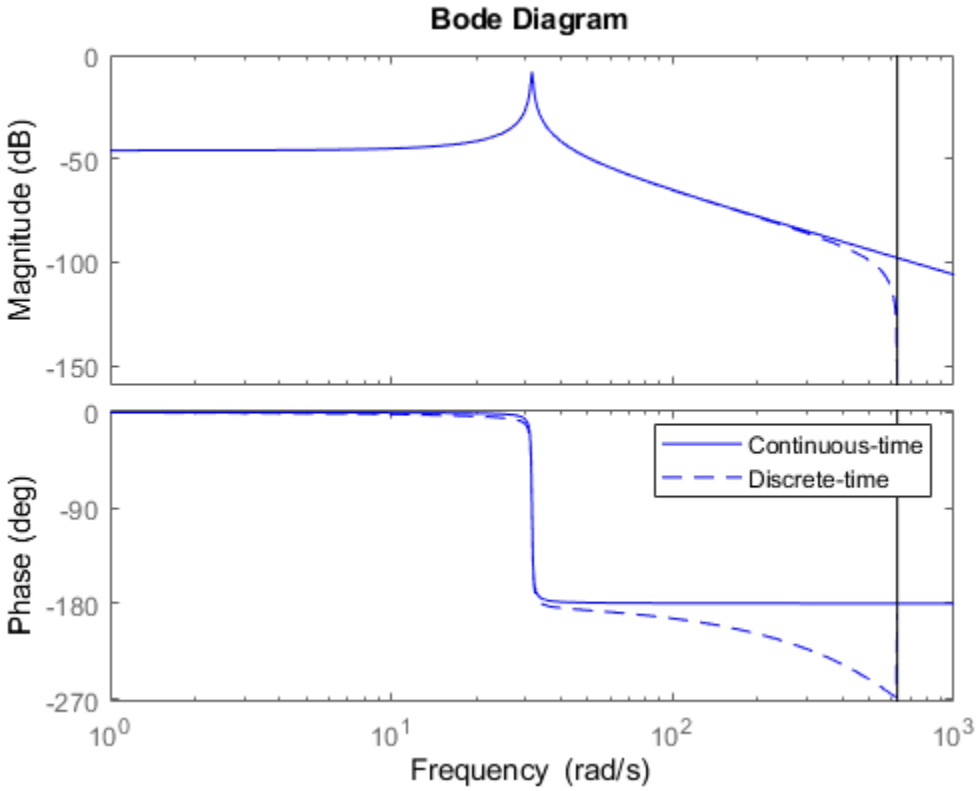
Discretize the model. Since the controller will have a digital-to-analog converter (with a zero-order hold) connected to its input, use the `c2d` command with the `'zoh'` discretization method.

```
Hd = c2d(H,Ts,'zoh')
```

```
Hd =

  6.233e-05 z + 6.229e-05
  -----------------------
   z^2 - 1.973 z + 0.998

Sample time: 0.005 seconds
Discrete-time transfer function.
```

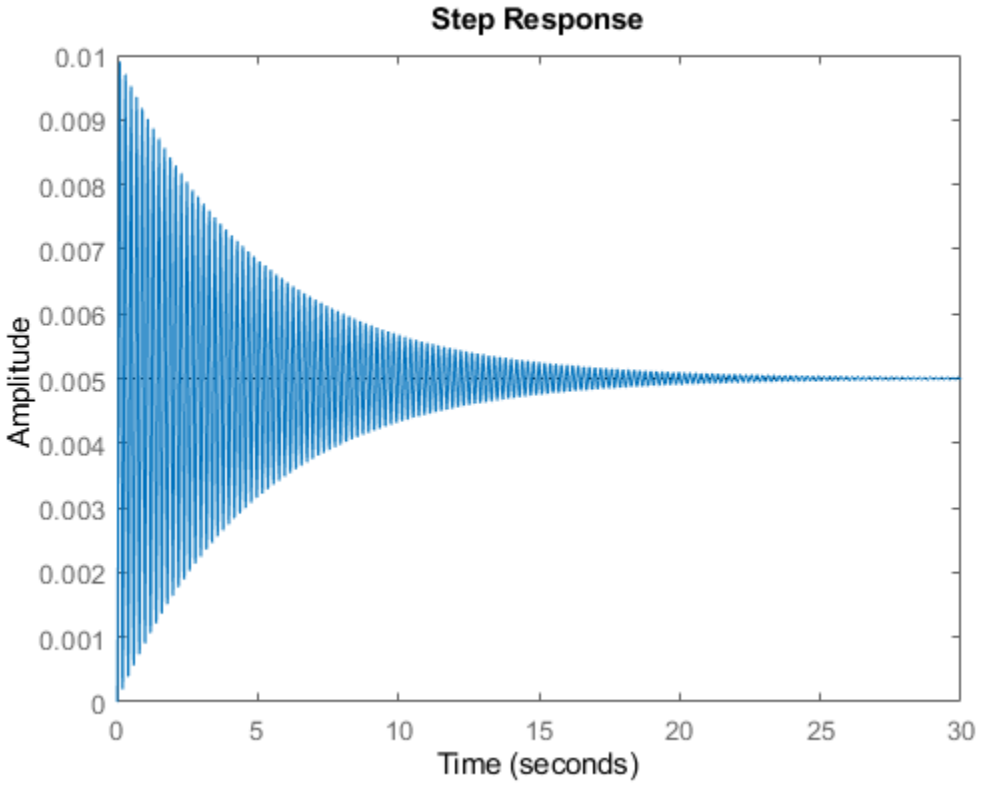
Compare the Bode plots of the continuous-time and discrete-time models.

```
bodeplot(H,'-',Hd,'--')
legend('Continuous-time','Discrete-time')
```



To analyze the discretized system, plot its step response.

```
stepplot(Hd)
```



The step response has significant oscillation, which is most likely due to light damping. Check the damping for the open-loop poles of the system.

```
damp(Hd)
```

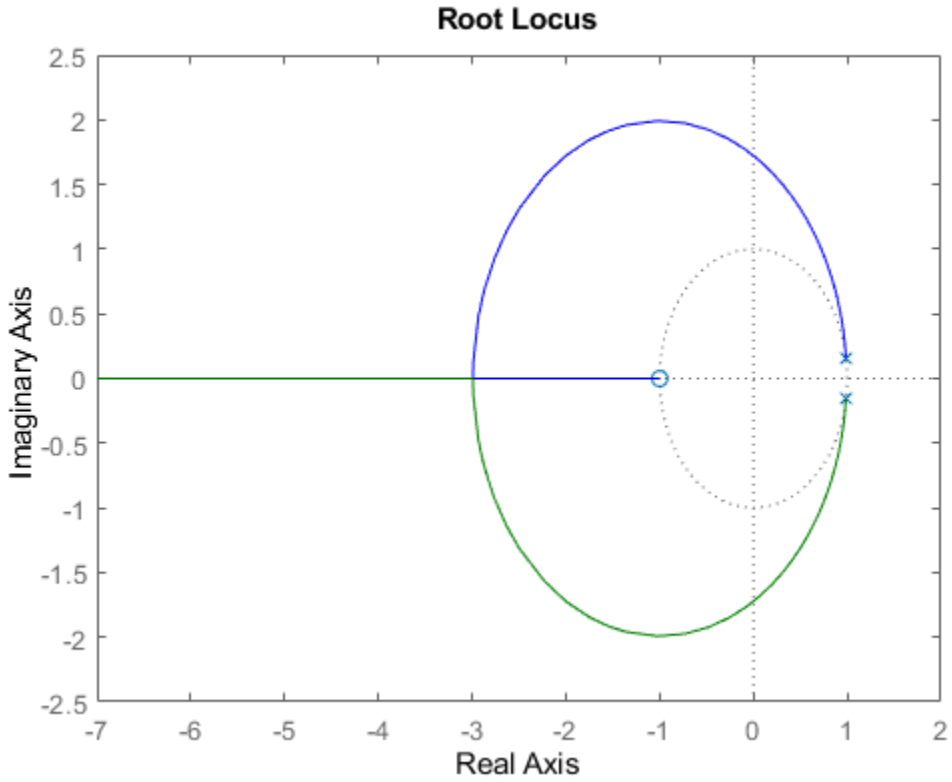
```
        Pole              Magnitude      Damping       Frequency      Time Constant
                                                     (rad/seconds)     (seconds)

  9.87e-01 + 1.57e-01i     9.99e-01     6.32e-03       3.16e+01         5.00e+00
  9.87e-01 - 1.57e-01i     9.99e-01     6.32e-03       3.16e+01         5.00e+00
```

As expected, the poles have light equivalent damping and are near the unit circle. Therefore, you must design a compensator that increases the damping in the system.

Add a Compensator Gain

The simplest compensator is a gain factor with no poles or zeros. Try to select an appropriate feedback gain using the root locus technique. The root locus plots the closed-loop pole trajectories as a function of the feedback gain.

```
rlocus(Hd)
```



The poles quickly leave the unit circle and go unstable. Therefore, you must introduce some lead to the system.

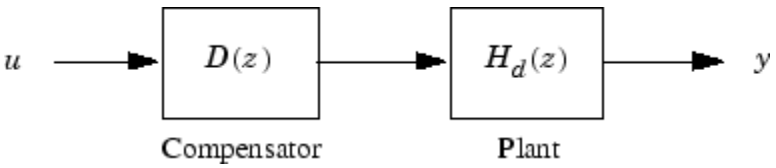
Add a Lead Network

Define a lead compensator with a zero at $\alpha = -0.85$ and a pole at $b = 0$.

$$D(z) = \frac{z+\alpha}{z+b}$$

```
D = zpk(0.85,0,1,Ts);
```

The corresponding open-loop model is the series connection of the compensator and plant.



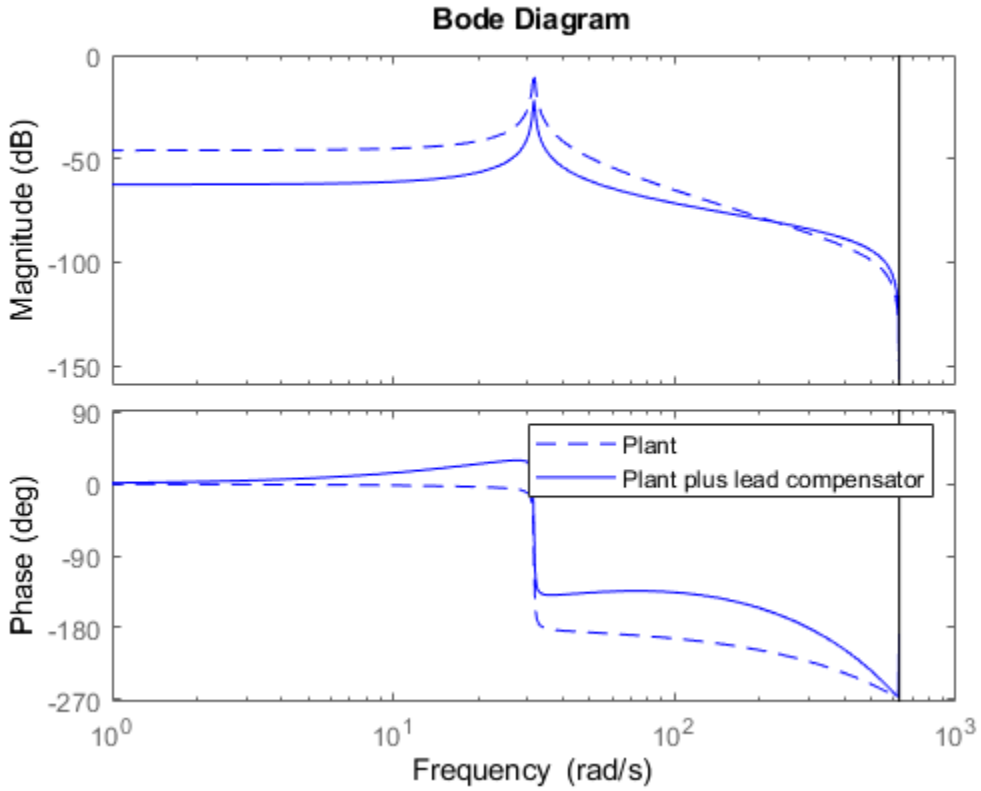

```
oloop = Hd*D
```

```
oloop =

  6.2328e-05 (z+0.9993) (z-0.85)
  ------------------------------
```

```
     z (z^2 - 1.973z + 0.998)

Sample time: 0.005 seconds
Discrete-time zero/pole/gain model.
```

To see how the lead compensator affects the open-loop frequency response, compare the Bode plots of `Hd` and `oloop`.

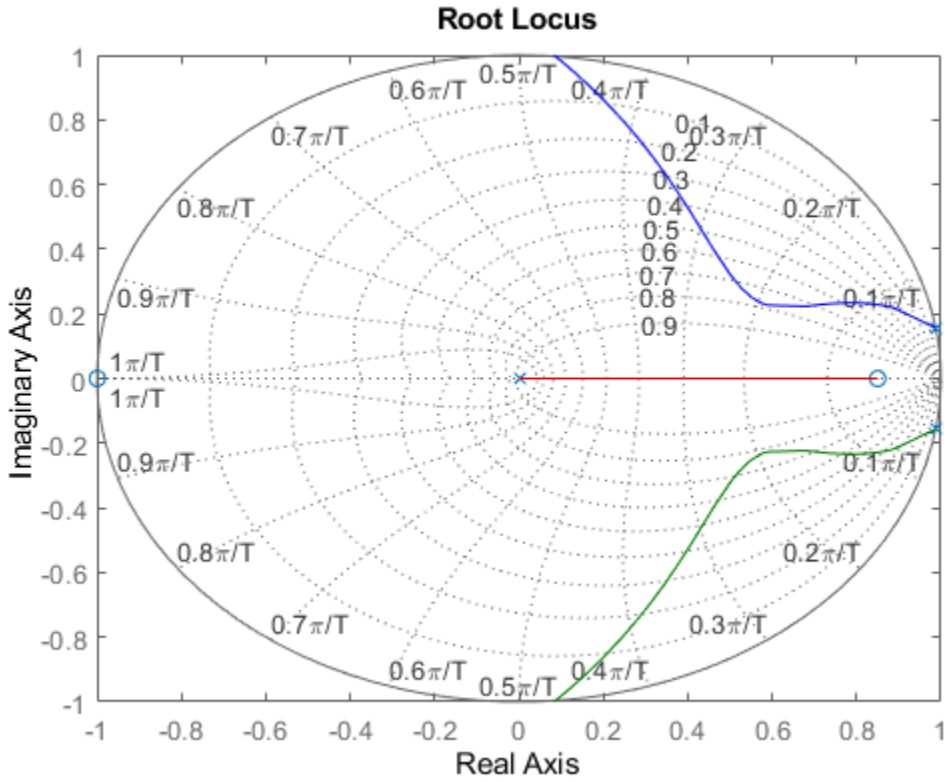
```
bodeplot(Hd,'--',oloop,'-')
legend('Plant','Plant plus lead compensator')
```



The compensator adds lead to the system, which shifts the phase response upward in the frequency range $\omega > 10$.

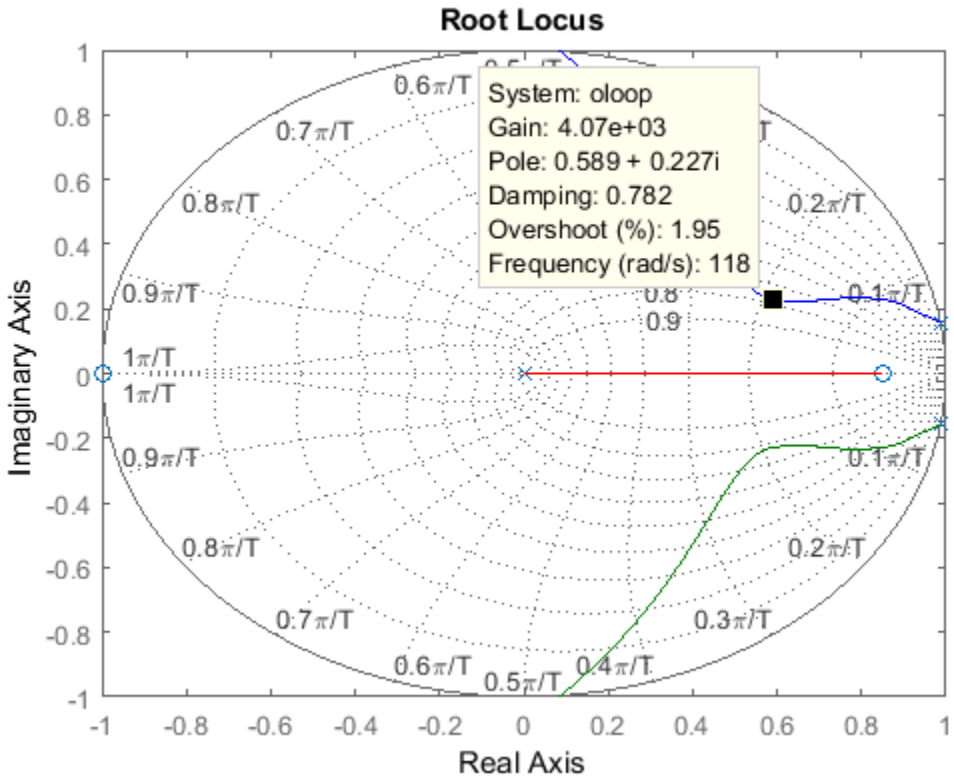
Examine the behavior of the closed-loop system poles using a root locus plot. Set the limits of both the `x`-axis and `y`-axis from `-1` to `1`.

```
rlocus(oloop)
zgrid
xlim([-1 1])
ylim([-1 1])
```



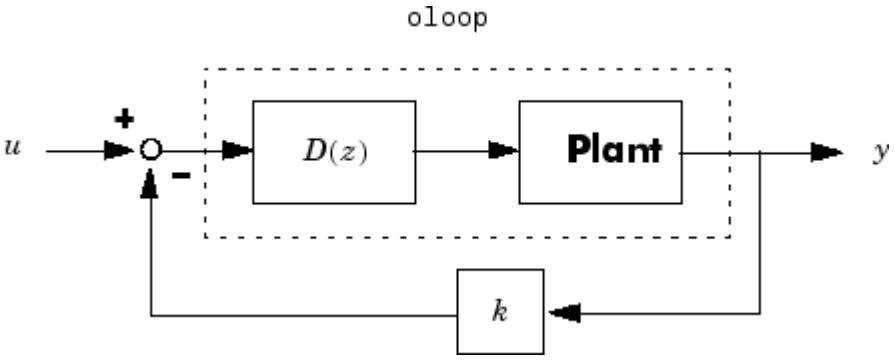
The closed-loop poles now remain within the unit circle for some time.

To create a data marker for the plot, click the root locus curve. Find the point on the curve where the damping is greatest by dragging the marker. The maximum damping of 0.782 corresponds to a feedback gain of `4.07e+03`.



Analyze Design

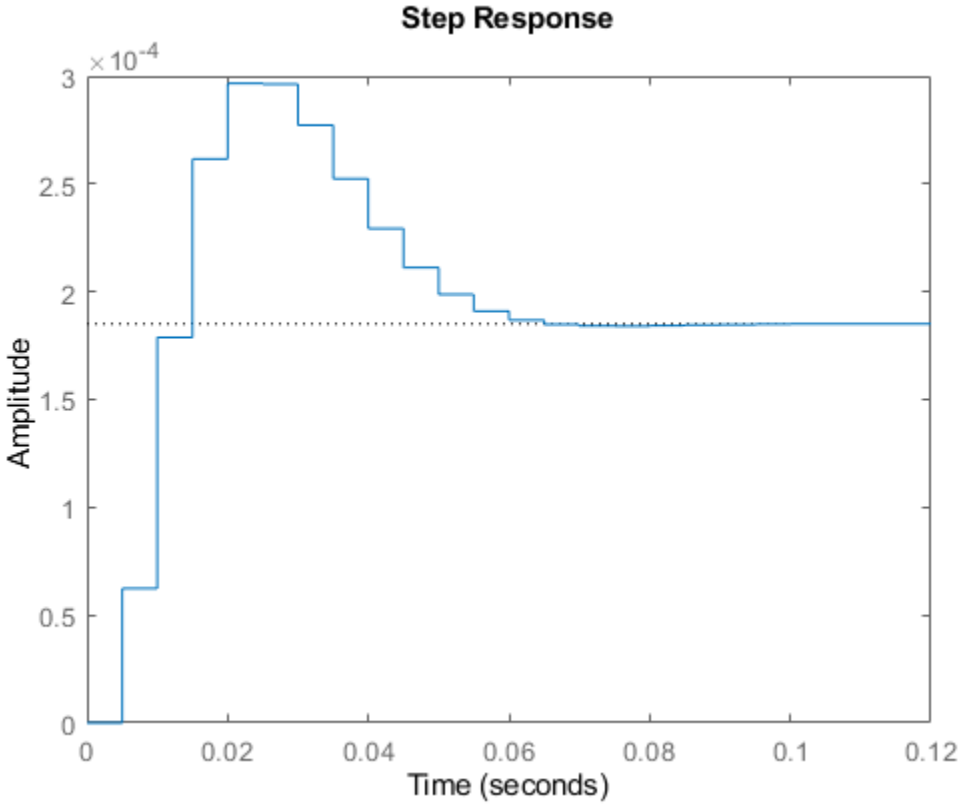
To analyze this design, first define the closed-loop system, which consists of the open-loop system with a feedback gain of `4.07e+03`.




```
k = 4.07e+03;
cloop = feedback(oloop,k);
```

Plot the closed-loop step response.

```
stepplot(cloop)
```



This response depends on your closed-loop setpoint. The one shown here is relatively fast and settles in about 0.06 seconds. Therefore, the closed-loop disk drive system has a seek time of 0.06 seconds. While this seek time is relatively slow by modern standards, you also started with a lightly-damped system.

It is good practice to examine the robustness of your design. To do so, compute the gain and phase margins for your system. First, form the unity feedback open-loop system by connecting the compensator, plant, and feedback gain in series.

```
olk = k*oloop;
```

Next, compute the margins for this open-loop model.

```
[Gm,Pm,Wcg,Wcp] = margin(olk)
```

```
Gm =

    3.8360


Pm =

   43.3068


Wcg =
```

```
  296.7976

Wcp =

  105.4680
```

This command returns the gain margin, `Gm`, the phase margin `Pm`, and their respective cross-over frequencies, `Wcg` and `Wcp`.

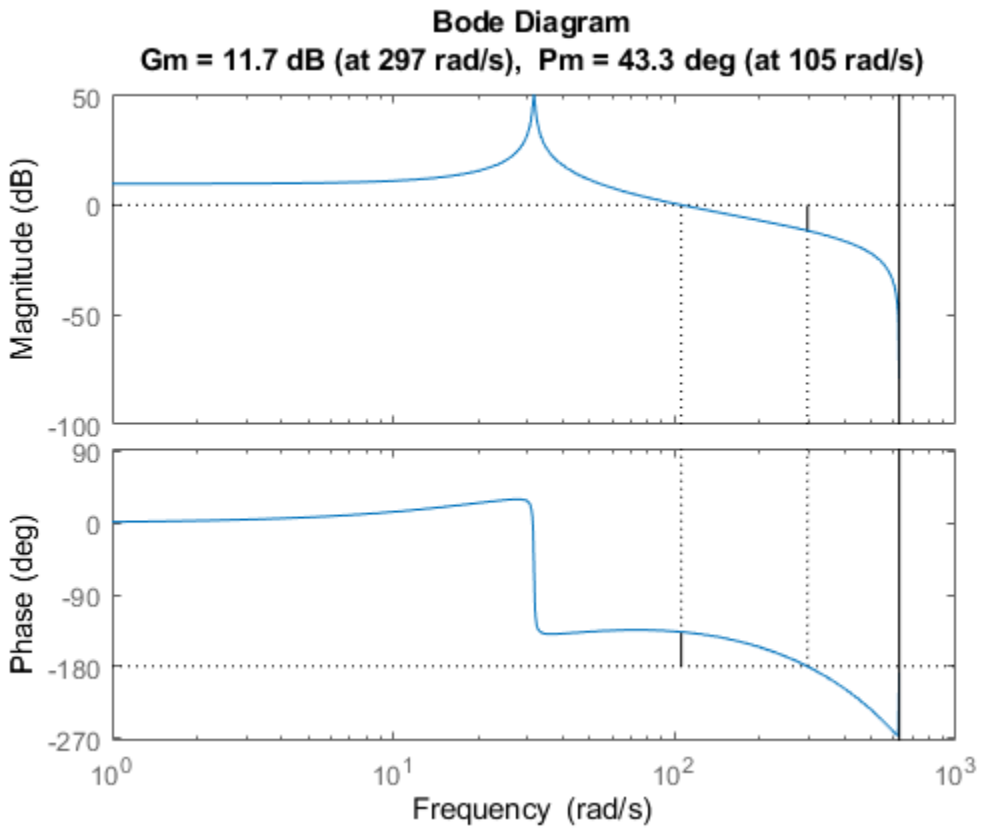
Convert the gain margin to dB.

```
20*log10(Gm)
```

```
ans =

   11.6775
```

You can also display the margins graphically.

```
margin(olk)
```



This design is robust and can tolerate an 11-dB gain increase or a 40-degree phase lag in the open-loop system without going unstable. By continuing this design process, you may be able to find a compensator that stabilizes the open-loop system and reduces the seek time further.

See Also

`rlocus` | `bodeplot` | `feedback` | `margin`

Design Compensator for Plant Model with Time Delays

This example shows how to design a compensator for a plant with time delays using Control System Designer.

Analysis and Design of Feedback Systems with Time Delays

When working with time delay systems it is advantageous to work with analysis and design tools that directly support time delays so that performance and stability can be evaluated exactly. However, many control design techniques and algorithms cannot directly handle time delays. A common workaround consists of replacing delays by their Pade approximations (all-pass filters). Because this approximation is only valid at low frequencies, it is important to choose the right approximation order and check the approximation validity.

Control System Designer provides a variety of design and analysis tools. Some of these tools support time delays exactly while others support time delays indirectly through approximations. Use these tools to design compensators for your control system and visualize the compromises made when using approximations.

Plant Model

For this example, which uses a unity feedback configuration, the plant model has a time delay:

$$G(s) = e^{-0.5s}\frac{1}{s+1}$$

Create the plant model.

```
G = tf(1,[1,1],'InputDelay',0.5);
```

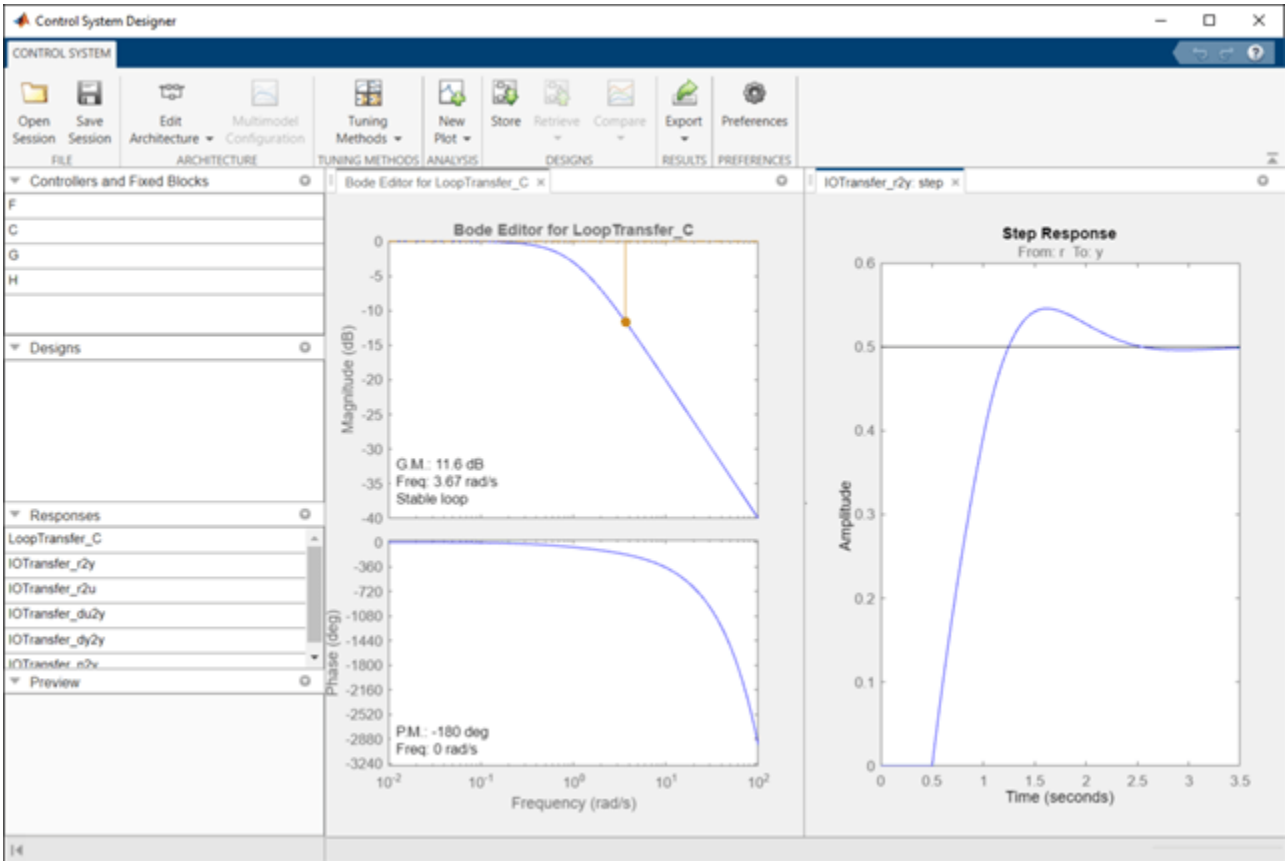
Tools That Support Time Delays

In the app, the following tools support time delays directly:

- Bode and Nichols Editors
- Time Response Plots
- Frequency Response Plots

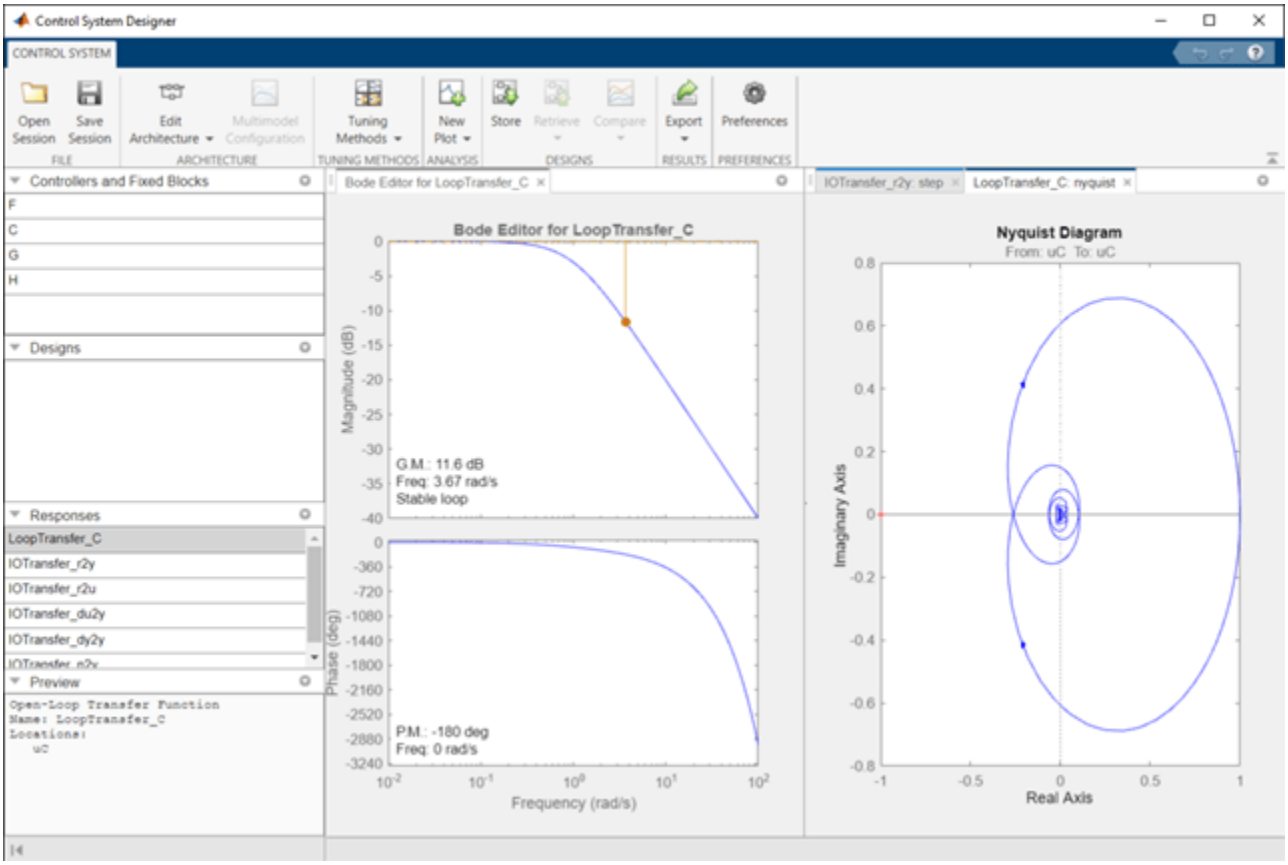
Open Control System Designer, importing the plant model and using a Bode editor configuration.

```
controlSystemDesigner({'bode'},G)
```



The phase response of the Bode plot shows the roll-off effect from the exact representation of the delay. The beginning of the step response shows an exact representation of the 0.5 second delay.

Open a Nyquist plot of the open-loop response. In the data browser, right-click `LoopTransfer_C`, and select **Plot > nyquist**.



The Nyquist response wrapping around the origin in a spiral fashion is the result of the exact representation of the time delay.

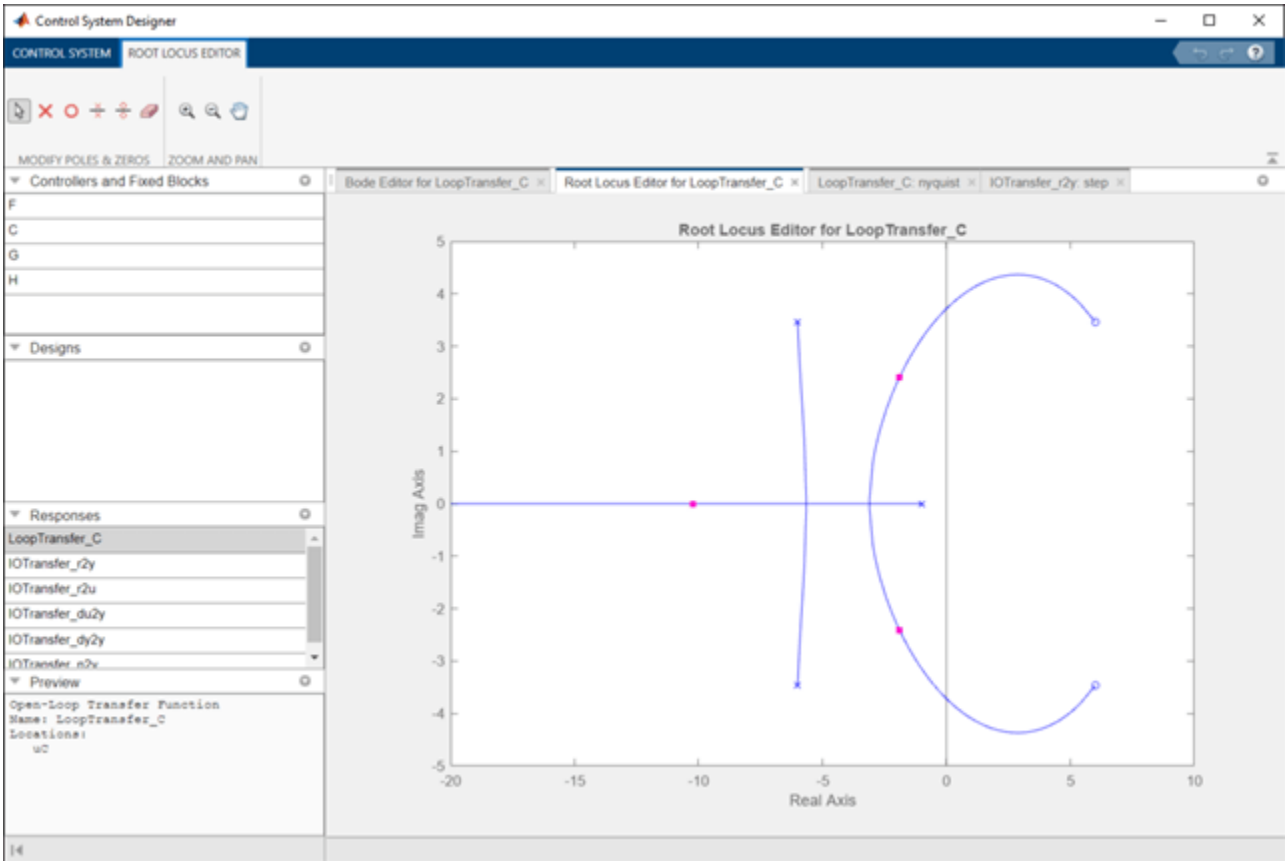
Tools That Approximate Time Delays

In the app, the following tools approximate time delays:

- Root Locus Editor
- Pole/Zero Plots
- Many of the automated tuning methods

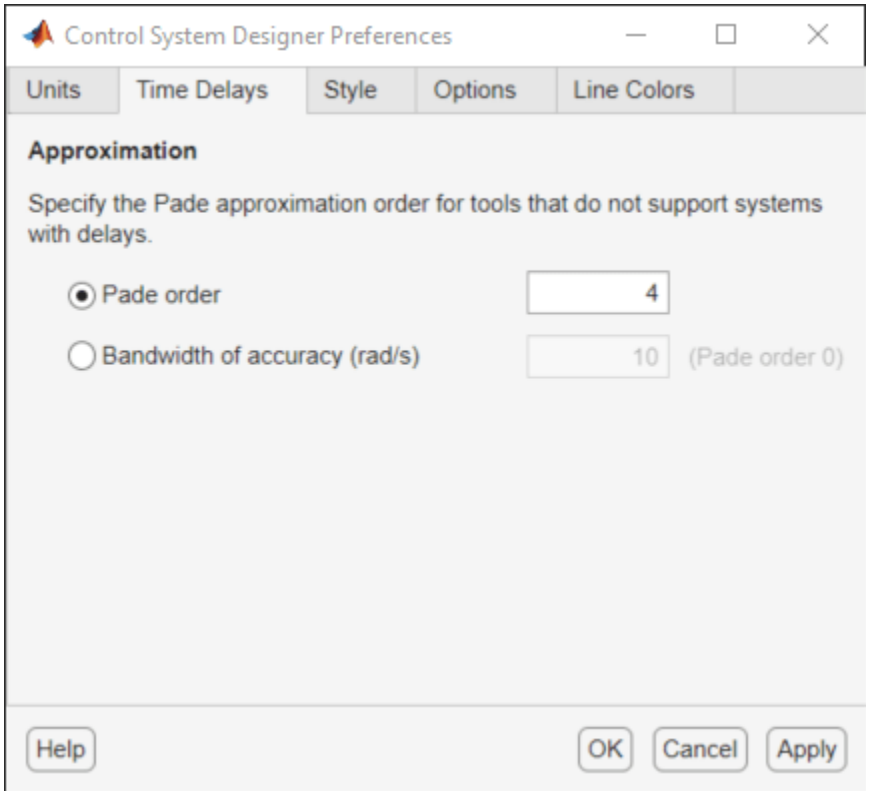
When using approximations, the results are not exact and depend on the validity of the approximation. Each tool in Control System Designer provides a warning pane to indicate when time-delays are approximated.

Open a root locus editor plot for the open-loop response. Click **Tuning Methods**, and select **Root Locus Editor**. In the Select Response to Edit dialog box, click **Plot**.

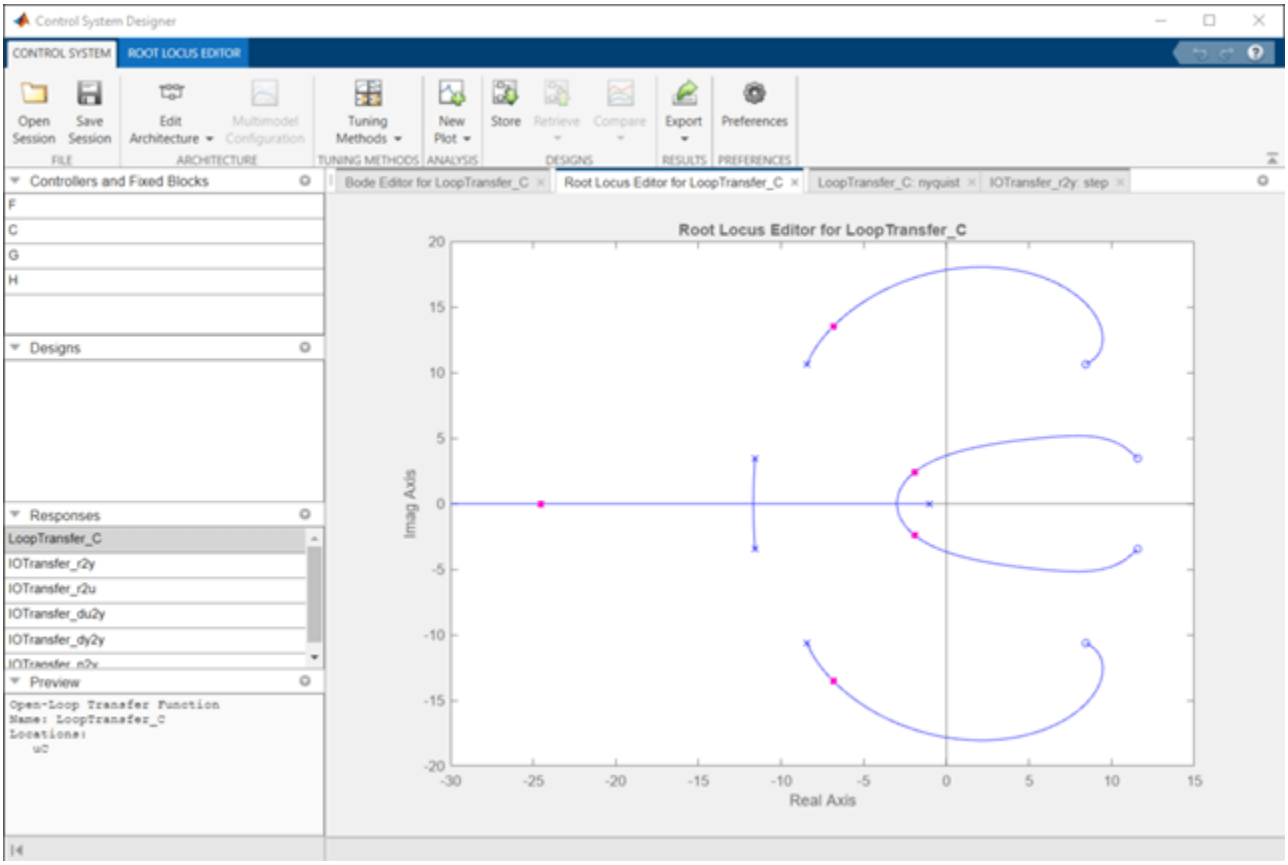


Change Approximation Settings

To change the approximation settings, click the hyperlink in the notification. In the Control System Designer Preferences dialog box, on the **Time Delays** tab, specify a **Pade order** of 4. Alternatively, you can set the bandwidth over which you want the approximation to be accurate.



The higher-order Pade approximation adds poles and zeros to the root locus plot.



See Also

Control System Designer

More About

- “Control System Designer Tuning Methods” on page 12-5
- “Analyze Designs Using Response Plots” on page 12-95

Design Compensator for Systems Represented by Frequency Response Data

This example shows how to design a compensator for a plant model defined by frequency response data (FRD) using **Control System Designer**.

Acquire Frequency Response Data (FRD) Plant Model

Nonparametric representations of plant models, such as frequency response data, are often used for analysis and control design. These FRD models are typically obtained from:

1) Signal analyzer hardware that performs frequency domain measurements on systems.

2) Nonparametric estimation techniques using the systems time response data. You can use the following products to estimate FRD models.

Simulink® Control Design™:

- Function: `frestimate` (Simulink Control Design)
- Example: “Frequency Response Estimation Using Simulation-Based Techniques” (Simulink Control Design).

Signal Processing Toolbox™:

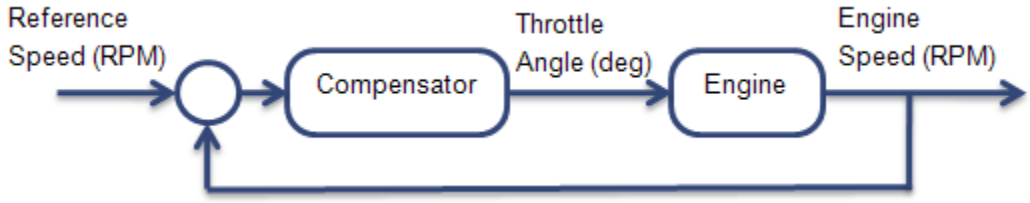
- Function: `tfestimate` (Signal Processing Toolbox).

System Identification Toolbox™:

- Functions: `etfe` (System Identification Toolbox), `spa` (System Identification Toolbox), `spafdr` (System Identification Toolbox)

FRD Model and Design Requirements

In this example, design an engine speed controller that actuates the engine throttle angle:



The frequency response of the engine is already estimated. Load and view the data.

```
load FRDPlantDemoData.mat
AnalyzerData
```

```
AnalyzerData = struct with fields:
          Response: [594x1 double]
         Frequency: [594x1 double]
    FrequencyUnits: 'rad/s'
```

Create an FRD model object:

```
FRDPlant = frd(AnalyzerData.Response,AnalyzerData.Frequency,...
    'Unit',AnalyzerData.FrequencyUnits);
```

The design requirements are:

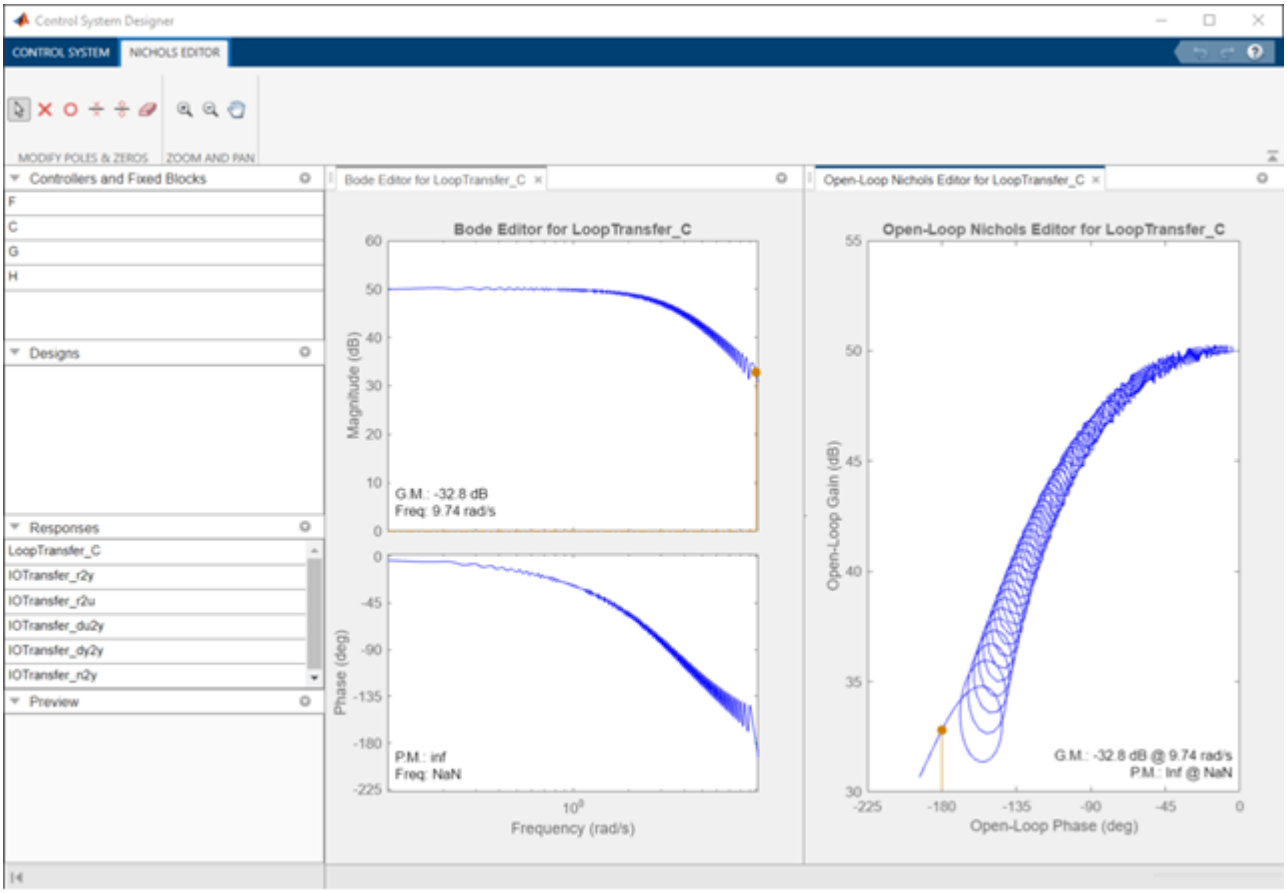
- Zero steady-state error for step reference speed changes
- Phase margin greater than 60 degrees
- Gain margin greater than 20 dB.

Design Compensator

Open **Control System Designer**.

```
controlSystemDesigner({'bode','nichols'},FRDPlant)
```

The app opens with both Bode and Nichols open-loop editors.



You can design the compensator by shaping the open-loop frequency response in either the Bode editor or Nichols editor. In these editors, interactively modify the gain, poles, and zeros of the compensator.

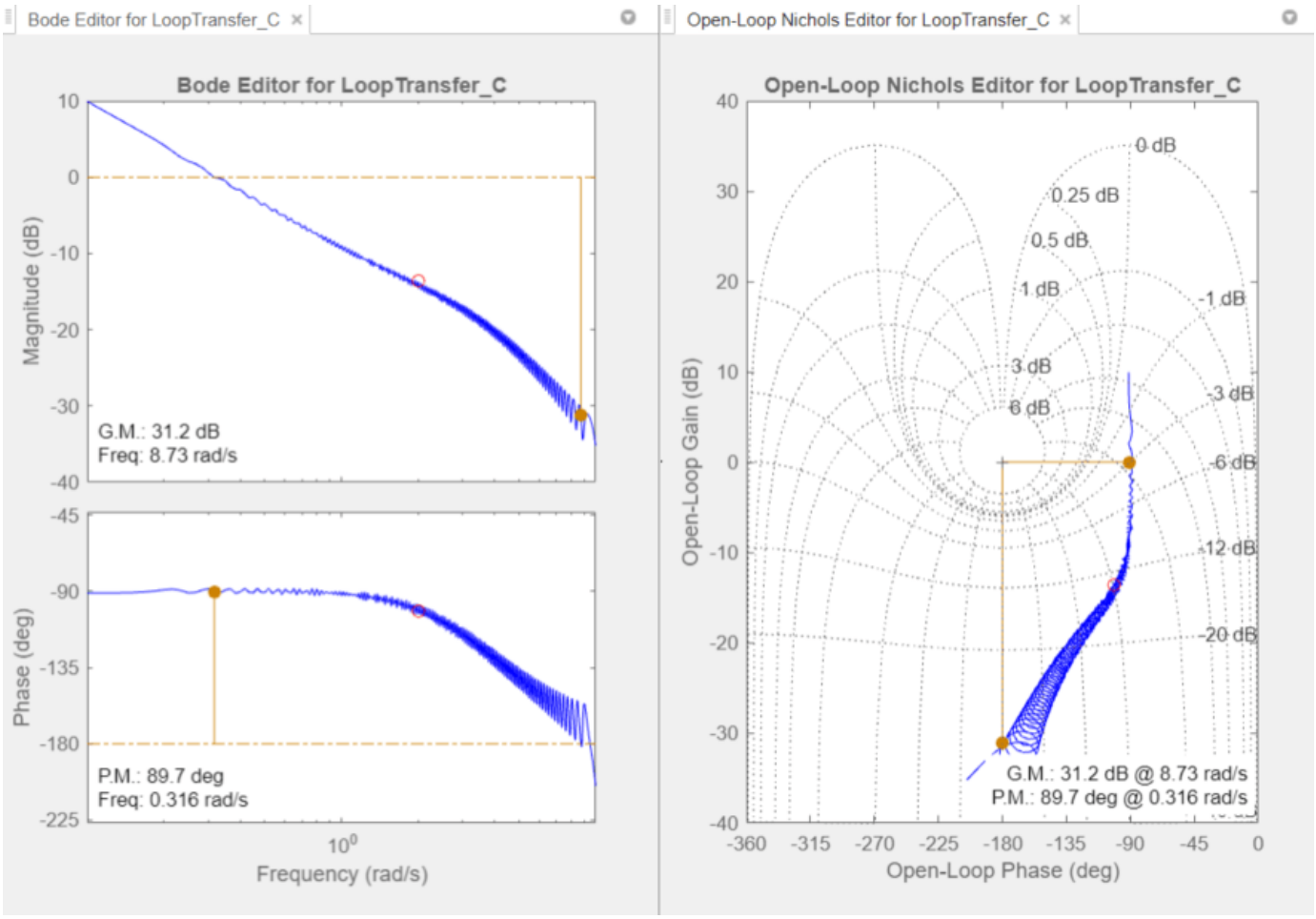
To satisfy the tracking requirement of zero steady-state error, add an integrator to the compensator. Right-click the Bode editor plot area, and select **Add Pole/Zero > Integrator**.

To meet the gain and phase margin requirements, add a zero to the compensator. Right-click the Bode editor plot area, and select **Add Pole/Zero > Real Zero**. Modify the location of the zero and the gain of the compensator until you satisfy the margin requirements.

One possible design that satisfies the design requirements is:

$$C(s) = \frac{0.001(s+4)}{s}.$$

This compensator design, which is a PI controller, achieves a 20.7 dB gain margin and a 70.8 degree phase margin.



Export the designed compensator to the workspace. Click **Export**.

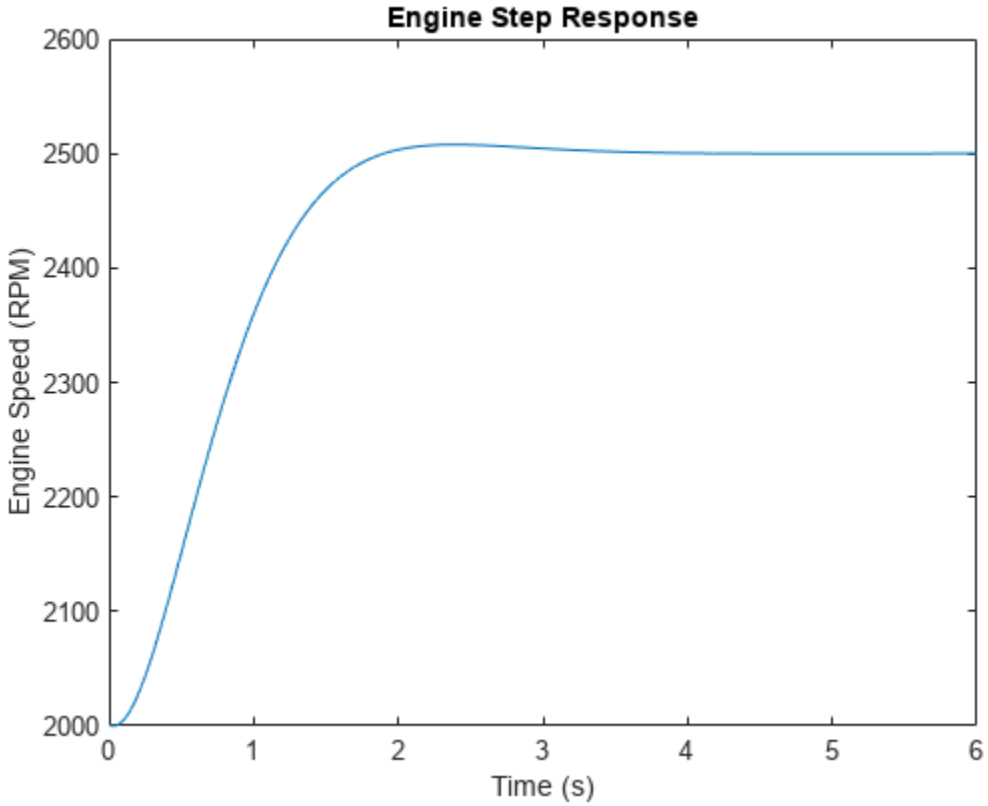
Validate the Design

Validate the controller performance by simulating the engine response using a nonlinear model in Simulink®. For this example, the validation simulation results are in `EngineStepResponse`.

Plot the response of the engine to a reference speed change from 2000 to 2500 RPM:

```
plot(EngineStepResponse.Time,EngineStepResponse.Speed)
title('Engine Step Response')
```

```
xlabel('Time (s)')
ylabel('Engine Speed (RPM)')
```



The response shows zero steady-state error and well-behaved transients with the following metrics.

```
stepinfo(EngineStepResponse.Speed,EngineStepResponse.Time)
```

```
ans = struct with fields:
         RiseTime: 0.7136
    TransientTime: 1.7194
     SettlingTime: 1.3738
      SettlingMin: 2.2505e+03
      SettlingMax: 2.5078e+03
        Overshoot: 0.3127
       Undershoot: 0
             Peak: 2.5078e+03
         PeakTime: 2.3853
```

See Also

Control System Designer

More About

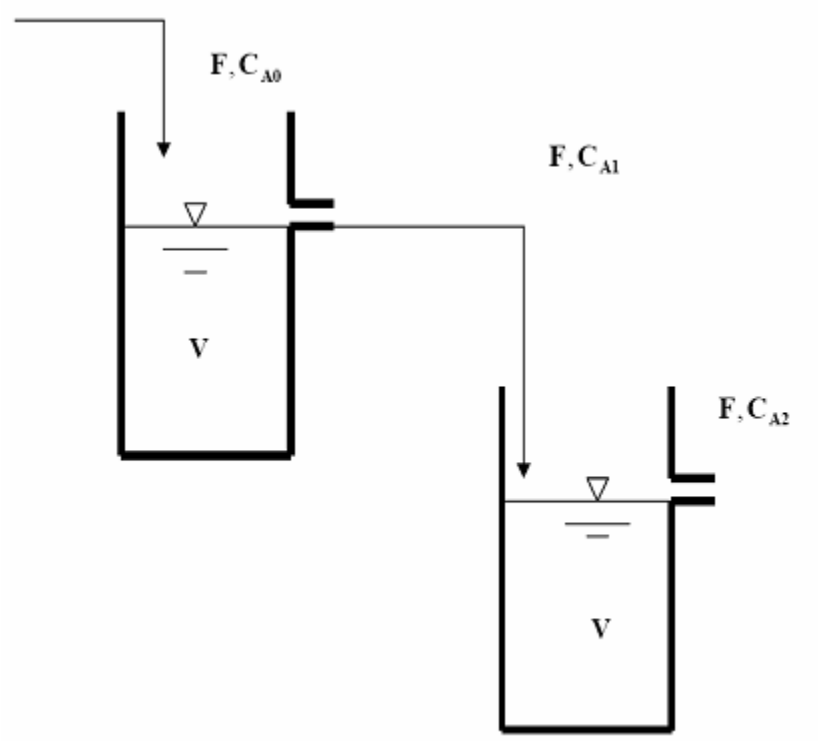
- “Control System Designer Tuning Methods” on page 12-5
- “Analyze Designs Using Response Plots” on page 12-95

Design Internal Model Controller for Chemical Reactor Plant

This example shows how to design a compensator in an IMC structure for series chemical reactors, using **Control System Designer**. Model-based control systems are often used to track setpoints and reject load disturbances in process control applications.

Plant Model

The plant for this example is a chemical reactor system, comprised of two well-mixed tanks.



The reactors are isothermal and the reaction in each reactor is first order on component A:

$$r_A = -kC_A$$

Material balance is applied to the system to generate a dynamic model of the system. The tank levels are assumed to stay constant because of the overflow nozzle and hence there is no level control involved.

For details about this plant, see Example 3.3 in Chapter 3 of "Process Control: Design Processes and Control Systems for Dynamic Performance" by Thomas E. Marlin.

The following differential equations describe the component balances:

$$V\frac{dC_{A1}}{dt} = F(C_{A0} - C_{A1}) - VkC_{A1}$$

$$V\frac{dC_{A2}}{dt} = F(C_{A1} - C_{A2}) - VkC_{A2}$$

At steady state,

$$\frac{dC_{A1}}{dt} = 0$$

$$\frac{dC_{A2}}{dt} = 0$$

the material balances are:

$$F^*(C_{A0}^* - C_{A1}^*) - VkC_{A1}^* = 0$$

$$F^*(C_{A1}^* - C_{A2}^*) - VkC_{A2}^* = 0$$

where C_{A0}^*, $C_{A1}*$, and $C_{A2}*$ are steady-state values.

Substitute the following design specifications and reactor parameters:

- $F^* = 0.085\ mole/min$
- $C_{A0}^* = 0.925\ mol/min$
- $V = 1.05\ m^3$
- $k = 0.04\ min^{-1}$

The resulting steady-state concentrations in the two reactors are:

$$C_{A1}^* = KC_{A0}^* = 0.6191 mol/m^3$$

$$C_{A2}^* = K^2C_{A0}^* = 0.4144 mol/m^3$$

where

$$K = \frac{F^*}{F^* + Vk} = 0.6693$$

For this example, design a controller to maintain the outlet concentration of reactant from the second reactor, C_{A2}^*, in the presence of any disturbance in feed concentration, C_{A0}. The manipulated variable is the molar flowrate of the reactant, F, entering the first reactor.

Linear Plant Models

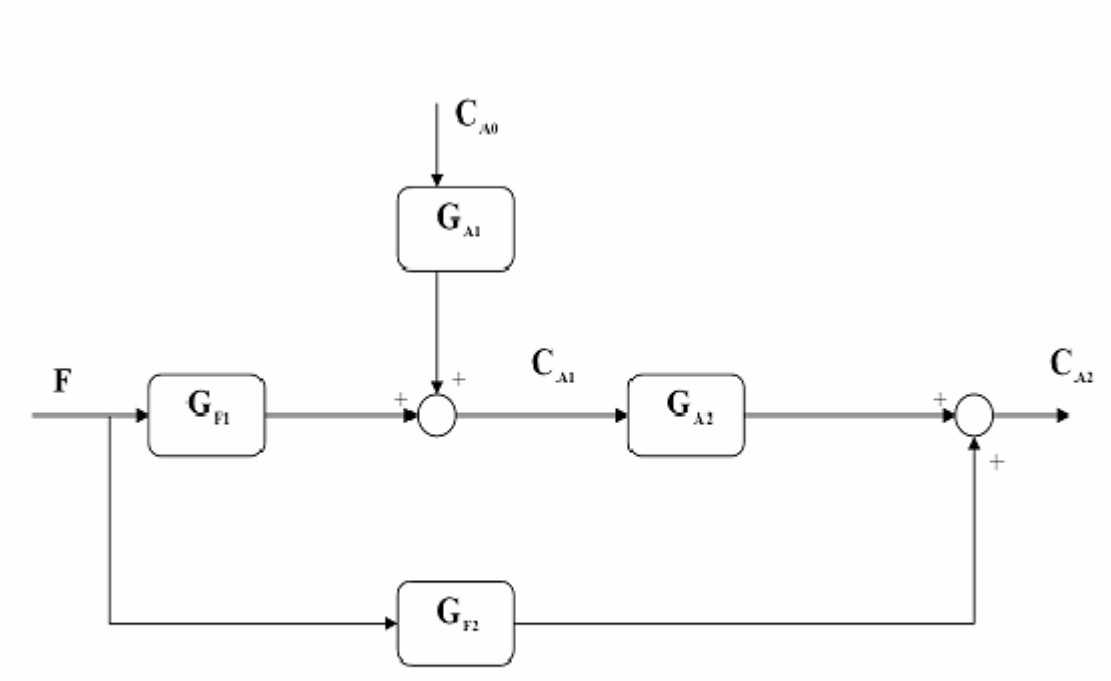
In this control design problem, the plant model is

$$\frac{C_{A2}(s)}{F(s)}$$

and the disturbance model is

$$\frac{C_{A0}(s)}{C_{A2}(s)}$$

This chemical process can be represented using the following block diagram:



where

$$G_{A1} = \frac{C_{A1}(s)}{C_{A0}(s)} = \frac{0.6693}{8.2677s + 1}$$

$$G_{F1} = \frac{C_{A1}(s)}{F(s)} = \frac{2.4087}{8.2677s + 1}$$

$$G_{A2} = \frac{C_{A2}(s)}{C_{A1}(s)} = \frac{0.6693}{8.2677s + 1}$$

$$G_{F2} = \frac{C_{A2}(s)}{F(s)} = \frac{1.6118}{8.2677s + 1}$$

Based on the block diagram, obtain the plant and disturbance models as follows:

$$\frac{C_{A2}(s)}{F(s)} = G_{F1}G_{A2} + G_{F2} = \frac{13.3259s + 3.2239}{(8.2677s + 1)^2}$$

$$\frac{C_{A2}}{C_{A0}} = G_{A1}G_{A2} = \frac{0.4480}{(8.2677s + 1)^2}$$

Create the plant model at the command line:

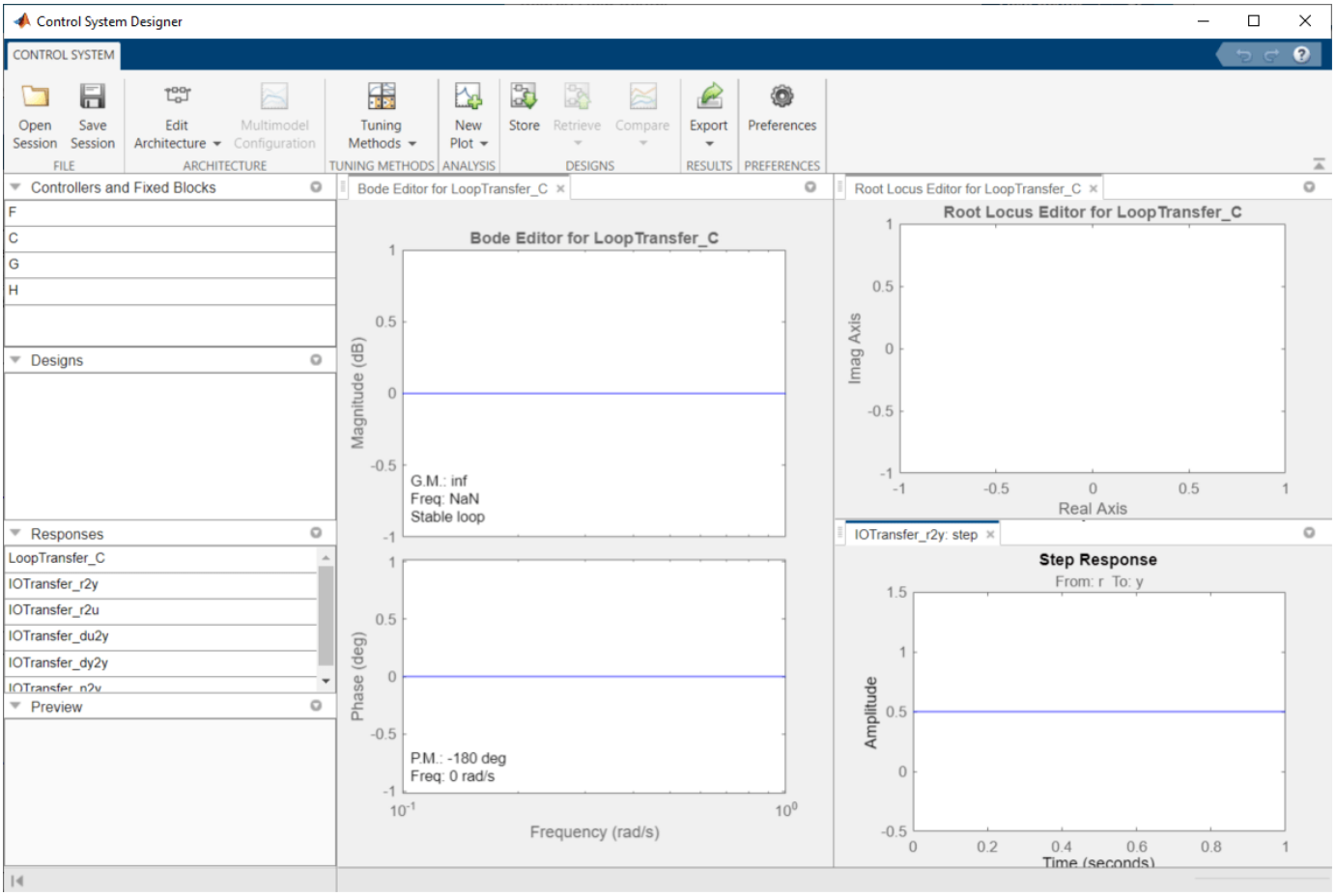
```
s = tf('s');
G1 = (13.3259*s+3.2239)/(8.2677*s+1)^2;
G2 = G1;
Gd = 0.4480/(8.2677*s+1)^2;
```

G1 is the real plant used in controller evaluation. G2 is an approximation of the real plant and it is used as the predictive model in the IMC structure. `G2 = G1` means that there is no model mismatch. `Gd` is the disturbance model.

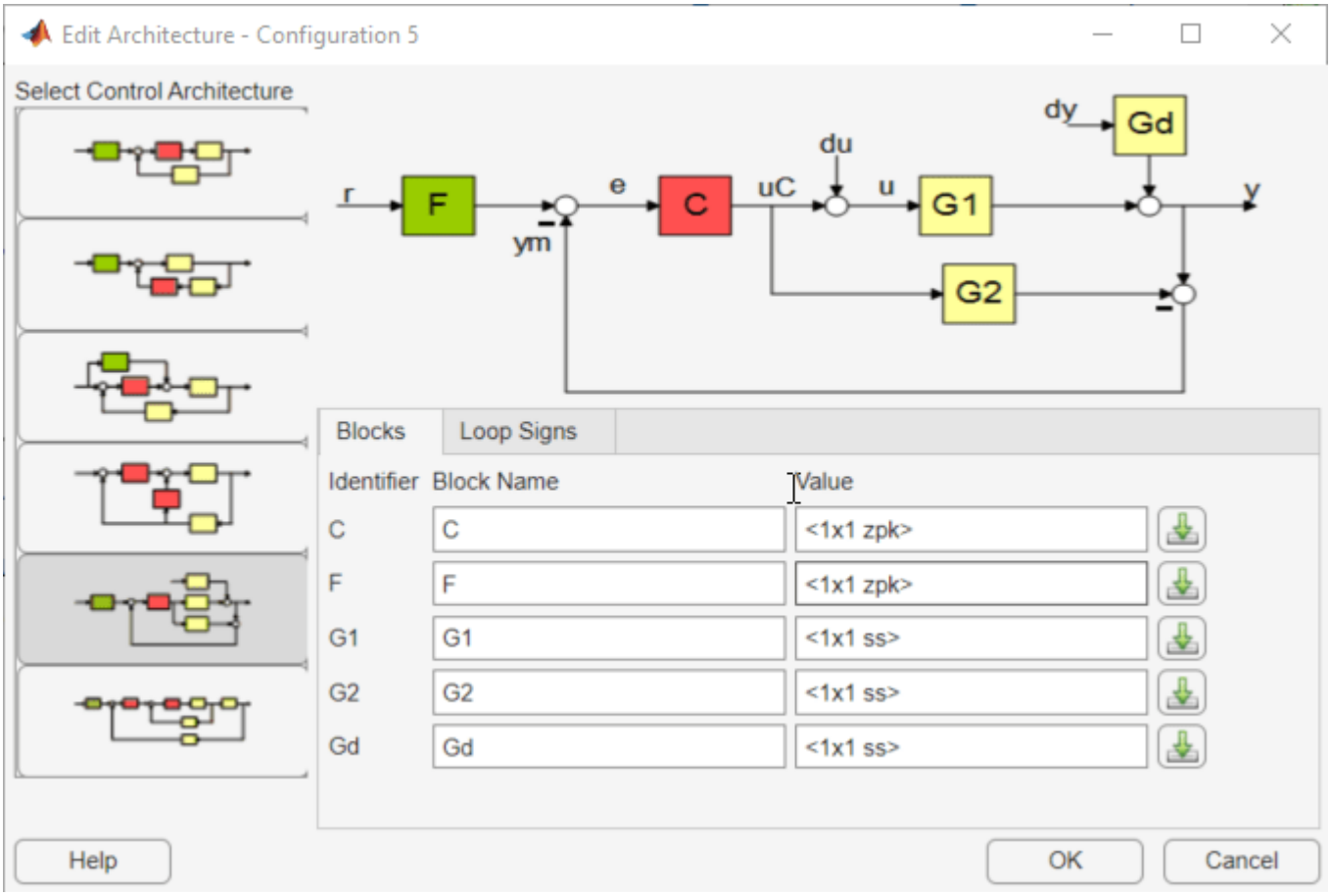
Define IMC Structure in Control System Designer

Open **Control System Designer**.

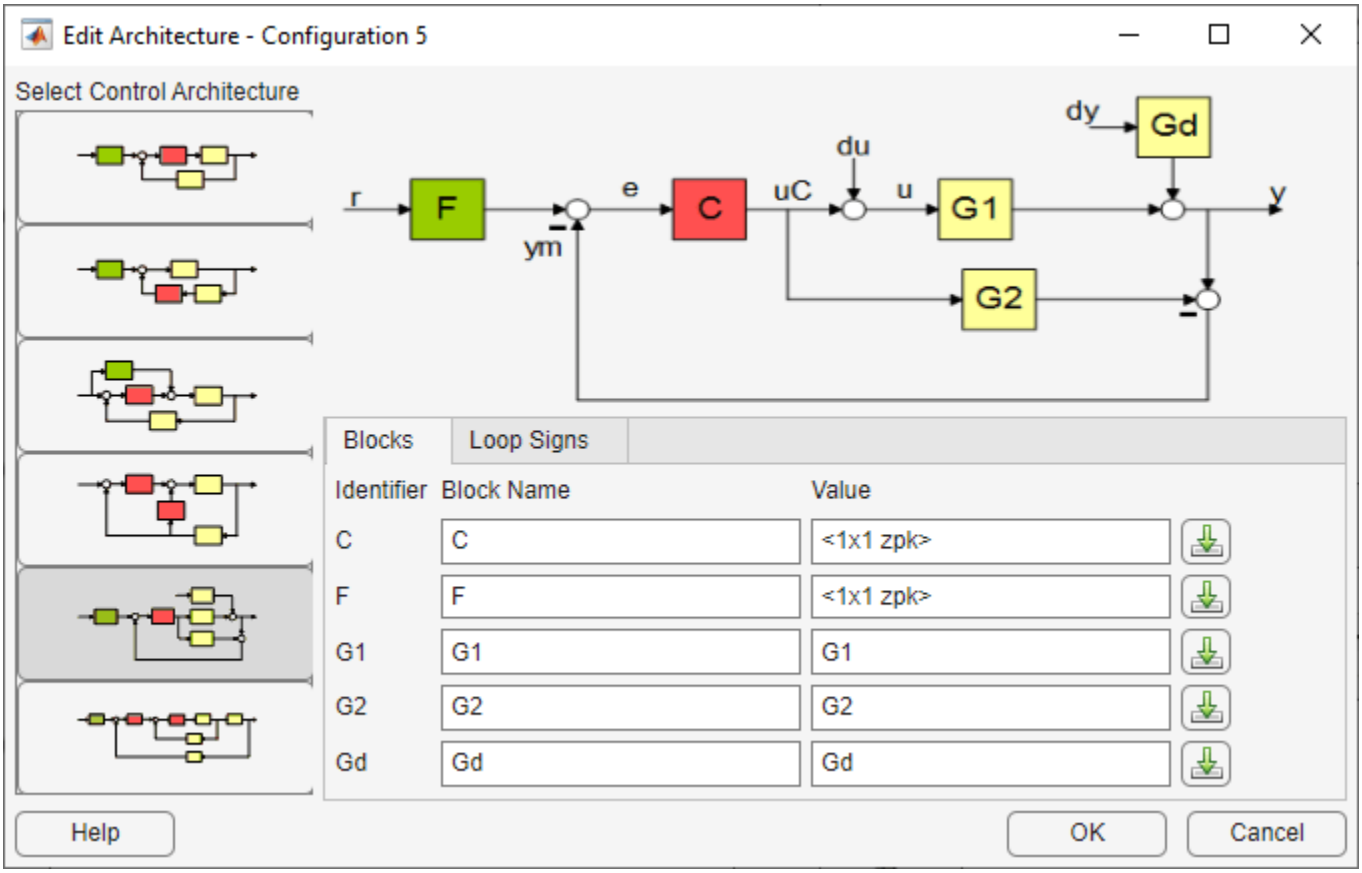
```
controlSystemDesigner
```



Select the IMC control architecture. In **Control System Designer**, click **Edit Architecture**. In the Edit Architecture dialog box, select Configuration 5.



Load the system data. For **G1**, **G2**, and **Gd**, specify a model **Value**.

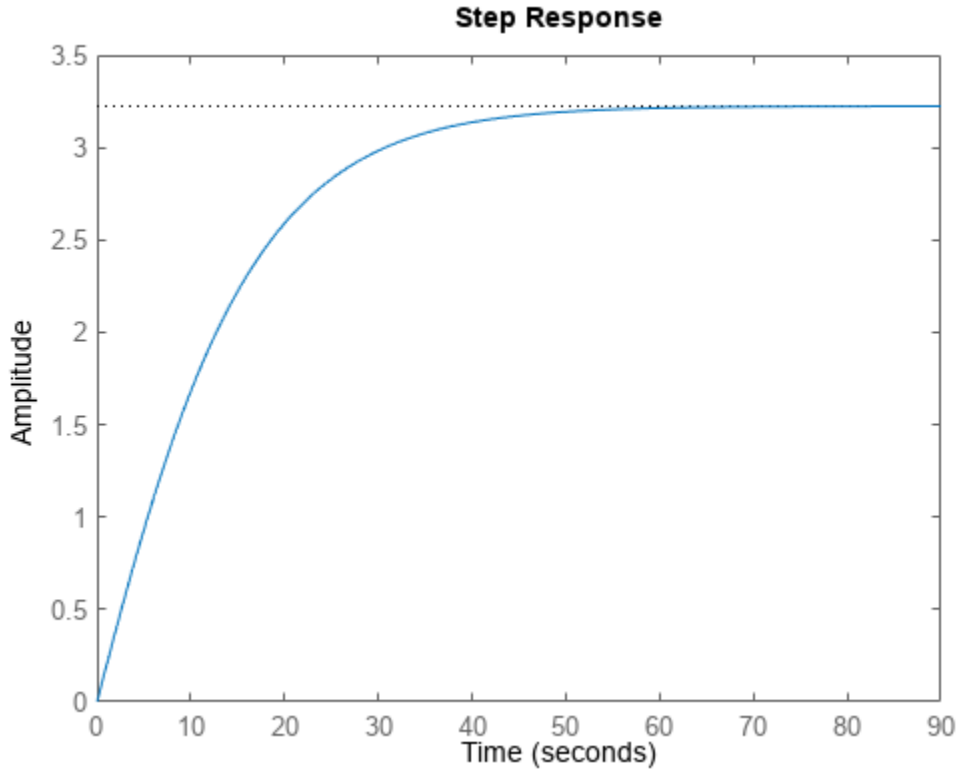


Click **OK**.

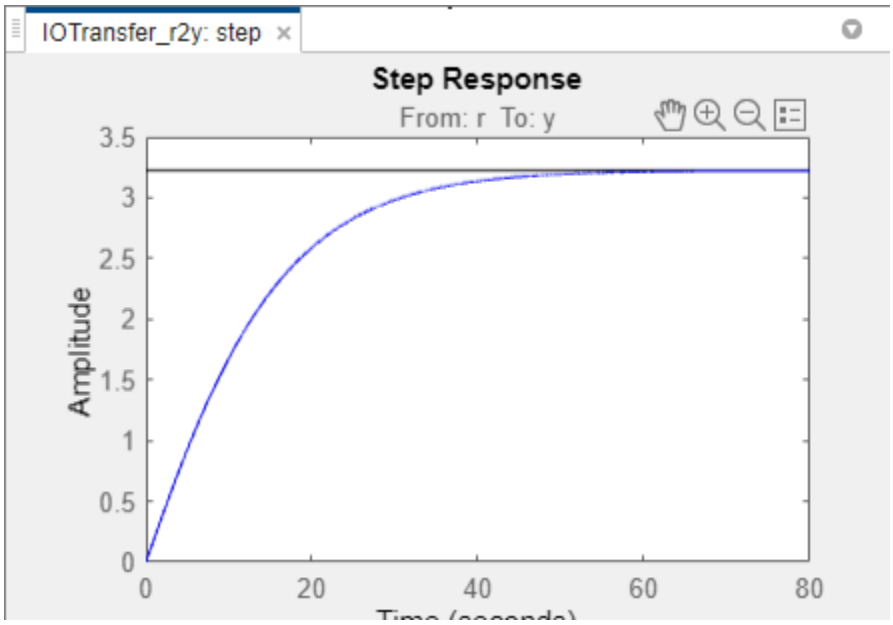
Tune Compensator

Plot the open-loop step response of `G1`.

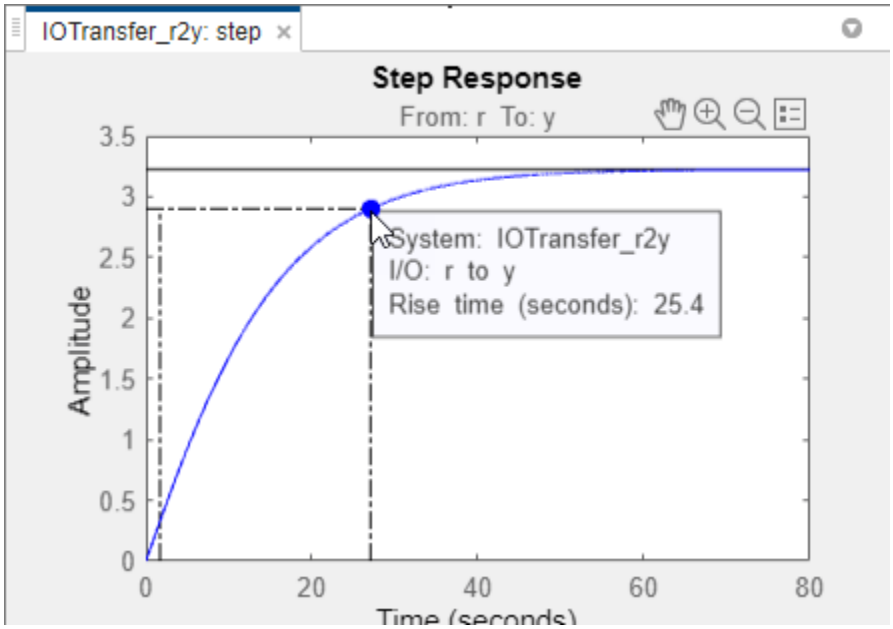
```
step(G1)
```



When **Control System Designer** it shows the closed-loop step response of the system.

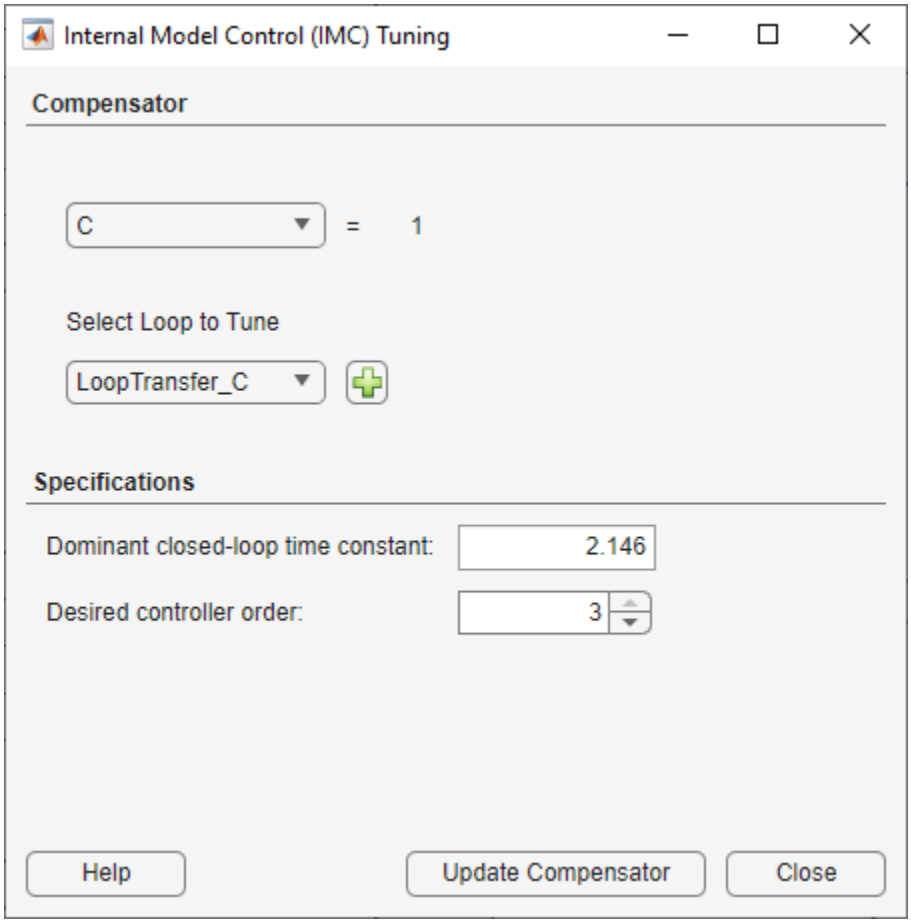


Right-click the plot and select **Characteristics > Rise Time** submenu. Hover your mouse over the rise time marker.

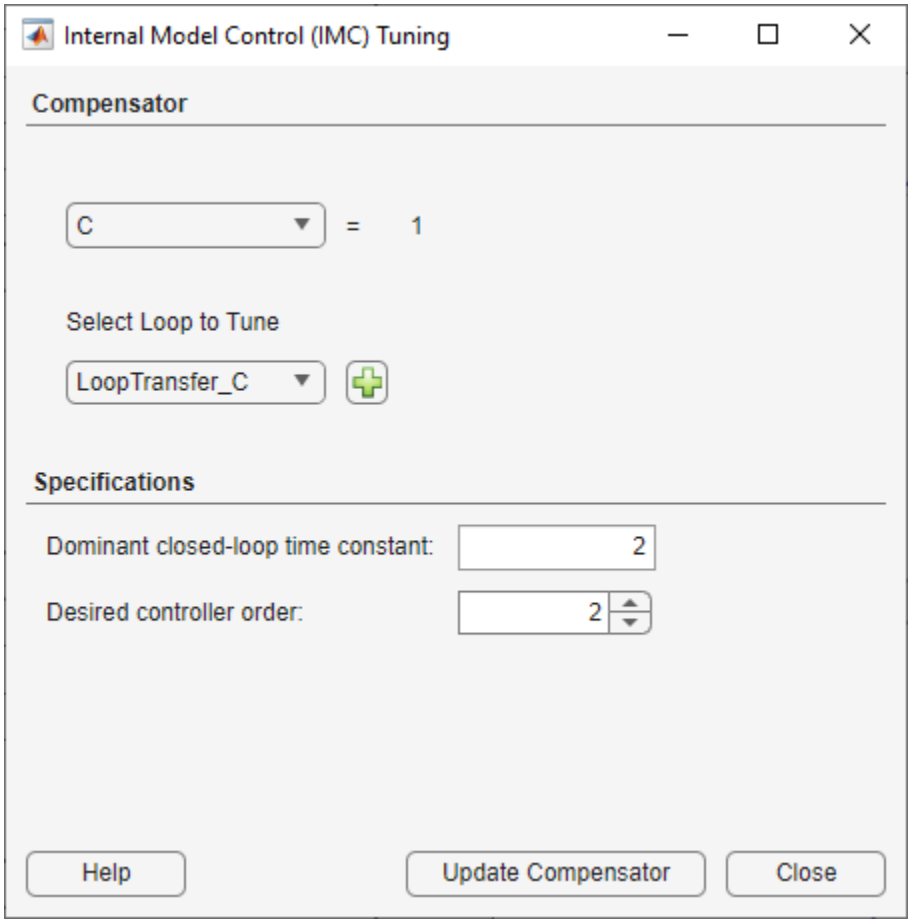


The rise time is about 25 seconds and you want to tune the IMC compensator to achieve a faster closed-loop response time.

To tune the IMC compensator, in **Control System Designer**, click **Tuning Methods**, and select **Internal Model Control (IMC) Tuning**.

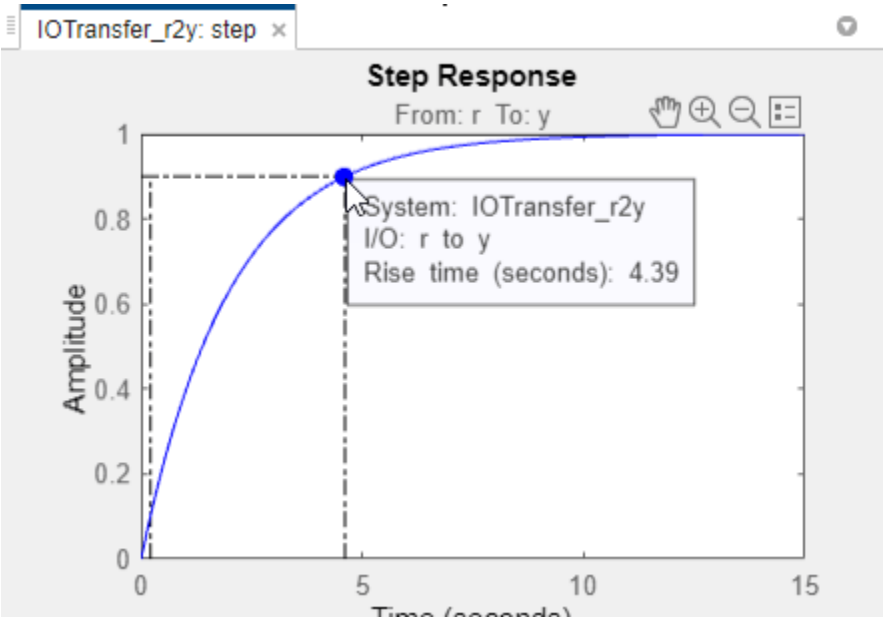


Select a **Dominant closed-loop time constant** of 2 and a **Desired controller order** of 2.



To update the controller, click **Udate Compensator**.

In the closed-loop step response plot, the tuned controller produces a closed-loop rise time around 4.4 seconds.

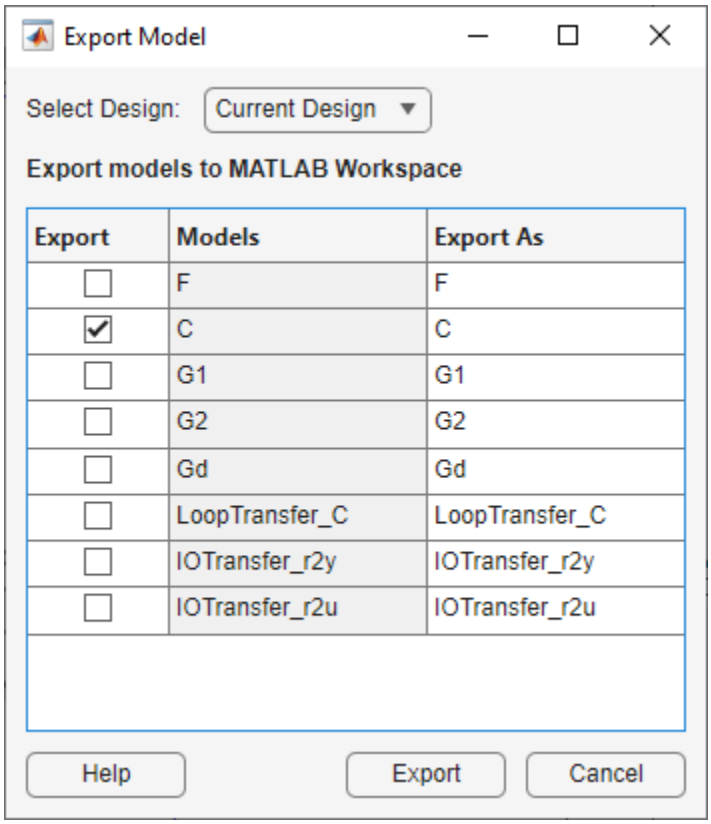


Control Performance with Model Mismatch

When designing the controller, you assumed G1 was equal to G2. In practice, they are often different, and the controller needs to be robust enough to track setpoints and reject disturbances.

Create model mismatches between G1 and G2 and examine the control performance at the MATLAB® command line in the presence of both setpoint change and load disturbance.

Export the IMC controller to the MATLAB workspace. Click **Export**. In the Export Model dialog box, select compensator model **C**.



Click **Export**.

The app exports the following cotnroller.

```
C = zpk([-0.121 -0.121],[-0.2419, -0.5],2.5647)
```

```
C =

  2.5647 (s+0.121)^2
  ------------------
  (s+0.2419) (s+0.5)

Continuous-time zero/pole/gain model.
```

Convert the IMC structure to a classic feedback control structure with the controller in the feedforward path and unit feedback.

```
C_new = feedback(C,G2,+1)
```

```
C_new =

               2.5647 (s+0.121)^4
  ------------------------------------------
  (s-0.0004396) (s+0.121) (s+0.1213) (s+0.242)

Continuous-time zero/pole/gain model.
```

Define the following plant models:

- No Model Mismatch:

```
G1p = (13.3259*s+3.2239)/(8.2677*s+1)^2;
```

- `G1` time constant changed by 5%:

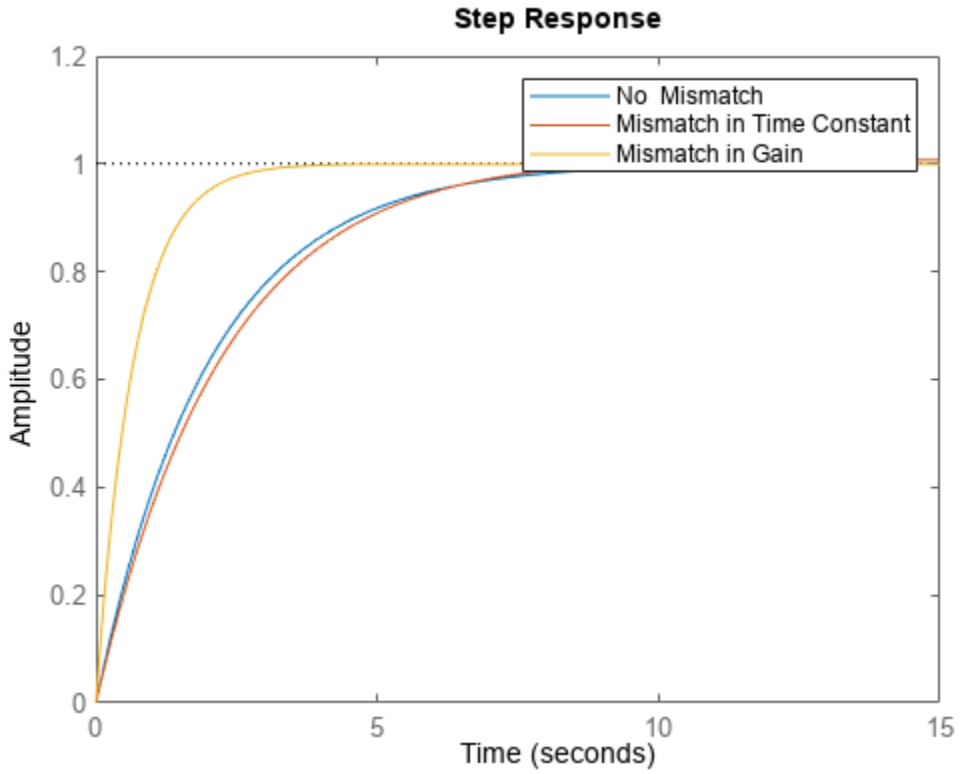
```
G1t = (13.3259*s+3.2239)/(8.7*s+1)^2;
```

- `G1` gain increased by 3 times:

```
G1g = 3*(13.3259*s+3.2239)/(8.2677*s+1)^2;
```

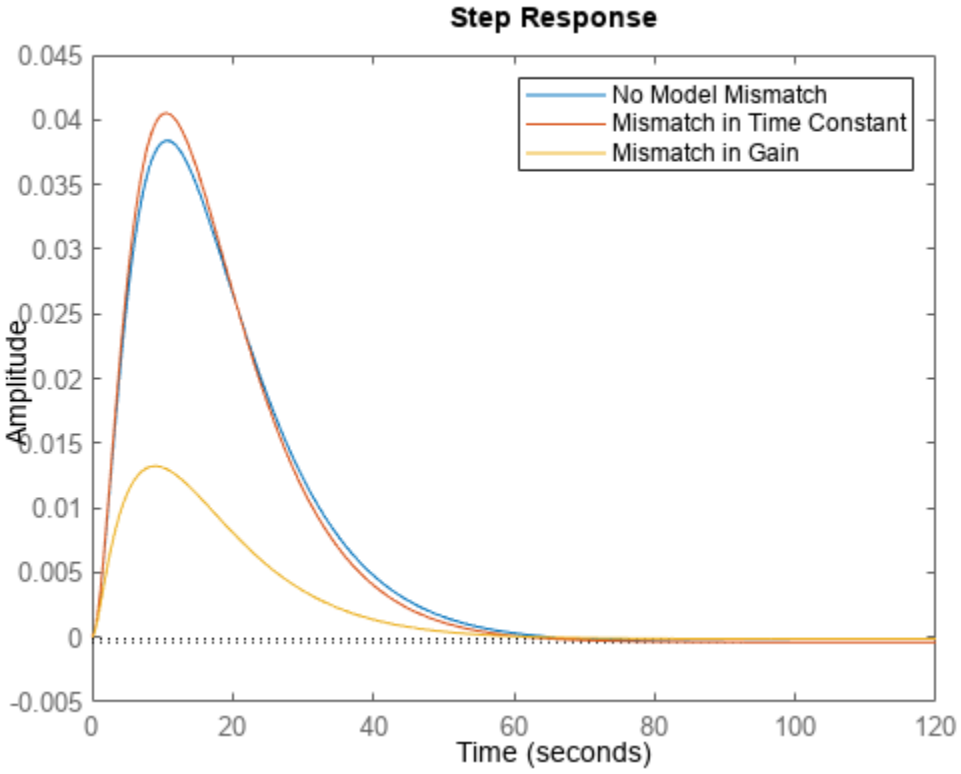
Evaluate the setpoint tracking performance.

```
step(feedback(G1p*C_new,1),feedback(G1t*C_new,1),feedback(G1g*C_new,1))
legend('No  Mismatch','Mismatch in Time Constant','Mismatch in Gain')
```



Evaluate the disturbance rejection performance.

```
step(Gd*feedback(1,G1p*C_new),Gd*feedback(1,G1t*C_new),Gd*feedback(1,G1g*C_new))
legend('No Model Mismatch','Mismatch in Time Constant','Mismatch in Gain')
```



The controller is fairly robust to uncertainties in the plant parameters.

See Also

Control System Designer

More About

- “Design Compensator Using Automated Tuning Methods” on page 12-83
- “Control System Designer Tuning Methods” on page 12-5
- “Analyze Designs Using Response Plots” on page 12-95

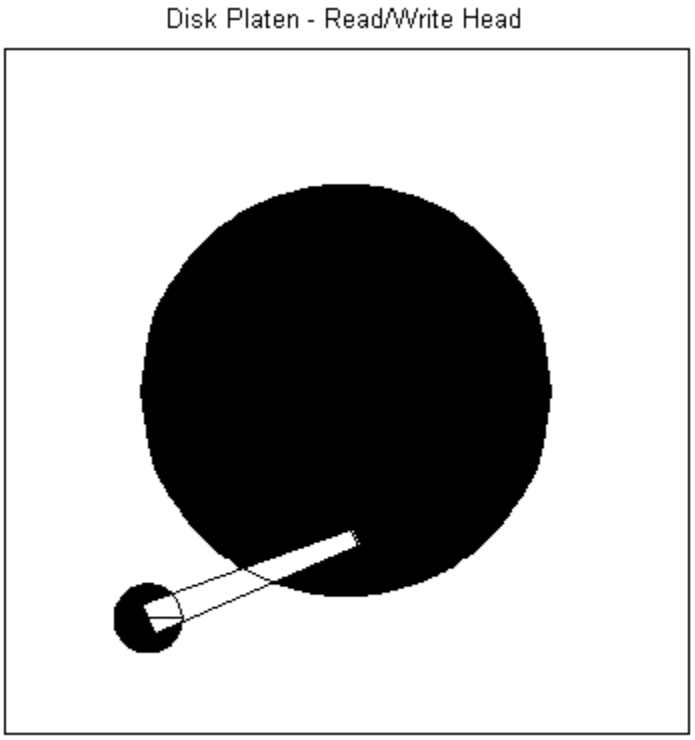
Design LQG Tracker Using Control System Designer

This example shows how to use LQG synthesis to design a feedback controller for a disk drive read/write head using Control System Designer.

For details about the system and model, see Chapter 14 of "Digital Control of Dynamic Systems," by Franklin, Powell, and Workman.

Disk Drive Model

Below is a picture of the system to be modeled.



The model input is the current driving the voice coil motor, and the output is the position error signal (PES, in % of track width). To learn more about the 10th order model, see “Digital Servo Control of a Hard-Disk Drive” on page 12-223. The plant includes a small time delay. For the purpose of this example, ignore this delay.

```
load diskdemo
Gr = tf(1e6,[1 12.5 0]);
Gf1 = tf(w1*[a1 b1*w1],[1 2*z1*w1 w1^2]); % first  resonance
Gf2 = tf(w2*[a2 b2*w2],[1 2*z2*w2 w2^2]); % second resonance
Gf3 = tf(w3*[a3 b3*w3],[1 2*z3*w3 w3^2]); % third  resonance
Gf4 = tf(w4*[a4 b4*w4],[1 2*z4*w4 w4^2]); % fourth resonance
G = (ss(Gf1)+Gf2+Gf3+Gf4) * Gr;     % convert to state space for accuracy
```

Design Overview

In this example, design a full-ordered LQG tracker, which places the read/write head at the correct position. Tune the LQG tracker to achieve specific performance requirements and reduce the controller order as much as possible. For example, turn the LQG tracker into a PI controller format.

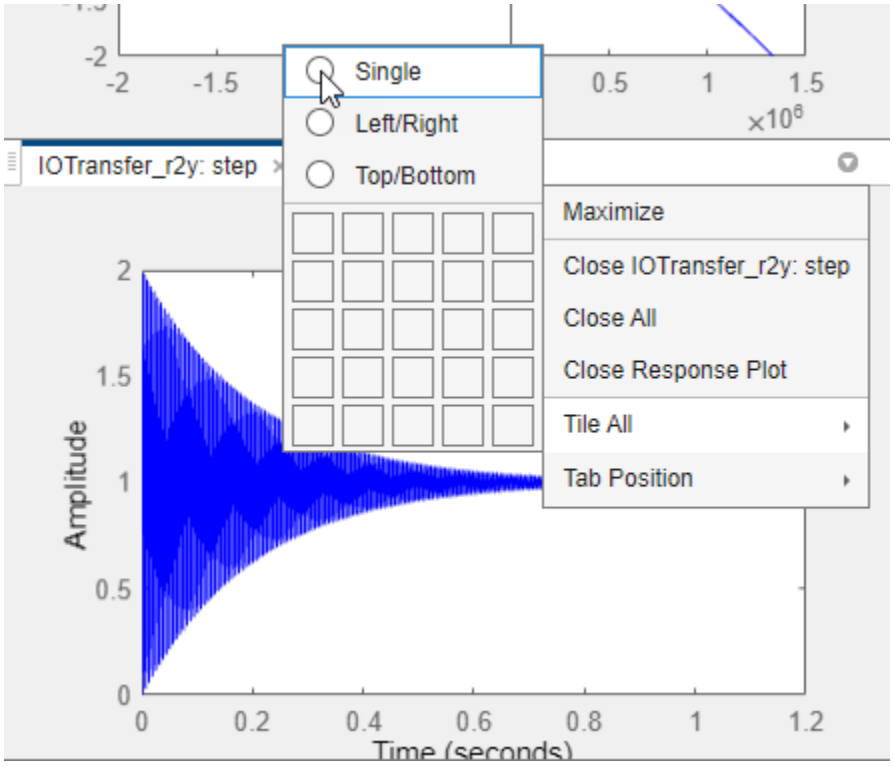
Open Control System Designer

Open **Control System Designer**, importing the plant model.

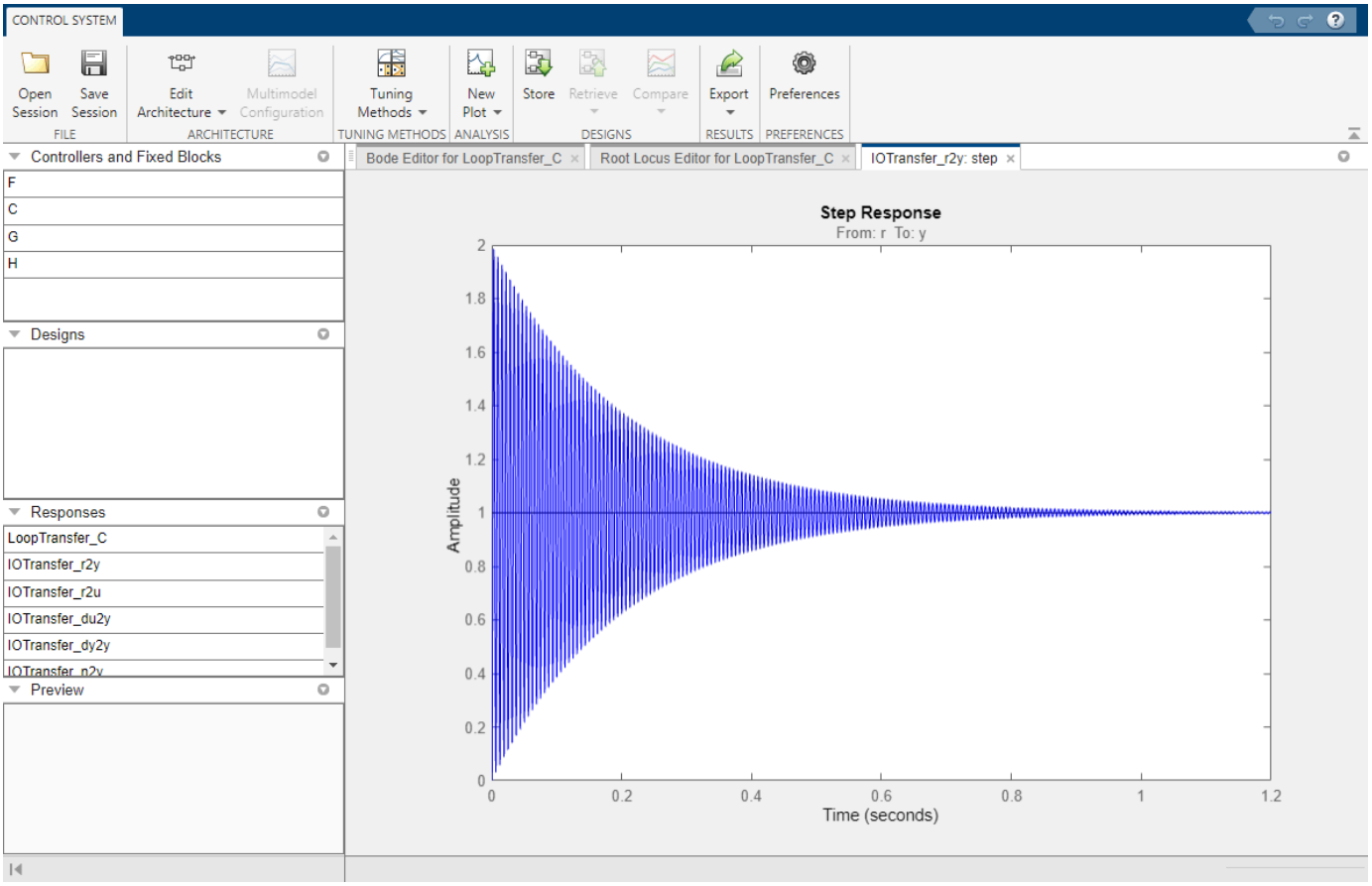
```
controlSystemDesigner(G)
```

By default, **Control System Designer** displays the step response of the closed-loop system along with Bode and root locus graphical editors for the open-loop response.

To view only the step response, click the arrow in the top right corner of the plot and select **Tile All > Single**.



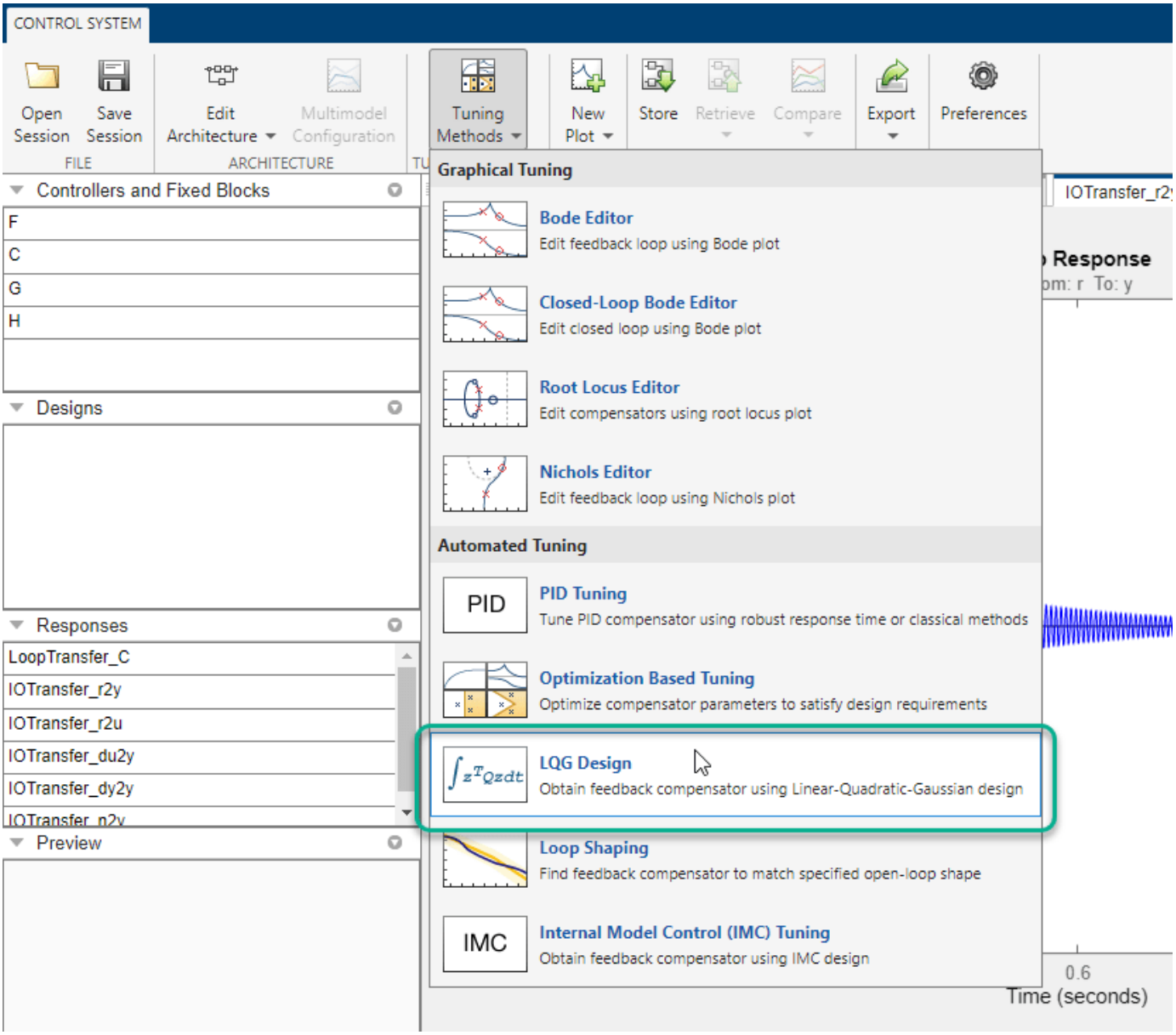
The app expands the step response plot in the document area.Details about how to use **Control System Designer** are described in “Getting Started with the Control System Designer” on page 12-178.



The default unity gain compensator produces a stable closed-loop system with large oscillations.

Design a Full-Order LQG Tracker

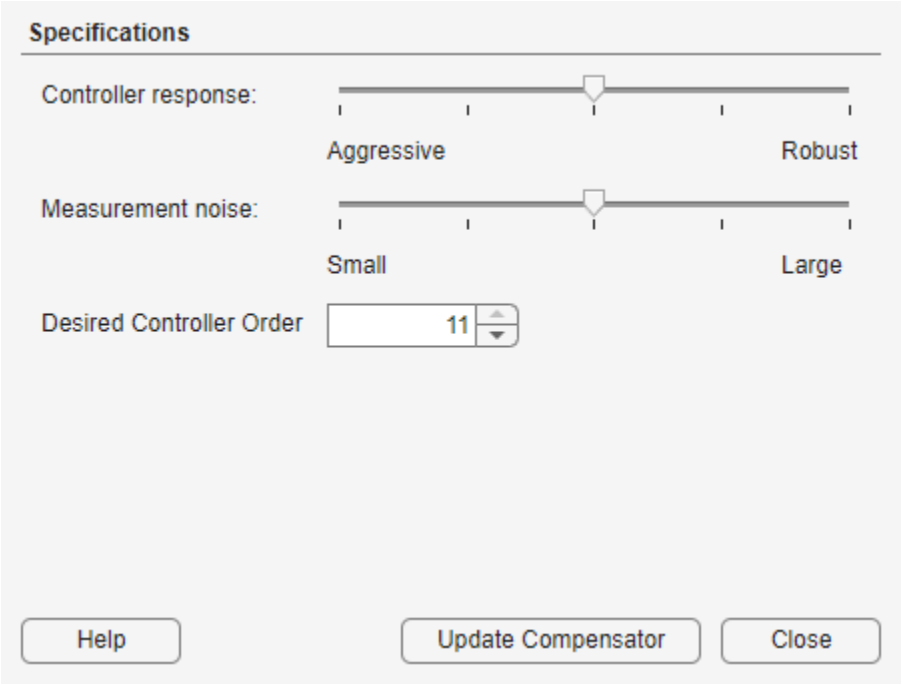
To design an LQG tracker, click **Tuning Methods**, and select **LQG Design**.



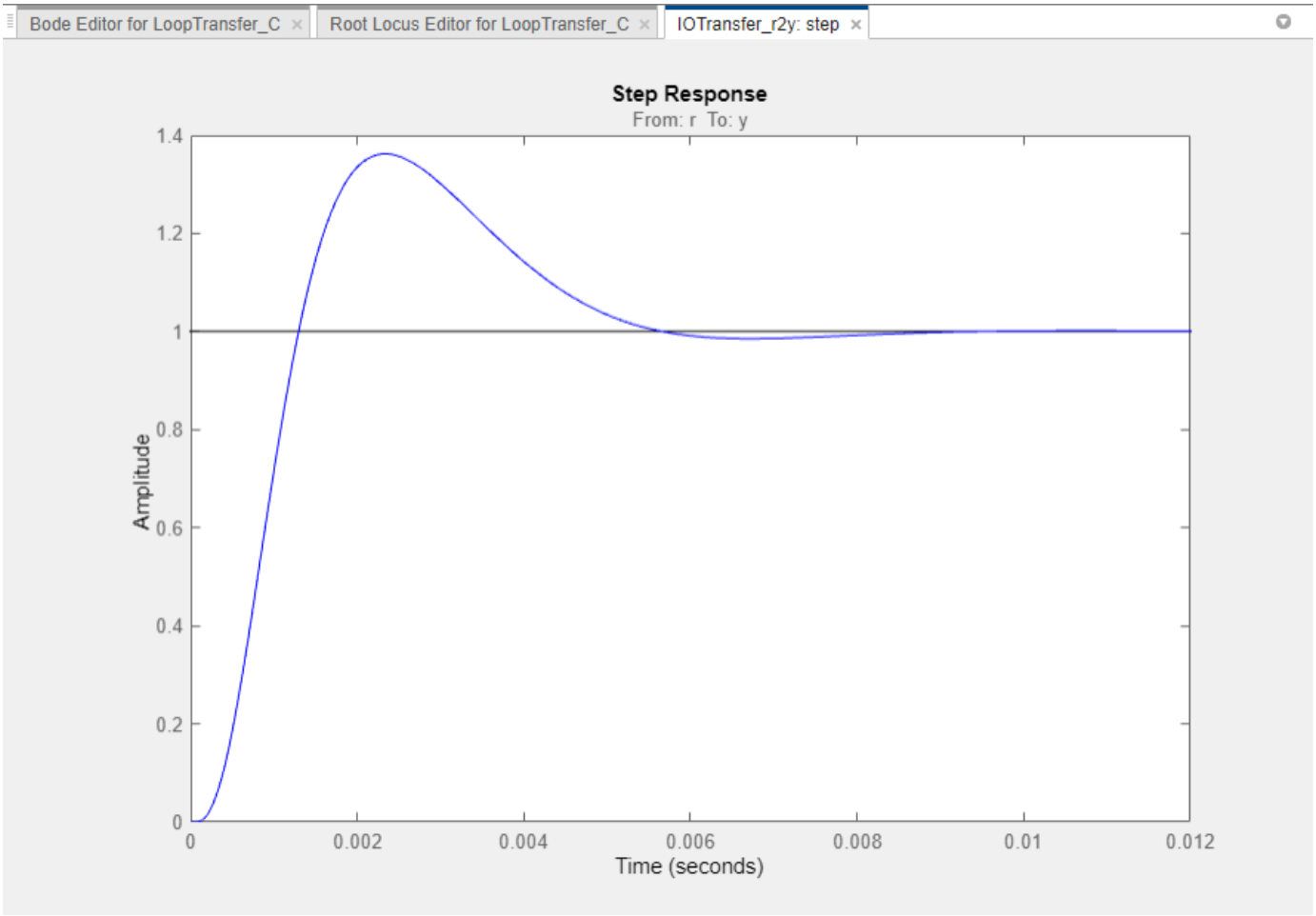
In the LQG Design dialog box, in the **Specifications** section, set requirements on the controller performance:

- **Controller response** - Specify the controller transient behavior. You can make the controller more aggressive at disturbance rejection or more robust against plant uncertainty. If you believe your model is accurate and that the manipulated variable has a large enough range, an aggressive controller is preferable.
- **Measurement noise** - Specify an estimate of the level of output measurement noise for your application. To produce a more robust controller, specify a larger noise estimate.
- **Desired controller order** - Specify your controller order preference.

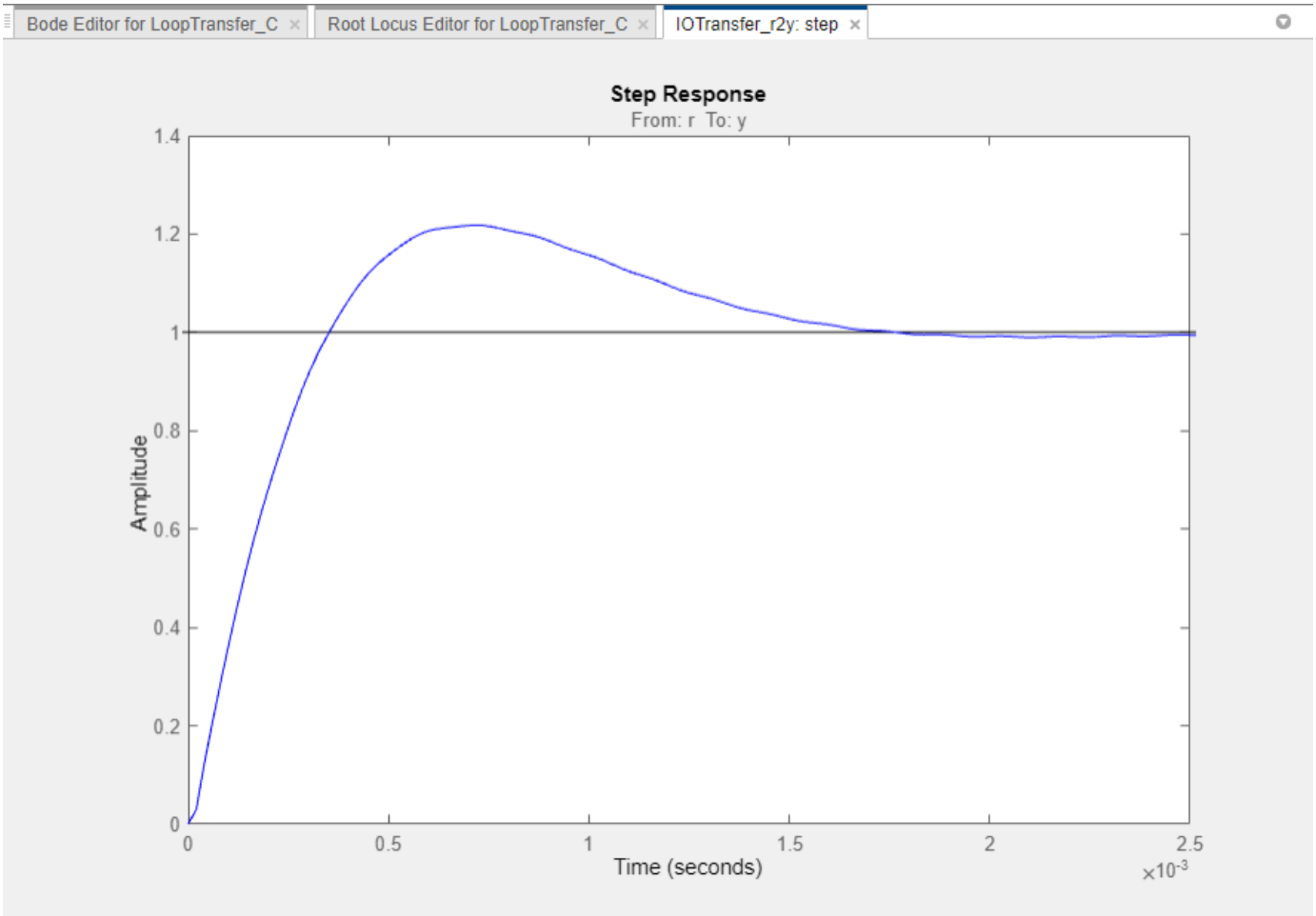
For an initial full-order controller design, set the **Controller response** and **Measurement** sliders to the center and select a **Desired Controller Order** of 11.



Click **Update Compensator**. The new **Compensator** is displayed, and the step response updates.

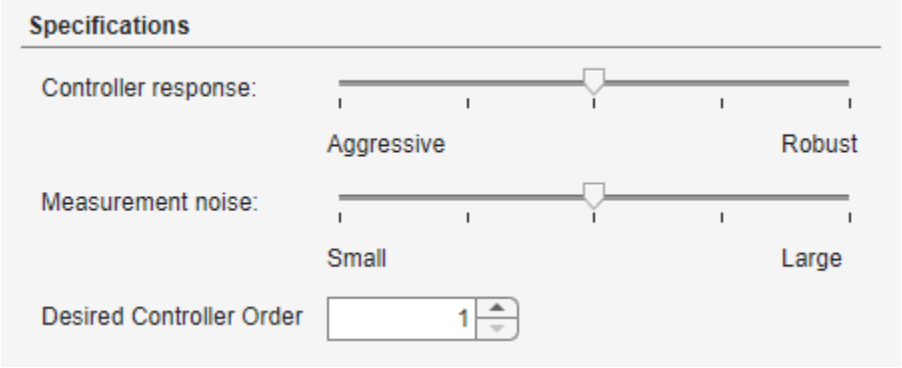


To design a more aggressive controller, move the **Controller response** slider to the far left and click **Update Compensator**. The more aggressive controller reduces the overshoot by 50% and reduces the settling time by 70%.

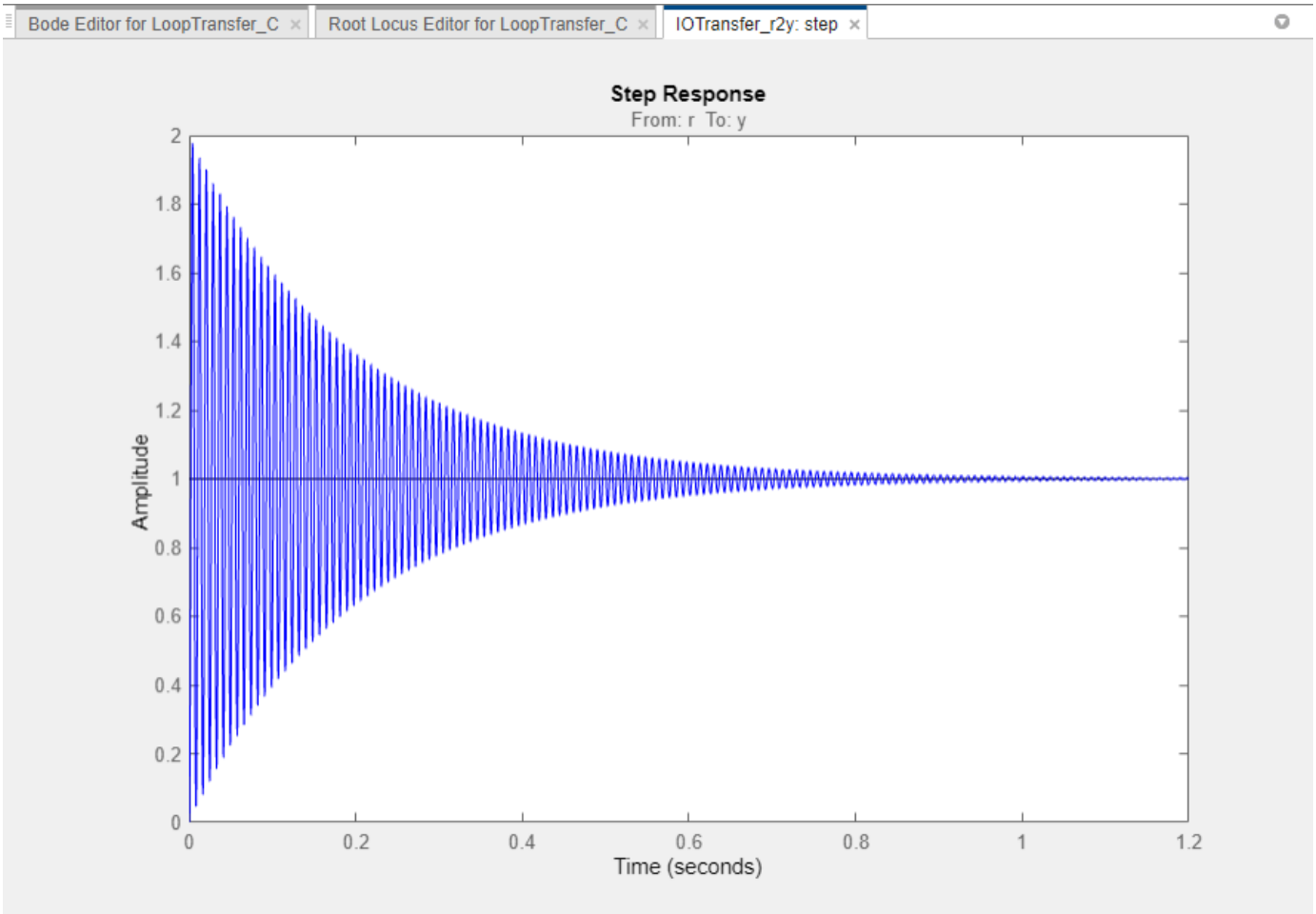


Design a Reduced-Order LQG Tracker

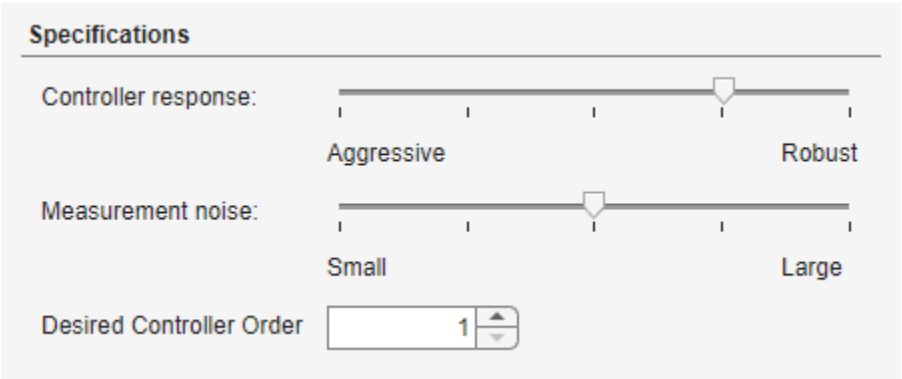
To create a PI controller, reset the **Controller response** slider to the middle value, and set the **Desired Controller Order** to 1.



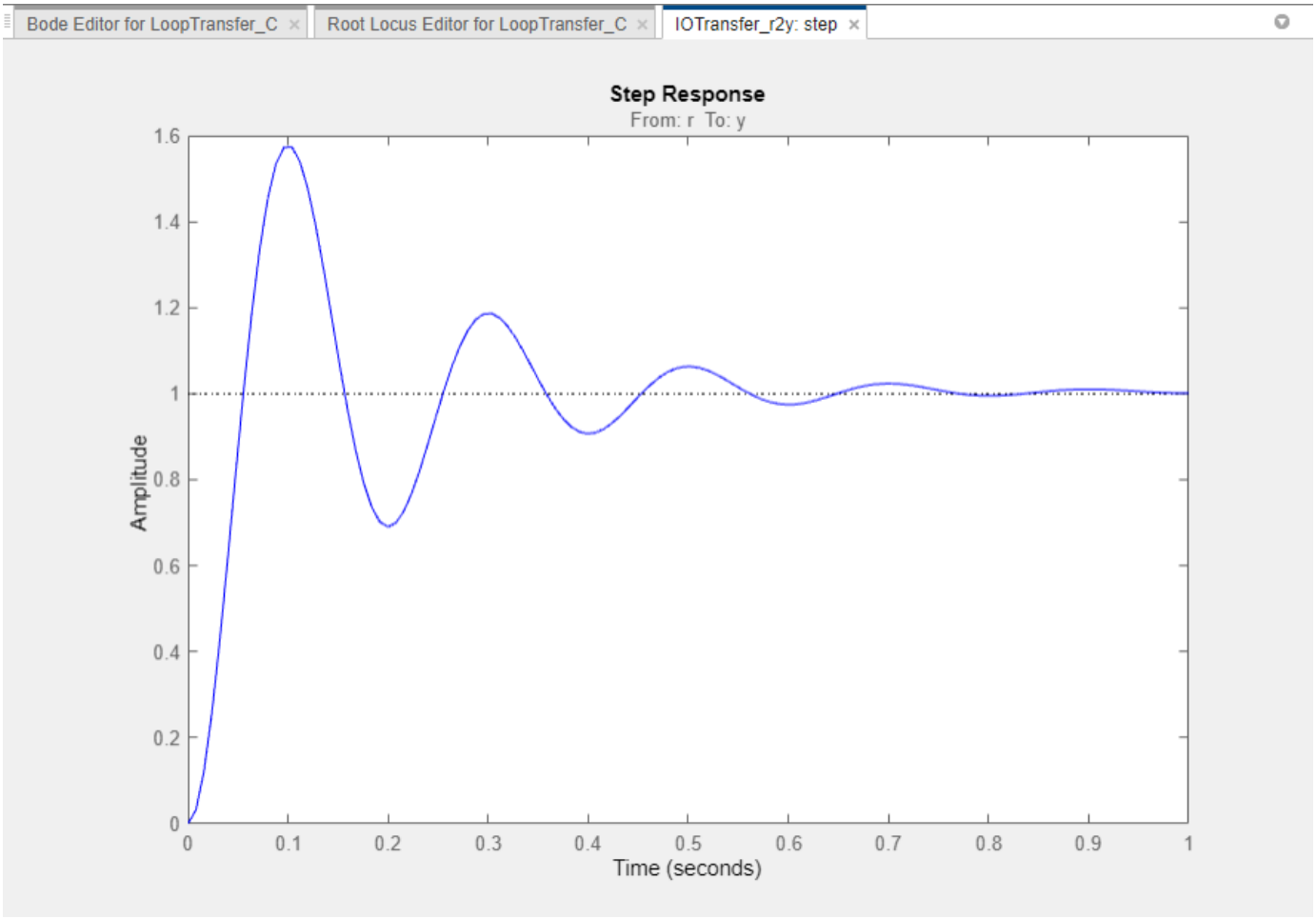
Click **Update Compensator**.



This controller produces a heavily oscillating closed-loop system. To make the controller less aggressive, move the **Controller response** slider to the right.



Click **Update Compensator**.



The step response shows that the PI controller design provides a good starting point for optimization-based design. For more information, see “Getting Started with the Control System Designer” on page 12-178.

See Also

Control System Designer

More About

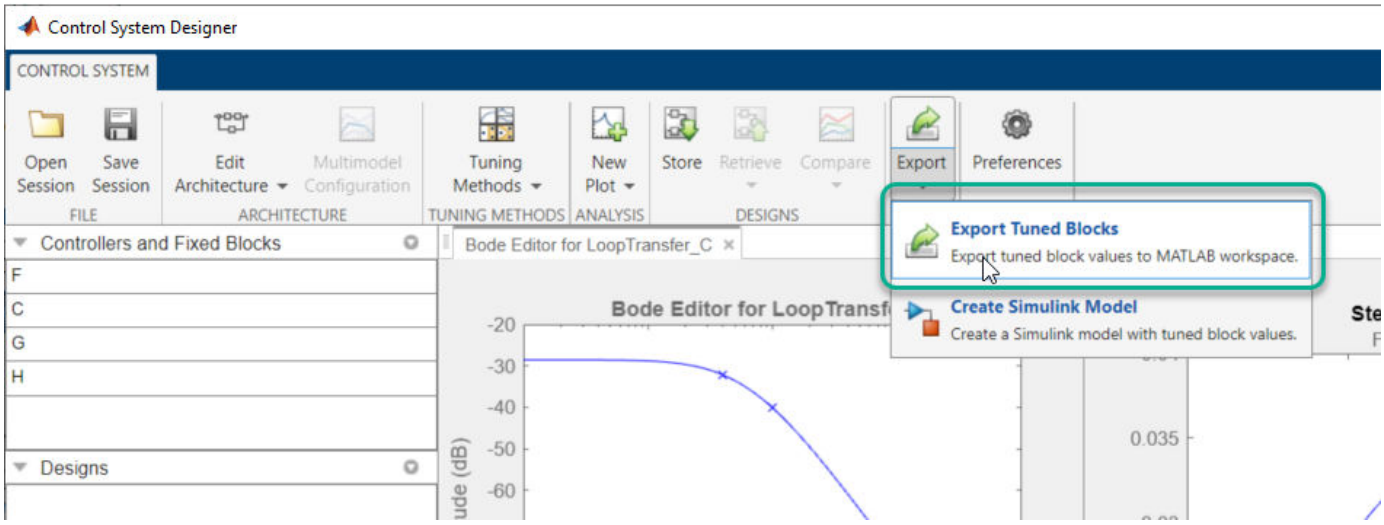
- “Design Compensator Using Automated Tuning Methods” on page 12-83
- “Control System Designer Tuning Methods” on page 12-5
- “Analyze Designs Using Response Plots” on page 12-95

Export Design to MATLAB Workspace

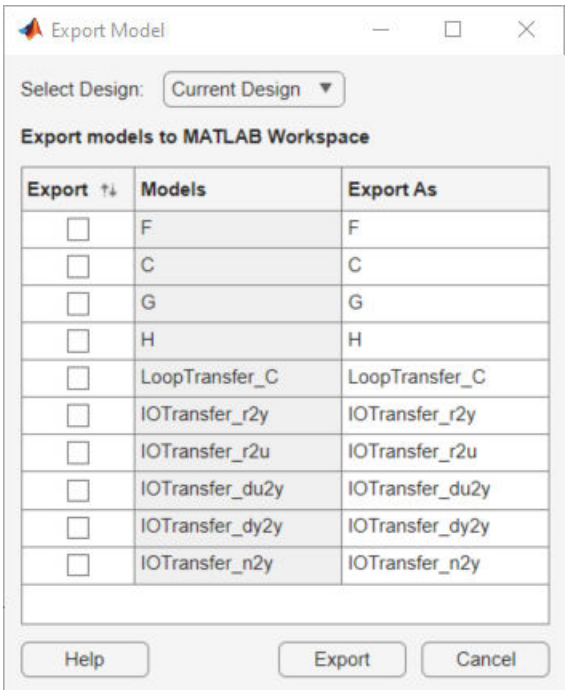
After designing your controller in **Control System Designer**, you can export your design to the MATLAB Workspace for further analysis or design.

To export your design:

1. In **Control System Designer**, on the **Control System** tab, under **Export**, click **Export tuned blocks**.



2. In the Export Model dialog box, in the **Select Design** drop-down list, choose the design that you want to export. You can select either the `Current Design` or one of the stored designs from the **Data Browser**.



3. In the **Export models to MATLAB Workspace** table, in the **Export** column, select the models you want to export.

   For all designs, you can export the controller and prefilter models. Also, for the `Current Design`, you can export the fixed block models and any responses from the **Data Browser**.

   For more information on the prefilter, controller, and fixed blocks in each control architecture, see “Feedback Control Architectures” on page 12-21.

4. In the **Export as** column, you can specify an alternate name for the exported model. Exporting a model with the same name as an existing variable in the MATLAB Workspace overwrites the variable.

5. To save the selected models to the MATLAB Workspace, click **Export**.

See Also

Apps
Control System Designer

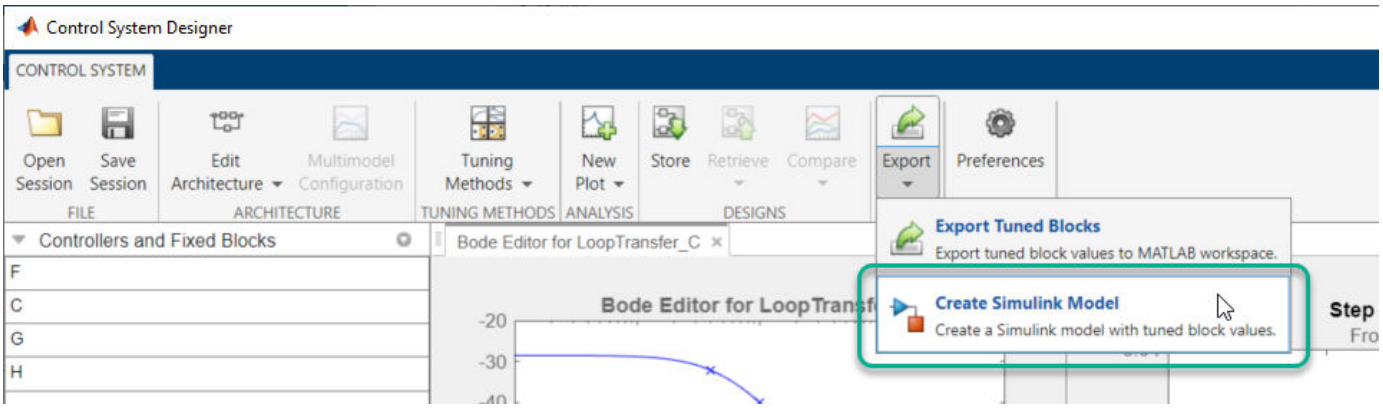
More About

- “Feedback Control Architectures” on page 12-21
- “Generate Simulink Model for Control Architecture” on page 12-151

Generate Simulink Model for Control Architecture

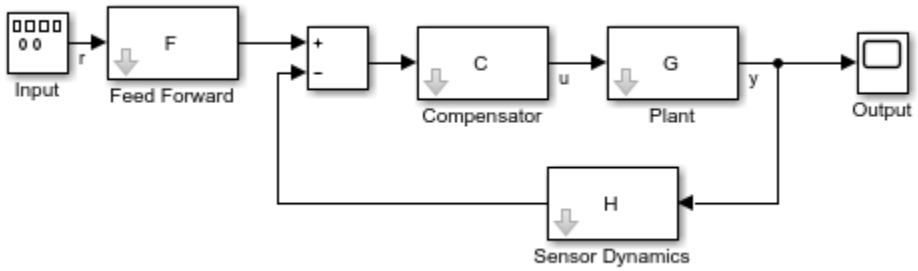
After designing your controllers in **Control System Designer**, to simulate your system, you can automatically generate a Simulink model for your control architecture.

To do so, on the **Control System** tab, under **Export**, click **Create Simulink Model**.



The app exports the controllers and fixed blocks for the current design to the MATLAB Workspace and generates a Simulink model that matches the current control architecture. For more information on the controllers and fixed blocks in each control architecture, see “Feedback Control Architectures” on page 12-21.

For example, if you design a control system using configuration 1, **Control System Designer** exports `C`, `F`, `G`, and `H` to the MATLAB Workspace and generates the following Simulink model.



In the generated model, the Input block is a Signal Generator. Using this block, you simulate your model with different input waveforms, such as sine waves or random signals. To generate a step response, replace the Input block with a Step block.

To generate a Simulink model for a stored design, first make that design current. On the **Control System** tab, under **Retrieve**, select the design for which you want to generate a model.

See Also

Control System Designer

More About

- “Feedback Control Architectures” on page 12-21
- “Export Design to MATLAB Workspace” on page 12-149

Tune Simulink Blocks Using Compensator Editor

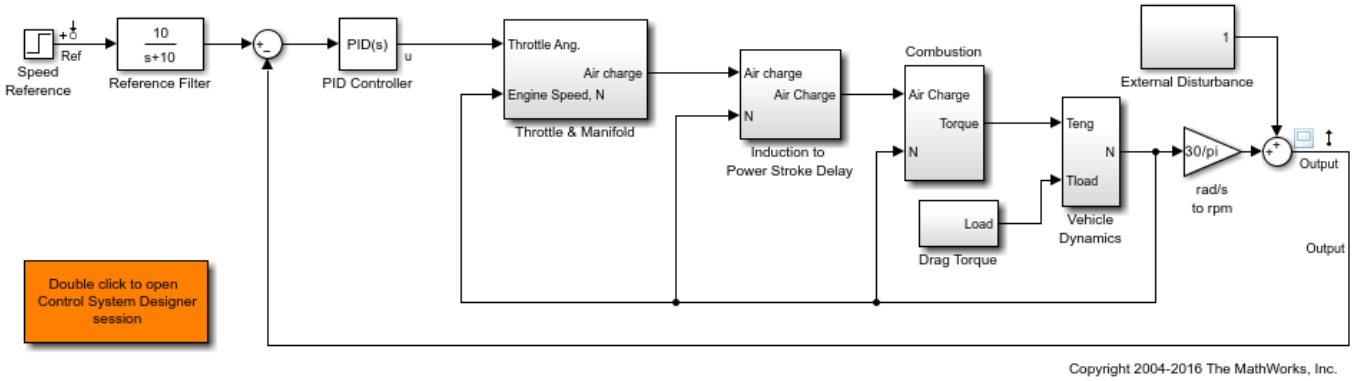
This example shows how to tune Simulink® blocks using the Compensator Editor dialog box in **Control System Designer**.

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” (Simulink Control Design).

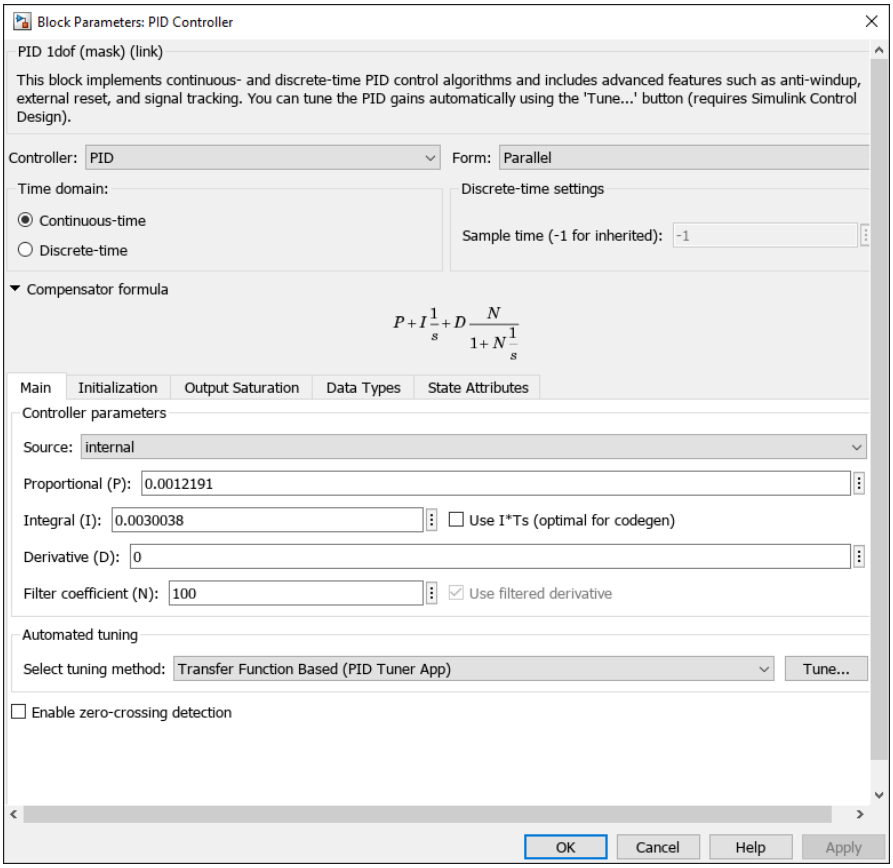
Open and explore the engine speed control model.

```
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:

$$G(s) = \frac{Ps(Ns+1) + I(Ns+1) + Ds^s}{s(Ns+1)} = \frac{K(s^2 + 2\zeta\omega_n + w_n^2)}{s(s+z)}$$

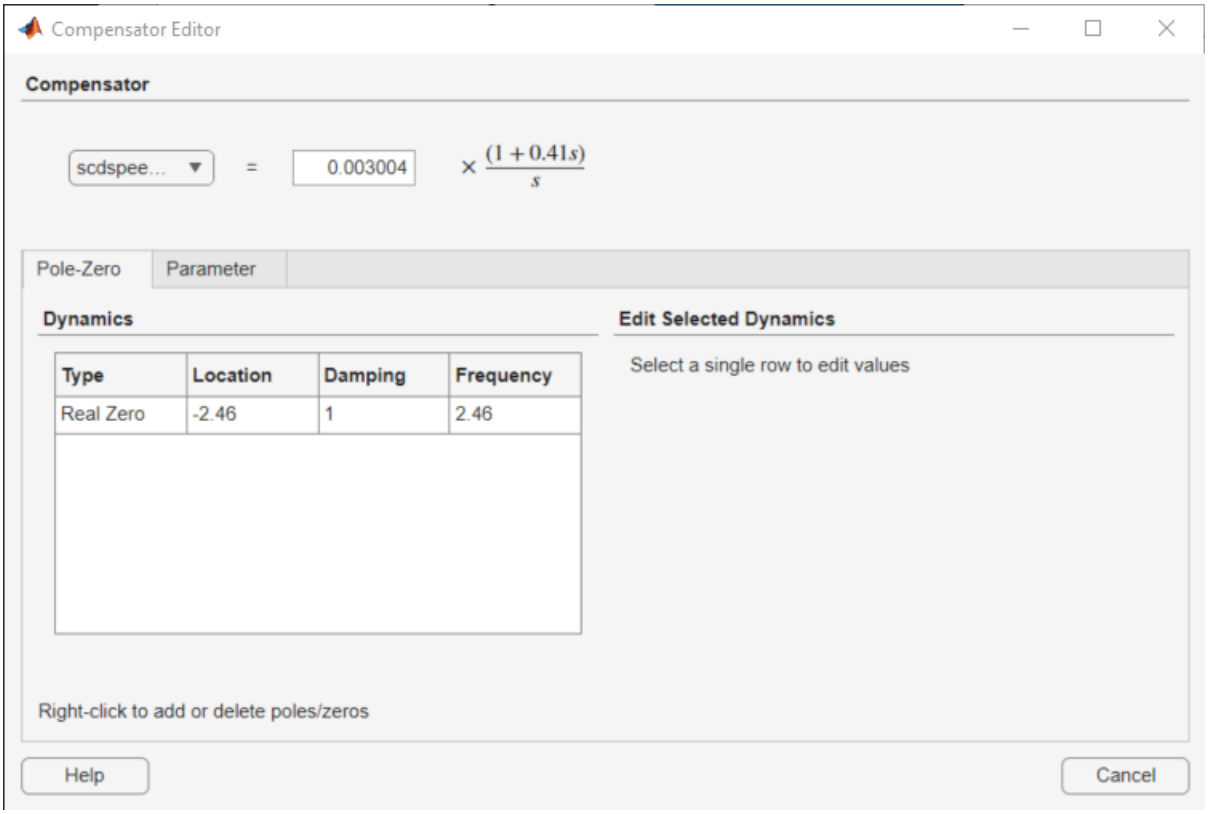
This formulation of poles, zeros, and gains allows for direct graphical tuning on design plots such as Bode, root locus, and Nichols plots. Additionally, **Control System Designer** allows for both representations to be tuned using the Compensator Editor. The tuning of both representations is available for all supported blocks in Simulink Control Design™. For more information, see “What Blocks Are Tunable?” (Simulink Control Design).

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.

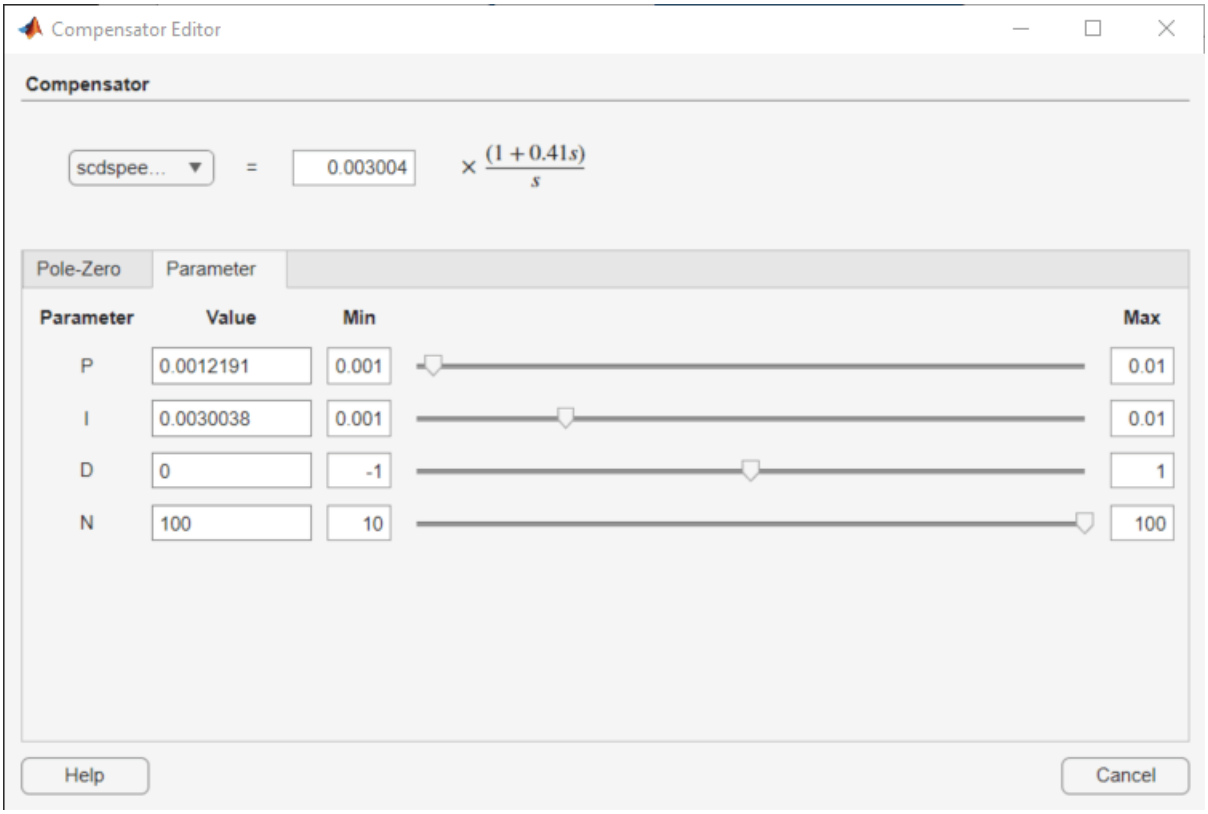
Compensator Editor Dialog Box

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” (Simulink Control Design) can be met with the following controller parameters:

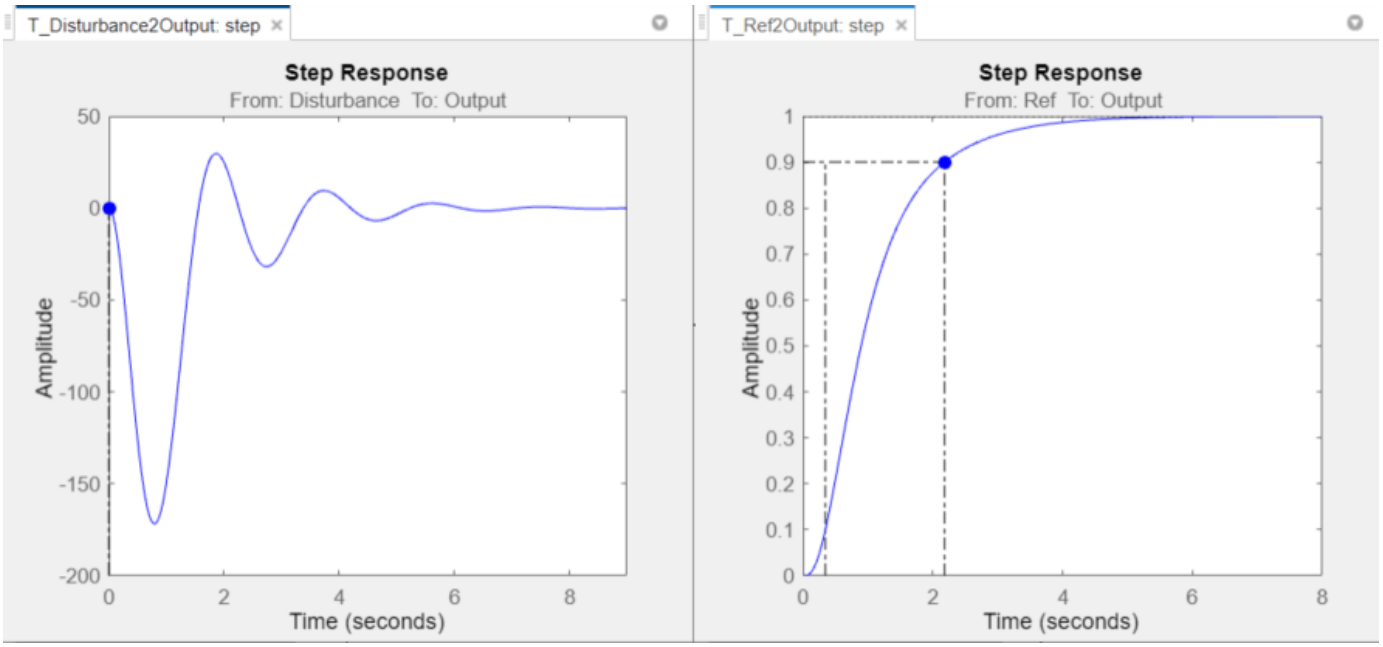
- `scdspeedctrl/PID Controller`:

```
P = 0.0012191
I = 0.0030038
```

- `scdspeedctrl/Reference Filter`:

```
Numerator = 10
Denominator = [1 10]
```

In the Compensator Editor dialog box, specify these parameters. 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” on page 12-78
- “Update Simulink Model and Validate Design” (Simulink Control Design)

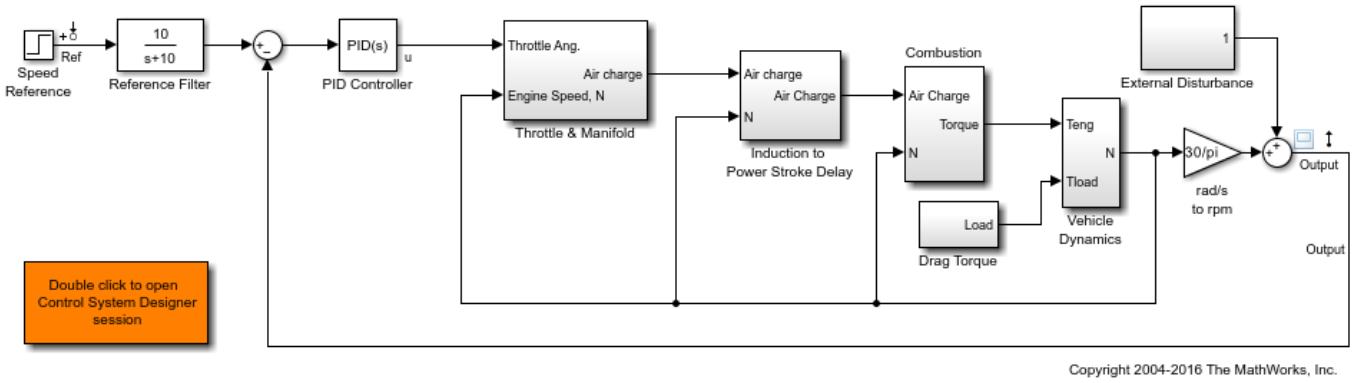
Single Loop Feedback/Prefilter Compensator Design

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

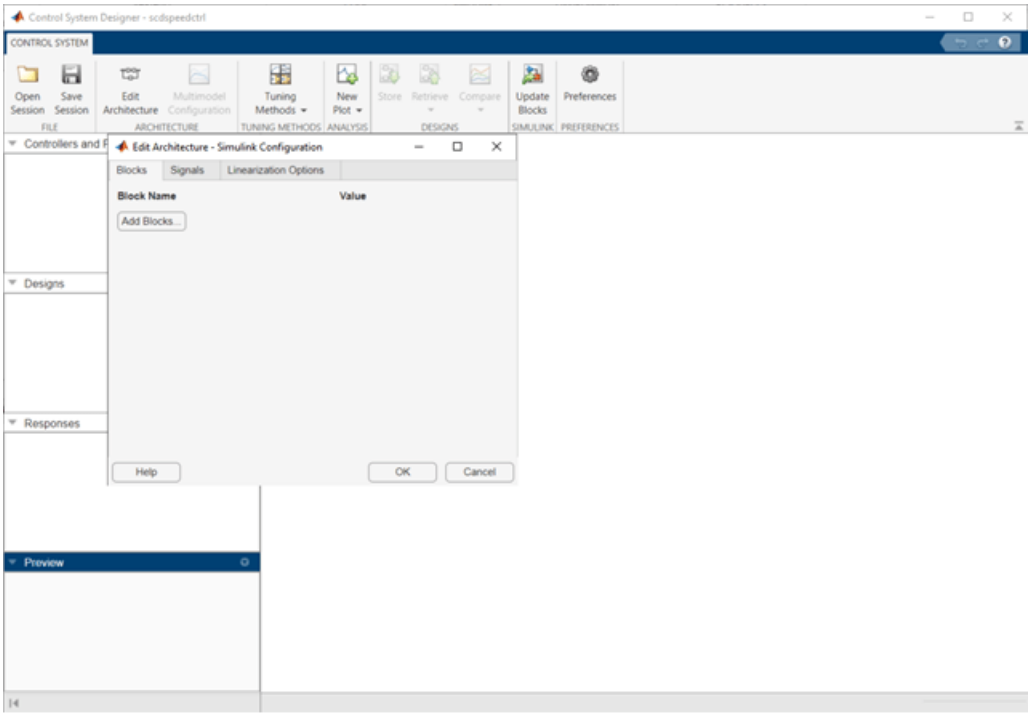
This example uses **Control System Designer** to tune the compensators in the feedback system. To open the **Control System Designer**

- 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

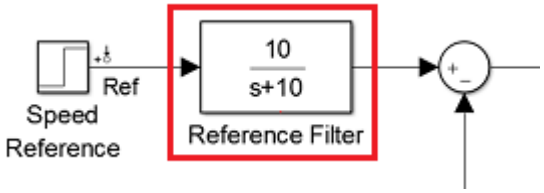
To open **Control System Designer**, in the Simulink model window, in the **Apps** gallery, click **Control System Designer**.

The **Edit Architecture** dialog box opens when the **Control System Designer** launches.

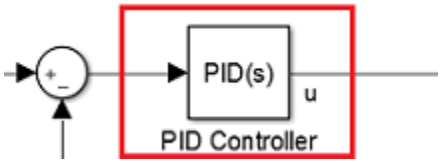


In the **Edit Architecture** dialog box, on the **Blocks** tab, click **Add Blocks**, and select the following blocks to tune:

- `scdspeedctrl/Reference Filter`

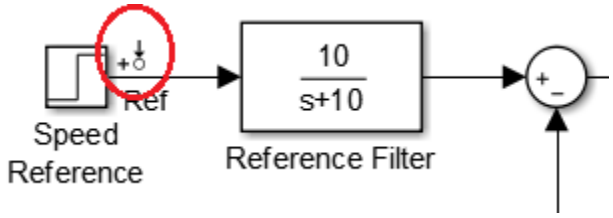


- `scdspeedctrl/PID Controller`

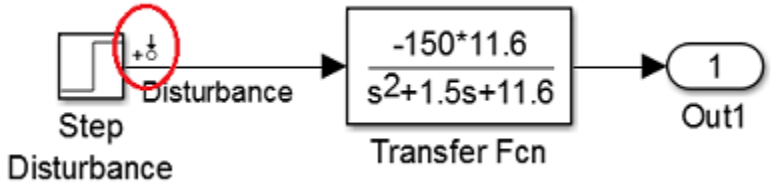


On the **Signals** tab, the analysis points defined in the Simulink model are automatically added as **Locations**.

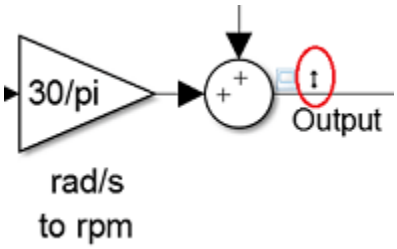
- Input: `scdspeedctrl/Speed Reference` output port 1



- Input `scdspeedctrl/External Disturbance/Step Disturbance` output port 1

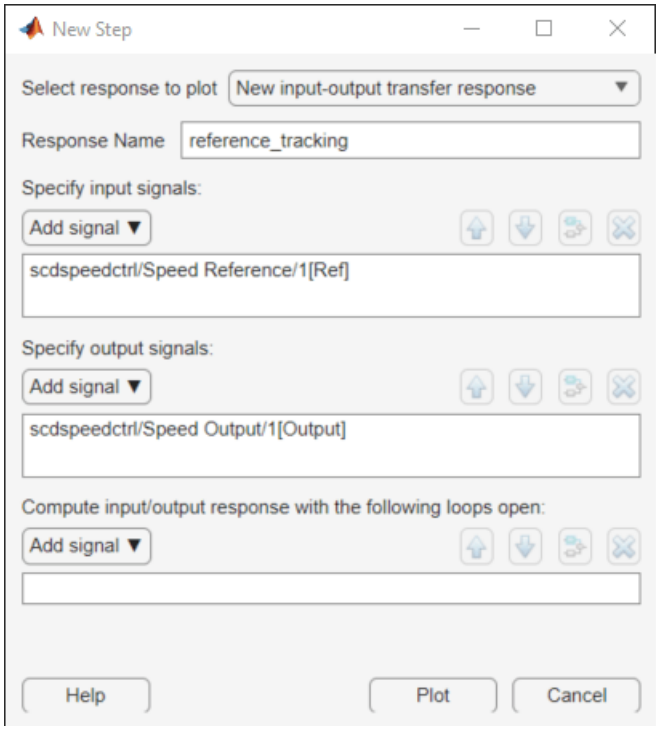


- Output `scdspeedctrl/Speed Output` output port 1



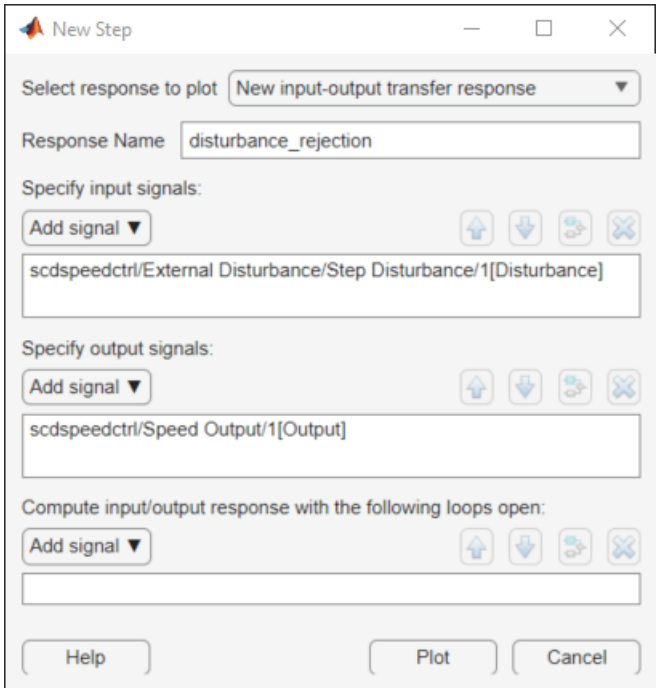
On the **Linearization Options** tab, in the **Operating Point** drop-down list, select **Model Initial Condition**.

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” (Simulink Control Design).
- 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:

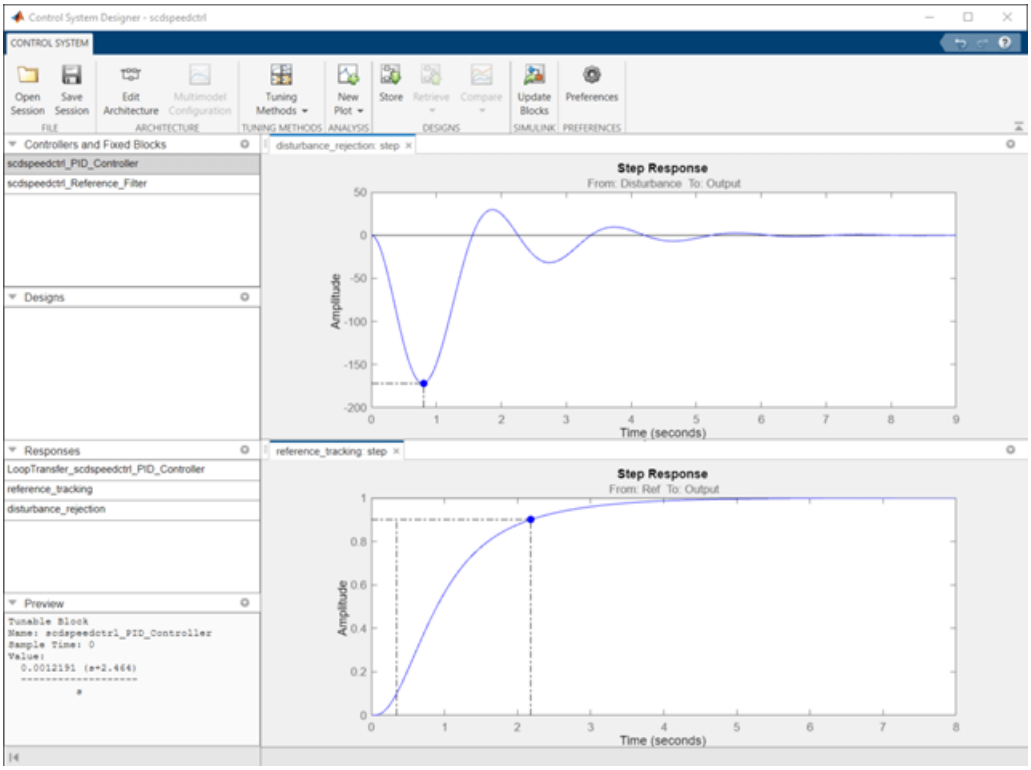
- `scdspeedctrl/PID Controller` has parameters:

```
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 12-5
- “Analyze Designs Using Response Plots” on page 12-95

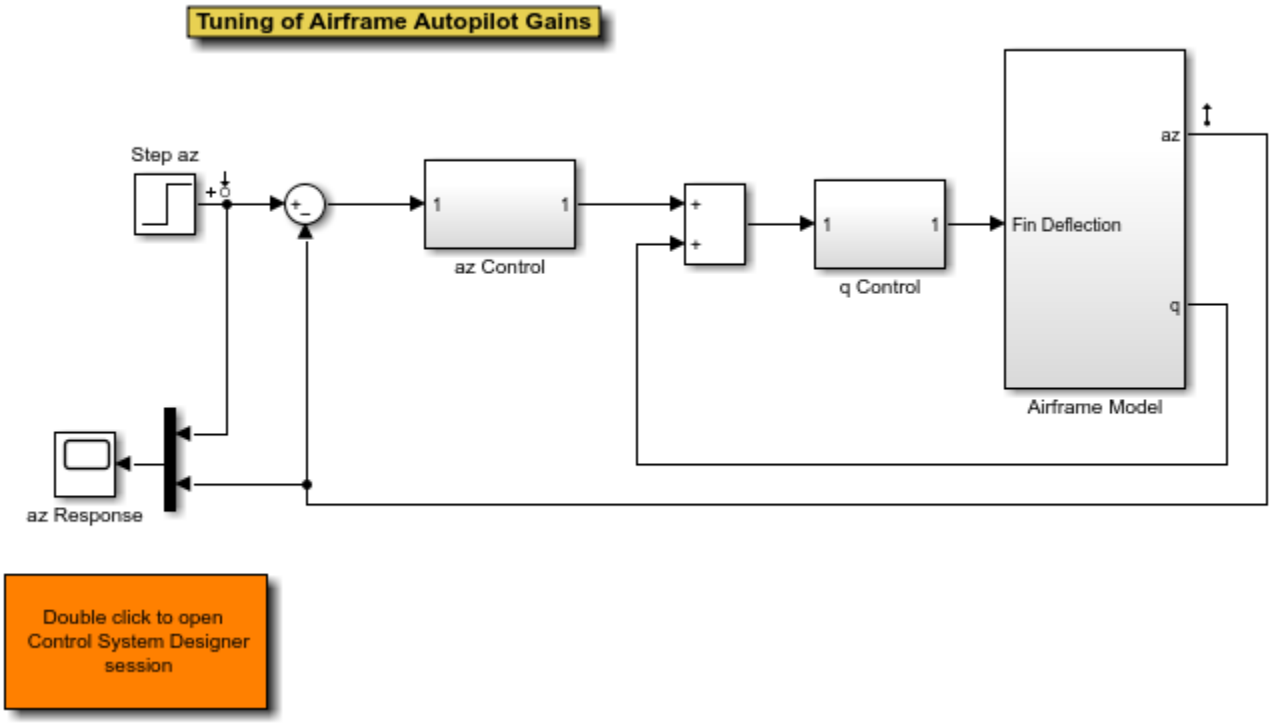
Cascaded Multiloop Feedback Design

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.

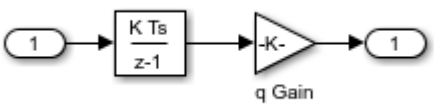
```
open_system('scdairframectrl')
```



The two feedback controllers are:

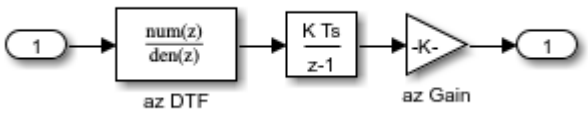
- `scdairframectrl/q Control` - A discrete-time integrator and a gain block stabilize the inner loop.

```
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.

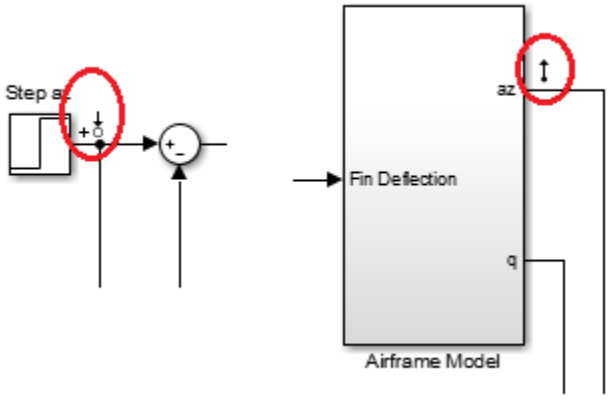
To open **Control System Designer**, in the Simulink model, in the **Apps** gallery, click **Control System Designer**.

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



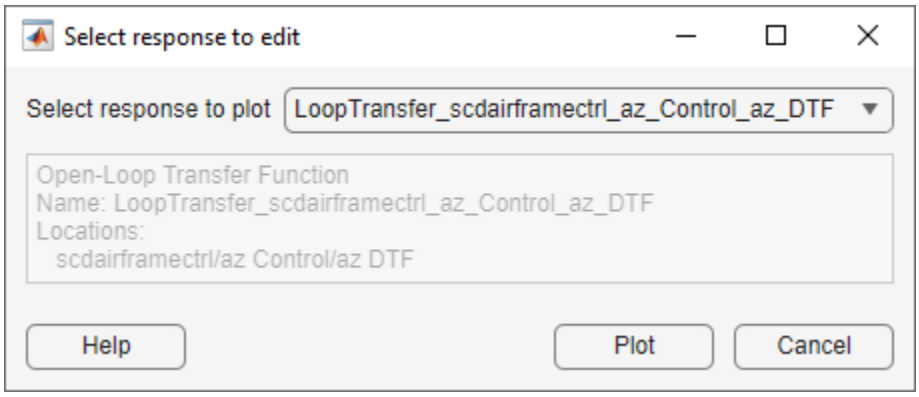
To use the selected blocks and signals, click **OK**.

In **Control System Designer**, in the data browser on the, the **Responses** section contains the following open-loop responses, which **Control System Designer** automatically recognizes as potential feedback loops for open-loop design.

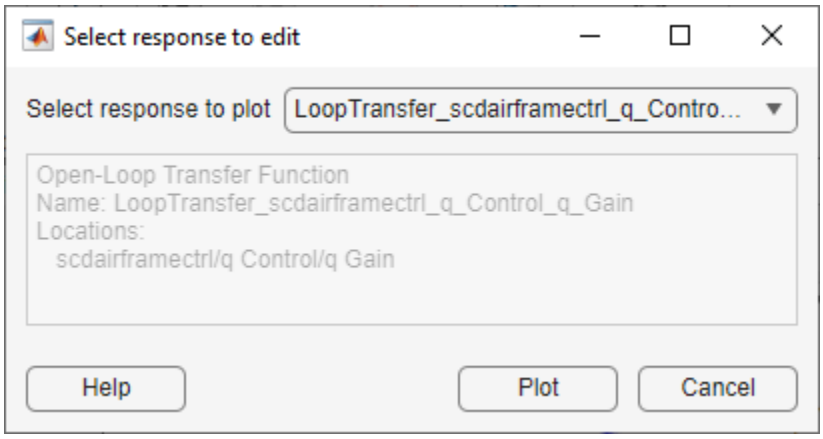
- Output port 1 of `scdairframectrl/az Control/az DTF`
- Output port 1 of `scdairframectrl/az Control/az Gain`
- Output port 1 of `scdairframectrl/q Control/q Gain`

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** dialog box, in the **Select response to plot** drop-down list, select the corresponding open-loop responses, and click **Plot**.

- Open Loop at outport 1 of `scdairframectrl/az Control/az DTF`

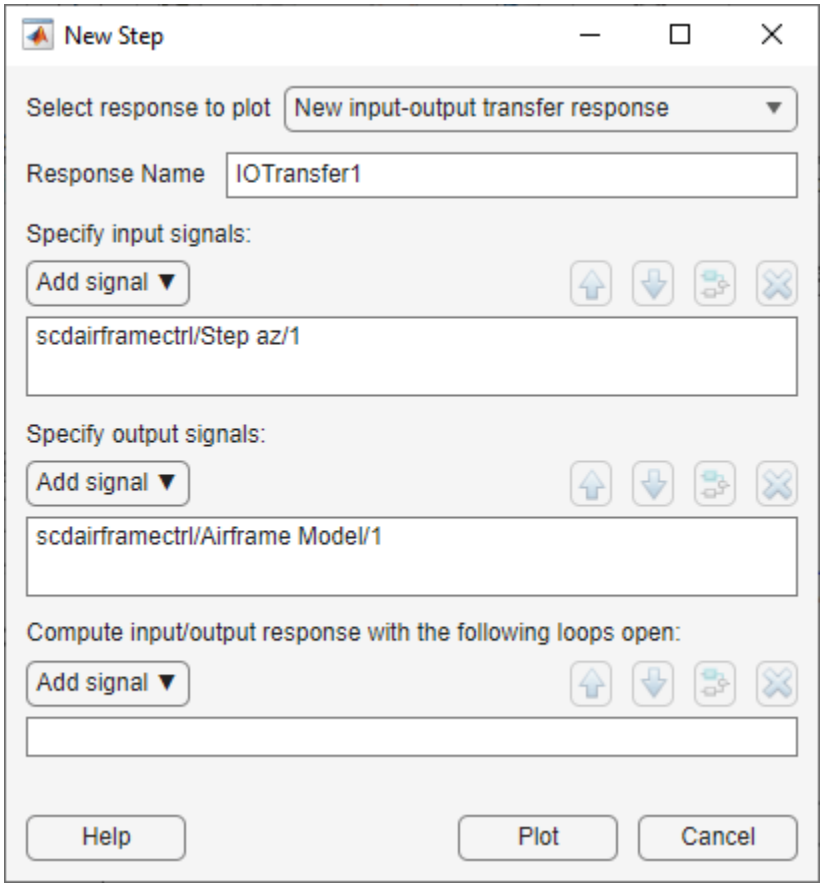


- Open Loop at outport 1 of `scdairframectrl/q Control/q Gain`



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 dialog box, in the **Select response to plot** drop-down list, select **New input-output transfer response**.

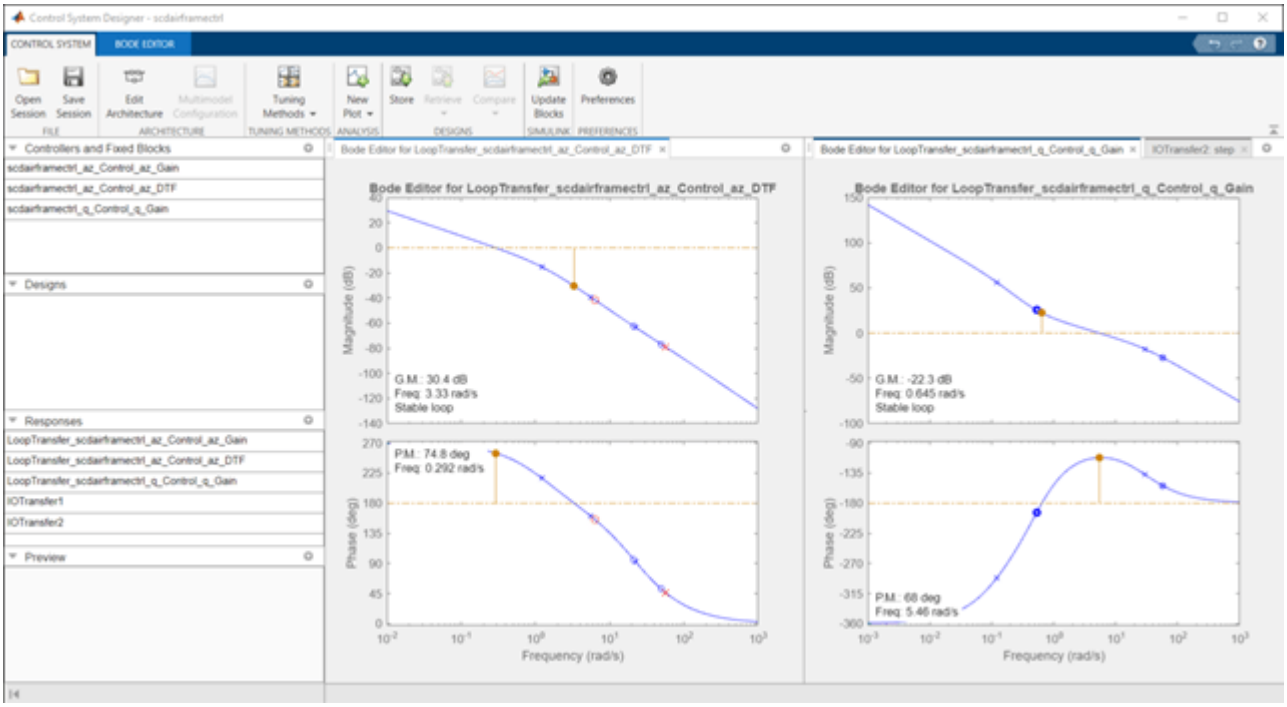
Add `scdairframectrl/Step az/1` as an input signal and `scdairframectrl/Airframe Model/1` as an output signal.



Click **Plot**.

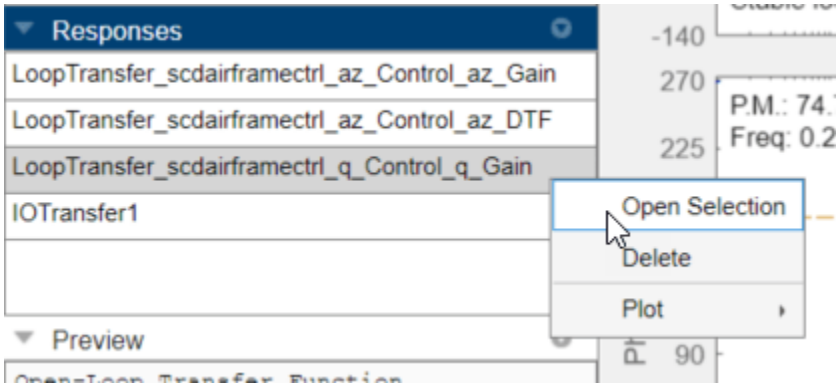
Remove Effect of Outer Feedback Loop

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.

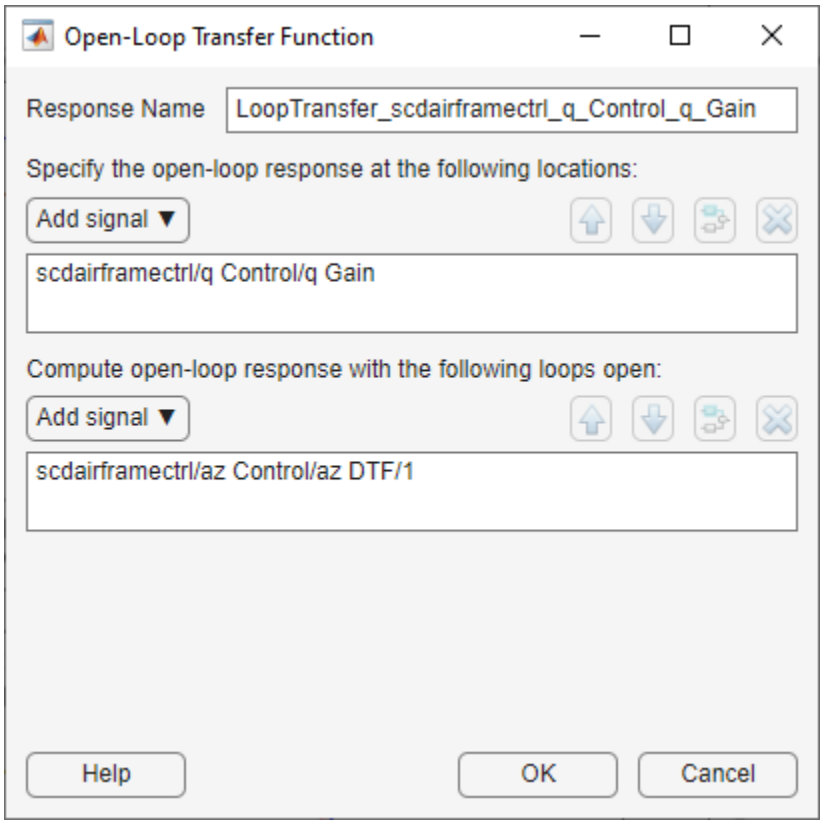


To remove the effect of the outer loop when designing the inner loop, add a loop opening to the open-loop response of the inner loop.

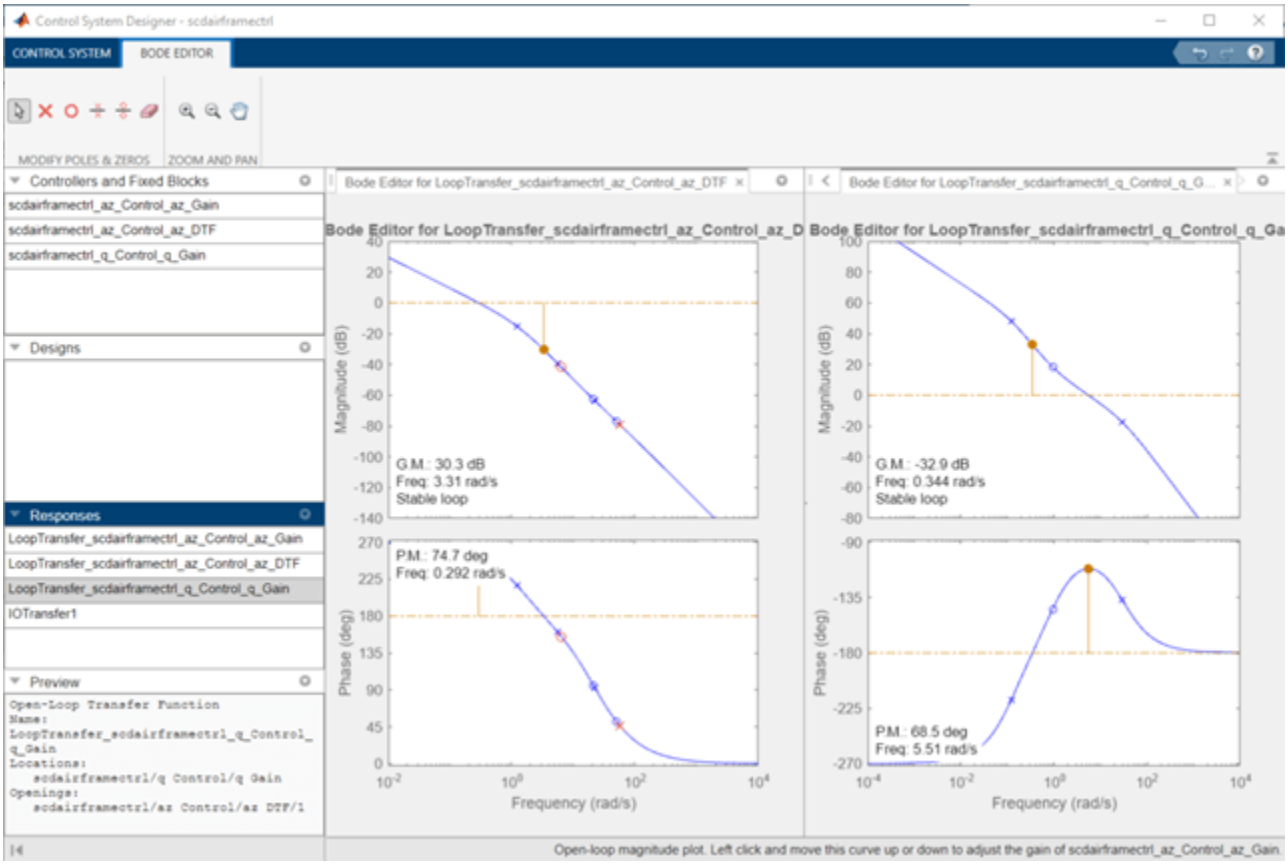
In the data browser, in the **Responses** area, right-click the inner loop response, and select **Open Selection**.



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” (Simulink Control Design).
- 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 Design**.

Complete Design

The following compensator parameters satisfy the design requirements:

- `scdairframectrl/q Control/q Gain`:

  ```
  K_q = 2.7717622
  ```

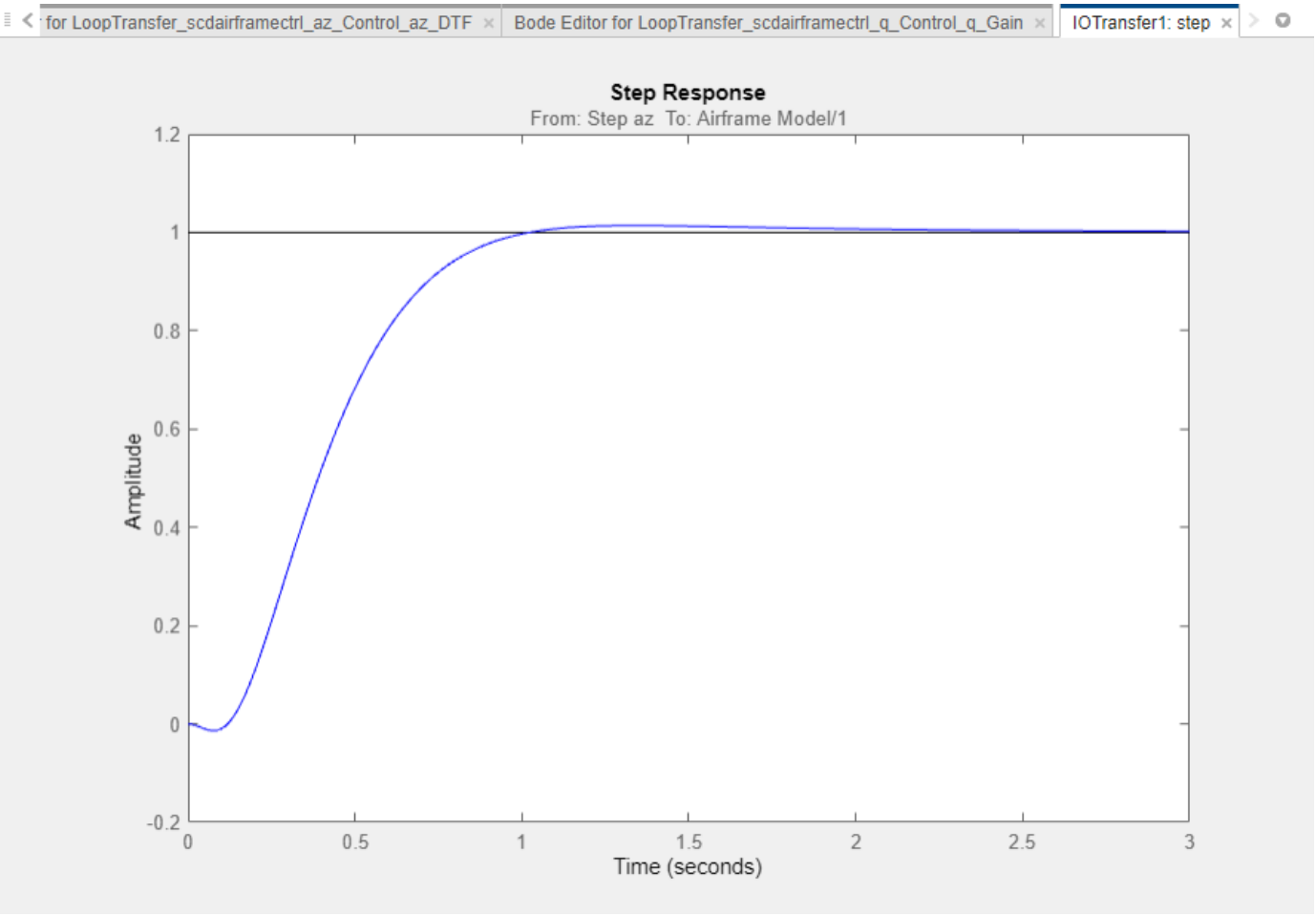
- `scdairframectrl/az Control/az Gain`:

  ```
  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 in the following figure.



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

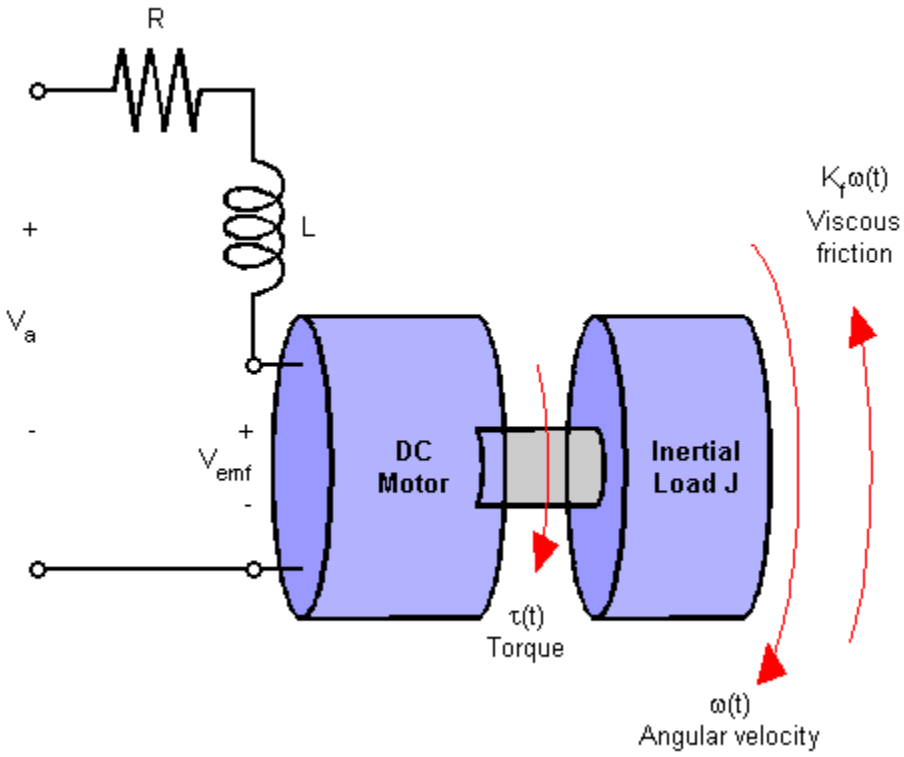
- “Design Multiloop Control System” on page 12-23
- “Control System Designer Tuning Methods” on page 12-5
- “Analyze Designs Using Response Plots” on page 12-95

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.

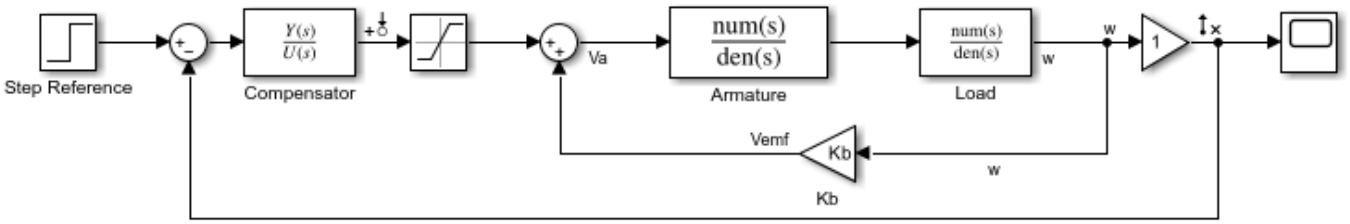
DC Motor Model

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)
```



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

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.

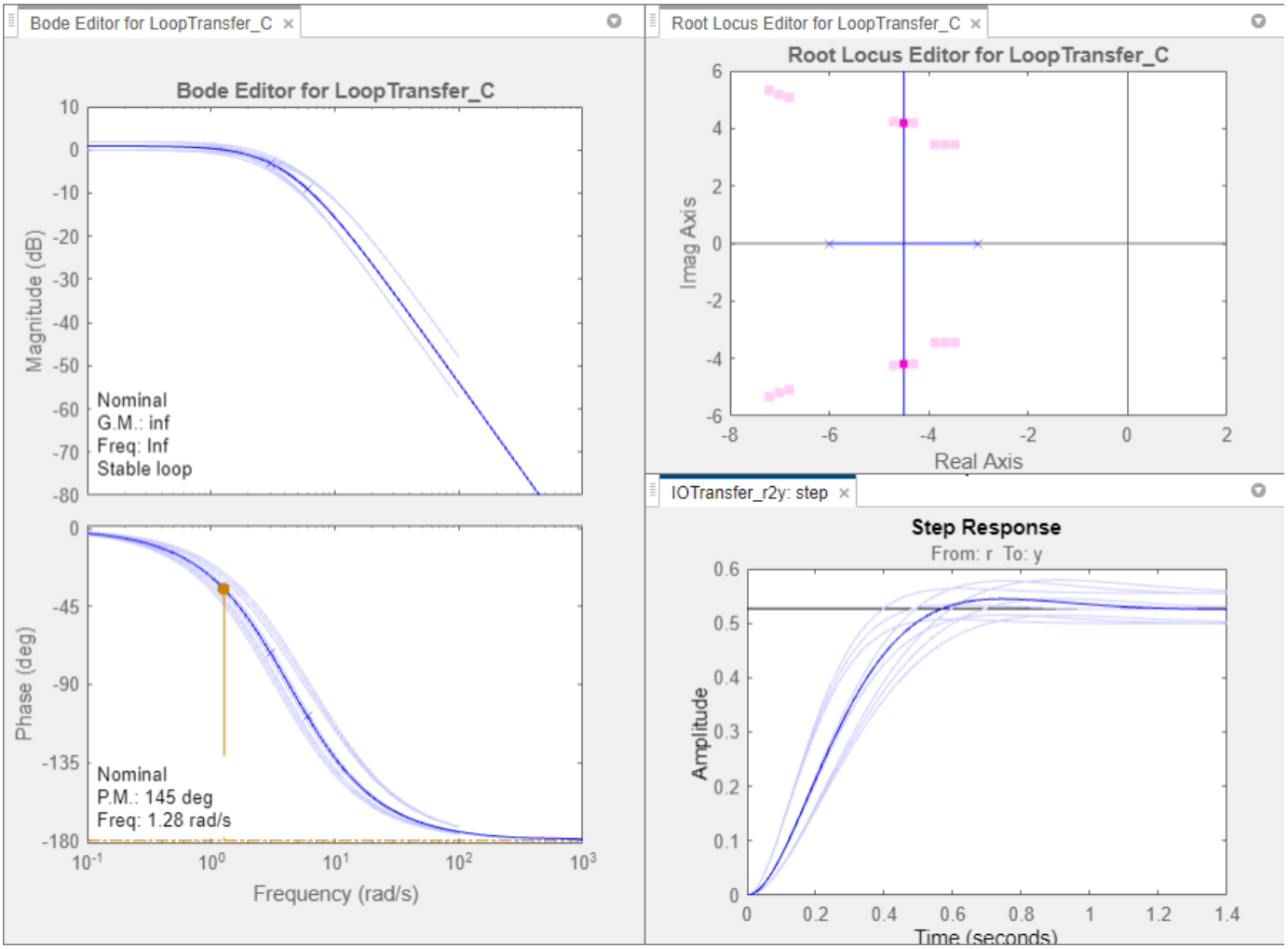
```
sys = linearize(mdl,io,params);
```

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.

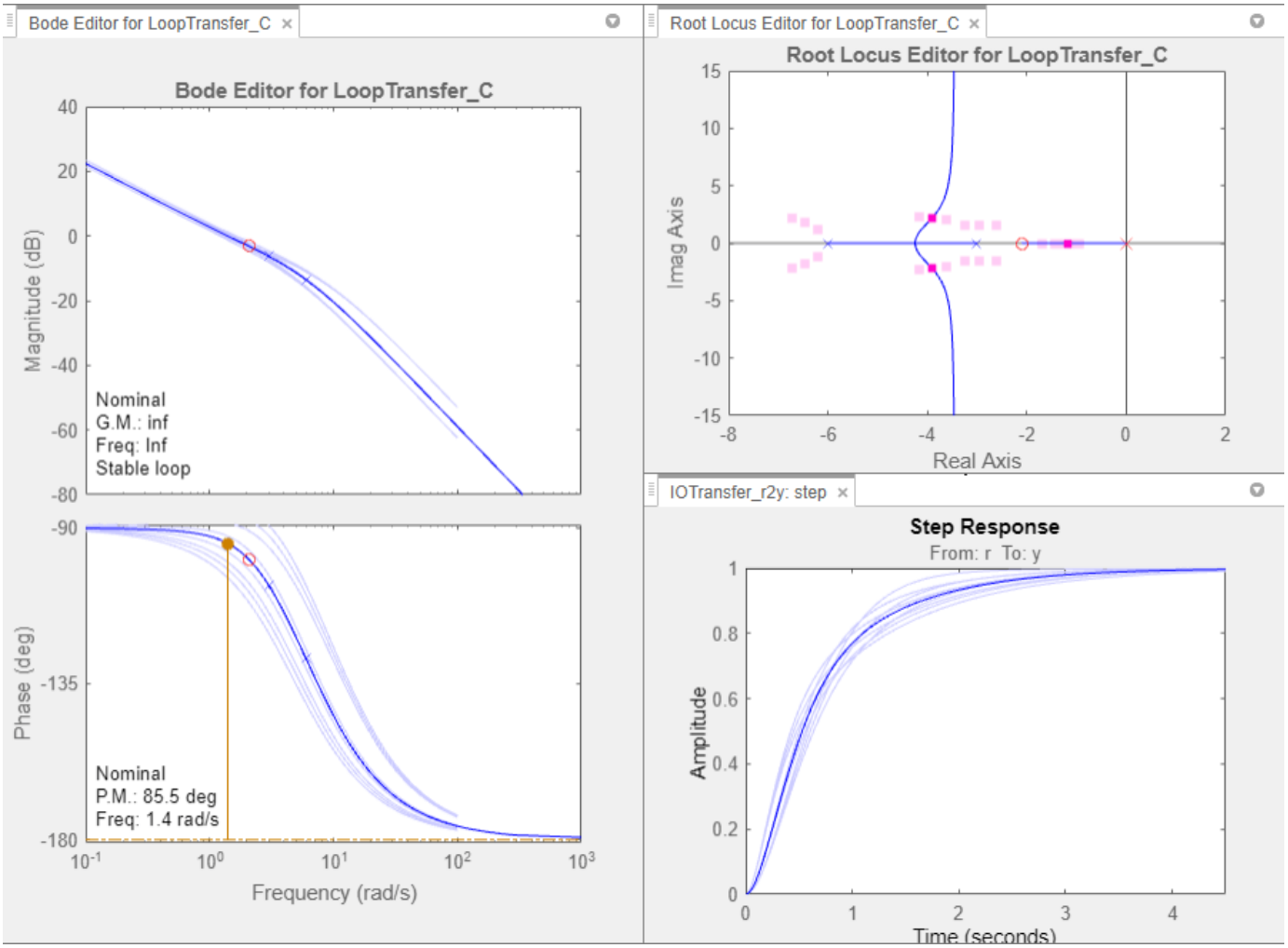
To view the step responses for all the plant models, right-click the **Step Response** plot and select **Multimodel Display > Individual Responses**. The step responses show that reference tracking is not achieved for any of the plant models.

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” on page 12-32.

See Also

Control System Designer

Related Examples

- “Control System Designer Tuning Methods” on page 12-5
- “Analyze Designs Using Response Plots” on page 12-95

Getting Started with the Control System Designer

This example shows how to tune compensators for a feedback control system using **Control System Designer**.

Using **Control System Designer** you can:

1) Define control design requirements on time, frequency, and pole/zero response plots.

2) Tune compensators using:

- Automated design methods, such as PID tuning, IMC, and LQG.
- Graphically tune poles and zeros on design plots, such as Bode and root locus.
- Optimization-based control design to meet time-domain and frequency-domain requirements using Simulink® Design Optimization™.

3) Visualize closed-loop and open-loop responses that dynamically update to display the control system performance.

Compensator Design Problem

For this example, design a compensator for the system

$$G(s) = \frac{1}{s+1}$$

with the following design requirements:

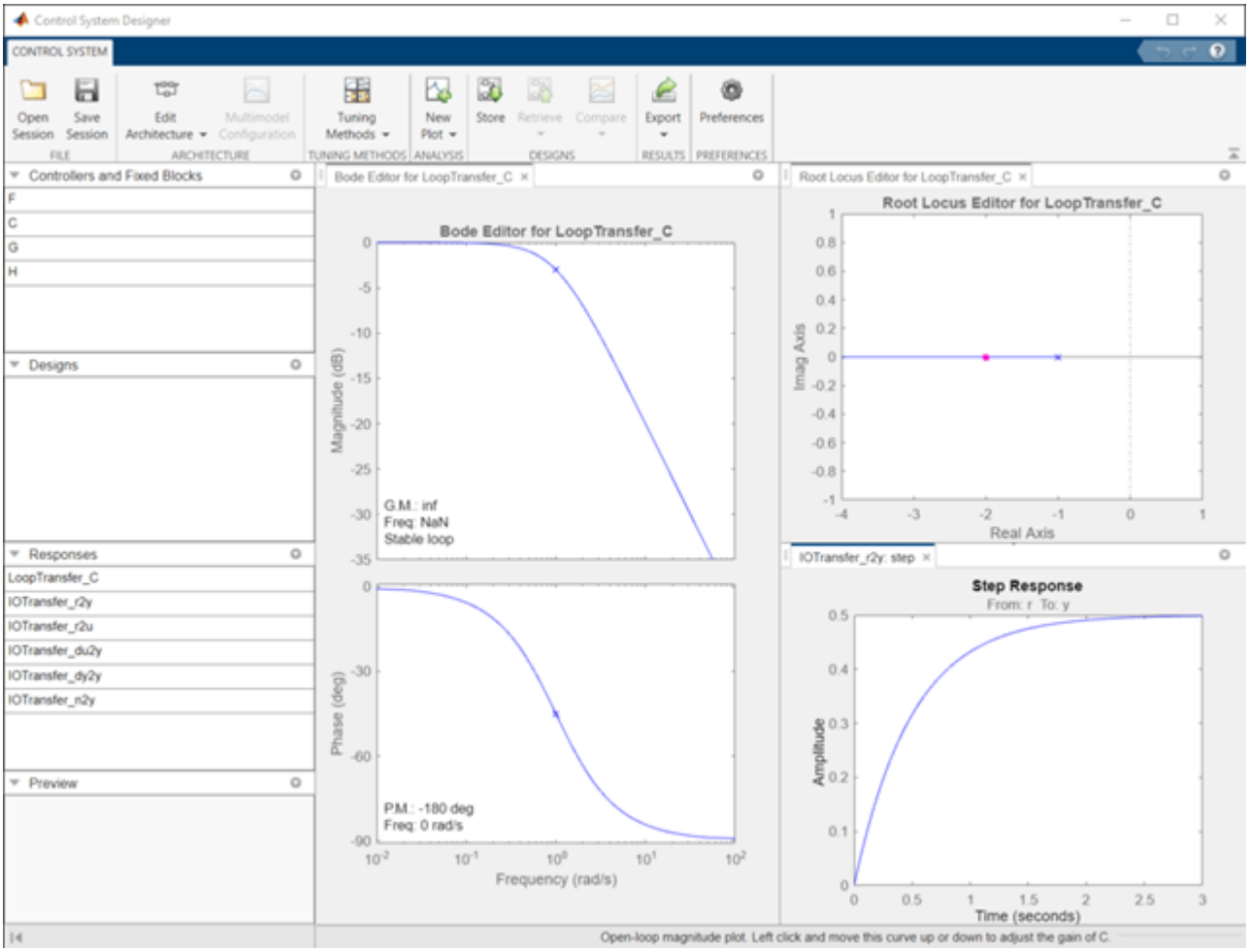
- Zero steady-state error with respect to a step input.
- 80% rise time less than 1 second.
- Settling time less than 2 seconds.
- Maximum overshoot less than 20%.
- Open-loop crossover frequency less than 5 rad/s.

Open Control System Designer

Use the standard feedback structure with the controller in the forward path. This structure is the default **Control System Designer** architecture.

Open **Control System Designer** with the specified plant.

```
controlSystemDesigner(tf(1,[1,1]))
```

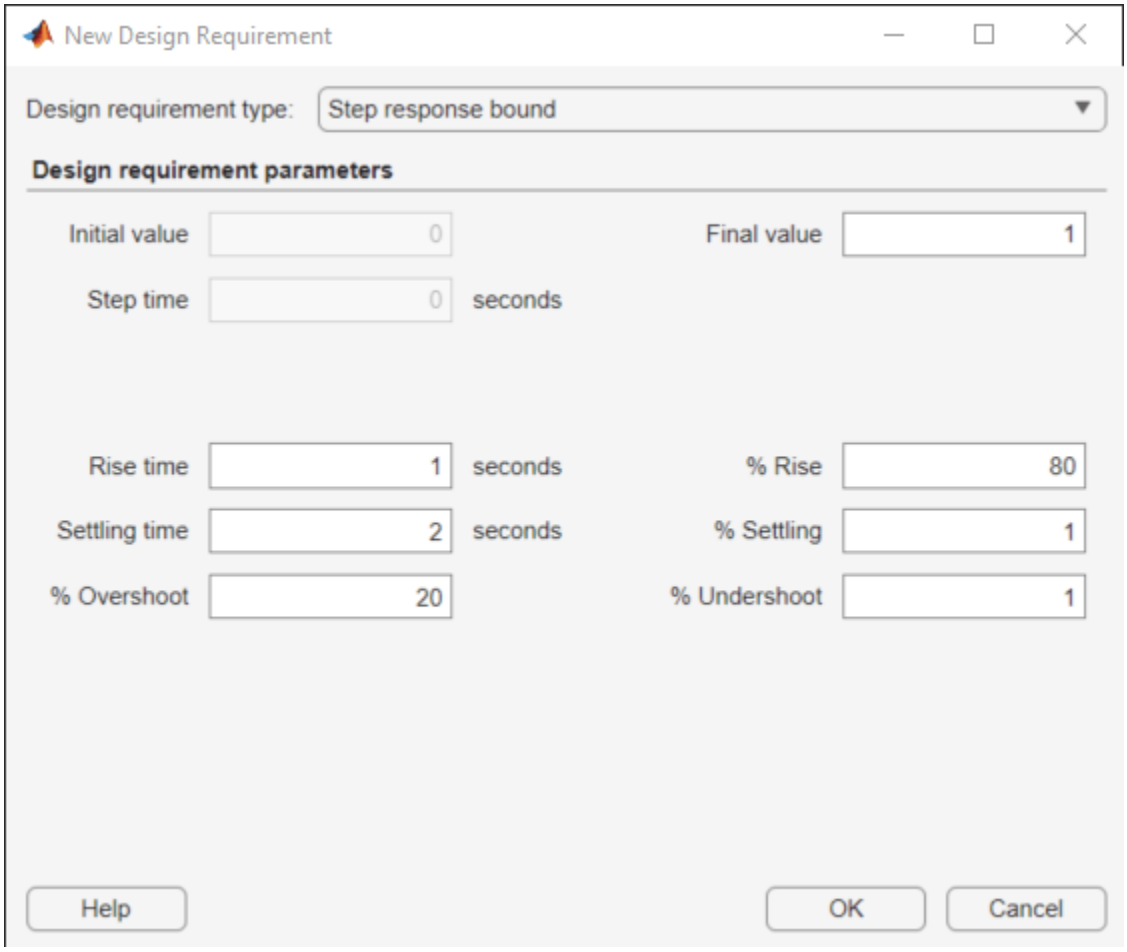


On the **Control System** tab, you can select a compensator tuning method, and create response plots for analyzing your controller performance. You can also store, compare, and export different control system designs.

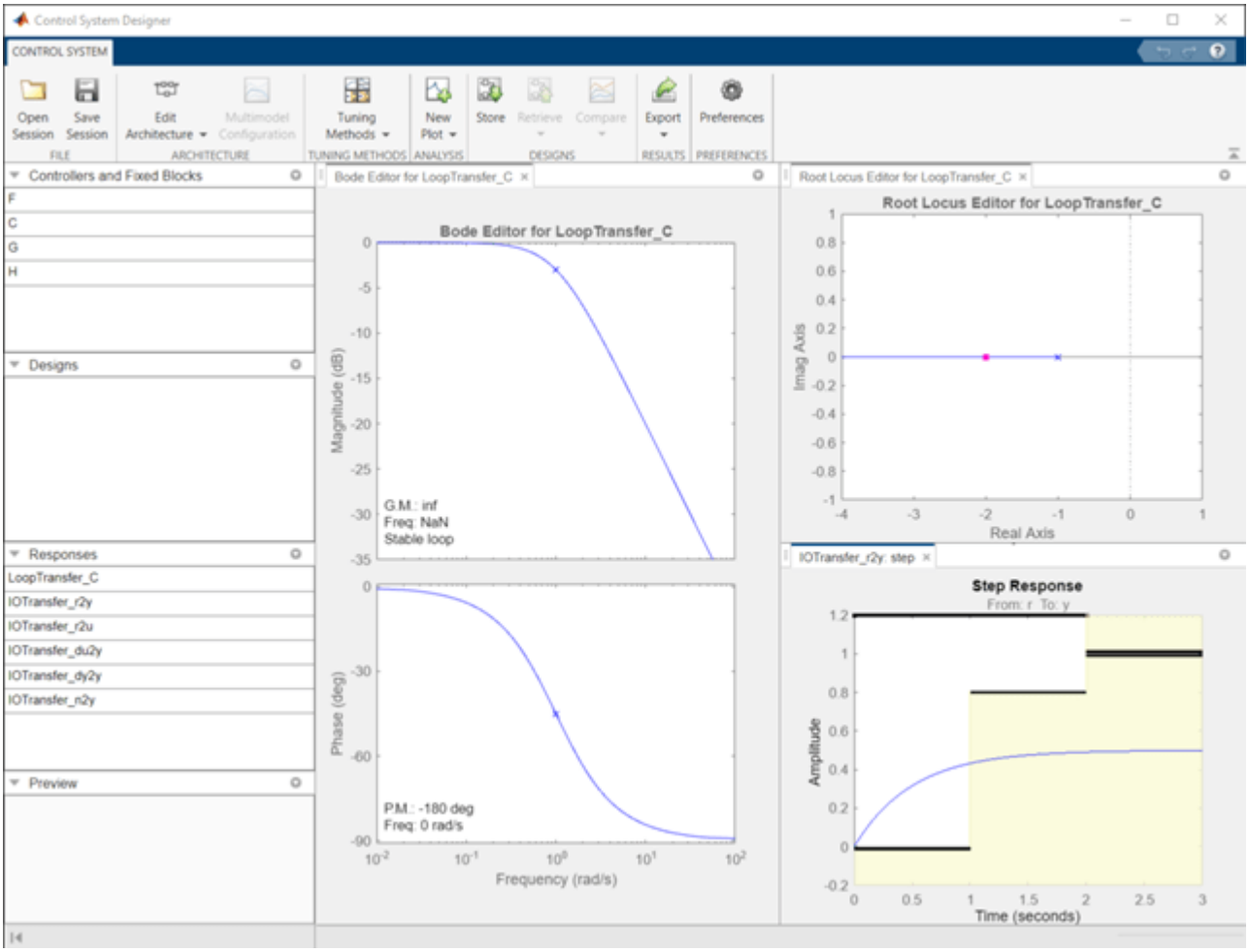
For this example, graphically tune your compensator using the **Root Locus Editor** and open-loop **Bode Editor**, and validate the design using the closed-loop **Step Response**. By default, **Control System Designer** displays these responses when it opens. To add additional response plots, click **New Plot**.

Add Design Requirements

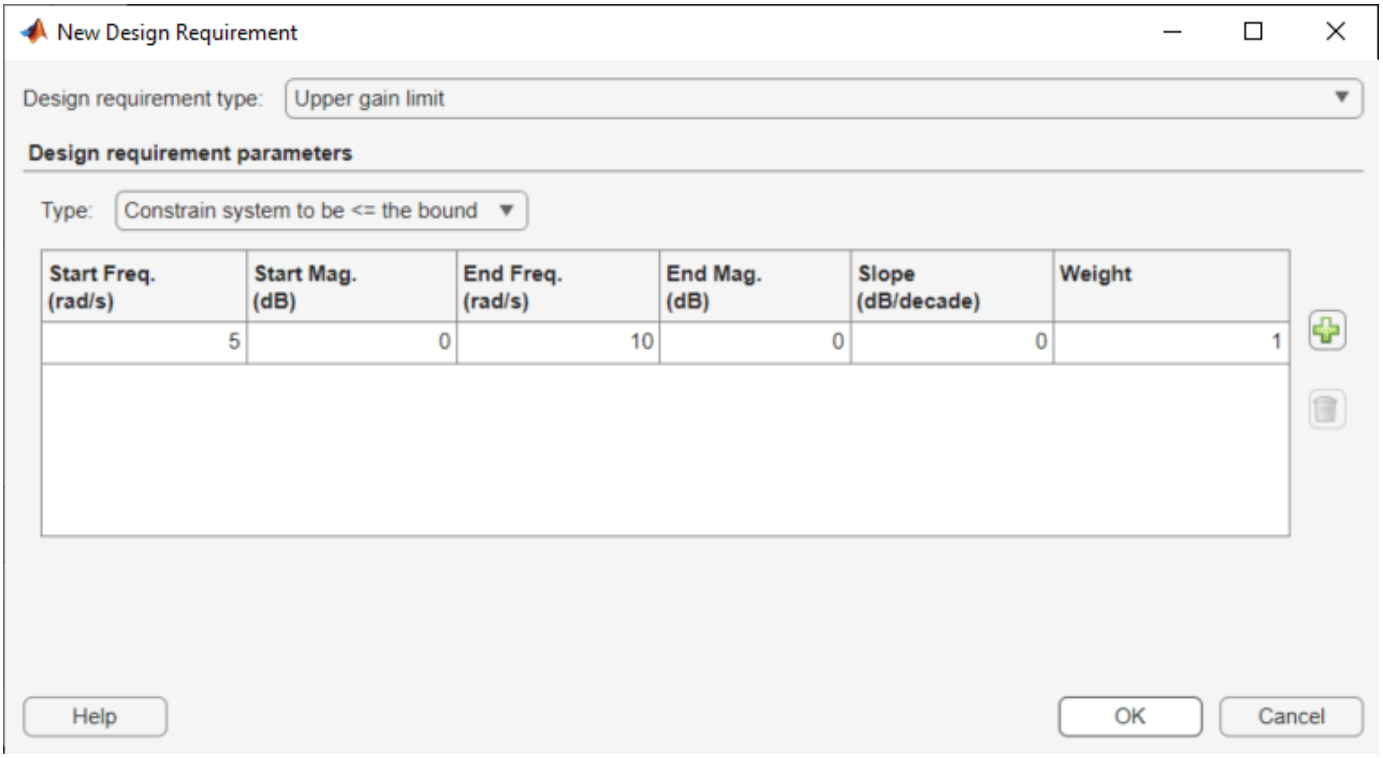
Add the time-domain design requirements to the **Step Response** plot. Right-click the plot area, and select **Design Requirements > New**. In the **Design requirement type** drop-down list, select `Step response bound`. Enter the time-domain design requirements.



Click **OK**. The app adds the design requirement to the step response plot as a shaded exclusion region. To meet the requirement, the step response must remain outside of this region.



To specify the frequency-domain crossover requirement, right-click the **Bode Editor** plot area, and select **Design Requirements > New**. In the **Design requirement type** drop-down list, select `Upper gain limit`, and specify the design requirement.



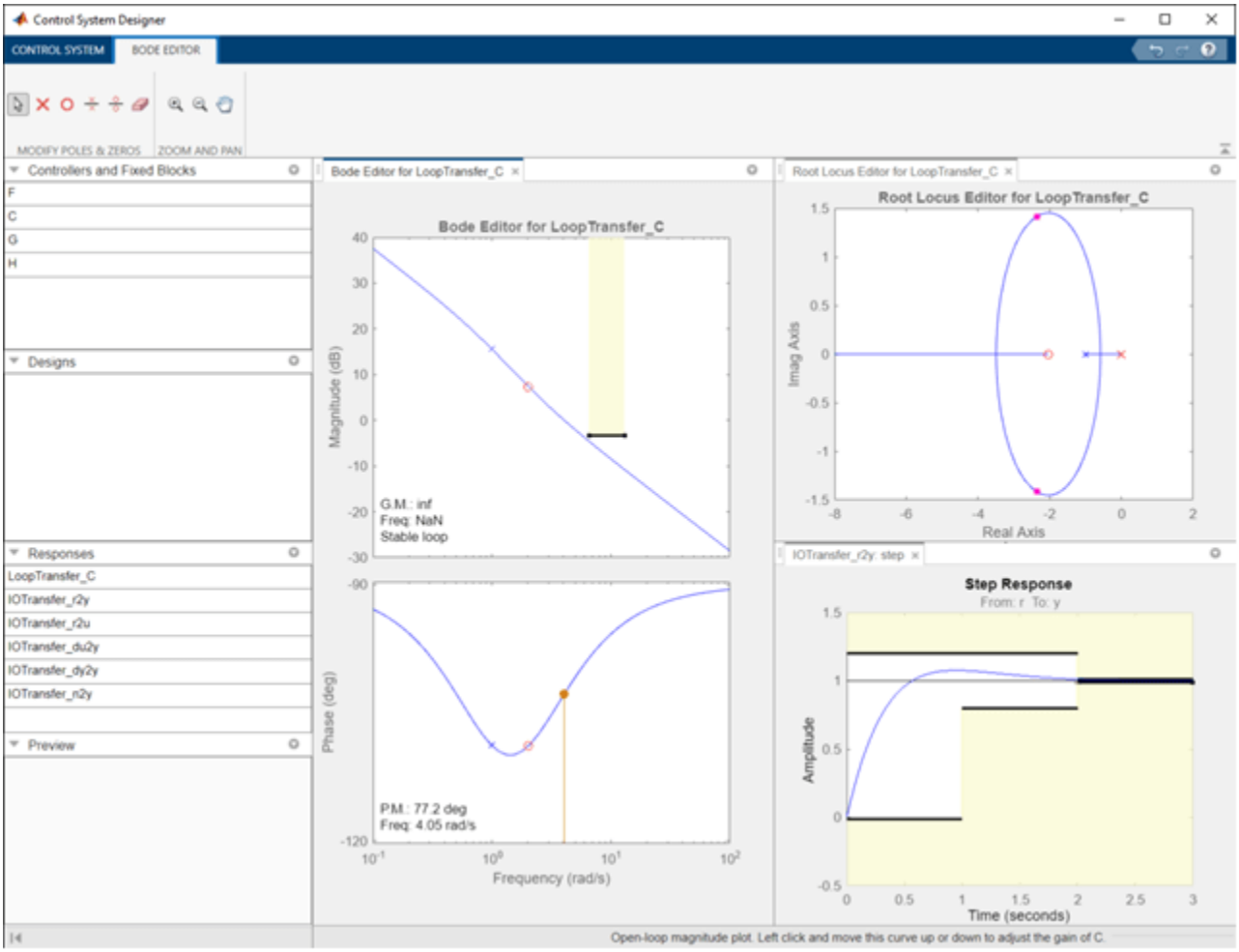
Click **OK**.

Tune Compensator

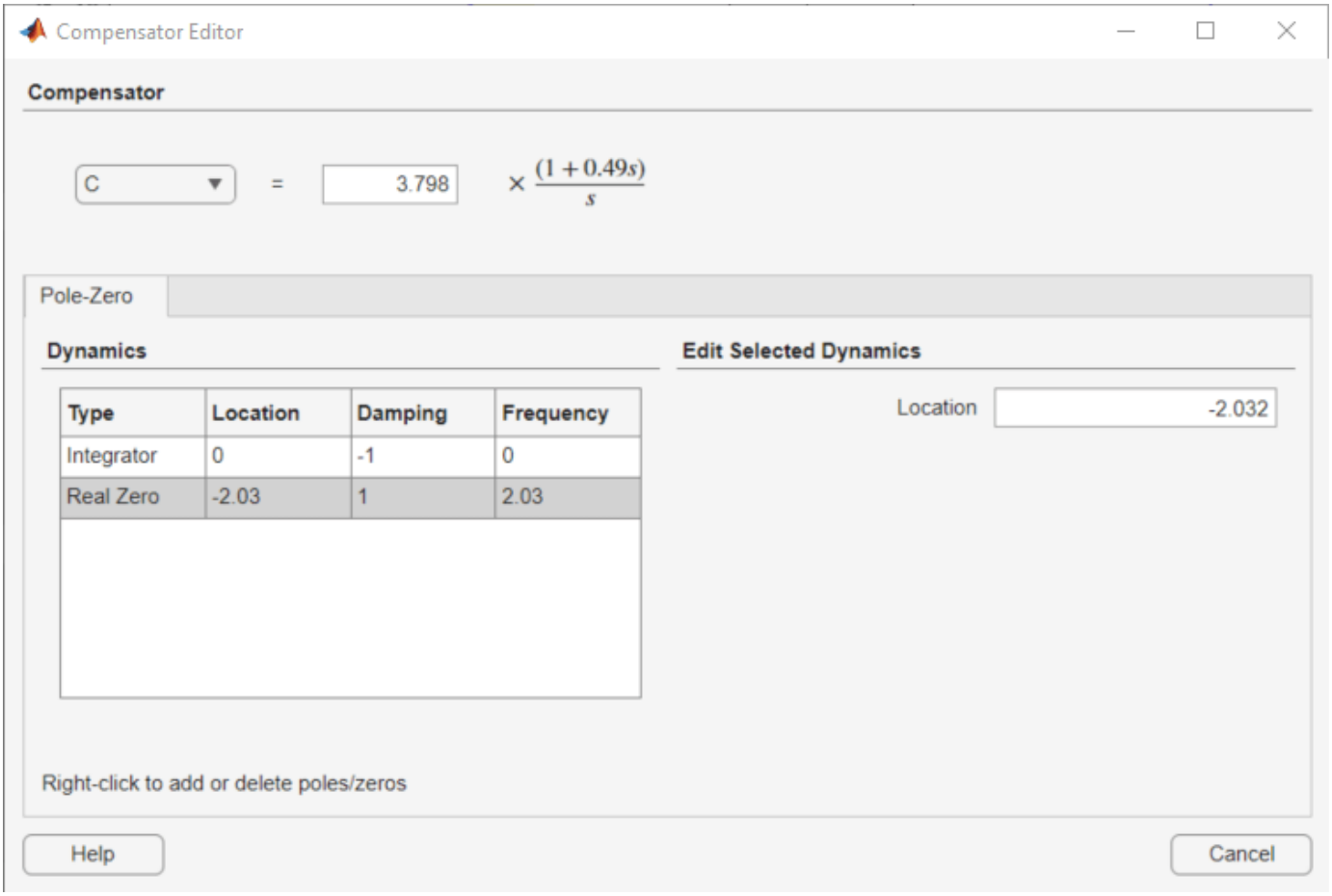
To meet the zero steady-state error design requirement, add an integrator to the compensator. Right-click the **Root Locus Editor** plot area, and select **Add Pole or Zero > Integrator**.

To create a desirable shape for the root locus plot, add a real zero near -2. Right-click the root locus plot area and select **Add Pole or Zero > Real Zero**. In the root locus plot, left-click the real axis near -2.

To create a faster response by increasing the compensator gain, in the **Bode Editor**, drag the magnitude response upward. To satisfy the crossover frequency requirement, keep the response below the exclusion region in the Bode editor.



To view the compensator, right-click in the **Bode Editor** or **Root Locus Editor** plot area, and select **Edit Compensator...**.



You can also tune the compensator parameters using the Compensator Editor dialog box.

Automated Compensator Tuning

In addition to graphical tuning, you can also use automated tuning methods. To select an automated tuning method, click **Tuning Methods**.

- **PID Tuning**, **IMC Tuning**, and **LQG Synthesis** - Compute initial compensator parameters based on tuning specifications such as closed-loop time constants. For an example, see “Design LQG Tracker Using Control System Designer” on page 12-140.
- **Optimization-Based Tuning** - Optimize compensators using both time-domain and frequency-domain design requirements (requires Simulink Design Optimization). For an example, see “DC Motor Controller Tuning” (Simulink Design Optimization).
- **Loop Shaping** - Specify a desired target loop shape (requires Robust Control Toolbox™).

See Also

Control System Designer

Related Examples

- “Control System Designer Tuning Methods” on page 12-5

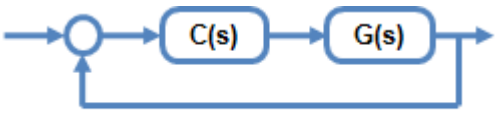
- “Analyze Designs Using Response Plots” on page 12-95

Compensator Design for a Set of Plant Models

This example shows how to design and analyze a controller for multiple plant models using **Control System Designer**.

Acquire a Set of Plant Models

For a typical feedback problem, the controller, `C`, is designed to satisfy some performance objective.



Typically, the dynamics of the plant, `G`, are not known exactly and can vary based on operating conditions. For example, the system dynamics can vary:

- Due to manufacturing tolerances that are typically defined as a range about the nominal value. For example, resistors have a specified tolerance range, such as 5 ohms +/- 1%.
- Operating conditions. For example, aircraft dynamics change based on altitude and speed.

When designing controllers for these types of systems, the performance objectives must be satisfied for all variations of the system.

You can model such systems as a set of LTI models stored in an LTI array. You can then use Control System Designer to design a controller for a nominal plant from the array and analyze the controller design for the entire set of plants.

The following list shows commands for creating an array of LTI models:

Control System Toolbox™:

- Functions: `stack`, `tf`, `zpk`, `ss`, `frd`

Simulink® Control Design™:

- Functions: `frestimate` (Simulink Control Design), `linearize` (Simulink Control Design)
- Example: “Reference Tracking of DC Motor with Parameter Variations” (Simulink Control Design).

Robust Control Toolbox™:

- Functions: `uss` (Robust Control Toolbox), `usample` (Robust Control Toolbox), `usubs` (Robust Control Toolbox).

System Identification Toolbox™:

- Functions: `pem` (System Identification Toolbox), `oe` (System Identification Toolbox), `arx` (System Identification Toolbox).

Create LTI Array

In this example, the plant model is the second-order system:

$$G(s) = \frac{\omega_n^2}{s^2 + 2\zeta\omega_n s + \omega_n^2}$$

where

$\omega_n = (1, 1.5, 2)$ and $\zeta = (.2, .5, .8)$.

Construct an LTI array for the combinations of ζ and ω_n.

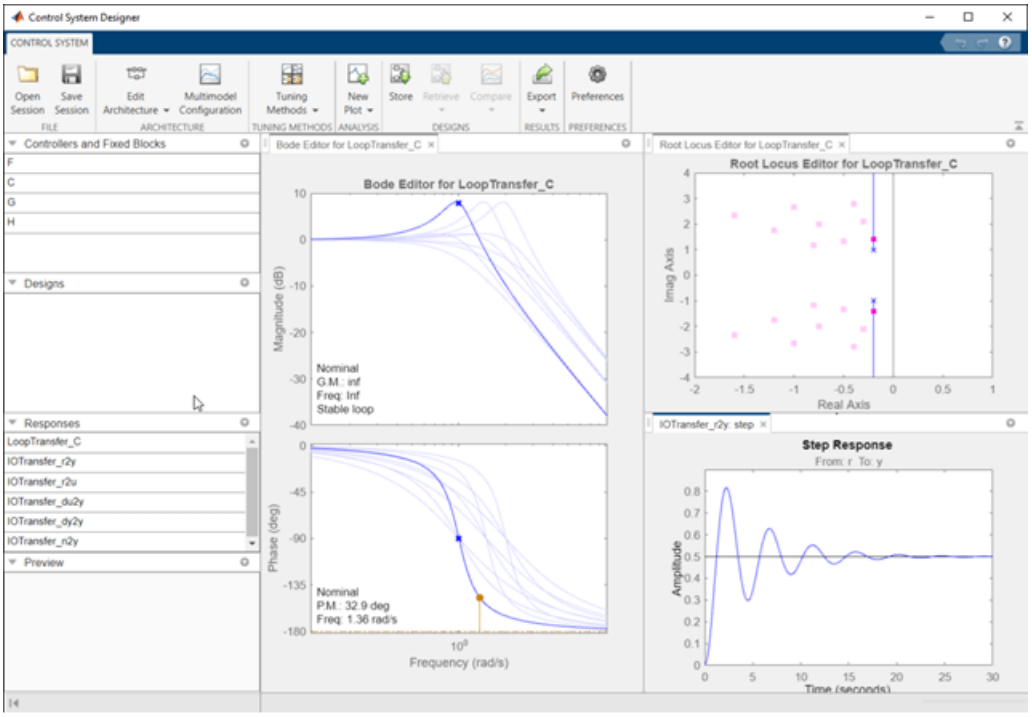
```
wn = [1,1.5,2];
zeta = [.2,.5,.8];
ct = 1;
for ct1 = 1:length(wn)
    for ct2 = 1:length(zeta)
        zetai = zeta(ct2);
        wni = wn(ct1);
        G(1,1,ct) = tf(wni^2,[1,2*zetai*wni,wni^2]);
        ct = ct+1;
    end
end

size(G)
```

```
9x1 array of transfer functions.
Each model has 1 outputs and 1 inputs.
```

Open Control System Designer

Start the Control System Designer.

```
controlSystemDesigner(G)
```



The app opens with Bode and root locus open-loop editors open along with a step response plot.

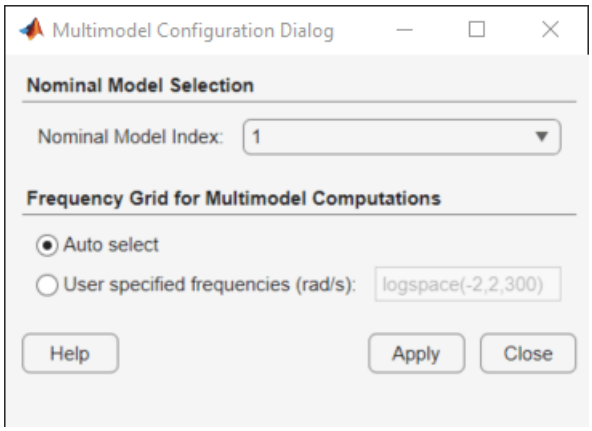
By default, the nominal model used for design is the first element in the LTI array.

- The root locus editor displays the root locus for the nominal model and the closed-loop pole locations associated with the set of plants.
- The Bode editor displays both the nominal model response and responses of the set of plants.

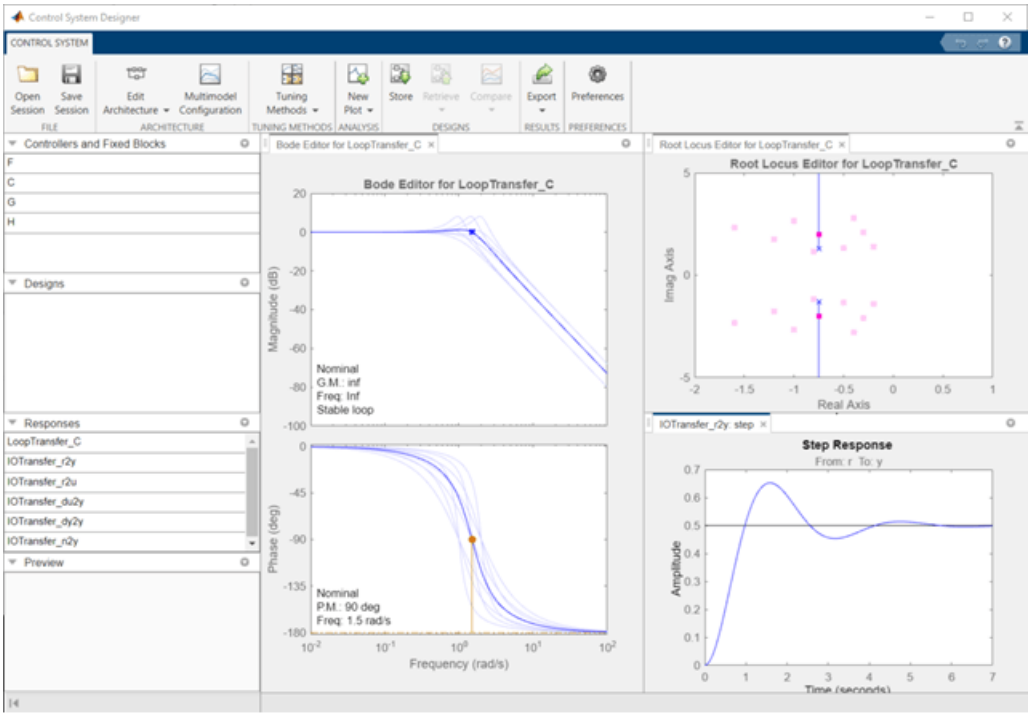
Using these editors, you can interactively tune the gain, poles, and zeros of the compensator, while simultaneously visualizing the effect on the set of plants.

Change the Nominal Model

To change the nominal model, in the app, click **Multimodel Configuration**.



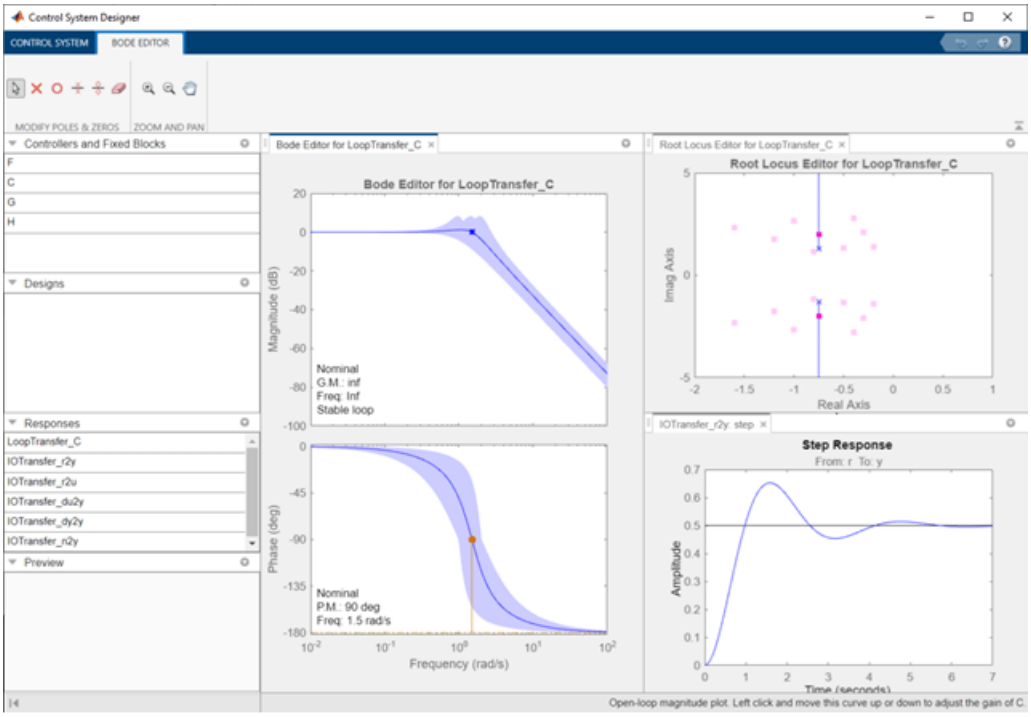
To select the fifth model in the array as the nominal model, in the Multimodel Configuration dialog box, set the **Nominal Model Index** to 5. The app response plots update automatically.



Options for Plotting Responses

The response plots always show the response of the nominal model. To view the other model responses, right-click the plot area and select:

- **Multimodel Display > Individual Responses** to view the response for each model.
- **Multimodel Display > Bounds** to view an envelope that encapsulates all of the responses.



See Also

Control System Designer

Related Examples

- “Control System Designer Tuning Methods” on page 12-5
- “Analyze Designs Using Response Plots” on page 12-95

Programmatically Initializing the Control System Designer

This example shows how to configure **Control System Designer** from the command line and how to create functions to customize the startup of a **Control System Designer** session.

Control System Designer Configurations

Control System Designer has six available feedback control system configurations:

1) Standard feedback loop with the compensator in the forward path and a prefilter.

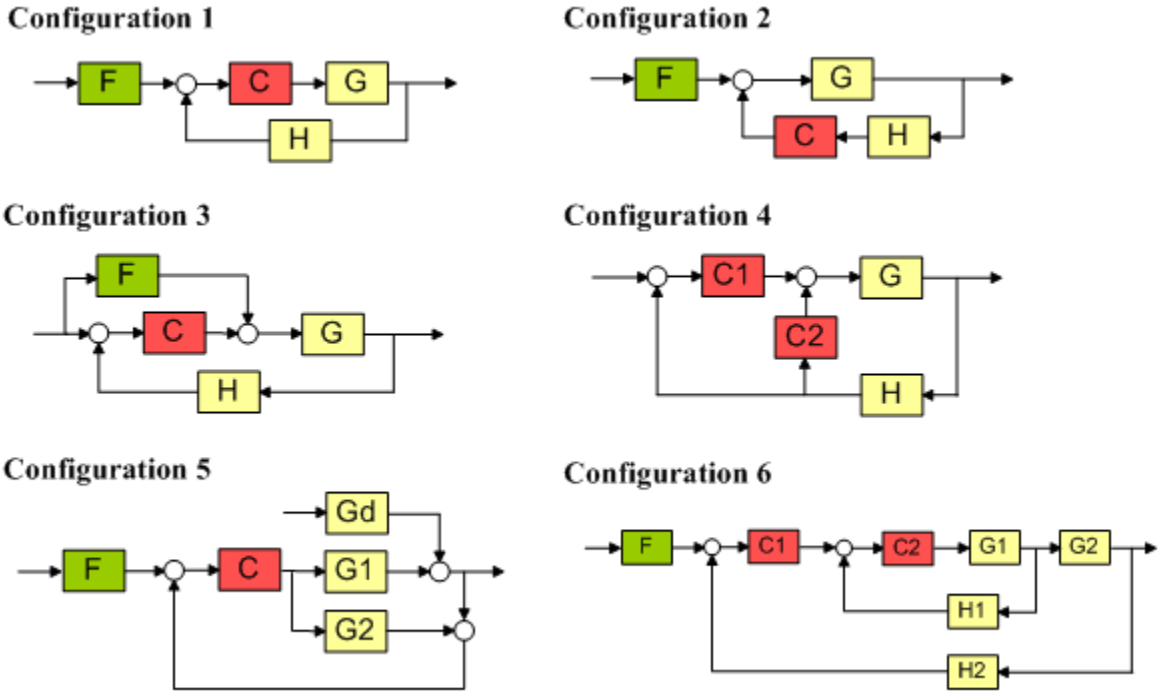
2) Standard feedback loop with the compensator in the feedback path and a prefilter.

3) Feedforward compensation and a feedback loop with a compensator in the forward path. This configuration is often used to attenuate disturbances that can be measured before they act on the system.

4) Nested multiloop design configuration. This configuration provides the ability to separate the design into steps by isolating portions of the control loops.

5) The standard Internal Model Control (IMC) structure.

6) Cascaded multiloop design configuration. This configuration provides the ability to separate the design into steps by isolating portions of the control loops.



By default, **Control System Designer** is initialized with configuration 1. You can change this within the app. Alternatively, you can initialize **Control System Designer** from command line.

Initialize Control System Designer

For this example, create a design initialization structure with the following settings:

- Feedback configuration 4
- Plant `G` with a value of tf(1,[1,1])
- Root locus and bode editors for the outer open-loop
- Nichols editor for the inner open-loop

Create a design initialization structure for configuration 4 using the `sisoinit` command.

```
s = sisoinit(4)
```

```
             Name: ''
    Configuration: 4
      Description: 'Design snapshot.'
     FeedbackSign: [2x1 double]
            Input: {4x1 cell}
           Output: {2x1 cell}
         LoopView: [10x1 sisodata.looptransfer]
                G: [1x1 sisodata.system]
                H: [1x1 sisodata.system]
               C1: [1x1 sisodata.TunedZPKSnapshot]
               C2: [1x1 sisodata.TunedZPKSnapshot]
              OL1: [1x1 sisodata.TunedLoopSnapshot]
              OL2: [1x1 sisodata.TunedLoopSnapshot]
```

In the initialization structure, `s`, the system model components are:

- Outer-loop compensator - `C1`
- Inner-loop compensator - `C2`
- Plant dynamics - `G`
- Sensor dynamics - `H`

The loop editor configurations for the system are:

- Outer loop - OL1
- Inner loop - OL2

Specify the value of the plant.

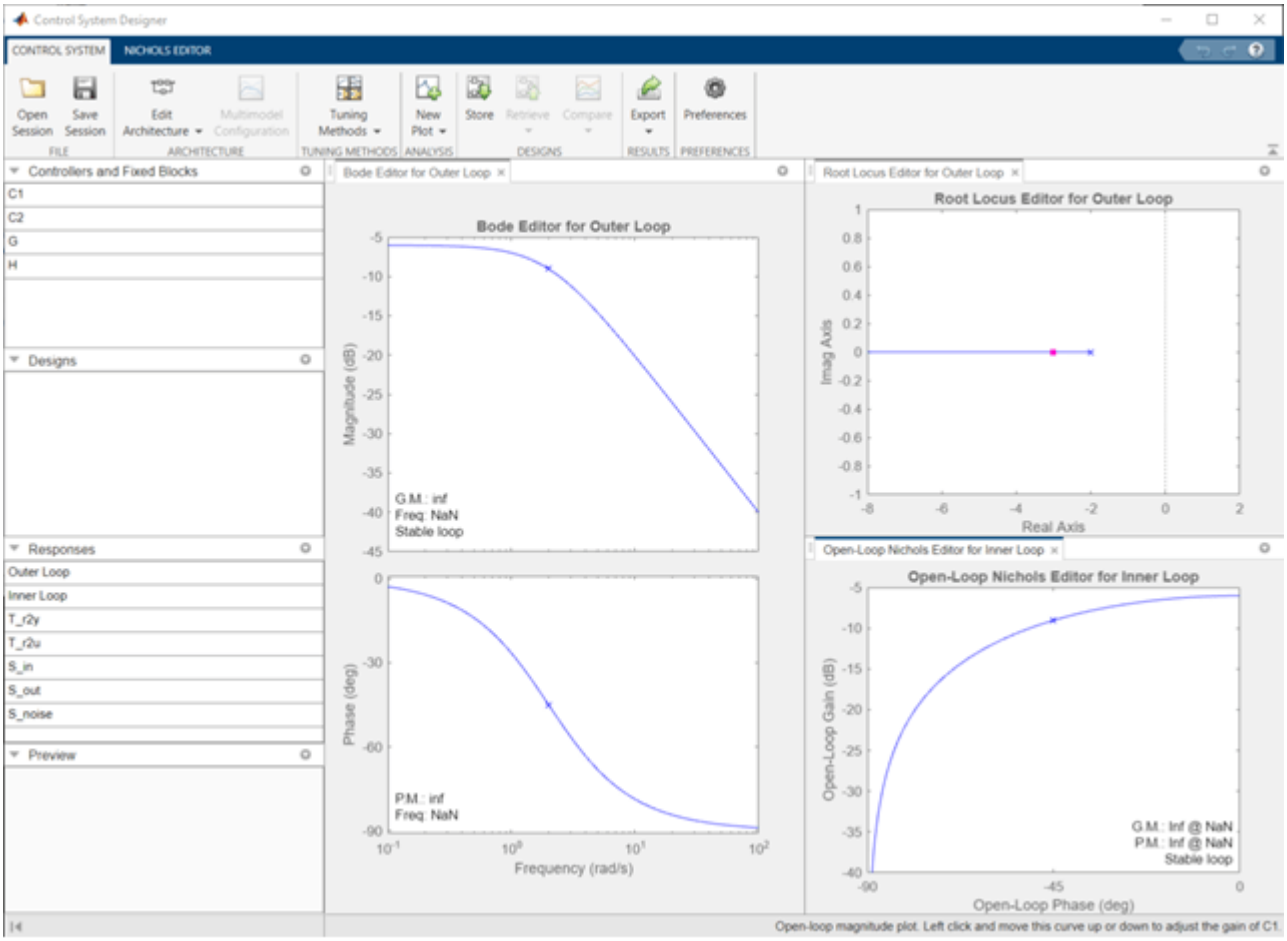
```
s.G.Value = tf(1,[1,1]);
```

Specify the editors to use for each open loop and meaningful loop names.

```
s.OL1.Name = 'Outer Loop';
s.OL1.View = {'rlocus','bode'};
s.OL2.Name = 'Inner Loop';
s.OL2.View = {'nichols'};
```

Open **Control System Designer** using the initialization structure.

```
controlSystemDesigner(s)
```



Create Custom Initialization Function

Creating a custom initialization function is useful for starting **Control System Designer** in a configuration that you use often. For example, the following initialization function creates an initialization structure using specified plant dynamics.

```
type mycustomcontrolsysdesignerfcn
```

```
function mycustomcontrolsysdesignerfcn(G)
% mycustomcontrolsysdesignerfcn(G)
%
% Create the following Control System Designer session:
%   1) Configuration 4 with the plant specified by G
%   2) Root locus and bode editors for the outer-loop
%   3) Bode editor for the inner-loop.

%   Copyright 1986-2005 The MathWorks, Inc.

% Create initialization object with configuration 4
s = sisoinit(4);

% Set the value of the plant
s.G.Value = G;
```

```
% Specify the editors for the Open-Loop Responses
s.OL1.View = {'rlocus','bode'};
s.OL2.View = {'nichols'};

controlSystemDesigner(s)
```

To open **Control System Designer** using this function, enter the following command.

```
mycustomcontrolsysdesignerfcn(G)
```

See Also

Control System Designer

Related Examples

- “Control System Designer Tuning Methods” on page 12-5
- “Analyze Designs Using Response Plots” on page 12-95

DC Motor Control

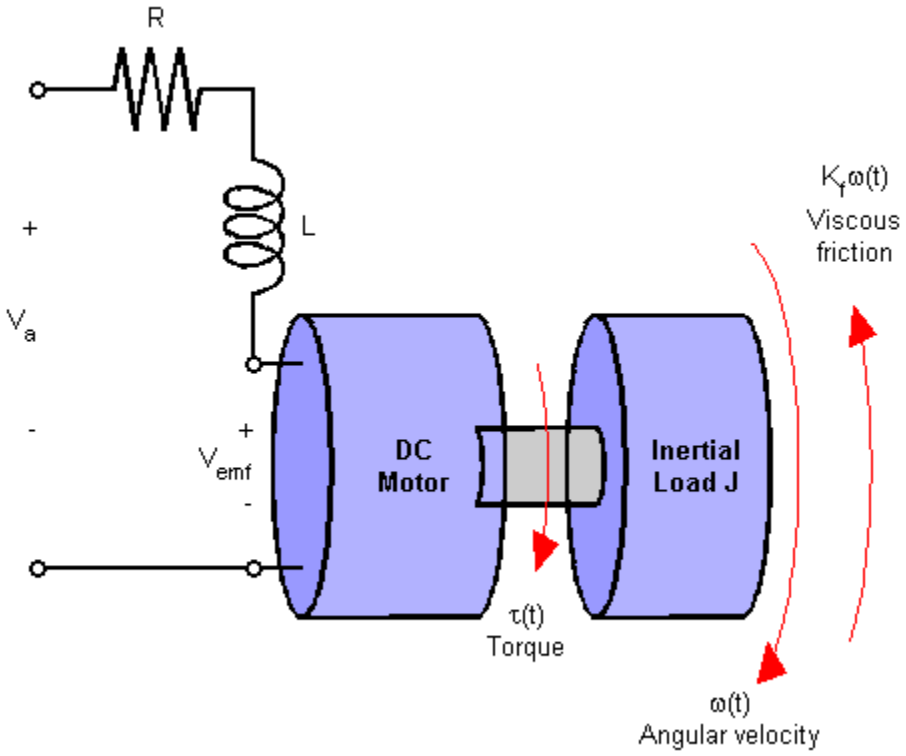
This example shows the comparison of three DC motor control techniques for tracking setpoint commands and reducing sensitivity to load disturbances:

- feedforward command
- integral feedback control
- LQR regulation

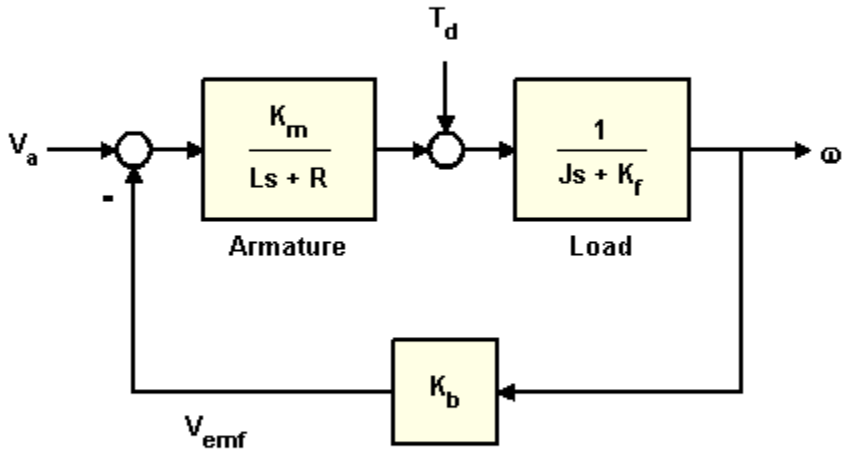
See "Getting Started:Building Models" for more details about the DC motor model.

Problem Statement

In armature-controlled DC motors, the applied voltage Va controls the angular velocity w of the shaft.



This example shows two DC motor control techniques for reducing the sensitivity of w to load variations (changes in the torque opposed by the motor load).



A simplified model of the DC motor is shown above. The torque Td models load disturbances. You must minimize the speed variations induced by such disturbances.

For this example, the physical constants are:

```
R = 2.0;                % Ohms
L = 0.5;                % Henrys
Km = 0.1;               % torque constant
Kb = 0.1;               % back emf constant
Kf = 0.2;               % Nms
J = 0.02;               % kg.m^2/s^2
```

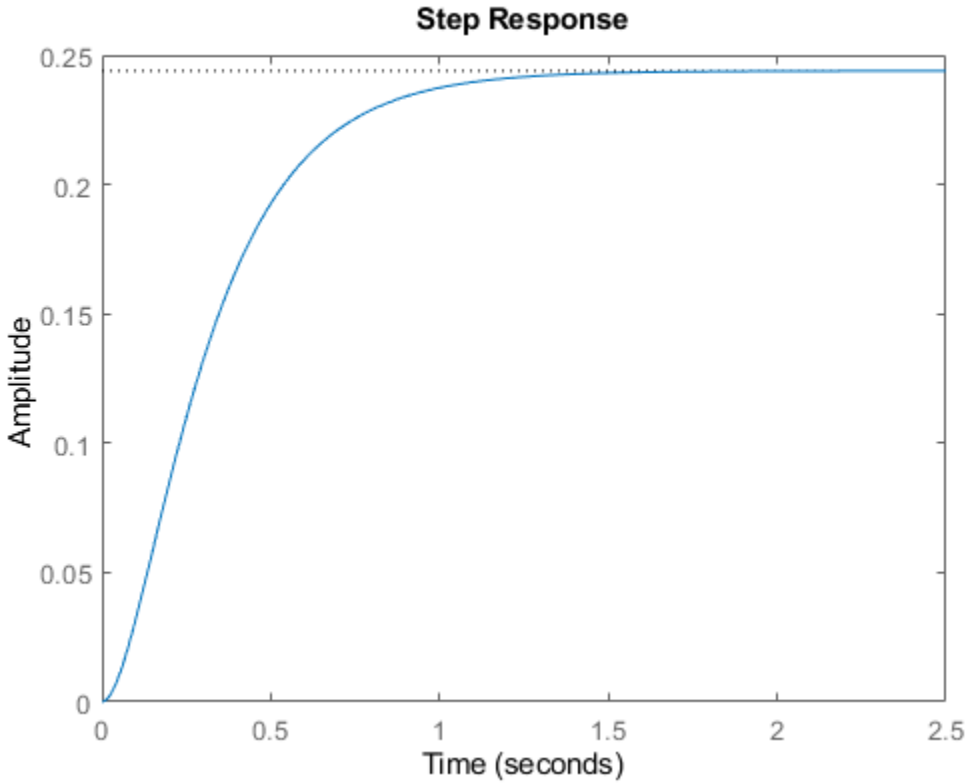
First construct a state-space model of the DC motor with two inputs (Va,Td) and one output (w):

```
h1 = tf(Km,[L R]);            % armature
h2 = tf(1,[J Kf]);            % eqn of motion

dcm = ss(h2) * [h1 , 1];      % w = h2 * (h1*Va + Td)
dcm = feedback(dcm,Kb,1,1);   % close back emf loop
```

Note: Compute with the state-space form to minimize the model order.

Now plot the angular velocity response to a step change in voltage Va:

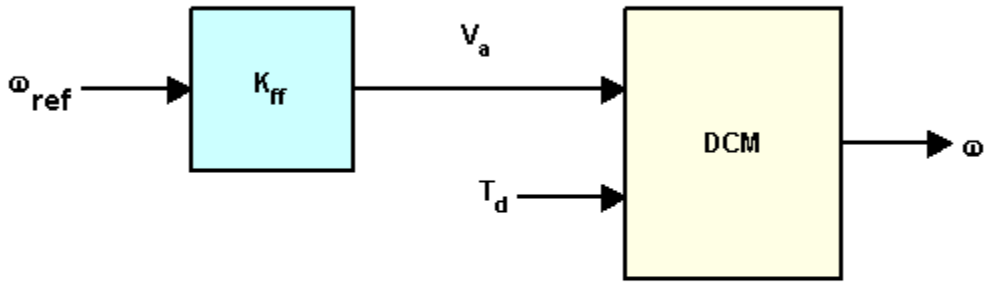
```
stepplot(dcm(1));
```



Right-click on the plot and select "Characteristics:Settling Time" to display the settling time.

Feedforward DC Motor Control Design

You can use this simple feedforward control structure to command the angular velocity w to a given value w_ref.



Feedforward Control

The feedforward gain Kff should be set to the reciprocal of the DC gain from Va to w.

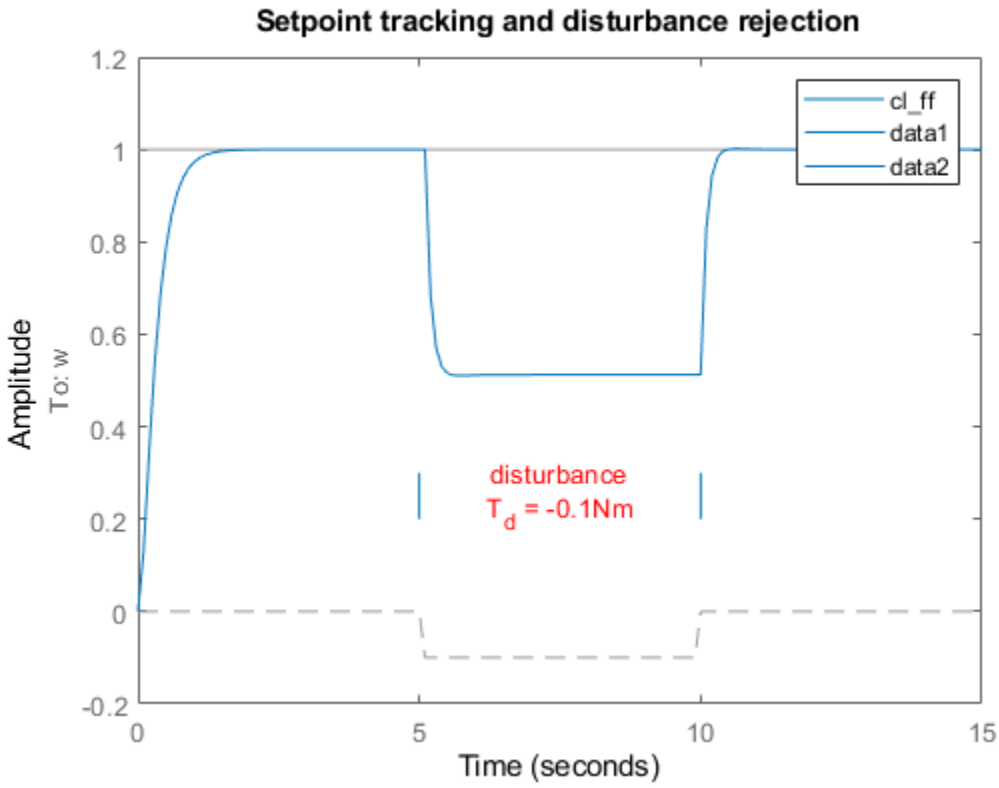
```
Kff = 1/dcgain(dcm(1))
```

```
Kff =

    4.1000
```

To evaluate the feedforward design in the face of load disturbances, simulate the response to a step command w_ref=1 with a disturbance Td = -0.1Nm between t=5 and t=10 seconds:

```
t = 0:0.1:15;
Td = -0.1 * (t>5 & t<10);       % load disturbance
u = [ones(size(t)) ; Td];       % w_ref=1 and Td

cl_ff = dcm * diag([Kff,1]);    % add feedforward gain
cl_ff.InputName = {'w_ref','Td'};
cl_ff.OutputName = 'w';

h = lsimplot(cl_ff,u,t);
title('Setpoint tracking and disturbance rejection')
legend('cl\_ff')

% Annotate plot
line([5,5],[.2,.3]);
line([10,10],[.2,.3]);
```

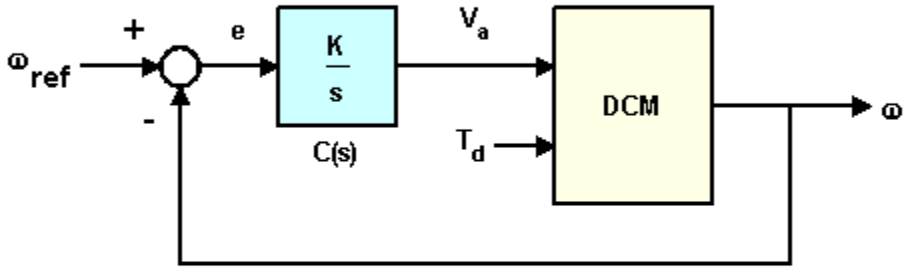
```
text(7.5,.25,{'disturbance','T_d = -0.1Nm'},...
             'vertic','middle','horiz','center','color','r');
```



Clearly feedforward control handles load disturbances poorly.

Feedback DC Motor Control Design

Next try the feedback control structure shown below.



Feedback Control

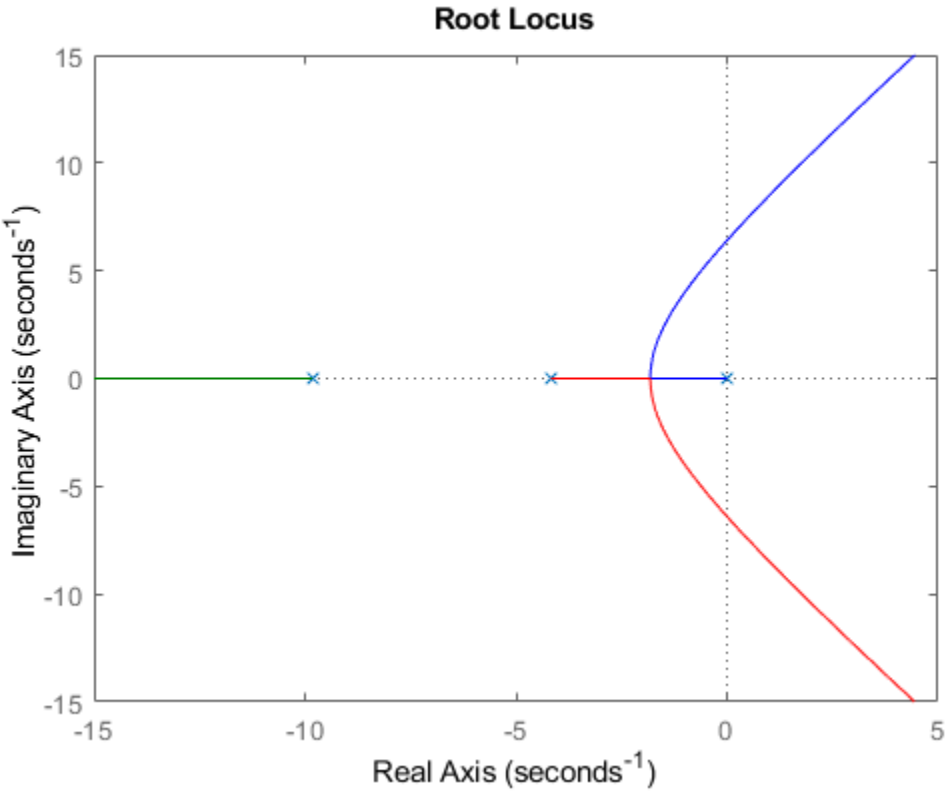
To enforce zero steady-state error, use integral control of the form

```
C(s) = K/s
```

```
where K is to be determined.
```

To determine the gain K, you can use the root locus technique applied to the open-loop 1/s * transfer(Va->w):

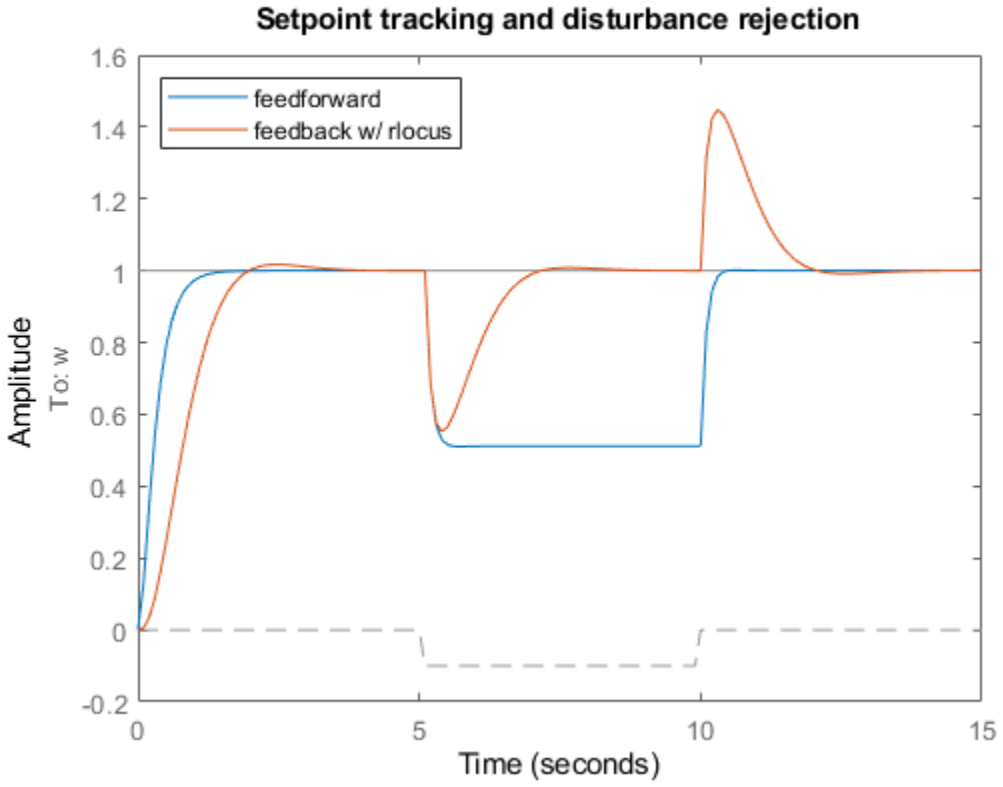
```
h = rlocusplot(tf(1,[1 0]) * dcm(1));
setoptions(h,'FreqUnits','rad/s');
xlim([-15 5]);
ylim([-15 15]);
```



Click on the curves to read the gain values and related info. A reasonable choice here is K = 5. The Control System Designer app is an interactive UI for performing such designs.

Compare this new design with the initial feedforward design on the same test case:

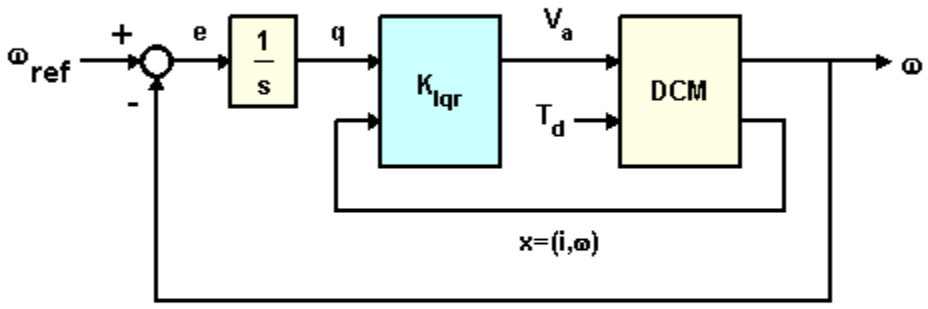
```
K = 5;
C = tf(K,[1 0]);            % compensator K/s

cl_rloc = feedback(dcm * append(C,1),1,1,1);
h = lsimplot(cl_ff,cl_rloc,u,t);
cl_rloc.InputName = {'w_ref','Td'};
cl_rloc.OutputName = 'w';
title('Setpoint tracking and disturbance rejection')
legend('feedforward','feedback w/ rlocus','Location','NorthWest')
```



The root locus design is better at rejecting load disturbances.

LQR DC Motor Control Design

To further improve performance, try designing a linear quadratic regulator (LQR) for the feedback structure shown below.



LQR Control

In addition to the integral of error, the LQR scheme also uses the state vector x=(i,w) to synthesize the driving voltage Va. The resulting voltage is of the form

```
Va = K1 * w + K2 * w/s + K3 * i
```

where i is the armature current.

For better disturbance rejection, use a cost function that penalizes large integral error, e.g., the cost function

$$C = \int_0^\infty (20q(t)^2 + \omega(t)^2 + 0.01V_a(t)^2)dt$$

where

$$q(s) = \omega(s)/s.$$

The optimal LQR gain for this cost function is computed as follows:

```
dc_aug = [1 ; tf(1,[1 0])] * dcm(1); % add output w/s to DC motor model

K_lqr = lqry(dc_aug,[1 0;0 20],0.01);
```

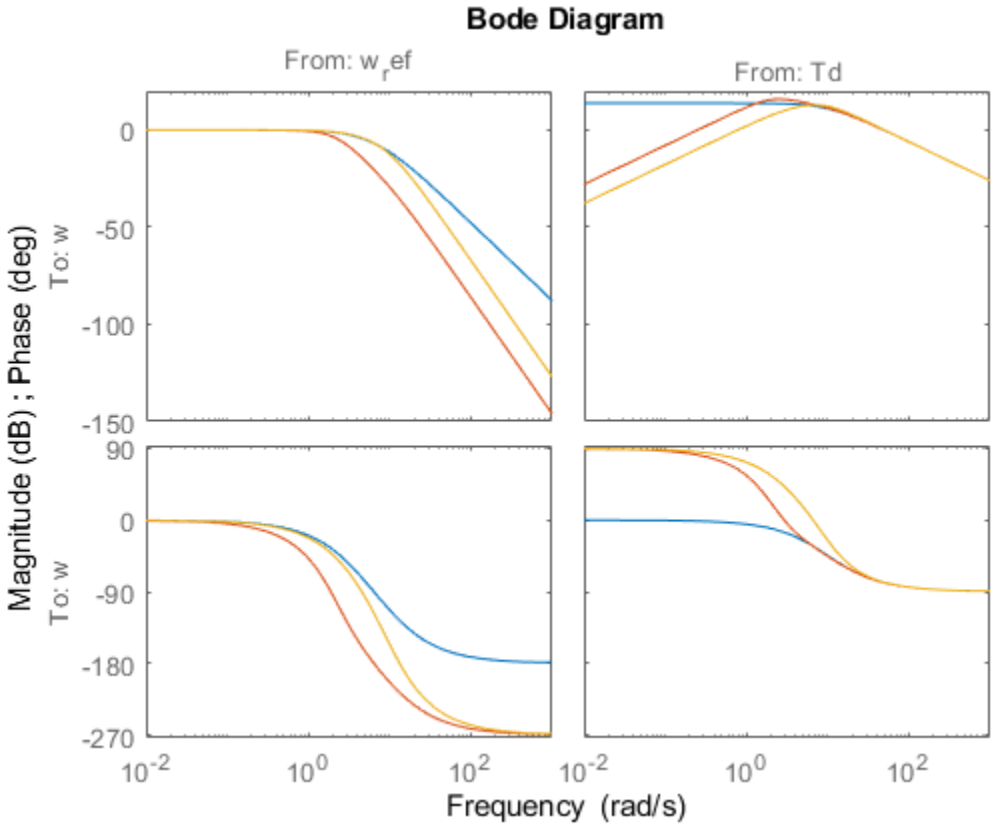
Next derive the closed-loop model for simulation purposes:

```
P = augstate(dcm);                    % inputs:Va,Td  outputs:w,x
C = K_lqr * append(tf(1,[1 0]),1,1);  % compensator including 1/s
```

```
OL = P * append(C,1);                   % open loop

CL = feedback(OL,eye(3),1:3,1:3);       % close feedback loops
cl_lqr = CL(1,[1 4]);                   % extract transfer (w_ref,Td)->w
```

This plot compares the closed-loop Bode diagrams for the three DC motor control designs

```
bodeplot(cl_ff,cl_rloc,cl_lqr);
```

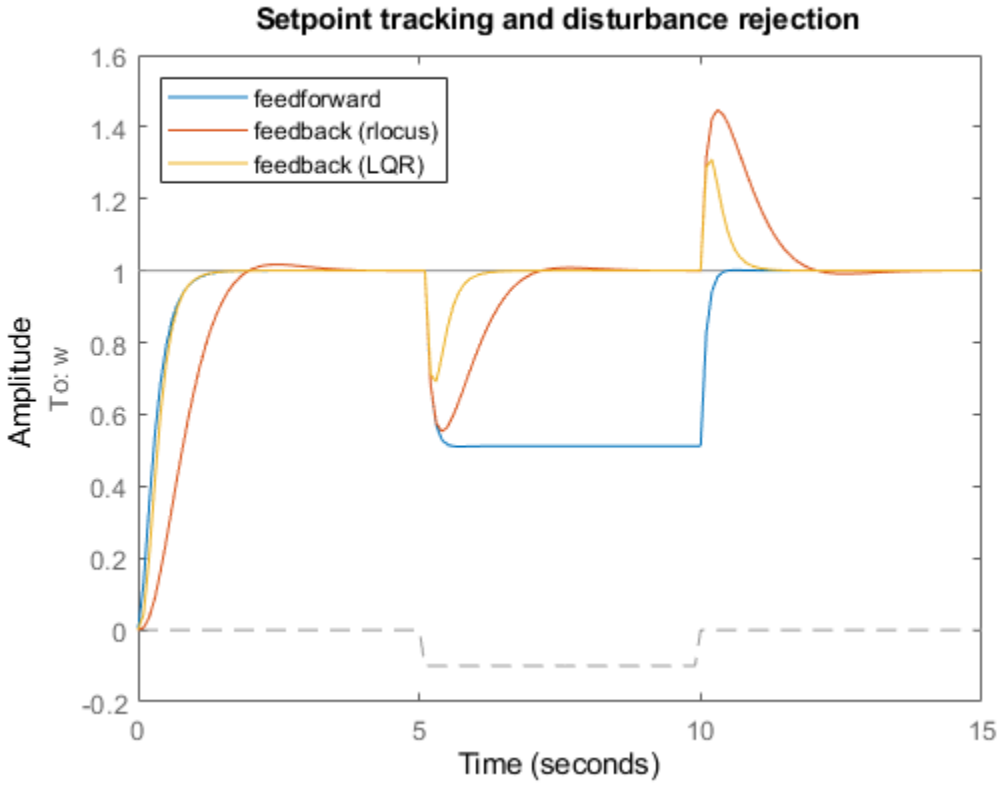


Click on the curves to identify the systems or inspect the data.

Comparison of DC Motor Control Designs

Finally we compare the three DC motor control designs on our simulation test case:

```
h = lsimplot(cl_ff,cl_rloc,cl_lqr,u,t);
title('Setpoint tracking and disturbance rejection')
legend('feedforward','feedback (rlocus)','feedback (LQR)','Location','NorthWest')
```



Thanks to its additional degrees of freedom, the LQR compensator performs best at rejecting load disturbances (among the three DC motor control designs discussed here).

Feedback Amplifier Design

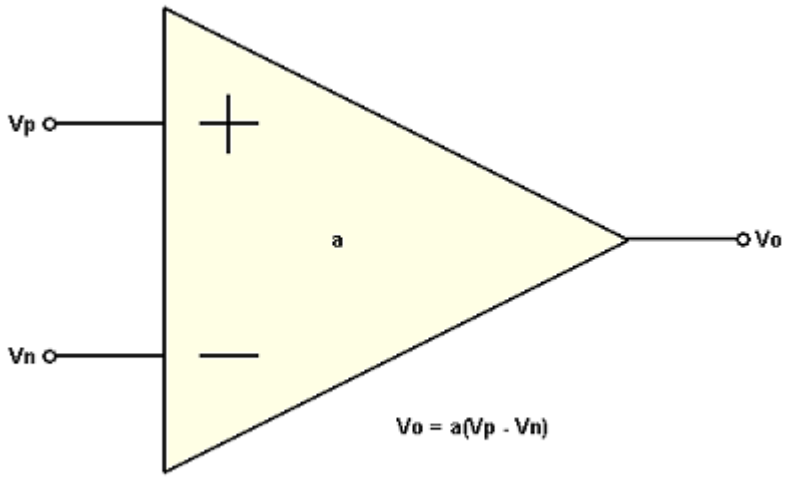
This example shows the design of a non-inverting feedback amplifier circuit using Control System Toolbox™. This design is built around the operational amplifier (op amp), a standard building block of electrical feedback circuits.

This tutorial shows how a real electrical system can be designed, modeled, and analyzed using the tools provided by Control System Toolbox.

Op Amp Description

The standard building block of electrical feedback circuits is the operational amplifier (op amp), a differential voltage amplifier designed to have extremely high dc gain, often in the range of 1e5 to 1e7.

The electrical symbol for the op amp is shown below.



This example assumes the use of an uncompensated op amp with 2 poles (at frequencies w1,w2) and high dc gain (a0). Assuming this op amp is operated in its linear mode (not saturated), then its open-loop transfer function can be represented as a linear time-invariant (LTI) system, as shown above.

Though higher-order poles will exist in a physical op amp, it has been assumed in this case that these poles lie in a frequency range where the magnitude has dropped well below unity.

Open-Loop Transfer Function:

$$a(s) = \frac{a_0}{(1 + s/\omega_1)(1 + s/\omega_2)}$$

The following system parameters are assumed:

$$a_0 = 1e5$$

$$\omega_1 = 1e4$$

$$\omega_2 = 1e6$$

```
a0 = 1e5;
w1 = 1e4;
w2 = 1e6;
```

Next, you want to create a transfer function model of this system using Control System Toolbox. This model will be stored in the MATLAB® workspace as an LTI object.

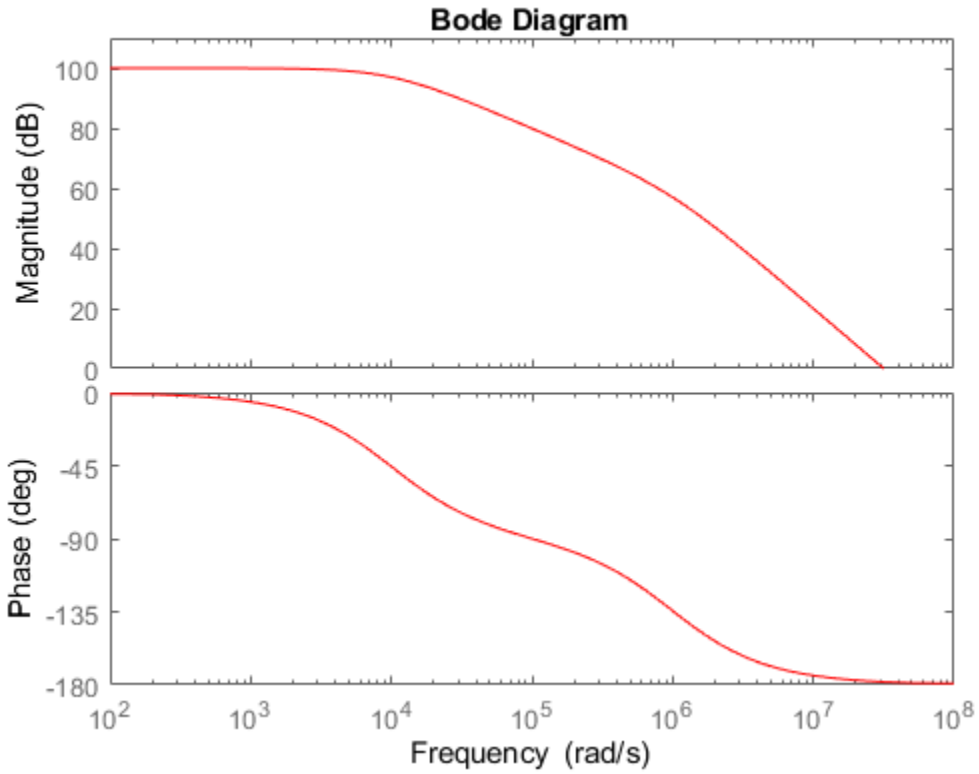
First, define the Laplace variable, s, using the TF command. Then use 's' to construct the open-loop transfer function, a(s):

```
s = tf('s');
a = a0/(1+s/w1)/(1+s/w2)
```

```
a =

           1e15
  ----------------------
  s^2 + 1.01e06 s + 1e10

Continuous-time transfer function.
```

You can view the frequency response of a(s) using the BODEPLOT command:

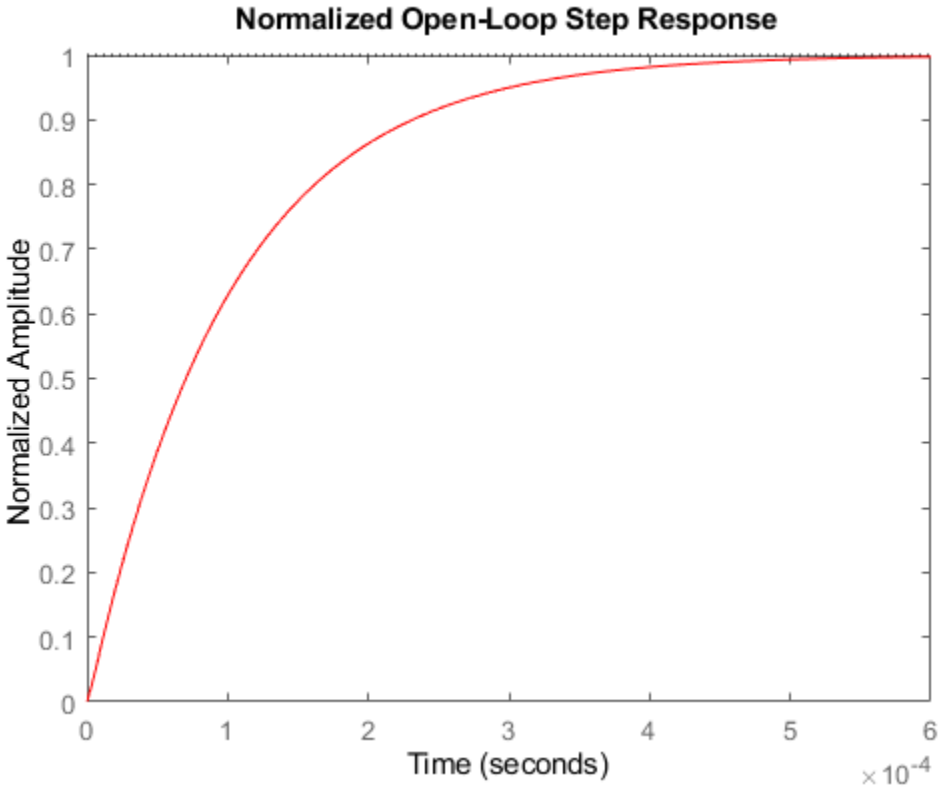
```
h = bodeplot(a,'r');

setoptions(h,'FreqUnits','rad/s','MagUnits','dB','PhaseUnits','deg',...
    'YLimMode','Manual','YLim',{[0,110],[-180,0]});
```



Right-click on the plot to access a menu of properties for this Bode Diagram. Left-click on the curves to create moveable data markers which can be used to obtain response details.

You can view the normalized step response of a(s) using the STEPPLOT and DCGAIN commands:

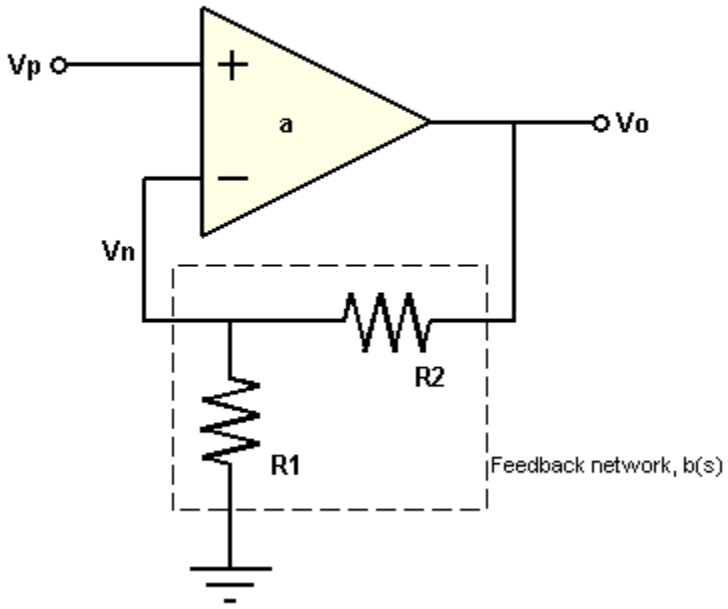
```
a_norm = a / dcgain(a);
stepplot(a_norm,'r')

title('Normalized Open-Loop Step Response');
ylabel('Normalized Amplitude');
```



Right-click on the plot and select "Characteristics -> Settling Time" to display the settling time. Hold the mouse over the settling time marker to reveal the exact value of the settling time.

Feedback Amplifier

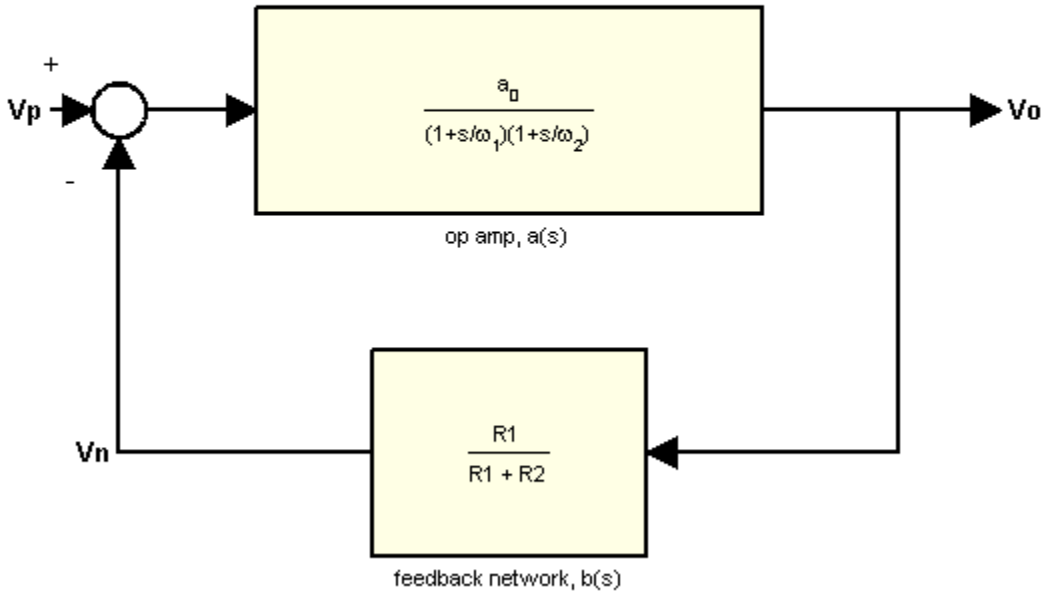
Now add a resistive feedback network and wire the system as a non-inverting amplifier.



This feedback network, b(s), is simply a voltage divider with input Vo and output Vn. Solving for the ratio Vn/Vo yields the transfer function for b(s):

```
b  =  Vn / Vo  =  R1 / (R1 + R2)
```

The block diagram representation of the system is shown below.



Solving for the ratio Vo/Vp yields the closed-loop gain, A(s):

```
A  =  Vo / Vp  =  a / (1 + ab)
```

If the product 'ab' is sufficiently large (>>1), then A(s) may be approximated as

```
A  =  1 / b
```

Now assume that you need to design an amplifier of dc gain (Vo/Vp) 10 and that R1 is fixed at 10 kOhm. Solving for R2 yields:

```
A0 = 10;
b = 1 / A0;    % approximation for ab>>1
R1 = 10000;
R2 = R1 * (1/b - 1)
```

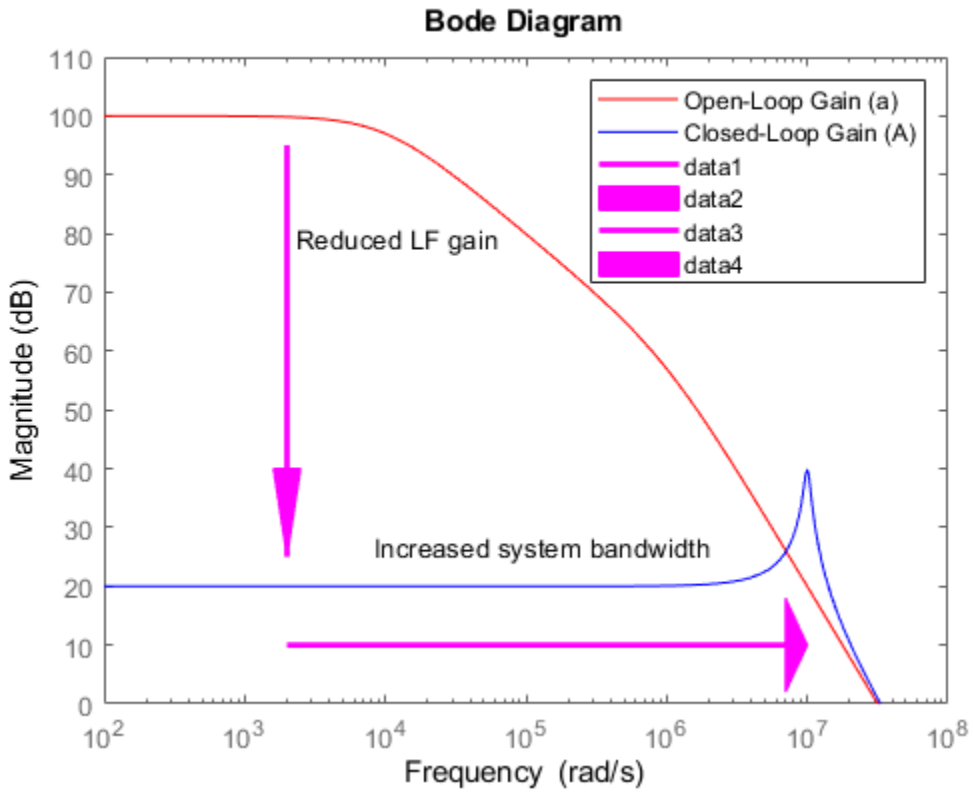
```
R2 =

       90000
```

Construct the closed-loop system using the FEEDBACK command:

```
A = feedback(a,b);
```

Next, plot the frequency responses of a(s) and A(s) together using the BODEMAG command:

```
bodemag(a,'r',A,'b');

legend('Open-Loop Gain (a)','Closed-Loop Gain (A)')
ylim([0,110]);

% Annotations
opampdemo_annotate(1)
```



The use of negative feedback to reduce the low-frequency (LF) gain has led to a corresponding increase in the system bandwidth (defined as the frequency where the gain drops 3dB below its maximum value).

This gain / bandwidth tradeoff is a powerful tool in the design of feedback amplifier circuits.

Since the gain is now dominated by the feedback network, a useful relationship to consider is the sensitivity of this gain to variation in the op amp's natural (open-loop) gain.

Before deriving the system sensitivity, however, it is useful to define the loop gain, L(s)=a(s)b(s), which is the total gain a signal experiences traveling around the loop:

```
L = a * b;
```

You will use this quantity to evaluate the system sensitivity and stability margins.

The system sensitivity, S(s), represents the sensitivity of A(s) to variation in a(s).

$$S = \frac{\delta A/A}{\delta a/a} = \frac{1}{1 + a(s)b(s)} = \frac{1}{1 + L(s)}$$

The inverse relationship between S(s) and L(s) reveals another benefit of negative feedback: "gain desensitivity".

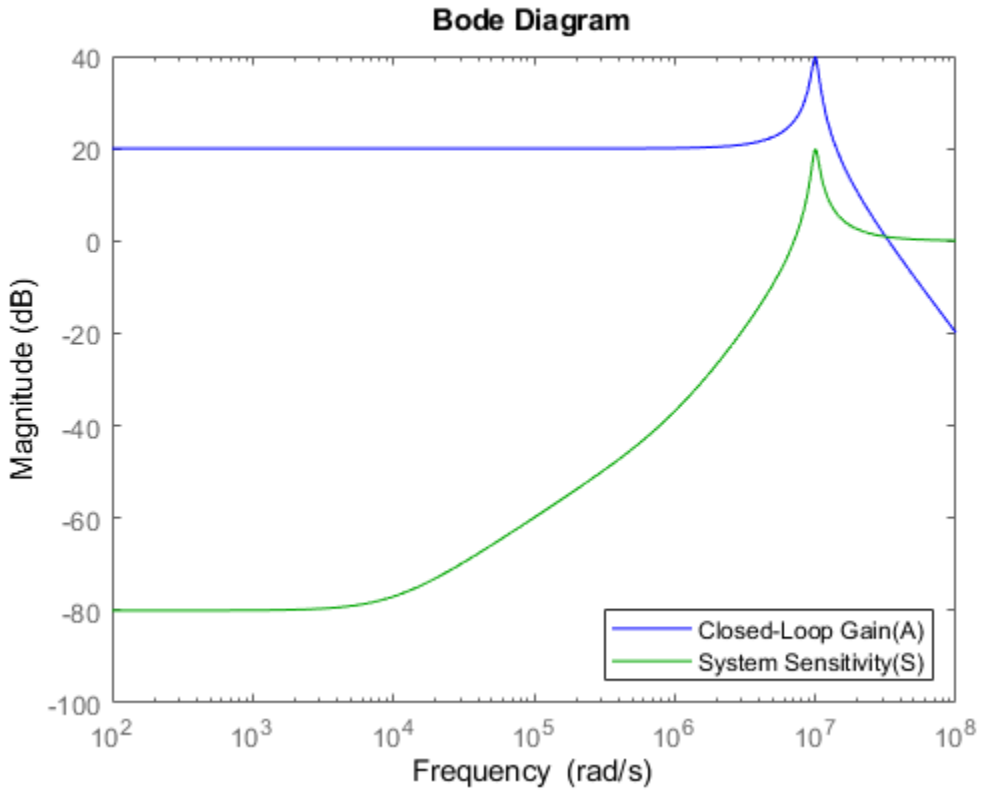
```
S = 1 / (1 + L);
```

S(s) has the same form as the feedback equation and, therefore, may be constructed using the more-robust FEEDBACK command:

```
S = feedback(1,L);
```

The magnitudes of S(s) and A(s) may be plotted together using the BODEMAG command:

```
bodemag(A,'b',S,'g')
legend('Closed-Loop Gain(A)', 'System Sensitivity(S)','Location','SouthEast')
```

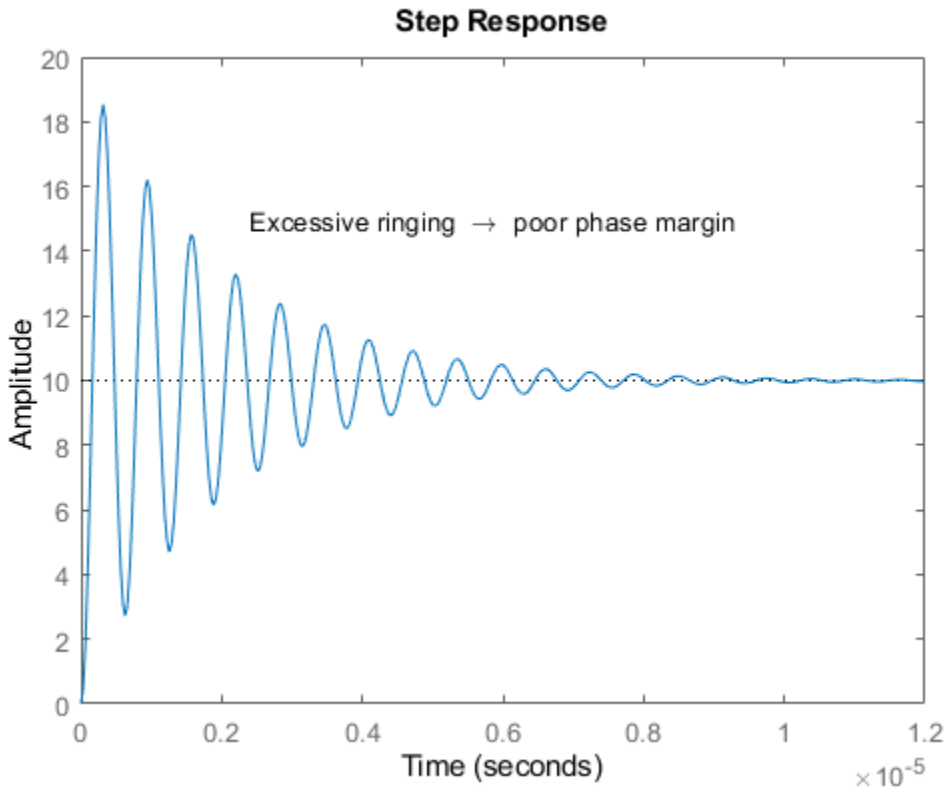


The very small low-frequency sensitivity (about -80 dB) indicates a design whose closed-loop gain suffers minimally from open-loop gain variation. Such variation in a(s) is common due to manufacturing variability, temperature change, etc.

You can check the step response of A(s) using the STEPPLOT command:

```
stepplot(A)
```

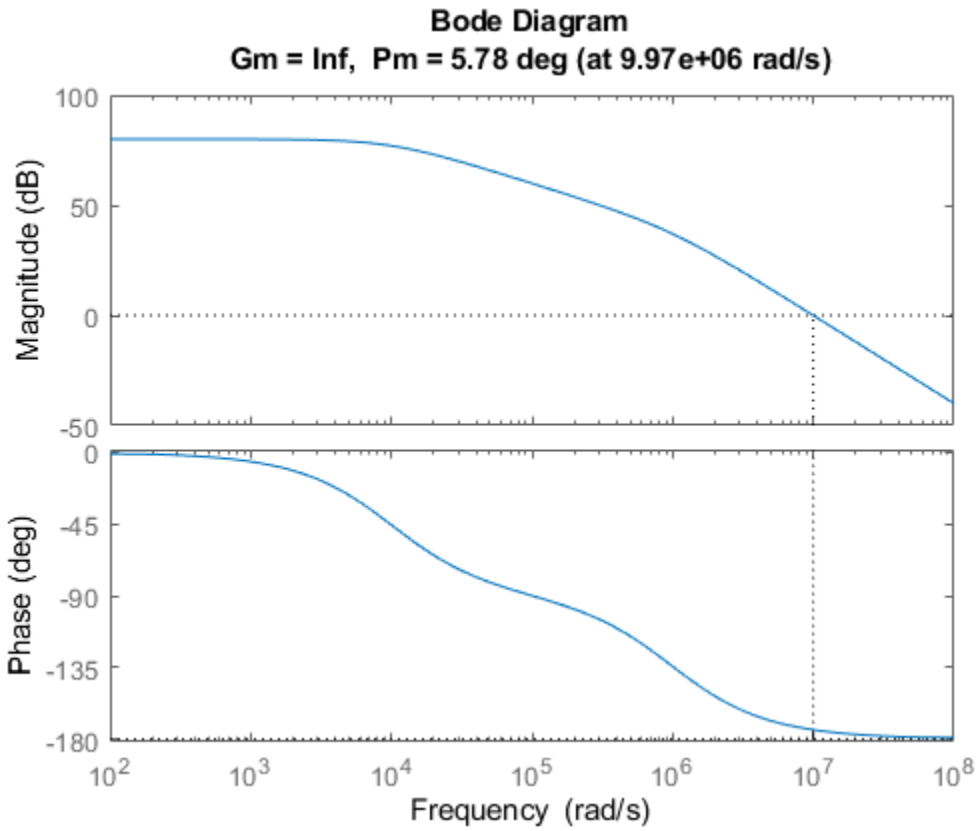
```
% Annotation
opampdemo_annotate(2)
```



Note that the use of feedback has greatly reduced the settling time (by about 98%). However, the step response now displays a large amount of ringing, indicating poor stability margin.

You can analyze the stability margin by plotting the loop gain, L(s), with the MARGIN command:

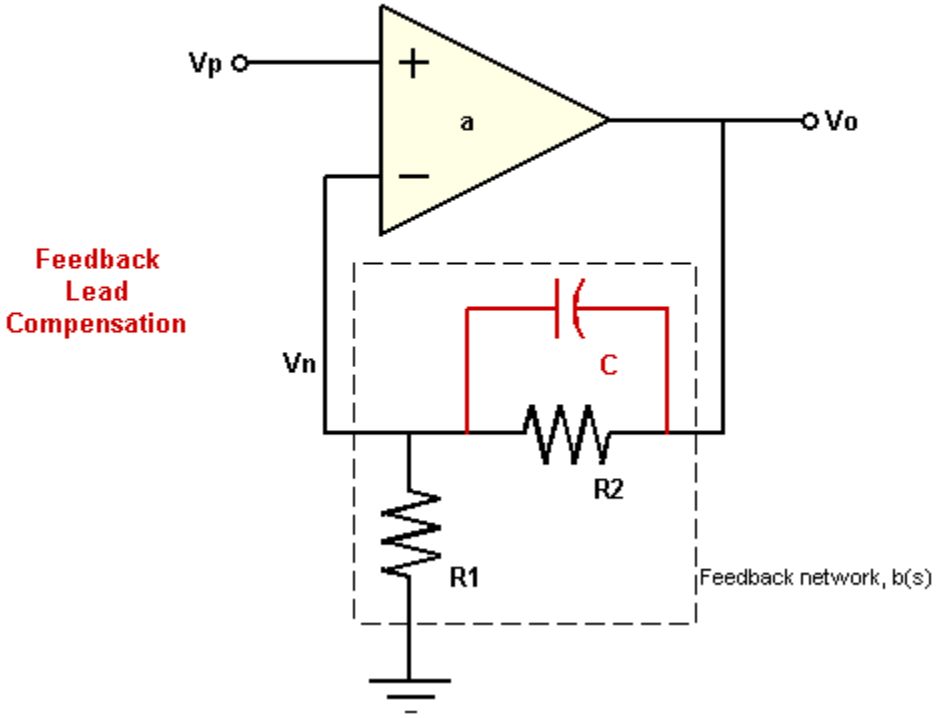
```
margin(L)
```



The resulting plot indicates a phase margin of less than 6 degrees. You will need to compensate this amplifier in order to raise the phase margin to an acceptable level (generally 45 deg or more), thus reducing excessive overshoot and ringing.

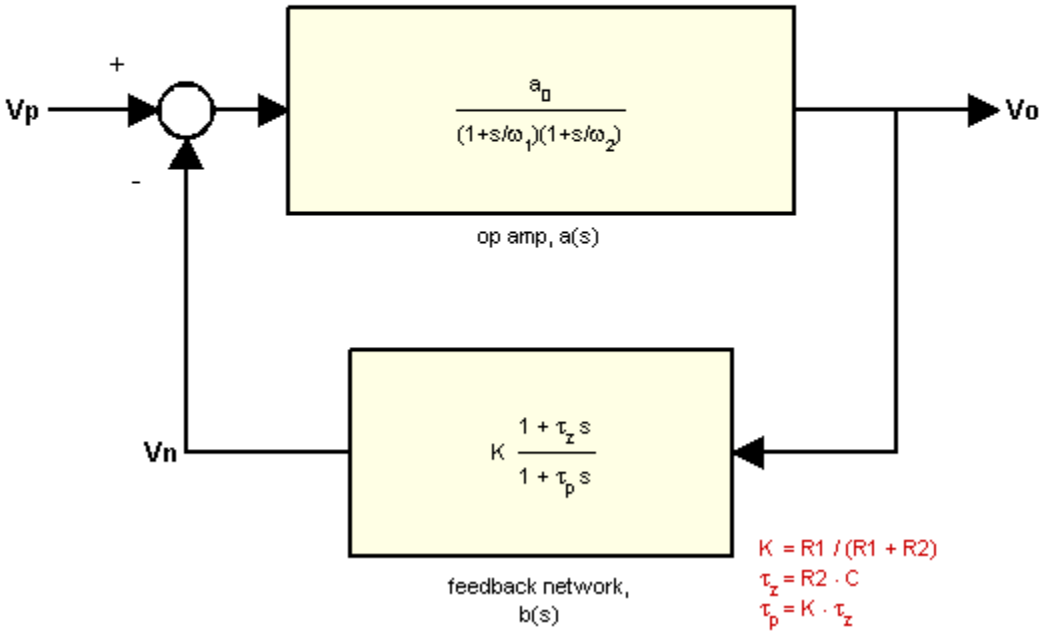
Feedback Lead Compensation

A commonly used method of compensation in this type of circuit is "feedback lead compensation". This technique modifies b(s) by adding a capacitor, C, in parallel with the feedback resistor, R2.



The capacitor value is chosen so as to introduce a phase lead to b(s) near the crossover frequency, thus increasing the amplifier's phase margin.

The new feedback transfer function is shown below.



You can approximate a value for C by placing the zero of b(s) at the 0dB crossover frequency of L(s):

```
[Gm,Pm,Wcg,Wcp] = margin(L);
C = 1/(R2*Wcp)
```

```
C =

   1.1139e-12
```

To study the effect of C on the amplifier response, create an LTI model array of b(s) for several values of C around your initial guess:

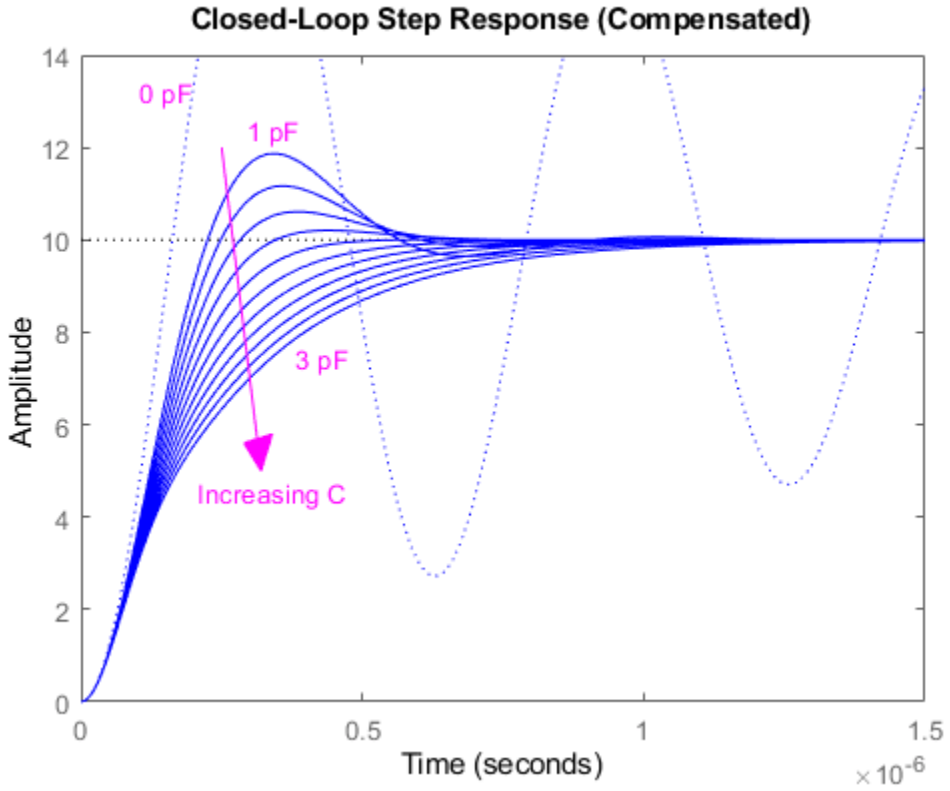

```
K = R1/(R1+R2);
C = [1:.2:3]*1e-12;
for n = 1:length(C)
    b_array(:,:,n) = tf([K*R2*C(n) K],[K*R2*C(n) 1]);
end
```

Now you can create LTI arrays for A(s) and L(s):

```
A_array = feedback(a,b_array);
L_array = a*b_array;
```

You can plot the step response of all models in the LTI array, A_array(s), together with A(s) using the STEPPLOT command:

```
stepplot(A,'b:',A_array,'b',[0:.005:1]*1.5e-6);
title('Closed-Loop Step Response (Compensated)');

% Plot Annotations
opampdemo_annotate(3)
```

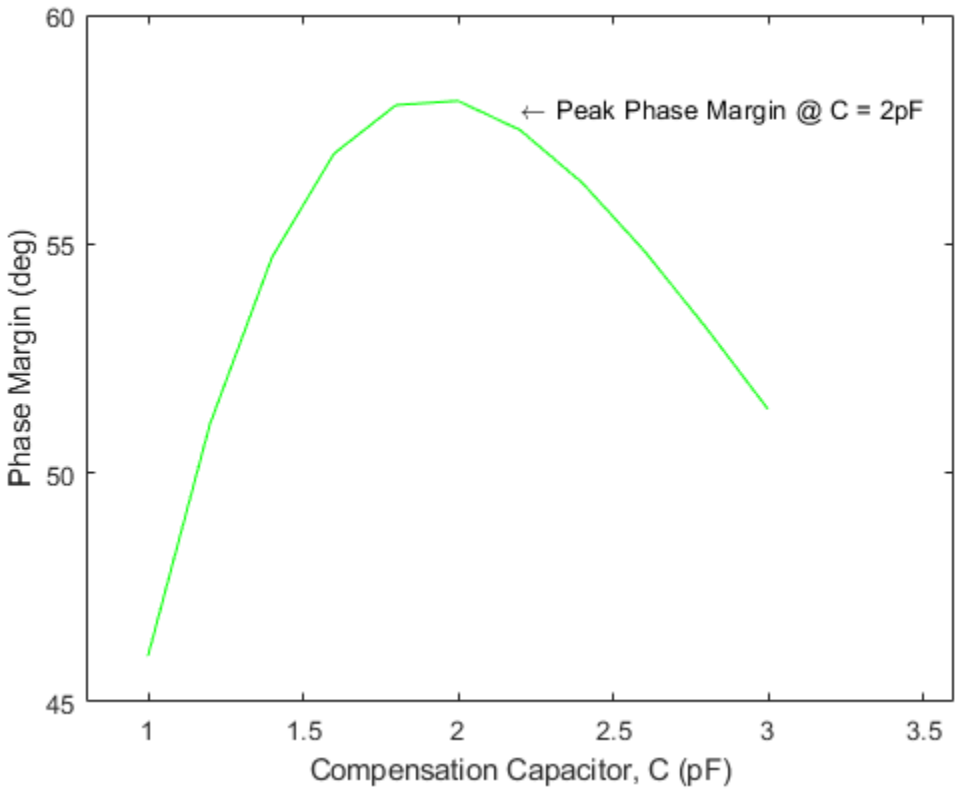


The phase margins for our loop gain array, L_array(s), are found using the MARGIN command:

```
[Gm,Pm,Wcg,Wcp] = margin(L_array);
```

The phase margins may now be plotted as a function of C.

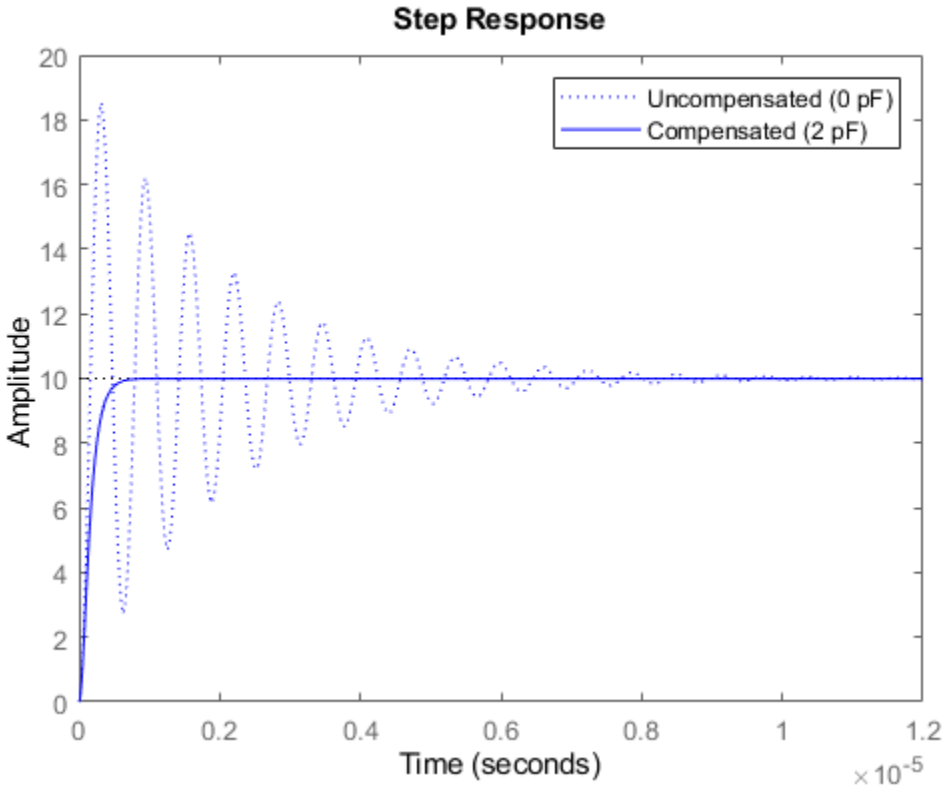
```
plot(C*1e12,Pm,'g');
ax = gca;
xlim([0.8 3.6]);
ylim([45 60]);
ax.Box = 'on';
xlabel('Compensation Capacitor, C (pF)');
ylabel('Phase Margin (deg)')

% Plot Annotations
opampdemo_annotate(4)
```



A maximum phase margin of 58 deg is obtained when C=2pF (2e-12).

The model corresponding to C=2pF is the sixth model in the LTI array, b_array(s). You can plot the step response of the closed- loop system for this model by selecting index 6 of the LTI array A_array(s):

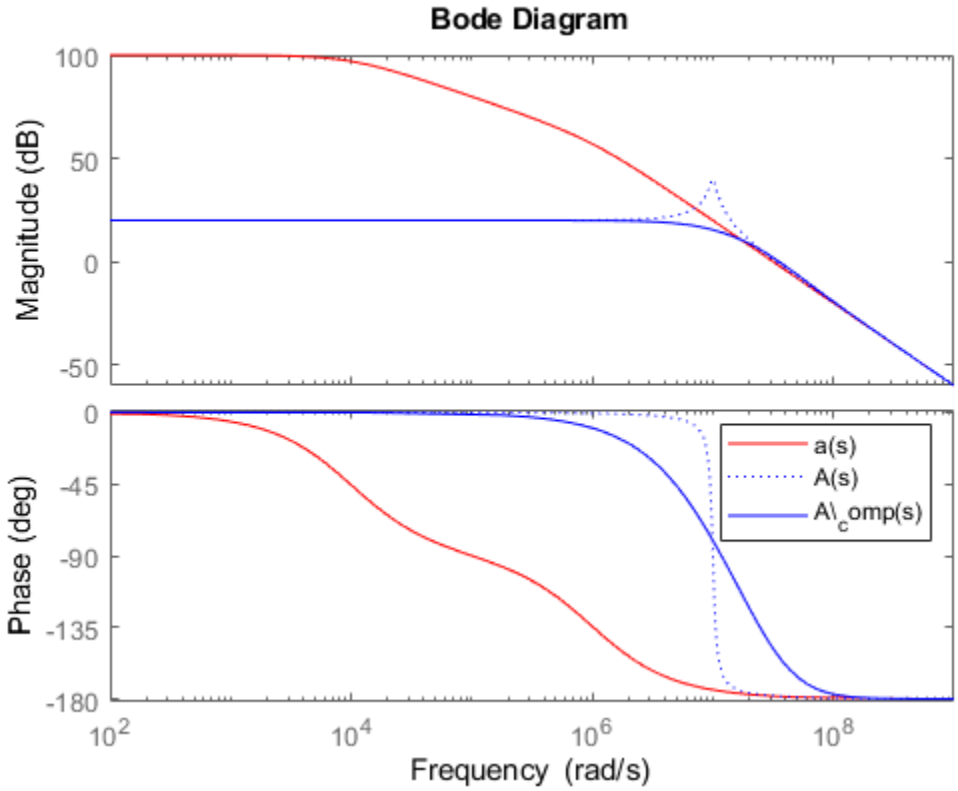
```
A_comp = A_array(:,:,6);
stepplot(A,'b:',A_comp,'b')
legend('Uncompensated (0 pF)','Compensated (2 pF)')
```



Note that the settling time has been further reduced (by an additional 85%).

We can overlay the frequency-response of all three models (open-loop, closed-loop, compensated closed-loop) using the BODE command:

```
bodeplot(a,'r',A,'b:',A_comp,'b')
legend('a(s)','A(s)','A\_comp(s)');
```



Note how the addition of the compensation capacitor has eliminated peaking in the closed-loop gain and also greatly extended the phase margin.

Summary

A brief summary of the choice of component values in the design of this non-inverting feedback amplifier circuit:

- Final component values: R1 = 10 kOhm, R2 = 90 kOhm, C = 2 pF.
- A resistive feedback network (R1,R2) was selected to yield a broadband amplifier gain of 10 (20 dB).
- Feedback lead compensation was used to tune the loop gain near the crossover frequency. The value for the compensation capacitor, C, was optimized to provide a maximum phase margin of about 58 degrees.

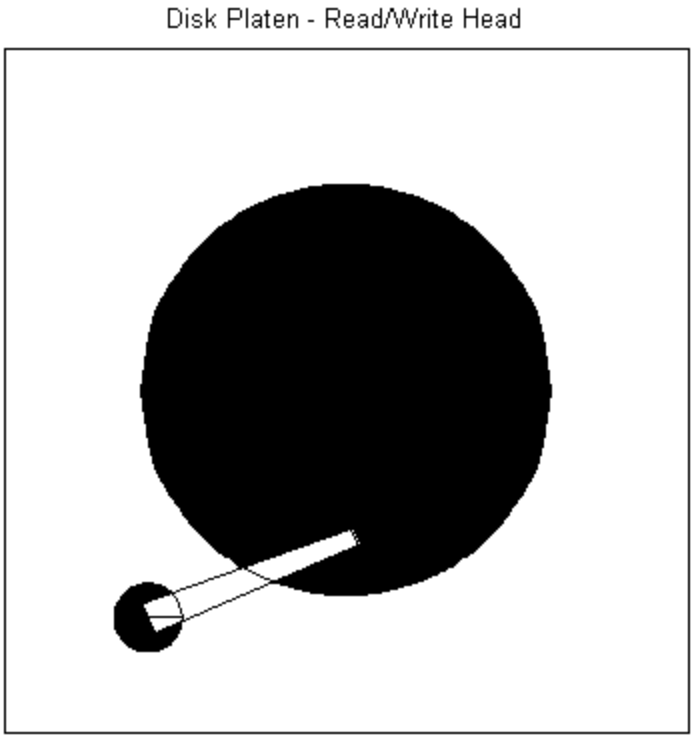
Digital Servo Control of a Hard-Disk Drive

This example shows how to use Control System Toolbox™ to design a digital servo controller for a disk drive read/write head.

For details about the system and model, see Chapter 14 of "Digital Control of Dynamic Systems," by Franklin, Powell, and Workman.

Disk Drive Model

Below is a picture of the system to be modeled.



The head-disk assembly (HDA) and actuators are modeled by a 10th-order transfer function including two rigid-body modes and the first four resonances.

The model input is the current ic driving the voice coil motor, and the output is the position error signal (PES, in % of track width). The model also includes a small delay.

Disk Drive Model:

$$G(s) = G_r(s)G_f(s)$$

$$G_r(s) = e^{-1e-5s}\frac{1e6}{s(s+12.5)}$$

$$G_f(s) = \sum_{i=1}^{4} \frac{\omega_i(a_i s + b_i \omega_i)}{s^2 + 2\zeta_i \omega_i s + \omega_i^2}$$

The coupling coefficients, damping, and natural frequencies (in Hz) for the dominant flexible modes are listed below.

Model Data:

$$(a_1, b_1, \zeta_1, \omega_1) = (.0000115, -.00575, .05, 70)$$

$$(a_2, b_2, \zeta_2, \omega_2) = (0, .0230, .005, 2200)$$

$$(a_3, b_3, \zeta_3, \omega_3) = (0, .8185, .05, 4000)$$

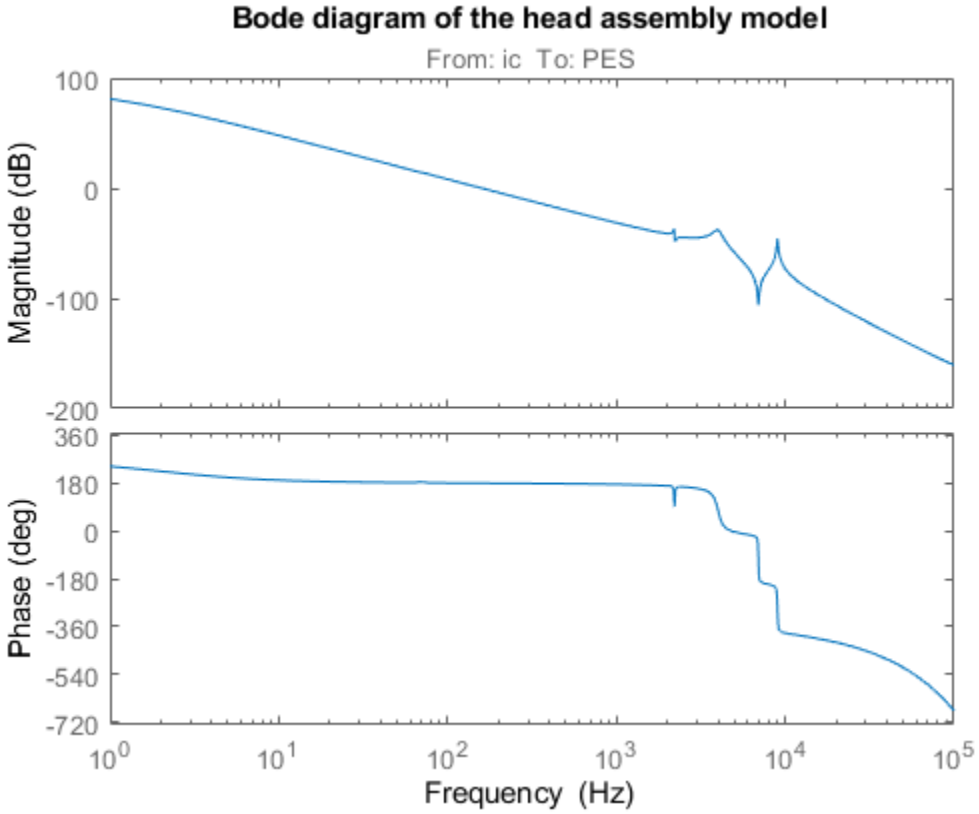
$$(a_4, b_4, \zeta_4, \omega_4) = (.0273, .1642, .005, 9000)$$

Given this data, construct a nominal model of the head assembly:

```
load diskdemo
Gr = tf(1e6,[1 12.5 0],'outputdelay',1e-5);
Gf1 = tf(w1*[a1 b1*w1],[1 2*z1*w1 w1^2]); % first  resonance
Gf2 = tf(w2*[a2 b2*w2],[1 2*z2*w2 w2^2]); % second resonance
Gf3 = tf(w3*[a3 b3*w3],[1 2*z3*w3 w3^2]); % third  resonance
Gf4 = tf(w4*[a4 b4*w4],[1 2*z4*w4 w4^2]); % fourth resonance
G = Gr * (ss(Gf1) + Gf2 + Gf3 + Gf4);     % convert to state space for accuracy
```

Plot the Bode response of the head assembly model:

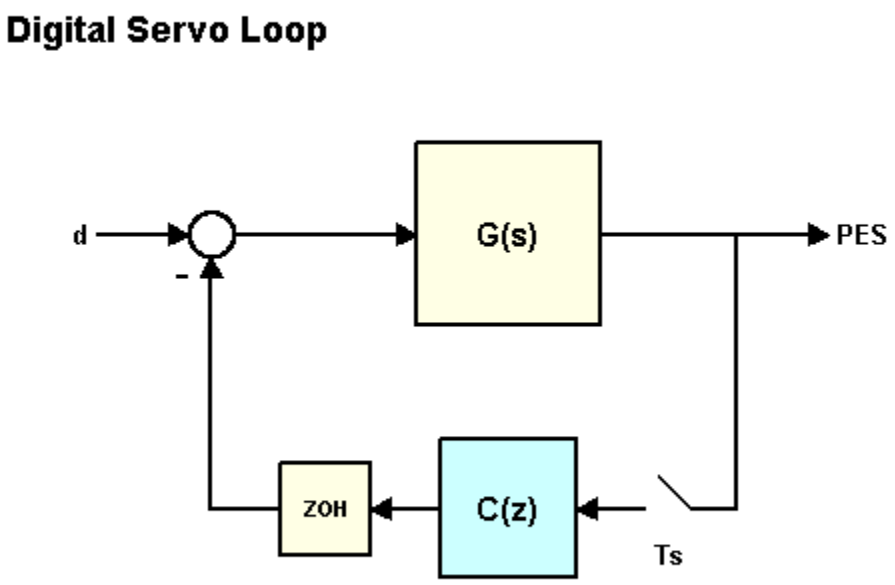
```
cla reset
G.InputName = 'ic';
G.OutputName = 'PES';
h = bodeplot(G);
title('Bode diagram of the head assembly model');
setoptions(h,'Frequnits','Hz','XLimMode','manual','XLim', {[1 1e5]});
```



Servo Controller

Servo control is used to keep the read/write head "on track." The servo controller C(z) is digital and designed to maintain the PES (offset from the track center) close to zero.

The disturbance considered here is a step variation d in the input current ic. Your task is to design a digital compensator C(z) with adequate disturbance rejection performance.



The sample time for the digital servo is Ts = 7e-5 sec (14.2 kHz).

Realistic design specs are listed below.

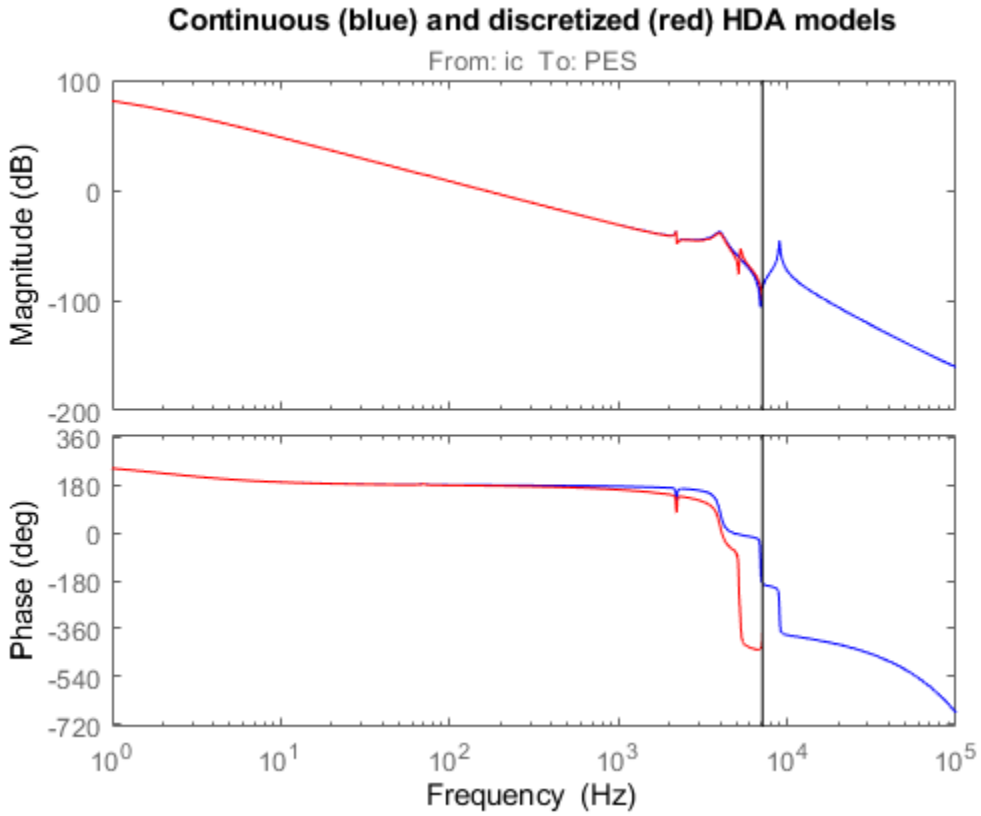
Design Specs:

- Open-loop gain > 20dB at 100 Hz
- Bandwidth > 800 Hz
- Gain margin > 10 dB
- Phase margin > 45 deg
- Peak closed-loop gain < 4 dB

Discretization of Model

Since the servo controller is digital, you can perform the design in the discrete domain. To this effect, discretize the HDA model using C2D and the zero-order hold (ZOH) method:

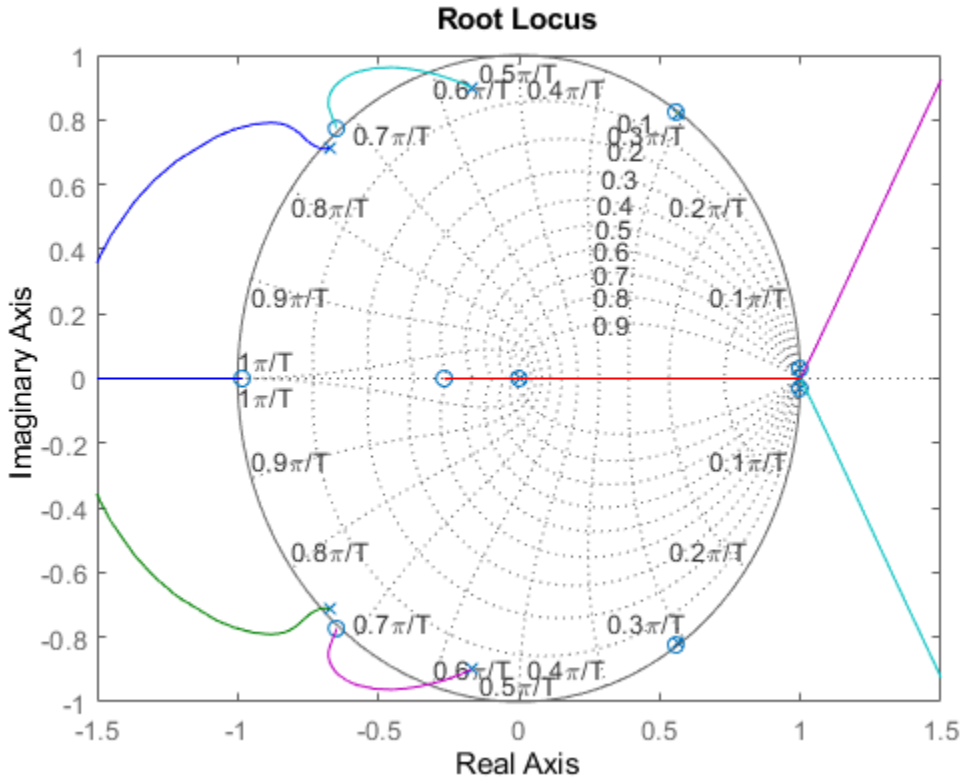
```
cla reset
Ts = 7e-5;
Gd = c2d(G,Ts);
h = bodeplot(G,'b',Gd,'r'); % compare with the continuous-time model
title('Continuous (blue) and discretized (red) HDA models');
setoptions(h,'Frequnits','Hz','XLimMode','manual','XLim', {[1 1e5]});
```



Controller Design

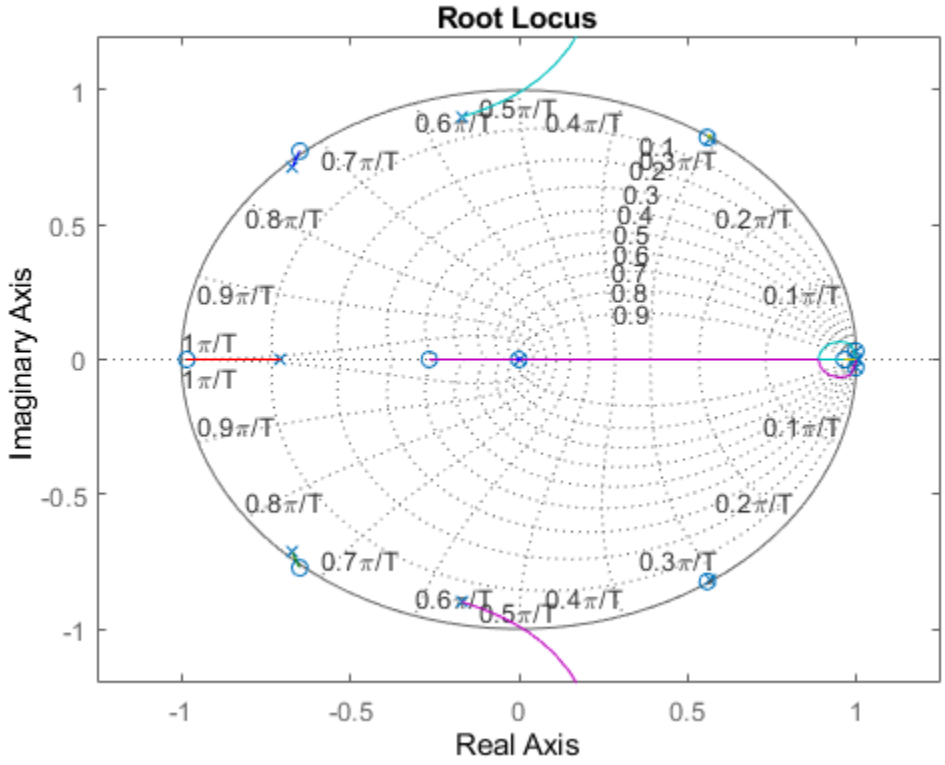
Now to the compensator design. Start with a pure integrator 1/(z-1) to ensure zero steady-state error, plot the root locus of the open-loop model Gd*C, and zoom around z=1 using the Zoom In option under the Tools menu.

```
C = tf(1,[1 -1],Ts);
h = rlocusplot(Gd*C);
setoptions(h,'Grid','on','XLimMode','Manual','XLim',{[-1.5,1.5]},...
    'YLimMode','Manual','YLim',{[-1,1]});
```



Because of the two poles at z=1, the servo loop is unstable for all positive gains. To stabilize the feedback loop, first add a pair of zeros near z=1.

```
C = C * zpk([.963,.963],-0.706,1,Ts);
h = rlocusplot(Gd*C);
setoptions(h,'Grid','on','XLimMode','Manual','XLim',{[-1.25,1.25]},...
    'YLimMode','Manual','YLim',{[-1.2,1.2]});
```

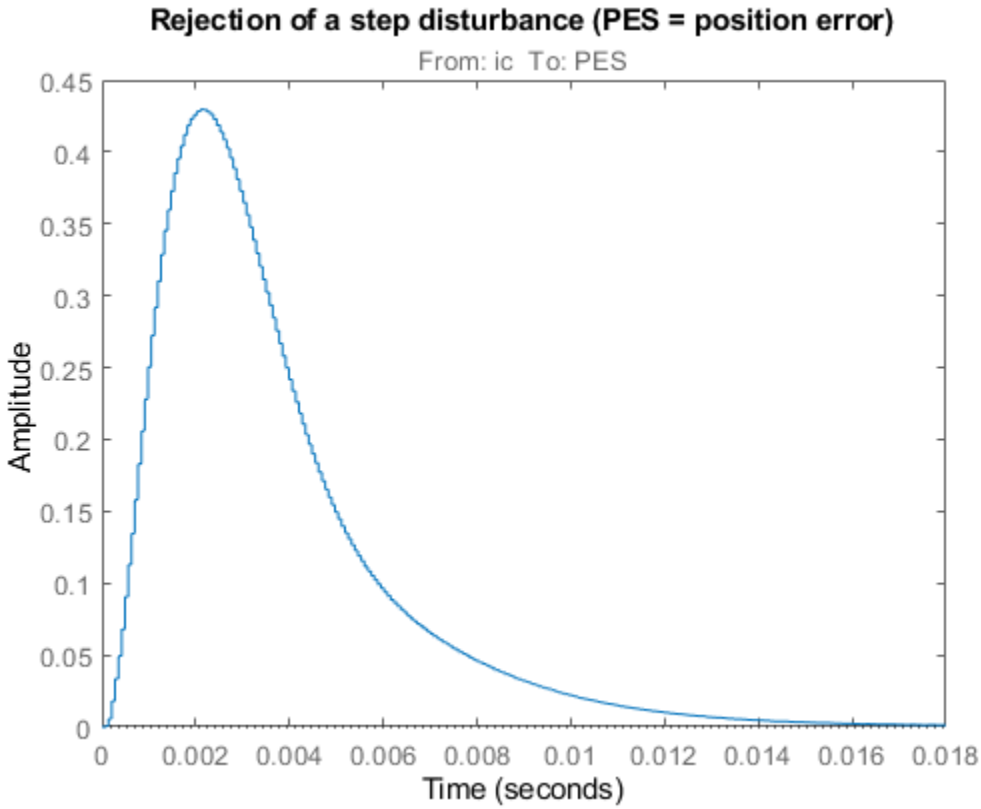


Next adjust the loop gain by clicking on the locus and dragging the black square inside the unit circle. The loop gain is displayed in the data marker. A gain of approximately 50 stabilizes the loop (set C1 = 50*C).

```
C1 = 50 * C;
```

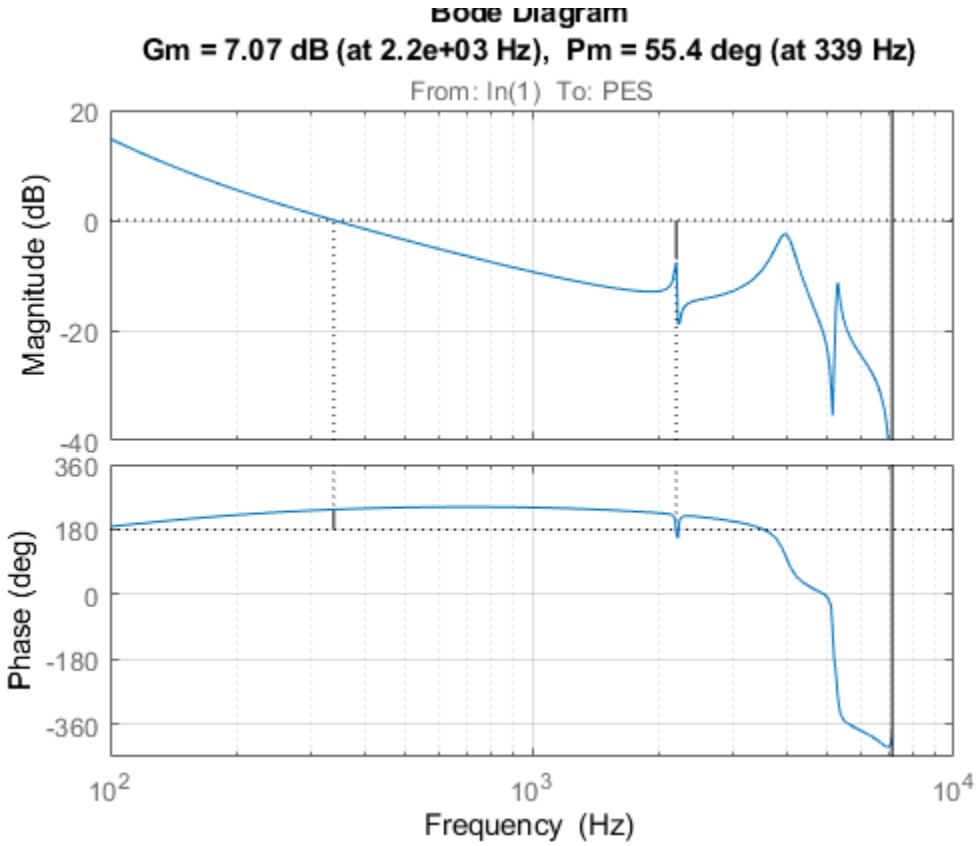
Now simulate the closed-loop response to a step disturbance in current. The disturbance is smoothly rejected, but the PES is too large (head deviates from track center by 45% of track width).

```
cl_step = feedback(Gd,C1);
h = stepplot(cl_step);
title('Rejection of a step disturbance (PES = position error)')
setoptions(h,'Xlimmode','auto','Ylimmode','auto','Grid','off');
```



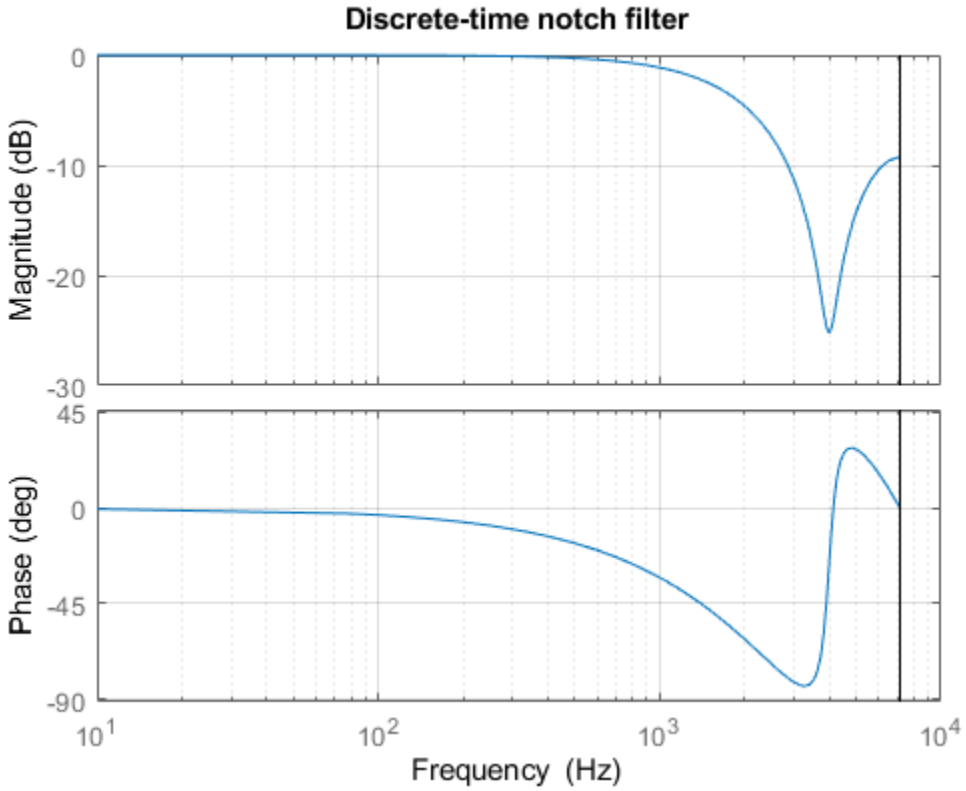
Next look at the open-loop Bode response and the stability margins. The gain at 100 Hz is only 15 dB (vs. spec of 20 dB) and the gain margin is only 7dB, so increasing the loop gain is not an option.

```
margin(Gd*C1)

diskdemo_aux1(1);
```



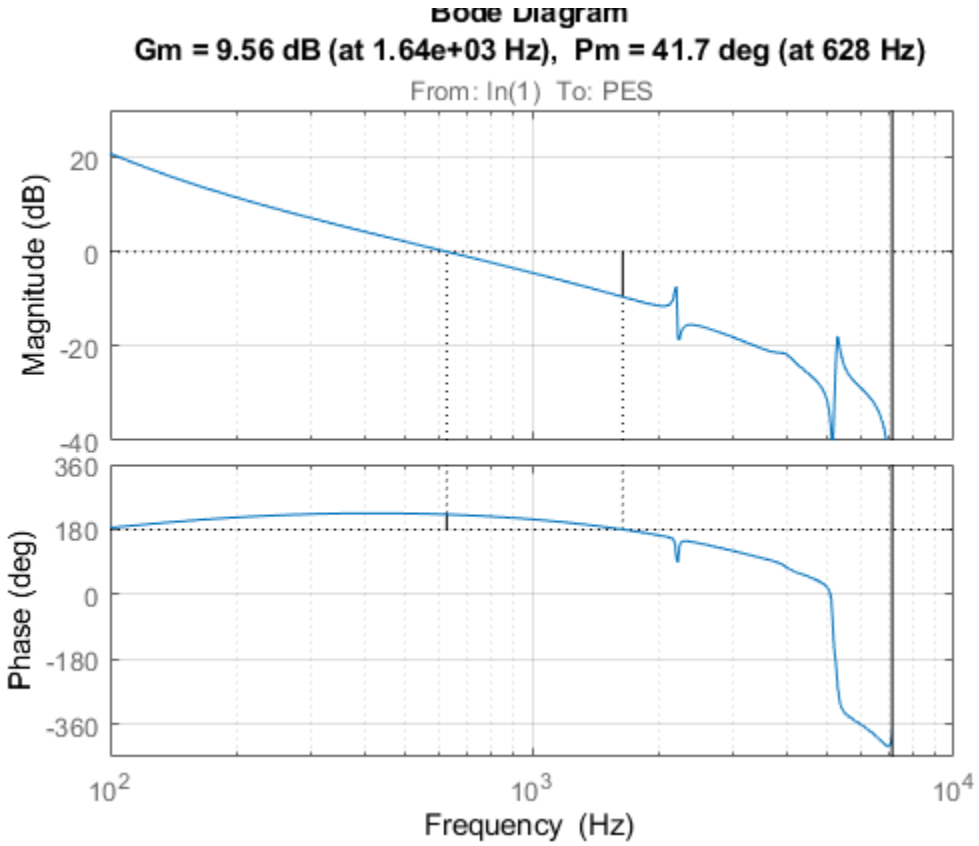
To make room for higher low-frequency gain, add a notch filter near the 4000 Hz resonance.

```
w0 = 4e3 * 2*pi;                                % notch frequency in rad/sec
notch = tf([1 2*0.06*w0 w0^2],[1 2*w0 w0^2]);   % continuous-time notch
notchd = c2d(notch,Ts,'matched');               % discrete-time notch
C2 = C1 * notchd;

h = bodeplot(notchd);
title('Discrete-time notch filter');
setoptions(h,'FreqUnits','Hz','Grid','on');
```



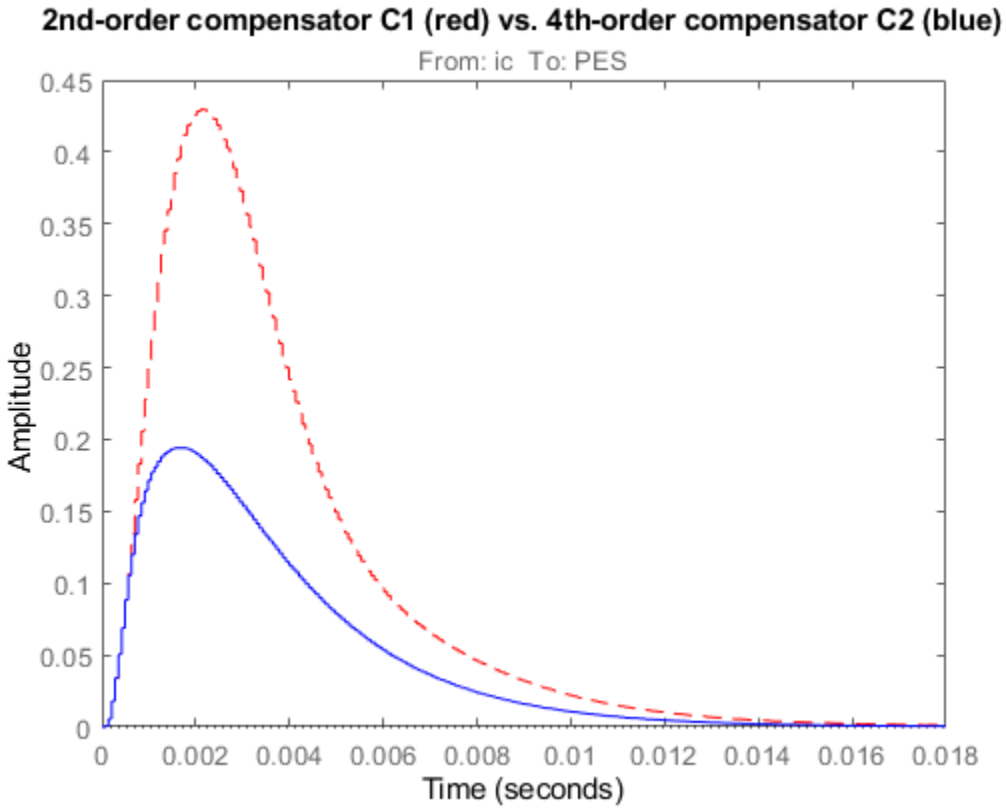
You can now safely double the loop gain. The resulting stability margins and gain at 100 Hz are within specs.

```
C2 = 2 * C2;
margin(Gd * C2)

diskdemo_aux1(2);
```



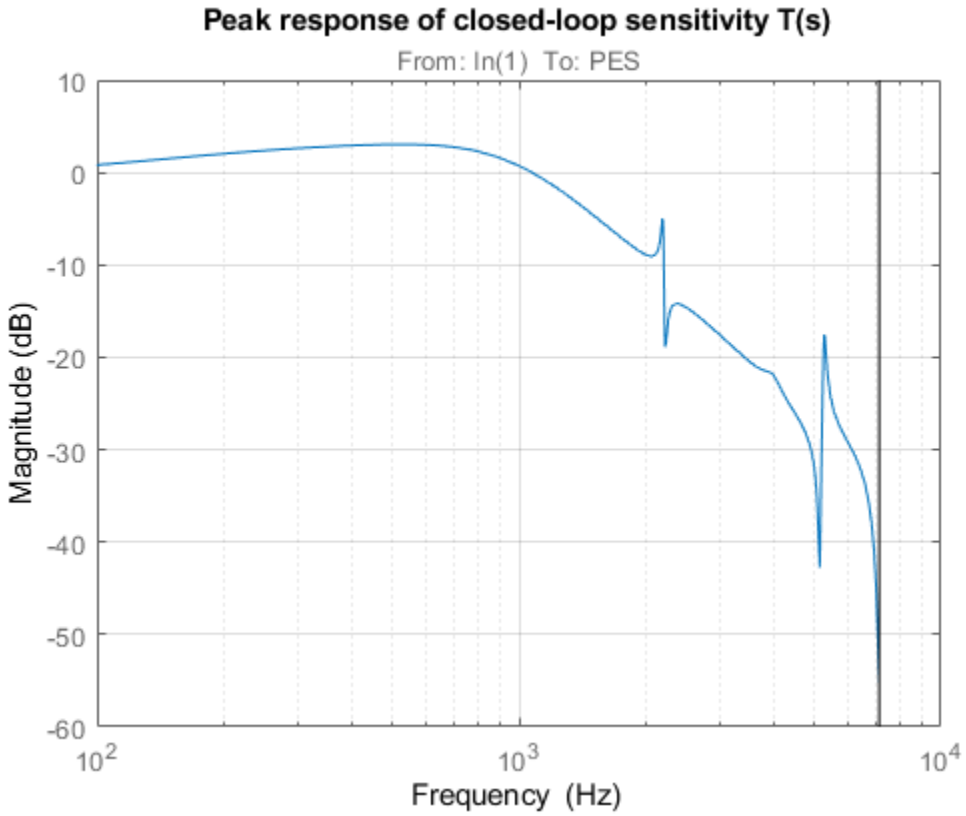
Step disturbance rejection has also greatly improved. The PES now stays below 20% of the track width.

```
cl_step1 = feedback(Gd,C1);
cl_step2 = feedback(Gd,C2);
stepplot(cl_step1,'r--',cl_step2,'b')
title('2nd-order compensator C1 (red) vs. 4th-order compensator C2 (blue)')
```



Check if the 3dB peak gain spec on T = Gd*C/(1+Gd*C) (closed-loop sensitivity) is met:

```
Gd = c2d(G,Ts);
Ts = 7e-5;

T = feedback(Gd*C2,1);
h = bodeplot(T);
title('Peak response of closed-loop sensitivity T(s)')

setoptions(h,'PhaseVisible','off','FreqUnits','Hz','Grid','on', ...
            'XLimMode','Manual','XLim',{[1e2 1e4]});
```



To see the peak value, right-click on the axis and choose the **Peak Response** option under the **Characteristics** menu, then hold the mouse over the blue marker, or just click on it.

Robustness Analysis

Finally let's analyze the robustness to variations in the damping and natural frequencies of the 2nd and 3rd flexible modes.

Parameter Variations:

$$\omega_2 = 2200 \pm 10\%$$

$$\omega_3 = 4000 \pm 20\%$$

$$\zeta_2 = 0.005 \pm 50\%$$

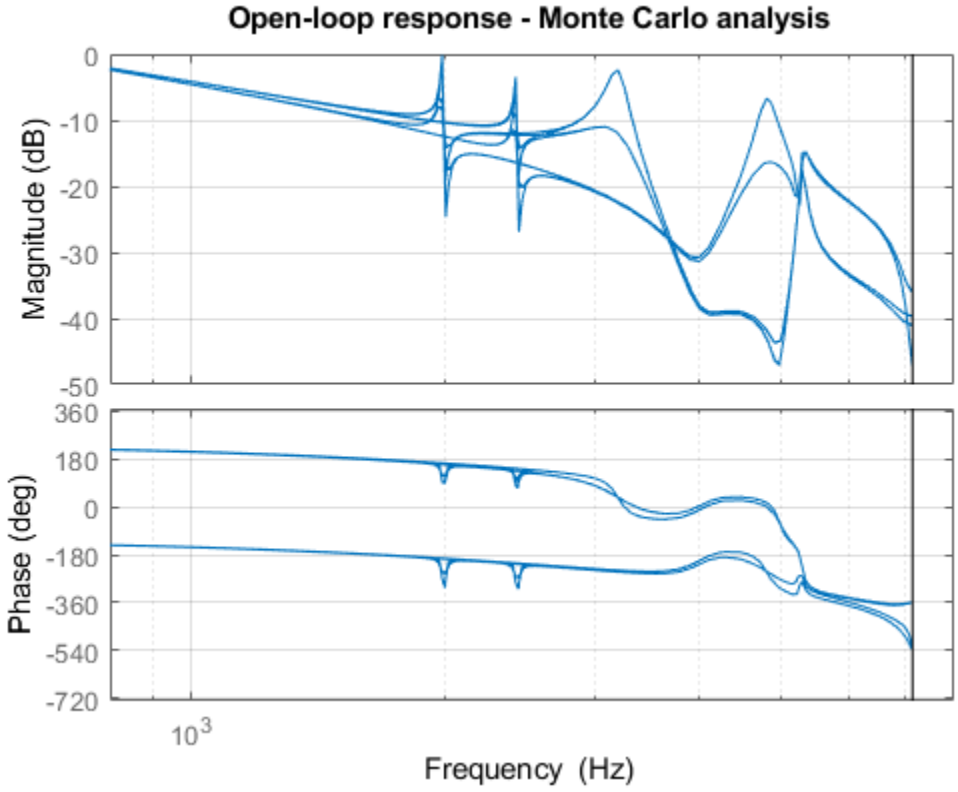
$$\zeta_3 = 0.05 \pm 50\%$$

Generate an array of 16 models corresponding to all combinations of extremal values of z2,w2,z3,w3:

```
[z2,w2,z3,w3] = ndgrid([.5*z2,1.5*z2],[.9*w2,1.1*w2],[.5*z3,1.5*z3],[.8*w3,1.2*w3]);
for j = 1:16,
    Gf21(:,:,j) = tf(w2(j)*[a2 b2*w2(j)] , [1 2*z2(j)*w2(j) w2(j)^2]);
    Gf31(:,:,j) = tf(w3(j)*[a3 b3*w3(j)] , [1 2*z3(j)*w3(j) w3(j)^2]);
end
G1 = Gr * (ss(Gf1) + Gf21 + Gf31 + Gf4);
```

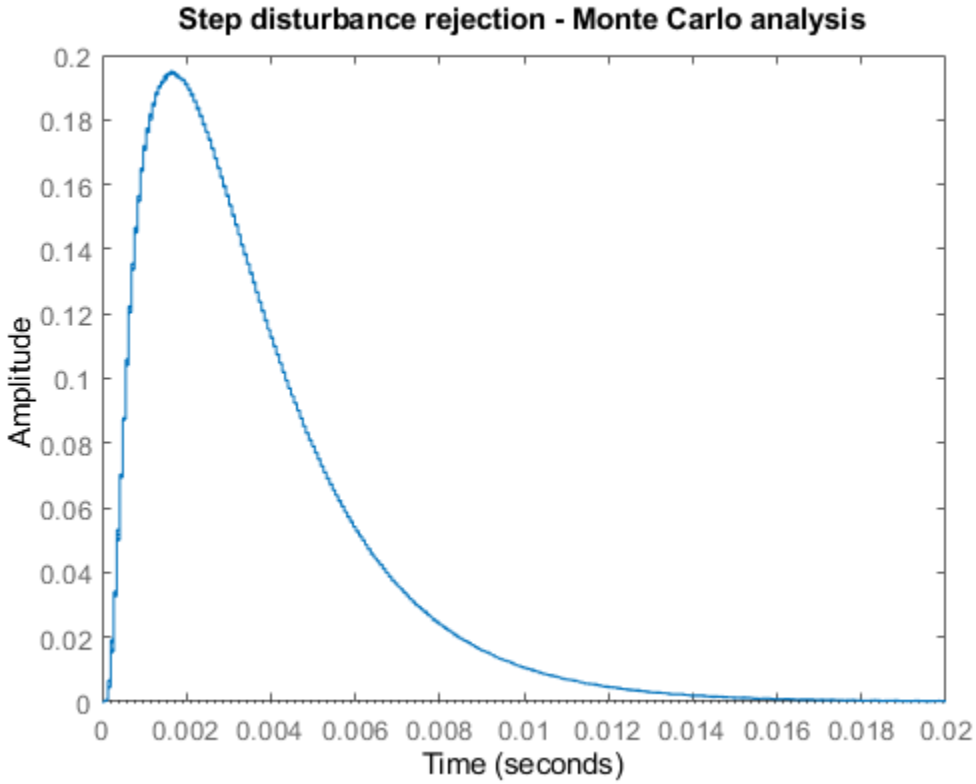
Discretize these 16 models at once and see how the parameter variations affect the open-loop response. Note: You can click on any curve to identify the underlying model.

```
Gd = c2d(G1,Ts);
h = bodeplot(Gd*C2);

title('Open-loop response - Monte Carlo analysis')
setoptions(h,'XLimMode','manual','XLim',{[8e2 8e3]},'YLimMode','auto',...
    'FreqUnits','Hz','MagUnits','dB','PhaseUnits','deg','Grid','on');
```



Plot the step disturbance rejection performance for these 16 models:

```
stepplot(feedback(Gd,C2))
title('Step disturbance rejection - Monte Carlo analysis')
```



All 16 responses are nearly identical: our servo design is robust!

Yaw Damper Design for a 747 Jet Aircraft

This example shows the design of a YAW DAMPER for a 747® aircraft using the classical control design features in Control System Toolbox™.

A simplified trim model of the aircraft during cruise flight

$$\frac{dx}{dt} = Ax + Bu$$

$$y = Cx + Du$$

has four states:

```
beta (sideslip angle), phi (bank angle), yaw rate, roll rate
```

and two inputs: the rudder and aileron deflections.

All angles and angular velocities are in radians and radians/sec.

Given the matrices A,B,C,D of the trim model, use the SS command to create the state-space model in MATLAB®:

```
A=[-.0558 -.9968 .0802 .0415;
     .598 -.115 -.0318 0;
```

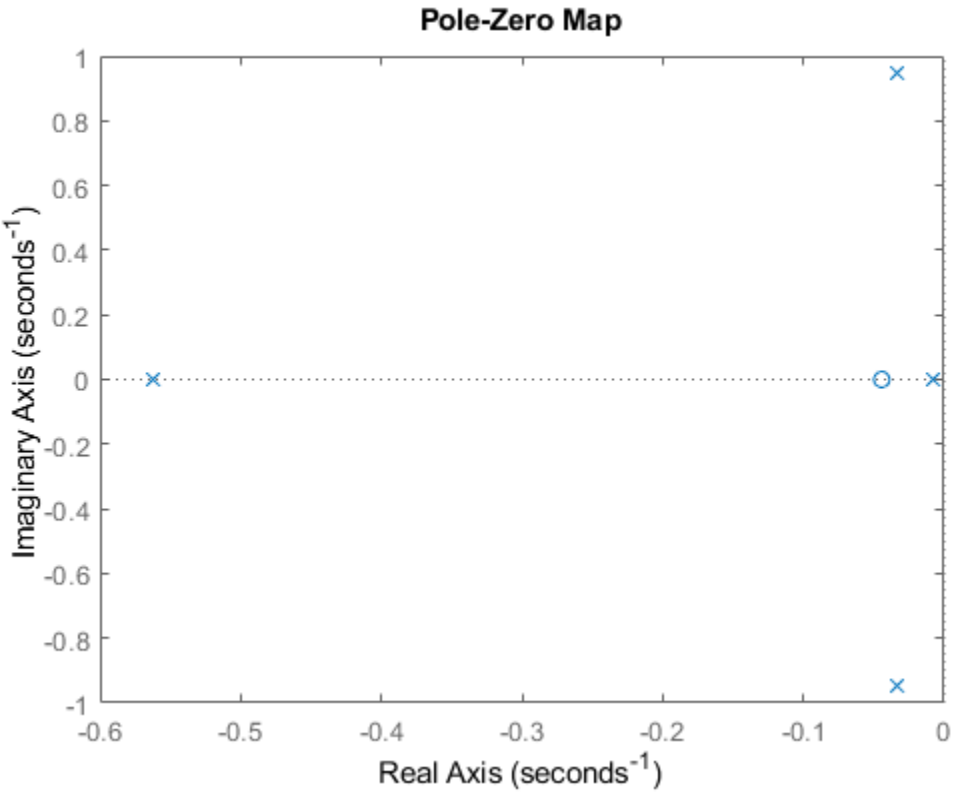
```
    -3.05 .388 -.4650 0;
        0 0.0805 1 0];

B=[ .00729  0;
   -0.475   0.00775;
    0.153   0.143;
     0      0];

C=[0 1 0 0;
   0 0 0 1];

D=[0 0;
   0 0];

sys = ss(A,B,C,D);
```

and label the inputs, outputs, and states:

```
set(sys, 'inputname', {'rudder' 'aileron'},...
         'outputname', {'yaw rate' 'bank angle'},...
         'statename', {'beta' 'yaw' 'roll' 'phi'});
```

This model has a pair of lightly damped poles. They correspond to what is called the Dutch roll mode. To see these modes, type

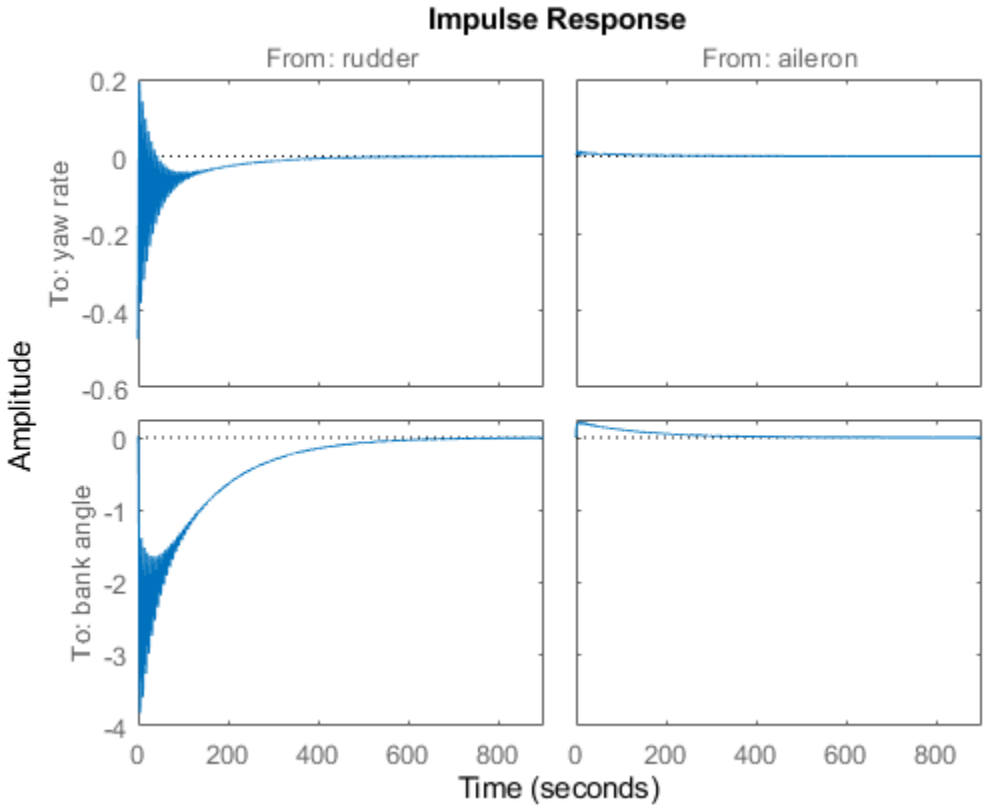
```
axis(gca,'normal')
h = pzplot(sys);
setoptions(h,'FreqUnits','rad/s','Grid','off');
```



Right-click and select "Grid" to plot the damping and natural frequency values. You need to design a compensator that increases the damping of these two poles.

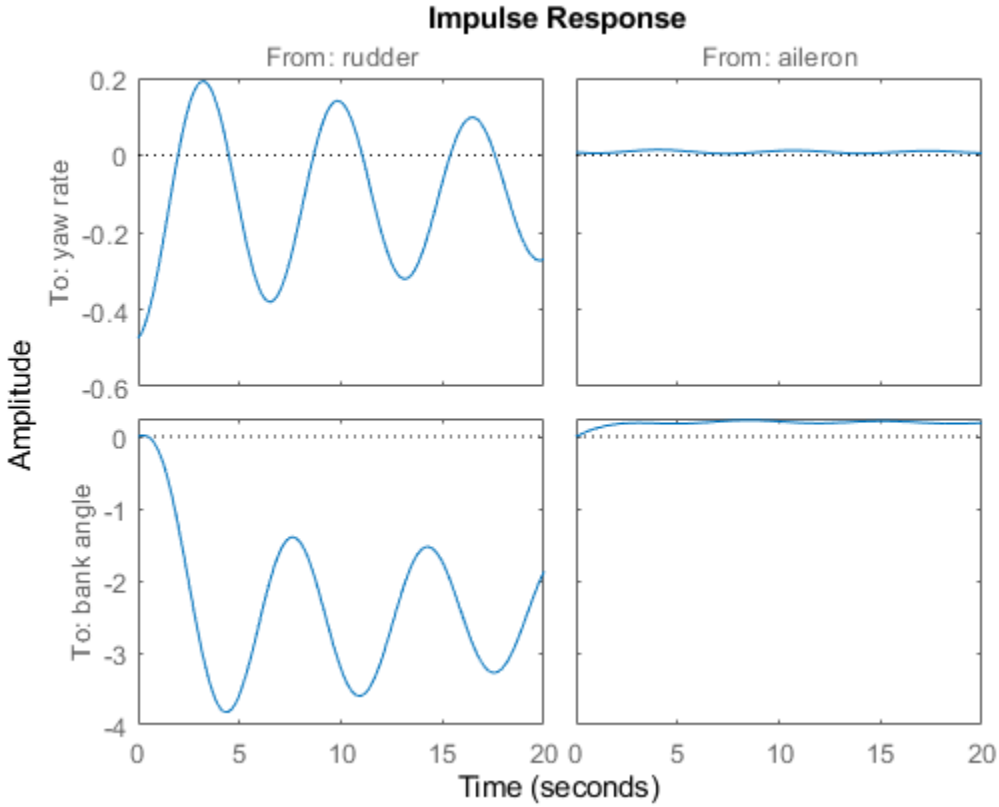
Start with some open loop analysis to determine possible control' strategies. The presence of lightly damped modes is confirmed by ' looking at the impulse response:'

```
impulseplot(sys)
```



To inspect the response over a smaller time frame of 20 seconds, you could also type

```
impulseplot(sys,20)
```

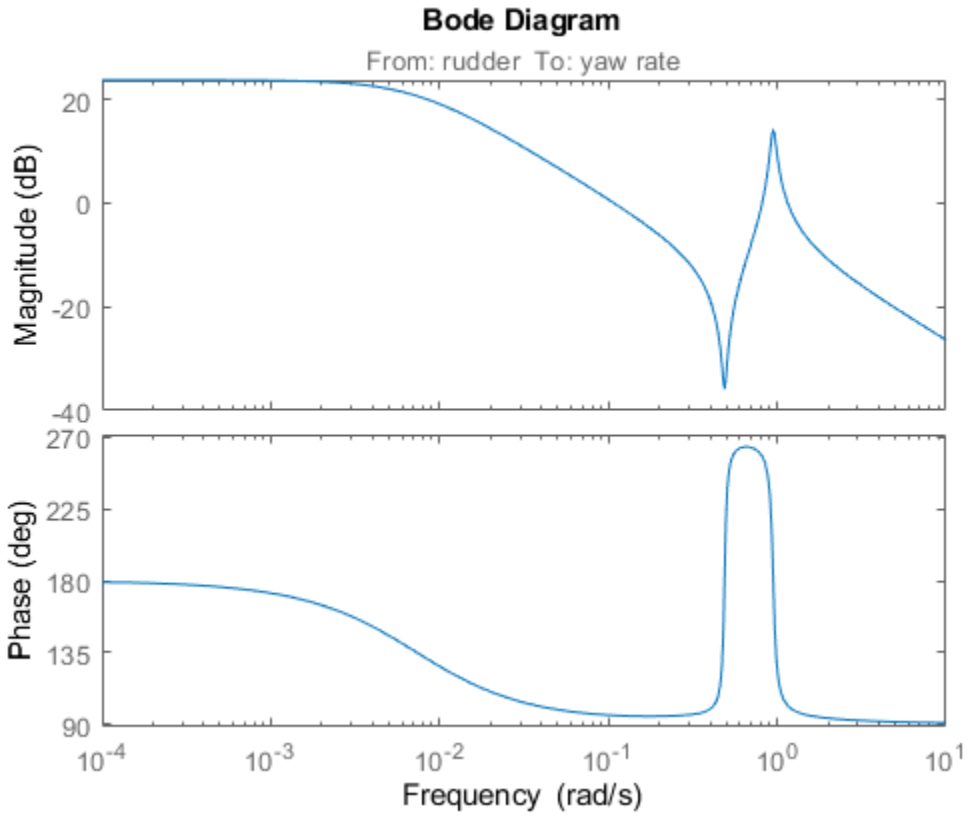


Look at the plot from aileron to bank angle phi. To show only this plot, right-click and choose "I/O Selector", then click on the (2,2) entry.

This plot shows the aircraft oscillating around a non-zero bank angle. Thus the aircraft turns in response to an aileron impulse. This behavior will be important later.

Typically yaw dampers are designed using yaw rate as the sensed output and rudder as the input. Inspect the frequency response for this I/O pair:

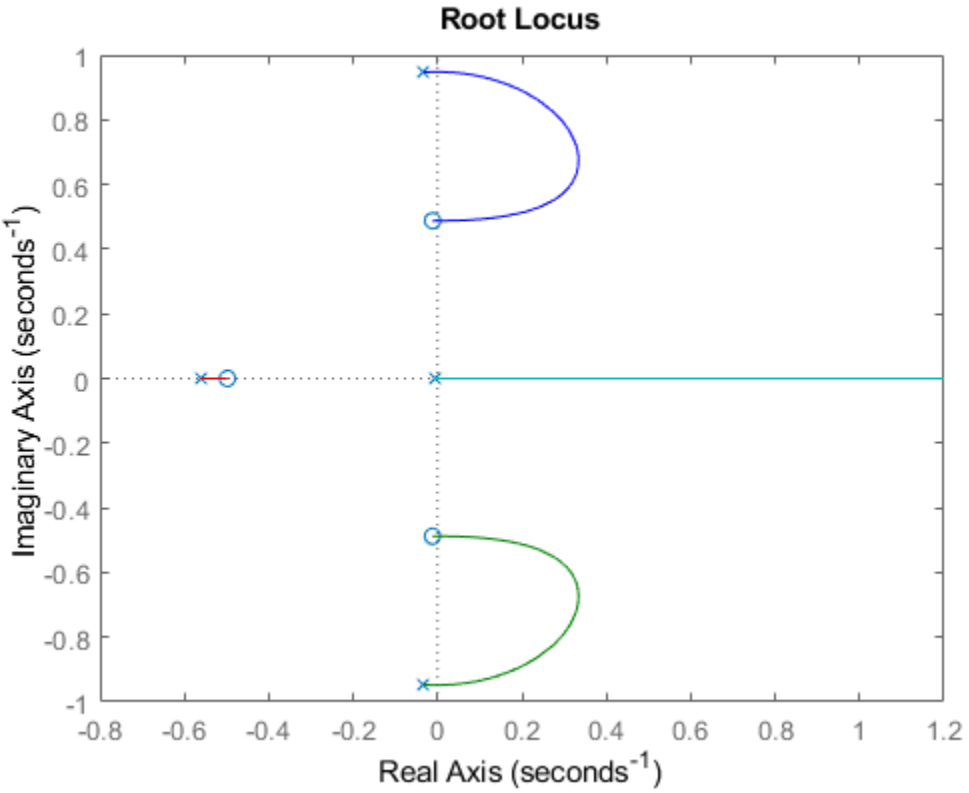
```
sys11 = sys('yaw','rudder');    % select I/O pair
h = bodeplot(sys11);
setoptions(h, 'FreqUnits','rad/s','MagUnits','dB','PhaseUnits','deg');
```



This plot shows that the rudder has a lot of authority around the lightly damped Dutch roll mode (1 rad/s).

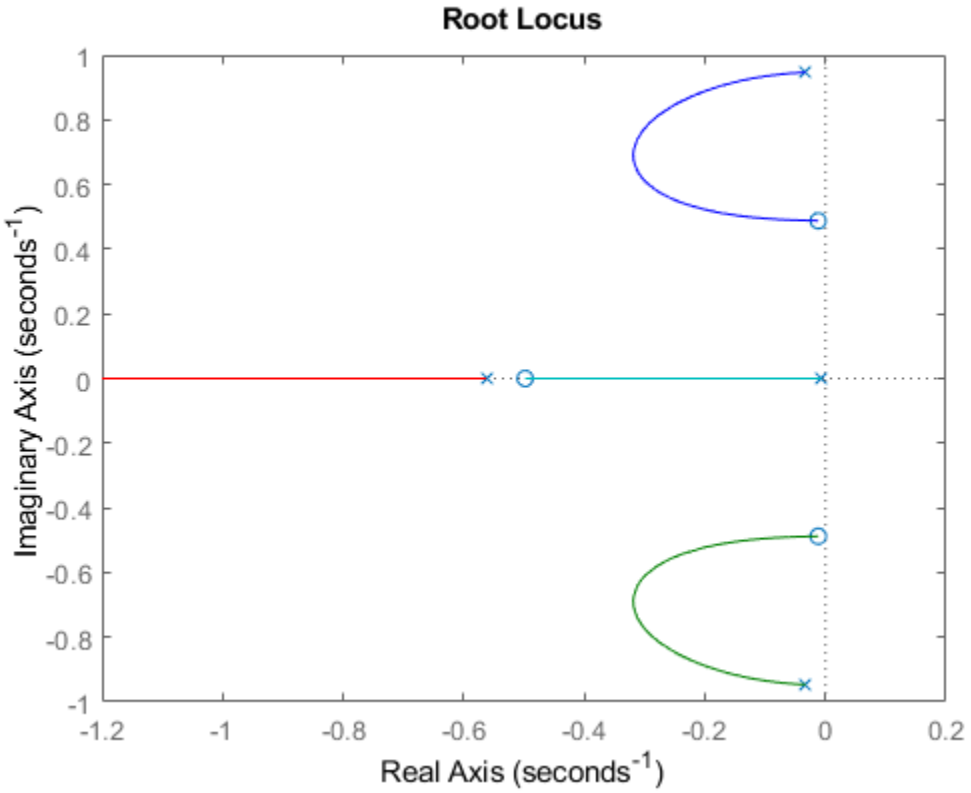
A reasonable design objective is to provide a damping ratio zeta > 0.35, with natural frequency Wn < 1.0 rad/s. The simplest compensator is a gain. Use the root locus technique to select an adequate feedback gain value:

```
h = rlocusplot(sys11);
setoptions(h,'FreqUnits','rad/s')
```



Oops, looks like we need positive feedback!

```
h = rlocusplot(-sys11);
setoptions(h,'FreqUnits','rad/s')
```



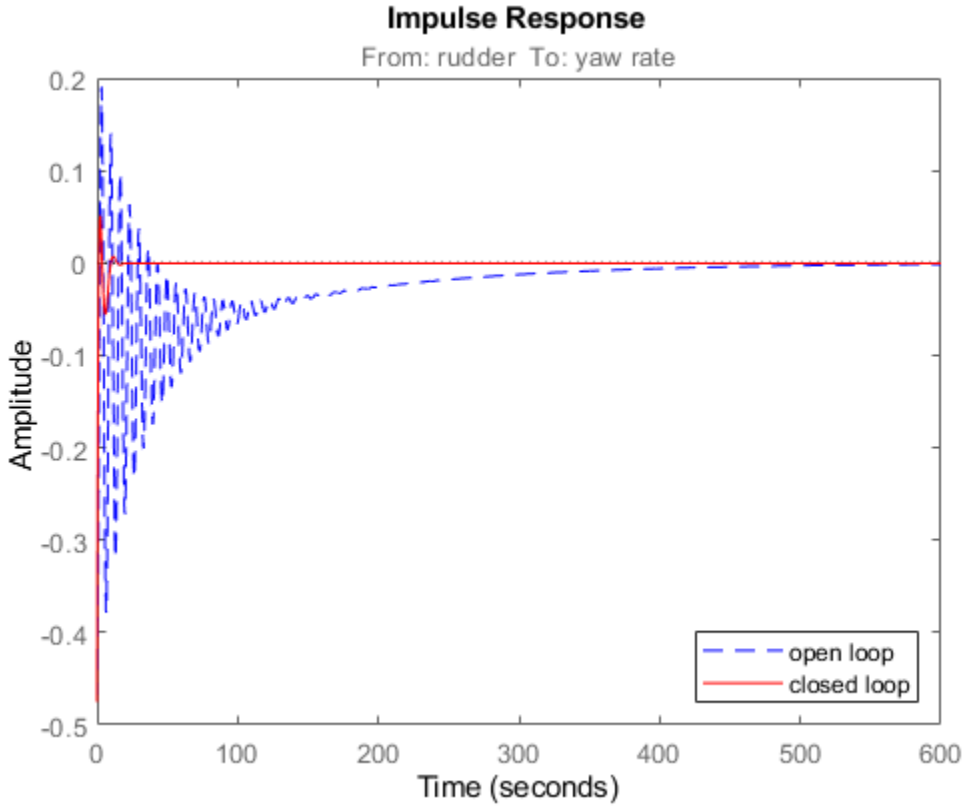
This looks better. Click on the blue curve and move the black square to track the gain and damping values. The best achievable closed-loop damping is about 0.45 for a gain of K=2.85.

Now close this SISO feedback loop and look at the impulse response

```
k = 2.85;
cl11 = feedback(sys11,-k);
```

Note: feedback assumes negative feedback by default

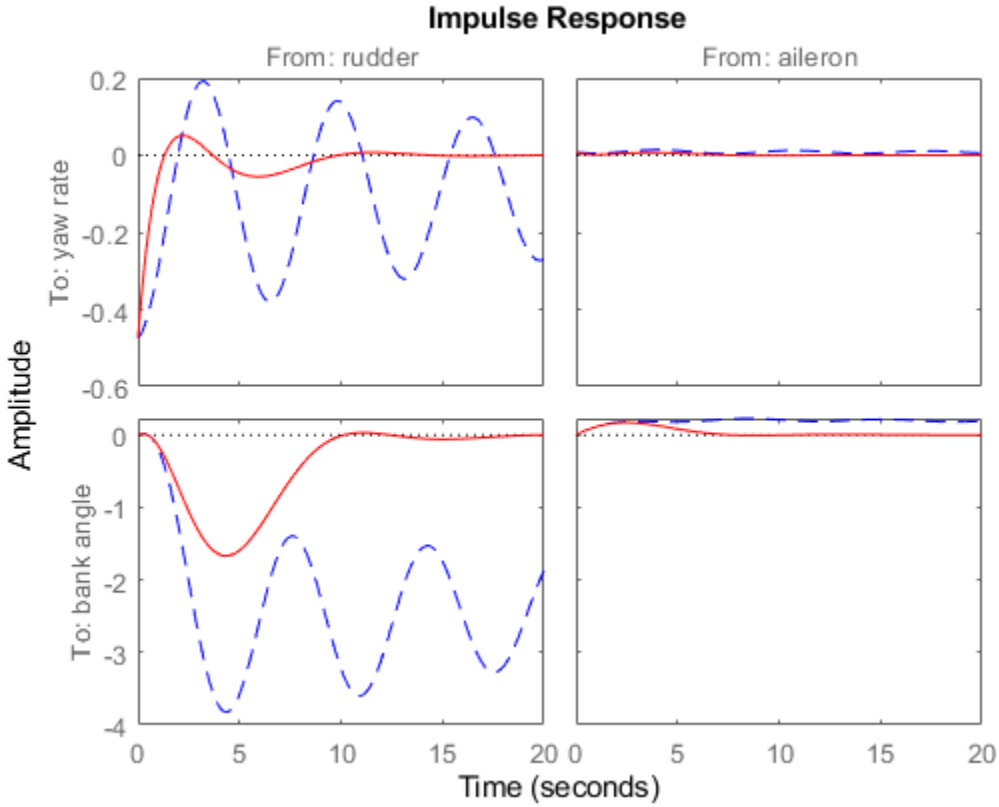
```
impulseplot(sys11,'b--',cl11,'r')
legend('open loop','closed loop','Location','SouthEast')
```



The response looks pretty good.

Now close the loop around the full MIMO model and see how the response from the aileron looks. The feedback loop involves input 1 and output 1 of the plant:

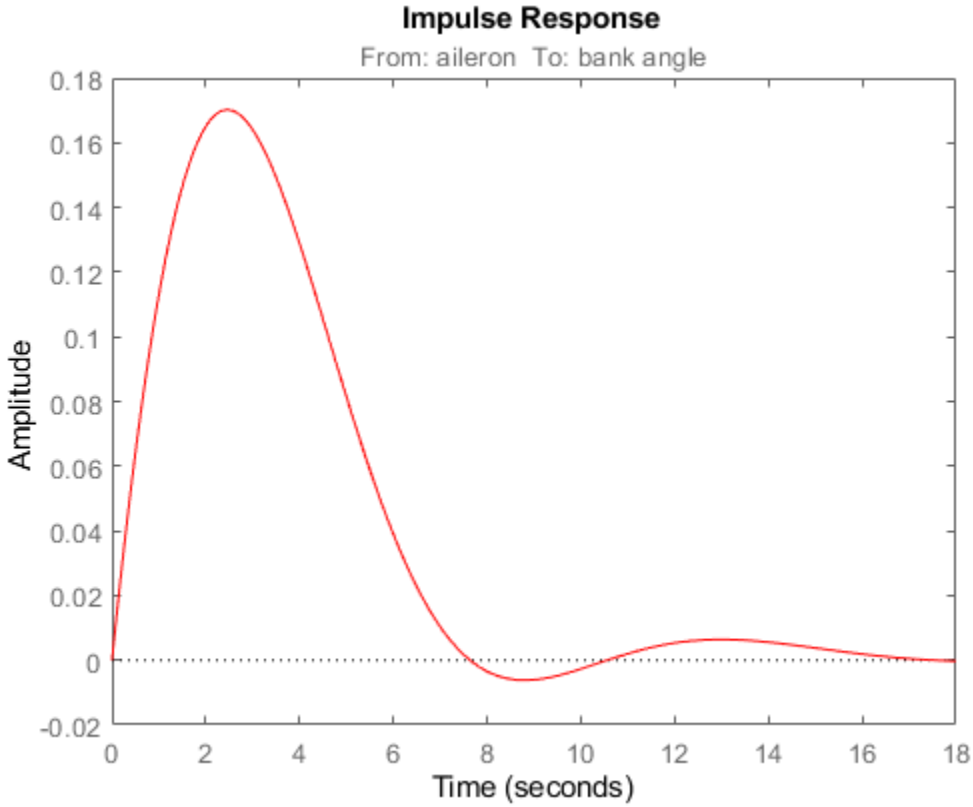
```
cloop = feedback(sys,-k,1,1);
impulseplot(sys,'b--',cloop,'r',20)           % MIMO impulse response
```



The yaw rate response is now well damped.

When moving the aileron, however, the system no longer continues to bank like a normal aircraft, as seen from

```
impulseplot(cloop('bank angle','aileron'),'r',18)
```



You have over-stabilized the spiral mode. The spiral mode is typically a very slow mode that allows the aircraft to bank and turn without constant aileron input. Pilots are used to this behavior and will not like a design that does not fly normally.

You need to make sure that the spiral mode doesn't move farther into the left-half plane when we close the loop. One way flight control designers have fixed this problem is by using a washout filter.

Washout Filter:

$$H(s) = \frac{ks}{s+a}$$

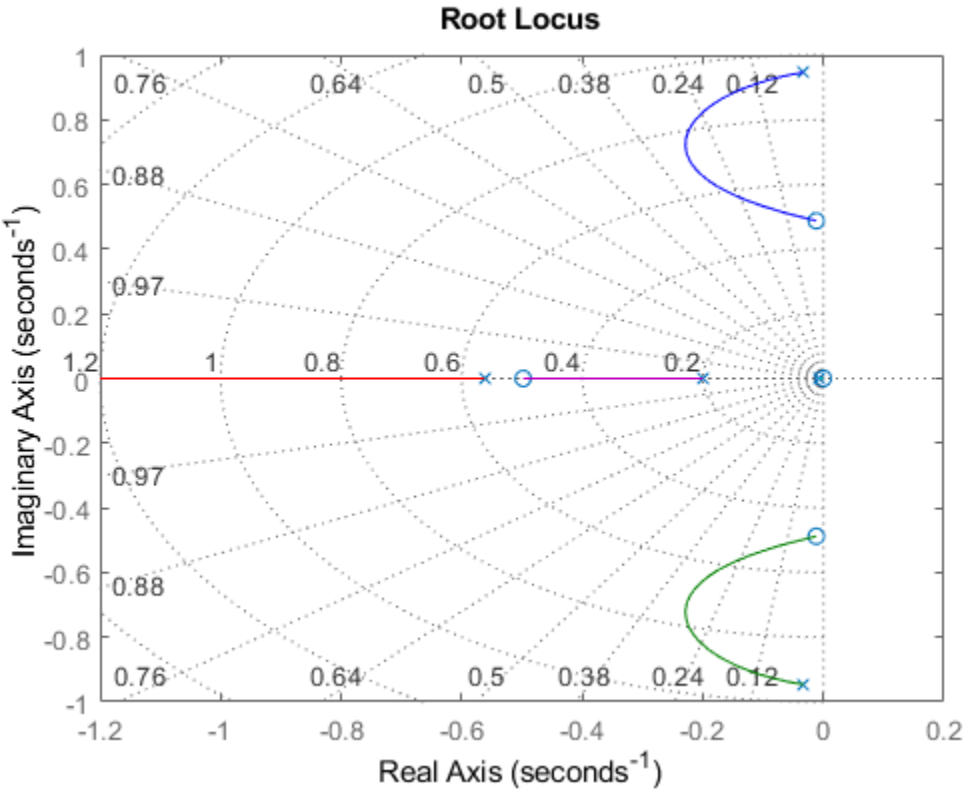
Using the SISO Design Tool (help sisotool), you can graphically tune the parameters k and a to find the best combination. In this example we choose a = 0.2 or a time constant of 5 seconds.

Form the washout filter for a=0.2 and k=1

```
H = zpk(0,-0.2,1);
```

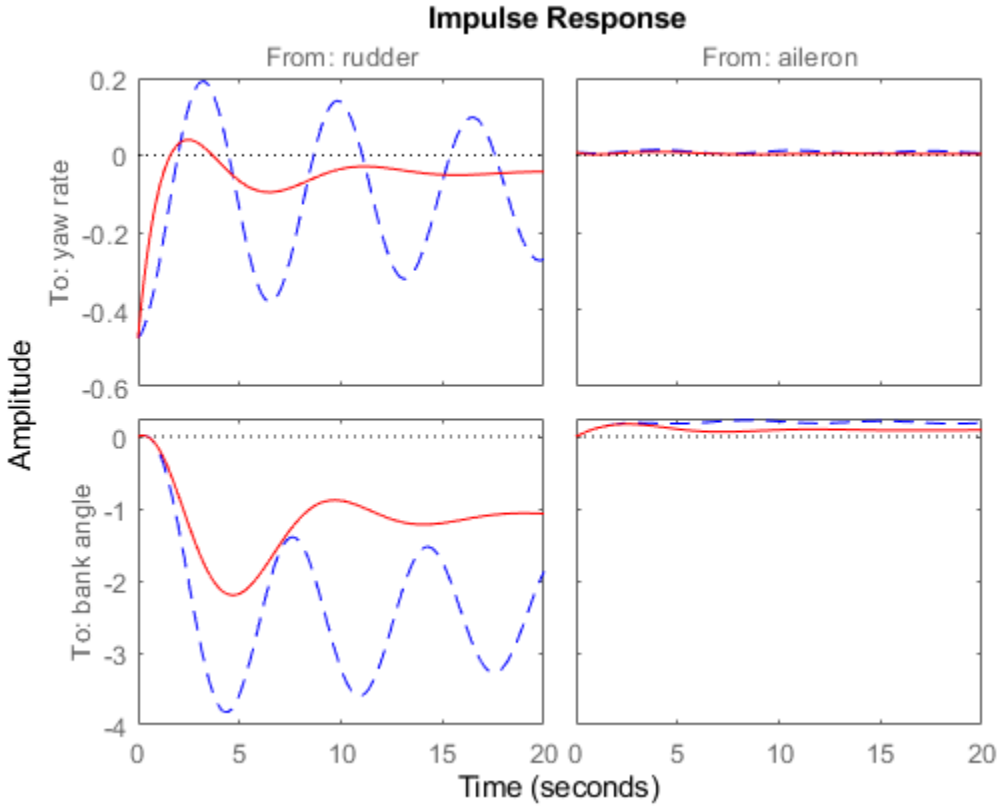
connect the washout in series with your design model, and use the root locus to determine the filter gain k:

```
oloop = H * (-sys11);          % open loop'
h = rlocusplot(oloop);
setoptions(h, 'FreqUnits','rad/s')
sgrid
```



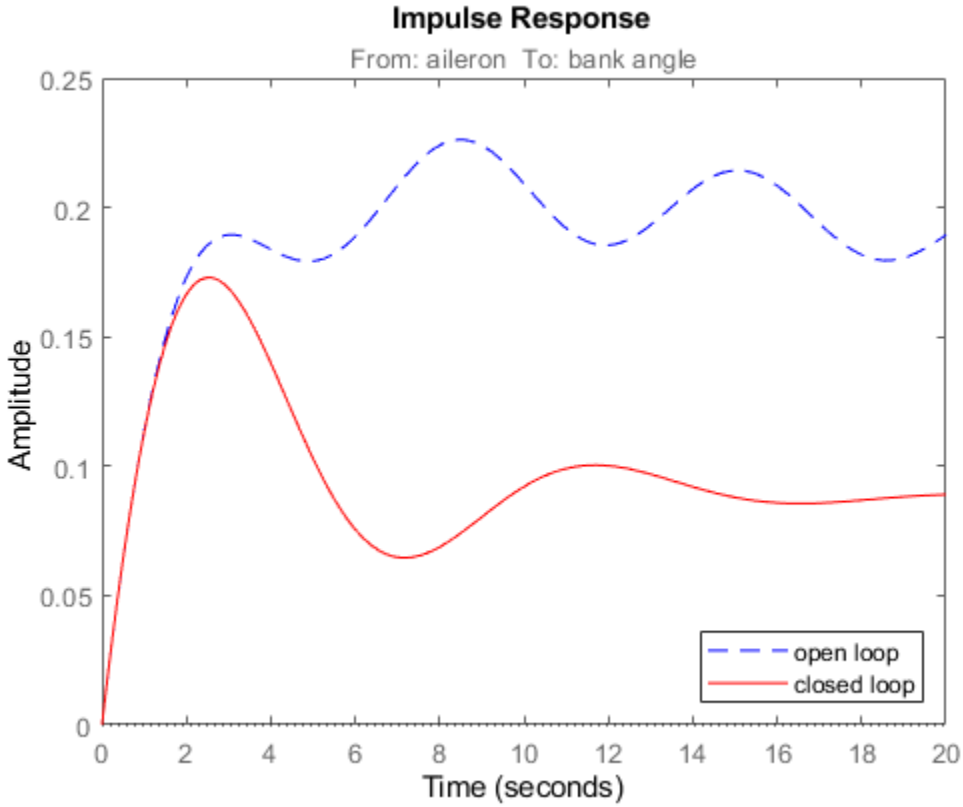
The best damping is now about zeta = 0.305 for k=2.34. Close the loop with the MIMO model and check the impulse response:

```
k = 2.34;
wof = -k * H;            % washout compensator
cloop = feedback(sys,wof,1,1);
impulseplot(sys,'b--',cloop,'r',20)
```



The washout filter has also restored the normal bank-and-turn behavior as seen by looking at the impulse response from aileron to bank angle.

```
impulseplot(sys(2,2),'b--',cloop(2,2),'r',20)
legend('open loop','closed loop','Location','SouthEast')
```



Although it doesn't quite meet the requirements, this design substantially increases the damping while allowing the pilot to fly the aircraft normally.

Thickness Control for a Steel Beam

This example shows how to design a MIMO LQG regulator to control the horizontal and vertical thickness of a steel beam in a hot steel rolling mill.

Rolling Stand Model

Figures 1 and 2 depict the process of shaping a beam of hot steel by compressing it with rolling cylinders.

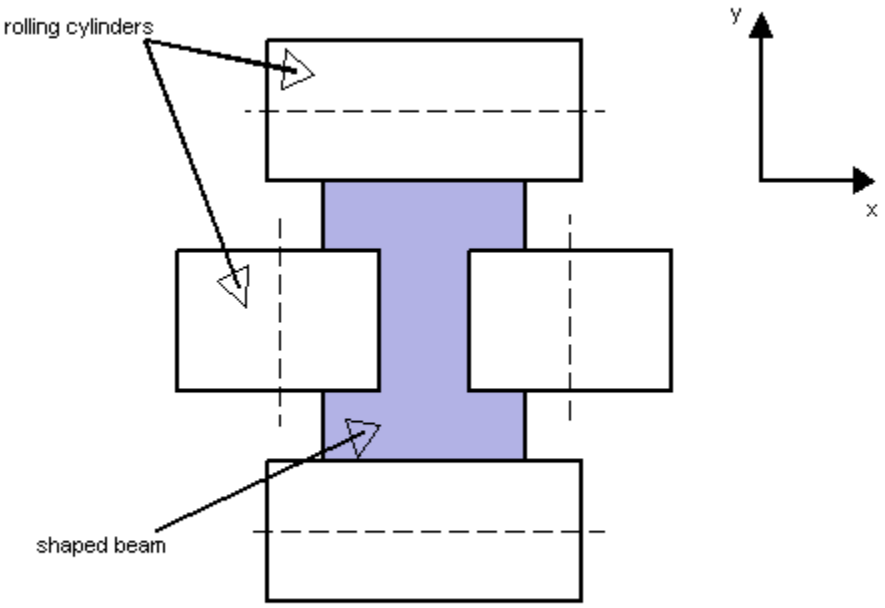


Figure 1: Beam Shaping by Rolling Cylinders.

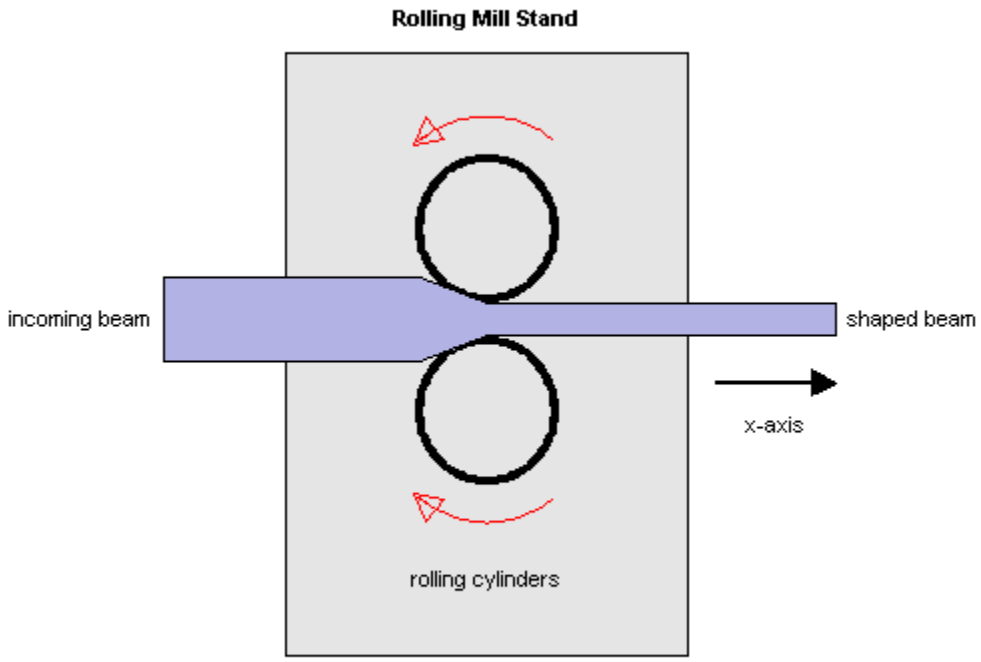


Figure 2: Rolling Mill Stand.

The desired H shape is impressed by two pairs of rolling cylinders (one per axis) positioned by hydraulic actuators. The gap between the two cylinders is called the roll gap. The goal is to maintain the x and y thickness within specified tolerances. Thickness variations arise primarily from variations in thickness and hardness of the incoming beam (input disturbance) and eccentricities of the rolling cylinders.

An open-loop model for the x or y axes is shown in Figure 3. The eccentricity disturbance is modeled as white noise `w_e` driving a band-pass filter `Fe`. The input thickness disturbance is modeled as white noise `w_i` driving a low-pass filter `Fi`. Feedback control is necessary to counter such disturbances. Because the roll gap `delta` cannot be measured close to the stand, the rolling force `f` is used for feedback.

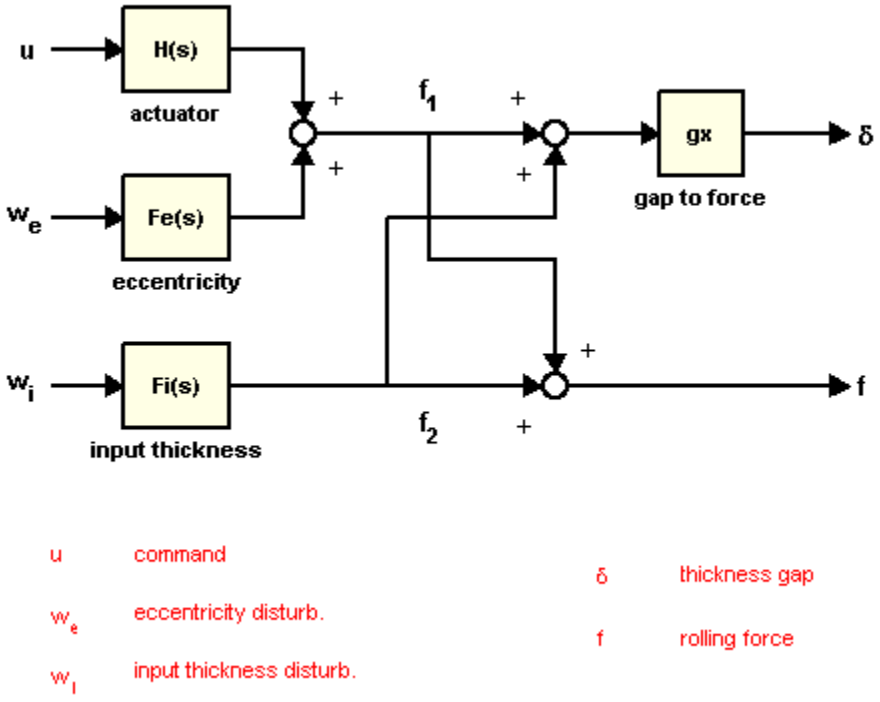


Figure 3: Open-Loop Model.

Building the Open-Loop Model

Empirical models for the filters `Fe` and `Fi` for the x axis are

$$F_{ex} = \frac{3 \times 10^4 s}{s^2 + 0.125s + 6^2}, \quad F_{ix} = \frac{10^4}{s + 0.05}$$

and the actuator and gap-to-force gain are modeled as

$$H_x = \frac{2.4 \times 10^8}{s^2 + 72s + 90^2}, \quad g_x = 10^{-6}$$

To construct the open-loop model in Figure 3, start by specifying each block:

```
Hx = tf(2.4e8 , [1  72  90^2] , 'inputname' , 'u_x');
Fex = tf([3e4 0] , [1 0.125 6^2] , 'inputname' , 'w_{ex}');
Fix = tf(1e4 , [1 0.05] , 'inputname' , 'w_{ix}');
gx = 1e-6;
```

Next construct the transfer function from `u,we,wi` to `f1,f2` using concatenation and `append` as follows. To improve numerical accuracy, switch to the state-space representation before you connect models:

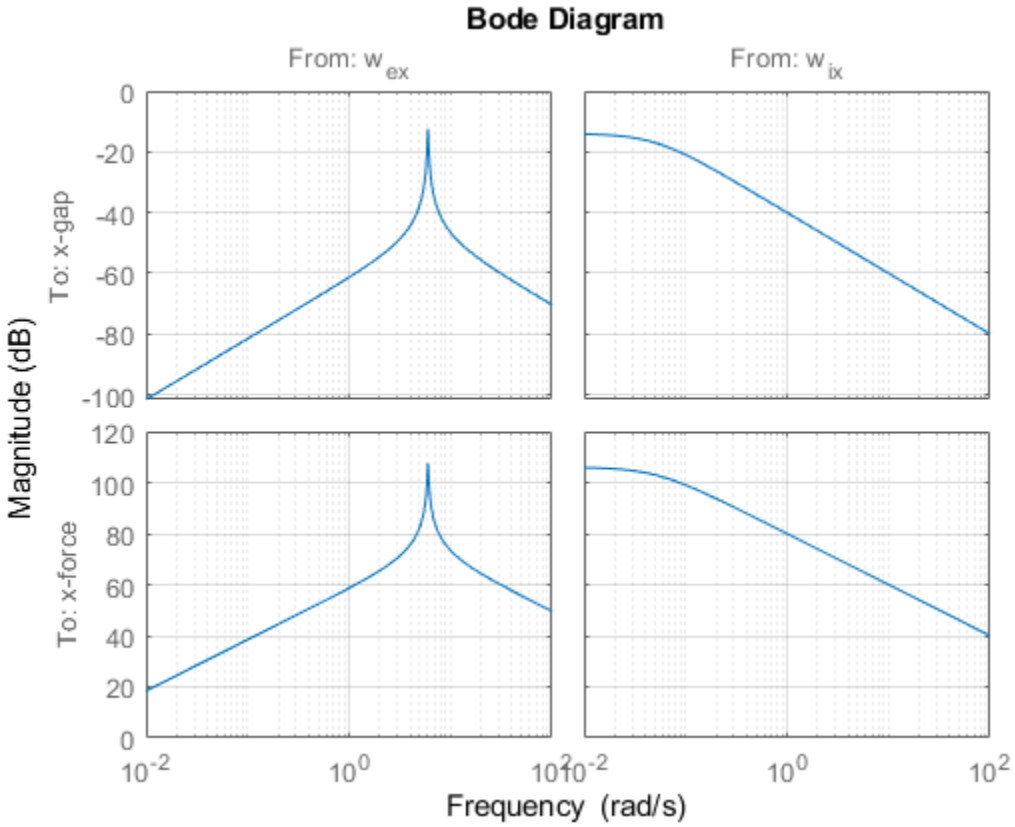
```
T = append([ss(Hx) Fex],Fix);
```

Finally, apply the transformation mapping `f1,f2` to `delta,f`:

```
Px = [-gx gx;1 1] * T;
Px.OutputName = {'x-gap' , 'x-force'};
```

Plot the frequency response magnitude from the normalized disturbances `w_e` and `w_i` to the outputs:

```
bodemag(Px(: , [2 3]),{1e-2,1e2}), grid
```



Note the peak at 6 rad/sec corresponding to the (periodic) eccentricity disturbance.

LQG Regulator Design for the X Axis

First design an LQG regulator to attenuate the thickness variations due to the eccentricity and input thickness disturbances `w_e` and `w_i`. LQG regulators generate actuator commands u = -K x_e where x_e is an estimate of the plant states. This estimate is derived from available measurements of the rolling force `f` using an observer called "Kalman filter."

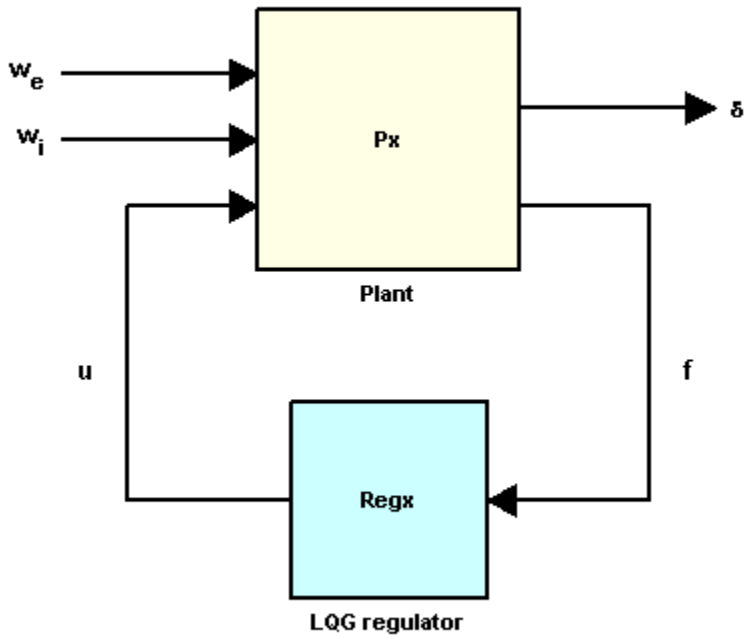


Figure 4: LQG Control Structure.

Use `lqry` to calculate a suitable state-feedback gain K. The gain K is chosen to minimize a cost function of the form

$$C(u) = \int_0^\infty \left(\delta^2(t) + \beta u^2(t)\right) dt$$

where the parameter `beta` is used to trade off performance and control effort. For `beta` = 1e-4, you can compute the optimal gain by typing

```
Pxdes = Px('x-gap','u_x');      % transfer u_x -> x-gap
Kx = lqry(Pxdes,1,1e-4)
```

```
Kx =

    0.0621    0.1315    0.0222   -0.0008   -0.0074
```

Next, use `kalman` to design a Kalman estimator for the plant states. Set the measurement noise covariance to 1e4 to limit the gain at high frequencies:

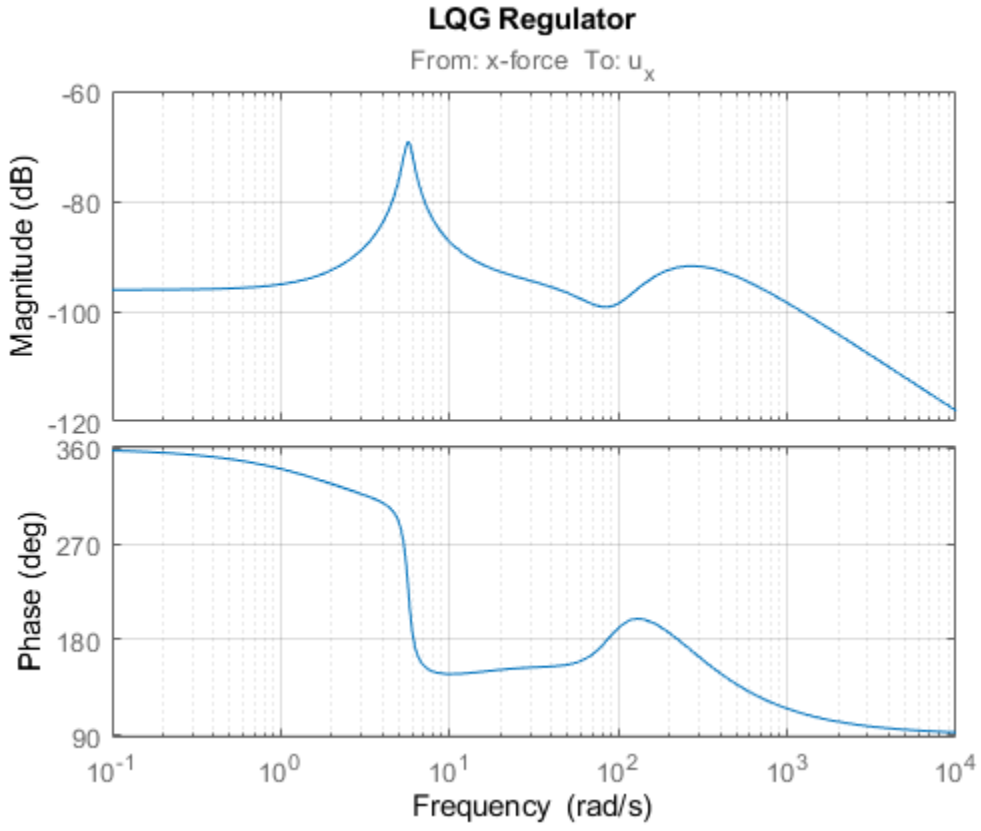
```
Ex = kalman(Px('x-force',:),eye(2),1e4);
```

Finally, use `lqgreg` to assemble the LQG regulator `Regx` from `Kx` and `Ex`:

```
Regx = lqgreg(Ex,Kx);
zpk(Regx)
```

```
ans =

  From input "x-force" to output "u_x":
       -0.012546 (s+10.97) (s-2.395) (s^2 + 72s + 8100)
  ------------------------------------------------------------
  (s+207.7) (s^2 + 0.738s + 32.33) (s^2 + 310.7s + 2.536e04)

Input groups:
       Name        Channels
    Measurement       1

Output groups:
      Name      Channels
    Controls       1

Continuous-time zero/pole/gain model.
```

```
bode(Regx),
grid, title('LQG Regulator')
```



LQG Regulator Evaluation

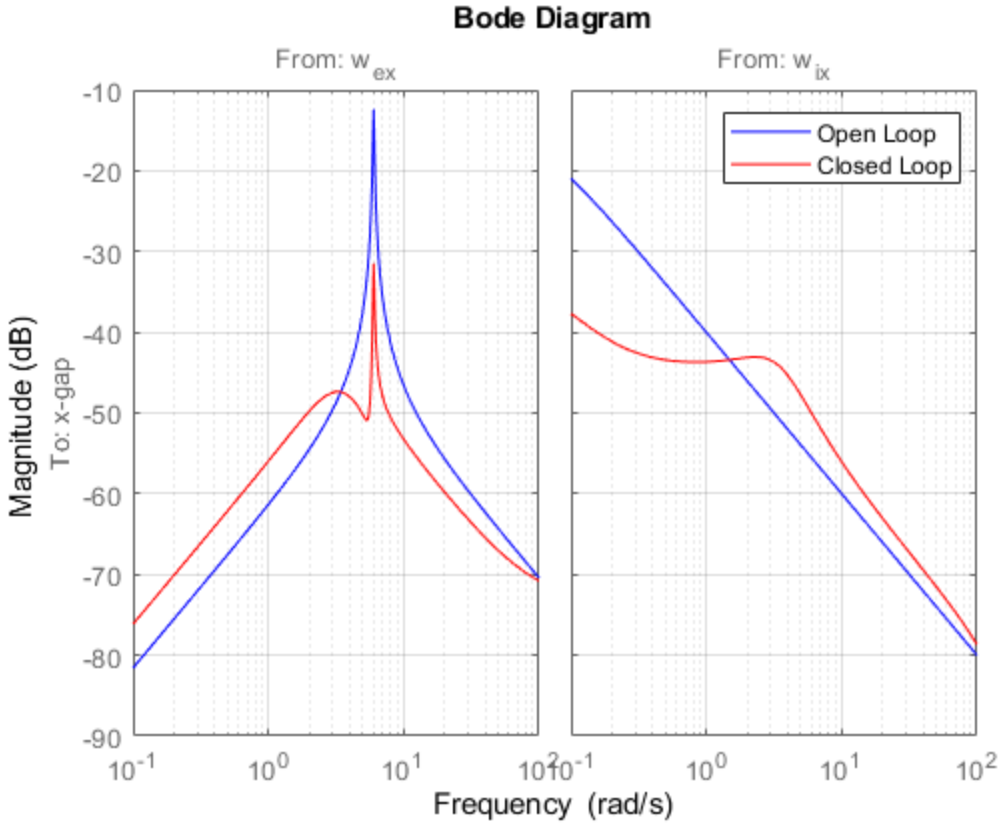
Close the regulation loop shown in Figure 4:

```
clx = feedback(Px,Regx,1,2,+1);
```

Note that in this command, the +1 accounts for the fact that `lqgreg` computes a positive feedback compensator.

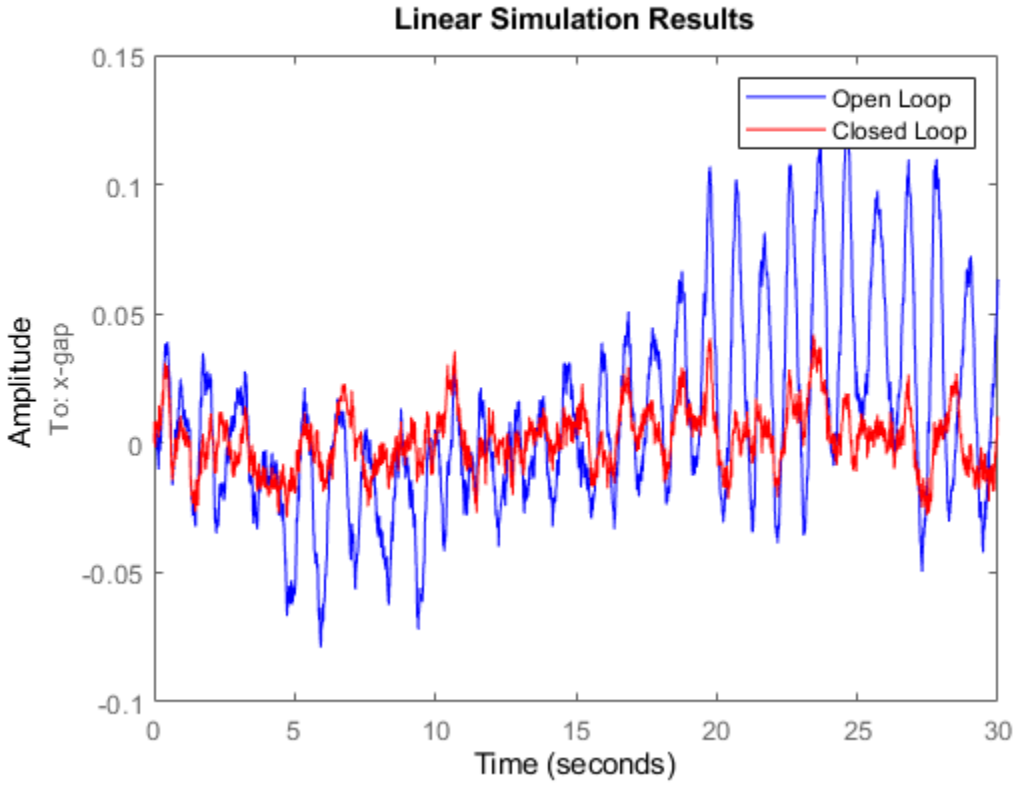
You can now compare the open- and closed-loop responses to eccentricity and input thickness disturbances:

```
bodemag(Px(1,2:3),'b',clx(1,2:3),'r',{1e-1,1e2})
grid, legend('Open Loop','Closed Loop')
```



The Bode plot indicates a 20 dB attenuation of disturbance effects. You can confirm this by simulating disturbance-induced thickness variations with and without the LQG regulator as follows:

```
dt = 0.01;  % simulation time step
t = 0:dt:30;
wx = sqrt(1/dt) * randn(2,length(t));    % sampled driving noise

h = lsimplot(Px(1,2:3),'b',clx(1,2:3),'r',wx,t);
h.Input.Visible = 'off';
legend('Open Loop','Closed Loop')
```



Two-Axis Design

You can design a similar LQG regulator for the y axis. Use the following actuator, gain, and disturbance models:

```
Hy = tf(7.8e8,[1 71 88^2],'inputname','u_y');
Fiy = tf(2e4,[1 0.05],'inputname','w_{iy}');
Fey = tf([1e5 0],[1 0.19 9.4^2],'inputn','w_{ey}');
gy = 0.5e-6;
```

You can construct the open-loop model by typing

```
Py = append([ss(Hy) Fey],Fiy);
Py = [-gy gy;1 1] * Py;
Py.OutputName = {'y-gap' 'y-force'};
```

You can then compute the corresponding LQG regulator by typing

```
ky = lqry(Py(1,1),1,1e-4);
Ey = kalman(Py(2,:),eye(2),1e4);
Regy = lqgreg(Ey,ky);
```

Assuming the x- and y-axis are decoupled, you can use these two regulators independently to control the two-axis rolling mill.

Cross-Coupling Effects

Treating each axis separately is valid as long as they are fairly decoupled. Unfortunately, rolling mills have some amount of cross-coupling between axes because an increase in force along x compresses the material and causes a relative decrease in force along the y axis.

Cross-coupling effects are modeled as shown in Figure 5 with gxy=0.1 and gyx=0.4.

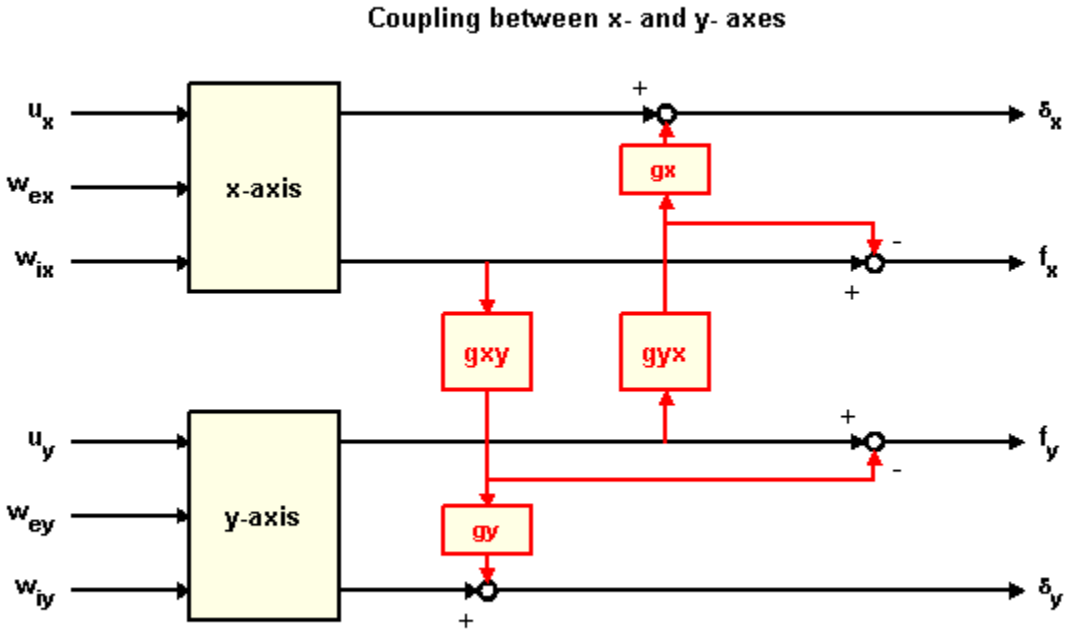


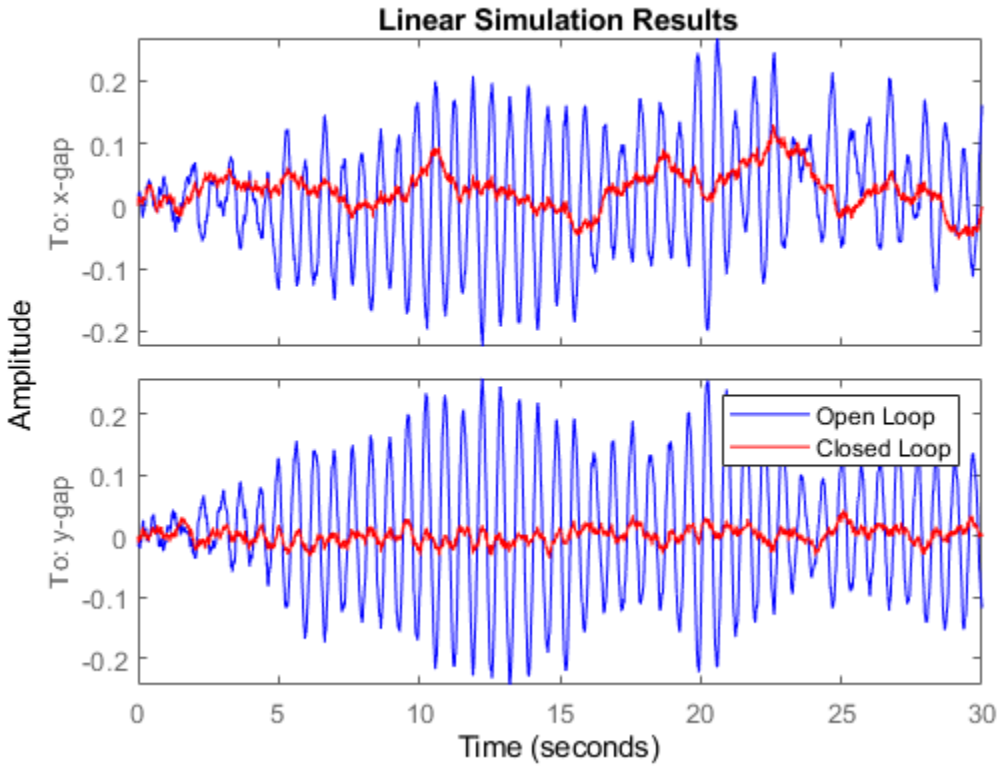
Figure 5: Cross-Coupling Model.

To study the effect of cross-coupling on decoupled SISO loops, construct the two-axis model in Figure 5 and close the x- and y-axis loops using the previously designed LQG regulators:

```
gxy = 0.1;
gyx = 0.4;
P = append(Px,Py);                % Append x- and y-axis models
P = P([1 3 2 4],[1 4 2 3 5 6]);   % Reorder inputs and outputs
CC = [1   0    0    gyx*gx  ;...  % Cross-coupling matrix
      0   1  gxy*gy    0    ;...
      0   0    1      -gyx  ;...
      0   0   -gxy     1  ];
Pxy = CC * P;                     % Cross-coupling model
Pxy.outputn = P.outputn;

clxy0 = feedback(Pxy,append(Regx,Regy),1:2,3:4,+1);
```

Now, simulate the x and y thickness gaps for the two-axis model:

```
wy = sqrt(1/dt) * randn(2,length(t));     % y-axis disturbances
wxy = [wx ; wy];

h = lsimplot(Pxy(1:2,3:6),'b',clxy0(1:2,3:6),'r',wxy,t);
h.Input.Visible = 'off';
legend('Open Loop','Closed Loop')
```



Note the high thickness variations along the x axis. Treating each axis separately is inadequate and you need to use a joint-axis, MIMO design to correctly handle cross-coupling effects.

MIMO Design

The MIMO design consists of a single regulator that uses both force measurements `fx` and `fy` to compute the actuator commands, `u_x` and `u_y`. This control architecture is depicted in Figure 6.

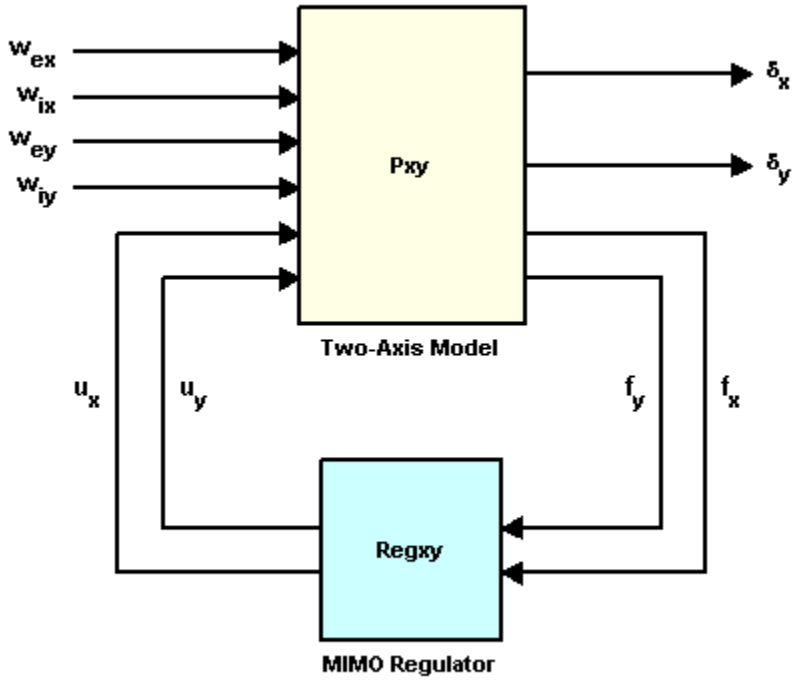


Figure 6: MIMO Control Structure.

You can design a MIMO LQG regulator for the two-axis model using the exact same steps as for earlier SISO designs. First, compute the state feedback gain, then compute the state estimator, and finally assemble these two components using `lqgreg`. Use the following commands to perform these steps:

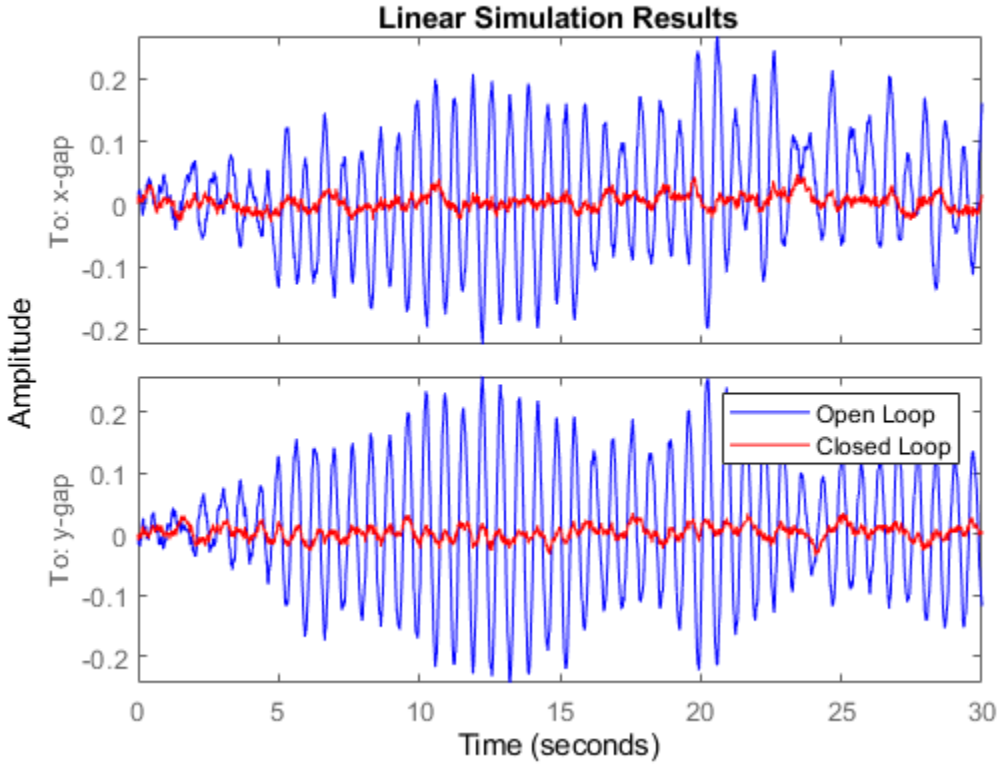
```
Kxy = lqry(Pxy(1:2,1:2),eye(2),1e-4*eye(2));
Exy = kalman(Pxy(3:4,:),eye(4),1e4*eye(2));
Regxy = lqgreg(Exy,Kxy);
```

To compare the performance of the MIMO and multi-loop SISO designs, close the MIMO loop in Figure 6:

```
clxy = feedback(Pxy,Regxy,1:2,3:4,+1);
```

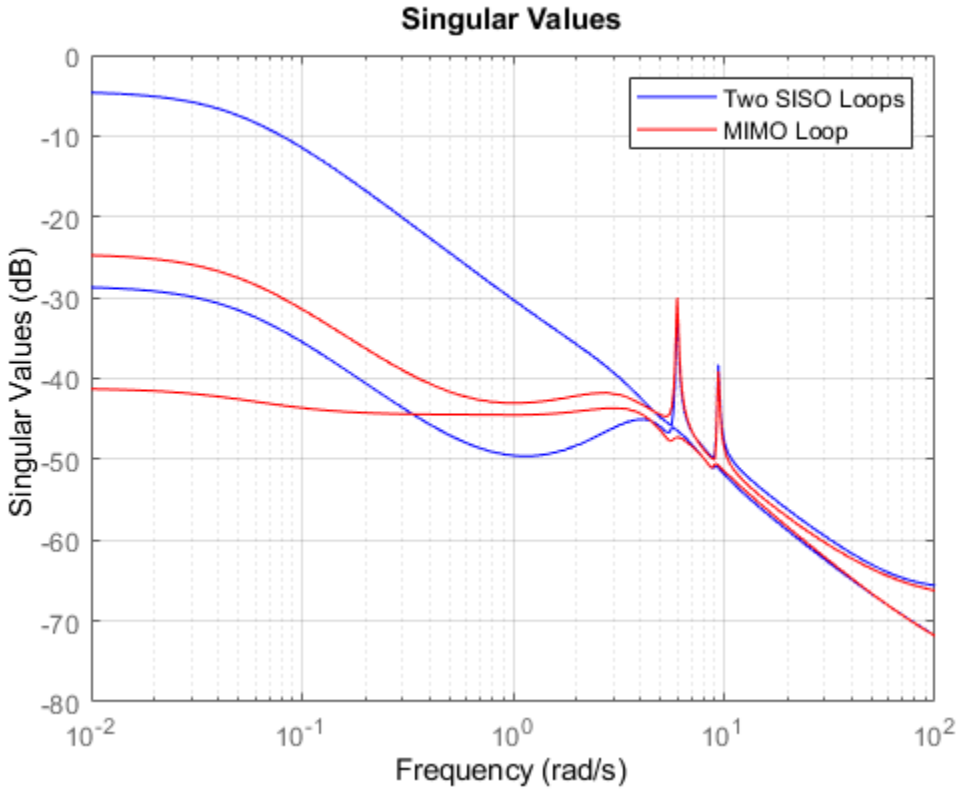
Then, simulate the x and y thickness gaps for the two-axis model:

```
h = lsimplot(Pxy(1:2,3:6),'b',clxy(1:2,3:6),'r',wxy,t);
h.Input.Visible = 'off';
legend('Open Loop','Closed Loop')
```



The MIMO design shows no performance loss in the x axis and the disturbance attenuation levels now match those obtained for each individual axis. The improvement is also evident when comparing the principal gains of the closed-loop responses from input disturbances to thickness gaps `x-gap, y-gap`:

```
sigma(clxy0(1:2,3:6),'b',clxy(1:2,3:6),'r',{1e-2,1e2})
grid, legend('Two SISO Loops','MIMO Loop')
```

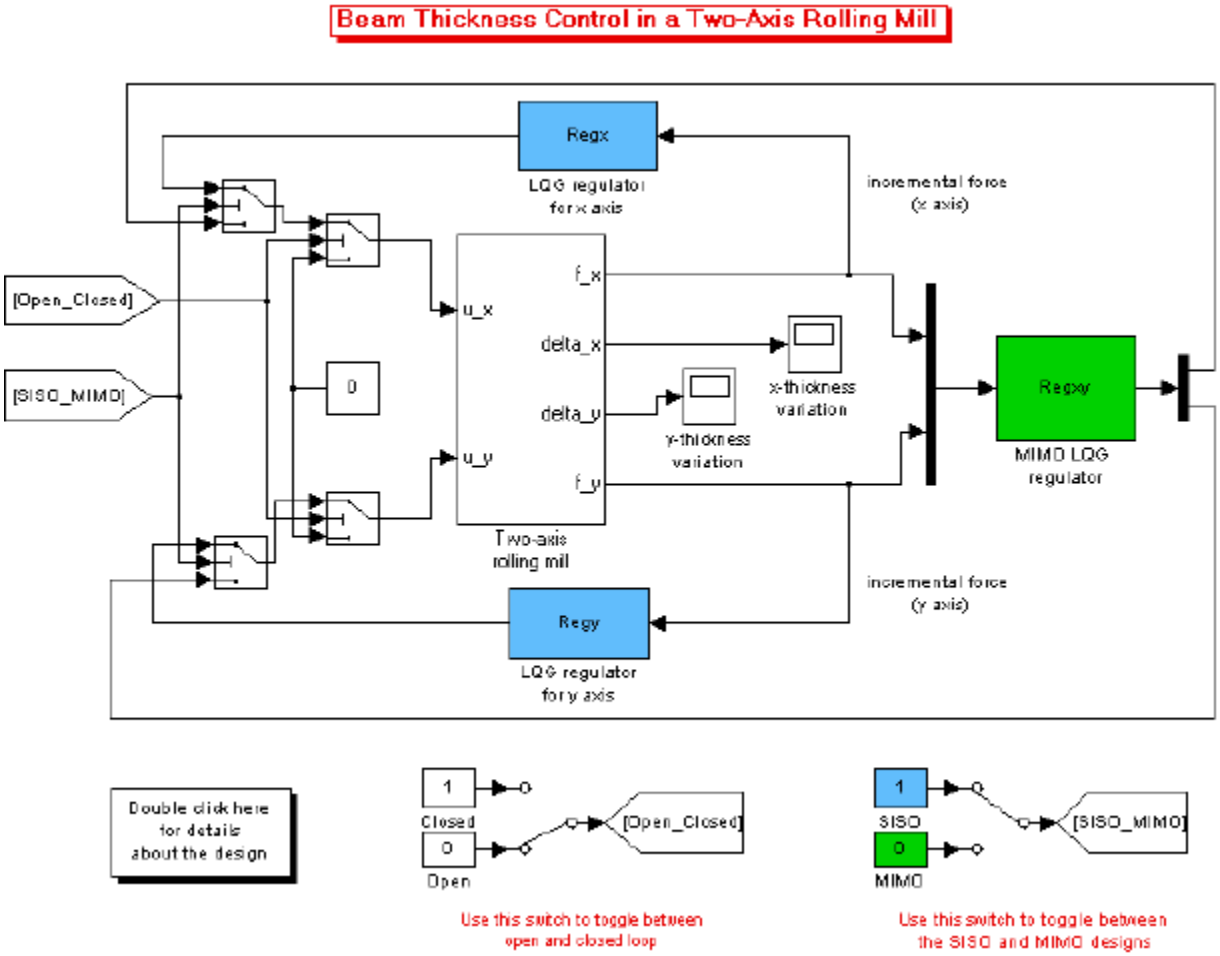


Note how the MIMO regulator does a better job at keeping the gain equally low in all directions.

Simulink® Model

If you are a Simulink® user, click on the link below to open a companion Simulink® model that implements both multi-loop SISO and MIMO control architectures. You can use this model to compare both designs by switching between designs during simulation.

Open Simulink model of two-axis rolling mill.


Beam Thickness Control in a Two-Axis Rolling Mill
Regx
LQG regulator for x axis
incremental force (x axis)
[Open_Closed]
[SISO_MIMO]
0
u_x
u_y
f_x
delta_x
delta_y
f_y
Two-axis rolling mill
x-thickness variation
y-thickness variation
Regxy
MIMO LQG regulator
incremental force (y axis)
Regy
LQG regulator for y axis
Double click here for details about the design
1
Closed
0
Open
[Open_Closed]
Use this switch to toggle between open and closed loop
1
SISO
0
MIMO
[SISO_MIMO]
Use this switch to toggle between the SISO and MIMO designs

Kalman Filtering

This example shows how to perform Kalman filtering. First, you design a steady-state filter using the `kalman` command. Then, you simulate the system to show how it reduces error from measurement noise. This example also shows how to implement a time-varying filter, which can be useful for systems with nonstationary noise sources.

Steady-State Kalman Filter

Consider the following discrete plant with Gaussian noise w on the input and measurement noise v on the output:

$$x[n+1] = Ax[n] + Bu[n] + Gw[n]$$
$$y[n] = Cx[n] + Du[n] + Hw[n] + v[n]$$

The goal is to design a Kalman filter to estimate the true plant output $y_t[n] = y[n] - v[n]$ based on the noisy measurements $y[n]$. This steady-state Kalman filter uses the following equations for this estimation.

Time update:

$$\hat{x}[n+1|n] = A\hat{x}[n|n-1] + Bu[n] + Gw[n]$$

Measurement update:

$$\hat{x}[n|n] = \hat{x}[n|n-1] + M_x(y[n] - C\hat{x}[n|n-1] - Du[n])$$
$$\hat{y}[n|n] = C\hat{x}[n|n-1] + Du[n] + M_y(y[n] - C\hat{x}[n|n-1] - Du[n])$$

Here,

- $\hat{x}[n|n-1]$ is the estimate of $x[n]$, given past measurements up to $y[n-1]$.
- $\hat{x}[n|n]$and $\hat{y}[n|n]$ are the estimated state values and measurement, updated based on the last measurement $y[n]$.
- M_x and M_y are the optimal innovation gains, chosen to minimize the steady-state covariance of the estimation error, given the noise covariances $E\left(w[n]w[n]^T\right) = Q$, $E\left(v[n]v[n]^T\right) = R$, and $N = E\left(w[n]v[n]^T\right) = 0$. (For details about how these gains are chosen, see `kalman`.)

(These update equations describe a `current` type estimator. For information about the difference between `current` estimators and `delayed` estimators, see `kalman`.)

Design the Filter

You can use the `kalman` function to design this steady-state Kalman filter. This function determines the optimal steady-state filter gain M for a particular plant based on the process noise covariance Q and the sensor noise covariance R that you provide. For this example, use the following values for the state-space matrices of the plant.

```
A = [1.1269   -0.4940    0.1129
     1.0000         0         0
          0    1.0000         0];
```

```
B = [-0.3832
      0.5919
      0.5191];

C = [1 0 0];

D = 0;
```

For this example, set $G = B$, meaning that the process noise w is additive input noise. Also, set $H = 0$, meaning that the input noise w has no direct effect on the output y. These assumptions yield a simpler plant model:

$$x[n+1] = Ax[n] + Bu[n] + Bw[n]$$
$$y[n] = Cx[n] + v[n]$$

When $H = 0$, it can be shown that $M_y = CM_x$ (see `kalman`). Together, these assumptions also simplify the update equations for the Kalman filter.

Time update:

$$\hat{x}[n+1|n] = A\hat{x}[n|n-1] + Bu[n] + Bw[n]$$

Measurement update:

$$\hat{x}[n|n] = \hat{x}[n|n-1] + M_x(y[n] - C\hat{x}[n|n-1])$$
$$\hat{y}[n|n] = C\hat{x}[n|n]$$

To design this filter, first create the plant model with an input for `w`. Set the sample time to `-1` to mark the plant as discrete (without a specific sample time).

```
Ts = -1;
sys = ss(A,[B B],C,D,Ts,'InputName',{'u' 'w'},'OutputName','y');  % Plant dynamics and additive
```

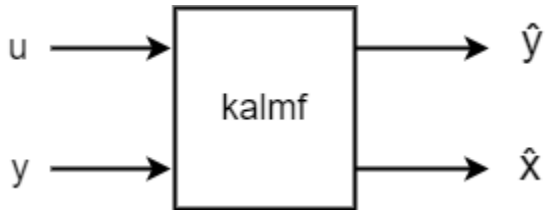
The process noise covariance `Q` and the sensor noise covariance `R` are values greater than zero that you typically obtain from studies or measurements of your system. For this example, specify the following values.

```
Q = 2.3;
R = 1;
```

Use the `kalman` command to design the filter.

```
[kalmf,L,~,Mx,Z] = kalman(sys,Q,R);
```

This command designs the Kalman filter, `kalmf`, a state-space model that implements the time-update and measurement-update equations. The filter inputs are the plant input u and the noisy plant output y. The first output of `kalmf` is the estimate $\hat{y}$ of the true plant output, and the remaining outputs are the state estimates $\hat{x}$.

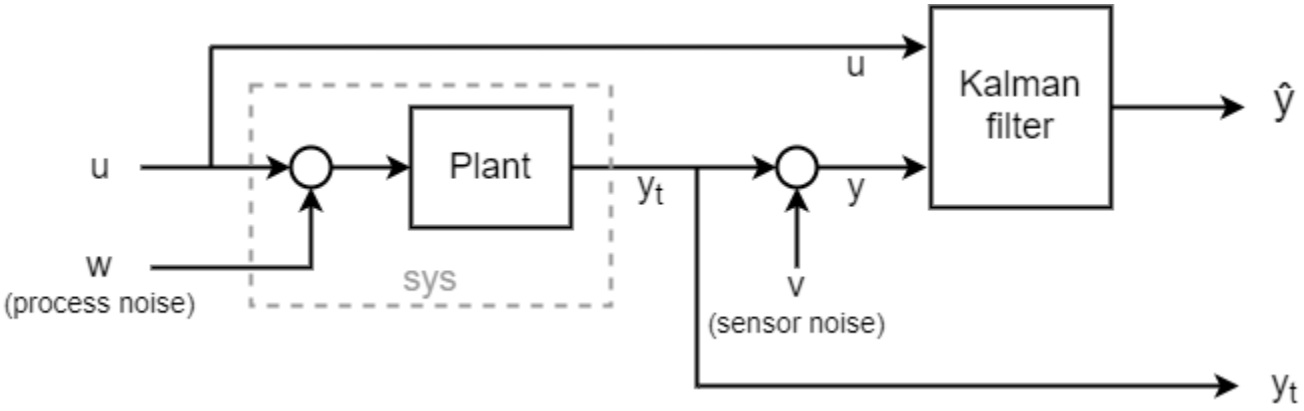


For this example, discard the state estimates and keep only the first output, $\hat{y}$.

```
kalmf = kalmf(1,:);
```

Use the Filter

To see how this filter works, generate some data and compare the filtered response with the true plant response. The complete system is shown in the following diagram.



To simulate this system, use a `sumblk` to create an input for the measurement noise `v`. Then, use `connect` to join `sys` and the Kalman filter together such that `u` is a shared input and the noisy plant output `y` feeds into the other filter input. The result is a simulation model with inputs `w`, `v`, and `u` and outputs `yt` (true response) and `ye` (the filtered or estimated response $\hat{y}$). The signals `yt` and `ye` are the outputs of the plant and the filter, respectively.

```
sys.InputName = {'u','w'};
sys.OutputName = {'yt'};
vIn = sumblk('y=yt+v');

kalmf.InputName = {'u','y'};
kalmf.OutputName = 'ye';

SimModel = connect(sys,vIn,kalmf,{'u','w','v'},{'yt','ye'});
```

To simulate the filter behavior, generate a known sinusoidal input vector.

```
t = (0:100)';
u = sin(t/5);
```

Generate process noise and sensor noise vectors using the same noise covariance values `Q` and `R` that you used to design the filter.

```
rng(10,'twister');
w = sqrt(Q)*randn(length(t),1);
v = sqrt(R)*randn(length(t),1);
```

Finally, simulate the response using `lsim`.

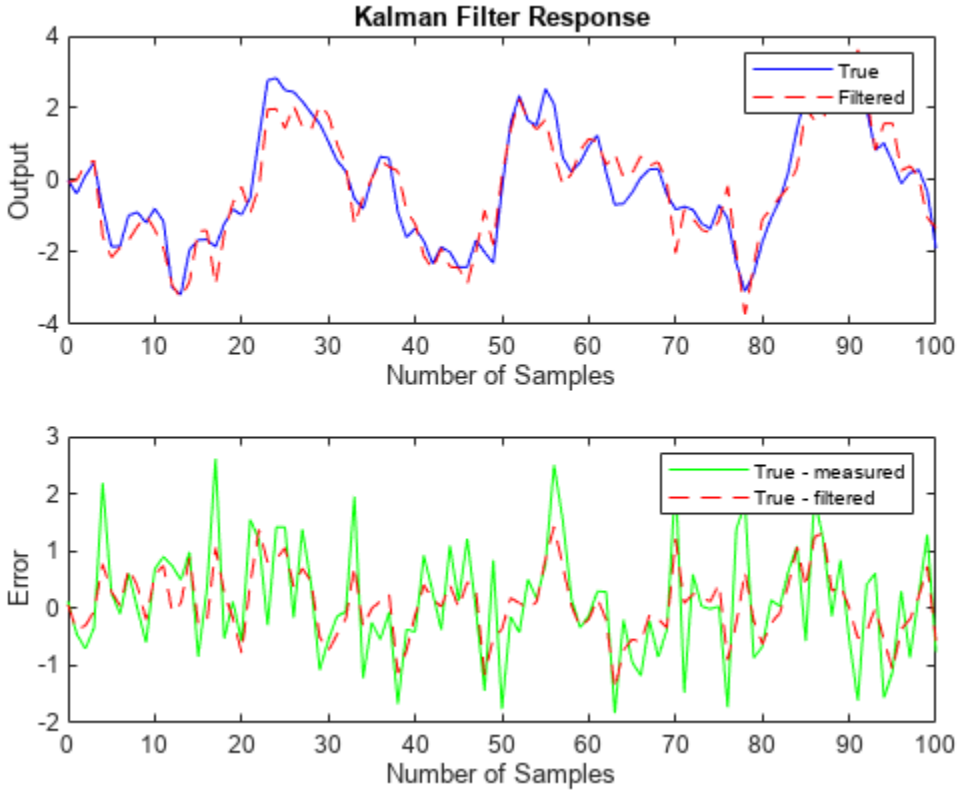
```
out = lsim(SimModel,[u,w,v]);
```

`lsim` generates the response at the outputs `yt` and ye to the inputs applied at `w`, `v`, and `u`. Extract the `yt` and ye channels and compute the measured response.

```
yt = out(:,1);   % true response
ye = out(:,2);  % filtered response
y = yt + v;     % measured response
```

Compare the true response with the filtered response.

```
clf
subplot(211), plot(t,yt,'b',t,ye,'r--'),
xlabel('Number of Samples'), ylabel('Output')
title('Kalman Filter Response')
legend('True','Filtered')
subplot(212), plot(t,yt-y,'g',t,yt-ye,'r--'),
xlabel('Number of Samples'), ylabel('Error')
legend('True - measured','True - filtered')
```



As the second plot shows, the Kalman filter reduces the error `yt - y` due to measurement noise. To confirm this reduction, compute the covariance of the error before filtering (measurement error covariance) and after filtering (estimation error covariance).

```
MeasErr = yt - y;
MeasErrCov = sum(MeasErr.*MeasErr)/length(MeasErr)
```

```
MeasErrCov = 0.9871
```

```
EstErr = yt - ye;
EstErrCov = sum(EstErr.*EstErr)/length(EstErr)
```

```
EstErrCov = 0.3479
```

Time-Varying Kalman Filter Design

The previous design assumed that the noise covariances do not change over time. A time-varying Kalman filter can perform well even when the noise covariance is not stationary.

The time-varying Kalman filter has the following update equations. In the time-varying filter, both the error covariance $P[n]$ and the innovation gain $M_x[n]$ can vary with time. You can modify the time and measurement update equations to account for time variation as follows. (See `kalman` for more detail on these expressions.)

Time update:

$$\hat{x}[n+1|n] = A\hat{x}[n|n] + Bu[n] + Bw[n]$$
$$P[n+1|n] = AP[n|n]A^T + BQB^T$$

Measurement update:

$$\hat{x}[n|n] = \hat{x}[n|n-1] + M_x[n](y[n] - C\hat{x}[n|n-1])$$
$$M_x[n] = P[n|n-1]C^T\left(CP[n|n-1]C^T + R[n]\right)^{-1}$$
$$P[n|n] = (I - M_x[n]C)P[n|n-1]$$
$$\hat{y}[n|n] = C\hat{x}[n|n]$$

You can implement a time-varying Kalman filter in Simulink® using the Kalman Filter block. For an example demonstrating the use of that block, see “State Estimation Using Time-Varying Kalman Filter” on page 12-273. For this example, implement the time-varying filter in MATLAB®.

To create the time-varying Kalman filter, first, generate the noisy plant response. Simulate the plant response to the input signal `u` and process noise `w` defined previously. Then, add the measurement noise `v` to the simulated true response `yt` to obtain the noisy response `y`. In this example, the covariances of the noise vectors `w` and `v` do not change with time. However, you can use the same procedure for nonstationary noise.

```
yt = lsim(sys,[u w]);
y = yt + v;
```

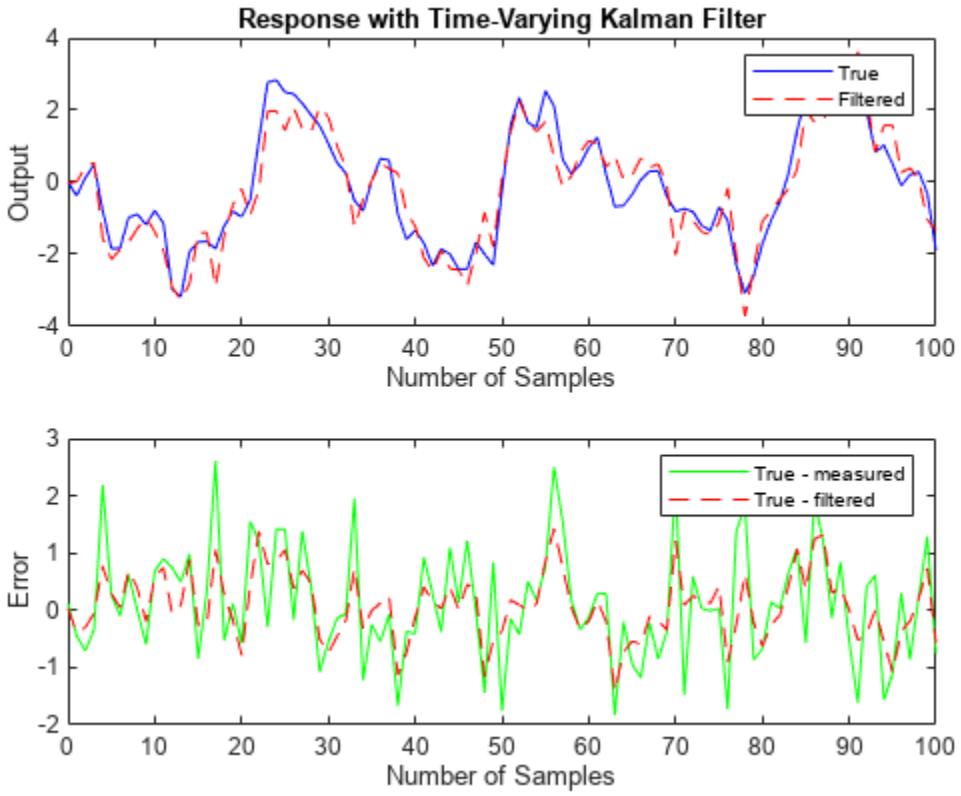
Next, implement the recursive filter update equations in a `for` loop.

```
P = B*Q*B';         % Initial error covariance
x = zeros(3,1);     % Initial condition on the state

ye = zeros(length(t),1);
ycov = zeros(length(t),1);
errcov = zeros(length(t),1);

for i=1:length(t)
  % Measurement update
  Mxn = P*C'/(C*P*C'+R);
  x = x + Mxn*(y(i)-C*x);   % x[n|n]
  P = (eye(3)-Mxn*C)*P;     % P[n|n]

  ye(i) = C*x;
  errcov(i) = C*P*C';
```

```
  % Time update
  x = A*x + B*u(i);         % x[n+1|n]
  P = A*P*A' + B*Q*B';      % P[n+1|n]
end
```

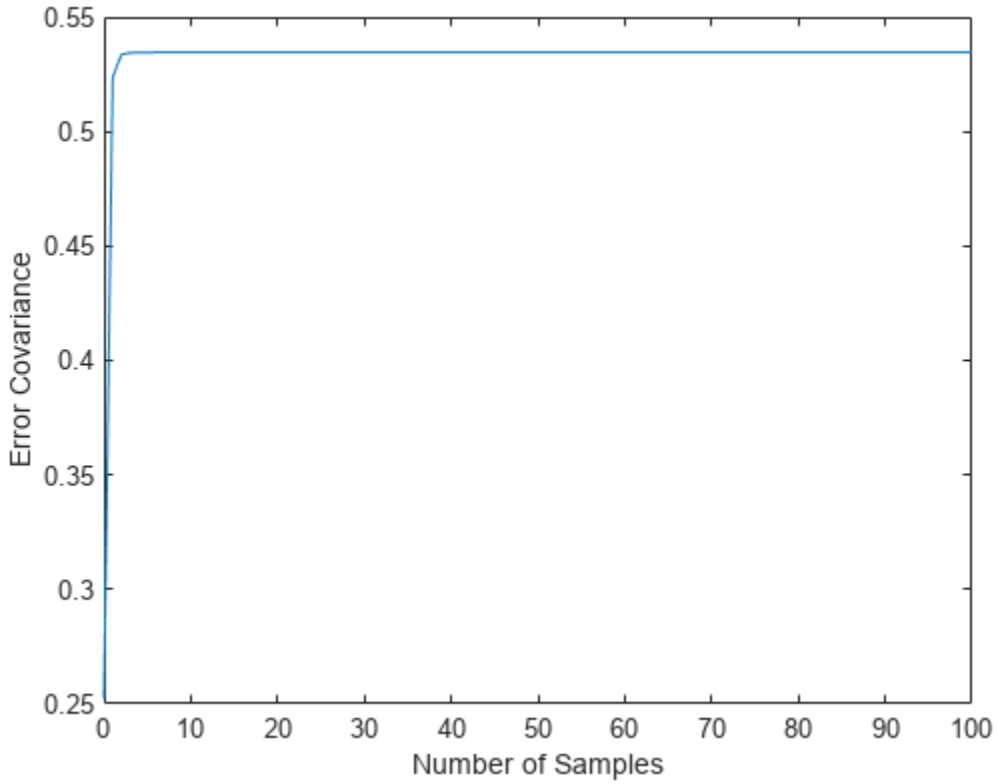
Compare the true response with the filtered response.

```
subplot(211), plot(t,yt,'b',t,ye,'r--')
xlabel('Number of Samples'), ylabel('Output')
title('Response with Time-Varying Kalman Filter')
legend('True','Filtered')
subplot(212), plot(t,yt-y,'g',t,yt-ye,'r--'),
xlabel('Number of Samples'), ylabel('Error')
legend('True - measured','True - filtered')
```



The time-varying filter also estimates the output covariance during the estimation. Because this example uses stationary input noise, the output covariance tends to a steady-state value. Plot the output covariance to confirm that the filter has reached a steady state.

```
figure
plot(t,errcov)
xlabel('Number of Samples'), ylabel('Error Covariance'),
```



From the covariance plot, you can see that the output covariance reaches a steady state in about five samples. From then on, the time-varying filter has the same performance as the steady-state version.

As in the steady-state case, the filter reduces the error due to measurement noise. To confirm this reduction, compute the covariance of the error before filtering (measurement error covariance) and after filtering (estimation error covariance).

```
MeasErr = yt - y;
MeasErrCov = sum(MeasErr.*MeasErr)/length(MeasErr)
```

```
MeasErrCov = 0.9871
```

```
EstErr = yt - ye;
EstErrCov = sum(EstErr.*EstErr)/length(EstErr)
```

```
EstErrCov = 0.3479
```

Finally, when the time-varying filter reaches steady state, the values in the gain matrix `Mxn` match those computed by `kalman` for the steady-state filter.

```
Mx,Mxn
```

```
Mx = 3×1

    0.5345
    0.0101
   -0.4776
```

```
Mxn = 3×1

    0.5345
    0.0101
   -0.4776
```

See Also

`kalman`

Related Examples

- “State Estimation Using Time-Varying Kalman Filter” on page 12-273
- “LQG Regulation: Rolling Mill Case Study” on page 23-14

External Websites

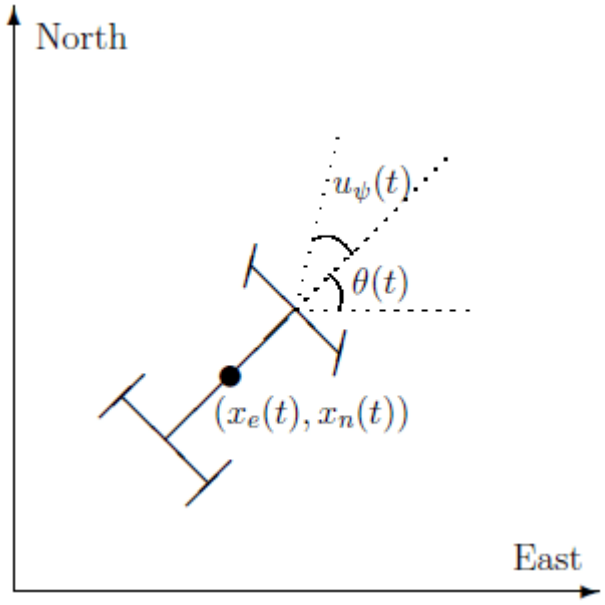
- Understanding Kalman Filters — MATLAB Video Series

State Estimation Using Time-Varying Kalman Filter

This example shows how to estimate states of linear systems using time-varying Kalman filters in Simulink®. You use the Kalman Filter block from the Control System Toolbox™ library to estimate the position and velocity of a ground vehicle based on noisy position measurements such as GPS sensor measurements. The plant model in Kalman filter has time-varying noise characteristics.

Introduction

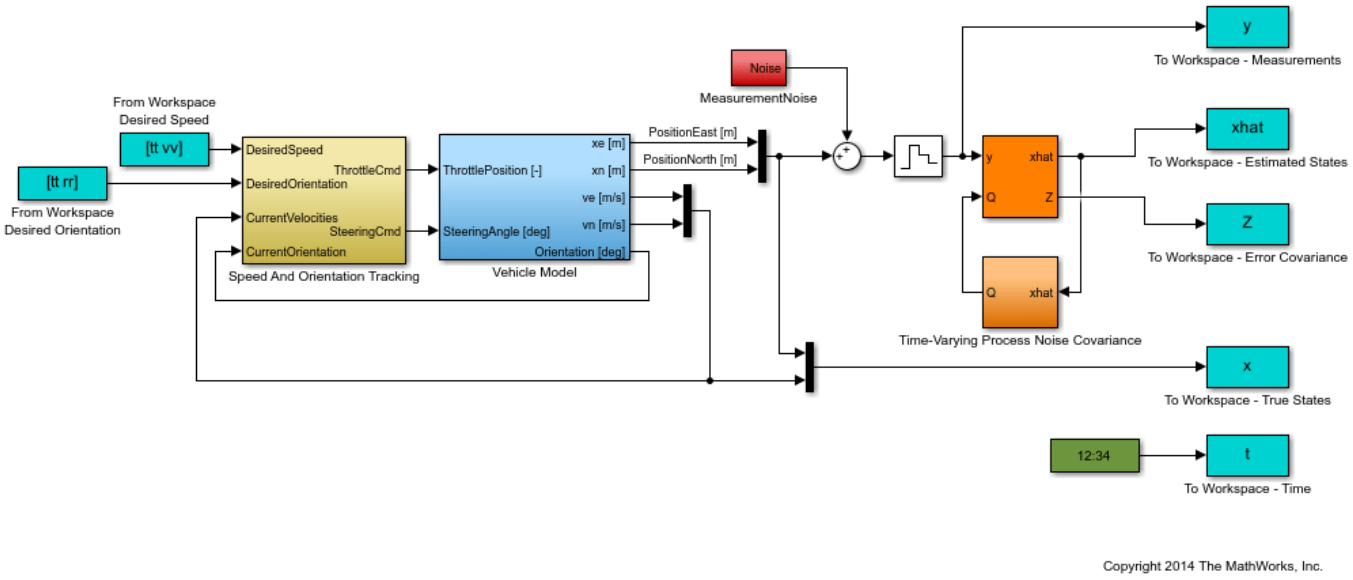
You want to estimate the position and velocity of a ground vehicle in the north and east directions. The vehicle can move freely in the two-dimensional space without any constraints. You design a multi-purpose navigation and tracking system that can be used for any object and not just a vehicle.



$x_e(t)$ and $x_n(t)$ are the vehicle's east and north positions from the origin, $\theta(t)$ is the vehicle orientation from east and $u_\psi(t)$ is the steering angle of the vehicle. t is the continuous-time variable.

The Simulink model consists of two main parts: Vehicle model and the Kalman filter. These are explained further in the following sections.

```
open_system('ctrlKalmanNavigationExample');
```



Vehicle Model

The tracked vehicle is represented with a simple point-mass model:

$$\frac{d}{dt}\begin{bmatrix} x_e(t) \\ x_n(t) \\ s(t) \\ \theta(t) \end{bmatrix} = \begin{bmatrix} s(t)\cos(\theta(t)) \\ s(t)\sin(\theta(t)) \\ (P\,\frac{u_T(t)}{s(t)} - A\,C_d\,s(t)^2)/m \\ s(t)\tan(u_\psi(t))/L \end{bmatrix}$$

where the vehicle states are:

$x_e(t)$	East position $[m]$
$x_n(t)$	North position $[m]$
$s(t)$	Speed $[m/s]$
$\theta(t)$	Orientation from east $[deg]$

the vehicle parameters are:

$P = 100000$	Peak engine power $[W]$
$A = 1$	Frontal area $[m^2]$
$C_d = 0.3$	Drag coefficient $[Unitless]$
$m = 1250$	Vehicle mass $[kg]$
$L = 2.5$	Wheelbase length $[m]$

and the control inputs are:

$u_T(t)$	Throttle position in the range of -1 and 1 $[Unitless]$
$u_\psi(t)$	Steering angle $[deg]$

The longitudinal dynamics of the model ignore tire rolling resistance. The lateral dynamics of the model assume that the desired steering angle can be achieved instantaneously and ignore the yaw moment of inertia.

The car model is implemented in the `ctrlKalmanNavigationExample/Vehicle Model` subsystem. The Simulink model contains two PI controllers for tracking the desired orientation and speed for the car in the `ctrlKalmanNavigationExample/Speed And Orientation Tracking` subsystem. This allows you to specify various operating conditions for the car and test the Kalman filter performance.

Kalman Filter Design

Kalman filter is an algorithm to estimate unknown variables of interest based on a linear model. This linear model describes the evolution of the estimated variables over time in response to model initial conditions as well as known and unknown model inputs. In this example, you estimate the following parameters/variables:

$$\hat{x}[n] = \begin{bmatrix} \hat{x}_e[n] \\ \hat{x}_n[n] \\ \hat{\dot{x}}_e[n] \\ \hat{\dot{x}}_n[n] \end{bmatrix}$$

where

$\hat{x}_e[n]$ East position estimate $[m]$
$\hat{x}_n[n]$ North position estimate $[m]$
$\hat{\dot{x}}_e[n]$ East velocity estimate $[m/s]$
$\hat{\dot{x}}_n[n]$ North velocity estimate $[m/s]$

The $\dot{x}$ terms denote velocities and not the derivative operator. n is the discrete-time index. The model used in the Kalman filter is of the form:

$$\begin{aligned} \hat{x}[n+1] &= A\hat{x}[n] + Gw[n] \\ y[n] &= C\hat{x}[n] + v[n] \end{aligned}$$

where $\hat{x}$ is the state vector, y is the measurements, w is the process noise, and v is the measurement noise. Kalman filter assumes that w and v are zero-mean, independent random variables with known variances $E[ww^T] = Q$, $E[vv^T] = R$, and $E[wv^T] = N$. Here, the A, G, and C matrices are:

$$A = \begin{bmatrix} 1 & 0 & T_s & 0 \\ 0 & 1 & 0 & T_s \\ 0 & 0 & 1 & 0 \\ 0 & 0 & 0 & 1 \end{bmatrix}$$

$$G = \begin{bmatrix} T_s/2 & 0 \\ 0 & T_s/2 \\ 1 & 0 \\ 0 & 1 \end{bmatrix}$$

$$C = \begin{bmatrix} 1 & 0 & 0 & 0 \\ 0 & 1 & 0 & 0 \end{bmatrix}$$

where $T_s = 1\ [s]$

The third row of A and G model the east velocity as a random walk: $\hat{\dot{x}}_e[n+1] = \hat{\dot{x}}_e[n] + w_1[n]$. In reality, position is a continuous-time variable and is the integral of velocity over time $\frac{d}{dt}\hat{x}_e = \hat{\dot{x}}_e$. The first row of the A and G represent a discrete approximation to this kinematic relationship: $(\hat{x}_e[n+1] - \hat{x}_e[n])/Ts = (\hat{\dot{x}}_e[n+1] + \hat{\dot{x}}_e[n])/2$. The second and fourth rows of the A and G represent the same relationship between the north velocity and position.

The C matrix represents that only position measurements are available. A position sensor, such as GPS, provides these measurements at the sample rate of 1Hz. The variance of the measurement noise v, the R matrix, is specified as $R = 50$. Since R is specified as a scalar, the Kalman filter block assumes that the matrix R is diagonal, its diagonals are 50 and is of compatible dimensions with y. If the measurement noise is Gaussian, R=50 corresponds to 68% of the position measurements being within $\pm\sqrt{50}\ m$ or the actual position in the east and north directions. However, this assumption is not necessary for the Kalman filter.

The elements of w capture how much the vehicle velocity can change over one sample time Ts. The variance of the process noise w, the Q matrix, is chosen to be time-varying. It captures the intuition that typical values of $w[n]$ are smaller when velocity is large. For instance, going from 0 to 10m/s is easier than going from 10 to 20m/s. Concretely, you use the estimated north and east velocities and a saturation function to construct Q[n]:

$$f_{sat}(z) = min(max(z, 25), 625)$$

$$Q[n] = \begin{bmatrix} 1 + \dfrac{250}{f_{sat}(\hat{\dot{x}}_e^2)} & 0 \\ 0 & 1 + \dfrac{250}{f_{sat}(\hat{\dot{x}}_n^2)} \end{bmatrix}$$

The diagonals of Q model the variance of w inversely proportional to the square of the estimated velocities. The saturation function prevents Q from becoming too large or small. The coefficient 250 is obtained from a least squares fit to 0-5, 5-10, 10-15, 15-20, 20-25m/s acceleration time data for a generic vehicle. Note that the diagonal Q choice represents a naive assumption that the velocity changes in the north and east direction are uncorrelated.

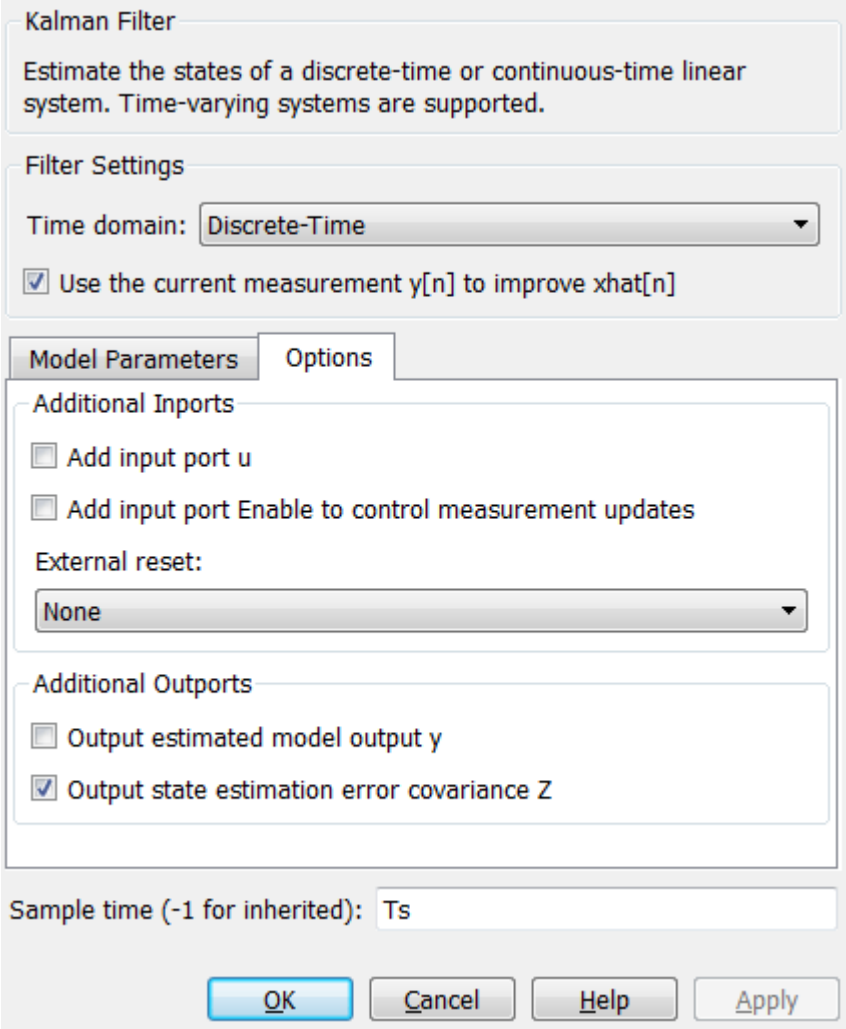
Kalman Filter Block Inputs and Setup

The 'Kalman Filter' block is in the `Control System Toolbox` library in Simulink. It is also in `System Identification Toolbox/Estimators` library. Configure the block parameters for discrete-time state estimation. Specify the following **Filter Settings** parameters:

- **Time domain:** Discrete-time. Choose this option to estimate discrete-time states.
- Select the **Use current measurement y[n] to improve xhat[n]** check box. This implements the "current estimator" variant of the discrete-time Kalman filter. This option improves the estimation accuracy and is more useful for slow sample times. However, it increases the computational cost. In addition, this Kalman filter variant has direct feedthrough, which leads to an algebraic loop if the Kalman filter is used in a feedback loop that does not contain any delays (the feedback loop itself also has direct feedthrough). The algebraic loop can further impact the simulation speed.

Click the **Options** tab to set the block inport and outport options:

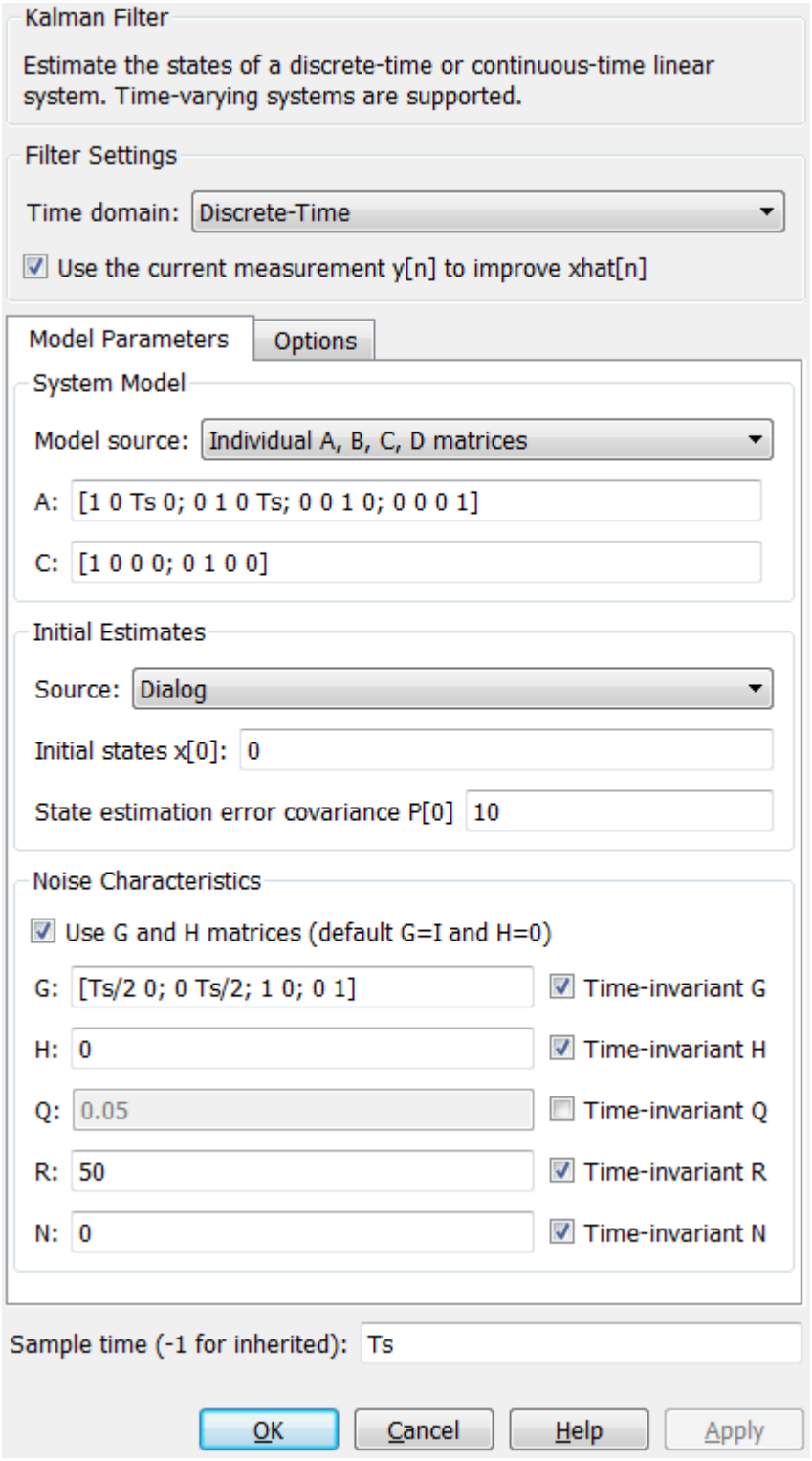
- Unselect the **Add input port u** check box. There are no known inputs in the plant model.
- Select the **Output state estimation error covariance Z** check box. The Z matrix provides information about the filter's confidence in the state estimates.



Click **Model Parameters** to specify the plant model and noise characteristics:

- **Model source:** Individual A, B, C, D matrices.
- **A**: A. The A matrix is defined earlier in this example.
- **C**: C. The C matrix is defined earlier in this example.
- **Initial Estimate Source**: Dialog
- **Initial states x[0]**: `0`. This represents an initial guess of 0 for the position and velocity estimates at t=0s.
- **State estimation error covariance P[0]**: `10`. Assume that the error between your initial guess x[0] and its actual value is a random variable with a standard deviation $\sqrt{10}$.

- Select the **Use G and H matrices (default G=I and H=0)** check box to specify a non-default G matrix.
- **G**: G. The G matrix is defined earlier in this example.
- **H**: `0`. The process noise does not impact the measurements y entering the Kalman filter block.
- Unselect the **Time-invariant Q** check box. The Q matrix is time-varying and is supplied through the block inport Q. The block uses a time-varying Kalman filter due to this setting. You can select this option to use a time-invariant Kalman filter. A time-invariant Kalman filter performs slightly worse for this problem, but is easier to design and has a lower computational cost.
- **R**: R. This is the covariance of the measurement noise $v[n]$. The R matrix is defined earlier in this example.
- **N**: `0`. Assume that there is no correlation between process and measurement noises.
- **Sample time (-1 for inherited)**: Ts, which is defined earlier in this example.


Kalman Filter
Estimate the states of a discrete-time or continuous-time linear system. Time-varying systems are supported.
Filter Settings
Time domain: Discrete-Time
Use the current measurement y[n] to improve xhat[n]
Model Parameters
Options
System Model
Model source: Individual A, B, C, D matrices
A: [1 0 Ts 0; 0 1 0 Ts; 0 0 1 0; 0 0 0 1]
C: [1 0 0 0; 0 1 0 0]
Initial Estimates
Source: Dialog
Initial states x[0]: 0
State estimation error covariance P[0] 10
Noise Characteristics
Use G and H matrices (default G=I and H=0)
G: [Ts/2 0; 0 Ts/2; 1 0; 0 1]
Time-invariant G
H: 0
Time-invariant H
Q: 0.05
Time-invariant Q
R: 50
Time-invariant R
N: 0
Time-invariant N
Sample time (-1 for inherited): Ts
OK
Cancel
Help
Apply

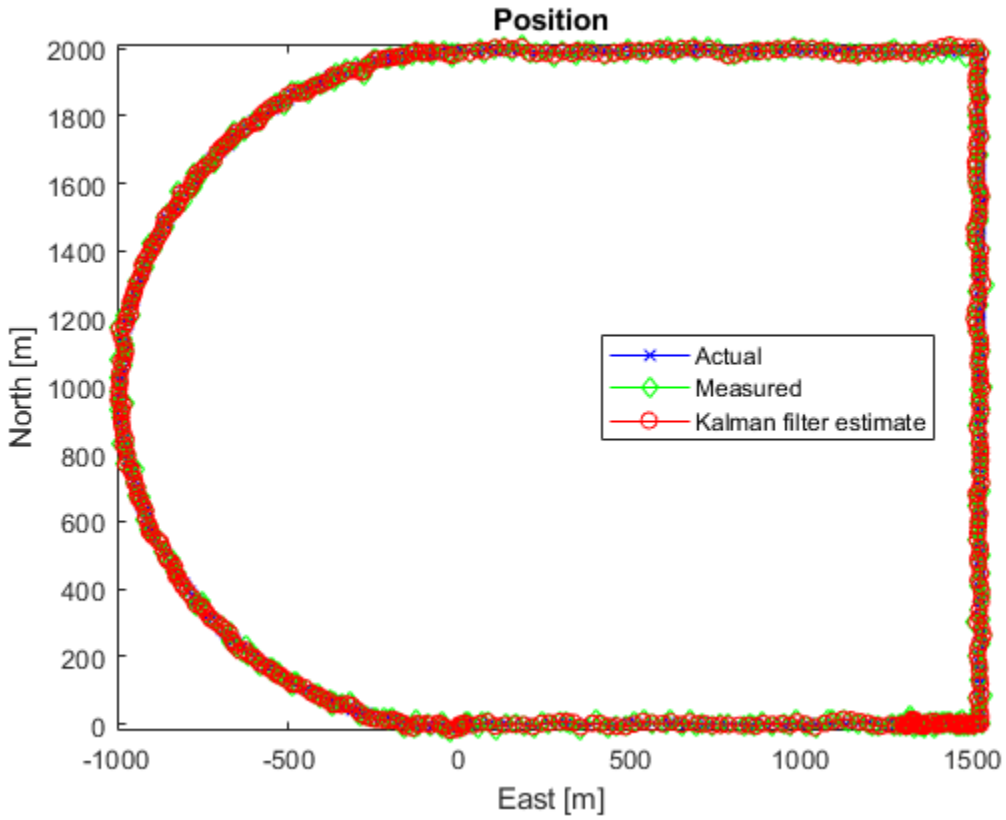
Results

Test the performance of the Kalman filter by simulating a scenario where the vehicle makes the following maneuvers:

- At t = 0 the vehicle is at $x_e(0) = 0$, $x_n(0) = 0$ and is stationary.
- Heading east, it accelerates to 25m/s. It decelerates to 5m/s at t=50s.
- At t = 100s, it turns toward north and accelerates to 20m/s.
- At t = 200s, it makes another turn toward west. It accelerates to 25m/s.
- At t = 260s, it decelerates to 15m/s and makes a constant speed 180 degree turn.

Simulate the Simulink model. Plot the actual, measured and Kalman filter estimates of vehicle position.

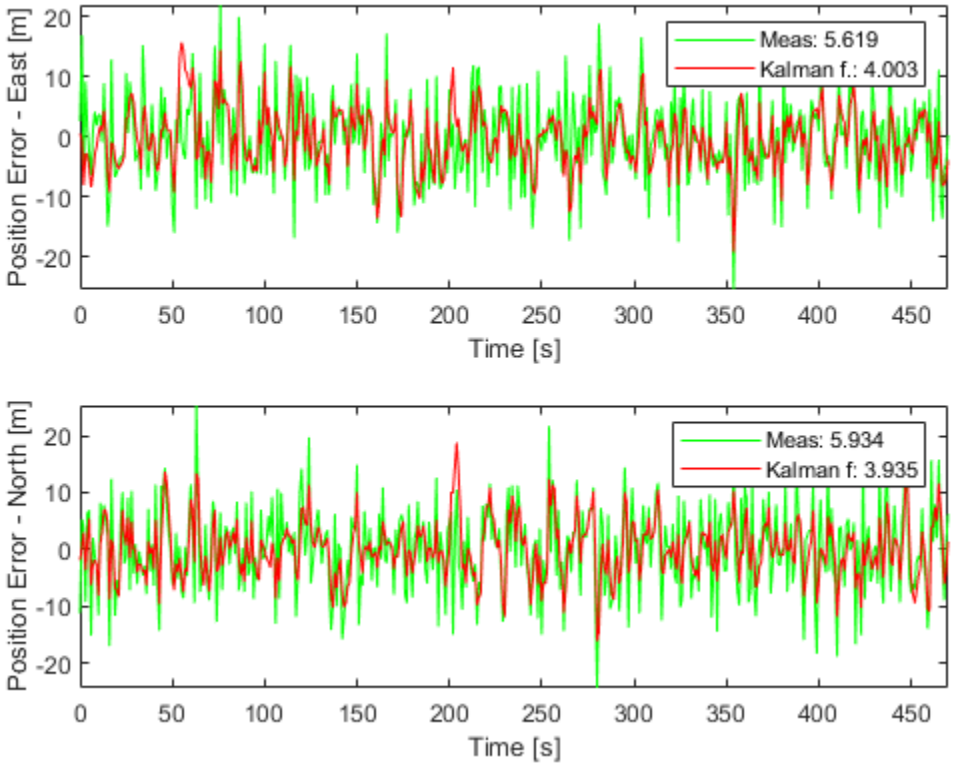
```
sim('ctrlKalmanNavigationExample');

figure;
% Plot results and connect data points with a solid line.
plot(x(:,1),x(:,2),'bx',...
    y(:,1),y(:,2),'gd',...
    xhat(:,1),xhat(:,2),'ro',...
    'LineStyle','-');
title('Position');
xlabel('East [m]');
ylabel('North [m]');
legend('Actual','Measured','Kalman filter estimate','Location','Best');
axis tight;
```



The error between the measured and actual position as well as the error between the Kalman filter estimate and actual position is:

```
% East position measurement error [m]
n_xe = y(:,1)-x(:,1);
% North position measurement error [m]
n_xn = y(:,2)-x(:,2);
% Kalman filter east position error [m]
e_xe = xhat(:,1)-x(:,1);
% Kalman filter north position error [m]
e_xn = xhat(:,2)-x(:,2);

figure;
% East Position Errors
subplot(2,1,1);
plot(t,n_xe,'g',t,e_xe,'r');
ylabel('Position Error - East [m]');
xlabel('Time [s]');
legend(sprintf('Meas: %.3f',norm(n_xe,1)/numel(n_xe)),...
    sprintf('Kalman f.: %.3f',norm(e_xe,1)/numel(e_xe)));
axis tight;
% North Position Errors
subplot(2,1,2);
plot(t,y(:,2)-x(:,2),'g',t,xhat(:,2)-x(:,2),'r');
ylabel('Position Error - North [m]');
xlabel('Time [s]');
legend(sprintf('Meas: %.3f',norm(n_xn,1)/numel(n_xn)),...
```

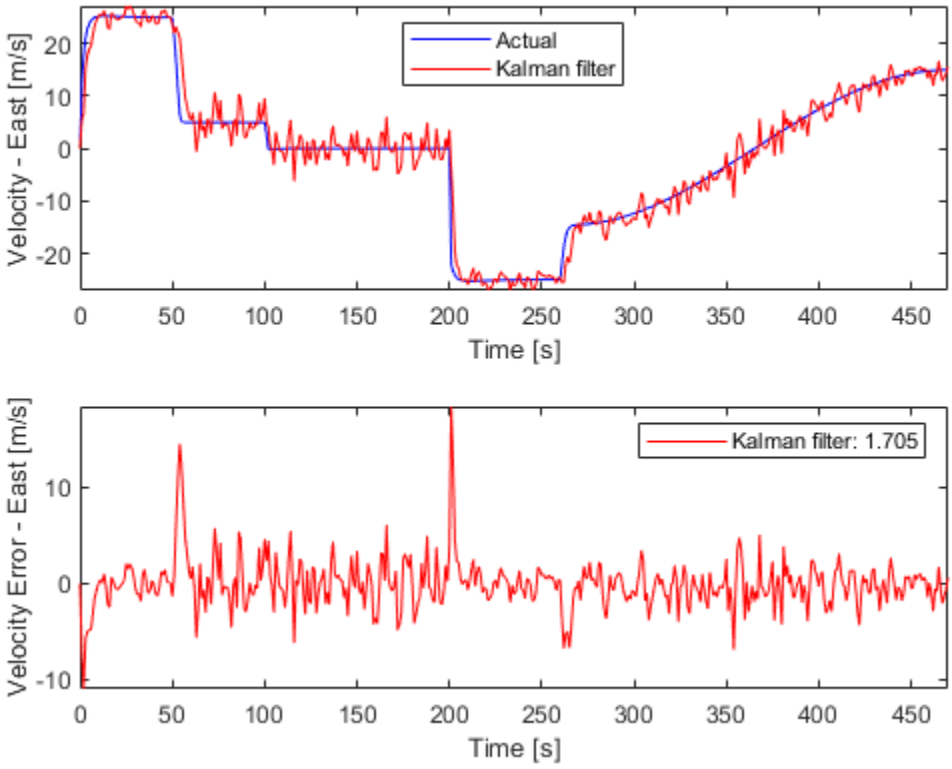
```
    sprintf('Kalman f: %.3f',norm(e_xn,1)/numel(e_xn)));
axis tight;
```



The plot legends show the position measurement and estimation error ($||x_e - \hat{x}_e||_1$ and $||x_n - \hat{x}_n||_1$) normalized by the number of data points. The Kalman filter estimates have about 25% percent less error than the raw measurements.

The actual velocity in the east direction and its Kalman filter estimate is shown below in the top plot. The bottom plot shows the estimation error.

```
e_ve = xhat(:,3)-x(:,3); % [m/s] Kalman filter east velocity error
e_vn = xhat(:,4)-x(:,4); % [m/s] Kalman filter north velocity error
figure;
% Velocity in east direction and its estimate
subplot(2,1,1);
plot(t,x(:,3),'b',t,xhat(:,3),'r');
ylabel('Velocity - East [m/s]');
xlabel('Time [s]');
legend('Actual','Kalman filter','Location','Best');
axis tight;
subplot(2,1,2);
% Estimation error
plot(t,e_ve,'r');
ylabel('Velocity Error - East [m/s]');
xlabel('Time [s]');
legend(sprintf('Kalman filter: %.3f',norm(e_ve,1)/numel(e_ve)));
axis tight;
```



The legend on the error plot shows the east velocity estimation error $||\dot{x}_e - \hat{\dot{x}}_e||_1$ normalized by the number of data points.

The Kalman filter velocity estimates track the actual velocity trends correctly. The noise levels decrease when the vehicle is traveling at high velocities. This is in line with the design of the Q matrix. The large two spikes are at t=50s and t=200s. These are the times when the car goes through sudden deceleration and a sharp turn, respectively. The velocity changes at those instants are much larger than the predictions from the Kalman filter, which is based on its Q matrix input. After a few time-steps, the filter estimates catch up with the actual velocity.

Summary

You estimated the position and velocity of a vehicle using the Kalman filter block in Simulink. The process noise dynamics of the model were time-varying. You validated the filter performance by simulating various vehicle maneuvers and randomly generated measurement noise. The Kalman filter improved the position measurements and provided velocity estimates for the vehicle.

```
bdclose('ctrlKalmanNavigationExample');
```

See Also

Kalman Filter

Related Examples

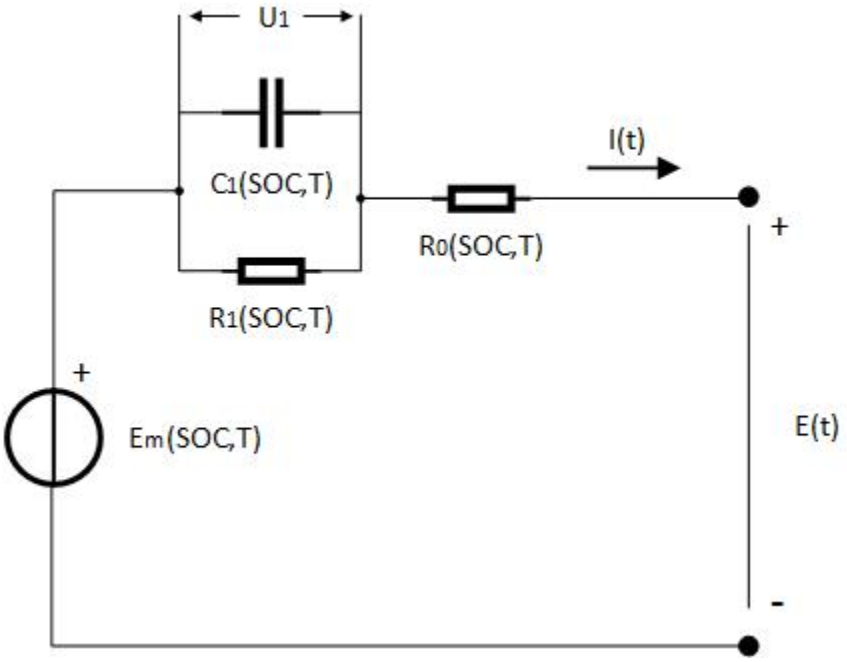
- “Kalman Filtering” on page 12-265
- “Nonlinear State Estimation Using Unscented Kalman Filter and Particle Filter” on page 13-18

Nonlinear State Estimation of a Degrading Battery System

This example shows how to estimate the states of a nonlinear system using an Unscented Kalman Filter in Simulink®. The example also illustrates how to develop an event-based Kalman Filter to update system parameters for more accurate state estimation. This example also requires Simscape™ and Stateflow®.

Overview

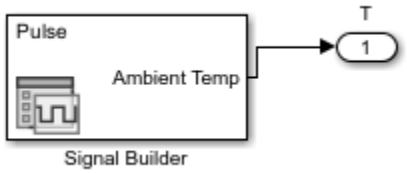
Consider a battery model with the following equivalent circuit [1]



The model consists of a voltage source E_m, a series resistor R_0 and a single RC block with components R_1 and C_1. The battery alternates between charging and discharging cycles. In this example, you estimate the state of charge (SOC) of the battery model using measured currents, voltages, and temperature of the battery. You assume the battery is a nonlinear system, and estimate the SOC using an Unscented Kalman Filter. The capacity of the battery degrades with every discharge-charge cycle, giving an inaccurate SOC estimation. You use an event-based linear Kalman filter to estimate the battery capacity when the battery transitions between charging and discharging. You then use the estimated capacity to indicate the health condition of the battery.

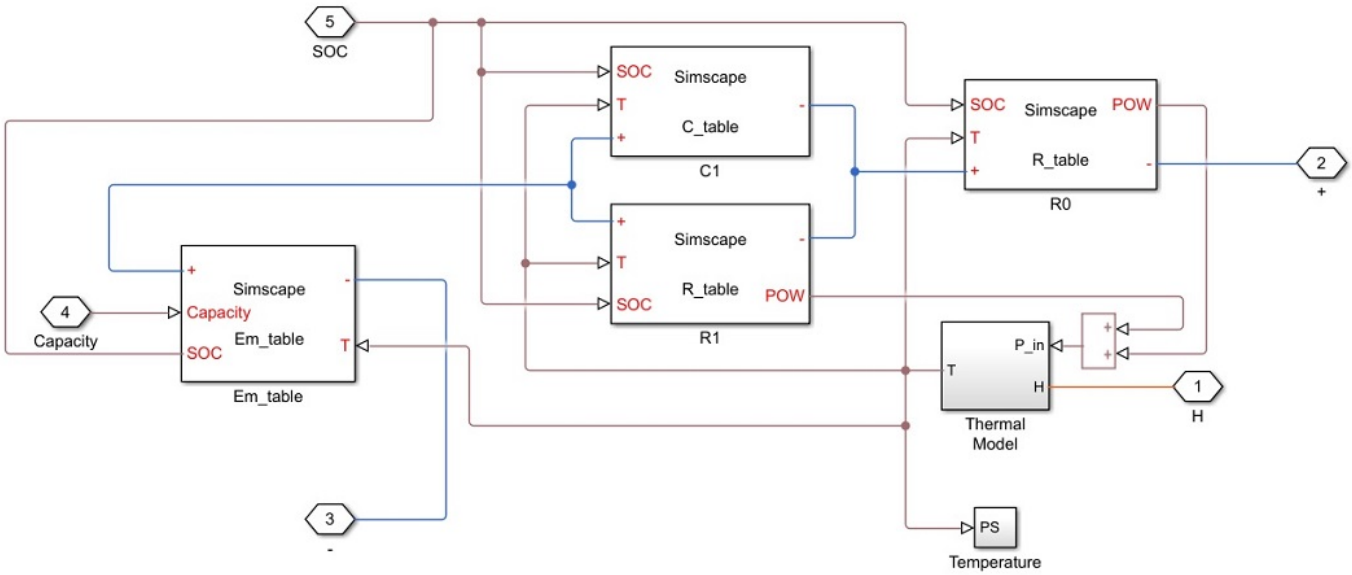
The Simulink model contains three major components: a battery model, an Unscented Kalman Filter block and an event-based Kalman Filter block.

```
open_system('BatteryExampleUKF/')
```



Battery Model

The battery model with thermal effect is implemented using Simscape software.

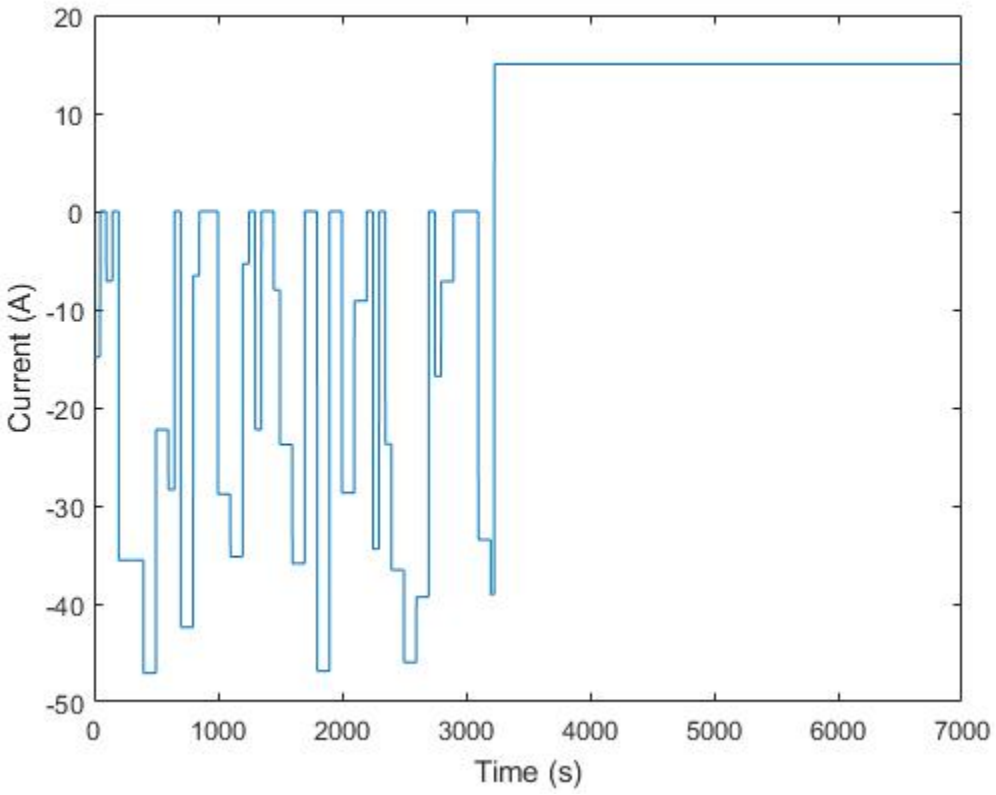


The state transition equations for the battery model are given by:

$$\frac{d}{dt}\begin{pmatrix} SOC \\ U_1 \end{pmatrix} = \begin{pmatrix} 0 \\ -\frac{1}{R_1(SOC,T_b)*C_1(SOC,T_b)}U_1 \end{pmatrix} + \begin{pmatrix} -\frac{1}{3600*C_q} \\ \frac{1}{C_1(SOC,T_b)} \end{pmatrix} I + W$$

where $R_1(SOC,T_b)$ and $C_1(SOC,T_b)$ are the thermal and SOC-dependent resistor and capacitor in the RC block, U_1 is the voltage across capacitor C_1, I is the input current, T_b is the battery temperature, C_q is the battery capacity (unit: Ah), and W is the process noise.

The input currents are randomly generated pulses when the battery is discharging and constant when the battery is charging.



The measurement equation is given by:

$$E = E_m(SOC, T_b) - U_1 - IR_0(SOC, T_b) + V$$

where E is the measured output voltage, $R_0(SOC, T_b)$ is the serial resistor, $E_m = E_m(SOC, T_b)$ is the electromotive force from voltage source, and V is the measurement noise.

In the model, R_0, R_1, C_1 and E_m are 2D look-up tables that are dependent on SOC and battery temperature. The parameters in the look-up tables are identified using experimental data [1].

Estimate State of Charge (SOC)

To use the Unscented Kalman Filter block, you specify the measurement and state transition functions using either MATLAB® or Simulink functions. This example demonstrates the use of Simulink functions. Since Unscented Kalman Filters are discrete-time filters, first discretize the state equations. In this example, Euler discretization is employed. Let the sampling time be Ts. For a general nonlinear system $\dot{x} = f(x, u)$, the system can be discretized as:

$$x_{T+1} = x_T + f(x_T, u_T) * Ts$$

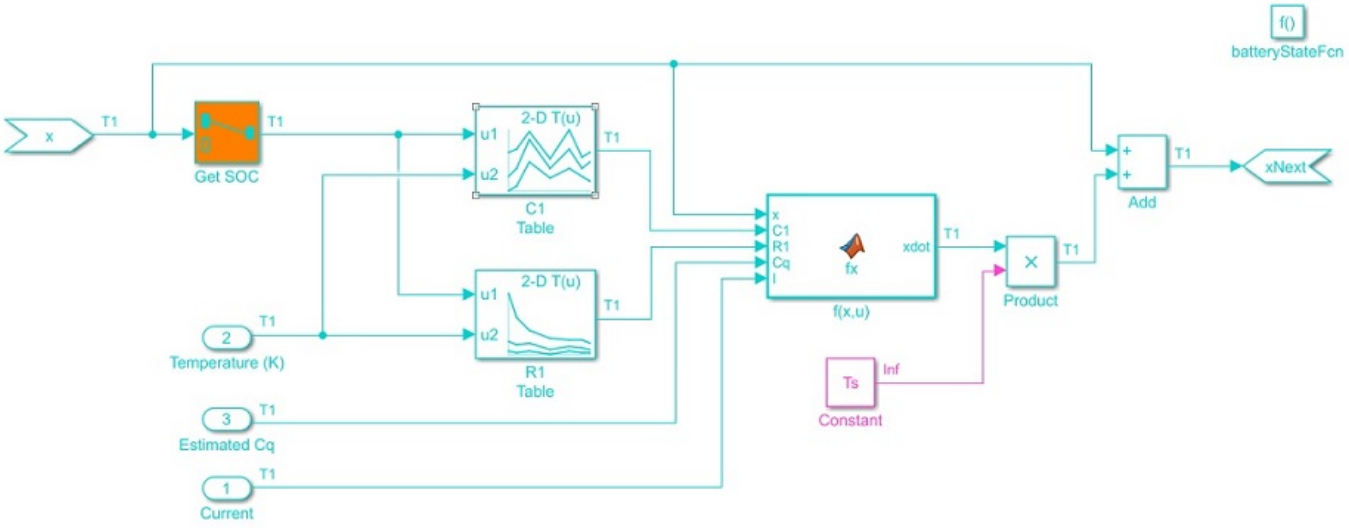
The state vectors of the nonlinear battery system are:

$$x_T = \begin{pmatrix} SOC_T \\ U_{1_T} \end{pmatrix}.$$

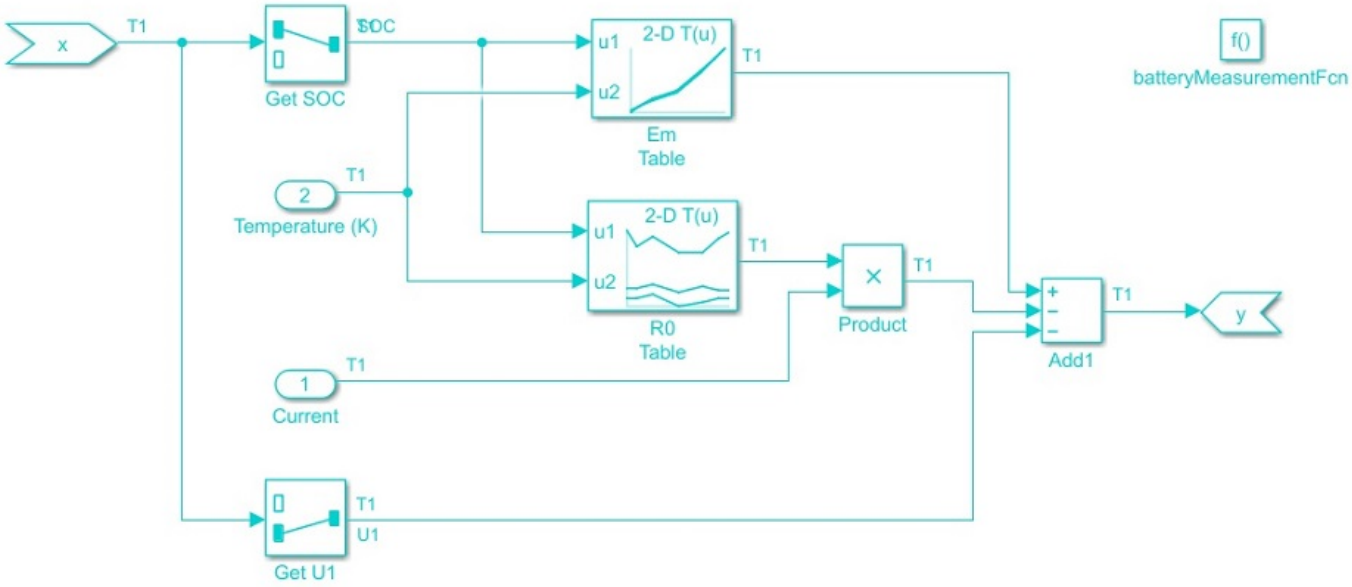
Applying Euler discretization gives the following equations:

$$\begin{pmatrix} SOC_{T+1} \\ U_{1_{T+1}} \end{pmatrix} = \begin{pmatrix} SOC_T \\ U_{1_T} \end{pmatrix} + \begin{pmatrix} -\frac{1}{3600*C_q} I \\ -\frac{1}{R_1(SOC_T,T_b)*C_1(SOC_T,T_b)} U_1 + \frac{1}{C_1(SOC_T,T_b)} I \end{pmatrix} Ts + W_T$$

The discretized state transition equation is implemented in the Simulink function named `batteryStateFcn`. The function input `x` is the state vector, and the function output `xNext` is the state vector at the next step, calculated using the discretized state transition equations. In the function, you need to specify the signal dimensions and data type of `x` and `xNext`. In this example, the signal dimension for `x` and `xNext` is 2, and the data type is double. Additional inputs to `batteryStateFcn` are the temperature, estimated capacity, and current. Note that the additional inputs are inputs to the state transition equations and are not required by the Unscented Kalman Filter block.

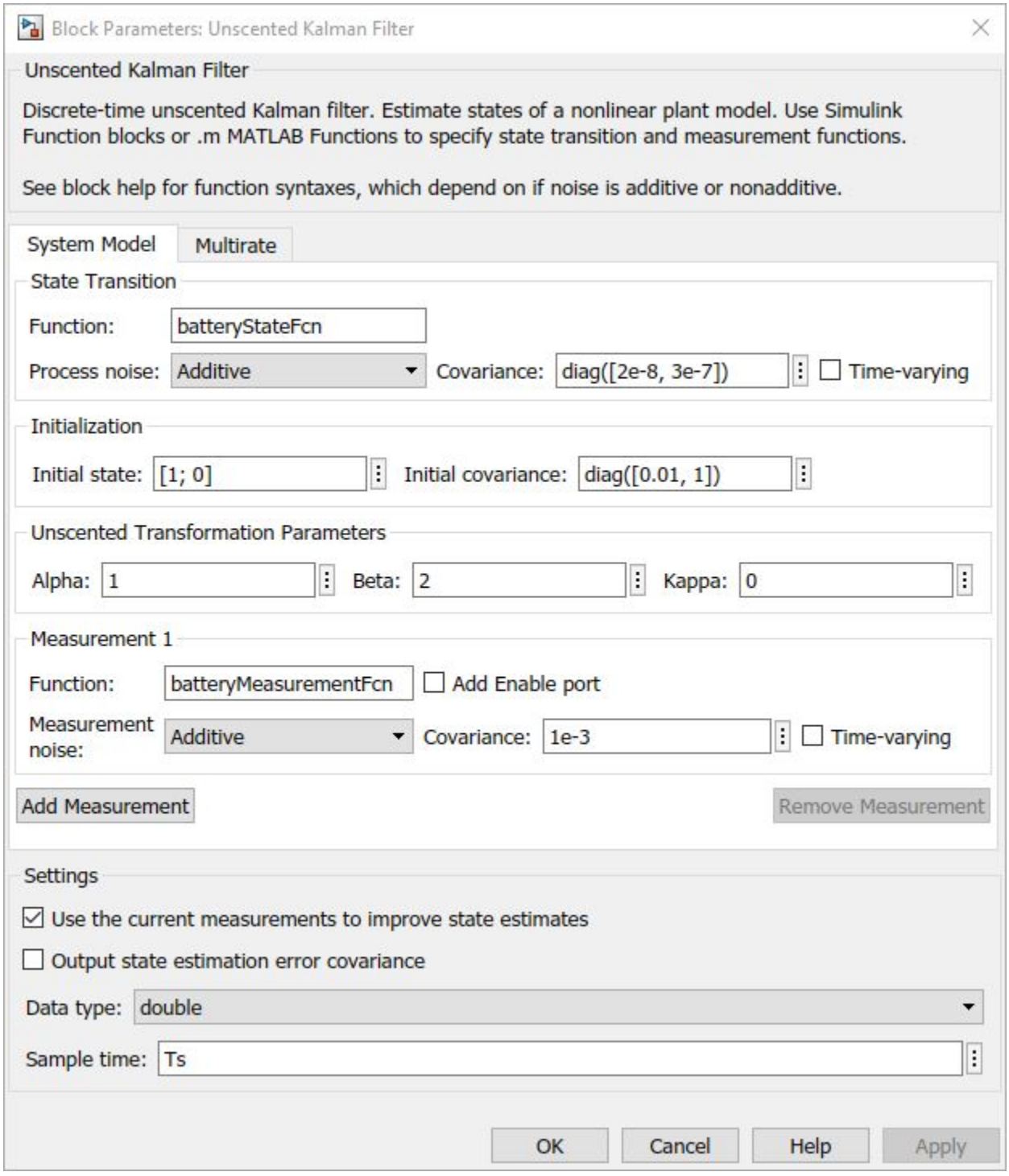


Similarly, the measurement function is also implemented in the Simulink function named `batteryMeasurementFcn`.



Configure the Unscented Kalman Filter block parameters as follows:

In the **System Model** tab, specify the block parameters as shown:



You specify the following parameters:

- **Function in State Transition:** `batteryStateFcn`.

The name of the Simulink function defined previously that implements the discretized state transition equation.

- **Process noise:** `Additive`, with time-invariant covariance $\left[\begin{array}{cc} 2e-8 & 0 \\ 0 & 3e-7 \end{array} \right]$. `Additive` implies that the noise term is added to the final signals directly.

The process noise for SOC and U_1 are estimated based on the dynamic characteristics of the battery system. The battery has a nominal capacity of 30 Ah and undergoes either discharge or charge cycles at an average current amplitude of 15A. Therefore, one discharging or charging process would take around 2 hours (7200 seconds). The maximum change is 100% for SOC and around 4 volts for U_1.

The maximum changes per step in SOC and U_1 are $max(|dSOC|) \approx \frac{100\%}{3600*2} * Ts$ and $max(|dU_1|) \approx \frac{4}{3600*2} * Ts$, where T_s is the sampling time of the filter. In this example, T_s is set to be 1 second.

Thus, the process noise W is:

$$W = \left[\begin{array}{cc} (max(|dSOC|))^2 & 0 \\ 0 & (max(|dU_1|))^2 \end{array} \right] \approx \left[\begin{array}{cc} 2e-8 & 0 \\ 0 & 3e-7 \end{array} \right].$$

- **Initial State**: $\left(\begin{array}{c} 1 \\ 0 \end{array} \right)$.

The initial value for SOC is assumed to be 100% (fully charged battery) while initial value for U_1 is set to be 0, as we do not have any prior information of U_1.

- **Initial Covariance**:

Initial covariance indicates how accurate and reliable the initial guesses are. Assume the maximum initial guess error is 10% for SOC and 1V for U_1. The initial covariance matrix is set to be $\left[\begin{array}{cc} 0.01 & 0 \\ 0 & 1 \end{array} \right]$.

- **Unscented Transformation Parameters**: The parameter are specified based on [2]

```
- Alpha: 1. Determine the spread of sigma points around x. Set Alpha to be 1 for larger spre
- Beta: 2. Used to incorporate prior knowledge of the distribution. The nominal value for Be
- Kappa: 0. Secondary scaling parameter. The nominal value for Kappa is 0.
```

- **Function in Measurement:** `batteryMeasurementFcn`.

The name of the Simulink function defined previously that implements the measurement function.

- **Measurement noise:** `Additive`, with time-invariant covariance 1e-3.

The measurement noise `V` is estimated based on measurement equipment accuracy. A voltage meter for battery voltage measurement has approximately 1% accuracy. The battery voltage is around 4V. Equivalently, we have $max(dE_m) \approx 4 * 1\% = 0.04$. Therefore, $V = (max(dE_m))^2 \approx 1e-3$.

- **Sample Time**: Ts.

Estimate Battery Degradation

The battery degradation is modeled by decreasing capacity C_q. In this example, the battery capacity is set to decrease 1 Ah per discharge-charge cycle to illustrate the effect of degradation. Since the degradation rate of capacity is not known in advance, set the state equation of C_q to a random walk:

$$C_{q_{k+1}} = C_{q_k} + W_{C_q},$$

where k is the number of discharge-charge cycles, and W_{C_q} is the process noise.

The battery is configured to automatically charge when the capacity is at 30% and switch to discharging when the capacity is 90%. Use this information to measure the battery capacity by integrating the current I over a charge or discharge cycle (coulomb counting).

The measurement equation for C_q is:

$$C_{q_k}^{Measured} = C_{q_k} + V_{C_q} = \frac{\int_{t_{k-1}}^{t_k} I dt}{(\Delta SOC)_{nominal}} = \frac{\int_{t_{k-1}}^{t_k} I dt}{|0.9 - 0.3|} = \frac{\int_{t_{k-1}}^{t_k} I dt}{0.6}$$

where V_{C_q} is the measurement noise.

The state transition and measurement equations of battery degradation can be put into the following state-space form:

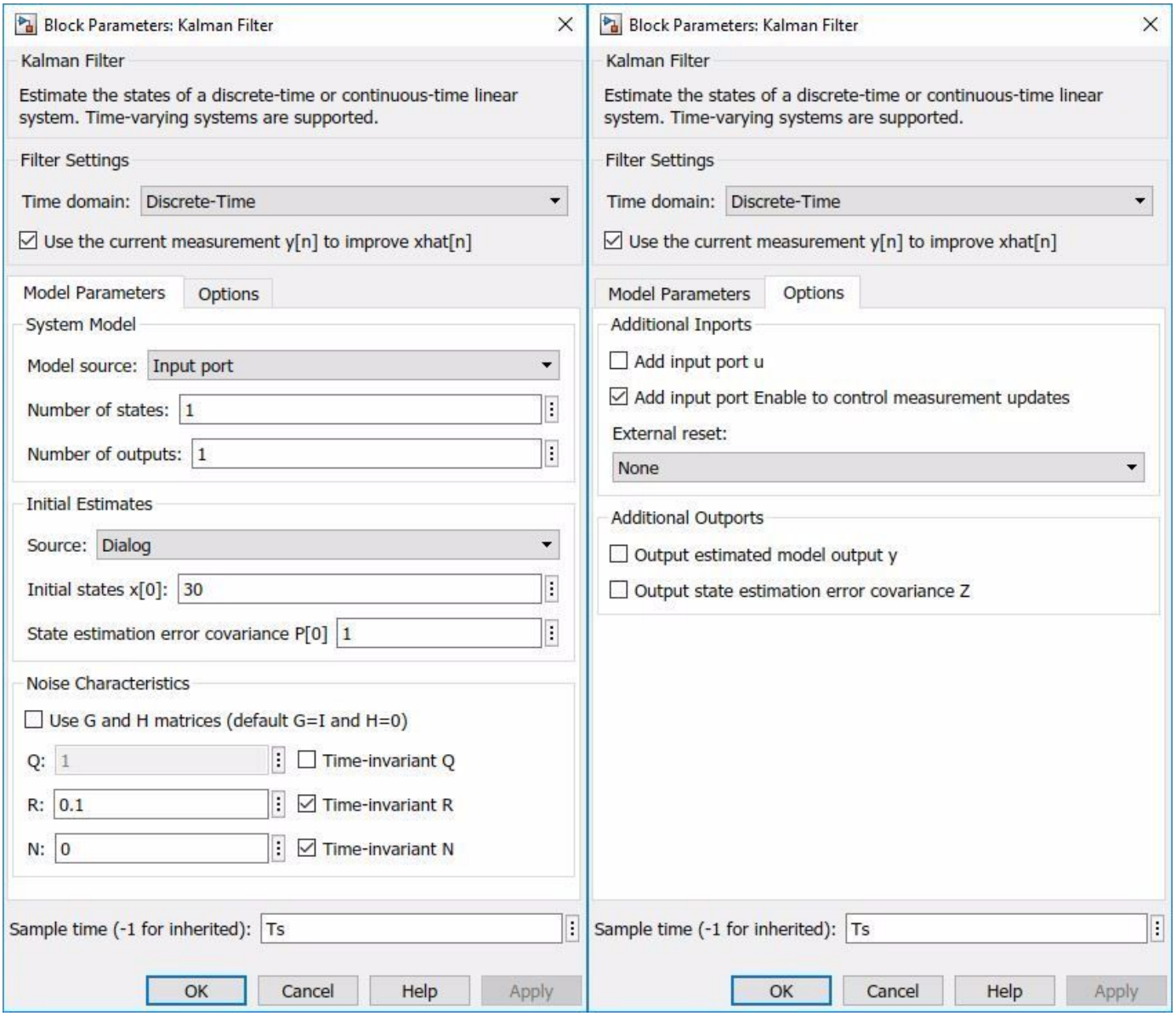
$$C_{q_{k+1}} = A_{C_q} C_{q_k} + W_{C_q}$$

$$C_{q_k}^{Measured} = C_{C_q} C_{q_k} + V_{C_q}$$

where A_{C_q} and C_{C_q} are equal to 1.

For the above linear system, use a Kalman Filter to estimate battery capacity. The estimated C_q from the linear Kalman Filter is used to improve SOC estimation. In the example, an event-based linear Kalman filter is used to estimate C_q. Since C_q is measured once over a charge or discharge cycle, the linear Kalman Filter is enabled only when charging or discharging ends.

Configure the block parameters as follows:



In the **Model Parameters** tab, specify the plant model and noise characteristics:

- **Model source:** `Input Port`.

To implement an event-based Kalman Filter, the state equation is enabled only when the event happens. In other word, the state equation is event-based as well. For a linear system $x_{t+1} = Ax_t + Bu_t + w_t$, set the state equation to be

$$x_{t+1} = \begin{cases} Ax_t + Bu_t + w_t, t = t_{enabled} \\ x_t, t \neq t_{enabled} \end{cases}.$$

- **A**: $\begin{cases} A_{C_q}, t = t_{enabled} \\ 1, t \neq t_{enabled} \end{cases}$. In this example, $A_{C_q} = 1$. Therefore, A equals to 1 all the time.

- **C**: 1, from $C_{q_k}^{Measured} = C_{q_k} + V_{C_q} = \frac{\int_{t_{k-1}}^{t_k} Idt}{0.6}$.
- **Initial Estimate Source**: `Dialog`. You specify initial states in `Initial state x[0]`
- **Initial states x[0]**: 30. This is the nominal capacity of the battery (30Ah).
- **Q**: $\begin{cases} 1, t = t_{enabled} \\ 0, t \neq t_{enabled} \end{cases}$

This is the covariance of the process noise W_{C_q}. Since the degradation rate in the capacity is around 1 Ah per discharge-charge cycle, set the process noise to be 1.

- **R**: 0.1. This is the covariance of the measurement noise V_{C_q}. Assume that the capacity measurement error is less than 1%. With a battery capacity of 30 Ah, the measurement noise $V_{C_q} \approx (0.3)^2 \approx 0.1$.
- **Sample Time**: Ts.

In the **Options** tab, add an input port `Enable` to control measurement updates. The enable port is used to update the battery capacity estimate on either charge or discharge events, as opposed to continually updating.

Note that setting `Enable` to 0 does not disable predictions using state equations. Therefore, the state equation is configured to be event-based as well. By setting an event-based A and Q for the Kalman Filter block, predictions using state equations are disabled when `Enable` is set to be 0.

Results

To simulate the system, load the battery parameters stored in the file `BatteryParameters.mat`. The file contains the battery parameters including $E_m(SOC,T)$, $R_0(SOC,T)$, $R_1(SOC,T)$.

```
load BatteryParameters.mat
```

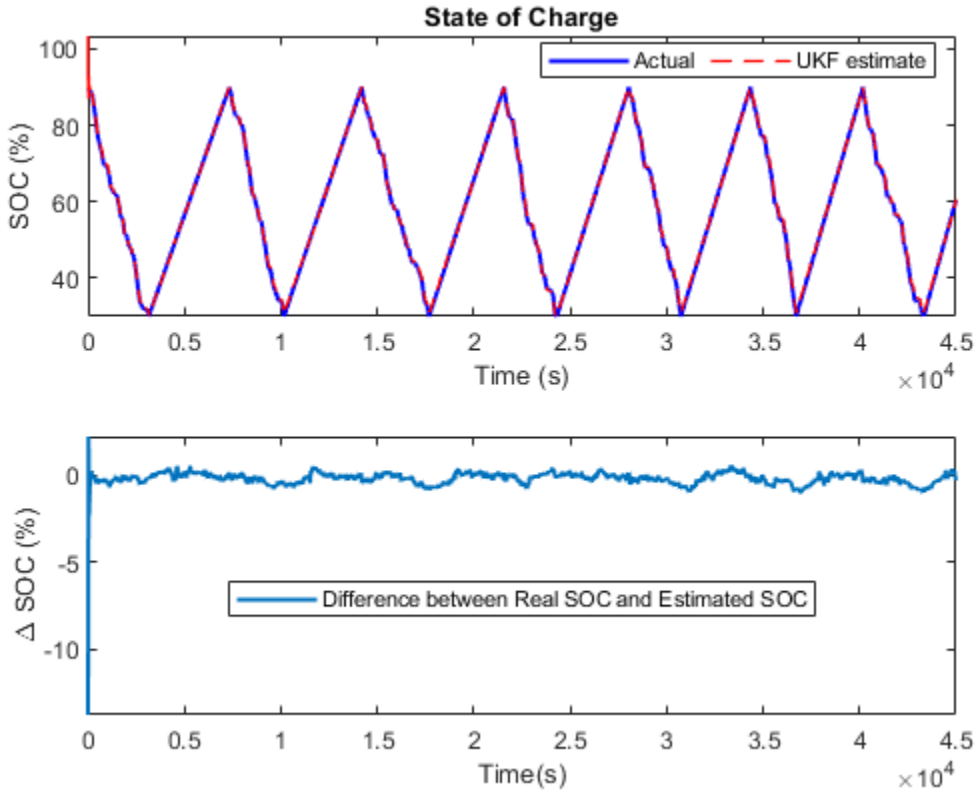
Simulate the system.

```
sim('BatteryExampleUKF')
```

At every time step, the Unscented Kalman Filter provides an estimation for SOC, based on voltage measurements E_m. Plot the real SOC, the estimated SOC, and the difference between them.

```
% Synchronize two time series
[RealSOC, EstimatedSOC] = synchronize(RealSOC, EstimatedSOC, 'intersection');

figure;
subplot(2,1,1)
plot(100*RealSOC,'b','LineWidth',1.5);
hold on
plot(100*EstimatedSOC,'r--','LineWidth',1);
title('State of Charge');
xlabel('Time (s)');
ylabel('SOC (%)');
legend('Actual','UKF estimate','Location','Best','Orientation','horizontal');
axis tight
```

```
subplot(2,1,2)
DiffSOC = 100*(RealSOC - EstimatedSOC);
plot(DiffSOC.Time, DiffSOC.Data, 'LineWidth', 1.5);
xlabel('Time(s)');
ylabel('\Delta SOC (%)','Interpreter','Tex');
legend('Difference between Real SOC and Estimated SOC','Location','Best')
axis tight
```

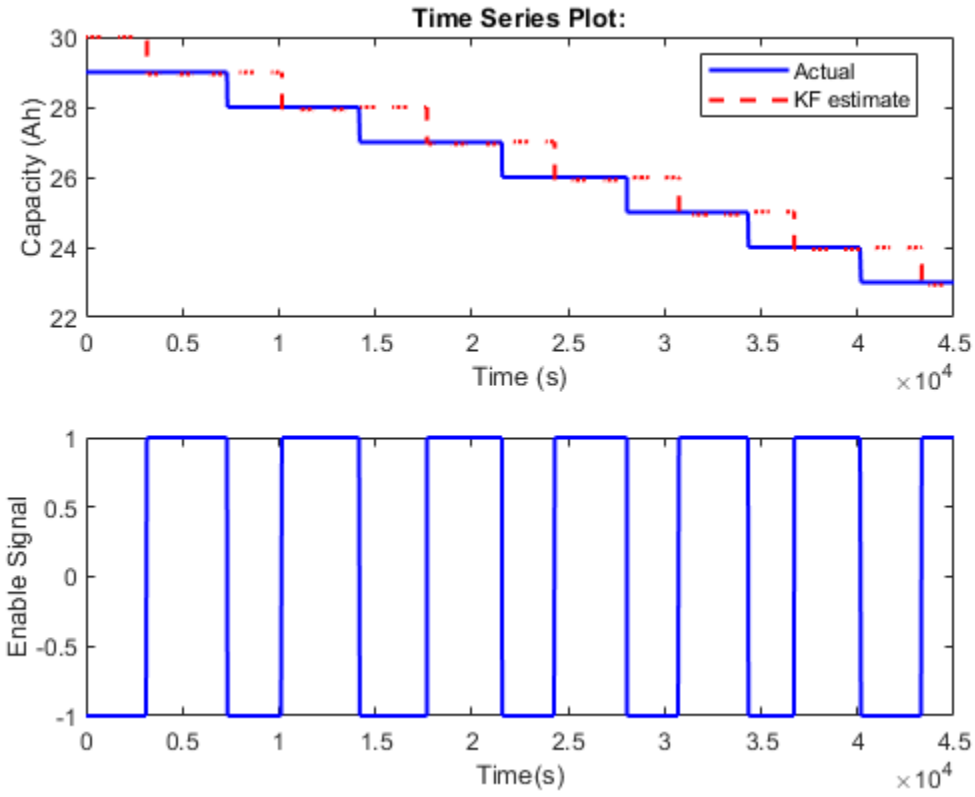


After an initial estimation error, the SOC converges quickly to the real SOC. The final estimation error is within 0.5% error. Thus, the Unscented Kalman Filter gives an accurate estimation of SOC.

At every discharge-charge transition, the battery capacity is estimated to improve the SOC estimation. The battery system outputs indicator signals to inform what process the battery is in. Discharging process is represented by -1 in the indicator signals while charging process is represented by 1. In this example, changes in the indicator signals are used to determine when to enable or disable Kalman Filter for capacity estimation. We plot the real and estimated capacity as well as the charge-discharge indicator signals.

```
figure;
subplot(2,1,1);
plot(RealCapacity,'b','LineWidth',1.5);
hold on
plot(EstimatedCapacity,'r--','LineWidth',1.5);
xlabel('Time (s)');
ylabel('Capacity (Ah)');
legend('Actual','KF estimate','Location','Best');
```

```
subplot(2,1,2);
plot(DischargeChargeIndicator.Time,DischargeChargeIndicator.Data,'b','LineWidth',1.5);
xlabel('Time(s)');
ylabel('Enable Signal');
```



In general, the Kalman Filter is able to track the real capacity. There is half cycle delay between estimated capacity and real capacity. This is because the battery capacity degradation happens when one full discharge-charge cycle ends. While the coulomb counting gives a capacity measurement of the last discharge or charge cycle.

Summary

This example shows how to use the Unscented Kalman Filter block to perform nonlinear state estimation for a lithium battery. In addition, steps to develop an event-based Kalman Filter for battery capacity estimation are illustrated. The newly estimated capacity is used to improve SOC estimation in the Unscented Kalman Filter.

Reference

[1] Huria, Tarun, et al. "High fidelity electrical model with thermal dependence for characterization and simulation of high power lithium battery cells." Electric Vehicle Conference (IEVC), 2012 IEEE International. IEEE, 2012.

[2] Wan, Eric A., and Rudolph Van Der Merwe. "The unscented Kalman filter for nonlinear estimation." Adaptive Systems for Signal Processing, Communications, and Control Symposium 2000. AS-SPCC. The IEEE 2000. Ieee, 2000.

See Also

Extended Kalman Filter | Unscented Kalman Filter | Particle Filter

More About

- “Extended and Unscented Kalman Filter Algorithms for Online State Estimation” on page 13-2
- “Validate Online State Estimation in Simulink” on page 13-14
- “Troubleshoot Online State Estimation” on page 13-17
- “Estimate States of Nonlinear System with Multiple, Multirate Sensors” on page 13-33

Parameter and State Estimation in Simulink Using Particle Filter Block

This example demonstrates the use of Particle Filter block in Control System Toolbox™. A discrete-time transfer function parameter estimation problem is reformulated and recursively solved as a state estimation problem.

Introduction

The Control System Toolbox has three Simulink blocks for nonlinear state estimation:

- Particle Filter: Implements a discrete-time particle filter algorithm.
- Extended Kalman Filter: Implements the first-order discrete-time extended Kalman filter algorithm.
- Unscented Kalman Filter: Implements the discrete-time unscented Kalman filter algorithm.

These blocks support state estimation using multiple sensors operating at different sample rates. A typical workflow for using these blocks is as follows:

1. Model your plant and sensor behavior using MATLAB or Simulink functions.
2. Configure the parameters of the block.
3. Simulate the filter and analyze results to gain confidence in filter performance.
4. Deploy the filter on your hardware. You can generate code for these filters using Simulink Coder™ software.

This example uses the Particle Filter block to demonstrate the first two steps of this workflow. The last two steps are briefly discussed in the **Next Steps** section. The goal in this example is to estimate the parameters of a discrete-time transfer function (an output-error model) recursively, where the model parameters are updated at each time step as new information arrives.

If you are interested in the Extended Kalman Filter, see the example "Estimate States of Nonlinear System with Multiple, Multirate Sensors". The use of Unscented Kalman Filter follows similar steps to Extended Kalman Filter.

Plant Modeling

Most state estimation algorithms rely on a plant model (state transition function) that describe the evolution of plant states from one time step to the next. This function is typically denoted as $x[k+1] = f(x[k], w[k], u[k])$ where x is the states, w is the process noise, u is the optional additional inputs, for instance system inputs or parameters. The particle filter block requires you to provide this function in a slightly different syntax, $X[k+1] = f_{pf}(X[k], u[k])$. The differences are:

- Particle filter works by following the trajectories of many state hypotheses (particles), and the block passes all state hypotheses to your function at once. Concretely, if your state vector x has N_s elements and you choose N_p particles to use, X has the dimensions $[N_s\ N_p]$ where each column is a state hypothesis.
- You calculate the impact of process noise w on the state hypotheses $X[k+1]$ in your function $f_{pf}(...)$. The block does not make any assumptions on the probability distribution of the process noise w, and does not need w as an input.

The function $f_{pf}(\dots)$ can be a MATLAB Function that comply with the restrictions of MATLAB Coder™, or a Simulink Function block. After you create $f_{pf}(\dots)$, you specify the function name in the Particle Filter block.

In this example, you are reformulating a discrete-time transfer function parameter estimation problem as a state estimation problem. This transfer function may be representing the dynamics of a discrete-time process, or it may be representing some continuous-time dynamics coupled with a signal reconstructor such as zero-order hold. Assume that you are interested in estimating the parameters of a first-order discrete-time transfer function:

$$y[k] = \frac{20q^{-1}}{1 - 0.7q^{-1}}u[k] + e[k]$$

Here $y[k]$ is the plant output, $u[k]$ is the plant input, $e[k]$ is the measurement noise, q^{-1} is the time-delay operator so that $q^{-1}u[k] = u[t-1]$. Parametrize the transfer function as $\frac{n\,q^{-1}}{1 + d\,q^{-1}}$, where n and d are parameters to be estimated. The transfer function and the parameters can be represented in the necessary state-space form in multiple ways, by choice of the state vector. One choice is $x[k] = [\,y[k]; d[k]; n[k]\,]$ where the second and third states represent the parameter estimates. Then the transfer function can be equivalently written as

$$x[k+1] = \begin{bmatrix} -x_2[k]x_1[k] + x_3[k]u[k] \\ x_2[k] \\ x_3[k] \end{bmatrix}$$

The measurement noise term $e[k]$ is handled in sensor modeling. In this example, you implement the expression above in a MATLAB Function, in a vectorized form for computational efficiency:

```
type pfBlockStateTransitionFcnExample
```

```
function xNext = pfBlockStateTransitionFcnExample(x,u)
% pfBlockStateTransitionFcnExample State transition function for particle
%                                  filter, for estimating parameters of a
%                                  SISO, first order, discrete-time transfer
%                                  function model
%
% Inputs:
%    x - Particles, a NumberOfStates-by-NumberOfParticles matrix
%    u - System input, a scalar
%
% Outputs:
%    xNext - Predicted particles, with the same dimensions as input x

% Implement the state-transition function
%    xNext = [x(1)*x(2) + x(3)*u;
%             x(2);
%             x(3)];
% in vectorized form (for all particles).
xNext = x;
xNext(1,:) = bsxfun(@times,x(1,:),-x(2,:)) + x(3,:)*u;
```

```
% Add a small process noise (relative to expected size of each state), to
% increase particle diversity
xNext = xNext + bsxfun(@times,[1; 1e-2; 1e-1],randn(size(xNext)));
end
```

Sensor modeling

The Particle Filter block requires you to provide a measurement likelihood function that calculates the likelihood (probability) of each state hypothesis. This function has the form $L[k] = h_{pf}(X[k], y[k], u[k])$. $L[k]$ is an N_p element vector, where N_p is the number of particles you choose. m^{th} element in $L[k]$ is the likelihood of the m^{th} particle (column) in $X[k]$. $y[k]$ is the sensor measurement. $u[k]$ is an optional input argument, which can differ from the inputs of the state transition function.

The sensor measures the first state in this example. This example assumes that the errors between the actual and predicted measurements are distributed according to a Gaussian distribution, but any arbitrary probability distribution or some other method can be used to calculate the likelihoods. You create $h_{pf}(...)$, nd specify the function name in the Particle Filter block.

```
type pfBlockMeasurementLikelihoodFcnExample
```

```
function likelihood = pfBlockMeasurementLikelihoodFcnExample(particles, measurement)
% pfBlockMeasurementLikelihoodFcnExample Measurement likelihood function for particle filter
%
% The measurement is the first state
%
% Inputs:
%    particles   - A NumberOfStates-by-NumberOfParticles matrix
%    measurement - System output, a scalar
%
% Outputs:
%    likelihood - A vector with NumberOfParticles elements whose n-th
%                 element is the likelihood of the n-th particle

%#codegen

% Predicted measurement
yHat = particles(1,:);

% Calculate likelihood of each particle based on the error between actual
% and predicted measurement
%
% Assume error is distributed per multivariate normal distribution with
% mean value of zero, variance 1. Evaluate the corresponding probability
% density function
e = bsxfun(@minus, yHat, measurement(:)'); % Error
numberOfMeasurements = 1;
mu = 0; % Mean
Sigma = eye(numberOfMeasurements); % Variance
measurementErrorProd = dot((e-mu), Sigma \ (e-mu), 1);
c = 1/sqrt((2*pi)^numberOfMeasurements * det(Sigma));
likelihood = c * exp(-0.5 * measurementErrorProd);
end
```

Filter Construction

Configure the Particle Filter block for estimation. You specify the state transition and measurement likelihood function names, number of particles, and the initial distribution of these particles.

In the **System Model** tab of the block dialog, specify the following parameters:

State Transition

1. Specify the state transition function, `pfBlockStateTransitionFcnExample`, in **Function**. When you enter the function name and click **Apply**, the block detects that your function has an extra input, u, and creates the input port **StateTransitionFcnInputs**. You connect your system input to this port.

Initialization

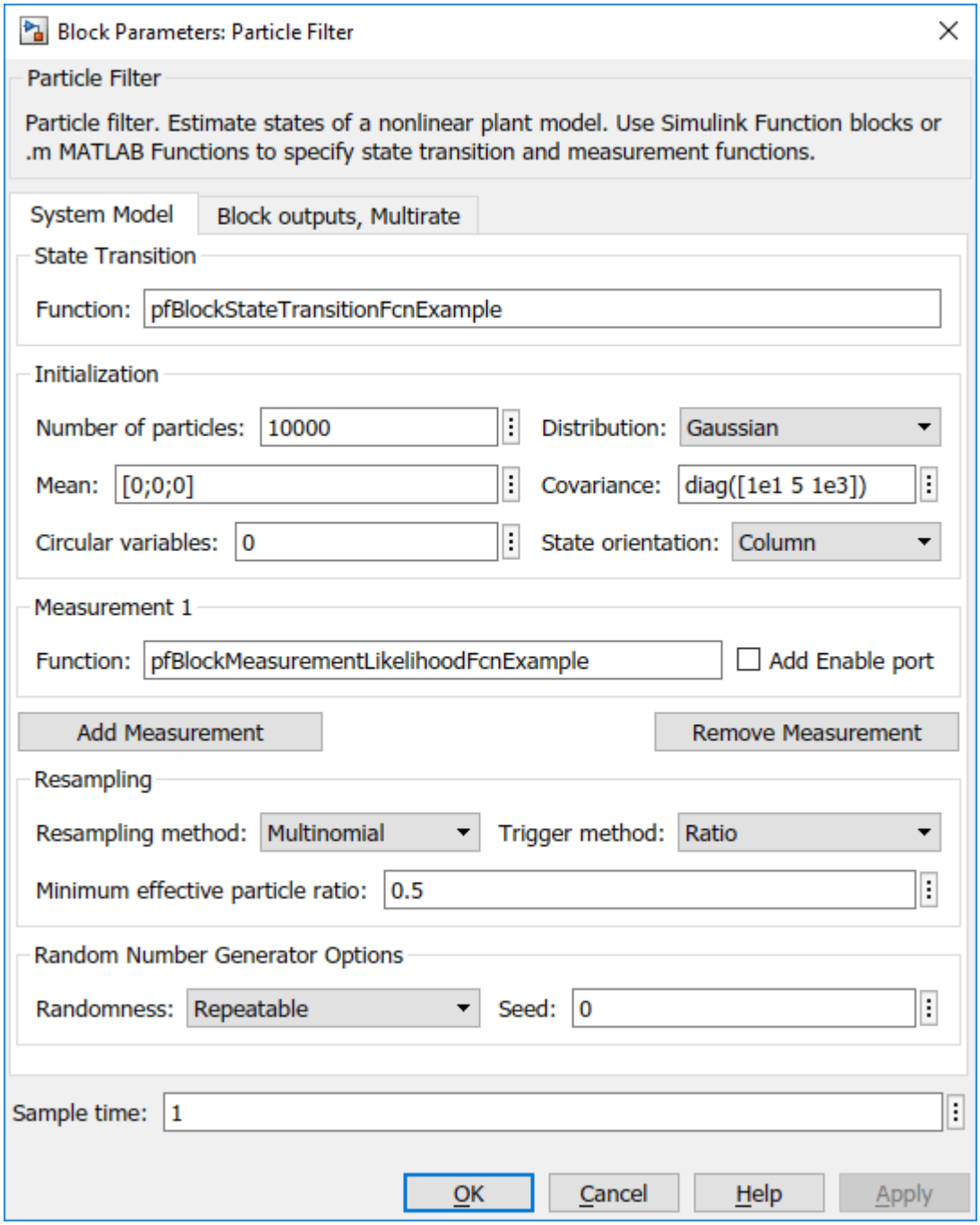
1. Specify `10000` in **Number of particles**. Higher number of particles typically correspond to better estimation, at increased computational cost.
2. Specify `Gaussian` in **Distribution** to get an initial set of particles from a multivariate Gaussian distribution. Then specify `[0; 0; 0]` in **Mean** because you have three states and this is your best guess. Specify `diag([10 5 100])` in **Covariance** to specify a large variance (uncertainty) in your guess for the third state, and smaller variance for the first two. It is critical that this initial set of particles are spread wide enough (large variance) to cover the potential true state.

Measurement 1

1. Specify the name of your measurement likelihood function, `pfBlockMeasurementLikelihoodFcnExample`, in **Function**.

Sample Time

1. At the bottom of the block dialog, enter 1 in **Sample time**. If you have a different sample time among state transition and measurement likelihood functions, or if you have multiple sensors with different sample times, these can be configured in the **Block outputs, Multirate** tab.



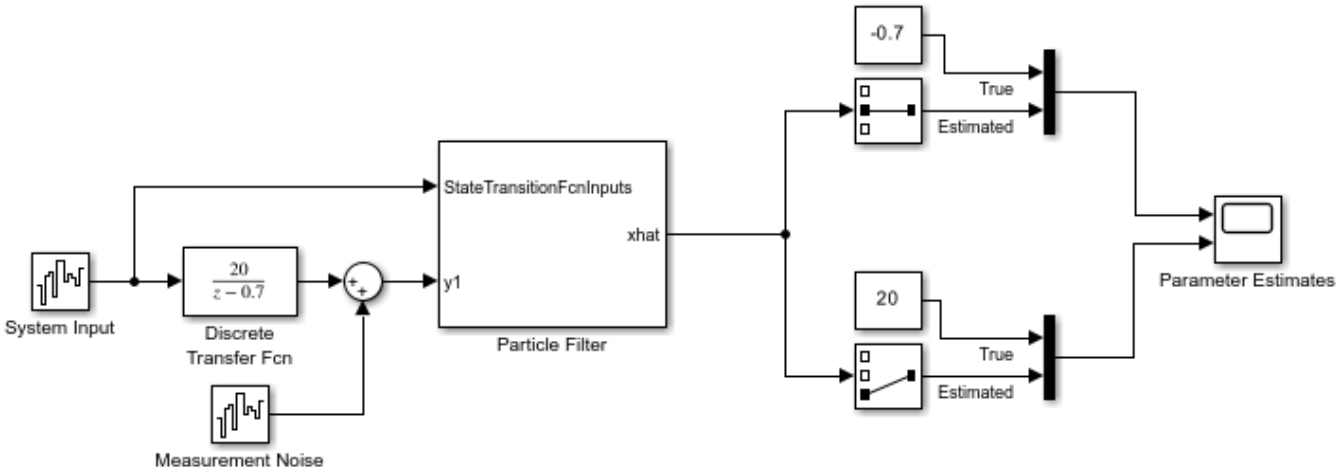
The particle filter involves deleting the particles with low likelihoods and seeding new particles using the ones with higher likelihoods. This is controlled by the options under the **Resampling** group. This example uses the default settings.

By default, the block only outputs the mean of the state hypotheses, weighted by their likelihoods. To see all the particles, weights, or to choose a different method of extracting a state estimate, check out the options in the **Block outputs, Multirate** tab.

Simulation and Results

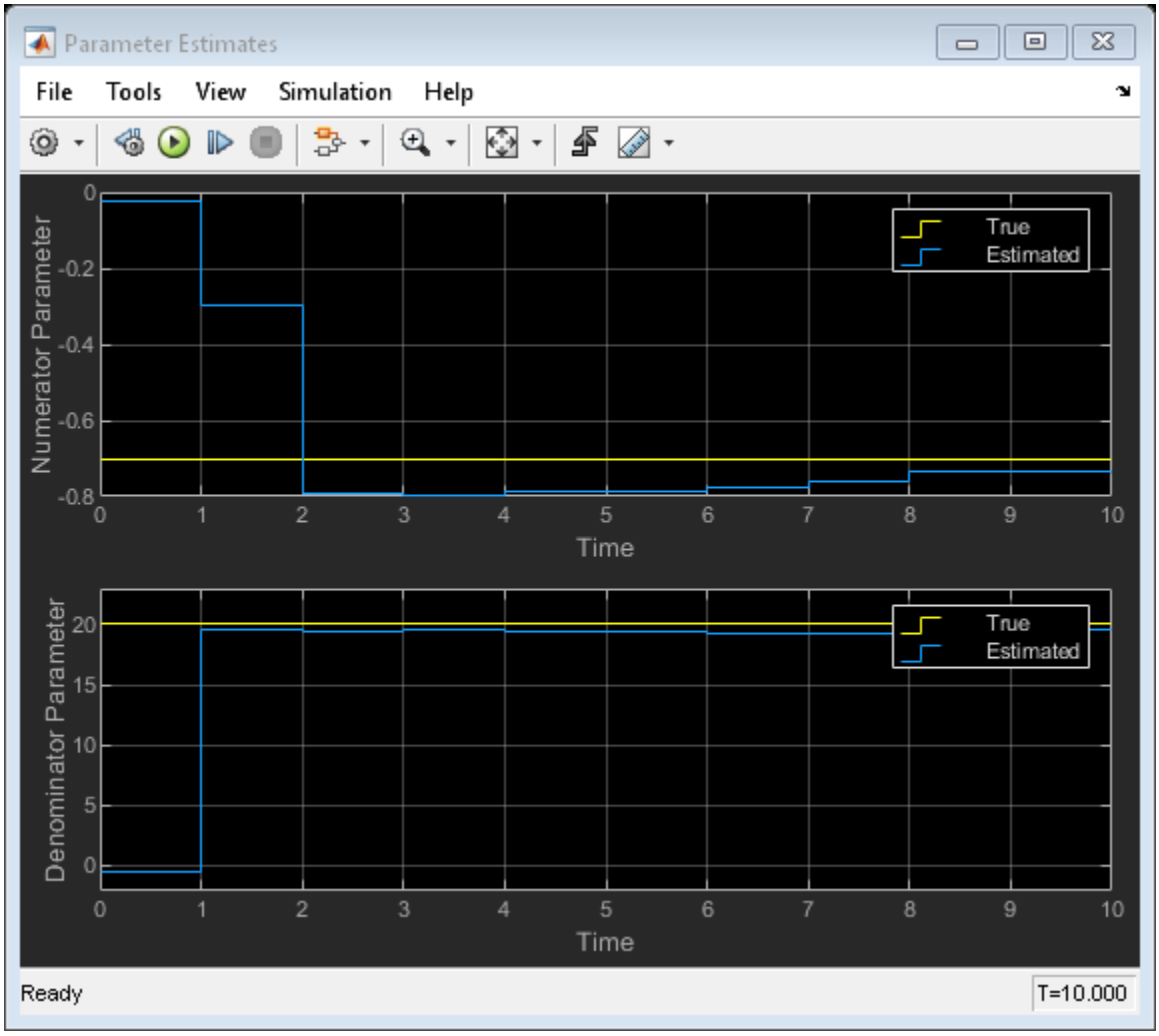
For a simple test, the true plant model is simulated with white noise inputs. The inputs and the noisy measurements from the plant are fed to the Particle Filter block. The following Simulink model represents this setup:

```
open_system('pfBlockExample')
```



Simulate the system and compare the estimated and true parameters:

```
sim('pfBlockExample')
open_system('pfBlockExample/Parameter Estimates')
```



The plot shows the true numerator and denominator parameters, and their particle filter estimates. The estimates approximately converge to the true values after 10 time steps. Convergence is obtained even though the initial state guess was far from the true values.

Troubleshooting

A few potential implementation issues and troubleshooting ideas are listed here, in case the particle filter is not performing as expected for your application.

Troubleshooting of the particle filter is typically performed by looking into the set of particles and their weights, which can be obtained by choosing **Output all particles** and **Output all weights** in the **Block outputs, Multirate** tab of the block dialog.

The first sanity check is to ensure that the state transition and measurement likelihood functions capture the behavior of your system reasonably well. If you have a simulation model for your system (and hence access to the true state in simulation), you can try initializing the filter with the true state. Then you can validate if the state transition function calculates the time-propagation of the true state accurately, and f the measurement likelihood function is calculating a high likelihood for these particles.

Initial set of particles is important. Ensure that at least some of the particles have high likelihood at the beginning of your simulation. If the true state is outside the initial spread of state hypotheses, the estimates can be inaccurate or even diverge.

If the state estimation accuracy is fine initially, but deteriorating over time, the issue may be particle degeneracy or particle impoverishment [2]. Particle degeneracy occurs when the particles are distributed too widely, while particle impoverishment occurs because of particles clumping together after resampling. Particle degeneracy leads to particle impoverishment as a result of direct resampling. The addition of artificial process noise in the state-transition function used in this example is one practical approach. There is a large collection of literature on resolving these issues and based on your application more systematic approaches may be available. [1], [2] are two references that can be helpful.

Next Steps

1. Validate the state estimation: Once the filter is performing as expected in a simulation, typically the performance is further validated using extensive Monte Carlo simulations. For more information, see “Validate Online State Estimation in Simulink” on page 13-14. You can use the options under **Randomness** group in the Particle Filter block dialog to facilitate these simulations.
2. Generate code: The Particle Filter block supports C and C++ code generation using Simulink Coder™ software. The functions you provide to this block must comply with the restrictions of MATLAB Coder™ software (if you are using MATLAB functions to model your system) and Simulink Coder software (if you are using Simulink Function blocks to model your system).

Summary

This example has shown how to use the Particle Filter block in Control System Toolbox. You estimated the parameters of a discrete-time transfer function recursively, where the parameters are updated at each time step as new information arrives.

```
close_system('pfBlockExample', 0)
```

References

[1] Simon, Dan. Optimal state estimation: Kalman, H infinity, and nonlinear approaches. John Wiley & Sons, 2006.

[2] Doucet, Arnaud, and Adam M. Johansen. "A tutorial on particle filtering and smoothing: Fifteen years later." Handbook of nonlinear filtering 12.656-704 (2009): 3.

See Also

Extended Kalman Filter | Unscented Kalman Filter | Particle Filter

More About

- “Validate Online State Estimation in Simulink” on page 13-14
- “Troubleshoot Online State Estimation” on page 13-17
- “Estimate States of Nonlinear System with Multiple, Multirate Sensors” on page 13-33

State-Space Control Design

- “Extended and Unscented Kalman Filter Algorithms for Online State Estimation” on page 13-2
- “Generate Code for Online State Estimation in MATLAB” on page 13-9
- “Validate Online State Estimation at the Command Line” on page 13-12
- “Validate Online State Estimation in Simulink” on page 13-14
- “Troubleshoot Online State Estimation” on page 13-17
- “Nonlinear State Estimation Using Unscented Kalman Filter and Particle Filter” on page 13-18
- “Estimate States of Nonlinear System with Multiple, Multirate Sensors” on page 13-33
- “Regulate Pressure in Drum Boiler” on page 13-43
- “State Estimation with Wrapped Measurements Using Extended Kalman Filter” on page 13-50
- “Detect Replay Attacks in DC Microgrids Using Distributed Watermarking” on page 13-57
- “Detect Attack in Cyber-Physical Systems Using Dynamic Watermarking” on page 13-65

Extended and Unscented Kalman Filter Algorithms for Online State Estimation

You can use discrete-time extended and unscented Kalman filter algorithms for online state estimation of discrete-time nonlinear systems. If you have a system with severe nonlinearities, the unscented Kalman filter algorithm may give better estimation results. You can perform the state estimation in Simulink and at the command line. To perform the state estimation, you first create the nonlinear state transition function and measurement function for your system.

At the command line, you use the functions to construct the `extendedKalmanFilter` or `unscentedKalmanFilter` object for desired algorithm, and specify whether the process and measurement noise terms in the functions are additive or non-additive. After you create the object, you use the `predict` and `correct` commands to estimate the states using real-time data. For information about the order in which to execute these commands, see the `predict` and `correct` reference pages.

In Simulink, you specify these function in the Extended Kalman Filter and Unscented Kalman Filter blocks. You also specify whether the process and measurement noise terms in the functions are additive or non-additive. In the blocks, the software decides the order in which prediction and correction of state estimates is done.

Extended Kalman Filter Algorithm

The `extendedKalmanFilter` command and Extended Kalman Filter block implement the first-order discrete-time Kalman filter algorithm. Assume that the state transition and measurement equations for a discrete-time nonlinear system have non-additive process and measurement noise terms with zero mean and covariance matrices Q and R, respectively:

$$x[k+1] = f(x[k], w[k], u_s[k])$$
$$y[k] = h(x[k], v[k], u_m[k])$$
$$w[k] \sim (0, Q[k])$$
$$v[k] \sim (0, R[k])$$

Here f is a nonlinear state transition function that describes the evolution of states `x` from one time step to the next. The nonlinear measurement function h relates `x` to the measurements `y` at time step `k`. These functions can also have additional input arguments that are denoted by $\mathtt{u_s}$ and $\mathtt{u_m}$. The process and measurement noise are `w` and `v`, respectively. You provide Q and R.

In the block, the software decides the order of prediction and correction of state estimates. At the command line, you decide the order. For information about the order in which to execute these commands, see the `predict` and `correct` reference pages. Assuming that you implement the `correct` command before `predict`, the software implements the algorithm as follows:

1 Initialize the filter object with initial values of the state, `x[0]`, and state estimation error covariance matrix, P.

$$\hat{x}[0|-1] = E(x[0])$$
$$P[0|-1] = E(x[0] - \hat{x}[0|-1])(^{x}$$

Here $\hat{x}$ is the state estimate and $\hat{x}[k_a|k_b]$ denotes the state estimate at time step k_a using measurements at time steps `0,1,...,`k_b. So $\hat{x}[0|-1]$ is the best guess of the state value before you make any measurements. You specify this value when you construct the filter.

2 For time steps $k = 0,1,2,3,...$, perform the following:

a Compute the Jacobian of the measurement function, and update the state and state estimation error covariance using the measured data, `y[k]`. At the command line, the `correct` command performs this update.

$$C[k] = \left.\frac{\partial h}{\partial x}\right|_{\hat{x}[k|k-1]}$$

$$S[k] = \left.\frac{\partial h}{\partial v}\right|_{\hat{x}[k|k-1]}$$

The software calculates these Jacobian matrices numerically unless you specify your own function handle for the Jacobian.

$$K[k] = P[k|k-1]C[^{k}(^{C}$$

$$\hat{x}[k|k] = \hat{x}[k|k-1] + K[k](y[k] - h(\hat{x}[k|k-1], 0, u_m[k])$$

$$\hat{P}[k|k] = P[k|k-1] - K[k]\,C[k]\,P[k|k-1]$$

Here `K` is the Kalman gain.

b Compute the Jacobian of the state transition function, and predict the state and state estimation error covariance at the next time step. In the software, the `predict` command performs this prediction.

$$A[k] = \left.\frac{\partial f}{\partial x}\right|_{\hat{x}[k|k]}$$

$$G[k] = \left.\frac{\partial f}{\partial w}\right|_{\hat{x}[k|k]}$$

The software calculates these Jacobian matrices numerically unless you specify the analytical Jacobian. This numerical computation may increase processing time and numerical inaccuracy of the state estimation.

$$P[k+1|k] = A[k]P[k|k]A[^{k} + G[k]Q[k]G[^{k}$$

$$\hat{x}[k+1|k] = f(\hat{x}[k|k], 0, u_s[k])$$

The `correct` function uses these values in the next time step. For better numerical performance, the software uses the square-root factorization of the covariance matrices. For more information on this factorization, see [2] .

The Extended Kalman Filter block supports multiple measurement functions. These measurements can have different sample times as long as their sample time is an integer multiple of the state transition sample time. In this case, a separate correction step is performed corresponding to measurements from each measurement function.

The algorithm steps described previously assume that you have non-additive noise terms in the state transition and measurement functions. If you have additive noise terms in the functions, the changes in the algorithm are:

- If the process noise `w` is additive, that is the state transition equation has the form $x[k] = f(x[k-1], u_s[k-1]) + w[k-1]$, then the Jacobian matrix `G[k]` is an identity matrix.
- If the measurement noise `v` is additive, that is the measurement equation has the form $y[k] = h(x[k], u_m[k]) + v[k]$, then the Jacobian matrix `S[k]` is an identity matrix.

Additive noise terms in the state and transition functions reduce the processing time.

The first-order extended Kalman filter uses linear approximations to nonlinear state transition and measurement functions. As a result, the algorithm may not be reliable if the nonlinearities in your system are severe. The unscented Kalman filter algorithm may yield better results in this case.

Unscented Kalman Filter Algorithm

The unscented Kalman filter algorithm and Unscented Kalman Filter block use the unscented transformation to capture the propagation of the statistical properties of state estimates through nonlinear functions. The algorithm first generates a set of state values called sigma points. These sigma points capture the mean and covariance of the state estimates. The algorithm uses each of the sigma points as an input to the state transition and measurement functions to get a new set of transformed state points. The mean and covariance of the transformed points is then used to obtain state estimates and state estimation error covariance.

Assume that the state transition and measurement equations for an M-state discrete-time nonlinear system have additive process and measurement noise terms with zero mean and covariances Q and R, respectively:

$$x[k+1] = f(x[k], u_s[k]) + w[k]$$
$$y[k] = h(x[k], u_m[k]) + v[k]$$
$$w[k] \sim (0, Q[k])$$
$$v[k] \sim (0, R[k])$$

You provide the initial values of Q and R in the `ProcessNoise` and `MeasurementNoise` properties of the unscented Kalman filter object.

In the block, the software decides the order of prediction and correction of state estimates. At the command line, you decide the order. For information about the order in which to execute these commands, see the `predict` and `correct` reference pages. Assuming that you implement the `correct` command before `predict`, the software implements the algorithm as follows:

1 Initialize the filter object with initial values of the state, `x[0]`, and state estimation error covariance, P.

$$\widehat{x}[0 \mid -1] = E(x[0])$$
$$P[0 \mid -1] = E(x[0] - \widehat{x}[0 \mid -1])(^{x}$$

Here $\widehat{x}$ is the state estimate and $\widehat{x}[k_a \mid k_b]$ denotes the state estimate at time step k_a using measurements at time steps `0,1,...,`k_b. So $\widehat{x}[0 \mid -1]$ is the best guess of the state value before you make any measurements. You specify this value when you construct the filter.

2 For each time step k, update the state and state estimation error covariance using the measured data, `y[k]`. In the software, the `correct` command performs this update.

a Choose the sigma points $\hat{x}^{(i)}[k \mid k-1]$ at time step k.

$$\hat{x}^{(0)}[k \mid k-1] = \hat{x}[k \mid k-1]$$

$$\hat{x}^{(i)}[k \mid k-1] = \hat{x}[k \mid k-1] + \Delta x^{(i)} \qquad i = 1, \ldots, 2M$$

$$\Delta x^{(i)} = (\sqrt{cP[k \mid k-1]})_i \quad i = 1, \ldots, M$$

$$\Delta x^{(M+i)} = -(\sqrt{cP[k \mid k-1]})_i \quad i = 1, \ldots, M$$

Where $c = \alpha^2(M + \kappa)$ is a scaling factor based on number of states M, and the parameters α and κ. For more information about the parameters, see “Effect of Alpha, Beta, and Kappa Parameters” on page 13-7. $\sqrt{cP}$ is the matrix square root of cP such that $\sqrt{cP}\,(\sqrt{cP})^T = cP$ and $(\sqrt{cP})_i$ is the ith column of $\sqrt{cP}$.

b Use the nonlinear measurement function to compute the predicted measurements for each of the sigma points.

$$\hat{y}^{(i)}[k \mid k-1] = h(\hat{x}^{(i)}[k \mid k-1], u_m[\mathrm{k}]) \qquad i = 0, 1, \ldots, 2M$$

c Combine the predicted measurements to obtain the predicted measurement at time k.

$$\hat{y}[k] = \sum_{i=0}^{2M} W_M^{(i)} \hat{y}^{(i)}[k \mid k-1]$$

$$W_M^{(0)} = 1 - \frac{M}{\alpha^2(M + \kappa)}$$

$$W_M^{i} = \frac{1}{2\alpha^2(M + \kappa)} \qquad i = 1, 2, \ldots, 2M$$

d Estimate the covariance of the predicted measurement. Add `R[k]` to account for the additive measurement noise.

$$P_y = \sum_{i=0}^{2M} W_c^{(i)} (\hat{y}^{(i)}[k \mid k-1] - \hat{y}[k])(\hat{y}^{(i)} + R[k]$$

$$W_c^{(0)} = (2 - \alpha^2 + \beta) - \frac{M}{\alpha^2(M + \kappa)}$$

$$W_c^{i} = 1/(2\alpha^2(M + \kappa)) \qquad i = 1, 2, \ldots, 2M$$

For information about β parameter, see “Effect of Alpha, Beta, and Kappa Parameters” on page 13-7.

e Estimate the cross-covariance between $\hat{x}[k \mid k-1]$ and $\hat{y}[k]$.

$$P_{xy} = \frac{1}{2\alpha^2(m + \kappa)} \sum_{i=1}^{2M} (\hat{x}^{(i)}[k \mid k-1] - \hat{x}[k \mid k-1])(\hat{y}^{(i)}[k \mid k-1] - \hat{y}[k])^T$$

The summation starts from $i = 1$ because $\hat{x}^{(0)}[k \mid k-1] - \hat{x}[k \mid k-1] = 0$.

f Obtain the estimated state and state estimation error covariance at time step k.

$$K = P_{xy}P_y^{-1}$$

$$\hat{x}[k|k] = \hat{x}[k|k-1] + K(y[k] - \hat{y}[k])$$

$$P[k|k] = P[k|k-1] - KP_yK_k^T$$

Here K is the Kalman gain.

3 Predict the state and state estimation error covariance at the next time step. In the software, the `predict` command performs this prediction.

a Choose the sigma points $\hat{x}^{(i)}[k|k]$ at time step k.

$$\hat{x}^{(0)}[k|k] = \hat{x}[k|k]$$

$$\hat{x}^{(i)}[k|k] = \hat{x}[k|k] + \Delta x^{(i)} \qquad i = 1, \ldots, 2M$$

$$\Delta x^{(i)} = (\sqrt{cP[k|k]})_i \quad i = 1, \ldots, M$$

$$\Delta x^{(M+i)} = -(\sqrt{cP[k|k]})_i \quad i = 1, \ldots, M$$

b Use the nonlinear state transition function to compute the predicted states for each of the sigma points.

$$\hat{x}^{(i)}[k+1|k] = f(\hat{x}^{(i)}[k|k], u_s[k])$$

c Combine the predicted states to obtain the predicted states at time $k+1$. These values are used by the `correct` command in the next time step.

$$\hat{x}[k+1|k] = \sum_{i=0}^{2M} W_M^{(i)}\hat{x}^{(i)}[k+1|k]$$

$$W_M^{(0)} = 1 - \frac{M}{\alpha^2(M+\kappa)}$$

$$W_M^i = \frac{1}{2\alpha^2(M+\kappa)} \qquad i = 1, 2, \ldots, 2M$$

d Compute the covariance of the predicted state. Add `Q[k]` to account for the additive process noise. These values are used by the `correct` command in the next time step.

$$P[k+1|k] = \sum_{i=0}^{2M} W_c^{(i)}(\hat{x}^{(i)}[k+1|k] - \hat{x}[k+1|k])(\hat{x}^{(i)}[k+1|k] - \hat{x}[k+1|k])^T + Q[k]$$

$$W_c^{(0)} = (2 - \alpha^2 + \beta) - \frac{M}{\alpha^2(M+\kappa)}$$

$$W_c^i = 1/(2\alpha^2(M+\kappa)) \quad i = 1, 2, \ldots, 2M$$

The Unscented Kalman Filter block supports multiple measurement functions. These measurements can have different sample times as long as their sample time is an integer multiple of the state transition sample time. In this case, a separate correction step is performed corresponding to measurements from each measurement function.

The previous algorithm is implemented assuming additive noise terms in the state transition and measurement equations. For better numerical performance, the software uses the square-root factorization of the covariance matrices. For more information on this factorization, see [2] .

If the noise terms are non-additive, the main changes to the algorithm are:

- The `correct` command generates `2*(M+V)+1` sigma points using `P[k|k-1]` and `R[k]`, where `V` is the number of elements in measurement noise `v[k]`. The `R[k]` term is no longer added in the algorithm step 2(d) because the extra sigma points capture the impact of measurement noise on `Py`.
- The `predict` command generates `2*(M+W)+1` sigma points using `P[k|k]` and `Q[k]`, where `W` is the number of elements in process noise `w[k]`. The `Q[k]` term is no longer added in the algorithm step 3(d) because the extra sigma points capture the impact of process noise on `P[k+1|k]`.

Effect of Alpha, Beta, and Kappa Parameters

To compute the state and its statistical properties at the next time step, the unscented Kalman filter algorithm generates a set of state values distributed around the mean state value. The algorithm uses each sigma points as an input to the state transition and measurement functions to get a new set of transformed state points. The mean and covariance of the transformed points is then used to obtain state estimates and state estimation error covariance.

The spread of the sigma points around the mean state value is controlled by two parameters α and κ. A third parameter, β, impacts the weights of the transformed points during state and measurement covariance calculations.

- α — Determines the spread of the sigma points around the mean state value. It is usually a small positive value. The spread of sigma points is proportional to α. Smaller values correspond to sigma points closer to the mean state.
- κ — A second scaling parameter that is usually set to 0. Smaller values correspond to sigma points closer to the mean state. The spread is proportional to the square-root of κ.
- β — Incorporates prior knowledge of the distribution of the state. For Gaussian distributions, $\beta = 2$ is optimal.

You specify these parameters in the `Alpha`, `Kappa`, and `Beta` properties of the unscented Kalman filter. If you know the distribution of state and state covariance, you can adjust these parameters to capture the transformation of higher-order moments of the distribution. The algorithm can track only a single peak in the probability distribution of the state. If there are multiple peaks in the state distribution of your system, you can adjust these parameters so that the sigma points stay around a single peak. For example, choose a small `Alpha` to generate sigma points close to the mean state value.

References

[1] Simon, Dan. *Optimal State Estimation: Kalman, H Infinity, and Nonlinear Approaches*. Hoboken, NJ: John Wiley and Sons, 2006.

[2] Van der Merwe, Rudolph, and Eric A. Wan. “The Square-Root Unscented Kalman Filter for State and Parameter-Estimation.” *2001 IEEE International Conference on Acoustics, Speech, and Signal Processing. Proceedings* (Cat. No.01CH37221), 6:3461–64. Salt Lake City, UT, USA: IEEE, 2001. https://doi.org/10.1109/ICASSP.2001.940586.

See Also

Functions

`extendedKalmanFilter` | `unscentedKalmanFilter`

Blocks
Extended Kalman Filter | Unscented Kalman Filter

External Websites

- Understanding Kalman Filters: Nonlinear State Estimators — MATLAB Video Series

Generate Code for Online State Estimation in MATLAB

You can generate C/C++ code from MATLAB code that uses `extendedKalmanFilter`, `unscentedKalmanFilter` and `particleFilter` objects for online state estimation. C/C++ code is generated using the `codegen` command from MATLAB Coder software. Use the generated code to deploy online estimation algorithms to an embedded target. You can also deploy online estimation code by creating a standalone application using MATLAB Compiler™ software.

To generate C/C++ code for online state estimation:

1. Create a function to declare your filter object as persistent, and initialize the object. You define the object as persistent to maintain the object states between calls.

   ```
   function [CorrectedX] = ukfcodegen(output)
   % Declare object as persistent.
   persistent obj;
   if isempty(obj)
   % Initialize the object.
   obj = unscentedKalmanFilter(@vdpStateFcn,@vdpMeasurementFcn,[2;0]);
   obj.MeasurementNoise = 0.01;
   end
   % Estimate the states.
   CorrectedX = correct(obj,output);
   predict(obj);
   end
   ```

   The function creates an unscented Kalman filter object for online state estimation of a van der Pol oscillator with two states and one output. You use the previously written and saved state transition and measurement functions, `vdpStateFcn.m` and `vdpMeasurementFcn.m`, and specify the initial state values for the two states as `[2;0]`. Here `output` is the measured output data. Save the `ukfcodegen.m` function on the MATLAB path. Alternatively, you can specify the full path name for this function.

   In the `ukfcodegen.m` function, the persistent object is initialized with condition `if isempty(obj)` to ensure that the object is initialized only once, when the function is called the first time. Subsequent calls to the function only execute the `predict` and `correct` commands to update the state estimates. During initialization, you specify the nontunable properties of the object, such as `StateTransitionFcn` (specified in `ukfcodegen.m` as `vdpStateFcn.m`) and `MeasurementFcn` (specified in `ukfcodegen.m` as `vdpMeasurementFcn.m`). After that, you can specify only the tunable properties. For more information, see “Tunable and Nontunable Object Properties” on page 13-10.

   In the state transition and measurement functions you must use only commands that are supported for code generation. For a list of these commands, see “Functions and Objects Supported for C/C++ Code Generation” (MATLAB Coder). Include the compilation directive `%#codegen` in these functions to indicate that you intend to generate code for the function. Adding this directive instructs the MATLAB Code Analyzer to help you diagnose and fix violations that would result in errors during code generation. For an example, type `vdpStateFcn.m` at the command line.
2. Generate C/C++ code and MEX-files using the `codegen` command from MATLAB Coder software.

   ```
   codegen ukfcodegen -args {1}
   ```

The syntax `-args {1}` specifies an example of an argument to your function. The argument sets the dimensions and data types of the function argument `output` as a double-precision scalar.

Note If you want a filter with single-precision floating-point variables, you must specify the initial value of the states as single-precision during object construction.

```
obj = unscentedKalmanFilter(@vdpStateFcn,@vdpMeasurementFcn,single([2;0]))
```

Then to generate code, use the following syntax.

```
codegen ukfcodegen -args {{single(1)}
```

3 Use the generated code.

- Use the generated C/C++ code to deploy online state estimation to an embedded target.
- Use the generated MEX-file for testing the compiled C/C++ code in MATLAB. The generated MEX-file is also useful for accelerating simulations of state estimation algorithms in MATLAB.

  Load the estimation data. Suppose that your output data is stored in the `measured_data.mat` file.

  ```
  load measured_data.mat output
  ```

  Estimate the states by calling the generated MEX-file.

  ```
  for i = 1:numel(output)
      XCorrected = ukfcodegen_mex(output(i));
  end
  ```

This example generates C/C++ code for compiling a MEX-file. To generate code for other targets, see `codegen` in the MATLAB Coder documentation.

Tunable and Nontunable Object Properties

Property Type	Extended Kalman Filter Object	Unscented Kalman Filter Object	Particle Filter Object
Tunable properties that you can specify multiple times either during object construction, or afterward using dot notation	`State`, `StateCovariance`, `ProcessNoise`, and `MeasurementNoise`	`State`, `StateCovariance`, `ProcessNoise`, `MeasurementNoise`, `Alpha`, `Beta`, and `Kappa`	`Particles` and `Weights`
Nontunable properties that you can specify only once, either during object construction, or afterward using dot notation, but before using the `predict` or `correct` commands	`StateTransitionFcn`, `MeasurementFcn`, `StateTransitionJacobianFcn`, and `MeasurementJacobianFcn`	`StateTransitionFcn` and `MeasurementFcn`	`StateTransitionFcn`, `MeasurementLikelihoodFcn`, `StateEstimationMethod`, `StateOrientation`, `ResamplingPolicy` and `ResamplingMethod`

Property Type	Extended Kalman Filter Object	Unscented Kalman Filter Object	Particle Filter Object
Nontunable properties that you must specify during object construction	`HasAdditiveProcessNoise` and `HasAdditiveMeasurementNoise`	`HasAdditiveProcessNoise` and `HasAdditiveMeasurementNoise`	

See Also

`extendedKalmanFilter` | `particleFilter` | `unscentedKalmanFilter`

More About

- “Nonlinear State Estimation Using Unscented Kalman Filter and Particle Filter” on page 13-18
- “Validate Online State Estimation at the Command Line” on page 13-12
- “Troubleshoot Online State Estimation” on page 13-17

Validate Online State Estimation at the Command Line

After you use the `extendedKalmanFilter`, `unscentedKalmanFilter` or `particleFilter` commands for online state estimation of a nonlinear system, validate the estimation before deploying the code in your application. If the validation indicates low confidence in the estimation, then see "Troubleshoot Online State Estimation" on page 13-17 for next steps. After you have validated the online estimation results, you can generate C/C++ code or a standalone application using MATLAB Coder or MATLAB Compiler software.

To validate the performance of your filter, perform state estimation using measured or simulated output data from different scenarios.

- Obtain output data from your system at different operating conditions and input values — To ensure that estimation works well under all operating conditions of interest. For example, suppose that you want to track the position and velocity of a vehicle using noisy position measurements. Measure the data at different vehicle speeds and slow and sharp maneuvers.
- For each operating condition of interest, obtain multiple sets of experimental or simulated data with different noise realizations — To ensure different noise values do not deteriorate estimation performance.

For each of these scenarios, test the filter performance by examining the output estimation error and state estimation error. For an example about performing and validating online state estimation, see "Nonlinear State Estimation Using Unscented Kalman Filter and Particle Filter" on page 13-18.

Examine Output Estimation Error

The output estimation error is the difference between the measured output, `y`, and the estimated output, `yEstimated`. You can obtain the estimated output at each time step by using the measurement function of the system. For example, if `vdpMeasurementFcn.m` is the measurement function for your nonlinear system, and you are performing state estimation using an extended Kalman filter object, `obj`, you can compute the estimated output using the current state estimates as:

```
yEstimated = vdpMeasurementFcn(obj.State);
estimationError = y-yEstimated;
```

Here `obj.State` is the state value $\hat{x}[k|k-1]$ after you estimate the states using the `predict` command. $\hat{x}[k|k-1]$ is the predicted state estimate for time `k`, estimated using measured output until a previous time `k-1`.

If you are using `extendedKalmanFilter` or `unscentedKalmanFilter`, you can also use `residual` to get the estimation error:

```
[residual,residualCovariance] = residual(obj,y);
```

The estimation errors (residuals) must have the following characteristics:

- Small magnitude — Small errors relative to the size of the outputs increase confidence in the estimated values.
- Zero mean
- Low autocorrelation, except at zero time lag — To compute the autocorrelation, you can use the MATLAB `xcorr` command.

Examine State Estimation Error for Simulated Data

When you simulate the output data of your nonlinear system and use that data for state estimation, you know the true state values. You can compute the errors between estimated and true state values and analyze the errors. The estimated state value at any time step is the value stored in `obj.State` after you estimate the states using the `predict` or `correct` command. The state estimation errors must satisfy the following characteristics:

- Small magnitude
- Zero mean
- Low autocorrelation, except at zero time lag

You can also compute the covariance of the state estimation error and compare it to the state estimation error covariance stored in the `StateCovariance` property of the filter. Similar values increase confidence in the performance of the filter.

See Also

`extendedKalmanFilter` | `unscentedKalmanFilter` | `particleFilter` | `residual`

More About

- “Nonlinear State Estimation Using Unscented Kalman Filter and Particle Filter” on page 13-18
- “Extended and Unscented Kalman Filter Algorithms for Online State Estimation” on page 13-2
- “Troubleshoot Online State Estimation” on page 13-17
- “Generate Code for Online State Estimation in MATLAB” on page 13-9

Validate Online State Estimation in Simulink

After you use the Extended Kalman Filter, Unscented Kalman Filter or Particle Filter blocks for online state estimation of a nonlinear system, validate the estimation before deploying the code in your application. If the validation indicates low confidence in the estimation, then see “Troubleshoot Online State Estimation” on page 13-17 for next steps. After you have validated the online estimation results, you can generate C/C++ code for the blocks using Simulink Coder software.

To validate the performance of your filter, perform state estimation using measured or simulated output data from these scenarios.

- Obtain output data from your system at different operating conditions and input values — To ensure that estimation works well under all operating conditions of interest. For example, suppose that you want to track the position and velocity of a vehicle using noisy position measurements. Measure the data at different vehicle speeds and slow and sharp maneuvers.
- For each operating condition of interest, obtain multiple sets of experimental or simulated data with different noise realizations — To ensure that different noise values do not deteriorate estimation performance.

For each of these scenarios, test the filter performance by examining the residuals and state estimation error.

Examine Residuals

The residual, or output estimation error, is the difference between the measured system output `yMeasured[k]`, and the estimated system output `yPredicted[k|k-1]` at time step `k`. Here, `yPredicted[k|k-1]` is the estimated output at time step `k`, which is predicted using output measurements until time step `k-1`.

The blocks do not explicitly output `yPredicted[k|k-1]`, however you can compute the output using the estimated state values and your state transition and measurement functions. For an example, see “Compute Residuals and State Estimation Errors” on page 13-15.

The residuals must have the following characteristics:

- Small magnitude — Small errors relative to the size of the outputs increase confidence in the estimated values.
- Zero mean
- Low autocorrelation, except at zero time lag — To compute the autocorrelation, you can use the Autocorrelation block from DSP System Toolbox™ software.

Examine State Estimation Error for Simulated Data

When you simulate the output data of your nonlinear system and use that data for state estimation, you know the true state values. You can compute the errors between estimated and true state values and analyze the errors. The estimated state value at any time step is output at the **xhat** port of the blocks. The state estimation errors must satisfy the following characteristics:

- Small magnitude
- Zero mean

- Low autocorrelation, except at zero time lag

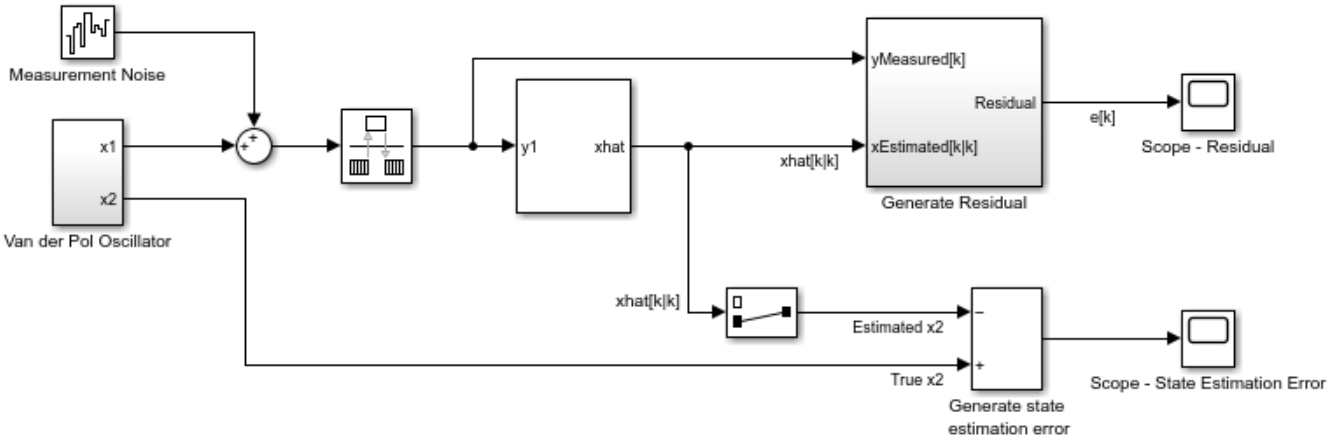
You can also compute the covariance of the state estimation error, and compare it to the state estimation error covariance that is output by the blocks in the **P** port of the blocks. Similar values increase confidence in the performance of the filter.

Compute Residuals and State Estimation Errors

This example shows how to estimate the states of a discrete-time Van der Pol oscillator and compute state estimation errors and residuals for validating the estimation. The residuals are the output estimation errors, that is, they are the difference between the measured and estimated outputs.

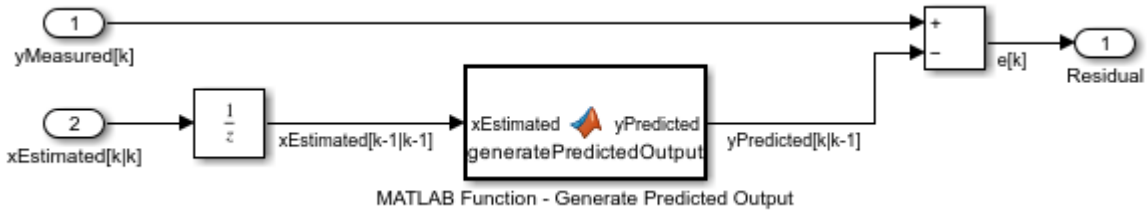
In the Simulink™ model `vdpStateEstimModel`, the Van der Pol Oscillator block implements the oscillator with nonlinearity parameter, mu, equal to 1. The oscillator has two states. A noisy measurement of the first state `x1` is available.

The model uses the Unscented Kalman Filter block to estimate the states of the oscillator. Since the block requires discrete-time inputs, the Rate Transition block samples `x1` to give the discretized output measurement `yMeasured[k]` at time step `k`. The Unscented Kalman Filter block outputs the estimated state values `xhat[k|k]` at time step `k`, using `yMeasured` until time `k`. The filter block uses the previously written and saved state transition and measurement functions, `vdpStateFcn.m` and `vdpMeasurementFcn.m`. For information about these functions, see “Nonlinear State Estimation Using Unscented Kalman Filter and Particle Filter” on page 13-18.



To validate the state estimation, the model computes the residuals in the Generate Residual block. In addition, since the true state values are known, the model also computes the state estimation errors.

To compute the residuals, the Generate Residual block first computes the estimated output `yPredicted[k|k-1]` using the estimated states and state transition and measurement functions. Here, `yPredicted[k|k-1]` is the estimated output at time step `k`, predicted using output measurements until time step `k-1`. The block then computes the residual at time step `k` as `yMeasured[k] - yPredicted[k|k-1]`.



Examine the residuals and state estimation errors, and ensure that they have a small magnitude, zero mean, and low autocorrelation.

In this example, the Unscented Kalman Filter block outputs `xhat[k|k]` because the **Use the current measurements to improve state estimates** parameter of the block is selected. If you clear this parameter, the block instead outputs `xhat[k|k-1]`, the predicted state value at time step `k`, using `yMeasured` until time `k-1`. In this case, compute `yPredicted[k|k-1] = MeasurementFcn(xhat[k|k-1])`, where `MeasurementFcn` is the measurement function for your system.

See Also

Kalman Filter | Extended Kalman Filter | Unscented Kalman Filter | Particle Filter

More About

- “Extended and Unscented Kalman Filter Algorithms for Online State Estimation” on page 13-2
- “Troubleshoot Online State Estimation” on page 13-17

Troubleshoot Online State Estimation

After you perform state estimation of a nonlinear system using linear, extended, or unscented Kalman filter or particle filter algorithms, you validate the estimation before deploying the code in your application. If the validation indicates low confidence in the estimation, check the following filter properties that you specified:

- Initial state and state covariance values — If you find that the measured and estimated outputs of your system are diverging at the beginning of state estimation, check the initial values that you specified.
- State transition and measurement functions — Verify that the functions you specify are a good representation of the nonlinear system. If the true system is continuous-time, to implement the algorithms, you discretize the state transition and measurement equations and use the discretized versions. If the state estimation results are not satisfactory, consider decreasing the sample time used for discretization. Alternatively, try a different discretization method. For an example of how to discretize a continuous-time state transition function, type `edit vdpStateFcn.m` at the command line. Also see, “Nonlinear State Estimation Using Unscented Kalman Filter and Particle Filter” on page 13-18.
- Process and measurement noise covariance values — If the difference between estimated and measured outputs of your system is large, try specifying different values for the process and measurement noise covariance values.
- Choice of algorithm — If you are using the extended Kalman filter algorithm, you can try the unscented Kalman filter, or the particle filter algorithm instead. The unscented Kalman filter and particle filter may capture the nonlinearities in the system better.

To troubleshoot state estimation, you can create multiple versions of the filter with different properties, perform state estimation, and choose the filter that gives the best validation results.

At the command line, if you want to copy an existing filter object and then modify properties of the copied object, use the `clone` command. Do not create additional objects using syntax `obj2 = obj`. Any changes made to the properties of the new object created in this way (`obj2`) also change the properties of the original object (`obj`).

See Also

Functions
`extendedKalmanFilter` | `unscentedKalmanFilter` | `particleFilter`

Blocks
Extended Kalman Filter | Unscented Kalman Filter | Particle Filter

More About

- “Validate Online State Estimation at the Command Line” on page 13-12
- “Generate Code for Online State Estimation in MATLAB” on page 13-9
- “Validate Online State Estimation in Simulink” on page 13-14

Nonlinear State Estimation Using Unscented Kalman Filter and Particle Filter

This example shows how to use the unscented Kalman filter and particle filter algorithms for nonlinear state estimation for the van der Pol oscillator.

This example also uses the Signal Processing Toolbox™.

Introduction

Control System Toolbox™ offers three commands for nonlinear state estimation:

- `extendedKalmanFilter`: First-order, discrete-time extended Kalman filter
- `unscentedKalmanFilter`: Discrete-time unscented Kalman filter
- `particleFilter`: Discrete-time particle filter

A typical workflow for using these commands is as follows:

1. Model your plant and sensor behavior.
2. Construct and configure the `extendedKalmanFilter`, `unscentedKalmanFilter` or `particleFilter` object.
3. Perform state estimation by using the `predict` and `correct` commands with the object.
4. Analyze results to gain confidence in filter performance
5. Deploy the filter on your hardware. You can generate code for these filters using MATLAB Coder™.

This example first uses the `unscentedKalmanFilter` command to demonstrate this workflow. Then it demonstrates the use of particleFilter.

Plant Modeling and Discretization

The unscented Kalman filter (UKF) algorithm requires a function that describes the evolution of states from one time step to the next. This is typically called the state transition function. unscentedKalmanFilter supports the following two function forms:

1. Additive process noise: $x[k] = f(x[k-1], u[k-1]) + w[k-1]$
2. Non-additive process noise: $x[k] = f(x[k-1], w[k-1], u[k-1])$

Here **f**(..) is the state transition function, **x** is the state, w is the process noise. u is optional and represents additional inputs to **f**, for instance system inputs or parameters. **u** can be specified as zero or more function arguments. Additive noise means that the state and process noise is related linearly. If the relationship is nonlinear, use the second form. When you create the `unscentedKalmanFilter` object, you specify **f**(..) and also whether the process noise is additive or non-additive.

The system in this example is the van der Pol oscillator with mu=1. This 2-state system is described with the following set of nonlinear ordinary differential equations (ODE):

$$\dot{x}_1 = x_2$$

$$\dot{x}_2 = (1 - x_1^2)x_2 - x_1$$

Denote this equation as $\dot{x} = f_c(x)$, where $x = [x_1; x_2]$. The process noise w does not appear in the system model. You can assume it is additive for simplicity.

unscentedKalmanFilter requires a discrete-time state transition function, but the plant model is continuous-time. You can use a discrete-time approximation to the continuous-time model. Euler discretization is one common approximation method. Assume that your sample time is T_s, and denote the continuous-time dynamics you have as $\dot{x} = f_c(x)$. Euler discretization approximates the derivative operator as $\dot{x} \approx \frac{x[k+1] - x[k]}{T_s}$. The resulting discrete-time state-transition function is:

$$\begin{aligned} x[k+1] &= x[k] + f_c(x[k])\, T_s \\ &= f(x[k]) \end{aligned}$$

The accuracy of this approximation depends on the sample time T_s. Smaller T_s values provide better approximations. Alternatively, you can use a different discretization method. For instance, higher order Runge-Kutta family of methods provide a higher accuracy at the expense of more computational cost, given a fixed sample time T_s.

Create this state-transition function and save it in a file named vdpStateFcn.m. Use the sample time $T_s = 0.05s$. You provide this function to the `unscentedKalmanFilter` during object construction.

```
type vdpStateFcn
```

```
function x = vdpStateFcn(x)
% vdpStateFcn Discrete-time approximation to van der Pol ODEs for mu = 1.
% Sample time is 0.05s.
%
% Example state transition function for discrete-time nonlinear state
% estimators.
%
% xk1 = vdpStateFcn(xk)
%
% Inputs:
%    xk - States x[k]
%
% Outputs:
%    xk1 - Propagated states x[k+1]
%
% See also extendedKalmanFilter, unscentedKalmanFilter

%   Copyright 2016 The MathWorks, Inc.

%#codegen

% The tag %#codegen must be included if you wish to generate code with
% MATLAB Coder.

% Euler integration of continuous-time dynamics x'=f(x) with sample time dt
dt = 0.05; % [s] Sample time
x = x + vdpStateFcnContinuous(x)*dt;
end

function dxdt = vdpStateFcnContinuous(x)
%vdpStateFcnContinuous Evaluate the van der Pol ODEs for mu = 1
dxdt = [x(2); (1-x(1)^2)*x(2)-x(1)];
end
```

Sensor Modeling

`unscentedKalmanFilter` also needs a function that describes how the model states are related to sensor measurements. `unscentedKalmanFilter` supports the following two function forms:

1. Additive measurement noise: $y[k] = h(x[k], u[k]) + v[k]$
2. Non-additive measurement noise: $y[k] = h(x[k], v[k], u[k])$

h(..) is the measurement function, **v** is the measurement noise. **u** is optional and represents additional inputs to **h**, for instance system inputs or parameters. u can be specified as zero or more function arguments. You can add additional system inputs following the **u** term. These inputs can be different than the inputs in the state transition function.

For this example, assume you have measurements of the first state x_1 within some percentage error:

$$y[k] = x_1[k]\,(1 + v[k])$$

This falls into the category of non-additive measurement noise because the measurement noise is not simply added to a function of states. You want to estimate both x_1 and x_2 from the noisy measurements.

Create this state transition function and save it in a file named `vdpMeasurementNonAdditiveNoiseFcn.m`. You provide this function to the `unscentedKalmanFilter` during object construction.

```
type vdpMeasurementNonAdditiveNoiseFcn
```

```
function yk = vdpMeasurementNonAdditiveNoiseFcn(xk,vk)
% vdpMeasurementNonAdditiveNoiseFcn Example measurement function for discrete
% time nonlinear state estimators with non-additive measurement noise.
%
% yk = vdpNonAdditiveMeasurementFcn(xk,vk)
%
% Inputs:
%    xk - x[k], states at time k
%    vk - v[k], measurement noise at time k
%
% Outputs:
%    yk - y[k], measurements at time k
%
% The measurement is the first state with multiplicative noise
%
% See also extendedKalmanFilter, unscentedKalmanFilter

%   Copyright 2016 The MathWorks, Inc.

%#codegen

% The tag %#codegen must be included if you wish to generate code with
% MATLAB Coder.

yk = xk(1)*(1+vk);
end
```

Unscented Kalman Filter Construction

Construct the filter by providing function handles to the state transition and measurement functions, followed by your initial state guess. The state transition model has additive noise. This is the default

setting in the filter, hence you do not need to specify it. The measurement model has non-additive noise, which you must specify through setting the `HasAdditiveMeasurementNoise` property of the object as `false`. This must be done during object construction. If your application has non-additive process noise in the state transition function, specify the `HasAdditiveProcessNoise` property as `false`.

```
% Your initial state guess at time k, utilizing measurements up to time k-1: xhat[k|k-1]
initialStateGuess = [2;0]; % xhat[k|k-1]
% Construct the filter
ukf = unscentedKalmanFilter(...
    @vdpStateFcn,... % State transition function
    @vdpMeasurementNonAdditiveNoiseFcn,... % Measurement function
    initialStateGuess,...
    'HasAdditiveMeasurementNoise',false);
```

Provide your knowledge of the measurement noise covariance

```
R = 0.2; % Variance of the measurement noise v[k]
ukf.MeasurementNoise = R;
```

The `ProcessNoise` property stores the process noise covariance. It is set to account for model inaccuracies and the effect of unknown disturbances on the plant. We have the true model in this example, but discretization introduces errors. This example did not include any disturbances for simplicity. Set it to a diagonal matrix with less noise on the first state, and more on the second state to reflect the knowledge that the second state is more impacted by modeling errors.

```
ukf.ProcessNoise = diag([0.02 0.1]);
```

Estimation Using predict and correct Commands

In your application, the measurement data arriving from your hardware in real-time are processed by the filters as they arrive. This operation is demonstrated in this example by generating a set of measurement data first, and then feeding it to the filter one step at a time.

Simulate the van der Pol oscillator for 5 seconds with the filter sample time 0.05 [s] to generate the true states of the system.

```
T = 0.05; % [s] Filter sample time
timeVector = 0:T:5;
[~,xTrue]=ode45(@vdp1,timeVector,[2;0]);
```

Generate the measurements assuming that a sensor measures the first state, with a standard deviation of 45% error in each measurement.

```
rng(1); % Fix the random number generator for reproducible results
yTrue = xTrue(:,1);
yMeas = yTrue .* (1+sqrt(R)*randn(size(yTrue))); % sqrt(R): Standard deviation of noise
```

Pre-allocate space for data that you will analyze later

```
Nsteps = numel(yMeas); % Number of time steps
xCorrectedUKF = zeros(Nsteps,2); % Corrected state estimates
PCorrected = zeros(Nsteps,2,2); % Corrected state estimation error covariances
e = zeros(Nsteps,1); % Residuals (or innovations)
```

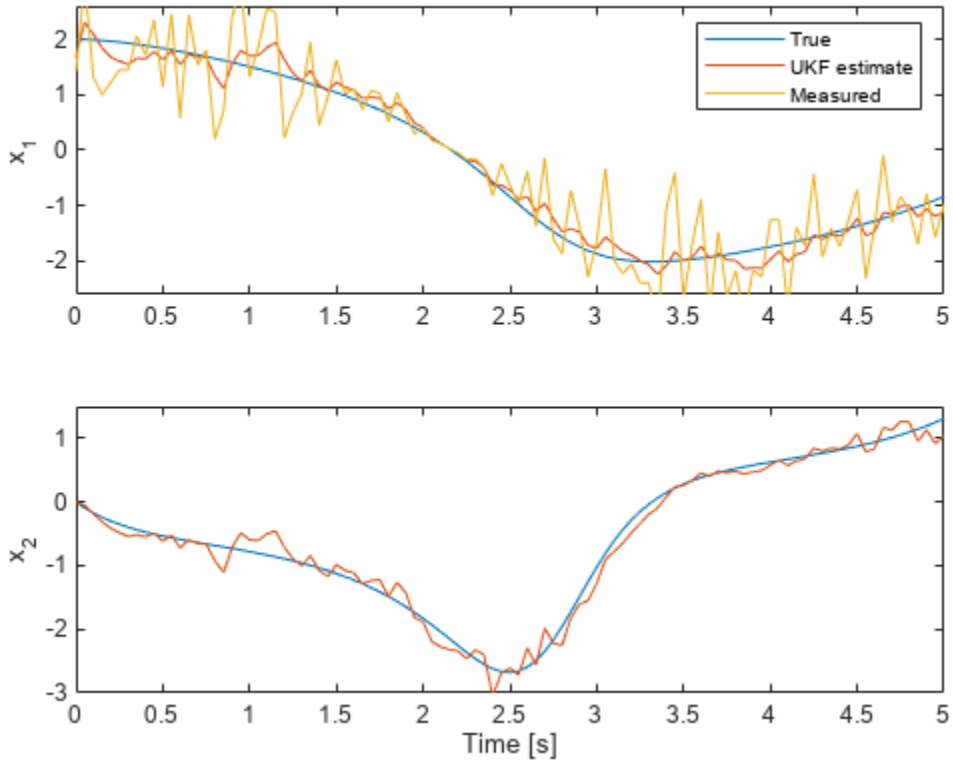
Perform online estimation of the states $\mathbf{x}$ using the `correct` and `predict` commands. Provide generated data to the filter one time step at a time.

```
for k=1:Nsteps
    % Let k denote the current time.
    %
    % Residuals (or innovations): Measured output - Predicted output
    e(k) = yMeas(k) - vdpMeasurementFcn(ukf.State); % ukf.State is x[k|k-1] at this point
    % Incorporate the measurements at time k into the state estimates by
    % using the "correct" command. This updates the State and StateCovariance
    % properties of the filter to contain x[k|k] and P[k|k]. These values
    % are also produced as the output of the "correct" command.
    [xCorrectedUKF(k,:), PCorrected(k,:,:)] = correct(ukf,yMeas(k));
    % Predict the states at next time step, k+1. This updates the State and
    % StateCovariance properties of the filter to contain x[k+1|k] and
    % P[k+1|k]. These will be utilized by the filter at the next time step.
    predict(ukf);
end
```

Unscented Kalman Filter Results and Validation

Plot the true and estimated states over time. Also plot the measured value of the first state.

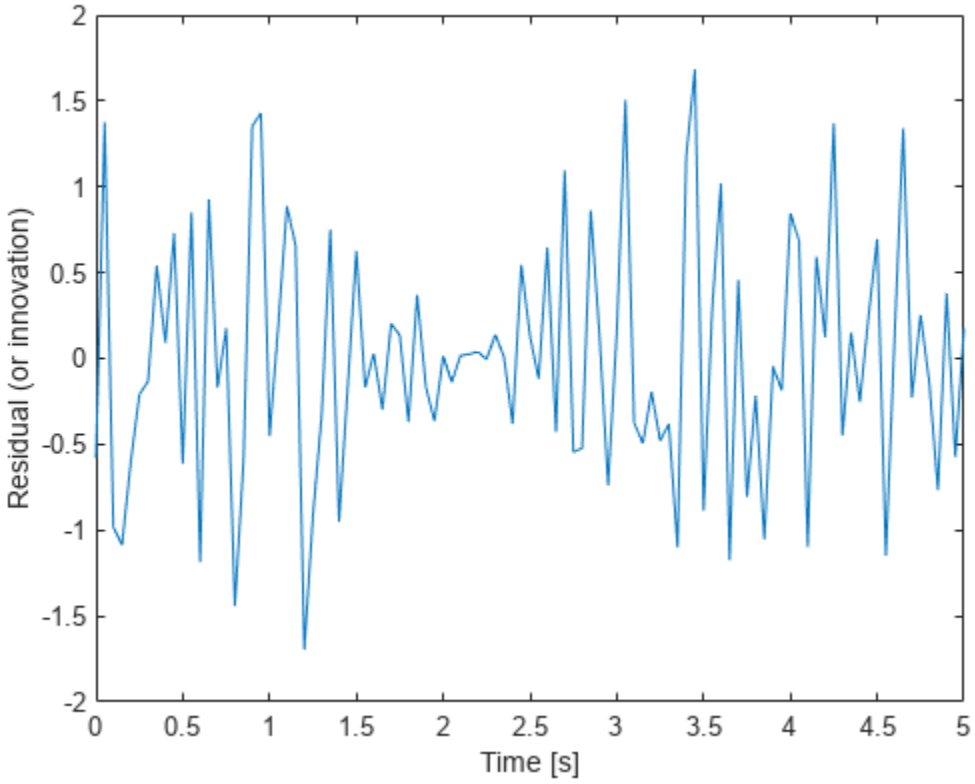
```
figure();
subplot(2,1,1);
plot(timeVector,xTrue(:,1),timeVector,xCorrectedUKF(:,1),timeVector,yMeas(:));
legend('True','UKF estimate','Measured')
ylim([-2.6 2.6]);
ylabel('x_1');
subplot(2,1,2);
plot(timeVector,xTrue(:,2),timeVector,xCorrectedUKF(:,2));
ylim([-3 1.5]);
xlabel('Time [s]');
ylabel('x_2');
```



The top plot shows the true, estimated, and the measured value of the first state. The filter utilizes the system model and noise covariance information to produce an improved estimate over the measurements. The bottom plot shows the second state. The filter is is successful in producing a good estimate.

The validation of unscented and extended Kalman filter performance is typically done using extensive Monte Carlo simulations. These simulations should test variations of process and measurement noise realizations, plant operating under various conditions, initial state and state covariance guesses. The key signal of interest used for validating the state estimation is the residuals (or innovations). In this example, you perform residual analysis for a single simulation. Plot the residuals.

```
figure();
plot(timeVector, e);
xlabel('Time [s]');
ylabel('Residual (or innovation)');
```



The residuals should have:

1. Small magnitude
2. Zero mean
3. No autocorrelation, except at zero lag

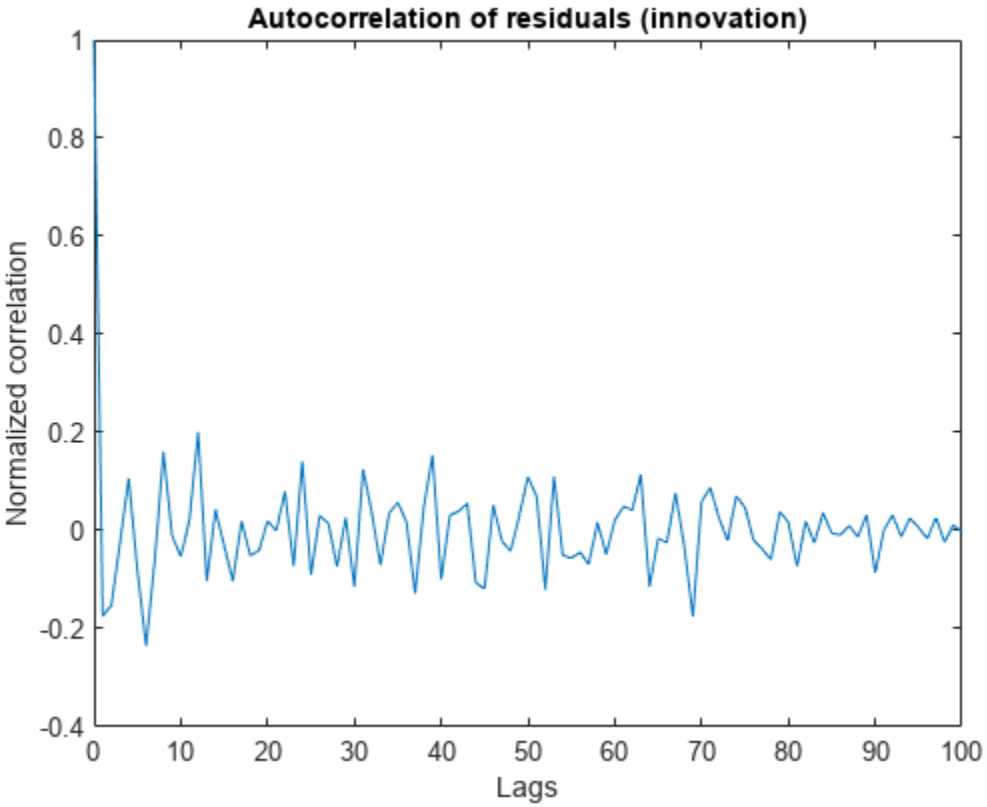
The mean value of the residuals is:

```
mean(e)
```

```
ans = -0.0012
```

This is small relative to the magnitude of the residuals. The autocorrelation of the residuals can be calculated with the xcorr function in the Signal Processing Toolbox.

```
[xe,xeLags] = xcorr(e,'coeff'); % 'coeff': normalize by the value at zero lag
% Only plot non-negative lags
idx = xeLags>=0;
figure();
plot(xeLags(idx),xe(idx));
xlabel('Lags');
ylabel('Normalized correlation');
title('Autocorrelation of residuals (innovation)');
```



The correlation is small for all lags except 0. The mean correlation is close to zero, and the correlation does not show any significant non-random variations. These characteristics increase confidence in filter performance.

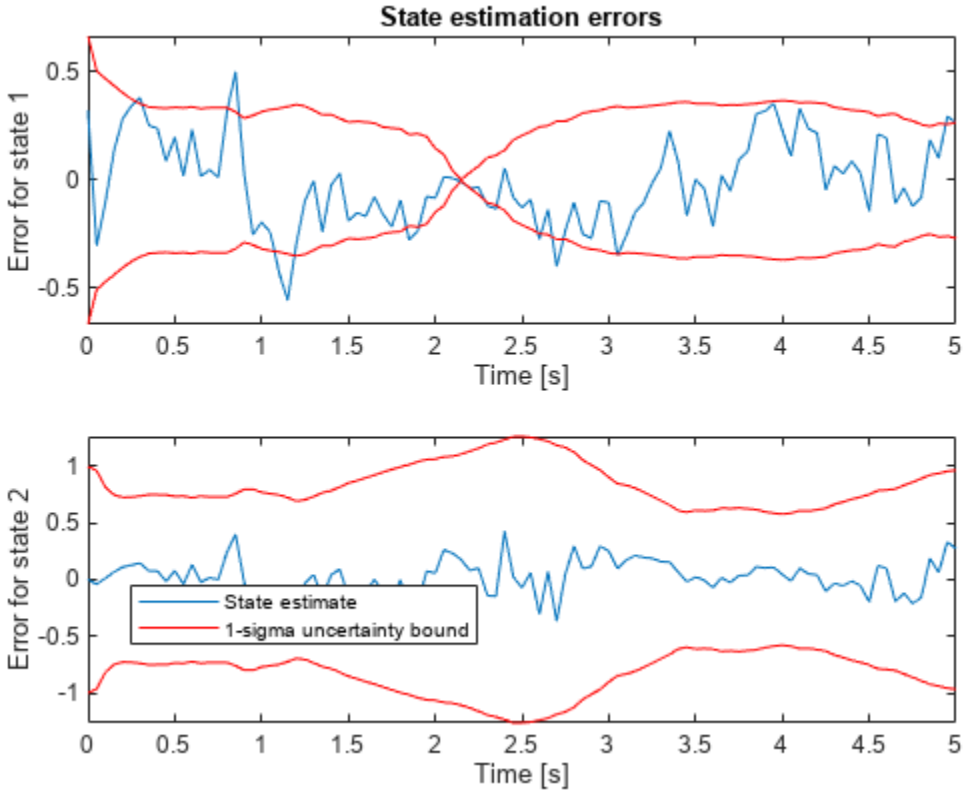
In reality the true states are never available. However, when performing simulations, you have access to real states and can look at the errors between estimated and true states. These errors must satisfy similar criteria to the residual:

1. Small magnitude
2. Variance within filter error covariance estimate
3. Zero mean
4. Uncorrelated.

First, look at the error and the 1σ uncertainty bounds from the filter error covariance estimate.

```
eStates = xTrue-xCorrectedUKF;
figure();
subplot(2,1,1);
plot(timeVector,eStates(:,1),...                % Error for the first state
    timeVector, sqrt(PCorrected(:,1,1)),'r', ... % 1-sigma upper-bound
    timeVector, -sqrt(PCorrected(:,1,1)),'r');   % 1-sigma lower-bound
xlabel('Time [s]');
ylabel('Error for state 1');
title('State estimation errors');
subplot(2,1,2);
plot(timeVector,eStates(:,2),...                % Error for the second state
```

```
    timeVector,sqrt(PCorrected(:,2,2)),'r', ...  % 1-sigma upper-bound
    timeVector,-sqrt(PCorrected(:,2,2)),'r');    % 1-sigma lower-bound
xlabel('Time [s]');
ylabel('Error for state 2');
legend('State estimate','1-sigma uncertainty bound',...
    'Location','Best');
```



The error bound for state 1 approaches 0 at t=2.15 seconds because of the sensor model (`MeasurementFcn`). It has the form $x_1[k]*(1 + v[k])$. At t=2.15 seconds the true and estimated states are near zero, which implies the measurement error in absolute terms is also near zero. This is reflected in the state estimation error covariance of the filter.

Calculate what percentage of the points are beyond the 1-sigma uncertainty bound.

```
distanceFromBound1 = abs(eStates(:,1))-sqrt(PCorrected(:,1,1));
percentageExceeded1 = nnz(distanceFromBound1>0) / numel(eStates(:,1));
distanceFromBound2 = abs(eStates(:,2))-sqrt(PCorrected(:,2,2));
percentageExceeded2 = nnz(distanceFromBound2>0) / numel(eStates(:,2));
[percentageExceeded1 percentageExceeded2]
```

```
ans = 1×2

    0.1386         0
```

The first state estimation errors exceed the 1σ uncertainty bound approximately 14% of the time steps. Less than 30% of the errors exceeding the 1-sigma uncertainty bound implies good estimation.

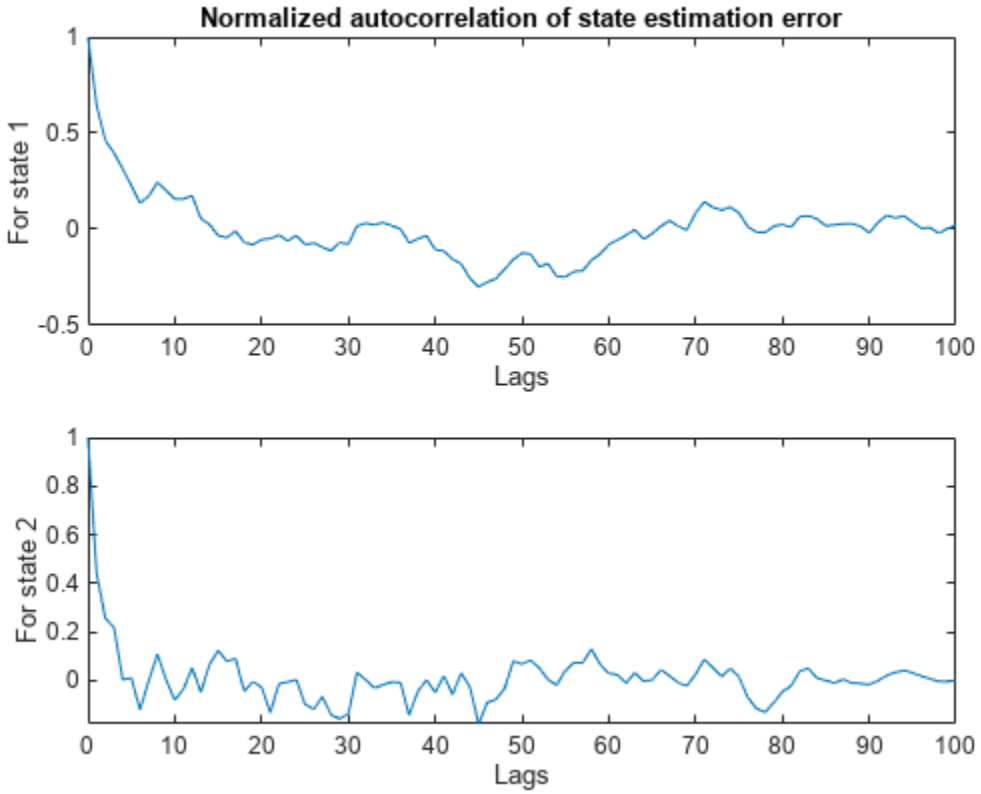
This criterion is satisfied for both states. The 0% percentage for the second state means that the filter is conservative: most likely the combined process and measurement noises are too high. Likely a better performance can be obtained by tuning these parameters.

Calculate the mean autocorrelation of state estimation errors. Also plot the autocorrelation.

```
mean(eStates)
```

```
ans = 1×2

   -0.0103    0.0201
```

```
[xeStates1,xeStatesLags1] = xcorr(eStates(:,1),'coeff'); % 'coeff': normalize by the value at ze
[xeStates2,xeStatesLags2] = xcorr(eStates(:,2),'coeff'); % 'coeff'
% Only plot non-negative lags
idx = xeStatesLags1>=0;
figure();
subplot(2,1,1);
plot(xeStatesLags1(idx),xeStates1(idx));
xlabel('Lags');
ylabel('For state 1');
title('Normalized autocorrelation of state estimation error');
subplot(2,1,2);
plot(xeStatesLags2(idx),xeStates2(idx));
xlabel('Lags');
ylabel('For state 2');
```



The mean value of the errors is small relative to the value of the states. The autocorrelation of state estimation errors shows little non-random variations for small lag values, but these are much smaller than the normalized peak value 1. Combined with the fact that the estimated states are accurate, this behavior of the residuals can be considered as satisfactory results.

Particle Filter Construction

Unscented and extended Kalman filters aim to track the mean and covariance of the posterior distribution of the state estimates by different approximation methods. These methods may not be sufficient if the nonlinearities in the system are severe. In addition, for some applications, just tracking the mean and covariance of the posterior distribution of the state estimates may not be sufficient. Particle filter can address these problems by tracking the evolution of many state hypotheses (particles) over time, at the expense of higher computational cost. The computational cost and estimation accuracy increases with the number of particles.

The `particleFilter` command in Control System Toolbox implements a discrete-time particle filter algorithm. This section walks you through constructing a `particleFilter` for the same van der Pol oscillator used earlier in this example, and highlights the similarities and differences with the unscented Kalman filter.

The state transition function you provide to `particleFilter` must perform two tasks. One, sampling the process noise from any distribution appropriate for your system. Two, calculating the time propagation of all particles (state hypotheses) to the next step, including the effect of process noise you calculated in step one.

```
type vdpParticleFilterStateFcn
```

```
function particles = vdpParticleFilterStateFcn(particles)
% vdpParticleFilterStateFcn Example state transition function for particle
%                           filter
%
% Discrete-time approximation to van der Pol ODEs for mu = 1.
% Sample time is 0.05s.
%
% predictedParticles = vdpParticleFilterStateFcn(particles)
%
% Inputs:
%    particles - Particles at current time. Matrix with dimensions
%                [NumberOfStates NumberOfParticles] matrix
%
% Outputs:
%    predictedParticles - Predicted particles for the next time step
%
% See also particleFilter

%   Copyright 2017 The MathWorks, Inc.

%#codegen

% The tag %#codegen must be included if you wish to generate code with
% MATLAB Coder.

[numberOfStates, numberOfParticles] = size(particles);

% Time-propagate each particle
%
% Euler integration of continuous-time dynamics x'=f(x) with sample time dt
```

```
dt = 0.05; % [s] Sample time
for kk=1:numberOfParticles
    particles(:,kk) = particles(:,kk) + vdpStateFcnContinuous(particles(:,kk))*dt;
end

% Add Gaussian noise with variance 0.025 on each state variable
processNoise = 0.025*eye(numberOfStates);
particles = particles + processNoise * randn(size(particles));
end

function dxdt = vdpStateFcnContinuous(x)
%vdpStateFcnContinuous Evaluate the van der Pol ODEs for mu = 1
dxdt = [x(2); (1-x(1)^2)*x(2)-x(1)];
end
```

There are differences between the state transition function you supply to `unscentedKalmanFilter` and `particleFilter`. The state transition function you used for unscented Kalman filter just described propagation of one state hypothesis to the next time step, instead of a set of hypotheses. In addition, the process noise distribution was defined in the `ProcessNoise` property of the `unscentedKalmanFilter`, just by its covariance. `particleFilter` can consider arbitrary distributions that may require more statistical properties to be defined. This arbitrary distribution and its parameters are fully defined in the state transition function you provide to `particleFilter`.

The measurement likelihood function you provide to `particleFilter` must also perform two tasks. One, calculating measurement hypotheses from particles. Two, calculating the likelihood of each particle from the sensor measurement and the hypotheses calculated in step one.

```
type vdpExamplePFMeasurementLikelihoodFcn
```

```
function likelihood = vdpExamplePFMeasurementLikelihoodFcn(particles,measurement)
% vdpExamplePFMeasurementLikelihoodFcn Example measurement likelihood function
%
% The measurement is the first state.
%
% likelihood = vdpParticleFilterMeasurementLikelihoodFcn(particles, measurement)
%
% Inputs:
%    particles - NumberOfStates-by-NumberOfParticles matrix that holds
%                the particles
%
% Outputs:
%    likelihood - A vector with NumberOfParticles elements whose n-th
%                 element is the likelihood of the n-th particle
%
% See also extendedKalmanFilter, unscentedKalmanFilter

%   Copyright 2017 The MathWorks, Inc.

%#codegen

% The tag %#codegen must be included if you wish to generate code with
% MATLAB Coder.

% Validate the sensor measurement
numberOfMeasurements = 1; % Expected number of measurements
validateattributes(measurement, {'double'}, {'vector', 'numel', numberOfMeasurements}, ...
    'vdpExamplePFMeasurementLikelihoodFcn', 'measurement');
```

```
% The measurement is first state. Get all measurement hypotheses from particles
predictedMeasurement = particles(1,:);

% Assume the ratio of the error between predicted and actual measurements
% follow a Gaussian distribution with zero mean, variance 0.2
mu = 0; % mean
sigma = 0.2 * eye(numberOfMeasurements); % variance

% Use multivariate Gaussian probability density function, calculate
% likelihood of each particle
numParticles = size(particles,2);
likelihood = zeros(numParticles,1);
C = det(2*pi*sigma) ^ (-0.5);
for kk=1:numParticles
    errorRatio = (predictedMeasurement(kk)-measurement)/predictedMeasurement(kk);
    v = errorRatio-mu;
    likelihood(kk) = C * exp(-0.5 * (v' / sigma * v) );
end
end
```

Now construct the filter, and initialize it with 1000 particles around the mean [2; 0] with 0.01 covariance. The covariance is small because you have high confidence in your guess [2; 0].

```
pf = particleFilter(@vdpParticleFilterStateFcn,@vdpExamplePFMeasurementLikelihoodFcn);
initialize(pf, 1000, [2;0], 0.01*eye(2));
```

Optionally, pick the state estimation method. This is set by the `StateEstimationMethod` property of `particleFilter`, which can take the value `'mean'` (default) or `'maxweight'`. When `StateEstimationMethod` is `'mean'`, the object extracts a weighted mean of the particles from the `Particles` and `Weights` properties as the state estimate. `'maxweight'` corresponds to choosing the particle (state hypothesis) with the maximum weight value in `Weights` as the state estimate. Alternatively, you can access `Particles` and `Weights` properties of the object and extract your state estimate via an arbitrary method of your choice.

```
pf.StateEstimationMethod

ans =
'mean'
```

`particleFilter` lets you specify various resampling options via its `ResamplingPolicy` and `ResamplingMethod` properties. This example uses the default settings in the filter. See the `particleFilter` documentation for further details on resampling.

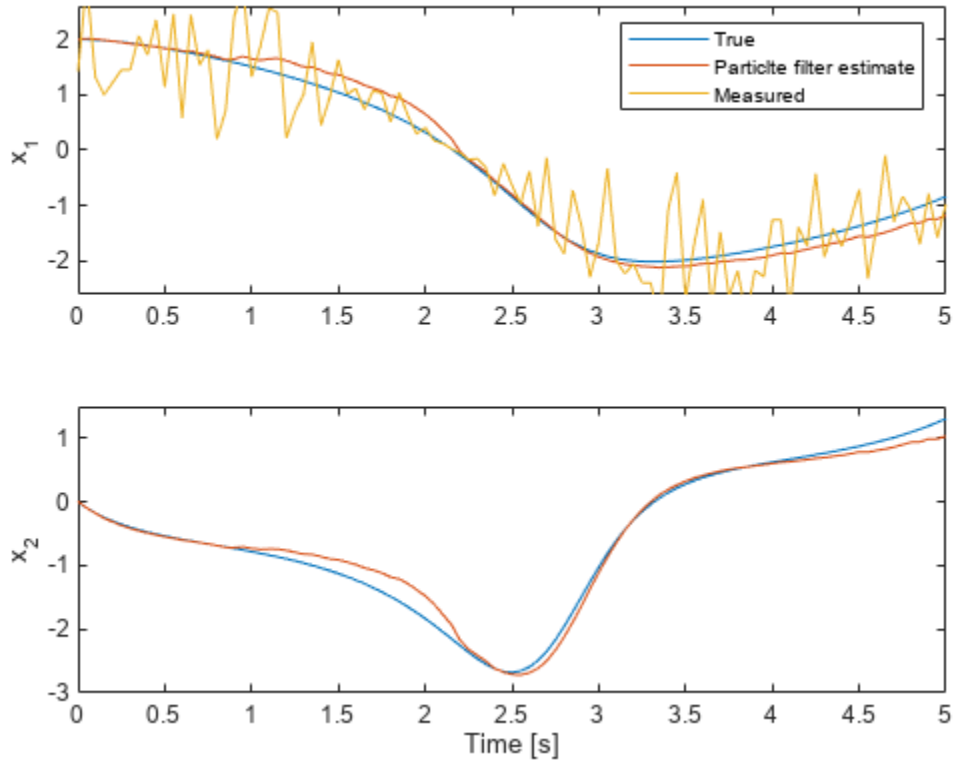
```
pf.ResamplingMethod

ans =
'multinomial'

pf.ResamplingPolicy

ans =
  particleResamplingPolicy with properties:

               TriggerMethod: 'ratio'
            SamplingInterval: 1
    MinEffectiveParticleRatio: 0.5000
```

Start the estimation loop. This represents measurements arriving over time, step by step.

```
% Estimate
xCorrectedPF = zeros(size(xTrue));
for k=1:size(xTrue,1)
    % Use measurement y[k] to correct the particles for time k
    xCorrectedPF(k,:) = correct(pf,yMeas(k)); % Filter updates and stores Particles[k|k], Weights
    % The result is x[k|k]: Estimate of states at time k, utilizing
    % measurements up to time k. This estimate is the mean of all particles
    % because StateEstimationMethod was 'mean'.
    %
    % Now, predict particles at next time step. These are utilized in the
    % next correct command
    predict(pf); % Filter updates and stores Particles[k+1|k]
end
```

Plot the state estimates from particle filter:

```
figure();
subplot(2,1,1);
plot(timeVector,xTrue(:,1),timeVector,xCorrectedPF(:,1),timeVector,yMeas(:));
legend('True','Particlte filter estimate','Measured')
ylim([-2.6 2.6]);
ylabel('x_1');
subplot(2,1,2);
plot(timeVector,xTrue(:,2),timeVector,xCorrectedPF(:,2));
ylim([-3 1.5]);
xlabel('Time [s]');
ylabel('x_2');
```



The top plot shows the true value, particle filter estimate, and the measured value of the first state. The filter utilizes the system model and noise information to produce an improved estimate over the measurements. The bottom plot shows the second state. The filter is successful in producing a good estimate.

The validation of the particle filter performance involves performing statistical tests on residuals, similar to those that were performed earlier in this example for unscented Kalman filter results.

Summary

This example has shown the steps of constructing and using an unscented Kalman filter and a particle filter for state estimation of a nonlinear system. You estimated states of a van der Pol oscillator from noisy measurements, and validated the estimation performance.

See Also

`extendedKalmanFilter` | `unscentedKalmanFilter` | `particleFilter`

More About

- “Extended and Unscented Kalman Filter Algorithms for Online State Estimation” on page 13-2
- “Validate Online State Estimation at the Command Line” on page 13-12
- “Troubleshoot Online State Estimation” on page 13-17
- “Generate Code for Online State Estimation in MATLAB” on page 13-9

Estimate States of Nonlinear System with Multiple, Multirate Sensors

This example shows how to perform nonlinear state estimation in Simulink™ for a system with multiple sensors operating at different sample rates. The Extended Kalman Filter block in Control System Toolbox™ is used to estimate the position and velocity of an object using GPS and radar measurements.

Introduction

The toolbox has three Simulink blocks for nonlinear state estimation:

- Extended Kalman Filter: Implements the first-order discrete-time extended Kalman filter algorithm.
- Unscented Kalman Filter: Implements the discrete-time unscented Kalman filter algorithm.
- Particle Filter: Implements a discrete-time particle filter algorithm.

These blocks support state estimation using multiple sensors operating at different sample rates. A typical workflow for using these blocks is as follows:

1. Model your plant and sensor behavior using MATLAB or Simulink functions.
2. Configure the parameters of the block.
3. Simulate the filter and analyze results to gain confidence in filter performance.
4. Deploy the filter on your hardware. You can generate code for these filters using Simulink Coder™ software.

This example uses the Extended Kalman Filter block to demonstrate the first two steps of this workflow. The last two steps are briefly discussed in the **Next Steps** section. The goal in this example is to estimate the states of an object using noisy measurements provided by a radar and a GPS sensor. The states of the object are its position and velocity, which are denoted as xTrue in the Simulink model.

If you are interested in the Particle Filter block, please see the example "Parameter and State Estimation in Simulink Using Particle Filter Block".

Plant Modeling

The extended Kalman filter (EKF) algorithm requires a state transition function that describes the evolution of states from one time step to the next. The block supports the following two function forms:

- Additive process noise: $x[k+1] = f(x[k], u[k]) + w[k]$
- Nonadditive process noise: $x[k+1] = f(x[k], w[k], u[k])$

Here f(..) is the state transition function, x is the state, and w is the process noise. u is optional, and represents additional inputs to f, for instance system inputs or parameters. Additive noise means that the next state $x[k+1]$ and process noise $w[k]$ are related linearly. If the relationship is nonlinear, use the nonadditive form.

The function f(...) can be a MATLAB Function that comply with the restrictions of MATLAB Coder™, or a Simulink Function block. After you create f(...), you specify the function name and whether the process noise is additive or nonadditive in the Extended Kalman Filter block.

In this example, you are tracking the north and east positions and velocities of an object on a 2-dimensional plane. The estimated quantities are:

$$\hat{x}[k] = \begin{bmatrix} \hat{x}_e[k] \\ \hat{x}_n[k] \\ \hat{v}_e[k] \\ \hat{v}_n[k] \end{bmatrix} \begin{array}{l} \text{East position estimate } [m] \\ \text{North position estimate } [m] \\ \text{East velocity estimate } [m/s] \\ \text{North velocity estimate } [m/s] \end{array}$$

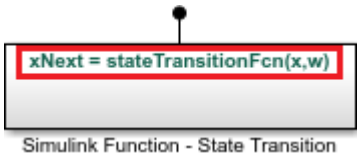
Here k is the discrete-time index. The state transition equation used is of the nonadditive form $\hat{x}[k+1] = A\hat{x}[k] + Gw[k]$, where $\hat{x}$ is the state vector, and w is the process noise. The filter assumes that w is a zero-mean, independent random variable with known variance $E[ww^T]$. The A and G matrices are:

$$A = \begin{bmatrix} 1 & 0 & T_s & 0 \\ 0 & 1 & 0 & T_s \\ 0 & 0 & 1 & 0 \\ 0 & 0 & 0 & 1 \end{bmatrix} \qquad G = \begin{bmatrix} T_s/2 & 0 \\ 0 & T_s/2 \\ 1 & 0 \\ 0 & 1 \end{bmatrix}$$

where T_s is the sample time. The third row of A and G model the east velocity as a random walk: $\hat{v}_e[k+1] = \hat{v}_e[k] + w_1[k]$. In reality, position is a continuous-time variable and is the integral of velocity over time $\frac{d}{dt}\hat{x}_e = \hat{v}_e$. The first row of A and G represent a discrete approximation to this kinematic relationship: $(\hat{x}_e[k+1] - \hat{x}_e[k])/T_s = (\hat{v}_e[k+1] + \hat{v}_e[k])/2$. The second and fourth rows of A and G represent the same relationship between the north velocity and position. This state transition model is linear, but the radar measurement model is nonlinear. This nonlinearity necessitates the use of a nonlinear state estimator such as the extended Kalman filter.

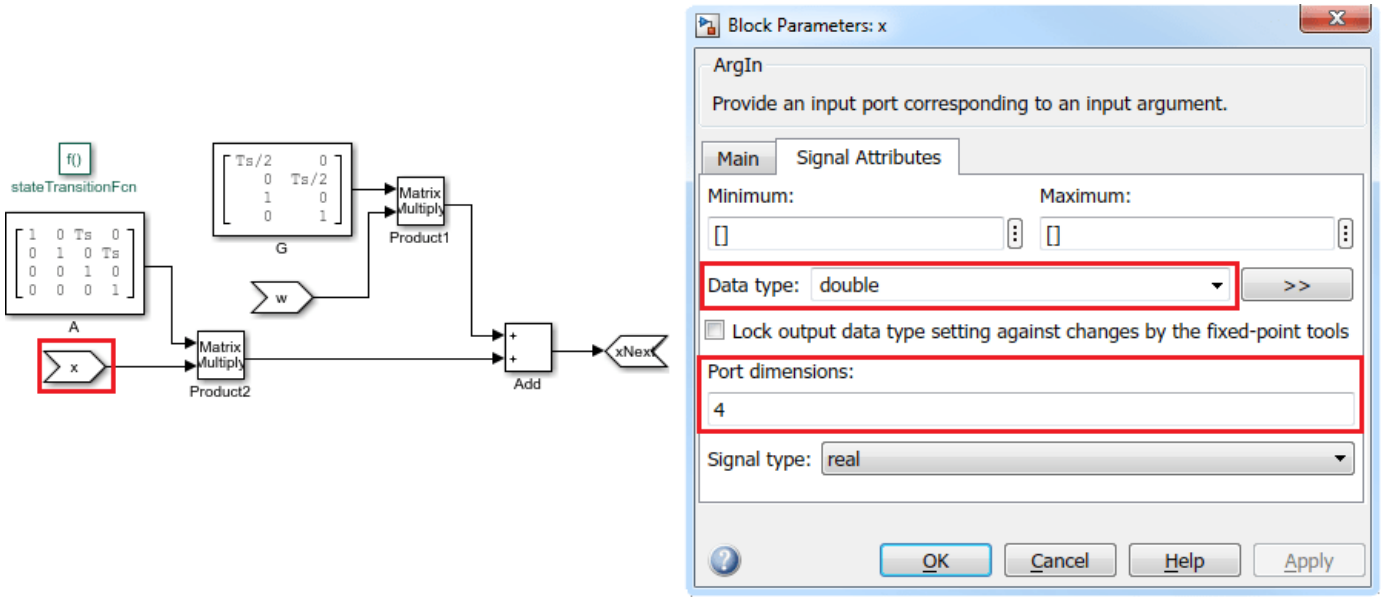
In this example you implement the state transition function using a Simulink Function block. To do so,

- Add a `Simulink Function` block to your model from the `Simulink/User-Defined Functions` library
- Click on the name shown on the Simulink Function block. Edit the function name, and add or remove input and output arguments, as necessary. In this example the name for the state transition function is `stateTransitionFcn`. It has one output argument (xNext) and two input arguments (x, w).



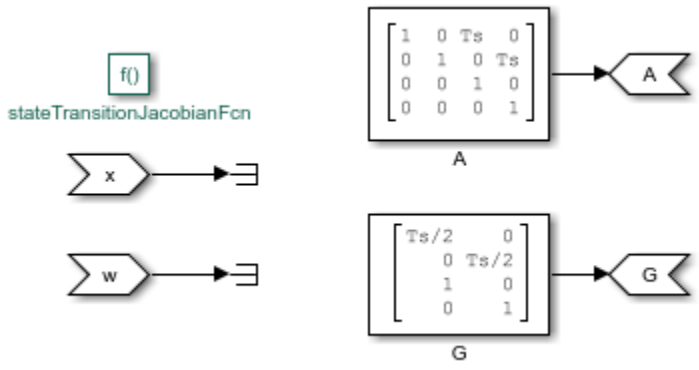
- Though it is not required in this example, you can use any signals from the rest of your Simulink model in the Simulink Function. To do so, add `Inport` blocks from the `Simulink/Sources` library. Note that these are different than the `ArgIn` and `ArgOut` blocks that are set through the signature of your function (xNext = stateTransitionFcn(x, w)).

- In the Simulink Function block, construct your function utilizing Simulink blocks.
- Set the dimensions for the input and output arguments x, w, and xNext in the **Signal Attributes** tab of the `ArgIn` and `ArgOut` blocks. The data type and port dimensions must be consistent with the information you provide in the `Extended Kalman Filter` block.



Analytical Jacobian of the state transition function is also implemented in this example. Specifying the Jacobian is optional. However, this reduces the computational burden, and in most cases increases the state estimation accuracy. Implement the Jacobian function as a Simulink function because the state transition function is a Simulink function.

```
open_system('multirateEKFExample')
open_system('multirateEKFExample/Simulink Function - State Transition Jacobian');
```



Sensor modeling - Radar

The Extended Kalman Filter block also needs a measurement function that describes how the states are related to measurements. The following two function forms are supported:

- Additive measurement noise: $y[k] = h(x[k], u[k]) + v[k]$

- Nonadditive measurement noise: $y[k] = h(x[k], v[k], u[k])$

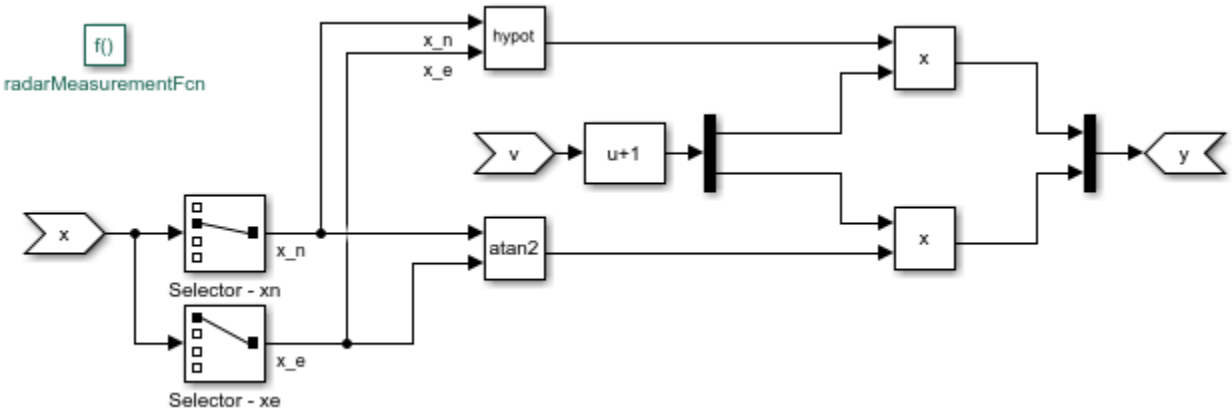
Here h(..) is the measurement function, and v is the measurement noise. u is optional, and represents additional inputs to h, for instance system inputs or parameters. These inputs can differ from the inputs in the state transition function.

In this example a radar located at the origin measures the range and angle of the object at 20 Hz. Assume that both of the measurements have about 5% noise. This can be modeled by the following measurement equation:

$$y_{radar}[k] = \begin{bmatrix} \sqrt{x_n[k]^2 + x_e[k]^2}\ (1 + v_1[k]) \\ atan2(x_n[k], x_e[k])\ (1 + v_2[k]) \end{bmatrix}$$

Here $v_1[k]$ and $v_2[k]$ are the measurement noise terms, each with variance 0.05^2. That is, most of the measurements have errors less than 5%. The measurement noise is nonadditive because $v_1[k]$ and $v_2[k]$ are not simply added to the measurements, but instead they depend on the states x. In this example, the radar measurement equation is implemented using a Simulink Function block.

```
open_system('multirateEKFExample/Simulink Function - Radar Measurements');
```



Sensor modeling - GPS

A GPS measures the east and north positions of the object at 1 Hz. Hence, the measurement equation for the GPS sensor is:

$$y_{GPS}[k] = \begin{bmatrix} x_e[k] \\ x_n[k] \end{bmatrix} + \begin{bmatrix} v_1[k] \\ v_2[k] \end{bmatrix}$$

Here $v_1[k]$ and $v_2[k]$ are measurement noise terms with the covariance matrix [10^2 0; 0 10^2]. That is, the measurements are accurate up to approximately 10 meters, and the errors are uncorrelated. The measurement noise is additive because the noise terms affect the measurements y_{GPS} linearly.

Create this function, and save it in a file named `gpsMeasurementFcn.m`. When the measurement noise is additive, you must not specify the noise terms in the function. You provide this function name and measurement noise covariance in the Extended Kalman Filter block.

```
type gpsMeasurementFcn
```

```
function y = gpsMeasurementFcn(x)
% gpsMeasurementFcn GPS measurement function for state estimation
%
% Assume the states x are:
%   [EastPosition; NorthPosition; EastVelocity; NorthVelocity]

%#codegen

% The %#codegen tag above is needed is you would like to use MATLAB Coder to
% generate C or C++ code for your filter

y = x([1 2]); % Position states are measured
end
```

Filter Construction

Configure the Extended Kalman Filter block to perform the estimation. You specify the state transition and measurement function names, initial state and state error covariance, and process and measurement noise characteristics.

In the **System Model** tab of the block dialog, specify the following parameters:

State Transition

1. Specify the state transition function, `stateTransitionFcn`, in **Function**. Since you have the Jacobian of this function, select **Jacobian**, and specify the Jacobian function, `stateTransitionJacobianFcn`.
2. Select `Nonadditive` in the **Process Noise** drop-down list because you explicitly stated how the process noise impacts the states in your function.
3. Specify the process noise covariance as [0.2 0; 0 0.2]. As explained in the **Plant Modeling** section of this example, process noise terms define the random walk of the velocities in each direction. The diagonal terms approximately capture how much the velocities can change over one sample time of the state transition function. The off-diagonal terms are set to zero, which is a naive assumption that velocity variations in the north and east directions are uncorrelated.

Initialization

1. Specify your best initial state estimate in **Initial state**. In this example, specify [100; 100; 0; 0].
2. Specify your confidence in your state estimate guess in **Initial covariance**. In this example, specify 10. The software interprets this value as the true state values are likely to be within $\pm\sqrt{10}$ of your initial estimate. You can specify a separate value for each state by setting `Initial covariance` as a vector. You can specify cross-correlations in this uncertainty by specifying it as a matrix.

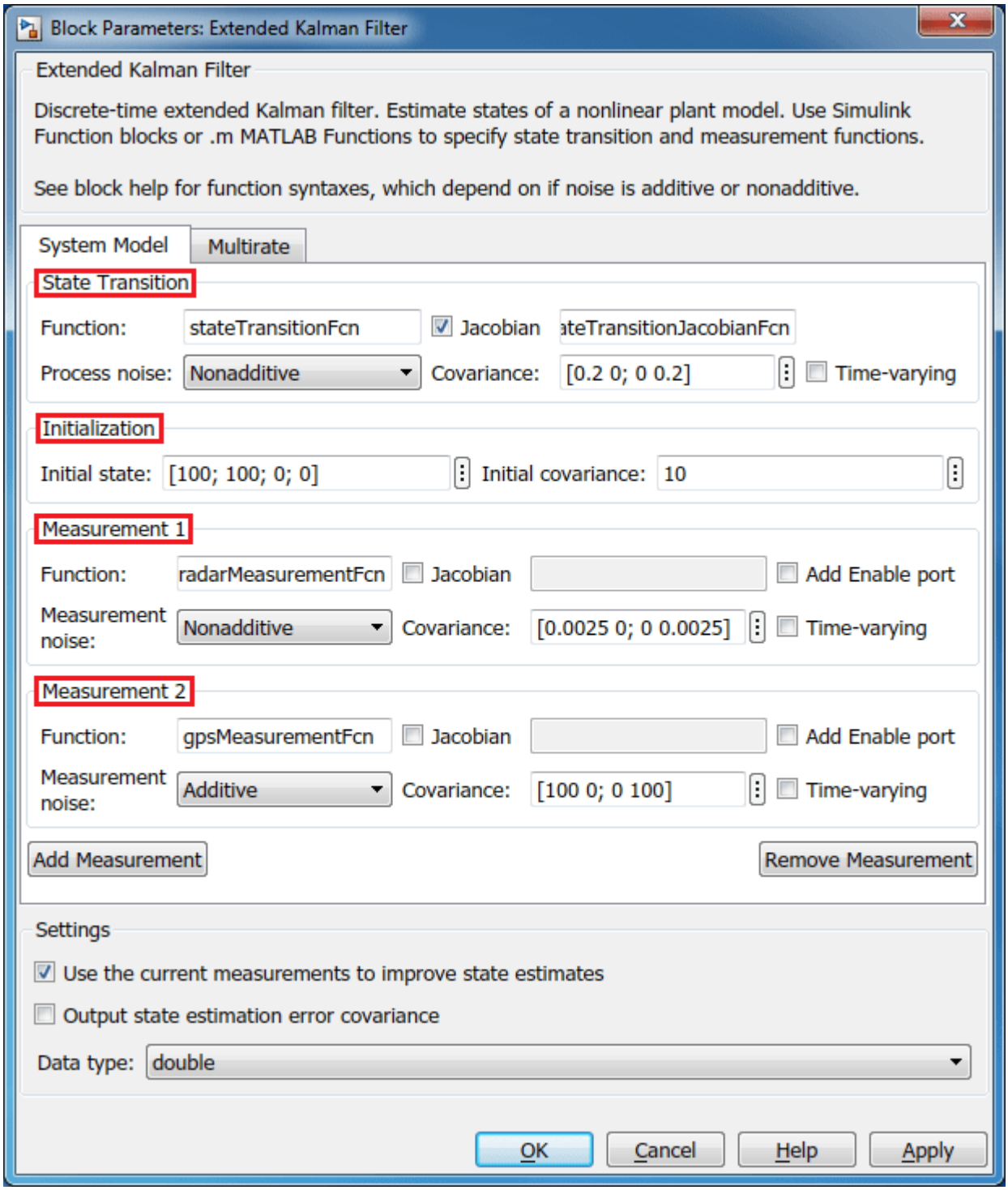
Since there are two sensors, click **Add Measurement** to specify a second measurement function.

Measurement 1

1. Specify the name of your measurement function, `radarMeasurementFcn`, in **Function**.
2. Select `Nonadditive` in the **Measurement Noise** drop-down list because you explicitly stated how the process noise impacts the measurements in your function.
3. Specify the measurement noise covariance as [0.05^2 0; 0 0.05^2] per the discussion in the **Sensor Modeling - Radar** section.

Measurement 2

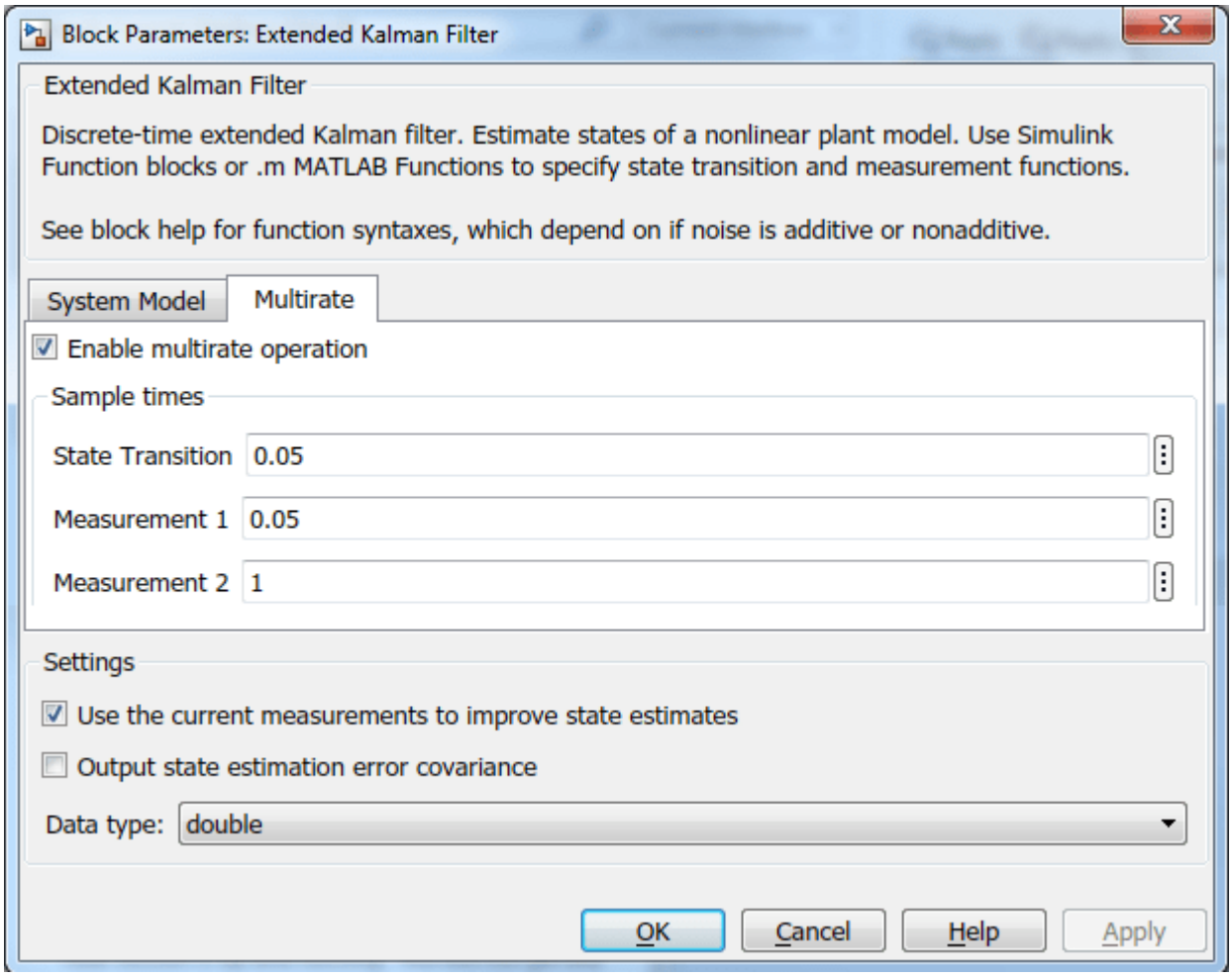
1. Specify the name of your measurement function, `gpsMeasurementFcn`, in **Function**.
2. This sensor model has additive noise. Therefore, specify the GPS measurement noise as `Additive` in the **Measurement Noise** drop-down list.
3. Specify the measurement noise covariance as [100 0; 0 100].



In the **Multirate** tab, since the two sensors are operating at different sample rates, perform the following configuration:

1 Select `Enable multirate operation`.

2. Specify the state transition sample time. The state transition sample time must be the smallest, and all measurement sample times must be an integer multiple of the state transition sample time. Specify **State Transition** sample time as 0.05, the sample time of the fastest measurement. Though not required in this example, it is possible to have a smaller sample time for state transition than all measurements. This means there will be some sample times without any measurements. For these sample times the filter generates state predictions using the state transition function.
3. Specify the **Measurement 1** sample time (Radar) as 0.05 seconds and **Measurement 2** (GPS) as 1 seconds.



Simulation and Results

Test the performance of the Extended Kalman filter by simulating a scenario where the object travels in a square pattern with the following maneuvers:

- At t = 0, the object starts at $x_e(0) = 100\ [\mathrm{m}], x_n(0) = 100\ [\mathrm{m}]$
- It heads north at $\dot{x}_n = 50\ [\mathrm{m/s}]$ until t = 20 seconds.
- It heads east at $\dot{x}_n = 40\ [\mathrm{m/s}]$ between t = 20 and t = 45 seconds.

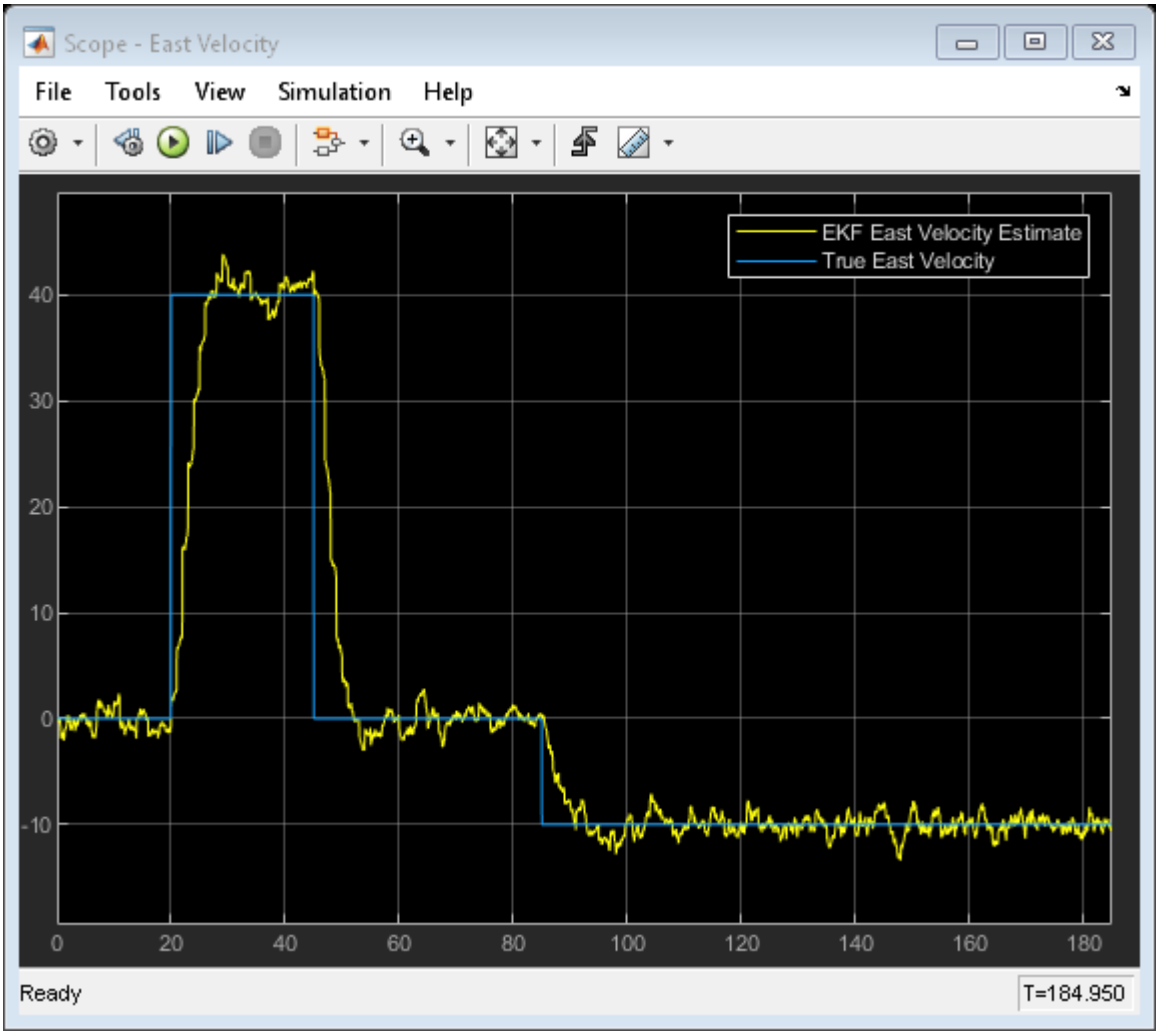
- It heads south at $\dot{x}_n = -25\ [\mathrm{m/s}]$ between t = 45 and t = 85 seconds.
- It heads west at $\dot{x}_e = -10\ [\mathrm{m/s}]$ between t = 85 and t = 185 seconds.

Generate the true state values corresponding to this motion:

```
Ts = 0.05; % [s] Sample rate for the true states
[t, xTrue] = generateTrueStates(Ts); % Generate position and velocity profile over 0-185 seconds
```

Simulate the model. For instance, look at the actual and estimated velocities in the east direction:

```
sim('multirateEKFExample');
open_system('multirateEKFExample/Scope - East Velocity');
```



The plot shows the true velocity in the east direction, and its extended Kalman filter estimates. The filter successfully tracks the changes in velocity. The multirate nature of the filter is most apparent in the time range t = 20 to 30 seconds. The filter makes large corrections every second (GPS sample rate), while the corrections due to radar measurements are visible every 0.05 seconds.

Next Steps

1. Validate the state estimation: The validation of unscented and extended Kalman filter performance is typically done using extensive Monte Carlo simulations. For more information, see “Validate Online State Estimation in Simulink” on page 13-14.
2. Generate code: The Unscented and Extended Kalman Filter blocks support C and C++ code generation using Simulink Coder™ software. The functions you provide to these blocks must comply with the restrictions of MATLAB Coder™ software (if you are using MATLAB functions to model your system) and Simulink Coder software (if you are using Simulink Function blocks to model your system).

Summary

This example has shown how to use the Extended Kalman Filter block in System Identification Toolbox. You estimated position and velocity of an object from two different sensors operating at different sampling rates.

```
close_system('multirateEKFExample', 0);
```

See Also

Extended Kalman Filter | Unscented Kalman Filter | Particle Filter

More About

- “Extended and Unscented Kalman Filter Algorithms for Online State Estimation” on page 13-2
- “Validate Online State Estimation in Simulink” on page 13-14
- “Troubleshoot Online State Estimation” on page 13-17

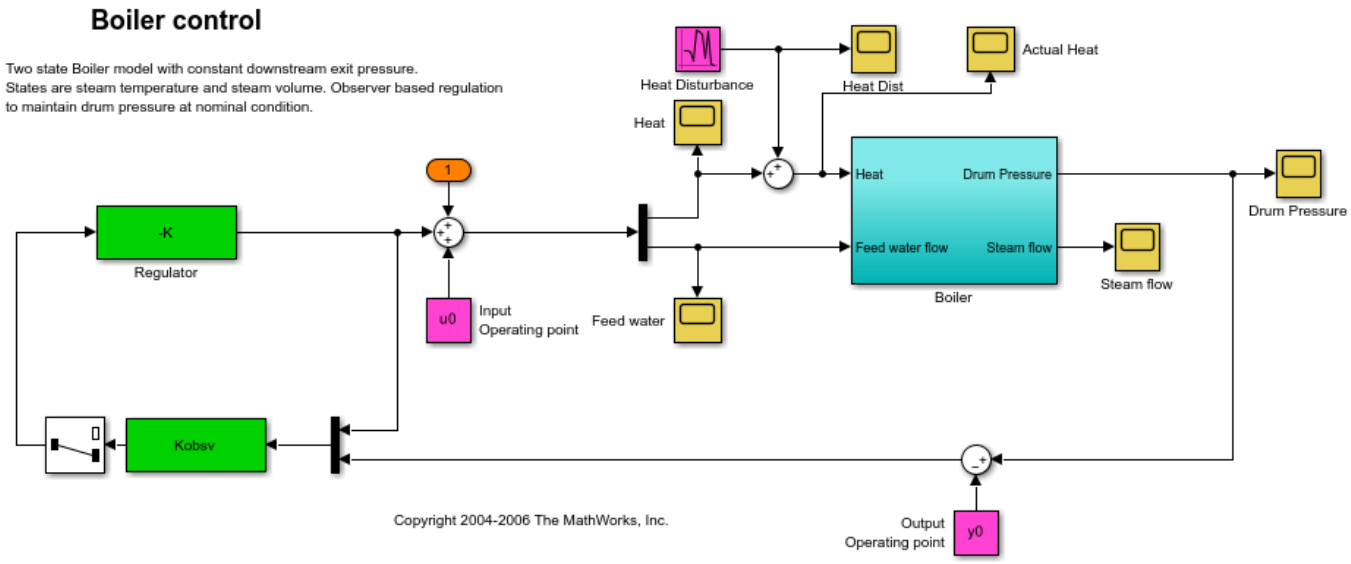
Regulate Pressure in Drum Boiler

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:
----------
    Min          x          Max        dxMin        dx         dxMax
_________ _________ _________ _________ _________ _________

(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:
----------
    Min          u          Max
_________ _________ _________

(1.) Boiler_Demo/Input
     0      241069.0782     Inf
     0       100.1327       Inf

Outputs:
----------
  Min       y       Max
_______ _______ _______

(1.) Boiler_Demo/Boiler
   0     1002.381   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

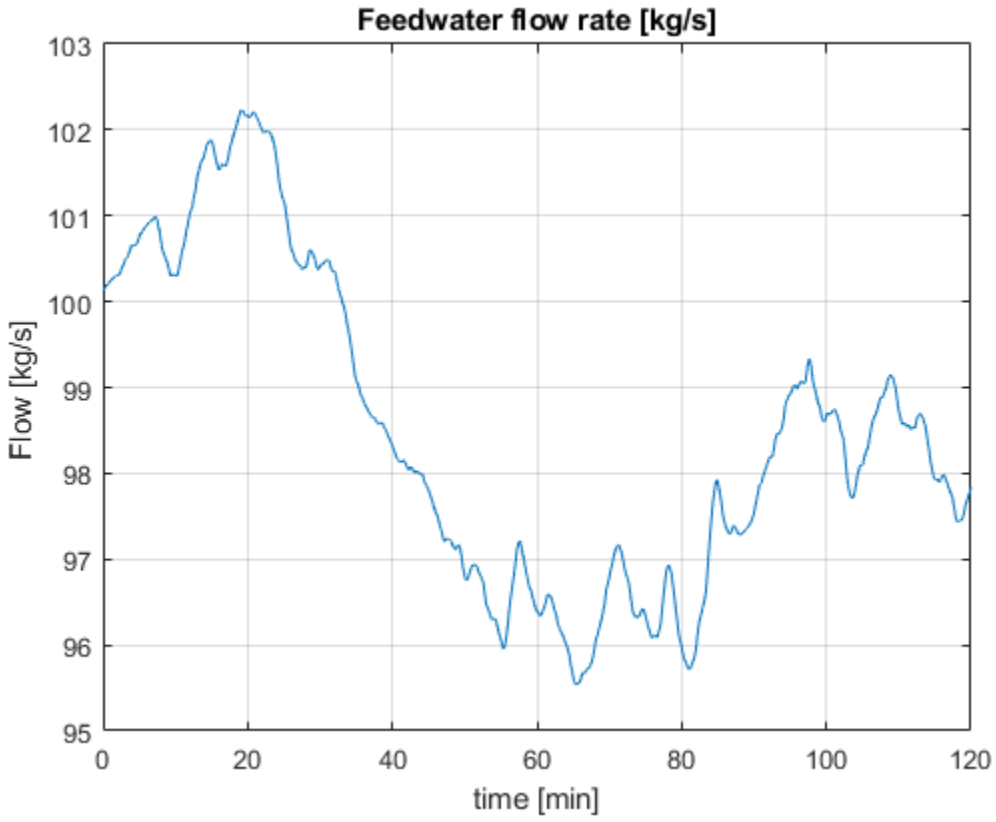
Simulate the model for the designed controller.

```
sim(mdl)
```

Plot the process input and output signals. The following figure shows the feed water actuation signal in kg/s.

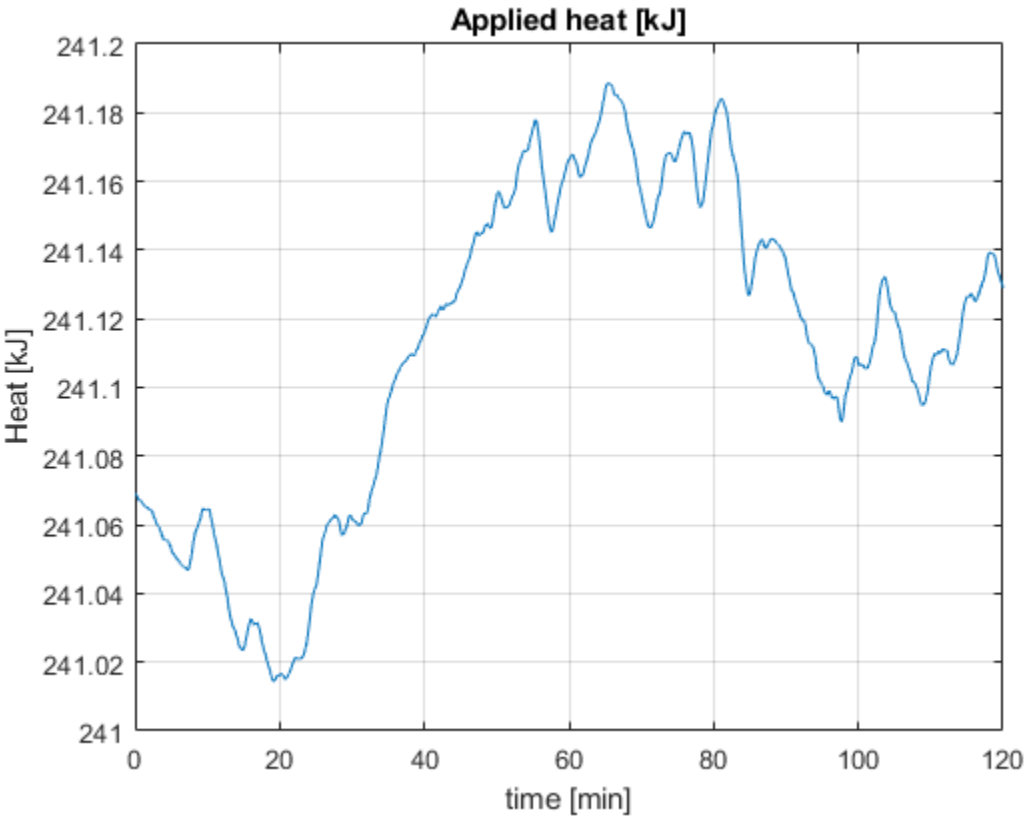
```
plot(FeedWater.time/60,FeedWater.signals.values)
title('Feedwater flow rate [kg/s]');
```

```
ylabel('Flow [kg/s]')
xlabel('time [min]')
grid on
```



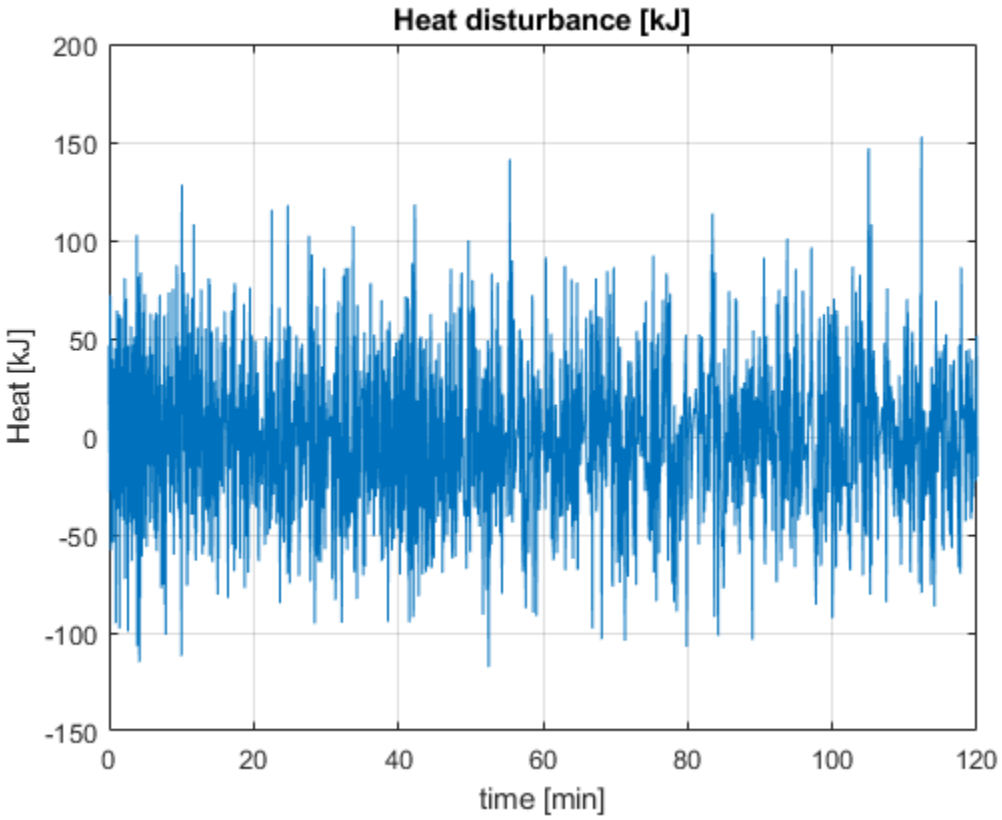
The following plot shows the heat actuation signal in kJ.

```
plot(Heat.time/60,Heat.signals.values/1000)
title('Applied heat [kJ]');
ylabel('Heat [kJ]')
xlabel('time [min]')
grid on
```



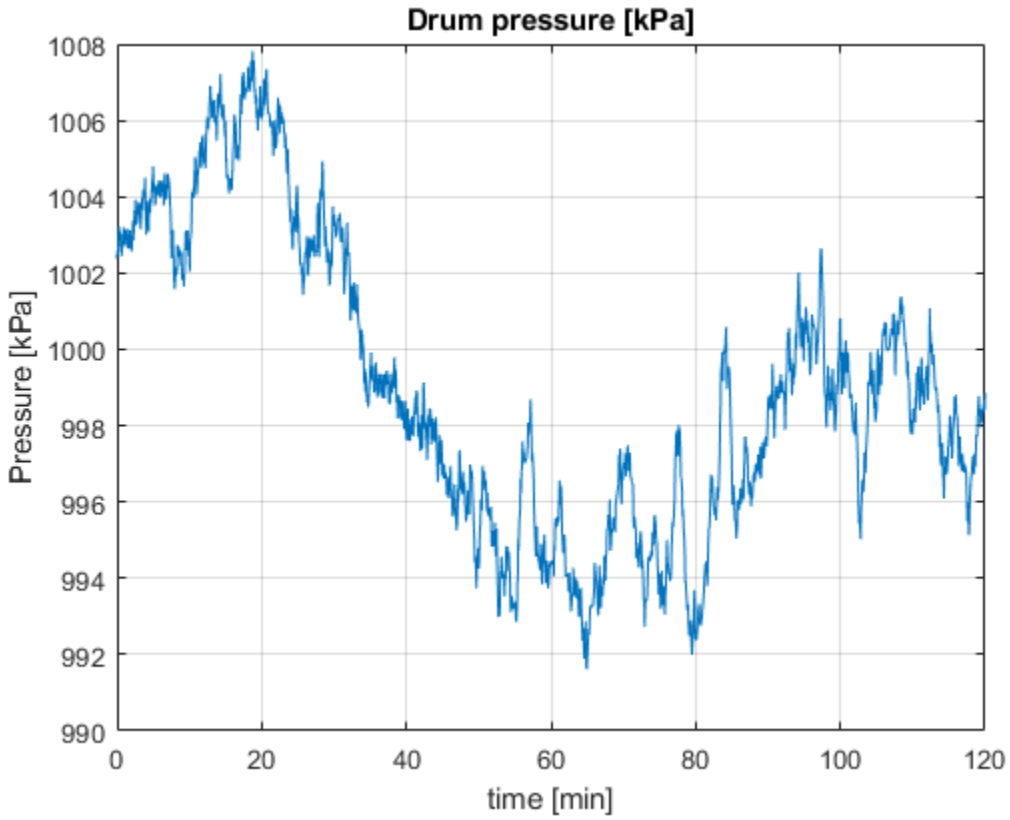
The next figure shows the heat disturbance in kJ. The disturbance varies by as much as 50% of the nominal heat value.

```
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`

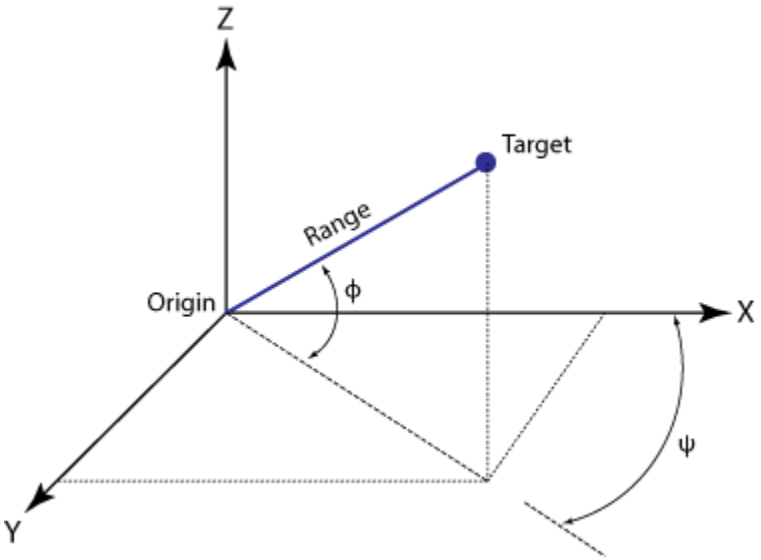
Related Examples

- “Kalman Filtering” on page 12-265
- “Linear-Quadratic-Gaussian (LQG) Design”

State Estimation with Wrapped Measurements Using Extended Kalman Filter

This example shows how to use the extended Kalman filter algorithm for nonlinear state estimation for 3D tracking involving circularly wrapped angle measurements. For target tracking, sensors usually employ a spherical frame to report the position of an object in terms of azimuth, range, and elevation. The angular measurements from this set are reported within certain bounds. For instance, azimuth angles are reported within a range of -180° to 180° or 0° to 360°. However, tracking is usually done in a rectangular frame, which necessitates the use of a nonlinear filter that can handle the nonlinear transformations required to transform spherical measurements into rectangular states.

This example uses a constant velocity model to track the three-dimensional positions and velocities of a target. An Extended Kalman Filter is used as the nonlinear filter to track the states of this object. This filter uses azimuth (ψ), range, and elevation (ϕ) readings as measurements during this tracking exercise.



The constant velocity 3D plant model for this example consists of positions and velocities in the three axes as follows:

$$States = \begin{bmatrix} x \\ \dot{x} \\ y \\ \dot{y} \\ z \\ \dot{z} \end{bmatrix} = \begin{bmatrix} x \\ v_x \\ y \\ v_y \\ z \\ v_z \end{bmatrix}$$

The range, azimuth, and elevation angles are given by:

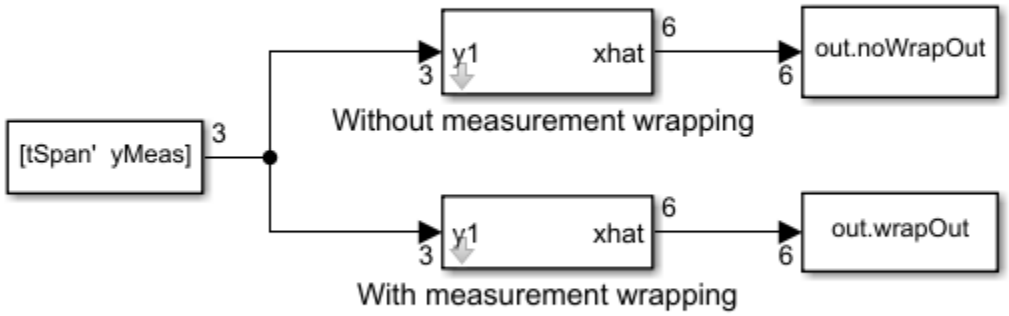
$$Range = \sqrt{x^2 + y^2 + z^2}$$

$$Elevation\ (\phi) = \sin^{-1}\left(\frac{z}{Range}\right)$$

$$Azimuth\ (\psi) = \tan^{-1}\left(\frac{y}{x}\right)$$

Computing the residual between the actual measurement and expected measurement is an essential step in the filtering process. Without the necessary precautions, the residual values may be large for an object located near the wrapping boundary. This causes the filter to diverge from the accurate state, leading to an inaccurate state estimate. This example demonstrates how to handle such situations by using measurement wrapping. A Simulink model is used to demonstrate the difference in tracked states when used with and without measurement wrapping.

EKF model with and without measurement wrapping



State Estimation - Without Measurement Wrapping

First provide your initial state guess and specify the model name.

```
rng(0)
initialStateGuess = [-100 0 0 0 0 0]';
modelName = 'modelEKFWrappedMeasurements';
```

Generate the measurements based on the noise covariance as follows:

```
dt = 3;
tSpan = 0:dt:100;
yTrue = zeros(length(tSpan),3);
yMeas = zeros(size(yTrue));
noiseCovariance = diag([20 0.5 0.5]);
xCurrent = initialStateGuess;

for i = 1:numel(tSpan)
   xTrue = stateTransitionFcn(xCurrent);
   yTrue(i,:) = (measurementFcn(xTrue))';
```

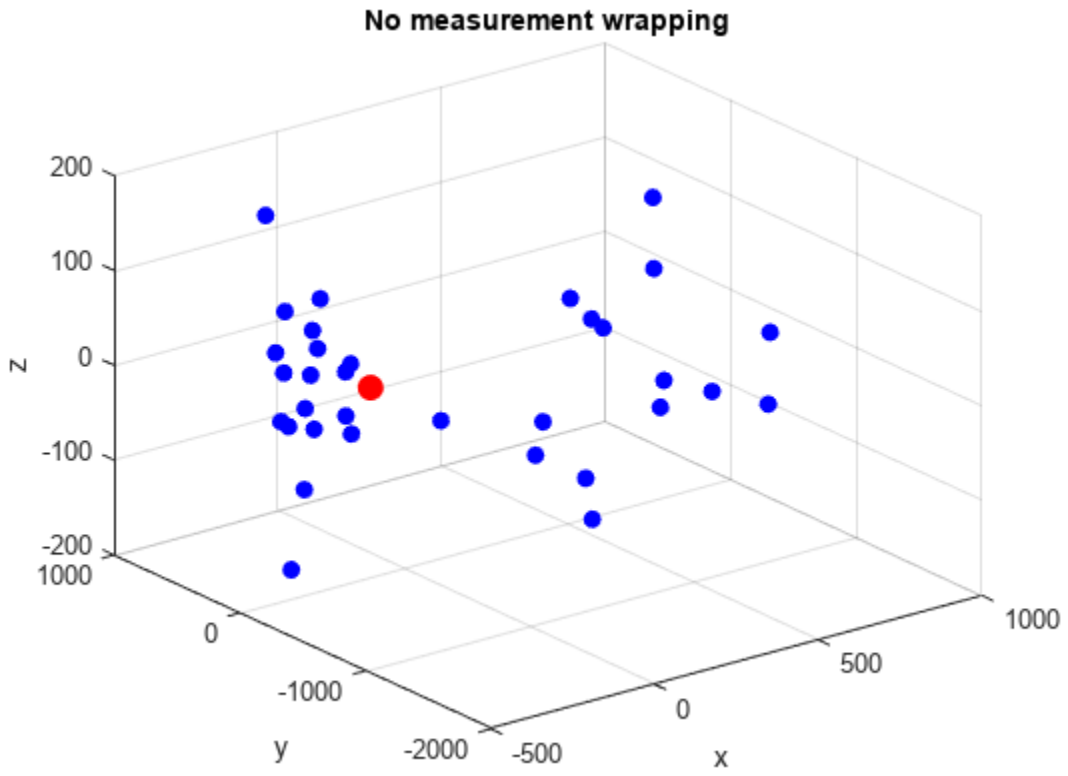
```
   yMeas(i,:) = yTrue(i,:) + (chol(noiseCovariance)*randn(3,1))';
   xCurrent = xTrue;
end
```

Simulate the model.

```
out = sim(modelName);
xEstNoWrap = zeros(numel(tSpan),length(initialStateGuess));
xEstWithWrap = zeros(numel(tSpan),length(initialStateGuess));
```

Run the filter and vizualize the state convergence on a 3D plot.

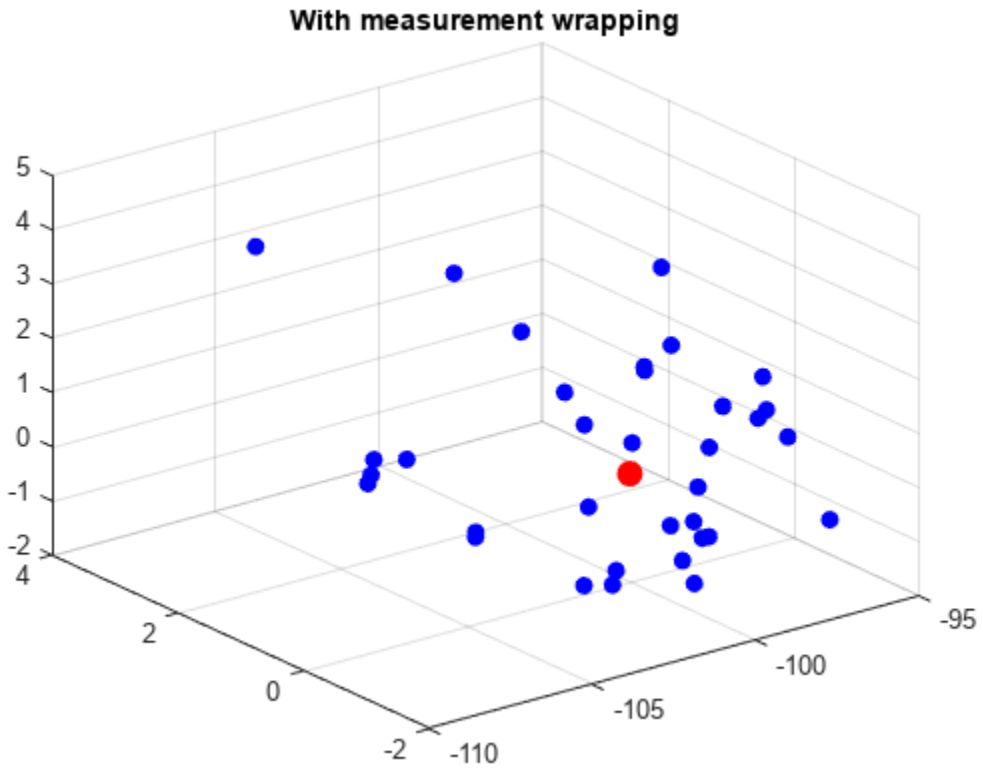
```
figure()
plot3(initialStateGuess(1), initialStateGuess(3), initialStateGuess(5),'ro','MarkerFaceColor','r
grid
hold on;

% Plot options
xlabel('x');
ylabel('y');
zlabel('z');
title('No measurement wrapping');

% Run filter
for i = 1:numel(tSpan)
   xEst = out.noWrapOut(i,:);
   plot3(xEst(1),xEst(3),xEst(5),'bo','MarkerFaceColor','b');
   hold on;
   xEstNoWrap(i,:) = xEst;
end
hold off
```



State Estimation - With Measurement Wrapping

Now, enable the measurement wrapping and run the filter. Visualize the state convergence on a 3D plot.

```
figure()
% Run filter
for i = 1:numel(tSpan)
   xEst = out.wrapOut(i,:);
   plot3(xEst(1),xEst(3),xEst(5),'bo','MarkerFaceColor','b');
   hold on;
   xEstWithWrap(i,:) = xEst;
end
plot3(initialStateGuess(1), initialStateGuess(3), initialStateGuess(5),'ro','MarkerFaceColor','r
grid
title('With measurement wrapping');
hold off
```



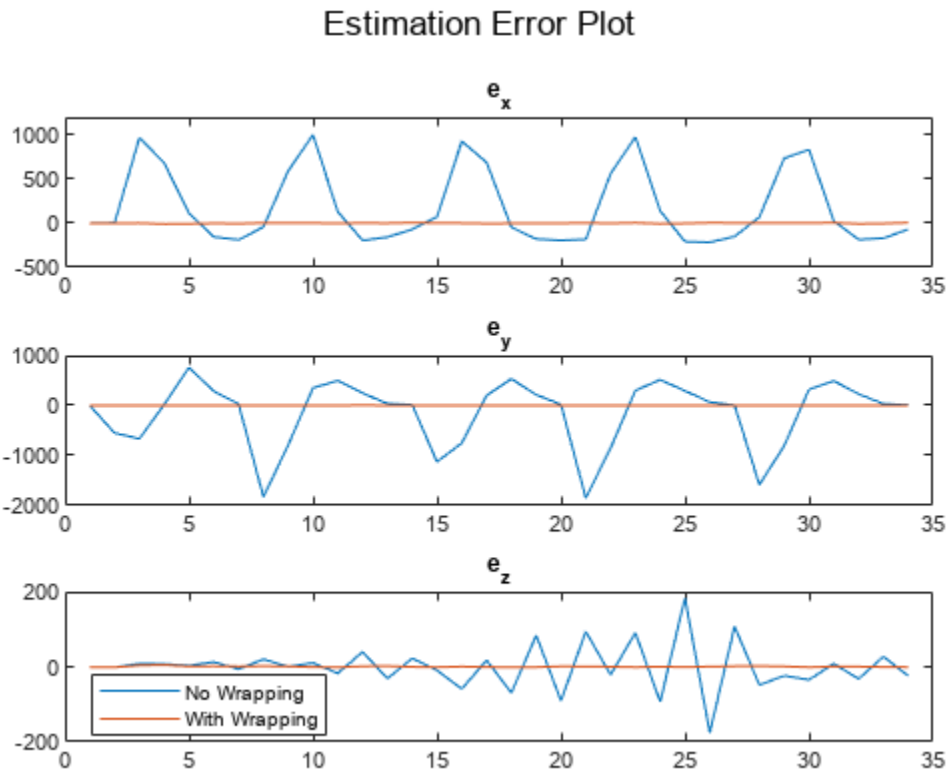
From the plot above, observe that the scale of the three axes is a lot smaller than the 3D plot for the case without measurement wrapping. This denotes that there is better convergence when measurement wrapping is enabled.

Validation

Compare the estimation errors for both cases on the same plot.

```
figure()
subplot(3,1,1)
plot(xEstNoWrap(:,1) - xTrue(1));
hold on
plot(xEstWithWrap(:,1) - xTrue(1));
ylim([-500 1200])
title('e_x');
hold off

subplot(3,1,2)
plot(xEstNoWrap(:,3) - xTrue(3));
hold on
plot(xEstWithWrap(:,3) - xTrue(3));
ylim([-2000 1000])
hold off
title('e_y');

subplot(3,1,3)
plot(xEstNoWrap(:,5) - xTrue(5));
```

```
hold on
plot(xEstWithWrap(:,5) - xTrue(5));
ylim([-200 200])
hold off
title('e_z');

sgtitle('Estimation Error Plot')
legend('No Wrapping', 'With Wrapping','location','best')
```



Observe that the errors are comparatively much larger in all the three axes for the case without measurement wrapping.

Summary

This example shows how to use an Extended Kalman Filter block for state estimation of a nonlinear system with and without measurement wrapping. The state estimation with wrapping enabled is observed to provide more accurate state estimates as determined by the 3D and error plots.

Supporting Functions

State Transition Function

The Extended Kalman Filter block requires a function that describes the evolution of states from one time step to the next. This is typically called the state transition function. For this example, the contents of the state transition function contained in `stateTransitionFcn.m` is as follows:

```
function x = stateTransitionFcn(x)
   dt = 3;
```

```
   A = [1 dt 0 0  0 0;
        0 1  0 0  0 0;
        0 0  1 dt 0 0;
        0 0  0 1  0 0;
        0 0  0 0  1 dt;
        0 0  0 0  0  1];
   x = A*x;
end
```

Measurement Function

The Extended Kalman Filter block also needs a function that describes how the model states are related to sensor measurements. This is the measurement function. For this example, the contents of the measurement function contained in `measurementFcn.m` is as follows:

```
function [y, bounds] = measurementFcn(x)
   xPos = x(1);
   yPos = x(3);
   zPos = x(5);

   % Range
   r = sqrt(xPos^2 + yPos^2 + zPos^2);
   % Azimuth
   psi = atan2d(yPos,xPos);
   % Elevation
   phi = asind(zPos/r);
   % Combined measurement
   y = [r ; psi; phi];
   bounds = [-inf inf;-180 180; -90 90];
end
```

See Also

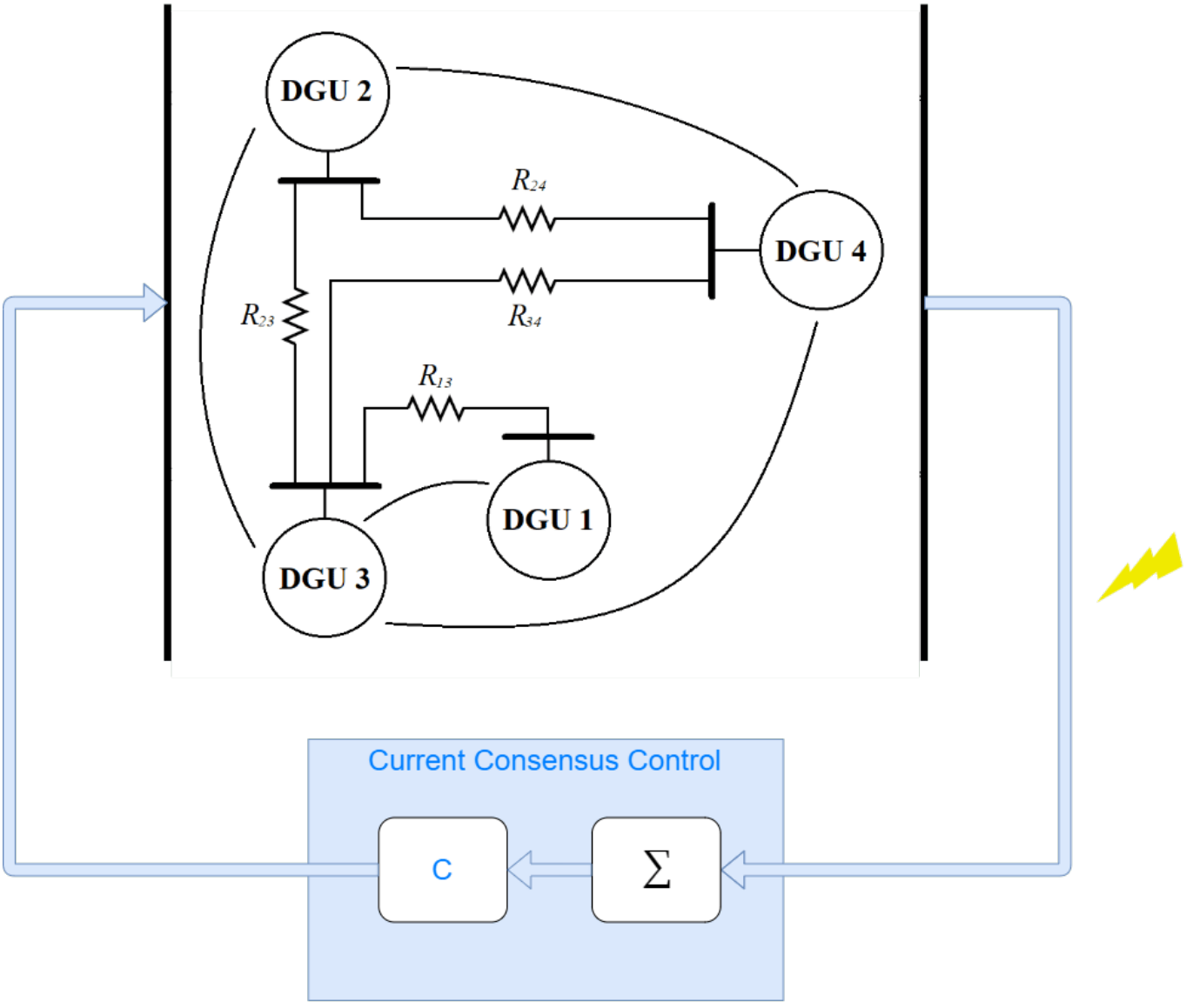
Extended Kalman Filter

Related Examples

- “Validate Online State Estimation in Simulink” on page 13-14
- “Troubleshoot Online State Estimation” on page 13-17

Detect Replay Attacks in DC Microgrids Using Distributed Watermarking

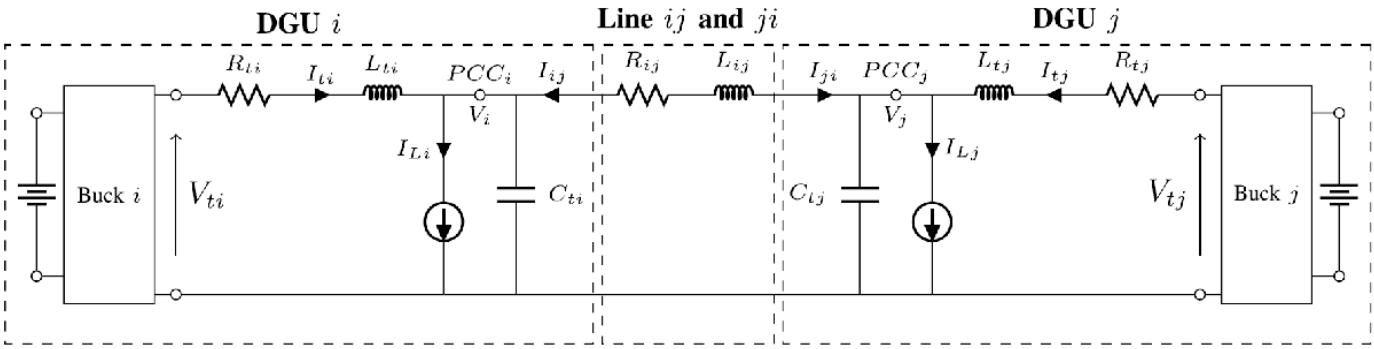
This example shows how to use distributed watermarking to detect replay attacks in a cyber-physical system (CPS). A DC Microgrid (DCmG) is an interconnected system of distributed generation units (DGU) that represent physical systems and are controlled using current consensus loops responsible for load sharing across multiple units.



This architecture requires the line currents of each DGU to communicate. The communication channel is open to attacks that can inject false information and alter the behavior of the central current consensus loop. One such attack is a replay attack, which delays the signals in the communication channel.

DC Microgrid Model

In this example, the DCmG consists of two connected distributed generation units modeled using the Average-Value DC-DC Converter (Simscape Electrical) blocks.



Set the variables that enable (`1`) or disable (`0`) the different modes. Start with all modes disabled. Use a sample time of 50 microseconds and a simulation duration of 5 seconds.

```
enableAttack = 0;
enableObserver = 0;
enableWatermarking = 0;
Ts = 5e-5;
stopTime = 5;
```

Specify the DGU parameters.

```
Vin = 100;
Ct = 2.2e-3;
Lt = 1.8e-3;
Rt = 0.2;
```

Specify the power line parameters.

```
Lij = 1.8e-6;
Rij = 0.05;
```

Modify the load current for the first unit to show the voltage and line current transients, as both units share the changed load.

```
ILoadInitial = 3;
ILoadFinal = 2;
tLoadChange = 4;
```

The model uses a PID controller for the primary local voltage control of each DGU. Specify parameters for the controllers.

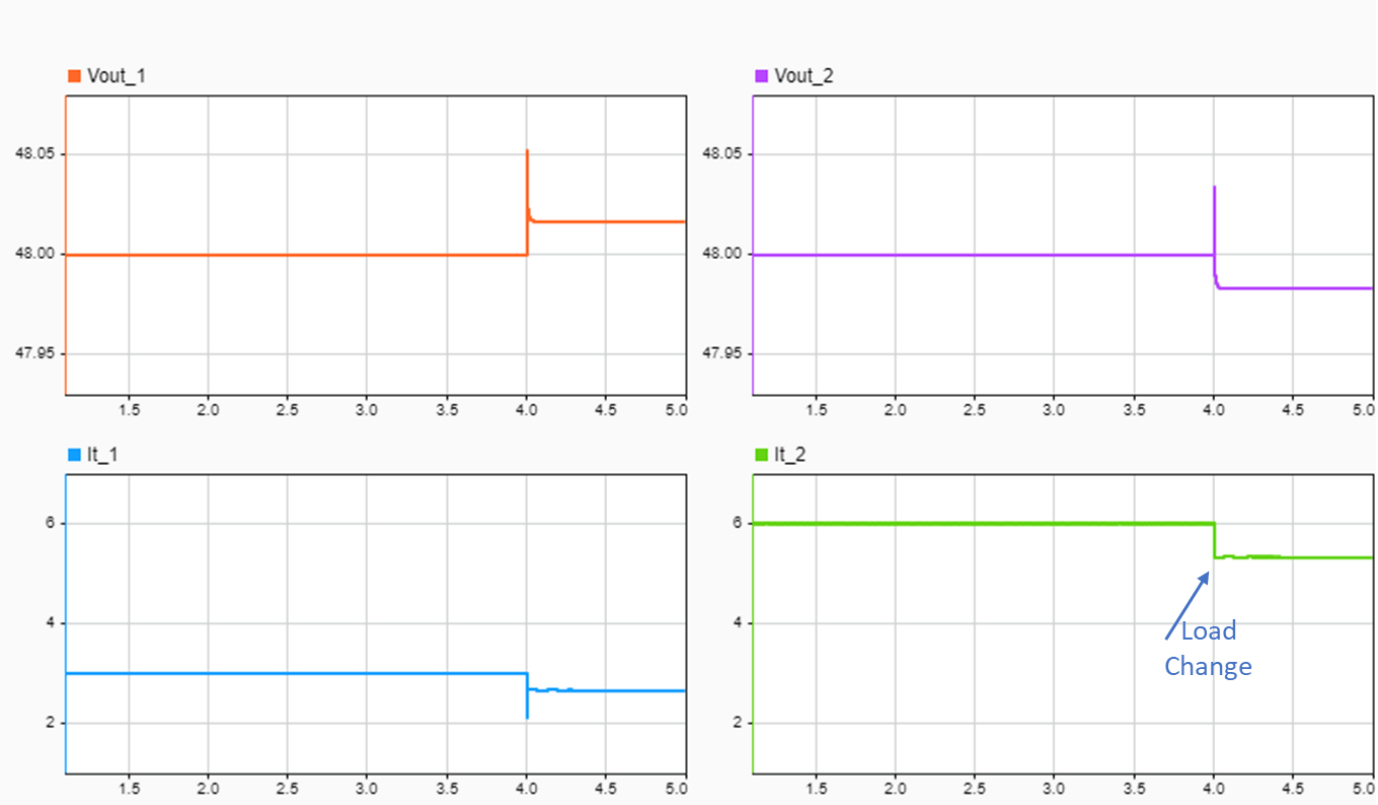
```
Vref = 48;
Kp = 0.1;
Ki = 6.15;
Kd = 4.6e-4;
N = 2.56e3;
Tf = 1/N;
```

The current consensus control is based on the weighted sum of normalized individual line currents by the rated capacity of the DGU. Define the capacities and PID control parameters.

```
ICapacity_1 = 3;
ICapacity_2 = 6;

CC_Kp = 0;
CC_Ki = 150;
CC_Kd = 0;
```

Open and simulate the model. You can see the current sharing between the two units at time `t` = 4 s, when the load in the first unit changes.

```
Simulink.sdi.view
Simulink.sdi.clear
open_system("AttackDetectionInDCMicrogrid.slx");
sim("AttackDetectionInDCMicrogrid.slx");
Simulink.sdi.loadView("DCmGCurrentSharingView.mldatx");
```



The currents in both units (bottom row of the plot) continue to be proportional to the individual rated capacities. The top row of the plot shows the unbalanced voltages.

Replay Attack

A replay attack is when a malicious agent can observe the signals in the communication channel and then replace current transmitted measurements with delayed recorded observations. This attack delays the signals in the communication channel. It is particularly deceptive when the system is at steady state.

This example uses a replay attack of the following form.

$$y_a(t) = y(t) + \beta(t - T_a)[-y(t) + y(t - t_0)]$$

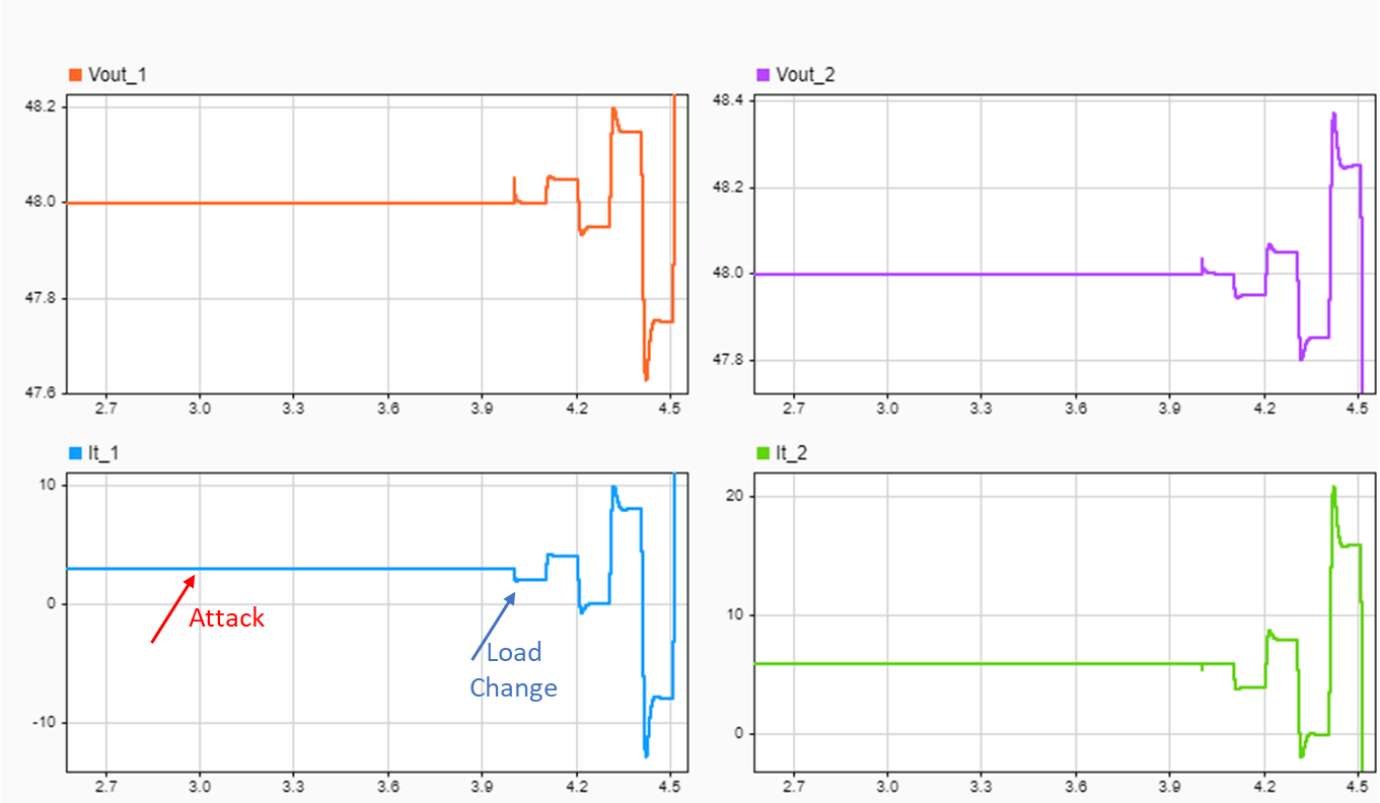
Here, $\beta(t - T_a)$ represents an activation function starting at time T_a that delays the signal $y(t)$ by t_0, and $y_a(t)$ represents the modified signal.

Enable the attack, and specify the time of attack and the delay the attack introduces in the signal.

```
enableAttack = 1;
timeOfAttack = 3;
attackTimeDelay = 0.1;
```

Simulate the model.

```
Simulink.sdi.clear
sim("AttackDetectionInDCMicrogrid.slx");
Simulink.sdi.loadView("DCmGAttackSimulationView.mldatx");
```



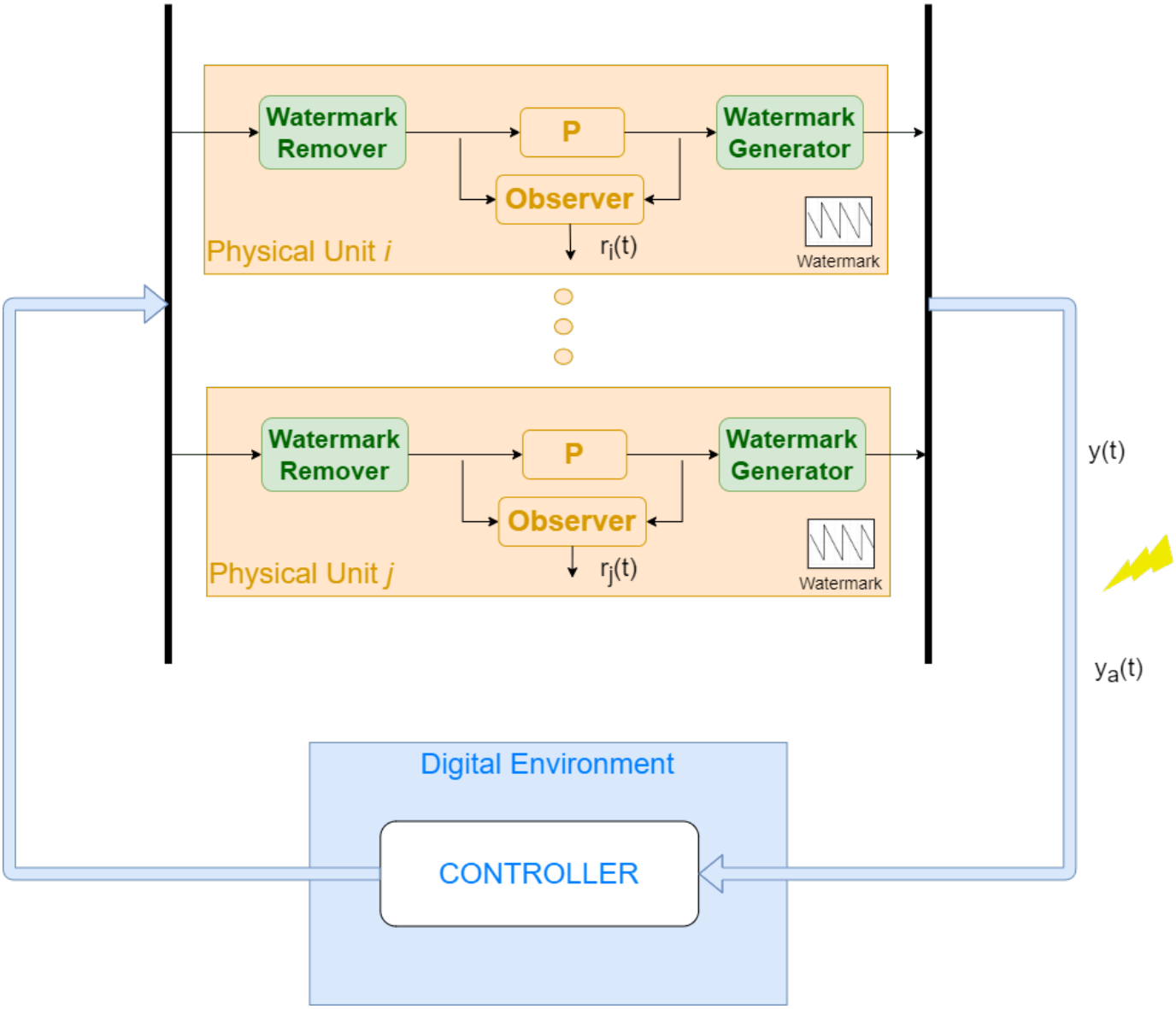
This attack results in undesired behavior when the load changes. The current consensus control does not perceive the changes to the line current due to the delay introduced in the communication channel.

The attack occurs from `t` = 3 seconds onwards, but it is not detected while the system is at steady state. The load changes at `t` = 4 seconds and incorrect measurements at the consensus controller result in an unstable system. Uncontrolled line currents (bottom row of the plot) exceed the rated capacity within 300 milliseconds of the load change, damaging the local DGU.

Watermarking and Attack Detection

Introduce a time-varying signal, unknown to the attacker, as a watermark and use the residuals generated by an observer to detect any modifications or false data injections to the signals in the communication channel.

```
enableObserver = 1;
enableWatermarking = 1;
```



The watermark signal is a periodic sawtooth wave, which the `Add Watermark` subsystems in the model inject into the voltage and current measurements communicated to the current consensus controller. The `Remove Watermark` subsystems subtract this signal from the feedback before the subsystems compute the adjustment to the reference voltage and generate the observer residuals. The watermark signal transforms the delay the attack introduces in the signals into a disturbance. Therefore, the system detects an attack when the observer generates a nonzero residual. The

amplitude of the watermark signal must be carefully set to allow the detection of a replay attack but not let currents exceed the rated capacity prior to the detection. In this example, the watermark signal amplitude is set to 0.1 and the frequency is set to 2 Hertz.

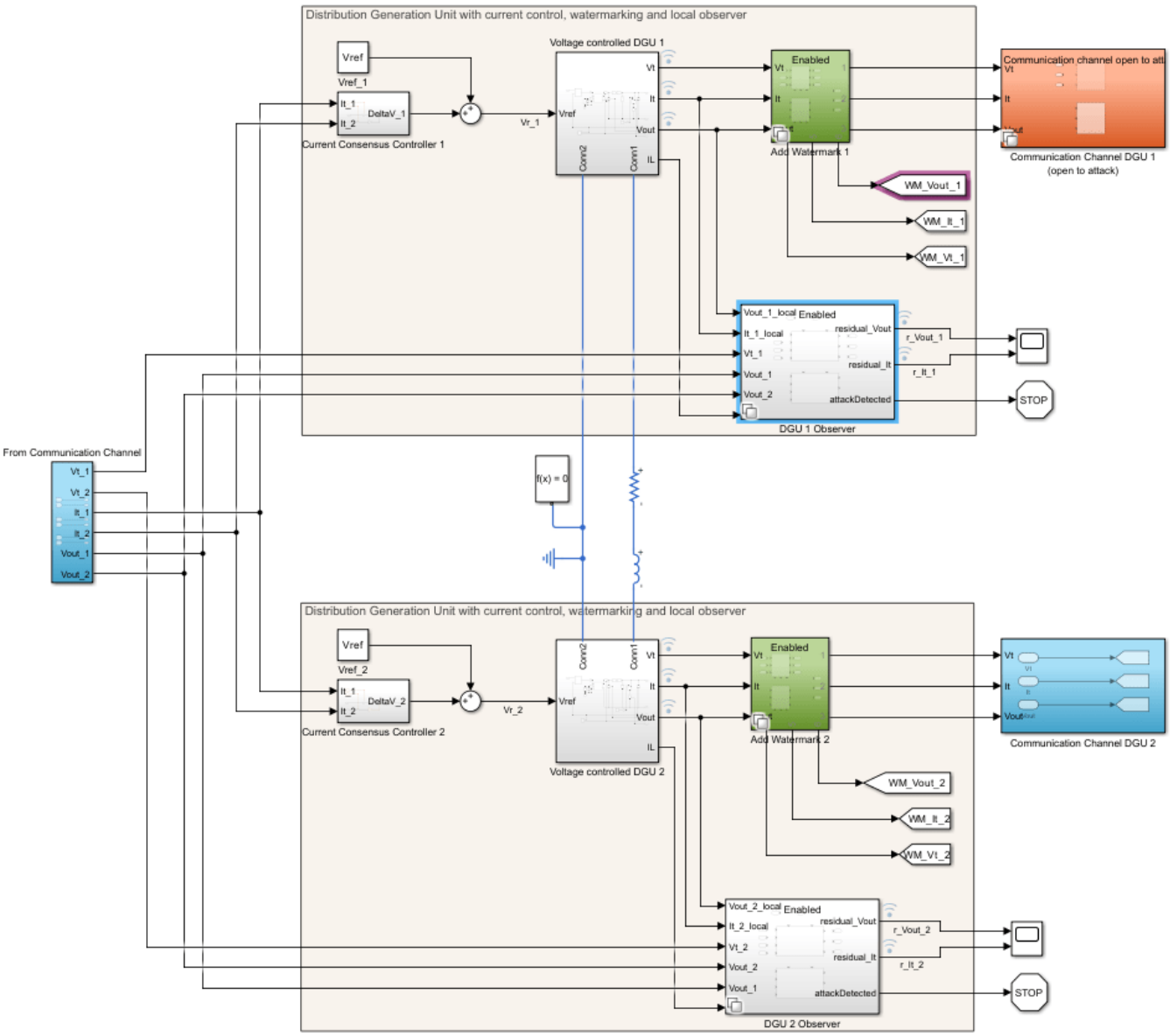
The model computes the estimates of the model states using a linear Kalman filter and the following model.

$$\dot{V}_i(t) = \frac{1}{C_{ti}} I_{ti}(t) - \frac{1}{C_{ti} R_{ij}} V_i(t) + \frac{1}{C_{ti} R_{ij}} V_j(t) - \frac{1}{C_{ti}} I_{Li}(t)$$

$$\dot{I}_{ti}(t) = -\frac{1}{L_{ti}} V_i(t) - \frac{R_{ti}}{L_{ti}} I_{ti}(t) + \frac{1}{L_{ti}} V_{ti}(t)$$

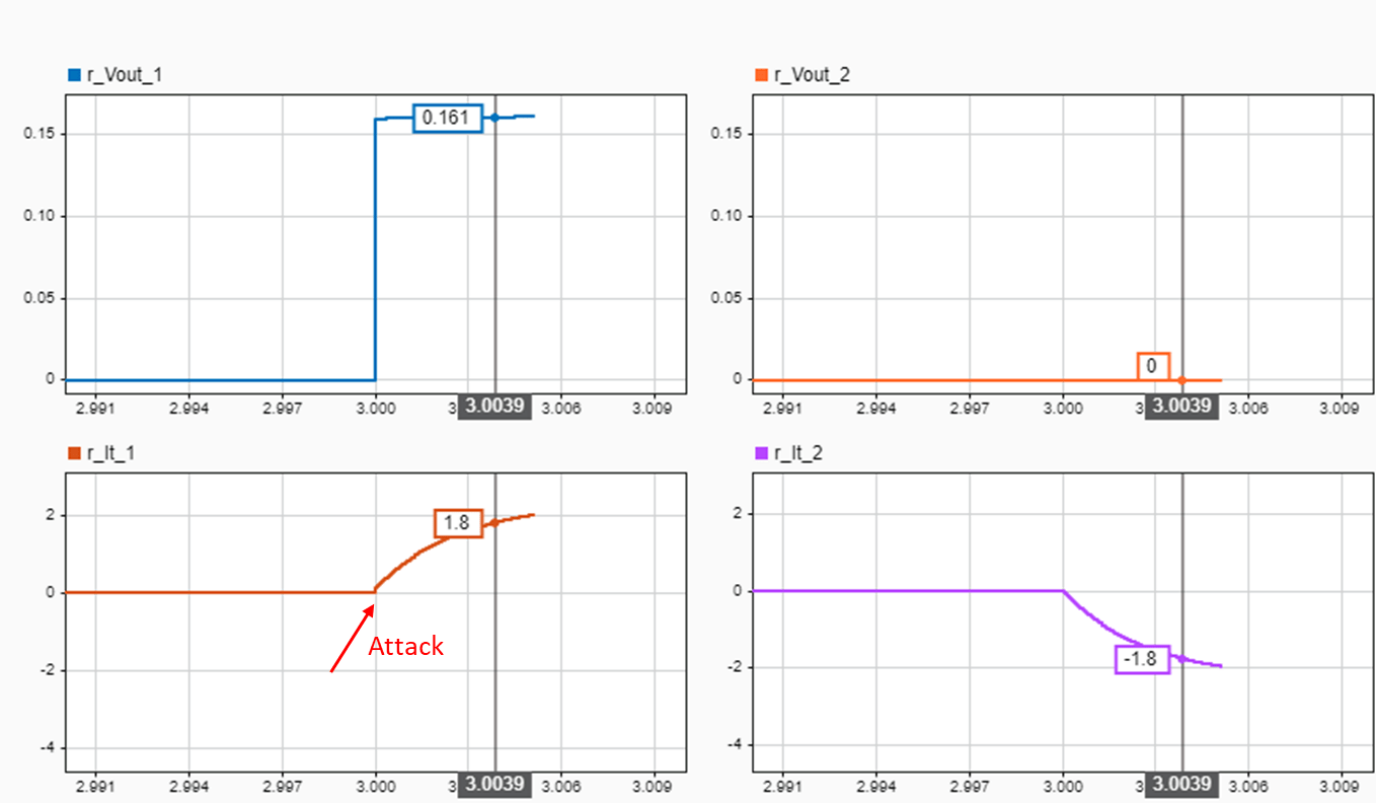
Specify the state-space model parameters.

```
At = [-1/(Rij*Ct),     1/Ct;
      -1/Lt, -Rt/Lt];
Bt = [0;
     1/Lt];
Mt = [-1/Ct;
       0];
Ht = [1 0];

Bj = [1/(Rij*Ct); 0];
BIl = [-1/Ct; 0];

dguPlant = ss(At,[Bt, Bj, BIl],Ht,0);
dguPlant.StateName = {'Vi','Iti'};
dguPlant.InputName = {'Vti','Vj','Ili'};
dguPlant.OutputName = {'Vi'};
dguPlantD = c2d(dguPlant,Ts);
```



Simulate the model.

```
Simulink.sdi.clear
sim("AttackDetectionInDCMicrogrid.slx");
Simulink.sdi.loadView("DCmGAttackDetectionView.mldatx");
```



The residual plots show the response to the attack at `t` = 3 seconds. The difference between the observed voltage in DGU 1 (top left), and the observed line currents in both DGUs (bottom row) is non-zero and crosses specified thresholds within 4 milliseconds. The difference in voltage of DGU 2 (top right) remains zero, indicating that the signals from DGU 1 were attacked.

You can extend this approach to detect attacks in multiple DGUs and use a threshold to trigger corrective actions, such as isolating the unit that is attacked.

Close the model.

```
Simulink.sdi.close
close_system('AttackDetectionInDCMicrogrid')
```

See Also

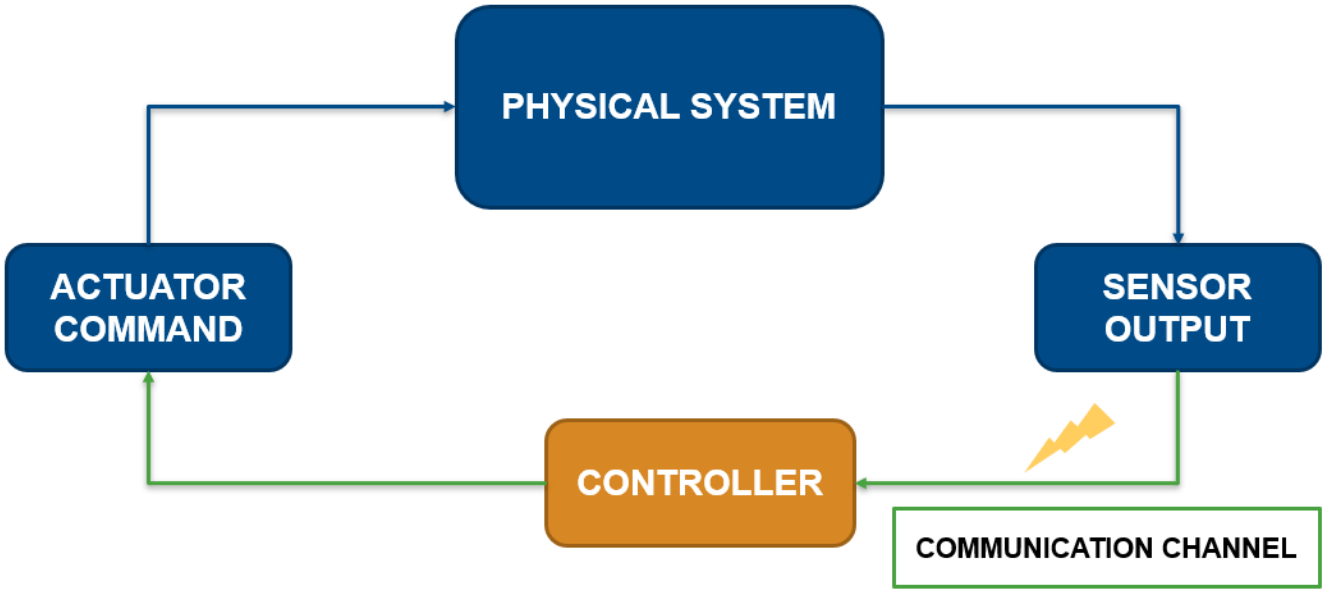
Average-Value DC-DC Converter | Kalman Filter

Related Examples

- “Detect Attack in Cyber-Physical Systems Using Dynamic Watermarking” on page 13-65

Detect Attack in Cyber-Physical Systems Using Dynamic Watermarking

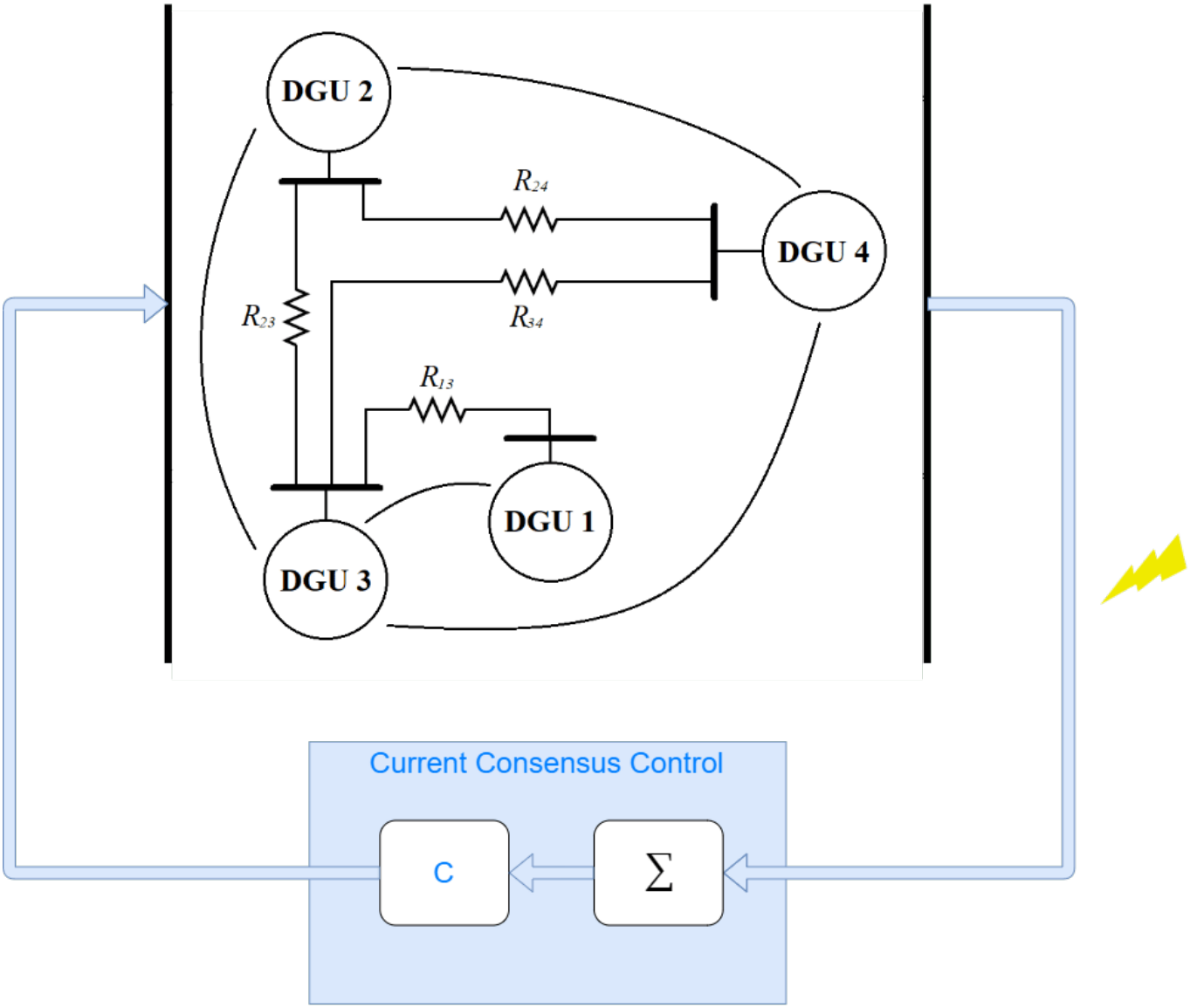
Cyber-physical systems (CPS) consist of communication channels that transmit data between the sensors and actuators in the physical plants and the controller. These channels are susceptible to faults and attacks that disrupt or degrade the operation and can even lead to catastrophic failures in such closed-loop systems.



This example demonstrates the detection of such attacks in a networked DC microgrid system using dynamic watermarking [1]. The technique adds an excitation (watermark) to the secure controller command and detects attacks by monitoring the observed signals for distortions of the watermark.

DC Microgrid Model with Current Consensus Control

Distributed generation units (DGU) in a DC microgrid (DCmG) represent physical nodes in a networked CPS. The central controller adjusts the reference voltages in each DGU to share current loads equally across multiple units.



See “Detect Replay Attacks in DC Microgrids Using Distributed Watermarking” on page 13-57 for more details on the plant model and controller parameters.

Use a sample time of 50 microseconds and a simulation duration of 3 seconds.

```
Ts = 5e-5;
Vin = 100;
Ct = 2.2e-3;
Lt = 1.8e-3;
Rt = 0.2;

Lij = 1.8e-6;
Rij = 0.05;

Vref = 48;
Kp = 0.1;
```

```
Ki = 6.15;
Kd = 4.6e-4;
N = 2.56e3;
Tf = 1/N;
```

The current controller is designed to share the load current in the individual nodes and maintain the ratio of the individual line capacities. Define the capacity of the first DGU to be twice of the other three nodes.

```
K = 150;
Icapacity = [2 1 1 1];

M = ones(4,4);
M(1,1) = -1/Icapacity(1);
M(1,[2 3 4]) = 1/sum(Icapacity([2 3 4]));
M(2,2) = -1/Icapacity(2);
M(2,[1 3 4]) = 1/sum(Icapacity([1 3 4]));
M(3,3) = -1/Icapacity(3);
M(3,[1 2 4]) = 1/sum(Icapacity([1 2 4]));
M(4,4) = -1/Icapacity(4);
M(4,[1 2 3]) = 1/sum(Icapacity([1 2 3]));
```

The load current is applied using Controlled Current Source blocks and is modified at different instances to see the effect on all nodes.

```
ILoadInitial = [5, 5, 5, 5];
ILoadChanged = [2, 4, 1.5, 3];
tLoadChanged = [0.5, 0.75, 1.0, 1.25];
```

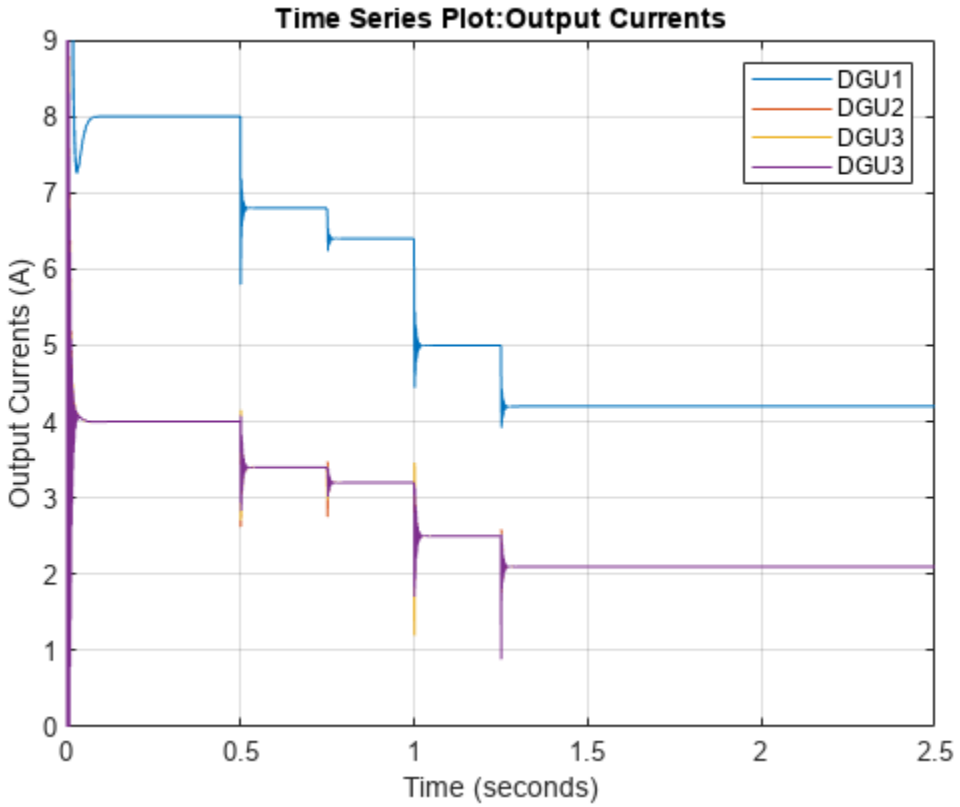
Simulate model in nominal condition

Set the variables that enable (`true`) or disable (`false`) the watermarking, attack, and attack detection subsystems.

```
enableWatermark = false;
enableAttack = false;
enableAttackDetection = false;
```

Simulate the model to see the effect of load changes. You can observe that the line currents of the different DGUs maintain the ratio of their capacities with every load change.

```
modelName = 'DynamicWatermarkingInDCMicroGrid';
open_system(modelName);

simInput = Simulink.SimulationInput(modelName);
out = sim(simInput);

figure;
plot(out.DGU_LineCurrents);
ylim([0 9]);
grid on
legend('DGU1','DGU2','DGU3','DGU3');
```



Linearize Model

The monitoring strategy uses the plant model for detecting changes in the closed-loop signals. Use `linearize` (Simulink Control Design) to obtain a linear approximation of the DCmG at a steady-state operating point.

You can use `findop` (Simulink Control Design) to get snapshot operating point after system is at steady-state, at t = 0.3 seconds. The local sample-based solver in the Solver Configuration block is disabled to support linearization operations.

```
op = findop(modelName,0.3);
```

Define the input to the plant as the output of the controller, and the load current inputs that are needed to prevent false detections of load disturbance changes as attacks or faults. The outputs of the plant are the line currents of each DGU.

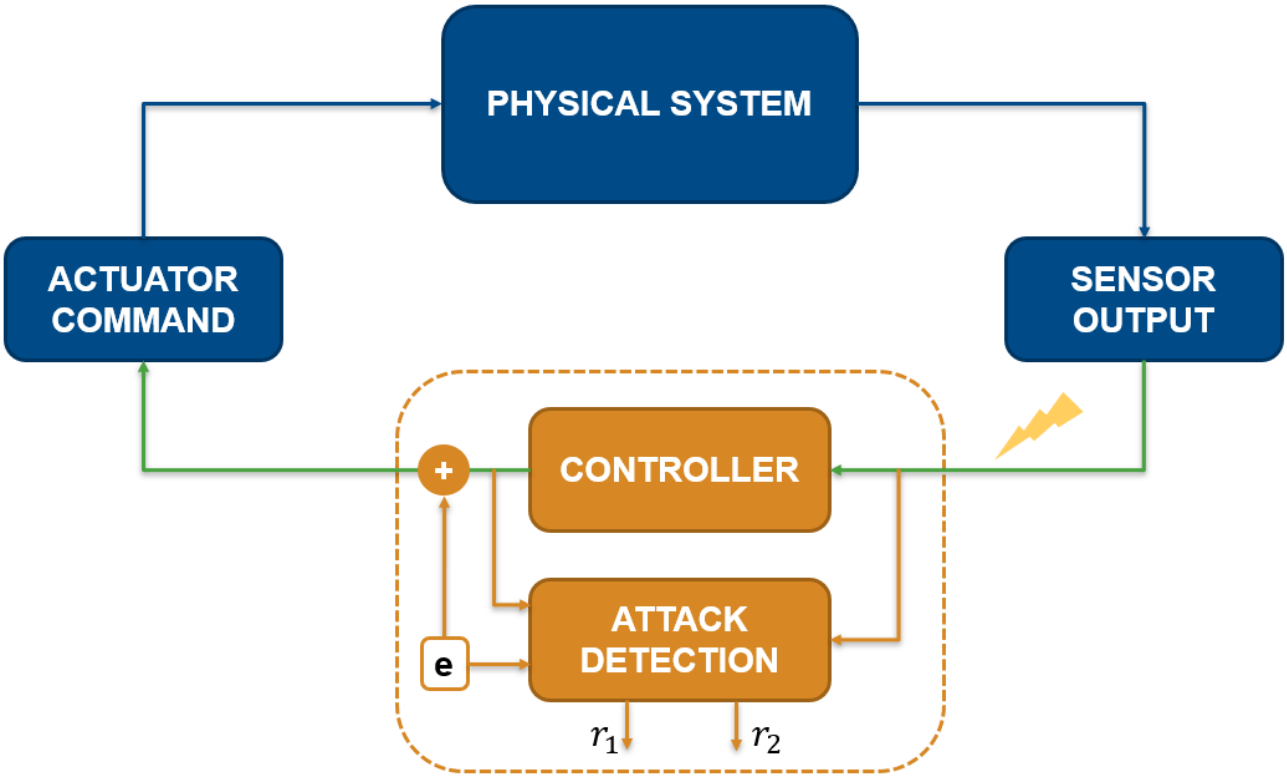
```
io(1) = linio([modelName,'/Current Consensus Controller'],1,'openinput');
io(2) = linio([modelName,'/DC Microgrid/ILoad_1'],1,'openinput');
io(3) = linio([modelName,'/DC Microgrid/ILoad_2'],1,'openinput');
io(4) = linio([modelName,'/DC Microgrid/ILoad_3'],1,'openinput');
io(5) = linio([modelName,'/DC Microgrid/ILoad_4'],1,'openinput');
io(6) = linio([modelName,'/Select line currents'],1,'openoutput');
```

Linearize the plants based on the defined analysis and operating points. Store the offsets information for use in the attack detection strategy.

```
opts = linearizeOptions(StoreOffsets=true,SampleTime=Ts);
[linsys,~,info] = linearize(modelName,io,op,opts);
```

Dynamic Watermarking

This model adds a random excitation signal, $e[t]$, drawn from a Gaussian distribution with zero mean, to the voltage command that the controller to each physical node communicates. Since the controller is secure, this signal is not known to the external agents. You must model the power of the excitation signal carefully based on voltage levels such that it does not modify the desired operation of the closed-loop control. However, it should be high enough to differentiate between attacks and disturbance due to noise.



The sensor measurements are correlated with the known, secure watermark, $e[t]$ with signal power σ_e^2. Statistical tests are used to develop conditions that should be satisified in when sensors are reporting the true output. Any distortion in the signals would result in the computed sequence of measurements cross the threshold indicating an attack.

Let the states of the multiple input, multiple output plant be represented by the following state-space model.

$$x[t+1] = Ax[t] + B(u[t] + e[t])$$

where $u[t]$ represents the controller command and $e[t]$ is the injected watermark.

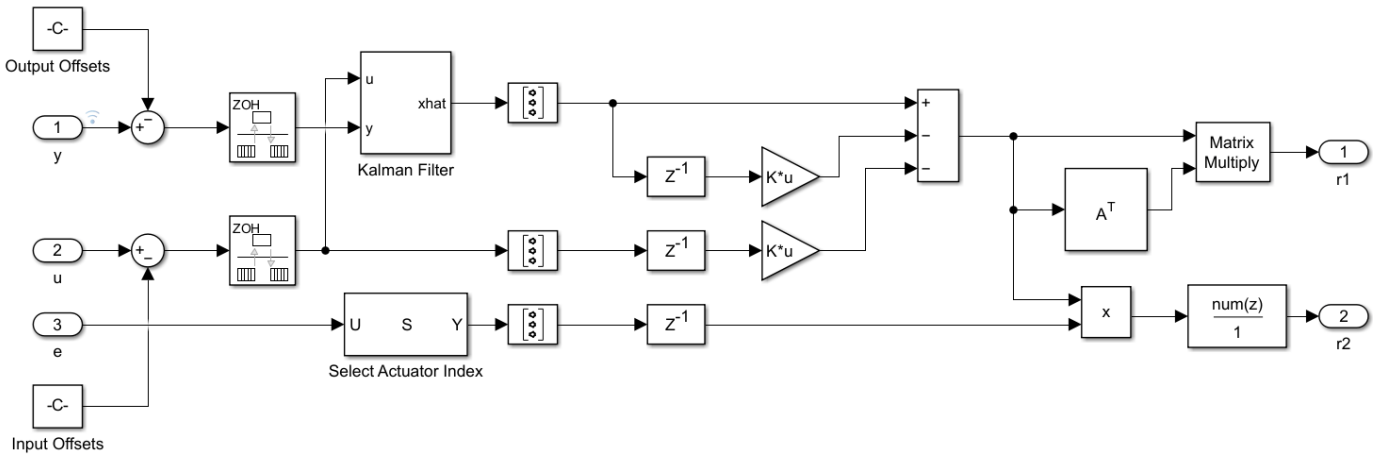
As shown in [1], the statisical tests are then defined as follows.

$$r_1 = \lim_{T \to \inf} \frac{1}{T} \sum_{k=0}^{T-1} (x[k+1] - Ax[k] - Bu[k])(x[k+1] - Ax[k] - Bu[k])^T = \sigma_e^2 BB^T$$

$$r_2 = \lim_{T \to \inf} \frac{1}{T} \sum_{k=0}^{T-1} e_i[k](x[k+1] - Ax[k] - Bu[k]) = B_{\cdot,i}\sigma_e^2$$

r_2 is generated for the i^{th} actuator, and $B_{\cdot,i}$ represents the i^{th} column of B.

For partially observed systems, such as the DCmG used in this example, use a Kalman Filter block with the linearized model `linsys` to estimate the state vector. The offset information stored in `info` is subtracted from the input and output measurements before being used to generate the estimated states and the statistical test outputs r_1 and r_2.




```
enableWatermark = true;
watermarkVariance = [1e-6, 1e-6, 1e-6, 1e-6];
```

Attack Detection

The communication channel is attacked at t = 2 seconds and the line current measurements are modified to change the balance of the DGUs in the networked cyber-physical system. The model settings is configured to start at steady state.

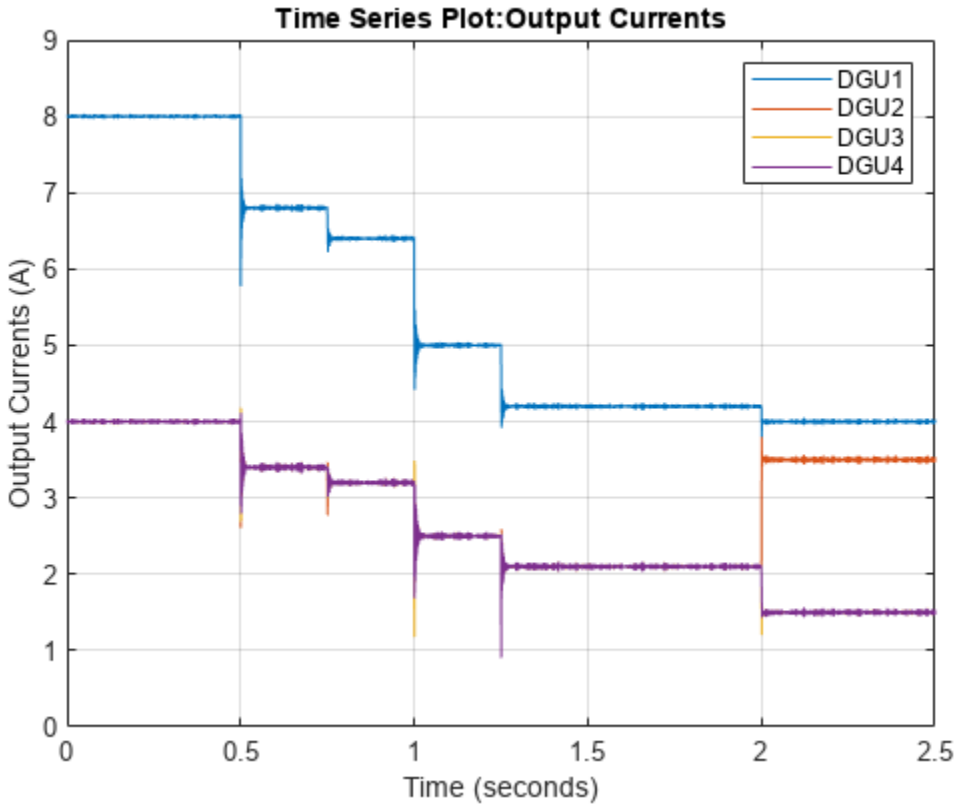
The detection tests r_1 and r_2 generate signals of dimensions Nx-by-Nx and Nx-by-1, respectively, where Nx is the order of the linearized plant system. Signals at specific indices, defined in `detectionTest1_StateIndex` and `detectionTest2_StateIndex`, are extracted and plotted to observe the detection response when an attack occurs.

```
enableAttack = true;
tAttack = 2;
IAttack = [1 0 -1 0 1 0 1 0];

enableAttackDetection = true;
actuatorIndex = 1;
detectionTest1_StateIndex = [1 1];
detectionTest2_StateIndex = 1;

simInput = setModelParameter(simInput,'LoadExternalInput','on');
simInput = setModelParameter(simInput,'ExternalInput','getinputstruct(op)');
simInput = setModelParameter(simInput,'LoadInitialState','on');
simInput = setModelParameter(simInput,'InitialState','getstatestruct(op)');
```

```
out = sim(simInput);
```

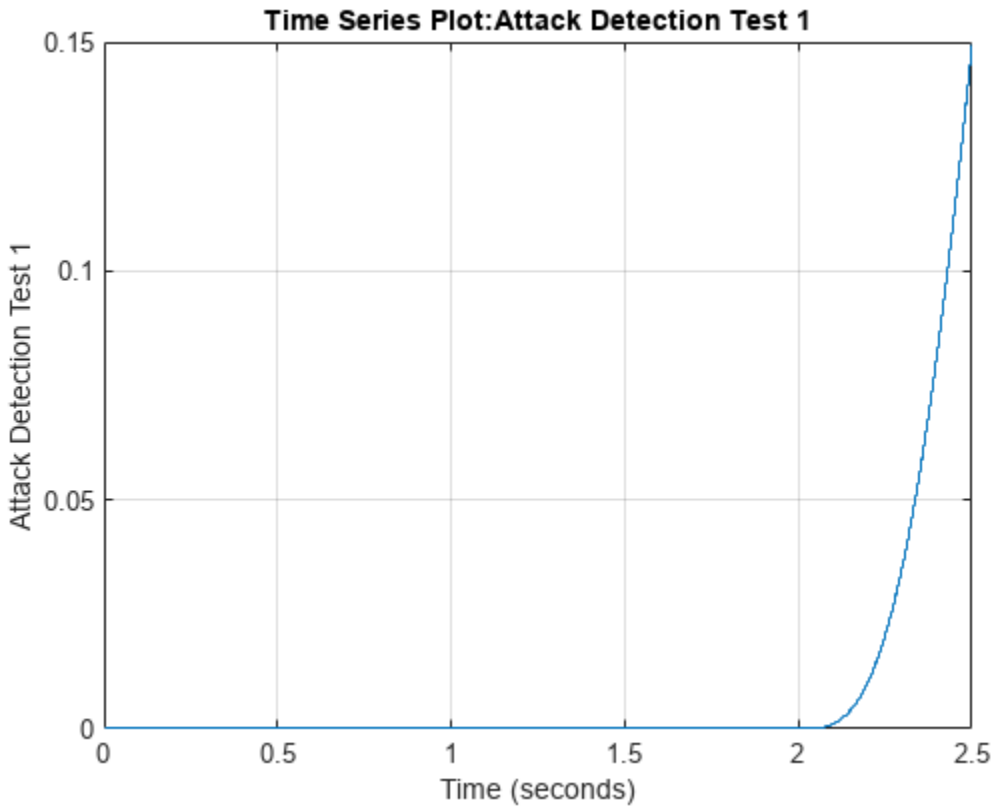
As seen in the output current plot below, the output of DGU 2 increases at the expense of other units when the system is attacked.

```
figure;
plot(out.DGU_LineCurrents);
grid on;
legend('DGU1','DGU2','DGU3','DGU4');
```

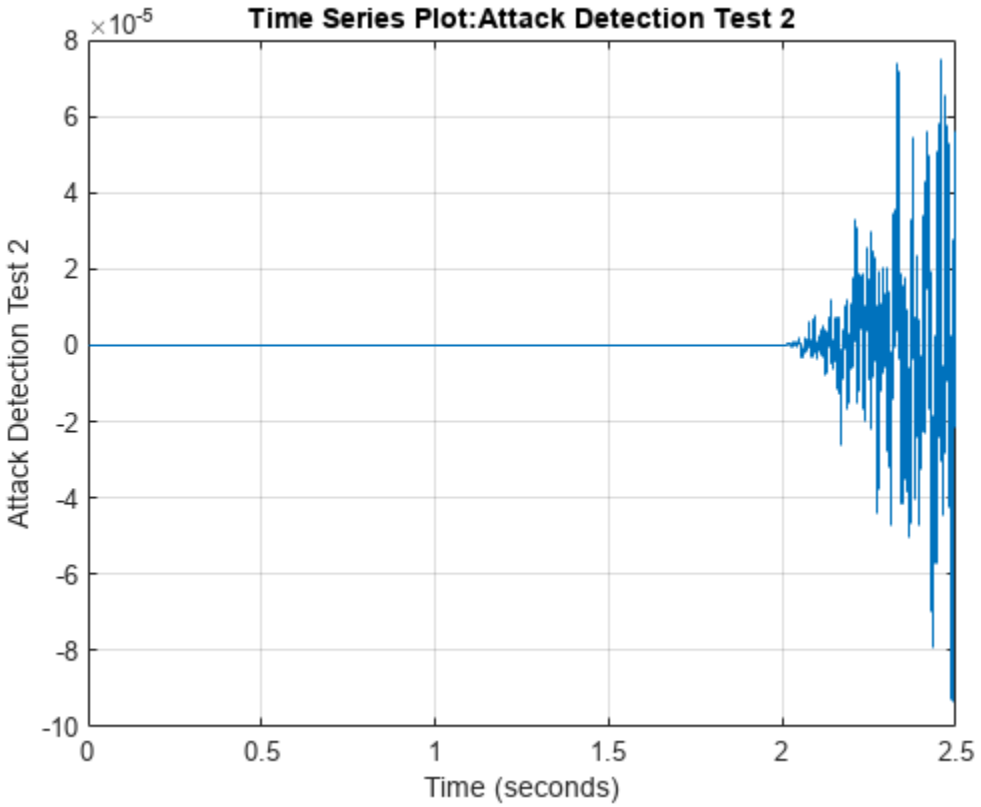


The detection tests generate two sequences. The first element of each sequence is extracted and shown below. Both tests indicate an attack after t = 2 seconds when it does not satisfy the condition based on the plant model.

```
figure;
plot(out.AttackDetectionTest1);
grid on
```




```
figure;
plot(out.AttackDetectionTest2);
grid on
```



References

[1] B. Satchidanandan and P. R. Kumar, "Dynamic Watermarking: Active Defense of Networked Cyber–Physical Systems," in *Proceedings of the IEEE*, vol. 105, no. 2, pp. 219-240, Feb. 2017, doi: 10.1109/JPROC.2016.2575064.

See Also

Average-Value DC-DC Converter | Kalman Filter | `linearize`

Related Examples

- “Detect Replay Attacks in DC Microgrids Using Distributed Watermarking” on page 13-57

Control System Tuning

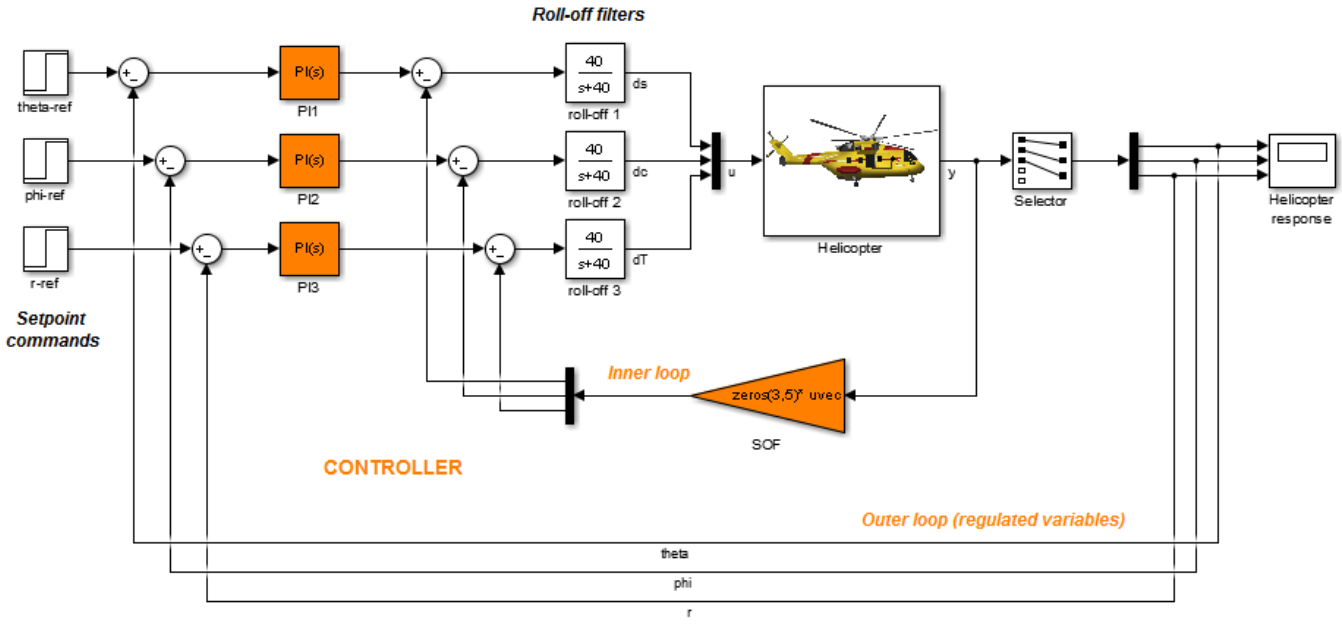
Control System Tuning

- “Automated Tuning Overview” on page 14-3
- “Choosing an Automated Tuning Approach” on page 14-4
- “Automated Tuning Workflow” on page 14-6
- “Specify Control Architecture in Control System Tuner” on page 14-7
- “Open Control System Tuner for Tuning Simulink Model” on page 14-10
- “Specify Operating Points for Tuning in Control System Tuner” on page 14-11
- “Specify Blocks to Tune in Control System Tuner” on page 14-17
- “View and Change Block Parameterization in Control System Tuner” on page 14-19
- “Setup for Tuning Control System Modeled in MATLAB” on page 14-25
- “How Tuned Simulink Blocks Are Parameterized” on page 14-26
- “Specify Goals for Interactive Tuning” on page 14-28
- “Quick Loop Tuning of Feedback Loops in Control System Tuner” on page 14-33
- “Quick Loop Tuning” on page 14-41
- “Step Tracking Goal” on page 14-44
- “Step Rejection Goal” on page 14-49
- “Transient Goal” on page 14-53
- “LQR/LQG Goal” on page 14-58
- “Gain Goal” on page 14-62
- “Variance Goal” on page 14-66
- “Reference Tracking Goal” on page 14-70
- “Overshoot Goal” on page 14-75
- “Disturbance Rejection Goal” on page 14-79
- “Sensitivity Goal” on page 14-84
- “Weighted Gain Goal” on page 14-88
- “Weighted Variance Goal” on page 14-91
- “Minimum Loop Gain Goal” on page 14-95
- “Maximum Loop Gain Goal” on page 14-100
- “Loop Shape Goal” on page 14-105
- “Margins Goal” on page 14-110
- “Passivity Goal” on page 14-114
- “Conic Sector Goal” on page 14-118
- “Weighted Passivity Goal” on page 14-123
- “Poles Goal” on page 14-127
- “Controller Poles Goal” on page 14-131
- “Manage Tuning Goals” on page 14-134

- “Generate MATLAB Code from Control System Tuner for Command-Line Tuning” on page 14-135
- “Interpret Numeric Tuning Results” on page 14-138
- “Visualize Tuning Goals” on page 14-141
- “Create Response Plots in Control System Tuner” on page 14-147
- “Examine Tuned Controller Parameters in Control System Tuner” on page 14-152
- “Compare Performance of Multiple Tuned Controllers” on page 14-154
- “Create and Configure slTuner Interface to Simulink Model” on page 14-157
- “Stability Margins in Control System Tuning” on page 14-161
- “Tune Control System at the Command Line” on page 14-166
- “Speed Up Tuning with Parallel Computing Toolbox Software” on page 14-167
- “Validate Tuned Control System” on page 14-168
- “Extract Responses from Tuned MATLAB Model at the Command Line” on page 14-171

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**

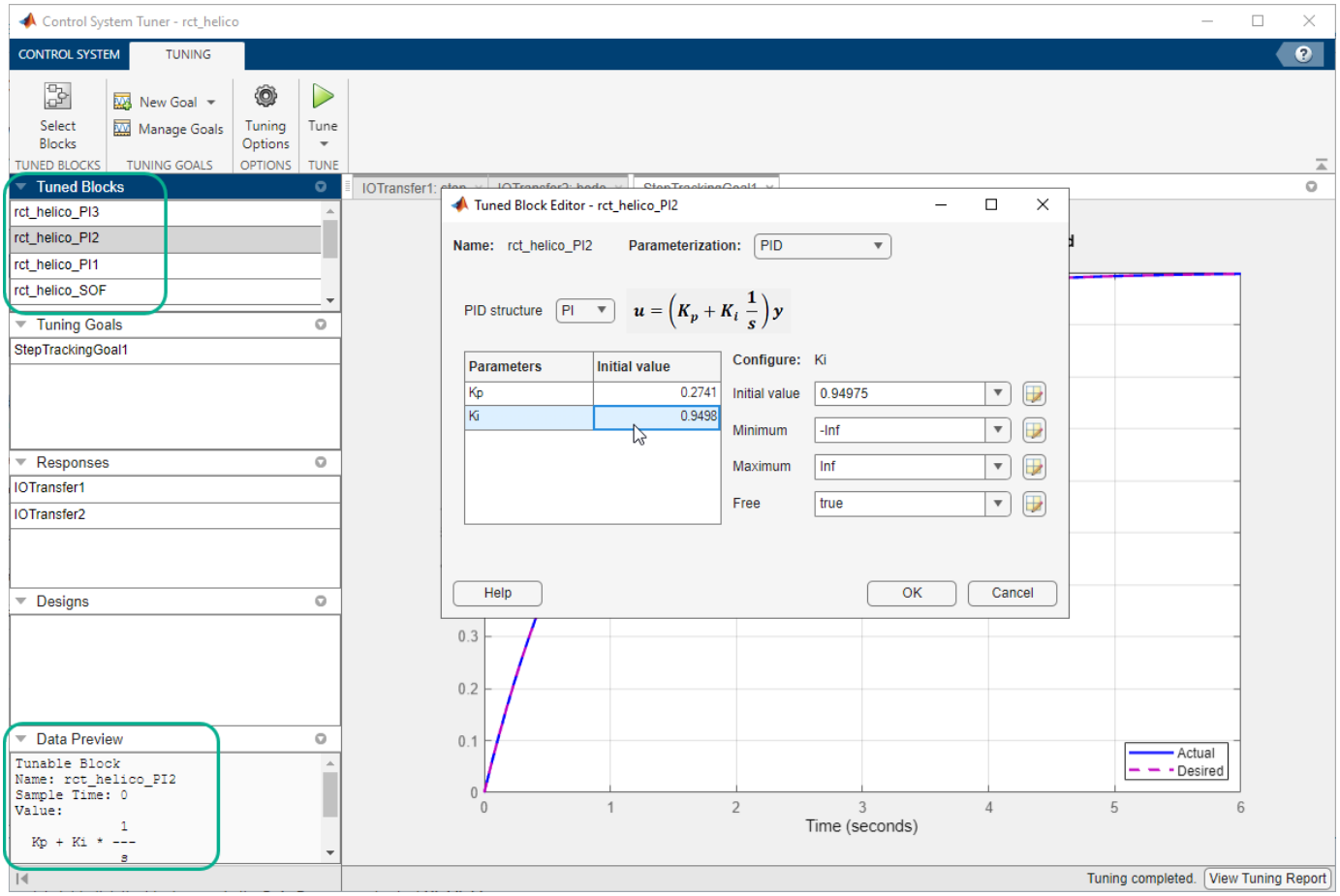
More About

- “Choosing an Automated Tuning Approach” on page 14-4
- “Automated Tuning Workflow” on page 14-6

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 14-6

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 14-7.

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 Setup for Tuning Simulink Models 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).

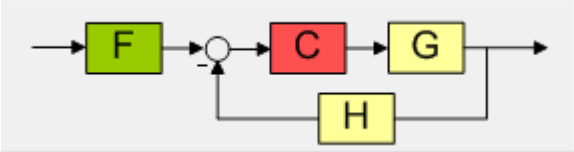
4. Tune the model. Use the `systune` command or **Control System Tuner** to optimize the tunable parameters of your control system to best meet your specified tuning goals. Then, analyze the tuned system responses and validate the design. Whether at the command line or in **Control System Tuner**, you 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:



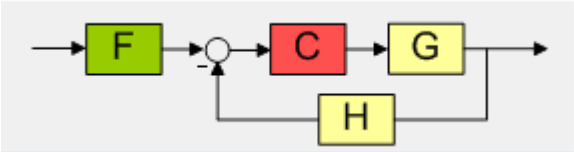
G is the plant model, and *H* the sensor dynamics. These are usually the fixed components of the control system. The prefilter *F* and feedback controller *C* are the tunable elements. Because control systems are so conveniently expressed in this block diagram form, these elements are referred to as 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**.



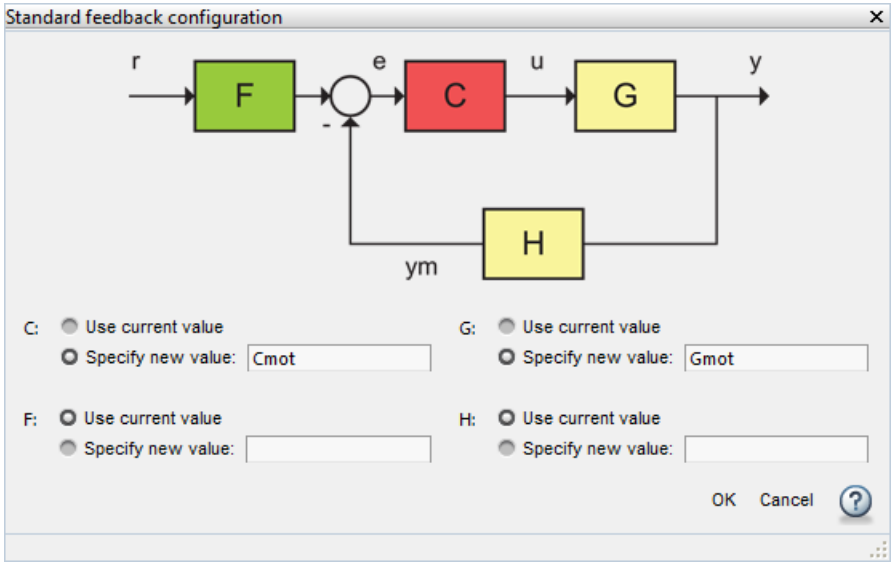
For example, suppose you have a DC motor for which you want to tune a PID controller. The response of the motor is modeled as $G(s) = 1/(s + 1)^2$. Create a fixed LTI model representing the plant, and a tunable PID controller model.

```
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**.

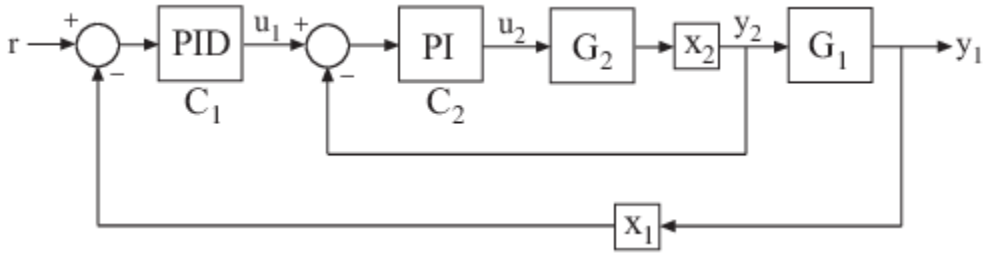
The default value for the sensor dynamics is a fixed unity-gain transfer function. The default value for the filter *F* is a tunable gain block.

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 14-10.

See Also

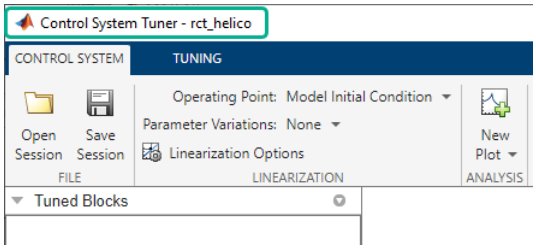
More About

- “Building Tunable Models” on page 17-9
- “Specify Blocks to Tune in Control System Tuner” on page 14-17
- “Specify Goals for Interactive Tuning” on page 14-28

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 14-11
- “Specify Blocks to Tune in Control System Tuner” on page 14-17

More About

- “Automated Tuning Workflow” on page 14-6

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:

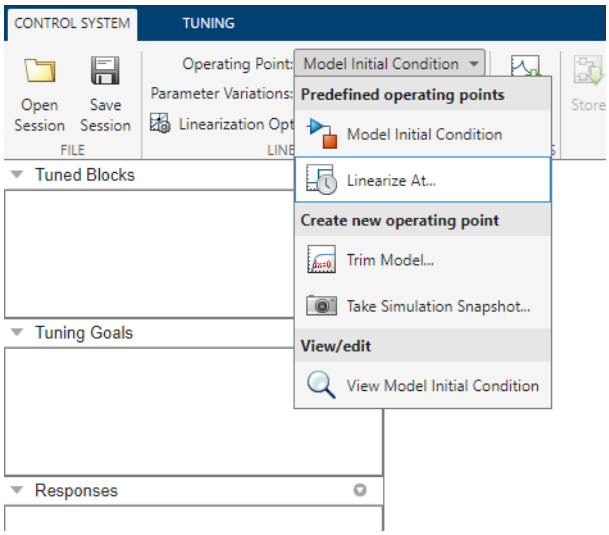
- Simulation snapshot time. **Control System Tuner** simulates the model for the amount of time you specify, and linearizes using the state values at that time. Simulation snapshot linearization is useful, for instance, when you know your model reaches an equilibrium state after a certain simulation 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” (Simulink Control Design) and “Compute Steady-State Operating Points” (Simulink Control Design).

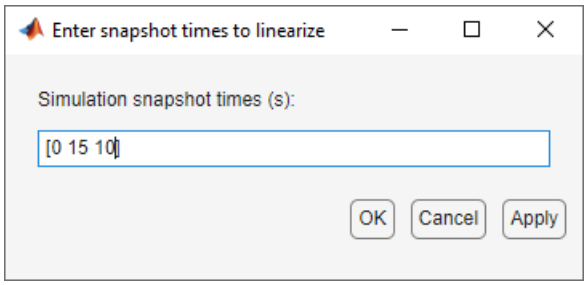
Linearize at Simulation Snapshot Times

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**.

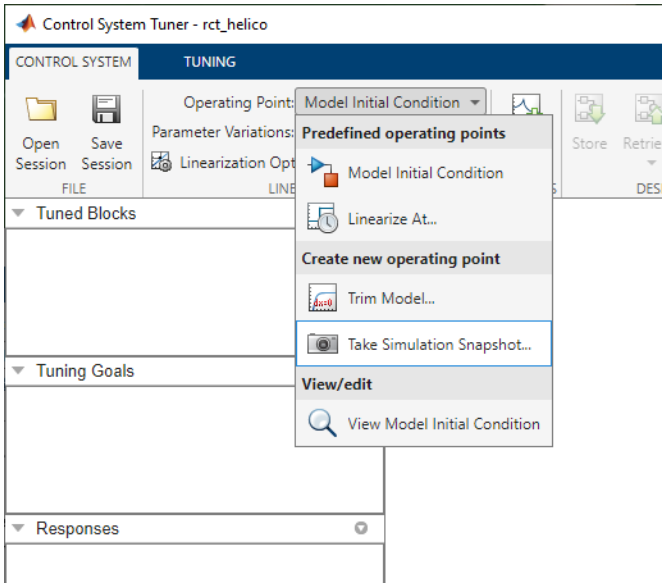


When you are ready to analyze system responses or tune your model, **Control System Tuner** computes linearizations at the specified snapshot times. If you enter multiple snapshot times, **Control System Tuner** computes an array of linearized models, and displays analysis plots that reflect the multiple 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.

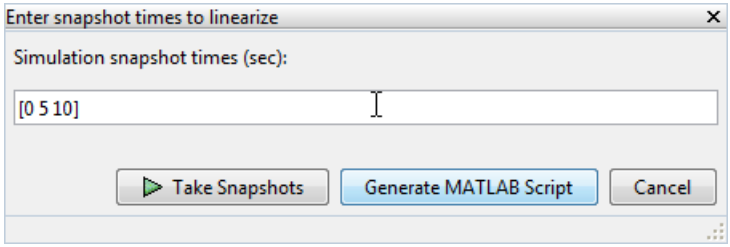
Compute Operating Points at Simulation Snapshot Times

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`.



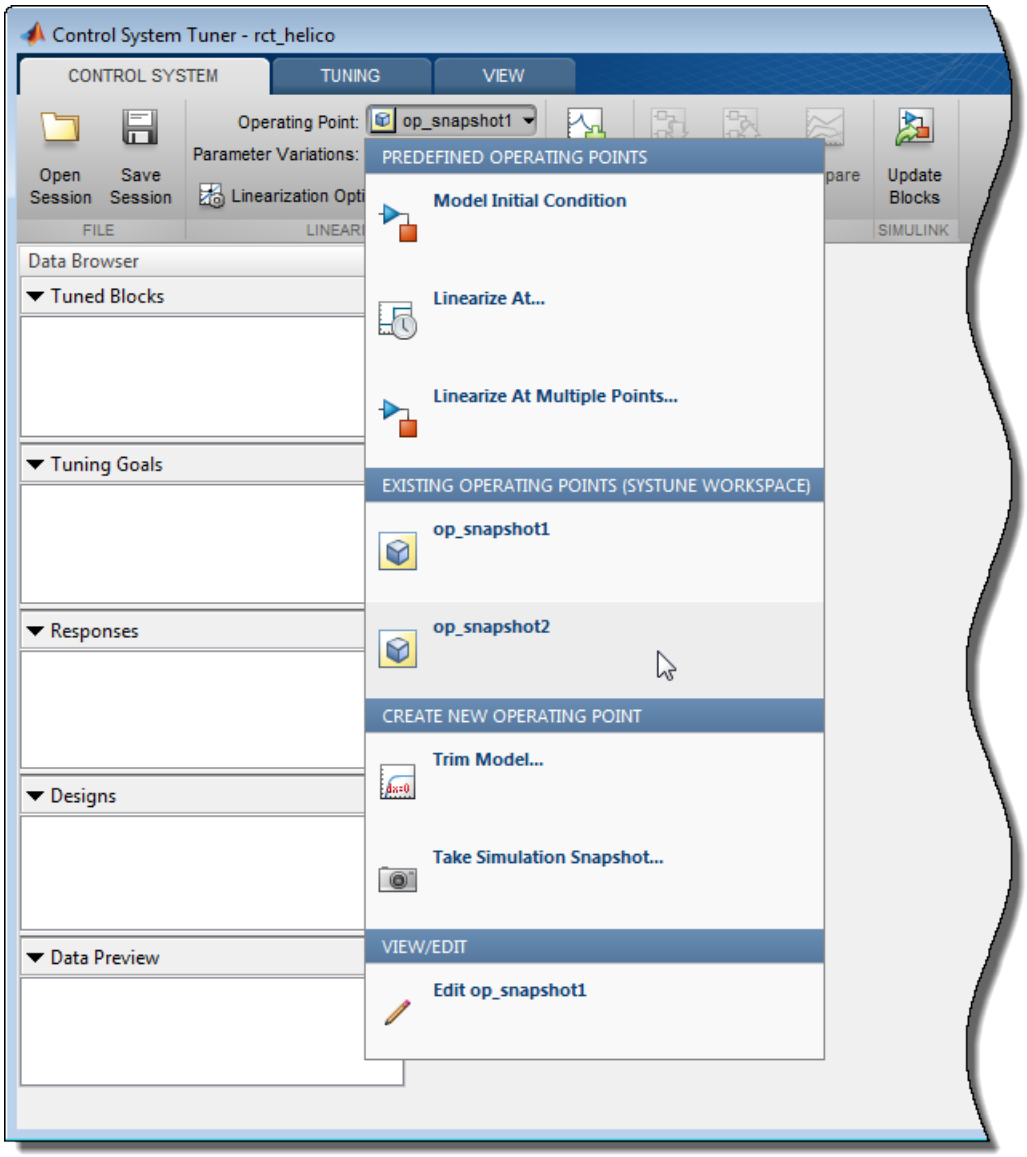
In the **Enter snapshot times to linearize** dialog box, in the **Simulation snapshot times** field, enter one or more simulation snapshot times. Enter multiple snapshot times as a vector.



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.

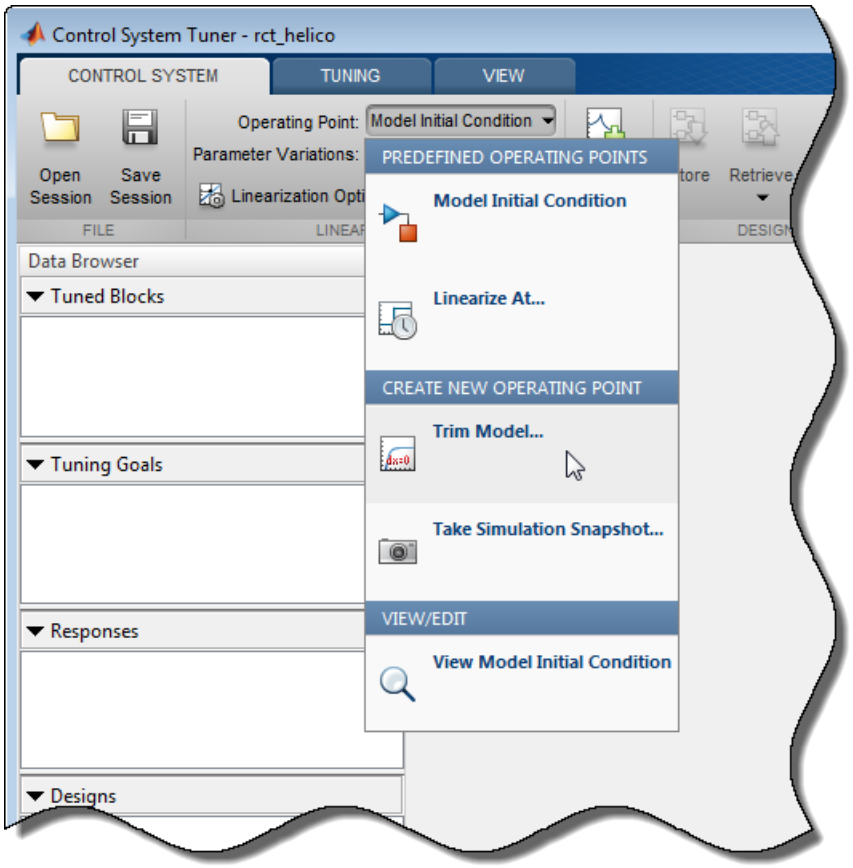


If you entered a vector of snapshot times, all the resulting operating points are stored together in an operating-point vector. You can use this vector to tune a control system at several operating points simultaneously.

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`.

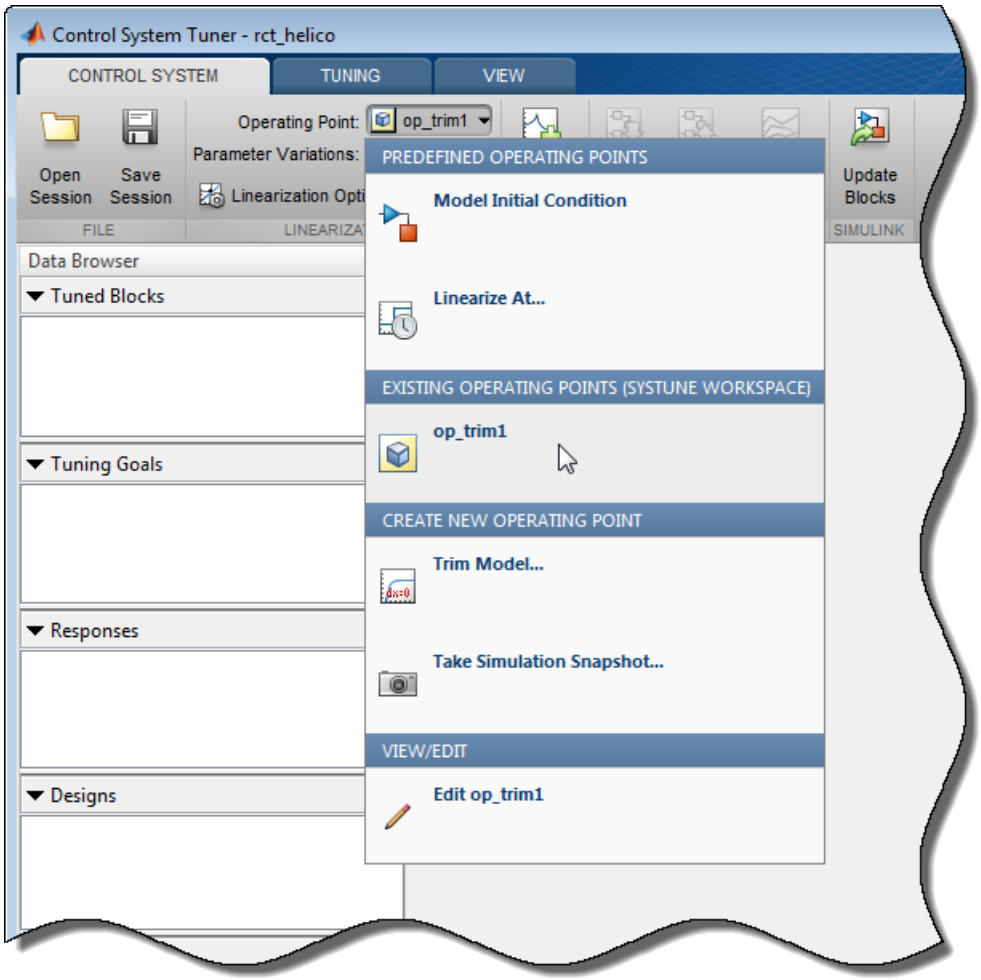


In the **Trim the model** dialog box, enter the specifications for the steady-state state values at which you want to find an operating point.

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” (Simulink Control Design).

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

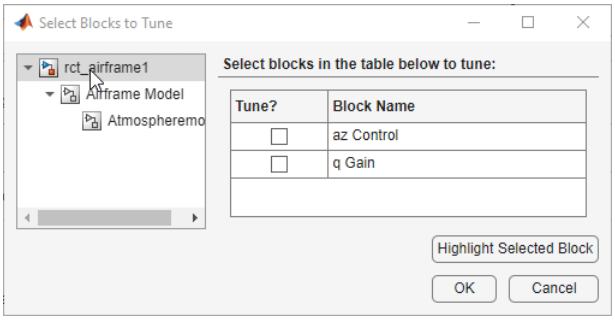
Related Examples

- “Specify Blocks to Tune in Control System Tuner” on page 14-17
- “Robust Tuning Approaches” (Robust Control Toolbox)

Specify Blocks to Tune in Control System Tuner

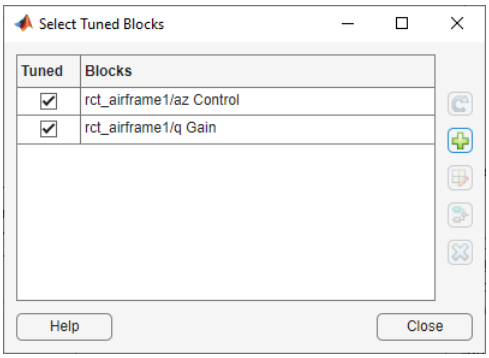
To select which blocks of your Simulink model to tune in **Control System Tuner**:

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 14-19

More About

- “How Tuned Simulink Blocks Are Parameterized” on page 14-26

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:

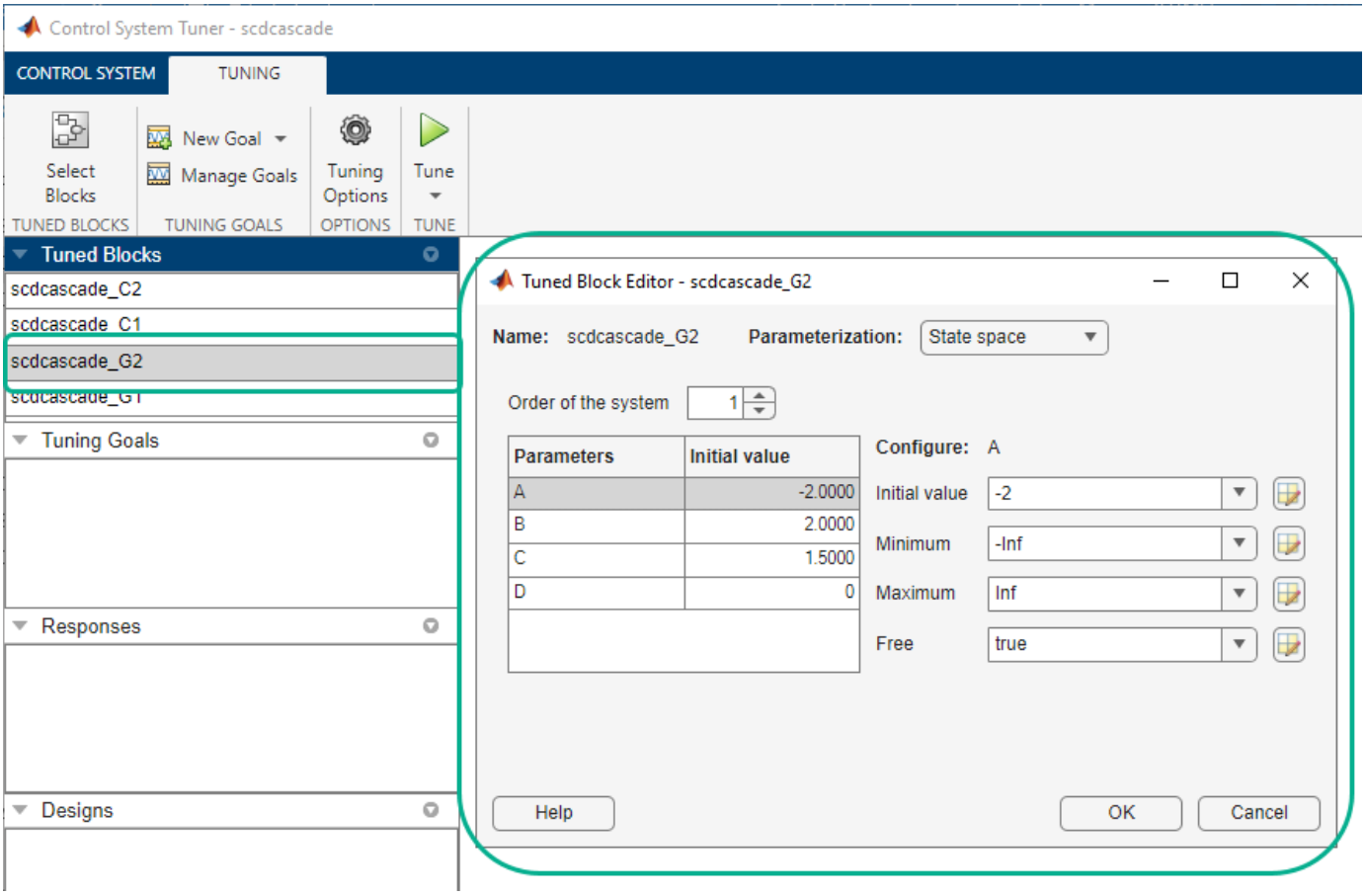
  $$u = K_p + K_i\frac{1}{s}.$$

  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 .

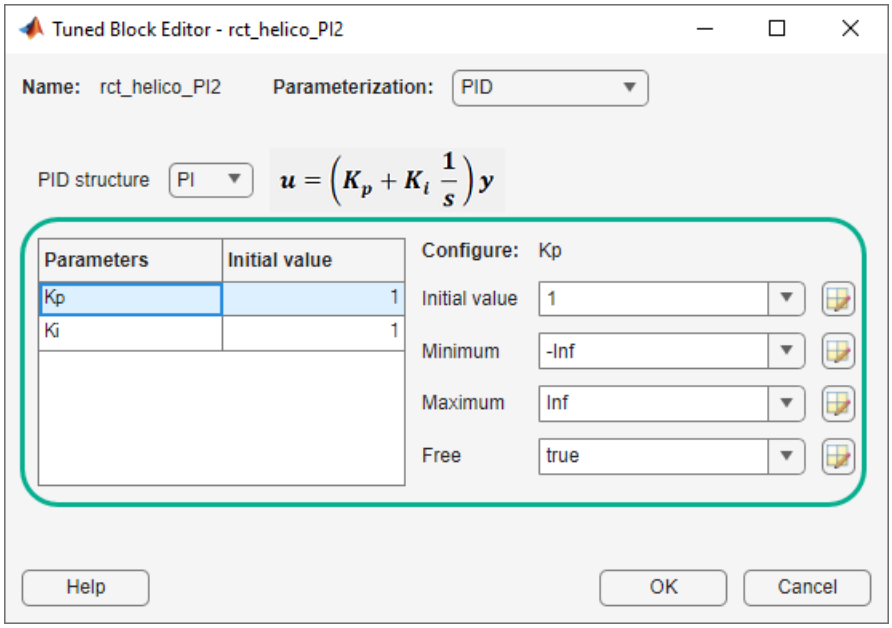
Note To find a tuned block in the Simulink model, right-click the block name in the **Data Browser** and select **Highlight**.

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 14-7, 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.

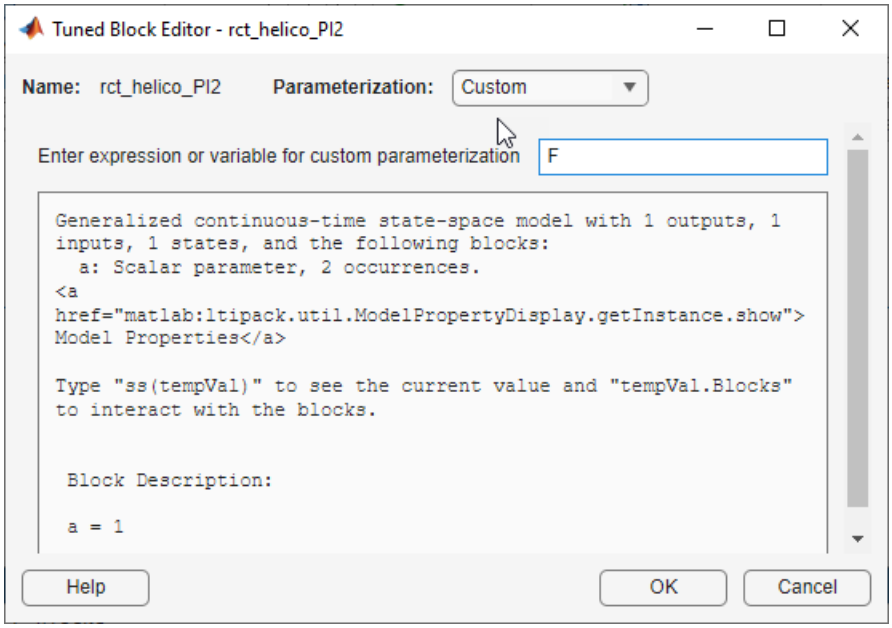
For example, suppose you want to specify a tunable low-pass filter, $F = a/(s + a)$, where a is the tunable parameter. First, at the MATLAB command line, create a tunable `genss` model that represents the low-pass filter structure.

```
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.
```

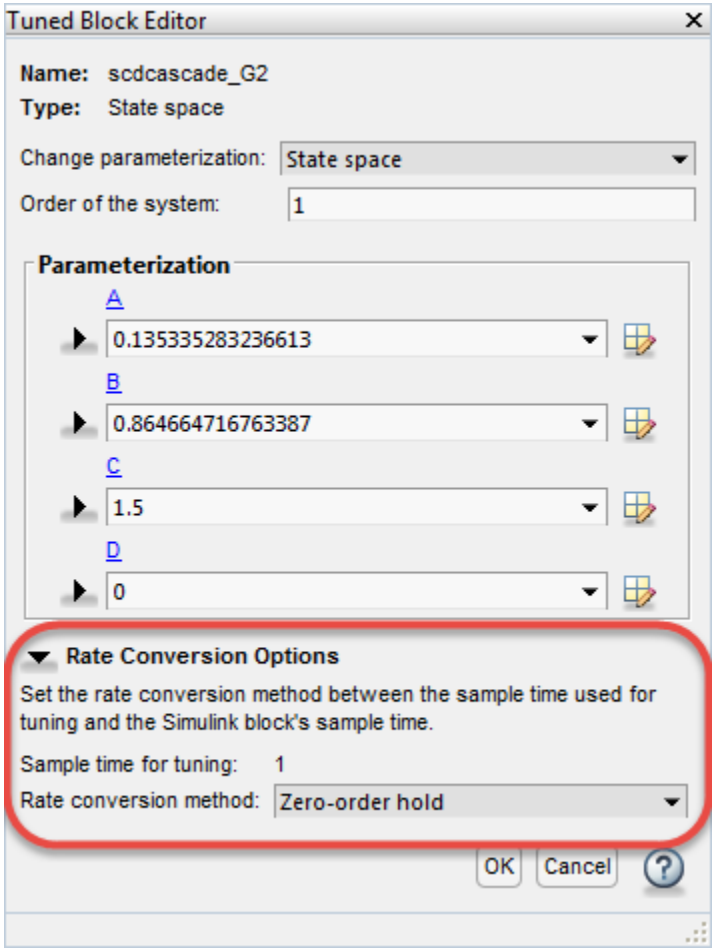
Then, in the Tuned Block Editor, enter `F` in the **Parameterization** area.



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” on page 2-19.

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” on page 5-20.

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 14-17

More About

- “How Tuned Simulink Blocks Are Parameterized” on page 14-26

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 14-28

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” (Simulink Control Design) 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 14-19.
- At the command line, use `getBlockParam` to view the current block parameterization. Use `setBlockParam` to change the block parameterization.

Specify Goals for Interactive Tuning

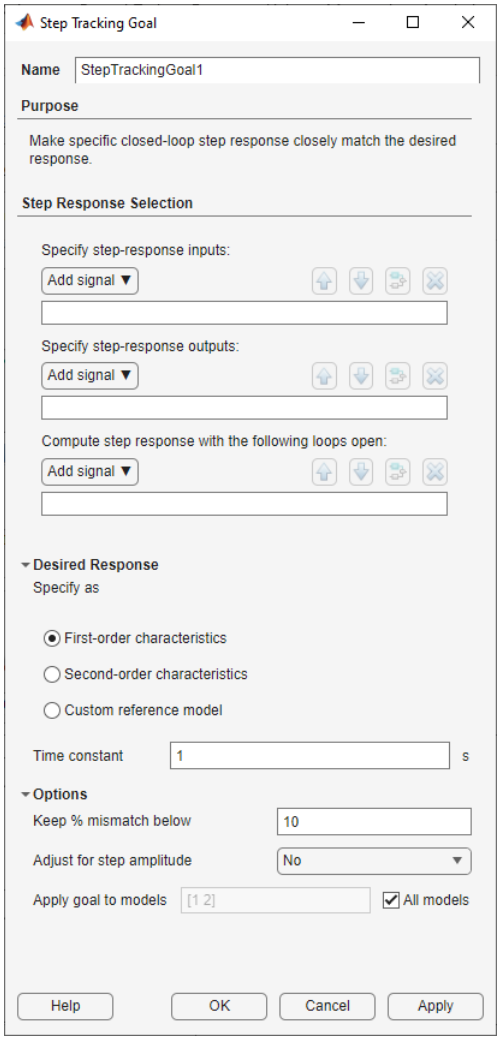
This example shows how to specify your tuning goals for automated tuning in **Control System Tuner**.

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.

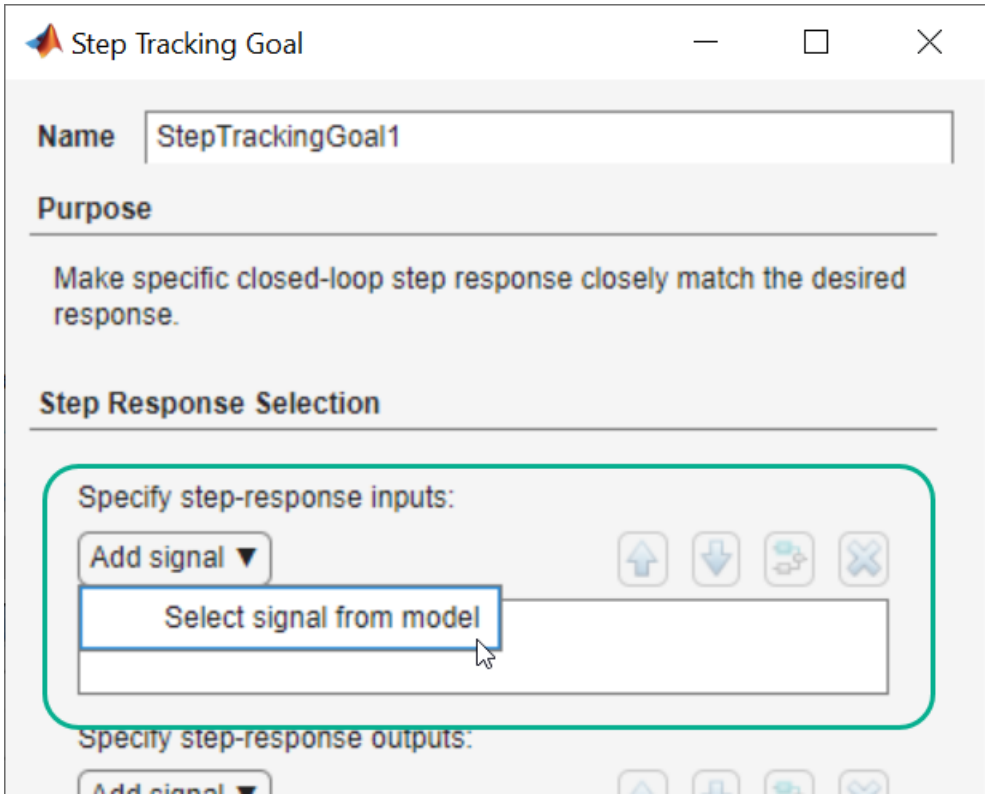


Choose Signal Locations for Evaluating Tuning Goal

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.)

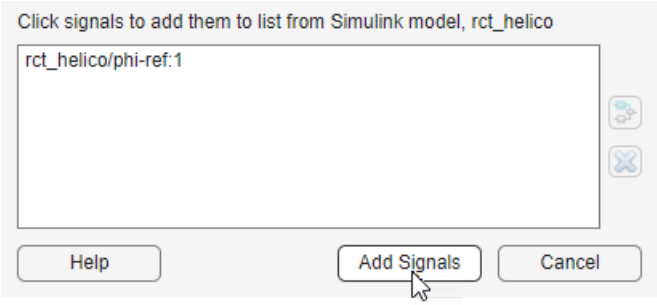
Under **Specify step-response inputs**, click **Add signal to list**. A list of available input locations appears.

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**.



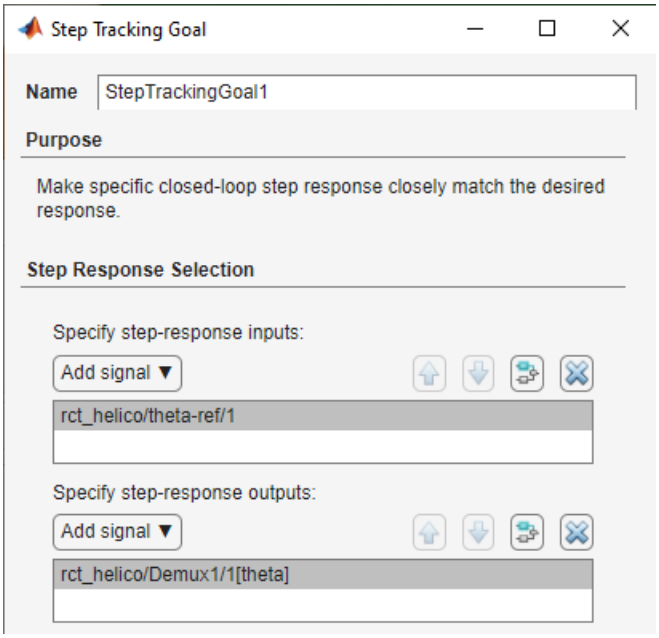
In the **Select signals** dialog box, build a list of the signals you want. To do so, click signals in the Simulink model editor. The signals that you click appear in the **Select signals** dialog box. Click one signal to create a SISO tuning goal, and click multiple signals to create a MIMO tuning goal.

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.

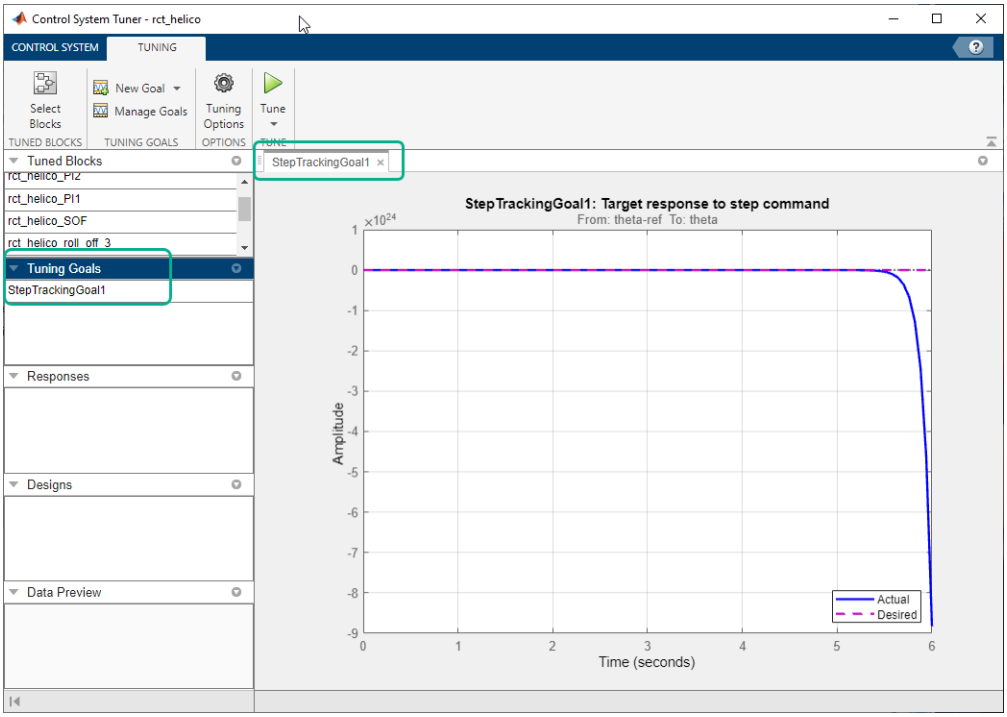
Define Other Specifications of the Tuning Goal

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.

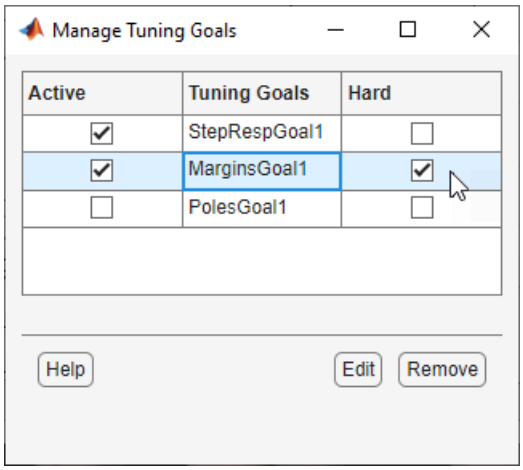
Activate the Tuning Goal for Tuning

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 14-134
- “Quick Loop Tuning of Feedback Loops in Control System Tuner” on page 14-33
- “Create Response Plots in Control System Tuner” on page 14-147

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.

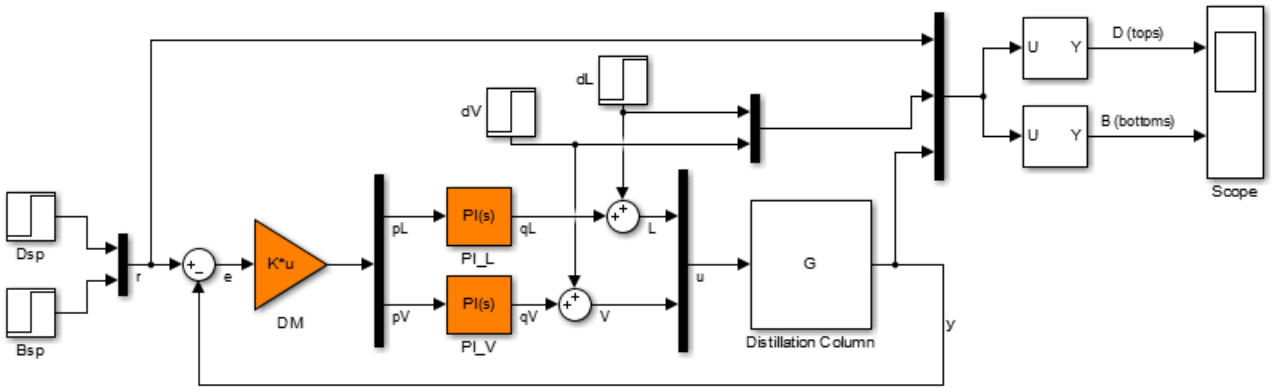
This example demonstrates how the **Quick Loop Tuning** option of **Control System Tuner** generates tuning goals from your crossover frequency and gain and phase margin specifications. This option lets you quickly set up SISO or MIMO feedback loops for tuning using a loop-shaping approach. The example 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” on page 15-15.)

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.

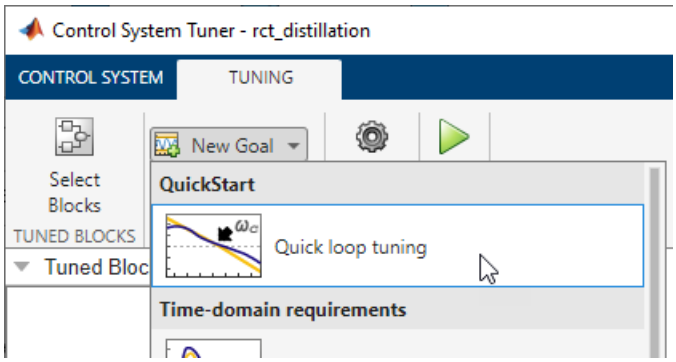
Open **Control System Tuner**. In the Simulink model window, in the **Apps** gallery, click **Control System Tuner**.

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 14-17.)

The model is now ready to tune to the target bandwidth and stability margins.

Specify the Goals for Quick Loop Tuning

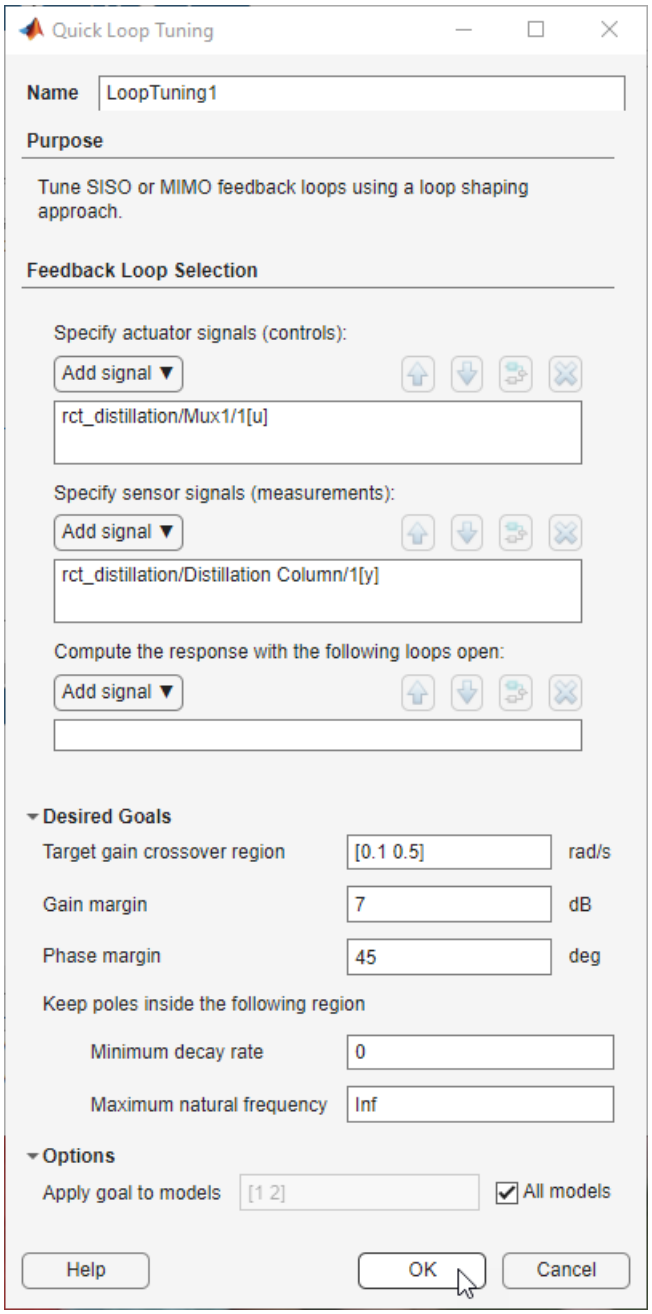
In the **Tuning** tab, select **New Goal** > **Quick Loop Tuning**.



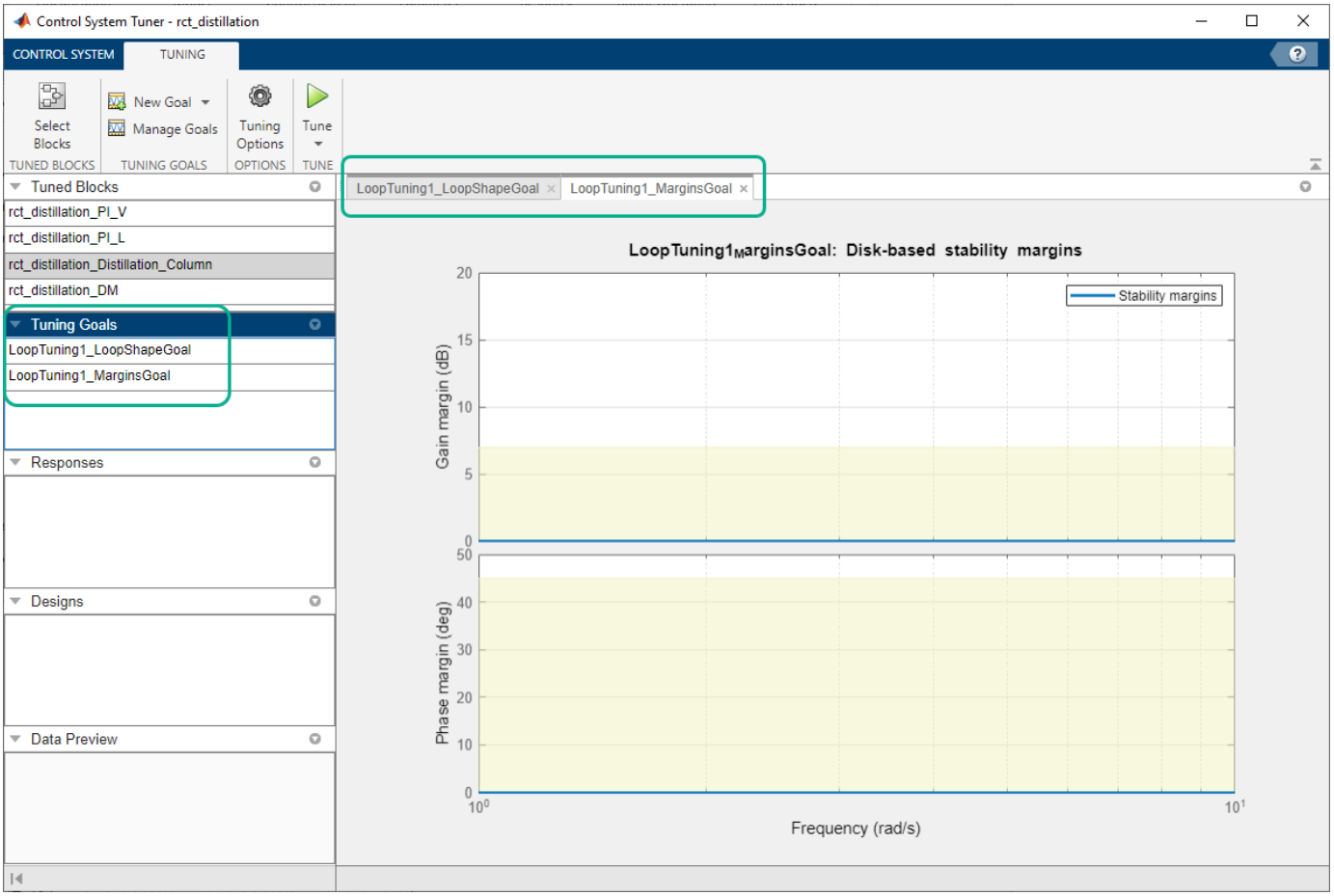
For Quick Loop Tuning, you need to identify the actuator signals and sensor signals that separate the plant portion of the control system from the controller, which for the purpose of Quick Loop Tuning is the rest of the control system. The actuator signals are the controller outputs that drive the plant, 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 14-28.)

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.



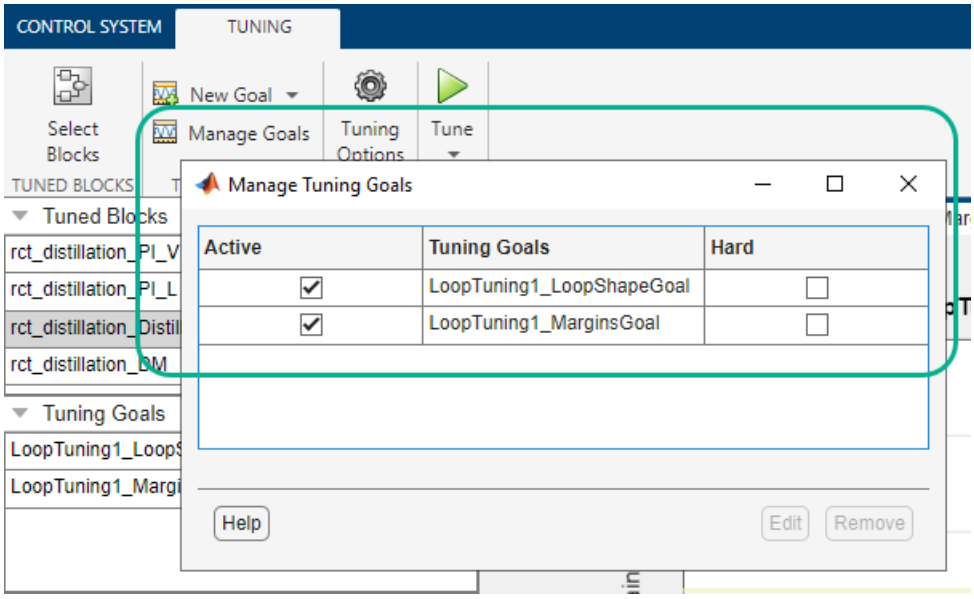
Click **OK**. **Control System Tuner** automatically generates tuning goals that capture the desired goals you entered in the dialog box.



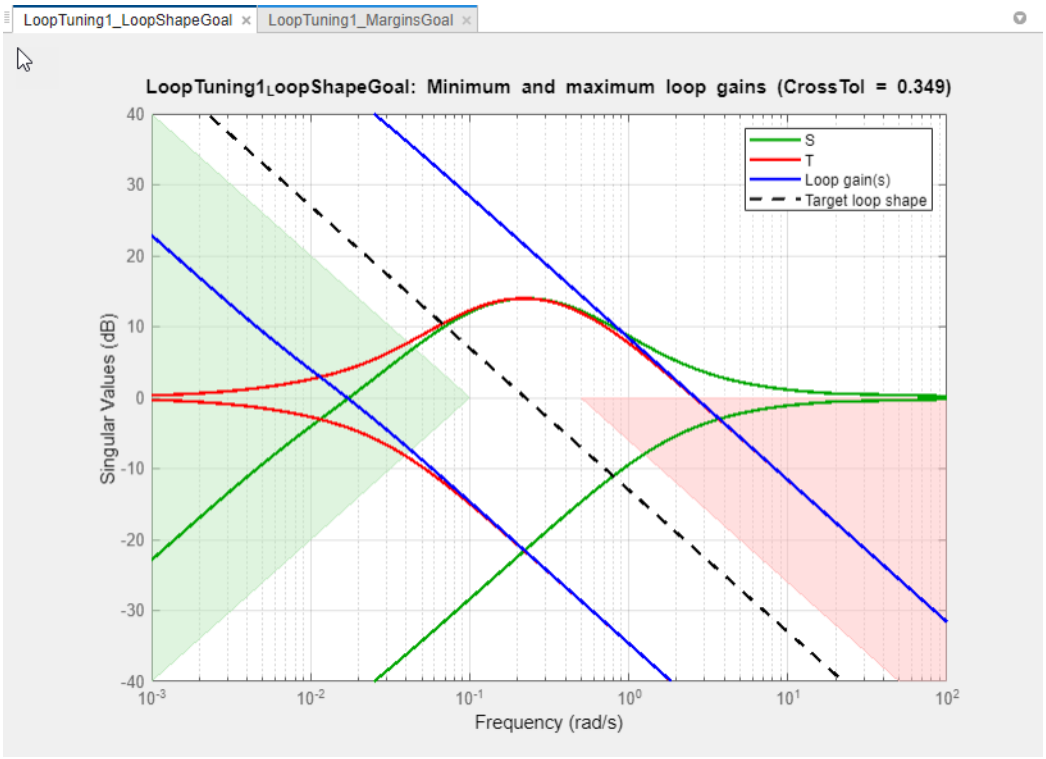
Examine the Automatically-Created Tuning Goals

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

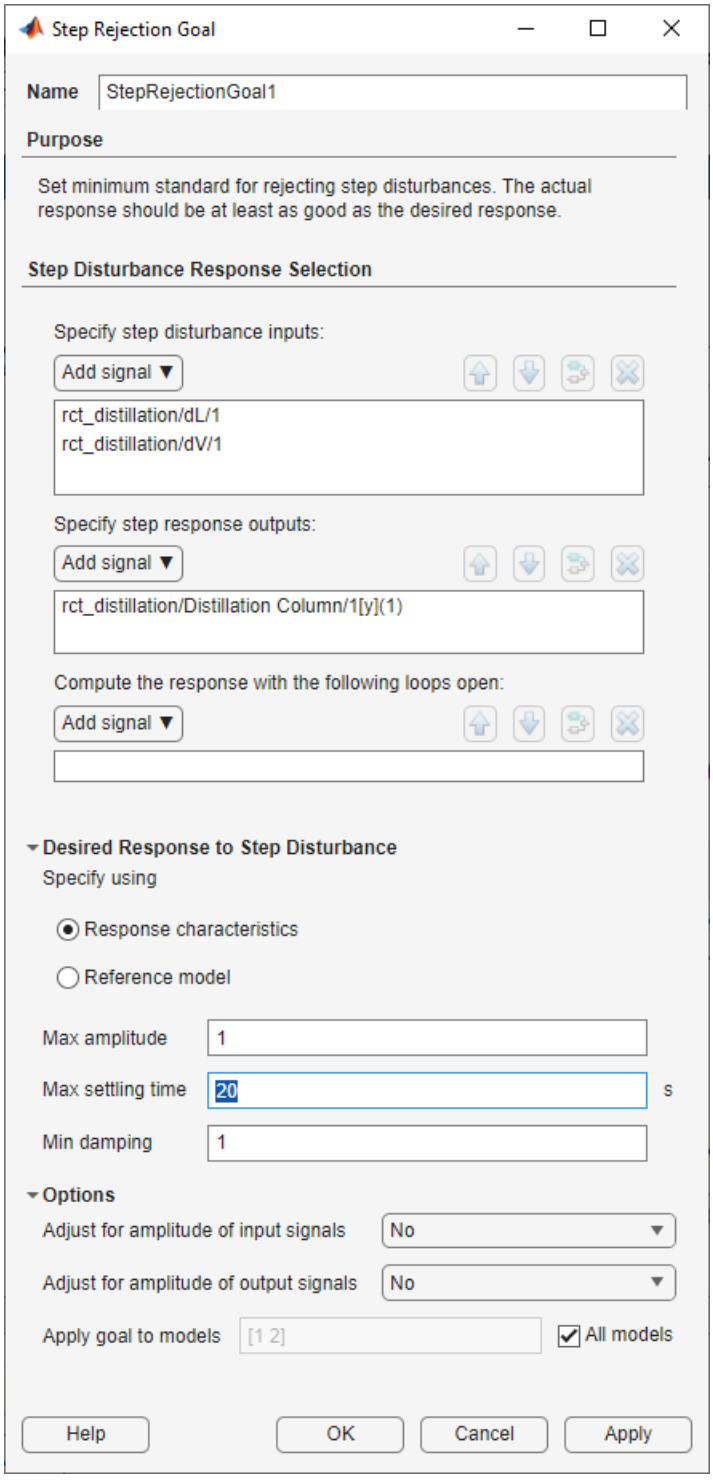
Click  **Tune** to tune the model to meet the automatically-created tuning goals. In the tuning goal plots, you can see that the requirements are satisfied.

To create additional plots for examining other system responses, see “Create Response Plots in Control System Tuner” on page 14-147.

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 14-28.)



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 14-11
- “Manage Tuning Goals” on page 14-134
- “Setup for Tuning Control System Modeled in MATLAB” on page 14-25
- “Stability Margins in Control System Tuning” on page 14-161

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 14-28.

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 14-33
- “Specify Goals for Interactive Tuning” on page 14-28
- “Visualize Tuning Goals” on page 14-141
- “Manage Tuning Goals” on page 14-134
- “Stability Margins in Control System Tuning” on page 14-161

Step Tracking Goal

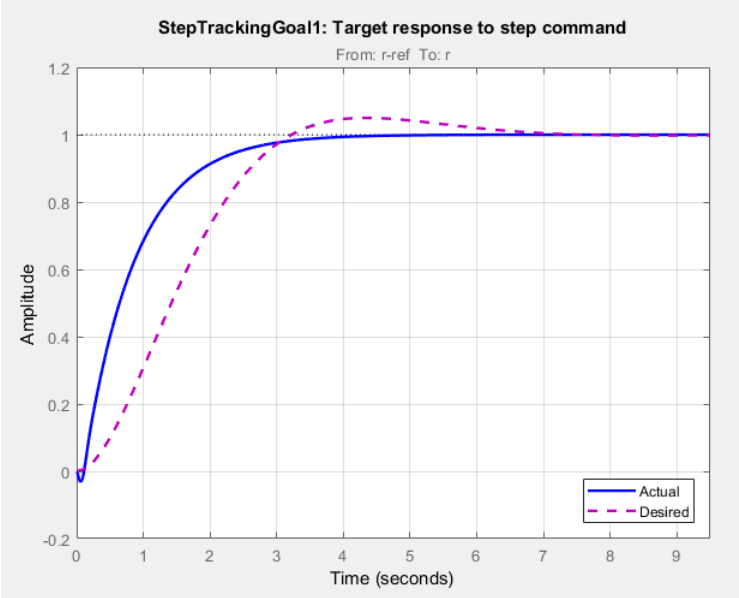
Purpose

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 14-28.

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 14-46). $\| \cdot \|_2$ 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 14-28
- “Manage Tuning Goals” on page 14-134
- “Visualize Tuning Goals” on page 14-141

Step Rejection Goal

Purpose

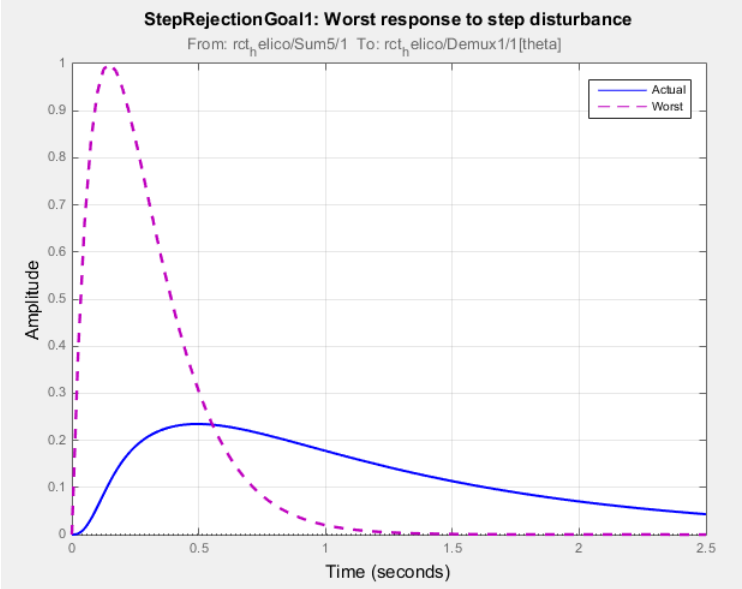
Set a minimum standard for rejecting step disturbances, when using **Control System Tuner**.

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 14-28.

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 14-141.

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 14-28
- “Manage Tuning Goals” on page 14-134
- “Visualize Tuning Goals” on page 14-141

Transient Goal

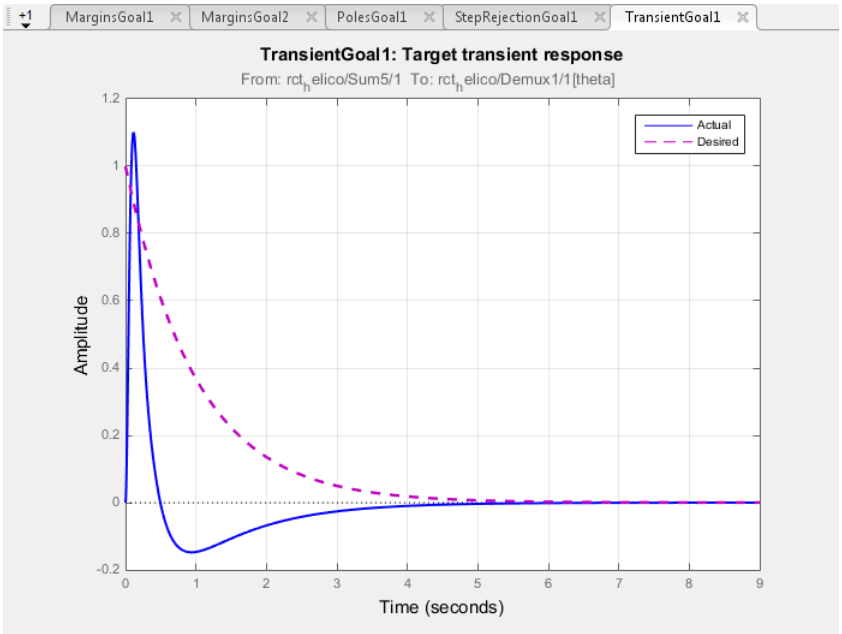
Purpose

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 14-28.

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 14-56), 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 14-19.
- 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 14-28
- “Manage Tuning Goals” on page 14-134
- “Visualize Tuning Goals” on page 14-141

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 14-28.

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 14-19.

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 14-28
- “Manage Tuning Goals” on page 14-134

Gain Goal

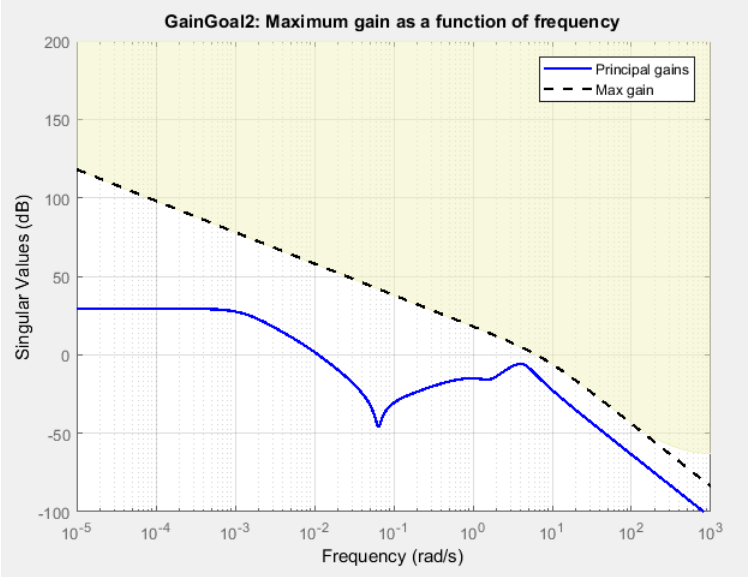
Purpose

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 14-28.

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 14-63. $\| \cdot \|_\infty$ 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 14-141.

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 14-28
- “Manage Tuning Goals” on page 14-134
- “Visualize Tuning Goals” on page 14-141

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 14-91 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 14-28.

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 14-68), 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 14-19.
- 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 14-28
- “Manage Tuning Goals” on page 14-134
- “Visualize Tuning Goals” on page 14-141

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.

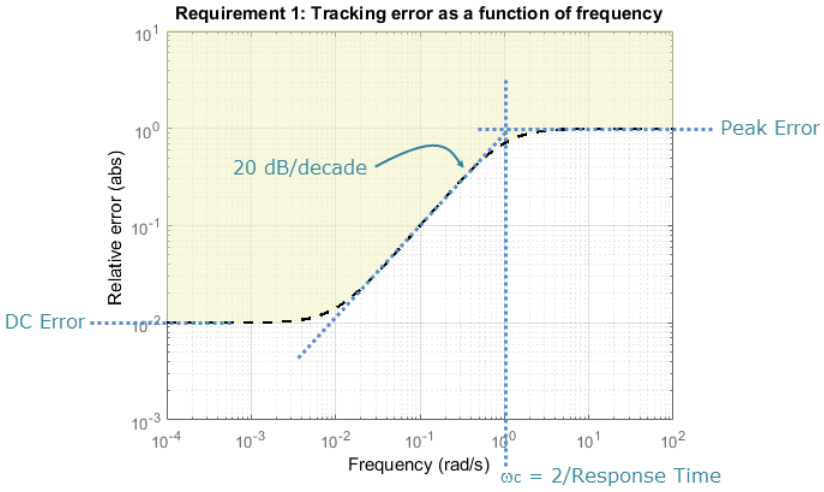
Description

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 14-28.

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 14-141.

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 14-28
- “Visualize Tuning Goals” on page 14-141
- “Manage Tuning Goals” on page 14-134

Overshoot Goal

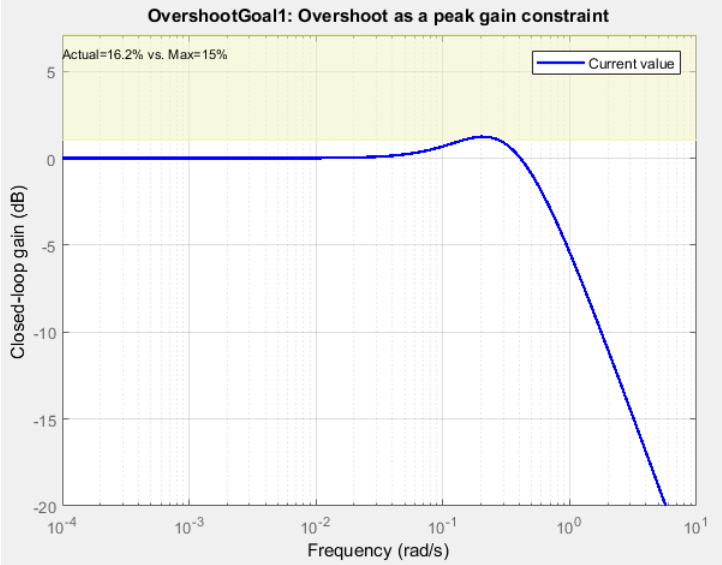
Purpose

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 14-28.

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 14-28

- “Manage Tuning Goals” on page 14-134
- “Visualize Tuning Goals” on page 14-141

Disturbance Rejection Goal

Purpose

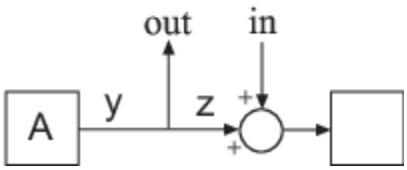
Attenuate disturbances at particular locations and in particular frequency bands, when using **Control System Tuner**.

Description

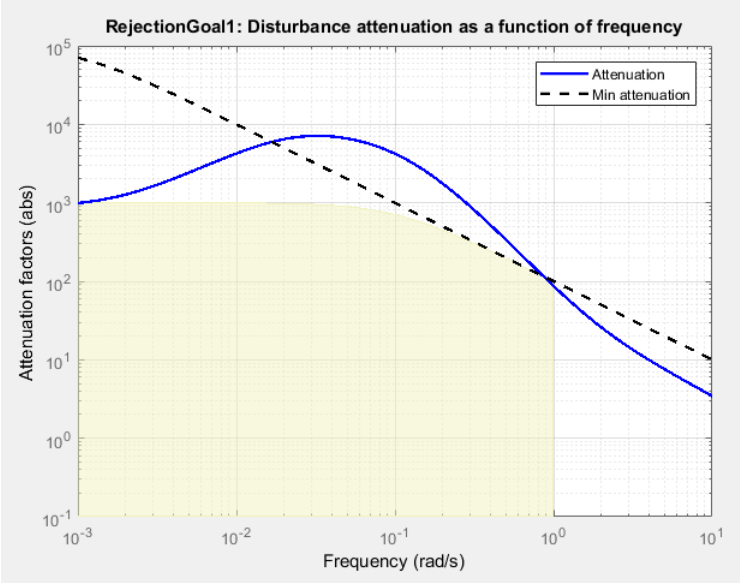
Disturbance Rejection Goal specifies the minimum attenuation of a disturbance injected at a specified location in a control system.

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 14-84.

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 14-28.

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 $s =$ `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 14-141.

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 14-28
- “Manage Tuning Goals” on page 14-134

- “Visualize Tuning Goals” on page 14-141

Sensitivity Goal

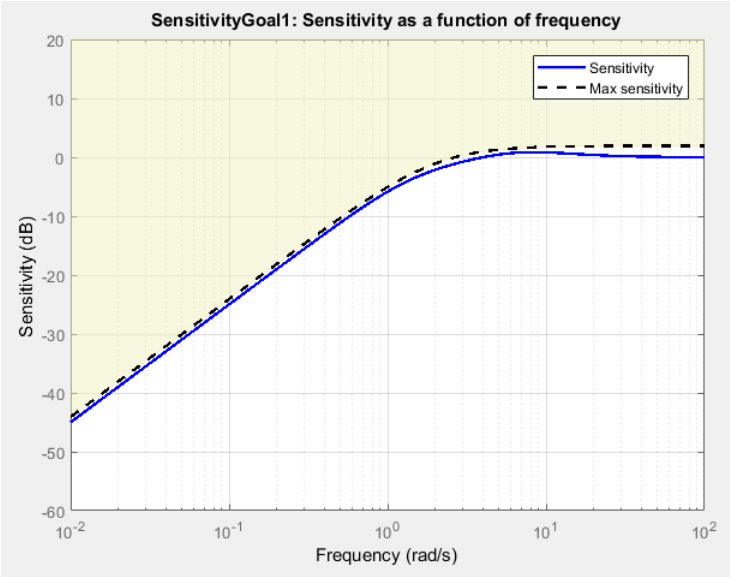
Purpose

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 14-79.

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 14-28.

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 14-141.

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 14-28
- “Manage Tuning Goals” on page 14-134
- “Visualize Tuning Goals” on page 14-141

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 14-28.

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 14-28
- “Visualize Tuning Goals” on page 14-141
- “Manage Tuning Goals” on page 14-134

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 14-28.

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 14-93), 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 14-19.
- 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 14-28
- “Visualize Tuning Goals” on page 14-141
- “Manage Tuning Goals” on page 14-134

Minimum Loop Gain Goal

Purpose

Boost gain of feedback loops at low frequency when using **Control System Tuner**.

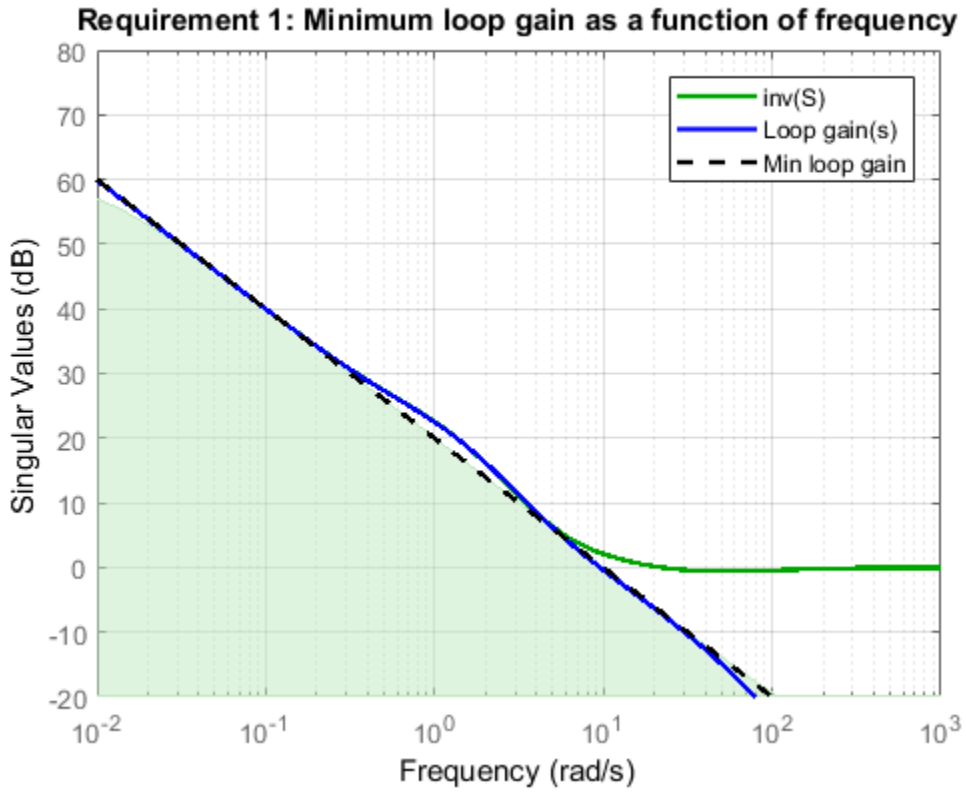
Description

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 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 14-105 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 14-28.

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 14-141.

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 14-28
- “Manage Tuning Goals” on page 14-134

- “Visualize Tuning Goals” on page 14-141

Maximum Loop Gain Goal

Purpose

Suppress gain of feedback loops at high frequency when using **Control System Tuner**.

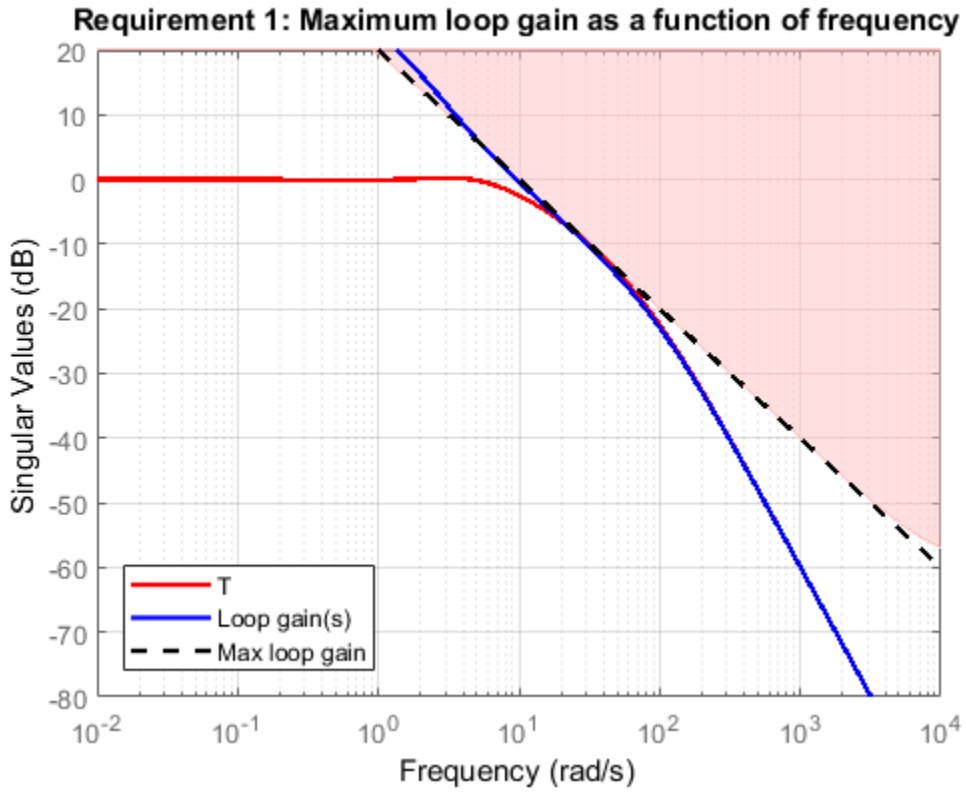
Description

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 14-105 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 14-28.

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 14-141.

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 14-28
- “Manage Tuning Goals” on page 14-134

- “Visualize Tuning Goals” on page 14-141

Loop Shape Goal

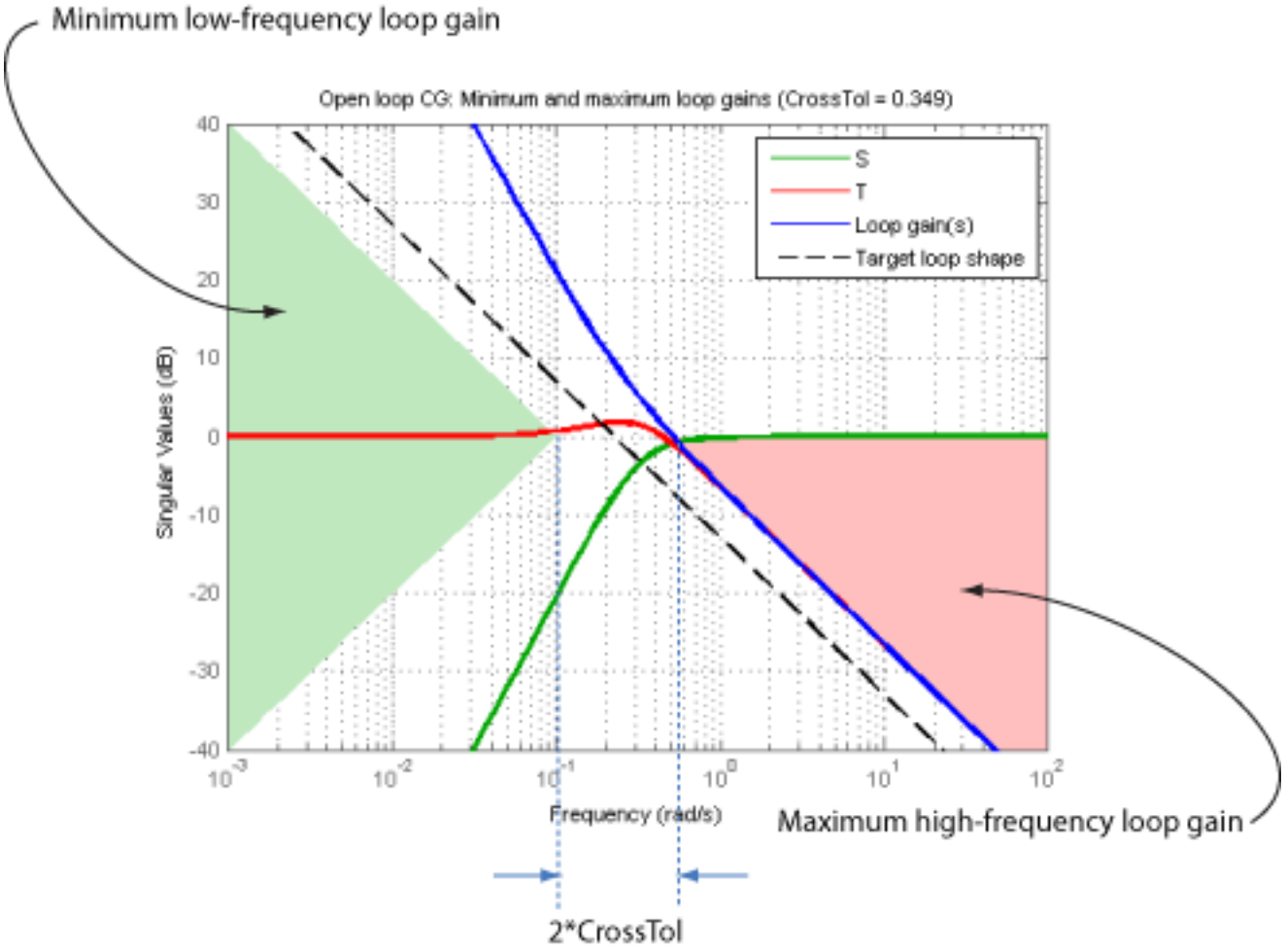
Purpose

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 14-95 or “Maximum Loop Gain Goal” on page 14-100. 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 14-28.

Desired Loop Shape

Use this section of the dialog box to specify the target loop shape.

- **Pure integrator wc/s**

  Check to specify a pure integrator and crossover frequency for the target loop shape. For example, to specify an integral gain profile with crossover frequency 10 rad/s, enter 10 in the **Crossover frequency wc** text box.

- **Other gain profile**

  Check to specify the target loop shape as a function of frequency. Enter a SISO numeric LTI model whose magnitude represents the desired gain profile. For example, you can specify a smooth transfer function (`tf`, `zpk`, or `ss` model). Alternatively, you can sketch a piecewise target loop shape using an `frd` model. When you do so, the software automatically maps the profile to a smooth transfer function that approximates the desired loop shape. For example, to specify a target loop shape of 100 (40 dB) below 0.1 rad/s, rolling off at a rate of –20 dB/decade at higher frequencies, enter `frd([100 100 10],[0 1e-1 1])`.

  If you are tuning in discrete time, you can specify the loop shape as a discrete-time model with the same sample time that you are using for tuning. If you specify the loop shape in continuous time, the tuning software discretizes it. Specifying the loop shape in discrete time gives you more control over the loop shape near the Nyquist frequency.

Options

Use this section of the dialog box to specify additional characteristics of the loop shape goal.

- **Enforce loop shape within**

  Specify the tolerance in the location of the crossover frequency, in decades. For example, to allow gain crossovers within half a decade on either side of the target crossover frequency, enter 0.5. Increase the crossover tolerance to increase the ability of the tuning algorithm to enforce the target loop shape for all loops in a MIMO control system.

- **Enforce goal in frequency range**

  Limit the enforcement of the tuning goal to a particular frequency band. Specify the frequency band as a row vector of the form `[min,max]`, expressed in frequency units of your model. For example, to create a tuning goal that applies only between 1 and 100 rad/s, enter `[1,100]`. By default, the tuning goal applies at all frequencies for continuous time, and up to the Nyquist frequency for discrete time.

- **Stabilize closed loop system**

  By default, the tuning goal imposes a stability requirement on the closed-loop transfer function from the specified inputs to outputs, in addition to the gain constraint. If stability is not required or cannot be achieved, select `No` to remove the stability requirement. For example, if the gain constraint applies to an unstable open-loop transfer function, select `No`.

- **Equalize loop interactions**

  For multi-loop or MIMO loop gain constraints, the feedback channels are automatically rescaled to equalize the off-diagonal (loop interaction) terms in the open-loop transfer function. Select `Off` to disable such scaling and shape the unscaled open-loop response.

- **Apply goal to**

  Use this option when tuning multiple models at once, such as an array of models obtained by linearizing a Simulink model at different operating points or block-parameter values. By default, active tuning goals are enforced for all models. To enforce a tuning requirement for a subset of models in an array, select **Only Models**. Then, enter the array indices of the models for which the goal is enforced. For example, suppose you want to apply the tuning goal to the second, third, and fourth models in a model array. To restrict enforcement of the requirement, enter `2:4` in the **Only Models** text box.

  For more information about tuning for multiple models, see “Robust Tuning Approaches” (Robust Control Toolbox).

Algorithms

Evaluating Tuning Goals

When you tune a control system, the software converts each tuning goal into a normalized scalar value $f(x)$. Here, x is the vector of free (tunable) parameters in the control system. The software then adjusts the parameter values to minimize $f(x)$ or to drive $f(x)$ below 1 if the tuning goal is a hard constraint.

For **Loop Shape Goal**, $f(x)$ is given by:

$$f(x) = \left\| \begin{matrix} W_S S \\ W_T T \end{matrix} \right\|_\infty .$$

$S = D^{-1}[I - L(s,x)]^{-1}D$ is the scaled sensitivity function.

$L(s,x)$ is the open-loop response being shaped.

D is an automatically-computed loop scaling factor. (If **Equalize loop interactions** is set to `Off`, then $D = I$.)

$T = S - I$ is the complementary sensitivity function.

W_S and W_T are frequency weighting functions derived from the specified loop shape. The gains of these functions roughly match your specified loop shape and its inverse, respectively, for values ranging from –20 dB to 60 dB. For numerical reasons, the weighting functions level off outside this range, unless the specified gain profile changes slope outside this range. Because poles of W_S or W_T close to $s = 0$ or $s =$ `Inf` might lead to poor numeric conditioning for tuning, it is not recommended to specify loop shapes with very low-frequency or very high-frequency dynamics. For more information about regularization and its effects, see “Visualize Tuning Goals” on page 14-141.

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 14-28
- “Manage Tuning Goals” on page 14-134
- “Visualize Tuning Goals” on page 14-141

Margins Goal

Purpose

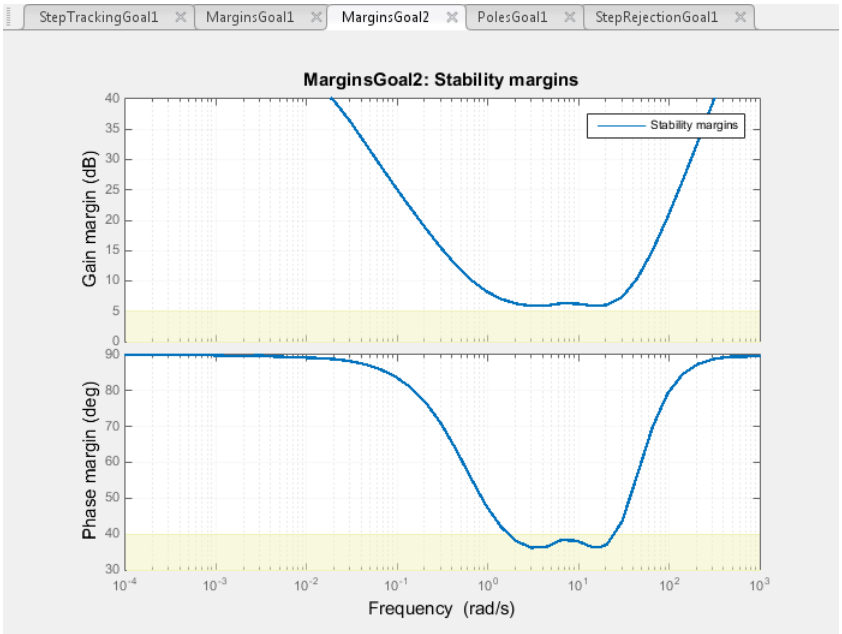
Enforce specified gain and phase margins when using **Control System Tuner**.

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 14-161.

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 14-28.

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 14-161.

- **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 14-161.

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 14-28
- “Manage Tuning Goals” on page 14-134
- “Visualize Tuning Goals” on page 14-141
- “Stability Margins in Control System Tuning” on page 14-161

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:

$$\int_0^T y(t)^T u(t) dt > 0,$$

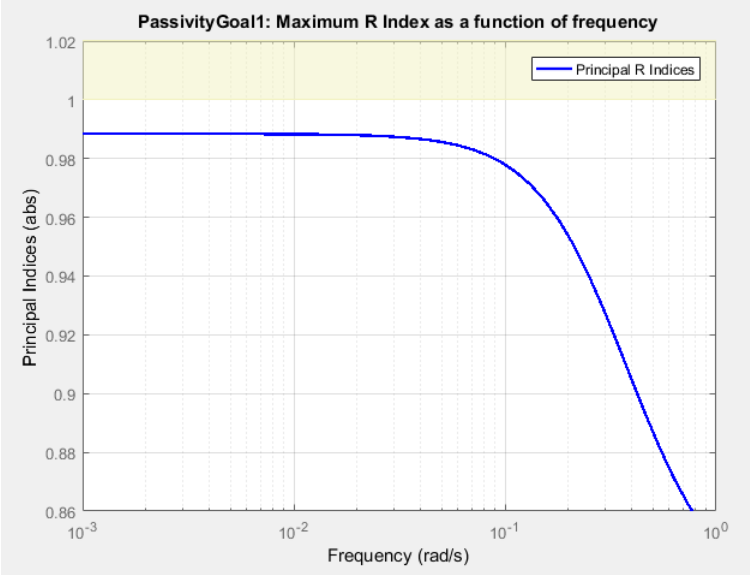
for all $T > 0$. Equivalently, a system is passive if its frequency response is positive real, which means that for all $\omega > 0$,

$$G(j\omega) + G(j\omega)^H > 0$$

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” on page 10-2 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 14-116.

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 14-28.

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 14-28
- “Manage Tuning Goals” on page 14-134
- “Visualize Tuning Goals” on page 14-141
- “Passive Control of Water Tank Level” on page 18-196
- “About Passivity and Passivity Indices” on page 10-2

Conic Sector Goal

Purpose

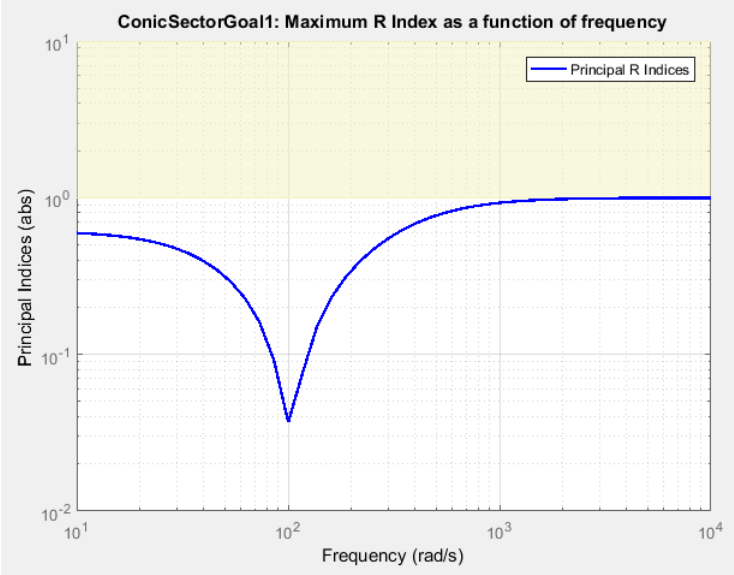
Enforce sector bound on specific input/output map when using **Control System Tuner**.

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” on page 10-7.

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 14-28.

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” on page 10-7.

- **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**

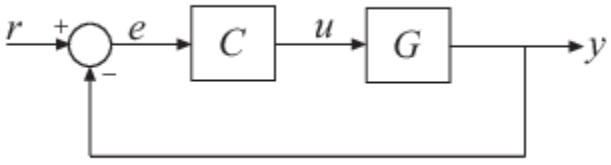
  Use this option when tuning multiple models at once, such as an array of models obtained by linearizing a Simulink model at different operating points or block-parameter values. By default, active tuning goals are enforced for all models. To enforce a tuning requirement for a subset of models in an array, 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” on page 10-7 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 2-68 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” on page 10-7). 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” on page 10-7
- “Specify Goals for Interactive Tuning” on page 14-28
- “Manage Tuning Goals” on page 14-134
- “Visualize Tuning Goals” on page 14-141

Weighted Passivity Goal

Purpose

Enforce passivity of a frequency-weighted transfer function when tuning in **Control System Tuner**.

Description

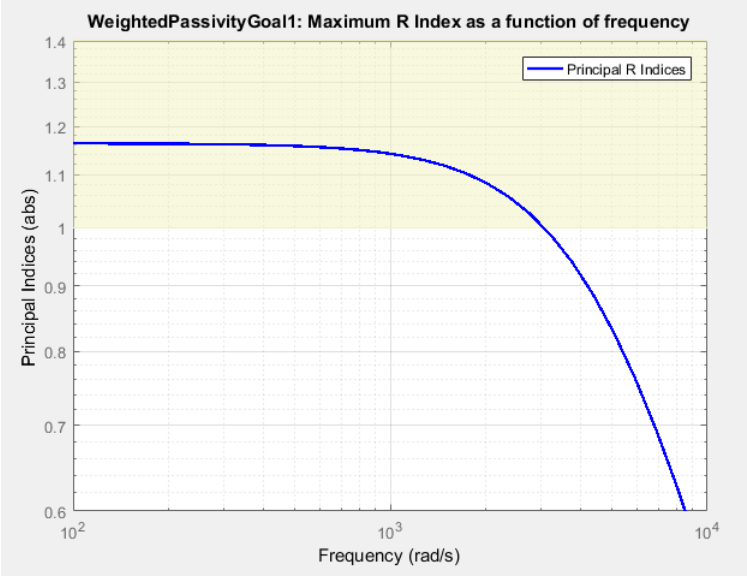
Weighted Passivity Goal enforces the passivity of $H(s) = W_L(s)T(s)W_R(s)$, where $T(s)$ is the transfer function from specified inputs to outputs. $W_L(s)$ and $W_R(s)$ are frequency weights used to emphasize particular frequency bands. A system is passive if all its I/O trajectories $(u(t),y(t))$ 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 14-126.

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 14-28.

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 14-28
- “Manage Tuning Goals” on page 14-134
- “Visualize Tuning Goals” on page 14-141
- “About Passivity and Passivity Indices” on page 10-2

Poles Goal

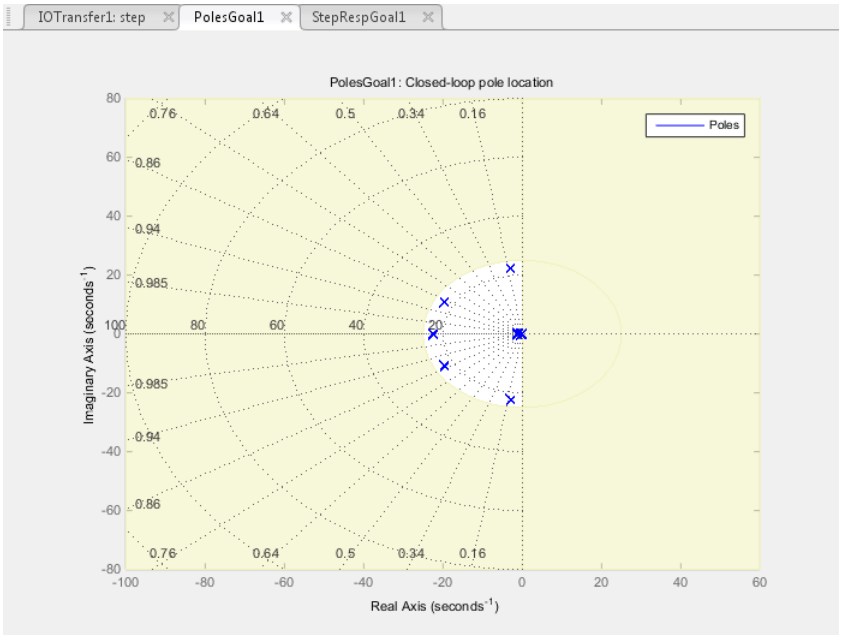
Purpose

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 14-131.

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 14-28.

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 14-28
- “Manage Tuning Goals” on page 14-134
- “Visualize Tuning Goals” on page 14-141

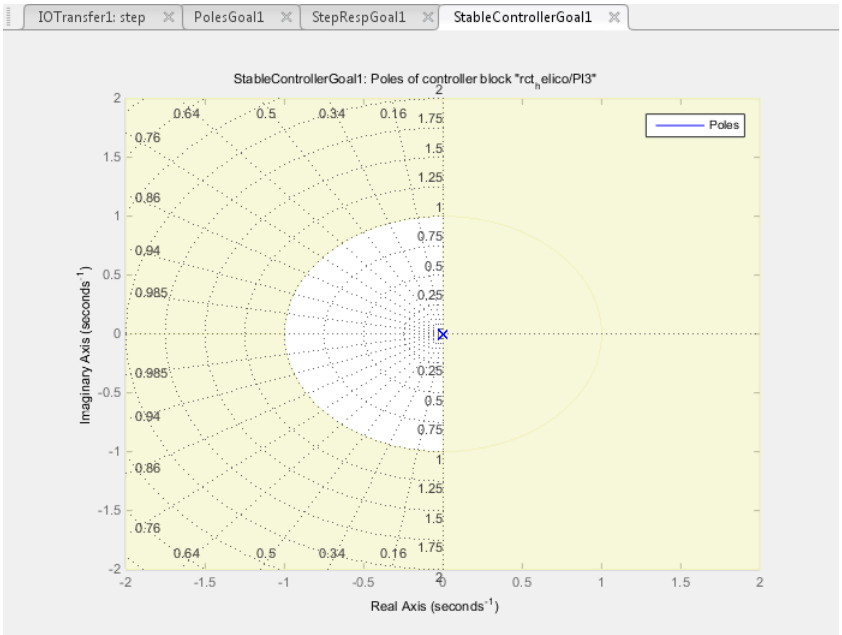
Controller Poles Goal

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 14-127.

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 14-17.

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 14-28
- “Manage Tuning Goals” on page 14-134
- “Visualize Tuning Goals” on page 14-141

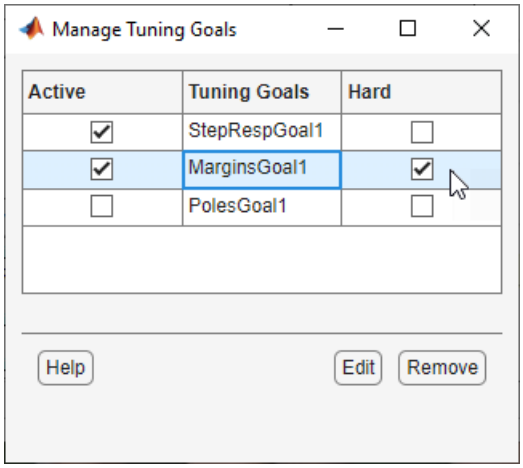
Manage Tuning Goals

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 14-28.

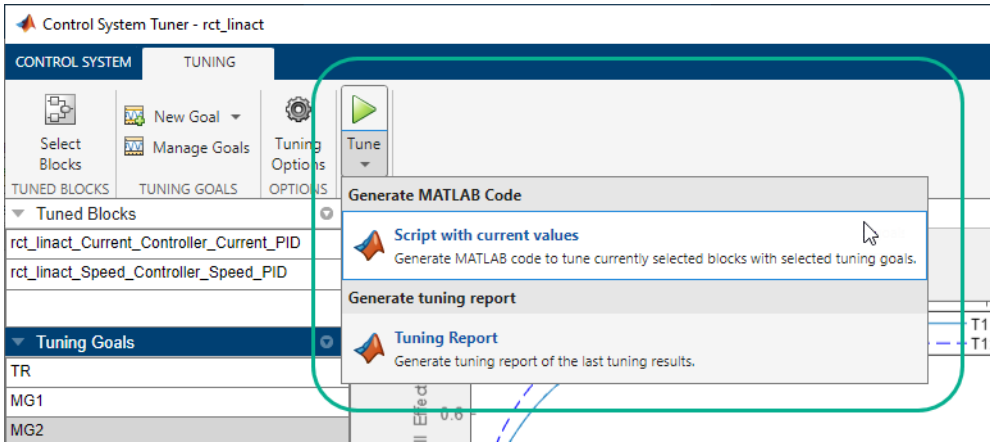
Generate MATLAB Code from Control System Tuner for Command-Line Tuning

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" on page 18-85. 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 14-157.)

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 14-134.)

```
%% 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 14-157
- “Tune Control System at the Command Line” on page 14-166
- “Validate Tuned Control System” on page 14-168

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 14-141.)

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” on page 18-11, `systune` is called with one soft requirement, `R1`, and two hard requirements `R2` and `R3`.

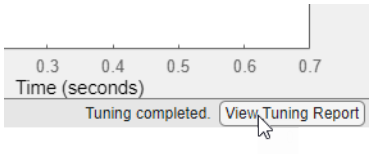
```
T1 = systune(T0,R1,[R2 R3]);

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 14-141.)

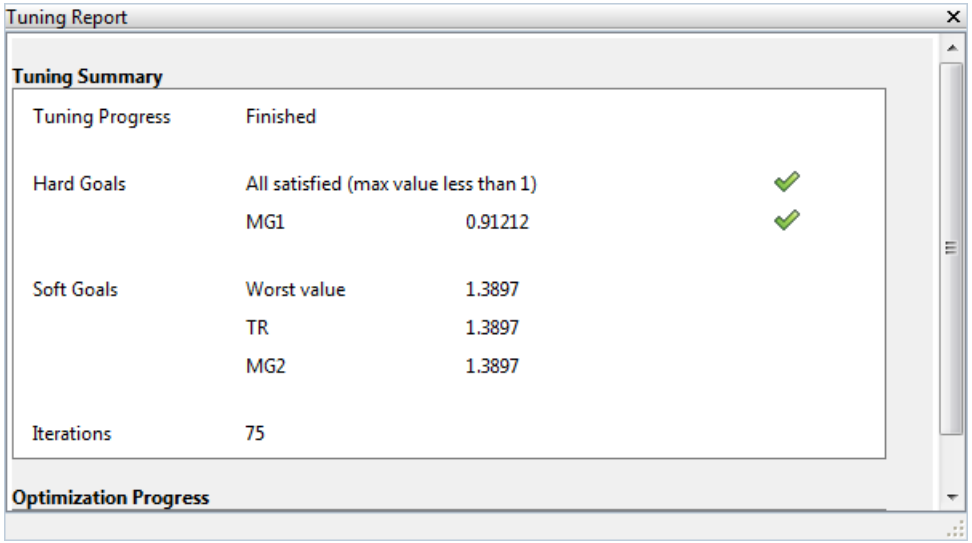
You can obtain additional information about the optimization progress and values using the `info` output of `systune`. To make `systune` display additional information during tuning, use `systuneOptions`.

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**.

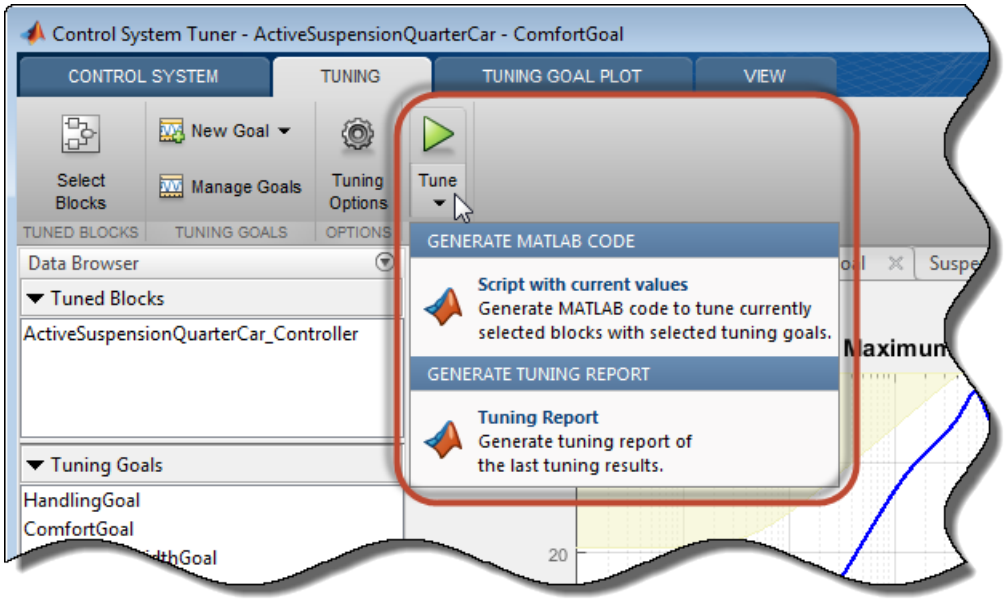


The tuning report displays the final $f_i(x)$ and $g_j(x)$ values obtained by the algorithm.



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 14-141.)

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 14-168.

See Also

`systune` | `systune (for slTuner)` | `viewGoal` | `evalGoal`

Related Examples

- “Visualize Tuning Goals” on page 14-141
- “Validate Tuned Control System” on page 14-168

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

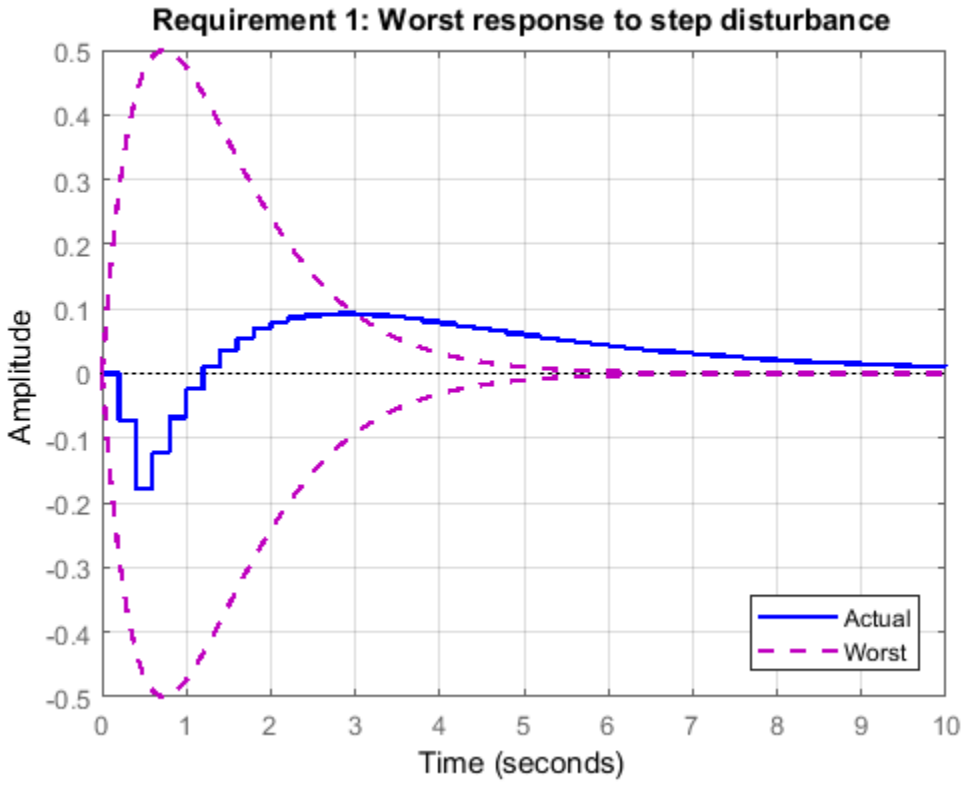
How you obtain tuning-goal plots depends on your work environment.

- 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.

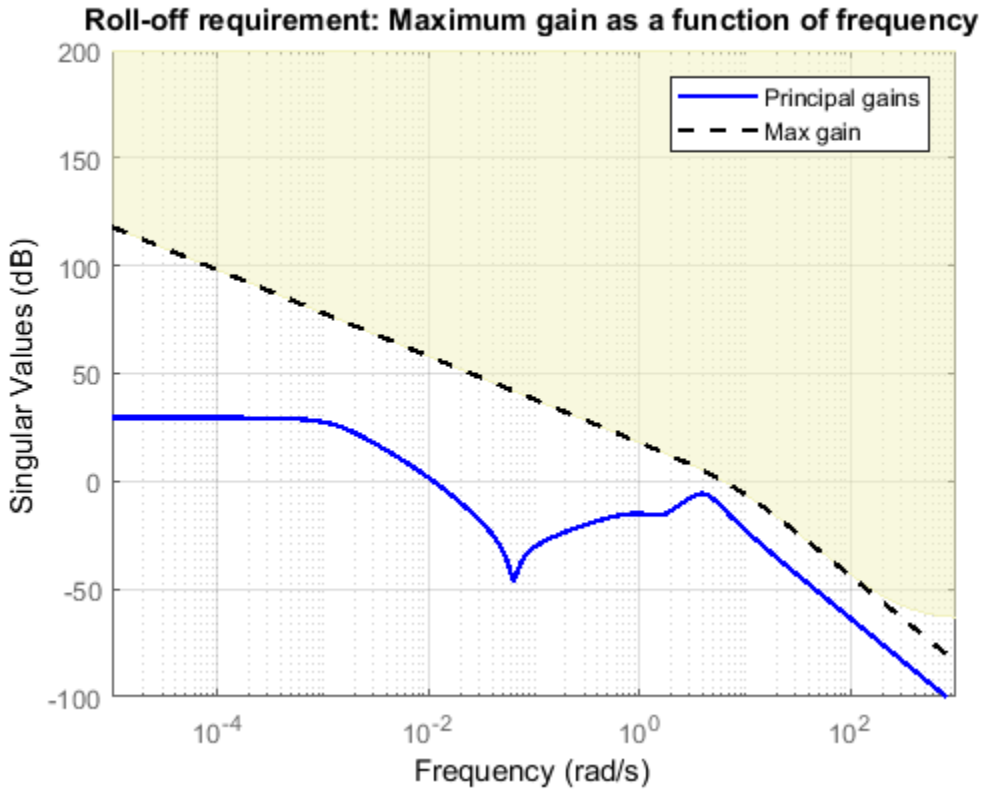
Time-Domain Goals

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” on page 18-141, 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” on page 18-176, 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 14-161.

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.

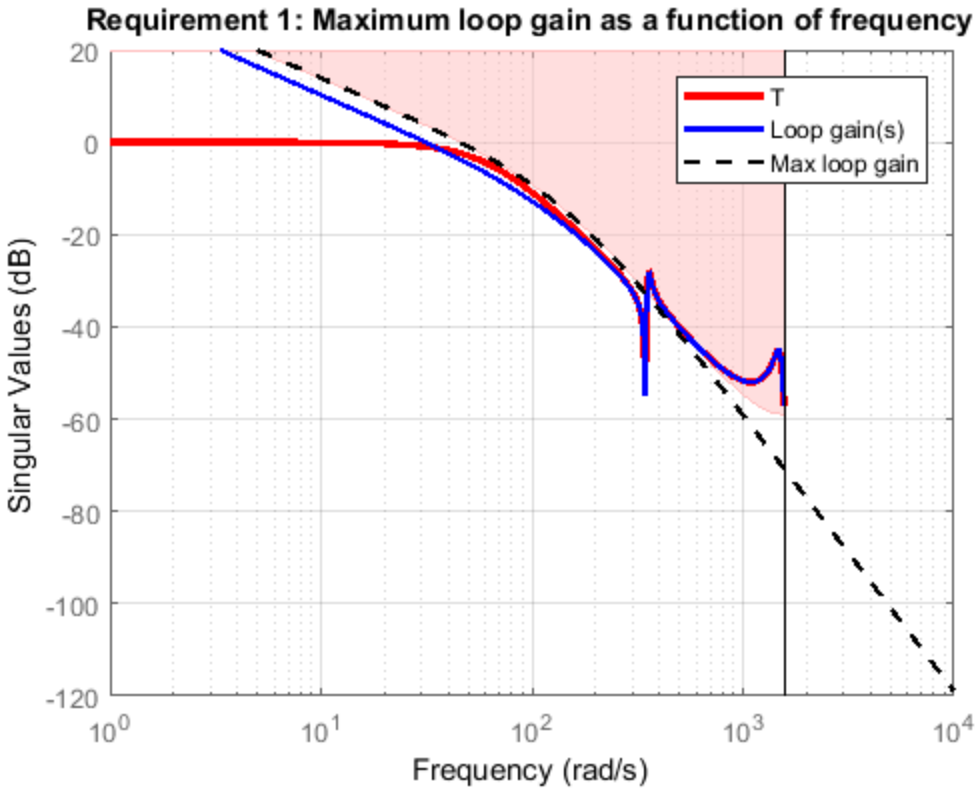
Continuous-Time Gain Profile for Discrete-Time Tuning

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.

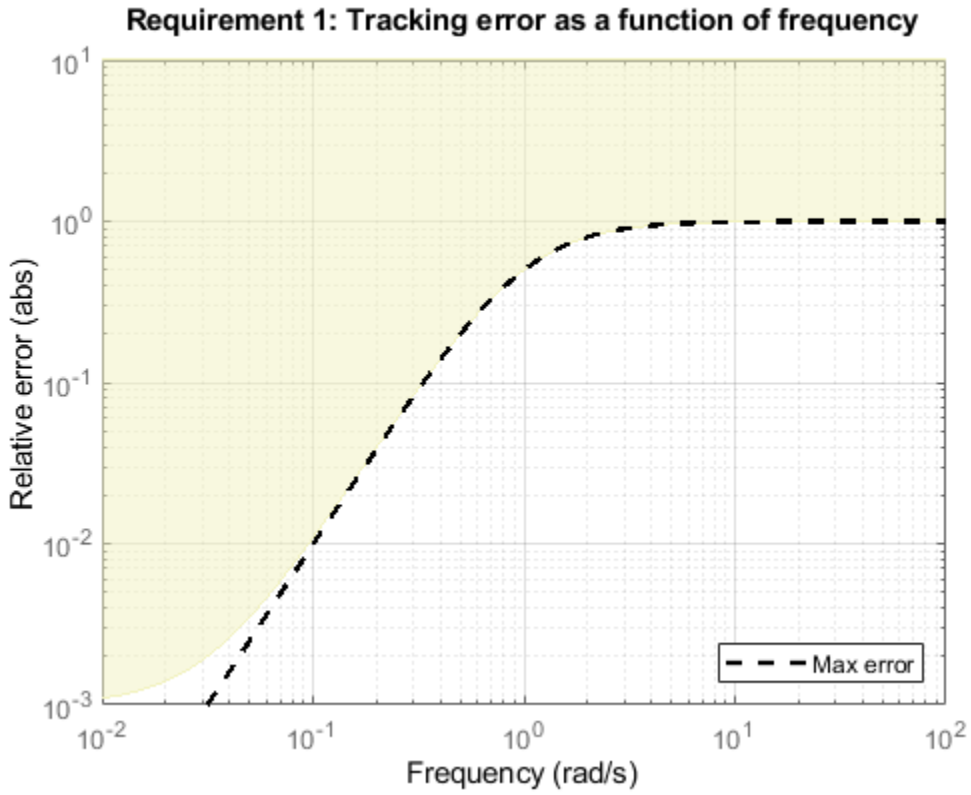
The plot highlights that sometimes it is preferable to specify tuning goals for discrete-time tuning using discrete-time gain profiles. In particular, specifying a discrete-time profile gives you more control over the behavior of the gain profile near the Nyquist frequency.

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.

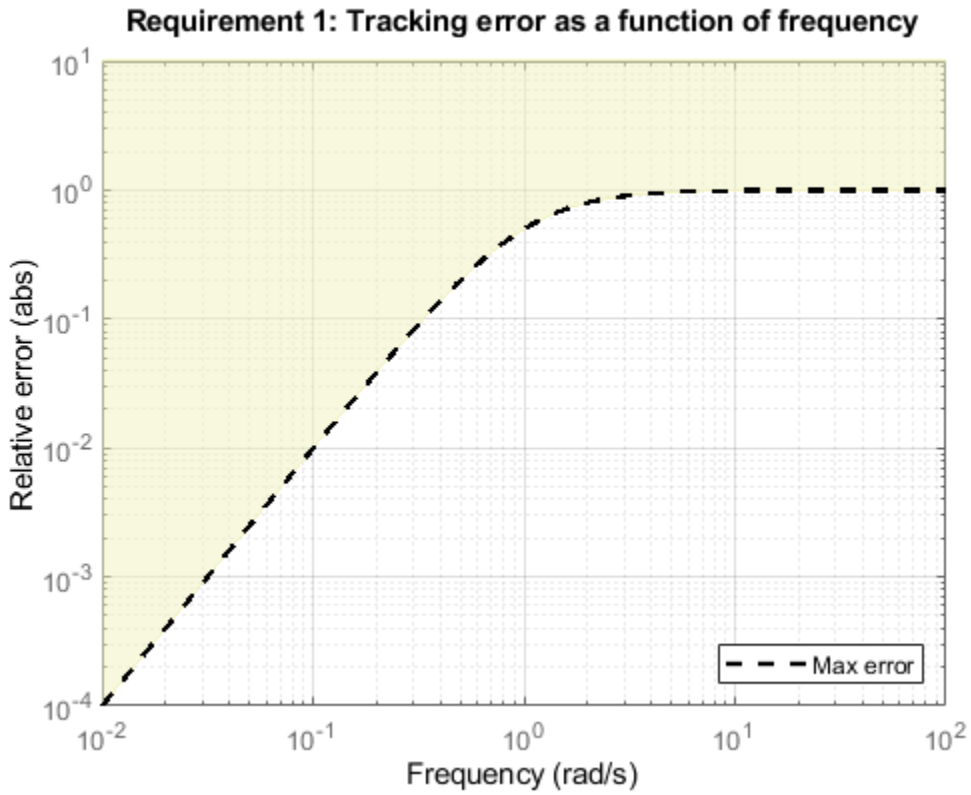
For example, consider the following tracking goal.

```
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 14-168.

See Also

`viewGoal`

Related Examples

- “Interpret Numeric Tuning Results” on page 14-138
- “Create Response Plots in Control System Tuner” on page 14-147
- “Validate Tuned Control System” on page 14-168

Create Response Plots in Control System Tuner

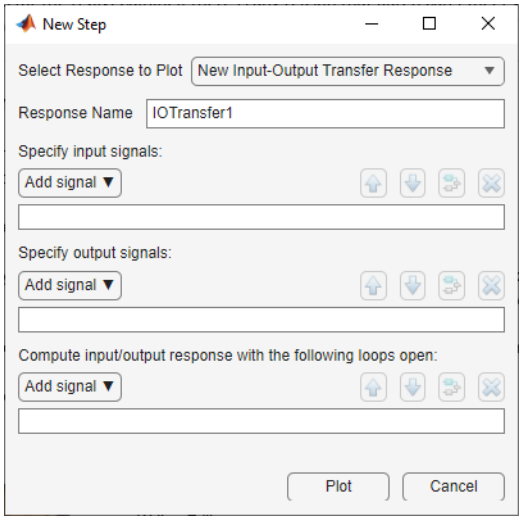
This example shows how to create response plots for analyzing system performance in **Control System Tuner**. **Control System Tuner** can generate many types of response plots in the time and frequency domains. You can view responses of SISO or MIMO transfer functions between inputs and outputs at any 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

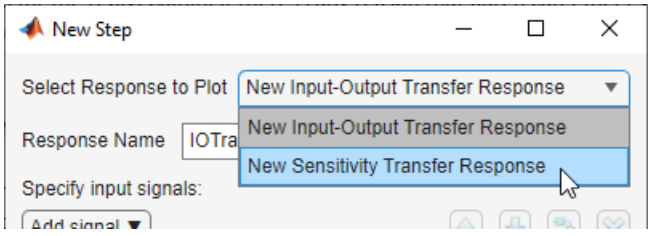
In **Control System Tuner**, in the **Control System** tab, click **New Plot**. Select the type of plot you want to create.

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

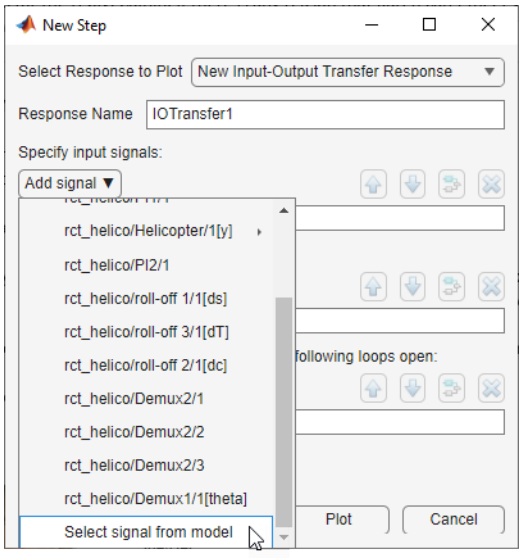
Type a name for the response in the **Response Name** text box. Once you have specified signal locations defining the response, **Control System Tuner** stores the response under this name. When you create additional new response plots, the response appears by this name in **Select Response to Plot** 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.)

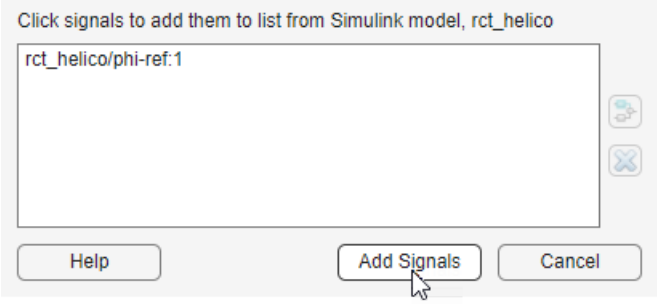
Under **Specify input signals**, click ➕ **Add signal to list**. A list of available input locations appears.

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**.



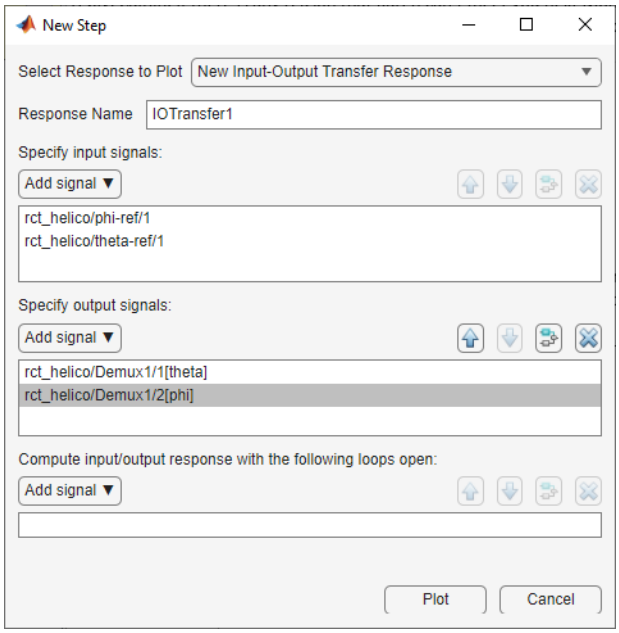
In the **Select signals** dialog box, build a list of the signals you want. To do so, click signals in the Simulink model editor. The signals that you click appear in the **Select signals** dialog box. Click one signal to create a SISO response, and click multiple signals to create a MIMO response.

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`.



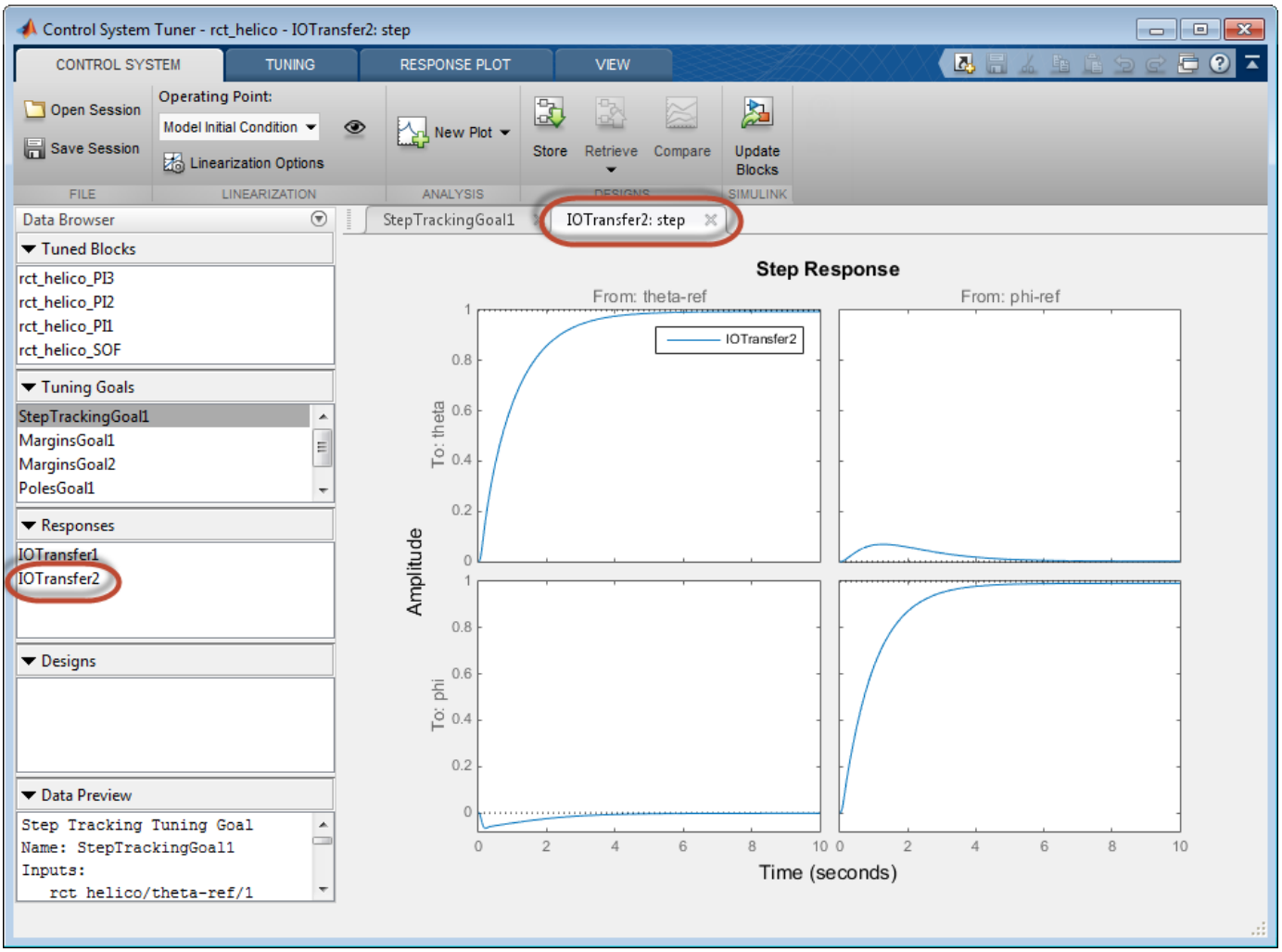
Tip To highlight any selected signal in the Simulink model, click . To remove a signal from the input or output list, click . When you have selected multiple signals, you can reorder them using and .

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.

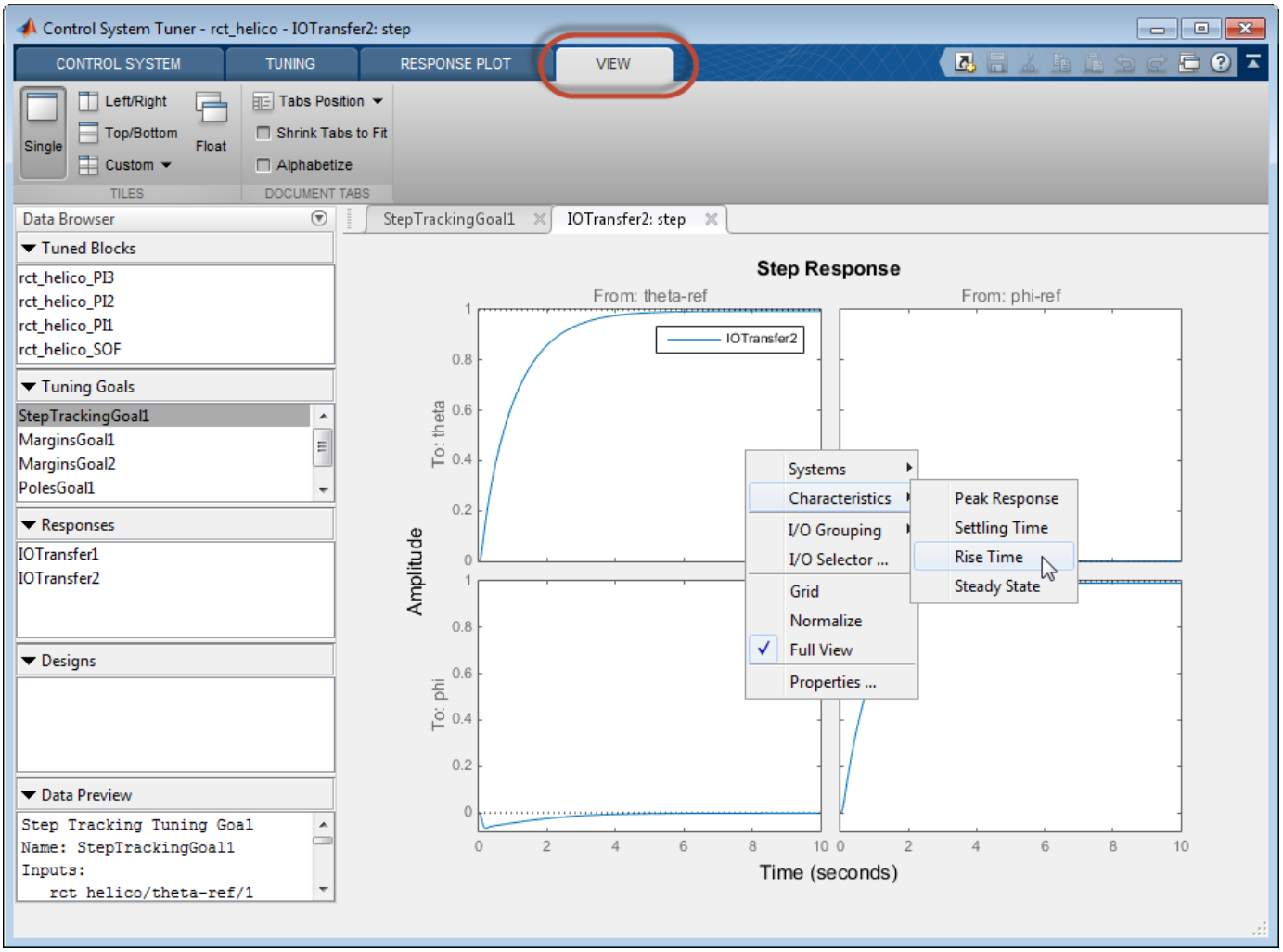
Store and Plot the Response

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

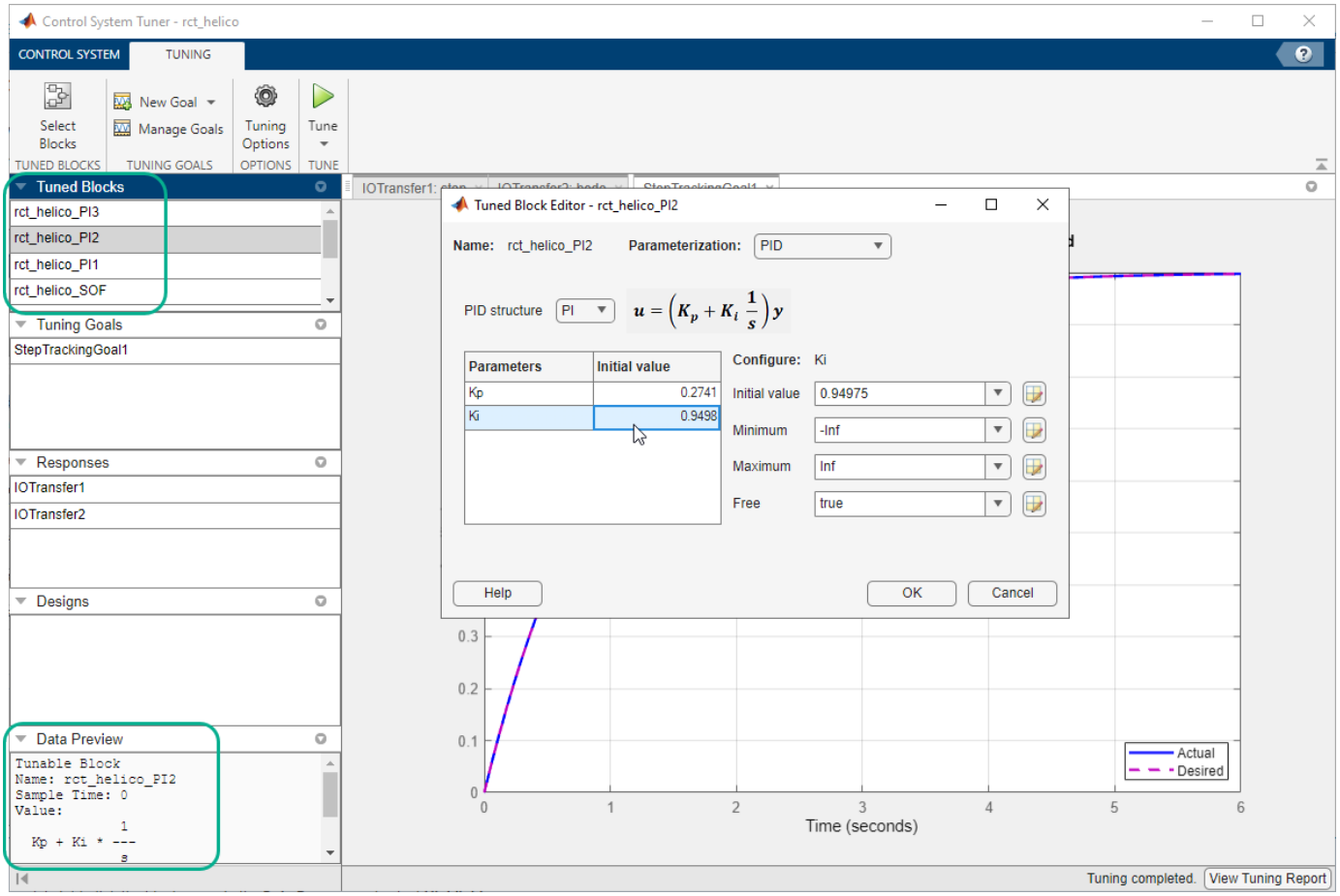
Related Examples

- “Compare Performance of Multiple Tuned Controllers” on page 14-154
- “Examine Tuned Controller Parameters in Control System Tuner” on page 14-152
- “Visualize Tuning Goals” on page 14-141

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 14-19

Compare Performance of Multiple Tuned Controllers

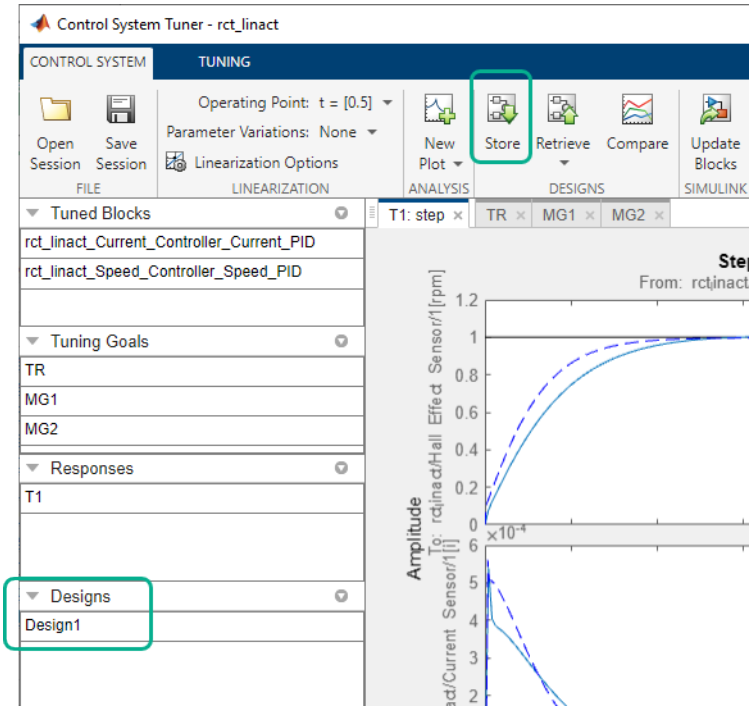
Control System Tuner lets you compare the performance of a control system tuned with two different sets of tuning goals. Such comparison is useful, for example, to see the effect on performance of changing a tuning goal from hard goal to soft goal. Comparing performance is also useful to see the effect 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.

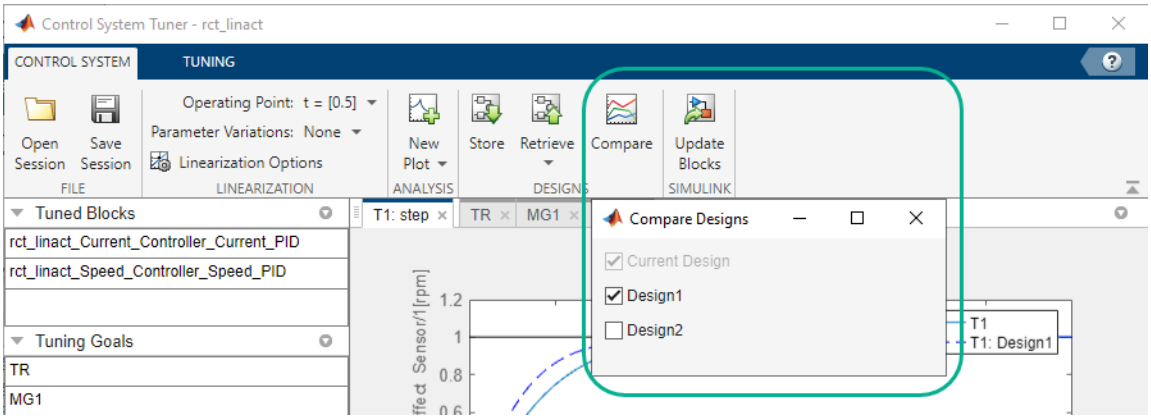
Compute New Design

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.

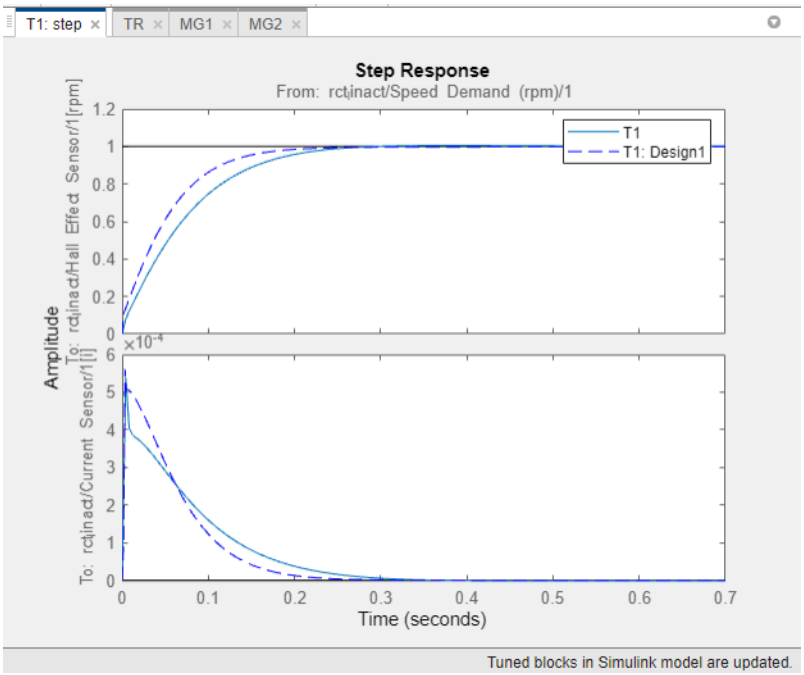
When you are ready, retune the control system with the new set of tuning goals. Click **Tune**. **Control System Tuner** updates the current design (the current set of controller parameters) with the new tuned design. All existing plots, which by default show the current design, are updated to reflect 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.

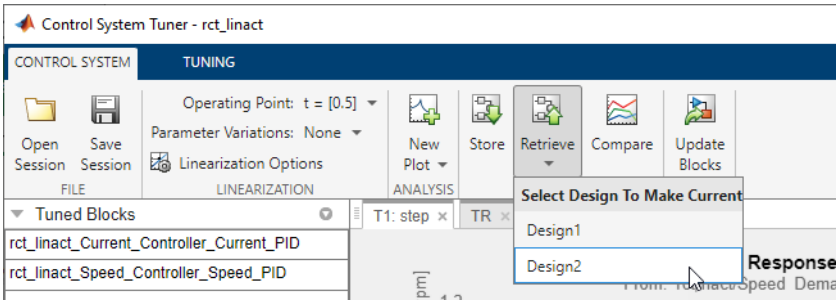


Use the same procedure save and compare as many designs as you need.

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

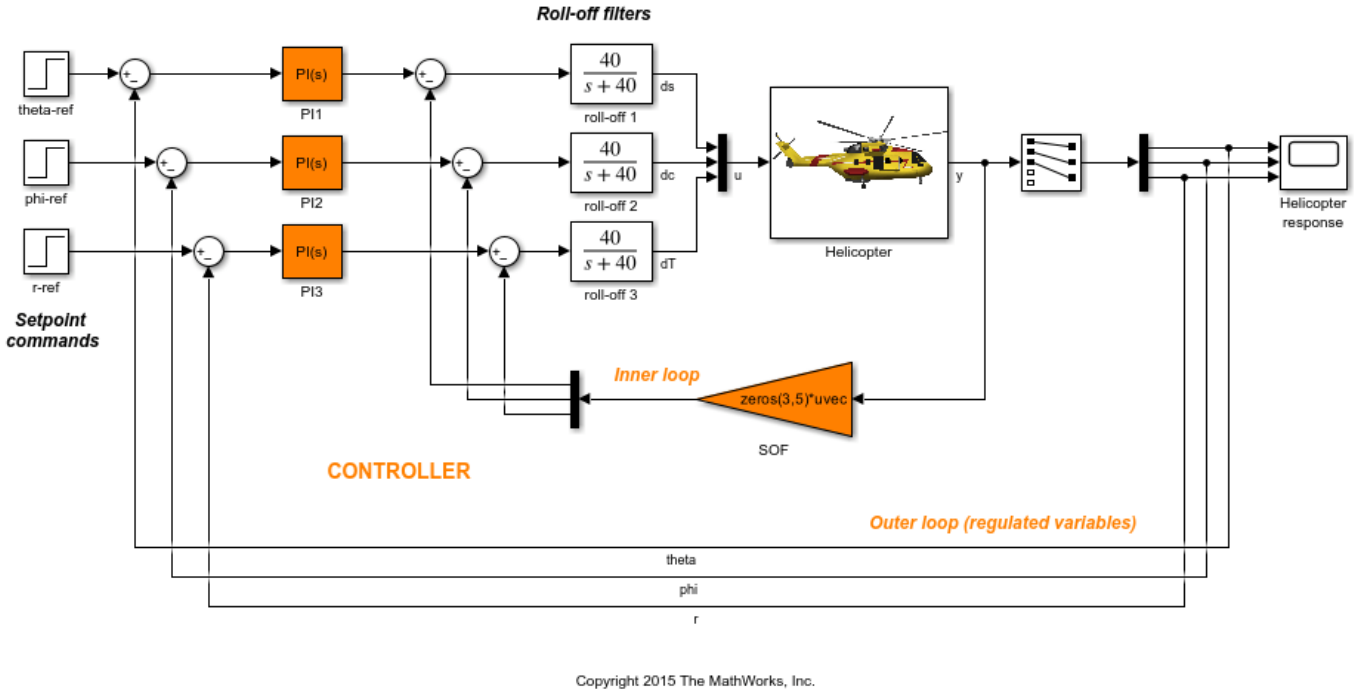
- “Create Response Plots in Control System Tuner” on page 14-147

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.

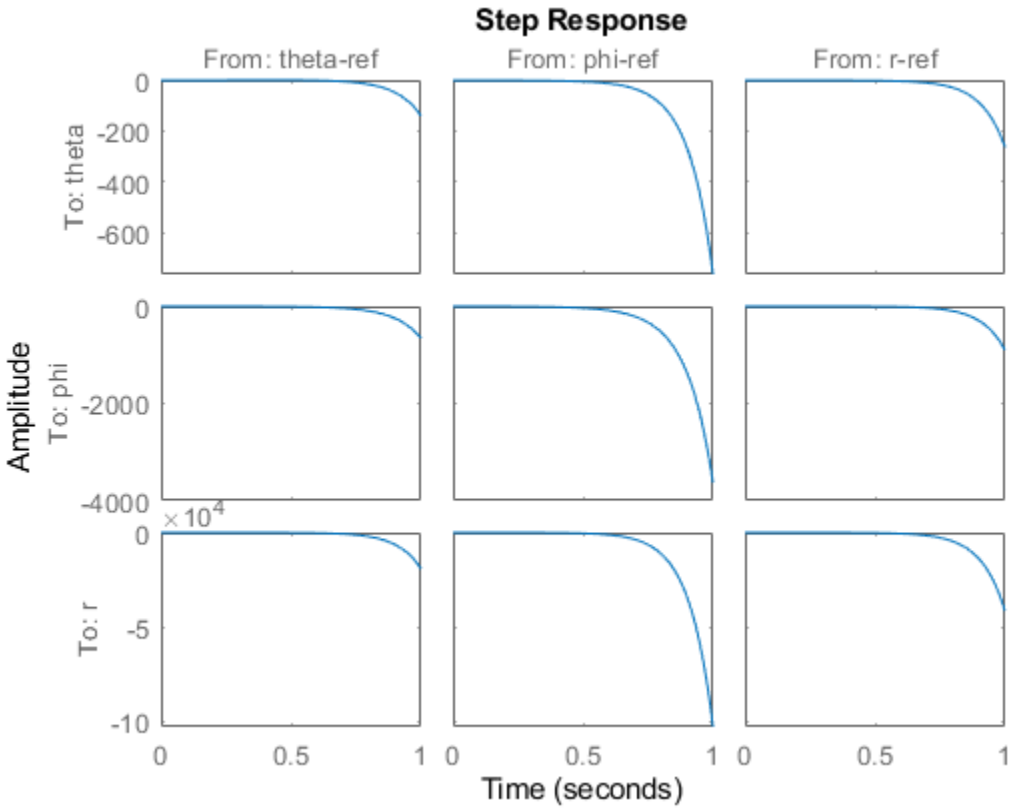
```
ST0
```

```
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'`.

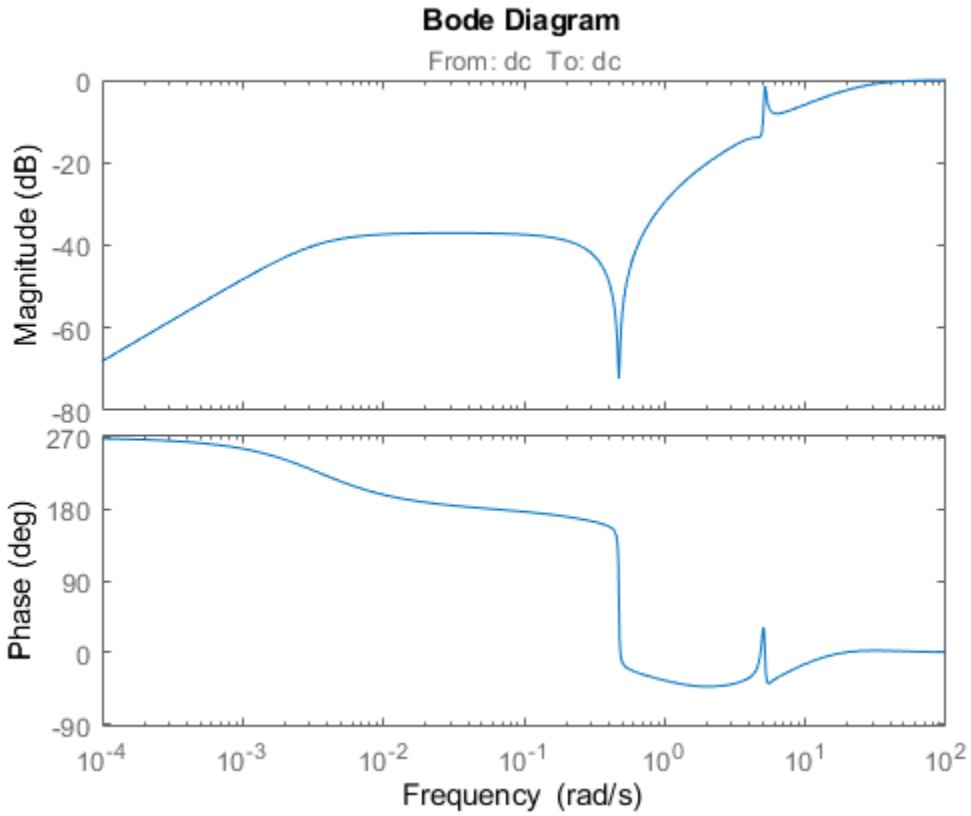
```
T0 = getIOTransfer(ST0,{'theta-ref','phi-ref','r-ref'},{'theta','phi','r'});
stepplot(T0,1)
```



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 2-68
- “Multiloop Control of a Helicopter” on page 18-169
- “Control of a Linear Electric Actuator” on page 18-76

Stability Margins in Control System Tuning

In control system tuning, you specify target gain and phase margins using “Margins Goal” on page 14-110 (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°.

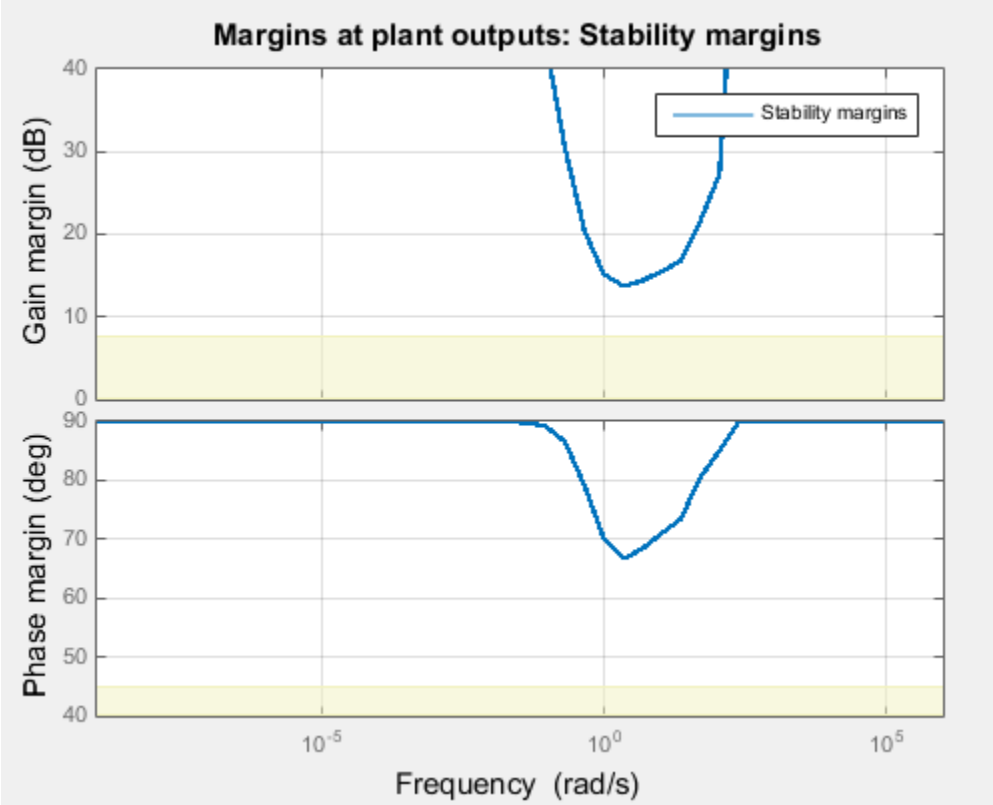
Interpret Gain and Phase Margin Plots

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 14-110 or “Quick Loop Tuning” on page 14-41.
- 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.

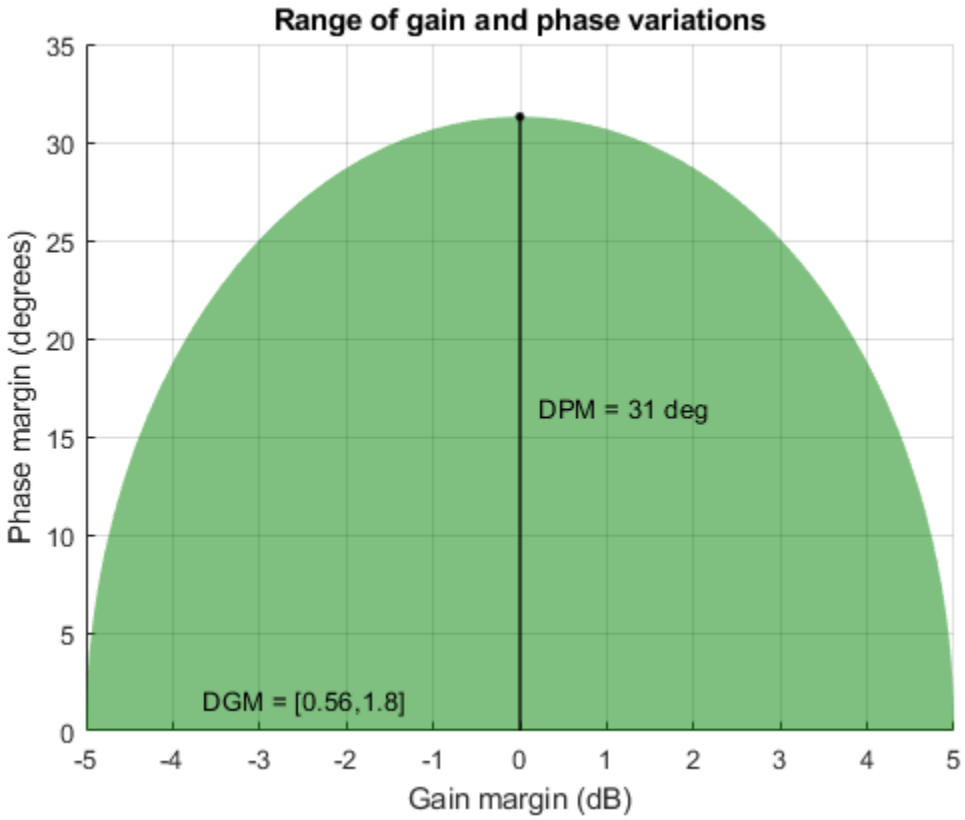


The plot shows that the frequency of the gain or phase variation can affect how large a perturbation the system can tolerate without going unstable. The minimum (worst) gain and phase margins occur at about 2 rad/s. At this frequency, the system can tolerate changes in open-loop gain of about ±14 dB, 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:

$$\frac{1}{\alpha} = \min_{D\,\text{diagonal}} \left\| D(j\omega)^{-1}(I - L(j\omega))(I + L(j\omega))^{-1} D(j\omega) \right\|_\infty .$$

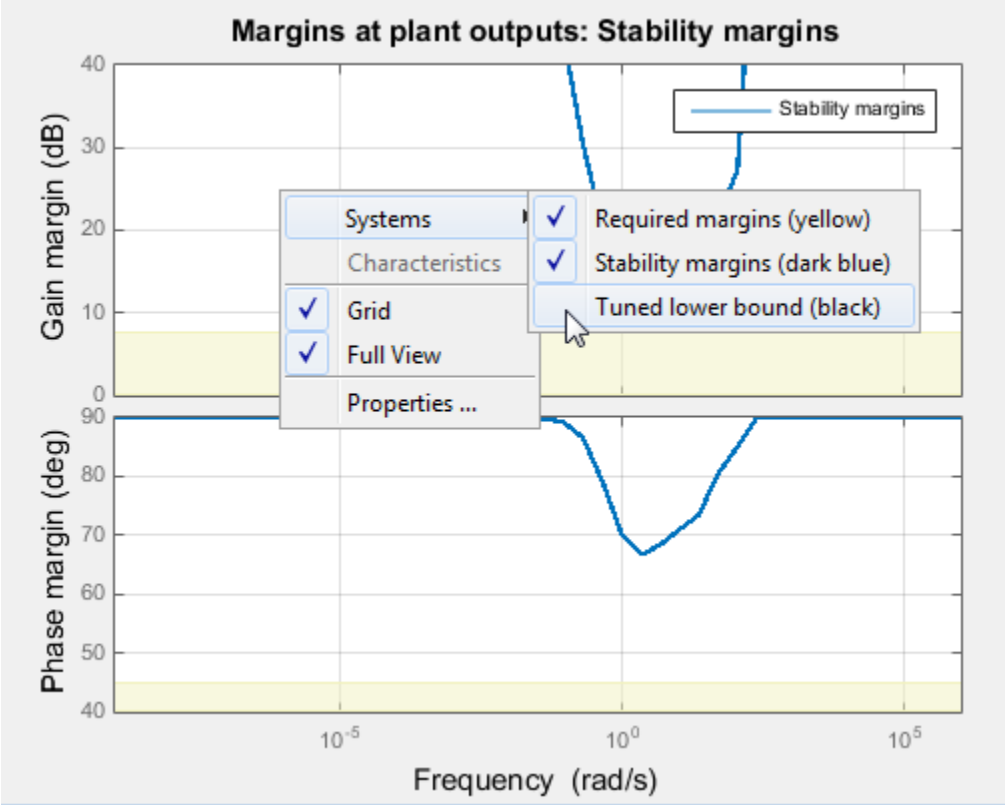
Unlike classical gain and phase margins, the disk margins and associated gain and phase margins guarantee that the open-loop response stays at a safe distance from the critical point at all frequencies.

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 18-34
- “Margins Goal” on page 14-110
- “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 14-138

Speed Up Tuning with Parallel Computing Toolbox Software

Commands for tuning fixed-structure control systems such as `systune`, `looptune`, `hinfstruct`, or `musyn`, you can use the `RandomStart` option to run multiple optimization starts using randomized initial parameter values. Doing so decreases the chances of falling into a local minimum in parameter space and obtaining a controller that does not perform as well as it could. However, additional optimization runs take time. If you have a Parallel Computing Toolbox license, you can use parallel computing to speed up tuning by distributing these independent 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.

Next, create an options set that specifies multiple random starts and sets the `UseParallel` flag to `true`. For example, the following options set specifies 20 random restarts to run in parallel for tuning with `systune`:

```
options = systuneOptions('RandomStart',20,'UseParallel',true);
```

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` | `systuneOptions` | `looptuneOptions` | `hinfstructOptions` | `musynOptions`

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 14-138 and “Visualize Tuning Goals” on page 14-141.) 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 14-141), 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`.

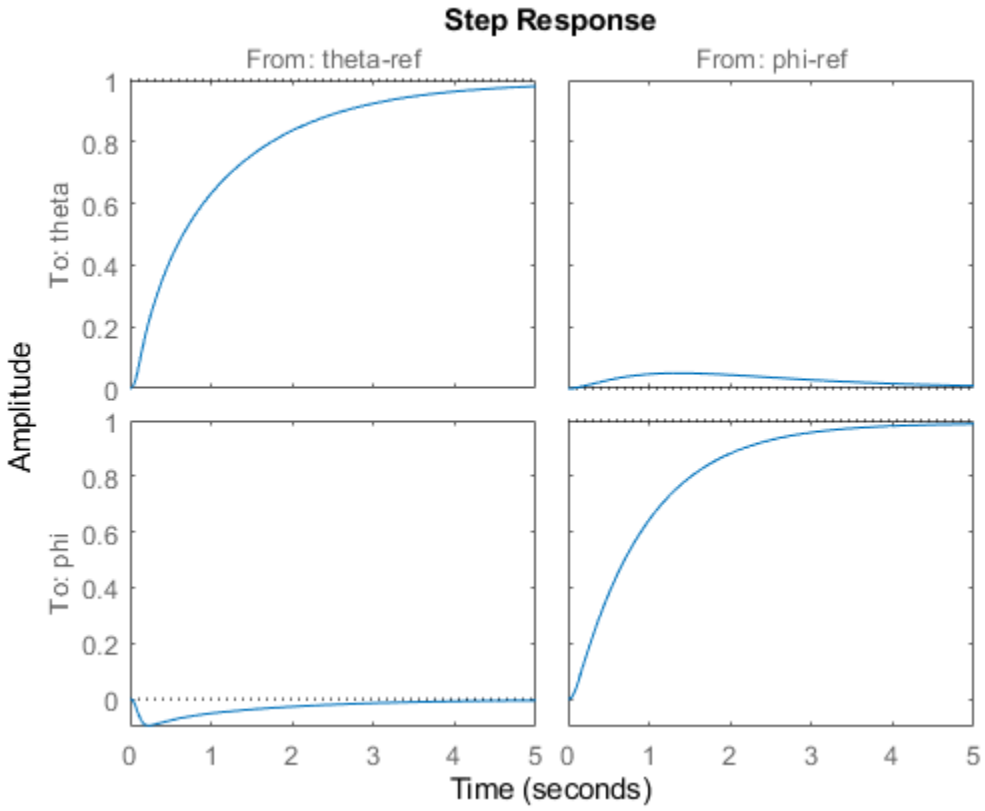
For instance, suppose that you are tuning the control system of the example “Multiloop Control of a Helicopter” on page 18-169. You have created an `slTuner` interface `ST0` for the Simulink model. You have also specified tuning goals `TrackReq`, `MarginReq1`, `MarginReq2`, and `PoleReq`. You tune the 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 14-171.

For additional examples, see “Validating Results” on page 18-42.

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 14-147.

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” (Simulink Control Design).

Write Parameters at the Command Line

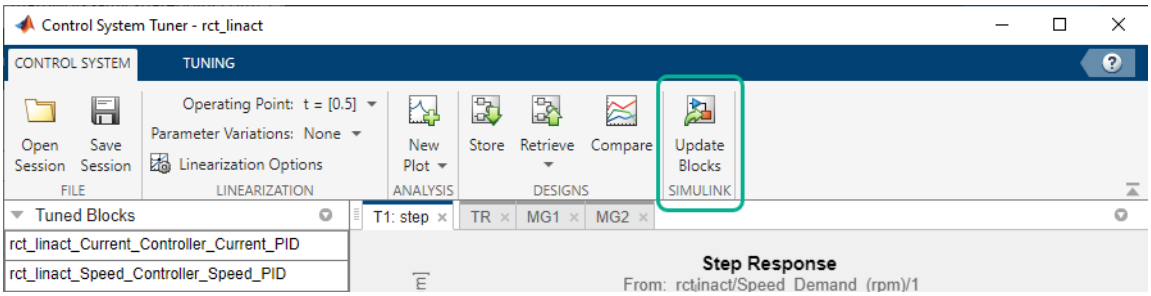
To write tuned block values from a tuned `slTuner` interface to the corresponding Simulink model, use the `writeBlockValue` command. For example, suppose `ST1` is the tuned `slTuner` interface returned by `systune`. The following command writes the tuned parameters from `ST1` to the associated Simulink 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

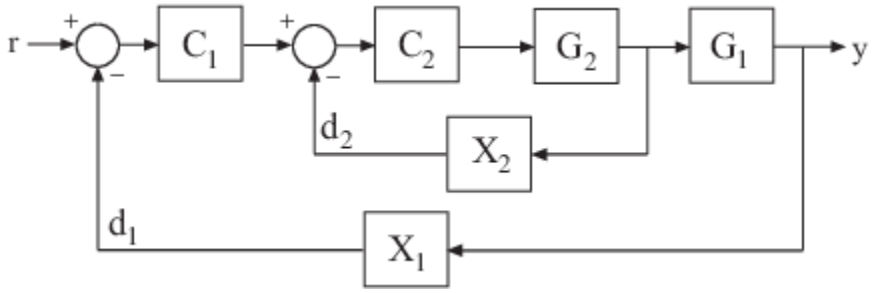
Related Examples

- “Extract Responses from Tuned MATLAB Model at the Command Line” on page 14-171
- “Create Response Plots in Control System Tuner” on page 14-147
- “Visualize Tuning Goals” on page 14-141

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

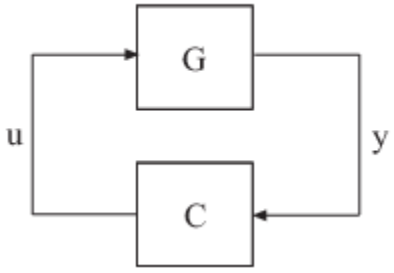
- “Interpret Numeric Tuning Results” on page 14-138

Loop-Shaping Design

- “Structure of Control System for Tuning With looptune” on page 15-2
- “Set Up Your Control System for Tuning with looptune” on page 15-3
- “Tune MIMO Control System for Specified Bandwidth” on page 15-4
- “Tune Feedback Loops Using looptune” on page 15-10
- “Decoupling Controller for a Distillation Column” on page 15-15
- “Tuning of a Digital Motion Control System” on page 15-26

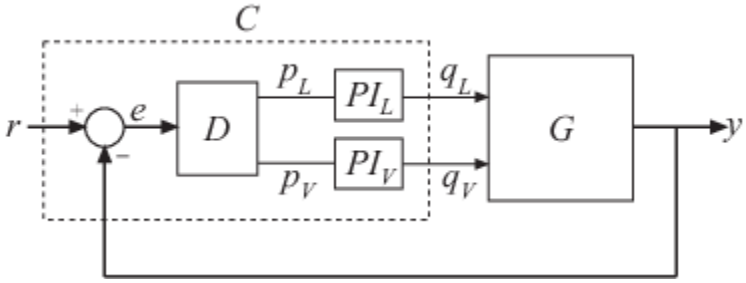
Structure of Control System for Tuning With looptune

`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 on page 1-10 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 14-26.

See Also

Related Examples

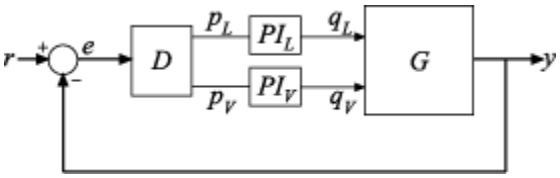
- “Tune MIMO Control System for Specified Bandwidth” on page 15-4
- “Tune Feedback Loops Using looptune” on page 15-10

More About

- “Structure of Control System for Tuning With looptune” on page 15-2

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” on page 15-15.

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.

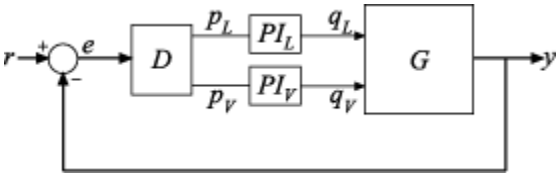
Represent the components of the controller.

```
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 = 1, Iterations = 25
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.

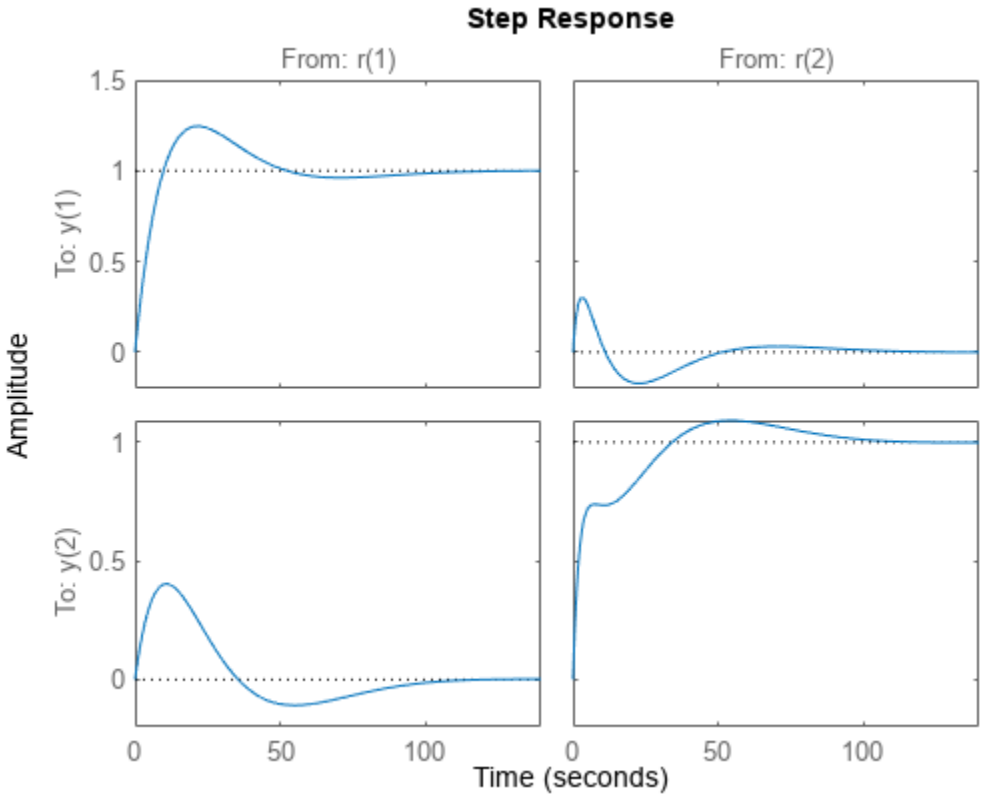
```
showTunable(C)
```

```
Decoupler =

  D =
           u1       u2
   y1    1.266  -0.8781
   y2   -1.505    1.222
```

```
Name: Decoupler
Static gain.
-----------------------------------
PI_L =

            1
  Kp + Ki * ---
            s

  with Kp = 2.19, Ki = 0.131

Name: PI_L
Continuous-time PI controller in parallel form.
-----------------------------------
PI_V =

            1
  Kp + Ki * ---
            s

  with Kp = -1.78, Ki = -0.0905

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.

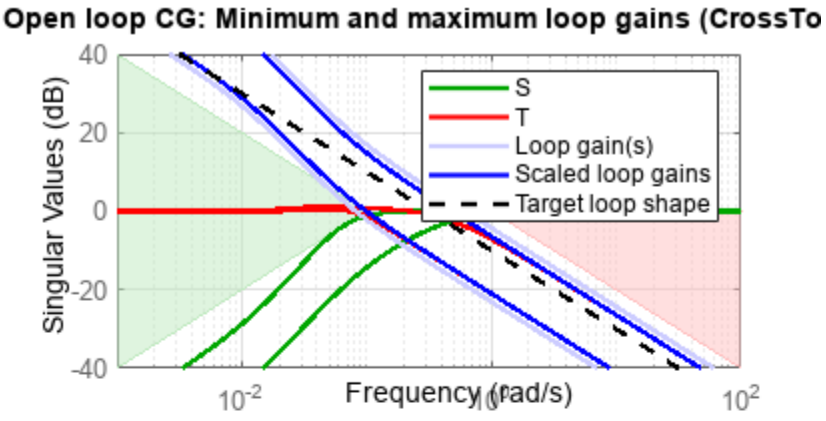
```
T = connect(G,C,'r','y');
step(T)
```

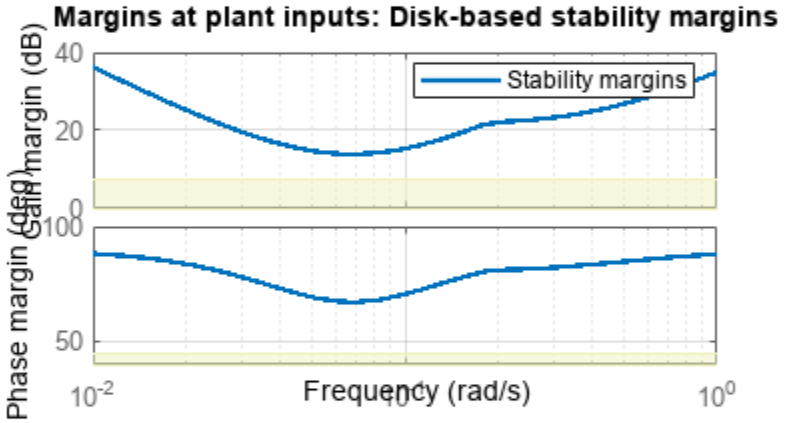


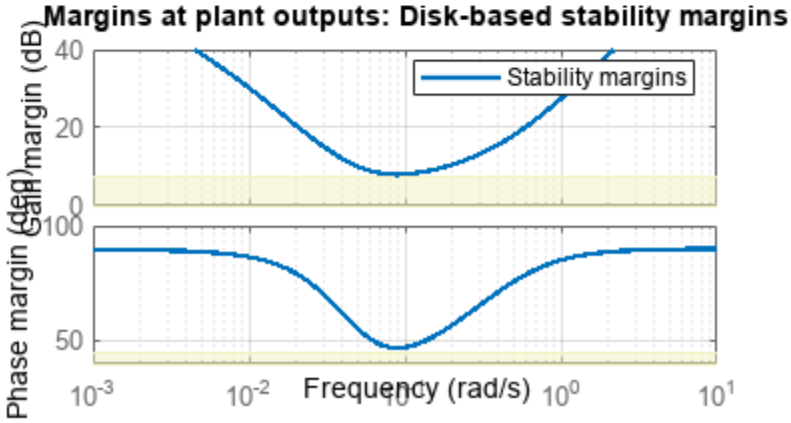
The decoupling matrix in the controller permits each channel of the two-channel output signal `y` to track the corresponding channel of the reference signal `r`, with minimal crosstalk. From the plot, you can how well this requirement is achieved when you tune the control system for bandwidth alone. 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 (CrossTo
Singular Values (dB)
40
20
0
-20
-40
S
T
Loop gain(s)
Scaled loop gains
Target loop shape
10^{-2}
Frequency (rad/s)
10^{2}


Margins at plant inputs: Disk-based stability margins
Gain margin (dB)
Phase margin (deg)
40
20
0
100
50
Stability margins
10^{-2}
Frequency (rad/s)
10^{0}


Margins at plant outputs: Disk-based stability margins
Gain margin (dB)
Phase margin (deg)
40
20
0
100
50
Stability margins
10^{-3}
10^{-2}
Frequency (rad/s)
10^{0}
10^{1}

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” on page 15-15

More About

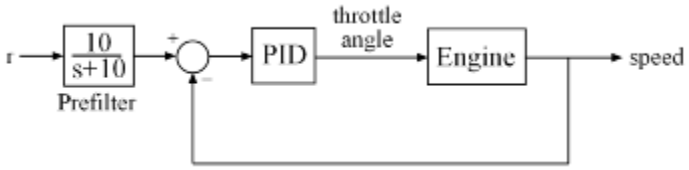
- “Structure of Control System for Tuning With looptune” on page 15-2

Tune Feedback Loops Using looptune

This example shows the basic workflow of tuning feedback loops with the `looptune` command. `looptune` is similar to `systune` and meant to facilitate loop shaping design by automatically generating the tuning requirements.

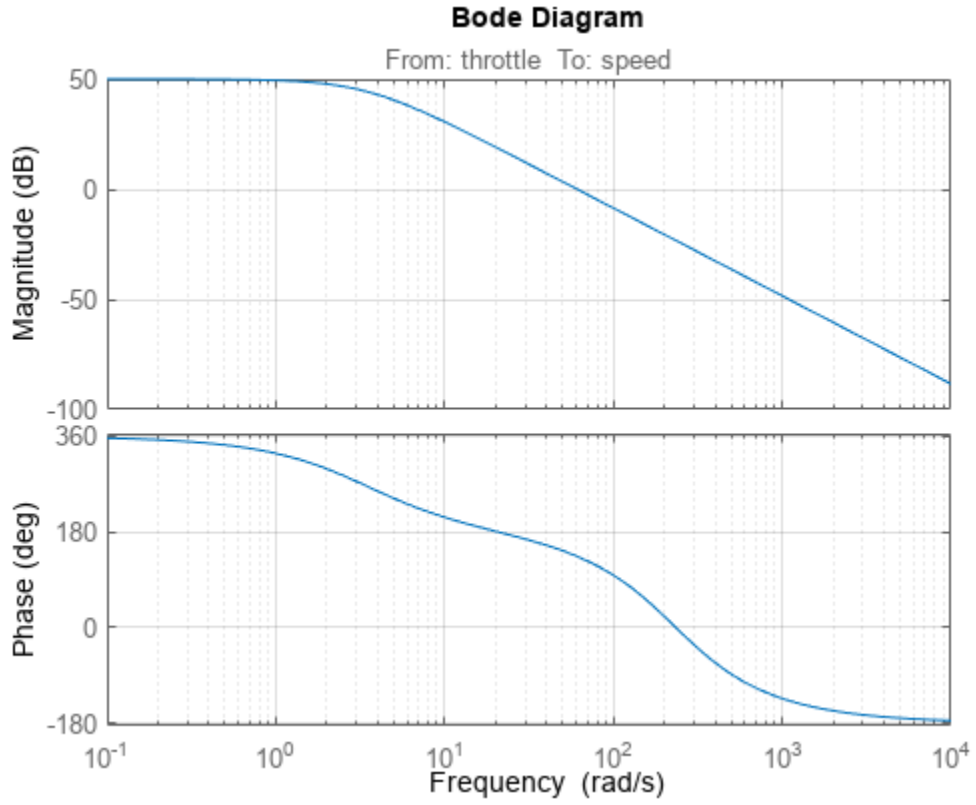
Engine Speed Control

This example uses a simple engine speed control application as shown in the following figure. 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, you want the response to settle in less than 5 seconds with little or no overshoot.



Use the following fourth-order model of the engine dynamics.

```
load rctExamples Engine
bode(Engine)
grid
```



Specify Tunable Elements

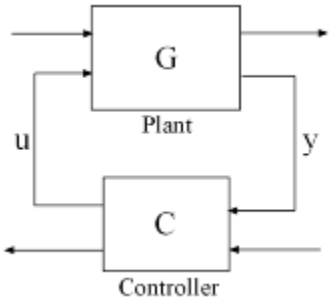
You need to tune the four PID gains to achieve the desired performance. Use a `tunablePID` object to parameterize the PID controller.

```
PID0 = tunablePID('SpeedController','pid')
```

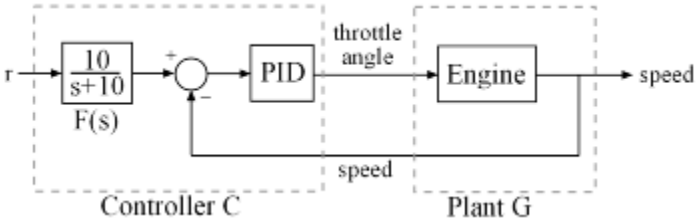
```
Tunable continuous-time PID controller "SpeedController" with formula:

                1             s
  Kp + Ki * --- + Kd * --------
                s          Tf*s+1

and tunable parameters Kp, Ki, Kd, Tf.

Type "pid(PID0)" to see the current value.
```

Build Tunable Model of Feedback Loop

`looptune` tunes the generic SISO or MIMO feedback loop shown in the following figure. This feedback loop models the interaction between the plant and the controller. Note that this is a *positive* feedback interconnection.



For the speed control loop, the plant G is the engine model and the controller C consists of a PID controller and prefilter F.



To use `looptune`, create models for G and C. Assign names to the inputs and outputs of each model to specify the feedback paths between plant and controller. Note that the controller C has two inputs: the speed reference `r` and the speed measurement `speed`.

```
F = tf(10,[1 10]);  % prefilter

G = Engine;
G.InputName = 'throttle';
G.OutputName = 'speed';
```

```
C0 = PID0 * [F , -1];
C0.InputName = {'r','speed'};
C0.OutputName = 'throttle';
```

Here, `C0` is a generalized state-space model (`genss`) that depends on the tunable PID block `PID0`.

Tune Controller Parameters

You can now use `looptune` to tune the PID gains subject to a simple control bandwidth requirement. To achieve the 5-second settling time, the gain crossover frequency of the open-loop response should be approximately 1 rad/s. Given this basic requirement, `looptune` automatically shapes the open-loop response to provide integral action, high-frequency roll-off, and adequate stability margins. Note that you could specify additional requirements to further constrain the design. For an example, see “Decoupling Controller for a Distillation Column” on page 15-15.

```
wc = 1;  % target gain crossover frequency

[~,C,~,Info] = looptune(G,C0,wc);
```

```
Final: Peak gain = 0.965, Iterations = 5
Achieved target gain value TargetGain=1.
```

The final value is less than 1, indicating that the desired bandwidth was achieved with adequate roll-off and stability margins. `looptune` returns the tuned controller `C`. Use `getBlockValue` to retrieve the tuned value of the PID block.

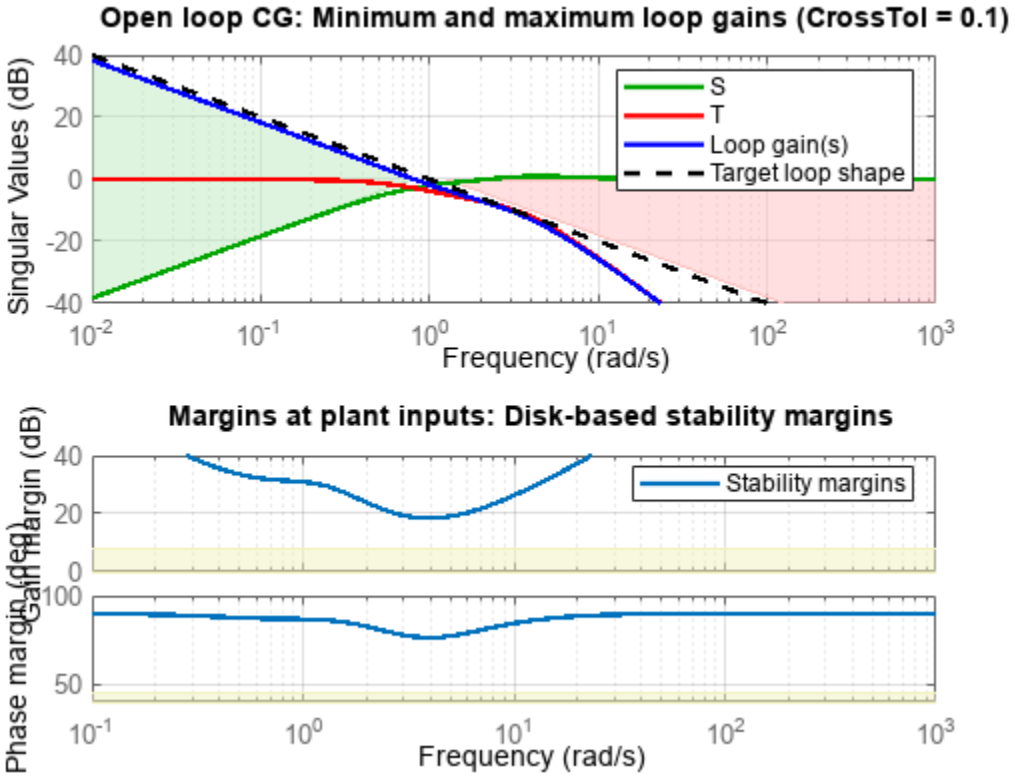
```
PIDT = getBlockValue(C,'SpeedController')
```

```
PIDT =

             1            s
  Kp + Ki * --- + Kd * --------
             s          Tf*s+1

  with Kp = 0.00111, Ki = 0.00253, Kd = 0.000359, Tf = 1

Name: SpeedController
Continuous-time PIDF controller in parallel form.
```

Validate Results

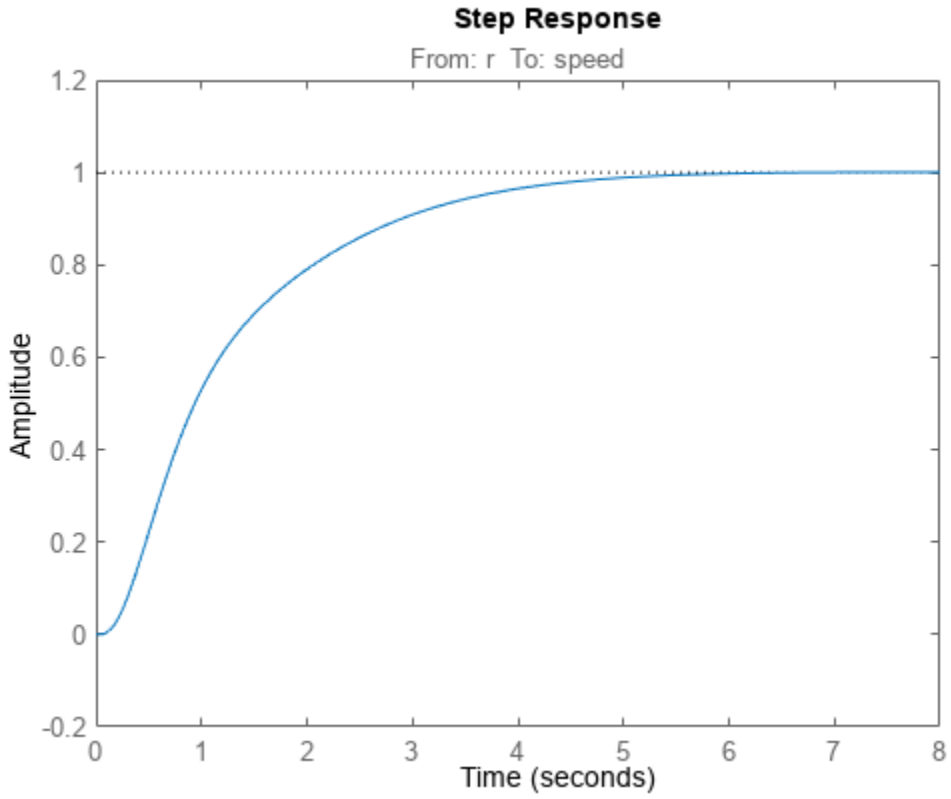
Use `loopview` to validate the design and visualize the loop shaping requirements implicitly enforced by `looptune`.

```
clf
loopview(G,C,Info)
```



Next, plot the closed-loop response to a step command in engine speed. The tuned response satisfies the requirements.

```
T = connect(G,C,'r','speed');  % closed-loop transfer function from r to speed
clf
step(T)
```



See Also

`looptune`

More About

- “Decoupling Controller for a Distillation Column” on page 15-15

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.

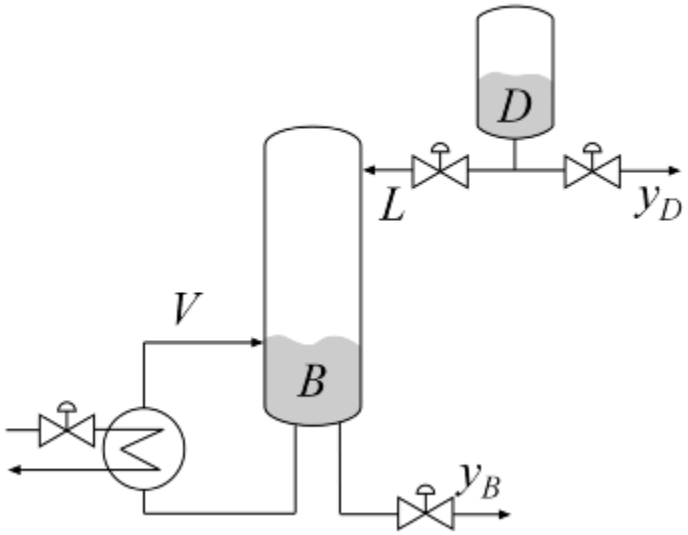


Figure 1: Distillation Column

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'};
```

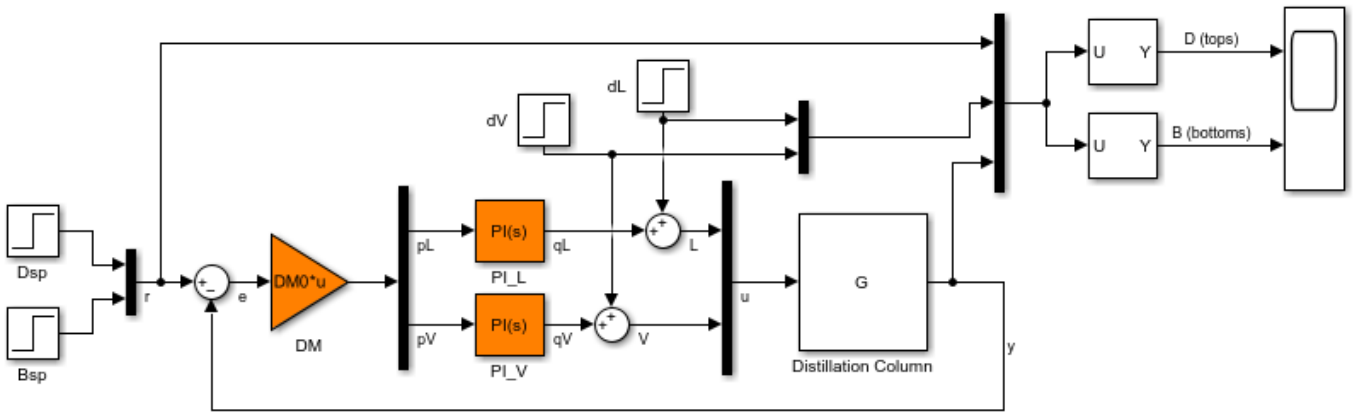
Control Architecture

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);
```

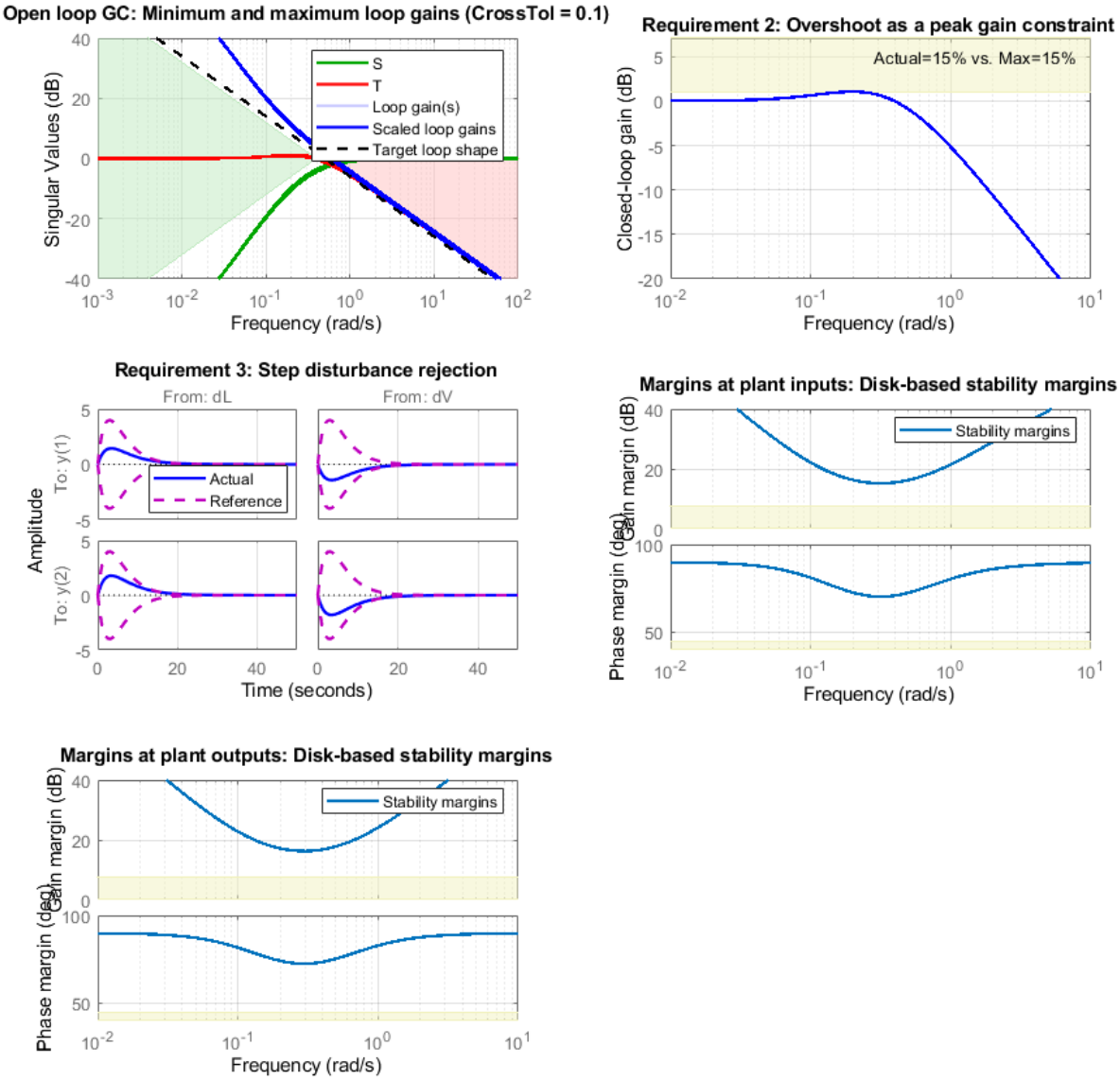
Next use `looptune` to tune the controller blocks `PI_L`, `PI_V`, and `DM` subject to the disturbance rejection requirement.

```
Controls = {'L','V'};
Measurements = 'y';
[ST,gam,Info] = looptune(ST0,Controls,Measurements,wc,OS,DR);
```

```
Final: Peak gain = 1, Iterations = 61
Achieved target gain value TargetGain=1.
```

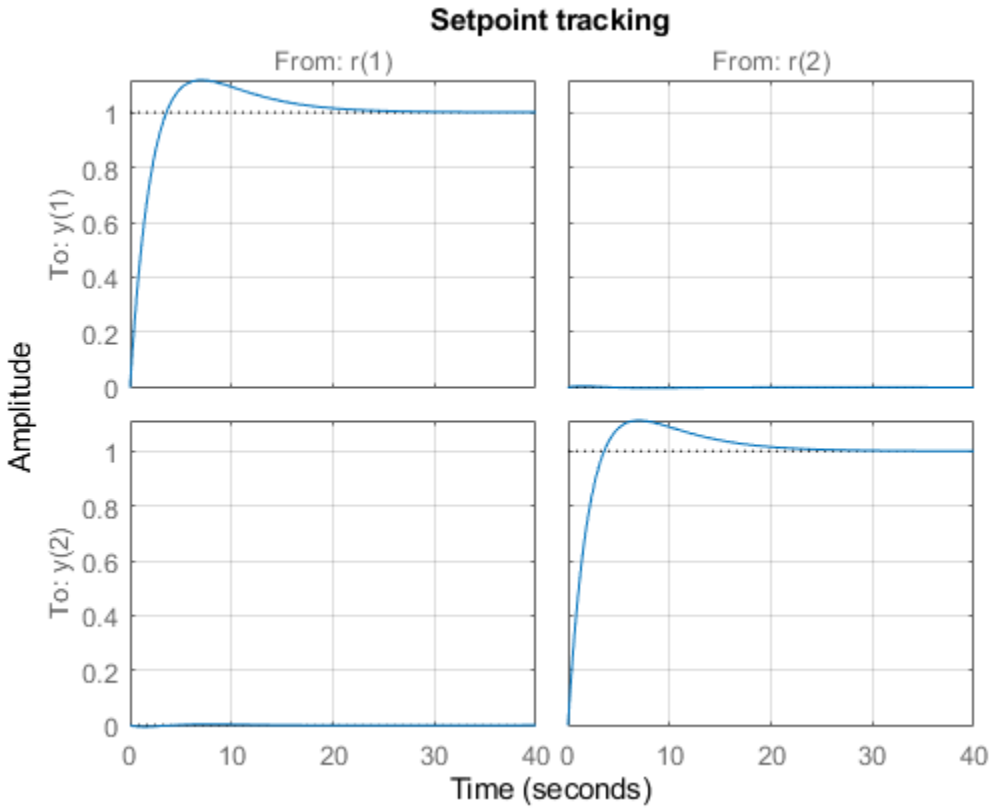
The final value is near 1 which indicates that all requirements were met. Use `loopview` to check the resulting design. The responses should stay outside the shaded areas.

```
figure('Position',[0,0,1000,1200])
loopview(ST,Info)
```

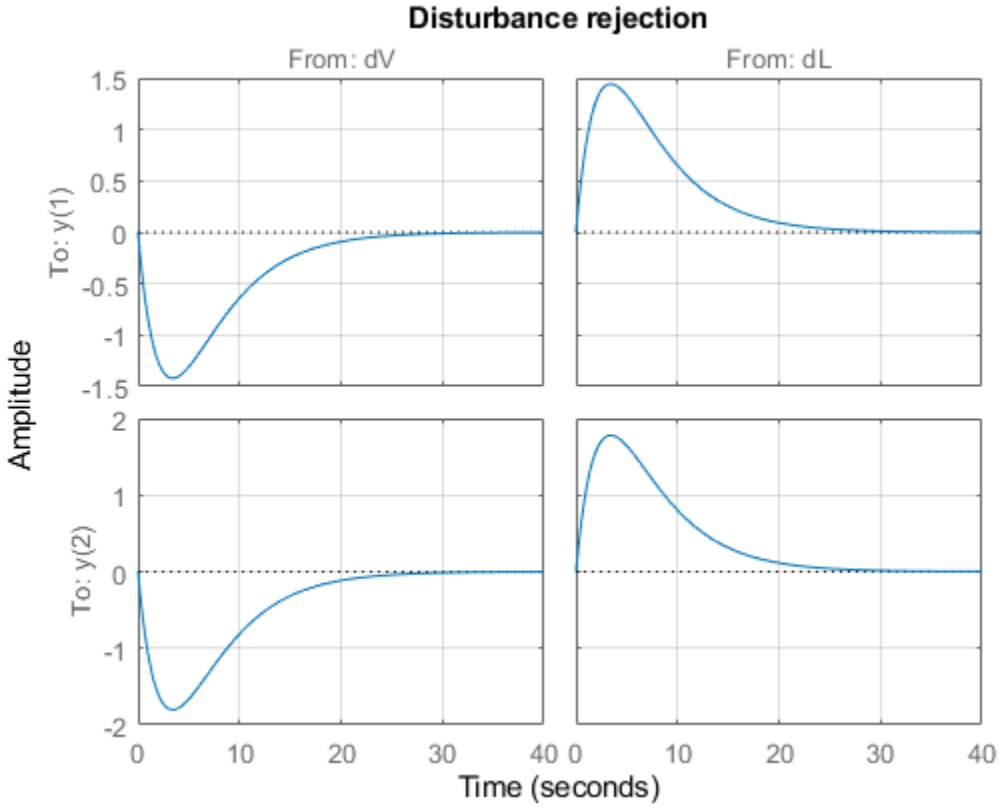


Use `getIOTransfer` to access and plot the closed-loop responses from reference and disturbance to the tops and bottoms concentrations. The tuned responses show a good compromise between tracking and disturbance rejection.

```
figure
Ttrack = getIOTransfer(ST,'r','y');
step(Ttrack,40), grid, title('Setpoint tracking')
```

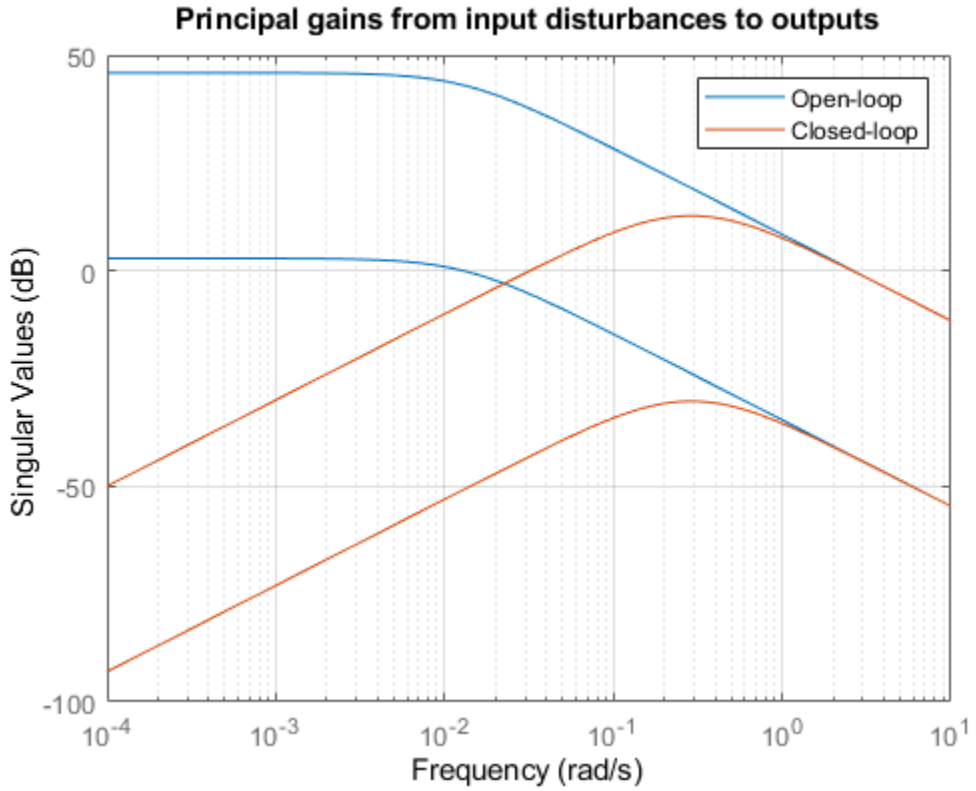



```
Treject = getIOTransfer(ST,{'dV','dL'},'y');
step(Treject,40), grid, title('Disturbance rejection')
```



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 = -4.21, Ki = -0.59

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)
```

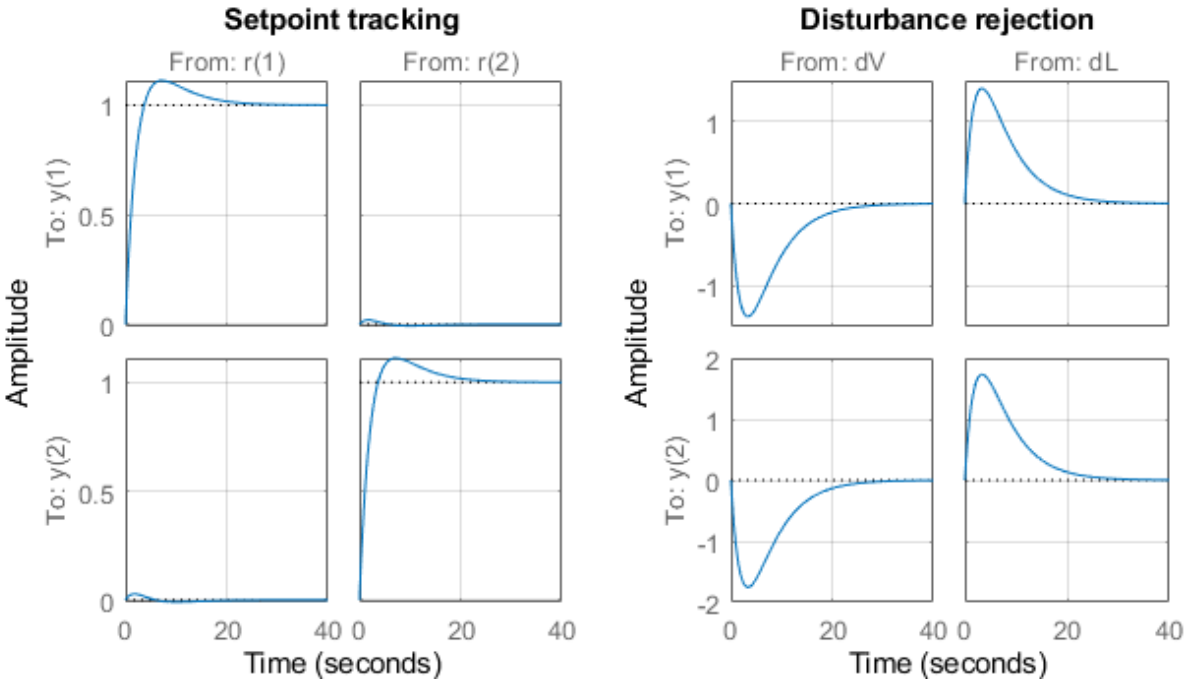
Re-tune the controller for the reduced set of tunable parameters.

```
[ST,gam,Info] = looptune(ST0,Controls,Measurements,wc,OS,DR);
```

```
Final: Peak gain = 0.998, Iterations = 88
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 = 16.4, Ki = 2.21

Name: PI_L
Continuous-time PI controller in parallel form.
```

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

Block 2: rct_distillation/PI_V =

             1
  Kp + Ki * ---
             s

  with Kp = 13.1, Ki = 1.76

Name: PI_V
Continuous-time PI controller in parallel form.

-----------------------------------

Block 3: rct_distillation/DM =

  D =
            u1       u2
   y1        1  -0.7834
   y2    1.235       -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.

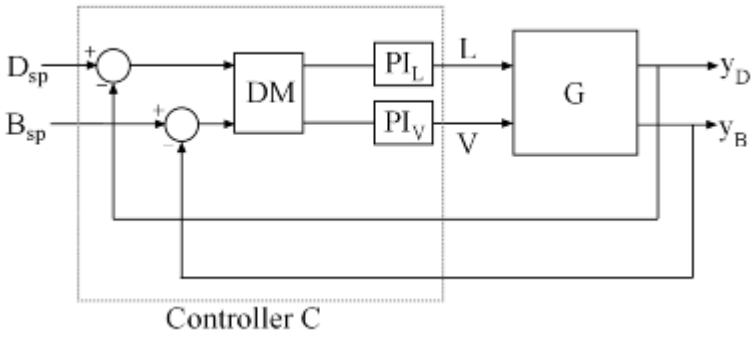


Figure 2: Block Diagram of Control System

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'};
```

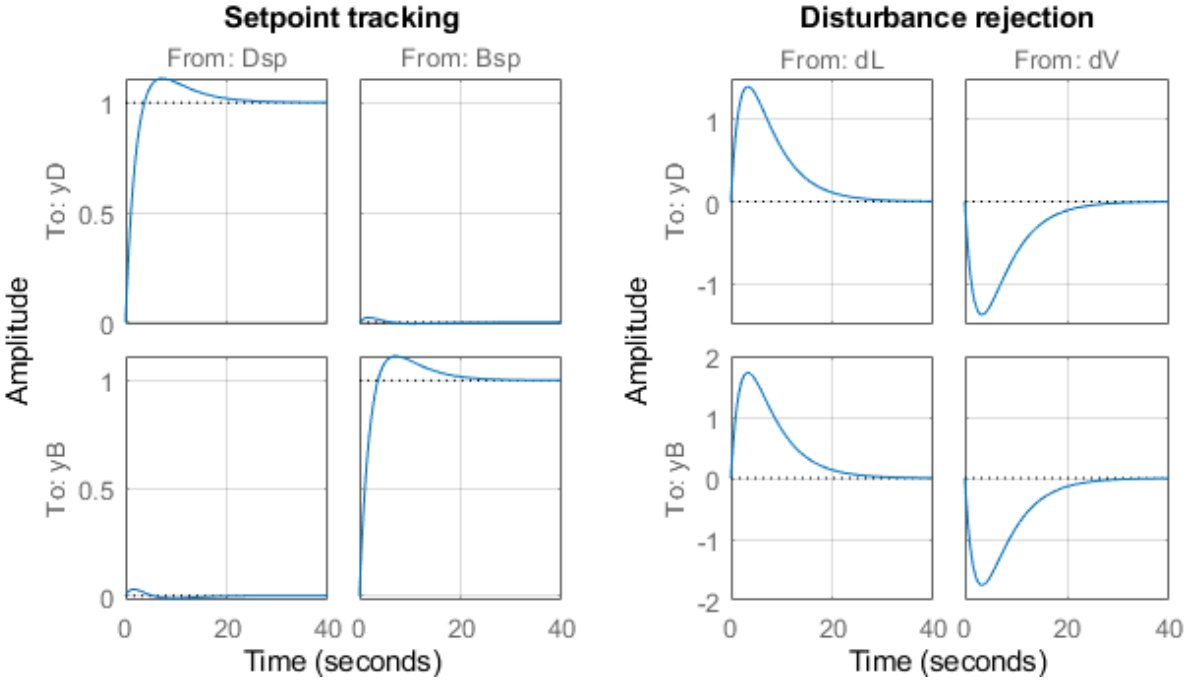
Now tune the controller parameters with `looptune` as done previously.

```
% 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 = 0.998, Iterations = 58
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” on page 15-26

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.

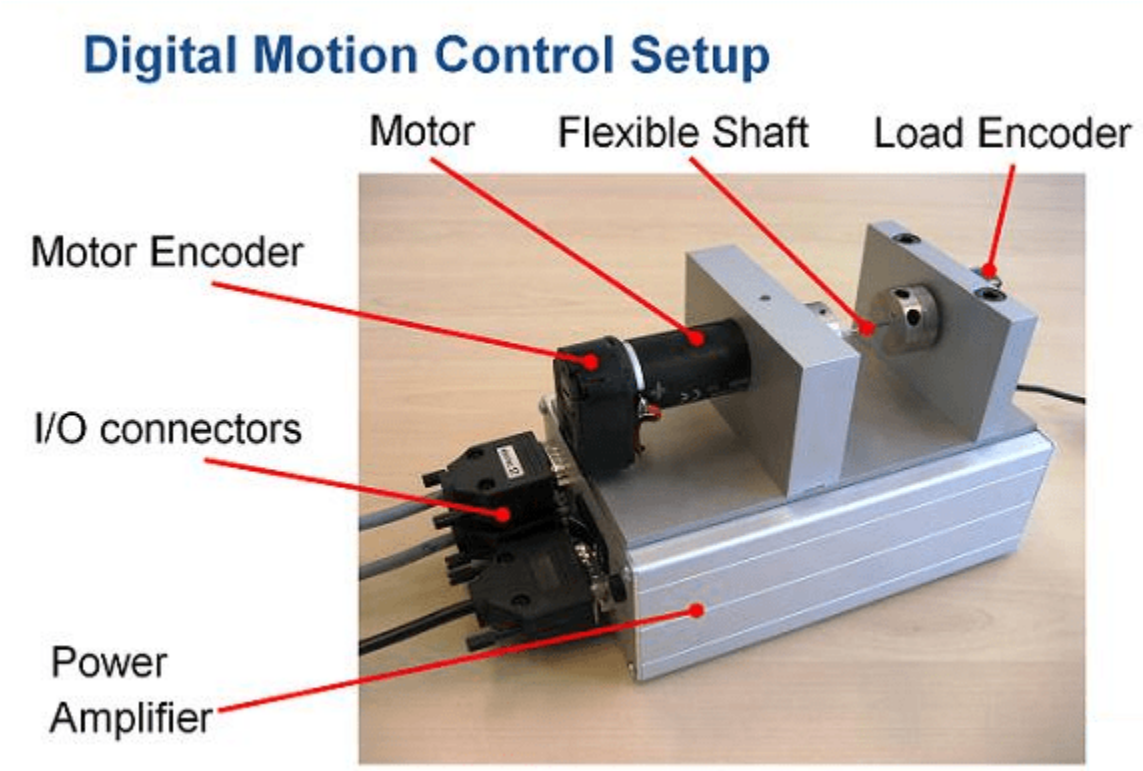
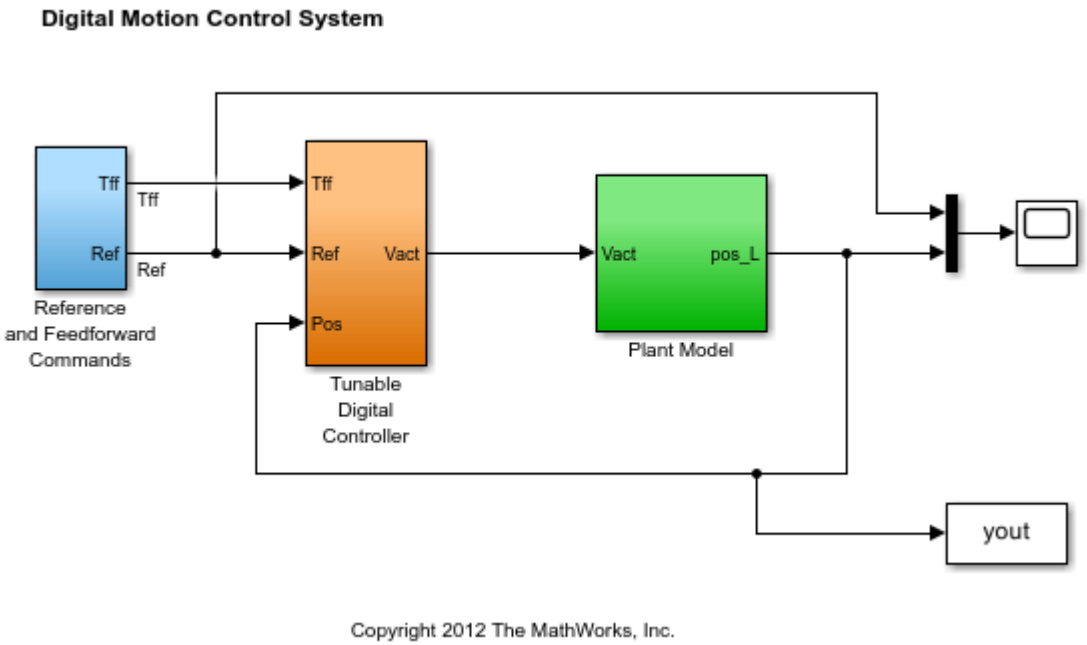


Figure 1: Digital motion control hardware

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.

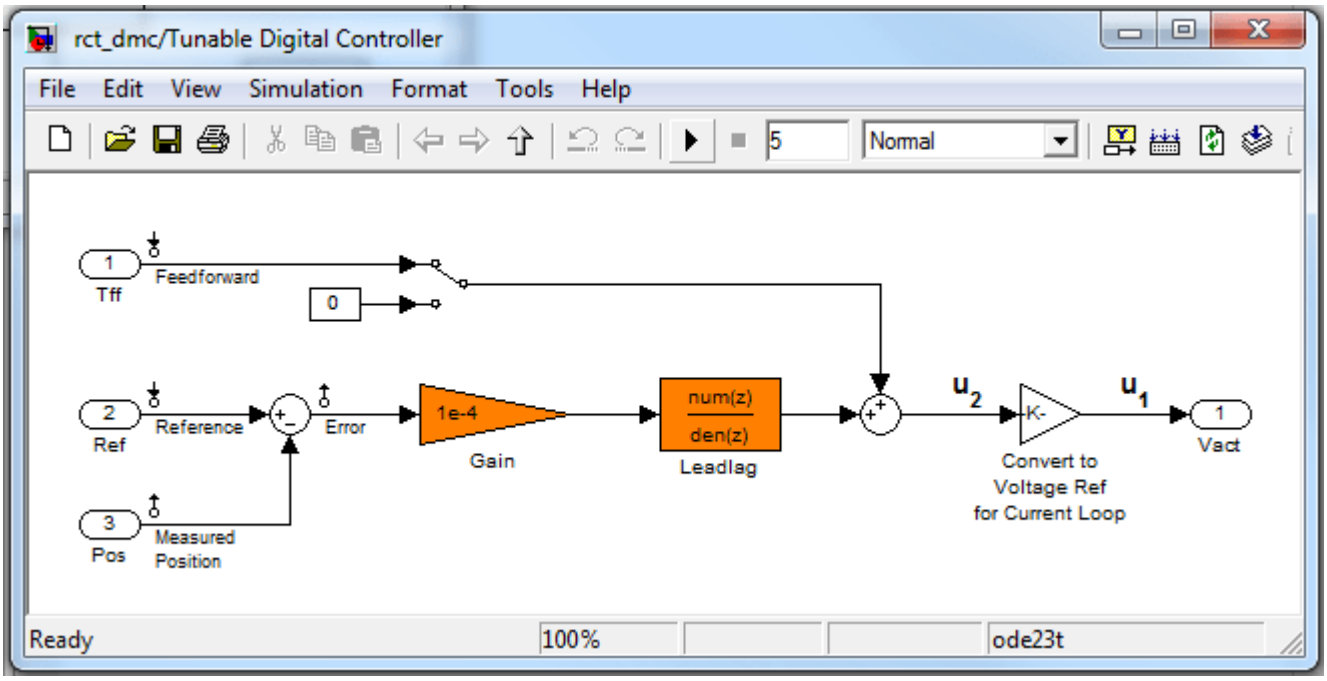
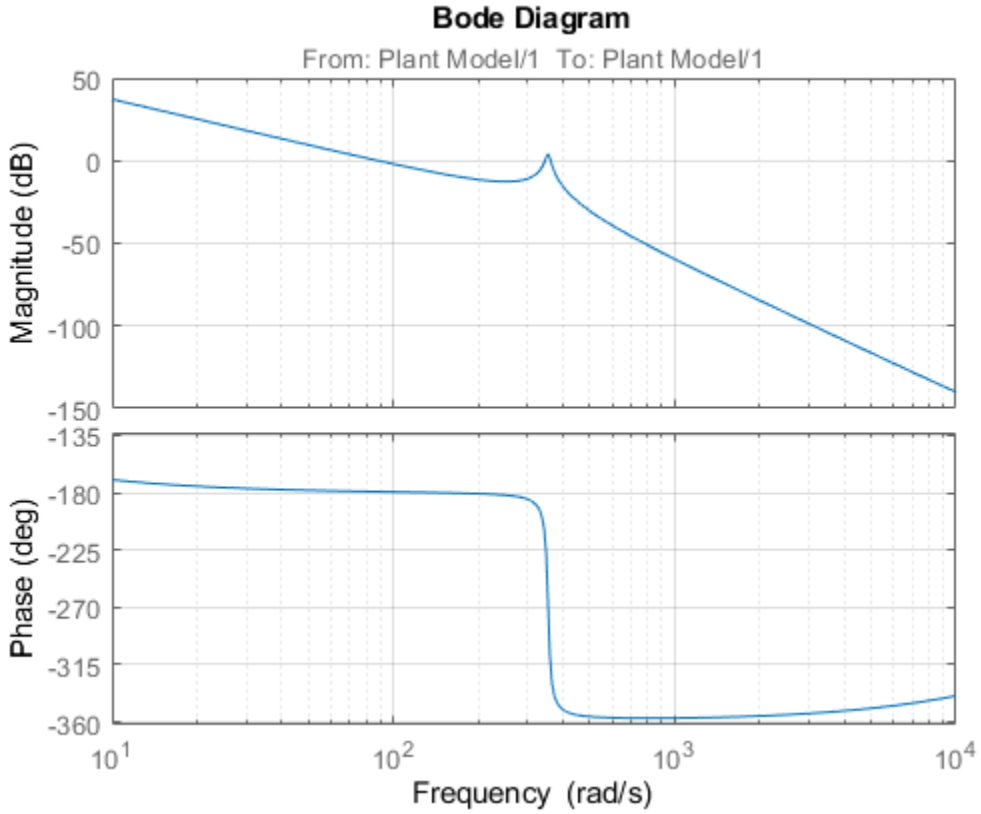


Figure 2: Digital controller

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)
```

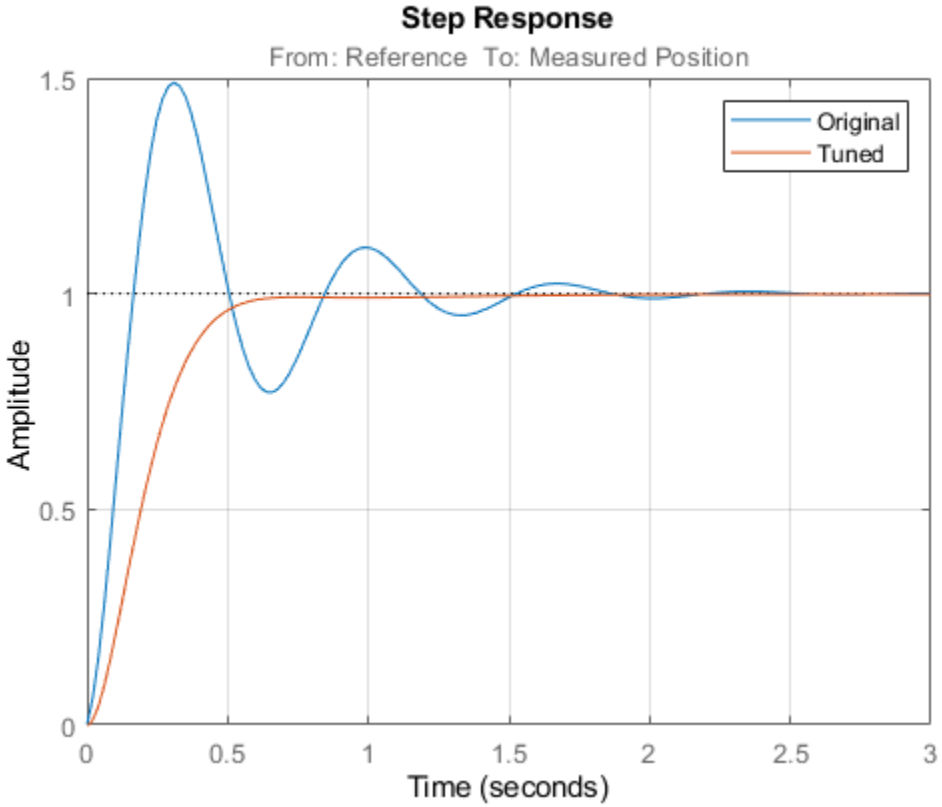
```
Block 1: rct_dmc/Tunable Digital Controller/Gain =
```

```
  D =
              u1
   y1  1.869e-05

Name: Gain
Static gain.

-----------------------------------

Block 2: rct_dmc/Tunable Digital Controller/Leadlag =

  3.855 s + 6.322
  ---------------
     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')
```



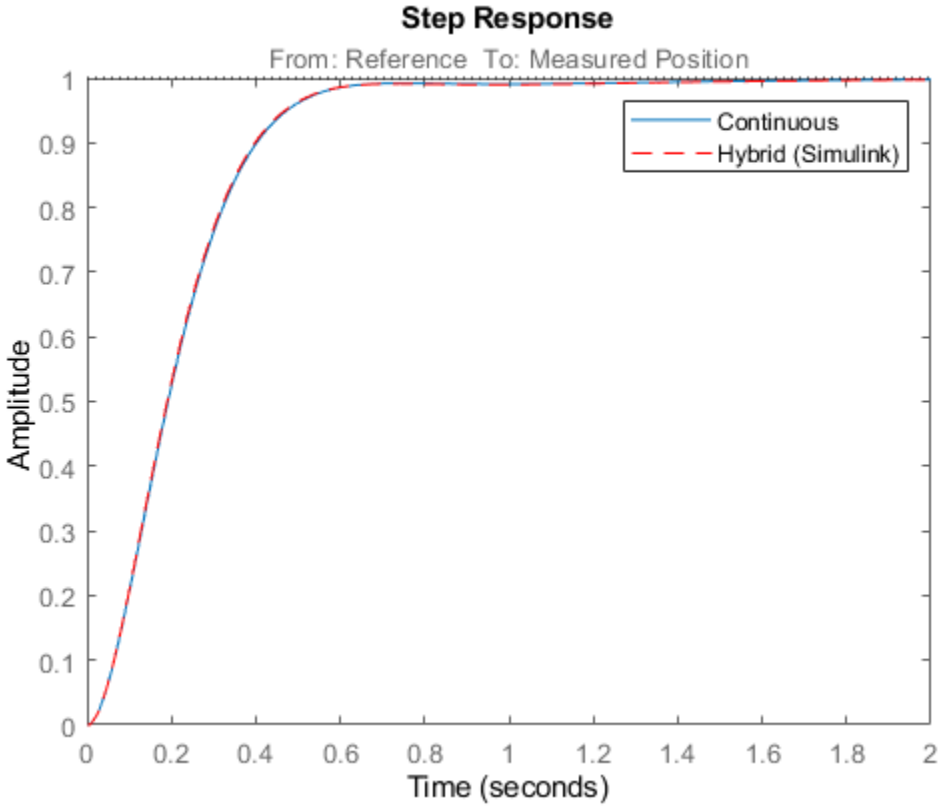
The tuned response has significantly less overshoot and satisfies the response time requirement. However these simulations are obtained using a continuous-time lead/lag compensator (`looptune` operates in continuous time) so we need to further validate the design in Simulink using a digital implementation 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.

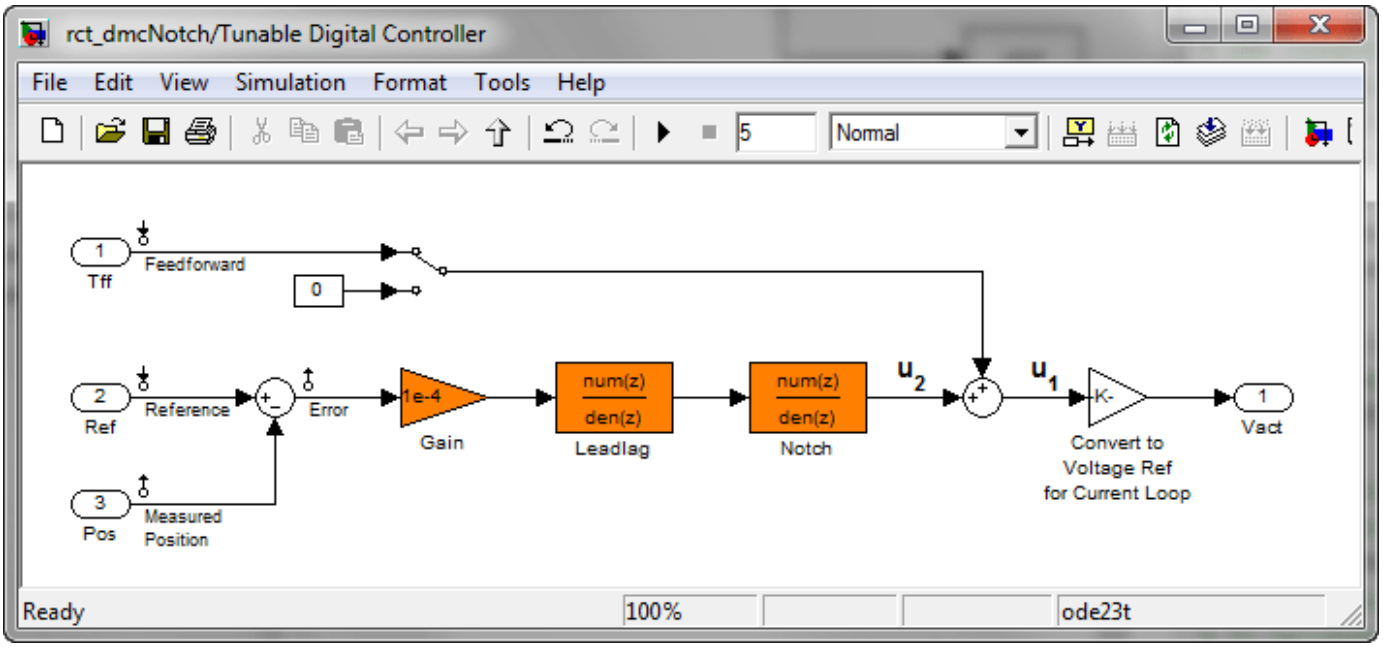


Figure 3: Digital Controller with Notch Filter

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.

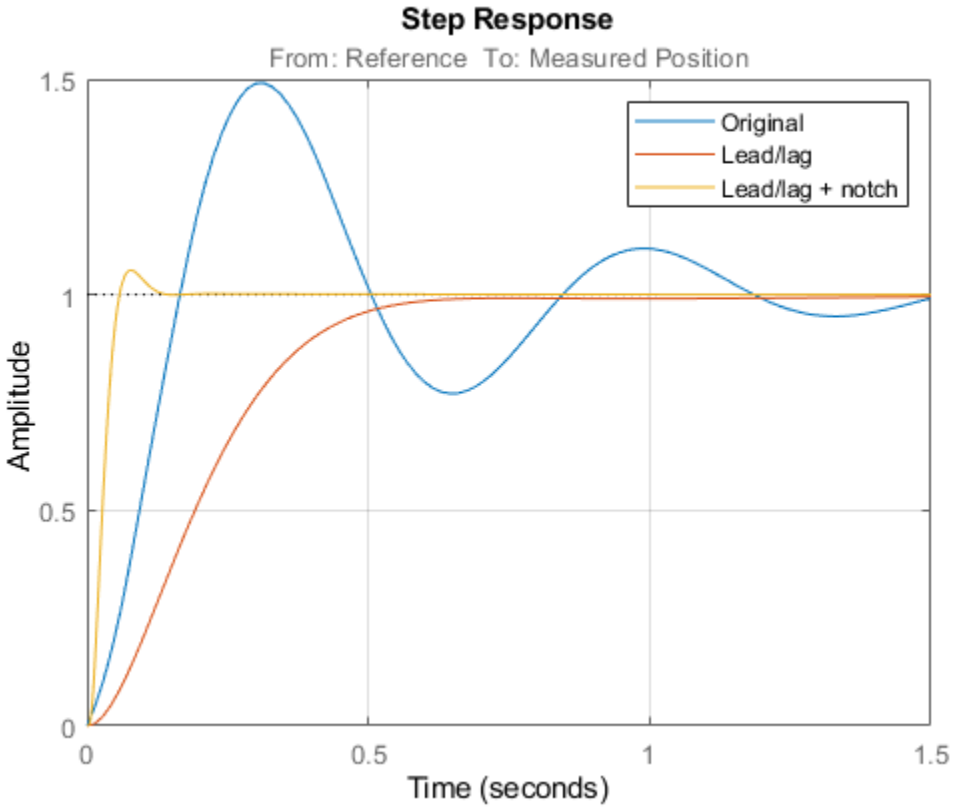
```
% Specify target loop shape with a few frequency points
Freqs = [5 50 500];
```

```
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
```

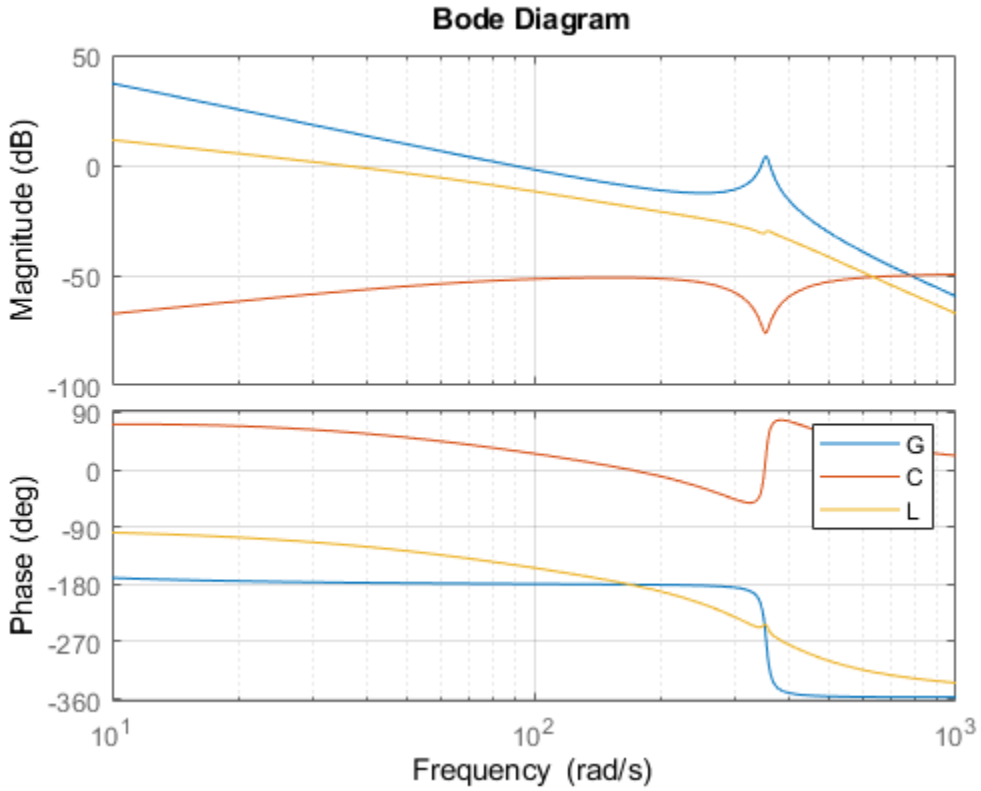
The final gain is close to 1, indicating that all requirements are met. Compare the closed-loop step response with the previous designs.

```
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')
```

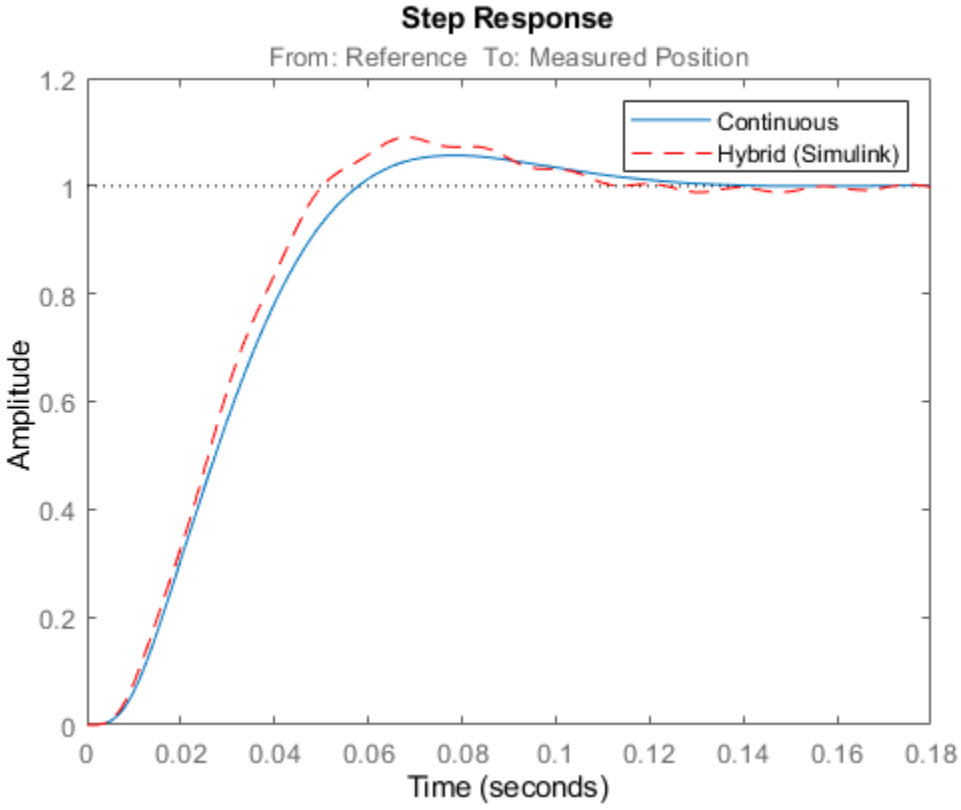


This Bode plot confirms that the plant resonance has been correctly "notched out."

Discretizing the Notch Filter

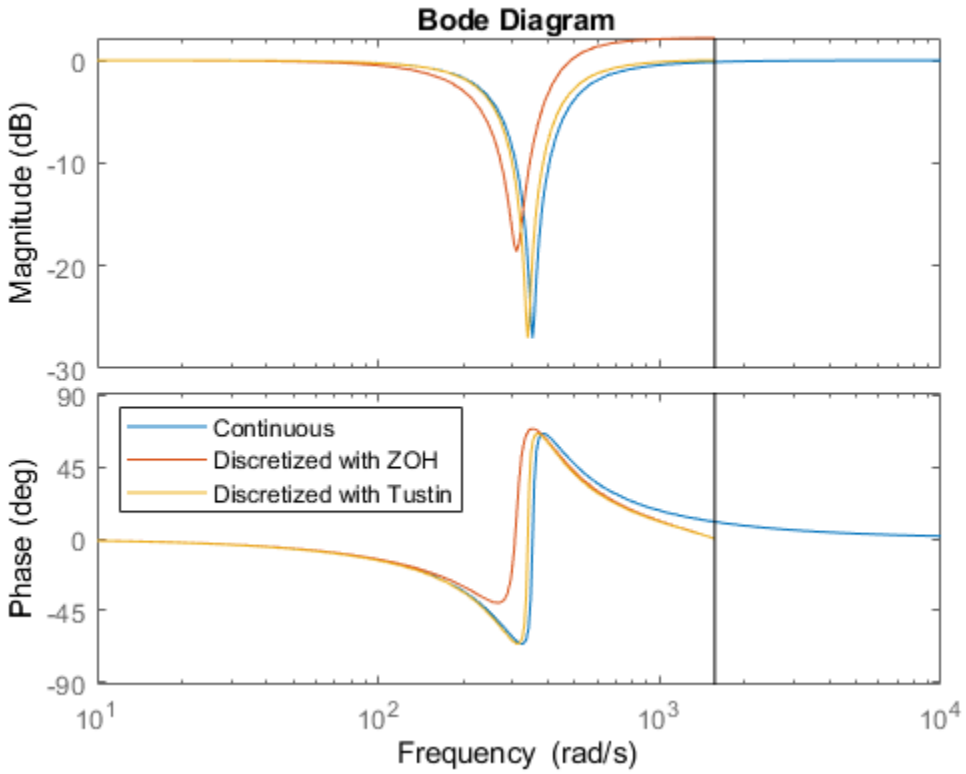
Again use `writeBlockValue` to discretize the tuned lead/lag and notch filters and write their values back to Simulink. Compare the MATLAB and Simulink responses:

```
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')
```

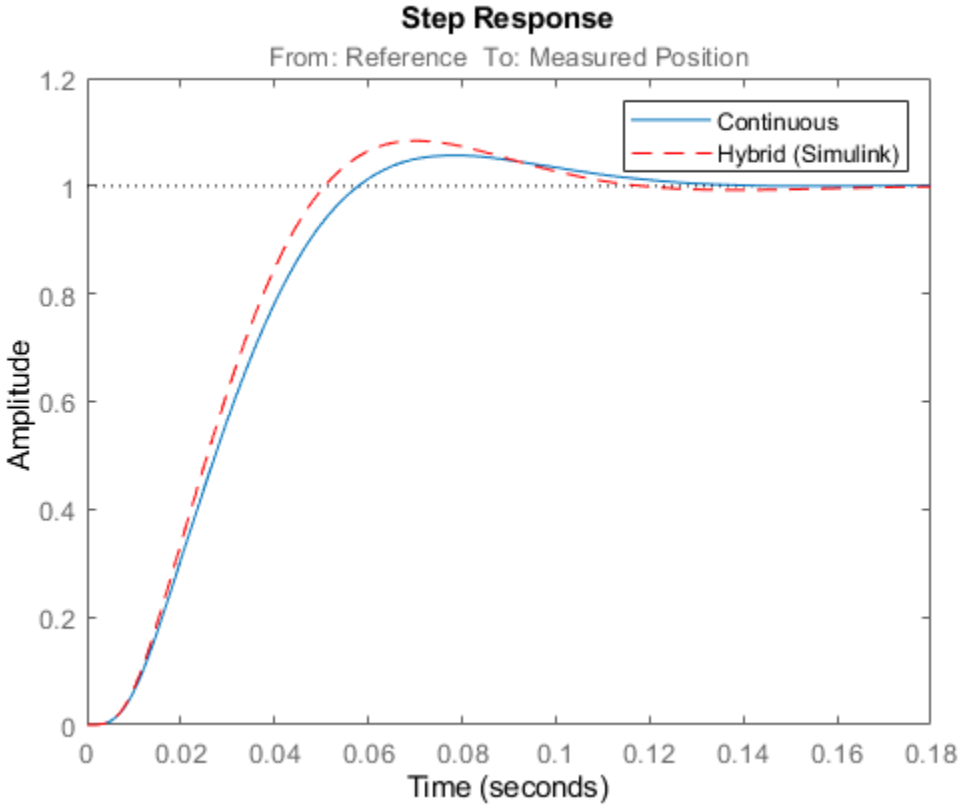


The ZOH method has significant distortion and prewarped Tustin should be used instead. To do this, specify the desired rate conversion method for the notch filter block:

```
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;
```

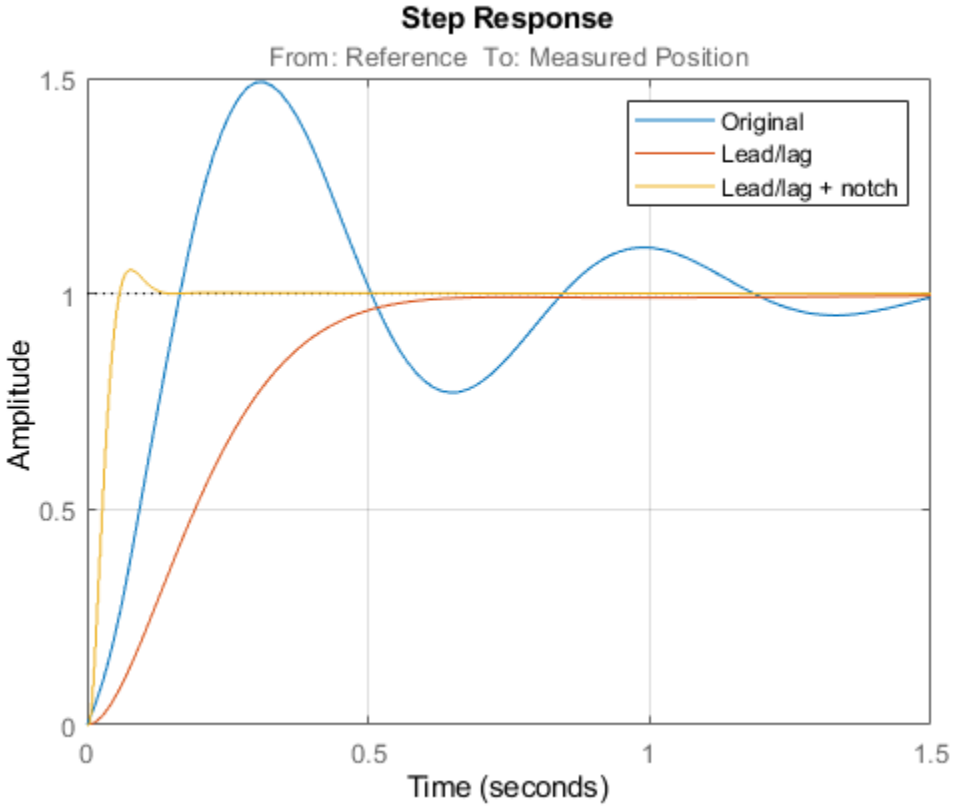
To prevent high-gain control and saturations, add a requirement that limits the gain from reference to control signal (output of Notch block).

```
GL = TuningGoal.Gain('Reference','Notch',0.01);
```

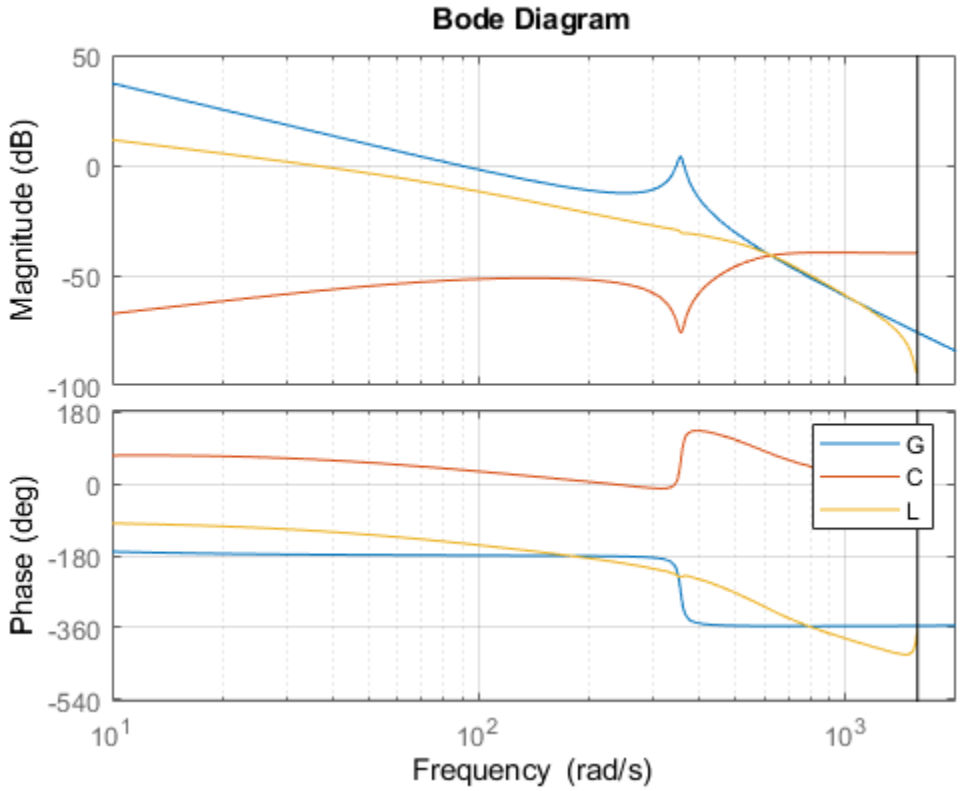
Now retune the controller at the specified sampling rate and verify the tuned open- and closed-loop responses.

```
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 = 37
```



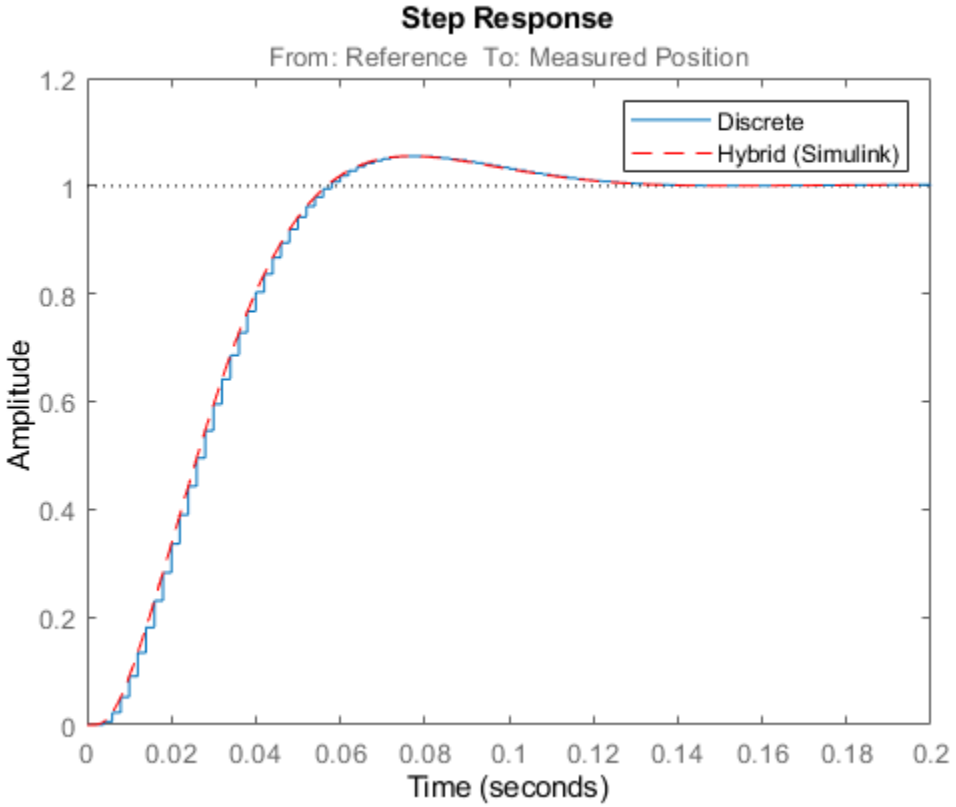

```
% Open-loop responses
[g,LL,N] = getBlockValue(ST2,'Gain','Leadlag','Notch');
C = N * LL * g;
L = getLoopTransfer(ST2,'Notch',-1);
bode(G,C,L,{1e1,2e3}), grid
legend('G','C','L')
```



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” on page 15-10

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 16-4.

Tune Gain Schedules

If you have Simulink Control Design, 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 16-12.

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 16-4
- “Tune Gain Schedules in Simulink” on page 16-12

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 16-12.
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.

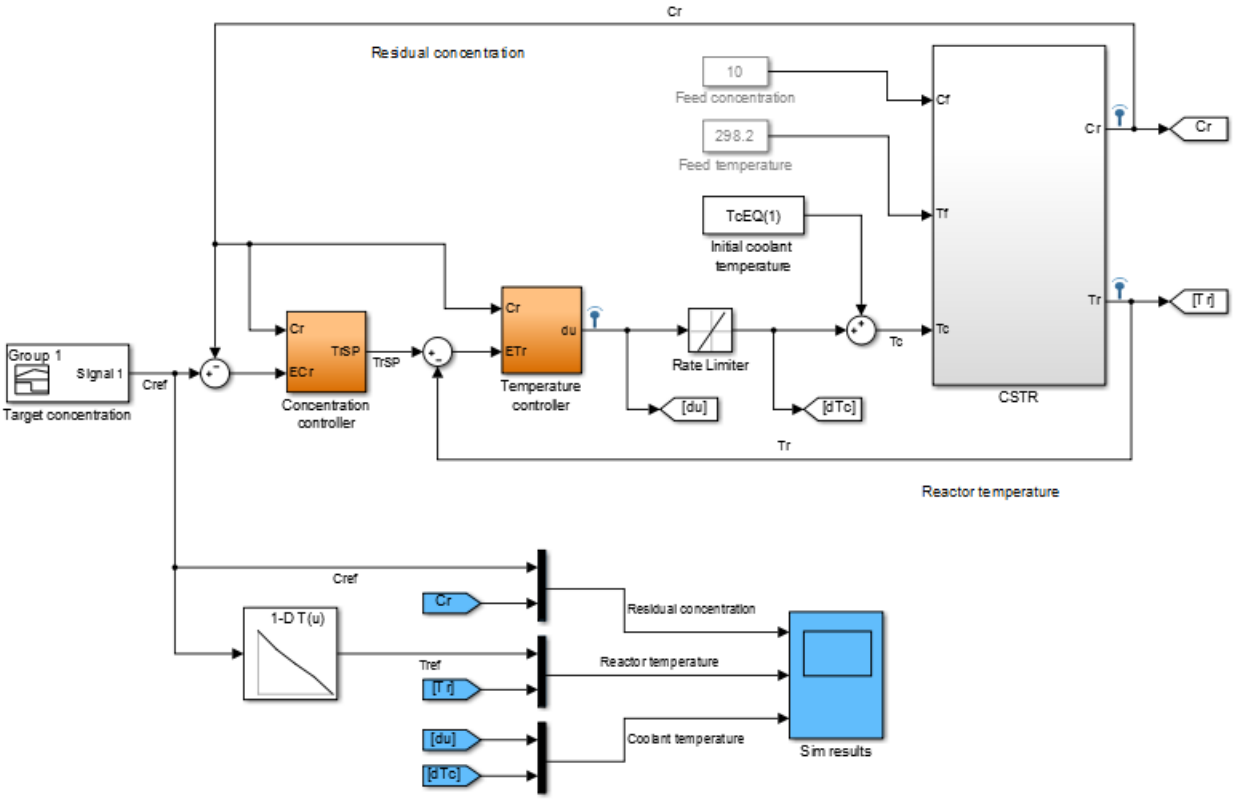
If you have Simulink Control Design, you can use `systune` to tune gain schedules implement as either lookup tables or MATLAB functions. See “Tune Gain Schedules in Simulink” on page 16-12.

Scheduled Gain in Controller

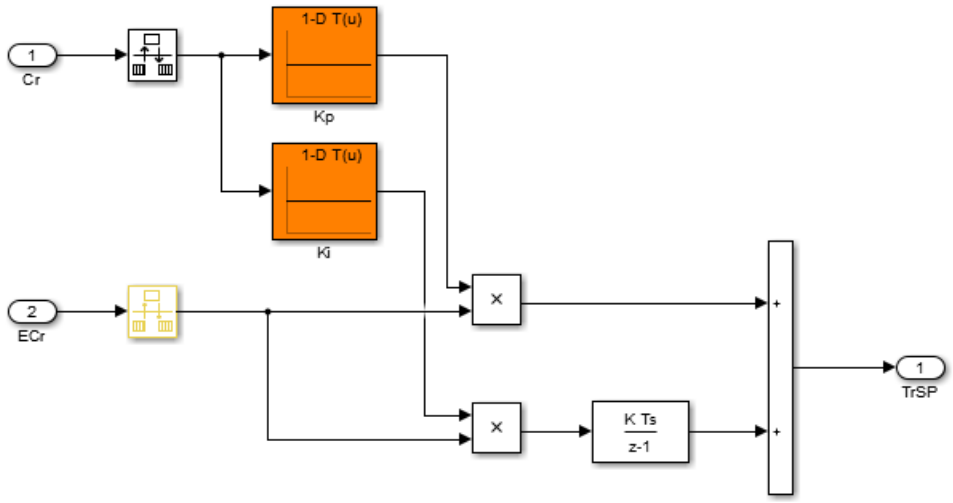
As an example, the model `rct_CSTR` includes a PI controller and a lead compensator in which the controller gains are implemented as lookup tables using 1-D Lookup Table blocks. For more information on this model, see “Gain-Scheduled Control of a Chemical Reactor” on page 16-41.

Copy the example files and open the `rct_CSTR` model.

```
openExample("control/GainScheduledProcessExample",...
    supportingFile="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.

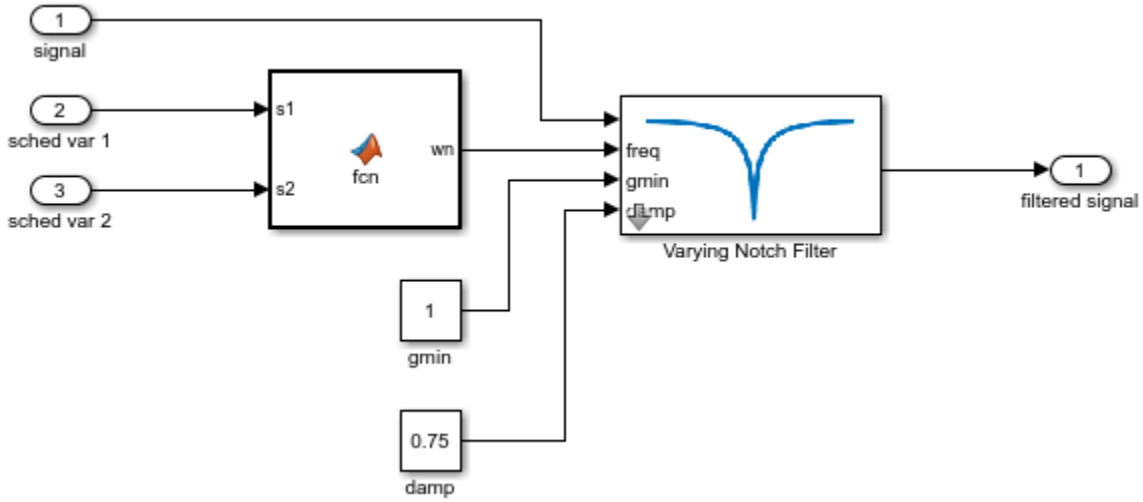
Caution When using the Varying State Space block, avoid scheduling the **C** and **D** matrices based on the system output **y**. If you have such dependence, the resulting state-space equation $y = C(y)x + D(y)u$ creates an algebraic loop, because computing the output value y requires knowing the output value. This algebraic loop is prone to instability and divergence. Instead, try expressing **C** and **D** in terms of the time t, the block input **u**, and the state outputs **x**.

For similar reasons, avoid scheduling **A** and **B** based on the **dx** output. Note that it is safe to for **A** and **B** to depend on **y** when **y** is a fixed combination of states and inputs, (in other words, when $y = Cx + Du$ where C and D are constant matrices).

Similarly, in the Discrete Varying State Space block, use $\mathbf{x_k}$ instead of $\mathbf{y_k}$ when scheduling **C** and **D**, and avoid using $\mathbf{x_{k+1}}$ to schedule **A** and **B**. You can use $\mathbf{y_k}$ to schedule **A** and **B** when $\mathbf{y_k}$ is a fixed combination of states and inputs.

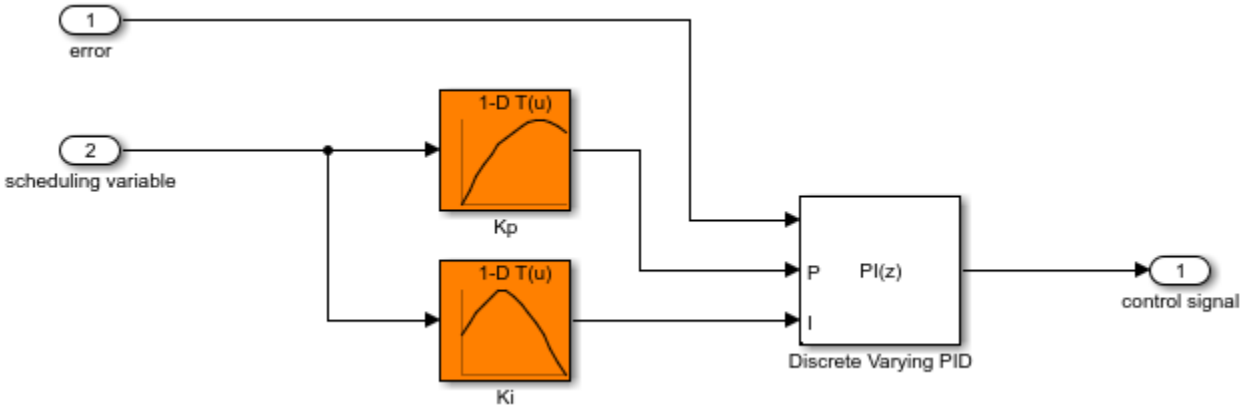
Gain-Scheduled Notch Filter

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:

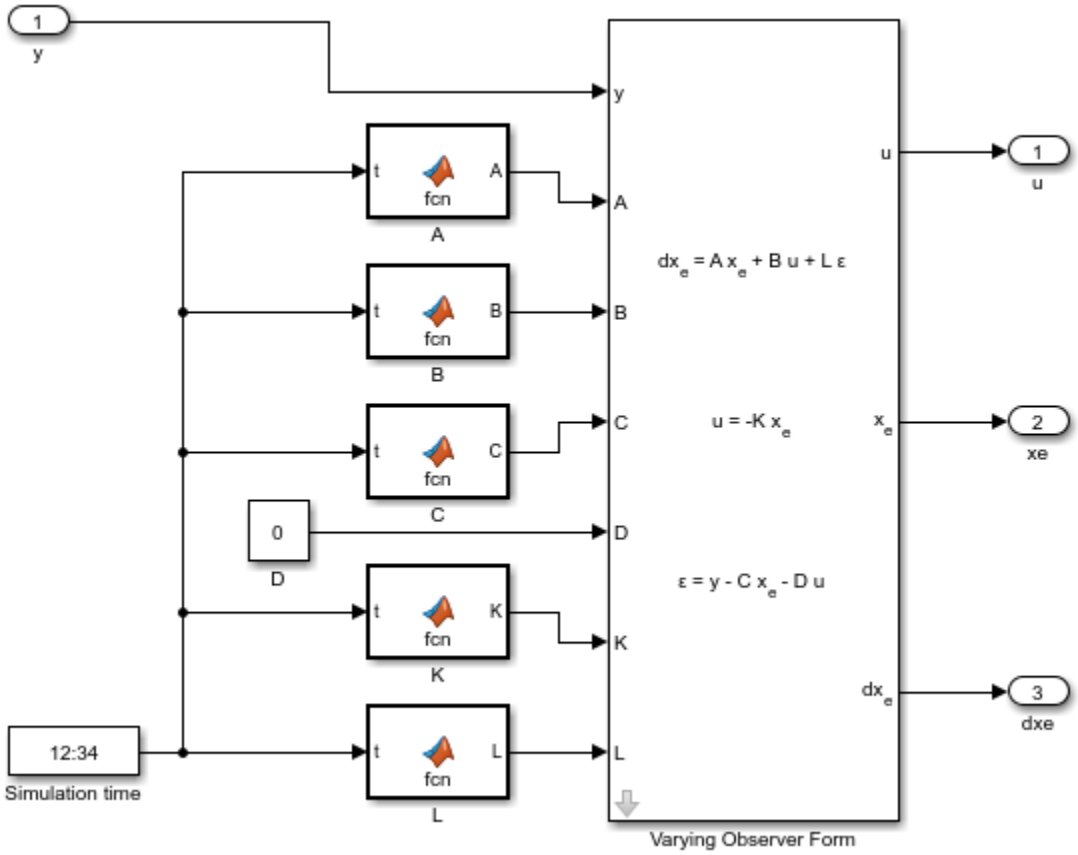
$$dx_e = Ax_e + Bu + L(y - Cx_e - Du)$$
$$u = -Kx_e,$$

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:

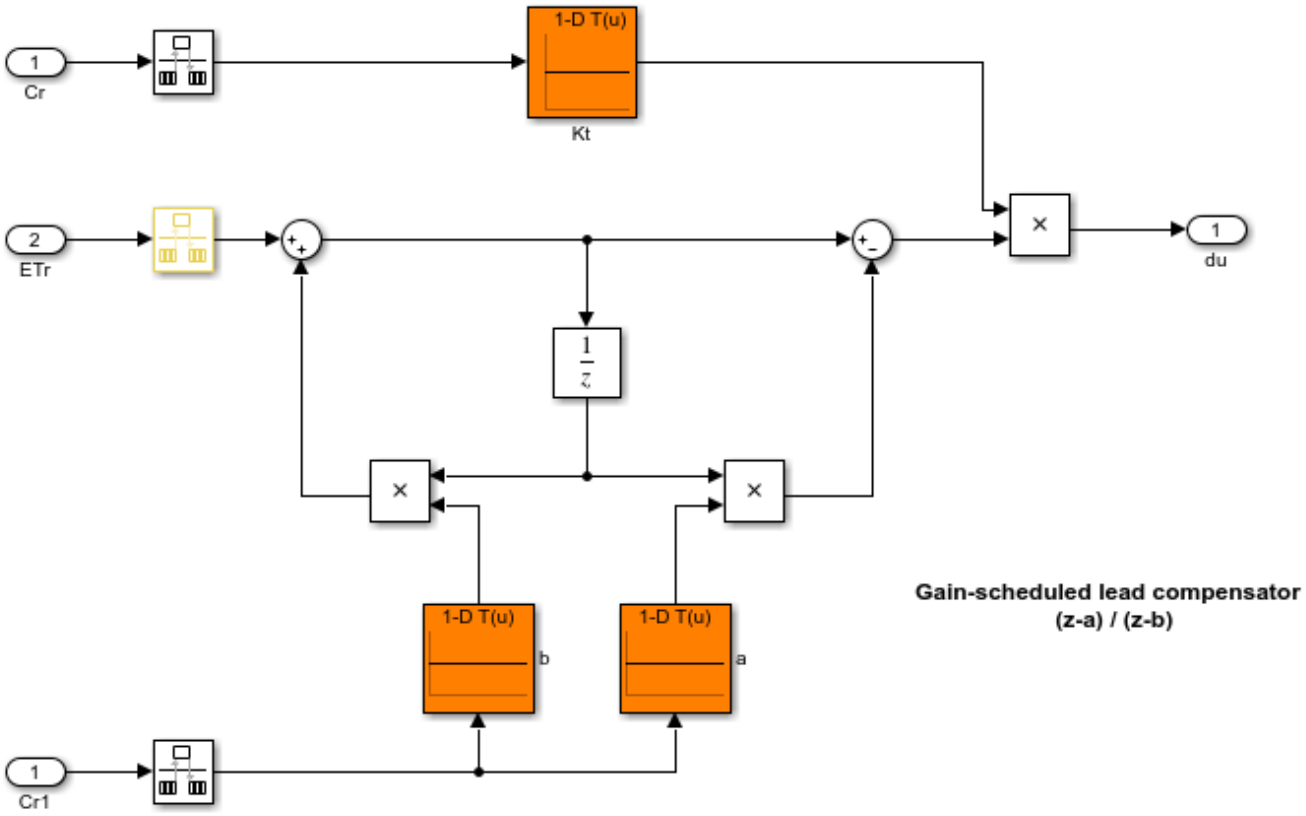


In this implementation, the time-varying matrices are each implemented as a MATLAB Function block in which the associated function takes the simulation time and returns a matrix of appropriate dimensions.

If you have Simulink Control Design, 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 16-12
- “Gain-Scheduled Control of a Chemical Reactor” on page 16-41

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 16-4.)

If you have Simulink Control Design, 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 16-14.
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 16-20.
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 16-24.
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 16-33.

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 16-36.

See Also

More About

- “Model Gain-Scheduled Control Systems in Simulink” on page 16-4
- “Gain-Scheduled Control of a Chemical Reactor” on page 16-41
- “Tuning of Gain-Scheduled Three-Loop Autopilot” on page 16-55

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:

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

$$A(\sigma) = \frac{\partial f}{\partial x}(x(\sigma), u(\sigma)) \quad B(\sigma) = \frac{\partial f}{\partial u}(x(\sigma), u(\sigma))$$

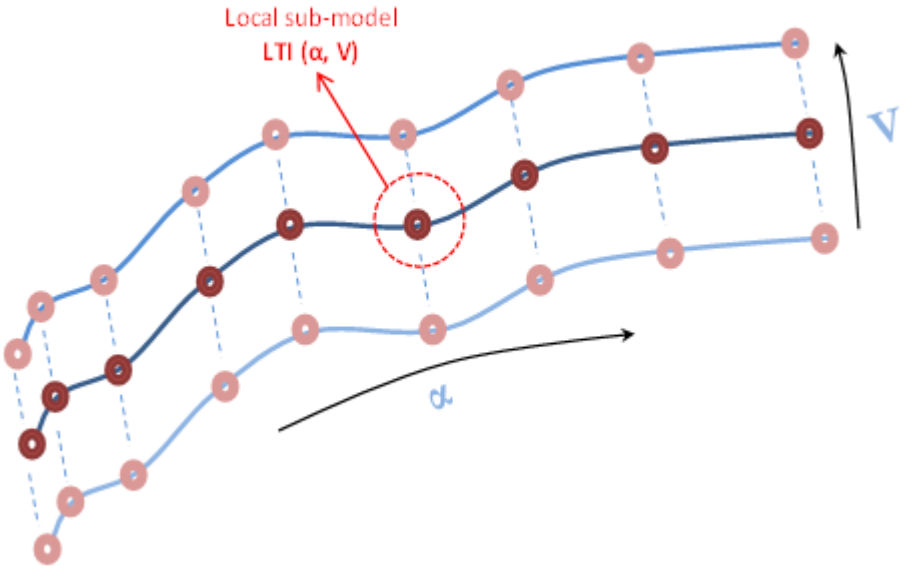
$$C(\sigma) = \frac{\partial g}{\partial x}(x(\sigma), u(\sigma)) \quad D(\sigma) = \frac{\partial g}{\partial u}(x(\sigma), u(\sigma))\,.$$

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 16-15.
- Linearize the Simulink model using parameter variation on page 16-18.
- If the scheduling variable is time, linearize the model at a series of simulation snapshots on page 16-18.
- If you have nonlinear differential equations that describe the plant, linearize them at the design points.

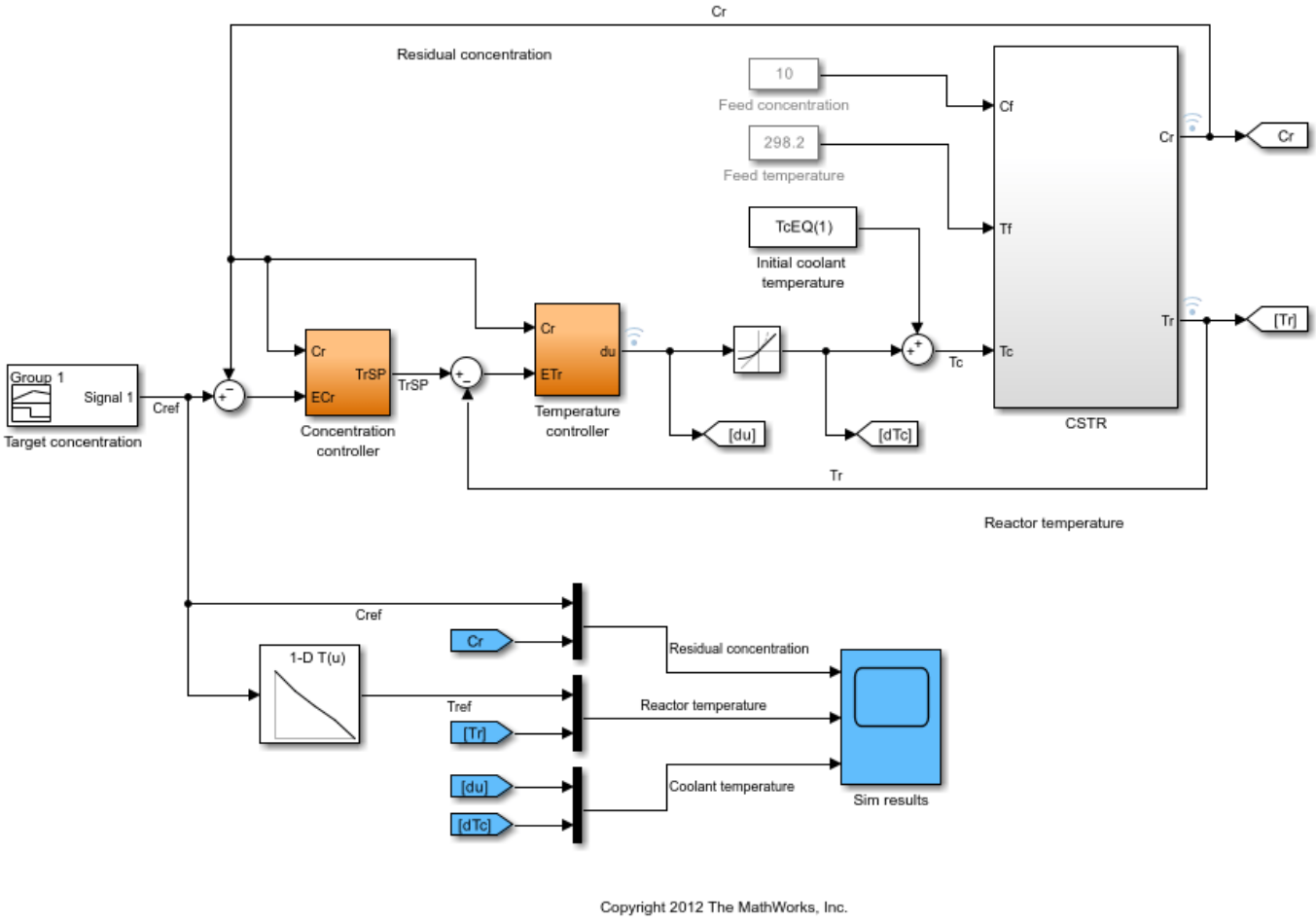
For tuning gain schedules, after you obtain the family of linear models, you must associate it with an `slTuner` interface to build a family of tunable closed-loop models. To do so, use block substitution, as described in “Multiple Design Points in slTuner Interface” on page 16-20.

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.

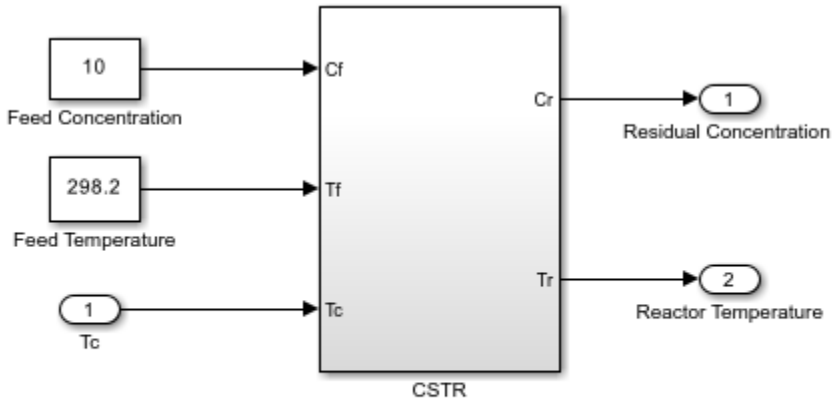
Open the `rct_CSTR` model.

```
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” (Simulink Control Design).

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” on page 16-20.

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” on page 16-68.

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 16-33.

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” on page 9-7.

See Also

`slTuner` | `findop` | `voidModel`

Related Examples

- “Parameterize Gain Schedules” on page 16-24
- “Tune Gain Schedules in Simulink” on page 16-12
- “Multiple Design Points in slTuner Interface” on page 16-20

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 16-14, 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:

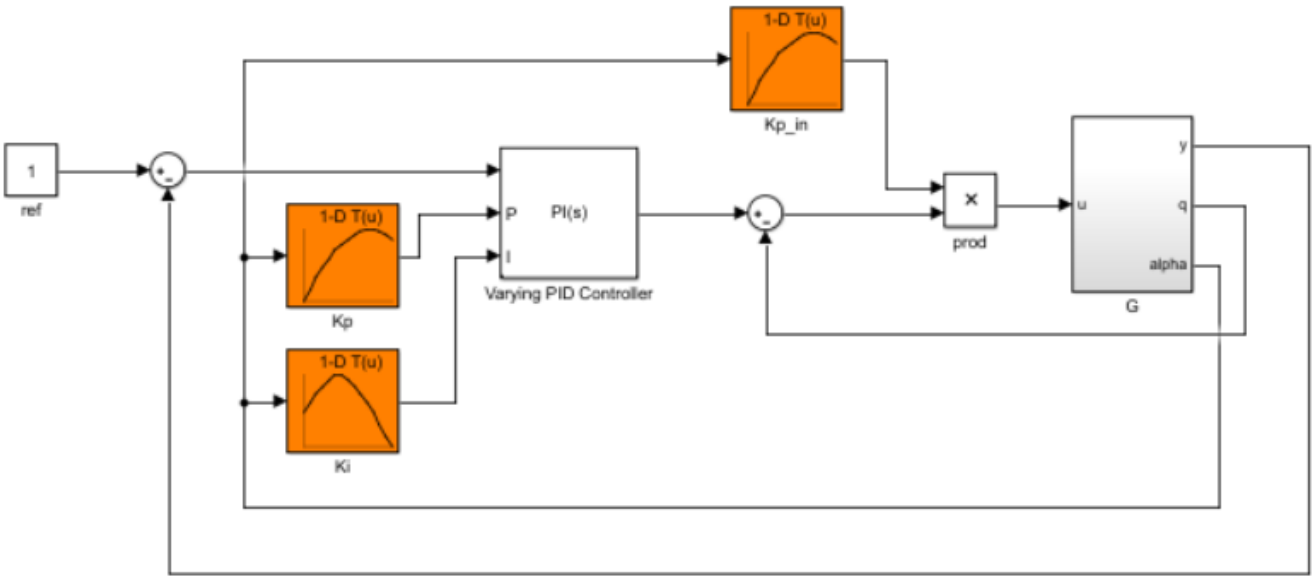
```
BlockSubs = struct('Name','mdl/G','Value',Garr);
st0 = slTuner('mdl',{'Kp','Ki'},BlockSubs);
```

`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 16-24, 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 16-75 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

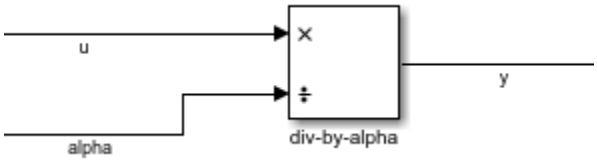
Next, suppose that you have already tuned the inner-loop gain schedule, and have obtained an array `Kp_in_tuned`, of values of `Kp_in` that correspond to each design point (each value of `alpha` at

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`.

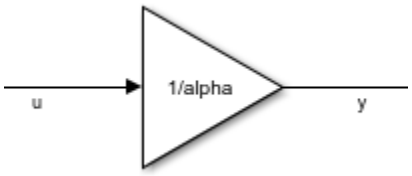
```
prodsub = [0 ss(Kp_in_tuned)];
BlockSubs1 = struct('Name','mdl/prod','Value',prodsub);
BlockSubs2 = struct('Name','mdl/G','Value',Garr);

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 16-81.

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`:



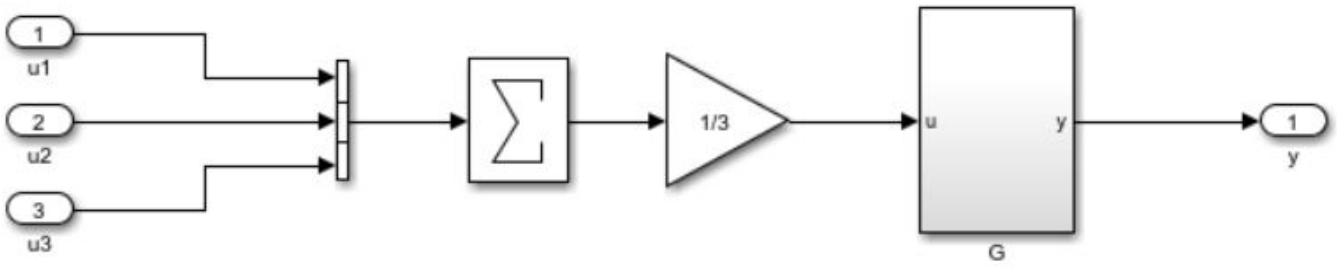
Therefore, you can use the following block substitution in your `slTuner` interface, where `alphagrid` is an array of `alpha` values at your design points.

```
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 16-20. 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 16-12
- “Plant Models for Gain-Scheduled Controller Tuning” on page 16-14
- “Parameterize Gain Schedules” on page 16-24

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:

$$C(s,\sigma) = K_p(\sigma) + \frac{K_i(\sigma)}{s}.$$

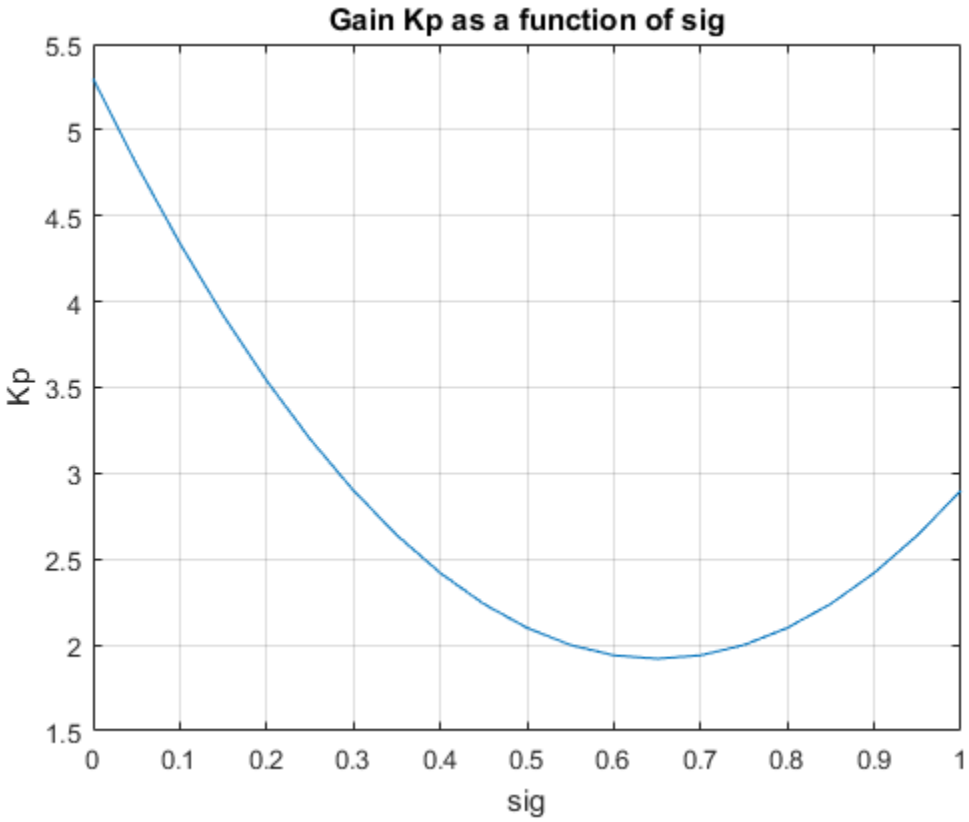
Tuning this controller requires determining the functional forms $K_p(\sigma)$ and $K_i(\sigma)$ that yield the best system performance over the operating range of σ values. However, tuning arbitrary functions is difficult. Therefore, it is necessary either to consider the function values at only a finite 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:

$$K_p(\sigma) = k_0 + k_1\sigma + k_2\sigma^2.$$

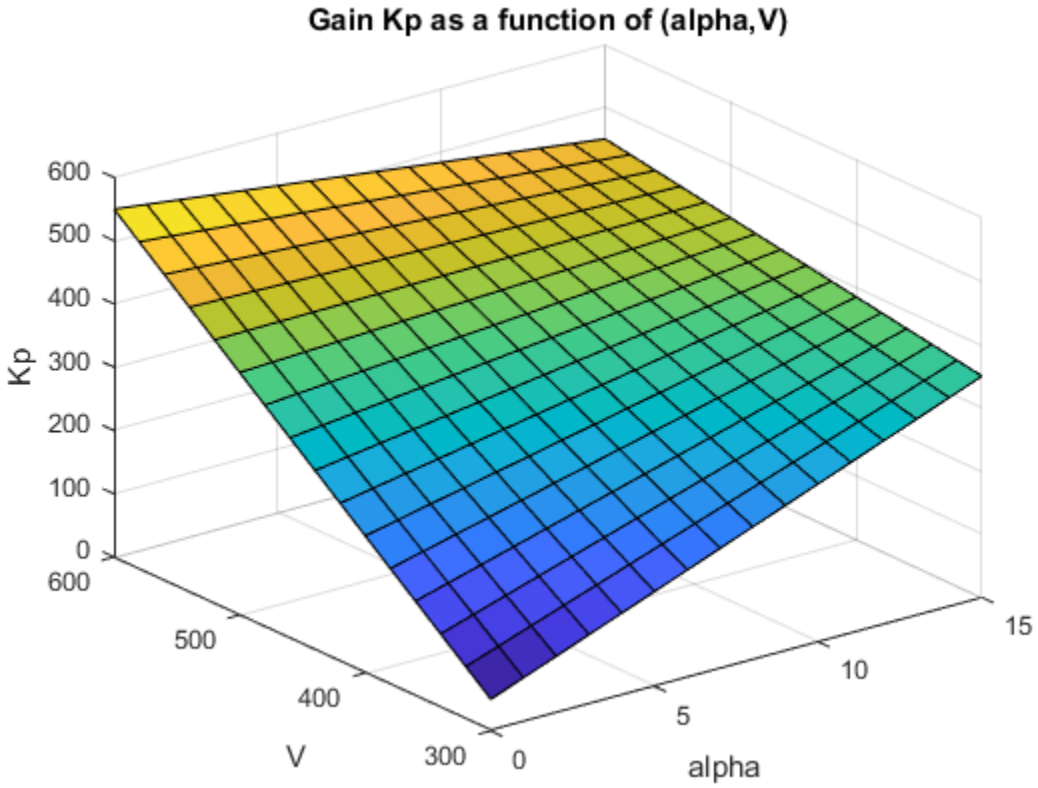
After tuning, this parametric gain might have a profile such as the following (the specific shape of the curve depends on the tuned coefficient values and range of σ):



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 16-14.)

```
[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:

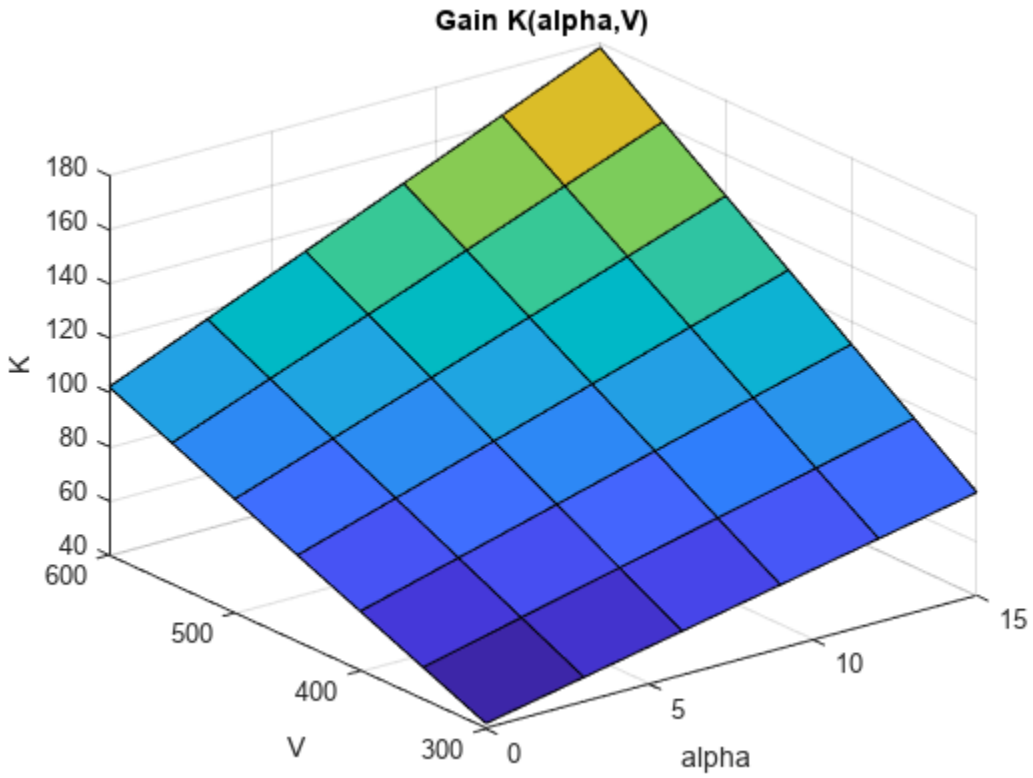
$$F_1(x,y) = x$$
$$F_2(x,y) = y$$
$$F_3(x,y) = xy\,.$$

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.

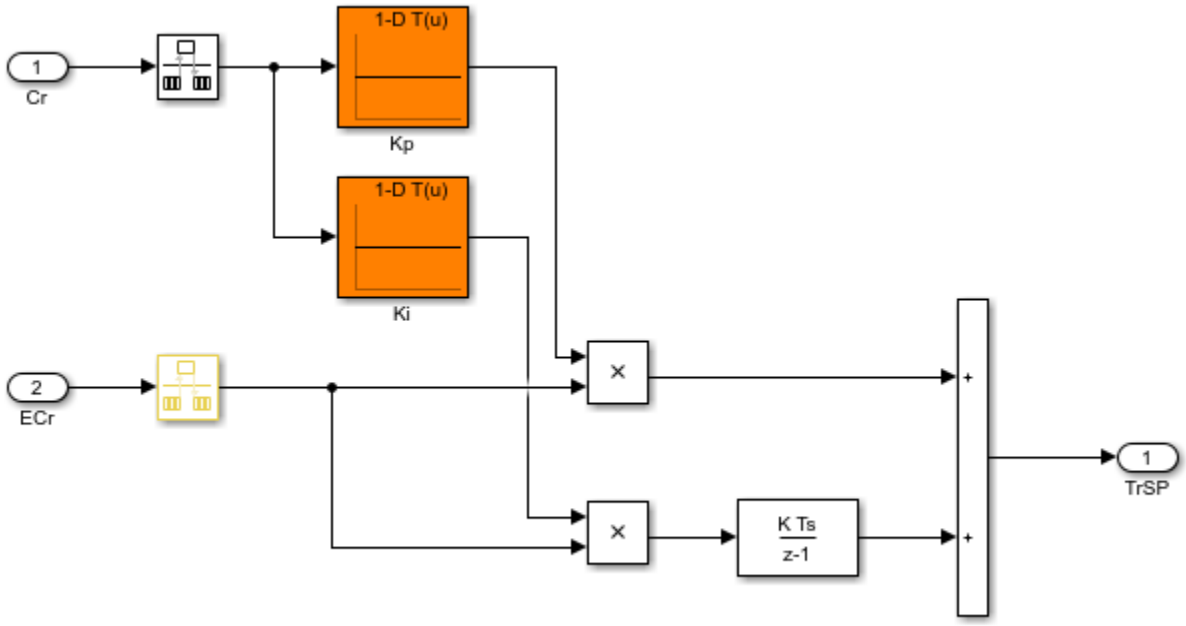
```
Ktuned = setData(K,[100,28,40,10]);
viewSurf(Ktuned)
```



`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 16-4. 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 16-14). 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” on page 16-41.

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” on page 16-41.

See Also

`tunableSurface`

Related Examples

- “Model Gain-Scheduled Control Systems in Simulink” on page 16-4
- “Multiple Design Points in slTuner Interface” on page 16-20

- “Tune Gain Schedules in Simulink” on page 16-12

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 16-36.

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 16-36
- “Tune Gain Schedules in Simulink” on page 16-12

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

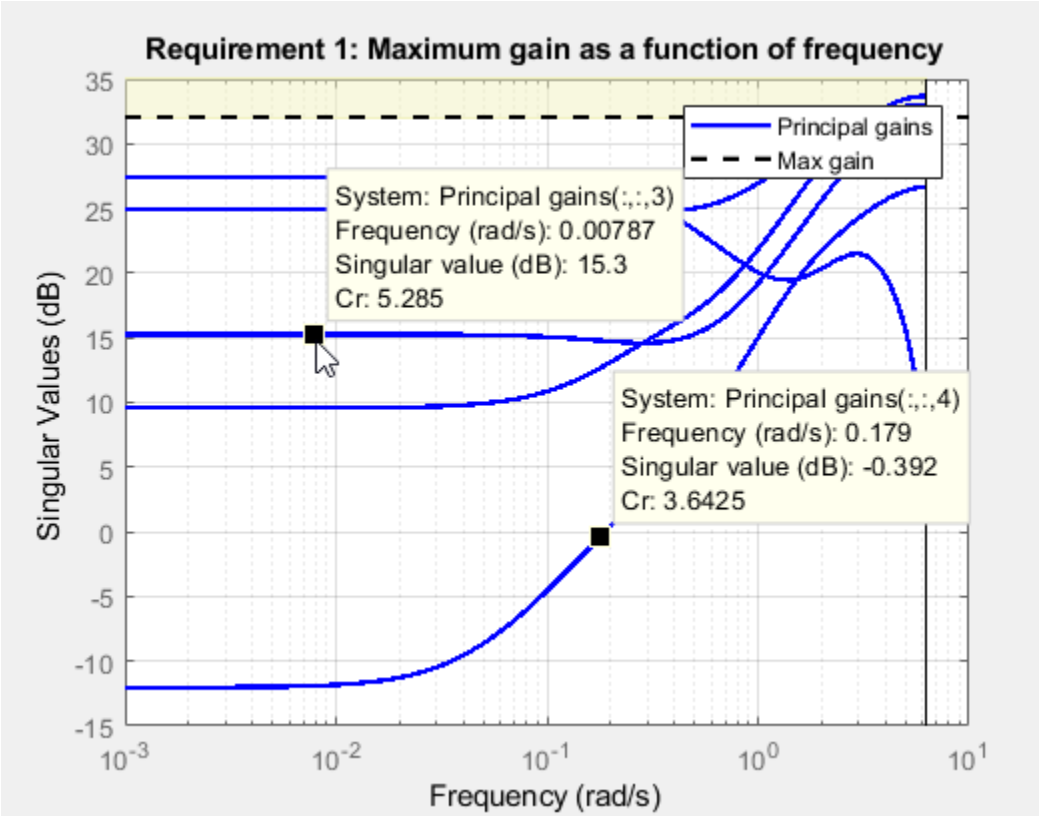
Use tuning-goal plots to visualize your design requirements against the linear response of the tuned control system. Tuning-goal plots show graphically where and by how much tuning goals are satisfied or violated. This visualization lets you examine how close your control system is to ideal performance. 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 14-141. 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” on page 16-41. 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`.

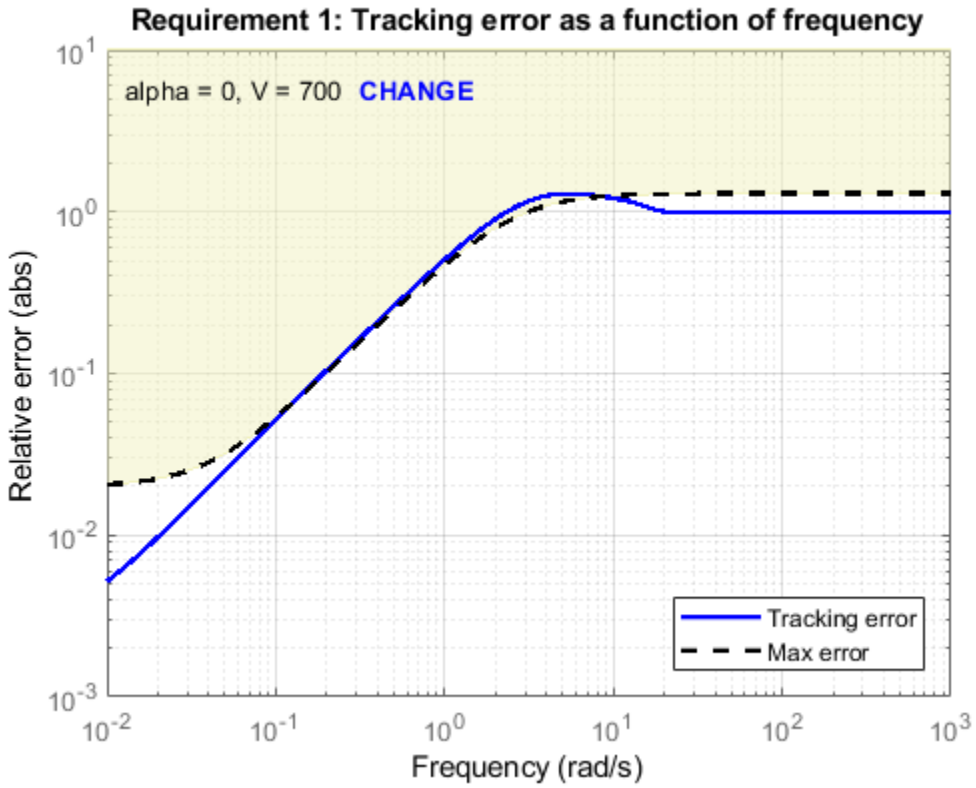
```
viewGoal(R3,ST)
```



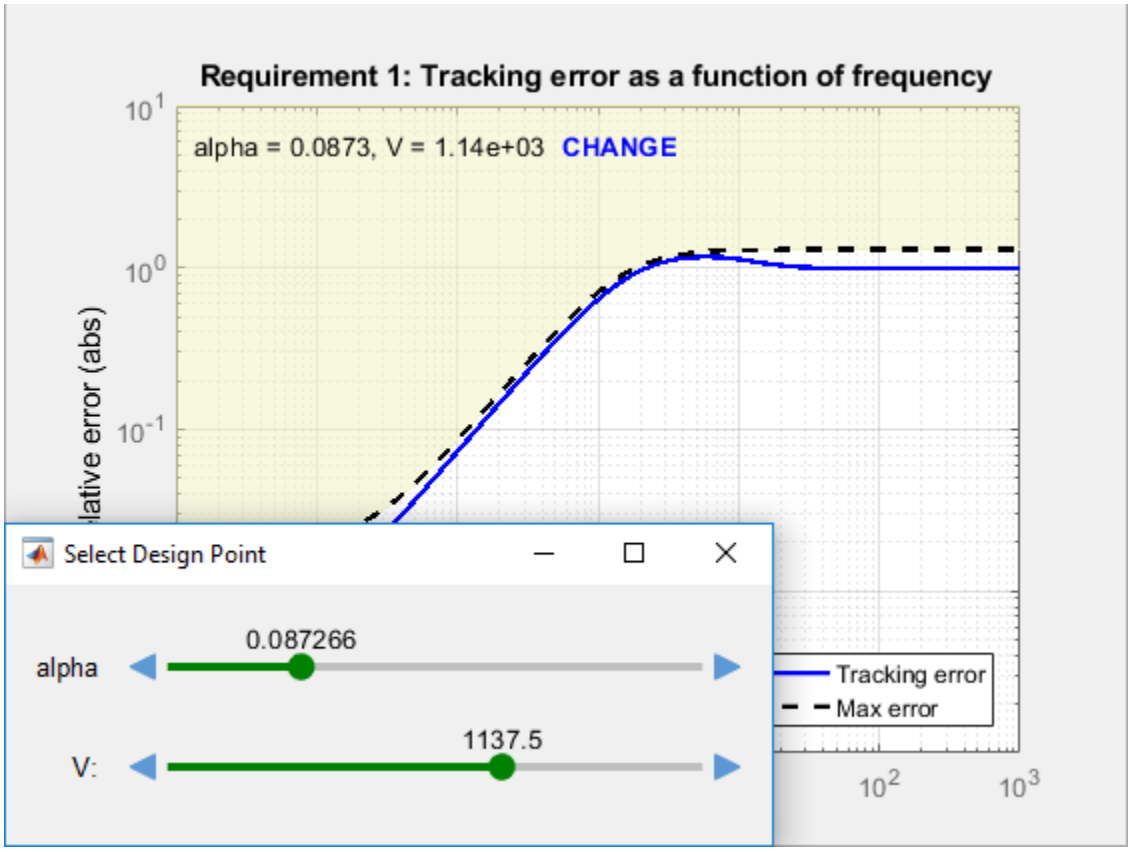
Varying Tuning Goals

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 14-168.

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” on page 16-55
- “Gain-Scheduled Control of a Chemical Reactor” on page 16-41
- “Validate Tuned Control System” on page 14-168

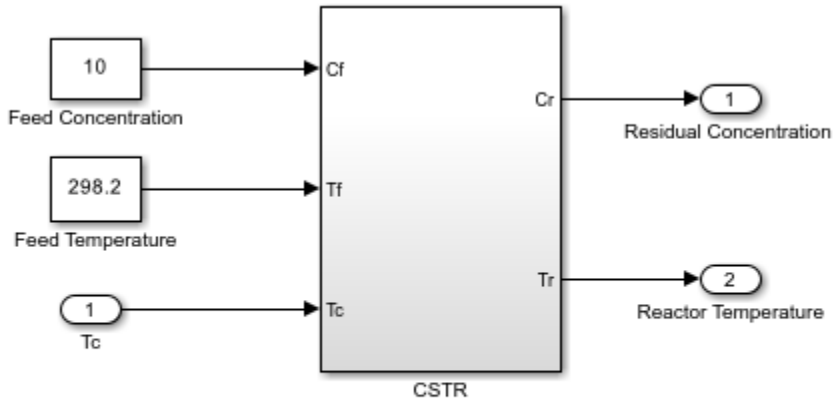
Gain-Scheduled Control of a Chemical Reactor

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')
```



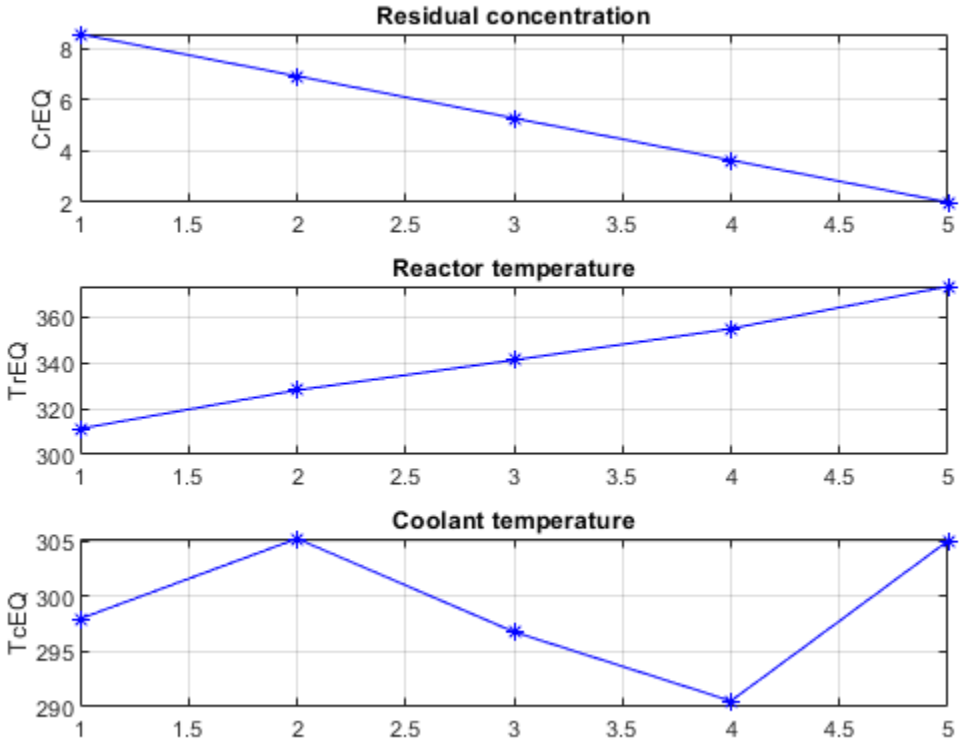
We want to transition from a residual concentration of 8.57 kmol/m^3 initially down to 2 kmol/m^3. To understand how the process dynamics evolve with the residual concentration `Cr`, find the equilibrium conditions for five values of `Cr` between 8.57 and 2 and linearize the process dynamics around each 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')
```



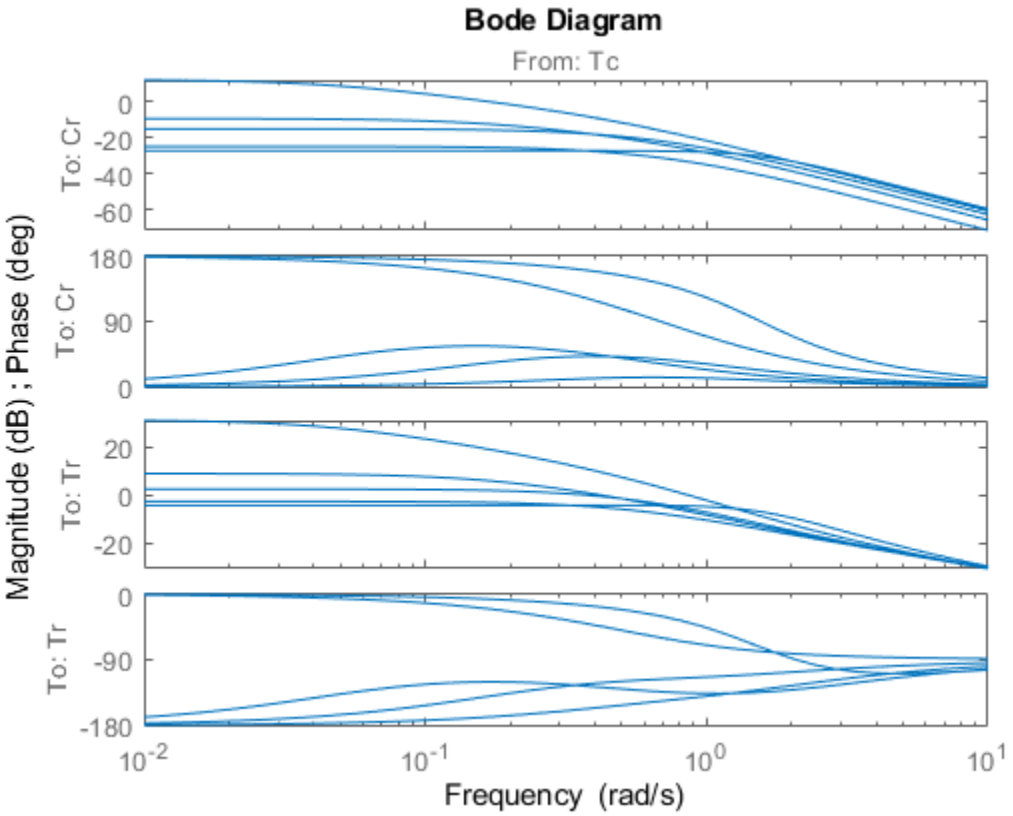
An open-loop control strategy consists of following the coolant temperature profile above to smoothly transition between the `Cr`=8.57 and `Cr`=2 equilibria. However, this strategy is doomed by the fact that the reaction is unstable in the mid range and must be properly cooled to avoid thermal runaway. 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.

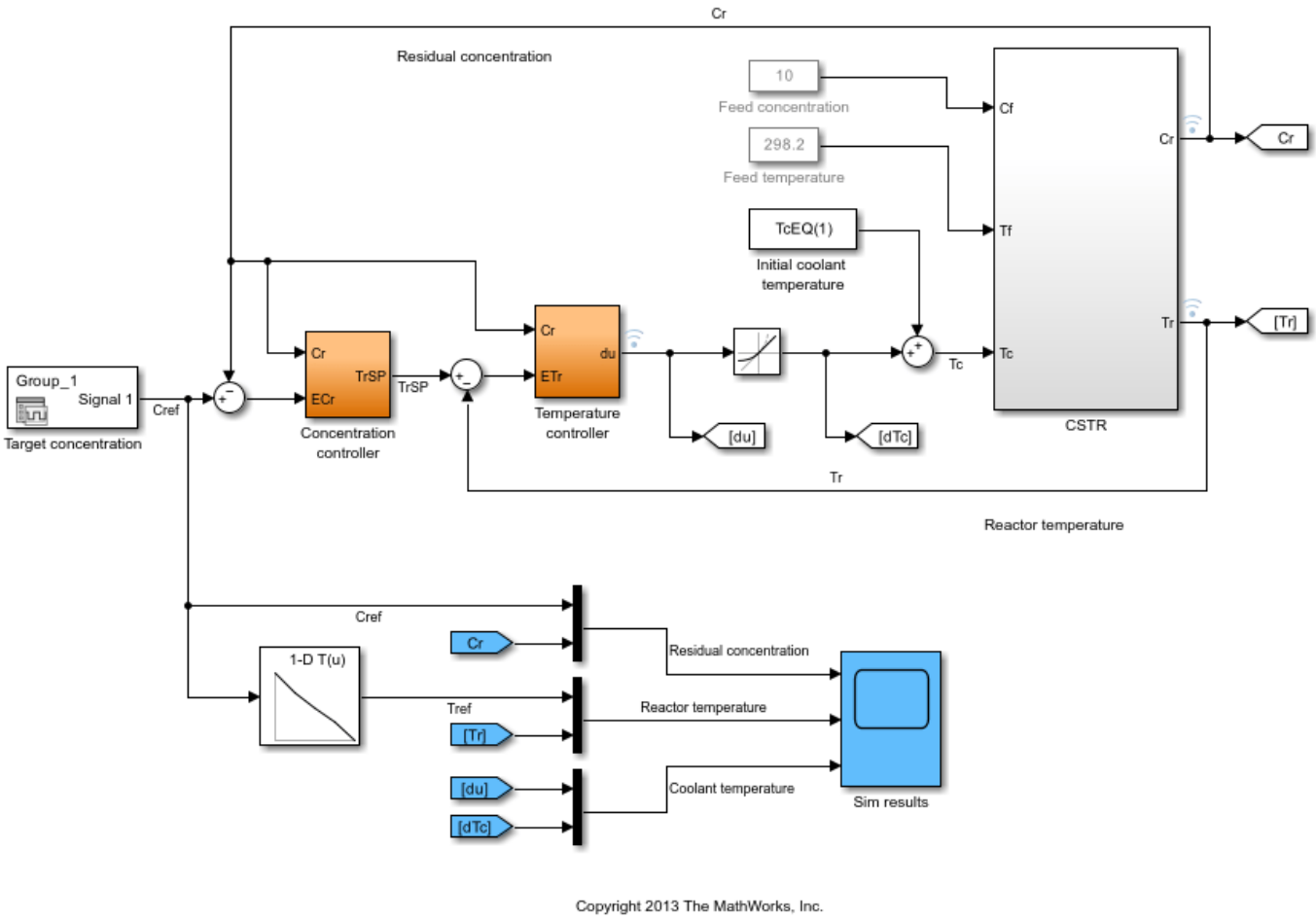
```
clf, bode(G(:,'Tc'),{0.01,10})
```



Feedback Control Strategy

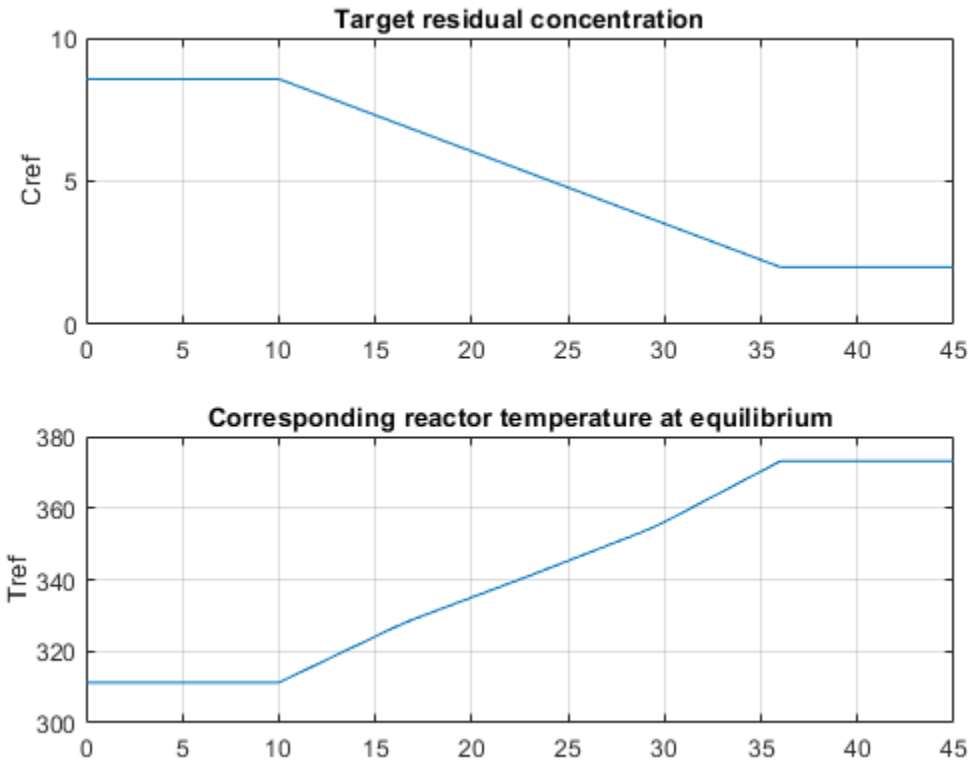
To prevent thermal runaway while ramping down the residual concentration, use feedback control to adjust the coolant temperature `Tc` based on measurements of the residual concentration `Cr` and reactor temperature `Tr`. For this application, we use a cascade control architecture where the inner loop 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.

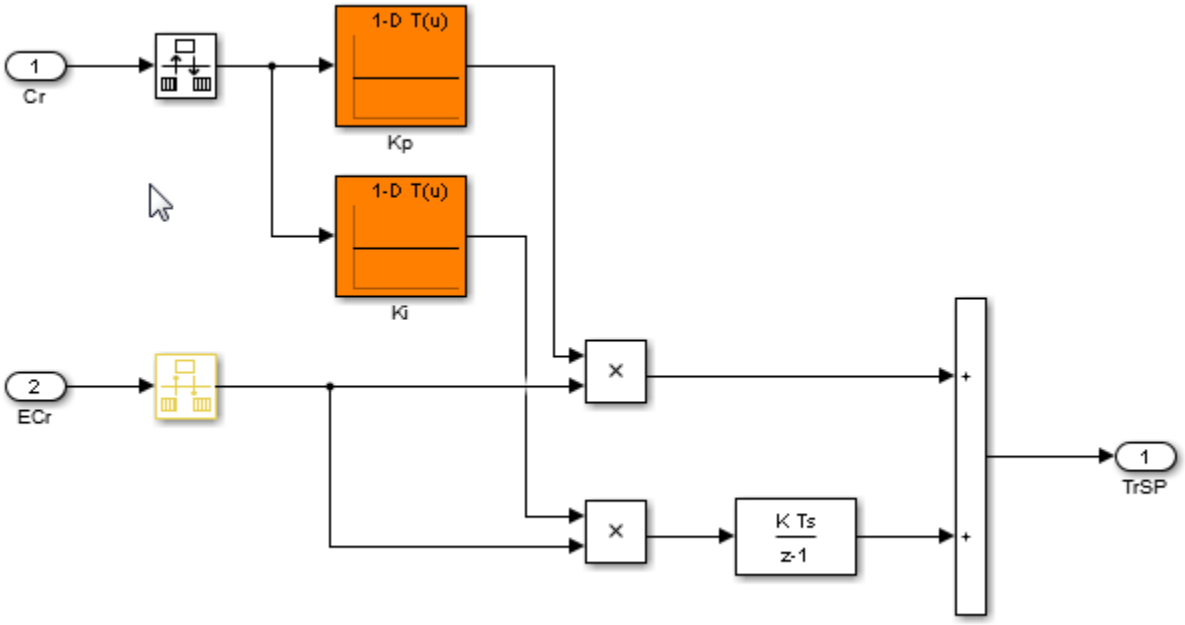


Figure 1: Gain-scheduled PI controller for concentration loop.

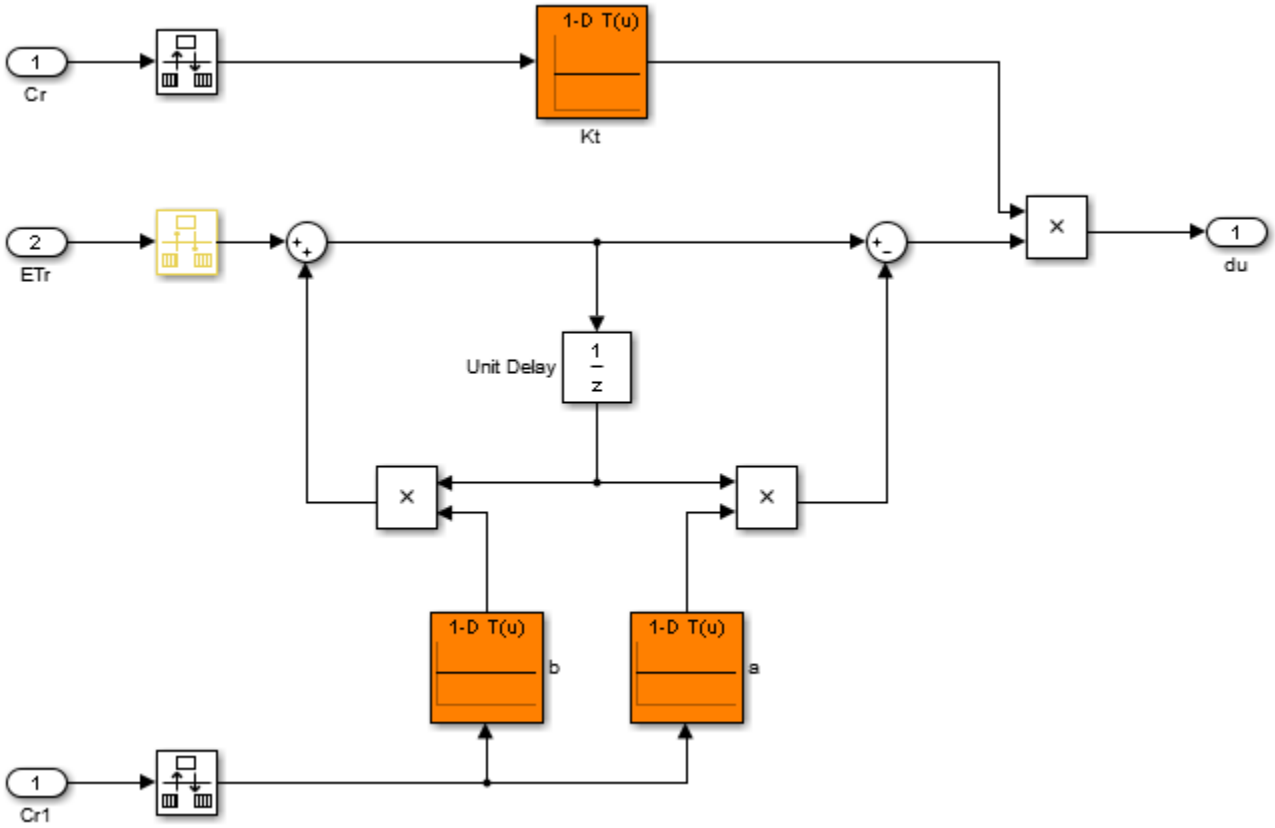


Figure 2: Gain-scheduled lead compensator for temperature loop.

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);
```

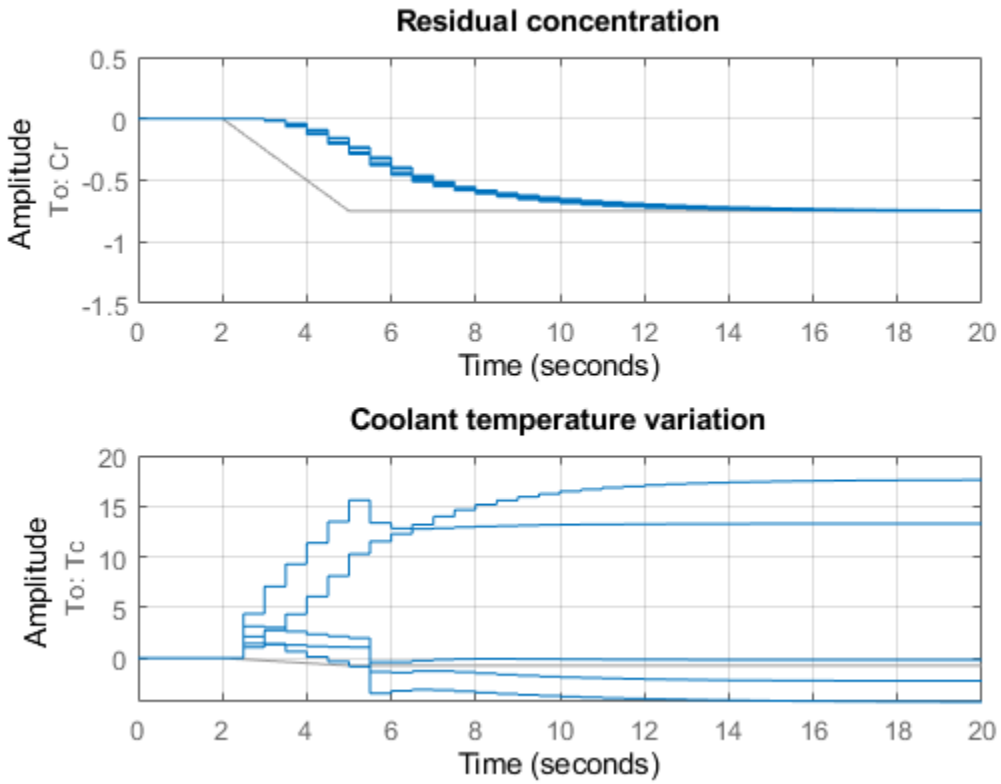
You can now use `systune` to tune the controller coefficients against the requirements `R1-R4`. Make the stability margin requirement a hard constraints and optimize the remaining requirements.

```
ST = systune(ST0,[R1 R2 R3],R4);
```

```
Final: Soft = 1.21, Hard = 0.9991, Iterations = 203
```

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')
```

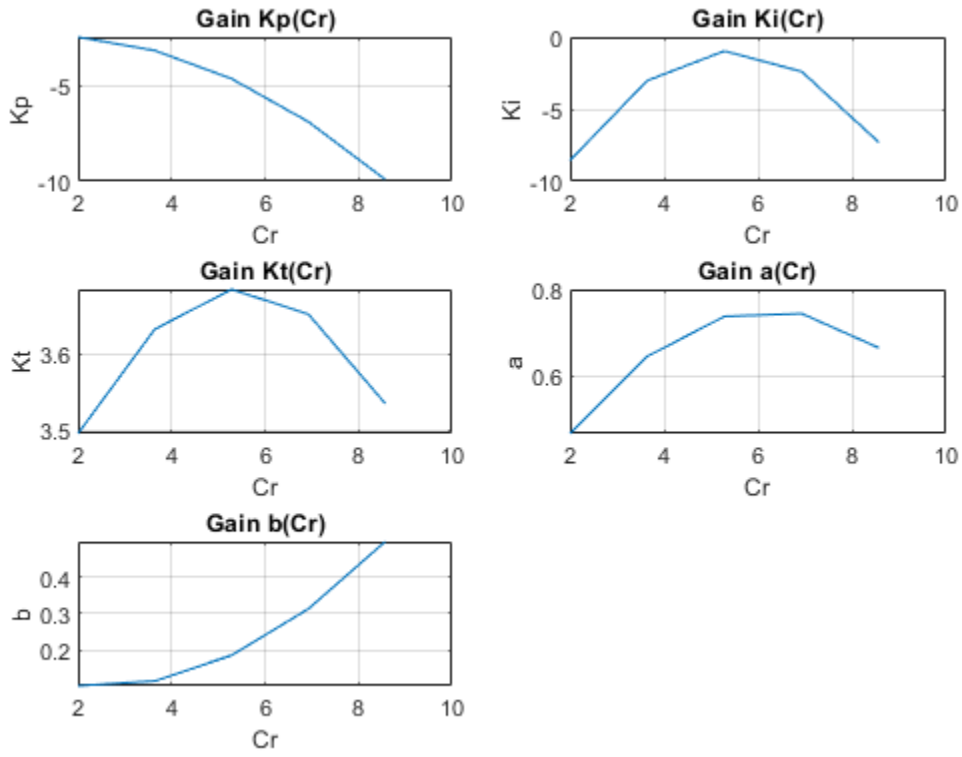


Note that rate of change of the coolant temperature remains within the physical limits (10 degrees per minute or 5 degrees per sample period).

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.

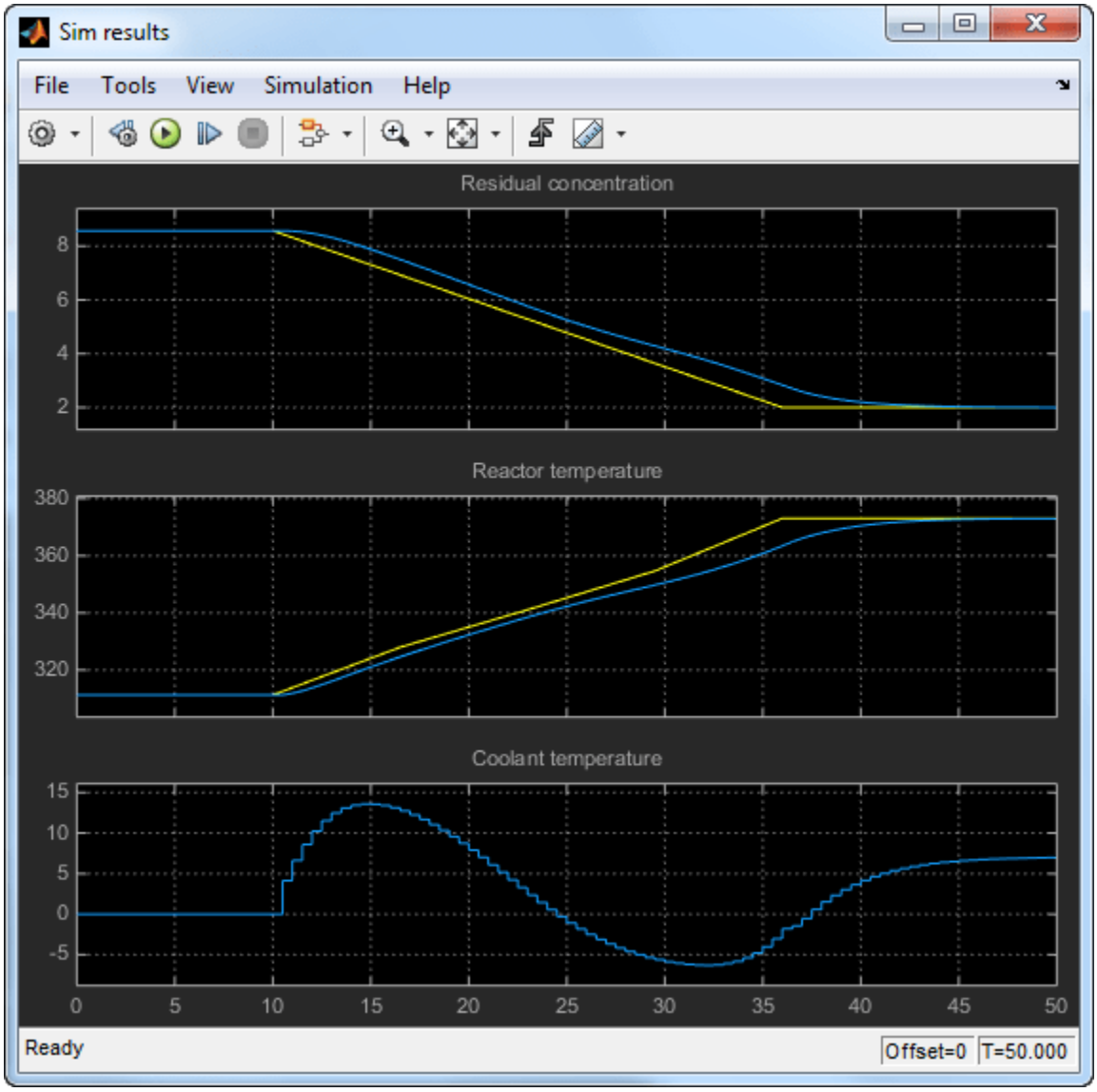


Figure 3: Transition with gain-scheduled cascade controller.

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'});
```

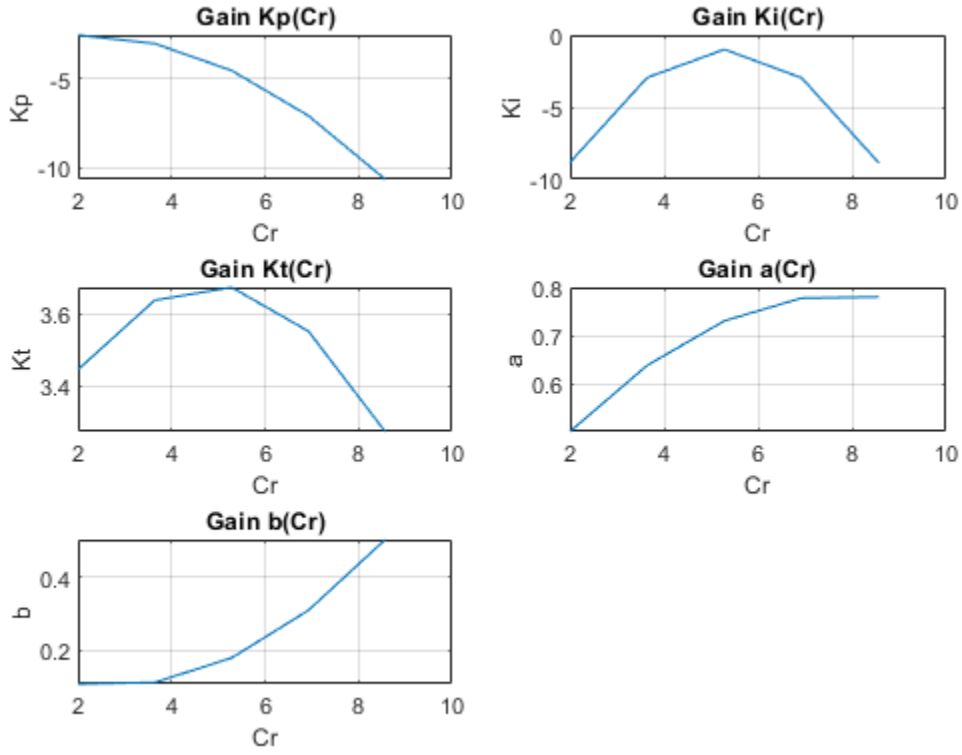
Finally, use `systune` to tune the gain schedule coefficients.

```
T = systune(T0,[R1 R2 R3],R4);
```

```
Final: Soft = 1.21, Hard = 0.99892, Iterations = 238
```

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 16-4
- “Tuning of Gain-Scheduled Three-Loop Autopilot” on page 16-55

More About

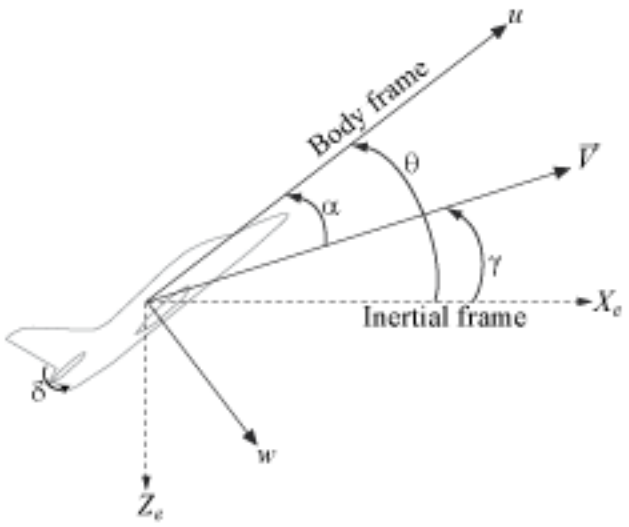
- “Parameterize Gain Schedules” on page 16-24

Tuning of Gain-Scheduled Three-Loop Autopilot

This example uses `systune` to generate smooth gain schedules for a three-loop autopilot.

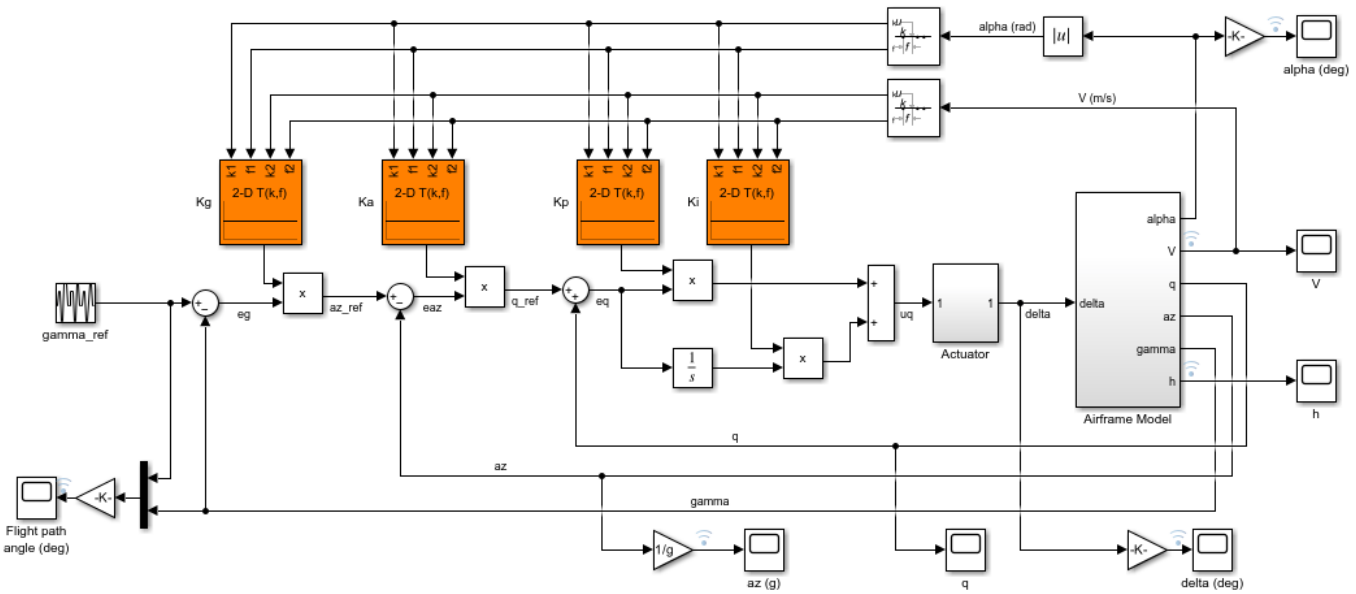
Airframe Model and Three-Loop Autopilot

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`.

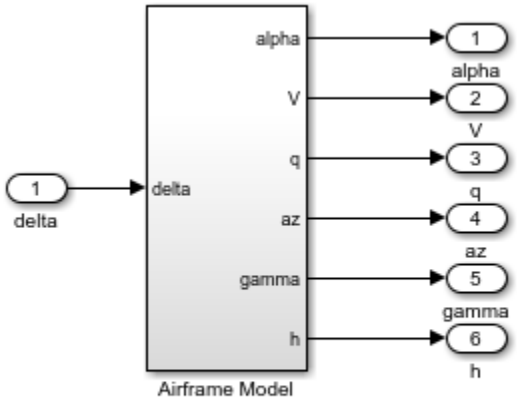
Trimming and Linearization

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” (Simulink Control Design). 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.

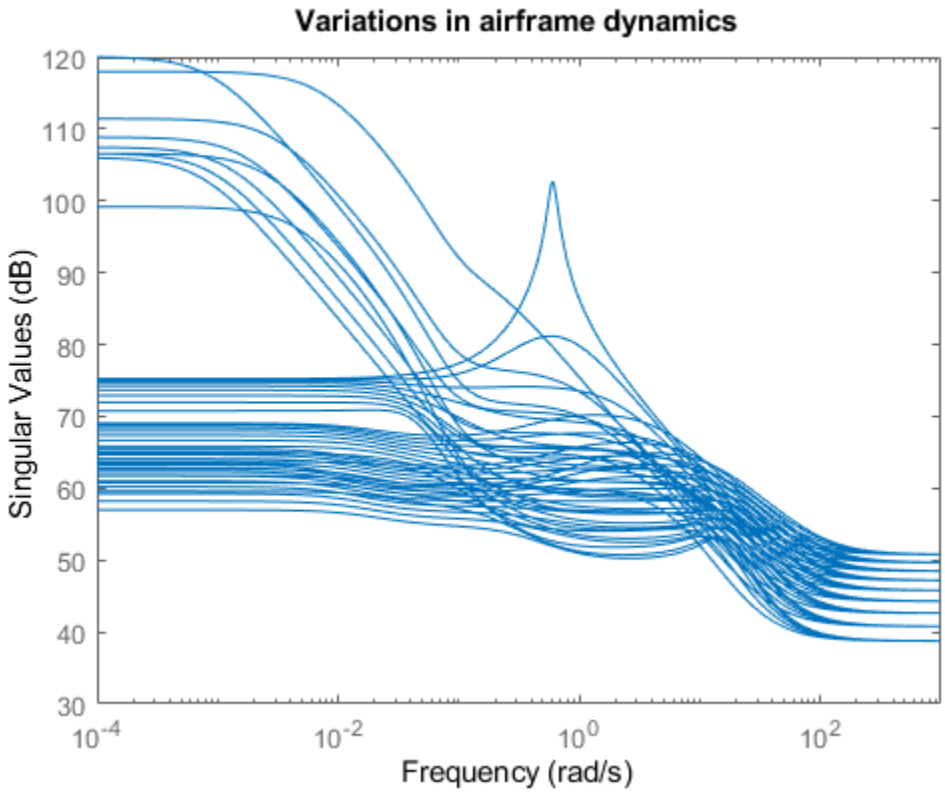
```
Options = findopOptions('DisplayReport','off');
op = findop(mdl,opspec,Options);
```

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:

$$K(\alpha,V) = K_0 + K_1\alpha + K_2V + K_3\alpha V.$$

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.

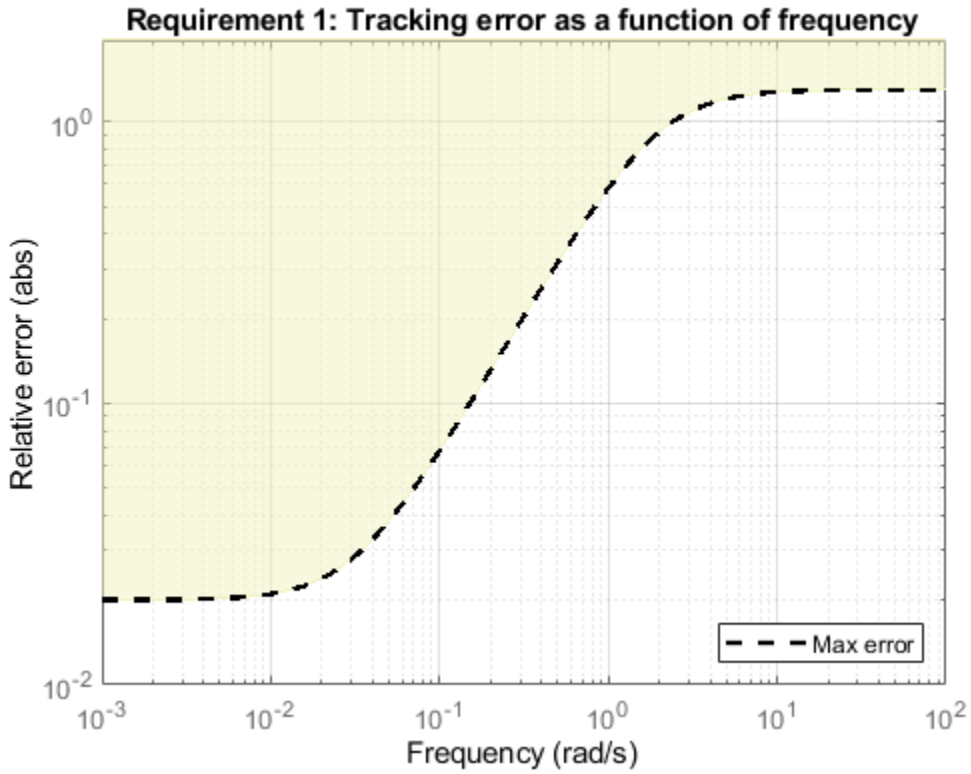
```
BlockSubs = struct('Name','rct_airframeGS/Airframe Model','Value',G);
ST0 = slTuner('rct_airframeGS',{'Kp','Ki','Ka','Kg'},BlockSubs);

% 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:

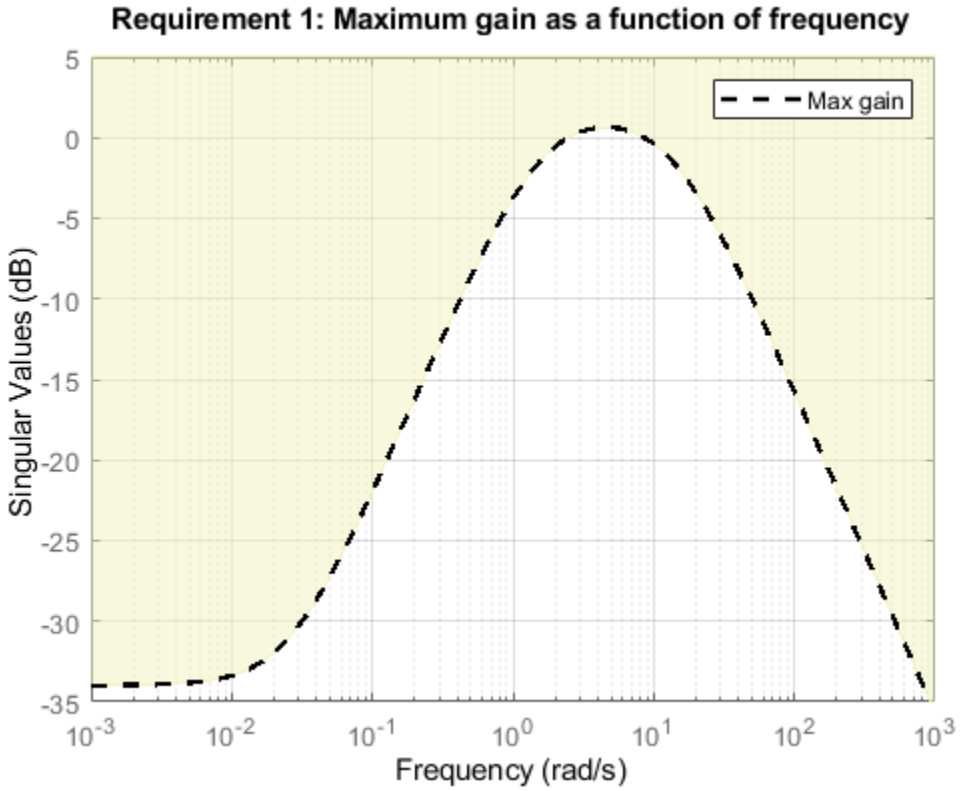
- γ loop: Track the setpoint with a 1 second response time, less than 2% steady-state error, and less than 30% peak error.

```
Req1 = TuningGoal.Tracking('gamma_ref','gamma',1,0.02,1.3);
viewGoal(Req1)
```



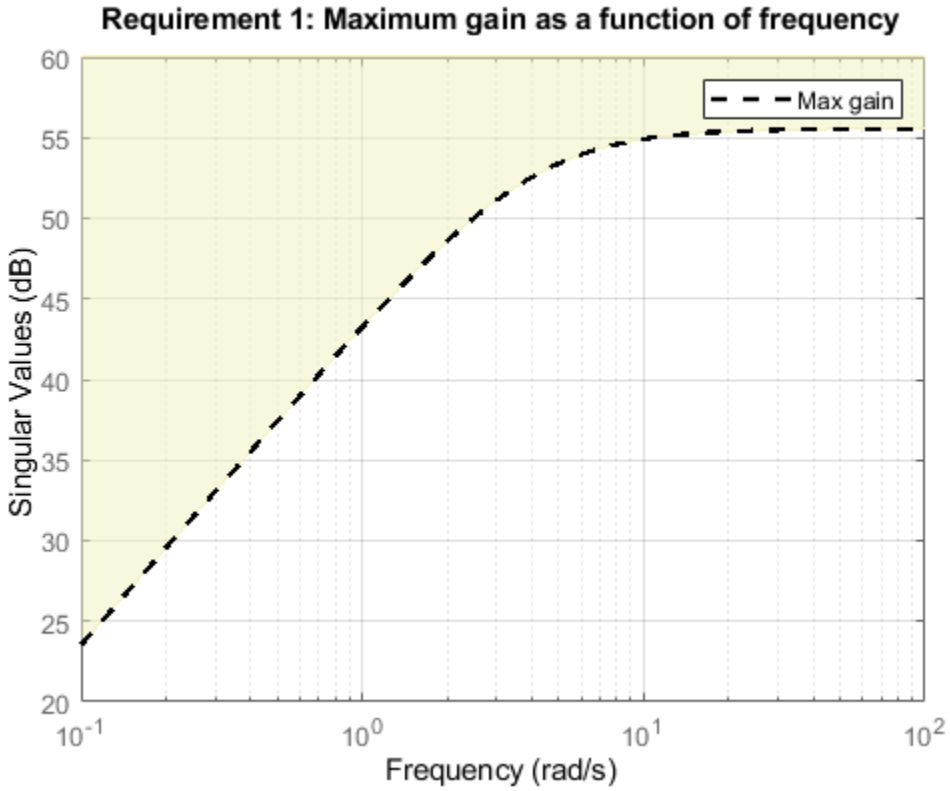
- a_z loop: Ensure good disturbance rejection at low frequency (to track acceleration demands) and past 10 rad/s (to be insensitive to measurement noise). The disturbance is injected at the `az_ref` location.

```
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.

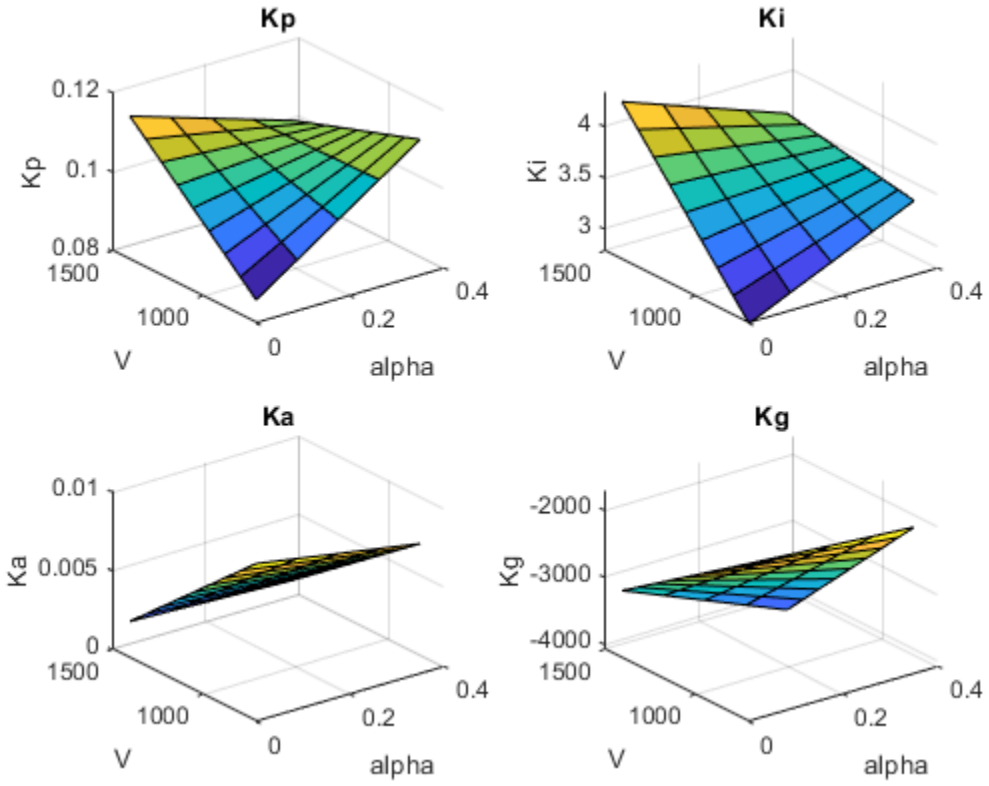
```
ST = systune(ST0,[Req1 Req2 Req3 Req4]);
```

```
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)
```

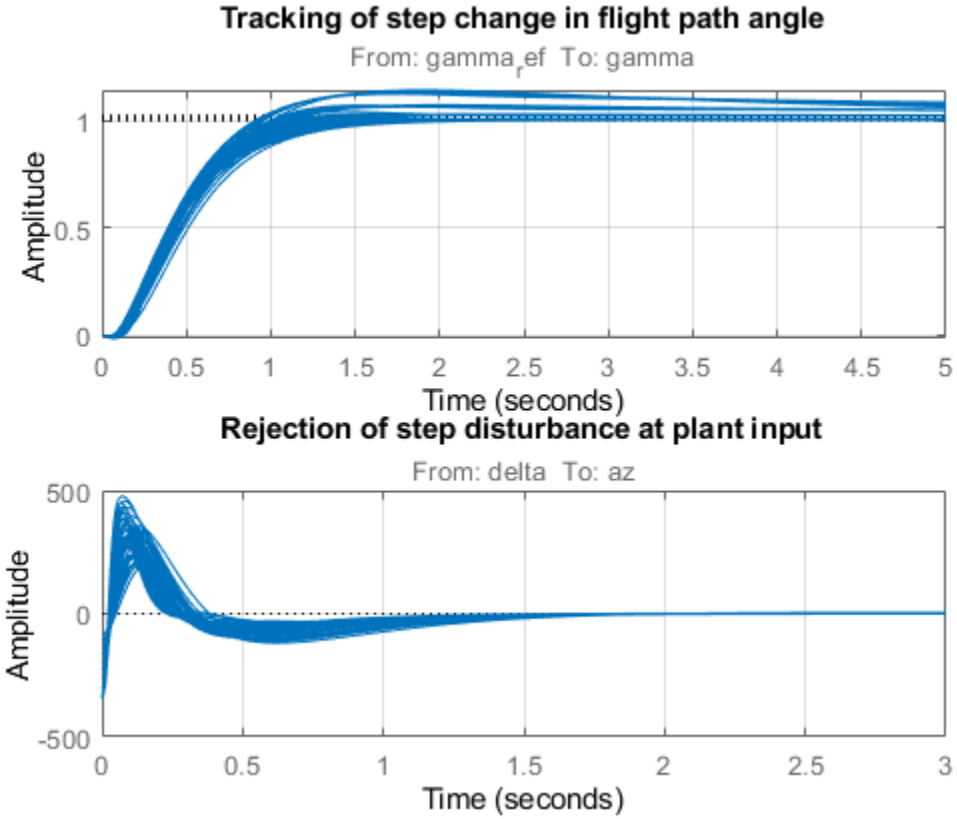
```
title('Ka')
subplot(2,2,4)
viewSurf(TGS.Kg)
title('Kg')
```



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')
```



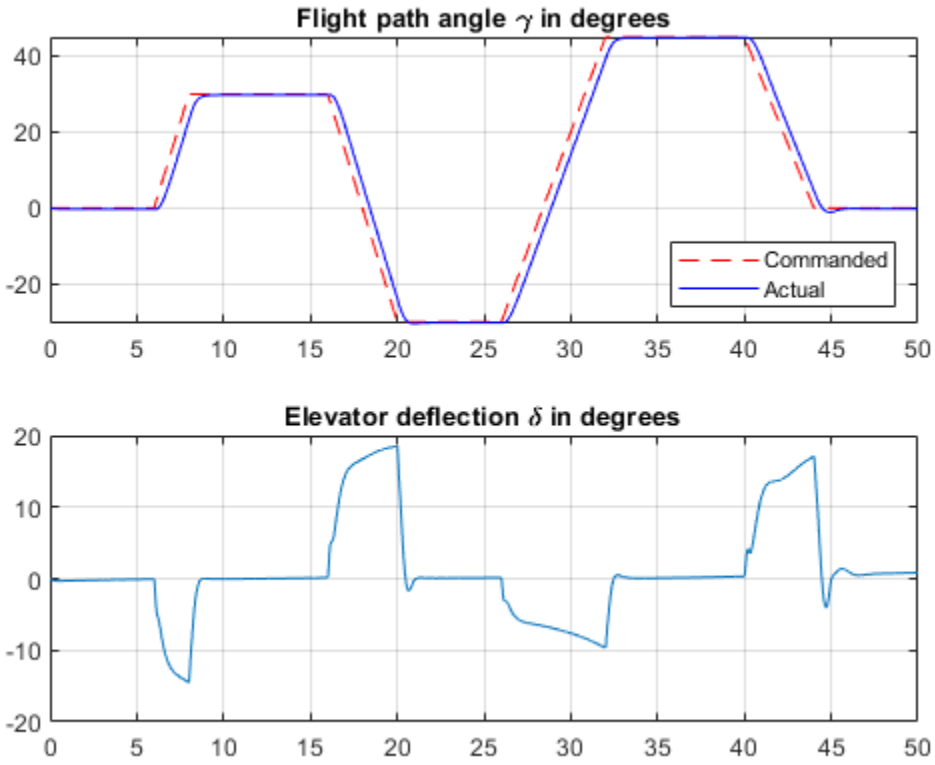
The responses are satisfactory at all flight conditions. Next validate the autopilot against the nonlinear airframe model. First use `writeBlockValue` to apply the tuning results to the Simulink model. This evaluates each gain surface formula at the breakpoints specified in the two Prelookup blocks and writes the result in the corresponding Interpolation block.

```
writeBlockValue(ST)
```

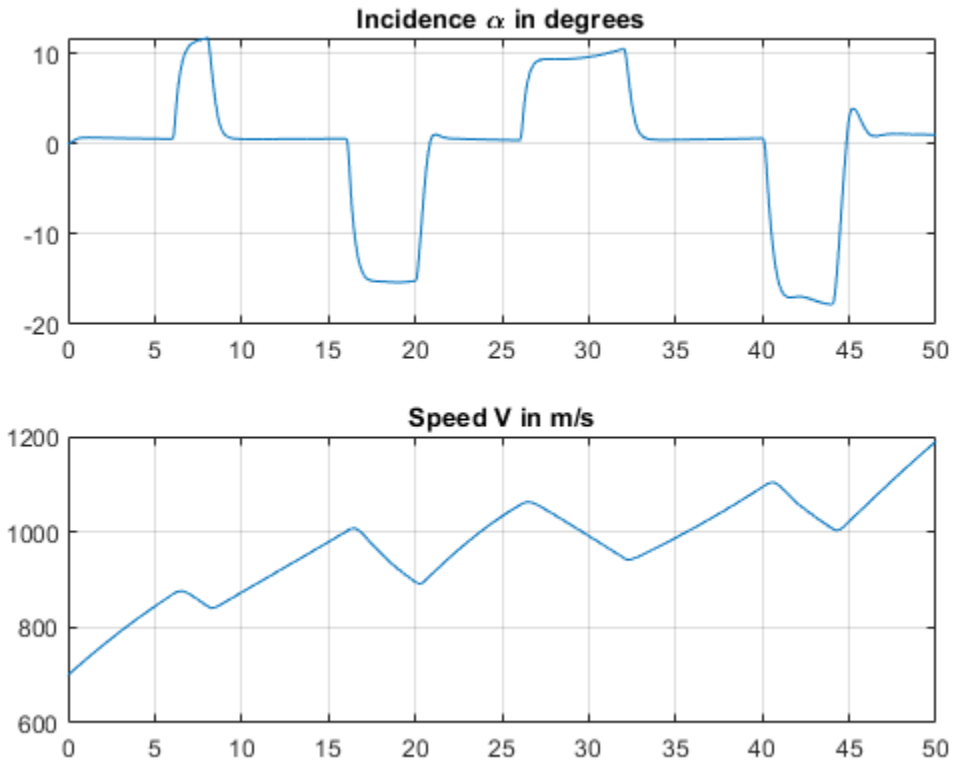
Simulate the autopilot performance for a maneuver that takes the airframe through a large portion of its flight envelope. The code below is equivalent to pressing the Play button in the Simulink model and inspecting the responses in the Scope blocks.

```
% 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');
```

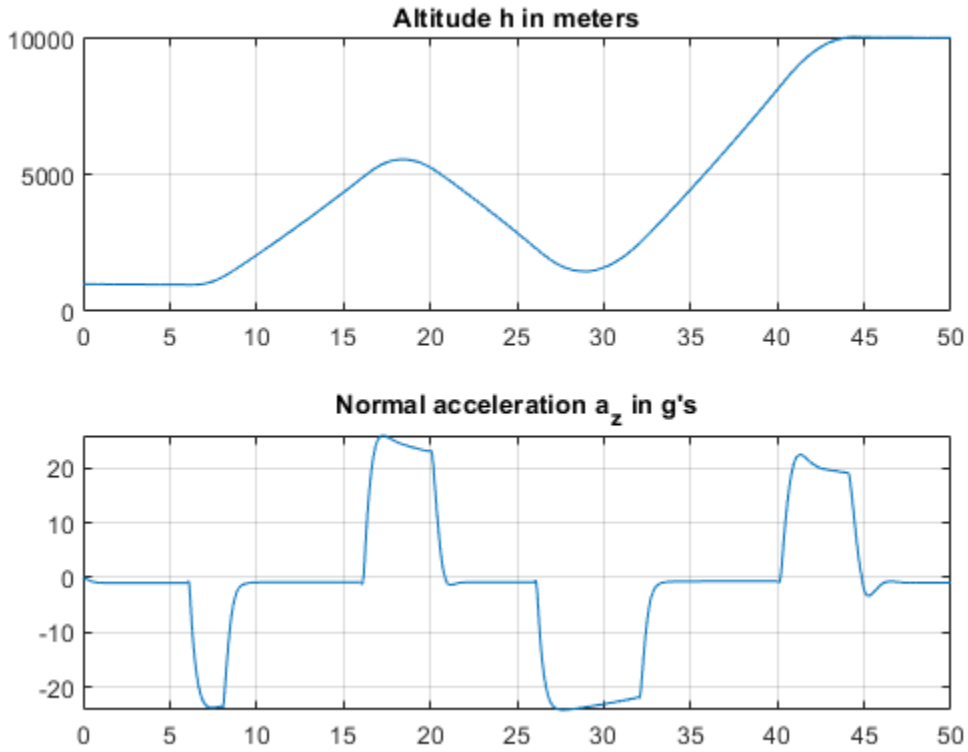
```
t = Sim_gamma.Values.Time;

% 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 16-4
- “Gain-Scheduled Control of a Chemical Reactor” on page 16-41

More About

- “Gain Scheduling Basics” on page 16-2
- “Parameterize Gain Schedules” on page 16-24

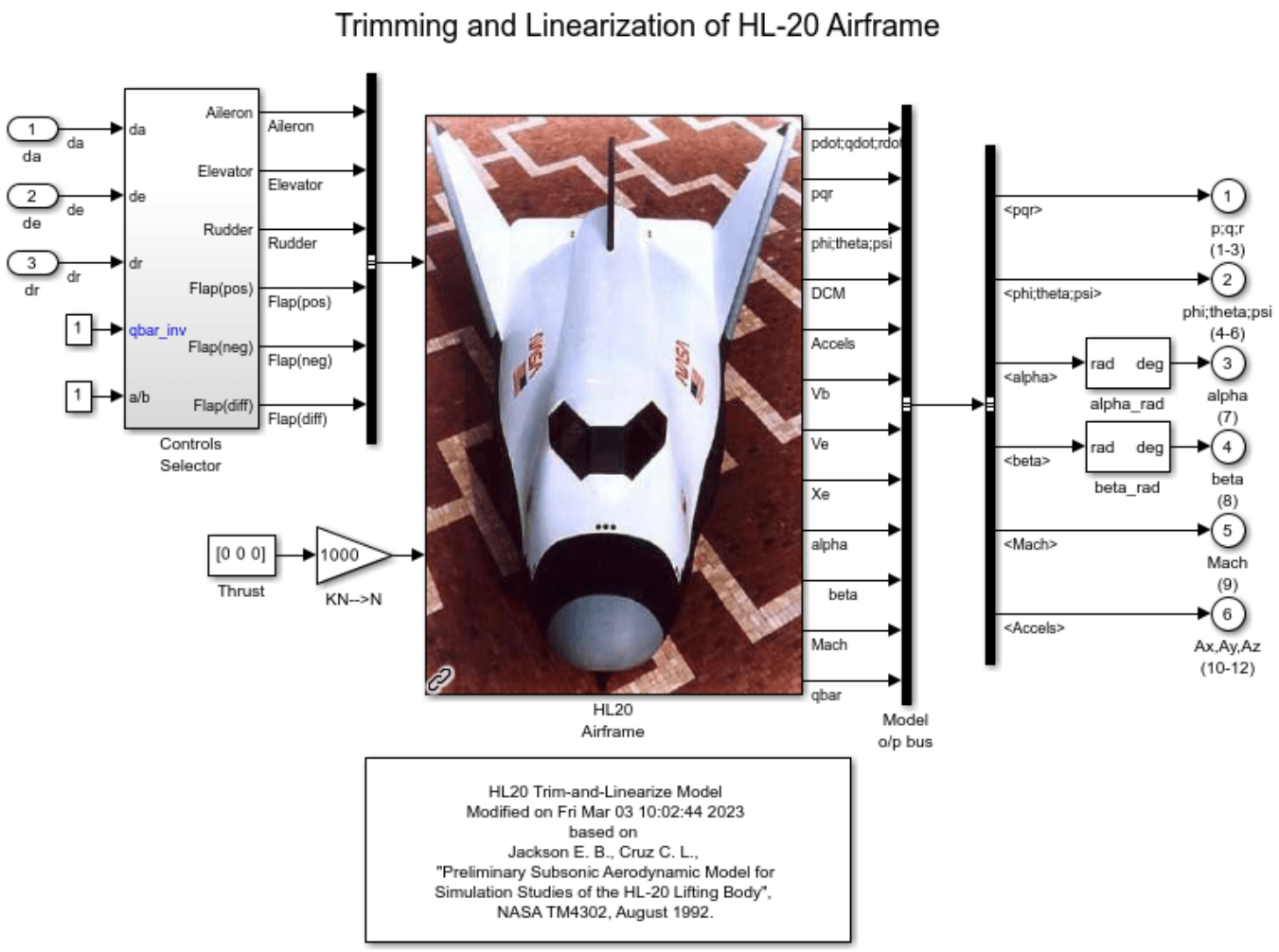
Trimming and Linearization of the HL-20 Airframe

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.

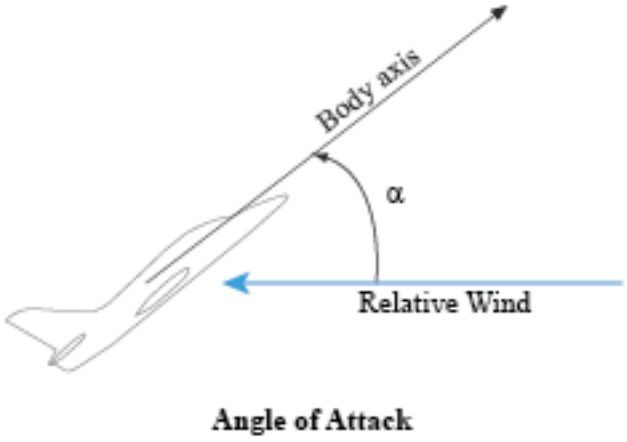
```
open_system('csthl20_trim')
```

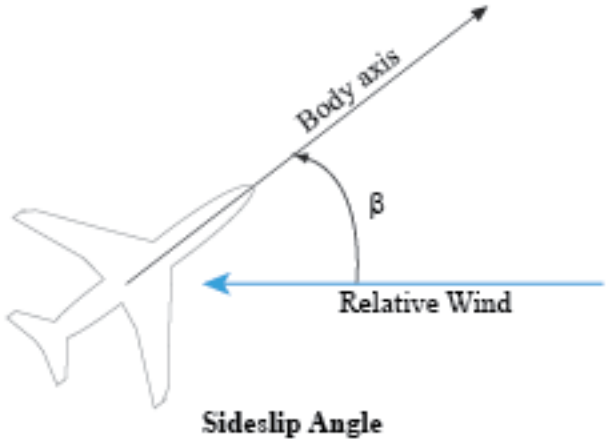


This version of the model includes the equations of motion (EOM), the force and moment calculation from the aerodynamic tables, the environment model, and the "Controls Selector" block which maps aileron, elevator, and rudder demands to deflections of the six control surfaces.

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:
----------
     x         Known     SteadyState      Min         Max        dxMin       dxMax
```

```
___________ ___________ ___________ ___________ ___________ ___________ ___________

(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
-12071.9115    false       true        -Inf         Inf        -Inf         Inf
     0         false       true        -Inf         Inf        -Inf         Inf
-3047.9999     false       true        -Inf         Inf        -Inf         Inf

Inputs:
----------
  u   Known  Min   Max
_____ _____ _____ _____

(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:
----------
  y   Known  Min   Max
_____ _____ _____ _____

(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:
----------
    x
__________

(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
 191.0911
 -34.2143
 -33.6945
(4.) csthl20_trim/HL20 Airframe/6DOF (Euler Angles)/xe,ye,ze
    0
    0
-3047.9999

Inputs:
----------
   u
________

(1.) csthl20_trim/da
-23.9841
(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” on page 16-75) 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” on page 16-75 for Part 2 of this example.

See Also

More About

- “Angular Rate Control in the HL-20 Autopilot” on page 16-75
- “Model Gain-Scheduled Control Systems in Simulink” on page 16-4

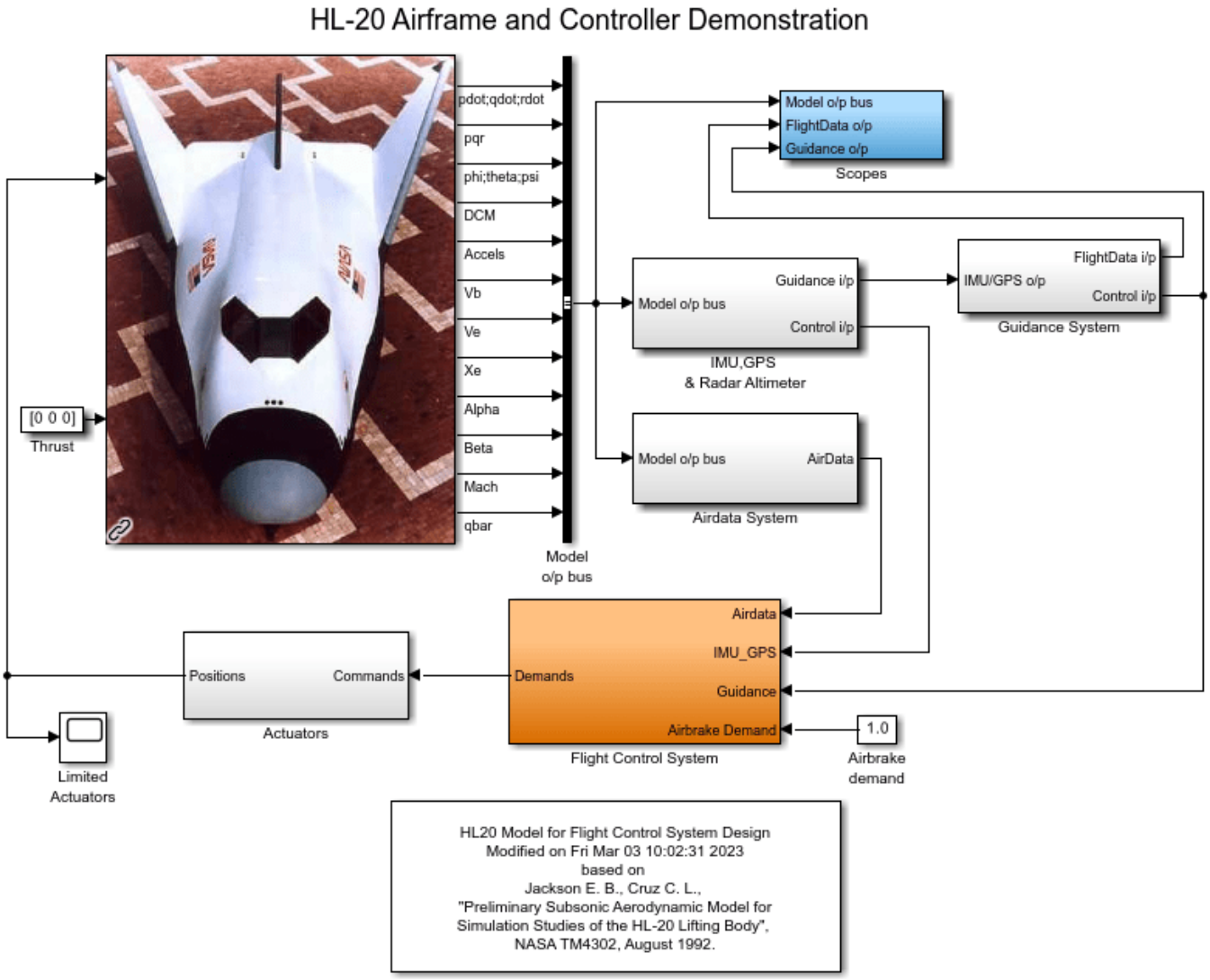
Angular Rate Control in the HL-20 Autopilot

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

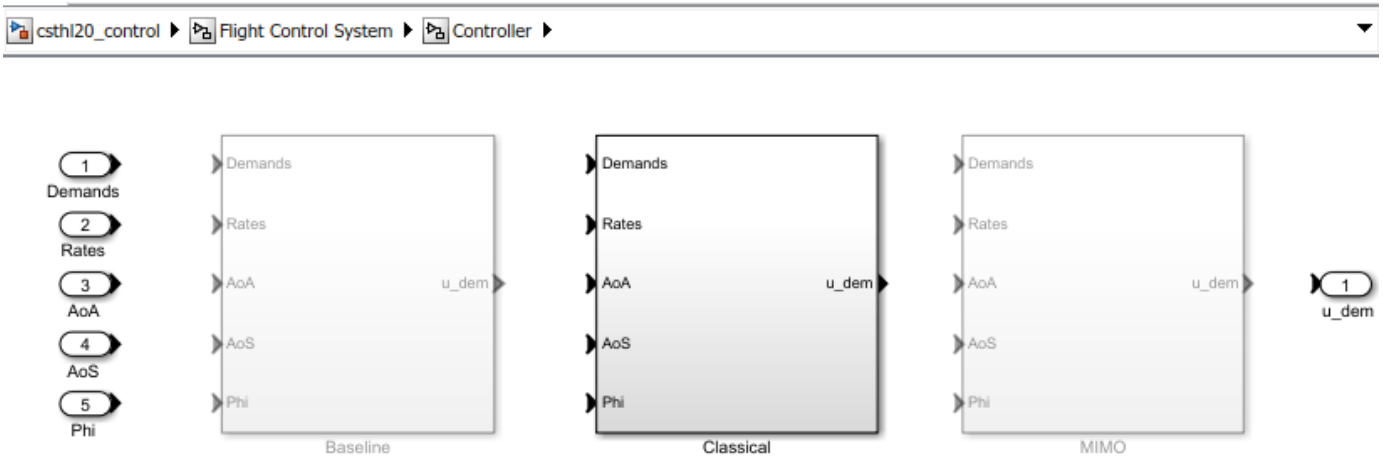
Open the HL-20 model with its flight control system.

```
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

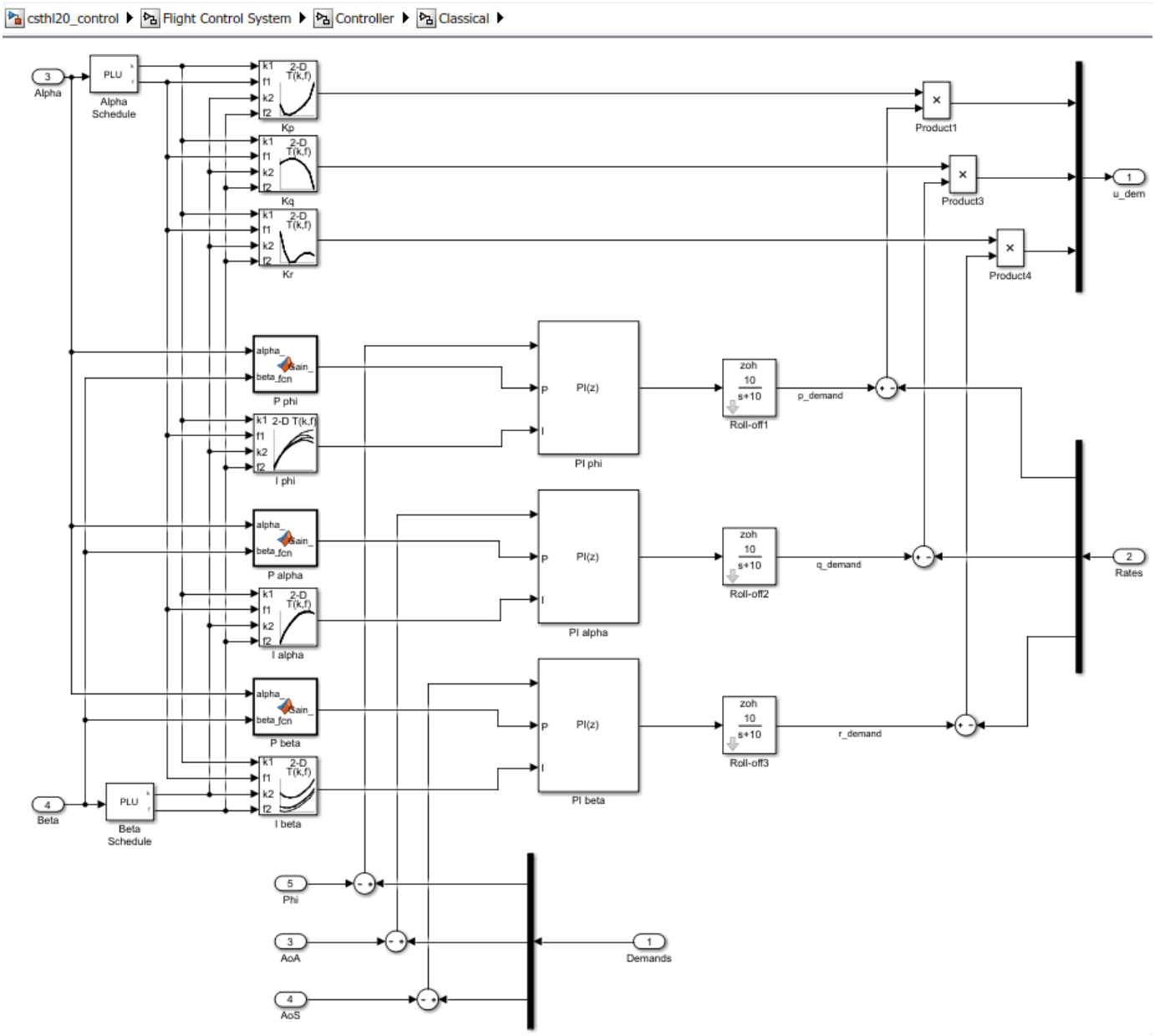
these commands. The "Controller" block inside the "Flight Control System" is a variant subsystem with different autopilot configurations.



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” on page 16-68), 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.

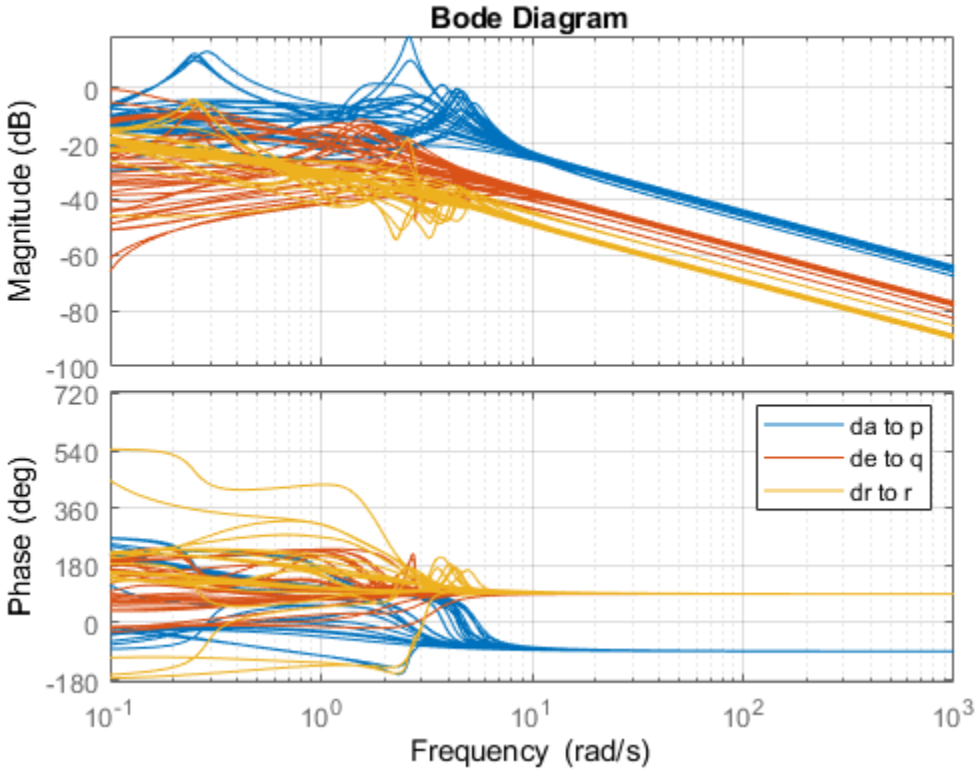
```
% Replace "HL20 Airframe" block by 8-by-5 array of linearized models G7
BlockSub1 = struct('Name','csthl20_control/HL20 Airframe','Value',G7);

% 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

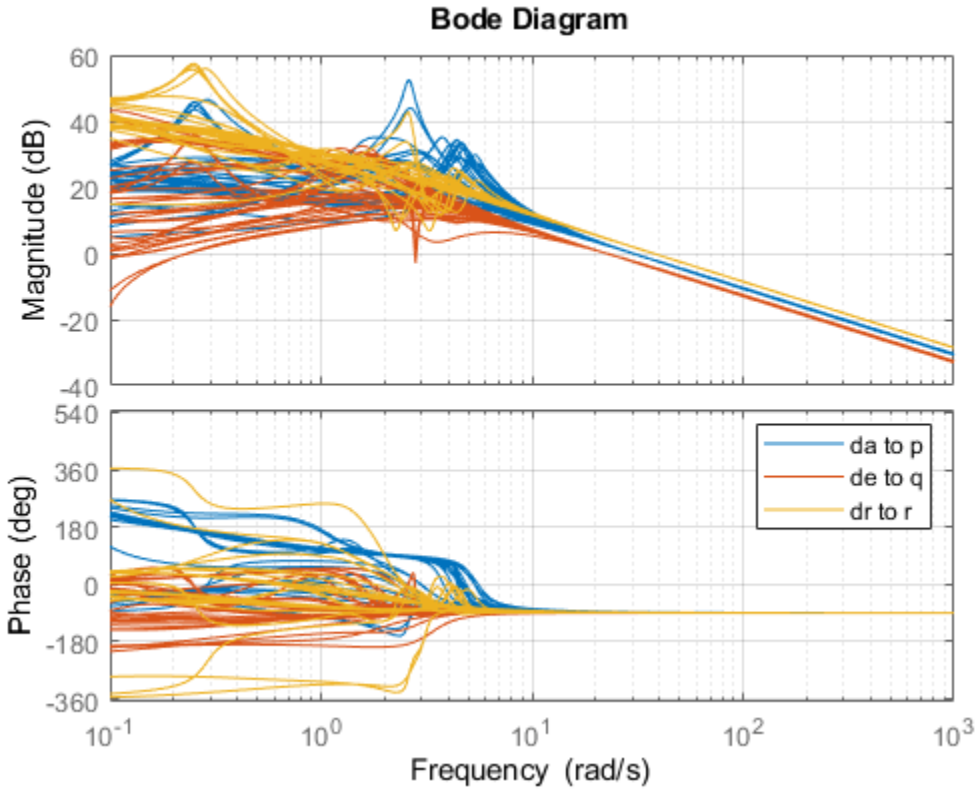
Begin with the three inner loops controlling the angular rates p,q,r. To get oriented, plot the open-loop transfer function from deflections (da,de,dr) to angular rates (p,q,r). With the `slTuner` interface, you can query the model for any transfer function of interest.

```
% 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” on page 16-81) shows how to tune a classic SISO architecture and Part 4 (“Attitude Control in the HL-20 Autopilot - MIMO Design” on page 16-91) looks into the benefits of a MIMO architecture.

See Also

More About

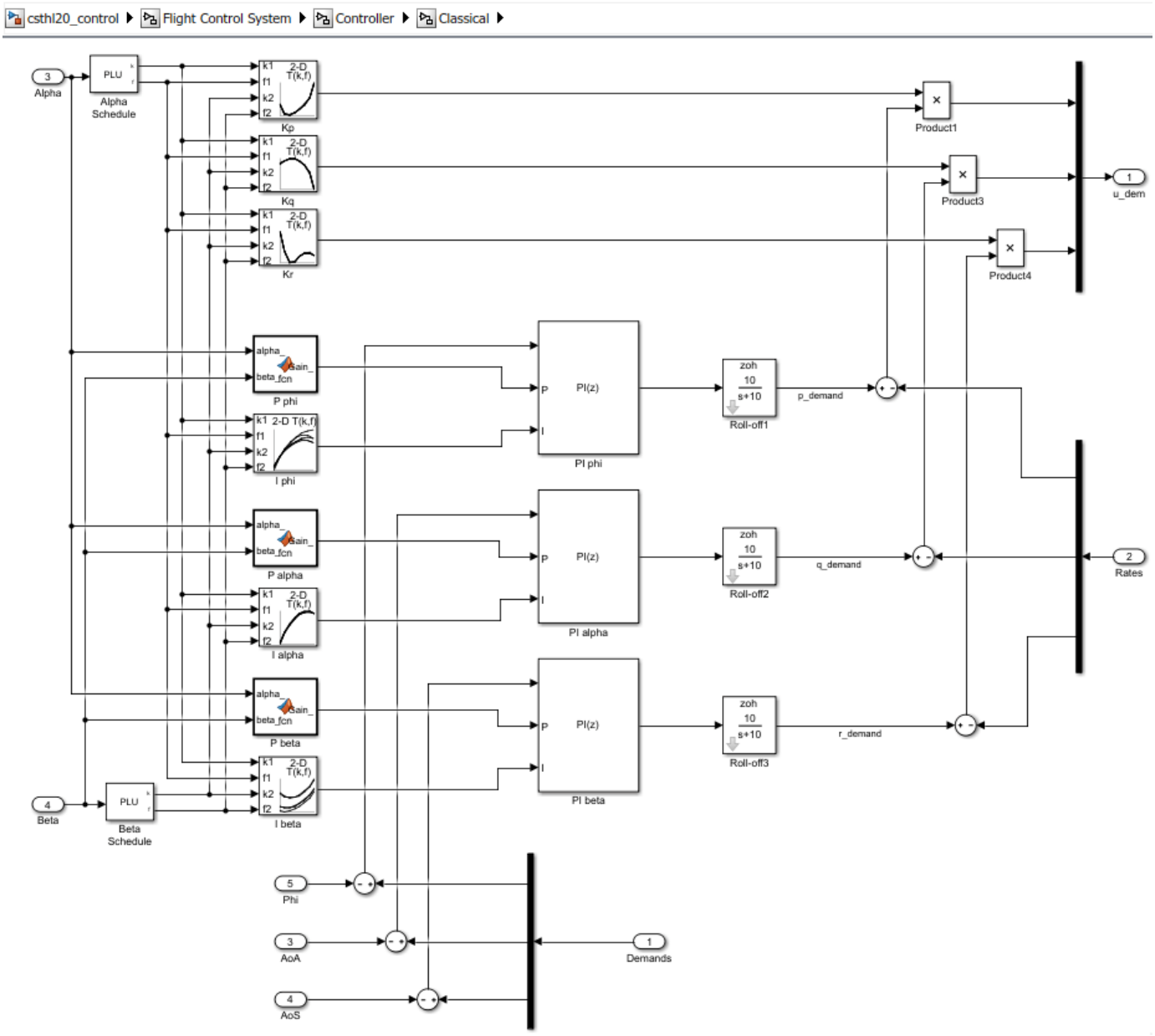
- “Trimming and Linearization of the HL-20 Airframe” on page 16-68
- “Attitude Control in the HL-20 Autopilot - SISO Design” on page 16-81
- “Tune Gain Schedules in Simulink” on page 16-12

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” on page 16-68) 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” on page 16-75), 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)]);
```

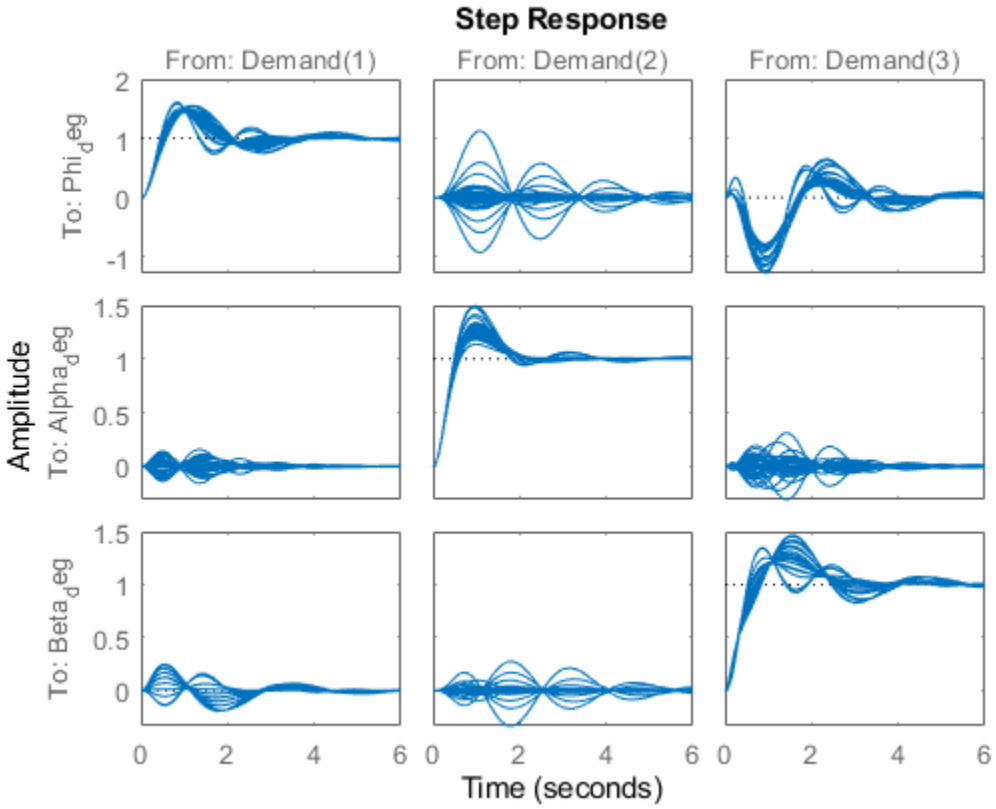
It is easily verified that this block linearization amounts to multiplying the error signal by the varying quantity Kp computed above. Similarly, replace the corresponding Product blocks for Kq and Kr by varying gains.

```
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.

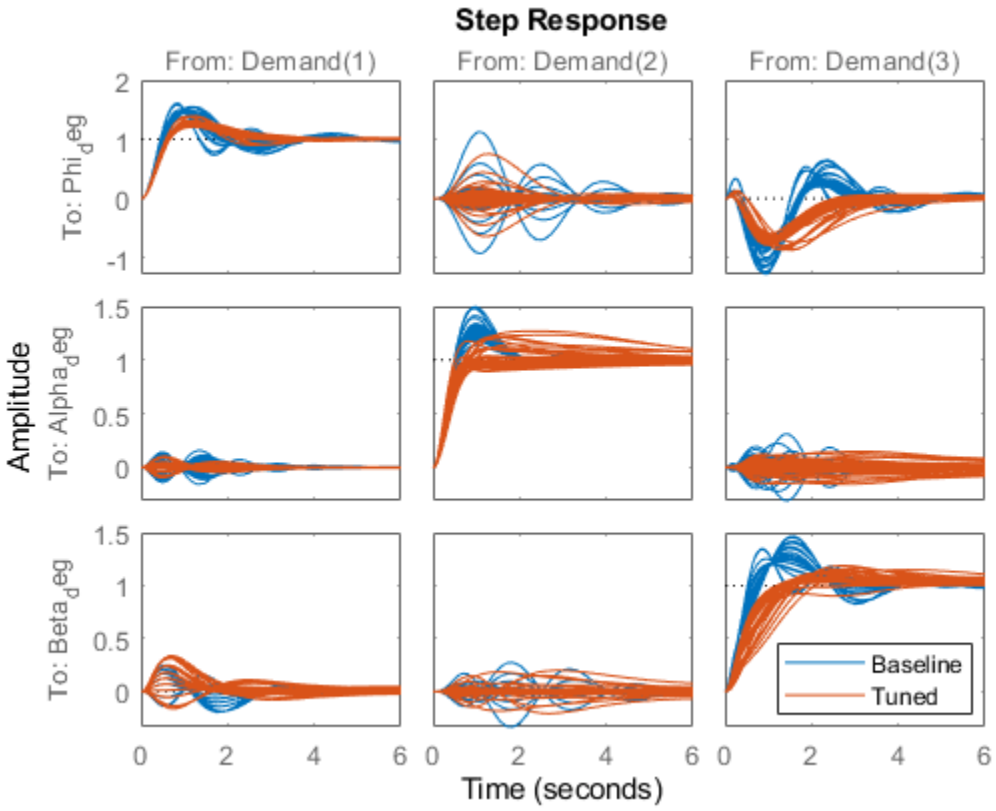
You can now use `systune` to tune the 6 gain surfaces against the three tuning goals.

```
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.

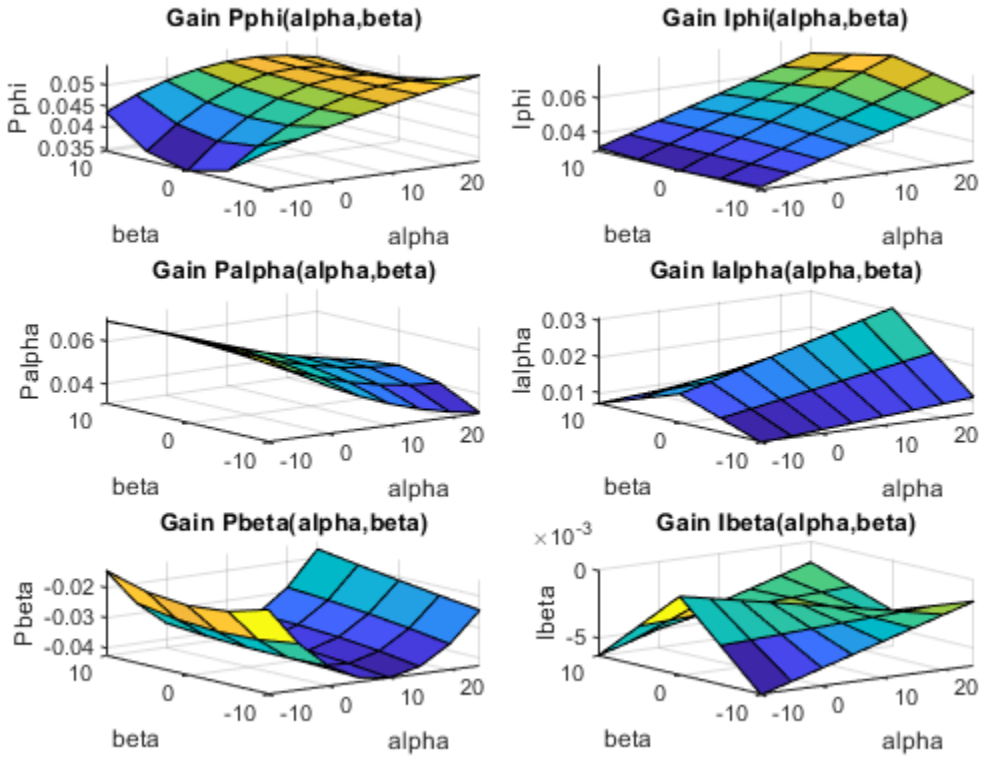
```
T = getIOTransfer(ST,'Demand',{'Phi_deg','Alpha_deg','Beta_deg'});
step(T0,T,6)
legend('Baseline','Tuned','Location','SouthEast')
```



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)
```

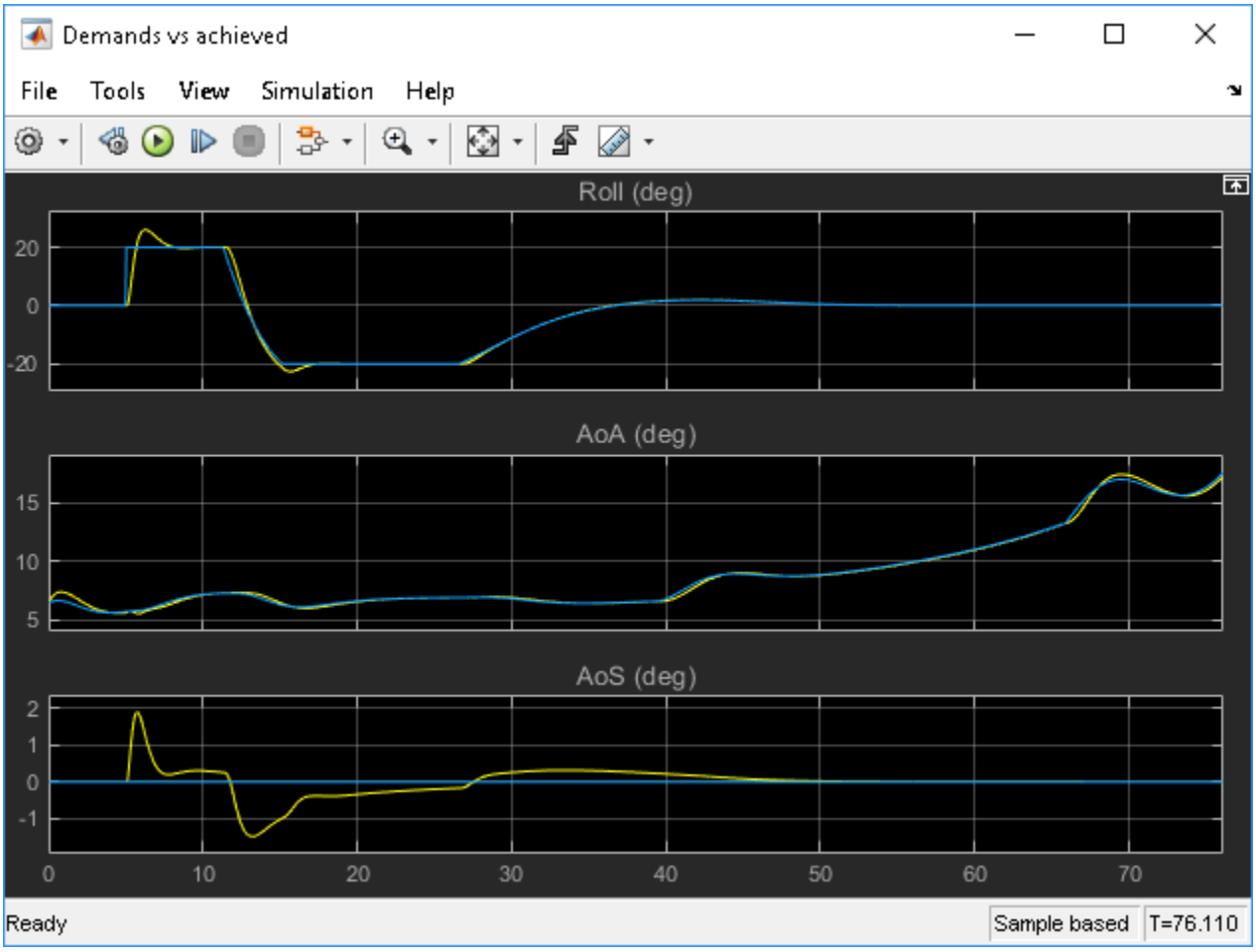
For the three lookup table blocks "I phi", "I alpha", "I beta", `writeBlockValue` samples the gain surfaces at the table breakpoints and updates the table data in the model workspace. For the MATLAB Function blocks "P phi", "P alpha", "P beta", `writeBlockValue` generates MATLAB code for the gain surface 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” on page 16-91).

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

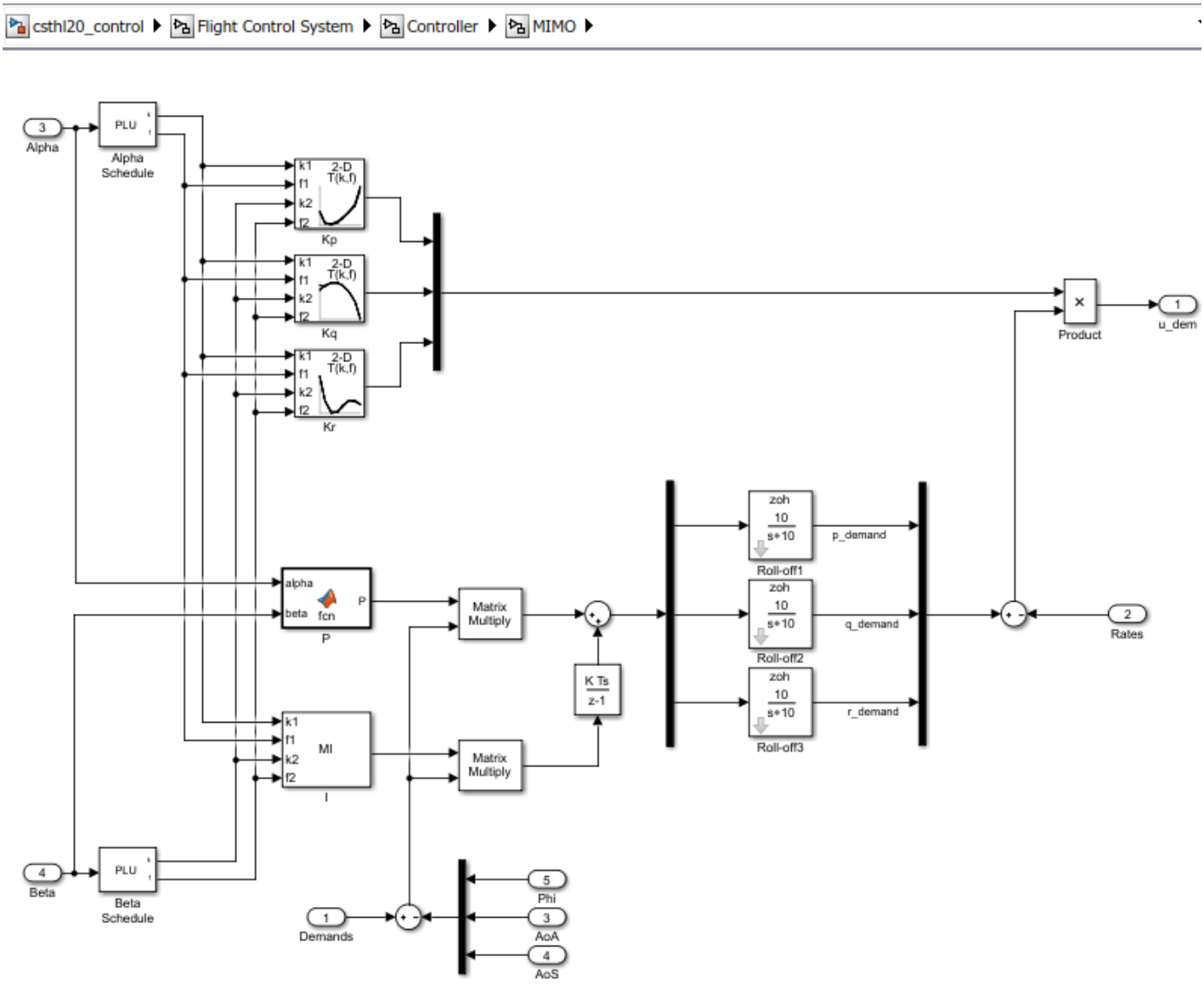
- “Attitude Control in the HL-20 Autopilot - MIMO Design” on page 16-91
- “Tune Gain Schedules in Simulink” on page 16-12

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” on page 16-68) 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” on page 16-75 and “Attitude Control in the HL-20 Autopilot - SISO Design” on page 16-81 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

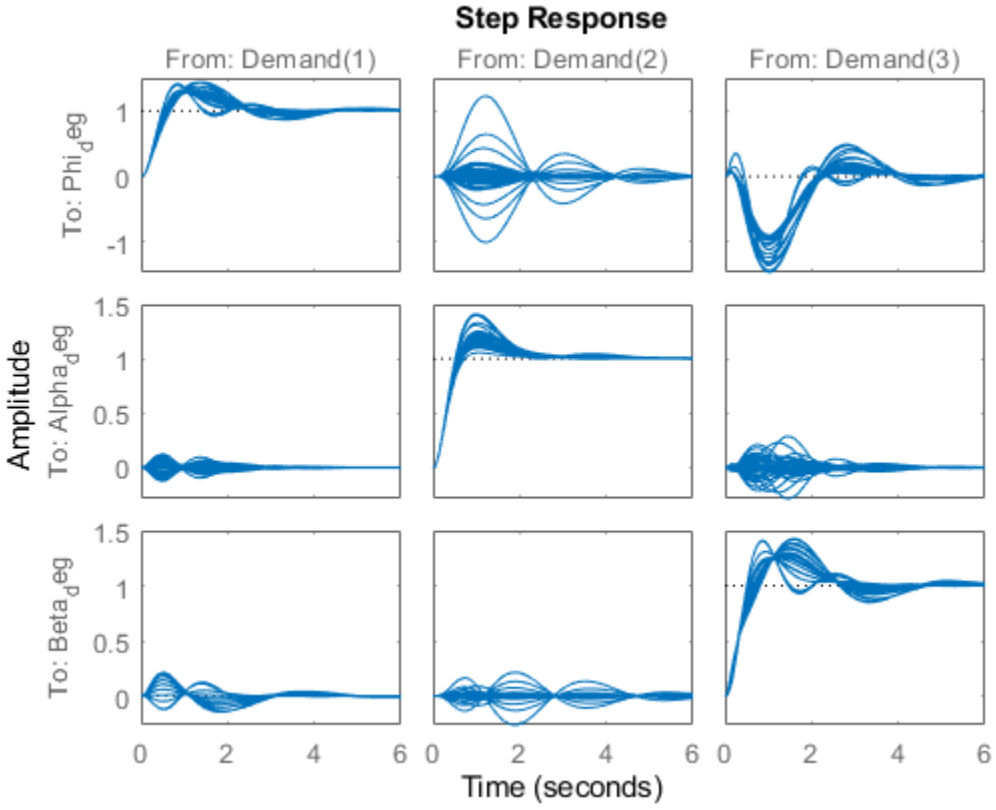
As in the SISO design (“Attitude Control in the HL-20 Autopilot - SISO Design” on page 16-81), the first step is to obtain a linearized model of the "plant" seen by the outer loops at each (alpha,beta) condition. To account for the fact that the inner-loop gains Kp,Kq,Kr vary with (alpha,beta), replace 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)
```

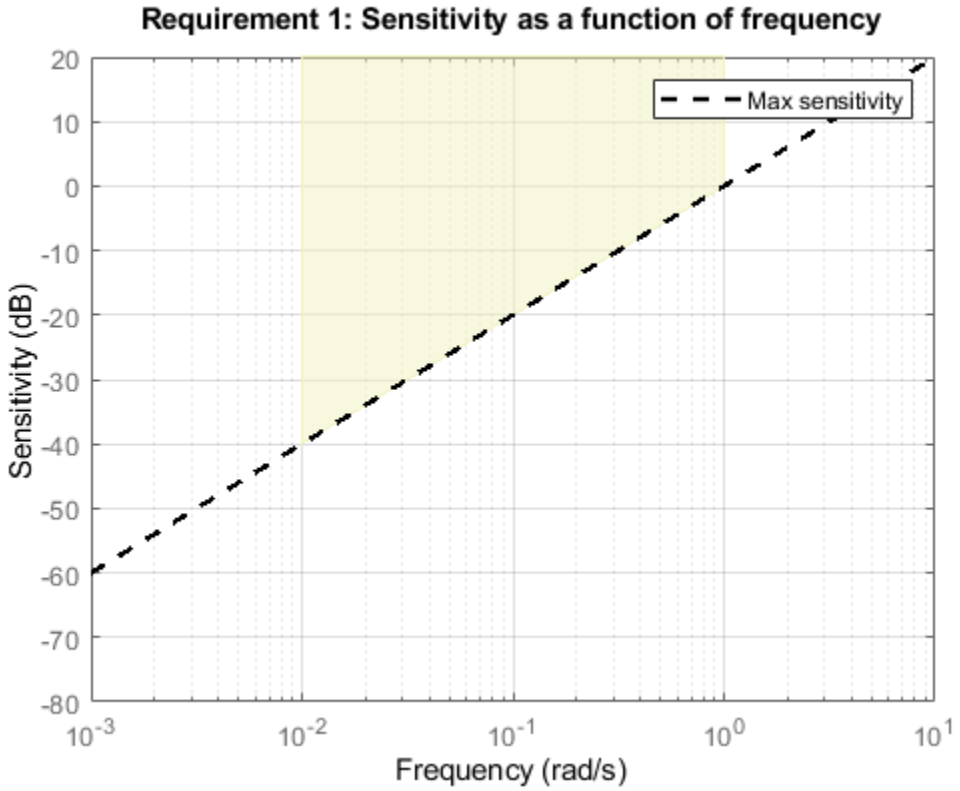


Tuning Goals

To tune the MIMO gain schedules we use the following three tuning goals:

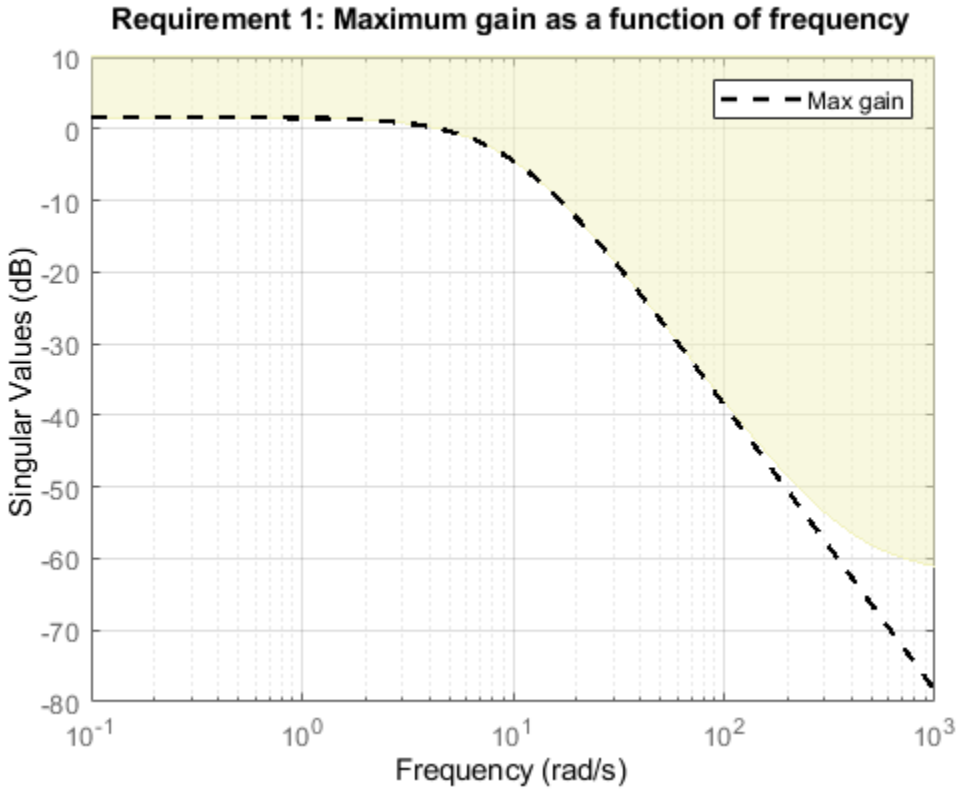
- A "Sensitivity" goal to specify the desired bandwidth (response time) and maximize decoupling at low frequency.

```
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);
```

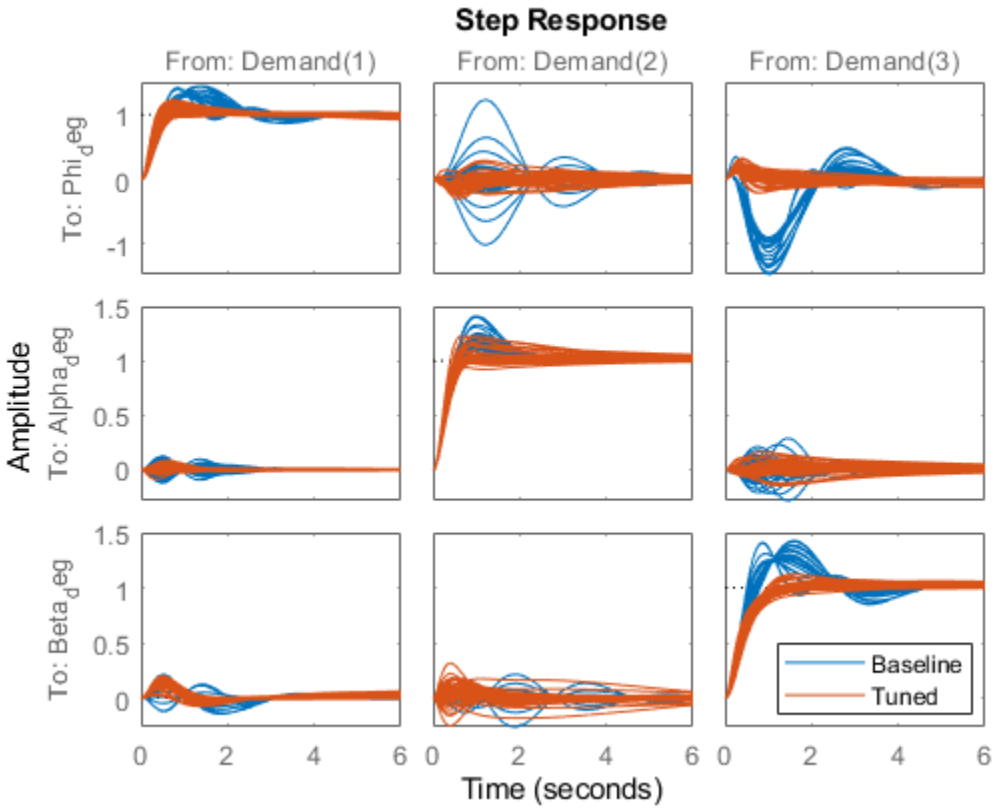
Finally, use `systune` to tune the 6 gain surfaces against the three tuning goals.

```
ST = systune(ST0,[R1 R2 R3]);
```

```
Final: Soft = 1.13, Hard = -Inf, Iterations = 109
```

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.

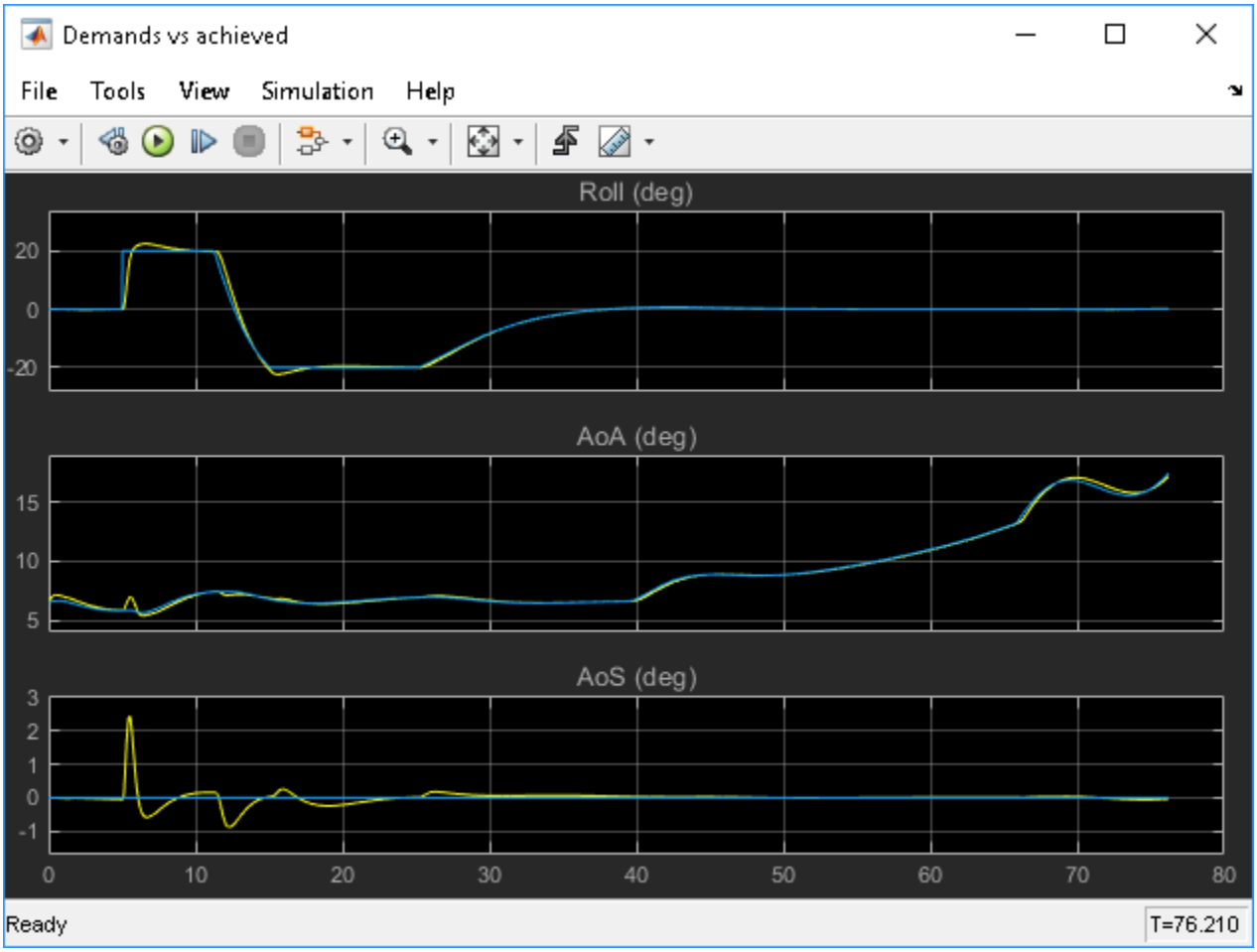
Validation

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” on page 16-81) 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

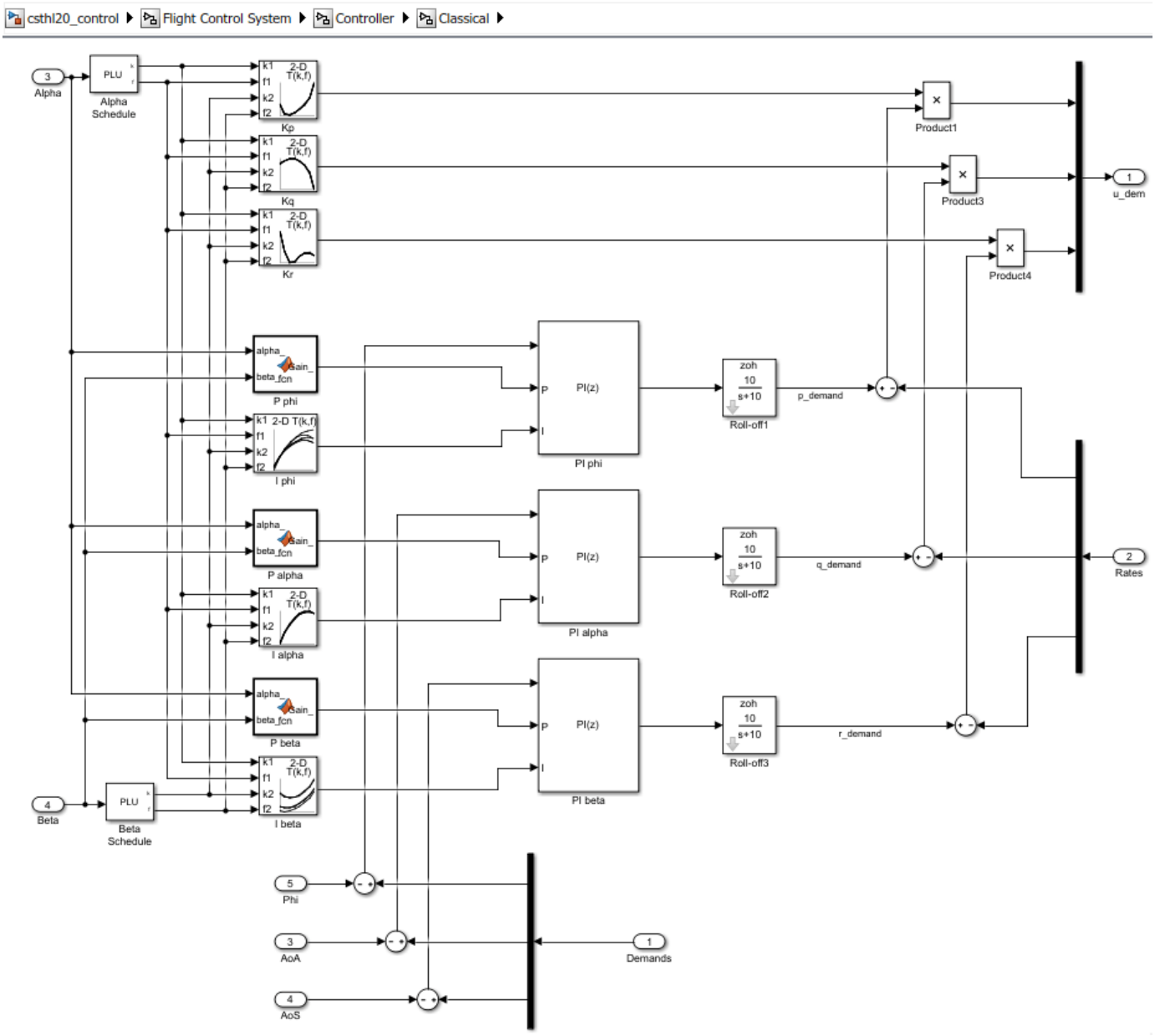
- “Attitude Control in the HL-20 Autopilot - SISO Design” on page 16-81
- “Tune Gain Schedules in Simulink” on page 16-12

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” on page 16-68) 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” on page 16-75) and Part 3 (“Attitude Control in the HL-20 Autopilot - SISO Design” on page 16-81), 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” on page 16-68). 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)
```

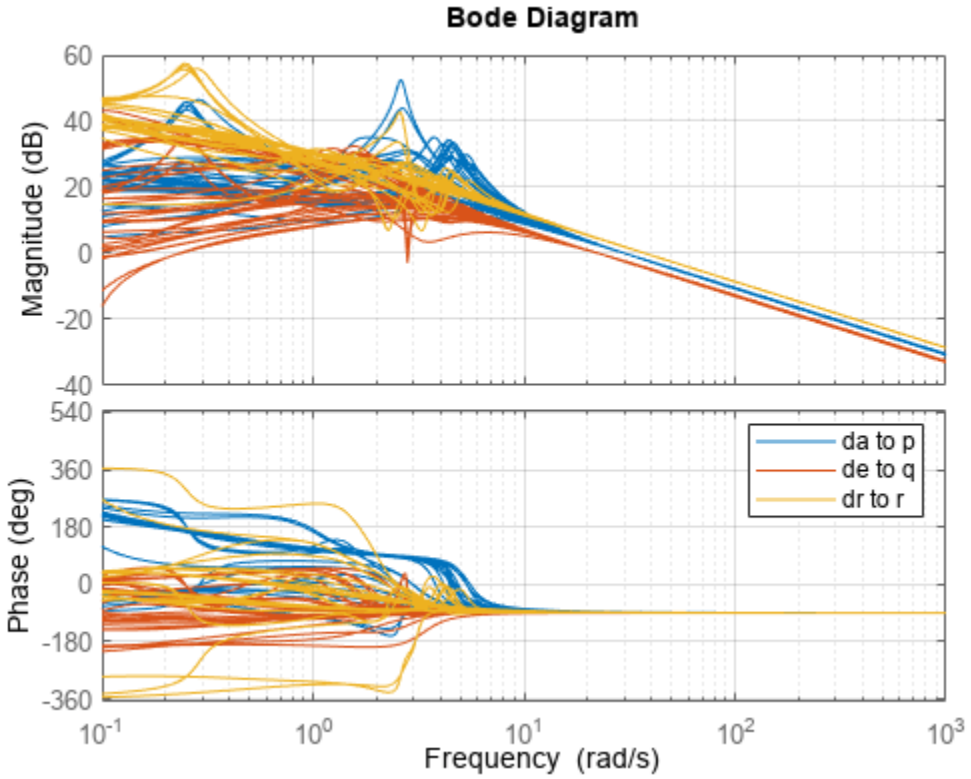
```
8x5 array of state-space models.
Each model has 6 outputs, 3 inputs, and 7 states.
```

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” on page 16-75). 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” on page 16-81), 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.

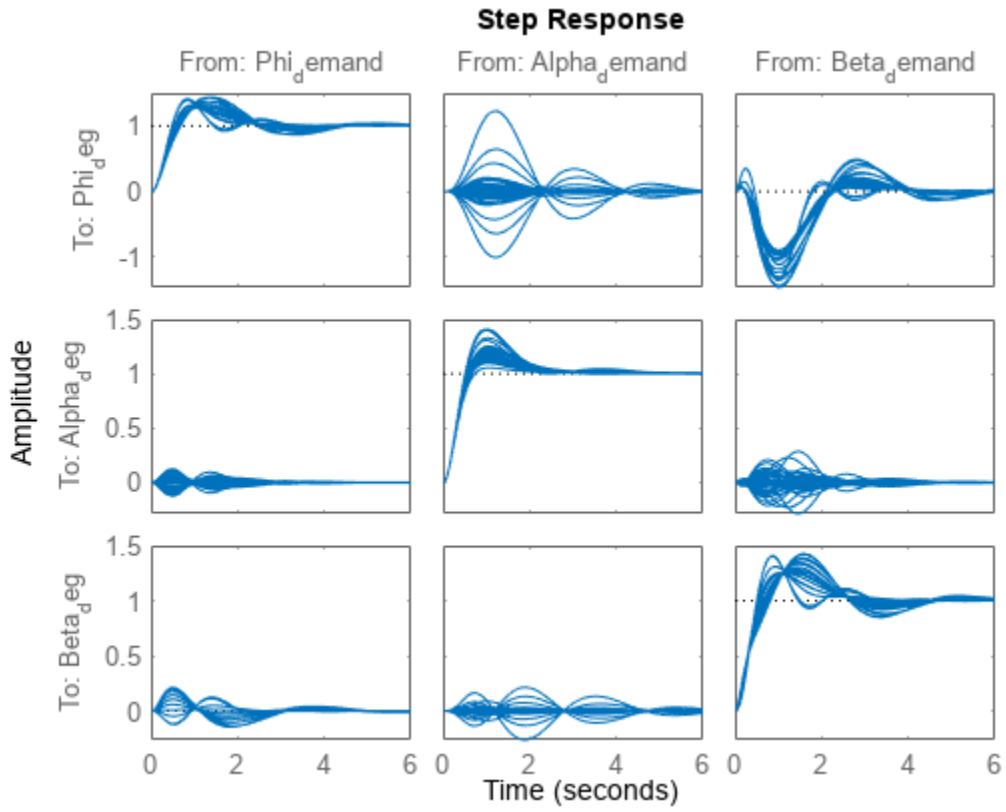
```
KP = append(P_PHI,P_ALPHA,P_BETA);
KI = append(I_PHI,I_ALPHA,I_BETA);
Cpos = KP + KI * tf(1,[1 0]);
```

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” on page 16-81). 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

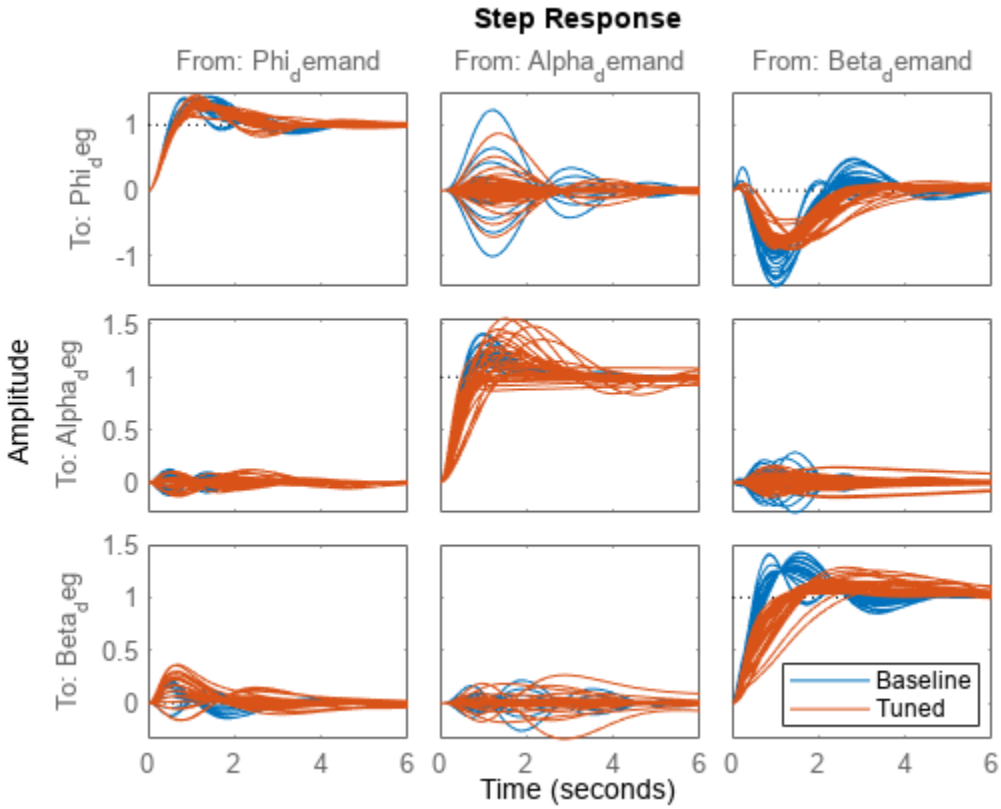
You can now use `systune` to shape the six gain surfaces against the tuning goals at all 40 design points.

```
T = systune(T0,[R1 R2 R3]);
```

```
Final: Soft = 1.03, Hard = -Inf, Iterations = 52
```

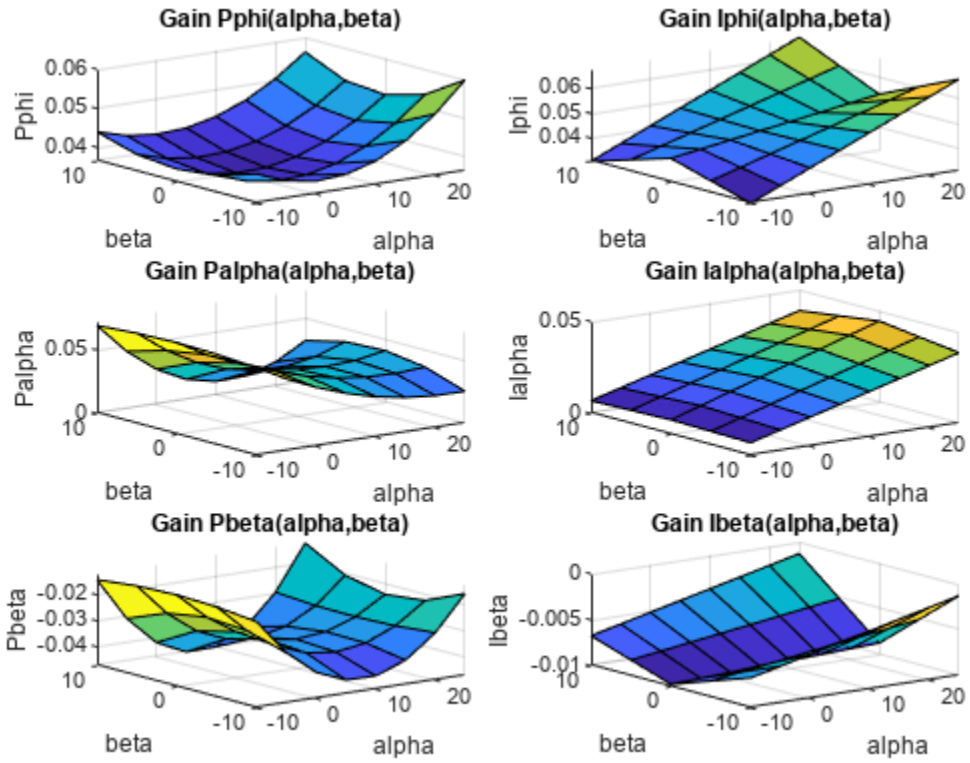
The final objective value is close to 1 so the tuning goals are essentially met. Plot the closed-loop angular responses and compare with the initial settings.

```
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” on page 16-68
- “Tune Gain Schedules in Simulink” on page 16-12

Control System Tuning Examples - Generalized LTI Models

- “Tune Control Systems Using systune” on page 17-2
- “Building Tunable Models” on page 17-9
- “Active Vibration Control in Three-Story Building” on page 17-15
- “Vibration Control in Flexible Beam” on page 17-25
- “Passive Control with Communication Delays” on page 17-34
- “Tune Phase-Locked Loop Using Loop-Shaping Design” on page 17-41
- “Feedback Amplifier Design for Voltage-Mode Boost Converter” on page 17-56

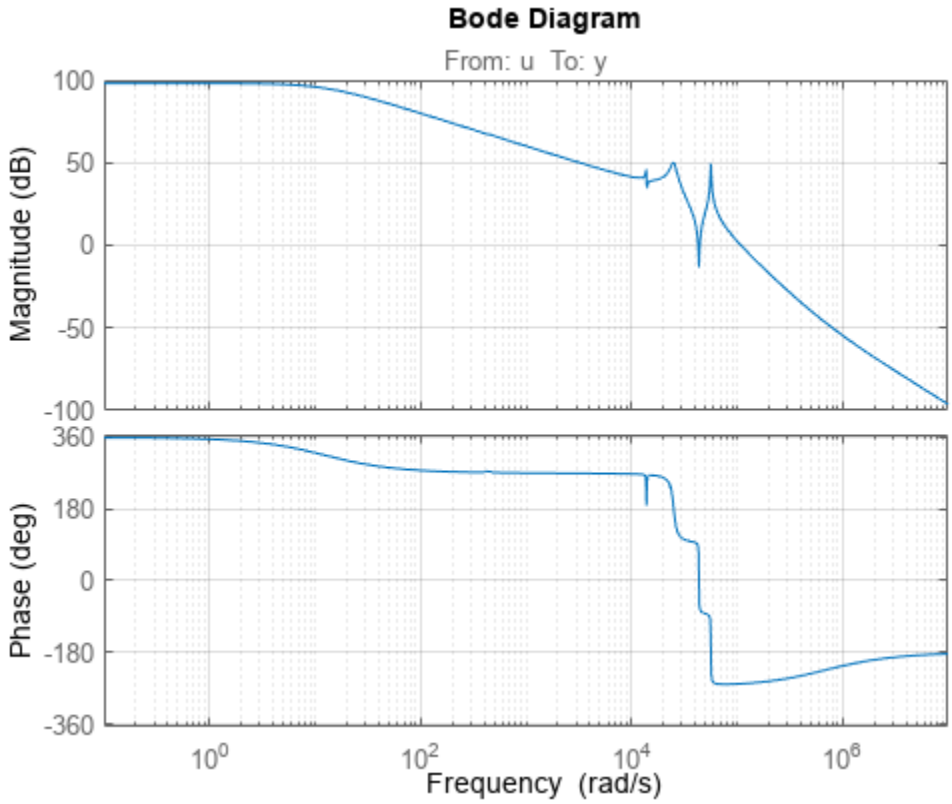
Tune Control Systems Using systune

The `systune` command can jointly tune the gains of your control system regardless of its architecture and number of feedback loops. This example outlines the `systune` workflow on a simple application.

Head-Disk Assembly Control

This example uses a 9th-order model of the head-disk assembly (HDA) in a hard-disk drive. This model captures the first few flexible modes in the HDA.

```
load rctExamples G
bode(G), grid
```



We use the feedback loop shown below to position the head on the correct track. This control structure consists of a PI controller and a low-pass filter in the return path. The head position `y` should track a step change `r` with a response time of about one millisecond, little or no overshoot, and no steady-state error.

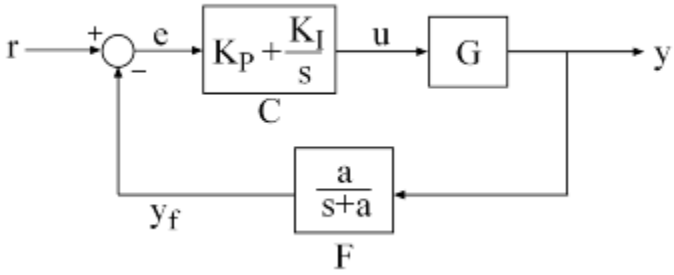


Figure 1: Control Structure

You can use `systune` to directly tune the PI gains and filter coefficient a subject to a variety of time- and frequency-domain requirements.

Specify Tunable Elements

There are two tunable elements in the control structure of Figure 1: the PI controller $C(s)$ and the low-pass filter

$$F(s) = \frac{a}{s+a}.$$

You can use the `tunablePID` object to parameterize the PI block:

```
C0 = tunablePID('C','pi');  % tunable PI
```

To parameterize the lowpass filter $F(s)$, create a tunable real parameter a and construct a first-order transfer function with numerator a and denominator $s + a$:

```
a = realp('a',1);     % filter coefficient
F0 = tf(a,[1 a]);     % filter parameterized by a
```

See the "*Building Tunable Models*" example for an overview of available tunable elements.

Build Tunable Closed-Loop Model

Next build a closed-loop model of the feedback loop in Figure 1. To facilitate open-loop analysis and specify open-loop requirements such as desired stability margins, add an analysis point at the plant input `u`:

```
AP = AnalysisPoint('u');
```

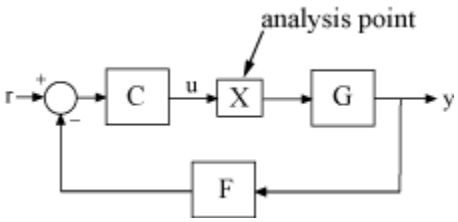


Figure 2: Analysis Point Block

Use `feedback` to build a model of the closed-loop transfer from reference `r` to head position `y`:

```
T0 = feedback(G*AP*C0,F0);  % closed-loop transfer from r to y
T0.InputName = 'r';
T0.OutputName = 'y';
```

The result `T0` is a generalized state-space model (`genss`) that depends on the tunable elements C and F.

Specify Design Requirements

To specify the desired behavior of your system, you can define control design requirements using tuning goal objects.You can specify requirements such as the response time, deterministic and stochastic gains, loop shape, stability margins, and pole locations. Here, we use two requirements to capture the control objectives:

- **Tracking requirement** : The position `y` should track the reference `r` with a 1 millisecond response time
- **Stability margin requirement** : The feedback loop should have 6dB of gain margin and 45 degrees of phase margin

Use the `TuningGoal.Tracking` and `TuningGoal.Margins` objects to capture these requirements. Note that the margins requirement applies to the open-loop response measured at the plant input `u` (location marked by the analysis point `AP`).

```
Req1 = TuningGoal.Tracking('r','y',0.001);
Req2 = TuningGoal.Margins('u',6,45);
```

Tune Controller Parameters

You can now use `systune` to tune the PI gain and filter coefficient a. This function takes the tunable closed-loop model `T0` and the requirements `Req1,Req2`. Use a few randomized starting points to improve the chances of getting a globally optimal design.

```
rng('default')
Options = systuneOptions('RandomStart',3);
[T,fSoft] = systune(T0,[Req1,Req2],Options);
```

```
Final: Soft = 1.35, Hard = -Inf, Iterations = 73
Final: Soft = 1.35, Hard = -Inf, Iterations = 115
Final: Soft = 2.78e+03, Hard = -Inf, Iterations = 160
       Some closed-loop poles are marginally stable (decay rate near 1e-07)
Final: Soft = 1.35, Hard = -Inf, Iterations = 63
```

All requirements are normalized so a requirement is satisfied when its value is less than 1. Here the final value is slightly greater than 1, indicating that the requirements are nearly satisfied. Use the output `fSoft` to see the tuned value of each requirement. Here we see that the first requirement (tracking) is slightly violated while the second requirement (margins) is satisfied.

```
fSoft
```

```
fSoft = 1×2

    1.3461    0.6326
```

The first output `T` of `systune` is the "tuned" closed-loop model. Use `showTunable` or `getBlockValue` to access the tuned values of the PI gains and filter coefficient:

```
getBlockValue(T,'C')  % tuned value of PI controller
```

```
ans =

             1
  Kp + Ki * ---
             s

  with Kp = 0.00104, Ki = 0.0122

Name: C
Continuous-time PI controller in parallel form.
```

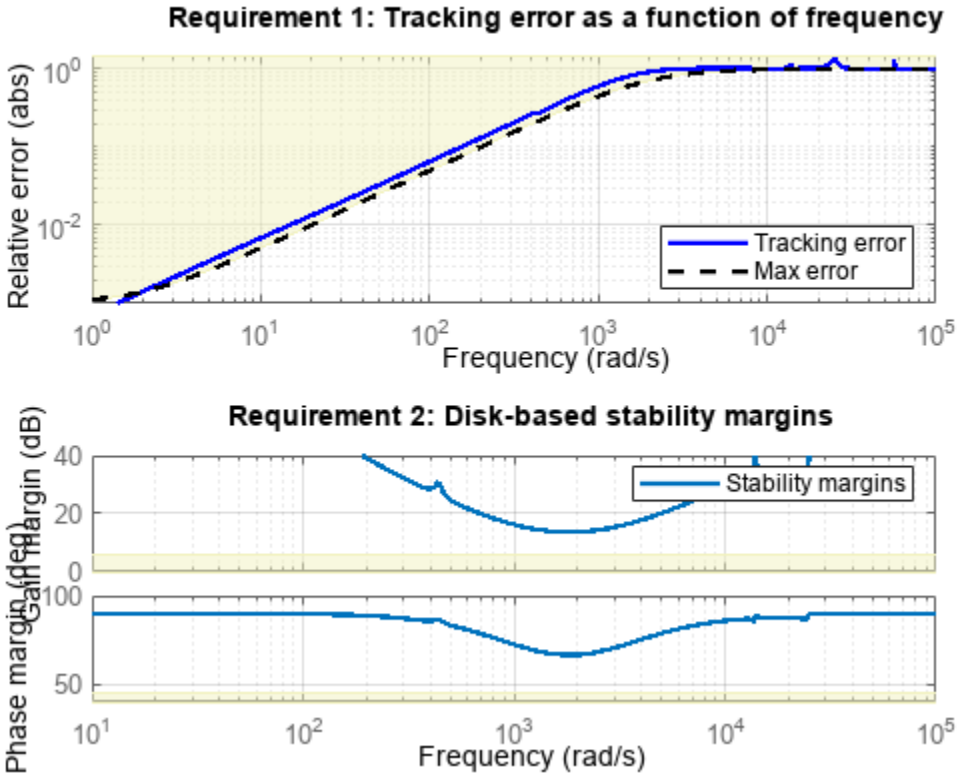
```
showTunable(T)  % tuned values of all tunable elements
```

```
C =

             1
  Kp + Ki * ---
             s

  with Kp = 0.00104, Ki = 0.0122

Name: C
Continuous-time PI controller in parallel form.
-----------------------------------
a = 3.19e+03
```

Validate Results

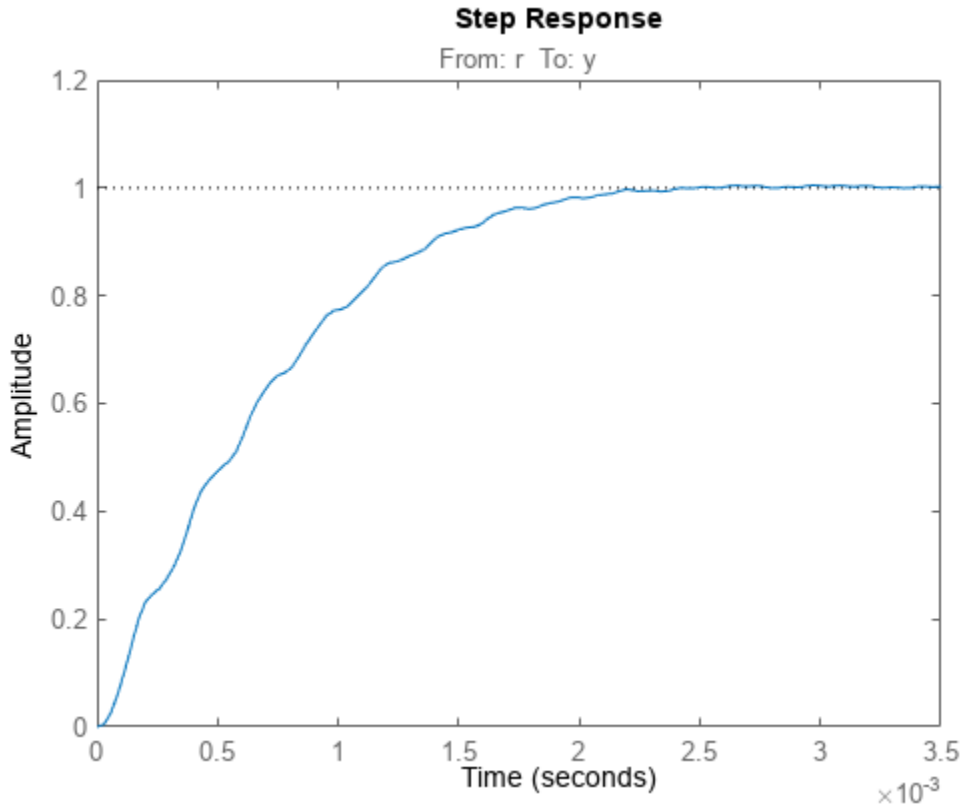
First use `viewGoal` to inspect how the tuned system does against each requirement. The first plot shows the tracking error as a function of frequency, and the second plot shows the normalized disk margins as a function of frequency (see `diskmargin`). See the "*Creating Design Requirements*" example for details.

```
clf, viewGoal([Req1 Req2],T)
```



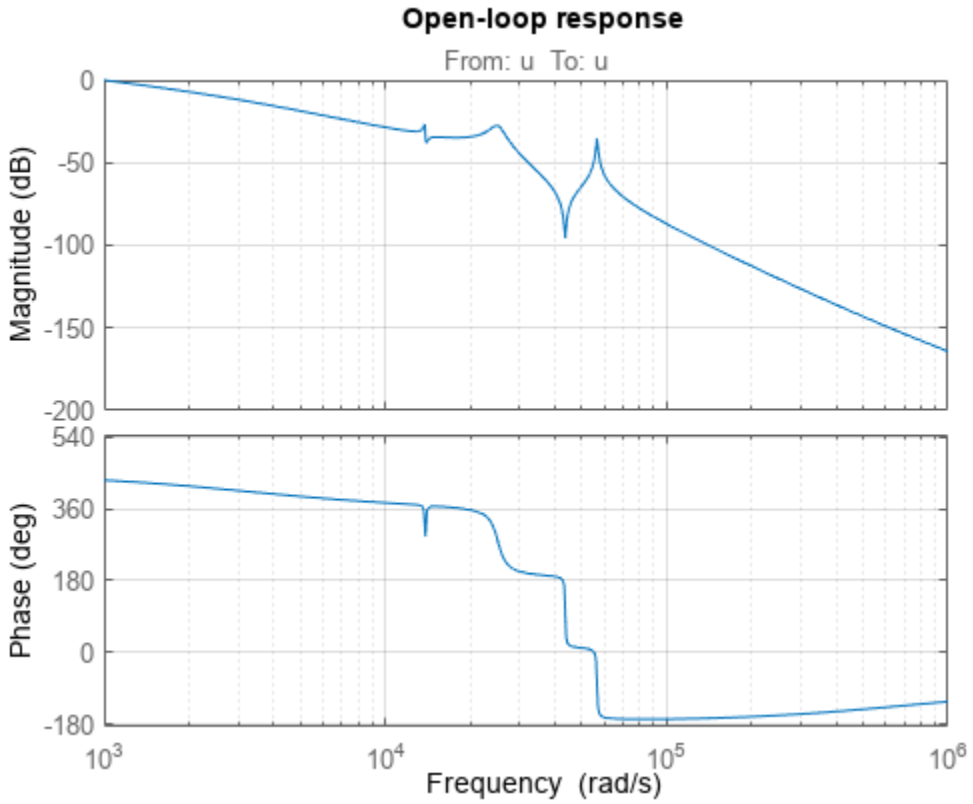
Next plot the closed-loop step response from reference `r` to head position `y`. The response has no overshoot but wobbles a little.

```
clf, step(T)
```



To investigate further, use `getLoopTransfer` to get the open-loop response at the plant input.

```
L = getLoopTransfer(T,'u');
bode(L,{1e3,1e6}), grid
title('Open-loop response')
```



The wobble is due to the first resonance after the gain crossover. To eliminate it, you could add a notch filter to the feedback loop and tune its coefficients along with the lowpass coefficient and PI gains using `systune`.

See Also

`systune` | `TuningGoal.Margins` | `TuningGoal.Tracking`

Related Examples

- “Building Tunable Models” on page 17-9
- “Tune Control Systems in Simulink” on page 18-50

Building Tunable Models

This example shows how to create tunable models of control systems for use with `systune` or `looptune`.

Background

You can tune the gains and parameters of your control system with `systune` or `looptune`. To use these commands, you need to construct a tunable model of the control system that identifies and parameterizes its tunable elements. This is done by combining numeric LTI models of the fixed elements with parametric models of the tunable elements.

Using Pre-Defined Tunable Elements

You can use one of the following "parametric" blocks to model commonly encountered tunable elements:

- **tunableGain**: Tunable gain
- **tunablePID**: Tunable PID controller
- **tunablePID2**: Tunable two-degree-of-freedom PID controller
- **tunableTF**: Tunable transfer function
- **tunableSS**: Tunable state-space model.

For example, create a tunable model of the feedforward/feedback configuration of Figure 1 where *C* is a tunable PID controller and *F* is a tunable first-order transfer function.

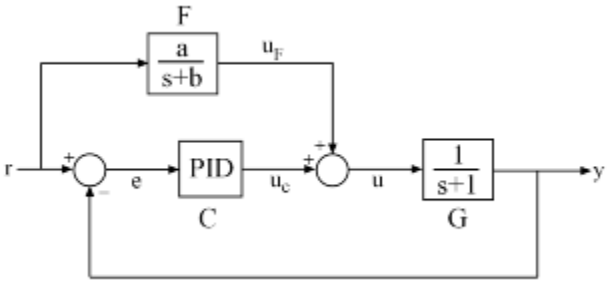


Figure 1: Control System with Feedforward and Feedback Paths

First model each block in the block diagram, using suitable parametric blocks for *C* and *F*.

```
G = tf(1,[1 1]);
C = tunablePID('C','pid');  % tunable PID block
F = tunableTF('F',0,1);     % tunable first-order transfer function
```

Then use `connect` to build a model of the overall block diagram. To specify how the blocks are connected, label the inputs and outputs of each block and model the summing junctions using `sumblk`.

```
G.u = 'u';  G.y = 'y';
C.u = 'e';  C.y = 'uC';
F.u = 'r';  F.y = 'uF';
```

```
% Summing junctions
S1 = sumblk('e = r-y');
S2 = sumblk('u = uF + uC');

T = connect(G,C,F,S1,S2,'r','y')

Generalized continuous-time state-space model with 1 outputs, 1 inputs, 3 states, and the follow
  C: Tunable PID controller, 1 occurrences.
  F: Tunable SISO transfer function, 0 zeros, 1 poles, 1 occurrences.

Type "ss(T)" to see the current value and "T.Blocks" to interact with the blocks.
```

This creates a generalized state-space model `T` of the closed-loop transfer function from `r` to `y`. This model depends on the tunable blocks `C` and `F`. You can use `systune` to automatically tune the PID gains and the feedforward coefficients `a,b` subject to your performance requirements. Use `showTunable` to see the current value of the tunable blocks.

```
showTunable(T)

C =

        1
  Ki * ---
       s

  with Ki = 0.001

Name: C
Continuous-time I-only controller.
-----------------------------------
F =

    10
  ------
  s + 10

Name: F
Continuous-time transfer function.
```

Interacting with the Tunable Parameters

You can adjust the parameterization of the tunable elements C and F by interacting with the objects `C` and `F`. Use `get` to see their list of properties.

```
get(C)

            Kp: [1x1 param.Continuous]
            Ki: [1x1 param.Continuous]
            Kd: [1x1 param.Continuous]
            Tf: [1x1 param.Continuous]
       IFormula: ''
       DFormula: ''
      InputName: {'e'}
      InputUnit: {''}
     InputGroup: [1x1 struct]
     OutputName: {'uC'}
     OutputUnit: {''}
```

```
    OutputGroup: [1x1 struct]
          Notes: [0x1 string]
       UserData: []
           Name: 'C'
             Ts: 0
       TimeUnit: 'seconds'
```

A PID controller has four tunable parameters `Kp,Ki,Kd,Tf`. The tunable block `C` contains a description of each of these parameters. Parameter attributes include current value, minimum and maximum values, and whether the parameter is free or fixed.

```
C.Kp
```

```
ans =

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


1x1 param.Continuous
```

Set the corresponding attributes to override defaults. For example, you can fix the time constant `Tf` to the value 0.1 by

```
C.Tf.Value = 0.1;
C.Tf.Free = false;
```

Creating Custom Tunable Elements

For tunable elements not covered by the pre-defined blocks listed above, you can create your own parameterization in terms of elementary real parameters (`realp`). Consider the low-pass filter

$$F(s) = \frac{a}{s+a}$$

where the coefficient a is tunable. To model this tunable element, create a real parameter a and define F as a transfer function whose numerator and denominator are functions of a. This creates a generalized state-space model `F` of the low-pass filter parameterized by the tunable scalar `a`.

```
a = realp('a',1);   % real tunable parameter, initial value 1
F = tf(a,[1 a])
```

```
Generalized continuous-time state-space model with 1 outputs, 1 inputs, 1 states, and the follow
  a: Scalar parameter, 2 occurrences.

Type "ss(F)" to see the current value and "F.Blocks" to interact with the blocks.
```

Similarly, you can use real parameters to model the notch filter

$$N(s) = \frac{s^2 + 2\zeta_1\omega_n s + \omega_n^2}{s^2 + 2\zeta_2\omega_n s + \omega_n^2}$$

with tunable coefficients $\omega_n, \zeta_1, \zeta_2$.

```
wn = realp('wn',100);
zeta1 = realp('zeta1',1);   zeta1.Maximum = 1;    % zeta1 <= 1
zeta2 = realp('zeta2',1);   zeta2.Maximum = 1;    % zeta2 <= 1
N = tf([1 2*zeta1*wn wn^2],[1 2*zeta2*wn wn^2]);  % tunable notch filter
```

You can also create tunable elements with matrix-valued parameters. For example, model the observer-based controller $C(s)$ with equations

$$\frac{dx}{dt} = Ax + Bu + L(y - Cx), \quad u = -Kx$$

and tunable gain matrices K and L.

```
% Plant with 6 states, 2 controls, 3 measurements
[A,B,C] = ssdata(rss(6,3,2));

K = realp('K',zeros(2,6));
L = realp('L',zeros(6,3));

C = ss(A-B*K-L*C,L,-K,0)
```

```
Generalized continuous-time state-space model with 2 outputs, 3 inputs, 6 states, and the follow
  K: Tunable 2x6 matrix, 2 occurrences.
  L: Tunable 6x3 matrix, 2 occurrences.

Type "ss(C)" to see the current value and "C.Blocks" to interact with the blocks.
```

Enabling Open-Loop Requirements

The `systune` command takes a closed-loop model of the overall control system, like the tunable model `T` built at the beginning of this example. Such models do not readily support open-loop analysis or open-loop specifications such as loop shapes and stability margins. To gain access to open-loop responses, insert an `AnalysisPoint` block as shown in Figure 2.

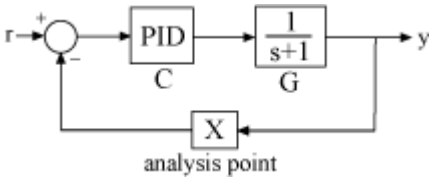


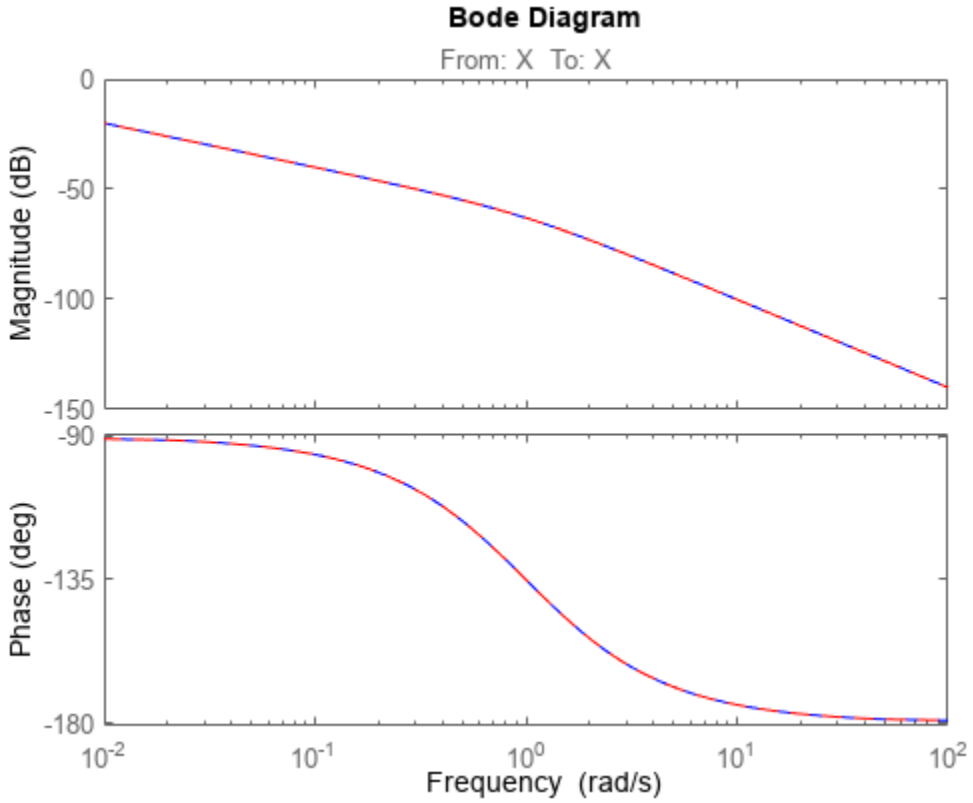
Figure 2: Analysis Point Block

The `AnalysisPoint` block can be used to mark internal signals of interest as well as locations where to open feedback loops and measure open-loop responses. This block evaluates to a unit gain and has no impact on the model responses. For example, construct a closed-loop model `T` of the feedback loop of Figure 2 where C is a tunable PID.

```
G = tf(1,[1 1]);
C = tunablePID('C','pid');
AP = AnalysisPoint('X');
T = feedback(G*C,AP);
```

You can now use `getLoopTransfer` to compute the (negative-feedback) loop transfer function measured at the location "X". Note that this loop transfer function is $L = GC$ for the feedback loop of Figure 2.

```
L = getLoopTransfer(T,'X',-1);  % loop transfer at "X"
clf, bode(L,'b',G*C,'r--')
```



You can also refer to the location "X" when specifying target loop shapes or stability margins for `systune`. The requirement then applies to the loop transfer measured at this location.

```
% Target loop shape for loop transfer at "X"
Req1 = TuningGoal.LoopShape('X',tf(5,[1 0]));

% Target stability margins for loop transfer at "X"
Req2 = TuningGoal.Margins('X',6,40);
```

In general, loop opening locations are specified in the `Location` property of `AnalysisPoint` blocks. For single-channel analysis points, the block name is used as default location name. For multi-channel analysis points, indices are appended to the block name to form the default location names.

```
AP = AnalysisPoint('Y',2);  % two-channel analysis point
AP.Location
```

```
ans = 2x1 cell
    {'Y(1)'}
    {'Y(2)'}
```

You can override the default location names and use more descriptive names by modifying the `Location` property.

```
% Rename loop opening locations to "InnerLoop" and "OuterLoop".
AP.Location = {'InnerLoop' ; 'OuterLoop'};
AP.Location
```

```
ans = 2x1 cell
    {'InnerLoop'}
    {'OuterLoop'}
```

See Also

`AnalysisPoint`

Related Examples

- “Generalized Models” on page 1-12
- “Models with Tunable Coefficients” on page 1-15
- “Mark Signals of Interest for Control System Analysis and Design” (Simulink Control Design)

Active Vibration Control in Three-Story Building

This example uses `systune` to control seismic vibrations in a three-story building.

Background

This example considers an Active Mass Driver (AMD) control system for vibration isolation in a three-story experimental structure. This setup is used to assess control design techniques for increasing safety of civil engineering structures during earthquakes. The structure consists of three stories with an active mass driver on the top floor which is used to attenuate ground disturbances. This application is borrowed from "*Benchmark Problems in Structural Control: Part I - Active Mass Driver System,*" B.F. Spencer Jr., S.J. Dyke, and H.S. Deoskar, Earthquake Engineering and Structural Dynamics, 27(11), 1998, pp. 1127-1139.

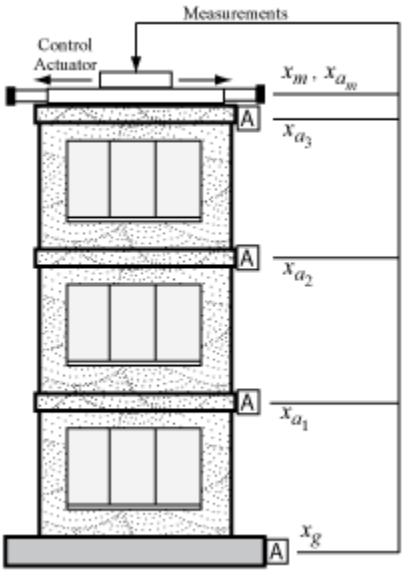


Figure 1: Active Mass Driver Control System

The plant `P` is a 28-state model with the following state variables:

- `x(i)`: displacement of i-th floor relative to the ground (cm)
- `xm`: displacement of AMD relative to 3rd floor (cm)
- `xv(i)`: velocity of i-th floor relative to the ground (cm/s)
- `xvm`: velocity of AMD relative to the ground (cm/s)
- `xa(i)`: acceleration of i-th floor relative to the ground (g)
- `xam`: acceleration of AMD relative to the ground (g)
- `d(1)=x(1)`, `d(2)=x(2)-x(1)`, `d(3)=x(3)-x(2)`: inter-story drifts

The inputs are the ground acceleration `xag` (in g) and the control signal `u`. We use 1 g = 981 cm/s^2.

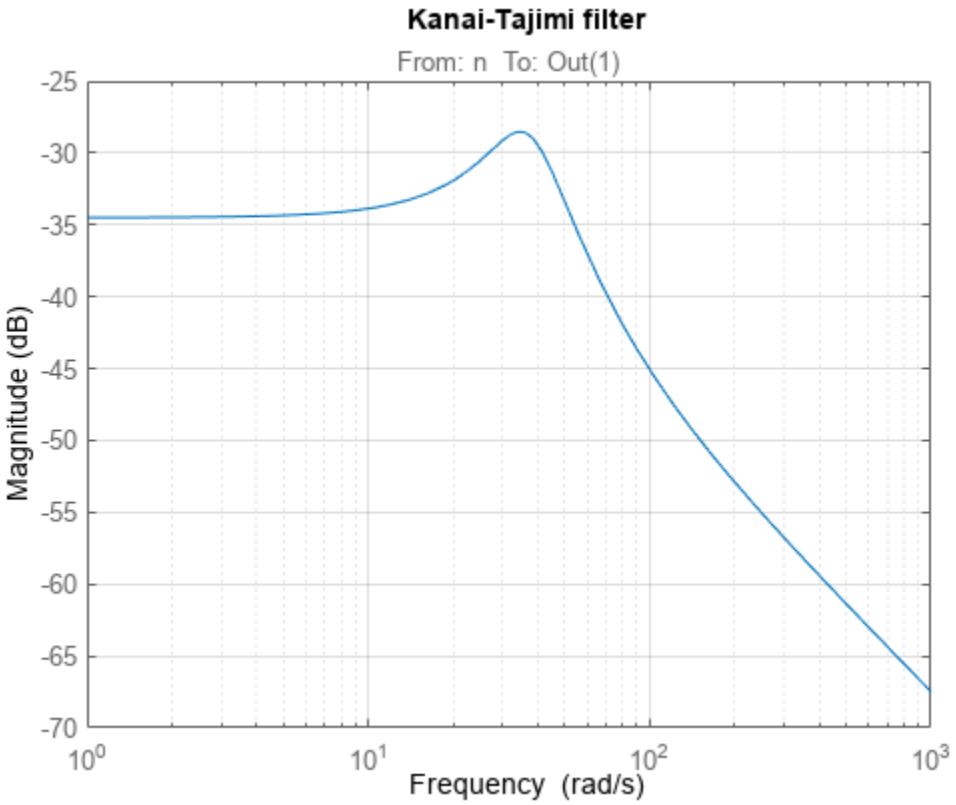
```
load ThreeStoryData
size(P)
```

```
State-space model with 20 outputs, 2 inputs, and 28 states.
```

Model of Earthquake Acceleration

The earthquake acceleration is modeled as a white noise process filtered through a Kanai-Tajimi filter.

```
zg = 0.3;
wg = 37.3;
S0 = 0.03*zg/(pi*wg*(4*zg^2+1));
Numerator = sqrt(S0)*[2*zg*wg wg^2];
Denominator = [1 2*zg*wg wg^2];

F = sqrt(2*pi)*tf(Numerator,Denominator);
F.InputName = 'n';   % white noise input

bodemag(F)
grid
title('Kanai-Tajimi filter')
```

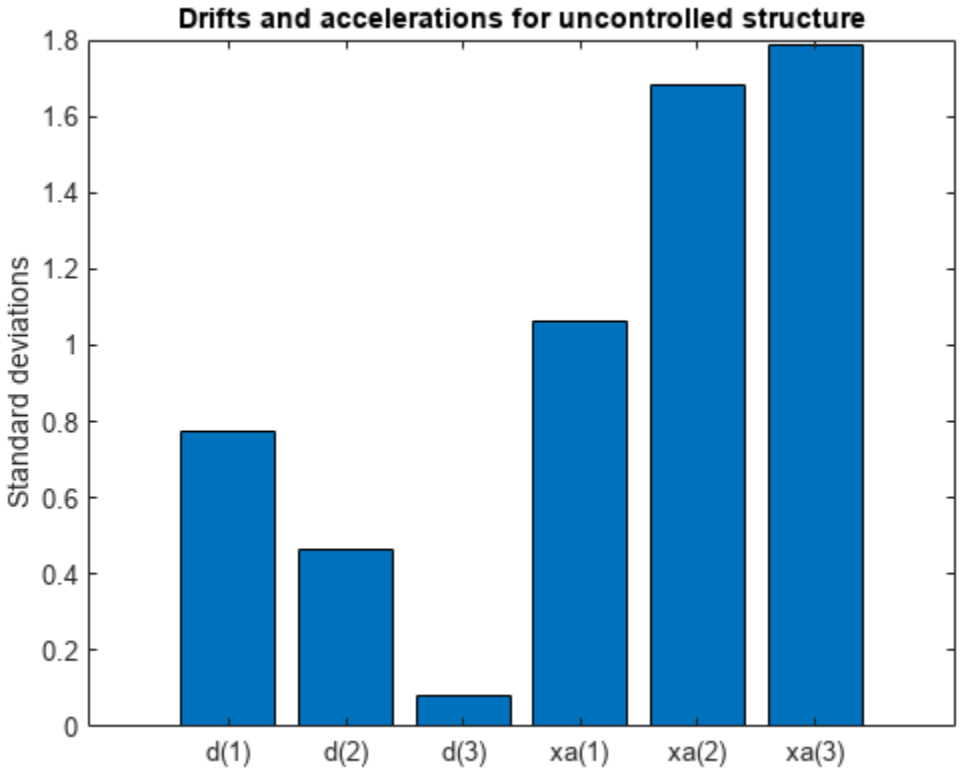


Open-Loop Characteristics

The effect of an earthquake on the uncontrolled structure can be simulated by injecting a white noise input `n` into the plant-filter combination. You can also use `covar` to directly compute the standard deviations of the resulting inter-story drifts and accelerations.

```
% Add Kanai-Tajimi filter to the plant
PF = P*append(F,1);
```

```
% Standard deviations of open-loop drifts
CV = covar(PF('d','n'),1);
d0 = sqrt(diag(CV));

% Standard deviations of open-loop acceleration
CV = covar(PF('xa','n'),1);
xa0 = sqrt(diag(CV));

% Plot open-loop RMS values
clf
bar([d0; xa0])
title('Drifts and accelerations for uncontrolled structure')
ylabel('Standard deviations')
set(gca,'XTickLabel',{'d(1)','d(2)','d(3)','xa(1)','xa(2)','xa(3)'})
```



Control Structure and Design Requirements

The control structure is depicted in Figure 2.

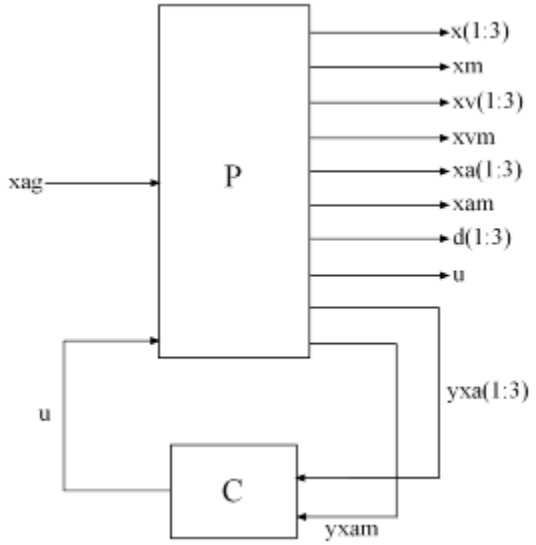


Figure 2: Control Structure

The controller *C* uses measurements `yxa` and `yxam` of `xa` and `xam` to generate the control signal `u`. Physically, the control `u` is an electrical current driving an hydraulic actuator that moves the masses of the AMD. The design requirements involve:

- Minimization of the inter-story drifts `d(i)` and accelerations `xa(i)`
- Hard constraints on control effort in terms of mass displacement `xm`, mass acceleration `xam`, and control effort `u`

All design requirements are assessed in terms of standard deviations of the corresponding signals. Use `TuningGoal.Variance` to express these requirements and scale each variable by its open-loop standard deviation to seek uniform relative improvement in all variables.

```
% Soft requirements on drifts and accelerations
Soft = [...
   TuningGoal.Variance('n','d(1)',d0(1)) ; ...
   TuningGoal.Variance('n','d(2)',d0(2)) ; ...
   TuningGoal.Variance('n','d(3)',d0(3)) ; ...
   TuningGoal.Variance('n','xa(1)',xa0(1)) ; ...
   TuningGoal.Variance('n','xa(2)',xa0(2)) ; ...
   TuningGoal.Variance('n','xa(3)',xa0(3))];

% Hard requirements on control effort
Hard = [...
   TuningGoal.Variance('n','xm',3) ; ...
   TuningGoal.Variance('n','xam',2) ; ...
   TuningGoal.Variance('n','u',1)];
```

Controller Tuning

`systune` lets you tune virtually any controller structure subject to these requirements. The controller complexity can be adjusted by trial-and-error, starting with sufficiently high order to gauge the limits of performance, then reducing the order until you observe a noticeable performance degradation. For this example, start with a 5th-order controller with no feedthrough term.

```
C = tunableSS('C',5,1,4);
C.D.Value = 0;
C.D.Free = false;   % Fix feedthrough to zero
```

Construct a tunable model `T0` of the closed-loop system of Figure 2 and tune the controller parameters with `systune`.

```
% Build tunable closed-loop model
T0 = lft(PF,C);

% Tune controller parameters
[T,fSoft,gHard]  = systune(T0,Soft,Hard);
```

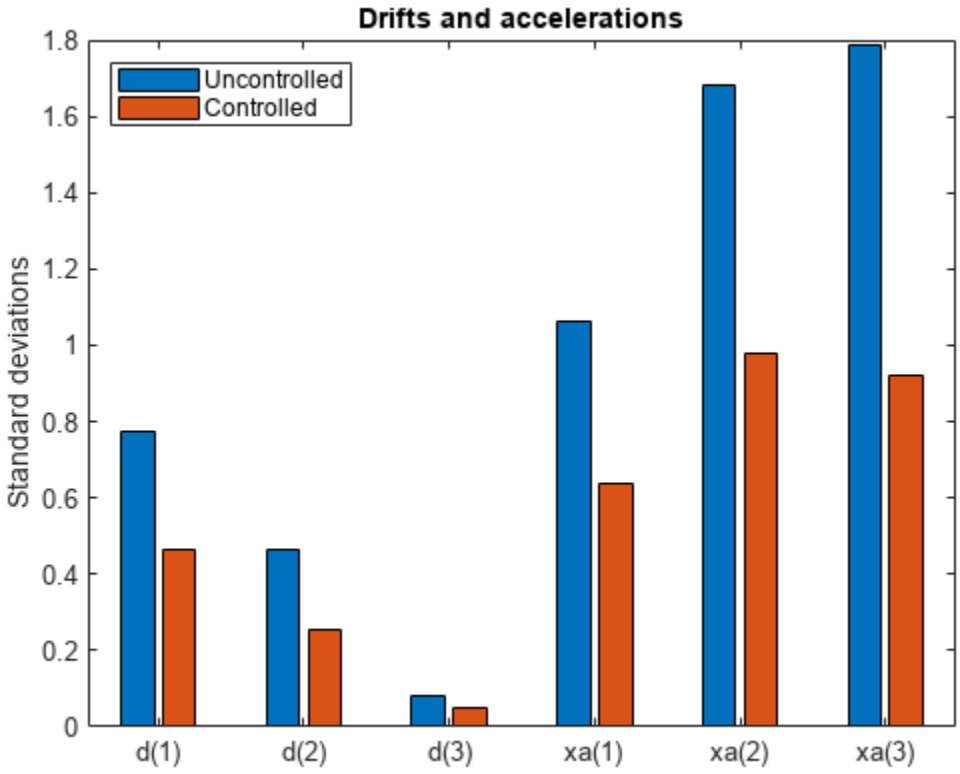
```
Final: Soft = 0.601, Hard = 0.99975, Iterations = 201
```

The summary indicates that we achieved an overall reduction of 40% in standard deviations (`Soft = 0.6`) while meeting all hard constraints (`Hard < 1`).

Validation

Compute the standard deviations of the drifts and accelerations for the controlled structure and compare with the uncontrolled results. The AMD control system yields significant reduction of both drift and acceleration.

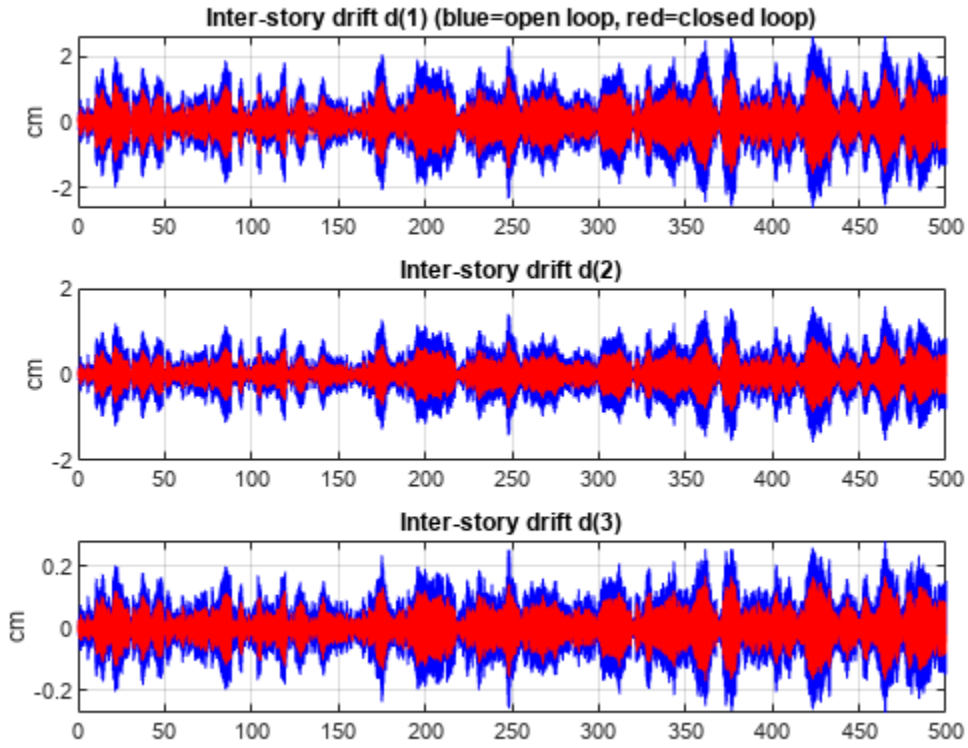
```
% Standard deviations of closed-loop drifts
CV = covar(T('d','n'),1);
d = sqrt(diag(CV));

% Standard deviations of closed-loop acceleration
CV = covar(T('xa','n'),1);
xa = sqrt(diag(CV));

% Compare open- and closed-loop values
clf
bar([d0 d; xa0 xa])
title('Drifts and accelerations')
ylabel('Standard deviations')
set(gca,'XTickLabel',{'d(1)','d(2)','d(3)','xa(1)','xa(2)','xa(3)'})
legend('Uncontrolled','Controlled','location','NorthWest')
```



Simulate the response of the 3-story structure to an earthquake-like excitation in both open and closed loop. The earthquake acceleration is modeled as a white noise process colored by the Kanai-Tajimi filter.

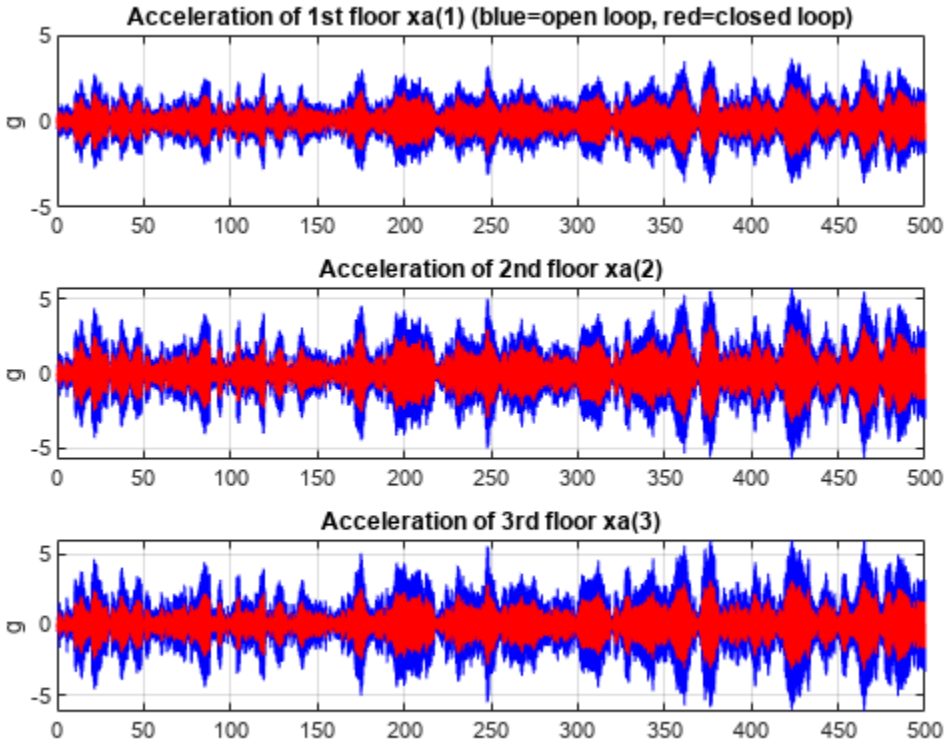
```
% Sampled white noise process
rng('default')
dt = 1e-3;
t = 0:dt:500;
n = randn(1,length(t))/sqrt(dt); % white noise signal

% Open-loop simulation
ysimOL = lsim(PF(:,1), n , t);

% Closed-loop simulation
ysimCL = lsim(T, n , t);

% Drifts
clf
subplot(3,1,1)
plot(t,ysimOL(:,13),'b',t,ysimCL(:,13),'r')
grid
title('Inter-story drift d(1) (blue=open loop, red=closed loop)')
ylabel('cm')
subplot(3,1,2)
plot(t,ysimOL(:,14),'b',t,ysimCL(:,14),'r')
grid
title('Inter-story drift d(2)')
```

```
ylabel('cm')
subplot(3,1,3)
plot(t,ysimOL(:,15),'b',t,ysimCL(:,15),'r')
grid
title('Inter-story drift d(3)')
ylabel('cm')
```



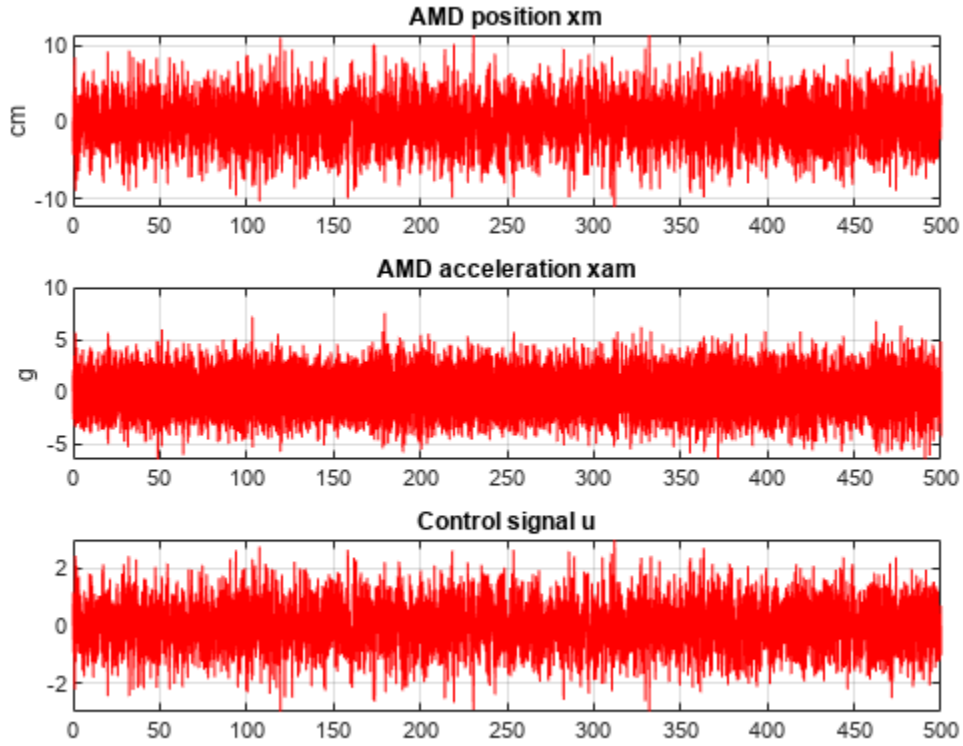
Accelerations

```
clf
subplot(3,1,1)
plot(t,ysimOL(:,9),'b',t,ysimCL(:,9),'r')
grid
title('Acceleration of 1st floor xa(1) (blue=open loop, red=closed loop)')
ylabel('g')
subplot(3,1,2)
plot(t,ysimOL(:,10),'b',t,ysimCL(:,10),'r')
grid
title('Acceleration of 2nd floor xa(2)')
ylabel('g')
subplot(3,1,3)
plot(t,ysimOL(:,11),'b',t,ysimCL(:,11),'r')
grid
title('Acceleration of 3rd floor xa(3)')
ylabel('g')
```



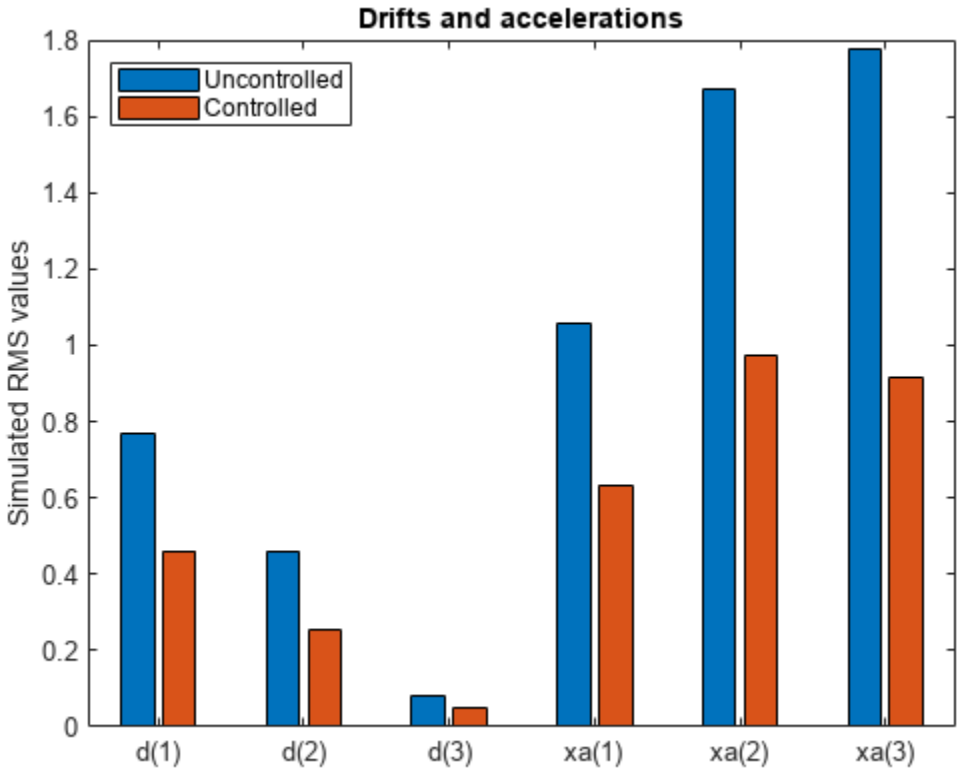
Control variables

```
clf
subplot(3,1,1)
plot(t,ysimCL(:,4),'r')
grid
title('AMD position xm')
ylabel('cm')
subplot(3,1,2)
plot(t,ysimCL(:,12),'r')
grid
title('AMD acceleration xam')
ylabel('g')
subplot(3,1,3)
plot(t,ysimCL(:,16),'r')
grid
title('Control signal u')
```



Plot the root-mean-square (RMS) of the simulated signals for both the controlled and uncontrolled scenarios. Assuming ergodicity, the RMS performance can be estimated from a single sufficiently long simulation of the process and coincides with the standard deviations computed earlier. Indeed the RMS plot closely matches the standard deviation plot obtained earlier.

```
clf
bar([std(ysimOL(:,13:15)) std(ysimOL(:,9:11)) ; ...
   std(ysimCL(:,13:15)) std(ysimCL(:,9:11))]')
title('Drifts and accelerations')
ylabel('Simulated RMS values')
set(gca,'XTickLabel',{'d(1)','d(2)','d(3)','xa(1)','xa(2)','xa(3)'})
legend('Uncontrolled','Controlled','location','NorthWest')
```



Overall, the controller achieves significant reduction of ground vibration both in terms of drift and acceleration for all stories while meeting the hard constraints on control effort and mass displacement.

See Also

`systune` | `isPassive` | `TuningGoal.Variance`

Related Examples

- “Vibration Control in Flexible Beam” on page 17-25

Vibration Control in Flexible Beam

This example shows how to tune a controller for reducing vibrations in a flexible beam.

Model of Flexible Beam

Figure 1 depicts an active vibration control system for a flexible beam.

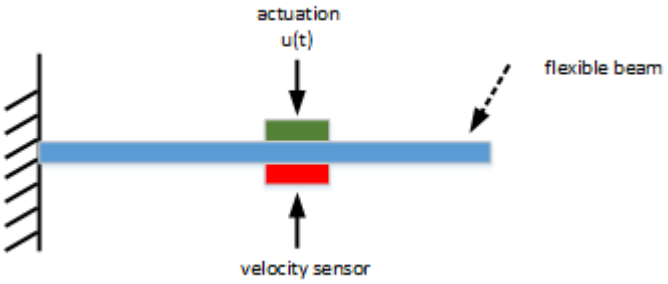


Figure 1: Active control of flexible beam

In this setup, the actuator delivering the force u and the velocity sensor are collocated. We can model the transfer function from control input u to the velocity y using finite-element analysis. Keeping only the first six modes, we obtain a plant model of the form

$$G(s) = \sum_{i=1}^{6} \frac{\alpha_i^2 s}{s^2 + 2\xi w_i s + w_i^2}$$

with the following parameter values.

```
% Parameters
xi = 0.05;
alpha = [0.09877, -0.309, -0.891, 0.5878, 0.7071, -0.8091];
w = [1, 4, 9, 16, 25, 36];
```

The resulting beam model for $G(s)$ is given by

```
% Beam model
G = tf(alpha(1)^2*[1,0],[1, 2*xi*w(1), w(1)^2]) + ...
    tf(alpha(2)^2*[1,0],[1, 2*xi*w(2), w(2)^2]) + ...
    tf(alpha(3)^2*[1,0],[1, 2*xi*w(3), w(3)^2]) + ...
    tf(alpha(4)^2*[1,0],[1, 2*xi*w(4), w(4)^2]) + ...
    tf(alpha(5)^2*[1,0],[1, 2*xi*w(5), w(5)^2]) + ...
    tf(alpha(6)^2*[1,0],[1, 2*xi*w(6), w(6)^2]);

G.InputName = 'uG';  G.OutputName = 'y';
```

With this sensor/actuator configuration, the beam is a passive system:

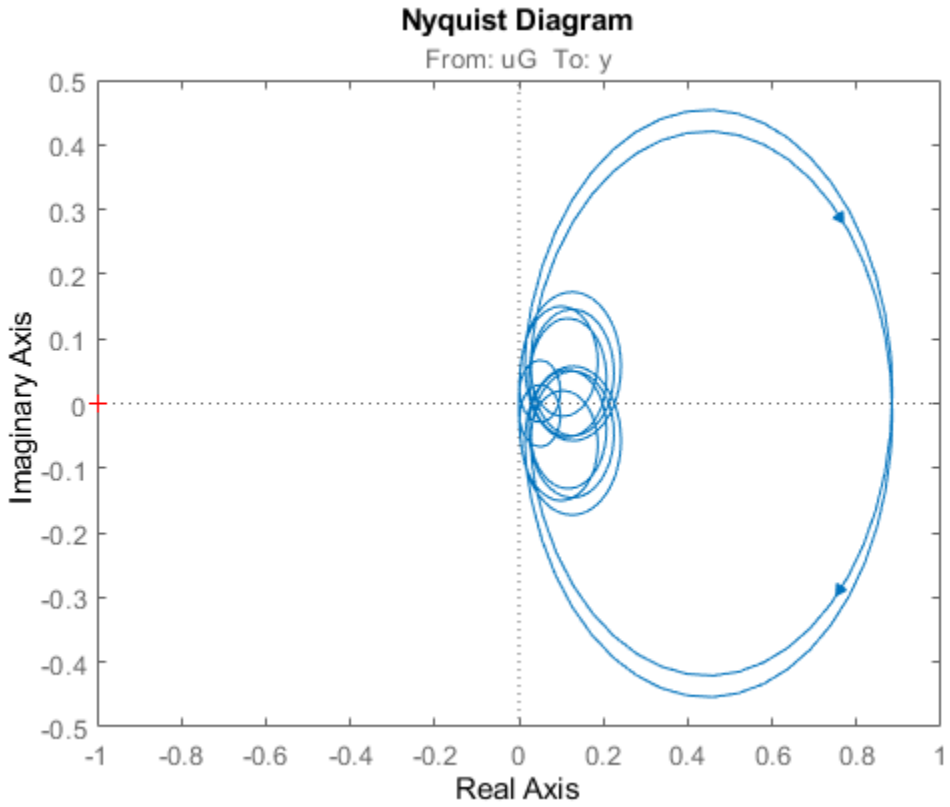
```
isPassive(G)
```

```
ans =
```

```
logical

 1
```

This is confirmed by observing that the Nyquist plot of G is positive real.

```
nyquist(G)
```



LQG Controller

LQG control is a natural formulation for active vibration control. The LQG control setup is depicted in Figure 2. The signals d and n are the process and measurement noise, respectively.

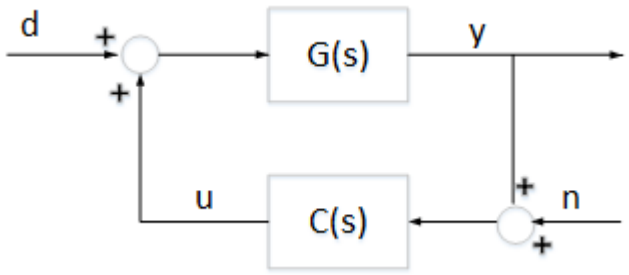


Figure 2: LQG control structure

First use `lqg` to compute the optimal LQG controller for the objective

$$J = \lim_{T\to\infty} E\left(\int_0^T (y^2(t) + 0.001u^2(t))dt\right)$$

with noise variances:

$$E(d^2(t)) = 1, \quad E(n^2(t)) = 0.01.$$

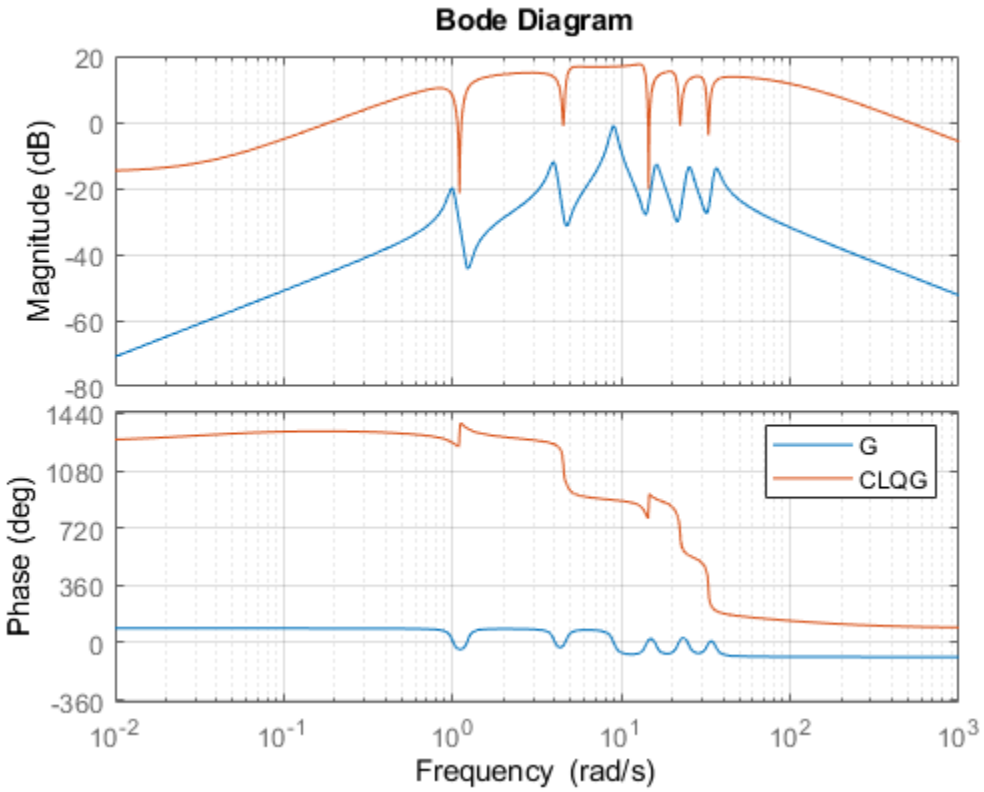
```
[a,b,c,d] = ssdata(G);
M = [c d;zeros(1,12) 1];  % [y;u] = M * [x;u]
QWV = blkdiag(b*b',1e-2);
QXU = M'*diag([1 1e-3])*M;
CLQG = lqg(ss(G),QXU,QWV);
```

The LQG-optimal controller `CLQG` is complex with 12 states and several notching zeros.

```
size(CLQG)
```

```
State-space model with 1 outputs, 1 inputs, and 12 states.
```

```
bode(G,CLQG,{1e-2,1e3}), grid, legend('G','CLQG')
```



Use the general-purpose tuner `systune` to try and simplify this controller. With `systune`, you are not limited to a full-order controller and can tune controllers of any order. Here for example, let's tune a 2nd-order state-space controller.

```
C = ltiblock.ss('C',2,1,1);
```

Build a closed-loop model of the block diagram in Figure 2.

```
C.InputName = 'yn';  C.OutputName = 'u';
S1 = sumblk('yn = y + n');
S2 = sumblk('uG = u + d');
CL0 = connect(G,C,S1,S2,{'d','n'},{'y','u'},{'yn','u'});
```

Use the LQG criterion J above as sole tuning goal. The LQG tuning goal lets you directly specify the performance weights and noise covariances.

```
R1 = TuningGoal.LQG({'d','n'},{'y','u'},diag([1,1e-2]),diag([1 1e-3]));
```

Now tune the controller C to minimize the LQG objective J.

```
[CL1,J1] = systune(CL0,R1);
```

```
Final: Soft = 0.478, Hard = -Inf, Iterations = 40
```

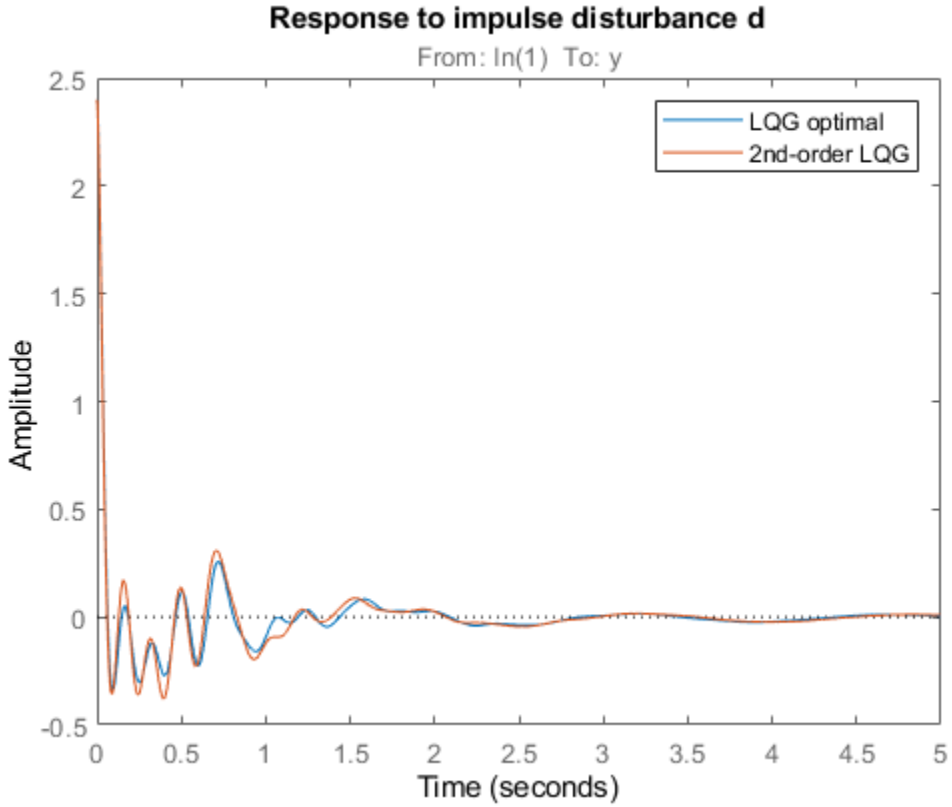
The optimizer found a 2nd-order controller with $J = 0.478$. Compare with the optimal J value for CLQG:

```
[~,Jopt] = evalGoal(R1,replaceBlock(CL0,'C',CLQG))
```

```
Jopt =

    0.4673
```

The performance degradation is less than 5%, and we reduced the controller complexity from 12 to 2 states. Further compare the impulse responses from d to y for the two controllers. The two responses are almost identical. You can therefore obtain near-optimal vibration attenuation with a simple second-order controller.

```
T0 = feedback(G,CLQG,+1);
T1 = getIOTransfer(CL1,'d','y');
impulse(T0,T1,5)
title('Response to impulse disturbance d')
legend('LQG optimal','2nd-order LQG')
```



Passive LQG Controller

We used an approximate model of the beam to design these two controllers. A priori, there is no guarantee that these controllers will perform well on the real beam. However, we know that the beam is a passive physical system and that the negative feedback interconnection of passive systems is always stable. So if $-C(s)$ is passive, we can be confident that the closed-loop system will be stable.

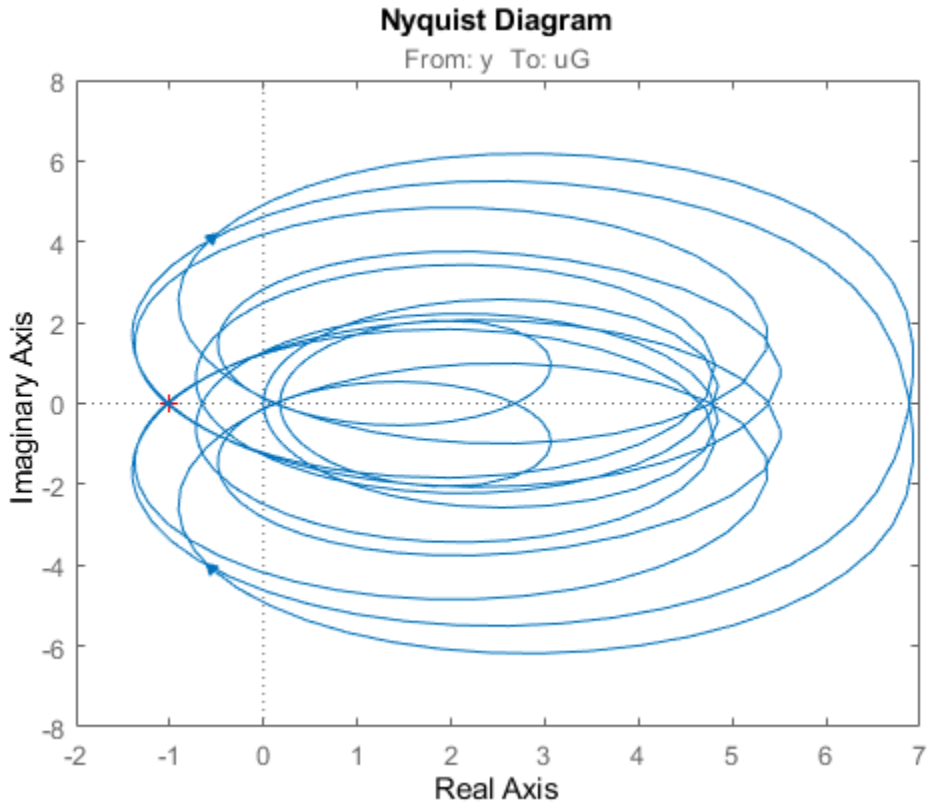
The optimal LQG controller is not passive. In fact, its relative passive index is infinite because $1 - CLQG$ is not even minimum phase.

```
getPassiveIndex(-CLQG)
```

```
ans =

   Inf
```

This is confirmed by its Nyquist plot.

```
nyquist(-CLQG)
```



Using `systune`, you can re-tune the second-order controller with the additional requirement that $-C(s)$ should be passive. To do this, create a passivity tuning goal for the open-loop transfer function from `yn` to `u` (which is $C(s)$). Use the "WeightedPassivity" goal to account for the minus sign.

```
R2 = TuningGoal.WeightedPassivity({'yn'},{'u'},-1,1);
R2.Openings = 'u';
```

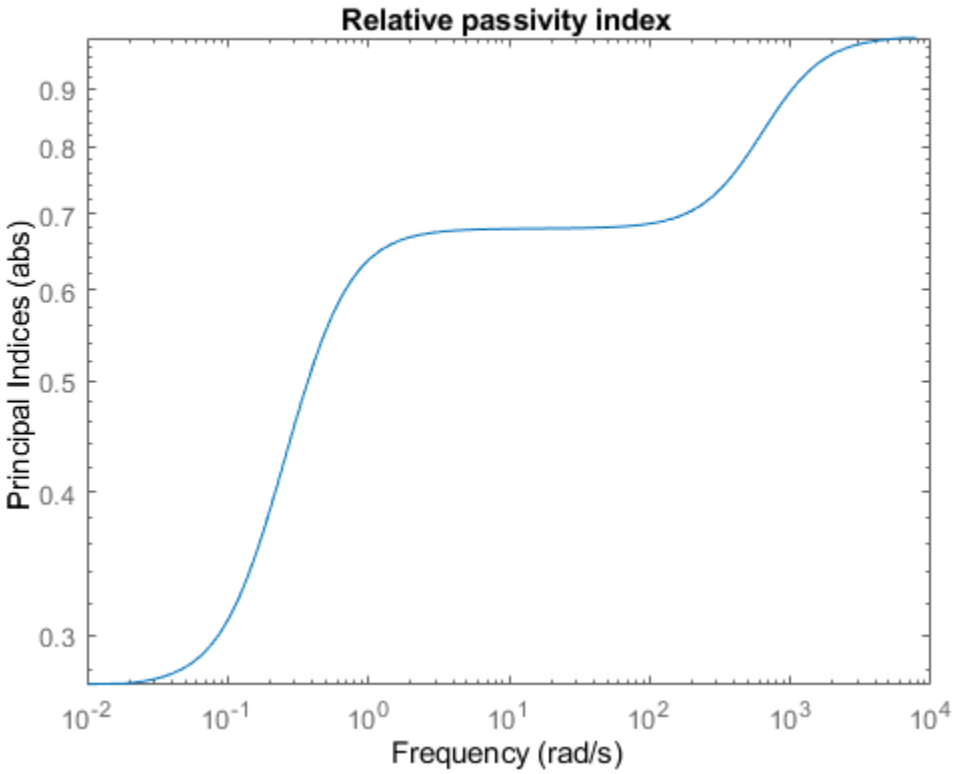
Now re-tune the closed-loop model `CL1` to minimize the LQG objective J subject to $-C(s)$ being passive. Note that the passivity goal `R2` is now specified as a hard constraint.

```
[CL2,J2,g] = systune(CL1,R1,R2);
```

```
Final: Soft = 0.478, Hard = 1, Iterations = 70
```

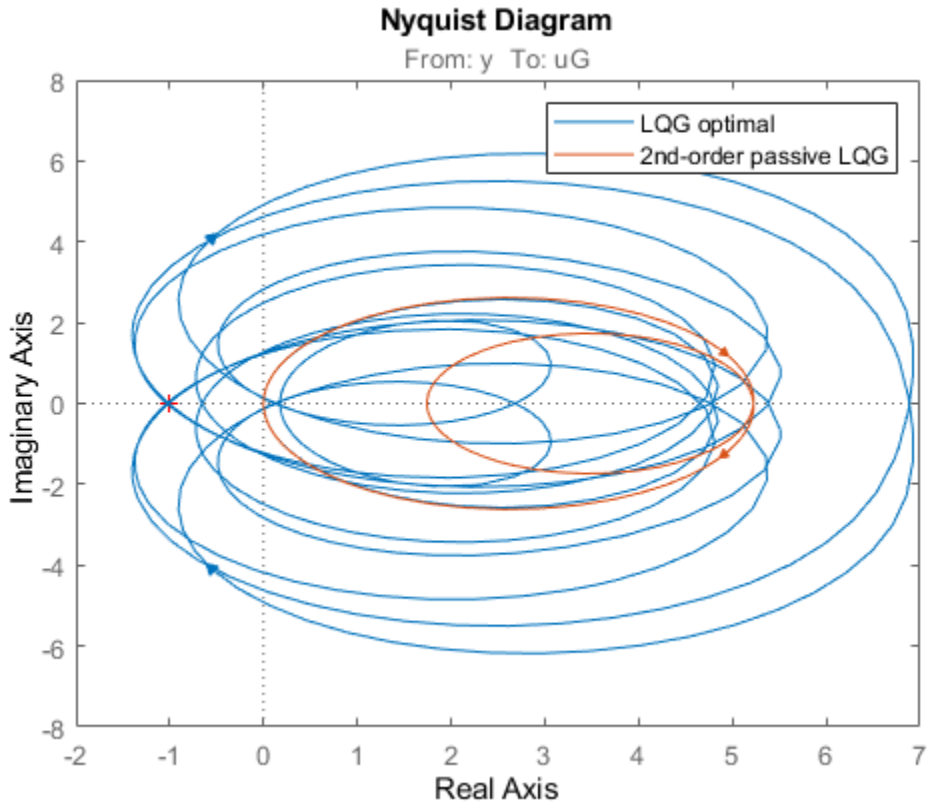
The tuner achieves the same J value as previously, while enforcing passivity (hard constraint less than 1). Verify that $-C(s)$ is passive.

```
C2 = getBlockValue(CL2,'C');
passiveplot(-C2)
```



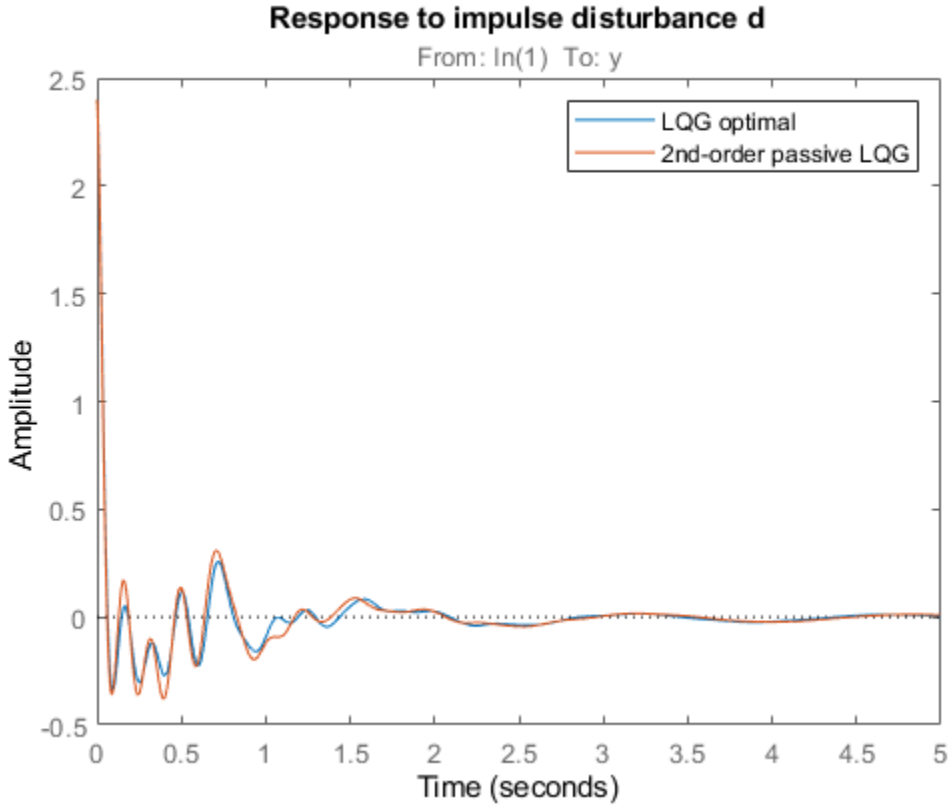
The improvement over the LQG-optimal controller is most visible in the Nyquist plot.

```
nyquist(-CLQG,-C2)
legend('LQG optimal','2nd-order passive LQG')
```



Finally, compare the impulse responses from d to y.

```
T2 = getIOTransfer(CL2,'d','y');
impulse(T0,T2,5)
title('Response to impulse disturbance d')
legend('LQG optimal','2nd-order passive LQG')
```



Using `systune`, you designed a second-order passive controller with near-optimal LQG performance.

See Also

`systune` | `TuningGoal.WeightedPassivity` | `TuningGoal.LQG`

Related Examples

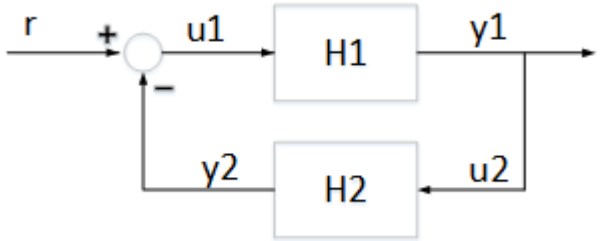
- "About Passivity and Passivity Indices" on page 10-2
- "Passive Control of Water Tank Level" on page 18-196
- "Passive Control with Communication Delays" on page 17-34

Passive Control with Communication Delays

This example shows how to mitigate communication delays in a passive control system.

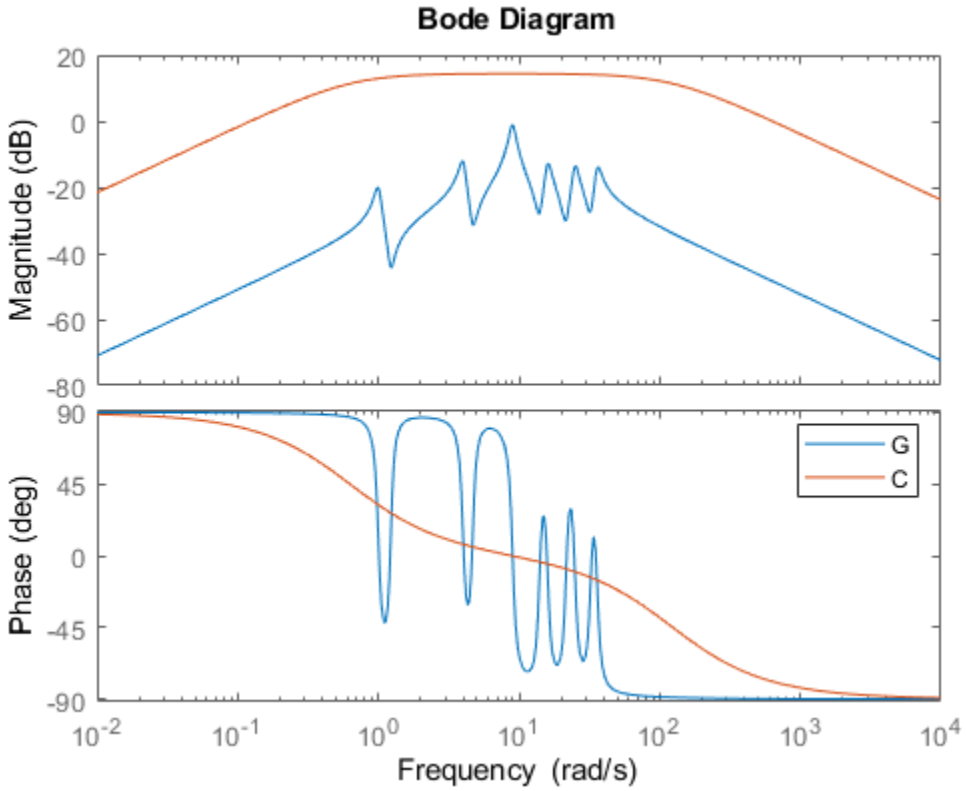
Passivity-Based Control

By the Passivity Theorem, the negative-feedback interconnection of two strictly passive systems H_1 and H_2 is always stable.

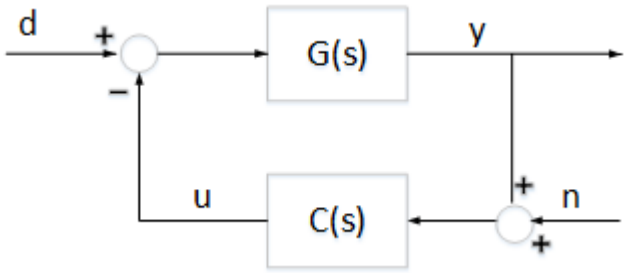


When the physical plant is passive, it is therefore advantageous to use a passive controller for robustness and safety reasons. In networked control systems, however, communication delays can undo the benefits of passivity-based control and lead to instability. To illustrate this point, we use the plant and 2nd-order passive controller from the "Vibration Control in Flexible Beam" example. See this example for background on the underlying control problem. Load the plant model G and passive controller C (note that C corresponds to $-C$ in the other example).

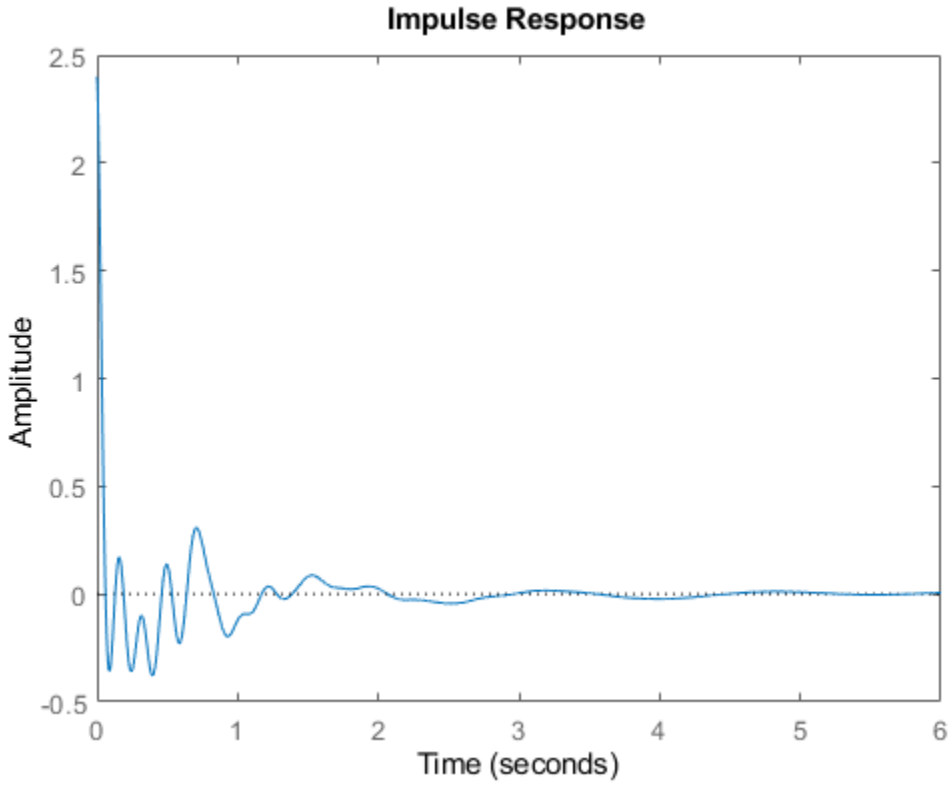
```
load BeamControl G C

bode(G,C,{1e-2,1e4})
legend('G','C')
```



The control configuration is shown below as well as the impulse response from d to y.



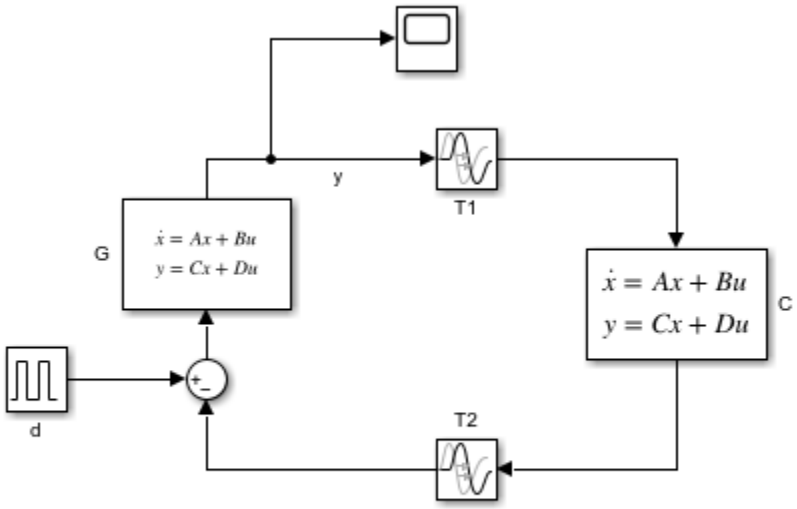

```
impulse(feedback(G,C))
```



Destabilizing Effect of Communication Delays

Now suppose there are substantial communication delays between the sensor and the controller, and between the controller and the actuator. This situation is modeled in Simulink as follows.

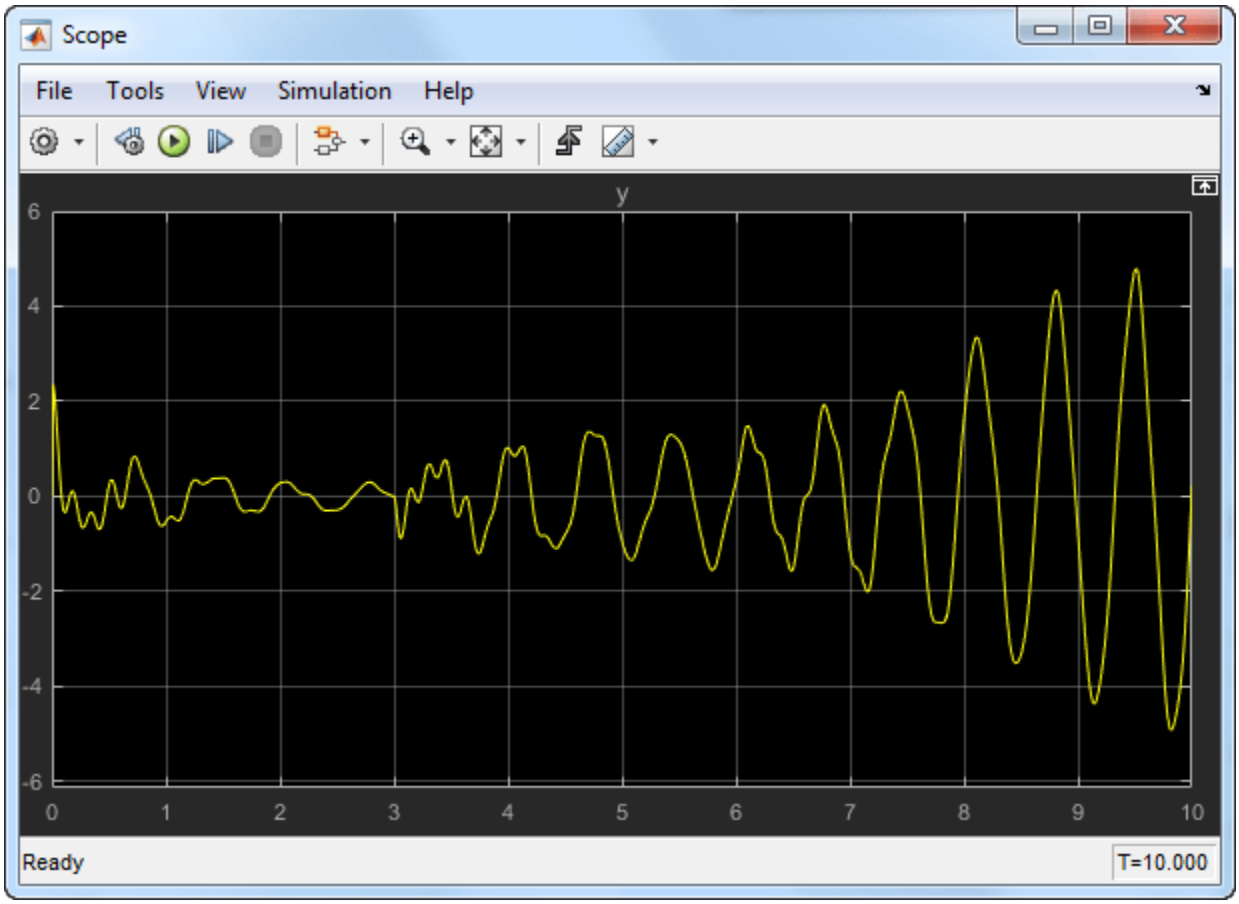
```
open_system('DelayedFeedback')
```



The communication delays are set to

```
T1 = 1;
T2 = 2;
```

Simulating this model shows that the communication delays destabilize the feedback loop.



Scattering Transformation

To mitigate the delay effects, you can use a simple linear transformation of the signals exchanged between the plant and controller over the network.

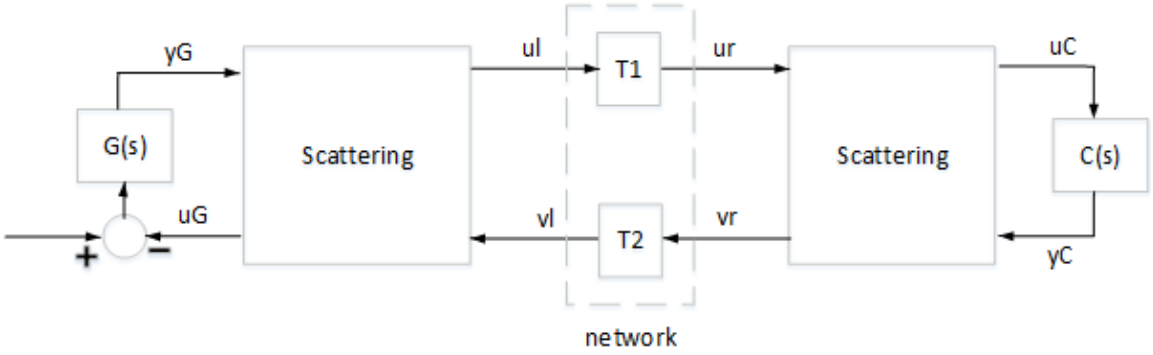


Figure 1: Networked Control System

This is called the "scattering transformation" and given by the formulas

$$\begin{pmatrix} u_l \\ v_l \end{pmatrix} = \begin{pmatrix} 1 & b \\ 1 & -b \end{pmatrix} \begin{pmatrix} u_G \\ y_G \end{pmatrix}, \quad \begin{pmatrix} u_r \\ v_r \end{pmatrix} = \begin{pmatrix} 1 & b \\ 1 & -b \end{pmatrix} \begin{pmatrix} y_C \\ u_C \end{pmatrix},$$

or equivalently

$$\begin{pmatrix} u_l \\ u_G \end{pmatrix} = S \begin{pmatrix} v_l \\ y_G \end{pmatrix}, \quad \begin{pmatrix} v_r \\ u_C \end{pmatrix} = S^{-1} \begin{pmatrix} u_r \\ y_C \end{pmatrix}, \quad S = \begin{pmatrix} 1 & 2b \\ 1 & b \end{pmatrix}$$

with $b > 0$. Note that in the absence of delays, the two scattering transformations cancel each other and the block diagram in Figure 1 is equivalent to the negative feedback interconnection of G and C.

When delays are present, however, (u_l, v_l) is no longer equal to (u_r, v_r) and this scattering transformation alters the properties of the closed-loop system. In fact, observing that

$$u_l = (1 - bC(s))/(1 + bC(s))v_l, \quad v_r = (G(s)/b - 1)/(G(s)/b + 1)u_r$$

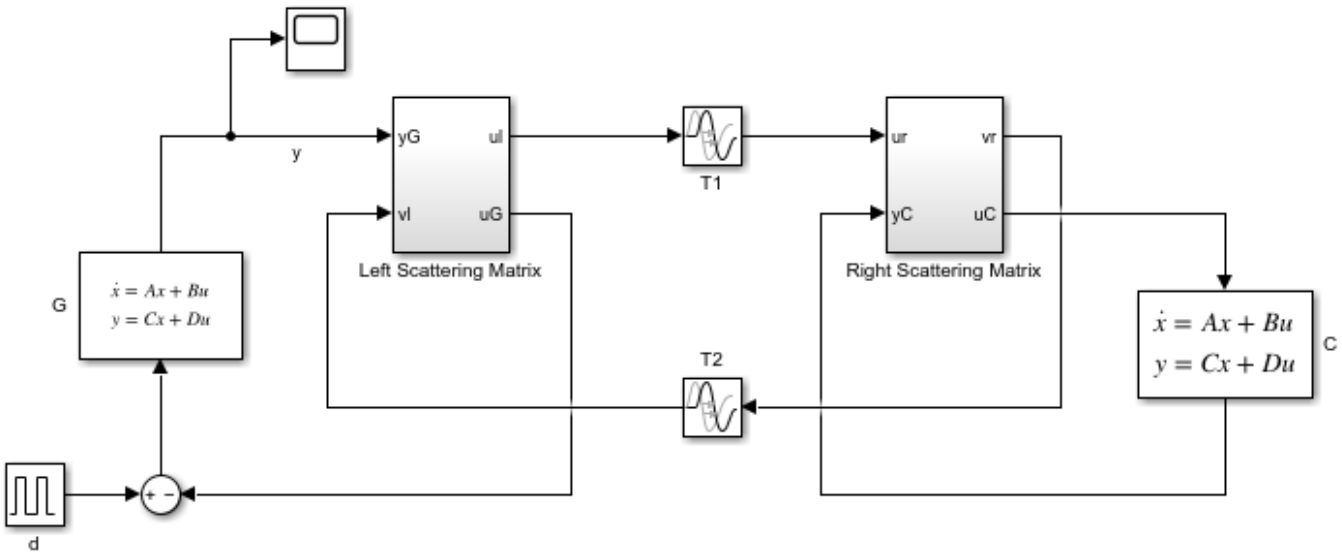
and that bC and G/b strictly passive ensures that

$$\|(1 - bC)/(1 + bC)\|_\infty < 1, \quad \|(G/b - 1)/(G/b + 1)\|_\infty < 1,$$

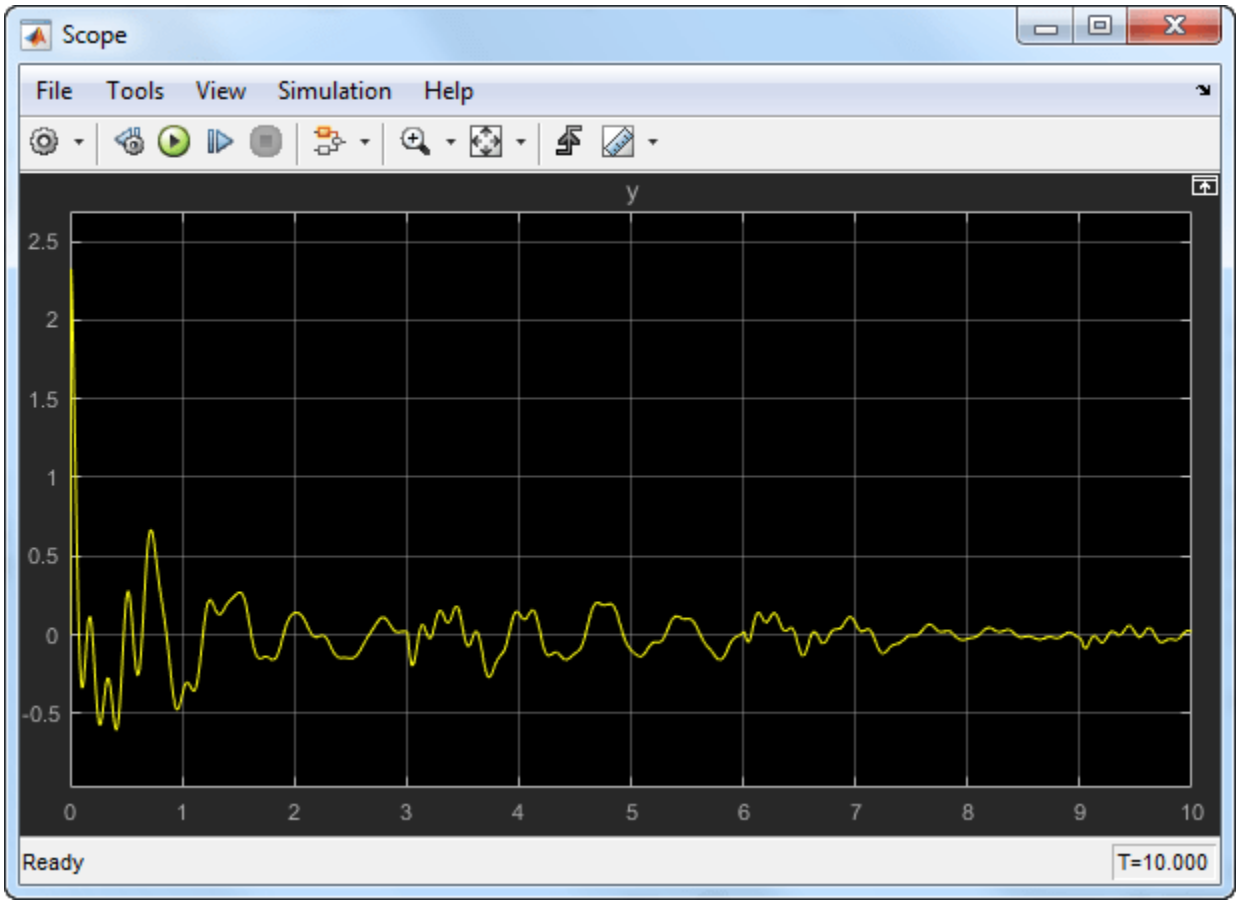
the Small Gain Theorem guarantees that the feedback interconnection of Figure 1 is always stable no matter how large the delays. Confirm this by building a Simulink model of the block diagram in Figure 1 for the value $b = 1$.

```
b = 1;
```

```
open_system('ScatteringTransformation')
```



Simulate the impulse response of the closed-loop system as done before. The response is now stable and similar to the delay-free response in spite of the large delays.



For more details on the scattering transformation, see T. Matiakis, S. Hirche, and M. Buss, "Independent-of-Delay Stability of Nonlinear Networked Control Systems by Scattering Transformation," Proceedings of the 2006 American Control Conference, 2006, pp. 2801-2806.

See Also

`getPassiveIndex` | `isPassive`

Related Examples

- “Vibration Control in Flexible Beam” on page 17-25

More About

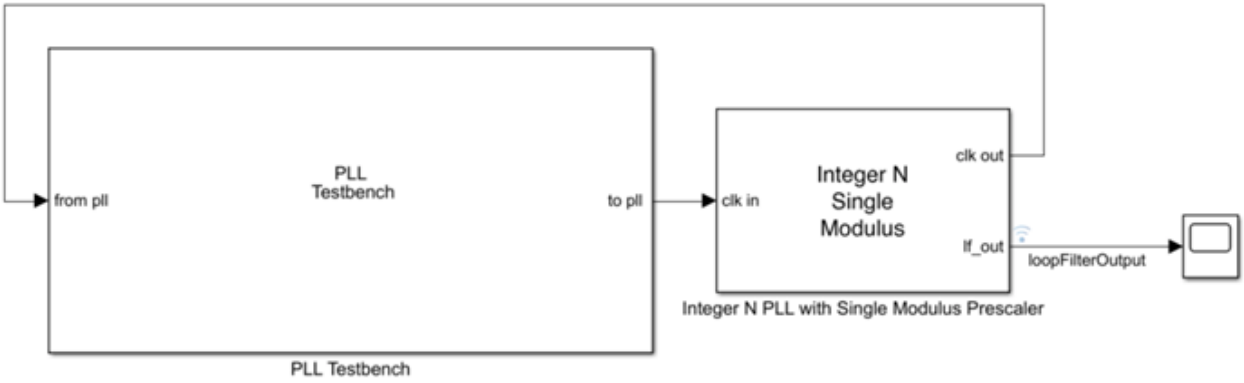
- “About Passivity and Passivity Indices” on page 10-2

Tune Phase-Locked Loop Using Loop-Shaping Design

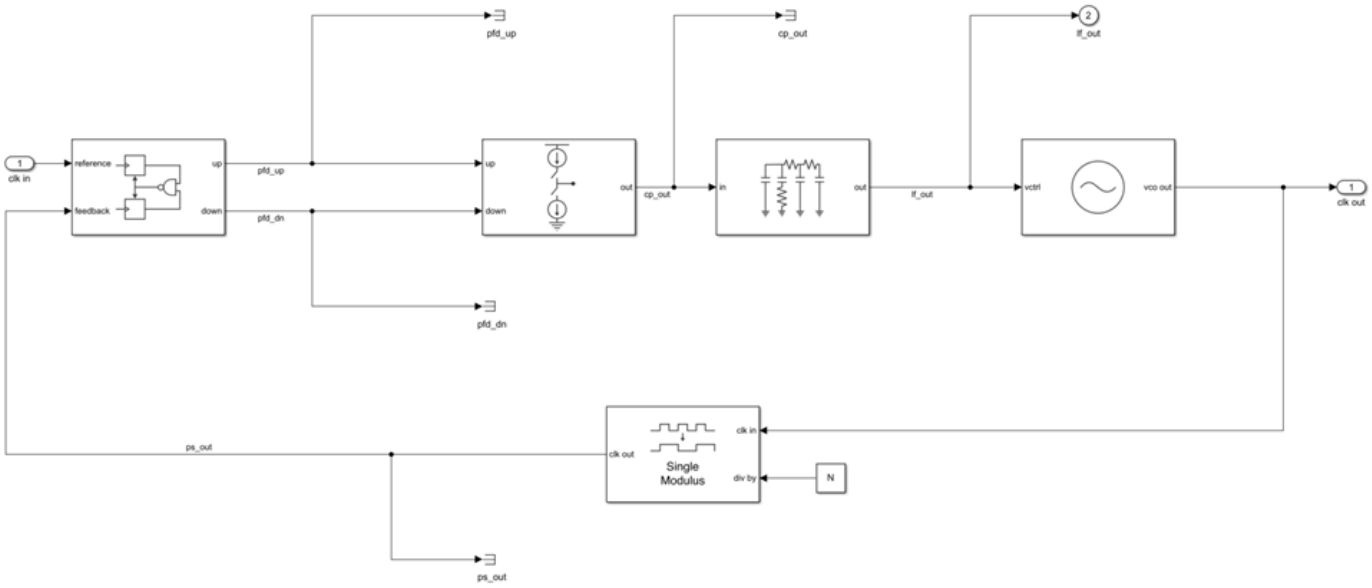
This example shows how to tune the components of a passive loop filter to improve the loop bandwidth of a phase-locked loop (PLL) system. To obtain a desired loop frequency response, this example computes the loop filter parameters using the fixed-structure tuning methods provided in the Control System Toolbox™ software. The PLL system is modeled using a reference architecture block from the Mixed-Signal Blockset™ library.

Introduction

A PLL is a closed-loop system that produces an output signal whose phase depends on the phase of its input signal. The following diagram shows a simple model with a PLL reference architecture block (Integer N PLL with Single Modulus Prescaler (Mixed-Signal Blockset)) and a PLL Testbench (Mixed-Signal Blockset) block.



The closed-loop architecture inside the PLL block consists of a phase-frequency detector (PFD), charge pump, loop filter, voltage controlled oscillator (VCO), and prescaler.



The Mixed-Signal Blockset library provides multiple reference architecture blocks to design and simulate PLL systems in Simulink®. You can tune the components of the Loop Filter (Mixed-Signal Blockset) block, which is a passive filter, to get the desired open-loop bandwidth and phase margin.

Using the Control System Toolbox software, you can specify the shape of the desired loop response and tune the parameters of a fixed-structure controller to approximate that loop shape. For more information on specifying a desired loop shape, see “Loop Shape and Stability Margin Specifications” on page 18-34. In the preceding PLL architecture model, the loop filter is defined as a fixed-order, fixed-structure controller. To achieve the target loop shape, the values of the resistances and capacitances of the loop filter are tuned. Doing so improves the open-loop bandwidth of the system and, as a result, reduces the measured lock time.

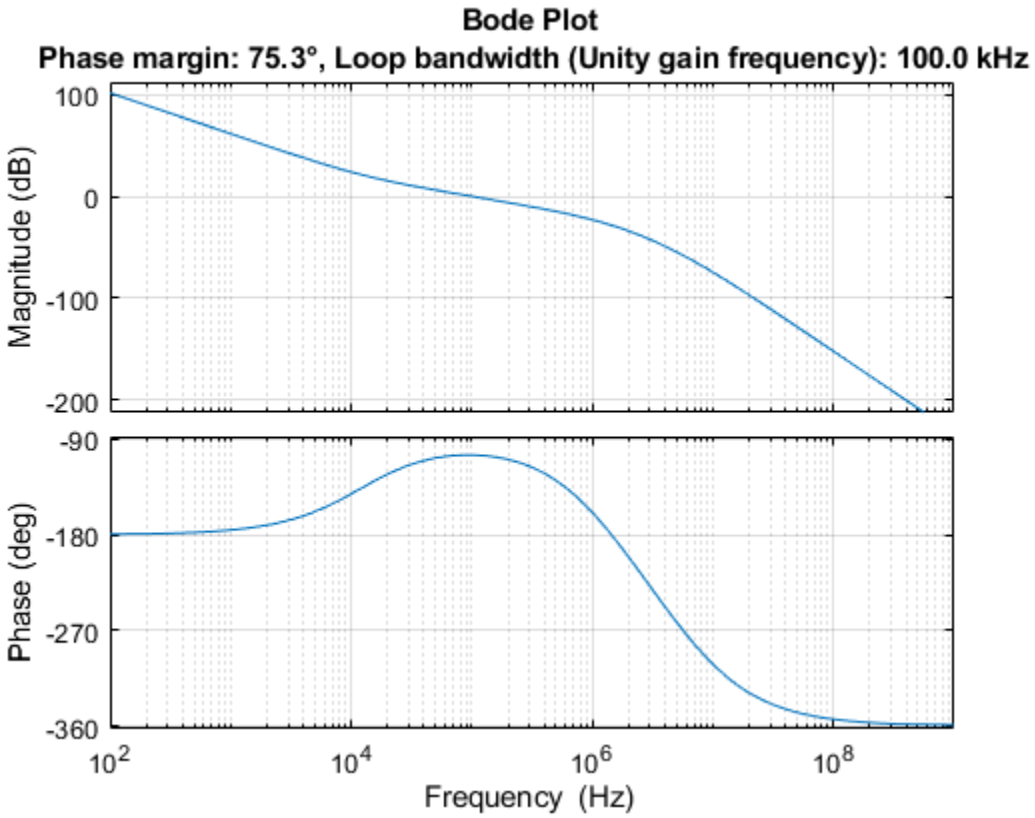
Set Up Phase-Locked Loop Model

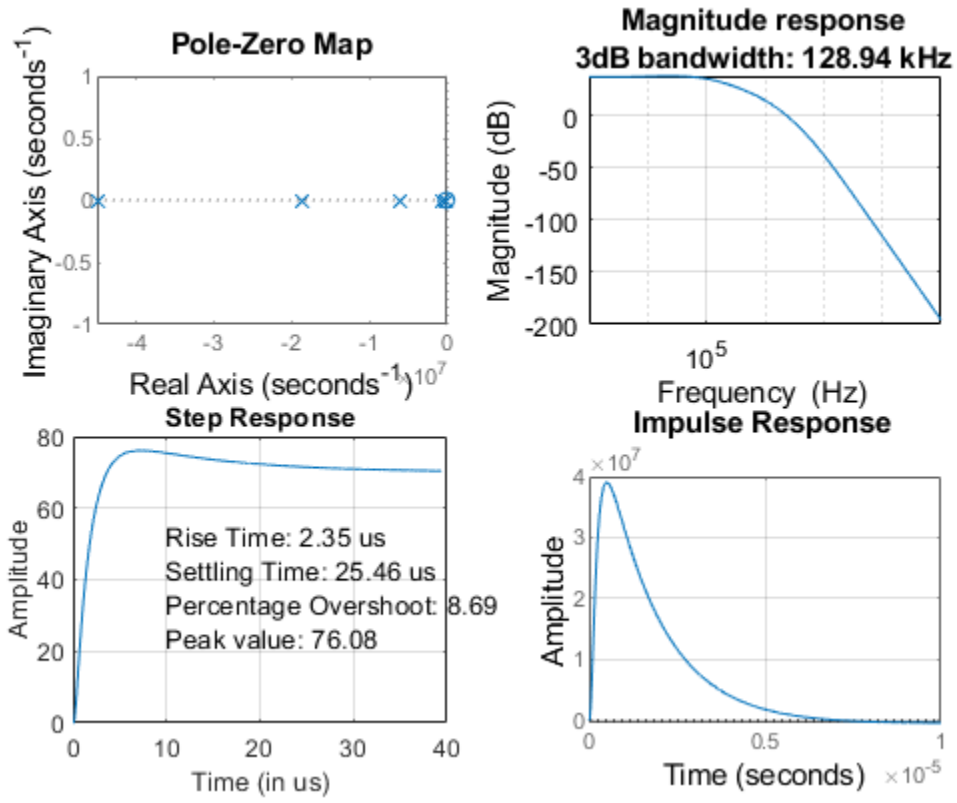
Open the model.

```
model = 'PLL_TuneLoopFilter';
open_system(model)
```

The PLL block uses the configuration specified in “Design and Evaluate Simple PLL Model” (Mixed-Signal Blockset) for the **PFD**, **Charge pump**, **VCO**, and **Prescalar** tabs in the block parameters. The **Loop Filter** tab specifies the type as a fourth-order filter, and sets the loop bandwidth to 100 kHz and phase margin to 60 degrees. The values for the resistances and capacitances are automatically computed.

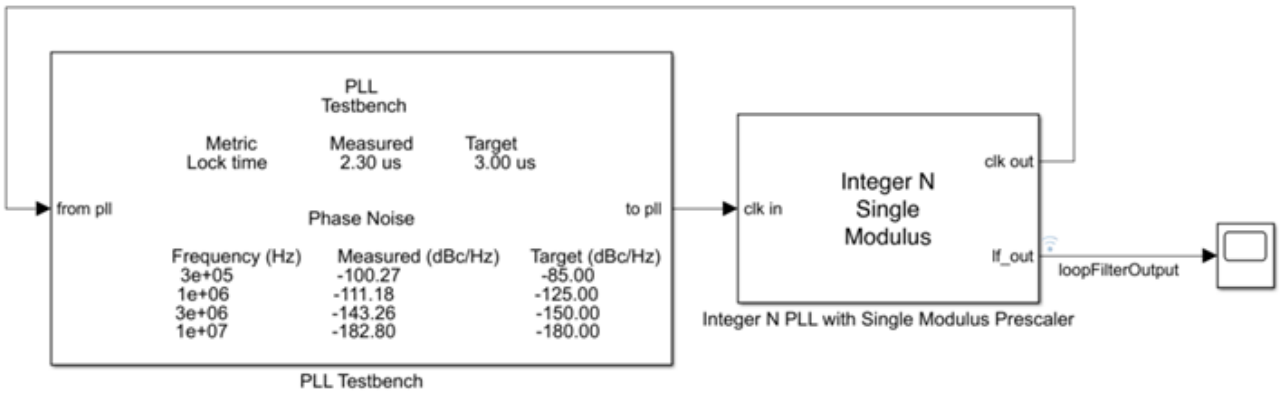
To observe the current loop dynamics of the PLL, in the block parameters, on the **Analysis** tab, select **Open Loop Analysis** and **Closed Loop Analysis**. The unity gain frequency is 100 kHz. The closed-loop system is stable and the 3-dB bandwidth is 128.94 kHz.

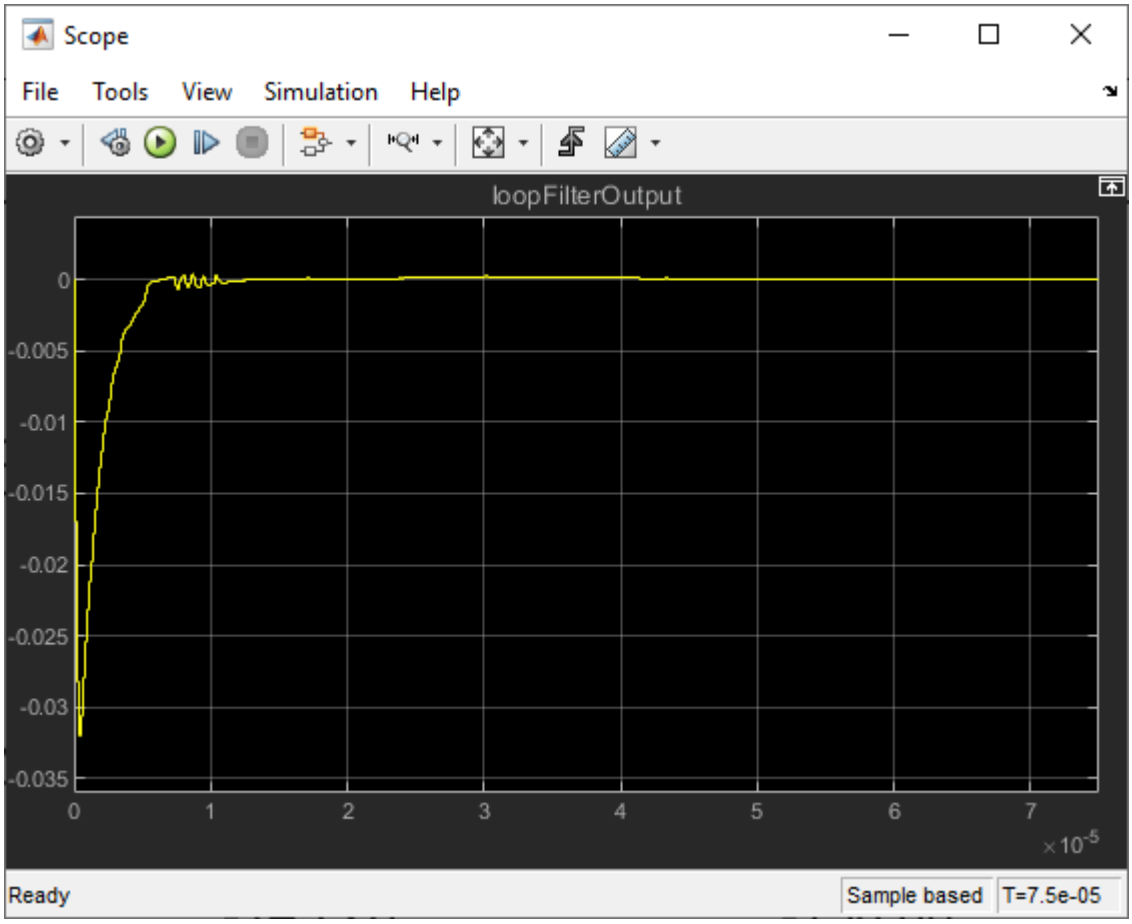

Bode Plot
Phase margin: 75.3°, Loop bandwidth (Unity gain frequency): 100.0 kHz
Magnitude (dB)
100
0
-100
-200
Phase (deg)
-90
-180
-270
-360
10^2
10^4
10^6
10^8
Frequency (Hz)

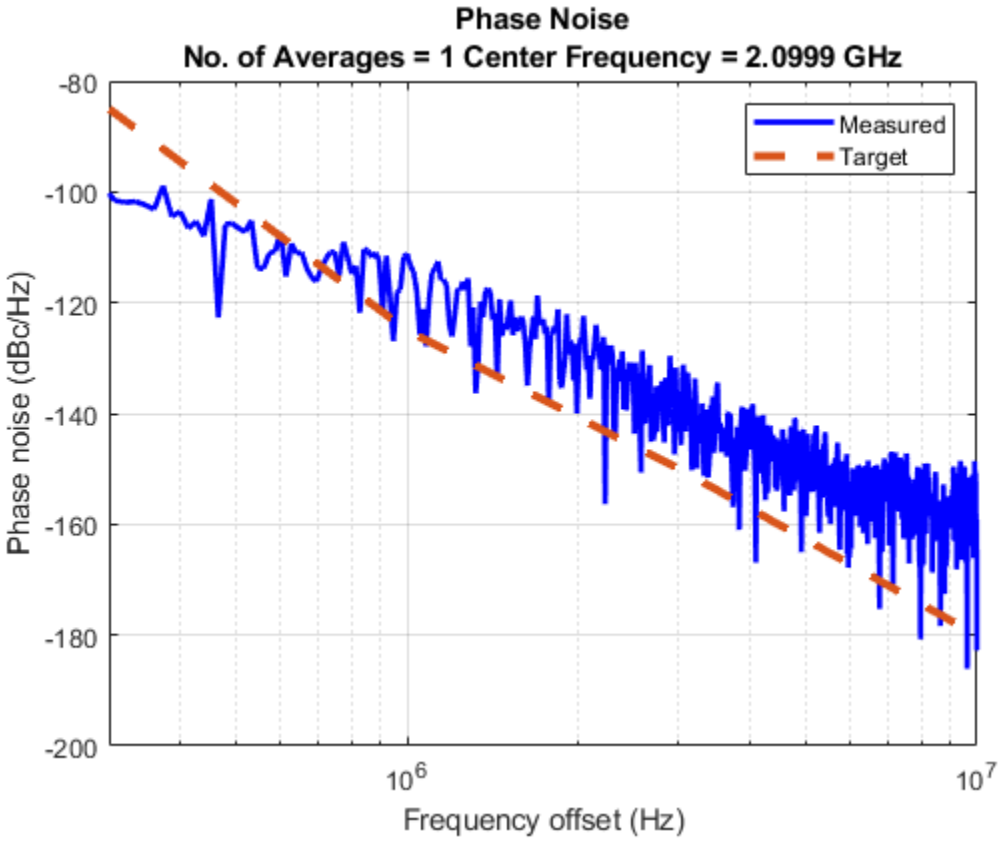


Simulate the model. The PLL Testbench block displays the PLL lock time and phase noise metrics. To plot and analyze the phase noise profile, in the PLL Testbench block parameters, on the **Stimulus** tab, select **Plot phase noise**. The measured lock time is 2.30 microseconds.

```
open_system([model,'/Scope'])
sim(model);
```




Scope
File
Tools
View
Simulation
Help
loopFilterOutput
0
-0.005
-0.01
-0.015
-0.02
-0.025
-0.03
-0.035
0
1
2
3
4
5
6
7
×10^-5
Ready
Sample based
T=7.5e-05



Define the PLL parameters needed to build the closed-loop system in MATLAB®.

```
PllKphi = 5e-3;     % Charge pump output current
PllKvco = 1e8;      % VCO sensitivity
PllN    = 70;       % Prescaler ratio

PllR2   = 88.3;     % Loop filter resistance for second-order response (ohms)
PllR3   = 253;      % Loop filter resistance for third-order response (ohms)
PllR4   = 642;      % Loop filter resistance for fourth-order response (ohms)
PllC1   = 8.13e-10; % Loop filter direct capacitance (F)
PllC2   = 1.48e-7;  % Loop filter capacitance for second-order response (F)
PllC3   = 1.59e-10; % Loop filter capacitance for third-order response (F)
PllC4   = 9.21e-11; % Loop filter capacitance for fourth-order response (F)
```

Build Custom Tunable System

To model the loop filter as a tunable element, first create tunable scalar real parameters (see `realp`) to represent each filter component. For each parameter, define the initial value and bounds. Also, specify whether the parameter is free to be tuned.

Use the current loop filter resistance and capacitance values as the initial numeric value of the tunable parameters.

```
% Resistances
R2 = realp('R2',PllR2);
R2.Minimum = 50;
R2.Maximum = 2000;
```

```
R2.Free = true;

R3 = realp('R3',PllR3);
R3.Minimum = 50;
R3.Maximum = 2000;
R3.Free = true;

R4 = realp('R4',PllR4);
R4.Minimum = 50;
R4.Maximum = 2000;
R4.Free = true;

% Capacitances
C1 = realp('C1',PllC1);
C1.Minimum = 1e-12;
C1.Maximum = 1e-7;
C1.Free = true;

C2 = realp('C2',PllC2);
C2.Minimum = 1e-12;
C2.Maximum = 1e-7;
C2.Free = true;

C3 = realp('C3',PllC3);
C3.Minimum = 1e-12;
C3.Maximum = 1e-7;
C3.Free = true;

C4 = realp('C4',PllC4);
C4.Minimum = 1e-12;
C4.Maximum = 1e-7;
C4.Free = true;
```

Using these tunable parameters, create a custom tunable model based on the loop filter transfer function equation specified in the More About section of the Loop Filter (Mixed-Signal Blockset) block reference page. `loopFilterSys` is a `genss` model parameterized by `R2`, `R3`, `R4`, `C1`, `C2`, `C3`, and `C4`.

$$Z(s) = \frac{R_2C_2s + 1}{s(A_4s^3 + A_3s^2 + A_2s + A_1)}$$

$$A_4 = C_1C_2C_3C_4R_2R_3R_4$$

$$A_3 = C_1C_2R_2R_3(C_3 + C_4) + C_4R_4(C_2C_3R_3 + C_1C_3R_3 + C_1C_2R_2 + C_2C_3R_2)$$

$$A_2 = C_2R_2(C_1 + C_3 + C_4) + R_3(C_1 + C_2)(C_3 + C_4) + C_4R_4(C_1 + C_2 + C_3)$$

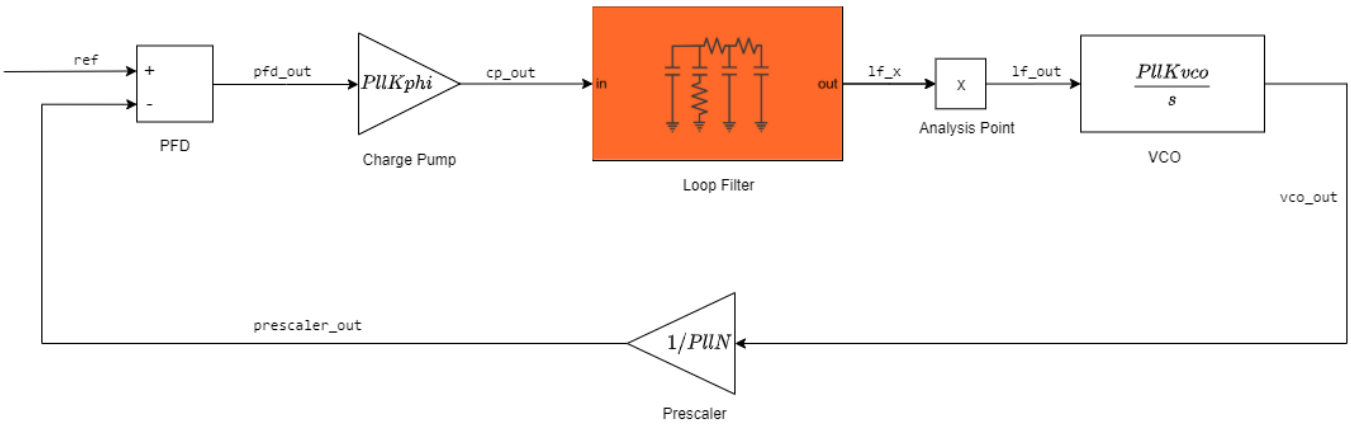
$$A_1 = C_1 + C_2 + C_3 + C_4$$

```
A4 = C1*C2*C3*C4*R2*R3*R4;
A3 = C1*C2*R2*R3*(C3+C4)+C4*R4*(C2*C3*R3+C1*C3*R3+C1*C2*R2+C2*C3*R2);
A2 = C2*R2*(C1+C3+C4)+R3*(C1+C2)*(C3+C4)+C4*R4*(C1+C2+C3);
A1 = C1+C2+C3+C4;

loopFilterSys = tf([R2*C2, 1],[A4, A3, A2, A1, 0]);
```

Use the transfer function representations to define the fixed blocks in the architecture (charge pump, VCO, and prescaler), based on their respective frequency response characteristics [1].

```
chargePumpSys    = tf(PllKphi,1);       % Linearized as a static gain
vcoSys           = tf(PllKvco,[1 0]);   % Linearized as an integrator
prescalerSys     = tf(1/PllN,1);        % Linearized as a static gain
```

Define input and output names for each block. Connect the elements based on signal names (see `connect`) to create a tunable closed-loop system (see `genss`) representing the PLL architecture as shown.




```
chargePumpSys.InputName = 'pfd_out';                % Charge pump (fixed block)
chargePumpSys.OutputName = 'cp_out';

loopFilterSys.InputName = 'cp_out';                 % Loop filter (tunable block)
loopFilterSys.OutputName = 'lf_x';

AP = AnalysisPoint('X');                            % Analysis point does not change the architec
AP.InputName = 'lf_x';
AP.OutputName = 'lf_out';

vcoSys.InputName = 'lf_out';                        % VCO (fixed block)
vcoSys.OutputName = 'vco_out';

prescalerSys.InputName = 'vco_out';                 % Prescaler (fixed block)
prescalerSys.OutputName = 'prescaler_out';

pfd = sumblk('pfd_out = ref - prescaler_out');      % Phase-frequency detector (sum block)

% Create a genss model for the closed-loop architecture
CL0 = connect(chargePumpSys,loopFilterSys,AP,vcoSys,prescalerSys,pfd,'ref','vco_out');
```

Loop-Shaping Design

Define the loop gain as a frequency-response data model by providing target gains for at least two decades below and two decades above the desired open-loop bandwidth. The desired roll-off is typically higher, which results in a higher attenuation of phase noise.

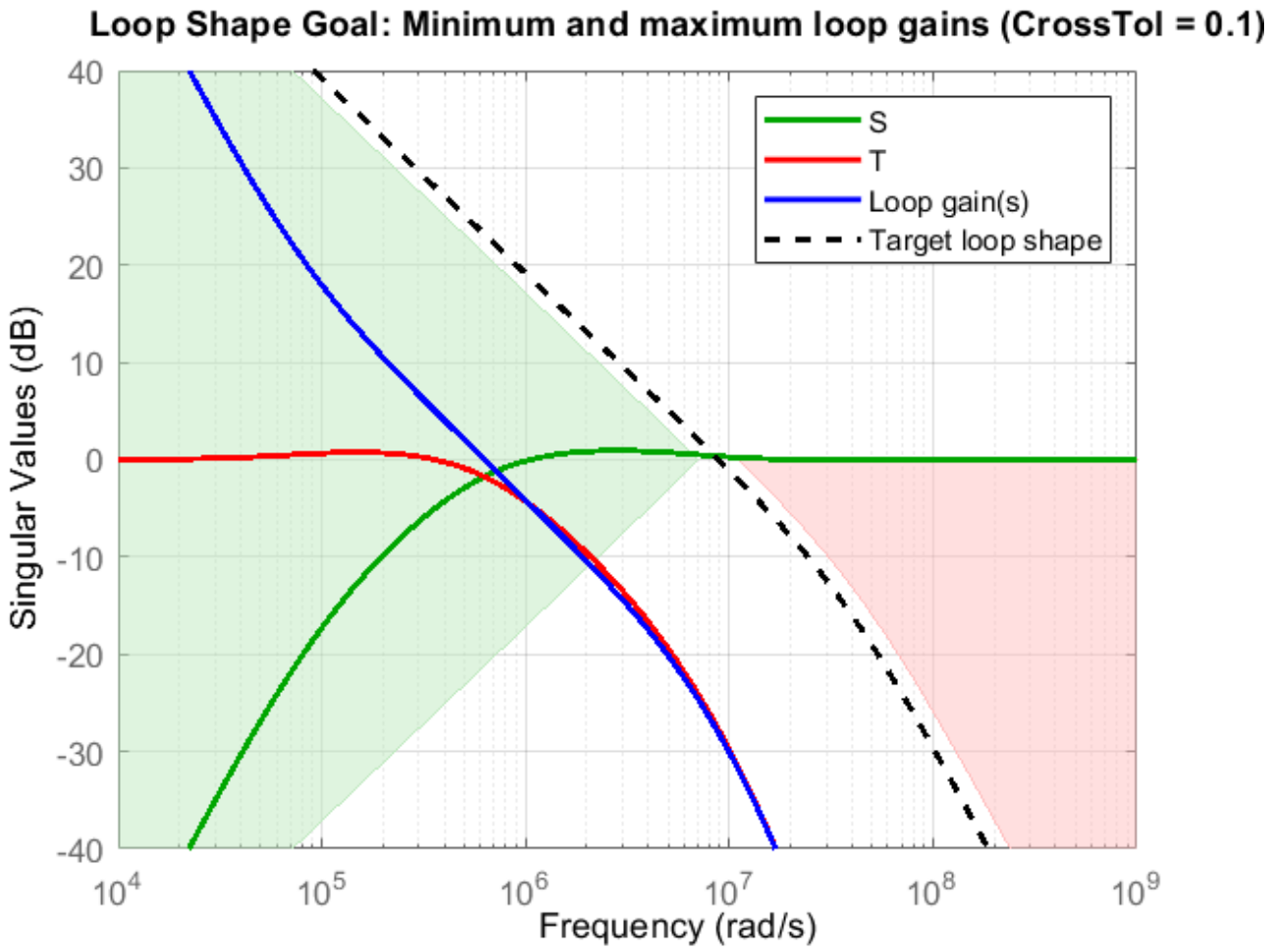
Specifying the appropriate target loop shape is the critical aspect of this design. The tunable compensator is a fourth-order system with a single integrator and a single zero, and the plant represents an integrator. The loop gains must be a feasible target for the open-loop structure.

For tuning the loop filter, create a tuning goal based on a target loop shape specifying the integral action, a 3 MHz crossover, and a roll-off requirement of 40 dB/decade. The goal is enforced for three decades below and above the desired open-loop bandwidth.

```
LoopGain = frd([100,10,1,1e-2,1e-4],2*pi*[1e4,1e5,3e6,3e7,3e8]);    % frd uses response data and
LoopShapeGoal = TuningGoal.LoopShape('X',LoopGain);                  % Use AnalysisPoint as locati
LoopShapeGoal.Focus = 2*pi*[1e3, 1e9];                               % Enforce goal in frequency r
LoopShapeGoal.Name = 'Loop Shape Goal';                              % Tuning goal name

MarginsGoal = TuningGoal.Margins('X',7.6,60);
MarginsGoal.Focus = [0 Inf];
MarginsGoal.Openings = {'X'};
MarginsGoal.Name = 'Margins Goal';
```

Observe the current open-loop shape of the PLL system with reference to the target loop shape. **S** represents the inverse sensitivity function and **T** represents the complementary sensitivity function. By default, Control System Toolbox plots use rad/s as the frequency unit. For more information on how to change the frequency unit to Hz, see “Toolbox Preferences Editor” on page 20-2.

```
figure
viewGoal(LoopShapeGoal,CL0)
```



Use `systune` to tune the fixed-structure feedback loop. Doing so computes the resistance and capacitance values to meet the soft design goal based on the target loop shape. Run the tuning algorithm with five different initial value sets in addition to the initial values defined during the creation of the tunable scalar real parameters.

```
Options = systuneOptions();
Options.SoftTol = 1e-5;      % Relative tolerance for termination
Options.MinDecay = 1e-12;    % Minimum decay rate for closed-loop poles
Options.MaxRadius = 1e12;    % Maximum spectral radius for stabilized dynamics
Options.RandomStart = 5;     % Number of different random starting points

[CL,fSoft,gHard,Info] = systune(CL0,[LoopShapeGoal; MarginsGoal],[],Options);
```

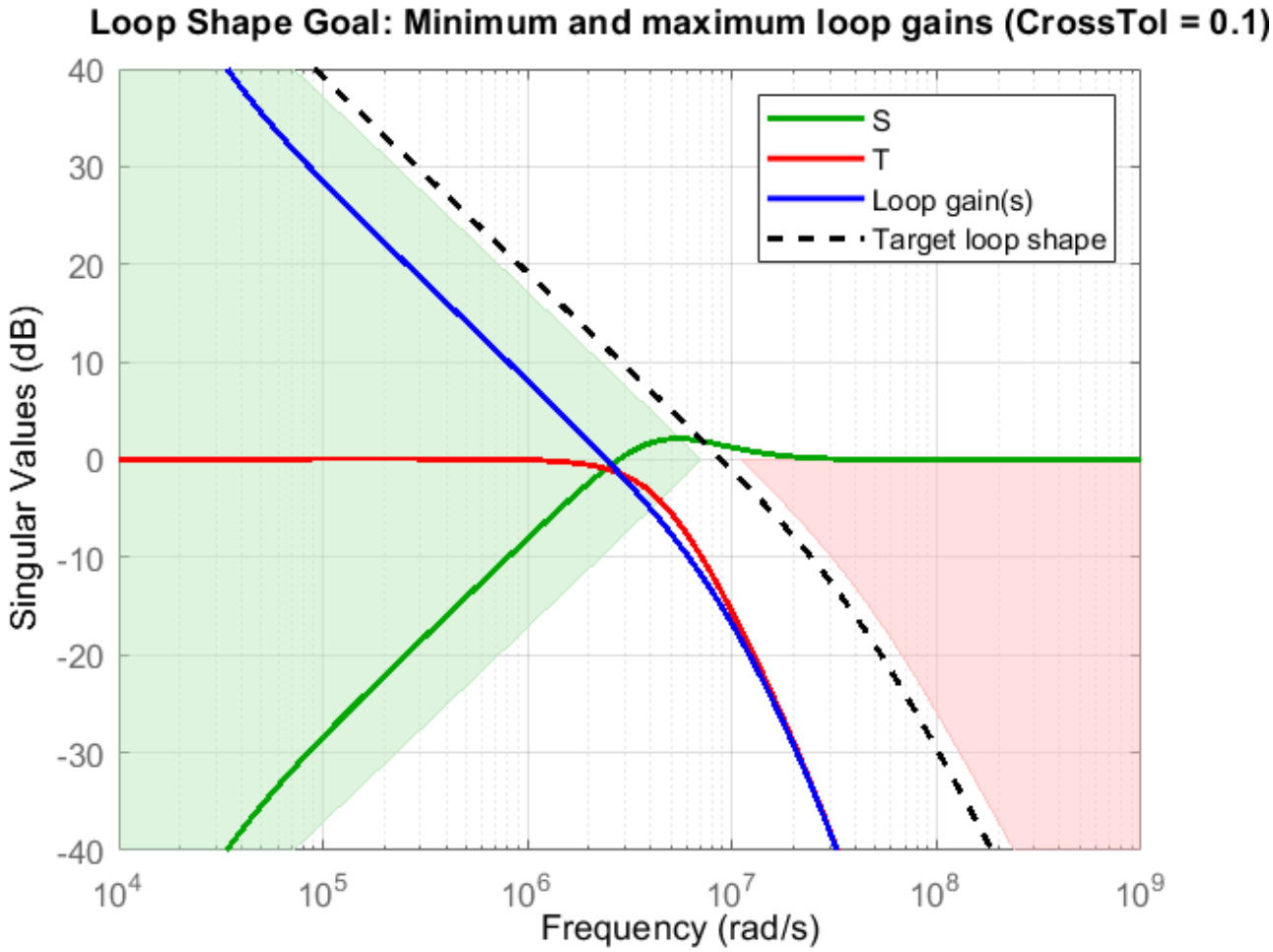
```
Final: Soft = 2.85, Hard = -Inf, Iterations = 50
Final: Failed to enforce closed-loop stability (max Re(s) = 3.1e+04)
Final: Failed to enforce closed-loop stability (max Re(s) = 6.8e+04)
Final: Failed to enforce closed-loop stability (max Re(s) = 6.2e+04)
Final: Failed to enforce closed-loop stability (max Re(s) = 8e+04)
Final: Failed to enforce closed-loop stability (max Re(s) = 4.9e+04)
```

`systune` returns the tuned closed-loop system `CL` in generalized state-space form.

The algorithm fails to converge for the random initial values, and provides a feasible solution only when the current loop filter component values are chosen as the initial conditions. For tuning problems that are less complex, such as a third-order loop filter, the algorithm is less sensitive to initial conditions and randomized starts are an effective technique for exploring the parameter space and converging to a feasible solution.

Examine the tuned open-loop shape with reference to the target loop shape. Observe that while the tuned loop shape does not meet the target, the open-loop bandwidth increases while the loop keeps the same high-frequency attenuation.

```
figure
viewGoal(LoopShapeGoal,CL)
```



Export Results to Simulink Model

Extract the tuned loop filter component values.

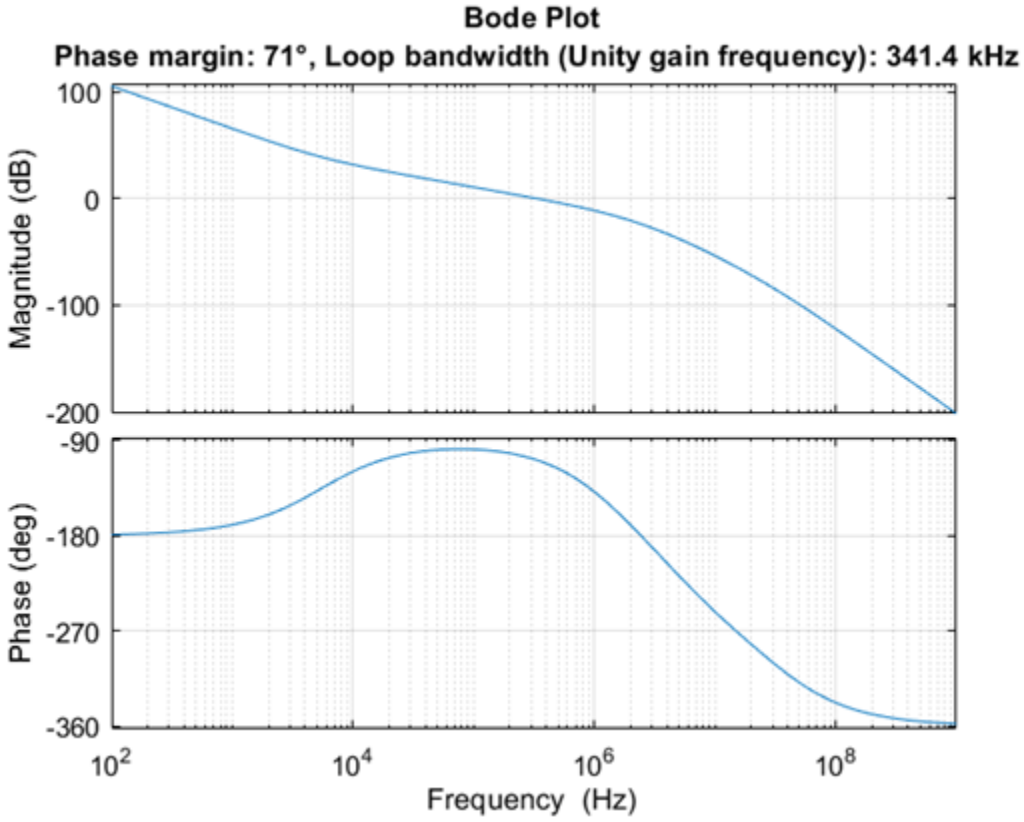
```
Rtuned = [getBlockValue(CL,'R2'),...
          getBlockValue(CL,'R3'),...
          getBlockValue(CL,'R4')];

Ctuned = [getBlockValue(CL,'C1'),...
          getBlockValue(CL,'C2'),...
          getBlockValue(CL,'C3'),...
          getBlockValue(CL,'C4')];
```

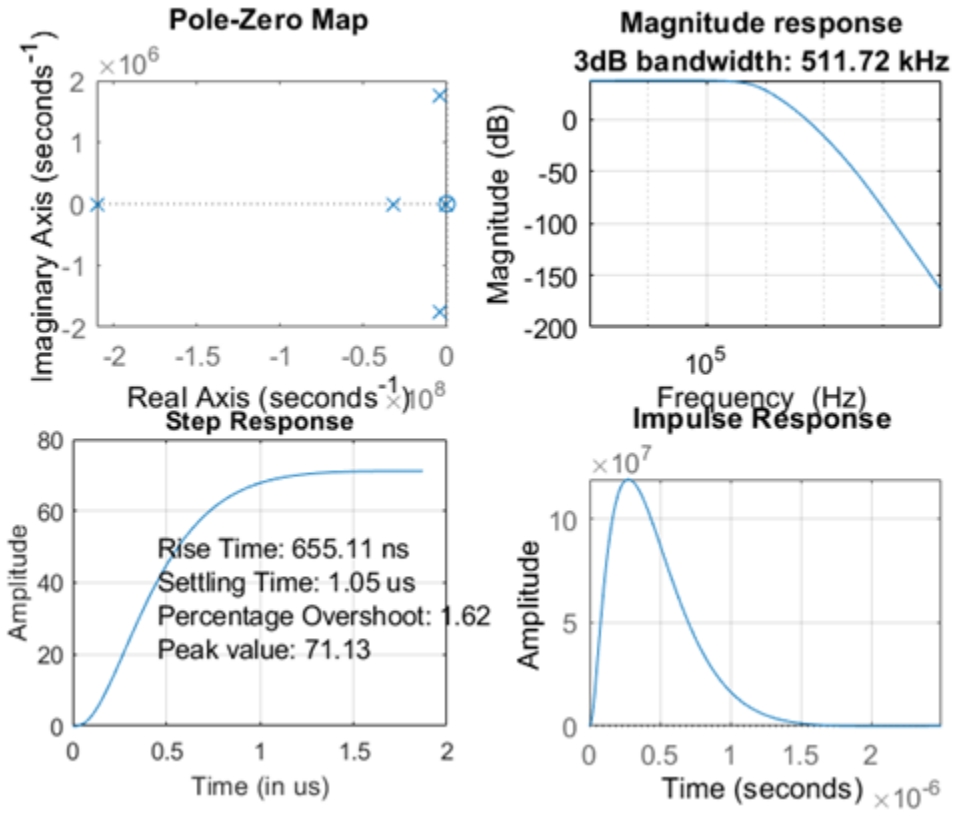
Write the tuned loop filter component values to the PLL block using the `setLoopFilterValue` helper function provided with the example.

```
blk = [model,'/Integer N PLL with Single Modulus Prescaler'];
setLoopFilterValue(blk,Rtuned,Ctuned);
```

Observe the Open Loop Analysis and Closed Loop Analysis plots from the **Analysis** tab in the Integer N PLL with Single Modulus Prescaler block parameters. The unity gain frequency and the 3-dB

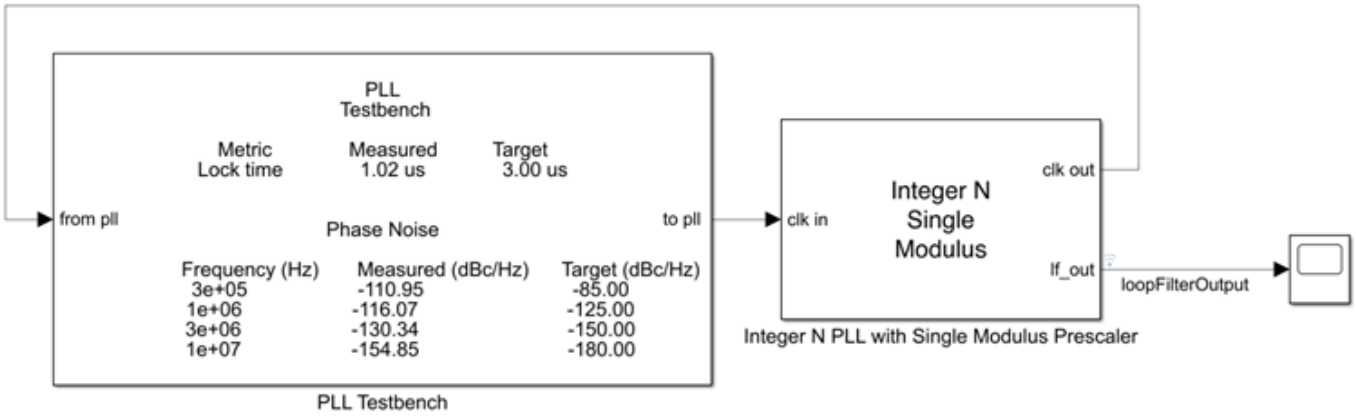
bandwidth show improvement and are now 341.4 kHz and 511.72 kHz, respectively, while the loop keeps the same phase noise profile.

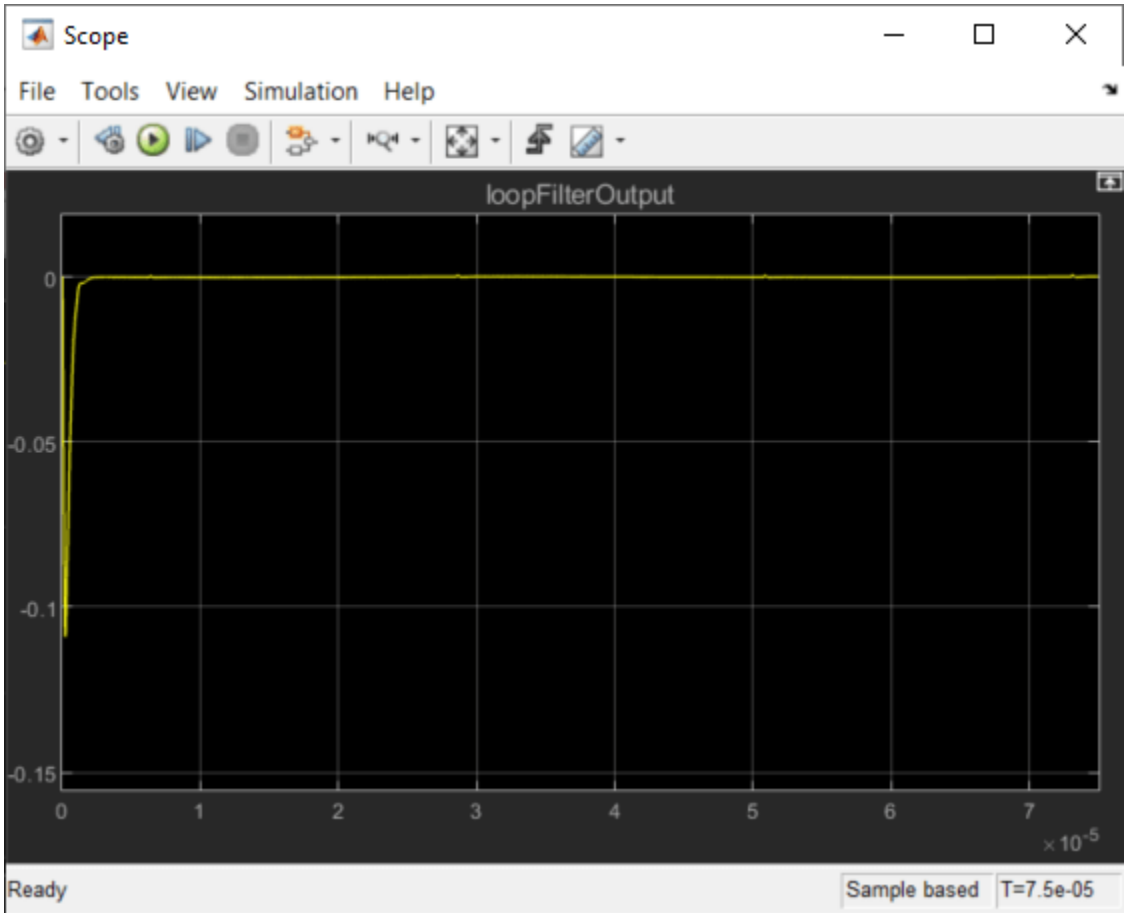


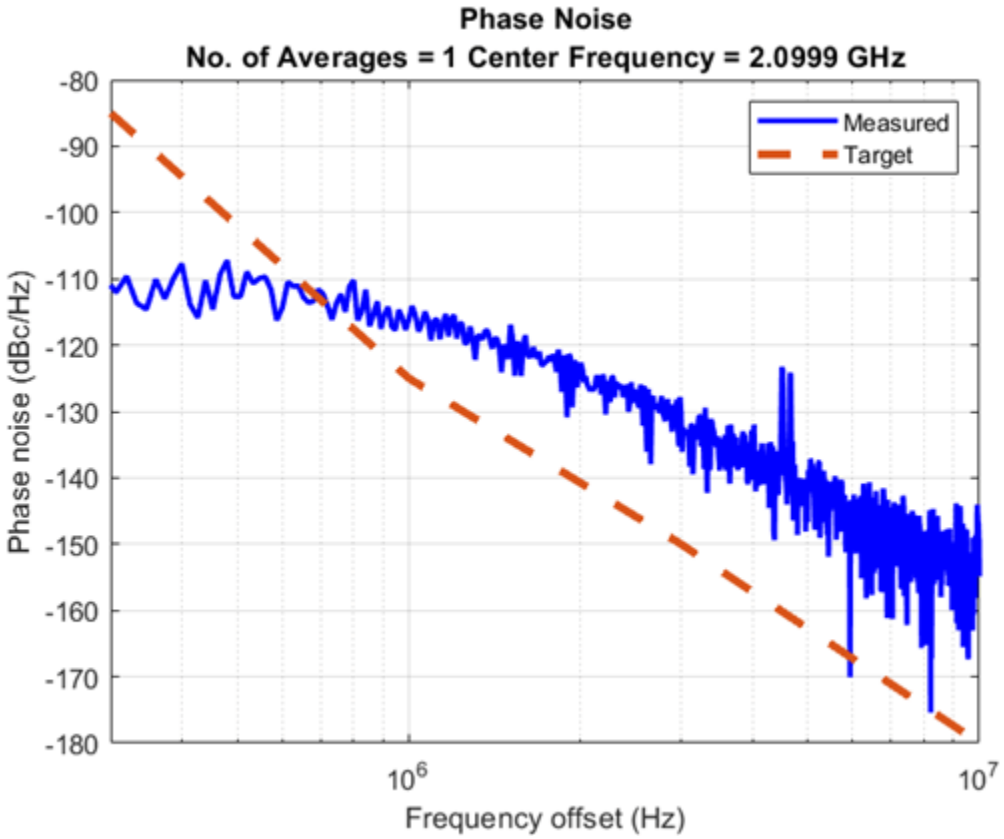


Simulate the model and get the PLL Testbench measurements and loop filter output with the tuned components.

```
sim(model);
```




Scope
File Tools View Simulation Help
loopFilterOutput
0
-0.05
-0.1
-0.15
0 1 2 3 4 5 6 7
×10^-5
Ready
Sample based
T=7.5e-05



References

[1] Banerjee, Dean. *PLL Performance, Simulation and Design*. Indianapolis, IN: Dog Ear Publishing, 2006.

See Also

`systune`

Related Examples

- “Design and Evaluate Simple PLL Model” (Mixed-Signal Blockset)
- “Loop Shape and Stability Margin Specifications” on page 18-34

More About

- “PLL Design and Verification Using Data Sheet Specifications” (Mixed-Signal Blockset)

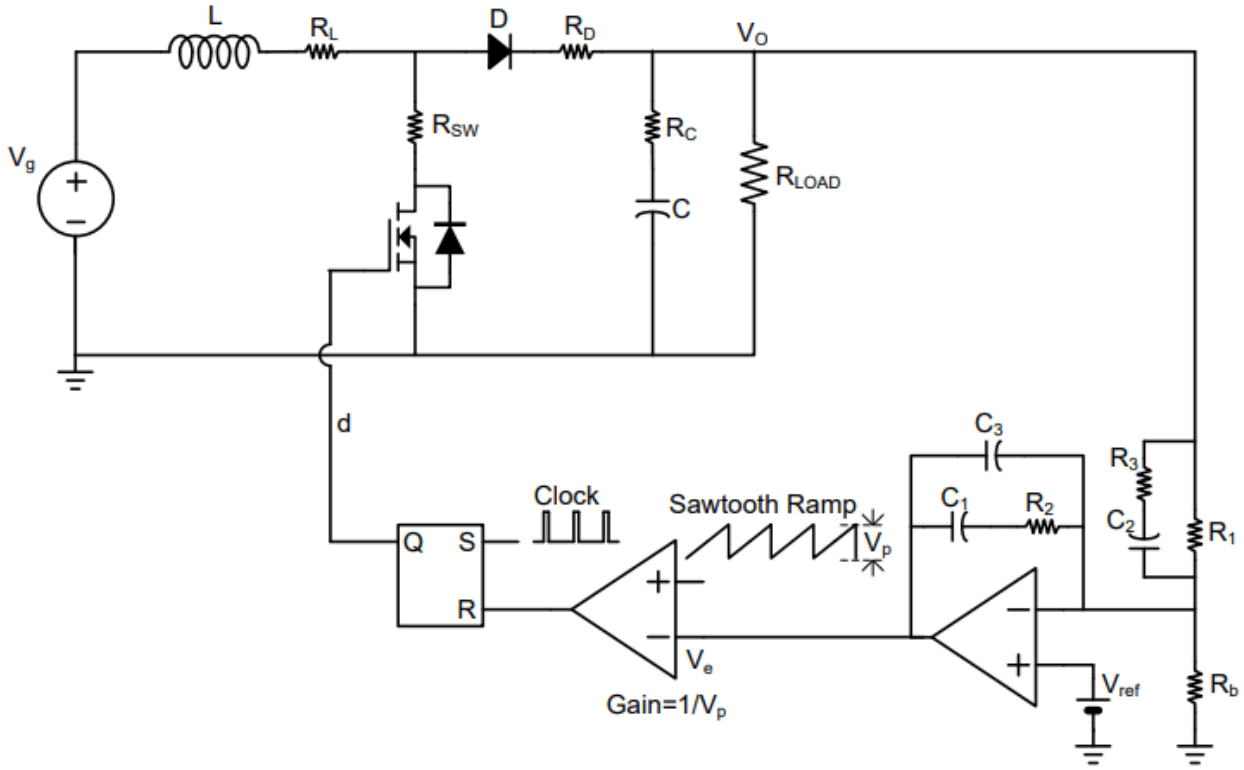
Feedback Amplifier Design for Voltage-Mode Boost Converter

This example shows how to tune the components of a power supply controller to control the output voltage of a boost converter using loop-shape design and fixed-structure tuning methods. This workflow is demonstrated using a boost converter model and a type-III controller.

You need a Mixed Signal Blockset® license to run this example.

Power Train

This example uses the boost converter and feedback amplifier circuit defined in [1].



The power supply system consists of a voltage source, boost power train, load, feedback amplifier and pulse width modulator. The boost converter converts input voltage V_g to output voltage V_o. The output voltage is measured across the load R_{LOAD}, and I_{LOAD} is the current through the load. This conversion is controlled by the pulse duty cycle d at the output of the pulse width modulator, which is in turn controlled by the feedback amplifier. The feedback amplifier senses the output voltage and attempts to make it a fixed multiple of the reference voltage V_{ref}, in response to changes in the source voltage, load current, and load resistance.

Boost Converter Model

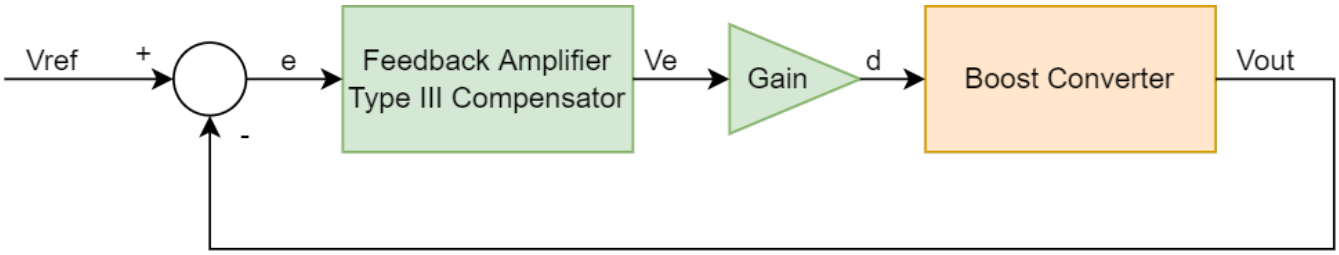
Create a transfer function model for the boost converter with defined component values, at an operating point specified using the duty cycle and output voltage values observed in [2]. Here, use the `getBoostConverterPlant` helper function (available in the folder for this example) to set up

this model. For more information, see “Design Controller for Boost Converter Model Using Frequency Response Data” (Simulink Control Design).

```
boostConverterPlant = getBoostConverterPlant();
boostConverterPlant.InputName = {'d'};
boostConverterPlant.OutputName = {'Vo'};
```

Tunable Control Architecture with Type-III Compensator

Now, set up a closed-loop system for the boost converter using a type-III compensator as a feedback amplifier and a simple gain block as the modulator gain, as shown in the following architecture diagram.



The objective is to tune the resistance (`R1`, `R2`, `R3`) and capacitance (`C1`, `C2`, `C3`) values of the type-III compensator which amplifies the error between the reference voltage and output voltage. A type-III compensator can increase phase in the mid-frequency range while improving both low and high frequency responses.

Create Tunable Linear System for Type-III Compensator

The type-III compensator structure is defined in the netlist file `TypeIII_simple.sp`. Run the `getControlModel` helper function, which uses a Linear Circuit Wizard block to parse the file and extract a tunable linear model. Configure the compensator using the `configureTunableBlock` helper function, and use a tunable gain in series to get the final `controlBlock`.

```
modelName = 'TypeIIICompensator';
load_system(modelName);
lcwBlock = [modelName,'/Linear Circuit Wizard'];
loadConfiguration(lcwBlock,'TypeIII_simpleCfg.mat');

% You can now open the mask of the Linear Circuit Wizard
% and examine or adjust the circuit configuration.

% Construct the symbolic model.
msblks.Circuit.packageCircuitAnalysis(lcwBlock,'Linear analysis');

compensator = getControlModel(lcwBlock,TypeIII_simpleCfgSymbolicModel);
compensator = configureTunableBlock(compensator);

K = realp('K',-1);
K.Minimum = -1.2;
K.Maximum = 0;

controlBlock = K*compensator;
controlBlock.InputName = {'e'};
controlBlock.OutputName = {'d'};
```

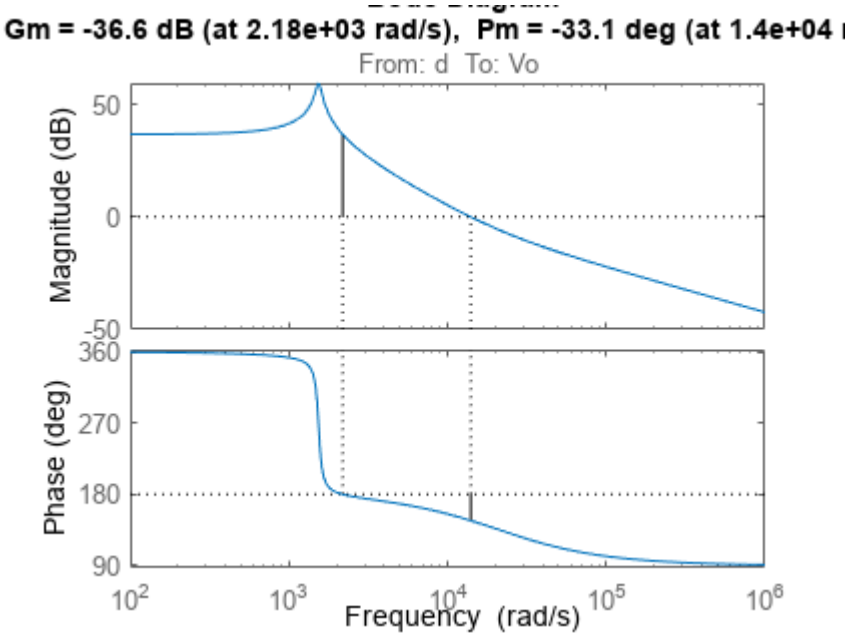
Connect Tunable and Fixed Blocks to Create Closed Loop System

```
eSum = sumblk('e = Vref - VoMeasured');
dSum = sumblk('VoMeasured = dVo + Vo');

closedLoopSystem = connect(controlBlock,boostConverterPlant,eSum,dSum,{'Vref','dVo'},'VoMeasured
```

Loop-Shaping Design

First, view the stability margins and frequency response of the open-loop boost converter plant.

```
figure;
margin(boostConverterPlant);
```



Design Requirements

Use the following design requirements to define a stable target loop shape:

- Increase gain at low frequencies. Doing so improves the transient performance of reference voltage tracking and output voltage disturbance rejection and also reduces steady state error.
- Add phase in middle frequencies to increase bandwidth (reduce response time) and achieve stability with positive gain and phase margins.
- Reduce gain at high frequencies to make the closed loop system robust by attenuating oscillations in the control signal created by introduced harmonics and noisy measurements in the output voltage.

Use the helper function `getTuningGoals` to create the following tuning goal objects, which help you define constraints and objectives when you tune the system.

- `marginsGoal` - `TuningGoal.Margins` with a gain margin constraint of 5 dB and phase margin constraint of 30 degrees. These constraints enforce stability and also help you visualize, using `viewGoal`, how much uncertainty the loop can tolerate at different frequencies before going unstable.

- `minLoopGainGoal` - `TuningGoal.MinLoopGain` with an integral action gain profile and a bandwidth of 1 KHz. The constraint is enforced in the low frequency range (10 Hz to 200 Hz).
- `maxLoopGainGoal` - `TuningGoal.MaxLoopGain` with a double integrator gain profile and a roll-off of -40 dB/decade, enforced in the high frequency range (1.5 KHz to 15 KHz).

```
[marginsGoal, minLoopGainGoal, maxLoopGainGoal] = getTuningGoals();
```

Tune System

Now, use `systune` to tune the system, that is, compute the values of the tunable parameters (`C1`, `C2`, `C3`, `R1`, `R2`, `R3`, and `K`) such that the open loop response meets the desired design requirements.

First, create a `systuneOptions` object and adjust the values for the minimum decay rate and maximum spectral radius to suit tuning for high bandwidth loop shapes. Also, reduce the relative tolerance criteria for termination.

```
opts = systuneOptions;
opts.MinDecay = 1e-15;
opts.MaxRadius = 1e15;
opts.SoftTol = 1e-10;
```

Tune the system using `systune`, with the margin goals enforced as objectives (soft goals) and minimum and maximum loop gain goals enforces as constraints (hard goals). The tuning returns a converged result that satisfies the hard goals and optimizes the soft goals. The best achieved values for the soft and hard goals are both less than 1.

```
tunedClosedLoopSystem = systune(closedLoopSystem,marginsGoal,[minLoopGainGoal,maxLoopGainGoal],op
```

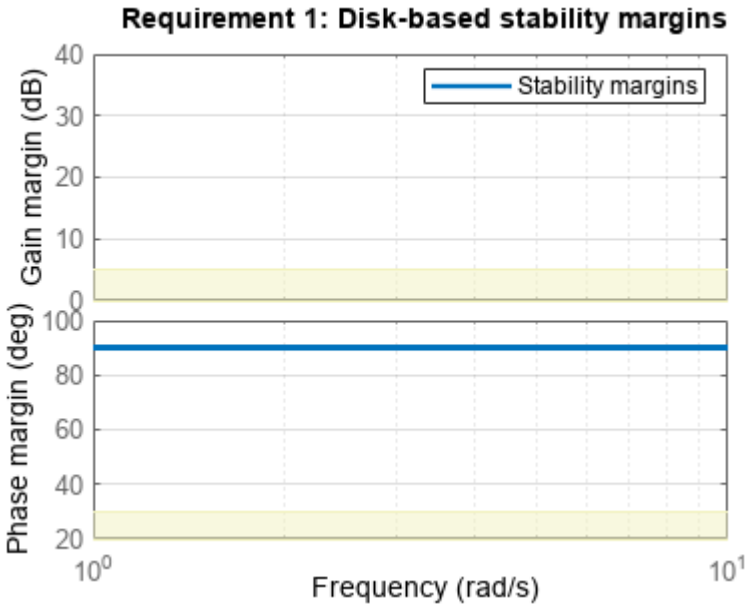
```
Final: Soft = 0.28, Hard = 99.504, Iterations = 20
```

Analyze Results

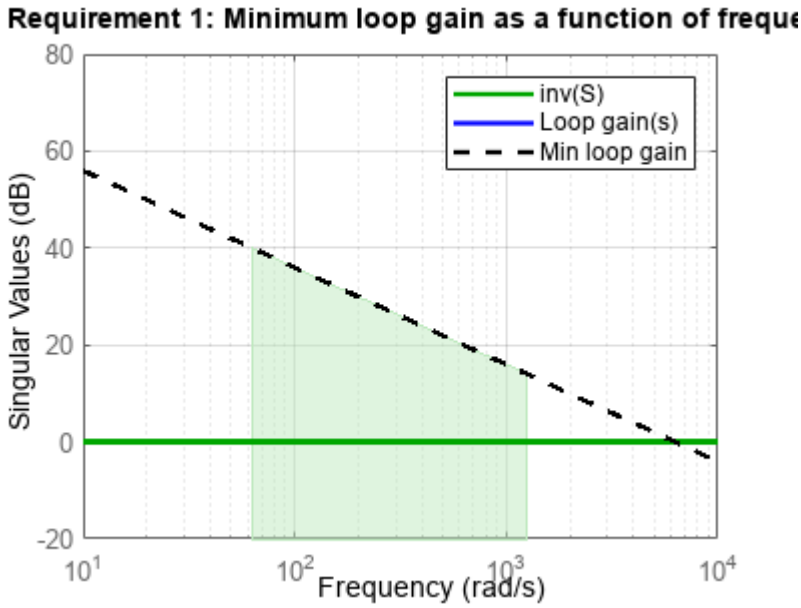
View Tuned System Results Against Desired Specifications

Create plots of the tuned system against the tuning goals to indicate how closely it meets the desired specifications. The shaded regions in each plot represent where the tuning goal is violated. The plots show that the constraints of tracking error and stability margins are met, while the objectives of minimum and maximum loop gains in specific frequency ranges are optimized.

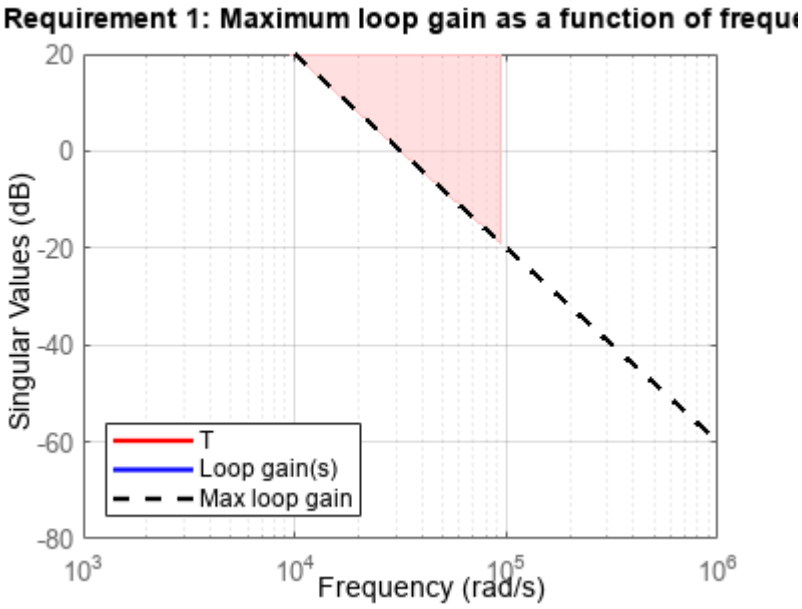
```
viewGoal(marginsGoal,tunedClosedLoopSystem);
```




```
viewGoal(minLoopGainGoal,tunedClosedLoopSystem);
```



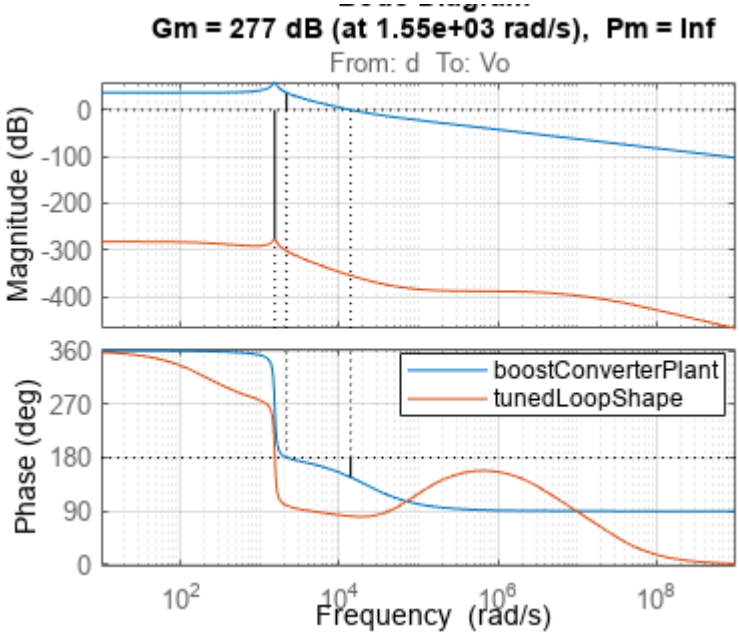

```
viewGoal(maxLoopGainGoal,tunedClosedLoopSystem);
```



View Tuned Loop Shape and Stability Margins

The margin plot of the tuned open loop system indicates the bandwidth and robustness along with the achieved gain and phase margins.

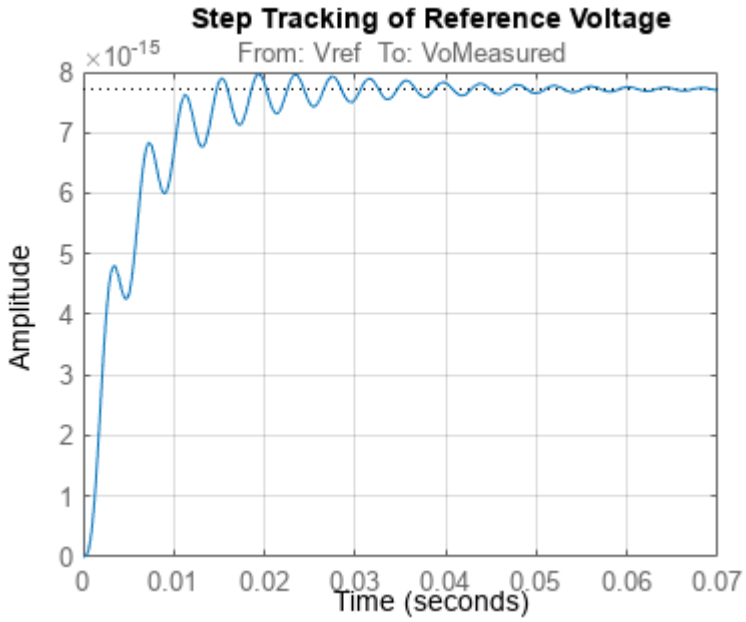
```
tunedLoopShape = getLoopTransfer(tunedClosedLoopSystem,'d',-1);
figure;
margin(boostConverterPlant);
hold on;
margin(tunedLoopShape);
grid on;
legend('show','Location','best');
```



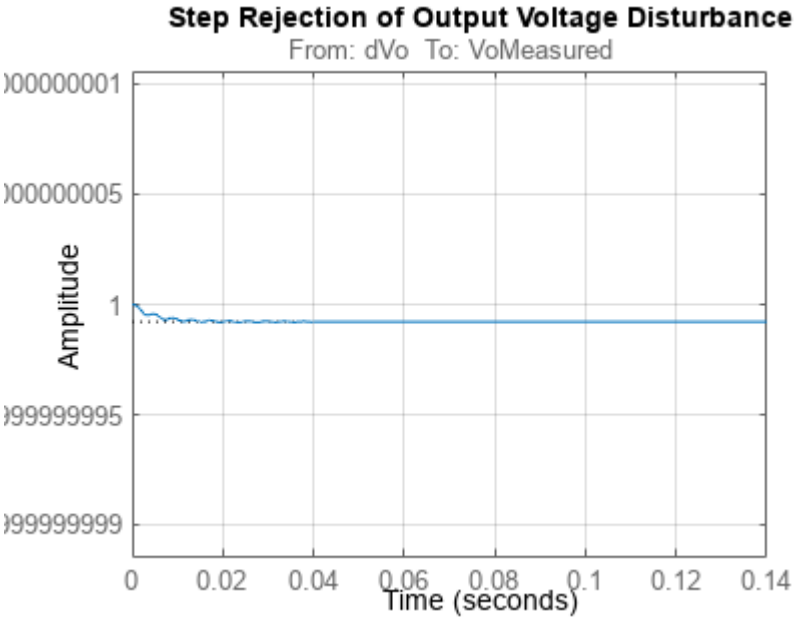
View Step Responses of Closed Loop System

Step response plots of reference voltage tracking and output voltage disturbance tracking show that the response settles without any steady state error with an acceptably small settling time.

```
figure;
step(getIOTransfer(tunedClosedLoopSystem,'Vref','VoMeasured'));
title('Step Tracking of Reference Voltage');
grid on;
```




```
figure;
step(getIOTransfer(tunedClosedLoopSystem,'dVo','VoMeasured'));
title('Step Rejection of Output Voltage Disturbance');
grid on;
```



Tuned Component Values

Get the transfer function from the controller input `e` to the duty cycle `d` and display the tuned component values.

```
tunedCompensator = getIOTransfer(tunedClosedLoopSystem,'e','d','d');
showTunable(tunedCompensator);
```

```
C1 = 4.08e-09
-----------------------------------
C2 = 1e-11
-----------------------------------
C3 = 1e-11
-----------------------------------
K = -1.2
-----------------------------------
R1 = 1e+06
-----------------------------------
R2 = 1e+04
-----------------------------------
R3 = 1e+04
```

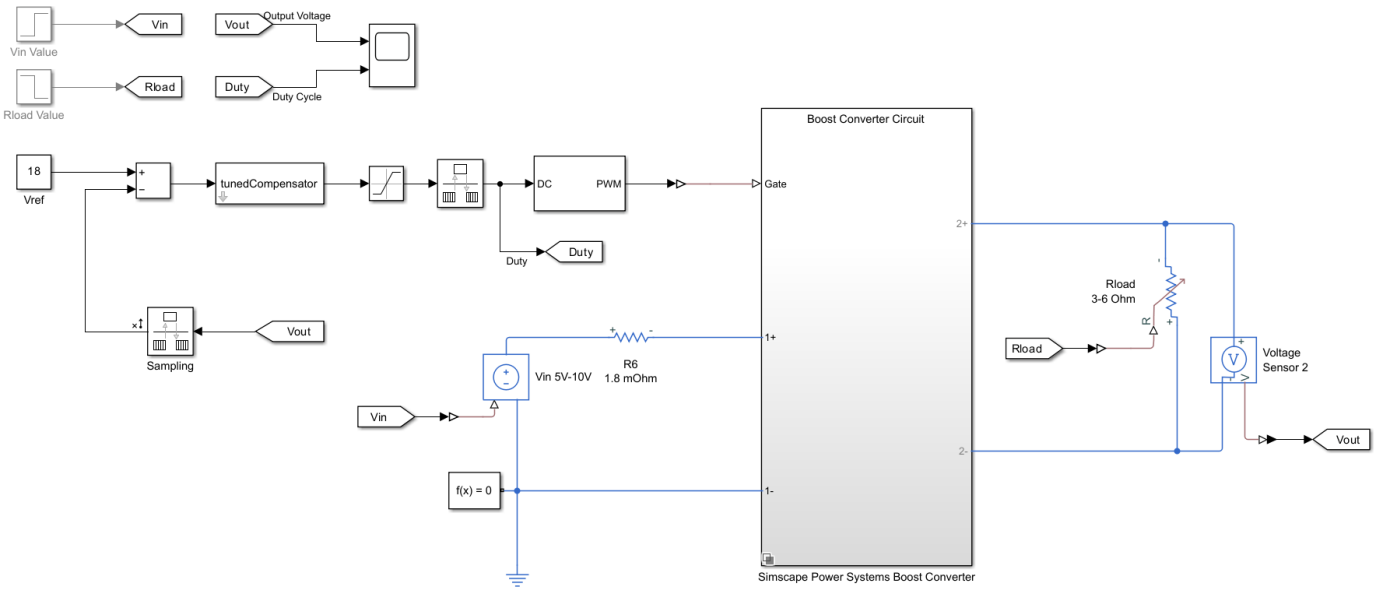
Simulation Results

Simulation Result with Tuned Controller

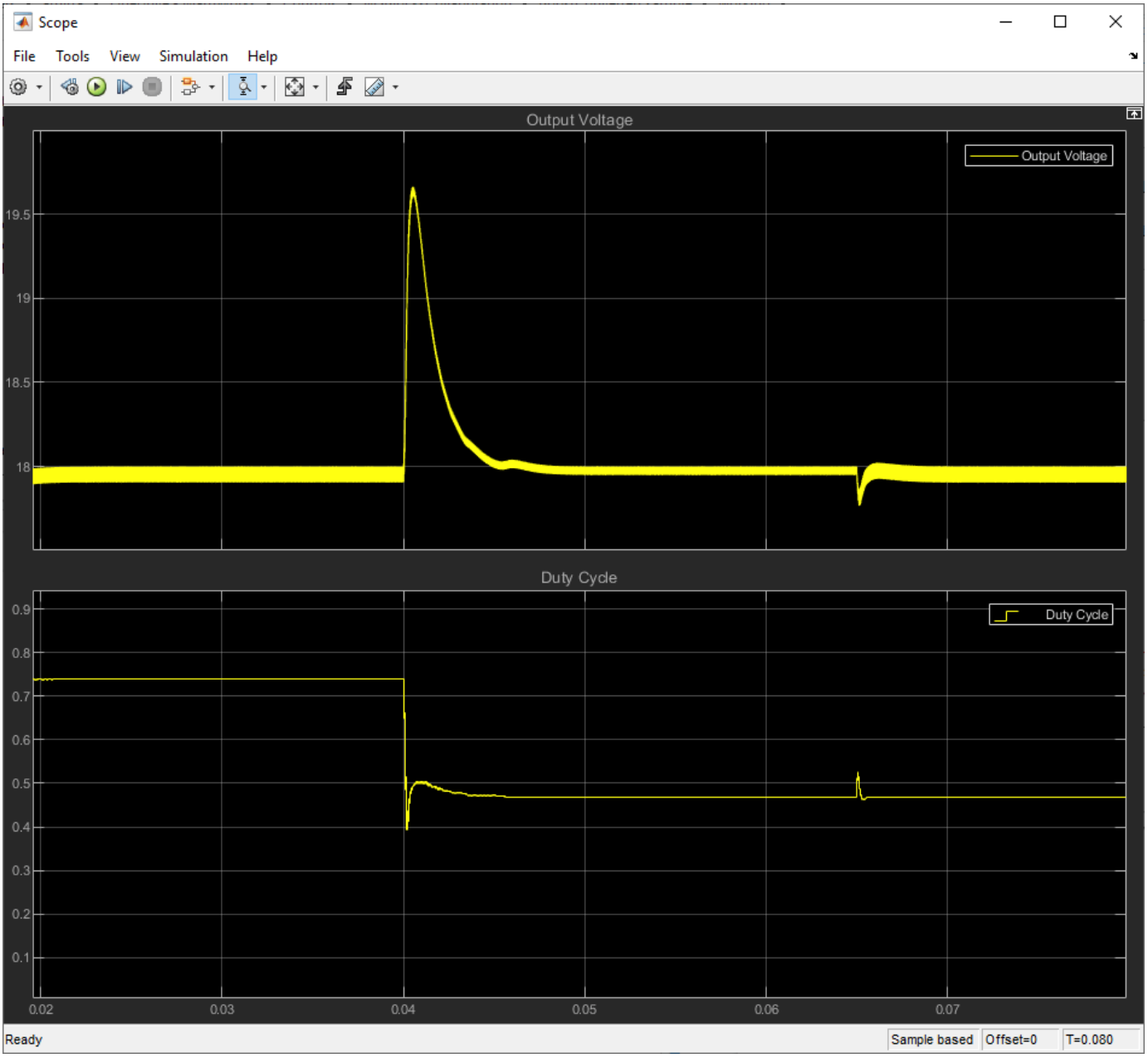
```
open_system("VoltageControlledBoostConverter.slx");
```

Use an LTI System Block to implement the type-III compensator and simulate the model to examine the performance. The model uses the following disturbances:

- Line disturbance at t = 0.04 seconds, which increases the input voltage `Vin` from 5 V to 10 V.
- Load disturbance at t = 0.065 seconds, which increases the load resistance `RLoad` from 3 ohms to 6 ohms.



The results show that the tuned feedback amplifier 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.

[2] 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. IEEE, March 2014. https://ieeexplore.ieee.org/document/6803449.

See Also

`systune`

Related Examples

- “Loop Shape and Stability Margin Specifications” on page 18-34

Control System Tuning Examples

- “Tuning Multiloop Control Systems” on page 18-2
- “PID Tuning for Setpoint Tracking vs. Disturbance Rejection” on page 18-11
- “Time-Domain Specifications” on page 18-20
- “Frequency-Domain Specifications” on page 18-26
- “Loop Shape and Stability Margin Specifications” on page 18-34
- “System Dynamics Specifications” on page 18-39
- “Configuring Design Requirements” on page 18-41
- “Validating Results” on page 18-42
- “Tune Control Systems in Simulink” on page 18-50
- “Tune a Control System Using Control System Tuner” on page 18-58
- “Using Parallel Computing to Accelerate Tuning” on page 18-72
- “Control of a Linear Electric Actuator” on page 18-76
- “Control of a Linear Electric Actuator Using Control System Tuner” on page 18-85
- “Multi-Loop PI Control of a Robotic Arm” on page 18-110
- “Control of an Inverted Pendulum on a Cart” on page 18-125
- “Digital Control of Power Stage Voltage” on page 18-132
- “MIMO Control of Diesel Engine” on page 18-141
- “Tuning of a Two-Loop Autopilot” on page 18-154
- “Multiloop Control of a Helicopter” on page 18-169
- “Fixed-Structure Autopilot for a Passenger Jet” on page 18-176
- “Fault-Tolerant Control of a Passenger Jet” on page 18-187
- “Passive Control of Water Tank Level” on page 18-196
- “Tuning for Multiple Values of Plant Parameters” on page 18-206

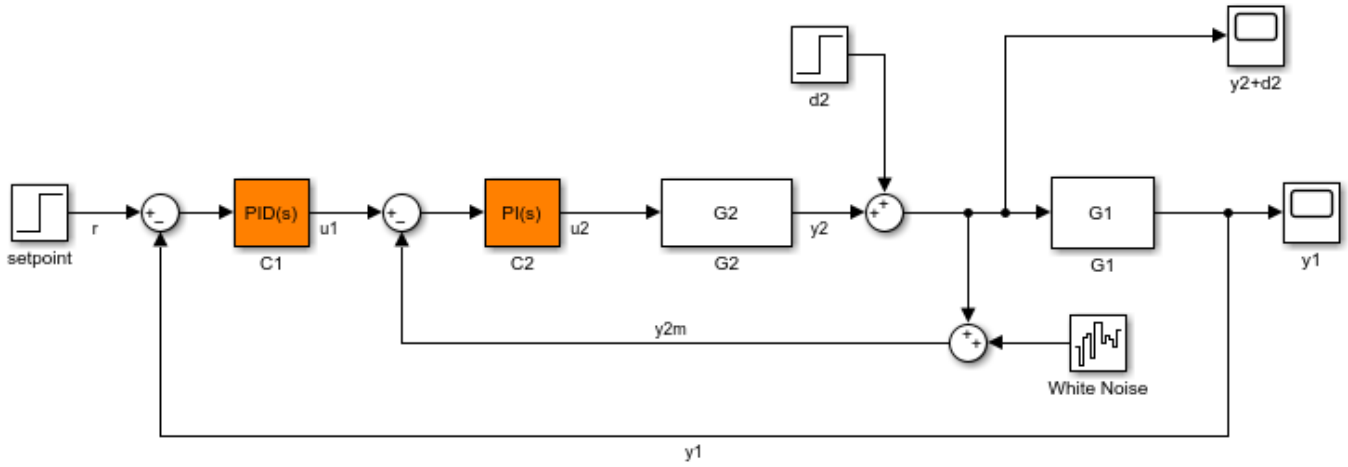
Tuning Multiloop Control Systems

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 = 0.861, Hard = -Inf, Iterations = 55
```

Use `showTunable` to see the tuned PID gains.

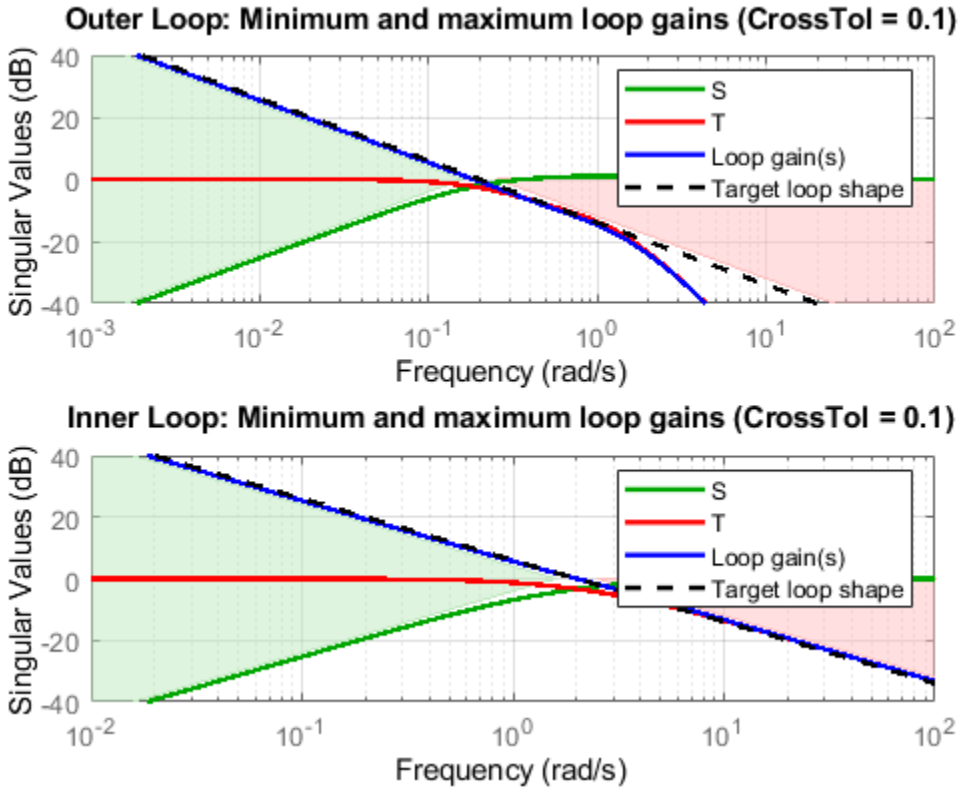
```
showTunable(ST)
```

```
Block 1: rct_cascade/C1 =

             1             s
  Kp + Ki * --- + Kd * --------
             s          Tf*s+1

  with Kp = 0.0518, Ki = 0.0186, Kd = 0.0472, Tf = 0.0228

Name: C1
Continuous-time PIDF controller in parallel form.

----------------------------------

Block 2: rct_cascade/C2 =

             1
  Kp + Ki * ---
             s

  with Kp = 0.722, Ki = 1.23

Name: C2
Continuous-time PI controller in parallel form.
```

Validating the Design

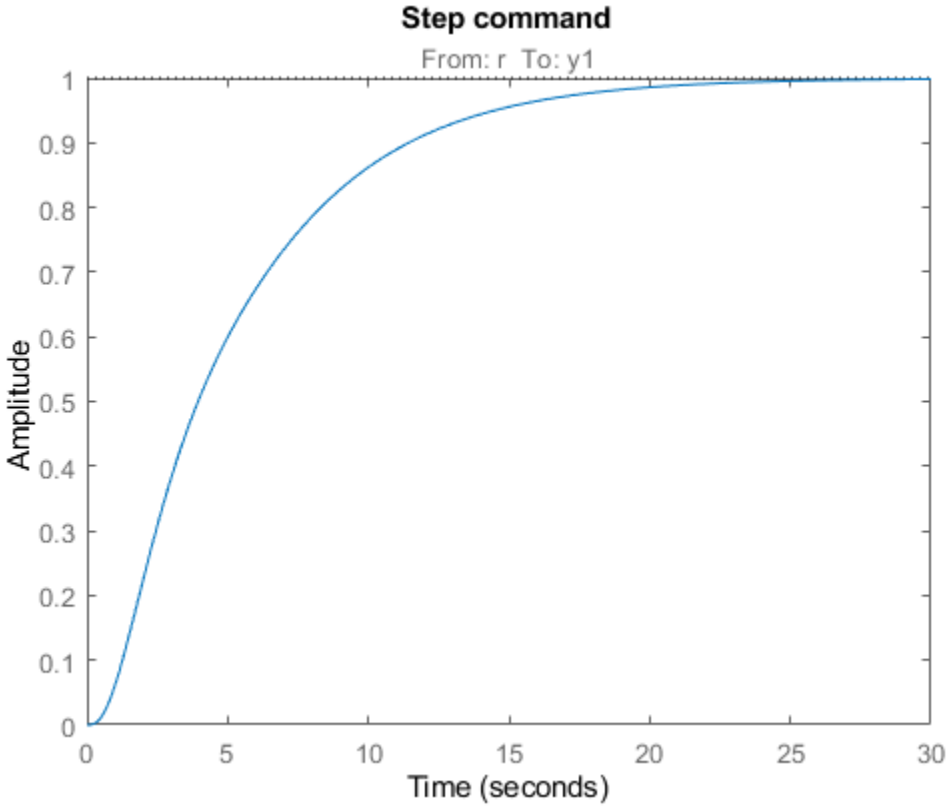
The final value is less than 1 which means that `systune` successfully met both loop shape requirements. Confirm this by inspecting the tuned control system `ST` with `viewGoal`

```
viewGoal([Req1,Req2],ST)
```

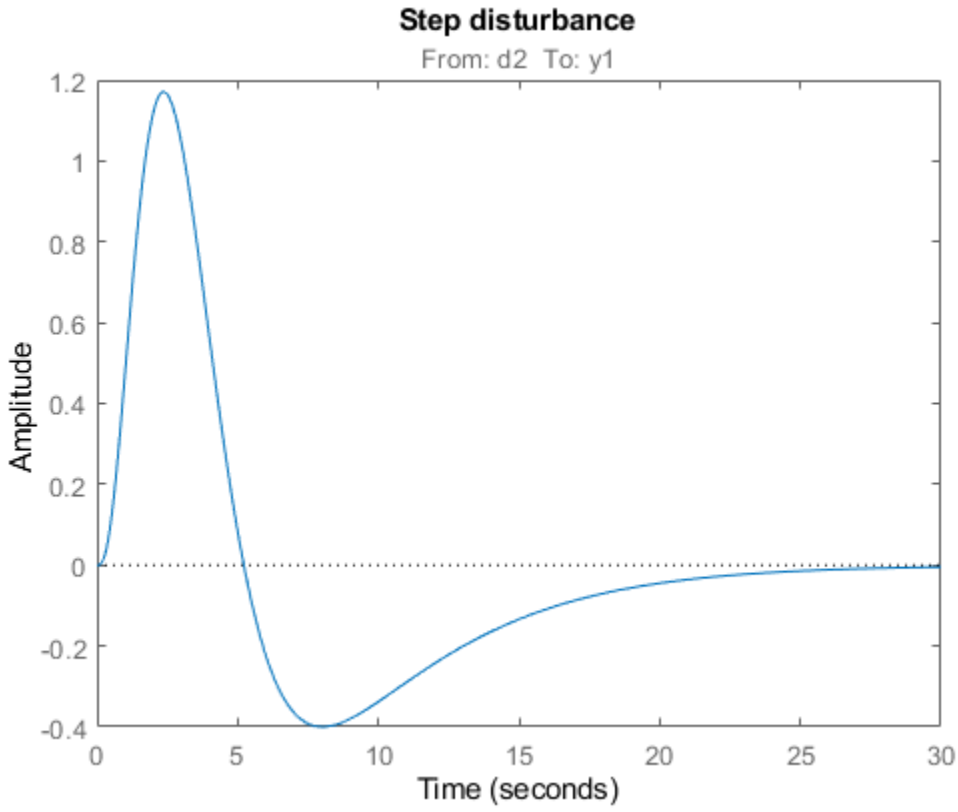


Note that the inner and outer loops have the desired gain crossover frequencies. To further validate the design, plot the tuned responses to a step command r and step disturbance d2:

```
% 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.

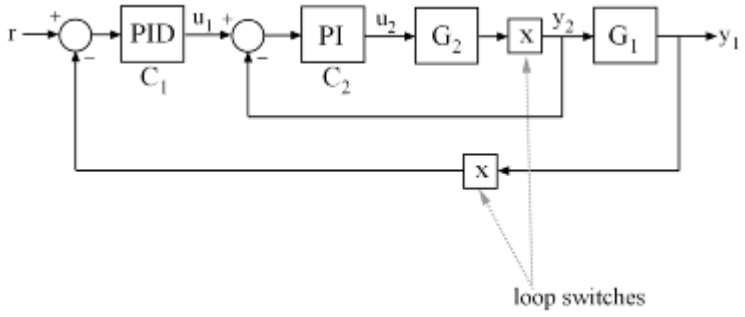


Figure 1: Cascade Architecture

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';
```

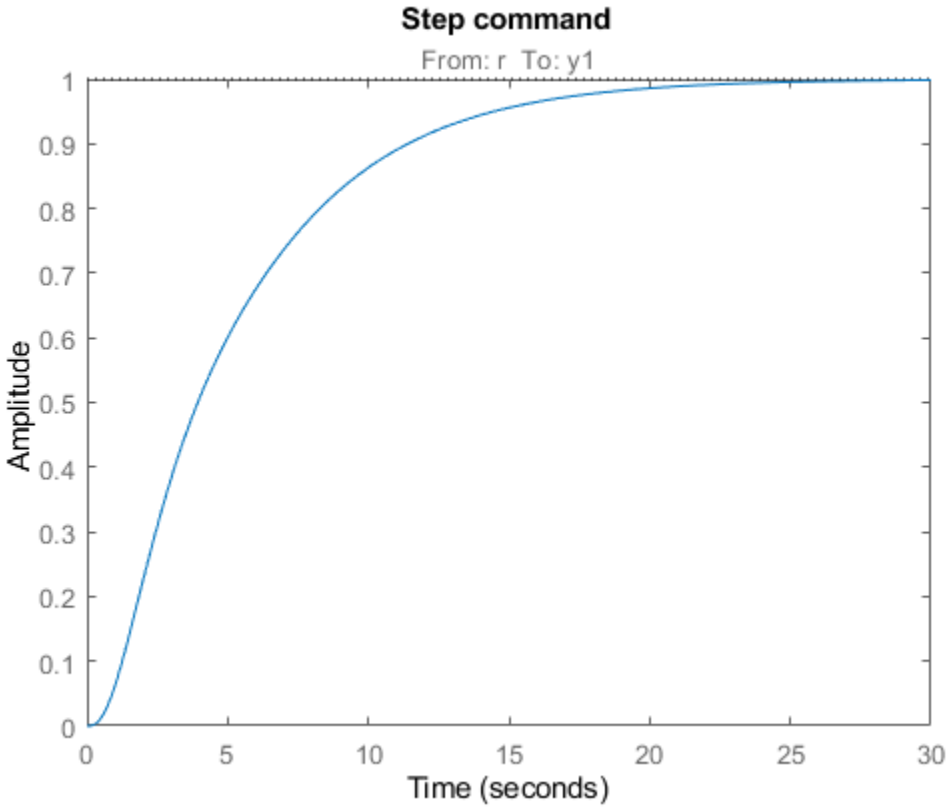
You can now tune the PID gains in `C1` and `C2` with `systune`.

```
T = systune(T0,[Req1,Req2]);
```

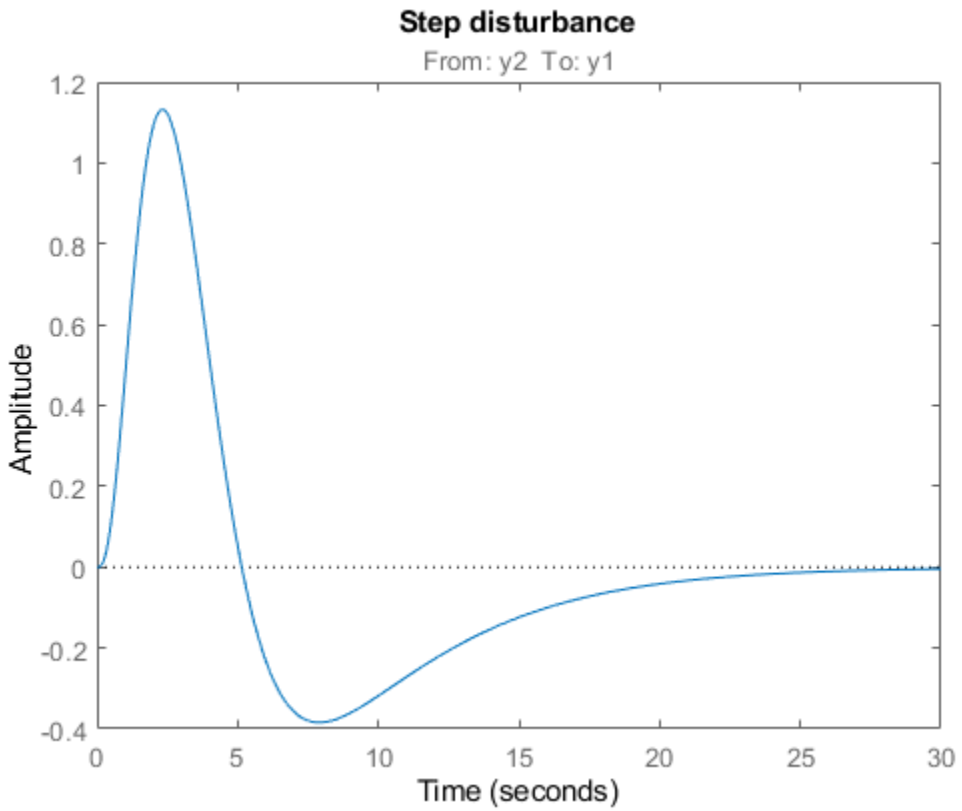
```
Final: Soft = 0.86, Hard = -Inf, Iterations = 139
```

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')
```

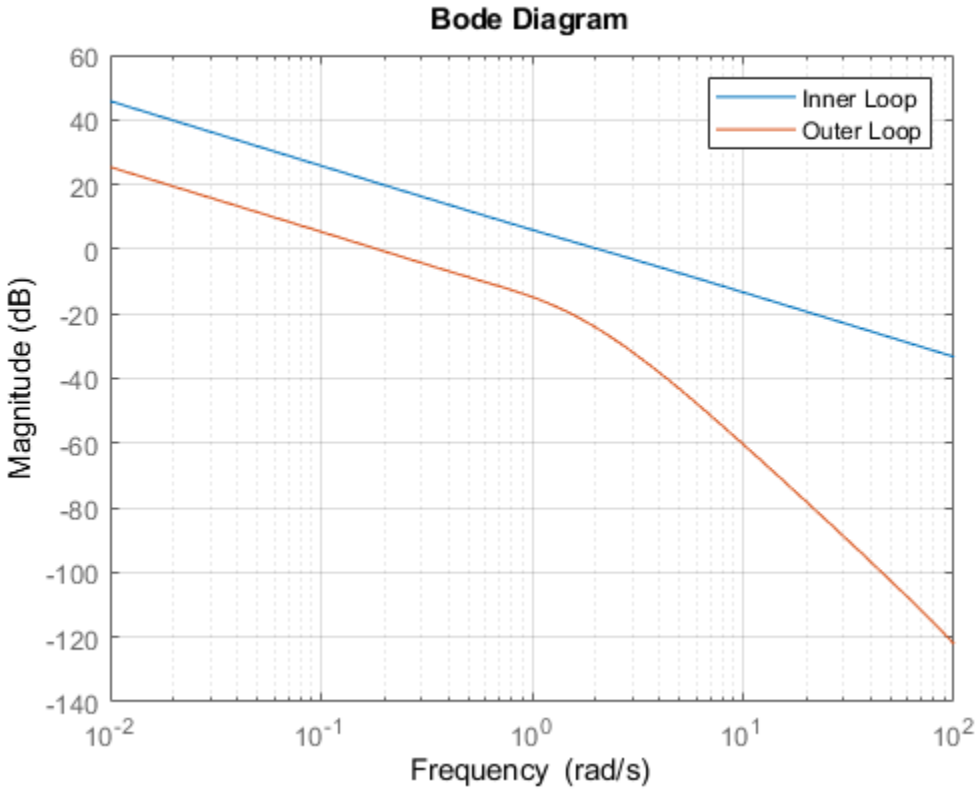


```
% Response to a step disturbance
H = getIOTransfer(T,'y2','y1');
step(H,30), title('Step disturbance')
```



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” on page 18-11

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 jrequires 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.

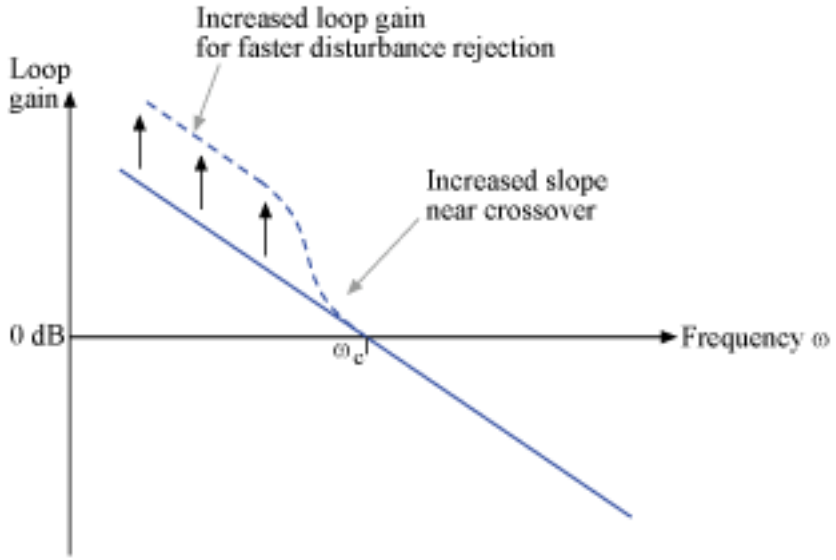


Figure 1: Trade-off in 1-DOF PID Tuning.

This example uses `systune` to explore this trade-off and find the right compromise for your application. You can also use the **PID Tuner** app to make such a trade-off. For more information, see “Tune PID Controller to Favor Reference Tracking or Disturbance Rejection (PID Tuner)”.

Tuning Setup

Consider the PID loop of Figure 2 with a load disturbance at the plant input.

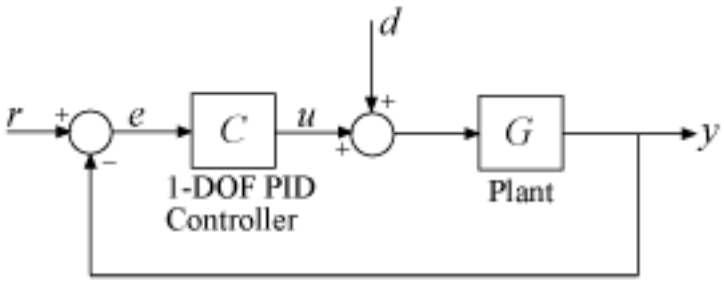


Figure 2: PID Control Loop.

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

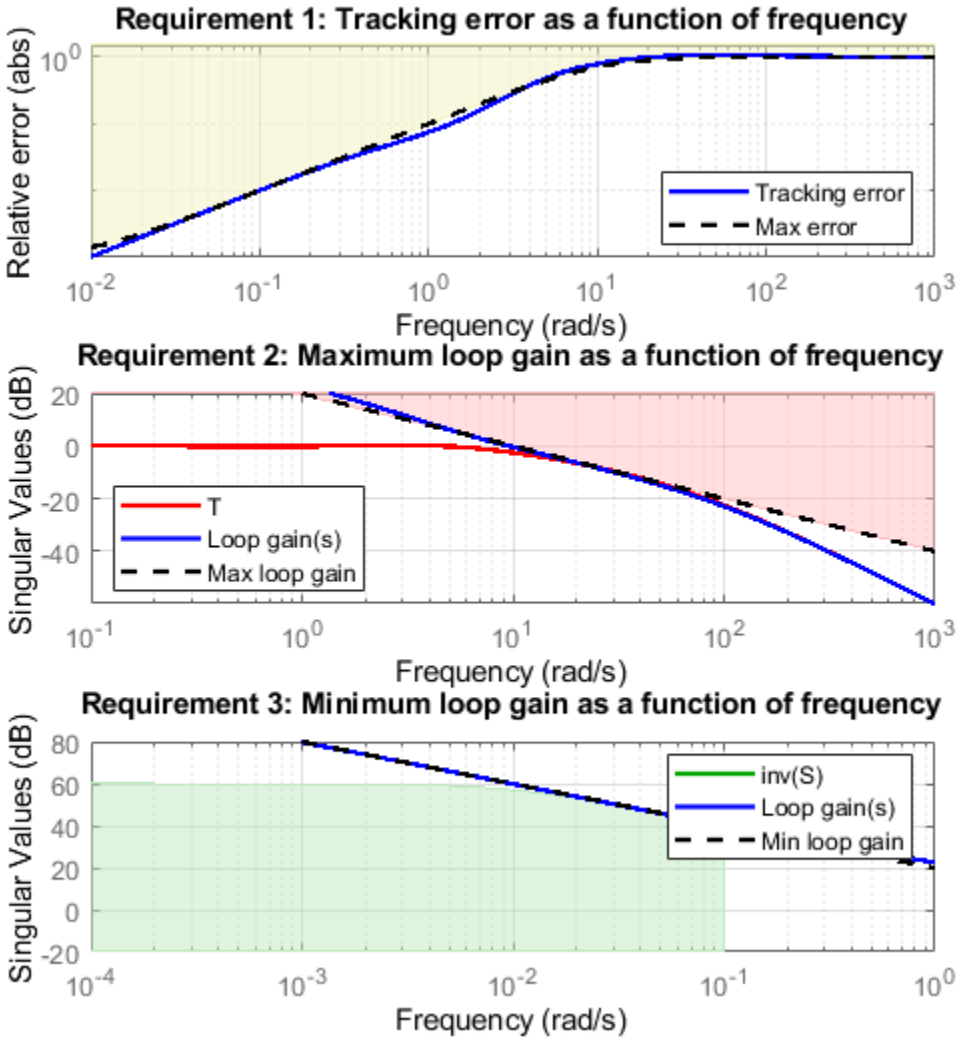
Use `systune` to tune the PID gains to meet these requirements. Treat the bandwidth and disturbance rejection goals as hard constraints and optimize tracking subject to these constraints.

```
T1 = systune(T0,R1,[R2 R3]);
```

```
Final: Soft = 1.12, Hard = 0.9998, Iterations = 157
```

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]);
```

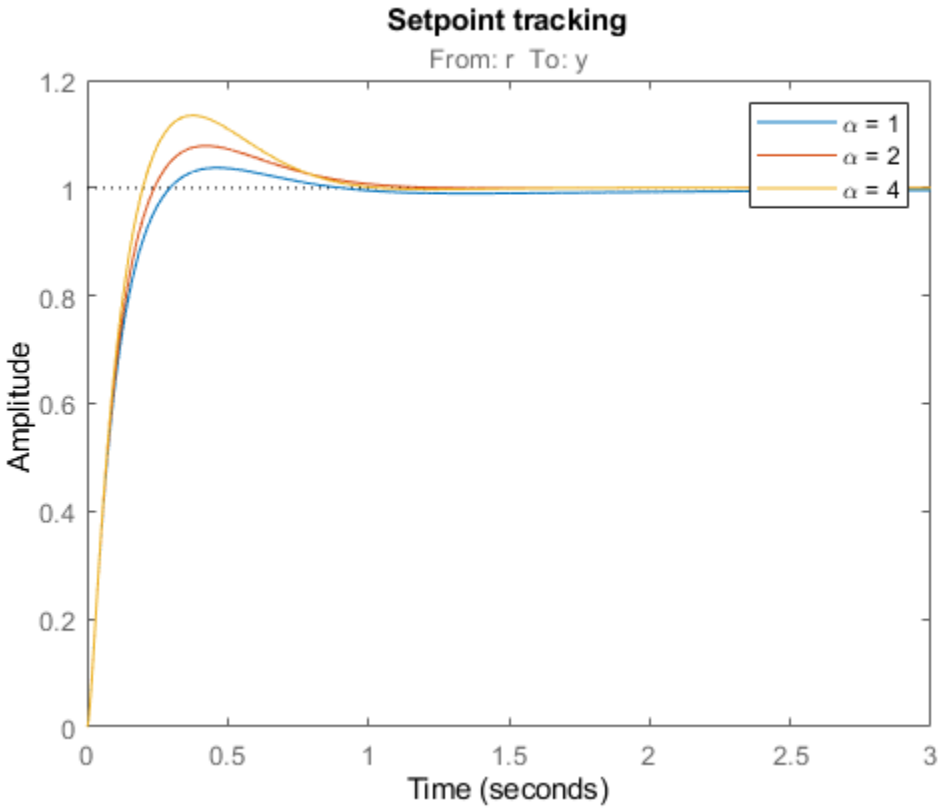
```
Final: Soft = 1.17, Hard = 0.99954, Iterations = 115
```

```
% 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
```

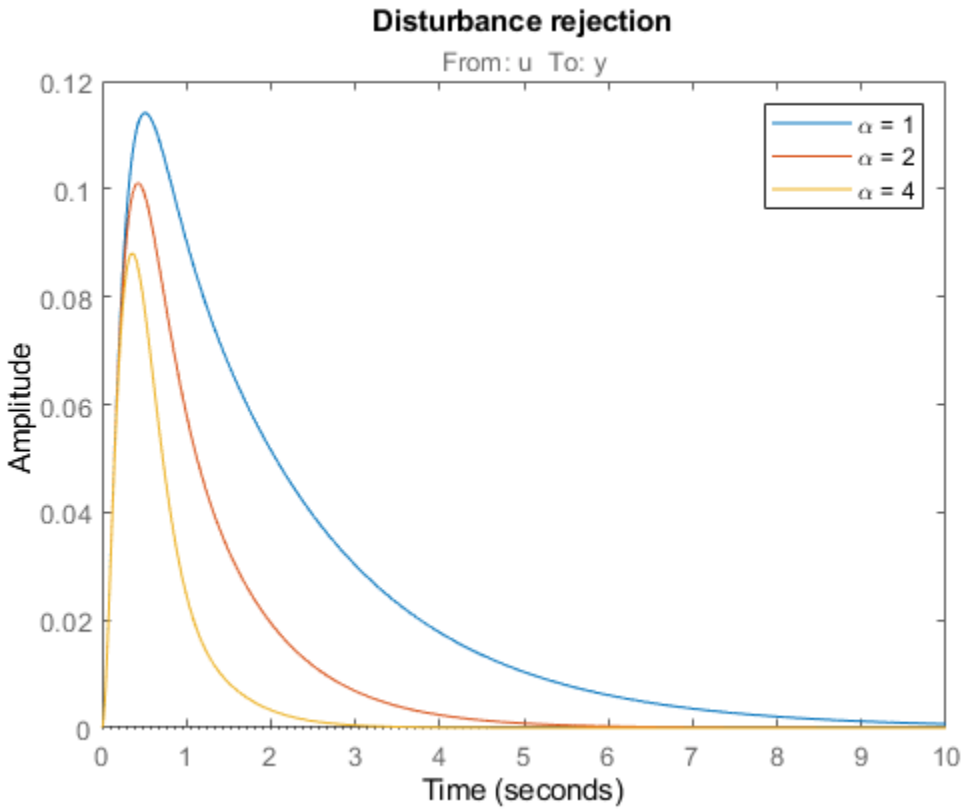
Compare the responses to a step command `r` and to a step disturbance `d` entering at the plant input `u`.

```
figure, step(T1,T2,T3,3)
title('Setpoint tracking')
legend('\alpha = 1','\alpha = 2','\alpha = 4')
```



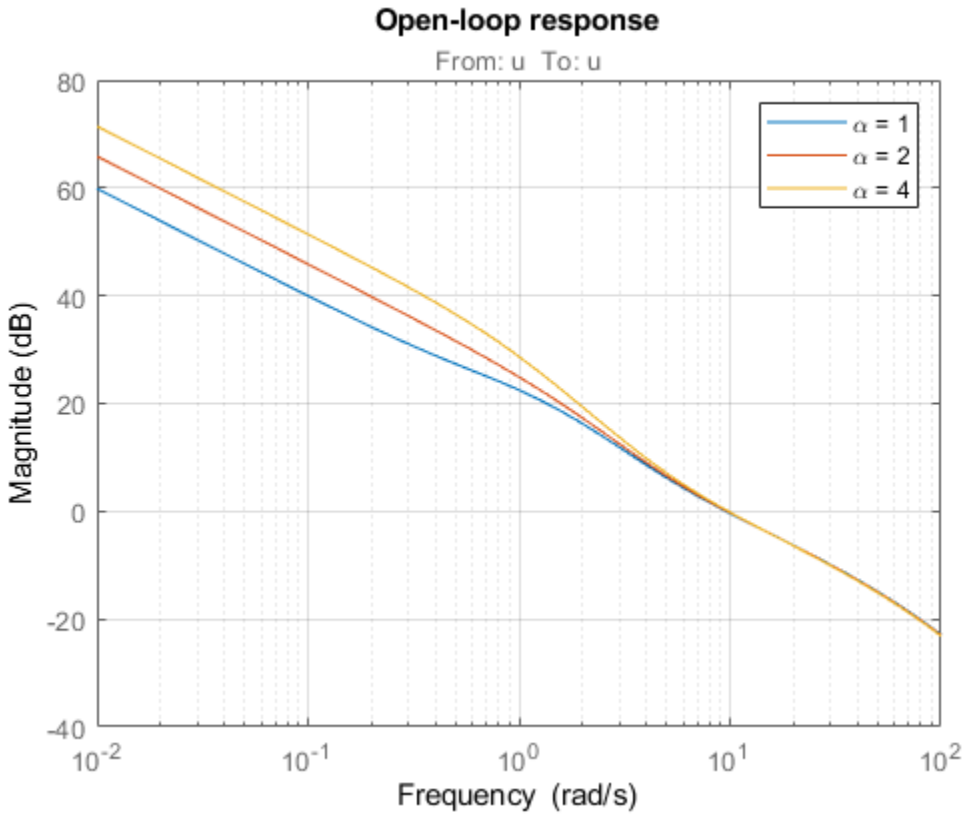

```
% Compute closed-loop transfer from u to y
D1 = getIOTransfer(T1,'u','y');
```

```
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.

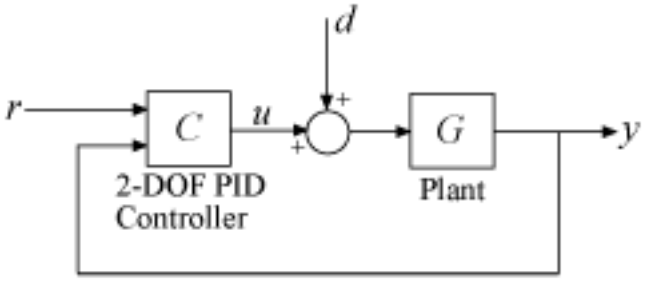


Figure 3: 2-DOF PID Control Loop.

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';
```

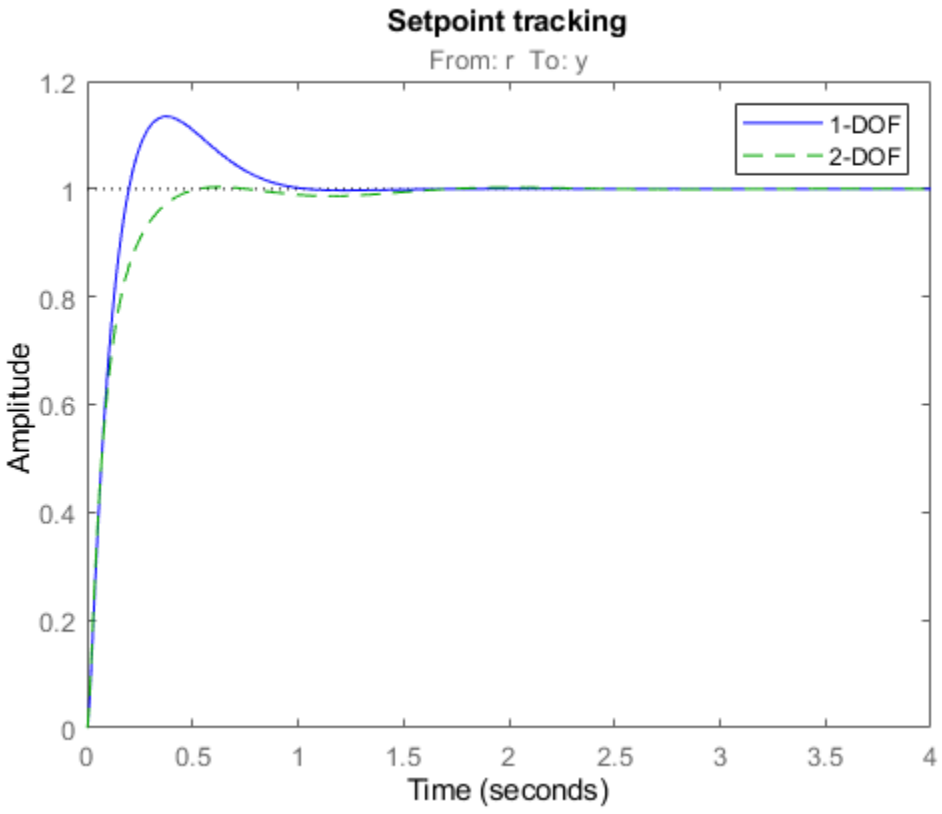
Next tune the 2-DOF PI controller for the largest loop gain tried earlier ($\alpha = 4$).

```
% 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]);
```

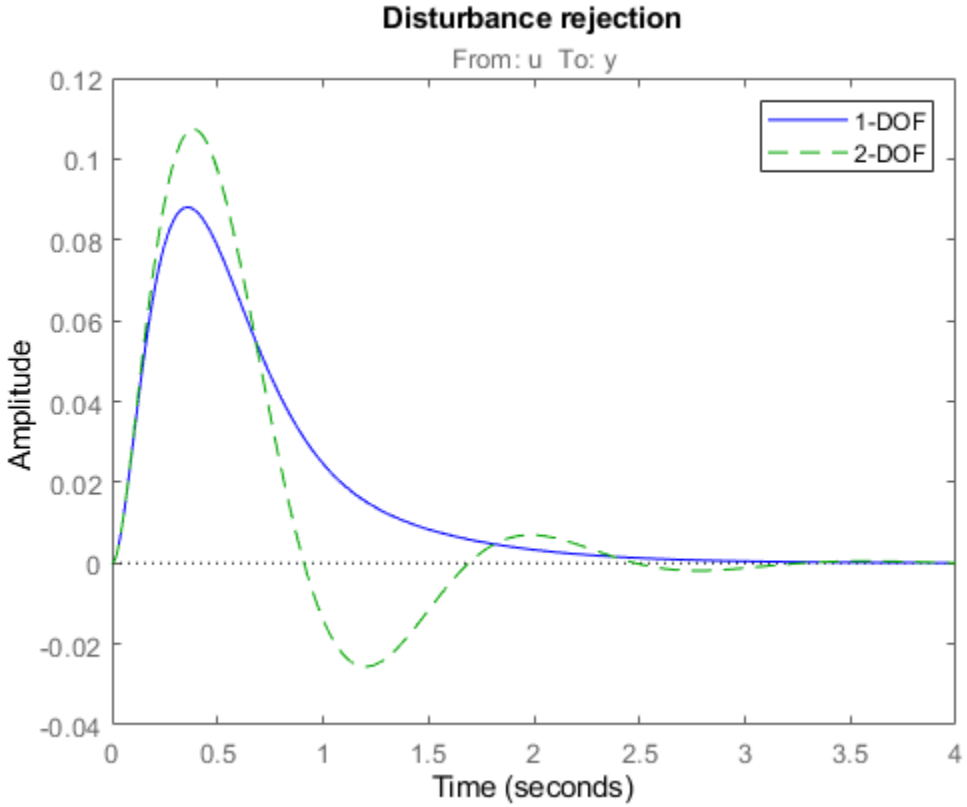
```
Final: Soft = 1.09, Hard = 0.8969, Iterations = 74
```

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 = 5.97, Ki = 21.4, Kd = 0.877, Tf = 0.01, b = 0.676, c = 1.26
```

```
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 18-110

Time-Domain Specifications

This example gives a tour of available time-domain requirements for control system tuning with `systune` or `looptune`.

The `systune` and `looptune` functions tune the parameters of fixed-structure control systems subject to a variety of time- and frequency-domain requirements. To specify these requirements, use tuning goal objects.

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,

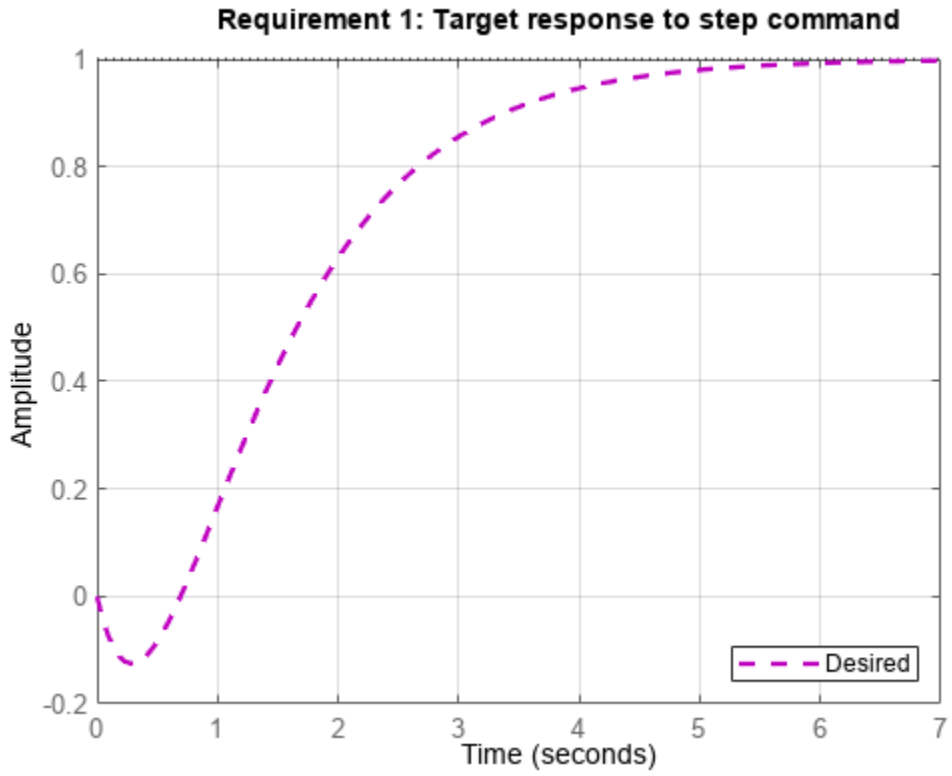
```
R1 = TuningGoal.StepTracking('r','y',0.5);
```

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.

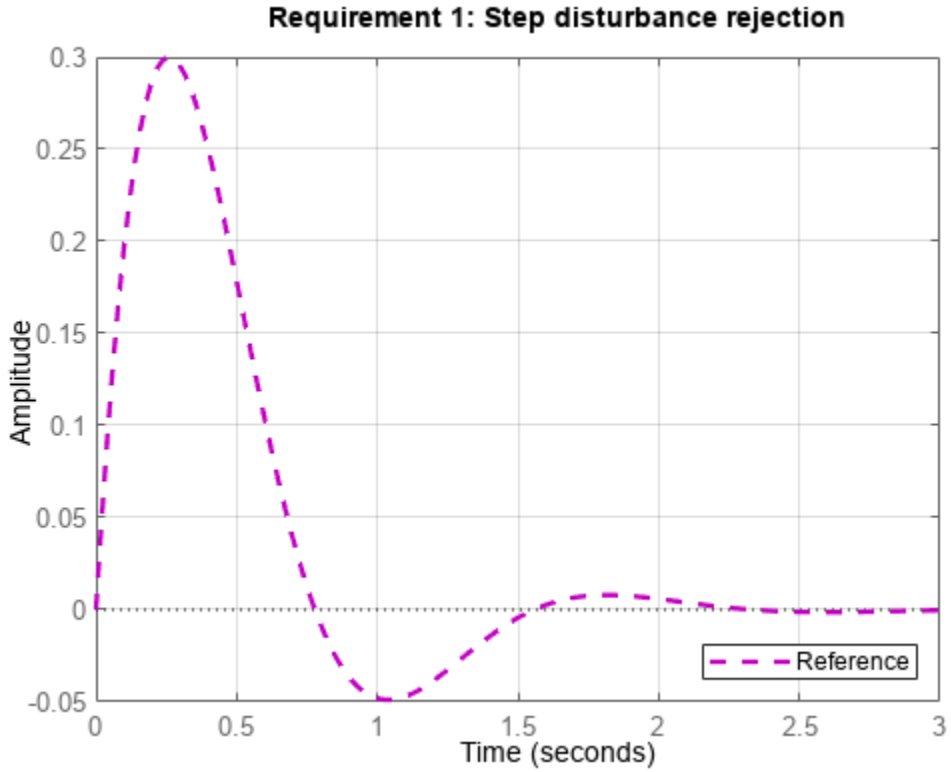
Step Disturbance Rejection

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).

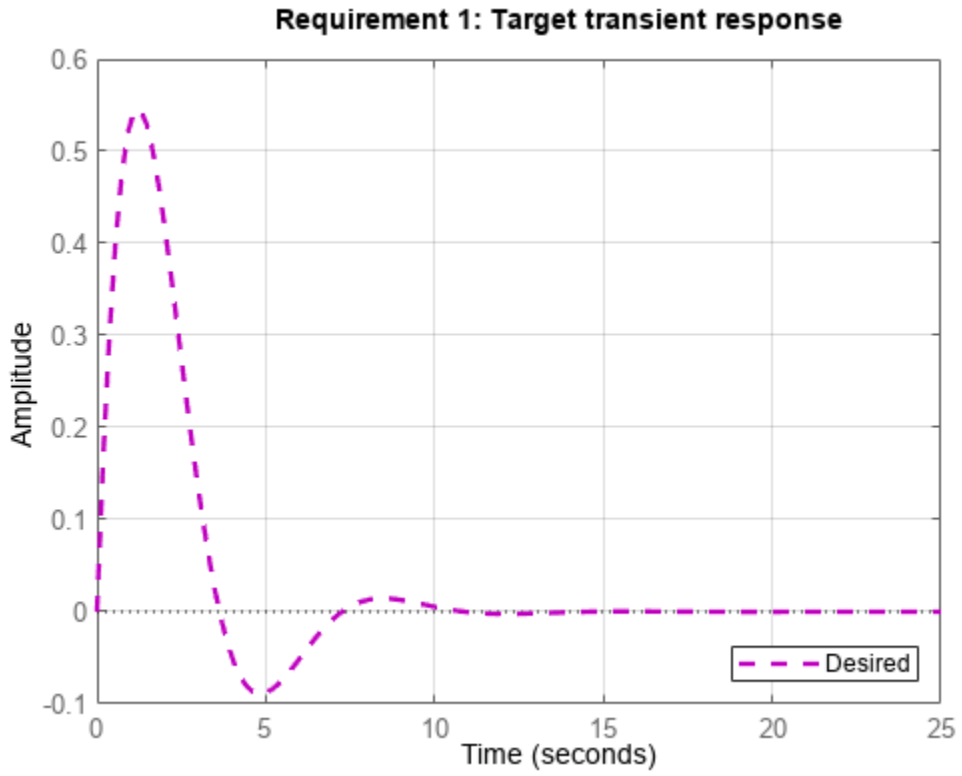
Transient Response Matching

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');
```

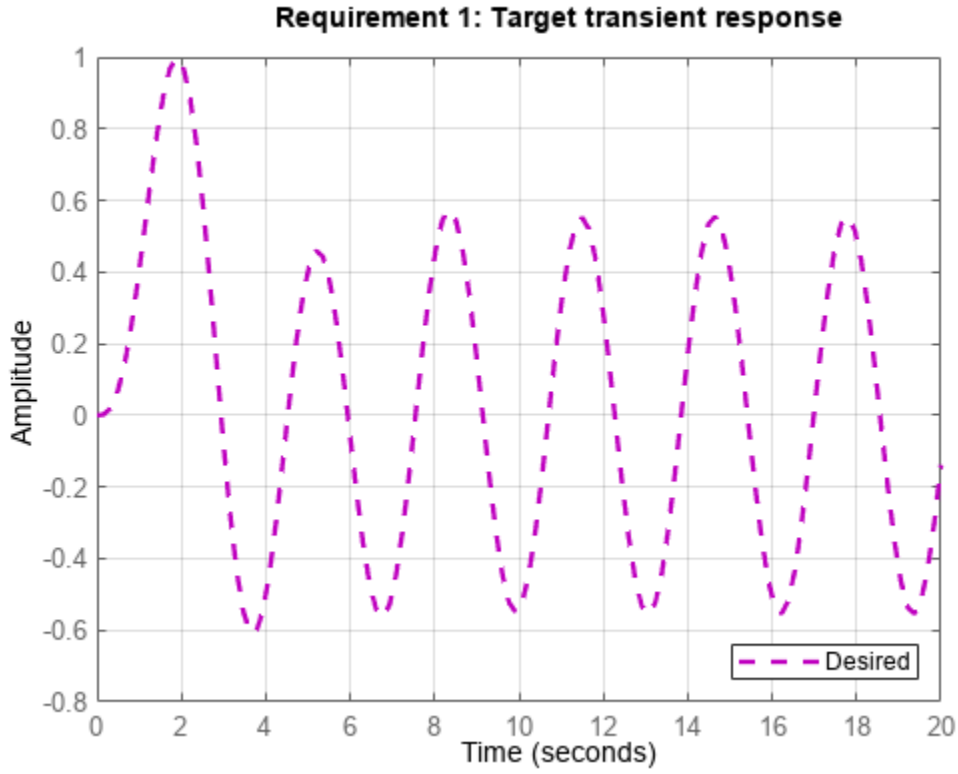
requires that the tuned response from r to y look like the impulse response of the reference model $1/(s^2+s+1)$.

```
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.

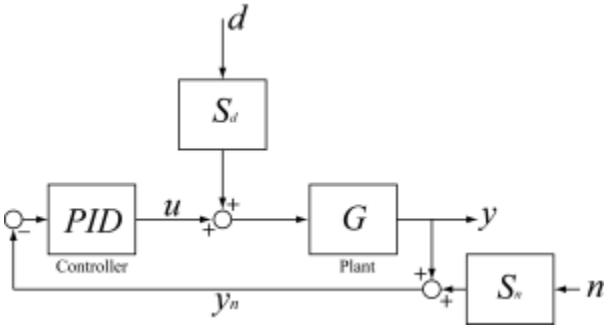


Figure 2: Regulation loop.

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” on page 18-26

Frequency-Domain Specifications

This example shows the available frequency-domain requirements for control system tuning with `systune` or `looptune`.

The `systune` and `looptune` functions tune the parameters of fixed-structure control systems subject to a variety of time- and frequency-domain requirements. To specify these requirements, use tuning goal objects.

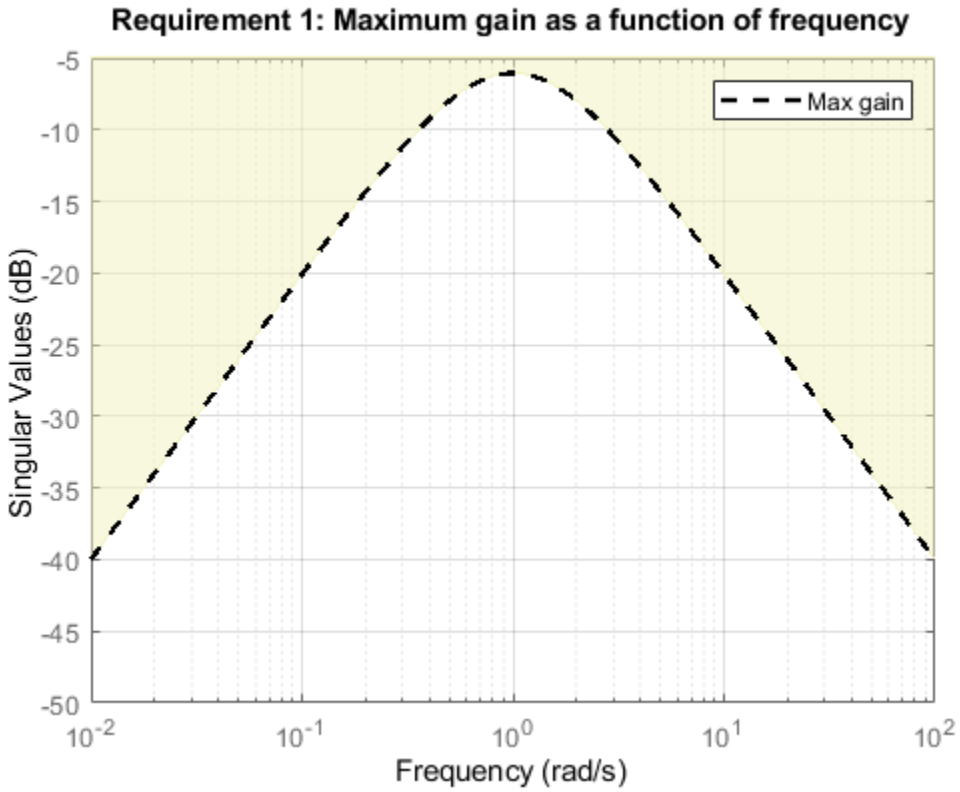
Gain Limit

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)
```

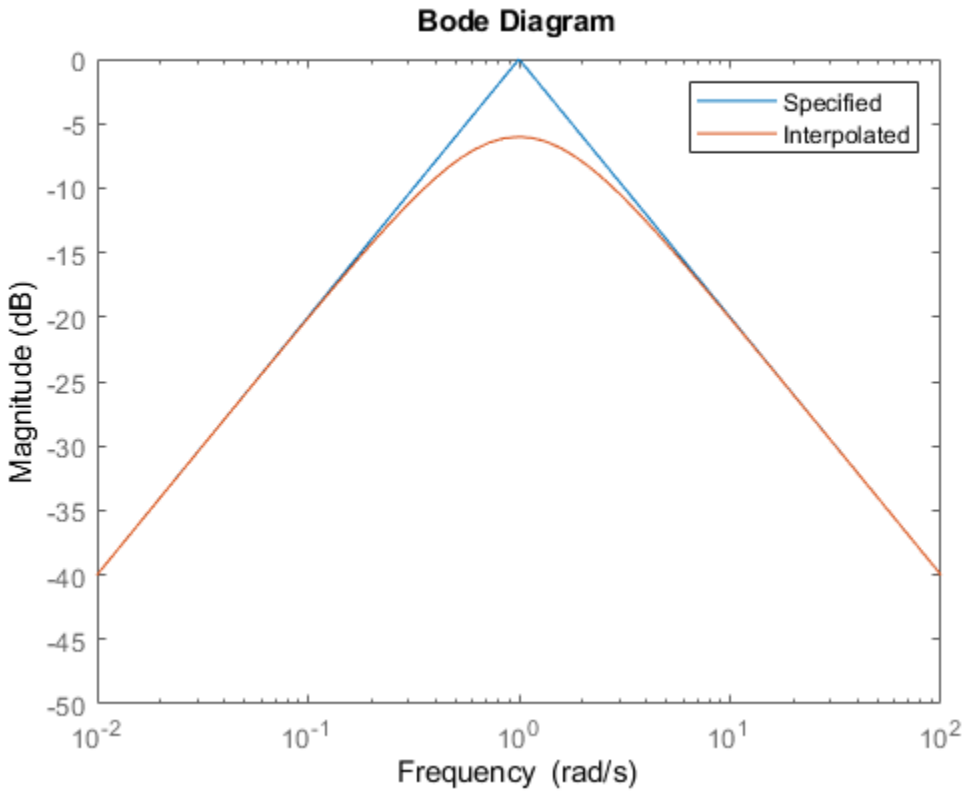


It is often convenient to just sketch the asymptotes of the desired gain profile. For example, instead of the transfer function $s/(s+1)^2$, we could just specify gain values of 0.01,1,0.01 at the frequencies 0.01,1,100, the point (1,1) being the breakpoint of the two asymptotes s and $1/s$.

```
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

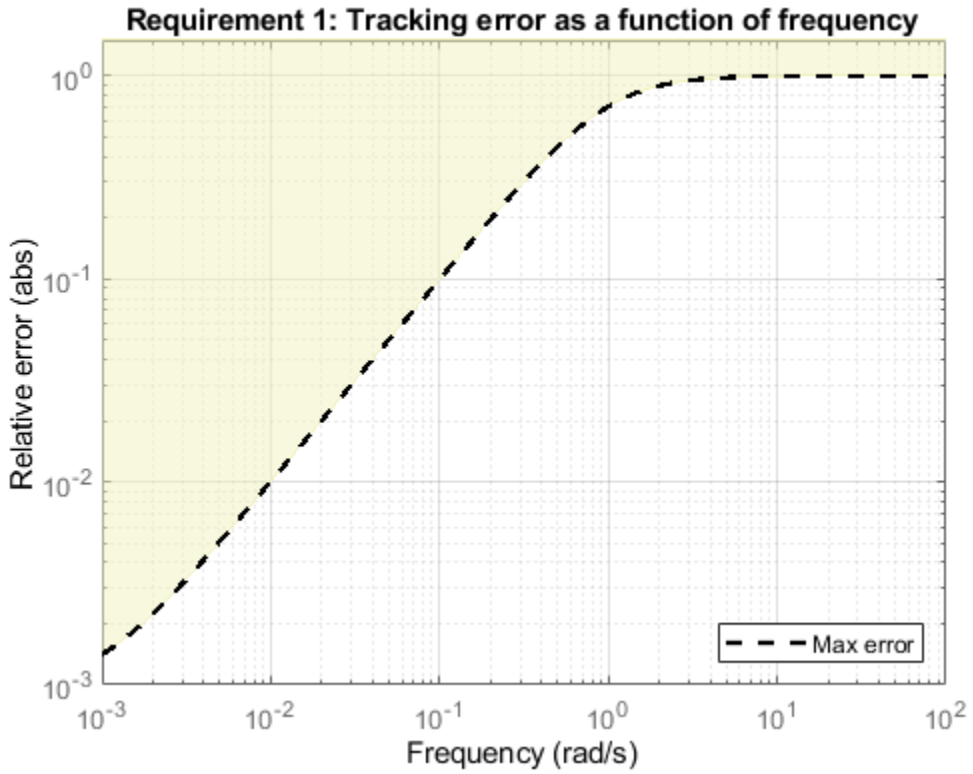
```
R1 = TuningGoal.Tracking('r','y',2);
```

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.

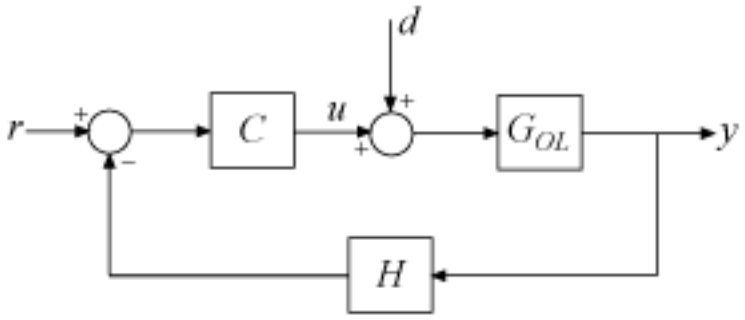


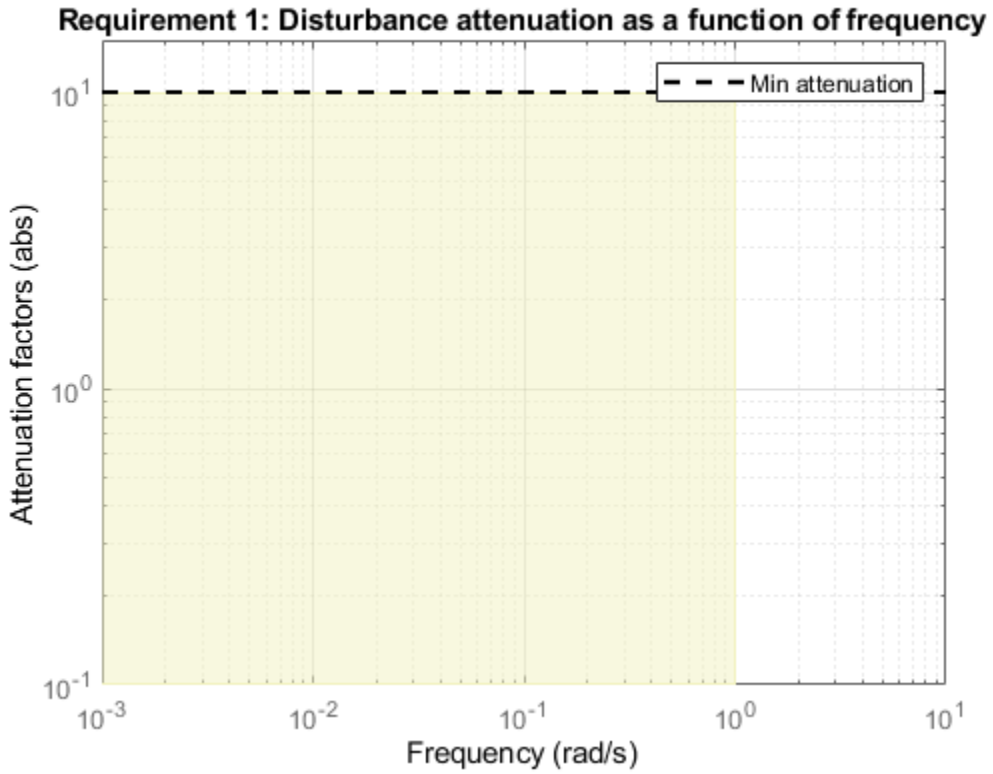
Figure 1: Sample feedback loop.

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)
```

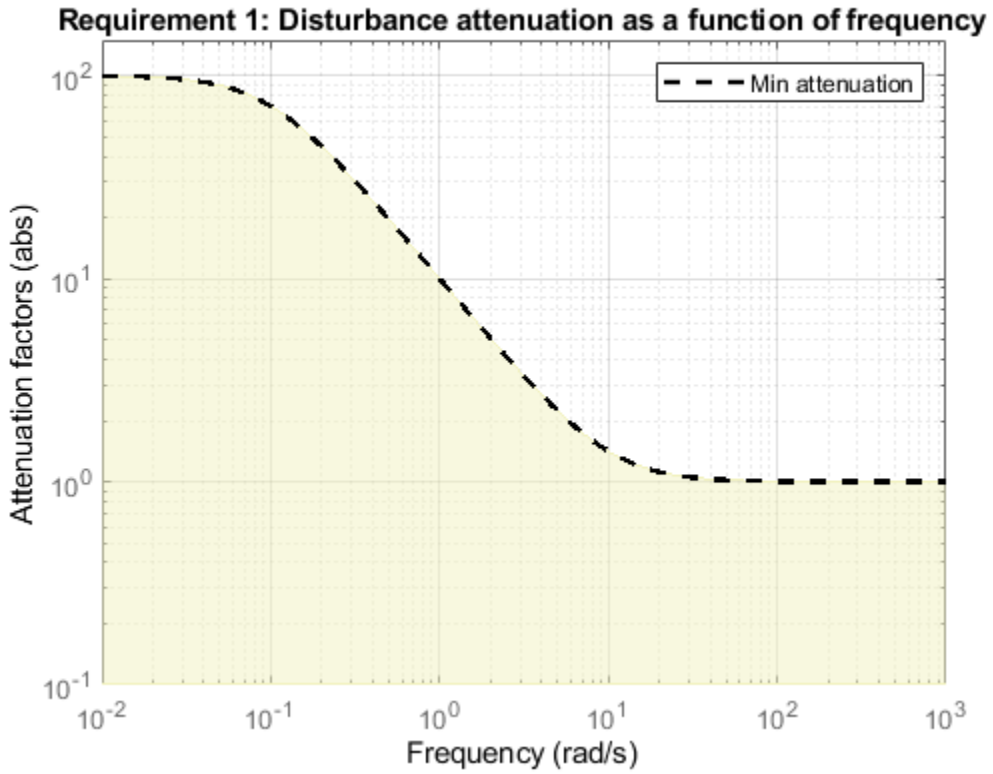


More generally, you can specify a frequency-dependent attenuation profile, for example

```
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)
```

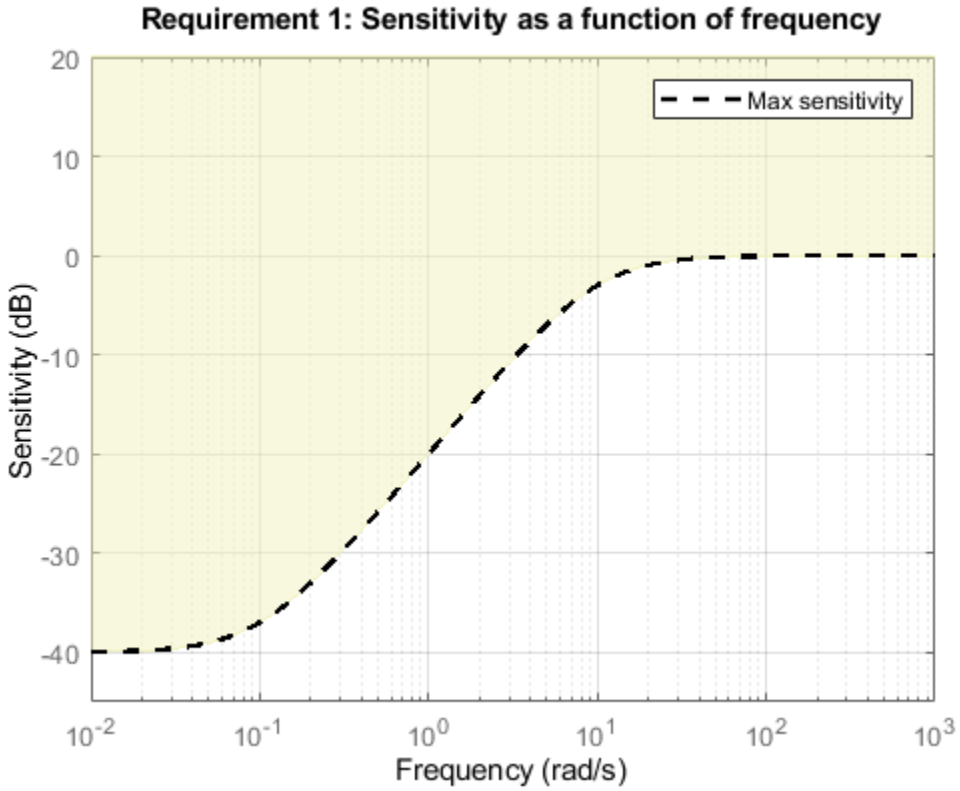


Instead of specifying the minimum attenuation, you can use the `TuningGoal.Sensitivity` requirement to specify the maximum sensitivity, that is, the maximum gain of $S = 1/(1+L)$. For example,

```
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)
```

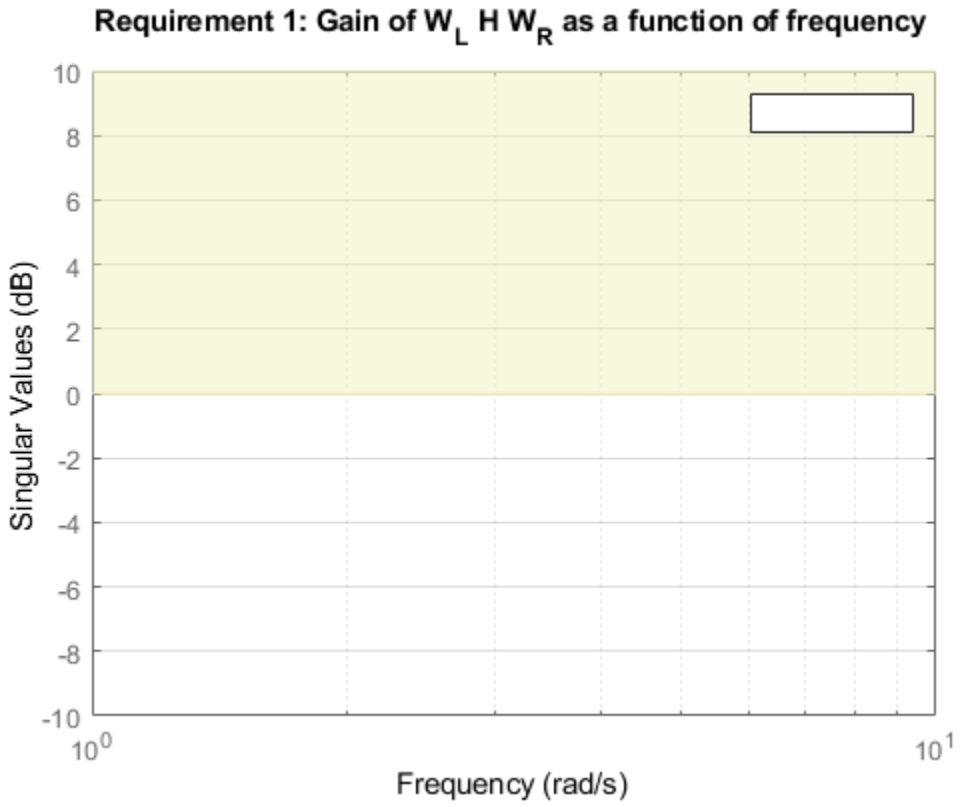


Frequency-Weighted Gain and Variance

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, specify following normalized gain constraint.

$$\left\| \begin{pmatrix} \frac{1}{s+0.001} T_{re} \\ \frac{s}{0.001s+1} T_{ry} \end{pmatrix} \right\|_\infty < 1$$

```
WL = blkdiag(1/(s+0.001),s/(0.001*s+1));
WR = [];
R = TuningGoal.WeightedGain('r',{'e','y'},WL,[]);

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” on page 18-20
- “Loop Shape and Stability Margin Specifications” on page 18-34

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`.

The `systune` and `looptune` functions tune the parameters of fixed-structure control systems subject to a variety of time- and frequency-domain requirements. To specify these design requirements, use tuning goal objects.

Loop Shape

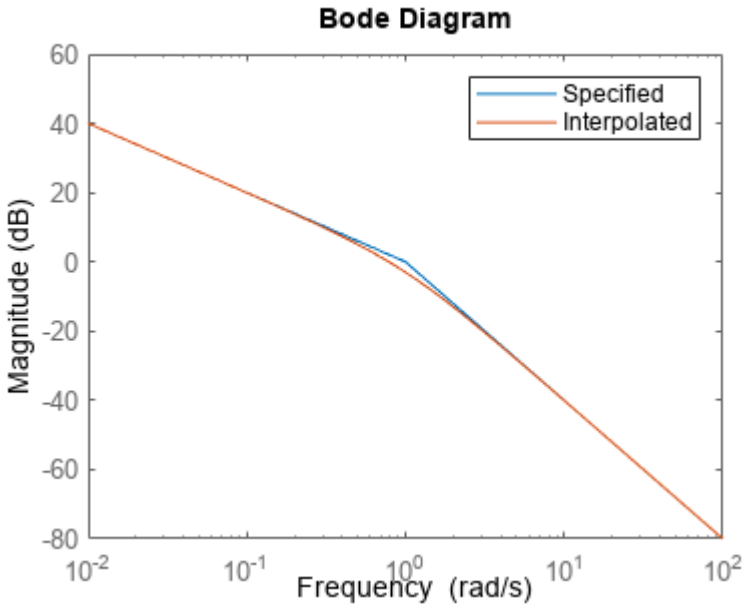
The `TuningGoal.LoopShape` requirement is used to shape the open-loop response gain(s), a design approach known as *loop shaping*. For example,

```
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.

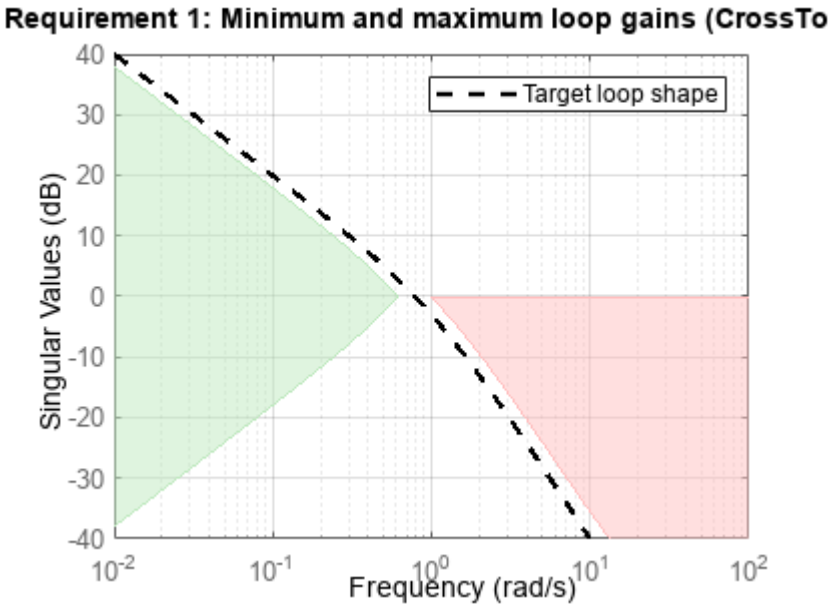
As with other gain specifications, you can just specify the asymptotes of the desired loop shape using a few frequency points. For example, to specify a loop shape with gain crossover at 1 rad/s, -20 dB/decade slope before 1 rad/s, and -40 dB/decade slope after 1 rad/s, just specify that the gain at 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).

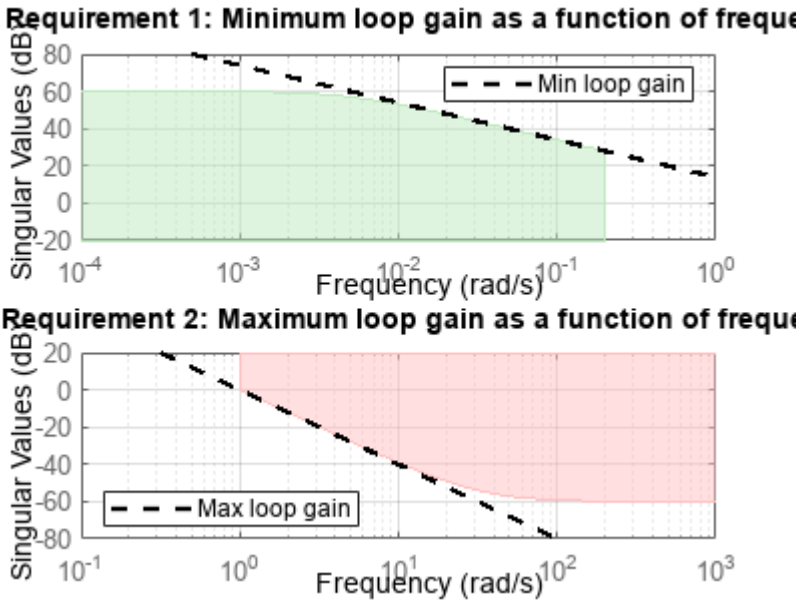
```
viewGoal(R2)
```



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

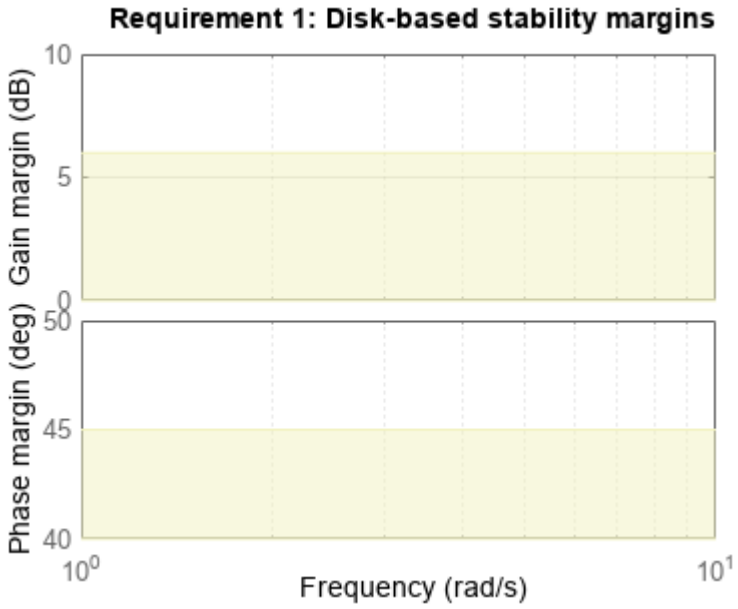
The `TuningGoal.Margins` requirement uses the notion of disk margin to enforce minimum amounts of gain and phase margins at the specified loop opening site(s). For MIMO feedback loops, this requirement guarantees stability for gain or phase variations in each feedback channel. The gain or phase can change in all channels simultaneously, and by a different amount in each channel. See “Stability Margins in Control System Tuning” on page 14-161 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” on page 17-9 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” (Simulink Control Design)). 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” on page 14-161
- “Frequency-Domain Specifications” on page 18-26

System Dynamics Specifications

This example shows how to constrain the poles of a control system tuned with `systune` or `looptune`.

The `systune` and `looptune` functions tune the parameters of fixed-structure control systems subject to a variety of time- and frequency-domain requirements. To specify these design requirements, use tuning goal objects.

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

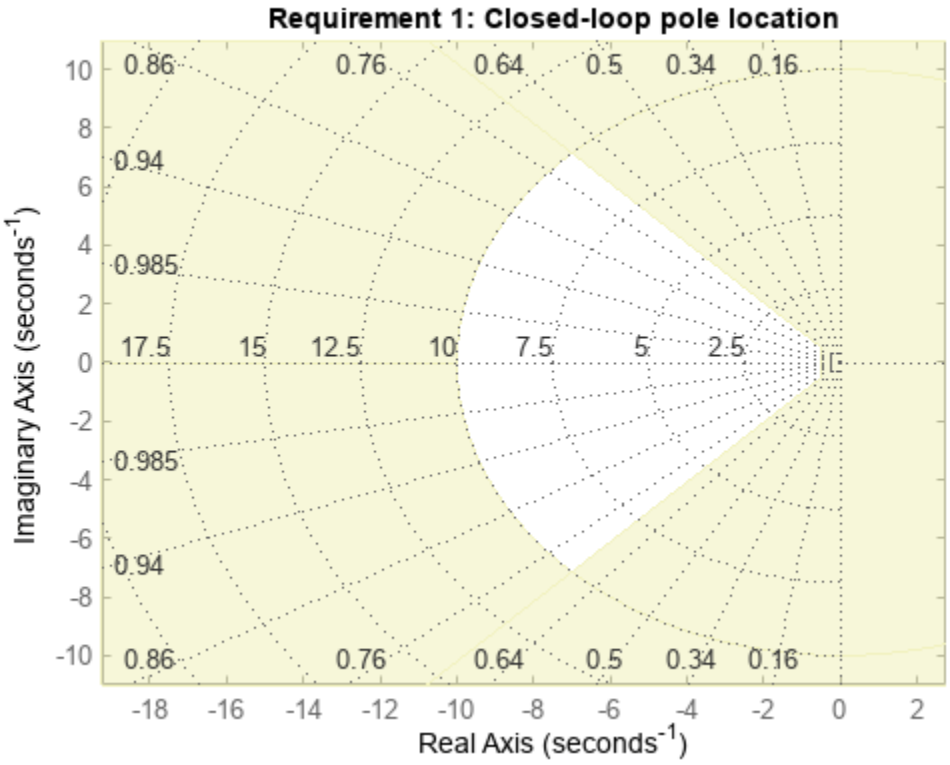
$$|s| < \omega_{\max}.$$

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” on page 18-34

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`.

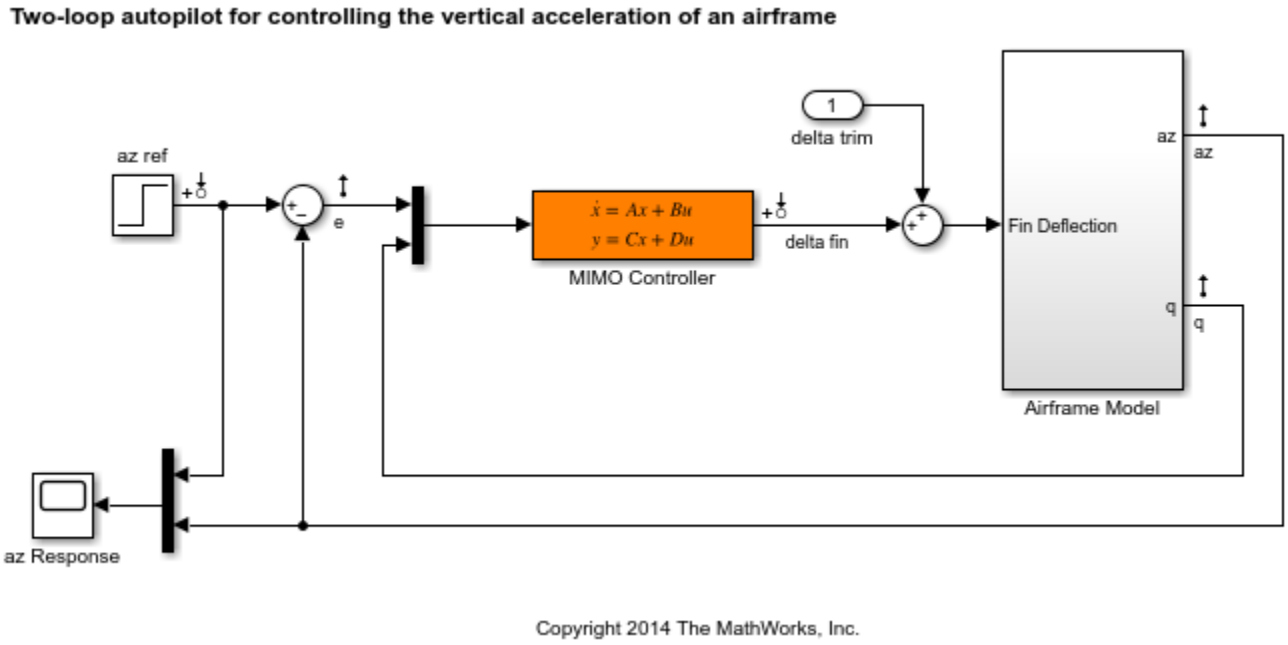
Background

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 = 105
Final: Soft = 1.15, Hard = -Inf, Iterations = 71
Final: Soft = 1.15, Hard = -Inf, Iterations = 111
```

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
```

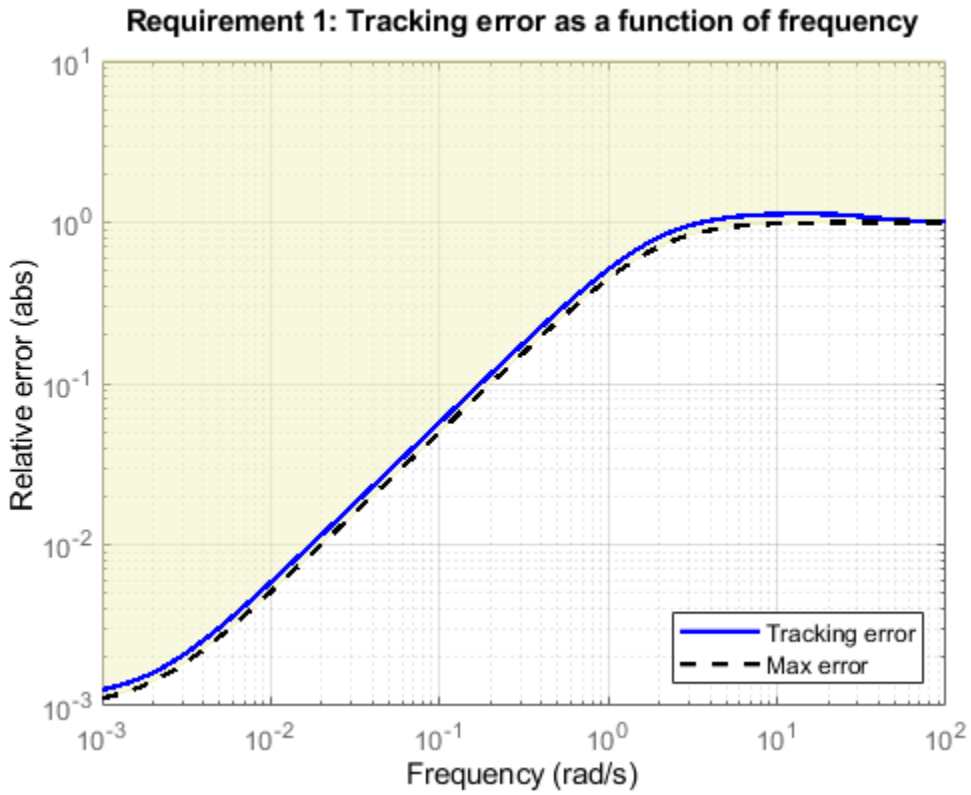
```
fSoft =

    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

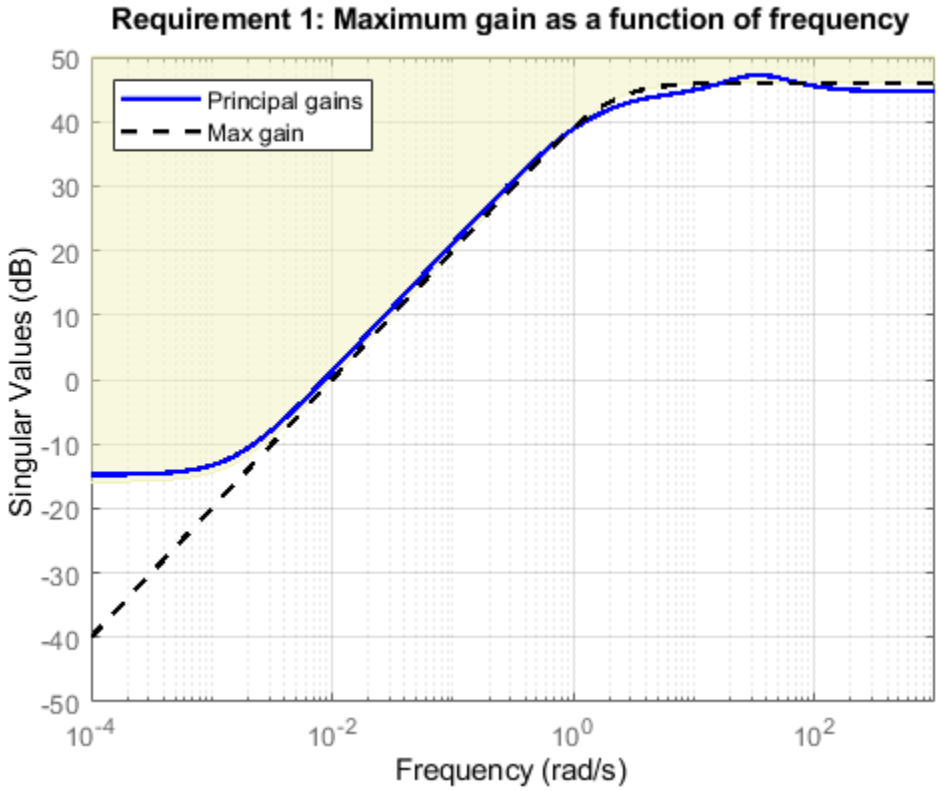
Use `viewGoal` to graphically inspect each requirement. This is useful to understand whether small violations are acceptable or what causes large violations. First verify the tracking requirement.

```
viewGoal(Req1,ST1)
```



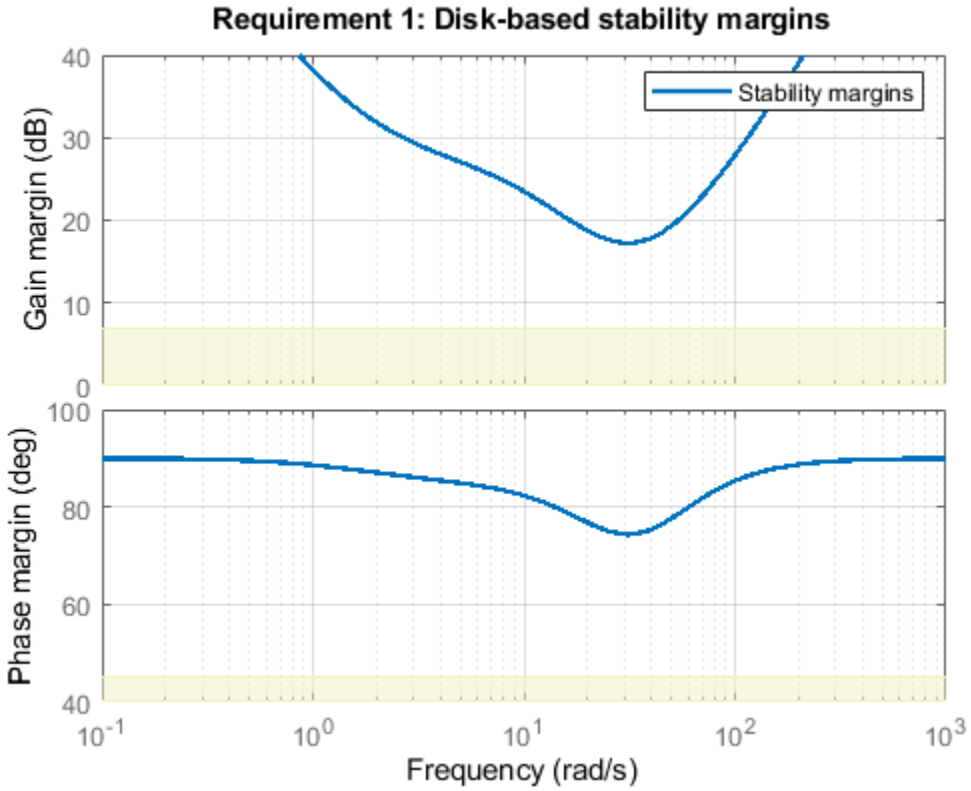
We observe a slight violation across frequency, suggesting that setpoint tracking will perform close to expectations. Similarly, verify the disturbance rejection requirement.

```
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`.

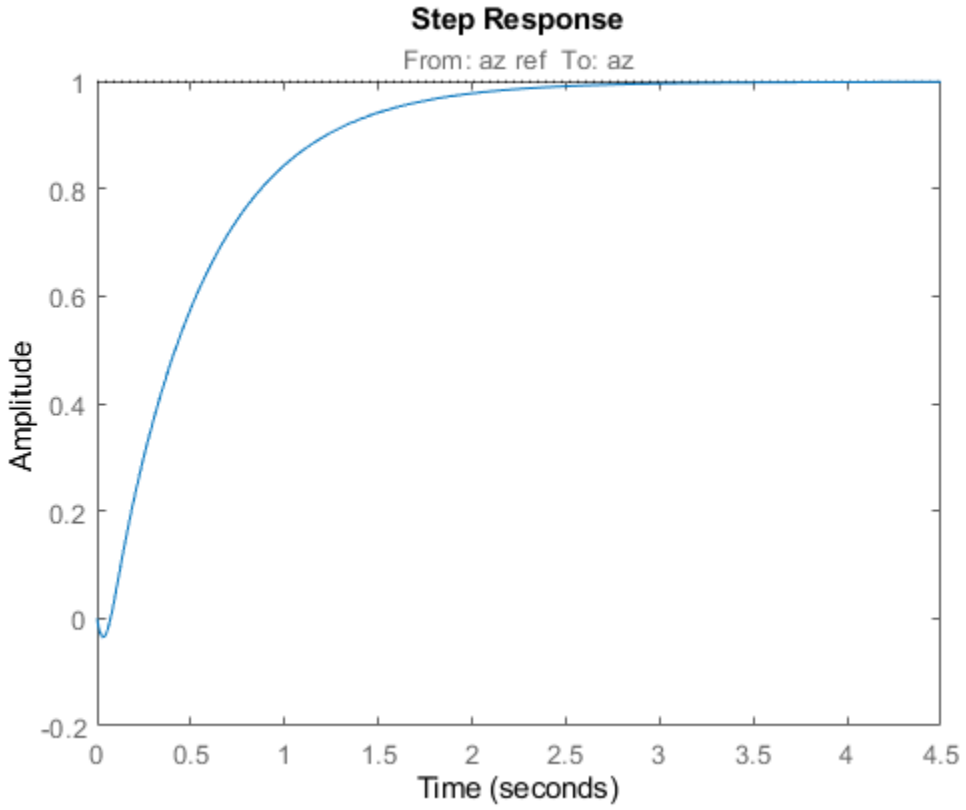
```
[f1 fSoft(1) getPeakGain(H1,1e-6)]
```

```
ans =

    1.1476    1.1476    1.1476
```

Analyzing System Responses

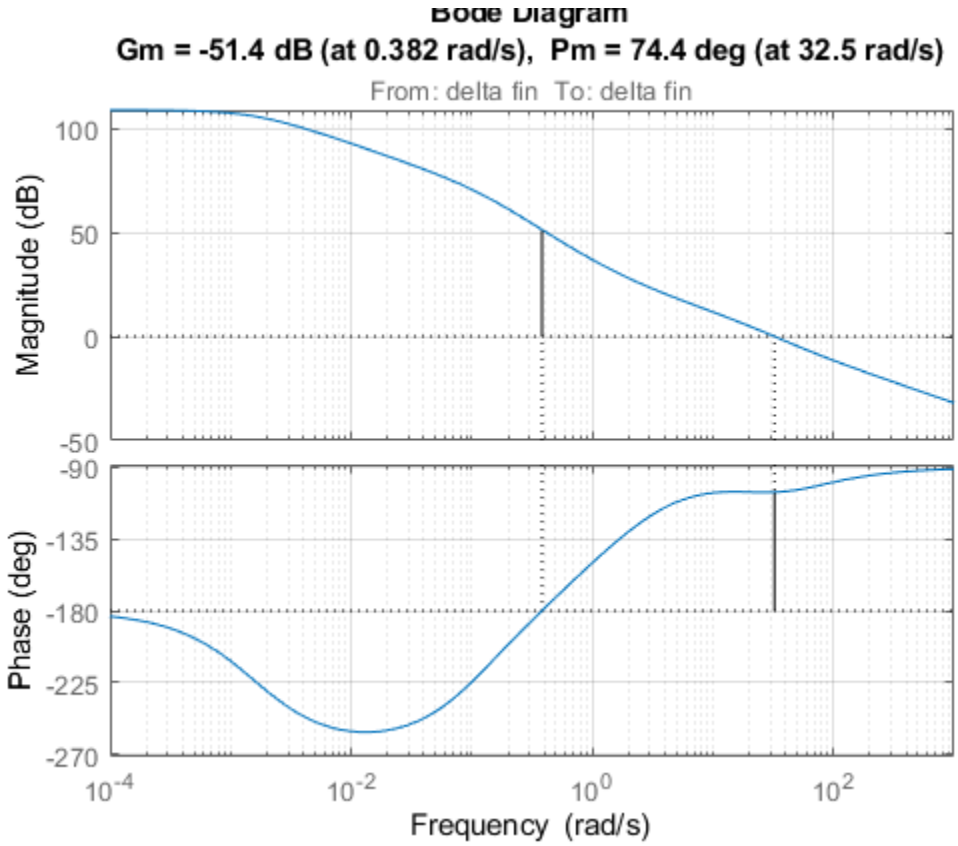
In addition to verifying requirements, you can perform basic open- and closed-loop analysis using `getIOTransfer` and `getLoopTransfer`. For example, verify tracking performance in the time domain by plotting the response `az` to a step command `azref` for the tuned system `ST1`.

```
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 = 0.99968, Iterations = 176
```

```
fSoft
```

```
fSoft =

    1.2805    1.2862
```

```
gHard
```

```
gHard =

    0.4665    0.9997
```

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.

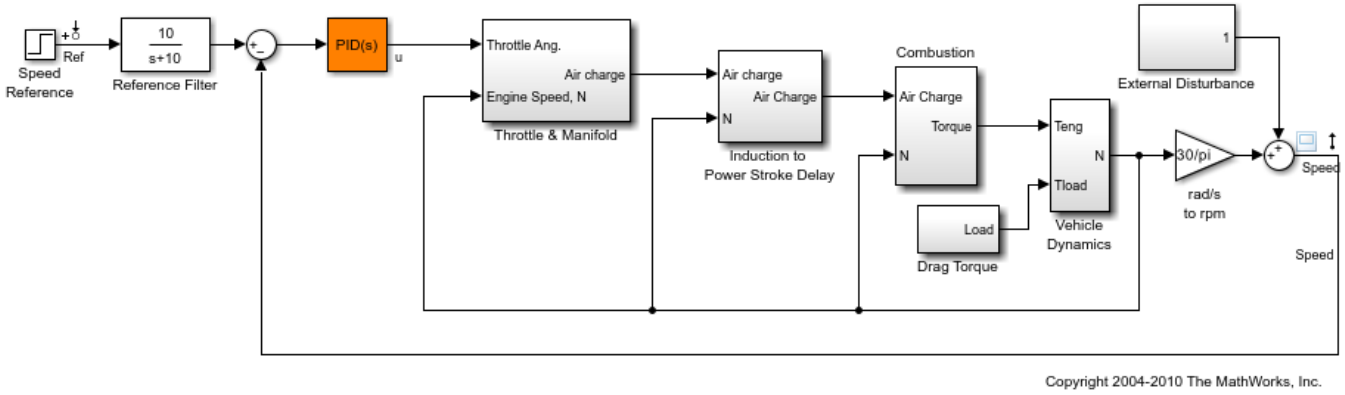
Tune Control Systems in Simulink

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`.

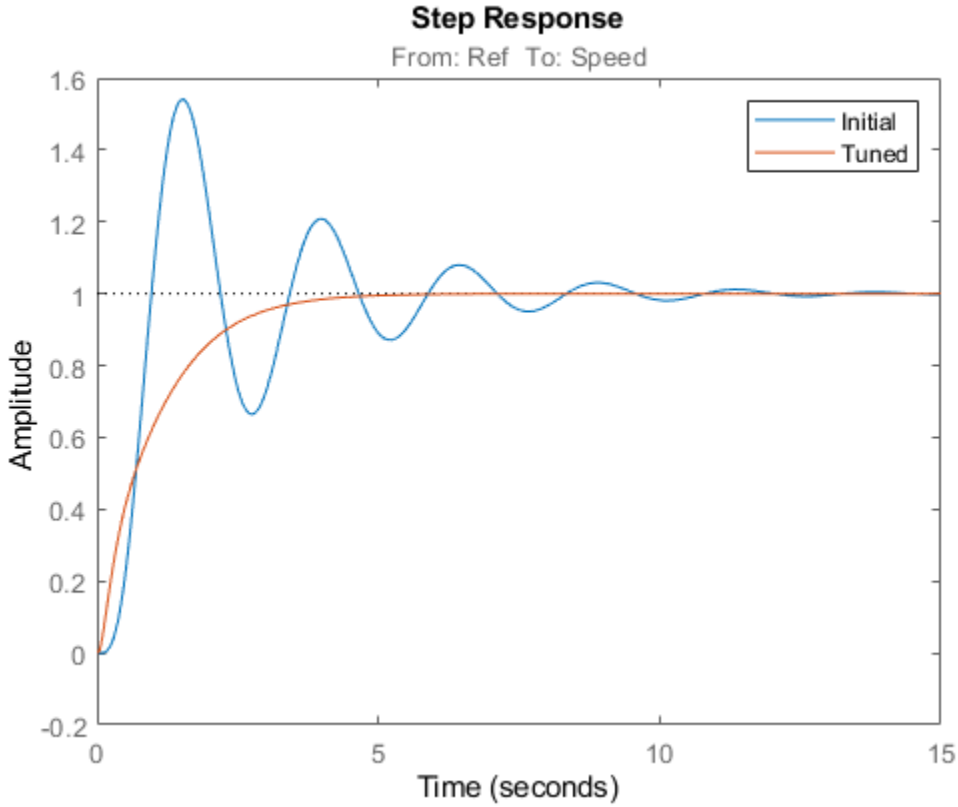
```
ST1 = systune(ST0,TrackReq);

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 = 1.24e-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

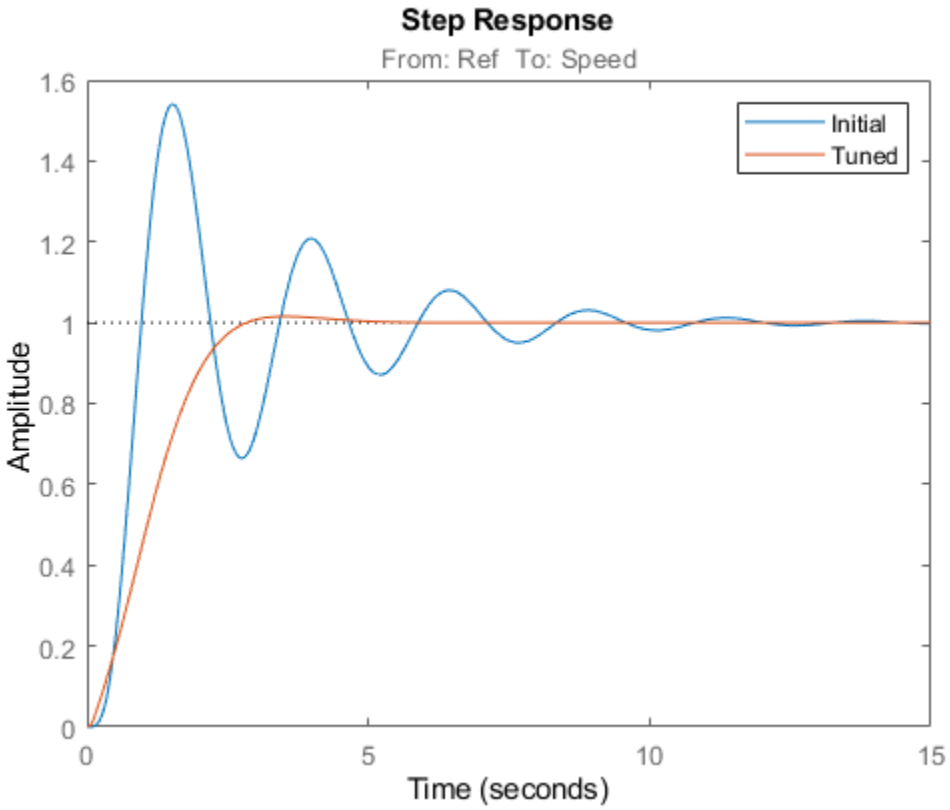
You can also use `looptune` to tune control systems modeled in Simulink. The `looptune` workflow is very similar to the `systune` workflow. One difference is that `looptune` needs to know the boundary between the plant and controller, which is specified in terms of *controls* and *measurements* signals. 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.
```

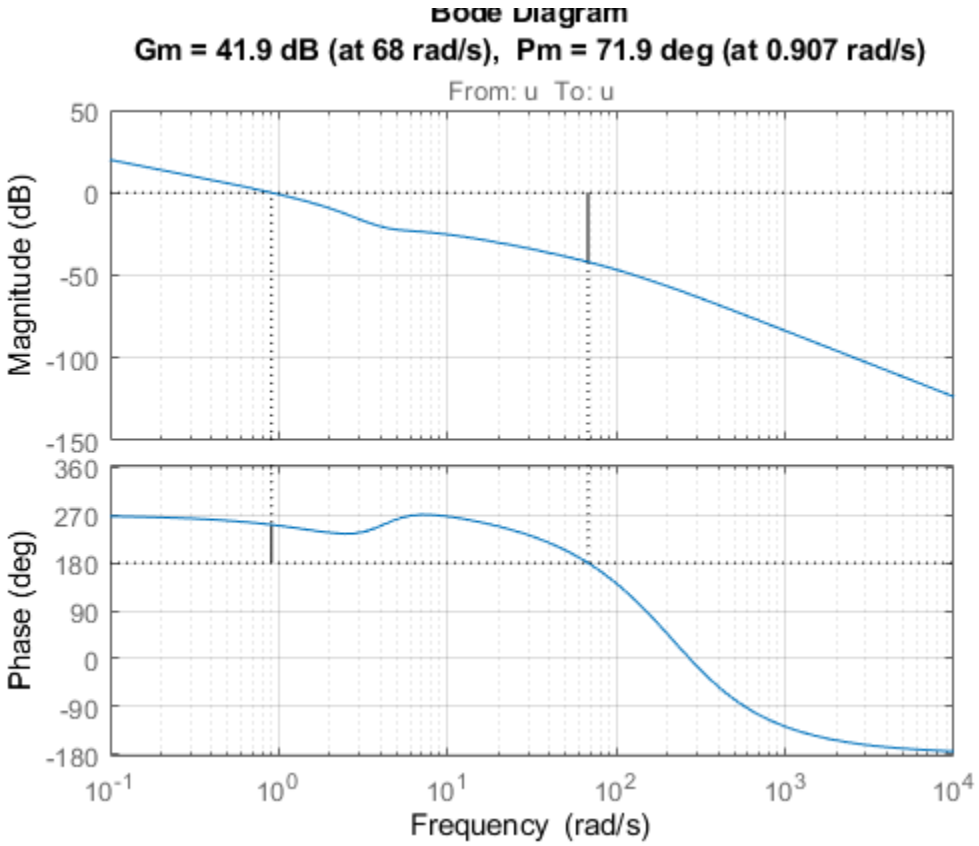
Again the final value is close to 1, indicating that the target control bandwidth was achieved. As with `systune`, use `getIOTransfer` to compute and plot the closed-loop response from speed command to actual speed. The result is very similar to that obtained with `systune`.

```
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
```

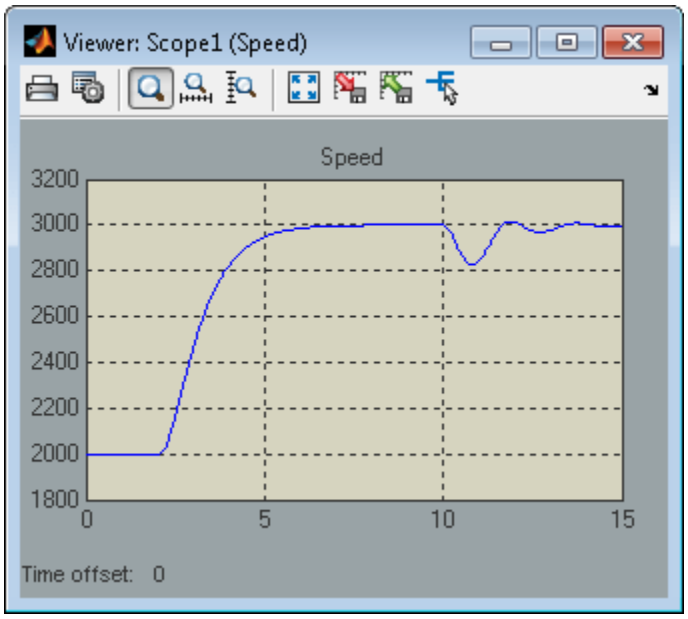


Validation in Simulink

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')
```

```
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.
```

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'))
```

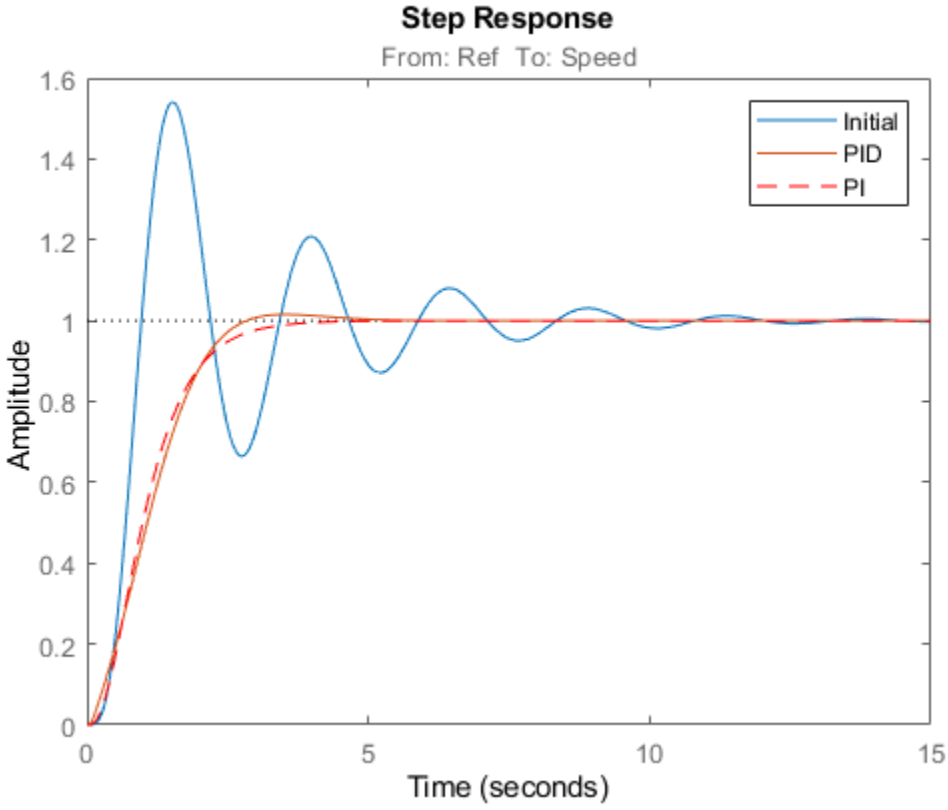
This specifies that the "PID Controller" block should now be parameterized as a mere PI controller. Next re-tune the control system for this simpler controller:

```
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” on page 14-157

Tune a Control System Using Control System Tuner

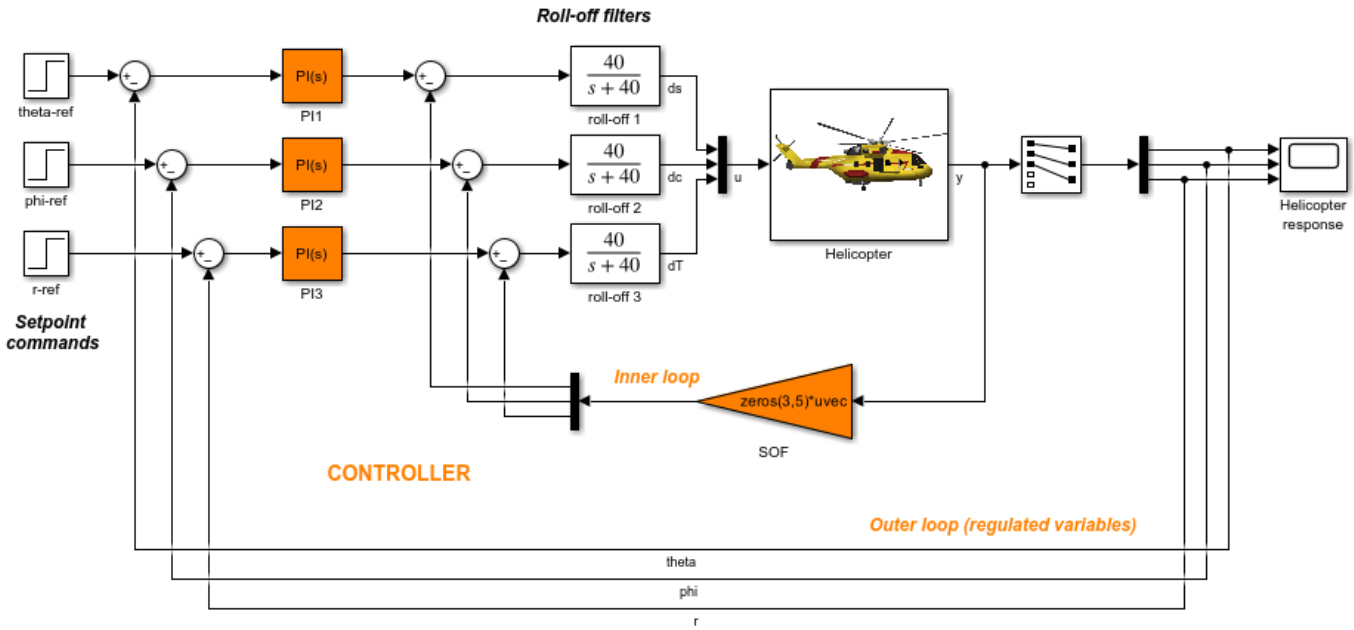
This example shows how to use the Control System Tuner app to tune a MIMO, multiloop control system modeled in Simulink®.

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.

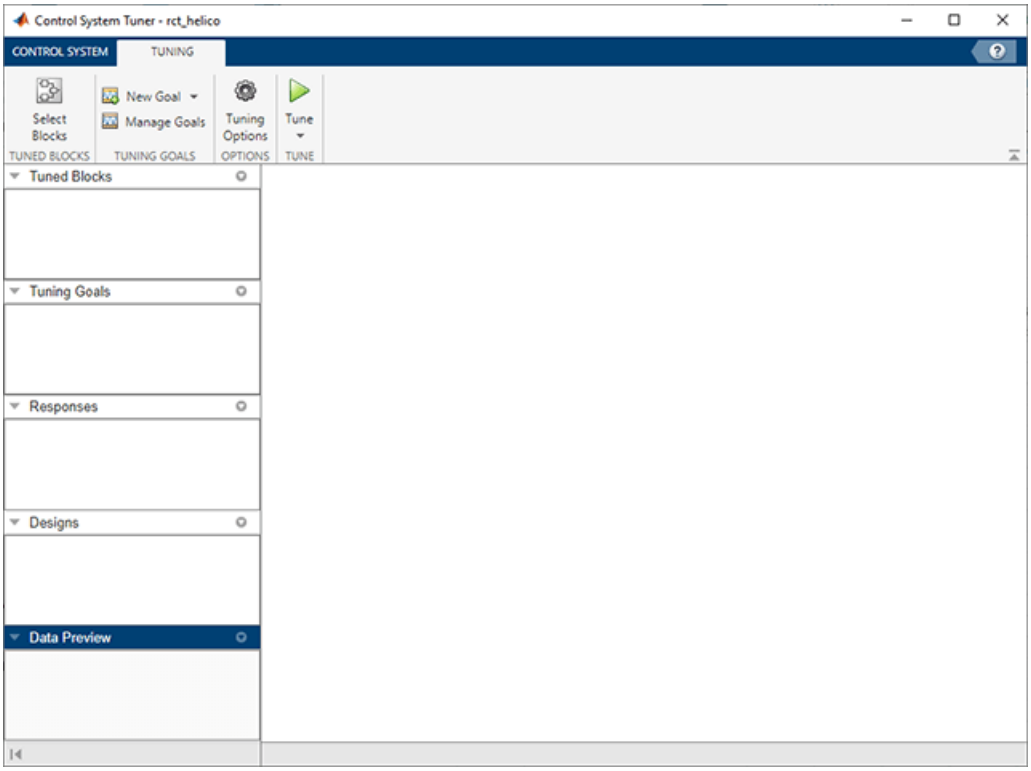
This example uses these control objectives:

- 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.)

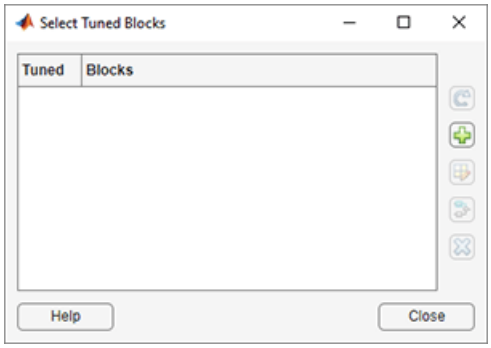
Set Up the Model for Tuning

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.

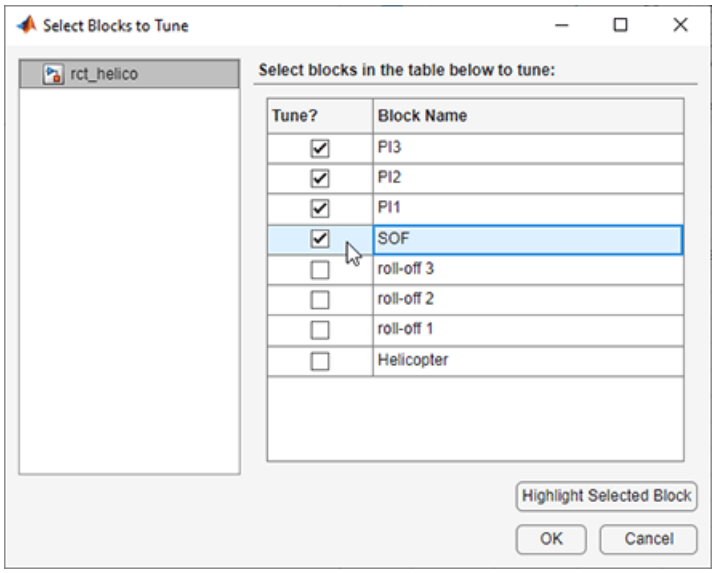
In the Simulink model window, under **Control Systems** in the **Apps** tab, select **Control System Tuner**.



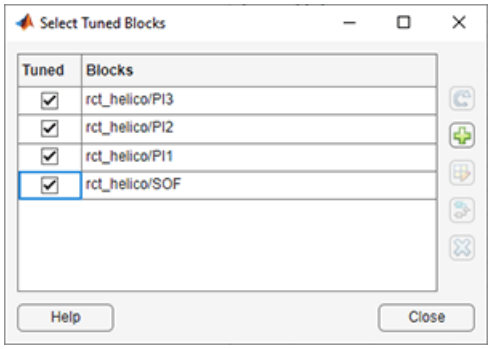
In Control System Tuner, on the **Tuning** tab, click **Select Blocks**. Use the Select tuned blocks dialog box to specify the blocks to tune.



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`.

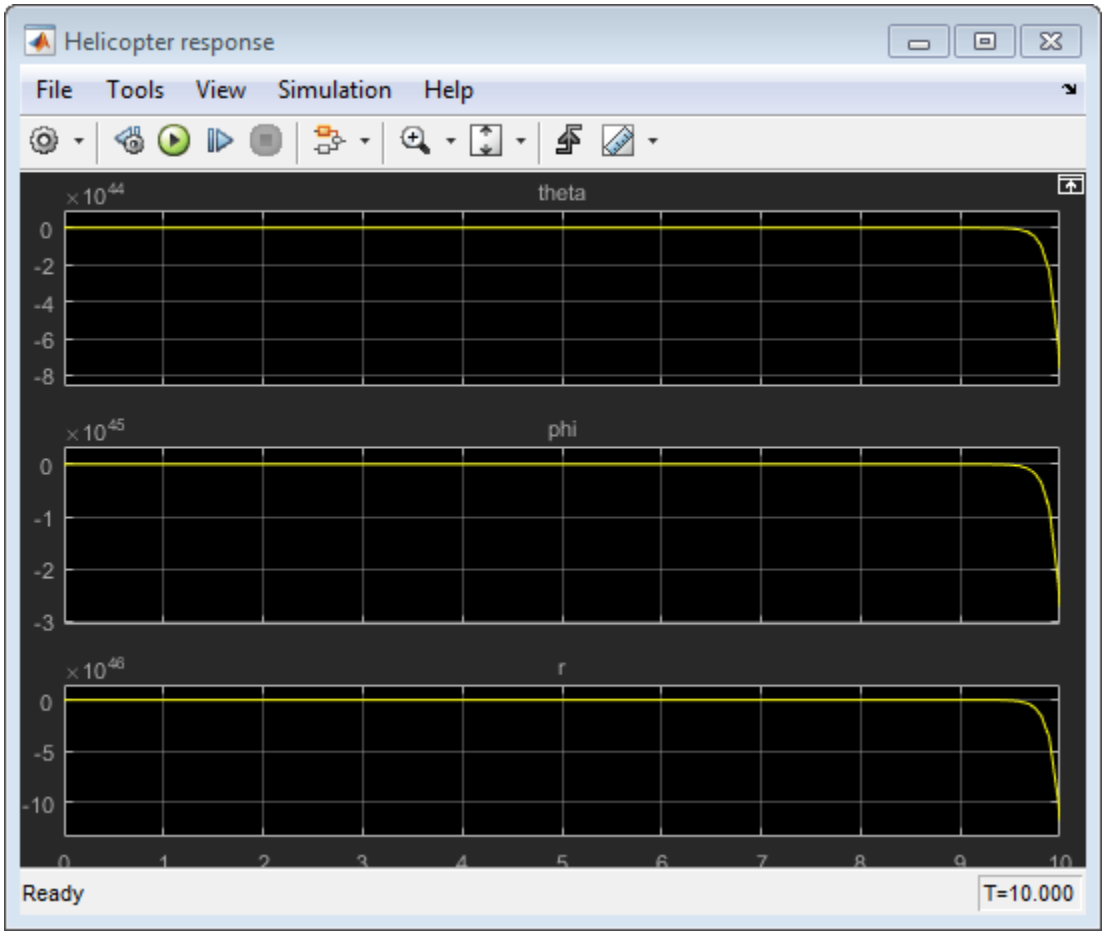


Click **OK**. The Select tuned blocks dialog box now reflects the blocks you added.



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.

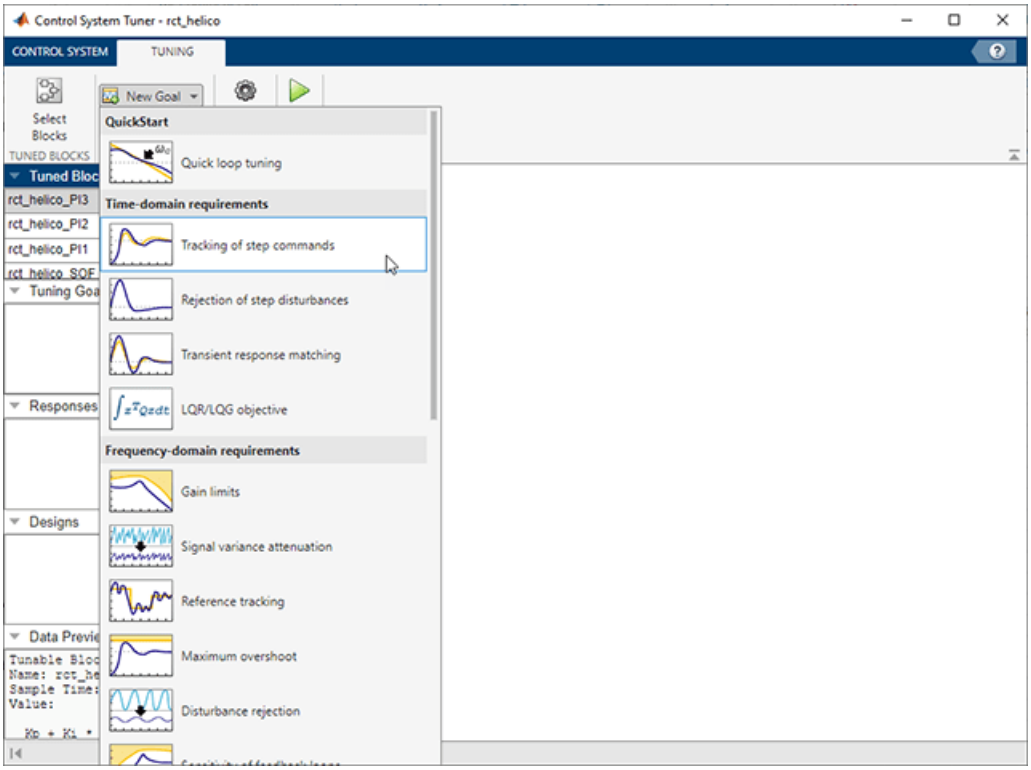
In this example, the PI controllers are initialized to $1 + 1/s$ and the static output-feedback gain is initialized to zero on all channels. Simulating the model shows that the control system is unstable for these initial values.



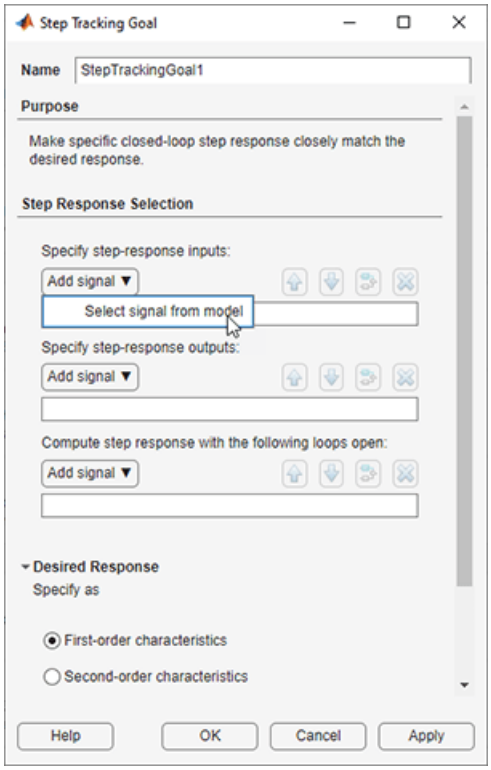
Specify Tuning Goals

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**.

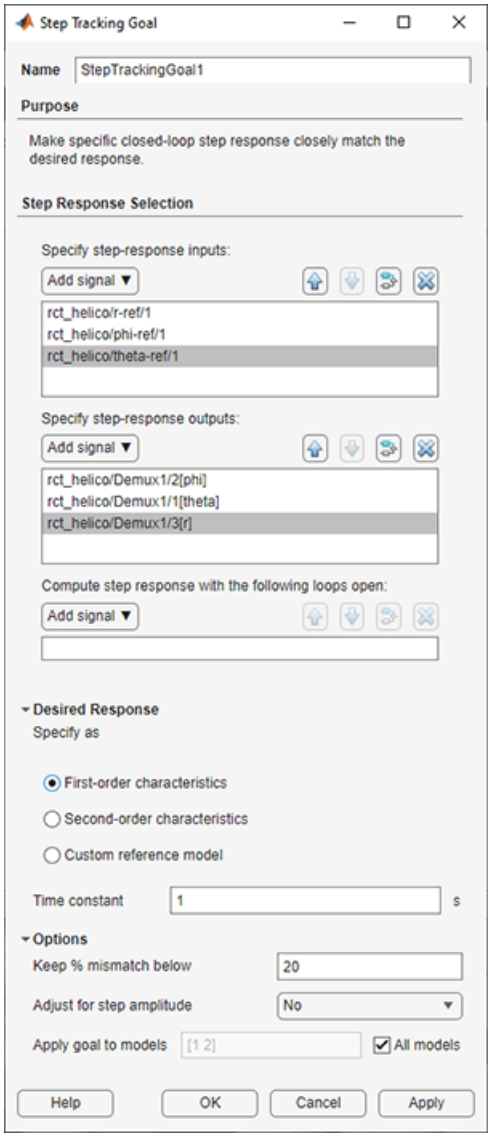


In the Simulink model editor, select the reference signals `theta_ref`, `phi_ref`, and `r_ref`. These signals appear in the Select signals dialog box. Click **Add Signal(s)** to add them to the step tracking goal.

Next, specify the outputs that you want to track those references. Under **Specify step-response outputs**, add the outputs `theta`, `phi`, and `r`.

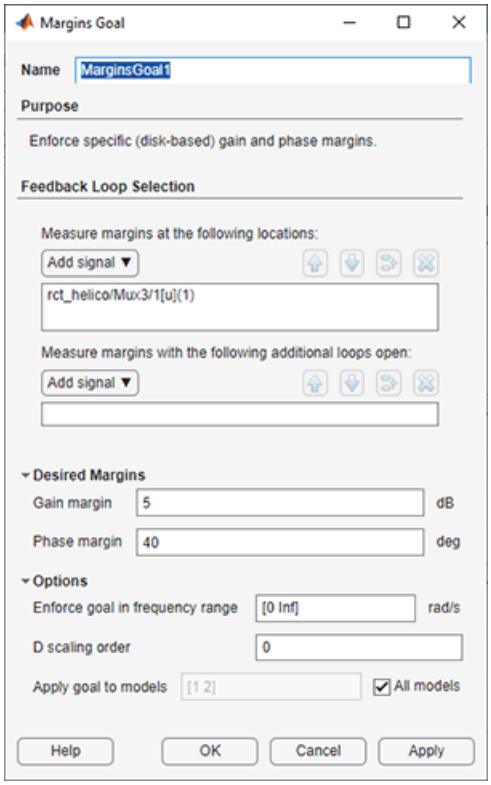
The requirement is that the responses at the outputs track the reference commands with a first-order response that has a one-second time constant. Enter these values in the **Desired Response** section of the dialog box. Also, for this example set **Keep mismatch below** to 20. This value sets a 20% 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.

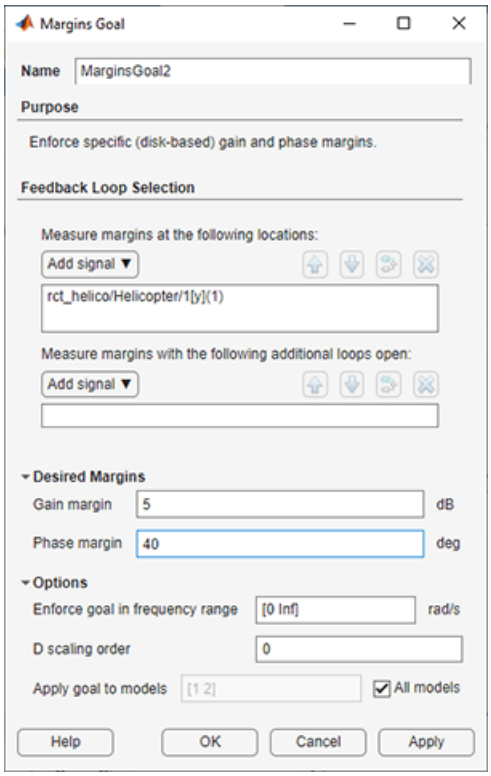


Next, create tuning goals for the desired stability margin requirements. For this example, the multivariable gain and phase margins at the plant inputs `u` and plant outputs `y` must be at least 5 dB

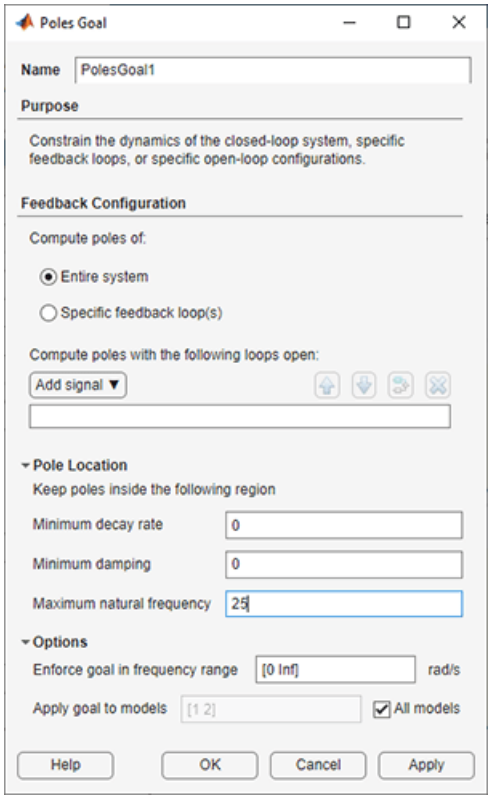
and 40 degrees. Create separate tuning goals for the input and output margin constraints. In the **New Goal** drop-down list, select **Minimum stability margins**. In the Margins Goal dialog box, add the input signal u under **Measure stability margins at the following locations**. Also, enter the gain and phase values 5 and 40 in the **Desired Margins** section of the dialog box. Click **OK** to save the input stability margin goal.



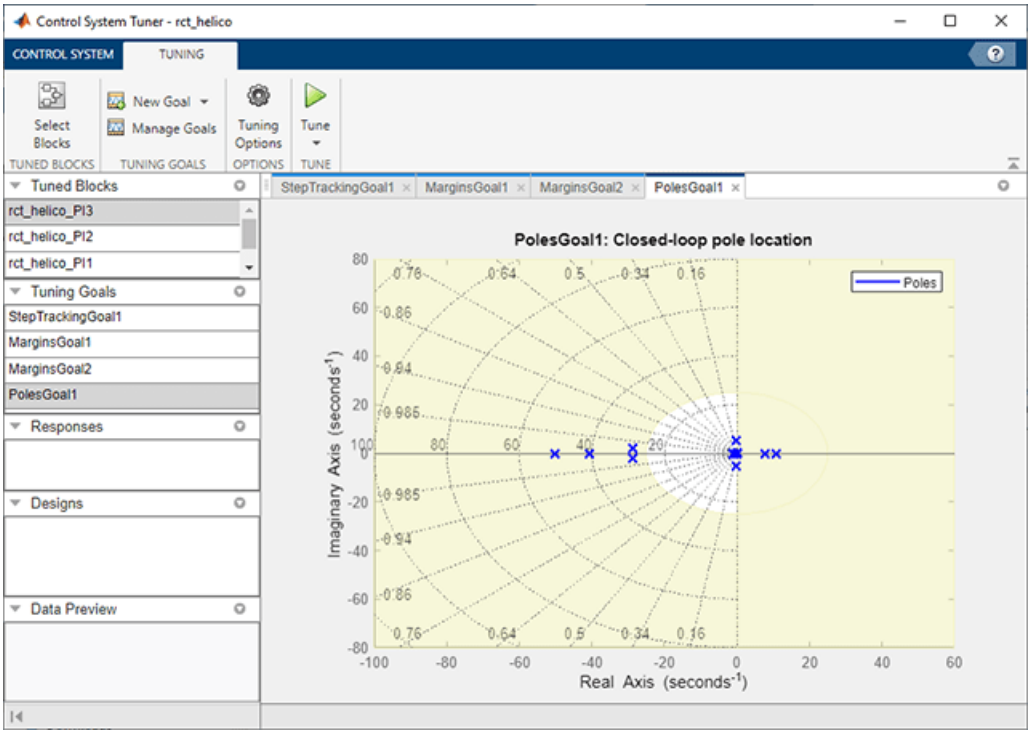
Create another Margins Goal for the output stability margin. Specify the output signal y and the target margins, as shown, and save the output stability margin goal.



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.

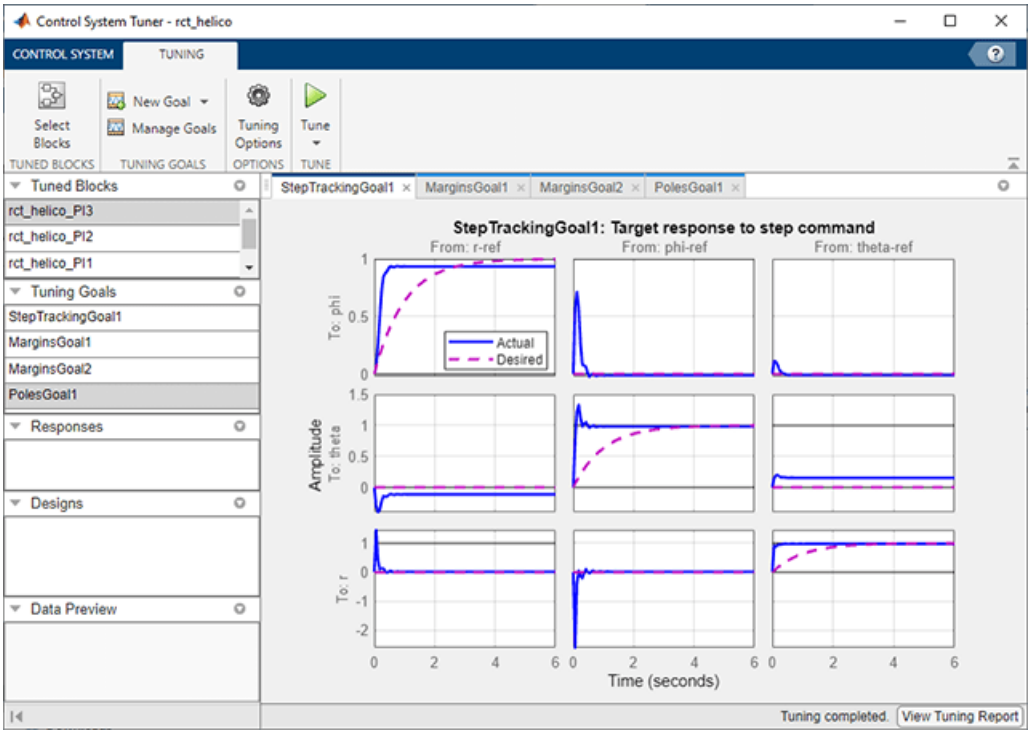


Tune the Control System

Tune the control system to meet the design requirements you have specified.

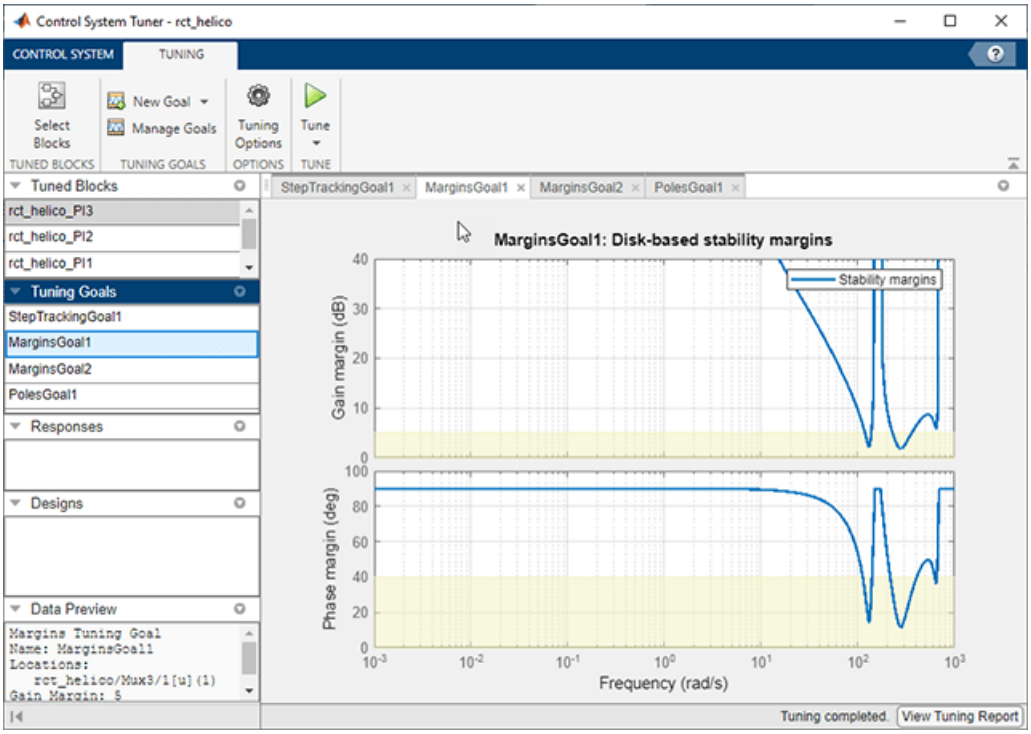
On the **Tuning** tab, click **Tune**. Control System Tuner adjusts the tunable parameters to values that best meet those requirements.

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.

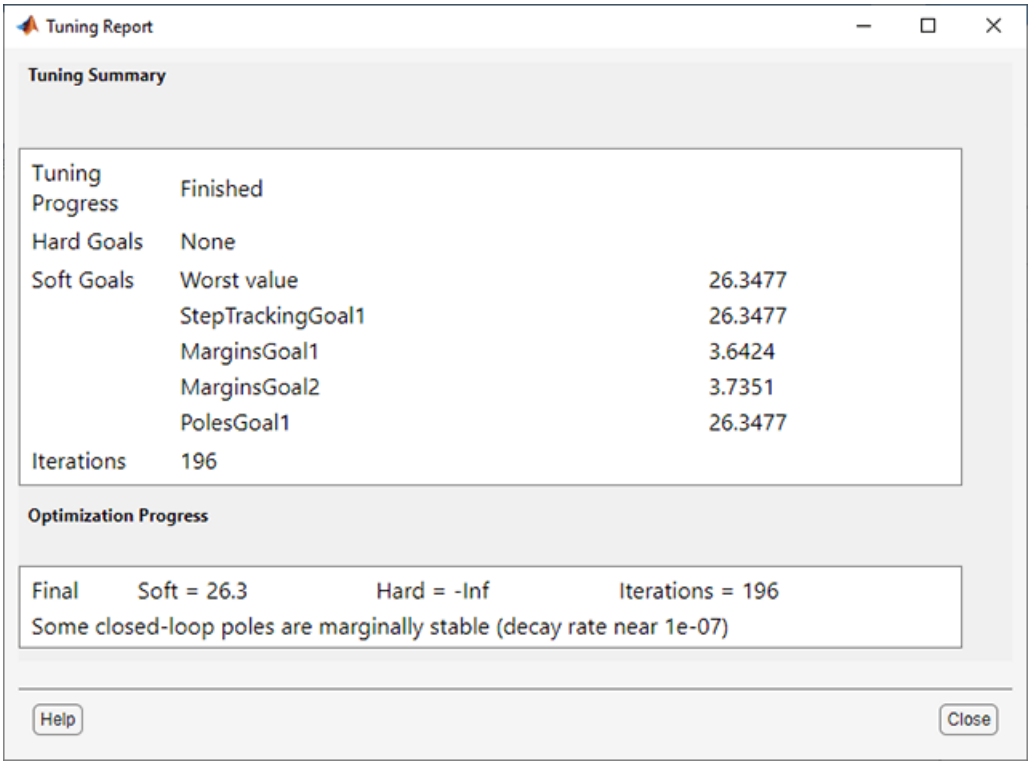


The blue line shows that the tuned response is very close to the target response, in pink. The rise time is about two seconds, and there is no overshoot and little cross-coupling.

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.

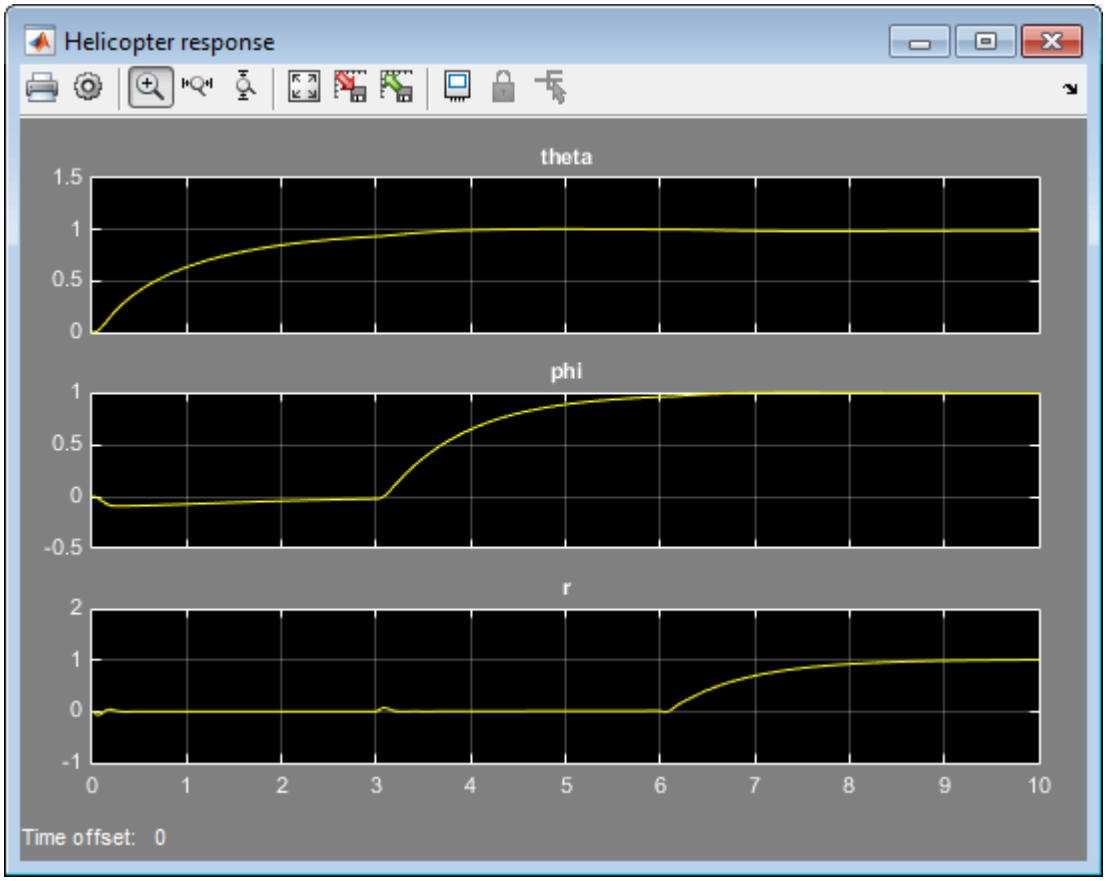


When you tune the model, Control System Tuner converts each tuning goal to a function of the tunable parameters of the system and adjusts the parameters to minimize the value of those functions. For this example, the tuning report shows that the final values for all tuning goals are close to 1, which 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” on page 14-11
- “Tuning for Multiple Values of Plant Parameters” on page 18-206

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

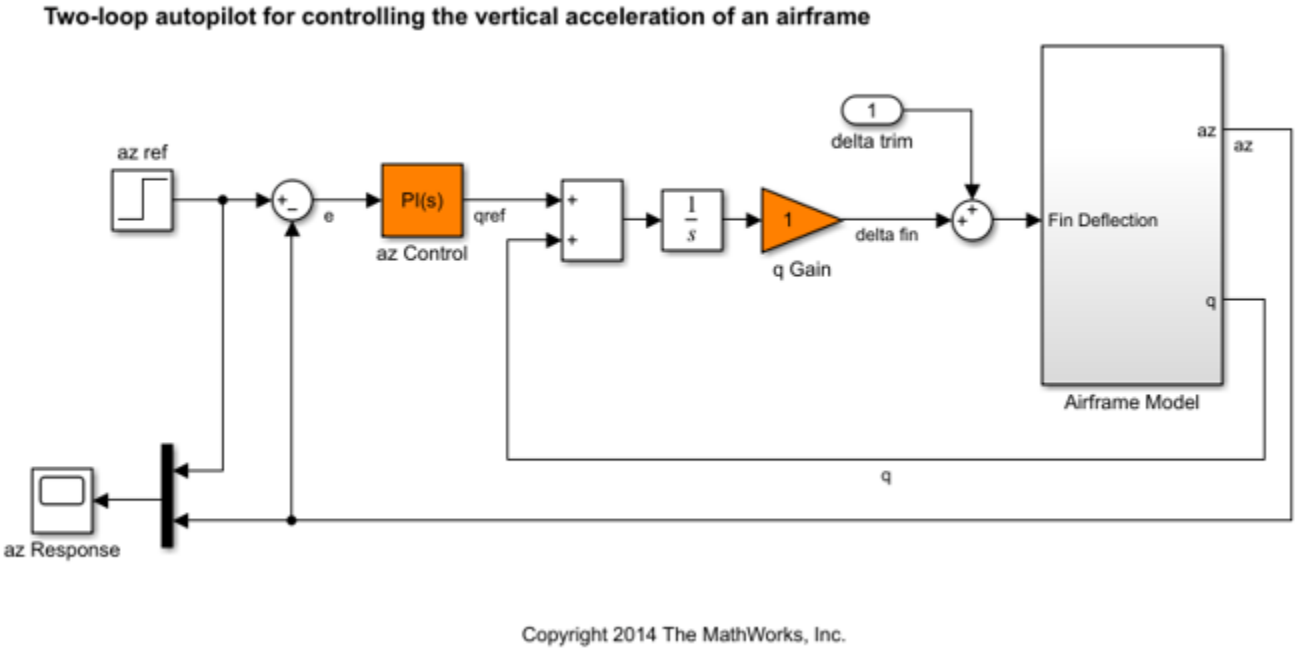
Both `systune` and `looptune` use local optimization methods for tuning the control architecture at hand. To mitigate the risk of ending up with a locally optimal but globally poor design, it is recommended to run several optimizations starting from different randomly generated initial points. If you 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 the parallel pool (number of workers: 6).
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) = 0.041)
Final: Failed to enforce closed-loop stability (max Re(s) = 0.041)
Final: Failed to enforce closed-loop stability (max Re(s) = 0.075)
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) = 0.041)
Final: Failed to enforce closed-loop stability (max Re(s) = 0.06)
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) = 0.041)
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) = 0.041)
Final: Peak gain = 1e+03, Iterations = 59
Final: Peak gain = 1.23, Iterations = 49
Final: Failed to enforce closed-loop stability (max Re(s) = 0.062)
Final: Failed to enforce closed-loop stability (max Re(s) = 0.041)
Final: Failed to enforce closed-loop stability (max Re(s) = 0.078)
Final: Peak gain = 1.23, Iterations = 133
Final: Peak gain = 1.23, Iterations = 59
Final: Peak gain = 1.23, Iterations = 61
Final: Failed to enforce closed-loop stability (max Re(s) = 0.083)
Final: Peak gain = 1e+03, Iterations = 61
Warning: Tuning goal "Open loop CG": Feedback configuration has fixed
integrators that cannot be stabilized with available tuning parameters. Make
sure these are modeling artifacts rather than physical instabilities.
```

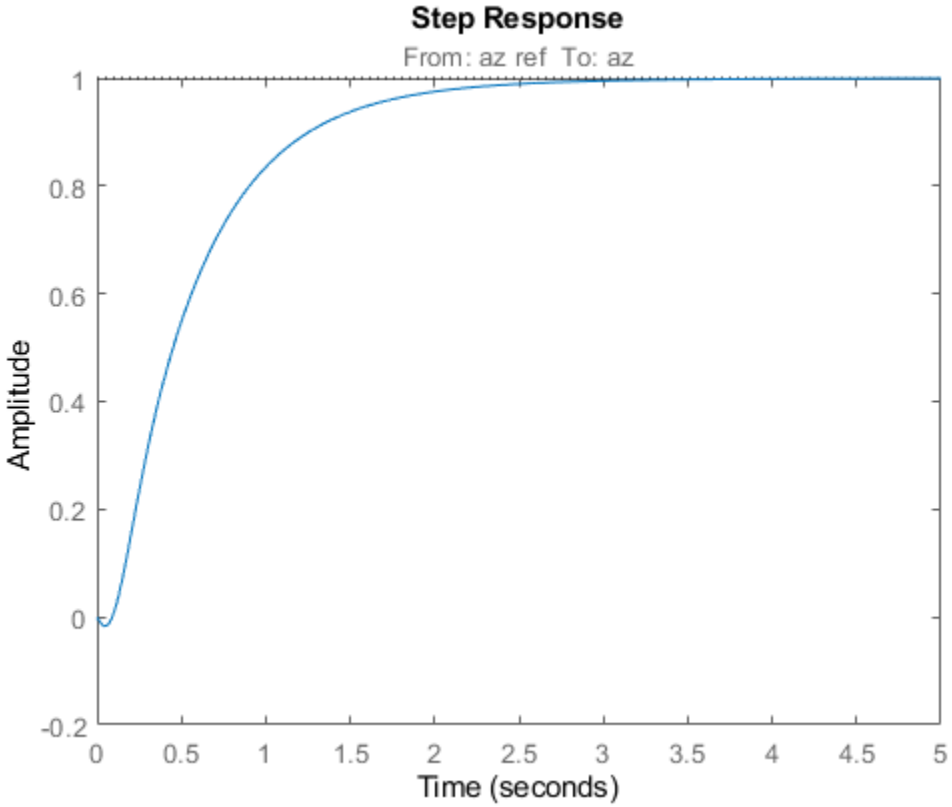
Most runs return 1.23 as optimal gain value, suggesting that this local minimum has a wide region of attraction and is likely to be the global optimum. Use `showBlockValue` to see the corresponding gain values:

```
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  1.983

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`

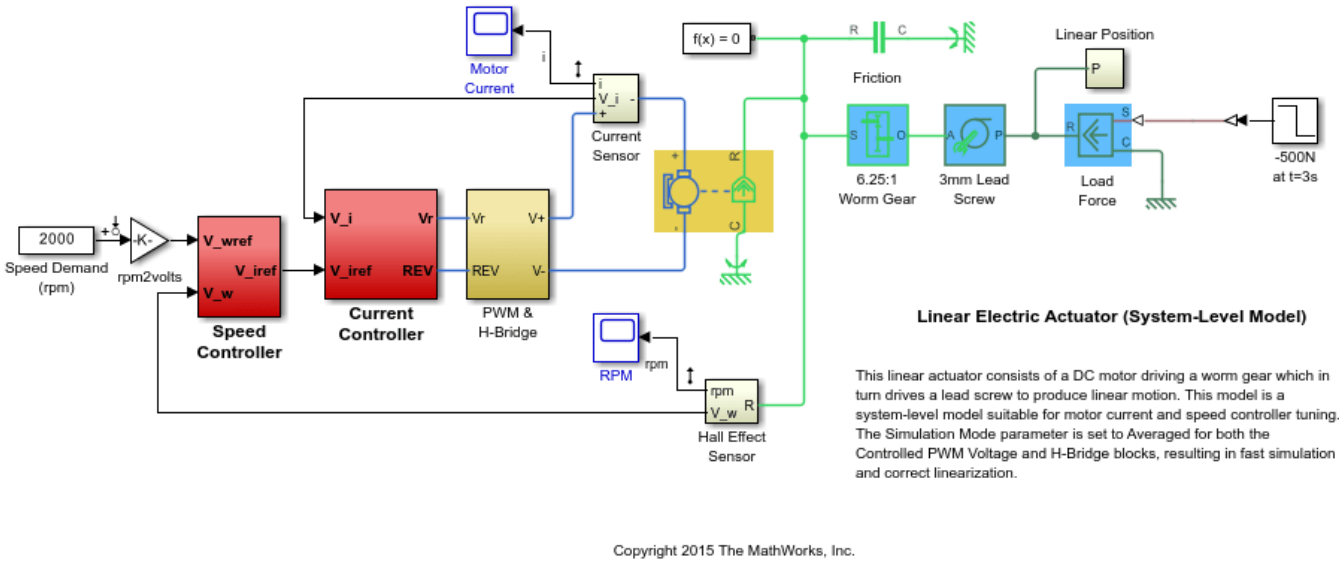
Control of a Linear Electric Actuator

This example shows how to use `slTuner` and `systune` to tune the current and velocity loops in a linear electric actuator with saturation limits.

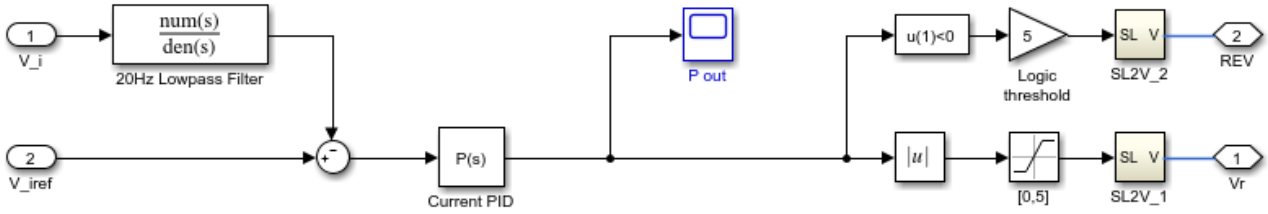
Linear Electric Actuator Model

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.



Current Controller

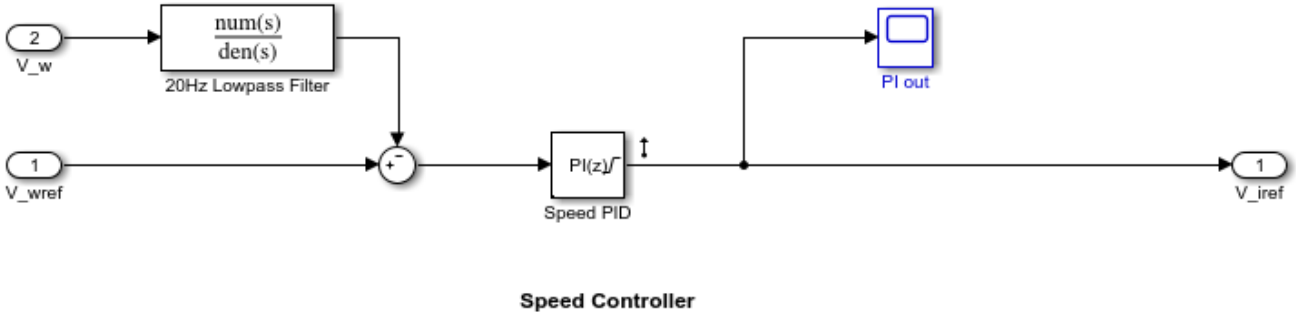


Figure 1: Current and Speed Controllers.

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.

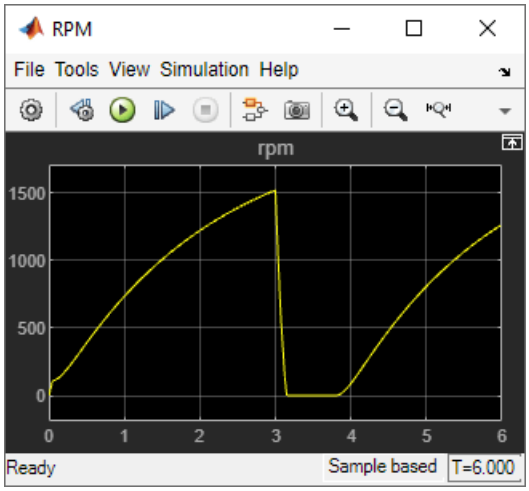


Figure 2: Untuned Response.

Control System Tuning

You can use `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);
```

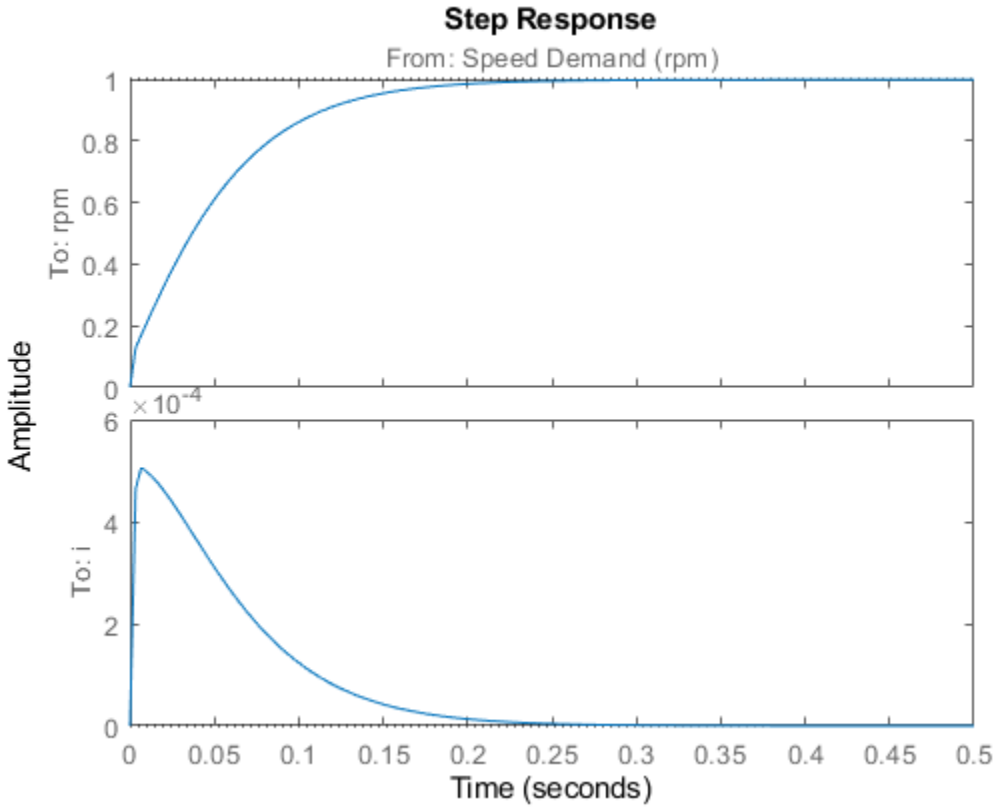
You can now tune the proportional and integral gains with `looptune`:

```
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.

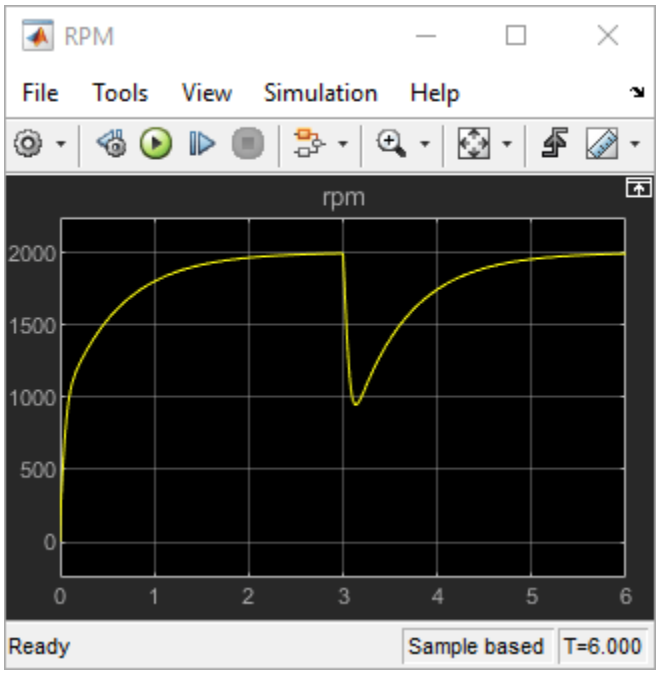


Figure 3: Nonlinear Simulation of Tuned Controller.

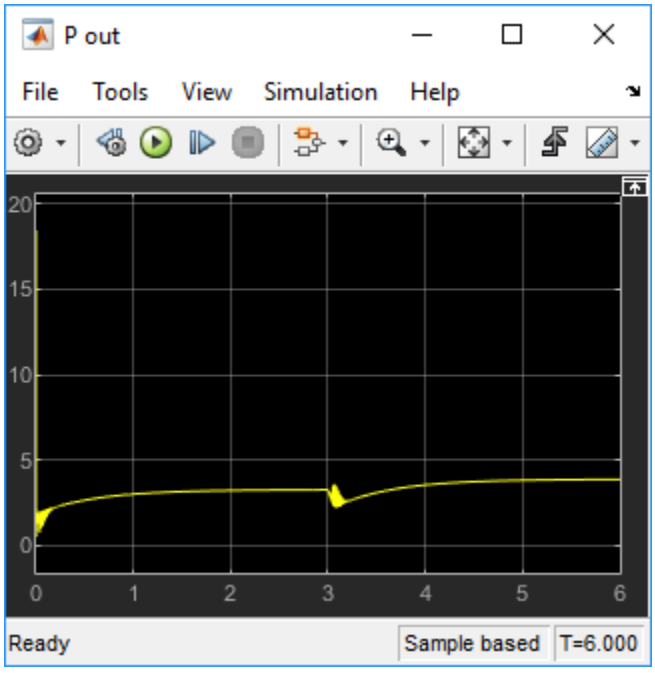


Figure 4: Current Controller Output.

Preventing Saturations

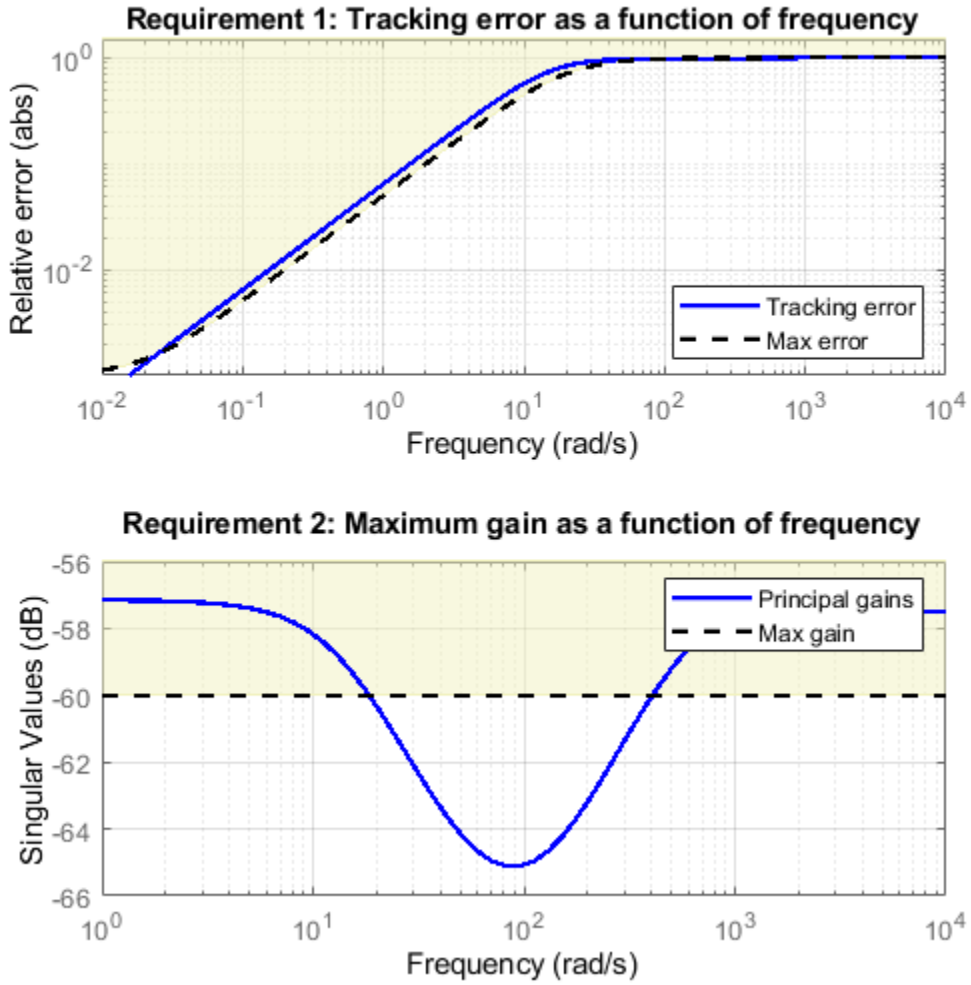
So far we have only specified a desired response time for the outer (speed) loop. This leaves `systune` free to allocate the control effort between the inner and outer loops. Saturation in the inner loop suggests that the proportional gain is too high and that some rebalancing is needed. One possible 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
```

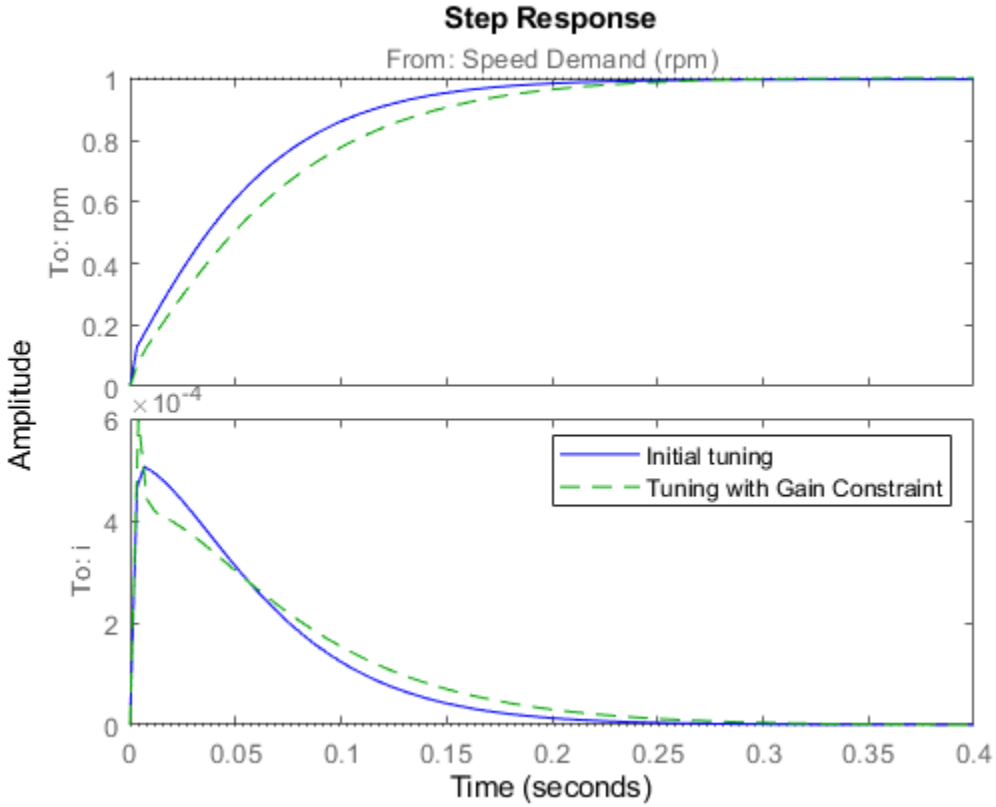
The final gain 1.39 indicates that the requirements are nearly but not exactly met (all requirements are met when the final gain is less than 1). Use `viewGoal` to inspect how the tuned controllers fare against each goal.

```
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 = 2.2

Name: Current_PID
P-only controller.

-----------------------------------

Block 2: rct_linact/Speed Controller/Speed PID =

             1
  Kp + Ki * ---
             s

  with Kp = 0.481, Ki = 4.94

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)
```

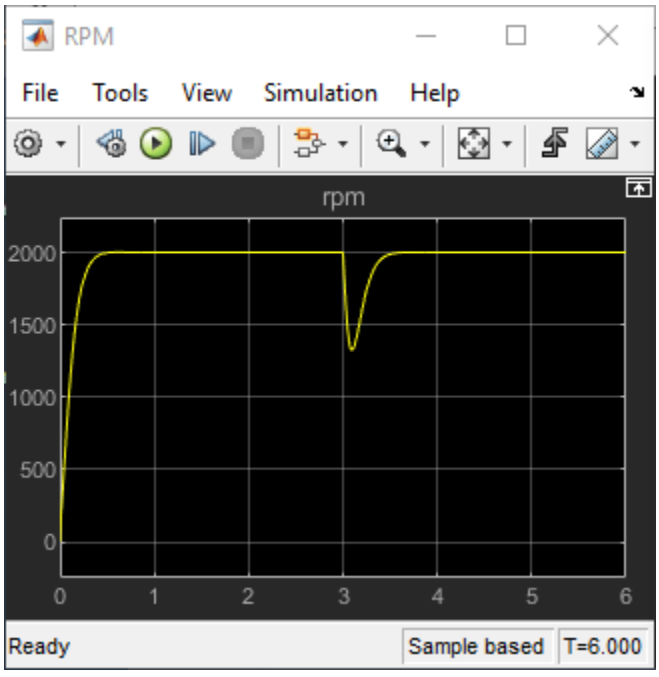


Figure 5: Nonlinear Response of Tuning with Gain Constraint.

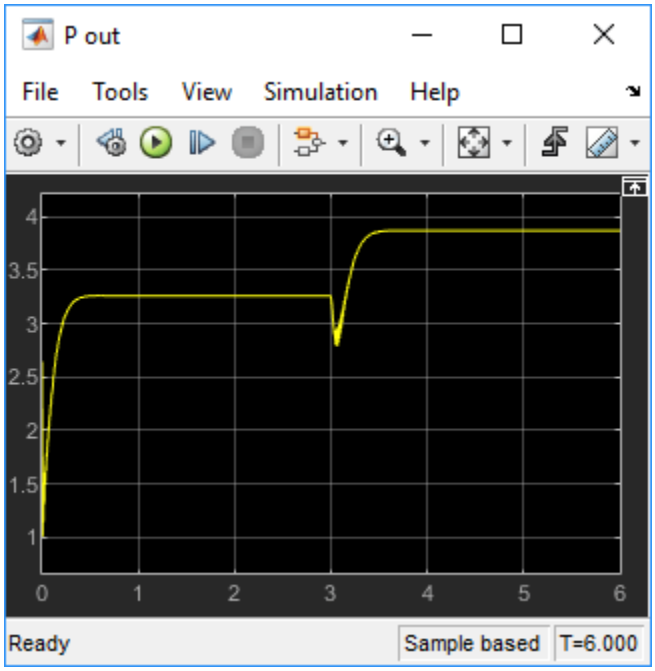


Figure 6: Current Controller Output.

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 18-85
- “Tune Control Systems in Simulink” on page 18-50
- “Tune Control Systems Using systune” on page 17-2

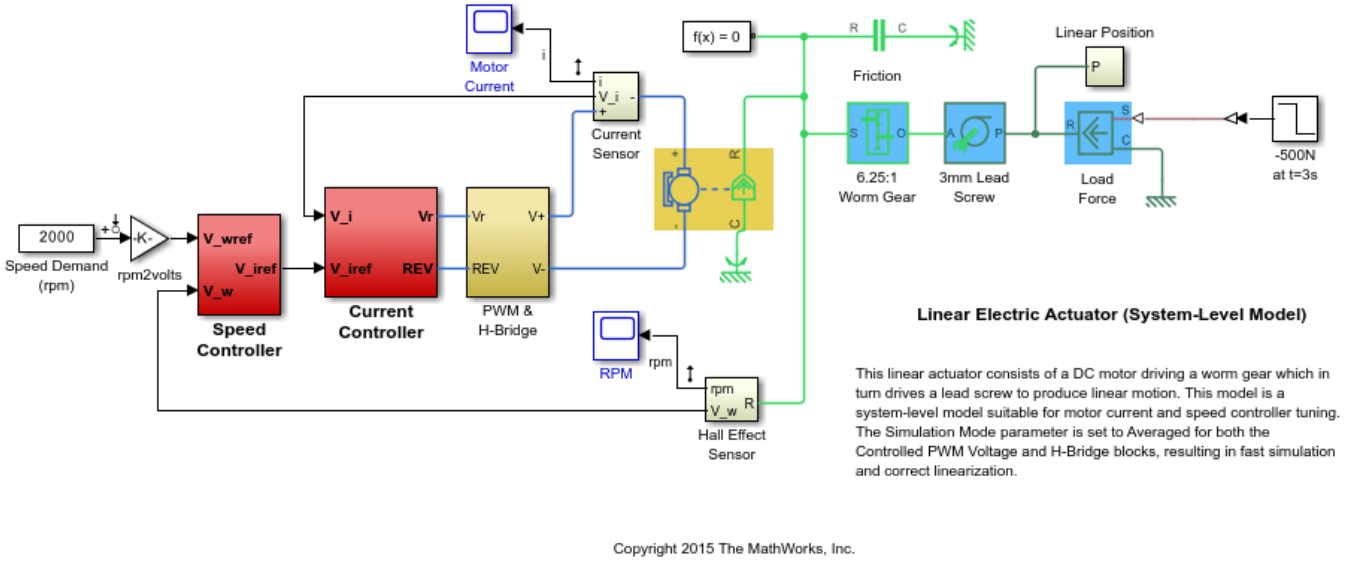
Control of a Linear Electric Actuator Using Control System Tuner

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.

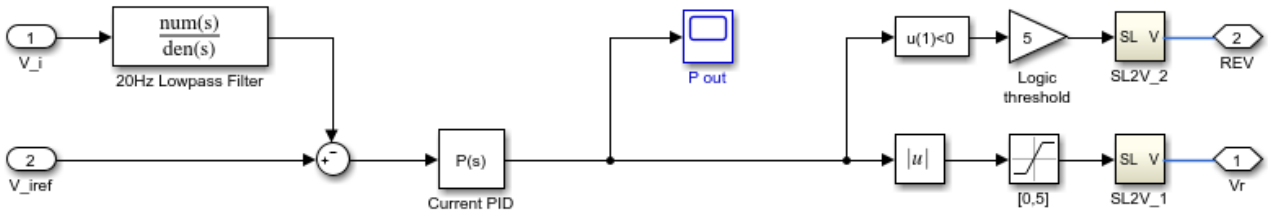
Linear Electric Actuator Model

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.



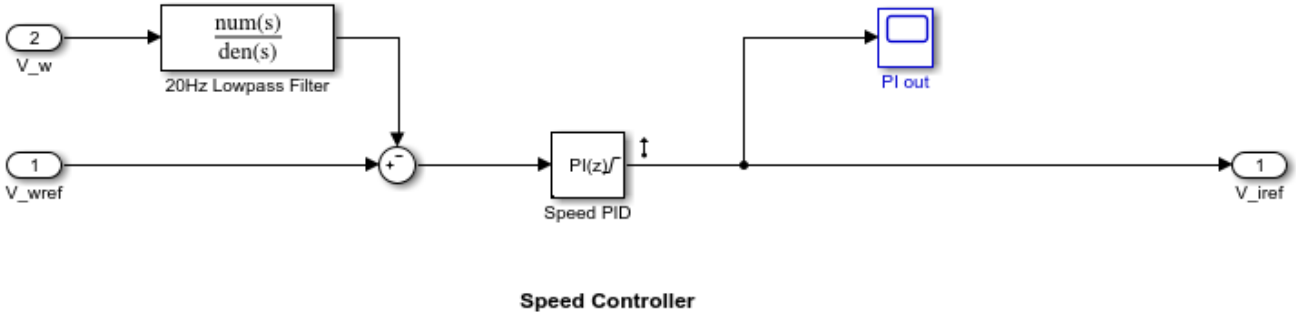


Figure 1: Current and Speed Controllers.

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.

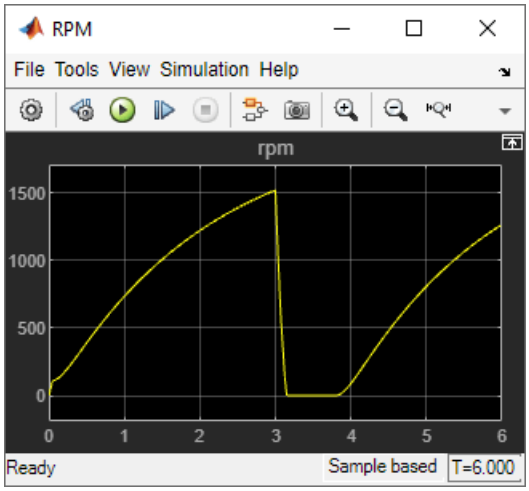


Figure 2: Untuned Response.

Control System Tuning

You can use Control System Tuner to jointly tune both feedback loops. First, open Control System Tuner from the Apps tab.

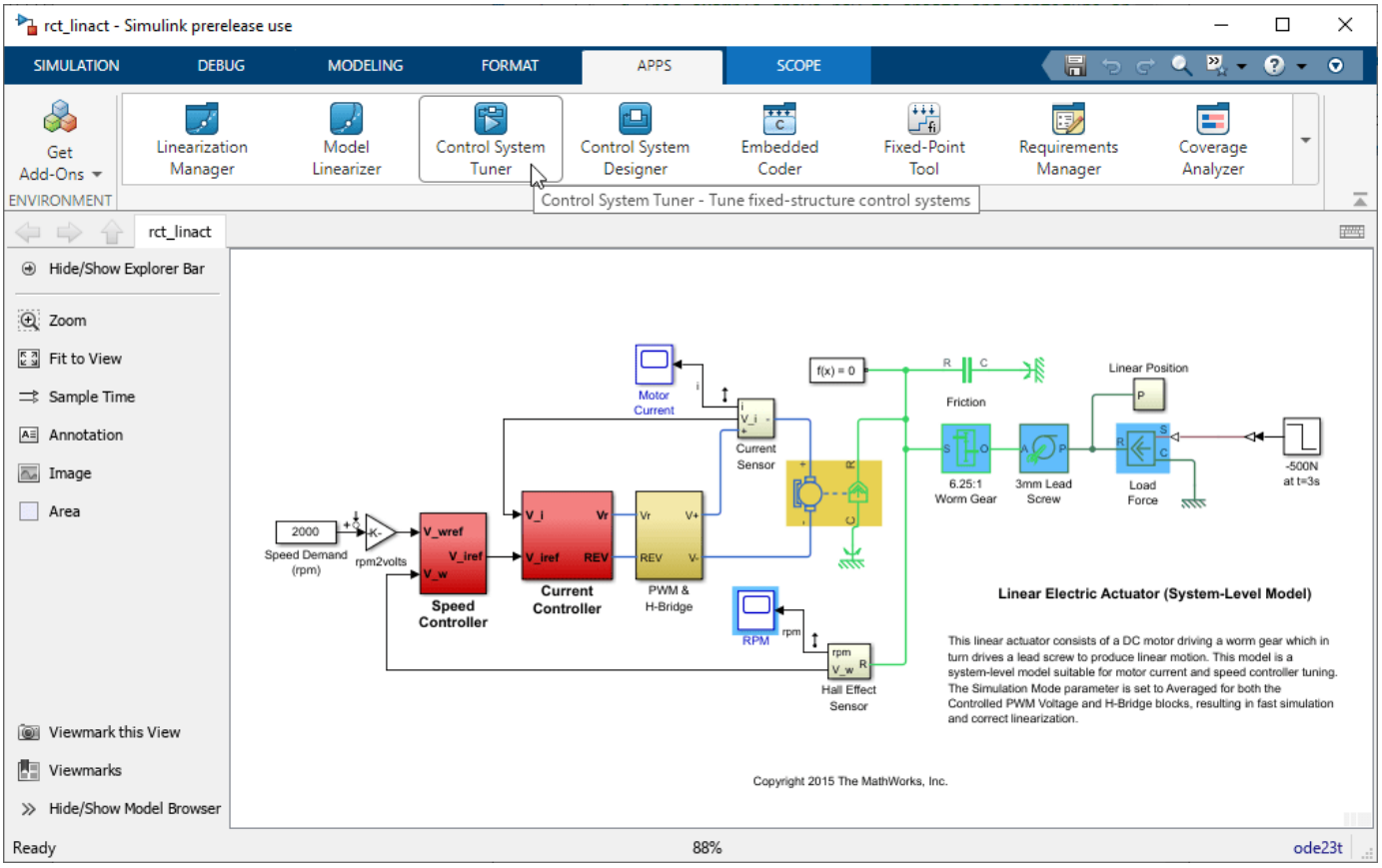


Figure 3: Opening Control System Tuner.

This opens Control System Tuner.

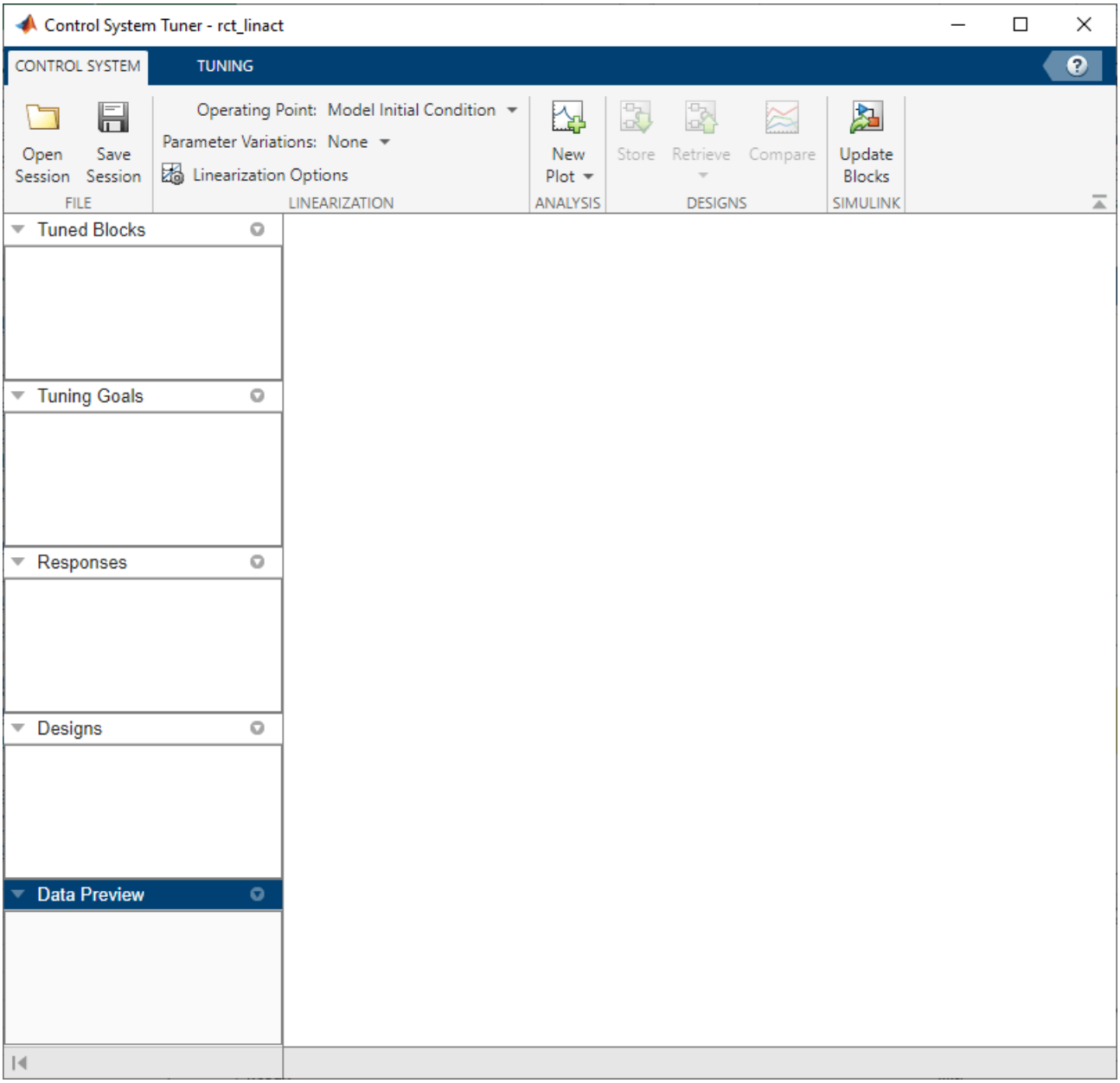


Figure 4: Control System Tuner.

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...`.

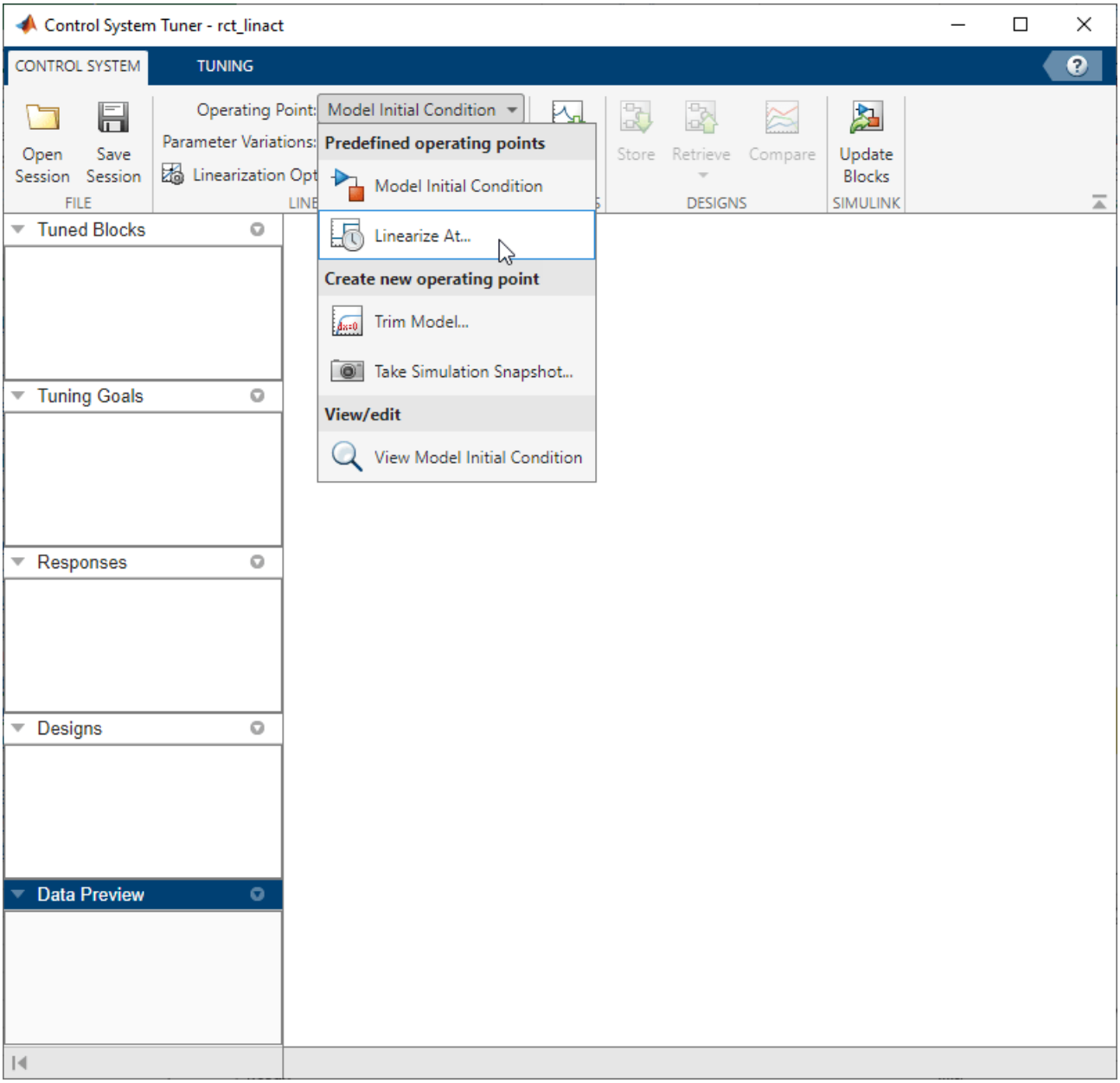


Figure 5: Setting Operating Point for Linearization.

Set the linearization snapshot time at t=0.5.

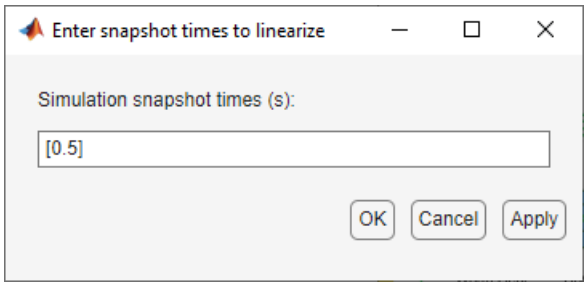


Figure 6: Setting the Linearization Snapshot Time.

In order to set the tuned blocks of the control system, open `Select Blocks` from `Tuning` tab.

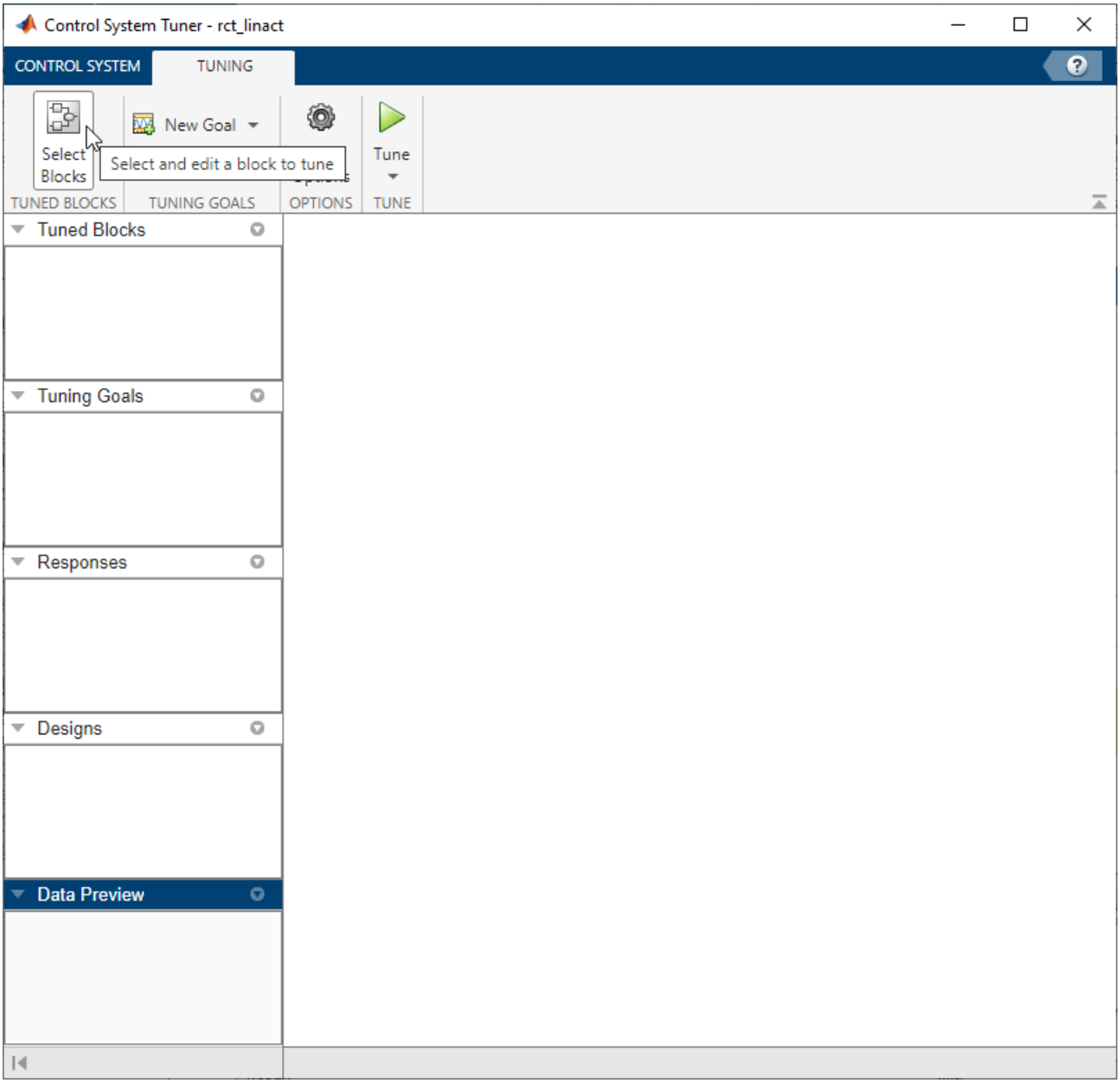


Figure 7: Tuning Tab of Control System Tuner.

This shows the editor for tuned blocks where you can `Add Blocks`.

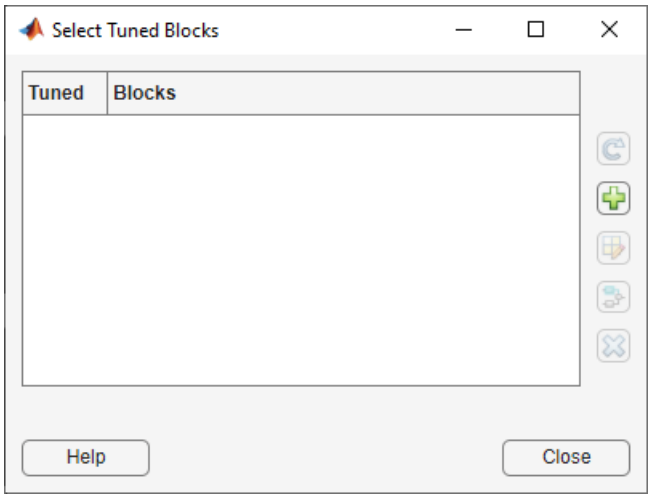


Figure 8: Editor for Tuned Blocks.

Set the tuned blocks `Current PID` and `Speed PID` by navigating through the tree on the left.

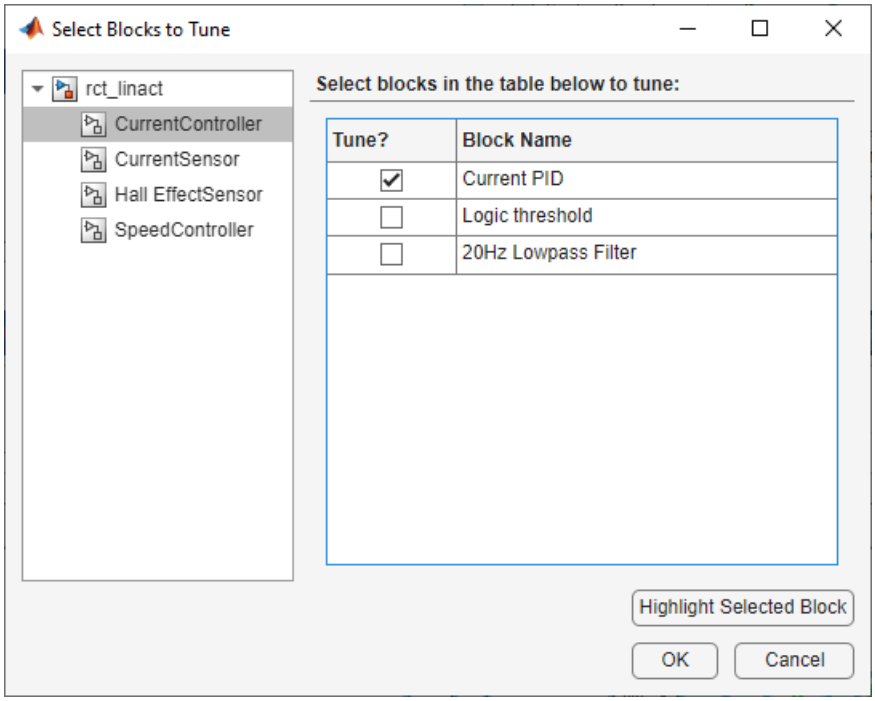


Figure 9: Selecting Tuned Block `Current PID`.

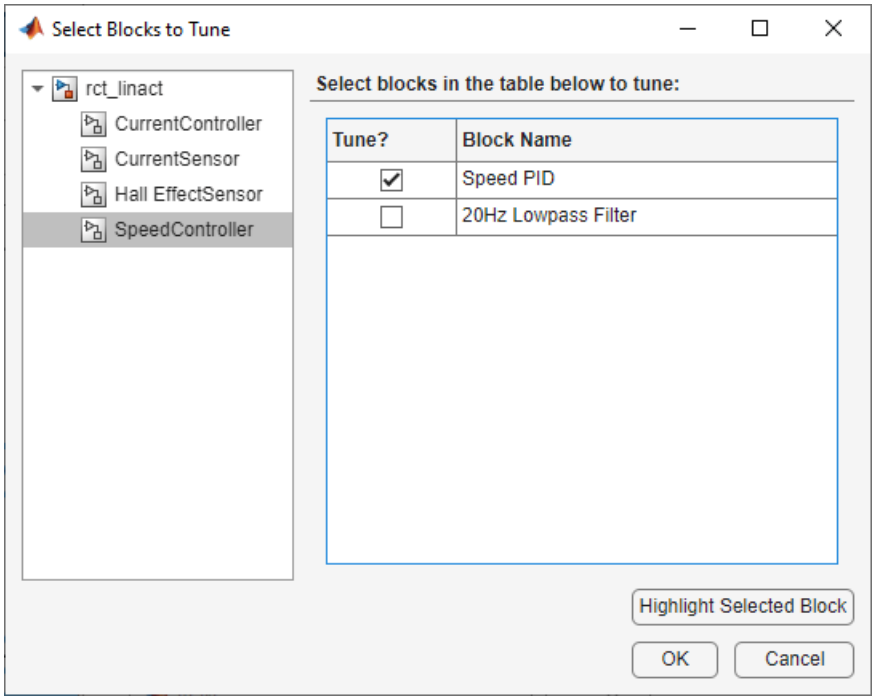


Figure 10: Selecting Tuned Block `Speed PID`.

Selected tuned blocks `Current PID` and `Speed PID` show in the editor for tuned blocks.

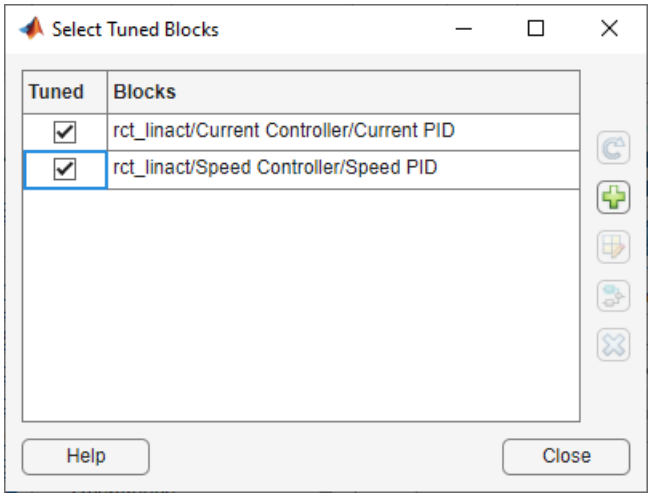


Figure 11: Editor Updated with Selected Tuned Blocks.

They also appear in the `Tuned Blocks` section of Data Browser on the left side of Control System Tuner.

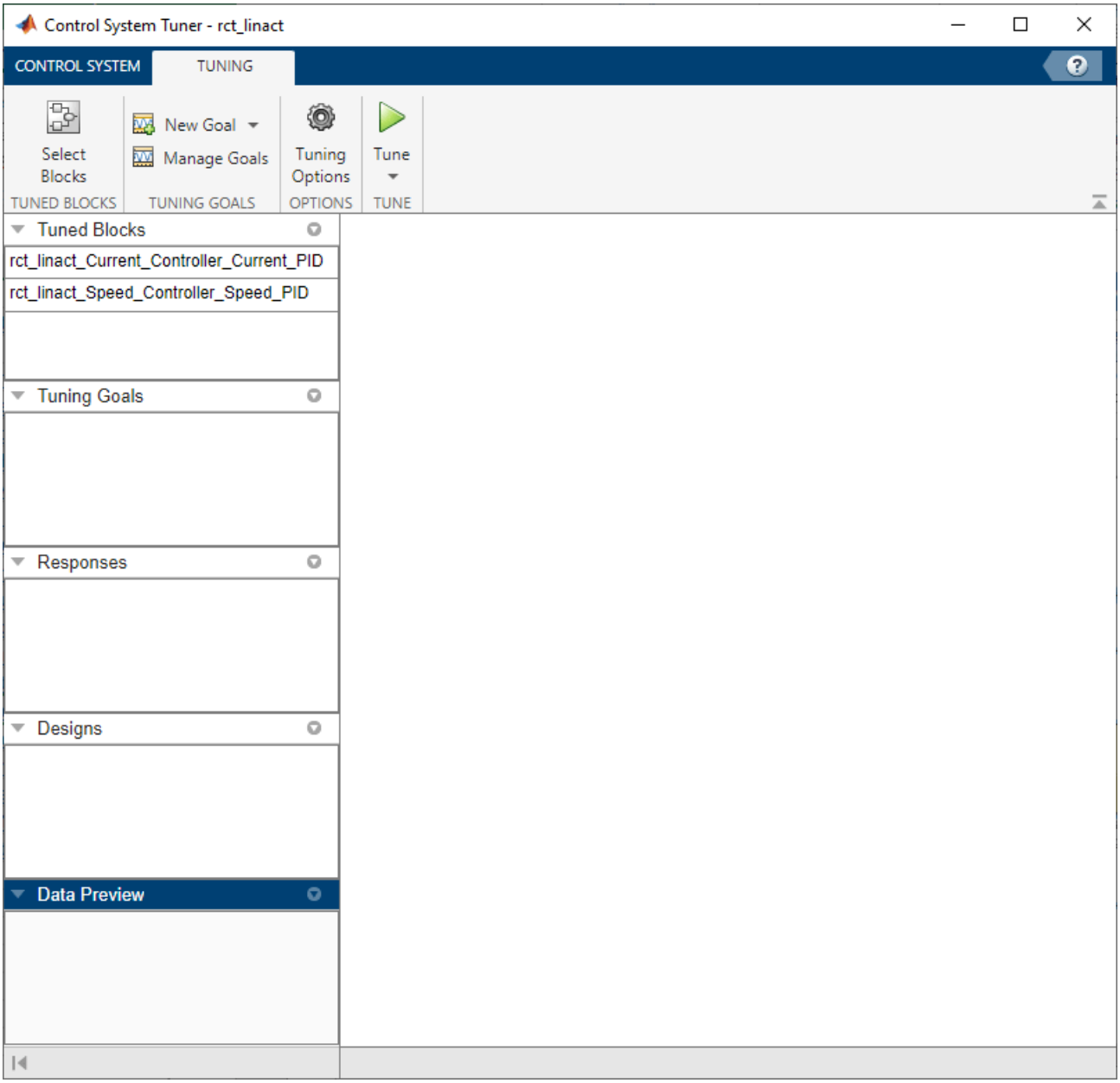


Figure 12: Updated Tuned Blocks in Control System Tuner.

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`.

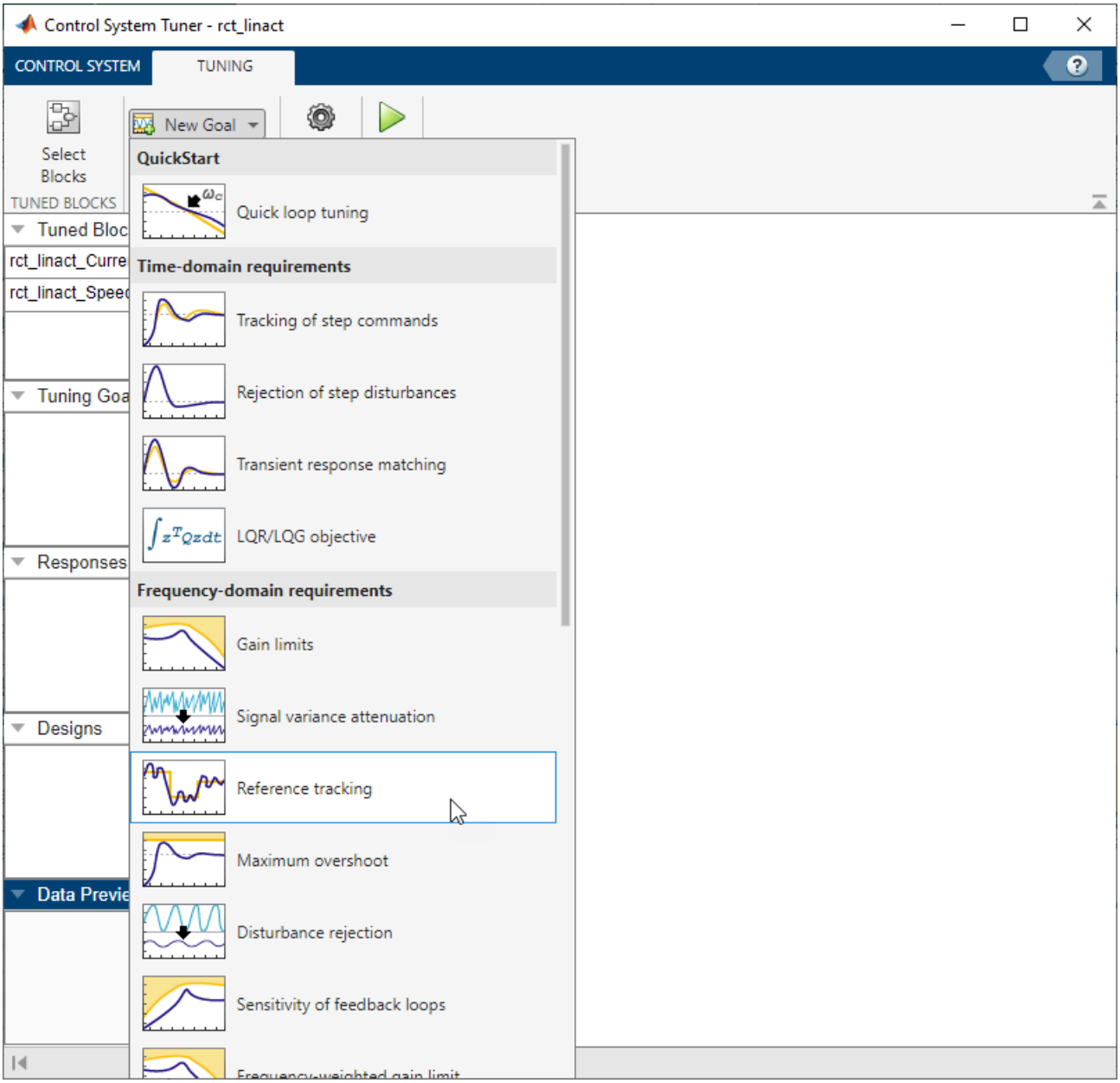


Figure 13: Available Goals for Selection in Control System Tuner.

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.

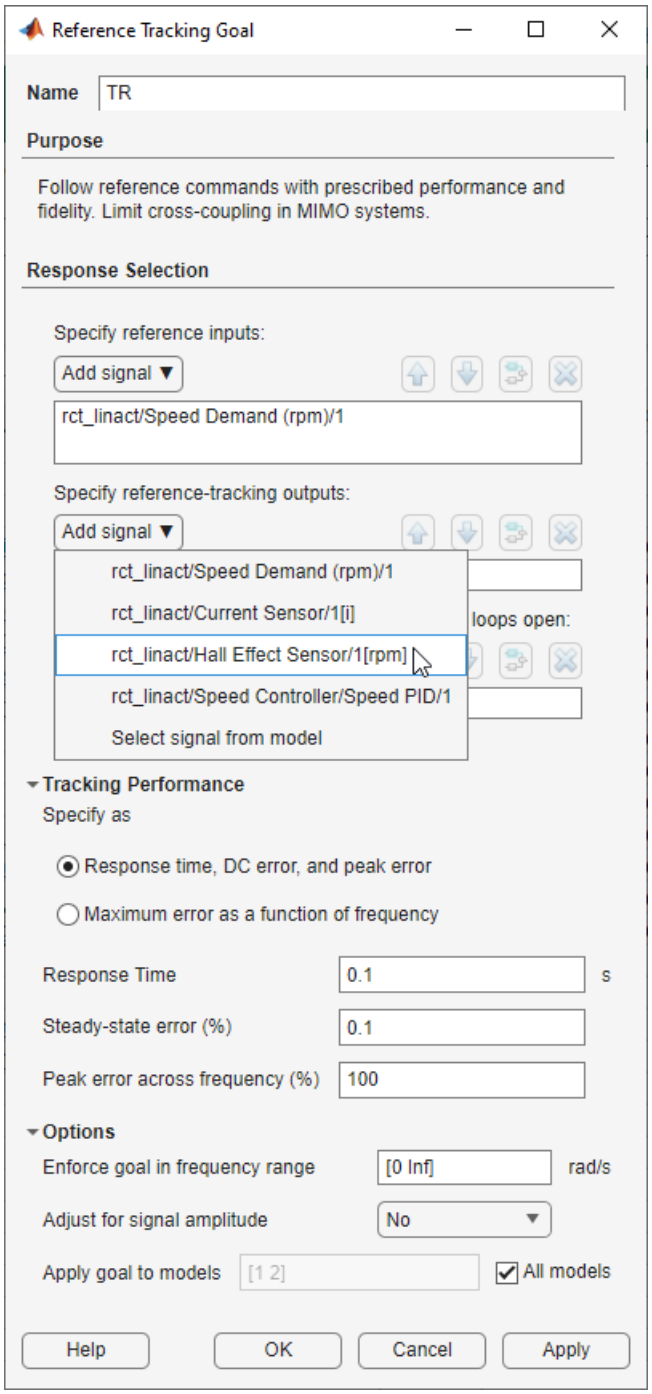


Figure 14: Reference Tracking Dialog in Control System Tuner.

The plot for specified tracking goal appears in Control System Tuner and `Tuning Goals` section of Data Browser on the left side is updated.

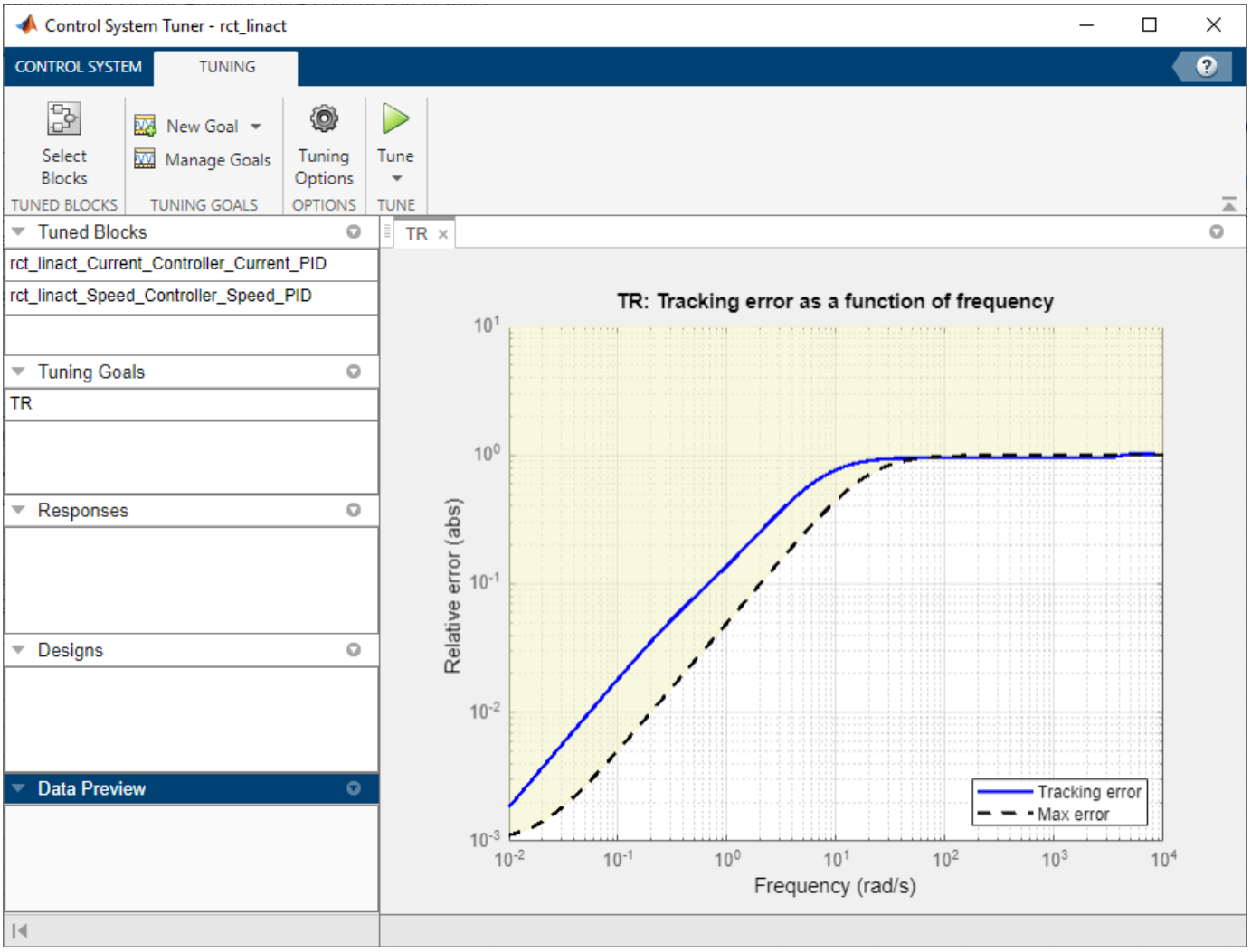


Figure 15: Tracking Tuning Goal in Control System Tuner.

You can now tune the proportional and integral gains with Control System Tuner from clicking Tune button. The plot for tracking goal is updated

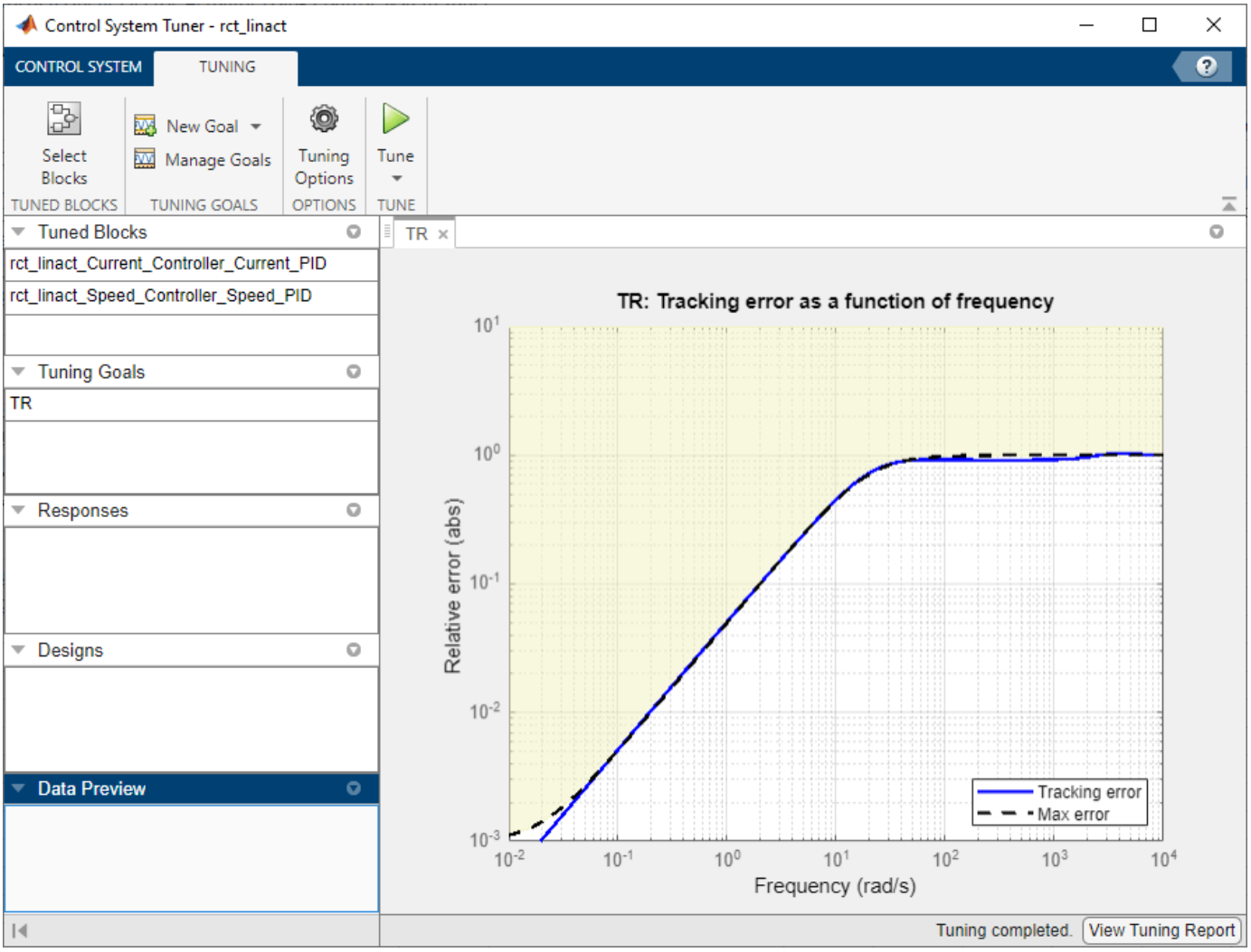


Figure 16: Updated Tracking Goal Plot with Tuned Blocks in Control System Tuner.

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.

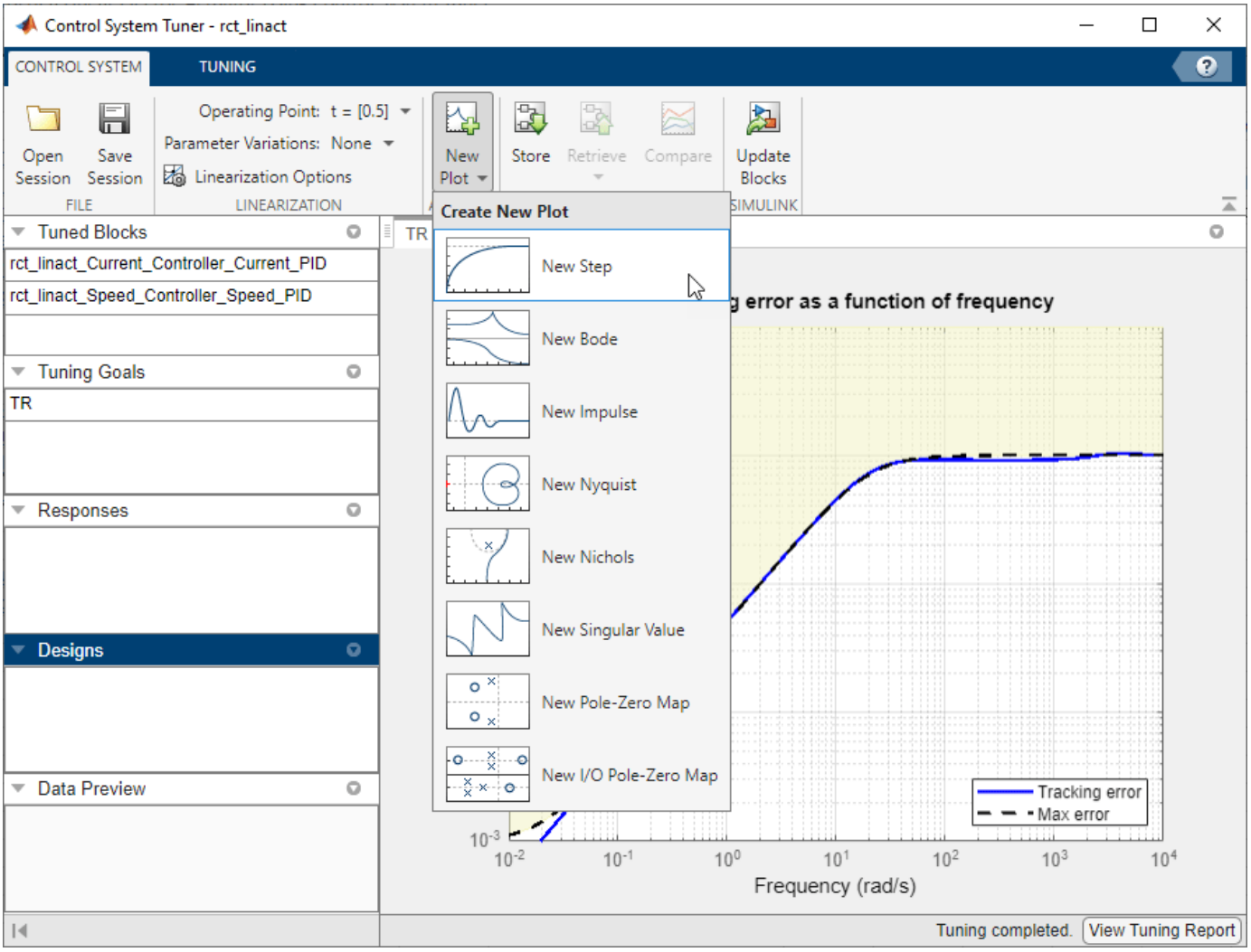


Figure 17: New Plot in Control System Tuner.

Specify the closed-loop response from speed demand to speed by the step plot dialog.

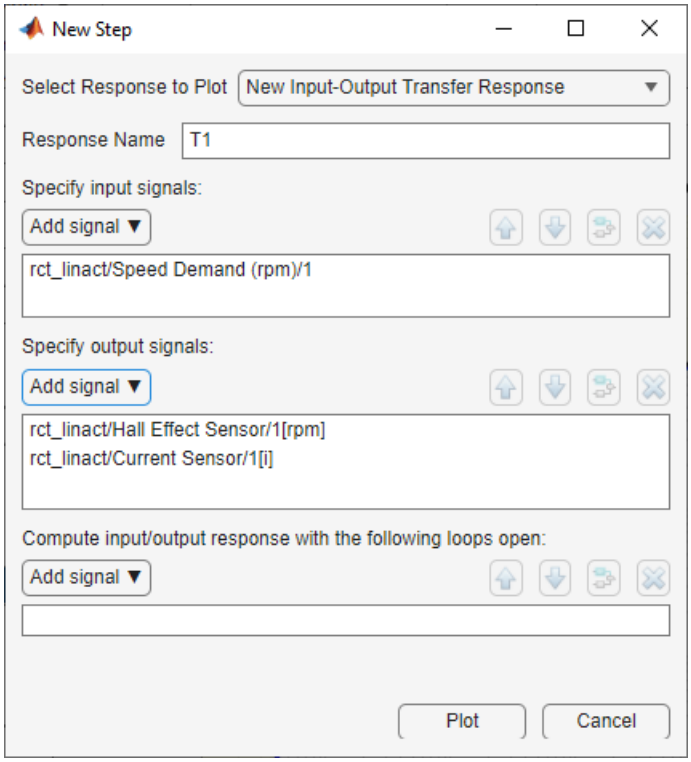


Figure 18: Step Plot Dialog in Control System Tuner.

You see the step plot of the response in Control System Tuner.

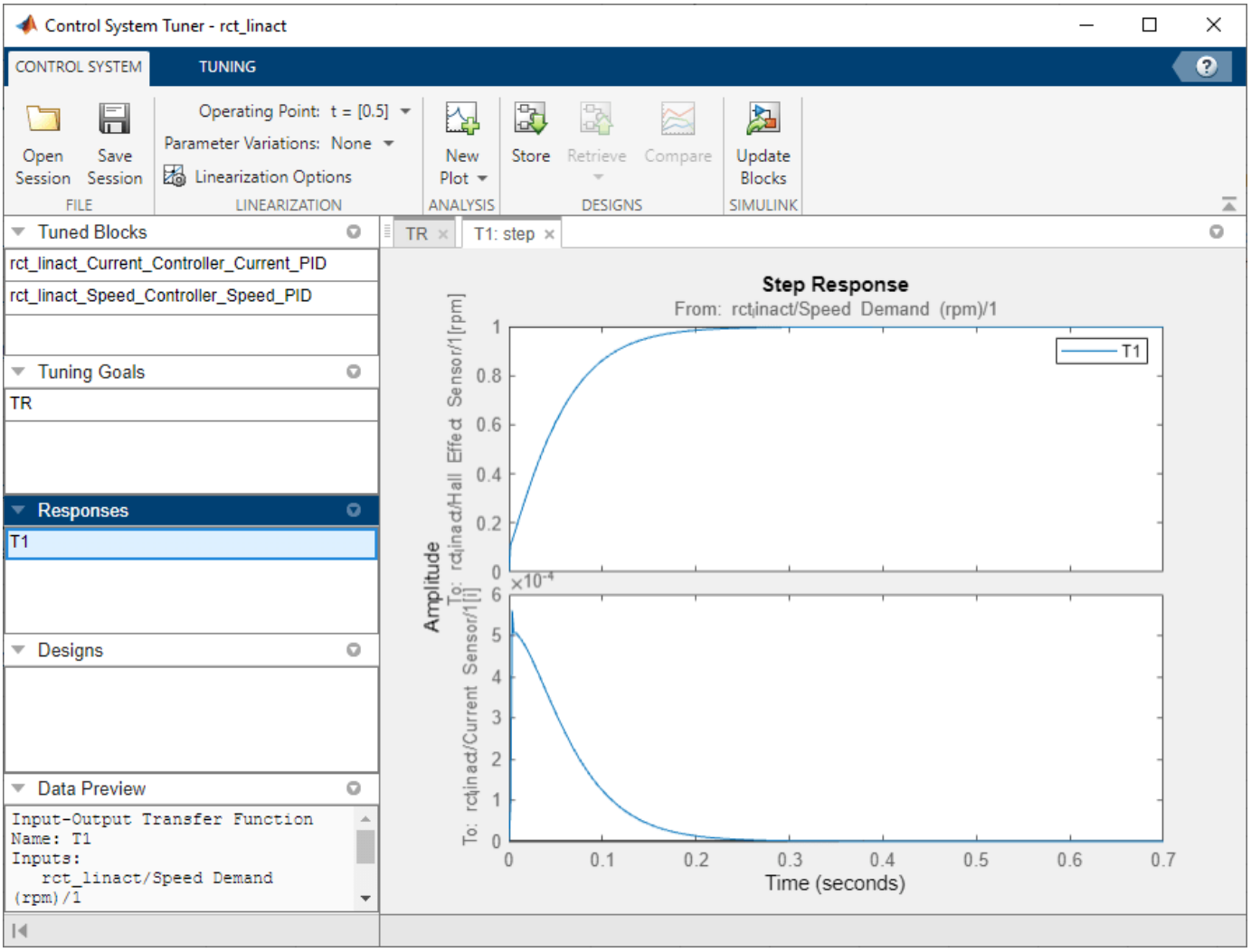


Figure 19: Step Plot in Control System Tuner.

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.

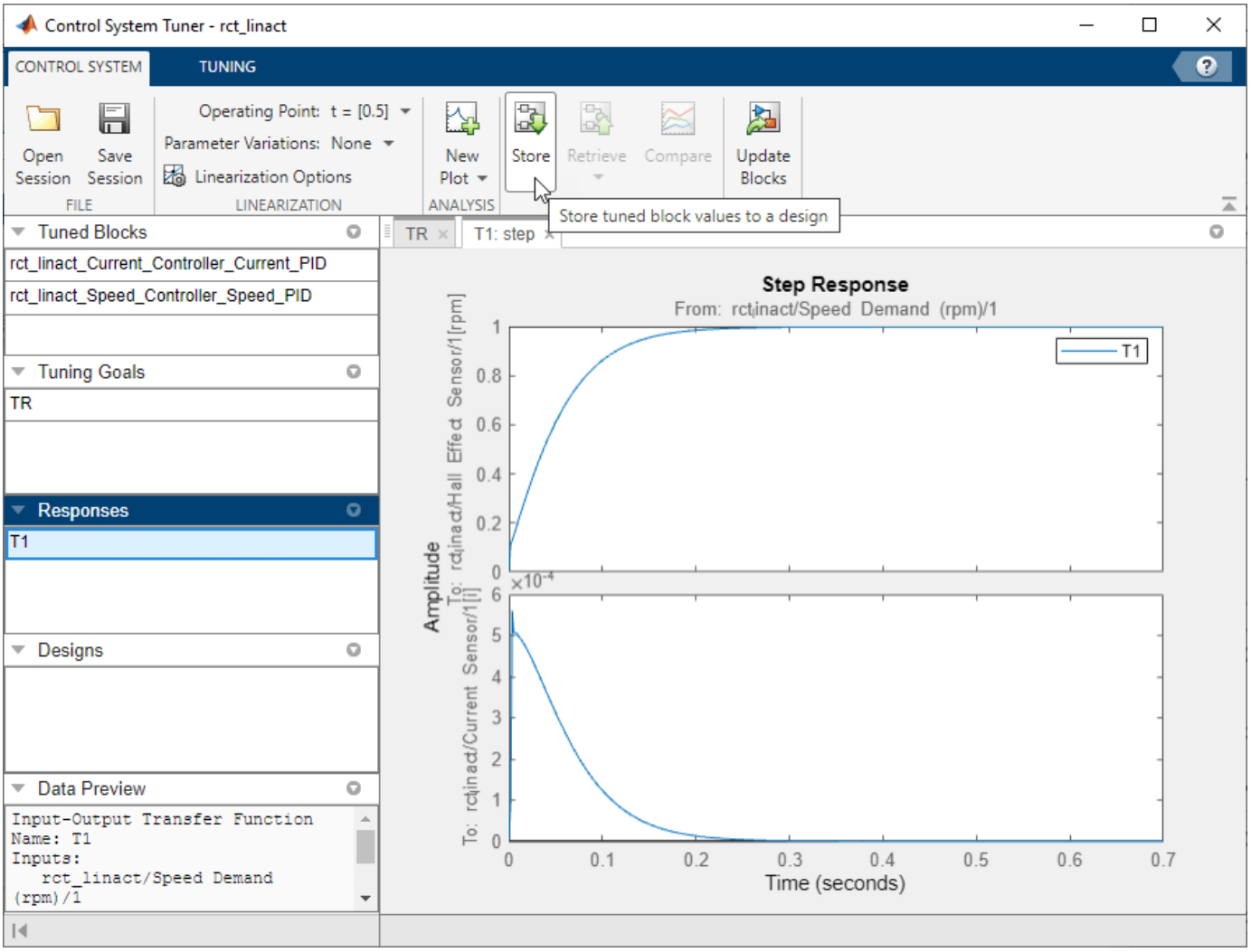


Figure 20: Stored Values of Tuned Blocks in Control System Tuner.

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).

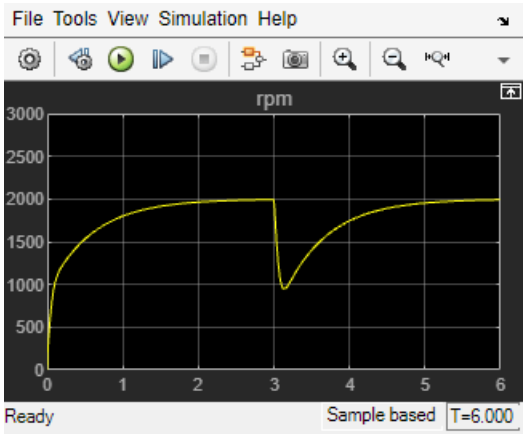


Figure 21: Nonlinear Simulation of Tuned Controller.

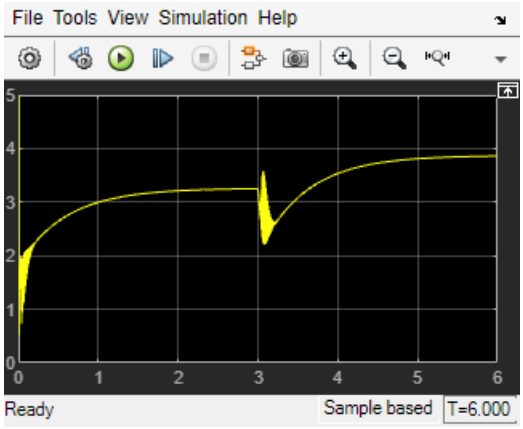


Figure 22: Current Controller Output (limited to plus/minus 5).

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.

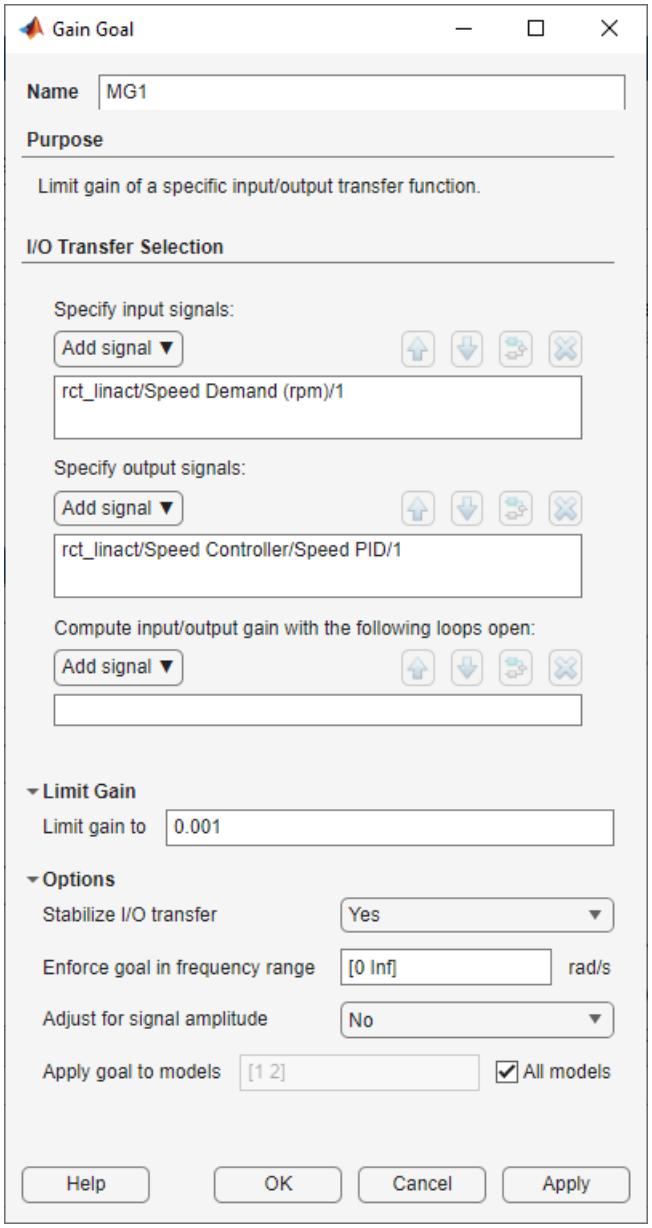


Figure 23: Gain Goal Dialog from Speed Demand to Control Signal of Speed PID.

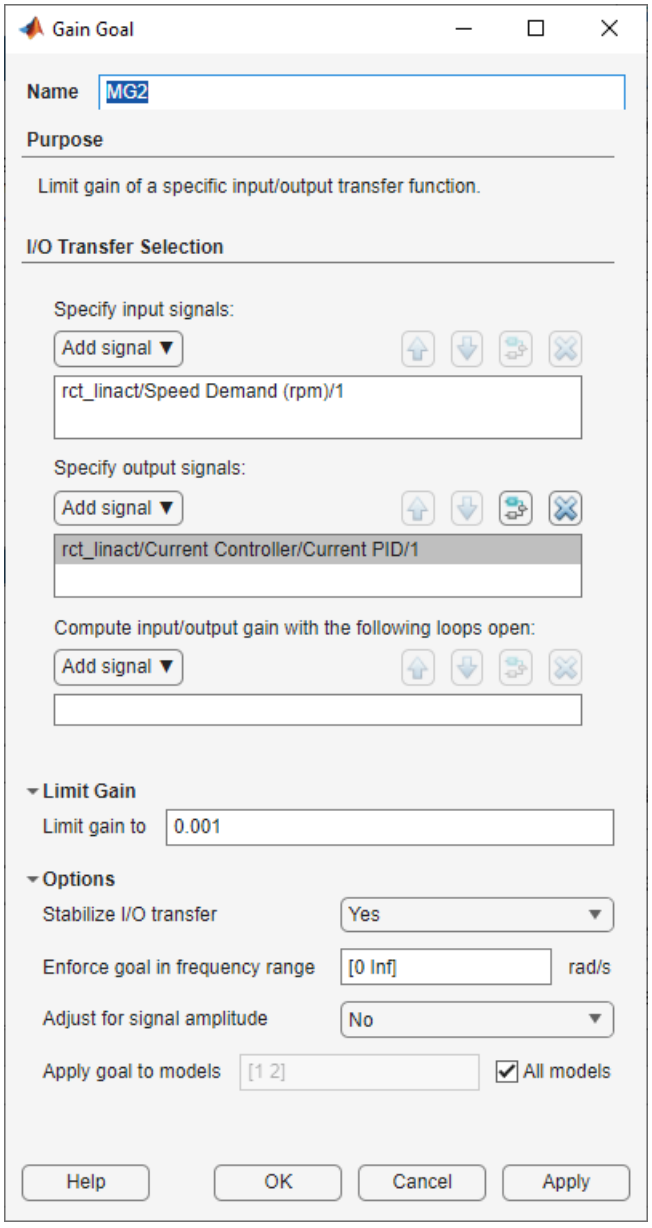


Figure 24: Gain Goal Dialog from Speed Demand to Control Signal of Current PID.

New gain goals appear in Tuning Goals section of Control System Tuner.

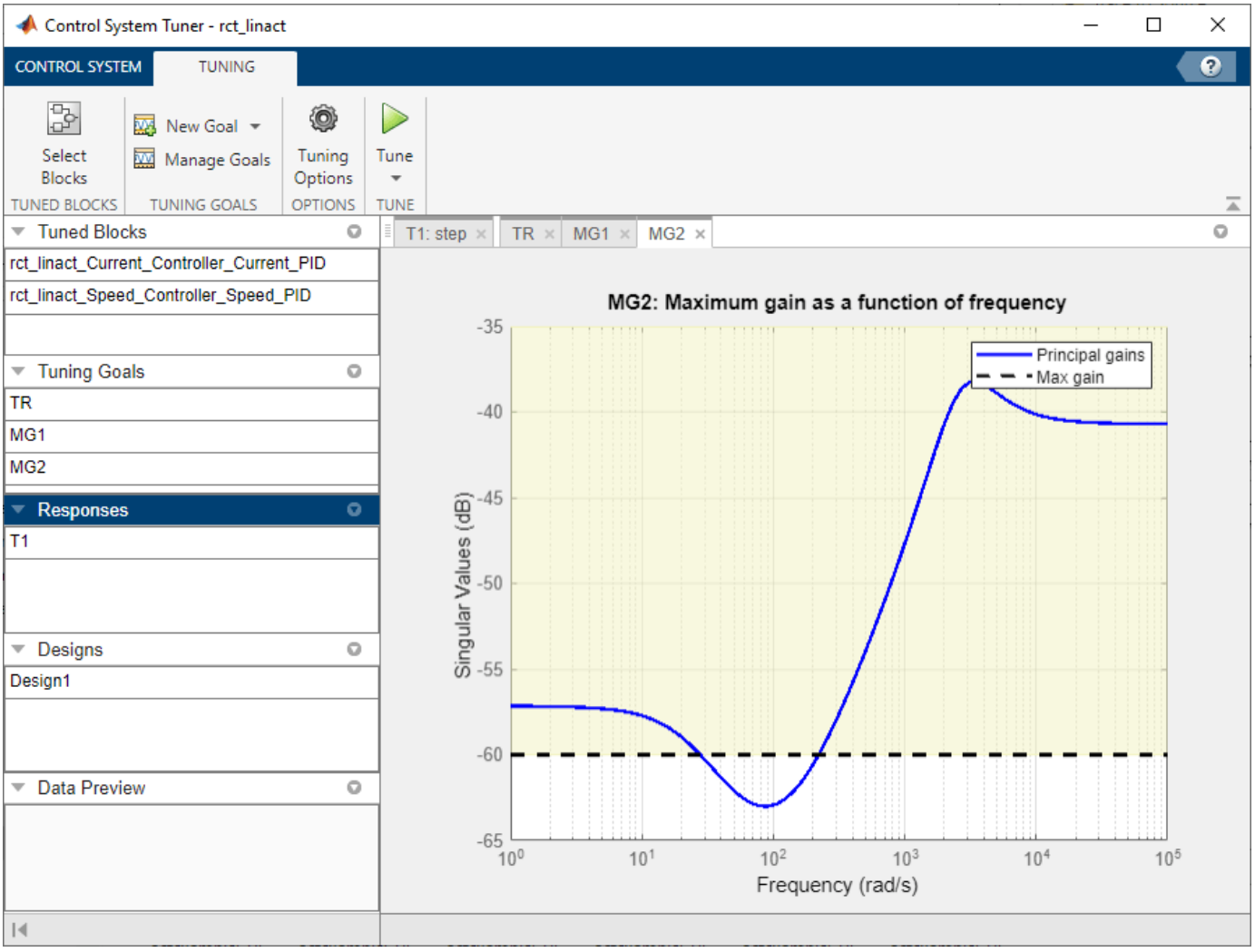


Figure 25: Two Gain Goals Added to Control System Tuner.

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).

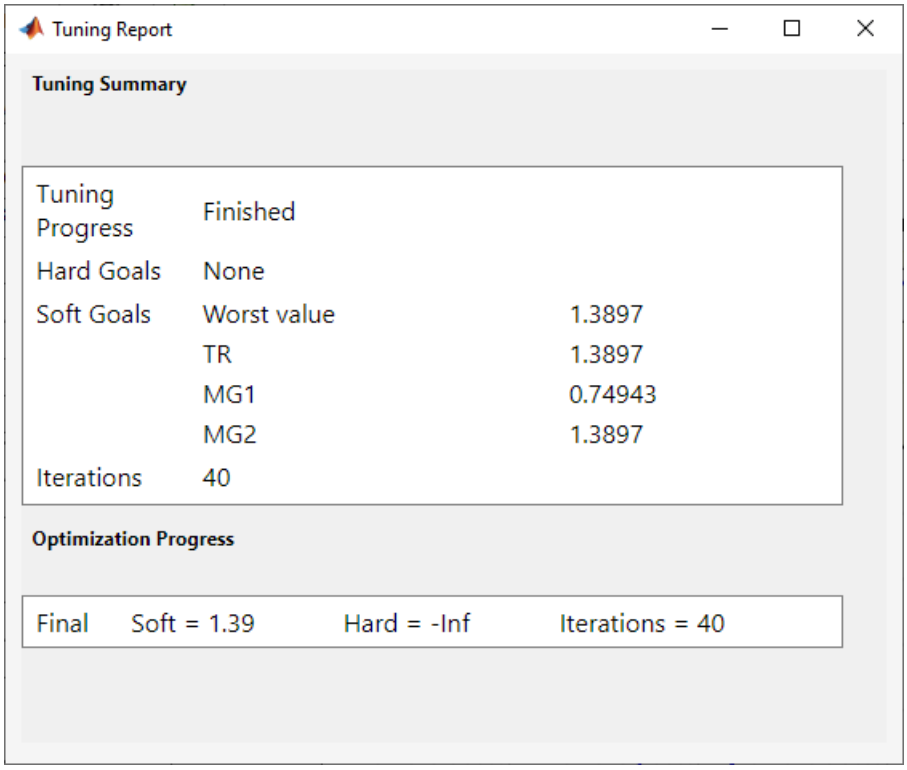


Figure 26: Tuning Report After Retuning.

Next compare the two designs in the linear domain by clicking `Compare` in `Control System` tab.

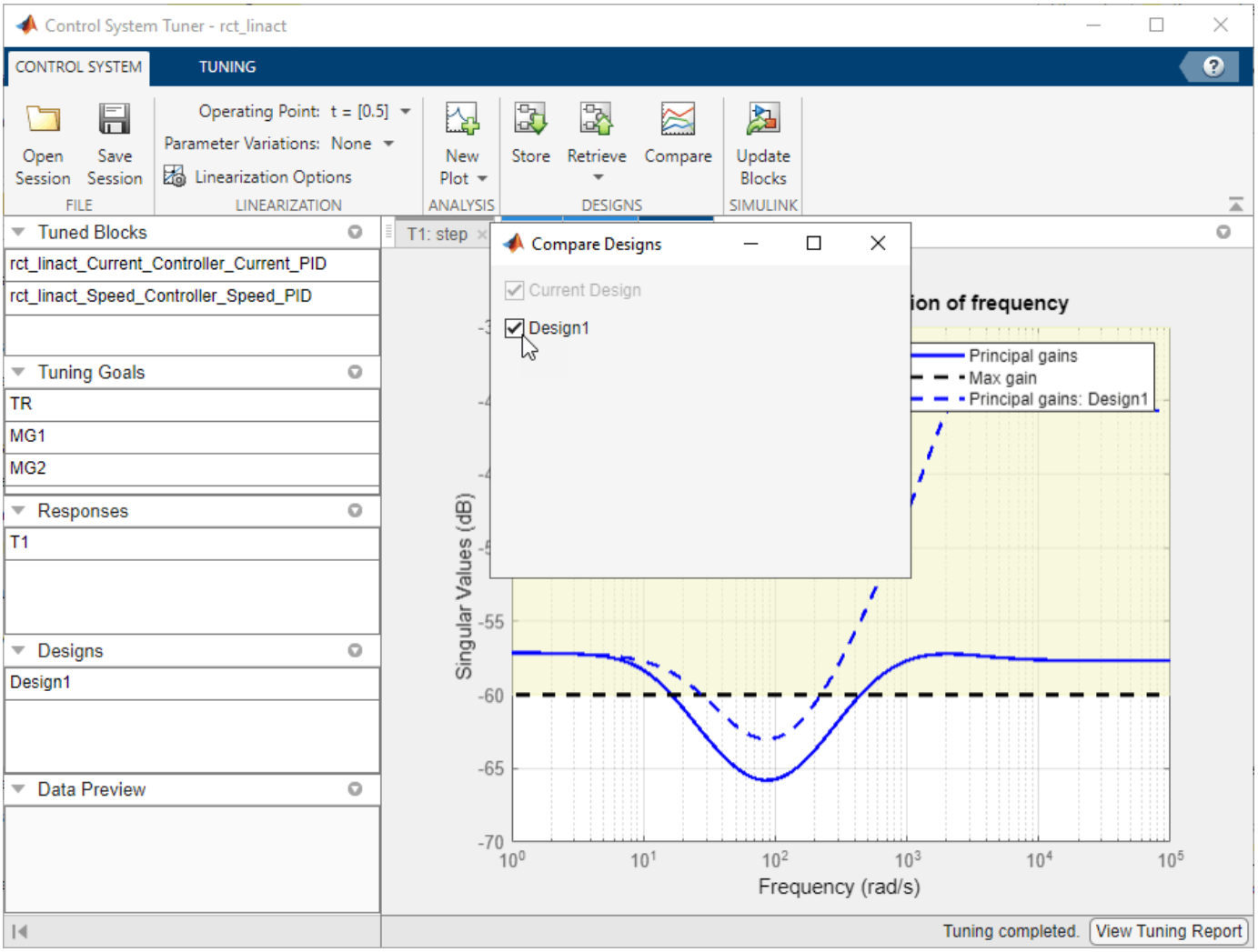


Figure 27: Comparing Two Designs.

The second design is less aggressive but still meets the response time requirement.

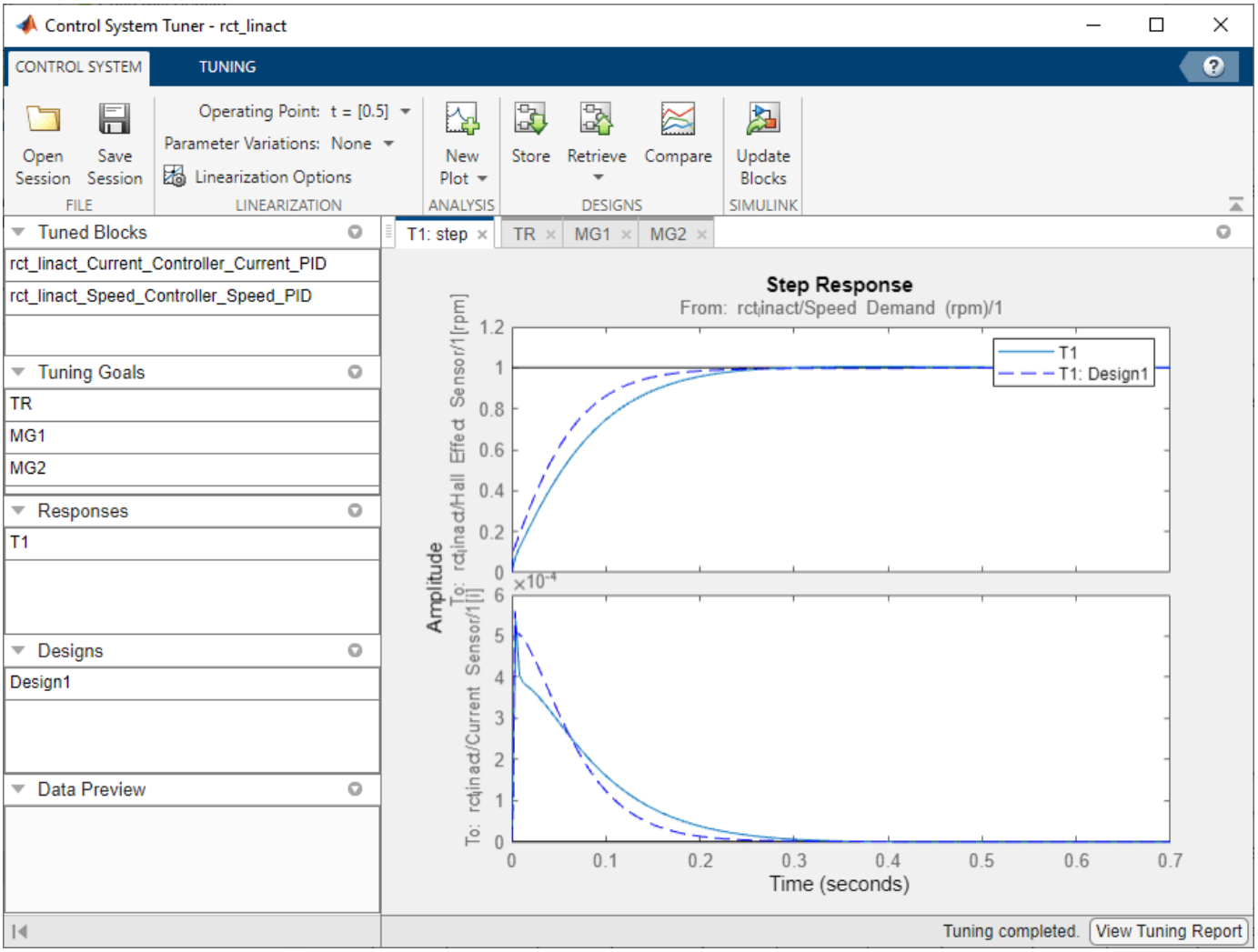


Figure 28: Step Responses of Two Designs.

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.

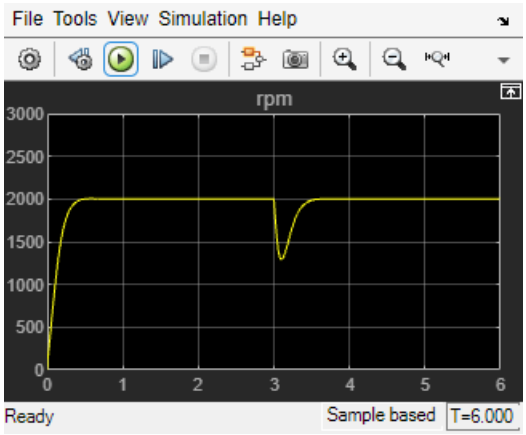


Figure 29: Nonlinear Response of Tuning with Gain Constraints.

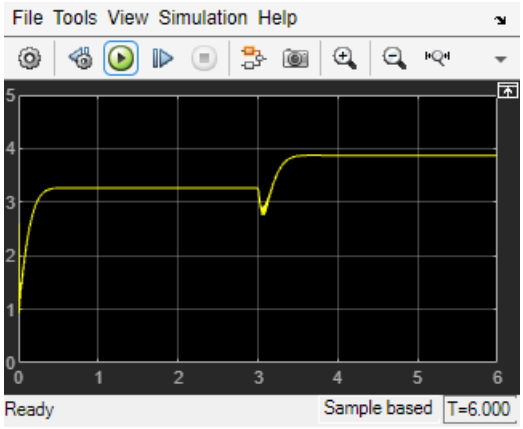


Figure 30: Current Controller Output.

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” on page 18-76

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.

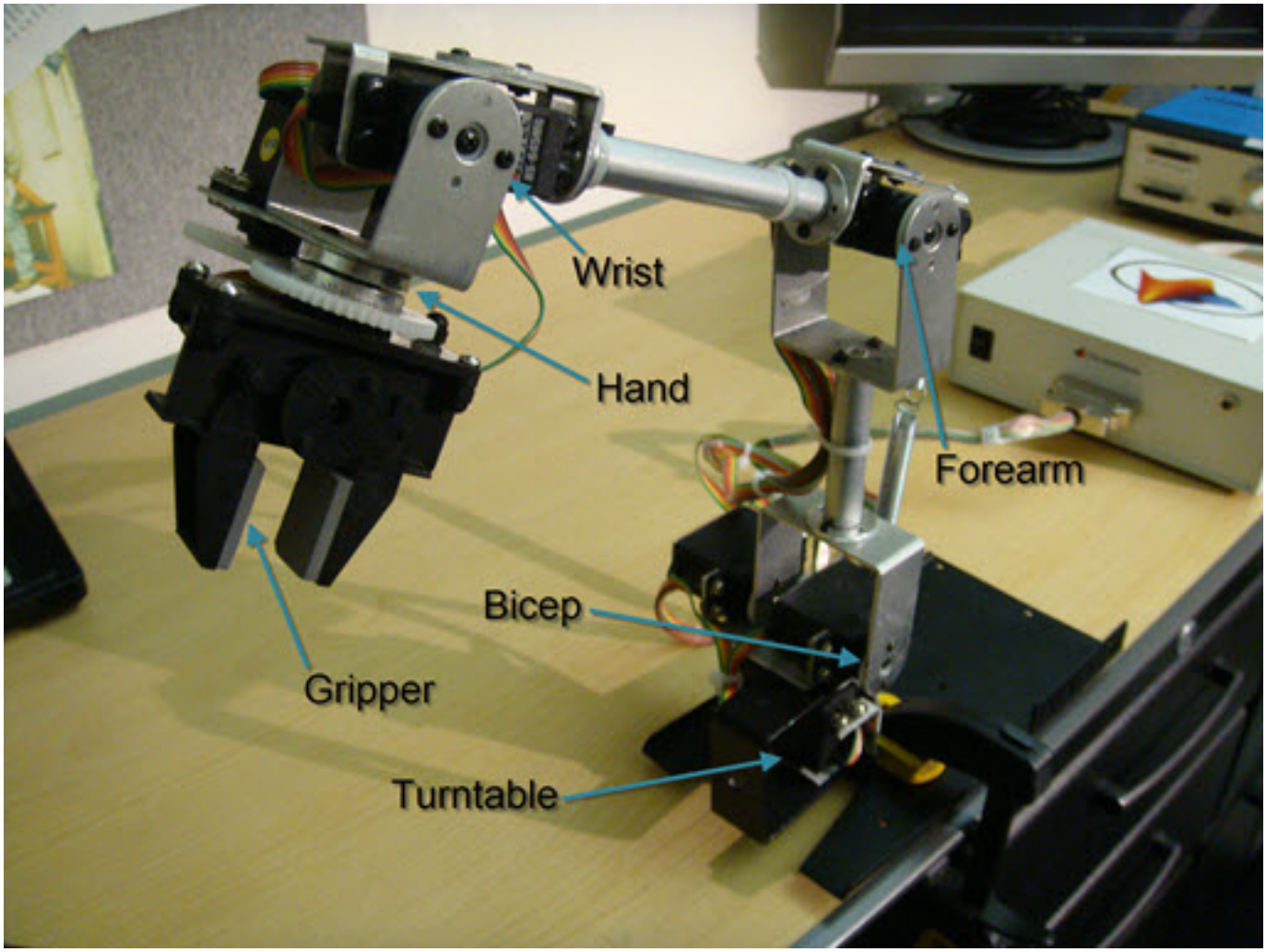


Figure 1: Robotic arm manipulator.

The file "cst_robotarm.slx" contains a Simulink model of the electrical and mechanical components of this system.

```
open_system("cst_robotarm");
```

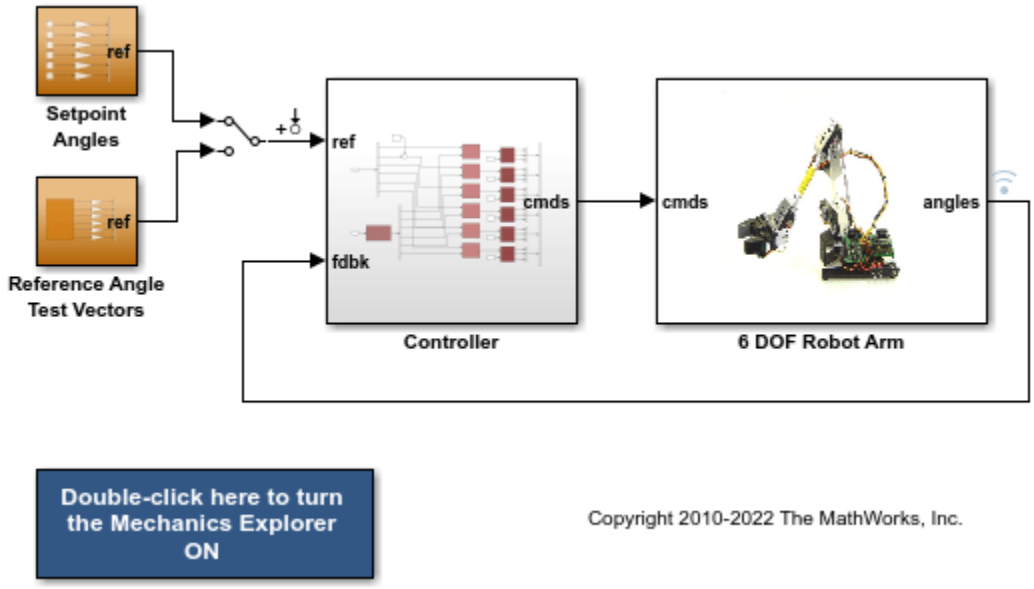


Figure 2: Simulink model of robotic arm.

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).

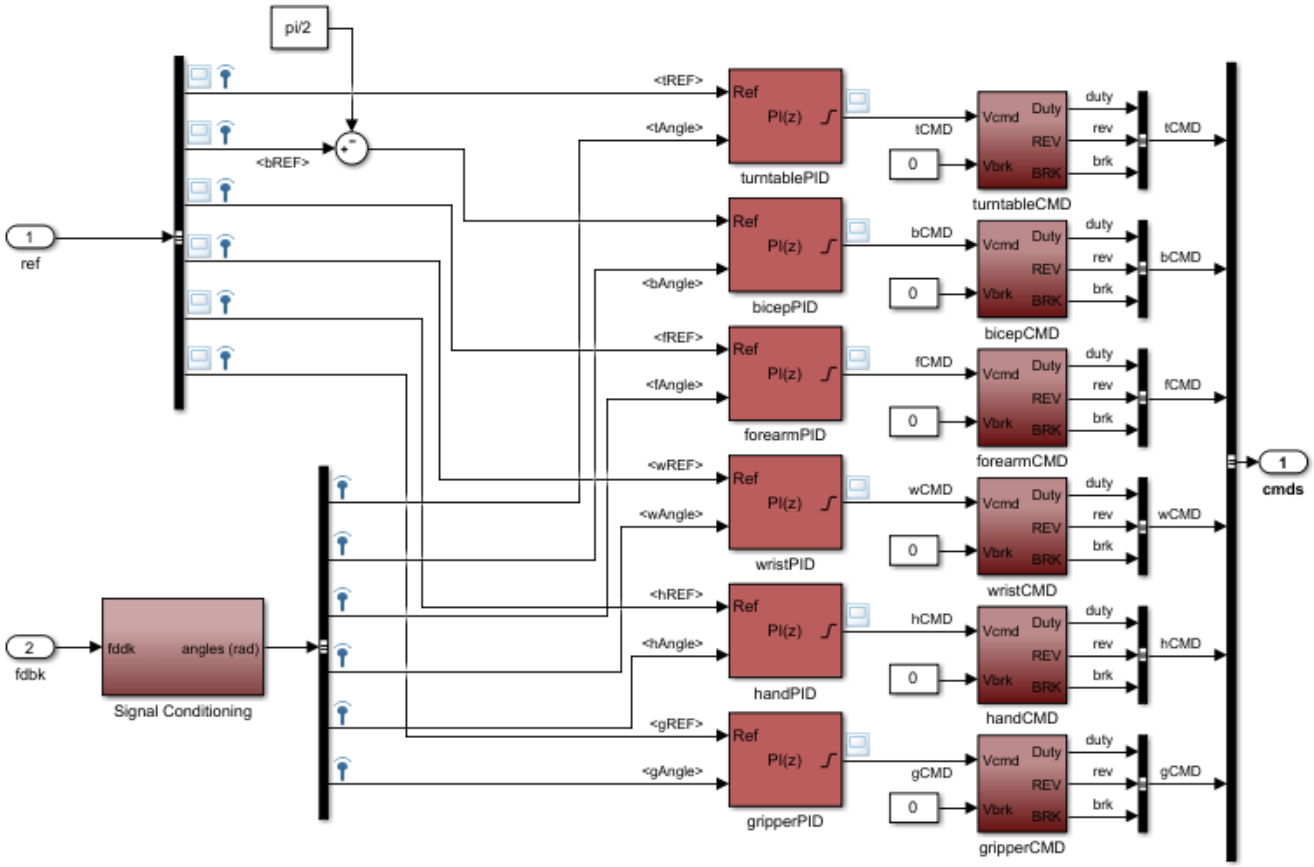
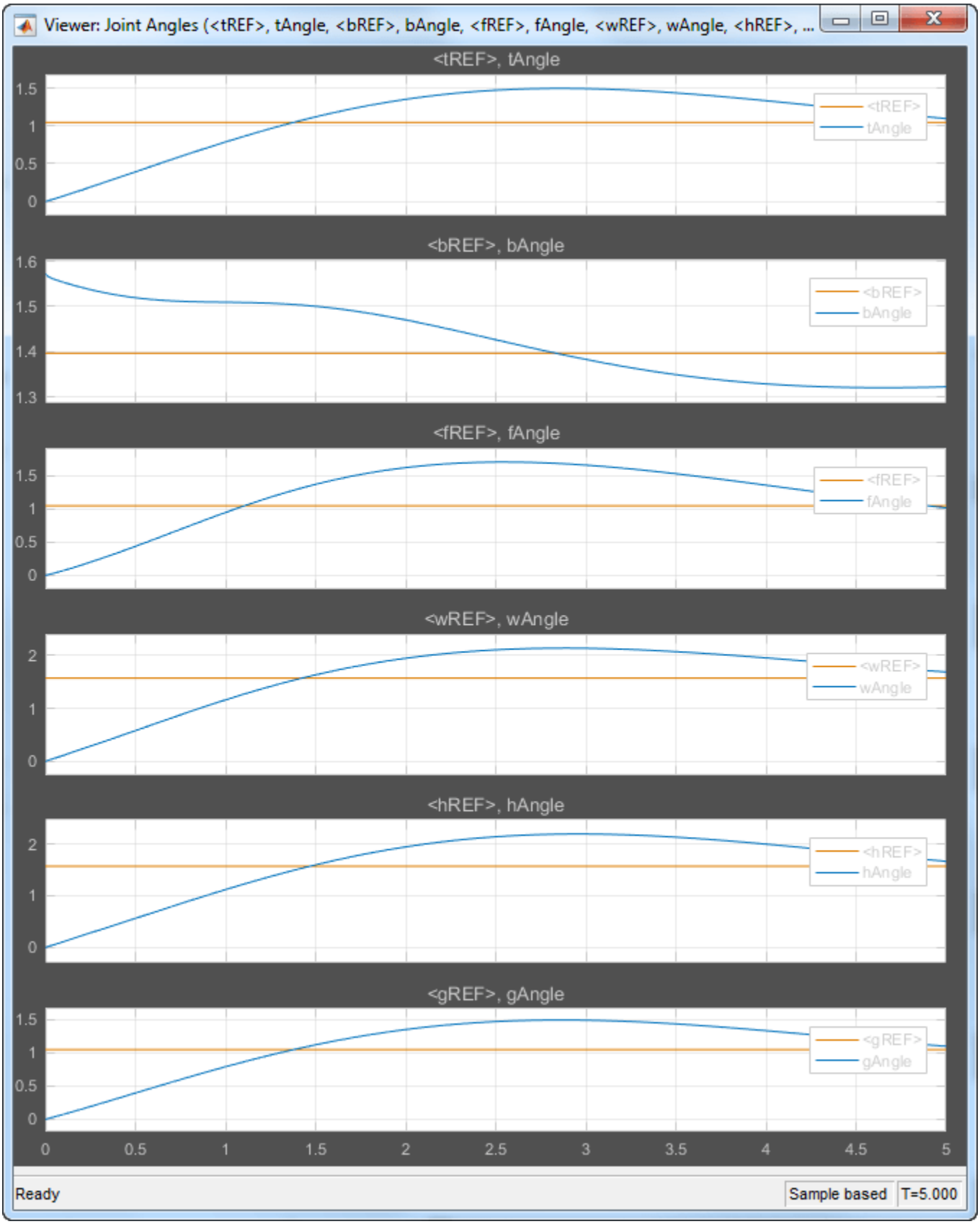


Figure 3: Controller structure.

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
Ready
Sample based
T=5.000

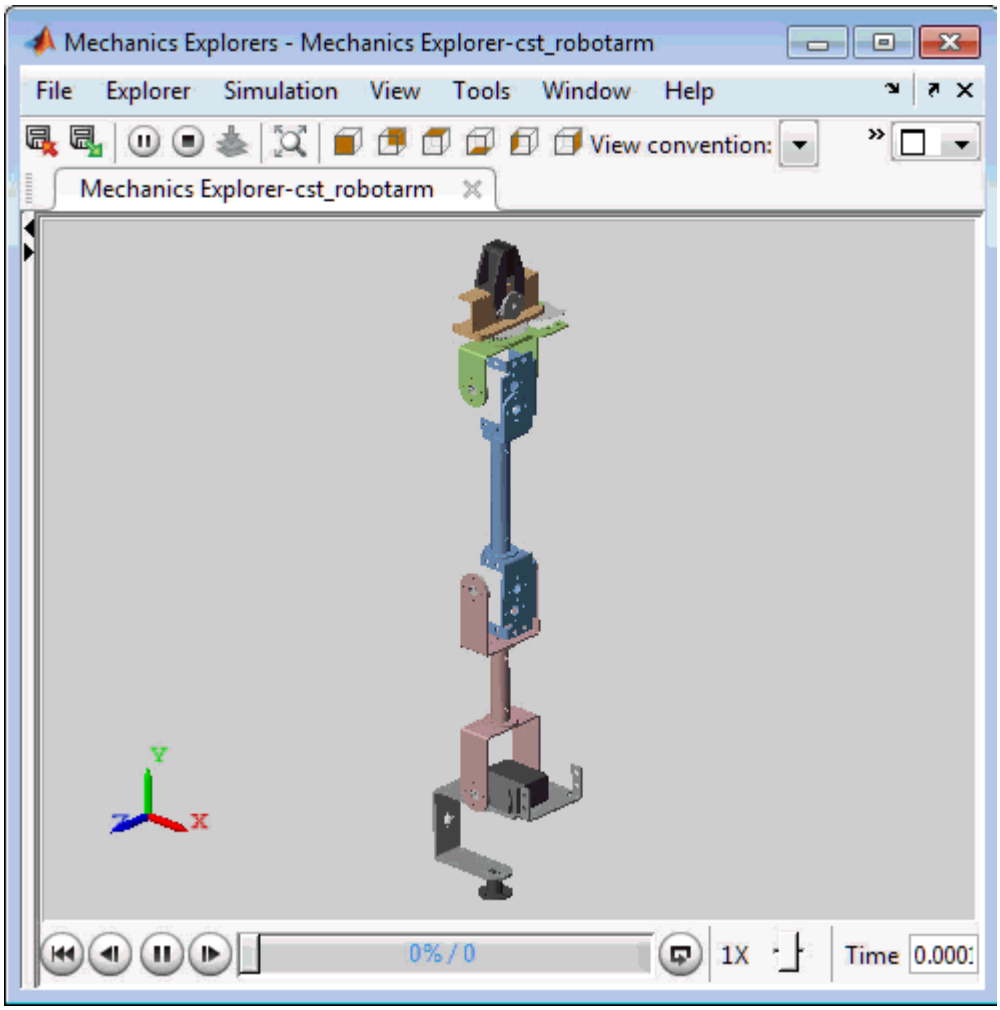


Figure 4: Untuned response.

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).

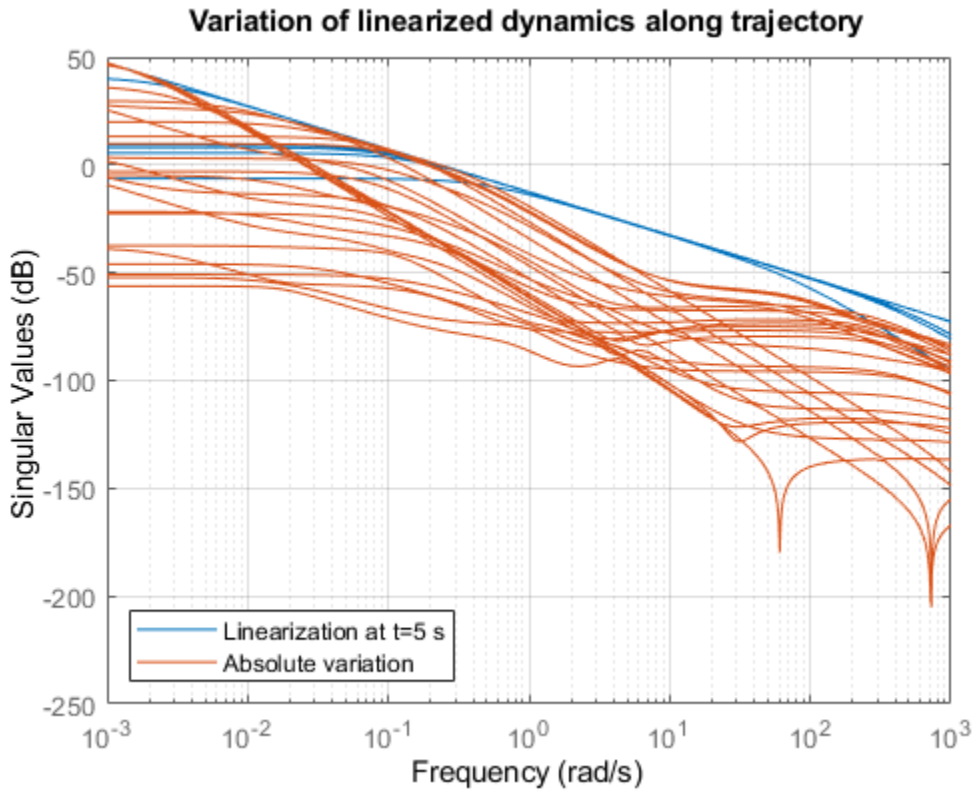
```
SnapshotTimes = 0:1:5;
% Plant is from PID outputs to Robot Arm outputs
LinIOs = [...
   linio('cst_robotarm/Controller/turntablePID',1,'openinput'),...
   linio('cst_robotarm/Controller/bicepPID',1,'openinput'),...
   linio('cst_robotarm/Controller/forearmPID',1,'openinput'),...
   linio('cst_robotarm/Controller/wristPID',1,'openinput'),...
   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)
```

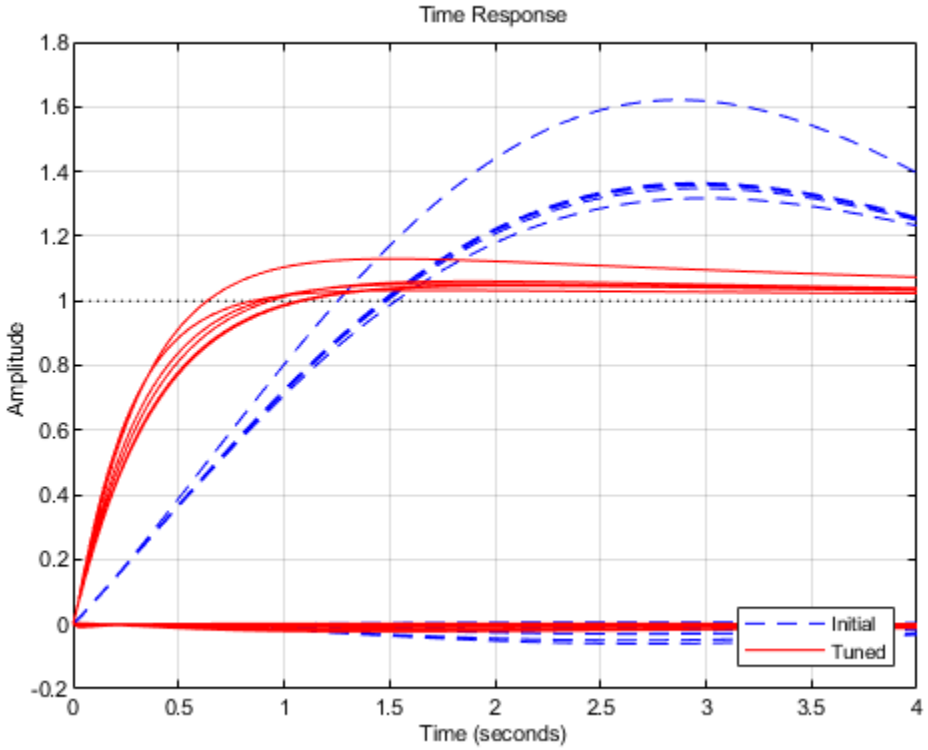
In its simplest use, `looptune` only needs to know the target control bandwidth, which should be at least twice the reciprocal of the desired response time. Here the desired response time is 1 second so try a target bandwidth of 3 rad/s (bearing in mind that the plant dynamics vary least near 10 rad/s).

```
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.

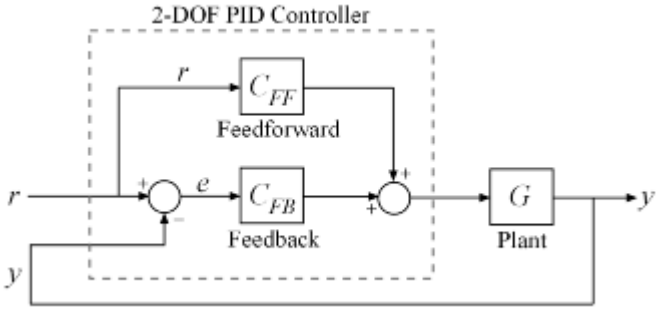


Figure 5: Two degree-of-freedom PID controllers.

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

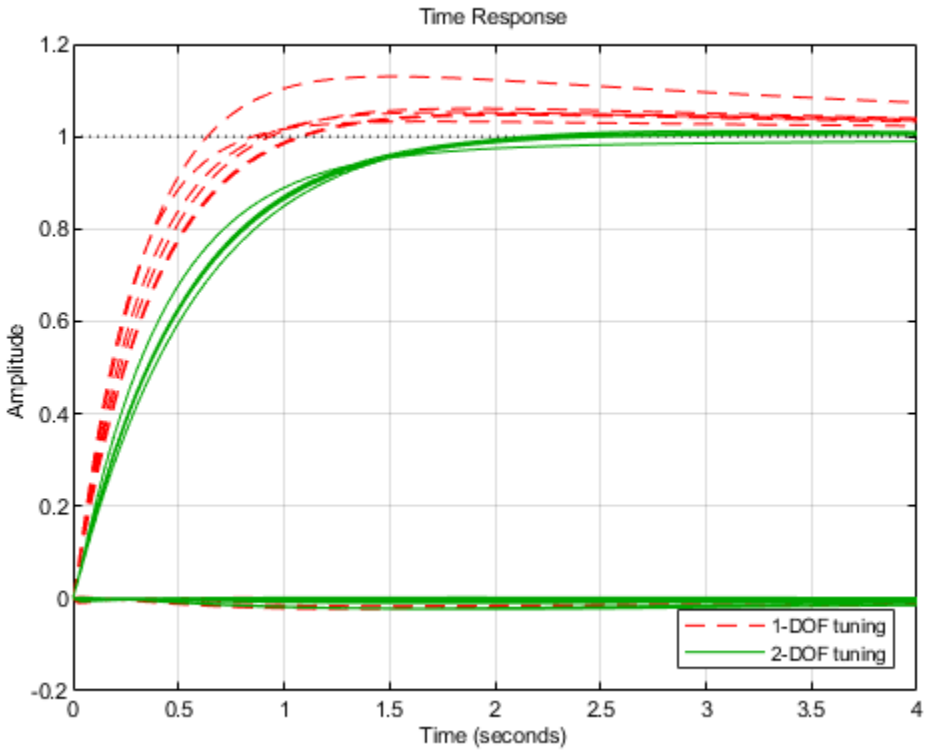
feedforward action and reduce overshoot, replace the bandwidth target by an explicit step tracking requirement from reference angles to joint angles.

```
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)
```

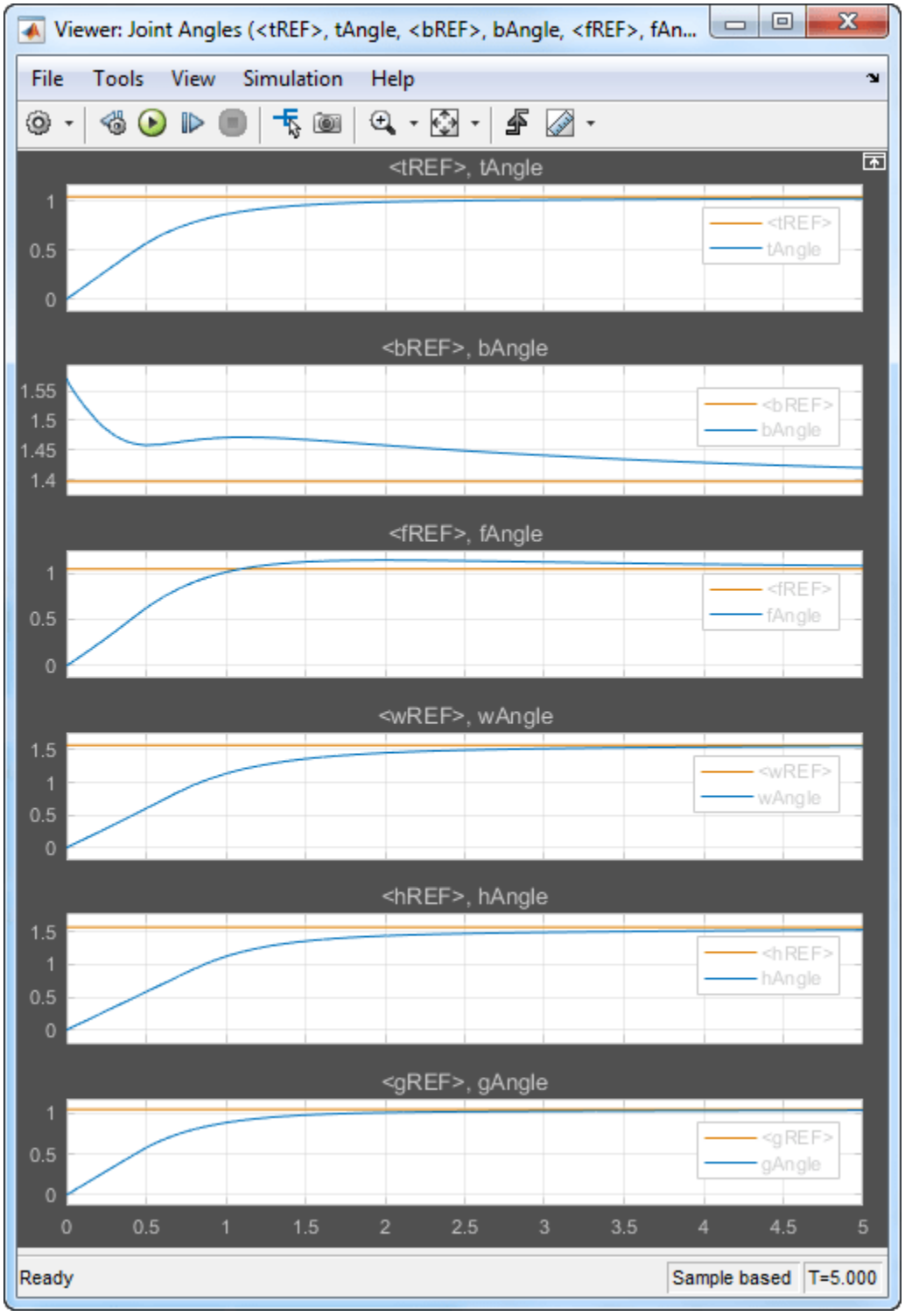


Figure 6: Tuned angular responses.

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).

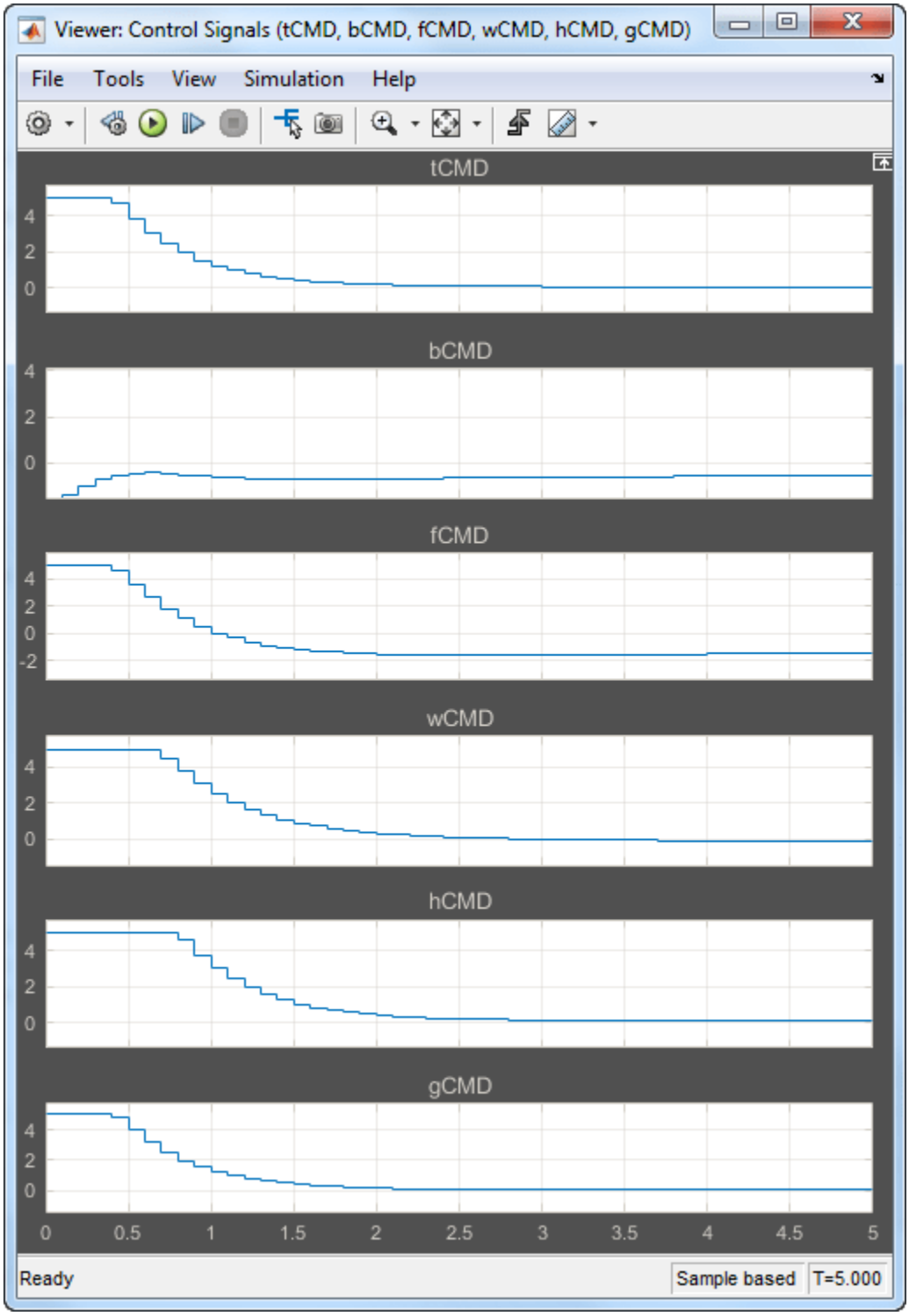
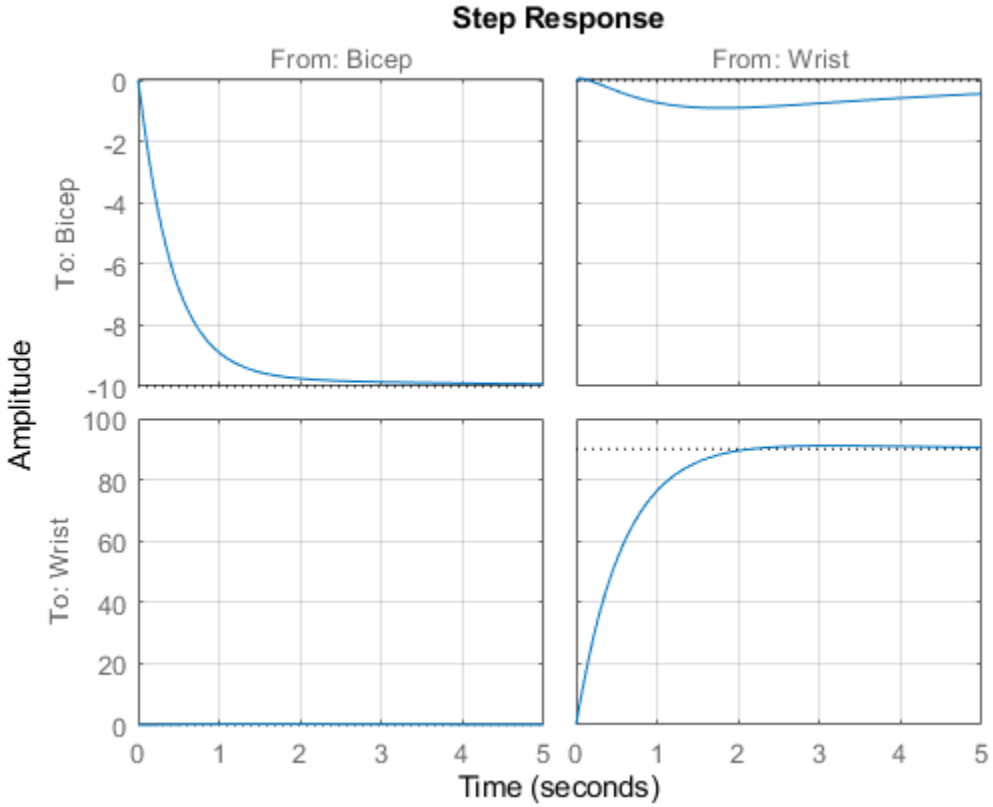


Figure 7: Input voltage to DC motors (control signal).

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);
```

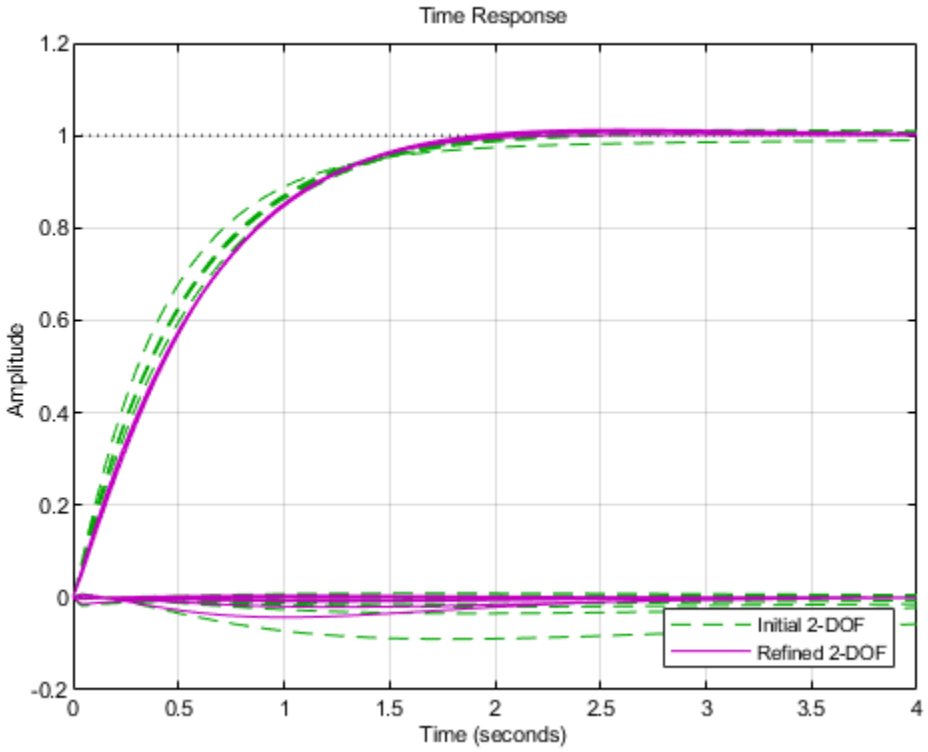
Retune the controller with these refined tuning goals.

```
ST3 = looptune(ST0,Controls,Measurements,TR,UR);
```

```
Final: Peak gain = 1.14, Iterations = 182
```

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.

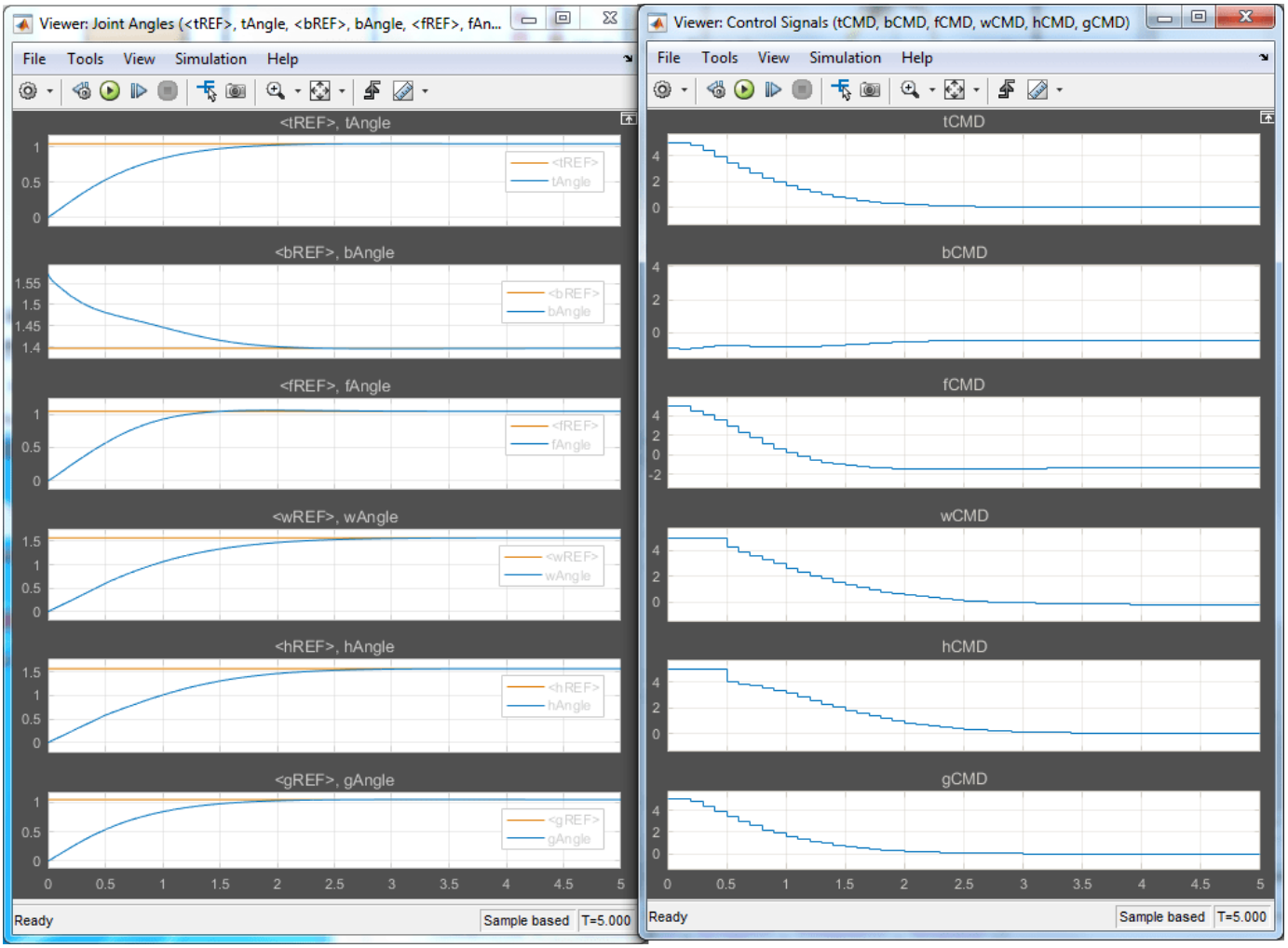


Figure 8: Angular positions and control signals with refined controller.

The 3D animation confirms that the arm now moves quickly and precisely to the desired configuration.

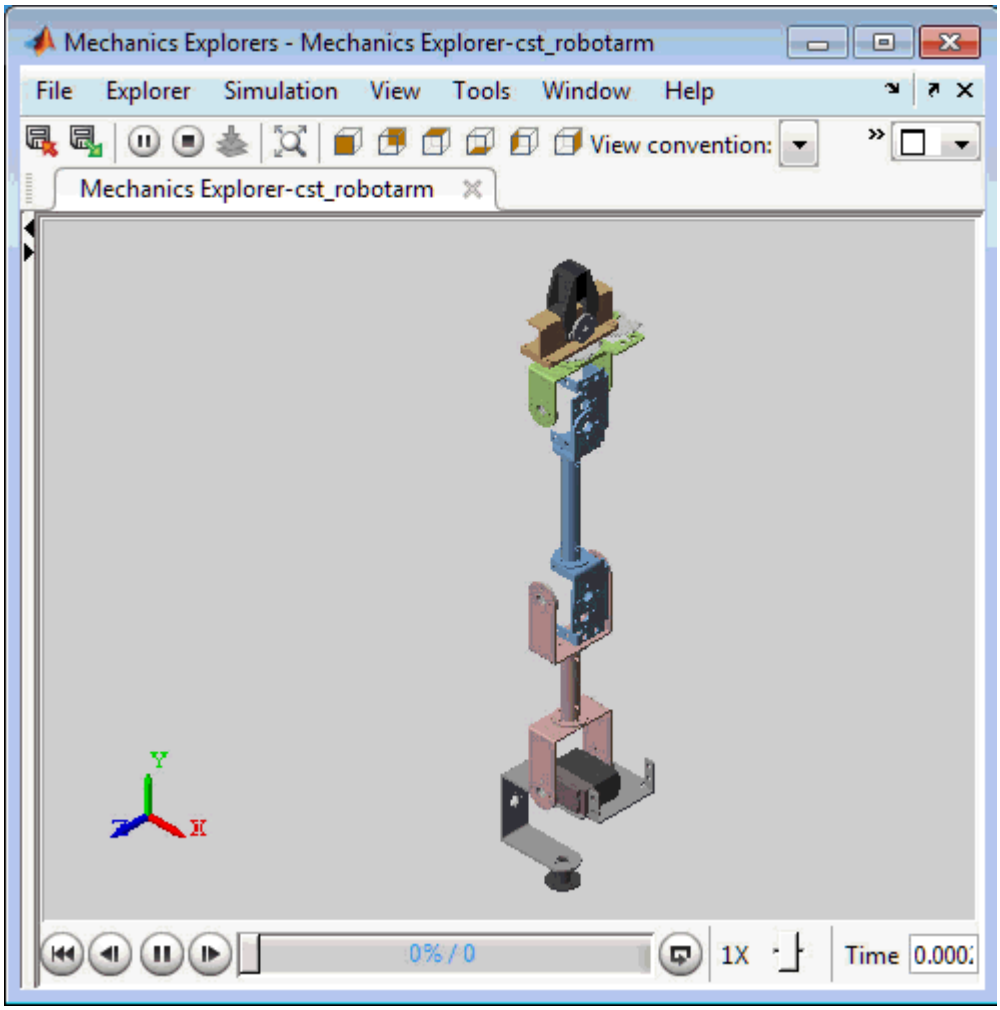


Figure 9: Fine-tuned response.

See Also

`systune` | `TuningGoal.Tracking` | `TuningGoal.MinLoopGain` | `TuningGoal.MaxLoopGain`

Related Examples

- “Active Vibration Control in Three-Story Building” on page 17-15

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™.

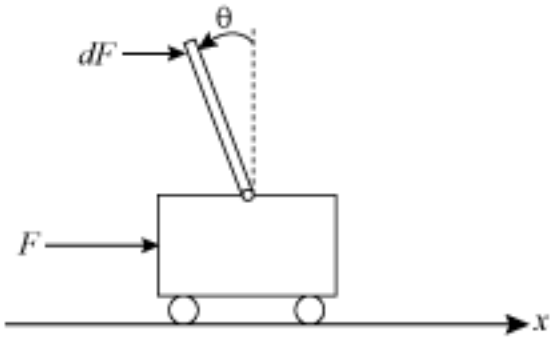


Figure 1: Inverted pendulum on a cart

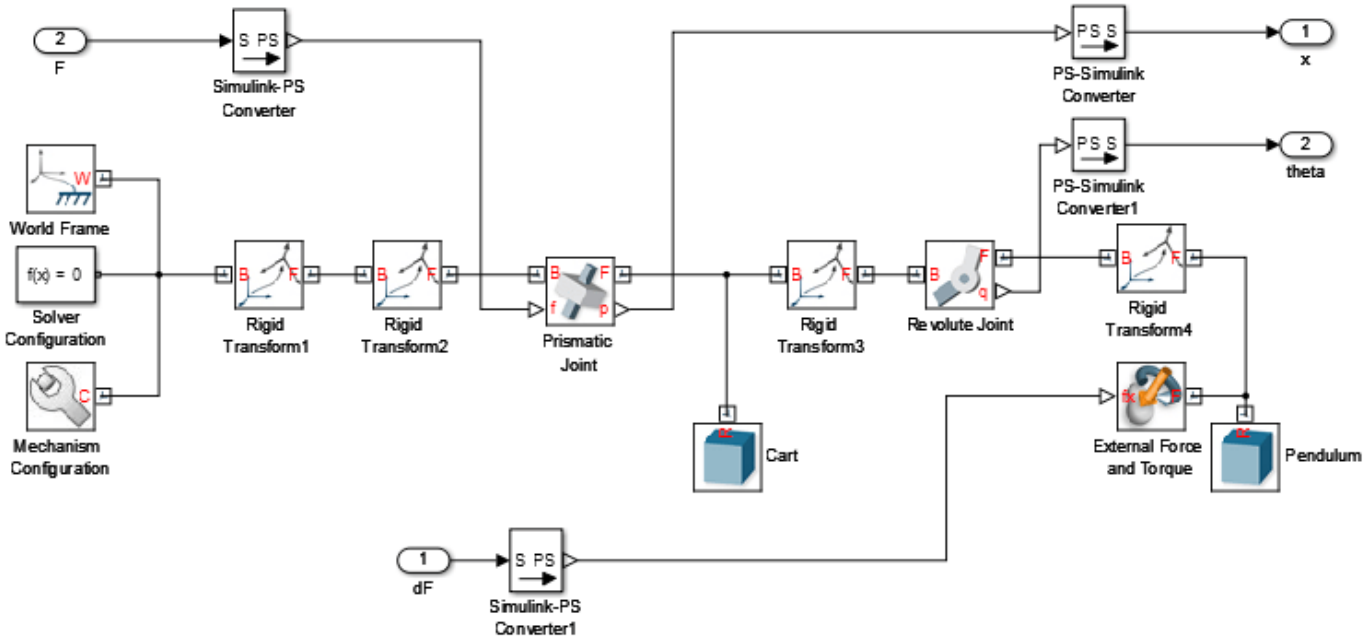


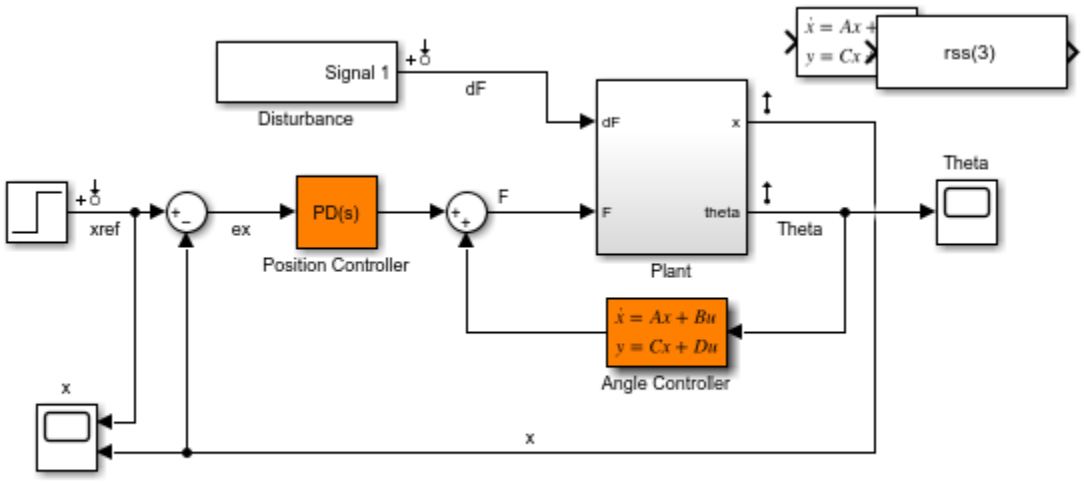
Figure 2: Simscape Multibody model

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);
```

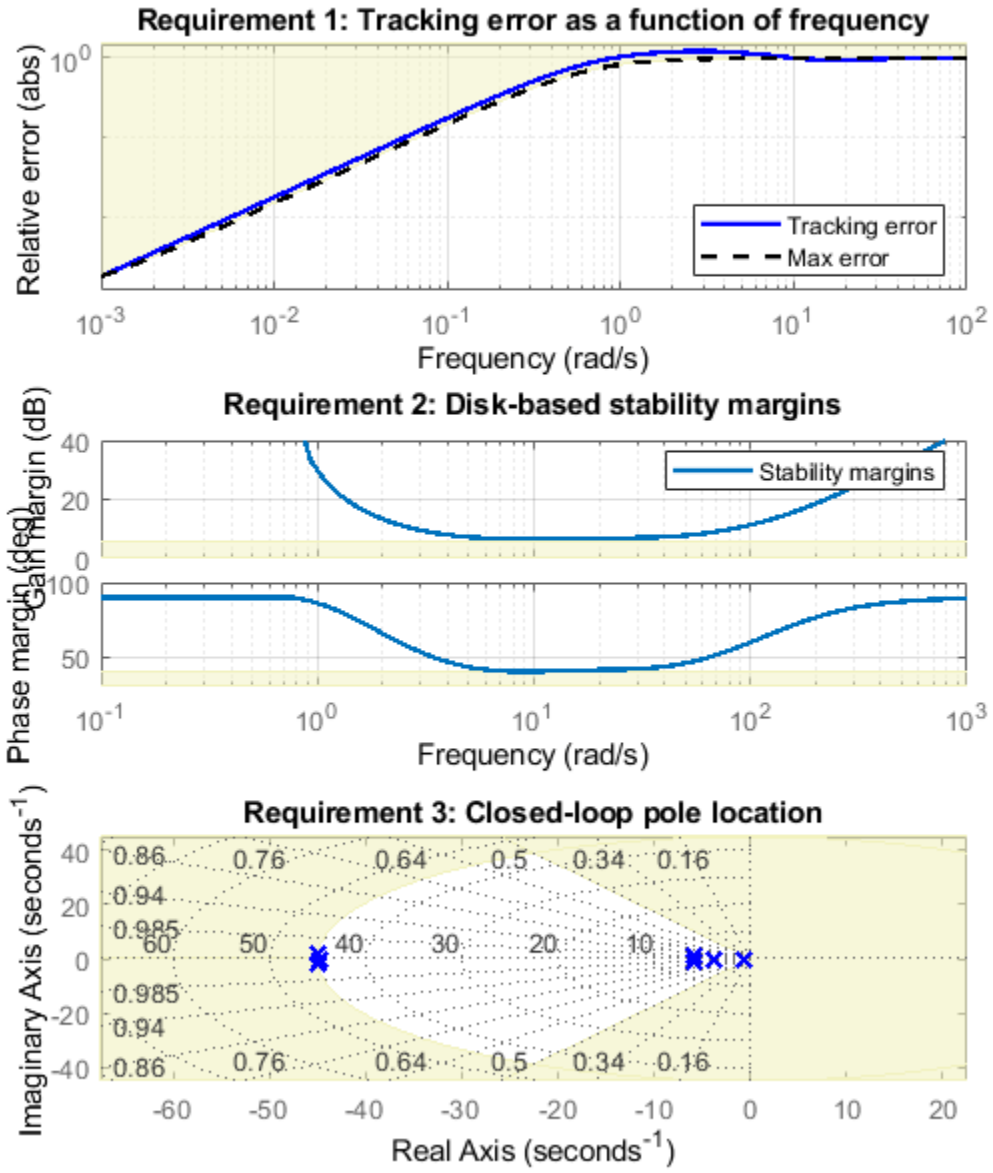
```
Final: Soft = 1.47, Hard = 0.97355, Iterations = 319
Final: Soft = 1.44, Hard = 0.999, Iterations = 243
Final: Soft = 1.27, Hard = 0.99815, Iterations = 294
Final: Soft = 1.36, Hard = 0.99693, Iterations = 322
Final: Soft = 1.27, Hard = 0.99803, Iterations = 307
Final: Soft = 1.36, Hard = 0.99898, Iterations = 262
```

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

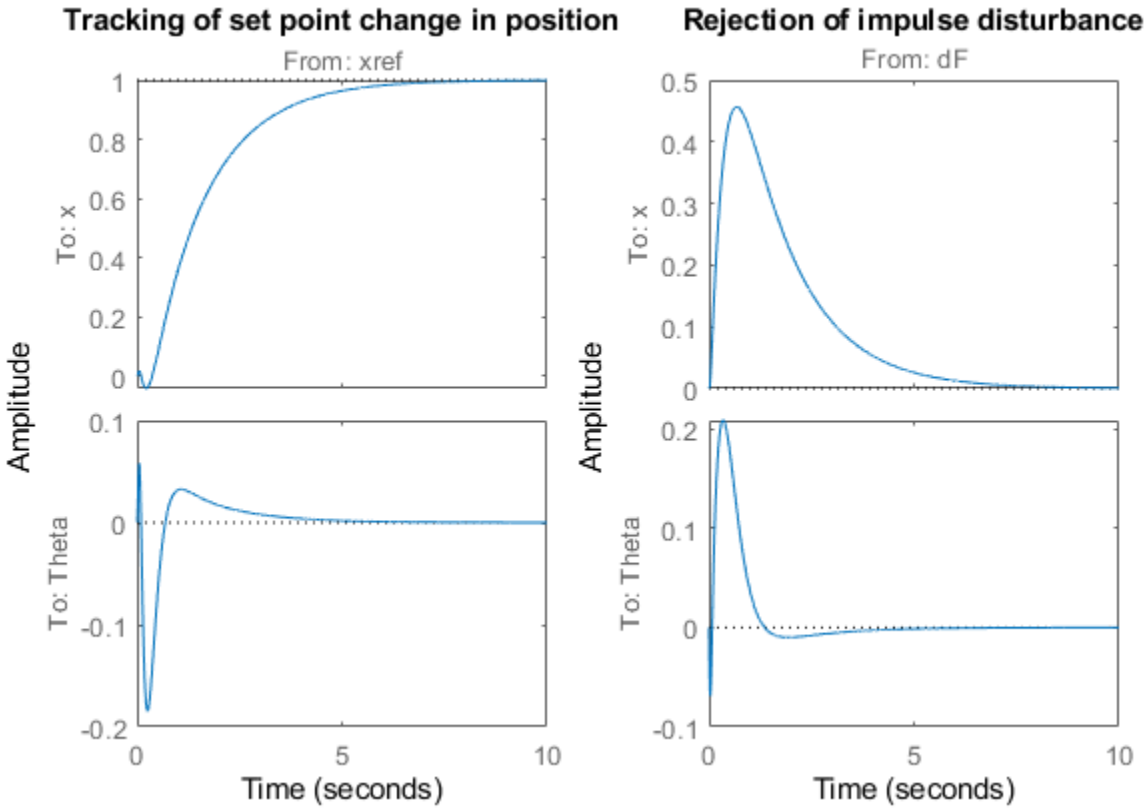
Use `viewGoal` to further analyze how the best design fares against each requirement.

```
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')
```

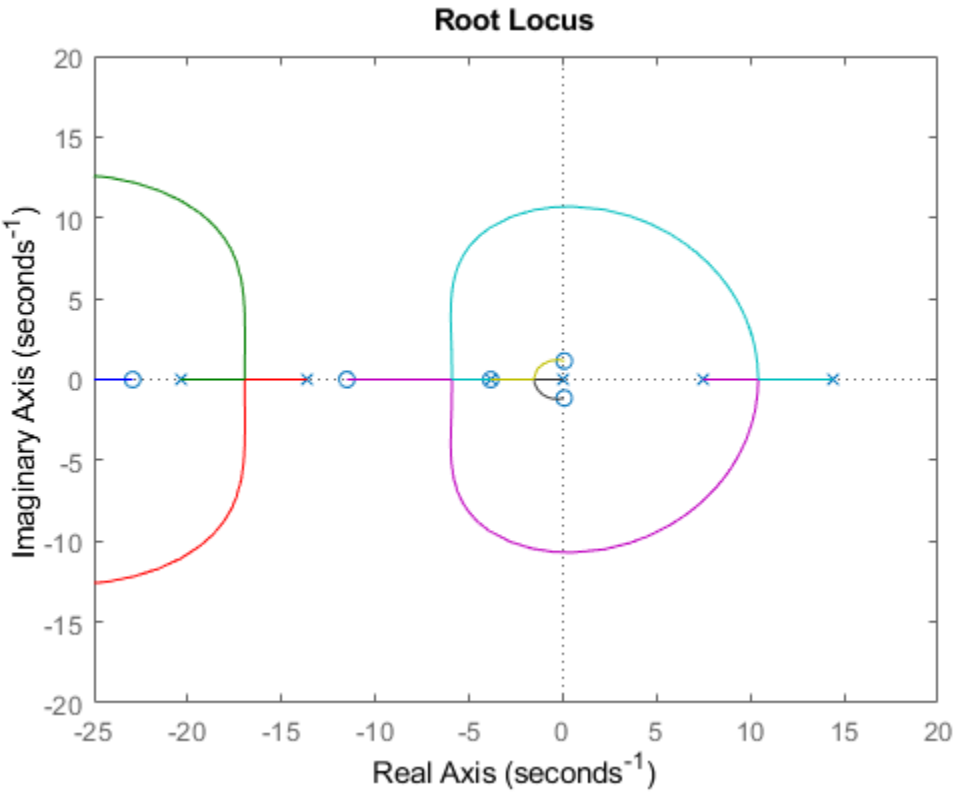
```
C1 =

                s
  Kp + Kd * --------
             Tf*s+1

  with Kp = 5.61, Kd = 1.63, Tf = 0.049

Name: Position_Controller
Continuous-time PDF controller in parallel form.
```

```
C2 = zpk(getBlockValue(ST,'Angle Controller'))
```

```
C2 =

  -1608.7 (s+13.26) (s+3.989)
  ---------------------------
      (s+134.8) (s-14.38)

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)
```

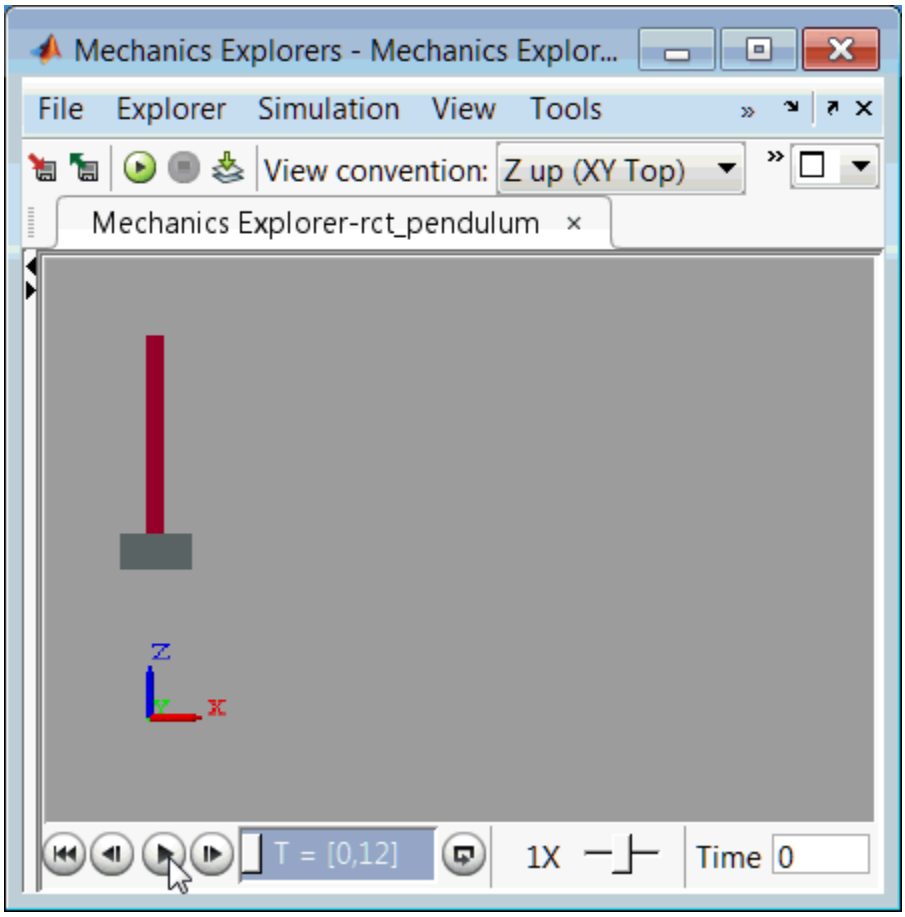


Figure 3: Cart/pendulum simulation with tuned controllers.

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” (Simulink Control Design)
- “Create and Configure slTuner Interface to Simulink Model” (Simulink Control Design)

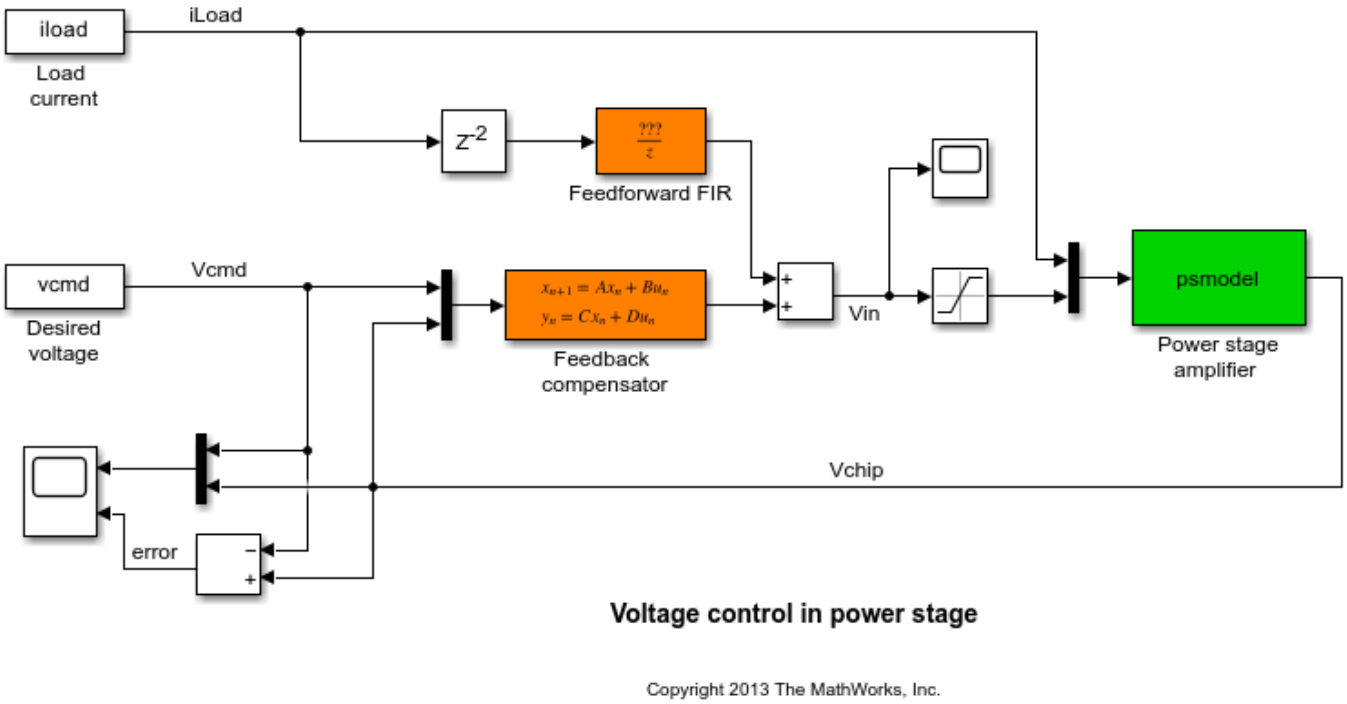
Digital Control of Power Stage Voltage

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:

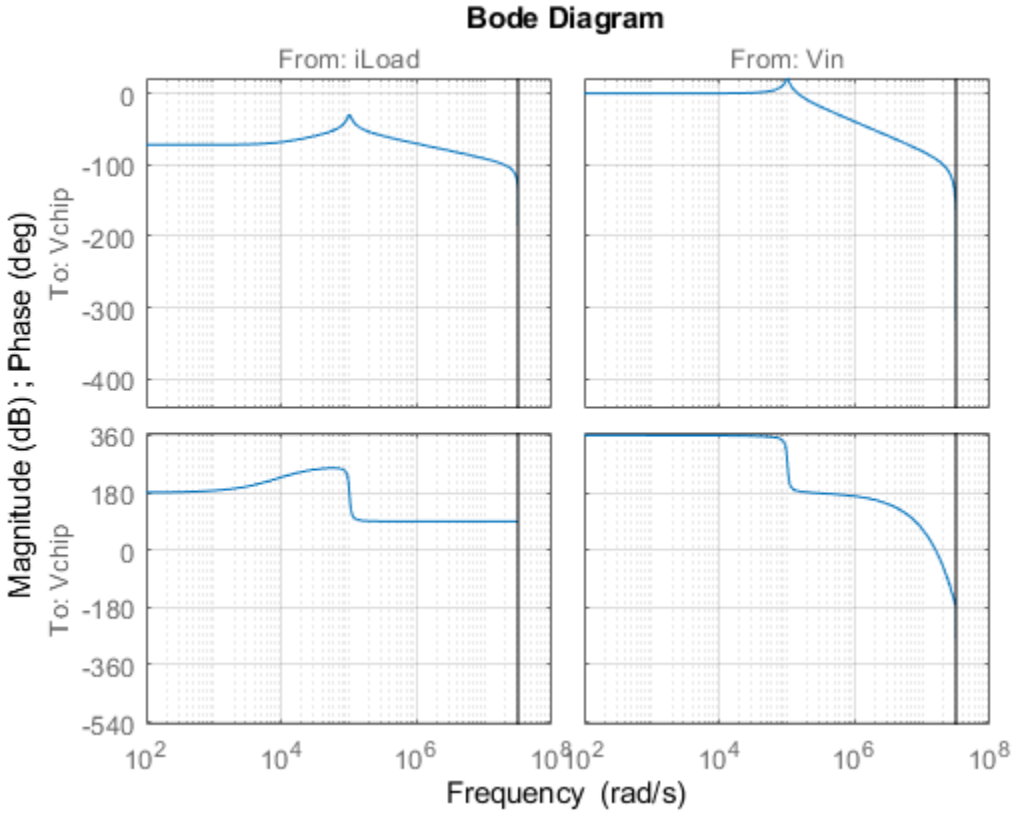
```
open_system('rct_powerstage')
```



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
```



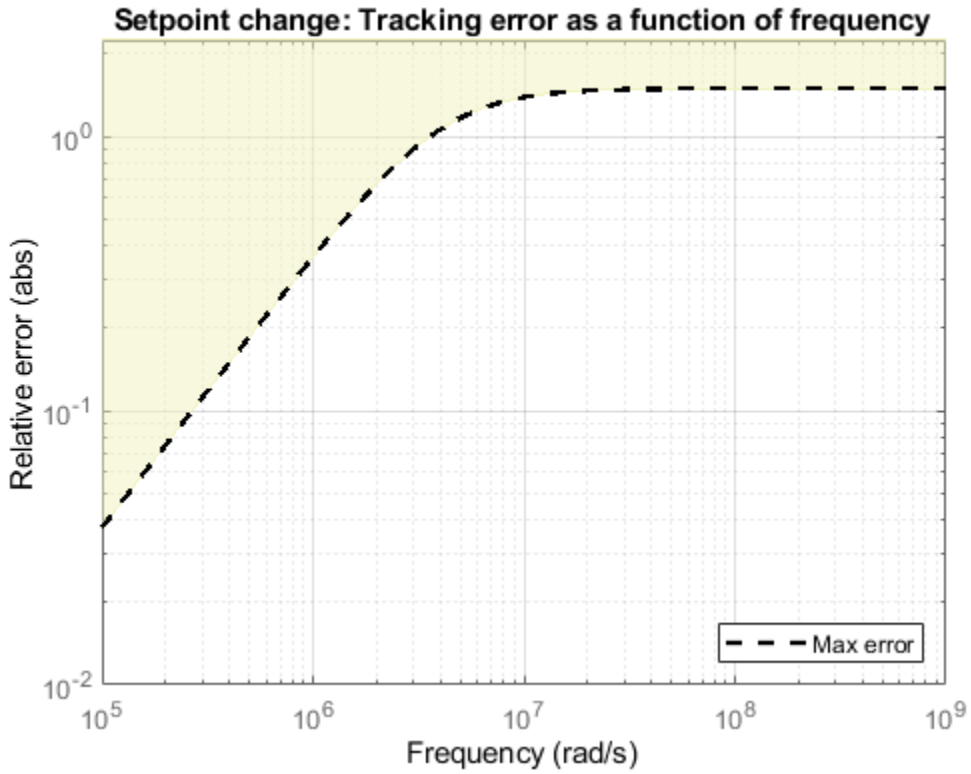
The controller must regulate the voltage `Vchip` delivered to the device to track the setpoint `Vcmd` and be insensitive to variations in load current `iLoad`. The control structure consists of a feedback compensator and a disturbance feedforward compensator. The voltage `Vin` going into the amplifier 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);
```

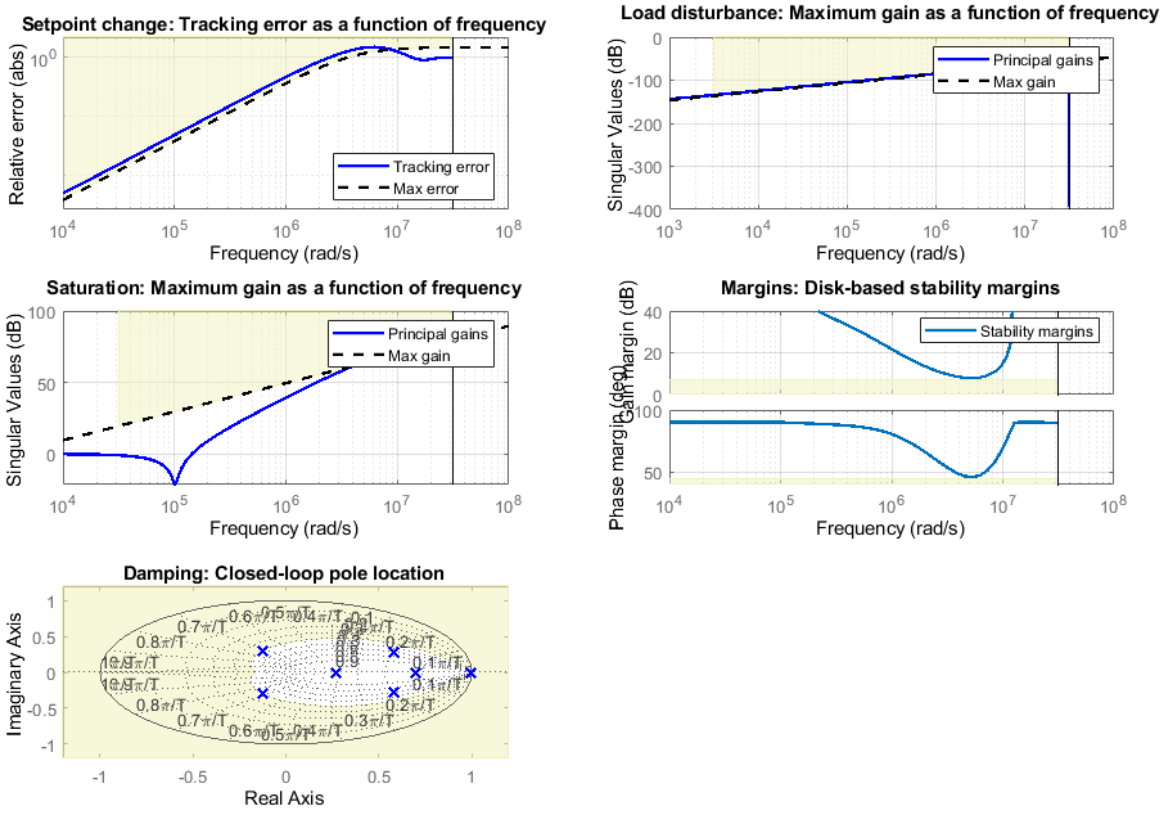
Note that `slTuner` automatically parameterizes the feedback compensator as a third-order state-space model (the order specified in the Simulink block). Next tune the feedforward and feedback compensators with `systune`. Treat the damping and margin requirements as hard constraints and try to best meet the remaining requirements.

```
rng(0)
topt = systuneOptions('RandomStart',6);
ST = systune(ST0,[Req1 Req2 Req3],[Req4 Req5],topt);
```

```
Final: Soft = 1.3, Hard = 0.89669, Iterations = 338
Final: Soft = 1.32, Hard = 0.94808, Iterations = 402
Final: Soft = 1.51, Hard = 0.91697, Iterations = 177
Final: Soft = 1.29, Hard = 0.9895, Iterations = 386
Final: Soft = 1.29, Hard = 0.99234, Iterations = 358
Final: Soft = 1.29, Hard = 0.92875, Iterations = 345
Final: Soft = 1.28, Hard = 0.99305, Iterations = 380
```

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.

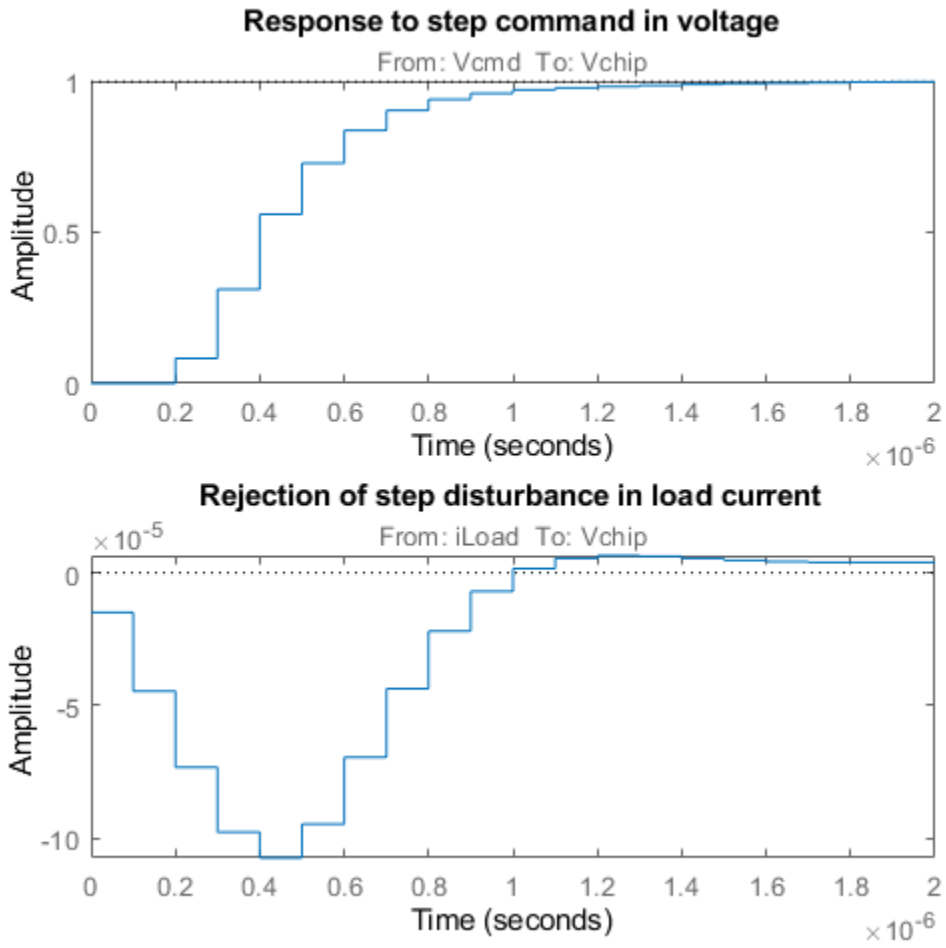
```
figure('Position',[10,10,1071,714])
viewGoal([Req1 Req2 Req3 Req4 Req5],ST)
```



Validation

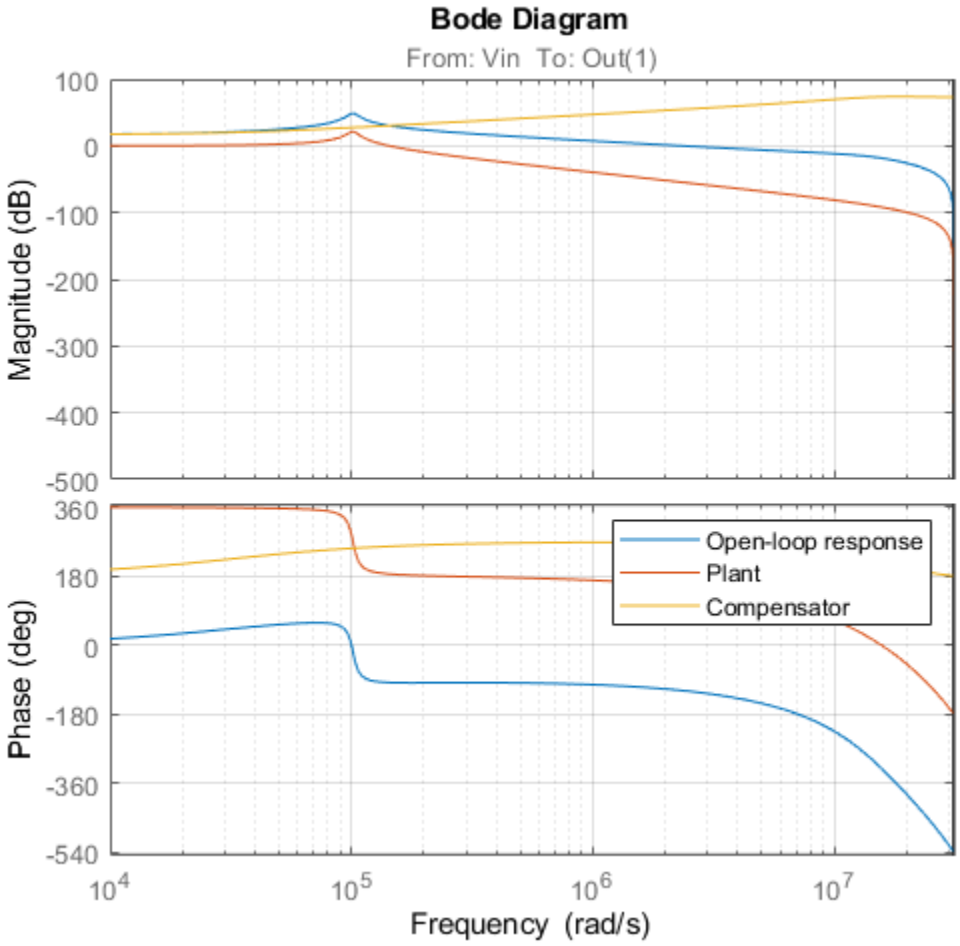
First validate the design in the linear domain using the `slTuner` interface. Plot the closed-loop response to a step command `Vcmd` and a step disturbance `iLoad`.

```
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.

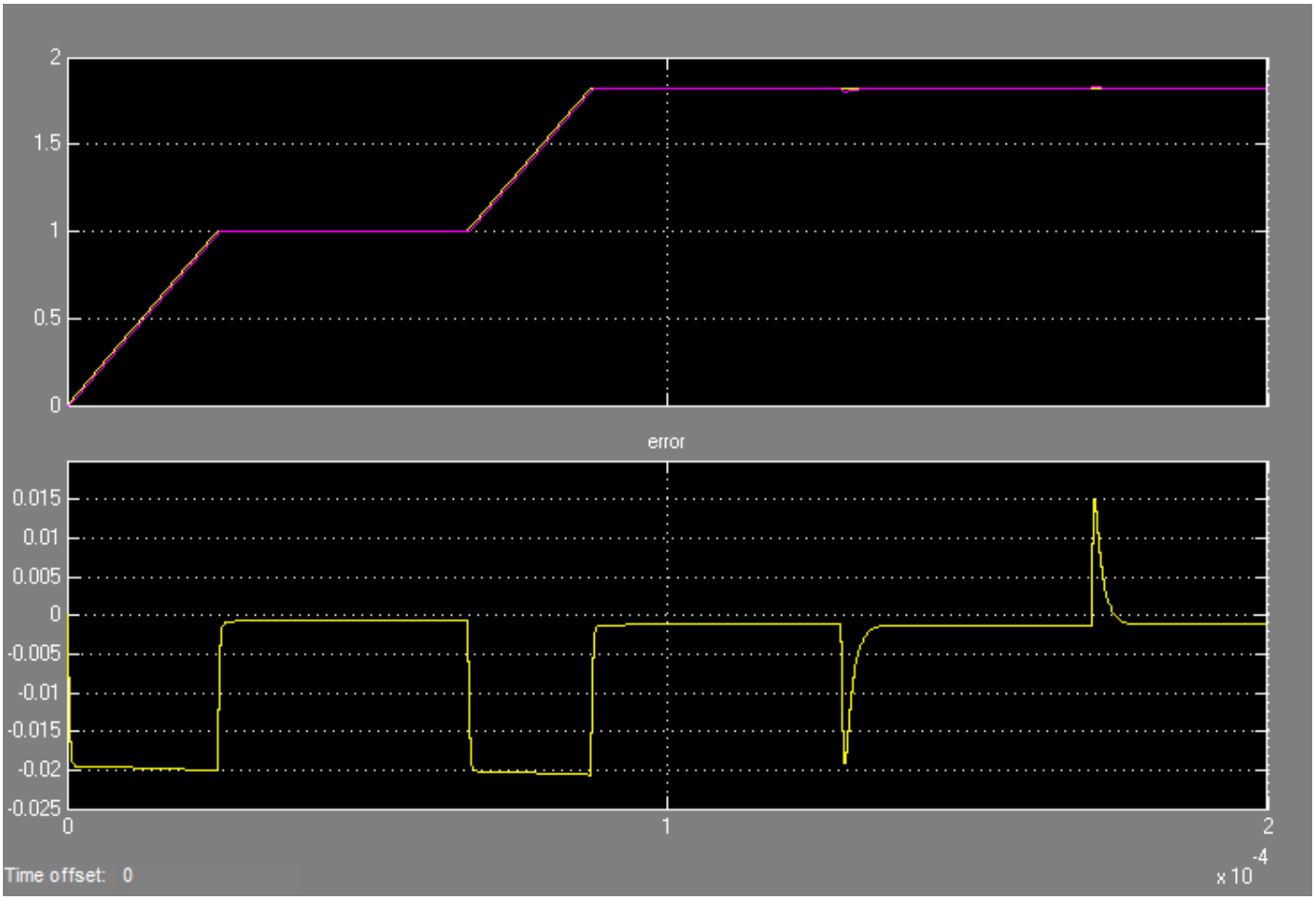


Figure 1: Response to ramp command and step load disturbances.

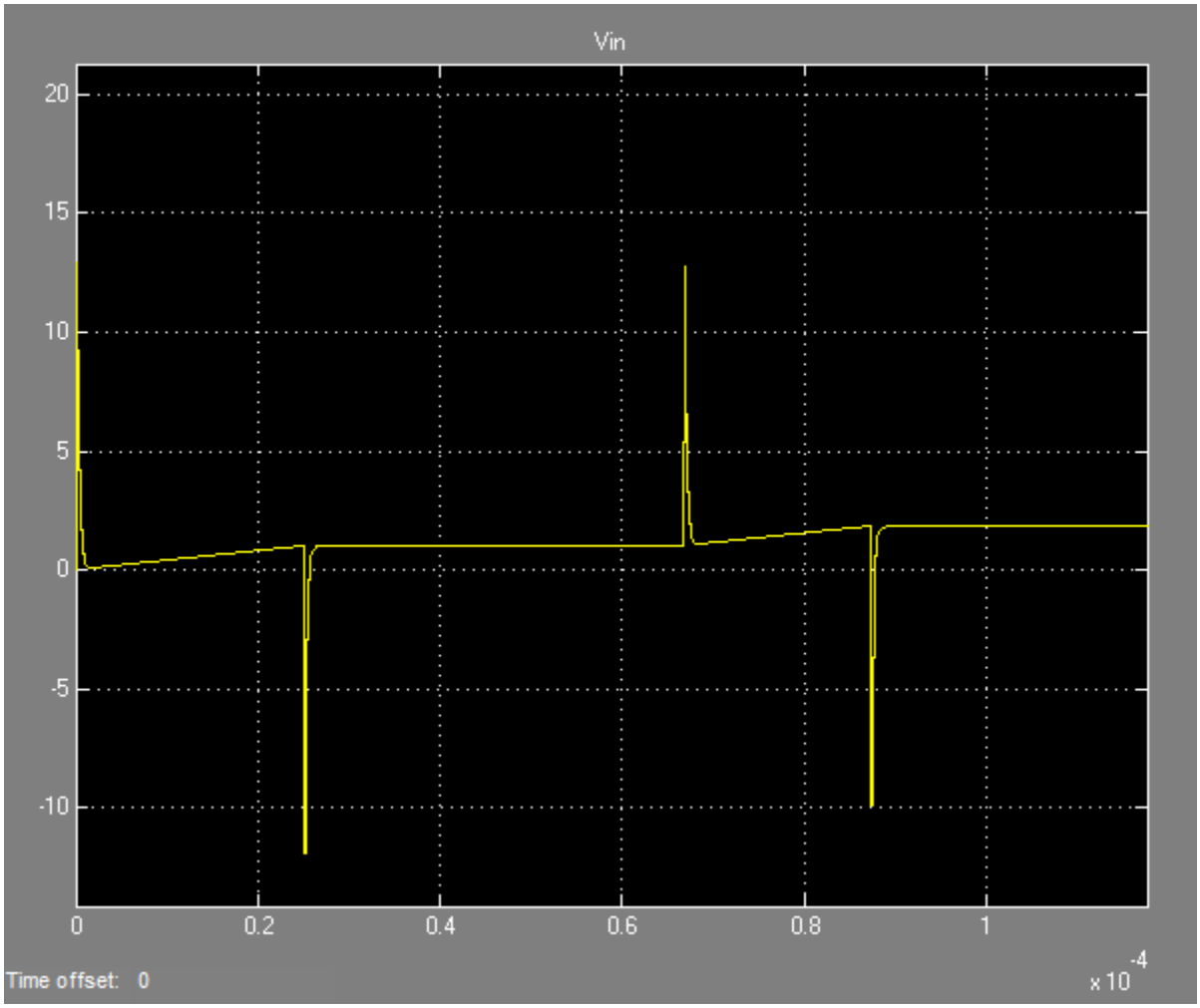


Figure 2: Amplitude of input voltage `Vin` during setpoint tracking phase.

See Also

`systune (slTuner)` | `slTuner` | `TuningGoal.Tracking` | `TuningGoal.Gain` | `TuningGoal.Margins`

Related Examples

- “MIMO Control of Diesel Engine” on page 18-141

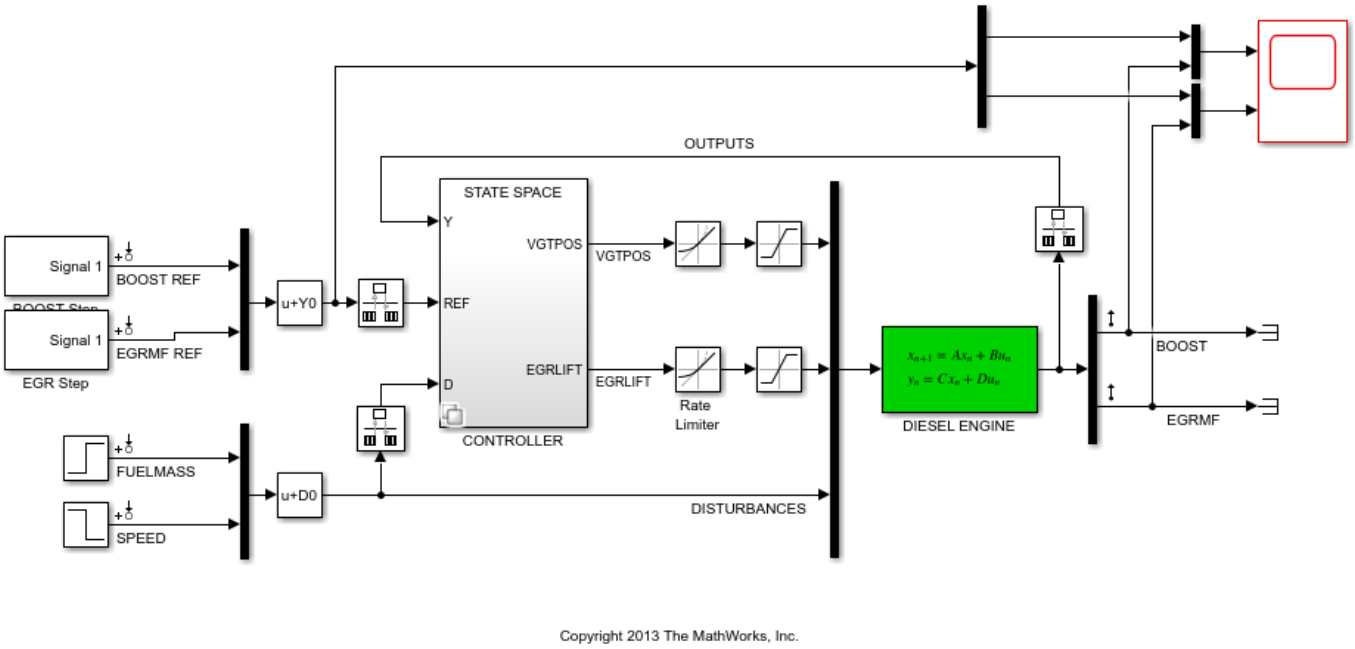
MIMO Control of Diesel Engine

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.

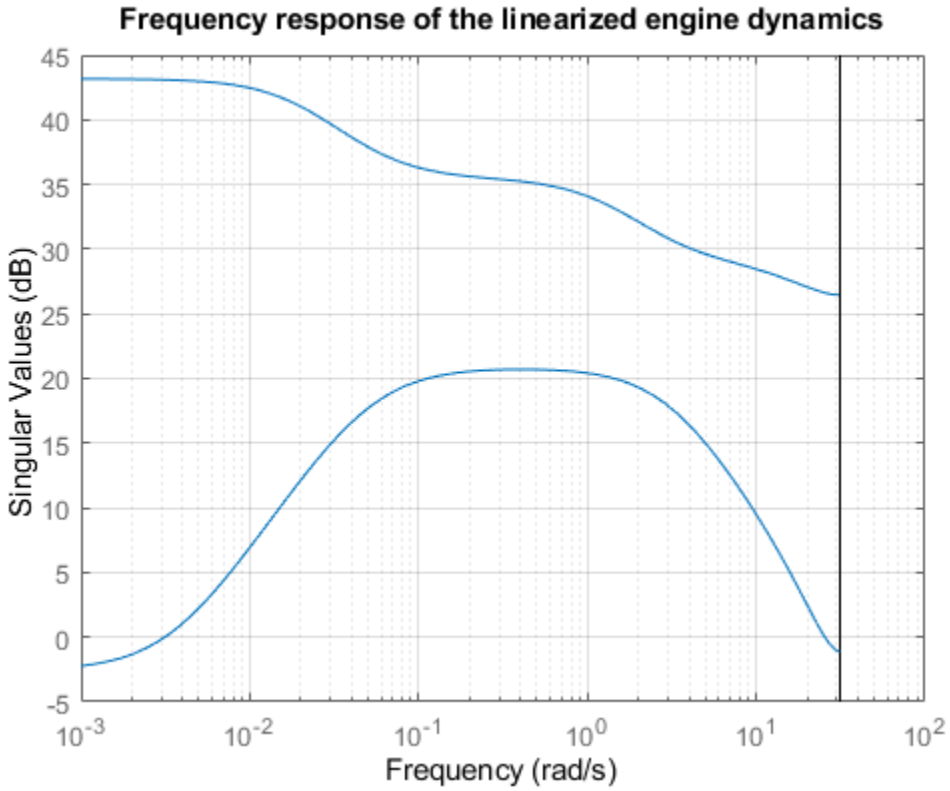
```
open_system('rct_diesel')
```



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];
```

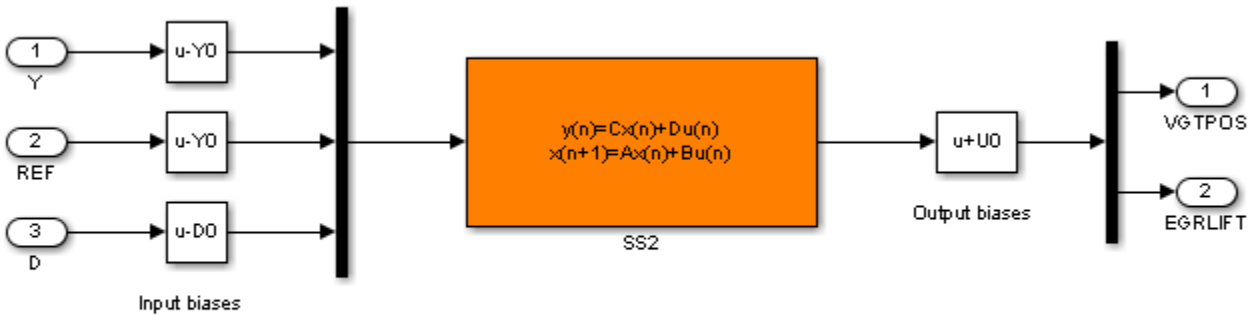
To provide adequate robustness to unmodeled dynamics and aliasing, limit the control bandwidth and impose sufficient stability margins at both the plant inputs and outputs. Because we are dealing with

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” on page 14-161 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'})
```

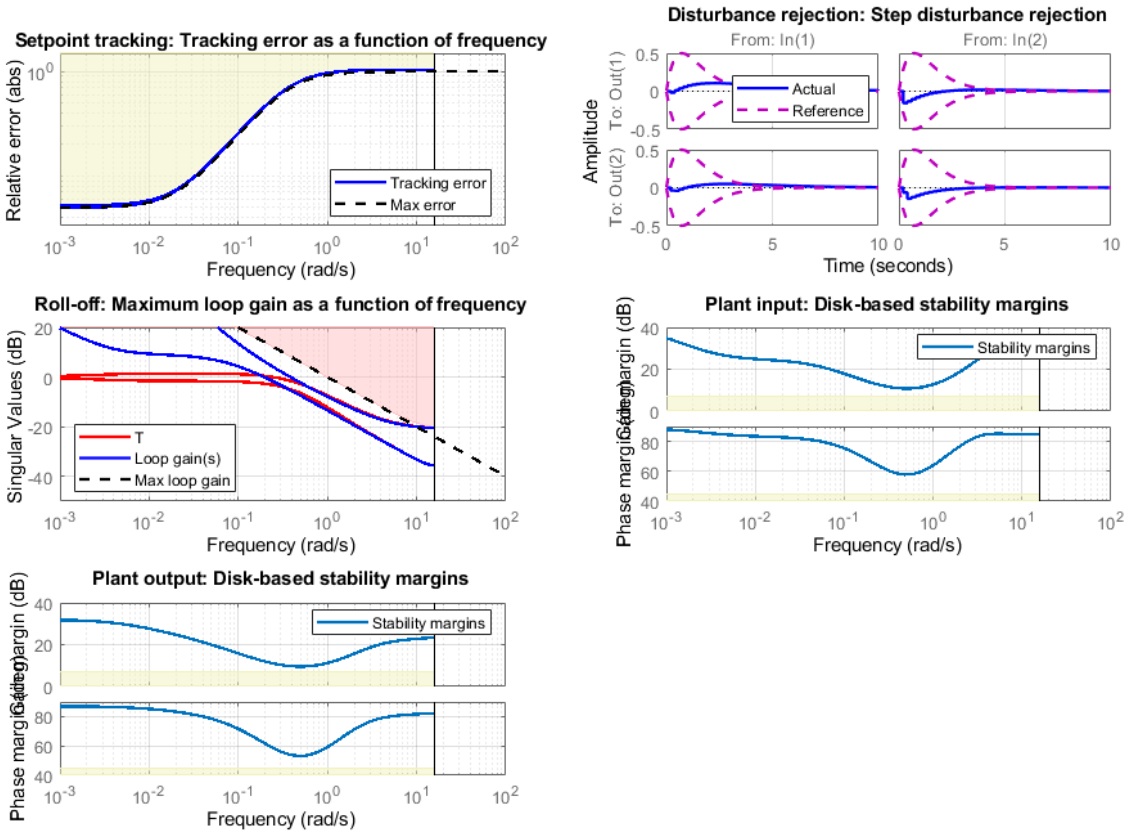
Now use `systune` to tune the state-space controller subject to our control objectives. Treat the stability margins and roll-off target as hard constraints and try to best meet the remaining objectives (soft goals). Randomize the starting point to reduce exposure to undesirable local minima.

```
Opt = systuneOptions('RandomStart',2);
rng(0), ST1 = systune(ST0,[TR DR],[M1 M2 RO],Opt);
```

```
Final: Soft = 1.28, Hard = 0.88766, Iterations = 396
Final: Soft = 1.05, Hard = 0.94955, Iterations = 506
Final: Soft = 1.05, Hard = 0.99312, Iterations = 437
```

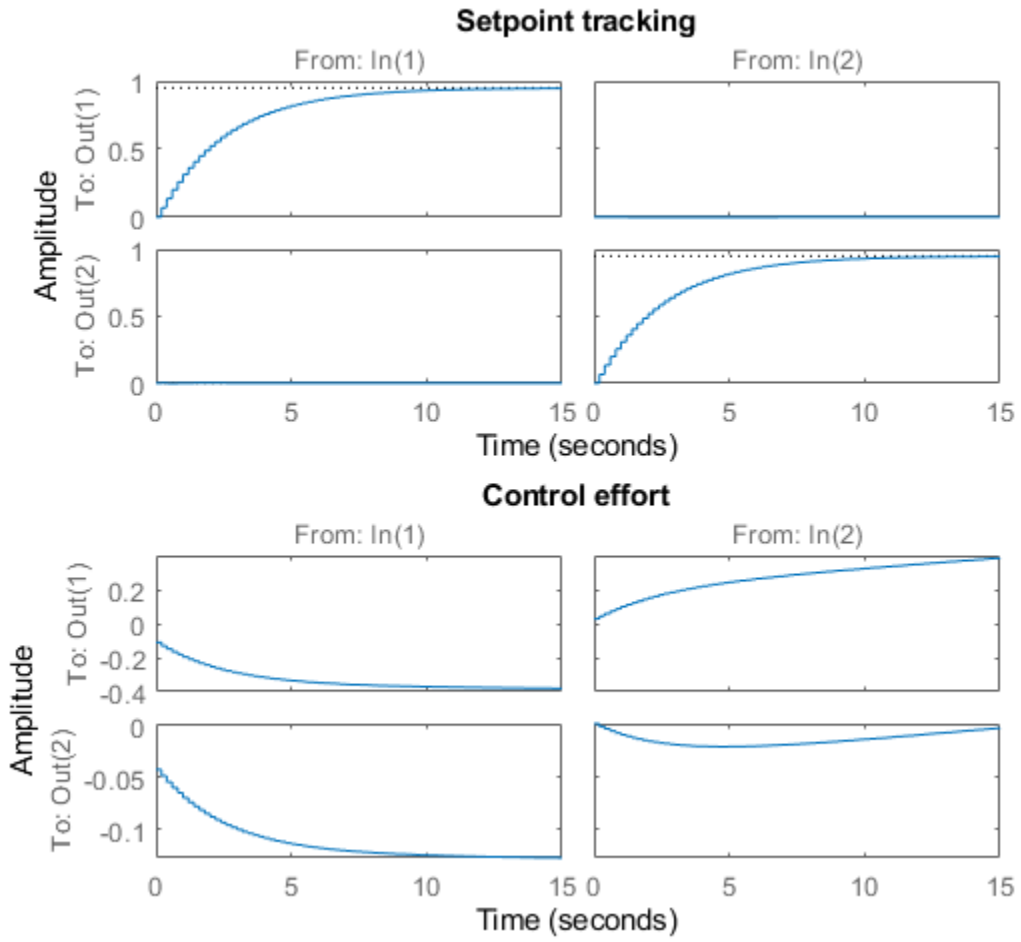
All requirements are nearly met (a requirement is satisfied when its normalized value is less than 1). Verify this graphically.

```
figure('Position',[10,10,1071,714])
viewGoal([TR DR RO M1 M2],ST1)
```

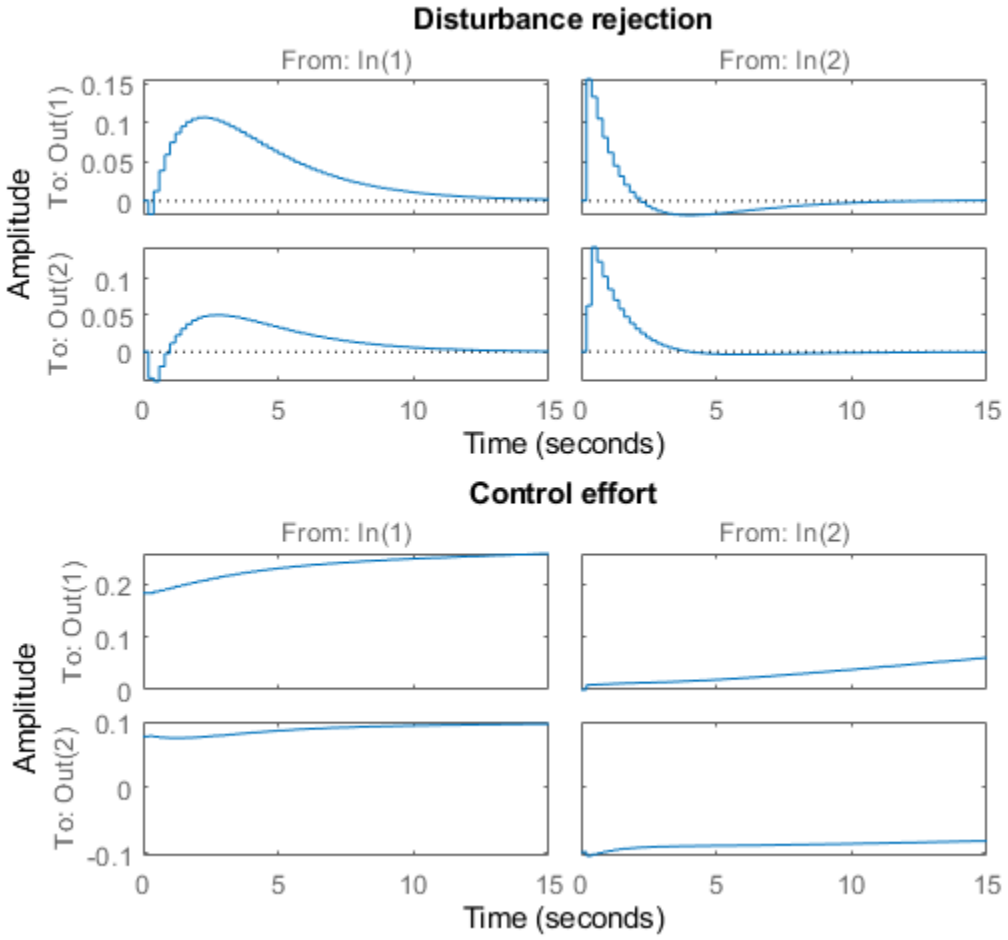


Plot the setpoint tracking and disturbance rejection responses. Scale by the signal amplitudes to show normalized effects (boost pressure changes by +10 KPa, EGR massflow by +3 g/s, fuel mass by +5 mg, and speed by -200 rpm).

```
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')
```

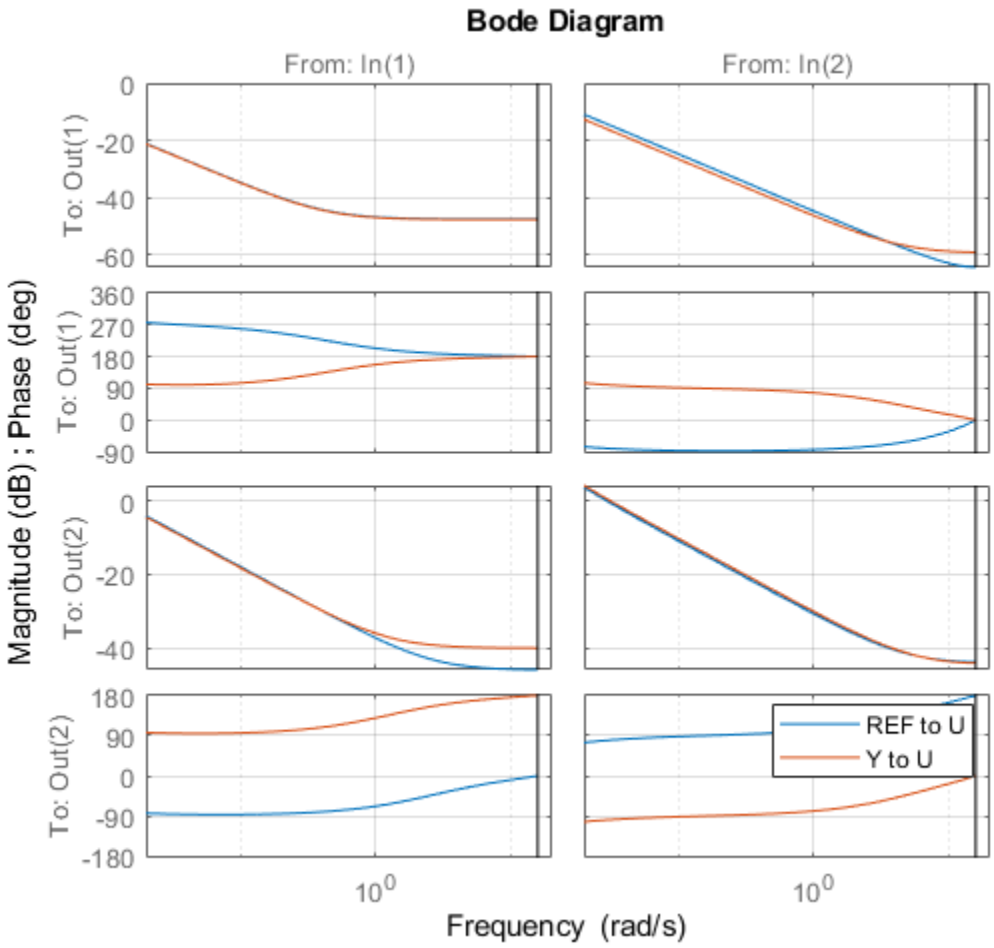


The controller responds in less than 5 seconds with minimum cross-coupling between the BOOST and EGRMF variables.

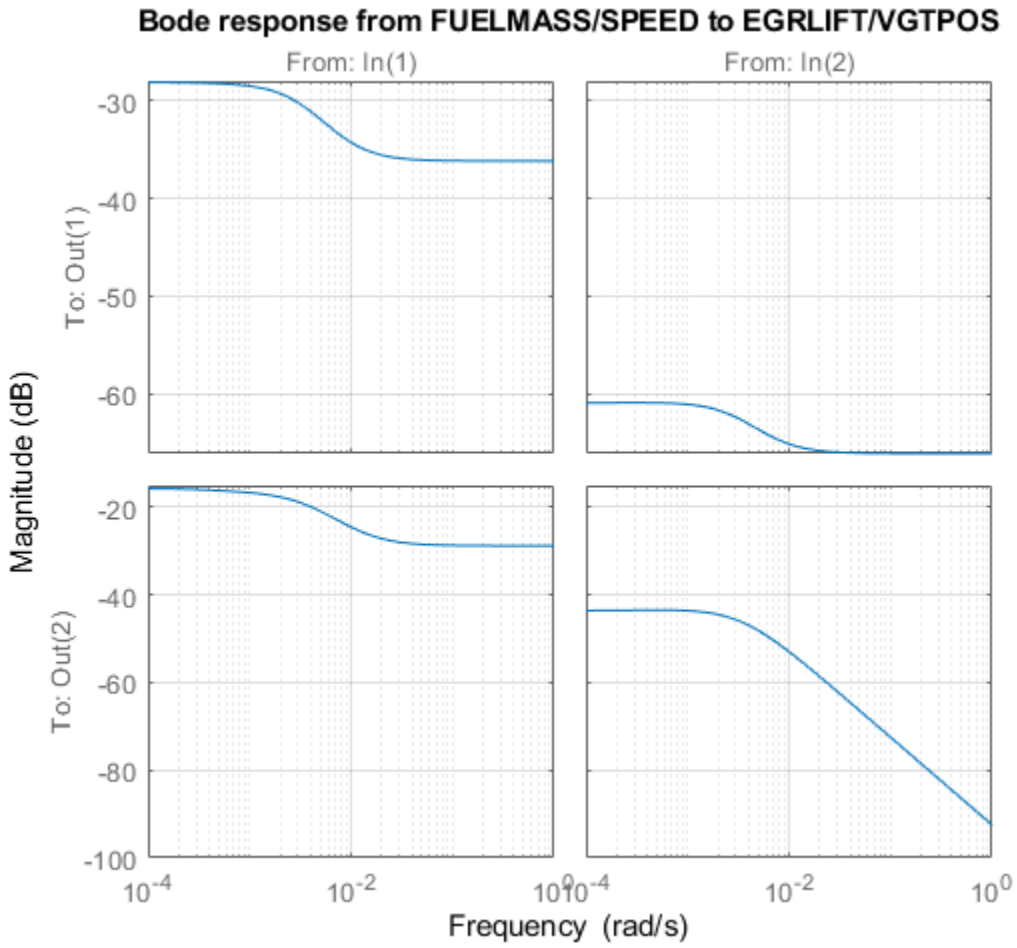
Tuning of Simplified Control Structure

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

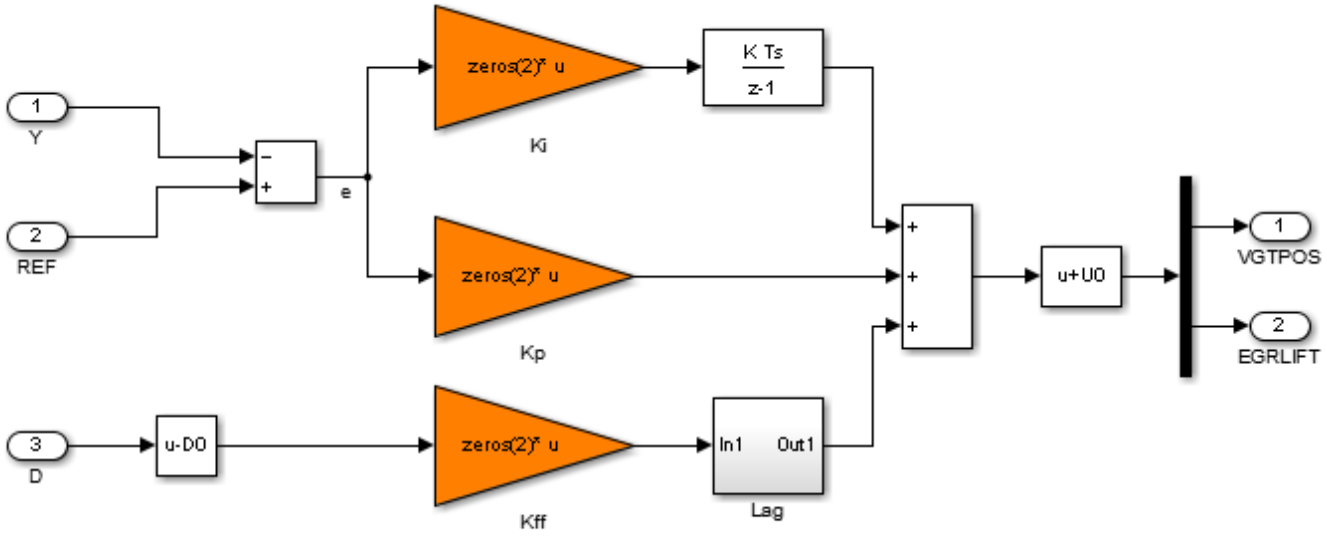
```
C = getBlockValue(ST1,'SS2');

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.

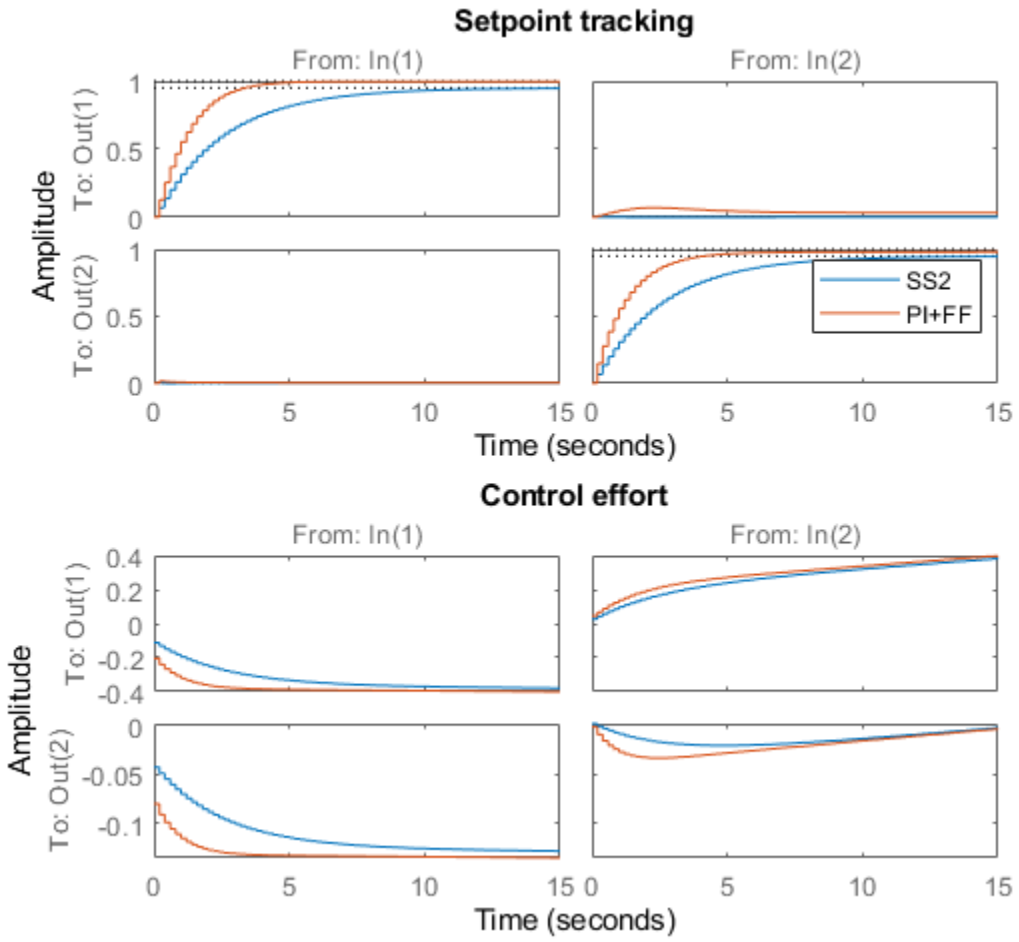
Using variant subsystems, you can implement both control structures in the same Simulink model and use a variable to switch between them. Here setting `MODE=2` selects the MIMO PI structure. As before, use `systune` to tune the three 2-by-2 gain matrices `Kp`, `Ki`, `Kff` in the simplified control structure.

```
% 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]);
```

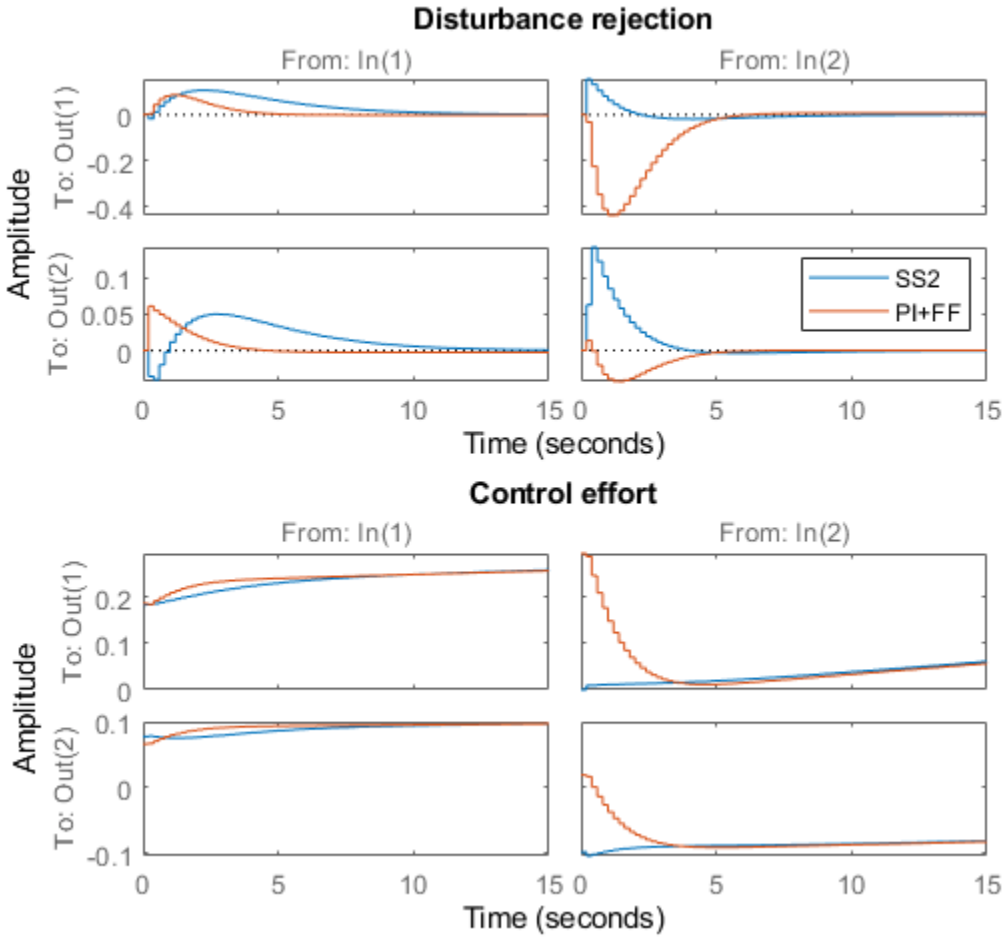
```
Final: Soft = 1.09, Hard = 0.99961, Iterations = 302
```

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    -0.00798  -0.0005209
   y2    -0.02047     0.01545

Name: Kp
Static gain.

-----------------------------------

Block 2: rct_diesel/CONTROLLER/MIMO PID/Ki =

  D =
            u1        u2
```

```
   y1  -0.01051   -0.0143
   y2  -0.03026   0.04698

Name: Ki
Static gain.

-----------------------------------

Block 3: rct_diesel/CONTROLLER/MIMO PID/Kff =

  D =
              u1          u2
   y1     0.01322  -9.457e-05
   y2     0.03708   -0.001465

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).

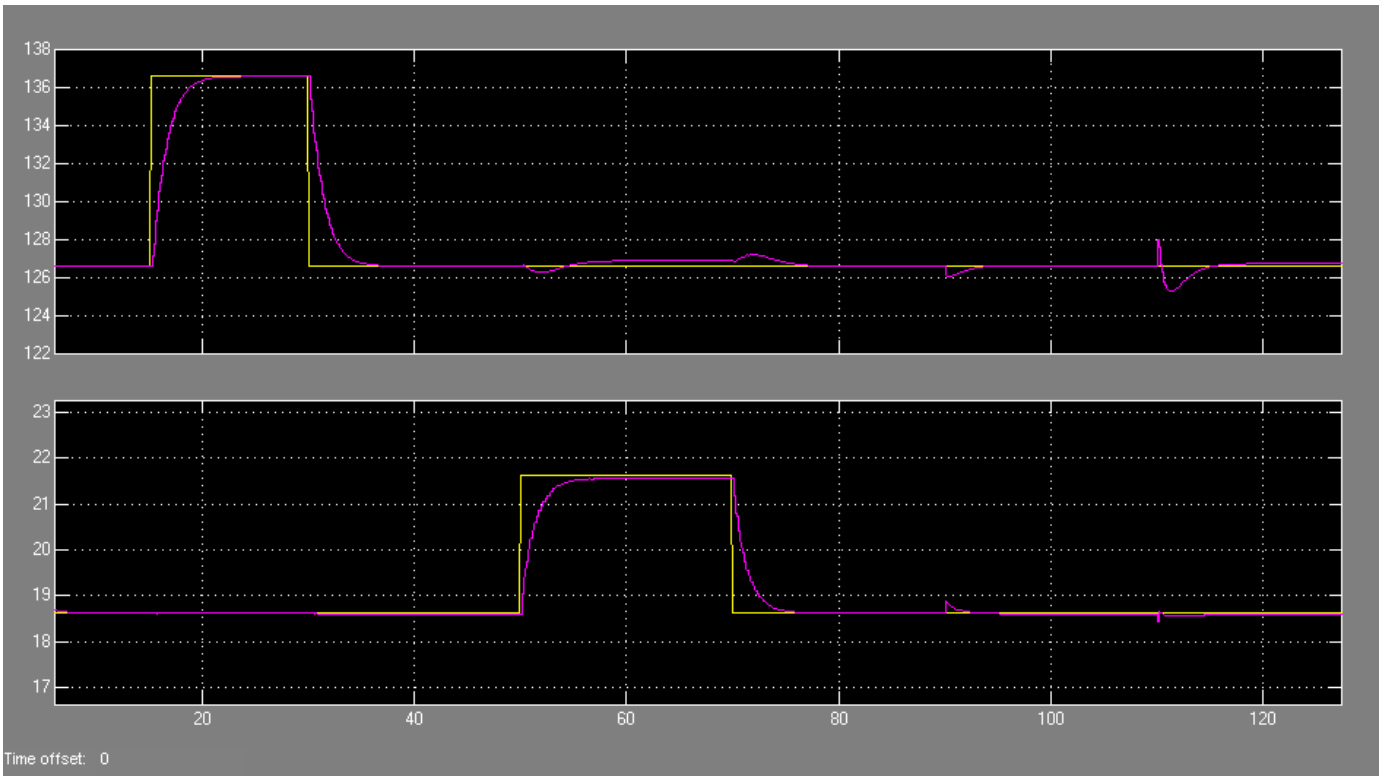


Figure 3: Simulation results with simplified controller.

See Also

`systune (slTuner)` | `slTuner` | `TuningGoal.Tracking` | `TuningGoal.StepRejection` | `TuningGoal.MaxLoopGain` | `TuningGoal.Margins`

Related Examples

- “Digital Control of Power Stage Voltage” on page 18-132

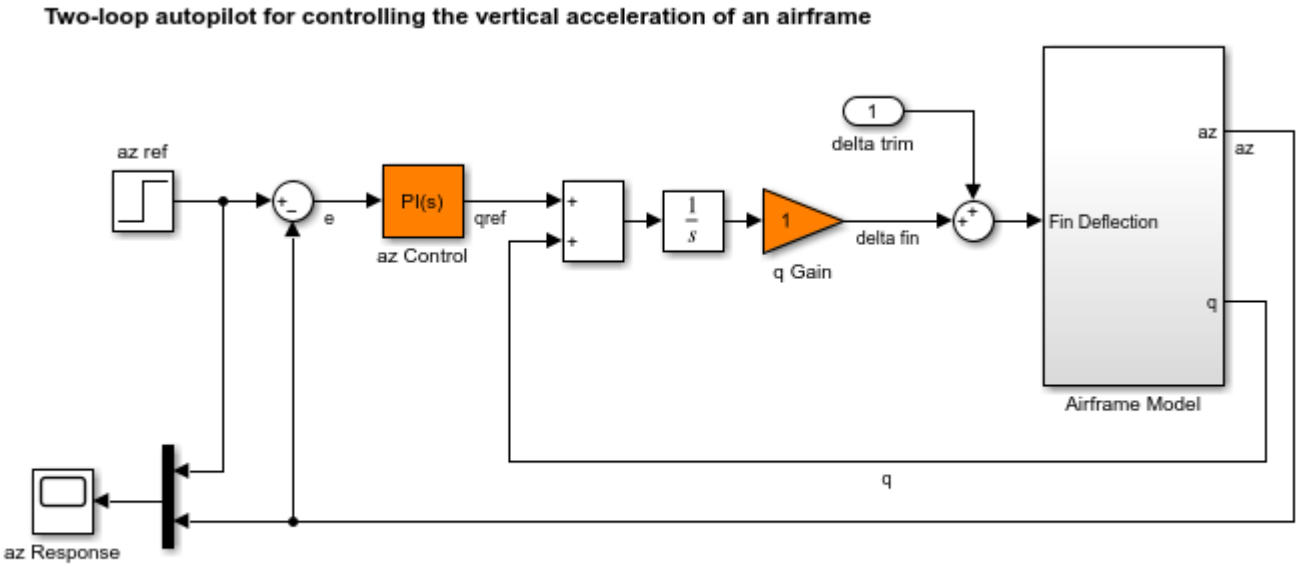
Tuning of a Two-Loop Autopilot

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.

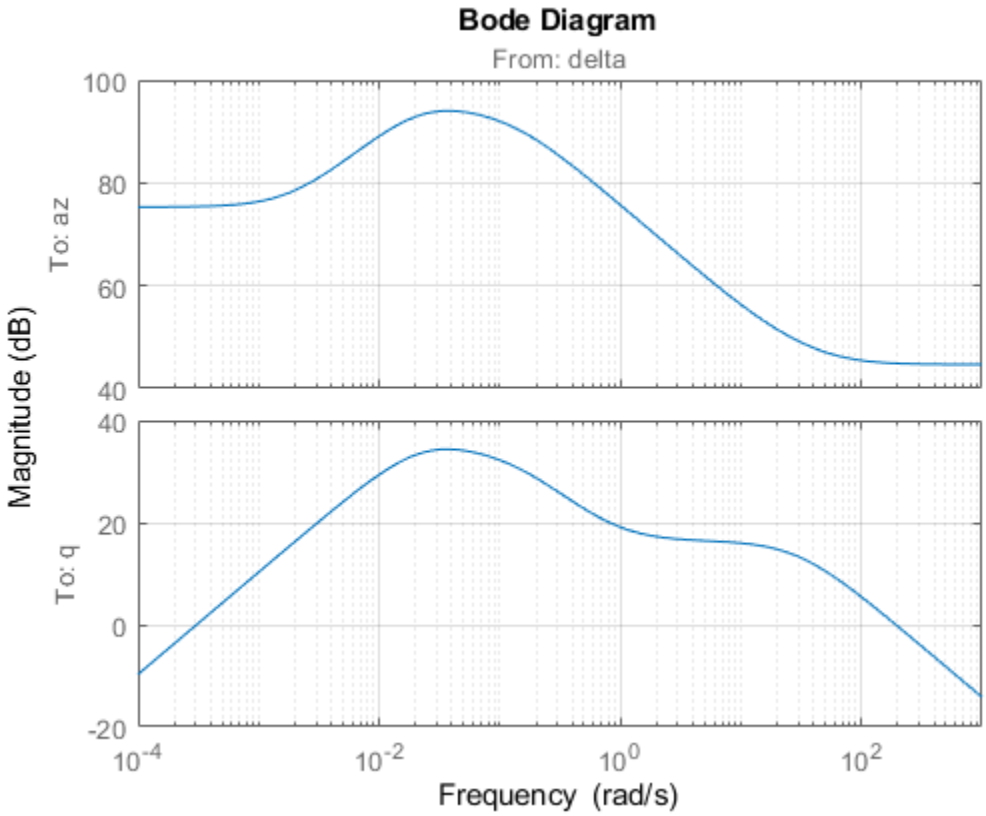
To analyze the airframe dynamics, trim the airframe for $\alpha = 0$ and $V = 984m/s$. The trim condition corresponds to zero normal acceleration and pitching moment (w and q steady). Use `findop` to compute the corresponding closed-loop operating condition. Note that we added a "delta trim" input port 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:
----------
    Min          x          Max        dxMin        dx         dxMax
__________ __________ __________ __________ __________ __________

(1.) rct_airframe1/Airframe Model/Aerodynamics & Equations of Motion/ Equations of Motion (Body A
     0          0          0         -Inf         984         Inf
-3047.9999  -3047.9999  -3047.9999     -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       22.6897       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:
----------
   Min         u         Max
_________ _________ _________

(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'});
```

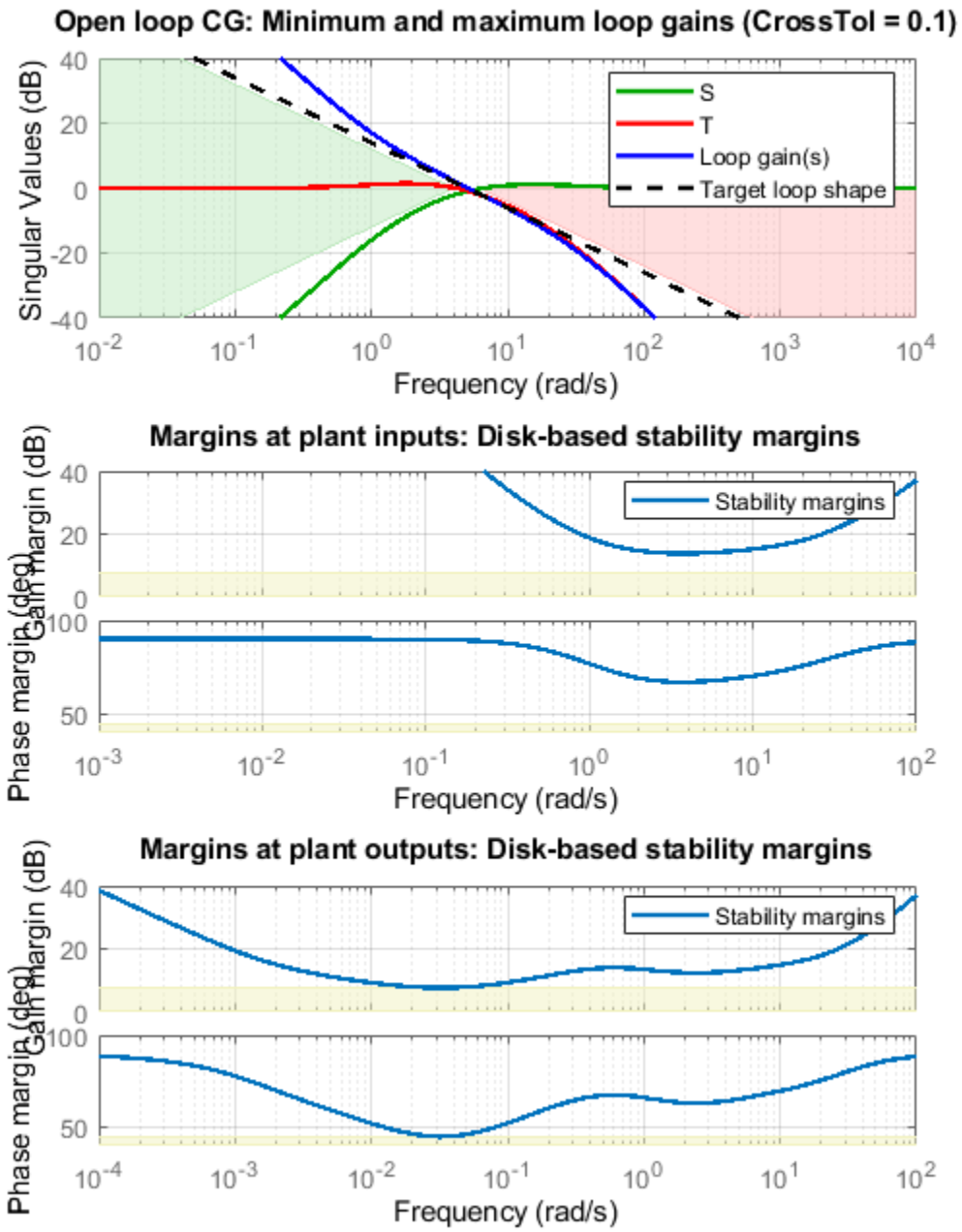
Finally, tune the control system parameters to meet the 1 second response time requirement. In the frequency domain, this roughly corresponds to a gain crossover frequency `wc = 5` rad/s for the open-loop response at the plant input "delta fin".

```
wc = 5;
Controls = 'delta fin';
Measurements = {'az','q'};
[ST,gam,Info] = looptune(ST0,Controls,Measurements,wc);
```

```
Final: Peak gain = 1.01, Iterations = 71
```

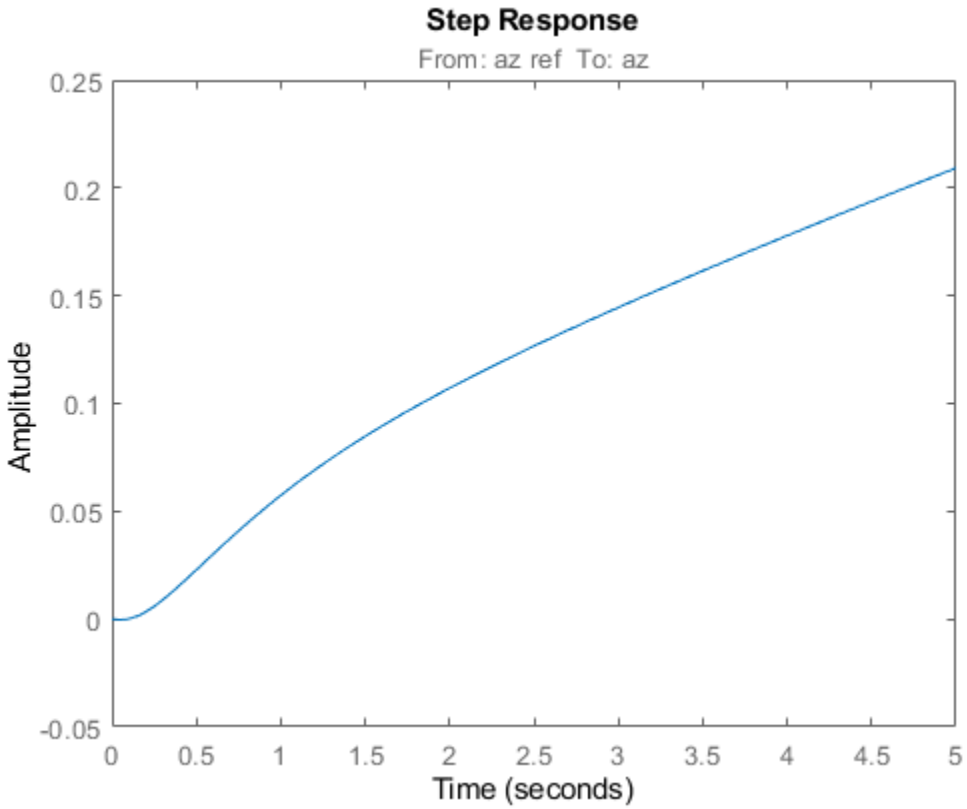
The requirements are normalized so a final value near 1 means that all requirements are met. Confirm this by graphically validating the design.

```
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

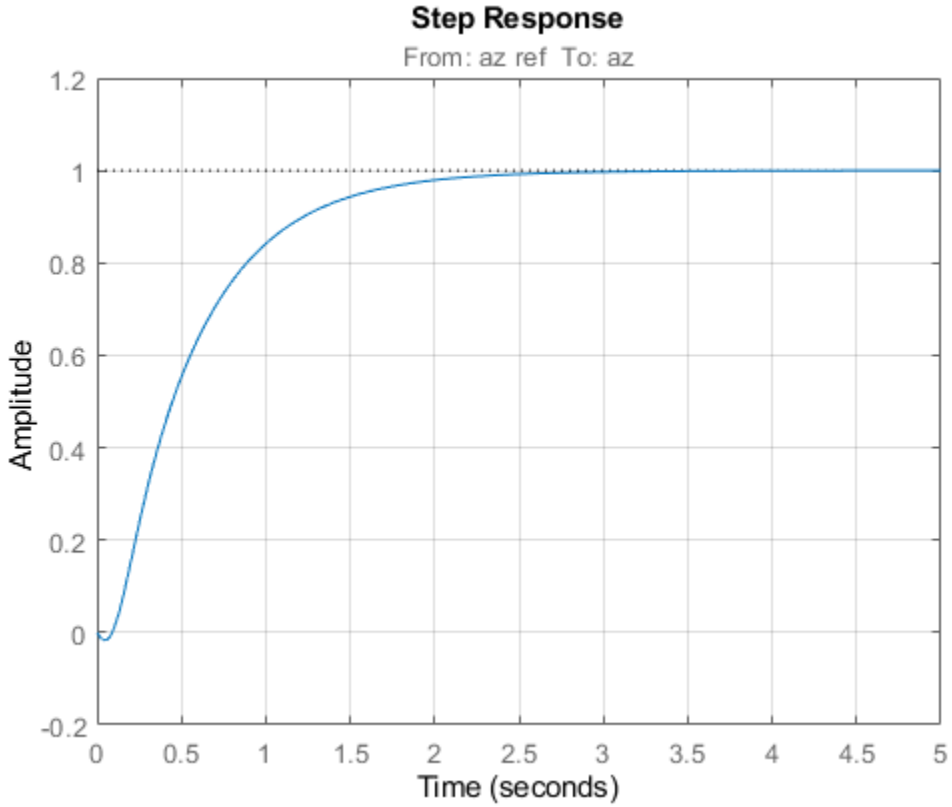
To remedy this issue, add an explicit requirement that `az` should follow the step command `azref` with a 1 second response time. Also relax the gain crossover requirement to the interval [3,12] to let the tuner find the appropriate gain crossover frequency.

```
TrackReq = TuningGoal.Tracking('az ref','az',1);
ST = looptune(ST0,Controls,Measurements,[3,12],TrackReq);
```

```
Final: Peak gain = 1.23, Iterations = 54
```

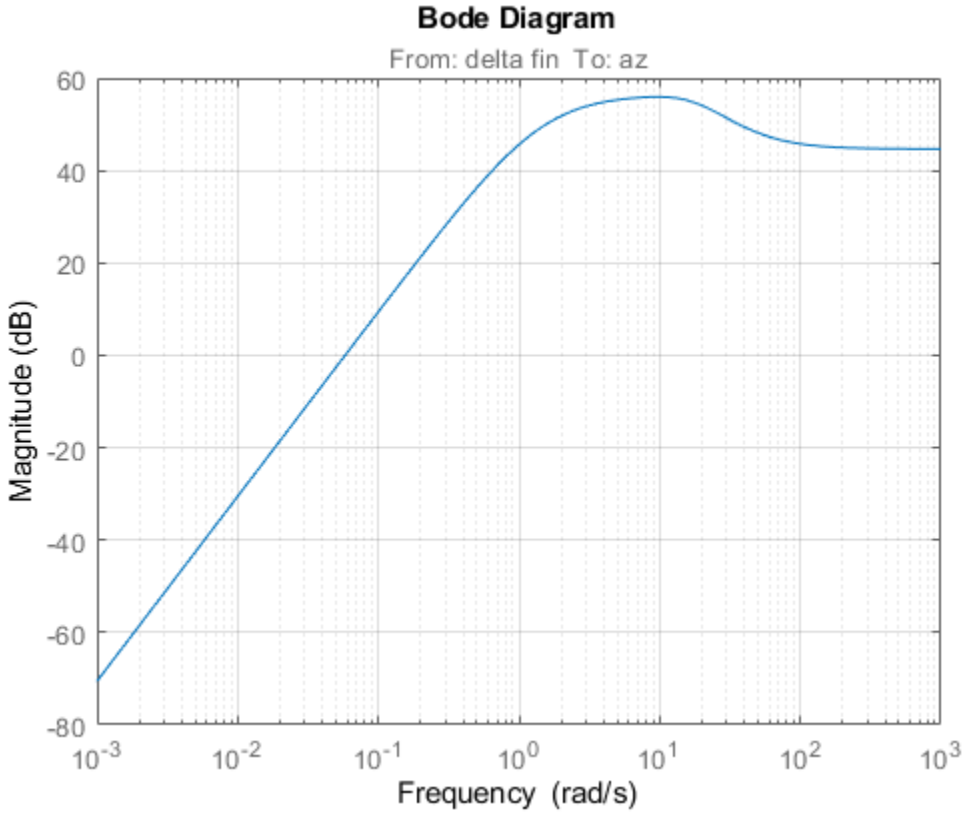
The step response from `azref` to `az` is now satisfactory:

```
Tr1 = getIOTransfer(ST,'az ref','az');
step(Tr1,5)
grid
```

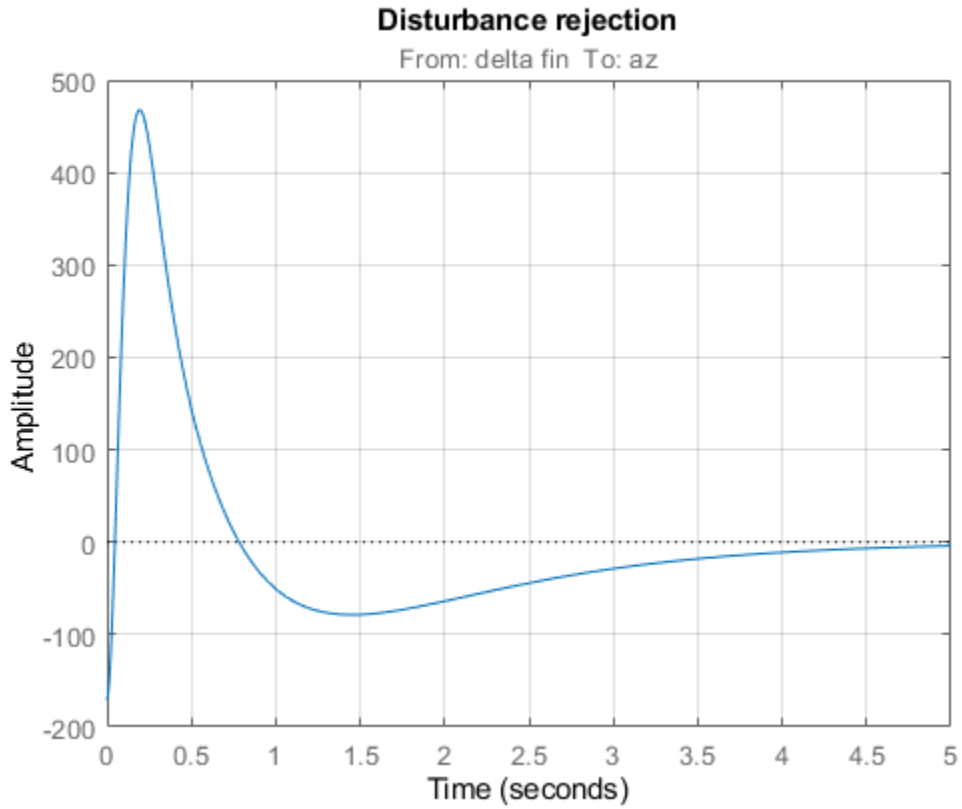


Also check the disturbance rejection characteristics by looking at the responses from a disturbance entering at the plant input

```
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.
```

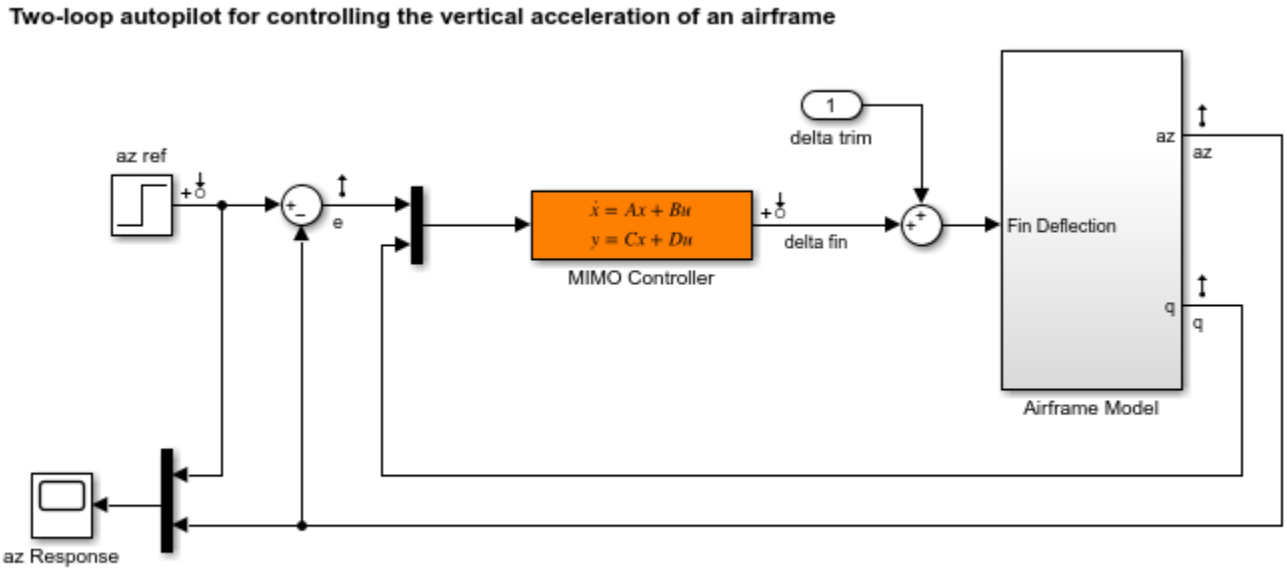
If this design is satisfactory, use `writeBlockValue` to apply the tuned values to the Simulink model and simulate the tuned controller in Simulink.

```
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:
----------
    Min          x          Max        dxMin        dx         dxMax
_________ _________ _________ _________ _________ _________

(1.) rct_airframe2/Airframe Model/Aerodynamics & Equations of Motion/ Equations of Motion (Body A
     0          0          0         -Inf         984         Inf
-3047.9999  -3047.9999  -3047.9999     -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       22.6897       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:
----------
   Min         u         Max
_________ _________ _________

(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];
```

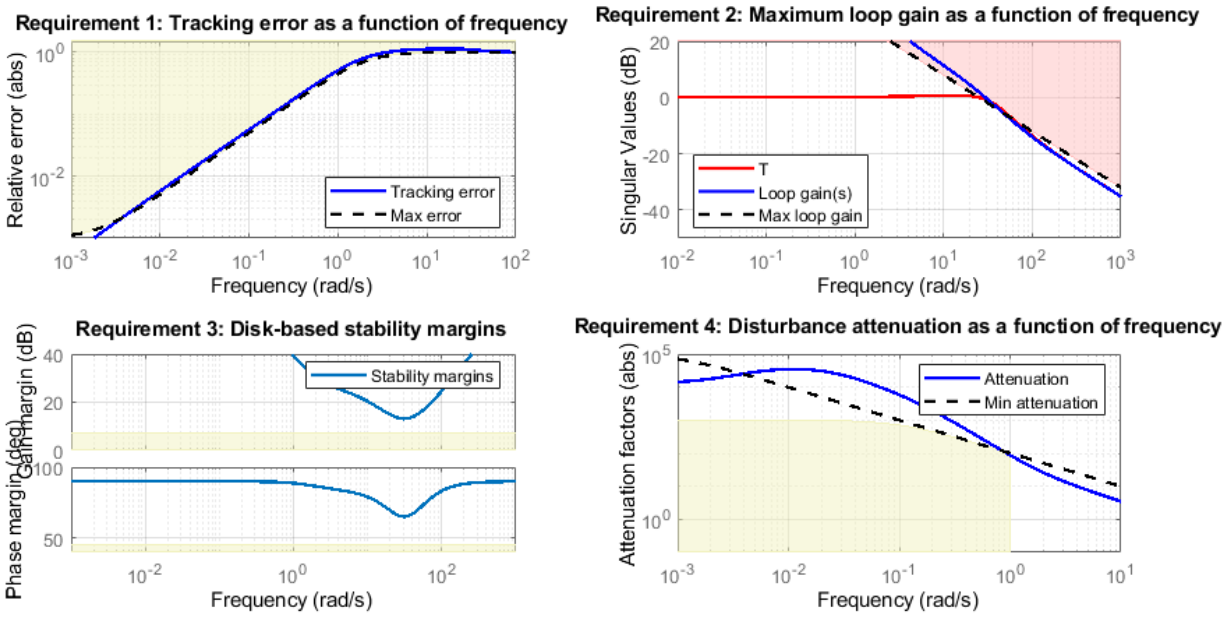
You can now use `systune` to tune the controller parameters subject to these requirements.

```
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 = 78
Final: Soft = 1.14, Hard = -Inf, Iterations = 119
```

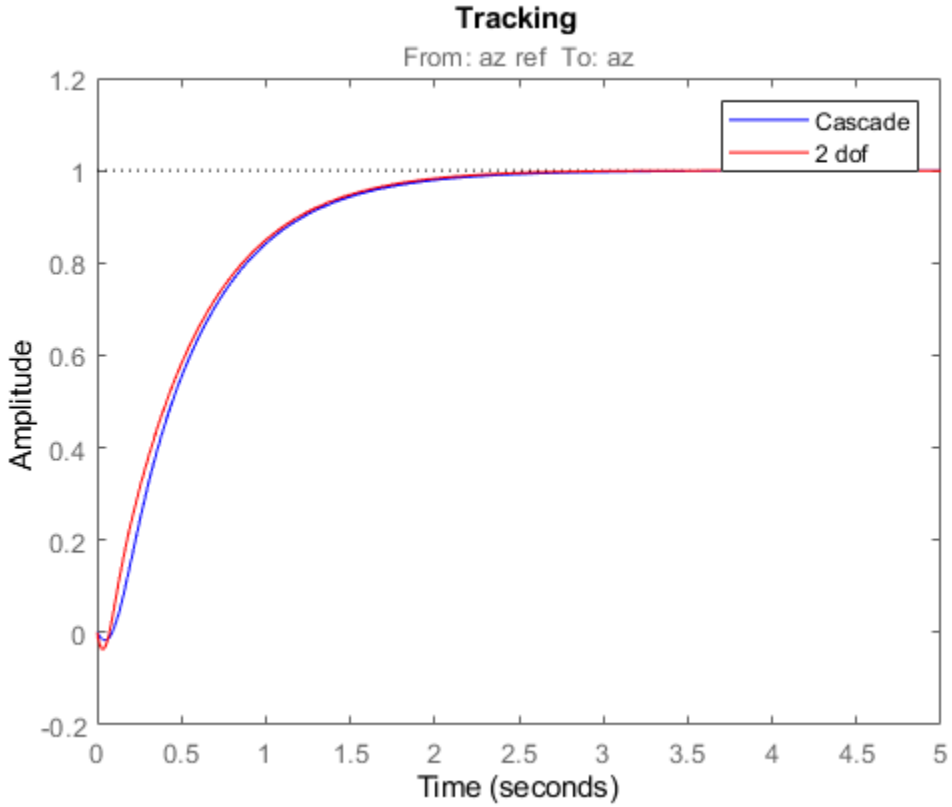
The best design has an overall objective value close to 1, indicating that all four requirements are nearly met. Use `viewGoal` to inspect each requirement for the best design.

```
figure('Position',[100,100,987,474])
viewGoal(AllReqs,ST)
```

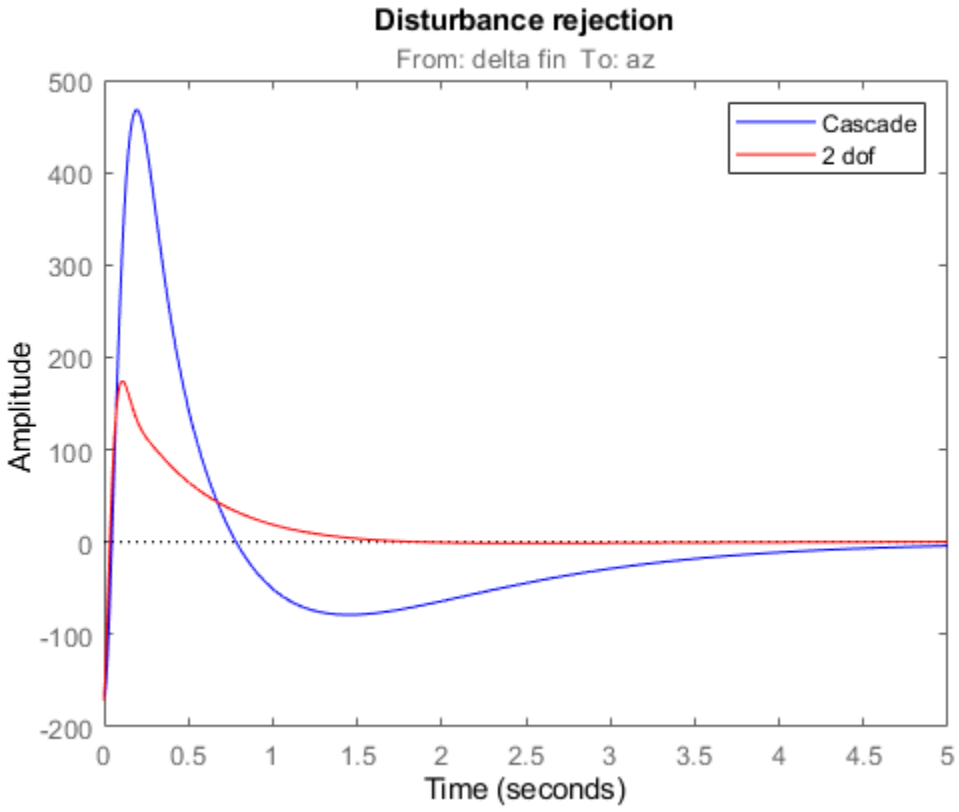


Compute the closed-loop responses and compare with the two-loop design.

```
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” on page 18-110
- “Decoupling Controller for a Distillation Column” on page 15-15

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

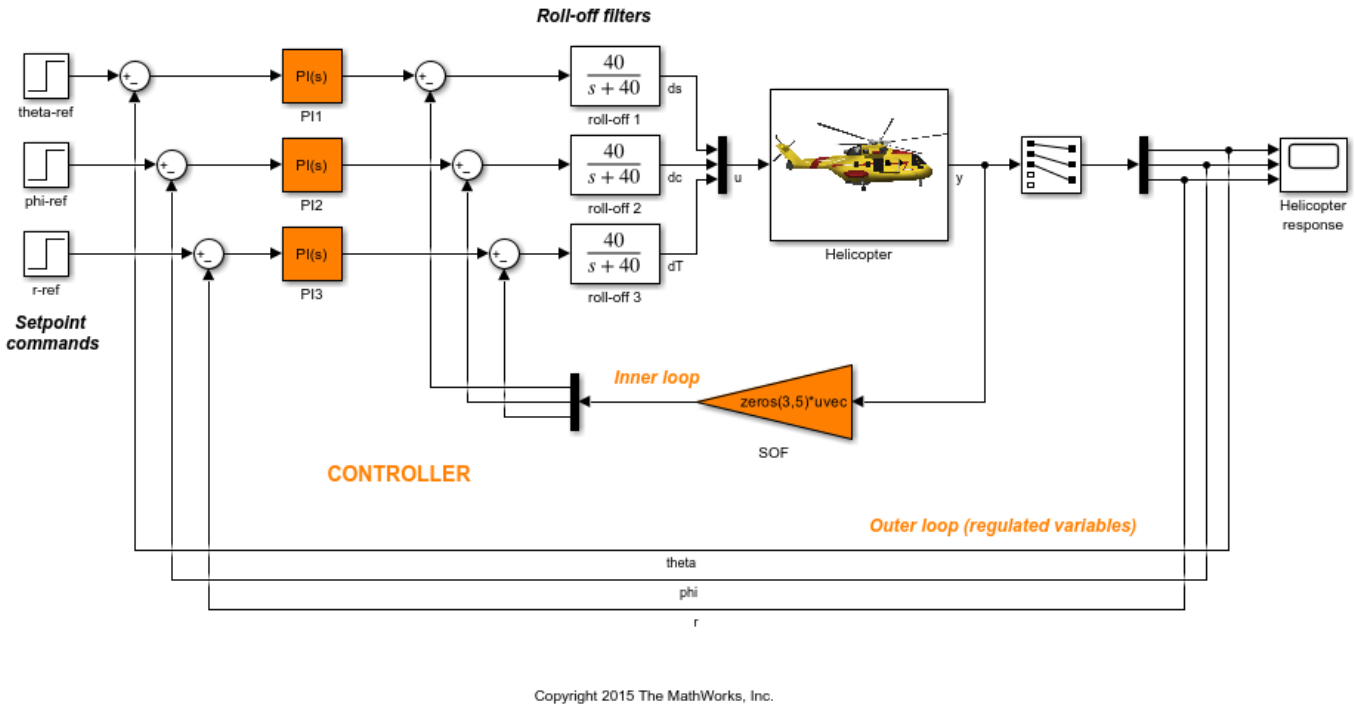
- Longitudinal velocity `u` (m/s)
- Lateral velocity `v` (m/s)
- Normal velocity `w` (m/s)
- Pitch angle `theta` (deg)
- Roll angle `phi` (deg)
- Roll rate `p` (deg/s)
- Pitch rate `q` (deg/s)
- Yaw rate `r` (deg/s).

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).

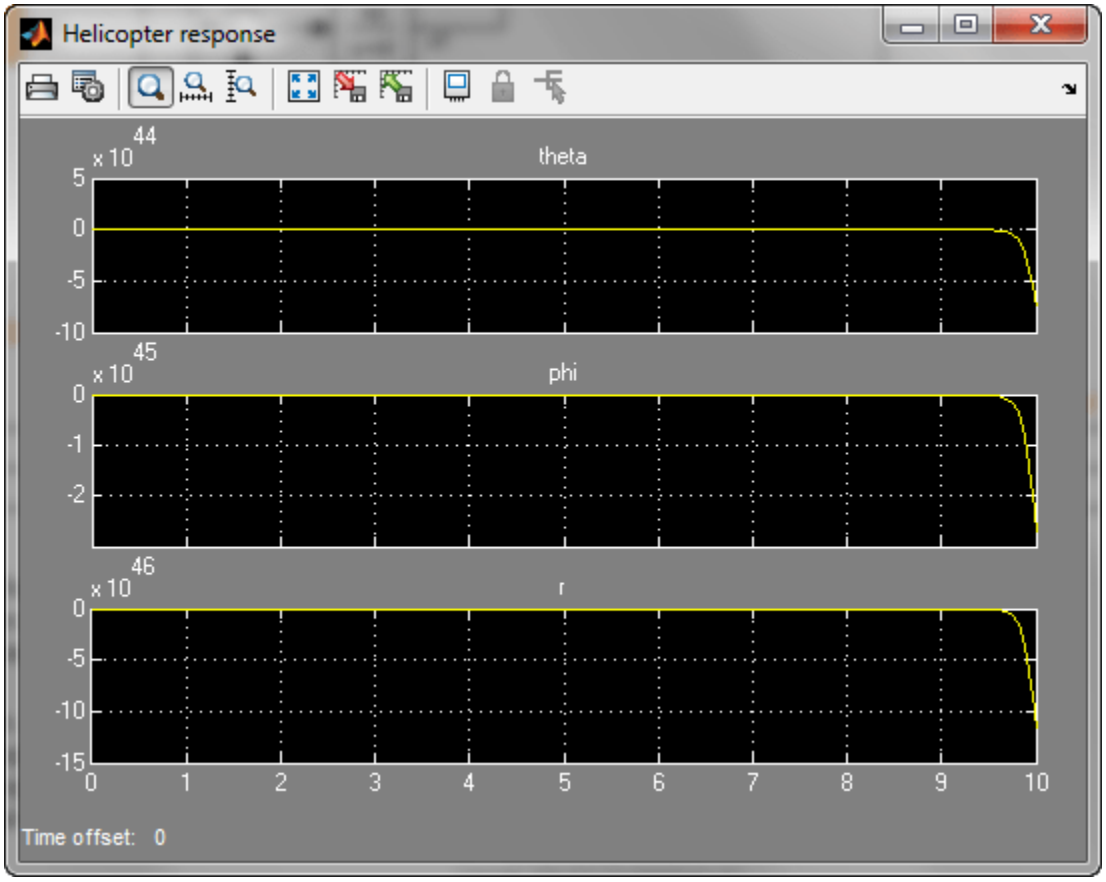
We use lowpass filters with cutoff at 40 rad/s to partially enforce the second objective.

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);
```

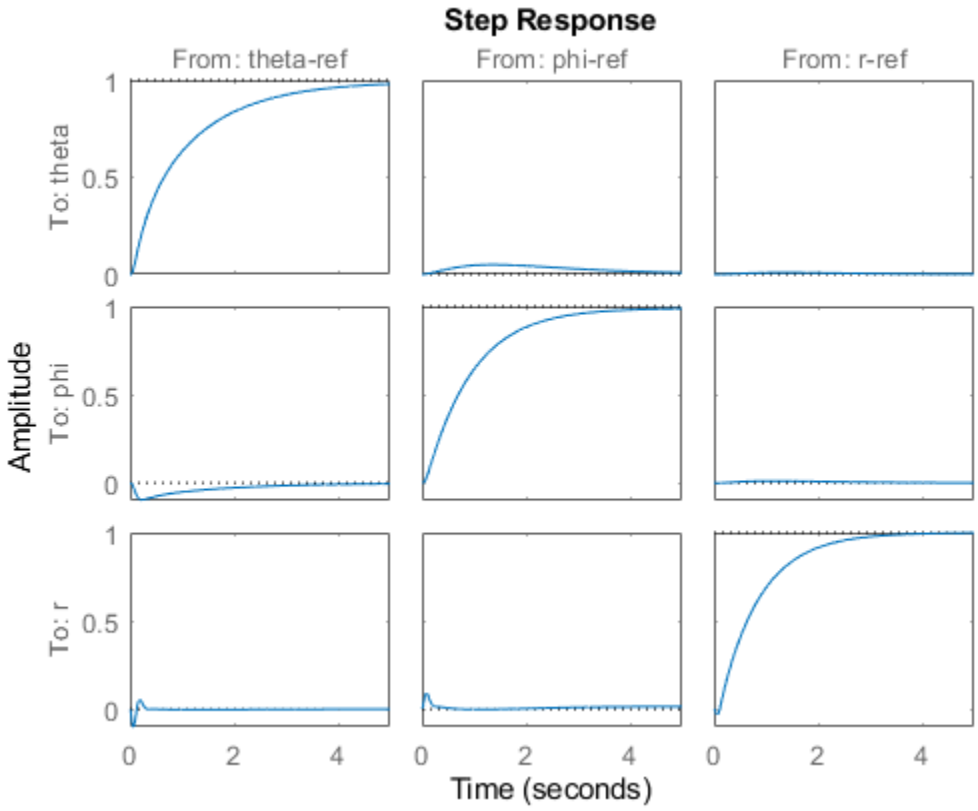
You can now use `systune` to jointly tune all controller parameters. This returns the tuned version `ST1` of the control system `ST0`.

```
AllReqs = [TrackReq,MarginReq1,MarginReq2,PoleReq];
ST1 = systune(ST0,AllReqs);
```

```
Final: Soft = 1.12, Hard = -Inf, Iterations = 71
```

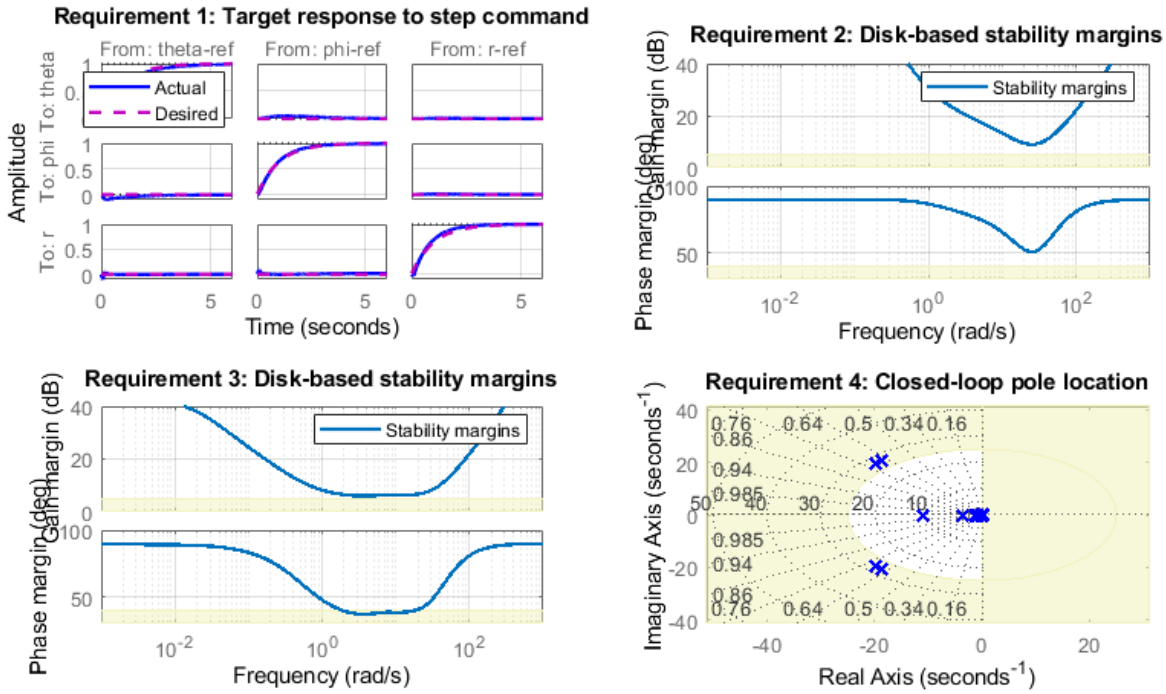
The final value is close to 1 so the requirements are nearly met. Plot the tuned responses to step commands in theta, phi, r:

```
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 = 2.07

Name: PI1
Continuous-time PI controller in parallel form.

-----------------------------------

Block 2: rct_helico/PI2 =

             1
  Kp + Ki * ---
             s

  with Kp = -0.099, Ki = -1.35

Name: PI2
Continuous-time PI controller in parallel form.

-----------------------------------

Block 3: rct_helico/PI3 =
```

```
             1
  Kp + Ki * ---
             s

  with Kp = 0.137, Ki = -2.2

Name: PI3
Continuous-time PI controller in parallel form.

-----------------------------------

Block 4: rct_helico/SOF =

  D =
              u1        u2        u3        u4        u5
   y1     2.211     -0.31  -0.00336    0.7854  -0.01518
   y2   -0.1923    -1.291   0.01821  -0.08501   -0.1195
   y3  -0.01941  -0.01208    -1.895  -0.00412   0.06795

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:

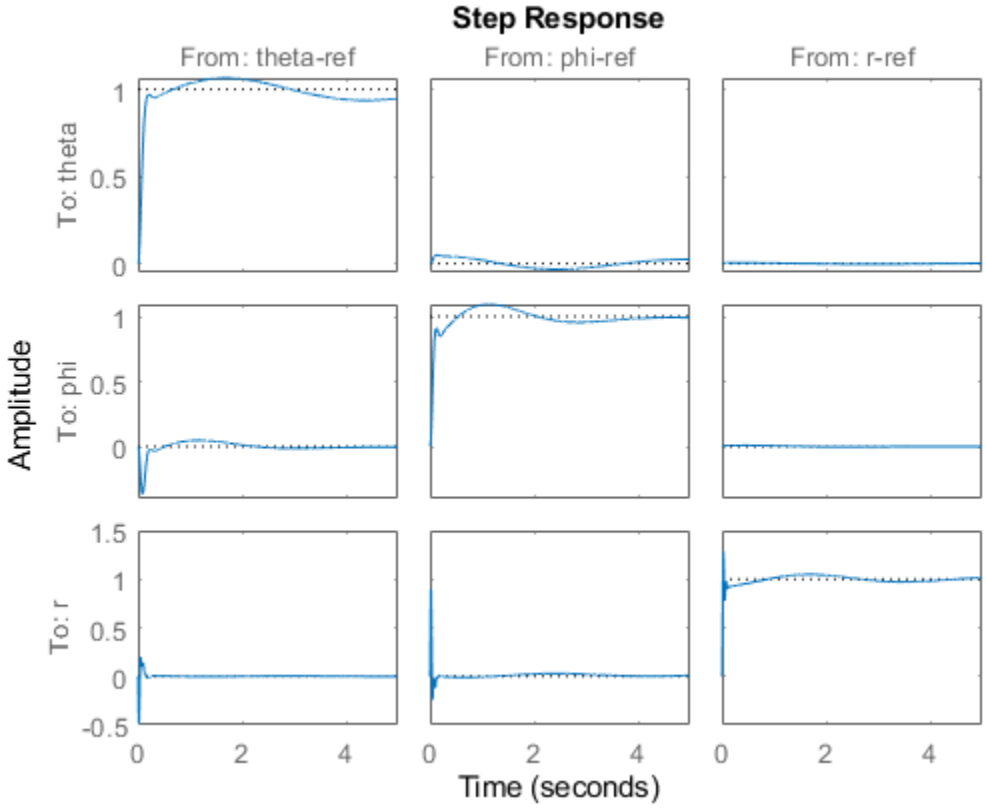
```
addOpening(ST0,'SOF')
```

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” on page 18-176

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.

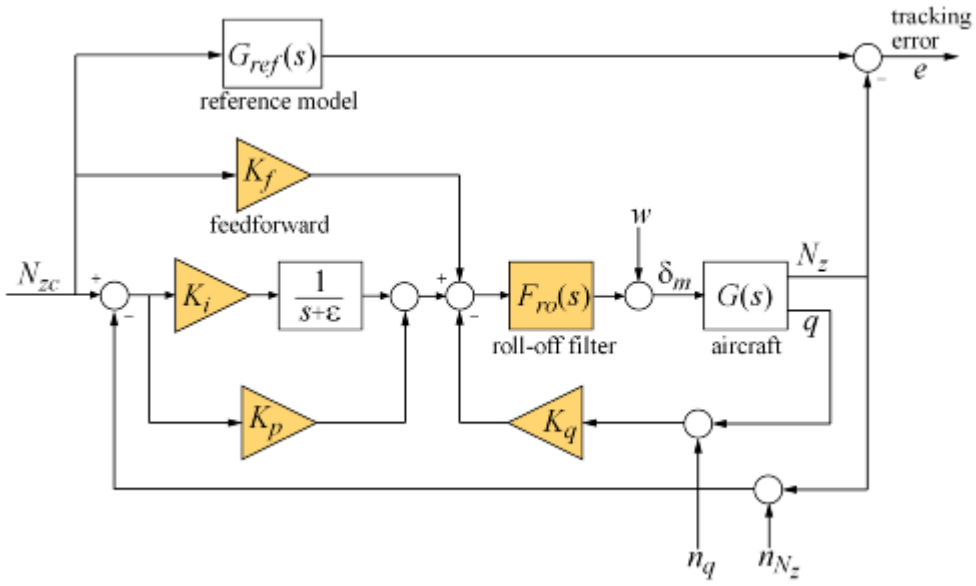
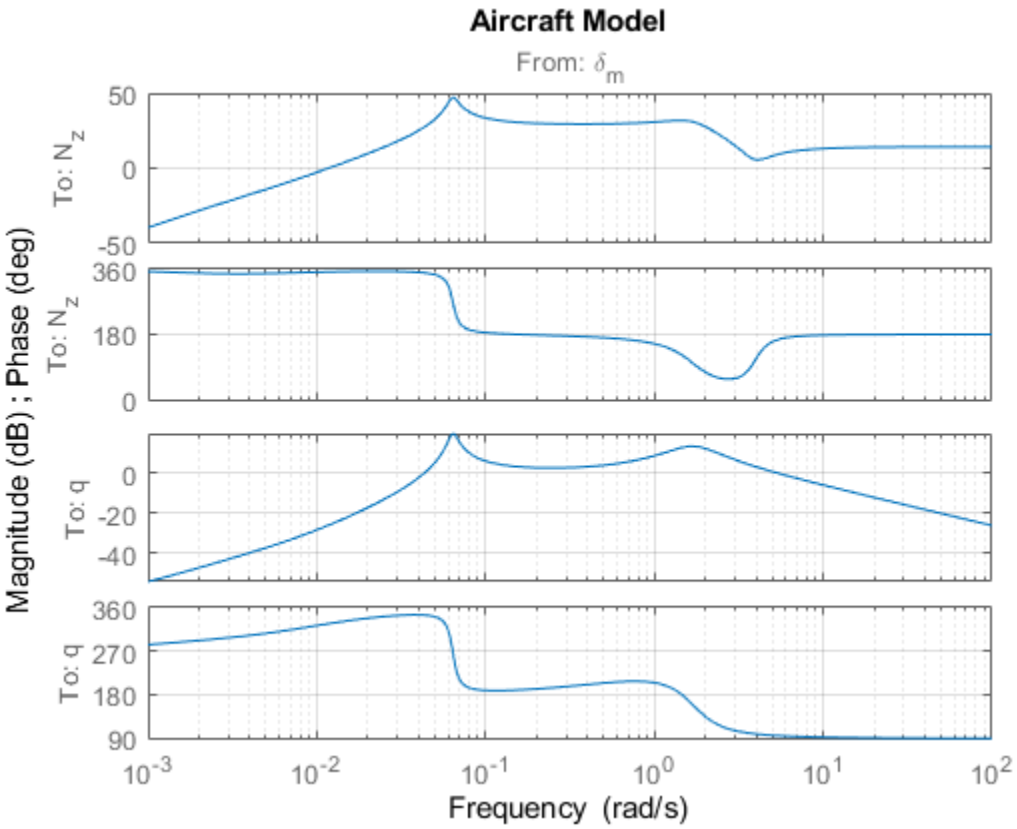


Figure 1: Longitudinal Autopilot Configuration.

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')
```

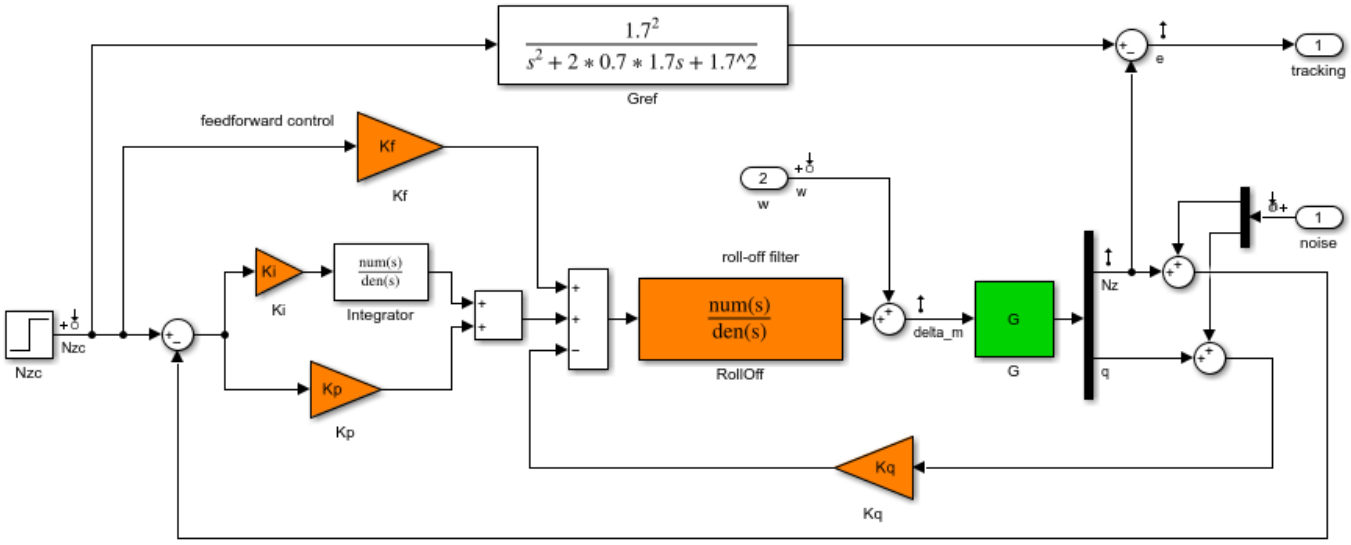


Note the zero at the origin in $G(s)$. Because of this zero, we cannot achieve zero steady-state error and must instead focus on the transient response to acceleration commands. Note that acceleration commands are transient in nature so steady-state behavior is not a concern. This zero at the origin 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

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.

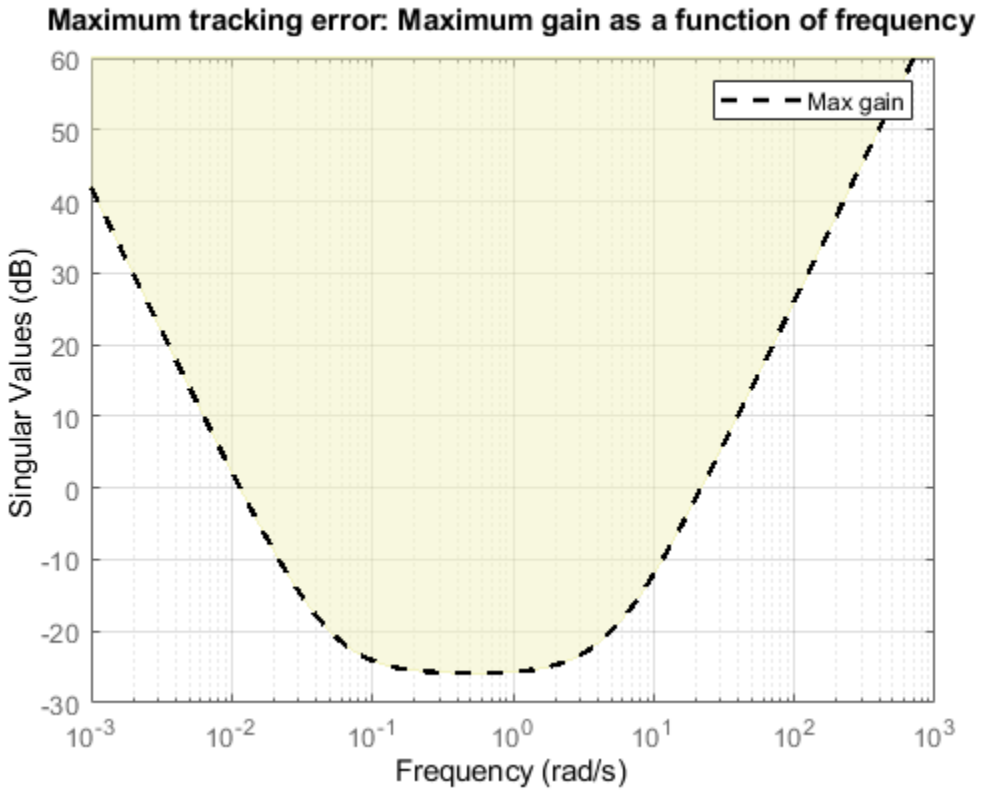
3. **Stability margins**: The stability margins at the plant input δ_m should be at least 7 dB and 45 degrees.

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';
```

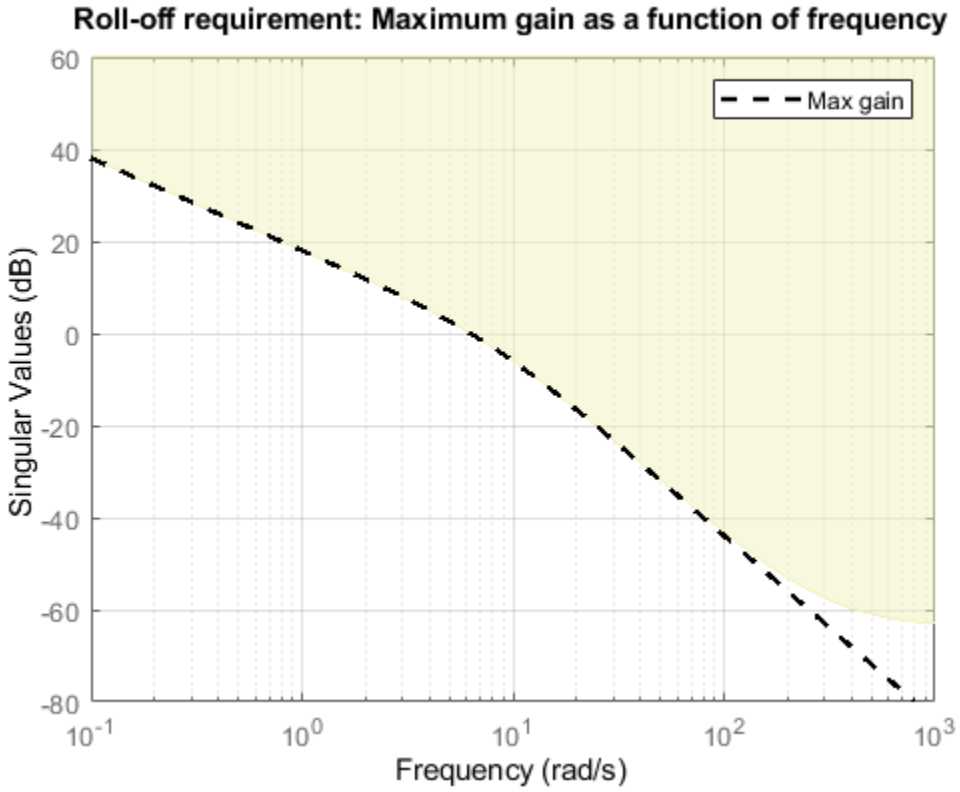
The `TuningGoal.Gain` constructor automatically turns the maximum error sketch into a smooth weighting function. Use `viewGoal` to graphically verify the desired error profile.

```
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 = 0.966, Hard = -Inf, Iterations = 131
```

Use `showTunable` to see the tuned block values.

```
showTunable(ST)
```

```
Block 1: rct_concorde/Ki =

  D =
            u1
   y1  -0.02949

Name: Ki
Static gain.

-----------------------------------

Block 2: rct_concorde/Kp =

  D =
             u1
   y1  -0.009886

Name: Kp
Static gain.

-----------------------------------

Block 3: rct_concorde/Kq =

  D =
           u1
   y1  -0.2812

Name: Kq
Static gain.

-----------------------------------

Block 4: rct_concorde/Kf =

  D =
            u1
   y1  -0.02201

Name: Kf
Static gain.

-----------------------------------

wn = 4.78
-----------------------------------
zeta = 0.504
```

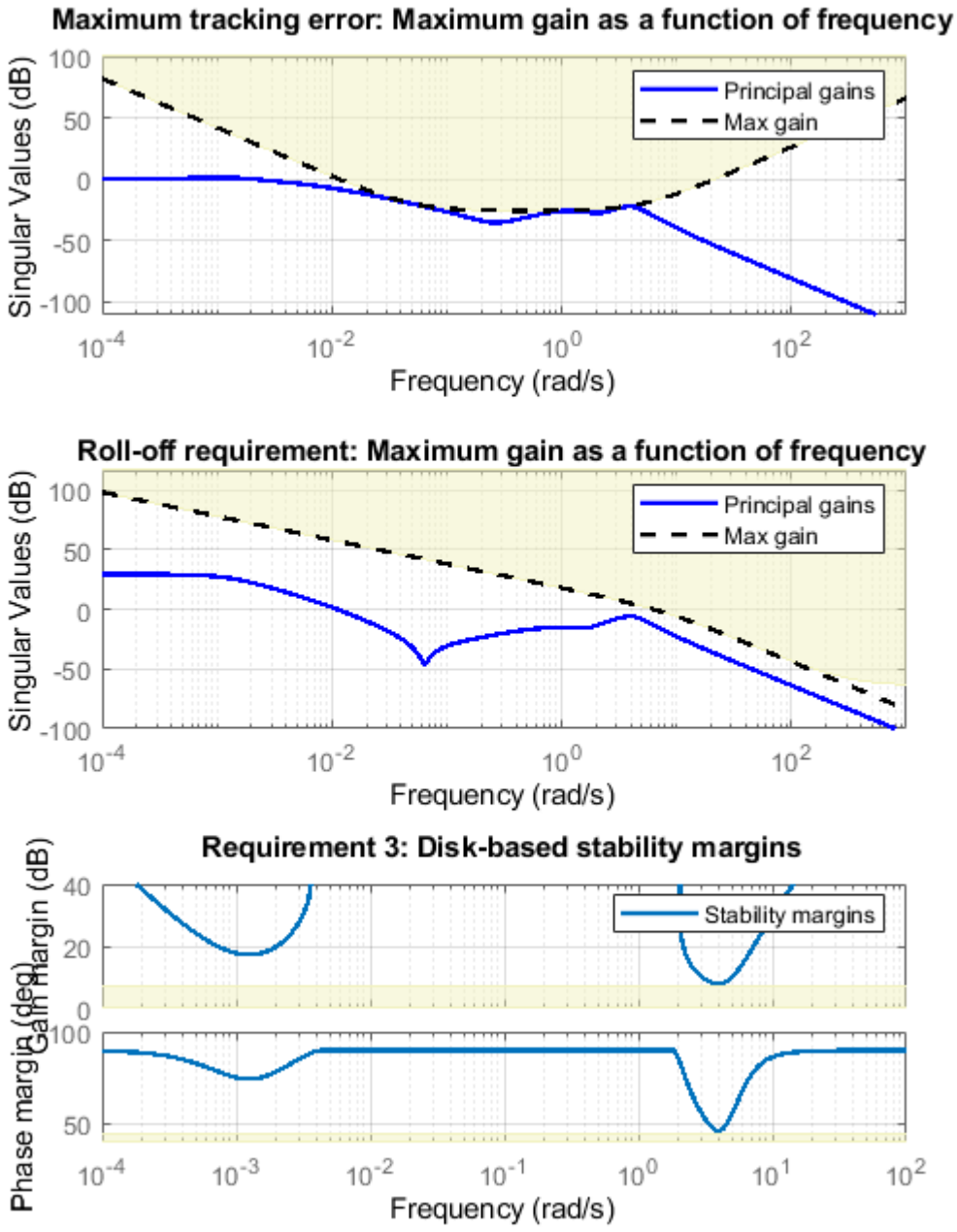
To get the tuned value of $F_{ro}(s)$, use `getBlockValue` to evaluate `Fro` for the tuned parameter values in `ST`:

```
Fro = getBlockValue(ST,'RollOff');
tf(Fro)
```

```
ans =

         22.84
  --------------------
  s^2 + 4.82 s + 22.84

Continuous-time transfer function.
```

Finally, use `viewGoal` to graphically verify that all requirements are satisfied.

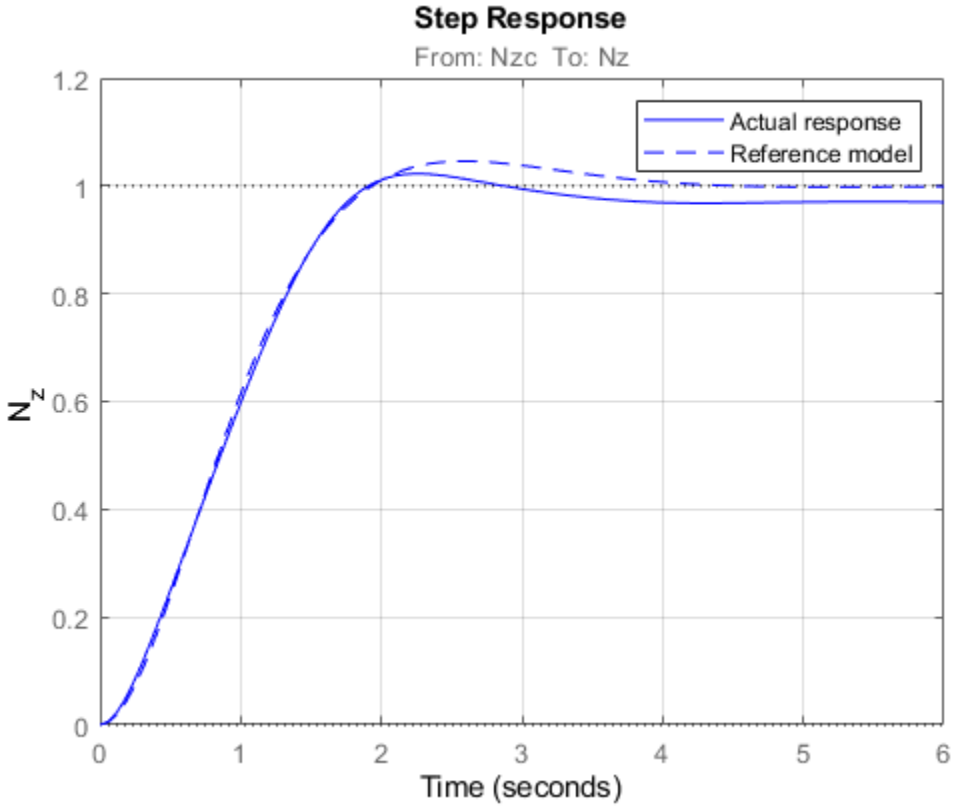
```
figure('Position',[100,100,550,710])
viewGoal([Req1 Req2 Req3],ST)
```



Closed-Loop Simulations

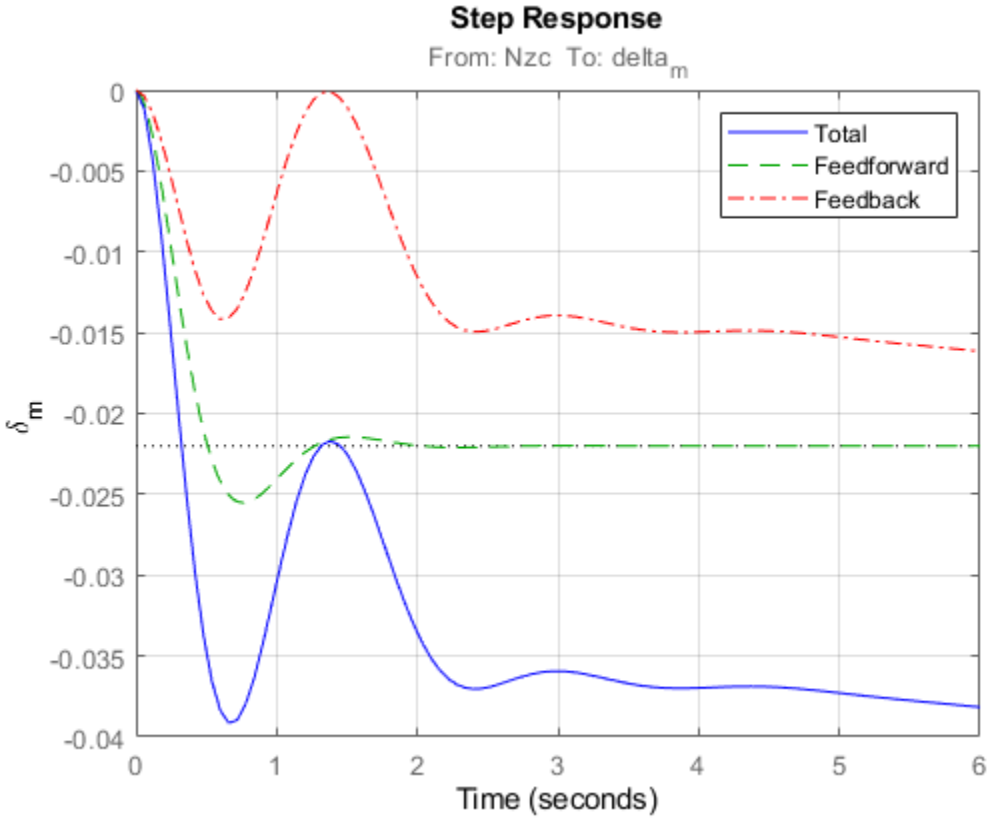
We now verify that the tuned autopilot satisfies the design requirements. First compare the step response of N_z with the step response of the reference model $G_{ref}(s)$. Again use `getIOTransfer` to compute the tuned closed-loop transfer from `Nzc` to `Nz`:

```
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')
```



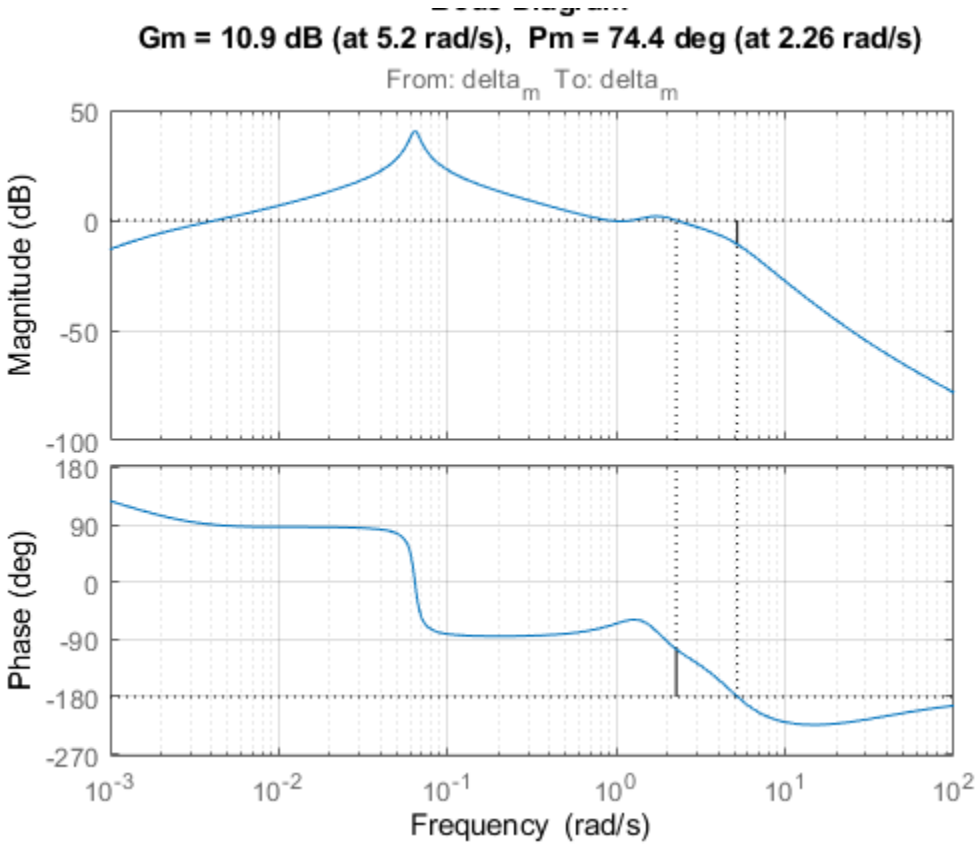
Also plot the deflection δ_m and the respective contributions of the feedforward and feedback paths:

```
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” on page 18-187

Fault-Tolerant Control of a Passenger Jet

This example shows how to tune a fixed-structure controller for multiple operating modes of the plant.

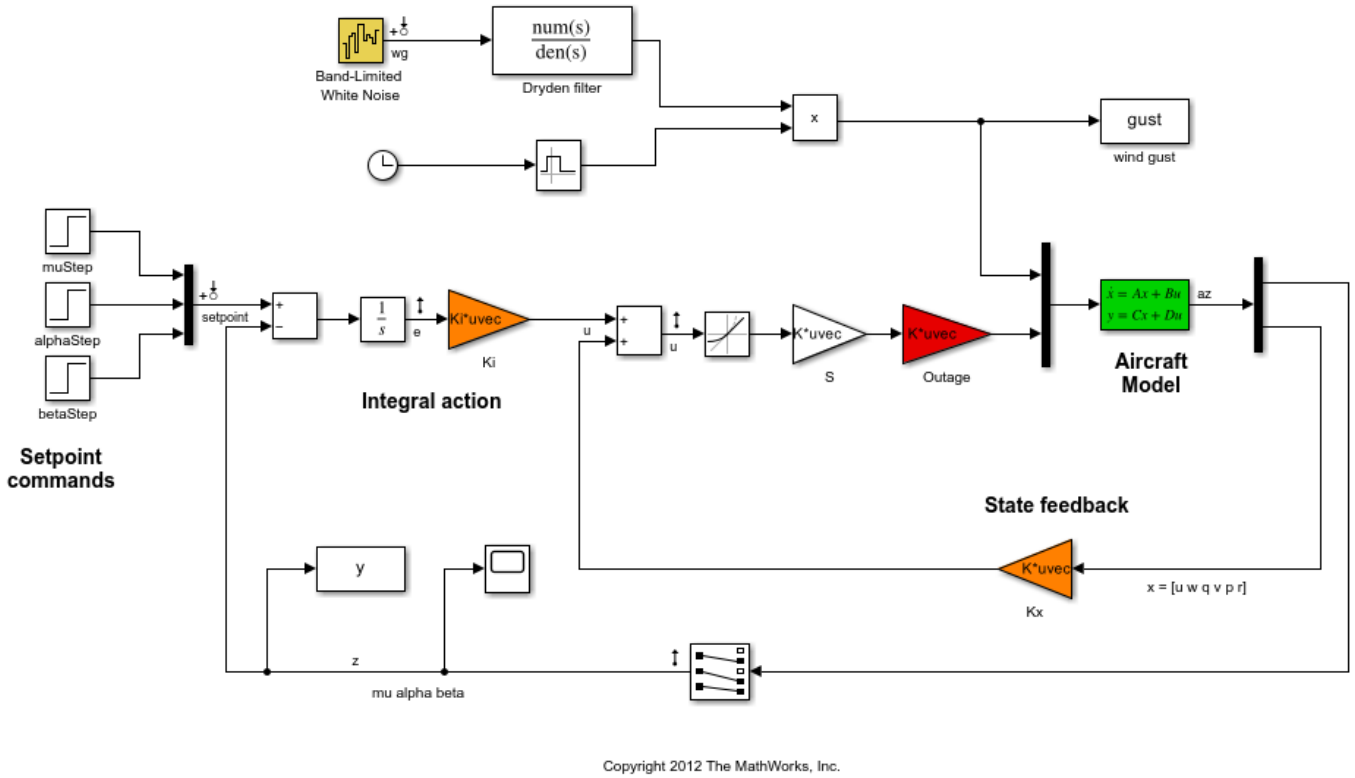
Background

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.

```
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);
```

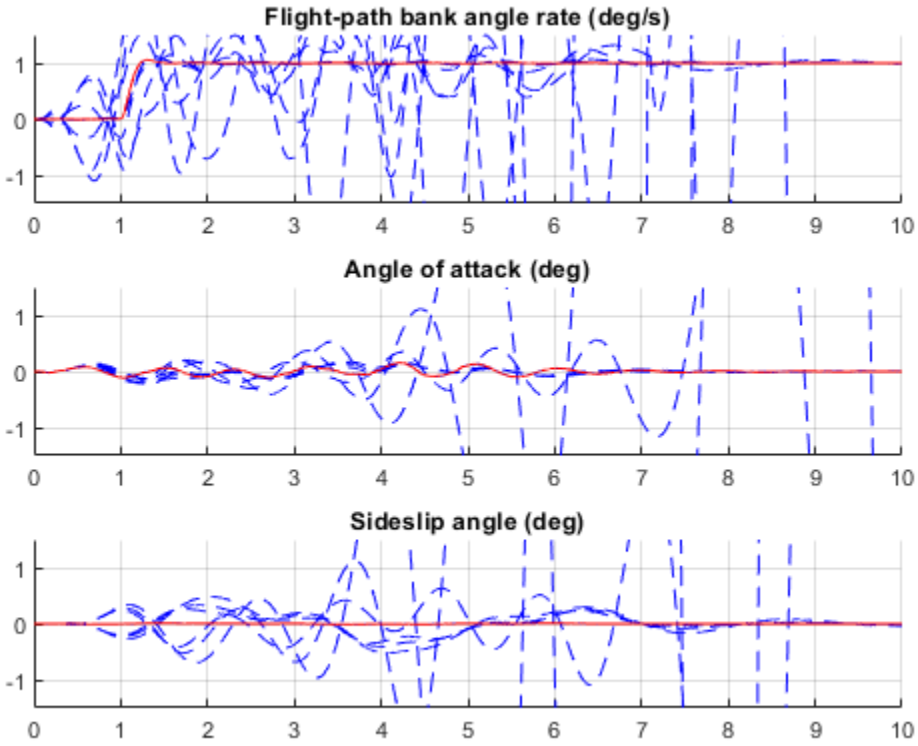
Use `systune` to tune the controller gains subject to the nominal requirements. Treat the wind gust alleviation as a hard constraint.

```
[ST,fSoft,gHard]  = systune(ST0,SoftNom,HardNom);
```

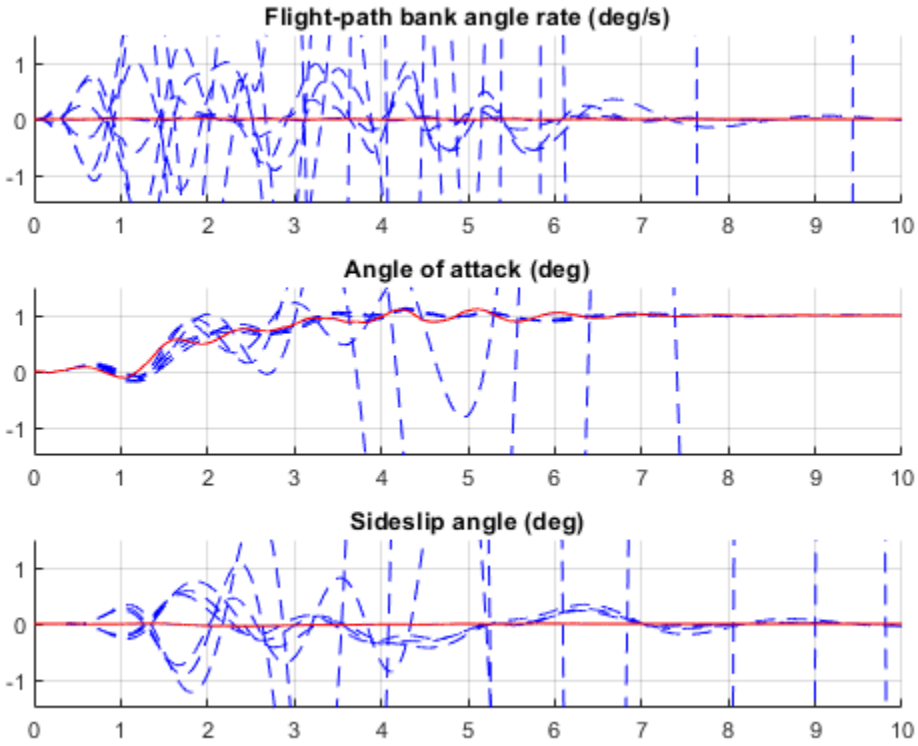
```
Final: Soft = 22.6, Hard = 0.99919, Iterations = 283
```

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.

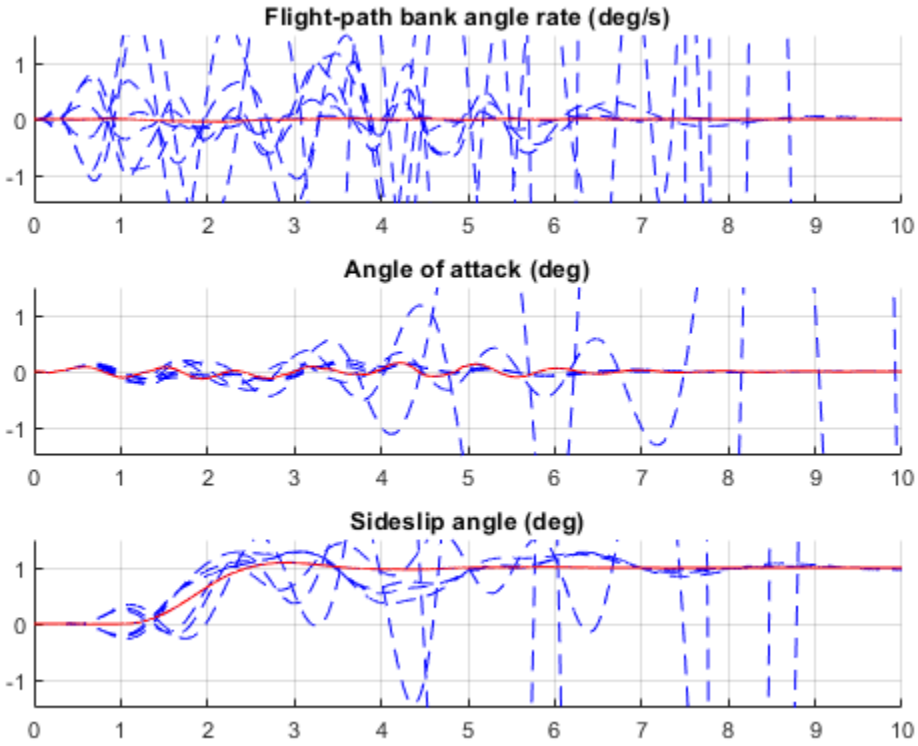
```
Ki = getBlockValue(ST, 'Ki');  Ki = Ki.d;
Kx = getBlockValue(ST, 'Kx');  Kx = Kx.d;

% 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

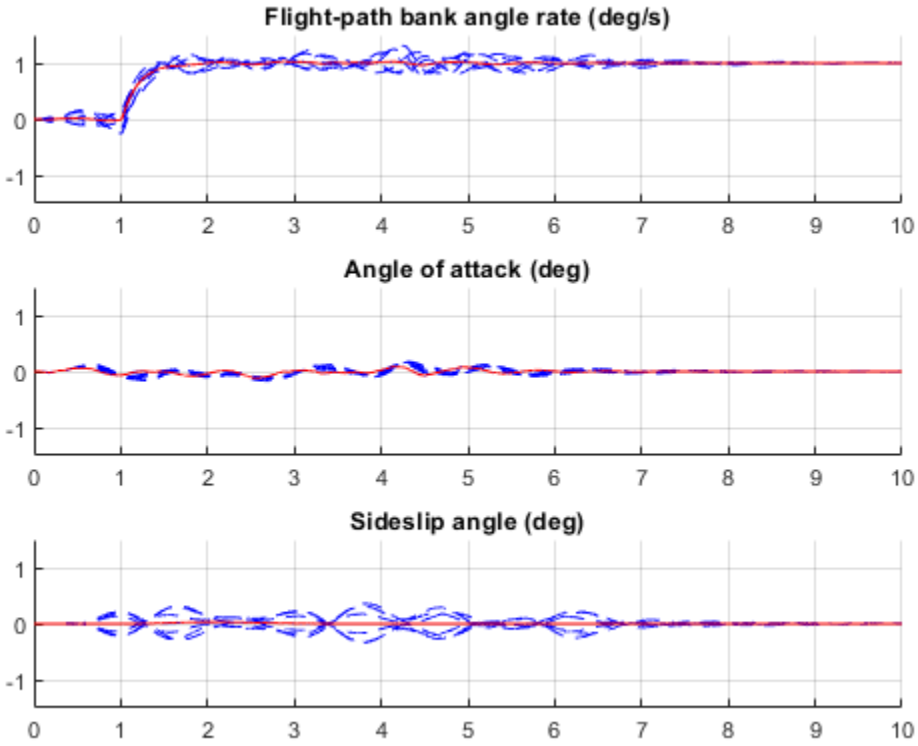
To improve reliability, retune the controller gains to meet the nominal requirement for the nominal plant as well as the relaxed requirements for all eight outage scenarios.

```
[ST,fSoft,gHard]  = systune(ST0,[SoftNom;SoftOut],[HardNom;HardOut]);
```

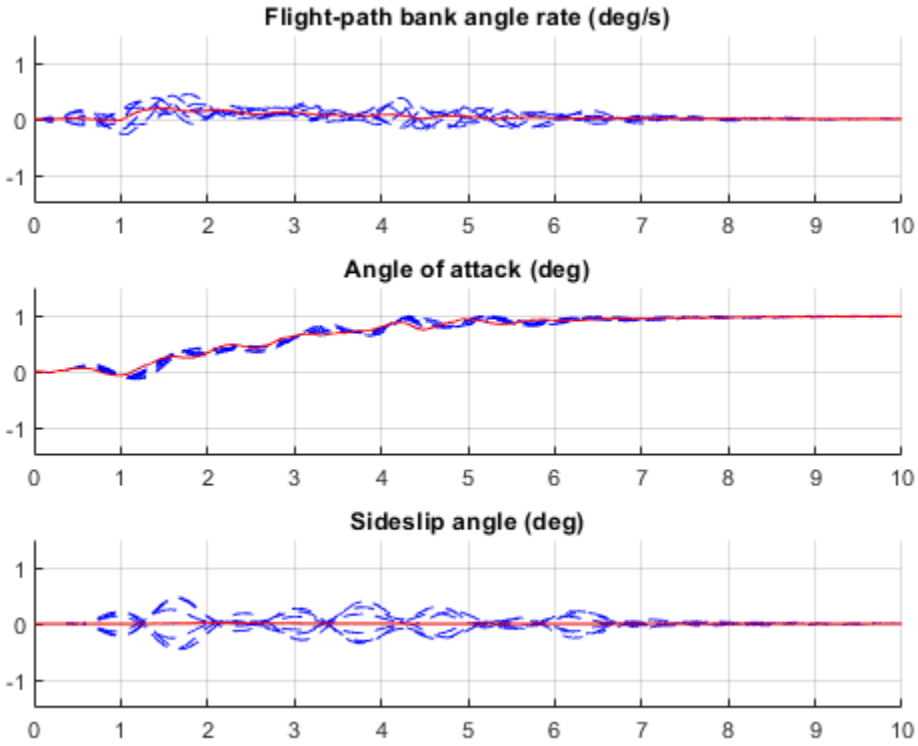
```
Final: Soft = 25.8, Hard = 0.99965, Iterations = 463
```

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).

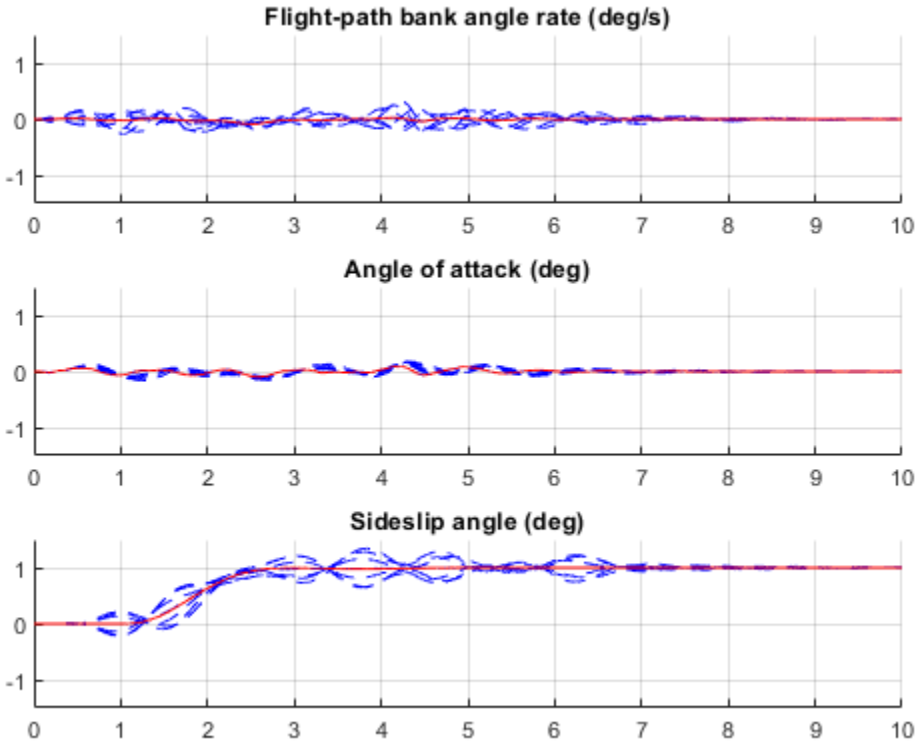
```
Ki = getBlockValue(ST, 'Ki');  Ki = Ki.d;
Kx = getBlockValue(ST, 'Kx');  Kx = Kx.d;

% 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.

See Also

`systune (slTuner)` | `slTuner` | `TuningGoal.WeightedVariance` | `TuningGoal.Variance`

Related Examples

- “Fixed-Structure Autopilot for a Passenger Jet” on page 18-176

Passive Control of Water Tank Level

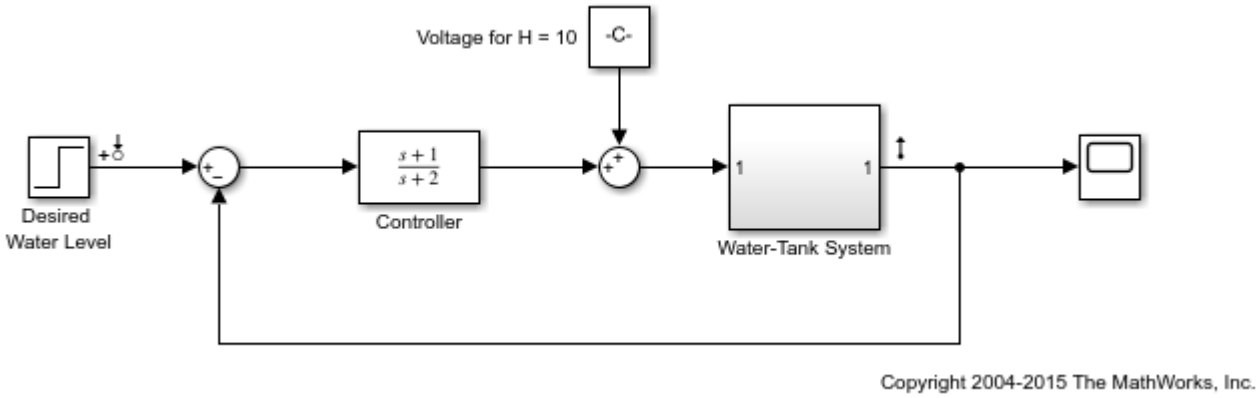
In this example, you learn how to use **Control System Tuner** app to design a controller for a nonlinear plant modeled in Simulink®. You accomplish the following tasks:

- Configure the model and app for compensator tuning
- Tune a first-order compensator using passivity-based design
- Simulate the closed-loop nonlinear response.

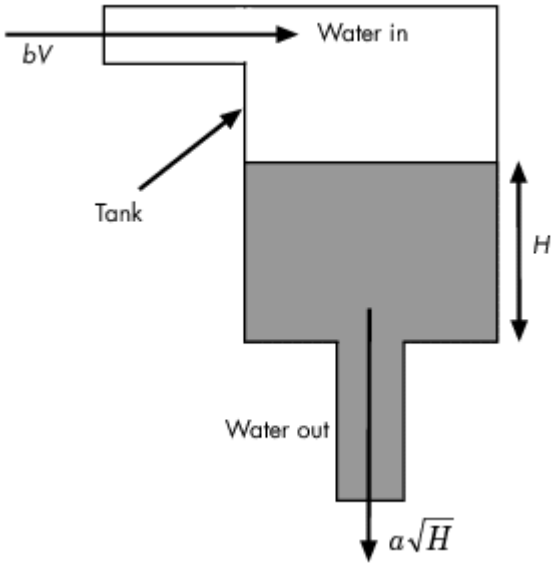
Simulink Model of the Control System

The `cst_watertank_comp_design model`, 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

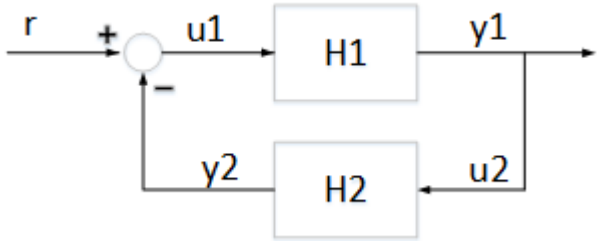
- $x = H$ denotes the height of water in the tank
- u denotes the voltage applied to the pump
- A denotes the cross-sectional area of the tank
- a and b are constants related to the flow rate into and out of the tank

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.



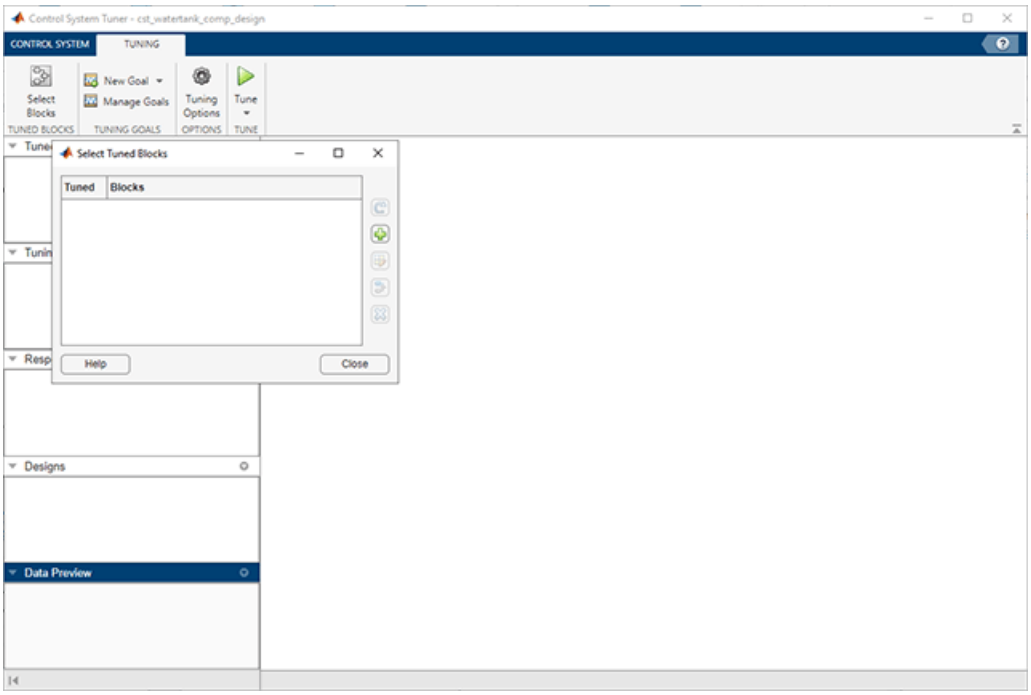
Since the water tank system is passive, it makes sense to require that the controller be strictly passive to guarantee closed-loop stability even when the plant model is inaccurate.

Compensator Tuning Using Control System Tuner

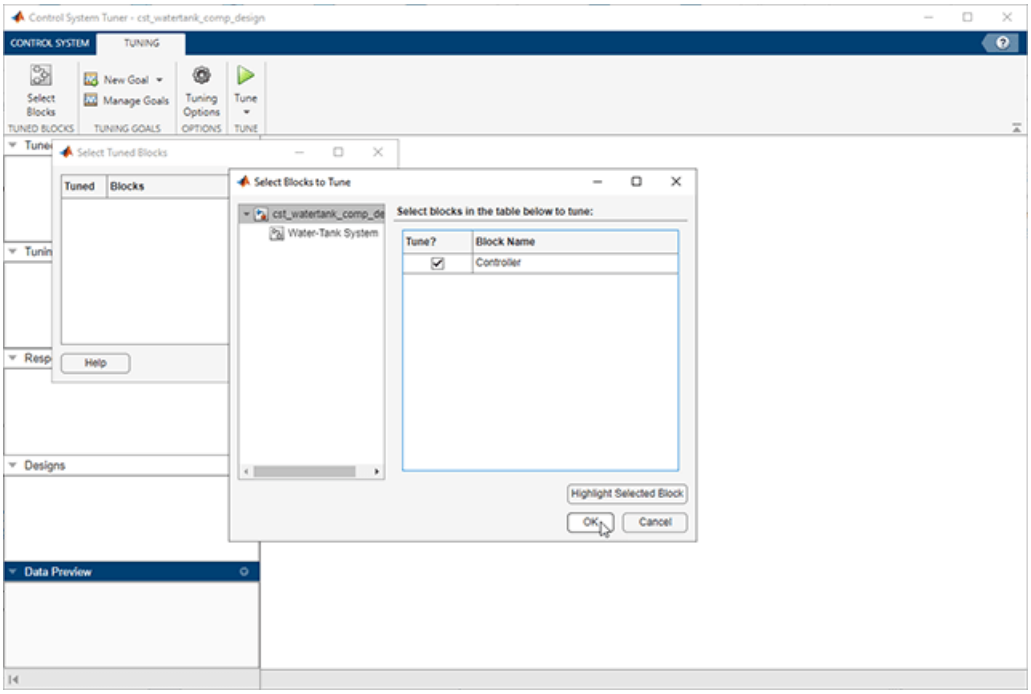
You can use the **Control System Tuner** app to tune the Controller block.

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



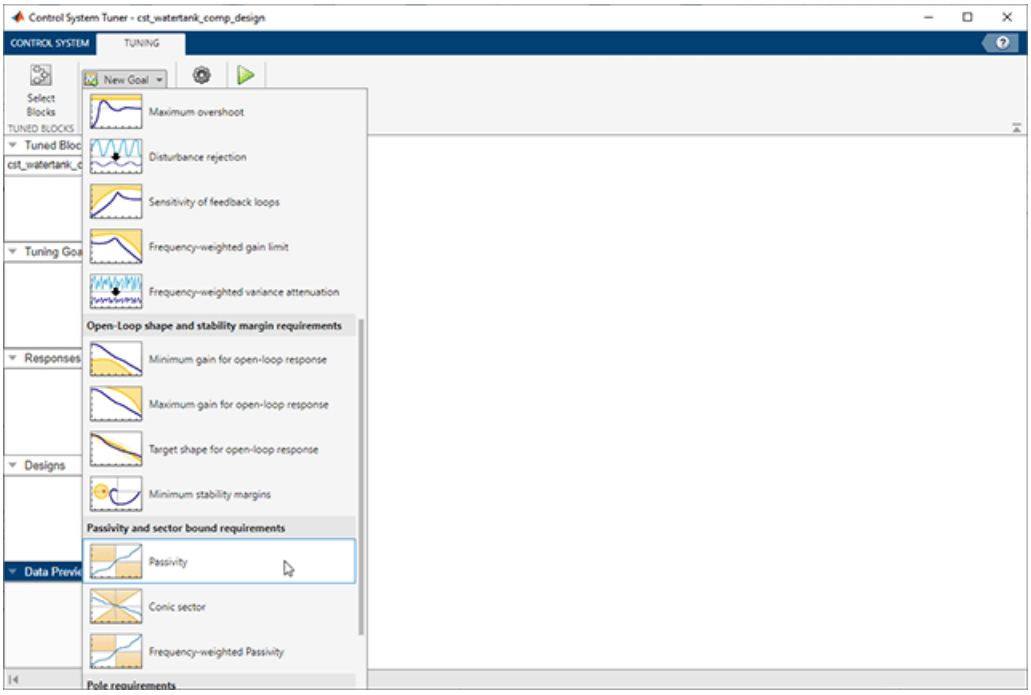
Step 3: Select the Controller block and Click OK. This block now appears in the Tuned Blocks list.



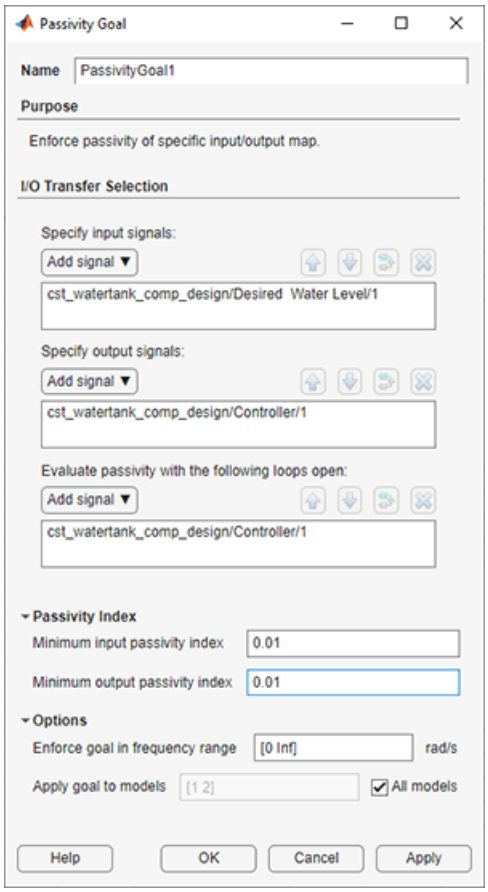
Step 4: Specify the tuning goals. Here, there are two main goals:

1. Track step changes in water level
2. Make the controller passive

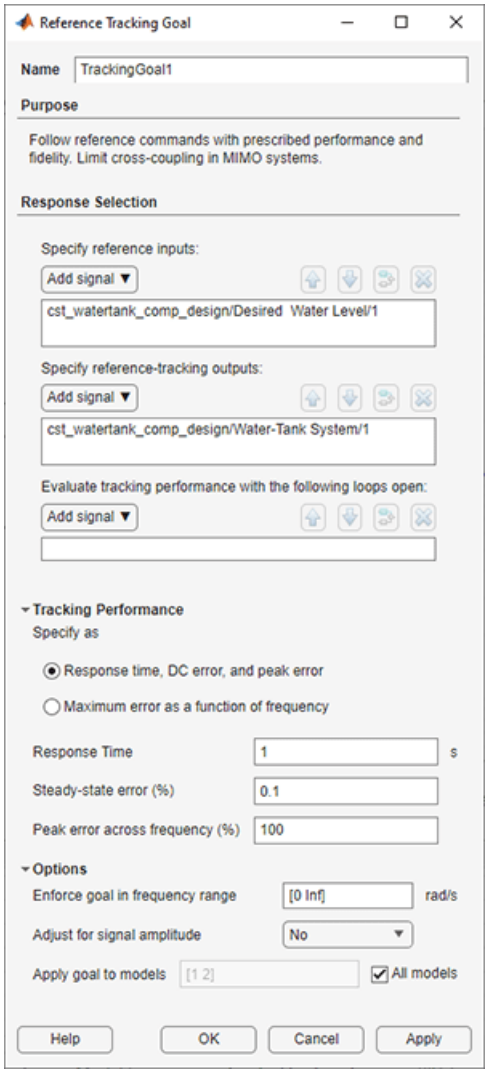
Click the **New Goal** drop-down list, and first add a **Passivity** goal.



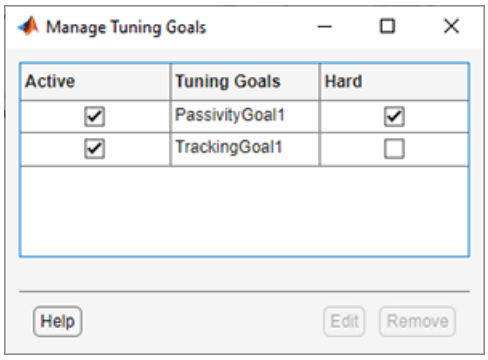
Configure this goal to apply to the Controller block only. This is done by setting the input signal to be the "Desired Water Level", the output signal to be the output of the Controller block, and the loop opening to be at the Controller block output. Also specify minimum passivity indices of 0.01 at the inputs and outputs to enforce strict passivity.



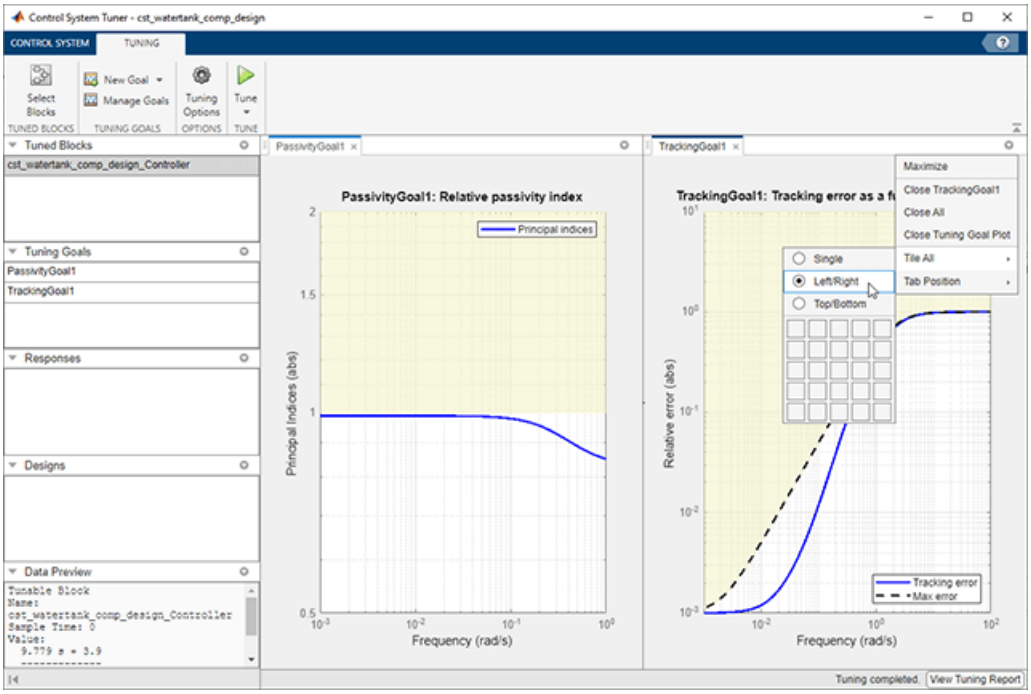
Next, add a **Reference Tracking** goal from the **New Goal** drop-down list. Configure this goal for a 1 second response time.



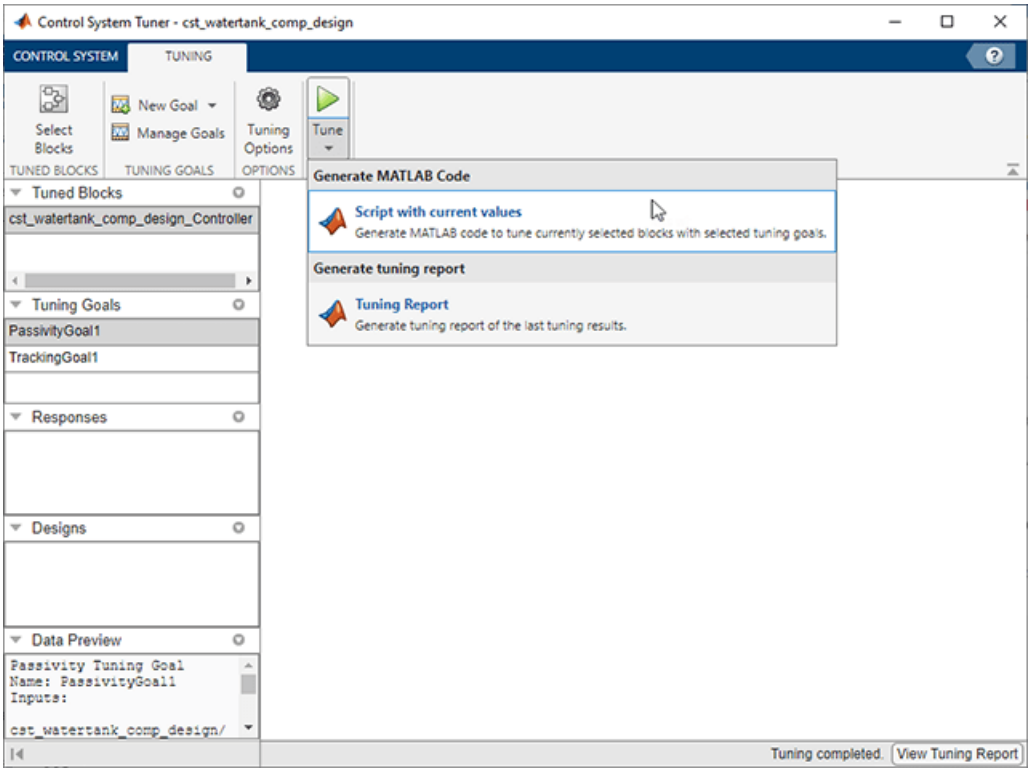
Finally, click on the **Manage Goals** button off the **Tuning** tab and mark the Passivity goal as a hard tuning constraint.



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.

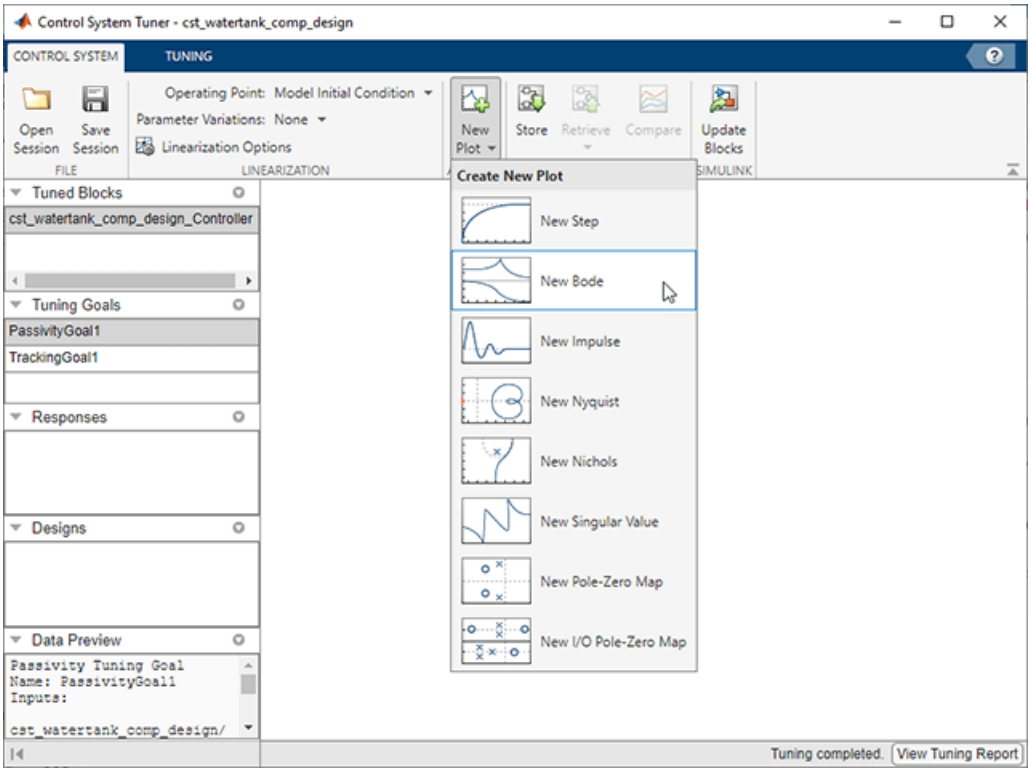


You can further analyze these results by generating a MATLAB script that reproduces this tuning process.

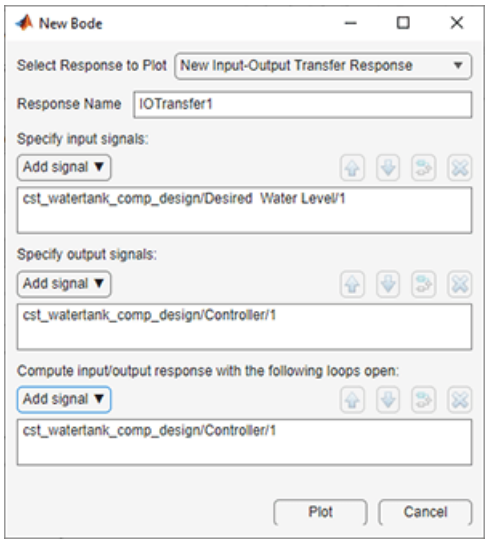


Closed-Loop Simulation

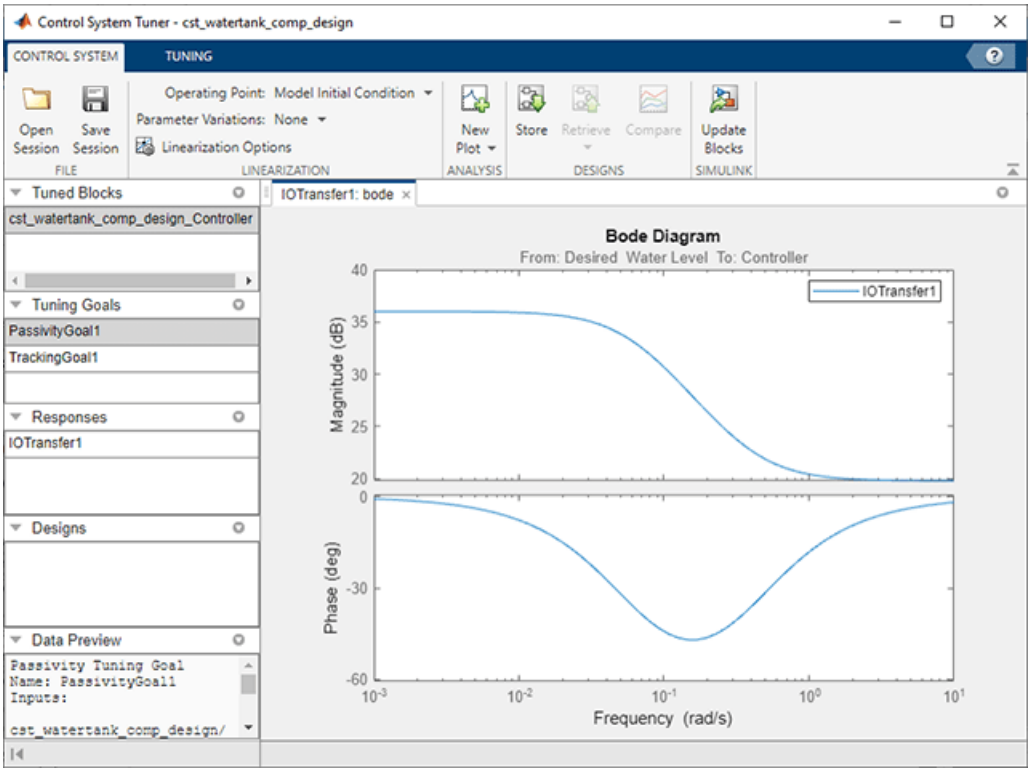
You can view the Bode plot of the tuned controller. Click on the **New Plot** button off the **Control System** tab. Select **New Bode** from the drop-down list.



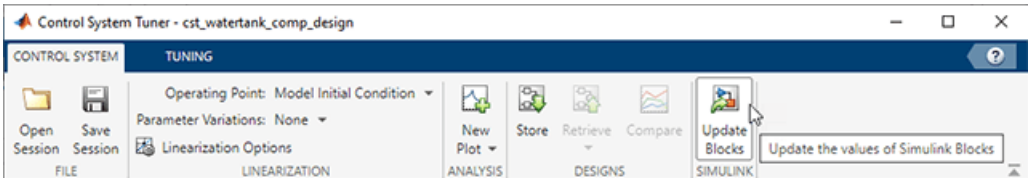
The controller response can be specified as follows.



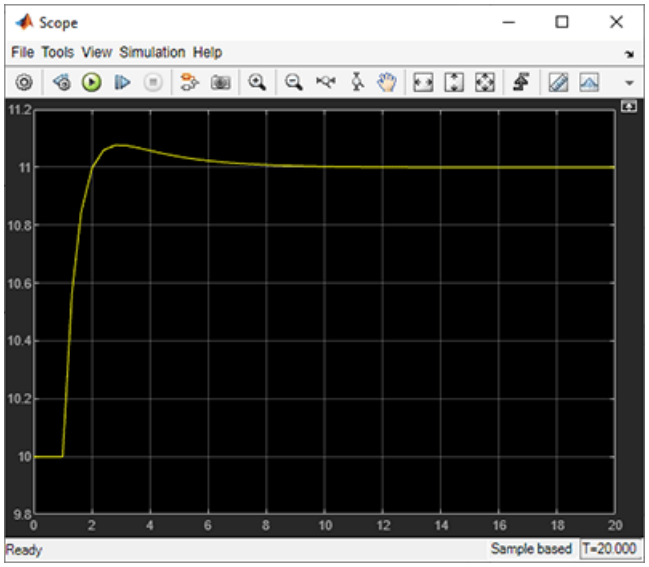
Click on the **Plot** button. The bode plot is shown in the following figure.



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” on page 10-2
- “Vibration Control in Flexible Beam” on page 17-25

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.

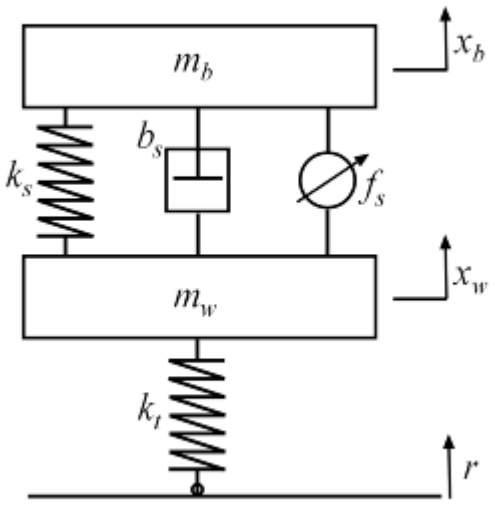
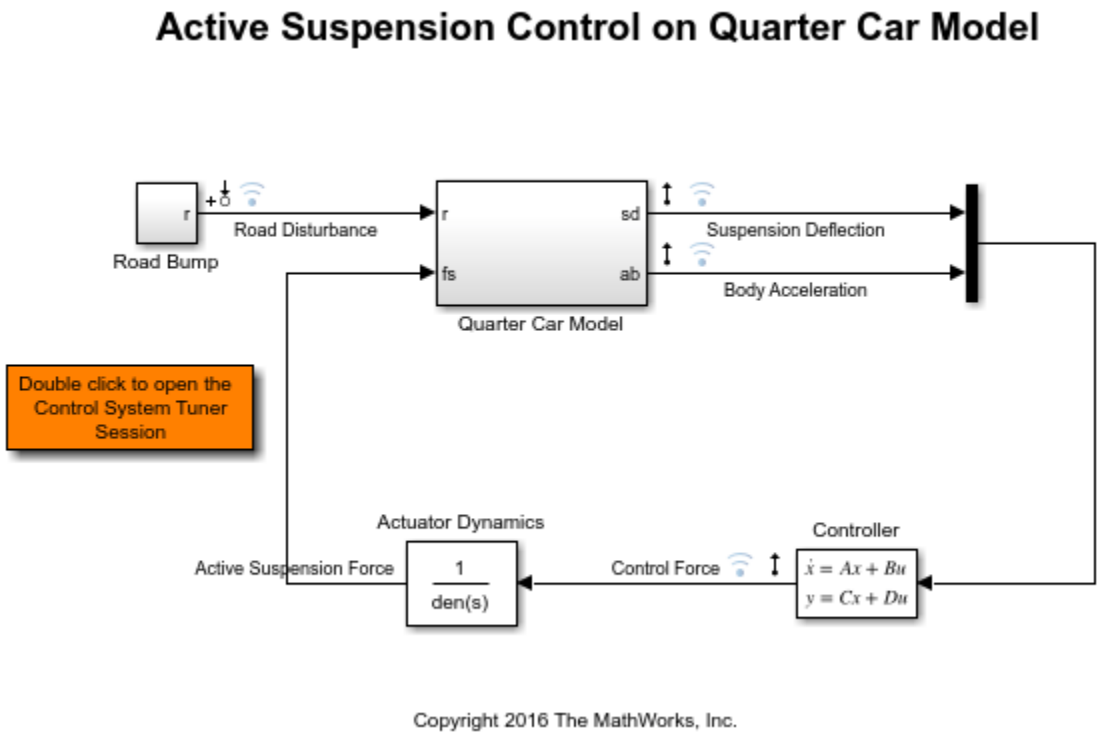


Figure 1: Quarter-car model of active suspension.

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 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.

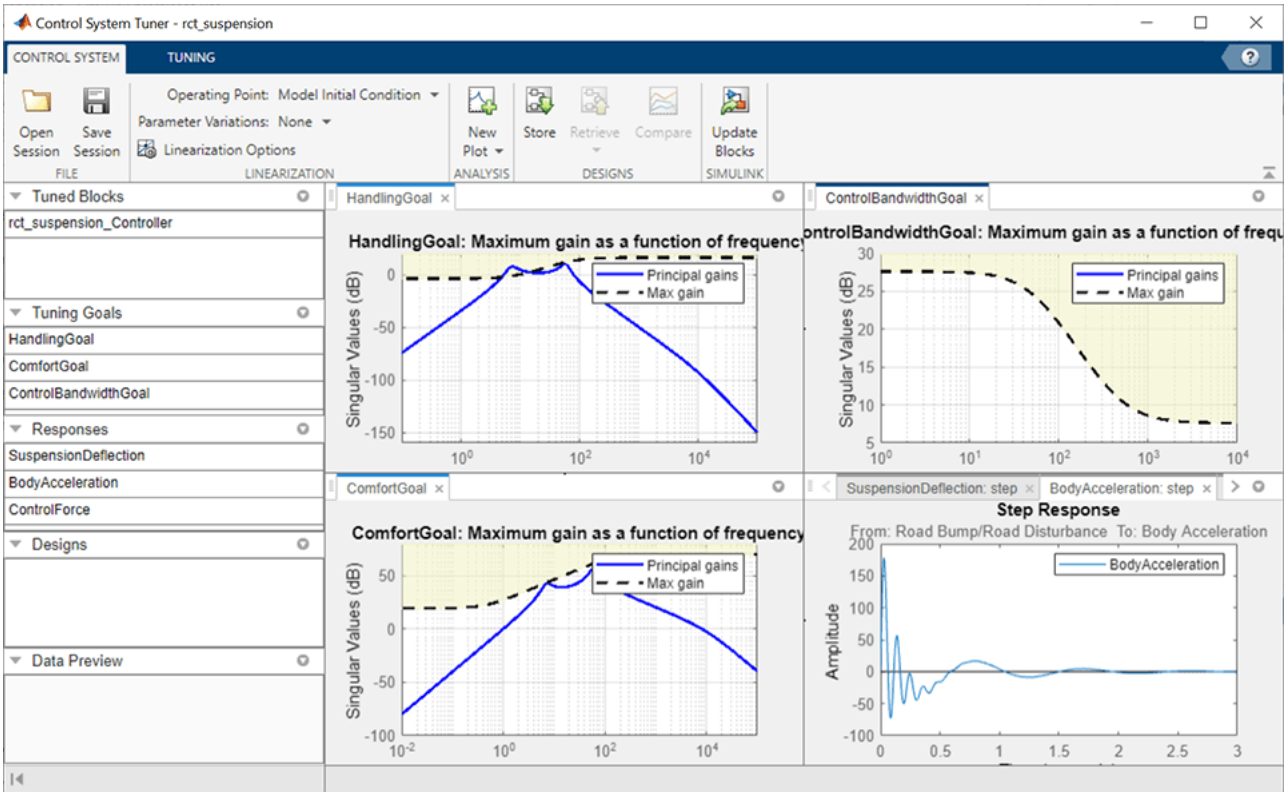


Figure 2: Control System Tuner with Session File.

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.

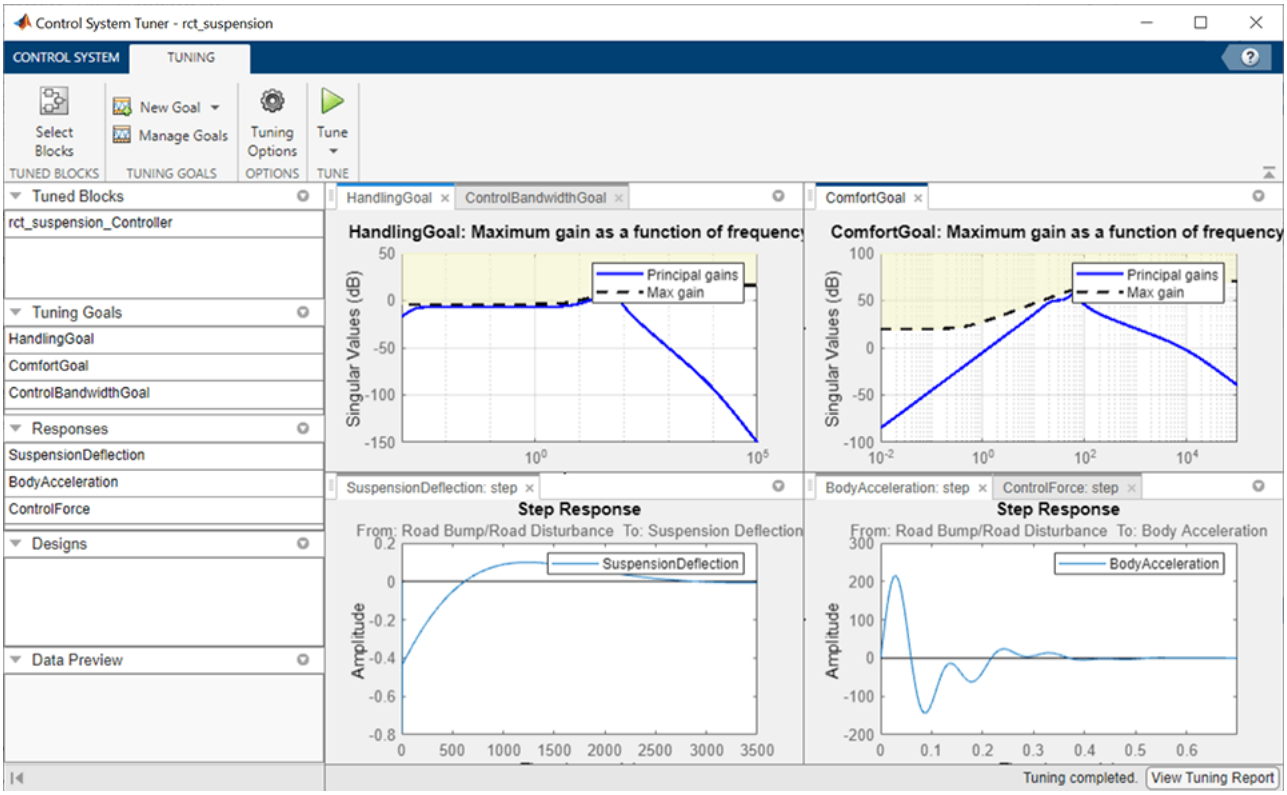


Figure 3: Control System Tuner after tuning.

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.

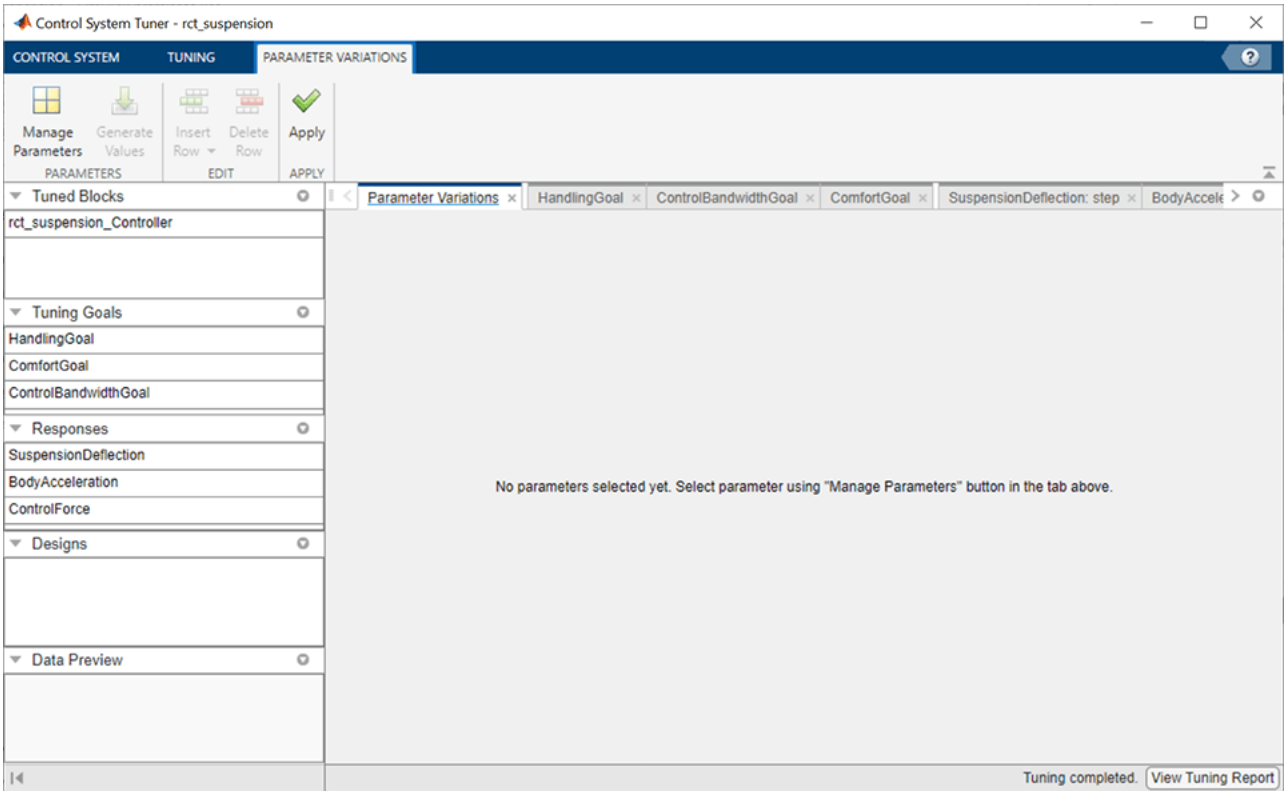


Figure 4: Defining parameter variations.

On the **Parameter Variations** tab, click **Manage Parameters**. In the Select model variables dialog box, select `Mb`.

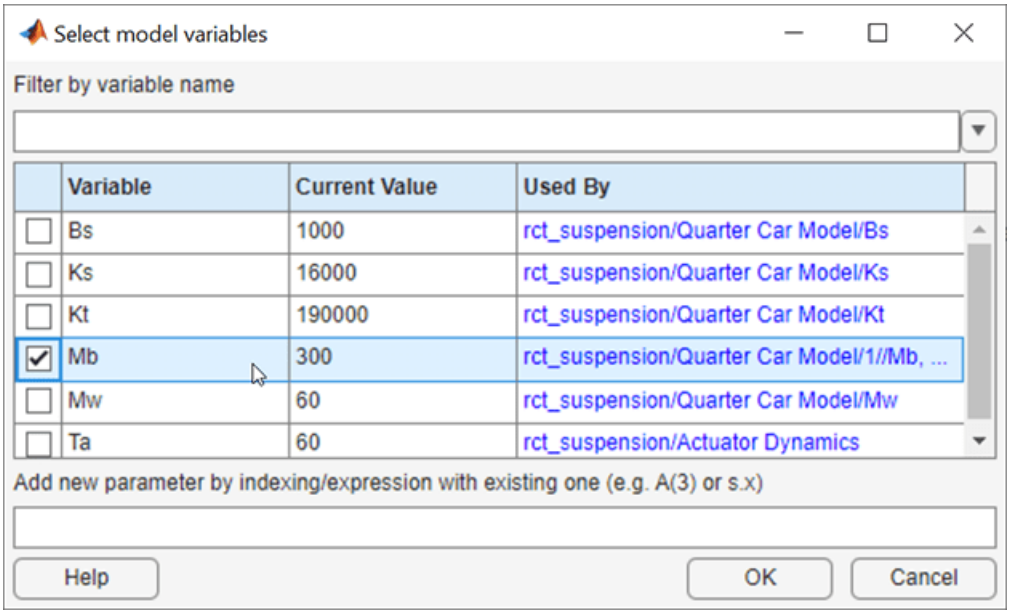


Figure 5: Select a parameter to vary from the model.

Now, the parameter `Mb` is added with default values in the parameter variations table.

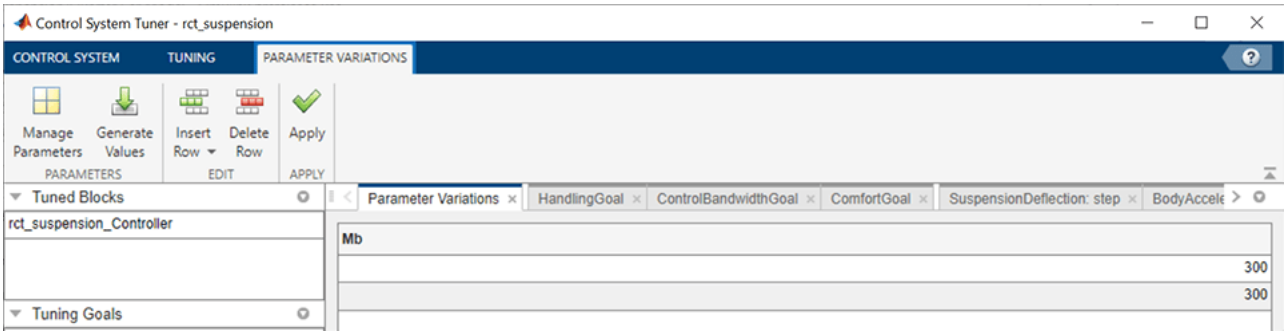


Figure 6: Parameter variations table with default values.

To generate variations quickly, click **Generate Values**. In the Generate Parameter Values dialog box, define values 100, 200, 300 for `Mb`, and click `Overwrite`.

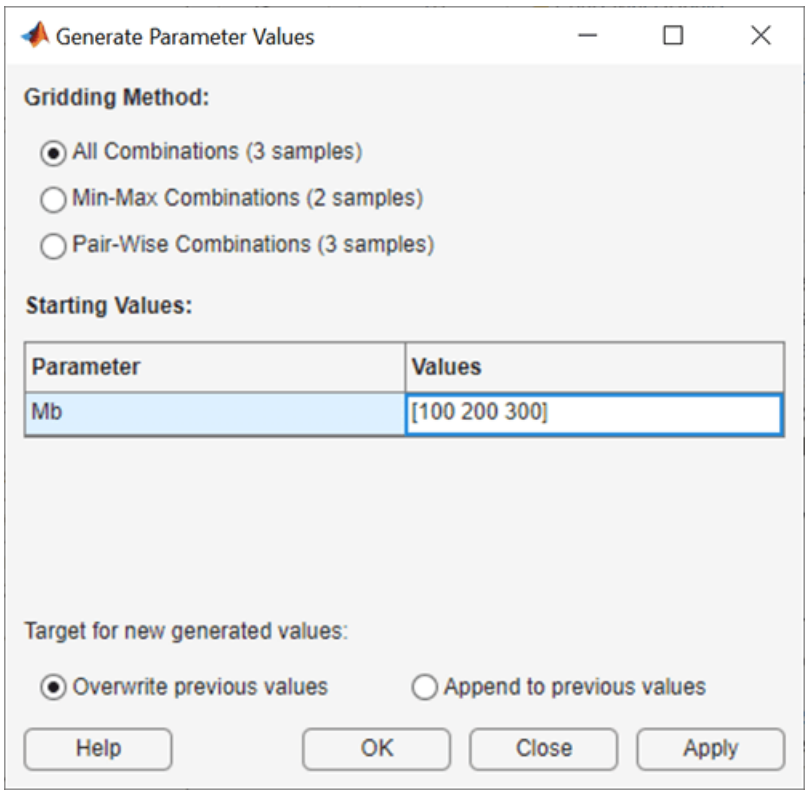


Figure 7: Generate values window.

All values are populated in the parameter variations table. To set the parameter variations to **Control System Tuner**, click **Apply**.

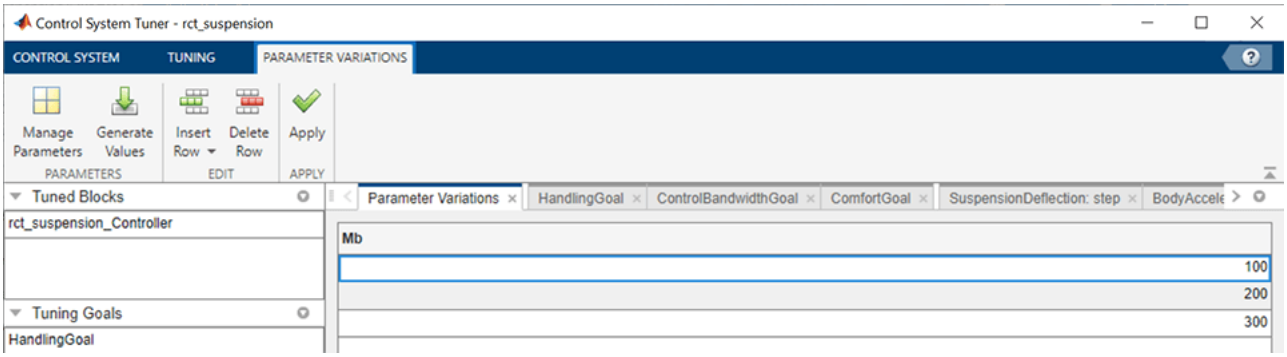


Figure 8: Parameter variations table with updated values.

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.

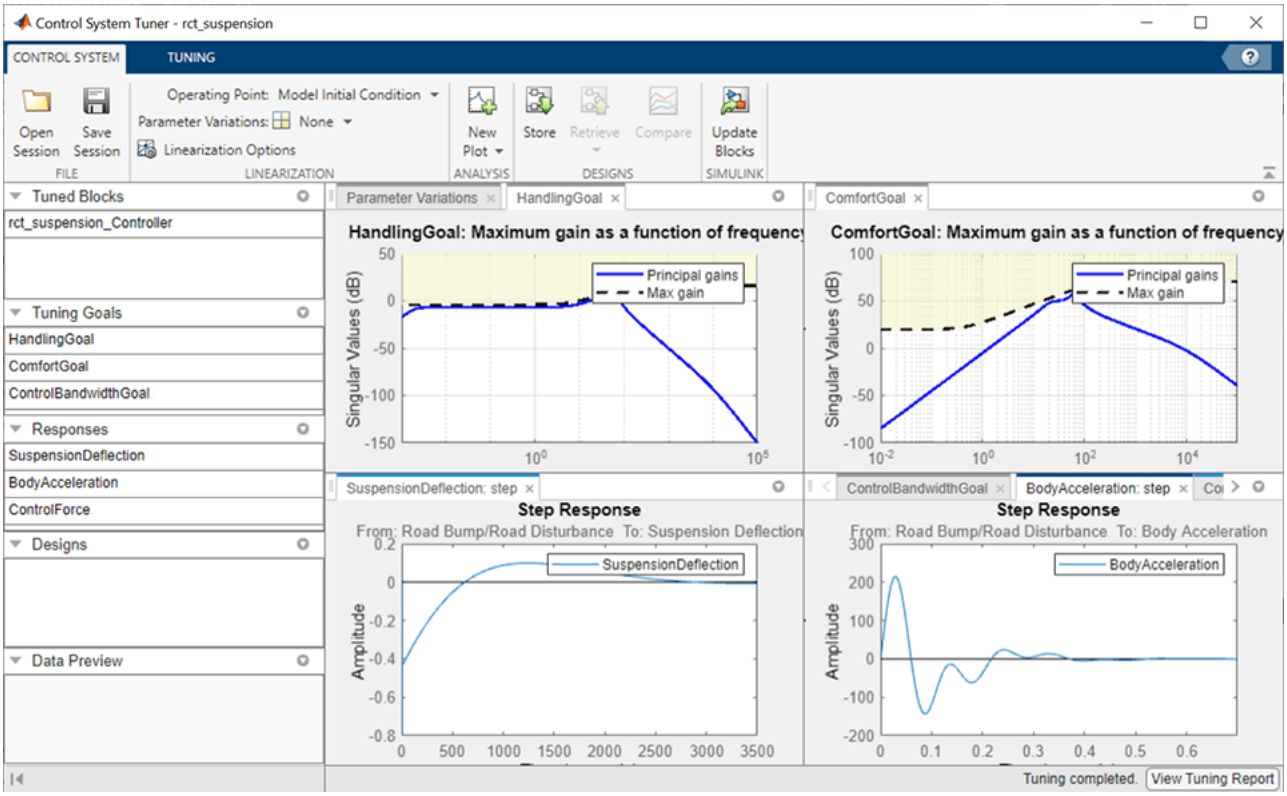


Figure 9: Control System Tuner with multiple parameter variations.

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.

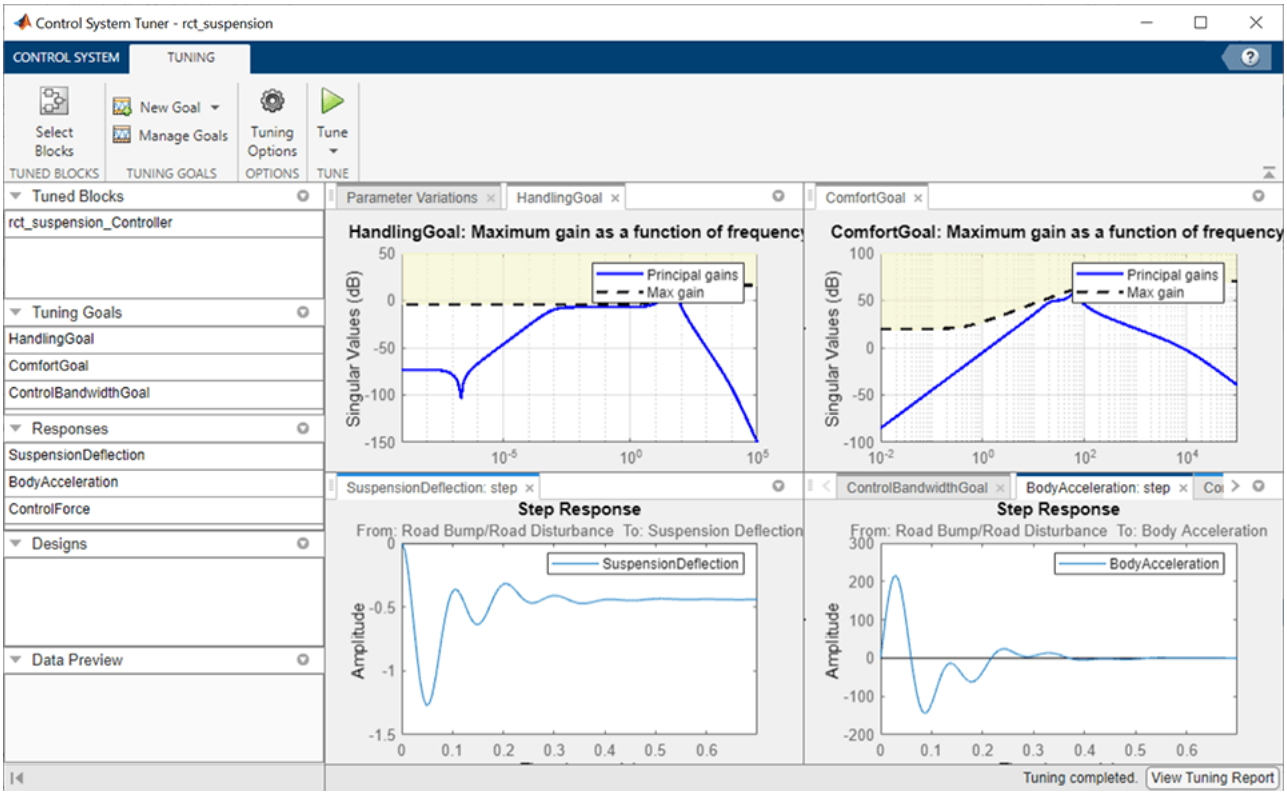


Figure 10: Control System Tuner with multiple parameter variations (Tuned).

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.

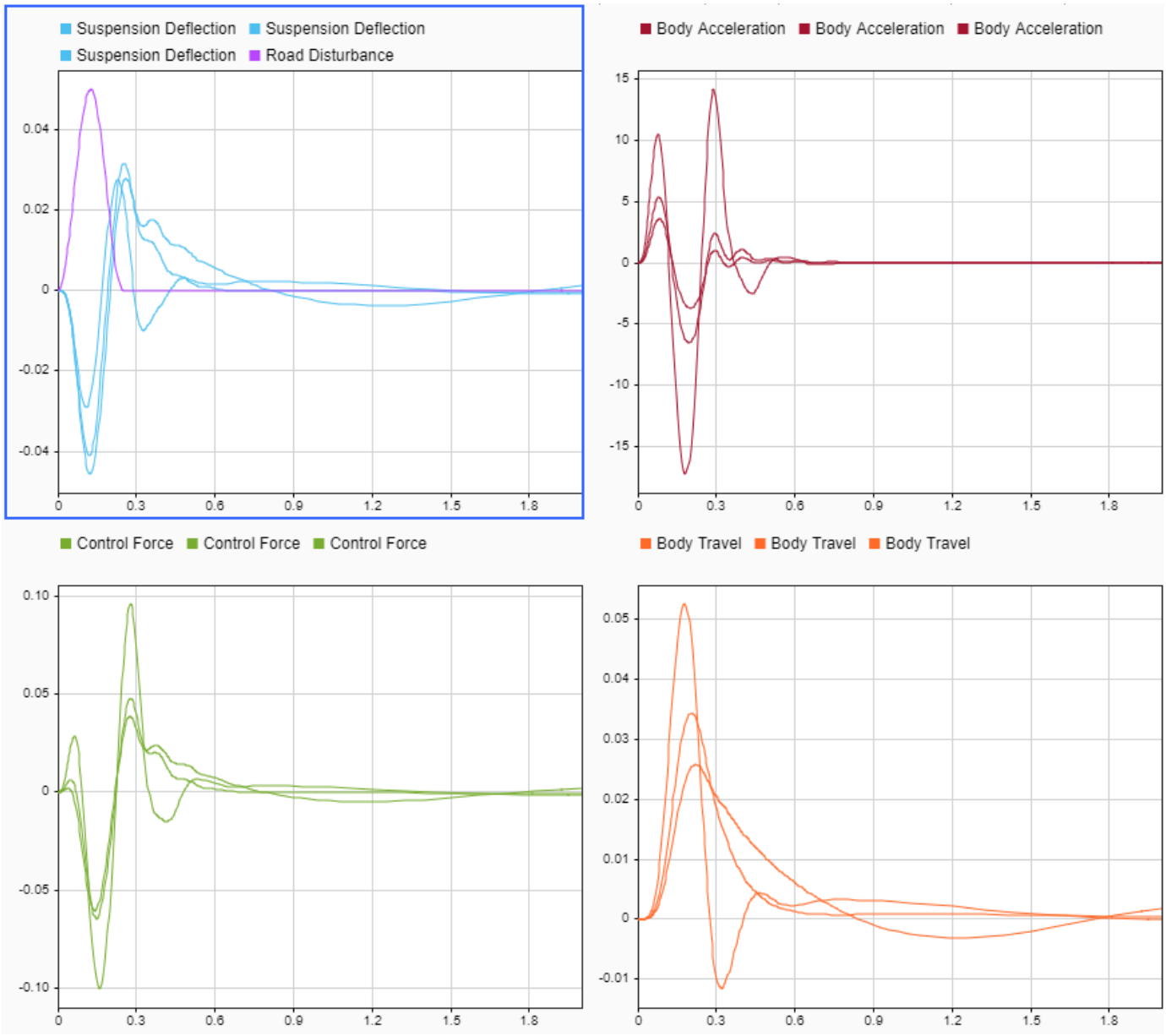


Figure 11: Controller performance on the Simulink model.

See Also

Control System Tuner

More About

- “Create Response Plots in Control System Tuner” on page 14-147

Customization

Preliminaries

- “Terminology” on page 19-2
- “Property and Preferences Hierarchy” on page 19-3
- “Ways to Customize Plots” on page 19-4

Terminology

You can use the Control System Toolbox editors to set properties and preferences in the **Control System Designer**, the **Linear System Analyzer**, and in any response plots that you create from the MATLAB command line.

Properties refer to settings that are specific to an individual response plot. These include the following:

- Axes labels, and limits
- Data units and scales
- Plot styles, such as grids, fonts, and axes foreground colors
- Plot characteristics, such as rise time, peak response, and gain and phase margins

Preferences refer to properties that persist either

- Within a single session for a specific instance of a **Linear System Analyzer** or **Control System Designer**.
- Across Control System Toolbox sessions.

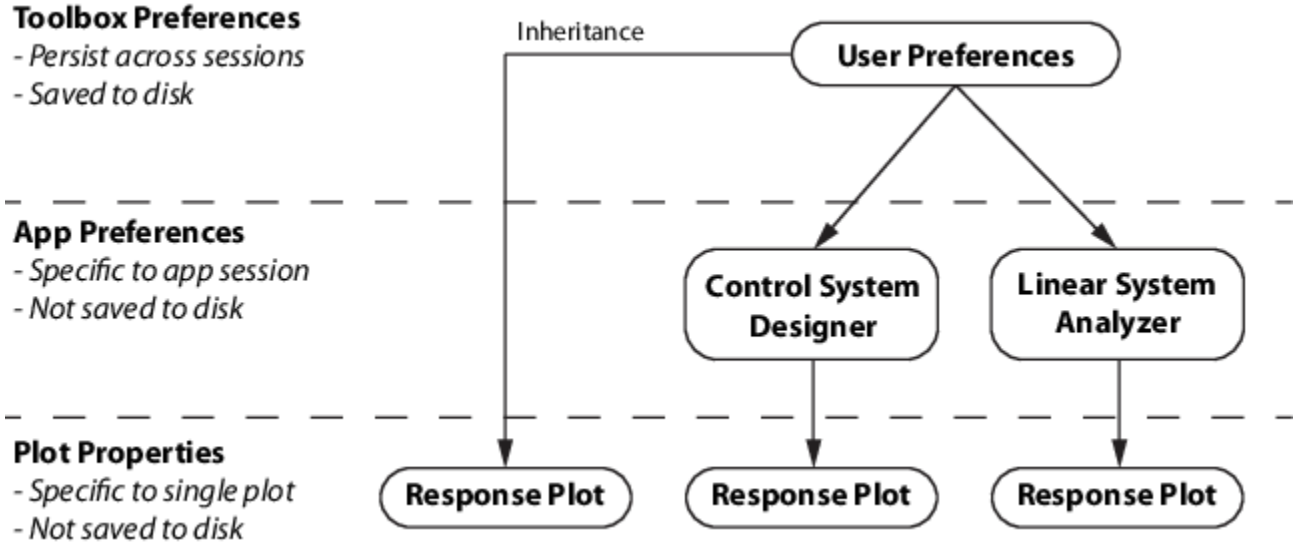
The former are called *app preferences*, the latter *toolbox preferences*.

Property and Preferences Hierarchy

To control the visualization of time-domain and frequency-domain response plots, you can set the following:

- Toolbox Preferences — Apply to all Control System Toolbox response plots.
- App Preferences — Apply to the current app session.
- Plot Properties — Apply to individual response plots.

Although you can set plot properties in any response plot, you can use the Toolbox Preferences Editor to set properties for any response plot that you generate. This figure shows the inheritance hierarchy from toolbox preference to plot properties.



To edit:

- Toolbox preferences:
  - In **Linear System Analyzer**, select **File > Toolbox Preferences**
  - At the MATLAB command line, enter:

    ```
    ctrlpref
    ```
- App preferences:
  - In **Linear System Analyzer**, select **Edit > Linear System Analyzer Preferences**.
  - In **Control System Designer**, on the **Control System tab**, click **Preferences**.
- Plot properties — In any Control System Toolbox response plot, use either of the following:
  - Double-click the plot the plot area.
  - Right-click the plot area, and select **Properties**.

Ways to Customize Plots

You can customize your plots by changing plot properties. For example, you can change the plot units. The following table describes ways that you can customize plots.

To change plot properties of	For more information, see
A single plot, directly from the plot	• "Customize Response Plots Using the Response Plots Property Editor" on page 22-2 and "Customizing Response Plots Using Plot Tools" on page 22-18 for response plots • "Linear System Analyzer Preferences Editor" on page 21-2 for Linear System Analyzer plots
A single plot or many plots, programmatically from the command line	"Customizing Response Plots from the Command Line" on page 22-20
All Control System Toolbox plots (changes apply globally to all plot types and persist from session to session)	"Toolbox Preferences Editor" on page 20-2

Setting Toolbox Preferences

Toolbox Preferences Editor

Overview of the Toolbox Preferences Editor

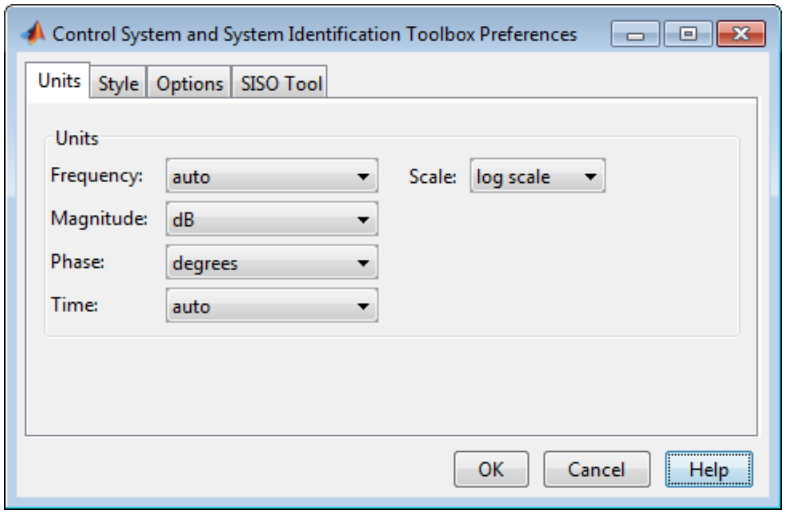
The Toolbox Preferences editor allows you to set plot preferences that will persist from session to session.

Opening the Toolbox Preferences Editor

To open the Toolbox Preferences editor, enter

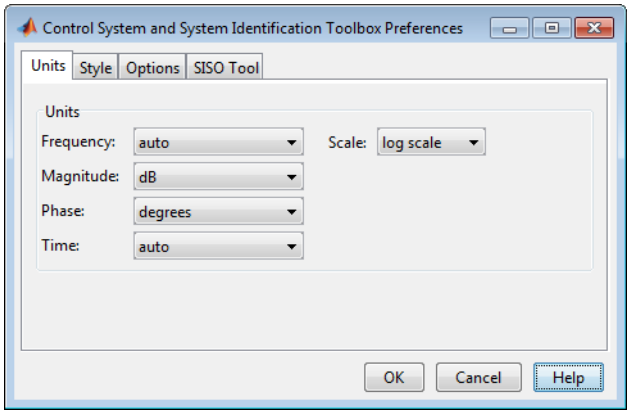
```
ctrlpref
```

at the MATLAB prompt.



In the **Linear System Analyzer** app, you can also open the Toolbox Preferences editor by selecting **File > Toolbox Preferences**.

Units Pane



Use the **Units** pane to set preferences for the following:

- **Frequency**

  The default `auto` option uses `rad/TimeUnit` as the frequency units relative to the system time units, where `TimeUnit` is the system time units specified in the `TimeUnit` property of the system on frequency-domain plots. For multiple systems with different time units, the units of the first system is used.

  For the frequency axis, you can select logarithmic or linear scales.

  Other Frequency Units Options

  - `'Hz'`
  - `'rad/s'`
  - `'rpm'`
  - `'kHz'`
  - `'MHz'`
  - `'GHz'`
  - `'rad/nanosecond'`
  - `'rad/microsecond'`
  - `'rad/millisecond'`
  - `'rad/minute'`
  - `'rad/hour'`
  - `'rad/day'`
  - `'rad/week'`
  - `'rad/month'`
  - `'rad/year'`
  - `'cycles/nanosecond'`
  - `'cycles/microsecond'`
  - `'cycles/millisecond'`
  - `'cycles/hour'`
  - `'cycles/day'`
  - `'cycles/week'`
  - `'cycles/month'`
  - `'cycles/year'`
- **Magnitude** — Decibels (dB) or absolute value (abs)
- **Phase** — Degrees or radians
- **Time**

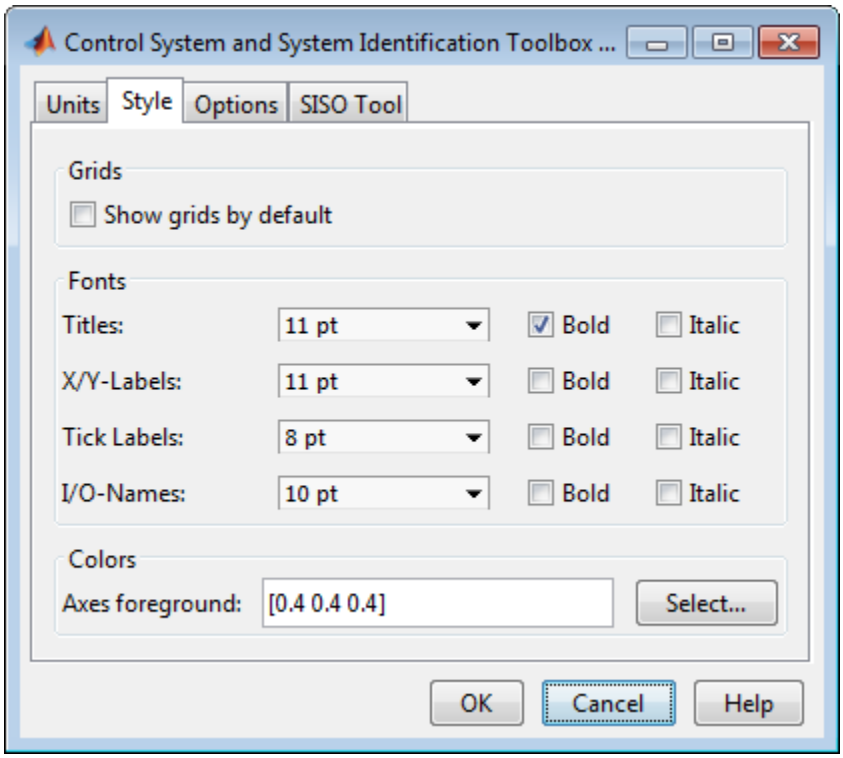
  The default `auto` option uses the time units specified in the `TimeUnit` property of the system on the time- and frequency-domain plots. For multiple systems with different time units, the units of the first system is used.

Other Time Units Options

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

Style Pane

Use the **Style** pane to toggle grid visibility and set font preferences and axes foreground colors for all plots you create. This figure shows the Style pane.



You have the following choices:

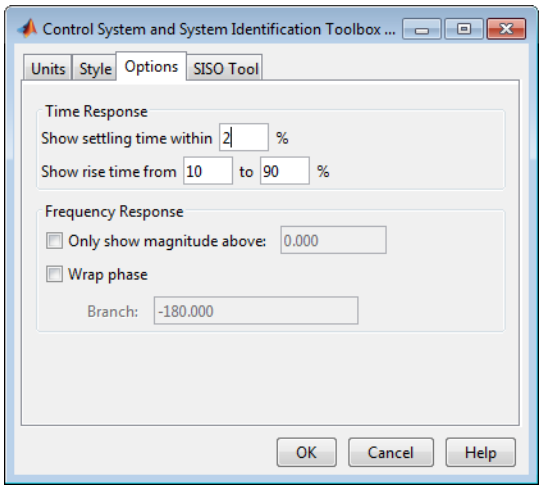
- **Grid** — Activate grids by default in new plots.
- **Fonts** — Set the font size, weight (bold), and angle (italic). Select font sizes from the menus or type any font-size values in the fields.

- **Colors** — Specify the color vector to use for the axes foreground, which includes the X-Y axes, grid lines, and tick labels. Use a three-element vector to represent red, green, and blue (RGB) values. Vector element values can range from 0 to 1.

  If you do not want to specify RGB values numerically, click the **Select** button to open the **Select Colors** dialog box.

Options Pane

The Options pane has selections for time responses and frequency responses. This figure shows the Options pane with default settings.



For time response plots, the following options are available:

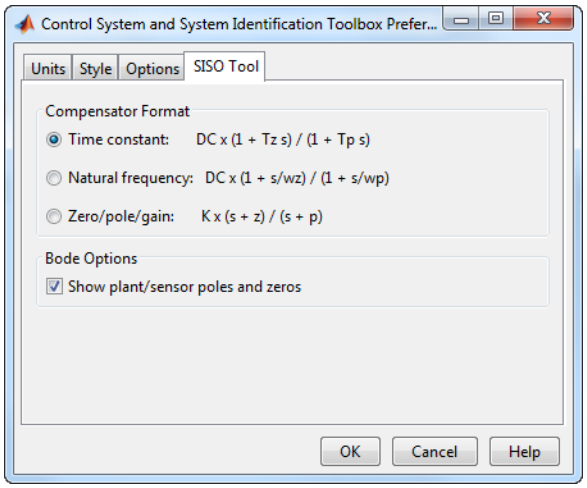
- **Show settling time within** *xx*% — Set the threshold of the settling time calculation to any percentage from 0 to 100%. The default is 2%.
- **Specify rise time from** *xx*% to *yy*%— The standard definition of rise time is the time it takes the signal to go from 10% to 90% of the final value. Specify any percentages you like (from 0% to 100%), provided that the first value is smaller than the second.

For frequency response plots, the following options are available:

- **Only show magnitude above** — Specify a lower limit for magnitude values in response plots so that you can focus on a region of interest.
- **Wrap phase** — Wrap the phase into the interval [–180º,180º). To wrap accumulated phase at a different value, enter the value in the **Branch** field. For example, entering 0 causes the plot to wrap the phase into the interval [0º,360º).

SISO Tool Pane

The SISO Tool pane has settings for **Control System Designer** . This figure shows the SISO Tool pane with default settings.



You can make the following selections:

- **Compensator Format** — Select the time constant, natural frequency, or zero/pole/gain format. The time constant format is a factorization of the compensator transfer function of the form

  $$K \times \frac{(1 + T_{z_1}s)(1 + T_{z_2}s)}{(1 + T_{p_1}s)(1 + T_{p_2}s)} \cdots$$

  where K is compensator DC gain, Tz_1, Tz_2, ..., are the zero time constants, and Tp_1, Tp_2, ..., are the pole time constants.

  The natural frequency format is

  $$K \times \frac{(1 + s/\omega_{z_1})(1 + s/\omega_{z_2})}{(1 + s/\omega_{p_1})(1 + s/\omega_{p_2})} \cdots$$

  where K is compensator DC gain, ω_{z1}, and ω_{z2}, ... and ω_{p1}, ω_{p2}, ..., are the natural frequencies of the zeros and poles, respectively.

  The zero/pole/gain format is

  $$K \times \frac{(s + z_1)(s + z_2)}{(s + p_1)(s + p_2)}$$

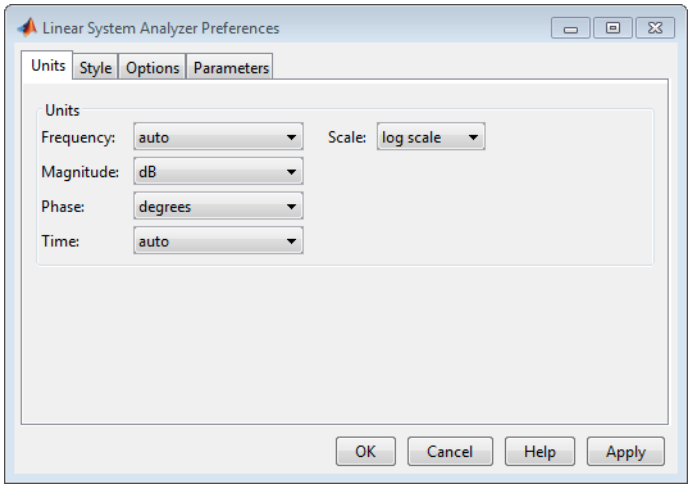
  where K is the overall compensator gain, and z_1, z_2, ... and p_1, p_2, ..., are the zero and pole locations, respectively.
- **Bode Options** — By default, the **Control System Designer** shows the plant and sensor poles and zeros as blue x's and o's, respectively. Clear this box to eliminate the plant's poles and zeros from the Bode plot. Note that the compensator poles and zeros (in red) will still appear.

Setting Tool Preferences

Linear System Analyzer Preferences Editor

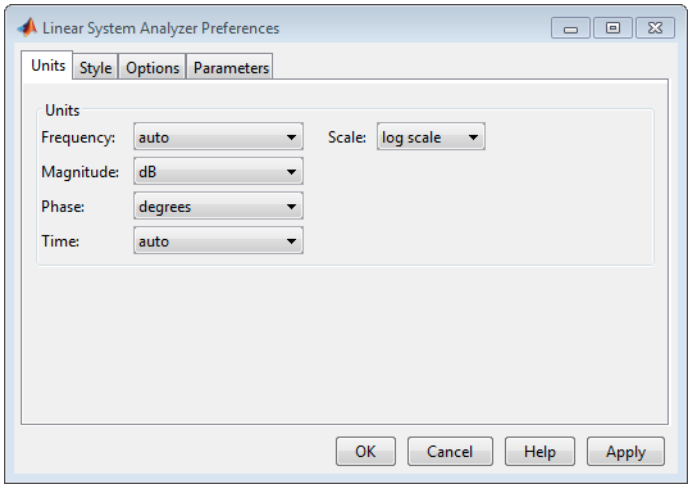
Opening the Linear System Analyzer Preference Editor

In the Linear System Analyzer, select **Edit > Linear System Analyzer Preferences**. The Linear System Analyzer Preferences dialog box let you customize various Linear System Analyzer properties, including units, fonts, and various other characteristics. This figure shows the editor open to its first pane.



- “Units Pane” on page 21-2
- “Style Pane” on page 21-4
- “Options Pane” on page 21-4
- “Parameters Pane” on page 21-5

Units Pane



You can select the following on the **Units** pane:

- **Frequency**

  The default `auto` option uses `rad/TimeUnit` as the frequency units relative to the system time units, where `TimeUnit` is the system time units specified in the `TimeUnit` property of the system on frequency-domain plots. For multiple systems with different time units, the units of the first system is used.

  For the frequency axis, you can select logarithmic or linear scales.

  Other Frequency Units Options

  - `'Hz'`
  - `'rad/s'`
  - `'rpm'`
  - `'kHz'`
  - `'MHz'`
  - `'GHz'`
  - `'rad/nanosecond'`
  - `'rad/microsecond'`
  - `'rad/millisecond'`
  - `'rad/minute'`
  - `'rad/hour'`
  - `'rad/day'`
  - `'rad/week'`
  - `'rad/month'`
  - `'rad/year'`
  - `'cycles/nanosecond'`
  - `'cycles/microsecond'`
  - `'cycles/millisecond'`
  - `'cycles/hour'`
  - `'cycles/day'`
  - `'cycles/week'`
  - `'cycles/month'`
  - `'cycles/year'`
- **Magnitude** — Decibels (dB) or absolute value (abs)
- **Phase** — Degrees or radians
- **Time**

  The default `auto` option uses the time units specified in the `TimeUnit` property of the system on the time- and frequency-domain plots. For multiple systems with different time units, the units of the first system is used.

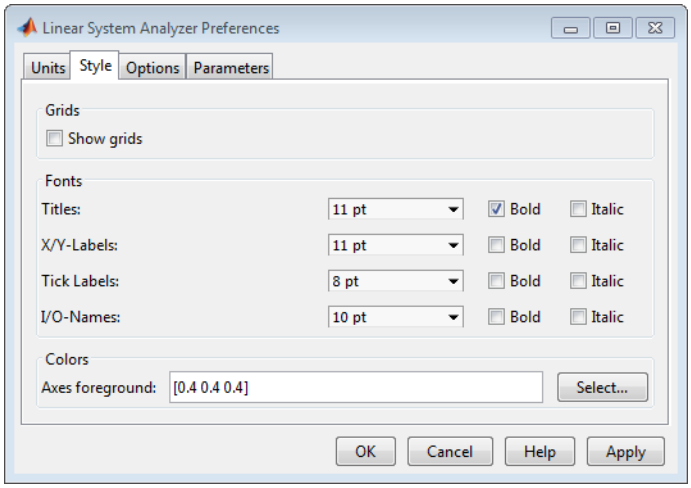
  Other Time Units Options

  - `'nanoseconds'`

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

Style Pane

Use the **Style** pane to toggle grid visibility and set font preferences and axes foreground colors for all plots in the Linear System Analyzer. This figure shows the Style pane.

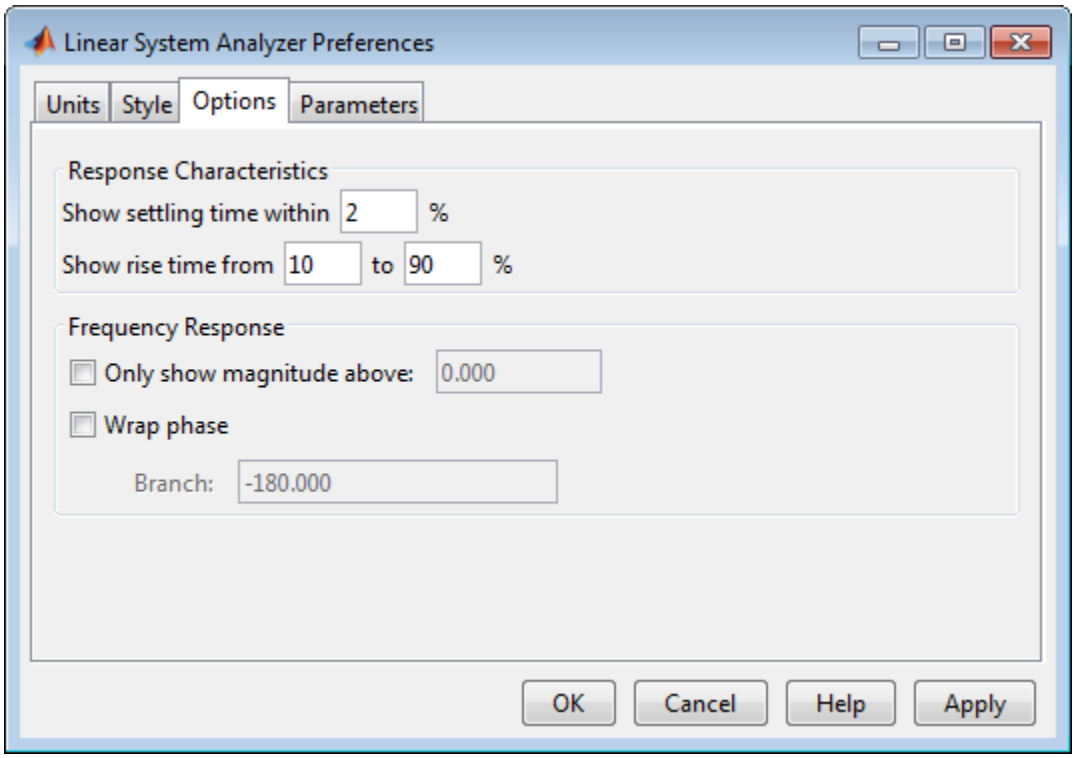


You have the following choices:

- **Grid** — Activate grids for all plots in the Linear System Analyzer
- **Fonts** — Set the font size, weight (bold), and angle (italic). Select font sizes from the menus or type any font-size values in the fields.
- **Colors** — Specify the color vector to use for the axes foreground, which includes the X-Y axes, grid lines, and tick labels. Use a three-element vector to represent red, green, and blue (RGB) values. Vector element values can range from 0 to 1.
- If you do not want to specify the RGB values numerically, press the **Select** button to open the **Select Colors** window.

Options Pane

The Options pane has selections for time responses and frequency responses.



For time response plots, the following options are available:

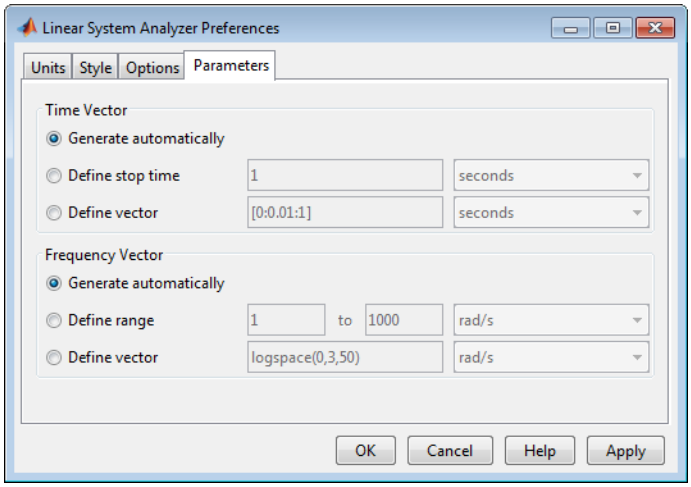
- **Show settling time within** *xx*% — Set the threshold of the settling time calculation to any percentage from 0 to 100%. The default is 2%.
- **Specify rise time from** *xx*% to *yy*%— The standard definition of rise time is the time it takes the signal to go from 10% to 90% of the final value. Specify any percentages you like (from 0% to 100%), provided that the first value is smaller than the second.

For frequency response plots, the following options are available:

- **Only show magnitude above** — Specify a lower limit for magnitude values in response plots so that you can focus on a region of interest.
- **Wrap phase** — Wrap the phase into the interval [–180º,180º). To wrap accumulated phase at a different value, enter the value in the **Branch** field. For example, entering 0 causes the plot to wrap the phase into the interval [0º,360º).

Parameters Pane

Use the **Parameters** pane, shown below, to specify input vectors for time and frequency simulation.



The defaults are to generate time and frequency vectors for your plots automatically. You can, however, override the defaults as follows:

- **Time Vector**:
  - Define stop time — Specify the final time value for your simulation
  - Define vector — Specify the time vector manually using equal-sized time steps
- **Frequency Vector**:
  - Define range — Specify the bandwidth of your response. Whether it's in rad/sec or Hz depends on the selection you made in the Units pane.
  - Define vector — Specify the vector for your frequency values. Any real, positive, strictly monotonically increasing vector is valid.

See Also

Linear System Analyzer

More About

- “Linear System Analyzer Overview” on page 26-2

Customizing Response Plot Properties

- “Customize Response Plots Using the Response Plots Property Editor” on page 22-2
- “Customizing Response Plots Using Plot Tools” on page 22-18
- “Customizing Response Plots from the Command Line” on page 22-20
- “Build GUI With Interactive Response-Plot Updates” on page 22-37

Customize Response Plots Using the Response Plots Property Editor

Opening the Property Editor

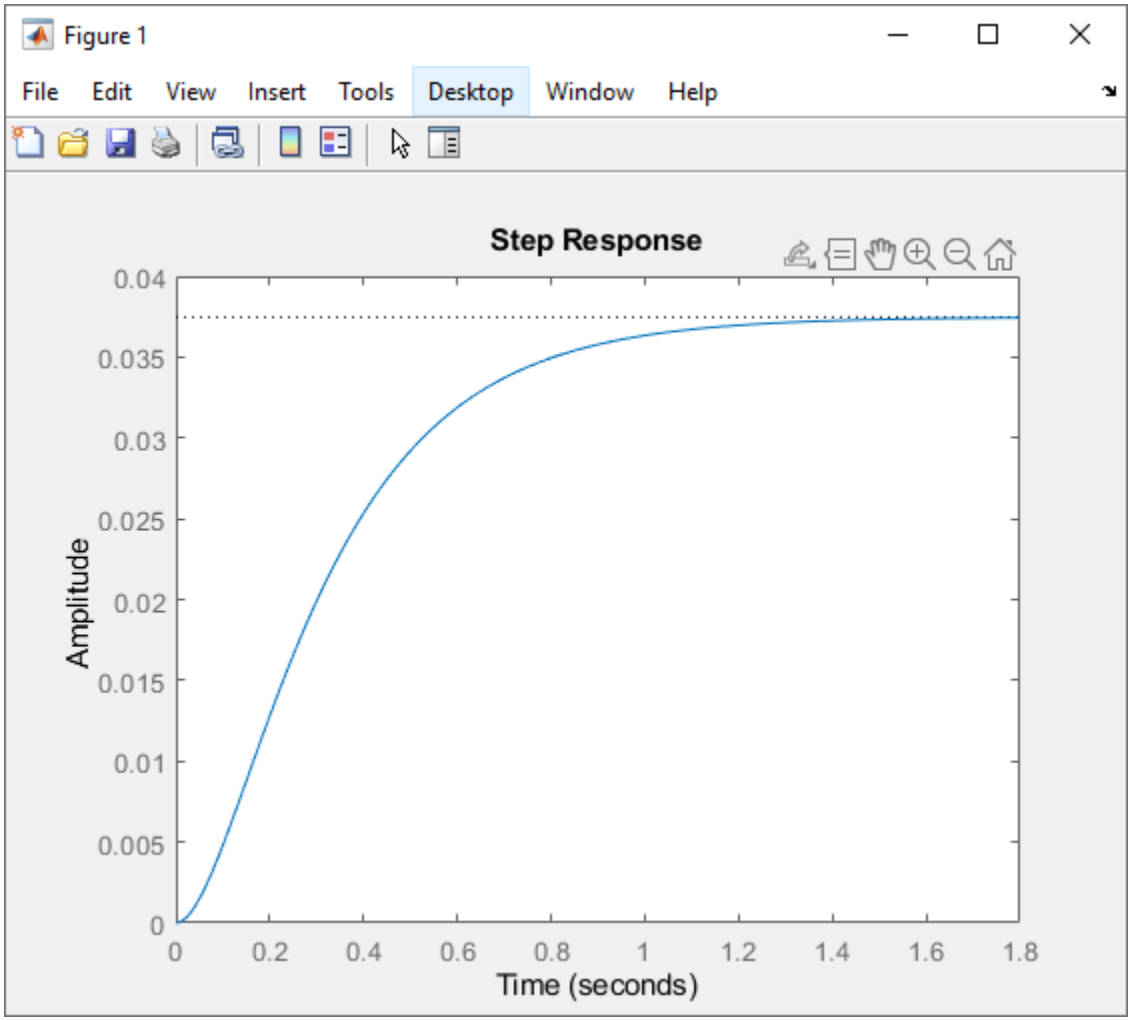
After you create a response plot, there are two ways to open the Property Editor:

- Double-click in the plot region.
- Right-click the plot, and select **Properties** from the context menu.

Before looking at the Property Editor, open a step response plot using these commands.

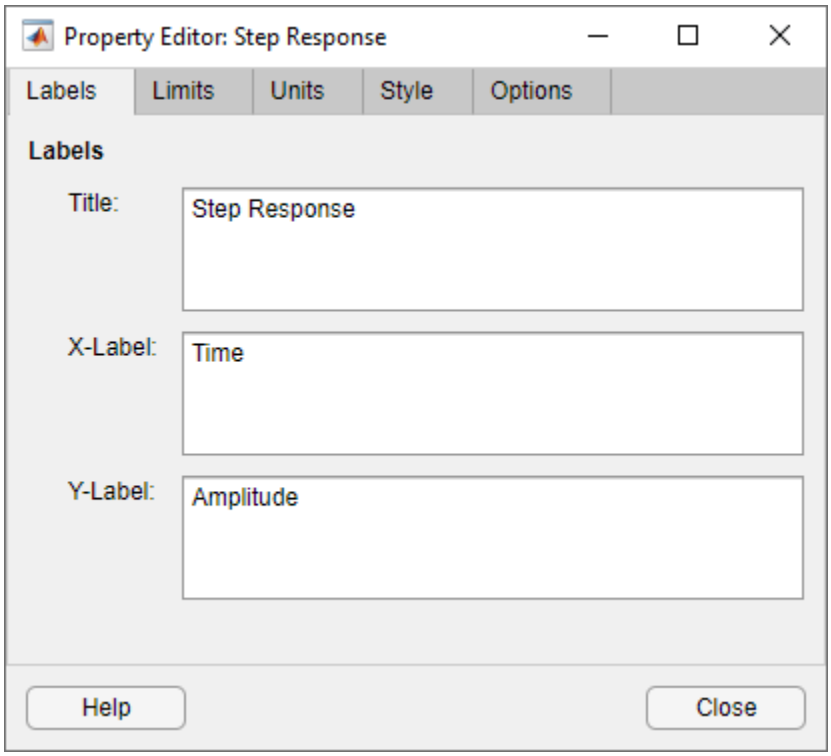
```
load ltiexamples
step(sys_dc)
```

This creates a step plot. Right-click the plot, and select **Properties** from the context menu. When you open the Property Editor, squares appear around the step response plot.



Overview of Response Plots Property Editor

The appearance of the Property Editor dialog box depends on the type of response plot. This figure shows the Property Editor dialog box for a step response.



The Property Editor for Step Response

In general, you can change the following properties of response plots. Only the **Labels** and **Limits** panes are available when using the Property Editor with Simulink Design Optimization software.

- Titles and X- and Y-labels in the **Labels** pane.
- Numerical ranges of the X and Y axes in the **Limits** pane.
- Units where applicable (e.g., rad/s to Hertz) in the **Units** pane.

  If you cannot customize units, the Property Editor displays that no units are available for the selected plot.
- Styles in the **Styles** pane.

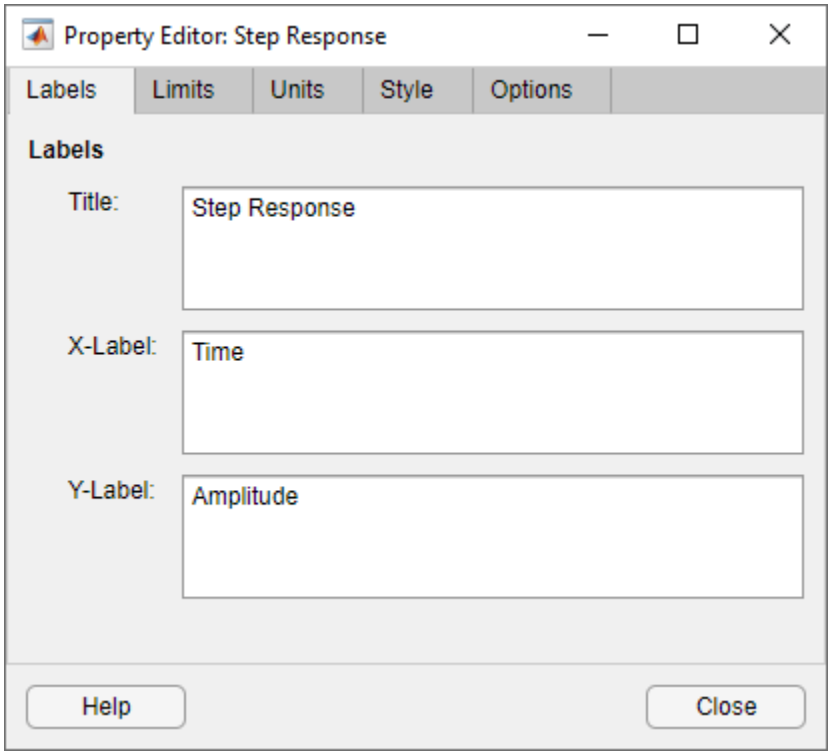
  You can show a grid, adjust font properties, such as font size, bold, and italics, and change the axes foreground color
- Change options where applicable in the **Options** pane.

  These include peak response, settling time, phase and gain margins, etc. Plot options change with each plot response type. The Property Editor displays only the options that make sense for the selected response plot. For example, phase and gain margins are not available for step responses.

As you make changes in the Property Editor, they display immediately in the response plot. Conversely, if you make changes in a plot using right-click menus, the Property Editor for that plot automatically updates. The Property Editor and its associated plot are dynamically linked.

Labels Pane

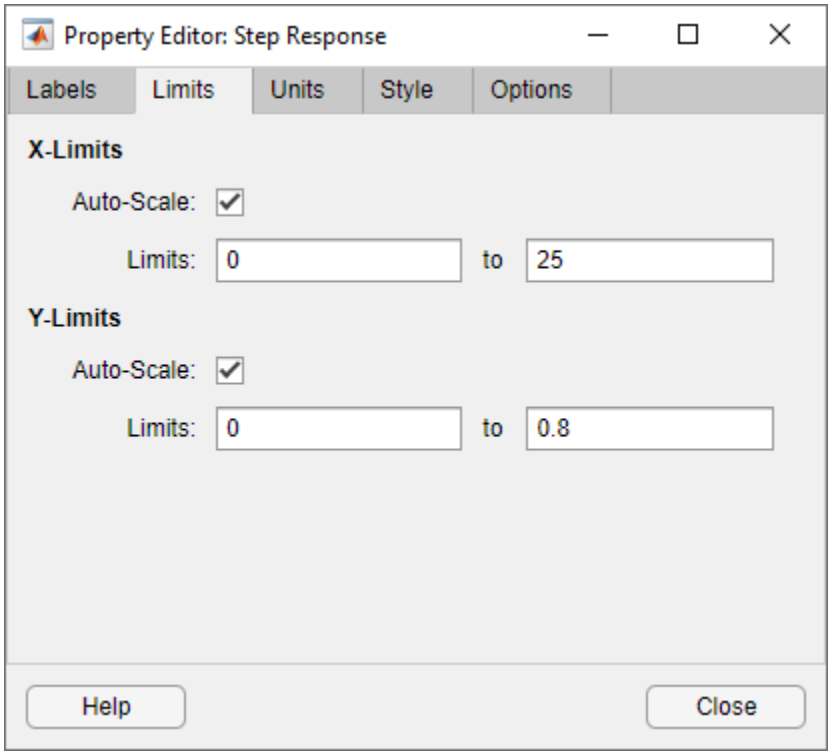
To specify new text for plot titles and axis labels, type the new names in the field next to the label you want to change. The label changes immediately as you type, so you can see how the new text looks as you are typing.



Limits Pane

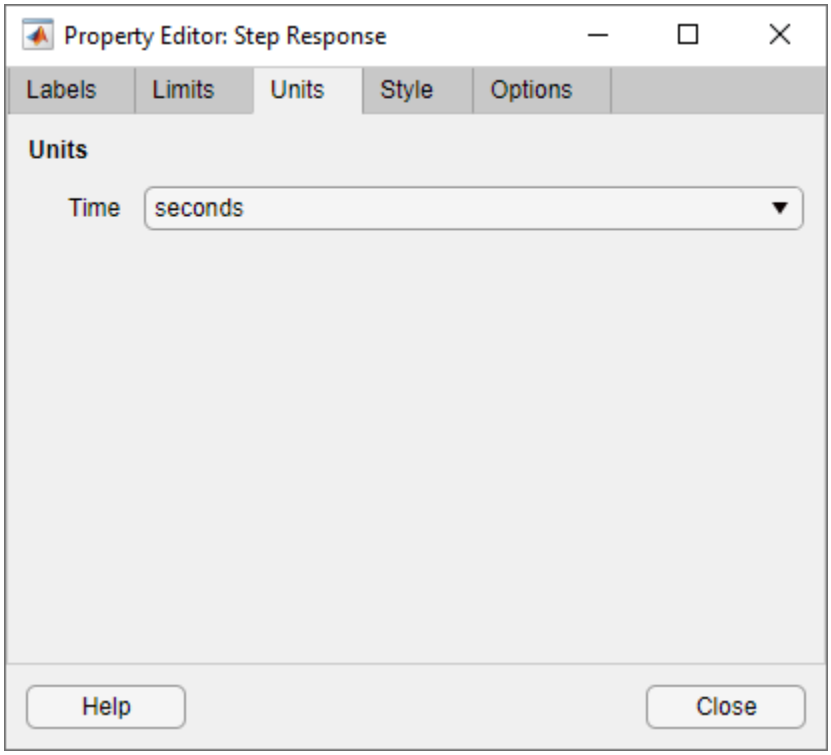
Default values for the axes limits make sure that the maximum and minimum *x* and *y* values are displayed. If you want to override the default settings, change the values in the Limits fields. The **Auto-Scale** box automatically clears if you click a different field. The new limits appear immediately in the response plot.

To re-establish the default values, select the **Auto-Scale** box again.



Units Pane

You can use the **Units** pane to change units in your response plot. The contents of this pane depend on the response plot associated with the editor. Use the menus to toggle between units.


Property Editor: Step Response
Labels
Limits
Units
Style
Options
Units
Time
seconds
Help
Close

Optional Unit Conversions for Response Plots

Response Plot	Unit Conversions
Bode and Bode Magnitude	• Frequency By default, shows `rad/TimeUnit` where `TimeUnit` is the system time units specified in the `TimeUnit` property of the input system. **Frequency Units Options** • `'Hz'` • `'rad/s'` • `'rpm'` • `'kHz'` • `'MHz'` • `'GHz'` • `'rad/nanosecond'` • `'rad/microsecond'` • `'rad/millisecond'` • `'rad/minute'` • `'rad/hour'` • `'rad/day'` • `'rad/week'` • `'rad/month'` • `'rad/year'` • `'cycles/nanosecond'` • `'cycles/microsecond'` • `'cycles/millisecond'` • `'cycles/hour'` • `'cycles/day'` • `'cycles/week'` • `'cycles/month'` • `'cycles/year'` • Frequency scale is logarithmic or linear. • Magnitude in decibels (dB) or the absolute value • Phase in degrees or radians

Response Plot	Unit Conversions
Impulse	• Time By default, shows the system time units specified in the `TimeUnit` property of the input system. **Time Units Options** • `'nanoseconds'` • `'microseconds'` • `'milliseconds'` • `'seconds'` • `'minutes'` • `'hours'` • `'days'` • `'weeks'` • `'months'` • `'years'`

Response Plot	Unit Conversions
Nichols Chart	• Frequency By default, shows `rad/TimeUnit` where `TimeUnit` is the system time units specified in the `TimeUnit` property of the input system. **Frequency Units Options** • `'Hz'` • `'rad/s'` • `'rpm'` • `'kHz'` • `'MHz'` • `'GHz'` • `'rad/nanosecond'` • `'rad/microsecond'` • `'rad/millisecond'` • `'rad/minute'` • `'rad/hour'` • `'rad/day'` • `'rad/week'` • `'rad/month'` • `'rad/year'` • `'cycles/nanosecond'` • `'cycles/microsecond'` • `'cycles/millisecond'` • `'cycles/hour'` • `'cycles/day'` • `'cycles/week'` • `'cycles/month'` • `'cycles/year'` • Phase in degrees or radians

Response Plot	Unit Conversions
Nyquist Diagram	• Frequency By default, shows `rad/TimeUnit` where `TimeUnit` is the system time units specified in the `TimeUnit` property of the input system. **Frequency Units Options** • `'Hz'` • `'rad/s'` • `'rpm'` • `'kHz'` • `'MHz'` • `'GHz'` • `'rad/nanosecond'` • `'rad/microsecond'` • `'rad/millisecond'` • `'rad/minute'` • `'rad/hour'` • `'rad/day'` • `'rad/week'` • `'rad/month'` • `'rad/year'` • `'cycles/nanosecond'` • `'cycles/microsecond'` • `'cycles/millisecond'` • `'cycles/hour'` • `'cycles/day'` • `'cycles/week'` • `'cycles/month'` • `'cycles/year'`

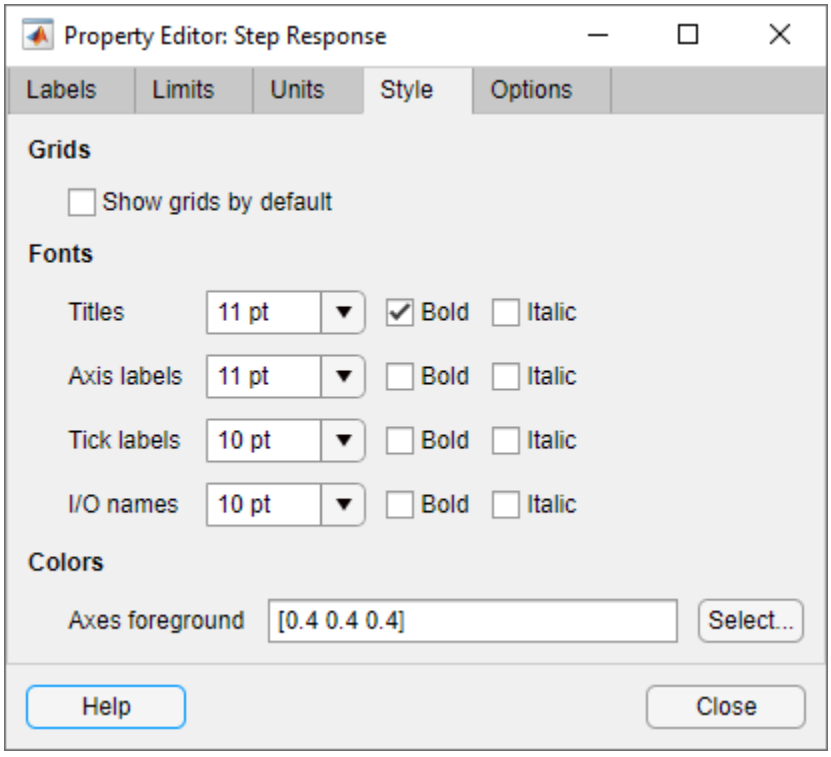
Response Plot	Unit Conversions
Pole/Zero Map	• Time By default, shows the system time units specified in the `TimeUnit` property of the input system. **Time Units Options** • `'nanoseconds'` • `'microseconds'` • `'milliseconds'` • `'seconds'` • `'minutes'` • `'hours'` • `'days'` • `'weeks'` • `'months'` • `'years'` • Frequency By default, shows `rad/TimeUnit` where `TimeUnit` is the system time units specified in the `TimeUnit` property of the input system. **Frequency Units Options** • `'Hz'` • `'rad/s'` • `'rpm'` • `'kHz'` • `'MHz'` • `'GHz'` • `'rad/nanosecond'` • `'rad/microsecond'` • `'rad/millisecond'` • `'rad/minute'` • `'rad/hour'` • `'rad/day'` • `'rad/week'` • `'rad/month'` • `'rad/year'` • `'cycles/nanosecond'` • `'cycles/microsecond'`

Response Plot	Unit Conversions
	• 'cycles/millisecond' • 'cycles/hour' • 'cycles/day' • 'cycles/week' • 'cycles/month' • 'cycles/year'
Singular Values	• Frequency By default, shows rad/TimeUnit where TimeUnit is the system time units specified in the TimeUnit property of the input system. **Frequency Units Options** • 'Hz' • 'rad/s' • 'rpm' • 'kHz' • 'MHz' • 'GHz' • 'rad/nanosecond' • 'rad/microsecond' • 'rad/millisecond' • 'rad/minute' • 'rad/hour' • 'rad/day' • 'rad/week' • 'rad/month' • 'rad/year' • 'cycles/nanosecond' • 'cycles/microsecond' • 'cycles/millisecond' • 'cycles/hour' • 'cycles/day' • 'cycles/week' • 'cycles/month' • 'cycles/year' • Frequency scale is logarithmic or linear. • Magnitude in decibels or the absolute value using logarithmic or linear scale

Response Plot	Unit Conversions
Step	• Time By default, shows the system time units specified in the `TimeUnit` property of the input system. **Time Units Options** • `'nanoseconds'` • `'microseconds'` • `'milliseconds'` • `'seconds'` • `'minutes'` • `'hours'` • `'days'` • `'weeks'` • `'months'` • `'years'`

Style Pane

Use the Style pane to toggle grid visibility and set font preferences and axes foreground colors for response plots.



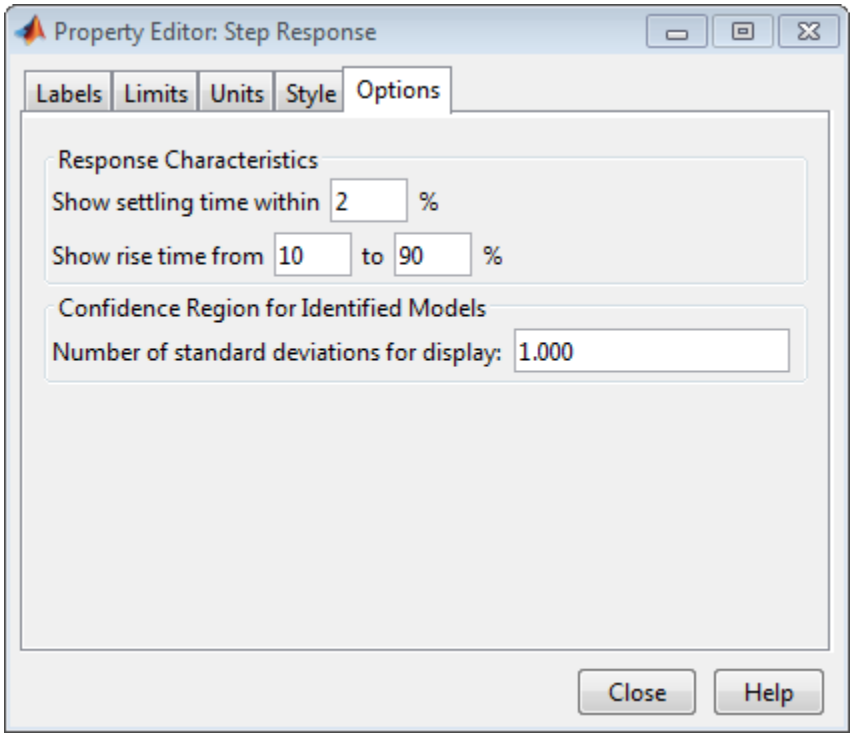
You have the following choices:

- **Grid** — Activate grids by default in new plots.
- **Fonts** — Set the font size, weight (bold), and angle (italic) for fonts used in response plot titles, X/Y-labels, tick labels, and I/O names. Select font sizes from the menus or type any font-size values in the fields.
- **Colors** — Specify the color vector to use for the axes foreground, which includes the X-Y axes, grid lines, and tick labels. Use a three-element vector to represent red, green, and blue (RGB) values. Vector element values can range from 0 to 1.

  If you do not want to specify RGB values numerically, click the **Select** button to open the Select Color dialog box.

Options Pane

The **Options** pane enables you to customize response characteristics for plots. Each response plot has its own set of characteristics and optional settings. When you change the value in a field, press **Enter** on your keyboard to update the response plot.



Response Characteristic Options for Response Plots

Plot	Customizable Feature
Bode Diagram and Bode Magnitude	• **Magnitude Response** Select lower magnitude limit. • **Phase Response** By default, plots display exact phase. Check **Wrap phase** to wrap the phase into the interval [–180º,180º). To wrap accumulated phase at a different value, enter the value in the **Branch** field. For example, entering 0 causes the plot to wrap the phase into the interval [0º,360º). Check **Adjust phase offsets** to keep phase close to a particular value, within a range of 0º–360º, at a given frequency. • **Confidence Region for Identified Models** This option is available with System Identification Toolbox. Specify number of standard deviations for plotting the response confidence region. To see the confidence region, right-click the plot, and select **Characteristics > Confidence Region**.
Impulse	• **Response Characteristics** Show settling time within *xx*% (specify the percentage). • **Confidence Region for Identified Models** These options are available with System Identification Toolbox. **Display using zero mean interval**: For an identified model with impulse response y and standard deviation Δy, plot the uncertainty $\pm\Delta y$ as a function of time (default). If cleared, $y\pm\Delta y$ as a function of time is plotted. **Number of standard deviations for display**: Specify number of standard deviations for plotting the uncertainty. To see the confidence interval, right-click the plot, and select **Characteristics > Confidence Region**.

Plot	Customizable Feature
Nichols Chart	• **Magnitude Response** Select lower magnitude limit. • **Phase Response** By default, plots display exact phase. Check **Wrap phase** to wrap the phase into the interval [–180º,180º). To wrap accumulated phase at a different value, enter the value in the **Branch** field. For example, entering 0 causes the plot to wrap the phase into the interval [0º,360º). Check **Adjust phase offsets** to keep phase close to a particular value, within a range of 0º–360º, at a given frequency.
Nyquist Diagram	• **Confidence Region for Identified Models** These options are available with System Identification Toolbox. **Number of standard deviations for display**: Specify number of standard deviations for plotting the confidence ellipses. **Display spacing**: Specify the frequency spacing of confidence ellipses. The default is 5, which means that the confidence ellipses are shown at every fifth frequency sample. To see the confidence ellipses, right-click the plot, and select **Characteristics > Confidence Region**.
Pole/Zero Map	• **Confidence Region for Identified Models** This option is available with System Identification Toolbox. Specify number of standard deviations for displaying the confidence region characteristic. To see the confidence region, right-click the plot, and select **Characteristics > Confidence Region**.
Singular Values	None

Plot	Customizable Feature
Step	• **Response Characteristics** Show settling time within *xx*% (specify the percentage). Show rise time from *xx* to *yy*% (specify the percentages) • **Confidence Region for Identified Models** This option is available with System Identification Toolbox. Specify number of standard deviations for plotting the response confidence region. To see the confidence region, right-click the plot, and select **Characteristics > Confidence Region**.

Editing Subplots Using the Property Editor

If you create more than one plot in a single figure window, you can edit each plot individually. For example, the following code creates a figure with two plots, a step and an impulse response with two randomly selected systems:

```
subplot(2,1,1)
step(rss(2,1))
subplot(2,1,2)
impulse(rss(1,1))
```

After the figure window appears, double-click in the upper (step response) plot to activate the Property Editor. A set of small squares appear around the step response, indicating that it is the active plot for the editor. To switch to the lower (impulse response) plot, click once in the impulse response plot region. The set of squares switches to the impulse response, and the Property Editor updates as well.

See Also

More About

- “Ways to Customize Plots” on page 19-4

Customizing Response Plots Using Plot Tools

Properties You Can Customize Using Plot Tools

The following table shows the plot properties you can customize using plot tools.

For...	You can customize the following properties:
Responses	• System name • Line color • Line style • Line width • Marker type For SISO systems, these changes apply to a single plot line or an array of plot lines representing the system on one axis. For MIMO systems, these changes apply to all of the plotted lines representing the system on multiple axes.
Plot axes	• Title • X-label • Y-label
Figures	• Figure name • Colormap • Figure color

Note To make other changes to response plots, see “Customize Response Plots Using the Response Plots Property Editor” on page 22-2 and “Customizing Response Plots from the Command Line” on page 22-20.

Opening and Working with Plot Tools

For information about how to open and work with Plot Tools in the Property Inspector, see `Property Inspector`.

Example of Changing Line Color Using Plot Tools

To change the line color of a MIMO system plot:

1. Create a step response plot of a MIMO system by typing

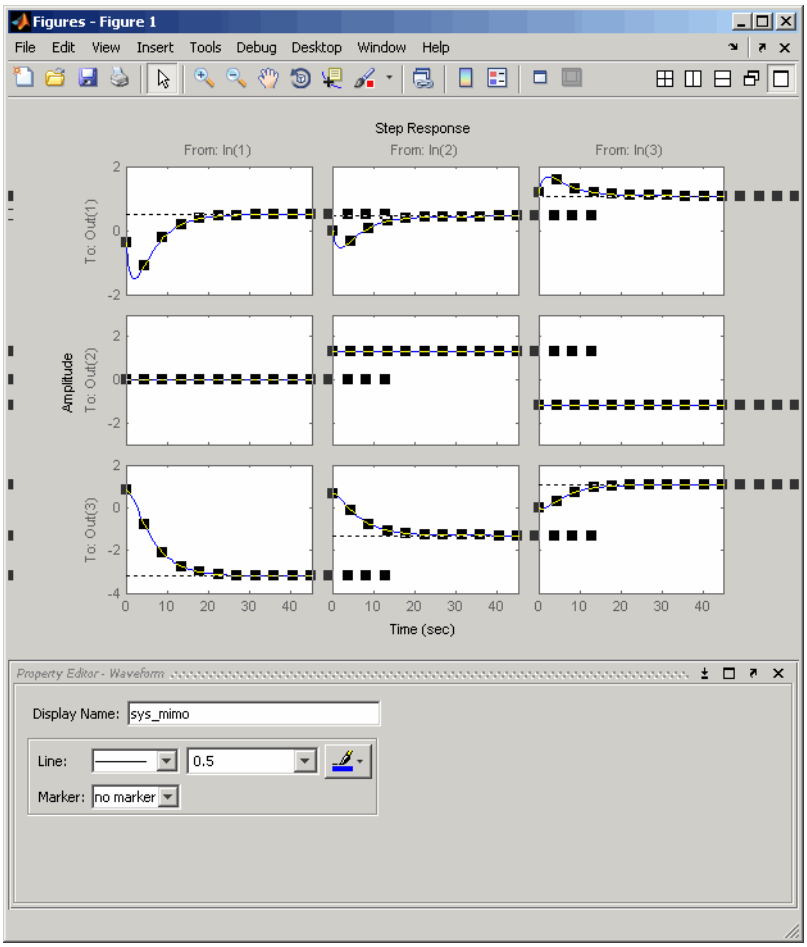
   ```
   sys_mimo=rss(3,3,3);
   stepplot(sys_mimo)
   ```

2. In the figure window, select **View** > **Property Editor**.

   This action opens the Plot Tools Property Editor.

3. Click the plot line in any of the nine axis.

This action selects the response for the `sys_mimo` system and updates the Plot Tools Property Editor to show the available editable properties for the response.



Note The Plot Tools Property Editor applies changes to the response of the MIMO system. Any change you make applies to all of the plotted lines in the figure.

Tip You can also change the properties of the response using the right-click menu while in plot edit mode.

4 In the **Property Editor – Waveform** pane, select the color red.

This action changes the color of the response that represents the MIMO system to red.

Customizing Response Plots from the Command Line

Overview of Customizing Plots from the Command Line

When to Customize Plots from the Command Line

You can customize any response plot from the command line. The command line is the most efficient way to customize a large number of plots. For example, if you have a batch job that produces many plots, you can change the *x*-axis units automatically for all the plot with just a few lines of code.

How to Customize Plots from the Command Line

You can use the Control System Toolbox application program interface (API) to customize plotting options for response plots from the command line.

Note This section assumes some very basic familiarity with MATLAB graphics objects. For more information, see “Graphics Objects”.

To customize plots from the command line:

1. Obtain the *plot handle*, which is an identifier for the plot, using the API's plotting syntax.

   For example,

   ```
   h = stepplot(sys)
   ```

   returns the plot handle `h` for the step plot.

   For more information on obtaining plot handles, see “Obtaining Plot Handles” on page 22-22.
2. Obtain the *plot options handle*, which is an identifier for all settable plot options. To get a plot options handle for a given plot, type

   ```
   p = getoptions(h);
   ```

   `p` is the plot options handle for plot handle `h`.

   For more information on obtaining plot options handles, see “Obtaining Plot Options Handles” on page 22-23.
3. Use `setoptions`, along with the plot handle and the plot options handle, to access and modify many plot options.

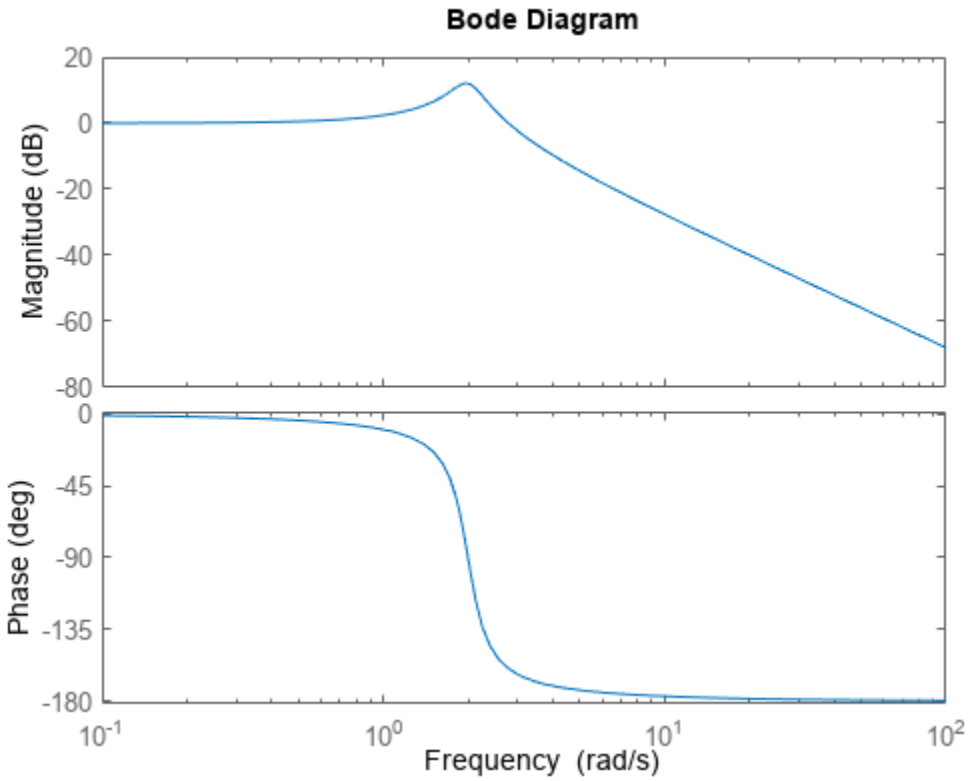
Note You can also use `setoptions` to customize plots using property/value pairs instead of the plot options handle. Using property/value pairs shortens the procedure to one line of code.

Change Bode Plot Units from the Command Line

This example shows how to change the units of a Bode plot from rad/s to Hz.

Create a system and generate a Bode Plot of the system's response. The plot uses the default units, rad/s.

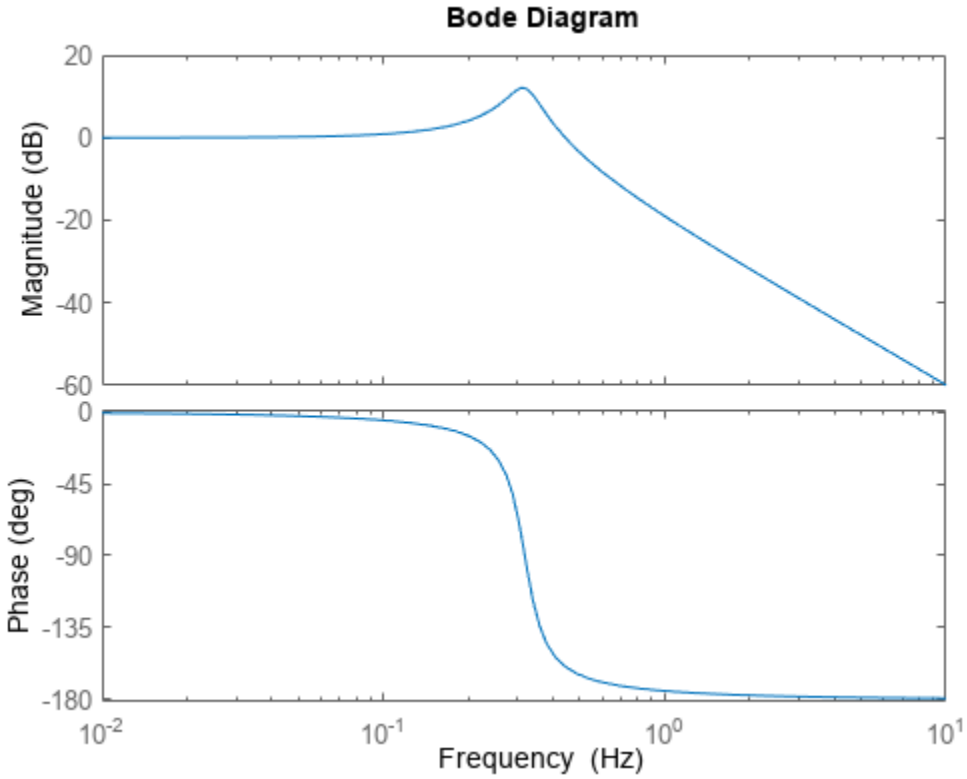
```
sys = tf(4,[1 0.5 4]);
h = bodeplot(sys);
```



The `bodeplot` command returns a plot handle that you can use to change properties of the plot.

Change the units to Hz.

```
p = getoptions(h);
p.FreqUnits = 'Hz';
setoptions(h,p)
```



The x-axis label updates to reflect the change of unit.

For more examples of customizing plots from the command line, see “Examples of Customizing Plots from the Command Line” on page 22-24.

Obtaining Plot Handles

To programmatically interact with response plot, you need the *plot handle*. This handle is an identifier to the response plot object. Because the Control System Toolbox plotting commands, `bode`, `rlocus`, etc., all use the plot handle internally, this API provides a set of commands that explicitly return the handle to your response plot. These functions all end with "plot," which makes them easy to identify. This table lists the functions.

Functions That Return the Plot Handle

Function	Plot
`bodeplot`	Bode magnitude and phase
`hsvplot`	Hankel singular values
`impulseplot`	Impulse response
`initialplot`	Initial condition
`iopzplot`	Pole/zero maps for input/output pairs
`lsimplot`	Time response to arbitrary inputs
`nicholsplot`	Nichols chart
`nyquistplot`	Nyquist
`pzplot`	Pole/zero
`rlocusplot`	Root locus
`sigmaplot`	Singular values of the frequency response
`stepplot`	Step response

To get a plot handle for any response plot, use the functions from the table. For example,

```
h = bodeplot(sys)
```

returns plot handle `h` (it also renders the Bode plot). Once you have this handle, you can modify the plot properties using the `setoptions` and `getoptions` methods of the plot object, in this case, a Bode plot handle.

Obtaining Plot Options Handles

Overview of Plot Options Handles

Once you have the plot handle, you need the *plot options handle*, which is an identifier for all the settable plot properties for a given response plot. There are two ways to create a plot options handle:

- Retrieving a Handle — Use `getoptions` to get the handle.
- Creating a Handle — Use `<`*`responseplot`*`>options` to instantiate a handle. See Functions for Creating Plot Options Handles for a complete list.

Retrieving a Handle

The `getoptions` function retrieves a plot options handle from a plot handle.

```
p=getoptions(h) % Returns plot options handle p for plot handle h.
```

If you specify a property name as an input argument, `getoptions` returns the property value associated with the property name.

```
property_value=getoptions(h,PropertyName) % Returns a property
                                          % value.
```

Creating a Handle

You can create a default plot options handle by using functions in the form of

`<responseplot>options`

For example,

```
p=bodeoptions;
```

instantiates a handle for Bode plots. See “Properties and Values Reference” on page 22-27 for a list of default values.

If you want to set the default values to the Control System Toolbox default values, pass `cstprefs` to the function. For example,

```
p = bodeoptions('cstprefs');
```

set the Bode plot property/value pairs to the Control System Toolbox default values.

This table lists the functions that create a plot options handle.

Functions for Creating Plot Options Handles

Function	Type of Plot Options Handle Created
`bodeoptions`	Bode phase and magnitude
`hsvoptions`	Hankel singular values
`nicholsoptions`	Nichols plot
`nyquistoptions`	Nyquist plot
`pzoptions`	Pole/zero plot
`sigmaoptions`	Sigma (singular values) plot
`timeoptions`	Time response (impulse, step, etc.)

Which Properties Can You Modify?

Use

`help <responseplot>options`

to see a list of available property value pairs that you can modify. For example,

```
help bodeoptions
```

You can modify any of these parameters using `setoptions`. The next topic provides examples of modifying various response plots.

See “Properties and Values Reference” on page 22-27 for a complete list of property/value pairs for response plots.

Examples of Customizing Plots from the Command Line

Manipulating Plot Options Handles

There are two fundamental ways to manipulate plot option handles:

- Dot notation — Treat the handle like a MATLAB structure.
- Property value pairs — Specify property/value pairs explicitly as input arguments to `setoptions`.

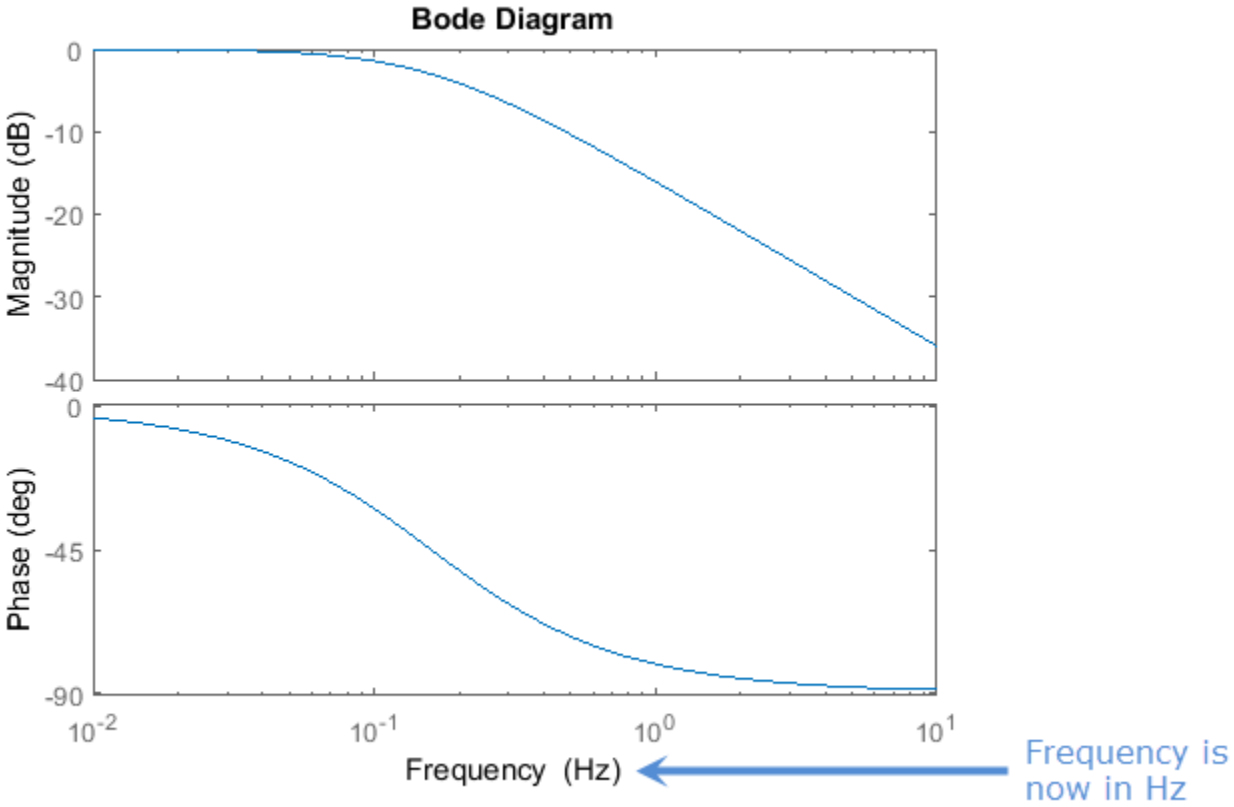
For some examples, both dot notation and property/value pairs approaches are shown. For all examples, use

```
sys = tf(1,[1 1]);
```

Changing Plot Units

Change the frequency units of a Bode plot from rad/s to Hz. To do so, extract the options `p` from the plot handle, edit the options, and assign them back to the plot.

```
h = bodeplot(sys);
p = getoptions(h);
p.FreqUnits = 'Hz';
setoptions(h,p)
```



Alternatively, instead of extracting `p`, set the options of `h` directly.

```
setoptions(h,'FreqUnits','Hz')
```

Create Plots Using Existing Plot Options Handle

You can use an existing plot options handle to customize a second plot:

```
h1 = bodeplot(sys);
p1 = getoptions(h1);
h2 = bodeplot(sys,p1);
```

or

```
h1 = bodeplot(sys);
h2 = bodeplot(sys2);
setoptions(h2,getoptions(h1))
```

Creating a Default Plot Options Handle

Instantiate a plot options handle with this code.

```
p = bodeoptions;
```

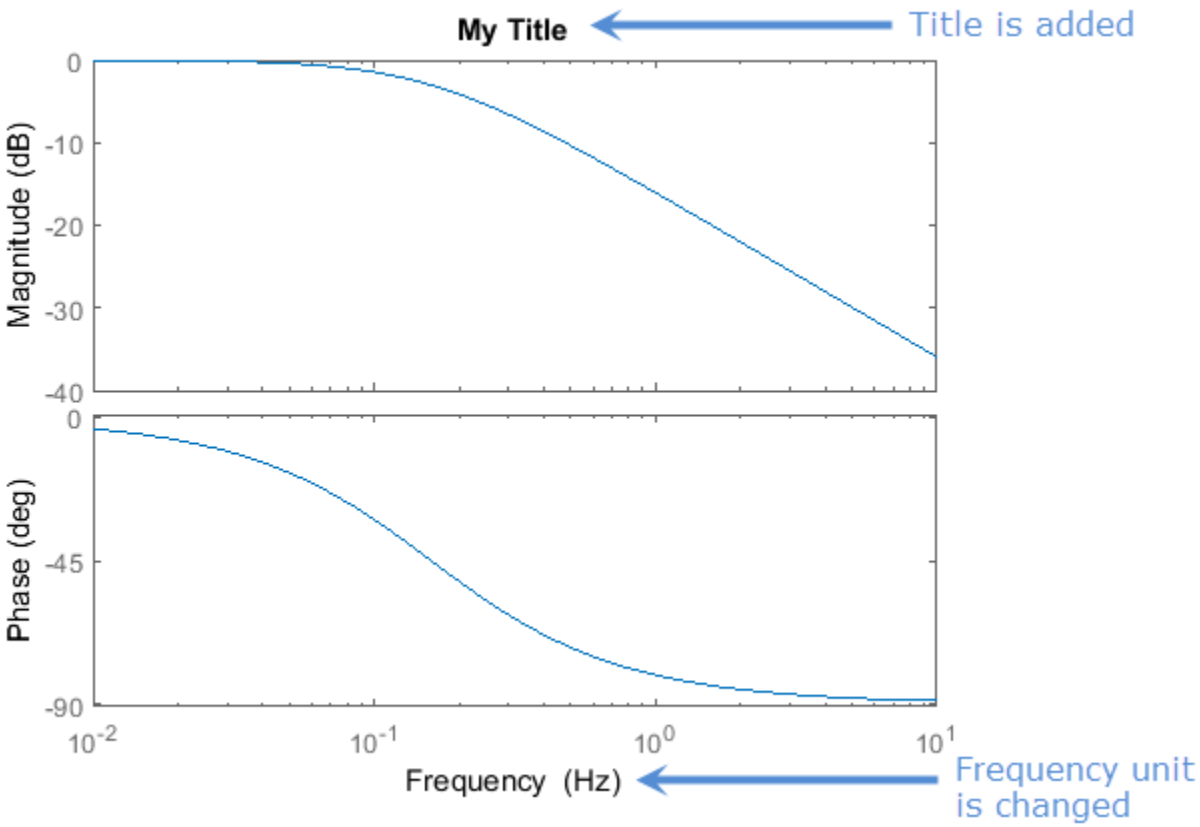
Change the frequency units and apply the changes to `sys`.

```
p.FreqUnits ='Hz';
h = bodeplot(sys,p);
```

Using Dot Notation Like a Structure

You can always use dot notation to assign values to properties, and change multiple plot properties at once.

```
h1 = bodeplot(sys);
p1 = getoptions(h1);
p1.FreqUnits = 'Hz';
p1.Title.String  =  'My Title';
setoptions(h1,p1)
```



Setting Property Pairs in setoptions

Instead of using dot notation, specify frequency units as property/value pairs in `setoptions`.

```
h1 = bodeplot(sys)
setoptions(h1,'FreqUnits','Hz')
```

Verify that the units have changed from rad/s to Hz.

```
getoptions(h1,'FreqUnits') % Returns frequency units for h1.

ans =

Hz
```

Properties and Values Reference

Property/Value Pairs Common to All Response Plots

The following tables discuss property/value pairs common to all response plots.

Title

Property	Default Value	Description
`Title.String`	none	Plot title, such as `'My Response Plot'`.
`Title.FontSize`	8	`Double`
`Title.FontWeight`	normal	`[light | normal | demi]`
`Title.FontAngle`	normal	`[normal | italic | oblique]`
`Title.Color`	[0 0 0]	`1-by-3 RGB vector`

X Label

Property	Default Value	Description
`XLabel.String`	none	X-axis label, such as `'Input Frequency'`.
`Xlabel.FontSize`	8	`Double`
`Xlabel.FontWeight`	normal	`[light | normal | demi]`
`XLabel.FontAngle`	normal	`[normal | italic | oblique]`
`Xlabel.Color`	[0 0 0]	`1-by-3 RGB vector`

Y Label

Property	Default Value	Description
`YLabel.String`	none	Y-axis label, such as `'Control Signal Magnitude'`.
`Ylabel.FontSize`	8	`Double`
`Ylabel.FontWeight`	normal	`[light | normal | demi]`
`YLabel.FontAngle`	normal	`[normal | italic | oblique]`
`Ylabel.Color`	[0 0 0]	`1-by-3 RGB vector`

Tick Label

Property	Default Value	Description
`TickLabel.FontSize`	8	`Double`
`TickLabel.FontWeight`	normal	`[light | normal | demi]`
`TickLabel.FontAngle`	normal	`[normal | italic | oblique]`
`Ticklabel.Color`	[0 0 0]	`1-by-3 RGB vector`

Grid and Axis Limits

Property	Default Value	Description
`grid`	`off`	`[on | off]`
`Xlim`	`{[]}`	A cell array of 1-by-2 doubles that specifies the *x*-axis limits when `XLimMode` is set to manual. When `XLim` is scalar, scalar expansion is applied; otherwise the length of the cell array must equal the number of columns (i.e., number of system inputs) for the plot. The 1-by-2 doubles must be a strictly increasing pair `[xmin, xmax]`.
`XLimMode`	`{'auto'}`	A cell array where each entry is either `'auto'` or `'manual'`. These entries specify the *x*-axis limits mode of the corresponding axis. When `XLimMode` is set to manual the limits are set to the values specified in `XLim`. When `XLim` is scalar, scalar expansion is applied; otherwise the length of the cell array must equal the number of columns (i.e., number of system inputs) for the plot.
`YLim`	`{[]}`	A cell array of 1-by-2 doubles specifies the *y*-axis limits when `YLimMode` is set to manual. When `YLim` is scalar, scalar expansion is applied; otherwise the length of the cell array must equal the number of rows (i.e., number of system outputs) for the plot. The 1-by-2 doubles must be a strictly increasing pair `[ymin, ymax]`.
`YLimMode`	`{'auto'}`	A cell array where each entry is either `'auto'` or `'manual'`. These entries specify the *y*-axis limits mode of the corresponding axis. When `YLimMode` is set to manual the limits are set to the values specified in `YLim`. When `YLim` is scalar, scalar expansion is applied; otherwise the length of the cell array must equal the number of rows (i.e., number of system outputs) for the plot.

I/O Grouping

Property	Default Value	Description
`IOGrouping`	none	`[none | inputs | outputs | all]` Specifies input/output groupings for responses.

Input Labels

Property	Default Value	Description
`InputLabels.FontSize`	8	`Double`
`InputLabels.FontWeight`	normal	`[light | normal | demi]`
`InputLabels.FontAngle`	normal	`[normal | italic | oblique]`
`InputLabels.Color`	[0 0 0]	`1-by-3 RGB vector`

Output Labels

Property	Default Value	Description
`OutputLabel.FontSize`	8	`Double`
`OutputLabels.FontWeight`	normal	`[light | normal | demi]`
`OutputLabels.FontAngle`	normal	`[normal | italic | oblique]`
`OutputLabels.Color`	[0 0 0]	`1-by-3 RGB vector`

Input/Output Visible

Property	Default Value	Description
`InputVisible`	{on}	`[on | off]` A cell array that specifies the visibility of each input channel. If the value is a scalar, scalar expansion is applied.
`OutputVisible`	{on}	`[on | off]` A cell array that specifies the visibility of each output channel. If the value is a scalar, scalar expansion is applied.

Bode Plots

Property	Default Value	Description
FreqUnits	rad/s	**Available Options** • 'Hz' • 'rad/s' • 'rpm' • 'kHz' • 'MHz' • 'GHz' • 'rad/nanosecond' • 'rad/microsecond' • 'rad/millisecond' • 'rad/minute' • 'rad/hour' • 'rad/day' • 'rad/week' • 'rad/month' • 'rad/year' • 'cycles/nanosecond' • 'cycles/microsecond' • 'cycles/millisecond' • 'cycles/hour' • 'cycles/day' • 'cycles/week' • 'cycles/month' • 'cycles/year'
FreqScale	log	[linear | log]
MagUnits	dB	[db | abs]
MagScale	linear	[linear | log]
PhaseUnits	deg	[rad | deg]
PhaseWrapping	off	[on | off] When you set PhaseWrapping to 'on', the plot wraps accumulated phase at the value specified by the PhaseWrappingBranch property.
PhaseWrappingBranch	–180	Double Phase value at which the plot wraps accumulated phase when PhaseWrapping is set to 'on'
MagVisible	on	[on | off]

Property	Default Value	Description
PhaseVisible	on	[on | off]
MagLowerLimMode	auto	[auto | manual] Enables a manual lower magnitude limit specification by MagLowerLim.
MagLowerLim	0	Double Specifies the lower magnitude limit when MagLowerLimMode is set to manual.
PhaseMatching	off	[on | off] Enables adjusting phase effects for phase response.
PhaseMatchingFreq	0	Double
PhaseMatchingValue	0	Double

Hankel Singular Values

Property	Default Value	Description
Yscale	linear	[linear | log]
AbsTol	0	Double See hsvd and stabsep for details.
RelTol	1*e-08	Double See hsvd and stabsep for details.
Offset	1*e-08	Double See hsvd and stabsep for details.

Nichols Plots

Property	Default Value	Description
FreqUnits	rad/s	**Available Options** • 'Hz' • 'rad/s' • 'rpm' • 'kHz' • 'MHz' • 'GHz' • 'rad/nanosecond' • 'rad/microsecond' • 'rad/millisecond' • 'rad/minute' • 'rad/hour' • 'rad/day' • 'rad/week' • 'rad/month' • 'rad/year' • 'cycles/nanosecond' • 'cycles/microsecond' • 'cycles/millisecond' • 'cycles/hour' • 'cycles/day' • 'cycles/week' • 'cycles/month' • 'cycles/year'
MagUnits	dB	[dB | abs]
PhaseUnits	deg	[rad | deg]
MagLowerLimMode	auto	[auto | manual]
MagLowerLim	0	double
PhaseWrapping	off	[on | off] When you set PhaseWrapping to 'on', the plot wraps accumulated phase at the value specified by the PhaseWrappingBranch property.

Property	Default Value	Description
PhaseWrappingBranch	–180	double Phase value at which the plot wraps accumulated phase when PhaseWrapping is set to 'on'
PhaseMatching	off	[on | off]
PhaseMatchingFreq	0	double
PhaseMatchingValue	0	double

Nyquist Charts

Property	Default Value	Description
FreqUnits	rad/s	**Available Options** • 'Hz' • 'rad/s' • 'rpm' • 'kHz' • 'MHz' • 'GHz' • 'rad/nanosecond' • 'rad/microsecond' • 'rad/millisecond' • 'rad/minute' • 'rad/hour' • 'rad/day' • 'rad/week' • 'rad/month' • 'rad/year' • 'cycles/nanosecond' • 'cycles/microsecond' • 'cycles/millisecond' • 'cycles/hour' • 'cycles/day' • 'cycles/week' • 'cycles/month' • 'cycles/year'
MagUnits	dB	[dB | abs]
PhaseUnits	deg	[rad | deg]
ShowFullContour	on	[on | off]

Pole/Zero Maps

Property	Default Value	Description
FreqUnits	rad/s	**Available Options** • 'Hz' • 'rad/s' • 'rpm' • 'kHz' • 'MHz' • 'GHz' • 'rad/nanosecond' • 'rad/microsecond' • 'rad/millisecond' • 'rad/minute' • 'rad/hour' • 'rad/day' • 'rad/week' • 'rad/month' • 'rad/year' • 'cycles/nanosecond' • 'cycles/microsecond' • 'cycles/millisecond' • 'cycles/hour' • 'cycles/day' • 'cycles/week' • 'cycles/month' • 'cycles/year'
TimeUnits	seconds	**Available Options** • 'nanoseconds' • 'microseconds' • 'milliseconds' • 'seconds' • 'minutes' • 'hours' • 'days' • 'weeks' • 'months' • 'years'

Sigma Plots

Property	Default Value	Description
FreqUnits	rad/s	**Available Options** • 'Hz' • 'rad/s' • 'rpm' • 'kHz' • 'MHz' • 'GHz' • 'rad/nanosecond' • 'rad/microsecond' • 'rad/millisecond' • 'rad/minute' • 'rad/hour' • 'rad/day' • 'rad/week' • 'rad/month' • 'rad/year' • 'cycles/nanosecond' • 'cycles/microsecond' • 'cycles/millisecond' • 'cycles/hour' • 'cycles/day' • 'cycles/week' • 'cycles/month' • 'cycles/year'
FreqScale	log	[linear | log]
MagUnits	dB	[dB | abs]
MagScale	linear	[linear | log]

Time Response Plots

Property	Default Value	Description
Normalize	off	[on | off] Normalize the *y*-scale of all responses in the plot.
SettleTimeThreshold	0.02	Double Specifies the settling time threshold. 0.02 = 2%.

Property	Default Value	Description
`RiseTimeLimits`	`[0.1, 0.9]`	`1-by-2 double` Specifies the limits used to define the rise time. `[0.1, 0.9]` is 10% to 90%.
`TimeUnits`	`seconds`	**Available Options** • `'nanoseconds'` • `'microseconds'` • `'milliseconds'` • `'seconds'` • `'minutes'` • `'hours'` • `'days'` • `'weeks'` • `'months'` • `'years'`

Build GUI With Interactive Response-Plot Updates

This example shows how to create a GUI to display a Control System Toolbox response plot that changes in response to interactive input.

The GUI in this example displays the step response of a second-order dynamic system of fixed natural frequency. The GUI includes a slider that sets the system's damping ratio. To cause the response plot to reflect the slider setting, you must define a callback for the slider. This callback uses the `updateSystem` command to update the plot with new system data in response to changes in the slider setting.

Set the initial values of the second-order dynamic system and create the system model.

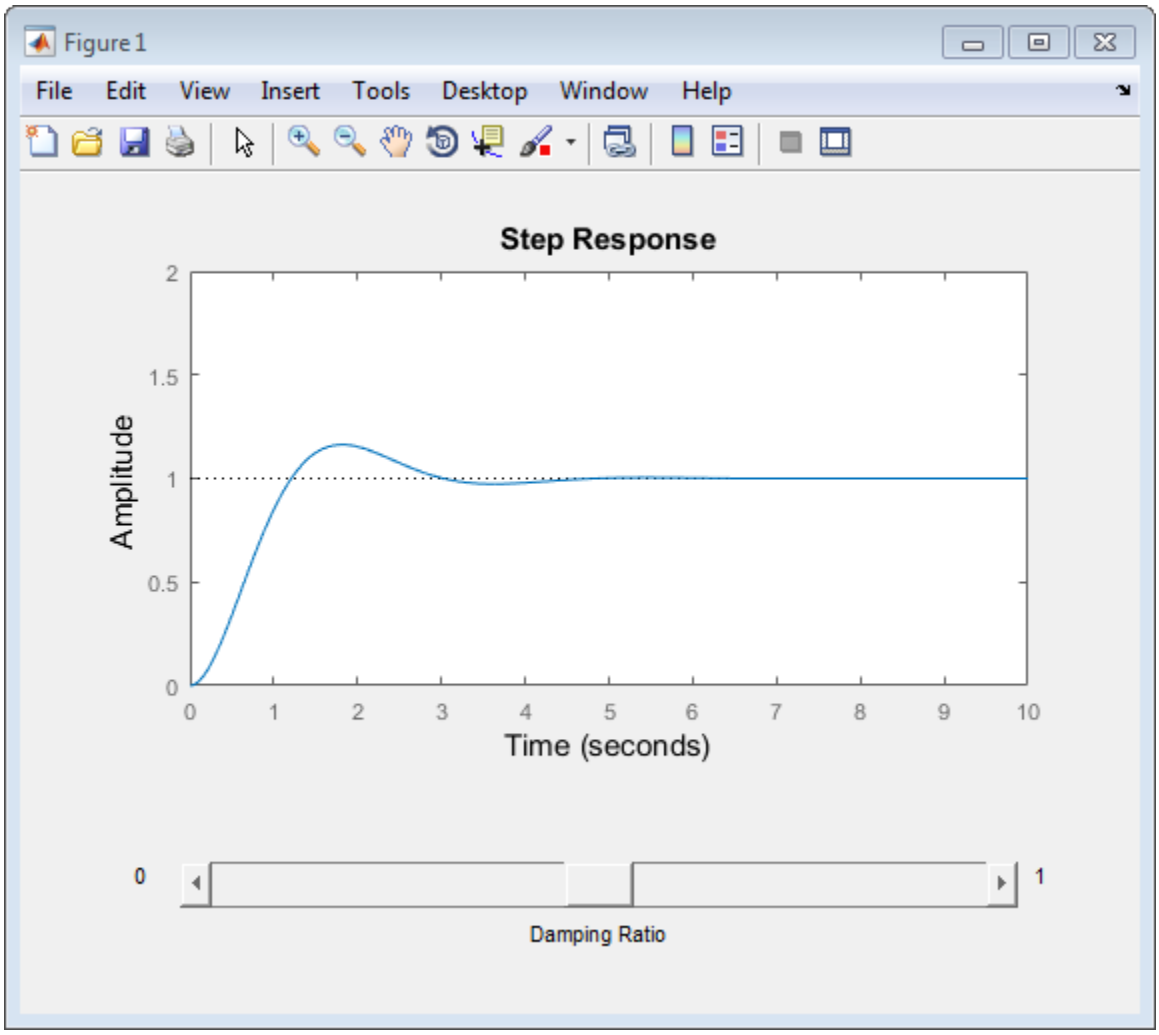
```
zeta = .5;                              % Damping Ratio
wn = 2;                                 % Natural Frequency
sys = tf(wn^2,[1,2*zeta*wn,wn^2]);
```

Create a figure for the GUI and configure the axes for displaying the step response.

```
f = figure;
ax = axes('Parent',f,'position',[0.13 0.39  0.77 0.54]);
h = stepplot(ax,sys);
setoptions(h,'XLim',[0,10],'YLim',[0,2]);
```

Add the slider and slider label text to the figure.

```
b = uicontrol('Parent',f,'Style','slider','Position',[81,54,419,23],...
              'value',zeta, 'min',0, 'max',1);
bgcolor = f.Color;
bl1 = uicontrol('Parent',f,'Style','text','Position',[50,54,23,23],...
                'String','0','BackgroundColor',bgcolor);
bl2 = uicontrol('Parent',f,'Style','text','Position',[500,54,23,23],...
                'String','1','BackgroundColor',bgcolor);
bl3 = uicontrol('Parent',f,'Style','text','Position',[240,25,100,23],...
                'String','Damping Ratio','BackgroundColor',bgcolor);
```



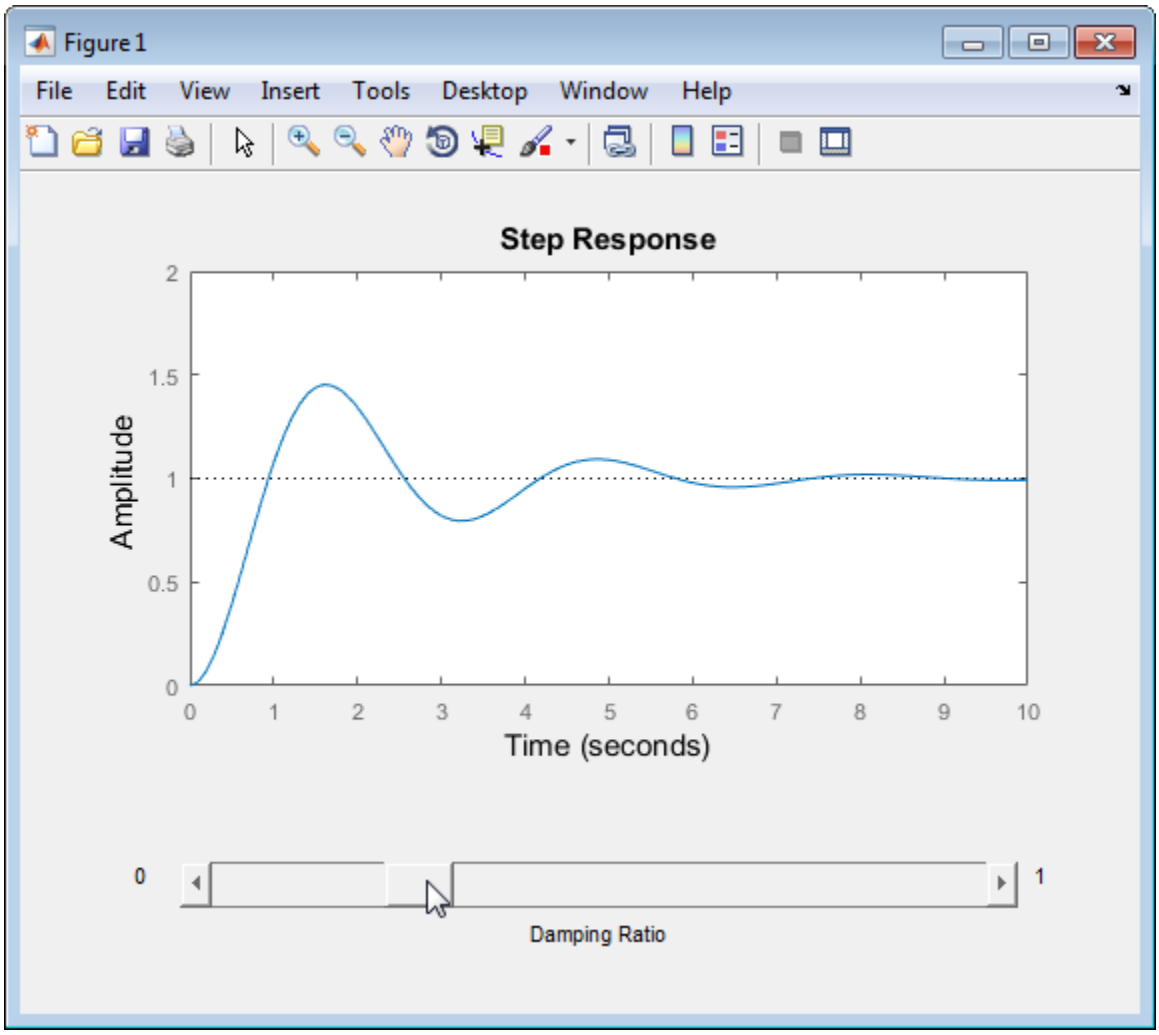
Set the callback that updates the step response plot as the damping ratio slider is moved.

```
b.Callback = @(es,ed) updateSystem(h,tf(wn^2,[1,2*(es.Value)*wn,wn^2]));
```

This code sets the callback for the slider (identified as `b`) to an anonymous function. The input arguments to this anonymous function, `es` and `ed`, are automatically passed to the callback when the slider is used. `es` is the handle of the `uicontrol` that represents the slider, and `ed` is the event data structure which the slider automatically passes to the callback. You do not need to define these variables in the workspace or set their values. (For more information about UI callbacks, see “Create Callbacks for Graphics Objects”.)

The callback is a call to the `updateSystem` function, which replaces the plotted response data with a response derived from a new transfer function. The callback uses the slider data `es.Value` to define a second-order system whose damping ratio is the current value of the slider.

Now that you have set the callback, move the slider. The displayed step response changes as expected.



See Also

`updateSystem` | `uicontrol`

Related Examples

- “Create Callbacks for Graphics Objects”
- “Create Callbacks for Apps Created Programmatically”

Design Case Studies

- “Design Yaw Damper for Jet Transport” on page 23-2
- “LQG Regulation: Rolling Mill Case Study” on page 23-14

Design Yaw Damper for Jet Transport

Overview of this Case Study

This case study demonstrates the tools for classical control design by stepping through the design of a yaw damper for a 747 jet transport aircraft.

Creating the Jet Model

The jet model during cruise flight at MACH = 0.8 and H = 40,000 ft. is

```
A=[-.0558 -.9968 .0802 .0415;
     .598 -.115 -.0318 0;
    -3.05 .388 -.4650 0;
        0 0.0805 1 0];

B=[ .00729  0;
   -0.475   0.00775;
    0.153   0.143;
     0      0];

C=[0 1 0 0;
   0 0 0 1];

D=[0 0;
   0 0];

sys = ss(A,B,C,D);
```

The following commands specify this state-space model as an LTI object and attach names to the states, inputs, and outputs.

```
states = {'beta' 'yaw' 'roll' 'phi'};
inputs = {'rudder' 'aileron'};
outputs = {'yaw' 'bank angle'};

sys = ss(A,B,C,D,'statename',states,...
                 'inputname',inputs,...
                 'outputname',outputs);
```

You can display the LTI model `sys` by typing `sys`. This command produces the following result.

```
a =
           beta      yaw     roll      phi
   beta  -0.0558  -0.9968   0.0802   0.0415
   yaw     0.598   -0.115  -0.0318        0
   roll    -3.05    0.388   -0.465        0
   phi         0   0.0805        1        0

b =
          rudder  aileron
   beta  0.00729        0
   yaw    -0.475  0.00775
   roll    0.153    0.143
   phi         0        0

c =
              beta   yaw  roll   phi
   yaw           0     1     0     0
```

```
   bank angle     0     0     0     1

d =
                rudder  aileron
   yaw               0        0
   bank angle        0        0

Continuous-time model.
```

The model has two inputs and two outputs. The units are radians for `beta` (sideslip angle) and `phi` (bank angle) and radians/sec for `yaw` (yaw rate) and `roll` (roll rate). The rudder and aileron deflections are in radians as well.

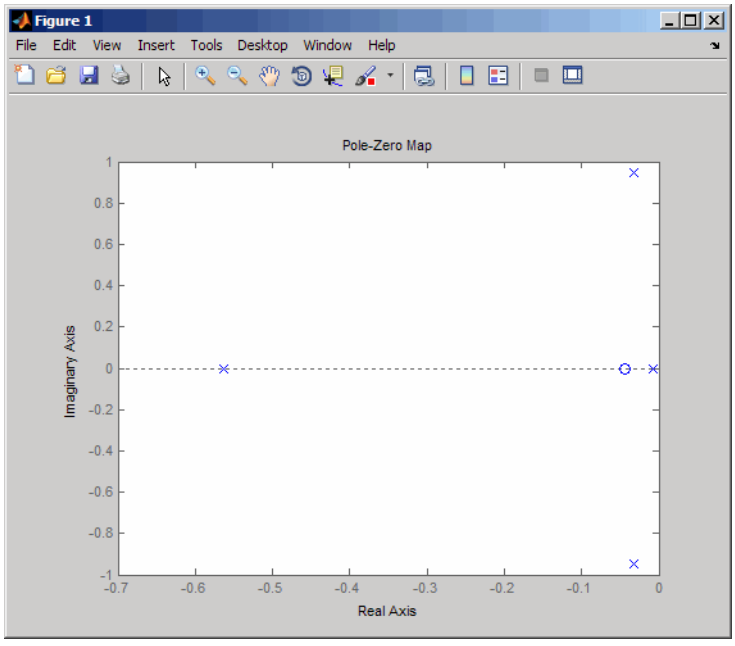
Computing Open-Loop Poles

Compute the open-loop poles and plot them in the *s*-plane.

```
>> damp(sys)
```

```
         Pole              Damping       Frequency      Time Constant
                                      (rad/seconds)        (seconds)

 -7.28e-03                 1.00e+00       7.28e-03          1.37e+02
 -5.63e-01                 1.00e+00       5.63e-01          1.78e+00
 -3.29e-02 + 9.47e-01i     3.48e-02       9.47e-01          3.04e+01
 -3.29e-02 - 9.47e-01i     3.48e-02       9.47e-01          3.04e+01
```

```
pzmap(sys)
```



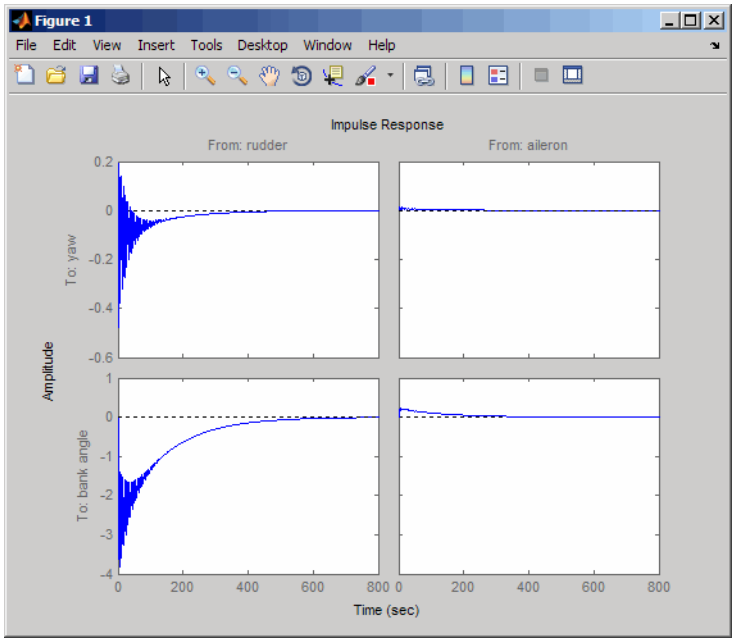
This model has one pair of lightly damped poles. They correspond to what is called the "Dutch roll mode."

Suppose you want to design a compensator that increases the damping of these poles, so that the resulting complex poles have a damping ratio $\zeta > 0.35$ with natural frequency $\omega_n < 1$ rad/sec. You can do this using the Control System Toolbox analysis tools.

Open-Loop Analysis

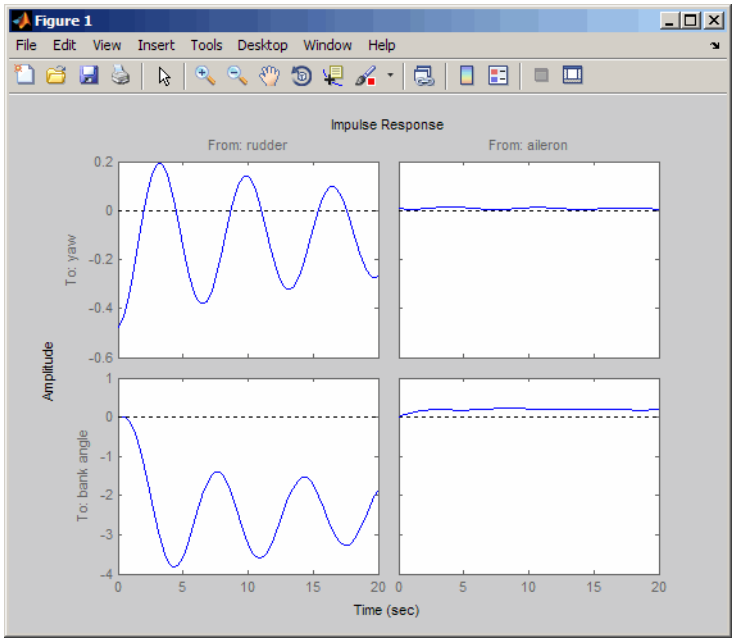
First, perform some open-loop analysis to determine possible control strategies. Start with the time response (you could use `step` or `impulse` here).

```
impulse(sys)
```

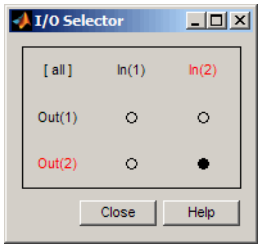


The impulse response confirms that the system is lightly damped. But the time frame is much too long because the passengers and the pilot are more concerned about the behavior during the first few seconds rather than the first few minutes. Next look at the response over a smaller time frame of 20 seconds.

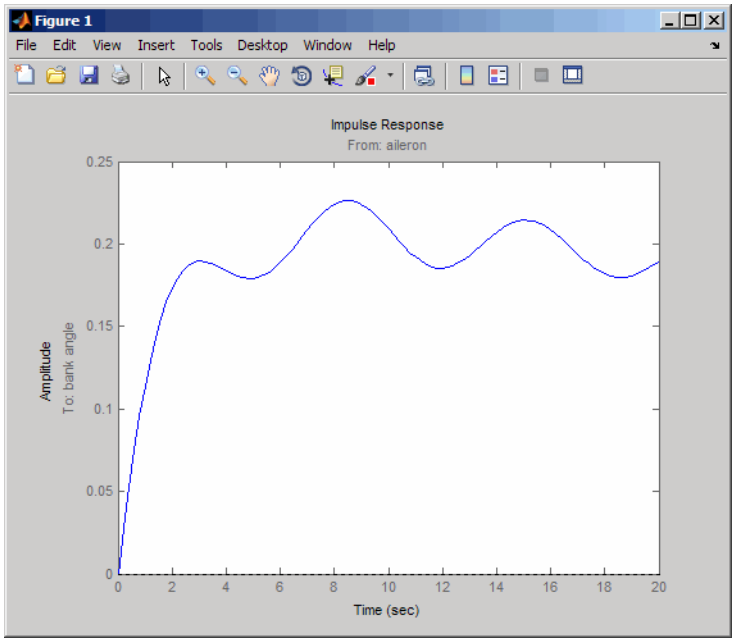
```
impulse(sys,20)
```



Look at the plot from aileron (input 2) to bank angle (output 2). To show only this plot, right-click and choose **I/O Selector**, then click on the (2,2) entry. The I/O Selector should look like this.



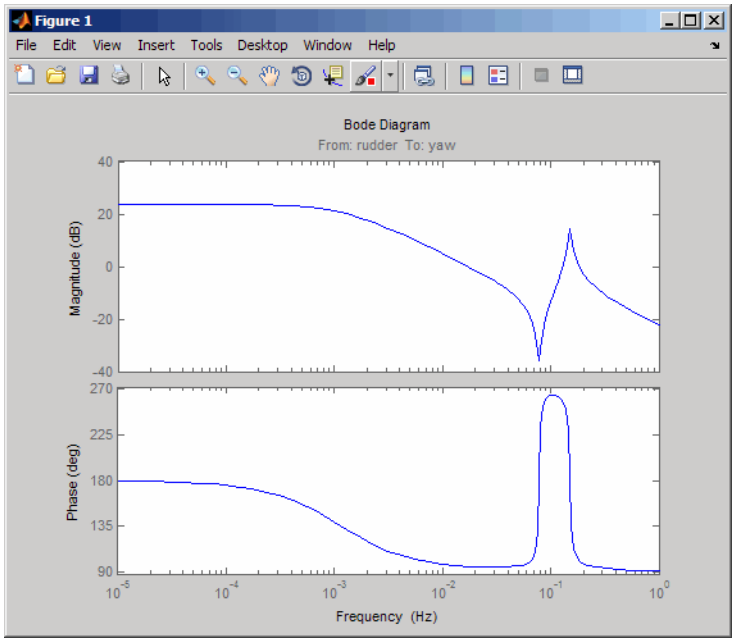
The new figure is shown below.



The aircraft is oscillating around a nonzero bank angle. Thus, the aircraft is turning in response to an aileron impulse. This behavior will prove important later in this case study.

Typically, yaw dampers are designed using the yaw rate as sensed output and the rudder as control input. Look at the corresponding frequency response.

```
sys11=sys('yaw','rudder') % Select I/O pair.
bode(sys11)
```

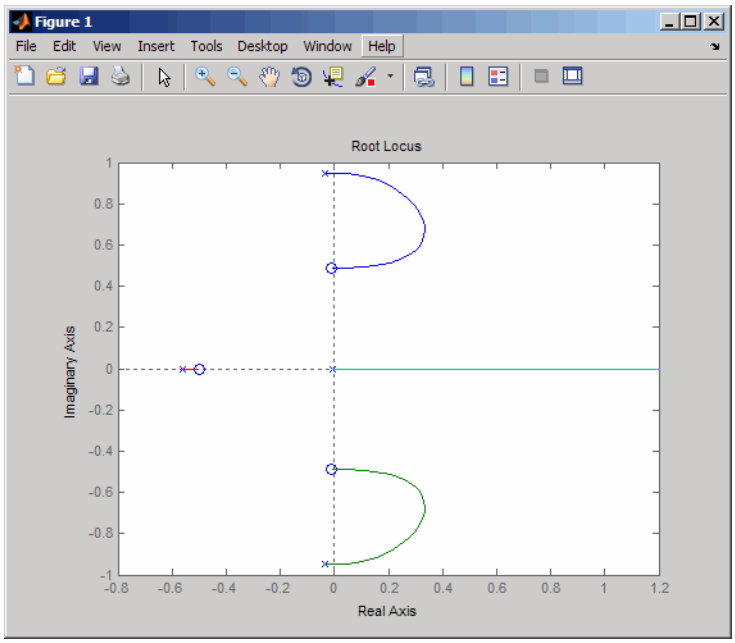


From this Bode diagram, you can see that the rudder has significant effect around the lightly damped Dutch roll mode (that is, near $\omega = 1$ rad/sec).

Root Locus Design

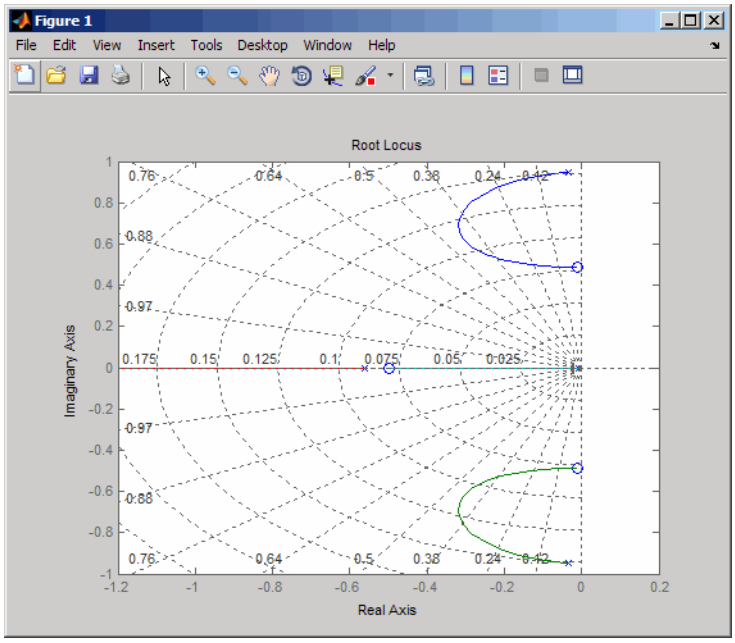
A reasonable design objective is to provide a damping ration $\zeta > 0.35$ with a natural frequency $\omega_n < 1.0$ rad/sec. Since the simplest compensator is a static gain, first try to determine appropriate gain values using the root locus technique.

```
% Plot the root locus for the rudder to yaw channel
rlocus(sys11)
```

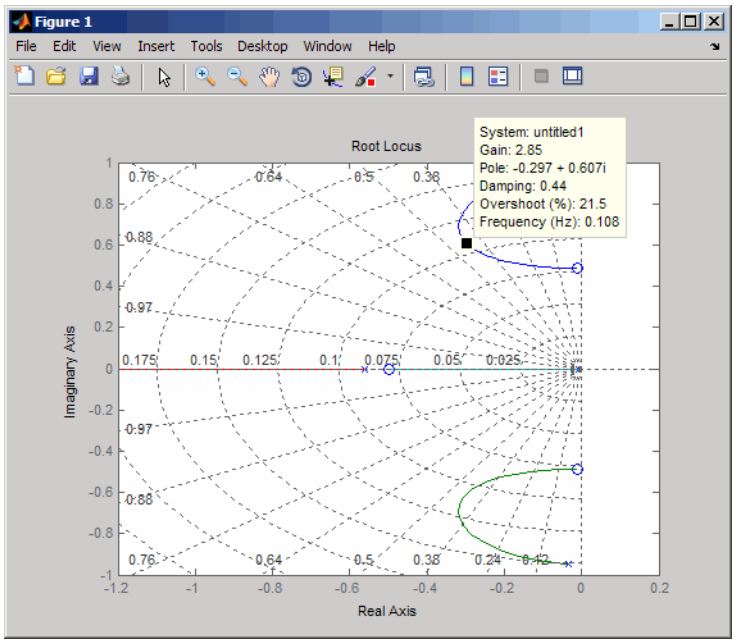


This is the root locus for negative feedback and shows that the system goes unstable almost immediately. If, instead, you use positive feedback, you may be able to keep the system stable.

```
rlocus(-sys11)
sgrid
```



This looks better. By using simple feedback, you can achieve a damping ratio of $\zeta > 0.45$. Click on the blue curve and move the data marker to track the gain and damping values. To achieve a 0.45 damping ratio, the gain should be about 2.85. This figure shows the data marker with similar values.

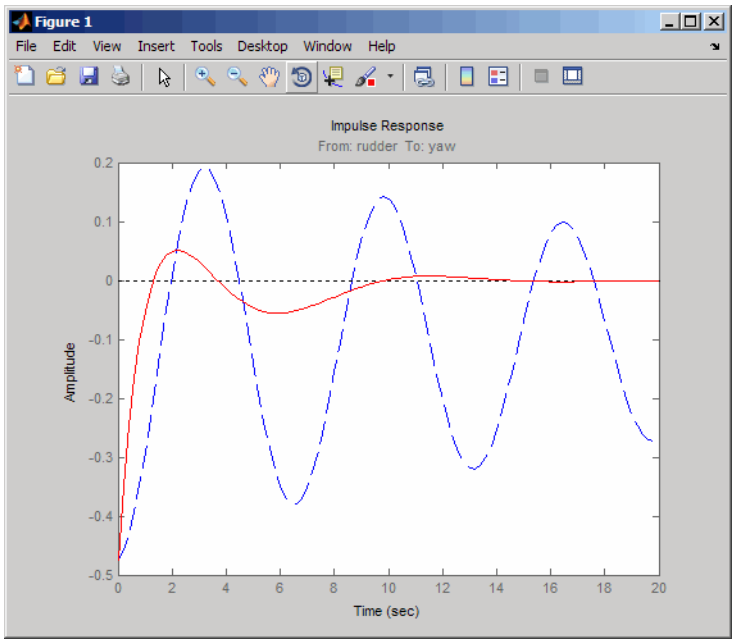


Next, close the SISO feedback loop.

```
K = 2.85;
cl11 = feedback(sys11,-K);  % Note: feedback assumes negative
                            % feedback by default
```

Plot the closed-loop impulse response for a duration of 20 seconds, and compare it to the open-loop impulse response.

```
impulse(sys11,'b--',cl11,'r',20)
```



The closed-loop response settles quickly and does not oscillate much, particularly when compared to the open-loop response.

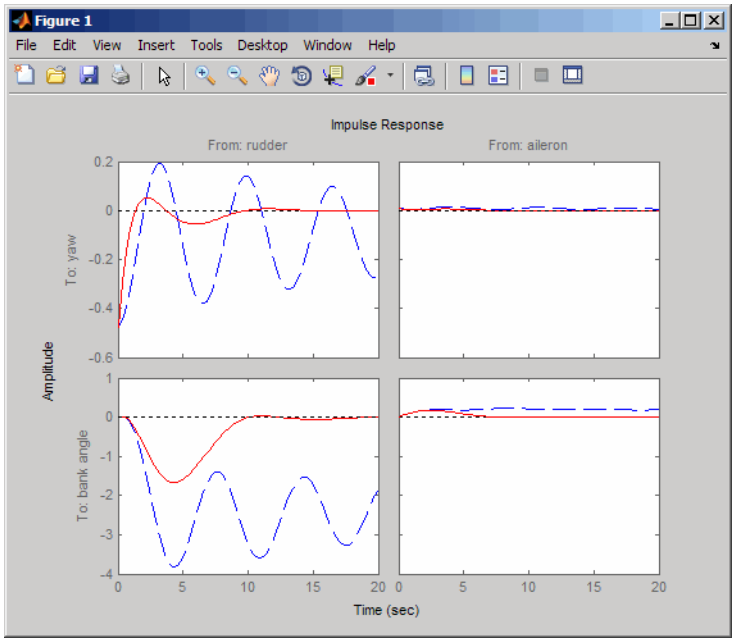
Now close the loop on the full MIMO model and see how the response from the aileron looks. The feedback loop involves input 1 and output 1 of the plant (use `feedback` with index vectors selecting this input/output pair). At the MATLAB prompt, type

```
cloop = feedback(sys,-K,1,1);
damp(cloop)   % closed-loop poles
```

```
         Pole              Damping       Frequency      Time Constant
                                      (rad/seconds)      (seconds)

 -3.42e-01                 1.00e+00       3.42e-01         2.92e+00
 -2.97e-01 + 6.06e-01i     4.40e-01       6.75e-01         3.36e+00
 -2.97e-01 - 6.06e-01i     4.40e-01       6.75e-01         3.36e+00
 -1.05e+00                 1.00e+00       1.05e+00         9.50e-01
```

Plot the MIMO impulse response.

```
impulse(sys,'b--',cloop,'r',20)
```



The yaw rate response is now well damped, but look at the plot from aileron (input 2) to bank angle (output 2). When you move the aileron, the system no longer continues to bank like a normal aircraft. You have over-stabilized the spiral mode. The spiral mode is typically a very slow mode and allows the aircraft to bank and turn without constant aileron input. Pilots are used to this behavior and will not like your design if it does not allow them to fly normally. This design has moved the spiral mode so that it has a faster frequency.

Washout Filter Design

What you need to do is make sure the spiral mode does not move further into the left-half plane when you close the loop. One way flight control designers have addressed this problem is to use a washout filter $kH(s)$ where

$$H(s) = \frac{s}{s + \alpha}$$

The washout filter places a zero at the origin, which constrains the spiral mode pole to remain near the origin. We choose $\alpha = 0.2$ for a time constant of five seconds and use the root locus technique to select the filter gain `H`. First specify the fixed part $s/(s + \alpha)$ of the washout by

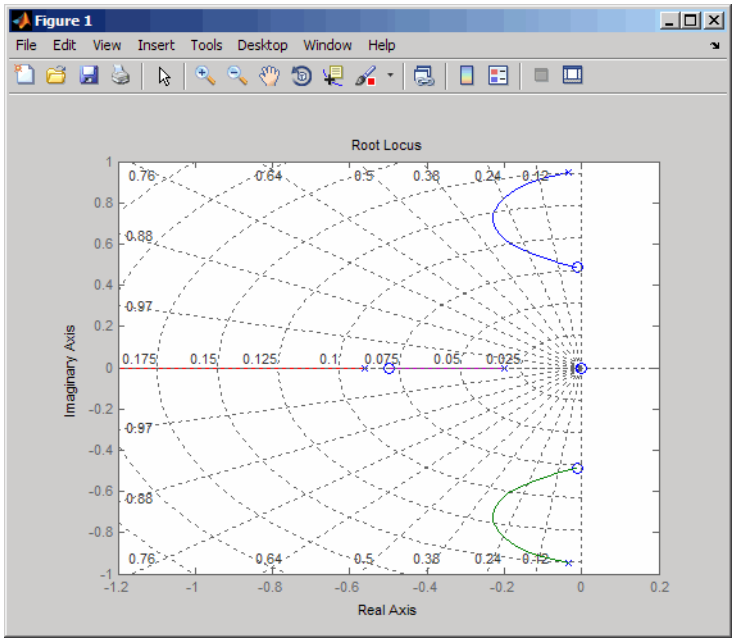
```
H = zpk(0,-0.2,1);
```

Connect the washout in series with the design model `sys11` (relation between input 1 and output 1) to obtain the open-loop model

```
oloop = H * sys11;
```

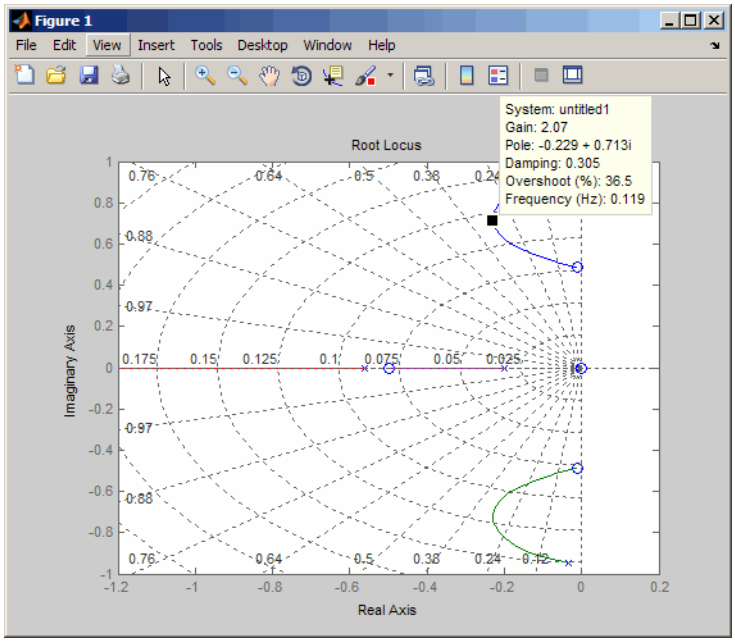
and draw another root locus for this open-loop model.

```
rlocus(-oloop)
sgrid
```



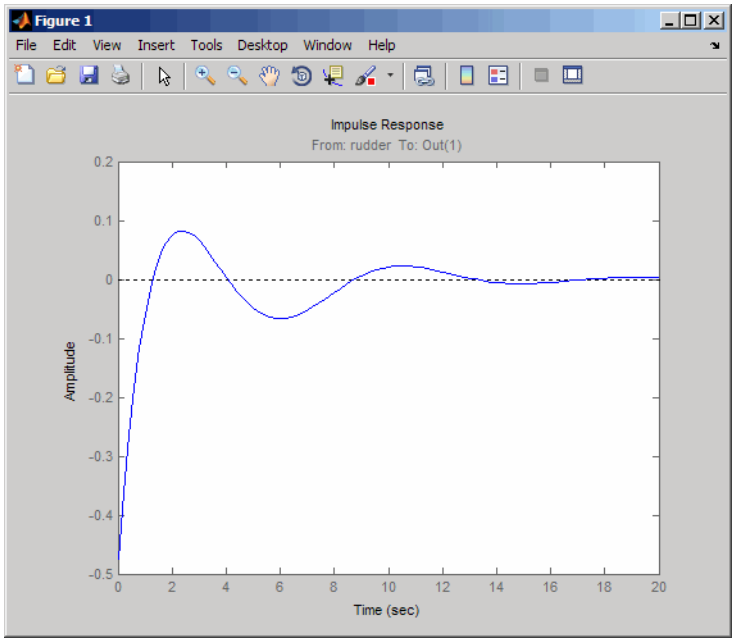
Create and drag a data marker around the upper curve to locate the maximum damping, which is about $\zeta = 0.3$.

This figure shows a data marker at the maximum damping ratio; the gain is approximately 2.07.



Look at the closed-loop response from rudder to yaw rate.

```
K = 2.07;
cl11 = feedback(oloop,-K);
impulse(cl11,20)
```

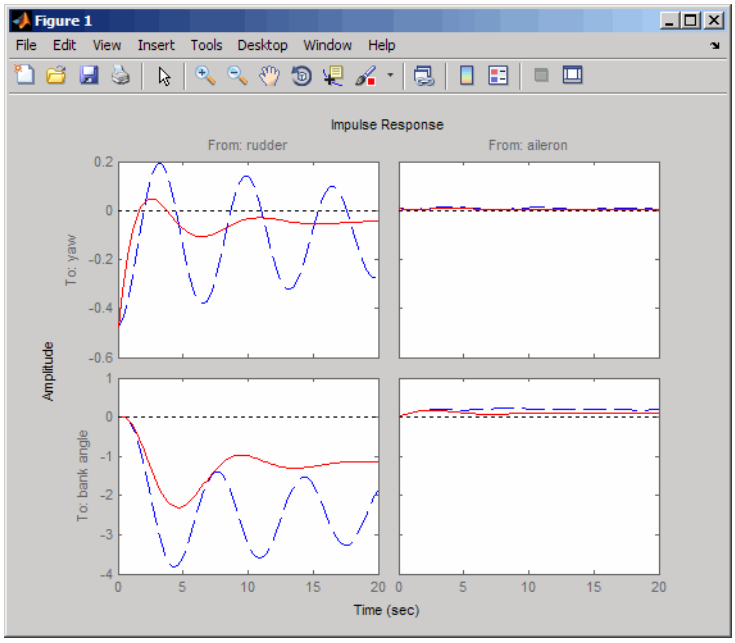


The response settles nicely but has less damping than your previous design. Finally, you can verify that the washout filter has fixed the spiral mode problem. First form the complete washout filter $kH(s)$ (washout + gain).

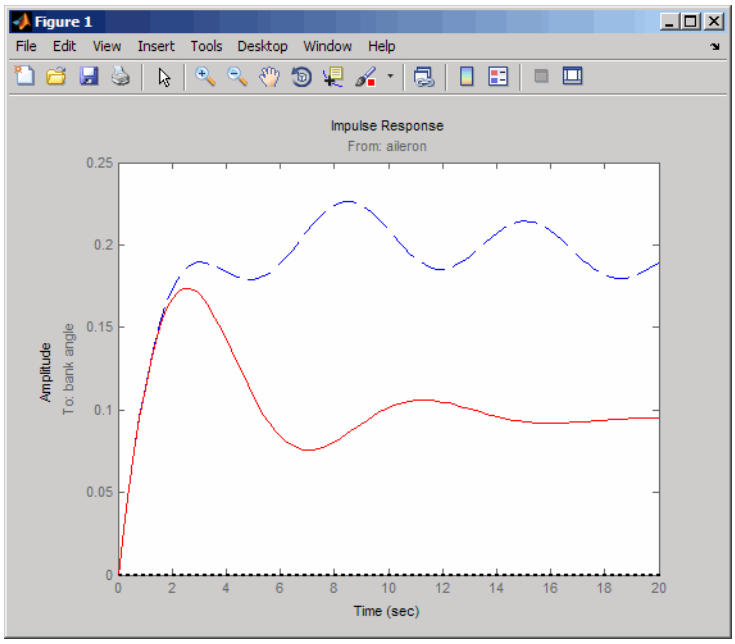
```
WOF = -K * H;
```

Then close the loop around the first I/O pair of the MIMO model `sys` and simulate the impulse response.

```
cloop = feedback(sys,WOF,1,1);

% Final closed-loop impulse response
impulse(sys,'b--',cloop,'r',20)
```



The bank angle response (output 2) due to an aileron impulse (input 2) now has the desired nearly constant behavior over this short time frame. To inspect the response more closely, use the I/O Selector in the right-click menu to select the (2,2) I/O pair.



Although you did not quite meet the damping specification, your design has increased the damping of the system substantially and now allows the pilot to fly the aircraft normally.

LQG Regulation: Rolling Mill Case Study

Overview of this Case Study

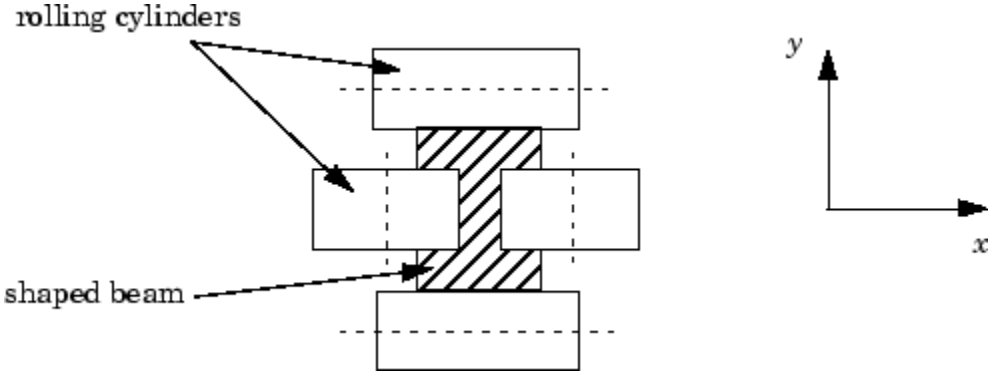
This case study demonstrates the use of the LQG design tools in a process control application. The goal is to regulate the horizontal and vertical thickness of the beam produced by a hot steel rolling mill. This example is adapted from [1]. The full plant model is MIMO and the example shows the advantage of direct MIMO LQG design over separate SISO designs for each axis. Type

```
milldemo
```

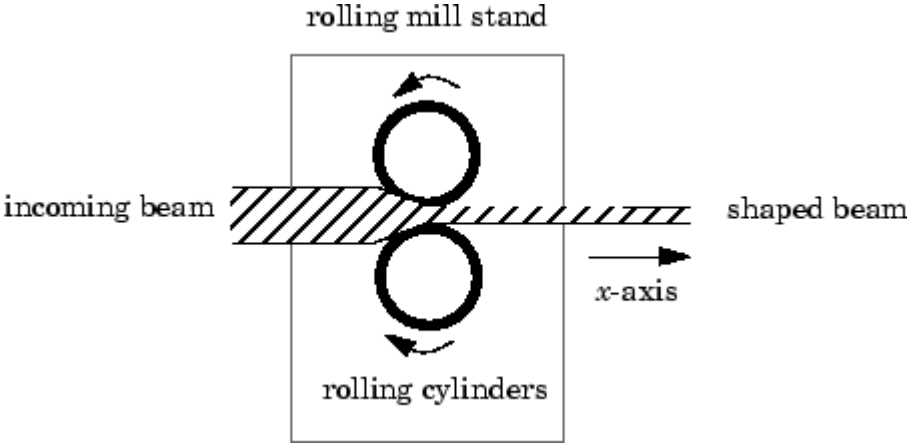
at the command line to run this demonstration interactively.

Process and Disturbance Models

The rolling mill is used to shape rectangular beams of hot metal. The desired outgoing shape is sketched below.



This shape is impressed by two pairs of rolling cylinders (one per axis) positioned by hydraulic actuators. The gap between the two cylinders is called the *roll gap*.



The objective is to maintain the beam thickness along the *x*- and *y*-axes within the quality assurance tolerances. Variations in output thickness can arise from the following:

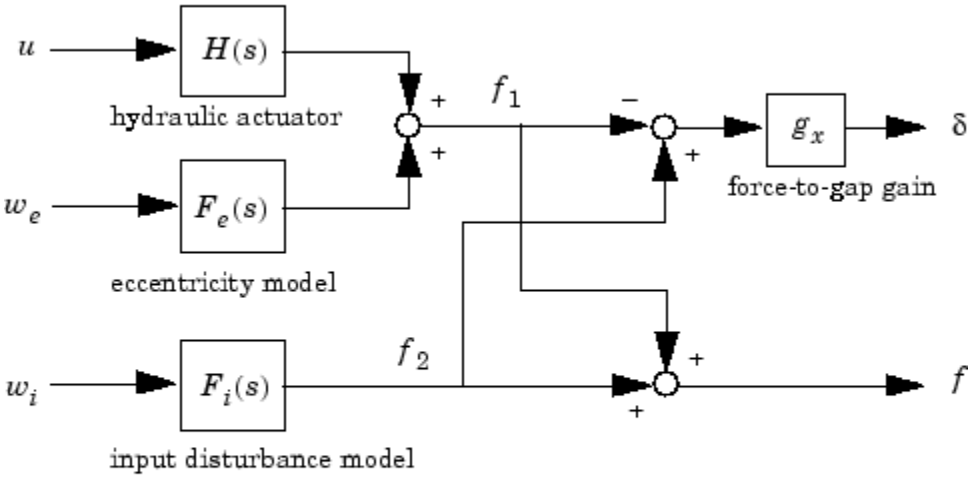
- Variations in the thickness/hardness of the incoming beam
- Eccentricity in the rolling cylinders

Feedback control is necessary to reduce the effect of these disturbances. Because the roll gap cannot be measured close to the mill stand, the rolling force is used instead for feedback.

The input thickness disturbance is modeled as a low pass filter driven by white noise. The eccentricity disturbance is approximately periodic and its frequency is a function of the rolling speed. A reasonable model for this disturbance is a second-order bandpass filter driven by white noise.

This leads to the following generic model for each axis of the rolling process.

Open-loop Model for x- or y-axis



u	command
δ	thickness gap (in mm)
f	incremental rolling force
w_i, w_e	driving white noise for disturbance models

The measured rolling force variation f is a combination of the incremental force delivered by the hydraulic actuator and of the disturbance forces due to eccentricity and input thickness variation. Note that:

- The outputs of $H(s)$, $F_e(s)$, and $F_i(s)$ are the incremental forces delivered by each component.
- An increase in hydraulic or eccentricity force *reduces* the output thickness gap δ.
- An increase in input thickness *increases* this gap.

The model data for each axis is summarized below.

Model Data for the x-Axis

$$H_x(s) = \frac{2.4 \times 10^8}{s^2 + 72s + 90^2}$$

$$F_{ix}(s) = \frac{10^4}{s + 0.05}$$

$$F_{ex}(s) = \frac{3 \times 10^4 s}{s^2 + 0.125s + 6^2}$$

$$g_x = 10^{-6}$$

Model Data for the y-Axis

$$H_y(s) = \frac{7.8 \times 10^8}{s^2 + 71s + 88^2}$$

$$F_{iy}(s) = \frac{2 \times 10^4}{s + 0.05}$$

$$F_{ey}(s) = \frac{10^5 s}{s^2 + 0.19s + 9.4^2}$$

$$g_y = 0.5 \times 10^{-6}$$

LQG Design for the x-Axis

As a first approximation, ignore the cross-coupling between the *x*- and *y*-axes and treat each axis independently. That is, design one SISO LQG regulator for each axis. The design objective is to reduce the thickness variations δ_x and δ_y due to eccentricity and input thickness disturbances.

Start with the *x*-axis. First specify the model components as transfer function objects.

```
% Hydraulic actuator (with input "u-x")
Hx = tf(2.4e8,[1 72 90^2],'inputname','u-x')

% Input thickness/hardness disturbance model
Fix = tf(1e4,[1 0.05],'inputn','w-ix')

% Rolling eccentricity model
Fex = tf([3e4 0],[1 0.125 6^2],'inputn','w-ex')

% Gain from force to thickness gap
gx = 1e-6;
```

Next build the open-loop model shown in “Process and Disturbance Models” on page 23-14. You could use the function `connect` for this purpose, but it is easier to build this model by elementary `append` and `series` connections.

```
% I/O map from inputs to forces f1 and f2
Px = append([ss(Hx) Fex],Fix)

% Add static gain from f1,f2 to outputs "x-gap" and "x-force"
Px = [-gx gx;1 1] * Px
```

```
% Give names to the outputs:
set(Px,'outputn',{'x-gap' 'x-force'})
```

Note To obtain minimal state-space realizations, always convert transfer function models to state space *before* connecting them. Combining transfer functions and then converting to state space may produce nonminimal state-space models.

The variable `Px` now contains an open-loop state-space model complete with input and output names.

```
Px.inputname

ans =
    'u-x'
    'w-ex'
    'w-ix'

Px.outputname

ans =
    'x-gap'
    'x-force'
```

The second output `'x-force'` is the rolling force measurement. The LQG regulator will use this measurement to drive the hydraulic actuator and reduce disturbance-induced thickness variations δ_x.

The LQG design involves two steps:

1. Design a full-state-feedback gain that minimizes an LQ performance measure of the form

   $$J(u_x) = \int_0^\infty \left\{ q\delta_x^2 + ru_x^2 \right\} dt$$

2. Design a Kalman filter that estimates the state vector given the force measurements `'x-force'`.

The performance criterion $J(u_x)$ penalizes low and high frequencies equally. Because low-frequency variations are of primary concern, eliminate the high-frequency content of δ_x with the low-pass filter $30/(s + 30)$ and use the filtered value in the LQ performance criterion.

```
lpf = tf(30,[1 30])

% Connect low-pass filter to first output of Px
Pxdes = append(lpf,1) * Px
set(Pxdes,'outputn',{'x-gap*' 'x-force'})

% Design the state-feedback gain using LQRY and q=1, r=1e-4
kx = lqry(Pxdes(1,1),1,1e-4)
```

Note `lqry` expects all inputs to be commands and all outputs to be measurements. Here the command `'u-x'` and the measurement `'x-gap*'` (filtered gap) are the first input and first output of `Pxdes`. Hence, use the syntax `Pxdes(1,1)` to specify just the I/O relation between `'u-x'` and `'x-gap*'`.

Next, design the Kalman estimator with the function `kalman`. The process noise

$$w_x = \begin{bmatrix} w_{ex} \\ w_{ix} \end{bmatrix}$$

has unit covariance by construction. Set the measurement noise covariance to 1000 to limit the high frequency gain, and keep only the measured output `'x-force'` for estimator design.

```
estx = kalman(Pxdes(2,:),eye(2),1000)
```

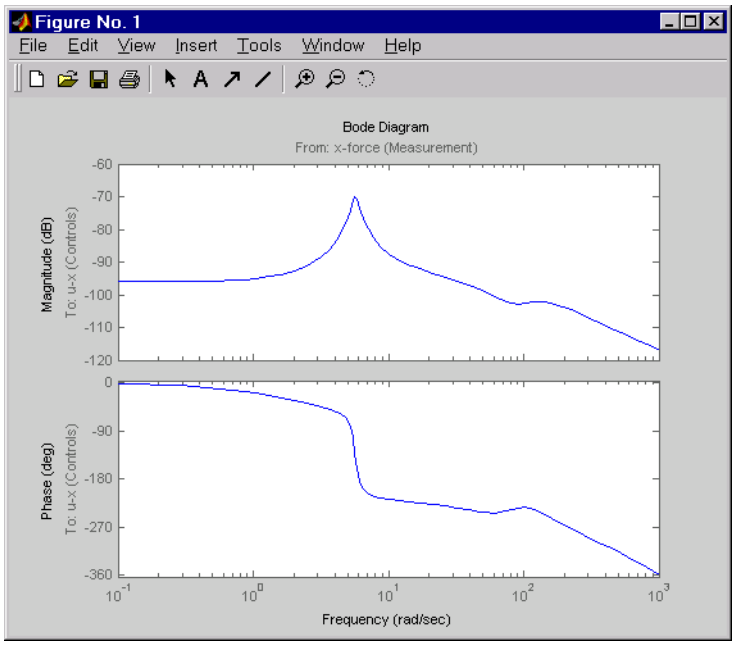
Finally, connect the state-feedback gain `kx` and state estimator `estx` to form the LQG regulator.

```
Regx = lqgreg(estx,kx)
```

This completes the LQG design for the $x-$axis.

Let's look at the regulator Bode response between 0.1 and 1000 rad/sec.

```
h = bodeplot(Regx,{0.1 1000})
setoptions(h,'PhaseMatching','on')
```



The phase response has an interesting physical interpretation. First, consider an increase in input thickness. This low-frequency disturbance boosts both output thickness and rolling force. Because the regulator phase is approximately 0º at low frequencies, the feedback loop then adequately reacts by increasing the hydraulic force to offset the thickness increase. Now consider the effect of eccentricity. Eccentricity causes fluctuations in the roll gap (gap between the rolling cylinders). When the roll gap is minimal, the rolling force increases and the beam thickness diminishes. The hydraulic force must then be reduced (negative force feedback) to restore the desired thickness. This is exactly what the LQG regulator does as its phase drops to -180º near the natural frequency of the eccentricity disturbance (6 rad/sec).

Next, compare the open- and closed-loop responses from disturbance to thickness gap. Use `feedback` to close the loop. To help specify the feedback connection, look at the I/O names of the plant `Px` and regulator `Regx`.

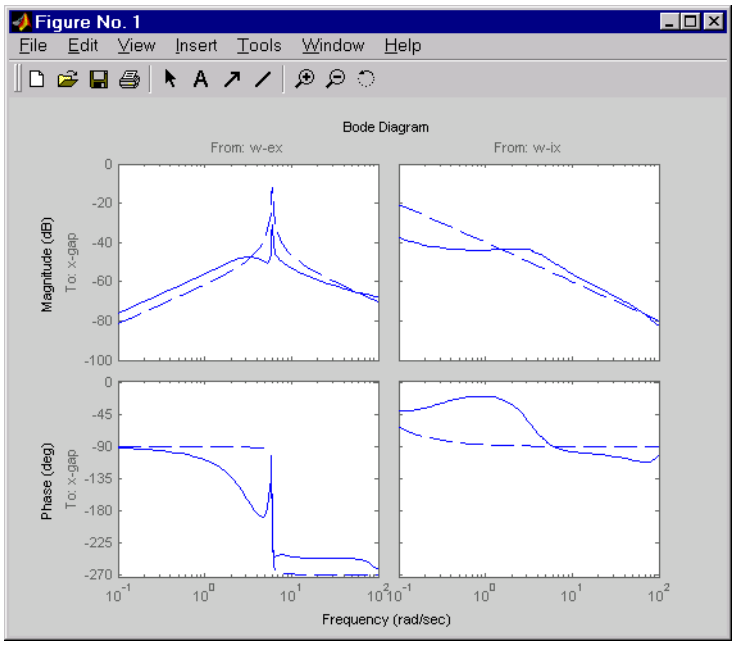
```
Px.inputname
ans =
    'u-x'
    'w-ex'
    'w-ix'

Regx.outputname
ans =
    'u-x'

Px.outputname
ans =
    'x-gap'
    'x-force'

Regx.inputname
ans =
    'x-force'
```

This indicates that you must connect the first input and second output of Px to the regulator.

```
clx = feedback(Px,Regx,1,2,+1)       % Note: +1 for positive feedback
```

You are now ready to compare the open- and closed-loop Bode responses from disturbance to thickness gap.

```
h = bodeplot(Px(1,2:3),'--',clx(1,2:3),'-',{0.1 100})
setoptions(h,'PhaseMatching','on')
```

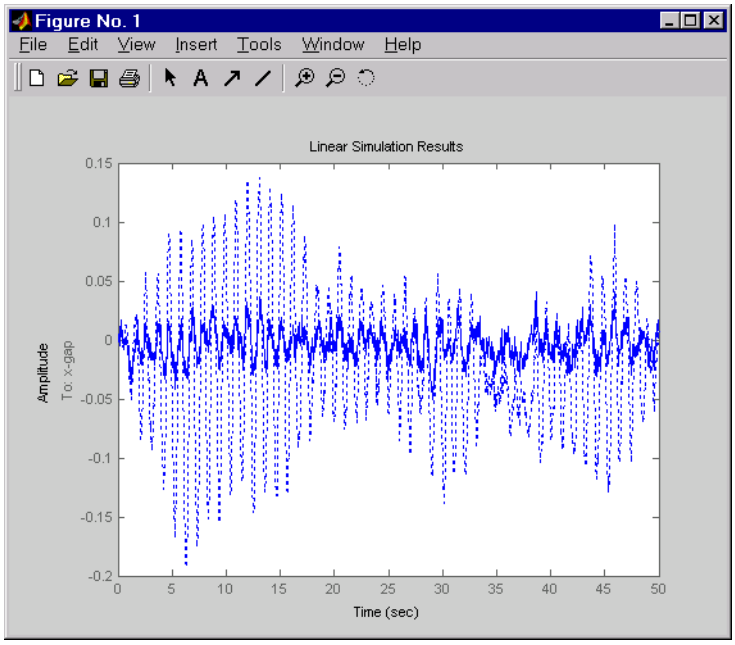


The dashed lines show the open-loop response. Note that the peak gain of the eccentricity-to-gap response and the low-frequency gain of the input-thickness-to-gap response have been reduced by about 20 dB.

Finally, use `lsim` to simulate the open- and closed-loop time responses to the white noise inputs w_{ex} and w_{ix}. Choose `dt=0.01` as sample time for the simulation, and derive equivalent discrete white noise inputs for this sampling rate.

```
dt = 0.01
t = 0:dt:50   % time samples

% Generate unit-covariance driving noise wx = [w-ex;w-ix].
% Equivalent discrete covariance is 1/dt
wx = sqrt(1/dt) * randn(2,length(t))

lsim(Px(1,2:3),':',clx(1,2:3),'-',wx,t)
```

Right-click on the plot that appears and select **Show Input** to turn off the display of the input.



The dotted lines correspond to the open-loop response. In this simulation, the LQG regulation reduces the peak thickness variation by a factor 4.

LQG Design for the y-Axis

The LQG design for the *y*-axis (regulation of the *y* thickness) follows the exact same steps as for the *x*-axis.

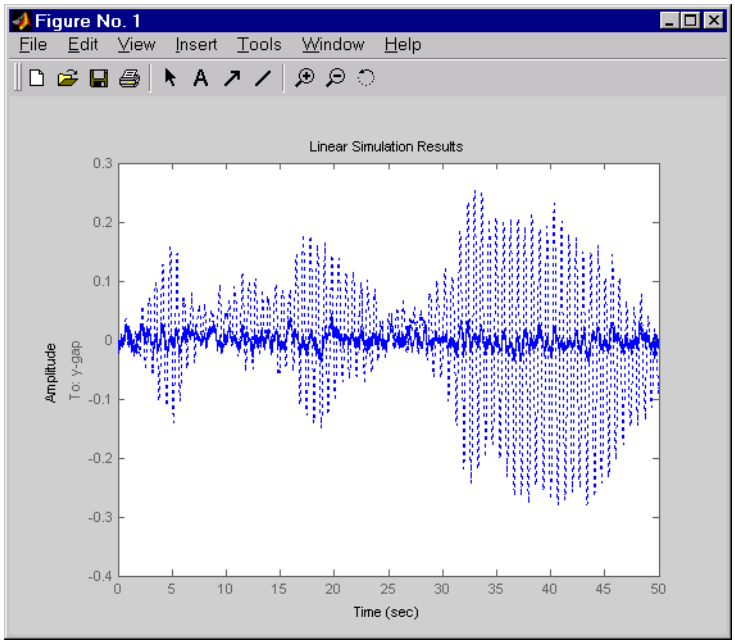
```
% Specify model components
Hy = tf(7.8e8,[1 71 88^2],'inputn','u-y')
Fiy = tf(2e4,[1 0.05],'inputn','w-iy')
Fey = tf([1e5 0],[1 0.19 9.4^2],'inputn','w-ey')
gy = 0.5e-6 % force-to-gap gain

% Build open-loop model
Py = append([ss(Hy) Fey],Fiy)
Py = [-gy gy;1 1] * Py
set(Py,'outputn',{'y-gap' 'y-force'})
```

```
% State-feedback gain design
Pydes = append(lpf,1) * Py % Add low-freq. weigthing
set(Pydes,'outputn',{'y-gap*' 'y-force'})
ky = lqry(Pydes(1,1),1,1e-4)

% Kalman estimator design
esty = kalman(Pydes(2,:),eye(2),1e3)

% Form SISO LQG regulator for y-axis and close the loop
Regy = lqgreg(esty,ky)
cly = feedback(Py,Regy,1,2,+1)
```

Compare the open- and closed-loop response to the white noise input disturbances.

```
dt = 0.01
t = 0:dt:50
wy = sqrt(1/dt) * randn(2,length(t))

lsim(Py(1,2:3),':',cly(1,2:3),'-',wy,t)
```

Right-click on the plot that appears and select **Show Input** to turn off the display of the input.

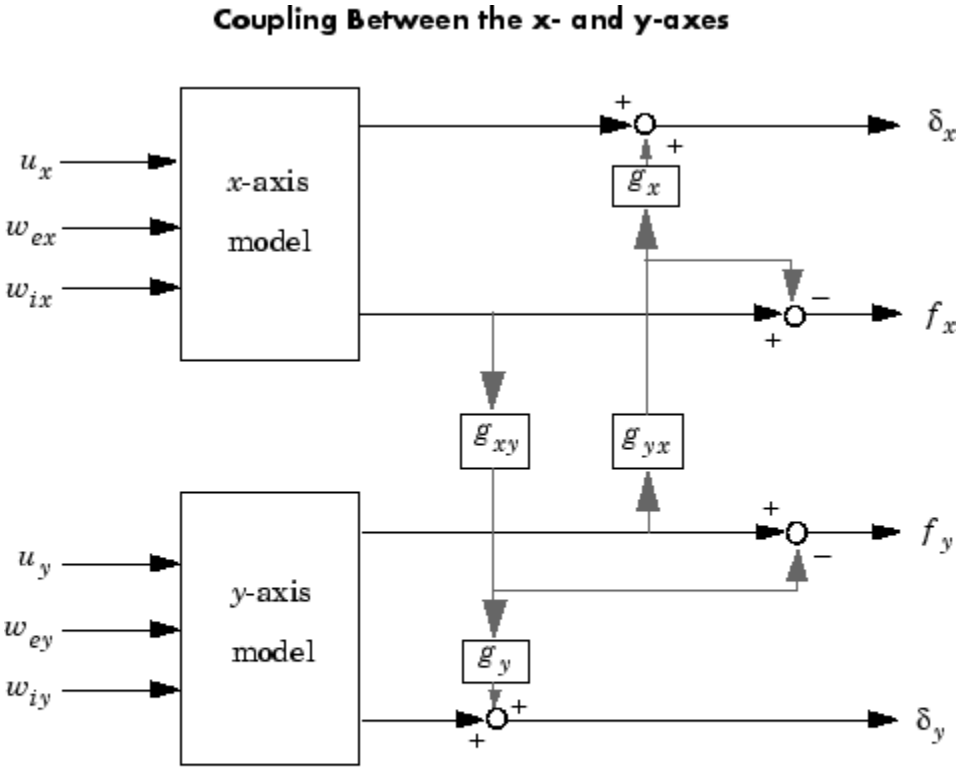


The dotted lines correspond to the open-loop response. The simulation results are comparable to those for the *x*-axis.

Cross-Coupling Between Axes

The *x*/*y* thickness regulation, is a MIMO problem. So far you have treated each axis separately and closed one SISO loop at a time. This design is valid as long as the two axes are fairly decoupled. Unfortunately, this rolling mill process exhibits some degree of cross-coupling between axes. Physically, an increase in hydraulic force along the *x*-axis compresses the material, which in turn boosts the repelling force on the *y*-axis cylinders. The result is an increase in *y*-thickness and an equivalent (relative) decrease in hydraulic force along the *y*-axis.

The figure below shows the coupling.



$$g_{xy} = 0.1$$
$$g_{yx} = 0.4$$

Accordingly, the thickness gaps and rolling forces are related to the outputs $\bar{\delta}_x, \bar{f}_x, \ldots$ of the *x*- and *y*-axis models by

$$\begin{bmatrix} \delta_x \\ \delta_y \\ f_x \\ f_x \end{bmatrix} = \underbrace{\begin{bmatrix} 1 & 0 & 0 & g_{yx}g_x \\ 0 & 1 & g_{xy}g_y & 0 \\ 0 & 0 & 1 & -g_{yx} \\ 0 & 0 & -g_{xy} & 1 \end{bmatrix}}_{\text{cross-coupling matrix}} \begin{bmatrix} \bar{\delta}_x \\ \bar{\delta}_y \\ \bar{f}_x \\ \bar{f}_y \end{bmatrix}$$

Let's see how the previous "decoupled" LQG design fares when cross-coupling is taken into account. To build the two-axes model, shown above, append the models `Px` and `Py` for the *x*- and *y*-axes.

```
P = append(Px,Py)
```

For convenience, reorder the inputs and outputs so that the commands and thickness gaps appear first.

```
P = P([1 3 2 4],[1 4 2 3 5 6])
P.outputname
```

```
ans =
    'x-gap'
    'y-gap'
    'x-force'
    'y-force'
```

Finally, place the cross-coupling matrix in series with the outputs.

```
gxy = 0.1; gyx = 0.4;
CCmat = [eye(2) [0 gyx*gx;gxy*gy 0] ; zeros(2) [1 -gyx;-gxy 1]]
Pc = CCmat * P
Pc.outputname = P.outputname
```

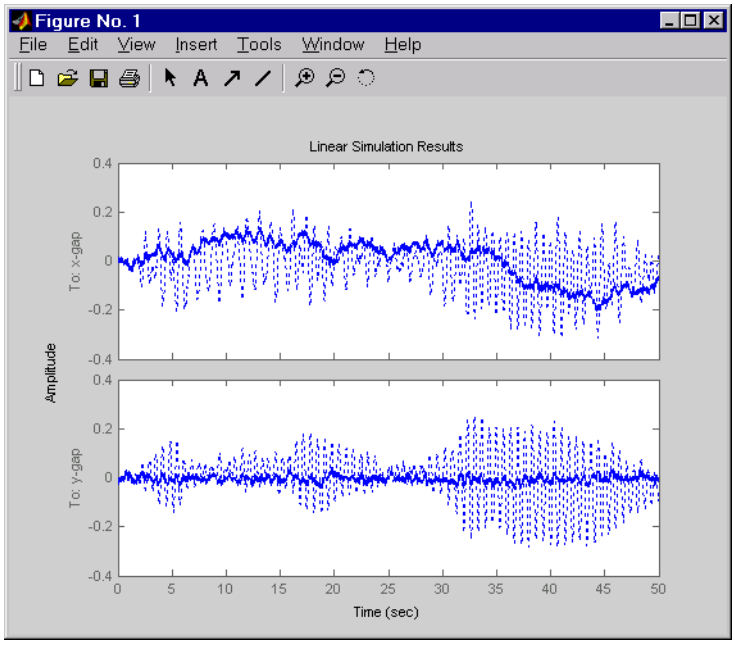
To simulate the closed-loop response, also form the closed-loop model by

```
feedin = 1:2 % first two inputs of Pc are the commands
feedout = 3:4 % last two outputs of Pc are the measurements
cl = feedback(Pc,append(Regx,Regy),feedin,feedout,+1)
```

You are now ready to simulate the open- and closed-loop responses to the driving white noises `wx` (for the *x*-axis) and `wy` (for the *y*-axis).

```
wxy = [wx ; wy]
lsim(Pc(1:2,3:6),':',cl(1:2,3:6),'-',wxy,t)
```

Right-click on the plot that appears and select **Show Input** to turn off the display of the input.



The response reveals a severe deterioration in regulation performance along the *x*-axis (the peak thickness variation is about four times larger than in the simulation without cross-coupling). Hence, designing for one loop at a time is inadequate for this level of cross-coupling, and you must perform a joint-axis MIMO design to correctly handle coupling effects.

MIMO LQG Design

Start with the complete two-axis state-space model `Pc` derived in “Cross-Coupling Between Axes” on page 23-21. The model inputs and outputs are

```
Pc.inputname

ans =
    'u-x'
    'u-y'
    'w-ex'
    'w-ix'
    'w_ey'
    'w_iy'

P.outputname

ans =
    'x-gap'
    'y-gap'
    'x-force'
    'y-force'
```

As earlier, add low-pass filters in series with the `'x-gap'` and `'y-gap'` outputs to penalize only low-frequency thickness variations.

```
Pdes = append(lpf,lpf,eye(2)) * Pc
Pdes.outputn = Pc.outputn
```

Next, design the LQ gain and state estimator as before (there are now two commands and two measurements).

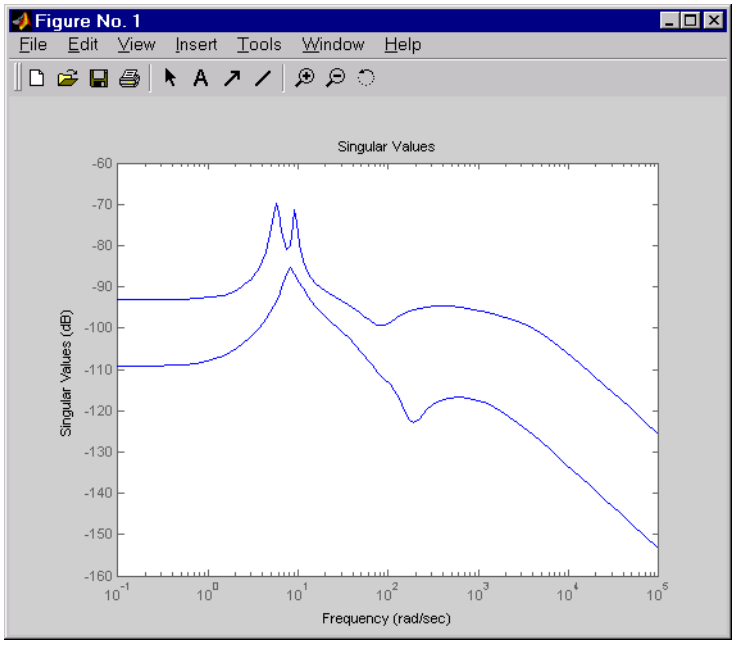
```
k = lqry(Pdes(1:2,1:2),eye(2),1e-4*eye(2))      % LQ gain
est = kalman(Pdes(3:4,:),eye(4),1e3*eye(2))     % Kalman estimator

RegMIMO = lqgreg(est,k)      % form MIMO LQG regulator
```

The resulting LQG regulator `RegMIMO` has two inputs and two outputs.

```
RegMIMO.inputname

ans =
    'x-force'
    'y-force'

RegMIMO.outputname

ans =
    'u-x'
    'u-y'
```

Plot its singular value response (principal gains).

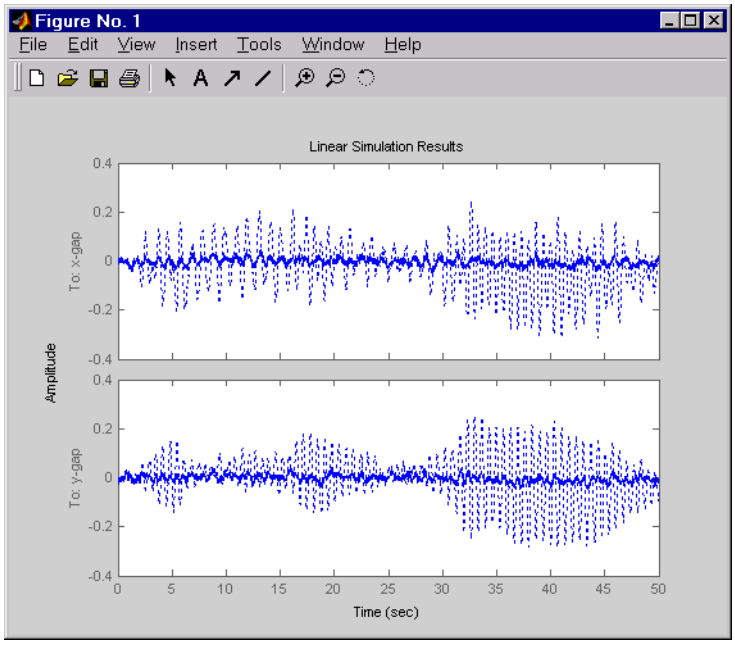
```
sigma(RegMIMO)
```



Next, plot the open- and closed-loop time responses to the white noise inputs (using the MIMO LQG regulator for feedback).

```
% Form the closed-loop model
cl = feedback(Pc,RegMIMO,1:2,3:4,+1);

% Simulate with lsim using same noise inputs
lsim(Pc(1:2,3:6),':',cl(1:2,3:6),'-',wxy,t)
```

Right-click on the plot that appears and select **Show Input** to turn off the display of the input.



The MIMO design is a clear improvement over the separate SISO designs for each axis. In particular, the level of *x*/*y* thickness variation is now comparable to that obtained in the decoupled case. This example illustrates the benefits of direct MIMO design for multivariable systems.

References

[1] Grimble, M.J., *Robust Industrial Control: Optimal Design Approach for Polynomial Systems*, Prentice Hall, 1994, p. 261 and pp. 443-456.

Canonical State-Space Realizations

State-Space Realizations

A state-space realization is an implementation of a given input-output behavior. If a system is modeled by a transfer matrix $H(s)$, then a realization is a set of matrices A, B, C, D such that $H(s) = C(sI - A)^{-1}B + D$. In other words, if the system has state vector x, the system behavior can be described by the following state equations:

$$\dot{x} = Ax + Bu$$
$$y = Cx + Du\,.$$

There are an infinite number of possible realizations of any system. A minimal realization is any realization in which A has the smallest possible dimension. That is, a given realization A, B, C, D is minimal if there is no other realization A', B', C', D' where A' has smaller dimensions than A.

A state transformation is a rotation of the state vector by an invertible matrix T such that $\widehat{x} = Tx$. State transformation yields an equivalent state-space representation of the system, with

$$\widehat{A} = TAT^{-1}$$
$$\widehat{B} = TB$$
$$\widehat{C} = CT^{-1}$$
$$\widehat{D} = D\,.$$

Certain minimal realizations known as canonical forms can be useful for some types of dynamic-system theory and analysis. This topic summarizes some of these canonical forms and related transformations.

Modal Form

Modal form is a diagonalized form that separates the system eigenvalues. In modal form, A is a block-diagonal matrix. The block size is typically 1-by-1 for real eigenvalues and 2-by-2 for complex eigenvalues. However, if there are repeated eigenvalues or clusters of nearby eigenvalues, the block size can be larger.

For instance, for a system with eigenvalues $(\lambda_1, \sigma \pm j\omega, \lambda_2)$, the modal A matrix is of the form

$$A_m = \begin{bmatrix} \lambda_1 & 0 & 0 & 0 \\ 0 & \sigma & \omega & 0 \\ 0 & -\omega & \sigma & 0 \\ 0 & 0 & 0 & \lambda_2 \end{bmatrix}.$$

Obtaining Modal Form

Modal form is the default form returned by the `canon` command, `Hmod = canon(H)`.

When performing system identification using `ssest`, obtain modal form by setting `Form` to `modal`.

Controllable Companion Form

In companion realizations, the characteristic polynomial of the system appears explicitly in the A matrix. For a SISO system with characteristic polynomial

$$P(s) = s^n + \alpha_{n-1}s^{n-1} + \alpha_{n-2}s^{n-2} + \ldots + \alpha_1 s + \alpha_0,$$

the corresponding controllable companion form has

$$A_{ccom} = \begin{bmatrix} 0 & 0 & 0 & \ldots & 0 & -\alpha_0 \\ 1 & 0 & 0 & \ldots & 0 & -\alpha_1 \\ 0 & 1 & 0 & \ldots & 0 & -\alpha_2 \\ 0 & 0 & 1 & \ldots & 0 & -\alpha_3 \\ \vdots & \vdots & \vdots & \ddots & \vdots & \vdots \\ 0 & 0 & 0 & \ldots & 1 & -\alpha_{n-1} \end{bmatrix}, \quad B_{ccom} = \begin{bmatrix} 1 \\ 0 \\ \vdots \\ 0 \end{bmatrix}.$$

For multi-input systems, A_{ccom} has the same form, and the first column of B_{ccom} is as shown. This form does not impose a particular structure on the rest of B_{ccom} or on C_{ccom} and D_{ccom}.

Obtaining Controllable Companion Form

The command `canon(H,"companion")` computes a controllable companion-form realization of `H` by using the state transformation `T = ctrb(H.A,H.B)` to put the A matrix into companion form.

When performing system identification using commands such as `ssest` or `n4sid`, obtain companion form by setting `Form` to `companion`.

The companion transformation requires that the system be controllable from the first input. The transformation to companion form is based on the controllability matrix, which is almost always numerically singular for mid-range orders. Hence, avoid using it for computation when possible.

Observable Companion Form

A related form is obtained using the observability state transformation `T = obsv(H.A,H.B)` instead of `T = ctrb(H.A,H.B)`. This form is the dual (transpose) of controllable companion form, as follows:

$$\begin{aligned} A_{ocom} &= A_{ccom}^T \\ B_{ocom} &= C_{ccom}^T \\ C_{ocom} &= B_{ccom}^T \\ D_{ocom} &= D_{ccom}^T. \end{aligned}$$

In particular,

$$A_{ocom} = \begin{bmatrix} 0 & 1 & 0 & 0 & \ldots & 0 \\ 0 & 0 & 1 & 0 & \ldots & 0 \\ 0 & 0 & 0 & 1 & \ldots & 0 \\ \vdots & \vdots & \vdots & \vdots & \ddots & \vdots \\ 0 & 0 & 0 & 0 & \ldots & 1 \\ -\alpha_0 & -\alpha_1 & -\alpha_2 & -\alpha_3 & \ldots & -\alpha_{n-1} \end{bmatrix}, \quad C_{ocom} = [1 \ 0 \ \ldots \ 0].$$

This form is sometimes known as observability canonical form [1], but it is different from observable canonical form on page 24-4.

Obtaining Observable Companion Form

When performing system identification using commands such as `ssest` or `n4sid`, obtain this form by setting `Form` to `canonical`.

When using the `canon` command, you can obtain observable companion form from the controllable companion form by performing the transpositions yourself. For example, for a state-space (`ss`) model `H`:

```
Hccom = canon(H,"companion");
Hocom = ss;
Hocom.A = Hccom.A';
Hocom.B = Hccom.C';
Hocom.C = Hccom.B';
Hocom.D = Hccom.D';
```

Controllable Canonical Form

For a strictly proper system with the transfer function

$$H(s) = \frac{\beta_{n-1}s^{n-1} + \ldots + \beta_1 s + \beta_0}{s^n + \alpha_{n-1}s^{n-1} + \ldots + \alpha_1 s + \alpha_0} + d_0,$$

the controllable canonical form [2] is given by:

$$A_{cont} = \begin{bmatrix} 0 & 1 & 0 & 0 & \ldots & 0 \\ 0 & 0 & 1 & 0 & \ldots & 0 \\ 0 & 0 & 0 & 1 & \ldots & 0 \\ \vdots & \vdots & \vdots & \vdots & \ddots & \vdots \\ 0 & 0 & 0 & 0 & \ldots & 1 \\ -\alpha_0 & -\alpha_1 & -\alpha_2 & -\alpha_3 & \ldots & -\alpha_{n-1} \end{bmatrix}, \quad B_{cont} = \begin{bmatrix} 0 \\ 0 \\ \vdots \\ 0 \\ 1 \end{bmatrix},$$

$$C_{cont} = \begin{bmatrix} \beta_0 & \beta_1 & \ldots & \beta_{n-1} \end{bmatrix}, \qquad D_{cont} = d_0 .$$

This form is also known as phase-variable canonical form. In this form, the coefficients of the characteristic polynomial appear in the last row of A_{cont}. Controllable canonical form is a minimal realization in which all model states are controllable. Like companion form and observable canonical form, it can be ill-conditioned for computation.

Obtaining Controllable Canonical Form

There is no MATLAB command for directly computing controllable canonical form. However, if you can obtain the system in the transfer-function form $H(s)$, then you can use the coefficients $\alpha_0,\ldots,\alpha_{n-1}$, $\beta_0,\ldots,\beta_{n-1}$, and d_0 to construct the controllable canonical-form matrices in MATLAB. Then, create the system with the `ss` command..

Observable Canonical Form

The observable canonical form of a system is the dual (transpose) of its controllable canonical form. In this form, the characteristic polynomial of the system appears explicitly in the last column of the `A` matrix. Observable canonical form can be obtained from the controllable canonical form as follows:

$$A_{obs} = A_{cont}^T$$
$$B_{obs} = C_{cont}^T$$
$$C_{obs} = B_{cont}^T$$
$$D_{obs} = D_{cont}^T .$$

Thus, for the system with transfer function

$$H(s) = \frac{\beta_{n-1}s^{n-1} + \ldots + \beta_1 s + \beta_0}{s^n + \alpha_{n-1}s^{n-1} + \ldots + \alpha_1 s + \alpha_0} + d_0,$$

the observable canonical form [2] is given by:

$$A_{obs} = \begin{bmatrix} 0 & 0 & 0 & \ldots & 0 & -\alpha_0 \\ 1 & 0 & 0 & \ldots & 0 & -\alpha_1 \\ 0 & 1 & 0 & \ldots & 0 & -\alpha_2 \\ 0 & 0 & 1 & \ldots & 0 & -\alpha_3 \\ \vdots & \vdots & \vdots & \ddots & \vdots & \vdots \\ 0 & 0 & 0 & \ldots & 1 & -\alpha_{n-1} \end{bmatrix}, \quad B_{obs} = \begin{bmatrix} \beta_0 \\ \beta_1 \\ \beta_2 \\ \vdots \\ \beta_{n-1} \end{bmatrix},$$
$$C_{obs} = [0 \ 0 \ \ldots \ 0 \ 1], \qquad D_{obs} = d_0 .$$

Like the companion form, in this form, the coefficients of the characteristic polynomial appear in the last column of A_{obs}. Observable canonical form is a minimal realization in which all model states are observable.

Obtaining Observable Canonical Form

As with controllable canonical form, there is no MATLAB command for directly computing observable canonical form. However, if you can obtain the system in the transfer-function form $H(s)$, then you can use the coefficients $\alpha_0,\ldots,\alpha_{n-1}$, $\beta_0,\ldots,\beta_{n-1}$, and d_0 to construct the observable canonical-form matrices in MATLAB. Then, create the system with the `ss` command.

References

[1] Baillieul, John, "Observability Canonical Form and the Theory of Observers," lecture notes, November 15, 2012, accessed June 10, 2022, https://people.bu.edu/johnb/501Lecture19.pdf.

[2] Gillis, James T., "State Space." In *Control System Fundamentals.*, edited by William S. Levine, 2d ed. The Electrical Engineering Handbook Series. Boca Raton: CRC Press, 2011.

See Also

`canon` | `ss` | `ssest` | `n4sid`

More About

- “Scaling State-Space Models” on page 25-2

Reliable Computations

Scaling State-Space Models

Why Scaling Is Important

When working with state-space models, proper scaling is important for accurate computations. A state-space model is well scaled when the following conditions exist:

- The entries of the A, B, and C matrices are homogeneous in magnitude.
- The model characteristics are insensitive to small perturbations in A, B, and C (in comparison to their norms).

Working with poorly scaled models can cause your model a severe loss of accuracy and puzzling results. An example of a poorly scaled model is a dynamic system with two states in the state vector that have units of light years and millimeters. Such disparate units may introduce both very large and very small entries into the A matrix. Over the course of computations, this mix of small and large entries in the matrix could destroy important characteristics of the model and lead to incorrect results.

For more information on the harmful affects of a poorly scaled model, see “Scaling State-Space Models to Maximize Accuracy” on page 5-59.

When to Scale Your Model

You can avoid scaling issues altogether by carefully selecting units to reduce the spread between small and large coefficients.

In general, you do not have to perform your own scaling when using the Control System Toolbox software. The algorithms automatically scale your model to prevent loss of accuracy. The automated scaling chooses a frequency range to maximize accuracy based on the dominant dynamics of the model.

In most cases, automated scaling provides high accuracy without your intervention. For some models with dynamics spanning a wide frequency range, however, it is impossible to achieve good accuracy at *all* frequencies and some tradeoff of accuracy in different frequency bands is necessary. In such cases, a warning alerts you of potential inaccuracies. If you receive this warning, evaluate the tradeoffs and consider manually adjusting the frequency interval where you most need high accuracy. For information on how to manually scale your model, see “Manually Scale Your Model” on page 25-2.

Note For models with satisfactory scaling, you can bypass automated scaling in the Control System Toolbox software. To do so, set the `Scaled` property of your state-space model to `1` (true). For information on how to set this property, see the `set` reference page.

Manually Scale Your Model

If automatic scaling produces a warning, you can use the `prescale` command to manually scale your model and adjust the frequency interval where you most need high accuracy.

The `prescale` command includes a Scaling Tool, which you can use to visualize accuracy tradeoffs and to adjust the frequency interval where this accuracy is maximized.

To scale your model using the Scaling Tool, perform the following steps:

- Open Scaling Tool.
- Specify frequency axis limits.
- Specify frequency band for maximum accuracy.
- Save the scaling.

For an example of using the Scaling Tool on a real model, see “Scaling State-Space Models to Maximize Accuracy” on page 5-59.

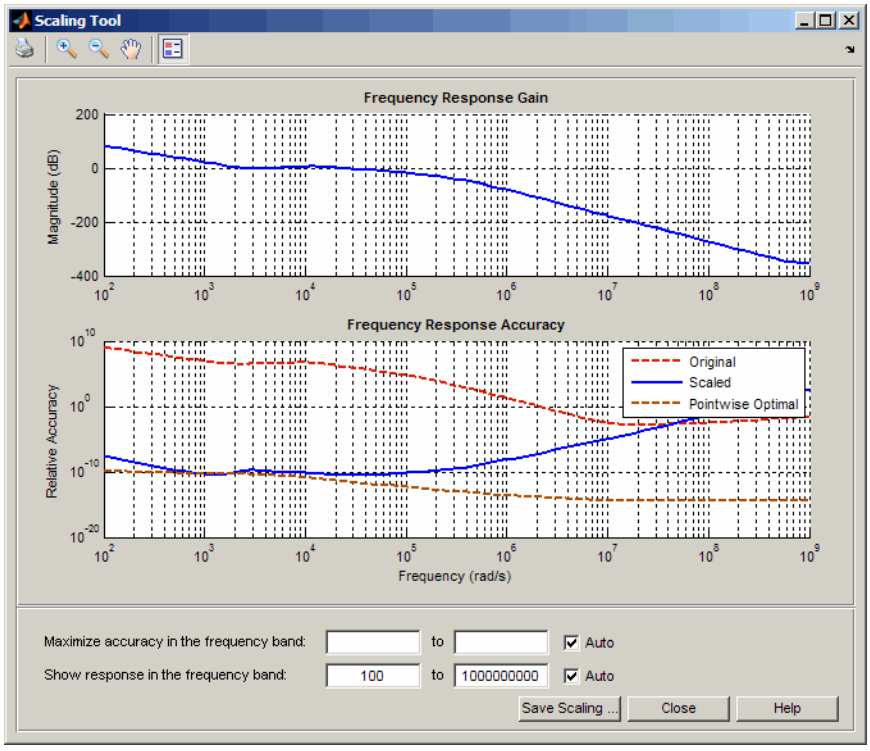
For more information about scaling models from the command line, see the `prescale` reference page.

Open the Scaling Tool

To open the Scaling Tool for a state-space model named `sys`, type

```
prescale(sys)
```

The Scaling Tool resembles one shown in the following figure.



The Scaling Tool contains the following plots:

- The **Frequency Response Gain** plot helps you determine the frequency band over which you want to maximize scaling.

  For SISO systems, this plot shows the gain of your model. For MIMO systems, the plot shows the principle gain (largest singular value) of your model.

- The **Frequency Response Accuracy** plot allows you to view the accuracy tradeoffs for your model when maximizing accuracy in a particular frequency bands.

  This plot shows the following information:

  - Relative accuracy of the response of the original unscaled model in red
  - Relative accuracy of the response of the scaled model in blue
  - Best achievable accuracy when using independent scaling at each frequency in brown

  When you compute some model characteristics, such as the frequency response or the system zeros, the software produces the exact answer for some perturbation of the model you specified. The *relative accuracy* is a measure of the worst-case relative gap between the frequency response of the original and perturbed models. The perturbation accounts for rounding errors during calculation. Any relative accuracy value greater than 1 implies poor accuracy.

  Tip If the blue Scaled curve is close to the brown Pointwise Optimal curve in a particular frequency band, you already have the best possible accuracy in that frequency band.

Specify Frequency Axis Limits

You can change the limits of the plot axis to view a particular frequency band of interest in the Scaling Tool. To view a particular frequency band, specify the band in the **Show response in the frequency band** fields.

This action updates the frequency axis of the Scaling tool to show the specified frequency band.

Tip To return to the default display, select the **Auto** check box.

Specify Frequency Band for Maximum Accuracy

To adjust the frequency band where you want maximum accuracy, set a new frequency band in the **Maximize accuracy in the frequency band** fields. You can visualize accuracy tradeoffs by trying out different frequency bands and viewing the resulting relative accuracy across the frequency band of interest.

Note You can use the **Frequency Response Gain** plot, which plots the gain of your model, to view the dynamics in your model to help determine the frequency band to maximize accuracy.

Each time you specify a new frequency band, the **Frequency Response Accuracy** plot updates with the result of the new scaling. Compare the Scaled curve (blue) to the Pointwise Optimal curve (brown) to determine where the new scaling is nearly optimal and where you need more accuracy.

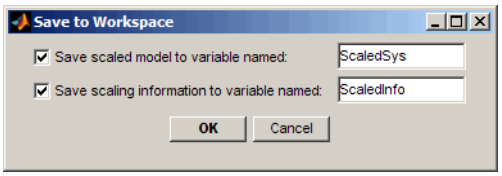
Tip To return to the default scaling, select the **Auto** check box.

Save the Scaling

When you find a good scaling for your model, save the scaled model as follows:

1. Click **Save Scaling**.

   This action opens the **Save to Workspace** dialog box.



2. In the **Save to Workspace** dialog box, verify that any of the following items you want to save are selected, and specify variable names for these items.

   - Scaled model
   - Scaling information, including:
     - Scaling factors
     - Frequencies used to test accuracy
     - Relative accuracy at each test frequency

   For details about the scaling information, see the `prescale` reference page.

3. Click **OK**.

   This action sets the State-Space (@ss) object `Scaled` property of your model to true. When you set this property to `True`, the Control System Toolbox algorithms skip the automated scaling of the model.

Linear System Analyzer

- “Linear System Analyzer Overview” on page 26-2
- “Using the Right-Click Menu in the Linear System Analyzer” on page 26-4
- “Importing, Exporting, and Deleting Models in the Linear System Analyzer” on page 26-8
- “Selecting Response Types” on page 26-11
- “Analyzing MIMO Models” on page 26-15
- “Customizing the Linear System Analyzer” on page 26-19

Linear System Analyzer Overview

The **Linear System Analyzer** app simplifies the analysis of linear, time-invariant systems. Use **Linear System Analyzer** to view and compare the response plots of SISO and MIMO systems, or of several linear models at the same time. You can generate time and frequency response plots to inspect key response parameters, such as rise time, maximum overshoot, and stability margins.

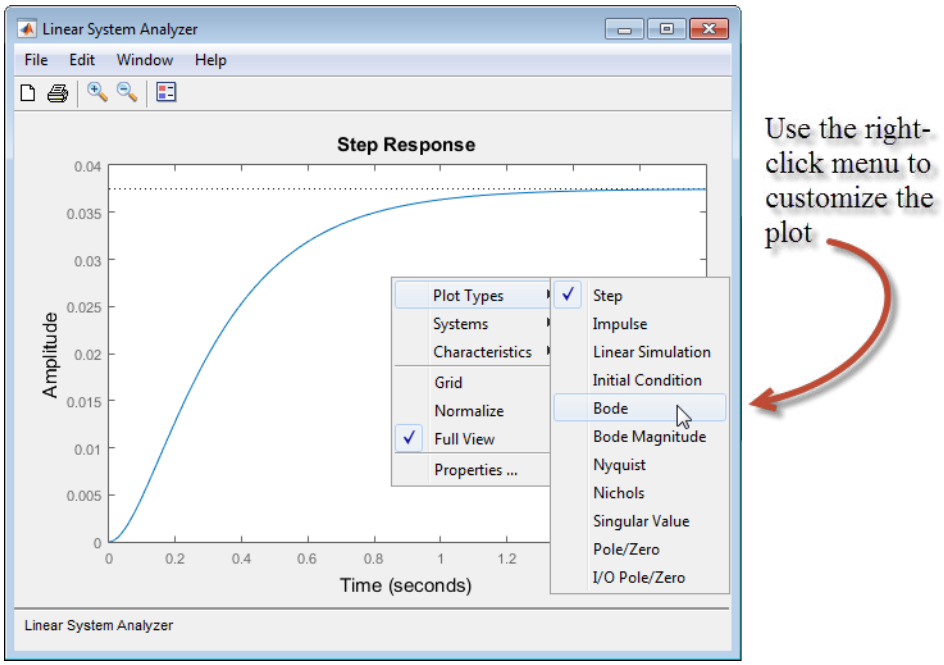
You can launch the **Linear System Analyzer** in two ways:

- Use the `linearSystemAnalyzer` command.
- In MATLAB, on the **Apps** tab under **Control System Design and Analysis**, click the app icon.

The easiest way to work with the **Linear System Analyzer** is to use the right-click menus. For example, type

```
load ltiexamples
linearSystemAnalyzer(sys_dc)
```

at the MATLAB prompt. The default plot is a step response.



The **Linear System Analyzer** can display up to six different plot types simultaneously, including step, impulse, Bode (magnitude and phase or magnitude only), Nyquist, Nichols, singular value, pole/zero, and I/O pole/zero.

For examples of how to use the **Linear System Analyzer**, see “Linear Analysis Using the Linear System Analyzer”. For more detailed information about **Linear System Analyzer** menus and options, see:

- “Using the Right-Click Menu in the Linear System Analyzer” on page 26-4
- “Importing, Exporting, and Deleting Models in the Linear System Analyzer” on page 26-8

- “Selecting Response Types” on page 26-11
- “Analyzing MIMO Models” on page 26-15
- “Customizing the Linear System Analyzer” on page 26-19

Using the Right-Click Menu in the Linear System Analyzer

Overview of the Right-Click Menu

The quickest way to manipulate views in the **Linear System Analyzer** is use the right-click menu. You can access several **Linear System Analyzer** controls and options, including:

- **Plot Type** — Changes the plot type
- **Systems** — Selects or deselects any of the models loaded in the **Linear System Analyzer**
- **Characteristics** — Displays key response characteristics and parameters
- **Grid** — Adds grids to your plot
- **Properties** — Opens the **Property Editor**, where you can customize plot attributes

In addition to right-click menus, all response plots include data markers. These allow you to scan the plot data, identify key data, and determine the source system for a given plot.

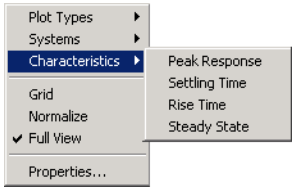
Setting Characteristics of Response Plots

The **Characteristics** menu changes for each plot response type. Characteristics refers to response plot information, such as peak response, or, in some cases, rise time, and settling time.

The next sections describe the menu items for each of the eight plot types.

Step Response

Step plots the model's response to a step input.



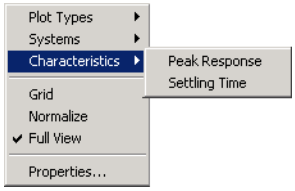
You can display the following information in the step response:

- **Peak Response** — The largest deviation from the steady-state value of the step response
- **Settling Time** — The time required for the step response to decline and stay at 5% of its final value
- **Rise Time** — The time require for the step response to rise from 10% to 90% of its final value
- **Steady-State** — The final value for the step response

> **Note** You can change the definitions of settling time and rise time using the **Options** pane of the "Toolbox Preferences Editor" on page 20-2, the "Linear System Analyzer Preferences Editor" on page 21-2, or the Property editor on page 22-2.

Impulse Response

Impulse Response plots the model's response to an impulse.

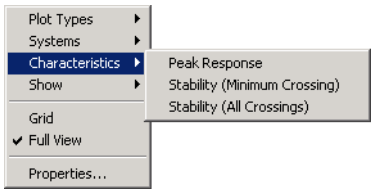


The **Linear System Analyzer** can display the following information in the impulse response:

- **Peak Response** — The maximum positive deviation from the steady-state value of the impulse response
- **Settling Time** — The time required for the step response to decline and stay at 5% of its final value

Bode Diagram

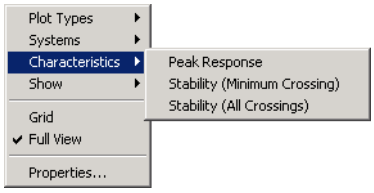
Bode plots the open-loop Bode phase and magnitude diagrams for the model.



The **Linear System Analyzer** can display the following information in the Bode diagram:

- **Peak Response** — The maximum value of the Bode magnitude plot over the specified region
- **Stability Margins (Minimum Crossing)** — The minimum phase and gain margins. The gain margin is defined to the gain (in dB) when the phase first crosses -180°. The phase margin is the distance, in degrees, of the phase from -180° when the gain magnitude is 0 dB.
- **Stability Margins (All Crossings)** — Display all stability margins

Bode Magnitude

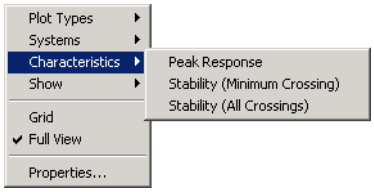


Bode Magnitude plots the Bode magnitude diagram for the model.

The **Linear System Analyzer** can display the following information in the Bode magnitude diagram:

- **Peak Response**, which is the maximum value of the Bode magnitude in decibels (dB), over the specified range of the diagram.
- **Stability (Minimum Crossing)** — The minimum gain margins. The gain margin is defined to the gain (in dB) when the phase first crosses -180°.
- **Stability (All Crossings)** — Display all gain stability margins

Nyquist Diagrams



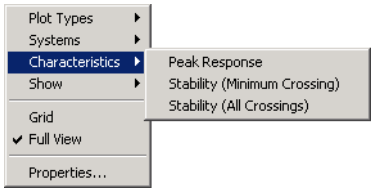
Nyquist plots the Nyquist diagram for the model.

The **Linear System Analyzer** can display the following types of information in the Nyquist diagram:

- **Peak Response** — The maximum value of the Nyquist diagram over the specified region
- **Stability (Minimum Crossing)** — The minimum gain and phase margins for the Nyquist diagram. The gain margin is the distance from the origin to the phase crossover of the Nyquist curve. The phase crossover is where the curve meets the real axis. The phase margin is the angle subtended by the real axis and the gain crossover on the circle of radius 1.
- **Stability (All Crossings)** — Display all gain stability margins

Nichols Charts

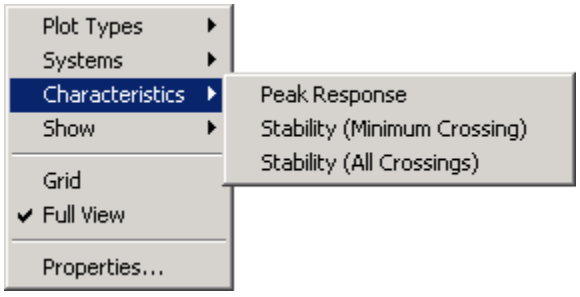
Nichols plots the Nichols Chart for the model.



The **Linear System Analyzer** can display the following types of information in the Nichols chart:

- **Peak Response** — The maximum value of the Nichols chart in the plotted region.
- **Stability (Minimum Crossing)** — The minimum gain and phase margins for the Nichols chart.
- **Stability (All Crossings)** — Display all gain stability margins

Singular Values



Singular Values plots the singular values for the model.

The **Linear System Analyzer** can display the **Peak Response**, which is the largest magnitude of the Singular Values curve over the plotted region.

Pole/Zero and I/O Pole/Zero

Pole/Zero plots the poles and zeros of the model with `x' for poles and `o' for zeros. I/O Pole/Zero plots the poles and zeros of I/O pairs.

There are no **Characteristics** available for pole-zero plots.

See Also

Linear System Analyzer

Related Examples

- “Joint Time-Domain and Frequency-Domain Analysis” on page 7-32

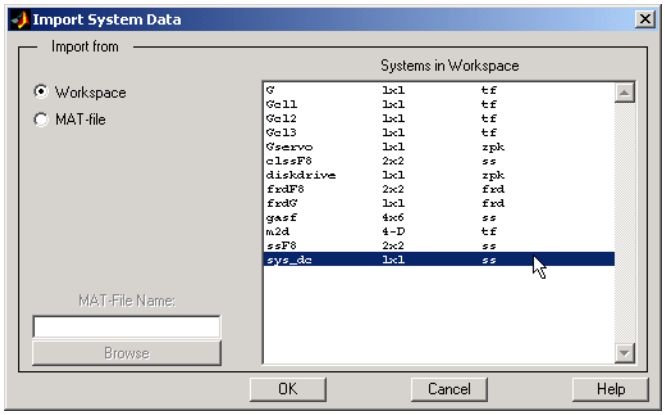
More About

- “Linear System Analyzer Overview” on page 26-2

Importing, Exporting, and Deleting Models in the Linear System Analyzer

Importing Models

To import models into the **Linear System Analyzer**, select **File > Import**. The **Import System Data** dialog box opens, as shown below.



Use the **Import System Data** dialog box to import LTI models into or from the **Linear System Analyzer** workspace.

To import a model:

- Click on the desired model in the LTI Browser List. To perform multiple selections:
  - Hold the Control key and click on nonadjacent models.
  - Hold the Shift key while clicking to select multiple adjacent models.
- Click the **OK** or **Apply** Button

Note that the **LTI Browser** lists only the LTI models in the MATLAB workspace.

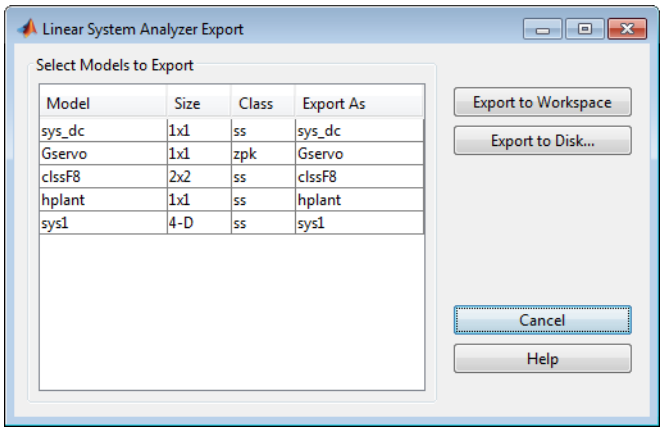
Alternatively, you can directly import a model into the **Linear System Analyzer** using the `linearSystemAnalyzer` function, as in

```
linearSystemAnalyzer({'step','bode'},modelname)
```

See the **Linear System Analyzer** reference page for more information.

Exporting Models

Use **Export** in the **File** menu to open the **Linear System Analyzer Export** window, shown below.

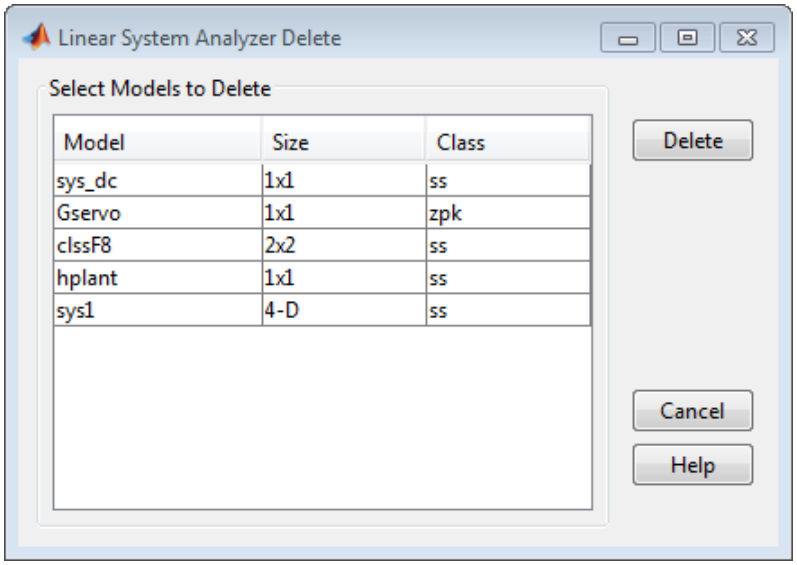


The **Linear System Analyzer Export** window lists all the models with responses currently displayed in your **Linear System Analyzer**. You can export models back to the MATLAB workspace or to disk.

To export single or multiple models, follow the steps described in the importing models section above. To save your models to disk in a MAT-file, choose **Export to Disk**.

Deleting Models

To remove models from the **Linear System Analyzer** workspace, select **Edit** > **Delete Systems**. The **Linear System Analyzer Delete** dialog box opens.



To delete a model:

- Click on the desired model in the Model list. To perform multiple selections:
  - **a** Click and drag over several variables in the list.
  - **b** Hold the Control key and click on individual variables.
  - **c** Hold the Shift key while clicking, to select a range.

Click the **Delete** button.

See Also

Linear System Analyzer

More About

- “Linear System Analyzer Overview” on page 26-2

Selecting Response Types

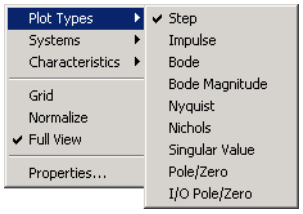
Methods for Selecting Response Types

There are two methods for selecting response plots in the **Linear System Analyzer**:

- Selecting **Plot Type** from the right-click menus
- Opening the **Plot Configurations** window

Right Click Menu: Plot Type

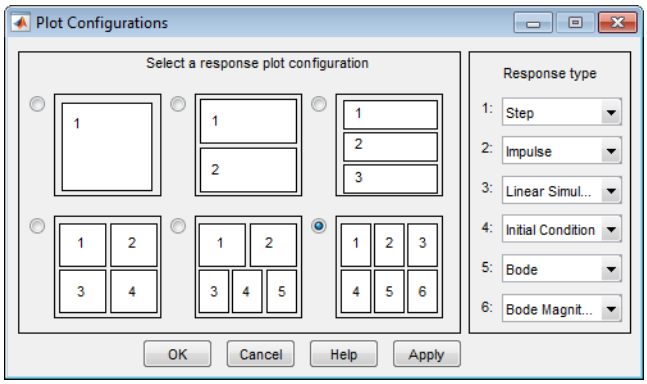
If you have a plot open in the **Linear System Analyzer**, you can switch to any other response plot available by selecting **Plot Type** from the right click menu.



To change the response plot, select the new plot type from the **Plot Type** submenu. The **Linear System Analyzer** automatically displays the new response plot.

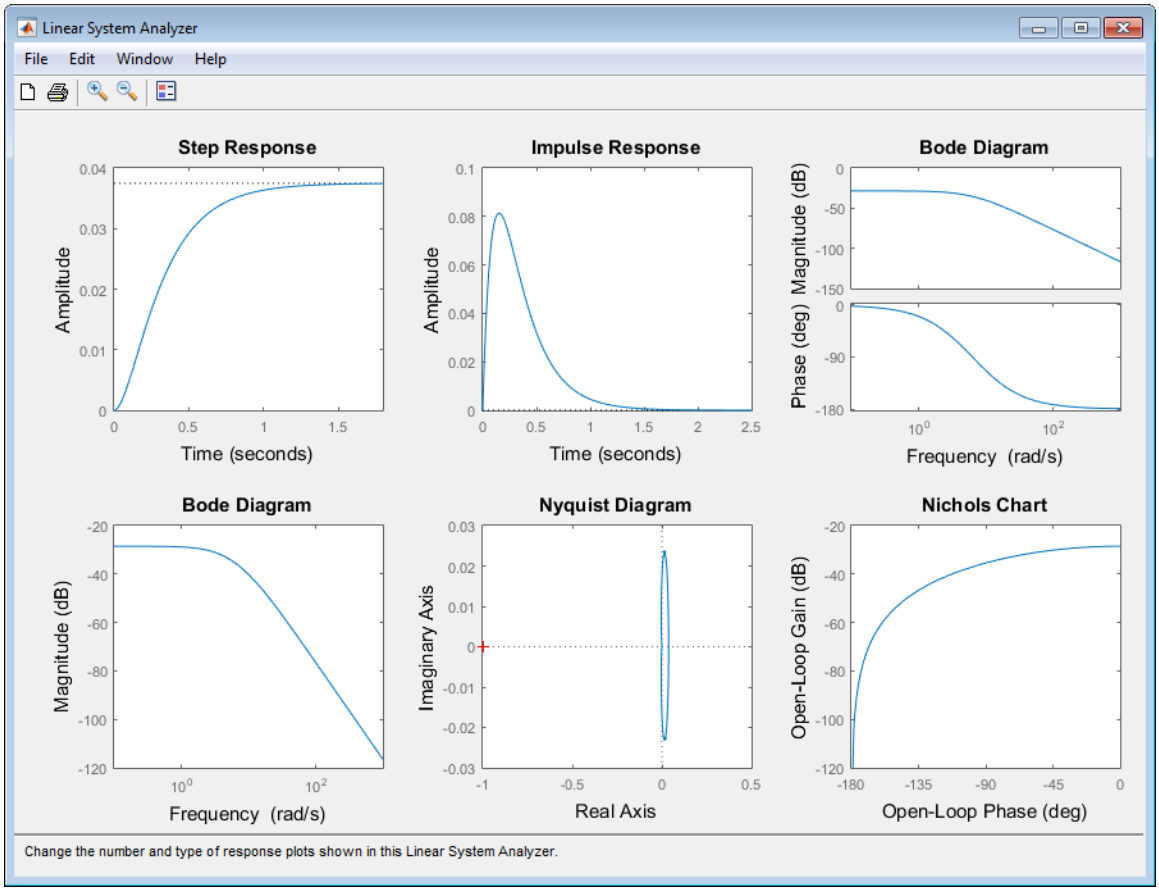
Plot Configurations Window

The Plot Type feature of the right-click menu works on existing plots, but you can also add plots to a **Linear System Analyzer** by using the **Plot Configurations** window. By default, the **Linear System Analyzer** opens with a closed-loop step response. To reconfigure an open viewer, select **Plot Configuration** in the **Edit** menu.



Use the radio buttons to select the number of plots you want displayed in your **Linear System Analyzer**. For each plot, select a response type from the menus located on the right-hand side of the window.

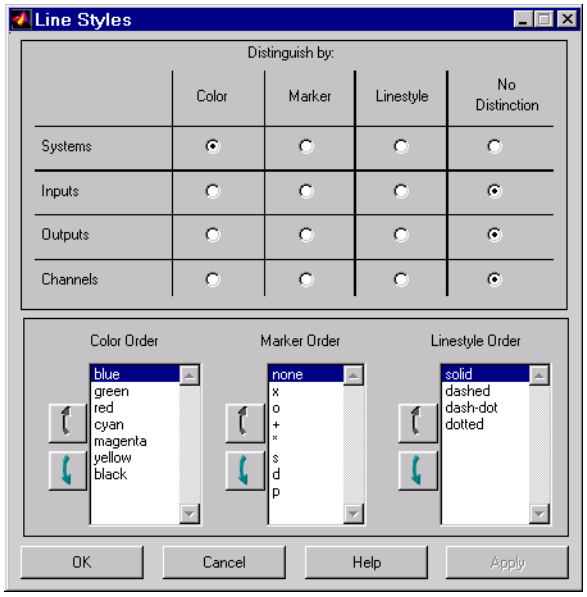
It's possible to configure a single **Linear System Analyzer** to contain up to six response plots.



Available response plots include: step, impulse, Bode (magnitude and phase, or magnitude only), Nyquist, Nichols, sigma, pole/zero maps, and I/O pole/zero maps.

Line Styles Editor

Select **Edit > Line Styles** to open the **Line Styles** editor.



The **Line Styles** editor is particularly useful when you have multiple systems imported. You can use it change line colors, add and rearrange markers, and alter line styes (solid, dashed, and so on).

You can use the **Linestyle Preferences** window to customize the appearance of the response plots by specifying:

- The line property used to distinguish different systems, inputs, or outputs
- The order in which these line properties are applied

Each **Linear System Analyzer** has its own **Linestyle Preferences** window.

Setting Preferences

You can use the "Distinguish by" matrix (the top half of the window) to specify the line property that will vary throughout the response plots. You can group multiple plot curves by systems, inputs, outputs, or channels (individual input/output relationships). Note that the Line Styles editor uses radio buttons, which means that you can only assign one property setting for each grouping (system, input, etc.).

Ordering Properties

The **Order** field allows you to change the default property order used when applying the different line properties. You can reorder the colors, markers, and linestyles (e.g., solid or dashed).

To change any of the property orders, click the up or down arrow button to the left of the associated property list to move the selected property up or down in the list.

See Also

Linear System Analyzer

Related Examples

- “Joint Time-Domain and Frequency-Domain Analysis” on page 7-32

More About

- “Linear System Analyzer Overview” on page 26-2

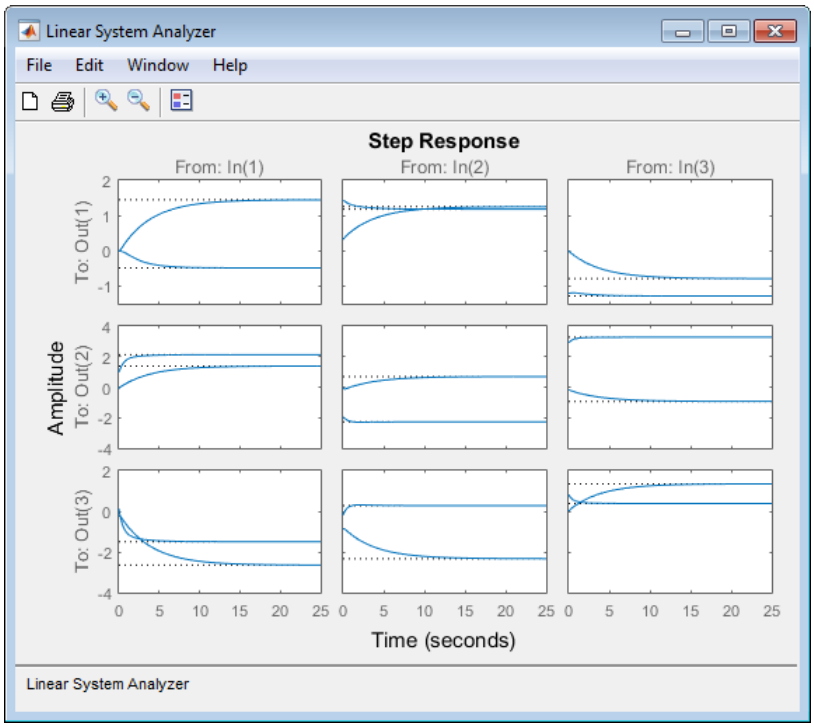
Analyzing MIMO Models

Overview of Analyzing MIMO Models

If you plot a MIMO system, or an LTI array containing multiple linear models, you can use special features of the right-click menu to group the response plots by input/output (I/O) pairs, or select individual plots for display. For example, generate an array of two random 3-input, 3-output MIMO systems and view them in the Linear System Analyzer:

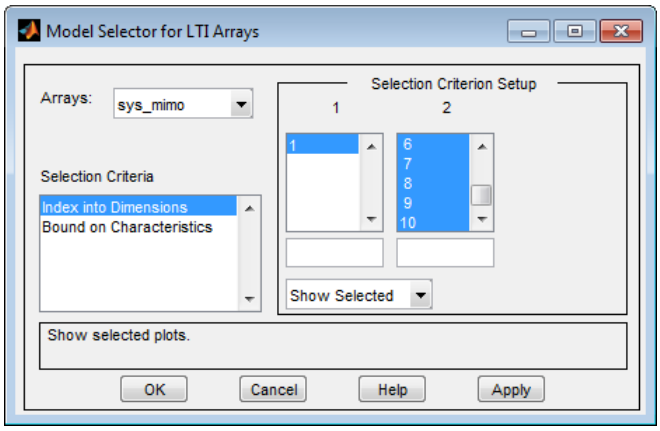
```
sys_mimo=stack(1,rss(3,3,3),rss(3,3,3));
linearSystemAnalyzer(sys_mimo);
```

A set of 9 plots appears, one from each input to each output, each showing the step responses of the corresponding I/Os of both models in the array.



Array Selector

If you import an LTI model array into the Linear System Analyzer, **Array Selector** appears as an option in the right-click menu. Selecting this option opens the **Model Selector for LTI Arrays**, shown below.



You can use this window to include or exclude models within the LTI array using various criteria.

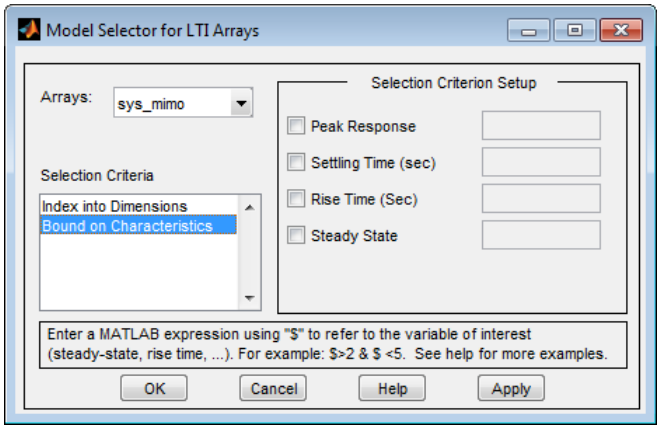
Arrays

Select the LTI array for model selection using the **Arrays** list.

Selection Criteria

There are two selection criteria. The default, **Index into Dimensions**, allows you to include or exclude specified indices of the LTI Array. Select systems from the **Selection Criterion Setup** section of the dialog box. Then, Specify whether to show or hide the systems using the pull-down menu below the Setup lists.

The second criterion is **Bound on Characteristics**. Selecting this options causes the Model Selector to reconfigure. The reconfigured window is shown below

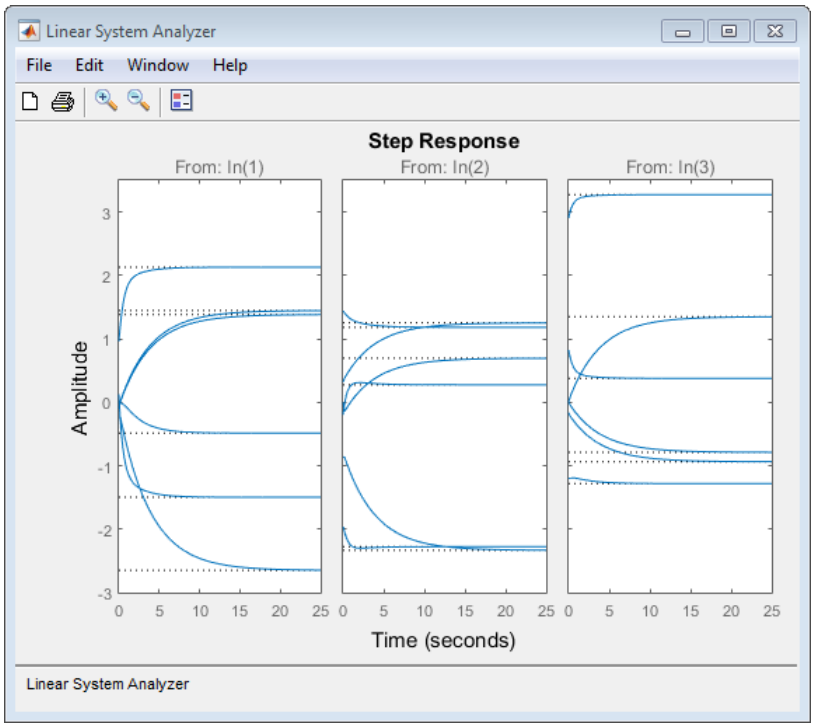


Use this option to select systems for inclusion or exclusion in your Linear System Analyzer based on their time response characteristics. The panel directly above the buttons describes how to set the inclusion or exclusion criteria based on which selection criteria you select from the reconfigured **Selection Criteria Setup** panel.

I/O Grouping for MIMO Models

You can group the plots by inputs, by outputs, or both by selecting **I/O Grouping** from the right-click menu, and then selecting **Inputs**, **Outputs**, or **All**.

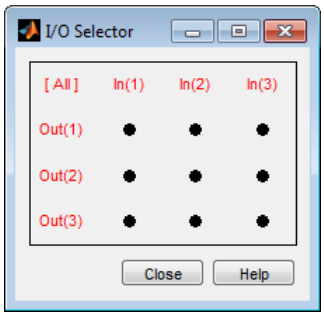
For example, if you select **Outputs**, the step plot reconfigures into 3 plots, grouping all the outputs together on each plot. Each plot now displays the responses from one of the inputs to all of the MIMO system's outputs, for all of the models in the array.



Selecting **None** returns to the default configuration, where all I/O pairs are displayed individually.

Selecting I/O Pairs

Another way to organize MIMO system information is to choose **I/O Selector** from the right-click menu, which opens the **I/O Selector** window.



This window automatically configures to the number of I/O pairs in your MIMO system. You can select:

- Any individual plot (only one at a time) by clicking on a button
- Any row or column by clicking on Y(*) or U(*)

- All of the plots by clicking [all]

Using these options, you can inspect individual I/O pairs, or look at particular I/O channels in detail.

See Also

Linear System Analyzer

More About

- “Model Arrays” on page 2-76

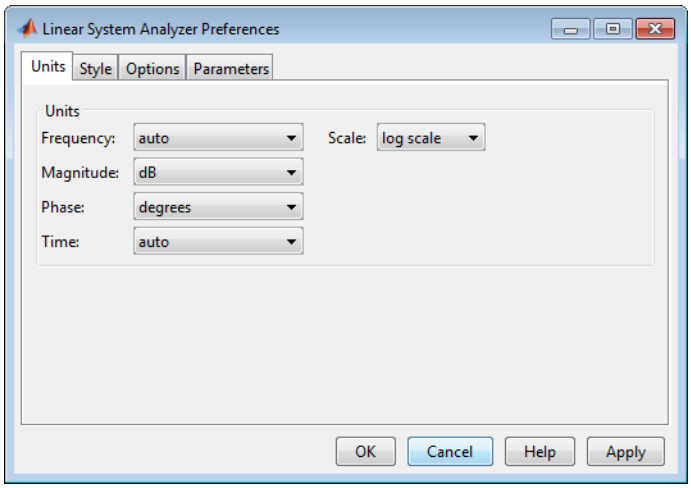
Customizing the Linear System Analyzer

Overview of Customizing the Linear System Analyzer

The **Linear System Analyzer** has a tool preferences editor, which allows you to set default characteristics for specific instances of **Linear System Analyzer**. If you open a new instance of either, each defaults to the characteristics specified in the **Toolbox Preferences** editor.

Linear System Analyzer Preferences Editor

Select **Edit > Linear System Analyzer Preferences** to open the preferences editor.



The **Linear System Analyzer Preferences** editor contains four panes:

- Units — Convert between various units, including rad/sec and Hertz
- Style — Customize grids, fonts, and colors
- Characteristics — Specify response plot characteristics, such as settling time tolerance
- Parameters — Set time and frequency ranges, stop times, and time step size

For more information about using the options in these panes in an instance of the **Linear System Analyzer**, see “Linear System Analyzer Preferences Editor” on page 21-2.

If you want to customize the settings for all instances of **Linear System Analyzer**, see “Toolbox Preferences Editor” on page 20-2.

See Also

Linear System Analyzer

More About

- “Linear System Analyzer Overview” on page 26-2