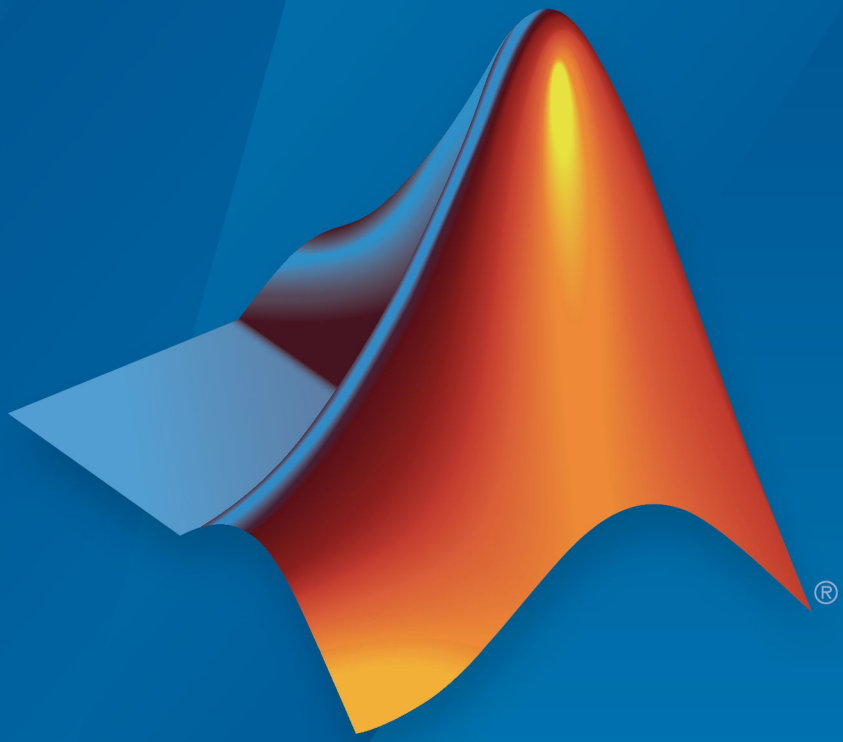
System Identification Toolbox™

User's Guide

Lennart Ljung



MATLAB® & SIMULINK®

R2017b

MathWorks®

How to Contact MathWorks

Latest news: www.mathworks.com

Sales and services: www.mathworks.com/sales_and_services

User community: www.mathworks.com/matlabcentral

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

Phone: 508-647-7000

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

System Identification Toolbox™ User's Guide

© COPYRIGHT 1988–2017 by The MathWorks, Inc.

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

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

Trademarks

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

Patents

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

Revision History

April 1988	First printing	
July 1991	Second printing	
May 1995	Third printing	
November 2000	Fourth printing	Revised for Version 5.0 (Release 12)
April 2001	Fifth printing	
July 2002	Online only	Revised for Version 5.0.2 (Release 13)
June 2004	Sixth printing	Revised for Version 6.0.1 (Release 14)
March 2005	Online only	Revised for Version 6.1.1 (Release 14SP2)
September 2005	Seventh printing	Revised for Version 6.1.2 (Release 14SP3)
March 2006	Online only	Revised for Version 6.1.3 (Release 2006a)
September 2006	Online only	Revised for Version 6.2 (Release 2006b)
March 2007	Online only	Revised for Version 7.0 (Release 2007a)
September 2007	Online only	Revised for Version 7.1 (Release 2007b)
March 2008	Online only	Revised for Version 7.2 (Release 2008a)
October 2008	Online only	Revised for Version 7.2.1 (Release 2008b)
March 2009	Online only	Revised for Version 7.3 (Release 2009a)
September 2009	Online only	Revised for Version 7.3.1 (Release 2009b)
March 2010	Online only	Revised for Version 7.4 (Release 2010a)
September 2010	Online only	Revised for Version 7.4.1 (Release 2010b)
April 2011	Online only	Revised for Version 7.4.2 (Release 2011a)
September 2011	Online only	Revised for Version 7.4.3 (Release 2011b)
March 2012	Online only	Revised for Version 8.0 (Release 2012a)
September 2012	Online only	Revised for Version 8.1 (Release 2012b)
March 2013	Online only	Revised for Version 8.2 (Release 2013a)
September 2013	Online only	Revised for Version 8.3 (Release 2013b)
March 2014	Online only	Revised for Version 9.0 (Release 2014a)
October 2014	Online only	Revised for Version 9.1 (Release 2014b)
March 2015	Online only	Revised for Version 9.2 (Release 2015a)
September 2015	Online only	Revised for Version 9.3 (Release 2015b)
March 2016	Online only	Revised for Version 9.4 (Release 2016a)
September 2016	Online only	Revised for Version 9.5 (Release 2016b)
March 2017	Online only	Revised for Version 9.6 (Release 2017a)
September 2017	Online only	Revised for Version 9.7 (Release 2017b)

Contents

1 Choosing Your System Identification Approach

Acknowledgments 1-2

What Are Model Objects? 1-3
Model Objects Represent Linear Systems 1-3
About Model Data 1-3

Types of Model Objects 1-5

Dynamic System Models 1-8

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

About Identified Linear Models 1-13
What are IDLTI Models? 1-13
Measured and Noise Component Parameterizations 1-14
Linear Model Estimation 1-17

Linear Model Structures 1-20
About System Identification Toolbox Model Objects 1-20
When to Construct a Model Structure Independently of Estimation 1-21
Commands for Constructing Linear Model Structures 1-21
Model Properties 1-22
See Also 1-24

Available Linear Models 1-25

Estimation Report 1-29
What is an Estimation Report? 1-29

Access Estimation Report 1-30
Compare Estimated Models Using Estimation Report 1-31
Analyze and Refine Estimation Results Using Estimation Report 1-32

Imposing Constraints on Model Parameter Values 1-34

Recommended Model Estimation Sequence 1-36

Supported Models for Time- and Frequency-Domain Data . 1-38
Supported Models for Time-Domain Data 1-38
Supported Models for Frequency-Domain Data 1-39
See Also 1-40

Supported Continuous- and Discrete-Time Models 1-41

Model Estimation Commands 1-44

Modeling Multiple-Output Systems 1-45
About Modeling Multiple-Output Systems 1-45
Modeling Multiple Outputs Directly 1-45
Modeling Multiple Outputs as a Combination of Single-Output Models 1-46
Improving Multiple-Output Estimation Results by Weighing Outputs During Estimation 1-46

Regularized Estimates of Model Parameters 1-48
What Is Regularization? 1-48
When to Use Regularization 1-51
Choosing Regularization Constants 1-53

Estimate Regularized ARX Model Using System Identification App 1-57

Loss Function and Model Quality Metrics 1-64
What is a Loss Function? 1-64
Options to Configure the Loss Function 1-65
Model Quality Metrics 1-73

Regularized Identification of Dynamic Systems 1-77

2 Data Import and Processing

Supported Data 2-3

Ways to Obtain Identification Data 2-5

Ways to Prepare Data for System Identification 2-6

Requirements on Data Sampling 2-8

Representing Data in MATLAB Workspace 2-9
- Time-Domain Data Representation 2-9
- Time-Series Data Representation 2-10
- Frequency-Domain Data Representation 2-10

Import Time-Domain Data into the App 2-16

Import Frequency-Domain Data into the App 2-19
- Importing Frequency-Domain Input/Output Signals into the App 2-19
- Importing Frequency-Response Data into the App 2-21

Import Data Objects into the App 2-25

Specifying the Data Sample Time 2-28

Specify Estimation and Validation Data in the App 2-30

Preprocess Data Using Quick Start 2-32

Create Data Sets from a Subset of Signal Channels 2-33

Create Multiexperiment Data Sets in the App 2-35
- Why Create Multiexperiment Data? 2-35
- Limitations on Data Sets 2-35
- Merging Data Sets 2-35
- Extracting Specific Experiments from a Multiexperiment Data Set into a New Data Set 2-39

Managing Data in the App 2-42
- Viewing Data Properties 2-42

Renaming Data and Changing Display Color 2-43
Distinguishing Data Types 2-46
Organizing Data Icons 2-46
Deleting Data Sets 2-47
Exporting Data to the MATLAB Workspace 2-48

Representing Time- and Frequency-Domain Data Using iddata Objects 2-50
iddata Constructor 2-50
iddata Properties 2-52
Select Data Channels, I/O Data and Experiments in iddata Objects 2-55
Increasing Number of Channels or Data Points of iddata Objects 2-58

Create Multiexperiment Data at the Command Line 2-61
Why Create Multiexperiment Data Sets? 2-61
Limitations on Data Sets 2-61
Entering Multiexperiment Data Directly 2-61
Merging Data Sets 2-62
Adding Experiments to an Existing iddata Object 2-62

Dealing with Multi-Experiment Data and Merging Models 2-64

Managing iddata Objects 2-79
Modifying Time and Frequency Vectors 2-79
Naming, Adding, and Removing Data Channels 2-82
Subreferencing iddata Objects 2-84
Concatenating iddata Objects 2-84

Representing Frequency-Response Data Using idfrd Objects 2-85
idfrd Constructor 2-85
idfrd Properties 2-86
Select I/O Channels and Data in idfrd Objects 2-87
Adding Input or Output Channels in idfrd Objects 2-88
Managing idfrd Objects 2-90
Operations that Create idfrd Objects 2-90

Is Your Data Ready for Modeling? 2-92

How to Plot Data in the App **2-93**
How to Plot Data in the App **2-93**
Manipulating a Time Plot **2-95**
Manipulating Data Spectra Plot **2-96**
Manipulating a Frequency Function Plot **2-98**

How to Plot Data at the Command Line **2-100**

How to Analyze Data Using the advice Command **2-102**

Select Subsets of Data **2-104**
Why Select Subsets of Data? **2-104**
Extract Subsets of Data Using the App **2-104**
Extract Subsets of Data at the Command Line **2-106**

Handling Missing Data and Outliers **2-108**
Handling Missing Data **2-108**
Handling Outliers **2-109**
See Also **2-110**

Extract and Model Specific Data Segments **2-111**

Handling Offsets and Trends in Data **2-114**
When to Detrend Data **2-114**
Alternatives for Detrending Data in App or at the Command-Line **2-115**
Next Steps After Detrending **2-116**

How to Detrend Data Using the App **2-117**

How to Detrend Data at the Command Line **2-118**
Detrending Steady-State Data **2-118**
Detrending Transient Data **2-118**

Resampling Data **2-120**
What Is Resampling? **2-120**
Resampling Data Without Aliasing Effects **2-121**

Resampling Data Using the App **2-125**

Resampling Data at the Command Line **2-127**

Filtering Data **2-129**
Supported Filters **2-129**
Choosing to Prefilter Your Data **2-129**

How to Filter Data Using the App **2-131**
Filtering Time-Domain Data in the App **2-131**
Filtering Frequency-Domain or Frequency-Response Data in the App **2-132**

How to Filter Data at the Command Line **2-134**
Simple Passband Filter **2-134**
Defining a Custom Filter **2-135**
Causal and Noncausal Filters **2-136**

Generate Data Using Simulation **2-137**
Commands for Generating Data Using Simulation **2-137**
Create Periodic Input Data **2-138**
Generate Output Data Using Simulation **2-140**
Simulating Data Using Other MathWorks Products **2-142**

Manipulating Complex-Valued Data **2-143**
Supported Operations for Complex Data **2-143**
Processing Complex iddata Signals at the Command Line . **2-143**

3 Transform Data

Supported Data Transformations **3-2**

Transform Time-Domain Data in the App **3-4**

Transform Frequency-Domain Data in the App **3-6**

Transform Frequency-Response Data in the App **3-8**

Transforming Between Time and Frequency-Domain Data **3-11**

Transforming Between Frequency-Domain and Frequency-Response Data **3-12**

4 Linear Model Identification

Black-Box Modeling 4-2
Selecting Black-Box Model Structure and Order 4-2
When to Use Nonlinear Model Structures? 4-4
Black-Box Estimation Example 4-4

Refine Linear Parametric Models 4-7
When to Refine Models 4-7
What You Specify to Refine a Model 4-7
Refine Linear Parametric Models Using System Identification App 4-7
Refine Linear Parametric Models at the Command Line 4-9

Refine ARMAX Model with Initial Parameter Guesses at Command Line 4-10

Refine Initial ARMAX Model at Command Line 4-12

Extracting Numerical Model Data 4-14

Transforming Between Discrete-Time and Continuous-Time Representations 4-17
Why Transform Between Continuous and Discrete Time? . . . 4-17
Using the c2d, d2c, and d2d Commands 4-17
Specifying Intersample Behavior 4-19
Effects on the Noise Model 4-19

Continuous-Discrete Conversion Methods 4-21
Choosing a Conversion Method 4-21
Zero-Order Hold 4-22
First-Order Hold 4-23
Impulse-Invariant Mapping 4-24
Tustin Approximation 4-25
Zero-Pole Matching Equivalents 4-29

Effect of Input Intersample Behavior on Continuous-Time Models 4-31

Transforming Between Linear Model Representations 4-35

Subreferencing Models 4-38
What Is Subreferencing? 4-38
Limitation on Supported Models 4-38
Subreferencing Specific Measured Channels 4-38
Separation of Measured and Noise Components of Models . . 4-39

Treating Noise Channels as Measured Inputs 4-41

Concatenating Models 4-43
About Concatenating Models 4-43
Limitation on Supported Models 4-43
Horizontal Concatenation of Model Objects 4-44
Vertical Concatenation of Model Objects 4-44
Concatenating Noise Spectrum Data of idfrd Objects 4-45
See Also 4-46

Merging Models 4-47

Determining Model Order and Delay 4-48

Model Structure Selection: Determining Model Order and Input Delay 4-49

Frequency Domain Identification: Estimating Models Using Frequency Domain Data 4-65

Building Structured and User-Defined Models Using System Identification Toolbox™ 4-92

5 Identifying Process Models

What Is a Process Model? 5-2

Data Supported by Process Models 5-4

Estimate Process Models Using the App 5-5
Assigning Estimation Weightings 5-9
Next Steps 5-10

Estimate Process Models at the Command Line **5-11**
Prerequisites **5-11**
Using procest to Estimate Process Models **5-11**
Estimate Process Models with Free Parameters **5-12**
Estimate Process Models with Fixed Parameters **5-14**

Building and Estimating Process Models Using System Identification Toolbox™ **5-18**

Process Model Structure Specification **5-44**

Estimating Multiple-Input, Multi-Output Process Models . . **5-46**

Disturbance Model Structure for Process Models **5-47**

Specifying Initial Conditions for Iterative Estimation Algorithms **5-49**

6 Identifying Input-Output Polynomial Models

What Are Polynomial Models? **6-2**
Polynomial Model Structure **6-2**
Understanding the Time-Shift Operator q **6-3**
Different Configurations of Polynomial Models **6-4**
Continuous-Time Representation of Polynomial Models **6-6**
Multi-Output Polynomial Models **6-6**

Data Supported by Polynomial Models **6-8**
Types of Supported Data **6-8**
Designating Data for Estimating Continuous-Time Models . . . **6-8**
Designating Data for Estimating Discrete-Time Models **6-9**

Preliminary Step – Estimating Model Orders and Input Delays **6-10**
Why Estimate Model Orders and Delays? **6-10**
Estimating Orders and Delays in the App **6-10**
Estimating Model Orders at the Command Line **6-13**
Estimating Delays at the Command Line **6-14**
Selecting Model Orders from the Best ARX Structure **6-15**

Estimate Polynomial Models in the App **6-18**
Assigning Estimation Weightings **6-21**
Next Steps **6-22**

Estimate Polynomial Models at the Command Line **6-23**
Using arx and iv4 to Estimate ARX Models **6-23**
Using polyest to Estimate Polynomial Models **6-24**

Polynomial Sizes and Orders of Multi-Output Polynomial Models **6-27**

Specifying Initial States for Iterative Estimation Algorithms **6-32**

Polynomial Model Estimation Algorithms **6-34**

Estimate Models Using armax **6-35**

7 Identifying State-Space Models

What Are State-Space Models? **7-2**
Definition of State-Space Models **7-2**
Continuous-Time Representation **7-2**
Discrete-Time Representation **7-3**
Relationship Between Continuous-Time and Discrete-Time State Matrices **7-4**
State-Space Representation of Transfer Functions **7-4**

Data Supported by State-Space Models **7-6**

Supported State-Space Parameterizations **7-7**

Estimate State-Space Model With Order Selection **7-8**
Estimate Model With Selected Order in the App **7-8**
Estimate Model With Selected Order at the Command Line **7-11**
Using the Model Order Selection Window **7-11**

Estimate State-Space Models in System Identification App **7-13**
Assigning Estimation Weightings **7-21**

Estimate State-Space Models at the Command Line **7-22**
Black Box vs. Structured State-Space Model Estimation **7-22**
Estimating State-Space Models Using ssest, ssregest and n4sid **7-23**
Choosing the Structure of A, B, C Matrices **7-24**
Choosing Between Continuous-Time and Discrete-Time Representations **7-24**
Choosing to Estimate D, K, and X0 Matrices **7-25**

Estimate State-Space Models with Free-Parameterization **7-29**

Estimate State-Space Models with Canonical Parameterization **7-30**
What Is Canonical Parameterization? **7-30**
Estimating Canonical State-Space Models **7-30**

Estimate State-Space Models with Structured Parameterization **7-32**
What Is Structured Parameterization? **7-32**
Specify the State-Space Model Structure **7-33**
Are Grey-Box Models Similar to State-Space Models with Structured Parameterization? **7-34**
Estimate Structured Discrete-Time State-Space Models **7-35**
Estimate Structured Continuous-Time State-Space Models . **7-37**

Estimate State-Space Equivalent of ARMAX and OE Models **7-40**

Specifying Initial States for Iterative Estimation Algorithms **7-42**

State-Space Model Estimation Methods **7-43**
References **7-43**

8 Identifying Transfer Function Models

What are Transfer Function Models? **8-2**
Definition of Transfer Function Models **8-2**
Continuous-Time Representation **8-2**
Discrete-Time Representation **8-2**
Delays **8-3**
Multi-Input Multi-Output Models **8-3**

Data Supported by Transfer Function Models **8-5**

Estimate Transfer Function Models in the System Identification App **8-6**

Estimate Transfer Function Models at the Command Line **8-13**

Transfer Function Structure Specification **8-14**

Estimate Transfer Function Models by Specifying Number of Poles **8-15**

Estimate Transfer Function Models with Transport Delay to Fit Given Frequency-Response Data **8-16**

Estimate Transfer Function Models With Prior Knowledge of Model Structure and Constraints **8-17**

Estimate Transfer Function Models with Unknown Transport Delays **8-19**

Estimate Transfer Functions with Delays **8-21**

Specifying Initial Conditions for Iterative Estimation Algorithms **8-22**

Troubleshoot Frequency-Domain Identification of Transfer Function Models **8-23**

9 Identifying Frequency-Response Models

What is a Frequency-Response Model? 9-2

Data Supported by Frequency-Response Models 9-4

Estimate Frequency-Response Models in the App 9-5

Estimate Frequency-Response Models at the Command Line 9-7

Selecting the Method for Computing Spectral Models 9-9

Controlling Frequency Resolution of Spectral Models 9-11
- What Is Frequency Resolution? 9-11
- Frequency Resolution for etfe and spa 9-11
- Frequency Resolution for spafdr 9-12
- etfe Frequency Resolution for Periodic Input 9-12

Spectrum Normalization 9-14

10 Identifying Impulse-Response Models

What Is Time-Domain Correlation Analysis? 10-2

Data Supported by Correlation Analysis 10-3

Estimate Impulse-Response Models Using System Identification App 10-4
- Next Steps 10-5

Estimate Impulse-Response Models at the Command Line 10-6
- Next Steps 10-7

Compute Response Values 10-8

Identify Delay Using Transient-Response Plots **10-9**

Correlation Analysis Algorithm **10-12**

11 Nonlinear Black-Box Model Identification

About Identified Nonlinear Models **11-2**
What Are Nonlinear Models? **11-2**
When to Fit Nonlinear Models **11-2**
Nonlinear Model Estimation **11-4**

Nonlinear Model Structures **11-7**
About System Identification Toolbox Model Objects **11-7**
When to Construct a Model Structure Independently of Estimation **11-8**
Commands for Constructing Nonlinear Model Structures . . . **11-8**
Model Properties **11-9**

Available Nonlinear Models **11-12**
Overview **11-12**
Nonlinear ARX Models **11-12**
Hammerstein-Wiener Models **11-13**
Nonlinear Grey-Box Models **11-13**

Preparing Data for Nonlinear Identification **11-15**

What are Nonlinear ARX Models? **11-17**
Nonlinear ARX Model Extends the Linear ARX Structure . **11-17**
Structure of Nonlinear ARX Models **11-18**
Nonlinear ARX Model Orders and Delay **11-19**

Identifying Nonlinear ARX Models **11-21**
Prepare Data for Identification **11-21**
Configure Nonlinear ARX Model Structure **11-22**
Specify Estimation Options for Nonlinear ARX Models **11-23**
Initialize Nonlinear ARX Estimation Using Linear Model . . **11-25**

Available Nonlinearity Estimators for Nonlinear ARX Models **11-27**

Estimate Nonlinear ARX Models in the App **11-29**

Estimate Nonlinear ARX Models at the Command Line . . . **11-32**
Estimate Model Using nlarx **11-32**
Configure Model Regressors **11-33**
Specify Linear and Nonlinear Regressors **11-36**
Configure Nonlinearity Estimator **11-37**
Iteratively Refine Model **11-40**
Troubleshoot Estimation **11-41**
Use nlarx to Estimate Nonlinear ARX Models **11-41**

Estimate Nonlinear ARX Models Initialized Using Linear ARX Models **11-44**

Validate Nonlinear ARX Models **11-49**
Compare Model Output to Measured Output **11-49**
Check Iterative Search Termination Conditions **11-49**
Check the Final Prediction Error and Loss Function Values **11-49**
Perform Residual Analysis **11-50**
Examine Nonlinear ARX Plots **11-50**

Using Nonlinear ARX Models **11-55**

How the Software Computes Nonlinear ARX Model Output **11-57**
Evaluating Nonlinearities **11-57**
Simulation and Prediction of Sigmoid Network **11-57**

Linear Approximation of Nonlinear Black-Box Models . . . **11-64**
Why Compute a Linear Approximation of a Nonlinear Model? **11-64**
Choosing Your Linear Approximation Approach **11-64**
Linear Approximation of Nonlinear Black-Box Models for a Given Input **11-64**
Tangent Linearization of Nonlinear Black-Box Models **11-65**
Computing Operating Points for Nonlinear Black-Box Models **11-66**

12 Identify Hammerstein-Wiener Models

What are Hammerstein-Wiener Models? **12-2**
Structure of Hammerstein-Wiener Models **12-2**

Identifying Hammerstein-Wiener Models **12-5**
Prepare Data for Identification **12-5**
Configure Hammerstein-Wiener Model Structure **12-6**
Specify Estimation Options for Hammerstein-Wiener Models **12-7**
Initialize Hammerstein-Wiener Estimation Using Linear Model **12-8**

Using Hammerstein-Wiener Models **12-11**

Available Nonlinearity Estimators for Hammerstein-Wiener Models **12-13**

Estimate Hammerstein-Wiener Models in the App **12-15**

Estimate Hammerstein-Wiener Models at the Command Line **12-18**
Estimate Model Using nlhw **12-18**
Configure Nonlinearity Estimators **12-19**
Exclude Input or Output Nonlinearity **12-20**
Iteratively Refine Model **12-20**
Improve Estimation Results Using Initial States **12-21**
Troubleshoot Estimation **12-22**
Estimate Multiple Hammerstein-Wiener Models **12-22**
Improve a Linear Model Using Hammerstein-Wiener Structure **12-23**

Validating Hammerstein-Wiener Models **12-27**
Compare Simulated Model Output to Measured Output . . . **12-27**
Check Iterative Search Termination Conditions **12-27**
Check the Final Prediction Error and Loss Function Values **12-28**
Perform Residual Analysis **12-28**
Examine Hammerstein-Wiener Plots **12-28**

How the Software Computes Hammerstein-Wiener Model Output **12-33**
Evaluating Nonlinearities (SISO) **12-33**
Evaluating Nonlinearities (MIMO) **12-33**
Simulation of Hammerstein-Wiener Model **12-34**

Estimate Hammerstein-Wiener Models Initialized Using Linear OE Models **12-37**

13 ODE Parameter Estimation (Grey-Box Modeling)

Supported Grey-Box Models **13-2**

Data Supported by Grey-Box Models **13-4**

Choosing idgrey or idnlgrey Model Object **13-5**

Estimate Linear Grey-Box Models **13-8**
Specifying the Linear Grey-Box Model Structure **13-8**
Create Function to Represent a Grey-Box Model **13-9**

Estimate Continuous-Time Grey-Box Model for Heat Diffusion **13-13**

Estimate Discrete-Time Grey-Box Model with Parameterized Disturbance **13-17**
Description of the SISO System **13-17**
Estimating the Parameters of an idgrey Model **13-17**

Estimate Coefficients of ODEs to Fit Given Solution **13-20**

Estimate Model Using Zero/Pole/Gain Parameters **13-28**

Estimate Nonlinear Grey-Box Models **13-35**
Specifying the Nonlinear Grey-Box Model Structure **13-35**
Constructing the idnlgrey Object **13-37**
Using nlgreyest to Estimate Nonlinear Grey-Box Models . . **13-37**
Represent Nonlinear Dynamics Using MATLAB File for Grey-Box Estimation **13-38**

Nonlinear Grey-Box Model Properties and Estimation Options 13-56

Creating IDNLGREY Model Files 13-59

Identifying State-Space Models with Separate Process and Measurement Noise Descriptions 13-72
General Model Structure 13-72
Innovations Form and One-Step Ahead Predictor 13-73
Model Identification 13-74
Summary 13-76

After Estimating Grey-Box Models 13-77

14 Time Series Identification

What Are Time Series Models? 14-2

Preparing Time-Series Data 14-4

Estimate Time-Series Power Spectra 14-5
How to Estimate Time-Series Power Spectra Using the App 14-5
How to Estimate Time-Series Power Spectra at the Command Line 14-6

Estimate AR and ARMA Models 14-8
Definition of AR and ARMA Models 14-8
Estimating Polynomial Time-Series Models in the App 14-8
Estimating AR and ARMA Models at the Command Line . . 14-11

Estimate State-Space Time Series Models 14-13
Definition of State-Space Time Series Model 14-13
Estimating State-Space Models at the Command Line 14-13

Identify Time-Series Models at the Command Line 14-14

Estimate ARIMA Models 14-20

Spectrum Estimation Using Complex Data - Marple's Test Case **14-23**

Analyze Time-Series Models **14-33**

Introduction to Forecasting of Dynamic System Response **14-37**
Forecasting Time Series Using Linear Models **14-37**
Forecasting Response of Linear Models with Exogenous Inputs **14-46**
Forecasting Response of Nonlinear Models **14-47**

Forecast the Output of a Dynamic System **14-50**
Forecast Time Series Data Using an ARMA Model **14-51**

15 Recursive Model Identification

Data Segmentation **15-2**

Detect Abrupt System Changes Using Identification Techniques **15-3**

16 Online Estimation

What Is Online Estimation? **16-2**
Online Parameter Estimation **16-2**
Online State Estimation **16-3**

How Online Parameter Estimation Differs from Offline Estimation **16-6**

Preprocess Online Parameter Estimation Data in Simulink **16-8**

Validate Online Parameter Estimation Results in Simulink **16-9**

Validate Online Parameter Estimation at the Command Line **16-11**
Examine the Estimation Error **16-11**
Simulate the Estimated Model **16-11**
Examine Parameter Covariance **16-12**
Use Validation Commands from System Identification Toolbox **16-12**

Troubleshoot Online Parameter Estimation **16-14**
Model Structure **16-14**
Model Order **16-15**
Estimation Data **16-15**
Initial Guess for Parameter Values **16-16**
Estimation Settings **16-16**

Generate Online Parameter Estimation Code in Simulink **16-18**

Recursive Algorithms for Online Parameter Estimation . . **16-20**
General Form of Recursive Estimation **16-20**
Types of Recursive Estimation Algorithms **16-21**

Perform Online Parameter Estimation at the Command Line **16-26**
Online Estimation System Object **16-26**
Workflow for Online Parameter Estimation at the Command Line **16-27**

Generate Code for Online Parameter Estimation in MATLAB **16-31**
Supported Online Estimation Commands **16-31**
Generate Code for Online Estimation **16-32**
Rules and Limitations When Using System Objects in Generated MATLAB Code **16-33**

Extended and Unscented Kalman Filter Algorithms for Online State Estimation **16-35**
Extended Kalman Filter Algorithm **16-35**
Unscented Kalman Filter Algorithm **16-38**

Validate Online State Estimation at the Command Line . . 16-44
Examine Output Estimation Error 16-44
Examine State Estimation Error for Simulated Data 16-45

Validate Online State Estimation in Simulink 16-47
Examine Residuals 16-47
Examine State Estimation Error for Simulated Data 16-48
Compute Residuals and State Estimation Errors 16-48

Generate Code for Online State Estimation in MATLAB . . 16-51
Tunable and Nontunable Object Properties 16-53

Troubleshoot Online State Estimation 16-55

Line Fitting with Online Recursive Least Squares Estimation 16-57

Online ARX Parameter Estimation for Tracking Time-Varying System Dynamics 16-66

Online Recursive Least Squares Estimation 16-82

Online ARMAX Polynomial Model Estimation 16-93

State Estimation Using Time-Varying Kalman Filter 16-110

Nonlinear State Estimation Using Unscented Kalman Filter and Particle Filter 16-124

Fault Detection Using an Extended Kalman Filter 16-145

Estimate States of Nonlinear System with Multiple, Multirate Sensors 16-160

17 Model Analysis

Validating Models After Estimation 17-3
Ways to Validate Models 17-3
Data for Model Validation 17-4

Supported Model Plots **17-5**

Plot Models in the System Identification App **17-7**

Simulating and Predicting Model Output **17-9**
Why Simulate or Predict Model Output? **17-9**
What are Simulation and Prediction? **17-9**

Simulation and Prediction in the App **17-12**
How to Plot Simulated and Predicted Model Output **17-12**
Interpret the Model Output Plot **17-12**
Change Model Output Plot Settings **17-14**
Definition: Confidence Interval **17-16**

Simulation and Prediction at the Command Line **17-17**
Simulation and Prediction Commands **17-17**
Initial States in Simulation and Prediction **17-18**
Simulate a Continuous-Time State-Space Model **17-19**
Simulate Model Output with Noise **17-20**

Compare Simulated Output with Measured Data **17-22**

Perform Multivariate Time Series Forecasting **17-25**

What Is Residual Analysis? **17-43**
Supported Model Types **17-43**
What Residual Plots Show for Different Data Domains **17-44**
Displaying the Confidence Interval **17-45**

How to Plot Residuals in the App **17-47**

How to Plot Residuals at the Command Line **17-50**

Examine Model Residuals **17-51**
Creating Residual Plots **17-51**
Description of the Residual Plot Axes **17-52**
Validating Models Using Analyzing Residuals **17-53**

Impulse and Step Response Plots **17-55**
Supported Models **17-55**
How Transient Response Helps to Validate Models **17-55**
What Does a Transient Response Plot Show? **17-56**
Displaying the Confidence Interval **17-57**

Plot Impulse and Step Response Using the System Identification App **17-59**

Plot Impulse and Step Response at the Command Line . . . **17-62**

Frequency Response Plots **17-64**
What Is Frequency Response? **17-64**
How Frequency Response Helps to Validate Models **17-65**
What Does a Frequency-Response Plot Show? **17-65**
Displaying the Confidence Interval **17-66**

Plot Bode Plots Using the System Identification App **17-68**

Plot Bode and Nyquist Plots at the Command Line **17-71**

Noise Spectrum Plots **17-73**
Supported Models **17-73**
What Does a Noise Spectrum Plot Show? **17-73**
Displaying the Confidence Interval **17-74**

Plot the Noise Spectrum Using the System Identification App **17-76**

Plot the Noise Spectrum at the Command Line **17-79**

Pole and Zero Plots **17-81**
Supported Models **17-81**
What Does a Pole-Zero Plot Show? **17-81**
Displaying the Confidence Interval **17-83**

Reducing Model Order Using Pole-Zero Plots **17-84**

Model Poles and Zeros Using the System Identification App **17-85**

Plot Poles and Zeros at the Command Line **17-87**

Analyzing MIMO Models **17-88**
Overview of Analyzing MIMO Models **17-88**
Array Selector **17-89**
I/O Grouping for MIMO Models **17-91**
Selecting I/O Pairs **17-92**

Customize Response Plots Using the Response Plots Property Editor **17-94**
Opening the Property Editor **17-94**
Overview of Response Plots Property Editor **17-95**
Labels Pane **17-97**
Limits Pane **17-98**
Units Pane **17-98**
Style Pane **17-108**
Options Pane **17-110**
Editing Subplots Using the Property Editor **17-114**

Computing Model Uncertainty **17-116**
Why Analyze Model Uncertainty? **17-116**
What Is Model Covariance? **17-116**
Types of Model Uncertainty Information **17-117**
Definition of Confidence Interval for Specific Model Plots **17-118**

Troubleshooting Model Estimation **17-119**
About Troubleshooting Models **17-119**
Model Order Is Too High or Too Low **17-119**
Substantial Noise in the System **17-120**
Unstable Models **17-120**
Missing Input Variables **17-121**
System Nonlinearities **17-122**
Nonlinearity Estimator Produces a Poor Fit **17-122**

Next Steps After Getting an Accurate Model **17-124**

18 Setting Toolbox Preferences

Toolbox Preferences Editor **18-2**
Overview of the Toolbox Preferences Editor **18-2**
Opening the Toolbox Preferences Editor **18-2**
Units Pane **18-3**
Style Pane **18-5**
Options Pane **18-7**
Control System Designer Pane **18-7**

19 Control Design Applications

Using Identified Models for Control Design Applications . . **19-2**
How Control System Toolbox Software Works with Identified Models **19-2**
Using balred to Reduce Model Order **19-2**
Compensator Design Using Control System Toolbox Software **19-3**
Converting Models to LTI Objects **19-3**
Viewing Model Response Using the Linear System Analyzer **19-4**
Combining Model Objects **19-5**

Create and Plot Identified Models Using Control System Toolbox Software **19-7**

20 System Identification Toolbox Blocks

Using System Identification Toolbox Blocks in Simulink Models **20-2**

Preparing Data **20-3**

Identifying Linear Models **20-4**

Simulating Identified Model Output in Simulink **20-5**
When to Use Simulation Blocks **20-5**
Summary of Simulation Blocks **20-5**
Specifying Initial Conditions for Simulation **20-6**
Simulate Identified Model Using Simulink Software **20-7**

21 System Identification App

Steps for Using the System Identification App **21-2**

Working with System Identification App **21-3**
Starting and Managing Sessions **21-3**
Managing Models **21-7**
Working with Plots **21-11**
Customizing the System Identification App **21-15**

22 Diagnostics and Prognostics

Time Series Prediction and Forecasting for Prognosis **22-2**

Fault Detection Using Data Based Models **22-16**

Choosing Your System Identification Approach

- "Acknowledgments" on page 1-2
- "What Are Model Objects?" on page 1-3
- "Types of Model Objects" on page 1-5
- "Dynamic System Models" on page 1-8
- "Numeric Models" on page 1-10
- "About Identified Linear Models" on page 1-13
- "Linear Model Structures" on page 1-20
- "Available Linear Models" on page 1-25
- "Estimation Report" on page 1-29
- "Imposing Constraints on Model Parameter Values" on page 1-34
- "Recommended Model Estimation Sequence" on page 1-36
- "Supported Models for Time- and Frequency-Domain Data" on page 1-38
- "Supported Continuous- and Discrete-Time Models" on page 1-41
- "Model Estimation Commands" on page 1-44
- "Modeling Multiple-Output Systems" on page 1-45
- "Regularized Estimates of Model Parameters" on page 1-48
- "Estimate Regularized ARX Model Using System Identification App" on page 1-57
- "Loss Function and Model Quality Metrics" on page 1-64
- "Regularized Identification of Dynamic Systems" on page 1-77

Acknowledgments

System Identification Toolbox software is developed in association with the following leading researchers in the system identification field:

Lennart Ljung. Professor Lennart Ljung is with the Department of Electrical Engineering at Linköping University in Sweden. He is a recognized leader in system identification and has published numerous papers and books in this area.

Qinghua Zhang. Dr. Qinghua Zhang is a researcher at Institut National de Recherche en Informatique et en Automatique (INRIA) and at Institut de Recherche en Informatique et Systèmes Aléatoires (IRISA), both in Rennes, France. He conducts research in the areas of nonlinear system identification, fault diagnosis, and signal processing with applications in the fields of energy, automotive, and biomedical systems.

Peter Lindskog. Dr. Peter Lindskog is employed by NIRA Dynamics AB, Sweden. He conducts research in the areas of system identification, signal processing, and automatic control with a focus on vehicle industry applications.

Anatoli Juditsky. Professor Anatoli Juditsky is with the Laboratoire Jean Kuntzmann at the Université Joseph Fourier, Grenoble, France. He conducts research in the areas of nonparametric statistics, system identification, and stochastic optimization.

What Are Model Objects?

In this section...
“Model Objects Represent Linear Systems” on page 1-3
“About Model Data” on page 1-3

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.

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" (MATLAB).

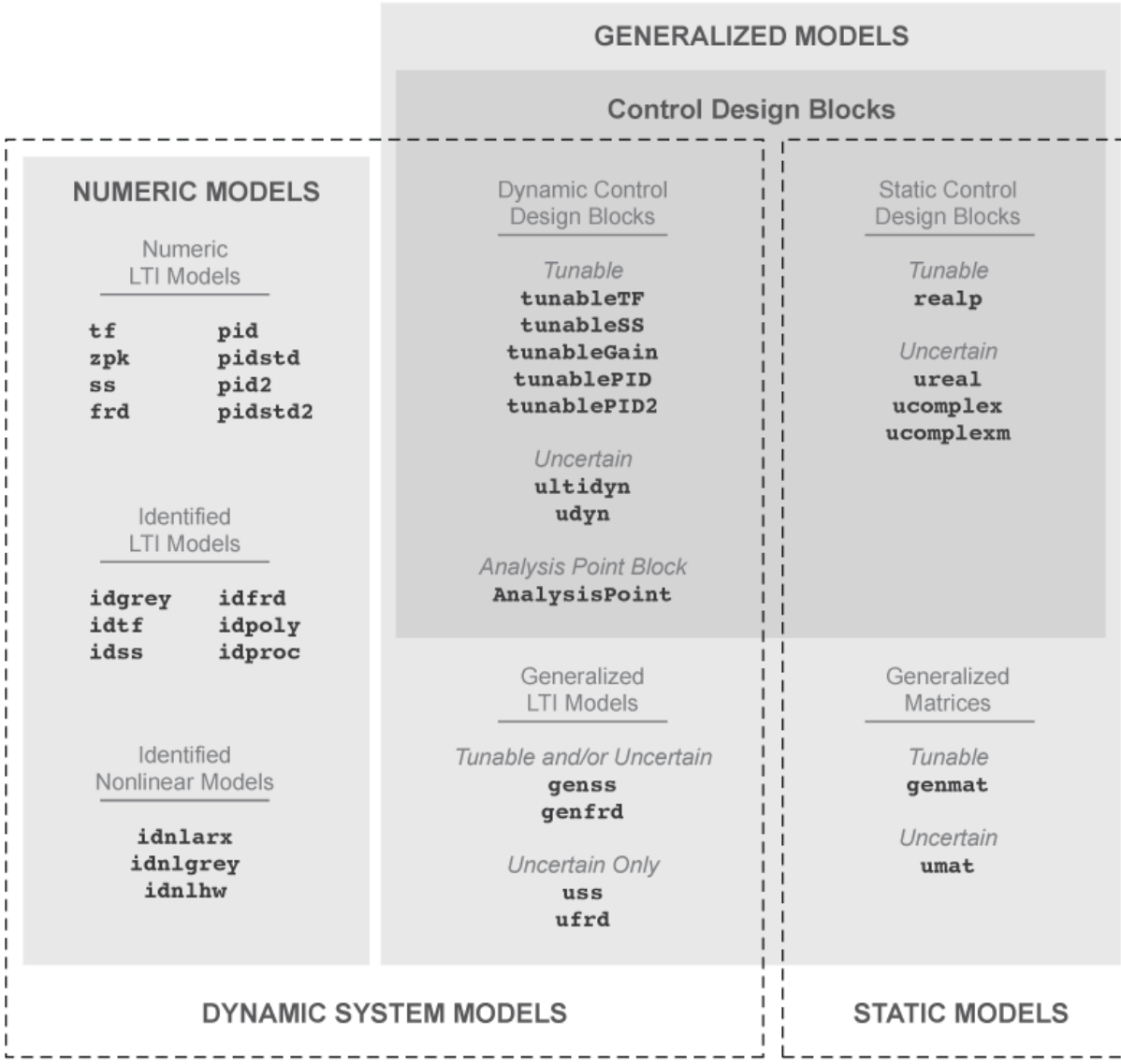
See Also

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 1-3
- "Dynamic System Models" on page 1-8
- "Numeric Models" on page 1-10

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`
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`

Model Family	Model Types
	`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`
	`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 1-11
- “Identified Nonlinear Models” on page 1-11

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” (Control System Toolbox)
- “State-Space Models” (Control System Toolbox)
- “Frequency Response Data (FRD) Models” (Control System Toolbox)
- “Proportional-Integral-Derivative (PID) Controllers” (Control System Toolbox)

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 (Control System Toolbox).
- 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. 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. 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

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

About Identified Linear Models

In this section...
"What are IDLTI Models?" on page 1-13
"Measured and Noise Component Parameterizations" on page 1-14
"Linear Model Estimation" on page 1-17

What are IDLTI Models?

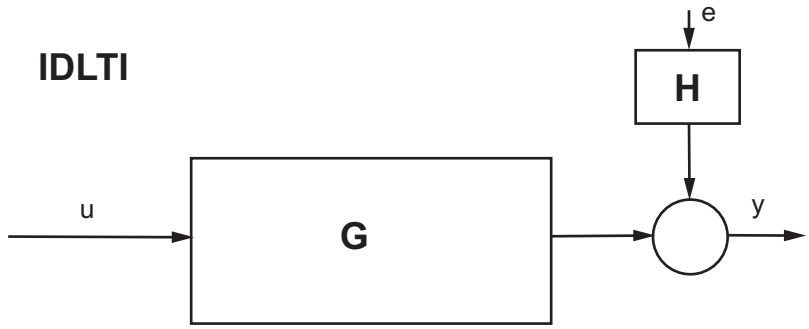
System Identification Toolbox software uses objects to represent a variety of linear and nonlinear model structures. These linear model objects are collectively known as Identified Linear Time-Invariant (IDLTI) models.

IDLTI models contain two distinct dynamic components:

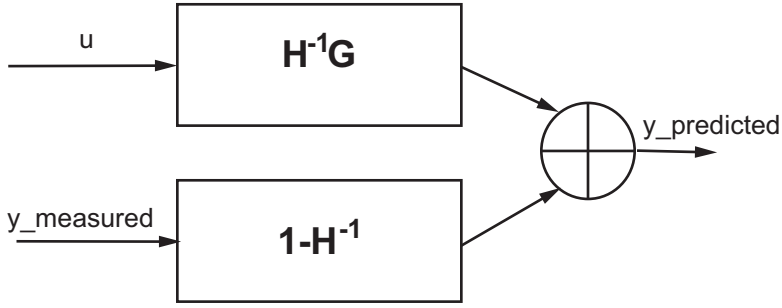
- **Measured component** — Describes the relationship between the measured inputs and the measured output (G)
- **Noise component** — Describes the relationship between the disturbances at the output and the measured output (H)

Models that only have the noise component `H` are called time-series or signal models. Typically, you create such models using time-series data that consist of one or more outputs `y(t)` with no corresponding input.

The total output is the sum of the contributions from the measured inputs and the disturbances: $y = G\,u + H\,e$, where u represents the measured inputs and e the disturbance. *e(t)* is modeled as zero-mean Gaussian white noise with variance Λ. The following figure illustrates an IDLTI model.



When you simulate an IDLTI model, you study the effect of input *u(t)* (and possibly initial conditions) on the output *y(t)*. The noise *e(t)* is not considered. However, with finite-horizon prediction of the output, both the measured and the noise components of the model contribute towards computation of the (predicted) response.



One-step ahead prediction model corresponding to a linear identified model (y = Gu+He)

Measured and Noise Component Parameterizations

The various linear model structures provide different ways of parameterizing the transfer functions `G` and `H`. When you construct an IDLTI model or estimate a model directly using input-output data, you can configure the structure of both *G* and *H*, as described in the following table:

Model Type	Transfer Functions G and H	Configuration Method
State space model (`idss`)	Represents an identified state-space model structure, governed by the equations: $\dot{x} = Ax + Bu + Ke$ $y = Cx + Du + e$ where the transfer function between the measured input *u* and output *y* is $G(s) = C(sI - A)^{-1}B + D$ and the noise transfer function is $H(s) = C(sI - A)^{-1}K + I$.	Construction: Use `idss` to create a model, specifying values of state-space matrices *A*, *B*, *C*, *D* and *K* as input arguments (using `NaNs` to denote unknown entries). Estimation: Use `ssest` or `n4sid`, specifying name-value pairs for various configurations, such as, canonical parameterization of the measured dynamics (`'Form'/'canonical'`), denoting absence of feedthrough by fixing *D* to zero (`'Feedthrough'/false`), and absence of noise dynamics by fixing *K* to zero (`'DisturbanceModel'/'none'`).

Model Type	Transfer Functions G and H	Configuration Method
Polynomial model (`idpoly`)	Represents a polynomial model such as ARX, ARMAX and BJ. An ARMAX model, for example, uses the input-output equation Ay(t) = Bu(t)+Ce(t), so that the measured transfer function *G* is $G(s) = A^{-1}B$, while the noise transfer function is $H(s) = A^{-1}C$. The ARMAX model is a special configuration of the general polynomial model whose governing equation is: $Ay(t) = \frac{B}{F}u(t) + \frac{C}{D}e(t)$ The autoregressive component, *A*, is common between the measured and noise components. The polynomials *B* and *F* constitute the measured component while the polynomials *C* and *D* constitute the noise component.	Construction: Use `idpoly` to create a model using values of active polynomials as input arguments. For example, to create an Output-Error model which uses `G = B/F` as the measured component and has a trivial noise component (`H = 1`). enter: `y = idpoly([],B,[],[],F)` Estimation: Use the `armax`, `arx`, or `bj`, specifying the orders of the polynomials as input arguments. For example, `bj` requires you to specify the orders of the *B*, *C*, *D*, and *F* polynomials to construct a model with governing equation $y(t) = \frac{B}{F}u(t) + \frac{C}{D}e(t)$
Transfer function model (`idtf`)	Represents an identified transfer function model, which has no dynamic elements to model noise behavior. This object uses the trivial noise model *H(s) = I*. The governing equation is $y(t) = \frac{num}{den}u(t) + e(t)$	Construction: Use `idtf` to create a model, specifying values of the numerator and denominator coefficients as input arguments. The numerator and denominator vectors constitute the measured component `G = num(s)/den(s)`. The noise component is fixed to `H = 1`. Estimation: Use `tfest`, specifying the number of poles and zeros of the measured component `G`.

Model Type	Transfer Functions G and H	Configuration Method

Process model (`idproc`)	Represents a process model, which provides options to represent the noise dynamics as either first- or second-order ARMA process (that is, *H(s)= C(s)/A(s)*, where *C(s)* and *A(s)* are monic polynomials of equal degree). The measured component, *G(s)*, is represented by a transfer function expressed in pole-zero form.	For process (and grey-box) models, the noise component is often treated as an on-demand extension to an otherwise measured component-centric representation. For these models, you can add a noise component by using the `DisturbanceModel` estimation option. For example:
		`model = procest(data,'P1D')`
		estimates a model whose equation is:
		$y(s) = K_p \frac{1}{(T_{p1}s+1)} e^{-sTd} u(s) + e(s).$
		To add a second order noise component to the model, use:
		`Options = procestOptions('DisturbanceModel','AR`
		`model = procest(data,'P1D',Options);`
		This model has the equation:
		$y(s) = K_p \frac{1}{(T_{p1}s+1)} e^{-sTd} u(s) + \frac{1+c_1 s}{1+d_1 s} e(s)$
		where the coefficients *c1* and *d1* parameterize the noise component of the model. If you are constructing a process model using the `idproc` command, specify the structure of the measured component using the `Type` input argument and the noise component by using the `NoiseTF` name-value pair. For example,
		`model = idproc('P1','Kp',1,'Tp1',1,'NoiseTF',..`
		`struct('num',[1 0.1],'den',[1 0.5]))`

Model Type	Transfer Functions G and H	Configuration Method
		creates the process model *y(s) = 1/(s+1) u(s) + (s + 0.1)/(s + 0.5) e(s)*

Sometimes, fixing coefficients or specifying bounds on the parameters are not sufficient. For example, you may have unrelated parameter dependencies in the model or parameters may be a function of a different set of parameters that you want to identify exclusively. For example, in a mass-spring-damper system, the `A` and `B` parameters both depend on the mass of the system. To achieve such parameterization of linear models, you can use grey-box modeling where you establish the link between the actual parameters and model coefficients by writing an ODE file. To learn more, see "Grey-Box Model Estimation".

Linear Model Estimation

You typically use estimation to create models in System Identification Toolbox. You execute one of the estimation commands, specifying as input arguments the measured data, along with other inputs necessary to define the structure of a model. To illustrate, the following example uses the state-space estimation command, `ssest`, to create a state space model. The first input argument `data` specifies the measured input-output data. The second input argument specifies the order of the model.

```
sys = ssest(data,4)
```

The estimation function treats the noise variable *e(t)* as prediction error – the residual portion of the output that cannot be attributed to the measured inputs. All estimation algorithms work to minimize a weighted norm of *e(t)* over the span of available measurements. The weighting function is defined by the nature of the noise transfer function *H* and the focus of estimation, such as simulation or prediction error minimization.

- "Black Box ("Cold Start") Estimation" on page 1-17
- "Structured Estimations" on page 1-18
- "Estimation Options" on page 1-19

Black Box ("Cold Start") Estimation

In a black-box estimation, you only have to specify the order to configure the structure of the model.

```
sys = estimator(data,orders)
```

where *estimator* is the name of an estimation command to use for the desired model type.

For example, you use `tfest` to estimate transfer function models, `arx` for ARX-structure polynomial models, and `procest` for process models.

The first argument, `data`, is time- or frequency domain data represented as an `iddata` or `idfrd` object. The second argument, `orders`, represents one or more numbers whose definitions depends upon the model type:

- For transfer functions, `orders` refers to the number of poles and zeros.
- For state-space models, `orders` refers to the number of states.
- For process models, `orders` denotes the structural elements of a process model, such as, the number of poles and presence of delay and integrator.

When working with the app, you specify the orders in the appropriate edit fields of corresponding model estimation dialogs.

Structured Estimations

In some situations, you want to configure the structure of the desired model more closely than what is achieved by simply specifying the orders. In such cases, you construct a template model and configure its properties. You then pass that template model as an input argument to the estimation commands in place of `orders`.

To illustrate, the following example assigns initial guess values to the numerator and the denominator polynomials of a transfer function model, imposes minimum and maximum bounds on their estimated values, and then passes the object to the estimator function.

```
% Initial guess for numerator
num = [1 2];
den = [1 2 1 1];
% Initial guess for the denominator
sys = idtf(num,den);
% Set min bound on den coefficients to 0.1
sys.Structure.Denominator.Minimum = [1 0.1 0.1 0.1];
sysEstimated = tfest(data,sys);
```

The estimation algorithm uses the provided initial guesses to kick-start the estimation and delivers a model that respects the specified bounds.

You can use such a model template to also configure auxiliary model properties such as input/output names and units. If the values of some of the model's parameters are initially unknown, you can use `NaNs` for them in the template.

Estimation Options

There are many options associated with a model's estimation algorithm that configure the estimation objective function, initial conditions and numerical search algorithm, among other things. For every estimation command, *`estimator`*, there is a corresponding option command named *`estimator`*`Options`. To specify options for a particular estimator command, such as `tfest`, use the options command that corresponds to the estimation command, in this case, `tfestOptions`. The options command returns an options set that you then pass as an input argument to the corresponding estimation command.

For example, to estimate an Output-Error structure polynomial model, you use `oe`. To specify `simulation` as the focus and `lsqnonlin` as the search method, you use `oeOptions`:

```
load iddata1 z1
Options = oeOptions('Focus','simulation','SearchMethod','lsqnonlin');
sys= oe(z1,[2 2 1],Options);
```

Information about the options used to create an estimated model is stored in the `OptionsUsed` field of the model's `Report` property. For more information, see "Estimation Report" on page 1-29.

See Also

More About

- "Types of Model Objects" on page 1-5
- "Available Linear Models" on page 1-25
- "About Identified Nonlinear Models" on page 11-2

Linear Model Structures

In this section...
“About System Identification Toolbox Model Objects” on page 1-20
“When to Construct a Model Structure Independently of Estimation” on page 1-21
“Commands for Constructing Linear Model Structures” on page 1-21
“Model Properties” on page 1-22
“See Also” on page 1-24

About System Identification Toolbox Model Objects

Objects are instances of model classes. Each *class* is a blueprint that defines the following information about your model:

- How the object stores data
- Which operations you can perform on the object

This toolbox includes nine classes for representing models. For example, `idss` represents linear state-space models and `idnlarx` represents nonlinear ARX models. For a complete list of available model objects, see “Available Linear Models” on page 1-25 and “Available Nonlinear Models” on page 11-12.

Model *properties* define how a model object stores information. Model objects store information about a model, such as the mathematical form of a model, names of input and output channels, units, names and values of estimated parameters, parameter uncertainties, and estimation report. For example, an `idss` model has an `InputName` property for storing one or more input channel names.

The allowed operations on an object are called *methods*. In System Identification Toolbox software, some methods have the same name but apply to multiple model objects. For example, `step` creates a step response plot for all dynamic system objects. However, other methods are unique to a specific model object. For example, `canon` is unique to state-space `idss` models and `linearize` to nonlinear black-box models.

Every class has a special method, called the *constructor*, for creating objects of that class. Using a constructor creates an instance of the corresponding class or *instantiates the object*. The constructor name is the same as the class name. For example, `idss` and

`idnlarx` are both the name of the class and the name of the constructor for instantiating the linear state-space models and nonlinear ARX models, respectively.

When to Construct a Model Structure Independently of Estimation

You use model constructors to create a model object at the command line by specifying all required model properties explicitly.

You must construct the model object independently of estimation when you want to:

- Simulate or analyze the effect of model parameters on its response, independent of estimation.
- Specify an initial guess for specific model parameter values before estimation. You can specify bounds on parameter values, or set up the auxiliary model information in advance, or both. Auxiliary model information includes specifying input/output names, units, notes, user data, and so on.

In most cases, you can use the estimation commands to both construct and estimate the model—without having to construct the model object independently. For example, the estimation command `tfest` creates a transfer function model using data and the number of poles and zeros of the model. Similarly, `nlarx` creates a nonlinear ARX model using data and model orders and delays that define the regressor configuration. For information about how to both construct and estimate models with a single command, see “Model Estimation Commands” on page 1-44.

In case of grey-box models, you must always construct the model object first and then estimate the parameters of the ordinary differential or difference equation.

Commands for Constructing Linear Model Structures

The following table summarizes the model constructors available in the System Identification Toolbox product for representing various types of linear models.

After model estimation, you can recognize the corresponding model objects in the MATLAB Workspace browser by their class names. The name of the constructor matches the name of the object it creates.

For information about how to both construct and estimate models with a single command, see “Model Estimation Commands” on page 1-44.

Summary of Model Constructors

Model Constructor	Resulting Model Class
`idfrd`	Nonparametric frequency-response model.
`idproc`	Continuous-time, low-order transfer functions (process models).
`idpoly`	Linear input-output polynomial models: • ARX • ARMAX • Output-Error • Box-Jenkins
`idss`	Linear state-space models.
`idtf`	Linear transfer function models.
`idgrey`	Linear ordinary differential or difference equations (grey-box models). You write a function that translates user parameters to state-space matrices. Can also be viewed as state-space models with user-specified parameterization.

For more information about when to use these commands, see “When to Construct a Model Structure Independently of Estimation” on page 1-21.

Model Properties

- “Categories of Model Properties” on page 1-22
- “Viewing Model Properties and Estimated Parameters” on page 1-23

Categories of Model Properties

The way a model object stores information is defined by the *properties* of the corresponding model class.

Each model object has properties for storing information that are relevant only to that specific model type. The `idtf`, `idgrey`, `idpoly`, `idproc`, and `idss` model objects are based on the `idlti` superclass and inherit all `idlti` properties.

In general, all model objects have properties that belong to the following categories:

- Names of input and output channels, such as `InputName` and `OutputName`
- Sample time of the model, such as `Ts`
- Units for time or frequency
- Model order and mathematical structure (for example, ODE or nonlinearities)
- Properties that store estimation results (`Report`)
- User comments, such as `Notes` and `Userdata`

For information about getting help on object properties, see the model reference pages.

Viewing Model Properties and Estimated Parameters

The following table summarizes the commands for viewing and changing model property values. Property names are not case sensitive. You do not need to type the entire property name if the first few letters uniquely identify the property.

Task	Command	Example

View all model properties and their values	`get`	Load sample data, compute an ARX model, and list the model properties:
		`load iddata8`
		`m_arx=arx(z8,[4 3 2 3 0 0 0]);`
		`get(m_arx)`

Access a specific model property	Use dot notation	View the *A* matrix containing the estimated parameters in the previous model: `m_arx.A`
	For properties, such as `Report`, that are configured like structures, use dot notation of the form `model.PropertyName.FieldName`. `FieldName` is the name of any field of the property.	View the method used in ARX model estimation: `m_arx.Report.Method`
Change model property values	dot notation	Change the input delays for all three input channels to `[1 1 1]` for an ARX model: `m_arx.InputDelay = [1 1 1]`

Task	Command	Example

Access model parameter values and uncertainty information	Use `getpar`, `getpvec` and `getcov`
	See Also: `polydata`, `idssdata`, `tfdata`, `zpkdata`
		• View a table of all parameter attributes:
		`getpar(m_arx)`
		• View the *A* polynomial and 1 standard uncertainty of an ARX model:
		`[a,~,~,~,~,da] = polydata(m_arx)`

Set model property values and uncertainty information	Use `setpar`, `setpvec` and `setcov`	• Set default parameter labels:
		`m_arx = setpar(m_arx,'label','default')`
		• Set parameter covariance data:
		`m_arx = setcov(m_arx,cov)`

Get number of parameters	Use `nparams`	Get the number of parameters: `nparams(sys)`

See Also

Validate each model directly after estimation to help fine-tune your modeling strategy. When you do not achieve a satisfactory model, you can try a different model structure and order, or try another identification algorithm. For more information about validating and troubleshooting models, see “Validating Models After Estimation” on page 17-3.

Available Linear Models

A linear model is often sufficient to accurately describe the system dynamics and, in most cases, you should first try to fit linear models. Available linear structures include transfer functions and state-space models, summarized in the following table.

Model Type	Usage	Learn More
Transfer function (`idtf`)	Use this structure to represent transfer functions: $y = \frac{num}{den}u + e$ where *num* and *den* are polynomials of arbitrary lengths. You can specify initial guesses for, and estimate, *num*, *den*, and transport delays.	"Transfer Function Models"
Process model (`idproc`)	Use this structure to represent process models that are low order transfer functions expressed in pole-zero form. They include integrator, delay, zero, and up to 3 poles.	"Process Models"

Model Type	Usage	Learn More
State-space model (`idss`)	Use this structure to represent known state-space structures and black-box structures. You can fix certain parameters to known values and estimate the remaining parameters. You can also prescribe minimum/maximum bounds on the values of the free parameters. If you need to specify parameter dependencies or parameterize the state-space matrices using your own parameters, use a grey-box model.	“State-Space Models”

Model Type	Usage	Learn More
Polynomial models (`idpoly`)	Use to represent linear transfer functions based on the general form input-output polynomial form: $Ay = \frac{B}{F}u + \frac{C}{D}e$ where A, B, C, D and F are polynomials with coefficients that the toolbox estimates from data. Typically, you begin modeling using simpler forms of this generalized structure (such as ARX: $Ay = Bu + e$ and OE: $y = \frac{B}{F}u + e$) and, if necessary, increase the model complexity.	"Input-Output Polynomial Models"
Grey-box model (`idgrey`)	Use to represent arbitrary parameterizations of state-space models. For example, you can use this structure to represent your ordinary differential or difference equation (ODE) and to define parameter dependencies.	"Linear Grey-Box Models"

See Also

More About

- "Linear Model Structures" on page 1-20
- "About Identified Linear Models" on page 1-13

Estimation Report

In this section...
“What is an Estimation Report?” on page 1-29
“Access Estimation Report” on page 1-30
“Compare Estimated Models Using Estimation Report” on page 1-31
“Analyze and Refine Estimation Results Using Estimation Report” on page 1-32

What is an Estimation Report?

The estimation report contains information about the results and options used for a model estimation. This report is stored in the `Report` property of the estimated model. The exact contents of the report depend on the estimator function you use to obtain the model.

Specifically, the estimation report has the following information:

- Status of the model — whether the model is constructed or estimated
- How the initial conditions are handled during estimation
- Termination conditions for iterative estimation algorithms
- Final prediction error (FPE), percent fit to estimation data, and mean-square error (MSE)
- Raw, normalized, and small sample-size corrected Akaike Information Criteria (AIC) and Bayesian Information Criterion (BIC)
- Type and properties of the estimation data
- All estimated quantities — parameter values, initial states for state-space and grey-box models, and their covariances
- The option set used for configuring the estimation algorithm

To learn more about the report produced for a specific estimator, see the corresponding reference page.

You can use the report to:

- Keep an estimation log, such as the data, default and other settings used, and estimated results such as parameter values, initial conditions, and fit. See “Access Estimation Report” on page 1-30.

- Compare options or results of separate estimations. See “Compare Estimated Models Using Estimation Report” on page 1-31.
- Configure another estimation using the previously specified options. See “Analyze and Refine Estimation Results Using Estimation Report” on page 1-32.

Access Estimation Report

This example shows how to access the estimation report.

The estimation report keeps a log of information such as the data used, default and other settings used, and estimated results such as parameter values, initial conditions, and fit.

After you estimate a model, use dot notation to access the estimation report. For example:

```
load iddata1 z1;
np = 2;
sys = tfest(z1,np);
sys_report = sys.Report
```

```
sys_report =

              Status: 'Estimated using TFEST'
              Method: 'TFEST'
          InitMethod: 'IV'
            N4Weight: 'Not applicable'
           N4Horizon: 'Not applicable'
    InitialCondition: 'estimate'
                 Fit: [1x1 struct]
          Parameters: [1x1 struct]
        OptionsUsed: [1x1 idoptions.tfest]
           RandState: []
            DataUsed: [1x1 struct]
        Termination: [1x1 struct]
```

Explore the options used during the estimation.

```
sys.Report.OptionsUsed
```

```
Option set for the tfest command:
```

```
          InitMethod: 'iv'
          InitOption: [1x1 struct]
    InitialCondition: 'auto'
             Display: 'off'
         InputOffset: []
        OutputOffset: []
             EstCovar: 1
      Regularization: [1x1 struct]
        SearchMethod: 'auto'
        SearchOption: [1x1 idoptions.search.identsolver]
     WeightingFilter: []
    EnforceStability: 0
        OutputWeight: []
            Advanced: [1x1 struct]
```

View the fit of the transfer function model with the estimation data.

```
sys.Report.Fit
```

```
ans =

  struct with fields:

    FitPercent: 70.7720
       LossFcn: 1.6575
           MSE: 1.6575
           FPE: 1.7252
           AIC: 1.0150e+03
          AICc: 1.0153e+03
          nAIC: 0.5453
           BIC: 1.0372e+03
```

Compare Estimated Models Using Estimation Report

This example shows how to compare multiple estimated models using the estimation report.

Load estimation data.

```
load iddata1 z1;
```

Estimate a transfer function model.

```
np = 2;
sys_tf = tfest(z1,np);
```

Estimate a state-space model.

```
sys_ss = ssest(z1,2);
```

Estimate an ARX model.

```
sys_arx = arx(z1, [2 2 1]);
```

Compare the percentage fit of the estimated models to the estimation data.

```
fit_tf = sys_tf.Report.Fit.FitPercent
```

```
fit_tf = 70.7720
```

```
fit_ss = sys_ss.Report.Fit.FitPercent
```

```
fit_ss = 76.3808
```

```
fit_arx = sys_arx.Report.Fit.FitPercent
```

```
fit_arx = 68.7220
```

The comparison shows that the state-space model provides the best percent fit to the data.

Analyze and Refine Estimation Results Using Estimation Report

This example shows how to analyze an estimation and configure another estimation using the estimation report.

Estimate a state-space model that minimizes the 1-step ahead prediction error.

```
load(fullfile(matlabroot,'toolbox','ident','iddemos','data','mrdamper.mat'));
z = iddata(F,V,Ts);
opt = ssestOptions;
opt.Focus = 'prediction';
opt.Display = 'on';
sys1 = ssest(z,2,opt);
```

`sys1` has good 1-step prediction ability as indicated by >90% fit of the prediction results to the data.

Use `compare(z,sys1)` to check the model's ability to simulate the measured output `F` using the input `V`. The model's simulated response has only 45% fit to the data.

Perform another estimation where you retain the original options used for `sys1` except that you change the focus to minimize the simulation error.

Fetch the options used by `sys1` stored in its `Report` property. This approach is useful when you have saved the estimated model but not the corresponding option set used for the estimation.

```
opt2 = sys1.Report.OptionsUsed;
```

Change the focus to simulation and re-estimate the model.

```
opt2.Focus = 'simulation';
sys2 = ssest(z,sys1,opt2);
```

Compare the simulated response to the estimation data using `compare(z,sys1,sys2)`. The fit improves to 53%.

See Also

More About

- “About Identified Linear Models” on page 1-13
- “About Identified Nonlinear Models” on page 11-2

Imposing Constraints on Model Parameter Values

All identified linear (IDLTI) models, except `idfrd`, contain a `Structure` property. The `Structure` property contains the adjustable entities (parameters) of the model. Each parameter has attributes such as value, minimum/maximum bounds, and free/fixed status that allow you to constrain them to desired values or a range of values during estimation. You use the `Structure` property to impose constraints on the values of various model parameters.

The `Structure` property contains the essential parameters that define the structure of a given model:

- For identified transfer functions, includes the numerator, denominator, and delay parameters
- For polynomial models, includes the list of active polynomials
- For state-space models, includes the list of state-space matrices

For information about other model types, see the model reference pages.

For example, the following example constructs an `idtf` model, specifying values for the `Numerator` and `Denominator` parameters:

```
num = [1 2];
den = [1 2 2];
sys = idtf(num,den)
```

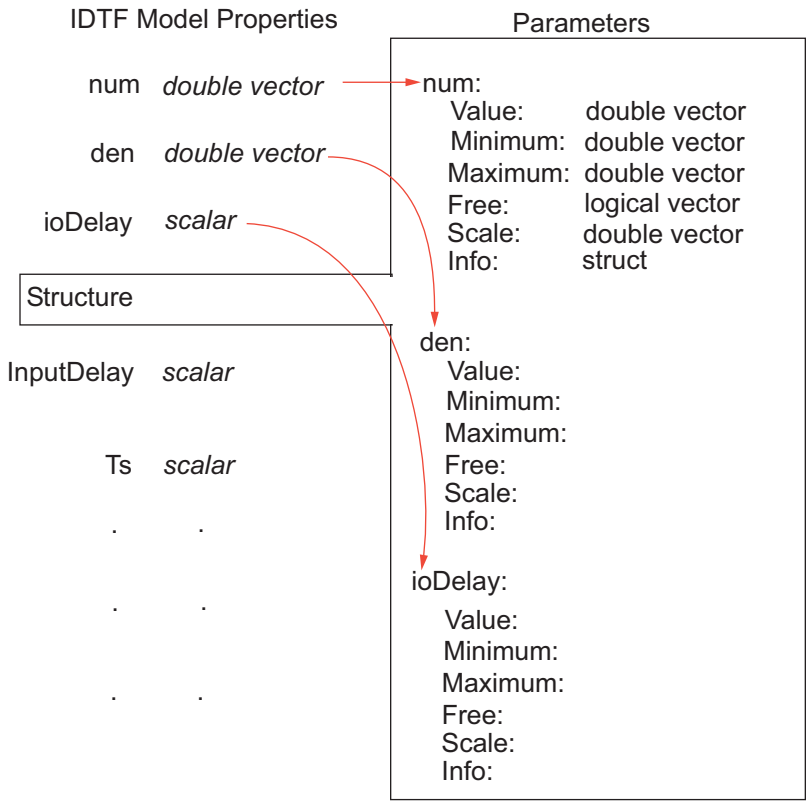
You can update the value of the `Numerator` and `Denominator` properties after you create the object as follows:

```
new_den = [1 1 10];
sys.Denominator = new_den;
```

To fix the denominator to the value you specified (treat its coefficients as fixed parameters), use the `Structure` property of the object as follows:

```
sys.Structure.Denominator.Value = new_den;
sys.Structure.Denominator.Free = false(1,3);
```

For a transfer function model, the `Numerator`, `Denominator`, and `IODelay` model properties are simply pointers to the `Value` attribute of the corresponding parameter in the `Structure` property.



Similar relationships exist for other model structures. For example, the `A` property of a state-space model contains the double value of the state matrix. It is an alias to the `A` parameter value stored in `Structure.A.Value`.

Recommended Model Estimation Sequence

System identification is an iterative process, where you identify models with different structures from data and compare model performance. You start by estimating the parameters of simple model structures. If the model performance is poor, you gradually increase the complexity of the model structure. Ultimately, you choose the simplest model that best describes the dynamics of your system.

Another reason to start with simple model structures is that higher-order models are not always more accurate. Increasing model complexity increases the uncertainties in parameter estimates and typically requires more data (which is common in the case of nonlinear models).

Note Model structure is not the only factor that determines model accuracy. If your model is poor, you might need to preprocess your data by removing outliers or filtering noise. For more information, see "Ways to Prepare Data for System Identification" on page 2-6.

Estimate impulse-response and frequency-response models first to gain insight into the system dynamics and assess whether a linear model is sufficient. For more information, see "Correlation Models" and "Frequency-Response Models". Then, estimate parametric models in the following order:

1. Transfer function, ARX polynomial, and state-space models provide the simplest structures. Estimation of ARX and state-space models let you determine the model orders.

   In the System Identification app. Choose to estimate the Transfer function models, ARX polynomial models, and the state-space model using the `n4sid` method.

   At the command line. Use the `tfest`, `arx`, and the `n4sid` commands, respectively.

   For more information, see "Input-Output Polynomial Models" and "State-Space Models".

2. ARMAX and BJ polynomial models provide more complex structures and require iterative estimation. Try several model orders and keep the model orders as low as possible.

In the System Identification app. Select to estimate the BJ and ARMAX polynomial models.

At the command line. Use the `bj` or `armax` commands.

For more information, see "Input-Output Polynomial Models".

3 Nonlinear ARX or Hammerstein-Wiener models provide nonlinear structures. For more information, see "Nonlinear Model Identification".

For general information about choosing you model strategy, see "System Identification Overview". For information about validating models, see "Validating Models After Estimation" on page 17-3.

Supported Models for Time- and Frequency-Domain Data

In this section...
“Supported Models for Time-Domain Data” on page 1-38
“Supported Models for Frequency-Domain Data” on page 1-39
“See Also” on page 1-40

Supported Models for Time-Domain Data

Continuous-Time Models

You can directly estimate the following types of continuous-time models:

- Transfer function models.
- Process models.
- State-space models.

You can also use `d2c` to convert an estimated discrete-time model into a continuous-time model.

Discrete-Time Models

You can estimate all linear on page 1-20 and nonlinear on page 11-7 models supported by the System Identification Toolbox product as discrete-time models, except process models, which are defined only in continuous-time..

ODEs (Grey-Box Models)

You can estimate both continuous-time and discrete-time models from time-domain data for linear and nonlinear differential and difference equations.

Nonlinear Models

You can estimate discrete-time Hammerstein-Wiener and nonlinear ARX models from time-domain data.

You can also estimate nonlinear grey-box models from time-domain data. See “Estimate Nonlinear Grey-Box Models” on page 13-35.

Supported Models for Frequency-Domain Data

There are two types of frequency-domain data:

- Frequency response data
- Frequency domain input/output signals which are Fourier Transforms of the corresponding time domain signals.

The data is considered continuous-time if its sample time (`Ts`) is `0`, and is considered discrete-time if the sample time is nonzero.

Continuous-Time Models

You can estimate the following types of continuous-time models directly:

- Transfer function models using continuous- or discrete-time data.
- Process models using continuous- or discrete-time data.
- Input-output polynomial models of output-error structure using continuous time data.
- State-space models using continuous- or discrete-time data.

  From continuous-time frequency-domain data, you can only estimate continuous-time models.

You can also use `d2c` to convert an estimated discrete-time model into a continuous-time model.

Discrete-Time Models

You can estimate all linear model types supported by the System Identification Toolbox product as discrete-time models, except process models, which are defined in continuous-time only. For estimation of discrete-time models, you must use discrete-time data.

The noise component of a model cannot be estimated using frequency domain data, except for ARX models. Thus, the K matrix of an identified state-space model, the noise component, is zero. An identified polynomial model has output-error (OE) or ARX structure; BJ/ARMAX or other polynomial structure with nontrivial values of C or D polynomials cannot be estimated.

ODEs (Grey-Box Models)

For linear grey-box models, you can estimate both continuous-time and discrete-time models from frequency-domain data. The noise component of the model, the K matrix, cannot be estimated using frequency domain data; it remains fixed to `0`.

Nonlinear grey-box models are supported only for time-domain data.

Nonlinear Black-Box Models

Nonlinear black box (nonlinear ARX and Hammerstein-Wiener models) cannot be estimated using frequency domain data.

See Also

"Supported Continuous- and Discrete-Time Models" on page 1-41

Supported Continuous- and Discrete-Time Models

For linear and nonlinear ODEs (grey-box models), you can specify any ordinary differential or difference equation to represent your continuous-time or discrete-time model in state-space form, respectively. In the linear case, both time-domain and frequency-domain data are supported. In the nonlinear case, only time-domain data is supported.

For black-box models, the following tables summarize supported continuous-time and discrete-time models.

Supported Continuous-Time Models

Model Type	Description
Transfer function models	Estimate continuous-time transfer function models directly using `tfest` from either time- and frequency-domain data. If you estimated a discrete-time transfer function model from time-domain data, then use `d2c` to transform it into a continuous-time model.
Low-order transfer functions (process models)	Estimate low-order process models for up to three free poles from either time- or frequency-domain data.
Linear input-output polynomial models	To get a linear, continuous-time model of arbitrary structure from time-domain data, you can estimate a discrete-time model, and then use `d2c` to transform it into a continuous-time model. You can estimate only polynomial models of Output Error structure using continuous-time frequency domain data.. Other structures that include noise models, such as Box-Jenkins (BJ) and ARMAX, are not supported for frequency-domain data.
State-space models	Estimate continuous-time state-space models directly using the estimation commands from either time- and frequency-domain data. If you estimated a discrete-time state-space model from time-domain data, then use `d2c` to transform it into a continuous-time model.
Linear ODEs (grey-box) models	If the MATLAB file returns continuous-time model matrices, then estimate the ordinary differential equation (ODE) coefficients using either time- or frequency-domain data.
Nonlinear ODEs (grey-box) models	If the MATLAB file returns continuous-time output and state derivative values, estimate arbitrary differential equations (ODEs) from time-domain data.

Supported Discrete-Time Models

Model Type	Description
Linear input-output polynomial models	Estimate arbitrary-order, linear parametric models from time- or frequency-domain data. To get a discrete-time model, your data sample time must be set to the (nonzero) value you used to sample in your experiment.
“Nonlinear Model Identification”	Estimate from time-domain data only.
Linear ODEs (grey-box) models	If the MATLAB file returns discrete-time model matrices, then estimate ordinary difference equation coefficients from time-domain or discrete-time frequency-domain data.
Nonlinear ODEs (grey-box) models	If the MATLAB file returns discrete-time output and state update values, estimate ordinary difference equations from time-domain data.

Model Estimation Commands

In most cases, a model can be created by using a model estimation command on a dataset. For example, `ssest(data,nx)` creates a continuous-time state-space model of order Nx using the input/output of frequency response data DATA.

Note For ODEs (grey-box models), you must first construct the model structure and then apply an estimation command (either `greyest` or `pem`) to the resulting model object.

The following table summarizes System Identification Toolbox estimation commands. For detailed information about using each command, see the corresponding reference page.

Commands for Constructing and Estimating Models

Model Type	Estimation Commands
Transfer function models	`tfest`
Process models	`procest`
Linear input-output polynomial models	`armax` (ARMAX only) `arx` (ARX only) `bj` (BJ only) `iv4` (ARX only) `oe` (OE only) `polyest` (for all models)
State-space models	`n4sid` `ssest`

Time-series models	`ar`
	`arx` (for multiple outputs)
	`ivar`
	`nlarx`(for nonlinear time-series models)

Nonlinear ARX models	`nlarx`
Hammerstein-Wiener models	`nlhw`

Modeling Multiple-Output Systems

In this section...
"About Modeling Multiple-Output Systems" on page 1-45
"Modeling Multiple Outputs Directly" on page 1-45
"Modeling Multiple Outputs as a Combination of Single-Output Models" on page 1-46
"Improving Multiple-Output Estimation Results by Weighing Outputs During Estimation" on page 1-46

About Modeling Multiple-Output Systems

You can estimate multiple-output model directly using all the measured inputs and outputs, or you can try building models for subsets of the most important input and output channels. To learn more about each approach, see:

- "Modeling Multiple Outputs Directly" on page 1-45
- "Modeling Multiple Outputs as a Combination of Single-Output Models" on page 1-46

Modeling multiple-output systems is more challenging because input/output couplings require additional parameters to obtain a good fit and involve more complex models. In general, a model is better when more data inputs are included during modeling. Including more outputs typically leads to worse simulation results because it is harder to reproduce the behavior of several outputs simultaneously.

If you know that some of the outputs have poor accuracy and should be less important during estimation, you can control how much each output is weighed in the estimation. For more information, see "Improving Multiple-Output Estimation Results by Weighing Outputs During Estimation" on page 1-46.

Modeling Multiple Outputs Directly

You can perform estimation with linear and nonlinear models for multiple-output data.

Tip Estimating multiple-output state-space models directly generally produces better results than estimating other types of multiple-output models directly.

Modeling Multiple Outputs as a Combination of Single-Output Models

You may find that it is harder for a single model to explain the behavior of several outputs. If you get a poor fit estimating a multiple-output model directly, you can try building models for subsets of the most important input and output channels.

Use this approach when no feedback is present in the dynamic system and there are no couplings between the outputs. If you are unsure about the presence of feedback, see "How to Analyze Data Using the advice Command" on page 2-102.

To construct partial models, use subreferencing to create partial data sets, such that each data set contains all inputs and one output. For more information about creating partial data sets, see the following topics:

- For working in the System Identification app, see "Create Data Sets from a Subset of Signal Channels" on page 2-33.
- For working at the command line, see the "Select Data Channels, I/O Data and Experiments in iddata Objects" on page 2-55.

After validating the single-output models, use vertical concatenation to combine these partial models into a single multiple-output model. For more information about concatenation, see "Increasing Number of Channels or Data Points of iddata Objects" on page 2-58 or "Adding Input or Output Channels in idfrd Objects" on page 2-88.

You can try refining the concatenated multiple-output model using the original (multiple-output) data set.

Improving Multiple-Output Estimation Results by Weighing Outputs During Estimation

When estimating linear and nonlinear black-box models for multiple-output systems, you can control the relative importance of output channels during the estimation process. The ability to control how much each output is weighed during estimation is useful when some of the measured outputs have poor accuracy or should be treated as less important during estimation. For example, if you have already modeled one output well, you might want to focus the estimation on modeling the remaining outputs. Similarly, you might want to refine a model for a subset of outputs.

Use the `OutputWeight` estimation option to indicate the desired output weighting. If you set this option to `'noise'`, an automatic weighting, equal to the inverse of the

estimated noise variance, is used for model estimation. You can also specify a custom weighting matrix, which must be a positive semi-definite matrix.

Note

- The `OutputWeight` option is not available for polynomial models, except ARX models, since their estimation algorithm estimates the parameters one output at a time.
- Transfer function (`idtf`) and process models (`idproc`) ignore `OutputWeight` when they contain nonzero or free transport delays. In the presence of delays, the estimation is carried out one output at a time.

For more information about the `OutputWeight` option, see the estimation option sets, such as `arxOptions`, `ssestOptions`, `tfestOptions`, `nlarxOptions`, and `nlhwOptions`.

Note For multiple-output `idnlarx` models containing `neuralnet` or `treepartition` nonlinearity estimators, output weighting is ignored because each output is estimated independently.

Regularized Estimates of Model Parameters

In this section...
"What Is Regularization?" on page 1-48
"When to Use Regularization" on page 1-51
"Choosing Regularization Constants" on page 1-53

What Is Regularization?

Regularization is the technique for specifying constraints on the flexibility of a model, thereby reducing uncertainty in the estimated parameter values.

Model parameters are obtained by fitting measured data to the predicted model response, such as a transfer function with three poles or a second-order state-space model. The model order is a measure of its flexibility — higher the order, the greater the flexibility. For example, a model with three poles is more flexible than one with two poles. Increasing the order causes the model to fit the observed data with increasing accuracy. However, the increased flexibility comes with the price of higher uncertainty in the estimates, measured by a higher value of random or variance error. On the other hand, choosing a model with too low an order leads to larger systematic errors. Such errors cannot be attributed to measurement noise and are also known as bias error.

Ideally, the parameters of a good model should minimize the mean square error (MSE), given by a sum of systematic error (bias) and random error (variance):

$$\text{MSE} = |\text{Bias}|^2 + \text{Variance}$$

The minimization is thus a tradeoff in constraining the model. A flexible (high order) model gives small bias and large variance, whereas a simpler (low order) model results in larger bias and smaller variance errors. Typically, you can investigate this tradeoff between bias and variance errors by cross-validation tests on a set of models of increasing flexibility. However, such tests do not always give full control in managing the parameter estimation behavior. For example:

- You cannot use the known (*a priori*) information about the model to influence the quality of the fits.
- In grey-box and other structured models, the order is fixed by the underlying ODEs and cannot be changed. If the data is not rich enough to capture the full range of dynamic behavior, this typically leads to high uncertainty in the estimated values.

- Varying the model order does not let you explicitly shape the variance of the underlying parameters.

Regularization gives you a better control over the bias versus variance tradeoff by introducing an additional term in the minimization criterion that penalizes the model flexibility. Without regularization, for a model with one output and no weighting, the parameter estimates are obtained by minimizing a weighted quadratic norm of the prediction errors $\varepsilon(t,\theta)$:

$$V_N(\theta) = \frac{1}{N}\sum_{t=1}^{N}\varepsilon^2(t,\theta)$$

where t is the time variable, N is the number of data samples, and $\varepsilon(t,\theta)$ is the predicted error computed as the difference between the observed output and the predicted output of the model.

Regularization modifies the cost function by adding a term proportional to the square of the norm of the parameter vector θ, so that the parameters θ are obtained by minimizing:

$$\hat{V}_N(\theta) = \frac{1}{N}\sum_{t=1}^{N}\varepsilon^2(t,\theta) + \frac{1}{N}\lambda\|\theta\|^2$$

where λ is a positive constant that has the effect of trading variance error in $V_N(\theta)$ for bias error — the larger the value of λ, the higher the bias and lower the variance of θ. The added term penalizes the parameter values with the effect of keeping their values small during estimation. In statistics, this type of regularization is called ridge regression. For more information, see “Introduction to Ridge Regression” (Statistics and Machine Learning Toolbox).

Note Another choice for the norm of θ vector is the L_1-norm, known as lasso regularization. However, System Identification Toolbox supports only the 2-norm based penalty, known as L_2 regularization, as shown in the previous equation.

The penalty term is made more effective by using a positive definite matrix R, which allows weighting and/or rotation of the parameter vector:

$$\hat{V}_N(\theta) = \frac{1}{N}\sum_{t=1}^{N}\varepsilon^2(t,\theta) + \frac{1}{N}\lambda\theta^T R\theta$$

The square matrix R gives additional freedom for:

- Shaping the penalty term to meet the required constraints, such as keeping the model stable
- Adding known information about the model parameters, such as reliability of the individual parameters in the θ vector

For structured models such as grey-box models, you may want to keep the estimated parameters close to their guess values to maintain the physical validity of the estimated

model. This can be achieved by generalizing the penalty term to $\lambda\left(\theta-\theta^*\right)^T R\left(\theta-\theta^*\right)$, such that the cost function becomes:

$$\hat{V}_N(\theta)=\frac{1}{N}\sum_{t=1}^{N}\varepsilon^2(t,\theta)+\frac{1}{N}\lambda\left(\theta-\theta^*\right)^T R\left(\theta-\theta^*\right)$$

Minimizing this cost function has the effect of estimating θ such that their values remain close to initial guesses θ*.

In regularization:

- θ* represents prior knowledge about the unknown parameters.
- λ*R represents the confidence in the prior knowledge of the unknown parameters. This implies that the larger the value, the higher the confidence.

A formal interpretation in a Bayesian setting is that θ has a prior distribution that is

Gaussian with mean θ* and covariance matrix $\sigma^2/\lambda R^{-1}$, where σ² is the variance of ε(t). The use of regularization can therefore be linked to some prior information about the system. This could be quite soft, such as the system is stable.

You can use the regularization variables λ and R as tools to find a good model that balances complexity and provides the best tradeoff between bias and variance. You can obtain regularized estimates of parameters for transfer function, state-space, polynomial, grey-box, process, and nonlinear black-box models. The three terms defining the penalty term, λ, R and θ*, are represented by regularization options `Lambda`, `R`, and `Nominal`, respectively in the toolbox. You can specify their values in the estimation option sets for both linear and nonlinear models. In the System Identification app, click **Regularization** in the linear model estimation dialog box or **Estimation Options** in the Nonlinear Models dialog box.

When to Use Regularization

Use regularization for:

- “Identifying Overparameterized Models” on page 1-51
- “Imposing A Priori Knowledge of Model Parameters in Structured Models” on page 1-52
- “Incorporating Knowledge of System Behavior in ARX and FIR Models” on page 1-52

Identifying Overparameterized Models

Over-parameterized models are rich in parameters. Their estimation typically yields parameter values with a high level of uncertainty. Over-parameterization is common for nonlinear ARX (`idnlarx`) models and can also be for linear state-space models using free parameterization.

In such cases, regularization improves the numerical conditioning of the estimation. You can explore the bias-vs.-variance tradeoff using various values of the regularization constant `Lambda`. Typically, the `Nominal` option is its default value of `0`, and `R` is an identity matrix such that the following cost function is minimized:

$$\hat{V}_N(\theta) = \frac{1}{N}\sum_{t=1}^{N}\varepsilon^2(t,\theta) + \frac{1}{N}\lambda\|\theta\|^2$$

In the following example, a nonlinear ARX model estimation using a large number of neurons leads to an ill-conditioned estimation problem.

```
% Load estimation data.
load regularizationExampleData.mat nldata
% Estimate model without regularization.
Orders = [1 2 1];
NL = sigmoidnet('NumberOfUnits',30);
sys = nlarx(nldata,Orders,NL);
compare(nldata,sys)
```

Applying even a small regularizing penalty produces a good fit for the model to the data.

```
% Estimate model using regularization constant λ = 1e-8.
opt = nlarxOptions;
opt.Regularization.Lambda = 1e-8;
sysr = nlarx(nldata,Orders,NL,opt);
compare(nldata,sysr)
```

Imposing A Priori Knowledge of Model Parameters in Structured Models

In models derived from differential equations, the parameters have physical significance. You may have a good guess for typical values of those parameters even if the reliability of the guess may be different for each parameter. Because the model structure is fixed in such cases, you cannot simplify the structure to reduce variance errors.

Using the regularization constant `Nominal`, you can keep the estimated values close to their initial guesses. You can also design `R` to reflect the confidence in the initial guesses of the parameters. For example, if θ is a 2-element vector and you can guess the value of the first element with more confidence than the second one, set `R` to be a diagonal matrix of size 2-by-2 such that R(1,1) >> R(2,2).

In the following example, a model of a DC motor is parameterized by static gain `G` and time constant τ. From prior knowledge, suppose you know that `G` is about 4 and τ is about 1. Also, assume that you have more confidence in the value of τ than `G` and would like to guide the estimation to remain close to the initial guess.

```
% Load estimation data.
load regularizationExampleData.mat motorData
% Create idgrey model for DC motor dynamics.
mi = idgrey(@DCMotorODE,{'G',4;'Tau',1},'cd',{}, 0);
mi = setpar(mi,'label','default');
% Configure Regularization options.
opt = greyestOptions;
opt.Regularization.Lambda = 100;
% Specify that the second parameter better known than the first.
opt.Regularization.R = [1, 1000];
% Specify initial  guess as Nominal.
opt.Regularization.Nominal = 'model';
% Estimate model.
sys = greyest(motorData,mi,opt)
getpar(sys)
```

Incorporating Knowledge of System Behavior in ARX and FIR Models

In many situations, you may know the shape of the system impulse response from impact tests. For example, it is quite common for stable systems to have an impulse response that is smooth and exponentially decaying. You can use such prior knowledge of system behavior to derive good values of regularization constants for linear-in-parameter models such as ARX and FIR structure models using the `arxRegul` command.

For black-box models of arbitrary structure, it is often difficult to determine the optimal values of `Lambda` and `R` that yield the best bias-vs.-variance tradeoff. Therefore, it is recommended that you start by obtaining the regularized estimate of an ARX or FIR structure model. Then, convert the model to a state-space, transfer function or polynomial model using the `idtf`, `idss`, or `idpoly` commands, followed by order reduction if required.

In the following example, direct estimation of a 15th order continuous-time transfer function model fails due to numerical ill-conditioning.

```
% Load estimation data.
load dryer2
Dryer = iddata(y2,u2,0.08);
Dryerd = detrend(Dryer,0);
Dryerde = Dryerd(1:500);
xe = Dryerd(1:500);
ze = Dryerd(1:500);
zv = Dryerd(501:end);
% Estimate model without regularization.
sys1 = tfest(ze,15);
```

Therefore, use regularized ARX estimation and then convert the model to transfer function structure.

```
% Specify regularization constants.
[L, R] = arxRegul(ze,[15 15 1]);
optARX = arxOptions;
optARX.Regularization.Lambda = L;
optARX.Regularization.R = R;
% Estimate ARX model.
sysARX = arx(ze,[15 15 1],optARX);
% Convert model to continuous time.
sysc = d2c(sysARX);
% Convert model to transfer function.
sys2 = idtf(sysc);
% Validate the models sys1 and sys2.
compare(zv,sys1,sys2)
```

Choosing Regularization Constants

A guideline for selecting the regularization constants λ and `R` is in the Bayesian interpretation. The added penalty term is an assumption that the parameter vector θ is a Gaussian random vector with mean θ* and covariance matrix $\sigma^2 / \lambda R^{-1}$.

You can relate naturally to such an assumption for a grey-box model, where the parameters are of known physical interpretation. In other cases, this may be more difficult. Then, you have to use ridge regression (`R` = 1; θ* = 0) and tune λ by trial and error.

Use the following techniques for determining λ and `R` values:

- "Incorporate Prior Information Using Tunable Kernels" on page 1-54
- "Perform Cross-Validation Tests" on page 1-54

Incorporate Prior Information Using Tunable Kernels

Tuning the regularization constants for ARX models in `arxRegul` is based on simple assumptions about the properties of the true impulse responses.

In the case of an FIR model, the parameter vector contains the impulse response coefficients b_k for the system. From prior knowledge of the system, it is often known that the impulse response is smooth and exponentially decaying:

$$E\left[b_k\right]^2 = C\mu^k, \quad corr\{b_k b_{k-1}\} = \rho$$

where *corr* means correlation. The equation is a parameterization of the regularization constants in terms of coefficients C, μ, and ρ and the chosen shape (decaying polynomial) is called a kernel. The kernel thus contains information about parameterization of the prior covariance of the impulse response coefficients.

You can estimate the parameters of the kernel by adjusting them to the measured data using the `RegulKernel` input of the `arxRegul` command. For example, the `DC` kernel

estimates all three parameters while the `TC` kernel links $\rho = \sqrt{\mu}$. This technique of tuning kernels applies to all linear-in-parameter models such as ARX and FIR models.

Perform Cross-Validation Tests

A general way to test and evaluate any regularization parameters is to estimate a model based on certain parameters on an estimation data set, and evaluate the model fit for another validation data set. This is known as cross-validation.

Cross-validation is entirely analogous to the method for selecting model order:

1 Generate a list of candidate λ and `R` values to be tested.

2 Estimate a model for each candidate regularization constant set.

3 Compare the model fit to the validation data.

4 Use the constants that give the best fit to the validation data.

For example:

```
% Create estimation and validation data sets.
ze = z(1:N/2);
zv = z(N/2:end);
% Specify regularization options and estimate models.
opt = ssestOptions;
for tests = 1:M
opt.Regularization.Lambda = Lvalue(test);
opt.Regularization.R = Rvalue(test);
m{test} = ssest(ze,order,opt);
end
% Compare models with validation data for model fit.
[~,fit] = compare(zv,m{:})
```

References

[1] L. Ljung. “Some Classical and Some New Ideas for Identification of Linear Systems.” *Journal of Control, Automation and Electrical Systems.* April 2013, Volume 24, Issue 1-2, pp 3-10.

[2] L. Ljung, and T. Chen. “What can regularization offer for estimation of dynamical systems?” *In Proceedings of IFAC International Workshop on Adaptation and Learning in Control and Signal Processing*, ALCOSP13, Caen, France, July 2013.

[3] L. Ljung, and T. Chen. “Convexity issues in system identification.” *In Proceedings of the 10th IEEE International Conference on Control & Automation*, ICCA 2013, Hangzhou, China, June 2013.

See Also

Related Examples

- “Estimate Regularized ARX Model Using System Identification App” on page 1-57
- “Regularized Identification of Dynamic Systems” on page 1-77

More About

- “Loss Function and Model Quality Metrics” on page 1-64

Estimate Regularized ARX Model Using System Identification App

This example shows how to estimate regularized ARX models using automatically generated regularization constants in the System Identification app.

Open a saved System Identification App session.

```
filename = fullfile(matlabroot,'help','toolbox',...
           'ident','examples','ex_arxregul.sid');
systemIdentification(filename)
```

The session imports the following data and model into the System Identification app:

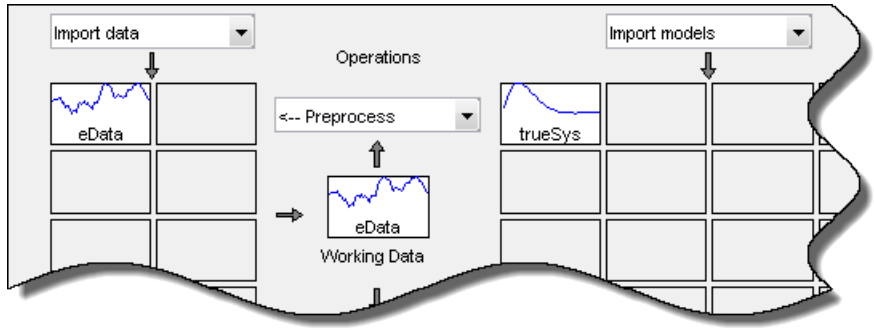
- Estimation data `eData`

  The data is collected by simulating a system with the following known transfer function:

  $$G(z) = \frac{0.02008 + 0.04017z^{-1} + 0.02008z^{-2}}{1 - 1.56z^{-1} + 0.6414z^{-2}}$$

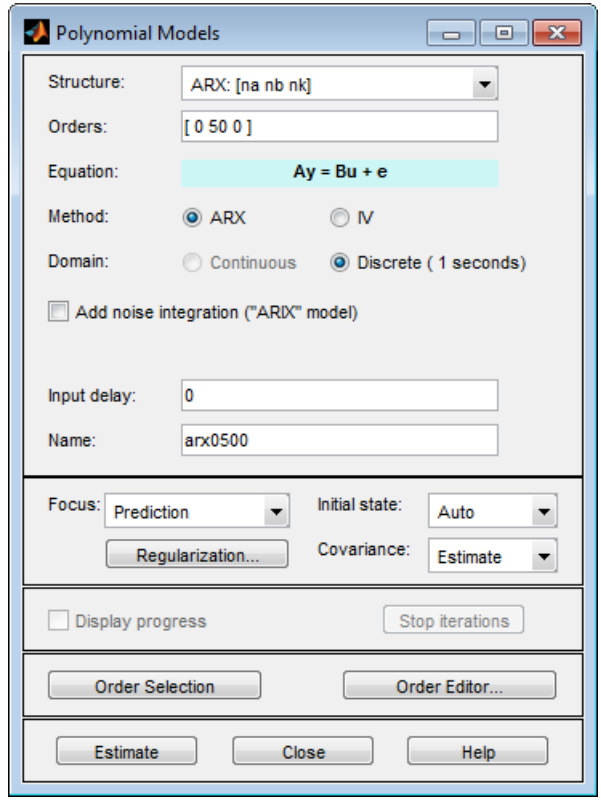
- Transfer function model `trueSys`

  `trueSys` is the transfer function model used to generate the estimation data `eData` described previously. You also use the impulse response of this model later to compare the impulse responses of estimated ARX models.



Estimate a 50th-order ARX model.

1. In the System Identification app, select **Estimate** > **Polynomial Models** to open the Polynomial Models dialog box.
2. Verify that ARX is selected in the **Structure** list.
3. In the **Orders** field, specify [0 50 0] as the ARX model order and delay.

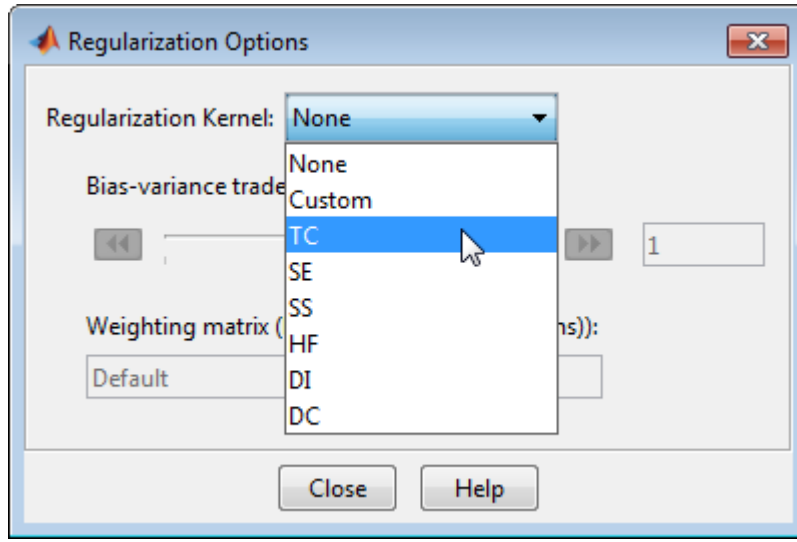


4. Click **Estimate** to estimate the model.

A model `arx0500` is added to the System Identification app.

Estimate a 50th-order regularized ARX model.

1. In the Polynomial Models dialog box, click **Regularization**.
2. In the Regularization Options dialog box, select `TC` from the **Regularization Kernel** drop-down list.



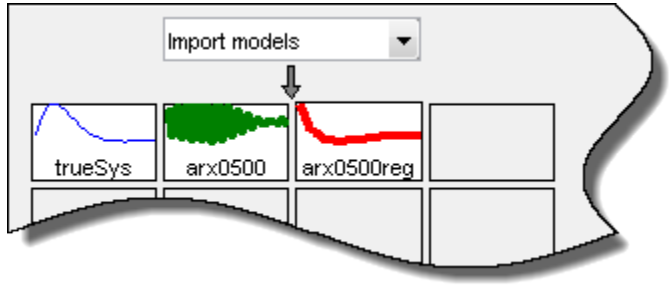
Specifying this option automatically determines regularization constants using the `TC` regularization kernel. To learn more, see the `arxRegul` reference page.

Click **Close** to close the dialog box.

3 In the **Name** field in the Polynomial Models dialog box, type `arx0500reg`.

4 Click **Estimate**.

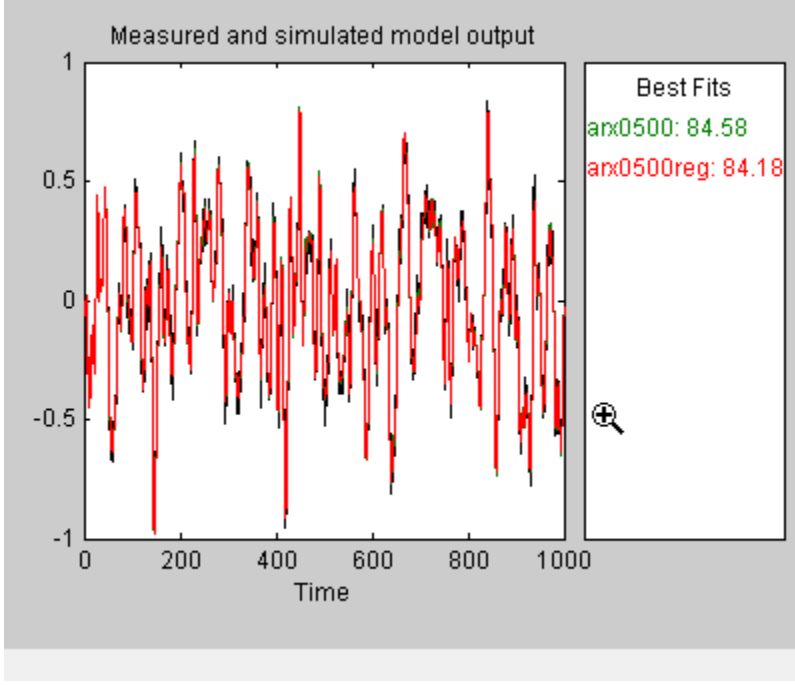
A model `arx0500reg` is added to the System Identification app.



Compare the unregularized and regularized model outputs to estimation data.

Select the **Model output** check box in the System Identification app.

The Measured and simulated model output plot shows that both the models have an 84% fit with the data.

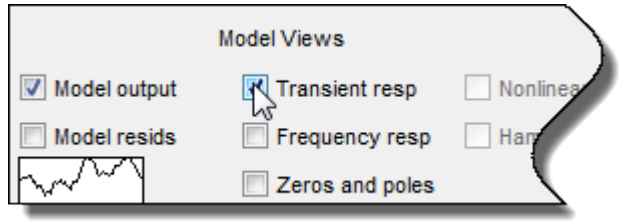


Determine if regularization leads to parameter values with less variance.

Because the model fit to the estimation data is similar with and without using regularization, compare the impulse response of the ARX models with the impulse responses of `trueSys`, the system used to collect the estimation data.

1 Click the `trueSys` icon in the model board of the System Identification app.

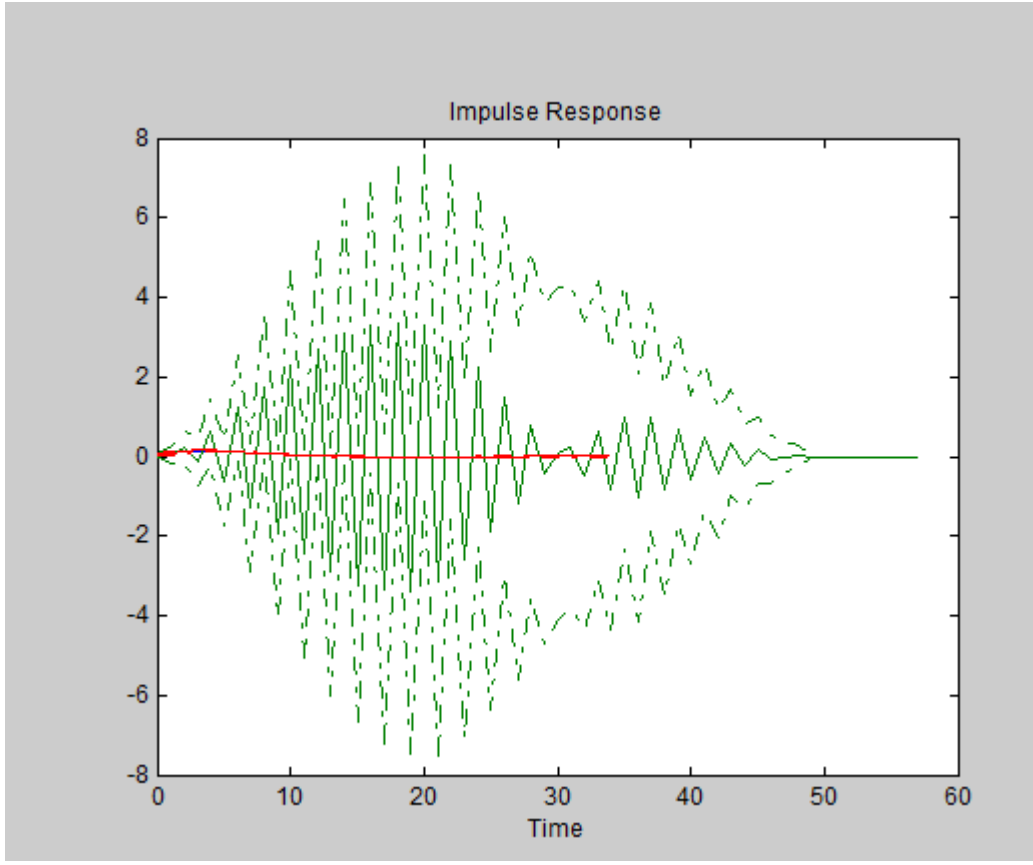
2 Select the **Transient resp** check box to open the Transient Response plot window.



By default, the plot shows the step response.

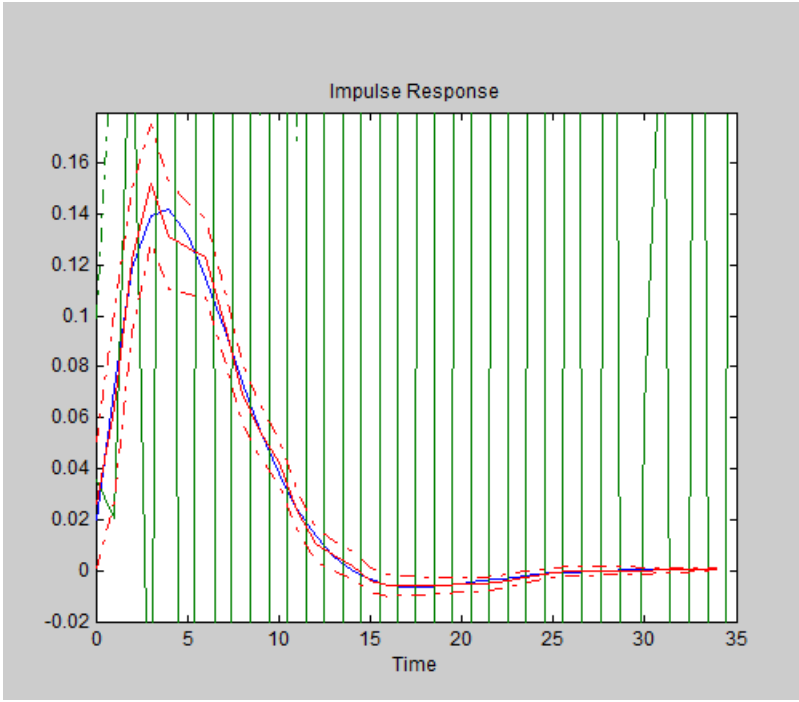
3 In the Transient response plot window, select **Options** > **Impulse response** to change to plot to display the impulse response.

4 Select **Options** > **Show 99% confidence intervals** to plot the confidence intervals.



The plot shows that the impulse response of the unregularized model `arx0500` is far off from the true system and has huge uncertainties.

To get a closer look at the model fits to the data and the variances, magnify a portion of the plot.



The fit of the regularized ARX model `arx0500reg` closely matches the impulse response of the true system and the variance is greatly reduced as compared to the unregularized model.

See Also

Related Examples

- “Regularized Identification of Dynamic Systems” on page 1-77

More About

- “Regularized Estimates of Model Parameters” on page 1-48

Loss Function and Model Quality Metrics

In this section...
"What is a Loss Function?" on page 1-64
"Options to Configure the Loss Function" on page 1-65
"Model Quality Metrics" on page 1-73

What is a Loss Function?

The System Identification Toolbox software estimates model parameters by minimizing the error between the model output and the measured response. This error, called loss function or cost function, is a positive function of prediction errors $e(t)$. In general, this function is a weighted sum of squares of the errors. For a model with ny-outputs, the loss function $V(\theta)$ has the following general form:

$$V(\theta) = \frac{1}{N}\sum_{t=1}^{N} e^T(t,\theta) W(\theta) e(t,\theta)$$

where:

- N is the number of data samples.
- $e(t,\theta)$ is ny-by-1 error vector at a given time t, parameterized by the parameter vector θ.
- $W(\theta)$ is the weighting matrix, specified as a positive semidefinite matrix. If W is a diagonal matrix, you can think of it as a way to control the relative importance of outputs during multi-output estimations. When W is a fixed or known weight, it does not depend on θ.

The software determines the parameter values by minimizing $V(\theta)$ with respect to θ.

For notational convenience, $V(\theta)$ is expressed in its matrix form:

$$V(\theta) = \frac{1}{N} trace\left(E^T(\theta) E(\theta) W(\theta) \right)$$

$E(\theta)$ is the error matrix of size N-by-ny. The i:th row of $E(\theta)$ represents the error value at time $t = i$.

The exact form of $V(\theta)$ depends on the following factors:

- Model structure. For example, whether the model that you want to estimate is an ARX or a state-space model.
- Estimator and estimation options. For example, whether you are using `n4sid` or `ssest` estimator and specifying options such as `Focus` and `OutputWeight`.

Options to Configure the Loss Function

You can configure the loss function for your application needs. The following estimation options, when available for the estimator, configure the loss function:

Estimation Option	Description	Notes
`Focus`	`Focus` option affects how $e(t)$ in the loss function is computed: • When `Focus` is `'prediction'`, $e(t)$ represents 1-step ahead prediction error: $e_p(t) = y_{measured}(t) - y_{predicted}(t)$ • When `Focus` is `'simulation'`, $e(t)$ represents the simulation error: $e_s(t) = y_{measured}(t) - y_{simulated}(t)$ **Note** For models whose noise component is trivial, ($H(q) = 1$), $e_p(t)$, and $e_s(t)$ are equivalent. The `Focus` option can also be interpreted as a weighting filter in the loss function. For more information, see "Effect of Focus and WeightingFilter Options on the Loss Function" on page 1-70.	• Specify the `Focus` option in the estimation option sets. • The estimation option sets for `oe` and `tfest` do not have a `Focus` option because the noise-component for the estimated models is trivial, and so $e_p(t)$ and $e_s(t)$ are equivalent.

Estimation Option	Description	Notes
`WeightingFilter`	When you specify a weighting filter, prefiltered prediction or simulation error is minimized: $e_f(t) = \mathrm{L}(e(t))$ where `L(.)` is a linear filter. The `WeightingFilter` option can be interpreted as a custom weighting filter that is applied to the loss function. For more information, see "Effect of Focus and WeightingFilter Options on the Loss Function" on page 1-70.	• Specify the `WeightingFilter` option in the estimation option sets. Not all options for `WeightingFilter` are available for all estimation commands.

Estimation Option	Description	Notes
`EnforceStability`	When `EnforceStability` is `true`, the minimization objective also contains a constraint that the estimated model must be stable.	• Specify the `EnforceStability` option in the estimation option sets. • The estimation option sets for `procest` and `ssregest` commands do not have an `EnforceStability` option. These estimation commands always yield a stable model. • The estimation commands `tfest` and `oe` always yield a stable model when used with time-domain estimation data. • Identifying unstable plants requires data collection under a closed loop with a stabilizing feedback controller. A reliable estimation of the plant dynamics requires a sufficiently rich noise component in the model structure to separate out the plant dynamics from feedback effects. As a result, models that use a trivial noise component ($H(q) = 1$),

Estimation Option	Description	Notes
		such as models estimated by `tfest` and `oe` commands, do not estimate good results for unstable plants.
`OutputWeight`	`OutputWeight` option configures the weighting matrix $W(\theta)$ in the loss function and lets you control the relative importance of output channels during multi-output estimations. • When `OutputWeight` is `'noise'`, $W(\theta)$ equals the inverse of the estimated variance of error $e(t)$: $$W(\theta)=\left(\frac{1}{N}E^T(\theta)E(\theta)\right)^{-1}$$ Because W depends on θ, the weighting is determined as a part of the estimation. Minimization of the loss function with this weight simplifies the loss function to: $$V(\theta)=det\left(\frac{1}{N}E^T(\theta)E(\theta)\right)$$ Using the inverse of the noise variance is the optimal weighting in the maximum likelihood sense. • When `OutputWeight` is an ny-by-ny positive semidefinite matrix, a constant weighting is used. This loss function then becomes a weighted sum of squared errors.	• Specify the `OutputWeight` option in the estimation option sets. Not all options for `OutputWeight` are available for all estimation commands. • `OutputWeight` is not available for polynomial model estimation because such models are always estimated one output at a time. • `OutputWeight` cannot be `'noise'` when `SearchMethod` is `'lsqnonlin'`.

Estimation Option	Description	Notes
`ErrorThreshold`	`ErrorThreshold` option specifies the threshold for when to adjust the weight of large errors from quadratic to linear. Errors larger than `ErrorThreshold` times the estimated standard deviation have a linear weight in the loss function. $$V(\theta)=\frac{1}{N}\left(\sum_{t\in I}e^{T}(t,\theta)W(\theta)e(t,\theta)+\sum_{t\in J}v^{T}(t,\theta)W(\theta)v(t,\theta)\right)$$ where: • I represents those time instants for which $\left\|e(t)\right\| < \rho * \sigma$, where ρ is the error threshold. • J represents the complement of I, that is, the time instants for which $\left\|e(t)\right\| >= \rho * \sigma$. • σ is the estimated standard deviation of the error. The error $v(t,\theta)$ is defined as: $$v(t,\theta)=e(t,\theta)*\sigma\frac{\rho}{\sqrt{\left\|e(t,\theta)\right\|}}$$	• Specify the `ErrorThreshold` option in the estimation option sets. • A typical value for the error threshold ρ = 1.6 minimizes the effect of data outliers on the estimation results.

Estimation Option	Description	Notes
`Regularization`	`Regularization` option modifies the loss function to add a penalty on the variance of the estimated parameters. The loss function is set up with the goal of minimizing the prediction errors. It does not include specific constraints on the variance (a measure of reliability) of estimated parameters. This can sometimes lead to models with large uncertainty in estimated model parameters, especially when the model has many parameters. `Regularization` introduces an additional term in the loss function that penalizes the model flexibility: $$V(\theta)=\frac{1}{N}\sum_{t=1}^{N}e^{T}(t,\theta)W(\theta)e(t,\theta)+\frac{1}{N}\lambda\left(\theta-\theta^{*}\right)^{T}R\left(\theta-\theta^{*}\right)$$ The second term is a weighted (R) and scaled (λ) variance of the estimated parameter set θ about its nominal value θ^*.	• Specify the `Regularization` option in the estimation option sets. • For linear-in-parameter models (FIR models) and ARX models, you can compute optimal values of the regularization variables R and λ using the `arxRegul` command.

Effect of Focus and WeightingFilter Options on the Loss Function

The `Focus` option can be interpreted as a weighting filter in the loss function. The `WeightingFilter` option is an additional custom weighting filter that is applied to the loss function.

To understand the effect of `Focus` and `WeightingFilter`, consider a linear single-input single-output model:

$$y(t) = G(q,\theta)\,u(t) + H(q,\theta)\,e(t)$$

Where $G(q,\theta)$ is the measured transfer function, $H(q,\theta)$ is the noise model, and $e(t)$ represents the additive disturbances modeled as white Gaussian noise. q is the time-shift operator.

In frequency domain, the linear model can be represented as:

$$Y(\omega) = G(\omega,\theta)U(\omega) + H(\omega,\theta)E(\omega)$$

where $Y(\omega)$, $U(\omega)$, and $E(\omega)$ are the Fourier transforms of the output, input, and output error, respectively. $G(\omega,\theta)$ and $H(\omega,\theta)$ represent the frequency response of the input-output and noise transfer functions, respectively.

The loss function to be minimized for the SISO model is given by:

$$V(\theta) = \frac{1}{N}\sum_{t=1}^{N} e^T(t,\theta)e(t,\theta)$$

Using Parseval's Identity, the loss function in frequency-domain is:

$$V(\theta,\omega) = \frac{1}{N}\left\|E(\omega)\right\|^2$$

Substituting for $E(\omega)$ gives:

$$V(\theta,\omega) = \frac{1}{N}\left\|\frac{Y(\omega)}{U(\omega)} - G(\theta,\omega))\right\|^2 \frac{\left\|U(\omega)\right\|^2}{\left\|H(\theta,\omega)\right\|^2}$$

Thus, you can interpret minimizing the loss function V as fitting $G(\theta,\omega)$ to the empirical transfer function $Y(\omega)/U(\omega)$, using $\frac{\left\|U(\omega)\right\|^2}{\left\|H(\theta,\omega)\right\|^2}$ as a weighting filter. This corresponds to specifying `Focus` as `'prediction'`. The estimation emphasizes frequencies where input has more power ($\left\|U(\omega)\right\|^2$ is greater) and de-emphasizes frequencies where noise is significant ($\left\|H(\theta,\omega)\right\|^2$ is large).

When `Focus` is specified as `'simulation'`, the inverse weighting with $\left\|H(\theta,\omega)\right\|^2$ is not used. That is, only the input spectrum is used to weigh the relative importance of the estimation fit in a specific frequency range.

When you specify a linear filter `L` as `WeightingFilter`, it is used as an additional custom weighting in the loss function.

$$V(\theta) = \frac{1}{N^2}\left\| \frac{Y(\omega)}{U(\omega)} - G(\theta)) \right\|^2 \frac{\|U(\omega)\|^2}{\|H(\theta)\|^2} \|\mathrm{L}(\omega)\|^2$$

Here $\mathrm{L}(\omega)$ is the frequency response of the filter. Use $\mathrm{L}(\omega)$ to enhance the fit of the model response to observed data in certain frequencies, such as to emphasize the fit close to system resonant frequencies.

The estimated value of input-output transfer function G is the same as what you get if you instead first prefilter the estimation data with `L(.)` using `idfilt`, and then estimate the model without specifying `WeightingFilter`. However, the effect of `L(.)` on the estimated noise model H depends on the choice of `Focus`:

- **`Focus is 'prediction'`** — The software minimizes the weighted prediction error $e_f(t) = \mathrm{L}(e_p(t))$, and the estimated model has the form:
  $y(t) = G(q)u(t) + H_1(q)e(t)$

  Where $H_1(q) = H(q) / \mathrm{L}(q)$. Thus, the estimation with prediction focus creates a biased estimate of H. This is the same estimated noise model you get if you instead first prefilter the estimation data with `L(.)` using `idfilt`, and then estimate the model.

  When H is parameterized independent of G, you can treat the filter `L(.)` as a way of affecting the estimation bias distribution. That is, you can shape the trade-off between fitting G to the system frequency response and fitting H / L to the disturbance spectrum when minimizing the loss function. For more details see, section 14.4 in *System Identification: Theory for the User*, Second Edition, by Lennart Ljung, Prentice Hall PTR, 1999.
- **`Focus is 'simulation'`** — The software first estimates G by minimizing the weighted simulation error $e_f(t) = \mathrm{L}(e_s(t))$, where $e_s(t) = y_{measured}(t) - G(q)u_{measured}(t)$. Once G is estimated, the software fixes it and computes H by minimizing pure prediction errors $e(t)$ using unfiltered data. The estimated model has the form:

$$y(t) = G(q)u(t) + He(t)$$

If you prefilter the data first, and then estimate the model, you get the same estimate for G but get a biased noise model $H / \mathtt{L}$.

Thus, the `WeightingFilter` has the same effect as prefiltering the estimation data for estimation of G. For estimation of H, the effect of `WeightingFilter` depends upon the choice of `Focus`. A prediction focus estimates a biased version of the noise model $H / \mathtt{L}$, while a simulation focus estimates H. Prefiltering the estimation data, and then estimating the model always gives $H / \mathtt{L}$ as the noise model.

Model Quality Metrics

After you estimate a model, use model quality metrics to assess the quality of identified models, compare different models, and pick the best one. The `Report.Fit` property of an identified model stores various metrics such as `FitPercent`, `LossFcn`, `FPE`, `MSE`, `AIC`, `nAIC`, `AICc`, and `BIC` values.

- `FitPercent`, `LossFcn`, and `MSE` are measures of the actual quantity that is minimized during the estimation. For example, if `Focus` is `'simulation'`, these quantities are computed for the simulation error $e_s(t)$. Similarly, if you specify the `WeightingFilter` option, then `LossFcn`, `FPE`, and `MSE` are computed using filtered residuals $e_f(t)$.
- `FPE`, `AIC`, `nAIC`, `AICc`, and `BIC` measures are computed as properties of the output disturbance according to the relationship:

  $$y(t) = G(q)u(t) + H(q)e(t)$$

  $G(q)$ and $H(q)$ represent the measured and noise components of the estimated model.

  Regardless of how the loss function is configured, the error vector $e(t)$ is computed as 1-step ahead prediction error using a given model and a given dataset. This implies that even when the model is obtained by minimizing the simulation error $e_s(t)$, the FPE and various AIC values are still computed using the prediction error $e_p(t)$. The actual value of $e_p(t)$ is determined using the `pe` command with prediction horizon of 1 and using the initial conditions specified for the estimation.

These metrics contain two terms — one for describing the model accuracy and another to describe its complexity. For example, in FPE, $det\left(\frac{1}{N}E^T E\right)$ describes the model accuracy and $\frac{1+\frac{np}{N}}{1-\frac{np}{N}}$ describes the model complexity.

By comparing models using these criteria, you can pick a model that gives the best (smallest criterion value) trade-off between accuracy and complexity.

Quality Metric	Description
`FitPercent`	Normalized Root Mean Squared Error (NRMSE) expressed as a percentage, defined as: $FitPercent = 100\left(1 - \frac{\|y_{measured} - y_{model}\|}{\|y_{measured} - \overline{y_{measured}}\|}\right)$ where: • y_{measured} is the measured output data. • $\overline{y_{measured}}$ is its (channel-wise) mean. • y_{model} is the simulated or predicted response of the model, governed by the `Focus`. • `||.||` indicates the 2-norm of a vector. `FitPercent` varies between `-Inf` (bad fit) to 100 (perfect fit). If the value is equal to zero, then the model is no better at fitting the measured data than a straight line equal to the mean of the data.
`LossFcn`	Value of the loss function when the estimation completes. It contains effects of error thresholds, output weight, and regularization used for estimation.

Quality Metric	Description
`MSE`	Mean Squared Error measure, defined as: $$MSE = \frac{1}{N}\sum_{t=1}^{N} e^T(t)e(t)$$ where: • $e(t)$ is the signal whose norm is minimized for estimation. • N is the number of data samples in the estimation dataset.
`FPE`	Akaike's Final Prediction Error (FPE), defined as: $$FPE = det\left(\frac{1}{N}E^T E\right)\left(\frac{1+\frac{n_p}{N}}{1-\frac{n_p}{N}}\right)$$ where: • n_p is the number of free parameters in the model. n_p includes the number of estimated initial states. • N is the number of samples in the estimation dataset. • E is the N-by-n_y matrix of prediction errors, where n_y is the number of output channels.
`AIC`	A raw measure of Akaike's Information Criterion, defined as: $$AIC = N * log\left(det\left(\frac{1}{N}E^T E\right)\right) + 2 * n_p + N\left(n_y * \log(2\pi) + 1\right)$$

Quality Metric	Description
`AICc`	Small sample-size corrected Akaike's Information Criterion, defined as: $$AICc = AIC + 2*n_p*\frac{(n_p+1)}{(N-n_p-1)}$$ This metric is often more reliable for picking a model of optimal complexity from a list of candidate models when the data size N is small.
`nAIC`	Normalized measure of Akaike's Information Criterion, defined as: $$nAIC = log\left(det\left(\frac{1}{N}E^T E\right)\right)+\frac{2*n_p}{N}$$
`BIC`	Bayesian Information Criterion, defined as: $$BIC = N*log\left(det\left(\frac{1}{N}E^T E\right)\right)+N*\left(n_y*\log(2\pi)+1\right)+n_p*\log(N)$$

See Also

`aic` | `fpe` | `goodnessofFit` | `nparams` | `pe` | `predict` | `sim`

More About

- "System Identification Overview"
- "Why Simulate or Predict Model Output?" on page 17-9
- "Assigning Estimation Weightings" on page 6-21
- "Modeling Multiple-Output Systems" on page 1-45
- "Regularized Estimates of Model Parameters" on page 1-48

Regularized Identification of Dynamic Systems

This example shows the benefits of regularization for identification of linear and nonlinear models.

What is Regularization

When a dynamic system is identified using measured data, the parameter estimates are determined as:

$$\hat{\theta} = arg\min_{\theta} V_N(\theta)$$

where the criterion typically is a weighted quadratic norm of the prediction errors $\varepsilon(t,\theta)$. An L_2 regularized criterion is modified as:

$$\hat{\theta} = arg\min_{\theta} V_N(\theta) + \lambda(\theta - \theta^*)^T R(\theta - \theta^*)$$

A common special case of this is when $\theta^* = 0, R = I$. This is called *ridge regression* in statistics, e.g, see the `ridge` command in Statistics and Machine Learning Toolbox™.

A useful way of thinking about regularization is that θ^* represents prior knowledge about the unknown parameter vector and that $\lambda * R$ describes the confidence in this knowledge. (The larger $\lambda * R$, the higher confidence). A formal interpretation in a Bayesian setting is that θ has a *prior distribution* that is Gaussian with mean θ^* and covariance matrix $\sigma^2/\lambda R^{-1}$, where σ^2 is the variance of the innovations.

The use of regularization can therefore be linked to some prior information about the system. This could be quite soft, like that the system is stable. The regularization variables λ and R can be seen as tools to find a good model complexity for best tradeoff between bias and variance. The regularization constants λ and R are represented by options called `Lambda` and `R` respectively in System Identification Toolbox™. The choice of θ^* is controlled by the `Nominal` regularization option.

Bias - Variance Tradeoff in FIR modeling

Consider the problem of estimating the impulse response of a linear system as an FIR model:

$$y(t) = \sum_{k=0}^{nb} g(k)u(t-k)$$

These are estimated by the command: `m = arx(z,[0 nb 0])`. The choice of order `nb` is a tradeoff between bias (large `nb` is required to capture slowly decaying impulse responses without too much error) and variance (large `nb` gives many parameters to estimate which gives large variance).

Let us illustrate it with a simulated example. We pick a simple second order butterworth filter as system:

$$G(z) = \frac{0.02008 + 0.04017z^{-}1 + 0.02008z^{-}2}{1 - 1.561z^{-}1 + 0.6414z^{-}2}$$

Its impulse response is shown in Figure 1:

```
trueSys = idtf([0.02008 0.04017 0.02008],[1 -1.561 0.6414],1);
[y0,t] = impulse(trueSys);
plot(t,y0)
xlabel('Time (seconds)'), ylabel('Amplitude'), title('Impulse Response')
```

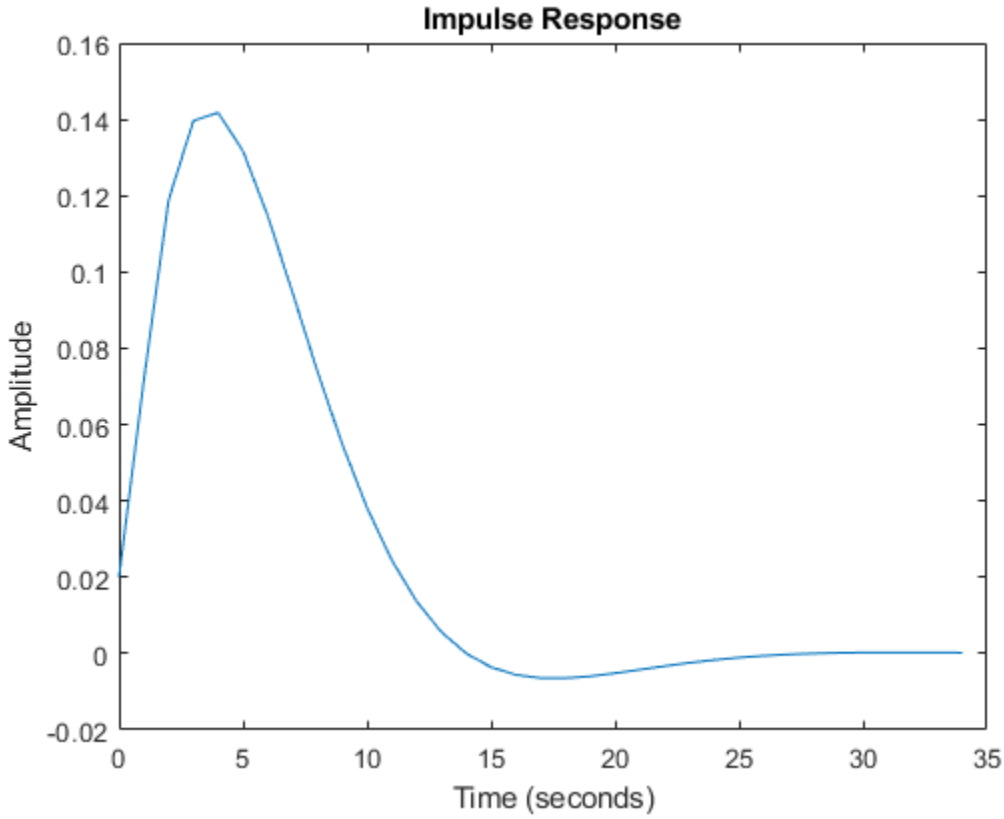


Figure 1: The true impulse response.

The impulse response has decayed to zero after less than 50 samples. Let us estimate it from data generated by the system. We simulate the system with low-pass filtered white noise as input and add a small white noise output disturbance with variance 0.0025 to the output. 1000 samples are collected. This data is saved in the regularizationExampleData.mat file and shown in Figure 2.

```
load regularizationExampleData.mat eData
plot(eData)
```

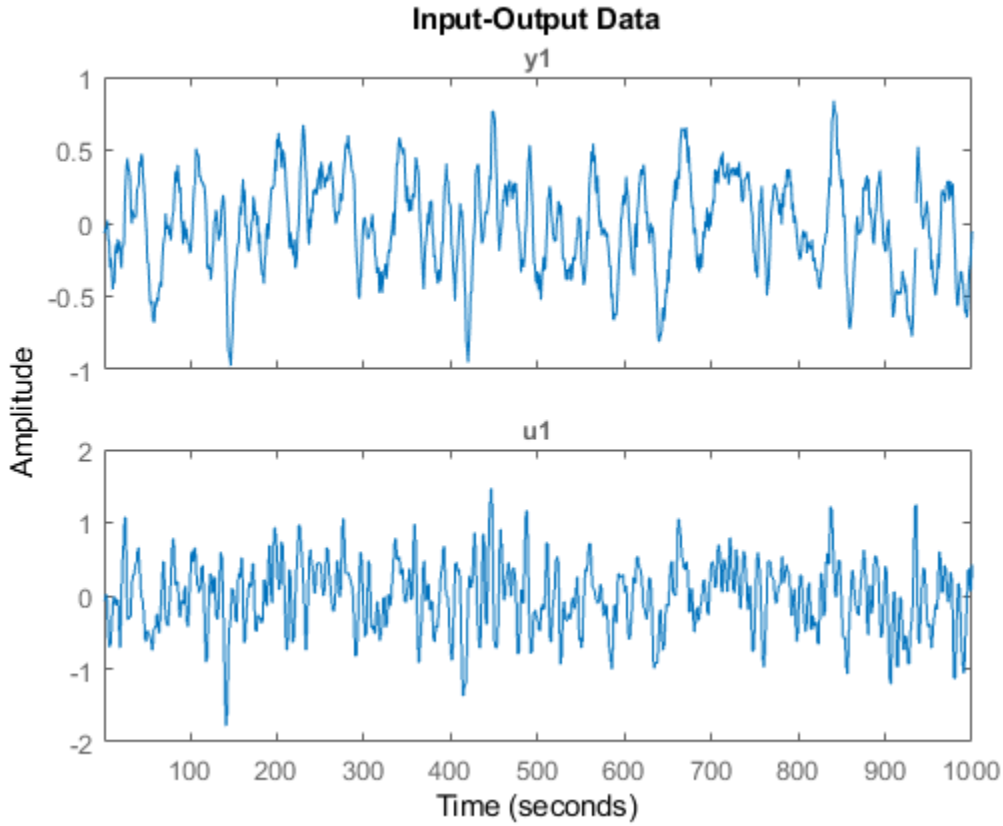


Figure 2: The data used for estimation.

To determine a good value for `nb` we basically have to try a few values and by some validation procedure evaluate which is best. That can be done in several ways, but since we know the true system in this case, we can determine the theoretically best possible value, by trying out all models with `nb=1,...,50` and find which one has the best fit to the true impulse response. Such a test shows that `nb = 13` gives the best error norm (mse = 0.2522) to the impulse response. This estimated impulse response is shown together with the true one in Figure 3.

```
nb = 13;
m13 = arx(eData,[0 nb 0]);
```

```
[y13,~,~,y13sd] = impulse(m13,t);
plot(t,y0,t,y13)
xlabel('Time (seconds)'), ylabel('Amplitude'), title('Impulse Response')
legend('True system','13:th order FIR model')
```

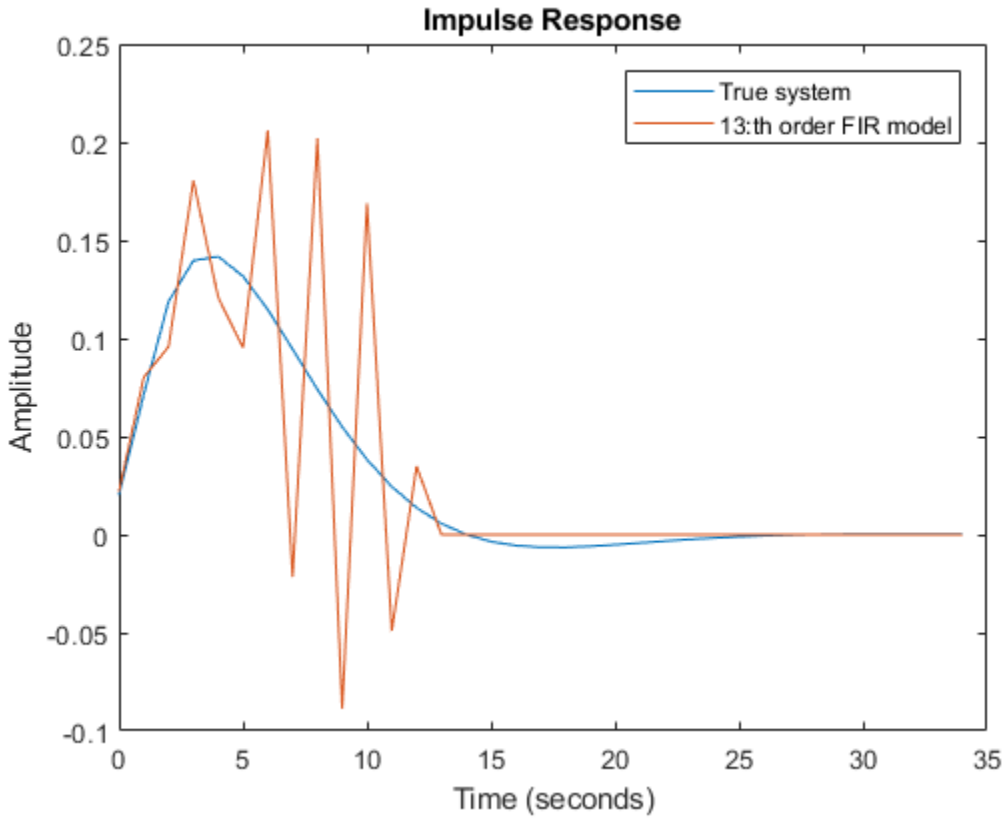


Figure 3: The true impulse response together with the estimate for order `nb = 13`.

Despite the 1000 data points with very good signal to noise ratio the estimate is not impressive. The uncertainty in the response is also quite large as shown by the 1 standard deviation values of response. The reason is that the low pass input has poor excitation.

```
plot(t,y0,t,y13,t,y13+y13sd,'r:',t,y13-y13sd,'r:')
xlabel('Time (seconds)'), ylabel('Amplitude'), title('Impulse Response')
legend('True system','13:th order FIR model','Bounds')
```

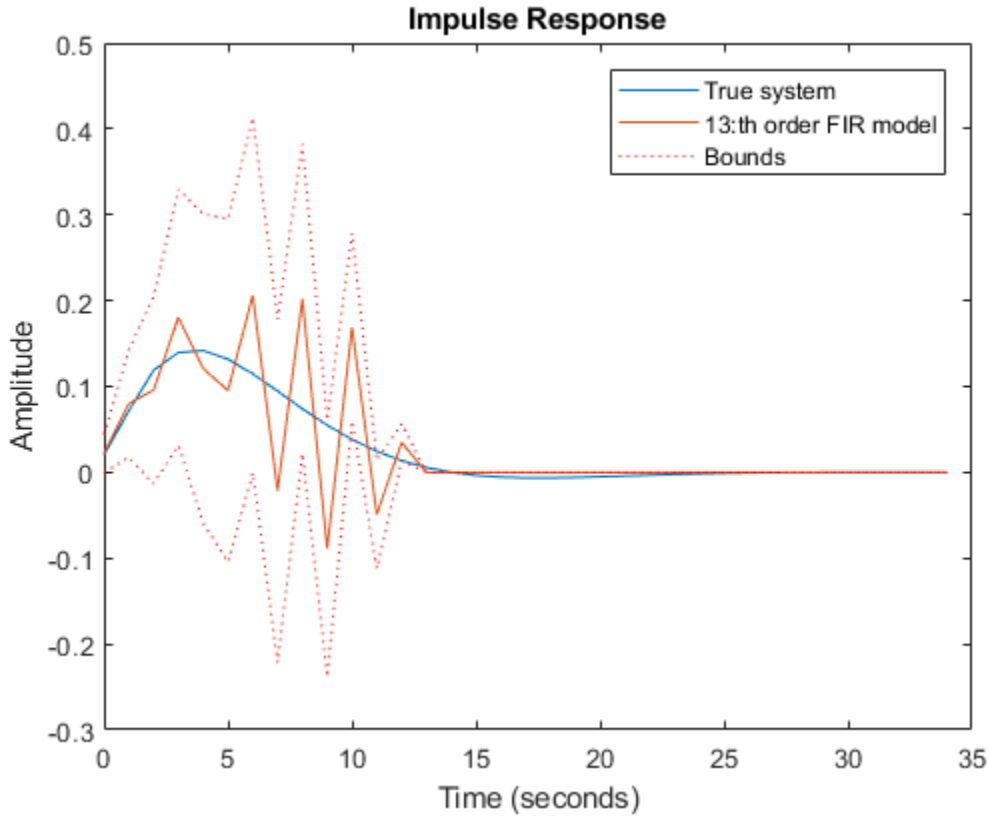


Figure 4: Estimated response with confidence bounds corresponding to 1 s.d.

Let us therefore try to reach a good bias-variance trade-off by ridge regression for a FIR model of order 50. Use `arxOptions` to configure the regularization constants. For this exercise we apply a simple penalty of $||\theta||^2$.

```
aopt = arxOptions;
aopt.Regularization.Lambda = 1;
m50r = arx(eData, [0 50 0], aopt);
```

The resulting estimate has an error norm of 0.1171 to the true impulse response and is shown in Figure 5 along with the confidence bounds.

```
[y50r,~,~,y50rsd] = impulse(m50r,t);
plot(t,y0,t,y50r,t,y50r+y50rsd,'r:',t,y50r-y50rsd,'r:')
xlabel('Time (seconds)'), ylabel('Amplitude'), title('Impulse Response')
legend('True system','50:th order regularized estimate')
```

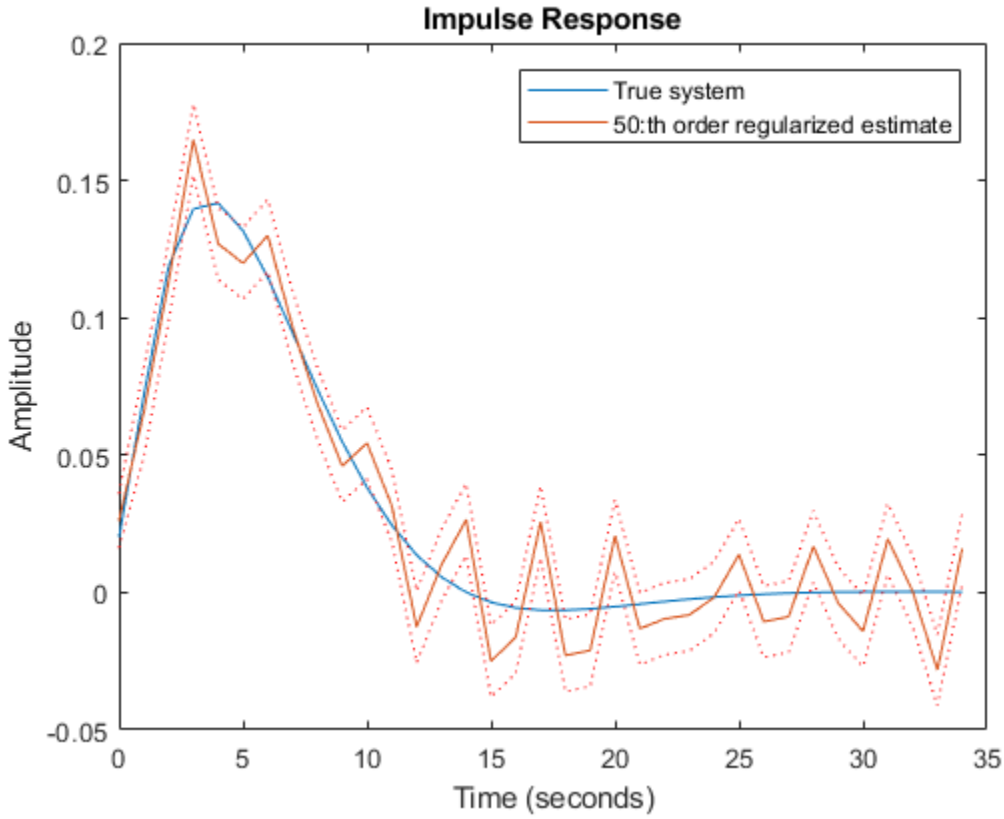


Figure 5: The true impulse response together with the ridge-regularized estimate for order `nb = 50`.

Clearly even this simple choice of regularization gives a much better bias-variance tradeoff, than selecting an optimal FIR order with no regularization.

Automatic Determination of Regularization Constants for FIR Models

We can do even better. By using the insight that the true impulse response decays to zero and is smooth, we can tailor the choice of R, λ to the data. This is achieved by the `arxRegul` function.

```
[L,R] = arxRegul(eData,[0 50 0],arxRegulOptions('RegulKernel','TC'));
aopt.Regularization.Lambda = L;
aopt.Regularization.R = R;
mrtc = arx(eData, [0 50 0], aopt);
[ytc,~,~,ytcsd] = impulse(mrtc,t);
```

`arxRegul` uses `fmincon` from Optimization Toolbox™ to compute the hyper-parameters associated with the regularization kernel ("TC" here). If Optimization Toolbox is not available, a simple Gauss-Newton search scheme is used instead; use the "Advanced.SearchMethod" option of `arxRegulOptions` to choose the search method explicitly. The estimated hyper-parameters are then used to derive the values of R and λ.

Using the estimated values of R and λ in ARX leads to an error norm of 0.0461 and the response is shown in Figure 6. This kind of tuned regularization is what is achieved also by the `impulseest` command. As the figure shows, the fit to the impulse response as well as the variance is greatly reduced as compared to the unregularized estimates. The price is a bias in the response estimate, which seems to be insignificant for this example.

```
plot(t,y0,t,ytc,t,ytc+ytcsd,'r:',t,ytc-ytcsd,'r:')
xlabel('Time (seconds)'), ylabel('Amplitude'), title('Impulse Response')
legend('True system','50:th order tuned regularized estimate')
```

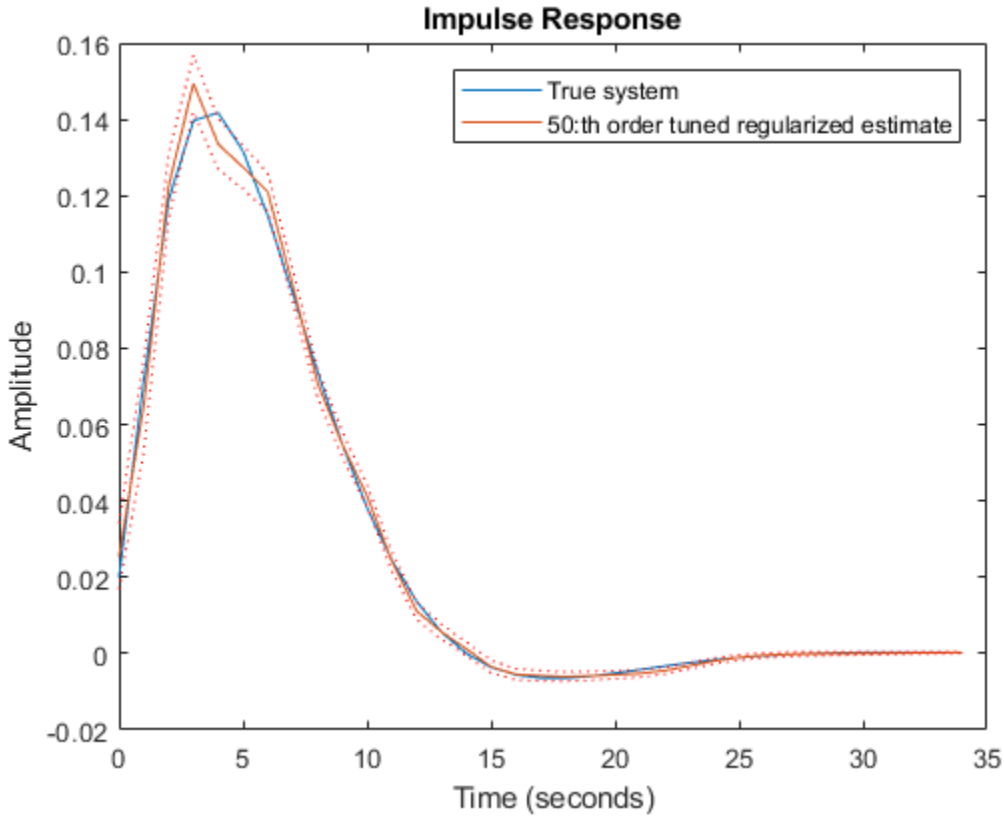


Figure 6: The true impulse response together with the tuned regularized estimate for order `nb = 50`.

Using Regularized ARX-models for Estimating State-Space Models

Consider a system `m0`, which is a 30:th order linear system with colored measurement noise:

$$y(t) = G(q)u(t) + H(q)e(t)$$

where `G(q)` is the input-to-output transfer function and `H(q)` is the disturbance transfer function. This system is stored in the regularizationExampleData.mat data file. The impulse responses of `G(q)` and `H(q)` are shown in Figure 7.

```
load regularizationExampleData.mat m0
m0H = noise2meas(m0); % the extracted noise component of the model
[yG,t] = impulse(m0);
yH = impulse(m0H,t);

clf
subplot(211)
plot(t, yG)
title('Impulse Response of G(q)'), ylabel('Amplitude')

subplot(212)
plot(t, yH)
title('Impulse Response of H(q)'), ylabel('Amplitude')
xlabel('Time (seconds)')
```

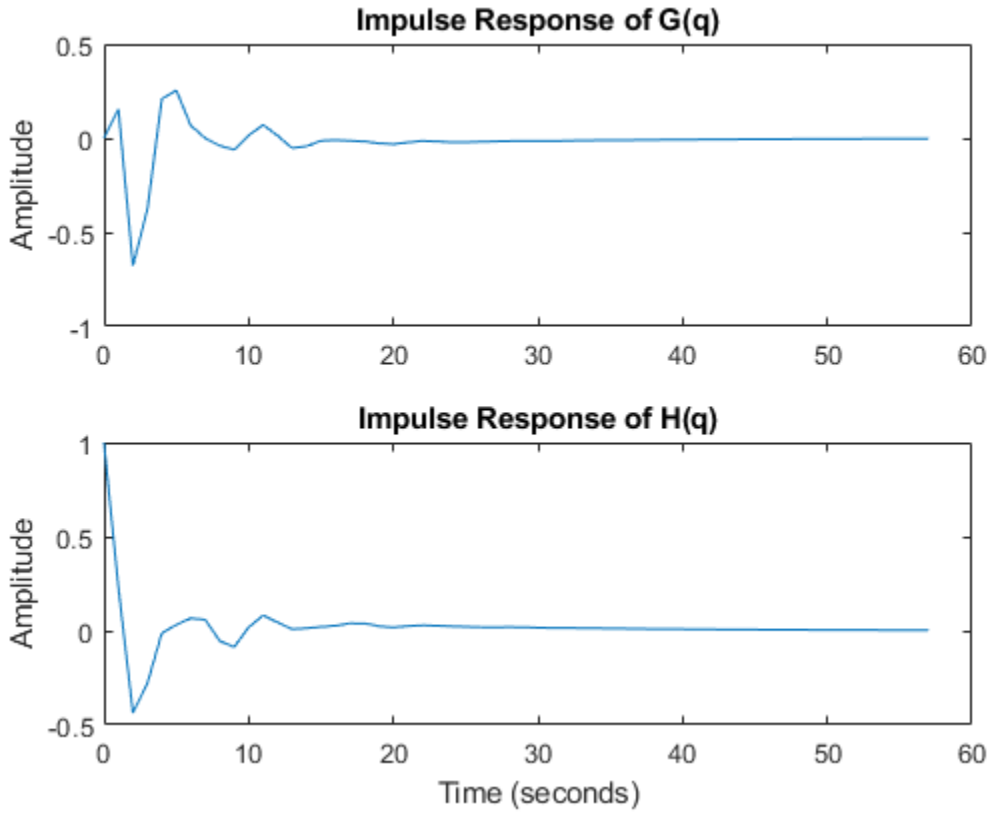


Figure 7: The impulse responses of `G(q)` (top) and `H(q)` (bottom).

We have collected 210 data points by simulating `m0` with a white noise input `u` with variance 1, and a noise level `e` with variance 0.1. This data is saved in regularizationExampleData.mat and is plotted below.

```
load regularizationExampleData.mat m0simdata
clf
plot(m0simdata)
```

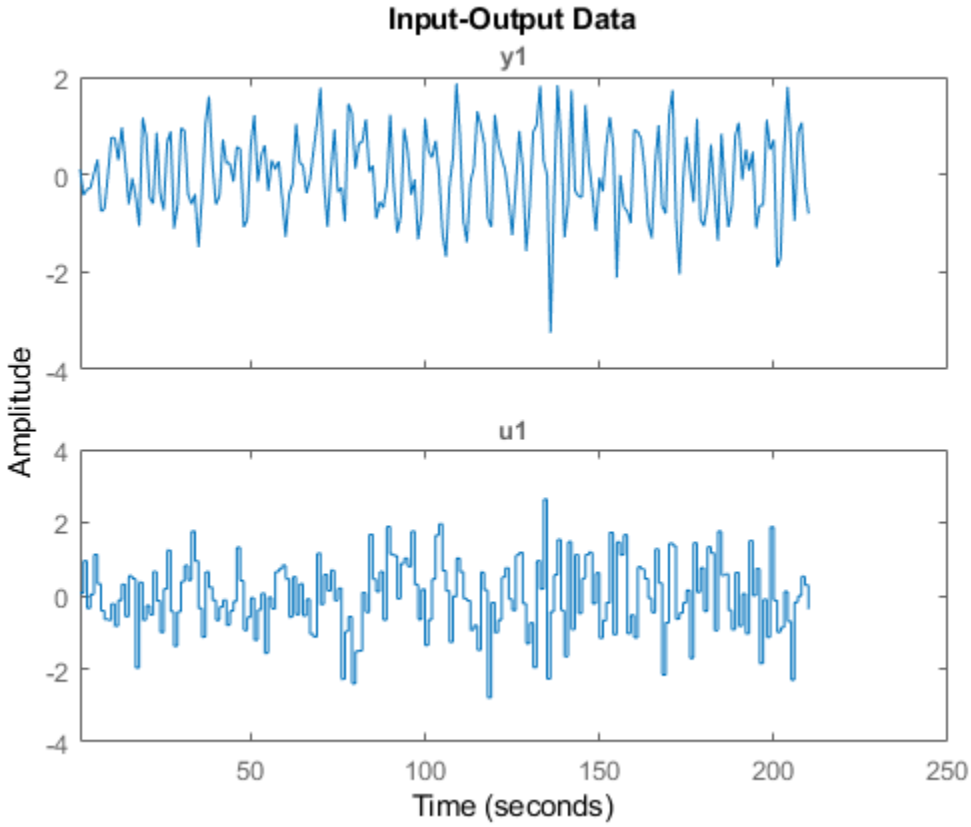


Figure 8: The data to be used for estimation.

To estimate the impulse responses of `m0` from these data, we can naturally employ state-space models in the innovations form (or equivalently ARMAX models) and compute the

impulse response using the `impulse` command as before. For computing the state-space model, we can use a syntax such as:

```
mk = ssest(m0simdata, k, 'Ts', 1);
```

The catch is to determine a good order `k`. There are two commonly used methods:

- *Cross validation CV*: Estimate `mk` for `k = 1,...,maxo` using the first half of the data `ze = m0simdata(1:150)` and evaluate the fit to the second half of the data `zv = m0simdata(151:end)` using the `compare` command: `[~,fitk] = compare(zv, mk, compareOptions('InitialCondition', 'z'))`. Determine the order `k` that maximizes the fit. Then reestimate the model using the whole data record.
- *Use the Akaike criterion AIC*: Estimate models for orders `k = 1,...,maxo` using the whole data set, and then pick that model that minimizes `aic(mk)`.

Applying these techniques to the data with a maximal order `maxo = 30` shows that CV picks `k = 15` and AIC picks `k = 3`.

The "Oracle" test: In addition to the CV and AIC tests, one can also check for what order `k` the fit between the true impulse response of `G(q)` (or `H(q)`) and the estimated model is maximized. This of course requires knowledge of the true system `m0` which is impractical. However, if we do carry on this comparison for our example where `m0` is known, we find that `k = 12` gives the best fit of estimated model's impulse response to that of `m0` (=|G(q)|). Similarly, we find that `k = 3` gives the best fit of estimated model's noise component's impulse response to that of the noise component of `m0` (=|H(q)|). The Oracle test sets a reference point for comparison of the quality of models generated by using various orders and regularization parameters.

Let us compare the impulse responses computed for various order selection criteria:

```
m3 = ssest(m0simdata, 3, 'Ts', 1);
m12 = ssest(m0simdata, 12, 'Ts', 1);
m15 = ssest(m0simdata, 15, 'Ts', 1);

y3 = impulse(m3, t);
y12 = impulse(m12, t);
y15 = impulse(m15, t);

plot(t,yG, t,y12, t,y15, t,y3)
xlabel('Time (seconds)'), ylabel('Amplitude'), title('Impulse Response')
legend('True G(q)',...
```

```
    sprintf('Oracle choice: %2.4g%%',100*goodnessOfFit(y12,yG,'NRMSE')),...
    sprintf('CV choice: %2.4g%%',100*goodnessOfFit(y15,yG,'NRMSE')),...
    sprintf('AIC choice: %2.4g%%',100*goodnessOfFit(y3,yG,'NRMSE')))
```

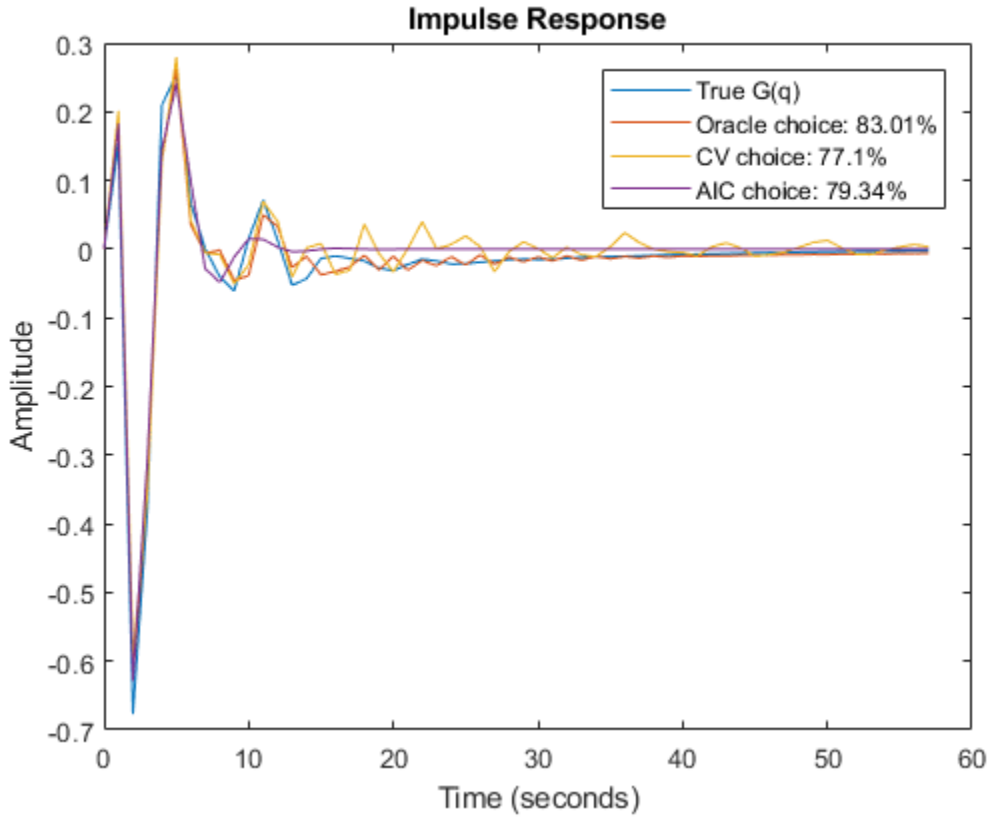


Figure 9: The true impulse response of `G(q)` compared to estimated models of various orders.

```
yH3 = impulse(noise2meas(m3), t);
yH15 = impulse(noise2meas(m15), t);

plot(t,yH, t,yH3, t,yH15, t,yH3)
xlabel('Time (seconds)'), ylabel('Amplitude'), title('Impulse Response')
legend('True H(q)',...
    sprintf('Oracle choice: %2.4g%%',100*goodnessOfFit(yH3,yH,'NRMSE')),...
```

```
sprintf('CV choice: %2.4g%%',100*goodnessOfFit(yH15,yH,'NRMSE')),...
sprintf('AIC choice: %2.4g%%',100*goodnessOfFit(yH3,yH,'NRMSE')))
```

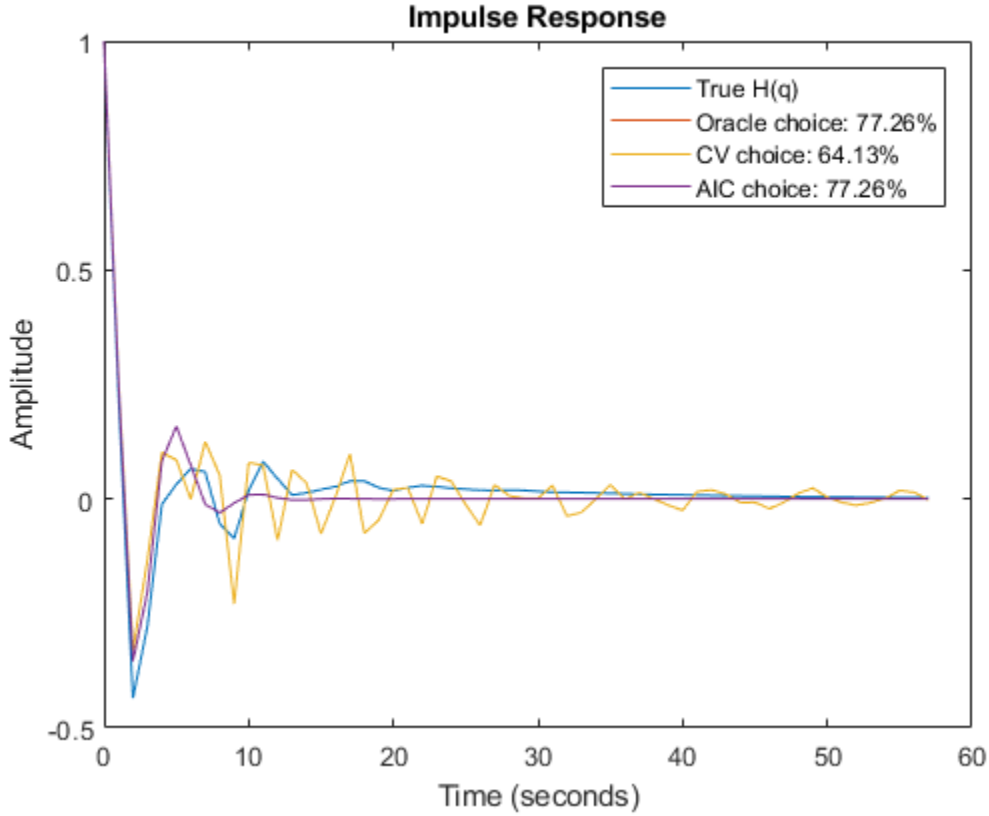


Figure 10: The true impulse response of `H(q)` compared to estimated noise models of various orders.

We see that a fit as good as 83% is possible to achieve for `G(q)` among the state-space models, but the order selection procedure may not find that best order.

We then turn to what can be obtained with regularization. We estimate a rather high order, regularized ARX-model by doing:

```
aopt = arxOptions;
[Lambda, R] = arxRegul(m0simdata, [5 60 0], arxRegulOptions('RegulKernel','TC'));
```

```
aopt.Regularization.R = R;
aopt.Regularization.Lambda = Lambda;
mr = arx(m0simdata, [5 60 0], aopt);
nmr = noise2meas(mr);
ymr = impulse(mr, t);
yHmr = impulse(nmr, t);
fprintf('Goodness of fit for ARX model is: %2.4g%%\n',100*goodnessOfFit(ymr,yG,'NRMSE')
fprintf('Goodness of fit for noise component of ARX model is: %2.4g%%\n',100*goodnessOf
```

```
Goodness of fit for ARX model is: 83.12%
Goodness of fit for noise component of ARX model is: 78.71%
```

It turns out that this regularized ARX model shows a fit to the true `G(q)` that is even better than the Oracle choice. The fit to `H(q)` is more than 80% which also is better that the Oracle choice of order for best noise model. It could be argued that `mr` is a high order (60 states) model, and it is unfair to compare it with lower order state space models. But this high order model can be reduced to, say, order 7 by using the `balred` command (requires Control System Toolbox™):

```
mred7 = balred(idss(mr),7);
nmred7 = noise2meas(mred7);
y7mr = impulse(mred7, t);
y7Hmr = impulse(nmred7, t);
```

Figures 11 and 12 show how the regularized and reduced order regularized models compare with the Oracle choice of state-space order for `ssest` without any loss of accuracy.

```
plot(t,yG, t,y12, t,ymr, t,y7mr)
xlabel('Time (seconds)'), ylabel('Amplitude'), title('Impulse Response')
legend('True G(q)',...
   sprintf('Oracle choice: %2.4g%%',100*goodnessOfFit(y12,yG,'NRMSE')),...
   sprintf('High order regularized: %2.4g%%',100*goodnessOfFit(ymr,yG,'NRMSE')),...
   sprintf('Reduced order: %2.4g%%',100*goodnessOfFit(y7mr,yG,'NRMSE')))
```

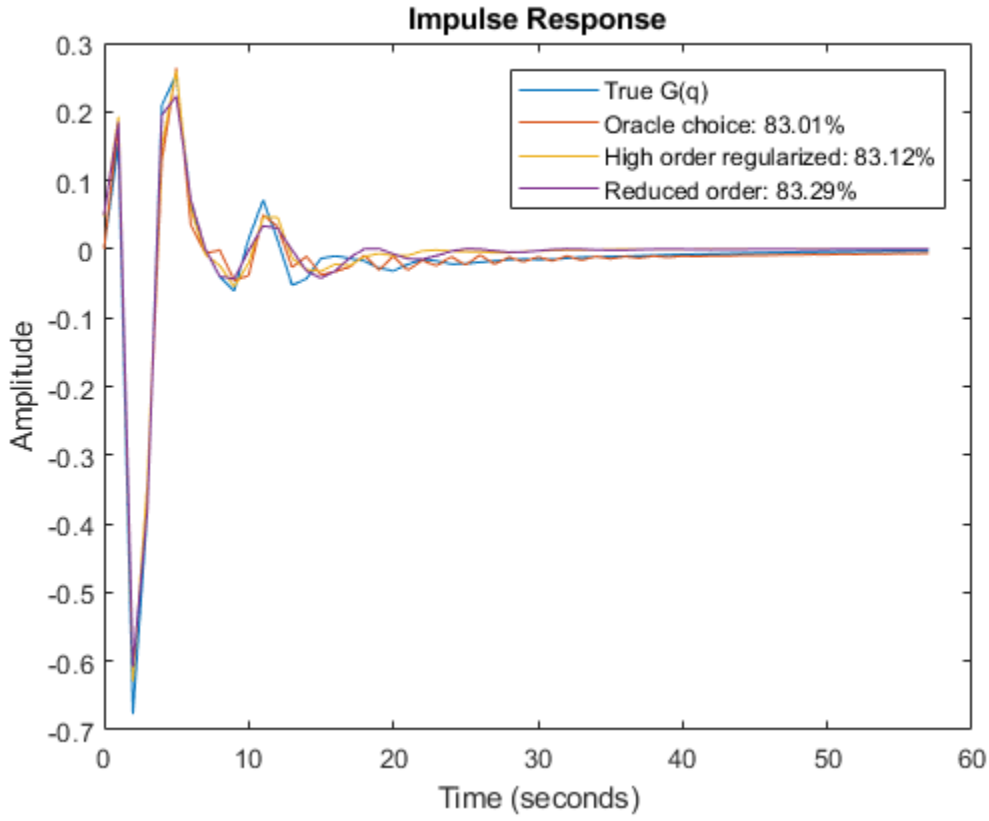


Figure 11: The regularized models compared to the Oracle choice for `G(q)`.

```
plot(t,yH, t,yH3, t,yHmr, t,y7Hmr)
xlabel('Time (seconds)'), ylabel('Amplitude'), title('Impulse Response')
legend('True H(q)',...
   sprintf('Oracle choice: %2.4g%%',100*goodnessOfFit(yH3,yH,'NRMSE')),...
   sprintf('High order regularized: %2.4g%%',100*goodnessOfFit(yHmr,yH,'NRMSE')),...
   sprintf('Reduced order: %2.4g%%',100*goodnessOfFit(y7Hmr,yH,'NRMSE')))
```

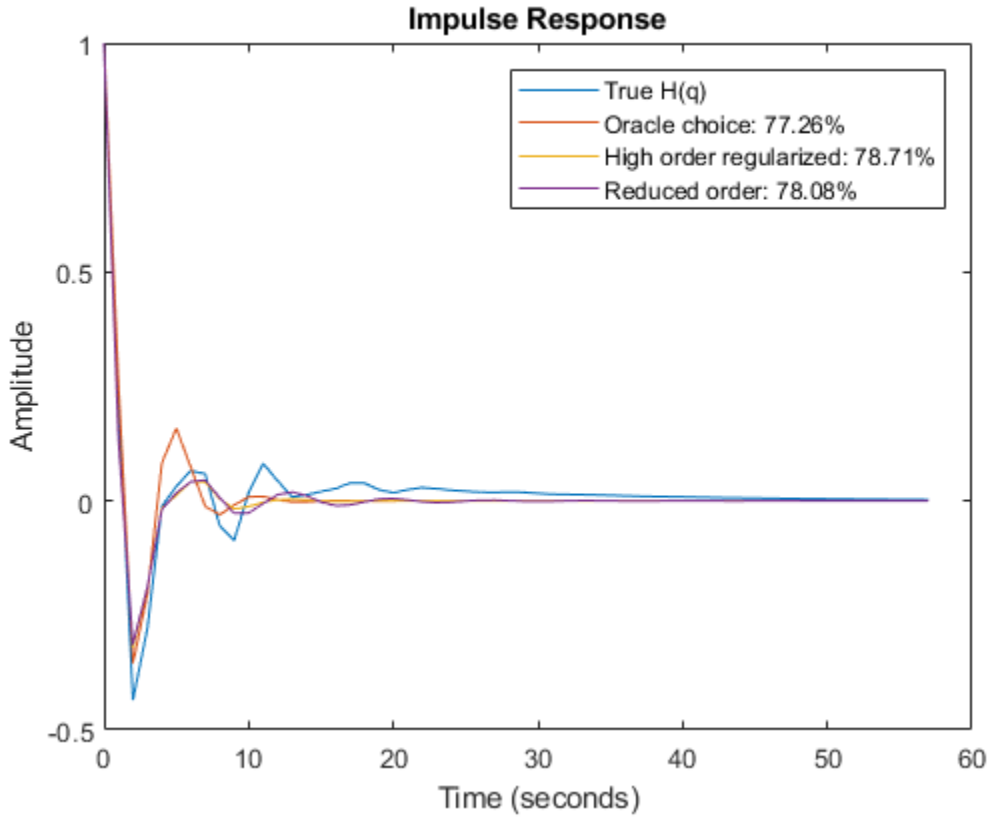


Figure 12: The regularized models compared to the Oracle choice for `H(q)`.

A natural question to ask is whether the choice of orders in the ARX model is as sensitive a decision as the state space model order in `ssest`. Simple test, using e.g. `arx(z,[10 50 0], aopt)`, shows only minor changes in the fit of `G(q)`.

State Space Model Estimation by Regularized Reduction Technique

The above steps of estimating a high-order ARX model, followed by a conversion to state-space and reduction to the desired order can be automated using the `ssregest` command. `ssregest` greatly simplifies this procedure while also facilitating other useful options such as search for optimal order and fine tuning of model structure by

specification of feedthrough and delay values. Here we simply reestimate the reduced model similar to `mred7` using `ssregest`:

```
opt = ssregestOptions('ARXOrder',[5 60 0]);
mred7_direct = ssregest(m0simdata, 7, 'Feedthrough', true, opt);
compare(m0simdata, mred7, mred7_direct)
```

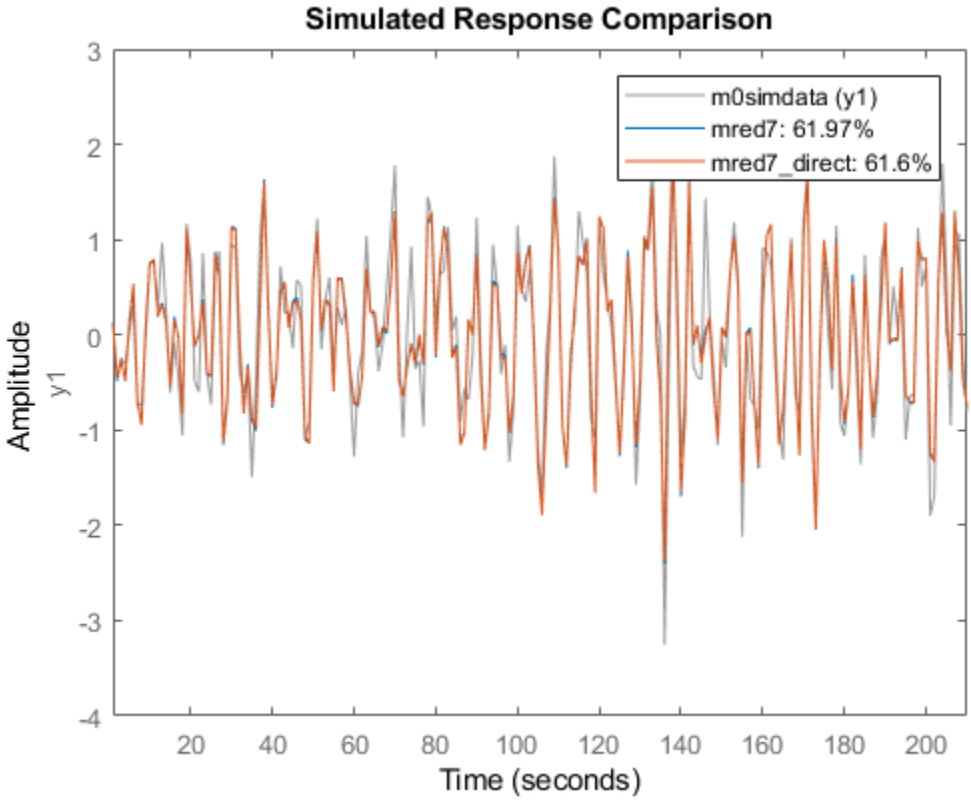


Figure 13: Comparing responses of state space models to estimation data.

```
h = impulseplot(mred7, mred7_direct, 40);
showConfidence(h,1) %  1 s.d. "zero interval"
hold on
s = stem(t,yG,'r');
```

```
s.DisplayName = 'True G(q)';
legend('show')
```

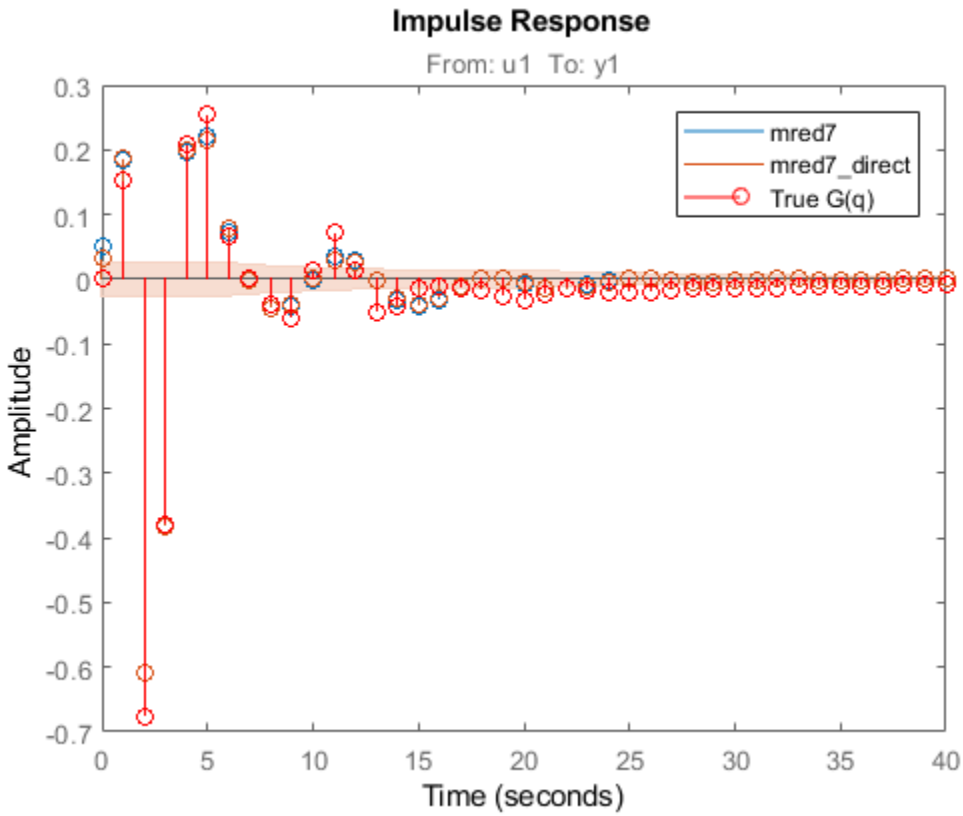


Figure 14: Comparing impulse responses of state space models.

In Figure 14, the confidence bound is only shown for the model `mred7_direct` since it was not calculated for the model `mred7`. You can use the `translatecov` command for generating confidence bounds for arbitrary transformations (here `balred`) of identified models. Note also that the `ssregest` command does not require you to provide the "ARXOrder" option value. It makes an automatic selection based on data length when no value is explicitly set.

Basic Bias - Variance Tradeoff in Grey Box Models

We shall discuss here grey box estimation which is a typical case where prior information meets information in observed data. It will be good to obtain a well balanced tradeoff between these information sources, and regularization is a prime tool for that.

Consider a DC motor (see e.g., `iddemo7`) with static gain `G` to angular velocity and time constant τ:

$$G(s) = \frac{G}{s(1+s\tau)}$$

In state-space form we have:

$$\dot{x_1} = x_2$$
$$\dot{x_2} = -1/\tau \cdot x_2$$
$$y = x + e$$

where $x = [x_1; x_2]$ is the state vector composed of the angle x_1 and the velocity x_2. We observe both states in noise as suggested by the output equation.

From prior knowledge and experience we think that G is about 4 and τ is about 1. We collect in `motorData` 400 data points from the system, with a substantial amount of noise (standard deviation of `e` is 50 in each component. We also save noise-free simulation data for the same model for comparison purposes. The data is shown in Figure 15.

```
load regularizationExampleData.mat motorData motorData_NoiseFree
t = motorData.SamplingInstants;
subplot(311)
plot(t,[motorData_NoiseFree.y(:,1),motorData.y(:,1)])
ylabel('Output 1')
subplot(312)
plot(t,[motorData_NoiseFree.y(:,2),motorData.y(:,2)])
ylabel('Output 2')
subplot(313)
plot(t,motorData_NoiseFree.u) % input is the same for both datasets
ylabel('Input')
xlabel('Time (seconds)')
```

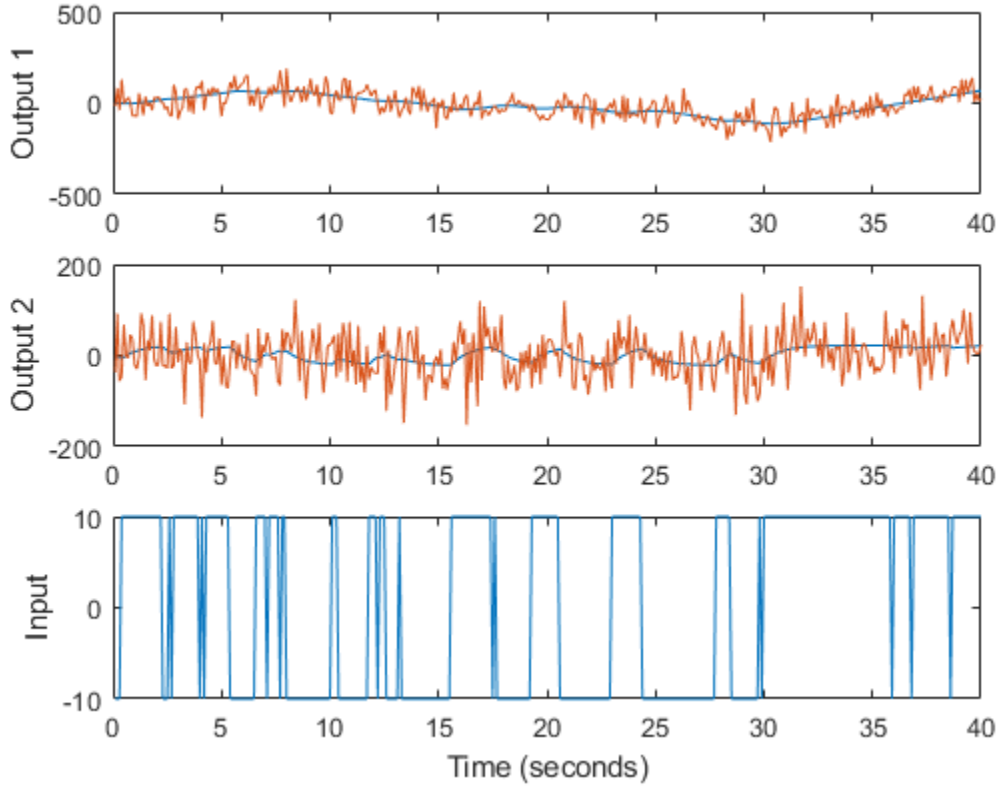


Figure 15: The noisy data to be used for grey box estimation superimposed over noise-free simulation data to be used for qualifications. From top to bottom: Angle, Angular Velocity, Input voltage.

The true parameter values in this simulation are `G` = 2.2 and τ = 0.8. To estimate the model we create an `idgrey` model file `DCMotorODE.m`.

```
type('DCMotorODE')
```

```
function [A,B,C,D] = DCMotorODE(G,Tau,Ts)
%DCMOTORODE ODE file representing the dynamics of a DC motor parameterized
%by gain G and time constant Tau.
%
```

```
%   [A,B,C,D,K,X0] = DCMOTORODE(G,Tau,Ts) returns the state space matrices
%   of the DC-motor with time-constant Tau and static gain G. The sample
%   time is Ts.
%
%   This file returns continuous-time representation if input argument Ts
%   is zero. If Ts>0, a discrete-time representation is returned.
%
% See also IDGREY, GREYEST.

%   Copyright 2013 The MathWorks, Inc.

A = [0 1;0 -1/Tau];
B = [0; G/Tau];
C = eye(2);
D = [0;0];
if Ts>0 % Sample the model with sample time Ts
   s = expm([[A B]*Ts; zeros(1,3)]);
   A = s(1:2,1:2);
   B = s(1:2,3);
end
```

An `idgrey` object is then created as:

```
mi = idgrey(@DCMotorODE,{'G', 4; 'Tau', 1},'cd',{}, 0);
```

where we have inserted the guessed parameter value as initial values. This model is adjusted to the information in observed data by using the `greyest` command:

```
m = greyest(motorData, mi)
```

```
m =
  Continuous-time linear grey box model defined by @DCMotorODE function:
      dx/dt = A x(t) + B u(t) + K e(t)
       y(t) = C x(t) + D u(t) + e(t)

  A =
          x1      x2
   x1      0       1
   x2      0  -1.741

  B =
         u1
   x1     0
   x2  3.721
```

```
  C =
       x1  x2
   y1   1   0
   y2   0   1

  D =
       u1
   y1   0
   y2   0

  K =
       y1  y2
   x1   0   0
   x2   0   0

  Model parameters:
   G = 2.138
   Tau = 0.5745

Parameterization:
   ODE Function: @DCMotorODE
   (parameterizes both continuous- and discrete-time equations)
   Disturbance component: none
   Initial state: 'auto'
   Number of free coefficients: 2
   Use "getpvec", "getcov" for parameters and their uncertainties.

Status:
Estimated using GREYEST on time domain data "motorData".
Fit to estimation data: [29.46;4.167]%
FPE: 6.074e+06, MSE: 4908
```

The model `m` has the parameters $\tau = 0.57$ and `G` = 2.14 and reproduces the data is shown in Figure 16.

```
copt = compareOptions('InitialCondition', 'z');
[ymi, fiti] = compare(motorData, mi, copt);
[ym, fit] = compare(motorData, m, copt);
t = motorData.SamplingInstants;
subplot(211)
plot(t, [motorData.y(:,1), ymi.y(:,1), ym.y(:,1)])
axis tight
ylabel('Output 1')
```

```
legend({'Measured output',...
   sprintf('Initial: %2.4g%%',fiti(1)),...
   sprintf('Estimated: %2.4g%%',fit(1))},...
   'Location','BestOutside')
subplot(212)
plot(t, [motorData.y(:,2), ymi.y(:,2), ym.y(:,2)])
ylabel('Output 2')
axis tight
legend({'Measured output',...
   sprintf('Initial: %2.4g%%',fiti(2)),...
   sprintf('Estimated: %2.4g%%',fit(2))},...
   'Location','BestOutside')
```

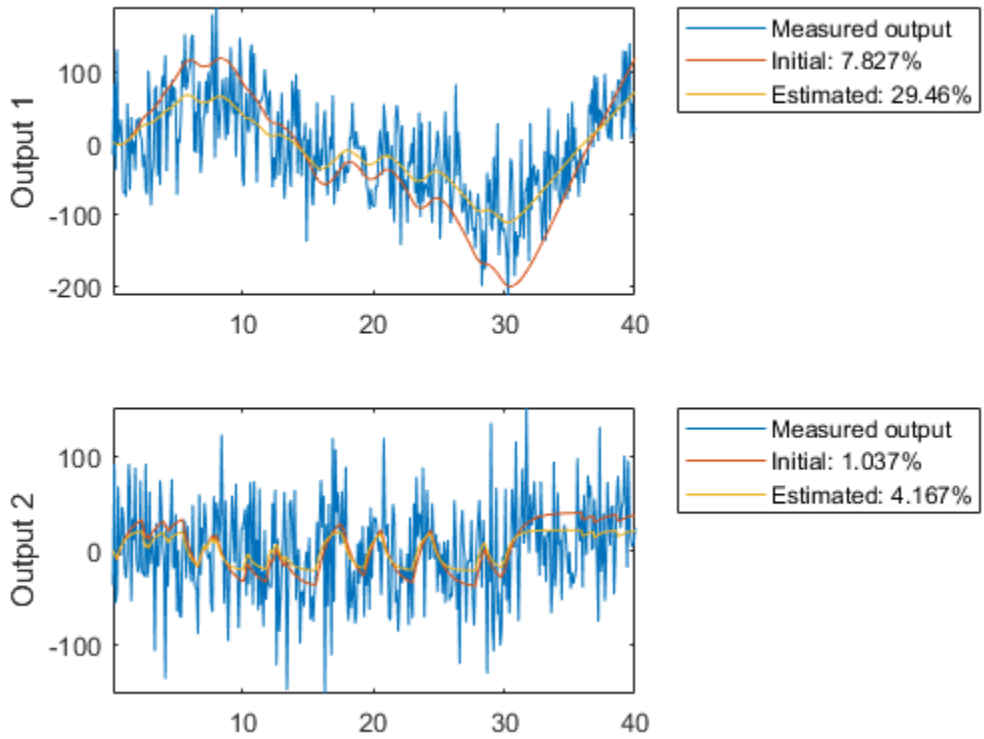


Figure 16: Measured output and model outputs for initial and estimated models.

In this simulated case we have also access to the noise-free data (`motorData_NoiseFree`) and depict the fit to the noise-free data in Figure 17.

```
[ymi, fiti] = compare(motorData_NoiseFree, mi, copt);
[ym, fit] = compare(motorData_NoiseFree, m, copt);
subplot(211)
plot(t, [motorData_NoiseFree.y(:,1), ymi.y(:,1), ym.y(:,1)])
axis tight
ylabel('Output 1')
legend({'Noise-free output',...
   sprintf('Initial: %2.4g%%',fiti(1)),...
   sprintf('Estimated: %2.4g%%',fit(1))},...
   'Location','BestOutside')
subplot(212)
plot(t, [motorData_NoiseFree.y(:,2), ymi.y(:,2), ym.y(:,2)])
ylabel('Output 2')
axis tight
legend({'Noise-free output',...
   sprintf('Initial: %2.4g%%',fiti(2)),...
   sprintf('Estimated: %2.4g%%',fit(2))},...
   'Location','BestOutside')
```

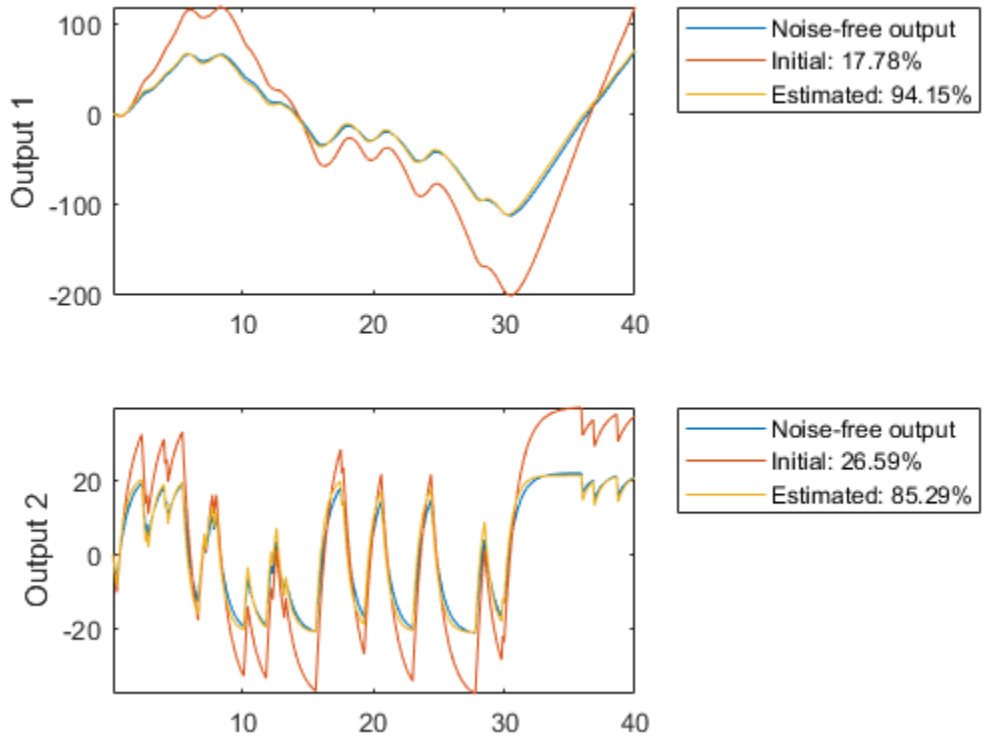


Figure 17: Noise-free output and model outputs for initial and estimated models.

We can look at the parameter estimates and see that the noisy data themselves give estimates that not quite agree with our prior physical information. To merge the data information with the prior information we use regularization:

```
opt = greyestOptions;
opt.Regularization.Lambda = 100;
opt.Regularization.R = [1, 1000]; % second parameter better known than first
opt.Regularization.Nominal = 'model';
mr = greyest(motorData, mi, opt)
```

```
mr =
```

```
  Continuous-time linear grey box model defined by @DCMotorODE function:
      dx/dt = A x(t) + B u(t) + K e(t)
       y(t) = C x(t) + D u(t) + e(t)

  A =
            x1      x2
   x1        0       1
   x2        0  -1.119

  B =
           u1
   x1       0
   x2   2.447

  C =
       x1  x2
   y1   1   0
   y2   0   1

  D =
       u1
   y1   0
   y2   0

  K =
       y1  y2
   x1   0   0
   x2   0   0

  Model parameters:
   G = 2.187
   Tau = 0.8938

Parameterization:
   ODE Function: @DCMotorODE
   (parameterizes both continuous- and discrete-time equations)
   Disturbance component: none
   Initial state: 'auto'
   Number of free coefficients: 2
   Use "getpvec", "getcov" for parameters and their uncertainties.

Status:
Estimated using GREYEST on time domain data "motorData".
```

```
Fit to estimation data: [29.34;3.848]%
FPE: 6.135e+06, MSE: 4933
```

We have here told the estimation process that we have some confidence in the initial parameter values, and believe more in our guess of T than in our guess of `G`. The resulting regularized estimate `mr` considers this information together with the information in measured data. They are weighed together with the help of `Lambda` and `R`. In Figure 18 it is shown how the resulting model can reproduce the output. Clearly, the regularized model does a better job than both the initial model (to which the parameters are "attracted") and the unregularized model.

```
[ymr, fitr] = compare(motorData_NoiseFree, mr, copt);
subplot(211)
plot(t, [motorData_NoiseFree.y(:,1), ymi.y(:,1), ym.y(:,1), ymr.y(:,1)])
axis tight
ylabel('Output 1')
legend({'Noise-free output',...
   sprintf('Initial: %2.4g%%',fiti(1)),...
   sprintf('Estimated: %2.4g%%',fit(1)),...
   sprintf('Regularized: %2.4g%%',fitr(1))},...
   'Location','BestOutside')
subplot(212)
plot(t, [motorData_NoiseFree.y(:,2), ymi.y(:,2), ym.y(:,2), ymr.y(:,2)])
ylabel('Output 2')
axis tight
legend({'Noise-free output',...
   sprintf('Initial: %2.4g%%',fiti(2)),...
   sprintf('Estimated: %2.4g%%',fit(2)),...
   sprintf('Regularized: %2.4g%%',fitr(2))},...
   'Location','BestOutside')
```

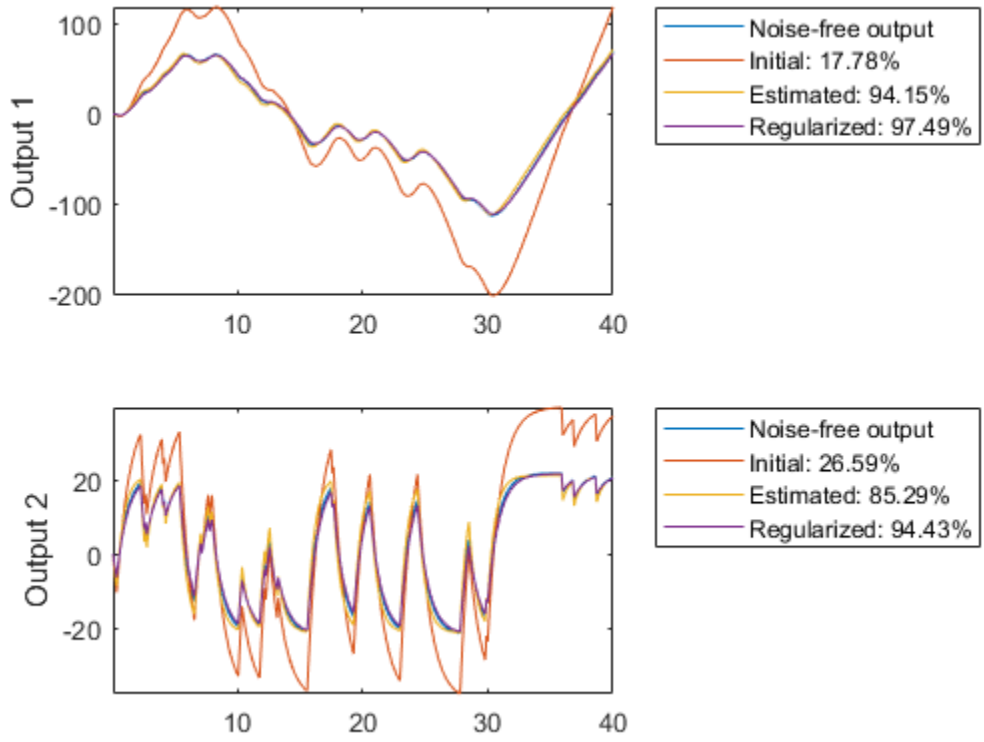


Figure 18: Noise-Free measured output and model outputs for initial, estimated and regularized models.

The regularized estimation also has reduced parameter variance as compared to the unregularized estimates. This is shown by tighter confidence bounds on the Bode plot of `mr` compare to that of `m`:

```
clf
showConfidence(bodeplot(m,mr,logspace(-1,1.4,100)),3) % 3 s.d. region
legend('show')
```

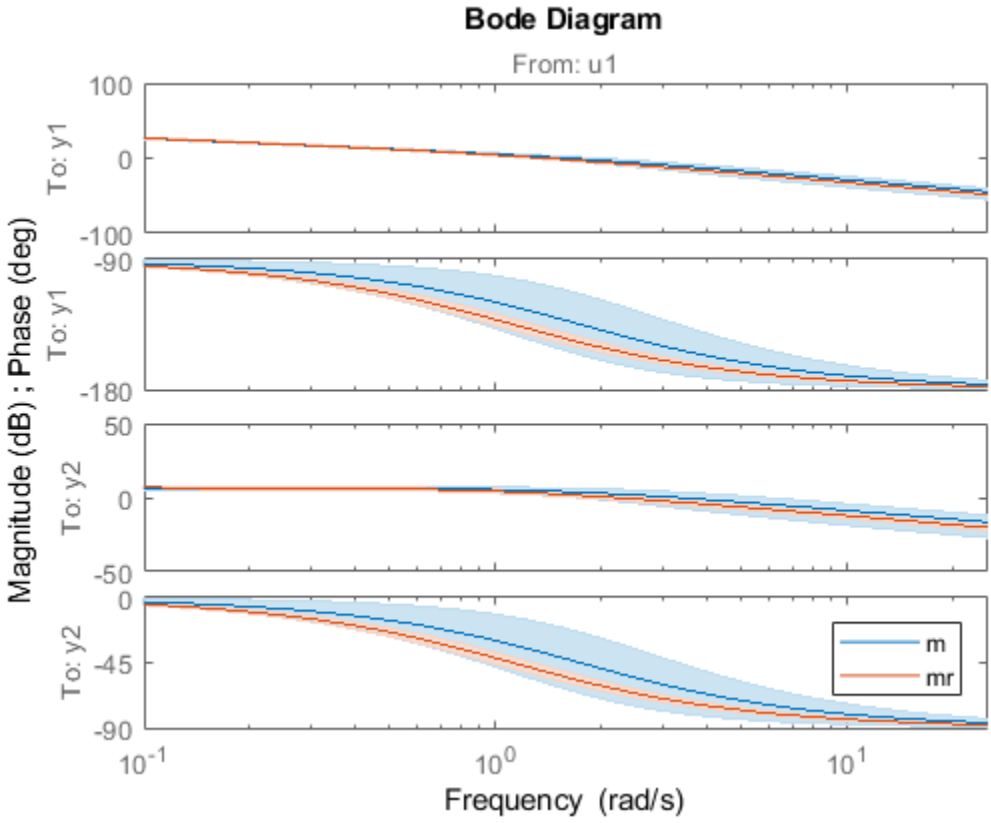


Figure 19: Bode plot of `m` and `mr` with confidence bounds

This was an illustration of how the merging prior and measurement information works. In practice we need a procedure to tune the size of `Lambda` to the existing information sources. A commonly used method is to use *cross validation*. That is:

- Split the data into two parts - the estimation and the validation data
- Compute the regularized model using the estimation data for various values of `Lambda`
- Evaluate how well these models can reproduce the validation data: tabulate NRMSE fit values delivered by the `compare` command or the `goodnessOfFit` command.
- Pick that `Lambda` which gives the model with the best fit to the validation data.

Use of Regularization to Robustify Large Nonlinear Models

Another use of regularization is to numerically stabilize the estimation of large (often nonlinear) models. We have given a data record `nldata` that has nonlinear dynamics. We try nonlinear ARX-model of neural network character, with more and more neurons:

```
load regularizationExampleData.mat nldata
opt = nlarxOptions('SearchMethod','lm');
m10 = nlarx(nldata, [1 2 1], sigmoidnet('NumberOfUnits',10),opt);
m20 = nlarx(nldata, [1 2 1], sigmoidnet('NumberOfUnits',20),opt);
m30 = nlarx(nldata, [1 2 1], sigmoidnet('NumberOfUnits',30),opt);

compare(nldata, m10, m20) % compare responses of m10, m20 to measured response
```

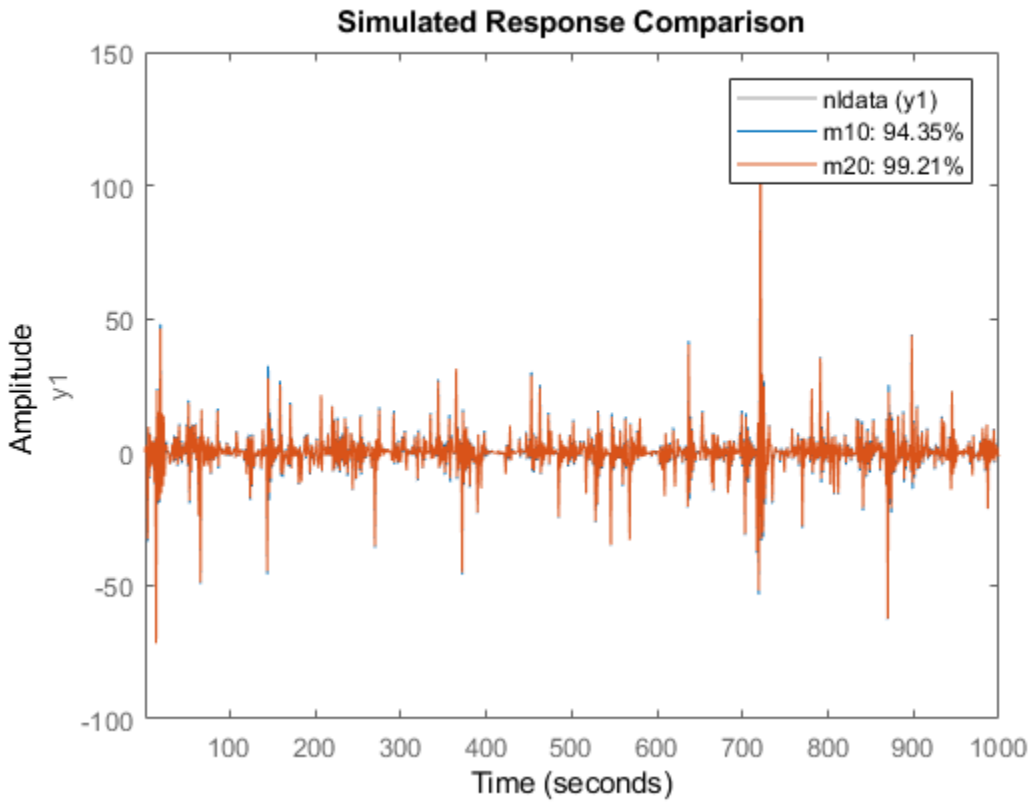


Figure 20: Comparison plot for models `m10` and `m20`.

```
fprintf('Number of parameters (m10, m20, m30): %s\n',...
   mat2str([nparams(m10),nparams(m20),nparams(m30)]))
compare(nldata, m30, m10, m20) % compare all three models
axis([1 800 -57 45])
```

```
Number of parameters (m10, m20, m30): [54 104 154]
```

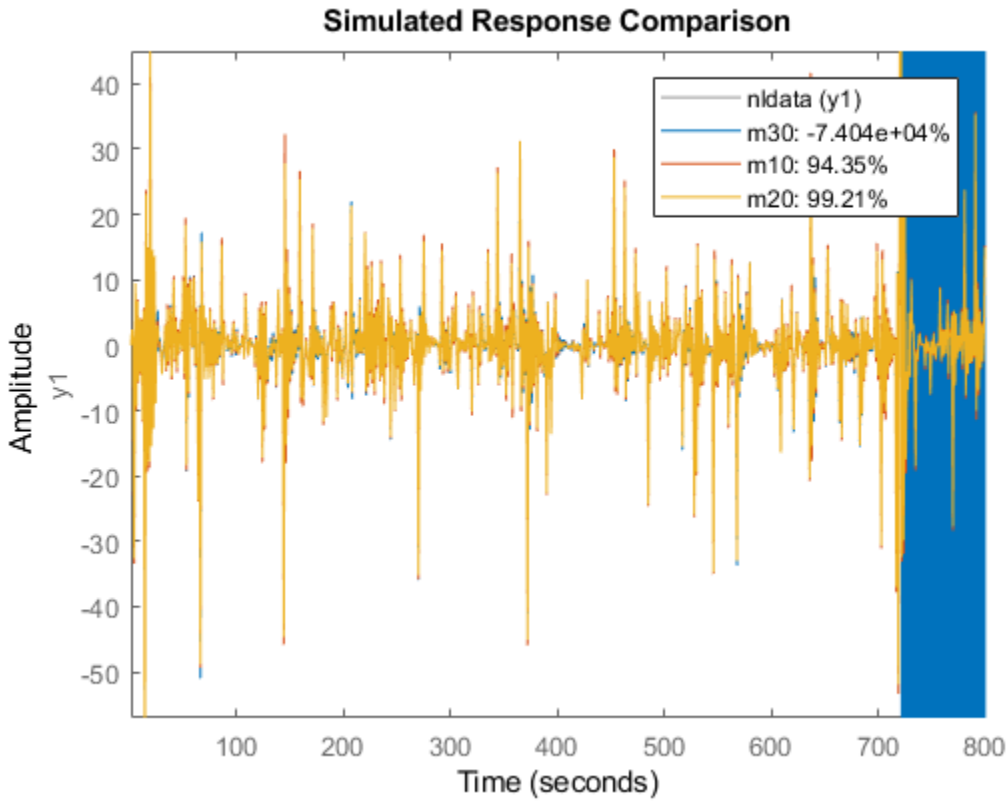


Figure 21: Comparison plot for models `m10`, `m20` and `m30`.

The first two models show good and improving fits. But when estimating the 154 parameters of `m30`, numerical problems seem to occur. We can then apply a small amount of regularization to get better conditioned matrices:

```
opt.Regularization.Lambda = 1e-8;
m30r = nlarx(nldata, [1 2 1], sigmoidnet('num',30), opt);
compare(nldata, m30r, m10, m20)
```

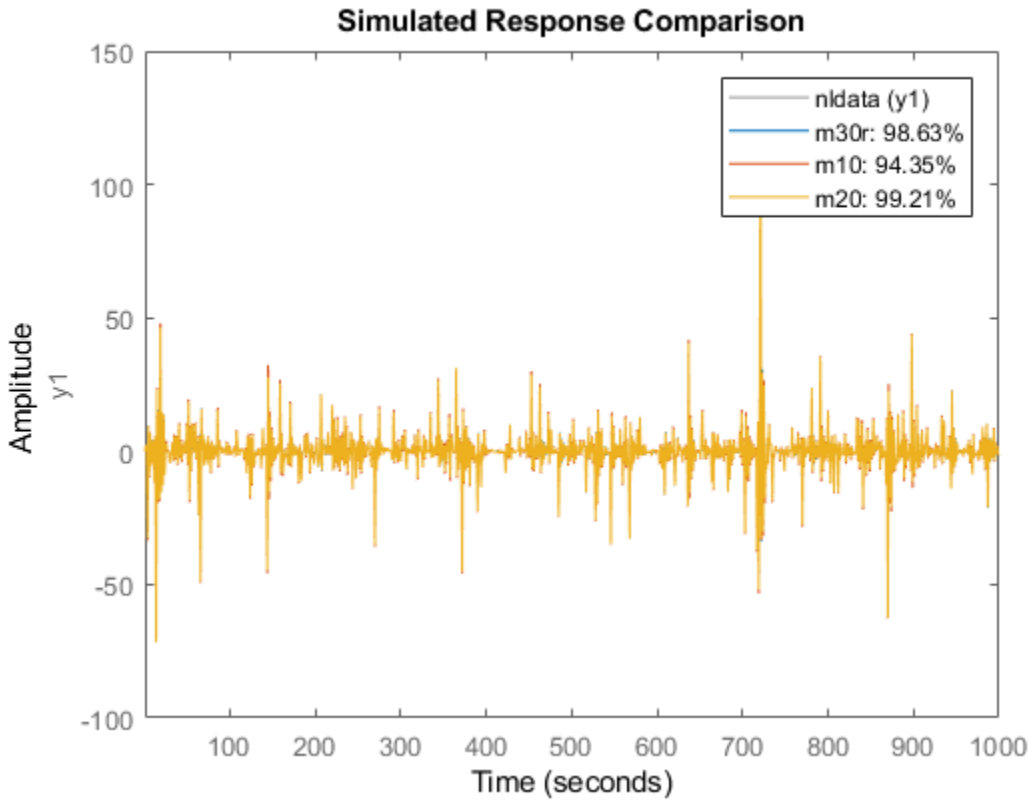


Figure 22: Comparison plot for models `m10`, `m20` and regularized model `m30r`.

The fit to estimation data has significantly improved for the model with 30 neurons. As discussed before, a systematic search for the `Lambda` value to use would require cross validation tests.

Conclusions

We discussed the benefit of regularization for estimation of FIR models, linear grey-box models and Nonlinear ARX models. Regularization can have significant impact on the

quality of the identified model provided the regularization constants `Lambda` and `R` are chosen appropriately. For ARX models, this can be done very easily using the `arxRegul` function. These automatic choices also feed into the dedicated state-space estimation algorithm `ssregest`.

For other types of estimations, you must rely on cross validation based search to determine `Lambda`. For structured models such as grey box models, `R` can be used to indicate the reliability of the corresponding initial value of the parameter. Then, using the `Nominal` regularization option, you can merge the prior knowledge of the parameter values with the information in the data.

Regularization options are available for all linear and nonlinear models including transfer functions and process models, state-space and polynomial models, Nonlinear ARX, Hammerstein-Wiener and linear/nonlinear grey box models.

Data Import and Processing

- "Supported Data" on page 2-3
- "Ways to Obtain Identification Data" on page 2-5
- "Ways to Prepare Data for System Identification" on page 2-6
- "Requirements on Data Sampling" on page 2-8
- "Representing Data in MATLAB Workspace" on page 2-9
- "Import Time-Domain Data into the App" on page 2-16
- "Import Frequency-Domain Data into the App" on page 2-19
- "Import Data Objects into the App" on page 2-25
- "Specifying the Data Sample Time" on page 2-28
- "Specify Estimation and Validation Data in the App" on page 2-30
- "Preprocess Data Using Quick Start" on page 2-32
- "Create Data Sets from a Subset of Signal Channels" on page 2-33
- "Create Multiexperiment Data Sets in the App" on page 2-35
- "Managing Data in the App" on page 2-42
- "Representing Time- and Frequency-Domain Data Using iddata Objects" on page 2-50
- "Create Multiexperiment Data at the Command Line" on page 2-61
- "Dealing with Multi-Experiment Data and Merging Models" on page 2-64
- "Managing iddata Objects" on page 2-79
- "Representing Frequency-Response Data Using idfrd Objects" on page 2-85
- "Is Your Data Ready for Modeling?" on page 2-92
- "How to Plot Data in the App" on page 2-93
- "How to Plot Data at the Command Line" on page 2-100
- "How to Analyze Data Using the advice Command" on page 2-102
- "Select Subsets of Data" on page 2-104
- "Handling Missing Data and Outliers" on page 2-108

- “Extract and Model Specific Data Segments” on page 2-111
- “Handling Offsets and Trends in Data” on page 2-114
- “How to Detrend Data Using the App” on page 2-117
- “How to Detrend Data at the Command Line” on page 2-118
- “Resampling Data” on page 2-120
- “Resampling Data Using the App” on page 2-125
- “Resampling Data at the Command Line” on page 2-127
- “Filtering Data” on page 2-129
- “How to Filter Data Using the App” on page 2-131
- “How to Filter Data at the Command Line” on page 2-134
- “Generate Data Using Simulation” on page 2-137
- “Manipulating Complex-Valued Data” on page 2-143

Supported Data

System Identification Toolbox software supports estimation of linear models from both time- and frequency-domain data. For nonlinear models, this toolbox supports only time-domain data. For more information, see “Supported Models for Time- and Frequency-Domain Data” on page 1-38.

The data can have single or multiple inputs and outputs, and can be either real or complex.

Your time-domain data should be sampled at discrete and uniformly spaced time instants to obtain an input sequence

$u=\{u(T),u(2T),...,u(NT)\}$

and a corresponding output sequence

$y=\{y(T),y(2T),...,y(NT)\}$

u(t) and *y(t)* are the values of the input and output signals at time *t*, respectively.

This toolbox supports modeling both single- or multiple-channel input-output data or time-series data.

Supported Data	Description
Time-domain I/O data	One or more input variables *u(t)* and one or more output variables *y(t)*, sampled as a function of time. Time-domain data can be either real or complex
Time-series data	Contains one or more outputs *y(t)* and no measured input. Can be time-domain or frequency-domain data.
Frequency-domain data	Fourier transform of the input and output time-domain signals. The data is the set of input and output signals in frequency domain; the frequency grid need not be uniform.

Supported Data	Description
Frequency-response data	Complex frequency-response values for a linear system characterized by its transfer function *G*, measurable directly using a spectrum analyzer. Also called *frequency function data*. Represented by `frd` or `idfrd` objects. The data sample time may be zero or nonzero. The frequency vector need not be uniformly spaced.

Note If your data is complex valued, see "Manipulating Complex-Valued Data" on page 2-143 for information about supported operations for complex data.

Ways to Obtain Identification Data

You can obtain identification data by:

- Measuring input and output signals from a physical system.

  Your data must capture the important system dynamics, such as dominant time constants. After measuring the signals, organize the data into variables, as described in "Representing Data in MATLAB Workspace" on page 2-9. Then, import it in the System Identification app or represent it as a data object for estimating models at the command line.
- Generating an input signal with desired characteristics, such as a random Gaussian or binary signal or a sinusoid, using `idinput`. Then, generate an output signal using this input to simulate a model with known coefficients. For more information, see "Generate Data Using Simulation" on page 2-137.

  Using input/output data thus generated helps you study the impact of input signal characteristics and noise on estimation.
- Logging signals from Simulink models.

  This technique is useful when you want to replace complex components in your model with identified models to speed up simulations or simplify control design tasks. For more information on how to log signals, see "Export Signal Data Using Signal Logging" (Simulink).

Ways to Prepare Data for System Identification

Before you can perform any task in this toolbox, your data must be in the MATLAB workspace. You can import the data from external data files or manually create data arrays at the command line. For more information about importing data, see “Representing Data in MATLAB Workspace” on page 2-9.

The following tasks help to prepare your data for identifying models from data:

Represent data for system identification

You can represent data in the format of this toolbox by doing one of the following:

- For working in the app, import data into the System Identification app.

  See “Represent Data”.
- For working at the command line, create an `iddata` or `idfrd` object.

  For time-domain or frequency-domain data, see “Representing Time- and Frequency-Domain Data Using iddata Objects” on page 2-50.

  For frequency-response data, see “Representing Frequency-Response Data Using idfrd Objects” on page 2-85.
- To simulate data with and without noise, see “Generate Data Using Simulation” on page 2-137.

Analyze data quality

You can analyze your data by doing either of the following:

- Plotting data to examine both time- and frequency-domain behavior.

  See “How to Plot Data in the App” on page 2-93 and “How to Plot Data at the Command Line” on page 2-100.
- Using the `advice` command to analyze the data for the presence of constant offsets and trends, delay, possible feedback, and signal excitation levels.

  See “How to Analyze Data Using the advice Command” on page 2-102.

Preprocess data

Review the data characteristics for any of the following features to determine if there is a need for preprocessing:

- Missing or faulty values (also known as *outliers*). For example, you might see gaps that indicate missing data, values that do not fit with the rest of the data, or noninformative values.

  See "Handling Missing Data and Outliers" on page 2-108.

- Offsets and drifts in signal levels (low-frequency disturbances).

  See "Handling Offsets and Trends in Data" on page 2-114 for information about subtracting means and linear trends, and "Filtering Data" on page 2-129 for information about filtering.

- High-frequency disturbances above the frequency interval of interest for the system dynamics.

  See "Resampling Data" on page 2-120 for information about decimating and interpolating values, and "Filtering Data" on page 2-129 for information about filtering.

Select a subset of your data

You can use data selection as a way to clean the data and exclude parts with noisy or missing information. You can also use data selection to create independent data sets for estimation and validation.

To learn more about selecting data, see "Select Subsets of Data" on page 2-104.

Combine data from multiple experiments

You can combine data from several experiments into a single data set. The model you estimate from a data set containing several experiments describes the average system that represents these experiments.

To learn more about creating multiple-experiment data sets, see "Create Multiexperiment Data Sets in the App" on page 2-35 or "Create Multiexperiment Data at the Command Line" on page 2-61.

Requirements on Data Sampling

A *sample time* is the time between successive data samples. It is sometimes also referred to as *sampling time* or *sample interval*.

The System Identification app only supports uniformly sampled data.

The System Identification Toolbox product provides limited support for nonuniformly sampled data. For more information about specifying uniform and nonuniform time vectors, see “Constructing an iddata Object for Time-Domain Data” on page 2-50.

Representing Data in MATLAB Workspace

In this section...
"Time-Domain Data Representation" on page 2-9
"Time-Series Data Representation" on page 2-10
"Frequency-Domain Data Representation" on page 2-10

Time-Domain Data Representation

Time-domain data consists of one or more input variables *u(t)* and one or more output variables *y(t)*, sampled as a function of time. If there is no input variable, see "Time-Series Data Representation" on page 2-10.

You must organize time-domain input/output data in the following format:

- For single-input/single-output (SISO) data, the sampled data values must be double column vectors.
- For multi-input/multi-output (MIMO) data with N_u inputs and N_y outputs, and N_s number of data samples (measurements):
  - The input data must be an N_s-by-N_u matrix
  - The output data must be an N_s-by-N_y matrix

To use time-domain data for identification, you must know the sample time. If you are working with uniformly sampled data, use the actual sample time from your experiment. Each data value is assigned a time instant, which is calculated from the start time and sample time. You can work with nonuniformly sampled data only at the command line by specifying a vector of time instants using the `SamplingInstants` property of `iddata`, as described in "Constructing an iddata Object for Time-Domain Data" on page 2-50.

For continuous-time models, you must also know the input intersample behavior, such as zero-order hold and first-order-hold.

For more information about importing data into MATLAB, see "Data Import and Export" (MATLAB).

After you have the variables in the MATLAB workspace, import them into the System Identification app or create a data object for working at the command line. For more

information, see “Import Time-Domain Data into the App” on page 2-16 and “Representing Time- and Frequency-Domain Data Using iddata Objects” on page 2-50.

Time-Series Data Representation

Time-series data is time-domain or frequency-domain data that consist of one or more outputs *y(t)* with no corresponding input. For more information on how to obtain identification data, see “Ways to Obtain Identification Data” on page 2-5.

You must organize time-series data in the following format:

- For single-input/single-output (SISO) data, the output data values must be a column vector.
- For data with N_y outputs, the output is an N_s-by-N_y matrix, where N_s is the number of output data samples (measurements).

To use time-series data for identification, you also need the sample time. If you are working with uniformly sampled data, use the actual sample time from your experiment. Each data value is assigned a sample time, which is calculated from the start time and the sample time. If you are working with nonuniformly sampled data at the command line, you can specify a vector of time instants using the `iddata SamplingInstants` property, as described in “Constructing an iddata Object for Time-Domain Data” on page 2-50. Note that model estimation cannot be performed using non-uniformly sampled data.

For more information about importing data into the MATLAB workspace, see “Data Import and Export” (MATLAB).

After you have the variables in the MATLAB workspace, import them into the System Identification app or create a data object for working at the command line. For more information, see “Import Time-Domain Data into the App” on page 2-16 and “Representing Time- and Frequency-Domain Data Using iddata Objects” on page 2-50.

For information about estimating time-series model parameters, see “Time Series Analysis”.

Frequency-Domain Data Representation

Frequency-domain data consists of either transformed input and output time-domain signals on page 2-11 or system frequency response on page 2-13 sampled as a function of the independent variable frequency.

- "Frequency-Domain Input/Output Signal Representation" on page 2-11
- "Frequency-Response Data Representation" on page 2-13

Frequency-Domain Input/Output Signal Representation

- "What Is Frequency-Domain Input/Output Signal?" on page 2-11
- "How to Represent Frequency-Domain Data in MATLAB" on page 2-12

What Is Frequency-Domain Input/Output Signal?

Frequency-domain data is the Fourier transform of the input and output time-domain signals. For continuous-time signals, the Fourier transform over the entire time axis is defined as follows:

$$Y(iw) = \int_{-\infty}^{\infty} y(t)e^{-iwt}dt$$

$$U(iw) = \int_{-\infty}^{\infty} u(t)e^{-iwt}dt$$

In the context of numerical computations, continuous equations are replaced by their discretized equivalents to handle discrete data values. For a discrete-time system with a sample time T, the frequency-domain output $Y(e^{iw})$ and input $U(e^{iw})$ is the time-discrete Fourier transform (TDFT):

$$Y(e^{iwT}) = T\sum_{k=1}^{N} y(kT)e^{-iwkT}$$

In this example, `k = 1,2,...,N`, where `N` is the number of samples in the sequence.

Note This form only discretizes the time. The frequency is continuous.

In practice, the Fourier transform cannot be handled for all continuous frequencies and you must specify a finite number of frequencies. The discrete Fourier transform (DFT) of time-domain data for N equally spaced frequencies between 0 and the sampling frequency $2\pi/N$ is:

$$Y(e^{iw_nT}) = \sum_{k=1}^{N} y(kT)e^{-iw_nkT}$$

$$w_n = \frac{2\pi n}{T} \quad n = 0,1,2,\ldots,N-1$$

The DFT is useful because it can be calculated very efficiently using the fast Fourier transform (FFT) method. Fourier transforms of the input and output data are complex numbers.

For more information on how to obtain identification data, see "Ways to Obtain Identification Data" on page 2-5.

How to Represent Frequency-Domain Data in MATLAB

You must organize frequency-domain data in the following format:

- Input and output
  - For single-input/single-output (SISO) data:
    - The input data must be a column vector containing the values $u\left(e^{i\omega kT}\right)$
    - The output data must be a column vector containing the values $y\left(e^{i\omega kT}\right)$

    $k=1, 2, \ldots, N_f$, where N_f is the number of frequencies.
  - For multi-input/multi-output data with N_u inputs, N_y outputs and N_f frequency measurements:
    - The input data must be an N_f-by-N_u matrix
    - The output data must be an N_f-by-N_y matrix
- Frequencies
  - Must be a column vector.

For more information about importing data into the MATLAB workspace, see "Data Import and Export" (MATLAB).

After you have the variables in the MATLAB workspace, import them into the System Identification app or create a data object for working at the command line. For more

information, see "Importing Frequency-Domain Input/Output Signals into the App" on page 2-19 and "Representing Time- and Frequency-Domain Data Using iddata Objects" on page 2-50.

Frequency-Response Data Representation

- "What Is Frequency-Response Data?" on page 2-13
- "How to Represent Frequency-Response Data in MATLAB" on page 2-14

What Is Frequency-Response Data?

Frequency-response data, also called *frequency-function* data, consists of complex frequency-response values for a linear system characterized by its transfer function *G*. Frequency-response data tells you how the system handles sinusoidal inputs. You can measure frequency-response data values directly using a spectrum analyzer, for example, which provides a compact representation of the input-output relationship (compared to storing input and output independently).

The transfer function *G* is an operator that takes the input *u* of a linear system to the output *y*:

$$y = Gu$$

For a continuous-time system, the transfer function relates the Laplace transforms of the input *U(s)* and output *Y(s)*:

$$Y(s) = G(s)U(s)$$

In this case, the frequency function *G(iw)* is the transfer function evaluated on the imaginary axis *s=iw*.

For a discrete-time system sampled with a time interval *T*, the transfer function relates the Z-transforms of the input *U(z)* and output *Y(z)*:

$$Y(z) = G(z)U(z)$$

In this case, the frequency function $G(e^{iwT})$ is the transfer function *G(z)* evaluated on the unit circle. The argument of the frequency function $G(e^{iwT})$ is scaled by the sample time *T* to make the frequency function periodic with the sampling frequency $2\pi/T$.

When the input to the system is a sinusoid of a specific frequency, the output is also a sinusoid with the same frequency. The amplitude of the output is $|G|$ times the

amplitude of the input. The phase of the shifted from the input by $\varphi = \arg G$. G is evaluated at the frequency of the input sinusoid.

Frequency-response data represents a (nonparametric) model of the relationship between the input and the outputs as a function of frequency. You might use such a model, which consists of a table or plot of values, to study the system frequency response. However, this model is not suitable for simulation and prediction. You should create parametric model from the frequency-response data.

For more information on how to obtain identification data, see "Ways to Obtain Identification Data" on page 2-5.

How to Represent Frequency-Response Data in MATLAB

You can represent frequency-response data in two ways:

- Complex-values $G(e^{i\omega})$, for given frequencies ω
- Amplitude $|G|$ and phase shift $\varphi = \arg G$ values

You can import both the formats directly in the System Identification app. At the command line, you must represent complex data using an `frd` or `idfrd` object. If the data is in amplitude and phase format, convert it to complex frequency-response vector using $h(\omega) = A(\omega)e^{j\Phi(\omega)}$.

You must organize frequency-response data in the following format:

Frequency-Response Data Representation	For Single-Input Single-Output (SISO) Data	For Multi-Input Multi-Output (MIMO) Data
Complex Values	• Frequency function must be a column vector. • Frequency values must be a column vector.	• Frequency function must be an N_y-by-N_u-by-N_f array, where N_u is the number of inputs, N_y is the number of outputs, and N_f is the number of frequency measurements. • Frequency values must be a column vector.

Frequency-Response Data Representation	For Single-Input Single-Output (SISO) Data	For Multi-Input Multi-Output (MIMO) Data
Amplitude and phase shift values	• Amplitude and phase must each be a column vector. • Frequency values must be a column vector.	• Amplitude and phase must each be an N_y-by-N_u-by-N_f array, where N_u is the number of inputs, N_y is the number of outputs, and N_f is the number of frequency measurements. • Frequency values must be a column vector.

For more information about importing data into the MATLAB workspace, see “Data Import and Export” (MATLAB).

After you have the variables in the MATLAB workspace, import them into the System Identification app or create a data object for working at the command line. For more information about importing data into the app, see “Importing Frequency-Response Data into the App” on page 2-21. To learn more about creating a data object, see “Representing Frequency-Response Data Using idfrd Objects” on page 2-85.

Import Time-Domain Data into the App

Before you can import time-domain data into the **System Identification** app, you must import the data into the MATLAB workspace, as described in "Time-Domain Data Representation" on page 2-9.

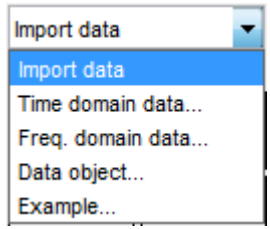
Note Your time-domain data must be sampled at equal time intervals. The input and output signals must have the same number of data samples.

To import data into the app:

1. Type the following command in the MATLAB Command Window to open the app:

   ```
   systemIdentification
   ```

2. In the System Identification app window, select **Import data** > **Time domain data**. This action opens the Import Data dialog box.



3. Specify the following options:

Note For time series, only import the output signal and enter `[]` for the input.

- **Input** — Enter the MATLAB variable name (column vector or matrix) or a MATLAB expression that represents the input data. The expression must evaluate to a column vector or matrix.
- **Output** — Enter the MATLAB variable name (column vector or matrix) or a MATLAB expression that represents the output data. The expression must evaluate to a column vector or matrix.
- **Data name** — Enter the name of the data set, which appears in the System Identification app window after the import operation is completed.

- **Starting time** — Enter the starting value of the time axis for time plots.
- **Sample time** — Enter the actual sample time in the experiment. For more information about this setting, see "Specifying the Data Sample Time" on page 2-28.

Tip The System Identification Toolbox product uses the sample time during model estimation and to set the horizontal axis on time plots. If you transform a time-domain signal to a frequency-domain signal, the Fourier transforms are computed as discrete Fourier transforms (DFTs) using this sample time.

---

4 (Optional) In the **Data Information** area, click **More** to expand the dialog box and enter the following settings:

Input Properties

- **InterSample** — This options specifies the behavior of the input signals between samples during data acquisition. It is used when transforming models from discrete-time to continuous-time and when resampling the data.
  - `zoh` (zero-order hold) indicates that the input was piecewise-constant during data acquisition.
  - `foh` (first-order hold) indicates that the output was piecewise-linear during data acquisition.
  - `bl` (bandwidth-limited behavior) specifies that the continuous-time input signal has zero power above the Nyquist frequency (equal to the inverse of the sample time).

Note See the `d2c` and `c2d` reference pages for more information about transforming between discrete-time and continuous-time models.

- **Period** — Enter `Inf` to specify a nonperiodic input. If the underlying time-domain data was periodic over an integer number of periods, enter the period of the input signal.

Note If your data is periodic, always include a whole number of periods for model estimation.

Channel Names

- **Input** — Enter the name of one or more input channels.

  Tip Naming channels helps you to identify data in plots. For multivariable input-output signals, you can specify the names of individual **Input** and **Output** channels, separated by commas.

- **Output** — Enter the name of one or more output channels.

Physical Units of Variables

- **Input** — Enter the input units.

  Tip When you have multiple inputs and outputs, enter a comma-separated list of **Input** and **Output** units corresponding to each channel.

- **Output** — Enter the output units.

Notes — Enter comments about the experiment or the data. For example, you might enter the experiment name, date, and a description of experimental conditions. Models you estimate from this data inherit your data notes.

5 Click **Import**. This action adds a new data icon to the System Identification app window.

6 Click **Close** to close the Import Data dialog box.

Import Frequency-Domain Data into the App

In this section...
"Importing Frequency-Domain Input/Output Signals into the App" on page 2-19 "Importing Frequency-Response Data into the App" on page 2-21

Importing Frequency-Domain Input/Output Signals into the App

Frequency-domain data consists of Fourier transforms of time-domain data (a function of frequency).

Before you can import frequency-domain data into the **System Identification** app, you must import the data into the MATLAB workspace, as described in "Frequency-Domain Input/Output Signal Representation" on page 2-11.

Note The input and output signals must have the same number of data samples.

To import data into the app:

1. Type the following command in the MATLAB Command Window to open the app:

   ```
   systemIdentification
   ```

2. In the System Identification app window, select **Import data > Freq. domain data**. This action opens the Import Data dialog box.
3. Specify the following options:

   - **Input** — Enter the MATLAB variable name (column vector or matrix) or a MATLAB expression that represents the input data. The expression must evaluate to a column vector or matrix.
   - **Output** — Enter the MATLAB variable name (column vector or matrix) or a MATLAB expression that represents the output data. The expression must evaluate to a column vector or matrix.
   - **Frequency** — Enter the MATLAB variable name of a vector or a MATLAB expression that represents the frequencies. The expression must evaluate to a column vector.

The frequency vector must have the same number of rows as the input and output signals.

- **Data name** — Enter the name of the data set, which appears in the System Identification app window after the import operation is completed.
- **Frequency unit** — Enter `Hz` for Hertz or keep the `rad/s` default value.
- **Sample time** — Enter the actual sample time in the experiment. For continuous-time data, enter `0`. For more information about this setting, see "Specifying the Data Sample Time" on page 2-28.

4 (Optional) In the **Data Information** area, click **More** to expand the dialog box and enter the following optional settings:

Input Properties

- **InterSample** — This options specifies the behavior of the input signals between samples during data acquisition. It is used when transforming models from discrete-time to continuous-time and when resampling the data.
  - `zoh` (zero-order hold) indicates that the input was piecewise-constant during data acquisition.
  - `foh` (first-order hold) indicates that the output was piecewise-linear during data acquisition.
  - `bl` (bandwidth-limited behavior) specifies that the continuous-time input signal has zero power above the Nyquist frequency (equal to the inverse of the sample time).

Note See the `d2c` and `c2d` reference page for more information about transforming between discrete-time and continuous-time models.

- **Period** — Enter `Inf` to specify a nonperiodic input. If the underlying time-domain data was periodic over an integer number of periods, enter the period of the input signal.

Note If your data is periodic, always include a whole number of periods for model estimation.

Channel Names

- **Input** — Enter the name of one or more input channels.

Tip Naming channels helps you to identify data in plots. For multivariable input and output signals, you can specify the names of individual **Input** and **Output** channels, separated by commas.

- **Output** — Enter the name of one or more output channels.

Physical Units of Variables

- **Input** — Enter the input units.

Tip When you have multiple inputs and outputs, enter a comma-separated list of **Input** and **Output** units corresponding to each channel.

- **Output** — Enter the output units.

Notes — Enter comments about the experiment or the data. For example, you might enter the experiment name, date, and a description of experimental conditions. Models you estimate from this data inherit your data notes.

5 Click **Import**. This action adds a new data icon to the System Identification app window.

6 Click **Close** to close the Import Data dialog box.

Importing Frequency-Response Data into the App

- "Prerequisite" on page 2-21
- "Importing Complex-Valued Frequency-Response Data" on page 2-21
- "Importing Amplitude and Phase Frequency-Response Data" on page 2-23

Prerequisite

Before you can import frequency-response data into the System Identification app, you must import the data into the MATLAB workspace, as described in "Frequency-Response Data Representation" on page 2-13.

Importing Complex-Valued Frequency-Response Data

To import frequency-response data consisting of complex-valued frequency values at specified frequencies:

1 Type the following command in the MATLAB Command Window to open the app:

```
systemIdentification
```

2. In the System Identification app window, select **Import data > Freq. domain data**. This action opens the Import Data dialog box.
3. In the **Data Format for Signals** list, select **Freq. Function (Complex)**.
4. Specify the following options:

   - **Response** — Enter the MATLAB variable name or a MATLAB expression that represents the complex frequency-response data $G(e^{iw})$.
   - **Frequency** — Enter the MATLAB variable name of a vector or a MATLAB expression that represents the frequencies. The expression must evaluate to a column vector.
   - **Data name** — Enter the name of the data set, which appears in the System Identification app window after the import operation is completed.
   - **Frequency unit** — Enter `Hz` for Hertz or keep the `rad/s` default value.
   - **Sample time** — Enter the actual sample time in the experiment. For continuous-time data, enter `0`. For more information about this setting, see "Specifying the Data Sample Time" on page 2-28.
5. (Optional) In the **Data Information** area, click **More** to expand the dialog box and enter the following optional settings:

   Channel Names

   - **Input** — Enter the name of one or more input channels.

     Tip Naming channels helps you to identify data in plots. For multivariable input and output signals, you can specify the names of individual **Input** and **Output** channels, separated by commas.

   - **Output** — Enter the name of one or more output channels.

   Physical Units of Variables

   - **Input** — Enter the input units.

     Tip When you have multiple inputs and outputs, enter a comma-separated list of **Input** and **Output** units corresponding to each channel.

   - **Output** — Enter the output units.

Notes — Enter comments about the experiment or the data. For example, you might enter the experiment name, date, and a description of experimental conditions. Models you estimate from this data inherit your data notes.

6 Click **Import**. This action adds a new data icon to the System Identification app window.

7 Click **Close** to close the Import Data dialog box.

Importing Amplitude and Phase Frequency-Response Data

To import frequency-response data consisting of amplitude and phase values at specified frequencies:

1 Type the following command in the MATLAB Command Window to open the app:

```
systemIdentification
```

2 In the System Identification app window, select **Import data > Freq. domain data**. This action opens the Import Data dialog box.

3 In the **Data Format for Signals** list, select **Freq. Function (Amp/Phase)**.

4 Specify the following options:

- **Amplitude** — Enter the MATLAB variable name or a MATLAB expression that represents the amplitude $|G|$.
- **Phase (deg)** — Enter the MATLAB variable name or a MATLAB expression that represents the phase $\varphi = \arg G$.
- **Frequency** — Enter the MATLAB variable name of a vector or a MATLAB expression that represents the frequencies. The expression must evaluate to a column vector.
- **Data name** — Enter the name of the data set, which appears in the System Identification app window after the import operation is completed.
- **Frequency unit** — Enter `Hz` for Hertz or keep the `rad/s` default value.
- **Sample time** — Enter the actual sample time in the experiment. For continuous-time data, enter `0`. For more information about this setting, see “Specifying the Data Sample Time” on page 2-28.

5 (Optional) In the **Data Information** area, click **More** to expand the dialog box and enter the following optional settings:

Channel Names

- **Input** — Enter the name of one or more input channels.

  Tip Naming channels helps you to identify data in plots. For multivariable input and output signals, you can specify the names of individual **Input** and **Output** channels, separated by commas.

- **Output** — Enter the name of one or more output channels.

Physical Units of Variables

- **Input** — Enter the input units.

  Tip When you have multiple inputs and outputs, enter a comma-separated list of **Input** and **Output** units corresponding to each channel.

- **Output** — Enter the output units.

Notes — Enter comments about the experiment or the data. For example, you might enter the experiment name, date, and a description of experimental conditions. Models you estimate from this data inherit your data notes.

6 Click **Import**. This action adds a new data icon to the System Identification app window.

7 Click **Close** to close the Import Data dialog box.

Import Data Objects into the App

You can import the System Identification Toolbox `iddata` and `idfrd` data objects into the System Identification app.

Before you can import a data object into the System Identification app, you must create the data object in the MATLAB workspace, as described in “Representing Time- and Frequency-Domain Data Using iddata Objects” on page 2-50 or “Representing Frequency-Response Data Using idfrd Objects” on page 2-85.

Note You can also import a Control System Toolbox `frd` object. Importing an `frd` object converts it to an `idfrd` object.

Select **Import data** > **Data object** to open the Import Data dialog box.

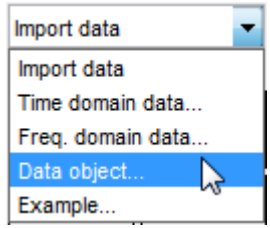
Import `iddata`, `idfrd`, or `frd` data object in the MATLAB workspace.

To import a data object into the app:

1. Type the following command in the MATLAB Command Window to open the app:

   ```
   systemIdentification
   ```

2. In the System Identification app window, select **Import data** > **Data object**.

   This action opens the Import Data dialog box. **IDDATA or IDFRD/FRD** is already selected in the **Data Format for Signals** list.

3. Specify the following options:

   - **Object** — Enter the name of the MATLAB variable that represents the data object in the MATLAB workspace. Press **Enter**.

- **Data name** — Enter the name of the data set, which appears in the System Identification app window after the import operation is completed.
- (Only for time-domain `iddata` object) **Starting time** — Enter the starting value of the time axis for time plots.
- (Only for frequency domain `iddata` or `idfrd` object) **Frequency unit** — Enter the frequency unit for response plots.
- **Sample time** — Enter the actual sample time in the experiment. For more information about this setting, see "Specifying the Data Sample Time" on page 2-28.

Tip The System Identification Toolbox product uses the sample time during model estimation and to set the horizontal axis on time plots. If you transform a time-domain signal to a frequency-domain signal, the Fourier transforms are computed as discrete Fourier transforms (DFTs) using this sample time.

4 (Optional) In the **Data Information** area, click **More** to expand the dialog box and enter the following optional settings:

(Only for `iddata` object) **Input Properties**

- **InterSample** — This options specifies the behavior of the input signals between samples during data acquisition. It is used when transforming models from discrete-time to continuous-time and when resampling the data.
  - `zoh` (zero-order hold) indicates that the input was piecewise-constant during data acquisition.
  - `foh` (first-order hold) indicates that the input was piecewise-linear during data acquisition.
  - `bl` (bandwidth-limited behavior) specifies that the continuous-time input signal has zero power above the Nyquist frequency (equal to the inverse of the sample time).

Note See the `d2c` and `c2d` reference page for more information about transforming between discrete-time and continuous-time models.

- **Period** — Enter `Inf` to specify a nonperiodic input. If the underlying time-domain data was periodic over an integer number of periods, enter the period of the input signal.

> **Note** If your data is periodic, always include a whole number of periods for model estimation.

Channel Names

- **Input** — Enter the name of one or more input channels.

  > **Tip** Naming channels helps you to identify data in plots. For multivariable input and output signals, you can specify the names of individual **Input** and **Output** channels, separated by commas.

- **Output** — Enter the name of one or more output channels.

Physical Units of Variables

- **Input** — Enter the input units.

  > **Tip** When you have multiple inputs and outputs, enter a comma-separated list of **Input** and **Output** units corresponding to each channel.

- **Output** — Enter the output units.

Notes — Enter comments about the experiment or the data. For example, you might enter the experiment name, date, and a description of experimental conditions. Models you estimate from this data inherit your data notes.

5 Click **Import**. This action adds a new data icon to the System Identification app window.

6 Click **Close** to close the Import Data dialog box.

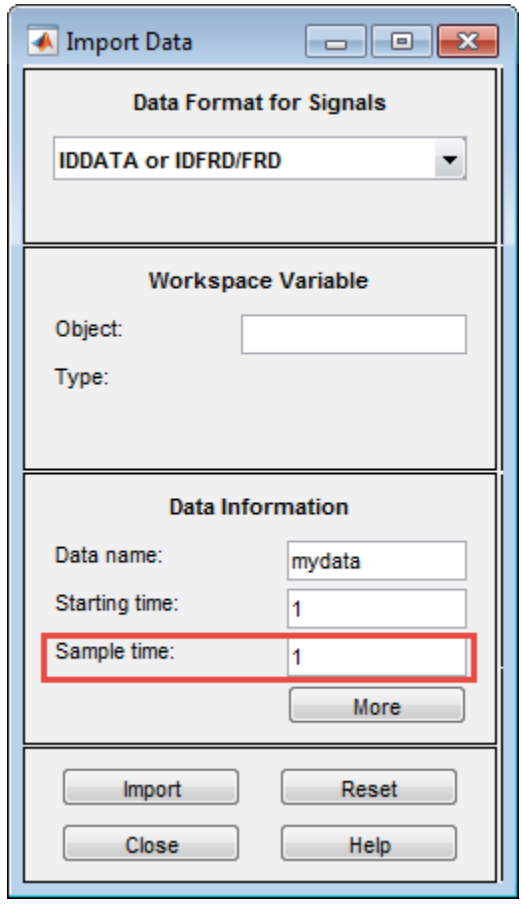
Specifying the Data Sample Time

When you import data into the app, you must specify the data sample time.

The *sample time* is the time between successive data samples in your experiment and must be the numerical time interval at which your data is sampled in any units. For example, enter `0.5` if your data was sampled every `0.5 s`, and enter `1` if your data was sampled every `1 s`.

You can also use the sample time as a flag to specify continuous-time data. When importing continuous-time frequency domain or frequency-response data, set the **Sample time** to `0`.

The sample time is used during model estimation. For time-domain data, the sample time is used together with the start time to calculate the sampling time instants. When you transform time-domain signals to frequency-domain signals (see the `fft` reference page), the Fourier transforms are computed as discrete Fourier transforms (DFTs) for this sample time. In addition, the sampling instants are used to set the horizontal axis on time plots.



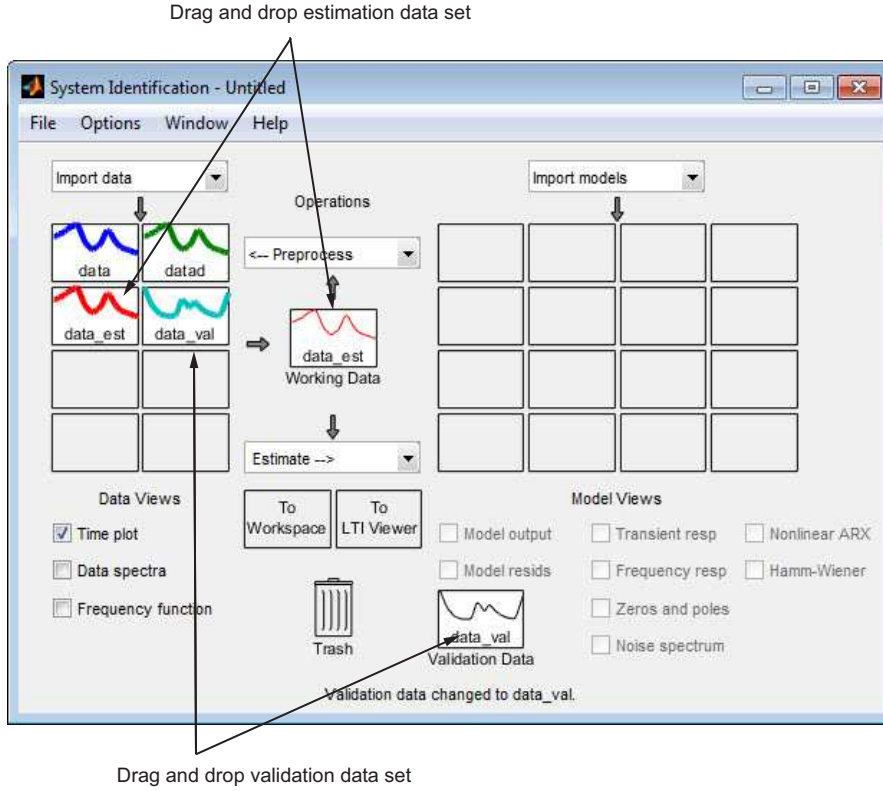
Sample Time in the Import Data dialog box

Specify Estimation and Validation Data in the App

You should use different data sets to estimate and validate your model for best validation results.

In the System Identification app, **Working Data** refers to estimation data. Similarly, **Validation Data** refers to the data set you use to validate a model. For example, when you plot the model output, the input to the model is the input signal from the validation data set. This plot compares model output to the measured output in the validation data set. Selecting **Model resids** performs residual analysis using the validation data.

To specify **Working Data**, drag and drop the corresponding data icon into the **Working Data** rectangle, as shown in the following figure. Similarly, to specify **Validation Data**, drag and drop the corresponding data icon into the **Validation Data** rectangle. Alternatively, right-click the icon to open the Data/model Info dialog box. Select the **Use as Working Data** or **Use as Validation Data** and click **Apply** to specify estimation and validation data, respectively.



See Also

More About

- "Select Subsets of Data" on page 2-104

Preprocess Data Using Quick Start

As a preprocessing shortcut for time-domain data, select **Preprocess** > **Quick start** to simultaneously perform the following four actions:

- Subtract the mean value from each channel.

  Note For information about when to subtract mean values from the data, see "Handling Offsets and Trends in Data" on page 2-114.

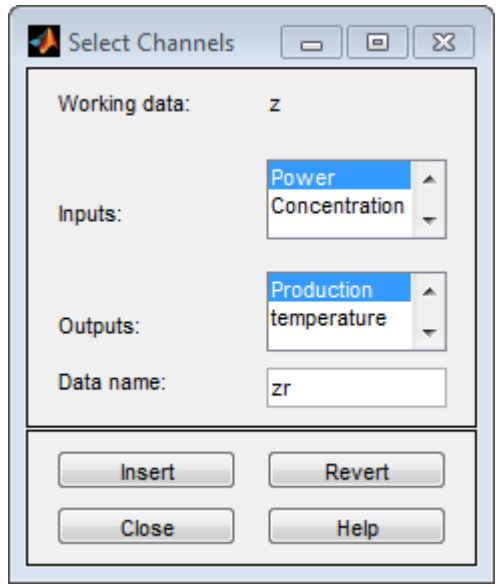
- Split data into two parts.
- Specify the first part as estimation data for models (or **Working Data**).
- Specify the second part as **Validation Data**.

Create Data Sets from a Subset of Signal Channels

You can create a new data set in the System Identification app by extracting subsets of input and output channels from an existing data set.

To create a new data set from selected channels:

1. In the System Identification app, drag the icon of the data from which you want to select channels to the **Working Data** rectangle.
2. Select **Preprocess** > **Select channels** to open the Select Channels dialog box.



   The **Inputs** list displays the input channels and the **Outputs** list displays the output channels in the selected data set.
3. In the **Inputs** list, select one or more channels in any of following ways:

   - Select one channel by clicking its name.
   - Select adjacent channels by pressing the **Shift** key while clicking the first and last channel names.
   - Select nonadjacent channels by pressing the **Ctrl** key while clicking each channel name.

Tip To exclude input channels and create time-series data, clear all selections by holding down the **Ctrl** key and clicking each selection. To reset selections, click **Revert**.

4 In the **Outputs** list, select one or more channels in any of following ways:

- Select one channel by clicking its name.
- Select adjacent channels by pressing the **Shift** key while clicking the first and last channel names.
- Select nonadjacent channels by pressing the **Ctrl** key while clicking each channel name.

Tip To reset selections, click **Revert**.

5 In the **Data name** field, type the name of the new data set. Use a name that is unique in the Data Board.

6 Click **Insert** to add the new data set to the Data Board in the System Identification app.

7 Click **Close**.

Create Multiexperiment Data Sets in the App

In this section...
"Why Create Multiexperiment Data?" on page 2-35 "Limitations on Data Sets" on page 2-35 "Merging Data Sets" on page 2-35 "Extracting Specific Experiments from a Multiexperiment Data Set into a New Data Set" on page 2-39

Why Create Multiexperiment Data?

You can create a time-domain or frequency-domain data set in the System Identification app that includes several experiments. Identifying models for multiexperiment data results in an *average* model.

Experiments can mean data that was collected during different sessions, or portions of the data collected during a single session. In the latter situation, you can create multiexperiment data by splitting a single data set into multiple segments that exclude corrupt data, and then merge the good data segments.

Limitations on Data Sets

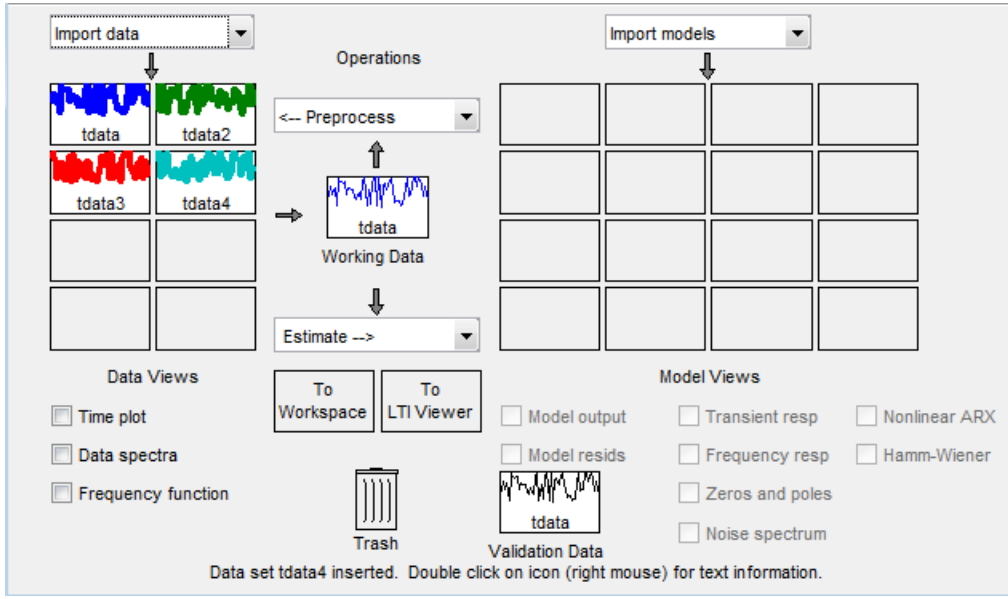
You can only merge data sets that have *all* of the following characteristics:

- Same number of input and output channels.
- Different names. The name of each data set becomes the experiment name in the merged data set.
- Same input and output channel names.
- Same data domain (that is, time-domain data or frequency-domain data only).

Merging Data Sets

You can merge data sets using the System Identification app.

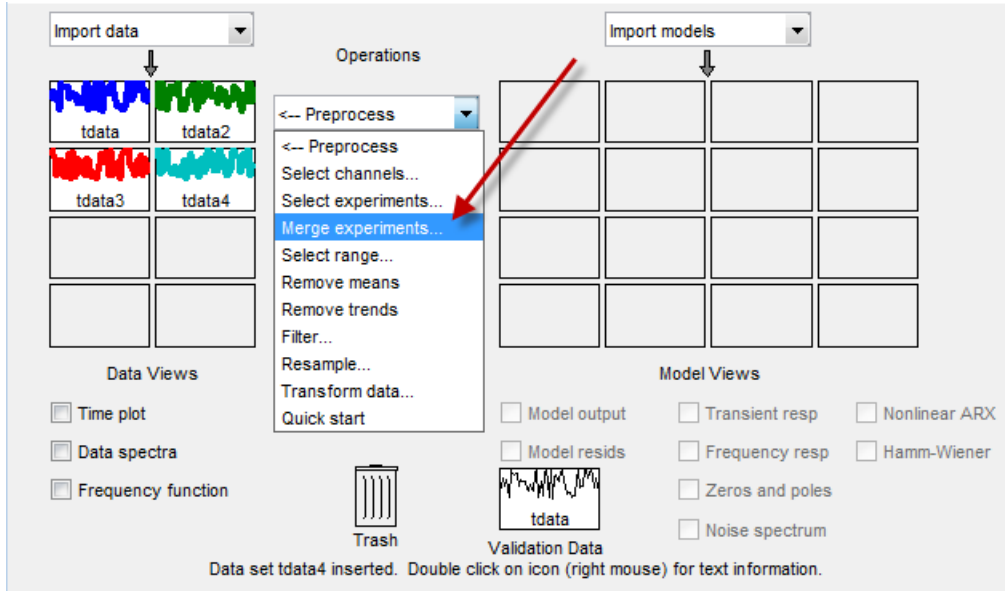
For example, suppose that you want to combine the data sets `tdata`, `tdata2`, `tdata3`, `tdata4` shown in the following figure.



App Contains Four Data Sets to Merge

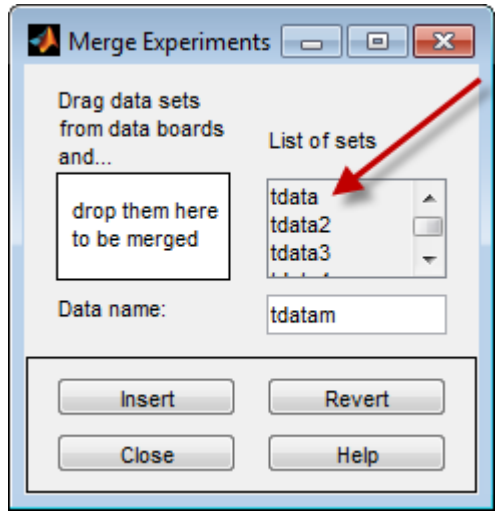
To merge data sets in the app:

1 In the **Operations** area, select **<--Preprocess > Merge experiments** from the drop-down menu to open the Merge Experiments dialog box.



2 In the System Identification app window, drag a data set icon to the Merge Experiments dialog box, to the **drop them here to be merged** rectangle.

The name of the data set is added to the **List of sets**. Repeat for each data set you want to merge.

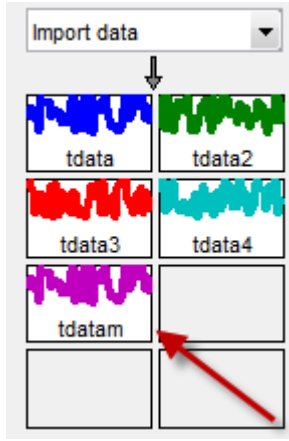


tdata and tdata2 to Be Merged

Tip To empty the list, click **Revert**.

3 In the **Data name** field, type the name of the new data set. This name must be unique in the Data Board.

4 Click **Insert** to add the new data set to the Data Board in the System Identification app window.



Data Board Now Contains tdatam with Merged Experiments

5 Click **Close** to close the Merge Experiments dialog box.

Tip To get information about a data set in the System Identification app, right-click the data icon to open the Data/model Info dialog box.

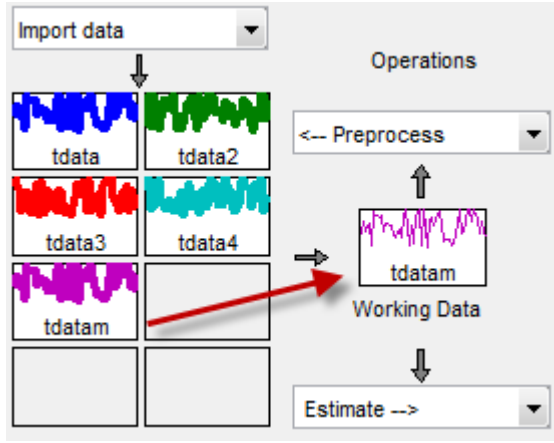
Extracting Specific Experiments from a Multiexperiment Data Set into a New Data Set

When a data set already consists of several experiments, you can extract one or more of these experiments into a new data set, using the System Identification app.

For example, suppose that `tdatam` consists of four experiments.

To create a new data set that includes only the first and third experiments in this data set:

1 In the System Identification app window, drag and drop the `tdatam` data icon to the **Working Data** rectangle.

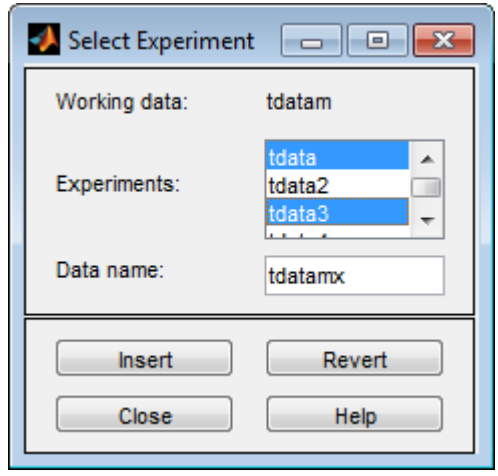


tdatam Is Set to Working Data

2 In the **Operations** area, select **Preprocess > Select experiments** from the drop-down menu to open the Select Experiment dialog box.

3 In the **Experiments** list, select one or more data sets in either of the following ways:

- Select one data set by clicking its name.
- Select adjacent data sets by pressing the **Shift** key while clicking the first and last names.
- Select nonadjacent data sets by pressing the **Ctrl** key while clicking each name.



4 In the **Data name** field, type the name of the new data set. This name must be unique in the Data Board.

5 Click **Insert** to add the new data set to the Data Board in the System Identification app.

6 Click **Close** to close the Select Experiment dialog box.

See Also

More About

- "Select Subsets of Data" on page 2-104
- "Create Multiexperiment Data at the Command Line" on page 2-61

Managing Data in the App

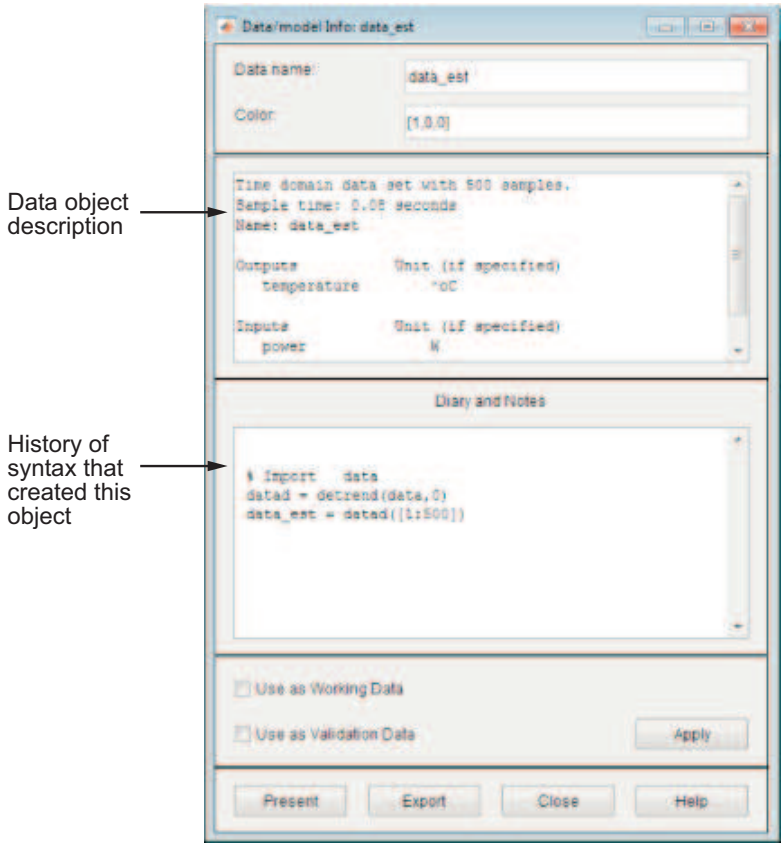
In this section...
“Viewing Data Properties” on page 2-42
“Renaming Data and Changing Display Color” on page 2-43
“Distinguishing Data Types” on page 2-46
“Organizing Data Icons” on page 2-46
“Deleting Data Sets” on page 2-47
“Exporting Data to the MATLAB Workspace” on page 2-48

Viewing Data Properties

You can get information about each data set in the System Identification app by right-clicking the corresponding data icon.

The Data/model Info dialog box opens. This dialog box describes the contents and the properties of the corresponding data set. It also displays any associated notes and the command-line equivalent of the operations you used to create this data.

Tip To view or modify properties for several data sets, keep this window open and right-click each data set in the System Identification app. The Data/model Info dialog box updates as you select each data set.



To displays the data properties in the MATLAB Command Window, click **Present**.

Renaming Data and Changing Display Color

You can rename data and change its display color by double-clicking the data icon in the System Identification app.

The Data/model Info dialog box opens. This dialog box describes both the contents and the properties of the data. The object description area displays the syntax of the operations you used to create the data in the app.

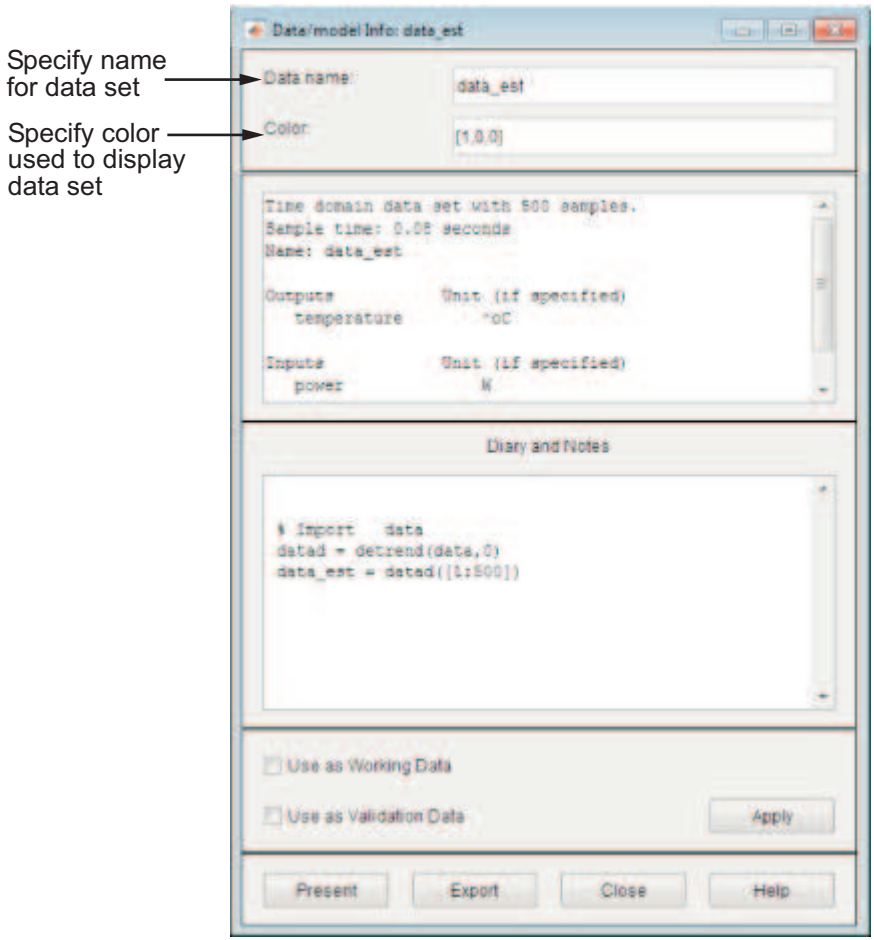
The Data/model Info dialog box also lets you rename the data by entering a new name in the **Data name** field.

You can also specify a new display color using three RGB values in the **Color** field. Each value is between `0` to `1` and indicates the relative presence of red, green, and blue, respectively. For more information about specifying default data color, see “Customizing the System Identification App” on page 21-15.

Tip As an alternative to using three RGB values, you can enter any *one* of the following:

```
'y' 'r' 'b' 'c' 'g' 'm' 'k'
```

These represent yellow, red, blue, cyan, green, magenta, and black, respectively.



Information About the Data

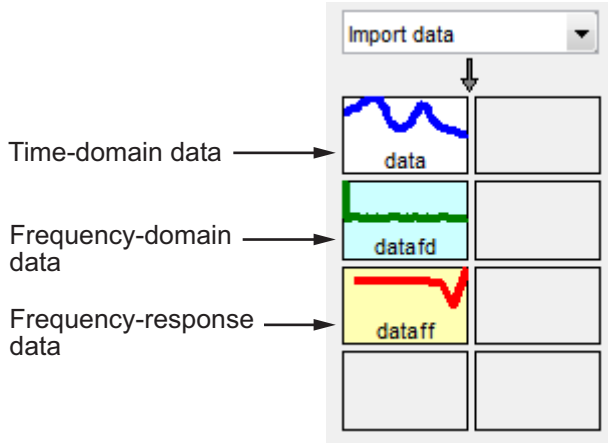
You can enter comments about the origin and state of the data in the **Diary And Notes** area. For example, you might want to include the experiment name, date, and the description of experimental conditions. When you estimate models from this data, these notes are associated with the models.

Clicking **Present** display portions of this information in the MATLAB Command Window.

Distinguishing Data Types

The background color of a data icon is color-coded, as follows:

- White background represents time-domain data.
- Blue background represents frequency-domain data.
- Yellow background represents frequency-response data.



Colors Representing Type of Data

Organizing Data Icons

You can rearrange data icons in the System Identification app by dragging and dropping the icons to empty Data Board rectangles in the app.

Note You cannot drag and drop a data icon into the model area on the right.

When you need additional space for organizing data or model icons, select **Options** > **Extra model/data board** in the System Identification app. This action opens an extra session window with blank rectangles for data and models. The new window is an extension of the current session and does not represent a new session.

Tip When you import or create data sets and there is insufficient space for the icons, an additional session window opens automatically.

You can drag and drop data between the main System Identification app and any extra session windows.

Type comments in the **Notes** field to describe the data sets. When you save a session, as described in "Saving, Merging, and Closing Sessions" on page 21-6, all additional windows and notes are also saved.

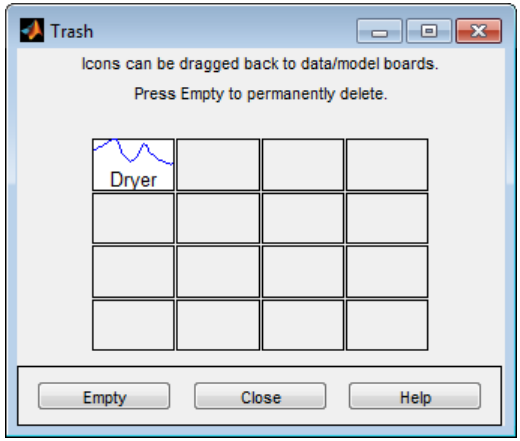
Deleting Data Sets

To delete data sets in the System Identification app, drag and drop the corresponding icon into **Trash**. You can also use the **Delete** key on your keyboard to move items to the **Trash**. Moving items to **Trash** does not permanently delete these items.

Note You cannot delete a data set that is currently designated as **Working Data** or **Validation Data**. You must first specify a different data set in the System Identification app to be **Working Data** or **Validation Data**, as described in "Specify Estimation and Validation Data in the App" on page 2-30.

To restore a data set from **Trash**, drag its icon from **Trash** to the Data or Model Board in the System Identification app window. You can view the **Trash** contents by double-clicking the **Trash** icon.

Note You must restore data to the Data Board; you cannot drag data icons to the Model Board.



To permanently delete all items in **Trash**, select **Options** > **Empty trash**.

Exiting a session empties the **Trash** automatically.

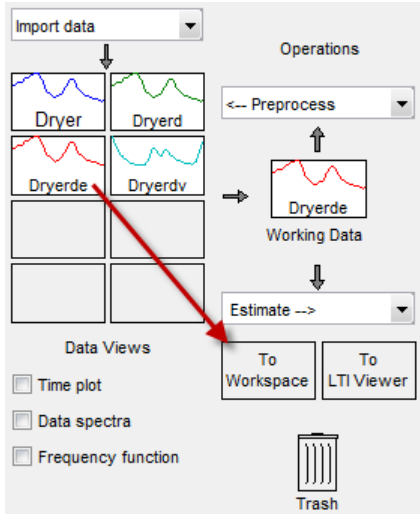
Exporting Data to the MATLAB Workspace

The data you create in the System Identification app is not available in the MATLAB workspace until you export the data set. Exporting to the MATLAB workspace is necessary when you need to perform an operation on the data that is only available at the command line.

To export a data set to the MATLAB workspace, do one of the following:

- Drag and drop the corresponding icon to the **To Workspace** rectangle.
- Right-click the icon to open the Data/model Info dialog box. Click **Export**.

When you export data to the MATLAB workspace, the resulting variables have the same name as in the System Identification app. For example, the following figure shows how to export the time-domain data object `datad`.



Exporting Data to the MATLAB Workspace

In this example, the MATLAB workspace contains a variable named `data` after export.

Representing Time- and Frequency-Domain Data Using iddata Objects

In this section...
“iddata Constructor” on page 2-50
“iddata Properties” on page 2-52
“Select Data Channels, I/O Data and Experiments in iddata Objects” on page 2-55
“Increasing Number of Channels or Data Points of iddata Objects” on page 2-58

iddata Constructor

Requirements for Constructing an iddata Object

To construct an `iddata` object, you must have already imported data into the MATLAB workspace, as described in “Representing Data in MATLAB Workspace” on page 2-9.

Constructing an iddata Object for Time-Domain Data

Use the following syntax to create a time-domain `iddata` object `data`:

```
data = iddata(y,u,Ts)
```

You can also specify additional properties, as follows:

```
data = iddata(y,u,Ts,'Property1',Value1,...,'PropertyN',ValueN)
```

For more information about accessing object properties, see “Properties”.

In this example, `Ts` is the sample time, or the time interval, between successive data samples. For uniformly sampled data, `Ts` is a scalar value equal to the sample time of your experiment. The default time unit is seconds, but you can set it to a new value using the `TimeUnit` property. For more information about `iddata` time properties, see “Modifying Time and Frequency Vectors” on page 2-79.

For nonuniformly sampled data, specify `Ts` as `[]`, and set the value of the `SamplingInstants` property as a column vector containing individual time values. For example:

```
data = iddata(y,u,[],'SamplingInstants',TimeVector)
```

Where `TimeVector` represents a vector of time values.

Note You can modify the property `SamplingInstants` by setting it to a new vector with the length equal to the number of data samples.

To represent time-series data, use the following syntax:

```
ts_data = iddata(y,[],Ts)
```

where `y` is the output data, `[]` indicates empty input data, and `Ts` is the sample time.

The following example shows how to create an `iddata` object using single-input/single-output (SISO) data from `dryer2.mat`. The input and output each contain 1000 samples with the sample time of `0.08` second.

```
% Load input u2 and output y2 .
load dryer2
% Create iddata object.
data = iddata(y2,u2,0.08)

data =

Time domain data set with 1000 samples.
Sample time: 0.08 seconds

Outputs      Unit (if specified)
   y1

Inputs       Unit (if specified)
   u1
```

The default channel name `'y1'` is assigned to the first and only output channel. When `y2` contains several channels, the channels are assigned default names `'y1','y2','y2',...,'yn'`. Similarly, the default channel name `'u1'` is assigned to the first and only input channel. For more information about naming channels, see “Naming, Adding, and Removing Data Channels” on page 2-82.

Constructing an iddata Object for Frequency-Domain Data

Frequency-domain data is the Fourier transform of the input and output signals at specific frequency values. To represent frequency-domain data, use the following syntax to create the `iddata` object:

```
data = iddata(y,u,Ts,'Frequency',w)
```

`'Frequency'` is an `iddata` property that specifies the frequency values `w`, where `w` is the frequency column vector that defines the frequencies at which the Fourier transform values of `y` and `u` are computed. `Ts` is the time interval between successive data samples in seconds for the original time-domain data. `w`, `y`, and `u` have the same number of rows.

Note You must specify the frequency vector for frequency-domain data.

For more information about `iddata` time and frequency properties, see "Modifying Time and Frequency Vectors" on page 2-79.

To specify a continuous-time system, set `Ts` to `0`.

You can specify additional properties when you create the `iddata` object, as follows:

```
data = iddata(y,u,Ts,'Property1',Value1,...,'PropertyN',ValueN)
```

For more information about accessing object properties, see "Properties".

iddata Properties

To view the properties of the `iddata` object, use the `get` command. For example, type the following commands at the prompt:

```
% Load input u2 and output y2.
load dryer2
% Create iddata object.
data = iddata(y2,u2,0.08);
% Get property values of data.
get(data)


ans =
```

```
  struct with fields:

              Domain: 'Time'
                Name: ''
          OutputData: [1000x1 double]
                   y: 'Same as OutputData'
          OutputName: {'y1'}
          OutputUnit: {''}
           InputData: [1000x1 double]
                   u: 'Same as InputData'
           InputName: {'u1'}
           InputUnit: {''}
              Period: Inf
        InterSample: 'zoh'
                  Ts: 0.0800
              Tstart: []
    SamplingInstants: [1000x0 double]
            TimeUnit: 'seconds'
      ExperimentName: 'Exp1'
               Notes: {}
            UserData: []
```

For a complete description of all properties, see the `iddata` reference page.

You can specify properties when you create an `iddata` object using the constructor syntax:

```
data = iddata(y,u,Ts,'Property1',Value1,...,'PropertyN',ValueN)
```

To change property values for an existing `iddata` object, use the `set` command or dot notation. For example, to change the sample time to `0.05`, type the following at the prompt:

```
set(data,'Ts',0.05)
```

or equivalently:

```
data.ts = 0.05
```

Property names are not case sensitive. You do not need to type the entire property name if the first few letters uniquely identify the property.

Tip You can use `data.y` as an alternative to `data.OutputData` to access the output values, or use `data.u` as an alternative to `data.InputData` to access the input values.

An `iddata` object containing frequency-domain data includes frequency-specific properties, such as `Frequency` for the frequency vector and `Units` for frequency units (instead of `Tstart` and `SamplingInstants`).

To view the property list, type the following command sequence at the prompt:

```
% Load input u2 and output y2.
  load dryer2;
% Create iddata object.
  data = iddata(y2,u2,0.08);
% Take the Fourier transform of the data
% transforming it to frequency domain.
  data = fft(data)
% Get property values of data.
  get(data)


data =

Frequency domain data set with responses at 501 frequencies,
ranging from 0 to 39.27 rad/seconds
Sample time: 0.08 seconds

Outputs      Unit (if specified)
   y1

Inputs       Unit (if specified)
   u1


ans =

  struct with fields:

           Domain: 'Frequency'
             Name: ''
       OutputData: [501x1 double]
                y: 'Same as OutputData'
       OutputName: {'y1'}
       OutputUnit: {''}
```

```
       InputData: [501x1 double]
               u: 'Same as InputData'
       InputName: {'u1'}
       InputUnit: {''}
          Period: Inf
     InterSample: 'zoh'
              Ts: 0.0800
  FrequencyUnit: 'rad/TimeUnit'
       Frequency: [501x1 double]
        TimeUnit: 'seconds'
 ExperimentName: 'Exp1'
           Notes: {}
        UserData: []
```

Select Data Channels, I/O Data and Experiments in iddata Objects

Subreferencing Input and Output Data

Subreferencing data and its properties lets you select data values and assign new data and property values.

Use the following general syntax to subreference specific data values in `iddata` objects:

```
data(samples,outputchannels,inputchannels,experimentname)
```

In this syntax, `samples` specify one or more sample indexes, `outputchannels` and `inputchannels` specify channel indexes or channel names, and `experimentname` specifies experiment indexes or names.

For example, to retrieve samples `5` through `30` in the `iddata` object `data` and store them in a new `iddata` object `data_sub`, use the following syntax:

```
data_sub = data(5:30)
```

You can also use logical expressions to subreference data. For example, to retrieve all data values from a single-experiment data set that fall between sample instants `1.27` and `9.3` in the `iddata` object `data` and assign them to `data_sub`, use the following syntax:

```
data_sub = data(data.sa>1.27&data.sa<9.3)
```

Note You do not need to type the entire property name. In this example, `sa` in `data.sa` uniquely identifies the `SamplingInstants` property.

You can retrieve the input signal from an `iddata` object using the following commands:

```
u = get(data,'InputData')
```

or

```
data.InputData
```

or

```
data.u    % u is the abbreviation for InputData
```

Similarly, you can retrieve the output data using

```
data.OutputData
```

or

```
data.y    % y is the abbreviation for OutputData
```

Subreferencing Data Channels

Use the following general syntax to subreference specific data channels in `iddata` objects:

```
data(samples,outputchannels,inputchannels,experiment)
```

In this syntax, `samples` specify one or more sample indexes, `outputchannels` and `inputchannels` specify channel indexes or channel names, and `experimentname` specifies experiment indexes or names.

To specify several channel names, you must use a cell array of character vectors of names.

For example, suppose the `iddata` object `data` contains three output channels (named `y1`, `y2`, and `y3`), and four input channels (named `u1`, `u2`, `u3`, and `u4`). To select all data samples in `y3`, `u1`, and `u4`, type the following command at the prompt:

```
% Use a cell array to reference
% input channels 'u1' and 'u4'
data_sub = data(:,'y3',{'u1','u4'})
```

or equivalently

```
% Use channel indexes 1 and 4
% to reference the input channels
  data_sub = data(:,3,[1 4])
```

Tip Use a colon (`:`) to specify all samples or all channels, and the empty matrix (`[]`) to specify no samples or no channels.

If you want to create a time-series object by extracting only the output data from an `iddata` object, type the following command:

```
data_ts = data(:,:,[])
```

You can assign new values to subreferenced variables. For example, the following command assigns the first 10 values of output channel `1` of `data` to values in samples `101` through `110` in the output channel `2` of `data1`. It also assigns the values in samples `101` through `110` in the input channel `3` of `data1` to the first 10 values of input channel `1` of `data`.

```
data(1:10,1,1) = data1(101:110,2,3)
```

Subreferencing Experiments

Use the following general syntax to subreference specific experiments in `iddata` objects:

```
data(samples,outputchannels,inputchannels,experimentname)
```

In this syntax, `samples` specify one or more sample indexes, `outputchannels` and `inputchannels` specify channel indexes or channel names, and `experimentname` specifies experiment indexes or names.

When specifying several experiment names, you must use a cell array of character vectors of names. The `iddata` object stores experiments name in the `ExperimentName` property.

For example, suppose the `iddata` object `data` contains five experiments with default names, `Exp1`, `Exp2`, `Exp3`, `Exp4`, and `Exp5`. Use the following syntax to subreference the first and fifth experiment in `data`:

```
data_sub = data(:,:,:,{'Exp1','Exp5'}) % Using experiment name
```

or

```
data_sub = data(:,:,:,[1 5])           % Using experiment index
```

Tip Use a colon (`:`) to denote all samples and all channels, and the empty matrix (`[]`) to specify no samples and no channels.

Alternatively, you can use the `getexp` command. The following example shows how to subreference the first and fifth experiment in `data`:

```
data_sub = getexp(data,{'Exp1','Exp5'}) % Using experiment name
```

or

```
data_sub = getexp(data,[1 5])           % Using experiment index
```

The following example shows how to retrieve the first 100 samples of output channels 2 and 3 and input channels 4 to 8 of Experiment 3:

```
dat(1:100,[2,3],[4:8],3)
```

Increasing Number of Channels or Data Points of iddata Objects

iddata Properties Storing Input and Output Data

The `InputData` `iddata` property stores column-wise input data, and the `OutputData` property stores column-wise output data. For more information about accessing `iddata` properties, see “iddata Properties” on page 2-52.

Horizontal Concatenation

Horizontal concatenation of `iddata` objects creates a new `iddata` object that appends all `InputData` information and all `OutputData`. This type of concatenation produces a single object with more input and output channels. For example, the following syntax performs horizontal concatenation on the `iddata` objects `data1,data2,...,dataN`:

```
data = [data1,data2,...,dataN]
```

This syntax is equivalent to the following longer syntax:

```
data.InputData =
     [data1.InputData,data2.InputData,...,dataN.InputData]
data.OutputData =
     [data1.OutputData,data2.OutputData,...,dataN.OutputData]
```

For horizontal concatenation, `data1,data2,...,dataN` must have the same number of samples and experiments , and the same`Ts` and `Tstart` values.

The channels in the concatenated `iddata` object are named according to the following rules:

- *Combining default channel names* — If you concatenate `iddata` objects with default channel names, such as `u1` and `y1`, channels in the new `iddata` object are automatically renamed to avoid name duplication.
- *Combining duplicate input channels* — If `data1,data2,...,dataN` have input channels with duplicate user-defined names, such that `dataK` contains channel names that are already present in `dataJ` with `J < K`, the `dataK` channels are ignored.
- *Combining duplicate output channels* — If `data1,data2,...,dataN` have input channels with duplicate user-defined names, only the output channels with unique names are added during the concatenation.

Vertical Concatenation

Vertical concatenation of `iddata` objects creates a new `iddata` object that vertically stacks the input and output data values in the corresponding data channels. The resulting object has the same number of channels, but each channel contains more data points. For example, the following syntax creates a `data` object such that its total number of samples is the sum of the samples in `data1,data2,...,dataN`.

```
data = [data1;data2;... ;dataN]
```

This syntax is equivalent to the following longer syntax:

```
data.InputData =
     [data1.InputData;data2.InputData;...;dataN.InputData]
data.OutputData =
     [data1.OutputData;data2.OutputData;...;dataN.OutputData]
```

For vertical concatenation, `data1,data2,...,dataN` must have the same number of input channels, output channels, and experiments.

See Also

`iddata`

More About

- “Representing Data in MATLAB Workspace” on page 2-9
- “Managing iddata Objects” on page 2-79

Create Multiexperiment Data at the Command Line

In this section...
"Why Create Multiexperiment Data Sets?" on page 2-61
"Limitations on Data Sets" on page 2-61
"Entering Multiexperiment Data Directly" on page 2-61
"Merging Data Sets" on page 2-62
"Adding Experiments to an Existing iddata Object" on page 2-62

Why Create Multiexperiment Data Sets?

You can create `iddata` objects that contain several experiments. Identifying models for an `iddata` object with multiple experiments results in an *average* model.

In the System Identification Toolbox product, *experiments* can either mean data collected during different sessions, or portions of the data collected during a single session. In the latter situation, you can create a multiexperiment `iddata` object by splitting the data from a single session into multiple segments to exclude bad data, and merge the good data portions.

Note The `idfrd` object does not support the `iddata` equivalent of multiexperiment data.

Limitations on Data Sets

You can only merge data sets that have all of the following characteristics:

- Same number of input and output channels.
- Same input and output channel names.
- Same data domain (that is, time-domain data or frequency-domain data).

Entering Multiexperiment Data Directly

To construct an `iddata` object that includes *N* data sets, you can use this syntax:

```
data = iddata(y,u,Ts)
```

where `y`, `u`, and `Ts` are 1-by-*N* cell arrays containing data from the different experiments. Similarly, when you specify `Tstart`, `Period`, `InterSample`, and `SamplingInstants` properties of the `iddata` object, you must assign their values as 1-by-*N* cell arrays.

Merging Data Sets

This example shows how to create a multiexperiment `iddata` object by merging `iddata` objects, where each contains data from a single experiment or is a multiexperiment data set.

Load iddata objects `z1` and `z3`.

```
load iddata1
load iddata3
```

Merge experiments `z1` and `z3` into the iddata object `z`.

```
z = merge(z1,z3)

z =

Time domain data set containing 2 experiments.

Experiment   Samples      Sample Time
   Exp1         300            0.1
   Exp2         300            1

Outputs      Unit (if specified)
   y1

Inputs       Unit (if specified)
   u1
```

These commands create an `iddata` object that contains two experiments, where the experiments are assigned default names `'Exp1'` and `'Exp2'`, respectively.

Adding Experiments to an Existing iddata Object

You can add experiments individually to an `iddata` object as an alternative approach to merging data sets.

For example, to add the experiments in the `iddata` object `dat4` to `data`, use the following syntax:

```
data(:,:,:,'Run4') = dat4
```

This syntax explicitly assigns the experiment name `'Run4'` to the new experiment. The `Experiment` property of the `iddata` object stores experiment names.

For more information about subreferencing experiments in a multiexperiment data set, see “Subreferencing Experiments” on page 2-57.

See Also

More About

- “Select Subsets of Data” on page 2-104
- “Dealing with Multi-Experiment Data and Merging Models” on page 2-64
- “Create Multiexperiment Data Sets in the App” on page 2-35

Dealing with Multi-Experiment Data and Merging Models

This example shows how to deal with multiple experiments and merging models when working with System Identification Toolbox™ for estimating and refining models.

Introduction

The analysis and estimation functions in System Identification Toolbox let you work with multiple batches of data. Essentially, if you have performed multiple experiments and recorded several input-output datasets, you can group them up into a single IDDATA object and use them with any estimation routine.

In some cases, you may want to "split up" your (single) measurement dataset to remove portions where the data quality is not good. For example, portion of data may be unusable due to external disturbance or a sensor failure. In those cases, each good portion of data may be separated out and then combined into a single multi-experiment IDDATA object.

For example, let us look at the dataset iddemo8.mat:

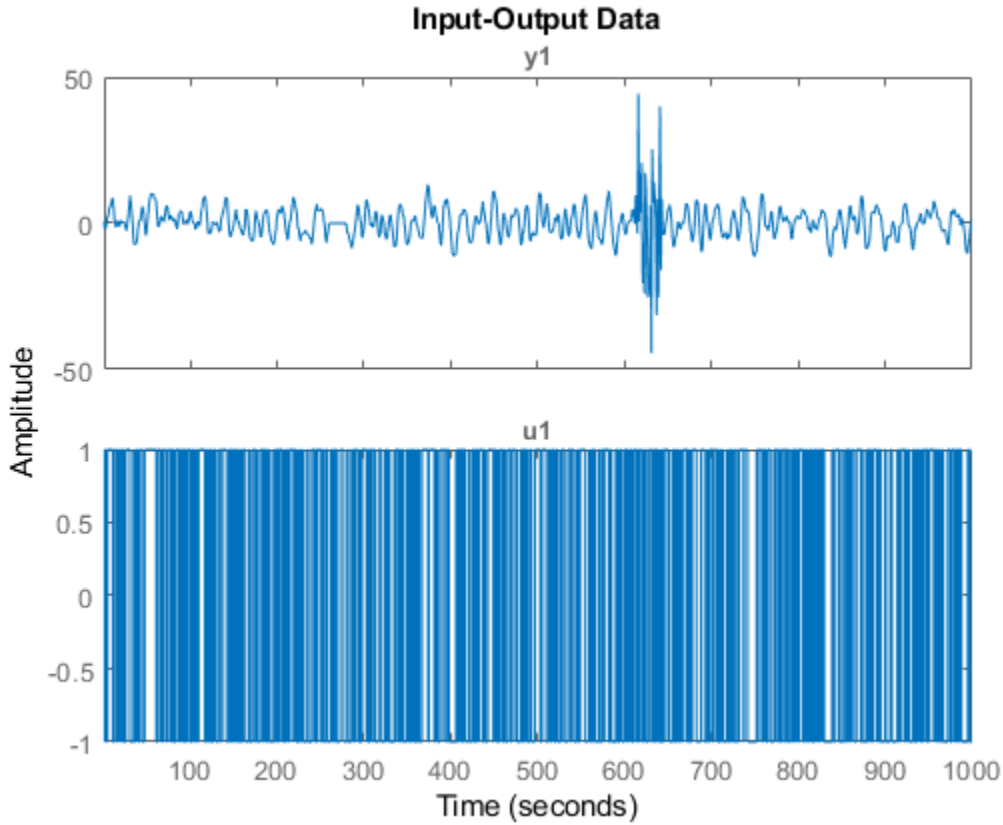
```
load iddemo8
```

The name of the data object is `dat`, and let us view it.

```
dat
```

```
dat =

Time domain data set with 1000 samples.
Sample time: 1 seconds

Outputs      Unit (if specified)
   y1

Inputs       Unit (if specified)
   u1
```

```
plot(dat)
```



We see that there are some problems with the output around sample 250-280 and around samples 600 to 650. These might have been sensor failures.

Therefore split the data into three separate experiments and put then into a multi-experiment data object:

```
d1 = dat(1:250);
d2 = dat(281:600);
d3 = dat(651:1000);
d = merge(d1,d2,d3) % merge lets you create multi-exp IDDATA object

d =

Time domain data set containing 3 experiments.
```

```
Experiment    Samples      Sample Time
   Exp1          250            1
   Exp2          320            1
   Exp3          350            1

Outputs       Unit (if specified)
   y1

Inputs        Unit (if specified)
   u1
```

The different experiments can be given other names, for example:

```
d.exp = {'Period 1';'Day 2';'Phase 3'}

d =

Time domain data set containing 3 experiments.

Experiment      Samples      Sample Time
   Period 1        250            1
   Day 2           320            1
   Phase 3         350            1

Outputs         Unit (if specified)
   y1

Inputs          Unit (if specified)
   u1
```

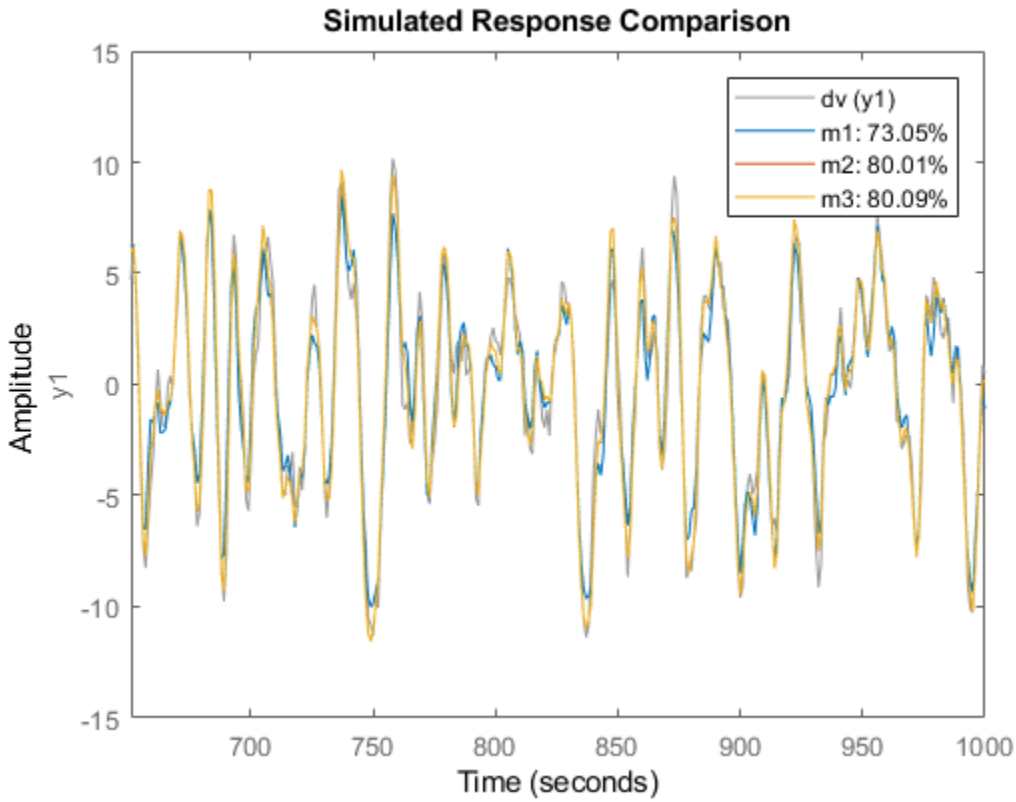
To examine it, use plot, as in `plot(d)`.

Performing Estimation Using Multi-Experiment Data

As mentioned before, all model estimation routines accept multi-experiment data and take into account that they are recorded at different periods. Let us use the two first experiments for estimation and the third one for validation:

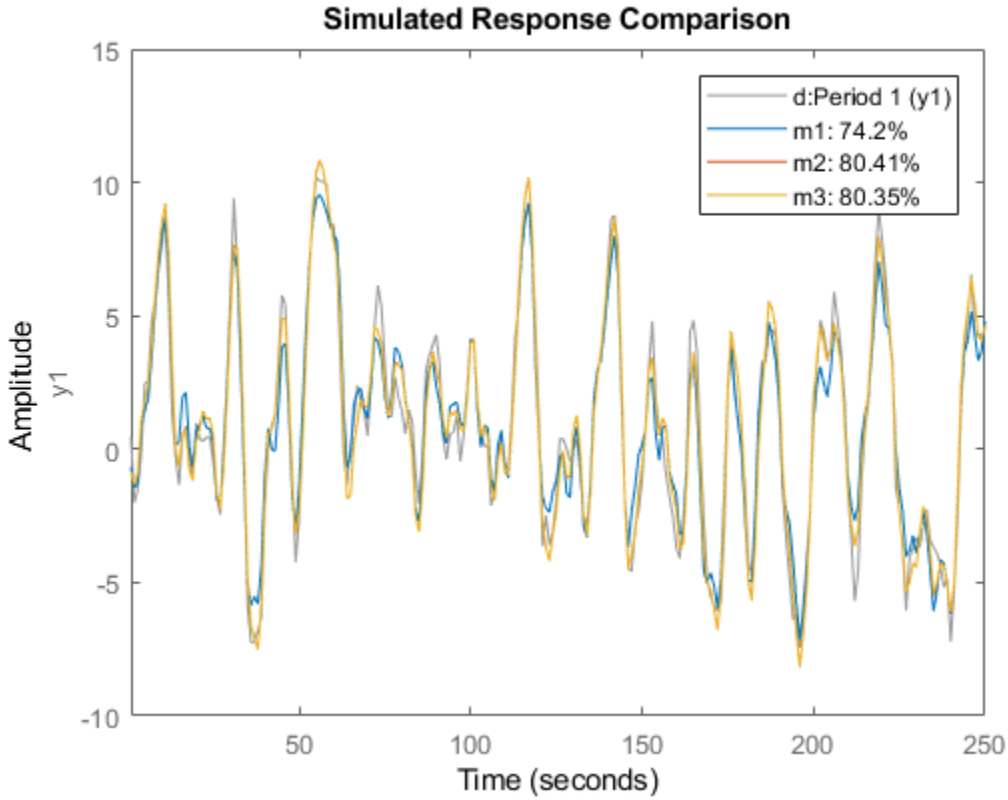
```
de = getexp(d,[1,2]);       % subselection is done using  the command GETEXP
dv = getexp(d,'Phase 3');   % using numbers or names.
m1 = arx(de,[2 2 1]);
m2 = n4sid(de,2);
```

```
m3 = armax(de,[2 2 2 1]);
compare(dv,m1,m2,m3)
```



The `compare` command also accepts multiple experiments. Use the right click menu to pick the experiment to use, one at a time.

```
compare(d,m1,m2,m3)
```



Also, `spa`, `etfe`, `resid`, `predict`, `sim` operate in the same way for multi-experiment data, as they do for single experiment data.

Merging Models After Estimation

There is another way to deal with separate data sets: a model can be computed for each set, and then the models can be merged:

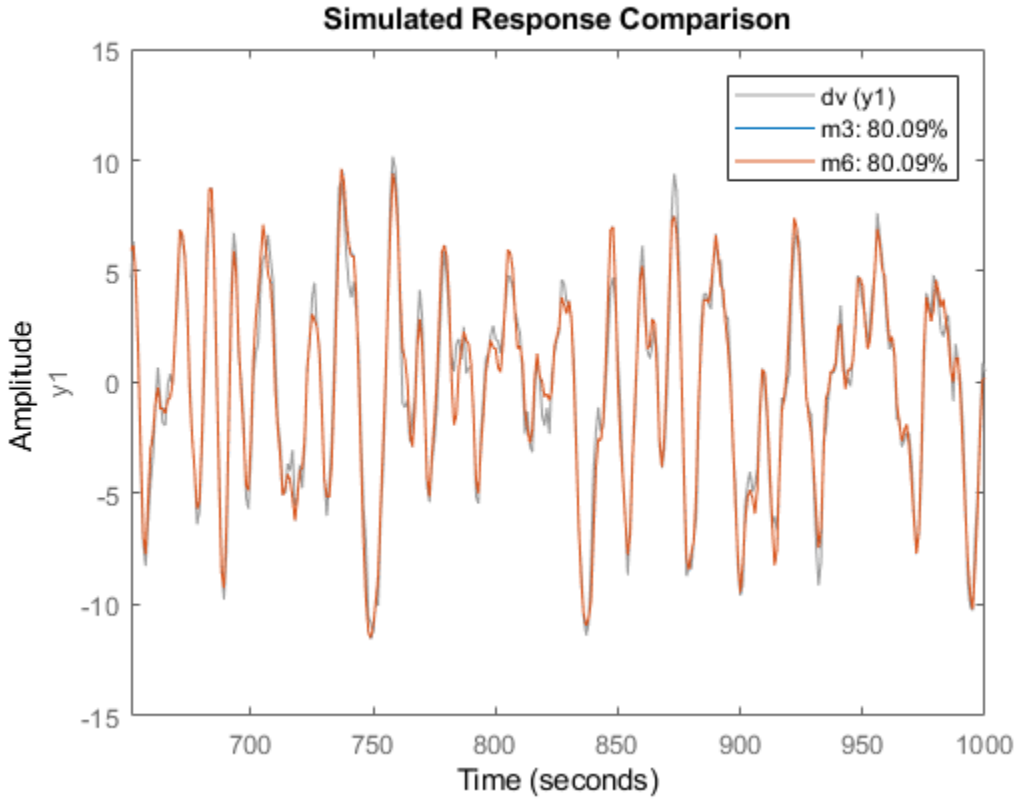
```
m4 = armax(getexp(de,1),[2 2 2 1]);
m5 = armax(getexp(de,2),[2 2 2 1]);
m6 = merge(m4,m5); % m4 and m5 are merged into m6
```

This is conceptually the same as computing `m` from the merged set `de`, but it is not numerically the same. Working on `de` assumes that the signal-to-noise ratios are (about)

the same in the different experiments, while merging separate models makes independent estimates of the noise levels. If the conditions are about the same for the different experiments, it is more efficient to estimate directly on the multi-experiment data.

We can check the models `m3` and `m6` that are both ARMAX models obtained on the same data in two different ways:

```
[m3.a;m6.a]

ans =

    1.0000   -1.5034    0.7008
    1.0000   -1.5022    0.7000


[m3.b;m6.b]

ans =

         0    1.0023    0.5029
         0    1.0035    0.5028


[m3.c;m6.c]

ans =

    1.0000   -0.9744    0.1578
    1.0000   -0.9751    0.1584


compare(dv,m3,m6)
```



Case Study: Concatenating Vs. Merging Independent Datasets

We now turn to another situation. Let us consider two data sets generated by the system m0. The system is given by:

```
m0
```

```
m0 =
  Discrete-time identified state-space model:
    x(t+Ts) = A x(t) + B u(t) + K e(t)
       y(t) = C x(t) + D u(t) + e(t)

  A =
             x1         x2         x3
```

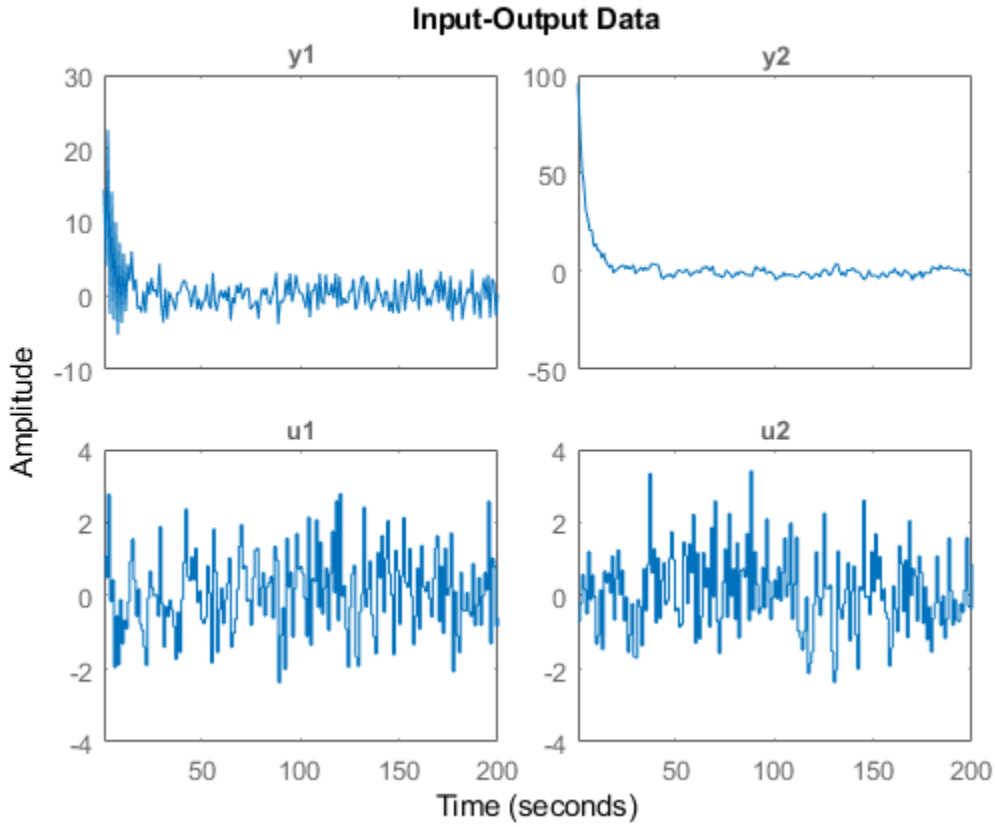
```
   x1    0.5296    -0.476    0.1238
   x2    -0.476  -0.09743    0.1354
   x3    0.1238    0.1354   -0.8233

  B =
              u1         u2
   x1    -1.146  -0.03763
   x2     1.191    0.3273
   x3         0         0

  C =
            x1        x2        x3
   y1  -0.1867  -0.5883  -0.1364
   y2   0.7258         0   0.1139

  D =
          u1      u2
   y1  1.067       0
   y2      0       0

  K =
       y1  y2
   x1   0   0
   x2   0   0
   x3   0   0

Sample time: 1 seconds

Parameterization:
   STRUCTURED form (some fixed coefficients in  A, B, C).
   Feedthrough: on some input channels
   Disturbance component: none
   Number of free coefficients: 23
   Use "idssdata", "getpvec", "getcov" for parameters and their uncertainties.

Status:
Created by direct construction or transformation. Not estimated.
```

The data sets that have been collected are `z1` and `z2`, obtained from m0 with different inputs, noise and initial conditions. These datasets are obtained from iddemo8.mat that was loaded earlier.

```
pause off
```

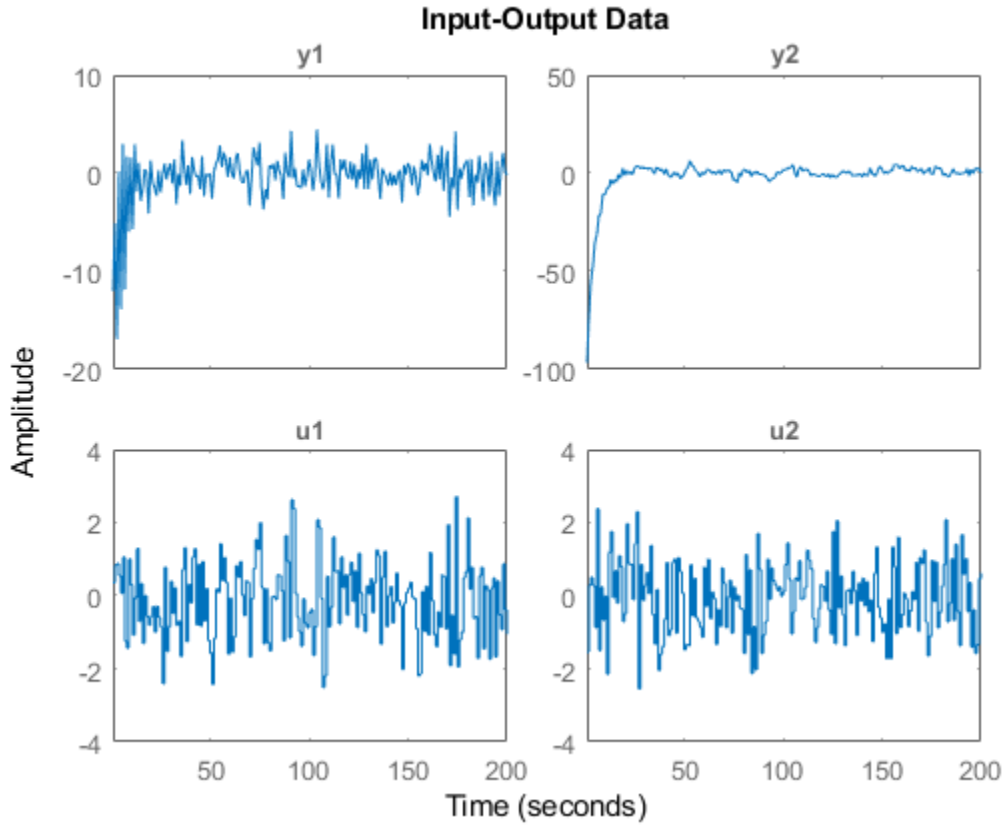
First data set:

```
plot(z1) %generates a separate plot for each I/O pair if pause is on; showing only the
```



The second set:

```
plot(z2) %generates a separate plot for each I/O pair if pause is on; showing only the
```

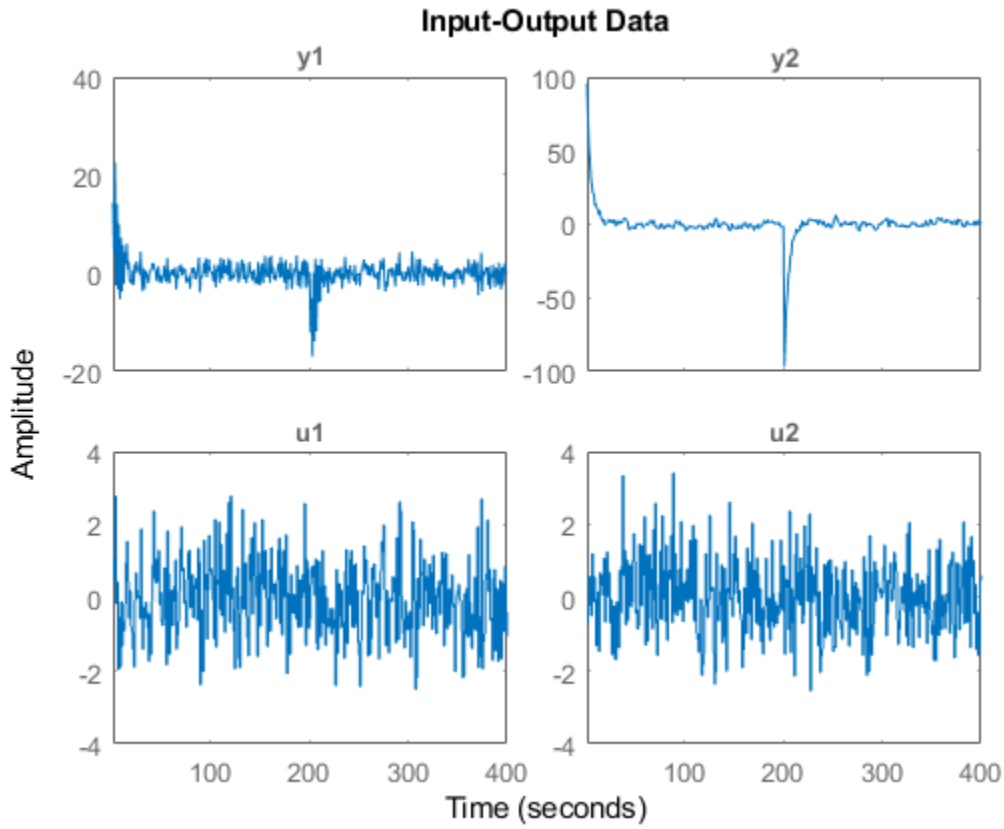


If we just concatenate the data we obtained:

```
zzl = [z1;z2]

zzl =

Time domain data set with 400 samples.
Sample time: 1 seconds

Outputs      Unit (if specified)
   y1
   y2

Inputs       Unit (if specified)
```

```
u1
u2
```

```
plot(zzl)
```



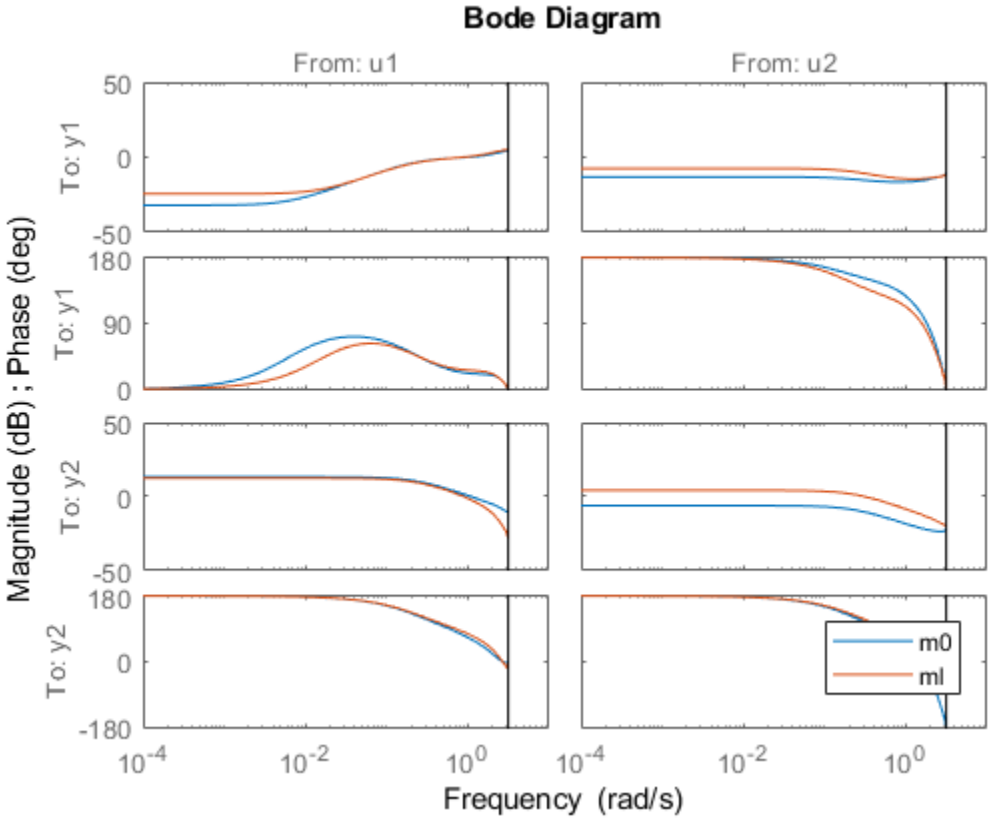

```
pause on
```

A discrete-time state-space model can be obtained by using `ssest`:

```
m1 = ssest(zzl,3,'Ts',1, 'Feedthrough', [true, false]);
```

Compare the bode response for models m0 and m1:

```
clf
bode(m0,m1)
legend('show')
```



This is not a very good model, as observed from the four Bode plots above.

Now, instead treat the two data sets as different experiments:

```
zzm = merge(z1,z2)
```

```
zzm =

Time domain data set containing 2 experiments.

Experiment   Samples      Sample Time
```

```
   Exp1          200            1
   Exp2          200            1

Outputs      Unit (if specified)
   y1
   y2

Inputs       Unit (if specified)
   u1
   u2
```

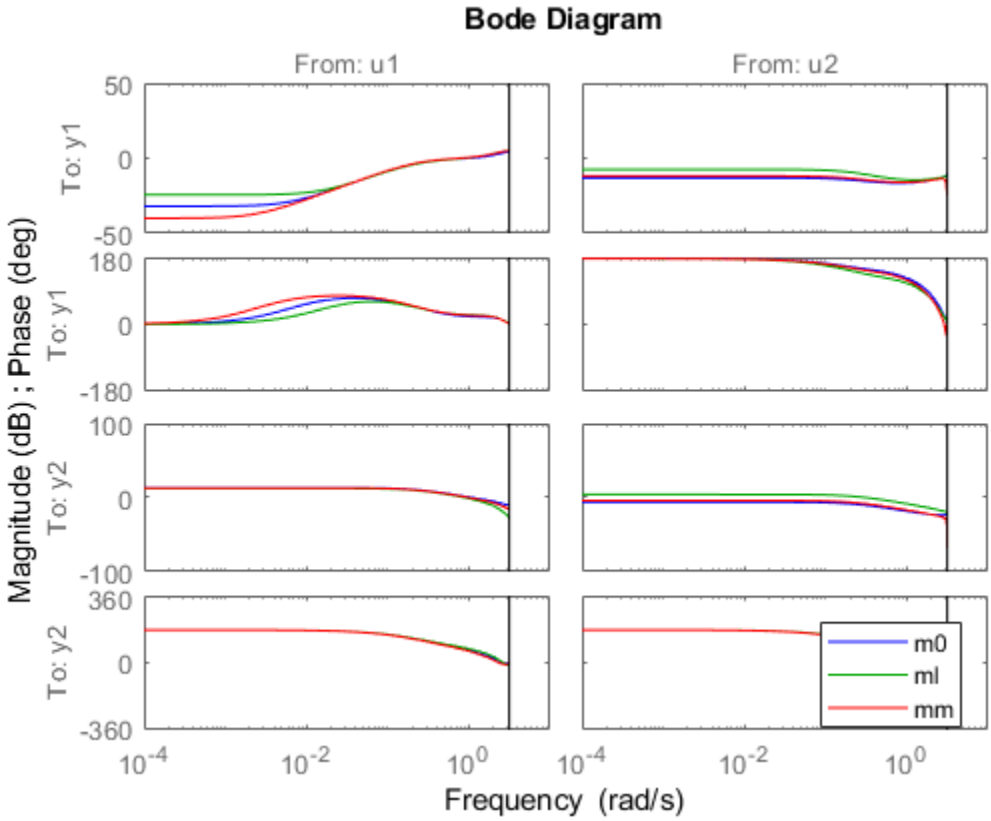
```
% The model for this data can be estimated as before (watching progress this time)
mm = ssest(zzm,3,'Ts',1,'Feedthrough',[true, false], ssestOptions('Display', 'on'));
```

Let us compare the Bode plots of the true system (blue)

the model from concatenated data (green) and the model from the

merged data set (red):

```
clf
bode(m0,'b',m1,'g',mm,'r')
legend('show')
```



The merged data give a better model, as observed from the plot above.

Conclusions

In this example we analyzed how to use multiple data sets together for estimation of one model. This technique is useful when you have multiple datasets from independent experiment runs or when you segment data into multiple sets to remove bad segments. Multiple experiments can be packaged into a single IDDATA object, which is then usable for all estimation and analysis requirements. This technique works for both time and frequency domain iddata.

It is also possible to merge models after estimation. This technique can be used to "average out" independently estimated models. If the noise characteristics on multiple

datasets are different, merging models after estimation works better than merging the datasets themselves before estimation.

Additional Information

For more information on identification of dynamic systems with System Identification Toolbox visit the System Identification Toolbox product information page.

See Also

More About

- “Select Subsets of Data” on page 2-104
- “Create Multiexperiment Data Sets in the App” on page 2-35
- “Create Multiexperiment Data at the Command Line” on page 2-61

Managing iddata Objects

In this section...
"Modifying Time and Frequency Vectors" on page 2-79
"Naming, Adding, and Removing Data Channels" on page 2-82
"Subreferencing iddata Objects" on page 2-84
"Concatenating iddata Objects" on page 2-84

Modifying Time and Frequency Vectors

The `iddata` object stores time-domain data or frequency-domain data and has several properties that specify the time or frequency values. To modify the time or frequency values, you must change the corresponding property values.

Note You can modify the property `SamplingInstants` by setting it to a new vector with the length equal to the number of data samples. For more information, see "Constructing an iddata Object for Time-Domain Data" on page 2-50.

The following tables summarize time-vector and frequency-vector properties, respectively, and provides usage examples. In each example, `data` is an `iddata` object.

Note Property names are not case sensitive. You do not need to type the entire property name if the first few letters uniquely identify the property.

iddata Time-Vector Properties

Property	Description	Syntax Example

`Ts`	Sample time.
	• For a single experiment, `Ts` is a scalar value.
	• For multiexperiment data with `Ne` experiments, `Ts` is a `1`-by-`Ne` cell array, and each cell contains the sample time of the corresponding experiment.
		To set the sample time to `0.05`:
		`set(data,'ts',0.05)`
		or
		`data.ts = 0.05`

`Tstart`	Starting time of the experiment. • For a single experiment, `Ts` is a scalar value. • For multiexperiment data with `Ne` experiments, `Ts` is a `1`-by-`Ne` cell array, and each cell contains the sample time of the corresponding experiment.	To change starting time of the first data sample to `24`: `data.Tstart = 24` Time units are set by the property `TimeUnit`.

`SamplingInstants`	Time values in the time vector, computed from the properties `Tstart` and `Ts`.
	• For a single experiment, `SamplingInstants` is an `N`-by-`1` vector.
	• For multiexperiment data with `Ne` experiments, this property is a `1`-by-`Ne` cell array, and each cell contains the sampling instants of the corresponding experiment.
		To retrieve the time vector for `iddata` object `data`, use:
		`get(data,'sa')`
		To plot the input data as a function of time:
		`plot(data.sa,data.u)`
		Note `sa` is the first two letters of the `SamplingInstants` property that uniquely identifies this property.

Property	Description	Syntax Example
`TimeUnit`	Unit of time. Specify as one of the following: `'nanoseconds'`, `'microseconds'`, `'milliseconds'`, `'seconds'`, `'minutes'`, `'hours'`, `'days'`, `'weeks'`, `'months'`, and `'years'`.	To change the unit of the time vector to `milliseconds`: `data.ti = 'milliseconds'`

iddata Frequency-Vector Properties

Property	Description	Syntax Example
`Frequency`	Frequency values at which the Fourier transforms of the signals are defined. • For a single experiment, `Frequency` is a scalar value. • For multiexperiment data with `Ne` experiments, `Frequency` is a `1`-by-`Ne` cell array, and each cell contains the frequencies of the corresponding experiment.	To specify `100` frequency values in log space, ranging between `0.1` and `100`, use the following syntax: `data.freq = logspace(-1,2,100)`
`FrequencyUnit`	Unit of Frequency. Specify as one of the following: be one of the following: `'rad/TimeUnit'`, `'cycles/TimeUnit'`, `'rad/s'`, `'Hz'`, `'kHz'`, `'MHz'`, `'GHz'`, and, `'rpm'`. Default: `'rad/TimeUnit'` For multi-experiment data with `Ne` experiments, `Units` is a `1`-by-`Ne` cell array, and each cell contains the frequency unit for each experiment.	Set the frequency unit to Hz: `data.FrequencyUnit = 'Hz'` Note that changing the frequency unit does not scale the frequency vector. For a proper translation of units, use `chgFreqUnit`.

Naming, Adding, and Removing Data Channels

- "What Are Input and Output Channels?" on page 2-83
- "Naming Channels" on page 2-83
- "Adding Channels" on page 2-83

- "Modifying Channel Data" on page 2-84

What Are Input and Output Channels?

A multivariate system might contain several input variables or several output variables, or both. When an input or output signal includes several measured variables, these variables are called *channels*.

Naming Channels

The `iddata` properties `InputName` and `OutputName` store the channel names for the input and output signals. When you plot the data, you use channel names to select the variable displayed on the plot. If you have multivariate data, it is helpful to assign a name to each channel that describes the measured variable. For more information about selecting channels on a plot, see "Selecting Measured and Noise Channels in Plots" on page 21-14.

You can use the `set` command to specify the names of individual channels. For example, suppose `data` contains two input channels (voltage and current) and one output channel (temperature). To set these channel names, use the following syntax:

```
set(data,'InputName',{'Voltage','Current'},
         'OutputName','Temperature')
```

Tip You can also specify channel names as follows:

```
data.una = {'Voltage','Current')
data.yna = 'Temperature'
```

`una` is equivalent to the property `InputName`, and `yna` is equivalent to `OutputName`.

If you do not specify channel names when you create the `iddata` object, the toolbox assigns default names. By default, the output channels are named `'y1','y2',...,'yn'`, and the input channels are named `'u1','u2',...,'un'`.

Adding Channels

You can add data channels to an `iddata` object.

For example, consider an `iddata` object named `data` that contains an input signal with four channels. To add a fifth input channel, stored as the vector `Input5`, use the following syntax:

```
data.u(:,5) = Input5;
```

`Input5` must have the same number of rows as the other input channels. In this example, `data.u(:,5)` references all samples as (indicated by `:`) of the input signal `u` and sets the values of the fifth channel. This channel is created when assigning its value to `Input5`.

You can also combine input channels and output channels of several `iddata` objects into one `iddata` object using concatenation. For more information, see "Increasing Number of Channels or Data Points of iddata Objects" on page 2-58.

Modifying Channel Data

After you create an `iddata` object, you can modify or remove specific input and output channels, if needed. You can accomplish this by subreferencing the input and output matrices and assigning new values.

For example, suppose the `iddata` object `data` contains three output channels (named `y1`, `y2`, and `y3`), and four input channels (named `u1`, `u2`, `u3`, and `u4`). To replace `data` such that it only contains samples in `y3`, `u1`, and `u4`, type the following at the prompt:

```
data = data(:,3,[1 4])
```

The resulting data object contains one output channel and two input channels.

Subreferencing iddata Objects

See "Select Data Channels, I/O Data and Experiments in iddata Objects" on page 2-55.

Concatenating iddata Objects

See "Increasing Number of Channels or Data Points of iddata Objects" on page 2-58.

Representing Frequency-Response Data Using idfrd Objects

In this section...
"idfrd Constructor" on page 2-85 "idfrd Properties" on page 2-86 "Select I/O Channels and Data in idfrd Objects" on page 2-87 "Adding Input or Output Channels in idfrd Objects" on page 2-88 "Managing idfrd Objects" on page 2-90 "Operations that Create idfrd Objects" on page 2-90

idfrd Constructor

The `idfrd` represents complex frequency-response data. Before you can create an `idfrd` object, you must import your data as described in "Frequency-Response Data Representation" on page 2-13.

Note The `idfrd` object can only encapsulate one frequency-response data set. It does not support the `iddata` equivalent of multiexperiment data.

Use the following syntax to create the data object `fr_data`:

```
fr_data = idfrd(response,f,Ts)
```

Suppose that `ny` is the number of output channels, `nu` is the number of input channels, and `nf` is a vector of frequency values. `response` is an `ny`-by-`nu`-by-`nf` 3-D array. `f` is the frequency vector that contains the frequencies of the response.`Ts` is the sample time, which is used when measuring or computing the frequency response. If you are working with a continuous-time system, set `Ts` to `0`.

`response(ky,ku,kf)`, where `ky`, `ku`, and `kf` reference the `k`th output, input, and frequency value, respectively, is interpreted as the complex-valued frequency response from input `ku` to output `ky` at frequency `f(kf)`.

Note When you work at the command line, you can only create `idfrd` objects from complex values of $G(e^{iw})$. For a SISO system, `response` can be a vector.

You can specify object properties when you create the `idfrd` object using the constructor syntax:

```
fr_data = idfrd(response,f,Ts,
               'Property1',Value1,...,'PropertyN',ValueN)
```

idfrd Properties

To view the properties of the `idfrd` object, you can use the `get` command. The following example shows how to create an `idfrd` object that contains 100 frequency-response values with a sample time of 0.1 s and get its properties:

```
f = logspace(-1,1,100);
[mag, phase] = bode(idtf([1 .2],[1 2 1 1]),f);
response = mag.*exp(1j*phase*pi/180);
fr_data = idfrd(response,f,0.1);
get(fr_data)
```

```
      FrequencyUnit: 'rad/TimeUnit'
             Report: [1x1 idresults.frdest]
       SpectrumData: []
     CovarianceData: []
    NoiseCovariance: []
        InterSample: {'zoh'}
       ResponseData: [1x1x100 double]
            IODelay: 0
         InputDelay: 0
        OutputDelay: 0
                 Ts: 0.1000
           TimeUnit: 'seconds'
          InputName: {''}
          InputUnit: {''}
         InputGroup: [1x1 struct]
         OutputName: {''}
         OutputUnit: {''}
        OutputGroup: [1x1 struct]
              Notes: [0x1 string]
           UserData: []
               Name: ''
       SamplingGrid: [1x1 struct]
          Frequency: [100x1 double]
```

For a complete description of all `idfrd` object properties, see the `idfrd` reference page.

To change property values for an existing `idfrd` object, use the `set` command or dot notation. For example, to change the name of the `idfrd` object, type the following command sequence at the prompt:

```
fr_data.Name = 'DC_Converter';
```

Select I/O Channels and Data in idfrd Objects

You can reference specific data values in the `idfrd` object using the following syntax:

```
fr_data(outputchannels,inputchannels)
```

Reference specific channels by name or by channel index.

Tip Use a colon (`:`) to specify all channels, and use the empty matrix (`[]`) to specify no channels.

For example, the following command references frequency-response data from input channel `3` to output channel `2`:

```
fr_data(2,3)
```

You can also access the data in specific channels using channel names. To list multiple channel names, use a cell array. For example, to retrieve the power output, and the voltage and speed inputs, use the following syntax:

```
fr_data('power',{'voltage','speed'})
```

To retrieve only the responses corresponding to frequency values between `200` and `300`, use the following command:

```
fr_data_sub = fselect(fr_data,[200:300])
```

You can also use logical expressions to subreference data. For example, to retrieve all frequency-response values between frequencies `1.27` and `9.3` in the `idfrd` object `fr_data`, use the following syntax:

```
fr_data_sub = fselect(fr_data,fr_data.f>1.27&fr_data.f<9.3)
```

Tip Use `end` to reference the last sample number in the data. For example, `data(77:end)`.

Note You do not need to type the entire property name. In this example, `f` in `fr_data.f` uniquely identifies the `Frequency` property of the `idfrd` object.

Adding Input or Output Channels in idfrd Objects

About Concatenating idfrd Objects

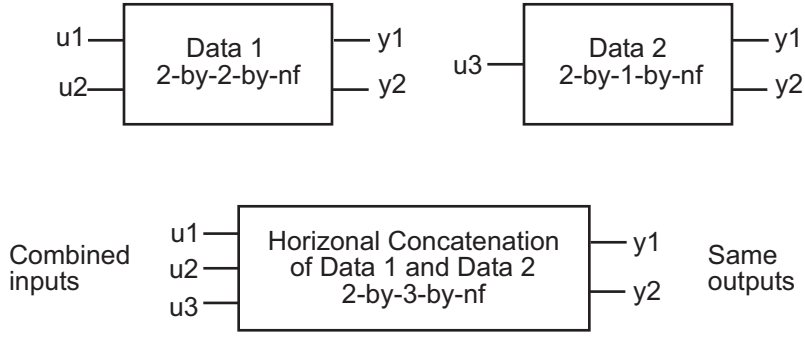
The horizontal and vertical concatenation of `idfrd` objects combine information in the `ResponseData` properties of these objects. `ResponseData` is an `ny`-by-`nu`-by-`nf` array that stores the response of the system, where `ny` is the number of output channels, `nu` is the number of input channels, and `nf` is a vector of frequency values (see "Properties").

Horizontal Concatenation of idfrd Objects

The following syntax creates a new `idfrd` object `data` that contains the horizontal concatenation of `data1,data2,...,dataN`:

```
data = [data1,data2,...,dataN]
```

`data` contains the frequency responses from all of the inputs in `data1,data2,...,dataN` to the same outputs. The following diagram is a graphical representation of horizontal concatenation of frequency-response data. The `(j,i,:)` vector of the resulting response data represents the frequency response from the `i`th input to the `j`th output at all frequencies.



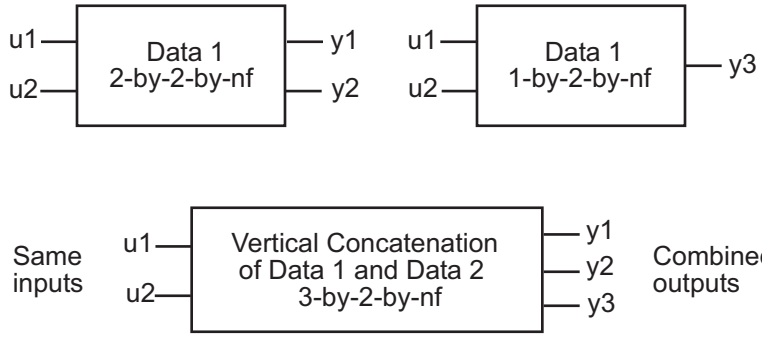
Note Horizontal concatenation of `idfrd` objects requires that they have the same outputs and frequency vectors. If the output channel names are different and their dimensions are the same, the concatenation operation resets the output names to their default values.

Vertical Concatenation of idfrd Objects

The following syntax creates a new `idfrd` object `data` that contains the vertical concatenation of `data1,data2,...,dataN`:

```
data = [data1;data2;... ;dataN]
```

The resulting `idfrd` object `data` contains the frequency responses from the same inputs in `data1,data2,...,dataN` to all the outputs. The following diagram is a graphical representation of vertical concatenation of frequency-response data. The `(j,i,:)` vector of the resulting response data represents the frequency response from the `i`th input to the `j`th output at all frequencies.



Note Vertical concatenation of `idfrd` objects requires that they have the same inputs and frequency vectors. If the input channel names are different and their dimensions are the same, the concatenation operation resets the input names to their default values.

Concatenating Noise Spectrum Data of idfrd Objects

When the `SpectrumData` property of individual `idfrd` objects is not empty, horizontal and vertical concatenation handle `SpectrumData`, as follows.

In case of horizontal concatenation, there is no meaningful way to combine the `SpectrumData` of individual `idfrd` objects and the resulting `SpectrumData` property is empty. An empty property results because each `idfrd` object has its own set of noise channels, where the number of noise channels equals the number of outputs. When the resulting `idfrd` object contains the same output channels as each of the individual `idfrd` objects, it cannot accommodate the noise data from all the `idfrd` objects.

In case of vertical concatenation, the toolbox concatenates individual noise models diagonally. The following shows that `data.SpectrumData` is a block diagonal matrix of the power spectra and cross spectra of the output noise in the system:

$$data.s = \begin{pmatrix} data1.s & & 0 \\ & \ddots & \\ 0 & & dataN.s \end{pmatrix}$$

`s` in `data.s` is the abbreviation for the `SpectrumData` property name.

Managing idfrd Objects

Subreferencing idfrd Objects

See “Select I/O Channels and Data in idfrd Objects” on page 2-87.

Concatenating idfrd Objects

See “Adding Input or Output Channels in idfrd Objects” on page 2-88.

Operations that Create idfrd Objects

The following operations create `idfrd` objects:

- Constructing `idfrd` objects.
- Estimating nonparametric models using `etfe`, `spa`, and `spafdr`. For more information, see “Frequency-Response Models”.
- Converting the Control System Toolbox `frd` object. For more information, see “Using Identified Models for Control Design Applications” on page 19-2.
- Converting any linear dynamic system using the `idfrd` command.

  For example:

```
sys_idpoly = idpoly([1 2 1],[0 2],'Ts',1);
G = idfrd(sys_idpoly,linspace(0,pi,128))

G =
IDFRD model.
Contains Frequency Response Data for 1 output(s) and 1 input(s), and the spectra for
Response data and disturbance spectra are available at 128 frequency points, ranging

Sample time: 1 seconds
Status:
Created by direct construction or transformation. Not estimated.
```

Is Your Data Ready for Modeling?

Before you start estimating models from data, you should check your data for the presence of any undesirable characteristics. For example, you might plot the data to identify drifts and outliers. You plot analysis might lead you to preprocess your data before model estimation.

The following data plots are available in the toolbox:

- Time plot — Shows data values as a function of time.

  Tip You can infer time delays from time plots, which are required inputs to most parametric models. A *time delay* is the time interval between the change in input and the corresponding change in output.

- Spectral plot — Shows a *periodogram* that is computed by taking the absolute squares of the Fourier transforms of the data, dividing by the number of data points, and multiplying by the sample time.
- Frequency-response plot — For frequency-response data, shows the amplitude and phase of the frequency-response function on a Bode plot. For time- and frequency-domain data, shows the empirical transfer function estimate (see `etfe`) .

See Also

Related Examples

- “How to Analyze Data Using the advice Command” on page 2-102
- “How to Plot Data in the App” on page 2-93
- “How to Plot Data at the Command Line” on page 2-100

More About

- “Ways to Prepare Data for System Identification” on page 2-6

How to Plot Data in the App

In this section...
“How to Plot Data in the App” on page 2-93
“Manipulating a Time Plot” on page 2-95
“Manipulating Data Spectra Plot” on page 2-96
“Manipulating a Frequency Function Plot” on page 2-98

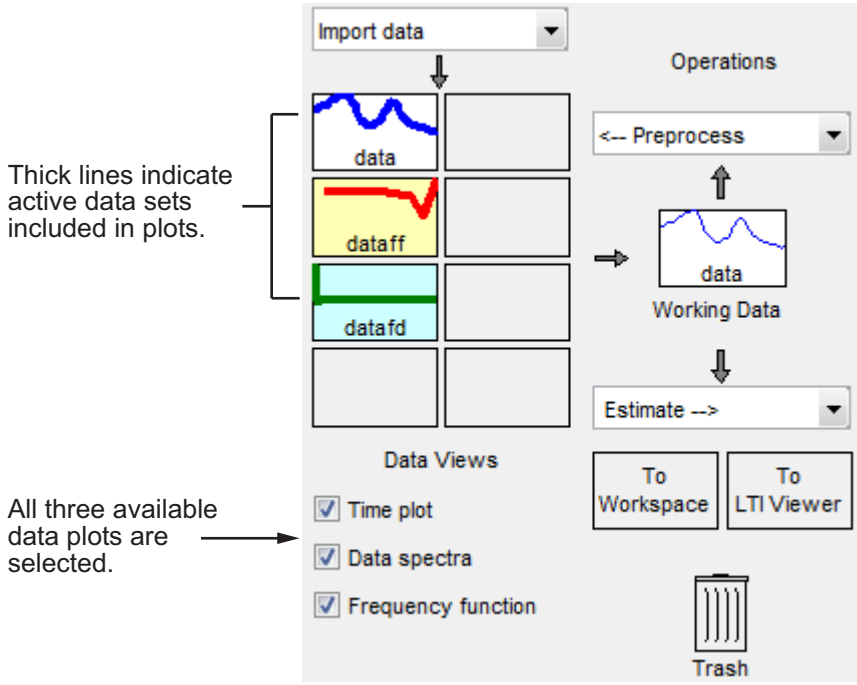
How to Plot Data in the App

After importing data into the System Identification app, as described in “Represent Data”, you can plot the data.

To create one or more plots, select the corresponding check box in the **Data Views** area of the System Identification app.

An *active* data icon has a thick line in the icon, while an *inactive* data set has a thin line. Only active data sets appear on the selected plots. To toggle including and excluding data on a plot, click the corresponding icon in the System Identification app. Clicking the data icon updates any plots that are currently open.

When you have several data sets, you can view different input-output channel pair by selecting that pair from the **Channel** menu. For more information about selecting different input and output pairs, see “Selecting Measured and Noise Channels in Plots” on page 21-14.



In this example, `data` and `dataff` are active and appear on the three selected plots.

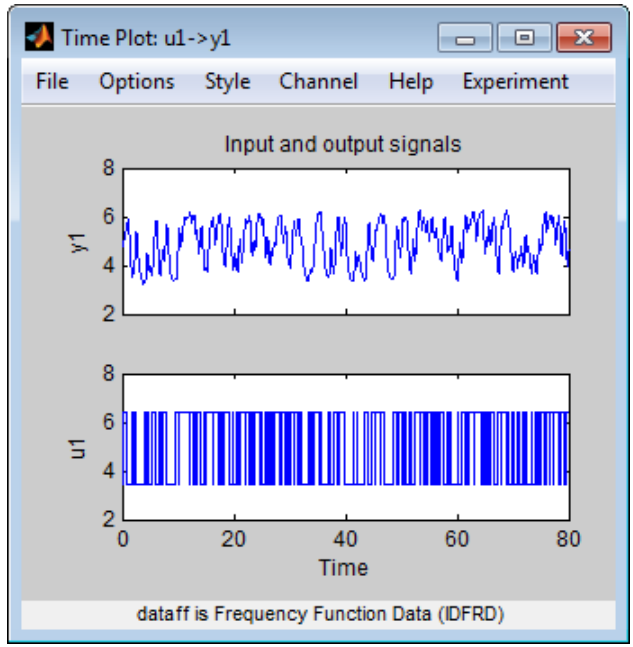
To close a plot, clear the corresponding check box in the System Identification app.

Tip To get information about working with a specific plot, select a help topic from the **Help** menu in the plot window.

The plots you create using the System Identification app provide options that are specific to the System Identification Toolbox product, such as selecting specific channel pairs in a multivariate signals or converting frequency units between Hertz and radians per second.

Manipulating a Time Plot

The **Time plot** only shows time-domain data. In this example, `data1` is displayed on the time plot because, of the three data sets, it is the only one that contains time-domain input and output.



Time Plot of data1

The following table summarizes options that are specific to time plots, which you can select from the plot window menus. For general information about working with System Identification Toolbox plots, see “Working with Plots” on page 21-11.

Time Plot Options

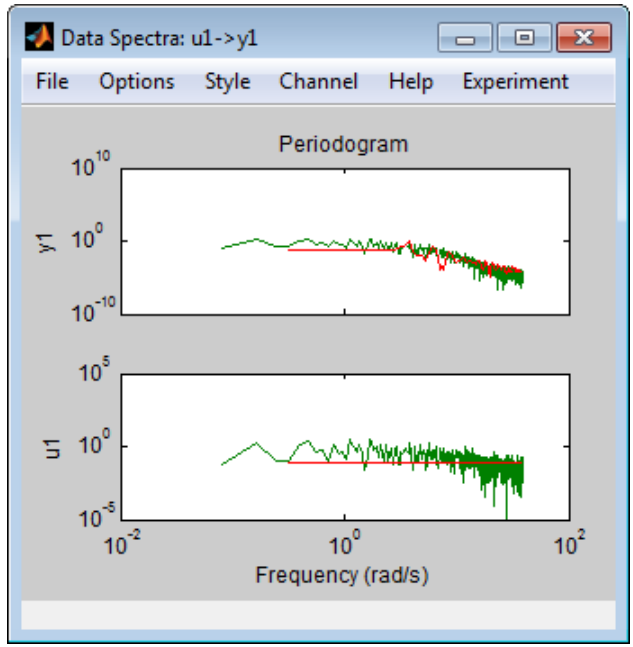
Action	Command
Toggle input display between piece-wise continuous (zero-order hold) and linear interpolation (first-order hold) between samples. **Note** This option only affects the display and not the intersample behavior specified when importing the data.	Select **Style** > **Staircase input** for zero-order hold or **Style** > **Regular input** for first-order hold.

Manipulating Data Spectra Plot

The **Data spectra** plot shows a periodogram or a spectral estimate of `data1` and `data3fd`.

The periodogram is computed by taking the absolute squares of the Fourier transforms of the data, dividing by the number of data points, and multiplying by the sample time. The spectral estimate for time-domain data is a smoothed spectrum calculated using `spa`. For frequency-domain data, the **Data spectra** plot shows the square of the absolute value of the actual data, normalized by the sample time.

The top axes show the input and the bottom axes show the output. The vertical axis of each plot is labeled with the corresponding channel name.



Periodograms of data1 and data3fd

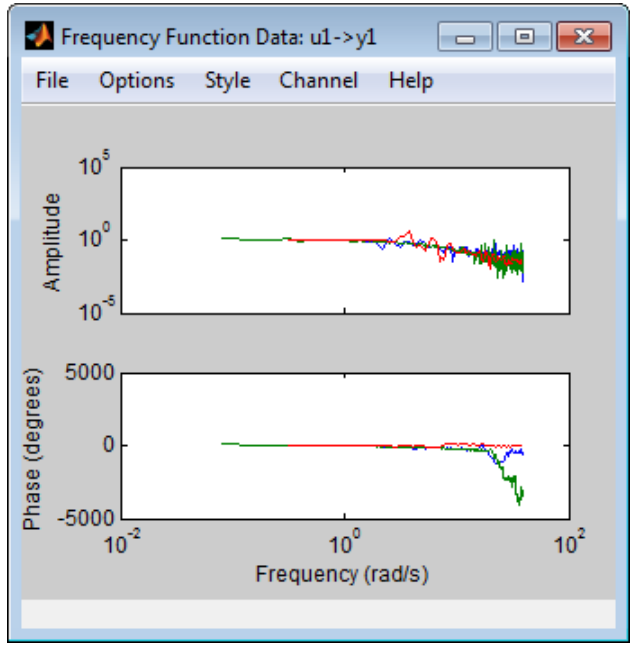
Data Spectra Plot Options

Action	Command
Toggle display between periodogram and spectral estimate.	Select **Options** > **Periodogram** or **Options** > **Spectral analysis**.
Change frequency units.	Select **Style** > **Frequency (rad/s)** or **Style** > **Frequency (Hz)**.
Toggle frequency scale between linear and logarithmic.	Select **Style** > **Linear frequency scale** or **Style** > **Log frequency scale**.
Toggle amplitude scale between linear and logarithmic.	Select **Style** > **Linear amplitude scale** or **Style** > **Log amplitude scale**.

Manipulating a Frequency Function Plot

For time-domain data, the **Frequency function** plot shows the empirical transfer function estimate (`etfe`). For frequency-domain data, the plot shows the ratio of output to input data.

The frequency-response plot shows the amplitude and phase plots of the corresponding frequency response. For more information about frequency-response data, see “Frequency-Response Data Representation” on page 2-13.



Frequency Functions of data1 and data3fd

Frequency Function Plot Options

Action	Command
Change frequency units.	Select **Style** > **Frequency (rad/s)** or **Style** > **Frequency (Hz)**.
Toggle frequency scale between linear and logarithmic.	Select **Style** > **Linear frequency scale** or **Style** > **Log frequency scale**.
Toggle amplitude scale between linear and logarithmic.	Select **Style** > **Linear amplitude scale** or **Style** > **Log amplitude scale**.

See Also

Related Examples

- “How to Plot Data at the Command Line” on page 2-100

How to Plot Data at the Command Line

The following table summarizes the commands available for plotting time-domain, frequency-domain, and frequency-response data.

Commands for Plotting Data

Command	Description	Example

`bode`, `bodeplot`	For frequency-response data only. Shows the magnitude and phase of the frequency response on a logarithmic frequency scale of a Bode plot.	To plot `idfrd` data:
		`bode(idfrd_data)`
		or:
		`bodeplot(idfrd_data)`

`plot`	The type of plot corresponds to the type of data. For example, plotting time-domain data generates a time plot, and plotting frequency-response data generates a frequency-response plot. When plotting time- or frequency-domain inputs and outputs, the top axes show the output and the bottom axes show the input.	To plot `iddata` or `idfrd` data: `plot(data)`

All plot commands display the data in the standard MATLAB Figure window, which provides options for formatting, saving, printing, and exporting plots to a variety of file formats.

To plot portions of the data, you can subreference specific samples (see “Select Data Channels, I/O Data and Experiments in iddata Objects” on page 2-55 and “Select I/O Channels and Data in idfrd Objects” on page 2-87. For example:

```
plot(data(1:300))
```

For time-domain data, to plot only the input data as a function of time, use the following syntax:

```
plot(data(:,[],:)
```

When `data.intersample = 'zoh'`, the input is piece-wise constant between sampling points on the plot. For more information about properties, see the `iddata` reference page.

You can generate plots of the input data in the time domain using:

```
plot(data.SamplingInstants,data.u)
```

To plot frequency-domain data, you can use the following syntax:

```
semilogx(data.Frequency,abs(data.u))
```

When you specify to plot a multivariable `iddata` object, each input-output combination is displayed one at a time in the same MATLAB Figure window. You must press **Enter** to update the Figure window and view the next channel combination. To cancel the plotting operation, press **Ctrl+C**.

Tip To plot specific input and output channels, use `plot(data(:,ky,ku))`, where `ky` and `ku` are specific output and input channel indexes or names. For more information about subreferencing channels, see "Subreferencing Data Channels" on page 2-56.

To plot several `iddata` sets `d1,...,dN`, use `plot(d1,...,dN)`. Input-output channels with the same experiment name, input name, and output name are always plotted in the same plot.

See Also

Related Examples

- "How to Plot Data in the App" on page 2-93

How to Analyze Data Using the advice Command

You can use the `advice` command to analyze time- or frequency- domain data before estimating a model. The resulting report informs you about the possible need to preprocess the data and identifies potential restrictions on the model accuracy. You should use these recommendations in combination with plotting the data and validating the models estimated from this data.

Note `advice` does not support frequency-response data.

Before applying the `advice` command to your data, you must have represented your data as an `iddata` object. For more information, see "Representing Time- and Frequency-Domain Data Using iddata Objects" on page 2-50.

If you are using the System Identification app, you must export your data to the MATLAB workspace before you can use the `advice` command on this data. For more information about exporting data, see "Exporting Models from the App to the MATLAB Workspace" on page 21-10.

Use the following syntax to get advice about an `iddata` object `data`:

```
advice(data)
```

For more information about the `advice` syntax, see the `advice` reference page.

Advice provide guidance for these kinds of questions:

- Does it make sense to remove constant offsets and linear trends from the data?
- What are the excitation levels of the signals and how does this affects the model orders?
- Is there an indication of output feedback in the data? When feedback is present in the system, only prediction-error methods work well for estimating closed-loop data.
- Is there an indication of nonlinearity in the process that generated the data?

See Also

`advice` | `delayest` | `detrend` | `feedback` | `pexcit`

Related Examples

- “How to Plot Data in the App” on page 2-93
- “How to Plot Data at the Command Line” on page 2-100

Select Subsets of Data

In this section...
"Why Select Subsets of Data?" on page 2-104 "Extract Subsets of Data Using the App" on page 2-104 "Extract Subsets of Data at the Command Line" on page 2-106

Why Select Subsets of Data?

You can use data selection to create independent data sets for estimation and validation.

You can also use data selection as a way to clean the data and exclude parts with noisy or missing information. For example, when your data contains missing values, outliers, level changes, and disturbances, you can select one or more portions of the data that are suitable for identification and exclude the rest.

If you only have one data set and you want to estimate linear models, you should split the data into two portions to create two independent data sets for estimation and validation, respectively. Splitting the data is selecting parts of the data set and saving each part independently.

You can merge several data segments into a single multiexperiment data set and identify an average model. For more information, see "Create Data Sets from a Subset of Signal Channels" on page 2-33 or "Representing Time- and Frequency-Domain Data Using iddata Objects" on page 2-50.

Note Subsets of the data set must contain enough samples to adequately represent the system, and the inputs must provide suitable excitation to the system.

Selecting potions of frequency-domain data is equivalent to filtering the data. For more information about filtering, see "Filtering Data" on page 2-129.

Extract Subsets of Data Using the App

- "Ways to Select Data in the App" on page 2-105
- "Selecting a Range for Time-Domain Data" on page 2-105

- "Selecting a Range of Frequency-Domain Data" on page 2-106

Ways to Select Data in the App

You can use System Identification app to select ranges of data on a time-domain or frequency-domain plot. Selecting data in the frequency domain is equivalent to passband-filtering the data.

After you select portions of the data, you can specify to use one data segment for estimating models and use the other data segment for validating models. For more information, see "Specify Estimation and Validation Data in the App" on page 2-30.

Note Selecting **<--Preprocess** > **Quick start** performs the following actions simultaneously:

- Remove the mean value from each channel.
- Split the data into two parts.
- Specify the first part as estimation data (or **Working Data**).
- Specify the second part as **Validation Data**.

Selecting a Range for Time-Domain Data

You can select a range of data values on a time plot and save it as a new data set in the System Identification app.

Note Selecting data does not extract experiments from a data set containing multiple experiments. For more information about multiexperiment data, see "Create Multiexperiment Data Sets in the App" on page 2-35.

To extract a subset of time-domain data and save it as a new data set:

1. Import time-domain data into the System Identification app, as described in "Create Data Sets from a Subset of Signal Channels" on page 2-33.
2. Drag the data set you want to subset to the **Working Data** area.
3. If your data contains multiple I/O channels, in the **Channel** menu, select the channel pair you want to view. The upper plot corresponds to the input signal, and the lower plot corresponds to the output signal.

Although you view only one I/O channel pair at a time, your data selection is applied to all channels in this data set.

4 Select the data of interest in either of the following ways:

- Graphically — Draw a rectangle on either the input-signal or the output-signal plot with the mouse to select the desired time interval. Your selection appears on both plots regardless of the plot on which you draw the rectangle. The **Time span** and **Samples** fields are updated to match the selected region.
- By specifying the **Time span** — Edit the beginning and the end times in seconds. The **Samples** field is updated to match the selected region. For example:

  ```
  28.5 56.8
  ```

- By specifying the **Samples** range — Edit the beginning and the end indices of the sample range. The **Time span** field is updated to match the selected region. For example:

  ```
  342 654
  ```

Note To clear your selection, click **Revert**.

5 In the **Data name** field, enter the name of the data set containing the selected data.

6 Click **Insert**. This action saves the selection as a new data set and adds it to the Data Board.

7 To select another range, repeat steps 4 to 6.

Selecting a Range of Frequency-Domain Data

Selecting a range of values in frequency domain is equivalent to filtering the data. For more information about data filtering, see “Filtering Frequency-Domain or Frequency-Response Data in the App” on page 2-132.

Extract Subsets of Data at the Command Line

Selecting ranges of data values is equivalent to subreferencing the data.

For more information about subreferencing time-domain and frequency-domain data, see “Select Data Channels, I/O Data and Experiments in iddata Objects” on page 2-55.

For more information about subreferencing frequency-response data, see “Select I/O Channels and Data in idfrd Objects” on page 2-87.

See Also

More About

- “Create Multiexperiment Data Sets in the App” on page 2-35
- “Create Multiexperiment Data at the Command Line” on page 2-61

Handling Missing Data and Outliers

In this section...
“Handling Missing Data” on page 2-108
“Handling Outliers” on page 2-109
“See Also” on page 2-110

Handling Missing Data

Data acquisition failures sometimes result in missing measurements both in the input and the output signals. When you import data that contains missing values using the MATLAB Import Wizard, these values are automatically set to `NaN`. `NaN` serves as a flag for nonexistent or undefined data. When you plot data on a time-plot that contains missing values, gaps appear on the plot where missing data exists.

You can use `misdata` to estimate missing values. This command linearly interpolates missing values to estimate the first model. Then, it uses this model to estimate the missing data as parameters by minimizing the output prediction errors obtained from the reconstructed data. You can specify the model structure you want to use in the `misdata` argument or estimate a default-order model using the `n4sid` method. For more information, see the `misdata` reference page.

Note You can only use `misdata` on time-domain data stored in an `iddata` object. For more information about creating `iddata` objects, see “Representing Time- and Frequency-Domain Data Using iddata Objects” on page 2-50.

For example, suppose `y` and `u` are output and input signals that contain `NaN`s. This data is sampled at `0.2` s. The following syntax creates a new `iddata` object with these input and output signals.

```
dat = iddata(y,u,0.2) % y and u contain NaNs
                      % representing missing data
```

Apply the `misdata` command to the new data object. For example:

```
dat1 = misdata(dat);
plot(dat,dat1)        % Check how the missing data
                      % was estimated on a time plot
```

Handling Outliers

Malfunctions can produce errors in measured values, called *outliers*. Such outliers might be caused by signal spikes or by measurement malfunctions. If you do not remove outliers from your data, this can adversely affect the estimated models.

To identify the presence of outliers, perform one of the following tasks:

- Before estimating a model, plot the data on a time plot and identify values that appear out of range.
- After estimating a model, plot the residuals and identify unusually large values. For more information about plotting residuals, see topics on the "Residual Analysis" page. Evaluate the original data that is responsible for large residuals. For example, for the model `Model` and validation data `Data`, you can use the following commands to plot the residuals:

```
% Compute the residuals
  E = resid(Data,Model)
% Plot the residuals
  plot(E)
```

Next, try these techniques for removing or minimizing the effects of outliers:

- Extract the informative data portions into segments and merge them into one multiexperiment data set (see "Extract and Model Specific Data Segments" on page 2-111). For more information about selecting and extracting data segments, see "Select Subsets of Data" on page 2-104.

  Tip The inputs in each of the data segments must be consistently exciting the system. Splitting data into meaningful segments for steady-state data results in minimum information loss. Avoid making data segments too small.

- Manually replace outliers with `NaN`s and then use the `misdata` command to reconstruct flagged data. This approach treats outliers as missing data and is described in "Handling Missing Data" on page 2-108. Use this method when your data contains several inputs and outputs, and when you have difficulty finding reliable data segments in all variables.
- Remove outliers by prefiltering the data for high-frequency content because outliers often result from abrupt changes. For more information about filtering, see "Filtering Data" on page 2-129.

Note The estimation algorithm can handle outliers by assigning a smaller weight to outlier data. A robust error criterion applies an error penalty that is quadratic for small and moderate prediction errors, and is linear for large prediction errors. Because outliers produce large prediction errors, this approach gives a smaller weight to the corresponding data points during model estimation. Set the `ErrorThreshold` estimation option (see `Advanced.ErrorThreshold` in, for example, `polyestOptions`) to a nonzero value to activate the correction for outliers in the estimation algorithm.

See Also

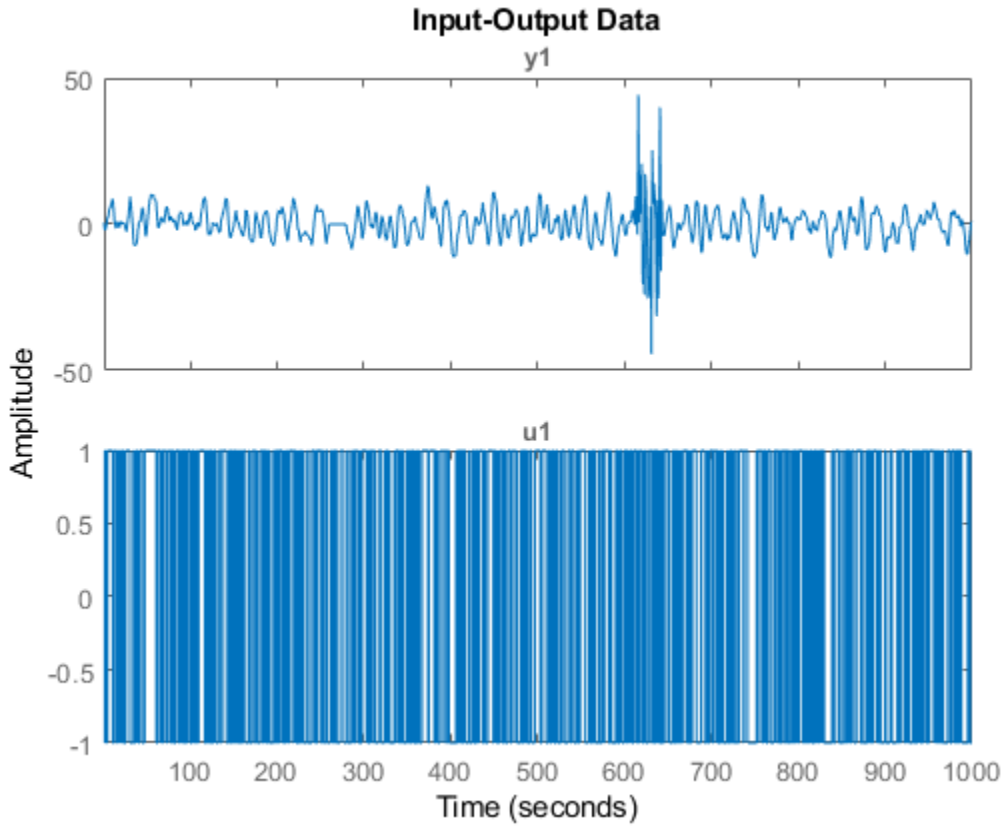
To learn more about the theory of handling missing data and outliers, see the chapter on preprocessing data in *System Identification: Theory for the User*, Second Edition, by Lennart Ljung, Prentice Hall PTR, 1999.

Extract and Model Specific Data Segments

This example shows how to create a multi-experiment, time-domain data set by merging only the accurate data segments and ignoring the rest.

Load and plot the data.

```
load iddemo8;
plot(dat);
```



The data has poor or no measurements from samples 251 to 280 and 601 to 650. You cannot simply concatenate the good data segments because the transients at the

connection points compromise the model. Instead, you must create a multiexperiment `iddata` object, where each experiment corresponds to a good segment of data.

Create multiexperiment data set by merging data segments.

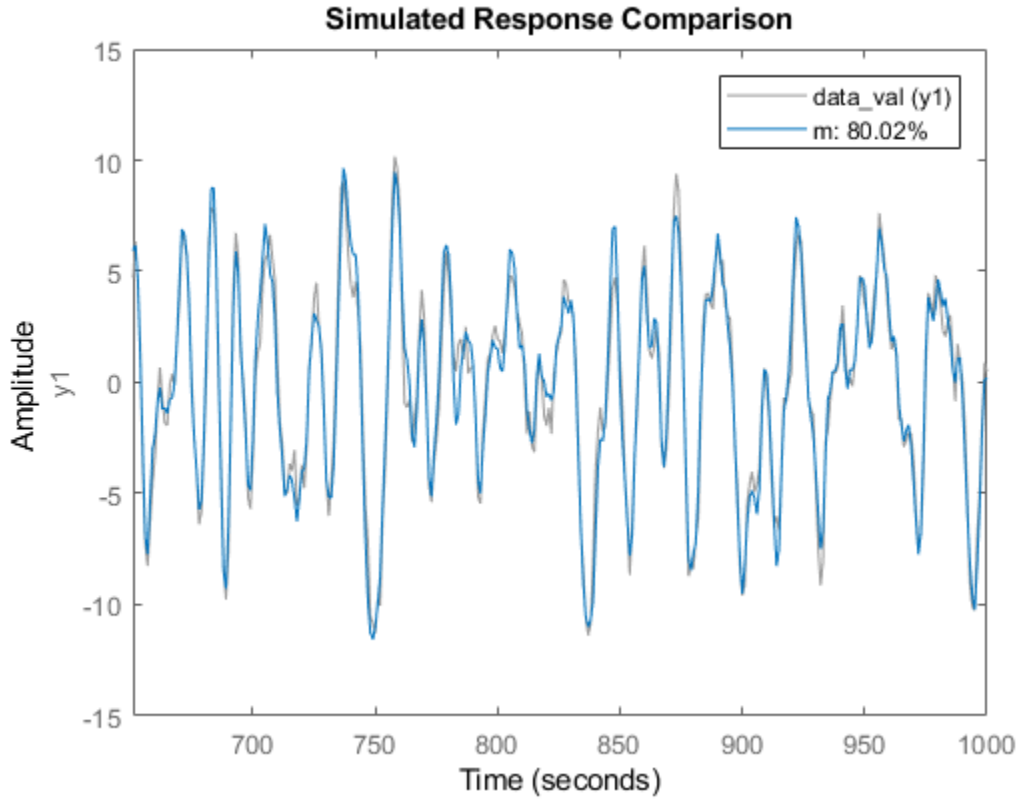
```
datam = merge(dat(1:250),dat(281:600),dat(651:1000));
```

Estimate a state-space model using the multiexperiment data set using experiments 1 and 2.

```
data_est = getexp(datam,[1,2]);
m = ssest(data_est,2);
```

Validate the model by comparing its output to the output data of experiment 3.

```
data_val = getexp(datam,3);
compare(data_val,m)
```


Simulated Response Comparison
data_val (y1)
m: 80.02%
Amplitude
y1
15
10
5
0
-5
-10
-15
700
750
800
850
900
950
1000
Time (seconds)

Handling Offsets and Trends in Data

In this section...
“When to Detrend Data” on page 2-114 “Alternatives for Detrending Data in App or at the Command-Line” on page 2-115 “Next Steps After Detrending” on page 2-116

When to Detrend Data

Detrending is removing means, offsets, or linear trends from regularly sampled time-domain input-output data signals. This data processing operation helps you estimate more accurate linear models because linear models cannot capture arbitrary differences between the input and output signal levels. The linear models you estimate from detrended data describe the relationship between the change in input signals and the change in output signals.

For steady-state data, you should remove mean values and linear trends from both input and output signals.

For transient data, you should remove physical-equilibrium offsets measured prior to the excitation input signal.

Remove one linear trend or several piecewise linear trends when the levels drift during the experiment. Signal drift is considered a low-frequency disturbance and can result in unstable models.

You should not detrend data before model estimation when you want:

- Linear models that capture offsets essential for describing important system dynamics. For example, when a model contains integration behavior, you could estimate a low-order transfer function (process model) from nondetrended data. For more information, see “Process Models”.
- Nonlinear black-box models, such as nonlinear ARX or Hammerstein-Wiener models. For more information, see “Nonlinear Model Identification”.

Tip When signals vary around a large signal level, you can improve computational accuracy of nonlinear models by detrending the signal means.

- Nonlinear ODE parameters (nonlinear grey-box models). For more information, see "Estimate Nonlinear Grey-Box Models" on page 13-35.

To simulate or predict the linear model response at the system operating conditions, you can restore the removed trend to the simulated or predicted model output using the `retrend` command.

For more information about handling drifts in the data, see the chapter on preprocessing data in *System Identification: Theory for the User*, Second Edition, by Lennart Ljung, Prentice Hall PTR, 1999.

Alternatives for Detrending Data in App or at the Command-Line

You can detrend data using the System Identification app and at the command line using the `detrend` command.

Both the app and the command line let you subtract the mean values and one linear trend from steady-state time-domain signals.

However, the `detrend` command provides the following additional functionality (not available in the app):

- Subtracting piecewise linear trends at specified breakpoints. A *breakpoint* is a time value that defines the discontinuities between successive linear trends.
- Subtracting arbitrary offsets and linear trends from transient data signals.
- Saving trend information to a variable so that you can apply it to multiple data sets.

As an alternative to detrending data beforehand, you can specify the offsets levels as estimation options and use them directly with the estimation command.

For example, suppose your data has an input offset, *u0*, and an output offset, *y0*. There are two ways to perform a linear model estimation (say, a transfer function model estimation) using this data:

- Using `detrend`:

```
T=getTrend(data)
T.InputOffset = u0;
T.OutputOffset = y0;
datad = detrend(data, T);

model = tfest(datad, np);
```

- Specify offsets as estimation options:

```
opt = tfestOptions('InputOffset',u0, 'OutputOffset', y0);

model = tfest(data, np, opt)
```

  The advantage of this approach is that there is a record of offset levels in the model in `model.Report.OptionsUsed`. The limitation of this approach is that it cannot handle linear trends, which can only be removed from the data by using `detrend`.

Next Steps After Detrending

After detrending your data, you might do the following:

- Perform other data preprocessing operations. See “Ways to Prepare Data for System Identification” on page 2-6.
- Estimate a linear model. See “Linear Model Identification”.

See Also

Related Examples

- “How to Detrend Data Using the App” on page 2-117
- “How to Detrend Data at the Command Line” on page 2-118

How to Detrend Data Using the App

Before you can perform this task, you must have regularly-sampled, steady-state time-domain data imported into the System Identification app. See “Import Time-Domain Data into the App” on page 2-16). For transient data, see “How to Detrend Data at the Command Line” on page 2-118.

Tip You can use the shortcut **Preprocess** > **Quick start** to perform several operations: remove the mean value from each signal, split data into two halves, specify the first half as model estimation data (or **Working Data**), and specify the second half as model **Validation Data**.

---

1. In the System Identification app, drag the data set you want to detrend to the **Working Data** rectangle.
2. Detrend the data.
   - To remove linear trends, select **Preprocess** > **Remove trends**.
   - To remove mean values from each input and output data signal, select **Preprocess** > **Remove means**.

See Also

More About

- “Handling Offsets and Trends in Data” on page 2-114

How to Detrend Data at the Command Line

In this section...
“Detrending Steady-State Data” on page 2-118 “Detrending Transient Data” on page 2-118

Detrending Steady-State Data

Before you can perform this task, you must have time-domain data as an `iddata` object. See “Representing Time- and Frequency-Domain Data Using iddata Objects” on page 2-50.

Note If you plan to estimate models from this data, your data must be regularly sampled.

Use the `detrend` command to remove the signal means or linear trends:

```
[data_d,T]=detrend(data,Type)
```

where `data` is the data to be detrended. The second input argument `Type=0` removes signal means or `Type=1` removes linear trends. `data_d` is the detrended data. `T` is a `TrendInfo` object that stores the values of the subtracted offsets and slopes of the removed trends.

Detrending Transient Data

Before you can perform this task, you must have

- Time-domain data as an `iddata` object. See “Representing Time- and Frequency-Domain Data Using iddata Objects” on page 2-50.

  Note If you plan to estimate models from this data, your data must be regularly sampled.

- Values of the offsets you want to remove from the input and output data. If you do not know these values, visually inspect a time plot of your data. For more information, see “How to Plot Data at the Command Line” on page 2-100.

1. Create a default object for storing input-output offsets that you want to remove from the data.

   ```
   T = getTrend(data)
   ```

   where `T` is a `TrendInfo` object.

2. Assign offset values to `T`.

   ```
   T.InputOffset=I_value;
   T.OutputOffset=O_value;
   ```

   where `I_value` is the input offset value, and `O_value` is the input offset value.

3. Remove the specified offsets from `data`.

   ```
   data_d = detrend(data,T)
   ```

   where the second input argument `T` stores the offset values as its properties.

See Also

`TrendInfo` | `detrend`

More About

- "Handling Offsets and Trends in Data" on page 2-114

Resampling Data

In this section...
“What Is Resampling?” on page 2-120
“Resampling Data Without Aliasing Effects” on page 2-121

What Is Resampling?

Resampling data signals in the System Identification Toolbox product applies an antialiasing (lowpass) FIR filter to the data and changes the sampling rate of the signal by decimation or interpolation.

If your data is sampled faster than needed during the experiment, you can decimate it without information loss. If your data is sampled more slowly than needed, there is a possibility that you miss important information about the dynamics at higher frequencies. Although you can resample the data at a higher rate, the resampled values occurring between measured samples do not represent new measured information about your system. Instead of resampling, repeat the experiment using a higher sampling rate.

Tip You should decimate your data when it contains high-frequency noise outside the frequency range of the system dynamics.

Resampling takes into account how the data behaves between samples, which you specify when you import the data into the System Identification app (zero-order or first-order hold). For more information about the data properties you specify before importing the data, see “Represent Data”.

You can resample data using the System Identification app or the `resample` command. You can only resample time-domain data at uniform time intervals.

For a detailed discussion about handling disturbances, see the chapter on preprocessing data in *System Identification: Theory for the User*, Second Edition, by Lennart Ljung, Prentice Hall PTR, 1999.

Resampling Data Without Aliasing Effects

Typically, you decimate a signal to remove the high-frequency contributions that result from noise from the total energy. Ideally, you want to remove the energy contribution due to noise and preserve the energy density of the signal.

The command `resample` performs the decimation without aliasing effects. This command includes a factor of T to normalize the spectrum and preserve the energy density after decimation. For more information about spectrum normalization, see “Spectrum Normalization” on page 9-14.

If you use manual decimation instead of `resample`—by picking every fourth sample from the signal, for example—the energy contributions from higher frequencies are folded back into the lower frequencies("aliasing"). Because the total signal energy is preserved by this operation and this energy must now be squeezed into a smaller frequency range, the amplitude of the spectrum at each frequency increases. Thus, the energy density of the decimated signal is not constant.

This example shows how `resample` avoids folding effects.

Construct a fourth-order moving-average process.

```
m0 = idpoly(1,[ ],[1 1 1 1]);
```

m0 is a time-series model with no inputs.

Generate error signal.

```
e = idinput(2000,'rgs');
```

Simulate the output using the error signal.

```
sim_opt = simOptions('AddNoise',true,'NoiseData',e);
y = sim(m0,zeros(2000,0),sim_opt);
y = iddata(y,[],1);
```

Estimate the signal spectrum.

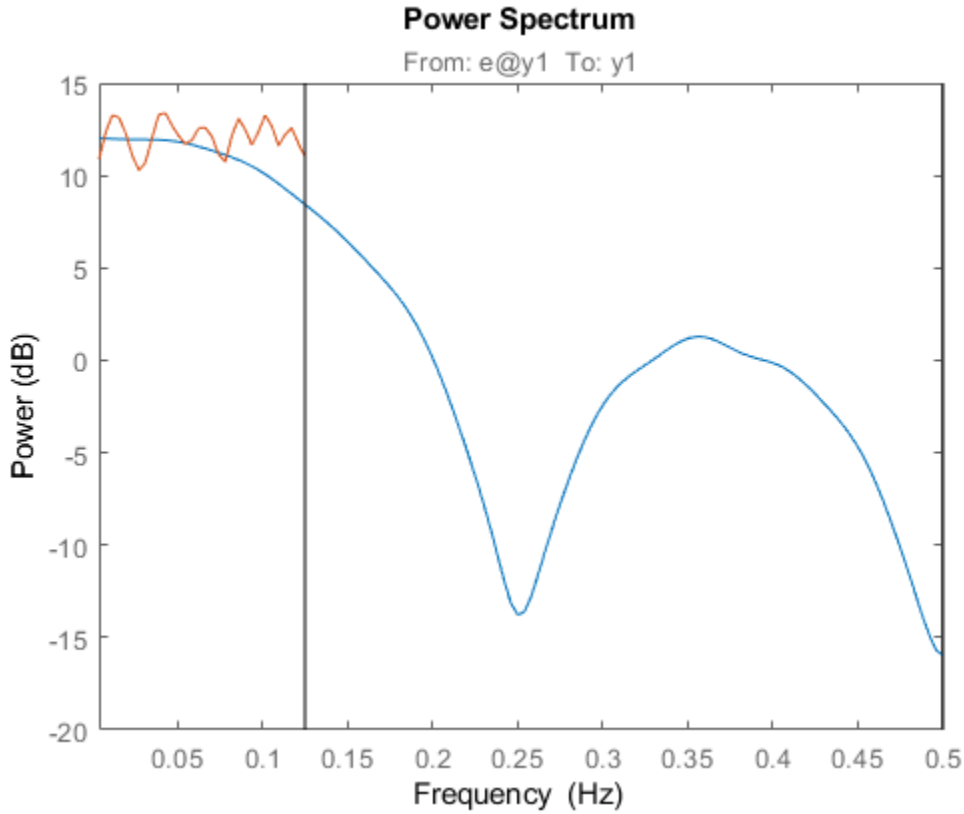
```
g1 = spa(y);
```

Estimate the spectrum of the modified signal including every fourth sample of the original signal. This command automatically sets Ts to 4.

```
g2 = spa(y(1:4:2000));
```

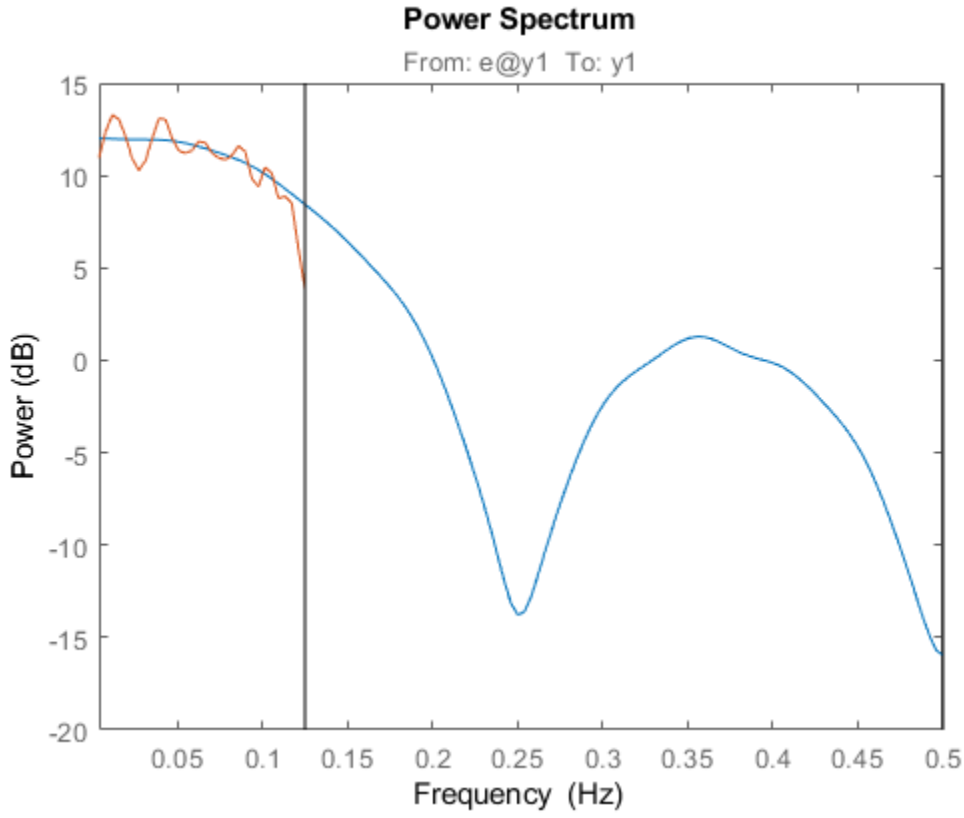
Plot the frequency response to view folding effects.

```
h = spectrumplot(g1,g2,g1.Frequency);
opt = getoptions(h);
opt.FreqScale = 'linear';
opt.FreqUnits = 'Hz';
setoptions(h,opt);
```



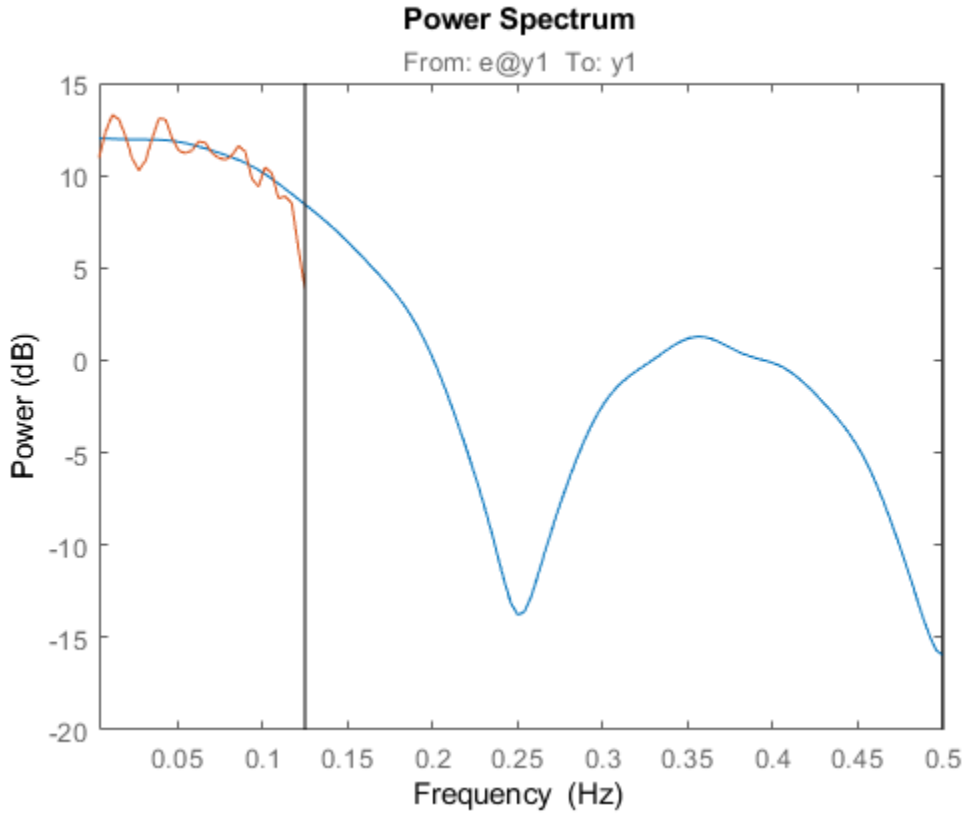
Estimate the spectrum after prefiltering that does not introduce folding effects.

```
g3 = spa(resample(y,1,4));
figure
spectrumplot(g1,g3,g1.Frequency,opt)
```



Use `resample` to decimate the signal before estimating the spectrum and plot the frequency response.

```
g3 = spa(resample(y,1,4));
figure
spectrumplot(g1,g3,g1.Frequency,opt)
```



The plot shows that the estimated spectrum of the resampled signal has the same amplitude as the original spectrum. Thus, there is no indication of folding effects when you use `resample` to eliminate aliasing.

See Also

Related Examples

- “Resampling Data Using the App” on page 2-125
- “Resampling Data at the Command Line” on page 2-127

Resampling Data Using the App

Use the System Identification app to resample time-domain data. To specify additional options, such as the prefilter order, see “Resampling Data at the Command Line” on page 2-127.

The System Identification app uses `idresamp` to interpolate or decimate the data. For more information about this command, type `help idresamp` at the prompt.

To create a new data set by resampling the input and output signals:

1. Import time-domain data into the System Identification app, as described in “Create Data Sets from a Subset of Signal Channels” on page 2-33.
2. Drag the data set you want to resample to the **Working Data** area.
3. In the **Resampling factor** field, enter the factor by which to multiply the current sample time:

   - For decimation (fewer samples), enter a factor greater than `1` to increase the sample time by this factor.
   - For interpolation (more samples), enter a factor less than `1` to decrease the sample time by this factor.

   Default = `1`.
4. In the **Data name** field, type the name of the new data set. Choose a name that is unique in the Data Board.
5. Click **Insert** to add the new data set to the Data Board in the System Identification Toolbox window.
6. Click **Close** to close the Resample dialog box.

See Also

Related Examples

- “Resampling Data at the Command Line” on page 2-127

More About

- “Resampling Data” on page 2-120

Resampling Data at the Command Line

Use `resample` to decimate and interpolate time-domain `iddata` objects. You can specify the order of the antialiasing filter as an argument.

Note `resample` uses the Signal Processing Toolbox™ command, when this toolbox is installed on your computer. If this toolbox is not installed, use `idresamp` instead. `idresamp` only lets you specify the filter order, whereas `resample` also lets you specify filter coefficients and the design parameters of the Kaiser window.

To create a new `iddata` object `datar` by resampling `data`, use the following syntax:

```
datar = resample(data,P,Q,filter_order)
```

In this case, `P` and `Q` are integers that specify the new sample time: the new sample time is `Q/P` times the original one. You can also specify the order of the resampling filter as a fourth argument `filter_order`, which is an integer (default is `10`). For detailed information about `resample`, see the corresponding reference page.

For example, `resample(data,1,Q)` results in decimation with the sample time modified by a factor `Q`.

The next example shows how you can increase the sampling rate by a factor of 1.5 and compare the signals:

```
plot(u)
ur = resample(u,3,2);
plot(u,ur)
```

When the Signal Processing Toolbox product is not installed, using `resample` calls `idresamp` instead.

`idresamp` uses the following syntax:

```
datar = idresamp(data,R,filter_order)
```

In this case, `R=Q/P`, which means that data is interpolated by a factor `P` and then decimated by a factor `Q`. To learn more about `idresamp`, type `help idresamp`.

The `data.InterSample` property of the `iddata` object is taken into account during resampling (for example, first-order hold or zero-order hold). For more information, see “iddata Properties” on page 2-52.

See Also

Related Examples

- “Resampling Data Using the App” on page 2-125

More About

- “Resampling Data” on page 2-120

Filtering Data

In this section...
“Supported Filters” on page 2-129 “Choosing to Prefilter Your Data” on page 2-129

Supported Filters

You can filter the input and output signals through a linear filter before estimating a model in the System Identification app or at the command line. How you want to handle the noise in the system determines whether it is appropriate to prefilter the data.

The filter available in the System Identification app is a fifth-order (passband) Butterworth filter. If you need to specify a custom filter, use the `idfilt` command.

Choosing to Prefilter Your Data

Prefiltering data can help remove high-frequency noise or low-frequency disturbances (drift). The latter application is an alternative to subtracting linear trends from the data, as described in “Handling Offsets and Trends in Data” on page 2-114.

In addition to minimizing noise, prefiltering lets you focus your model on specific frequency bands. The frequency range of interest often corresponds to a passband over the breakpoints on a Bode plot. For example, if you are modeling a plant for control-design applications, you might prefilter the data to specifically enhance frequencies around the desired closed-loop bandwidth.

Prefiltering the input and output data through the same filter does not change the input-output relationship for a linear system. However, prefiltering does change the noise characteristics and affects the estimated noise model of the system.

To get a reliable noise model in the app, instead of prefiltering the data, set **Focus** to `Filter`, and specify the filter. To get a reliable noise model at the command line, instead of prefiltering the data, specify the filter in the `WeightingFilter` estimation option of the estimation command. If the `Focus` option is available, specify it as `'simulation'`.

For more information about prefiltering data, see the chapter on preprocessing data in *System Identification: Theory for the User*, Second Edition, by Lennart Ljung, Prentice Hall PTR, 1999.

For practical examples of prefiltering data, see the section on posttreatment of data in *Modeling of Dynamic Systems*, by Lennart Ljung and Torkel Glad, Prentice Hall PTR, 1994.

See Also

Related Examples

- “How to Filter Data Using the App” on page 2-131
- “How to Filter Data at the Command Line” on page 2-134

How to Filter Data Using the App

In this section...
“Filtering Time-Domain Data in the App” on page 2-131
“Filtering Frequency-Domain or Frequency-Response Data in the App” on page 2-132

Filtering Time-Domain Data in the App

The System Identification app lets you filter time-domain data using a fifth-order Butterworth filter by enhancing or selecting specific passbands.

To create a filtered data set:

1. Import time-domain data into the System Identification app, as described in “Represent Data”.
2. Drag the data set you want to filter to the **Working Data** area.
3. Select **<--Preprocess** > **Filter**. By default, this selection shows a periodogram of the input and output spectra (see the `etfe` reference page).

   Note To display smoothed spectral estimates instead of the periodogram, select **Options** > **Spectral analysis**. This spectral estimate is computed using `spa` and your previous settings in the Spectral Model dialog box. To change these settings, select **<--Estimate** > **Spectral model** in the System Identification app, and specify new model settings.
4. If your data contains multiple input/output channels, in the **Channel** menu, select the channel pair you want to view. Although you view only one channel pair at a time, the filter applies to all input/output channels in this data set.
5. Select the data of interest using one of the following ways:

   - Graphically — Draw a rectangle with the mouse on either the input-signal or the output-signal plot to select the desired frequency interval. Your selection is displayed on both plots regardless of the plot on which you draw the rectangle. The **Range** field is updated to match the selected region. If you need to clear your selection, right-click the plot.
   - Specify the **Range** — Edit the beginning and the end frequency values.

     For example:

`8.5 20.0` (rad/s).

> **Tip** To change the frequency units from `rad/s` to `Hz`, select **Style > Frequency (Hz)**. To change the frequency units from `Hz` to `rad/s`, select **Style > Frequency (rad/s)**.

6 In the **Range is** list, select one of the following:

- `Pass band` — Allows data in the selected frequency range.
- `Stop band` — Excludes data in the selected frequency range.

7 Click **Filter** to preview the filtered results. If you are satisfied, go to step 8. Otherwise, return to step 5.

8 In the **Data name** field, enter the name of the data set containing the selected data.

9 Click **Insert** to save the selection as a new data set and add it to the Data Board.

10 To select another range, repeat steps 5 to 9.

Filtering Frequency-Domain or Frequency-Response Data in the App

For frequency-domain and frequency-response data, *filtering* is equivalent to selecting specific data ranges.

To select a range of data in frequency-domain or frequency-response data:

1 Import data into the System Identification app, as described in “Represent Data”.

2 Drag the data set you want you want to filter to the **Working Data** area.

3 Select **<--Preprocess > Select range**. This selection displays one of the following plots:

- Frequency-domain data — Plot shows the absolute of the squares of the input and output spectra.
- Frequency-response data — Top axes show the frequency response magnitude equivalent to the ratio of the output to the input, and the bottom axes show the ratio of the input signal to itself, which has the value of `1` at all frequencies.

4 If your data contains multiple input/output channels, in the **Channel** menu, select the channel pair you want to view. Although you view only one channel pair at a time, the filter applies to all input/output channels in this data set.

5 Select the data of interest using one of the following ways:

- Graphically — Draw a rectangle with the mouse on either the input-signal or the output-signal plot to select the desired frequency interval. Your selection is displayed on both plots regardless of the plot on which you draw the rectangle. The **Range** field is updated to match the selected region.

  If you need to clear your selection, right-click the plot.

- Specify the **Range** — Edit the beginning and the end frequency values.

  For example:

  `8.5 20.0` (rad/s).

  Tip If you need to change the frequency units from `rad/s` to `Hz`, select **Style > Frequency (Hz)**. To change the frequency units from `Hz` to `rad/s`, select **Style > Frequency (rad/s)**.

6 In the **Range is** list, select one of the following:

- `Pass band` — Allows data in the selected frequency range.
- `Stop band` — Excludes data in the selected frequency range.

7 In the **Data name** field, enter the name of the data set containing the selected data.

8 Click **Insert**. This action saves the selection as a new data set and adds it to the Data Board.

9 To select another range, repeat steps 5 to 8.

See Also

Related Examples

- “How to Filter Data at the Command Line” on page 2-134

More About

- “Filtering Data” on page 2-129

How to Filter Data at the Command Line

In this section...
“Simple Passband Filter” on page 2-134 “Defining a Custom Filter” on page 2-135 “Causal and Noncausal Filters” on page 2-136

Simple Passband Filter

Use `idfilt` to apply passband and other custom filters to a time-domain or a frequency-domain `iddata` object.

In general, you can specify any custom filter. Use this syntax to filter an `iddata` object `data` using the filter called `filter`:

```
fdata = idfilt(data,filter)
```

In the simplest case, you can specify a passband filter for time-domain data using the following syntax:

```
fdata = idfilt(data,[wl wh])
```

In this case, `w1` and `wh` represent the low and high frequencies of the passband, respectively.

You can specify several passbands, as follows:

```
filter=[[w1l,w1h];[ w2l,w2h]; ....;[wnl,wnh]]
```

The filter is an `n`-by-`2` matrix, where each row defines a passband in radians per second.

To define a stopband between `ws1` and `ws2`, use

```
filter = [0 ws1; ws2 Nyqf]
```

where, `Nyqf` is the Nyquist frequency.

For time-domain data, the passband filtering is cascaded Butterworth filters of specified order. The default filter order is 5. The Butterworth filter is the same as `butter` in the Signal Processing Toolbox product. For frequency-domain data, select the indicated portions of the data to perform passband filtering.

Defining a Custom Filter

Use `idfilt` to apply passband and other custom filters to a time-domain or a frequency-domain `iddata` object.

In general, you can specify any custom filter. Use this syntax to filter an `iddata` object `data` using the filter called `filter`:

```
fdata = idfilt(data,filter)
```

You can define a general single-input/single-output (SISO) system for filtering time-domain or frequency-domain data. For frequency-domain only, you can specify the (nonparametric) frequency response of the filter.

You use this syntax to filter an `iddata` object `data` using a custom filter specified by `filter`:

```
fdata = idfilt(data,filter)
```

`filter` can be also any of the following:

```
filter = idm
filter = {num,den}
filter = {A,B,C,D}
```

`idm` is a SISO identified linear model on page 1-13 or LTI object. For more information about LTI objects, see the Control System Toolbox documentation.

`{num,den}` defines the filter as a transfer function as a cell array of numerator and denominator filter coefficients.

`{A,B,C,D}` is a cell array of SISO state-space matrices.

Specifically for frequency-domain data, you specify the frequency response of the filter:

```
filter = Wf
```

Here, `Wf` is a vector of real or complex values that define the filter frequency response, where the inputs and outputs of `data` at frequency `data.Frequency(kf)` are multiplied by `Wf(kf)`. `Wf` is a column vector with the length equal to the number of frequencies in `data`.

When `data` contains several experiments, `Wf` is a cell array with the length equal to the number of experiments in `data`.

Causal and Noncausal Filters

For time-domain data, the filtering is causal by default. Causal filters typically introduce a phase shift in the results. To use a noncausal zero-phase filter (corresponding to `filtfilt` in the Signal Processing Toolbox product), specify a third argument in `idfilt`:

```
fdata = idfilt(data,filter,'noncausal')
```

For frequency-domain data, the signals are multiplied by the frequency response of the filter. With the filters defined as passband filters, this calculation gives ideal, zero-phase filtering ("brick wall filters"). Frequencies that have been assigned zero weight by the filter (outside the passband or via frequency response) are removed.

When you apply `idfilt` to an `idfrd` data object, the data is first converted to a frequency-domain `iddata` object (see "Transforming Between Frequency-Domain and Frequency-Response Data" on page 3-12). The result is an `iddata` object.

See Also

More About

- "Filtering Data" on page 2-129

Generate Data Using Simulation

In this section...
“Commands for Generating Data Using Simulation” on page 2-137
“Create Periodic Input Data” on page 2-138
“Generate Output Data Using Simulation” on page 2-140
“Simulating Data Using Other MathWorks Products” on page 2-142

Commands for Generating Data Using Simulation

You can generate input data and then use it with a model to create output data.

Simulating output data requires that you have a model with known coefficients. For more information about commands for constructing models, see “Commands for Constructing Linear Model Structures” on page 1-21.

To generate input data, use `idinput` to construct a signal with the desired characteristics, such as a random Gaussian or binary signal or a sinusoid. `idinput` returns a matrix of input values.

The following table lists the commands you can use to simulate output data. For more information about these commands, see the corresponding reference pages.

Commands for Generating Data

Command	Description	Example
`idinput`	Constructs a signal with the desired characteristics, such as a random Gaussian or binary signal or a sinusoid, and returns a matrix of input values.	`u = iddata([],...` `  idinput(400,'rbs',[0 0.3]));`
`sim`	Simulates response data based on existing linear or nonlinear parametric model in the MATLAB workspace.	To simulate the model output `y` for a given input, use the following command: `y = sim(m,data)` `m` is the model object name, and `data` is input data matrix or `iddata` object.

Create Periodic Input Data

This example shows how to create a periodic random Gaussian input signal using `idinput`.

Create a periodic input for one input and consisting of five periods, where each period is 300 samples.

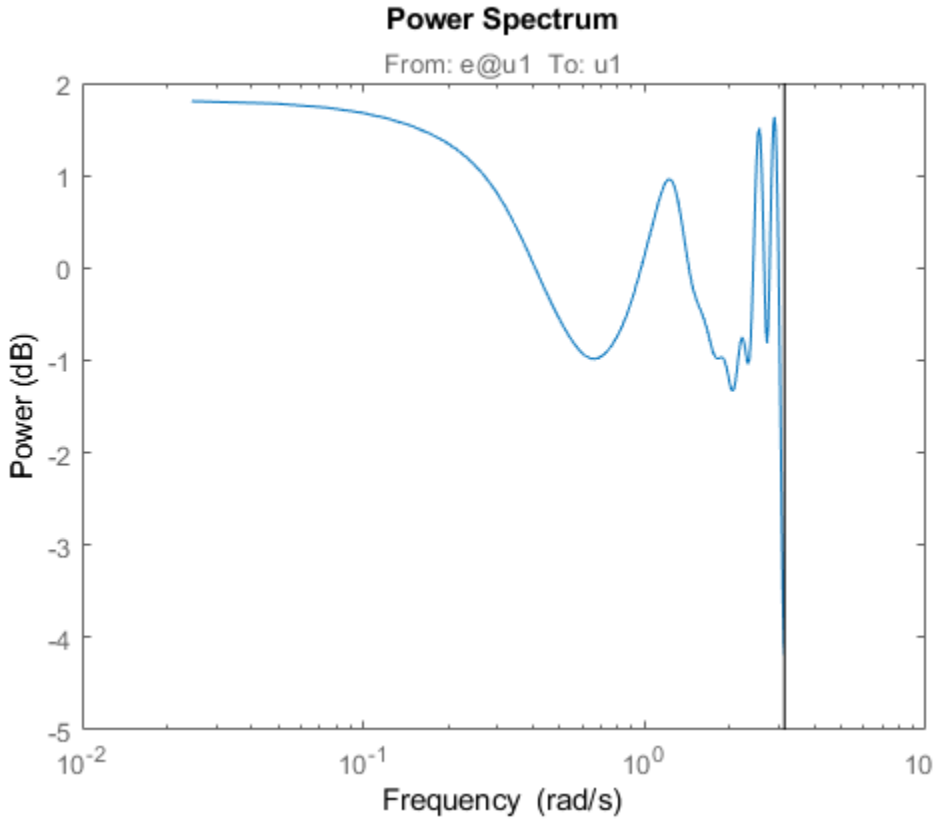
```
per_u = idinput([300 1 5]);
```

Create an `iddata` object using the periodic input and leaving the output empty.

```
u = iddata([],per_u,'Period',.300);
```

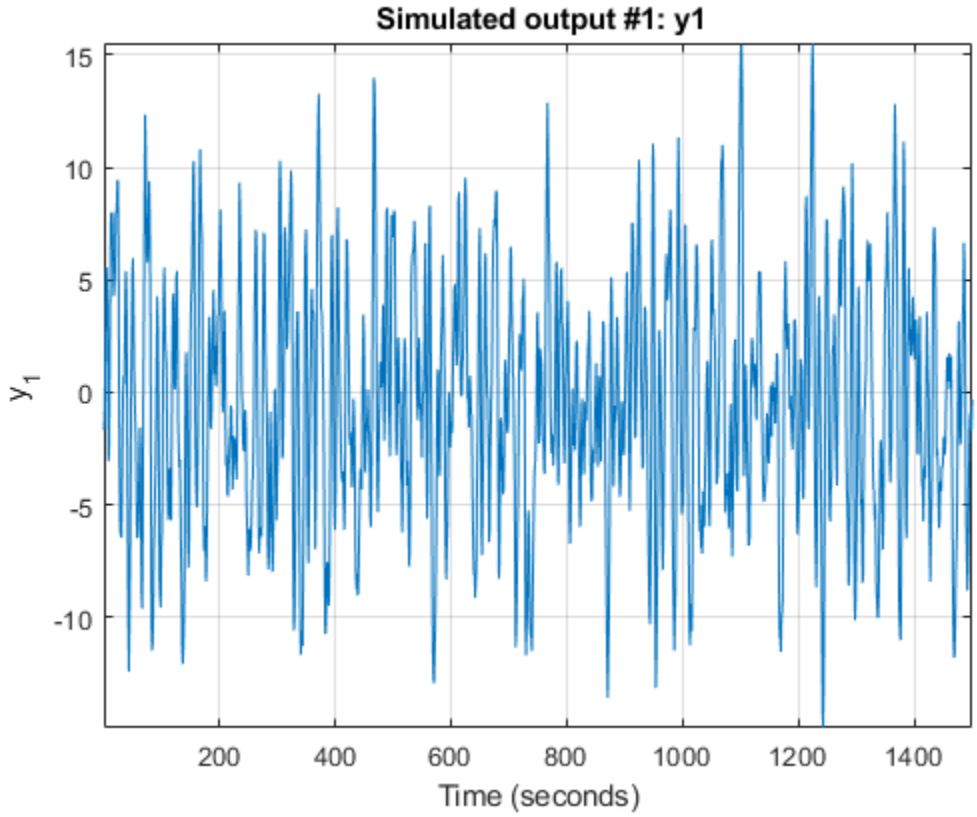
View the data characteristics in time- and frequency-domain.

```
% Plot data in time-domain.
plot(u)
% Plot the spectrum.
spectrum(spa(u))
```



(Optional) Simulate model output using the data.

```
% Construct a polynomial model.
m0 = idpoly([1 -1.5 0.7],[0 1 0.5]);
% Simulate model output with Gaussian noise.
sim_opt = simOptions('AddNoise',true);
sim(m0,u,sim_opt)
```



Generate Output Data Using Simulation

This example shows how to generate output data by simulating a model using an input signal created using `idinput`.

You use the generated data to estimate a model of the same order as the model used to generate the data. Then, you check how closely both models match to understand the effects of input data characteristics and noise on the estimation.

Create an ARMAX model with known coefficients.

```
A = [1 -1.2 0.7];
B = {[0 1 0.5 0.1],[0 1.5 -0.5],[0 -0.1 0.5 -0.1]};
```

```
C = [1 0 0 0 0];
Ts = 1;
m0 = idpoly(A,B,C,'Ts',1);
```

The leading zeros in the `B` matrix indicate the input delay (`nk`), which is `1` for each input channel.

Construct a pseudorandom binary input data.

```
u = idinput([255,3],'prbs');
```

Simulate model output with noise using the input data.

```
y = sim(m0,u,simOptions('AddNoise',true));
```

Represent the simulation data as an `iddata` object.

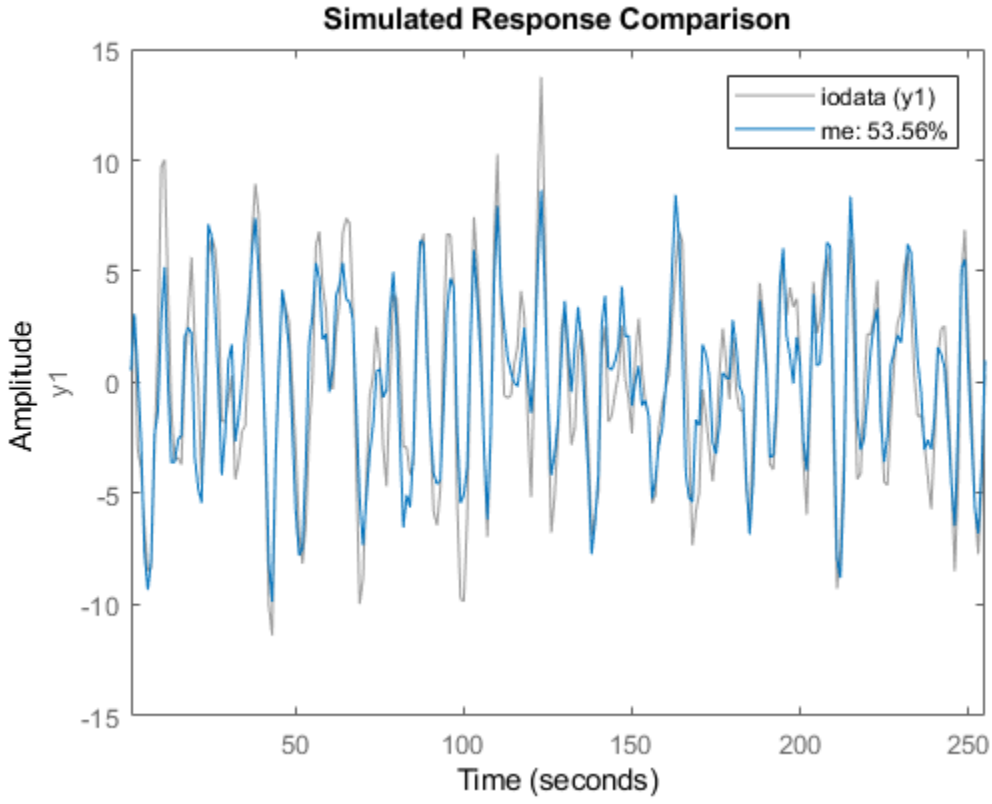
```
iodata = iddata(y,u,m0.Ts);
```

(Optional) Estimate a model of the same order as `m0` using `iodata`.

```
na = 2;
nb = [3 2 3];
nc = 4;
nk = [1 1 1];
me = armax(iodata,[na,nb,nc,nk]);
```

Use `bode(m0,me)` and `compare(iodata,me)` to check how closely `me` and `m0` match.

```
compare(iodata,me);
```



Simulating Data Using Other MathWorks Products

You can also simulate data using the Simulink and Signal Processing Toolbox software. Data simulated outside the System Identification Toolbox product must be in the MATLAB workspace as double matrices. For more information about simulating models using the Simulink software, see “Simulating Identified Model Output in Simulink” on page 20-5.

Manipulating Complex-Valued Data

In this section...
“Supported Operations for Complex Data” on page 2-143 “Processing Complex iddata Signals at the Command Line” on page 2-143

Supported Operations for Complex Data

System Identification Toolbox estimation algorithms support complex data. For example, the following estimation commands estimate complex models from complex data: `ar`, `armax`, `arx`, `bj`, `ivar`, `iv4`, `oe`, `pem`, `spa`, `tfest`, `ssest`, and `n4sid`.

Model transformation routines, such as `freqresp` and `zpkdata`, work for complex-valued models. However, they do not provide pole-zero confidence regions. For complex models, the parameter variance-covariance information refers to the complex-valued parameters and the accuracy of the real and imaginary is not computed separately.

The display commands `compare` and `plot` also work with complex-valued data and models. To plot the real and imaginary parts of the data separately, use `plot(real(data))` and `plot(imag(data))`, respectively.

Processing Complex iddata Signals at the Command Line

If the `iddata` object `data` contains complex values, you can use the following commands to process the complex data and create a new `iddata` object.

Command	Description
`abs(data)`	Absolute value of complex signals in `iddata` object.
`angle(data)`	Phase angle (in radians) of each complex signals in `iddata` object.
`complex(data)`	For time-domain data, this command makes the `iddata` object complex—even when the imaginary parts are zero. For frequency-domain data that only stores the values for nonnegative frequencies, such that `realdata(data)=1`, it adds signal values for negative frequencies using complex conjugation.
`imag(data)`	Selects the imaginary parts of each signal in `iddata` object.

Command	Description
`isreal(data)`	`1` when `data` (time-domain or frequency-domain) contains only real input and output signals, and returns `0` when `data` (time-domain or frequency-domain) contains complex signals.
`real(data)`	Real part of complex signals in `iddata` object.
`realdata(data)`	Returns a value of `1` when `data` is a real-valued, time-domain signal, and returns `0` otherwise.

For example, suppose that you create a frequency-domain `iddata` object `Datf` by applying `fft` to a real-valued time-domain signal to take the Fourier transform of the signal. The following is true for `Datf`:

```
isreal(Datf) = 0
realdata(Datf) = 1
```

Transform Data

- "Supported Data Transformations" on page 3-2
- "Transform Time-Domain Data in the App" on page 3-4
- "Transform Frequency-Domain Data in the App" on page 3-6
- "Transform Frequency-Response Data in the App" on page 3-8
- "Transforming Between Time and Frequency-Domain Data" on page 3-11
- "Transforming Between Frequency-Domain and Frequency-Response Data" on page 3-12

Supported Data Transformations

The following table shows the different ways you can transform data from one data domain to another. If the transformation is supported for a given row and column combination in the table, the command used by the software is listed in the cell at their intersection.

Original Data Format	To Time-Domain Data (`iddata` object)	To Frequency-Domain Data (`iddata` object)	To Frequency-Response Data (`idfrd` object)
Time-Domain Data (`iddata` object)	N/A	Use `fft`	• Use `etfe`, `spa`, or `spafdr`. • Estimate a linear parametric model from the `iddata` object, and use `idfrd` to compute frequency-response data.
Frequency-Domain Data (`iddata` object)	Use `ifft` (works only for evenly spaced frequency-domain data).	N/A	• Use `etfe`, `spa`, or `spafdr`. • Estimate a linear parametric model from the `iddata` object, and use `idfrd` to compute frequency-response data.
Frequency-Response Data (`idfrd` object)	Not supported	Use `iddata`. The software creates a frequency-domain `iddata` object that has the same ratio between output and input as the original `idfrd` object frequency-response data.	• Use `spafdr`. The software calculates frequency-response data with a different resolution (number and spacing of frequencies) than the original data.

Transforming from time-domain or frequency-domain data to frequency-response data is equivalent to creating a frequency-response model from the data. For more information, see “Frequency-Response Models”.

See Also

Related Examples

- “Transforming Between Time and Frequency-Domain Data” on page 3-11
- “Transform Time-Domain Data in the App” on page 3-4
- “Transform Frequency-Domain Data in the App” on page 3-6
- “Transform Frequency-Response Data in the App” on page 3-8

More About

- “Representing Data in MATLAB Workspace” on page 2-9

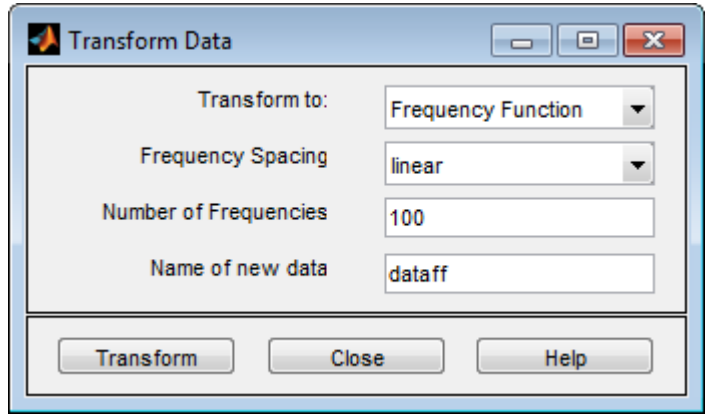
Transform Time-Domain Data in the App

In the System Identification app, time-domain data has an icon with a white background. You can transform time-domain data to frequency-domain or frequency-response data. The frequency values of the resulting frequency vector range from `0` to the Nyquist frequency $f_S = \pi / T_s$, where T_s is the sample time.

Transforming from time-domain to frequency-response data is equivalent to estimating a model from the data using the `spafdr` method.

1. In the System Identification app, drag the icon of the data you want to transform to the **Working Data** rectangle.
2. In the **Operations** area, select **<–Preprocess** > **Transform data** in the drop-down menu to open the Transform Data dialog box.
3. In the **Transform to** list, select one of the following:

   - `Frequency Function` — Create a new `idfrd` object using the `spafdr` method. Go to step 4.

   - `Frequency Domain Data` — Create a new `iddata` object using the `fft` method. Go to step 6.
4. In the **Frequency Spacing** list, select the spacing of the frequencies at which the frequency function is estimated:

   - `linear` — Uniform spacing of frequency values between the endpoints.

- `logarithmic` — Base-10 logarithmic spacing of frequency values between the endpoints.

5 In the **Number of Frequencies** field, enter the number of frequency values.

6 In the **Name of new data** field, type the name of the new data set. This name must be unique in the Data Board.

7 Click **Transform** to add the new data set to the Data Board in the System Identification app.

8 Click **Close** to close the Transform Data dialog box.

See Also

Related Examples

- "Transforming Between Time and Frequency-Domain Data" on page 3-11
- "Transform Frequency-Domain Data in the App" on page 3-6
- "Transform Frequency-Response Data in the App" on page 3-8

More About

- "Representing Data in MATLAB Workspace" on page 2-9
- "Supported Data Transformations" on page 3-2

Transform Frequency-Domain Data in the App

In the System Identification app, frequency-domain data has an icon with a green background. You can transform frequency-domain data to time-domain or frequency-response (frequency-function) data.

Transforming from time-domain or frequency-domain data to frequency-response data is equivalent to estimating a nonparametric model of the data using the `spafdr` method.

1. In the System Identification app, drag the icon of the data you want to transform to the **Working Data** rectangle.
2. Select **<--Preprocess** > **Transform data**.
3. In the **Transform to** list, select one of the following:
   - `Frequency Function` — Create a new `idfrd` object using the `spafdr` method. Go to step 4.
   - `Time Domain Data` — Create a new `iddata` object using the `ifft` (inverse fast Fourier transform) method. Go to step 6.
4. In the **Frequency Spacing** list, select the spacing of the frequencies at which the frequency function is estimated:
   - `linear` — Uniform spacing of frequency values between the endpoints.
   - `logarithmic` — Base-10 logarithmic spacing of frequency values between the endpoints.
5. In the **Number of Frequencies** field, enter the number of frequency values.
6. In the **Name of new data** field, type the name of the new data set. This name must be unique in the Data Board.
7. Click **Transform** to add the new data set to the Data Board in the System Identification app.
8. Click **Close** to close the Transform Data dialog box.

See Also

Related Examples

- “Transforming Between Time and Frequency-Domain Data” on page 3-11
- “Transform Time-Domain Data in the App” on page 3-4
- “Transform Frequency-Response Data in the App” on page 3-8

More About

- “Representing Data in MATLAB Workspace” on page 2-9
- “Supported Data Transformations” on page 3-2

Transform Frequency-Response Data in the App

In the System Identification app, frequency-response data has an icon with a yellow background. You can transform frequency-response data to frequency-domain data (`iddata` object) or to frequency-response data with a different frequency resolution.

When you select to transform single-input/single-output (SISO) frequency-response data to frequency-domain data, the toolbox creates outputs that equal the frequency responses, and inputs equal to 1. Therefore, the ratio between the Fourier transform of the output and the Fourier transform of the input is equal to the system frequency response.

For the multiple-input case, the toolbox transforms the frequency-response data to frequency-domain data as if each input contributes independently to the entire output of the system and then combines information. For example, if a system has three inputs, `u1`, `u2`, and `u3` and two frequency samples, the input matrix is set to:

$$\begin{bmatrix} 1 & 0 & 0 \\ 1 & 0 & 0 \\ 0 & 1 & 0 \\ 0 & 1 & 0 \\ 0 & 0 & 1 \\ 0 & 0 & 1 \end{bmatrix}$$

In general, for `nu` inputs and `ns` samples (the number of frequencies), the input matrix has `nu` columns and (`ns` · `nu`) rows.

Note To create a separate experiment for the response from each input, see "Transforming Between Frequency-Domain and Frequency-Response Data" on page 3-12.

When you transform frequency-response data by changing its frequency resolution, you can modify the number of frequency values by changing between linear or logarithmic spacing. You might specify variable frequency spacing to increase the number of data points near the system resonance frequencies, and also make the frequency vector coarser in the region outside the system dynamics. Typically, high-frequency noise dominates away from frequencies where interesting system dynamics occur. The System

Identification app lets you specify logarithmic frequency spacing, which results in a variable frequency resolution.

Note The `spafdr` command lets you lets you specify any variable frequency resolution.

1. In the System Identification app, drag the icon of the data you want to transform to the **Working Data** rectangle.
2. Select **<--Preprocess** > **Transform data**.
3. In the **Transform to** list, select one of the following:
   - `Frequency Domain Data` — Create a new `iddata` object. Go to step 6.
   - `Frequency Function` — Create a new `idfrd` object with different resolution (number and spacing of frequencies) using the `spafdr` method. Go to step 4.
4. In the **Frequency Spacing** list, select the spacing of the frequencies at which the frequency function is estimated:
   - `linear` — Uniform spacing of frequency values between the endpoints.
   - `logarithmic` — Base-10 logarithmic spacing of frequency values between the endpoints.
5. In the **Number of Frequencies** field, enter the number of frequency values.
6. In the **Name of new data** field, type the name of the new data set. This name must be unique in the Data Board.
7. Click **Transform** to add the new data set to the Data Board in the System Identification app.
8. Click **Close** to close the Transform Data dialog box.

See Also

Related Examples

- “Transforming Between Time and Frequency-Domain Data” on page 3-11
- “Transform Time-Domain Data in the App” on page 3-4
- “Transform Frequency-Domain Data in the App” on page 3-6

More About

- “Representing Data in MATLAB Workspace” on page 2-9
- “Supported Data Transformations” on page 3-2

Transforming Between Time and Frequency-Domain Data

The `iddata` object stores time-domain or frequency-domain data. The following table summarizes the commands for transforming data between time and frequency domains.

Command	Description	Syntax Example
`fft`	Transforms time-domain data to the frequency domain. You can specify `N`, the number of frequency values.	To transform time-domain `iddata` object `t_data` to frequency-domain `iddata` object `f_data` with `N` frequency points, use: `f_data = fft(t_data,N)`
`ifft`	Transforms frequency-domain data to the time domain. Frequencies are linear and equally spaced.	To transform frequency-domain`iddata` object `f_data` to time-domain `iddata` object `t_data`, use: `t_data = ifft(f_data)`

See Also

Related Examples

- “Transforming Between Frequency-Domain and Frequency-Response Data” on page 3-12
- “Transform Time-Domain Data in the App” on page 3-4
- “Transform Frequency-Domain Data in the App” on page 3-6
- “Transform Frequency-Response Data in the App” on page 3-8

More About

- “Representing Data in MATLAB Workspace” on page 2-9
- “Supported Data Transformations” on page 3-2

Transforming Between Frequency-Domain and Frequency-Response Data

You can transform frequency-response data to frequency-domain data (`iddata` object). The `idfrd` object represents complex frequency-response of the system at different frequencies. For a description of this type of data, see "Frequency-Response Data Representation" on page 2-13.

When you select to transform single-input/single-output (SISO) frequency-response data to frequency-domain data, the toolbox creates outputs that equal the frequency responses, and inputs equal to 1. Therefore, the ratio between the Fourier transform of the output and the Fourier transform of the input is equal to the system frequency response.

For information about changing the frequency resolution of frequency-response data to a new constant or variable (frequency-dependent) resolution, see the `spafdr` reference page. You might use this feature to increase the number of data points near the system resonance frequencies and make the frequency vector coarser in the region outside the system dynamics. Typically, high-frequency noise dominates away from frequencies where interesting system dynamics occur.

Note You cannot transform an `idfrd` object to a time-domain `iddata` object.

To transform an `idfrd` object with the name `idfrdobj` to a frequency-domain `iddata` object, use the following syntax:

```
dataf = iddata(idfrdobj)
```

The resulting frequency-domain `iddata` object contains values at the same frequencies as the original `idfrd` object.

For the multiple-input case, the toolbox represents frequency-response data as if each input contributes independently to the entire output of the system and then combines information. For example, if a system has three inputs, `u1`, `u2`, and `u3` and two frequency samples, the input matrix is set to:

$$\begin{bmatrix} 1 & 0 & 0 \\ 1 & 0 & 0 \\ 0 & 1 & 0 \\ 0 & 1 & 0 \\ 0 & 0 & 1 \\ 0 & 0 & 1 \end{bmatrix}$$

In general, for `nu` inputs and `ns` samples, the input matrix has `nu` columns and (`ns`⋅ `nu`) rows.

If you have `ny` outputs, the transformation operation produces an output matrix has `ny` columns and (`ns`⋅ `nu`) rows using the values in the complex frequency response *G(iw)* matrix (`ny`-by-`nu`-by-`ns`). In this example, `y1` is determined by unfolding `G(1,1,:)`, `G(1,2,:)`, and `G(1,3,:)` into three column vectors and vertically concatenating these vectors into a single column. Similarly, `y2` is determined by unfolding `G(2,1,:)`, `G(2,2,:)`, and `G(2,3,:)` into three column vectors and vertically concatenating these vectors.

If you are working with multiple inputs, you also have the option of storing the contribution by each input as an independent experiment in a multiexperiment data set. To transform an `idfrd` object with the name `idfrdobj` to a multiexperiment data set `datf`, where each experiment corresponds to each of the inputs in `idfrdobj`

```
datf = iddata(idfrdobj,'me')
```

In this example, the additional argument `'me'` specifies that multiple experiments are created.

By default, transformation from frequency-response to frequency-domain data strips away frequencies where the response is `inf` or `NaN`. To preserve the entire frequency vector, use `datf = iddata(idfrdobj,'inf')`. For more information, type `help idfrd/iddata`.

See Also

Related Examples

- “Transforming Between Time and Frequency-Domain Data” on page 3-11
- “Transform Time-Domain Data in the App” on page 3-4
- “Transform Frequency-Domain Data in the App” on page 3-6
- “Transform Frequency-Response Data in the App” on page 3-8

More About

- “Representing Data in MATLAB Workspace” on page 2-9
- “Supported Data Transformations” on page 3-2

Linear Model Identification

- "Black-Box Modeling" on page 4-2
- "Refine Linear Parametric Models" on page 4-7
- "Refine ARMAX Model with Initial Parameter Guesses at Command Line" on page 4-10
- "Refine Initial ARMAX Model at Command Line" on page 4-12
- "Extracting Numerical Model Data" on page 4-14
- "Transforming Between Discrete-Time and Continuous-Time Representations" on page 4-17
- "Continuous-Discrete Conversion Methods" on page 4-21
- "Effect of Input Intersample Behavior on Continuous-Time Models" on page 4-31
- "Transforming Between Linear Model Representations" on page 4-35
- "Subreferencing Models" on page 4-38
- "Treating Noise Channels as Measured Inputs" on page 4-41
- "Concatenating Models" on page 4-43
- "Merging Models" on page 4-47
- "Determining Model Order and Delay" on page 4-48
- "Model Structure Selection: Determining Model Order and Input Delay" on page 4-49
- "Frequency Domain Identification: Estimating Models Using Frequency Domain Data" on page 4-65
- "Building Structured and User-Defined Models Using System Identification Toolbox™" on page 4-92

Black-Box Modeling

In this section...
“Selecting Black-Box Model Structure and Order” on page 4-2 “When to Use Nonlinear Model Structures?” on page 4-4 “Black-Box Estimation Example” on page 4-4

Selecting Black-Box Model Structure and Order

Black-box modeling is useful when your primary interest is in fitting the data regardless of a particular mathematical structure of the model. The toolbox provides several linear and nonlinear black-box model structures, which have traditionally been useful for representing dynamic systems. These model structures vary in complexity depending on the flexibility you need to account for the dynamics and noise in your system. You can choose one of these structures and compute its parameters to fit the measured response data.

Black-box modeling is usually a trial-and-error process, where you estimate the parameters of various structures and compare the results. Typically, you start with the simple linear model structure and progress to more complex structures. You might also choose a model structure because you are more familiar with this structure or because you have specific application needs.

The simplest linear black-box structures require the fewest options to configure:

- Transfer function on page 8-2, with a given number of poles and zeros.
- Linear ARX model on page 6-4, which is the simplest input-output polynomial model.
- State-space model on page 7-2, which you can estimate by specifying the number of model states

Estimation of some of these structures also uses noniterative estimation algorithms, which further reduces complexity.

You can configure a model structure using the model order. The definition of model order varies depending on the type of model you select. For example, if you choose a transfer function representation, the model order is related to the number of poles and zeros. For state-space representation, the model order corresponds to the number of states. In some

cases, such as for linear ARX and state-space model structures, you can estimate the model order from the data.

If the simple model structures do not produce good models, you can select more complex model structures by:

- Specifying a higher model order for the same linear model structure. Higher model order increases the model flexibility for capturing complex phenomena. However, unnecessarily high orders can make the model less reliable.
- Explicitly modeling the noise:

  $y(t)=Gu(t)+He(t)$

  where H models the additive disturbance by treating the disturbance as the output of a linear system driven by a white noise source $e(t)$.

  Using a model structure that explicitly models the additive disturbance can help to improve the accuracy of the measured component G. Furthermore, such a model structure is useful when your main interest is using the model for predicting future response values.
- Using a different linear model structure.

  See “Linear Model Structures” on page 1-20.
- Using a nonlinear model structure.

  Nonlinear models have more flexibility in capturing complex phenomena than linear models of similar orders. See “Nonlinear Model Structures” on page 11-7.

Ultimately, you choose the simplest model structure that provides the best fit to your measured data. For more information, see “Estimating Linear Models Using Quick Start”.

Regardless of the structure you choose for estimation, you can simplify the model for your application needs. For example, you can separate out the measured dynamics (G) from the noise dynamics (H) to obtain a simpler model that represents just the relationship between y and u. You can also linearize a nonlinear model about an operating point.

When to Use Nonlinear Model Structures?

A linear model is often sufficient to accurately describe the system dynamics and, in most cases, you should first try to fit linear models. If the linear model output does not adequately reproduce the measured output, you might need to use a nonlinear model.

You can assess the need to use a nonlinear model structure by plotting the response of the system to an input. If you notice that the responses differ depending on the input level or input sign, try using a nonlinear model. For example, if the output response to an input step up is faster than the response to a step down, you might need a nonlinear model.

Before building a nonlinear model of a system that you know is nonlinear, try transforming the input and output variables such that the relationship between the transformed variables is linear. For example, consider a system that has current and voltage as inputs to an immersion heater, and the temperature of the heated liquid as an output. The output depends on the inputs via the power of the heater, which is equal to the product of current and voltage. Instead of building a nonlinear model for this two-input and one-output system, you can create a new input variable by taking the product of current and voltage and then build a linear model that describes the relationship between power and temperature.

If you cannot determine variable transformations that yield a linear relationship between input and output variables, you can use nonlinear structures such as Nonlinear ARX or Hammerstein-Wiener models. For a list of supported nonlinear model structures and when to use them, see “Nonlinear Model Structures” on page 11-7.

Black-Box Estimation Example

You can use the System Identification app or commands to estimate linear and nonlinear models of various structures. In most cases, you choose a model structure and estimate the model parameters using a single command.

Consider the mass-spring-damper system, described in “About Dynamic Systems and Models”. If you do not know the equation of motion of this system, you can use a black-box modeling approach to build a model. For example, you can estimate transfer functions or state-space models by specifying the orders of these model structures.

A transfer function is a ratio of polynomials:

$$G(s) = \frac{\left(b_0 + b_1 s + b_2 s^2 + \ldots\right)}{\left(1 + f_1 s + f_2 s^2 + \ldots\right)}$$

For the mass-spring damper system, this transfer function is:

$$G(s) = \frac{1}{\left(ms^2 + cs + k\right)}$$

which is a system with no zeros and 2 poles.

In discrete-time, the transfer function of the mass-spring-damper system can be:

$$G\left(z^{-1}\right) = \frac{bz^{-1}}{\left(1 + f_1 z^{-1} + f_2 z^{-2}\right)}$$

where the model orders correspond to the number of coefficients of the numerator and the denominator (`nb` = 1 and `nf` = 2) and the input-output delay equals the lowest order exponent of z^{-1} in the numerator (`nk` = 1).

In continuous-time, you can build a linear transfer function model using the `tfest` command:

```
m = tfest(data,2,0)
```

where `data` is your measured input-output data, represented as an `iddata` object and the model order is the set of number of poles (2) and the number of zeros (0).

Similarly, you can build a discrete-time model Output Error structure using the following command:

```
m = oe(data,[1 2 1])
```

The model order is [`nb nf nk`] = [`1 2 1`]. Usually, you do not know the model orders in advance. You should try several model order values until you find the orders that produce an acceptable model.

Alternatively, you can choose a state-space structure to represent the mass-spring-damper system and estimate the model parameters using the `ssest` or the `n4sid` command:

```
m = ssest(data,2)
```

where *order* = 2 represents the number of states in the model.

In black-box modeling, you do not need the system's equation of motion—only a guess of the model orders.

For more information about building models, see "Steps for Using the System Identification App" on page 21-2 and "Model Estimation Commands" on page 1-44.

Refine Linear Parametric Models

In this section...
"When to Refine Models" on page 4-7 "What You Specify to Refine a Model" on page 4-7 "Refine Linear Parametric Models Using System Identification App" on page 4-7 "Refine Linear Parametric Models at the Command Line" on page 4-9

When to Refine Models

There are two situations where you can refine estimates of linear parametric models.

In the first situation, you have already estimated a parametric model and wish to update the values of its free parameters to improve the fit to the estimation data. This is useful if your previous estimation terminated because of search algorithm constraints such as maximum number of iterations or function evaluations allowed reached. However, if your model captures the essential dynamics, it is usually not necessary to continue improving the fit—especially when the improvement is a fraction of a percent.

In the second situation, you might have constructed a model using one of the model constructors described in "Commands for Constructing Linear Model Structures" on page 1-21. In this case, you built initial parameter guesses into the model structure and wish to refine these parameter values.

What You Specify to Refine a Model

When you refine a model, you must provide two inputs:

- Parametric model
- Data — You can either use the same data set for refining the model as the one you originally used to estimate the model, or you can use a different data set.

Refine Linear Parametric Models Using System Identification App

The following procedure assumes that the model you want to refine is already in the System Identification app. You might have estimated this model in the current session or

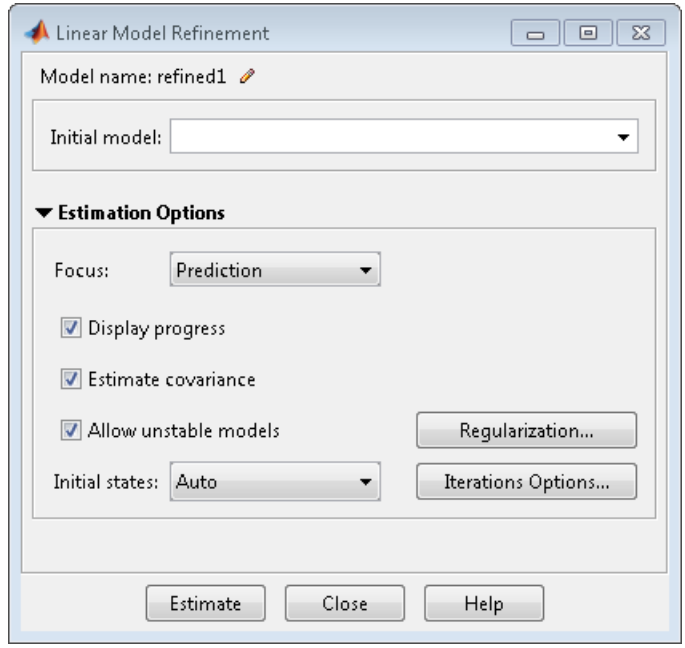
imported the model from the MATLAB workspace. For information about importing models into the app, see “Importing Models into the App” on page 21-7.

To refine your model:

1. In the System Identification app, verify that you have the correct data set in the **Working Data** area for refining your model.

   If you are using a different data set than the one you used to estimate the model, drag the correct data set into the **Working Data** area. For more information about specifying estimation data, see “Specify Estimation and Validation Data in the App” on page 2-30.

2. Select **Estimate** > **Refine Existing Models** to open the Linear Model Refinement dialog box.



   For more information on the options in the dialog box, click **Help**.

3. Select the model you want to refine in the `Initial Model` drop-down list or type the model name.

The model name must be in the Model Board of the System Identification app or a variable in the MATLAB workspace. The model can be a state-space, polynomial, process, transfer function or linear grey-box model. The input-output dimensions of the model must match that of the working data.

4 (Optional) Modify the **Estimation Options**.

When you enter the model name, the estimation options in the Linear Model Refinement dialog box override the initial model settings.

5 Click **Regularization** to obtain regularized estimates of model parameters. Specify the regularization constants in the Regularization Options dialog box. To learn more, see "Regularized Estimates of Model Parameters" on page 1-48.

6 Click **Estimate** to refine the model.

7 Validate the new model. See "Ways to Validate Models" on page 17-3.

Refine Linear Parametric Models at the Command Line

If you are working at the command line, you can use `pem` to refine parametric model estimates. You can also use the various model-structure specific estimators — `ssest` for `idss` models, `polyest` for `idpoly` models, `tfest` for `idtf` models, and `greyest` for `idgrey` models.

The general syntax for refining initial models is as follows:

```
m = pem(data,init_model)
```

`pem` uses the properties of the initial model.

You can also specify the estimation options configuring the objective function and search algorithm settings. For more information, see the reference page of the estimating function.

Refine ARMAX Model with Initial Parameter Guesses at Command Line

This example shows how to refine models for which you have initial parameter guesses.

Estimate an ARMAX model for the data by initializing the A , B , and C polynomials. You must first create a model object and set the initial parameter values in the model properties. Next, you provide this initial model as input to `armax` , `polyest` , or `pem` , which refine the initial parameter guesses using the data.

Load estimation data.

```
load iddata8
```

Define model parameters.

Leading zeros in B indicate input delay (nk), which is 1 for each input channel.

```
A = [1 -1.2 0.7];
B{1} = [0 1 0.5 0.1]; % first input
B{2} = [0 1.5 -0.5]; % second input
B{3} = [0 -0.1 0.5 -0.1]; % third input
C = [1 0 0 0 0];
Ts = 1;
```

Create model object.

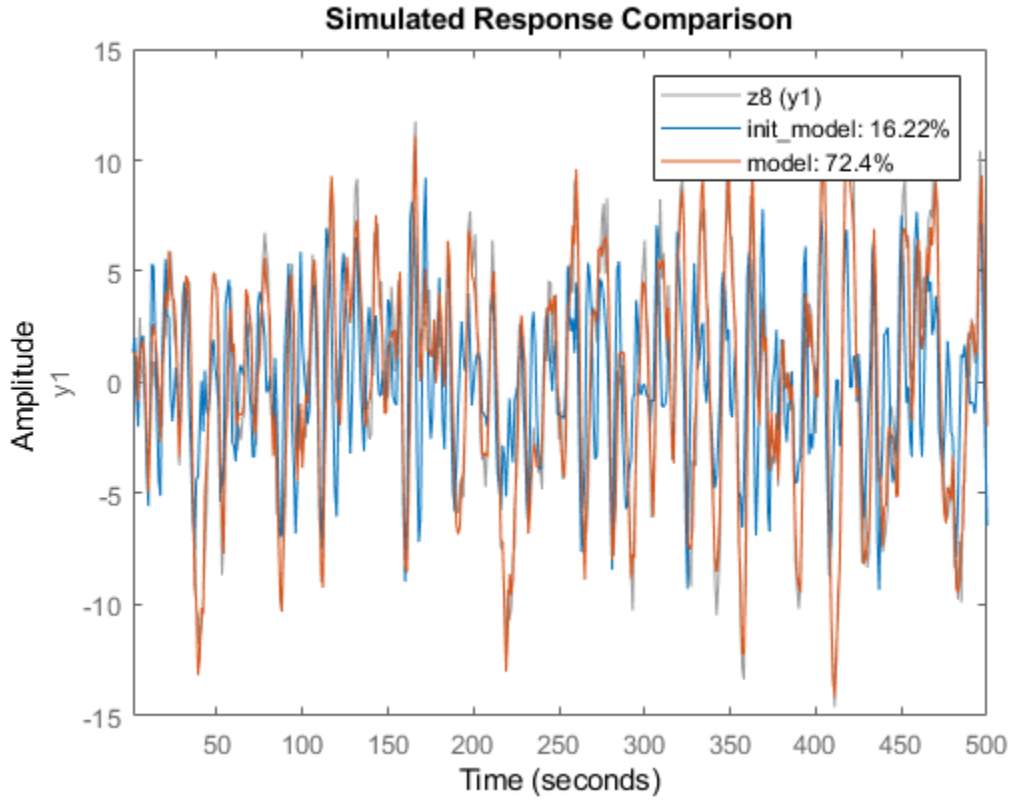
```
init_model = idpoly(A,B,C,'Ts',1);
```

Use polyest to update the parameters of the initial model.

```
model = polyest(z8,init_model);
```

Compare the two models.

```
compare(z8,init_model,model)
```



See Also

"Input-Output Polynomial Models"

Refine Initial ARMAX Model at Command Line

This example shows how to estimate an initial model and refine it using `pem`.

Load measured data.

```
load iddata8
```

Split the data into an initial estimation data set and a refinement data set.

```
init_data = z8(1:100);
refine_data = z8(101:end);
```

`init_data` is an `iddata` object containing the first 100 samples from `z8` and `refine_data` is an `iddata` object representing the remaining data in `z8`.

Estimate an ARMAX model.

```
na = 4;
nb = [3 2 3];
nc = 2;
nk = [0 0 0];

sys = armax(init_data,[na nb nc nk]);
```

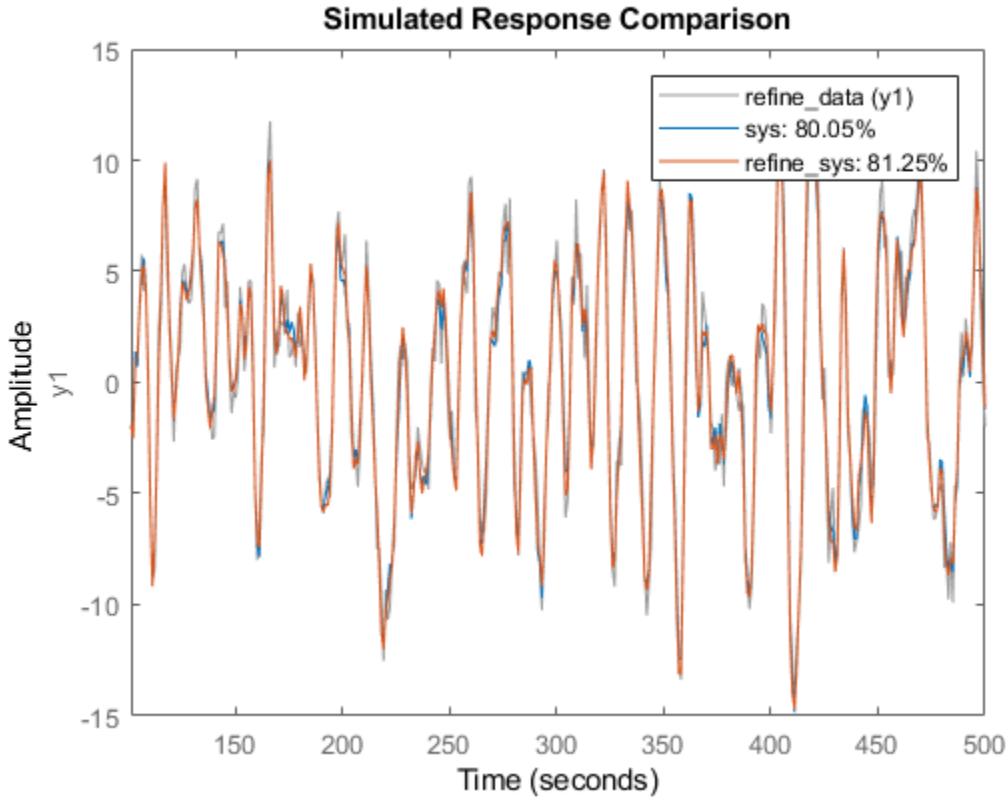
`armax` uses the default algorithm properties to estimate `sys`.

Refine the estimated model by specifying the estimation algorithm options. Specify stricter tolerance and increase the maximum iterations.

```
opt = armaxOptions;
opt.SearchOption.Tolerance = 1e-5;
opt.SearchOption.MaxIter = 50;

refine_sys = pem(refine_data,sys,opt);
```

Compare the fit of the initial and refined models.

```
compare(refine_data,sys,refine_sys)
```



`refine_sys` provides a closer fit to the data than `sys`.

You can similarly use `polyest` or `armax` to refine the estimated model.

See Also

Functions

`armax` | `pem` | `polyest`

Extracting Numerical Model Data

You can extract the following numerical data from linear model objects:

- Coefficients and uncertainty

  For example, extract state-space matrices (`A`, `B`, `C`, `D` and `K`) for state-space models, or polynomials (`A`, `B`, `C`, `D` and `F`) for polynomial models.

  If you estimated model uncertainty data, this information is stored in the model in the form of the parameter covariance matrix. You can fetch the covariance matrix (in its raw or factored form) using the `getcov` command. The covariance matrix represents uncertainties in parameter estimates and is used to compute:

  - Confidence bounds on model output plots, Bode plots, residual plots, and pole-zero plots
  - Standard deviation in individual parameter values. For example, one standard deviation in the estimated value of the `A` polynomial in an ARX model, returned by the `polydata` command and displayed by the `present` command.

  The following table summarizes the commands for extracting model coefficients and uncertainty.

Commands for Extracting Model Coefficients and Uncertainty Data

Command	Description	Syntax
`freqresp`	Extracts frequency-response data (`H`) and corresponding covariance (`CovH`) from any linear identified model.	`[H,w,CovH] = freqresp(m)`
`polydata`	Extracts polynomials (such as `A`) from any linear identified model. The polynomial uncertainties (such as `dA`) are returned only for `idpoly` models.	`[A,B,C,D,F,dA,dB,dC,dD,dF] = ...` `polydata(m)`

`idssdata`	Extracts state-space matrices (such as `A`) from any linear identified model. The matrix uncertainties (such as `dA`) are returned only for `idss` models.	`[A,B,C,D,K,X0,...`
		`dA,dB,dC,dD,dK,dX0] = ...`
		`idssdata(m)`

`tfdata`	Extracts numerator and denominator polynomials (`Num`, `Den`) and their uncertainties (`dnum`, `dden`) from any linear identified model.	`[Num,Den,Ts,dNum,dDen] = ...` `tfdata(m)`
`zpkdata`	Extracts zeros, poles, and gains (`Z`, `P`, `K`) and their covariances (`covZ`, `covP`, `covK`) from any linear identified model.	`[Z,P,K,Ts,covZ,covP,covK] = ...` `zpkdata(m)`

Command	Description	Syntax
`getpvec`	Obtain a list of model parameters and their uncertainties. To access parameter attributes such as values, free status, bounds or labels, use `getpar`.	`pvec = getpvec(m)`
`getcov`	Obtain parameter covariance information	`cov_data = getcov(m)`

You can also extract numerical model data by using dot notation to access model properties. For example, `m.A` displays the *A* polynomial coefficients from model `m`. Alternatively, you can use the `get` command, as follows: `get(m,'A')`.

Tip To view a list of model properties, type `get(model)`.

- Dynamic and noise models

  For linear models, the general symbolic model description is given by:

  $$y = Gu + He$$

  G is an operator that takes the measured inputs *u* to the outputs and captures the system dynamics, also called the *measured model*. *H* is an operator that describes the properties of the additive output disturbance and takes the hypothetical (unmeasured) noise source inputs *e* to the outputs, also called the *noise model*. When you estimate a noise model, the toolbox includes one noise channel *e* for each output in your system.

You can operate on extracted model data as you would on any other MATLAB vectors, matrices and cell arrays. You can also pass these numerical values to Control System Toolbox commands, for example, or Simulink blocks.

Transforming Between Discrete-Time and Continuous-Time Representations

In this section...

"Why Transform Between Continuous and Discrete Time?" on page 4-17
"Using the c2d, d2c, and d2d Commands" on page 4-17
"Specifying Intersample Behavior" on page 4-19
"Effects on the Noise Model" on page 4-19

Why Transform Between Continuous and Discrete Time?

Transforming between continuous-time and discrete-time representations is useful, for example, if you have estimated a discrete-time linear model and require a continuous-time model instead for your application.

You can use `c2d` and `d2c` to transform any linear identified model between continuous-time and discrete-time representations. `d2d` is useful is you want to change the sample time of a discrete-time model. All of these operations change the sample time, which is called *resampling* the model.

These commands do not transform the estimated model uncertainty. If you want to translate the estimated parameter covariance during the conversion, use `translatecov`.

Note `c2d` and `d2d` correctly approximate the transformation of the noise model only when the sample time `T` is small compared to the bandwidth of the noise.

Using the c2d, d2c, and d2d Commands

The following table summarizes the commands for transforming between continuous-time and discrete-time model representations.

Command	Description	Usage Example
`c2d`	Converts continuous-time models to discrete-time models. You cannot use `c2d` for `idproc` models and for `idgrey` models whose `FunctionType` is not `'cd'`. Convert these models into `idpoly`, `idtf`, or `idss` models before calling `c2d`.	To transform a continuous-time model `mod_c` to a discrete-time form, use the following command: `mod_d = c2d(mod_c,T)` where `T` is the sample time of the discrete-time model.
`d2c`	Converts parametric discrete-time models to continuous-time models. You cannot use `d2c` for `idgrey` models whose `FunctionType` is not `'cd'`. Convert these models into `idpoly`, `idtf`, or `idss` models before calling `d2c`.	To transform a discrete-time model `mod_d` to a continuous-time form, use the following command: `mod_c = d2c(mod_d)`
`d2d`	Resample a linear discrete-time model and produce an equivalent discrete-time model with a new sample time. You can use the resampled model to simulate or predict output with a specified time interval.	To resample a discrete-time model `mod_d1` to a discrete-time form with a new sample time `Ts`, use the following command: `mod_d2 = d2d(mod_d1,Ts)`

The following commands compare estimated model `m` and its continuous-time counterpart `mc` on a Bode plot:

```
% Estimate discrete-time ARMAX model
% from the data
m = armax(data,[2 3 1 2]);
```

```
% Convert to continuous-time form
mc = d2c(m);
% Plot bode plot for both models
bode(m,mc)
```

Specifying Intersample Behavior

A sampled signal is characterized only by its values at the sampling instants. However, when you apply a continuous-time input to a continuous-time system, the output values at the sampling instants depend on the inputs at the sampling instants and on the inputs between these points. Thus, the `InterSample` data property describes how the algorithms should handle the input between samples. For example, you can specify the behavior between the samples to be piece-wise constant (zero-order hold, `zoh`) or linearly interpolated between the samples (first order hold, `foh`). The transformation formulas for `c2d` and `d2c` are affected by the intersample behavior of the input.

By default, `c2d` and `d2c` use the intersample behavior you assigned to the estimation data. To override this setting during transformation, add an extra argument in the syntax. For example:

```
% Set first-order hold intersample behavior
mod_d = c2d(mod_c,T,'foh')
```

Effects on the Noise Model

`c2d`, `d2c`, and `d2d` change the sample time of both the dynamic model and the noise model. Resampling a model affects the variance of its noise model.

A parametric noise model is a time-series model with the following mathematical description:

$$y(t) = H(q)e(t)$$

$$Ee^2 = \lambda$$

The noise spectrum is computed by the following discrete-time equation:

$$\Phi_v(\omega) = \lambda T \left| H\left(e^{i\omega T}\right) \right|^2$$

where λ is the variance of the white noise *e(t)*, and λT represents the spectral density of *e(t)*. Resampling the noise model preserves the spectral density λT. The spectral density

λT is invariant up to the Nyquist frequency. For more information about spectrum normalization, see “Spectrum Normalization” on page 9-14.

`d2d` resampling of the noise model affects simulations with noise using `sim`. If you resample a model to a faster sampling rate, simulating this model results in higher noise level. This higher noise level results from the underlying continuous-time model being subject to continuous-time white noise disturbances, which have infinite, instantaneous variance. In this case, the *underlying continuous-time model* is the unique representation for discrete-time models. To maintain the same level of noise after interpolating the noise signal, scale the noise spectrum by $\sqrt{T_{New}/T_{Old}}$, where T_{new} is the new sample time and T_{old} is the original sample time. before applying `sim`.

See Also

More About

- “Continuous-Discrete Conversion Methods” on page 4-21

Continuous-Discrete Conversion Methods

In this section...
“Choosing a Conversion Method” on page 4-21 “Zero-Order Hold” on page 4-22 “First-Order Hold” on page 4-23 “Impulse-Invariant Mapping” on page 4-24 “Tustin Approximation” on page 4-25 “Zero-Pole Matching Equivalents” on page 4-29

Choosing a Conversion Method

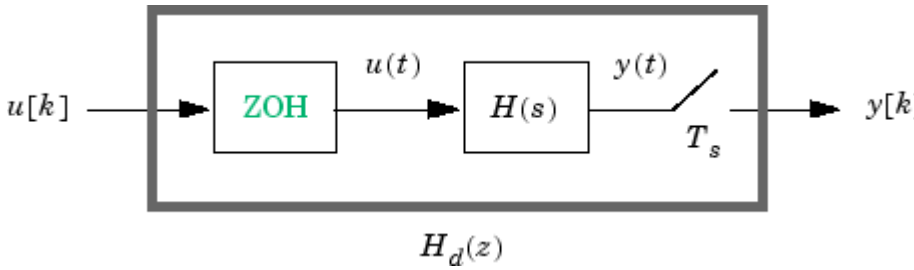
The `c2d` command discretizes continuous-time models. Conversely, `d2c` converts discrete-time models to continuous time. Both commands support several discretization and interpolation methods, as shown in the following table.

Discretization Method	Use when:
“Zero-Order Hold” on page 4-22	You want an exact discretization in the time domain for staircase inputs.
“First-Order Hold” on page 4-23	You want an exact discretization in the time domain for piecewise linear inputs.
“Impulse-Invariant Mapping” on page 4-24 (`c2d` only)	You want an exact discretization in the time domain for impulse train inputs.
“Tustin Approximation” on page 4-25	• 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 4-29	You have a SISO model, and you want good matching in the frequency domain between the continuous- and discrete-time models.

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 $H_d(z)$.

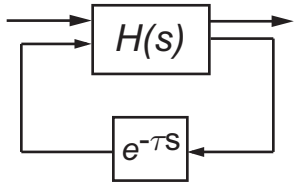
The ZOH discrete-to-continuous conversion has the following limitations:

- `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, the software instead introduces a conjugate pair of complex poles in the s domain.

ZOH Method for Systems with Time Delays

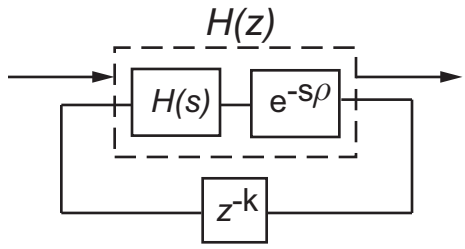
You can use the ZOH method to discretize SISO or MIMO continuous-time models with time delays. The ZOH method yields an exact discretization for systems with input delays, output delays, or transfer delays.

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:

$$u(t) = u[k] + \frac{t - kT_s}{T_s}\big(u[k+1] - u[k]\big), \quad kT_s \le t \le (k+1)T_s$$

This method is generally 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.

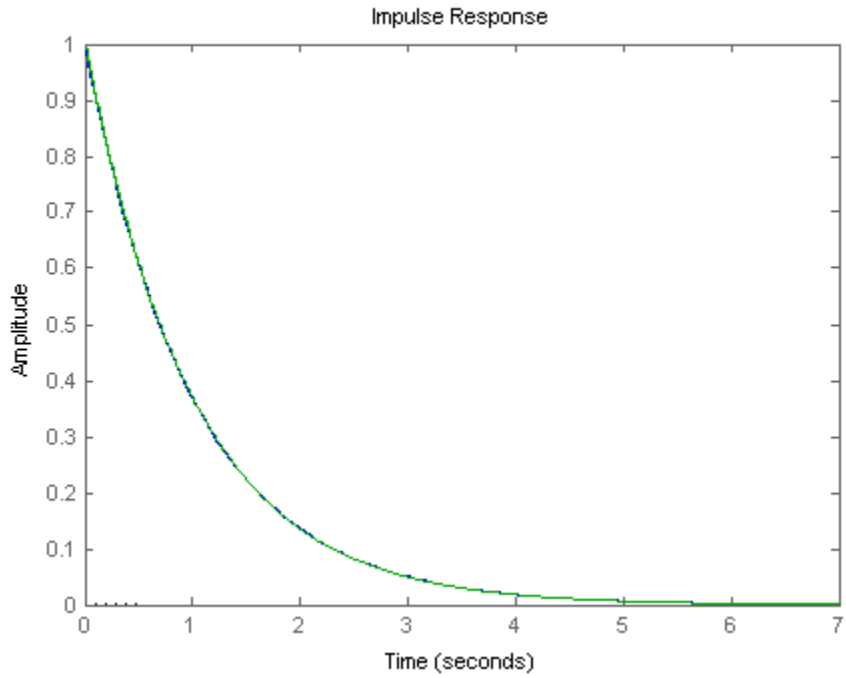
FOH Method for Systems with Time Delays

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 4-22.

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 delay, 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:

$$w[kT_s] = \left(I - A\frac{T_s}{2}\right)x(kT_s) - \frac{T_s}{2}Bu(kT_s) = x(kT_s) - \frac{T_s}{2}\left(Ax(kT_s) + Bu(kT_s)\right).$$

T_s is the sample time of the discrete-time model. A and B are state-space matrices of the continuous-time model.

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:

$$H(j\omega) = H_d\left(e^{j\omega T_s}\right)$$

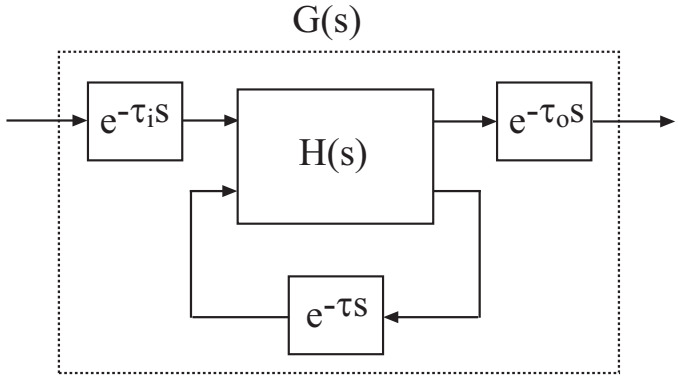
Tustin Approximation for Systems with Time Delays

You can use the Tustin approximation to discretize SISO or MIMO continuous-time models with time delays.

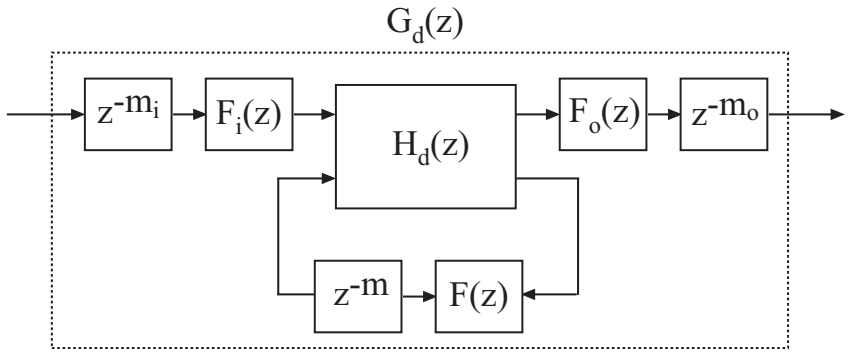
By default, the Tustin method rounds any time delay to the nearest multiple of the sample time. Therefore, for any time delay `tau`, the integer portion of the delay, `k*Ts`, maps to a delay of `k` sampling periods in the discretized model. This approach ignores the residual fractional delay, `tau - 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`.

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 transfer delays into a single transfer 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

The method of conversion by computing 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 4-27.

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. `http://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

`c2d` | `c2dOptions` | `d2c` | `d2cOptions` | `d2d` | `thiran`

Effect of Input Intersample Behavior on Continuous-Time Models

The intersample behavior of the input signals influences the estimation, simulation and prediction of continuous-time models. A sampled signal is characterized only by its values at the sampling instants. However, when you apply a continuous-time input to a continuous-time system, the output values at the sampling instants depend on the inputs at the sampling instants and on the inputs between these points.

The `iddata` and `idfrd` objects have an `InterSample` property which stores how the input behaves between the sampling instants. You can specify the behavior between the samples to be piecewise constant (zero-order hold), linearly interpolated between the samples (first-order hold) or band-limited. A band-limited intersample behavior of the input signal means:

- A filtered input signal (an input of finite bandwidth) was used to excite the system dynamics.
- The input was measured using a sampling device (A/D converter with antialiasing) that reported it to be band-limited even though the true input entering the system was piecewise constant or linear. In this case, the sampling devices can be assumed to be a part of the system being modeled.

When the input signal is a band-limited discrete-time frequency-domain data (`iddata` with `domain = 'frequency'` or `idfrd` with sample time *Ts≠0*), the model estimation is performed by treating the data as continuous-time data (*Ts = 0*). For more information, see Pintelon, R. and J. Schoukens, *System Identification. A Frequency Domain Approach*, section 10.2, pp-352-356,Wiley-IEEE Press, New York, 2001.

The intersample behavior of the input data also affects the results of simulation and prediction of continuous-time models. `sim` and `predict` commands use the `InterSample` property to choose the right algorithm for computing model response.

The following example simulates a system using first-order hold (`foh`) intersample behavior for input signal.

```
sys = idtf([-1 -2],[1 2 1 0.5]);
rng('default')
u = idinput([100 1 5],'sine',[],[],[5 10 1]);
Ts = 2;
y = lsim(sys,u,(0:Ts:999)','foh');
```

Create an `iddata` object for the simulated input-output data.

```
data = iddata(y,u,Ts);
```

The default intersample behavior is zero-order hold (`zoh`).

```
data.InterSample
```

```
ans =
'zoh'
```

Estimate a transfer function using this data.

```
np = 3; % number of poles
nz = 1; % number of zeros
opt = tfestOptions('InitMethod','all','Display','on');
opt.SearchOption.MaxIter = 100;
modelZOH = tfest(data,np,nz,opt)
```

```
modelZOH =

  From input "u1" to output "y1":
          -217.2 s - 391.6
  ---------------------------------
  s^3 + 354.4 s^2 + 140.2 s + 112.4

Continuous-time identified transfer function.

Parameterization:
   Number of poles: 3   Number of zeros: 1
   Number of free coefficients: 5
   Use "tfdata", "getpvec", "getcov" for parameters and their uncertainties.

Status:
Estimated using TFEST on time domain data "data".
Fit to estimation data: 81.38%
FPE: 0.1146, MSE: 0.111
```

The model gives about 80% fit to data. The sample time of the data is large enough that intersample inaccuracy (using `zoh` rather than `foh`) leads to significant modeling errors.

Re-estimate the model using `foh` intersample behavior.

```
data.InterSample = 'foh';
modelFOH = tfest(data,np,nz,opt)
```

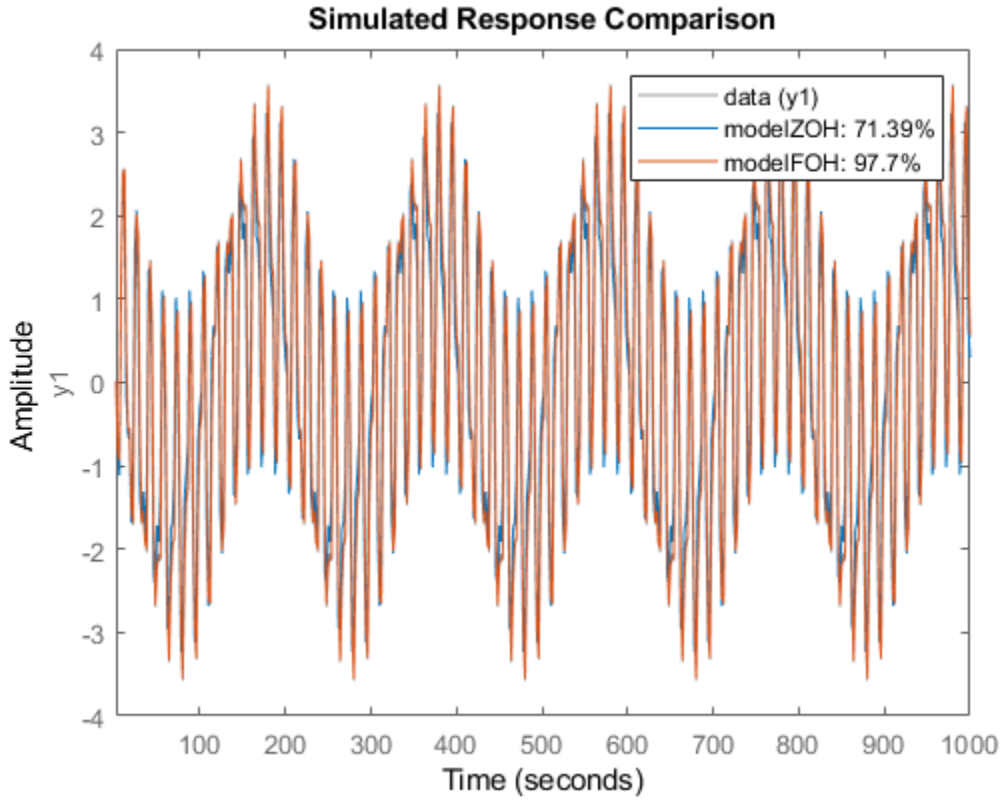
```
modelFOH =
```

```
  From input "u1" to output "y1":
           -1.197 s - 0.06843
  -------------------------------------
  s^3 + 0.4824 s^2 + 0.3258 s + 0.01723

Continuous-time identified transfer function.

Parameterization:
   Number of poles: 3   Number of zeros: 1
   Number of free coefficients: 5
   Use "tfdata", "getpvec", "getcov" for parameters and their uncertainties.

Status:
Estimated using TFEST on time domain data "data".
Fit to estimation data: 97.7%
FPE: 0.001748, MSE: 0.001693
```

`modelFOH` is able to retrieve the original system correctly.

Compare the model outputs with data.

```
compare(data,modelZOH,modelFOH)
```



`modelZOH` is compared to data whose intersample behavior is `foh`. Therefore, its fit decreases to around 70%.

See Also

`iddata` | `idfrd`

More About

- “Frequency Domain Identification: Estimating Models Using Frequency Domain Data” on page 4-65

Transforming Between Linear Model Representations

You can transform linear models between state-space and polynomial forms. You can also transform between frequency-response, state-space, and polynomial forms.

If you used the System Identification app to estimate models, you must export the models to the MATLAB workspace before converting models.

For detailed information about each command in the following table, see the corresponding reference page.

Commands for Transforming Model Representations

Command	Model Type to Convert	Usage Example

`idfrd`	Converts any linear model to an `idfrd` model.
	
	If you have the Control System Toolbox product, this command converts any numeric LTI model too.
		To get frequency response of `m` at default frequencies, use the following command:
		
		`m_f = idfrd(m)`
		
		To get frequency response at specific frequencies, use the following command:
		
		`m_f = idfrd(m,f)`
		
		To get frequency response for a submodel from input `2` to output `3`, use the following command:
		
		`m_f = idfrd(m(2,3))`

`idpoly`	Converts any linear identified model, except `idfrd`, to ARMAX representation if the original model has a nontrivial noise component, or OE if the noise model is trivial (`H = 1`). If you have the Control System Toolbox product, this command converts any numeric LTI model, except `frd`.	To get an ARMAX model from state-space model `m_ss`, use the following command: `m_p = idpoly(m_ss)`
`idss`	Converts any linear identified model, except `idfrd`, to state-space representation. If you have the Control System Toolbox product, this command converts any numeric LTI model, except `frd`.	To get a state-space model from an ARX model `m_arx`, use the following command: `m_ss = idss(m_arx)`

Command	Model Type to Convert	Usage Example
`idtf`	Converts any linear identified model, except `idfrd`, to transfer function representation. The noise component of the original model is lost since an `idtf` object has no elements to model noise dynamics. If you have the Control System Toolbox product, this command converts any numeric LTI model, except `frd`.	To get a transfer function from a state-space model `m_ss`, use the following command: `m_tf = idtf(m_ss)`

Note Most transformations among identified models (among `idss`, `idtf`, `idpoly`) causes the parameter covariance information to be lost, with few exceptions:

- Conversion of an `idtf` model to an `idpoly` model.
- Conversion of an `idgrey` model to an `idss` model.

If you want to translate the estimated parameter covariance during conversion, use `translatecov`.

Subreferencing Models

In this section...

"What Is Subreferencing?" on page 4-38
"Limitation on Supported Models" on page 4-38
"Subreferencing Specific Measured Channels" on page 4-38
"Separation of Measured and Noise Components of Models" on page 4-39

What Is Subreferencing?

You can use subreferencing to create models with subsets of inputs and outputs from existing multivariable models. Subreferencing is also useful when you want to generate model plots for only certain channels, such as when you are exploring multiple-output models for input channels that have minimal effect on the output.

The toolbox supports subreferencing operations for `idtf`, `idpoly`, `idproc`, `idss`, and `idfrd` model objects.

Subreferencing is not supported for `idgrey` models. If you want to analyze the sub-model, convert it into an `idss` model first, and then subreference the I/Os of the `idss` model. If you want a grey-box representation of a subset of I/Os, create a new `idgrey` model that uses an ODE function returning the desired I/O dynamics.

In addition to subreferencing the model for specific combinations of measured inputs and output, you can subreference dynamic and noise models individually.

Limitation on Supported Models

Subreferencing nonlinear models is not supported.

Subreferencing Specific Measured Channels

Use the following general syntax to subreference specific input and output channels in models:

```
model(outputs,inputs)
```

In this syntax, `outputs` and `inputs` specify channel indexes or channel names.

To select all output or all input channels, use a colon (`:`). To select no channels, specify an empty matrix (`[]`). If you need to reference several channel names, use a cell array of character vectors.

For example, to create a new model `m2` from `m` from inputs `1` (`'power'`) and `4` (`'speed'`) to output number `3` (`'position'`), use either of the following equivalent commands:

```
m2 = m('position',{'power','speed'})
```

or

```
m2 = m(3,[1 4])
```

For a single-output model, you can use the following syntax to subreference specific input channels without ambiguity:

```
m3 = m(inputs)
```

Similarly, for a single-input model, you can use the following syntax to subreference specific output channels:

```
m4 = m(outputs)
```

Separation of Measured and Noise Components of Models

For linear models, the general symbolic model description is given by:

$$y = Gu + He$$

G is an operator that takes the measured inputs u to the outputs and captures the system dynamics.

H is an operator that describes the properties of the additive output disturbance and takes the hypothetical (unmeasured) noise source inputs to the outputs. H represents the noise model. When you specify to estimate a noise model, the resulting model include one noise channel e at the input for each output in your system.

Thus, linear, parametric models represent input-output relationships for two kinds of input channels: measured inputs and (unmeasured) noise inputs. For example, consider the ARX model given by one of the following equations:

$$A(q)y(t) = B(q)u\left(t - nk\right) + e(t)$$

or

$$y(t) = \frac{B(q)}{A(q)}u(t) + \frac{1}{A(q)}e(t)$$

In this case, the dynamic model is the relationship between the measured input u and output y, $G = {B(q)}/{A(q)}$. The noise model is the contribution of the input noise e to the output y, given by $H = {1}/{A(q)}$.

Suppose that the model `m` contains both a dynamic model G and a noise model H. To create a new model that only has G and no noise contribution, simply set its `NoiseVariance` property value to zero value.

To create a new model by subreferencing H due to unmeasured inputs, use the following syntax:

```
m_H = m(:,[])
```

This operation creates a time-series model from `m` by ignoring the measured input.

The covariance matrix of e is given by the model property `NoiseVariance`, which is the matrix Λ:

$$\Lambda = LL^T$$

The covariance matrix of e is related to v, as follows:

$$e = Lv$$

where v is white noise with an identity covariance matrix representing independent noise sources with unit variances.

See Also

More About

- “Treating Noise Channels as Measured Inputs” on page 4-41

Treating Noise Channels as Measured Inputs

A linear models is given by:

$$y = Gu + He$$

Where G is an operator that takes the measured inputs u to the outputs and captures the system dynamics. H is an operator that describes the properties of the additive output disturbance and takes the hypothetical (unmeasured) noise source inputs to the outputs. H represents the noise model. When you specify to estimate a noise model, the resulting model include one noise channel e at the input for each output in your system.

To study noise contributions in more detail, it might be useful to convert the noise channels to measured channels using `noisecnv`:

```
m_GH = noisecnv(m)
```

This operation creates a model `m_GH` that represents both measured inputs `u` and noise inputs e, treating both sources as measured signals. `m_GH` is a model from `u` and e to `y`, describing the transfer functions G and H.

Converting noise channels to measured inputs loses information about the variance of the innovations `e`. For example, step response due to the noise channels does not take into consideration the magnitude of the noise contributions. To include this variance information, normalize e such that v becomes white noise with an identity covariance matrix, where

$$e = Lv$$

To normalize e, use the following command:

```
m_GH = noisecnv(m,'Norm')
```

This command creates a model where u and v are treated as measured signals, as follows:

$$y(t) = Gu(t) + HLv = \begin{bmatrix} G & HL \end{bmatrix} \begin{bmatrix} u \\ v \end{bmatrix}$$

For example, the scaling by L causes the step responses from v to y to reflect the size of the disturbance influence.

The converted noise sources are named in a way that relates the noise channel to the corresponding output. Unnormalized noise sources e are assigned names such as `'e@y1'`,

`'e@y2'`, ..., `'e@yn'`, where `'e@yn'` refers to the noise input associated with the output `yn`. Similarly, normalized noise sources `v`, are named `'v@y1'`, `'v@y2'`, ..., `'v@yn'`.

If you want to create a model that has only the noise channels of an identified model as its measured inputs, use the `noise2meas` command. It results in a model with $y(t) = He$ or $y(t) = HLv$, where e or v is treated as a measured input.

Note When you plot models in the app that include noise sources, you can select to view the response of the noise model corresponding to specific outputs. For more information, see "Selecting Measured and Noise Channels in Plots" on page 21-14.

See Also

`noise2meas` | `noisecnv`

More About

- "Subreferencing Models" on page 4-38

Concatenating Models

In this section...
“About Concatenating Models” on page 4-43
“Limitation on Supported Models” on page 4-43
“Horizontal Concatenation of Model Objects” on page 4-44
“Vertical Concatenation of Model Objects” on page 4-44
“Concatenating Noise Spectrum Data of idfrd Objects” on page 4-45
“See Also” on page 4-46

About Concatenating Models

You can perform horizontal and vertical concatenation of linear model objects to grow the number of inputs or outputs in the model.

When you concatenate identified models, such as `idtf`, `idpoly`, `idproc`, and `idss` model objects, the resulting model combines the parameters of the individual models. However, the estimated parameter covariance is lost. If you want to translate the covariance information during concatenation, use `translatecov`.

Concatenation is not supported for `idgrey` models; convert them to `idss` models first if you want to perform concatenation.

You can also concatenate nonparametric models, which contain the estimated impulse-response (`idtf` object) and frequency-response (`idfrd` object) of a system.

In case of `idfrd` models, concatenation combines information in the `ResponseData` properties of the individual model objects. `ResponseData` is an `ny`-by-`nu`-by-`nf` array that stores the response of the system, where `ny` is the number of output channels, `nu` is the number of input channels, and `nf` is the number of frequency values. The `(j,i,:)` vector of the resulting response data represents the frequency response from the `i`th input to the `j`th output at all frequencies.

Limitation on Supported Models

Concatenation is supported for linear models only.

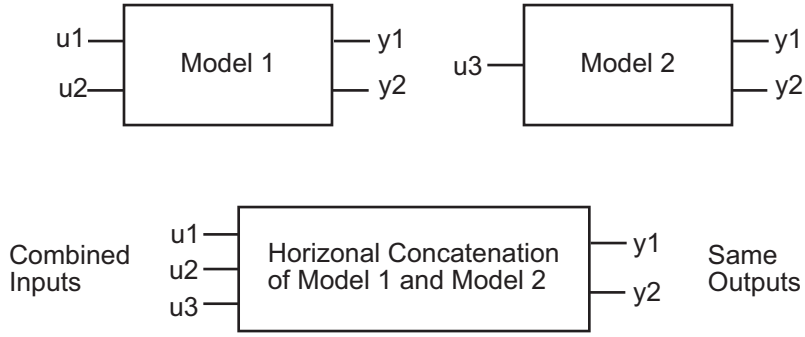
Horizontal Concatenation of Model Objects

Horizontal concatenation of model objects requires that they have the same outputs. If the output channel names are different and their dimensions are the same, the concatenation operation resets the output names to their default values.

The following syntax creates a new model object `m` that contains the horizontal concatenation of `m1,m2,...,mN`:

```
m = [m1,m2,...,mN]
```

`m` takes all of the inputs of `m1,m2,...,mN` to the same outputs as in the original models. The following diagram is a graphical representation of horizontal concatenation of the models.



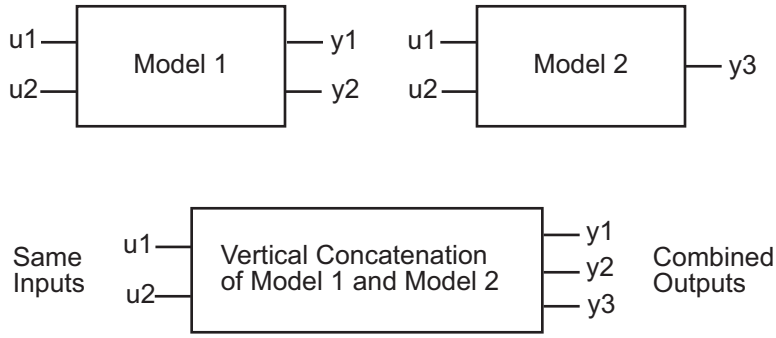
Vertical Concatenation of Model Objects

Vertical concatenation combines output channels of specified models. Vertical concatenation of model objects requires that they have the same inputs. If the input channel names are different and their dimensions are the same, the concatenation operation resets the input channel names to their default (`''`) values.

The following syntax creates a new model object `m` that contains the vertical concatenation of `m1,m2,...,mN`:

```
m = [m1;m2;... ;mN]
```

`m` takes the same inputs in the original models to all of the output of `m1,m2,...,mN`. The following diagram is a graphical representation of vertical concatenation of frequency-response data.



Concatenating Noise Spectrum Data of idfrd Objects

When `idfrd` models are obtained as a result of estimation (such as using `spa`), the `SpectrumData` property is not empty and contains the power spectra and cross spectra of the output noise in the system. For each output channel, this toolbox estimates one noise channel to explain the difference between the output of the model and the measured output.

When the `SpectrumData` property of individual `idfrd` objects is not empty, horizontal and vertical concatenation handle `SpectrumData`, as follows.

In case of horizontal concatenation, there is no meaningful way to combine the `SpectrumData` of individual `idfrd` objects, and the resulting `SpectrumData` property is empty. An empty property results because each `idfrd` object has its own set of noise channels, where the number of noise channels equals the number of outputs. When the resulting `idfrd` object contains the same output channels as each of the individual `idfrd` objects, it cannot accommodate the noise data from all the `idfrd` objects.

In case of vertical concatenation, this toolbox concatenates individual noise models diagonally. The following shows that `m.SpectrumData` is a block diagonal matrix of the power spectra and cross spectra of the output noise in the system:

$$m.s = \begin{pmatrix} m1.s & & 0 \\ & \ddots & \\ 0 & & mN.s \end{pmatrix}$$

`s` in `m.s` is the abbreviation for the `SpectrumData` property name.

See Also

If you have the Control System Toolbox product, see “Combining Model Objects” on page 19-5 about additional functionality for combining models.

Merging Models

You can merge models of the same structure to obtain a single model with parameters that are statistically weighed means of the parameters of the individual models. When computing the merged model, the covariance matrices of the individual models determine the weights of the parameters.

You can perform the merge operation for the `idtf`, `idgrey`, `idpoly`, `idproc`, and `idss` model objects.

Note Each merge operation merges the same type of model object.

Merging models is an alternative to merging data sets into a single multiexperiment data set, and then estimating a model for the merged data. Whereas merging data sets assumes that the signal-to-noise ratios are about the same in the two experiments, merging models allows greater variations in model uncertainty, which might result from greater disturbances in an experiment.

When the experimental conditions are about the same, merge the data instead of models. This approach is more efficient and typically involves better-conditioned calculations. For more information about merging data sets into a multiexperiment data set, see “Create Multiexperiment Data at the Command Line” on page 2-61.

For more information about merging models, see the `merge` reference page.

Determining Model Order and Delay

Estimation requires you to specify the model order and delay. Many times, these values are not known. You can determine the model order and delay in one of the following ways:

- Guess their values by visually inspecting the data or based on the prior knowledge of the system.
- Estimate delay as a part of `idproc` or `idtf` model estimation. These models treat delay as an estimable parameter and you can determine their values by the estimation commands `procest` and `tfest`, respectively. However automatic estimation of delays can cause errors. Therefore, it is recommended that you analyze the data for delays in advance.
- To estimate delays, you can also use one of the following tools:
  - Estimate delay using `delayest`. The choice of the order of the underlying ARX model and the lower/upper bound on the value of the delay to be estimated influence the value returned by `delayest`.
  - Compute impulse response using `impulseest`. Plot the impulse response with a confidence interval of sufficient standard deviations (usually 3). The delay is indicated by the number of response samples that are inside the statistically zero region (marked by the confidence bound) before the response goes outside that region.
  - Select the model order in `n4sid` by specifying the model order as a vector.
  - Choose the model order of an ARX model using `arxstruc` or `ivstruc` and `selstruc`. These command select the number of poles, zeros and delay.

See “Model Structure Selection: Determining Model Order and Input Delay” on page 4-49 for an example of using these tools.

Model Structure Selection: Determining Model Order and Input Delay

This example shows some methods for choosing and configuring the model structure. Estimation of a model using measurement data requires selection of a model structure (such as state-space or transfer function) and its order (e.g., number of poles and zeros) in advance. This choice is influenced by prior knowledge about the system being modeled, but can also be motivated by an analysis of data itself. This example describes some options for determining model orders and input delay.

Introduction

Choosing a model structure is usually the first step towards its estimation. There are various possibilities for structure - state-space, transfer functions and polynomial forms such as ARX, ARMAX, OE, BJ etc. If you do not have detailed prior knowledge of your system, such as its noise characteristics and indication of feedback, the choice of a reasonable structure may not be obvious. Also for a given choice of structure, the order of the model needs to be specified before the corresponding parameters are estimated. System Identification Toolbox™ offers some tools to assist in the task of model order selection.

The choice of a model order is also influenced by the amount of delay. A good idea of the input delay simplifies the task of figuring out the orders of other model coefficients. Discussed below are some options for input delay determination and model structure and order selection.

Choosing and Preparing Example Data for Analysis

This example uses the hair dryer data, also used by iddemo1 ("Estimating Simple Models from Real Laboratory Process Data"). The process consists of air being fanned through a tube. The air is heated at the inlet of the tube, and the input is the voltage applied to the heater. The output is the temperature at the outlet of the tube.

Let us begin by loading the measurement data and doing some basic preprocessing:

```
load dry2
```

Form a data set for estimation of the first half, and a reference set for validation purposes of the second half:

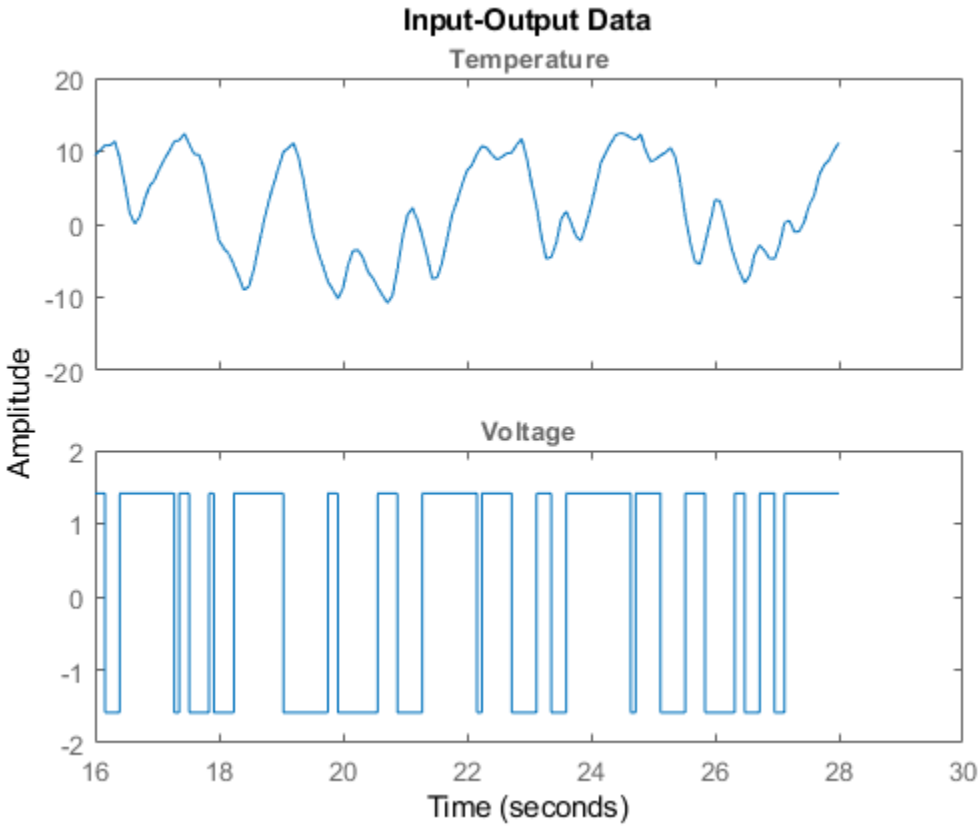
```
ze = dry2(1:500);
zr = dry2(501:1000);
```

Detrend each of the sets:

```
ze = detrend(ze);
zr = detrend(zr);
```

Let us look at a portion of the estimation data:

```
plot(ze(200:350))
```



Estimating Input Delay

There are various options available for determining the time delay from input to output. These are:

- Using the DELAYEST utility.
- Using a non-parametric estimate of the impulse response, using IMPULSEEST.
- Using the state-space model estimator N4SID with a number of different orders and finding the delay of the 'best' one.

Using `delayest`:

Let us discuss the above options in detail. Function `delayest` returns an estimate of the delay for a given choice of orders of numerator and denominator polynomials. This function evaluates an ARX structure:

```
y(t) + a1*y(t-1) + ... + ana*y(t-na) = b1*u(t-nk) + ...+bnb*u(t-nb-
nk+1)
```

with various delays and chooses the delay value that seems to return the best fit. In this process, chosen values of `na` and `nb` are used.

```
delay = delayest(ze) % na = nb = 2 is used, by default
```

```
delay =

     3
```

A value of 3 is returned by default. But this value may change a bit if the assumed orders of numerator and denominator polynomials (2 here) is changed. For example:

```
delay = delayest(ze,5,4)
```

```
delay =

     2
```

returns a value of 2. To gain insight into how `delayest` works, let us evaluate the loss function for various choices of delays explicitly. We select a second order model (`na=nb=2`), which is the default for `delayest`, and try out every time delay between 1 and 10. The loss function for the different models are computed using the validation data set:

```
V = arxstruc(ze,zr,struc(2,2,1:10));
```

We now select that delay that gives the best fit for the validation data:

```
[nn,Vm] = selstruc(V,0); % nn is given as [na nb nk]
```

The chosen structure was:

```
nn
```

```
nn =

     2     2     3
```

which show the best model has a delay of nn(3) = 3.

We can also check how the fit depends on the delay. This information is returned in the second output `Vm`. The logarithms of a quadratic loss function are given as the first row, while the indexes `na`, `nb` and `nk` are given as a column below the corresponding loss function.

```
Vm
```

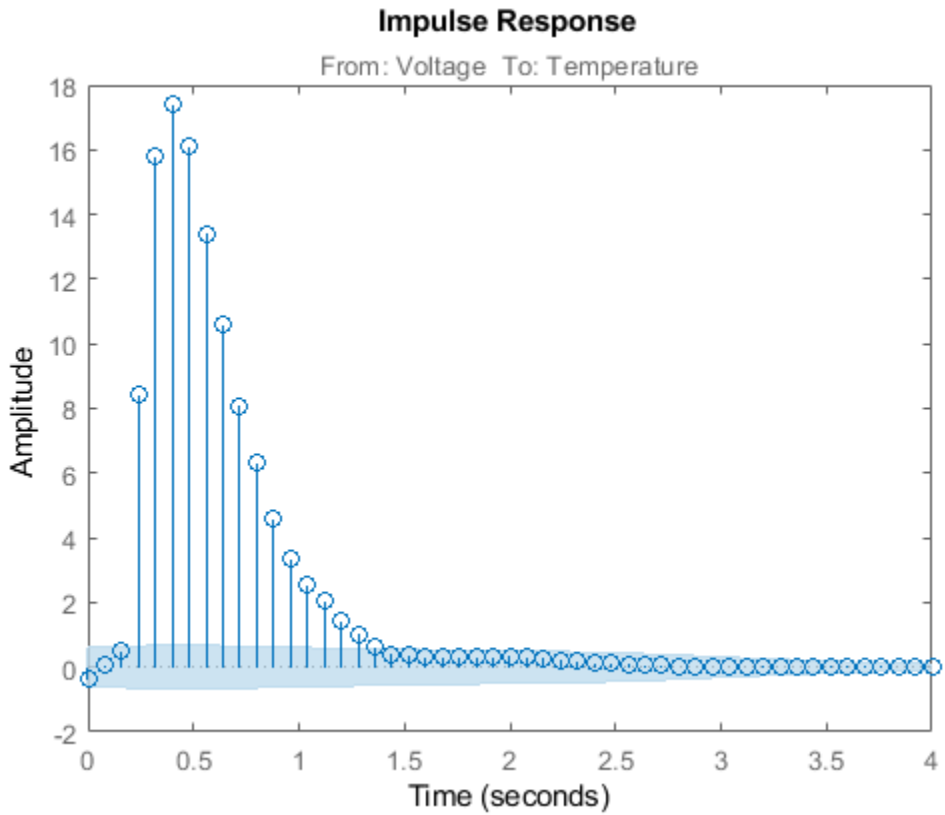
```
Vm =

  Columns 1 through 7

   -0.1480   -1.3275   -1.8747   -0.2403   -0.0056    0.0736    0.1763
    2.0000    2.0000    2.0000    2.0000    2.0000    2.0000    2.0000
    2.0000    2.0000    2.0000    2.0000    2.0000    2.0000    2.0000
    1.0000    2.0000    3.0000    4.0000    5.0000    6.0000    7.0000

  Columns 8 through 10

    0.1906    0.1573    0.1474
    2.0000    2.0000    2.0000
    2.0000    2.0000    2.0000
    8.0000    9.0000   10.0000
```

```
The choice of 3 delays is thus rather clear, since the corresponding
loss is minimum.
```

Using `impulse`

To gain a better insight into the dynamics, let us compute the impulse response of the system. We will use the function `impulseest` to compute a non-parametric impulse response model. We plot this response with a confidence interval represented by 3 standard deviations.

```
FIRModel = impulseest(ze);
clf
h = impulseplot(FIRModel);
showConfidence(h,3)
```

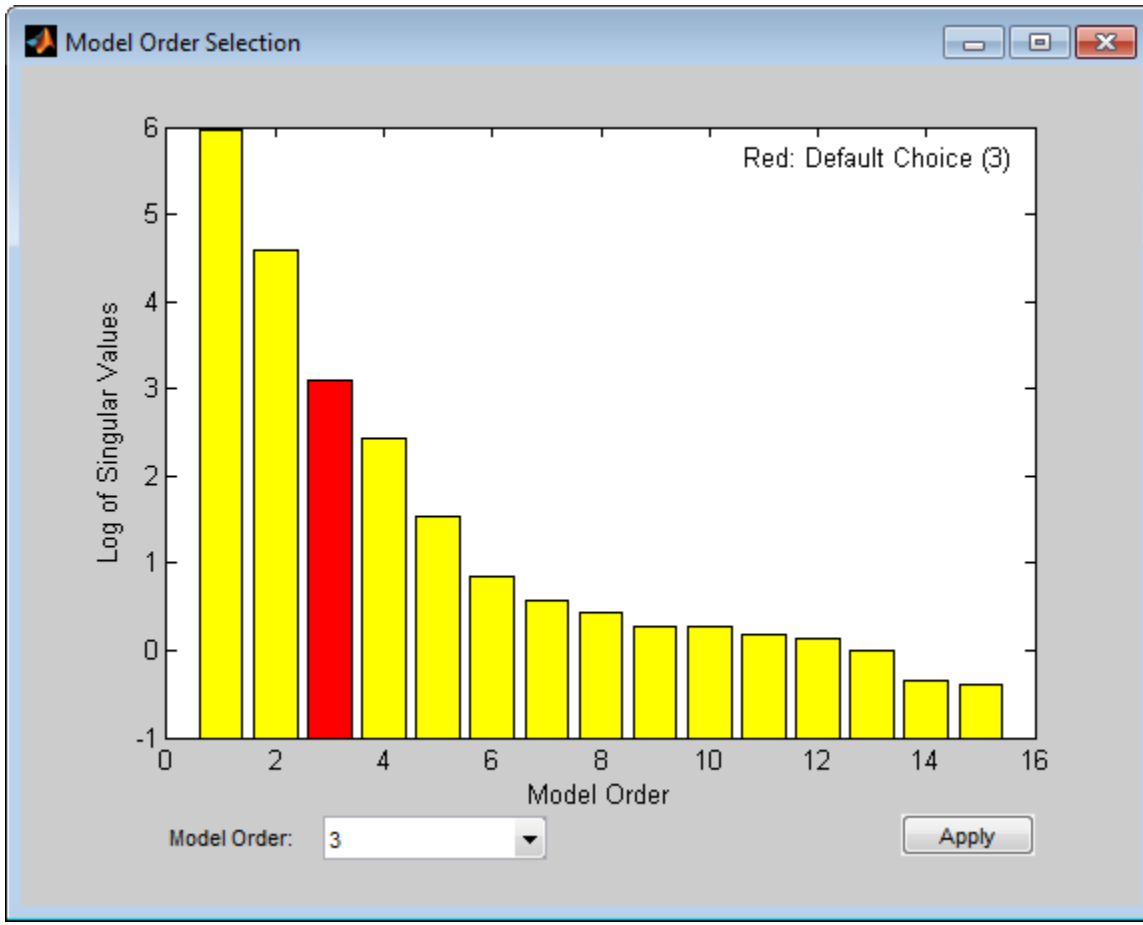


The filled light-blue region shows the confidence interval for the insignificant response in this estimation. There is a clear indication that the impulse response "takes off" (leaves the uncertainty region) after 3 samples. This points to a delay of three intervals.

Using `n4sid` based state-space evaluation

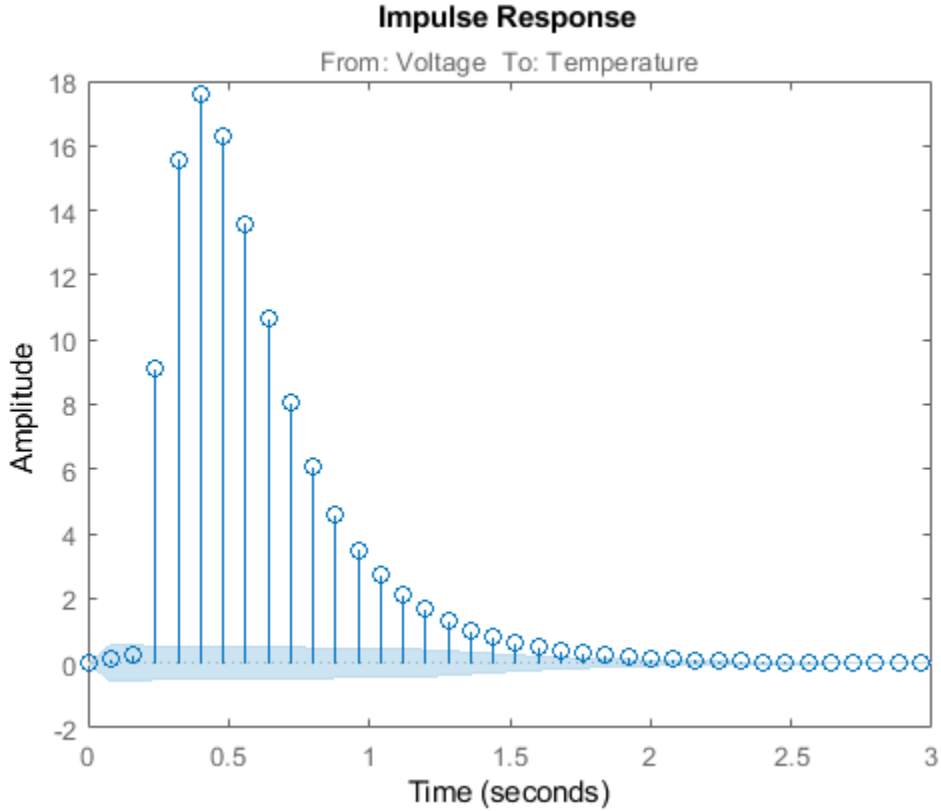
We may also estimate a family of parametric models to find the delay corresponding to the "best" model. In case of state-space models, a range of orders may be evaluated simultaneously and the best order picked from a Hankel Singular Value plot. Execute the following command to invoke `n4sid` in an interactive mode:

```
m = n4sid(ze,1:15); % All orders between 1 and 15.
```



The plot indicates an order of 3 as the best value. For this choice, let us compute the impulse response of the model `m`:

```
m = n4sid(ze, 3);
showConfidence(impulseplot(m),3)
```



As with non-parametric impulse response, there is a clear indication that the delay from input to output is of three samples.

Choosing a Reasonable Model Structure

In lack of any prior knowledge, it is advisable to try out various available choices and use the one that seems to work the best. State-space models may be a good starting point since only the number of states needs to be specified in order to estimate a model. Also, a range of orders may be evaluated quickly, using `n4sid`, for determining the best order, as described in the next section. For polynomial models, a similar advantage is realized

using the arx estimator. Output-error (OE) models may also be good choice for a starting polynomial model because of their simplicity.

Determining Model Order

Once you have decided upon a model structure to use, the next task is to determine the order(s). In general, the aim should be to not use a model order higher than necessary. This can be determined by analyzing the improvement in %fit as a function of model order. When doing this, it is advisable to use a separate, independent dataset for validation. Choosing an independent validation data set (zr in our example) would improve the detection of over-fitting.

In addition to a progressive analysis of multiple model orders, explicit determination of optimum orders can be performed for some model structures. Functions arxstruc and selstruc may be used for choosing the best order for ARX models. For our example, let us check the fit for all 100 combinations of up to 10 b-parameters and up to 10 a-parameters, all with a delay value of 3:

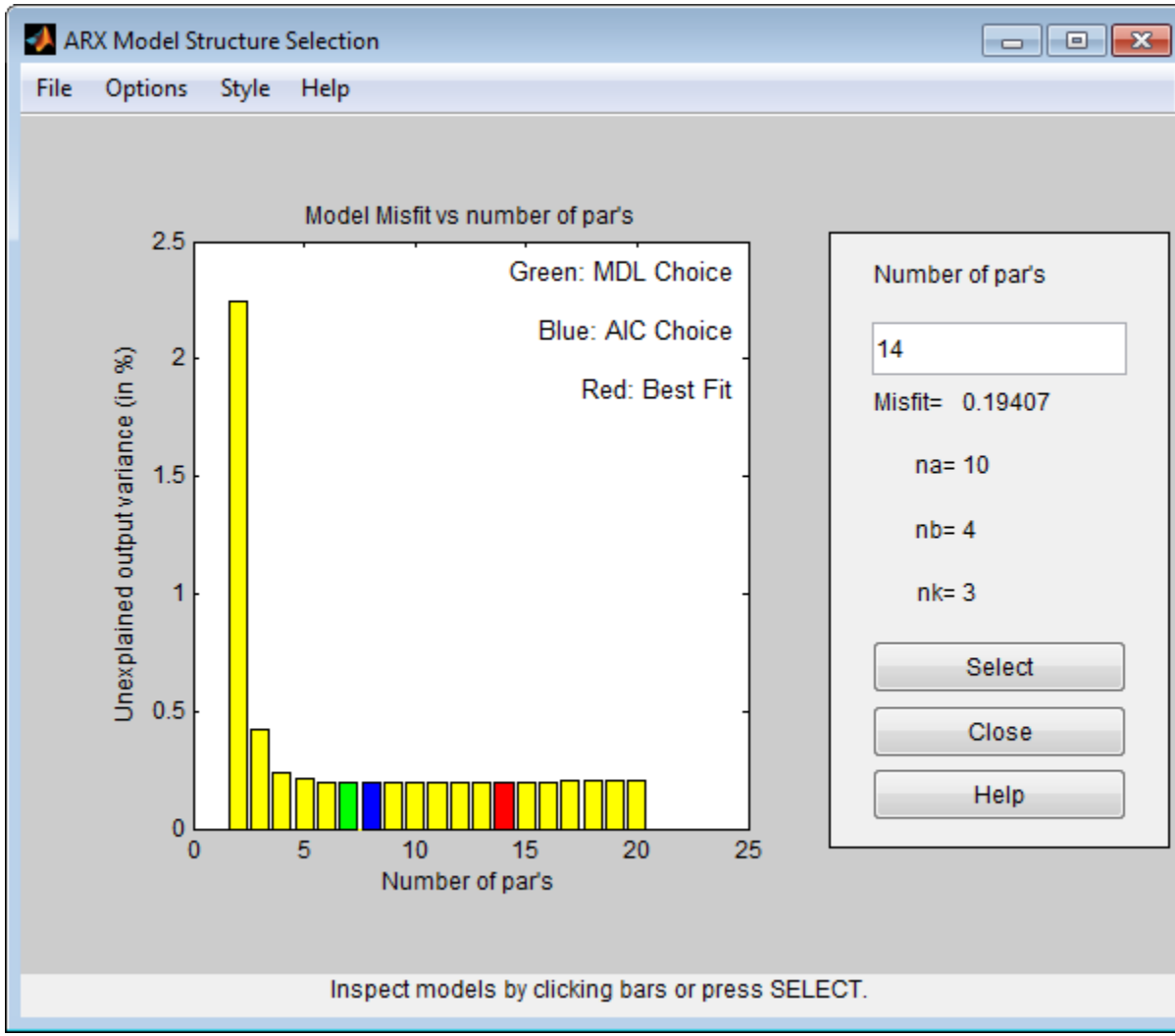
```
V = arxstruc(ze,zr,struc(1:10,1:10,3));
```

The best fit for the validation data set is obtained for:

```
nn = selstruc(V,0)


nn =

    10     4     3
```

Let us check how much the fit is improved for the higher order models. For this, we use the function selstruc with only one input. In this case, a plot showing the fit as a function of the number of parameters used is generated. The user is also prompted to enter the number of parameters. The routine then selects a structure with these many parameters that gives the best fit. Note that several different model structures use the same number of parameters. Execute the following command to choose a model order interactively:

```
nns = selstruc(V) %invoke selstruc in an interactive mode
```



The best fit is thus obtained for nn = [4 4 3], while we see that the improved fit compared to nn = [2 2 3] is rather marginal.

We may also approach this problem from the direction of reducing a higher order model. If the order is higher than necessary, then the extra parameters are basically used to "model" the measurement noise. These "extra" poles are estimated with a lower level of

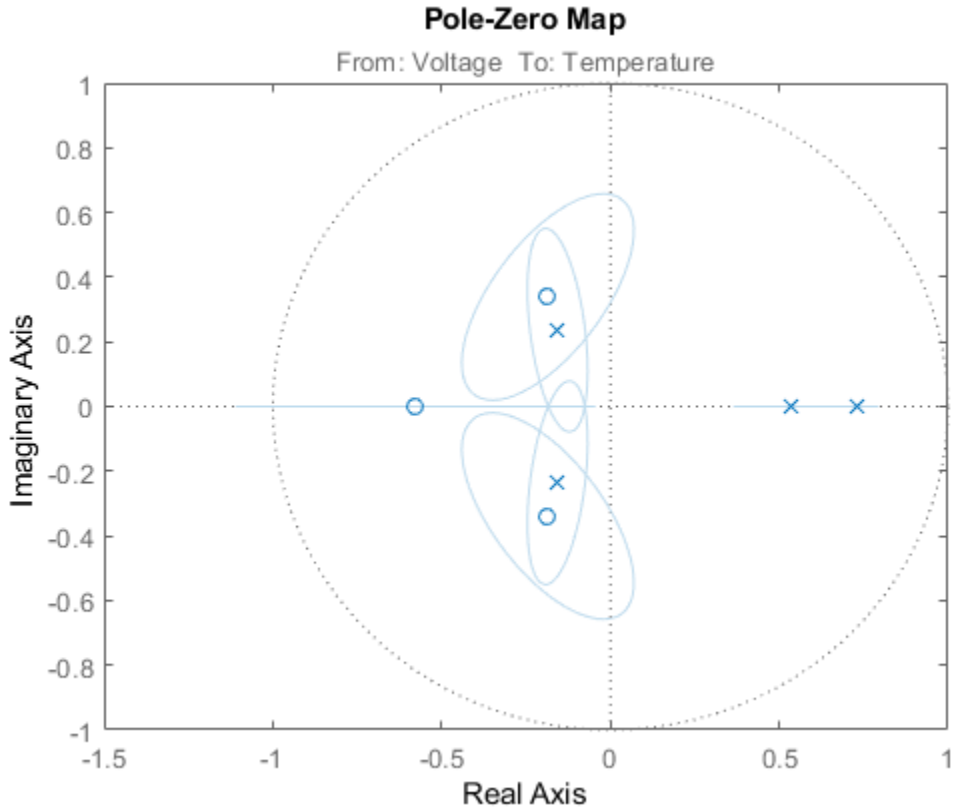
accuracy (large confidence interval). If their are cancelled by a zero located nearby, then it is an indication that this pole-zero pair may not be required to capture the essential dynamics of the system.

For our example, let us compute a 4th order model:

```
th4 = arx(ze,[4 4 3]);
```

Let us check the pole-zero configuration for this model. We can also include confidence regions for the poles and zeros corresponding to 3 standard deviations, in order to determine how accurately they are estimated and also how close the poles and zeros are to each other.

```
h = iopzplot(th4);
showConfidence(h,3)
```

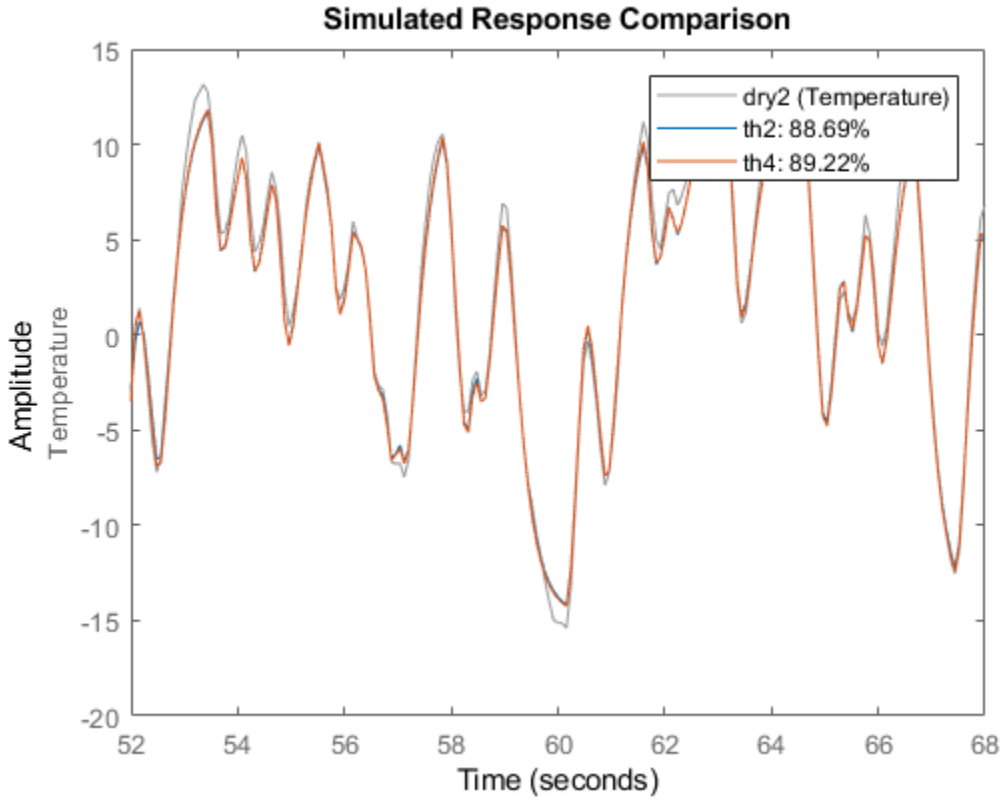


The confidence intervals for the two complex-conjugate poles and zeros overlap, indicating they are likely to cancel each other. Hence, a second order model might be adequate. Based on this evidence, let us compute a 2nd order ARX model:

```
th2 = arx(ze,[2 2 3]);
```

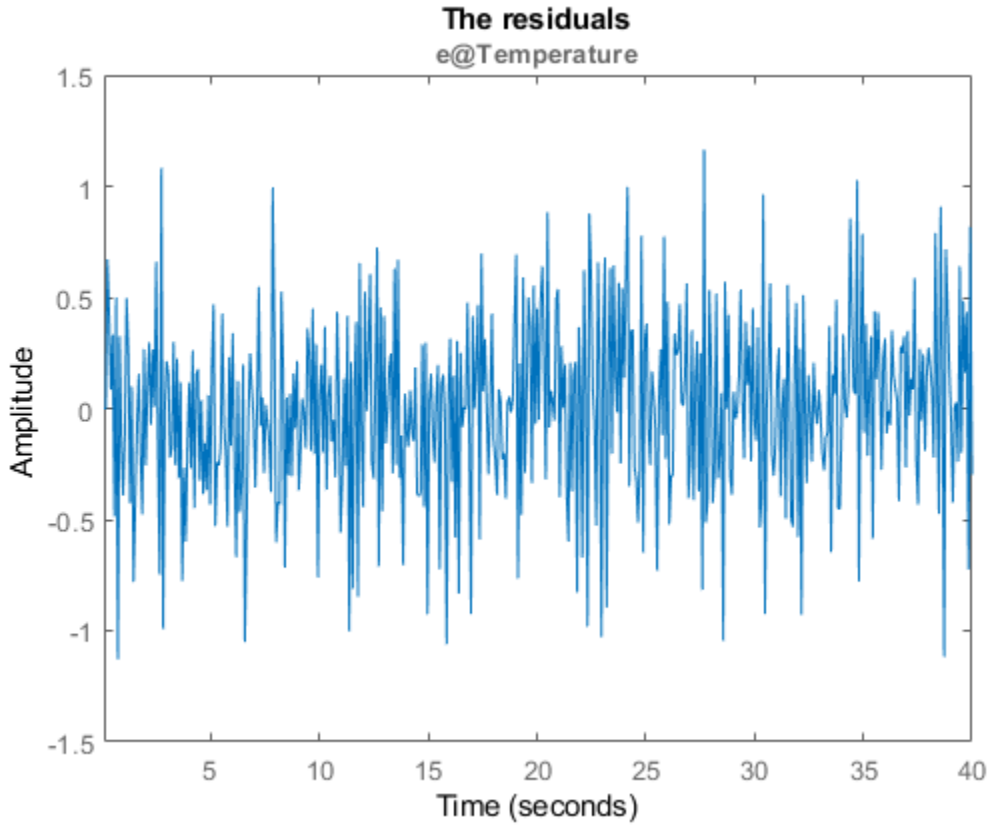
We can test how well this model (`th2`) is capable of reproducing the validation data set. To compare the simulated output from the two models with the actual output (plotting the mid 200 data points) we use the `compare` utility:

```
compare(zr(150:350),th2,th4)
```



The plot indicates that there was no significant loss of accuracy in reducing the order from 4 to 2. We can also check the residuals ("leftovers") of this model, i.e., what is left unexplained by the model.

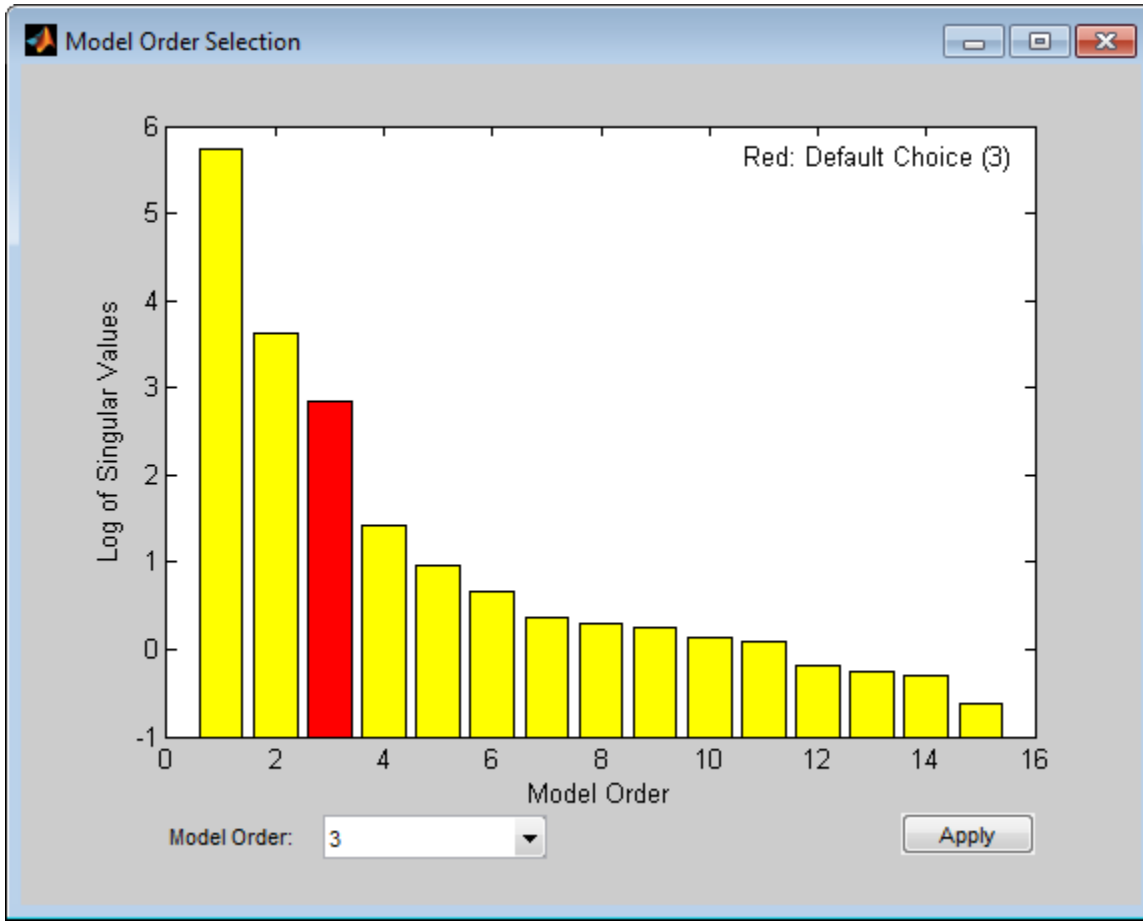
```
e = resid(ze,th2);
plot(e(:,1,[])), title('The residuals')
```



We see that the residuals are quite small compared to the signal level of the output, that they are reasonably well (although not perfectly) uncorrelated with the input and among themselves. We can thus be (provisionally) satisfied with the model `th2`.

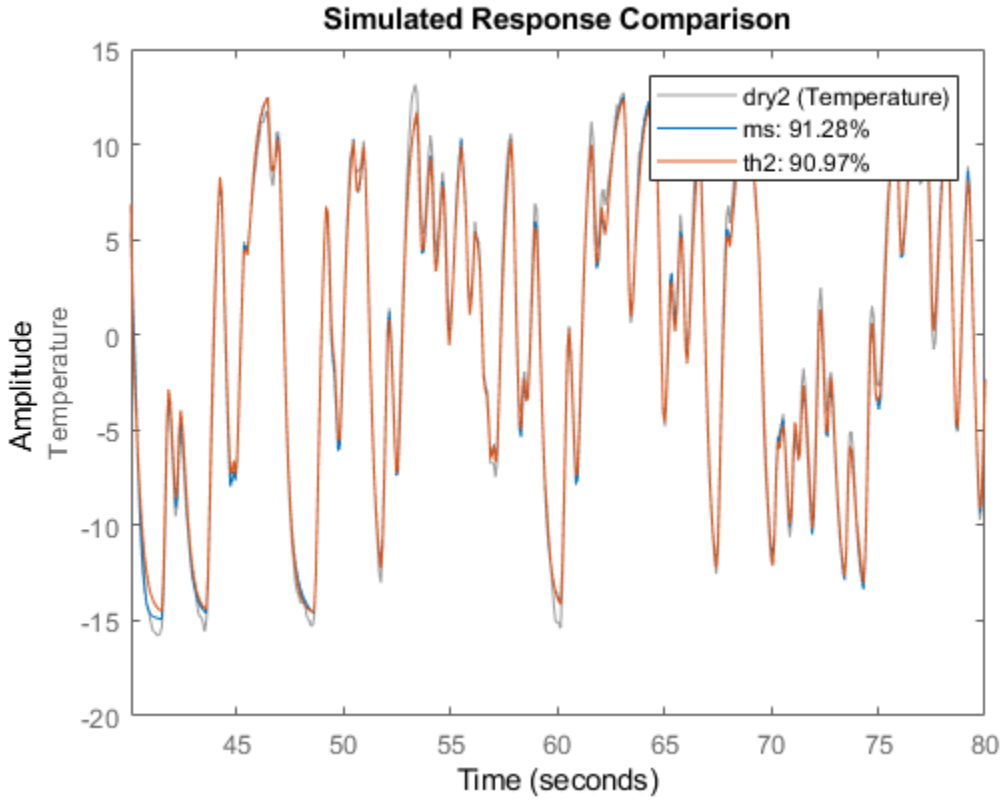
Let us now check if we can determine the model order for a state-space structure. As before, we know the delay is 3 samples. We can try all orders from 1 to 15 with a total lag of 3 samples in `n4sid`. Execute the following command to try various orders and choose one interactively.

```
ms = n4sid(ze,[1:15],'InputDelay',2); %n4sid estimation with variable orders
```



The "InputDelay" was set to 2 because by default `n4sid` estimates a model with no feedthrough (which accounts for one sample lag between input and output). The default order, indicated in the figure above, is 3, that is in good agreement with our earlier findings. Finally, we compare how the state-space model `ms` and the ARX model `th2` compare in reproducing the measured output of the validation data:

```
ms = n4sid(ze,3,'InputDelay',2);
compare(zr,ms,th2)
```



The comparison plot indicates that the two models are practically identical.

Conclusions

This example described some options for choosing a reasonable model order. Determining delay in advance can simplify the task of choosing orders. With ARX and state-space structures, we have some special tools (`arx` and `n4sid` estimators) for automatically evaluating a whole set of model orders, and choosing the best one among them. The information revealed by this exercise (using utilities such as `arxstruc`, `selstruc`, `n4sid` and `delayest`) could be used as a starting point when estimating models of other structures, such as BJ and ARMAX.

Additional Information

For more information on identification of dynamic systems with System Identification Toolbox visit the System Identification Toolbox product information page.

Frequency Domain Identification: Estimating Models Using Frequency Domain Data

This example shows how to estimate models using frequency domain data. The estimation and validation of models using frequency domain data work the same way as they do with time domain data. This provides a great amount of flexibility in estimation and analysis of models using time and frequency domain as well as spectral (FRF) data. You may simultaneously estimate models using data in both domains, compare and combine these models. A model estimated using time domain data may be validated using spectral data or vice-versa.

Frequency domain data cannot be used for estimation or validation of nonlinear models.

Introduction

Frequency domain experimental data are common in many applications. It could be that the data was collected as frequency response data (frequency functions: FRF) from the process using a frequency analyzer. It could also be that it is more practical to work with the input's and output's Fourier transforms (FFT of time-domain data), for example to handle periodic or band-limited data. (A band-limited continuous time signal has no frequency components above the Nyquist frequency). In System Identification Toolbox, frequency domain I/O data are represented the same way as time-domain data, i.e., using `iddata` objects. The 'Domain' property of the object must be set to 'Frequency'. Frequency response data are represented as complex vectors or as magnitude/phase vectors as a function of frequency. IDFRD objects in the toolbox are used to encapsulate FRFs, where a user specifies the complex response data and a frequency vector. Such IDDATA or IDFRD objects (and also FRD objects of Control System Toolbox) may be used seamlessly with any estimation routine (such as `procest`, `tfest` etc).

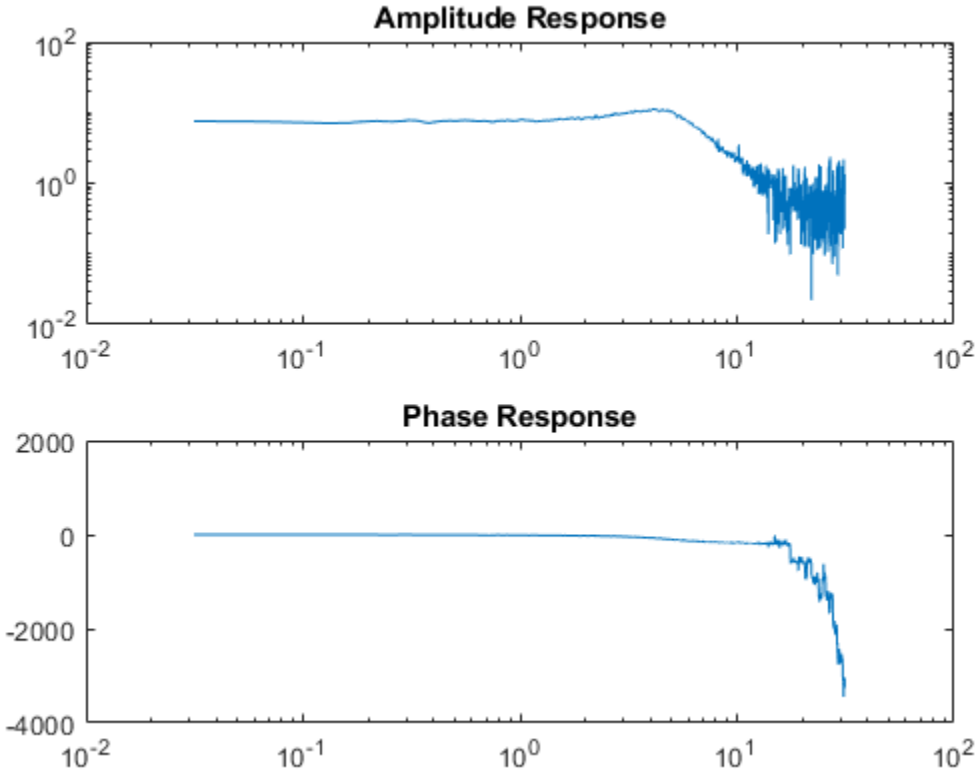
Inspecting Frequency Domain Data

Let us begin by loading some frequency domain data:

```
load demofr
```

This MAT-file contains frequency response data at frequencies `W`, with the amplitude response `AMP` and the phase response `PHA`. Let us first have a look at the data:

```
subplot(211), loglog(W,AMP),title('Amplitude Response')
subplot(212), semilogx(W,PHA),title('Phase Response')
```



This experimental data will now be stored as an IDFRD object. First transform amplitude and phase to a complex valued response:

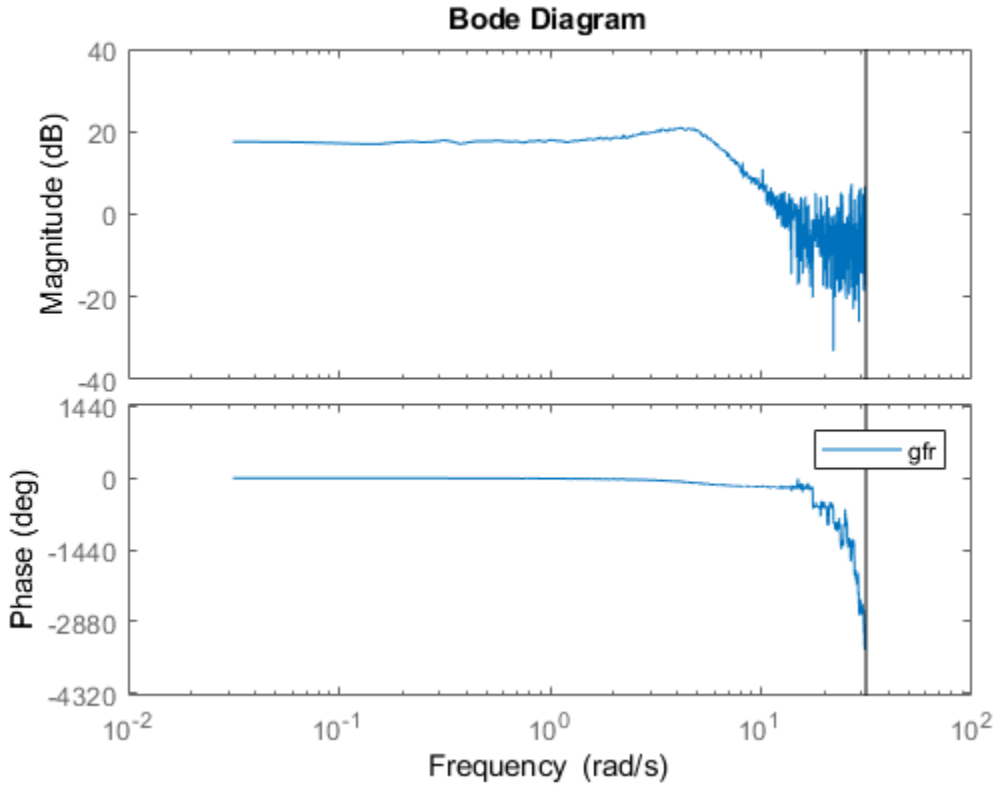
```
zfr = AMP.*exp(1i*PHA*pi/180);
Ts = 0.1;
gfr = idfrd(zfr,W,Ts);
```

`Ts` is the sample time of the underlying data. If the data corresponds to continuous time, for example since the input has been band-limited, use Ts = 0.

Note: If you have the Control System Toolbox™, you could use an FRD object instead of the IDFRD object. IDFRD has options for more information, like disturbance spectra and uncertainty measures which are not available in FRD objects.

The IDFRD object `gfr` now contains the data, and it can be plotted and analyzed in different ways. To view the data, we may use `plot` or `bode`:

```
clf
bode(gfr), legend('gfr')
```



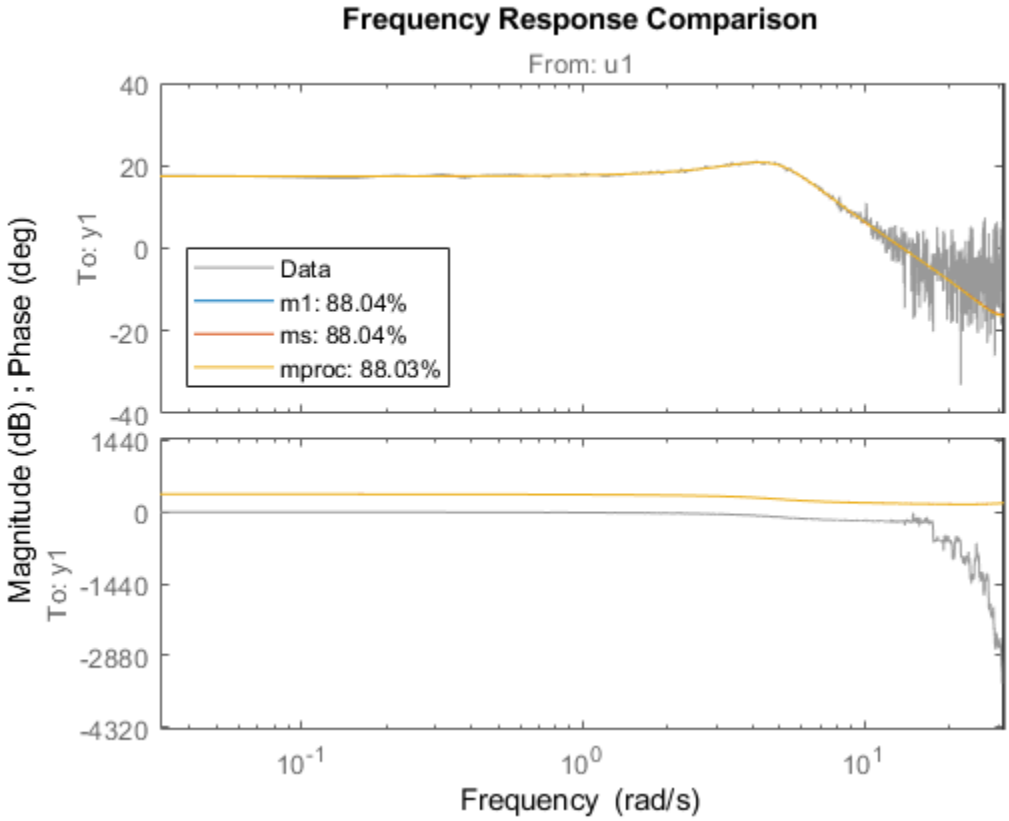
Estimating Models Using Frequency Response (FRF) Data

To estimate models, you can now use `gfr` as a data set with all the commands of the toolbox in a transparent fashion. The only restriction is that noise models cannot be built. This means that for polynomial models only OE (output-error models) apply, and for state-space models, you have to fix `K = 0`.

```
m1 = oe(gfr,[2 2 1]) % Discrete-time Output error (transfer function) model
ms = ssest(gfr) % Continuous-time state-space model with default choice of order
mproc = procest(gfr,'P2UDZ') % 2nd-order, continuous-time model with underdamped poles
compare(gfr,m1,ms,mproc)
L = findobj(gcf,'type','legend');
L.Location = 'southwest'; % move legend to non-overlapping location
```

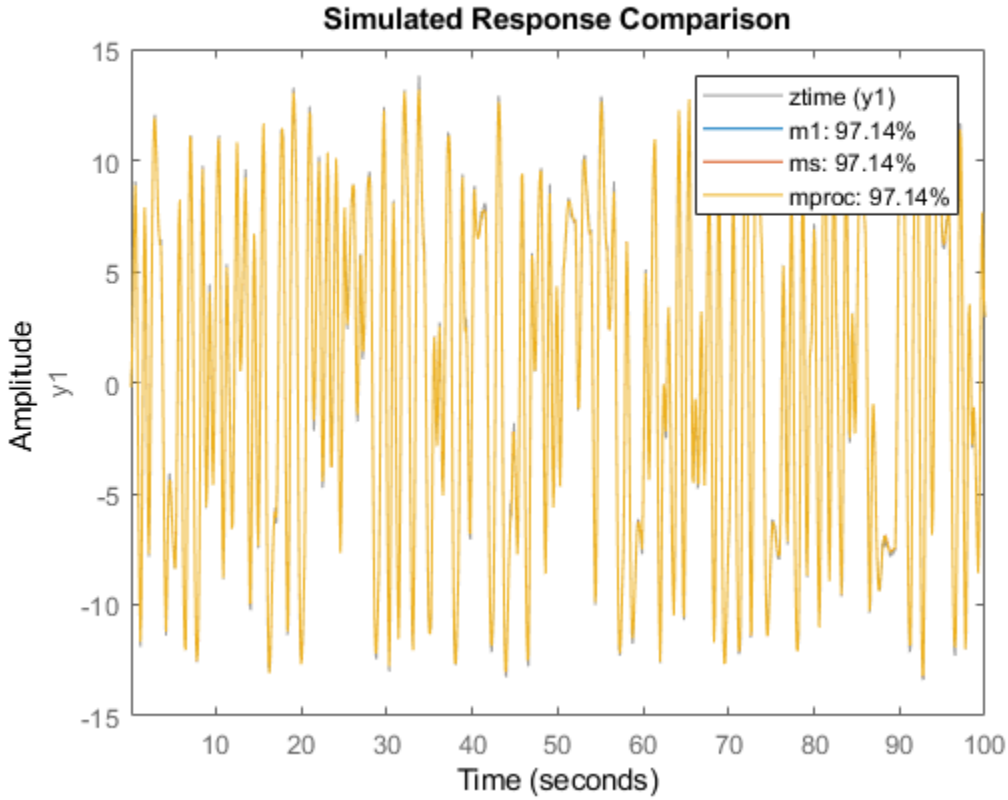
```
m1 =
Discrete-time OE model:  y(t) = [B(z)/F(z)]u(t) + e(t)
  B(z) = 0.9986 z^-1 + 0.4968 z^-2

  F(z) = 1 - 1.499 z^-1 + 0.6998 z^-2

Sample time: 0.1 seconds

Parameterization:
   Polynomial orders:   nb=2   nf=2   nk=1
   Number of free coefficients: 4
   Use "polydata", "getpvec", "getcov" for parameters and their uncertainties.

Status:
Estimated using OE on frequency response data "gfr".
Fit to estimation data: 88.55%
FPE: 0.2492, MSE: 0.2482

ms =
  Continuous-time identified state-space model:
      dx/dt = A x(t) + B u(t) + K e(t)
       y(t) = C x(t) + D u(t) + e(t)

  A =
           x1       x2
   x1  -0.9009    6.635
   x2   -3.307   -2.668

  B =
          u1
   x1   -32.9
   x2  -28.33

  C =
           x1       x2
   y1  -0.5073    0.499
```

```
  D =
       u1
   y1   0

  K =
       y1
   x1   0
   x2   0

Parameterization:
   FREE form (all coefficients in A, B, C free).
   Feedthrough: none
   Disturbance component: none
   Number of free coefficients: 8
   Use "idssdata", "getpvec", "getcov" for parameters and their uncertainties.

Status:
Estimated using SSEST on frequency response data "gfr".
Fit to estimation data: 88.55%
FPE: 0.2492, MSE: 0.2482

mproc =
Process model with transfer function:
                     1+Tz*s
  G(s) = Kp * ---------------------- * exp(-Td*s)
              1+2*Zeta*Tw*s+(Tw*s)^2

         Kp = 7.4619
         Tw = 0.20245
       Zeta = 0.36242
         Td = 0
         Tz = 0.013617

Parameterization:
    'P2DUZ'
   Number of free coefficients: 5
   Use "getpvec", "getcov" for parameters and their uncertainties.

Status:
Estimated using PROCEST on frequency response data "gfr".
Fit to estimation data: 88.55%
FPE: 0.2495, MSE: 0.2482
```



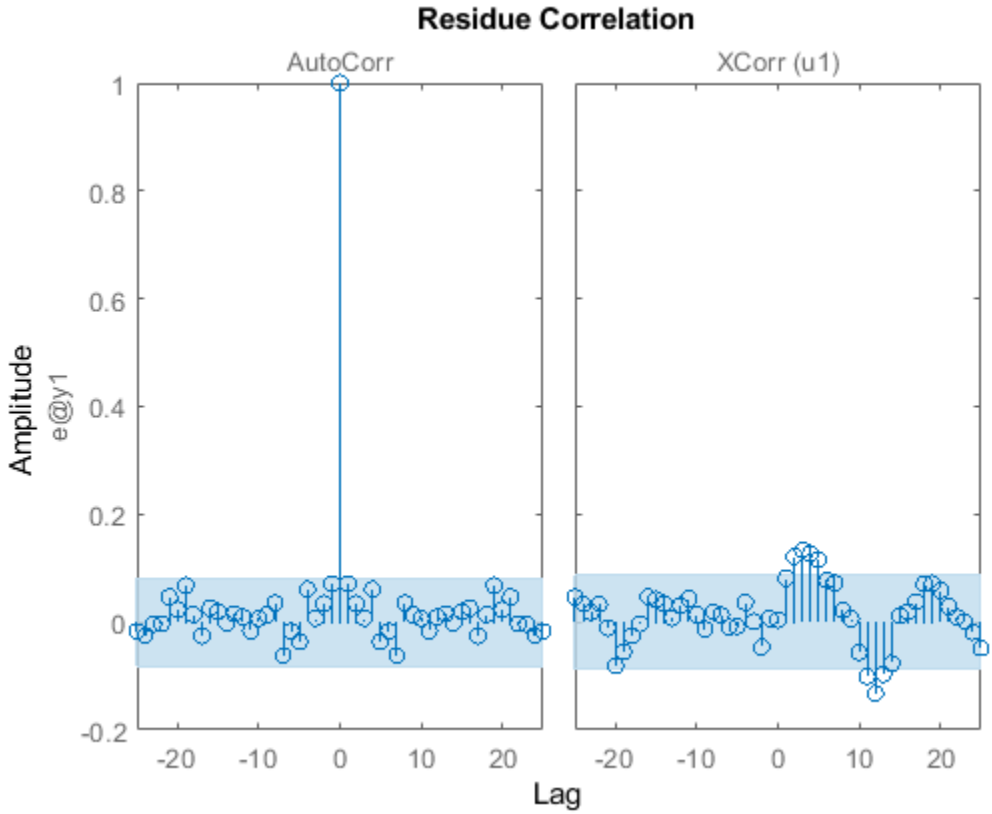
As shown above a variety of linear model types may be estimated in both continuous and discrete time domains, using spectral data. These models may be validated using, time-domain data. The time-domain I/O data set `ztime`, for example, is collected from the same system, and can be used for validation of `m1`, `ms` and `mproc`:

```
compare(ztime,m1,ms,mproc) %validation in a different domain
```



We may also look at the residuals to affirm the quality of the model using the validation data `ztime`. As observed, the residuals are almost white:

```
resid(ztime,mproc) % Residuals plot
```

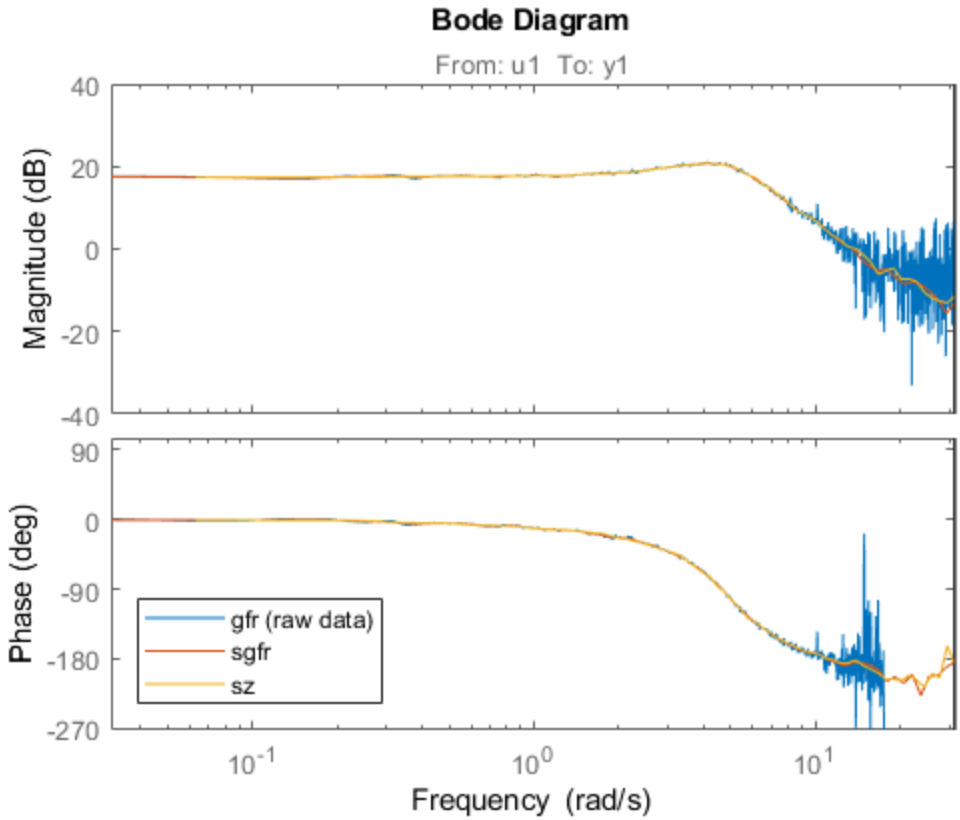


Condensing Data Using SPAFDR

An important reason to work with frequency response data is that it is easy to condense the information with little loss. The command SPAFDR allows you to compute smoothed response data over limited frequencies, for example with logarithmic spacing. Here is an example where the `gfr` data is condensed to 100 logarithmically spaced frequency values. With a similar technique, also the original time domain data can be condensed:

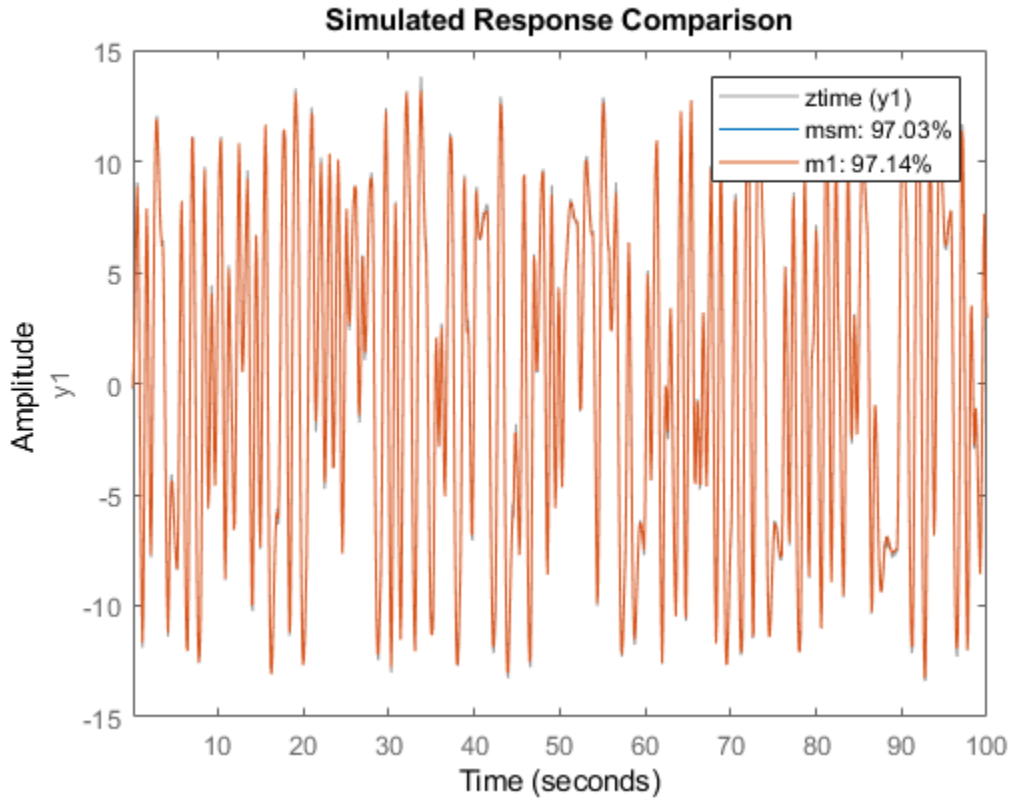
```
sgfr = spafdr(gfr) % spectral estimation with frequency-dependent resolution
sz = spafdr(ztime); % spectral estimation using time-domain data
clf
bode(gfr,sgfr,sz)
axis([pi/100 10*pi, -272 105])
legend('gfr (raw data)','sgfr','sz','location','southwest')
```

```
sgfr =
IDFRD model.
Contains Frequency Response Data for 1 output(s) and 1 input(s), and the spectra for di
Response data and disturbance spectra are available at 100 frequency points, ranging fr

Sample time: 0.1 seconds
Output channels: 'y1'
Input channels: 'u1'
Status:
Estimated using SPAFDR on frequency response data "gfr".
```



The Bode plots show that the information in the smoothed data has been taken well care of. Now, these data records with 100 points can very well be used for model estimation. For example:

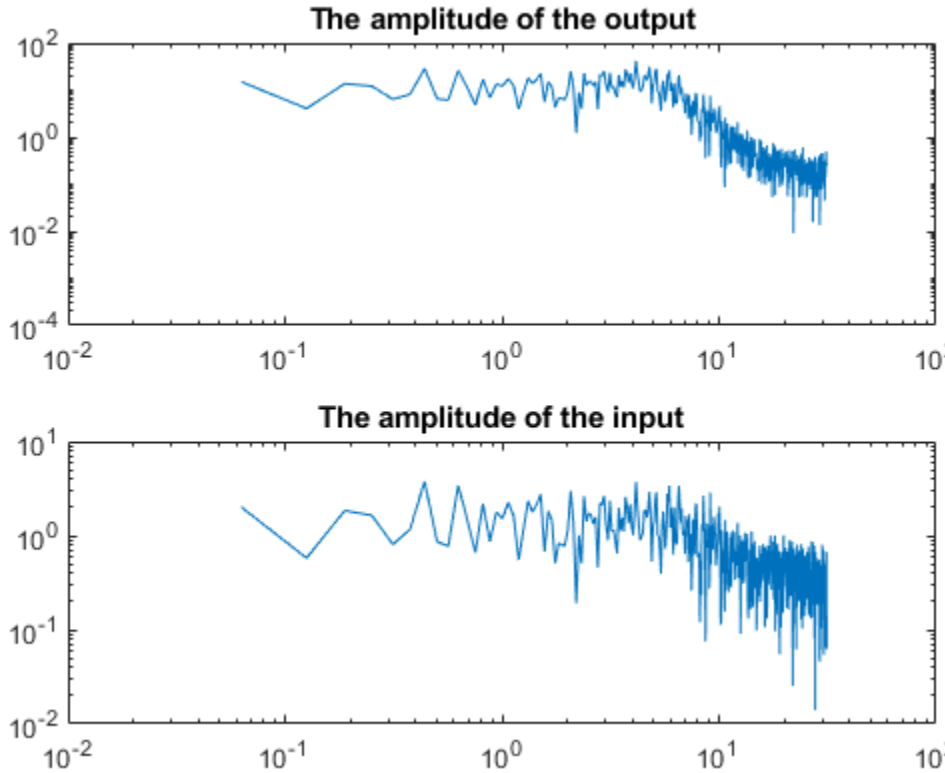
```
msm = oe(sgfr,[2 2 1]);
compare(ztime,msm,m1) % msm has the same accuracy as M1 (based on 1000 points)
```



Estimation Using Frequency-Domain I/O Data

It may be that the measurements are available as Fourier transforms of inputs and output. Such frequency domain data from the system are given as the signals Y and U. In loglog plots they look like

```
Wfd = (0:500)'*10*pi/500;
subplot(211),loglog(Wfd,abs(Y)),title('The amplitude of the output')
subplot(212),loglog(Wfd,abs(U)),title('The amplitude of the input')
```



The frequency response data is essentially the ratio between `Y` and `U`. To collect the frequency domain data as an IDDATA object, do as follows:

```
ZFD = iddata(Y, U, 'Ts', 0.1, 'Frequency', Wfd)
```

```
ZFD =

Frequency domain data set with responses at 501 frequencies,
ranging from 0 to 31.416 rad/seconds
```

```
Sample time: 0.1 seconds

Outputs      Unit (if specified)
   y1

Inputs       Unit (if specified)
   u1
```

Now, again the frequency domain data set `ZFD` can be used as data in all estimation routines, just as time domain data and frequency response data:

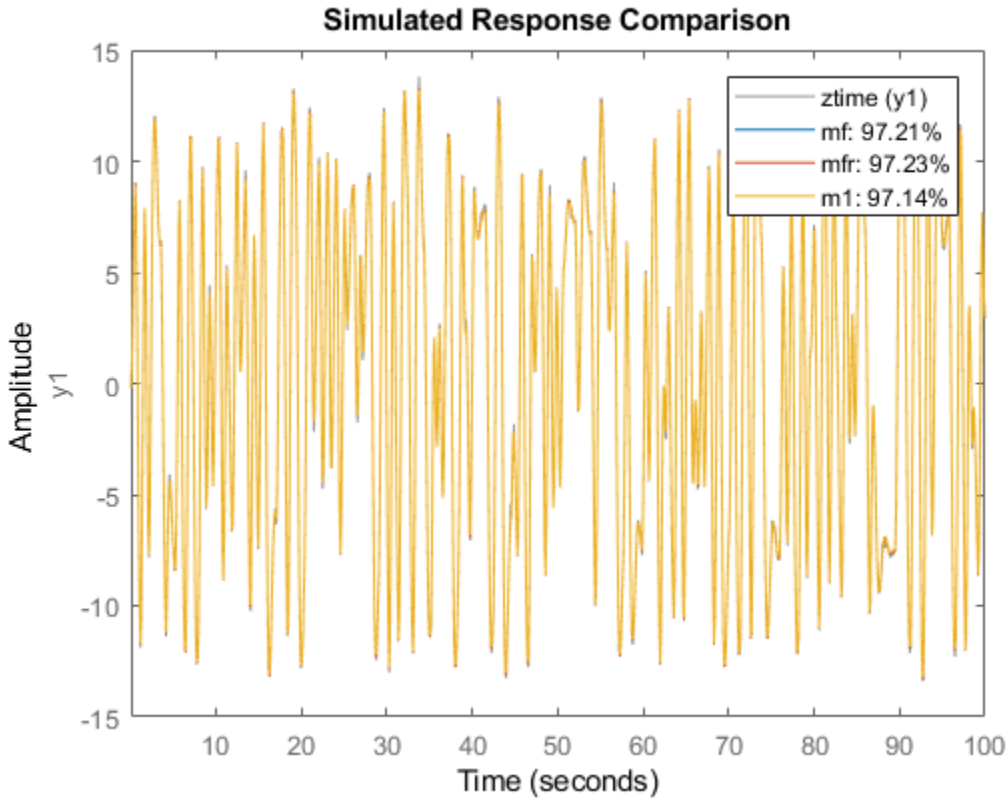
```
mf = ssest(ZFD)   % SSEST picks best order in 1:10 range when called this way
mfr = ssregest(ZFD) % an alternative regularized reduction based state-space estimator
clf
compare(ztime,mf,mfr,m1)
```

```
mf =
  Continuous-time identified state-space model:
      dx/dt = A x(t) + B u(t) + K e(t)
       y(t) = C x(t) + D u(t) + e(t)

  A =
          x1      x2
   x1  -1.501   6.791
   x2  -3.115  -2.059

  B =
          u1
   x1  -28.11
   x2  -33.39

  C =
           x1       x2
   y1  -0.5844   0.4129

  D =
       u1
   y1   0

  K =
       y1
   x1   0
   x2   0
```

```
Parameterization:
   FREE form (all coefficients in A, B, C free).
   Feedthrough: none
   Disturbance component: none
   Number of free coefficients: 8
   Use "idssdata", "getpvec", "getcov" for parameters and their uncertainties.

Status:
Estimated using SSEST on frequency domain data "ZFD".
Fit to estimation data: 97.21%
FPE: 0.0423, MSE: 0.04179

mfr =
  Discrete-time identified state-space model:
    x(t+Ts) = A x(t) + B u(t) + K e(t)
       y(t) = C x(t) + D u(t) + e(t)

  A =
              x1         x2         x3         x4         x5         x6
   x1      0.7607     0.3671    -0.3322   -0.08998    0.01548    0.09474
   x2      -0.227    -0.4068    -0.3153      0.233    -0.1859     0.2795
   x3       0.262    -0.3038     0.6404     0.1926   -0.02235    -0.2056
   x4     -0.0205    -0.1497    0.03978    -0.1501     -0.353    -0.3459
   x5     0.03394    0.01352     0.2328    -0.4938    -0.2778   -0.09414
   x6     0.01766     0.3918     0.1908     0.2817    -0.0382      0.141
   x7    -0.03115    -0.3107   -0.02915    -0.2754     0.3491     0.5479
   x8     0.01892    0.06892    0.09164    -0.1246    -0.4316    0.01104
   x9   -0.003668    -0.1277    -0.1057    0.03904     0.1206     -0.423
   x10     0.0174    0.04442   -0.02417     0.1074    0.02164    -0.2264

              x7         x8         x9        x10
   x1     -0.1456   0.007799    0.05833    -0.1396
   x2      -0.336   0.009541     0.2668    -0.1218
   x3       0.233    -0.1177    0.03473   -0.06575
   x4     -0.1299      0.166     0.1195    -0.2518
   x5     -0.6382    -0.2387    0.04806    0.04836
   x6     -0.2343    -0.4258     0.6103    -0.1953
   x7     -0.2887    -0.6828      0.136    -0.4042
   x8      0.5226    -0.2932  -0.007668    0.09615
   x9     -0.1939    -0.3768   -0.09409      0.507
   x10   -0.02294    -0.6167    -0.4447    -0.5943

  B =
```

```
              u1
   x1       1.588
   x2     -0.1338
   x3    -0.09661
   x4    -0.02391
   x5    -0.02324
   x6     0.03246
   x7   0.0002968
   x8    -0.03677
   x9     0.06961
   x10  -0.006027

  C =
            x1       x2       x3       x4       x5       x6       x7
   y1   0.7727  -0.5047    2.644    1.195  -0.5607    1.542    -1.08

            x8       x9      x10
   y1  0.06959  -0.9512   0.2948

  D =
       u1
   y1   0

  K =
                y1
   x1      -0.0306
   x2     -0.01498
   x3      0.08894
   x4     -0.04447
   x5      0.04233
   x6     -0.01548
   x7      0.01024
   x8   -0.0004959
   x9    -0.003125
   x10   -0.001307

Sample time: 0.1 seconds

Parameterization:
   FREE form (all coefficients in A, B, C free).
   Feedthrough: none
   Disturbance component: estimate
   Number of free coefficients: 130
   Use "idssdata", "getpvec", "getcov" for parameters and their uncertainties.
```

```
Status:
Estimated using SSREGEST on frequency domain data "ZFD".
Fit to estimation data: 65.8% (prediction focus)
FPE: 6.813, MSE: 6.289
```



Transformations Between Data Representations (Time - Frequency)

Time and frequency domain input-output data sets can be transformed to either domain by using FFT and IFFT. These commands are adapted to IDDATA objects:

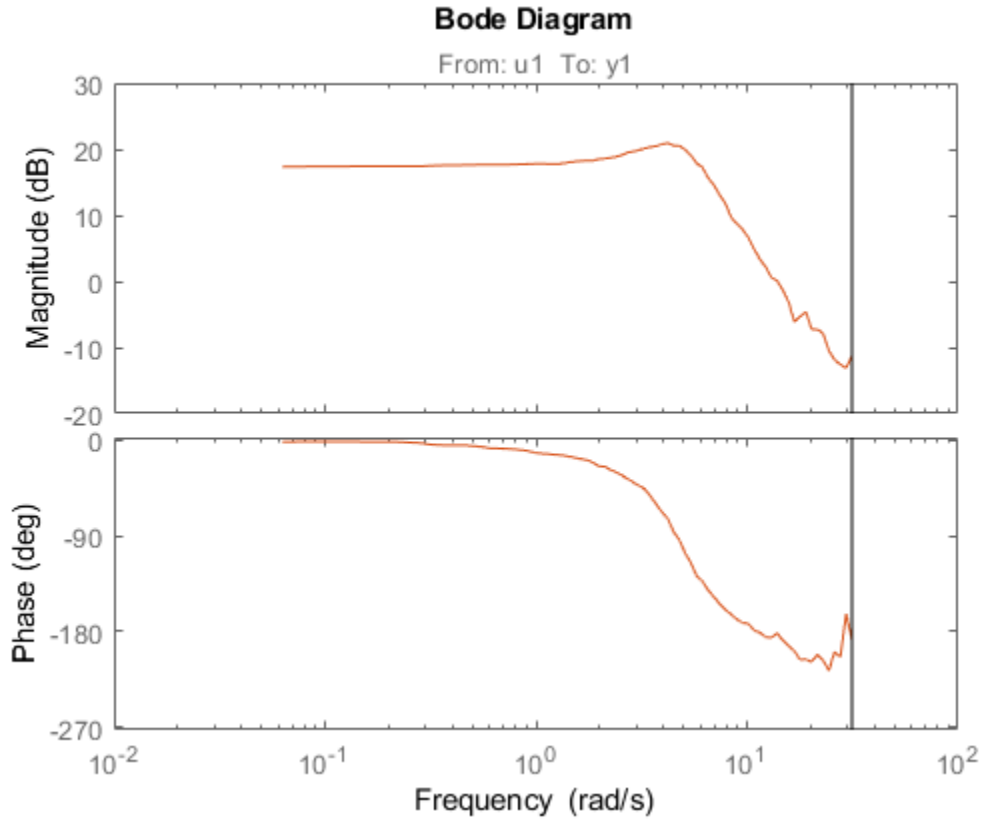
```
dataf = fft(ztime)
datat = ifft(dataf)
```

```
dataf =

Frequency domain data set with responses at 501 frequencies,
ranging from 0 to 31.416 rad/seconds
Sample time: 0.1 seconds

Outputs      Unit (if specified)
   y1

Inputs       Unit (if specified)
   u1


datat =

Time domain data set with 1000 samples.
Sample time: 0.1 seconds

Outputs      Unit (if specified)
   y1

Inputs       Unit (if specified)
   u1
```

Time and frequency domain input-output data can be transformed to frequency response data by SPAFDR, SPA and ETFE:

```
g1 = spafdr(ztime)
g2 = spafdr(ZFD);
clf;
bode(g1,g2)
```

```
g1 =
IDFRD model.
Contains Frequency Response Data for 1 output(s) and 1 input(s), and the spectra for di
Response data and disturbance spectra are available at 100 frequency points, ranging fr

Sample time: 0.1 seconds
Output channels: 'y1'
Input channels: 'u1'
Status:
Estimated using SPAFDR on time domain data "ztime".
```



Frequency response data can also be transformed to more smoothed data (less resolution and less data) by SPAFDR and SPA;

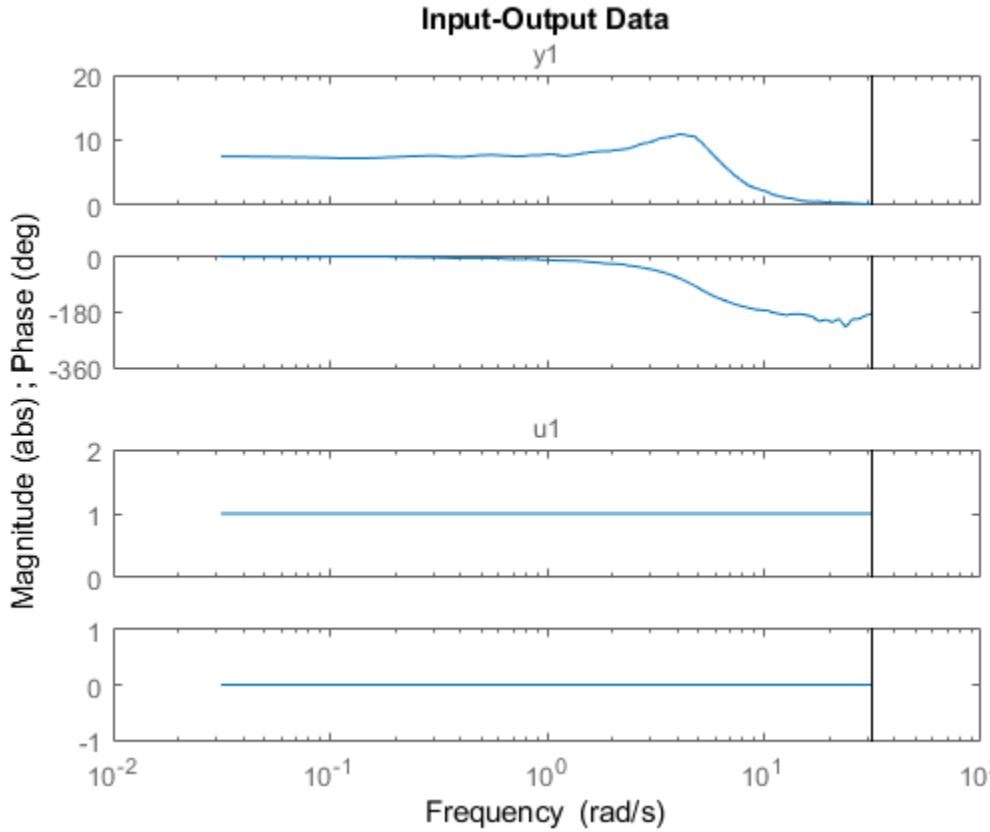
```
g3 = spafdr(gfr);
```

Frequency response data can be transformed to frequency domain input-output signals by the command IDDATA:

```
gfd = iddata(g3)
plot(gfd)
```

```
gfd =
```

```
Frequency domain data set with responses at 100 frequencies,
ranging from 0.031416 to 31.416 rad/seconds
Sample time: 0.1 seconds

Outputs      Unit (if specified)
   y1

Inputs       Unit (if specified)
   u1
```



Using Continuous-time Frequency-domain Data to Estimate Continuous-time Models

Time domain data can naturally only be stored and dealt with as discrete-time, sampled data. Frequency domain data have the advantage that continuous time data can be represented correctly. Suppose that the underlying continuous time signals have no frequency information above the Nyquist frequency, e.g. because they are sampled fast, or the input has no frequency component above the Nyquist frequency and that the data has been collected from a steady-state experiment. Then the Discrete Fourier transforms (DFT) of the data also are the Fourier transforms of the continuous time signals, at the chosen frequencies. They can therefore be used to directly fit continuous time models.

This will be illustrated by the following example.

Consider the continuous time system:

$$G(s) = \frac{1}{s^2 + s + 1}$$

```
m0 = idpoly(1,1,1,1,[1 1 1],'ts',0)
```

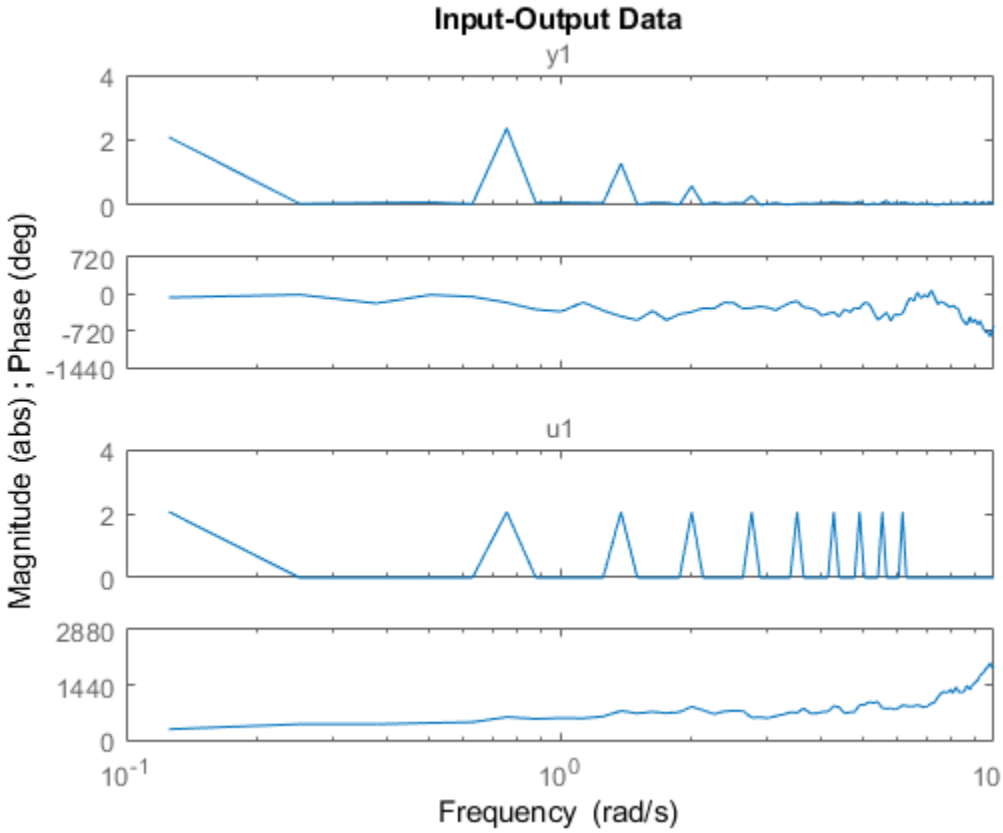
```
m0 =
Continuous-time OE model:  y(t) = [B(s)/F(s)]u(t) + e(t)
  B(s) = 1

  F(s) = s^2 + s + 1

Parameterization:
   Polynomial orders:   nb=1   nf=2   nk=0
   Number of free coefficients: 3
   Use "polydata", "getpvec", "getcov" for parameters and their uncertainties.

Status:
Created by direct construction or transformation. Not estimated.
```

Load data that comes from steady-state simulation of this system using periodic inputs. The collected data was converted into frequency domain and saved in CTFDData.mat file.

```
load CTFDData.mat dataf % load continuous-time frequency-domain data.
```

Look at the data:

```
plot(dataf)
set(gca,'XLim',[0.1 10])
```



Using `dataf` for estimation will by default give continuous time models: State-space:

```
m4 = ssest(dataf,2); %Second order continuous-time model
```

For a polynomial model with `nb = 2` numerator coefficient and `nf = 2` estimated denominator coefficients use:

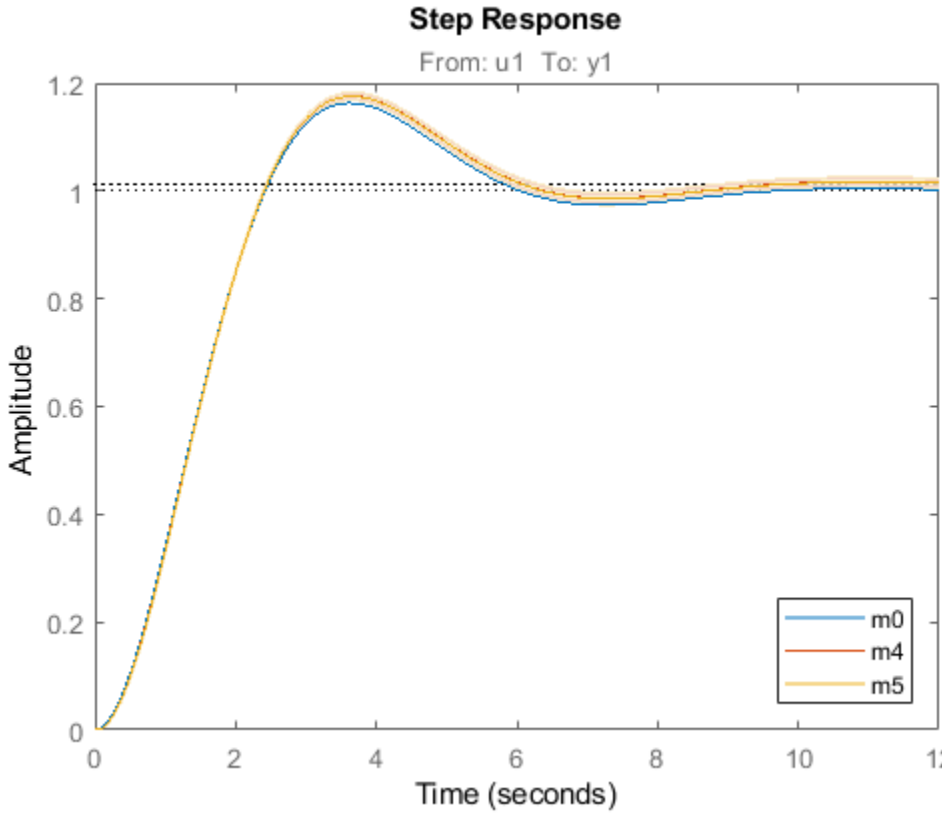
```
nb = 2;
nf = 2;
m5 = oe(dataf,[nb nf])
```

```
m5 =
Continuous-time OE model:  y(t) = [B(s)/F(s)]u(t) + e(t)
```

```
  B(s) = -0.01826 s + 1.008

  F(s) = s^2 + 1.001 s + 0.9967

Parameterization:
   Polynomial orders:   nb=2   nf=2   nk=0
   Number of free coefficients: 4
   Use "polydata", "getpvec", "getcov" for parameters and their uncertainties.

Status:
Estimated using OE on frequency domain data "dataf".
Fit to estimation data: 70.08%
FPE: 0.004842, MSE: 0.004728
```

Compare step responses with uncertainty of the true system `m0` and the models `m4` and `m5`. The confidence intervals are shown with patches in the plot.

```
clf
h = stepplot(m0,m4,m5);
showConfidence(h,1)
legend('show','location','southeast')
```



Although it was not necessary in this case, it is generally advised to focus the fit to a limited frequency band (low pass filter the data) when estimating using continuous time data. The system has a bandwidth of about 3 rad/s, and was excited by sinusoids up to 6.2 rad/s. A reasonable frequency range to focus the fit to is then [0 7] rad/s:

```
m6 = ssest(dataf,2,ssestOptions('WeightingFilter',[0 7])) % state space model
```

```
m6 =
  Continuous-time identified state-space model:
      dx/dt = A x(t) + B u(t) + K e(t)
       y(t) = C x(t) + D u(t) + e(t)

  A =
```

```
           x1       x2
   x1   -0.274   -1.809
   x2   0.4403  -0.7283

  B =
           u1
   x1   0.5786
   x2  -0.3114

  C =
          x1      x2
   y1  0.7686   1.485

  D =
       u1
   y1   0

  K =
       y1
   x1   0
   x2   0

Parameterization:
   FREE form (all coefficients in A, B, C free).
   Feedthrough: none
   Disturbance component: none
   Number of free coefficients: 8
   Use "idssdata", "getpvec", "getcov" for parameters and their uncertainties.

Status:
Estimated using SSEST on frequency domain data "dataf".
Fit to estimation data: 87.08% (data prefiltered)
FPE: 0.004804, MSE: 0.003682
```

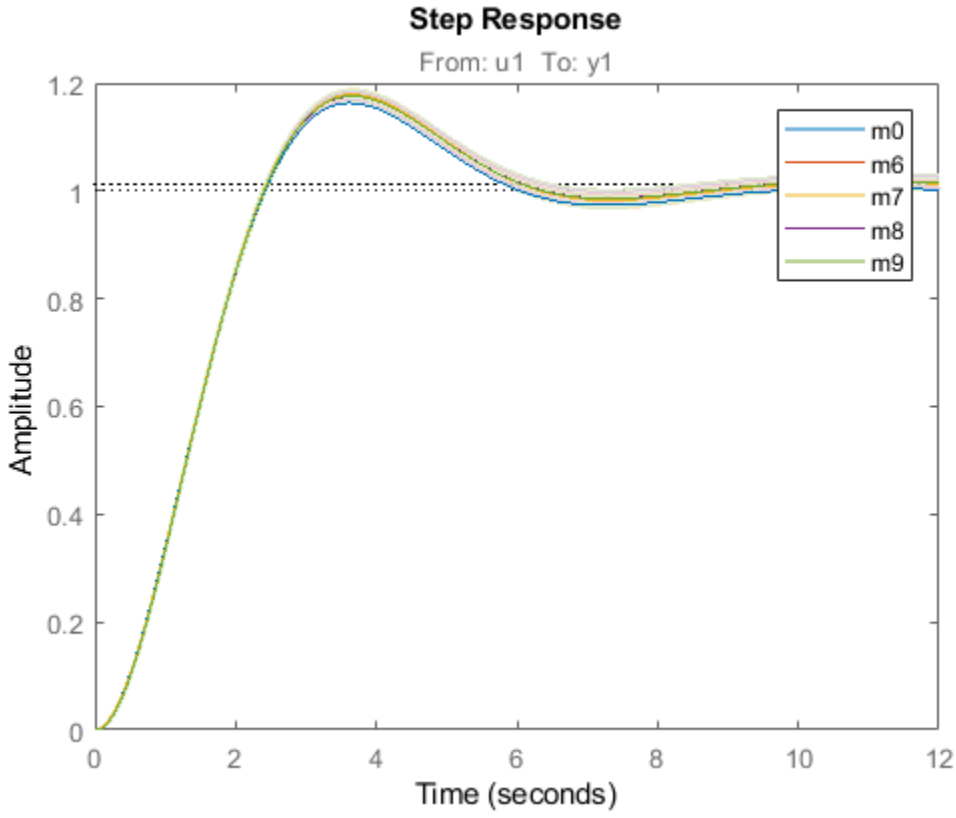
```
m7 = oe(dataf,[1 2],oeOptions('WeightingFilter',[0 7])) % polynomial model of Output Er
```

```
m7 =
Continuous-time OE model:  y(t) = [B(s)/F(s)]u(t) + e(t)
  B(s) = 0.9866

  F(s) = s^2 + 0.9791 s + 0.9761

Parameterization:
   Polynomial orders:   nb=1   nf=2   nk=0
```

```
   Number of free coefficients: 3
   Use "polydata", "getpvec", "getcov" for parameters and their uncertainties.

Status:
Estimated using OE on frequency domain data "dataf".
Fit to estimation data: 87.05% (data prefiltered)
FPE: 0.004829, MSE: 0.0037

opt = procestOptions('SearchMethod','lsqnonlin',...
   'WeightingFilter',[0 7]); % Requires Optimization Toolbox(TM)
m8 = procest(dataf,'P2UZ',opt)  % process model with underdamped poles


m8 =
Process model with transfer function:
                      1+Tz*s
  G(s) = Kp * ----------------------
              1+2*Zeta*Tw*s+(Tw*s)^2

         Kp = 1.0124
         Tw = 1.0019
       Zeta = 0.5021
         Tz = -0.017474

Parameterization:
    'P2UZ'
   Number of free coefficients: 4
   Use "getpvec", "getcov" for parameters and their uncertainties.

Status:
Estimated using PROCEST on frequency domain data "dataf".
Fit to estimation data: 87.08% (data prefiltered)
FPE: 0.004804, MSE: 0.003682

opt = tfestOptions('SearchMethod','lsqnonlin',...
   'WeightingFilter',[0 7]); % Requires Optimization Toolbox
m9 = tfest(dataf,2,opt) % transfer function with 2 poles


m9 =

  From input "u1" to output "y1":
  -0.01662 s + 1.007
  ------------------
   s^2 + s + 0.995
```

```
Continuous-time identified transfer function.

Parameterization:
   Number of poles: 2   Number of zeros: 1
   Number of free coefficients: 4
   Use "tfdata", "getpvec", "getcov" for parameters and their uncertainties.

Status:
Estimated using TFEST on frequency domain data "dataf".
Fit to estimation data: 87.09% (data prefiltered)
FPE: 0.004842, MSE: 0.003679
```

```
h = stepplot(m0,m6,m7,m8,m9);
showConfidence(h,1)
legend('show')
```



Conclusions

We saw how time, frequency and spectral data can seamlessly be used to estimate a variety of linear models in both continuous and discrete time domains. The models may be validated and compared in domains different from the ones they were estimated in. The data formats (time, frequency and spectrum) are interconvertible, using methods such as `fft`, `ifft`, `spafdr` and `spa`. Furthermore, direct, continuous-time estimation is achievable by using `tfest`, `ssest` and `procest` estimation routines. The seamless use

of data in any domain for estimation and analysis is an important feature of System Identification Toolbox.

See Also

`oe` | `procest` | `ssest` | `tfest`

More About

- “Estimating Models Using Frequency-Domain Data”

Building Structured and User-Defined Models Using System Identification Toolbox™

This example shows how to estimate parameters in user-defined model structures. Such structures are specified by IDGREY (linear state-space) or IDNLGREY (nonlinear state-space) models. We shall investigate how to assign structure, fix parameters and create dependencies among them.

Experiment Data

We shall investigate data produced by a (simulated) dc-motor. We first load the data:

```
load dcmdata
who
```

```
Your variables are:

text  u     y
```

The matrix `y` contains the two outputs: `y1` is the angular position of the motor shaft and `y2` is the angular velocity. There are 400 data samples and the sample time is 0.1 seconds. The input is contained in the vector `u`. It is the input voltage to the motor.

```
z = iddata(y,u,0.1); % The IDDATA object
z.InputName = 'Voltage';
z.OutputName = {'Angle';'AngVel'};
plot(z(:,1,:))
```

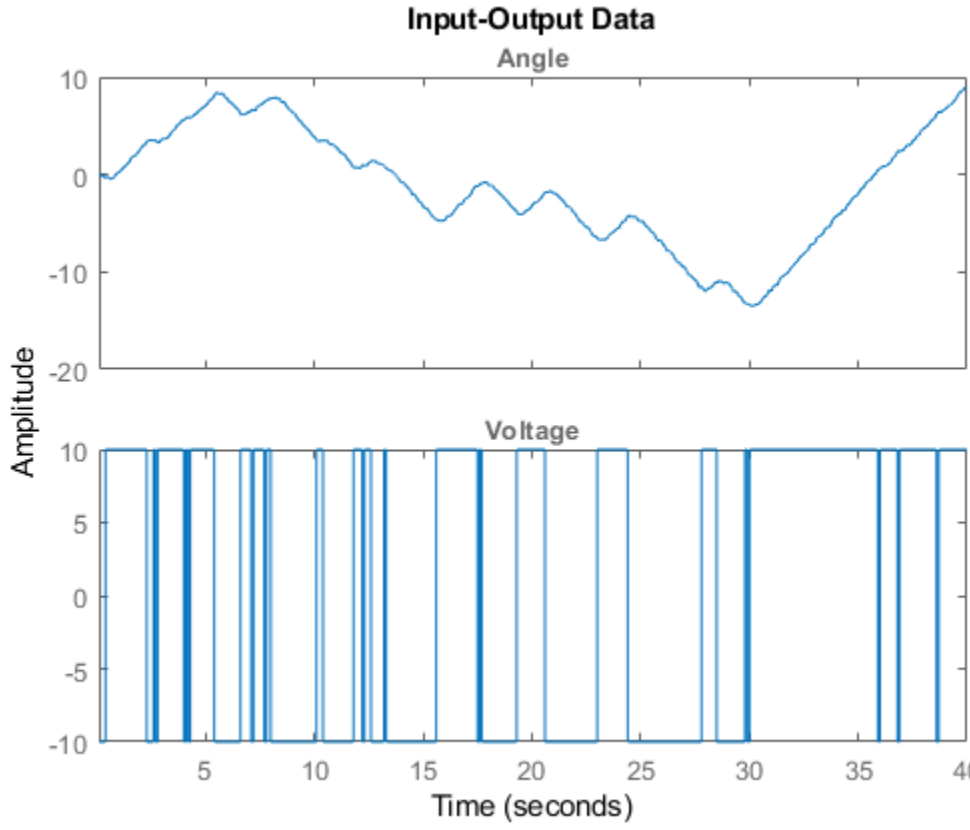


Figure: Measurement Data: Voltage to Angle

```
plot(z(:,2,:))
```

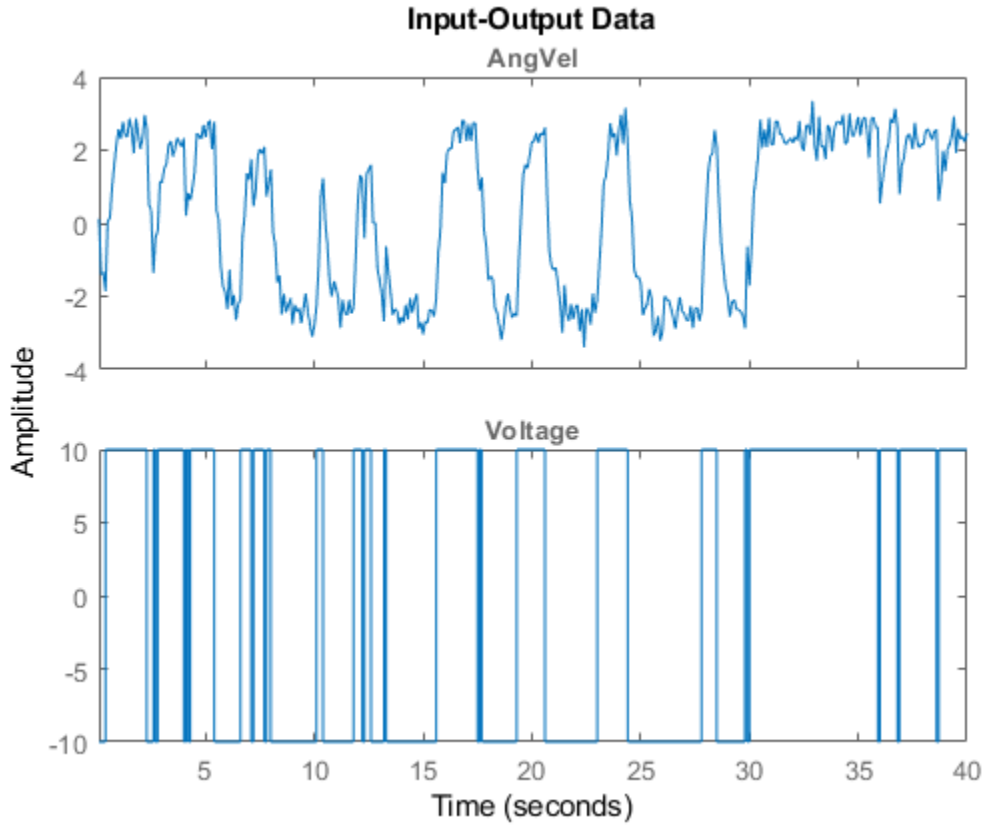


Figure: Measurement Data: Voltage to Angle

Model Structure Selection

```
d/dt x = A x + B u + K e
   y   = C x + D u + e
```

We shall build a model of the dc-motor. The dynamics of the motor is well known. If we choose x1 as the angular position and x2 as the angular velocity it is easy to set up a state-space model of the following character neglecting disturbances: (see Example 4.1 in Ljung(1999):

```
          | 0     1  |       | 0   |
d/dt x =  |          | x  +  |     | u
          | 0   -th1 |       | th2 |

         | 1  0 |
    y =  |      | x
         | 0  1 |
```

The parameter `th1` is here the inverse time-constant of the motor and `th2` is such that `th2/th1` is the static gain from input to the angular velocity. (See Ljung(1987) for how `th1` and `th2` relate to the physical parameters of the motor). We shall estimate these two parameters from the observed data. The model structure (parameterized state space) described above can be represented in MATLAB® using IDSS and IDGREY models. These models let you perform estimation of parameters using experimental data.

Specification of a Nominal (Initial) Model

If we guess that th1=1 and th2 = 0.28 we obtain the nominal or initial model

```
A = [0 1; 0 -1]; % initial guess for A(2,2) is -1
B = [0; 0.28]; % initial guess for B(2) is 0.28
C = eye(2);
D = zeros(2,1);
```

and we package this into an IDSS model object:

```
ms = idss(A,B,C,D);
```

The model is characterized by its matrices, their values, which elements are free (to be estimated) and upper and lower limits of those:

```
ms.Structure.a
```

```
ans =

       Name: 'A'
      Value: [2x2 double]
    Minimum: [2x2 double]
    Maximum: [2x2 double]
       Free: [2x2 logical]
      Scale: [2x2 double]
       Info: [2x2 struct]
```

```
1x1 param.Continuous


ms.Structure.a.Value
ms.Structure.a.Free


ans =

     0     1
     0    -1


ans =

  2x2 logical array

   1   1
   1   1
```

Specification of Free (Independent) Parameters Using IDSS Models

So we should now mark that it is only A(2,2) and B(2,1) that are free parameters to be estimated.

```
ms.Structure.a.Free = [0 0; 0 1];
ms.Structure.b.Free = [0; 1];
ms.Structure.c.Free = 0; % scalar expansion used
ms.Structure.d.Free = 0;
ms.Ts = 0;  % This defines the model to be continuous
```

The Initial Model

```
ms % Initial model


ms =
  Continuous-time identified state-space model:
      dx/dt = A x(t) + B u(t) + K e(t)
       y(t) = C x(t) + D u(t) + e(t)

  A =
       x1  x2
   x1   0   1
```

```
   x2   0  -1

  B =
         u1
   x1     0
   x2  0.28

  C =
       x1  x2
   y1   1   0
   y2   0   1

  D =
       u1
   y1   0
   y2   0

  K =
       y1  y2
   x1   0   0
   x2   0   0

Parameterization:
   STRUCTURED form (some fixed coefficients in  A, B, C).
   Feedthrough: none
   Disturbance component: none
   Number of free coefficients: 2
   Use "idssdata", "getpvec", "getcov" for parameters and their uncertainties.

Status:
Created by direct construction or transformation. Not estimated.
```

Estimation of Free Parameters of the IDSS Model

The prediction error (maximum likelihood) estimate of the parameters is now computed by:

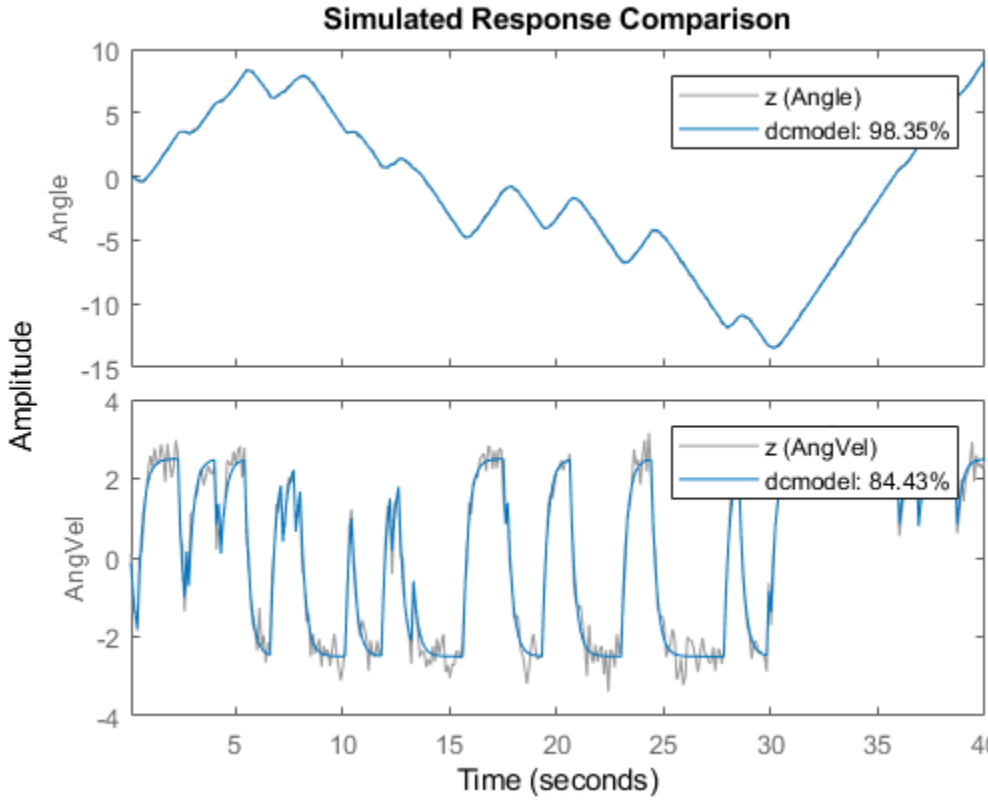
```
dcmodel = ssest(z,ms,ssestOptions('Display','on'));
dcmodel
```

```
dcmodel =
  Continuous-time identified state-space model:
      dx/dt = A x(t) + B u(t) + K e(t)
       y(t) = C x(t) + D u(t) + e(t)
```

```
  A =
            x1      x2
   x1        0       1
   x2        0  -4.013

  B =
       Voltage
   x1        0
   x2    1.002

  C =
            x1  x2
   Angle     1   0
   AngVel    0   1

  D =
            Voltage
   Angle          0
   AngVel         0

  K =
         Angle  AngVel
   x1        0       0
   x2        0       0

Parameterization:
   STRUCTURED form (some fixed coefficients in  A, B, C).
   Feedthrough: none
   Disturbance component: none
   Number of free coefficients: 2
   Use "idssdata", "getpvec", "getcov" for parameters and their uncertainties.

Status:
Estimated using SSEST on time domain data "z".
Fit to estimation data: [98.35;84.42]%
FPE: 0.001071, MSE: 0.1192
```

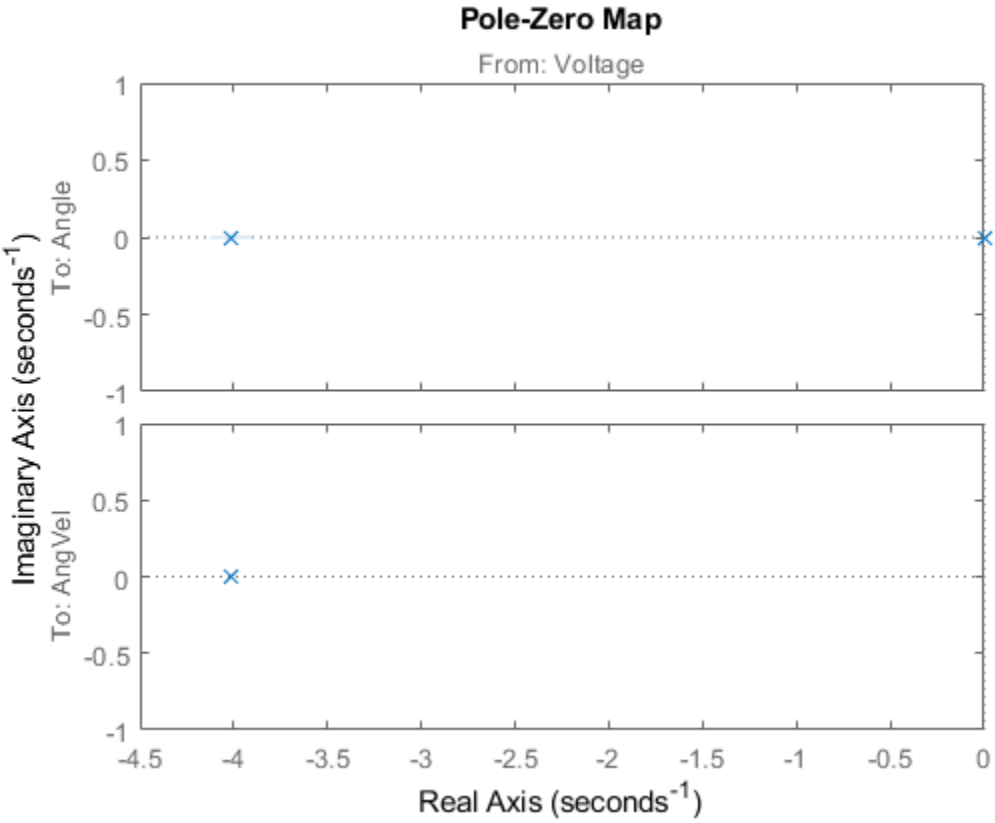
The estimated values of the parameters are quite close to those used when the data were simulated (-4 and 1). To evaluate the model's quality we can simulate the model with the actual input by and compare it with the actual output.

```
compare(z,dcmodel);
```



We can now, for example plot zeros and poles and their uncertainty regions. We will draw the regions corresponding to 3 standard deviations, since the model is quite accurate. Note that the pole at the origin is absolutely certain, since it is part of the model structure; the integrator from angular velocity to position.

```
clf
showConfidence(iopzplot(dcmodel),3)
```



Now, we may make various modifications. The 1,2-element of the A-matrix (fixed to 1) tells us that `x2` is the derivative of `x1`. Suppose that the sensors are not calibrated, so that there may be an unknown proportionality constant. To include the estimation of such a constant we just "let loose" `A(1,2)` and re-estimate:

```
dcmodel2 = dcmodel;
dcmodel2.Structure.a.Free(1,2) = 1;
dcmodel2 = pem(z,dcmodel2,ssestOptions('Display','on'));
```

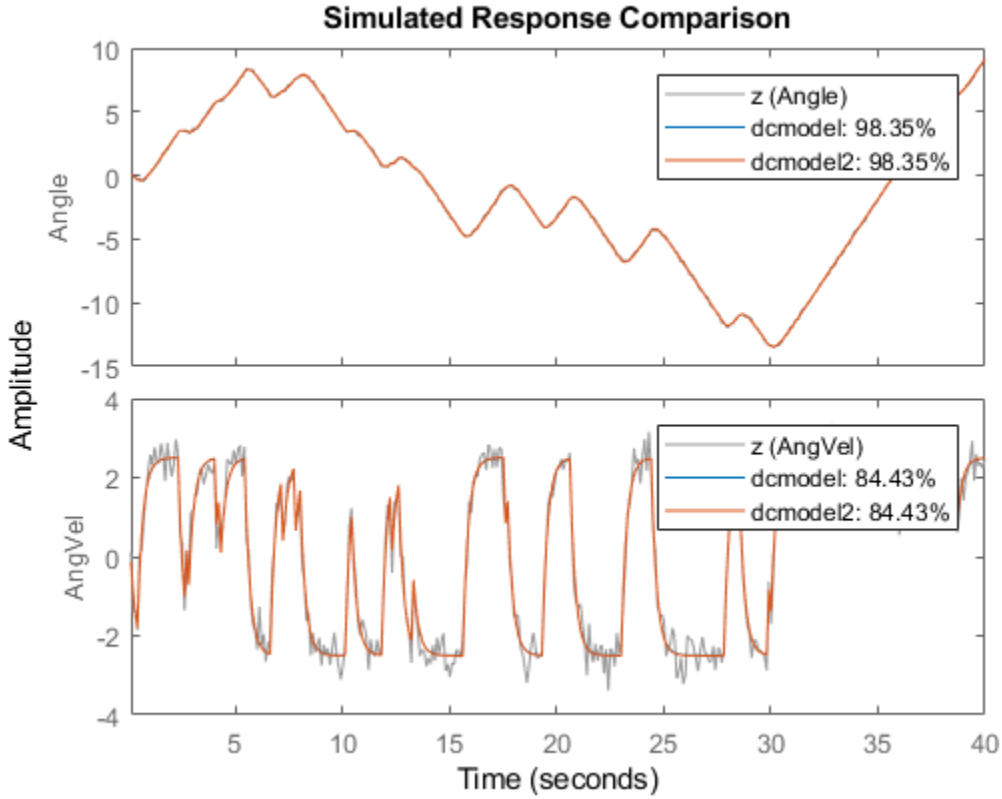
The resulting model is

```
dcmodel2
```

```
dcmodel2 =
  Continuous-time identified state-space model:
      dx/dt = A x(t) + B u(t) + K e(t)
       y(t) = C x(t) + D u(t) + e(t)

  A =
           x1      x2
   x1       0  0.9975
   x2       0  -4.011

  B =
        Voltage
   x1         0
   x2     1.004

  C =
           x1  x2
   Angle    1   0
   AngVel   0   1

  D =
           Voltage
   Angle         0
   AngVel        0

  K =
         Angle  AngVel
   x1        0       0
   x2        0       0

Parameterization:
   STRUCTURED form (some fixed coefficients in  A, B, C).
   Feedthrough: none
   Disturbance component: none
   Number of free coefficients: 3
   Use "idssdata", "getpvec", "getcov" for parameters and their uncertainties.

Status:
Estimated using PEM on time domain data "z".
Fit to estimation data: [98.35;84.42]%
FPE: 0.001077, MSE: 0.1192
```

We find that the estimated `A(1,2)` is close to 1. To compare the two model we use the `compare` command:

```
compare(z,dcmodel,dcmodel2)
```



Specification of Models with Coupled Parameters Using IDGREY Objects

Suppose that we accurately know the static gain of the dc-motor (from input voltage to angular velocity, e.g. from a previous step-response experiment. If the static gain is `G`, and the time constant of the motor is t, then the state-space model becomes

```
           |0     1|     |  0  |
d/dt x =   |       |x + |     | u
           |0  -1/t|     | G/t |
```

```
          |1   0|
   y   =  |     | x
          |0   1|
```

With `G` known, there is a dependence between the entries in the different matrices. In order to describe that, the earlier used way with "Free" parameters will not be sufficient. We thus have to write a MATLAB file which produces the A, B, C, and D, and optionally also the K and X0 matrices as outputs, for each given parameter vector as input. It also takes auxiliary arguments as inputs, so that the user can change certain things in the model structure, without having to edit the file. In this case we let the known static gain `G` be entered as such an argument. The file that has been written has the name 'motorDynamics.m'.

```
type motorDynamics
```

```
function [A,B,C,D,K,X0] = motorDynamics(par,ts,aux)
%MOTORDYNAMICS ODE file representing the dynamics of a motor.
%
%   [A,B,C,D,K,X0] = motorDynamics(Tau,Ts,G)
%   returns the State Space matrices of the DC-motor with
%   time-constant Tau (Tau = par) and known static gain G. The sample
%   time is Ts.
%
%   This file returns continuous-time representation if input argument Ts
%   is zero. If Ts>0, a discrete-time representation is returned. To make
%   the IDGREY model that uses this file aware of this flexibility, set the
%   value of Structure.FcnType property to 'cd'. This flexibility is useful
%   for conversion between continuous and discrete domains required for
%   estimation and simulation.
%
%   See also IDGREY, IDDEMO7.

%   L. Ljung
%   Copyright 1986-2015 The MathWorks, Inc.

t = par(1);
G = aux(1);

A = [0 1;0 -1/t];
B = [0;G/t];
C = eye(2);
D = [0;0];
K = zeros(2);
```

```
X0 = [0;0];
if ts>0 % Sample the model with sample time Ts
   s = expm([[A B]*ts; zeros(1,3)]);
   A = s(1:2,1:2);
   B = s(1:2,3);
end
```

We now create an IDGREY model object corresponding to this model structure: The assumed time constant will be

```
par_guess = 1;
```

We also give the value 0.25 to the auxiliary variable `G` (gain) and sample time.

```
aux = 0.25;
dcmm = idgrey('motorDynamics',par_guess,'cd',aux,0);
```

The time constant is now estimated by

```
dcmm = greyest(z,dcmm,greyestOptions('Display','on'));
```

We have thus now estimated the time constant of the motor directly. Its value is in good agreement with the previous estimate.

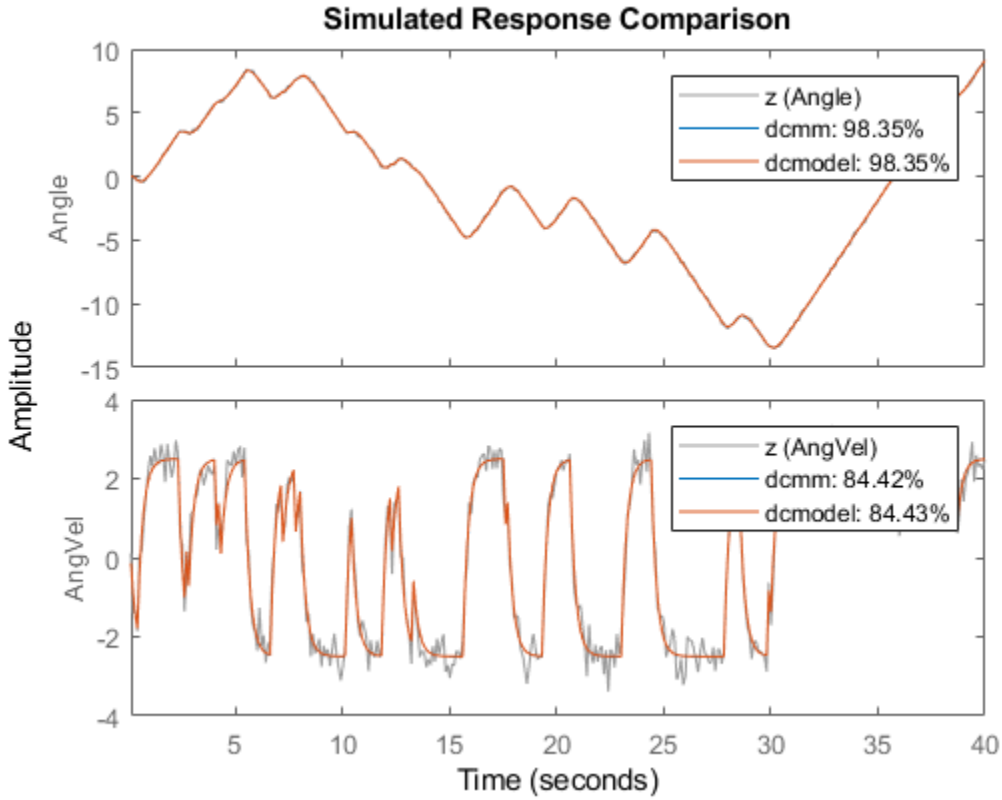
```
dcmm
```

```
dcmm =
  Continuous-time linear grey box model defined by "motorDynamics" function:
      dx/dt = A x(t) + B u(t) + K e(t)
       y(t) = C x(t) + D u(t) + e(t)

  A =
           x1      x2
   x1       0       1
   x2       0  -4.006

  B =
      Voltage
   x1       0
   x2   1.001

  C =
          x1  x2
   Angle   1   0
```

```
   AngVel   0   1

  D =
            Voltage
   Angle          0
   AngVel         0

  K =
         Angle  AngVel
   x1        0       0
   x2        0       0

  Model parameters:
   Par1 = 0.2496

Parameterization:
   ODE Function: motorDynamics
   (parameterizes both continuous- and discrete-time equations)
   Disturbance component: parameterized by the ODE function
   Initial state: parameterized by the ODE function
   Number of free coefficients: 1
   Use "getpvec", "getcov" for parameters and their uncertainties.

Status:
Estimated using GREYEST on time domain data "z".
Fit to estimation data: [98.35;84.42]%
FPE: 0.00107, MSE: 0.1193
```

With this model we can now proceed to test various aspects as before. The syntax of all the commands is identical to the previous case. For example, we can compare the idgrey model with the other state-space model:

```
compare(z,dcmm,dcmodel)
```



They are clearly very close.

Estimating Multivariate ARX Models

The state-space part of the toolbox also handles multivariate (several outputs) ARX models. By a multivariate ARX-model we mean the following:

```
A(q) y(t) = B(q) u(t) + e(t)
```

Here A(q) is a ny | ny matrix whose entries are polynomials in the delay operator 1/q. The k-l element is denoted by:

$a_{kl}(q)$

where:

$$a_{kl}(q) = 1 + a_1 q^{-1} + ... + a_{nakl} q^{-nakl} q$$

It is thus a polynomial in `1/q` of degree `nakl`.

Similarly B(q) is a ny | nu matrix, whose kj-element is:
$$b_{kj}(q) = b_0 q^{-nkk} + b_1 q^{-nkkj-1} + ... + b_{nbkj} q^{-nkkj-nbkj}$$

There is thus a delay of `nkkj` from input number `j` to output number `k`. The most common way to create those would be to use the ARX-command. The orders are specified as: `nn = [na nb nk]` with `na` being a `ny-by-ny` matrix whose `kj`-entry is `nakj`; `nb` and `nk` are defined similarly.

Let's test some ARX-models on the dc-data. First we could simply build a general second order model:

```
dcarx1 = arx(z,'na',[2,2;2,2],'nb',[2;2],'nk',[1;1])
```

```
dcarx1 =
Discrete-time ARX model:
  Model for output "Angle": A(z)y_1(t) = - A_i(z)y_i(t) + B(z)u(t) + e_1(t)
    A(z) = 1 - 0.5545 z^-1 - 0.4454 z^-2

    A_2(z) = -0.03548 z^-1 - 0.06405 z^-2

    B(z) = 0.004243 z^-1 + 0.006589 z^-2

  Model for output "AngVel": A(z)y_2(t) = - A_i(z)y_i(t) + B(z)u(t) + e_2(t)
    A(z) = 1 - 0.2005 z^-1 - 0.2924 z^-2

    A_1(z) = 0.01849 z^-1 - 0.01937 z^-2

    B(z) = 0.08642 z^-1 + 0.03877 z^-2

Sample time: 0.1 seconds

Parameterization:
   Polynomial orders:   na=[2 2;2 2]   nb=[2;2]   nk=[1;1]
   Number of free coefficients: 12
   Use "polydata", "getpvec", "getcov" for parameters and their uncertainties.

Status:
Estimated using ARX on time domain data "z".
```

```
Fit to estimation data: [97.87;83.44]% (prediction focus)
FPE: 0.002157, MSE: 0.1398
```

The result, `dcarx1`, is stored as an IDPOLY model, and all previous commands apply. We could for example explicitly list the ARX-polynomials by:

```
dcarx1.a


ans =

  2x2 cell array

    {1x3 double}    {1x3 double}
    {1x3 double}    {1x3 double}
```

as cell arrays where e.g. the {1,2} element of dcarx1.a is the polynomial A(1,2) described earlier, relating y2 to y1.

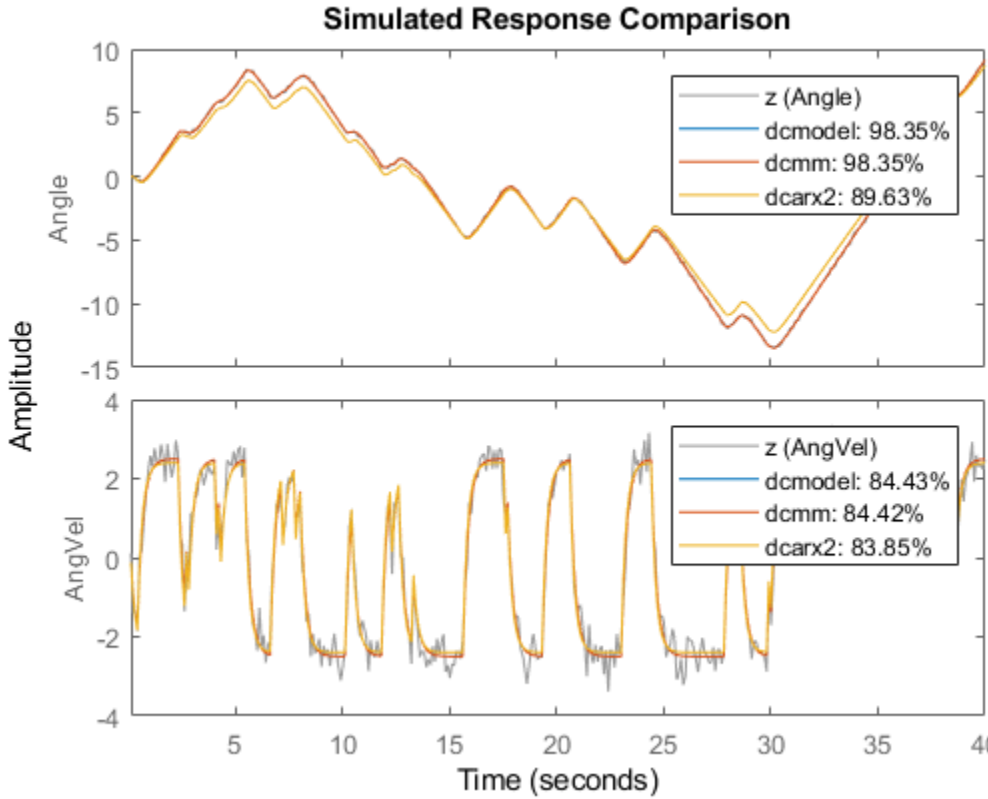
We could also test a structure, where we know that `y1` is obtained by filtering `y2` through a first order filter. (The angle is the integral of the angular velocity). We could then also postulate a first order dynamics from input to output number 2:

```
na = [1 1; 0 1];
nb = [0 ; 1];
nk = [1 ; 1];
dcarx2 = arx(z,[na nb nk])


dcarx2 =
Discrete-time ARX model:
  Model for output "Angle": A(z)y_1(t) = - A_i(z)y_i(t) + B(z)u(t) + e_1(t)
    A(z) = 1 - 0.9992 z^-1

    A_2(z) = -0.09595 z^-1

    B(z) = 0

  Model for output "AngVel": A(z)y_2(t) = B(z)u(t) + e_2(t)
    A(z) = 1 - 0.6254 z^-1

    B(z) = 0.08973 z^-1

Sample time: 0.1 seconds
```

```
Parameterization:
   Polynomial orders:   na=[1 1;0 1]   nb=[0;1]   nk=[1;1]
   Number of free coefficients: 4
   Use "polydata", "getpvec", "getcov" for parameters and their uncertainties.

Status:
Estimated using ARX on time domain data "z".
Fit to estimation data: [97.52;81.46]% (prediction focus)
FPE: 0.003452, MSE: 0.177
```

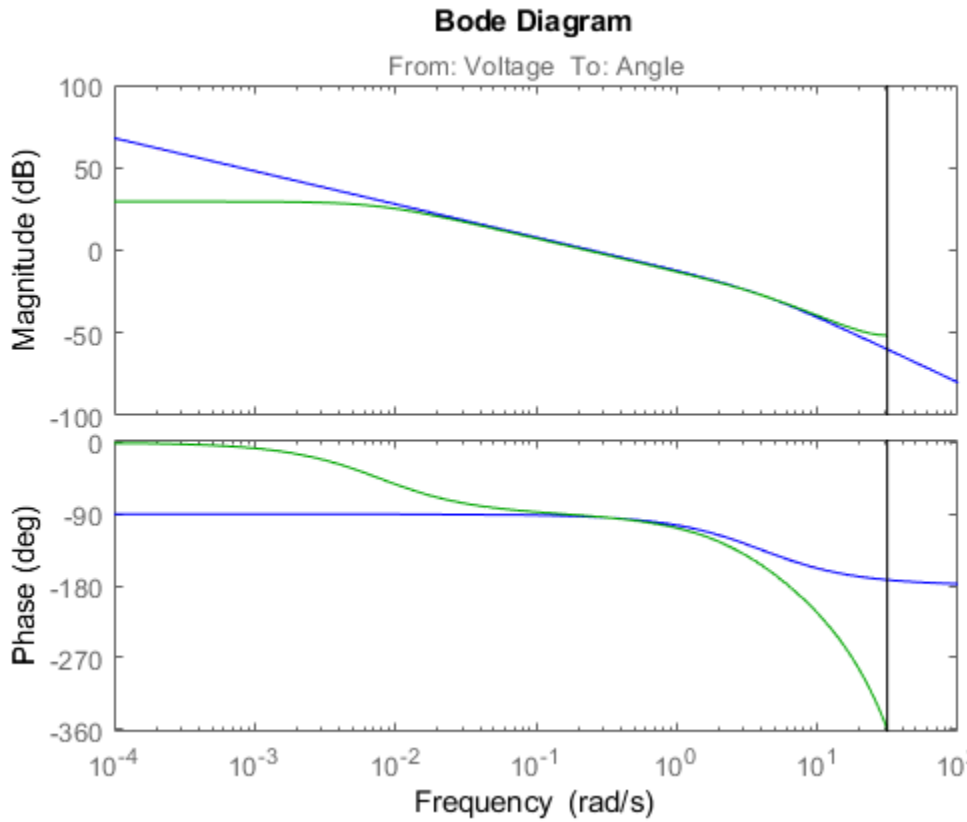
To compare the different models obtained we use

```
compare(z,dcmodel,dcmm,dcarx2)
```



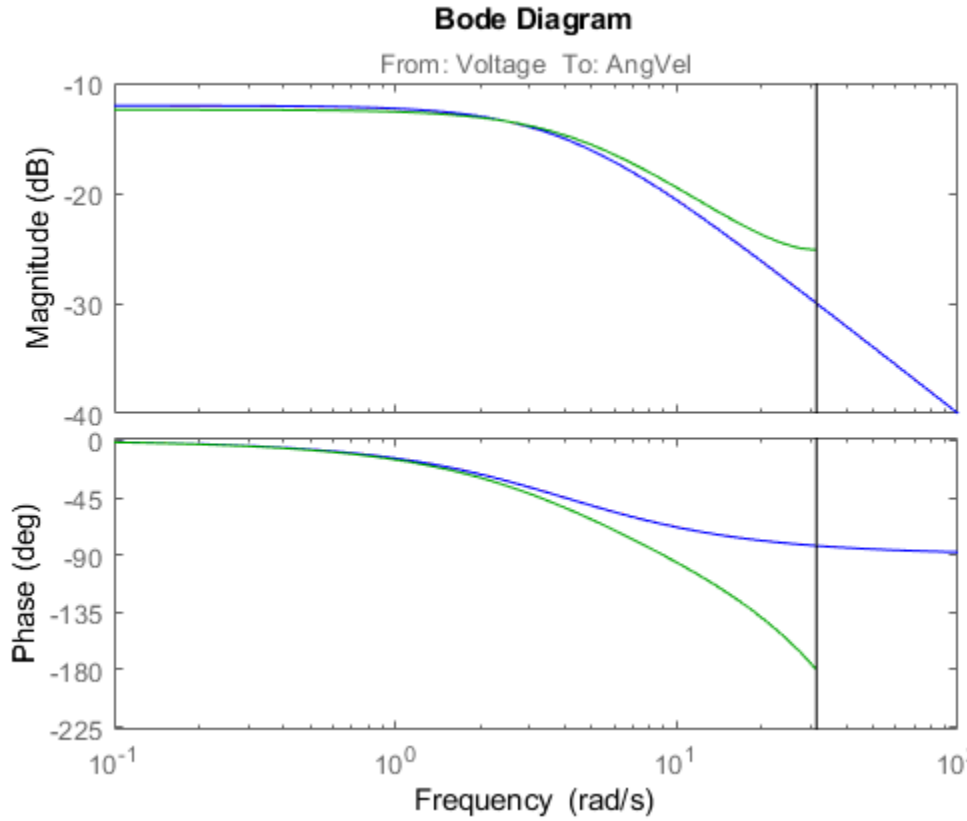
Finally, we could compare the bodeplots obtained from the input to output one for the different models by using `bode`: First output:

```
dcmm2 = idss(dcmm); % convert to IDSS for subreferencing
bode(dcmodel(1,1),'r',dcmm2(1,1),'b',dcarx2(1,1),'g')
```



Second output:

```
bode(dcmodel(2,1),'r',dcmm2(2,1),'b',dcarx2(2,1),'g')
```



The two first models are more or less in exact agreement. The ARX-models are not so good, due to the bias caused by the non-white equation error noise. (We had white measurement noise in the simulations).

Conclusions

Estimation of models with pre-selected structures can be performed using System Identification toolbox. In state-space form, parameters may be fixed to their known values or constrained to lie within a prescribed range. If relationship between parameters or other constraints need to be specified, IDGREY objects may be used. IDGREY models evaluate a user-specified MATLAB file for estimating state-space system parameters. Multi-variate ARX models offer another option for quickly estimating multi-output models with user-specified structure.

Additional Information

For more information on identification of dynamic systems with System Identification Toolbox visit the System Identification Toolbox product information page.

Identifying Process Models

- "What Is a Process Model?" on page 5-2
- "Data Supported by Process Models" on page 5-4
- "Estimate Process Models Using the App" on page 5-5
- "Estimate Process Models at the Command Line" on page 5-11
- "Building and Estimating Process Models Using System Identification Toolbox™" on page 5-18
- "Process Model Structure Specification" on page 5-44
- "Estimating Multiple-Input, Multi-Output Process Models" on page 5-46
- "Disturbance Model Structure for Process Models" on page 5-47
- "Specifying Initial Conditions for Iterative Estimation Algorithms" on page 5-49

What Is a Process Model?

The structure of a *process model* is a simple continuous-time transfer function that describes linear system dynamics in terms of one or more of the following elements:

- Static gain K_p.
- One or more time constants T_{pk}. For complex poles, the time constant is called T_ω — equal to the inverse of the natural frequency—and the damping coefficient is ζ (`zeta`).
- Process zero T_z.
- Possible time delay T_d before the system output responds to the input (*dead time*).
- Possible enforced integration.

Process models are popular for describing system dynamics in many industries and apply to various production environments. The advantages of these models are that they are simple, support transport delay estimation, and the model coefficients have an easy interpretation as poles and zeros.

You can create different model structures by varying the number of poles, adding an integrator, or adding or removing a time delay or a zero. You can specify a first-, second-, or third-order model, and the poles can be real or complex (underdamped modes). For more information, see “Process Model Structure Specification” on page 5-44.

For example, the following model structure is a first-order continuous-time process model, where K is the static gain, T_{p1} is a time constant, and T_d is the input-to-output delay:

$$G(s) = \frac{K_p}{1 + sT_{p1}} e^{-sT_d}$$

Such that, $Y(s) = G(s)U(s) + E(s)$, where $Y(s)$, $U(s)$, and $E(s)$ represent the Laplace transforms of the output, input, and output error, respectively. The output error, $e(t)$, is white Gaussian noise with variance λ. You can account for colored noise at the output by adding a disturbance model, $H(s)$, such that $Y(s) = G(s)U(s) + H(s)E(s)$. For more information, see the `NoiseTF` property of `idproc`.

A multi-input multi-output (MIMO) process model contains a SISO process model corresponding to each input-output pair in the system. For example, for a two-input, two-output process model:

$$Y_1(s) = G_{11}(s)U_1(s) + G_{12}(s)U_2(s) + E_1(s)$$

$$Y_2(s) = G_{21}(s)U_1(s) + G_{22}(s)U_2(s) + E_2(s)$$

Where, $G_{ij}(s)$ is the SISO process model between the i^{th} output and the j^{th} input. $E_1(s)$ and $E_2(s)$ are the Laplace transforms of the two output errors.

See Also

Related Examples

- "Estimate Process Models Using the App" on page 5-5
- "Estimate Process Models at the Command Line" on page 5-11
- "Data Supported by Process Models" on page 5-4
- "Process Model Structure Specification" on page 5-44

Data Supported by Process Models

You can estimate low-order (up to third order), continuous-time transfer functions using regularly sampled time- or frequency-domain `iddata` or `idfrd` data objects. The frequency-domain data may have a zero sample time.

You must import your data into the MATLAB workspace, as described in "Data Preparation".

See Also

Related Examples

- "Estimate Process Models Using the App" on page 5-5
- "Estimate Process Models at the Command Line" on page 5-11

More About

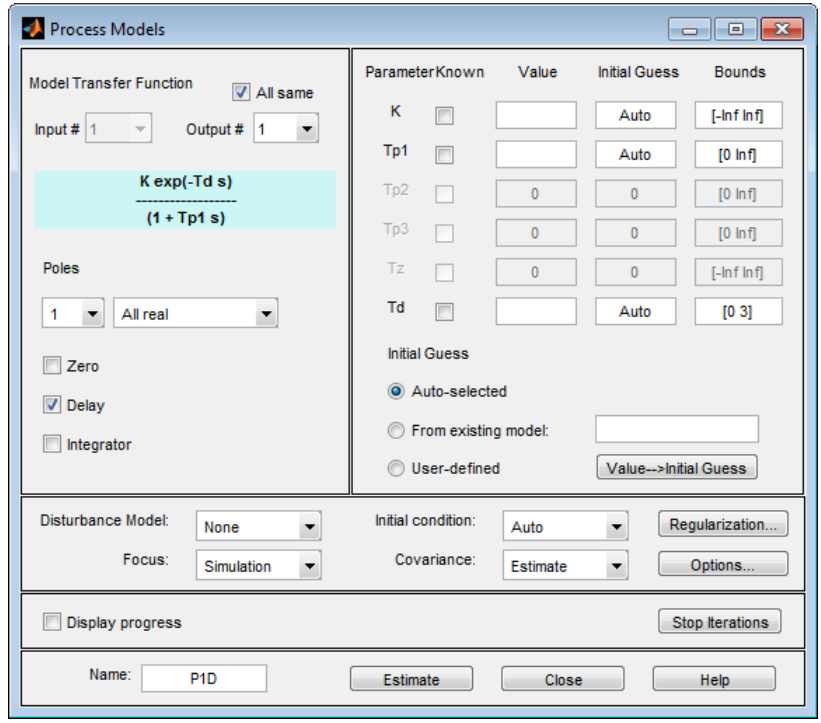
- "What Is a Process Model?" on page 5-2

Estimate Process Models Using the App

Before you can perform this task, you must have

- Imported data into the System Identification app. See “Import Time-Domain Data into the App” on page 2-16. For supported data formats, see “Data Supported by Process Models” on page 5-4.
- Performed any required data preprocessing operations. If you need to model nonzero offsets, such as when model contains integration behavior, do not detrend your data. In other cases, to improve the accuracy of your model, you should detrend your data. See “Ways to Prepare Data for System Identification” on page 2-6.

1 In the System Identification app, select **Estimate** > **Process models** to open the Process Models dialog box.



To learn more about the options in the dialog box, click **Help**.

2 If your model contains multiple inputs or multiple outputs, you can specify whether to estimate the same transfer function for all input-output pairs, or a different transfer function for each. Select the input and output channels in the **Input** and **Output** fields. The fields only appears when you have multiple inputs or outputs. For more information, see "Estimating Multiple-Input, Multi-Output Process Models" on page 5-46.

3 In the **Model Transfer Function** area, specify the model structure using the following options:

- Under **Poles**, select the number of poles, and then select `All real` or `Underdamped`.

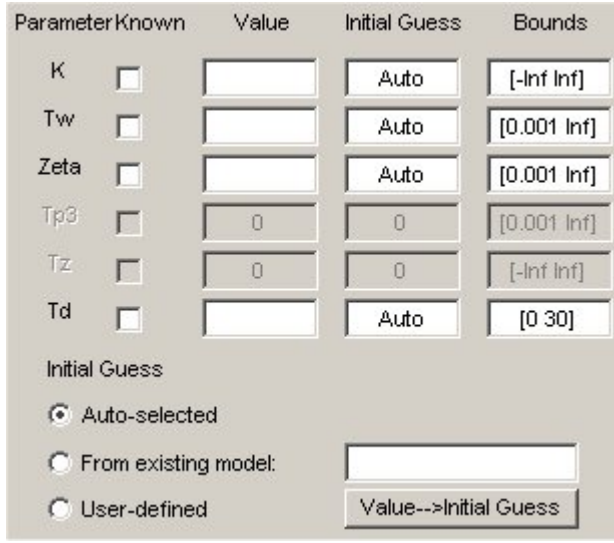
  Note You need at least two poles to allow underdamped modes (complex-conjugate pair).

- Select the **Zero** check box to include a zero, which is a numerator term other than a constant, or clear the check box to exclude the zero.
- Select the **Delay** check box to include a delay, or clear the check box to exclude the delay.
- Select the **Integrator** check box to include an integrator (self-regulating process), or clear the check box to exclude the integrator.

The **Parameter** area shows as many active parameters as you included in the model structure.

Note By default, the model **Name** is set to the acronym that reflects the model structure, as described in "Process Model Structure Specification" on page 5-44.

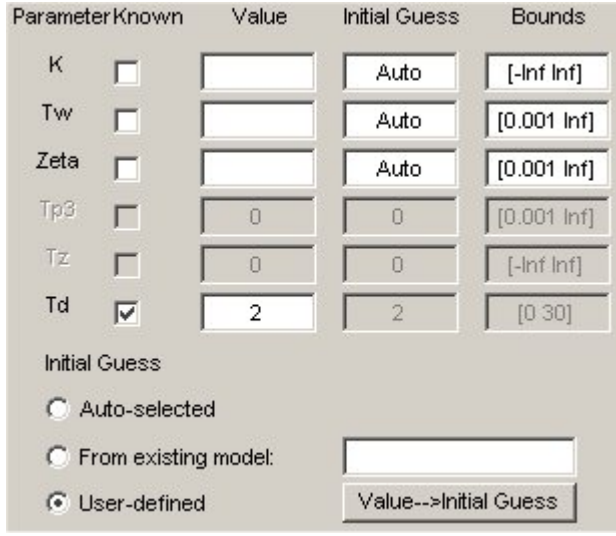
4 In the **Initial Guess** area, select `Auto-selected` to calculate the initial parameter values for the estimation. The **Initial Guess** column in the Parameter table displays `Auto`. If you do not have a good guess for the parameter values, `Auto` works better than entering an ad hoc value.



5 (Optional) If you approximately know a parameter value, enter this value in the **Initial Guess** column of the Parameter table, and press the **Enter** key. The estimation algorithm uses this value as a starting point. If you know a parameter value exactly, enter this value in the **Initial Guess** column, and press the **Enter** key. Select the corresponding **Known** check box in the table to fix its value.

If you know the range of possible values for a parameter, enter these values into the corresponding **Bounds** field to help the estimation algorithm. Press the **Enter** key after you specify the values.

For example, the following figure shows that the delay value `Td` is fixed at `2` s and is not estimated.



6 In the **Disturbance Model** drop-down list, select one of the available options. For more information about each option, see "Disturbance Model Structure for Process Models" on page 5-47.

7 In the **Focus** drop-down list, select how to weigh the relative importance of the fit at different frequencies. For more information about each option, see "Assigning Estimation Weightings" on page 5-9.

8 In the **Initial condition** drop-down list, specify how you want the algorithm to treat initial states. For more information about the available options, see "Specifying Initial Conditions for Iterative Estimation Algorithms" on page 5-49.

Tip If you get a bad fit, you might try setting a specific method for handling initial states, rather than choosing it automatically.

---

9 In the **Covariance** drop-down list, select `Estimate` if you want the algorithm to compute parameter uncertainties. Effects of such uncertainties are displayed on plots as model confidence regions.

To omit estimating uncertainty, select `None`. Skipping uncertainty computation might reduce computation time for complex models and large data sets.

10 In the **Model Name** field, edit the name of the model or keep the default. The name of the model should be unique in the Model Board.

11 To view the estimation progress, select the **Display Progress** check box. This opens a progress viewer window in which the estimation progress is reported.

12 Click **Regularization** to obtain regularized estimates of model parameters. Specify the regularization constants in the Regularization Options dialog box. To learn more, see "Regularized Estimates of Model Parameters" on page 1-48.

13 Click **Estimate** to add this model to the Model Board in the System Identification app.

14 To stop the search and save the results after the current iteration has been completed, click **Stop Iterations**. To continue iterations from the current model, click the **Continue** button to assign current parameter values as initial guesses for the next search.

Assigning Estimation Weightings

You can specify how the estimation algorithm weighs the fit at various frequencies. In the app, set **Focus** to one of the following options:

- `Prediction` — Uses the inverse of the noise model H to weigh the relative importance of how closely to fit the data in various frequency ranges. Corresponds to minimizing one-step-ahead prediction, which typically favors the fit over a short time interval. Optimized for output prediction applications.
- `Simulation` — Uses the input spectrum to weigh the relative importance of the fit in a specific frequency range. Does not use the noise model to weigh the relative importance of how closely to fit the data in various frequency ranges. Optimized for output simulation applications.
- `Stability` — Behaves the same way as the `Prediction` option, but also forces the model to be stable. For more information about model stability, see "Unstable Models" on page 17-120.
- `Filter` — Specify a custom filter to open the Estimation Focus dialog box, where you can enter a filter, as described in "Simple Passband Filter" on page 2-134 or "Defining a Custom Filter" on page 2-135. This prefiltering applies only for estimating the dynamics from input to output. The disturbance model is determined from the estimation data.

Next Steps

- Validate the model by selecting the appropriate check box in the **Model Views** area of the System Identification app. For more information about validating models, see “Validating Models After Estimation” on page 17-3.
- Refine the model by clicking the **Value —> Initial Guess** button to assign current parameter values as initial guesses for the next search, edit the **Name** field, and click **Estimate**.
- Export the model to the MATLAB workspace for further analysis by dragging it to the **To Workspace** rectangle in the System Identification app.

See Also

Related Examples

- “Identify Low-Order Transfer Functions (Process Models) Using System Identification App”
- “Estimate Process Models at the Command Line” on page 5-11

Estimate Process Models at the Command Line

In this section...
"Prerequisites" on page 5-11
"Using procest to Estimate Process Models" on page 5-11
"Estimate Process Models with Free Parameters" on page 5-12
"Estimate Process Models with Fixed Parameters" on page 5-14

Prerequisites

Before you can perform this task, you must have

- Input-output data as an `iddata` object or frequency response data as `frd` or `idfrd` objects. See "Representing Time- and Frequency-Domain Data Using iddata Objects" on page 2-50. For supported data formats, see "Data Supported by Process Models" on page 5-4.
- Performed any required data preprocessing operations. When working with time domain data, if you need to model nonzero offsets, such as when model contains integration behavior, do not detrend your data. In other cases, to improve the accuracy of your model, you should detrend your data. See "Ways to Prepare Data for System Identification" on page 2-6.

Using procest to Estimate Process Models

You can estimate process models using the iterative estimation method `procest` that minimizes the prediction errors to obtain maximum likelihood estimates. The resulting models are stored as `idproc` model objects.

You can use the following general syntax to both configure and estimate process models:

```
m = procest(data,mod_struc,opt)
```

`data` is the estimation data and `mod_struc` is one of the following:

- A character vector that represents the process model structure, as described in "Process Model Structure Specification" on page 5-44.

- A template `idproc` model. `opt` is an option set for configuring the estimation of the process model, such as handling of initial conditions, input offset and numerical search method.

Tip You do not need to construct the model object using `idproc` before estimation unless you want to specify initial parameter guesses, minimum/maximum bounds, or fixed parameter values, as described in "Estimate Process Models with Fixed Parameters" on page 5-14.

For more information about validating a process model, see "Validating Models After Estimation" on page 17-3.

You can use `procest` to refine parameter estimates of an existing process model, as described in "Refine Linear Parametric Models" on page 4-7.

For detailed information, see `procest` and `idproc`.

Estimate Process Models with Free Parameters

This example shows how to estimate the parameters of a first-order process model:

$$G(s) = \frac{K_p}{1 + sT_{p1}} e^{-sT_d}$$

This process has two inputs and the response from each input is estimated by a first-order process model. All parameters are free to vary.

Load estimation data.

```
load co2data
```

Specify known sample time of 0.5 min.

```
Ts = 0.5;
```

Split data set into estimation data `ze` and validation data `zv`.

```
ze = iddata(Output_exp1,Input_exp1,Ts,...
                          'TimeUnit','min');
```

```
zv = iddata(Output_exp2,Input_exp2,Ts,...
                          'TimeUnit','min');
```

Estimate model with one pole, a delay, and a first-order disturbance component. The data contains known offsets. Specify them using the `InputOffset` and `OutputOffset` options.

```
opt = procestOptions;
opt.InputOffset = [170;50];
opt.OutputOffset = -45;
opt.Display = 'on';
opt.DisturbanceModel = 'arma1';
m = procest(ze,'p1d',opt)
```

```
m =
Process model with 2 inputs: y = G11(s)u1 + G12(s)u2
  From input "u1" to output "y1":
               Kp
  G11(s) = ---------- * exp(-Td*s)
            1+Tp1*s

         Kp = 2.6553
        Tp1 = 0.15515
         Td = 2.3175

  From input "u2" to output "y1":
               Kp
  G12(s) = ---------- * exp(-Td*s)
            1+Tp1*s

         Kp = 9.9756
        Tp1 = 2.0653
         Td = 4.9195

  An additive ARMA disturbance model exists for output "y1":
      y = G u + (C/D)e

      C(s) = s + 2.676
      D(s) = s + 0.6228

Parameterization:
    'P1D'    'P1D'
   Number of free coefficients: 8
   Use "getpvec", "getcov" for parameters and their uncertainties.
```

```
Status:
Estimated using PROCEST on time domain data "ze".
Fit to estimation data: 91.07% (prediction focus)
FPE: 2.431, MSE: 2.412
```

Use dot notation to get the value of any model parameter. For example, get the value of dc gain parameter `Kp` .

```
m.Kp

ans =

    2.6553    9.9756
```

Estimate Process Models with Fixed Parameters

This example shows how to estimate a process model with fixed parameters.

When you know the values of certain parameters in the model and want to estimate only the values you do not know, you must specify the fixed parameters after creating the `idproc` model object. Use the following commands to prepare the data and construct a process model with one pole and a delay:

Load estimation data.

```
load co2data
```

Specify known sample time is 0.5 minutes.

```
Ts = 0.5;
```

Split data set into estimation data `ze` and validation data `zv`.

```
ze = iddata(Output_exp1,Input_exp1,Ts,...
                        'TimeUnit','min');
zv = iddata(Output_exp2,Input_exp2,Ts,...
                        'TimeUnit','min');
mod = idproc({'p1d','p1d'},'TimeUnit','min')

mod =
Process model with 2 inputs: y = G11(s)u1 + G12(s)u2
  From input 1 to output 1:
```

```
             Kp
  G11(s) = ---------- * exp(-Td*s)
           1+Tp1*s

         Kp = NaN
        Tp1 = NaN
         Td = NaN

  From input 2 to output 1:
             Kp
  G12(s) = ---------- * exp(-Td*s)
           1+Tp1*s

         Kp = NaN
        Tp1 = NaN
         Td = NaN

Parameterization:
    'P1D'    'P1D'
   Number of free coefficients: 6
   Use "getpvec", "getcov" for parameters and their uncertainties.

Status:
Created by direct construction or transformation. Not estimated.
```

The model parameters `Kp` , `Tp1` , and `Td` are assigned `NaN` values, which means that the parameters have not yet been estimated from the data.

Use the `Structure` model property to specify the initial guesses for unknown parameters, minimum/maximum parameter bounds and fix known parameters.

Set the value of `Kp` for the second transfer function to `10` and specify it as a fixed parameter. Initialize the delay values for the two transfer functions to 2 and 5 minutes, respectively. Specify them as free estimation parameters.

```
mod.Structure(2).Kp.Value = 10;
mod.Structure(2).Kp.Free = false;

mod.Structure(1).Tp1.Value = 2;
mod.Structure(2).Td.Value = 5;
```

Estimate `Tp1` and `Td` only.

```
mod_proc = procest(ze,mod)
```

```
mod_proc =
Process model with 2 inputs: y = G11(s)u1 + G12(s)u2
  From input "u1" to output "y1":
              Kp
  G11(s) = ---------- * exp(-Td*s)
            1+Tp1*s

         Kp = -3.2213
        Tp1 = 2.1707
         Td = 4.44

  From input "u2" to output "y1":
              Kp
  G12(s) = ---------- * exp(-Td*s)
            1+Tp1*s

         Kp = 10
        Tp1 = 2.0764
         Td = 4.5205

Parameterization:
    'P1D'    'P1D'
   Number of free coefficients: 5
   Use "getpvec", "getcov" for parameters and their uncertainties.

Status:
Estimated using PROCEST on time domain data "ze".
Fit to estimation data: 77.44%
FPE: 15.5, MSE: 15.39
```

In this case, the value of `Kp` is fixed, but `Tp1` and `Td` are estimated.

See Also

`idproc` | `procest`

Related Examples

- “Building and Estimating Process Models Using System Identification Toolbox™” on page 5-18
- “Estimate Process Models Using the App” on page 5-5

- “Loss Function and Model Quality Metrics” on page 1-64

Building and Estimating Process Models Using System Identification Toolbox™

This example shows how to build simple process models using System Identification Toolbox™. Techniques for creating these models and estimating their parameters using experimental data is described. This example requires Simulink®.

Introduction

This example illustrates how to build simple process models often used in process industry. Simple, low-order continuous-time transfer functions are usually employed to describe process behavior. Such models are described by IDPROC objects which represent the transfer function in a pole-zero-gain form.

Process models are of the basic type 'Static Gain + Time Constant + Time Delay'. They may be represented as:

$$P(s) = K.e^{-T_d*s}.\frac{1+T_z*s}{(1+T_{p1}*s)(1+T_{p2}*s)}$$

or as an integrating process:

$$P(s) = K.e^{-T_d*s}.\frac{1+T_z*s}{s(1+T_{p1}*s)(1+T_{p2}*s)}$$

where the user can determine the number of real poles (0, 1, 2 or 3), as well as the presence of a zero in the numerator, the presence of an integrator term (`1/s`) and the presence of a time delay (`Td`). In addition, an underdamped (complex) pair of poles may replace the real poles.

Representation of Process Models using IDPROC Objects

IDPROC objects define process models by using the letters `P` (for process model), `D` (for time delay), `Z` (for a zero) and `I` (for integrator). An integer will denote the number of poles. The models are generated by calling `idproc` with a character vector created using these letters.

For example:

```
idproc('P1') % transfer function with only one pole (no zeros or delay)
idproc('P2DIZ') % model with 2 poles, delay integrator and delay
idproc('P0ID') % model with no poles, but an integrator and a delay
```

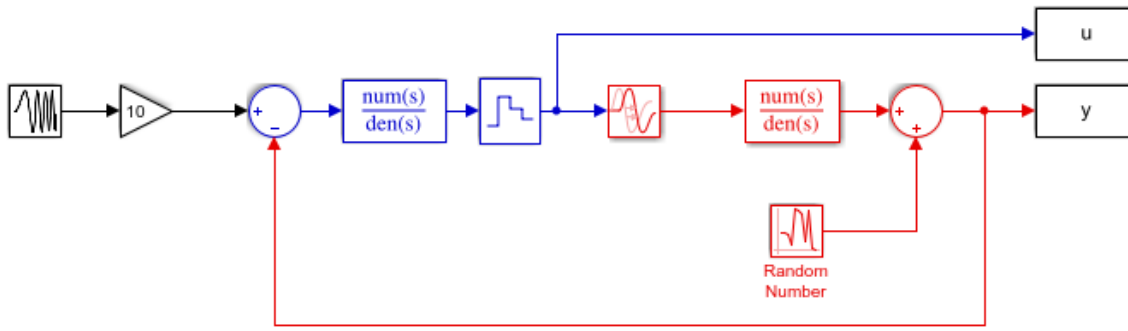
```
ans =
Process model with transfer function:
             Kp
  G(s) = ----------
         1+Tp1*s

        Kp = NaN
       Tp1 = NaN

Parameterization:
    'P1'
   Number of free coefficients: 2
   Use "getpvec", "getcov" for parameters and their uncertainties.

Status:
Created by direct construction or transformation. Not estimated.

ans =
Process model with transfer function:
                   1+Tz*s
  G(s) = Kp * ------------------- * exp(-Td*s)
              s(1+Tp1*s)(1+Tp2*s)

         Kp = NaN
        Tp1 = NaN
        Tp2 = NaN
         Td = NaN
         Tz = NaN

Parameterization:
    'P2DIZ'
   Number of free coefficients: 5
   Use "getpvec", "getcov" for parameters and their uncertainties.

Status:
Created by direct construction or transformation. Not estimated.

ans =
Process model with transfer function:
          Kp
  G(s) = --- * exp(-Td*s)
          s
```

```
        Kp = NaN
        Td = NaN

Parameterization:
    'P0DI'
   Number of free coefficients: 2
   Use "getpvec", "getcov" for parameters and their uncertainties.

Status:
Created by direct construction or transformation. Not estimated.
```

Creating an IDPROC Object (using a Simulink® Model as Example)

Consider the system described by the following SIMULINK model:

```
open_system('iddempr1')
set_param('iddempr1/Random Number','seed','0')
```



Red part: system to be modeled using IDPROC
Blue part: Controller

The red part is the system, the blue part is the controller and the reference signal is a swept sinusoid (a chirp signal). The data sample time is set to 0.5 seconds. As observed, the system is a continuous-time transfer function, and can hence be described using model objects in System Identification Toolbox, such as `idss`, `idpoly` or `idproc`.

Let us describe the system using `idpoly` and `idproc` objects. Using `idpoly` object, the system may be described as:

```
m0 = idpoly(1,0.1,1,1,[1 0.5],'Ts',0,'InputDelay',1.57,'NoiseVariance',0.01);
```

The IDPOLY form used above is useful for describing transfer functions of arbitrary orders. Since the system we are considering here is quite simple (one pole and no zeros), and is continuous-time, we may use the simpler IDPROC object to capture its dynamics:

```
m0p = idproc('p1d','Kp',0.2,'Tp1',2,'Td',1.57) % one pole+delay, with initial values
                                                % for gain, pole and delay specified.
```

```
m0p =
Process model with transfer function:
             Kp
  G(s) = ---------- * exp(-Td*s)
         1+Tp1*s

        Kp = 0.2
       Tp1 = 2
        Td = 1.57

Parameterization:
    'P1D'
   Number of free coefficients: 3
   Use "getpvec", "getcov" for parameters and their uncertainties.

Status:
Created by direct construction or transformation. Not estimated.
```

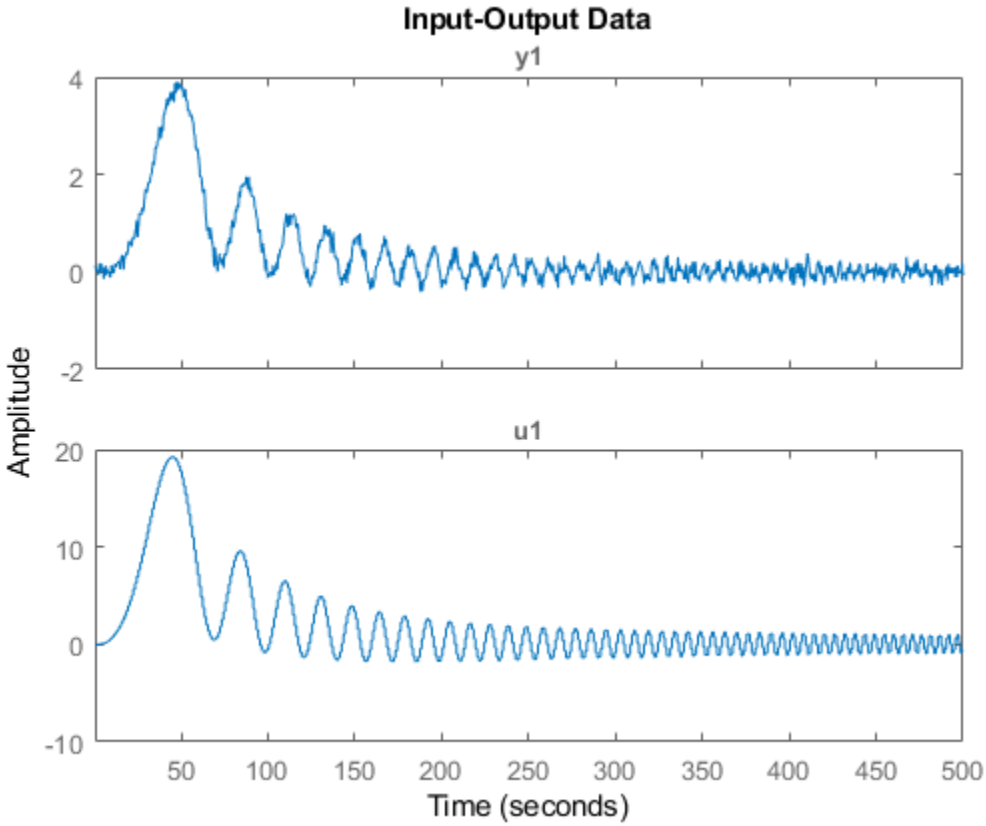
Estimating Parameters of IDPROC Models

Once a system is described by a model object, such as IDPROC, it may be used for estimation of its parameters using measurement data. As an example, we consider the problem of estimation of parameters of the Simulink model's system (red portion) using simulation data. We begin by acquiring data for estimation:

```
sim('iddempr1')
dat1e = iddata(y,u,0.5); % The IDDATA object for storing measurement data
```

Let us look at the data:

```
plot(dat1e)
```



We can identify a process model using `procest` command, by providing the same structure information specified to create IDPROC models. For example, the 1-pole+delay model may be estimated by calling `procest` as follows:

```
m1 = procest(dat1e,'p1d'); % estimation of idproc model using data 'dat1e'.

% Check the result of estimation:
m1
```

```
m1 =
Process model with transfer function:
             Kp
  G(s) = ---------- * exp(-Td*s)
```

```
          1+Tp1*s

         Kp = 0.20045
        Tp1 = 2.0431
         Td = 1.499

Parameterization:
    'P1D'
   Number of free coefficients: 3
   Use "getpvec", "getcov" for parameters and their uncertainties.

Status:
Estimated using PROCEST on time domain data "dat1e".
Fit to estimation data: 87.34%
FPE: 0.01069, MSE: 0.01062
```

To get information about uncertainties, use

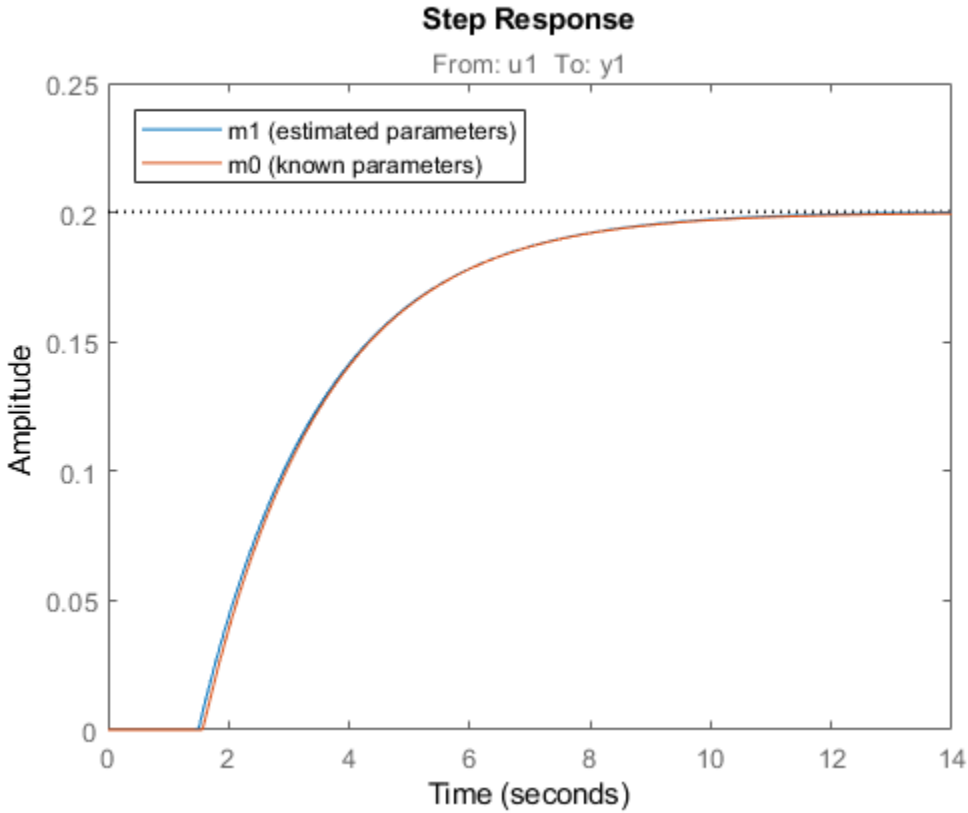
```
present(m1)


m1 =
Process model with transfer function:
             Kp
  G(s) = ---------- * exp(-Td*s)
          1+Tp1*s

         Kp = 0.20045 +/- 0.00077275
        Tp1 = 2.0431 +/- 0.061216
         Td = 1.499 +/- 0.040854

Parameterization:
    'P1D'
   Number of free coefficients: 3
   Use "getpvec", "getcov" for parameters and their uncertainties.

Status:
Termination condition: Near (local) minimum, (norm(g) < tol).
Number of iterations: 4, Number of function evaluations: 9

Estimated using PROCEST on time domain data "dat1e".
Fit to estimation data: 87.34%
FPE: 0.01069, MSE: 0.01062
More information in model's "Report" property.
```

The model parameters, `K`, `Tp1` and `Td` are now shown with one standard deviation uncertainty range.

Computing Time and Frequency Response of IDPROC Models

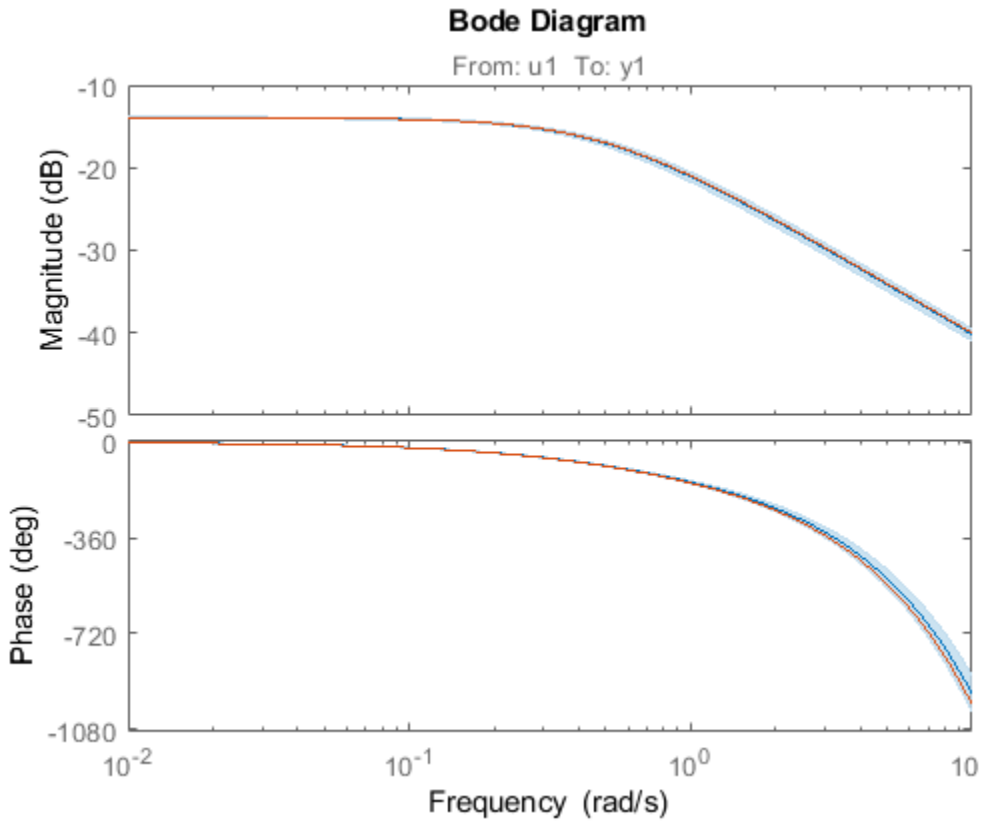
The model `m1` estimated above is an IDPROC model object to which all of the toolbox's model commands can be applied:

```
step(m1,m0) %step response of models m1 (estimated) and m0 (actual)
legend('m1 (estimated parameters)','m0 (known parameters)','location','northwest')
```



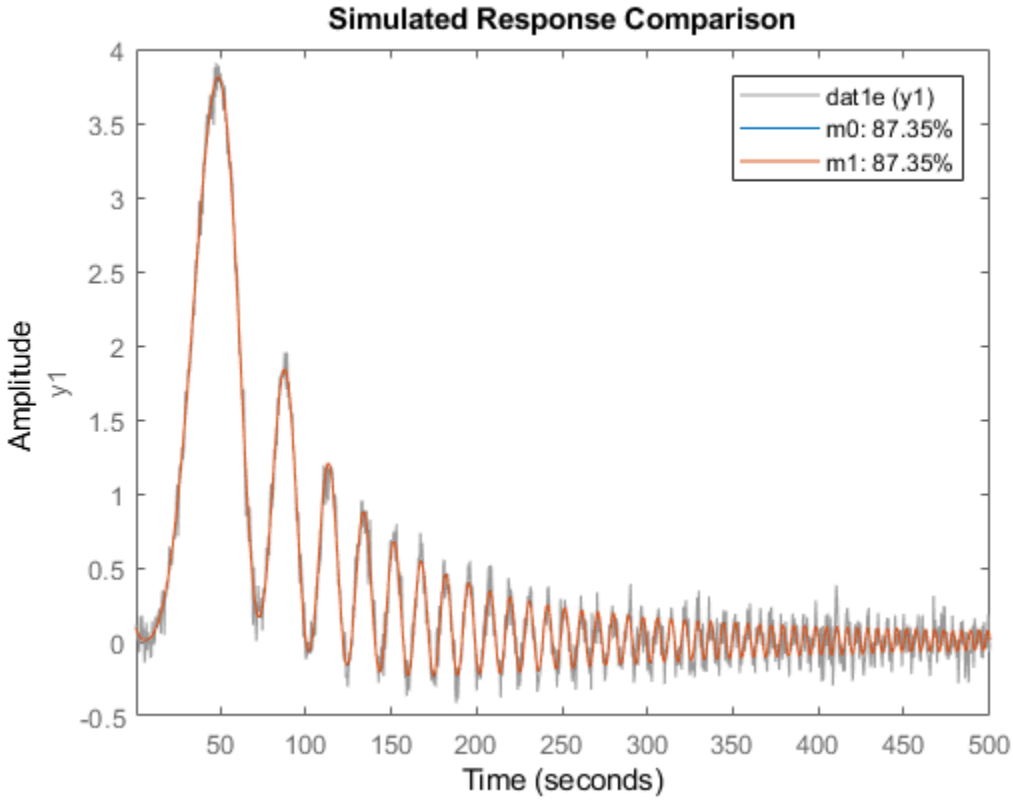
Bode response with confidence region corresponding to 3 standard deviations may be computed by doing:

```
h = bodeplot(m1,m0);
showConfidence(h,3)
```



Similarly, the measurement data may be compared to the models outputs using `compare` as follows:

```
compare(dat1e,m0,m1)
```



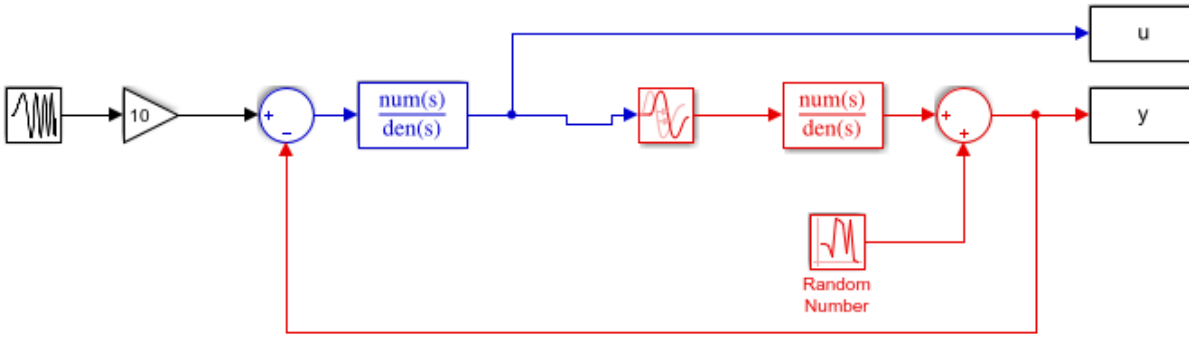
Other operations such as `sim`, `impulse`, `c2d` are also available, just as they are for other model objects.

```
bdclose('iddempr1')
```

Accommodating the Effect of Intersample Behavior in Estimation

It may be important (at least for slow sampling) to consider the intersample behavior of the input data. To illustrate this, let us study the same system as before, but without the sample-and-hold circuit:

```
open_system('iddempr5')
```



Red part: system to be modeled using IDPROC
Blue part: Controller

Simulate this system with the same sample time:

```
sim('iddempr5')
dat1f = iddata(y,u,0.5); % The IDDATA object for the simulated data
```

We estimate an IDPROC model using `data1f` while also imposing an upper bound on the allowable value delay. We will use 'lm' as search method and also choose to view the estimation progress.

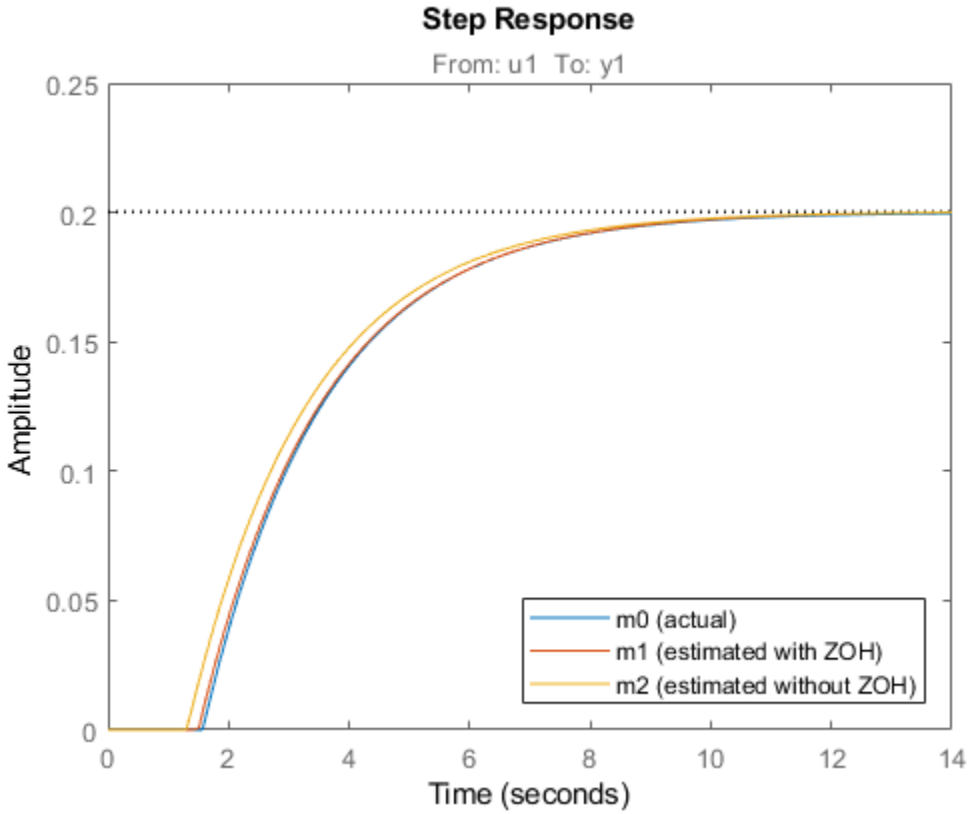
```
m2_init = idproc('P1D');
m2_init.Structure.Td.Maximum = 2;
opt = procestOptions('SearchMethod','lm','Display','on');
m2 = procest(dat1f,m2_init,opt);
m2
```

```
m2 =
Process model with transfer function:
                Kp
  G(s) = ---------- * exp(-Td*s)
           1+Tp1*s

         Kp = 0.20038
        Tp1 = 2.01
         Td = 1.31

Parameterization:
    'P1D'
```

```
   Number of free coefficients: 3
   Use "getpvec", "getcov" for parameters and their uncertainties.

Status:
Estimated using PROCEST on time domain data "dat1f".
Fit to estimation data: 87.26%
FPE: 0.01067, MSE: 0.01061
```

This model has a slightly less precise estimate of the delay than the previous one, m1:

```
[m0p.Td, m1.Td, m2.Td]
step(m0,m1,m2)
legend('m0 (actual)','m1 (estimated with ZOH)','m2 (estimated without ZOH)','location',


ans =

    1.5700    1.4990    1.3100
```



However, by telling the estimation process that the intersample behavior is first-order-hold (an approximation to the true continuous) input, we do better:

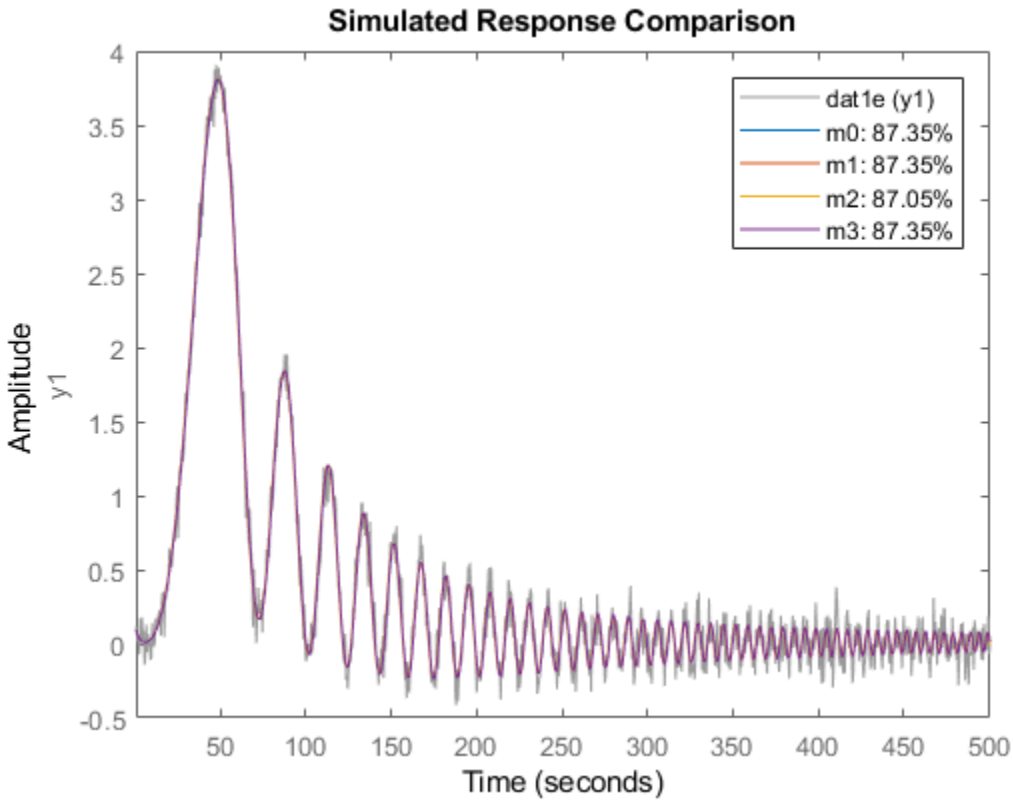
```
dat1f.InterSample = 'foh';
m3 = procest(dat1f,m2_init,opt);
```

Compare the four models m0 (true) m1 (obtained from zoh input) m2 (obtained for continuous input, with zoh assumption) and m3 (obtained for the same input, but with foh assumption)

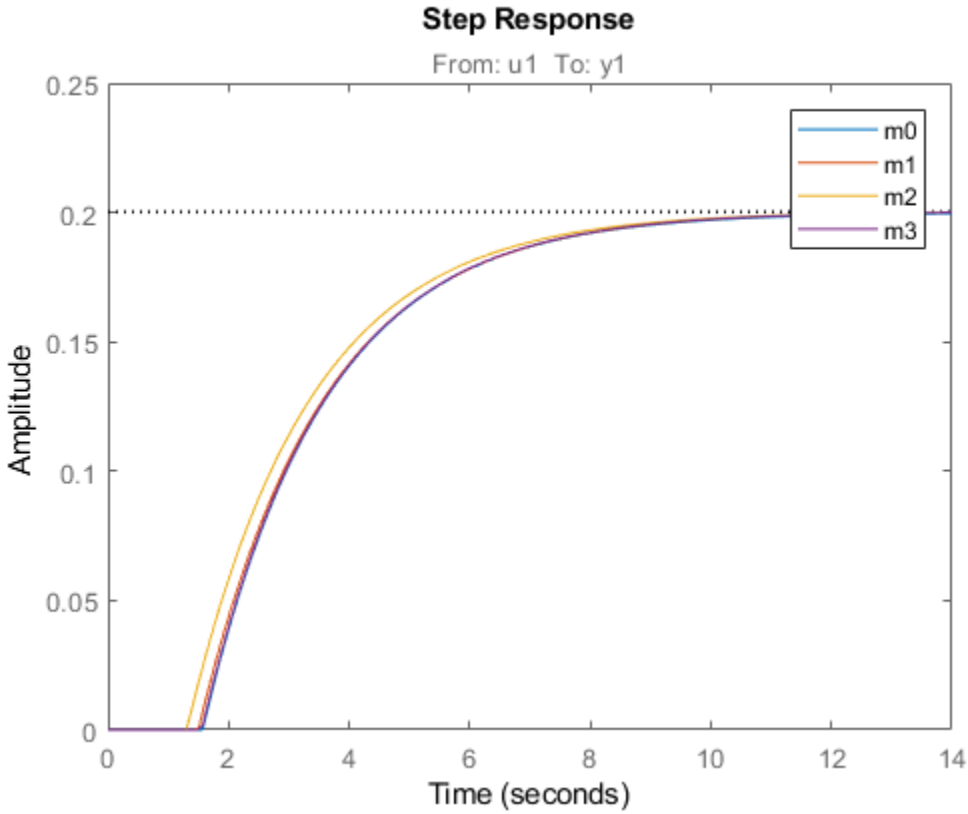
```
[m0p.Td, m1.Td, m2.Td, m3.Td]
compare(dat1e,m0,m1,m2,m3)
```

```
ans =
```

```
    1.5700    1.4990    1.3100    1.5570
```



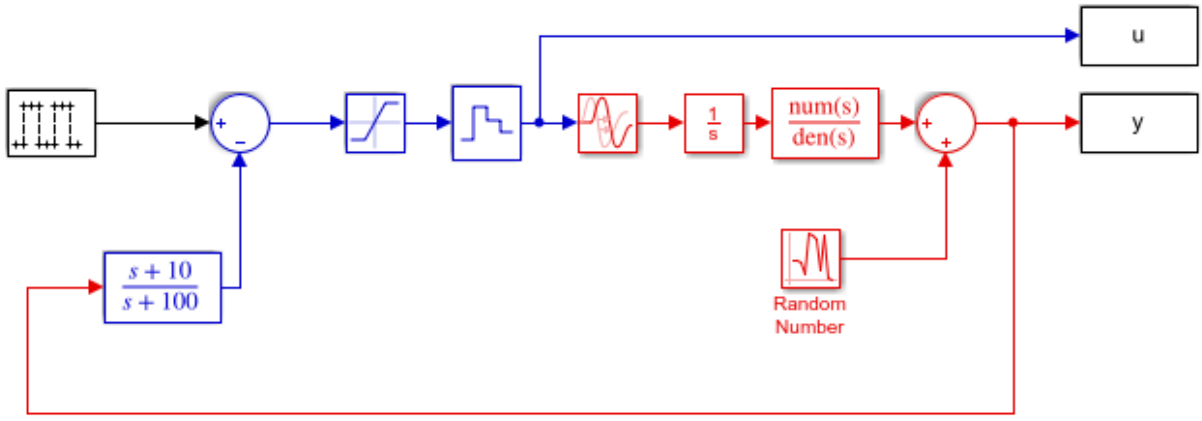

```
step(m0,m1,m2,m3)
legend('m0','m1','m2','m3')
bdclose('iddempr5')
```



Modeling a System Operating in Closed Loop

Let us now consider a more complex process, with integration, that is operated in closed loop:

```
open_system('iddempr2')
```



Red part: system to be modeled using IDPROC
Blue part: Controller

The true system can be represented by:

```
m0 = idproc('P2ZDI','Kp',1,'Tp1',1,'Tp2',5,'Tz',3,'Td',2.2);
```

The process is controlled by a PD regulator with limited input amplitude and a zero order hold device. The sample time is 1 second.

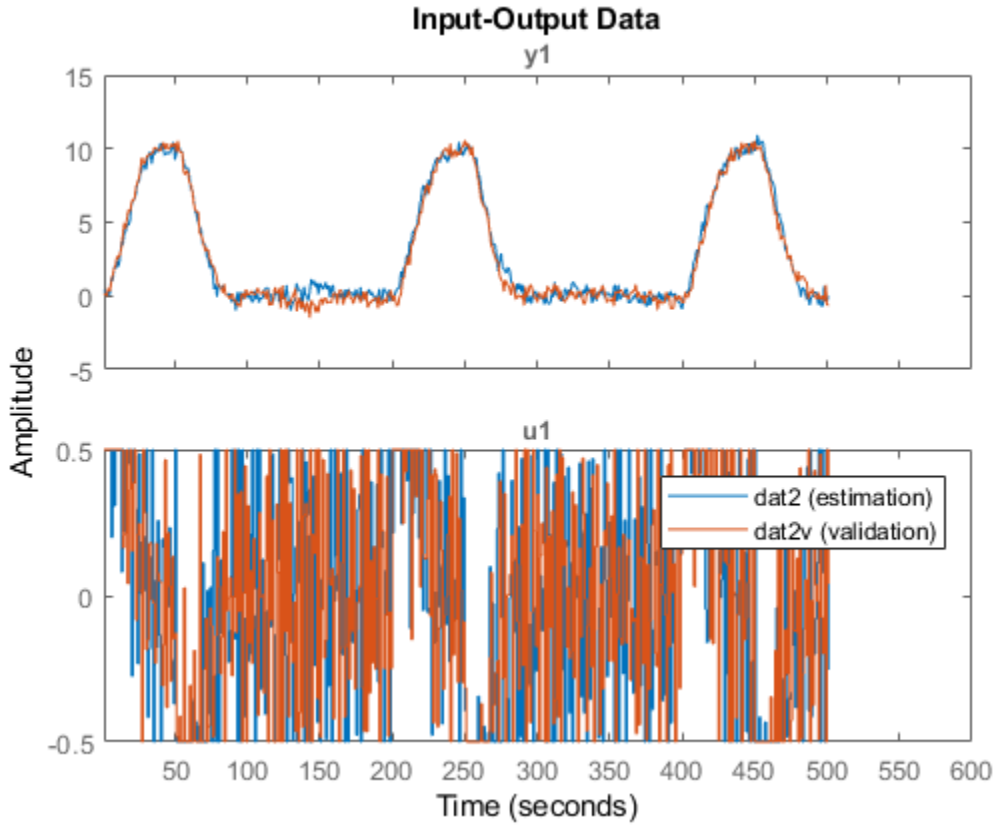
```
set_param('iddempr2/Random Number','seed','0')
sim('iddempr2')
dat2 = iddata(y,u,1); % IDDATA object for estimation
```

Two different simulations are made, the first for estimation and the second one for validation purposes.

```
set_param('iddempr2/Random Number','seed','13')
sim('iddempr2')
dat2v = iddata(y,u,1); % IDDATA object for validation purpose
```

Let us look at the data (estimation and validation).

```
plot(dat2,dat2v)
legend('dat2 (estimation)','dat2v (validation)')
```



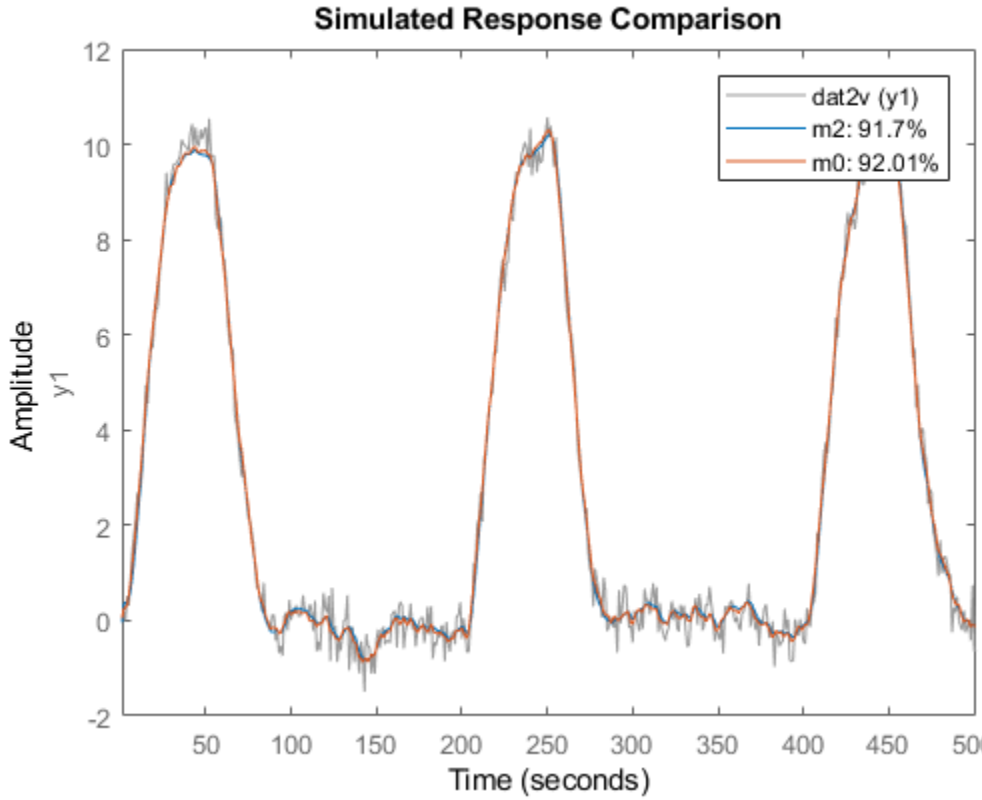
Let us now perform estimation using `dat2`.

```
Warn = warning('off','Ident:estimation:underdampedIDPROC');
m2_init = idproc('P2ZDI');
m2_init.Structure.Td.Maximum = 5;
m2_init.Structure.Tp1.Maximum = 2;
opt = processOptions('SearchMethod','lsqnonlin','Display','on');
opt.SearchOption.MaxIter  = 100;
m2 = process(dat2, m2_init, opt)
```

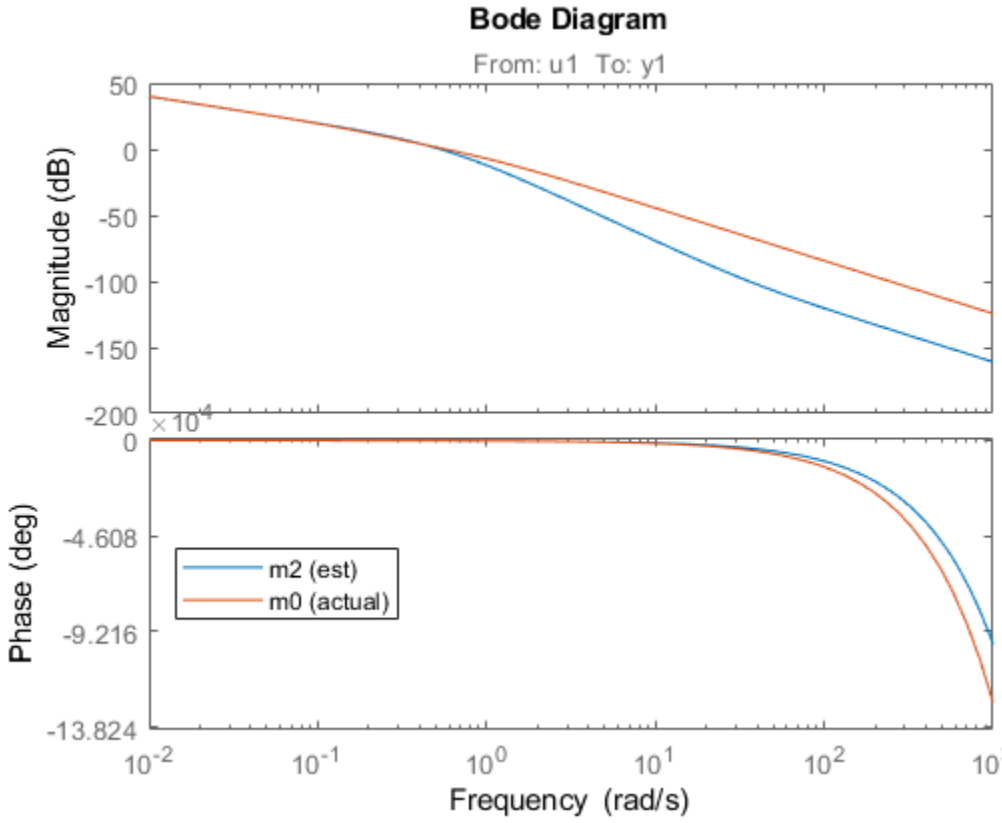
```
m2 =
Process model with transfer function:
                  1+Tz*s
```

```
  G(s) = Kp * -------------------- * exp(-Td*s)
              s(1+Tp1*s)(1+Tp2*s)

          Kp = 0.98558
         Tp1 = 2
         Tp2 = 1.4842
          Td = 1.711
          Tz = 0.027145

Parameterization:
    'P2DIZ'
   Number of free coefficients: 5
   Use "getpvec", "getcov" for parameters and their uncertainties.

Status:
Estimated using PROCEST on time domain data "dat2".
Fit to estimation data: 91.51%
FPE: 0.1129, MSE: 0.1094
```

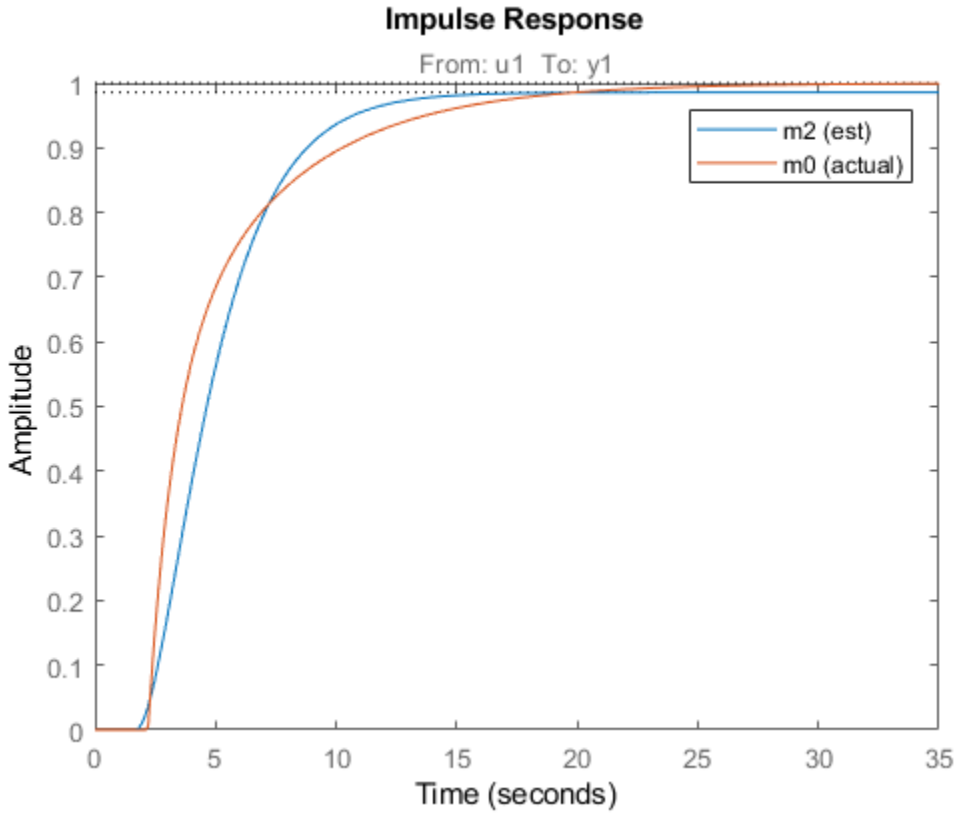
```
compare(dat2v,m2,m0) % Gives very good agreement with data
```




```
bode(m2,m0)
legend({'m2 (est)','m0 (actual)'},'location','west')
```




```
impulse(m2,m0)
legend({'m2 (est)','m0 (actual)'})
```



Compare also with the parameters of the true system:

```
present(m2)
[getpvec(m0), getpvec(m2)]
```

```
m2 =
Process model with transfer function:
                    1+Tz*s
  G(s) = Kp * ------------------- * exp(-Td*s)
              s(1+Tp1*s)(1+Tp2*s)

         Kp = 0.98558 +/- 0.013696
        Tp1 = 2 +/- 8.2115
```

```
        Tp2 = 1.4842 +/- 10.172
         Td = 1.711 +/- 63.736
         Tz = 0.027145 +/- 65.538

Parameterization:
    'P2DIZ'
   Number of free coefficients: 5
   Use "getpvec", "getcov" for parameters and their uncertainties.

Status:
Termination condition: Change in cost was less than the specified tolerance.
Number of iterations: 3, Number of function evaluations: 4

Estimated using PROCEST on time domain data "dat2".
Fit to estimation data: 91.51%
FPE: 0.1129, MSE: 0.1094
More information in model's "Report" property.

ans =

    1.0000    0.9856
    1.0000    2.0000
    5.0000    1.4842
    2.2000    1.7110
    3.0000    0.0271
```

A word of caution. Identification of several real time constants may sometimes be an ill-conditioned problem, especially if the data are collected in closed loop.

To illustrate this, let us estimate a model based on the validation data:

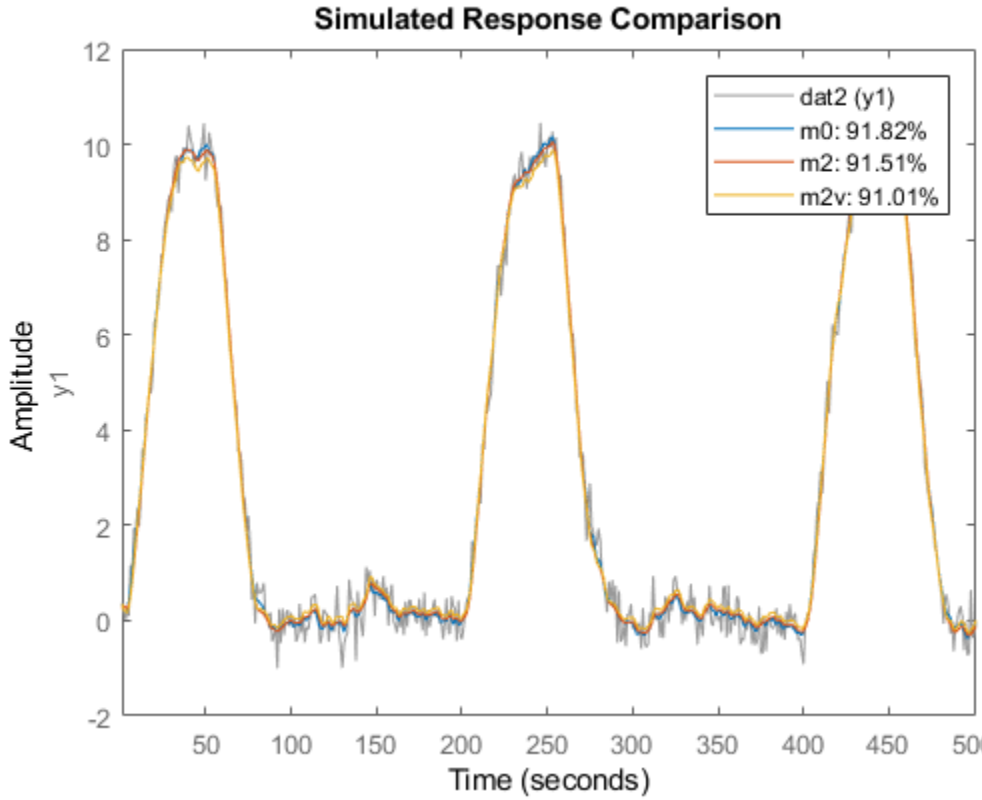
```
m2v = process(dat2v, m2_init, opt)
[getpvec(m0), getpvec(m2), getpvec(m2v)]
```

```
m2v =
Process model with transfer function:
                    1+Tz*s
  G(s) = Kp * ------------------- * exp(-Td*s)
              s(1+Tp1*s)(1+Tp2*s)

         Kp = 0.95747
        Tp1 = 1.999
        Tp2 = 0.60819
```

```
         Td = 2.314
         Tz = 0.0010561

Parameterization:
    'P2DIZ'
   Number of free coefficients: 5
   Use "getpvec", "getcov" for parameters and their uncertainties.

Status:
Estimated using PROCEST on time domain data "dat2v".
Fit to estimation data: 90.65%
FPE: 0.1397, MSE: 0.1353

ans =

    1.0000    0.9856    0.9575
    1.0000    2.0000    1.9990
    5.0000    1.4842    0.6082
    2.2000    1.7110    2.3140
    3.0000    0.0271    0.0011
```

This model has much worse parameter values. On the other hand, it performs nearly identically to the true system m0 when tested on the other data set dat2:

```
compare(dat2,m0,m2,m2v)
```



Fixing Known Parameters During Estimation

Suppose we know from other sources that one time constant is 1:

```
m2v.Structure.Tp1.Value = 1;
m2v.Structure.Tp1.Free = false;
```

We can fix this value, while estimating the other parameters:

```
m2v = procest(dat2v,m2v)
%
```

```
m2v =
Process model with transfer function:
```

```
                  1+Tz*s
  G(s) = Kp * ------------------- * exp(-Td*s)
              s(1+Tp1*s)(1+Tp2*s)

         Kp = 1.0111
        Tp1 = 1
        Tp2 = 5.3014
         Td = 2.195
         Tz = 3.231

Parameterization:
    'P2DIZ'
   Number of free coefficients: 4
   Use "getpvec", "getcov" for parameters and their uncertainties.

Status:
Estimated using PROCEST on time domain data "dat2v".
Fit to estimation data: 92.05%
FPE: 0.09952, MSE: 0.09794
```

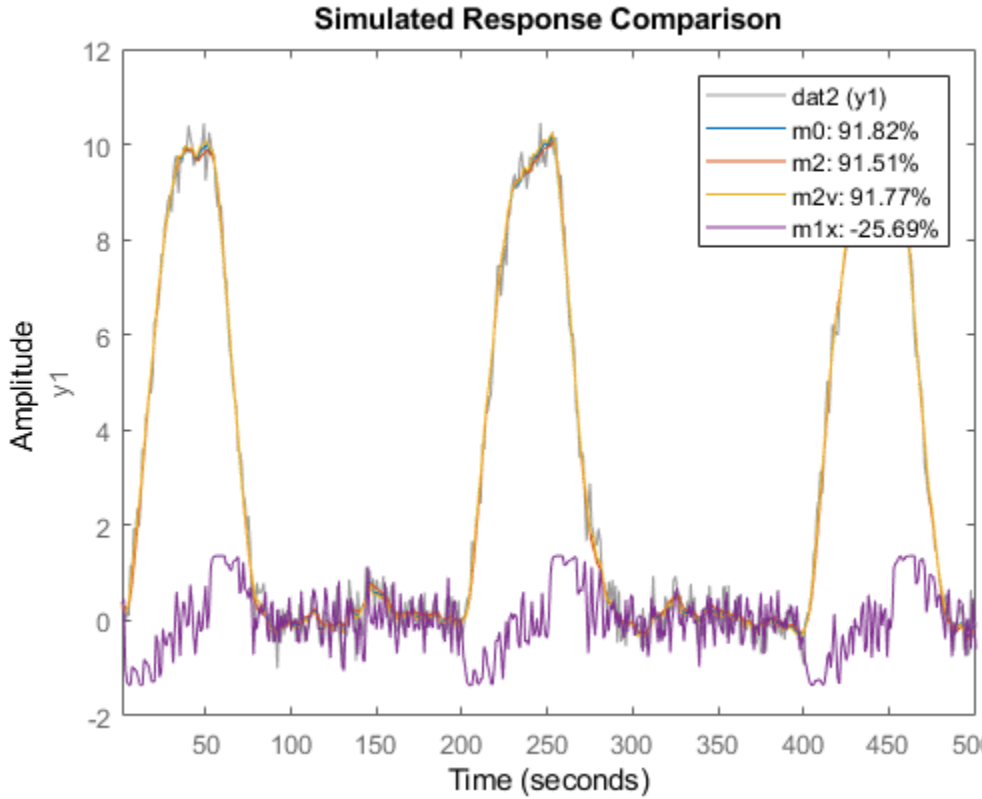
As observed, fixing `Tp1` to its known value dramatically improves the estimates of the remaining parameters in model `m2v`.

This also indicates that simple approximation should do well on the data:

```
m1x_init = idproc('P2D'); % simpler structure (no zero, no integrator)
m1x_init.Structure.Td.Maximum = 2;
m1x = procest(dat2v, m1x_init)
compare(dat2,m0,m2,m2v,m1x)
```

```
m1x =
Process model with transfer function:
                Kp
  G(s) = ----------------- * exp(-Td*s)
         (1+Tp1*s)(1+Tp2*s)

         Kp = -2.7149
        Tp1 = 9.8914e-07
        Tp2 = 0.59802
         Td = 0.615

Parameterization:
    'P2D'
   Number of free coefficients: 4
```

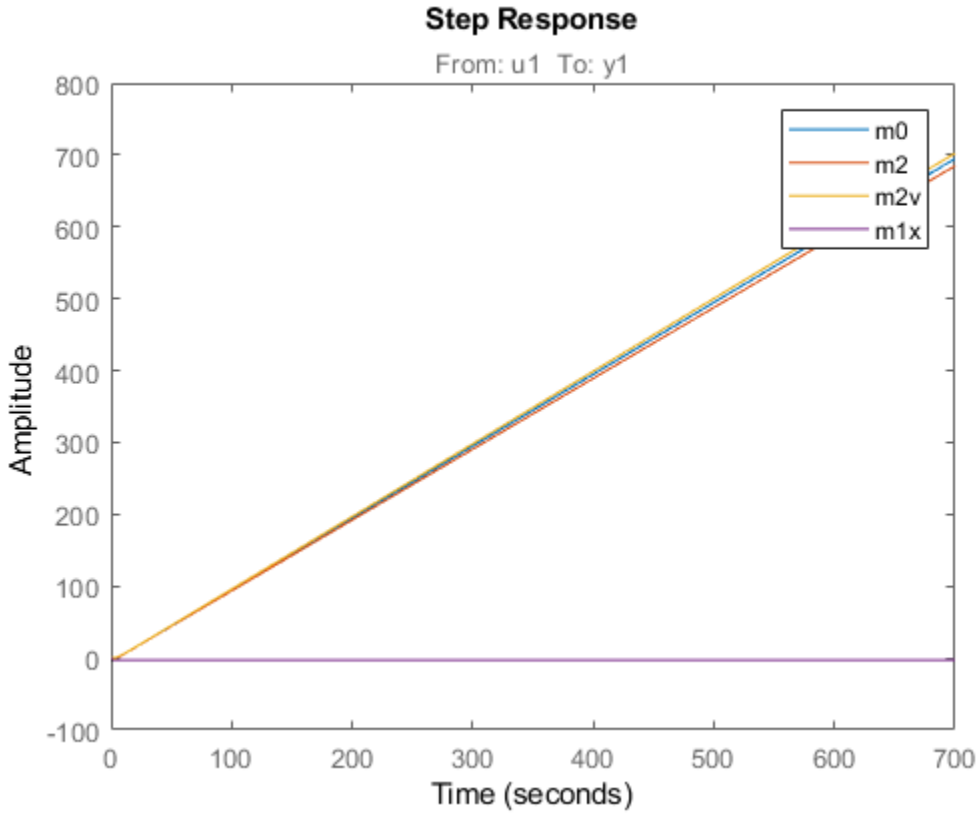
```
   Use "getpvec", "getcov" for parameters and their uncertainties.

Status:
Estimated using PROCEST on time domain data "dat2v".
Fit to estimation data: -23.27%
FPE: 23.92, MSE: 23.54
```



Thus, the simpler model is able to estimate system output pretty well. However, `m1x` does not contain any integration, so the open loop long time range behavior will be quite different:

```
step(m0,m2,m2v,m1x)
legend('m0','m2','m2v','m1x')
```

```
bdclose('iddempr2')
warning(Warn)
```



Additional Information

For more information on identification of dynamic systems with System Identification Toolbox visit the System Identification Toolbox product information page.

Process Model Structure Specification

This topic describes how to specify the model structure in the estimation procedures “Estimate Process Models Using the App” on page 5-5 and “Estimate Process Models at the Command Line” on page 5-11.

In the System Identification app, specify the model structure by selecting the number of real or complex poles, and whether to include a zero, delay, and integrator. The resulting transfer function is displayed in the Process Models dialog box.

At the command line, specify the model structure using an acronym that includes the following letters and numbers:

- (Required) `P` for a process model
- (Required) `0`, `1`, `2` or `3` for the number of poles
- (Optional) `D` to include a time-delay term e^{-sT_d}
- (Optional) `Z` to include a process zero (numerator term)
- (Optional) `U` to indicate possible complex-valued (underdamped) poles
- (Optional) `I` to indicate enforced integration

Typically, you specify the model-structure acronym as an argument in the estimation command `procest`:

- `procest(data,'P1D')` to estimate the following structure:

  $$G(s) = \frac{K_p}{1 + sT_{p1}} e^{-sT_d}$$

- `procest(data,'P2ZU')` to estimate the following structure:

  $$G(s) = \frac{K_p \left(1 + sT_z\right)}{1 + 2s\zeta T_w + s^2 T_w^2}$$

- `procest(data,'P0ID')` to estimate the following structure:

  $$G(s) = \frac{K_p}{s} e^{-sT_d}$$

- `procest(data,'P3Z')` to estimate the following structure:

$$G(s) = \frac{K_p\left(1 + sT_z\right)}{\left(1 + sT_{p1}\right)\left(1 + sT_{p2}\right)\left(1 + sT_{p3}\right)}$$

For more information about estimating models, see “Estimate Process Models at the Command Line” on page 5-11.

See Also

More About

- “What Is a Process Model?” on page 5-2

Estimating Multiple-Input, Multi-Output Process Models

If your model contains multiple inputs, multiple outputs, or both, you can specify whether to estimate the same transfer function for all input-output pairs, or a different transfer function for each. The information in this section supports the estimation procedures “Estimate Process Models Using the App” on page 5-5 and “Estimate Process Models at the Command Line” on page 5-11.

In the System Identification app — To fit a data set with multiple inputs, or multiple outputs, or both, in the Process Models dialog box, configure the process model settings for one input-output pair at a time. Use the input and output selection lists to switch to a different input/output pair.

If you want the same transfer function to apply to all input/output pairs, select the **All same** check box. To apply a different structure to each channel, leave this check box clear, and create a different transfer function for each input.

At the command line — Specify the model structure as a cell array of character vectors in the estimation command `procest`. For example, use this command to specify the first-order transfer function for the first input, and a second-order model with a zero and an integrator for the second input:

```
m = idproc({'P1','P2ZI'})
m = procest(data,m)
```

To apply the same structure to all inputs, define a single structure in `idproc`.

See Also

More About

- “Data Supported by Process Models” on page 5-4

Disturbance Model Structure for Process Models

This section describes how to specify a noise model in the estimation procedures “Estimate Process Models Using the App” on page 5-5 and “Estimate Process Models at the Command Line” on page 5-11.

In addition to the transfer function G, a linear system can include an additive noise term He, as follows:

$$y = Gu + He$$

where e is white noise.

You can estimate only the dynamic model G, or estimate both the dynamic model and the disturbance model H. For process models, H is a rational transfer function C/D, where the C and D polynomials for a first- or second-order ARMA model.

In the System Identification app, to specify whether to include or exclude a noise model in the Process Models dialog box, select one of the following options from the **Disturbance Model** list:

- `None` — The algorithm does not estimate a noise model (C=D=1). This option also sets **Focus** to `Simulation`.
- `Order 1` — Estimates a noise model as a continuous-time, first-order ARMA model.
- `Order 2` — Estimates a noise model as a continuous-time, second-order ARMA model.

At the command line, specify the disturbance model using the `procestOptions` option set. For example, use this command to estimate a first-order transfer function and a first-order noise model:

```
opt = procestOptions;
opt.DisturbanceModel = 'arma1';
model = procest(data, 'P1D', opt);
```

For a complete list of values for the `DisturbanceModel` model property, see the `procestOptions` reference page.

See Also

`procestOptions`

More About

- “Estimate Process Models Using the App” on page 5-5
- “Estimate Process Models at the Command Line” on page 5-11

Specifying Initial Conditions for Iterative Estimation Algorithms

You can optionally specify how the iterative algorithm treats initial conditions for estimation of model parameters. This information supports the estimation procedures “Estimate Process Models Using the App” on page 5-5 and “Estimate Process Models at the Command Line” on page 5-11.

In the System Identification app, set **Initial condition** to one of the following options:

- `Zero` — Sets all initial states to zero.
- `Estimate` — Treats the initial states as an unknown vector of parameters and estimates these states from the data.
- `Backcast` — Estimates initial states using a backward filtering method (least-squares fit).
- `U-level est` — Estimates both the initial conditions and input offset levels. For multiple inputs, the input level for each input is estimated individually. Use if you included an integrator in the transfer function.
- `Auto` — Automatically chooses one of the preceding options based on the estimation data. If the initial conditions have negligible effect on the prediction errors, they are taken to be zero to optimize algorithm performance.

At the command line, specify the initial conditions using the `InitialCondition` model estimation option, configured using the `procestOptions` command. For example, use this command to estimate a first-order transfer function and set the initial states to zero:

```
opt = procestOptions('InitialCondition','zero');
model = procest(data,'P1D',opt)
```

See Also

`procestOptions`

More About

- “Estimate Process Models Using the App” on page 5-5
- “Estimate Process Models at the Command Line” on page 5-11

6

Identifying Input-Output Polynomial Models

- "What Are Polynomial Models?" on page 6-2
- "Data Supported by Polynomial Models" on page 6-8
- "Preliminary Step – Estimating Model Orders and Input Delays" on page 6-10
- "Estimate Polynomial Models in the App" on page 6-18
- "Estimate Polynomial Models at the Command Line" on page 6-23
- "Polynomial Sizes and Orders of Multi-Output Polynomial Models" on page 6-27
- "Specifying Initial States for Iterative Estimation Algorithms" on page 6-32
- "Polynomial Model Estimation Algorithms" on page 6-34
- "Estimate Models Using armax" on page 6-35

What Are Polynomial Models?

In this section...
"Polynomial Model Structure" on page 6-2
"Understanding the Time-Shift Operator q" on page 6-3
"Different Configurations of Polynomial Models" on page 6-4
"Continuous-Time Representation of Polynomial Models" on page 6-6
"Multi-Output Polynomial Models" on page 6-6

Polynomial Model Structure

A polynomial model uses a generalized notion of transfer functions to express the relationship between the input, $u(t)$, the output $y(t)$, and the noise $e(t)$ using the equation:

$$A(q)y(t) = \sum_{i=1}^{nu} \frac{B_i(q)}{F_i(q)} u_i\left(t - nk_i\right) + \frac{C(q)}{D(q)} e(t)$$

The variables A, B, C, D, and F are polynomials expressed in the time-shift operator `q^-1`. u_i is the ith input, nu is the total number of inputs, and nk_i is the ith input delay that characterizes the transport delay. The variance of the white noise *e(t)* is assumed to be λ. For more information about the time-shift operator, see "Understanding the Time-Shift Operator q" on page 6-3.

In practice, not all the polynomials are simultaneously active. Often, simpler forms, such as ARX, ARMAX, Output-Error, and Box-Jenkins are employed. You also have the option of introducing an integrator in the noise source so that the general model takes the form:

$$A(q)y(t) = \sum_{i=1}^{nu} \frac{B_i(q)}{F_i(q)} u_i\left(t - nk_i\right) + \frac{C(q)}{D(q)} \frac{1}{1 - q^{-1}} e(t)$$

For more information, see "Different Configurations of Polynomial Models" on page 6-4.

You can estimate polynomial models using time or frequency domain data.

For estimation, you must specify the *model order* as a set of integers that represent the number of coefficients for each polynomial you include in your selected structure—*na* for

A, *nb* for *B*, *nc* for *C*, *nd* for *D*, and *nf* for *F*. You must also specify the number of samples *nk* corresponding to the input delay—*dead time*—given by the number of samples before the output responds to the input.

The number of coefficients in denominator polynomials is equal to the number of poles, and the number of coefficients in the numerator polynomials is equal to the number of zeros plus 1. When the dynamics from *u(t)* to *y(t)* contain a delay of *nk* samples, then the first *nk* coefficients of *B* are zero.

For more information about the family of transfer-function models, see the corresponding section in *System Identification: Theory for the User*, Second Edition, by Lennart Ljung, Prentice Hall PTR, 1999.

Understanding the Time-Shift Operator q

The general polynomial equation is written in terms of the time-shift operator q^{-1}. To understand this time-shift operator, consider the following discrete-time difference equation:

$$y(t) + a_1 y(t-T) + a_2 y(t-2T) = b_1 u(t-T) + b_2 u(t-2T)$$

where *y(t)* is the output, *u(t)* is the input, and T is the sample time. q^{-1} is a time-shift operator that compactly represents such difference equations using $q^{-1}u(t) = u(t-T)$:

$$y(t) + a_1 q^{-1} y(t) + a_2 q^{-2} y(t) = b_1 q^{-1} u(t) + b_2 q^{-2} u(t)$$

or

$$A(q)y(t) = B(q)u(t)$$

In this case, $A(q) = 1 + a_1 q^{-1} + a_2 q^{-2}$ and $B(q) = b_1 q^{-1} + b_2 q^{-2}$.

Note This q description is completely equivalent to the Z-transform form: q corresponds to z.

Different Configurations of Polynomial Models

These model structures are subsets of the following general polynomial equation:

$$A(q)y(t) = \sum_{i=1}^{nu} \frac{B_i(q)}{F_i(q)} u_i\left(t - nk_i\right) + \frac{C(q)}{D(q)} e(t)$$

The model structures differ by how many of these polynomials are included in the structure. Thus, different model structures provide varying levels of flexibility for modeling the dynamics and noise characteristics.

The following table summarizes common linear polynomial model structures supported by the System Identification Toolbox product. If you have a specific structure in mind for your application, you can decide whether the dynamics and the noise have common or different poles. *A(q)* corresponds to poles that are common for the dynamic model and the noise model. Using common poles for dynamics and noise is useful when the disturbances enter the system at the input. F_i determines the poles unique to the system dynamics, and *D* determines the poles unique to the disturbances.

Model Structure	Equation	Description
ARX	$A(q)y(t) = \sum_{i=1}^{nu} B_i(q)u_i\left(t - nk_i\right) + e(t)$	The noise model is $\frac{1}{A}$ and the noise is coupled to the dynamics model. ARX does not let you model noise and dynamics independently. Estimate an ARX model to obtain a simple model at good signal-to-noise ratios.
ARIX	$Ay = Bu + \frac{1}{1 - q^{-1}} e$	Extends the ARX structure by including an integrator in the noise source, *e*(*t*). This is useful in cases where the disturbance is not stationary.

Model Structure	Equation	Description
ARMAX	$A(q)y(t) = \sum_{i=1}^{nu} B_i(q)u_i\left(t - nk_i\right) + C(q)e(t)$	Extends the ARX structure by providing more flexibility for modeling noise using the *C* parameters (a moving average of white noise). Use ARMAX when the dominating disturbances enter at the input. Such disturbances are called *load disturbances*.
ARIMAX	$Ay = Bu + C\frac{1}{1-q^{-1}}e$	Extends the ARMAX structure by including an integrator in the noise source, *e*(*t*). This is useful in cases where the disturbance is not stationary.
Box-Jenkins (BJ)	$y(t) = \sum_{i=1}^{nu} \frac{B_i(q)}{F_i(q)}u_i\left(t - nk_i\right) + \frac{C(q)}{D(q)}e(t)$	Provides completely independent parameterization for the dynamics and the noise using rational polynomial functions. Use BJ models when the noise does not enter at the input, but is primary a measurement disturbance, This structure provides additional flexibility for modeling noise.
Output-Error (OE)	$y(t) = \sum_{i=1}^{nu} \frac{B_i(q)}{F_i(q)}u_i\left(t - nk_i\right) + e(t)$	Use when you want to parameterize dynamics, but do not want to estimate a noise model. **Note** In this case, the noise models is $H = 1$ in the general equation and the white noise source *e(t)* affects only the output.

The polynomial models can contain one or more outputs and zero or more inputs.

The System Identification app supports direct estimation of ARX, ARMAX, OE and BJ models. You can add a noise integrator to the ARX, ARMAX and BJ forms. However, you

can use `polyest` to estimate all five polynomial or any subset of polynomials in the general equation. For more information about working with pem, see "Using polyest to Estimate Polynomial Models" on page 6-24.

Continuous-Time Representation of Polynomial Models

In continuous time, the general frequency-domain equation is written in terms of the Laplace transform variable s, which corresponds to a differentiation operation:

$$A(s)Y(s) = \frac{B(s)}{F(s)}U(s) + \frac{C(s)}{D(s)}E(s)$$

In the continuous-time case, the underlying time-domain model is a differential equation and the model order integers represent the number of estimated numerator and denominator coefficients. For example, n_a=3 and n_b=2 correspond to the following model:

$$A(s) = s^4 + a_1 s^3 + a_2 s^2 + a_3$$
$$B(s) = b_1 s + b_2$$

You can only estimate continuous-time polynomial models directly using continuous-time frequency-domain data. In this case, you must set the `Ts` data property to 0 to indicate that you have continuous-time frequency-domain data, and use the `oe` command to estimate an Output-Error polynomial model. Continuous-time models of other structures such as ARMAX or BJ cannot be estimated. You can obtain those forms only by direct construction (using `idpoly`), conversion from other model types, or by converting a discrete-time model into continuous-time (`d2c`). Note that the OE form represents a transfer function expressed as a ratio of numerator (B) and denominator (F) polynomials. For such forms consider using the transfer function models, represented by `idtf` models. You can estimate transfer function models using both time and frequency domain data. In addition to the numerator and denominator polynomials, you can also estimate transport delays. See `idtf` and `tfest` for more information.

Multi-Output Polynomial Models

For a MIMO polynomial model with ny outputs and nu inputs, the relation between inputs and outputs for the l^{th} output can be written as:

$$\sum_{j=1}^{ny} A_{lj}(q)y_j(t) = \sum_{i=1}^{nu} \frac{B_{li}(q)}{F_{li}(q)}u_i\left(t - nk_i\right) + \frac{C_l(q)}{D_l(q)}e_l(t)$$

The A polynomial array (A_{ij}; i=1:ny, j=1:ny) are stored in the `A` property of the `idpoly` object. The diagonal polynomials (A_{ii}; i=1:ny) are monic, that is, the leading coefficients are one. The off-diagonal polynomials (A_{ij}; $i \neq j$) contain a delay of at least one sample, that is, they start with zero. For more details on the orders of multi-output models, see "Polynomial Sizes and Orders of Multi-Output Polynomial Models" on page 6-27.

You can create multi-output polynomial models by using the `idpoly` command or estimate them using `ar`, `arx`, `bj`, `oe`, `armax`, and `polyest`. In the app, you can estimate such models by choosing a multi-output data set and setting the orders appropriately in the **Polynomial Models** dialog box.

See Also

`ar` | `armax` | `arx` | `bj` | `idpoly` | `oe` | `polyest`

Related Examples

- "Estimate Polynomial Models in the App" on page 6-18
- "Estimate Polynomial Models at the Command Line" on page 6-23

More About

- "Data Supported by Polynomial Models" on page 6-8

Data Supported by Polynomial Models

In this section...
“Types of Supported Data” on page 6-8 “Designating Data for Estimating Continuous-Time Models” on page 6-8 “Designating Data for Estimating Discrete-Time Models” on page 6-9

Types of Supported Data

You can estimate linear, black-box polynomial models from data with the following characteristics:

- Time- or frequency-domain data (`iddata` or `idfrd` data objects).

  Note For frequency-domain data, you can only estimate ARX and OE models.

  To estimate polynomial models for time-series data, see “Time Series Analysis”.
- Real data or complex data in any domain.
- Single-output and multiple-output.

You must import your data into the MATLAB workspace, as described in “Data Preparation”.

Designating Data for Estimating Continuous-Time Models

To get a linear, continuous-time model of arbitrary structure for time-domain data, you can estimate a discrete-time model, and then use `d2c` to transform it to a continuous-time model.

For continuous-time frequency-domain data, you can estimate directly only Output-Error (OE) continuous-time models. Other structures include noise models, which is not supported for frequency-domain data.

Tip To denote continuous-time frequency-domain data, set the data sample time to 0. You can set the sample time when you import data into the app or set the `Ts` property of the data object at the command line.

Designating Data for Estimating Discrete-Time Models

You can estimate arbitrary-order, linear state-space models for both time- or frequency-domain data.

Set the data property `Ts` to:

- `0`, for frequency response data that is measured directly from an experiment.
- Equal to the `Ts` of the original data, for frequency response data obtained by transforming time-domain `iddata` (using `spa` and `etfe`).

Tip You can set the sample time when you import data into the app or set the `Ts` property of the data object at the command line.

See Also

Related Examples

- “Estimate Polynomial Models in the App” on page 6-18
- “Estimate Polynomial Models at the Command Line” on page 6-23

More About

- “What Are Polynomial Models?” on page 6-2

Preliminary Step – Estimating Model Orders and Input Delays

In this section...
"Why Estimate Model Orders and Delays?" on page 6-10
"Estimating Orders and Delays in the App" on page 6-10
"Estimating Model Orders at the Command Line" on page 6-13
"Estimating Delays at the Command Line" on page 6-14
"Selecting Model Orders from the Best ARX Structure" on page 6-15

Why Estimate Model Orders and Delays?

To estimate polynomial models, you must provide input delays and model orders. If you already have insight into the physics of your system, you can specify the number of poles and zeros.

In most cases, you do not know the model orders in advance. To get initial model orders and delays for your system, you can estimate several ARX models with a range of orders and delays and compare the performance of these models. You choose the model orders that correspond to the best model performance and use these orders as an initial guess for further modeling.

Because this estimation procedure uses the ARX model structure, which includes the A and B polynomials, you only get estimates for the na, nb, and nk parameters. However, you can use these results as initial guesses for the corresponding polynomial orders and input delays in other model structures, such as ARMAX, OE, and BJ.

If the estimated nk is too small, the leading nb coefficients are much smaller than their standard deviations. Conversely, if the estimated nk is too large, there is a significant correlation between the residuals and the input for lags that correspond to the missing B terms. For information about residual analysis plots, see topics on the "Residual Analysis" page.

Estimating Orders and Delays in the App

The following procedure assumes that you have already imported your data into the app and performed any necessary preprocessing operations. For more information, see "Represent Data".

To estimate model orders and input delays in the System Identification app:

1. In the System Identification app, select **Estimate** > **Polynomial Models** to open the Polynomials Models dialog box.

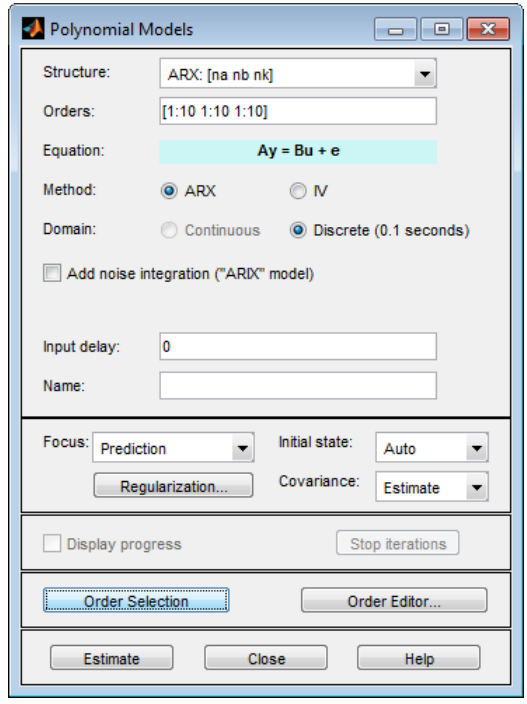
   The ARX model is already selected by default in the **Structure** list.

   Note For time-series models, select the AR model structure.

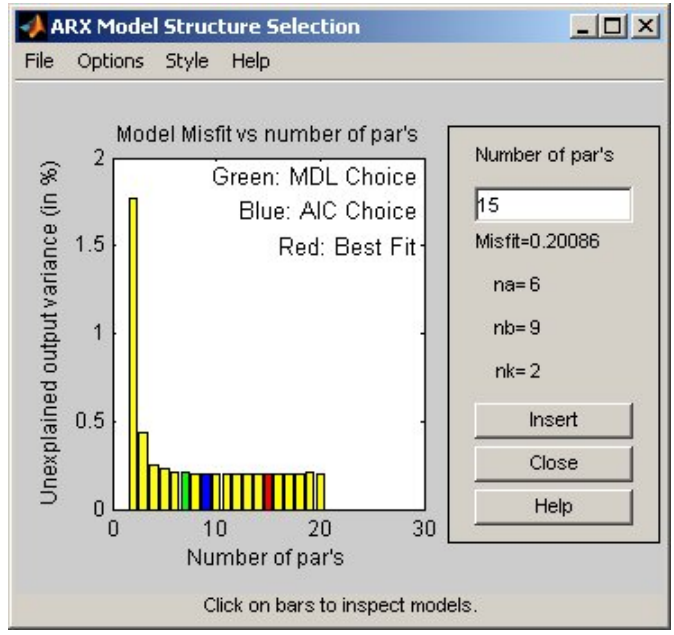
2. Edit the **Orders** field to specify a range of poles, zeros, and delays. For example, enter the following values for *na*, *nb*, and *nk*:

   ```
   [1:10 1:10 1:10]
   ```

   Tip As a shortcut for entering `1:10` for each required model order, click **Order Selection**.



3 Click **Estimate** to open the ARX Model Structure Selection window, which displays the model performance for each combination of model parameters. The following figure shows an example plot.



4 Select a rectangle that represents the optimum parameter combination and click **Insert** to estimates a model with these parameters. For information about using this plot, see "Selecting Model Orders from the Best ARX Structure" on page 6-15.

This action adds a new model to the Model Board in the System Identification app. The default name of the parametric model contains the model type and the number of poles, zeros, and delays. For example, `arx692` is an ARX model with n_a=6, n_b=9, and a delay of two samples.

5 Click **Close** to close the ARX Model Structure Selection window.

Note You cannot estimate model orders when using multi-output data.

After estimating model orders and delays, use these values as initial guesses for estimating other model structures, as described in "Estimate Polynomial Models in the App" on page 6-18.

Estimating Model Orders at the Command Line

You can estimate model orders using the `struc`, `arxstruc`, and `selstruc` commands in combination.

If you are working with a multiple-output system, you must use the `struc`, `arxstruc`, and `selstruc` commands one output at a time. You must subreference the correct output channel in your estimation and validation data sets.

For each estimation, you use two independent data sets—an estimation data set and a validation data set. These independent data set can be from different experiments, or data subsets from a single experiment. For more information about subreferencing data, see "Select Data Channels, I/O Data and Experiments in iddata Objects" on page 2-55 and "Select I/O Channels and Data in idfrd Objects" on page 2-87.

For an example of estimating model orders for a multiple-input system, see "Estimating Delays in the Multiple-Input System" in *System Identification Toolbox Getting Started Guide*.

struc

The `struc` command creates a matrix of possible model-order combinations for a specified range of n_a, n_b, and n_k values.

For example, the following command defines the range of model orders and delays `na=2:5`, `nb=1:5`, and `nk=1:5`:

```
NN = struc(2:5,1:5,1:5))
```

arxstruc

The `arxstruc` command takes the output from `struc`, estimates an ARX model for each model order, and compares the model output to the measured output. `arxstruc` returns the *loss* for each model, which is the normalized sum of squared prediction errors.

For example, the following command uses the range of specified orders `NN` to compute the loss function for single-input/single-output estimation data `data_e` and validation data `data_v`:

```
V = arxstruc(data_e,data_v,NN);
```

Each row in `NN` corresponds to one set of orders:

```
[na nb nk]
```

selstruc

The `selstruc` command takes the output from `arxstruc` and opens the ARX Model Structure Selection window to guide your choice of the model order with the best performance.

For example, to open the ARX Model Structure Selection window and interactively choose the optimum parameter combination, use the following command:

```
selstruc(V);
```

For more information about working with the ARX Model Structure Selection window, see “Selecting Model Orders from the Best ARX Structure” on page 6-15.

To find the structure that minimizes Akaike's Information Criterion, use the following command:

```
nn = selstruc(V,'AIC');
```

where `nn` contains the corresponding `na`, `nb`, and `nk` orders.

Similarly, to find the structure that minimizes the Rissanen's Minimum Description Length (MDL), use the following command:

```
nn = selstruc(V,'MDL');
```

To select the structure with the smallest loss function, use the following command:

```
nn = selstruc(V,0);
```

After estimating model orders and delays, use these values as initial guesses for estimating other model structures, as described in “Using polyest to Estimate Polynomial Models” on page 6-24.

Estimating Delays at the Command Line

The `delayest` command estimates the time delay in a dynamic system by estimating a low-order, discrete-time ARX model and treating the delay as an unknown parameter.

By default, `delayest` assumes that n_a=n_b=2 and that there is a good signal-to-noise ratio, and uses this information to estimate n_k.

To estimate the delay for a data set `data`, type the following at the prompt:

```
delayest(data);
```

If your data has a single input, MATLAB computes a scalar value for the input delay—equal to the number of data samples. If your data has multiple inputs, MATLAB returns a vector, where each value is the delay for the corresponding input signal.

To compute the actual delay time, you must multiply the input delay by the sample time of the data.

You can also use the ARX Model Structure Selection window to estimate input delays and model order together, as described in "Estimating Model Orders at the Command Line" on page 6-13.

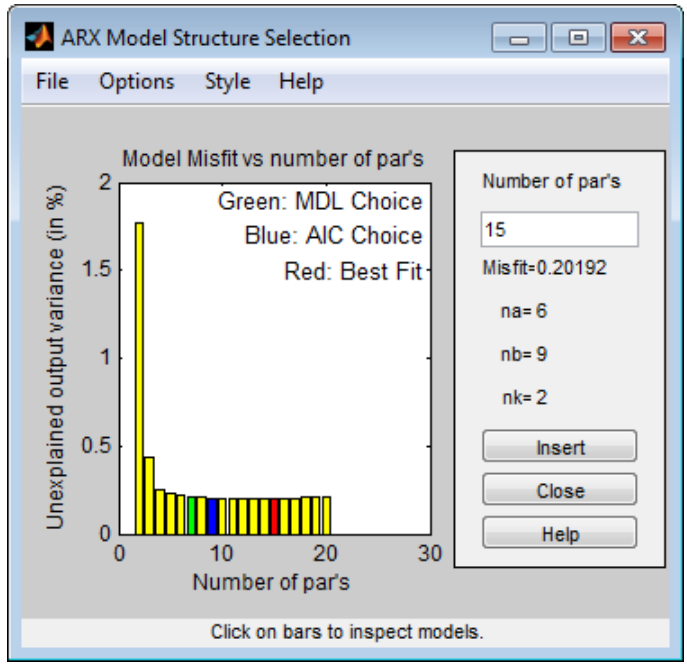
Selecting Model Orders from the Best ARX Structure

You generate the ARX Model Structure Selection window for your data to select the best-fit model.

For a procedure on generating this plot in the System Identification app, see "Estimating Orders and Delays in the App" on page 6-10. To open this plot at the command line, see "Estimating Model Orders at the Command Line" on page 6-13.

The following figure shows a sample plot in the ARX Model Structure Selection window.

You use this plot to select the best-fit model.



- The horizontal axis is the total number of parameters — $n_a + n_b$.
- The vertical axis, called **Unexplained output variance (in %)**, is the portion of the output not explained by the model—the ARX model prediction error for the number of parameters shown on the horizontal axis.

  The *prediction error* is the sum of the squares of the differences between the validation data output and the model one-step-ahead predicted output.
- n_k is the delay.

Three rectangles are highlighted on the plot in green, blue, and red. Each color indicates a type of best-fit criterion, as follows:

- Red — Best fit minimizes the sum of the squares of the difference between the validation data output and the model output. This rectangle indicates the overall best fit.
- Green — Best fit minimizes Rissanen MDL criterion.
- Blue — Best fit minimizes Akaike AIC criterion.

In the ARX Model Structure Selection window, click any bar to view the orders that give the best fit. The area on the right is dynamically updated to show the orders and delays that give the best fit.

For more information about the AIC criterion, see “Loss Function and Model Quality Metrics” on page 1-64.

See Also

Related Examples

- “Estimate Polynomial Models in the App” on page 6-18
- “Estimate Polynomial Models at the Command Line” on page 6-23
- “Model Structure Selection: Determining Model Order and Input Delay” on page 4-49

More About

- “What Are Polynomial Models?” on page 6-2

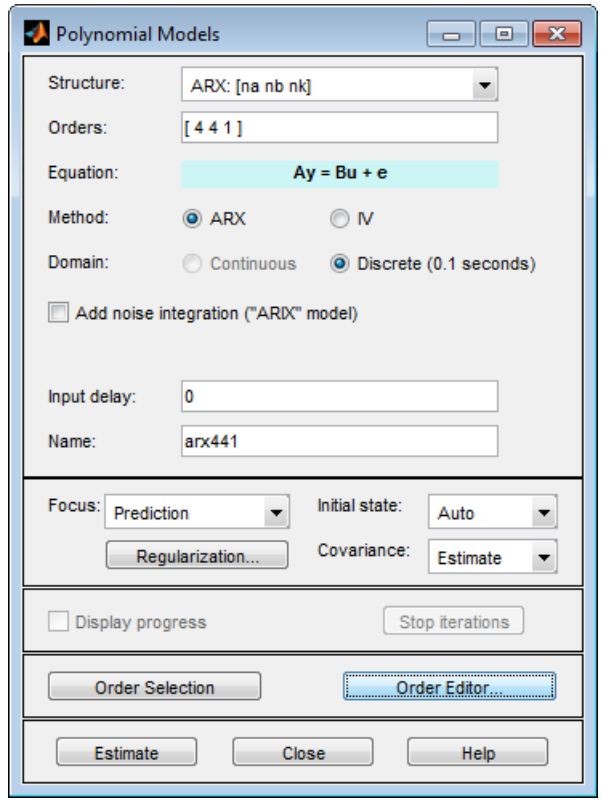
Estimate Polynomial Models in the App

Prerequisites

Before you can perform this task, you must have:

- Imported data into the System Identification app. See “Import Time-Domain Data into the App” on page 2-16. For supported data formats, see “Data Supported by Polynomial Models” on page 6-8.
- Performed any required data preprocessing operations. To improve the accuracy of your model, you should detrend your data. Removing offsets and trends is especially important for Output-Error (OE) models and has less impact on the accuracy of models that include a flexible noise model structure, such as ARMAX and Box-Jenkins. See “Ways to Prepare Data for System Identification” on page 2-6.
- Select a model structure, model orders, and delays. For a list of available structures, see “What Are Polynomial Models?” on page 6-2 For more information about how to estimate model orders and delays, see “Estimating Orders and Delays in the App” on page 6-10. For multiple-output models, you must specify order matrices in the MATLAB workspace, as described in “Polynomial Sizes and Orders of Multi-Output Polynomial Models” on page 6-27.

1 In the System Identification app, select **Estimate** > **Polynomial Models** to open the Polynomial Models dialog box.



For more information on the options in the dialog box, click **Help**.

2 In the **Structure** list, select the polynomial model structure you want to estimate from the following options:

- `ARX:[na nb nk]`
- `ARMAX:[na nb nc nk]`
- `OE:[nb nf nk]`
- `BJ:[nb nc nd nf nk]`

This action updates the options in the Polynomial Models dialog box to correspond with this model structure. For information about each model structure, see “What Are Polynomial Models?” on page 6-2.

Note For time-series data, only AR and ARMA models are available. For more information about estimating time-series models, see "Time Series Analysis".

3 In the **Orders** field, specify the model orders and delays, as follows:

- For single-output polynomial models, enter the model orders and delays according to the sequence displayed in the **Structure** field. For multiple-input models, specify `nb` and `nk` as row vectors with as many elements as there are inputs. If you are estimating BJ and OE models, you must also specify `nf` as a vector.

  For example, for a three-input system, `nb` can be `[1 2 4]`, where each element corresponds to an input.
- For multiple-output models, enter the model orders as described in "Polynomial Sizes and Orders of Multi-Output Polynomial Models" on page 6-27.

Tip To enter model orders and delays using the Order Editor dialog box, click **Order Editor**.

4 (ARX models only) Select the estimation **Method** as **ARX** or **IV** (instrumental variable method). For information about the algorithms, see "Polynomial Model Estimation Algorithms" on page 6-34.

5 (ARX, ARMAX, and BJ models only) Check the **Add noise integration** check box to add an integrator to the noise source, *e*.

6 Specify the delay using the **Input delay** edit box. The value must be a vector of length equal to the number of input channels in the data. For discrete-time estimations (any estimation using data with nonzero sample-time), the delay must be expressed in the number of lags. These delays are separate from the "in-model" delays specified by the `nk` order in the **Orders** edit box.

7 In the **Name** field, edit the name of the model or keep the default.

8 In the **Focus** list, select how to weigh the relative importance of the fit at different frequencies. For more information about each option, see "Assigning Estimation Weightings" on page 6-21.

9 In the **Initial state** list, specify how you want the algorithm to treat initial conditions. For more information about the available options, see "Specifying Initial Conditions for Iterative Estimation Algorithms" on page 5-49.

Tip If you get an inaccurate fit, try setting a specific method for handling initial states rather than choosing it automatically.

10 In the **Covariance** list, select `Estimate` if you want the algorithm to compute parameter uncertainties. Effects of such uncertainties are displayed on plots as model confidence regions.

   To omit estimating uncertainty, select `None`. Skipping uncertainty computation for large, multiple-output models might reduce computation time.

11 Click **Regularization** to obtain regularized estimates of model parameters. Specify the regularization constants in the Regularization Options dialog box. To learn more, see "Regularized Estimates of Model Parameters" on page 1-48.

12 (ARMAX, OE, and BJ models only) To view the estimation progress in the MATLAB Command Window, select the **Display progress** check box. This launches a progress viewer window in which estimation progress is reported.

13 Click **Estimate** to add this model to the Model Board in the System Identification app.

14 (Prediction-error method only) To stop the search and save the results after the current iteration has been completed, click **Stop Iterations**. To continue iterations from the current model, click the **Continue iter** button to assign current parameter values as initial guesses for the next search.

Assigning Estimation Weightings

You can specify how the estimation algorithm weighs the fit at various frequencies. In the app, set **Focus** to one of the following options:

- `Prediction` — Uses the inverse of the noise model H to weigh the relative importance of how closely to fit the data in various frequency ranges. Corresponds to minimizing one-step-ahead prediction, which typically favors the fit over a short time interval. Optimized for output prediction applications.
- `Simulation` — Uses the input spectrum to weigh the relative importance of the fit in a specific frequency range. Does not use the noise model to weigh the relative importance of how closely to fit the data in various frequency ranges. Optimized for output simulation applications.
- `Stability` — Estimates the best stable model. For more information about model stability, see "Unstable Models" on page 17-120.
- `Filter` — Specify a custom filter to open the Estimation Focus dialog box, where you can enter a filter, as described in "Simple Passband Filter" on page 2-134 or "Defining

a Custom Filter" on page 2-135. This prefiltering applies only for estimating the dynamics from input to output. The disturbance model is determined from the unfiltered estimation data.

Next Steps

- Validate the model by selecting the appropriate check box in the **Model Views** area of the System Identification app. For more information about validating models, see "Validating Models After Estimation" on page 17-3.
- Export the model to the MATLAB workspace for further analysis by dragging it to the **To Workspace** rectangle in the System Identification app.

Tip For ARX and OE models, you can use the exported model for initializing a nonlinear estimation at the command line. This initialization may improve the fit of the model. See "Initialize Nonlinear ARX Estimation Using Linear Model" on page 11-25, and "Initialize Hammerstein-Wiener Estimation Using Linear Model" on page 12-8.

See Also

Related Examples

- "Preliminary Step – Estimating Model Orders and Input Delays" on page 6-10
- "Estimate Polynomial Models at the Command Line" on page 6-23

More About

- "What Are Polynomial Models?" on page 6-2
- "Data Supported by Polynomial Models" on page 6-8

Estimate Polynomial Models at the Command Line

In this section...
"Using arx and iv4 to Estimate ARX Models" on page 6-23 "Using polyest to Estimate Polynomial Models" on page 6-24

Prerequisites

Before you can perform this task, you must have

- Input-output data as an `iddata` object or frequency response data as an `frd` or `idfrd` object. See "Representing Time- and Frequency-Domain Data Using iddata Objects" on page 2-50. For supported data formats, see "Data Supported by Polynomial Models" on page 6-8.
- Performed any required data preprocessing operations. To improve the accuracy of results when using time domain data, you can detrend the data or specify the input/output offset levels as estimation options. Removing offsets and trends is especially important for Output-Error (OE) models and has less impact on the accuracy of models that include a flexible noise model structure, such as ARMAX and Box-Jenkins. See "Ways to Prepare Data for System Identification" on page 2-6.
- Select a model structure, model orders, and delays. For a list of available structures, see "What Are Polynomial Models?" on page 6-2 For more information about how to estimate model orders and delays, see "Estimating Model Orders at the Command Line" on page 6-13 and "Estimating Delays at the Command Line" on page 6-14. For multiple-output models, you must specify order matrices in the MATLAB workspace, as described in "Polynomial Sizes and Orders of Multi-Output Polynomial Models" on page 6-27.

Using arx and iv4 to Estimate ARX Models

You can estimate single-output and multiple-output ARX models using the `arx` and `iv4` commands. For information about the algorithms, see "Polynomial Model Estimation Algorithms" on page 6-34.

You can use the following general syntax to both configure and estimate ARX models:

```
% Using ARX method
m = arx(data,[na nb nk],opt);
```

```
% Using IV method
m = iv4(data,[na nb nk],opt);
```

`data` is the estimation data and `[na nb nk]` specifies the model orders, as discussed in "What Are Polynomial Models?" on page 6-2.

The third input argument `opt` contains the options for configuring the estimation of the ARX model, such as handling of initial conditions and input offsets. You can create and configure the option set `opt` using the `arxOptions` and `iv4Options` commands. The three input arguments can also be followed by name and value pairs to specify optional model structure attributes such as `InputDelay`, `IODelay`, and `IntegrateNoise`.

To get discrete-time models, use the time-domain data (`iddata` object).

Note Continuous-time polynomials of ARX structure are not supported.

For more information about validating you model, see "Validating Models After Estimation" on page 17-3.

You can use `pem` or `polyest` to refine parameter estimates of an existing polynomial model, as described in "Refine Linear Parametric Models" on page 4-7.

For detailed information about these commands, see the corresponding reference page.

Tip You can use the estimated ARX model for initializing a nonlinear estimation at the command line, which improves the fit of the model. See "Initialize Nonlinear ARX Estimation Using Linear Model" on page 11-25.

Using polyest to Estimate Polynomial Models

You can estimate any polynomial model using the iterative prediction-error estimation method `polyest`. For Gaussian disturbances of unknown variance, this method gives the maximum likelihood estimate. The resulting models are stored as `idpoly` model objects.

Use the following general syntax to both configure and estimate polynomial models:

```
m = polyest(data,[na nb nc nd nf nk],opt,Name,Value);
```

where `data` is the estimation data. `na`, `nb`, `nc`, `nd`, `nf` are integers that specify the model orders, and `nk` specifies the input delays for each input.For more information about model orders, see "What Are Polynomial Models?" on page 6-2.

Tip You do not need to construct the model object using `idpoly` before estimation.

If you want to estimate the coefficients of all five polynomials, *A*, *B*, *C*, *D*, and *F*, you must specify an integer order for each polynomial. However, if you want to specify an ARMAX model for example, which includes only the *A*, *B*, and *C* polynomials, you must set `nd` and `nf` to zero matrices of the appropriate size. For some simpler configurations, there are dedicated estimation commands such as `arx`, `armax`, `bj`, and `oe`, which deliver the required model by using just the required orders. For example, `oe(data,[nb nf nk],opt)` estimates an output-error structure polynomial model.

Note To get faster estimation of ARX models, use `arx` or `iv4` instead of `polyest`.

In addition to the polynomial models listed in "What Are Polynomial Models?" on page 6-2, you can use `polyest` to model the ARARX structure—called the *generalized least-squares model*—by setting `nc=nf=0`. You can also model the ARARMAX structure—called the *extended matrix model*—by setting `nf=0`.

The third input argument, `opt`, contains the options for configuring the estimation of the polynomial model, such as handling of initial conditions, input offsets and search algorithm. You can create and configure the option set `opt` using the `polyestOptions` command. The three input arguments can also be followed by name and value pairs to specify optional model structure attributes such as `InputDelay`, `IODelay`, and `IntegrateNoise`.

For ARMAX, Box-Jenkins, and Output-Error models—which can only be estimated using the iterative prediction-error method—use the `armax`, `bj`, and `oe` estimation commands, respectively. These commands are versions of `polyest` with simplified syntax for these specific model structures, as follows:

```
m = armax(Data,[na nb nc nk]);
m = oe(Data,[nb nf nk]);
m = bj(Data,[nb nc nd nf nk]);
```

Similar to `polyest`, you can specify as input arguments the option set configured using commands `armaxOptions`, `oeOptions`, and `bjOptions` for the estimators `armax`, `oe`, and `bj` respectively. You can also use name and value pairs to configure additional model structure attributes.

Tip If your data is sampled fast, it might help to apply a lowpass filter to the data before estimating the model, or specify a frequency range for the `WeightingFilter` property during estimation. For example, to model only data in the frequency range 0-10 rad/s, use the `WeightingFilter` property, as follows:

```
opt = oeOptions('WeightingFilter',[0 10]);
m = oe(Data, [nb nf nk], opt);
```

For more information about validating your model, see “Validating Models After Estimation” on page 17-3.

You can use `pem` or `polyest` to refine parameter estimates of an existing polynomial model (of any configuration), as described in “Refine Linear Parametric Models” on page 4-7.

For more information, see `polyest`, `pem` and `idpoly`.

See Also

Related Examples

- “Estimate Models Using armax” on page 6-35
- “Preliminary Step – Estimating Model Orders and Input Delays” on page 6-10
- “Estimate Polynomial Models in the App” on page 6-18

More About

- “What Are Polynomial Models?” on page 6-2
- “Data Supported by Polynomial Models” on page 6-8
- “Loss Function and Model Quality Metrics” on page 1-64

Polynomial Sizes and Orders of Multi-Output Polynomial Models

For a model with *Ny* (`Ny > 1`) outputs and *Nu* inputs, the polynomials *A*, *B*, *C*, *D*, and *F* are specified as cell arrays of row vectors. Each entry in the cell array contains the coefficients of a particular polynomial that relates input, output, and noise values. *Orders* are matrices of integers used as input arguments to the estimation commands.

Polynomial	Dimension	Relation Described	Orders
`A`	N_y-by-N_y array of row vectors	`A{i,j}` contains coefficients of relation between output y_i and output y_j	`na`: N_y-by-N_y matrix such that each entry contains the degree of the corresponding *A* polynomial.

Polynomial	Dimension	Relation Described	Orders
`B`	N_y-by-N_u array of row vectors	`B{i,j}` contain coefficients of relations between output y_i and input u_j	`nk`: N_y-by-N_u matrix such that each entry contains the number of leading fixed zeros of the corresponding B polynomial (input delay). `nb`: N_y-by-N_u matrix such `nb(i,j) = length(B{i,j})-nk(i,j).`

Polynomial	Dimension	Relation Described	Orders
`C,D`	N_y-by-1 array of row vectors	`C{i}` and `D{i}` contain coefficients of relations between output y_i and noise e_i	`nc` and `nd` are N_y-by-1 matrices such that each entry contains the degree of the corresponding C and D polynomial, respectively.
`F`	N_y-by-N_u array of row vectors	`F{i,j}` contains coefficients of relations between output y_i and input u_j	`nf`: N_y-by-N_u matrix such that each entry contains the degree of the corresponding F polynomial.

For more information, see `idpoly`.

For example, consider the ARMAX set of equations for a 2 output, 1 input model:

$$y_1(t) + 0.5\, y_1(t\text{-}1) + 0.9\, y_2(t\text{-}1) + 0.1\, y_2(t\text{-}2) = u(t) + 5\, u(t\text{-}1) + 2\, u(t\text{-}2) + e_1(t) + 0.01\, e_1(t\text{-}1)$$

$$y_2(t) + 0.05\, y_2(t\text{-}1) + 0.3\, y_2(t\text{-}2) = 10\, u(t\text{-}2) + e_2(t) + 0.1\, e_2(t\text{-}1) + 0.02\, e_2(t\text{-}2)$$

y_1 and y_2 represent the two outputs and u represents the input variable. e_1 and e_2 represent the white noise disturbances on the outputs, y_1 and y_2, respectively. To

represent these equations as an ARMAX form polynomial using `idpoly`, configure the *A*, *B*, and *C* polynomials as follows:

```
A = cell(2,2);
A{1,1} = [1 0.5];
A{1,2} = [0 0.9 0.1];
A{2,1} = [0];
A{2,2} = [1 0.05 0.3];

B = cell(2,1);
B{1,1} = [1 5 2];
B{2,1}  = [0 0 10];

C = cell(2,1);
C{1} = [1 0.01];
C{2} = [1 0.1 0.02];

model = idpoly(A,B,C)
```

```
model =
Discrete-time ARMAX model:
  Model for output number 1: A(z)y_1(t) = - A_i(z)y_i(t) + B(z)u(t) + C(z)e_1(t)
    A(z) = 1 + 0.5 z^-1

    A_2(z) = 0.9 z^-1 + 0.1 z^-2

    B(z) = 1 + 5 z^-1 + 2 z^-2

    C(z) = 1 + 0.01 z^-1

  Model for output number 2: A(z)y_2(t) = B(z)u(t) + C(z)e_2(t)
    A(z) = 1 + 0.05 z^-1 + 0.3 z^-2

    B(z) = 10 z^-2

    C(z) = 1 + 0.1 z^-1 + 0.02 z^-2

Sample time: unspecified

Parameterization:
   Polynomial orders:   na=[1 2;0 2]   nb=[3;1]   nc=[1;2]
   nk=[0;2]
   Number of free coefficients: 12
   Use "polydata", "getpvec", "getcov" for parameters and their uncertainties.
```

```
Status:
Created by direct construction or transformation. Not estimated.
```

`model` is a discrete-time ARMAX model with unspecified sample-time. When estimating such models, you need to specify the orders of these polynomials as input arguments.

In the System Identification app, you can enter the matrices directly in the **Orders** field.

At the command line, define variables that store the model order matrices and specify these variables in the model-estimation command.

Tip To simplify entering large matrices orders in the System Identification app, define the variable `NN=[NA NB NK]` at the command line. You can specify this variable in the **Orders** field.

See Also

`ar` | `armax` | `arx` | `bj` | `idpoly` | `oe` | `polyest`

Related Examples

- “Estimate Polynomial Models in the App” on page 6-18
- “Estimate Polynomial Models at the Command Line” on page 6-23

More About

- “What Are Polynomial Models?” on page 6-2

Specifying Initial States for Iterative Estimation Algorithms

When you use the `pem` or `polyest` to estimate ARMAX, Box-Jenkins (BJ), Output-Error (OE), you must specify how the algorithm treats initial conditions.

This information supports the estimation procedures "Estimate Polynomial Models in the App" on page 6-18 and "Using polyest to Estimate Polynomial Models" on page 6-24.

In the System Identification app, for ARMAX, OE, and BJ models, set **Initial state** to one of the following options:

- `Auto` — Automatically chooses `Zero`, `Estimate`, or `Backcast` based on the estimation data. If initial states have negligible effect on the prediction errors, the initial states are set to zero to optimize algorithm performance.
- `Zero` — Sets all initial states to zero.
- `Estimate` — Treats the initial states as an unknown vector of parameters and estimates these states from the data.
- `Backcast` — Estimates initial states using a smoothing filter.

At the command line, specify the initial conditions as an estimation option. Use `polyestOptions` to configure options for the `polyest` command, `armaxOptions` for the `armax` command etc. Set the `InitialCondition` option to the desired value in the option set. For example, use this command to estimate an ARMAX model and set the initial states to zero:

```
opt = armaxOptions('InitialCondition','zero');
m = armax(data,[2 2 2 3],opt);
```

For a complete list of values for the `InitialCondition` estimation option, see the `armaxOptions` reference page.

See Also

`armaxOptions` | `arxOptions` | `bjOptions` | `iv4Options` | `oeOptions` | `polyestOptions`

Related Examples

- "Estimate Polynomial Models in the App" on page 6-18

- “Estimate Polynomial Models at the Command Line” on page 6-23

Polynomial Model Estimation Algorithms

For linear ARX and AR models, you can choose between the ARX and IV algorithms. *ARX* implements the least-squares estimation method that uses QR-factorization for overdetermined linear equations. *IV* is the *instrument variable method*. For more information about IV, see the section on variance-optimal instruments in *System Identification: Theory for the User*, Second Edition, by Lennart Ljung, Prentice Hall PTR, 1999.

The ARX and IV algorithms treat noise differently. ARX assumes white noise. However, the instrumental variable algorithm, IV, is not sensitive to noise color. Thus, use IV when the noise in your system is not completely white and it is incorrect to assume white noise. If the models you obtained using ARX are inaccurate, try using IV.

Note AR models apply to time-series data, which has no input. For more information, see "Time Series Analysis". For more information about working with AR and ARX models, see "Input-Output Polynomial Models".

See Also

`ar` | `arx` | `iv4`

More About

- "What Are Polynomial Models?" on page 6-2

Estimate Models Using armax

This example shows how to estimate a linear, polynomial model with an ARMAX structure for a three-input and single-output (MISO) system using the iterative estimation method `armax`. For a summary of all available estimation commands in the toolbox, see “Model Estimation Commands” on page 1-44.

Load a sample data set `z8` with three inputs and one output, measured at `1` -second intervals and containing 500 data samples.

```
load iddata8
```

Use `armax` to both construct the `idpoly` model object, and estimate the parameters:

$$A(q)y(t) = \sum_{i=1}^{nu} B_i(q)u_i(t - nk_i) + C(q)e(t)$$

Typically, you try different model orders and compare results, ultimately choosing the simplest model that best describes the system dynamics. The following command specifies the estimation data set, `z8` , and the orders of the A , B , and C polynomials as `na` , `nb` , and `nc`, respectively. `nk` of `[0 0 0]` specifies that there is no input delay for all three input channels.

```
opt = armaxOptions;
opt.Focus = 'simulation';
opt.SearchOption.MaxIter = 50;
opt.SearchOption.Tolerance = 1e-5;
na = 4;
nb = [3 2 3];
nc = 4;
nk = [0 0 0];
m_armax = armax(z8, [na nb nc nk], opt);
```

`Focus`, `Tolerance`, and `MaxIter` are estimation options that configure the estimation objective function and the attributes of the search algorithm. The `Focus` option specifies whether the model is optimized for simulation or prediction applications. The `Tolerance` and `MaxIter` search options specify when to stop estimation. For more information about these properties, see the `armaxOptions` reference page.

`armax` is a version of `polyest` with simplified syntax for the ARMAX model structure. The `armax` method both constructs the `idpoly` model object and estimates its parameters.

View information about the resulting model object.

```
m_armax
```

```
m_armax =
Discrete-time ARMAX model:  A(z)y(t) = B(z)u(t) + C(z)e(t)
  A(z) = 1 - 1.284 z^-1 + 0.3048 z^-2 + 0.2648 z^-3 - 0.05708 z^-4

  B1(z) = -0.07547 + 1.087 z^-1 + 0.7166 z^-2

  B2(z) = 1.019 + 0.1142 z^-1

  B3(z) = -0.06739 + 0.06828 z^-1 + 0.5509 z^-2

  C(z) = 1 - 0.06096 z^-1 - 0.1296 z^-2 + 0.02489 z^-3 - 0.04699 z^-4

Sample time: 1 seconds

Parameterization:
   Polynomial orders:   na=4   nb=[3 2 3]   nc=4   nk=[0 0 0]
   Number of free coefficients: 16
   Use "polydata", "getpvec", "getcov" for parameters and their uncertainties.

Status:
Estimated using ARMAX on time domain data "z8".
Fit to estimation data: 80.86% (simulation focus)
FPE: 2.888, MSE: 0.9868
```

`m_armax` is an `idpoly` model object. The coefficients represent estimated parameters of this polynomial model. You can use `present(m_armax)` to show additional information about the model, including parameter uncertainties.

View all property values for this model.

```
get(m_armax)
```

```
                A: [1 -1.2836 0.3048 0.2648 -0.0571]
                B: {[-0.0755 1.0870 0.7166]  [1.0188 0.1142]  [1x3 double]}
                C: [1 -0.0610 -0.1296 0.0249 -0.0470]
                D: 1
```

```
                F: {[1]  [1]  [1]}
    IntegrateNoise: 0
          Variable: 'z^-1'
           IODelay: [0 0 0]
         Structure: [1x1 pmodel.polynomial]
     NoiseVariance: 2.7984
            Report: [1x1 idresults.polyest]
        InputDelay: [3x1 double]
       OutputDelay: 0
                Ts: 1
          TimeUnit: 'seconds'
         InputName: {3x1 cell}
         InputUnit: {3x1 cell}
        InputGroup: [1x1 struct]
        OutputName: {'y1'}
        OutputUnit: {''}
       OutputGroup: [1x1 struct]
             Notes: [0x1 string]
          UserData: []
              Name: ''
      SamplingGrid: [1x1 struct]
```

The `Report` model property contains detailed information on the estimation results. To view the properties and values inside `Report`, use dot notation. For example:

```
m_armax.Report
```

```
ans =
              Status: 'Estimated using ARMAX with simulation focus'
              Method: 'ARMAX'
    InitialCondition: 'zero'
                 Fit: [1x1 struct]
          Parameters: [1x1 struct]
         OptionsUsed: [1x1 idoptions.polyest]
           RandState: [1x1 struct]
            DataUsed: [1x1 struct]
         Termination: [1x1 struct]
```

This action displays the contents of estimation report such as model quality measures (`Fit`), search termination criterion (`Termination`), and a record of estimation data (`DataUsed`) and options (`OptionsUsed`).

See Also

Related Examples

- “Estimate Polynomial Models at the Command Line” on page 6-23

More About

- “What Are Polynomial Models?” on page 6-2

Identifying State-Space Models

- "What Are State-Space Models?" on page 7-2
- "Data Supported by State-Space Models" on page 7-6
- "Supported State-Space Parameterizations" on page 7-7
- "Estimate State-Space Model With Order Selection" on page 7-8
- "Estimate State-Space Models in System Identification App" on page 7-13
- "Estimate State-Space Models at the Command Line" on page 7-22
- "Estimate State-Space Models with Free-Parameterization" on page 7-29
- "Estimate State-Space Models with Canonical Parameterization" on page 7-30
- "Estimate State-Space Models with Structured Parameterization" on page 7-32
- "Estimate State-Space Equivalent of ARMAX and OE Models" on page 7-40
- "Specifying Initial States for Iterative Estimation Algorithms" on page 7-42
- "State-Space Model Estimation Methods" on page 7-43

What Are State-Space Models?

In this section...
“Definition of State-Space Models” on page 7-2
“Continuous-Time Representation” on page 7-2
“Discrete-Time Representation” on page 7-3
“Relationship Between Continuous-Time and Discrete-Time State Matrices” on page 7-4
“State-Space Representation of Transfer Functions” on page 7-4

Definition of State-Space Models

State-space models are models that use state variables to describe a system by a set of first-order differential or difference equations, rather than by one or more *n*th-order differential or difference equations. State variables *x(t)* can be reconstructed from the measured input-output data, but are not themselves measured during an experiment.

The state-space model structure is a good choice for quick estimation because it requires you to specify only one input, the *model order*, `n`. The *model order* is an integer equal to the dimension of *x(t)* and relates to, but is not necessarily equal to, the number of delayed inputs and outputs used in the corresponding linear difference equation.

Continuous-Time Representation

It is often easier to define a parameterized state-space model in continuous time because physical laws are most often described in terms of differential equations. In continuous-time, the state-space description has the following form:

$$\dot{x}(t) = Fx(t) + Gu(t) + \tilde{K}w(t)$$

$$y(t) = Hx(t) + Du(t) + w(t)$$

$$x(0) = x0$$

The matrices *F*, *G*, *H*, and *D* contain elements with physical significance—for example, material constants. *x0* specifies the initial states.

Note $\tilde{K} = 0$ gives the state-space representation of an Output-Error model. For more information, see "What Are Polynomial Models?" on page 6-2.

You can estimate continuous-time state-space model using both time- and frequency-domain data.

Discrete-Time Representation

The discrete-time state-space model structure is often written in the *innovations form* that describes noise:

$$x(kT+T) = Ax(kT) + Bu(kT) + Ke(kT)$$
$$y(kT) = Cx(kT) + Du(kT) + e(kT)$$
$$x(0) = x0$$

where T is the sample time, $u(kT)$ is the input at time instant kT, and $y(kT)$ is the output at time instant kT.

Note K=0 gives the state-space representation of an Output-Error model. For more information about Output-Error models, see "What Are Polynomial Models?" on page 6-2.

Discrete-time state-space models provide the same type of linear difference relationship between the inputs and outputs as the linear ARMAX model, but are rearranged such that there is only one delay in the expressions.

You cannot estimate a discrete-time state-space model using continuous-time frequency-domain data.

The innovations form uses a single source of noise, $e(kT)$, rather than independent process and measurement noise. If you have prior knowledge about the process and measurement noise, you can use linear grey-box estimation to identify a state-space model with structured independent noise sources. For more information, see "Identifying State-Space Models with Separate Process and Measurement Noise Descriptions" on page 13-72.

Relationship Between Continuous-Time and Discrete-Time State Matrices

The relationships between the discrete state-space matrices *A*, *B*, *C*, *D*, and *K* and the continuous-time state-space matrices *F*, *G*, *H*, *D*, and $\tilde{K}$ are given for piece-wise-constant input, as follows:

$$A = e^{FT}$$

$$B = \int_0^T e^{F\tau} G d\tau$$

$$C = H$$

These relationships assume that the input is piece-wise-constant over time intervals $kT \le t < (k+1)T$.

The exact relationship between *K* and $\tilde{K}$ is complicated. However, for short sample time *T*, the following approximation works well:

$$K = \int_0^T e^{F\tau} \tilde{K} d\tau$$

State-Space Representation of Transfer Functions

For linear models, the general model description is given by:

$$y = Gu + He$$

G is a transfer function that takes the input *u* to the output *y*. *H* is a transfer function that describes the properties of the additive output noise model.

The relationships between the transfer functions and the discrete-time state-space matrices on page 7-3 are given by the following equations:

$$G(q) = C(qI_{nx} - A)^{-1}B + D$$

$$H(q) = C(qI_{nx} - A)^{-1}K + I_{ny}$$

Here, I_{nx} is the nx-by-nx identity matrix, and nx is the number of states. I_{ny} is the ny-by-ny identity matrix, and ny is the dimension of y and e.

The state-space representation in the continuous-time case is similar.

See Also

Related Examples

- "Estimate State-Space Models in System Identification App" on page 7-13
- "Estimate State-Space Models at the Command Line" on page 7-22

More About

- "Data Supported by State-Space Models" on page 7-6
- "Supported State-Space Parameterizations" on page 7-7

Data Supported by State-Space Models

You can estimate linear state-space models from data with the following characteristics:

- Time- or frequency-domain data

  To estimate state-space models for time-series data, see “Time Series Analysis”.
- Real data or complex data
- Single-output and multiple-output

To estimate state-space models, you must first import your data into the MATLAB workspace, as described in “Data Preparation”.

Supported State-Space Parameterizations

System Identification Toolbox software supports the following parameterizations that indicate which parameters are estimated and which remain fixed at specific values:

- **Free parameterization** results in the estimation of all elements of the system matrices *A*, *B*, *C*, *D*, and *K*. See "Estimate State-Space Models with Free-Parameterization" on page 7-29.
- **Canonical parameterization** represents a state-space system in a reduced-parameter form where many entries of the *A*, *B* and *C* matrices are fixed to zeros and ones. The free parameters appear in only a few of the rows and columns in the system matrices *A*, *B*, *C* and *D*. The software supports companion, modal decomposition and observability canonical forms. See "Estimate State-Space Models with Canonical Parameterization" on page 7-30.
- **Structured parameterization** lets you specify the fixed values of specific parameters and exclude these parameters from estimation. You choose which entries of the system matrices to estimate and which to treat as fixed. See "Estimate State-Space Models with Structured Parameterization" on page 7-32.
- **Completely arbitrary mapping** of parameters to state-space matrices. See "Estimate Linear Grey-Box Models" on page 13-8.

See Also

- "Estimate State-Space Models with Free-Parameterization" on page 7-29
- "Estimate State-Space Models with Canonical Parameterization" on page 7-30
- "Estimate State-Space Models with Structured Parameterization" on page 7-32

Estimate State-Space Model With Order Selection

In this section...
“Estimate Model With Selected Order in the App” on page 7-8
“Estimate Model With Selected Order at the Command Line” on page 7-11
“Using the Model Order Selection Window” on page 7-11

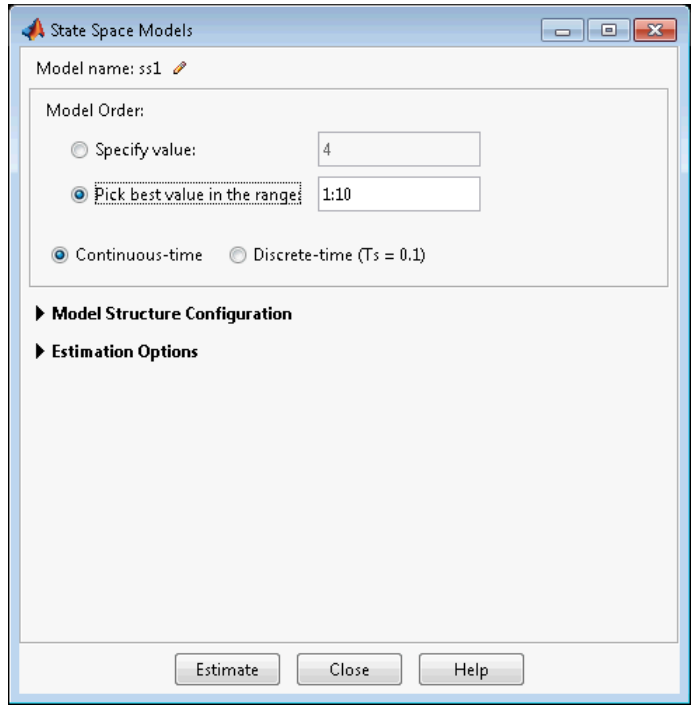
To estimate a state-space model, you must provide a value of its order, which represents the number of states. When you do not know the order, you can search and select an order using the following procedures.

Estimate Model With Selected Order in the App

You must have already imported your data into the app, as described in “Represent Data”.

To estimate model orders for a specific input delay:

1. In the System Identification app, select **Estimate** > **State Space Models** to open the State Space Models dialog box.
2. Select the **Pick best value in the range** option and specify a range in the adjacent field. The default range is `1:10`.

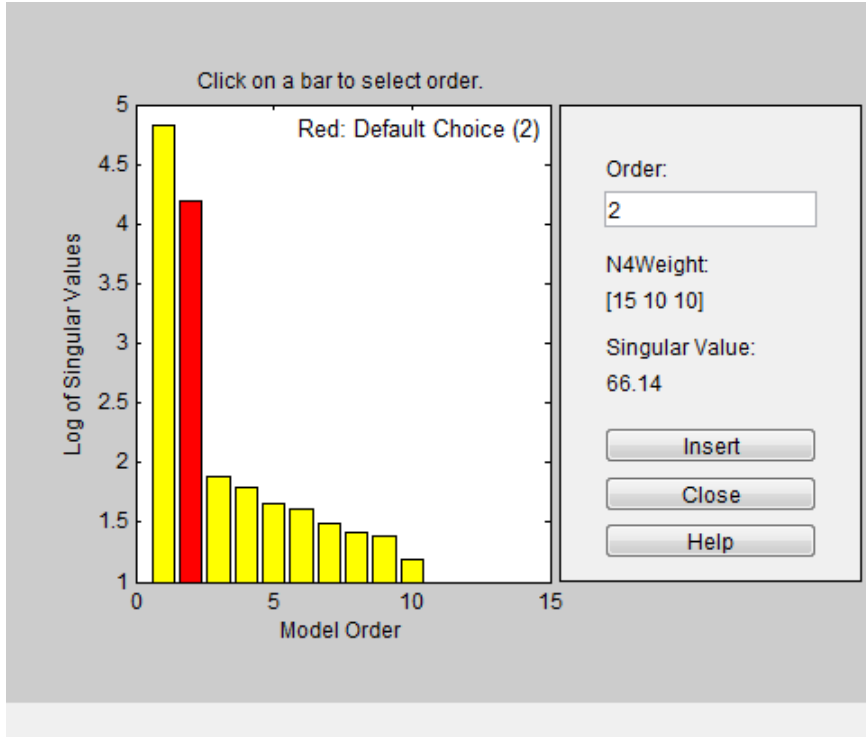


3 (Optional) Expand **Model Structure Configuration** to specify additional attributes of the model structure when searching for best orders. Such attributes include disturbance component, input delays, presence of feedthrough, and parameterization.

4 Expand **Estimation Options** and verify that **Subspace (N4SID)** is selected as the **Method**.

5 Click **Estimate**.

This action opens the Model Order Selection window, which displays the relative measure of how much each state contributes to the input-output behavior of the model (*log of singular values of the covariance matrix*). The following figure shows an example plot.



6 Select the rectangle that represents the cutoff for the states on the left that provide a significant contribution to the input-output behavior.

In the previous figure, states 1 and 2 provide the most significant contribution. The contributions to the right of state 2 drop significantly. Click **Insert** to estimate a model with this order. Red indicates the recommended choice. For information about using the Model Order Selection window, see “Using the Model Order Selection Window” on page 7-11.

This action adds a new model to the Model Board in the System Identification app. The default name of the model is `ss1`. You can use this model as an initial guess for estimating other state-space models, as described in “Estimate State-Space Models in System Identification App” on page 7-13.

7 Click **Close** to close the window.

Estimate Model With Selected Order at the Command Line

You can estimate a state-space model with selected order using `n4sid`, `ssest` or `ssregest`.

Use the following syntax to specify the range of model orders to try for a specific input delay:

```
m = n4sid(data,n1:n2);
```

where `data` is the estimation data set, `n1` and `n2` specify the range of orders.

The command opens the Model Order Selection window. For information about using this plot, see "Using the Model Order Selection Window" on page 7-11.

Alternatively, use `ssest` or `ssregest`:

```
m1 = ssest(data,nn)
m2 = ssregest(data,nn)
```

where `nn = [n1,n2,...,nN]` specifies the vector or range of orders you want to try.

`n4sid` and `ssregest` estimate a model whose sample time matches that of `data` by default, hence a discrete-time model for time-domain data. `ssest` estimates a continuous-time model by default. You can change the default setting by including the `Ts` name-value pair input arguments in the estimation command. For example, to estimate a discrete-time model of optimal order, assuming `Data.Ts>0`, type:

```
model = ssest(data,nn,'Ts',data.Ts);
```

or

```
model = ssregest(data,nn,'Ts',data.Ts);
```

To automatically select the best order without opening the Model Order Selection window, type `m = n4sid(data,'best')`, `m = ssest(data,'best')` or `m = ssregest(data,'best')`.

Using the Model Order Selection Window

The following figure shows a sample Model Order Selection window.

You use this plot to decide which states provide a significant relative contribution to the input-output behavior, and which states provide the smallest contribution. Based on this plot, select the rectangle that represents the cutoff for the states on the left that provide a significant contribution to the input-output behavior. The recommended choice is shown in red. To learn how to generate this plot, see "Estimate Model With Selected Order in the App" on page 7-8 or "Estimate Model With Selected Order at the Command Line" on page 7-11.

The horizontal axis corresponds to the model order `n`. The vertical axis, called **Log of Singular values**, shows the singular values of a covariance matrix constructed from the observed data.

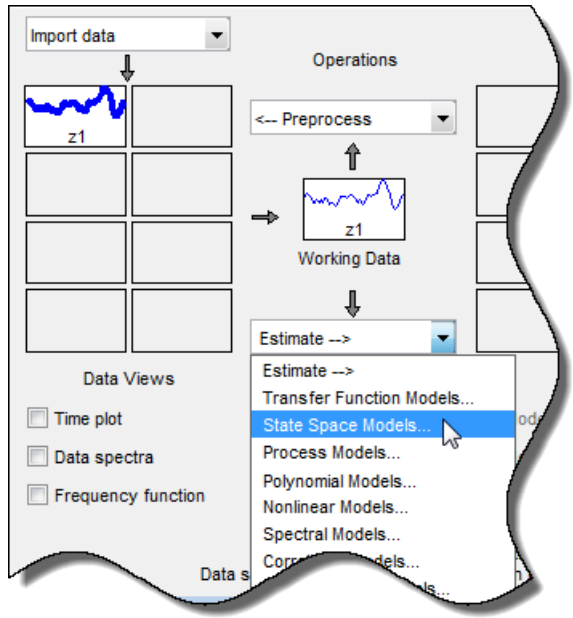
For example, in the previous figure, states 1 and 2 provide the most significant contribution. However, the contributions of the states to the right of state 2 drop significantly. This sharp decrease in the log of the singular values after `n`=2 indicates that using two states is sufficient to get an accurate model.

Estimate State-Space Models in System Identification App

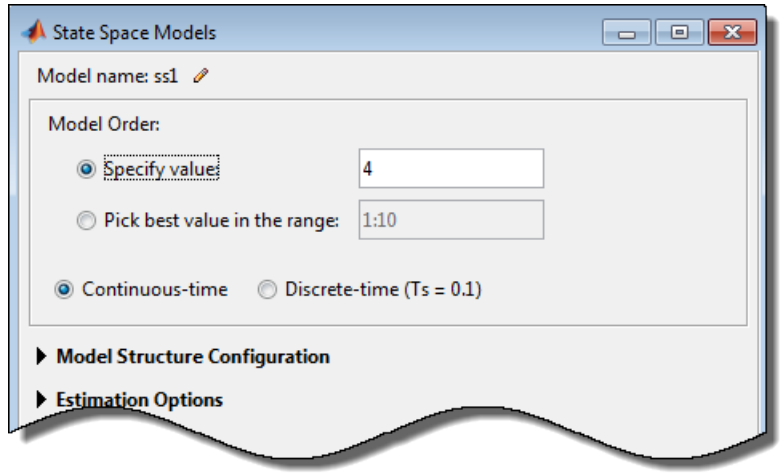
Prerequisites

- Import data into the System Identification app. See “Represent Data”. For supported data formats, see “Data Supported by State-Space Models” on page 7-6.
- Perform data preprocessing. To improve the accuracy of your model, you detrend your data. See “Ways to Prepare Data for System Identification” on page 2-6.

1 Select **Estimate** > **State Space Models**.



The State Space Models dialog box opens.



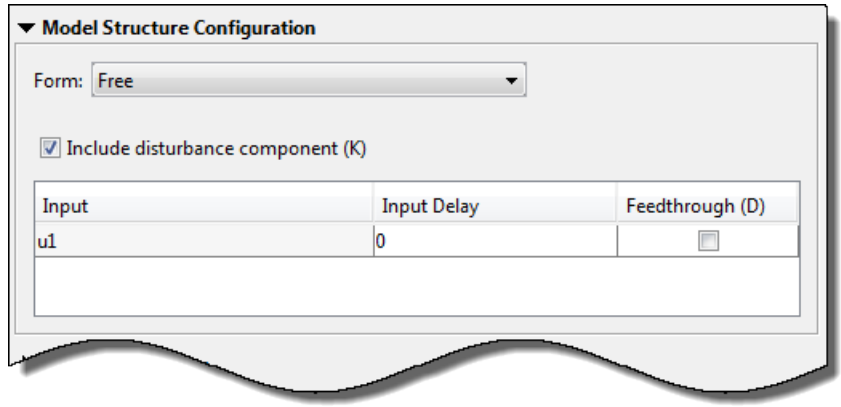
Tip For more information on the options in the dialog box, click **Help**.

2. Specify a model name by clicking adjacent to **Model name**. The name of the model must be unique in the Model Board.
3. Select the **Specify value** option (if not already selected) and specify the model order in the edit field. Model order refers to the number of states in the state-space model.

Tip When you do not know the model order, search for and select an order. For more information, see "Estimate Model With Selected Order in the App" on page 7-8.

4. Select the **Continuous-time** or **Discrete-time** option to specify the type of model to estimate.

   You cannot estimate a discrete-time model if the working data is continuous-time frequency-domain data.
5. Expand the **Model Structure Configuration** section to select the model structure, such as canonical form, whether to estimate the disturbance component (K matrix) and specification of feedthrough and input delays.

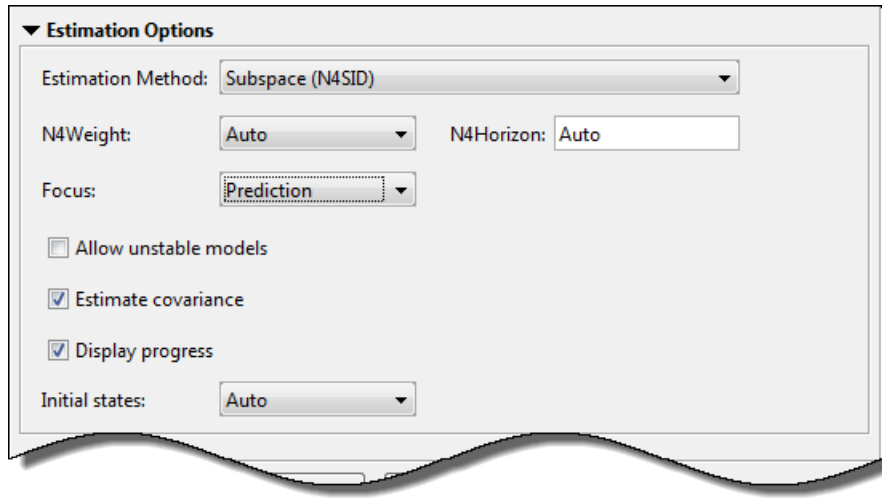


For more information about the type of state-space parameterization, see “Supported State-Space Parameterizations” on page 7-7.

6 Expand the **Estimation Options** section to select the estimation method and configure the cost function.

Select one of the following **Estimation Method** from the drop-down list and configure the options. For more information about these methods, see “State-Space Model Estimation Methods” on page 7-43.

`Subspace (N4SID)`



a In the **N4Weight** drop-down list, specify the weighting scheme used for singular-value decomposition by the N4SID algorithm.

   The N4SID algorithm is used both by the subspace and `Prediction Error Minimization (PEM)` methods.

b In the **N4Horizon** field, specify the forward and backward prediction horizons used by the N4SID algorithm.

   The N4SID algorithm is used both by the subspace and PEM methods.

c In the **Focus** drop-down list, select how to weigh the relative importance of the fit at different frequencies. For more information about each option, see "Assigning Estimation Weightings" on page 7-21.

d Select the **Allow unstable models** check box to specify whether to allow the estimation process to use parameter values that may lead to unstable models.

   Setting this option is same as setting the estimation option `Focus` to `'prediction'` at the command line. An unstable model is delivered only if it produces a better fit to the data than other stable models computed during the estimation process.

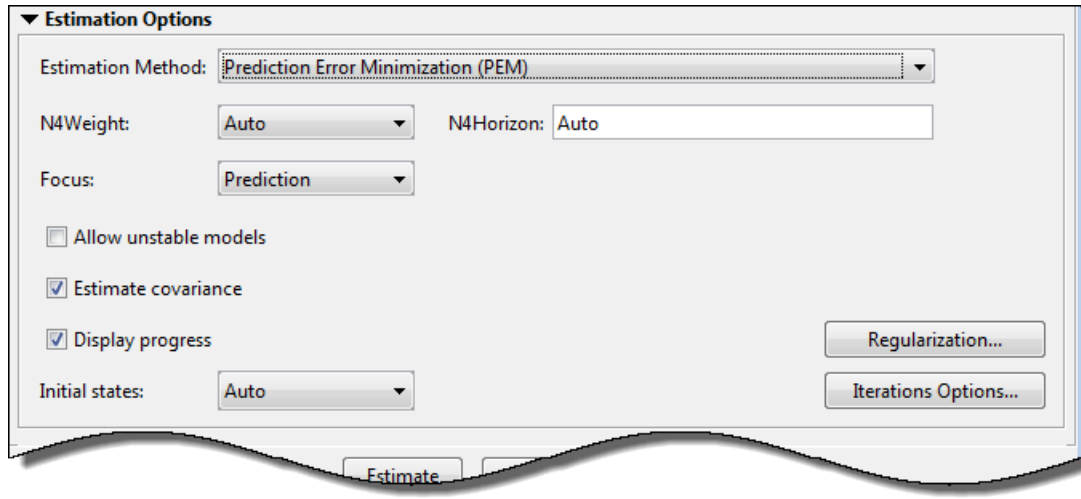
e Select the **Estimate covariance** check box if you want the algorithm to compute parameter uncertainties.

   Effects of such uncertainties are displayed on plots as model confidence regions. Skipping uncertainty computation reduces computation time for complex models and large data sets.

f Select the **Display progress** check box to open a progress viewer window during estimation.

g In the **Initial state** list, specify how you want the algorithm to treat initial states. For more information about the available options, see "Specifying Initial States for Iterative Estimation Algorithms" on page 7-42.

`Prediction Error Minimization (PEM)`



- In the **N4Weight** drop-down list, specify the weighting scheme used for singular-value decomposition by the N4SID algorithm.

  The N4SID algorithm is used both by the subspace and `Prediction Error Minimization (PEM)` methods.

- In the **N4Horizon** field, specify the forward and backward prediction horizons used by the N4SID algorithm.

  The N4SID algorithm is used both by the subspace and PEM methods.

- In the **Focus** drop-down list, select how to weigh the relative importance of the fit at different frequencies. For more information about each option, see "Assigning Estimation Weightings" on page 7-21.

- Select the **Allow unstable models** check box to specify whether to allow the estimation process to use parameter values that may lead to unstable models.

  Setting this option is same as setting the estimation option `Focus` to `'prediction'` at the command line. An unstable model is delivered only if it produces a better fit to the data than other stable models computed during the estimation process.

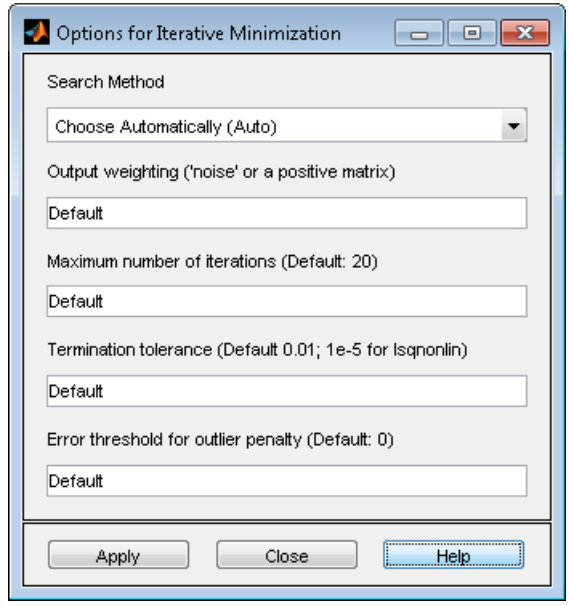
- Select the **Estimate covariance** check box if you want the algorithm to compute parameter uncertainties.

Effects of such uncertainties are displayed on plots as model confidence regions. Skipping uncertainty computation reduces computation time for complex models and large data sets.

- Select the **Display progress** check box to open a progress viewer window during estimation.
- In the **Initial state** list, specify how you want the algorithm to treat initial states. For more information about the available options, see "Specifying Initial States for Iterative Estimation Algorithms" on page 7-42.

Tip If you get an inaccurate fit, try setting a specific method for handling initial states rather than choosing it automatically.

- Click **Regularization** to obtain regularized estimates of model parameters. Specify the regularization constants in the Regularization Options dialog box. To learn more, see "Regularized Estimates of Model Parameters" on page 1-48.
- Click **Iteration Options** to specify options for controlling the iterations. The Options for Iterative Minimization dialog box opens.

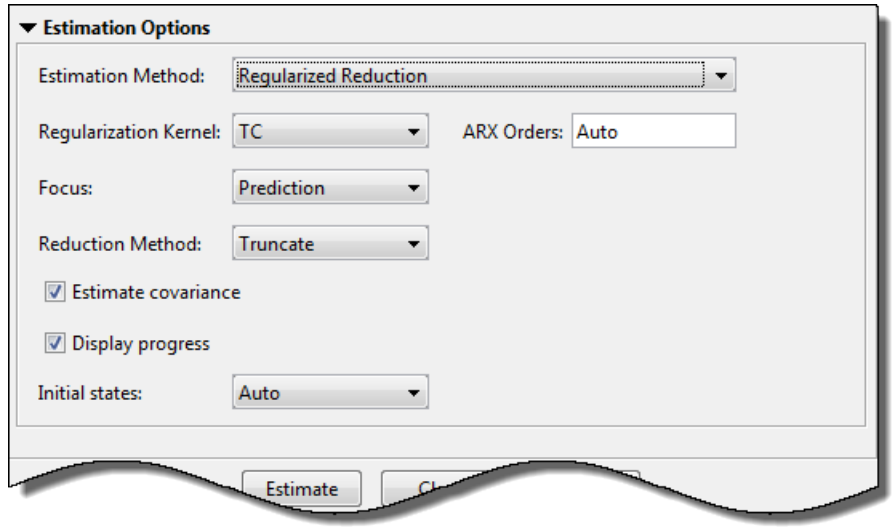


Iteration Options

In the Options for Iterative Minimization dialog box, you can specify the following iteration options:

- **Search Method** — Method used by the iterative search algorithm. Search method is `auto` by default. The descent direction is calculated using `gn` (Gauss-Newton), `gna` (Adaptive Gauss-Newton), `lm` (Levenberg-Marquardt), `lsqnonlin` (Trust-Region Reflective Newton), and `grad` (Gradient Search) successively at each iteration until a sufficient reduction in error is achieved.
- **Output weighting** — Weighting applied to the loss function to be minimized. Use this option for multi-output estimations only. Specify as `'noise'` or a positive semidefinite matrix of size equal the number of outputs.
- **Maximum number of iterations** — Maximum number of iterations to use during search.
- **Termination tolerance** — Tolerance value when the iterations should terminate.
- **Error threshold for outlier penalty** — Robustification of the quadratic criterion of fit.

`Regularized Reduction`



- In the **Regularization Kernel** drop-down list, select the regularizing kernel to use for regularized estimation of the underlying ARX model. To learn more, see "Regularized Estimates of Model Parameters" on page 1-48.
- In the **ARX Orders** field, specify the order of the underlying ARX model. By default, the orders are automatically computed by the estimation algorithm. If you specify a value, it is recommended that you use a large value for `nb` order. To learn more about ARX orders, see `arx`.
- In the **Focus** drop-down list, select how to weigh the relative importance of the fit at different frequencies. For more information about each option, see "Assigning Estimation Weightings" on page 7-21.
- In the **Reduction Method** drop-down list, specify the reduction method:
  - `Truncate` — Discards the specified states without altering the remaining states. This method tends to product a better approximation in the frequency domain, but the DC gains are not guaranteed to match.
  - `MatchDC` — Discards the specified states and alters the remaining states to preserve the DC gain.
- Select the **Estimate covariance** check box if you want the algorithm to compute parameter uncertainties.

  Effects of such uncertainties are displayed on plots as model confidence regions. Skipping uncertainty computation reduces computation time for complex models and large data sets.
- Select the **Display progress** check box to open a progress viewer window during estimation.
- In the **Initial state** list, specify how you want the algorithm to treat initial states. For more information about the available options, see "Specifying Initial States for Iterative Estimation Algorithms" on page 7-42.

  Tip If you get an inaccurate fit, try setting a specific method for handling initial states rather than choosing it automatically.

The estimation process uses parameter values that always lead to a stable model.

7 Click **Estimate** to estimate the model. A new model gets added to the System Identification app.

Next Steps

- Validate the model by selecting the appropriate response type in the **Model Views** area of the app. For more information about validating models, see "Validating Models After Estimation" on page 17-3.
- Export the model to the MATLAB workspace for further analysis by dragging it to the **To Workspace** rectangle in the app.

Assigning Estimation Weightings

You can specify how the estimation algorithm weights the fit at various frequencies. In the app, set **Focus** to one of the following options:

- `Prediction` — Uses the inverse of the noise model H to weigh the relative importance of how closely to fit the data in various frequency ranges. Corresponds to minimizing one-step-ahead prediction, which typically favors the fit over a short time interval. Optimized for output prediction applications.
- `Simulation` — Uses the input spectrum to weigh the relative importance of the fit in a specific frequency range. Does not use the noise model to weigh the relative importance of how closely to fit the data in various frequency ranges. Optimized for output simulation applications.
- `Stability` — Estimates the best stable model. For more information about model stability, see "Unstable Models" on page 17-120.
- `Filter` — Specify a custom filter to open the Estimation Focus dialog box, where you can enter a filter, as described in "Simple Passband Filter" on page 2-134 or "Defining a Custom Filter" on page 2-135. This prefiltering applies only for estimating the dynamics from input to output. The disturbance model is determined from the estimation data.

Estimate State-Space Models at the Command Line

In this section...
“Black Box vs. Structured State-Space Model Estimation” on page 7-22
“Estimating State-Space Models Using ssest, ssregest and n4sid” on page 7-23
“Choosing the Structure of A, B, C Matrices” on page 7-24
“Choosing Between Continuous-Time and Discrete-Time Representations” on page 7-24
“Choosing to Estimate D, K, and X0 Matrices” on page 7-25

Black Box vs. Structured State-Space Model Estimation

You can estimate state-space models in two ways at the command line, depending upon your prior knowledge of the nature of the system and your requirements.

- “Black Box Estimation” on page 7-22
- “Structured Estimation” on page 7-22

Black Box Estimation

In this approach, you specify the model order, and, optionally, additional model structure attributes that configure the overall structure of the state-space matrices. You call `ssest`, `ssregest` or `n4sid` with data and model order as primary input arguments, and use name-value pairs to specify any additional attributes, such as model sample time, presence of feedthrough, absence of noise component, etc. You do not work directly with the coefficients of the *A*, *B*, *C*, *D*, *K*, and *X0* matrices.

Structured Estimation

In this approach, you create and configure an `idss` model that contains the initial values for all the system matrices. You use the `Structure` property of the `idss` model to specify all the parameter constraints. For example, you can designate certain coefficients of system matrices as fixed and impose minimum/maximum bounds on the values of the others. For quick configuration of the parameterization and whether to estimate feedthrough and disturbance dynamics, use `ssform`.

After configuring the `idss` model with desired constraints, you specify this model as an input argument to the `ssest` command. You cannot use `n4sid` or `ssregest` for structured estimation.

Note

- The structured estimation approach is also referred to as grey-box modeling. However, in this toolbox, the "grey box modeling" terminology is used only when referring to `idgrey` and `idnlgrey` models.
- Using the structured estimation approach, you cannot specify relationships among state-space coefficients. Each coefficient is essentially considered to be independent of others. For imposing dependencies, or to use more complex forms of parameterization, use the `idgrey` model and `greyest` estimator.

Estimating State-Space Models Using ssest, ssregest and n4sid

Prerequisites

- Represent input-output data as an `iddata` object or frequency-response data as an `frd` or `idfrd` object. See "Representing Time- and Frequency-Domain Data Using iddata Objects" on page 2-50. For supported data formats, see "Data Supported by State-Space Models" on page 7-6.
- Perform data preprocessing. To improve the accuracy of results when using time-domain data, you can detrend the data or specify the input/output offset levels as estimation options. See "Ways to Prepare Data for System Identification" on page 2-6.
- Select a model order. When you do not know the model order, search and select for an order. For more information, see "Estimate Model With Selected Order at the Command Line" on page 7-11.

You can estimate continuous-time and discrete-time state-space models using the iterative estimation command `ssest` that minimizes the prediction errors to obtain maximum-likelihood values.

Use the following general syntax to both configure and estimate state-space models:

```
m = ssest(data,n,opt,Name,Value)
```

where `data` is the estimation data, `n` is the model order, and `opt` contains options for configuring the estimation of the state-space models. These options include the handling of the initial conditions, input and output offsets, estimation focus and search algorithm options. opt can be followed by name-value pair input arguments that specify optional model structure attributes such as the presence of feedthrough, the canonical form of the model, and input delay.

As an alternative to `ssest`, you can use the noniterative subspace estimators `n4sid` or `ssregest`:

```
m = n4sid(data,n,opt,Name,Value)
m = ssregest(data,n,opt,Name,Value)
```

Unless you specify the sample time as a name-value pair input argument, `n4sid` and `ssregest` estimate a discrete-time model, while `ssest` estimates a continuous-time model.

Note `ssest` uses `n4sid` to initialize the state-space matrices, and takes longer than `n4sid` to estimate a model but typically provides a better fit to data.

For information about validating your model, see "Validating Models After Estimation" on page 17-3

Choosing the Structure of A, B, C Matrices

By default, all entries of the *A*, *B*, and *C* state-space matrices are treated as free parameters. Using the `Form` name-value pair input argument of `ssest` , you can choose various canonical forms, such as the companion and modal forms, that use fewer parameters.

For more information about estimating a specific state-space parameterization, see:

- "Estimate State-Space Models with Free-Parameterization" on page 7-29
- "Estimate State-Space Models with Canonical Parameterization" on page 7-30
- "Estimate State-Space Models with Structured Parameterization" on page 7-32

Choosing Between Continuous-Time and Discrete-Time Representations

For estimation of state-space models, you have the option of switching the model sample time between zero and that of the estimation data. You can do this using the `Ts` name-value pair input argument.

- By default, `ssest` estimates a continuous-time model. If you are using data set with nonzero sample time, `data`, which includes all time domain data, you can also estimate a discrete-time model by using:

  ```
  model = ssest(data,nx,'Ts',data.Ts);
  ```

  If you are using continuous-time frequency-domain data, you cannot estimate a discrete-time model.

- By default, `n4sid` and `ssregest` estimate a model whose sample time matches that of the data. Thus, for time-domain data, `n4sid` and `ssregest` deliver a discrete-time model. You can estimate a continuous-time model by using:

  ```
  model = n4sid(data,nx,'Ts',0);
  ```

  or

  ```
  model = ssregest(data,nx,'Ts',0);
  ```

Choosing to Estimate D, K, and X0 Matrices

For state-space models with any parameterization, you can specify whether to estimate the *D*, *K* and *X0* matrices, which represent the input-to-output feedthrough, noise model and the initial states, respectively.

For state-space models with structured parameterization, you can also specify to estimate the *D* matrix. However, for free and canonical forms, the structure of the *D* matrix is set based on your choice for the `'Feedthrough'` name-value pair input argument.

D Matrix

By default, the *D* matrix is not estimated and its value is fixed to zero, except for static models.

- **Black box estimation:** Use the `Feedthrough` name-value pair input argument to denote the presence or absence of feedthrough from individual inputs. For example, in case of a two input model such that there is feedthrough from only the second input, use:

  ```
  model = n4sid(data,n,'Feedthrough',[false true]);
  ```

- **Structured estimation:** Configure the values of the `init_sys.Structure.D`, where `init_sys` is an `idss` model that represents the desired model structure. To force no feedthrough for the i-th input, set:

  ```
  init_sys.Structure.D.Value(:,i) = 0;
  init_sys.Structure.D.Free = true;
  init_sys.Structure.D.Free(:,i) = false;
  ```

  The first line specifies the value of the i-th column of D as zero. The next line specifies all the elements of D as free, estimable parameters. The last line specifies that the i-th column of the D matrix is fixed for estimation.

  Alternatively, use `ssform` with `'Feedthrough'` name-value pair.

K Matrix

K represents the noise matrix of the model, such that the noise component of the model is:.

$$\dot{x} = Ax + Ke$$
$$y_n = Cx + e$$

For frequency-domain data, no noise model is estimated and K is set to 0. For time-domain data, K is estimated by default in the black box estimation setup. y^n is the contribution of the disturbances to the model output.

- **Black box estimation:** Use the `DisturbanceModel` name-value pair input argument to indicate if the disturbance component is fixed to zero (specify `Value = 'none'`) or estimated as a free parameter (specify `Value = 'estimate'`). For example, use :

  ```
  model = n4sid(data,n,'DisturbanceModel','none');
  ```

- **Structured estimation:** Configure the value of the `init_sys.Structure.K` parameter, where `init_sys` is an `idss` model that represents the desired model structure. You can fix some K matrix coefficients to known values and prescribe minimum/maximum bounds for free coefficients. For example, to estimate only the first column of the K matrix for a two output model:

  ```
  kpar = init_sys.Structure.K;
  kpar.Free(:,1) = true;
  kpar.Free(:,2) = false;
  kpar.Value(:,2) = 0; % second column value is fixed to zero
  init_sys.Structure.K = kpar;
  ```

Alternatively, use `ssform`.

When not sure how to easily fix or free all coefficients of *K*, initially you can omit estimating the noise parameters in *K* to focus on achieving a reasonable model for the system dynamics. After estimating the dynamic model, you can use `ssest` to refine the model while configuring the *K* parameters to be free. For example:

```
init_sys = ssest(data, n,'DisturbanceModel','none');
init_sys.Structure.K.Free = true;
sys = ssest(data,init_sys);
```

where `init_sys` is the dynamic model without noise.

To set *K* to zero in an existing model, you can set its `Value` to `0` and `Free` flag to `false`:

```
m.Structure.K.Value = 0;
m.Structure.K.Free = false;
```

X0 Matrices

The initial state vector *X0* is obtained as the by-product of model estimation. The `n4sid`, `ssest` and `ssregest` commands return the value of *X0* as their second output arguments. You can choose how to handle initial conditions during model estimation by using the `InitialState` estimation option. Use `n4sidOptions` (for `n4sid`), `ssestOptions` (for `ssest`) or `ssregestOptions` (for `ssregest`) to create the estimation option set. For example, in order to hold the initial states to zero during estimation using `n4sid`:

```
opt = n4sidOptions;
opt.InitialState = 'zero';
[m,X0] = n4sid(data,n,opt);
```

The returned `X0` variable is a zero vector of length `n`.

When you estimate models using multiexperiment data, the `X0` matrix contains as many columns as data experiments.

For a complete list of values for the `InitialStates` option, see “Specifying Initial States for Iterative Estimation Algorithms” on page 7-42.

See Also

More About

- "Loss Function and Model Quality Metrics" on page 1-64

Estimate State-Space Models with Free-Parameterization

The default parameterization of the state-space matrices A, B, C, D, and K is free; that is, any elements in the matrices are adjustable by the estimation routines. Because the parameterization of A, B, and C is free, a basis for the state-space realization is automatically selected to give well-conditioned calculations.

To estimate the disturbance model K, you must use time-domain data.

Suppose that you have no knowledge about the internal structure of the discrete-time state-space model. To quickly get started, use the following syntax:

```
m = ssest(data)
```

or

```
m = ssregest(data)
```

where `data` is your estimation data. `ssest` estimates a continuous-time state-space model for an automatically selected order between 1 and 10. `ssregest` estimates a discrete-time model.

To find a model of a specific order `n`, use the following syntax:

```
m = ssest(data,n)
```

or

```
m = ssregest(dat,n)
```

The iterative algorithm `ssest` is initialized by the subspace method `n4sid`. You can use `n4sid` directly, as an alternative to `ssest`:

```
m = n4sid(data)
```

which automatically estimates a discrete-time model of the best order in the 1:10 range.

Estimate State-Space Models with Canonical Parameterization

In this section...
"What Is Canonical Parameterization?" on page 7-30
"Estimating Canonical State-Space Models" on page 7-30

What Is Canonical Parameterization?

Canonical parameterization represents a state-space system in a reduced parameter form where many elements of A, B and C matrices are fixed to zeros and ones. The free parameters appear in only a few of the rows and columns in state-space matrices A, B, C, D, and K. The free parameters are identifiable — they can be estimated to unique values. The remaining matrix elements are fixed to zeros and ones.

The software supports the following canonical forms:

- **Companion form**: The characteristic polynomial appears in the rightmost column of the A matrix.
- **Modal decomposition form**: The state matrix A is block diagonal, with each block corresponding to a cluster of nearby modes.

  Note The modal form has a certain symmetry in its block diagonal elements. If you update the parameters of a model of this form (as a structured estimation using `ssest`), the symmetry is not preserved, even though the updated model is still block-diagonal.

- **Observability canonical form**: The free parameters appear only in select rows of the A matrix and in the B and K matrices.

  For more information about the distribution of free parameters in the observability canonical form, see the Appendix 4A, pp 132-134, on identifiability of black-box multivariable model structures in *System Identification: Theory for the User*, Second Edition, by Lennart Ljung, Prentice Hall PTR, 1999 (equation 4A.16).

Estimating Canonical State-Space Models

You can estimate state-space models with chosen parameterization at the command line.

For example, to specify an observability canonical form, use the `'Form'` name-value pair input argument, as follows:

```
m = ssest(data,n,'Form','canonical')
```

Similarly, set `'Form'` as `'modal'` or `'companion'` to specify modal decomposition and companion canonical forms, respectively.

If you have time-domain data, the preceding command estimates a continuous-time model. If you want a discrete-time model, specify the data sample time using the `'Ts'` name-value pair input argument:

```
md = ssest(data, n,'Form','canonical','Ts',data.Ts)
```

If you have continuous-time frequency-domain data, you can only estimate a continuous-time model.

Estimate State-Space Models with Structured Parameterization

In this section...
“What Is Structured Parameterization?” on page 7-32 “Specify the State-Space Model Structure” on page 7-33 “Are Grey-Box Models Similar to State-Space Models with Structured Parameterization?” on page 7-34 “Estimate Structured Discrete-Time State-Space Models” on page 7-35 “Estimate Structured Continuous-Time State-Space Models” on page 7-37

What Is Structured Parameterization?

Structured parameterization lets you exclude specific parameters from estimation by setting these parameters to specific values. This approach is useful when you can derive state-space matrices from physical principles and provide initial parameter values based on physical insight. You can use this approach to discover what happens if you fix specific parameter values or if you free certain parameters.

There are two stages to the structured estimation procedure:

1. Specify the state-space model structure, as described in “Specify the State-Space Model Structure” on page 7-33
2. Estimate the free model parameters, as described in “Estimate State-Space Models at the Command Line” on page 7-22

This approach differs from estimating models with free and canonical parameterizations, where it is not necessary to specify initial parameter values before the estimation. For free parameterization, there is no structure to specify because it is assumed to be unknown. For canonical parameterization, the structure is fixed to a specific form.

Note To estimate structured state-space models in the System Identification app, define the corresponding model structures at the command line and import them into the System Identification app.

Specify the State-Space Model Structure

To specify the state-space model structure:

1 Use `idss` to create a state-space model. For example:

```
A = [0 1; 0 -1];
B = [0; 0.28];
C = eye(2);
D = zeros(2,1);
m = idss(A,B,C,D,K,'Ts',T)
```

creates a discrete-time state-space structure, where `A`, `B`, `C`, `D`, and `K` specify the initial values for the free parameters. `T` is the sample time.

2 Use the `Structure` property of the model to specify which parameters to estimate and which to set to specific values.

More about Structure

`Structure` contains parameters for the five state-space matrices, *A*, *B*, *C*, *D*, and *K*.

For each parameter, you can set the following attributes:

- `Value` — Parameter values. For example, `sys.Structure.A.Value` contains the initial or estimated values of the *A* matrix.

  `NaN` represents unknown parameter values.

  Each property `sys.A`, `sys.B`, `sys.C`, and `sys.D` is an alias to the corresponding `Value` entry in the `Structure` property of `sys`. For example, `sys.A` is an alias to the value of the property `sys.Structure.A.Value`
- `Minimum` — Minimum value that the parameter can assume during estimation. For example, `sys.Structure.K.Minimum = 0` constrains all entries in the *K* matrix to be greater than or equal to zero.
- `Maximum` — Maximum value that the parameter can assume during estimation.
- `Free` — Boolean specifying whether the parameter is a free estimation variable. If you want to fix the value of a parameter during estimation, set the corresponding `Free = false`. For example, if *A* is a 3-by-3 matrix, `sys.Structure.A.Free = eyes(3)` fixes all of the off-diagonal entries in *A*, to

the values specified in `sys.Structure.A.Value`. In this case, only the diagonal entries in *A* are estimable.

- `Scale` — Scale of the parameter's value. `Scale` is not used in estimation.
- `Info` — Structure array for storing parameter units and labels. The structure has `Label` and `Unit` fields.

  Specify parameter units and labels as character vectors. For example, `'Time'`.

For example, if you want to fix `A(1,2)=A(2,1)=0`, use:

```
m.Structure.A.Value(1,2) = 0;
m.Structure.A.Value(2,1) = 0;
m.Structure.A.Free(1,2) = false;
m.Structure.A.Free(2,1) = false;
```

The estimation algorithm only estimates the parameters in `A` for which `m.Structure.A.Free` is `true`.

Use physical insight, whenever possible, to initialize the parameters for the iterative search algorithm. Because it is possible that the numerical minimization gets stuck in a local minimum, try several different initialization values for the parameters. For random initialization, use `init`. When the model structure contains parameters with different orders of magnitude, try to scale the variables so that the parameters are all roughly the same magnitude.

Alternatively, to quickly configure the parameterization and whether to estimate feedthrough and disturbance dynamics, use `ssform`.

3 Use `ssest` to estimate the model, as described in "Estimate State-Space Models at the Command Line" on page 7-22.

The iterative search computes gradients of the prediction errors with respect to the parameters using numerical differentiation. The step size is specified by the `nuderst` command. The default step size is equal to 10^{-4} times the absolute value of a parameter or equal to 10^{-7}, whichever is larger. To specify a different step size, edit the `nuderst` MATLAB file.

Are Grey-Box Models Similar to State-Space Models with Structured Parameterization?

You estimate state-space models with structured parameterization on page 7-32 when you know some parameters of a linear system and need to estimate the others. These

models are therefore similar to grey-box models. However, in this toolbox, the "grey box modeling" terminology is used only when referring to `idgrey` and `idnlgrey` models. In these models, you can specify complete linear or nonlinear models with complicated relationships between the unknown parameters.

If you have independent unknown matrix elements in a linear state-space model structure, then it is easier and quicker to use state-space models with structured parameterizations. For imposing dependencies, or to use more complex forms of parameterization, use the `idgrey` model and the associated `greyest` estimator. For more information, see “Grey-Box Model Estimation”.

If you want to incorporate prior knowledge regarding the state and output covariances into the estimation process, use an `idgrey` model to identify your system using a general state-space model structure. For more information, see “Identifying State-Space Models with Separate Process and Measurement Noise Descriptions” on page 13-72.

Estimate Structured Discrete-Time State-Space Models

This example shows how to estimate the unknown parameters of a discrete-time model.

In this example, you estimate $\theta_1, \theta_2, \theta_3, \theta_4, \theta_5$ in the following discrete-time model:

$$x(t+1) = \begin{bmatrix} 1 & \theta_1 \\ 0 & 1 \end{bmatrix} x(t) + \begin{bmatrix} \theta_2 \\ \theta_3 \end{bmatrix} u(t) + \begin{bmatrix} \theta_4 \\ \theta_5 \end{bmatrix} e(t)$$

$$y(t) = \begin{bmatrix} 1 & 0 \end{bmatrix} x(t) + e(t)$$

$$x(0) = \begin{bmatrix} 0 \\ 0 \end{bmatrix}$$

Suppose that the nominal values of the unknown parameters ($\theta_1, \theta_2, \theta_3, \theta_4, \theta_5$) are `-1`, `2`, `3`, `4`,and `5`, respectively.

The discrete-time state-space model structure is defined by the following equation:

$$x(kT+T) = Ax(kT) + Bu(kT) + Ke(kT)$$

$$y(kT) = Cx(kT) + Du(kT) + e(kT)$$

$$x(0) = x0$$

Construct the parameter matrices and initialize the parameter values using the nominal parameter values.

```
A = [1,-1;0,1];
B = [2;3];
C = [1,0];
D = 0;
K = [4;5];
```

Construct the state-space model object.

```
m = idss(A,B,C,D,K);
```

Specify the parameter values in the structure matrices that you do not want to estimate.

```
S = m.Structure;
S.A.Free(1,1) = false;
S.A.Free(2,:) = false;
S.C.Free = false;
m.Structure = S;
```

`D` is initialized, by default, as a fixed value, and `K` and `B` are initialized as free values. Suppose you want to fix the initial states to known zero values. To enforce this, configure the `InitialState` estimation option.

```
opt = ssestOptions;
opt.InitialState = 'zero';
```

Load estimation data.

```
load iddata1 z1;
```

Estimate the model structure.

```
m = ssest(z1,m,opt);
```

where `z1` is name of the `iddata` object. The data can be time-domain or frequency-domain data. The iterative search starts with the nominal values in the `A`, `B`, `C`, `D`, and `K` matrices.

Estimate Structured Continuous-Time State-Space Models

This example shows how to estimate the unknown parameters of a continuous-time model.

In this example, you estimate $\theta_1, \theta_2, \theta_3$ in the following continuous-time model:

$$\dot{x}(t) = \begin{bmatrix} 0 & 1 \\ 0 & \theta_1 \end{bmatrix} x(t) + \begin{bmatrix} 0 \\ \theta_2 \end{bmatrix} u(t)$$
$$y(t) = \begin{bmatrix} 1 & 0 \\ 0 & 1 \end{bmatrix} x(t) + e(t)$$
$$x(0) = \begin{bmatrix} \theta_3 \\ 0 \end{bmatrix}$$

This equation represents an electrical motor, where $y_1(t) = x_1(t)$ is the angular position of the motor shaft, and $y_2(t) = x_2(t)$ is the angular velocity. The parameter $-\theta_1$ is the inverse time constant of the motor, and θ_2 / θ_1 is the static gain from the input to the angular velocity.

The motor is at rest at `t=0` , but its angular position θ_3 is unknown. Suppose that the approximate nominal values of the unknown parameters are $\theta_1 = -1$ and $\theta_2 = 0.25$.

The variance of the errors in the position measurement is `0.01` , and the variance in the angular velocity measurements is `0.1` . For more information about this example, see the section on state-space models in *System Identification: Theory for the User*, Second Edition, by Lennart Ljung, Prentice Hall PTR, 1999.

The continuous-time state-space model structure is defined by the following equation:
$\dot{x}(t) = Fx(t) + Gu(t) + \tilde{K}w(t)$
$y(t) = Hx(t) + Du(t) + w(t)$
$x(0) = x0$

Construct the parameter matrices and initialize the parameter values using the nominal parameter values.

```
A = [0 1;0 -1];
B = [0;0.25];
C = eye(2);
D = [0;0];
```

```
K = zeros(2,2);
x0 = [0;0];
```

The matrices correspond to continuous-time representation. However, to be consistent with the `idss` object property name, this example uses `A`, `B`, and `C` instead of `F`, `G`, and `H`.

Construct the continuous-time state-space model object.

```
m = idss(A,B,C,D,K,'Ts',0);
```

Specify the parameter values in the structure matrices that you do not want to estimate.

```
S = m.Structure;
S.A.Free(1,:) = false;
S.A.Free(2,1) = false;
S.B.Free(1) = false;
S.C.Free = false;
S.D.Free = false;
S.K.Free = false;
m.Structure = S;
m.NoiseVariance = [0.01 0; 0 0.1];
```

The initial state is partially unknown. Use the `InitialState` option of the `ssestOptions` option set to configure the estimation behavior of `X0`.

```
opt = ssestOptions;
opt.InitialState = idpar(x0);
opt.InitialState.Free(2) = false;
```

Estimate the model structure.

```
load(fullfile(matlabroot,'toolbox','ident','iddemos','data','dcmotordata'));
z = iddata(y,u,0.1);
m = ssest(z,m,opt);
```

The iterative search for a minimum is initialized by the parameters in the nominal model `m`. The continuous-time model is sampled using the same sample time as the data during estimation.

Simulate this system using the sample time `T=0.1` for input `u` and the noise realization `e`.

```
e = randn(300,2);
u1 = idinput(300);
```

```
simdat = iddata([],u1,'Ts',0.1);
simopt = simOptions('AddNoise',true,'NoiseData',e);
y1 = sim(m,simdat,simopt);
```

The continuous system is sampled using `Ts=0.1` for simulation purposes. The noise sequence is scaled according to the matrix `m.NoiseVariance` .

If you discover that the motor was not initially at rest, you can estimate $x_2(0)$ by setting the second element of the `InitialState` parameter to be free.

```
opt.InitialState.Free(2) = true;
m_new = ssest(z,m,opt);
```

Estimate State-Space Equivalent of ARMAX and OE Models

This example shows how to estimate ARMAX and OE-form models using the state-space estimation approach.

You can estimate the equivalent of multiple-output ARMAX and Output-Error (OE) models using state-space model structures:

- For an `armax` model, specify to estimate the K matrix for the state-space model.
- For an `oe` model, set $K = 0$.

Convert the resulting models into `idpoly` models to see them in the commonly defined ARMAX or OE forms.

Load measured data.

```
load iddata1 z1
```

Estimate state-space models.

```
mss_noK = n4sid(z1,2,'DisturbanceModel','none');
mss = n4sid(z1,2);
```

`mss_noK` is a second order state-space model with no disturbance model used during estimation. `mss` is also a second order state-space model, but with an estimated noise component. Both models use the measured data set `z1` for estimation.

Convert the state-space models to polynomial models.

```
mOE = idpoly(mss_noK);
mARMAX = idpoly(mss);
```

Converting to polynomial models results in the parameter covariance information for `mOE` and `mARMAX` to be lost.

You can use one of the following to recompute the covariance:

- Zero-iteration update using the same estimation data.
- `translatecov` as a Gauss approximation formula-based translation of covariance of `mss_noK` and `mss` into covariance of `mOE` and `mARMAX`.

Reestimate `mOE` and `mARMAX` for the parameters of the polynomial model using a zero iteration update.

```
opt = polyestOptions;
opt.SearchOption.MaxIter = 0;

mOE = polyest(z1,mOE,opt);
mARMAX = polyest(z1,mARMAX,opt);
```

The options object, `opt`, specifies a zero iteration update for `mOE` and `mARMAX`. Consequently, the model parameters remain unchanged and only their covariance information is updated.

Alternatively, you can use `translatecov` to convert the estimated models into polynomial form.

```
fcn = @(x)idpoly(x);
mOEt = translatecov(fcn,mss_noK);
mARMAXt = translatecov(fcn,mss);
```

Because `polyest` and `translatecov` use different computation algorithms, the covariance data obtained by running a zero-iteration update may not match that obtained using `translatecov`.

You can view the uncertainties of the model parameters using `present(mOE)` and `present(mARMAX)`.

You can use a state-space model with $K = 0$ (Output-Error (OE) form) for initializing a Hammerstein-Wiener estimation at the command line. This initialization may improve the fit of the model. See “Initialize Hammerstein-Wiener Estimation Using Linear Model” on page 12-8.

For more information about ARMAX and OE models, see “Input-Output Polynomial Models”.

Specifying Initial States for Iterative Estimation Algorithms

When you estimate state-space models, you can specify how the algorithm treats initial states. This information supports the estimation procedures “Estimate State-Space Models in System Identification App” on page 7-13 and “Estimate State-Space Models at the Command Line” on page 7-22.

In the System Identification app, set **Initial state** to one of the following options:

- `Auto` — Automatically chooses `Zero`, `Estimate`, or `Backcast` based on the estimation data. If initial states have negligible effect on the prediction errors, the initial states are set to zero to optimize algorithm performance.
- `Zero` — Sets all initial states to zero.
- `Estimate` — Treats the initial states as an unknown vector of parameters and estimates these states from the data.
- `Backcast` — Estimates initial states using a backward filtering method (least-squares fit).

At the command line, specify the method for handling initial states using the `InitialState` estimation option. For example, to estimate a fourth-order state-space model and set the initial states to be estimated from the data:

```
opt = ssestOptions('InitialState','estimate');
m = ssest(data,4,opt)
```

For a complete list of values for the `InitialState` model property, see the `ssestOptions`, `n4sidOptions` and `ssregestOptions` reference pages.

Note For the `n4sid` algorithm, `'auto'` and `'backcast'` are equivalent to `'estimate'`.

State-Space Model Estimation Methods

You can estimate state-space models using one of the following estimation methods:

- *N4SID* — Noniterative, subspace method. The method works on both time-domain and frequency-domain data and is typically faster than the *SSEST* algorithm. You can choose the subspace algorithms such as `CVA`, `SSARX`, or `MOESP` using the `n4Weight` option. You can also use this method to get an initial model (see `n4sid`), and then refine the initial estimate using the iterative prediction-error method `ssest`.

  For more information about this algorithm, see [1].

- *SSEST* — Iterative method that uses *prediction error minimization* algorithm. The method works on both time-domain and frequency-domain data. For black-box estimation, the method initializes the model parameters using `n4sid` and then updates the parameters using an iterative search to minimize the prediction errors. You can also use this method for structured estimation using an initial model with initial values of one or more parameters fixed in value.

  For more information on this algorithm, see [2].

- *SSREGEST* — Noniterative method. The method works on discrete time-domain data and frequency-domain data. It first estimates a high-order regularized ARX or FIR model, converts it to a state-space model and then performs balanced reduction on it. This method provides improved accuracy on short, noisy data sets.

With all the estimation methods, you have the option of specifying how to handle initial state, delays, feedthrough behavior and disturbance component of the model.

References

[1] van Overschee, P., and B. De Moor. *Subspace Identification of Linear Systems: Theory, Implementation, Applications.* Springer Publishing: 1996.

[2] Ljung, L. *System Identification: Theory For the User*, Second Edition, Upper Saddle River, N.J: Prentice Hall, 1999.

[3] T. Chen, H. Ohlsson, and L. Ljung. “On the Estimation of Transfer Functions, Regularizations and Gaussian Processes - Revisited”, *Automatica*, Volume 48, August 2012.

Identifying Transfer Function Models

- "What are Transfer Function Models?" on page 8-2
- "Data Supported by Transfer Function Models" on page 8-5
- "Estimate Transfer Function Models in the System Identification App" on page 8-6
- "Estimate Transfer Function Models at the Command Line" on page 8-13
- "Transfer Function Structure Specification" on page 8-14
- "Estimate Transfer Function Models by Specifying Number of Poles" on page 8-15
- "Estimate Transfer Function Models with Transport Delay to Fit Given Frequency-Response Data" on page 8-16
- "Estimate Transfer Function Models With Prior Knowledge of Model Structure and Constraints" on page 8-17
- "Estimate Transfer Function Models with Unknown Transport Delays" on page 8-19
- "Estimate Transfer Functions with Delays" on page 8-21
- "Specifying Initial Conditions for Iterative Estimation Algorithms" on page 8-22
- "Troubleshoot Frequency-Domain Identification of Transfer Function Models" on page 8-23

What are Transfer Function Models?

In this section...
“Definition of Transfer Function Models” on page 8-2
“Continuous-Time Representation” on page 8-2
“Discrete-Time Representation” on page 8-2
“Delays” on page 8-3
“Multi-Input Multi-Output Models” on page 8-3

Definition of Transfer Function Models

Transfer function models describe the relationship between the inputs and outputs of a system using a ratio of polynomials. The *model order* is equal to the order of the denominator polynomial. The roots of the denominator polynomial are referred to as the model *poles*. The roots of the numerator polynomial are referred to as the model *zeros*.

The parameters of a transfer function model are its poles, zeros and transport delays.

Continuous-Time Representation

In continuous-time, a transfer function model has the form:

$$Y(s) = \frac{num(s)}{den(s)}U(s) + E(s)$$

Where, $Y(s)$, $U(s)$ and $E(s)$ represent the Laplace transforms of the output, input and noise, respectively. $num(s)$ and $den(s)$ represent the numerator and denominator polynomials that define the relationship between the input and the output.

Discrete-Time Representation

In discrete-time, a transfer function model has the form:

$$y(t) = \frac{num(q^{-1})}{den(q^{-1})}u(t) + e(t)$$

$$num(q^{-1}) = b_0 + b_1 q^{-1} + b_2 q^{-2} + \ldots$$

$$den(q^{-1}) = 1 + a_1 q^{-1} + a_2 q^{-2} + \ldots$$

The roots of *num*(*q*^*-1*) and *den*(*q*^*-1*) are expressed in terms of the lag variable *q*^*-1*.

If you take the Z-transform, the transfer function has the form:

$$Y(z^{-1}) = \frac{num(z^{-1})}{den(z^{-1})}U(z^{-1}) + E(z^{-1})$$

$$num(z^{-1}) = b_0 + b_1 z^{-1} + b_2 z^{-2} + \ldots$$

$$den(z^{-1}) = 1 + a_1 z^{-1} + a_2 z^{-2} + \ldots$$

Where, $Y(z^{-1})$, $U(z^{-1})$ and $E(z^{-1})$ represent the Z-transforms of the output, input and noise, respectively. z^{-1} is the Z-transform of the lag operator.

Delays

In continuous-time, input and transport delays are of the form:

$$Y(s) = \frac{num(s)}{den(s)}e^{-s\tau}U(s) + E(s)$$

Where τ represents the delay.

In discrete-time:

$$y(t) = \frac{num}{den}u(t-\tau) + e(t)$$

where *num* and *den* are polynomials in the lag operator `q^(-1)`.

Multi-Input Multi-Output Models

A single-input single-output (SISO) continuous transfer function has the form $G(s) = \frac{num(s)}{den(s)}$. The corresponding transfer function model can be represented as:

$$Y(s) = G(s)U(s) + E(s)$$

A multi-input multi-output (MIMO) transfer function contains a SISO transfer function corresponding to each input-output pair in the system. For example, a continuous-time transfer function model with two inputs and two outputs has the form:

$$Y_1(s) = G_{11}(s)U_1(s) + G_{12}(s)U_2(s) + E_1(s)$$
$$Y_2(s) = G_{21}(s)U_1(s) + G_{22}(s)U_2(s) + E_2(s)$$

Where, $G_{ij}(s)$ is the SISO transfer function between the i^{th} output and the j^{th} input. $E_1(s)$ and $E_2(s)$ are the Laplace transforms of the noise corresponding to the two outputs.

The representation of discrete-time MIMO transfer function models is analogous.

See Also

More About

- "Data Supported by Transfer Function Models" on page 8-5
- "Estimate Transfer Function Models in the System Identification App" on page 8-6
- "Estimate Transfer Function Models at the Command Line" on page 8-13

Data Supported by Transfer Function Models

You can estimate transfer function models from data with the following characteristics:

- Real data or complex data
- Single-output and multiple-output
- Time- or frequency-domain data

  Note that you cannot use time-series data for transfer function model identification.

You must first import your data into the MATLAB workspace, as described in "Data Preparation".

See Also

More About

- "Estimate Transfer Function Models in the System Identification App" on page 8-6
- "Estimate Transfer Function Models at the Command Line" on page 8-13

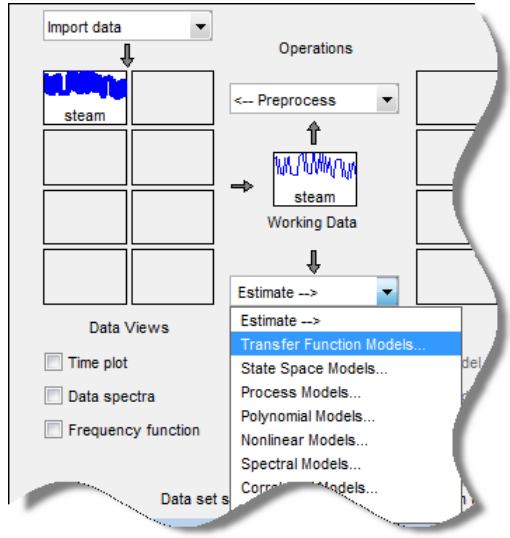
Estimate Transfer Function Models in the System Identification App

This topic shows how to estimate transfer function models in the System Identification app.

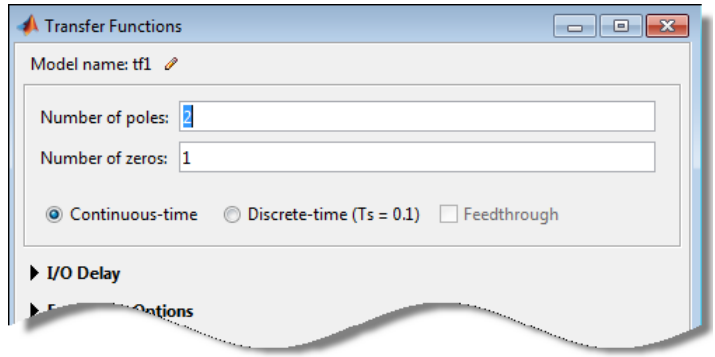
Prerequisites

- Import data into the System Identification app. See “Represent Data”. For supported data formats, see “Data Supported by Transfer Function Models” on page 8-5.
- Perform any required data preprocessing operations. If input and/or output signals contain nonzero offsets, consider detrending your data. See “Ways to Prepare Data for System Identification” on page 2-6.

1 In the System Identification app, select **Estimate > Transfer Function Models**



The Transfer Functions dialog box opens.



Tip For more information on the options in the dialog box, click **Help**.

2 In the **Number of poles** and **Number of zeros** fields, specify the number of poles and zeros of the transfer function as nonnegative integers.

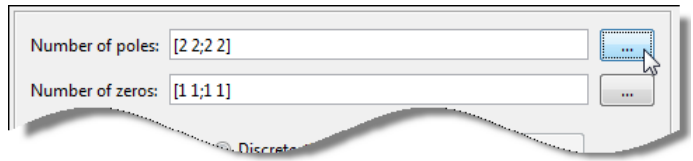
Multi-Input, Multi-Output Models

For systems that are multiple input, multiple output, or both:

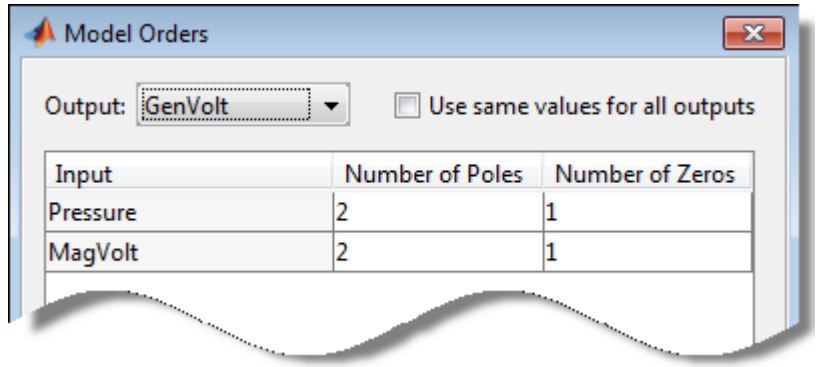
- To use the same number of poles or zeros for all the input/output pairs, specify a scalar.
- To use a different number of poles and zeros for the input/output pairs, specify an *ny*-by-*nu* matrix. *ny* is the number of outputs and *nu* is the number of inputs.

Alternatively, click

.



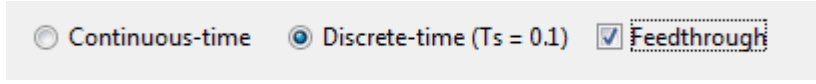
The Model Orders dialog box opens where you specify the number of poles and zeros for each input/output pair. Use the **Output** list to select an output.



3 Select **Continuous-time** or **Discrete-time** to specify whether the model is a continuous- or discrete-time transfer function.

For discrete-time models, the number of poles and zeros refers to the roots of the numerator and denominator polynomials expressed in terms of the lag variable `q^-1`.

4 (For discrete-time models only) Specify whether to estimate the model feedthrough. Select the **Feedthrough** check box.



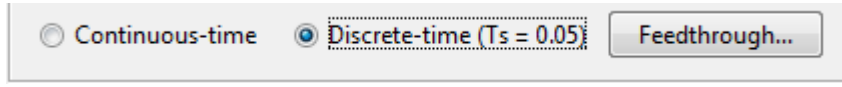
A discrete-time model with 2 poles and 3 zeros takes the following form:

$$H(z^{-1}) = \frac{b0 + b1z^{-1} + b2z^{-2} + b3z^{-3}}{1 + a1z^{-1} + a2z^{-2}}$$

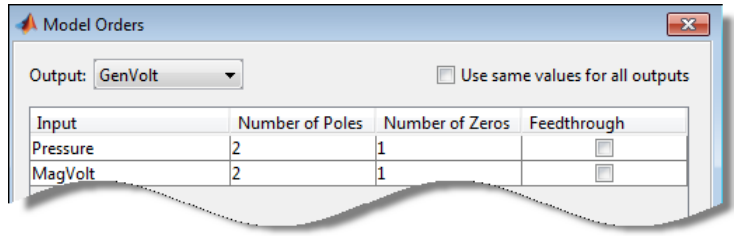
When the model has direct feedthrough, `b0` is a free parameter whose value is estimated along with the rest of the model parameters `b1`, `b2`, `b3`, `a1`, `a2`. When the model has no feedthrough, `b0` is fixed to zero.

Multi-Input, Multi-Output Models

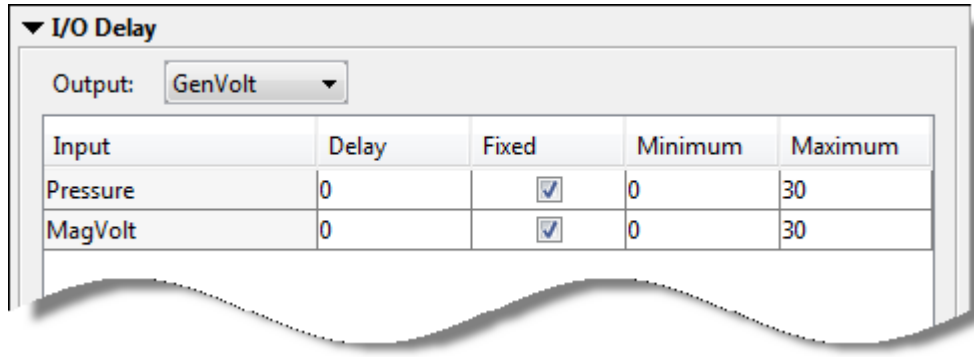
For models that are multi input, multi output or both, click **Feedthrough**.



The Model Orders dialog box opens, where you specify to estimate the feedthrough for each input/output pair separately. Use the **Output** list to select an output.

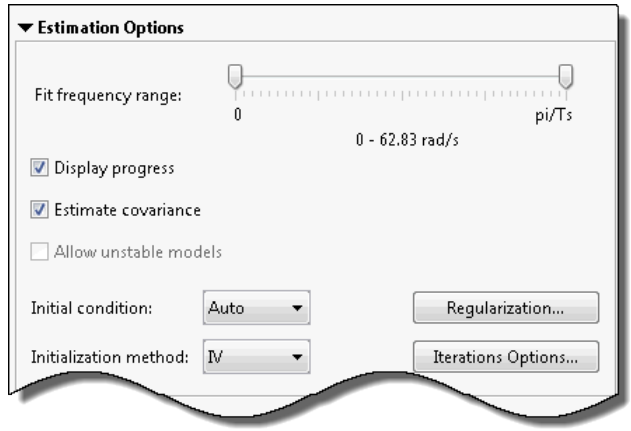


5 Expand the **I/O Delay** section to specify nominal values and constraints for transport delays for different input/output pairs.



Use the **Output** list to select an output. Select the **Fixed** check box to specify a transport delay as a fixed value. Specify its nominal value in the **Delay** field.

6 Expand the **Estimation Options** section to specify estimation options.



- Set the range slider to the desired passband to specify the frequency range over which the transfer function model must fit the data. By default the entire frequency range (0 to Nyquist frequency) is covered.
- Select **Display progress** to view the progress of the optimization.
- Select **Estimate covariance** to estimate the covariance of the transfer function parameters.
- (For frequency-domain data only) Specify whether to allow the estimation process to use parameter values that may lead to unstable models. Select the **Allow unstable models** option.

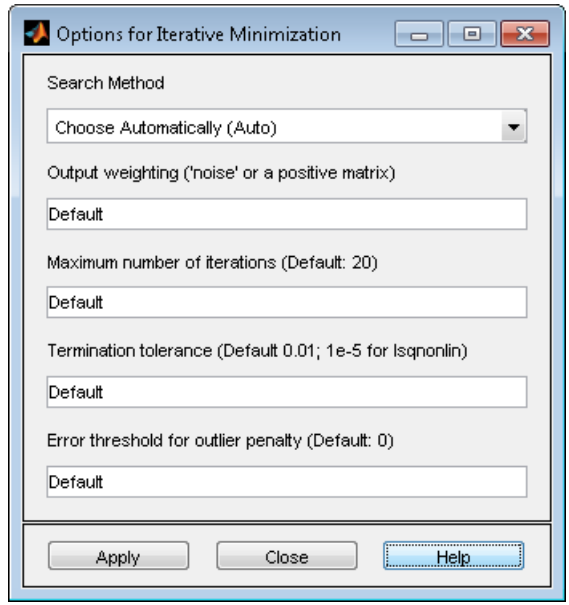
  Setting this option is same as setting the estimation option `Focus` to `'prediction'` at the command line. An unstable model is delivered only if it produces a better fit to the data than other stable models computed during the estimation process.
- Specify how to treat the initial conditions in the **Initial condition** list. For more information, see “Specifying Initial Conditions for Iterative Estimation Algorithms” on page 8-22.
- Specify the algorithm used to initialize the values of the numerator and denominator coefficients in the **Initialization method** list.

  - `IV` — Instrument Variable approach.
  - `SVF` — State Variable Filters approach.
  - `N4SID` — Generalized Poisson Moment Functions approach.

- `GPMF` — Subspace state-space estimation approach.
- `All` — Combination of all of the above approaches. The software tries all the above methods and selects the method that yields the smallest value of prediction error norm.

7 Click **Regularization** to obtain regularized estimates of model parameters. Specify the regularization constants in the Regularization Options dialog box. To learn more, see "Regularized Estimates of Model Parameters" on page 1-48.

8 Click **Iterations Options** to specify options for controlling the iterations. The Options for Iterative Minimization dialog box opens.



Iteration Options

In the Options for Iterative Minimization dialog box, you can specify the following iteration options:

- **Search Method** — Method used by the iterative search algorithm. Search method is `auto` by default. The descent direction is calculated using `gn` (Gauss-Newton), `gna` (Adaptive Gauss-Newton), `lm` (Levenberg-Marquardt), `lsqnonlin` (Trust-Region Reflective Newton), and `grad` (Gradient Search) successively at each iteration until a sufficient reduction in error is achieved.

- **Output weighting** — Weighting applied to the loss function to be minimized. Use this option for multi-output estimations only. Specify as `'noise'` or a positive semidefinite matrix of size equal the number of outputs.
- **Maximum number of iterations** — Maximum number of iterations to use during search.
- **Termination tolerance** — Tolerance value when the iterations should terminate.
- **Error threshold for outlier penalty** — Robustification of the quadratic criterion of fit.

9 Click **Estimate** to estimate the model. A new model gets added to the System Identification app.

Next Steps

- Validate the model by selecting the appropriate check box in the **Model Views** area of the System Identification app. For more information about validating models, see “Validating Models After Estimation” on page 17-3.
- Export the model to the MATLAB workspace for further analysis. Drag the model to the **To Workspace** rectangle in the System Identification app.

Estimate Transfer Function Models at the Command Line

This topic shows how to estimate transfer function models at the command line.

Before you estimate a transfer function model, you must have:

- Input/Output data. See "Representing Time- and Frequency-Domain Data Using iddata Objects" on page 2-50. For supported data formats, see "Data Supported by Transfer Function Models" on page 8-5.
- Performed any required data preprocessing operations. You can detrend your data before estimation. For more information, see "Ways to Prepare Data for System Identification" on page 2-6.

  Alternatively, you can specify the input/output offset for the data using an estimation option set. Use `tfestOptions` to create the estimation option set. Use the `InputOffset` and `OutputOffset` name and value pairs to specify the input/output offset.

Estimate continuous-time and discrete-time transfer function models using `tfest`. The output of `tfest` is an `idtf` object, which represents the identified transfer function.

The general workflow in estimating a transfer function model is:

1. Create a data object (`iddata` or `idfrd`) that captures the experimental data.
2. (Optional) Specify estimation options using `tfestOptions`.
3. (Optional) Create a transfer function model that specifies the expected model structure and any constraints on the estimation parameters.
4. Use `tfest` to identify the transfer function model, based on the data.
5. Validate the model. See "Model Validation".

See Also

Related Examples

- "Troubleshoot Frequency-Domain Identification of Transfer Function Models" on page 8-23

Transfer Function Structure Specification

You can use a priori knowledge of the expected transfer function model structure to initialize the estimation. The `Structure` property of an `idtf` model contains parameters that allow you to specify the values and constraints for the numerator, denominator and transport delays.

For example, specify a third-order transfer function model that contains an integrator and has a transport delay of at most 1.5 seconds:

```
init_sys = idtf([nan nan],[1 2 1 0]);
init_sys.Structure.IODelay.Maximum = 1.5;
init_sys.Structure.Denominator.Free(end) = false;
```

`int_sys` is an `idtf` model with three poles and one zero. The denominator coefficient for the `s^0` term is zero and implies that one of the poles is an integrator.

`init_sys.Structure.IODelay.Maximum = 1.5` constrains the transport delay to a maximum of 1.5 seconds. The last element of the denominator coefficients (associated with the `s^0` term) is not a free estimation variable. This constraint forces one of the estimated poles to be at `s = 0`.

For more information regarding configuring the initial parameterization of an estimated transfer function, see `Structure` in `idtf`.

Estimate Transfer Function Models by Specifying Number of Poles

This example shows how to identify a transfer function containing a specified number of poles for given data.

Load time-domain system response data and use it to estimate a transfer function for the system.

```
load iddata1 z1;
np = 2;
sys = tfest(z1,np);
```

`z1` is an `iddata` object that contains time-domain, input-output data.

`np` specifies the number of poles in the estimated transfer function.

`sys` is an `idtf` model containing the estimated transfer function.

To see the numerator and denominator coefficients of the resulting estimated model `sys`, enter:

```
sys.Numerator;
sys.Denominator;
```

To view the uncertainty in the estimates of the numerator and denominator and other information, use `tfdata`.

Estimate Transfer Function Models with Transport Delay to Fit Given Frequency-Response Data

This example shows how to identify a transfer function to fit a given frequency response data (FRD) containing additional phase roll off induced by input delay.

This example requires a Control System Toolbox™ license.

Obtain frequency response data.

For this example, use `bode` to obtain the magnitude and phase response data for the following system:

$$H(s) = e^{-5s}\frac{s+0.2}{s^3+2s^2+s+1}$$

Use 100 frequency points, ranging from 0.1 rad/s to 10 rad/s, to obtain the frequency response data. Use `frd` to create a frequency-response data object.

```
freq = logspace(-1,1,100);
[mag, phase] = bode(tf([1 .2],[1 2 1 1],'InputDelay',.5),freq);
data = frd(mag.*exp(1j*phase*pi/180),freq);
```

`data` is an `iddata` object that contains frequency response data for the described system.

Estimate a transfer function using `data`. Specify an unknown transport delay for the identified transfer function.

```
np = 3;
nz = 1;
iodelay = NaN;
sys = tfest(data,np,nz,iodelay);
```

`np` and `nz` specify the number of poles and zeros in the identified transfer function, respectively.

`iodelay` specifies an unknown transport delay for the identified transfer function.

`sys` is an `idtf` model containing the identified transfer function.

Estimate Transfer Function Models With Prior Knowledge of Model Structure and Constraints

This example shows how to estimate a transfer function model when the structure of the expected model is known and apply constraints to the numerator and denominator coefficients.

Load time-domain data.

```
load iddata1 z1;
z1.y = cumsum(z1.y);
```

`cumsum` integrates the output data of `z1`. The estimated transfer function should therefore contain an integrator.

Create a transfer function model with the expected structure.

```
init_sys = idtf([100 1500],[1 10 10 0]);
```

`int_sys` is an `idtf` model with three poles and one zero. The denominator coefficient for the `s^0` term is zero which indicates that `int_sys` contains an integrator.

Specify constraints on the numerator and denominator coefficients of the transfer function model. To do so, configure fields in the `Structure` property:

```
init_sys.Structure.Numerator.Minimum = eps;
init_sys.Structure.Denominator.Minimum = eps;
init_sys.Structure.Denominator.Free(end) = false;
```

The constraints specify that the numerator and denominator coefficients are nonnegative. Additionally, the last element of the denominator coefficients (associated with the `s^0` term) is not an estimable parameter. This constraint forces one of the estimated poles to be at `s = 0`.

Create an estimation option set that specifies using the Levenberg–Marquardt search method.

```
opt = tfestOptions('SearchMethod','lm');
```

Estimate a transfer function for `z1` using `init_sys` and the estimation option set.

```
sys = tfest(z1,init_sys,opt);
```

`tfest` uses the coefficients of `init_sys` to initialize the estimation of `sys`. Additionally, the estimation is constrained by the constraints you specify in the `Structure` property of `init_sys`. The resulting `idtf` model `sys` contains the parameter values that result from the estimation.

Estimate Transfer Function Models with Unknown Transport Delays

This example shows how to estimate a transfer function model with unknown transport delays and apply an upper bound on the unknown transport delays.

Create a transfer function model with the expected numerator and denominator structure and delay constraints.

For this example, the experiment data consists of two inputs and one output. Both transport delays are unknown and have an identical upper bound. Additionally, the transfer functions from both inputs to the output are identical in structure.

```
init_sys = idtf(NaN(1,2),[1, NaN(1,3)],'IODelay',NaN);
init_sys.Structure(1).IODelay.Free = true;
init_sys.Structure(1).IODelay.Maximum = 7;
```

`init_sys` is an `idtf` model describing the structure of the transfer function from one input to the output. The transfer function consists of one zero, three poles and a transport delay. `NaN` indicates unknown coefficients.

`init_sys.Structure(1).IODelay.Free = true` indicates that the transport delay is not fixed.

`init_sys.Structure(1).IODelay.Maximum = 7` sets the upper bound for the transport delay to 7 seconds.

Specify the transfer function from both inputs to the output.

```
init_sys = [init_sys,init_sys];
```

Load time-domain system response data and detrend the data.

```
load co2data;
Ts = 0.5;
data = iddata(Output_exp1,Input_exp1,Ts);
T = getTrend(data);
T.InputOffset = [170,50];
T.OutputOffset = mean(data.y(1:75));
data = detrend(data, T);
```

Identify a transfer function model for the measured data using the specified delay constraints.

```
sys = tfest(data,init_sys);
```

`sys` is an `idtf` model containing the identified transfer function.

Estimate Transfer Functions with Delays

This example shows how to estimate transfer function models with I/O delays.

The `tfest` command supports estimation of IO delays. In the simplest case, if you specify `NaN` as the value for the `IODelay` input argument, `tfest` estimates the corresponding delay value.

```
load iddata1 z1
sys = tfest(z1,2,2,NaN); % 2 poles, 2 zeros, unknown transport delay
```

If you want to assign an initial guess to the value of delay or prescribe bounds for its value, you must first create a template `idtf` model and configure `IODelay` using the model's `Structure` property:

```
sys0 = idtf([nan nan nan],[1 nan nan]);
sys0.Structure.IODelay.Value = 0.1; % initial guess
sys0.Structure.IODelay.Maximum = 1; % maximum allowable value for delay
sys0.Structure.IODelay.Free = true; % treat delay as estimatable quantity
sys = tfest(z1,sys0);
```

If estimation data is in the time-domain, the delays are not estimated iteratively. If a finite initial value is specified, that value is retained as is with no iterative updates. The same is true of discrete-time frequency domain data. Thus in the example above, if `data` has a nonzero sample time, the estimated value of delay in the returned model `sys` is 0.1 (same as the initial guess specified for `sys0`). The delays are updated iteratively only for continuos-time frequency domain data. If, on the other hand, a finite initial value for delay is not specified (e.g., `sys0.Structure.IODelay.Value = NaN`), then a value for delay is determined using the `delayest` function, regardless of the nature of the data.

Determination of delay as a quantity independent of the model's poles and zeros is a difficult task. Estimation of delays becomes especially difficult for multi-input or multi-output data. It is strongly recommended that you perform some investigation to determine delays before estimation. You can use functions such as `delayest`, `arxstruc`, `selstruc` and impulse response analysis to determine delays. Often, physical knowledge of the system or dedicated transient tests (how long does it take for a step change in input to show up in a measured output?) will reveal the value of transport delays. Use the results of such analysis to assign initial guesses as well as minimum and maximum bounds on the estimated values of delays.

Specifying Initial Conditions for Iterative Estimation Algorithms

If you estimate transfer function models using `tfest`, you can specify how the algorithm treats initial conditions.

In the System Identification app, set **Initial condition** to one of the following options:

- `auto` — Automatically chooses `Zero`, `Estimate`, or `Backcast` based on the estimation data. If initial conditions have negligible effect on the prediction errors, the initial conditions are set to zero to optimize algorithm performance.
- `Zero` — Sets all initial conditions to zero.
- `Estimate` — Treats the initial conditions as an estimation parameters.
- `Backcast` — Estimates initial conditions using a backward filtering method (least-squares fit).

At the command line, specify the initial conditions by using an estimation option set. Use `tfestOptions` to create the estimation option set. For example, create an options set that sets the initial conditions to zero:

```
opt = tfestOptions('InitialCondition','zero');
```

See Also

`tfest` | `tfestOptions`

More About

- “Estimate Transfer Function Models in the System Identification App” on page 8-6
- “Estimate Transfer Function Models at the Command Line” on page 8-13

Troubleshoot Frequency-Domain Identification of Transfer Function Models

This example shows how to perform and troubleshoot the identification of a SISO system using frequency-response data (FRD). The techniques explained here can also be applied to MIMO models and frequency-domain input-output data.

When you use the `tfest` command to estimate a SISO transfer function model from the frequency-response data, the estimation algorithm minimizes the following least-squares loss (cost) function:

$$\underset{G(\omega)}{\text{minimize}} \sum_{k=1}^{N_f} \left| W(\omega_k) \left(G(\omega_k) - f(\omega_k) \right) \right|^2$$

Here `W` is a frequency-dependent weight that you specify, `G` is the transfer function that is to be estimated, `f` is the measured frequency-response data, and w is the frequency. `Nf` is the number frequencies at which the data is available. $G(w) - f(w)$ is the frequency-response error.

In this example, you first estimate the model without preprocessing the data or using estimation options to specify a weighting filter. You then apply these troubleshooting techniques to improve the model estimation.

Estimate the Model Without Preprocessing and Filtering

Load the measured continuous-time frequency response data.

```
load troubleshooting_example_data Gfrd;
```

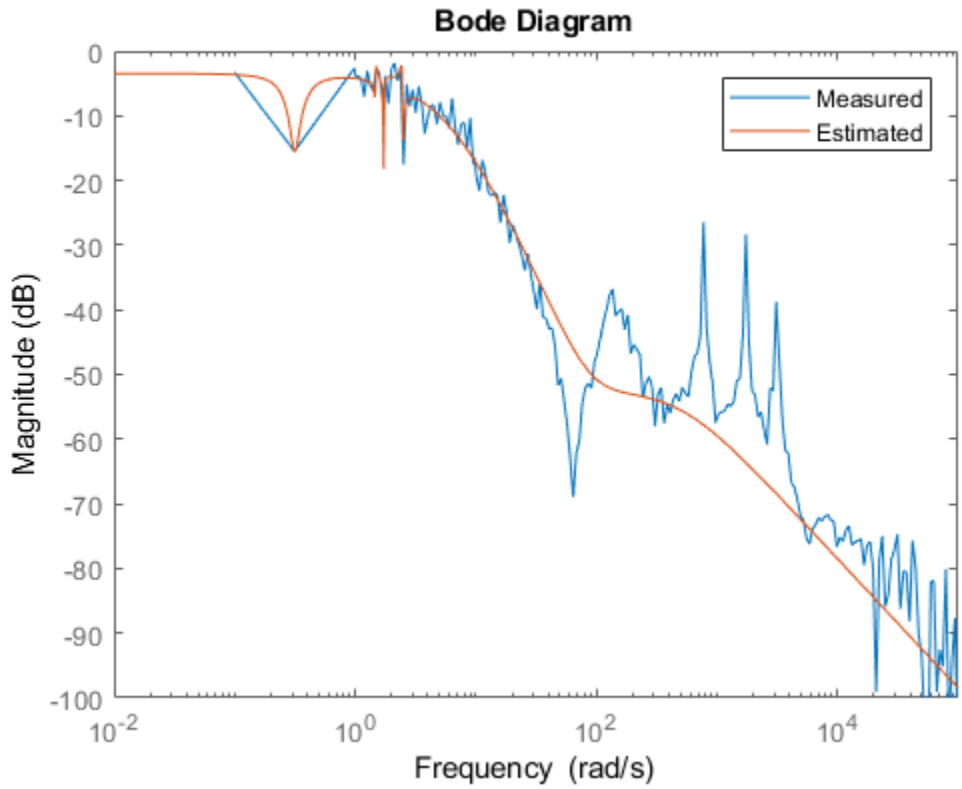
`Gfrd` is an `idfrd` object that stores the data.

Estimate an initial transfer function model with 11 poles and 10 zeros by using the estimation data.

```
Gfit = tfest(Gfrd,11,10);
```

Plot the frequency-response magnitude of the estimated model and the measured frequency-response data.

```
bodemag(Gfrd,Gfit);
ylim([-100 0])
legend('Measured','Estimated')
```

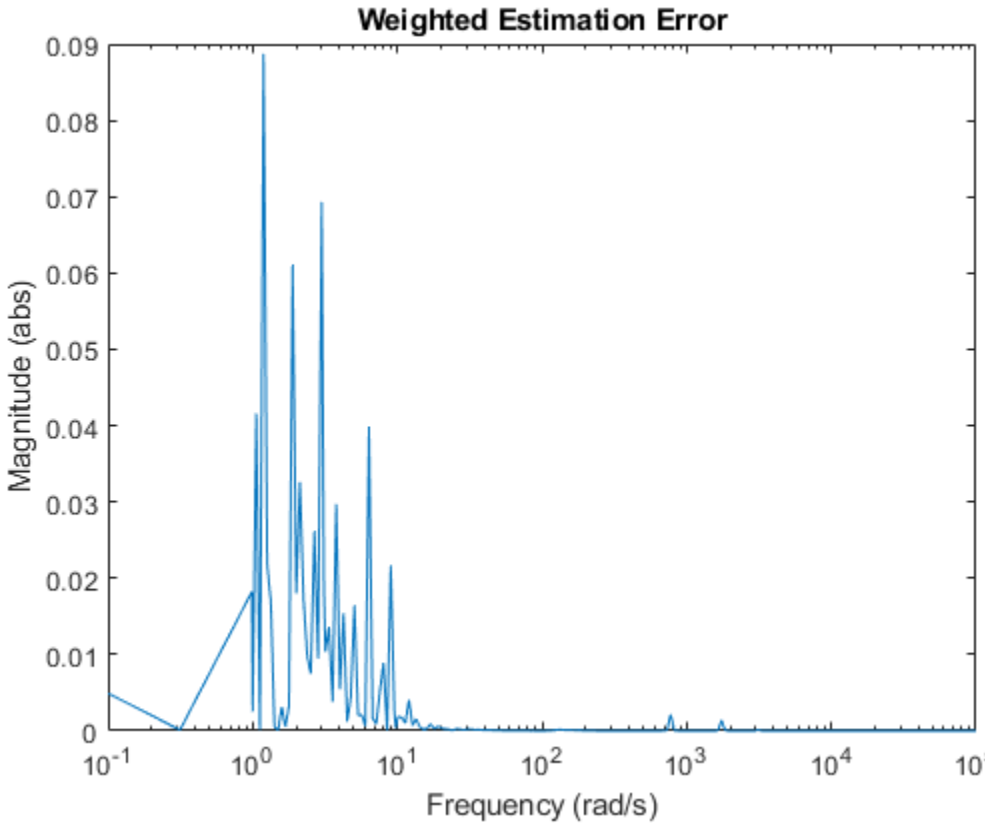


The estimated model contains spurious dynamics. The estimation algorithm misses the valley at 60 rad/s and the resonant peaks after that. Therefore, the model is not a good fit to the data.

The algorithm minimizes the squared error magnitude, $|W(w)\,(G(w)-f(w))|^2$, in the loss function. Plot this quantity as a function of frequency. This error plot provides a view of which data points contribute the most to the loss function, and so are the likely limiting factors during estimation. The plot can help you identify why there are spurious or uncaptured dynamics.

Because you have not specified a frequency-dependent weight, $W(w)$ is 1.

```
w = Gfrd.Frequency;
r1 = squeeze(freqresp(Gfit,w));
r2 = squeeze(freqresp(Gfrd,w));
fitError = r1-r2;
semilogx(w,abs(fitError).^2)
title('Weighted Estimation Error');
xlabel('Frequency (rad/s)');
ylabel('Magnitude (abs)')
```



From the data, model, and error plots you can see that:

- The largest fitting errors are below 10 rad/s.

- The algorithm focusses on fitting the noisy high magnitude data points below 10 rad/s, which have a large contribution to the optimization loss function. As a result, the algorithm assigns spurious poles and zeros to this data region. To address this issue, you can preprocess the data to improve signal-to-noise ratio in this region. You can also use frequency-dependent weights to make the algorithm put less focus on this region.
- Below approximately 40 rad/s, most variations in data are due to noise. There are no significant system modes (valleys or peaks) in the data. To address this issue, you can use a moving-average filter over the data to smooth the measured response.
- The algorithm ignores the valley around 60 rad/s and the three lightly damped resonant peaks that follow it. These features contribute little to the loss function because the fitting error is small at these frequencies. To address this issue, you can specify frequency-dependent weights to make the algorithm pay more attention to these frequencies.

Preprocess Data

To improve the estimated model quality, preprocess the data. To do so, you truncate the low signal-to-noise portions of data below 1 rad/s and above 2e4 rad/s that are not interesting. Then you use a moving-average filter to smooth data in the low-frequency high-magnitude region below 40 rad/s. At these frequencies, the data has a low signal-to-noise ratio, but has dynamics that you are interested in capturing. Do not apply the filter at frequencies above 40 rad/s to avoid smoothing data where you see the valley and the three peaks that follow it.

Make a copy of the original `idfrd` data object.

```
GfrdProcessed = Gfrd;
```

Truncate the data below 1 rad/s and above 2e4 rad/s.

```
GfrdProcessed = fselect(GfrdProcessed,1,2e4);
```

Apply a three-point centered moving-average filter to smooth out the frequency-response data below 40 rad/s that contains spurious dynamics. The response data is stored in the `ResponseData` property of the object.

```
w = GfrdProcessed.Frequency;
f = squeeze(GfrdProcessed.ResponseData);
idx = w<40;
f(idx) = movmean(f(idx),3);
```

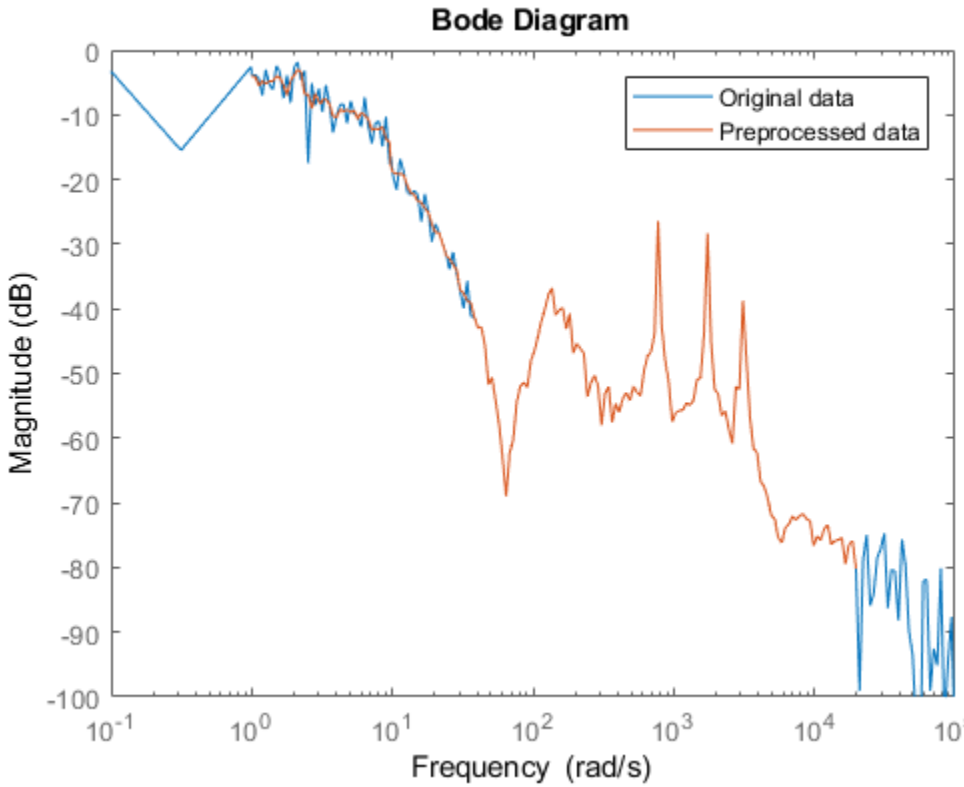
Here `f(idx)` is the frequency-response data at frequencies less than 40 rad/s.

Place the filtered data back into the original data object.

```
GfrdProcessed.ResponseData = f;
```

Plot the original and preprocessed data.

```
bodemag(Gfrd,GfrdProcessed);
ylim([-100 0]);
legend('Original data','Preprocessed data');
```



The plot shows that all desired dynamics are intact after preprocessing.

Specify Weighting Filter

Use a low weight for the low frequency region under 10 rad/s where spurious dynamics exist. This low weight and the smoothing applied earlier to this data reduce the chance of spurious peaks in the estimated model response in this region.

```
Weight = ones(size(f));
idx = w<10;
Weight(idx) = Weight(idx)/10;
```

Use a high weight for data in the frequency range 40-6e3 rad/s where you want to capture the dynamics but the response data magnitude is low.

```
idx = w>40 & w<6e3;
Weight(idx) = Weight(idx)*30;
```

Specify the weights in the `WeightingFilter` option of the estimation option set.

```
tfestOpt = tfestOptions('WeightingFilter',Weight);
```

Note that `Weight` is a custom weighting filter. You can also specify `WeightingFilter` as `'inv'` or `'invsqrt'` for frequency-response data. These options specify the weight as $1/|f(w)|$ and $1/\sqrt{|f(w)|}$, respectively. These options enable you to quickly test the effect of using a higher weight for low magnitude regions of data. `'invsqrt'` is typically a good initial choice. If these weights do not yield good estimation results, you can provide custom weights as shown in this example.

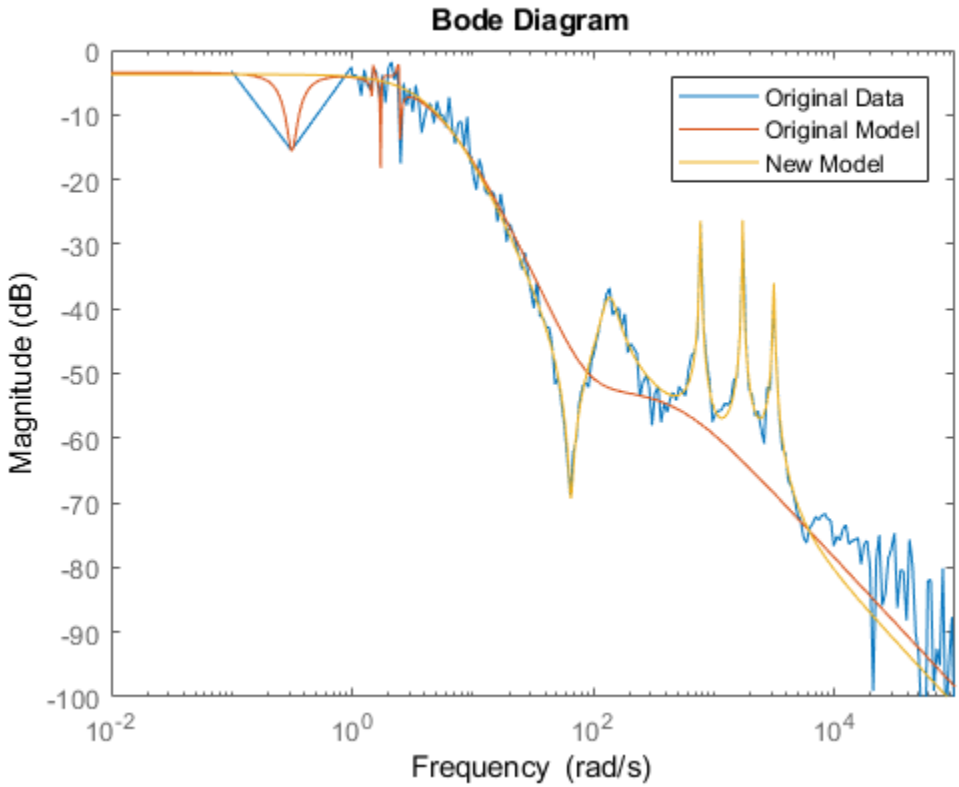
Estimate Model Using Preprocessed and Filtered Data

Estimate a transfer function model with 11 poles and 10 zeros using the preprocessed data and specified weighting filter.

```
GfitNew = tfest(GfrdProcessed,11,10,tfestOpt);
```

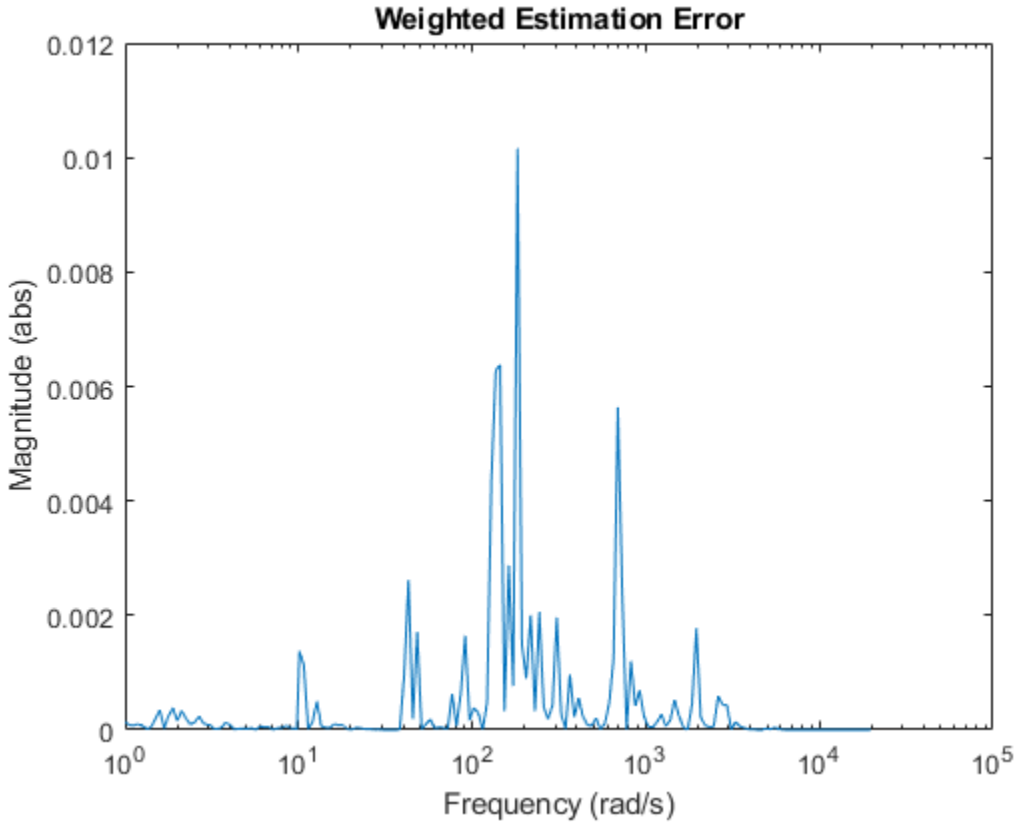
Plot the original data, the initial model response, and the new model response.

```
bodemag(Gfrd,Gfit,GfitNew);
ylim([-100 0]);
legend('Original Data','Original Model','New Model');
```



Plot the estimation error. Compute the estimation error by including the weighting filter `Weight` that you used for estimating `GfitNew`.

```
w = GfrdProcessed.Frequency;
r1 = squeeze(freqresp(GfitNew,w));
r2 = squeeze(freqresp(GfrdProcessed,w));
fitErrorNew = Weight.*(r1-r2);
semilogx(w,abs(fitErrorNew).^2)
title('Weighted Estimation Error');
xlabel('Frequency (rad/s)');
ylabel('Magnitude (abs)');
```



The new model successfully captures all system dynamics of interest.

You can use the weighted error plot for further troubleshooting if your initial choice of weights does not yield a satisfactory result.

See Also

`tfest` | `tfestOptions`

More About

- "Estimating Models Using Frequency-Domain Data"

- “Estimate Transfer Function Models at the Command Line” on page 8-13

Identifying Frequency-Response Models

- "What is a Frequency-Response Model?" on page 9-2
- "Data Supported by Frequency-Response Models" on page 9-4
- "Estimate Frequency-Response Models in the App" on page 9-5
- "Estimate Frequency-Response Models at the Command Line" on page 9-7
- "Selecting the Method for Computing Spectral Models" on page 9-9
- "Controlling Frequency Resolution of Spectral Models" on page 9-11
- "Spectrum Normalization" on page 9-14

What is a Frequency-Response Model?

A *frequency-response model* is the frequency response of a linear system evaluated over a range of frequency values. The model is represented by an `idfrd` model object that stores the frequency response, sample time, and input-output channel information.

The frequency-response function describes the steady-state response of a system to sinusoidal inputs. For a linear system, a sinusoidal input of a specific frequency results in an output that is also a sinusoid with the same frequency, but with a different amplitude and phase. The frequency-response function describes the amplitude change and phase shift as a function of frequency.

You can estimate frequency-response models and visualize the responses on a Bode plot, which shows the amplitude change and the phase shift as a function of the sinusoid frequency.

For a discrete-time system sampled with a time interval T, the transfer function $G(z)$ relates the Z-transforms of the input $U(z)$ and output $Y(z)$:

$$Y(z) = G(z)U(z) + H(z)E(z)$$

The frequency-response is the value of the transfer function, $G(z)$, evaluated on the unit circle ($z = \exp^{iwT}$) for a vector of frequencies, w. $H(z)$ represents the noise transfer function, and $E(z)$ is the Z-transform of the additive disturbance $e(t)$ with variance λ. The values of G are stored in the `ResponseData` property of the `idfrd` object. The noise spectrum is stored in the `SpectrumData property`.

Where, the noise spectrum is defined as:

$$\Phi_v(\omega) = \lambda T \left| H(e^{i\omega T}) \right|^2$$

A MIMO frequency-response model contains frequency-responses corresponding to each input-output pair in the system. For example, for a two-input, two-output model:

$$Y_1(z) = G_{11}(z)U_1(z) + G_{12}(z)U_2(z) + H_1(z)E_1(z)$$
$$Y_2(z) = G_{21}(z)U_1(z) + G_{22}(z)U_2(z) + H_2(z)E_2(z)$$

Where, G_{ij} is the transfer function between the i^{th} output and the j^{th} input. $H_1(z)$ and $H_2(z)$ represent the noise transfer functions for the two outputs. $E_1(z)$ and $E_2(z)$ are the Z-transforms of the additive disturbances, $e_1(t)$ and $e_2(t)$, at the two model outputs, respectively.

Similar expressions apply for continuous-time frequency response. The equations are represented in Laplace domain. For more details, see the `idfrd` reference page.

See Also

Related Examples

- “Estimate Frequency-Response Models in the App” on page 9-5
- “Estimate Frequency-Response Models at the Command Line” on page 9-7

More About

- “Data Supported by Frequency-Response Models” on page 9-4

Data Supported by Frequency-Response Models

You can estimate spectral analysis models from data with the following characteristics:

- Complex or real data.
- Time- or frequency-domain `iddata` or `idfrd` data object. To learn more about estimating time-series models, see “Time Series Analysis”.
- Single- or multiple-output data.

See Also

More About

- “What is a Frequency-Response Model?” on page 9-2

Estimate Frequency-Response Models in the App

You must have already imported your data into the app and performed any necessary preprocessing operations. For more information, see "Data Preparation".

To estimate frequency-response models in the System Identification app:

1. In the System Identification app, select **Estimate > Spectral models** to open the Spectral Model dialog box.
2. In the **Method** list, select the spectral analysis method you want to use. For information about each method, see "Selecting the Method for Computing Spectral Models" on page 9-9.
3. Specify the frequencies at which to compute the spectral model in *one* of the following ways:

   - In the **Frequencies** field, enter either a vector of values, a MATLAB expression that evaluates to a vector, or a variable name of a vector in the MATLAB workspace. For example, `logspace(-1,2,500)`.
   - Use the combination of **Frequency Spacing** and **Frequencies** to construct the frequency vector of values:

     - In the **Frequency Spacing** list, select `Linear` or `Logarithmic` frequency spacing.

     Note For `etfe`, only the `Linear` option is available.

     - In the **Frequencies** field, enter the number of frequency points.

     For time-domain data, the frequency ranges from 0 to the Nyquist frequency. For frequency-domain data, the frequency ranges from the smallest to the largest frequency in the data set.
4. In the **Frequency Resolution** field, enter the frequency resolution, as described in "Controlling Frequency Resolution of Spectral Models" on page 9-11. To use the default value, enter `default` or, equivalently, the empty matrix `[]`.
5. In the **Model Name** field, enter the name of the correlation analysis model. The model name should be unique in the Model Board.
6. Click **Estimate** to add this model to the Model Board in the System Identification app.

7 In the Spectral Model dialog box, click **Close**.

8 To view the frequency-response plot, select the **Frequency resp** check box in the System Identification app. For more information about working with this plot, see “Frequency Response Plots” on page 17-64.

9 To view the estimated disturbance spectrum, select the **Noise spectrum** check box in the System Identification app. For more information about working with this plot, see “Noise Spectrum Plots” on page 17-73.

10 Validate the model after estimating it. For more information, see “Model Validation”.

To export the model to the MATLAB workspace, drag it to the **To Workspace** rectangle in the System Identification app. You can retrieve the responses from the resulting `idfrd` model object using the `bode` or `nyquist` command.

See Also

Related Examples

- “Estimate Frequency-Response Models at the Command Line” on page 9-7

More About

- “What is a Frequency-Response Model?” on page 9-2
- “Data Supported by Frequency-Response Models” on page 9-4
- “Selecting the Method for Computing Spectral Models” on page 9-9
- “Controlling Frequency Resolution of Spectral Models” on page 9-11

Estimate Frequency-Response Models at the Command Line

You can use the `etfe`, `spa`, and `spafdr` commands to estimate spectral models. The following table provides a brief description of each command and usage examples.

The resulting models are stored as `idfrd` model objects. For detailed information about the commands and their arguments, see the corresponding reference page.

Commands for Frequency Response

Command	Description	Usage
`etfe`	Estimates an empirical transfer function using Fourier analysis.	To estimate a model `m`, use the following syntax: `m=etfe(data)`
`spa`	Estimates a frequency response with a fixed frequency resolution using spectral analysis.	To estimate a model `m`, use the following syntax: `m=spa(data)`
`spafdr`	Estimates a frequency response with a variable frequency resolution using spectral analysis.	To estimate a model `m`, use the following syntax: `m=spafdr(data,R,w)` where `R` is the resolution vector and `w` is the frequency vector.

Validate the model after estimating it. For more information, see “Model Validation”.

See Also

Related Examples

- “Estimate Frequency-Response Models in the App” on page 9-5

More About

- “What is a Frequency-Response Model?” on page 9-2
- “Data Supported by Frequency-Response Models” on page 9-4
- “Selecting the Method for Computing Spectral Models” on page 9-9

- “Controlling Frequency Resolution of Spectral Models” on page 9-11

Selecting the Method for Computing Spectral Models

This section describes how to select the method for computing spectral models in the estimation procedures “Estimate Frequency-Response Models in the App” on page 9-5 and “Estimate Frequency-Response Models at the Command Line” on page 9-7.

You can choose from the following three spectral-analysis methods:

- `etfe` (**E**mpirical **T**ransfer **F**unction **E**stimate)

  For input-output data — This method computes the ratio of the Fourier transform of the output to the Fourier transform of the input.

  For time-series data — This method computes a periodogram as the normalized absolute squares of the Fourier transform of the time series.

  ETFE works well for highly resonant systems or narrowband systems. The drawback of this method is that it requires linearly spaced frequency values, does not estimate the disturbance spectrum, and does not provide confidence intervals. ETFE also works well for periodic inputs and computes exact estimates at multiples of the fundamental frequency of the input and their ratio.

- `spa` (**SP**ectral **A**nalysis)

  This method is the Blackman-Tukey spectral analysis method, where windowed versions of the covariance functions are Fourier transformed.

- `spafdr` (**SP**ectral **A**nalysis with **F**requency **D**ependent **R**esolution)

  This method is a variant of the Blackman-Tukey spectral analysis method with frequency-dependent resolution. First, the algorithm computes Fourier transforms of the inputs and outputs. Next, the products of the transformed inputs and outputs with the conjugate input transform are smoothed over local frequency regions. The widths of the local frequency regions can vary as a function of frequency. The ratio of these averages computes the frequency-response estimate.

See Also

Related Examples

- “Estimate Frequency-Response Models in the App” on page 9-5
- “Estimate Frequency-Response Models at the Command Line” on page 9-7

Controlling Frequency Resolution of Spectral Models

In this section...
“What Is Frequency Resolution?” on page 9-11
“Frequency Resolution for etfe and spa” on page 9-11
“Frequency Resolution for spafdr” on page 9-12
“etfe Frequency Resolution for Periodic Input” on page 9-12

This section supports the estimation procedures “Estimate Frequency-Response Models in the App” on page 9-5 and “Estimate Frequency-Response Models at the Command Line” on page 9-7.

What Is Frequency Resolution?

Frequency resolution is the size of the smallest frequency for which details in the frequency response and the spectrum can be resolved by the estimate. A resolution of 0.1 rad/s means that the frequency response variations at frequency intervals at or below 0.1 rad/s are not resolved.

Note Finer resolution results in greater uncertainty in the model estimate.

Specifying the frequency resolution for `etfe` and `spa` is different than for `spafdr`.

Frequency Resolution for etfe and spa

For `etfe` and `spa`, the frequency resolution is approximately equal to the following value:

$$\frac{2\pi}{M}\left(\frac{\text{radians}}{\text{sampling interval}}\right)$$

M is a scalar integer that sets the size of the lag window. The value of `M` controls the trade-off between bias and variance in the spectral estimate.

The default value of M for `spa` is good for systems without sharp resonances. For `etfe`, the default value of M gives the maximum resolution.

A large value of *M* gives good resolution, but results in more uncertain estimates. If a true frequency function has sharp peak, you should specify higher `M` values.

Frequency Resolution for spafdr

In case of `etfe` and `spa`, the frequency response is defined over a uniform frequency range, *0-F_s/2* radians per second, where F_s is the sampling frequency—equal to twice the Nyquist frequency. In contrast, `spafdr` lets you increase the resolution in a specific frequency range, such as near a resonance frequency. Conversely, you can make the frequency grid coarser in the region where the noise dominates—at higher frequencies, for example. Such customizing of the frequency grid assists in the estimation process by achieving high fidelity in the frequency range of interest.

For `spafdr`, the frequency resolution around the frequency *k* is the value *R(k)*. You can enter *R(k)* in any *one* of the following ways:

- Scalar value of the constant frequency resolution value in radians per second.

  Note The scalar *R* is inversely related to the *M* value used for `etfe` and `spa`.

- Vector of frequency values the same size as the frequency vector.
- Expression using MATLAB workspace variables and evaluates to a resolution vector that is the same size as the frequency vector.

The default value of the resolution for `spafdr` is twice the difference between neighboring frequencies in the frequency vector.

etfe Frequency Resolution for Periodic Input

If the input data is marked as periodic and contains an integer number of periods (`data.Period` is an integer), `etfe` computes the frequency response at frequencies

$$\frac{2\pi k}{T}\left(\frac{\mathrm{k}}{\mathrm{Period}}\right) \quad \text{where } k = 1,2,\ldots,\mathrm{Period}\ .$$

For periodic data, the frequency resolution is ignored.

See Also

`etfe` | `spa` | `spafdr`

Related Examples

- “Estimate Frequency-Response Models in the App” on page 9-5
- “Estimate Frequency-Response Models at the Command Line” on page 9-7

Spectrum Normalization

The *spectrum* of a signal is the square of the Fourier transform of the signal. The spectral estimate using the commands `spa`, `spafdr`, and `etfe` is normalized by the sample time T:

$$\Phi_y(\omega) = T \sum_{k=-M}^{M} R_y(kT) e^{-i\omega T} W_M(k)$$

where $W_M(k)$ is the lag window, and M is the width of the lag window. The output covariance $R_y(kT)$ is given by the following discrete representation:

$$\hat{R}_y(kT) = \frac{1}{N} \sum_{l=1}^{N} y(lT - kT) y(lT)$$

Because there is no scaling in a discrete Fourier transform of a vector, the purpose of T is to relate the discrete transform of a vector to the physically meaningful transform of the measured signal. This normalization sets the units of $\Phi_y(\omega)$ as power per radians per unit time, and makes the frequency units radians per unit time.

The scaling factor of T is necessary to preserve the energy density of the spectrum after interpolation or decimation.

By Parseval's theorem, the average energy of the signal must equal the average energy in the estimated spectrum, as follows:

$$Ey^2(t) = \frac{1}{2\pi} \int_{-\pi/T}^{\pi/T} \Phi_y(\omega) d\omega$$

$$S1 \equiv Ey^2(t)$$

$$S2 \equiv \frac{1}{2\pi} \int_{-\pi/T}^{\pi/T} \Phi_y(\omega) d\omega$$

To compare the left side of the equation (`S1`) to the right side (`S2`), enter the following commands. In this code, `phiy` contains $\Phi_y(\omega)$ between $\omega = 0$ and $\omega = \pi/T$ with the frequency step given as follows:

$$\left(\frac{\pi}{T \cdot \text{length(phiy)}} \right)$$

```
load iddata1
```

Create a time-series iddata object.

```
y = z1(:,1,[]);
```

Define sample interval from the data.

```
T = y.Ts;
```

Estimate the frequency response.

```
sp = spa(y);
```

Remove spurious dimensions

```
phiy = squeeze(sp.spec);
```

Compute average energy of the signal.

```
S1 = sum(y.y.^2)/size(y,1)
```

```
S1 = 19.4646
```

Compute average energy from the estimated energy spectrum, where S2 is scaled by T.

```
S2 = sum(phiy)/length(phiy)/T
```

```
S2 = 19.2076
```

Thus, the average energy of the signal approximately equals the average energy in the estimated spectrum.

See Also

`etfe` | `spa` | `spafdr`

Identifying Impulse-Response Models

- "What Is Time-Domain Correlation Analysis?" on page 10-2
- "Data Supported by Correlation Analysis" on page 10-3
- "Estimate Impulse-Response Models Using System Identification App" on page 10-4
- "Estimate Impulse-Response Models at the Command Line" on page 10-6
- "Compute Response Values" on page 10-8
- "Identify Delay Using Transient-Response Plots" on page 10-9
- "Correlation Analysis Algorithm" on page 10-12

What Is Time-Domain Correlation Analysis?

Time-domain correlation analysis refers to non-parametric estimation of the impulse response of dynamic systems as a finite impulse response (FIR) model from the data. The estimated model is stored as transfer function model object (`idtf`). For information about transfer function models, see “What are Transfer Function Models?” on page 8-2.

Correlation analysis assumes a linear system and does not require a specific model structure.

Impulse response is the output signal that results when the input is an impulse and has the following definition for a discrete model:

$u(t) = 0 \quad t > 0$

$u(t) = 1 \quad t = 0$

The response to an input *u(t)* is equal to the convolution of the impulse response, as follows:

$$y(t) = \int_0^t h(t - z) \cdot u(z) dz$$

See Also

Related Examples

- “Estimate Impulse-Response Models Using System Identification App” on page 10-4
- “Estimate Impulse-Response Models at the Command Line” on page 10-6

More About

- “Data Supported by Correlation Analysis” on page 10-3

Data Supported by Correlation Analysis

You can estimate impulse-response models from data with the following characteristics:

- Real or complex data.
- Single- or multiple-output data.
- Time- or frequency-domain data with nonzero sample time.

  Time-domain data must be regularly sampled. You cannot use time-series data for correlation analysis.

See Also

More About

- “What Is Time-Domain Correlation Analysis?” on page 10-2

Estimate Impulse-Response Models Using System Identification App

Before you can perform this task, you must have:

- Imported data into the System Identification app. See “Import Time-Domain Data into the App” on page 2-16. For supported data formats, see “Data Supported by Correlation Analysis” on page 10-3.
- Performed any required data preprocessing operations. To improve the accuracy of your model, you should detrend your data. See “Ways to Prepare Data for System Identification” on page 2-6.

To estimate in the System Identification app using time-domain correlation analysis:

1. In the System Identification app, select **Estimate > Correlation models** to open the Correlation Model dialog box.
2. In the **Time span (s)** field, specify a scalar value as the time interval over which the impulse or step response is calculated. For a scalar time span *T*, the resulting response is plotted from *-T/4* to *T*.

   Tip You can also enter a 2-D vector in the format `[min_value max_value]`.
3. In the **Order of whitening filter** field, specify the filter order.

   The prewhitening filter is determined by modeling the input as an autoregressive process of order *N*. The algorithm applies a filter of the form *A(q)u(t)=u_F(t)*. That is, the input *u(t)* is subjected to an FIR filter *A* to produce the filtered signal *u_F(t)*. *Prewhitening* the input by applying a whitening filter before estimation might improve the quality of the estimated impulse response *g*.

   The order of the prewhitening filter, *N*, is the order of the *A* filter. *N* equals the number of lags. The default value of *N* is `10`, which you can also specify as `[]`.
4. In the **Model Name** field, enter the name of the correlation analysis model. The name of the model should be unique in the Model Board.
5. Click **Estimate** to add this model to the Model Board in the System Identification app.
6. In the Correlation Model dialog box, click **Close**.

Next Steps

- Export the model to the MATLAB workspace for further analysis by dragging it to the **To Workspace** rectangle in the System Identification app.
- View the transient response plot by selecting the **Transient resp** check box in the System Identification app. For more information about working with this plot and selecting to view impulse- versus step-response, see “Impulse and Step Response Plots” on page 17-55.

See Also

Related Examples

- “Identify Delay Using Transient-Response Plots” on page 10-9
- “Estimate Impulse-Response Models at the Command Line” on page 10-6

More About

- “What Is Time-Domain Correlation Analysis?” on page 10-2
- “Data Supported by Correlation Analysis” on page 10-3
- “Correlation Analysis Algorithm” on page 10-12

Estimate Impulse-Response Models at the Command Line

Before you can perform this task, you must have:

- Input/output or frequency-response data. See “Representing Time- and Frequency-Domain Data Using iddata Objects” on page 2-50. For supported data formats, see “Data Supported by Correlation Analysis” on page 10-3.
- Performed any required data preprocessing operations. If you use time-domain data, you can detrend it before estimation. See “Ways to Prepare Data for System Identification” on page 2-6.

Use `impulseest` to compute impulse response models. `impulseest` estimates a high-order, noncausal FIR model using correlation analysis. The resulting models are stored as `idtf` model objects and contain impulse-response coefficients in the model numerator.

To estimate the model `m` and plot the impulse or step response, use the following syntax:

```
m=impulseest(data,N);
impulse(m,Time);
step(m,Time);
```

where `data` is a single- or multiple-output `iddata` or `idfrd` object. `N` is a scalar value specifying the order of the FIR system corresponding to the time range 0:Ts:(N-1)*Ts, where *Ts* is the data sample time.

You can also specify estimation options, such as regularizing kernel, pre-whitening filter order and data offsets, using `impulseestOptions` and pass them as an input to `impulseest`. For example:

```
opt = impulseestOptions('RegulKernel','TC'));
m = impulseest(data,N,opt);
```

To view the confidence region for the estimated response, use `impulseplot` and `stepplot` to create the plot. Then use `showConfidence`.

For example:

```
h = stepplot(m,Time);
showConfidence(h,3) % 3 std confidence region
```

Note `cra` is an alternative method for computing impulse response from time-domain data only.

Next Steps

- Perform model analysis. See “Validating Models After Estimation” on page 17-3.

See Also

Related Examples

- “Identify Delay Using Transient-Response Plots” on page 10-9
- “Compute Response Values” on page 10-8
- “Estimate Impulse-Response Models Using System Identification App” on page 10-4

More About

- “What Is Time-Domain Correlation Analysis?” on page 10-2
- “Data Supported by Correlation Analysis” on page 10-3
- “Correlation Analysis Algorithm” on page 10-12

Compute Response Values

You can use `impulse` and `step` commands with output arguments to get the numerical impulse- and step-response vectors as a function of time, respectively.

To get the numerical response values:

1. Compute the FIR model by using `impulseest`, as described in "Estimate Impulse-Response Models at the Command Line" on page 10-6.
2. Apply the following syntax on the resulting model:

```
% To compute impulse-response data
[y,t,~,ysd] = impulse(model)
% To compute step-response data
[y,t,~,ysd] = step(model)
```

   where `y` is the response data, `t` is the time vector, and `ysd` is the standard deviations of the response.

See Also

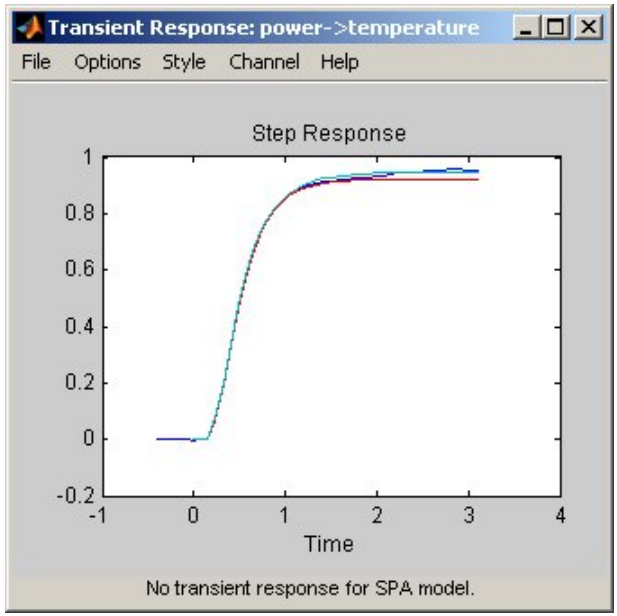
Related Examples

- "Estimate Impulse-Response Models at the Command Line" on page 10-6

Identify Delay Using Transient-Response Plots

You can use transient-response plots to estimate the input delay, or *dead time*, of linear systems. Input delay represents the time it takes for the output to respond to the input.

In the System Identification app: To view the transient response plot, select the **Transient resp** check box in the System Identification app. For example, the following step response plot shows a time delay of about 0.25 s before the system responds to the input.



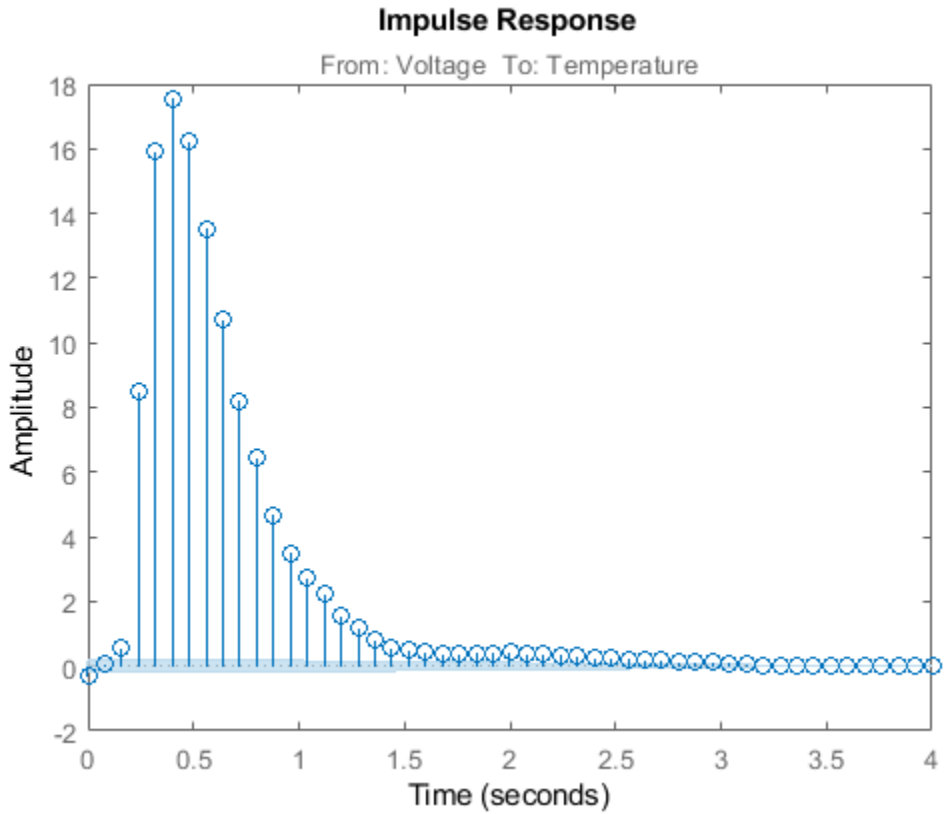
Step Response Plot

At the command line: You can use `impulseplot` to plot the impulse response. The time delay is equal to the first positive peak in the transient response magnitude that is greater than the confidence region for positive time values.

For example, the following commands create an impulse-response plot with a 1-standard-deviation confidence region:

```
load dry2
ze = dry2(1:500);
opt = impulseestOptions('RegulKernel','TC');
```

```
sys = impulseest(ze,40,opt);
h = impulseplot(sys);
showConfidence(h,1);
```



The resulting figure shows that the first positive peak of the response magnitude, which is greater than the confidence region for positive time values, occurs at 0.24 s.

Instead of using `showConfidence`, you can plot the confidence interval interactively, by right-clicking on the plot and selecting **Characteristics > Confidence Region**.

See Also

Related Examples

- “Estimate Impulse-Response Models Using System Identification App” on page 10-4
- “Estimate Impulse-Response Models at the Command Line” on page 10-6

Correlation Analysis Algorithm

Correlation analysis refers to methods that estimate the impulse response of a linear model, without specific assumptions about model orders.

The impulse response, g, is the system's output when the input is an impulse signal. The output response to a general input, $u(t)$, is obtained as the convolution with the impulse response. In continuous time:

$$y(t) = \int_{-\infty}^{t} g(\tau)u(t-\tau)d\tau$$

In discrete-time:

$$y(t) = \sum_{k=1}^{\infty} g(k)u(t-k)$$

The values of $g(k)$ are the discrete time impulse response coefficients.

You can estimate the values from observed input-output data in several different ways. `impulseest` estimates the first n coefficients using the least-squares method to obtain a finite impulse response (FIR) model of order n.

Several important options are associated with the estimate:

- **Prewhitening** — The input can be pre-whitened by applying an input-whitening filter of order `PW` to the data. This minimizes the effect of the neglected tail (`k > n`) of the impulse response.

  1. A filter of order `PW` is applied such that it whitens the input signal `u`:

     `1/A = A(u)e`, where `A` is a polynomial and `e` is white noise.
  2. The inputs and outputs are filtered using the filter:

     `uf = Au`, `yf = Ay`
  3. The filtered signals `uf` and `yf` are used for estimation.

  You can specify prewhitening using the `PW` name-value pair argument of `impulseestOptions`.
- **Regularization** — The least-squares estimate can be regularized. This means that a prior estimate of the decay and mutual correlation among `g(k)` is formed and used to

merge with the information about `g` from the observed data. This gives an estimate with less variance, at the price of some bias. You can choose one of the several kernels to encode the prior estimate.

This option is essential because, often, the model order `n` can be quite large. In cases where there is no regularization, `n` can be automatically decreased to secure a reasonable variance.

You can specify the regularizing kernel using the `RegulKernel` Name-Value pair argument of `impulseestOptions`.

- **Autoregressive Parameters** — The basic underlying FIR model can be complemented by `NA` autoregressive parameters, making it an ARX model.

$$y(t) = \sum_{k=1}^{n} g(k)u(t-k) - \sum_{k=1}^{NA} a_k y(t-k)$$

This gives both better results for small `n` and allows unbiased estimates when data are generated in closed loop. `impulseest` uses NA = 5 for t>0 and NA = 0 (no autoregressive component) for t<0.

- **Noncausal effects** — Response for negative lags. It may happen that the data has been generated partly by output feedback:

$$u(t) = \sum_{k=0}^{\infty} h(k)y(t-k) + r(t)$$

where $h(k)$ is the impulse response of the regulator and r is a setpoint or disturbance term. The existence and character of such feedback h can be estimated in the same way as g, simply by trading places between y and u in the estimation call. Using `impulseest` with an indication of negative delays,

`mi = impulseest(`$data, nk, nb$`)`, $nk < 0$, returns a model `mi` with an impulse response $[h(-nk), h(-nk-1), \ldots, h(0), g(1), g(2), \ldots, g(nb+nk)]$

aligned so that it corresponds to lags $[nk, nk+1, .., 0, 1, 2, \ldots, nb+nk]$. This is achieved because the input delay (`InputDelay`) of model `mi` is `nk`.

For a multi-input multi-output system, the impulse response $g(k)$ is an ny-by-nu matrix, where ny is the number of outputs and nu is the number of inputs. The i–j element of the matrix $g(k)$ describes the behavior of the ith output after an impulse in the jth input.

See Also

Related Examples

- “Estimate Impulse-Response Models Using System Identification App” on page 10-4
- “Estimate Impulse-Response Models at the Command Line” on page 10-6

Nonlinear Black-Box Model Identification

- "About Identified Nonlinear Models" on page 11-2
- "Nonlinear Model Structures" on page 11-7
- "Available Nonlinear Models" on page 11-12
- "Preparing Data for Nonlinear Identification" on page 11-15
- "What are Nonlinear ARX Models?" on page 11-17
- "Identifying Nonlinear ARX Models" on page 11-21
- "Available Nonlinearity Estimators for Nonlinear ARX Models" on page 11-27
- "Estimate Nonlinear ARX Models in the App" on page 11-29
- "Estimate Nonlinear ARX Models at the Command Line" on page 11-32
- "Estimate Nonlinear ARX Models Initialized Using Linear ARX Models" on page 11-44
- "Validate Nonlinear ARX Models" on page 11-49
- "Using Nonlinear ARX Models" on page 11-55
- "How the Software Computes Nonlinear ARX Model Output" on page 11-57
- "Linear Approximation of Nonlinear Black-Box Models" on page 11-64

About Identified Nonlinear Models

In this section...
“What Are Nonlinear Models?” on page 11-2 “When to Fit Nonlinear Models” on page 11-2 “Nonlinear Model Estimation” on page 11-4

What Are Nonlinear Models?

Dynamic models in System Identification Toolbox software are mathematical relationships between the inputs *u(t)* and outputs *y(t)* of a system. The model is *dynamic* because the output value at the current time depends on the input-output values at previous time instants. Therefore, dynamic models have memory of the past. You can use the input-output relationships to compute the current output from previous inputs and outputs. Dynamic models have states, where a state vector contains the information of the past.

The general form of a model in discrete time is:

$y(t) = f(u(t - 1), y(t - 1), u(t - 2), y(t - 2), \ldots)$

Such a model is nonlinear if the function *f* is a nonlinear function. *f* may represent arbitrary nonlinearities, such as switches and saturations.

The toolbox uses objects to represent various linear and nonlinear model structures. The nonlinear model objects are collectively known as *identified nonlinear models*. These models represent nonlinear systems with coefficients that are identified using measured input-output data. See “Nonlinear Model Structures” on page 11-7 for more information.

When to Fit Nonlinear Models

In practice, all systems are nonlinear and the output is a nonlinear function of the input variables. However, a linear model is often sufficient to accurately describe the system dynamics. In most cases, you should first try to fit linear models.

Here are some scenarios when you might need the additional flexibility of nonlinear models:

- "Linear Model Is Not Good Enough" on page 11-3
- "Physical System Is Weakly Nonlinear" on page 11-3
- "Physical System Is Inherently Nonlinear" on page 11-3
- "Linear and Nonlinear Dynamics Are Captured Separately" on page 11-4

Linear Model Is Not Good Enough

You might need nonlinear models when a linear model provides a poor fit to the measured output signals and cannot be improved by changing the model structure or order. Nonlinear models have more flexibility in capturing complex phenomena than the linear models of similar orders.

Physical System Is Weakly Nonlinear

From physical insight or data analysis, you might know that a system is weakly nonlinear. In such cases, you can estimate a linear model and then use this model as an initial model for nonlinear estimation. Nonlinear estimation can improve the fit by using nonlinear components of the model structure to capture the dynamics not explained by the linear model. For more information, see "Initialize Nonlinear ARX Estimation Using Linear Model" on page 11-25 and "Initialize Hammerstein-Wiener Estimation Using Linear Model" on page 12-8.

Physical System Is Inherently Nonlinear

You might have physical insight that your system is nonlinear. Certain phenomena are inherently nonlinear in nature, including dry friction in mechanical systems, actuator power saturation, and sensor nonlinearities in electromechanical systems. You can try modeling such systems using the Hammerstein-Wiener model structure, which lets you interconnect linear models with static nonlinearities. For more information, see "Identifying Hammerstein-Wiener Models" on page 12-5.

Nonlinear models might be necessary to represent systems that operate over a range of operating points. In some cases, you might fit several linear models, where each model is accurate at specific operating conditions. You can also try using the nonlinear ARX model structure with tree partitions to model such systems. For more information, see "Identifying Nonlinear ARX Models" on page 11-21.

If you know the nonlinear equations describing a system, you can represent this system as a nonlinear grey-box model and estimate the coefficients from experimental data. In this case, the coefficients are the parameters of the model. For more information, see "Grey-Box Model Estimation".

Before fitting a nonlinear model, try transforming your input and output variables such that the relationship between the transformed variables becomes linear. For example, you might be dealing with a system that has current and voltage as inputs to an immersion heater, and the temperature of the heated liquid as an output. In this case, the output depends on the inputs via the power of the heater, which is equal to the product of current and voltage. Instead of fitting a nonlinear model to two-input and one-output data, you can create a new input variable by taking the product of current and voltage. You can then fit a linear model to the single-input/single-output data.

Linear and Nonlinear Dynamics Are Captured Separately

You might have multiple data sets that capture the linear and nonlinear dynamics separately. For example, one data set with low amplitude input (excites the linear dynamics only) and another data set with high amplitude input (excites the nonlinear dynamics). In such cases, first estimate a linear model using the first data set. Next, use the model as an initial model to estimate a nonlinear model using the second data set. For more information, see “Initialize Nonlinear ARX Estimation Using Linear Model” on page 11-25 and “Initialize Hammerstein-Wiener Estimation Using Linear Model” on page 12-8.

Nonlinear Model Estimation

- “Black Box Estimation” on page 11-4
- “Refining Existing Models” on page 11-5
- “Initializing Estimations with Known Information About Linear Component” on page 11-5
- “Estimation Options” on page 11-5

Black Box Estimation

In a black-box or “cold start” estimation, you only have to specify the order to configure the structure of the model.

```
sys = estimator(data,orders)
```

where *`estimator`* is the name of an estimation command to use for the desired model type.

For example, you use `nlarx` to estimate nonlinear ARX models, and `nlhw` for Hammerstein-Wiener models.

The first argument, `data`, is time-domain data represented as an `iddata` object. The second argument, `orders`, represents one or more numbers whose definition depends upon the model type.

- For nonlinear ARX models, `orders` refers to the model orders and delays for defining the regressor configuration.
- For Hammerstein-Wiener models, `orders` refers to the model order and delays of the linear subsystem transfer function.

When working in the System Identification app, you specify the orders in the appropriate edit fields of corresponding model estimation dialog boxes.

Refining Existing Models

You can refine the parameters of a previously estimated nonlinear model using the following command:

```
sys = estimator(data,sys0)
```

This command updates the parameters of an existing model `sys0` to fit the data and returns the results in output model `sys`. For nonlinear systems, *estimator* can be `nlarx`, `nlhw`, or `nlgreyest`.

Initializing Estimations with Known Information About Linear Component

Nonlinear ARX (`idnlarx`) and Hammerstein-Wiener (`idnlhw`) models contain a linear component in their structure. If you have knowledge of the linear dynamics, such as through identification of a linear model using low-amplitude data, you can incorporate it during the estimation of nonlinear models. In particular, you can replace the `orders` input argument with a previously estimated linear model using the following command:

```
sys = estimator(data,LinModel)
```

This command uses the linear model `LinModel` to determine the order of the nonlinear model `sys` as well as initialize the coefficients of its linear component.

Estimation Options

There are many options associated with an estimation algorithm that configures the estimation objective function, initial conditions, and numerical search algorithm, among other things of the model. For every estimation command, *`estimator`*, there is a corresponding option command named *`estimator`*`Options`. For example, use

`nlarxOptions` to generate the option set for `nlarx`. The options command returns an option set that you then pass as an input argument to the corresponding estimation command.

For example, to estimate a nonlinear ARX model with `simulation` as the focus and `lsqnonlin` as the search method, use `nlarxOptions`.

```
load iddata1 z1
Options = nlarxOptions('Focus','simulation','SearchMethod','lsqnonlin');
sys= nlarx(z1,[2 2 1],Options);
```

Information about the options used to create an estimated model is stored in `sys.Report.OptionsUsed`. For more information, see “Estimation Report” on page 1-29.

See Also

Related Examples

- “Identifying Nonlinear ARX Models” on page 11-21
- “Identifying Hammerstein-Wiener Models” on page 12-5
- “Represent Nonlinear Dynamics Using MATLAB File for Grey-Box Estimation” on page 13-38

More About

- “Nonlinear Model Structures” on page 11-7
- “Available Nonlinear Models” on page 11-12
- “About Identified Linear Models” on page 1-13

Nonlinear Model Structures

In this section...
“About System Identification Toolbox Model Objects” on page 11-7
“When to Construct a Model Structure Independently of Estimation” on page 11-8
“Commands for Constructing Nonlinear Model Structures” on page 11-8
“Model Properties” on page 11-9

About System Identification Toolbox Model Objects

Objects are instances of model classes. Each *class* is a blueprint that defines the following information about your model:

- How the object stores data
- Which operations you can perform on the object

This toolbox includes nine classes for representing models. For example, `idss` represents linear state-space models and `idnlarx` represents nonlinear ARX models. For a complete list of available model objects, see “Available Linear Models” on page 1-25 and “Available Nonlinear Models” on page 11-12.

Model *properties* define how a model object stores information. Model objects store information about a model, such as the mathematical form of a model, names of input and output channels, units, names and values of estimated parameters, parameter uncertainties, and estimation report. For example, an `idss` model has an `InputName` property for storing one or more input channel names.

The allowed operations on an object are called *methods*. In System Identification Toolbox software, some methods have the same name but apply to multiple model objects. For example, `step` creates a step response plot for all dynamic system objects. However, other methods are unique to a specific model object. For example, `canon` is unique to state-space `idss` models and `linearize` to nonlinear black-box models.

Every class has a special method, called the *constructor*, for creating objects of that class. Using a constructor creates an instance of the corresponding class or *instantiates the object*. The constructor name is the same as the class name. For example, `idss` and `idnlarx` are both the name of the class and the name of the constructor for instantiating the linear state-space models and nonlinear ARX models, respectively.

When to Construct a Model Structure Independently of Estimation

You use model constructors to create a model object at the command line by specifying all required model properties explicitly.

You must construct the model object independently of estimation when you want to:

- Simulate or analyze the effect of model parameters on its response, independent of estimation.
- Specify an initial guess for specific model parameter values before estimation. You can specify bounds on parameter values, or set up the auxiliary model information in advance, or both. Auxiliary model information includes specifying input/output names, units, notes, user data, and so on.

In most cases, you can use the estimation commands to both construct and estimate the model—without having to construct the model object independently. For example, the estimation command `tfest` creates a transfer function model using data and the number of poles and zeros of the model. Similarly, `nlarx` creates a nonlinear ARX model using data and model orders and delays that define the regressor configuration. For information about how to both construct and estimate models with a single command, see “Model Estimation Commands” on page 1-44.

In case of grey-box models, you must always construct the model object first and then estimate the parameters of the ordinary differential or difference equation.

Commands for Constructing Nonlinear Model Structures

The following table summarizes the model constructors available in the System Identification Toolbox product for representing various types of nonlinear models.

After model estimation, you can recognize the corresponding model objects in the MATLAB Workspace browser by their class names. The name of the constructor matches the name of the object it creates.

For information about how to both construct and estimate models with a single command, see “Model Estimation Commands” on page 1-44.

Summary of Model Constructors

Model Constructor	Resulting Model Class
`idnlgrey`	Nonlinear ordinary differential or difference equation (grey-box models). You write a function or MEX-file to represent the governing equations.
`idnlarx`	Nonlinear ARX models, which define the predicted output as a nonlinear function of past inputs and outputs.
`idnlhw`	Nonlinear Hammerstein-Wiener models, which include a linear dynamic system with nonlinear static transformations of inputs and outputs.

For more information about when to use these commands, see "When to Construct a Model Structure Independently of Estimation" on page 11-8.

Model Properties

A model object stores information in the *properties* of the corresponding model class.

The nonlinear models `idnlarx`, `idnlhw`, and `idnlgrey` are based on the `idnlmodel` superclass and inherit all `idnlmodel` properties.

In general, all model objects have properties that belong to the following categories:

- Names of input and output channels, such as `InputName` and `OutputName`
- Sample time of the model, such as `Ts`
- Time units
- Model order and mathematical structure (for example, ODE or nonlinearities)
- Properties that store estimation results (`Report`)
- User comments, such as `Notes` and `Userdata`

For information about getting help on object properties, see the model reference pages.

The following table summarizes the commands for viewing and changing model property values. Property names are not case sensitive. You do not need to type the entire property name if the first few letters uniquely identify the property.

Task	Command	Example

View all model properties and their values	Use `get`.	Load sample data, compute a nonlinear ARX model, and list the model properties.
		`load iddata1`
		`sys = nlarx(z1,[4 4 1]);`
		`get(sys)`

Access a specific model property	Use dot notation.	View the nonlinearity estimator in the previous model. `sys.Nonlinearity`
	For properties, such as `Report`, that are configured like structures, use dot notation of the form `model.PropertyName.FieldName`. `FieldName` is the name of any field of the property.	View the options used in the nonlinear ARX model estimation. `sys.Report.OptionsUsed`
Change model property values	Use dot notation.	Change the nonlinearity estimator. `sys.Nonlinearity = 'sigmoidnet';`
Access model parameter values and uncertainty information	Use `getpvec` and `getcov` (for `idnlgrey` models only).	Model parameters and associated uncertainty data. `getpvec(sys)`
Set model parameter values and uncertainty information	Use `setpar` and `setcov` (for `idnlgrey` models only).	Set the parameter vector. `sys = setpar(sys,'Value',parlist)`
Get number of parameters	Use `nparams`.	Get the number of parameters. `nparams(sys)`

See Also

Related Examples

- “Identifying Nonlinear ARX Models” on page 11-21
- “Identifying Hammerstein-Wiener Models” on page 12-5
- “Represent Nonlinear Dynamics Using MATLAB File for Grey-Box Estimation” on page 13-38

More About

- “About Identified Nonlinear Models” on page 11-2
- “Available Nonlinear Models” on page 11-12
- “About Identified Linear Models” on page 1-13

Available Nonlinear Models

In this section...
“Overview” on page 11-12
“Nonlinear ARX Models” on page 11-12
“Hammerstein-Wiener Models” on page 11-13
“Nonlinear Grey-Box Models” on page 11-13

Overview

The System Identification Toolbox software provides three types of nonlinear model structures:

- “Nonlinear ARX Models” on page 11-12
- “Hammerstein-Wiener Models” on page 11-13
- “Nonlinear Grey-Box Models” on page 11-13

The toolbox refers to Nonlinear ARX and Hammerstein-Wiener collectively as "nonlinear black box" models. You can configure these models in a variety of ways to represent various behavior using nonlinear functions such as wavelet networks, tree partitions, piece-wise linear functions, polynomials, saturation and dead zones.

The nonlinear grey-box models lets you to estimate coefficients of nonlinear differential equations.

Nonlinear ARX Models

Nonlinear ARX models extend the linear ARX models on page 6-2 to the nonlinear case and have this structure:

$$y(t) = f(y(t - 1), ..., y(t - na), u(t - nk), ..., u(t - nk - nb + 1))$$

where the function f depends on a finite number of previous inputs u and outputs y. na is the number of past output terms and nb is the number of past input terms used to predict the current output. nk is the delay from the input to the output, specified as the number of samples.

Use this model to represent nonlinear extensions of linear models. This structure allows you to model complex nonlinear behavior using flexible nonlinear functions, such as wavelet and sigmoid networks. Typically, you use nonlinear ARX models as black-box structures. The nonlinear function of the nonlinear ARX model is a flexible nonlinearity estimator with parameters that need not have physical significance.

System Identification Toolbox software uses `idnlarx` objects to represent nonlinear ARX models. For more information about estimation, see "Nonlinear ARX Models".

Hammerstein-Wiener Models

Hammerstein-Wiener models describe dynamic systems using one or two static nonlinear blocks in series with a linear block. The linear block is a discrete transfer function and represents the dynamic component of the model.

You can use the Hammerstein-Wiener structure to capture physical nonlinear effects in sensors and actuators that affect the input and output of a linear system, such as dead zones and saturation. Alternatively, use Hammerstein-Wiener structures as black box structures that do not represent physical insight into system processes.

System Identification Toolbox software uses `idnlhw` objects to represent Hammerstein-Wiener models. For more information about estimation, see "Hammerstein-Wiener Models".

Nonlinear Grey-Box Models

Nonlinear state-space models have this representation:

$$\dot{x}(t) = F\left(x(t), u(t)\right)$$
$$y(t) = H\left(x(t), u(t)\right)$$

where F and H can have any parameterization. A nonlinear ordinary differential equation of high order can be represented as a set of first order equations. You use the `idnlgrey` object to specify the structures of such models based on physical insight about your system. The parameters of such models typically have physical interpretations. Use this model to represent nonlinear ODEs with unknown parameters.

For more information about estimating nonlinear state-space models, see "Grey-Box Model Estimation".

See Also

Related Examples

- “Identifying Nonlinear ARX Models” on page 11-21
- “Identifying Hammerstein-Wiener Models” on page 12-5
- “Represent Nonlinear Dynamics Using MATLAB File for Grey-Box Estimation” on page 13-38

More About

- “About Identified Nonlinear Models” on page 11-2
- “Nonlinear Model Structures” on page 11-7

Preparing Data for Nonlinear Identification

Estimating nonlinear ARX and Hammerstein-Wiener models requires uniformly sampled time-domain data. Your data can have one or more input and output channels.

For time-series data, you can only fit nonlinear ARX models and nonlinear state-space models on page 13-35.

Tip Whenever possible, use different data sets for model estimation and validation.

Before estimating models, import your data into the MATLAB workspace and do *one* of the following:

- **In the System Identification app.** Import data into the app, as described in "Represent Data".
- **At the command line.** Represent your data as an `iddata` object, as described in the corresponding reference page.

You can analyze data quality and preprocess data by interpolating missing values, filtering to emphasize a specific frequency range, or resampling using a different sample time (see "Ways to Prepare Data for System Identification" on page 2-6).

Data detrending can be useful in certain cases, such as before modeling the relationship between the change in input and the change in output about an operating point. However, most applications do not require you to remove offsets and linear trends from the data before nonlinear modeling.

See Also

Related Examples

- "Identifying Nonlinear ARX Models" on page 11-21
- "Identifying Hammerstein-Wiener Models" on page 12-5
- "Represent Nonlinear Dynamics Using MATLAB File for Grey-Box Estimation" on page 13-38

More About

- “About Identified Nonlinear Models” on page 11-2
- “Available Nonlinear Models” on page 11-12

What are Nonlinear ARX Models?

In this section...
"Nonlinear ARX Model Extends the Linear ARX Structure" on page 11-17
"Structure of Nonlinear ARX Models" on page 11-18
"Nonlinear ARX Model Orders and Delay" on page 11-19

Nonlinear ARX models extend the linear ARX models on page 6-2 to the nonlinear case. The structure of these models enables you to model complex nonlinear behavior using flexible nonlinear functions, such as wavelet and sigmoid networks. For information about when to fit nonlinear models, see "About Identified Nonlinear Models" on page 11-2.

Nonlinear ARX Model Extends the Linear ARX Structure

A linear SISO ARX model on page 6-2 has the following structure:

$$y(t)+a_1y(t-1)+a_2y(t-2)+\ldots+a_{na}y(t-na)=b_1u(t)+b_2u(t-1)+\ldots+b_{nb}u(t-nb+1)+e(t)$$

Where, u,y, and e are the input, output, and noise. This structure implies that the current output $y(t)$ is predicted as a weighted sum of past output values and current and past input values. na is the number of past output terms, and nb is the number of past input terms used to predict the current output. The input delay nk is set to zero to simplify the notation. Rewriting the equation as a product gives:

$$y_p(t)=\left[-a_1,-a_2,\ldots,-a_{na},b_1,b_2,\ldots,b_{nb}\right]*\left[y(t-1),y(t-2),\ldots,y(t-na),u(t),u(t-1),\ldots,u(t-nb-1)\right]^T$$

where $y(t-1),y(t-2),\ldots,y(t-na),u(t),u(t-1),\ldots,u(t-nb-1)$ are delayed input and output variables, called *regressors*. The coefficients vector *[–a_1, ... ,b_{nb}]* represents the weighting applied to these regressors. The linear ARX model thus predicts the current output y_p as a weighted sum of its regressors.

The structure of a nonlinear ARX model allows the following additional flexibility:

- Instead of the weighted sum of the regressors that represents a linear mapping, the nonlinear ARX model has a more flexible nonlinear mapping function, F.

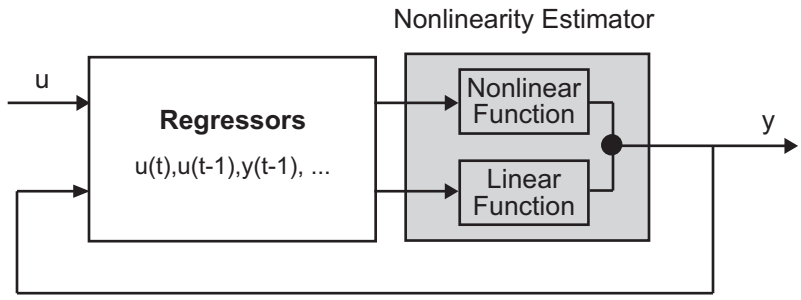
  $$y_p(t)=F(y(t-1),y(t-2),y(t-3),\ldots,u(t),u(t-1),u(t-2),\ldots)$$

Inputs to F are model regressors. When you specify the nonlinear ARX model structure, you can choose one of several available nonlinear functions. For example, F can represent a weighted sum of wavelets that operate on the distance of the regressors from their means. For more information, see "Available Nonlinearity Estimators for Nonlinear ARX Models" on page 11-27.

- Nonlinear ARX regressors can be both delayed input-output variables and more complex, nonlinear expressions of delayed input and output variables. Examples of such nonlinear regressors are $y(t\text{-}1)^2$, $u(t\text{-}1)*y(t\text{-}2)$, abs($u(t\text{-}1)$), and max($u(t\text{-}1)*y(t\text{-}3)$,-10).

Structure of Nonlinear ARX Models

A nonlinear ARX model consists of model regressors and a nonlinearity estimator. The nonlinearity estimator comprises both linear and nonlinear functions that act on the model regressors to give the model output. This block diagram represents the structure of a nonlinear ARX model in a simulation scenario.



The software computes the nonlinear ARX model output y in two stages:

1 It computes regressor values from the current and past input values and past output data.

   In the simplest case, regressors are delayed inputs and outputs, such as *u(t-1)* and *y(t-3)*. These kind of regressors are called *standard regressors*. You specify the standard regressors using the model orders and delay. For more information, see "Nonlinear ARX Model Orders and Delay" on page 11-19. You can also specify *custom* regressors, which are nonlinear functions of delayed inputs and outputs. For example, $u(t\text{-}1)*y(t\text{-}3)$. To create a set of polynomial type regressors, use `polyreg`.

By default, all regressors are inputs to both the linear and the nonlinear function blocks of the nonlinearity estimator. You can choose a subset of regressors as inputs to the nonlinear function block.

2 It maps the regressors to the model output using the nonlinearity estimator block. The nonlinearity estimator block can include linear and nonlinear blocks in parallel. For example:

$$F(x) = L^T(x - r) + d + g\left(Q(x - r)\right)$$

Here, x is a vector of the regressors, and r is the mean of the regressors x. $L^T(x) + d$ is the output of the linear function block and is affine when d ≠ 0. d is a scalar offset.

$g\left(Q(x - r)\right)$ represents the output of the nonlinear function block. Q is a projection matrix that makes the calculations well conditioned. The exact form of $F(x)$ depends on your choice of the nonlinearity estimator. You can select from available nonlinearity estimators on page 11-27, such as tree-partition networks, wavelet networks, and multilayer neural networks. You can also exclude either the linear or the nonlinear function block from the nonlinearity estimator.

When estimating a nonlinear ARX model, the software computes the model parameter values, such as L, r, d, Q, and other parameters specifying g.

Resulting nonlinear ARX models are `idnlarx` objects that store all model data, including model regressors and parameters of the nonlinearity estimator. For more information about these objects, see "Nonlinear Model Structures" on page 11-7.

Typically, you use nonlinear ARX models as black-box structures. The nonlinear function of the nonlinear ARX model is a flexible nonlinearity estimator with parameters that need not have physical significance. You can estimate nonlinear ARX in the **System Identification** app or at the command line using the `nlarx` command. You can use uniformly sampled time-domain input-output data or time-series data (no inputs) for estimating nonlinear ARX models. Your data can have one or more input and output channels. You cannot use frequency-domain data for estimation.

Nonlinear ARX Model Orders and Delay

You use the orders and delays of a nonlinear ARX model to define the standard regressors of the model. The orders and delay are defined as follows:

- *na* — Number of past output terms used to predict the current output.
- *nb* — Number of past input terms used to predict the current output.
- *nk* — Delay from input to the output in terms of the number of samples.

The meaning of *na*, *nb*, and *nk* is similar to that for linear ARX model parameters. Orders are specified as scalars for SISO data, and as *ny*-by-*nu* matrices for MIMO data, where *ny* and *nu* are the number of outputs and inputs. If you are not sure what values to use for the orders and delays, you can estimate them as described in "Preliminary Step – Estimating Model Orders and Input Delays" on page 6-10. Such an estimate is based on linear ARX models and only provides initial guidance. The best orders for a linear ARX model might not be the best orders for a nonlinear ARX model.

System Identification Toolbox software computes standard regressors using the model orders and delays. For example, suppose that you specify `na = 2`, `nb = 3`, and `nk = 5` for a SISO model with input u and output y. The toolbox computes standard regressors $y(t\text{-}2)$, $y(t\text{-}1)$,$u(t\text{-}5)$, $u(t\text{-}6)$, and $u(t\text{-}7)$.

You can also specify custom regressors in addition to standard regressors. For more information, see "Estimate Nonlinear ARX Models in the App" on page 11-29 and "Estimate Nonlinear ARX Models at the Command Line" on page 11-32.

See Also

`idnlarx` | `nlarx`

More About

- "About Identified Nonlinear Models" on page 11-2
- "Identifying Nonlinear ARX Models" on page 11-21
- "Available Nonlinearity Estimators for Nonlinear ARX Models" on page 11-27
- "Using Nonlinear ARX Models" on page 11-55
- "How the Software Computes Nonlinear ARX Model Output" on page 11-57

Identifying Nonlinear ARX Models

Nonlinear ARX models extend the linear ARX model to the nonlinear case. For information about the structure of nonlinear ARX models, see “What are Nonlinear ARX Models?” on page 11-17

You can estimate nonlinear ARX models in the **System Identification** app or at the command line using the `nlarx` command. To estimate a nonlinear ARX model, you first prepare the estimation data. You then configure the model structure and estimation algorithm, and then perform the estimation. After estimation, you can validate the estimated model as described in “Validate Nonlinear ARX Models” on page 11-49.

Prepare Data for Identification

You can use uniformly sampled time-domain input-output data or time-series data (no inputs) for estimating nonlinear ARX models. Your data can have zero or more input channels and one or more output channels. You cannot use frequency-domain data for estimation.

To prepare the data for model estimation, import your data into the MATLAB workspace, and do *one* of the following:

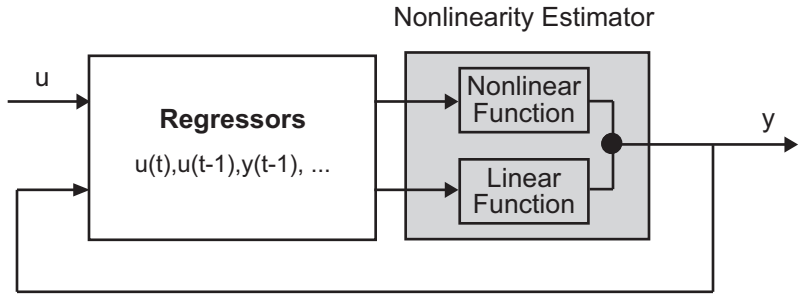
- **In the System Identification app** — Import data into the app, as described in “Represent Data”.
- **At the command line** — Represent your data as an `iddata` object.

After importing the data, you can analyze data quality and preprocess data by interpolating missing values, filtering to emphasize a specific frequency range, or resampling using a different sample time. For more information, see “Ways to Prepare Data for System Identification” on page 2-6. For most applications, you do not need to remove offsets and linear trends from the data before nonlinear modeling. However, data detrending can be useful in some cases, such as before modeling the relationship between the change in input and output about an operating point.

After preparing your estimation data, you can configure your model structure, loss function, and estimation algorithm, and then estimate the model using the estimation data.

Configure Nonlinear ARX Model Structure

A nonlinear ARX model consists of standard and custom regressors, and a nonlinearity estimator. The block diagram represents the structure of a nonlinear ARX model on page 11-18 in a simulation scenario.



To configure the structure of a nonlinear ARX model:

1 Configure the model regressors.

Choose the standard and custom regressors based on your knowledge of the physical system you are trying to model.

a Specify *standard regressors* in *one* of the following ways:

- Specify model orders and delay to create the set of standard regressors. For more information, see “Nonlinear ARX Model Orders and Delay” on page 11-19.
- Initialize using a linear ARX model. You can perform this operation at the command line only. The initialization configures the nonlinear ARX model to use standard regressors of the linear model. For more information, see “Initialize Nonlinear ARX Estimation Using Linear Model” on page 11-25.

b Specify *custom regressors*. Custom regressors are arbitrary functions of past inputs and outputs, such as products, powers, and other MATLAB expressions of input and output variables. Specify custom regressors in addition to, or instead of, standard regressors for greater flexibility in modeling your data.

c Include only a subset of standard and custom regressors as inputs to the nonlinear function of the nonlinear estimator block. Including only a subset of regressors can help reduce model complexity and keep the estimation well-conditioned.

The choice of the subset to use can require multiple trials. You can examine a nonlinear ARX plot on page 11-50 to help you gain insight into which regressors have the strongest effect on the model output. Understanding the relative importance of the regressors on the output can then help you decide which regressors to include in the nonlinear function.

2 Configure the nonlinearity estimator block.

Specify and configure the nonlinearity estimator, $F(x)$.

$$F(x) = L^T(x - r) + d + g\left(Q(x - r)\right)$$

Here, x is a vector of the regressors, and r is the mean of the regressors x. $L^T(x) + d$ is the output of the linear function block and is affine when d ≠ 0. d is a scalar offset.

$g\left(Q(x - r)\right)$ represents the output of the nonlinear function block. Q is a projection matrix that makes the calculations well conditioned. The exact form of $F(x)$ depends on your choice of the nonlinearity estimator. The default nonlinearity is a wavelet network. For information about the available nonlinearity estimators, see "Available Nonlinearity Estimators for Nonlinear ARX Models" on page 11-27.

You can also perform one of the following tasks:

a Exclude the nonlinear function from the nonlinearity estimator such that $F(x) = L^T(x) + d$.

b Exclude the linear function from the nonlinearity estimator such that $F(x) = g\left(Q(x - r)\right)$.

Note You cannot exclude the linear function from tree partitions and neural networks.

For information about how to configure the model structure at the command line and in the app, see "Estimate Nonlinear ARX Models at the Command Line" on page 11-32 and "Estimate Nonlinear ARX Models in the App" on page 11-29.

Specify Estimation Options for Nonlinear ARX Models

To configure the model estimation, specify the loss function to be minimized, and choose the estimation algorithm and other estimation options to perform the minimization.

Configure Loss Function

The loss function or cost function is a function of the error between the model output and the measured output. For more information about loss functions, see “Loss Function and Model Quality Metrics” on page 1-64.

At the command line, use the `nlarx` option set, `nlarxOptions` to configure your loss function. You can specify the following options:

- `Focus` — Specifies whether the simulation or prediction error is minimized during parameter estimation. By default, the software minimizes one-step prediction errors, which correspond to a `Focus` value of `'prediction'`. If you want a model that is optimized for reproducing simulation behavior, specify `Focus` as `'simulation'`. Minimization of simulation error requires differentiable nonlinear functions and takes more time than one-step-ahead prediction error minimization. Thus, you cannot use `treepartition` and `neuralnet` nonlinearities when minimizing the simulation error because these nonlinearity estimators are not differentiable.
- `OutputWeight` — Specifies a weighting of the error in multi-output estimations.
- `Regularization` — Modifies the loss function to add a penalty on the variance of the estimated parameters. For more information, see “Regularized Estimates of Model Parameters” on page 1-48.

Specify Estimation Algorithm

To estimate a nonlinear ARX model, the software uses iterative search algorithms to minimize the error between the simulated or predicted model output and the measured output. At the command line, use `nlarxOptions` to specify the search algorithm and other estimation options. Some of the options you can specify are:

- `SearchMethod` — Search method for minimization of prediction or simulation errors, such as Gauss-Newton and Levenberg-Marquardt line search, and Trust-Region-Reflective Newton approach.
- `SearchOption` — Option set for the search algorithm, with fields that depend on the value of `SearchMethod`, such as:
  - `MaxIter` — Maximum number of iterations.
  - `Tolerance` — Condition for terminating iterative search when the expected improvement of the parameter values is less than a specified value.

To see a complete list of available estimation options, see `nlarxOptions`. For details about how to specify these estimation options in the app, see "Estimate Nonlinear ARX Models in the App" on page 11-29.

After preprocessing the estimation data and configuring the model structure and estimation options, you can estimate the model in the **System Identification** app, or using `nlarx` at the command line. The resulting model is an `idnlarx` object that stores all model data, including model regressors and parameters of the nonlinearity estimator. For more information about these model objects, see "Nonlinear Model Structures" on page 11-7. You can validate the estimated model as described in "Validate Nonlinear ARX Models" on page 11-49.

Initialize Nonlinear ARX Estimation Using Linear Model

At the command line, you can use an ARX structure polynomial model (`idpoly` with only *A* and *B* as active polynomials) for nonlinear ARX estimation. To learn more about when to use linear models, see "When to Fit Nonlinear Models" on page 11-2.

Typically, you create a linear ARX model using the `arx` command. You can provide the linear model when constructing or estimating a nonlinear ARX model. For example, use the following syntax to estimate a nonlinear ARX model using estimation data and a linear ARX model `LinARXModel`.

```
m = nlarx(data,LinARXModel)
```

Here `m` is an `idnlarx` object, and `data` is a time-domain `iddata` object. The software uses the linear model for initializing the nonlinear ARX estimation by:

- Assigning the linear ARX model orders and delays as initial values of the nonlinear ARX model orders (`na` and `nb` properties of the `idnlarx` object) and delays (`nk` property). The software uses these orders and delays to compute standard regressors in the nonlinear ARX model structure on page 11-18.
- Using the *A* and *B* polynomials of the linear model to compute the linear function of the nonlinearity estimators (`LinearCoef` parameter of the nonlinearity estimator object), except if the nonlinearity estimator is a neural network.

During estimation, the estimation algorithm uses these values to adjust the nonlinear model to the data.

Note When you use the same data for estimation, a nonlinear ARX model initialized using a linear ARX model produces a better fit to measured output than the linear ARX model itself.

By default, the nonlinearity estimator is the wavelet network (`wavenet` object). You can also specify different input and output nonlinearity estimators. For example, you can specify a sigmoid network nonlinearity estimator.

```
m = nlarx(data,LinARXModel,'sigmoid')
```

For an example, see “Estimate Nonlinear ARX Models Initialized Using Linear ARX Models” on page 11-44.

See Also

Functions

`idnlarx` | `nlarx`

Apps

System Identification

More About

- “What are Nonlinear ARX Models?” on page 11-17
- “Estimate Nonlinear ARX Models in the App” on page 11-29
- “Estimate Nonlinear ARX Models at the Command Line” on page 11-32
- “Validate Nonlinear ARX Models” on page 11-49
- “Using Nonlinear ARX Models” on page 11-55

Available Nonlinearity Estimators for Nonlinear ARX Models

System Identification Toolbox software provides several nonlinearity estimators $F(x)$ for nonlinear ARX models. For more information about $F(x)$, see "Structure of Nonlinear ARX Models" on page 11-18.

Each nonlinearity estimator corresponds to an object class in this toolbox. When you estimate nonlinear ARX models in the app, System Identification Toolbox creates and configures objects based on these classes. You can also create and configure nonlinearity estimators at the command line.

Most nonlinearity estimators represent the nonlinear function as a summed series of nonlinear units, such as wavelet networks or sigmoid functions. You can configure the number of nonlinear units n for estimation. For a detailed description of each estimator, see the references page of the corresponding nonlinearity class.

Nonlinearity	Class	Structure	Comments
Wavelet network (default)	`wavenet`	$g(x)=\sum_{k=1}^{n}\alpha_k\kappa\left(\beta_k(x-\gamma_k)\right)$ where $\kappa(s)$ is the wavelet function.	By default, the estimation algorithm determines the number of units n automatically.
One layer sigmoid network	`sigmoidnet`	$g(x)=\sum_{k=1}^{n}\alpha_k\kappa\left(\beta_k(x-\gamma_k)\right)$ where $\kappa(s)=\left(e^s+1\right)^{-1}$ is the sigmoid function. β_k is a row vector such that $\beta_k(x-\gamma_k)$ is a scalar.	Default number of units n is 10.

Nonlinearity	Class	Structure	Comments
Tree partition	`treepartition`	Piecewise linear function over partitions of the regressor space defined by a binary tree.	The estimation algorithm determines the number of units automatically. Try using tree partitions for modeling data collected over a range of operating conditions.
F is linear in x	`linear`	This estimator produces a model that is similar to the linear ARX model, but offers the additional flexibility of specifying custom regressors.	Use to specify custom regressors as the nonlinearity estimator and exclude a nonlinearity mapping function.
Multilayered neural network	`neuralnet`	Uses as a `network` object created using the Neural Network Toolbox™ software.	
Custom network (user-defined)	`customnet`	Similar to sigmoid network but you specify $\kappa(s)$.	(For advanced use) Uses the unit function that you specify.

See Also

More About

- "What are Nonlinear ARX Models?" on page 11-17
- "Identifying Nonlinear ARX Models" on page 11-21

Estimate Nonlinear ARX Models in the App

You can estimate nonlinear ARX models in the **System Identification** app after you perform the following tasks:

- Import data into the System Identification app (see "Preparing Data for Nonlinear Identification" on page 11-15).
- (Optional) Choose a nonlinearity estimator in "Available Nonlinearity Estimators for Nonlinear ARX Models" on page 11-27.

To estimate a nonlinear ARX model using the imported estimation data and chosen nonlinearity estimators:

1. In the **System Identification** app, select **Estimate** > **Nonlinear models** to open the Nonlinear Models dialog box.
2. In the **Configure** tab, verify that `Nonlinear ARX` is selected in the **Model type** list.
3. (Optional) Edit the **Model name** by clicking . The name of the model should be unique to all nonlinear ARX models in the System Identification app.
4. (Optional) If you want to refine the parameters of a previously estimated model or configure the model structure to match that of an existing model:

   a. Click **Initialize**. A Initial Model Specification dialog box opens.

   b. In the Initial Model drop-down list, select a nonlinear ARX model.

      The model must be in the Model Board of the System Identification app and the input/output dimensions of this initial model must match that of the estimation data, selected as **Working Data** in the app.

   c. Click **OK**.

      The model structure as well as the parameter values are updated to match that of the selected model.

   Clicking **Estimate** causes the estimation to use the parameters of the initial model as the starting point.

Note When you select an initial model, you can optionally update the estimation algorithm settings to match those used for the initial model by selecting the **Inherit the model's algorithm properties** option.

5 Keep the default settings in the Nonlinear Models dialog box that specify the model structure and the algorithm, or modify these settings:

Note For more information about available options, click **Help** in the Nonlinear Models dialog box to open the app help.

What to Configure	Options in Nonlinear Models GUI	Comment
Model order	In the **Regressors** tab, edit the **No. of Terms** corresponding to each input and output channel.	Model order *na* is the output number of terms and *nb* is the input number of terms.
Input delay	In the **Regressors** tab, edit the **Delay** corresponding to an input channel.	If you do not know the input delay value, click **Infer Input Delay**. This action opens the Infer Input Delay dialog box to suggest possible delay values.
Regressors	In the **Regressors** tab, click **Edit Regressors**.	This action opens the Model Regressors dialog box. Use this dialog box to create custom regressors or to include specific regressors in the nonlinear block.
Nonlinearity estimator	In the **Model Properties** tab.	To use all standard and custom regressors in the linear block only, you can exclude the nonlinear block by setting **Nonlinearity** to `None`.
Estimation algorithm	In the **Estimate** tab, click **Algorithm Options**.	

6 To obtain regularized estimates of model parameters, in the **Estimate** tab, click **Estimation Options**. Specify the regularization constants in the

Regularization_Tradeoff_Constant and **Regularization_Weighting** fields. To learn more, see "Regularized Estimates of Model Parameters" on page 1-48.

7 Click **Estimate** to add this model to the System Identification app.

The **Estimate** tab displays the estimation progress and results.

8 Validate the model response by selecting the desired plot in the **Model Views** area of the System Identification app. For more information about validating models, see "Validate Nonlinear ARX Models" on page 11-49.

If you get a poor fit, try changing the model structure or algorithm configuration in step 5.

You can export the model to the MATLAB workspace by dragging it to **To Workspace** in the **System Identification** app.

See Also

Apps

System Identification

More About

- "What are Nonlinear ARX Models?" on page 11-17
- "Available Nonlinearity Estimators for Nonlinear ARX Models" on page 11-27
- "Identifying Nonlinear ARX Models" on page 11-21
- "Validate Nonlinear ARX Models" on page 11-49
- "Using Nonlinear ARX Models" on page 11-55

Estimate Nonlinear ARX Models at the Command Line

You can estimate nonlinear ARX models after you perform the following tasks:

- Prepare your data, as described in "Preparing Data for Nonlinear Identification" on page 11-15.
- (Optional) Estimate model orders and delays the same way you would for linear ARX models. See "Preliminary Step – Estimating Model Orders and Input Delays" on page 6-10.
- (Optional) Choose a nonlinearity estimator in "Available Nonlinearity Estimators for Nonlinear ARX Models" on page 11-27.
- (Optional) Estimate or construct a linear ARX model for initialization of nonlinear ARX model. See "Initialize Nonlinear ARX Estimation Using Linear Model" on page 11-25.

Estimate Model Using nlarx

Use `nlarx` to both construct and estimate a nonlinear ARX model. After each estimation, validate the model on page 11-49 by comparing it to other models and simulating or predicting the model response.

Basic Estimation

Start with the simplest estimation using `m = nlarx(data,[na nb nk])`. For example:

```
load iddata1;
% na = nb = 2 and nk = 1
m = nlarx(z1,[2 2 1])

m =
Nonlinear ARX model with 1 output and 1 input
 Inputs: u1
 Outputs: y1
 Standard regressors corresponding to the orders
   na = 2, nb = 2, nk = 1
 No custom regressor
 Nonlinear regressors:
   y1(t-1)
   y1(t-2)
   u1(t-1)
   u1(t-2)
```

```
 Nonlinearity: wavenet with 1 unit

Sample time: 0.1 seconds

Status:
Estimated using NLARX on time domain data "z1".
Fit to estimation data: 68.83% (prediction focus)
FPE: 1.975, MSE: 1.885
```

The second input argument `[na nb nk]` specify the model orders and delays. By default, the nonlinearity estimator is the wavelet network (see the `wavenet` reference page), which takes all standard regressors as inputs to its linear and nonlinear functions. `m` is an `idnlarx` object.

For MIMO systems, *nb*, *nf*, and *nk* are *ny*-by-*nu* matrices. See the `nlarx` reference page for more information about MIMO estimation.

Specify a different nonlinearity estimator (for example, sigmoid network).

```
M = nlarx(z1,[2 2 1],'sigmoid');
```

Create an `nlarxOptions` option set and configure the `Focus` property to minimize simulation error.

```
opt = nlarxOptions('Focus','simulation');
M = nlarx(z1,[2 2 1],'sigmoid',opt);
```

Configure Model Regressors

Standard Regressors

Change the model order to create a model structure with different model regressors, which are delayed input and output variables that are inputs to the nonlinearity estimator.

Custom Regressors

Explore including custom regressors in the nonlinear ARX model structure. Custom regressors are in addition to the standard model regressors (see “Nonlinear ARX Model Orders and Delay” on page 11-19).

Use `polyreg` or `customreg` to construct custom regressors in terms of model input-output variables. You can specify custom regressors using the `CustomRegressors`

property of the `idnlarx` class or `addreg` to append custom regressors to an existing model.

For example, generate regressors as polynomial functions of inputs and outputs:

```
load iddata1
m = nlarx(z1,[2 2 1],'sigmoidnet');
getreg(m) % displays all regressors

Regressors:
    y1(t-1)
    y1(t-2)
    u1(t-1)
    u1(t-2)

% Generate polynomial regressors up to order 2:
reg = polyreg(m)

4x1  array of Custom Regressors with fields: Function, Arguments, Delays, Vectorized.
```

Append polynomial regressors to CustomRegressors.

```
m = addreg(m,reg);
getreg(m)

Regressors:
    y1(t-1)
    y1(t-2)
    u1(t-1)
    u1(t-2)
    y1(t-1).^2
    y1(t-2).^2
    u1(t-1).^2
    u1(t-2).^2
```

`m` now includes polynomial regressors.

You can also specify arbitrary functions of input and output variables. For example:

```
load iddata1
m = nlarx(z1,[2 2 1],'sigmoidnet','CustomReg',{'y1(t-1)^2','y1(t-2)*u1(t-3)'});
getreg(m) % displays all regressors

Regressors:
    y1(t-1)
```

```
    y1(t-2)
    u1(t-1)
    u1(t-2)
    y1(t-1)^2
    y1(t-2)*u1(t-3)
```

Append polynomial regressors to CustomRegressors.

```
m = addreg(m,reg);
getreg(m) % polynomial regressors

Regressors:
    y1(t-1)
    y1(t-2)
    u1(t-1)
    u1(t-2)
    y1(t-1)^2
    y1(t-2)*u1(t-3)
    y1(t-1).^2
    y1(t-2).^2
    u1(t-1).^2
    u1(t-2).^2
```

Manipulate custom regressors using the `CustomRegressors` property of the `idnlarx` class. For example, to get the function handle of the first custom regressor in the array:

```
CReg1 = m.CustomReg(1).Function;
```

View the regressor expression.

```
m.CustomReg(1).Display

ans =
'y1(t-1)^2'
```

You can exclude all standard regressors and use only custom regressors in the model structure by setting `na=nb=nk=0`:

```
m = nlarx(z1,[0 0 0],'linear','CustomReg',{'y1(t-1)^2','y1(t-2)*u1(t-3)'});
```

In advanced applications, you can specify advanced estimation options for nonlinearity estimators. For example, `wavenet` and `treepartition` classes provide the `Options` property for setting such estimation options.

Specify Linear and Nonlinear Regressors

By default, all model regressors enter as inputs to both linear and nonlinear function blocks of the nonlinearity estimator. To reduce model complexity and keep the estimation well-conditioned, use a subset of regressors as inputs to the nonlinear function of the nonlinear estimator block. For example, specify a nonlinear ARX model to be linear in past outputs and nonlinear in past inputs.

```
m = nlarx(z1,[2 2 1]); % all standard regressors are
                       % inputs to the nonlinear function
getreg(m); % lists all standard regressors

Regressors:
    y1(t-1)
    y1(t-2)
    u1(t-1)
    u1(t-2)

m = nlarx(z1,[4 4 1],sigmoidnet,'nlreg',[5 6 7 8]);
```

This example uses `getreg` to determine the index of each regressor from the complete list of all model regressors. Only regressor numbers 5 through 8 are inputs to the nonlinear function - `getreg` shows that these regressors are functions of the input variable `u1`. `nlreg` is an abbreviation for the `NonlinearRegressors` property of the `idnlarx` class. Alternatively, include only input regressors in the nonlinear function block using:

```
m = nlarx(z1,[4 4 1],sigmoidnet,'nlreg','input');
```

When you are not sure which regressors to include as inputs to the nonlinear function block, specify to search during estimation for the optimum regressor combination:

```
m = nlarx(z1,[4 4 1],sigmoidnet,'nlreg','search');
```

This search typically takes a long time. You can display the search progress using:

```
opt = nlarxOptions('Display','on');
m = nlarx(z1,[4 4 1],sigmoidnet,'nlreg','search',opt);
```

After estimation, use `m.NonlinearRegressors` to view which regressors were selected by the automatic regressor search.

```
m.NonlinearRegressors
```

```
ans =

     3     5     6     7
```

Configure Nonlinearity Estimator

Specify the nonlinearity estimator directly in the estimation command as:

- A character vector of the nonlinearity name, which uses the default nonlinearity configuration.

```
m = nlarx(z1, [2 2 1],'sigmoidnet');
```

- Nonlinearity object.

```
m = nlarx(z1,[2 2 1],wavenet('num',5));
```

This estimation uses a nonlinear ARX model with a wavelet nonlinearity that has 5 units. To construct the nonlinearity object before providing it as an input to the nonlinearity estimator:

```
w = wavenet('num', 5);
m = nlarx(z1,[2 2 1],w);
% or
w = wavenet;
w.NumberOfUnits = 5;
m = nlarx(z1,[2 2 1],w);
```

For MIMO systems, you can specify a different nonlinearity for each output. For example, to specify `sigmoidnet` for the first output and `wavenet` for the second output:

```
load iddata1 z1
load iddata2 z2
data = [z1, z2(1:300)];
M = nlarx(data,[[1 1;1 1] [2 1;1 1] [2 1;1 1]],[sigmoidnet wavenet]);
```

If you want the same nonlinearity for all output channels, specify one nonlinearity.

```
M = nlarx(data,[[1 1;1 1] [2 1;1 1] [2 1;1 1]],sigmoidnet);
```

The following table summarizes values that specify nonlinearity estimators.

Nonlinearity	Value (Default Nonlinearity Configuration)	Class
Wavelet network (default)	`'wavenet'` or `'wave'`	`wavenet`
One layer sigmoid network	`'sigmoidnet'` or `'sigm'`	`sigmoidnet`
Tree partition	`'treepartition'` or `'tree'`	`treepartition`
F is linear in x	`'linear'` or `[ ]`	`linear`

Additional available nonlinearities include multilayered neural networks and custom networks that you create.

Specify a multilayered neural network using:

```
m = nlarx(data,[na nb nk],NNet)
```

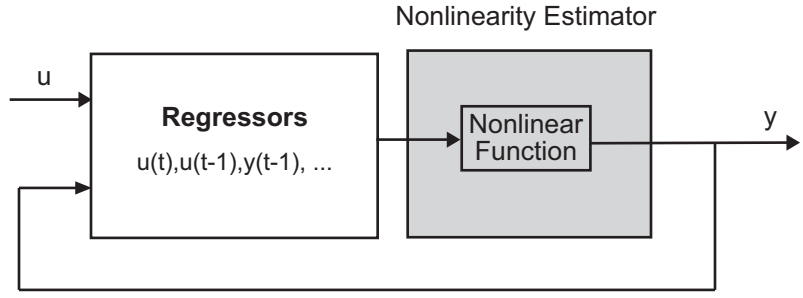
where `NNet` is the neural network object you create using the Neural Network Toolbox software. See the `neuralnet` reference page.

Specify a custom network by defining a function called `gaussunit.m`, as described in the `customnet` reference page. Define the custom network object `CNetw` and estimate the model:

```
CNetw = customnet(@gaussunit);
m = nlarx(data,[na nb nk],CNetw)
```

Exclude Linear Function in Nonlinearity Estimator

If your model includes `wavenet`, `sigmoidnet`, and `customnet` nonlinearity estimators, you can exclude the linear function using the `LinearTerm` property of the nonlinearity estimator. The nonlinearity estimator becomes $F(x) = g\left(Q(x - r)\right)$.



For example:

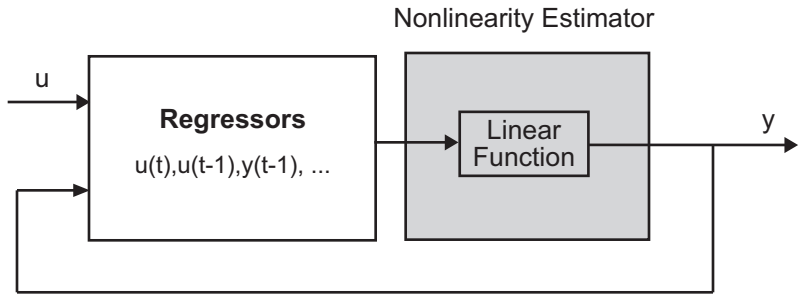
```
SNL = sigmoidnet('LinearTerm','off');
m = nlarx(z1,[2 2 1],SNL);
```

Note You cannot exclude the linear function from tree partition and neural network nonlinearity estimators.

Exclude Nonlinear Function in Nonlinearity Estimator

Configure the nonlinear ARX structure to include only the linear function in the nonlinearity estimator by setting the nonlinearity to `linear`. In this case, $F(x)=$

$L^T(x)+d$ is a weighted sum of model regressors plus an offset. Such models provide a bridge between purely linear ARX models and fully flexible nonlinear models.



In the simplest case, a model with only standard regressors is linear (affine). For example, this structure:

```
m = nlarx(z1,[2 2 1],'linear');
```

is similar to the linear ARX model:

```
lin_m = arx(z1,[2 2 1]);
```

However, the nonlinear ARX model m is more flexible than the linear ARX model `lin_m` because it contains the offset term, d. This offset term provides the additional flexibility of capturing signal offsets, which is not available in linear models.

A popular nonlinear ARX configuration in many applications uses polynomial regressors to model system nonlinearities. In such cases, the system is considered to be a linear

combination of products of (delayed) input and output variables. Use the `polyreg` command to easily generate combinations of regressor products and powers.

For example, suppose that you know the output $y(t)$ of a system to be a linear combination of $(y(t-1))^2$ and $y(t-2)*u(t-3)$. To model such a system, use:

```
M = nlarx(z1,[0 0 0],'linear','CustomReg',{'y1(t-1)^2','y1(t-2)*u1(t-3)'});
```

`M` has no standard regressors and the nonlinearity in the model is described only by the custom regressors.

Iteratively Refine Model

If your model structure includes nonlinearities that support iterative search (see "Specify Estimation Options for Nonlinear ARX Models" on page 11-23), you can use `nlarx` to refine model parameters:

```
m1 = nlarx(z1,[2 2 1],'sigmoidnet');
m2 = nlarx(z1,m1); % can repeatedly run this command
```

You can also use `pem` to refine the original model:

```
m2 = pem(z1,m1);
```

Check the search termination criterion `m.Report.Termination.WhyStop` . If `WhyStop` indicates that the estimation reached the maximum number of iterations, try repeating the estimation and possibly specifying a larger value for the `nlarxOptions.SearchOption.MaxIter` estimation option:

```
opt = nlarxOptions;
opt.SearchOption.MaxIter = 30;
m2 = nlarx(z1,m1,opt); % runs 30 more iterations
                       % starting from m1
```

When the `m.Report.Termination.WhyStop` value is `Near (local) minimum, (norm( g) < tol` or `No improvement along the search direction with line search` , validate your model to see if this model adequately fits the data. If not, the solution might be stuck in a local minimum of the cost-function surface. Try adjusting the `SearchOption.Tolerance` value or the `SearchMethod` option in the nlarxOptions option set, and repeat the estimation.

You can also try perturbing the parameters of the last model using `init` (called *randomization*) and refining the model using `nlarx`:

```
M1 = nlarx(z1, [2 2 1], 'sigm'); % original model
M1p = init(M1);    % randomly perturbs parameters about nominal values
M2 = nlarx(z1, M1p);  % estimates parameters of perturbed model
```

You can display the progress of the iterative search in the MATLAB Command Window using the `nlarxOptions.Display` estimation option:

```
opt = nlarxOptions('Display','on');
M2= nlarx(z1,M1p,opt);
```

Troubleshoot Estimation

If you do not get a satisfactory model after many trials with various model structures and algorithm settings, it is possible that the data is poor. For example, your data might be missing important input or output variables and does not sufficiently cover all the operating points of the system.

Nonlinear black-box system identification usually requires more data than linear model identification to gain enough information about the system.

Use nlarx to Estimate Nonlinear ARX Models

This example shows how to use `nlarx` to estimate a nonlinear ARX model for measured input/output data.

Prepare the data for estimation.

```
load twotankdata
z = iddata(y, u, 0.2);
ze = z(1:1000); zv = z(1001:3000);
```

Estimate several models using different model orders, delays, and nonlinearity settings.

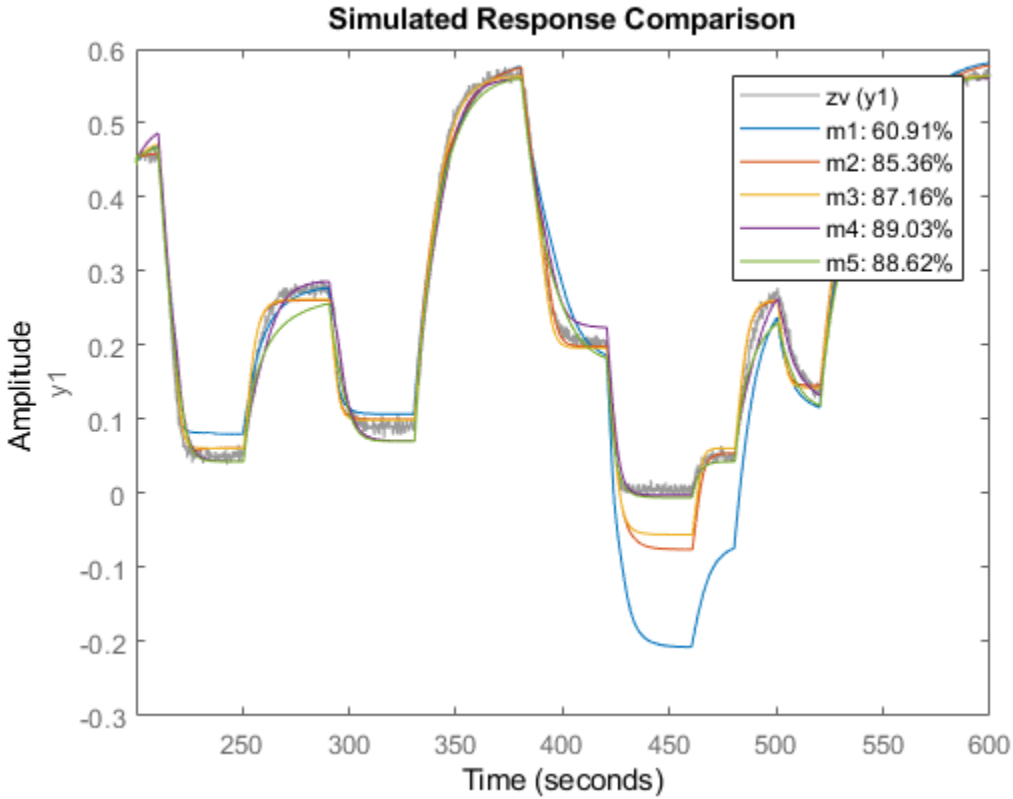
```
m1 = nlarx(ze,[2 2 1]);
m2 = nlarx(ze,[2 2 3]);
m3 = nlarx(ze,[2 2 3],wavenet('num',8));
m4 = nlarx(ze,[2 2 3],wavenet('num',8),...
                      'nlr', [1 2]);
```

An alternative way to perform the estimation is to configure the model structure first, and then to estimate this model.

```
m5 = idnlarx([2 2 3],sigmoidnet('num',14),'nlr',[1 2]);
m5 = nlarx(ze,m5);
```

Compare the resulting models by plotting the model outputs with the measured output.

```
compare(zv, m1,m2,m3,m4,m5)
```



See Also

Functions

`idnlarx` | `nlarx`

More About

- “Available Nonlinearity Estimators for Nonlinear ARX Models” on page 11-27
- “Identifying Nonlinear ARX Models” on page 11-21
- “Validate Nonlinear ARX Models” on page 11-49
- “Using Nonlinear ARX Models” on page 11-55
- “Estimate Nonlinear ARX Models Initialized Using Linear ARX Models” on page 11-44

Estimate Nonlinear ARX Models Initialized Using Linear ARX Models

This example shows how to estimate nonlinear ARX models by using linear ARX models.

Load the estimation data.

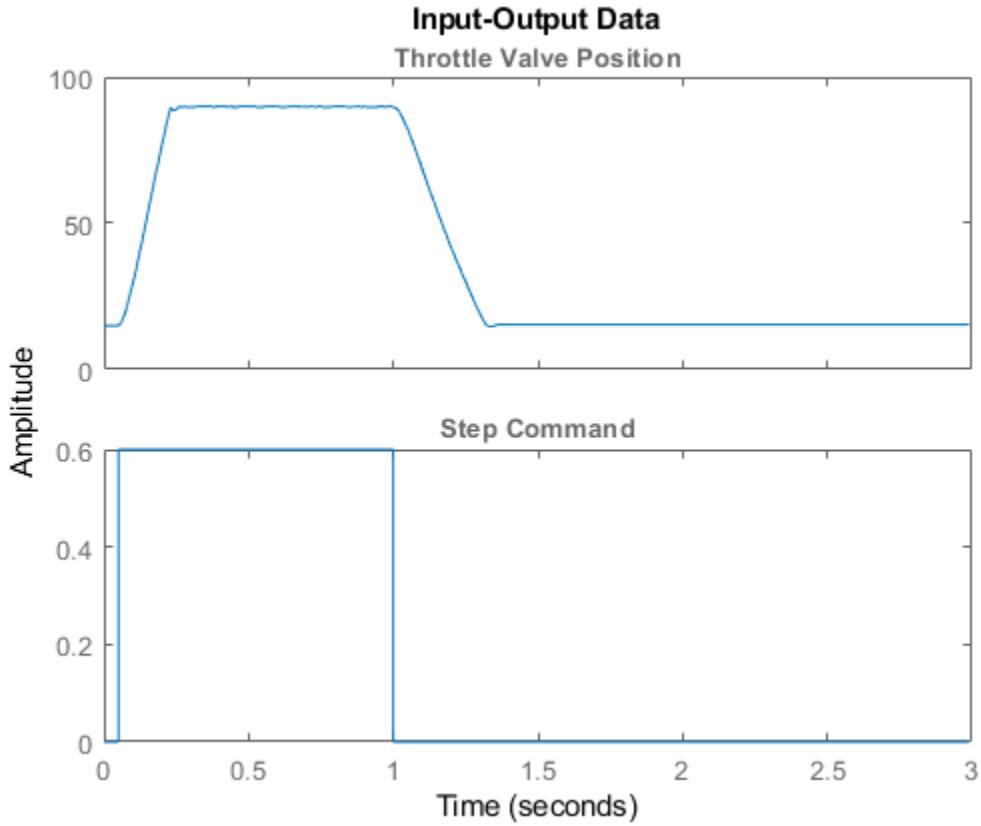
```
load throttledata.mat
```

This command loads the data object `ThrottleData` into the workspace. The object contains input and output samples collected from an engine throttle system, sampled at a rate of 100 Hz.

A DC motor controls the opening angle of the butterfly valve in the throttle system. A step signal (in volts) drives the DC motor. The output is the angular position (in degrees) of the valve.

Plot the data to view and analyze the data characteristics.

```
plot(ThrottleData)
```



In the normal operating range of 15-90 degrees, the input and output variables have a linear relationship. You use a linear model of low order to model this relationship.

In the throttle system, a hard stop limits the valve position to `90` degrees, and a spring brings the valve to `15` degrees when the DC motor is turned off. These physical components introduce nonlinearities that a linear model cannot capture.

Estimate an ARX model to model the linear behavior of this single-input single-output system in the normal operating range.

Detrend the data because linear models cannot capture offsets.

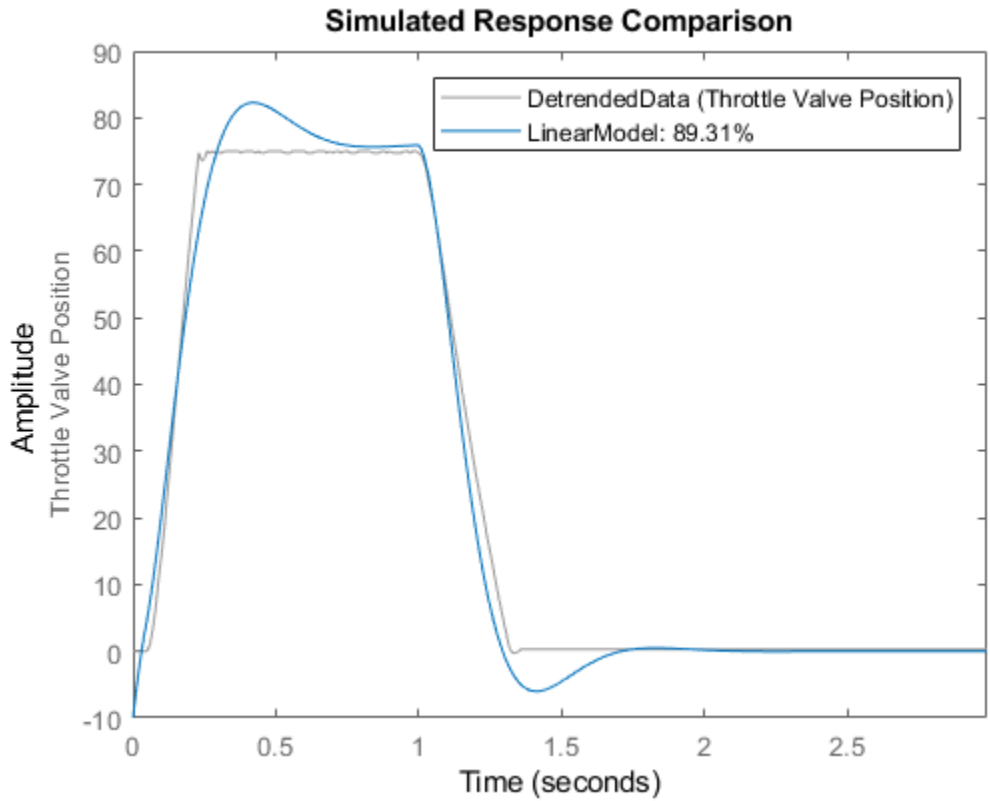
```
Tr = getTrend(ThrottleData);
Tr.OutputOffset = 15;
DetrendedData = detrend(ThrottleData,Tr);
```

Estimate a linear ARX model with na=2, nb=1, nk=1.

```
opt = arxOptions('Focus','simulation');
LinearModel = arx(DetrendedData,[2 1 1],opt);
```

Compare the simulated model response with the estimation data.

```
compare(DetrendedData, LinearModel)
```



The linear model captures the rising and settling behavior in the linear operating range but does not account for output saturation at 90 degrees.

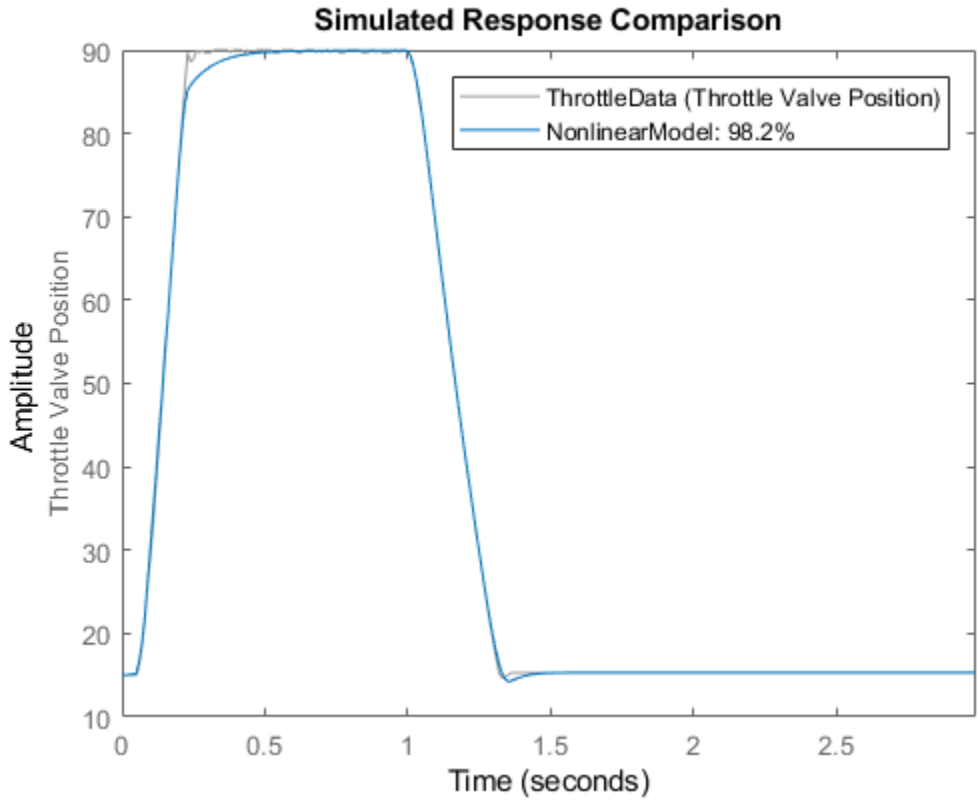
Estimate a nonlinear ARX model to model the output saturation.

```
optNL = nlarxOptions('Focus','simulation');
NonlinearModel = nlarx(ThrottleData,LinearModel,'sigmoidnet',optNL);
```

The software uses the orders and delay of the linear model for the orders of the nonlinear model. In addition, the software computes the linear function of `sigmoidnet` nonlinearity estimator.

Compare the nonlinear model with the estimation data.

```
compare(ThrottleData, NonlinearModel)
```



The model captures the nonlinear effects (output saturation) and improves the overall fit to data.

See Also

More About

- “Identifying Nonlinear ARX Models” on page 11-21
- “Estimate Nonlinear ARX Models in the App” on page 11-29
- “Estimate Nonlinear ARX Models at the Command Line” on page 11-32

Validate Nonlinear ARX Models

After estimating a nonlinear ARX model on page 11-21 for your system, you can validate whether it reproduces the system behavior within acceptable bounds. You can validate your model in different ways. It is recommended that you use separate data sets for estimating and validating your model. If the validation indicates low confidence in the estimation, then see "Troubleshooting Model Estimation" on page 17-119 for next steps. For general information about validating models, see "Model Validation".

Compare Model Output to Measured Output

Plot simulated or predicted model output and measured output data for comparison, and compute best fit values. At the command line, use `compare` command. You can also use `sim` and `predict` to simulate and predict model response. For information about plotting simulated and predicted output in the app, see "Simulation and Prediction in the App" on page 17-12.

Check Iterative Search Termination Conditions

The estimation report that is generated after model estimation lists the reason the software terminated the estimation. For example, suppose that the report indicates that the estimation reached the maximum number of iterations. You can try repeating the estimation by specifying a larger value for the maximum number of iterations. For information about how to configure the maximum number of iterations and other estimation options, see "Specify Estimation Options for Nonlinear ARX Models" on page 11-23.

To view the estimation report in the app, after model estimation is complete, view the **Estimation Report** area of the **Estimate** tab. At the command line, use `M.Report.Termination` to display the estimation termination conditions, where `M` is the estimated Nonlinear ARX model. For example, check the `M.Report.Termination.WhyStop` field that describes why the estimation was stopped.

For more information about the estimation report, see "Estimation Report" on page 1-29.

Check the Final Prediction Error and Loss Function Values

You can compare the performance of several estimated models by comparing the final prediction error and loss function values that are shown in the estimation report.

To view these values for an estimated model `M` at the command line, use the `M.Report.Fit.FPE` (final prediction error) and `M.Report.Fit.LossFcn` (value of loss function at estimation termination) properties. Smaller values typically indicate better performance. However, `M.Report.Fit.FPE` values can be unreliable when the model contains many parameters relative to the estimation data size. Use these indicators with other validation techniques to draw reliable conclusions.

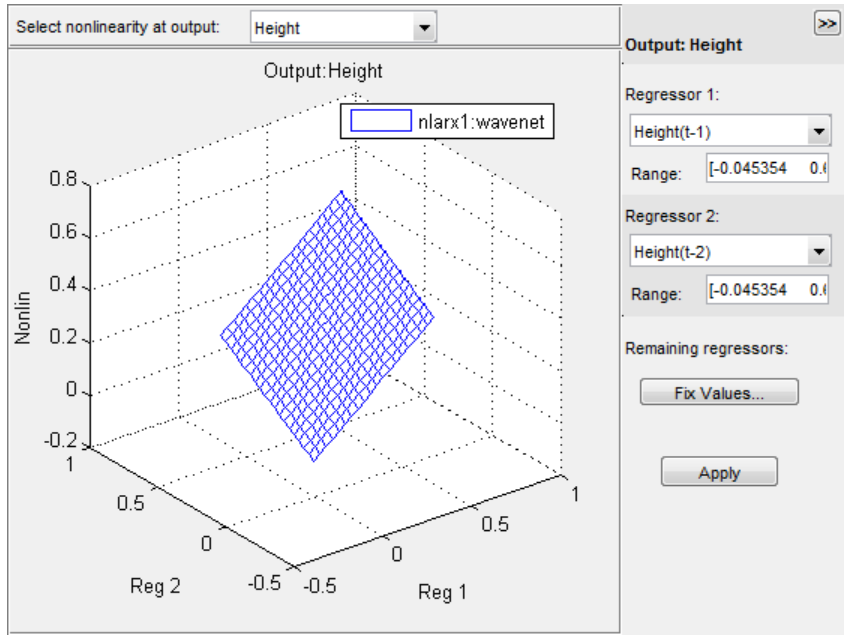
Perform Residual Analysis

Residuals are differences between the model output and the measured output. Thus, residuals represent the portion of the output not explained by the model. You can analyze the residuals using techniques such as the whiteness test and the independence test. For more information about these tests, see “What Is Residual Analysis?” on page 17-43.

At the command line, use `resid` to compute, plot, and analyze the residuals. To plot residuals in the app, see “How to Plot Residuals in the App” on page 17-47.

Examine Nonlinear ARX Plots

A nonlinear ARX plot displays the evaluated model nonlinearity for a chosen model output as a function of one or two model regressors. For a model `M`, the model nonlinearity (`M.Nonlinearity`) is a nonlinearity estimator function, such as `wavenet`, `sigmoidnet`, or `treepartition`, that uses model regressors as its inputs.



To understand what is plotted, suppose that `{r1,r2,…,rN}` are the `N` regressors used by a nonlinear ARX model `M` with nonlinearity `nl` corresponding to a model output. You can use `getreg(M)` to view these regressors. The expression `Nonlin = evaluate(nl, [v1,v2,...,vN])` returns the model output for given values of these regressors, that is, `r1` = `v1`, `r2` = `v2`, ..., `rN` = `vN`. For plotting the nonlinearities, you select one or two of the `N` regressors, for example, `rsub = {r1,r4}`. The software varies the values of these regressors in a specified range, while fixing the value of the remaining regressors, and generates the plot of `Nonlin` vs. `rsub`. By default, the software sets the values of the remaining fixed regressors to their estimated means, but you can change these values. The regressor means are stored in the `Nonlinearity.Parameters.RegressorMean` property of the model.

Examining a nonlinear ARX plot can help you gain insight into which regressors have the strongest effect on the model output. Understanding the relative importance of the regressors on the output can help you decide which regressors to include in the nonlinear function for that output. If the shape of the plot looks like a plane for all the chosen regressor values, then the model is probably linear in those regressors. In this case, you can remove the corresponding regressors from nonlinear block, and repeat the estimation.

Furthermore, you can create several nonlinear models for the same data using different nonlinearity estimators, such a `wavenet` network and `treepartition`, and then compare the nonlinear surfaces of these models. Agreement between plots for various models increases the confidence that these nonlinear models capture the true dynamics of the system.

Creating a Nonlinear ARX Plot

To create a nonlinear ARX plot in the **System Identification** app, select the **Nonlinear ARX** check box in the **Model Views** area. To include or exclude a model on the plot, click the corresponding model icon in the app. For general information about creating and working with plots in the app, see “Working with Plots” on page 21-11.

At the command line, after you have estimated a nonlinear ARX model `M`, use `plot` to view the shape of the nonlinearity.

```
plot(M)
```

You can use additional `plot` arguments to specify the following information:

- Include multiple nonlinear ARX models on the plot.
- Configure the regressor values for computing the nonlinearity values.

Configuring a Nonlinear ARX Plot

To configure the nonlinear ARX plot:

1. Select the output channel in the **Select nonlinearity at output** drop-down list. The nonlinearity values that correspond to the selected output channel are displayed.
2. Select **Regressor 1** from the list of available regressors. In the **Range** field, enter the range of values to include on the plot for this regressor. The regressor values are plotted on the **Reg1** axis of the plot.

   If the regressor selection options are not visible, click to expand the Nonlinear ARX Model Plot window.
3. Specify **Regressor 2** as one of the following options:

   - To display three axes on the plot, select **Regressor 2**. In the **Range** field, enter the range of values to include on the plot for this regressor. The regressor values are plotted on the **Reg2** axis of the plot.

- To display only two axes, select `<none>` in the **Regressor 2** list.

4 Fix the values of the regressors that are not displayed by clicking **Fix Values**. In the Fix Regressor Values dialog box, double-click the **Value** cell to edit the constant value of the corresponding regressor. The default values are determined during model estimation. Click **OK**.

If you generate the nonlinear ARX plot in the app, you can perform the following additional tasks:

Action	Procedure
Change the grid spacing of the regressors along each axis.	In the plot window, select **Options > Set number of samples**, and enter the number of samples to use for each regressor. Click **Apply** and then **Close**. For example, if the number of samples is 20, each regressor variable contains 20 points in its specified range. For a 3-D plots, this results in evaluating the nonlinearity at 20 x 20 = 400 points.
Change axis limits.	Select **Options > Set axis limits** to open the Axis Limits dialog box, and edit the limits. Click **Apply**.
Hide or show the plot legend.	Select **Style > Legend**. Select this option again to show the legend.
Rotate in three dimensions. **Note** Available only when you have selected two regressors as independent variables.	Select **Style > Rotate 3D** and drag the axes on the plot to a new orientation. To disable three-dimensional rotation, select **Style > Rotate 3D** again.

See Also

`compare` | `predict` | `resid` | `sim`

More About

- “Identifying Nonlinear ARX Models” on page 11-21

- “Estimate Nonlinear ARX Models in the App” on page 11-29
- “Estimate Nonlinear ARX Models at the Command Line” on page 11-32
- “Using Nonlinear ARX Models” on page 11-55

Using Nonlinear ARX Models

After identifying a nonlinear ARX model, you can use the model for the following tasks:

- **Simulation and prediction** — At the command line, use `sim` and `predict` to simulate and predict the model output. To compare models to measured output and to each other, use `compare`. For information about plotting simulated and predicted output in the app, see "Simulation and Prediction in the App" on page 17-12. You can also specify the initial conditions for simulation and prediction. The toolbox provides several options to facilitate how you specify initial states. For example, you can use `findstates` and `data2state` to compute state values based on the requirement to maximize fit to measured output or based on operating conditions. See the `idnlarx` reference page for a definition of the nonlinear ARX model states. To learn more about how `sim` and `predict` compute the model output, see "How the Software Computes Nonlinear ARX Model Output" on page 11-57.

  You can also forecast the response of a dynamic system by using the `forecast` command. The command predicts future outputs of the system using past output measurements. For more information, see "Forecasting Response of Nonlinear ARX Models" on page 14-47.

- **Linearization** — Compute linear approximation of nonlinear ARX models using `linearize` or `linapp`.

  The `linearize` command provides a first-order Taylor series approximation of the system about an operating point. `linapp` computes a linear approximation of a nonlinear model for a given input data. For more information, see the "Linear Approximation of Nonlinear Black-Box Models" on page 11-64. You can compute the operating point for linearization using `findop`.

  After computing a linear approximation of a nonlinear model, you can perform linear analysis and control design on your model using Control System Toolbox commands. For more information, see "Using Identified Models for Control Design Applications" on page 19-2 and "Create and Plot Identified Models Using Control System Toolbox Software" on page 19-7.

- **Simulation and code generation using Simulink** — You can import estimated nonlinear ARX models into the Simulink software using the Nonlinear ARX block (IDNLARX Model) from the System Identification Toolbox block library. Import the `idnlarx` object from the workspace into Simulink using this block to simulate the model output.

The IDNLARX Model block supports code generation with Simulink Coder™ software, using both generic and embedded targets. Code generation does not work when the model contains `customnet` or `neuralnet` nonlinearity estimator, or custom regressors. For more information, see “Simulating Identified Model Output in Simulink” on page 20-5.

See Also

More About

- “What are Nonlinear ARX Models?” on page 11-17
- “Identifying Nonlinear ARX Models” on page 11-21
- “Linear Approximation of Nonlinear Black-Box Models” on page 11-64

How the Software Computes Nonlinear ARX Model Output

This topic describes how the software evaluates the output of nonlinearity estimators and uses this output to compute the response of a nonlinear ARX model.

Evaluating Nonlinearities

Evaluating the predicted output of a nonlinearity for a specific regressor value x requires that you first extract the nonlinearity F and regressors from the model:

```
F = m.Nonlinearity;
x = getreg(m,'all',data) % computes regressors
```

Evaluate $F(x)$:

```
y = evaluate(F,x)
```

where `x` is a row vector of regressor values.

You can also evaluate predicted output values at multiple time instants by evaluating F for several regressor vectors simultaneously:

```
y = evaluate(F,[x1;x2;x3])
```

Simulation and Prediction of Sigmoid Network

This example shows how the software computes the simulated and predicted output of a nonlinear ARX model as a result of evaluating the output of its nonlinearity estimator for given regressor values.

Estimating and Exploring a Nonlinear ARX Model

Estimate nonlinear ARX model with sigmoid network nonlinearity.

```
load twotankdata
estData = iddata(y,u,0.2,'Tstart',0);
M = nlarx(estData,[1 1 0],'sig');
```

Inspect the model properties and estimation result.

```
present(M)
```

```
M =
Nonlinear ARX model with 1 output and 1 input
 Inputs: u1
 Outputs: y1
 Standard regressors corresponding to the orders
   na = 1, nb = 1, nk = 0
 No custom regressor
 Nonlinear regressors:
   y1(t-1)
   u1(t)
 Nonlinearity: sigmoidnet with 10 units

Sample time: 0.2 seconds

Status:
Termination condition: Maximum number of iterations reached.
Number of iterations: 20, Number of function evaluations: 243

Estimated using NLARX on time domain data "estData".
Fit to estimation data: 96.31% (prediction focus)
FPE: 4.804e-05, MSE: 4.666e-05
More information in model's "Report" property.
```

This command provides information about input and output variables, regressors, and nonlinearity estimator.

Inspect the nonlinearity estimator.

```
NL = M.Nonlinearity; % equivalent to M.nl
class(NL)   % nonlinearity class

ans =
'sigmoidnet'

display(NL) % equivalent to NL

Sigmoid Network:
    NumberOfUnits: 10
       LinearTerm: 'on'
       Parameters: [1x1 struct]
```

Inspect the sigmoid network parameter values.

```
NL.Parameters;
```

Prediction of Output

The model output is:

$y1(t)= f(y1(t-1),u1(t))$

where f is the sigmoid network function. The model regressors $y1(t-1)$ and $u1(t)$ are inputs to the nonlinearity estimator. Time t is a discrete variable representing kT, where `k = 0, 1, ...` , and T is the sampling interval. In this example, `T=0.2` second.

The output prediction equation is:

$yp(t)=f(y1_meas(t-1),u1_meas(t))$

where $yp(t)$ is the predicted value of the response at time t. $y1_meas(t-1)$ and $u1_meas(t)$ are the measured output and input values at times $t-1$ and t, respectively.

Computing the predicted response includes:

- Computing regressor values from input-output data.
- Evaluating the nonlinearity for given regressor values.

To compute the predicted value of the response using initial conditions and current input:

Estimate model from data and get nonlinearity parameters.

```
load twotankdata
estData = iddata(y,u,0.2,'Tstart',0);
M = nlarx(estData,[1 1 0],'sig');
NL = M.Nonlinearity;
```

Specify zero initial states.

```
x0 = 0;
```

The model has one state because there is only one delayed term `y1(t-1)`. The number of states is equal to `sum(getDelayInfo(M))`.

Compute the predicted output at time t=0.

```
RegValue = [0,estData.u(1)]; % input to nonlinear function f
yp_0 = evaluate(NL,RegValue);
```

`RegValue` is the vector of regressors at `t=0` . The predicted output is *yp(t=0)=f(y1_meas(t=-1),u1_meas(t=0))*. In terms of MATLAB variables, this output is `f(0,estData.u(1))`, where

- *y1_meas(t*=0) is the measured output value at `t=0`, which is to `estData.y(1)`.
- *u1_meas(t* =1) is the second input data sample `estData.u(2)`.

Perform one-step-ahead prediction at all time values for which data is available.

```
RegMat = getreg(M,[],estData,x0);
yp = evaluate(NL,RegMat);
```

This code obtains a matrix of regressors `RegMat` for all the time samples using `getreg`. `RegMat` has as many rows as there are time samples, and as many columns as there are regressors in the model - two, in this example.

These steps are equivalent to the predicted response computed in a single step using predict:

```
yp = predict(M,estData,1,'InitialState',x0);
```

Simulation of Output

The model output is:

y1(t)=f(y1(t-1),u1(t))

where *f* is the sigmoid network function. The model regressors *y1(t-1)* and *u1(t)* are inputs to the nonlinearity estimator. Time *t* is a discrete variable representing kT , where k= 0, 1,.., and T is the sampling interval. In this example, T=0.2 second.

The simulated output is:

ys(t) = f(ys(t-1),u1_meas(t))

where *ys(t)* is the simulated value of the response at time *t*. The simulation equation is the same as the prediction equation, except that the past output value `ys(t-1)` results from the simulation at the previous time step, rather than the measured output value.

Computing the simulated response includes:

- Computing regressor values from input-output data using simulated output values.

- Evaluating the nonlinearity for given regressor values.

To compute the simulated value of the response using initial conditions and current input:

Estimate model from data and get nonlinearity parameters.

```
load twotankdata
estData = iddata(y,u,0.2,'Tstart',0);
M = nlarx(estData,[1 1 0],'sig');
NL = M.Nonlinearity;
```

Specify zero initial states.

```
x0 = 0;
```

The model has one state because there is only one delayed term *y1(t-1)*. The number of states is equal to `sum(getDelayInfo(M))`.

Compute the simulated output at time *t* =0, *ys(t=0)*.

```
RegValue = [0,estData.u(1)];
ys_0 = evaluate(NL,RegValue);
```

RegValue is the vector of regressors at *t*=0. *ys(t=0)=f(y1(t=-1),u1_meas(t=0))*. In terms of MATLAB variables, this output is `f(0,estData.u(1))`, where

- *y1(t=-1)* is the initial state `x0 (=0)`.
- *u1_meas(t=0)* is the value of the input at *t* =0, which is the first input data sample `estData.u(1)`.

Compute the simulated output at time *t*=1, *ys(t*=1).

```
RegValue = [ys_0,estData.u(2)];
ys_1 = evaluate(NL,RegValue);
```

The simulated output *ys(t=1)=f(ys(t=0),u1_meas(t=1))*. In terms of MATLAB variables, this output is `f(ys_0,estData.u(2))`, where

- *ys(t=0)* is the simulated value of the output at t=0.
- *u1_meas(t=1)* is the second input data sample estData.u(2).

Compute the simulated output at time *t*=2.

```
RegValue = [ys_1,estData.u(3)];
ys_2 = evaluate(NL,RegValue);
```

Unlike for output prediction, you cannot use `getreg` to compute regressor values for all time values. You must compute regressors values at each time sample separately because the output samples required for forming the regressor vector are available iteratively, one sample at a time.

These steps are equivalent to the simulated response computed in a single step using `sim(idnlarx)`:

```
ys = sim(M,estData,x0);
```

Nonlinearity Evaluation

This examples performs a low-level computation of the nonlinearity response for the `sigmoidnet` network function:

$$F(x) = (x-r)PL + a_1 f((x-r)Qb_1 + c_1) + \ldots$$
$$+ a_n f((x-r)Qb_n + c_n) + d$$

where *f* is the sigmoid function, given by the following equation:

$$f(z) = \frac{1}{e^{-z}+1}$$

In `F(x)`, the input to the sigmoid function is `x-r`. `x` is the regressor value and `r` is regressor mean, computed from the estimation data. a_n , n_n, and c_n are the network parameters stored in the model property `M.nl.par`, where `M` is an `idnlarx` object.

Compute the output value at time t=1, when the regressor values are `x=[estData.y(1),estData.u(2)]`:

Estimate model from sample data.

```
load twotankdata
estData = iddata(y,u,0.2,'Tstart',0);
M = nlarx(estData,[1 1 0],'sig');
NL = M.Nonlinearity;
```

Assign values to the parameters in the expression for `F(x)`.

```
x = [estData.y(1),estData.u(2)]; % regressor values at t=1
r = NL.Parameters.RegressorMean;
P = NL.Parameters.LinearSubspace;
L = NL.Parameters.LinearCoef;
d = NL.Parameters.OutputOffset;
Q = NL.Parameters.NonLinearSubspace;
aVec = NL.Parameters.OutputCoef;   % [a_1; a_2; ...]
cVec = NL.Parameters.Translation;  % [c_1; c_2; ...]
bMat = NL.Parameters.Dilation;     % [b_1; b_2; ...]
```

Compute the linear portion of the response (plus offset).

```
yLinear = (x-r)*P*L+d;
```

Compute the nonlinear portion of the response.

```
f = @(z)1/(exp(-z)+1); % anonymous function for sigmoid unit
yNonlinear = 0;
for k = 1:length(aVec)
    fInput = (x-r)*Q* bMat(:,k)+cVec(k);
    yNonlinear = yNonlinear+aVec(k)*f(fInput);
end
```

Compute total response `y = F(x) = yLinear + yNonlinear`.

```
y = yLinear + yNonlinear;
```

`y` is equal to `evaluate(NL,x)`.

See Also

`evaluate`

More About

- "What are Nonlinear ARX Models?" on page 11-17
- "Identifying Nonlinear ARX Models" on page 11-21

Linear Approximation of Nonlinear Black-Box Models

In this section...
“Why Compute a Linear Approximation of a Nonlinear Model?” on page 11-64
“Choosing Your Linear Approximation Approach” on page 11-64
“Linear Approximation of Nonlinear Black-Box Models for a Given Input” on page 11-64
“Tangent Linearization of Nonlinear Black-Box Models” on page 11-65
“Computing Operating Points for Nonlinear Black-Box Models” on page 11-66

Why Compute a Linear Approximation of a Nonlinear Model?

Control design and linear analysis techniques using Control System Toolbox software require linear models. You can use your estimated nonlinear model in these applications after you linear the model. After you linearize your model, you can use the model for control design and linear analysis.

Choosing Your Linear Approximation Approach

System Identification Toolbox software provides two approaches for computing a linear approximation of nonlinear ARX on page 11-21 and Hammerstein-Wiener on page 12-5 models.

To compute a linear approximation of a nonlinear model for a given input signal, use the `linapp` command. The resulting model is only valid for the same input that you use to compute the linear approximation. For more information, see “Linear Approximation of Nonlinear Black-Box Models for a Given Input” on page 11-64.

If you want a tangent approximation of the nonlinear dynamics that is accurate near the system operating point, use the `linearize` command. The resulting model is a first-order Taylor series approximation for the system about the operating point, which is defined by a constant input and model state values. For more information, see “Tangent Linearization of Nonlinear Black-Box Models” on page 11-65.

Linear Approximation of Nonlinear Black-Box Models for a Given Input

`linapp` computes the best linear approximation, in a mean-square-error sense, of a nonlinear ARX or Hammerstein-Wiener model for a given input or a randomly generated

input. The resulting linear model might only be valid for the same input signal as you the one you used to generate the linear approximation.

`linapp` estimates the best linear model that is structurally similar to the original nonlinear model and provides the best fit between a given input and the corresponding simulated response of the nonlinear model.

To compute a linear approximation of a nonlinear black-box model for a given input, you must have these variables:

- Nonlinear ARX model (`idnlarx` object) or Hammerstein-Wiener model (`idnlhw` object)
- Input signal for which you want to obtain a linear approximation, specified as a real matrix or an `iddata` object

`linapp` uses the specified input signal to compute a linear approximation:

- For nonlinear ARX models, `linapp` estimates a linear ARX model using the same model orders `na`, `nb`, and `nk` as the original model.
- For Hammerstein-Wiener models, `linapp` estimates a linear Output-Error (OE) model using the same model orders `nb`, `nf`, and `nk`.

To compute a linear approximation of a nonlinear black-box model for a randomly generated input, you must specify the minimum and maximum input values for generating white-noise input with a magnitude in this rectangular range, `umin` and `umax`.

For more information, see the `linapp` reference page.

Tangent Linearization of Nonlinear Black-Box Models

`linearize` computes a first-order Taylor series approximation for nonlinear system dynamics about an *operating point*, which is defined by a constant input and model state values. The resulting linear model is accurate in the local neighborhood of this operating point.

To compute a tangent linear approximation of a nonlinear black-box model, you must have these variables:

- Nonlinear ARX model (`idnlarx` object) or Hammerstein-Wiener model (`idnlhw` object)

- Operating point

To specify the operating point of your system, you must specify the constant input and the states. For more information about state definitions for each type of parametric model, see these reference pages:

- `idnlarx` — Nonlinear ARX model
- `idnlhw` — Nonlinear Hammerstein-Wiener model

If you do not know the operating point values for your system, see "Computing Operating Points for Nonlinear Black-Box Models" on page 11-66.

For more information, see the `idnlarx/linearize` or `idnlhw/linearize` reference page.

Computing Operating Points for Nonlinear Black-Box Models

An *operating point* is defined by a constant input and model state values.

If you do not know the operating conditions of your system for linearization, you can use `findop` to compute the operating point from specifications:

- "Computing Operating Point from Steady-State Specifications" on page 11-66
- "Computing Operating Points at a Simulation Snapshot" on page 11-67

Computing Operating Point from Steady-State Specifications

Use `findop` to compute an operating point from steady-state specifications:

- Values of input and output signals.
  If either the steady-state input or output value is unknown, you can specify it as `NaN` to estimate this value. This is especially useful when modeling MIMO systems, where only a subset of the input and output steady-state values are known.
- More complex steady-state specifications.

  Construct an object that stores specifications for computing the operating point, including input and output bounds, known values, and initial guesses. For more information, see `idnlarx/operspec` or `idnlhw/operspec`.

For more information, see the `idnlarx/findop` or `idnlhw/findop` reference page.

Computing Operating Points at a Simulation Snapshot

Compute an operating point at a specific time during model simulation (snapshot) by specifying the snapshot time and the input value. To use this method for computing the equilibrium operating point, choose an input that leads to a steady-state output value. Use that input and the time value at which the output reaches steady state (*snapshot* time) to compute the operating point.

It is optional to specify the initial conditions for simulation when using this method because initial conditions often do not affect the steady-state values. By default, the initial conditions are zero.

However, for nonlinear ARX models, the steady-state output value might depend on initial conditions. For these models, you should investigate the effect of initial conditions on model response and use the values that produce the desired output. You can use `data2state` to map the input-output signal values from before the simulation starts to the model's initial states. Because the initial states are a function of the past history of the model's input and output values, `data2state` generates the initial states by transforming the data.

See Also

`idnlarx/linearize` | `idnlhw/linearize`

More About

- “Using Nonlinear ARX Models” on page 11-55
- “Using Hammerstein-Wiener Models” on page 12-11

Identify Hammerstein-Wiener Models

- "What are Hammerstein-Wiener Models?" on page 12-2
- "Identifying Hammerstein-Wiener Models" on page 12-5
- "Using Hammerstein-Wiener Models" on page 12-11
- "Available Nonlinearity Estimators for Hammerstein-Wiener Models" on page 12-13
- "Estimate Hammerstein-Wiener Models in the App" on page 12-15
- "Estimate Hammerstein-Wiener Models at the Command Line" on page 12-18
- "Validating Hammerstein-Wiener Models" on page 12-27
- "How the Software Computes Hammerstein-Wiener Model Output" on page 12-33
- "Estimate Hammerstein-Wiener Models Initialized Using Linear OE Models" on page 12-37

What are Hammerstein-Wiener Models?

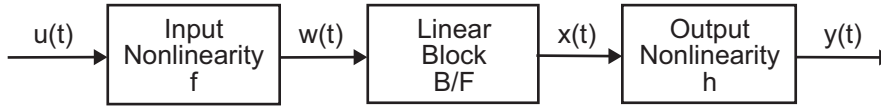
When the output of a system depends nonlinearly on its inputs, sometimes it is possible to decompose the input-output relationship into two or more interconnected elements. In this case, you can represent the dynamics by a linear transfer function and capture the nonlinearities using nonlinear functions of inputs and outputs of the linear system. The Hammerstein-Wiener model achieves this configuration as a series connection of static nonlinear blocks with a dynamic linear block. Hammerstein-Wiener model applications span several areas, such as modeling electromechanical system and radio frequency components, audio and speech processing, and predictive control of chemical processes. These models have a convenient block representation, a transparent relationship to linear systems, and are easier to implement than heavy-duty nonlinear models such as neural networks and Volterra models.

You can use a Hammerstein-Wiener model as a black-box model structure because it provides a flexible parameterization for nonlinear models. For example, you can estimate a linear model and try to improve its fidelity by adding an input or output nonlinearity to this model. You can also use a Hammerstein-Wiener model as a grey-box structure to capture physical knowledge about process characteristics. For example, the input nonlinearity can represent typical physical transformations in actuators and the output nonlinearity can describe common sensor characteristics. For more information about when to fit nonlinear models, see “About Identified Nonlinear Models” on page 11-2.

Structure of Hammerstein-Wiener Models

Hammerstein-Wiener models describe dynamic systems using one or two static nonlinear blocks in series with a linear block. The linear block is a discrete transfer function that represents the dynamic component of the model.

This block diagram represents the structure of a Hammerstein-Wiener model:



Where,

- f is a nonlinear function that transforms input data $u(t)$ as $w(t) = f(u(t))$.

  $w(t)$, an internal variable, is the output of the Input Nonlinearity block and has the same dimension as $u(t)$.

- B/F is a linear transfer function that transforms $w(t)$ as $x(t) = (B/F)w(t)$.

  $x(t)$, an internal variable, is the output of the Linear block and has the same dimension as $y(t)$.

  B and F are similar to polynomials in a linear Output-Error model. For more information about Output-Error models, see "What Are Polynomial Models?" on page 6-2.

  For ny outputs and nu inputs, the linear block is a transfer function matrix containing entries:

  $$\frac{B_{j,i}(q)}{F_{j,i}(q)}$$

  where $j = 1, 2, \ldots, ny$ and $i = 1, 2, \ldots, nu$.
- h is a nonlinear function that maps the output of the linear block $x(t)$ to the system output $y(t)$ as $y(t) = h(x(t))$.

Because f acts on the input port of the linear block, this function is called the *input nonlinearity*. Similarly, because h acts on the output port of the linear block, this function is called the *output nonlinearity*. If your system contains several inputs and outputs, you must define the functions f and h for each input and output signal. You do not have to include both the input and the output nonlinearity in the model structure. When a model contains only the input nonlinearity f, it is called a Hammerstein model. Similarly, when the model contains only the output nonlinearity h, it is called a *Wiener* model.

The software computes the Hammerstein-Wiener model output y in three stages:

1. Compute $w(t) = f(u(t))$ from the input data.

   $w(t)$ is an input to the linear transfer function B/F.

   The input nonlinearity is a static (*memoryless*) function, where the value of the output a given time t depends only on the input value at time t.

   You can configure the input nonlinearity as a sigmoid network, wavelet network, saturation, dead zone, piecewise linear function, one-dimensional polynomial, or a custom network. You can also remove the input nonlinearity.
2. Compute the output of the linear block using $w(t)$ and initial conditions: $x(t) = (B/F)w(t)$.

You can configure the linear block by specifying the orders of numerator B and denominator F.

3 Compute the model output by transforming the output of the linear block $x(t)$ using the nonlinear function h as $y(t) = h(x(t))$.

Similar to the input nonlinearity, the output nonlinearity is a static function. You can configure the output nonlinearity in the same way as the input nonlinearity. You can also remove the output nonlinearity, such that $y(t) = x(t)$.

Resulting models are `idnlhw` objects that store all model data, including model parameters and nonlinearity estimators. For more information about these objects, see "Nonlinear Model Structures" on page 11-7.

You can estimate Hammerstein-Wiener models in the **System Identification** app or at the command line using the `nlhw` command. You can use uniformly sampled time-domain input-output data for estimating Hammerstein-Wiener models. Your data can have one or more input and output channels. You cannot use time series data (output only) or frequency-domain data for estimation. If you have time series data, to fit a nonlinear model, identify nonlinear ARX models or nonlinear grey-box models. For more information about these models, see "Identifying Nonlinear ARX Models" on page 11-21 and "Estimate Nonlinear Grey-Box Models" on page 13-35.

See Also

`idnlhw` | `nlhw`

More About

- "About Identified Nonlinear Models" on page 11-2
- "Identifying Hammerstein-Wiener Models" on page 12-5
- "Available Nonlinearity Estimators for Hammerstein-Wiener Models" on page 12-13
- "Using Hammerstein-Wiener Models" on page 12-11
- "How the Software Computes Hammerstein-Wiener Model Output" on page 12-33

Identifying Hammerstein-Wiener Models

In this section...
“Prepare Data for Identification” on page 12-5
“Configure Hammerstein-Wiener Model Structure” on page 12-6
“Specify Estimation Options for Hammerstein-Wiener Models” on page 12-7
“Initialize Hammerstein-Wiener Estimation Using Linear Model” on page 12-8

Hammerstein-Wiener models describe dynamic systems using one or two static nonlinear blocks in series with a linear block. The linear block is a discrete transfer function and represents the dynamic component of the model. For more information about the structure of these models, see “What are Hammerstein-Wiener Models?” on page 12-2

You can estimate Hammerstein-Wiener models in the **System Identification** app or at the command line using the `nlhw` command. To estimate a Hammerstein-Wiener model, you first prepare the estimation data. You then configure the model structure and estimation algorithm, and then perform estimation. After estimation, you can validate the estimated model as described in “Validating Hammerstein-Wiener Models” on page 12-27.

Prepare Data for Identification

You can use only uniformly sampled time-domain input-output data for estimating Hammerstein-Wiener models. Your data can have one or more input and output channels. You cannot use time series data (output only) or frequency-domain data for estimation. Use nonlinear ARX on page 11-21 or nonlinear grey-box models on page 13-35 for time series data.

To prepare the data for model estimation, import your data into the MATLAB workspace, and do *one* of the following:

- **In the System Identification app** — Import data into the app, as described in “Represent Data”.
- **At the command line** — Represent your data as an `iddata` object.

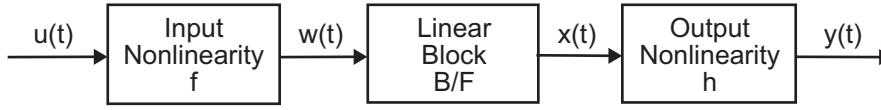
After importing the data, you can analyze data quality and preprocess data by interpolating missing values, filtering to emphasize a specific frequency range, or resampling using a different sample time. For more information, see “Ways to Prepare

Data for System Identification" on page 2-6. For most applications, you do not need to remove offsets and linear trends from the data before nonlinear modeling. However, data detrending can be useful in some cases, such as before modeling the relationship between the change in input and output about an operating point.

After preparing your estimation data, you can configure your model structure, loss function, and estimation algorithm, and then estimate the model using the estimation data.

Configure Hammerstein-Wiener Model Structure

The Hammerstein-Wiener model structure consists of input and output nonlinear blocks in series with a linear block. The linear block is a discrete transfer function and represents the dynamic component of the model.



To configure the structure of a Hammerstein-Wiener model:

1 Configure the linear transfer function block.

Perform *one* of the following:

- Specify model order and input delay for the linear transfer function as:
  - *nb* — Number of zeros plus one. *nb* is the length of the numerator (*B*) polynomial.
  - *nf* — Number of poles. *nf* is the order of the transfer function denominator (*F* polynomial).
  - *nk* — Delay from input to the output in terms of the number of samples.

  For MIMO systems with *Ny* outputs and *Nu* inputs, *nb*, *nf*, and *nk* are *Ny*-by-*Nu* matrices.
- Initialize the linear block using a discrete-time linear model — You can initialize using linear models at the command line only. The initialization sets the transfer function of the linear block to that of the specified linear model. For more information, see "Initialize Hammerstein-Wiener Estimation Using Linear Model" on page 12-8.

2 Configure the input and output nonlinearities, *f* and *h* respectively.

The default input and output nonlinearity estimators are piecewise linear functions. See the `pwlinear` reference page for more information. To configure the input and output nonlinearity estimators:

a Choose the type of input and output nonlinearity estimators, and configure their properties.

For a list of available nonlinearity estimators, see "Available Nonlinearity Estimators for Hammerstein-Wiener Models" on page 12-13.

b Exclude the input or output nonlinear block.

You do not have to include both the input and the output nonlinearity in the model structure. When a model contains only the input nonlinearity *f*, it is called a Hammerstein model. Similarly, when the model contains only the output nonlinearity *h*, it is called a *Wiener* model.

For information about how to configure the model structure at the command line and in the app, see "Estimate Hammerstein-Wiener Models at the Command Line" on page 12-18 and "Estimate Hammerstein-Wiener Models in the App" on page 12-15.

Specify Estimation Options for Hammerstein-Wiener Models

To configure the model estimation, specify the loss function to be minimized, and choose the estimation algorithm and other estimation options to perform the minimization.

Configure Loss Function

The loss function or cost function is a function of the error between the model output and the measured output. For more information about loss functions, see "Loss Function and Model Quality Metrics" on page 1-64.

At the command line, use the `nlhw` option set, `nlhwOptions` to configure your loss function. You can specify the following options:

- `OutputWeight` — Specify a weighting of the error in multi-output estimations.
- `Regularization` — Modify the loss function to add a penalty on the variance of the estimated parameters. For more information, see "Regularized Estimates of Model Parameters" on page 1-48.

For details about how to specify these options in the app, see “Estimate Hammerstein-Wiener Models in the App” on page 12-15.

Specify Estimation Algorithm

To estimate a Hammerstein-Wiener model, the software uses iterative search algorithms to minimize the loss function. At the command line, use `nlhwOptions` to specify the search algorithm and other estimation options. Some of the options you can specify are:

- `SearchMethod` — Search method for minimization of prediction or simulation errors, such as Gauss-Newton and Levenberg-Marquardt line search, and Trust-region reflective Newton approach.
- `SearchOption` — Option set for the search algorithm, with fields that depend on the value of `SearchMethod`, such as:
  - `MaxIter` — Maximum number of iterations to perform.
  - `Tolerance` — Condition for terminating iterative search when the expected improvement of the parameter values is less than a specified value.
- `InitialCondition` — By default, the software treats the initial states of the model as zero and does not estimate the states. You can choose to estimate initial states, which sometimes can improve parameter estimates.

To see a complete list of available estimation options, see `nlhwOptions`. For details about how to specify these estimation options in the app, see “Estimate Hammerstein-Wiener Models in the App” on page 12-15.

After preprocessing the estimation data and configuring the model structure, loss function, and estimation options, you can estimate the model in the **System Identification** app, or using `nlhw`. The resulting model is an `idnlhw` object that stores all model data, including model parameters and nonlinearity estimator. For more information about these model objects, see “Nonlinear Model Structures” on page 11-7. You can validate the estimated model as described in “Validating Hammerstein-Wiener Models” on page 12-27.

Initialize Hammerstein-Wiener Estimation Using Linear Model

At the command line, you can use one of the following linear models to initialize the linear block of a Hammerstein-Wiener model:

- Polynomial model of Output-Error (OE) structure (`idpoly`)
- State-space model with no disturbance component (`idss` model with K = 0)
- Transfer function (`idtf` model)

Typically, you use the `oe`, `n4sid`, or `tfest` commands to obtain the linear model. You can provide the linear model when constructing or estimating a Hammerstein-Wiener model. For example, use the following syntax to estimate a Hammerstein-Wiener model using estimation data and a linear model `LinModel`.

```
m = nlhw(data,LinModel)
```

Here `m` is an `idnlhw` object, and `data` is a time-domain `iddata` object. The software uses the linear model for initializing the Hammerstein-Wiener estimation by:

- Assigning the linear model orders as initial values of nonlinear model orders (`nb` and `nf` properties of the Hammerstein-Wiener (`idnlhw`) and delays (`nk` property).
- Setting the *B* and *F* polynomials of the linear transfer function in the Hammerstein-Wiener model structure on page 12-2.

During estimation, the estimation algorithm uses these values to adjust the nonlinear model to the data. By default, both the input and output nonlinearity estimators are piecewise linear functions (see `pwlinear`).

You can also specify different input and output nonlinearity estimators. For example, a sigmoid network input nonlinearity estimator and a dead-zone output nonlinearity estimator.

```
m = nlhw(data,LinModel,'sigmoidnet','deadzone')
```

For an example, see "Estimate Hammerstein-Wiener Models Initialized Using Linear OE Models" on page 12-37.

See Also

More About

- "What are Hammerstein-Wiener Models?" on page 12-2
- "Estimate Hammerstein-Wiener Models in the App" on page 12-15

- "Estimate Hammerstein-Wiener Models at the Command Line" on page 12-18
- "Validating Hammerstein-Wiener Models" on page 12-27
- "Using Hammerstein-Wiener Models" on page 12-11

Using Hammerstein-Wiener Models

After identifying a Hammerstein-Wiener model, you can use the model for the following tasks:

- **Simulation** — At the command line, use `sim` to simulate the model output. To compare models to measured output and to each other, use `compare`. Note that for Hammerstein-Wiener models, the simulated and predicted model output are equivalent because these models have a trivial noise component, that is disturbance in these models is white noise. For information about plotting simulated output in the app, see "Simulation and Prediction in the App" on page 17-12.

  You can also specify the initial conditions for simulation. The toolbox provides various options to facilitate how you specify initial states. For example, you can use `findstates` to automatically search for state values in simulation and prediction applications. You can also specify the states manually. See the `idnlhw` reference page for a definition of the Hammerstein-Wiener model states.

  To learn more about how `sim` computes the model output, see "How the Software Computes Hammerstein-Wiener Model Output" on page 12-33.

- **Linearization** — Compute linear approximation of Hammerstein-Wiener models using `linearize` or `linapp`.

  The `linearize` command provides a first-order Taylor series approximation of the system about an operating point. `linapp` computes a linear approximation of a nonlinear model for a given input data. For more information, see the "Linear Approximation of Nonlinear Black-Box Models" on page 11-64. You can compute the operating point for linearization using `findop`.

  After computing a linear approximation of a nonlinear model, you can perform linear analysis and control design on your model using Control System Toolbox commands. For more information, see "Using Identified Models for Control Design Applications" on page 19-2 and "Create and Plot Identified Models Using Control System Toolbox Software" on page 19-7.

- **Simulation and code generation using Simulink** — You can import the estimated Hammerstein-Wiener model into Simulink software using the Hammerstein-Wiener block (IDNLHW Model) from the System Identification Toolbox block library. After you bring the `idnlhw` object from the workspace into Simulink, you can simulate the model output.

The IDNLHW Model block supports code generation with Simulink Coder software, using both generic and embedded targets. Code generation does not work when the model contains `customnet` as the input or output nonlinearity. For more information, see “Simulating Identified Model Output in Simulink” on page 20-5.

See Also

More About

- “What are Hammerstein-Wiener Models?” on page 12-2
- “Identifying Hammerstein-Wiener Models” on page 12-5
- “Linear Approximation of Nonlinear Black-Box Models” on page 11-64

Available Nonlinearity Estimators for Hammerstein-Wiener Models

System Identification Toolbox software provides several scalar nonlinearity estimators, for Hammerstein-Wiener models. The nonlinearity estimators are available for both the input and output nonlinearities *f* and *h*, respectively. For more information about *f* and *h*, see "Structure of Hammerstein-Wiener Models" on page 12-2.

Each nonlinearity estimator corresponds to an object class in this toolbox. When you estimate Hammerstein-Wiener models in the **System Identification** app, the toolbox creates and configures objects based on these classes. You can also create and configure nonlinearity estimators at the command line. For a detailed description of each estimator, see the references page of the corresponding nonlinearity class.

Nonlinearity	Class	Structure	Comments
Piecewise linear (default)	`pwlinear`	A piecewise linear function parameterized by breakpoint locations.	By default, the number of breakpoints is 10.
One layer sigmoid network	`sigmoidnet`	$g(x)=\sum_{k=1}^{n}\alpha_k\kappa\left(\beta_k\left(x-\gamma_k\right)\right)$ $\kappa(s)$ is the sigmoid function $\kappa(s)=\left(e^s+1\right)^{-1}$. β_k is a row vector such that $\beta_k(x-\gamma_k)$ is a scalar.	Default number of units *n* is 10.
Wavelet network	`wavenet`	$g(x)=\sum_{k=1}^{n}\alpha_k\kappa\left(\beta_k\left(x-\gamma_k\right)\right)$ where $\kappa(s)$ is the wavelet function.	By default, the estimation algorithm determines the number of units *n* automatically.
Saturation	`saturation`	Parameterize hard limits on the signal value as upper and lower saturation limits.	Use to model known saturation effects on signal amplitudes.

Nonlinearity	Class	Structure	Comments
Dead zone	`deadzone`	Parameterize dead zones in signals as the duration of zero response.	Use to model known dead zones in signal amplitudes.
One-dimensional polynomial	`poly1d`	Single-variable polynomial of a degree that you specify.	By default, the polynomial degree is 1.
Unit gain	`unitgain`	Excludes the input or output nonlinearity from the model structure to achieve a Wiener or Hammerstein configuration, respectively. **Note** Excluding both the input and output nonlinearities reduces the Hammerstein-Wiener structure to a linear transfer function.	Useful for configuring multi-input, multi-output (MIMO) models to exclude nonlinearities from specific input and output channels.
Custom network (user-defined)	`customnet`	Similar to sigmoid network but you specify $\kappa(s)$.	(For advanced use) Uses the unit function that you specify.

See Also

More About

- "What are Hammerstein-Wiener Models?" on page 12-2
- "Identifying Hammerstein-Wiener Models" on page 12-5

Estimate Hammerstein-Wiener Models in the App

You can estimate Hammerstein-Wiener models in the **System Identification** app after performing the following tasks:

- Import data into the System Identification app (see "Preparing Data for Nonlinear Identification" on page 11-15).
- (Optional) Choose a nonlinearity estimator in "Available Nonlinearity Estimators for Hammerstein-Wiener Models" on page 12-13.
- (Optional) Estimate or construct an OE or state-space linear model to use for initialization. See "Initialize Hammerstein-Wiener Estimation Using Linear Model" on page 12-8.

To estimate a Hammerstein-Wiener model using the imported estimation data, chosen nonlinearity estimators, and initial linear models:

1. In the **System Identification** app, select **Estimate** > **Nonlinear models** to open the Nonlinear Models dialog box.
2. In the **Configure** tab, select `Hammerstein-Wiener` from the **Model type** list.
3. (Optional) Edit the **Model name** by clicking the pencil icon. The name of the model should be unique to all Hammerstein-Wiener models in the System Identification app.
4. (Optional) If you want to refine a previously estimated model, click **Initialize** to select a previously estimated model from the **Initial Model** list.

   Note Refining a previously estimated model starts with the parameter values of the initial model and uses the same model structure. You can change these settings.

   The **Initial Model** list includes models that:

   - Exist in the System Identification app.
   - Have the same number of inputs and outputs as the dimensions of the estimation data (selected as **Working Data** in the System Identification app).
5. Keep the default settings in the Nonlinear Models dialog box that specify the model structure, or modify these settings:

Note For more information about available options, click **Help** in the Nonlinear Models dialog box to open the app help.

What to Configure	Options in Nonlinear Models GUI	Comment
Input or output nonlinearity	In the **I/O Nonlinearity** tab, select the **Nonlinearity** and specify the **No. of Units**.	If you do not know which nonlinearity to try, use the (default) piecewise linear nonlinearity. When you estimate from binary input data, you cannot reliably estimate the input nonlinearity. In this case, set **Nonlinearity** for the input channel to `None`. For multiple-input and multiple-output systems, you can assign nonlinearities to specific input and output channels.
Model order and delay	In the **Linear Block** tab, specify **B Order**, **F Order**, and **Input Delay**. For MIMO systems, select the output channel and specify the orders and delays from each input channel.	If you do not know the input delay values, click **Infer Input Delay**. This action opens the Infer Input Delay dialog box which suggests possible delay values.
Estimation algorithm	In the **Estimate** tab, click **Estimation Options**.	You can specify to estimate initial states.

6 To obtain regularized estimates of model parameters, in the **Estimate** tab, click **Estimation Options**. Specify the regularization constants in the **Regularization_Tradeoff_Constant** and **Regularization_Weighting** fields. To learn more, see “Regularized Estimates of Model Parameters” on page 1-48.

7 Click **Estimate** to add this model to the System Identification app.

The **Estimate** tab displays the estimation progress and results.

8 Validate the model response by selecting the desired plot in the **Model Views** area of the System Identification app.

If you get a poor fit, try changing the model structure or algorithm configuration in step 5.

You can export the estimated model to the MATLAB workspace by dragging it to **To Workspace** in the System Identification app.

See Also

More About

- "What are Hammerstein-Wiener Models?" on page 12-2
- "Available Nonlinearity Estimators for Hammerstein-Wiener Models" on page 12-13
- "Identifying Hammerstein-Wiener Models" on page 12-5
- "Validating Hammerstein-Wiener Models" on page 12-27

Estimate Hammerstein-Wiener Models at the Command Line

You can estimate Hammerstein-Wiener models after performing the following tasks:

- Prepare your data, as described in "Preparing Data for Nonlinear Identification" on page 11-15.
- (Optional) Choose a nonlinearity estimator in "Available Nonlinearity Estimators for Hammerstein-Wiener Models" on page 12-13.
- (Optional) Estimate or construct an input-output polynomial model of Output-Error (OE) structure (`idpoly`) or a state-space model with no disturbance component (`idss` with K=0) for initialization of Hammerstein-Wiener model. See "Initialize Hammerstein-Wiener Estimation Using Linear Model" on page 12-8.

Estimate Model Using nlhw

Use `nlhw` to both construct and estimate a Hammerstein-Wiener model. After each estimation, validate the model on page 12-27 by comparing it to other models and simulating or predicting the model response.

Basic Estimation

Start with the simplest estimation using `m = nlhw(data,[nb nf nk])`. For example:

```
load iddata3;
% nb = nf = 2 and nk = 1
m = nlhw(z3,[2 2 1])

m =
Hammerstein-Wiener model with 1 output and 1 input
 Linear transfer function corresponding to the orders  nb = 2, nf = 2, nk = 1
 Input nonlinearity: pwlinear with 10 units
 Output nonlinearity: pwlinear with 10 units
Sample time: 1 seconds

Status:
Estimated using NLHW on time domain data "z3".
Fit to estimation data: 75.31%
FPE: 2.019, MSE: 1.472
```

The second input argument [*nb nf nk*] sets the order of the linear transfer function, where *nb* is the number of zeros plus 1, *nf* is the number of poles, and *nk* is the input

delay. By default, both the input and output nonlinearity estimators are piecewise linear functions (see the `pwlinear` reference page). `m` is an `idnlhw` object.

For MIMO systems, *nb*, *nf*, and *nk* are *ny*-by-*nu* matrices. See the `nlhw` reference page for more information about MIMO estimation.

Configure Nonlinearity Estimators

You can specify a different nonlinearity estimator than the default piecewise linear estimators.

```
m = nlhw(data,[nb,nf,nk],InputNL,OutputNL)
```

`InputNL` and `OutputNL` are nonlinearity estimator objects. If your input signal is binary, set `InputNL` to `unitgain`.

To use nonlinearity estimators with default settings, specify `InputNL` and `OutputNL` using character vectors (such as `'wavenet'` for wavelet network or `'sigmoidnet'` for sigmoid network).

```
load iddata3;
m = nlhw(z3,[2 2 1],'sigmoidnet','deadzone');
```

If you need to configure the properties of a nonlinearity estimator, use its object representation. For example, to estimate a Hammerstein-Wiener model that uses saturation as its input nonlinearity and one-dimensional polynomial of degree 3 as its output nonlinearity:

```
m = nlhw(z3,[2 2 1],'saturation',poly1d('Degree',3));
```

The third input `'saturation'` specifies the saturation nonlinearity with default property values. `poly1d('Degree',3)` creates a one-dimensional polynomial object of degree 3.

For MIMO models, specify the nonlinearities using objects unless you want to use the same nonlinearity with default configuration for all channels.

This table summarizes values that specify the nonlinearity estimators.

Nonlinearity	Value (Default Nonlinearity Configuration)	Class
Piecewise linear (default)	`'pwlinear'`	`pwlinear`
One layer sigmoid network	`'sigmoidnet'`	`sigmoidnet`
Wavelet network	`'wavenet'`	`wavenet`
Saturation	`'saturation'`	`saturation`
Dead zone	`'deadzone'`	`deadzone`
One-dimensional polynomial	`'poly1d'`	`poly1d`
Unit gain	`'unitgain'` or `[ ]`	`unitgain`

Additional available nonlinearities include custom networks that you create. Specify a custom network by defining a function called `gaussunit.m`, as described in the `customnet` reference page. Define the custom network object `CNetw` as:

For more information, see “Available Nonlinearity Estimators for Hammerstein-Wiener Models” on page 12-13.

Exclude Input or Output Nonlinearity

Exclude a nonlinearity for a specific channel by specifying the `unitgain` value for the `InputNonlinearity` or `OutputNonlinearity` properties.

If the input signal is binary, set *`InputNL`* to `unitgain`.

For more information about model estimation and properties, see the `nlhw` and `idnlhw` reference pages.

For a description of each nonlinearity estimator, see “Available Nonlinearity Estimators for Hammerstein-Wiener Models” on page 12-13.

Iteratively Refine Model

Estimate a Hammerstein-Wiener model and then use `nlhw` command to iteratively refine the model.

```
load iddata3;
m1 = nlhw(z3,[2 2 1],'sigmoidnet','wavenet');
m2 = nlhw(z3,m1);
```

Alternatively, use `pem` to refine the model.

```
m2 = pem(z3,m1);
```

Check the search termination criterion in `m.Report.Termination.WhyStop`. If `WhyStop` indicates that the estimation reached the maximum number of iterations, try repeating the estimation and possibly specifying a larger value for the `MaxIter`.

Run 30 more iterations starting at model `m1`.

```
opt = nlhwOptions;
opt.SearchOption.MaxIter = 30;
m2 = nlhw(z3,m1,opt);
```

When the `m.Report.Termination.WhyStop` value is `Near (local) minimum, (norm(g) < tol` or `No improvement along the search direction with line search`, validate your model to see if this model adequately fits the data. If not, the solution might be stuck in a local minimum of the cost-function surface. Try adjusting the `SearchOption.Tolerance` or the `SearchMethod` option of the `nlhw` option set, and repeat the estimation.

You can also try perturbing the parameters of the last model using `init`, and then refine the model using `nlhw` command.

Randomly perturb parameters of original model `m1` about nominal values.

```
m1p = init(m1);
```

Estimate the parameters of perturbed model.

```
M2 = nlhw(z3,m1p);
```

Note that using `init` does not guarantee a better solution on further refinement.

Improve Estimation Results Using Initial States

If your estimated Hammerstein-Wiener model provides a poor fit to measured data, you can repeat the estimation using the initial state values estimated from the data. By

default, the initial states corresponding to the linear block of the Hammerstein-Wiener model are zero.

To specify estimating initial states during model estimation:

```
load iddata3;
opt = nlhwOptions('InitialCondition', 'estimate');
m = nlhw(z3,[2 2 1],sigmoidnet,[],opt);
```

Troubleshoot Estimation

If you do not get a satisfactory model after many trials with various model structures and estimation options, it is possible that the data is poor. For example, your data might be missing important input or output variables and does not sufficiently cover all the operating points of the system.

Nonlinear black-box system identification usually requires more data than linear model identification to gain enough information about the system. See also “Troubleshooting Model Estimation” on page 17-119.

Estimate Multiple Hammerstein-Wiener Models

This example shows how to estimate and compare multiple Hammerstein-Wiener models using measured input-output data.

Load estimation and validation data.

```
load twotankdata
z = iddata(y,u,0.2);
ze = z(1:1000);
zv = z(1001:3000);
```

Estimate several models using the estimation data `ze` and different model orders, delays, and nonlinearity settings.

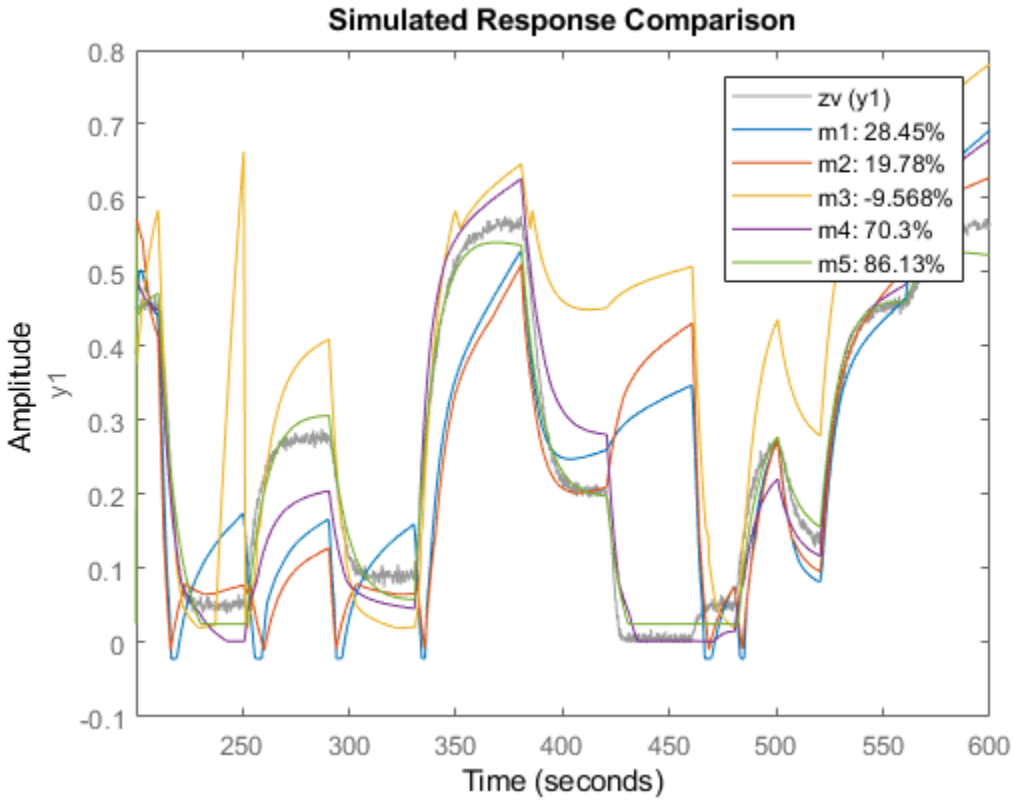
```
m1 = nlhw(ze,[2 3 1]);
m2 = nlhw(ze,[2 2 3]);
m3 = nlhw(ze,[2 2 3],pwlinear('NumberofUnits',13),pwlinear('NumberofUnits',10));
m4 = nlhw(ze,[2 2 3],sigmoidnet('NumberofUnits',2),pwlinear('NumberofUnits',10));
```

An alternative way to perform the estimation is to configure the model structure first using `idnlhw`, and then estimate the model.

```
m5 = idnlhw([2 2 3],'deadzone','saturation');
m5 = nlhw(ze,m5);
```

Compare the resulting models by plotting the model outputs and the measured output in validation data `zv`.

```
compare(zv,m1,m2,m3,m4,m5)
```



Improve a Linear Model Using Hammerstein-Wiener Structure

This example shows how to use the Hammerstein-Wiener model structure to improve a previously estimated linear model.

After estimating the linear model, insert it into the Hammerstein-Wiener structure that includes input or output nonlinearities.

Estimate a linear model.

```
load iddata1
LM = arx(z1,[2 2 1]);
```

Extract the transfer function coefficients from the linear model.

```
[Num,Den] = tfdata(LM);
```

Create a Hammerstein-Wiener model, where you initialize the linear block properties `B` and `F` using `Num` and `Den`, respectively.

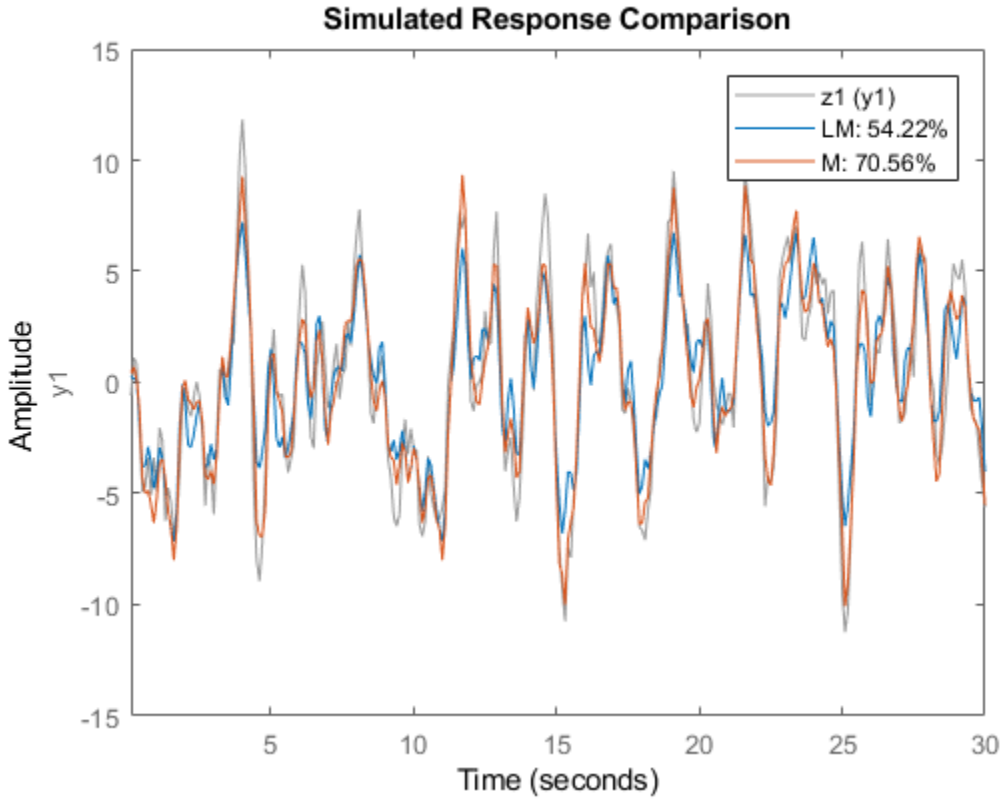
```
nb = 1;        % In general, nb = ones(ny,nu)
               % ny is number of outputs and nu is number of inputs
nf = nb;
nk = 0;        % In general, nk = zeros(ny,nu)
               % ny is number of outputs and nu is number of inputs
M = idnlhw([nb nf nk],[],'pwlinear');
M.B = Num;
M.F = Den;
```

Estimate the model coefficients, which refines the linear model coefficients in `Num` and `Den` .

```
M = nlhw(z1,M);
```

Compare responses of linear and nonlinear model against measured data.

```
compare(z1,LM,M);
```



See Also

More About

- “What are Hammerstein-Wiener Models?” on page 12-2
- “Identifying Hammerstein-Wiener Models” on page 12-5
- “Validating Hammerstein-Wiener Models” on page 12-27
- “Estimate Hammerstein-Wiener Models Initialized Using Linear OE Models” on page 12-37

- “Identifying Nonlinear ARX and Hammerstein-Wiener Models Using Measured Data”

Validating Hammerstein-Wiener Models

After estimating a Hammerstein-Wiener model on page 12-5 for your system, you can validate whether it reproduces the system behavior within acceptable bounds. It is recommended that you use separate data sets for estimating and validating your model. If the validation indicates low confidence in the estimation, then see "Troubleshooting Model Estimation" on page 17-119 for next steps. For general information about validating models, see "Model Validation".

Compare Simulated Model Output to Measured Output

Plot simulated model output and measured output data for comparison, and compute best fit values. At the command line, use `compare` command. You can also use `sim` to simulate model response. Note that for Hammerstein-Wiener models, the simulated and predicted model output are equivalent because these models have a trivial noise-component, that is the additive disturbance in these models is white noise. For information about plotting simulated output in the app, see "Simulation and Prediction in the App" on page 17-12.

Check Iterative Search Termination Conditions

The estimation report that is generated after model estimation lists the reason the software terminated the estimation. For example, suppose that the report indicates that the estimation reached the maximum number of iterations. You can try repeating the estimation by specifying a larger value for the maximum number of iterations. For information about how to configure the maximum number of iterations and other estimation options, see "Specify Estimation Algorithm" on page 12-8.

To view the estimation report in the app, after model estimation is complete, view the **Estimation Report** area of the **Estimate** tab. At the command line, use `M.Report.Termination` to display the estimation termination conditions, where `M` is the estimated Hammerstein-Wiener model. For example, check the `M.Report.Termination.WhyStop` field that describes why the estimation was stopped.

For more information about the estimation report, see "Estimation Report" on page 1-29.

Check the Final Prediction Error and Loss Function Values

You can compare the performance of several estimated models by comparing the final prediction error and loss function values that are shown in the estimation report.

To view these values for an estimated model `M` at the command line, use the `M.Report.Fit.FPE` (final prediction error) and `M.Report.Fit.LossFcn` (value of loss function at estimation termination) properties. Smaller values typically indicate better performance. However, `M.Report.Fit.FPE` values can be unreliable when the model contains many parameters relative to the estimation data size. Use these indicators with other validation techniques to draw reliable conclusions.

Perform Residual Analysis

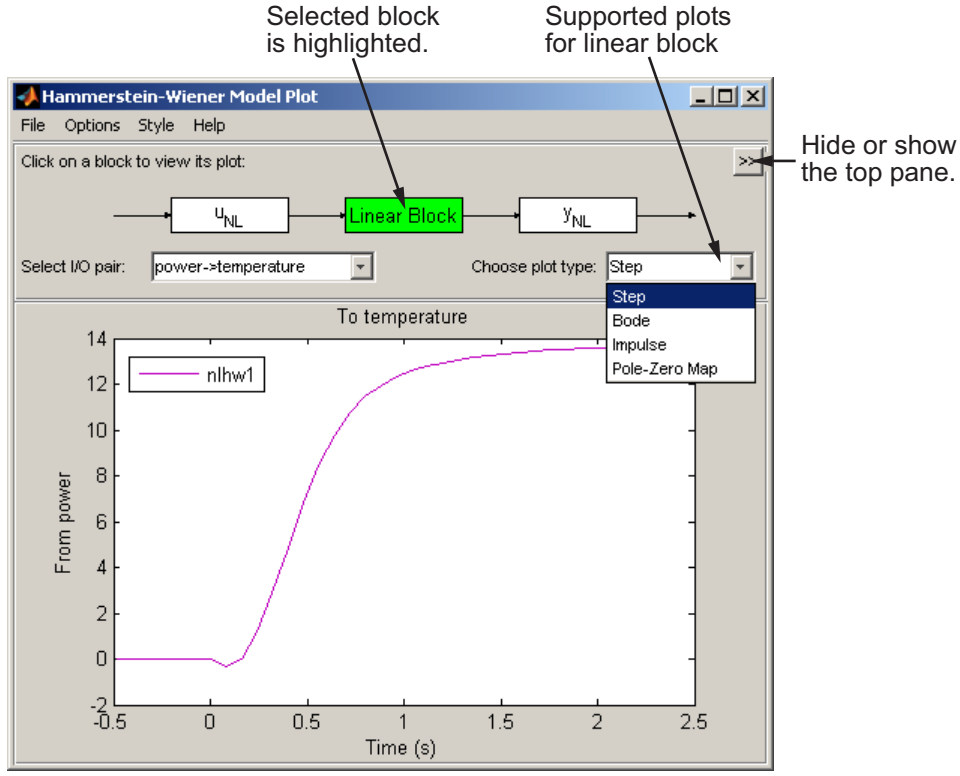
Residuals are differences between the model output and the measured output. Thus, residuals represent the portion of the output not explained by the model. You can analyze the residuals using techniques such as the whiteness test and the independence test. For more information about these tests, see “What Is Residual Analysis?” on page 17-43

At the command line, use `resid` to compute, plot, and analyze the residuals. To plot residuals in the app, see “How to Plot Residuals in the App” on page 17-47.

Examine Hammerstein-Wiener Plots

A Hammerstein-Wiener plot displays the static input and output nonlinearities and linear responses of a Hammerstein-Wiener model.

Examining a Hammerstein-Wiener plot can help you determine whether you have selected a complicated nonlinearity for modeling your system. For example, suppose you use a piecewise-linear input nonlinearity to estimate your model, but the plot indicates saturation behavior. You can estimate a new model using the simpler saturation nonlinearity instead. For multivariable systems, you can use the Hammerstein-Wiener plot to determine whether to exclude nonlinearities for specific channels. If the nonlinearity for a specific input or output channel does not exhibit strong nonlinear behavior, you can estimate a new model after setting the nonlinearity at that channel to unit gain.



You can generate these plots in the **System Identification** app and at the command line. In the plot window, you can view the nonlinearites and linear responses by clicking one of the three blocks that represent the model:

- u_{NL} (*input nonlinearity*)— Click this block to view the static nonlinearity at the input to the `Linear Block`. The plot displays `evaluate(M.InputNonlinearity,u)` where `M` is the Hammerstein-Wiener model, and `u` is the input to the input nonlinearity block. For information about the blocks, see "Structure of Hammerstein-Wiener Models" on page 12-2.
- `Linear Block` — Click this block to view the Step, impulse, Bode, and pole-zero response plots of the embedded linear model (`M.LinearModel`). By default, a step plot of the linear model is displayed.

- y_{NL} (*output nonlinearity*) — Click this block to view the static nonlinearity at the output of the `Linear Block`. The plot displays `evaluate(M.OutputNonlinearity,x)`, where `x` is the output of the linear block.

Creating a Hammerstein-Wiener Plot

To create a Hammerstein-Wiener plot in the System Identification app, after you have estimated the model, select the **Hamm-Wiener** check box in the **Model Views** area. For general information about creating and working with plots in the app, see "Working with Plots" on page 21-11.

At the command line, after you have estimated a Hammerstein-Wiener model `M`, you can access the objects representing the input and output nonlinearity estimators using `M.InputNonlinearity` and `M.OutputNonlinearity`.

Use `plot` to view the shape of the nonlinearities and the properties of the linear block.

```
plot(M)
```

You can use additional `plot` arguments to specify the following information:

- Include several Hammerstein-Wiener models on the plot.
- Configure how to evaluate the nonlinearity at each input and output channel.
- Specify the time or frequency values for computing transient and frequency response plots of the linear block.

Configuring a Hammerstein-Wiener Plot

To configure the plots of the nonlinear blocks:

1 In the Hammerstein-Wiener Model Plot window, select the nonlinear block you want to plot.

   - To plot the response of the input nonlinearity function, click the u_{NL} block.
   - To plot the response of the output nonlinearity function, click the y_{NL} block.

   The selected block is highlighted green.

   Note The input to the output nonlinearity block y_{NL} is the output from the Linear Block and not the measured input data.

2. If your model contains multiple inputs or outputs, select the channel in the **Select nonlinearity at channel** list. Selecting the channel updates the plot and displays the nonlinearity values versus the corresponding input to this nonlinear block.
3. Change the range of the horizontal axis of the plot. This feature is available only for plots generated in the **System Identification** app.

   In the plot window, select **Options > Set input range** to open the Range for Input to Nonlinearity dialog box. This feature is only available in the **System Identification** app.

   Enter the range using the format `[MinValue MaxValue]`. Click **Apply** and then **Close** to update the plot.

To configure the linear block response plot:

1. In the Hammerstein-Wiener Model Plot window, click the `Linear Block`.
2. Select the input-output data pair for which you want to view the response in the **Select I/O pair** list.
3. Select the type of linear response plot. In the **Choose plot type** list, choose from the following options:

   - `Step`
   - `Impulse`
   - `Bode`
   - `Pole-Zero Map`
4. Set the time span for a step or impulse response plot. This feature is available only for plots generated in the **System Identification** app.

   In the plot window, select **Options > Time span**. In the Time Range dialog box, specify the time span in the units of time you specified for the model. For a time span T, the resulting response is plotted from $-T/4$ to T. Click **Apply** and then **Close**.
5. Set the frequency range for a Bode plot. This feature is available only for plots generated in the app.

   The default frequency vector is 128 linearly distributed values, greater than zero and less than or equal to the Nyquist frequency. To change the range, select **Options > Frequency range**. In the Frequency Range dialog box, specify a new frequency vector in units of rad per model time units using one of following methods:

- MATLAB expression, such as `(1:100)*pi/100` or `logspace(-3,-1,200)`. The expression cannot contain variables in the MATLAB workspace.
- Row vector of values, such as `(1:0.1:100)`.

Click **Apply** and then **Close**.

See Also

More About

- "Identifying Hammerstein-Wiener Models" on page 12-5
- "Estimate Hammerstein-Wiener Models in the App" on page 12-15
- "Estimate Hammerstein-Wiener Models at the Command Line" on page 12-18
- "Using Hammerstein-Wiener Models" on page 12-11

How the Software Computes Hammerstein-Wiener Model Output

This topic describes how the software evaluates the output of nonlinearity estimators and uses this output to compute the response of a Hammerstein-Wiener model.

Evaluating Nonlinearities (SISO)

Evaluating the output of a nonlinearity for a input u requires that you first extract the input or output nonlinearity from the model:

```
F = M.InputNonlinearity;
H = M.OutputNonlinearity;
```

Evaluate $F(u)$:

```
w = evaluate(F,u)
```

where `u` is a scalar representing the value of the input signal at a given time.

You can evaluate output at multiple time instants by evaluating F for several time values simultaneously using a column vector of input values:

```
w = evaluate(F,[u1;u2;u3])
```

Similarly, you can evaluate the value of the nonlinearity H using the output of the linear block $x(t)$ as its input:

```
y = evaluate(H,x)
```

Evaluating Nonlinearities (MIMO)

For MIMO models, F and H are vectors of length nu and ny, respectively. nu is the number of inputs and ny is the number of outputs. In this case, you must evaluate the predicted output of each nonlinearity separately.

For example, suppose that you estimate a two-input model:

```
M = nlhw(data,[nb nf nk],[wavenet;poly1d],'saturation')
```

In the input nonlinearity:

```
F = M.InputNonlinearity
F1 = F(1);
F2 = F(2);
```

`F` is a vector function containing two elements: `F=[F1(u1_value);F2(u2_value)]`, where `F1` is a `wavenet` object and `F2` is a `poly1d` object. `u1_value` is the first input signal and `u2_value` is the second input signal.

Evaluate `F` by evaluating `F1` and `F2` separately:

```
w1 = evaluate(F(1),u1_value);
w2 = evaluate(F(2),u2_value);
```

The total input to the linear block, `w`, is a vector of `w1` and `w2` (`w = [w1 w2]`).

Similarly, you can evaluate the value of the nonlinearity *H*:

```
H = M.OutputNonlinearity;
```

Simulation of Hammerstein-Wiener Model

This example shows how the software evaluates the simulated output by first computing the output of the input and output nonlinearity estimators.

Estimate a Hammerstein-Wiener model.

```
load twotankdata
estData = iddata(y,u,0.2);
M = nlhw(estData,[1 5 3],'pwlinear','poly1d');
```

Extract the input nonlinearity, linear model, and output nonlinearity as separate variables.

```
uNL = M.InputNonlinearity;
linModel = M.LinearModel;
yNL = M.OutputNonlinearity;
```

Simulate the output of the input nonlinearity estimator.

Input data for simulation

```
u = estData.u;
```

Compute output of input nonlinearity

```
w = evaluate(uNL,u);
```

Compute response of linear model to input `w` and zero initial conditions.

```
x = sim(linModel,w);
```

Compute the output of the Hammerstein-Wiener model `M` as the output of the output nonlinearity estimator to input `x`.

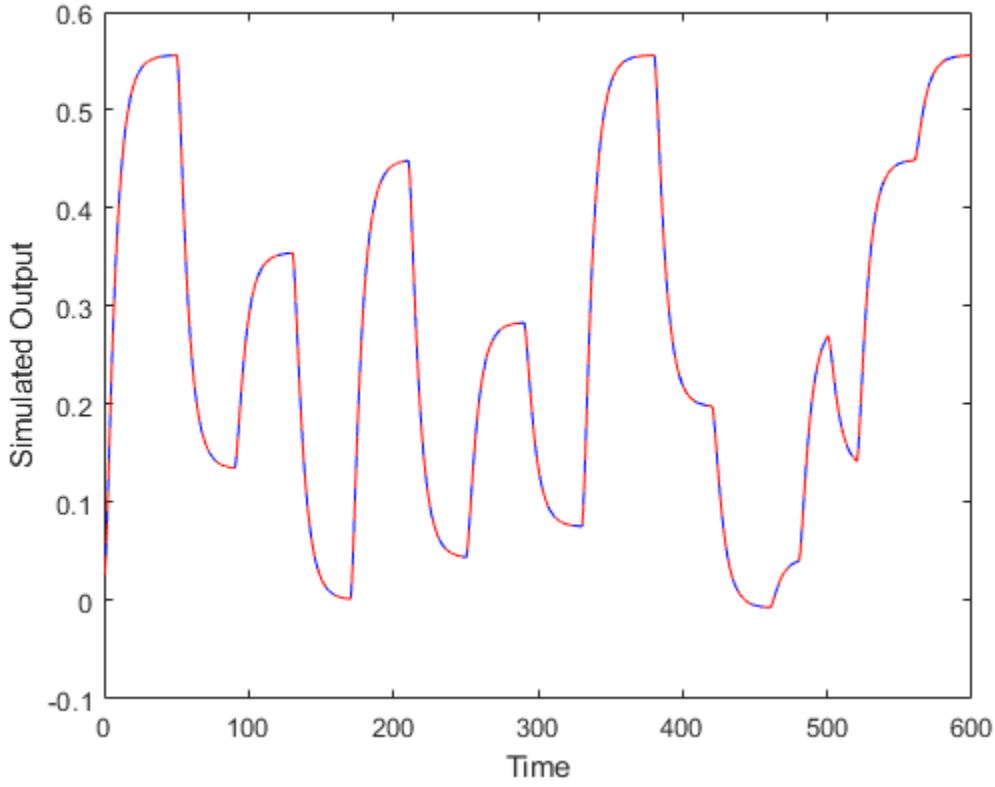
```
y = evaluate(yNL,x);
```

The previous set of commands are equivalent to directly simulating the output of `M` using the `sim` command.

```
ysim = sim(M,u);
```

Plot `y` and `ysim`, the manual and direct simulation results, respectively.

```
time = estData.SamplingInstants;
plot(time,y,'b',time,ysim,'--r');
xlabel('Time');
ylabel('Simulated Output')
```



The plot indicates that `y` and `ysim` are the same.

See Also

`evaluate`

More About

- "What are Hammerstein-Wiener Models?" on page 12-2
- "Identifying Hammerstein-Wiener Models" on page 12-5

Estimate Hammerstein-Wiener Models Initialized Using Linear OE Models

This example shows how to estimate Hammerstein-Wiener models using linear OE models.

Load the estimation data.

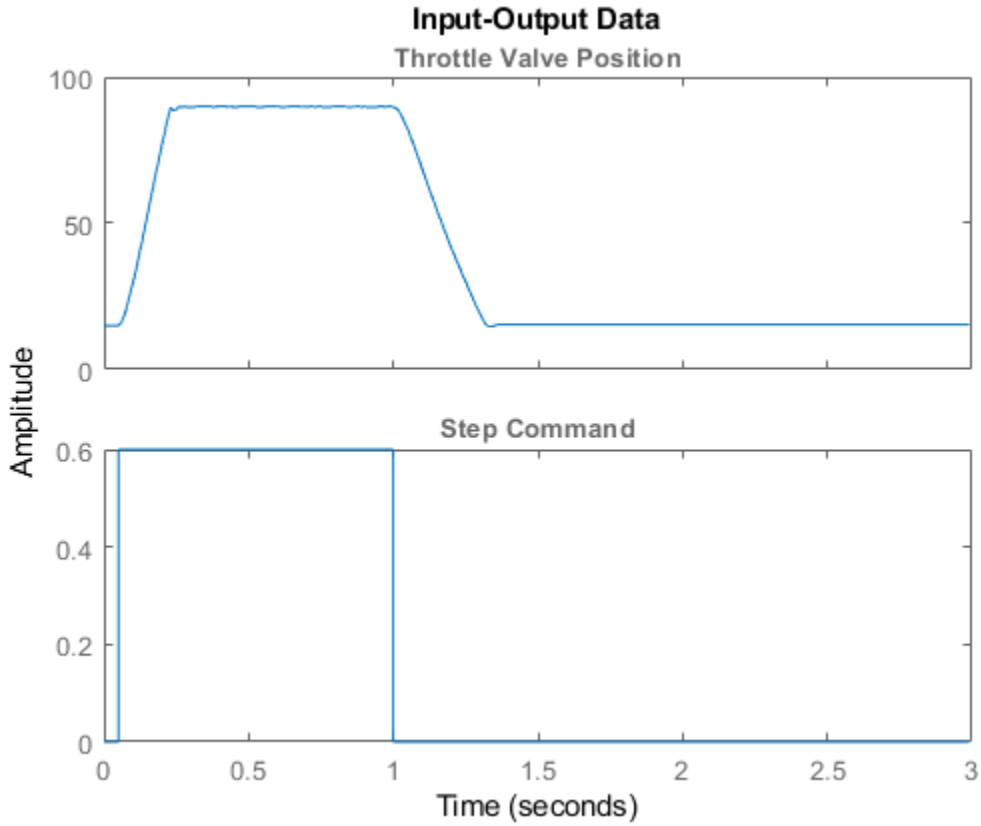
```
load throttledata.mat
```

This command loads the data object `ThrottleData` into the workspace. The object contains input and output samples collected from an engine throttle system, sampled at a rate of 100Hz.

A DC motor controls the opening angle of the butterfly valve in the throttle system. A step signal (in volts) drives the DC motor. The output is the angular position (in degrees) of the valve.

Plot the data to view and analyze the data characteristics.

```
plot(ThrottleData)
```



In the normal operating range of 15-90 degrees, the input and output variables have a linear relationship. You use a linear model of low order to model this relationship.

In the throttle system, a hard stop limits the valve position to `90` degrees, and a spring brings the valve to `15` degrees when the DC motor is turned off. These physical components introduce nonlinearities that a linear model cannot capture.

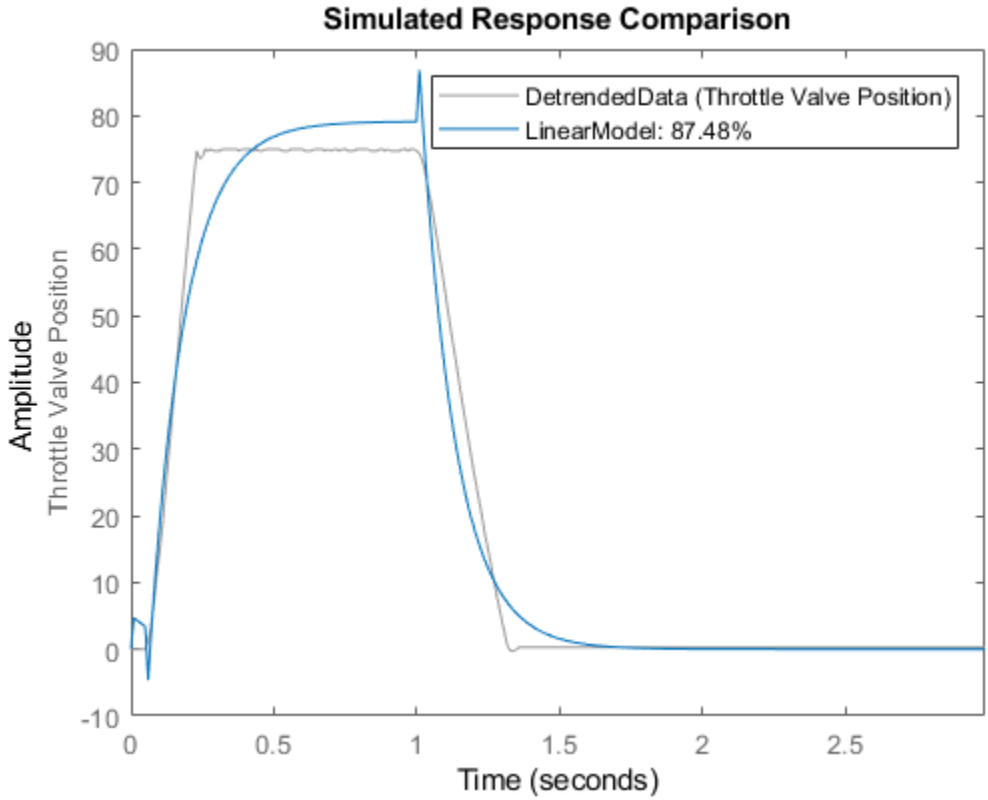
Estimate a Hammerstein-Wiener model to model the linear behavior of this single-input single-output system in the normal operating range.

```
% Detrend the data because linear models cannot capture offsets.
Tr = getTrend(ThrottleData);
Tr.OutputOffset = 15;
DetrendedData = detrend(ThrottleData,Tr);
```

```
% Estimate a linear OE model with na=2, nb=1, nk=1.
opt = oeOptions('Focus','simulation');
LinearModel = oe(DetrendedData,[2 1 1],opt);
```

Compare the simulated model response with estimation data.

```
compare(DetrendedData, LinearModel)
```



The linear model captures the rising and settling behavior in the linear operating range but does not account for output saturation at 90 degrees.

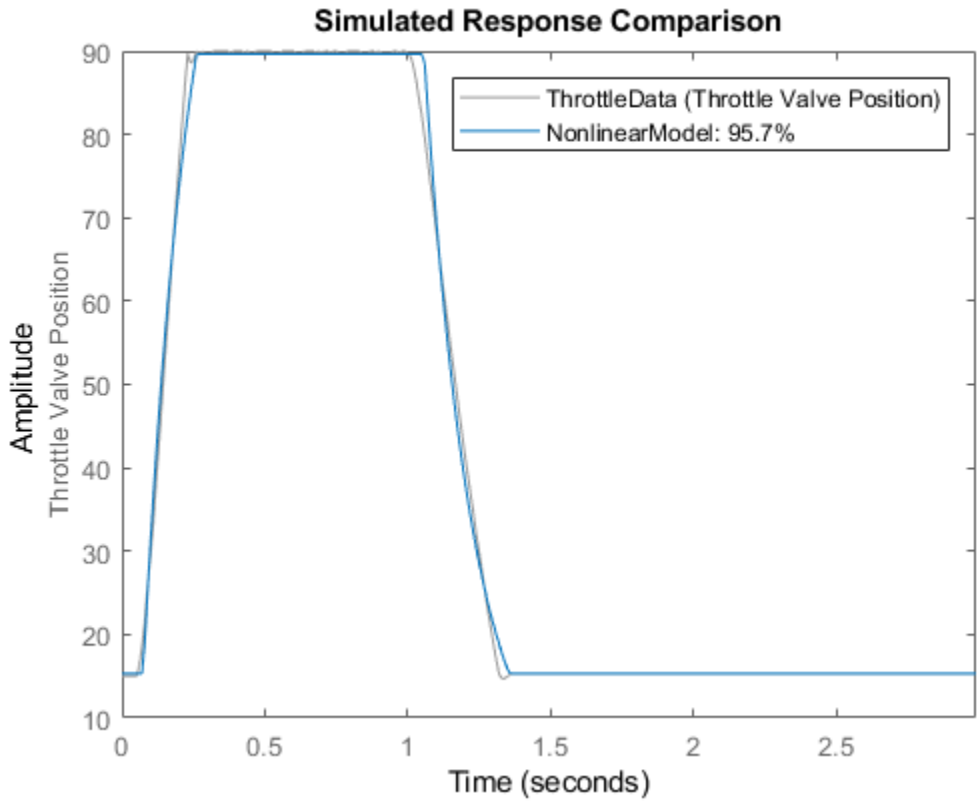
Estimate a Hammerstein-Wiener model to model the output saturation.

```
NonlinearModel = nlhw(ThrottleData, LinearModel, [], 'saturation');
```

The software uses the orders and delay of the linear model for the orders of the nonlinear model. In addition, the software uses the B and F polynomials of the linear transfer function.

Compare the nonlinear model with data.

```
compare(ThrottleData, NonlinearModel)
```



See Also

More About

- “What are Hammerstein-Wiener Models?” on page 12-2
- “Identifying Hammerstein-Wiener Models” on page 12-5
- “Estimate Hammerstein-Wiener Models at the Command Line” on page 12-18

ODE Parameter Estimation (Grey-Box Modeling)

- "Supported Grey-Box Models" on page 13-2
- "Data Supported by Grey-Box Models" on page 13-4
- "Choosing idgrey or idnlgrey Model Object" on page 13-5
- "Estimate Linear Grey-Box Models" on page 13-8
- "Estimate Continuous-Time Grey-Box Model for Heat Diffusion" on page 13-13
- "Estimate Discrete-Time Grey-Box Model with Parameterized Disturbance" on page 13-17
- "Estimate Coefficients of ODEs to Fit Given Solution" on page 13-20
- "Estimate Model Using Zero/Pole/Gain Parameters" on page 13-28
- "Estimate Nonlinear Grey-Box Models" on page 13-35
- "Creating IDNLGREY Model Files" on page 13-59
- "Identifying State-Space Models with Separate Process and Measurement Noise Descriptions" on page 13-72
- "After Estimating Grey-Box Models" on page 13-77

Supported Grey-Box Models

If you understand the physics of your system and can represent the system using ordinary differential or difference equations (ODEs) with unknown parameters, then you can use System Identification Toolbox commands to perform linear or nonlinear grey-box modeling. *Grey-box model* ODEs specify the mathematical structure of the model explicitly, including couplings between parameters. Grey-box modeling is useful when you know the relationships between variables, constraints on model behavior, or explicit equations representing system dynamics.

The toolbox supports both continuous-time and discrete-time linear and nonlinear models. However, because most laws of physics are expressed in continuous time, it is easier to construct models with physical insight in continuous time, rather than in discrete time.

In addition to dynamic input-output models, you can also create time-series models that have no inputs and static models that have no states.

If it is too difficult to describe your system using known physical laws, you can use the black-box modeling approach. For more information, see "Linear Model Identification" and "Nonlinear Model Identification".

You can also use the `idss` model object to perform structured model estimation by using its `Structure` property to fix or free specific parameters. However, you cannot use this approach to estimate arbitrary structures (arbitrary parameterization). For more information about structure matrices, see "Estimate State-Space Models with Structured Parameterization" on page 7-32.

See Also

`idgrey` | `idnlgrey` | `idss`

Related Examples

- "Estimate Linear Grey-Box Models" on page 13-8
- "Estimate Nonlinear Grey-Box Models" on page 13-35

More About

- "Data Supported by Grey-Box Models" on page 13-4
- "Choosing idgrey or idnlgrey Model Object" on page 13-5

Data Supported by Grey-Box Models

You can estimate both continuous-time or discrete-time grey-box models for data with the following characteristics:

- Time-domain or frequency-domain data, including time-series data with no inputs.

  Note Nonlinear grey-box models support only time-domain data.

- Single-output or multiple-output data

You must first import your data into the MATLAB workspace. You must represent your data as an `iddata` or `idfrd` object. For more information about preparing data for identification, see “Data Preparation”.

See Also

`idgrey` | `idnlgrey` | `idss`

Related Examples

- “Estimate Linear Grey-Box Models” on page 13-8
- “Estimate Nonlinear Grey-Box Models” on page 13-35

More About

- “Supported Grey-Box Models” on page 13-2
- “Choosing idgrey or idnlgrey Model Object” on page 13-5

Choosing idgrey or idnlgrey Model Object

Grey-box models require that you specify the structure of the ODE model in a file. You use this file to create the `idgrey` or `idnlgrey` model object. You can use both the `idgrey` and the `idnlgrey` objects to model linear systems. However, you can only represent nonlinear dynamics using the `idnlgrey` model object.

The `idgrey` object requires that you write a function to describe the linear dynamics in the state-space form, such that this file returns the state-space matrices as a function of your parameters. For more information, see “Specifying the Linear Grey-Box Model Structure” on page 13-8.

The `idnlgrey` object requires that you write a function or MEX-file to describe the dynamics as a set of first-order differential equations, such that this file returns the output and state derivatives as a function of time, input, state, and parameter values. For more information, see “Specifying the Nonlinear Grey-Box Model Structure” on page 13-35.

The following table compares `idgrey` and `idnlgrey` model objects.

Comparison of idgrey and idnlgrey Objects

Settings and Operations	Supported by idgrey?	Supported by idnlgrey?
Set bounds on parameter values.	Yes	Yes
Handle initial states individually.	Yes	Yes
Perform linear analysis.	Yes For example, use the `bode` command.	No
Honor stability constraints.	Yes Specify constraints using the `Advanced.StabilityThreshold` estimation option. For more information, see `greyestOptions`.	No **Note** You can use parameter bounds to ensure stability of an `idnlgrey` model, if these bounds are known.
Estimate a disturbance model.	Yes The disturbance model is represented by `K` in state-space equations.	No
Optimize estimation results for simulation or prediction.	Yes Set the `Focus` estimation option to `'simulation'` or `'prediction'`. For more information, see `greyestOptions`.	No Because `idnlgrey` models are Output-Error models, there is no difference between simulation and prediction results.

See Also

`idgrey` | `idnlgrey` | `idss`

Related Examples

- “Estimate Linear Grey-Box Models” on page 13-8
- “Estimate Nonlinear Grey-Box Models” on page 13-35

More About

- “Supported Grey-Box Models” on page 13-2
- “Data Supported by Grey-Box Models” on page 13-4

Estimate Linear Grey-Box Models

In this section...
“Specifying the Linear Grey-Box Model Structure” on page 13-8 “Create Function to Represent a Grey-Box Model” on page 13-9

Specifying the Linear Grey-Box Model Structure

You can estimate linear discrete-time and continuous-time grey-box models for arbitrary ordinary differential or difference equations using single-output and multiple-output time-domain data, or time-series data (output-only).

You must represent your system equations in state-space form. *State-space models* use state variables *x(t)* to describe a system as a set of first-order differential equations, rather than by one or more *n*th-order differential equations.

The first step in grey-box modeling is to write a function that returns state-space matrices as a function of user-defined parameters and information about the model.

Use the following format to implement the linear grey-box model in the file:

```
[A,B,C,D] = myfunc(par1,par2,...,parN,Ts,aux1,aux2,...)
```

where the output arguments are the state-space matrices and `myfunc` is the name of the file. `par1,par2,...,parN` are the *N* parameters of the model. Each entry may be a scalar, vector or matrix.`Ts` is the sample time. `aux1,aux2,...` are the optional input arguments that `myfunc` uses to compute the state-space matrices in addition to the parameters and sample time. `aux` contains auxiliary variables in your system. You use auxiliary variables to vary system parameters at the input to the function, and avoid editing the file.

You can write the contents of `myfunc` to parameterize either a continuous-time, or a discrete-time state-space model, or both. When you create the linear grey-box model using `myfunc`, you can specify the nature of the output arguments of `myfunc`. The continuous-time state-space model has the form:

In continuous-time, the state-space description has the following form:

$$\dot{x}(t) = Ax(t) + Bu(t) + Ke(t)$$
$$y(t) = Cx(t) + Du(t) + e(t)$$
$$x(0) = x0$$

where, A,B,C and D are matrices that are parameterized by the parameters `par1,par2,...,parN`. The noise matrix K and initial state vector, $x0$, are not parameterized by `myfunc`. In some applications, you may want to express K and $x0$ as quantities that are parameterized by chosen parameters, just as the A, B, C and D matrices. To handle such cases, you can write the ODE file, `myfunc`, to return K and $x0$ as additional output arguments:

```
[A,B,C,D,K,x0] = myfunc(par1,par2,...,parN,Ts,aux1,aux2,...)
```

K and $x0$ are thus treated in the same way as the A, B, C and D matrices. They are all functions of the parameters `par1,par2,...,parN`. To configure the handling of initial states, $x0$, and the disturbance component, K, during estimation, use the `greyestOptions` option set.

In discrete-time, the state-space description has a similar form:

$$x(k+1) = Ax(k) + Bu(k) + Ke(t)$$
$$y(k) = Cx(k) + Du(k) + e(t)$$
$$x(0) = x0$$

where, A, B, C and D are now the discrete-time matrices that are parameterized by the parameters `par1,par2,...,parN`. K and $x0$ are not directly parameterized, but can be estimated if required by configuring the corresponding estimation options.

After creating the function or MEX-file with your model structure, you must define an `idgrey` model object.

Create Function to Represent a Grey-Box Model

This example shows how to represent the structure of the following continuous-time model:

$$\dot{x}(t) = \begin{bmatrix} 0 & 1 \\ 0 & \theta_1 \end{bmatrix} x(t) + \begin{bmatrix} 0 \\ \theta_2 \end{bmatrix} u(t)$$
$$y(t) = \begin{bmatrix} 1 & 0 \\ 0 & 1 \end{bmatrix} x(t) + e(t)$$
$$x(0) = \begin{bmatrix} \theta_3 \\ 0 \end{bmatrix}$$

This equation represents an electrical motor, where $y_1(t) = x_1(t)$ is the angular position of the motor shaft, and $y_2(t) = x_2(t)$ is the angular velocity. The parameter $-\theta_1$ is the inverse time constant of the motor, and $-\theta_2/\theta_1$ is the static gain from the input to the angular velocity.

The motor is at rest at `t` = 0, but its angular position θ_3 is unknown. Suppose that the approximate nominal values of the unknown parameters are $\theta_1 = -1$, $\theta_2 = 0.25$ and $\theta_3 = 0$. For more information about this example, see the section on state-space models in *System Identification: Theory for the User* , Second Edition, by Lennart Ljung, Prentice Hall PTR, 1999.

The continuous-time state-space model structure is defined by the following equation:
$\dot{x}(t) = Fx(t) + Gu(t) + \tilde{K}w(t)$
$y(t) = Hx(t) + Du(t) + w(t)$
$x(0) = x0$

If you want to estimate the same model using a structured state-space representation, see “Estimate Structured Continuous-Time State-Space Models” on page 7-37.

To prepare this model for estimation:

- Create the following file to represent the model structure in this example:

```
function [A,B,C,D,K,x0] = myfunc(par,T)
A = [0 1; 0 par(1)];
B = [0;par(2)];
C = eye(2);
D = zeros(2,1);
K = zeros(2,2);
x0 = [par(3);0];
```

Save the file such that it is in the MATLAB® search path.

- Use the following syntax to define an `idgrey` model object based on the `myfunc` file:

```
par = [-1; 0.25; 0];
aux = {};
T = 0;
m = idgrey('myfunc',par,'c',aux,T);
```

where `par` represents a vector of all the user-defined parameters and contains their nominal (initial) values. In this example, all the scalar-valued parameters are grouped in the `par` vector. The scalar-valued parameters could also have been treated as independent input arguments to the ODE function `myfunc`. `'c'` specifies that the underlying parameterization is in continuous time. `aux` represents optional arguments. As `myfunc` does not have any optional arguments, use `aux = {}`. `T` specifies the sample time; `T = 0` indicates a continuous-time model.

Load the estimation data.

```
load(fullfile(matlabroot,'toolbox','ident','iddemos','data','dcmotordata'));
data = iddata(y,u,0.1);
```

Use `greyest` to estimate the grey-box parameter values:

```
m_est = greyest(data,m);
```

where `data` is the estimation data and `m` is an estimation initialization `idgrey` model. `m_est` is the estimated `idgrey` model.

See Also

`greyest` | `idgrey`

Related Examples

- “Estimate Continuous-Time Grey-Box Model for Heat Diffusion” on page 13-13
- “Estimate Discrete-Time Grey-Box Model with Parameterized Disturbance” on page 13-17
- “Estimate Coefficients of ODEs to Fit Given Solution” on page 13-20
- “Estimate Model Using Zero/Pole/Gain Parameters” on page 13-28

- “Estimate Nonlinear Grey-Box Models” on page 13-35

Estimate Continuous-Time Grey-Box Model for Heat Diffusion

This example shows how to estimate the heat conductivity and the heat-transfer coefficient of a continuous-time grey-box model for a heated-rod system.

This system consists of a well-insulated metal rod of length L and a heat-diffusion coefficient κ. The input to the system is the heating power *u(t)* and the measured output *y(t)* is the temperature at the other end.

Under ideal conditions, this system is described by the heat-diffusion equation—which is a partial differential equation in space and time.

$$\frac{\partial x(t,\xi)}{\partial t} = \kappa \frac{\partial^2 x(t,\xi)}{\partial \xi^2}$$

To get a continuous-time state-space model, you can represent the second-derivative using the following difference approximation:

$$\frac{\partial^2 x(t,\xi)}{\partial \xi^2} = \frac{x(t,\xi+\Delta L) - 2x(t,\xi) + x(t,\xi-\Delta L)}{(\Delta L)^2}$$

where $\xi = k \cdot \Delta L$

This transformation produces a state-space model of order $n = \frac{L}{\Delta L}$, where the state variables $x(t, k \cdot \Delta L)$ are lumped representations for $x(t,\xi)$ for the following range of values:

$k \cdot \Delta L \le \xi < (k+1)\Delta L$

The dimension of x depends on the spatial grid size ΔL in the approximation.

The heat-diffusion equation is mapped to the following continuous-time state-space model structure to identify the state-space matrices:

$$\dot{x}(t) = Fx(t) + Gu(t) + \tilde{K}w(t)$$
$$y(t) = Hx(t) + Du(t) + w(t)$$
$$x(0) = x0$$

The state-space matrices are parameterized by the heat diffusion coefficient κ and the heat transfer coefficient at the far end of the rod h_{tf}. The expressions also depend upon

the grid size, *Ngrid*, and the length of the rod L. The initial conditions x0 are a function of the initial room temperature, treated as a known quantity in this example.

1 Create a MATLAB file.

The following code describes the state-space equation for this model. The parameters are κ and h_{tf} while the auxiliary variables are *Ngrid*, *L* and initial room temperature `temp`. The grid size is supplied as an auxiliary variable so that the ODE function can be easily adapted for various grid sizes.

```
function [A,B,C,D,K,x0] = heatd(kappa,htf,T,Ngrid,L,temp)
% ODE file parameterizing the heat diffusion model

% kappa (first parameter) - heat diffusion coefficient
% htf (second parameter) - heat transfer coefficient
%                          at the far end of rod

% Auxiliary variables for computing state-space matrices:
% Ngrid: Number of points in the space-discretization
% L: Length of the rod
% temp: Initial room temperature (uniform)

% Compute space interval
deltaL = L/Ngrid;

% A matrix
A = zeros(Ngrid,Ngrid);
for kk = 2:Ngrid-1
  A(kk,kk-1) = 1;
  A(kk,kk) = -2;
  A(kk,kk+1) = 1;
end

% Boundary condition on insulated end
A(1,1) = -1; A(1,2) = 1;
A(Ngrid,Ngrid-1) = 1;
A(Ngrid,Ngrid) = -1;
A = A*kappa/deltaL/deltaL;

% B matrix
B = zeros(Ngrid,1);
B(Ngrid,1) = htf/deltaL;

% C matrix
```

If the outputs were coupled (`na` was not a diagonal matrix), the situation will be more complex and simply adding an integrator to the second noise channel will not work.

Spectrum Estimation Using Complex Data - Marple's Test Case

This example shows how to perform spectral estimation on time series data. We use Marple's test case (The complex data in L. Marple: S.L. Marple, Jr, Digital Spectral Analysis with Applications, Prentice-Hall, Englewood Cliffs, NJ 1987.)

Test Data

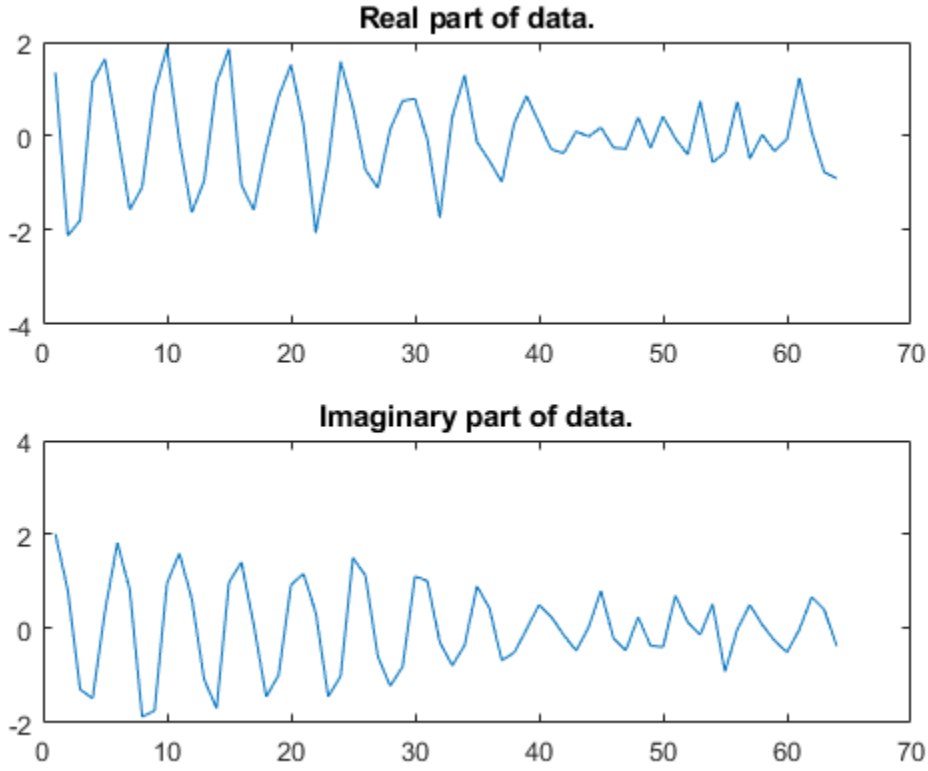
Let us begin by loading the test data:

```
load marple
```

Most of the routines in System Identification Toolbox™ support complex data. For plotting we examine the real and imaginary parts of the data separately, however.

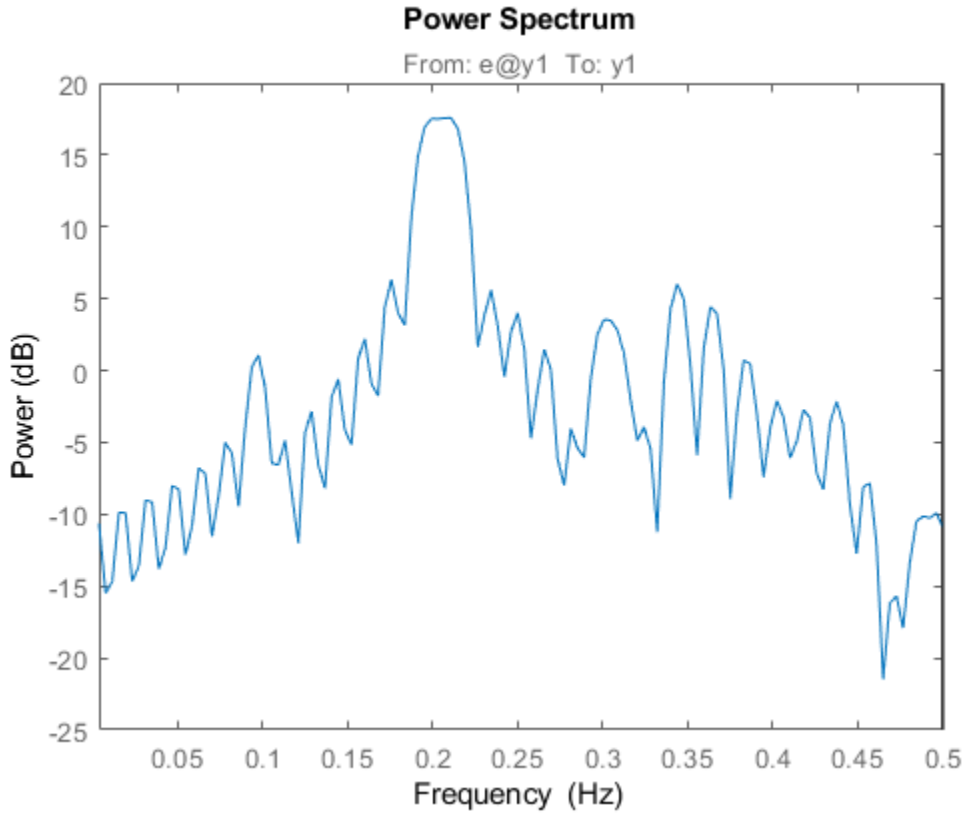
First, take a look at the data:

```
subplot(211),plot(real(marple)),title('Real part of data.')
subplot(212),plot(imag(marple)),title('Imaginary part of data.')
```



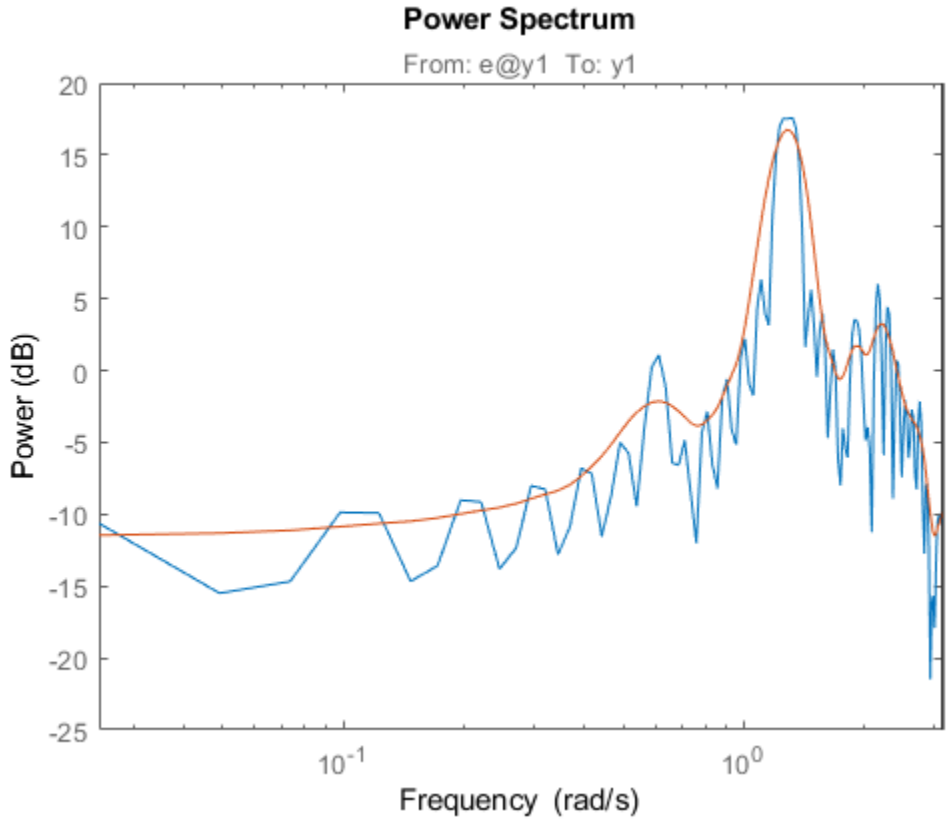
As a preliminary analysis step, let us check the periodogram of the data:

```
per = etfe(marple);
w = per.Frequency;
clf
h = spectrumplot(per,w);
opt = getoptions(h);
opt.FreqScale = 'linear';
opt.FreqUnits = 'Hz';
setoptions(h,opt)
```



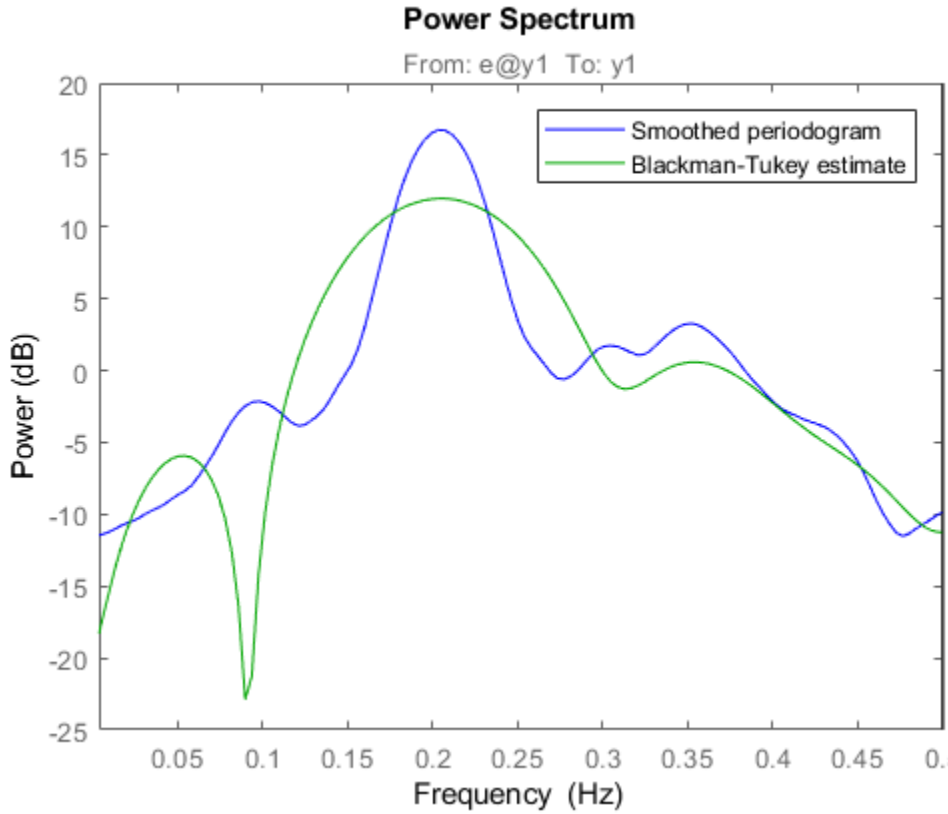
Since the data record is only 64 samples, and the periodogram is computed for 128 frequencies, we clearly see the oscillations from the narrow frequency window. We therefore apply some smoothing to the periodogram (corresponding to a frequency resolution of 1/32 Hz):

```
sp = etfe(marple,32);
spectrumplot(per,sp,w);
```



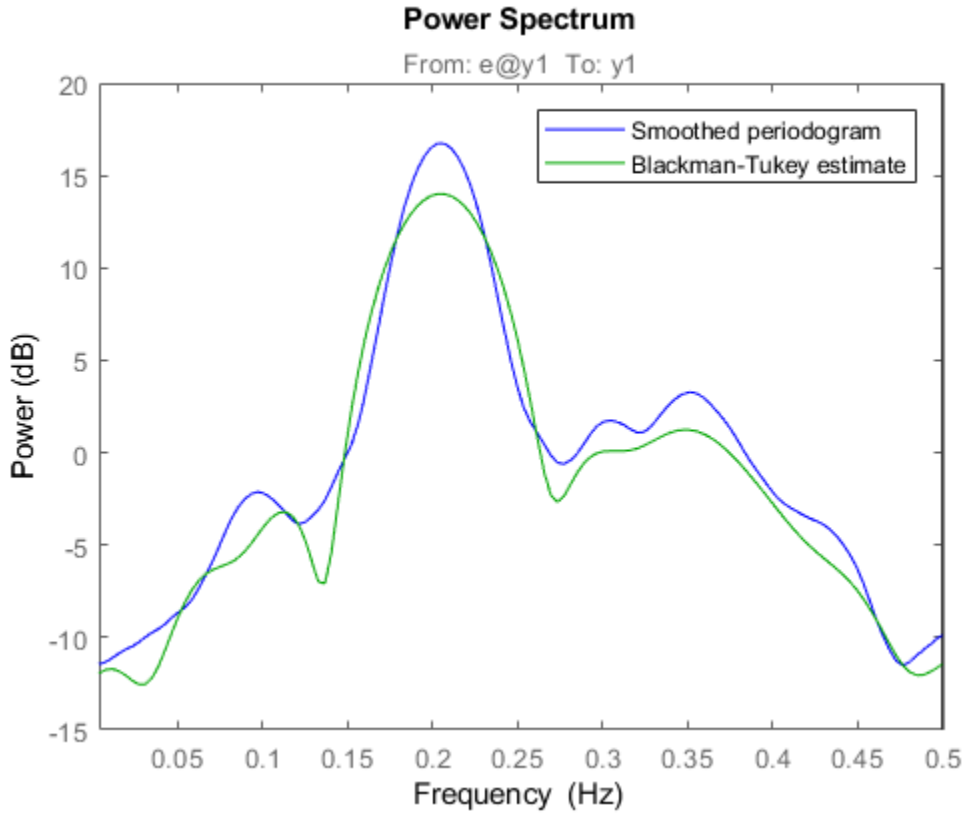
Let us now try the Blackman-Tukey approach to spectrum estimation:

```
ssm = spa(marple); % Function spa performs spectral estimation
spectrumplot(sp,'b',ssm,'g',w,opt);
legend({'Smoothed periodogram','Blackman-Tukey estimate'});
```



The default window length gives a very narrow lag window for this small amount of data. We can choose a larger lag window by:

```
ss20 = spa(marple,20);
spectrumplot(sp,'b',ss20,'g',w,opt);
legend({'Smoothed periodogram','Blackman-Tukey estimate'});
```



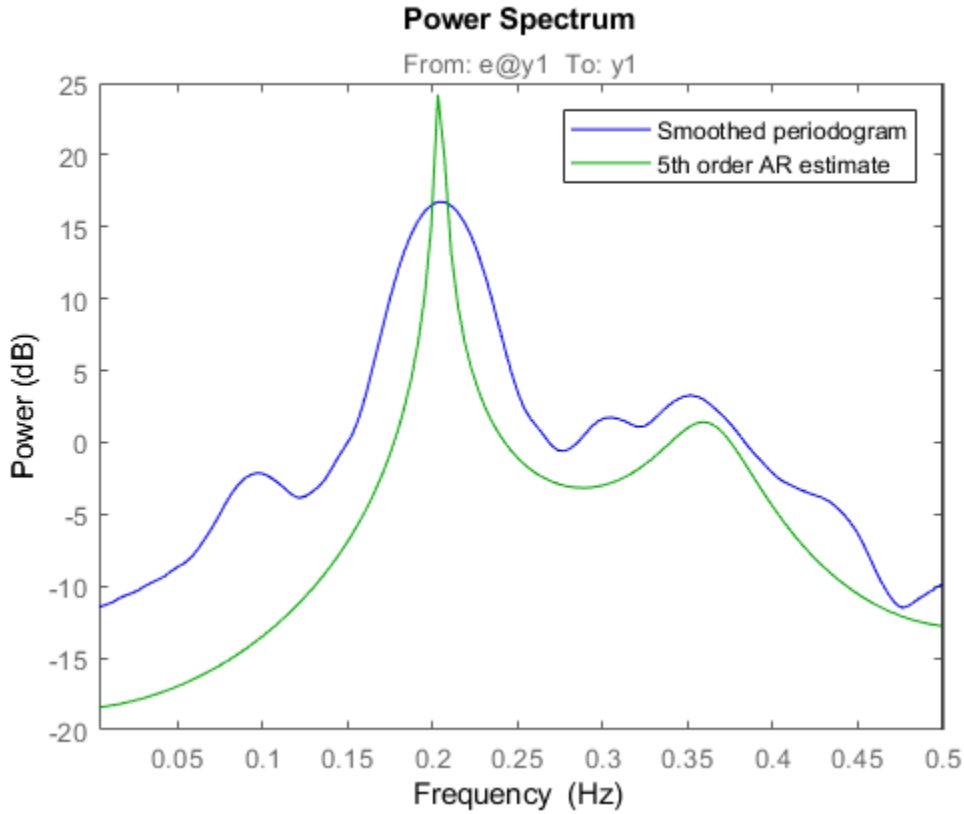
Estimating an Autoregressive (AR) Model

A parametric 5-order AR-model is computed by:

```
t5 = ar(marple,5);
```

Compare with the periodogram estimate:

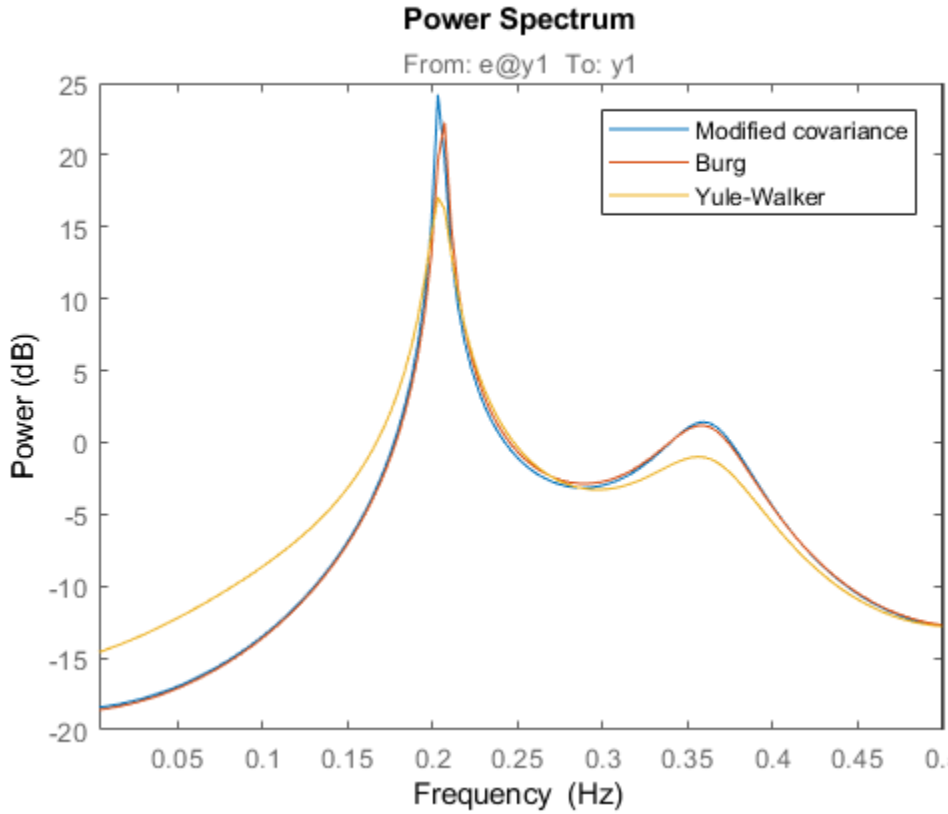
```
spectrumplot(sp,'b',t5,'g',w,opt);
legend({'Smoothed periodogram','5th order AR estimate'});
```



The AR-command in fact covers 20 different methods for spectrum estimation. The above one was what is known as 'the modified covariance estimate' in Marple's book.

Some other well known ones are obtained with:

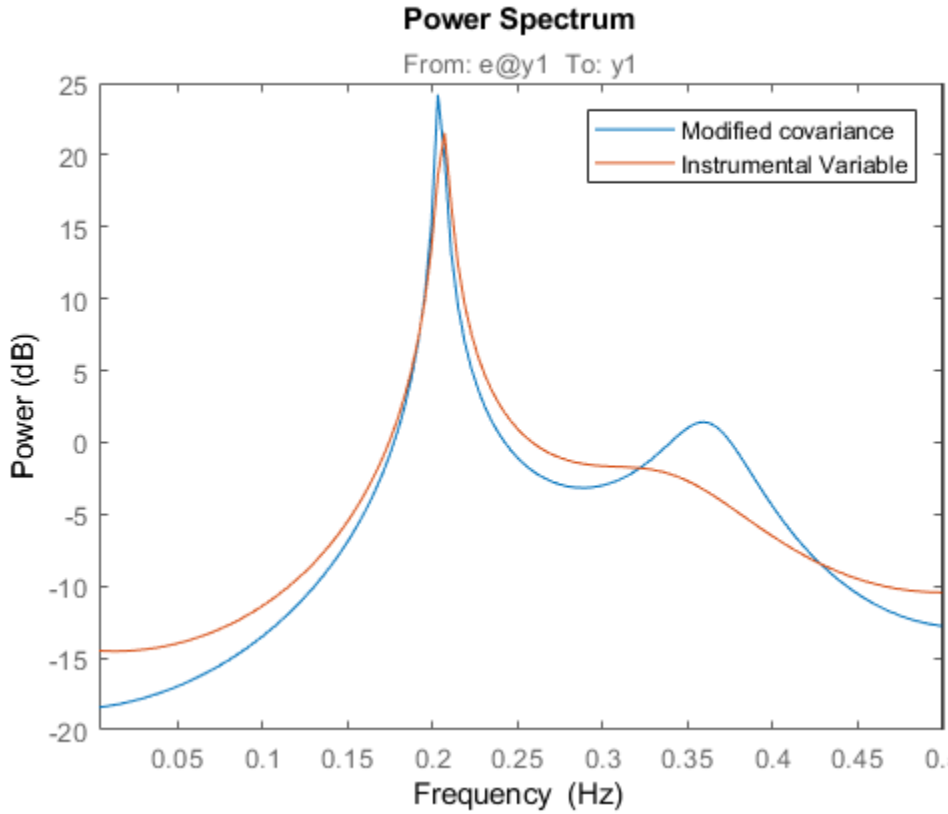
```
tb5 = ar(marple,5,'burg');       % Burg's method
ty5 = ar(marple,5,'yw');         % The Yule-Walker method
spectrumplot(t5,tb5,ty5,w,opt);
legend({'Modified covariance','Burg','Yule-Walker'})
```



Estimating AR Model using Instrumental Variable Approach

AR-modeling can also be done using the Instrumental Variable approach. For this, we use the function `ivar`:

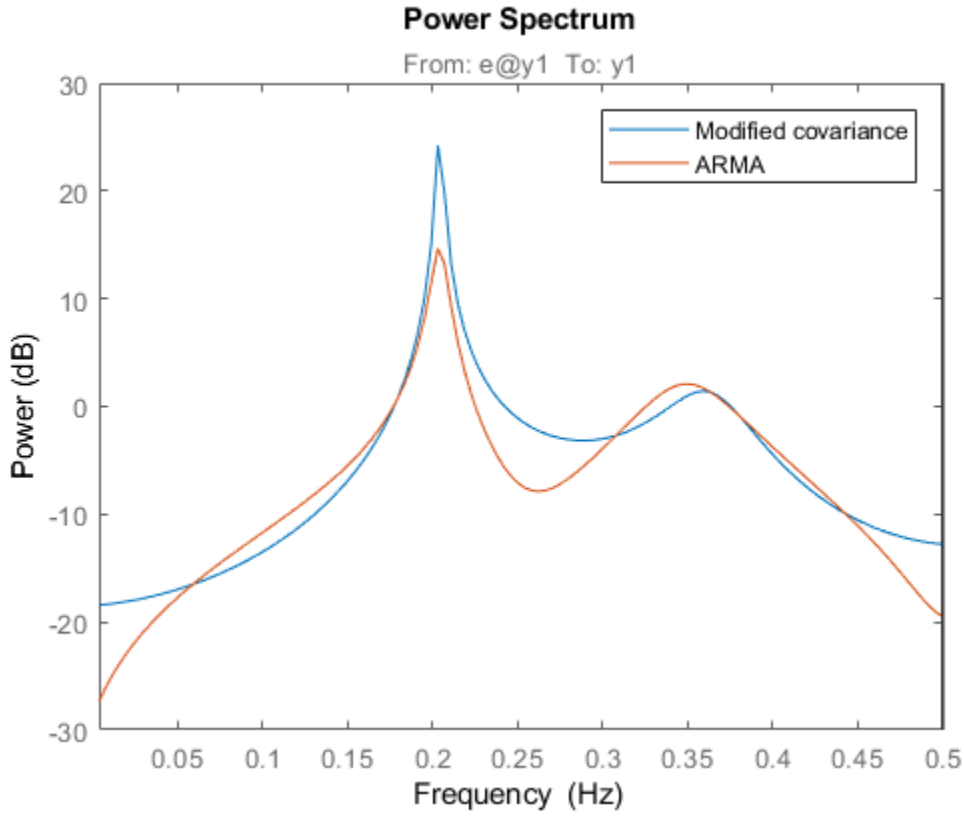
```
ti = ivar(marple,4);
spectrumplot(t5,ti,w,opt);
legend({'Modified covariance','Instrumental Variable'})
```



Autoregressive-Moving Average (ARMA) Model of the Spectra

Furthermore, System Identification Toolbox covers ARMA-modeling of spectra:

```
ta44 = armax(marple,[4 4]); % 4 AR-parameters and 4 MA-parameters
spectrumplot(t5,ta44,w,opt);
legend({'Modified covariance','ARMA'})
```



Additional Information

For more information on identification of dynamic systems with System Identification Toolbox visit the System Identification Toolbox product information page.

Analyze Time-Series Models

This example shows how to analyze time-series models.

A time-series model has no inputs. However, you can use many response computation commands on such models. The software treats (implicitly) the noise source `e(t)` as a measured input. Thus, `step(sys)` plots the step response assuming that the step input was applied to the noise channel `e(t)`.

To avoid ambiguity in how the software treats a time-series model, you can transform it explicitly into an input-output model using `noise2meas`. This command causes the noise input `e(t)` to be treated as a measured input and transforms the linear time series model with `Ny` outputs into an input-output model with `Ny` outputs and `Ny` inputs. You can use the resulting model with commands, such as, `bode`, `nyquist`, and `iopzmap` to study the characteristics of the `H` transfer function.

Estimate a time-series model.

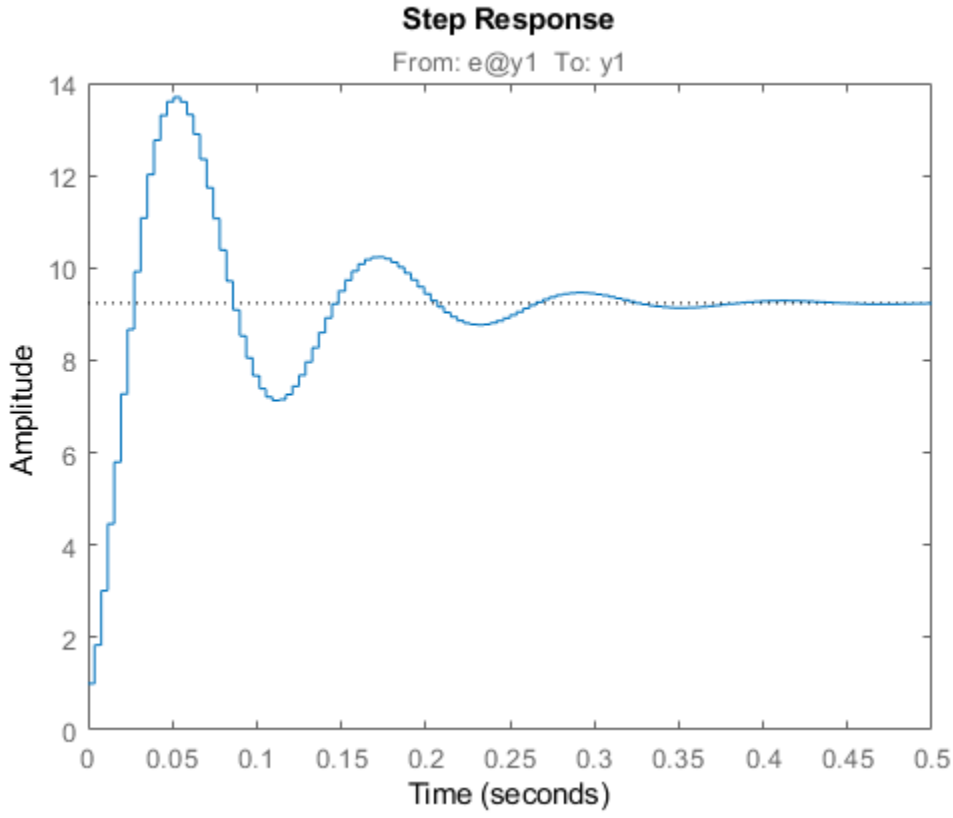
```
load iddata9
sys = ar(z9,4);
```

Convert the time-series model to an input-output model.

```
iosys = noise2meas(sys);
```

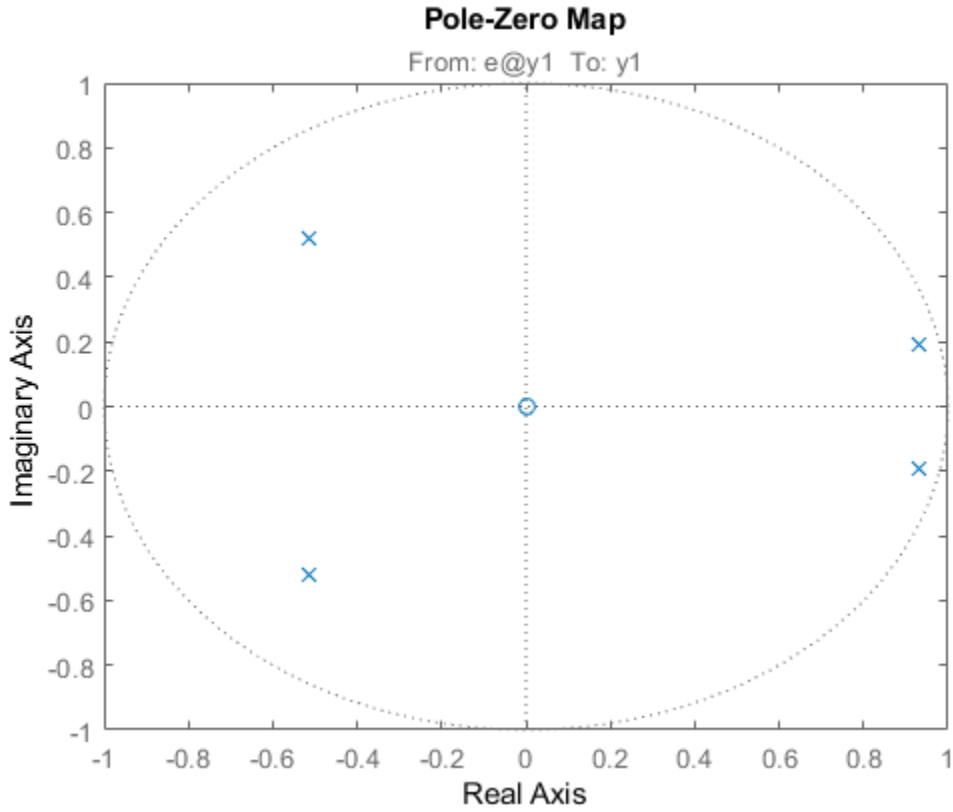
Plot the step response of `H`.

```
step(iosys);
```



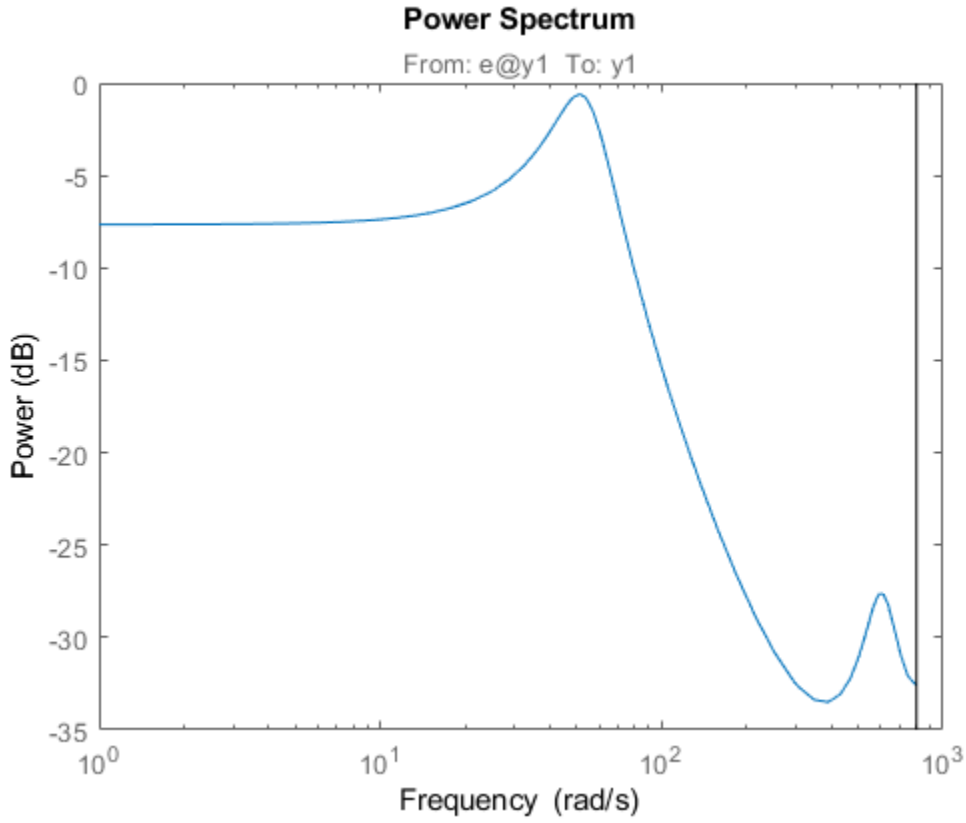
Plot the poles and zeros of `H`.

```
iopzmap(iosys);
```



Calculate and plot the time-series spectrum directly without converting to an input-output model.

```
spectrum(sys);
```



The command plots the time-series spectrum amplitude $\Phi(\omega) = \|H(\omega)\|^2$.

See Also

`noise2meas` | `step`

More About

- "What Are Time Series Models?" on page 14-2

Introduction to Forecasting of Dynamic System Response

Forecasting the response of a dynamic system is the prediction of future outputs of the system using past output measurements. In other words, given observations *y(t)* = {*y(1)*, …, *y(N)*} of the output of a system, forecasting is the prediction of the outputs *y(N+1)*, …, *y(N+H)* until a future time horizon *H*.

When you perform forecasting in System Identification Toolbox software, you first identify a model that fits past measured data from the system. The model can be a linear time series model such as AR, ARMA, and state-space models, or a nonlinear ARX model. If exogenous inputs influence the outputs of the system, you can perform forecasting using input-output models such as ARX and ARMAX. After identifying the model, you then use the `forecast` command to compute *y(N+1)*, …, *y(N+H)*. The command computes the forecasted values by:

- Generating a predictor model using the identified model.
- Computing the final state of the predictor using past measured data.
- Simulating the identified model until the desired forecasting horizon, *H*, using the final state as initial conditions.

This topic illustrates these forecasting steps for linear and nonlinear models. Forecasting the response of systems without external inputs (time series data) is illustrated, followed by forecasting for systems with an exogenous input. For information about how to perform forecasting in the toolbox, see “Forecast the Output of a Dynamic System” on page 14-50.

Forecasting Time Series Using Linear Models

The toolbox lets you forecast time series (output only) data using linear models such as AR, ARMA, and state-space models. Here is an illustration of forecasting the response of an autoregressive model, followed by the forecasting steps for more complex models such as moving-average and state-space models.

Autoregressive Models

Suppose that you have collected time series data *y*(*t*) = {*y(1)*, …, *y(N)*} of a stationary random process. Assuming the data is a second-order autoregressive (AR) process, you can describe the dynamics by the following AR model:

$$y(t) + a_1 y(t-1) + a_2 y(t-2) = e(t)$$

Where a_1 and a_2 are the fit coefficients and $e(t)$ is the noise term.

You can identify the model using the `ar` command. The software computes the fit coefficients and variance of $e(t)$ by minimizing the 1-step prediction errors between the observations {*y(1)*, …, *y(N)*} and model response.

Assuming that the innovations $e(t)$ are a zero mean white sequence, you can compute the predicted output $\hat{y}(t)$ using the formula:

$$\hat{y}(t) = -a_1 y(t-1) - a_2 y(t-2)$$

Where *y(t-1)* and *y(t-2)* are either measured data, if available, or previously predicted values. For example, the forecasted outputs five steps in the future are:

$$\hat{y}(N+1) = -a_1 y(N) - a_2 y(N-1)$$
$$\hat{y}(N+2) = -a_1 \hat{y}(N+1) - a_2 y(N)$$
$$\hat{y}(N+3) = -a_1 \hat{y}(N+2) - a_2 \hat{y}(N+1)$$
$$\hat{y}(N+4) = -a_1 \hat{y}(N+3) - a_2 \hat{y}(N+2)$$
$$\hat{y}(N+5) = -a_1 \hat{y}(N+4) - a_2 \hat{y}(N+3)$$

Note that the computation of $\hat{y}(N+2)$ uses the previously predicted value $\hat{y}(N+1)$ because measured data is not available beyond time step *N*. Thus, the direct contribution of measured data diminishes as you forecast further into the future.

The forecasting formula is more complex for time series processes that contain moving-average terms.

Moving-Average Models

In moving-average (MA) models, the output depends on current and past innovations ($e(t)$,$e(t\text{-}1)$, $e(t\text{-}2)$, $e(t\text{-}3)$....). Thus, forecasting the response of MA models requires knowledge of the initial conditions of the measured data.

Suppose that time series data $y(t)$ from your system can be fit to a second-order moving-average model:

$$y(t) = e(t) + c_1 e(t-1) + c_2 e(t-2)$$

Suppose that `y(1)` and `y(2)` are the only available observations, and their values equal 5 and 10, respectively. You can estimate the model coefficients c_1 and c_2 using the `armax`

command. Assume that the estimated c_1 and c_2 values are 0.1 and 0.2, respectively. Then assuming as before that $e(t)$ is a random variable with zero mean, you can predict the output value at time t using the following formula:

$$\hat{y}(t) = c_1 e(t-1) + c_2 e(t-2)$$

Where `e(t-1)` and `e(t-2)` are the differences between the measured and the predicted response at times `t-1` and `t-2`, respectively. If measured data does not exist for these times, a zero value is used because the innovations process $e(t)$ is assumed to be zero-mean white Gaussian noise.

Therefore, forecasted output at time $t = 3$ is:

$$\hat{y}(3) = 0.1e(2) + 0.2e(1)$$

Where, the innovations `e(1)` and `e(2)` are the difference between the observed and forecasted values of output at time t equal to `1` and `2`, respectively:

$$e(2) = y(2) - y(2) = y(2) - [0.1e(1) + 0.2\ e(0)]$$

$$e(1) = y(1) - y(1) = y(1) - [0.1e(0) + 0.2\ e(-1)]$$

Because the data was measured from time t equal to `1`, the values of `e(0)` and `e(-1)` are unknown. Thus, to compute the forecasted outputs, the value of these initial conditions `e(0)` and `e(-1)` is required. You can either assume zero initial conditions, or estimate them.

- **Zero initial conditions**: If you specify that `e(0)` and `e(-1)` are equal to 0, the error values and forecasted outputs are:

  $$e(1) = 5 - (0.1 * 0 + 0.2 * 0) = 5$$

  $$e(2) = 10 - (0.1 * 5 + 0.2 * 0) = 9.5$$

  $$y(3) = 0.1 * 9.5 + 0.2 * 5 = 1.95$$

  The forecasted values at times $t = 4$ and 5 are:

  $$\hat{y}(4) = 0.1e(3) + 0.2e(2)$$

  $$\hat{y}(5) = 0.1e(4) + 0.2e(3)$$

  Here `e(3)` and `e(4)` are assumed to be zero as there are no measurements beyond time $t = 2$. This assumption yields, $\hat{y}(4) = 0.2 * e(2) = 0.2 * 9.5 = 1.9$, and $\hat{y}(5) = 0$.

Thus, for this second-order MA model, the forecasted outputs that are more than two time steps beyond the last measured data point ($t = 2$) are all zero. In general, when zero initial conditions are assumed, the forecasted values beyond the order of a pure MA model with no autoregressive terms are all zero.

- **Estimated initial conditions**: You can estimate the initial conditions by minimizing the squared sum of 1-step prediction errors of all the measured data.

  For the MA model described previously, estimation of the initial conditions `e(0)` and `e(-1)` requires minimization of the following least-squares cost function:

  $$V = e(1)^2 + e(2)^2 = (y(1) - [0.1\ e(0) + 0.2\ e(-1)])^2 + (y(2) - [0.1\ e(1) + 0.2\ e(0)])^2$$

  Substituting `a = e(0)` and `b = e(-1)`, the cost function is:

  $$V(a,b) = (5 - [0.1a + 0.2b])^2 + (10 - [0.1(5 - [0.1a + 0.2b]) + 0.2a])^2$$

  Minimizing V yields `e(0)` = 50 and `e(-1)` = 0, which gives:

  $$e(1) = 5 - (0.1 * 50 + 0.2 * 0) = 0$$
  $$e(2) = 10 - (0.1 * 0 + 0.2 * 50) = 0$$
  $$y(3) = 0$$
  $$y(4) = 0$$

  Thus, for this system, if the prediction errors are minimized over the available two samples, all future predictions are equal to zero, which is the mean value of the process. If there were more than two observations available, you would estimate `e(-1)` and `e(0)` using a least-squares approach to minimize the 1-step prediction errors over all the available data.

  This example shows how to reproduce these forecasted results using the `forecast` command.

  Load the measured data.

  ```
  PastData = [5;10];
  ```

  Create an MA model with A and C polynomial coefficients equal to 1 and [1 0.1 0.2], respectively.

  ```
  model = idpoly(1,[],[1 0.1 0.2]);
  ```

  Specify zero initial conditions, and forecast the output five steps into the future.

```
opt = forecastOptions('InitialCondition','z');
yf_zeroIC = forecast(model,PastData,5,opt)

yf_zeroIC =

    1.9500
    1.9000
         0
         0
         0
```

Specify that the software estimate initial conditions, and forecast the output.

```
opt = forecastOptions('InitialCondition','e');
yf_estimatedIC = forecast(model,PastData,5,opt)

yf_estimatedIC =

   1.0e-15 *

   -0.3553
   -0.3553
         0
         0
         0
```

For arbitrary structure models, such as models with autoregressive *and* moving-average terms, the forecasting procedure can be involved and is therefore best described in the state-space form.

State-Space Models

The discrete-time state-space model of time series data has the form:

$$x(t+1) = Ax(t) + Ke(t)$$

$$y(t) = Cx(t) + e(t)$$

Where, $x(t)$ is the state vector, $y(t)$ are the outputs, $e(t)$ is the noise-term. A, C, and K are fixed-coefficient state-space matrices.

You can represent any arbitrary structure linear model in state-space form. For example, it can be shown that the ARMA model described previously is expressed in state-space

form using `A = [0 0;1 0]`, `K = [0.5;0]` and `C = [0.2 0.4]`. You can estimate a state-space model from observed data using commands such as `ssest` and `n4sid`. You can also convert an existing polynomial model such as AR, MA, ARMA, ARX, and ARMAX into the state-space form using the `idss` command.

The advantage of state-space form is that any autoregressive or moving-average model with multiple time lag terms (`t-1`,`t-2`,`t-3`,...) only has a single time lag (`t-1`) in state variables when the model is converted to state-space form. As a result, the required initial conditions for forecasting translate into a single value for the initial state vector `X(0)`. The `forecast` command converts all linear model to state-space form and then performs forecasting.

To forecast the response of a state-space model:

1. Generate a 1-step ahead predictor model for the identified model. The predictor model has the form:

   $$\hat{x}(t+1) = (A - K * C)\ \hat{x}(t) + Ky(t)$$
   $$\hat{y}(t) = C * \hat{x}(t)$$

   Where $y(t)$ is the measured output and $\hat{y}(t)$ is the predicted value. The measured output is available until time step N and is used as an input in the predictor model.

   The initial state vector is $\hat{x}(0) = x_0$.

2. Assign a value to the initial state vector x_0.

   The initial states are either specified as zero, or estimated by minimizing the prediction error over the measured data time span.

   Specify a zero initial condition if the system was in a state of rest before the observations were collected. You can also specify zero initial conditions if the predictor model is sufficiently stable because stability implies the effect of initial conditions diminishes rapidly as the observations are gathered. The predictor model is stable if the eigenvalues of `A-K*C` are inside the unit circle.

3. Compute $\hat{x}(N+1)$, the value of the states at the time instant `t = N+1`, the time instant following the last available data sample.

   To do so, simulate the predictor model using the measured observations as inputs:

$$\hat{x}(1) = (A - K * C)\ x_0 + Ky(0)$$
$$\hat{x}(2) = (A - K * C)\hat{x}(1) + Ky(1)$$
$$\vdots$$
$$\hat{x}(N+1) = (A - K * C)\hat{x}(N) + Ky(N)$$

4. Simulate the response of the identified model for `H` steps using $\hat{x}(N+1)$ as initial conditions, where `H` is the prediction horizon. This response is the forecasted response of the model.

Reproduce the Output of forecast Command

This example shows how to manually reproduce forecasting results that are obtained using the `forecast` command. You first use the `forecast` command to forecast time series data into the future. You then compare the forecasted results to a manual implementation of the forecasting algorithm.

Load time series data.

```
load iddata9 z9
```

`z9` is an `iddata` object that stores time series data (no inputs).

Specify data to use for model estimation.

```
observed_data = z9(1:128);
Ts = observed_data.Ts;
t = observed_data.SamplingInstants;
y = observed_data.y;
```

`Ts` is the sample time of the measured data, `t` is the time vector, and `y` is the vector of measured data.

Estimate a discrete-time state space model of 4th order.

```
sys = ssest(observed_data,4,'Ts',Ts);
```

Forecast the output of the state-space model 100 steps into the future using the `forecast` command.

```
H = 100;
yh1 = forecast(sys,observed_data,H);
```

`yh1` is the forecasted output obtained using the `forecast` command. Now reproduce the output by manually implementing the algorithm used by the `forecast` command.

Retrieve the estimated state-space matrices to create the predictor model.

```
A = sys.A;
K = sys.K;
C = sys.C;
```

Generate a 1-step ahead predictor where the `A` matrix of the `Predictor` model is `A-K*C` and the `B` matrix is `K`.

```
Predictor = idss((A-K*C),K,C,0,'Ts',Ts);
```

Estimate initial states that minimize the difference between the observed output `y` and the 1-step predicted response of the identified model `sys`.

```
x0 = findstates(sys,observed_data,1);
```

Propagate the state vector to the end of observed data. To do so, simulate the predictor using `y` as input and `x0` as initial states.

```
Input = iddata([],y,Ts);
opt = simOptions('InitialCondition',x0);
[~,~,x] = sim(Predictor,Input,opt);
xfinal = x(end,:)';
```

`xfinal` is the state vector value at time `t(end)`, the last time instant when observed data is available. Forecasting 100 time steps into the future starts at the next time step, `t1 = t(end)+Ts`.

To implement the forecasting algorithm, the state vector value at time `t1` is required. Compute the state vector by applying the state update equation of the `Predictor` model to `xfinal`.

```
x0_for_forecasting = Predictor.A*xfinal + Predictor.B*y(end);
```

Simulate the identified model for `H` steps using `x0_for_forecasting` as initial conditions.

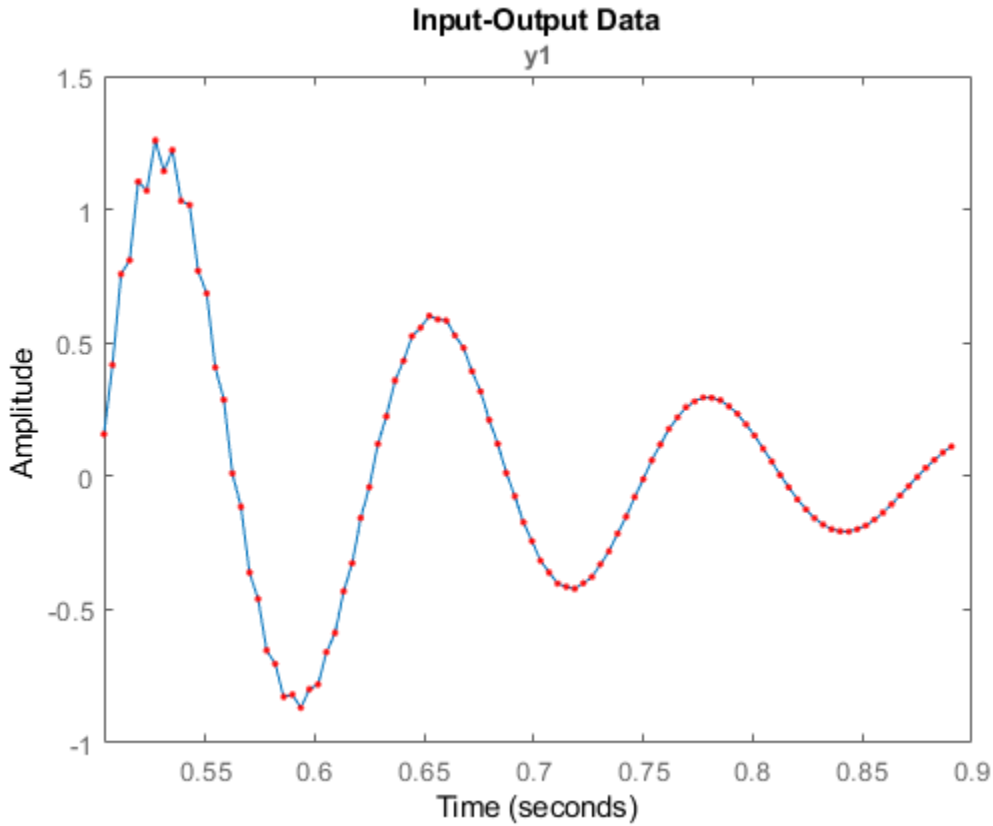
```
opt = simOptions('InitialCondition',x0_for_forecasting);
```

Because `sys` is a time series model, specify inputs for simulation as an `H`-by-0 signal, where `H` is the wanted number of simulation output samples.

```
Input = iddata([],zeros(H,0),Ts,'Tstart',t(end)+Ts);
yh2 = sim(sys,Input,opt);
```

Compare the results of the `forecast` command `yh1` with the manually computed results `yh2`.

```
plot(yh1,yh2,'r.')
```



The plot shows that the results match.

Forecasting Response of Linear Models with Exogenous Inputs

When there are exogenous stimuli affecting the system, the system cannot be considered stationary. However, if these stimuli are measurable then you can treat them as inputs to the system and account for their effects when forecasting the output of the system. The workflow for forecasting data with exogenous inputs is similar to that for forecasting time series data. You first identify a model to fit the measured input-output data. You then specify the anticipated input values for the forecasting time span, and forecast the output of the identified model using the `forecast` command. If you do not specify the anticipated input values, they are assumed to be zero.

This example shows how to forecast an ARMAX model with exogenous inputs in the toolbox:

Load input-output data.

```
load iddata1 z1
```

`z1` is an `iddata` object with input-output data at 300 time points.

Use the first half of the data as past data for model identification.

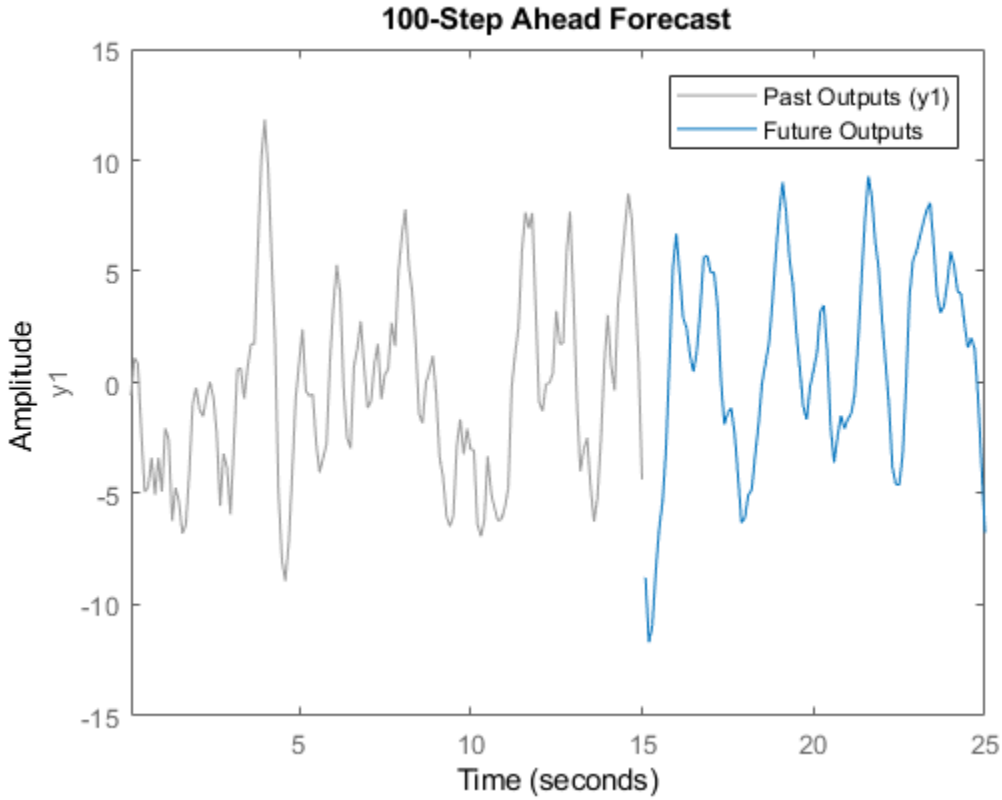
```
past_data = z1(1:150);
```

Identify an ARMAX model Ay(t) = Bu(t-1) + Ce(t), of order [2 2 2 1].

```
na = 2; % A polynomial order
nb = 2; % B polynomial order
nc = 2; % C polynomial order
nk = 1; % input delay
sys = armax(past_data,[na nb nc nk]);
```

Forecast the response 100 time steps into the future, beyond the last sample of observed data `past_data`. Specify the anticipated inputs at the 100 future time points.

```
H = 100;
FutureInputs = z1.u(151:250);
forecast(sys,past_data,H,FutureInputs)
legend('Past Outputs','Future Outputs')
```



Forecasting Response of Nonlinear Models

The toolbox also lets you forecast data using nonlinear ARX, Hammerstein-Wiener, and nonlinear grey-box models.

Hammerstein-Wiener, and nonlinear grey-box models have a trivial noise-component, that is disturbance in the model is described by white noise. As a result, forecasting using the `forecast` command is the same as performing a pure simulation.

Forecasting Response of Nonlinear ARX Models

A time series nonlinear ARX model has the following structure:

$$y(t) = f(y(t-1),\ y(t-2),\ \ldots\ ,\ y(t-N)) + e(t)$$

Where f is a nonlinear function with inputs $R(t)$, the model regressors. The regressors can be the time-lagged variables *y(t-1)*, *y(t-2)*,... , *y(t-N)* and their nonlinear expressions, such as $y(t-1)^2$,*y(t-1)y(t-2)*, *abs(y(t-1))*. When you estimate a nonlinear ARX model from the measured data, you specify the model regressors. You can also specify the structure of f using different structures such as wavelet networks and tree partitions. For more information, see the reference page for the `nlarx` estimation command.

Suppose that time series data from your system can be fit to a second-order linear-in-regressor model with the following polynomial regressors:

$$R(t) = [y(t-1), y(t-2), y(t-1)^2, y(t-2)^2, y(t-1)y(t-2)]^T$$

Then `f(R)=W'*R+c`, where `W=[w1 w2 w3 w4 w5]` is a weighting vector, and `c` is the output offset.

The nonlinear ARX model has the form:

$$y(t) = w_1 y(t-1) + w_2 y(t-2) + w_3 y(t-1)^2 + w_4 y(t-2)^2 + w_5 y(t-1)y(t-2) + c + e(t)$$

When you estimate the model using the `nlarx` command, the software estimates the model parameters `W` and `c`.

When you use the `forecast` command, the software computes the forecasted model outputs by simulating the model H time steps into the future, using the last N measured output samples as initial conditions. Where N is the largest lag in the regressors, and H is the forecast horizon you specify.

For the linear-in-regressor model, suppose that you have measured 100 samples of the output y, and you want to forecast four steps into the future ($H = 4$). The largest lag in the regressors of the model is $N = 2$. Therefore, the software takes the last two samples of the data `y(99)` and `y(100)` as initial conditions, and forecasts the outputs as:

$$\hat{y}(101) = w_1 y(100) + w_2 y(99) + w_3 y(100)^2 + w_4 y(99)^2 + w_5 y(100)y(99)$$

$$\hat{y}(102) = w_1 \hat{y}(101) + w_2 y(100) + w_3 \hat{y}(101)^2 + w_4 y(100)^2 + w_5 \hat{y}(101)y(100)$$

$$\hat{y}(103) = w_1 \hat{y}(102) + w_2 \hat{y}(101) + w_3 \hat{y}(102)^2 + w_4 \hat{y}(101)^2 + w_5 \hat{y}(102)\hat{y}(101)$$

$$\hat{y}(104) = w_1 \hat{y}(103) + w_2 \hat{y}(102) + w_3 \hat{y}(103)^2 + w_4 \hat{y}(102)^2 + w_5 \hat{y}(103)\hat{y}(102)$$

If your system has exogenous inputs, the nonlinear ARX model also includes regressors that depend on the input variables. The forecasting process is similar to that for time series data. You first identify the model, `sys`, using input-output data, `past_data`. When you forecast the data, the software simulates the identified model *H* time steps into the future, using the last *N* measured output samples as initial conditions. You also specify the anticipated input values for the forecasting time span, `FutureInputs`. The syntax for forecasting the response of nonlinear models with exogenous inputs is the same as that for linear models, `forecast(sys,past_data,H,FutureInputs)`.

See Also

`forecast` | `predict` | `sim`

Related Examples

- "Forecast the Output of a Dynamic System" on page 14-50
- "Perform Multivariate Time Series Forecasting" on page 17-25
- "Time Series Prediction and Forecasting for Prognosis" on page 22-2

More About

- "Simulating and Predicting Model Output" on page 17-9

Forecast the Output of a Dynamic System

To forecast the output of a dynamic system, you first identify a model that fits past measured data from the system, and then forecast the future outputs of the identified model. Suppose that you have a set Y of N measurements of the output of a system ($Y = \{y_1, y_2, ...y_N\}$). To forecast the outputs into the future:

1. Identify a model of the system using time series estimation commands such as `ar`, `arx`, `armax`, and `ssest`.

   The software estimates the models by minimizing the squared sum of one-step ahead prediction errors. You can identify linear models such as AR, ARMA, and state-space models. You can also estimate nonlinear ARX and nonlinear grey-box models.

2. Validate the identified model using `predict` command.

   The `predict` command predicts the output of an identified model over the time span of measured data ($Y_p = y_{p1}, y_{p2}, ...y_{pN}$). Use `predict` to determine if the predicted results Y_p match the observed outputs Y for a desired prediction horizon. If the predictions are good over the time span of available data, use the model for forecasting.

3. Specify forecasting options such as how the software computes initial conditions of the measured data. To specify the options, use the `forecastOptions` option set.

4. Compute the output of the identified model until a future time horizon H, ($y_{N+1}, y_{N+2},..., y_{N+H}$) using the `forecast` command. Unlike the `predict` command, the `forecast` command performs prediction into the future, in a time range beyond the last instant of measured data.

   The software computes the forecasted values by:

   - Generating a predictor model using the identified model.
   - Computing the final state of the predictor using measured (available) data.
   - Simulating the identified model using the final state as initial conditions.

   For more information, see “Introduction to Forecasting of Dynamic System Response” on page 14-37.

You can also forecast outputs for systems where measurable exogenous inputs $u(t)$ influence the output observations. In this case, you first identify an input-output model using measured $y(t)$ and $u(t)$, and then use the `forecast` command.

Forecast Time Series Data Using an ARMA Model

This example shows how to forecast time series data from a system using an ARMA model. Load the time series data that is to be forecasted.

```
load iddata9 z9
past_data = z9.OutputData(1:50);
```

Fit an ARMA model of order `[4 3]` to the measured data.

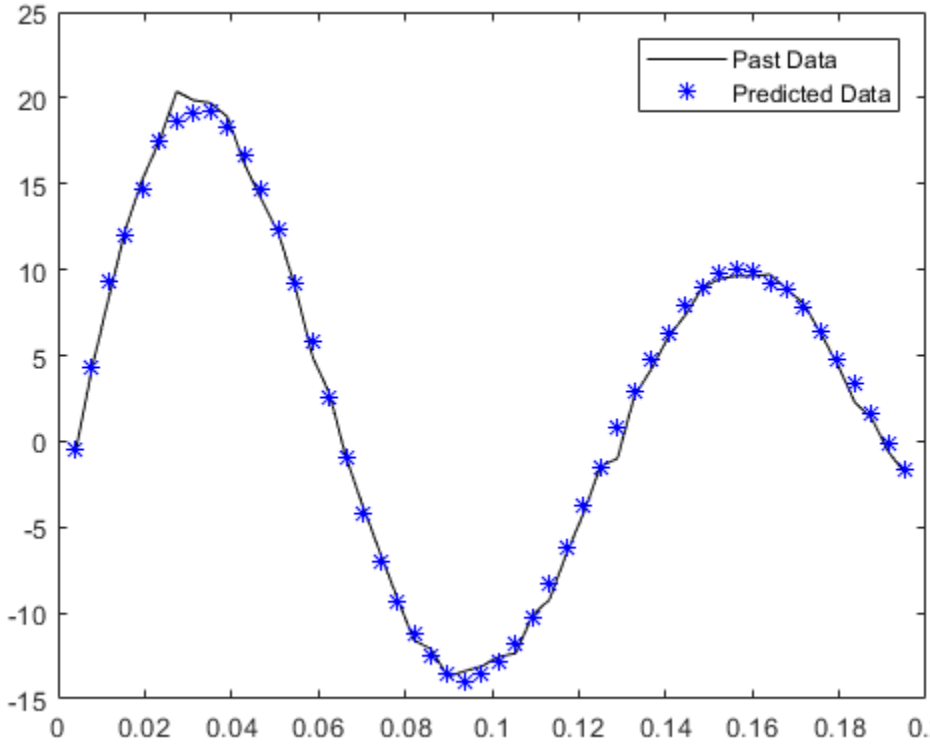
```
sys = armax(past_data,[4 3]);
```

Perform a 10-step ahead prediction to validate the model over the time-span of the measured data.

```
yp = predict(sys,past_data,10);
```

Plot the predicted response and the measured data.

```
t = z9.SamplingInstants;
t1 = t(1:50);
plot(t1,past_data,'k',t1,yp,'*b')
legend('Past Data','Predicted Data')
```



The plot shows that `sys` is a good prediction model that can be used for forecasting.

Specify zero initial conditions for the measured data.

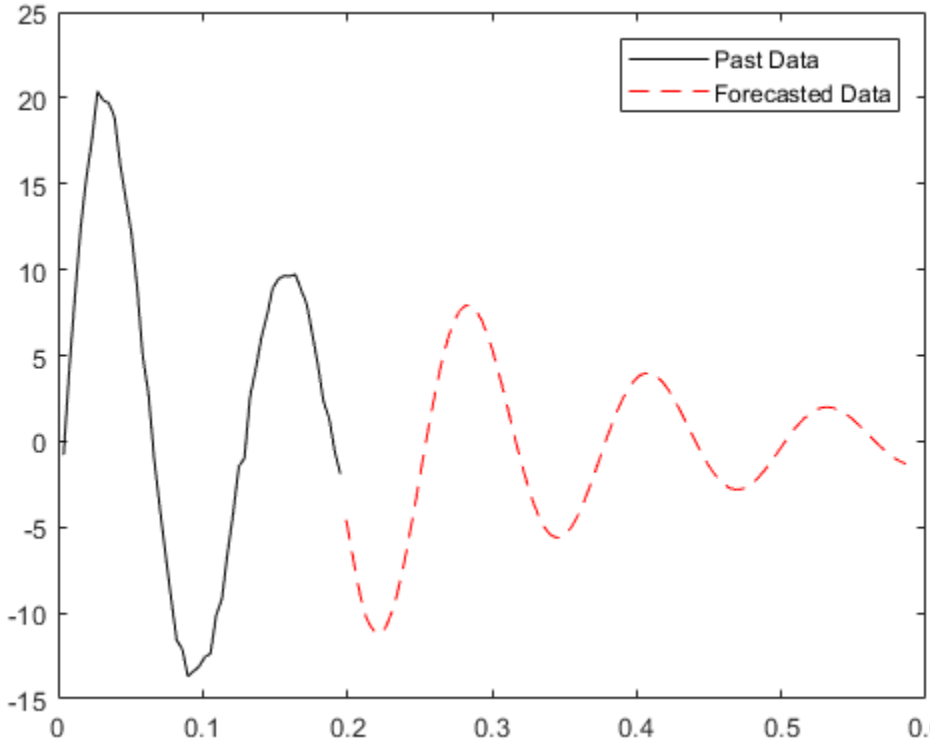
```
opt = forecastOptions('InitialCOndition','z');
```

Forecast model output 100 steps beyond the estimation data.

```
H = 100;
yf = forecast(sys,past_data,H,opt);
```

Plot the past and forecasted data.

```
t2 = t(51:150)';
plot(t1,past_data,'k',t2,yf,'--r')
legend('Past Data','Forecasted Data')
```



See Also

More About

- “Perform Multivariate Time Series Forecasting” on page 17-25
- “Time Series Prediction and Forecasting for Prognosis” on page 22-2

- “Introduction to Forecasting of Dynamic System Response” on page 14-37

Recursive Model Identification

- "Data Segmentation" on page 15-2
- "Detect Abrupt System Changes Using Identification Techniques" on page 15-3

Data Segmentation

For systems that exhibit abrupt changes while the data is being collected, you might want to develop models for separate data segments such that the system does not change during a particular data segment. Such modeling requires identification of the time instants when the changes occur in the system, breaking up the data into segments according to these time instants, and identification of models for the different data segments.

The following cases are typical applications for *data segmentation*:

- Segmentation of speech signals, where each data segment corresponds to a phonem.
- Detection of trend breaks in time series.
- Failure detection, where the data segments correspond to operation with and without failure.
- Estimating different working modes of a system.

Use `segment` to build polynomial models, such as ARX, ARMAX, AR, and ARMA, so that the model parameters are piece-wise constant over time. For detailed information about this command, see the corresponding reference page.

See Also

Related Examples

- “Detect Abrupt System Changes Using Identification Techniques” on page 15-3

Detect Abrupt System Changes Using Identification Techniques

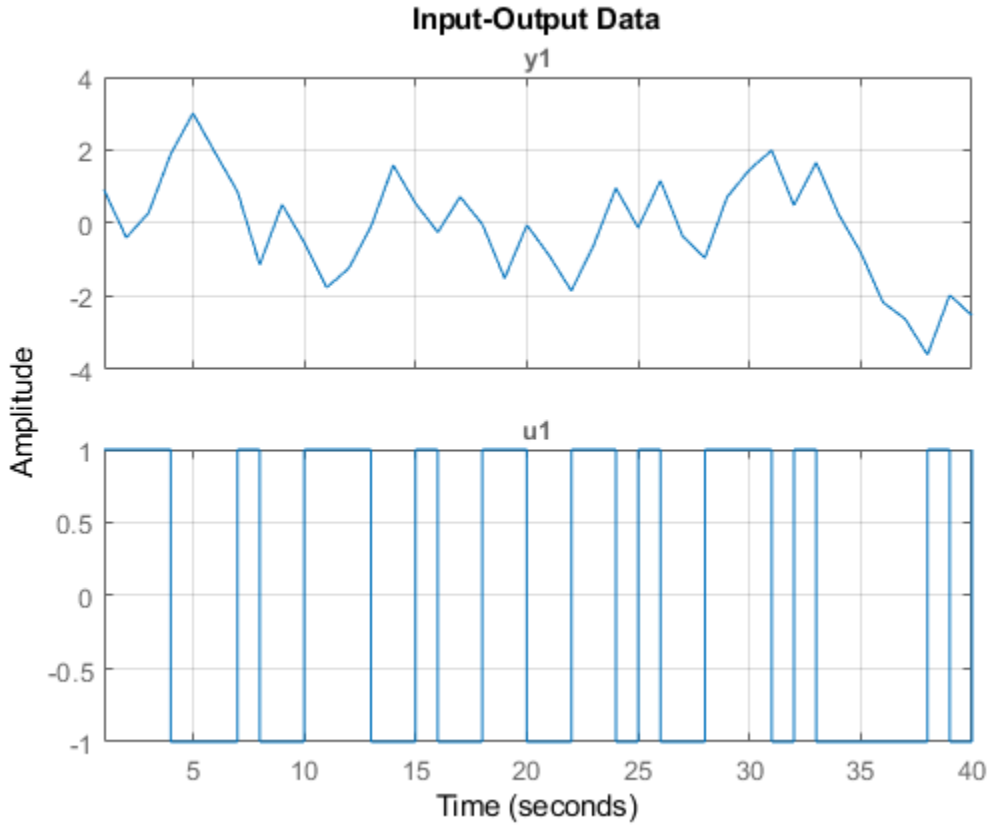
This example shows how to detect abrupt changes in the behavior of a system using online estimation and automatic data segmentation techniques.

Problem Description

Consider a linear system whose transport delay changes from two to one second. Transport delay is the time taken for the input to affect the measured output. In this example, you detect the change in transport delay using online estimation and data segmentation techniques. Input-output data measured from the system is available in the data file `iddemo6m.mat`.

Load and plot the data.

```
load iddemo6m.mat
z = iddata(z(:,1),z(:,2));
plot(z)
grid on
```



The transport delay change takes place around 20 seconds, but is not easy to see in the plot.

Model the system using an ARX structure with one `A` polynomial coefficient, two `B` polynomial coefficients, and one delay.

$$y(t) + ay(t-1) = b1u(t-1) + b2u(t-2)$$

Here, `A = [1 a]` and `B = [0 b1 b2]`.

The leading coefficient of the `B` polynomial is zero because the model has no feedthrough. As the system dynamics change, the values of the three coefficients `a`, `b1`, and `b2` change. When `b1` is close to zero, the effective transport delay will be 2 samples because the `B`

polynomial has 2 leading zeros. When `b1` is larger, the effective transport delay will be 1 sample.

Thus, to detect changes in transport delay you can monitor changes in the `B` polynomial coefficients.

Use Online Estimation for Change Detection

Online estimation algorithms update model parameters and state estimates in a recursive manner, as new data becomes available. You can perform online estimation using Simulink blocks from the System Identification Toolbox library or at the command line using recursive identification routines such as `recursiveARX`. Online estimation can be used to model time varying dynamics such as aging machinery and changing weather patterns, or to detect faults in electromechanical systems.

As the estimator updates the model parameters, a change in system dynamics (delay) will be indicated by a larger than usual change in the values of the parameters `b1` and `b2`. Changes in the `B` polynomial coefficients will be tracked by computing:

$$L(t) = abs(B(t) - B(t-1))$$

Use the `recursiveARX` object for online parameter estimation of the ARX model.

```
na = 1;
nb = 2;
nk = 1;
Estimator = recursiveARX([na nb nk]);
```

Specify the recursive estimation algorithm as `NormalizedGradient` and the adaptation gain as 0.9.

```
Estimator.EstimationMethod = 'NormalizedGradient';
Estimator.AdaptationGain = .9;
```

Extract the raw data from the `iddata` object, `z`.

```
Output = z.OutputData;
Input = z.InputData;
t = z.SamplingInstants;
N = length(t);
```

Use animated lines to plot the estimated parameter values and `L`. Initialize these animated lines prior to estimation. To simulate streaming data, feed the data to the

estimator one sample at a time. Initialize the model parameters before estimation, and then perform online estimation.

```
%% Initialize plot
Colors = {'r','g','b'};
ax = gca;
cla(ax)
for k = 1:3
   h(k) = animatedline('Color',Colors{k}); % lines for a, b1 and b2 parameters
end
h(4) =  animatedline('Marker','.','Color',[0 0 0]); % line for L
legend({'a','b1','b2','Deviation'},'location','southeast')
title('ARX Recursive Parameter Estimation')
xlabel('Time (seconds)')
ylabel('Parameter value')
ax.XLim = [t(1),t(end)];
ax.YLim = [-2, 2];
grid on
box on

%% Now perform recursive estimation and show results
n0 = 6;
L = NaN(N,nk);
B_old = NaN(1,3);
for ct = 1:N
   [A,B] = step(Estimator,Output(ct),Input(ct));
   if ct>n0
      L(ct) = norm(B-B_old);
      B_old = B;
   end
   addpoints(h(1),t(ct),A(2))
   addpoints(h(2),t(ct),B(2))
   addpoints(h(3),t(ct),B(3))
   addpoints(h(4),t(ct),L(ct))
   pause(0.1)
end
```

The first `n0` = 6 samples of the data are not used for computing the change-detector, `L`. During this interval the parameter changes are large owing to the unknown initial conditions.

Find the location of all peaks in `L` by using the `findpeaks` command from Signal Processing Toolbox.

```
[v,Loc] = findpeaks(L);
[~,I] = max(v);
line(t(Loc(I)),L(Loc(I)),'parent',ax,'Marker','o','MarkerEdgeColor','r',...
   'MarkerFaceColor','y','MarkerSize',12)
```

```
fprintf('Change in system delay detected at %g:th sample.\n',t(Loc(I)-1));
```

```
Change in system delay detected at 20:th sample.
```

The location of the largest peak corresponds to the largest change in the `B` polynomial coefficients, and is thus the location of a change in transport delay.

While online estimation techniques provide more options for choosing estimation methods and model structure, the data segmentation method can help automate detection of abrupt and isolated changes.

Use Data Segmentation for Change Detection

A data segmentation algorithm automatically segments the data into regions of different dynamic behavior. This is useful for capturing abrupt changes arising from a failure or change of operating conditions. The `segment` command facilitates this operation for single-output data. `segment` is an alternative to online estimation techniques when you do not need to capture the time-varying behavior during system operation.

Applications of data segmentation include segmentation of speech signals (each segment corresponds to a phonem), failure detection (the segments correspond to operation with and without failures), and estimation of different working modes of a system.

Inputs to the `segment` command include the measured data, the model orders, and a guess for the variance, `r2`, of the noise that affects the system. If the variance is entirely unknown, it can be estimated automatically. Perform data segmentation using an ARX model of the same orders as used for online estimation. Set the variance to 0.1.

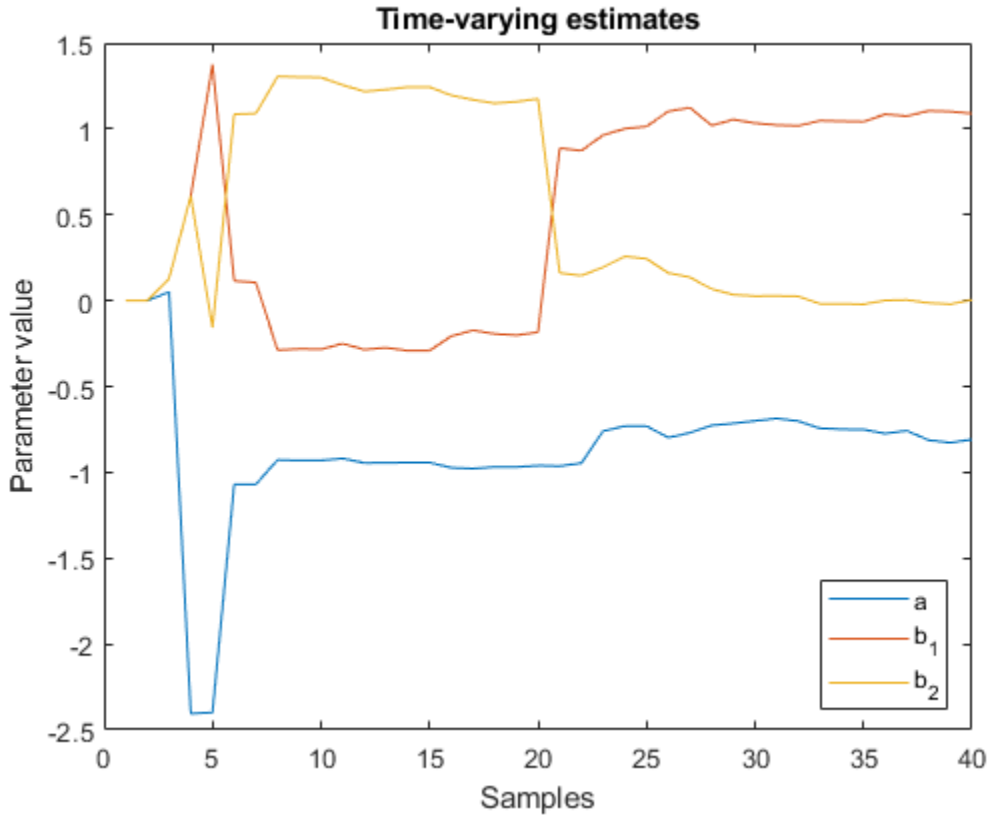
```
[seg,V,tvmod] = segment(z,[na nb nk],0.1);
```

The method for segmentation is based on AFMM (adaptive forgetting through multiple models). For details about the method, see Andersson, Int. J. Control Nov 1985.

A multi-model approach is used to track the time-varying system. The resulting tracking model is an average of the multiple models and is returned as the third output argument of `segment`, `tvmod`.

Plot the parameters of the tracking model.

```
plot(tvmod)
legend({'a','b_1','b_2'},'Location','best')
xlabel('Samples'), ylabel('Parameter value')
title('Time-varying estimates')
```



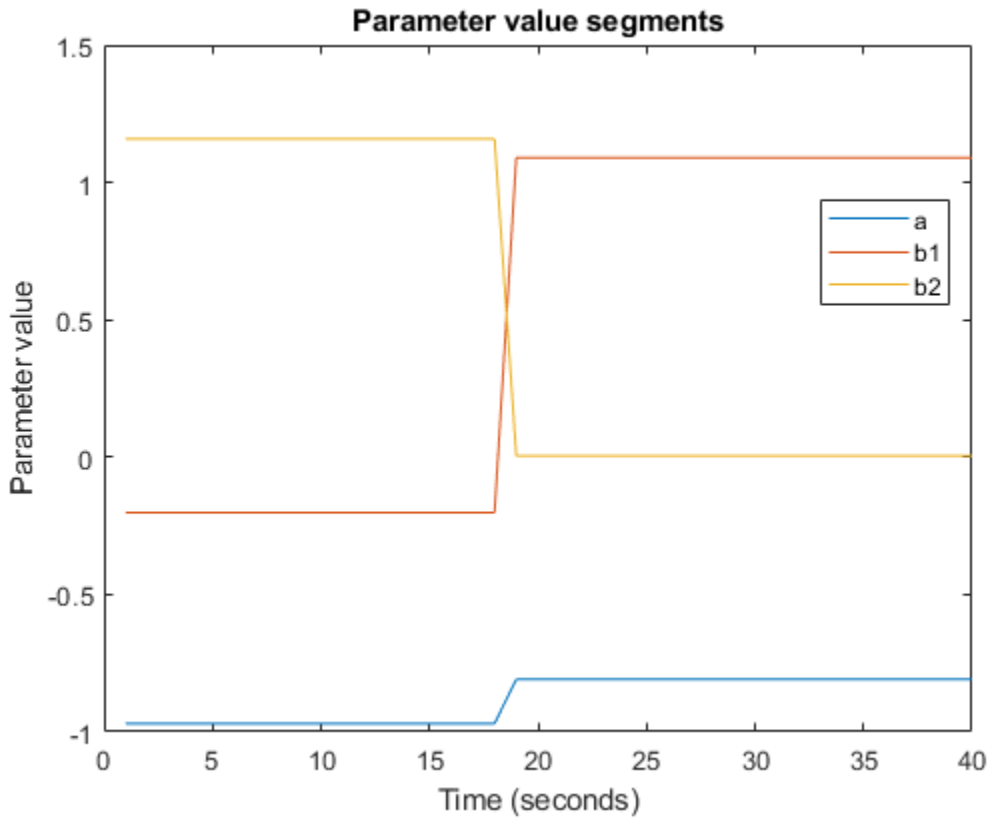
Note the similarity between these parameter trajectories and those estimated using `recursiveARX`.

`segment` determines the time points when changes have occurred using `tvmod` and `q`, the probability that a model exhibits abrupt changes. These time points are used to construct the segmented model by employing a smoothing procedure over the tracking model.

The parameter values of the segmented model are returned in `seg`, the first output argument of `segment`. The values in each successive row are the parameter values of the underlying segmented model at the corresponding time instants. These values remain constant over successive rows and change only when the system dynamics are determined to have changed. Thus, values in `seg` are piecewise constant.

Plot the estimated values for parameters `a`, `b1`, and `b2`.

```
plot(seg)
title('Parameter value segments')
legend({'a','b1','b2'},'Location','best')
xlabel('Time (seconds)')
ylabel('Parameter value')
```



A change is seen in the parameter values around sample number 19. The value of `b1` changes from a small (close to zero) to large (close to 1) value. The value of `b2` shows the opposite pattern. This change in the values of the `B` parameters indicates a change in the transport delay.

The second output argument of `segment`, `V`, is the loss function for the segmented model (i.e. the estimated prediction error variance for the segmented model). You can use `V` to asses the quality of the segmented model.

Note that the two most important inputs for the segmentation algorithm are `r2` and `q`, the fourth input argument to `segment`. In this example, `q` was not specified because the default value, 0.01, was adequate. A smaller value of `r2` and a larger value of `q` will result in more segmentation points. To find appropriate values, you can vary `r2` and `q` and use the ones that work the best. Typically, the segmentation algorithm is more sensitive to `r2` than `q`.

Conclusions

The use of online estimation and data segmentation techniques for detecting abrupt changes in system dynamics was evaluated. Online estimation techniques offer more flexibility and more control over the estimation process. However, for changes that are infrequent or abrupt, `segment` facilitates an automatic detection technique based on smoothing of time-varying parameter estimates.

Online Estimation

- "What Is Online Estimation?" on page 16-2
- "How Online Parameter Estimation Differs from Offline Estimation" on page 16-6
- "Preprocess Online Parameter Estimation Data in Simulink" on page 16-8
- "Validate Online Parameter Estimation Results in Simulink" on page 16-9
- "Validate Online Parameter Estimation at the Command Line" on page 16-11
- "Troubleshoot Online Parameter Estimation" on page 16-14
- "Generate Online Parameter Estimation Code in Simulink" on page 16-18
- "Recursive Algorithms for Online Parameter Estimation" on page 16-20
- "Perform Online Parameter Estimation at the Command Line" on page 16-26
- "Generate Code for Online Parameter Estimation in MATLAB" on page 16-31
- "Extended and Unscented Kalman Filter Algorithms for Online State Estimation" on page 16-35
- "Validate Online State Estimation at the Command Line" on page 16-44
- "Validate Online State Estimation in Simulink" on page 16-47
- "Generate Code for Online State Estimation in MATLAB" on page 16-51
- "Troubleshoot Online State Estimation" on page 16-55
- "Line Fitting with Online Recursive Least Squares Estimation" on page 16-57
- "Online ARX Parameter Estimation for Tracking Time-Varying System Dynamics" on page 16-66
- "Online Recursive Least Squares Estimation" on page 16-82
- "Online ARMAX Polynomial Model Estimation" on page 16-93
- "State Estimation Using Time-Varying Kalman Filter" on page 16-110
- "Nonlinear State Estimation Using Unscented Kalman Filter and Particle Filter" on page 16-124
- "Fault Detection Using an Extended Kalman Filter" on page 16-145
- "Estimate States of Nonlinear System with Multiple, Multirate Sensors" on page 16-160

What Is Online Estimation?

Online estimation algorithms estimate the parameters and states of a model when new data is available during the operation of the physical system. The System Identification Toolbox software uses linear, extended, and unscented Kalman filter, or particle filter algorithms for online state estimation. The toolbox uses recursive prediction error minimization algorithms for online parameter estimation.

Consider a heating and cooling system that does not have prior information about the environment in which it operates. Suppose that this system must heat or cool a room to achieve a certain temperature in a given amount of time. To fulfill its objective, the system must obtain knowledge of the temperature and insulation characteristics of the room. You can estimate the insulation characteristics of the room while the system is online (operational). For this estimation, use the system effort as the input and the room temperature as the output. You can use the estimated model to improve system behavior. Online estimation is ideal for estimating small deviations in the parameter values of a system at a known operating point.

Common applications of online estimation include:

- Adaptive control — Estimate a plant model to modify the controller based on changes in the plant model.
- Fault detection — Compare the online plant model with the idealized or reference plant model to detect a fault (anomaly) in the plant.
- Soft sensing — Generate a “measurement” based on the estimated plant model, and use this measurement for feedback control or fault detection.
- Verification of the experiment-data quality before starting offline estimation — Before using the measured data for offline estimation, perform online estimation for a few iterations. The online estimation provides a quick check of whether the experiment used excitation signals that captured the relevant system dynamics.

Online Parameter Estimation

Online parameter estimation is typically performed using a recursive algorithm. To estimate the parameter values at a time step, recursive algorithms use the current measurements and previous parameter estimates. Therefore, recursive algorithms are efficient in terms of memory usage. Also, recursive algorithms have smaller computational demands. This efficiency makes them suited to online and embedded

applications. For more information about the algorithms, see “Recursive Algorithms for Online Parameter Estimation” on page 16-20.

In System Identification Toolbox you can perform online parameter estimation in Simulink or at the command line:

- In Simulink, use the Recursive Least Squares Estimator and Recursive Polynomial Model Estimator blocks to perform online parameter estimation. You can also estimate a state-space model online from these models by using the Recursive Polynomial Model Estimator and Model Type Converter blocks together. You can generate C/C++ code and Structured Text for these blocks using Simulink Coder and Simulink PLC Coder™ softwares.
- At the command line, use `recursiveAR`, `recursiveARMA`, `recursiveARX`, `recursiveARMAX`, `recursiveOE`, `recursiveBJ`, and `recursiveLS` commands to estimate model parameters for your model structure. Unlike estimation in Simulink, you can change the properties of the recursive estimation algorithm during online estimation. You can generate code and standalone applications using MATLAB Coder and MATLAB Compiler™ software.

When you perform online parameter estimation in Simulink or at the command line, the following requirements apply:

- Model must be discrete-time linear or nearly linear with parameters that vary slowly with time.
- Structure of the estimated model must be fixed during estimation.
- `iddata` object is not supported during online parameter estimation. Specify estimation output data as a real scalar and input data as a real scalar or vector.

Online State Estimation

You can perform online state estimation of systems at the command line and in Simulink:

- In Simulink, use the Kalman Filter, Extended Kalman Filter, and Unscented Kalman Filter blocks to perform online state estimation of discrete-time linear and nonlinear systems. You can generate C/C++ code for these blocks using Simulink Coder software. For the Kalman Filter block, you can also generate Structured Text using Simulink PLC Coder software.
- At the command line, use `extendedKalmanFilter`, `unscentedKalmanFilter` and `particleFilter` commands to estimate states of discrete-time nonlinear systems.

These commands implement discrete-time extended Kalman filter (EKF), unscented Kalman filter (UKF) and particle filter algorithms. For more information about the algorithms, see “Extended and Unscented Kalman Filter Algorithms for Online State Estimation” on page 16-35. You can generate code and standalone applications using MATLAB Coder and MATLAB Compiler softwares.

When you perform online state estimation in Simulink or at the command line, the following requirements apply:

- System must be discrete-time. If you are using the Kalman Filter block, the system can also be continuous-time.
- `iddata` object is not supported during online state estimation. Specify estimation input-output data as real scalars or vectors.

References

[1] Ljung, L. *System Identification: Theory for the User*. Upper Saddle River, NJ: Prentice-Hall PTR, 1999, pp. 428–440.

[2] Simon, D. *Optimal State Estimation: Kalman, H Infinity, and Nonlinear Approaches*. John Wiley and Sons Inc., 2006.

See Also

Functions

`extendedKalmanFilter` | `particleFilter` | `recursiveAR` | `recursiveARMA` | `recursiveARMAX` | `recursiveARX` | `recursiveBJ` | `recursiveLS` | `recursiveOE` | `unscentedKalmanFilter`

Blocks

Extended Kalman Filter | Kalman Filter | Recursive Least Squares Estimator | Recursive Polynomial Model Estimator | Unscented Kalman Filter

Related Examples

- “How Online Parameter Estimation Differs from Offline Estimation” on page 16-6
- “Online ARMAX Polynomial Model Estimation” on page 16-93
- “Line Fitting with Online Recursive Least Squares Estimation” on page 16-57

- “State Estimation Using Time-Varying Kalman Filter” on page 16-110
- “Nonlinear State Estimation Using Unscented Kalman Filter and Particle Filter” on page 16-124

How Online Parameter Estimation Differs from Offline Estimation

Online estimation algorithms estimate the parameters of a model when new data is available during the operation of the model. In offline estimation, you first collect all the input/output data and then estimate the model parameters . Parameter values estimated using online estimation can vary with time, but parameters estimated using offline estimation do not.

To perform offline estimation, use commands such as `arx`, `pem`, `ssest`, `tfest`, `nlarx`, and the System Identification app.

To perform online parameter estimation in Simulink, use the Recursive Least Squares Estimator and Recursive Polynomial Model Estimator blocks. For online estimation at the command line on page 16-26, use commands such as `recursiveARX` to create a System object™, and then use `step` command to update the model parameters.

Online estimation differs from offline estimation in the following ways:

- Model delays — You can estimate model delays in offline estimation using tools such as `delayest` (see "Determining Model Order and Delay" on page 4-48). Online estimation provides limited ability to estimate delays. For polynomial model estimation using the Recursive Polynomial Model Estimation block or the online estimation commands, you can specify a known value of the input delay (nk). If nk is unknown, choose a sufficiently large value for the number of coefficients of B (nb). The number of leading coefficients of the estimated B polynomial that are close to zero represent the input delay.
- Data preprocessing — For offline estimation data preprocessing, you can use functions such as `detrend`, `retrend`, `idfilt`, and the System Identification app.

  For online estimation using Simulink, use the tools available in the Simulink environment. For more information, see "Preprocess Online Parameter Estimation Data in Simulink" on page 16-8.

  For online parameter estimation at the command line, you cannot use preprocessing tools in System Identification Toolbox. These tools support only data specified as `iddata` objects. Implement preprocessing code as required by your application. To be able to generate C and C++ code, use commands supported by MATLAB Coder. For a list of these commands, see "Functions and Objects Supported for C/C++ Code Generation — Category List" (MATLAB Coder).

- Resetting of estimation — You cannot reset offline estimation. Online estimation lets you reset the estimation at a specific time step during estimation. For example, reset the estimation when the system changes modes or if you are not satisfied with the estimation. The reset operation sets the model states, estimated parameters, and estimated parameter covariance to their initial values.

  To reset online estimation in Simulink, in the online estimation block dialog box, on the **Algorithm and Block Options** tab, select the appropriate **External reset** option. At the command line, use the `reset` command.

- Enabling or disabling of estimation — You cannot selectively enable or disable offline estimation. You can use preprocessing tools to remove or filter certain portions of the data before the estimation. Online estimation lets you enable or disable estimation for chosen time spans. For example, suppose that the measured data is especially noisy or faulty (contains many outliers) for a specific time interval. Disable online estimation for this interval.

  To enable or disable estimation in Simulink, in the online estimation block dialog box, on the **Algorithm and Block Options** tab, select the **Add enable port** check box.

  At the command line, use the `EnableAdaptation` property of the System object created using online estimation commands, such as `recursiveARMAX` and `recursiveLS`. Even if you set `EnableAdaptation` to `false`, execute the `step` command. Do not skip `step` to keep parameter values constant because parameter estimation depends on current and past input/output measurements. `step` ensures that past input-output data is stored, even when it does not update the parameters.

See Also

More About

- “What Is Online Estimation?” on page 16-2

Preprocess Online Parameter Estimation Data in Simulink

Estimation data that contains deficiencies, such as drift, offset, missing samples, seasonalities, equilibrium behavior, and outliers, can adversely affect the quality of the estimation. Therefore, it is recommended that you preprocess your estimation data as needed.

Use the tools in the Simulink software to preprocess data for online estimation. Common tools to perform data preprocessing in Simulink are:

- Blocks in the Math Operations (Simulink) library. Use these blocks, for example, to subtract or add an offset or normalize a signal.
- Blocks in the “Continuous” (Simulink) and “Discrete” (Simulink) library. Use these blocks, for example, to filter a signal.
- Rate Transition block, which allows you to handle the transfer of data between blocks operating at different rates. Use this block, for example, to resample your data from a source that is operating at a different sampling rate than the online estimation block.
- MATLAB Function block, which allows you to include MATLAB code in your model. Use this block, for example, to implement a custom preprocessing algorithm.

See “Online ARMAX Polynomial Model Estimation” on page 16-93 for an example of how you can preprocess an estimation input signal by removing its mean.

See Also

Kalman Filter | Recursive Least Squares Estimator | Recursive Polynomial Model Estimator

More About

- “What Is Online Estimation?” on page 16-2

Validate Online Parameter Estimation Results in Simulink

Use the following approaches to validate an online estimation performed using the Recursive Least Squares Estimator or Recursive Polynomial Model Estimator block:

- Examine the estimation error (residuals), which is the difference between the measured and estimated outputs. The estimation error can be large at the beginning of the estimation or when there are large parameter variations. The error should get smaller as the parameter estimates converge. Small errors with respect to the size of the outputs give confidence in the estimated values.

  You can also analyze the residuals using techniques such as the whiteness test and the independence test. For such analysis, use the measured data and estimation error collected after the parameter values have settled to approximately constant values. For more information regarding these tests, see "What Is Residual Analysis?" on page 17-43

  To obtain the estimation error, in the **Algorithm and Block Options** tab of the online estimation block's dialog, select the **Output estimation error** check box. The software adds an Error outport to the block, which you can monitor using a Scope block. This outport provides the one-step-ahead estimation error, $e(t) = y(t)–y_{est}(t)$. For the time step, t, y and y_{est} are the measured and estimated outputs, respectively.
- The parameter covariance is a measure of estimated uncertainty in the parameters, and is calculated when the forgetting factor or Kalman filter estimation algorithms are used.

  Parameter covariance is computed assuming that the residuals are white noise, and the variance of these residuals is 1. To obtain the parameter covariance, in the **Algorithm and Block Options** tab of the online estimation block's dialog, select the **Output parameter covariance matrix** check box. The software adds a Covariance outport to the block, which you can monitor using a Display block. This outport provides the parameter covariance matrix, `P`.

  The estimated parameters can be considered as random variables with variance equal to the corresponding diagonal of the parameter covariance matrix, scaled by the variance of the residuals (`residualVariance`) at each time step. You use prior knowledge, or calculate `residualVariance` from the residuals, `e`. Where, `e` is the vector of estimation errors, $e(t)$.

  ```
  residualVariance = var(e);
  ```

Scale the parameter covariance to calculate the variance of the estimated parameters.

```
paramVariance = diag(P)*residualVariance;
```

A smaller variance value gives confidence in the estimated values.

- Simulate the estimated model and compare the simulated and measured outputs. To do so, feed the measured input into a model that uses the estimated time-varying parameter values. Then, compare the model output with the measured output. The simulated output closely matching the measured output gives confidence in the estimated values.

For examples of such validation, see “Online Recursive Least Squares Estimation” on page 16-82 and “Online ARMAX Polynomial Model Estimation” on page 16-93.

If the validation indicates low confidence in the estimation, then refer to the Troubleshooting topics on the “Online Estimation” page.

See Also

Kalman Filter | Recursive Least Squares Estimator | Recursive Polynomial Model Estimator

Validate Online Parameter Estimation at the Command Line

In this section...
“Examine the Estimation Error” on page 16-11
“Simulate the Estimated Model” on page 16-11
“Examine Parameter Covariance” on page 16-12
“Use Validation Commands from System Identification Toolbox” on page 16-12

This topic shows how to validate online parameter estimation at the command line. If the validation indicates low confidence in the estimation, then see “Troubleshoot Online Parameter Estimation” on page 16-14. After you have validated the online estimation results, you can generate C/C++ code or a standalone application using MATLAB Coder or MATLAB Compiler.

Examine the Estimation Error

The estimation error is the difference between measured output, `y`, and the estimated output, `EstimatedOutput` at each time step.

```
obj = recursiveARX;
[A,B,EstimatedOutput] = step(obj,y,u);
estimationError = y-EstimatedOutput;
```

Here, `u` is the input data at that time step.

The estimation errors (*residuals*) can be large at the beginning of the estimation or when there are large parameter variations. The error should get smaller as the parameter estimates converge. Small errors relative to the size of the outputs increase confidence in the estimated values.

Simulate the Estimated Model

Simulate the estimated model and compare the simulated and measured outputs. To do so, feed the measured input into a model that uses the estimated time-varying parameter values. Then compare the model output with the measured output. A close match between the simulated output and the measured output gives confidence in the estimated values.

Examine Parameter Covariance

Parameter covariance is a measure of estimated uncertainty in the parameters. The covariance is calculated when the forgetting factor or Kalman filter estimation algorithms are used.

Parameter covariance is computed assuming that the residuals are white noise, and the variance of these residuals is 1. You view the parameter covariance matrix using the `ParameterCovariance` property of your System object.

```
P = obj.ParameterCovariance;
```

The estimated parameters can be considered as random variables with variance equal to the corresponding diagonal of the parameter covariance matrix, scaled by the variance of the residuals (`residualVariance`) at each time step. You use prior knowledge, or calculate `residualVariance` from the residuals, `e`. Where, `e` is the vector of estimation errors, `estimationError`.

```
 residualVariance = var(e);
```

Scale the parameter covariance to calculate the variance of the estimated parameters.

```
paramVariance = diag(P)*residualVariance;
```

A smaller variance value gives confidence in the estimated values.

Use Validation Commands from System Identification Toolbox

You can validate a snapshot of the estimated model using validation commands for offline estimation. This validation only captures the behavior of a time-invariant model. For available offline validation techniques in System Identification Toolbox, see “Model Validation”.

To use offline commands, convert your online estimation System object, `obj`, into an `idpoly` model object. Also convert your stream of input-output validation data, *Output(t)* and *Input(t)*, into an `iddata` object.

```
sys = idpoly(obj);
sys.Ts = Ts;
z = iddata(Output,Input,Ts)
```

Here, `Ts` is the sample time.

Note This conversion and any subsequent analysis are not supported by MATLAB Coder.

The validation techniques include:

- Analysis of the residuals using techniques such as the whiteness test and the independence test using offline commands such as `resid`. For example, use `resid(z,sys)`. For information about these tests, see "What Is Residual Analysis?" on page 17-43.
- Comparison of model output and measured output. For example, use `compare(z,sys)`.
- Comparison of different online estimation System objects.

  You can create multiple versions of a System object with different object properties, convert each of them to `idpoly` model objects, and use `compare` to choose the best one.

  If you want to copy an existing System 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 System object created this way (`obj2`) also change the properties of the original System object (`obj`).

See Also

`recursiveAR` | `recursiveARMA` | `recursiveARMAX` | `recursiveARX` | `recursiveBJ` | `recursiveLS` | `recursiveOE`

Related Examples

- "Perform Online Parameter Estimation at the Command Line" on page 16-26
- "Generate Code for Online Parameter Estimation in MATLAB" on page 16-31
- "Line Fitting with Online Recursive Least Squares Estimation" on page 16-57
- "Online ARX Parameter Estimation for Tracking Time-Varying System Dynamics" on page 16-66

Troubleshoot Online Parameter Estimation

In this section...
“Model Structure” on page 16-14
“Model Order” on page 16-15
“Estimation Data” on page 16-15
“Initial Guess for Parameter Values” on page 16-16
“Estimation Settings” on page 16-16

To troubleshoot online parameter estimation, check the following:

Model Structure

Check that you are using the simplest model structure that adequately captures the system dynamics.

AR and ARX model structures are good first candidates for estimating linear models. The underlying estimation algorithms for these model structures are simpler than those for ARMA, ARMAX, Output-Error, and Box-Jenkins model structures. In addition, these simpler AR and ARX algorithms are less sensitive to initial parameter guesses.

The more generic recursive least squares (RLS) estimation also has the advantage of algorithmic simplicity like AR and ARX model estimation. RLS lets you estimate parameters for a wider class of models than ARX and AR and can include nonlinearities. However, configuring an AR or ARX structure is simpler.

Consider the following when choosing a model structure:

- AR and ARX model structures — If you are estimating a time-series model (no inputs), try the AR model structure. If you are estimating an input-output model, try the ARX model structure. Also try different model orders with these model structures. These models estimate the system output based on time-shifted versions of the output and inputs signals. For example, the a and b parameters of the system $y(t) = b_1u(t) + b_2u(t\text{-}1) - a_1y(t\text{-}1)$ can be estimated using ARX models.

  For more information regarding AR and ARX models, see “What Are Polynomial Models?” on page 6-2.

- RLS estimation— If you are estimating a system that is linear in the estimated parameters, but does not fit into AR or ARX model structures, try RLS estimation. You can estimate the system output based on the time-shifted versions of input-outputs signals like the AR and ARX, and can also add nonlinear functions. For example, you can estimate the parameters p_1, p_2, and p_3 of the system $y(t) = p_1y(t\text{-}1) + p_2u(t\text{-}1) + p_3u(t\text{-}1)^2$. You can also estimate static models, such as the line-fit problem of estimating parameters a and b in $y(t) = au(t)+b$.
- ARMA, ARMAX, Output-Error, Box-Jenkins model structures — These model structures provide more flexibility compared to AR and ARX model structures to capture the dynamics of linear systems. For instance, an ARMAX model has more dynamic elements (C polynomial parameters) compared to ARX for estimating noise models. This flexibility can help when AR and ARX are not sufficient to capture the system dynamics of interest.

  Specifying initial parameter and parameter covariance values are especially recommended for these model structures. This is because the estimation method used for these model structures can get stuck at a local optima. For more information about these models, see “What Are Polynomial Models?” on page 6-2.

Model Order

Check the order of your specified model structure. You can under-fit (model order is too low) or over-fit (model order is too high) data by choosing an incorrect model order.

Ideally, you want the lowest-order model that adequately captures your system dynamics. Under-fitting prevents algorithms from finding a good fit to the model, even if all other estimation settings are good, and there is good excitation of system dynamics. Over-fitting typically leads to high sensitivity of parameters to the measurement noise or the choice of input signals.

Estimation Data

Use inputs that excite the system dynamics adequately. Simple inputs, such as a step input, typically does not provide sufficient excitation and are good for estimating only a very limited number of parameters. One solution is to inject extra input perturbations.

Estimation data that contains deficiencies can lead to poor estimation results. Data deficiencies include drift, offset, missing samples, equilibrium behavior, seasonalities, and outliers. It is recommended that you preprocess the estimation data as needed.

For information on how to preprocess estimation data in Simulink, see “Preprocess Online Parameter Estimation Data in Simulink” on page 16-8.

For online parameter estimation at the command line, you cannot use preprocessing tools in System Identification Toolbox. These tools support only data specified as `iddata` objects. Implement preprocessing code as required by your application. To be able to generate C and C++ code, use commands supported by MATLAB Coder. For a list of these commands, see “Functions and Objects Supported for C/C++ Code Generation — Category List” (MATLAB Coder).

Initial Guess for Parameter Values

Check the initial guesses you specify for the parameter values and initial parameter covariance matrix. Specifying initial parameter guesses and initial parameter covariance matrix is recommended. These initial guesses could be based on your knowledge of the system or be obtained via offline estimation.

Initial parameter covariance represents the uncertainty in your guess for the initial values. When you are confident about your initial parameter guesses, and if the initial parameter guesses are much smaller than the default initial parameter covariance value, `10000`, specify a smaller initial parameter covariance. Typically, the default initial parameter covariance is too large relative to the initial parameter values. The result is that initial guesses are given less importance during estimation.

Initial parameter and parameter covariance guesses are especially important for ARMA, ARMAX, Output-Error, and Box-Jenkins models. Poor or no guesses can result in the algorithm finding a local minima of the objective function in the parameter space, which can lead to a poor fit.

Estimation Settings

Check that you have specified appropriate settings for the estimation algorithm. For example, for the forgetting factor algorithm, choose the forgetting factor, λ, carefully. If λ is too small, the estimation algorithm assumes that the parameter value is varying quickly with time. Conversely, if λ is too large, the estimation algorithm assumes that the parameter value does not vary much with time. For more information regarding the estimation algorithms, see “Recursive Algorithms for Online Parameter Estimation” on page 16-20.

See Also

Related Examples

- "What Is Online Estimation?" on page 16-2
- "Validate Online Parameter Estimation Results in Simulink" on page 16-9
- "Validate Online Parameter Estimation at the Command Line" on page 16-11

Generate Online Parameter Estimation Code in Simulink

You can generate C/C++ code and Structured Text for Recursive Least Squares Estimator and other online estimation blocks using products such as Simulink Coder and Simulink PLC Coder. The Model Type Converter block, which you can use with the Recursive Polynomial Model Estimator block, also supports code generation. Use the generated code to deploy online model estimation to an embedded target. For example, you can estimate the coefficients of a time-varying plant from measured input-output data and feed the coefficients to an adaptive controller. After validating the online estimation in simulation, you can generate code for your Simulink model and deploy that code to the target.

To generate code for online estimation, use the following workflow:

1. Develop a Simulink model that simulates the online model estimation. For example, create a model that simulates the input/output data, performs online estimation for this data, and uses the estimated parameter values.
2. After validating the online estimation performance in simulation, create a subsystem for the online estimation block. If you preprocess the inputs or postprocess the parameter estimates, include the relevant blocks in the subsystem.
3. Convert the subsystem to a referenced model. You generate code for this referenced model, so ensure that it uses only the blocks that support code generation. For a list of blocks that support code generation, see “Simulink Built-In Blocks That Support Code Generation” (Simulink Coder).

   The original model, which now contains a model reference, is now referred to as the top model.
4. In the top model, replace the model source and sink blocks with their counterpart hardware blocks. For example, replace the simulated inputs/output blocks with the relevant hardware source block. You generate code for this model, which includes the online estimation. So, ensure that it uses only blocks that support code generation.
5. Generate code for the top model.

For details on configuring the subsystem and converting it to a referenced model, see the “Generate Code for Referenced Models” (Simulink Coder) example.

See Also

Kalman Filter | Model Type Converter | Recursive Least Squares Estimator | Recursive Polynomial Model Estimator

Related Examples

- “Generate Code for Referenced Models” (Simulink Coder)

More About

- “Code Generation of Referenced Models” (Simulink Coder)
- “Simulink Built-In Blocks That Support Code Generation” (Simulink Coder)

Recursive Algorithms for Online Parameter Estimation

In this section...
"General Form of Recursive Estimation" on page 16-20
"Types of Recursive Estimation Algorithms" on page 16-21

General Form of Recursive Estimation

The general form of the recursive estimation algorithm is as follows:

$$\hat{\theta}(t) = \hat{\theta}(t-1) + K(t)\left(y(t) - \hat{y}(t)\right)$$

$\hat{\theta}(t)$ is the parameter estimate at time *t*. *y(t)* is the observed output at time *t* and $\hat{y}(t)$ is the prediction of *y(t)* based on observations up to time *t-1*. The gain, *K(t)*, determines how much the current prediction error $y(t) - \hat{y}(t)$ affects the update of the parameter estimate. The estimation algorithms minimize the prediction-error term $y(t) - \hat{y}(t)$.

The gain has the following form:

$$K(t) = Q(t)\psi(t)$$

The recursive algorithms supported by the System Identification Toolbox product differ based on different approaches for choosing the form of *Q(t)* and computing $\psi(t)$, where $\psi(t)$ represents the gradient of the predicted model output $\hat{y}(t \mid \theta)$ with respect to the parameters θ.

The simplest way to visualize the role of the gradient $\psi(t)$ of the parameters, is to consider models with a linear-regression form:

$$y(t) = \psi^T(t)\theta_0(t) + e(t)$$

In this equation, $\psi(t)$ is the *regression vector* that is computed based on previous values of measured inputs and outputs. $\theta_0(t)$ represents the true parameters. *e(t)* is the noise

source (*innovations*), which is assumed to be white noise. The specific form of $\psi(t)$ depends on the structure of the polynomial model.

For linear regression equations, the predicted output is given by the following equation:

$$\hat{y}(t) = \psi^T(t)\hat{\theta}(t-1)$$

For models that do not have the linear regression form, it is not possible to compute exactly the predicted output and the gradient $\psi(t)$ for the current parameter estimate $\hat{\theta}(t-1)$. To learn how you can compute approximation for $\psi(t)$ and $\hat{\theta}(t-1)$ for general model structures, see the section on recursive prediction-error methods in [1].

Types of Recursive Estimation Algorithms

The System Identification Toolbox software provides the following recursive estimation algorithms for online estimation:

- "Forgetting Factor" on page 16-21
- "Kalman Filter" on page 16-22
- "Normalized and Unnormalized Gradient" on page 16-24

The forgetting factor and Kalman Filter formulations are more computationally intensive than gradient and unnormalized gradient methods. However, they typically have better convergence properties.

Forgetting Factor

The following set of equations summarizes the *forgetting factor* adaptation algorithm:

$$\hat{\theta}(t) = \hat{\theta}(t-1) + K(t)\left(y(t) - \hat{y}(t)\right)$$

$$\hat{y}(t) = \psi^T(t)\hat{\theta}(t-1)$$

$$K(t) = Q(t)\psi(t)$$

$$Q(t) = \frac{P(t-1)}{\lambda + \psi^T(t)P(t-1)\psi(t)}$$

$$P(t) = \frac{1}{\lambda}\left(P(t-1) - \frac{P(t-1)\psi(t)\psi(t)^T P(t-1)}{\lambda + \psi(t)^T P(t-1)\psi(t)} \right)$$

The software ensures *P(t)* is a positive-definite matrix by using a square-root algorithm to update it [2]. The software computes `P` assuming that the residuals (difference between estimated and measured outputs) are white noise, and the variance of these residuals is 1. R_2`/2 * P` is approximately equal to the covariance matrix of the estimated parameters, where R_2 is the true variance of the residuals.

Q(t) is obtained by minimizing the following function at time *t*:

$\sum_{k=1}^{t} \lambda^{t-k}(y(k) - y(k))^2$

See section 11.2 in [1] for details.

This approach discounts old measurements exponentially such that an observation that is τ samples old carries a weight that is equal to λ^{τ} times the weight of the most recent observation. $\tau = 1/(1-\lambda)$ represents the *memory horizon* of this algorithm. Measurements older than $\tau = 1/(1-\lambda)$ typically carry a weight that is less than about 0.3.

λ is called the forgetting factor and typically has a positive value between `0.98` and `0.995`. Set $\lambda = 1$ to estimate time-invariant (constant) parameters. Set $\lambda < 1$ to estimate time-varying parameters.

Note The forgetting factor algorithm for $\lambda = 1$ is equivalent to the Kalman filter algorithm with R_1=0 and R_2=1. For more information about the Kalman filter algorithm, see "Kalman Filter" on page 16-22.

Kalman Filter

The following set of equations summarizes the *Kalman filter* adaptation algorithm:

$$\hat{\theta}(t) = \hat{\theta}(t-1) + K(t)\left(y(t) - \hat{y}(t)\right)$$

$$\hat{y}(t) = \psi^T(t)\hat{\theta}(t-1)$$

$$K(t) = Q(t)\psi(t)$$

$$Q(t) = \frac{P(t-1)}{R_2 + \psi^T(t)P(t-1)\psi(t)}$$

$$P(t) = P(t-1) + R_1 - \frac{P(t-1)\psi(t)\psi(t)^T P(t-1)}{R_2 + \psi(t)^T P(t-1)\psi(t)}$$

The software ensures *P(t)* is a positive-definite matrix by using a square-root algorithm to update it [2]. The software computes `P` assuming that the residuals (difference between estimated and measured outputs) are white noise, and the variance of these residuals is 1. R_2* `P` is the covariance matrix of the estimated parameters, and R_1 /R_2 is the covariance matrix of the parameter changes. Where, R_1 is the covariance matrix of parameter changes that you specify.

This formulation assumes the linear-regression form of the model:

$$y(t) = \psi^T(t)\theta_0(t) + e(t)$$

Q(t) is computed using a Kalman filter.

This formulation also assumes that the true parameters $\theta_0(t)$ are described by a random walk:

$$\theta_0(t) = \theta_0(t-1) + w(t)$$

w(t) is Gaussian white noise with the following covariance matrix, or *drift matrix* R_1:

$$Ew(t)w^T(t) = R_1$$

R_2 is the variance of the innovations *e(t)* in the following equation:

$$y(t) = \psi^T(t)\theta_0(t) + e(t)$$

The Kalman filter algorithm is entirely specified by the sequence of data *y(t)*, the gradient $\psi(t)$, R_1, R_2, and the initial conditions $\theta(t=0)$ (initial guess of the parameters) and $P(t=0)$ (covariance matrix that indicates parameters errors).

Note It is assumed that R_1 and $P(t = 0)$ matrices are scaled such that $R_2 = 1$. This scaling does not affect the parameter estimates.

Normalized and Unnormalized Gradient

In the linear regression case, the gradient methods are also known as the *least mean squares* (LMS) methods.

The following set of equations summarizes the *unnormalized gradient* and *normalized gradient* adaptation algorithm:

$$\hat{\theta}(t) = \hat{\theta}(t-1) + K(t)\left(y(t) - \hat{y}(t)\right)$$

$$\hat{y}(t) = \psi^T(t)\hat{\theta}(t-1)$$

$$K(t) = Q(t)\psi(t)$$

In the unnormalized gradient approach, $Q(t)$ is given by:

$$Q(t) = \gamma$$

In the normalized gradient approach, $Q(t)$ is given by:

$$Q(t) = \frac{\gamma}{\left|\psi(t)\right|^2 + Bias}$$

The normalized gradient algorithm scales the adaptation gain, γ, at each step by the square of the two-norm of the gradient vector. If the gradient is close to zero, this can cause jumps in the estimated parameters. To prevent these jumps, a bias term is introduced in the scaling factor.

These choices of $Q(t)$ for the gradient algorithms update the parameters in the negative gradient direction, where the gradient is computed with respect to the parameters. See pg. 372 in [1] for details.

References

[1] Ljung, L. *System Identification: Theory for the User*. Upper Saddle River, NJ: Prentice-Hall PTR, 1999.

[2] Carlson, N.A. "Fast triangular formulation of the square root filter." *AIAA Journal*, Vol. 11, Number 9, 1973, pp. 1259-1265.

See Also

Recursive Least Squares Estimator | Recursive Polynomial Model Estimator | `recursiveAR` | `recursiveARMA` | `recursiveARMAX` | `recursiveARX` | `recursiveBJ` | `recursiveLS` | `recursiveOE`

More About

- “What Is Online Estimation?” on page 16-2

Perform Online Parameter Estimation at the Command Line

This topic shows how to perform online parameter estimation at the command line. The online estimation commands create a System object on page 16-26 for your model structure.

In this section...
"Online Estimation System Object" on page 16-26 "Workflow for Online Parameter Estimation at the Command Line" on page 16-27

Online Estimation System Object

A System object is a specialized MATLAB object designed specifically for implementing and simulating dynamic systems with inputs that change over time. System objects use internal states to store past behavior, which is used in the next computational step.

After you create a System object, you use commands to process data or obtain information from or about the object. System objects use a minimum of two commands to process data — a constructor to create the object and the `step` command to update object parameters using real-time data. This separation of declaration from execution lets you create multiple, persistent, reusable objects, each with different settings.

You can use the following commands with the online estimation System objects in System Identification Toolbox:

Command	Description
`step`	Update model parameter estimates using recursive estimation algorithms and real-time data. `step` puts the object into a locked state. In a locked state, you cannot change any nontunable properties or input specifications, such as model order, data type, or estimation algorithm. During execution, you can only change tunable properties.

Command	Description
`release`	Unlock the System object. Use this command to enable setting of nontunable parameters.
`reset`	Reset the internal states of a locked System object to the initial values, and leave the object locked.
`clone`	Create another System object with the same object property values. Do not create additional objects using syntax `obj2 = obj`. Any changes made to the properties of the new object created this way (`obj2`) also change the properties of the original object (`obj`).
`isLocked`	Query locked status for input attributes and nontunable properties of the System object.

Note If all data necessary for estimation is available at once, and you are estimating a time-invariant model, use the offline estimation commands for model parameter estimation. For example, use `arx` instead of `recursiveARX`.

Workflow for Online Parameter Estimation at the Command Line

1 Choose a model structure for your application.

Ideally, you want the simplest model structure that adequately captures the system dynamics. For considerations to keep in mind, see “Model Structure” on page 16-14.

2 Create an online estimation System object for your model structure by using one of the following commands:

- `recursiveAR` — Time-series AR model
- `recursiveARMA` — Time-series ARMA model
- `recursiveARX` — SISO or MISO ARX model

- `recursiveARMAX` — SISO ARMAX model
- `recursiveOE` — SISO output-error polynomial model
- `recursiveBJ` — SISO Box-Jenkins polynomial model
- `recursiveLS` — Single-output system that is linear in estimated parameters

```
obj = recursiveARX;
```

You can specify additional object properties such as the recursive estimation algorithm and initial parameter guesses. For information about the algorithms used, see “Recursive Algorithms for Online Parameter Estimation” on page 16-20.

3 Acquire input-output data in real time.

Specify estimation output data, `y`, as a real scalar, and input data, `u`, as a real scalar or vector. Data specified as an `iddata` object is not supported for online estimation.

4 Preprocess the estimation data.

Estimation data that contains deficiencies can lead to poor estimation results. Data deficiencies include drift, offset, missing samples, equilibrium behavior, seasonalities, and outliers. Preprocess the estimation data as needed. For considerations to keep in mind, see “Estimation Data” on page 16-15.

For online parameter estimation at the command line, you cannot use preprocessing tools in System Identification Toolbox. These tools support only data specified as `iddata` objects. Implement preprocessing code as required by your application. To be able to generate C and C++ code, use commands supported by MATLAB Coder. For a list of these commands, see “Functions and Objects Supported for C/C++ Code Generation — Category List” (MATLAB Coder).

5 Update the parameters of the model using incoming input-output data.

Use the `step` command to execute the specified recursive algorithm over each measurement of input-output data.

```
[A,B,yhat] = step(obj,y,u);
```

The output of the `step` command gives the estimated parameters (`A` and `B`), and estimated model output (`yhat`), at each set of input-output data.

Calling `step` on an object puts that object into a locked state. You can check the locked status of a System object using `isLocked`. When the object is locked, you

cannot change any nontunable properties or input specifications such as model order, data type, or estimation algorithm. To change a nontunable property, use the `release` command to unlock the System object. You can use `release` on a System object in code generated from MATLAB, but once you release its resources, you cannot use that System object again.

6 Post-process estimated parameters.

If necessary, you can post-process the estimated parameters. For instance, you can use a low-pass filter to smooth out noisy parameter estimates. To be able to generate C and C++ code, use commands supported by MATLAB Coder. For a list of these commands, see "Functions and Objects Supported for C/C++ Code Generation — Category List" (MATLAB Coder).

7 Validate the online estimation.

For details about the validation, see "Validate Online Parameter Estimation at the Command Line" on page 16-11. If you are not satisfied with the estimation, use the `reset` command to set the parameters of the System object to their initial value.

8 Use the estimated parameters for your application.

After validating the online parameter estimation, you can use MATLAB Compiler or MATLAB Coder to deploy the code in your application.

See Also

`clone` | `isLocked` | `recursiveAR` | `recursiveARMA` | `recursiveARMAX` | `recursiveARX` | `recursiveBJ` | `recursiveLS` | `recursiveOE` | `release` | `reset` | `step`

Related Examples

- "Validate Online Parameter Estimation at the Command Line" on page 16-11
- "Generate Code for Online Parameter Estimation in MATLAB" on page 16-31
- "Line Fitting with Online Recursive Least Squares Estimation" on page 16-57
- "Online ARX Parameter Estimation for Tracking Time-Varying System Dynamics" on page 16-66

More About

- “What Is Online Estimation?” on page 16-2
- “How Online Parameter Estimation Differs from Offline Estimation” on page 16-6
- “Recursive Algorithms for Online Parameter Estimation” on page 16-20

Generate Code for Online Parameter Estimation in MATLAB

In this section...
“Supported Online Estimation Commands” on page 16-31 “Generate Code for Online Estimation” on page 16-32 “Rules and Limitations When Using System Objects in Generated MATLAB Code” on page 16-33

This topic shows how to generate C/C++ code from online estimation MATLAB code that uses a System object. C/C++ code is generated using the `codegen` command from MATLAB Coder. 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. MATLAB Compiler software supports System objects for use inside MATLAB functions, but does not support System objects for use in MATLAB scripts.

For Simulink based workflows, use the online estimator blocks from System Identification Toolbox, such as Recursive Least Squares Estimator and Recursive Polynomial Model Estimator. You can generate C/C++ code and Structured Text for the online estimation blocks using Simulink Coder and Simulink PLC Coder.

Supported Online Estimation Commands

Code generation support is available for these online estimation System objects:

- `recursiveAR`
- `recursiveARMA`
- `recursiveARX`
- `recursiveARMAX`
- `recursiveOE`
- `recursiveBJ`
- `recursiveLS`

Code generation support is available only for the following System object commands:

- `step`

- `reset`
- `release`
- `isLocked`

Generate Code for Online Estimation

To generate code for online estimation:

1. Create a function to declare your System object as persistent, and initialize the object. You define the System object as persistent to maintain the object states between calls.

```
function [A,B,EstimatedOutput] = arxonline(output,input)
% Declare System object as persistent
persistent obj;
if isempty(obj)
    obj = recursiveARX([1 2 2],'EstimationMethod','Gradient');
end
[A,B,EstimatedOutput] = step(obj,output,input);
end
```

   The function creates a System object for online estimation of an ARX model of order `[1 2 1]`, using the unnormalized gradient algorithm, and estimation data, `input` and `output`. Save this function on the MATLAB path. Alternatively, you can specify the full path name for this function.

   The persistent System 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 just execute the `step` command to update the estimated parameters. During initialization you specify the nontunable properties of the object, such as `EstimationMethod`, `Orders`, and `DataType`.

2. Generate C/C++ code and MEX-files using the `codegen` command from MATLAB Coder.

```
codegen arxonline -args {1,1}
```

   The syntax `-args {1,1}` specifies a set of example arguments to your function. The example arguments set the dimensions and data types of the function arguments `output`and `input` as double-precision scalars.

3. Use the generated code.

- Use the generated C/C++ code to deploy online model 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 parameter estimation algorithms in MATLAB.

  Load the estimation data. In this example, use a static data set for illustration.

  ```
  load iddata3
  output = z3.y;
  input = z3.u;
  ```

  Update the model parameters by calling the generated MEX-file.

  ```
  for i = 1:numel(input)
      [A,B,EstimatedOutput] = arxonline_mex(output(i),input(i));
  end
  ```

Rules and Limitations When Using System Objects in Generated MATLAB Code

The following rules and limitations apply to using online estimation System objects when writing MATLAB code suitable for code generation.

Object Construction and Initialization

- If System objects are stored in persistent variables, initialize objects once by embedding the object handles in an `if` statement with a call to `isempty( )`.
- Set arguments to System object constructors as compile-time constants when using the `codegen` command. For more information, see `coder.Constant`.
- Do not initialize System objects properties with other MATLAB class objects as default values in code generation. Initialize these properties in the constructor.

Inputs and Outputs

- The data type of the System object inputs should not change.
- Do not pass a System object as an example input argument to a function being compiled with `codegen`.
- Do not pass a System object to functions declared as extrinsic (functions called in interpreted mode) using the `coder.extrinsic` function. System objects returned

from extrinsic functions and scope System objects that automatically become extrinsic can be used as inputs to another extrinsic function, but they do not generate code.

Cell Arrays

- Cell arrays cannot contain System objects.

Tunable and Nontunable Properties of System Objects

- The value assigned to a nontunable property must be a constant, and there can be at most one assignment to that property (including the assignment in the constructor).
- You can set the tunable properties of online estimation System objects at construction time or by using dot notation after that.

See Also

`codegen` | `isLocked` | `recursiveAR` | `recursiveARMA` | `recursiveARMAX` | `recursiveARX` | `recursiveBJ` | `recursiveLS` | `recursiveOE` | `release` | `reset` | `step`

Related Examples

- “Perform Online Parameter Estimation at the Command Line” on page 16-26
- “Validate Online Parameter Estimation at the Command Line” on page 16-11
- “Line Fitting with Online Recursive Least Squares Estimation” on page 16-57
- “Online ARX Parameter Estimation for Tracking Time-Varying System Dynamics” on page 16-66

More About

- “What Is Online Estimation?” on page 16-2

Extended and Unscented Kalman Filter Algorithms for Online State Estimation

In this section...
“Extended Kalman Filter Algorithm” on page 16-35 “Unscented Kalman Filter Algorithm” on page 16-38

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 nonadditive. 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 nonadditive. 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 nonadditive process and measurement noise terms with zero mean and covariances 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 u_s and 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, P.

$$\hat{x}[0|-1] = E(x[0])$$

$$P[0|-1] = E(x[0] - \hat{x}[0|-1])(x[0] - \hat{x}[0|-1])^T$$

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 $0, 1, \ldots, k_b$. 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 Jacobians 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.

$$C[k] = \left.\frac{\partial h}{\partial x}\right|_{x[k|k-1]}$$

$$S[k] = \left.\frac{\partial h}{\partial v}\right|_{x[k|k-1]}$$

The software calculates these Jacobian matrices numerically unless you specify the analytical Jacobian.

$$K[k] = P[k \mid k-1]C[k]^T(C[k]P[k \mid k-1]C[k]^T + S[k]R[k]S[k]^T)^{-1}$$
$$x[k \mid k] = x[k \mid k-1] + K[k](y[k] - h(x[k \mid k-1], 0, u_m[k])$$
$$P[k \mid k] = P[k \mid k\text{-}1] - K[k]\,C[k]\,P[k \mid k\text{-}1]$$

Here `K` is the Kalman gain.

b Compute the Jacobians 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.

$$A[k] = \left.\frac{\partial f}{\partial x}\right|_{x[k|k]}$$
$$G[k] = \left.\frac{\partial f}{\partial w}\right|_{x[k|k]}$$

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.

$$P[k+1 \mid k] = A[k]P[k \mid k]A[k]^T + G[k]Q[k]G[k]^T$$
$$x[k+1 \mid k] = f(x[k \mid k], 0, u_s[k])$$

These values are used by the `correct` command in the next time step.

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 nonadditive 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.

$$\hat{x}[0 \mid -1] = E(x[0])$$
$$P[0 \mid -1] = E(x[0] - \hat{x}[0 \mid -1])(x[0] - \hat{x}[0 \mid -1])^T$$

Here $\hat{x}$ is the state estimate and $\hat{x}[k_a \mid k_b]$ denotes the state estimate at time step `k_a` using measurements at time steps `0,1,...,k_b`. So $\hat{x}[0 \mid -1]$ 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.

$$\hat{x}^{(0)}[k \mid k-1] = \hat{x}[k \mid k-1]$$

$$\hat{x}^{(i)}[k \mid k-1] = \hat{x}[k \mid k-1] + \Delta x^{(i)} \qquad i = 1,\ldots,2M$$

$$\Delta x^{(i)} = \left(\sqrt{cP[k \mid k-1]}\right)_i \quad i = 1,\ldots,M$$

$$\Delta x^{(M+i)} = -\left(\sqrt{cP[k \mid k-1]}\right)_i \quad i = 1,\ldots,M$$

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 16-42. $\sqrt{cP}$ is the matrix square root of cP such that $\sqrt{cP}\left(\sqrt{cP}\right)^T = cP$ and $\left(\sqrt{cP}\right)_i$ 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.

$$P_y = \sum_{i=0}^{2M} W_c^{(i)} (y^{(i)}[k \mid k-1] - y[k])(y^{(i)}[k \mid k-1] - y[k])^T + R[k]$$

$$W_c^{(0)} = (2 - \alpha^2 + \beta) - \frac{M}{\alpha^2 (M + \kappa)}$$

$$W_c^i = 1/(2\alpha^2 (M + \kappa)) \qquad i = 1, 2, \ldots, 2M$$

For information about β parameter, see "Effect of Alpha, Beta, and Kappa Parameters" on page 16-42.

e Estimate the cross-covariance between $\hat{x}[k \mid k-1]$ and $\hat{y}[k]$.

$$P_{xy} = \frac{1}{2\alpha^2 (m + \kappa)} \sum_{i=1}^{2M} (x^{(i)}[k \mid k-1] - x[k \mid k-1])(y^{(i)}[k \mid k-1] - y[k])^T$$

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.

$$K = P_{xy} P_y^{-1}$$

$$\hat{x}[k \mid k] = \hat{x}[k \mid k-1] + K(y[k] - \hat{y}[k])$$

$$P[k \mid k] = P[k \mid k-1] - K P_y K_k^T$$

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 $\hat{x}^{(i)}[k \mid k]$ at time step k.

$$\hat{x}^{(0)}[k \mid k] = \hat{x}[k \mid k]$$

$$\hat{x}^{(i)}[k \mid k] = \hat{x}[k \mid k] + \Delta x^{(i)} \qquad i = 1, \ldots, 2M$$

$$\Delta x^{(i)} = \left(\sqrt{cP[k \mid k]}\right)_i \quad i = 1, \ldots, M$$

$$\Delta x^{(M+i)} = -\left(\sqrt{cP[k \mid k]}\right)_i \quad i = 1, \ldots, M$$

b Use the nonlinear state transition function to compute the predicted states for each of the sigma points.

$$\hat{x}^{(i)}[k+1 \mid k] = f(\hat{x}^{(i)}[k \mid k], u_s[\mathrm{k}])$$

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.

$$\hat{x}[k+1 \mid k] = \sum_{i=0}^{2M} W_M^{(i)} \hat{x}^{(i)}[k+1 \mid k]$$

$$W_M^{(0)} = 1 - \frac{M}{\alpha^2(M+\kappa)}$$

$$W_M^{i} = \frac{1}{2\alpha^2(M+\kappa)} \qquad i = 1, 2, \ldots, 2M$$

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.

$$P[k+1 \mid k] = \sum_{i=0}^{2M} W_c^{(i)} (x^{(i)}[k+1 \mid k] - x[k+1 \mid k])(x^{(i)}[k+1 \mid k] - x[k+1 \mid k])^T + Q[k]$$

$$W_c^{(0)} = (2 - \alpha^2 + \beta) - \frac{M}{\alpha^2(M+\kappa)}$$

$$W_c^{i} = 1/(2\alpha^2(M+\kappa)) \quad i = 1, 2, \ldots, 2M$$

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. If the noise terms are nonadditive, 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, D. *Optimal State Estimation: Kalman, H Infinity, and Nonlinear Approaches*. John Wiley and Sons Inc., 2006.

See Also

Functions

`extendedKalmanFilter` | `unscentedKalmanFilter`

Blocks

Extended Kalman Filter | Unscented Kalman Filter

Validate Online State Estimation at the Command Line

In this section...
“Examine Output Estimation Error” on page 16-44 “Examine State Estimation Error for Simulated Data” on page 16-45

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 16-55 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 16-124.

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`.

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 `xcorr` command from Signal Processing 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 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` | `particleFilter` | `unscentedKalmanFilter`

More About

- “Nonlinear State Estimation Using Unscented Kalman Filter and Particle Filter” on page 16-124
- “Extended and Unscented Kalman Filter Algorithms for Online State Estimation” on page 16-35
- “Troubleshoot Online State Estimation” on page 16-55
- “Generate Code for Online State Estimation in MATLAB” on page 16-51

Validate Online State Estimation in Simulink

After you use the Extended Kalman Filter or Unscented Kalman 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 16-55 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 16-48.

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

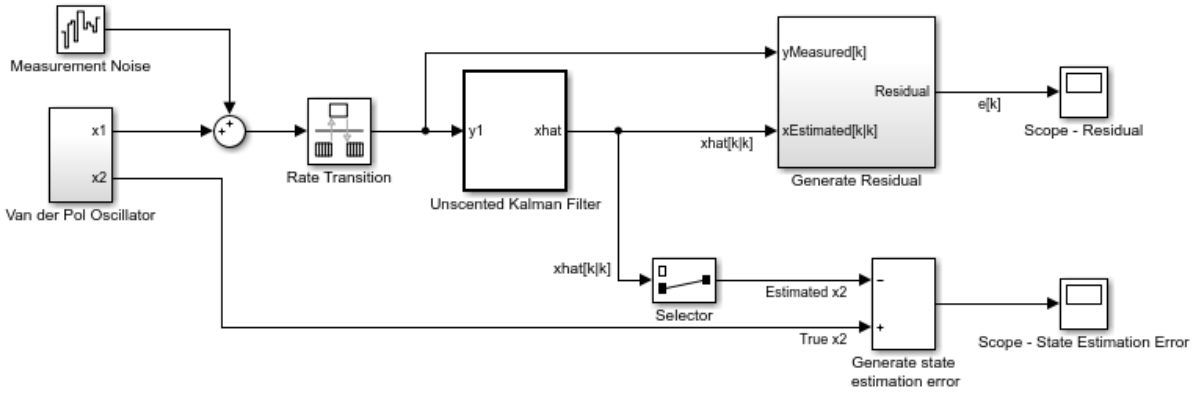
You can also compute the covariance of the state estimation error, and compare it to the state estimation error covariance that is output by the blocks in the **P** port of the blocks. Similar values increase confidence in the performance of the filter.

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.

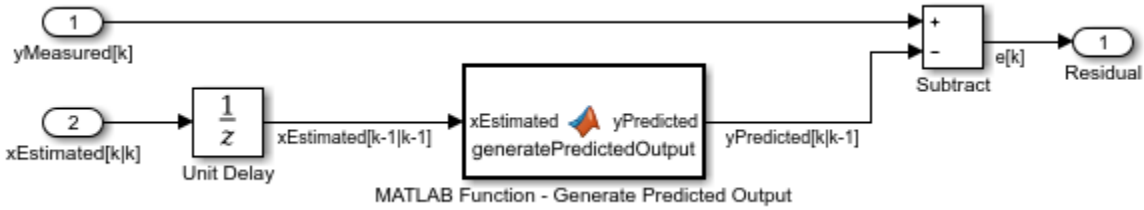
In the Simulink™ model, the Van der Pol Oscillator block implements the oscillator with nonlinearity parameter, mu, equal to 1. The oscillator has two states. A noisy measurement of the first state `x1` is available.

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 16-124.



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

Extended Kalman Filter | Kalman Filter | Unscented Kalman Filter

More About

- "Extended and Unscented Kalman Filter Algorithms for Online State Estimation" on page 16-35
- "Troubleshoot Online State Estimation" on page 16-55

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 16-53.

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 — Category List" (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`
Nontunable properties that you must specify during object construction	`HasAdditiveProcessNoise` and `HasAdditiveMeasurementNoise`	`HasAdditiveProcessNoise` and `HasAdditiveMeasurementNoise`	

See Also

`extendedKalmanFilter` | `particleFilter` | `unscentedKalmanFilter`

More About

- “What Is Online Estimation?” on page 16-2
- “Nonlinear State Estimation Using Unscented Kalman Filter and Particle Filter” on page 16-124
- “Validate Online State Estimation at the Command Line” on page 16-44
- “Troubleshoot Online State Estimation” on page 16-55

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 16-124.
- 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` | `particleFilter` | `unscentedKalmanFilter`

Blocks

Extended Kalman Filter | Unscented Kalman Filter

More About

- “Validate Online State Estimation at the Command Line” on page 16-44
- “Validate Online State Estimation in Simulink” on page 16-47
- “Generate Code for Online State Estimation in MATLAB” on page 16-51

Line Fitting with Online Recursive Least Squares Estimation

This example shows how to perform online parameter estimation for line-fitting using recursive estimation algorithms at the MATLAB command line. You capture the time-varying input-output behavior of the hydraulic valve of a continuously variable transmission.

Physical System

The system is a continuously variable transmission (CVT) driven by a hydraulic valve, inspired by reference [1]. The valve pressure is connected to the CVT which allows it to change its speed ratio and to transmit torque from the engine to the wheels. The input-output behavior of the valve can be approximated by:

$$y(t) = k(t)u(t) + b(t) \quad \text{for} \quad \frac{-b(t)}{k(t)} < u \leq 1$$

Here, t is the current time, y(t) is the valve pressure in bar, u(t) is the unitless input in the range of [0, 1]. The condition $\frac{-b}{k} < u$ is the dead-band of the valve.

The slope, k(t), and offset, b(t), depend on the system temperature. They vary as the system warms up from cold start to typical operating temperature. You want to estimate k(t) and b(t) based on noisy measurements of u(t) and y(t).

Data

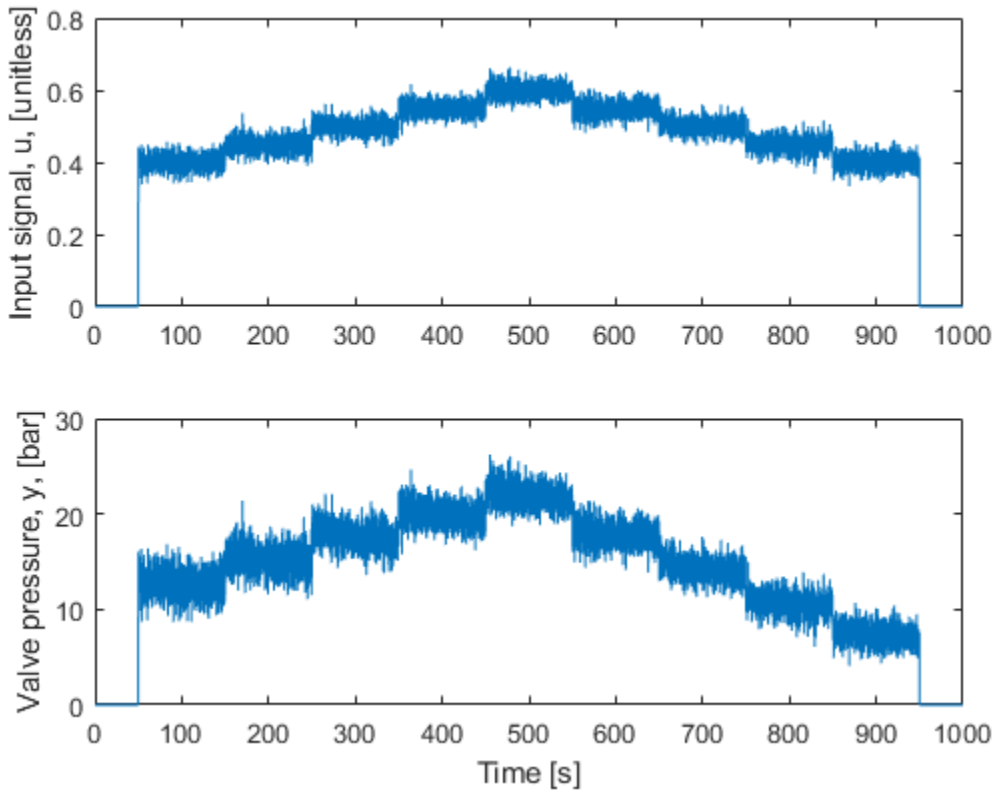
The true slope and offset parameters are k(0)=70 and b(0)=-15 at time t=0s. At t=50s the engine starts. The parameters vary over time until they reach k(950)=50 and b(950)=-13 at t=950s. The sample time is Ts=0.1s.

The content of the input signal u is critical for parameter estimation. Consider a case where u, and hence y, is constant. Then there are infinitely many k and b values that satisfy y = k u + b. u(t) must be persistently exciting the system for successful estimation of k(t) and b(t). In this example, the input u:

- is zero from t=0s until t=50s.
- has step changes to 0.40, 0.45, 0.50, 0.55, 0.60, 0.55, 0.50, 0.45, 0.40 every 100s, from t=50s until t=950s.
- a Gaussian random variable with zero mean, 0.02 standard deviation was added at each time step from t=50s until t=950s to provide extra excitation of system dynamics for identification purposes.

The output is generated with the aforementioned true values of k(t), b(t) along with the input signal u(t), using y(t) = k(t) u(t) + b(t) + e(t). e(t), measurement noise, is a Gaussian random variable with zero mean and standard deviation 0.05.

```
load LineFittingRLSExample u y k b t;
figure();
subplot(2,1,1);
plot(t,u);
ylabel('Input signal, u, [unitless]');
subplot(2,1,2);
plot(t,y);
ylabel('Valve pressure, y, [bar]');
xlabel('Time [s]');
```



Online Parameter Estimation Using Recursive Least Squares

Write the valve input-output model using vector notation:

$$\begin{aligned} y(t) &= k(t)u(t) + b(t) + e(t) \\ &= [u(t)\ 1][k(t)\ b(t)]^T + e(t) \\ &= H(t)x(t) + e(t) \end{aligned}$$

where $H(t) = [u(t)\ 1]$ is the regressors and $x = [k(t)\ b(t)]^T$ is the parameters to be estimated. e(t) is the unknown noise. You use the recursievLS estimation command to create a System object for online parameter estimation. You then use the step command to update the parameter estimates, x(t), at each time-step based on H(t) and y(t).

You specify the following recursiveLS System Object properties:

- **Number of parameters:** 2.
- **EstimationMethod:** 'ForgettingFactor' (default). This method has only one scalar parameter, ForgettingFactor, which requires limited prior information regarding parameter values.
- **ForgettingFactor:** 0.95. The parameters are expected to vary over time, hence less than 1. $\frac{1}{1-\lambda} = 20$ is the number of past data samples that influence the estimates most.
- **InitialParameters:** [70; -15], an initial guess for the parameter values. Optional, but recommended for reducing initial transients.
- **InitialParameterCovariance:** Your estimate of uncertainty in the initial parameter guess. Set it to a small value, 1% of the absolute value of the initial parameters, if you have confidence in the initial parameter guesses. Optional but recommended, especially when you specify InitialParameters. This is only utilized with the ForgettingFactor and KalmanFilter estimation methods.

```
X = recursiveLS(2,... % 2=number of estimated parameters
    'EstimationMethod','ForgettingFactor',...
    'ForgettingFactor',0.95,...
    'InitialParameters',[70; -15],...
    'InitialParameterCovariance',[0.7 0.15]);
```

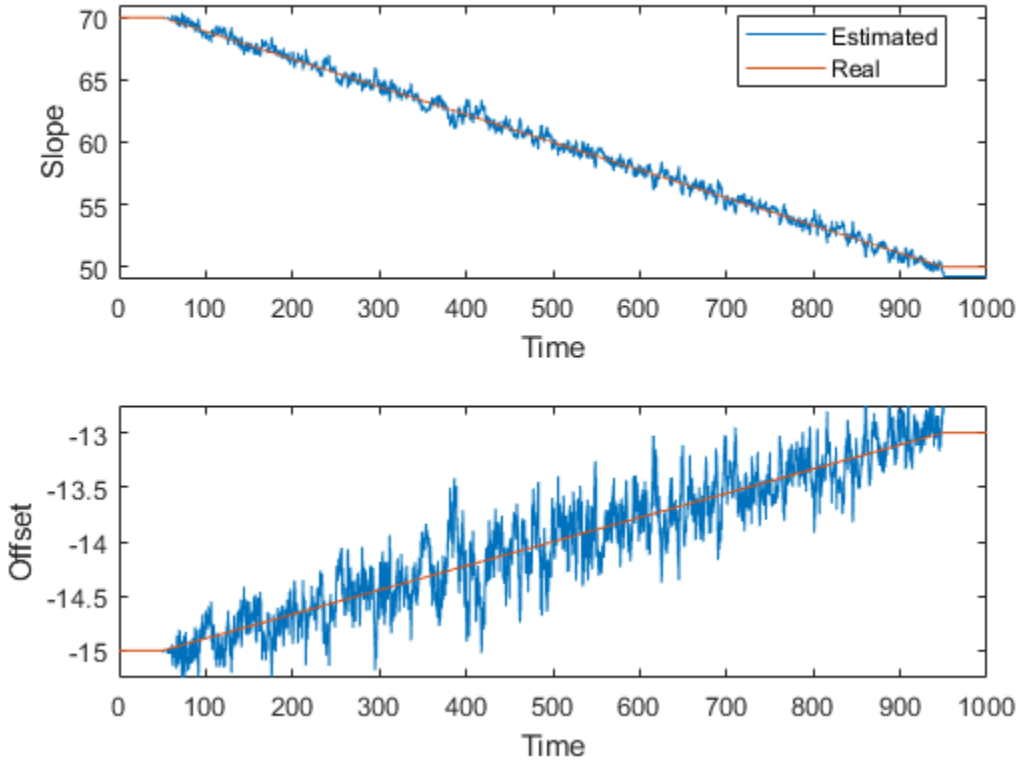
This example simulates the online operation of the estimator by providing one (y(t),H(t)) pair to the estimator at a time. Call the step command to update parameters with each new pair. The parameter adaptation is enabled only when the input u is outside the dead band (u>0.3).

```
theta = zeros(numel(u),2);
yHat = zeros(numel(u),1);
PHat = zeros(numel(u),2,2);
for kk=1:numel(u)
    % enable parameter estimation only when u is outside the dead-band
    if u(kk)>=0.3
        X.EnableAdaptation = true();
    else
        X.EnableAdaptation = false();
    end
    [theta(kk,:),yHat(kk)] = step(X,y(kk),[u(kk) 1]); % get estimated parameters and ou
    PHat(kk,:,:) = X.ParameterCovariance; % get estimated uncertainty in parameters
    % perform any desired tasks with the parameters
end
```

Estimated Parameters

The true and estimated parameter values are:

```
figure();
subplot(2,1,1);
plot(t,theta(:,1),t,k); % Estimated and real slope, respectively
ylabel('Slope');
xlabel('Time');
ylim([49 71]);
legend('Estimated','Real','Location','Best');
subplot(2,1,2);
plot(t,theta(:,2),t,b); % Estimated and real offset, respectively
ylabel('Offset');
xlabel('Time');
ylim([-15.25 -12.75]);
```



Validating the Estimated Model

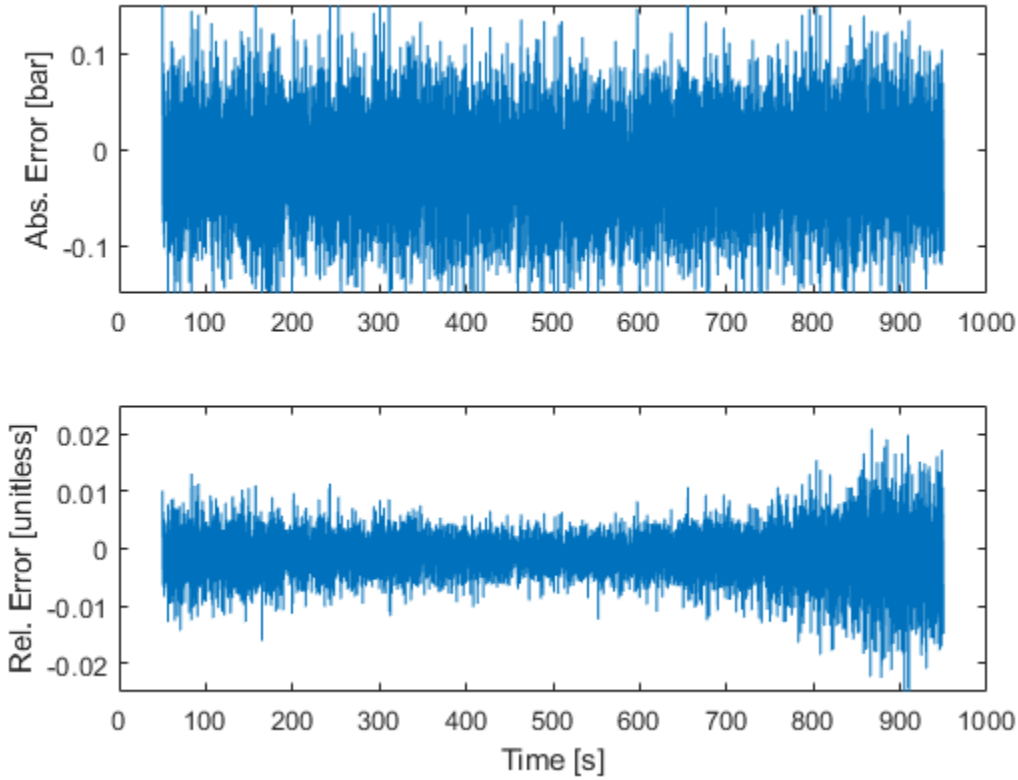
The estimator provides the following two tools to judge the quality of the parameter estimates:

1. **Output estimate $\hat{y}(t)$:** The second output argument of the step method is $\hat{y}(t) = H(t)\hat{x}(t)$. The relative and absolute error between y and $\hat{y}$ are measures of the goodness of the fit.

2. **Parameter covariance estimate $\hat{P}(t)$:** This is available with the ForgettingFactor and KalmanFilter algorithms. It is stored in the ParameterCovarianceMatrix property of the estimator. The diagonals of $\hat{P}$ are the estimated variances of the parameters. The lower the better.

The output measurement and its estimate, as well as the associated absolute and relative errors when the engine is on are:

```
engineOn = t>50 & t<950;
figure();
subplot(2,1,1);
absoluteError = y-yHat;
plot(t(engineOn),absoluteError(engineOn));
ylim([-0.15 0.15]);
ylabel('Abs. Error [bar]');
subplot(2,1,2);
relativeError = (y-yHat)./y;
plot(t(engineOn),relativeError(engineOn));
ylim([-0.025 0.025]);
ylabel('Rel. Error [unitless]');
xlabel('Time [s]');
```

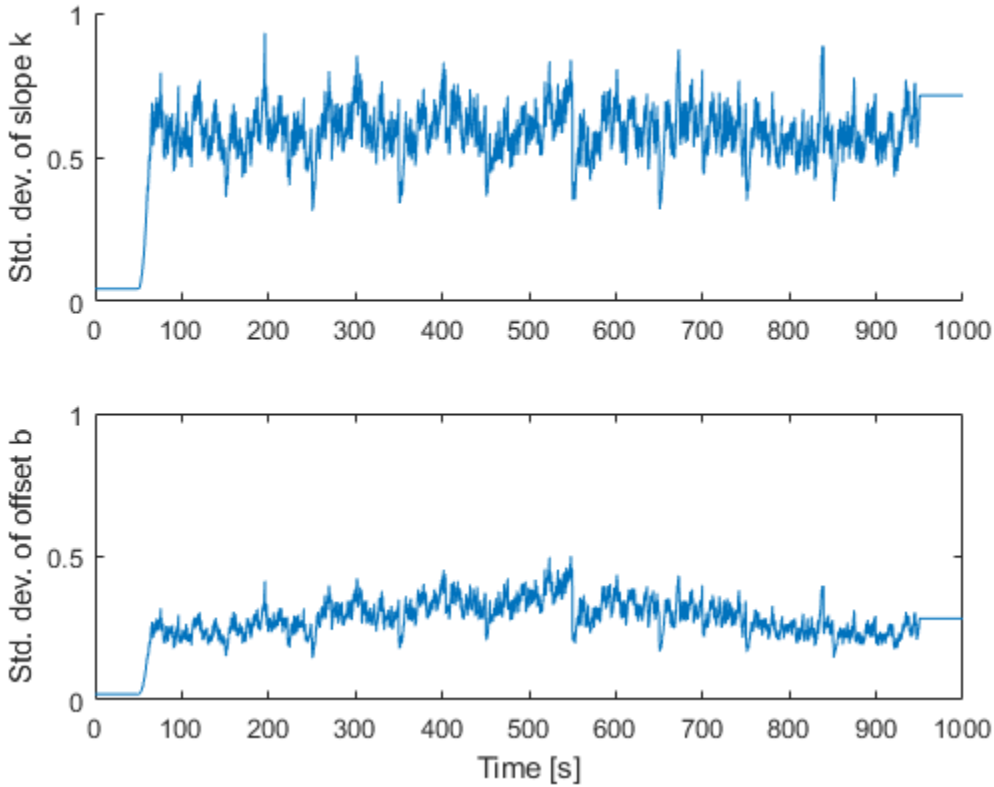


The absolute errors are about 0.1bar. The relative errors are below 2%. Both quantities are small.

The diagonals of the parameter covariance matrix, scaled by the variance of the residuals $y(t) - \hat{y}(t)$, capture the variances of parameter estimates. The square-root of the variances are the standard deviations of the parameter estimates.

```
noiseVariance = var(y(engineOn)-yHat(engineOn));
figure();
subplot(2,1,1);
hold on;
plot(t,sqrt(PHat(:,1,1)*noiseVariance));
ylim([0 1]);
ylabel('Std. dev. of slope k');
```

```
subplot(2,1,2);
plot(t,sqrt(PHat(:,2,2)*noiseVariance));
ylim([0 1]);
ylabel('Std. dev. of offset b');
xlabel('Time [s]');
hold on;
```



The standard deviation of the slope k fluctuates around 0.7. This is small relative to the range of values of k [50, 70]. This gives confidence in the parameter estimates. The situation is similar with the offset b, which is in the range [-15 -13].

Note that the parameter standard deviations are also estimates. They are based on the assumption that the residuals $y(t) - \hat{y}(t)$ are white. This depends on the estimation

method, its associated parameters, structure of the estimated model, and the input signal u. Differences between the assumed and the actual model structure, lack of persistent input excitation, or unrealistic estimation method settings can lead to overly optimistic or pessimistic uncertainty estimates.

Summary

You performed a line fit using recursive least squares to capture the time-varying input-output behavior of a hydraulic valve. You evaluated the quality of fit by looking at two signals: the error between estimated and measured system output, and the parameter covariance estimates.

References

[1] Gauthier, Jean-Philippe, and Philippe Micheau. "Regularized RLS and DHOBE: An Adaptive Feedforward for a Solenoid Valve." Control Systems Technology, IEEE Transactions on 20.5 (2012): 1311-1318

See Also

`clone` | `isLocked` | `recursiveAR` | `recursiveARMA` | `recursiveARMAX` | `recursiveARX` | `recursiveBJ` | `recursiveLS` | `recursiveOE` | `release` | `reset` | `step`

Related Examples

- “Perform Online Parameter Estimation at the Command Line” on page 16-26
- “Validate Online Parameter Estimation at the Command Line” on page 16-11
- “Generate Code for Online Parameter Estimation in MATLAB” on page 16-31
- “Online ARX Parameter Estimation for Tracking Time-Varying System Dynamics” on page 16-66

Online ARX Parameter Estimation for Tracking Time-Varying System Dynamics

This example shows how to perform online parameter estimation for a time-varying ARX model at the MATLAB command line. The model parameters are updated at each time step with incoming new data. This model captures the time-varying dynamics of a linear plant.

Plant

The plant can be represented as:
$y(s) = G(s)u(s) + e(s)$

Here, G is the transfer function and e is the white-noise. The plant has two operating modes. In the first operating mode, the transfer function is:

$$G1(s) = \frac{4500}{(s+5)(s^2+1.2s+900)}$$

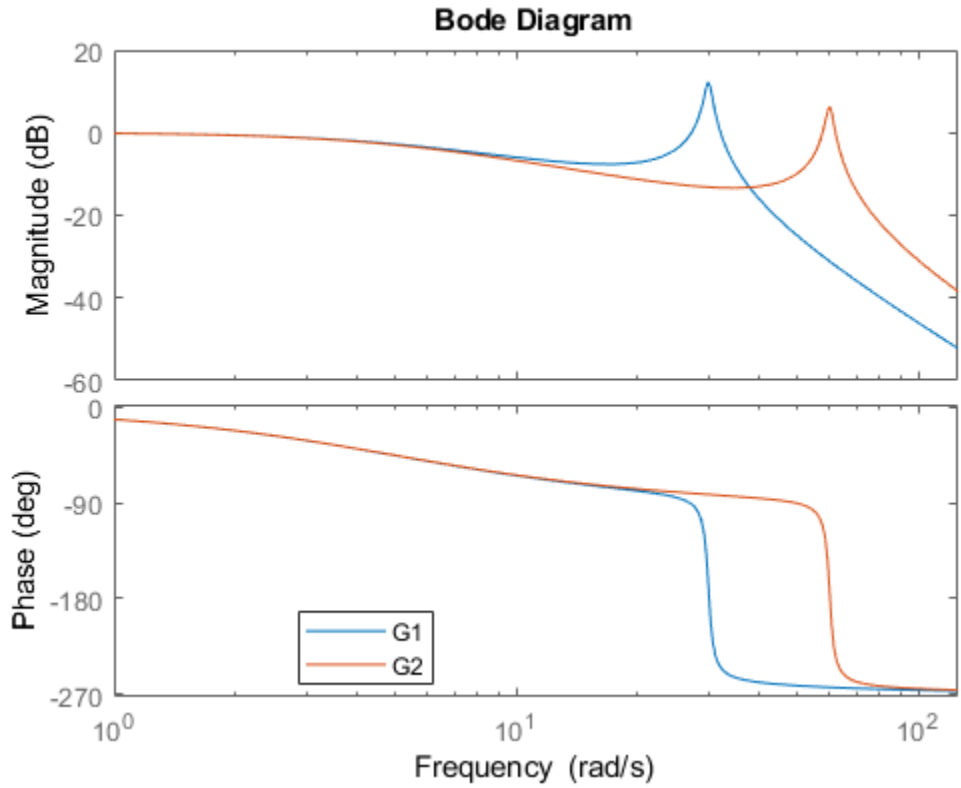
The lightly damped poles in G1(s) have the damping 0.02 and natural frequency 30rad/s. In the second operating mode, the natural frequency of these poles is 60rad/s. In this mode, the transfer function is:

$$G2(s) = \frac{18000}{(s+5)(s^2+2.4s+3600)}$$

The plant operates in the first mode until t=10s, and then switches to the second mode.

The Bode plots of G1 and G2 are:

```
wn = 30; % natural frequency of the lightly damped poles
ksi = 0.02; % damping of the poles
G1 = tf(1,conv([1/5 1],[1/wn^2 2*ksi/wn 1])); % plant in mode 1
wn = 60; % natural frequency in the second operating mode
G2 = tf(1,conv([1/5 1],[1/wn^2 2*ksi/wn 1])); % plant in mode 2
bode(G1,G2,{1,125});
legend('G1','G2','Location','Best');
```



Online ARX Parameter Estimation

The aim is to estimate the dynamics of the plant during its operation. ARX is a common model structure used for this purpose. ARX models have the form:

$$y(t) = \frac{B(q)}{A(q)}u(t) + \frac{1}{A(q)}e(t)$$

Here, q^{-1} is the time-shift operator. The ratio of the polynomials B(q)/A(q) captures the input-output model (u(t) to y(t)), and 1/A(q) captures the noise model (e(t) to y(t)). You are estimating the coefficients of the A(q) and B(q) polynomials. e(t) is white noise.

ARX model structure is a good first candidate for estimating linear models. The related estimation methods have low computational burden, are numerically robust, and have

the convexity property. The convexity property ensures there is no risk of parameter estimation getting stuck at a local optima. However, ARX model structure does not provide flexibility for noise models.

The lack of flexibility in noise modeling can pose difficulties if the structure of the plant does not match with the ARX model structure, or if the noise is not white. Two approaches to remedy this issue are:

1. Data Filtering: If the noise model is not important for your application, you can use data filtering techniques. For more details, see the 'Filter the Data' section.
2. Different model structures: Use ARMAX, Output-Error, and Box-Jenkins polynomial models for more flexibility in model structures.

Select Sample Time

The sample time choice is important for good approximation of the continuous-time plant by a discrete-time model. A rule of thumb is to choose the sampling frequency as 20 times the dominant dynamics of the system. The plant has the fastest dominant dynamics at 60rad/s, or about 10Hz. The sampling frequency is therefore 200Hz.

```
Ts = 0.005; % [s], Sample time, Ts=1/200
```

System Excitation

For successful estimation the plant inputs must persistently excite its dynamics. Simple inputs such as a single step input are typically not sufficient. In this example the plant is driven by a pulse with amplitude 10 and period 1 seconds. The pulse width is 50% of its period.

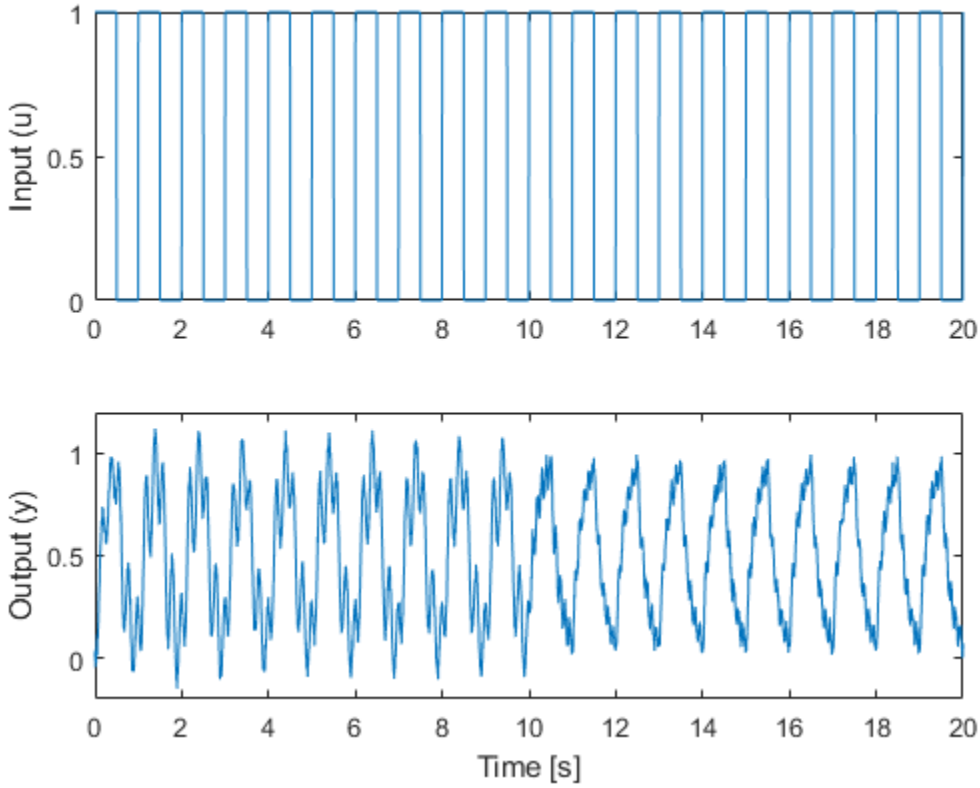
Generate plant input and output signals:

```
t = 0:Ts:20; % Time vector
u = double(rem(t/1,1)-0.5 < 0); % pulse
y = zeros(size(u));
% Store random number generator's states for reproducible results.
sRNG = rng;
rng('default');
% Simulate the mode-switching plant with a zero-order hold.
G1d = c2d(G1,Ts,'zoh');
B1 = G1d.num{1}.';
A1 = G1d.den{1}.'; % B1 and A1 corresponds to G1.
G2d = c2d(G2,Ts,'zoh');
B2 = G2d.num{1}.';
```

```
A2 = G2d.den{1}.'; % B2 and A2 corresponds to G2.
idx = numel(B2):-1:1;
for ct=(1+numel(B2)):numel(t)
    idx = idx + 1;
    if t(ct)<10 % switch mode after t=10s
        y(ct) = u(idx)*B1-y(idx(2:end))*A1(2:end);
    else
        y(ct) = u(idx)*B2-y(idx(2:end))*A2(2:end);
    end
end
% Add measurement noise
y = y + 0.02*randn(size(y));
% Restore the random number generator's states.
rng(sRNG);
```

Plot the input-output data:

```
figure();
subplot(2,1,1);
plot(t,u);
ylabel('Input (u)');
subplot(2,1,2);
plot(t,y);
ylim([-0.2 1.2])
ylabel('Output (y)');
xlabel('Time [s]');
```



Filter the Data

The plant has the form:
$y(t) = G(q)u(t) + e(t)$

where e(t) is the white noise. In contrast, the ARX models have the form

$$y(t) = \frac{B(q)}{A(q)}u(t) + \frac{1}{A(q)}e(t)$$

The estimator will use B(q) and A(q) to approximate G(q). However, note the difference in the noise models. The plant has white-noise e(t) directly impacting y(t), but the ARX model assumes that a white noise term filtered by 1/A(q) impacts y(t). This mismatch will negatively affect the estimation.

When the noise model is not of interest, one method to reduce the impact of this mismatch is to use a data filter. Use a filter $F(q)$ on both u(t) and y(t) to obtain $u_f(t) = F(q)u(t)$ and $y_f(t) = F(q)y(t)$. Then use the filtered signals $u_f(t)$ and $y_f(t)$ in the estimator instead of the plant input u(t) and output y(t). The choice of data filter lets you reduce the influence of e(t) on the estimation.

The data filter F(q) is typically a low-pass or a band-pass filter based on the frequency range of importance for the application, and the characteristics of e(t). Here, a 4th order Butterworth low-pass filter with cutoff frequency 10Hz is used. This is approximately the frequency of the fastest dominant dynamics in the plant (60rad/s). Low-pass filter is suffcient here because the noise term does not have low-frequency content.

```
% Filter coefficients
Fa = [1 -3.1806 3.8612 -2.1122 0.4383]; % denominator
Fb = [4.1660e-04 1.6664e-03 2.4996e-03 1.6664e-03 4.1660e-04]; % numerator
% Filter the plant input for estimation
uf = filter(Fb,Fa,u);
% Filter the plant output
yf = filter(Fb,Fa,y);
```

Set Up the Estimation

Use the recursiveARX command for online parameter estimation. The command creates a System object™ for online parametre estimation of an ARX mdoel structure. Specify the following properties of the object:

- **Model orders:** [3 1 0]. na = 3 because the plant has 3 poles. nk = 0 because plant does not have input delay. nb = 1 corresponds to no zeros in the system. nb was set after a few iterations, starting from nb=4 which corresponds to three zeros, and hence a proper model. Smaller number of estimated parameters are desirable and nb=1 yielded sufficient results.
- **EstimationMethod:** 'ForgettingFactor' (default). This method has only one scalar parameter, ForgettingFactor, which requires limited prior information regarding parameter values.
- **ForgettingFactor:** 0.995. The forgetting factor, λ, is less than one as the parameters vary over time. $\frac{1}{1-\lambda} = 200$ is the number of past data samples that influence the estimates most.

```
X = recursiveARX([3 1 0]); % [na nb nk]
X.ForgettingFactor = 0.995;
```

Create arrays to store estimation results. These are useful for validating the algorithms.

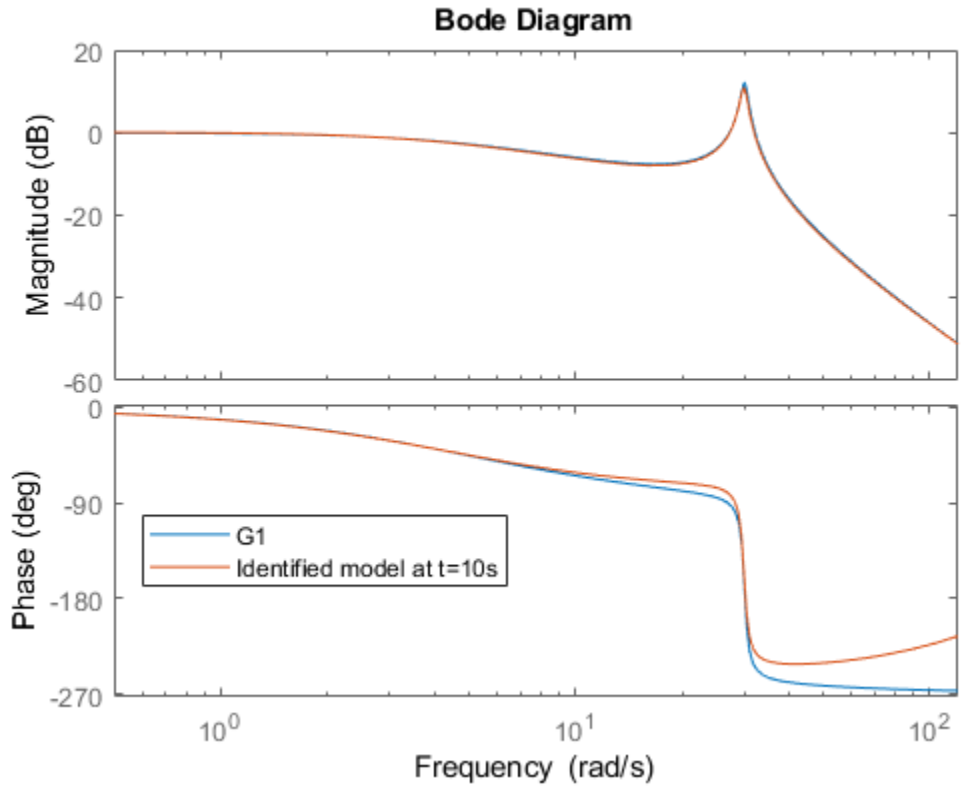
```
np = size(X.InitialParameterCovariance,1);
PHat = zeros(numel(u),np,np);
A = zeros(numel(u),numel(X.InitialA));
B = zeros(numel(u),numel(X.InitialB));
yHat = zeros(1,numel(u));
```

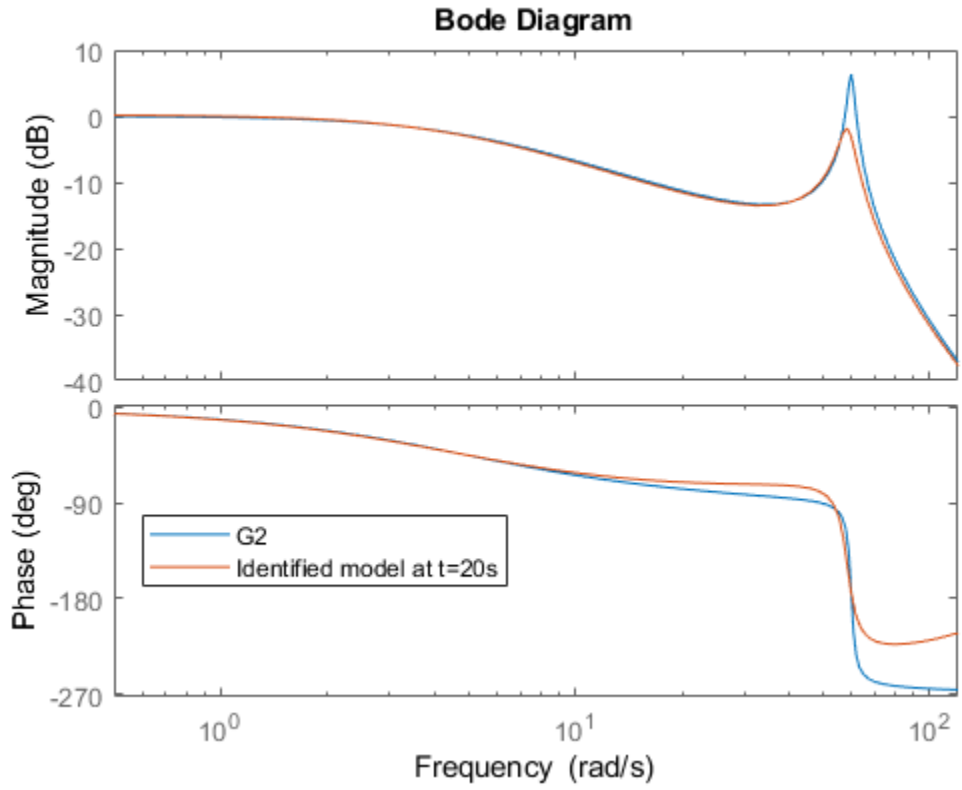
Use the step command to update the parameter values using one set of input-output data at each time step. This illustrates the online operation of the estimator.

```
for ct=1:numel(t)
    % Use the filtered output and input signals in the estimator
    [A(ct,:),B(ct,:),yHat(ct)] = step(X,yf(ct),uf(ct));
    PHat(ct,:,:) = X.ParameterCovariance;
end
```

View the Bode plot of the estimated transfer functions:

```
G1Hat = idpoly(A(1000,:),B(1000,:),1,1,1,[],Ts); % Model snapshot at t=10s
G2Hat = idpoly(X); % Snapshot of the latest model, at t=20s
G2Hat.Ts = G1d.Ts; % Set the sample time of the snapshot
figure();
bode(G1,G1Hat);
xlim([0.5 120]);
legend('G1','Identified model at t=10s','Location','Best');
figure();
bode(G2,G2Hat);
xlim([0.5 120]);
legend('G2','Identified model at t=20s','Location','Best');
```


Bode Diagram
Magnitude (dB)
20
0
-20
-40
-60
Phase (deg)
0
-90
-180
-270
G1
Identified model at t=10s
10^0
10^1
10^2
Frequency (rad/s)



Validating the Estimated Model

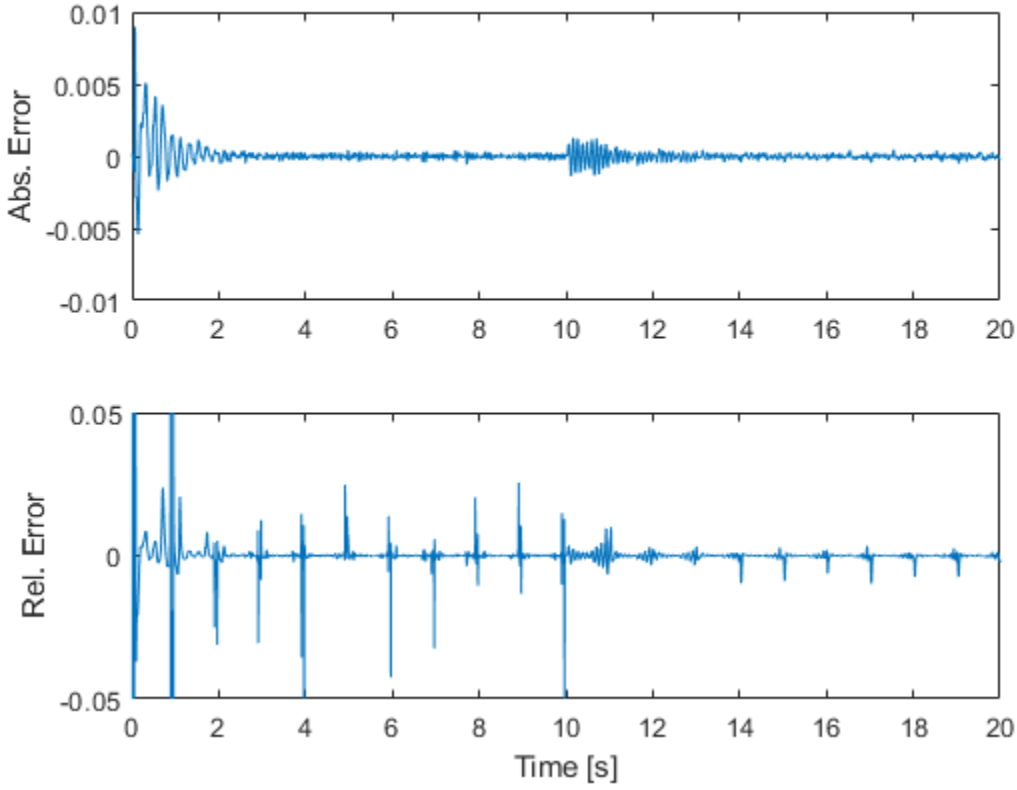
Use the following techniques to validate the parameter estimation:

1. **View output estimate, yhat(t):** The third output argument of the step method is the one-step ahead prediction of the output yhat(t). This is based on current model parameters as well as current and past input-output measurements. The relative and absolute error between y(t) and yhat(t) are measures of the goodness of the fit.
2. **View the parameter covariance estimate, Phat(t):** This is available with the ForgettingFactor and KalmanFilter estimation methods. It is stored in the ParameterCovarianceMatrix property of the estimator. The diagonals of Phat(t) are the estimated variances of the parameters. It should be bounded, and the lower the better.

3 **Simulate the estimated time-varying model:** Use u(t) and the estimated parameters to simulate the model to obtain a simulated output, ysim(t). Then compare y(t) and ysim(t). This is a more strict validation than comparing y(t) and yhat(t) because ysim(t) is generated without the plant output measurements.

The absolute error yf(t)-yhat(t) and the relative error (yf(t)-yhat(t))/yf(t) are:

```
figure();
subplot(2,1,1);
plot(t,yf-yHat);
ylabel('Abs. Error');
subplot(2,1,2);
plot(t,(yf-yHat)./yf);
ylim([-0.05 0.05]);
ylabel('Rel. Error');
xlabel('Time [s]');
```



The absolute errors are on the order of 1e-3, which is small compared to the measured output signal itself. The relative error plot at the bottom confirms this, with errors being less than 5% except at the beginning of the simulation.

The diagonals of the parameter covariance matrix, scaled by the variance of the residuals y(t)-yhat(t), capture the variances of parameter estimates. The square-root of the variances are the standard deviations of the parameter estimates. The first three elements on the diagonals are the three parameters estimated in the A(q) polynomial. The last element is the single parameter in the B(q) polynomial. Let's look at the first estimated parameter in A(q)

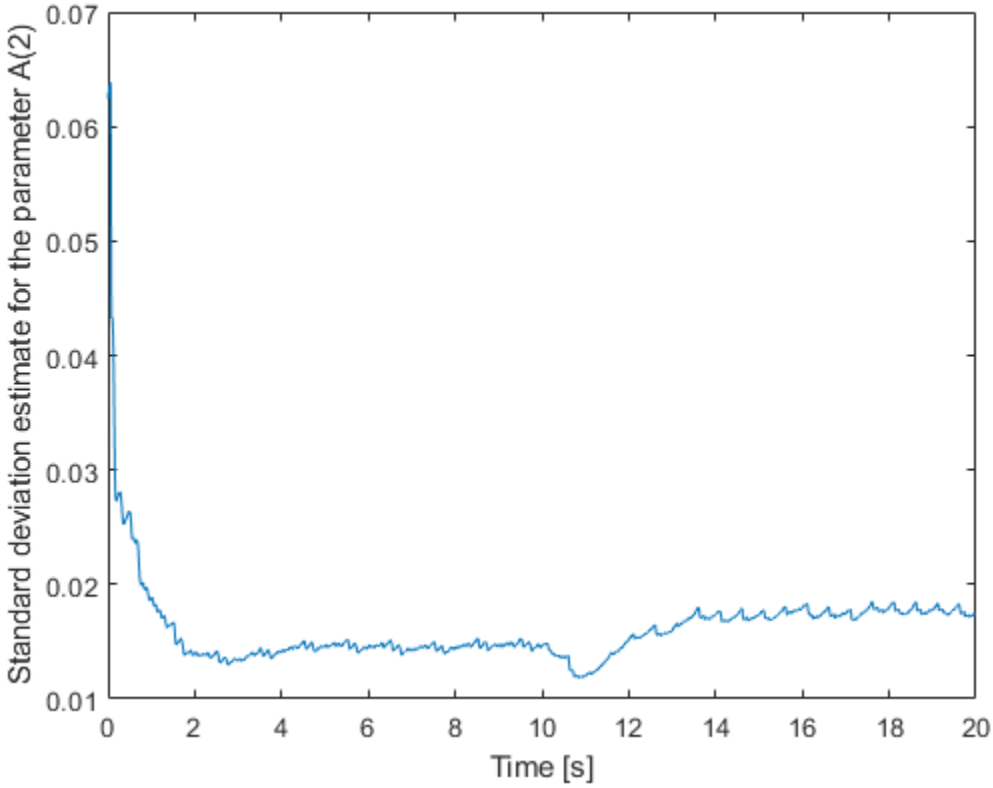
```
noiseVariance = var(yf-yHat);
X.A(2) % The first estimated parameter. X.A(1) is fixed to 1
sqrt(X.ParameterCovariance(1,1)*noiseVariance)
```

```
ans =

   -2.8635
```

```
ans =

    0.0175
```

The standard deviation 0.0175 is small relative to the absolute value of the parameter value 2.86. This indicates good confidence in the estimated parameter.

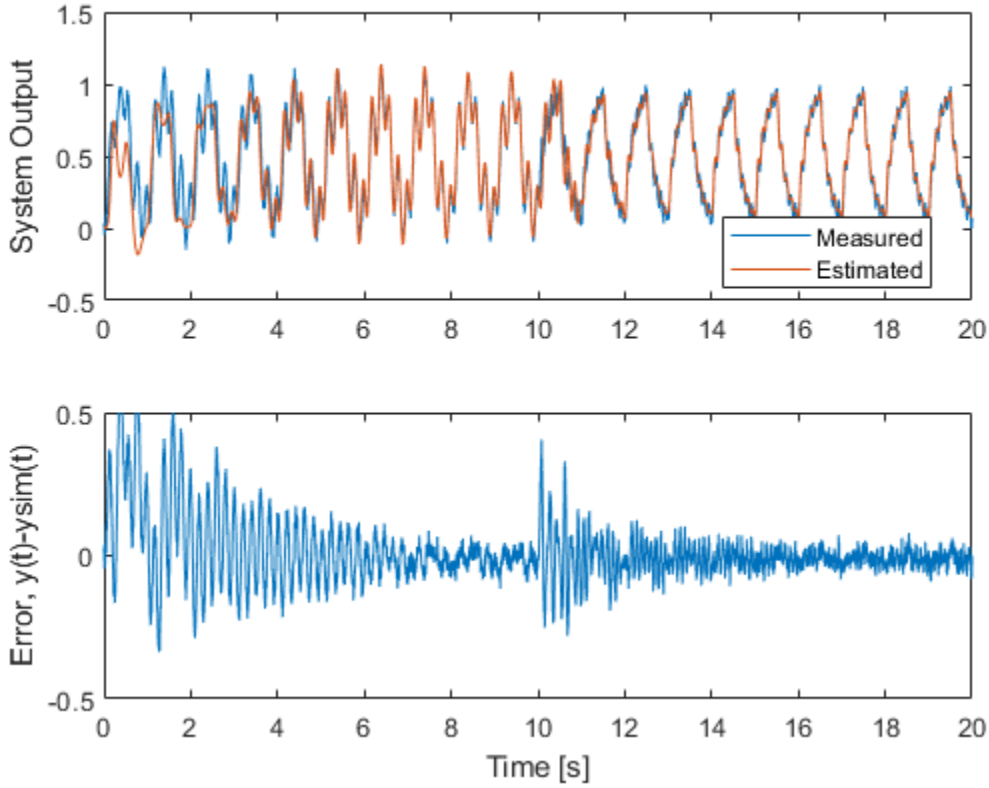
```
figure();
plot(t,sqrt(PHat(:,1,1)*noiseVariance));
ylabel('Standard deviation estimate for the parameter A(2)')
xlabel('Time [s]');
```



The uncertainty is small and bounded throughout the estimation. However, note that the parameter standard deviations are also estimates. They are based on the assumption that the residuals y(t)-yhat(t) are white. This depends on the estimation method, its associated parameters, structure of the estimated model, and the input signal u. Differences between the assumed and the actual model structure, lack of persistent input excitation, or unrealistic estimation method settings can lead to overly optimistic or pessimistic uncertainty estimates.

Lastly, simulate the estimated ARX model using the stored history of estimated parameters. This simulation can also be done simultaneously with the estimation loop for validation during online operation.

```
ysim = zeros(size(y));
idx = numel(B2):-1:1;
for ct=(1+numel(B2)):numel(t)
    idx = idx + 1;
    ysim(ct) = u(idx(1))*B(idx(1),:)-ysim(idx(2:end))*A(ct,2:end)';
end
figure();
subplot(2,1,1);
plot(t,y,t,ysim);
ylabel('System Output');
legend('Measured','Estimated','Location','Best');
subplot(2,1,2);
plot(t,y-ysim);
ylim([-0.5 0.5]);
ylabel('Error, y(t)-ysim(t)');
xlabel('Time [s]');
```



The error is large initially, but it settles to a smaller value around t=5s for the first operating mode. The large initial error can be reduced by providing the estimator an initial guess for the model parameters and initial parameter covariance. When the plant switches to the second mode, the errors grow initially but settle down as time goes on as well. This gives confidence that the estimated model parameters are good at capturing the model behavior for the given input signal.

Summary

You performed online parameter estimation for an ARX model. This model captured the dynamics of a mode-switching plant. You validated the estimated model by looking at the

error between estimated, simulated, measured system outputs as well as the parameter covariance estimates.

See Also

`clone` | `isLocked` | `recursiveAR` | `recursiveARMA` | `recursiveARMAX` | `recursiveARX` | `recursiveBJ` | `recursiveLS` | `recursiveOE` | `release` | `reset` | `step`

Related Examples

- "Perform Online Parameter Estimation at the Command Line" on page 16-26
- "Validate Online Parameter Estimation at the Command Line" on page 16-11
- "Generate Code for Online Parameter Estimation in MATLAB" on page 16-31
- "Line Fitting with Online Recursive Least Squares Estimation" on page 16-57

Online Recursive Least Squares Estimation

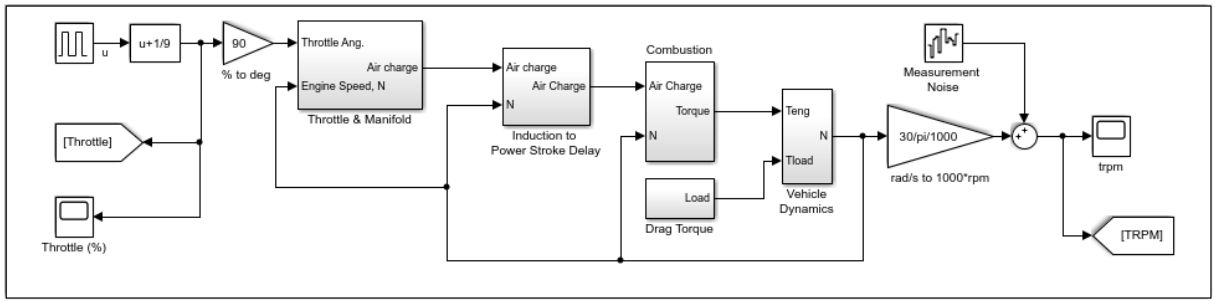
This example shows how to implement an online recursive least squares estimator. You estimate a nonlinear model of an internal combustion engine and use recursive least squares to detect changes in engine inertia.

Engine Model

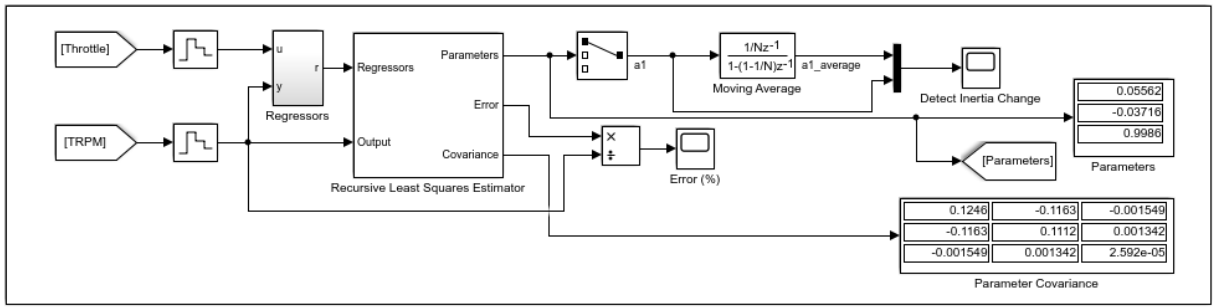
The engine model includes nonlinear elements for the throttle and manifold system, and the combustion system. The model input is the throttle angle and the model output is the engine speed in rpm.

```
open_system('iddemo_engine');
sim('iddemo_engine')
```

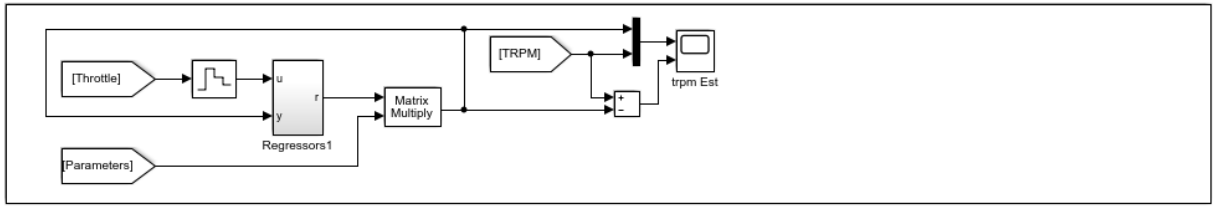
Engine Model



Recursive Estimator



Estimated Model



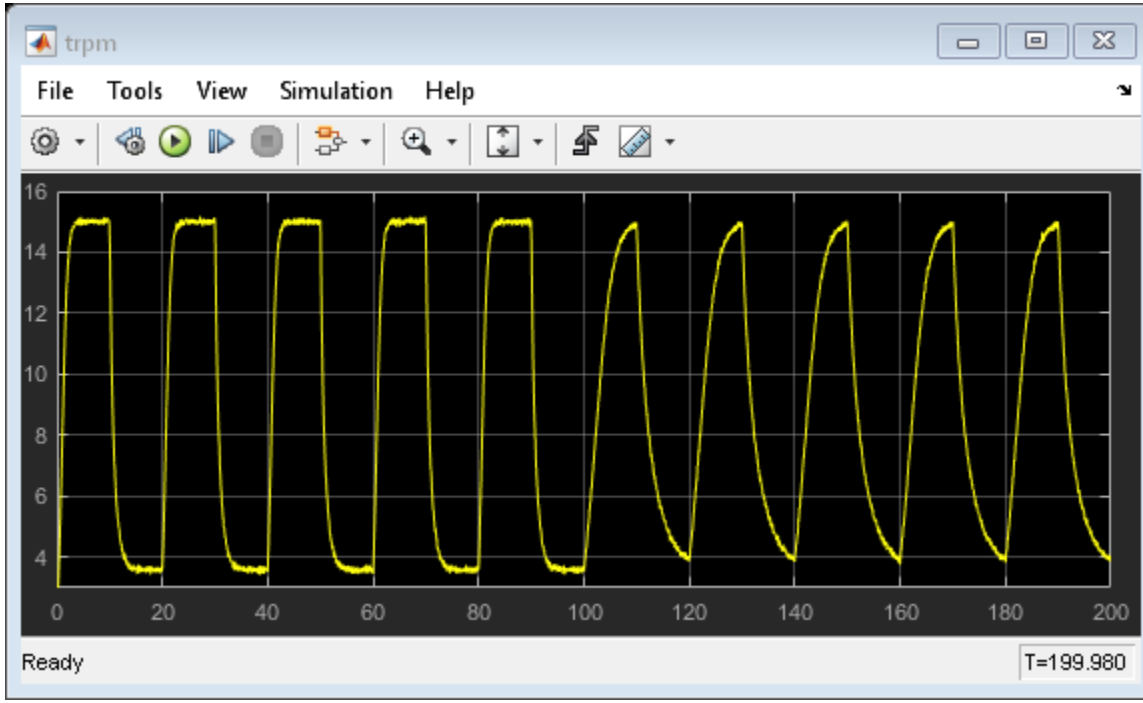
Copyright 2013-2015 The MathWorks, Inc.

The engine model is set up with a pulse train driving the throttle angle from open to closed. The engine response is nonlinear, specifically the engine rpm response time when the throttle is open and closed are different.

At 100 seconds into the simulation an engine fault occurs causing the engine inertia to increase (the engine inertia, `J`, is modeled in the `iddemo_engine/Vehicle Dynamics`

block). The inertia change causes engine response times at open and closed throttle positions to increase. You use online recursive least squares to detect the inertia change.

```
open_system('iddemo_engine/trpm')
```



Estimation Model

The engine model is a damped second order system with input and output nonlinearities to account for different response times at different throttle positions. Use the recursive least squares block to identify the following discrete system that models the engine:
$y_n = a_1 u_{n-1} + a_2 u_{n-1}^2 + a_3 y_{n-1}$

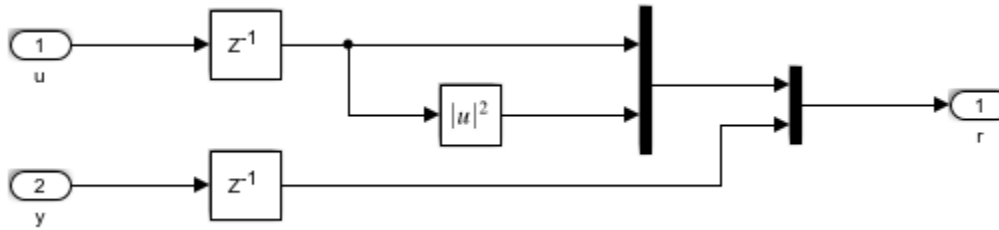
Since the estimation model does not explicitly include inertia we expect the a values to change as the inertia changes. We use the changing a values to detect the inertia change.

The engine has significant bandwidth up to 16Hz. Set the estimator sampling frequency to 2*160Hz or a sample time of $T_s = 0.003$ seconds.

Recursive Least Squares Estimator Block Setup

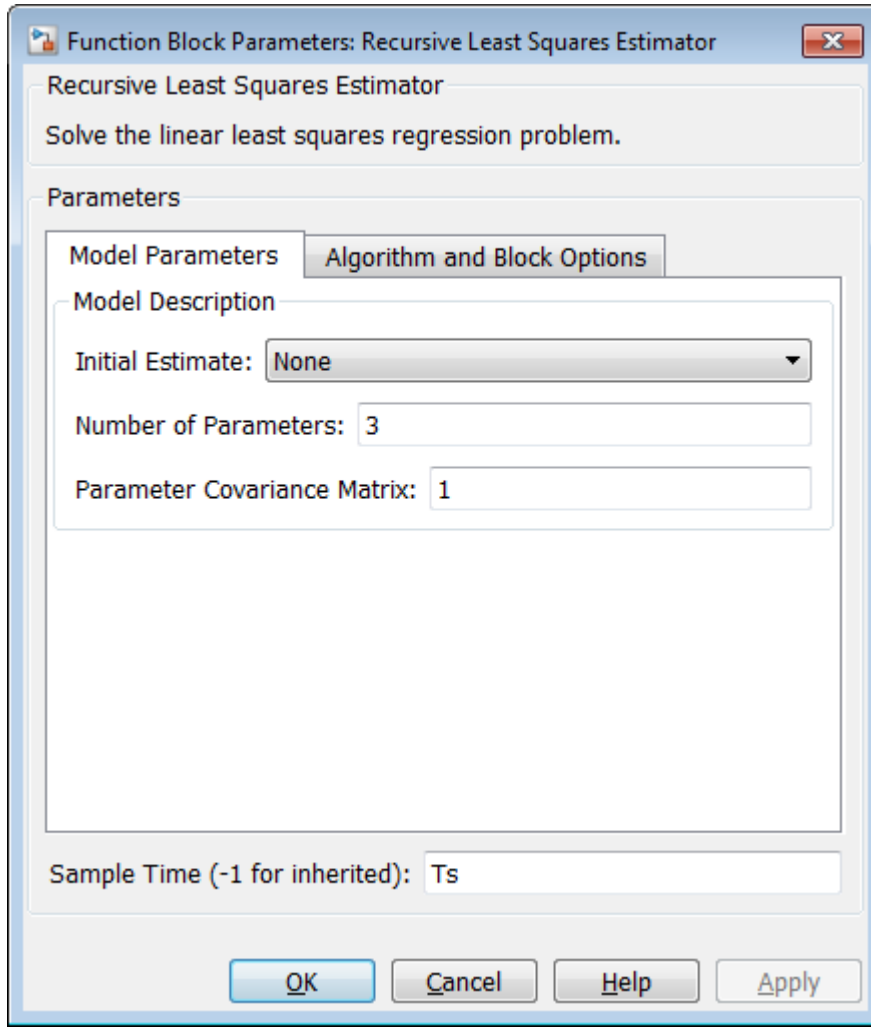
The $u_{n-1}, u_{n-1}^2, y_{n-1}$ terms in the estimated model are the *model regressors* and inputs to the recursive least squares block that estimates the a values. You can implement the regressors as shown in the `iddemo_engine/Regressors` block.

```
open_system('iddemo_engine/Regressors');
```



Configure the Recursive Least Squares Estimator block:

- **Initial Estimate:** None. By default, the software uses a value of 1.
- **Number of parameters**: 3, one for each a regressor coefficient.
- **Parameter Covariance Matrix**: 1, the amount of uncertainty in initial guess of 1. Concretely, treat the estimated parameters as a random variable with variance 1.
- **Sample Time**: T_s.

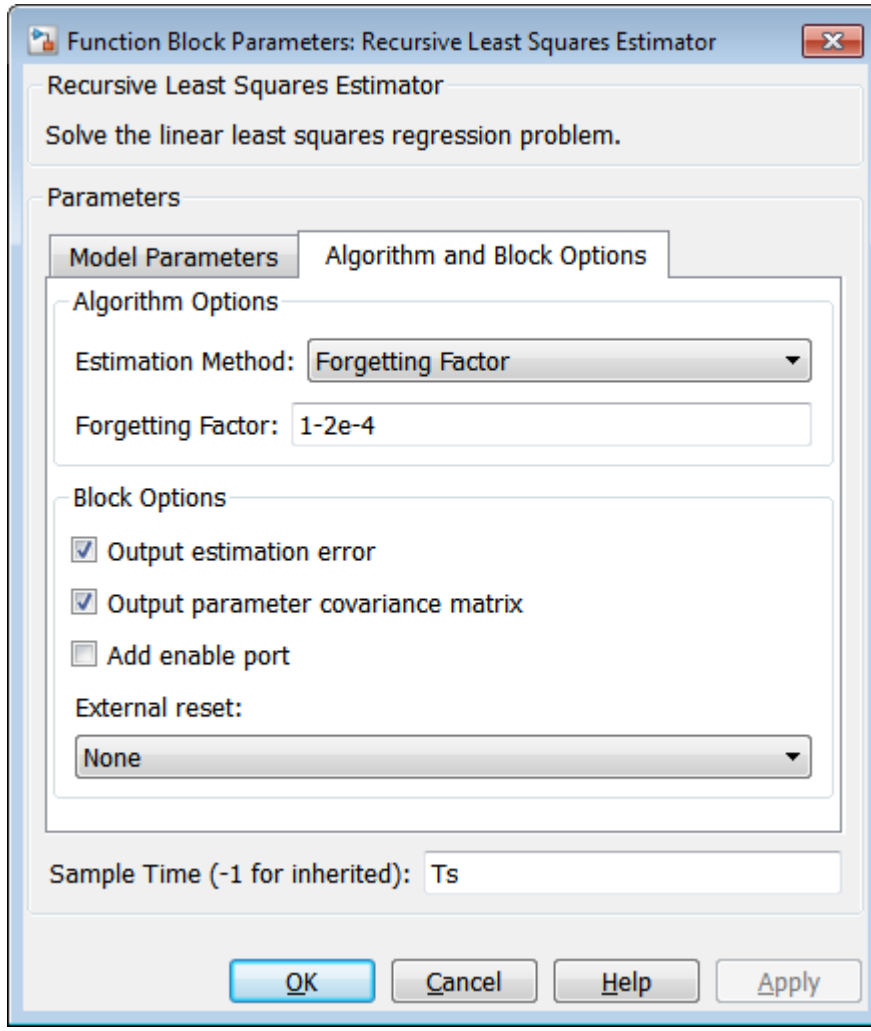


Click **Algorithm and Block Options** to set the estimation options:

- **Estimation Method**: `Forgetting Factor`
- **Forgetting Factor**: 1-2e-4. Since the estimated a values are expected to change with the inertia, set the forgetting factor to a value less than 1. Choose λ = 1-2e-4 which corresponds to a memory time constant of $T_0 = \frac{T_s}{1-\lambda}$ or 15 seconds. A 15 second

memory time ensures that significant data from both the open and closed throttle position are used for estimation as the position is changed every 10 seconds.

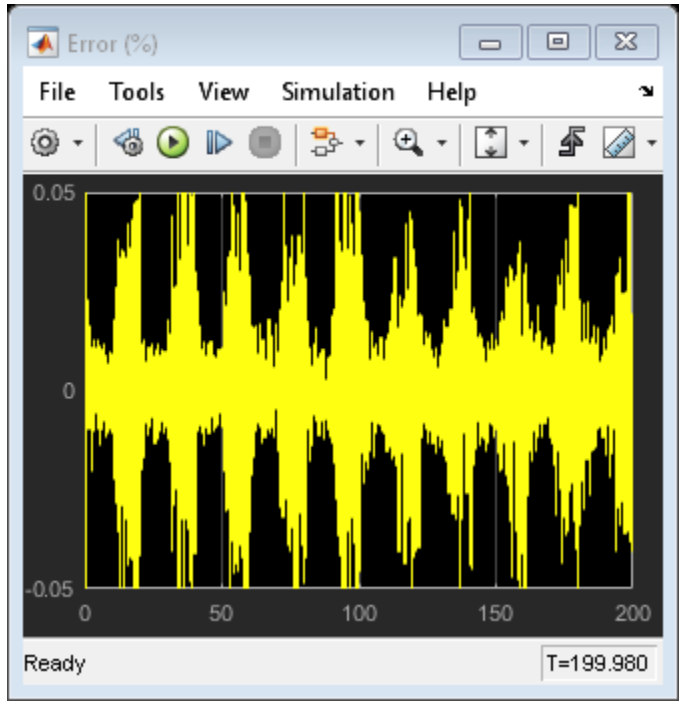
- Select the **Output estimation error** check box. You use this block output to validate the estimation.
- Select the **Output parameter covariance matrix** check box. You use this block output to validate the estimation.
- Clear the **Add enable port** check box.
- **External reset**: `None`.



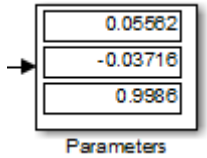
Validating the Estimated Model

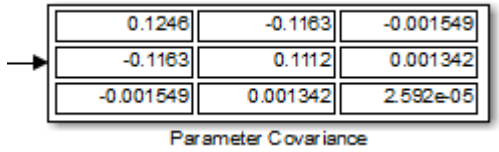
The `Error` output of the `Recursive Least Squares Estimator` block gives the one-step-ahead error for the estimated model. This error is less than 5% indicating that for one-step-ahead prediction the estimated model is accurate.

```
open_system('iddemo_engine/Error (%)')
```



The diagonal of the parameter covariances matrix gives the variances for the a_n parameters. The a_3 variance is small relative to the parameter value indicating good confidence in the estimated value. In contrast, the a_1, a_2 variances are large relative to the parameter values indicating a low confidence in these values.





While the small estimation error and covariances give confidence that the model is being estimated correctly, it is limited in that the error is a one-step-ahead predictor. A more rigorous check is to use the estimated model in a simulation model and compare with the actual model output. The **Estimated Model** section of the simulink model implements this.

The `Regressors1` block is identical to the `Regressors` block use in the recursive estimator. The only difference is that the y signal is not measured from the plant but fed back from the output of the estimated model. The Output of the regressors block is multiplied by estimated a_n values to give $\hat{y}_n$ an estimate of the engine speed.

```
open_system('iddemo_engine/trpm Est')
```

The estimated model output matches the model output fairly well. The steady-state values are close and the transient behavior is slightly different but not significantly so. Note that after 100 seconds when the engine inertia changes the estimated model output

differs slightly more from the model output. This implies that the chosen regressors cannot capture the behavior of the model as well after the inertia change. This also suggests a change in system behavior.

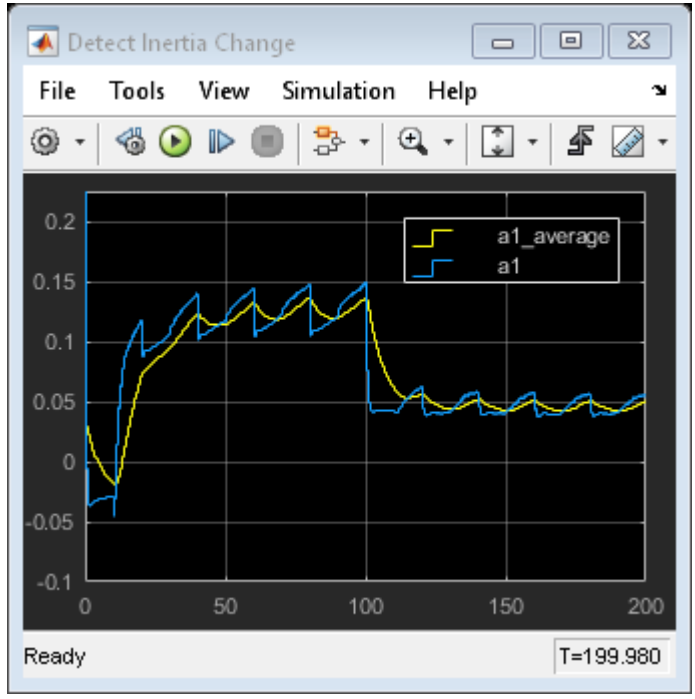
The estimated model output combined with the low one-step-ahead error and parameter covariances gives us confidence in the recursive estimator.

Detecting Changes in Engine Inertia

The engine model is setup to introduce an inertia change 100 seconds into the simulation. The recursive estimator can be used to detect the change in inertia.

The recursive estimator takes around 50 seconds to converge to an initial set of parameter values. To detect the inertia change we examine the a_1 model coefficient that influences the $a_1 u_{n-1}$ term of the estimated model.

```
open_system('iddemo_engine/Detect Inertia Change')
```



The covariance for a_1, 0.05562, is large relative to the parameter value 0.1246 indicating low confidence in the estimated value. The time plot of a_1 shows why the covariance is large. Specifically a_1 is varying as the throttle position varies indicating that the estimated model is not rich enough to fully capture different rise times at different throttle positions and needs to adjust a_1. However, we can use this to identify the inertia changes as the average value of a_1 changes as the inertia changes. You can use a threshold detector on the moving average of the a_1 parameter to detect changes in the engine inertia.

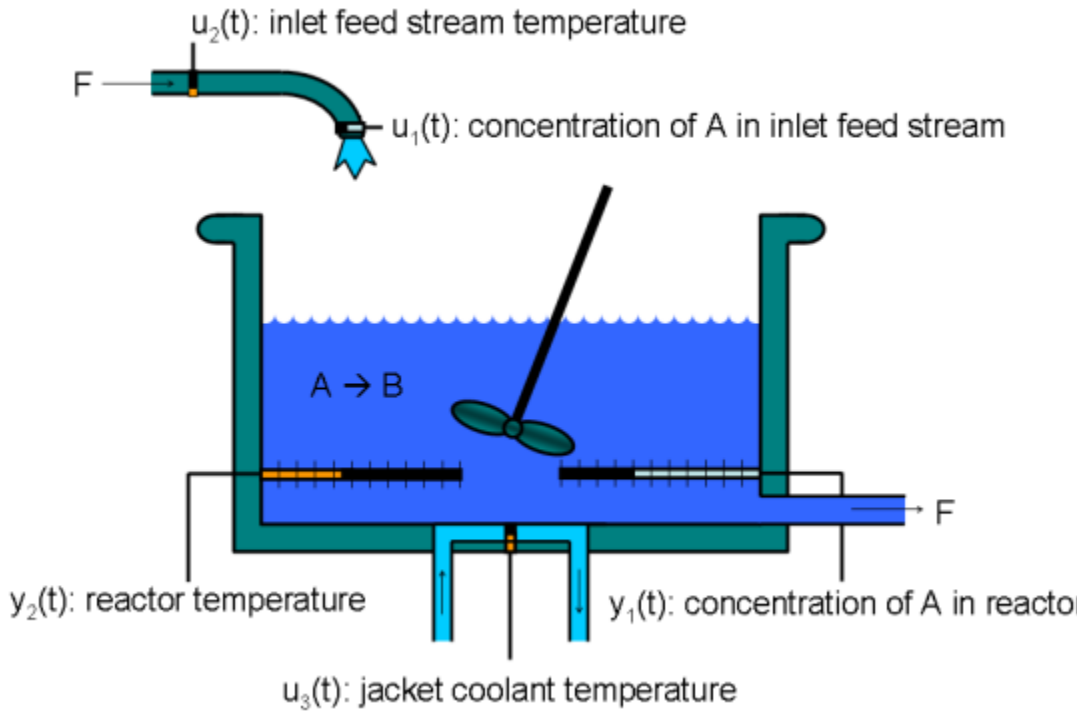
```
bdclose('iddemo_engine')
```

Online ARMAX Polynomial Model Estimation

This example shows how to implement an online polynomial model estimator. You estimate two ARMAX models for a nonlinear chemical reaction process. These models capture the behavior of the process at two operating conditions. The model behavior is identified online and used to adjust the gains of an adaptive PI controller during system operation.

Continuously Stirred Tank Reactor

A Continuously Stirred Tank Reactor (CSTR) is a common chemical system in the process industry. A schematic of the CSTR system is:



This is a jacketed diabatic (i.e., nondiabatic) tank reactor described extensively in Bequette's book "Process Dynamics: Modeling, Analysis and Simulation", published by Prentice-Hall, 1998. The vessel is assumed to be perfectly mixed, and a single first-order

exothermic and irreversible reaction, A --> B, takes place. The inlet stream of reagent A is fed to the tank at a constant rate. After stirring, the end product streams out of the vessel at the same rate as reagent A is fed into the tank (the volume in the reactor tank is constant). Details of the operation of the CSTR and its 2-state nonlinear model used in this example are explained in the example “Non-Adiabatic Continuous Stirred Tank Reactor: MATLAB File Modeling with Simulations in Simulink®”.

The inputs of the CSTR model are:

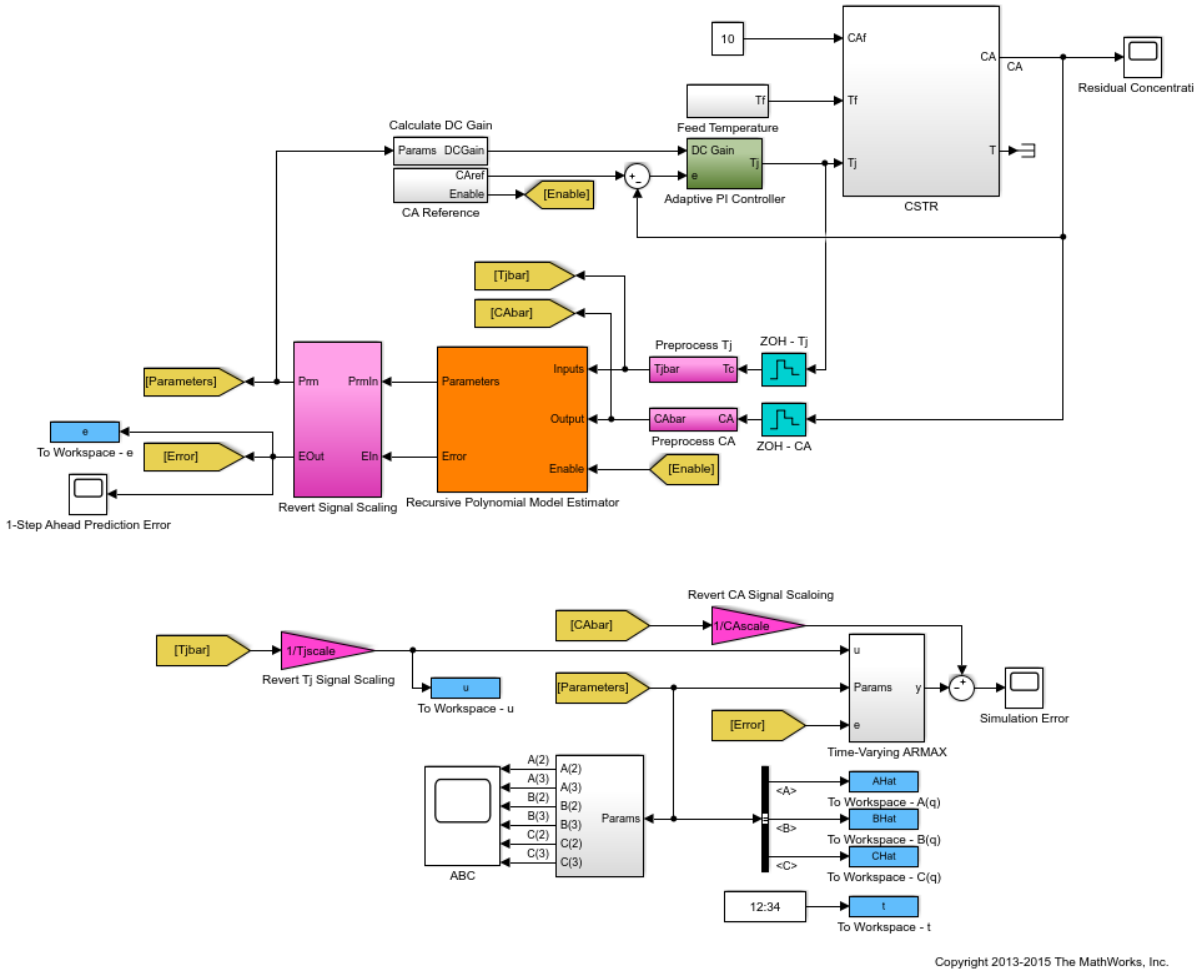
$u_1(t) = C_{Af}(t)$ Concentration of A in inlet feed stream$[kgmol/m^3]$
$u_2(t) = T_f(t)$ Inlet feed stream temperature$[K]$
$u_3(t) = T_j(t)$ Jacket coolant temperature$[K]$

and the outputs (y(t)), which are also the states of the model (x(t)), are:

$y_1(t) = x_1(t) = C_A(t)$ Concentration of A in reactor tank$[kgmol/m^3]$
$y_2(t) = x_2(t) = T(t)$ Reactor temperature$[K]$

The control objective is to maintain the concentration of reagent A, $C_A(t)$ at the desired level $C_{Aref}(t)$, which changes over time. The jacket temperature $T_j(t)$ is manipulated by a PI controller in order to reject disturbances arising from the inlet feed stream temperature $T_f(t)$. The input of the PI controller is the tracking error signal, $C_{Aref}(t) - C_A(t)$. The inlet feed stream concentration, $C_{Af}(t)$, is assumed to be constant. The Simulink model `iddemo_cstr` implements the CSTR plant as the block `CSTR`.

```
open_system('iddemo_cstr');
```



The $T_f(t)$ feed temperature input consists of a white noise disturbance on top of a constant offset. The noise power is 0.0075 [K^2]. This noise level causes up to 2% deviation from the desired $C_{Aref}(t)$.

The $C_{Aref}(t)$ signal in this example contains a step change from 1.5 [kgmol/m^3] to 2 [kgmol/m^3] at time $t = 400$. In addition to this step change, $C_{Aref}(t)$ also contains a white noise perturbation for t in the [0,200) and [400,600) ranges. The power of this

white noise signal is 0.015. The noise power is adjusted empirically to approximately give a signal-to-noise ratio of 10. Not having sufficient excitation in the reference signal in closed-loop identification can lead to not having sufficient information to identify a unique model. The implementation of $C_{Aref}(t)$ is in the `iddemo_cstr/CA Reference` block.

Online Estimation for Adaptive Control

It is known from the nonlinear model that the CSTR output $C_A(t)$ is more sensitive to the control input $T_j(t)$ at higher $C_A(t)$ levels. The Recursive Polynomial Model Estimator block is used to detect this change in sensitivity. This information is used to adjust the gains of the PI controller as $C_A(t)$ varies. The aim is to avoid having a a high gain control loop which may lead to instability.

You estimate a discrete transfer-function from $T_j(t)$ to $C_A(t)$ online with the Recursive Polynomial Model Estimator block. The adaptive control algorithm uses the DC gain of this transfer function. The tracking error $C_{Aref}(t) - C_A(t)$, is divided by the normalized DC gain of the estimated transfer function. This normalization is done to have a gain of 1 on the tracking error at the initial operating point, for which the PI controller is designed. For instance, the error signal is divided by 2 if the DC gain becomes 2 times its original value. This corresponds to dividing the PI controller gains by 2. This adpative controller is implemented in `iddemo_cstr/Adaptive PI Controller`.

Recursive Polynomial Model Estimator Block Inputs

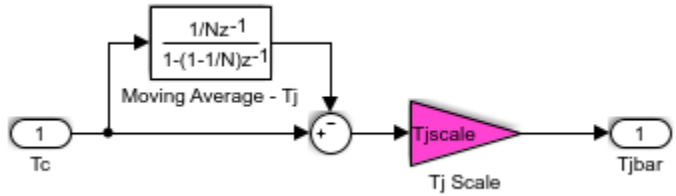
The 'Recursive Polynomial Model Estimator' block is found under the `System Identification Toolbox/Estimators` library in Simulink. You use this block to estimate linear models with ARMAX structure. ARMAX models have the form:
$A(q)\bar{y}(t) = B(q)\bar{u}(t) + C(q)\bar{e}(t)$

$A(q) = 1 + a_1 z^{-1} + a_2 z^{-2} + \cdots + a_{na} z^{-na}$
$B(q) = (b_0 1 + b_1 z^{-1} + b_2 z^{-2} + \cdots + a_{nb-1} z^{-nb+1}) z^{-nk}$
$C(q) = 1 + c_1 z^{-1} + c_2 z^{-2} + \cdots + c_{nc} z^{-nc}$

- The `Inputs` and `Output` inport of the recursive polynomial model estimator block correspond to $\bar{u}(t)$ and $\bar{y}(t)$ respectively. For the CSTR model $\bar{y}$ and $\bar{u}$ are deviations from the jacket temperature and A concentration trim operating points: $\bar{y} = C_A(t) - \bar{C}_A(t)$, $\bar{u} = T_j(t) - \bar{T}_j(t)$. It is good to scale $\bar{u}$ and $\bar{y}$ to have a peak

amplitude of 1 to improve the numerical condition of the estimation problem. The trim operating points, $\bar{C}_A(t)$ and $\bar{T}_j(t)$, are not known exactly before system operation. They are estimated and extracted from the measured signals by using a first-order moving average filter. These preprocessing filters are implemented in the `iddemo_cstr/Preprocess Tj` and `iddemo_cstr/Preprocess CA` blocks.

```
open_system('iddemo_cstr/Preprocess Tj');
```



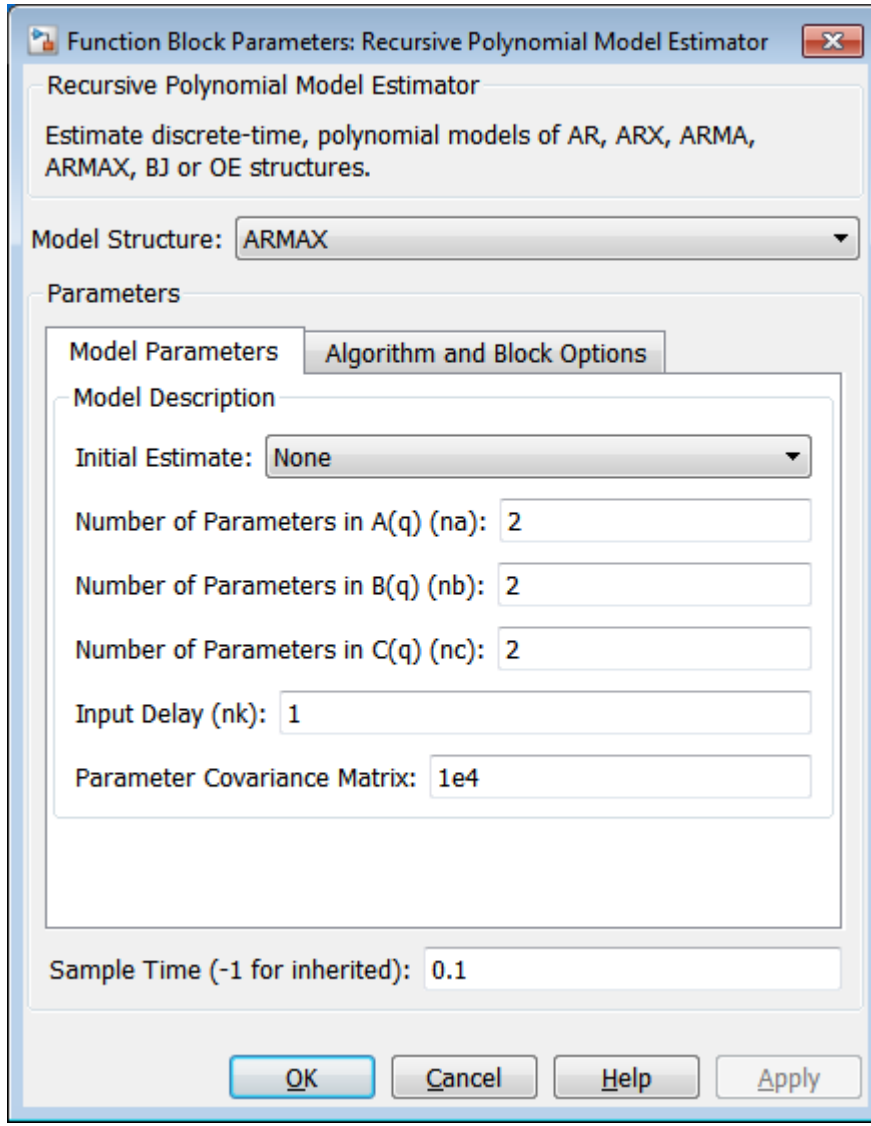
- The optional `Enable` inport of the Recursive Polynomial Model Estimator block controls the parameter estimation in the block. Parameter estimation is disabled when the Enable signal is zero. Parameter estimation is enabled for all other values of the `Enable` signal. In this example the estimation is disabled for the time intervals $t \in [200, 400)$ and $t \in [600, 800)$. During these intervals the measured input $T_j(t)$ does not contain sufficient excitation for closed-loop system identification.

Recursive Polynomial Model Estimator Block Setup

Configure the block parameters to estimate a second-order ARMAX model. In the **Model Parameters** tab, specify:

- **Model Structure:** ARMAX. Choose ARMAX since the current and past values of the disturbances acting on the system, $T_f(t)$, are expected to impact the CSTR system output $C_A(t)$.
- **Initial Estimate:** None. By default, the software uses a value of 0 for all estimated parameters.
- **Number of parameters in A(q) (na)**: `2`. The nonlinear model has 2 states.
- **Number of parameters in B(q) (nb)**: `2`.
- **Number of parameters in C(q) (nc)**: `2`. The estimated model corresponds to a second order model since the maximum of na, nb, and nc are 2.

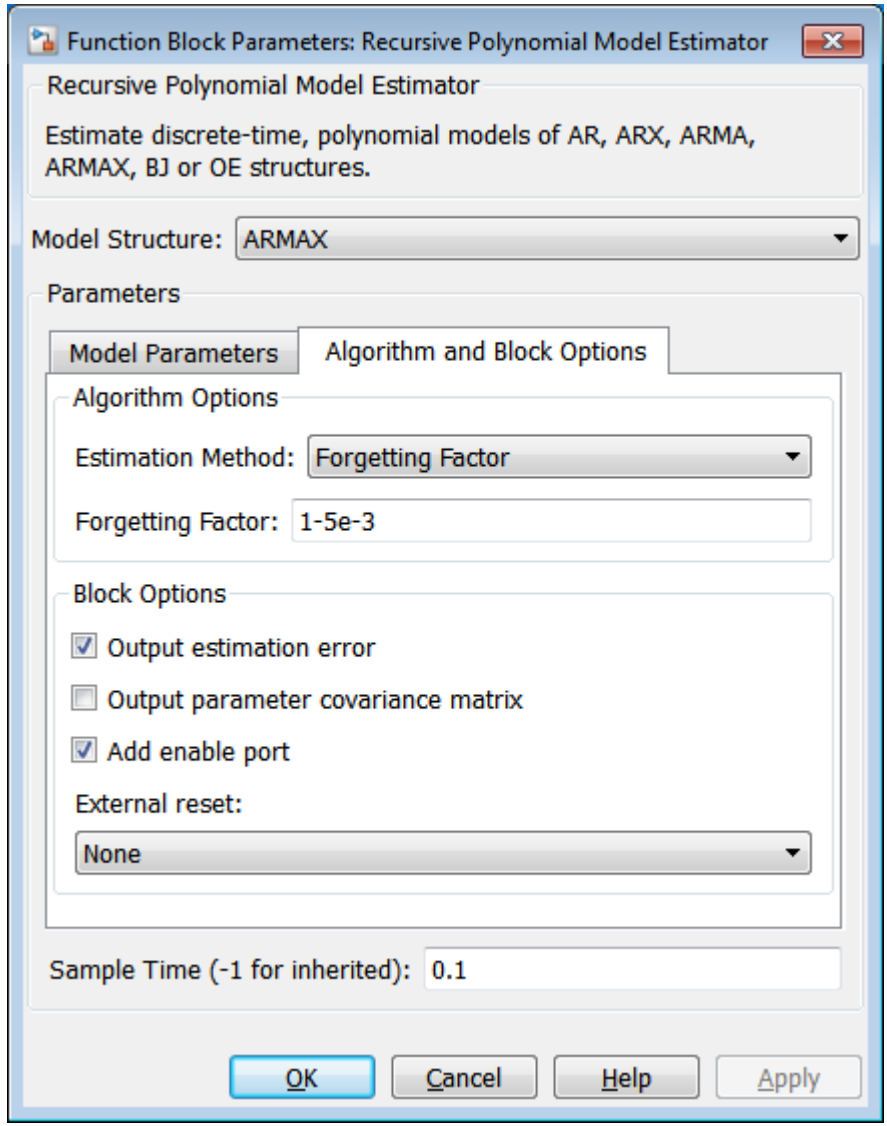
- **Input Delay (nk)**: `1`. Like most physical systems, the CSTR system does not have direct feedthrough. Also, there are no extra time delays between its I/Os.
- **Parameter Covariance Matrix**: `1e4`. Specify a high covariance value because the initial guess values are highly uncertain.
- **Sample Time**: `0.1`. The CSTR model is known to have a bandwidth of about 0.25Hz. $T_s = 0.1$ chosen such that 1/0.1 is greater than 20 times this bandwidth (5Hz).



Click **Algorithm and Block Options** to set the estimation options:

- **Estimation Method**: `Forgetting Factor`

- **Forgetting Factor**: `1-5e-3`. Since the estimated parameters are expected to change with the operating point, set the forgetting factor to a value less than 1. Choose $\lambda = 1 - 5e - 3$ which corresponds to a memory time constant of $T_0 = \frac{Ts}{1-\lambda} = 100$ seconds. A 100 second memory time ensures that a significant amount data used for identification is coming from the 200 second identification period at each operating point.
- Select the **Output estimation error** check box. You use this block output to validate the estimation.
- Select the **Add enable port** check box. You only want to adapt the estimated model parameters when extra noise is injected in the reference port. The parameter estimation n is disabled through this port when the extra noise is no longer injected.
- **External reset**: `None`.



Recursive Polynomial Model Estimator Block Outputs

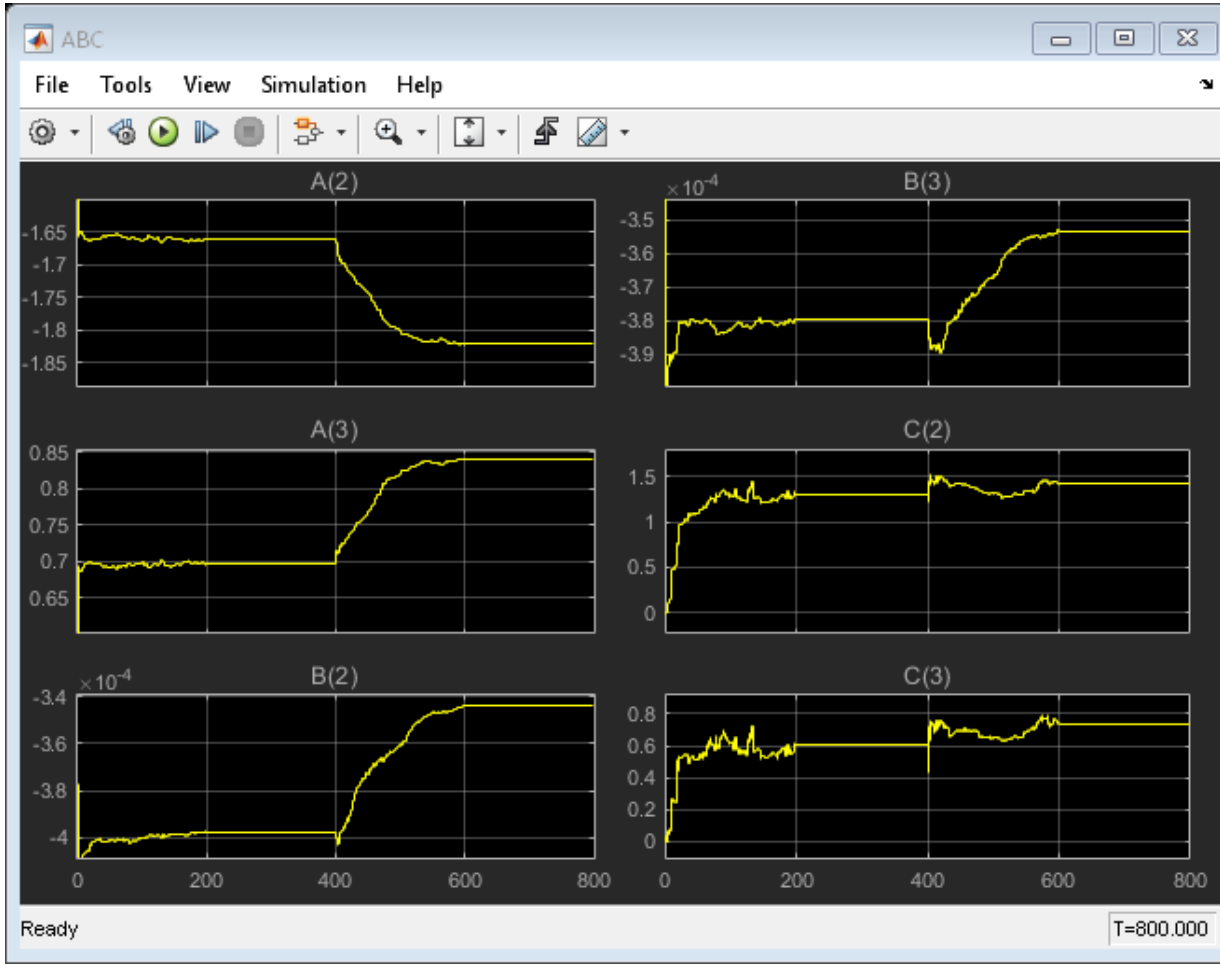
At every time step, the recursive polynomial model estimator provides an estimate for $A(q)$, $B(q)$, $C(q)$, and the estimation error $\bar{e}$. The `Error` outport of the polynomial model

estimator block contains $\bar{e}(t)$ and is also known as the one-step-ahead prediction error. The `Parameters` outport of the block contains the A(q), B(q), and C(q) polynomial coefficients in a bus signal. Given the chosen polynomial orders ($na = 2$, $nb = 2$, $nc = 2$, $nk = 1$) the `Parameters` bus elements contain:

$A(t) = [1\ a_1(t)\ a_2(t)]$
$B(t) = [0\ b_0(t)\ b_1(t)]$
$C(t) = [1\ c_1(t)\ c_2(t)]$

The estimated parameters in the A(q), B(q), and C(q) polynomials change during simulation as follows:

```
sim('iddemo_cstr');
close_system('iddemo_cstr/Preprocess Tj');
open_system('iddemo_cstr/ABC');
```



The parameter estimates quickly change from their initial values of 0 due to the high value chosen for the initial parameter covariance matrix. The parameters in the $A(q)$ and $B(q)$ polynomials approach their values at $t = 200$ rapidly. However, the parameters in the $C(q)$ polynomial show some fluctuations. One reason behind these fluctuations is that the disturbance $T_f(t)$ to CSTR output $C_A(t)$ is not fully modelled by the ARMAX structure. The error model $C(q)$ is not important for the control problem studied here

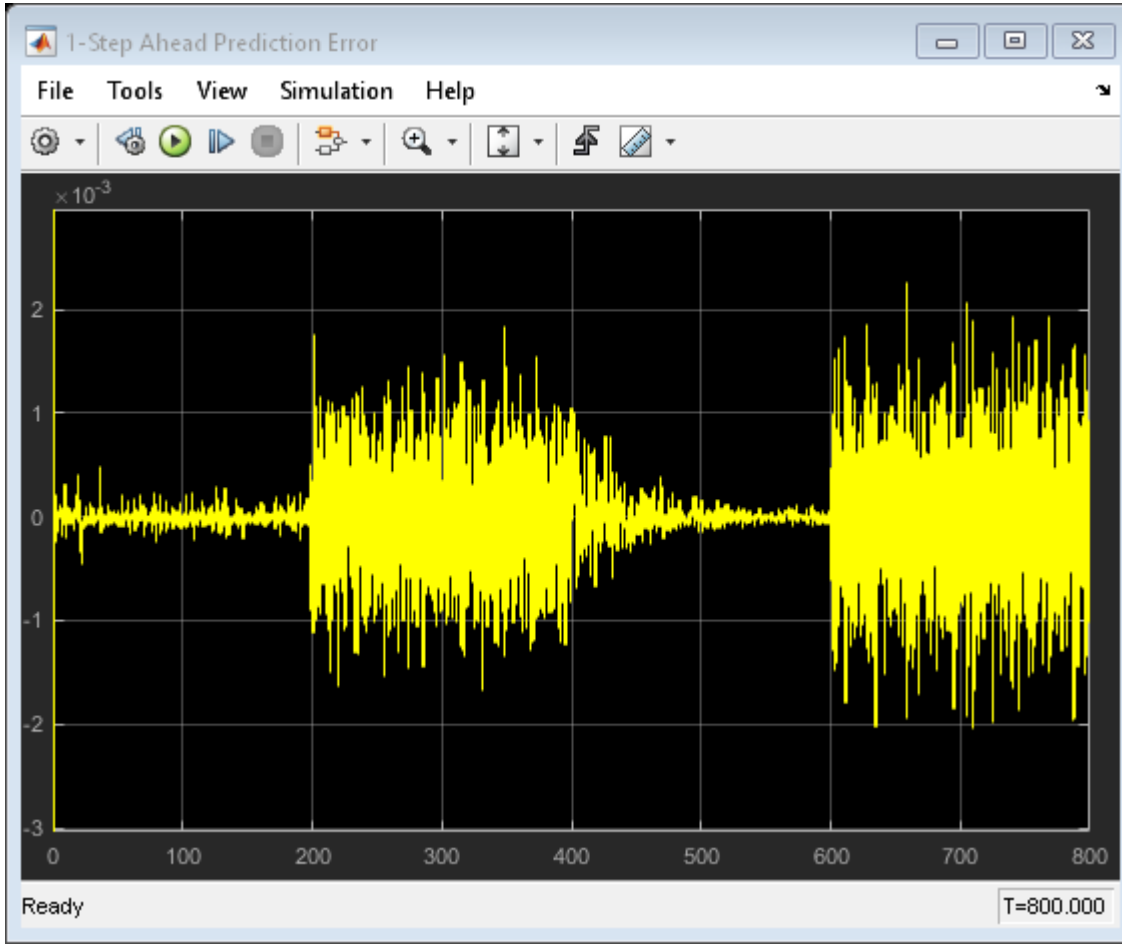
since the $T_j(t)$ to $C_A(t)$ relationship is captured by the transfer function $\frac{B(q)}{A(q)}$. Therefore, the fluctuation in $C(q)$ is not a concern for this identification and control problem.

The parameter estimates are held constant for $t \in [200, 400)$ since the estimator block was disabled for this interval (0 signal to the `Enable` inport). The parameter estimation is enabled at $t = 400$ when the CSTR tank starts switching to its new operating point. The parameters of $A(q)$ and $B(q)$ converge to their new values by $t = 600$, and then held constant by setting the `Enable` port to 0. The convergence of $A(q)$ and $B(q)$ is slower at this operating point. This slow convergence is because of the smaller eigenvalues of the parameter covariance matrix at t=400 compared to the initial 1e4 values set at t=0. The parameter covariance, which is a measure of confidence in the estimates, is updated with each time step. The algorithm quickly changed the parameter estimates when the confidence in estimates were low at t=0. The improved parameter estimates capture the system behavior better, resulting in smaller one-step-ahead prediction errors and smaller eigenvalues in the parameter covariance matrix (increased confidence). The system behavior changes in t=400. However, the block is slower to change the parameter estimates due to the increased confidence in the estimates. You can use the External Reset option of the Recursive Polynomial Model Estimator block to provide a new value for parameter covariance at t=400. To see the value of the parameter covariance, select the **Output parameter covariance matrix** check box in the Recursive Polynomial Model Estimator block.

Validating the Estimated Model

The `Error` output of the `Recursive Polynomial Model Estimator` block gives the one-step-ahead error for the estimated model.

```
close_system('iddemo_cstr/ABC');
open_system('iddemo_cstr/1-Step Ahead Prediction Error');
```

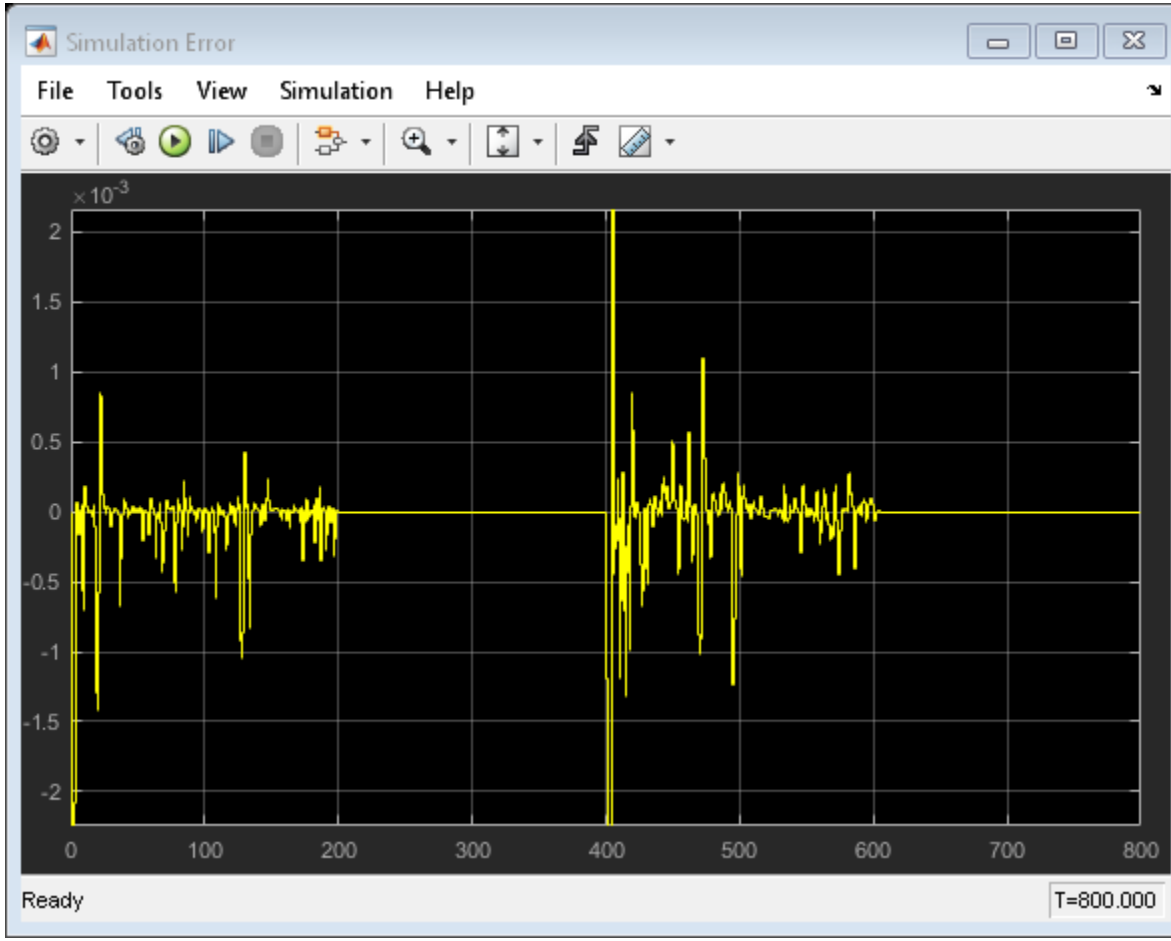


The one-step-ahead error is higher when there are no extra perturbations injected in the $T_j(t)$ channel for system identification. These higher errors may be caused by the lack of sufficient information in the $T_j(t)$ input channel that the estimator block relies on. However, even this higher error is low and bounded when compared to the measured fluctuations in $C_A(t)$. This gives confidence in the estimated parameter values.

A more rigorous check of the estimated model is to simulate the estimated model and compare with the actual model output. The `iddemo_cstr/Time-Varying ARMAX` block implements the time-varying ARMAX model estimated by the Online Polynomial Model

Estimator block. The error between the output of the CSTR system and the estimated time-varying ARMAX model output is:

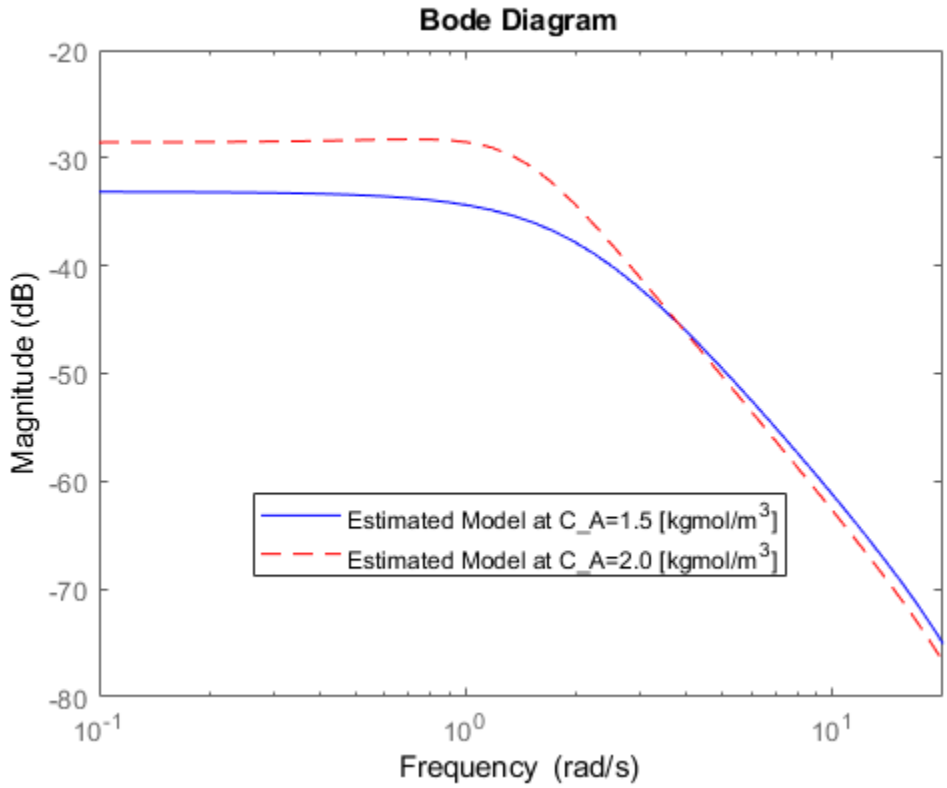
```
close_system('iddemo_cstr/1-Step Ahead Prediction Error');
open_system('iddemo_cstr/Simulation Error');
```



The simulation error is again bounded and low when compared to the fluctuations in the $C_A(t)$. This further provides confidence that the estimated linear models are able to predict the nonlinear CSTR model behavior.

The identified models can be further analyzed in MATLAB. The model estimates for the operating points $C_A = 1.5[kgmol/m^3]$ and $C_A = 2[kgmol/m^3]$ can be obtained by looking at the estimated A(q), B(q), and C(q) polynomials at $t = 200$ and $t = 600$ respectively. Bode plots of these models are:

```
Ts = 0.1;
tidx = find(t>=200,1);
P200 = idpoly(AHat(:,:,tidx),BHat(:,:,tidx),CHat(:,:,tidx),1,1,[],Ts);
tidx = find(t>=600,1);
P600 = idpoly(AHat(:,:,tidx),BHat(:,:,tidx),CHat(:,:,tidx),1,1,[],Ts);
bodemag(P200,'b',P600,'r--',{10^-1,20});
legend('Estimated Model at C_A=1.5 [kgmol/m^3]', ...
       'Estimated Model at C_A=2.0 [kgmol/m^3]', ...
       'Location', 'Best');
```



The estimated model has a higher gain at higher concentration levels. This is in agreement with prior knowledge about the nonlinear CSTR plant. The transfer function at $C_A(t) = 2[kgmol/m^3]$ has a $6dB$ higher gain (double the amplitude) at low frequencies.

Summary

You estimated two ARMAX models to capture the behavior of the nonlinear CSTR plant at two operating conditions. The estimation was done during closed-loop operation with an adaptive controller. You looked at two signals to validate the estimation results: One step ahead prediction errors and the errors between the CSTR plant output and the simulation of the estimation model. Both of these errors signals were bounded and small

compared to the CSTR plant output. This provided confidence in the estimated ARMAX model parameters.

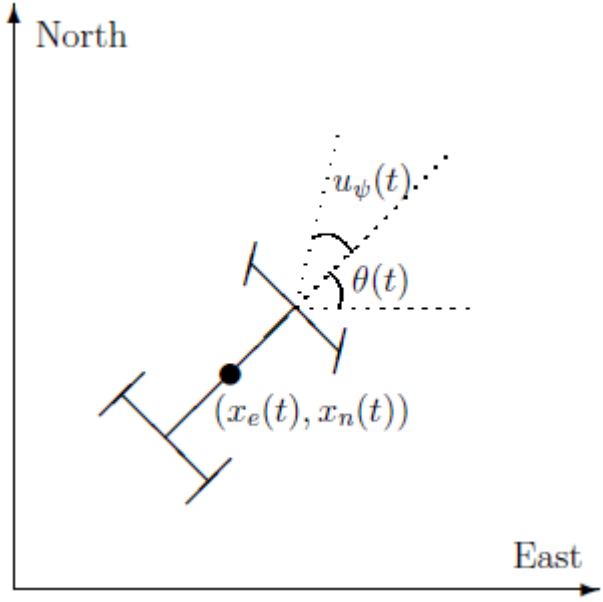
```
bdclose('iddemo_cstr');
```

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 `System Identification Toolbox/Estimators` 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

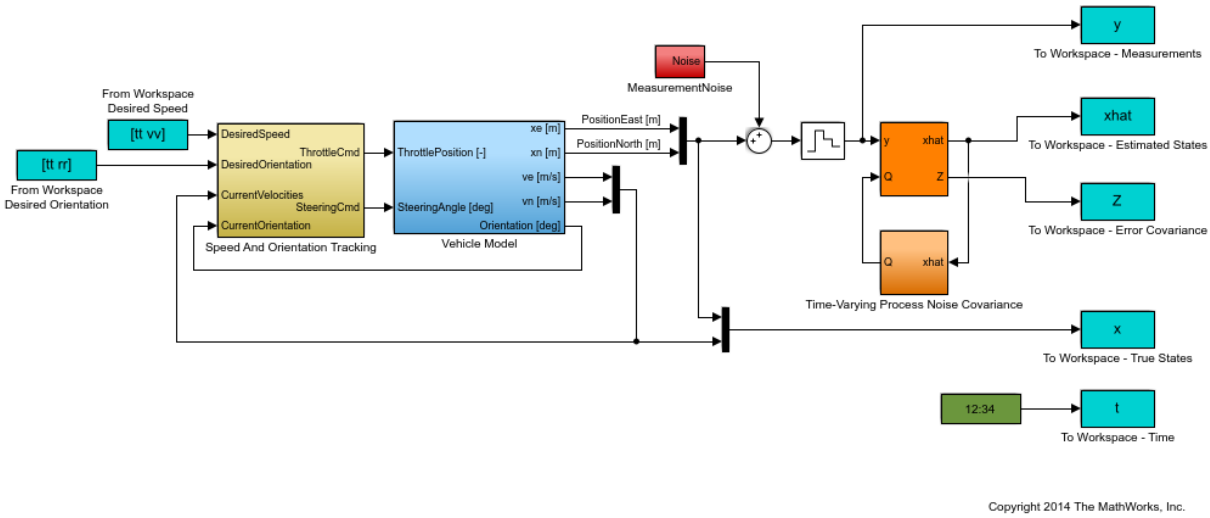
You want to estimate the position and velocity of a ground vehicle in the north and east directions. The vehicle can move freely in the two-dimensional space without any constraints. You design a multi-purpose navigation and tracking system that can be used for any object and not just a vehicle.



$x_e(t)$ 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 disrete 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 measurment 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]:

$f_{sat}(z) = min(max(z, 25), 625)$

$$Q[n] = \begin{bmatrix} 1 + \frac{250}{f_{sat}(\hat{\dot{x}}_e^2)} & 0 \\ 0 & 1 + \frac{250}{f_{sat}(\hat{\dot{x}}_n^2)} \end{bmatrix}$$

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 implies a naive approach that assumes that the velocity changes in north and east direction are uncorrelated.

Kalman Filter Block Inputs and Setup

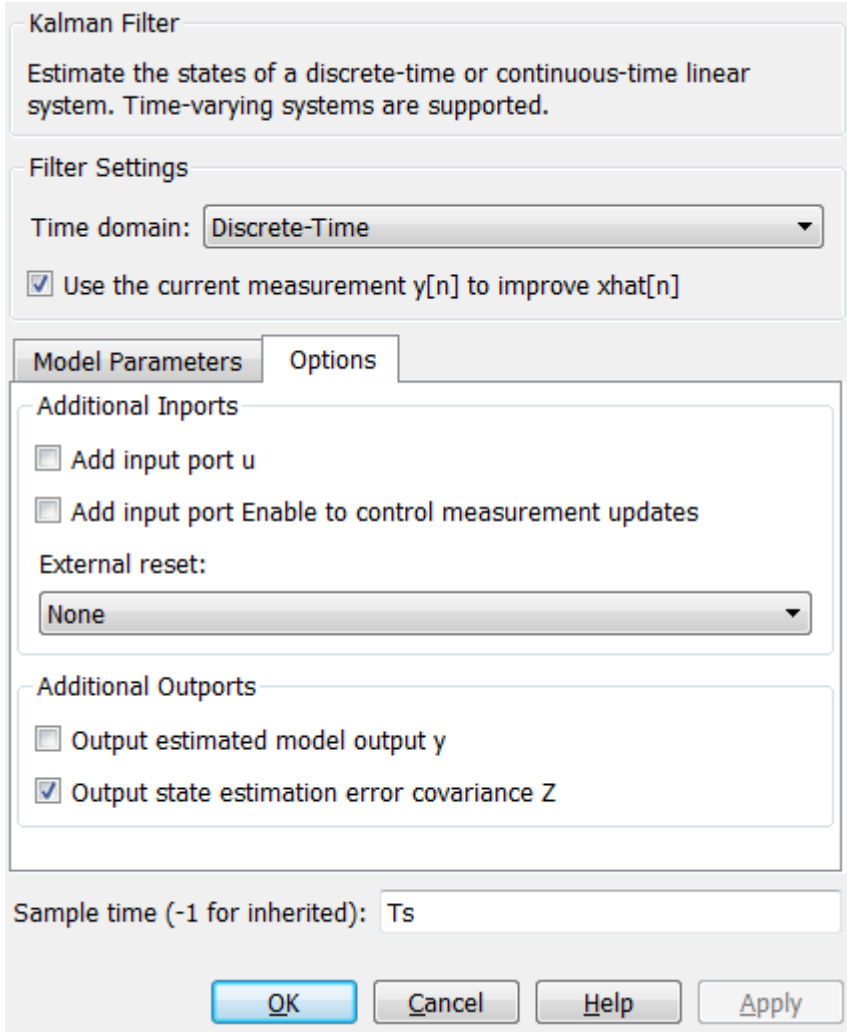
The 'Kalman Filter' block is in the `System Identification Toolbox/Estimators` library in Simulink. It is also in `Control System Toolbox` library. Configure the block parameters for discrete-time state estimation. Specify the following **Filter Settings** parameters:

- **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.

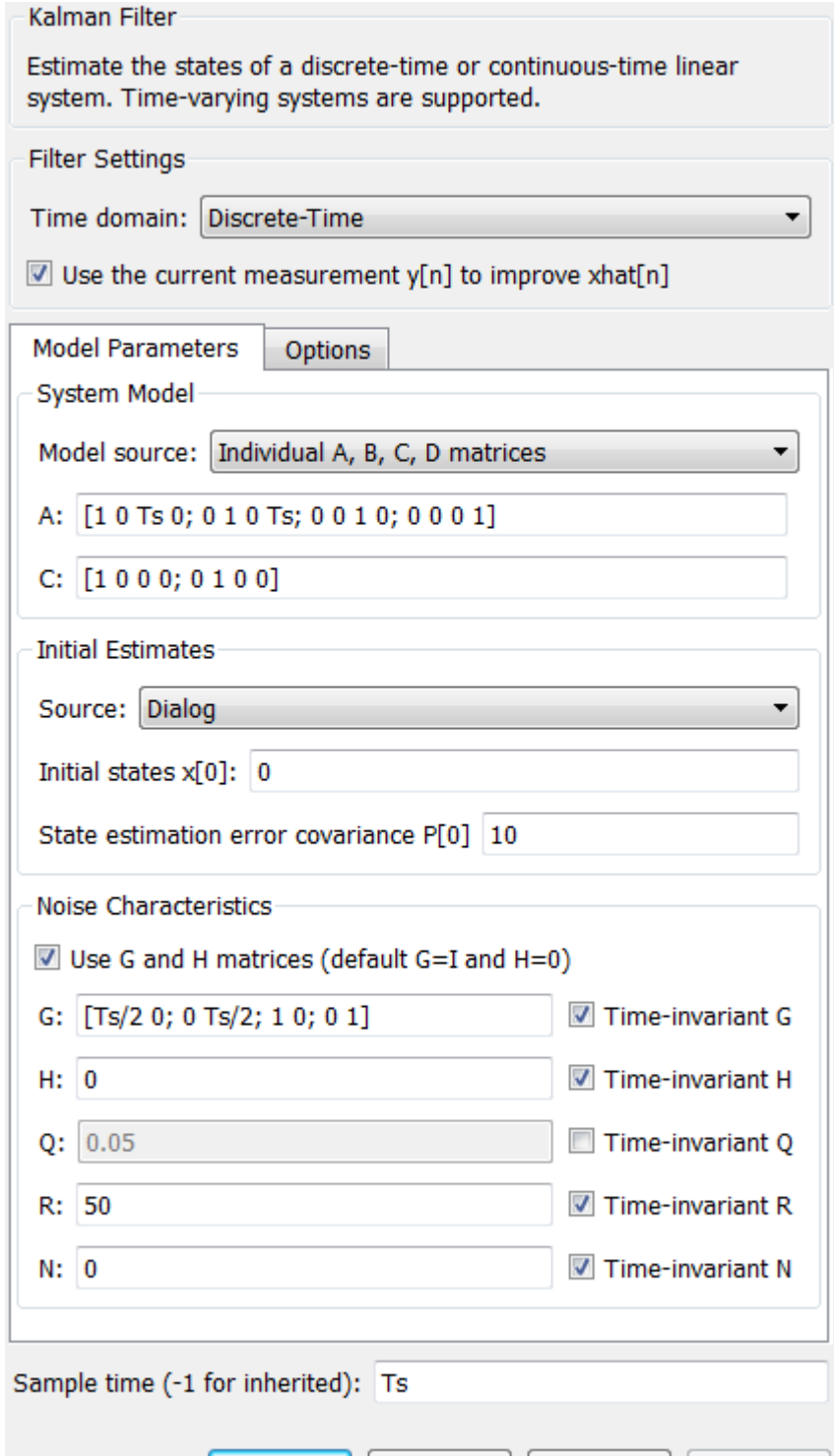
- Select the **Output state estimation error covariance Z** check box. The Z matrix provides information about the filter's confidence in the state estimates.



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 measurments 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. 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

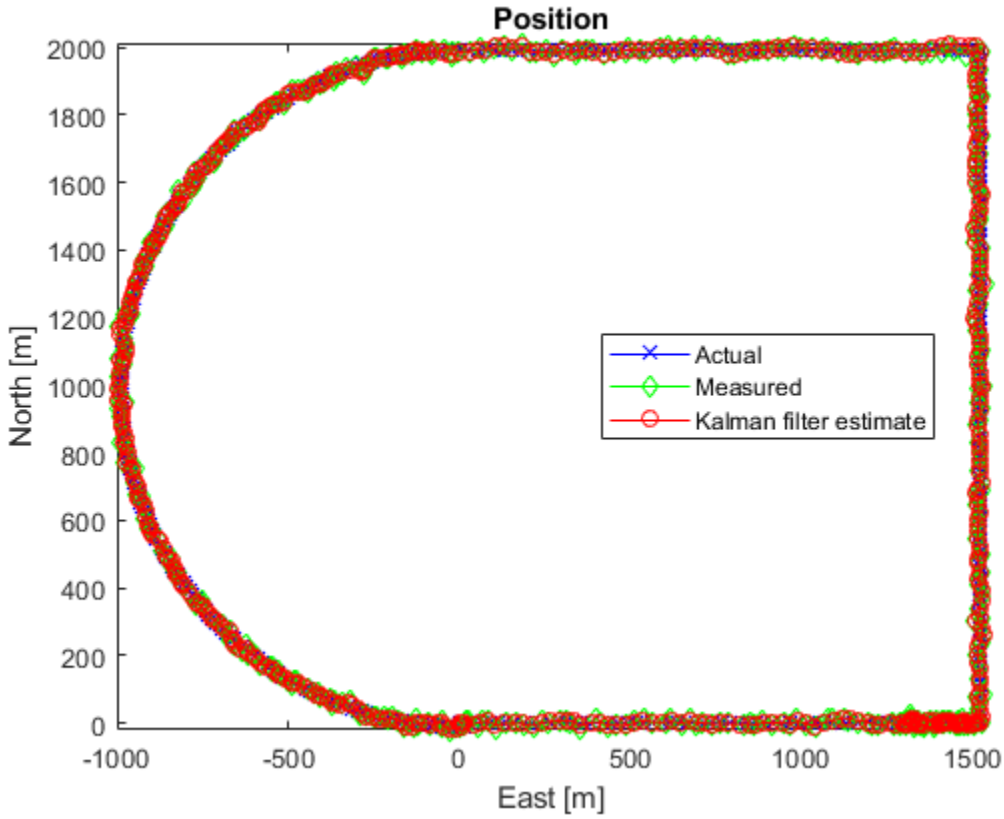
Results

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.

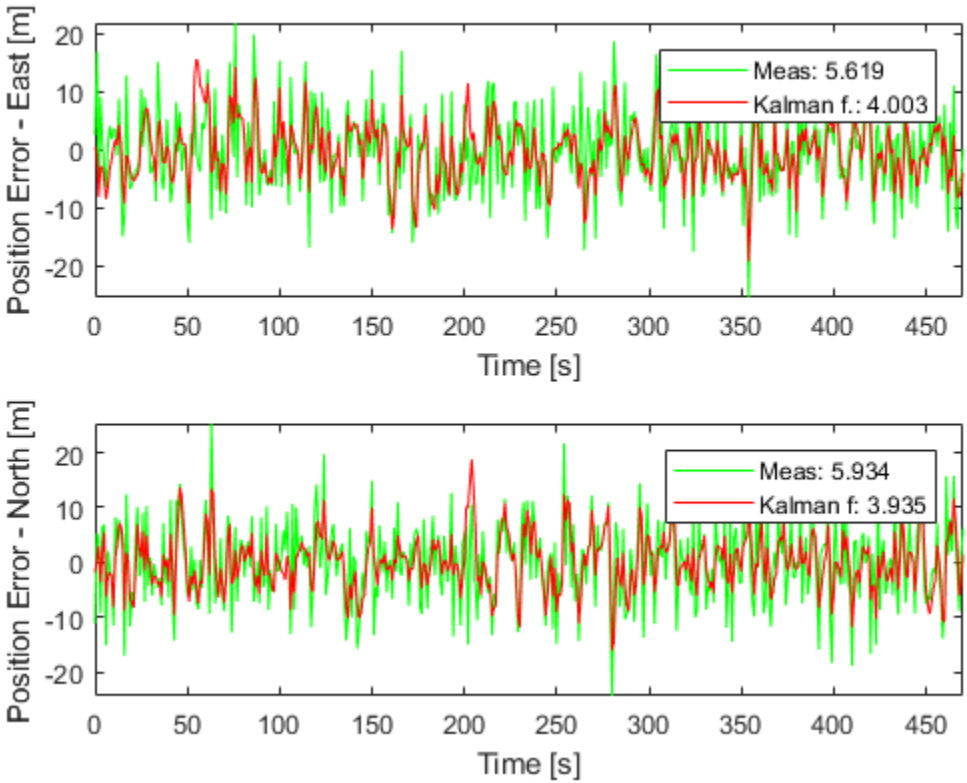
```
sim('ctrlKalmanNavigationExample');

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
```

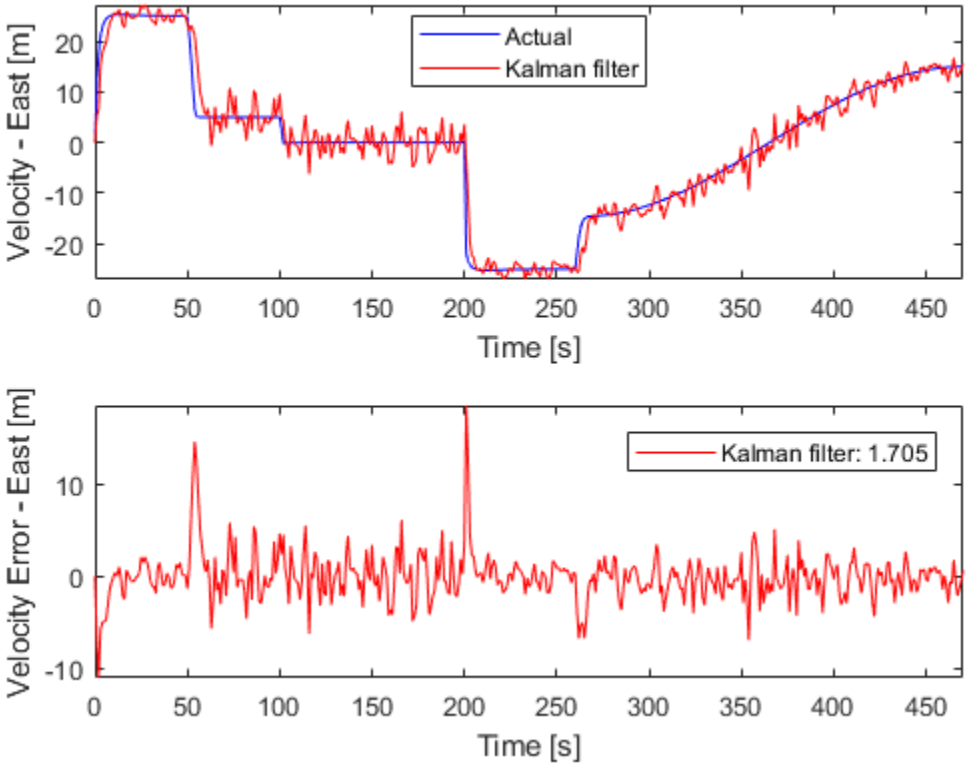
```
subplot(2,1,1);
plot(t,n_xe,'g',t,e_xe,'r');
ylabel('Position Error - East [m]');
xlabel('Time [s]');
legend(sprintf('Meas: %.3f',norm(n_xe,1)/numel(n_xe)),sprintf('Kalman f.: %.3f',norm(e_
axis tight;
% North Position Errors
subplot(2,1,2);
plot(t,y(:,2)-x(:,2),'g',t,xhat(:,2)-x(:,2),'r');
ylabel('Position Error - North [m]');
xlabel('Time [s]');
legend(sprintf('Meas: %.3f',norm(n_xn,1)/numel(n_xn)),sprintf('Kalman f: %.3f',norm(e_x
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]');
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]');
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 decelearation 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');
```

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

System Identification 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;
```

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 x using the `correct` and `predict` commands. Provide generated data to the filter one time step at a time.

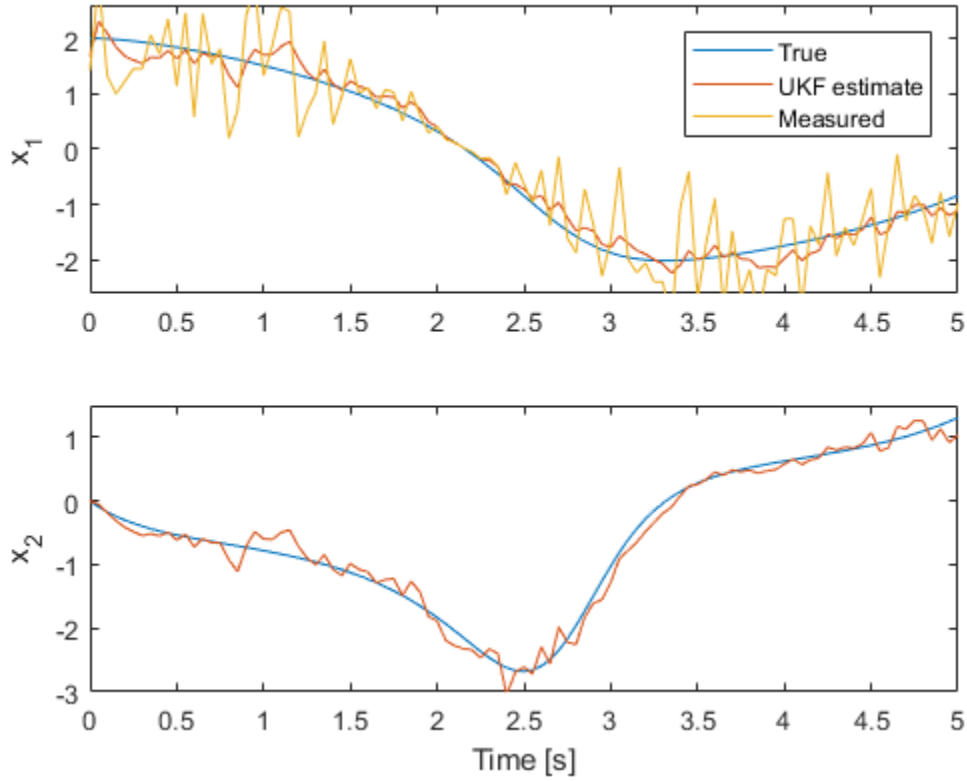
```
for k=1:Nsteps
    % Let k denote the current time.
    %
    % Residuals (or innovations): Measured output - Predicted output
    e(k) = yMeas(k) - vdpMeasurementFcn(ukf.State); % ukf.State is x[k|k-1] at this poi
    % Incorporate the measurements at time k into the state estimates by
```

```
    % 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.

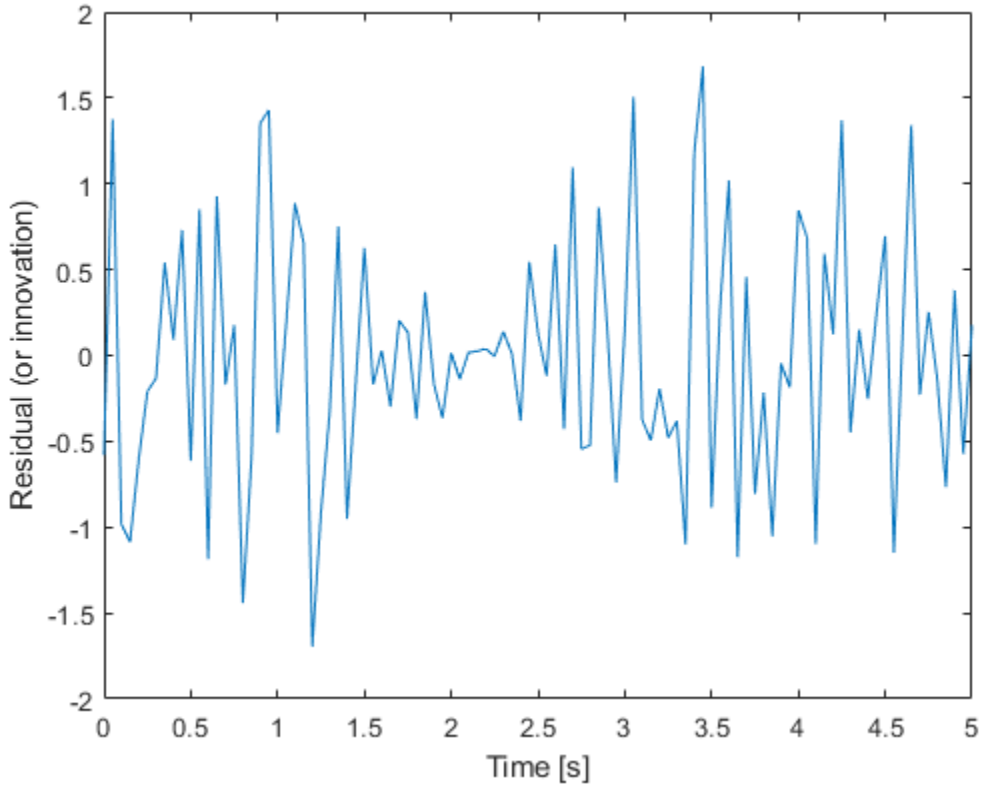
```
figure();
subplot(2,1,1);
plot(timeVector,xTrue(:,1),timeVector,xCorrectedUKF(:,1),timeVector,yMeas(:));
legend('True','UKF estimate','Measured')
ylim([-2.6 2.6]);
ylabel('x_1');
subplot(2,1,2);
plot(timeVector,xTrue(:,2),timeVector,xCorrectedUKF(:,2));
ylim([-3 1.5]);
xlabel('Time [s]');
ylabel('x_2');
```



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 able to produce 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

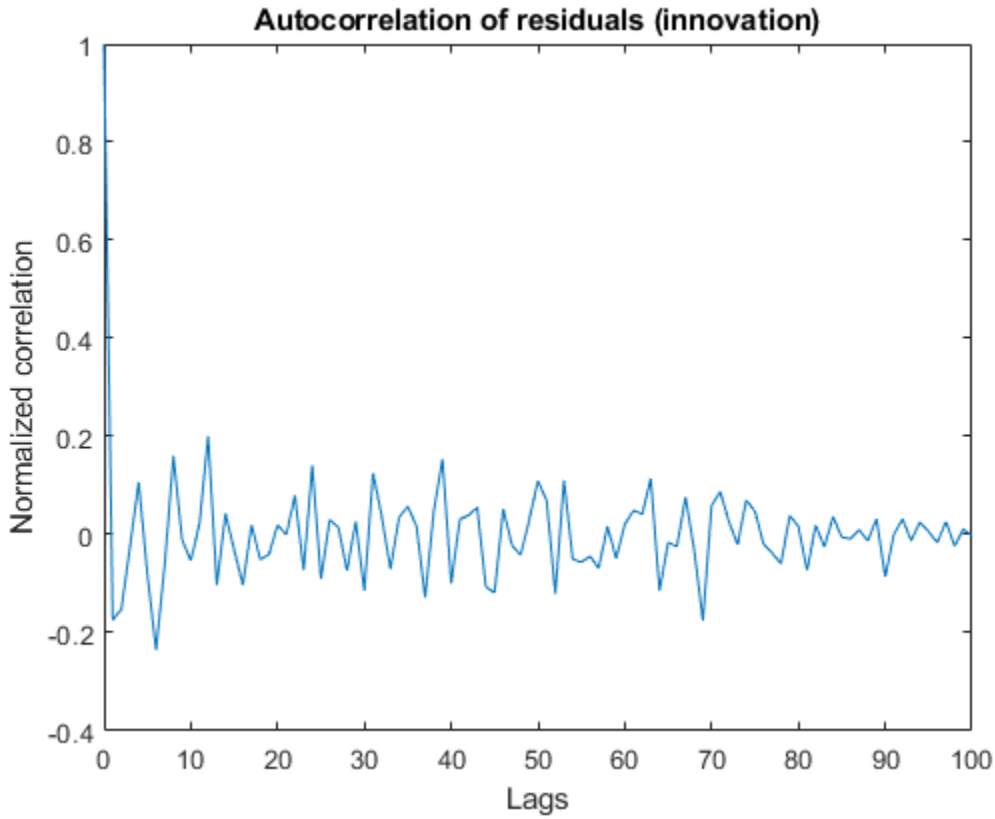
The mean value of the residuals is:

```
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)');
```



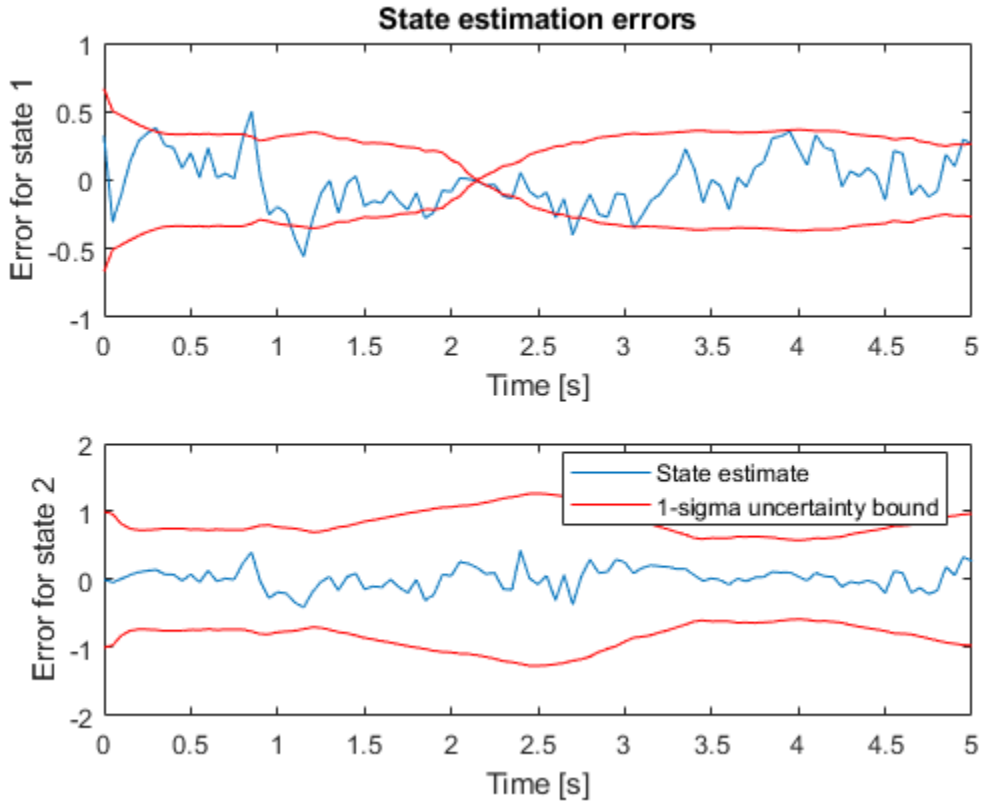
The correlation is small for all lags except 0. The mean correlation is close to zero, and the correlation does not show any significant non-random variations. These characteristics increase confidence in filter performance.

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 =

    0.1386         0
```

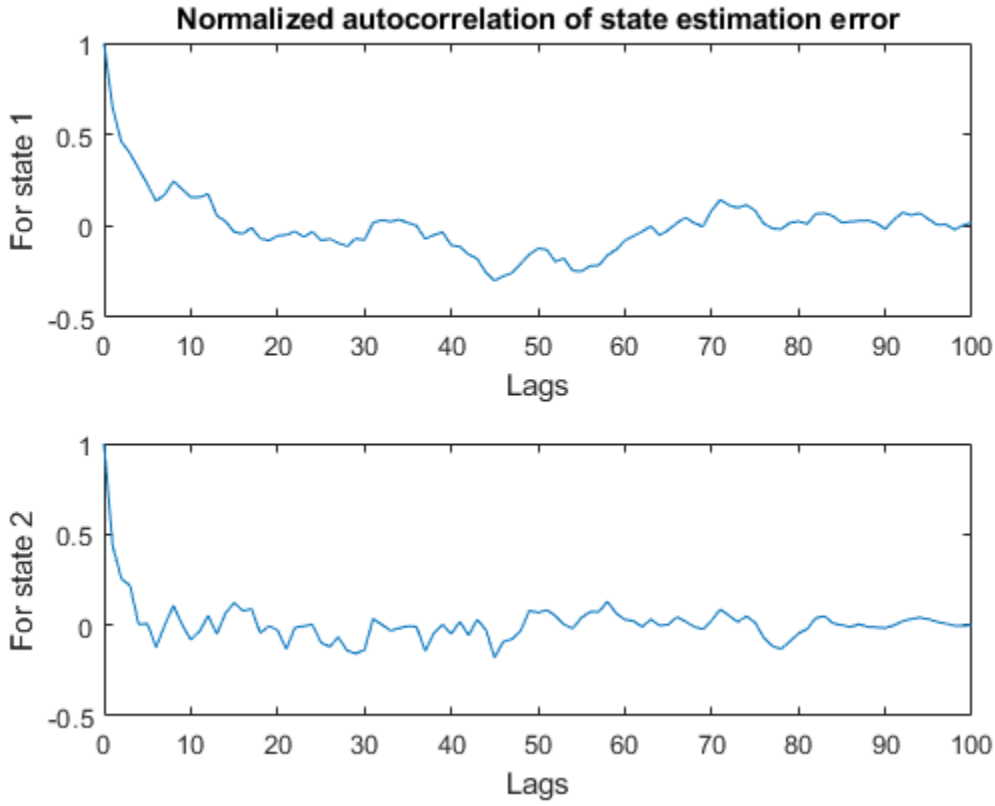
The first state estimation errors exceed the 1σ uncertainty bound approximately 14% of the time steps. Less than 30% of the errors exceeding the 1-sigma uncertainty bound implies good estimation. This criterion is satisfied for both states. The 0% percentage for the second state means that the filter 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 =

   -0.0103    0.0200
```

```
[xeStates1,xeStatesLags1] = xcorr(eStates(:,1),'coeff'); % 'coeff': normalize by the va
[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 System Identification Toolbox implements a discrete-time particle filter algorithm. This section walks you thorugh 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 ProcessNoise property of the unscentedKalmanFilter, just by its covariance. Particle filter 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 the 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) 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 reasmpling 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
```

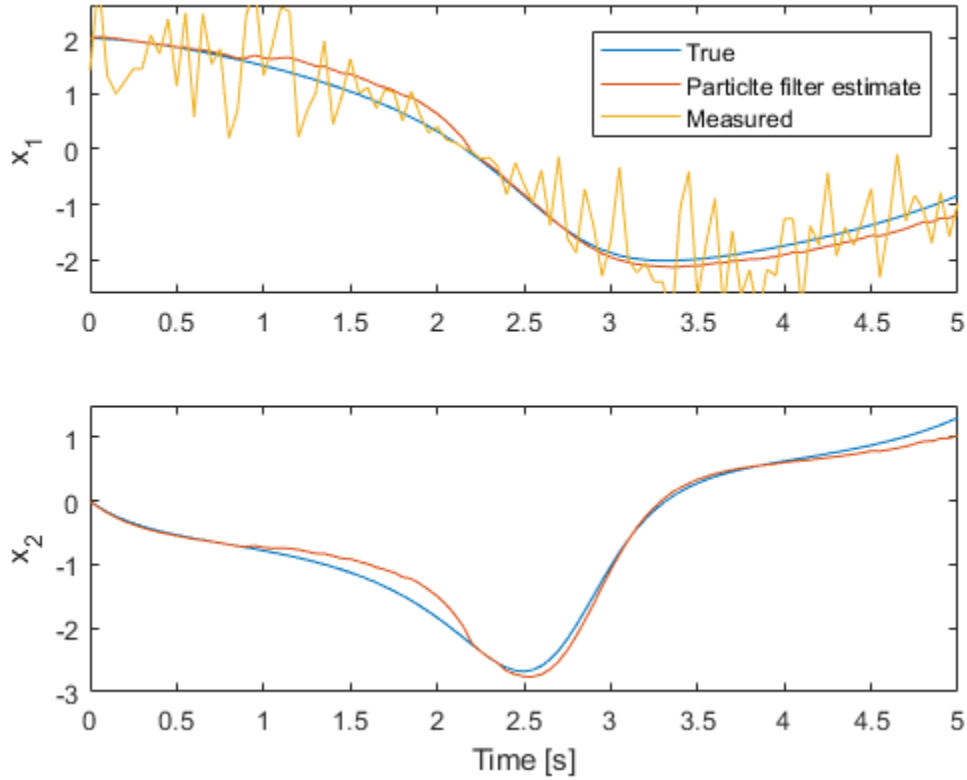
Start the estimation loop. This represents measurements arriving over time, step by step.

```
% 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
    % 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 able to produce 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` | `particleFilter` | `unscentedKalmanFilter`

More About

- “Extended and Unscented Kalman Filter Algorithms for Online State Estimation” on page 16-35
- “Validate Online State Estimation at the Command Line” on page 16-44
- “Troubleshoot Online State Estimation” on page 16-55
- “Generate Code for Online State Estimation in MATLAB” on page 16-51
- “Fault Detection Using an Extended Kalman Filter” on page 16-145

Fault Detection Using an Extended Kalman Filter

This example shows how to use an extended Kalman filter for fault detection. The example uses an extended Kalman filter for online estimation of the friction of a simple DC motor. Significant changes in the estimated friction are detected and indicate a fault.

Motor Model

The motor is modelled as an inertia J with damping coefficient c, driven by a torque u. The motor angular velocity w and acceleration $\dot{w}$, are the measured outputs.
$\dot{w} = (u - cw)/J$

To estimate the damping coefficient c using an extended Kalman filter, introduce an auxiliary state for the damping coefficient and set its derivative to zero.
$\dot{c} = 0$

Thus, the model state, x = [w;c], and measurement, y, equations are:

$$\begin{bmatrix} \dot{w} \\ \dot{c} \end{bmatrix} = \begin{bmatrix} (u - cw)/J \\ 0 \end{bmatrix}$$

$$y = \begin{bmatrix} w \\ (u - cw)/J \end{bmatrix}$$

The continuous-time equations are transformed to discrete time using the approximation $\dot{x} = \frac{x_{n+1} - x_n}{T_s}$, where Ts is the discrete sampling period. This gives the discrete-time model equations which are implemented in the `stateUpdate_MotorModel.m` and `measurement_MotorModel.m` functions.

$$\begin{bmatrix} w_{n+1} \\ c_{n+1} \end{bmatrix} = \begin{bmatrix} w_n + (u_n - c_n w_n) T_s/J \\ c_n \end{bmatrix}$$

$$y_n = \begin{bmatrix} w_n \\ (u_n - c_n w_n)/J \end{bmatrix}$$

Specify motor parameters.

```
J  = 10;    % Inertia
Ts = 0.01;  % Sample time
```

Specify initial states.

```
x0 = [...
    0; ...  % Angular velocity
    1];     % Friction

type stateUpdate_MotorModel
type measurement_MotorModel


function x1 = stateUpdate_MotorModel(x0,varargin)
%STATEUPDATE_MOTORMODEL
%
% State update equations for a motor with friction as a state
%
%  x1 = stateUpdate_MotorModel(x0,u,J,Ts)
%
%  Inputs:
%    x0 - initial state with elements [angular velocity; friction]
%    u  - motor torque input
%    J  - motor inertia
%    Ts - sampling time
%
%  Outputs:
%    x1 - updated states
%

%  Copyright 2016 The MathWorks, Inc.

% Extract data from inputs
u  = varargin{1};   % Input
J  = varargin{2};   % System innertia
Ts = varargin{3};   % Sample time

% State update equation
x1 = [...
    x0(1)+Ts/J*(u-x0(1)*x0(2)); ...
    x0(2)];
end

function y = measurement_MotorModel(x,varargin)
%MEASUREMENT_MOTORMODEL
%
% Measurement equations for a motor with friction as a state
%
%  y = measurement_MotorModel(x0,u,J,Ts)
%
```

```
%  Inputs:
%    x  - motor state with elements [angular velocity; friction]
%    u  - motor torque input
%    J  - motor inertia
%    Ts - sampling time
%
%  Outputs:
%    y - motor measurements with elements [angular velocity; angular acceleration]
%

%  Copyright 2016 The MathWorks, Inc.

% Extract data from inputs
u  = varargin{1};   % Input
J  = varargin{2};   % System innertia

% Output equation
y = [...
    x(1); ...
    (u-x(1)*x(2))/J];
end
```

The motor experiences state (process) noise disturbances, q, and measurement noise disturbances, r. The noise terms are additive.

$$\begin{bmatrix} w_{n+1} \\ c_{n+1} \end{bmatrix} = \begin{bmatrix} w_n + (u_n - c_n w_n) T_s / J \\ c_n \end{bmatrix} + q$$

$$y_n = \begin{bmatrix} w_n \\ (u_n - c_n w_n) / J \end{bmatrix} + r$$

The process and measurement noise have zero mean, `E[q]=E[r]=0`, and covariances `Q = E[qq']` and `R = E[rr']`. The friction state has a high process noise disturbance. This reflects the fact that we expect the friction to vary during normal operation of the motor and want the filter to track this variation. The acceleration and velocity state noise is low but the velocity and acceleration measurements are relatively noisy.

Specify the process noise covariance.

```
Q = [...
    1e-6 0; ...    % Angular velocity
    0 1e-2];       % Friction
```

Specify the measurement noise covariance.

```
R = [...
    1e-4 0; ...  % Velocity measurement
    0 1e-4];     % Acceleration measurement
```

Creating an Extended Kalman Filter

Create an extended Kalman Filter to estimate the states of the model. We are particularly interested in the damping state because dramatic changes in this state value indicate a fault event.

Create an `extendedKalmanFilter` object, and specify the Jacobians of the state transition and measurement functions.

```
ekf = extendedKalmanFilter(...
    @stateUpdate_MotorModel, ...
    @measurement_MotorModel, ...
    x0,...
    'StateCovariance',            [1 0; 0 1000], ...[1 0 0; 0 1 0; 0 0 100], ...
    'ProcessNoise',               Q, ...
    'MeasurementNoise',           R, ...
    'StateTransitionJacobianFcn', @stateJacobian_MotorModel, ...
    'MeasurementJacobianFcn',     @measurementJacobian_MotorModel);
```

The extended Kalman filter has as input arguments the state transition and measurement functions defined previously. The initial state value `x0`, initial state covariance, and process and measurement noise covariances are also inputs to the extended Kalman filter. In this example, the exact Jacobian functions can be derived from the state transition function `f`, and measurement function `h`:

$$\frac{\partial}{\partial x}f = \begin{bmatrix} 1 - T_s c_n/J & -T_s w_n/J \\ 0 & 1 \end{bmatrix}$$

$$\frac{\partial}{\partial x}h = \begin{bmatrix} 1 & 0 \\ -c_n/J & -w_n/J \end{bmatrix}$$

The state Jacobian is defined in the `stateJacobian_MotorModel.m` function and the measurement Jacobian is defined in the `measurementJacobian_MotorModel.m` function.

```
type stateJacobian_MotorModel
type measurementJacobian_MotorModel


function Jac = stateJacobian_MotorModel(x,varargin)
```

```
%STATEJACOBIAN_MOTORMODEL
%
% Jacobian of motor model state equations. See stateUpdate_MotorModel for
% the model equations.
%
%  Jac = stateJacobian_MotorModel(x,u,J,Ts)
%
%  Inputs:
%    x  - state with elements [angular velocity; friction]
%    u  - motor torque input
%    J  - motor inertia
%    Ts - sampling time
%
%  Outputs:
%    Jac - state Jacobian computed at x
%

%  Copyright 2016 The MathWorks, Inc.

% Model properties
J  = varargin{2};
Ts = varargin{3};

% Jacobian
Jac = [...
    1-Ts/J*x(2) -Ts/J*x(1); ...
    0 1];
end

function J = measurementJacobian_MotorModel(x,varargin)
%MEASUREMENT_MOTORMODEL
%
% Jacobian of motor model measurement equations. See measurement_MotorModel for
% the model equations.
%
%  Jac = measurementJacobian_MotorModel(x,u,J,Ts)
%
%  Inputs:
%    x  - state with elements [angular velocity; friction]
%    u  - motor torque input
%    J  - motor inertia
%    Ts - sampling time
%
%  Outputs:
```

```
%    Jac - measurement Jacobian computed at x
%

%  Copyright 2016 The MathWorks, Inc.

% System parameters
J  = varargin{2};   % System innertia

% Jacobian
J = [ ...
    1 0;
    -x(2)/J -x(1)/J];
end
```

Simulation

To simulate the plant, create a loop and introduce a fault in the motor (a dramatic change in the motor fiction). Within the simulation loop, use the extended Kalman filter to estimate the motor states and to specifically track the friction state to detect when there is a statistically significant change in friction.

The motor is simulated with a pulse train that repeatedly accelerates and decelerates the motor. This type of motor operation is typical for a picker robot in a production line.

```
t  = 0:Ts:20;                  % Time, 20s with Ts sampling period
u  = double(mod(t,1)<0.5)-0.5; % Pulse train, period 1, 50% duty cycle
nt = numel(t);                 % Number of time points
nx = size(x0,1);               % Number of states
ySig = zeros([2, nt]);         % Measured motor outputs
xSigTrue = zeros([nx, nt]);    % Unmeasured motor states
xSigEst = zeros([nx, nt]);     % Estimated motor states
xstd = zeros([nx nx nt]);      % Standard deviation of the estimated states
ySigEst = zeros([2, nt]);      % Estimated model outputs
fMean = zeros(1,nt);           % Mean estimated friction
fSTD = zeros(1,nt);            % Standard deviation of estimated friction
fKur = zeros(2,nt);            % Kurtosis of estimated friction
fChanged = false(1,nt);        % Flag indicating friction change detection
```

When simulating the motor, add process and measurement noise similar to the Q and R noise covariance values used when constructing the extended Kalman filter. For the friction, use a much smaller noise value because the friction is mostly constant except when the fault occurs. Artificially induce the fault during the simulation.

```
rng('default');
Qv = chol(Q);   % Standard deviation for process noise
```

```
Qv(end) = 1e-2; % Smaller friction noise
Rv = chol(R);   % Standard deviation for measurement noise
```

Simulate the model using the state update equation, and add process noise to the model states. Ten seconds into the simulation, force a change in the motor friction. Use the model measurement function to simulate the motor sensors, and add measurement noise to the model outputs.

```
for ct = 1:numel(t)

   % Model output update
   y = measurement_MotorModel(x0,u(ct),J,Ts);
   y = y+Rv*randn(2,1);    % Add measurement noise
   ySig(:,ct) = y;

   % Model state update
   xSigTrue(:,ct) = x0;
   x1 = stateUpdate_MotorModel(x0,u(ct),J,Ts);
   % Induce change in friction
   if t(ct) == 10
       x1(2) = 10;  % Step change
   end
   x1n = x1+Qv*randn(nx,1);  % Add process noise
   x1n(2) = max(x1n(2),0.1); % Lower limit on friction
   x0 = x1n; % Store state for next simulation iteration
```

To estimate the motor states from the motor measurements, use the `predict` and `correct` commands of the extended Kalman Filter.

```
   % State estimation using the Extended Kalman Filter
   x_corr = correct(ekf,y,u(ct),J,Ts); % Correct the state estimate based on current me
   xSigEst(:,ct) = x_corr;
   xstd(:,:,ct) = chol(ekf.StateCovariance);
   predict(ekf,u(ct),J,Ts);            % Predict next state given the current state and
```

To detect changes in friction, compute the estimated friction mean and standard deviation using a 4 second moving window. After an initial 7-second period, lock the computed mean and standard deviation. This initially computed mean is the expected no-fault mean value for the friction. After 7 seconds, if the estimated friction is greater than 3 standard deviations away from the expected no-fault mean value, it signifies a significant change in the friction. To reduce the effect of noise and variability in the estimated friction, use the mean of the estimated friction when comparing to the 3-standard-deviations bound.

```
    if t(ct) < 7
        % Compute mean and standard deviation of estimated fiction.
        idx = max(1,ct-400):max(1,ct-1); % Ts = 0.01 seconds
        fMean(ct) = mean( xSigEst(2, idx) );
        fSTD(ct)  = std( xSigEst(2, idx) );
    else
        % Store the computed mean and standard deviation without
        % recomputing.
        fMean(ct) = fMean(ct-1);
        fSTD(ct)  = fSTD(ct-1);
        % Use the expected friction mean and standard deviation to detect
        % friction changes.
        estFriction = mean(xSigEst(2,max(1,ct-10):ct));
        fChanged(ct) = (estFriction > fMean(ct)+3*fSTD(ct)) || (estFriction < fMean(ct)-
    end
    if fChanged(ct) && ~fChanged(ct-1)
        % Detect a rising edge in the friction change signal |fChanged|.
        fprintf('Significant friction change at %f\n',t(ct));
    end
```

```
Significant friction change at 10.450000
```

Use the estimated state to compute the estimated output. Compute the error between the measured and estimated outputs, and calculate the error statistics. The error statistics can be used for detecting the friction change. This is discussed in more detail later.

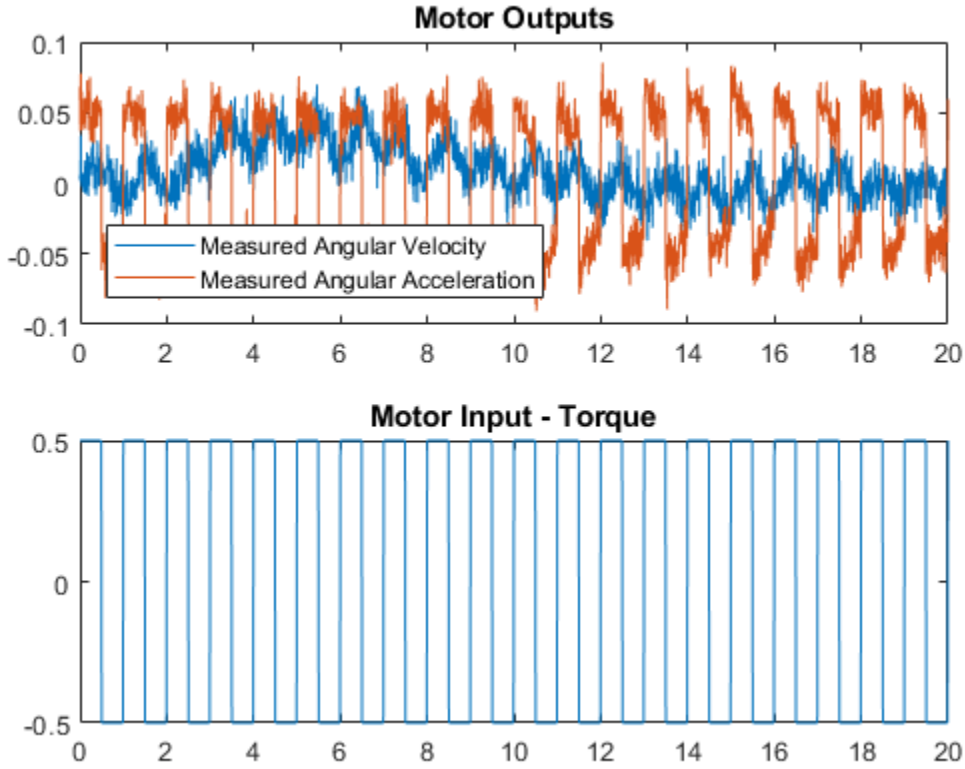
```
    ySigEst(:,ct) = measurement_MotorModel(x_corr,u(ct),J,Ts);
    idx = max(1,ct-400):ct;
    fKur(:,ct) = [...
        kurtosis(ySigEst(1,idx)-ySig(1,idx)); ...
        kurtosis(ySigEst(2,idx)-ySig(2,idx))];

end
```

Extended Kalman Filter Performance

Note that a friction change was detected at 10.45 seconds. We now describe how this fault-detection rule was derived. First examine the simulation results and filter performance.

```
figure,
subplot(211), plot(t,ySig(1,:),t,ySig(2,:));
title('Motor Outputs')
legend('Measured Angular Velocity','Measured Angular Acceleration', 'Location','SouthWe
```

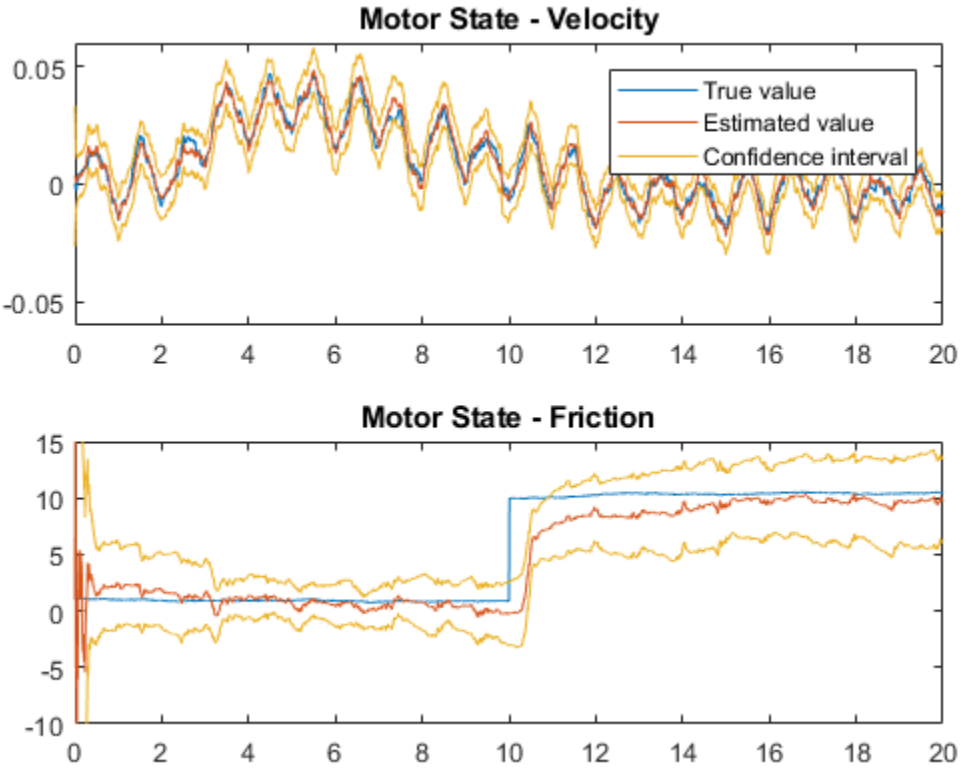
```
subplot(212), plot(t,u);
title('Motor Input - Torque')
```



The model input-output responses indicate that it is difficult to detect the friction change directly from the measured signals. The extended Kalman filter enables us to estimate the states, in particular the friction state. Compare the true model states and estimated states. The estimated states are shown with confidence intervals corresponding to 3 standard deviations.

```
figure,
subplot(211),plot(t,xSigTrue(1,:), t,xSigEst(1,:), ...
    [t nan t],[xSigEst(1,:)+3*squeeze(xstd(1,1,:))', nan, xSigEst(1,:)-3*squeeze(xstd(1
axis([0 20 -0.06 0.06]),
legend('True value','Estimated value','Confidence interval')
```

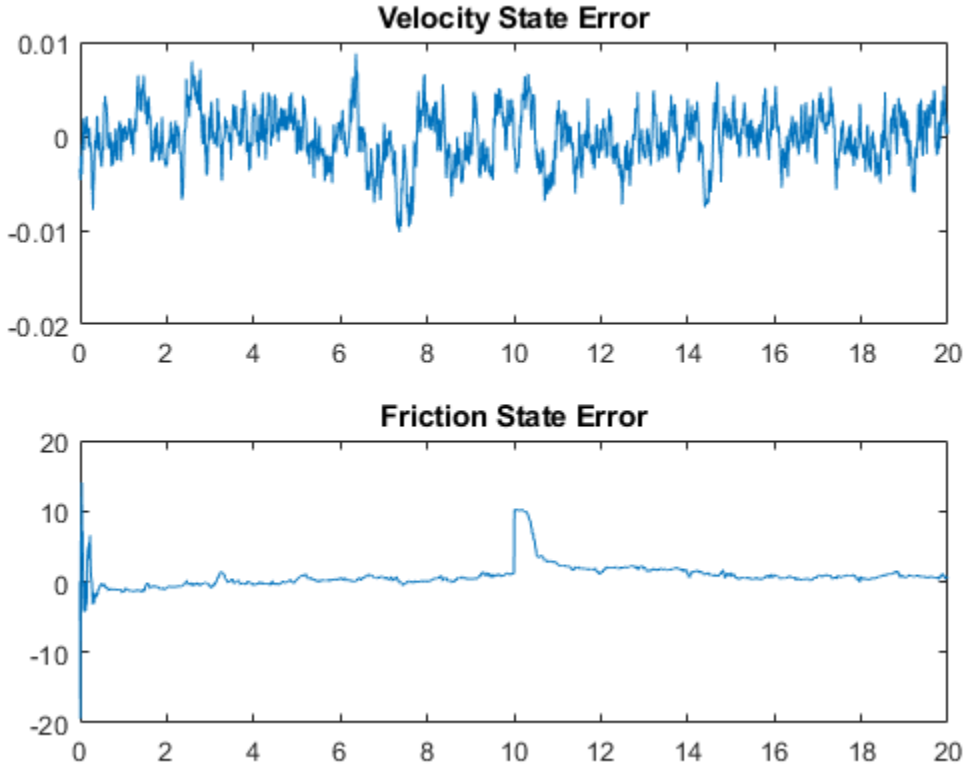
```
title('Motor State - Velocity')
subplot(212),plot(t,xSigTrue(2,:), t,xSigEst(2,:),  ...
    [t nan t],[xSigEst(2,:)+3*squeeze(xstd(2,2,:))' nan xSigEst(2,:)-3*squeeze(xstd(2,2
axis([0 20 -10 15])
title('Motor State - Friction');
```



Note that the filter estimate tracks the true values, and that the confidence intervals remain bounded. Examining the estimation errors provide more insight into the filter behavior.

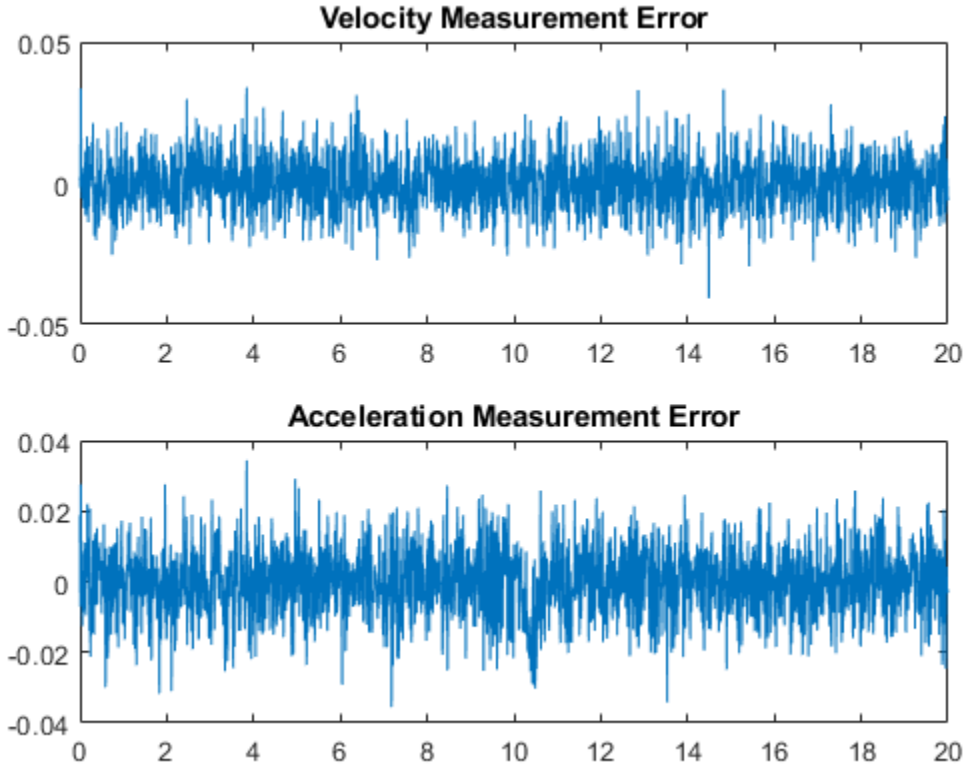
```
figure,
subplot(211),plot(t,xSigTrue(1,:)-xSigEst(1,:))
title('Velocity State Error')
```

```
subplot(212),plot(t,xSigTrue(2,:)-xSigEst(2,:))
title('Friction State Error')
```



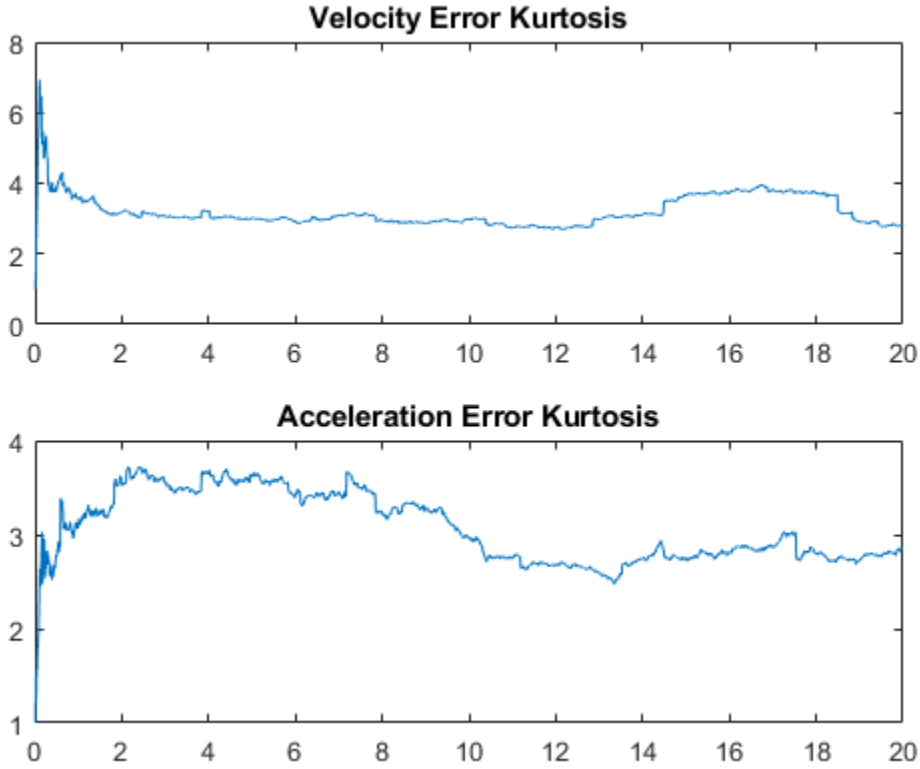
The error plots show that the filter adapts after the friction change at 10 seconds and reduces the estimation errors to zero. However, the error plots cannot be used for fault detection as they rely on knowing the true states. Comparing the measured state value to the estimated state values for acceleration and velocity could provide a detection mechanism.

```
figure
subplot(211), plot(t,ySig(1,:)-ySigEst(1,:))
title('Velocity Measurement Error')
subplot(212),plot(t,ySig(2,:)-ySigEst(2,:))
title('Acceleration Measurement Error')
```



The acceleration error plot shows a minor difference in mean error around 10 seconds when the fault is introduced. View the error statistics to see if the fault can be detected from the computed errors. The acceleration and velocity errors are expected to be normally distributed (the noise models are all Gaussian). Therefore, the kurtosis of the acceleration error may help identify when the error distribution change from symmetrical to asymmetrical due to the friction change and resulting change in error distribution.

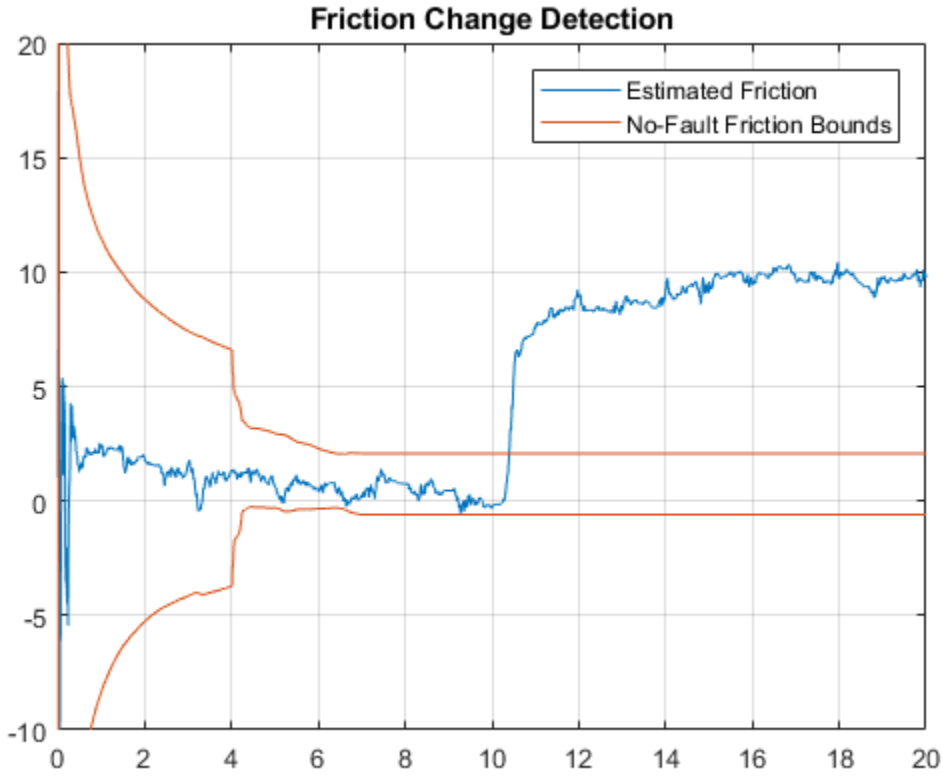
```
figure,
subplot(211),plot(t,fKur(1,:))
title('Velocity Error Kurtosis')
subplot(212),plot(t,fKur(2,:))
title('Acceleration Error Kurtosis')
```



Ignoring the first 4 seconds when the estimator is still converging and data is being collected, the kurtosis of the errors is relatively constant with minor variations around 3 (the expected kurtosis value for a Gaussian distribution). Thus, the error statistics cannot be used to automatically detect friction changes in this application. Using the kurtosis of the errors is also difficult in this application as the filter is adapting and continually driving the errors to zero, only giving a short time window where the error distributions differ from zero.

Thus in this application, using the changes in estimated friction provide the best way to automatically detect faults in the motor. The friction estimates (mean and standard deviation) from known no-fault data provide expected bounds for the friction and it is easy to detect when these bounds are violated. The following plot highlights this fault-detection approach.

```
figure
plot(t,xSigEst(2,:),[t nan t],[fMean+3*fSTD,nan,fMean-3*fSTD])
title('Friction Change Detection')
legend('Estimated Friction','No-Fault Friction Bounds')
axis([0 20 -10 20])
grid on
```



Summary

This example has shown how to use an extended Kalman filter to estimate the friction in a simple DC motor and use the friction estimate for fault detection.

See Also

`extendedKalmanFilter` | `unscentedKalmanFilter`

More About

- "Extended and Unscented Kalman Filter Algorithms for Online State Estimation" on page 16-35
- "Validate Online State Estimation at the Command Line" on page 16-44
- "Troubleshoot Online State Estimation" on page 16-55
- "Generate Code for Online State Estimation in MATLAB" on page 16-51
- "Nonlinear State Estimation Using Unscented Kalman Filter and Particle Filter" on page 16-124

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 System Identification Toolbox™ is used to estimate the position and velocity of an object using GPS and radar measurements.

Introduction

The toolbox has two Simulink blocks for nonlinear state estimation:

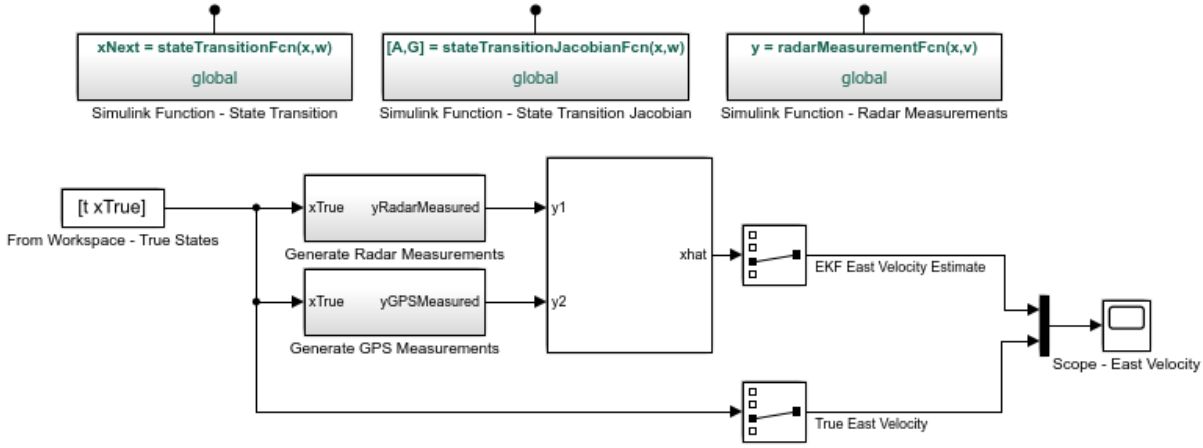
- Extended Kalman Filter: Implements the first-order discrete-time extended Kalman filter algorithm.
- Unscented Kalman Filter: Implements the discrete-time unscented Kalman filter algorithm.

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 Extended or Unscented Kalman Filter 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.

```
addpath(fullfile(matlabroot,'examples','ident_featured','main')) % add example data
open_system('multirateEKFExample');
```



Copyright 2016-2017 The MathWorks, Inc.

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, 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:

$$\hat{x}[k] = \begin{bmatrix} \hat{x}_e[k] \\ \hat{x}_n[k] \\ \hat{v}_e[k] \\ \hat{v}_n[k] \end{bmatrix} \begin{matrix} \text{East position estimate } [m] \\ \text{North position estimate } [m] \\ \text{East velocity estimate } [m/s] \\ \text{North velocity estimate } [m/s] \end{matrix}$$

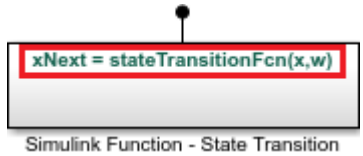
Here k is the discrete-time index. The state transition equation used is of the nonadditive form $\hat{x}[k+1] = A\hat{x}[k] + Gw[k]$, where $\hat{x}$ is the state vector, and w is the process noise. The filter assumes that w is a zero-mean, independent random variable with known variance $E[ww^T]$. The A and G matrices are:

$$A = \begin{bmatrix} 1 & 0 & T_s & 0 \\ 0 & 1 & 0 & T_s \\ 0 & 0 & 1 & 0 \\ 0 & 0 & 0 & 1 \end{bmatrix} \qquad G = \begin{bmatrix} T_s/2 & 0 \\ 0 & T_s/2 \\ 1 & 0 \\ 0 & 1 \end{bmatrix}$$

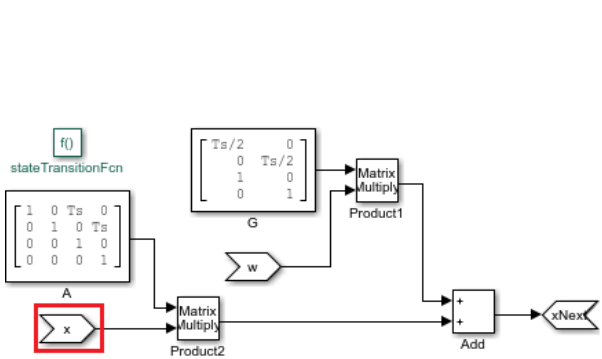
where T_s is the sample time. The third row of A and G model the east velocity as a random walk: $\hat{v}_e[k+1] = \hat{v}_e[k] + w_1[k]$. In reality, position is a continuous-time variable and is the integral of velocity over time $\frac{d}{dt}\hat{x}_e = \hat{v}_e$. The first row of A and G represent 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.

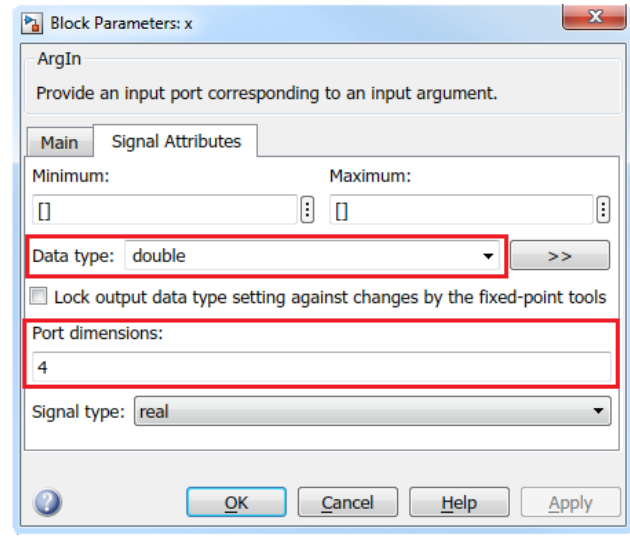
In this example you implement the state transition function using a Simulink Function block. To do so,

- 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).



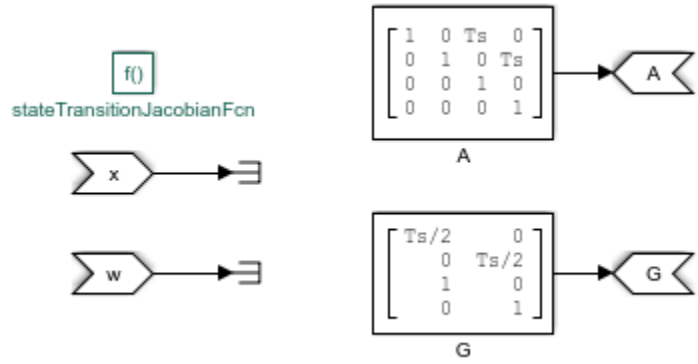
- Though it is not required in this example, you can use any signals from the rest of your Simulink model in the Simulink Function. To do so, add `Inport` blocks from the `Simulink/Sources` library. Note that these are different than the `ArgIn` and `ArgOut` blocks that are set through the signature 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/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: $y[k] = h(x[k], v[k], u[k])$

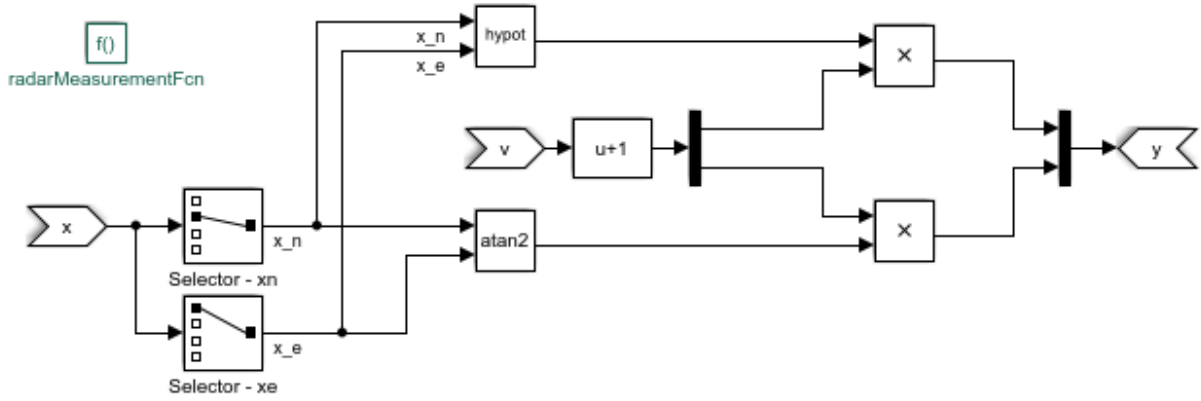
Here h(..) is the measurement function, and v is the measurement noise. u is optional, and represents additional inputs to h, for instance system inputs or parameters. These inputs can differ from the inputs in the state transition function.

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.

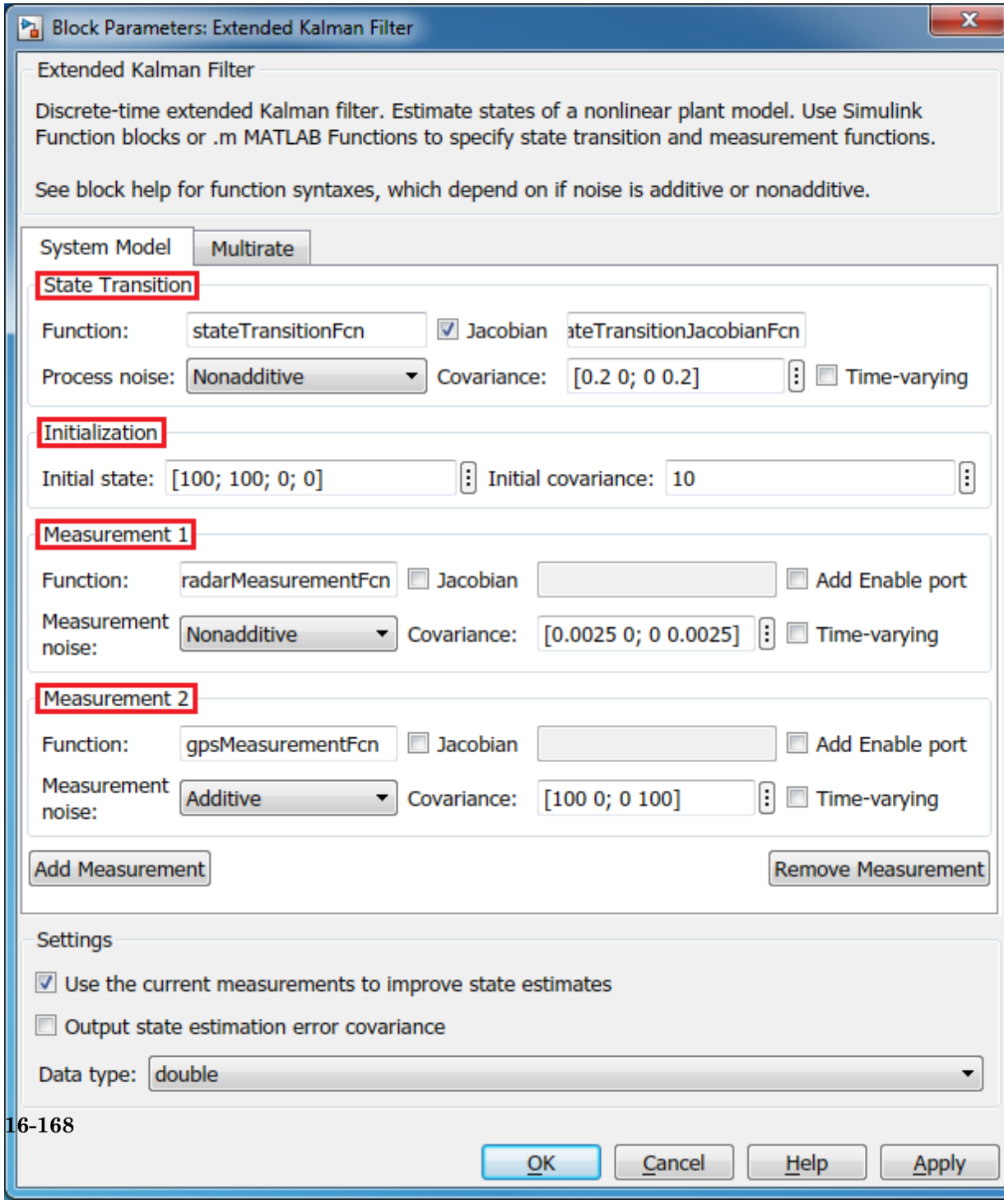
Since there are two sensors, click **Add Measurement** to specify a second measurement function.

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

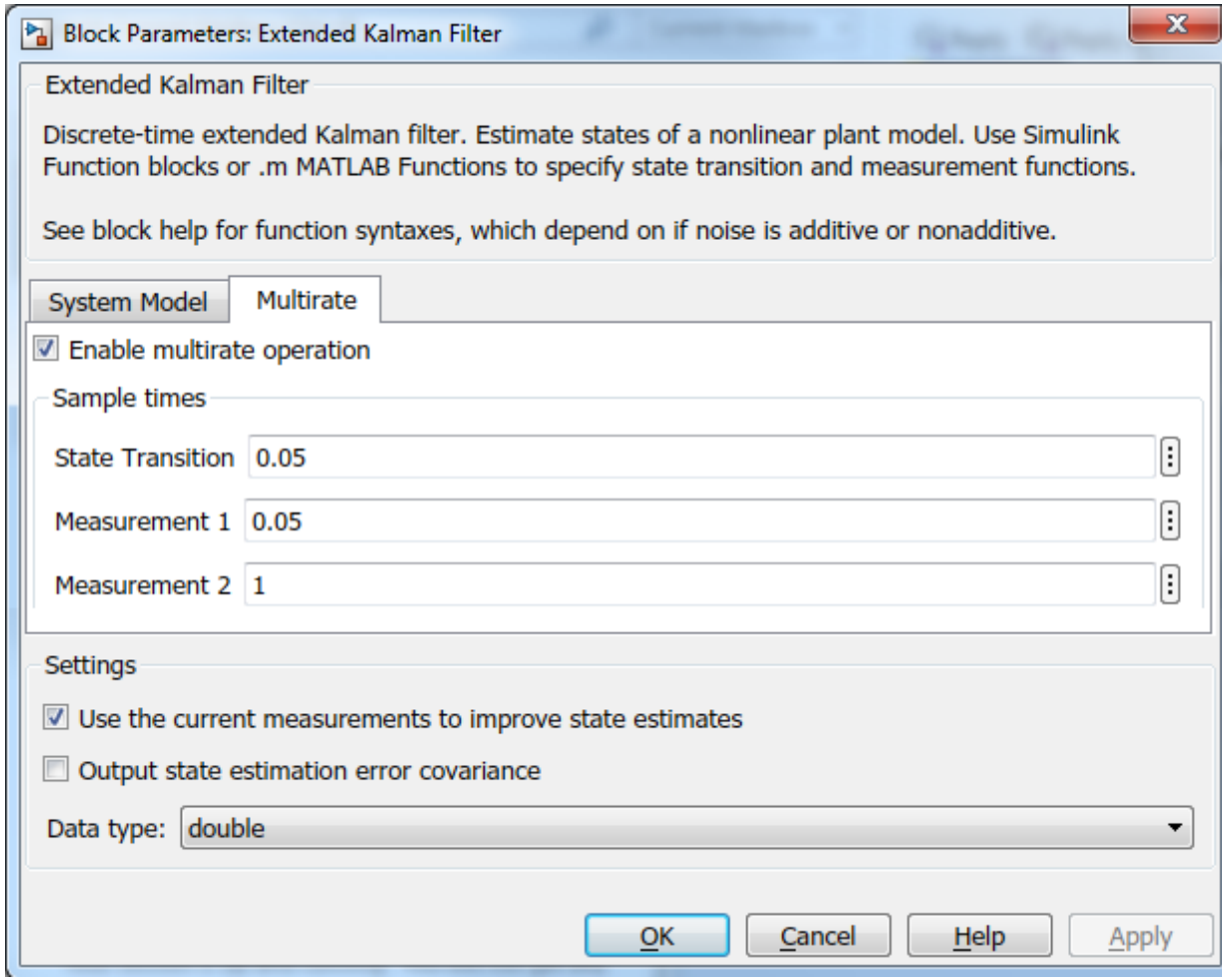
1. Specify the name of your measurement function, `gpsMeasurementFcn`, in **Function**.
2. This sensor model has additive noise. Therefore, specify the GPS measurement noise as `Additive` in the **Measurement Noise** drop-down list.
3. Specify the measurement noise covariance as [100 0; 0 100].


Block Parameters: Extended Kalman Filter
Extended Kalman Filter
Discrete-time extended Kalman filter. Estimate states of a nonlinear plant model. Use Simulink Function blocks or .m MATLAB Functions to specify state transition and measurement functions.
See block help for function syntaxes, which depend on if noise is additive or nonadditive.
System Model
Multirate
State Transition
Function: stateTransitionFcn
Jacobian
ateTransitionJacobianFcn
Process noise: Nonadditive
Covariance: [0.2 0; 0 0.2]
Time-varying
Initialization
Initial state: [100; 100; 0; 0]
Initial covariance: 10
Measurement 1
Function: radarMeasurementFcn
Jacobian
Add Enable port
Measurement noise: Nonadditive
Covariance: [0.0025 0; 0 0.0025]
Time-varying
Measurement 2
Function: gpsMeasurementFcn
Jacobian
Add Enable port
Measurement noise: Additive
Covariance: [100 0; 0 100]
Time-varying
Add Measurement
Remove Measurement
Settings
Use the current measurements to improve state estimates
Output state estimation error covariance
Data type: double
OK
Cancel
Help
Apply

16-168

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 $x_e(0) = 100$ [m], $x_n(0) = 100$ [m]
- It heads north at $\dot{x}_n = 50$ [m/s] until t = 20 seconds.

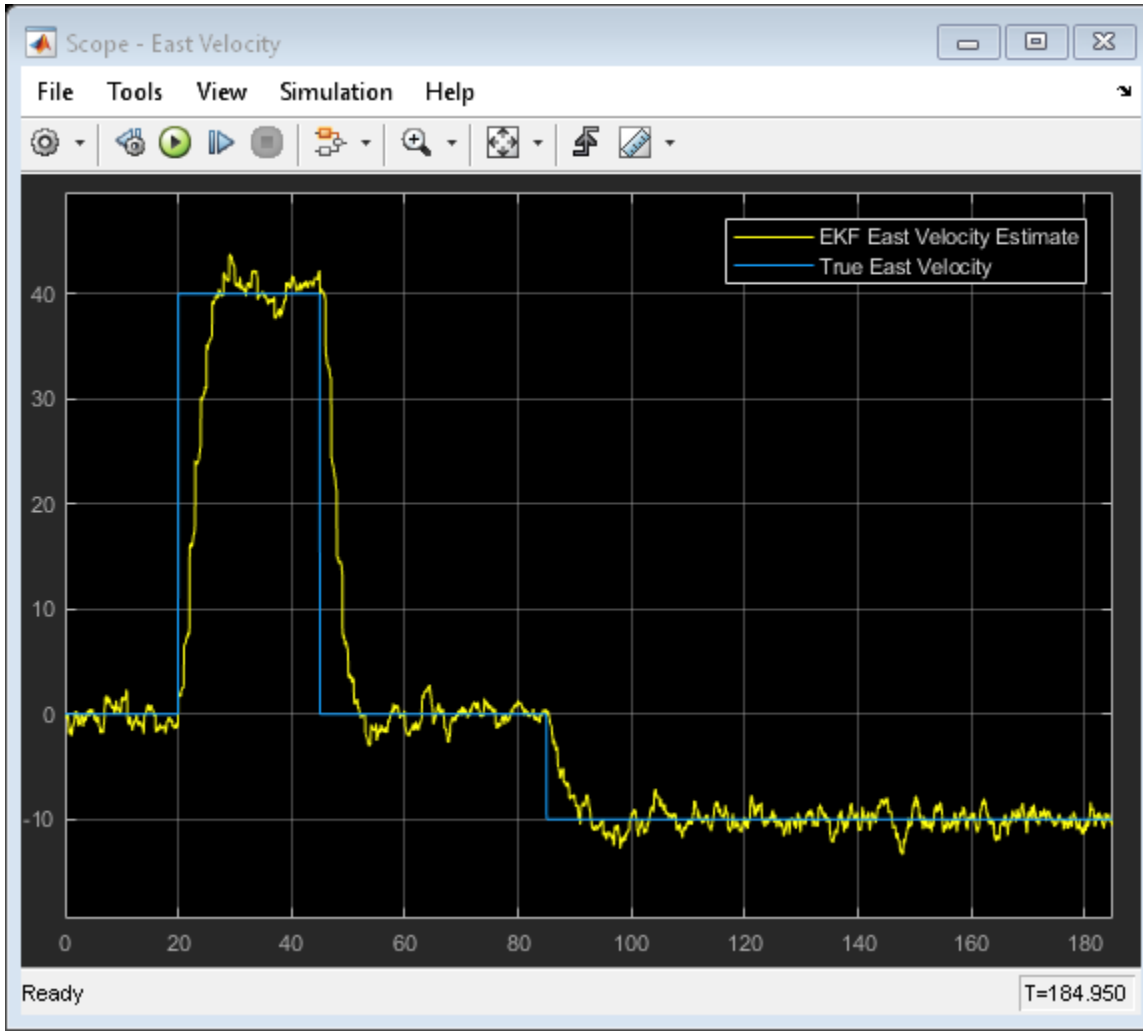
- It heads east at $\dot{x}_n = 40\ [\mathrm{m/s}]$ between t = 20 and t = 45 seconds.
- It heads south at $\dot{x}_n = -25\ [\mathrm{m/s}]$ between t = 45 and t = 85 seconds.
- It heads west at $\dot{x}_e = -10\ [\mathrm{m/s}]$ between t = 85 and t = 185 seconds.

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 0-18
```

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 16-47.
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);
rmpath(fullfile(matlabroot,'examples','ident_featured','main')) % remove example data
```

See Also

Extended Kalman Filter | Unscented Kalman Filter

More About

- “Extended and Unscented Kalman Filter Algorithms for Online State Estimation” on page 16-35
- “Validate Online State Estimation in Simulink” on page 16-47
- “Troubleshoot Online State Estimation” on page 16-55

Model Analysis

- "Validating Models After Estimation" on page 17-3
- "Supported Model Plots" on page 17-5
- "Plot Models in the System Identification App" on page 17-7
- "Simulating and Predicting Model Output" on page 17-9
- "Simulation and Prediction in the App" on page 17-12
- "Simulation and Prediction at the Command Line" on page 17-17
- "Compare Simulated Output with Measured Data" on page 17-22
- "Perform Multivariate Time Series Forecasting" on page 17-25
- "What Is Residual Analysis?" on page 17-43
- "How to Plot Residuals in the App" on page 17-47
- "How to Plot Residuals at the Command Line" on page 17-50
- "Examine Model Residuals" on page 17-51
- "Impulse and Step Response Plots" on page 17-55
- "Plot Impulse and Step Response Using the System Identification App" on page 17-59
- "Plot Impulse and Step Response at the Command Line" on page 17-62
- "Frequency Response Plots" on page 17-64
- "Plot Bode Plots Using the System Identification App" on page 17-68
- "Plot Bode and Nyquist Plots at the Command Line" on page 17-71
- "Noise Spectrum Plots" on page 17-73
- "Plot the Noise Spectrum Using the System Identification App" on page 17-76
- "Plot the Noise Spectrum at the Command Line" on page 17-79
- "Pole and Zero Plots" on page 17-81
- "Reducing Model Order Using Pole-Zero Plots" on page 17-84
- "Model Poles and Zeros Using the System Identification App" on page 17-85
- "Plot Poles and Zeros at the Command Line" on page 17-87

- "Analyzing MIMO Models" on page 17-88
- "Customize Response Plots Using the Response Plots Property Editor" on page 17-94
- "Computing Model Uncertainty" on page 17-116
- "Troubleshooting Model Estimation" on page 17-119
- "Next Steps After Getting an Accurate Model" on page 17-124

Validating Models After Estimation

In this section...
“Ways to Validate Models” on page 17-3
“Data for Model Validation” on page 17-4

Ways to Validate Models

You can use the following approaches to validate models:

- Comparing simulated or predicted model output to measured output.

  See “Simulating and Predicting Model Output” on page 17-9.

  To simulate identified models in the Simulink environment, see “Simulating Identified Model Output in Simulink” on page 20-5.
- Analyzing autocorrelation and cross-correlation of the residuals with input.

  See “What Is Residual Analysis?” on page 17-43.
- Analyzing model response. For more information, see the following:
  - “Impulse and Step Response Plots” on page 17-55
  - “Frequency Response Plots” on page 17-64

  For information about the response of the noise model, see “Noise Spectrum Plots” on page 17-73.
- Plotting the poles and zeros of the linear parametric model.

  For more information, see “Pole and Zero Plots” on page 17-81.
- Comparing the response of nonparametric models, such as impulse-, step-, and frequency-response models, to parametric models, such as linear polynomial models, state-space model, and nonlinear parametric models.

  Note Do not use this comparison when feedback is present in the system because feedback makes nonparametric models unreliable. To test if feedback is present in the system, use the `advice` command on the data.

- Compare models using Akaike Information Criterion or Akaike Final Prediction Error.

  For more information, see the `aic` and `fpe` reference page.
- Plotting linear and nonlinear blocks of Hammerstein-Wiener and nonlinear ARX models.

Displaying confidence intervals on supported plots helps you assess the uncertainty of model parameters. For more information, see “Computing Model Uncertainty” on page 17-116.

Data for Model Validation

For plots that compare model response to measured response and perform residual analysis, you designate two types of data sets: one for estimating the models (*estimation data*), and the other for validating the models (*validation data*). Although you can designate the same data set to be used for estimating and validating the model, you risk over-fitting your data. When you validate a model using an independent data set, this process is called *cross-validation*.

Note Validation data should be the same in frequency content as the estimation data. If you detrended the estimation data, you must remove the same trend from the validation data. For more information about detrending, see “Handling Offsets and Trends in Data” on page 2-114.

Supported Model Plots

The following table summarizes the types of supported model plots.

Plot Type	Supported Models	Learn More
Model Output	All linear and nonlinear models	"Simulating and Predicting Model Output" on page 17-9
Residual Analysis	All linear and nonlinear models	"What Is Residual Analysis?" on page 17-43
Transient Response	• All linear parametric models • Correlation analysis (nonparametric) models • For nonlinear models, only step response.	"Impulse and Step Response Plots" on page 17-55
Frequency Response	All linear models	"Frequency Response Plots" on page 17-64
Noise Spectrum	• All linear parametric models • Spectral analysis (nonparametric) models	"Noise Spectrum Plots" on page 17-73
Poles and Zeros	All linear parametric models	"Pole and Zero Plots" on page 17-81
Nonlinear ARX	Nonlinear ARX models only	Nonlinear ARX Plots on page 11-50
Hammerstein-Wiener	Hammerstein-Wiener models only	Hammerstein-Wiener Plots on page 12-28

See Also

Related Examples

- "Plot Models in the System Identification App" on page 17-7

- “Compare Simulated Output with Measured Data” on page 17-22

More About

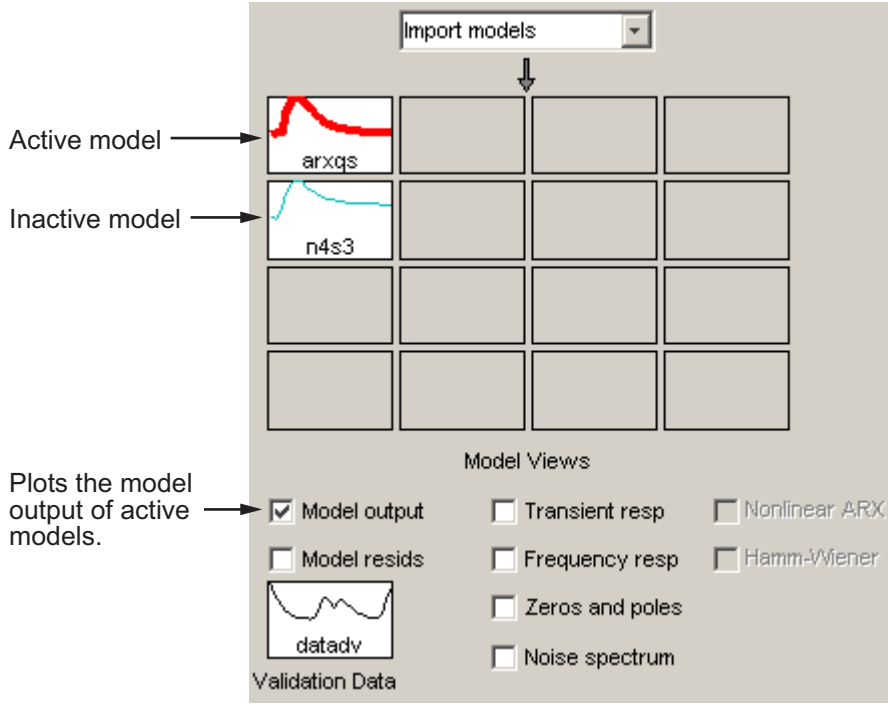
- “Validating Models After Estimation” on page 17-3

Plot Models in the System Identification App

To create one or more plots of your models, select the corresponding check box in the **Model Views** area of the System Identification app. An *active* model icon has a thick line in the icon, while an *inactive* model has a thin line. Only active models appear on the selected plots.

To include or exclude a model on a plot, click the corresponding icon in the System Identification app. Clicking the model icon updates any plots that are currently open.

For example, in the following figure, **Model output** is selected. In this case, the models `n4s3` is not included on the plot because only `arxqs` is active.



Plots Include Only Active Models

To close a plot, clear the corresponding check box in the System Identification app.

Tip To get information about a specific plot, select a help topic from the **Help** menu in the plot window.

See Also

Related Examples

- “Interpret the Model Output Plot” on page 17-12
- “Change Model Output Plot Settings” on page 17-14
- “Working with Plots” on page 21-11
- “Compare Simulated Output with Measured Data” on page 17-22

Simulating and Predicting Model Output

Why Simulate or Predict Model Output?

You primarily use a model is to simulate its output, i.e., calculate the output ($y(t)$) for given input values. You can also predict model output, i.e., compute a qualified guess of future output values based on past observations of system's inputs and outputs.

You also validate linear parametric models and nonlinear models by checking how well the simulated or predicted output of the model matches the measured output. You can use either time or frequency domain data for simulation or prediction. For frequency domain data, the simulation and prediction results are products of the Fourier transform of the input and frequency function of the model.

Simulation provides a better validation test for the model than prediction. However, how you validate the model output should match how you plan to use the model. For example, if you plan to use your model for control design, you can validate the model by predicting its response over a time horizon that represents the dominating time constants of the model.

What are Simulation and Prediction?

Simulation means computing the model response using input data and initial conditions. The time samples of the model response match the time samples of the input data used for simulation.

For a continuous-time system, simulation means solving a differential equation. For a discrete-time system, simulation means directly applying the model equations.

For example, consider a dynamic model described by a first-order difference equation that uses a sample time of 1 second:

$$y(t) + ay(t–1) = bu(t–1),$$

where y is the output and u is the input. For parameter values $a = –0.9$ and $b = 1.5$, the equation becomes:

$$y(t) – 0.9y(t–1) = 1.5u(t–1).$$

Suppose you want to compute the values $y(1)$, $y(2)$, $y(3)$,... for given input values $u(0) = 2$, $u(1) = 1$, $u(2) = 4$,...Here, $y(1)$ is the value of output at the first sampling instant. Using initial condition of $y(0) = 0$, the values of $y(t)$ for times $t = 1$, 2 and 3 can be computed as:

$y(1) = 0.9y(0) + 1.5u(0) = 0.9*0 + 1.5*2 = 3$

$y(2) = 0.9y(1) + 1.5u(1) = 0.9*3 + 1.5*1 = 4.2$

$y(3) = 0.9y(2) + 1.5u(2) = 0.9*4.2 + 1.5*4 = 9.78$

...

Prediction forecasts the model response k steps ahead into the future using the current and past values of measured input and output values. k is called the prediction horizon, and corresponds to predicting output at time kT_s, where T_s is the sample time.

For example, suppose you use sensors to measure the input signal $u(t)$ and output signal $y(t)$ of the physical system, described in the previous first-order equation. At the tenth sampling instant ($t = 10$), the output $y(10)$ is 16 mm and the corresponding input $u(10)$ is 12 N. Now, you want to predict the value of the output at the future time $t = 11$. Using the previous equation:

$y(11) = 0.9y(10) + 1.5u(10)$

Hence, the predicted value of future output $y(11)$ at time $t = 10$ is:

$y(11) = 0.9*16 + 1.5*12 = 32.4$

In general, to predict the model response k steps into the future ($k \geq 1$) from the current time t, you should know the inputs up to time $t+k$ and outputs up to time t:

$$y_p(t+k) = \mathrm{f}(u(t+k),u(t+k-1),...,u(t),u(t-1),...,u(0)\\ y(t),y(t-1),y(t-2),...,y(0))$$

$u(0)$ and $y(0)$ are the initial states. `f()` represents the predictor, which is a dynamic model whose form depends on the model structure. For example, the one-step-ahead predictor y_p of the model $y(t) + ay(t-1) = bu(t)$ is:

$y_p(t+1) = -ay(t) + bu(t+1)$

The difference between prediction and simulation is that in prediction, the past values of outputs used for calculation are measured values while in simulation the outputs are themselves a result of calculation using inputs and initial conditions.

The way information in past outputs is used depends on the disturbance model H of the model. For the previous dynamic model, $H(z) = \frac{1}{1 + az^{-1}}$. In models of Output-Error (OE) structure ($H(z) = 1$), there is no information in past outputs that can be used for predicting future output values. In this case, predictions and simulations coincide. For state-space models (`idss`), output-error structure corresponds to models with `K=0`. For polynomial models (`idpoly`), this corresponds to models with polynomials `a=c=d=1`.

Note Prediction with $k=\infty$ means that no previous outputs are used in the computation and prediction returns the same result as simulation.

Both simulation and prediction require initial conditions, which correspond to the states of the model at the beginning of the simulation or prediction.

Tip If you do not know the initial conditions and have input and output measurements available, you can estimate the initial condition using this toolbox.

See Also

Related Examples

- “Simulation and Prediction in the App” on page 17-12
- “Simulation and Prediction at the Command Line” on page 17-17

More About

- “Simulating Identified Model Output in Simulink” on page 20-5

Simulation and Prediction in the App

In this section...
“How to Plot Simulated and Predicted Model Output” on page 17-12
“Interpret the Model Output Plot” on page 17-12
“Change Model Output Plot Settings” on page 17-14
“Definition: Confidence Interval” on page 17-16

How to Plot Simulated and Predicted Model Output

To create a model output plot for parametric linear and nonlinear models in the System Identification app, select the **Model output** check box in the **Model Views** area. By default, this operation estimates the initial states from the data and plots the output of selected models for comparison.

To include or exclude a model on the plot, click the corresponding model icon in the System Identification app. Active models display a thick line inside the Model Board icon.

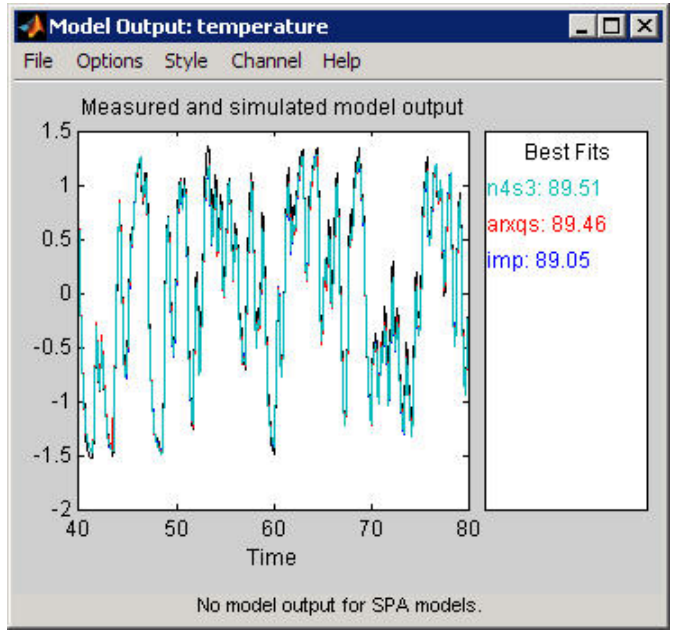
To learn how to interpret the model output plot, see “Interpret the Model Output Plot” on page 17-12.

To change plot settings, see “Change Model Output Plot Settings” on page 17-14.

For general information about creating and working with plots, see “Working with Plots” on page 21-11.

Interpret the Model Output Plot

The following figure shows a sample Model Output plot, created in the System Identification app.



The model output plot shows different information depending on the domain of the input-output validation data, as follows:

- For time-domain validation data, the plot shows simulated or predicted model output.
- For frequency-domain data, the plot shows the amplitude of the model response to the frequency-domain input signal. The model response is equal to the product of the Fourier transform of the input and the model's frequency function.
- For frequency-response data, the plot shows the amplitude of the model frequency response.

For linear models, you can estimate a model using time-domain data, and then validate the model using frequency domain data. For nonlinear models, you can only use time-domain data for both estimation and validation.

The right side of the plot displays the percentage of the output that the model reproduces (**Best Fit**), computed using the following equation:

$$\text{Best Fit} = \left(1 - \frac{|y - \hat{y}|}{|y - \bar{y}|}\right) \times 100$$

In this equation, y is the measured output, $\hat{y}$ is the simulated or predicted model output, and $\bar{y}$ is the mean of y. 100% corresponds to a perfect fit, and 0% indicates that the fit is no better than guessing the output to be a constant ($\hat{y} = \bar{y}$).

Because of the definition of **Best Fit**, it is possible for this value to be negative. A negative best fit is worse than 0% and can occur for the following reasons:

- The estimation algorithm failed to converge.
- The model was not estimated by minimizing $|y - \hat{y}|$. **Best Fit** can be negative when you minimized 1-step-ahead prediction during the estimation, but validate using the simulated output $\hat{y}$.
- The validation data set was not preprocessed in the same way as the estimation data set.

Change Model Output Plot Settings

The following table summarizes the Model Output plot settings.

Model Output Plot Settings

Action	Command
Display confidence intervals. **Note** Confidence intervals are only available for simulated model output of linear models. Confidence internal are not available for nonlinear ARX and Hammerstein-Wiener models. See "Definition: Confidence Interval" on page 17-16.	• To display the dashed lines on either side of the nominal model curve, select **Options > Show confidence intervals**. Select this option again to hide the confidence intervals. • To change the confidence value, select **Options > Set % confidence level**, and choose a value from the list. • To enter your own confidence level, select **Options > Set confidence level > Other**. Enter the value as a probability (between 0 and 1) or as the number of standard deviations of a Gaussian distribution.
Change between simulated output or predicted output. **Note** Prediction is only available for time-domain validation data.	• Select **Options > Simulated output** or **Options > k step ahead predicted output**. • To change the prediction horizon, select **Options > Set prediction horizon**, and select the number of samples. • To enter your own prediction horizon, select **Options > Set prediction horizon > Other**. Enter the value in terms of the number of samples.
Display the actual output values (**Signal plot**), or the difference between model output and measured output (**Error plot**).	Select **Options > Signal plot** or **Options > Error plot**.
(Time-domain validation data only) Set the time range for model output and the time interval for which the **Best Fit** value is computed.	Select **Options > Customized time span for fit** and enter the minimum and maximum time values. For example: `[1 20]`
(Multiple-output system only) Select a different output.	Select the output by name in the **Channel** menu.

Definition: Confidence Interval

The *confidence interval* corresponds to the range of output values with a specific probability of being the actual output of the system. The toolbox uses the estimated uncertainty in the model parameters to calculate confidence intervals and assumes the estimates have a Gaussian distribution.

For example, for a 95% confidence interval, the region around the nominal curve represents the range of values that have a 95% probability of being the true system response. You can specify the confidence interval as a probability (between 0 and 1) or as the number of standard deviations of a Gaussian distribution. For example, a probability of 0.99 (99%) corresponds to 2.58 standard deviations.

Note The calculation of the confidence interval assumes that the model sufficiently describes the system dynamics and the model residuals pass independence tests.

In the app, you can display a confidence interval on the plot to gain insight into the quality of a linear model. To learn how to show or hide confidence interval, see “Change Model Output Plot Settings” on page 17-14.

See Also

Related Examples

- “Simulation and Prediction at the Command Line” on page 17-17

More About

- “Simulating and Predicting Model Output” on page 17-9
- “Simulating Identified Model Output in Simulink” on page 20-5

Simulation and Prediction at the Command Line

In this section...
“Simulation and Prediction Commands” on page 17-17 “Initial States in Simulation and Prediction” on page 17-18 “Simulate a Continuous-Time State-Space Model” on page 17-19 “Simulate Model Output with Noise” on page 17-20

Simulation and Prediction Commands

Note If you estimated a linear model from detrended data and want to simulate or predict the output at the original operation conditions, use `retrend` to add trend data back into the simulated or predicted output.

Command	Description	Example
`compare`	Determine how closely the simulated model response matches the measured output signal. Plots simulated or predicted output of one or more models on top of the measured output. You should use an independent validation data set as input to the model.	To plot five-step-ahead predicted output of the model `mod` against the validation data `data`, use the following command: `compare(data,mod,5)` **Note** Omitting the third argument assumes an infinite horizon and results in the comparison of the simulated response to the input data.
`sim`	Simulate and plot the model output only.	To simulate the response of the model `model` using input data `data`, use the following command: `sim(model,data)`

Command	Description	Example

`predict`	Predict and plot the model output only.	To perform one-step-ahead prediction of the response for the model `model` and input data `data`, use the following command:
		`predict(model,data,1)`
		Use the following syntax to compute `k`-step-ahead prediction of the output signal using model `m`:
		`yhat = predict(m,[y u],k)`
		`predict` computes the prediction results only over the time range of `data`. It does not forecast results beyond the available data range.

`forecast`	Forecast a time series into the future.	To forecast the value of a time series in an arbitrary number of steps into the future, use the following command: `forecast(model,past_data,K)` Here, `model` is a time series model, `past_data` is a record of the observed values of the time series, and `K` is the forecasting horizon.

Initial States in Simulation and Prediction

The process of computing simulated and predicted responses over a time range starts by using the initial conditions to compute the first few output values. `sim`, `forecast`, and `predict` commands provide defaults for handling initial conditions.

Simulation: Default initial conditions are zero for all model types except `idnlgrey` model, in which case the default initial conditions are the internal model initial states (model property `x0`). You can specify other initial conditions using the `InitialCondition` simulation option (see `simOptions`).

Use the `compare` command to validate models by simulation because its algorithm estimates the initial states of a model to optimize the model fit to a given data set.

If you use `sim`, the simulated and the measured responses might differ when the initial conditions of the estimated model and the system that measured the validation data set differ—especially at the beginning of the response. To minimize this difference, estimate the initial state values from the data using `findstates` and specify these initial states using the `InitialCondition` simulation option (see `simOptions`). For example, to compute the initial states that optimize the fit of the model `m` to the output data in `z`:

```
% Estimate the initial states
X0est = findstates(m,z);
% Simulate the response using estimated initial states
opt = simOptions('InitialCondition',X0est);
sim(m,z.InputData,opt)
```

Prediction: Default initial conditions depend on the type of model. You can specify other initial conditions using the `InitialCondition` option (see `predictOptions`). For example, to compute the initial states that optimize the 1-step-ahead predicted response of the model `m` to the output data `z`:

```
opt = predictOptions('InitialCondition','estimate');
[Yp,X0est] = predict(m,z,1,opt);
```

This command returns the estimated initial states as the output argument `X0est`. For information about other ways to specify initials states, see the `predictOptions` reference page.

Simulate a Continuous-Time State-Space Model

This example shows how to simulate a continuous-time state-space model using a random binary input `u` and a sample time of `0.1 s`.

Consider the following state-space model:

$$\dot{x} = \begin{bmatrix} -1 & 1 \\ -0.5 & 0 \end{bmatrix} x + \begin{bmatrix} 1 \\ 0.5 \end{bmatrix} u + \begin{bmatrix} 0.5 \\ 0.5 \end{bmatrix} e$$

$$y = \begin{bmatrix} 1 & 0 \end{bmatrix} x + e$$

where e is Gaussian white noise with variance 7.

Create a continuous-time state-space model.

```
A = [-1 1; -0.5 0];
B = [1;0.5];
C = [1 0];
D = 0;
K = [0.5;0.5];
% Ts = 0 indicates continuous time
model_ss = idss(A,B,C,D,K,'Ts',0,'NoiseVariance',7);
```

Create a random binary input.

```
u = idinput(400,'rbs',[0 0.3]);
```

Create an `iddata` object with empty output to represent just the input signal.

```
data = iddata([],u);
data.ts = 0.1;
```

Simulate the output using the model

```
opt = simOptions('AddNoise',true);
y = sim(model_ss,data,opt);
```

Simulate Model Output with Noise

This example shows how you can create input data and a model, and then use the data and the model to simulate output data.

In this example, you create the following ARMAX model with Gaussian noise e:

$$y(t) - 1.5y(t-1) + 0.7y(t-2) = u(t-1) + 0.5u(t-2) + e(t) - e(t-1) + 0.2e(t-1)$$

Then, you simulate output data with random binary input `u`.

Create an ARMAX model.

```
m_armax = idpoly([1 -1.5 0.7],[0 1 0.5],[1 -1 0.2]);
```

Create a random binary input.

```
u = idinput(400,'rbs',[0 0.3]);
```

Simulate the output data.

```
opt = simOptions('AddNoise',true);
y = sim(m_armax,u,opt);
```

The `'AddNoise'` option specifies to include in the simulation the Gaussian noise `e` present in the model. Set this option to `false` (default behavior) to simulate the noise-free response to the input `u` , which is equivalent to setting `e` to zero.

See Also

`compare` | `forecast` | `predict` | `sim`

Related Examples

- “Compare Simulated Output with Measured Data” on page 17-22
- “Perform Multivariate Time Series Forecasting” on page 17-25
- “Simulation and Prediction in the App” on page 17-12

More About

- “Simulating and Predicting Model Output” on page 17-9
- “Simulating Identified Model Output in Simulink” on page 20-5

Compare Simulated Output with Measured Data

This example shows how to validate an estimated model by comparing the simulated model output with measured data.

Create estimation and validation data.

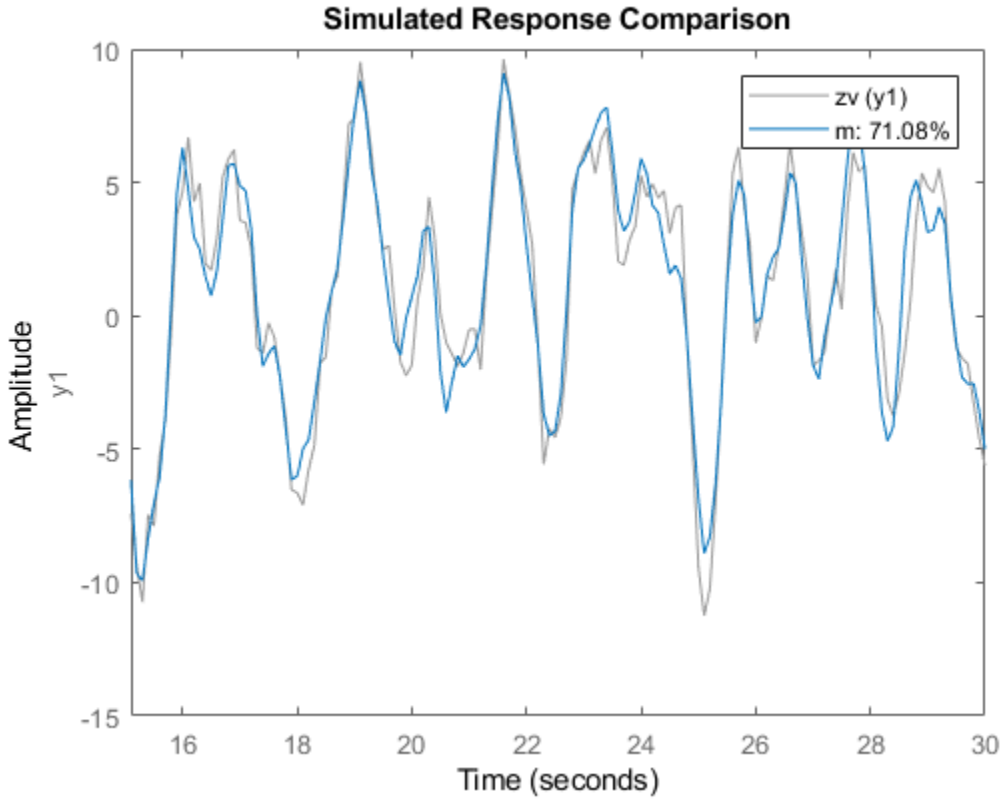
```
load iddata1;
ze = z1(1:150);
zv = z1(151:300);
```

Estimate an ARMAX model.

```
m = armax(ze,[2 3 1 0]);
```

Compare simulated model output with measured data.

```
compare(zv,m);
```



See Also

Related Examples

- “Simulation and Prediction at the Command Line” on page 17-17
- “Perform Multivariate Time Series Forecasting” on page 17-25

More About

- “Simulating and Predicting Model Output” on page 17-9

- “Why Simulate or Predict Model Output?” on page 17-9

Perform Multivariate Time Series Forecasting

This example shows how to perform multivariate time series forecasting of data measured from predator and prey populations in a prey crowding scenario. The predator-prey population-change dynamics are modeled using linear and nonlinear time series models. Forecasting performance of these models is compared.

Data Description

The data is a bivariate time series consisting of 1-predator 1-prey populations (in thousands) collected 10 times a year for 20 years. For more information about the data, see "Three Ecological Population Systems: MATLAB and C MEX-File Modeling of Time-Series".

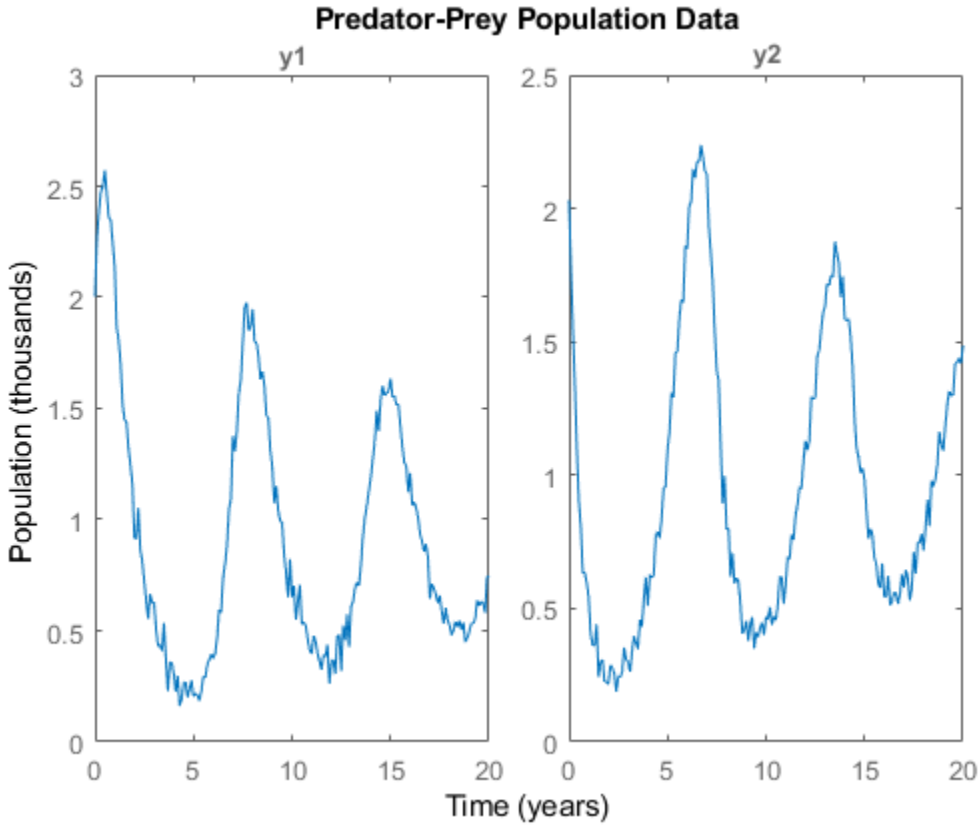
Load the time series data.

```
load PredPreyCrowdingData
z = iddata(y,[],0.1,'TimeUnit','years','Tstart',0);
```

`z` is an `iddata` object containing two output signals, `y1` and `y2`, which refer to the predator and prey populations, respectively. The `OutputData` property of `z` contains the population data as a 201-by-2 matrix, such that `z.OutputData(:,1)` is the predator population and `z.OutputData(:,2)` is the prey population.

Plot the data.

```
plot(z)
title('Predator-Prey Population Data')
ylabel('Population (thousands)')
```



The data exhibits a decline in predator population due to crowding.

Use the first half as estimation data for identifying time series models.

```
ze = z(1:120);
```

Use the remaining data to search for model orders, and to validate the forecasting results.

```
zv = z(121:end);
```

Estimate a Linear Model

Model the time series as a linear, autoregressive process. Linear models can be created in polynomial form or state-space form using commands such as `ar` (for scalar time series

only), `arx`, `armax`, `n4sid` and `ssest`. Since the linear models do not capture the data offsets (non-zero conditional mean), first detrend the data.

```
[zed, Tze] = detrend(ze, 0);
[zvd, Tzv] = detrend(zv, 0);
```

Identification requires specification of model orders. For polynomial models, you can find suitable orders using the `arxstruc` command. Since `arxstruc` works only on single-output models, perform the model order search separately for each output.

```
na_list = (1:10)';
V1 = arxstruc(zed(:,1,:),zvd(:,1,:),na_list);
na1 = selstruc(V1,0);
V2 = arxstruc(zed(:,2,:),zvd(:,2,:),na_list);
na2 = selstruc(V2,0);
```

The `arxstruc` command suggests autoregressive models of orders 7 and 8, respectively.

Use these model orders to estimate a multi-variance ARMA model where the cross terms have been chosen arbitrarily.

```
na = [na1 na1-1; na2-1 na2];
nc = [na1; na2];
sysARMA = armax(zed,[na nc])
```

```
sysARMA =
Discrete-time ARMA model:
  Model for output "y1": A(z)y_1(t) = - A_i(z)y_i(t) + C(z)e_1(t)

    A(z) = 1 - 0.885 z^-1 - 0.1493 z^-2 + 0.8089 z^-3 - 0.2661 z^-4
                                - 0.9487 z^-5 + 0.8719 z^-6 - 0.2896 z^-7


    A_2(z) = 0.3433 z^-1 - 0.2802 z^-2 - 0.04949 z^-3 + 0.1018 z^-4
                                           - 0.02683 z^-5 - 0.2416 z^-6


    C(z) = 1 - 0.4534 z^-1 - 0.4127 z^-2 + 0.7874 z^-3 + 0.298 z^-4
                                - 0.8684 z^-5 + 0.6106 z^-6 + 0.3616 z^-7

  Model for output "y2": A(z)y_2(t) = - A_i(z)y_i(t) + C(z)e_2(t)

    A(z) = 1 - 0.5826 z^-1 - 0.4688 z^-2 - 0.5949 z^-3 - 0.0547 z^-4
                  + 0.5062 z^-5 + 0.4024 z^-6 - 0.01544 z^-7 - 0.1766 z^-8
```

```
    A_1(z) = 0.2386 z^-1 + 0.1564 z^-2 - 0.2249 z^-3 - 0.2638 z^-4 - 0.1019 z^-5
                                                - 0.07821 z^-6 + 0.2982 z^-7


    C(z) = 1 - 0.1717 z^-1 - 0.09877 z^-2 - 0.5289 z^-3 - 0.24 z^-4
                + 0.06555 z^-5 + 0.2217 z^-6 - 0.05765 z^-7 - 0.1824 z^-8

Sample time: 0.1 years

Parameterization:
   Polynomial orders:   na=[7 6;7 8]   nc=[7;8]
   Number of free coefficients: 43
   Use "polydata", "getpvec", "getcov" for parameters and their uncertainties.

Status:
Estimated using ARMAX on time domain data "zed".
Fit to estimation data: [89.85;90.97]% (prediction focus)
FPE: 3.814e-05, MSE: 0.007533
```

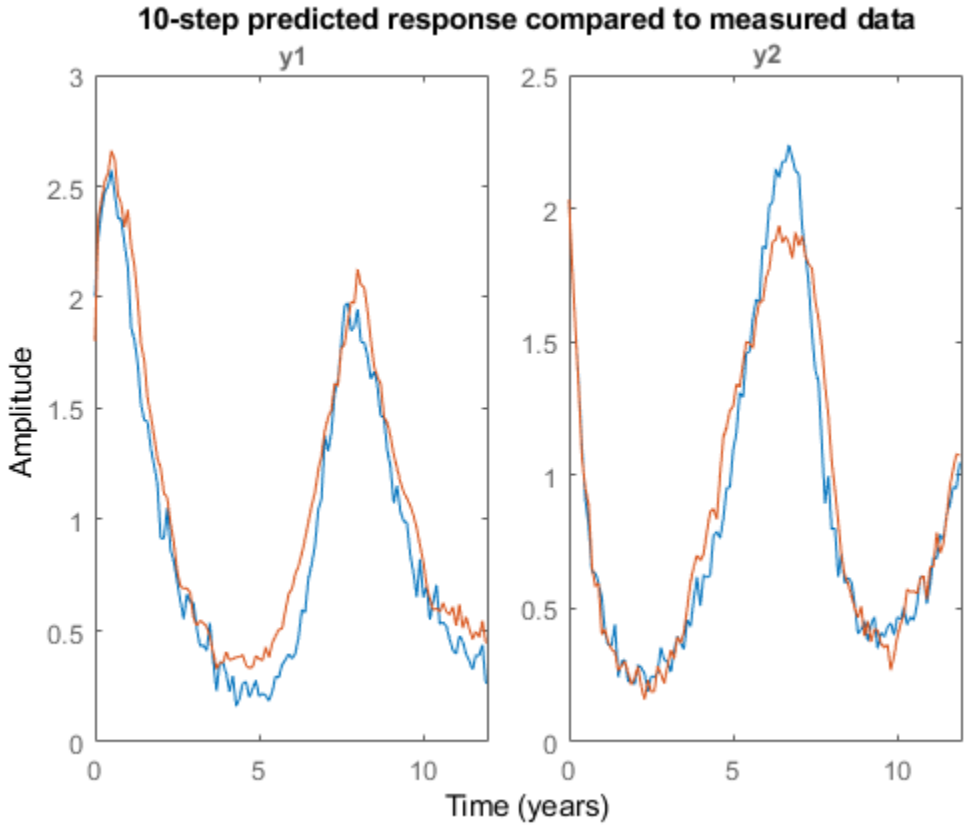
Compute a 10-step-ahead (1 year) predicted output to validate the model over the time span of the estimation data. Since the data was detrended for estimation, you need to specify those offsets for meaningful predictions.

```
predOpt = predictOptions('OutputOffset',Tze.OutputOffset');
yhat1 = predict(sysARMA,ze,10, predOpt);
```

The `predict` command predicts the response over the time span of measured data and is a tool for validating the quality of an estimated model. The response at time `t` is computed using measured values at times t = 0, ..., t-10.

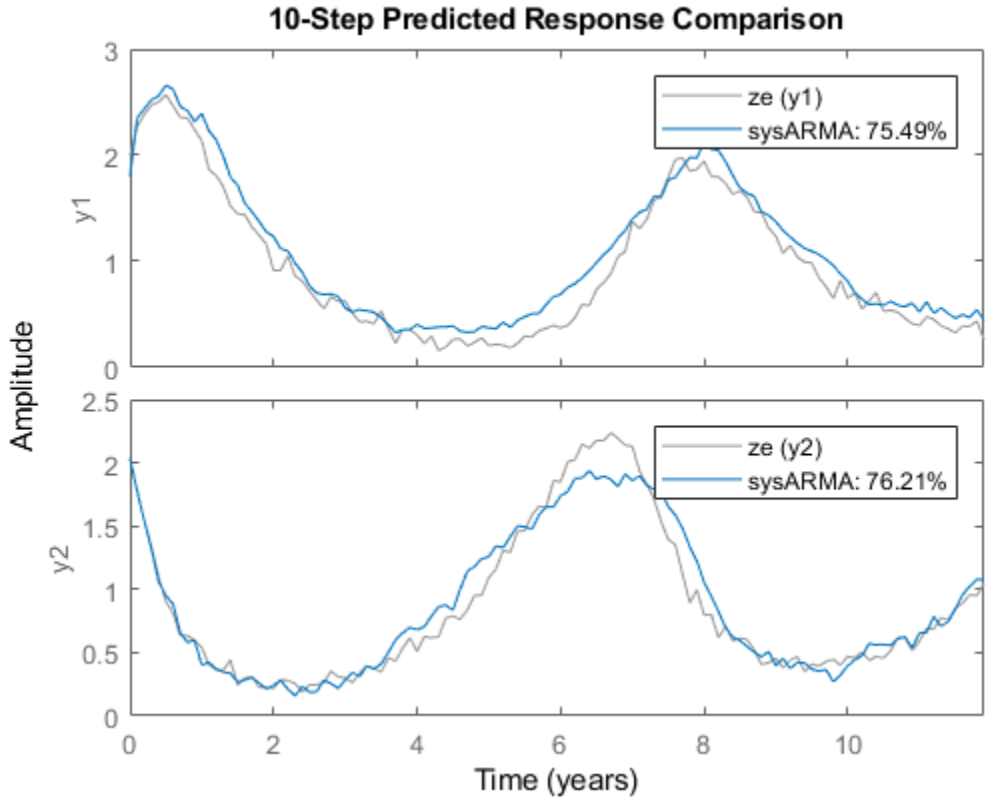
Plot the predicted response and the measured data.

```
plot(ze,yhat1)
title('10-step predicted response compared to measured data')
```



Note, the generation of predicted response and plotting it with the measured data, can be automated using the `compare` command.

```
compareOpt = compareOptions('OutputOffset',Tze.OutputOffset');
compare(ze,sysARMA,10,compareOpt)
```



The plot generated using `compare` also shows the normalized root mean square (NRMSE) measure of goodness of fit in percent form.

After validating the data, forecast the output of the model `sysARMA` 100 steps (10 years) beyond the estimation data, and calculate output standard deviations.

```
forecastOpt = forecastOptions('OutputOffset',Tze.OutputOffset');
[yf1,x01,sysf1,ysd1] = forecast(sysARMA, ze, 100, forecastOpt);
```

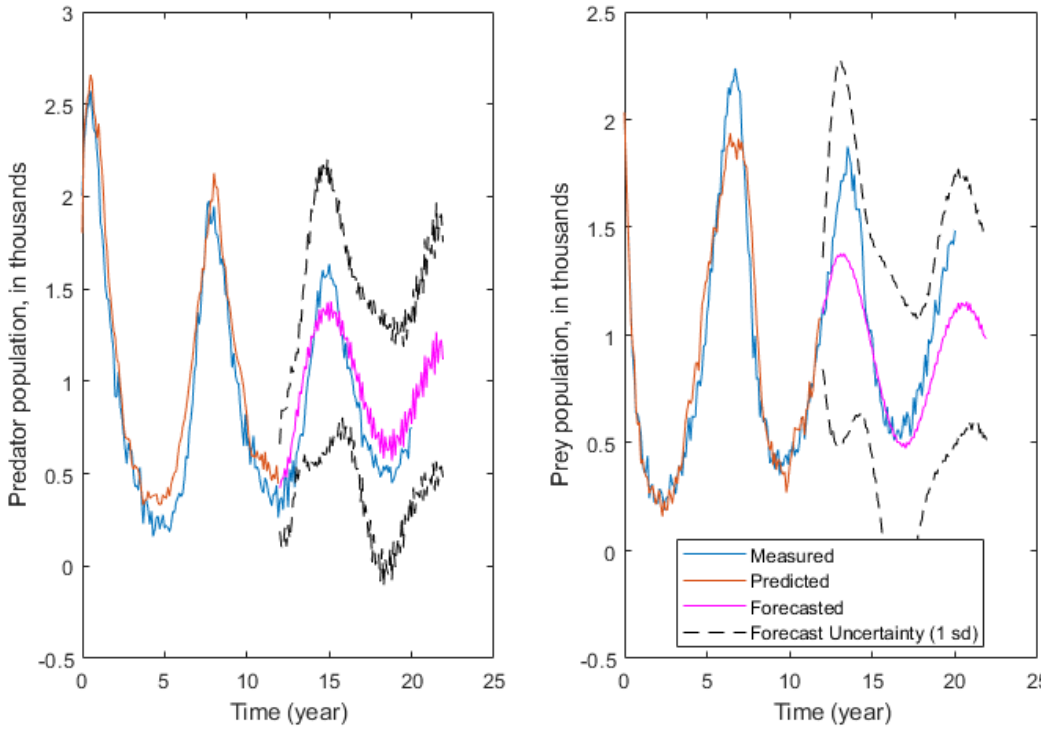
`yf1` is the forecasted response, returned as an `iddata` object. `yf1.OutputData` contains the forecasted values.

`sysf1` is a system similar to `sysARMA` but is in state-space form. Simulation of `sysf1` using the `sim` command, with initial conditions, `x01`, reproduces the forecasted response, `yf1`.

`ysd1` is the matrix of standard deviations. It measures the uncertainty is forecasting owing to the effect of additiive disturbances in the data (as measured by `sysARMA.NoiseVariance`), parameter uncertainty (as reported by `getcov(sysARMA)`) and uncertainities associated with the process of mapping past data to the initial conditions required for forecasting (see `data2state`).

Plot the measured, predicted, and forecasted output for model `sysARMA`.

```
t = yf1.SamplingInstants;
te = ze.SamplingInstants;
t0 = z.SamplingInstants;
subplot(1,2,1);
plot(t0,z.y(:,1),...
   te,yhat1.y(:,1),...
   t,yf1.y(:,1),'m',...
   t,yf1.y(:,1)+ysd1(:,1),'k--', ...
   t,yf1.y(:,1)-ysd1(:,1), 'k--')
xlabel('Time (year)');
ylabel('Predator population, in thousands');
subplot(1,2,2);
plot(t0,z.y(:,2),...
   te,yhat1.y(:,2),...
   t,yf1.y(:,2),'m',...
   t,yf1.y(:,2)+ysd1(:,2),'k--', ...
   t,yf1.y(:,2)-ysd1(:,2),'k--')
% Make the figure larger.
fig = gcf;
p = fig.Position;
fig.Position = [p(1),p(2)-p(4)*0.2,p(3)*1.4,p(4)*1.2];
xlabel('Time (year)');
ylabel('Prey population, in thousands');
legend({'Measured','Predicted','Forecasted','Forecast Uncertainty (1 sd)'},...
   'Location','best')
```



The plots show that forecasting using a linear ARMA model (with added handling of offsets) worked somewhat and the results showed high uncertainty compared to the actual populations over the 12-20 years time span. This indicates that the population change dynamics might be nonlinear.

Estimate a Nonlinear Black-Box Model

Fit a nonlinear black-box model to the estimation data. You do not require prior knowledge about the equations governing the estimation data. A linear-in-regressor form of Nonlinear ARX model will be estimated.

Create a nonlinear ARX model with 2 outputs and no inputs.

```
sysNLARX = idnlarx([1 1;1 1],[],'Ts',0.1,'TimeUnit','years');
```

`sysNLARX` is a first order nonlinear ARX model that uses no nonlinear function; it predicts the model response using a weighted sum of its first-order regressors.

```
getreg(sysNLARX)
```

```
Regressors:
  For output 1:
    y1(t-1)
    y2(t-1)
  For output 2:
    y1(t-1)
    y2(t-1)
```

To introduce a nonlinearity function, add polynomial regressors to the model.

Create regressors up to power 2, and include cross terms (products of standard regressors listed above). Add those regressors to the model as custom regressors.

```
R = polyreg(sysNLARX,'MaxPower',2,'CrossTerm','on');
sysNLARX.CustomRegressors = R;
getreg(sysNLARX)
```

```
Regressors:
  For output 1:
    y1(t-1)
    y2(t-1)
    y1(t-1).^2
    y1(t-1).*y2(t-1)
    y2(t-1).^2
  For output 2:
    y1(t-1)
    y2(t-1)
    y1(t-1).^2
    y1(t-1).*y2(t-1)
    y2(t-1).^2
```

Estimate the coefficients (the regressor weightings and the offset) of the model using estimation data, `ze`.

```
sysNLARX = nlarx(ze,sysNLARX)
```

```
sysNLARX =
Nonlinear ARX model with 2 outputs and 0 input
 Inputs
 Outputs: y1, y2
```

```
 Standard regressors corresponding to the orders:
   na = [1 1; 1 1]
   nb = []
   nk = []
 Custom regressors:
   For output 1:
     y1(t-1).^2
     y1(t-1).*y2(t-1)
     y2(t-1).^2
   For output 2:
     y1(t-1).^2
     y1(t-1).*y2(t-1)
     y2(t-1).^2
 Nonlinear regressors:
  For output 1:
    none
  For output 2:
    none
 Model outputs are linear in their regressors

Sample time: 0.1 years

Status:
Estimated using NLARX on time domain data "ze".
Fit to estimation data: [88.34;88.91]% (prediction focus)
FPE: 3.265e-05, MSE: 0.01048
```

Compute a 10-step-ahead predicted output to validate the model.

```
yhat2 = predict(sysNLARX,ze,10);
```

Forecast the output of the model 100 steps beyond the estimation data.

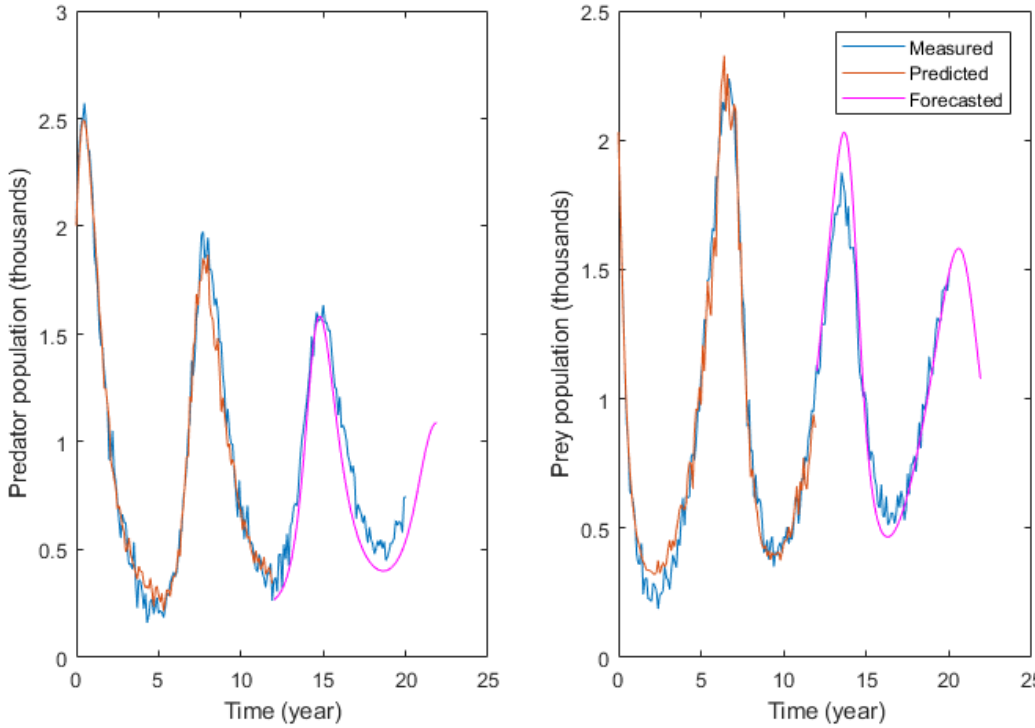
```
yf2 = forecast(sysNLARX,ze,100);
```

The standard deviations of the forecasted response are not computed for nonlinear ARX models. This data is unavailable because the parameter covariance information is not computed during estimation of these models.

Plot the measured, predicted, and forecasted outputs.

```
t = yf2.SamplingInstants;
subplot(1,2,1);
plot(t0,z.y(:,1),...
   te,yhat2.y(:,1),...
```

```
    t,yf2.y(:,1),'m')
xlabel('Time (year)');
ylabel('Predator population (thousands)');
subplot(1,2,2);
plot(t0,z.y(:,2),...
    te,yhat2.y(:,2),...
    t,yf2.y(:,2),'m')
legend('Measured','Predicted','Forecasted')
xlabel('Time (year)');
ylabel('Prey population (thousands)');
```



The plots show that forecasting using a nonlinear ARX model gave better forecasting results than using a linear model. Nonlinear black-box modeling did not require prior knowledge about the equations governing the data.

Note that to reduce the number of regressors, you can pick an optimal subset of (transformed) variables using principal component analysis (see `pca`) or feature selection (see `sequentialfs`) in the Statistics and Machine Learning Toolbox™.

If you have prior knowledge of the system dynamics, you can fit the estimation data using a nonlinear grey-box model.

Estimate a Nonlinear Grey-Box Model

Knowledge about the nature of the dynamics can help improve the model quality and thus the forecasting accuracy. For the predator-prey dynamics, the changes in the predator (`y1`) and prey (`y2`) population can be represented as:

$$\dot{y}_1(t) = p_1 * y_1(t) + p_2 * y_2(t) * y_1(t)$$

$$\dot{y}_2(t) = p_3 * y_2(t) - p_4 * y_1(t) * y_2(t) - p_5 * y_2(t)^2$$

For more information about the equations, see "Three Ecological Population Systems: MATLAB and C MEX-File Modeling of Time-Series".

Construct a nonlinear grey-box model based on these equations.

Specify a file describing the model structure for the predator-prey system. The file specifies the state derivatives and model outputs as a function of time, states, inputs, and model parameters. The two outputs (predator and prey populations) are chosen as states to derive a nonlinear state-space description of the dynamics.

```
FileName = 'predprey2_m';
```

Specify the model orders (number of outputs, inputs, and states)

```
Order = [2 0 2];
```

Specify the initial values for the parameters p_1, p_2, p_3, p_4, and p_5, and indicate that all parameters are to be estimated. Note that the requirement to specify initial guesses for parameters did not exist when estimating the black box models `sysARMA` and `sysNLARX`.

```
Parameters = struct('Name',{'Survival factor, predators' 'Death factor, predators' ...
   'Survival factor, preys' 'Death factor, preys' ...
   'Crowding factor, preys'}, ...
   'Unit',{'1/year' '1/year' '1/year' '1/year' '1/year'}, ...
```

```
   'Value',{-1.1 0.9 1.1 0.9 0.2}, ...
   'Minimum',{-Inf -Inf -Inf -Inf -Inf}, ...
   'Maximum',{Inf Inf Inf Inf Inf}, ...
   'Fixed',{false false false false false});
```

Similarly, specify the initial states of the model, and indicate that both initial states are to be estimated.

```
InitialStates = struct('Name',{'Predator population' 'Prey population'}, ...
   'Unit',{'Size (thousands)' 'Size (thousands)'}, ...
   'Value',{1.8 1.8}, ...
   'Minimum', {0 0}, ...
   'Maximum',{Inf Inf}, ...
   'Fixed',{false false});
```

Specify the model as a continuous-time system.

```
Ts = 0;
```

Create a nonlinear grey-box model with specified structure, parameters, and states.

```
sysGrey = idnlgrey(FileName,Order,Parameters,InitialStates,Ts,'TimeUnit','years');
```

Estimate the model parameters.

```
sysGrey = nlgreyest(ze,sysGrey);
present(sysGrey)
```

```
sysGrey =
Continuous-time nonlinear grey-box model defined by 'predprey2_m' (MATLAB file):

   dx/dt = F(t, x(t), p1, ..., p5)
    y(t) = H(t, x(t), p1, ..., p5) + e(t)

with 2 states, 2 outputs, and 5 free parameters (out of 5).

 States:                                          initial value
    x(1)  Predator population(t) [Size (thou..]   xinit@exp1   2.01325   (est) in [0, I
    x(2)  Prey population(t) [Size (thou..]       xinit@exp1   1.99687   (est) in [0, I
 Outputs:
    y(1)  y1(t)
    y(2)  y2(t)
 Parameters:                                  value       standard dev
    p1   Survival factor, predators [1/year]   -0.995895   0.0125269   (est) in [-Inf,
```

```
    p2   Death factor, predators [1/year]        1.00441   0.0129368   (est) in [-Inf,
    p3   Survival factor, preys [1/year]         1.01234   0.0135413   (est) in [-Inf,
    p4   Death factor, preys [1/year]            1.01909   0.0121026   (est) in [-Inf,
    p5   Crowding factor, preys [1/year]        0.103244   0.0039285   (est) in [-Inf,

Status:
Termination condition: Change in cost was less than the specified tolerance.
Number of iterations: 6, Number of function evaluations: 7

Estimated using Solver: ode45; Search: lsqnonlin on time domain data "ze".
Fit to estimation data: [91.21;92.07]%
FPE: 8.613e-06, MSE: 0.005713
More information in model's "Report" property.
```

Compute a 10-step-ahead predicted output to validate the model.

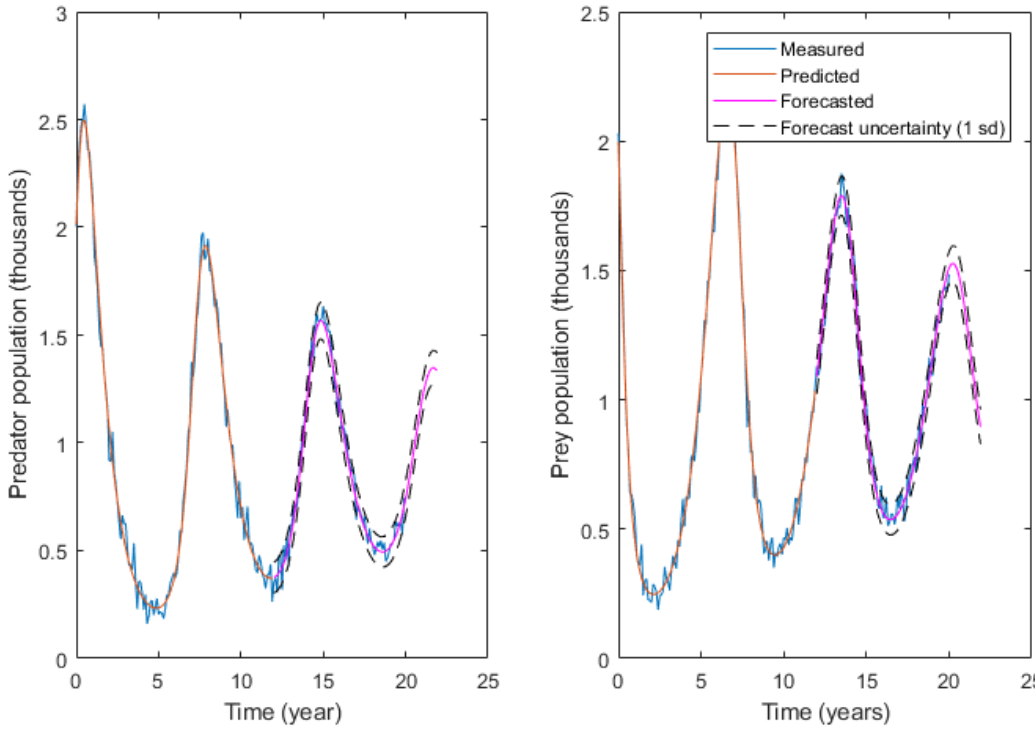
```
yhat3 = predict(sysGrey,ze,10);
```

Forecast the output of the model 100 steps beyond the estimation data, and calculate output standard deviations.

```
[yf3,x03,sysf3,ysd3] = forecast(sysGrey,ze,100);
```

Plot the measured, predicted, and forecasted outputs.

```
t = yf3.SamplingInstants;
subplot(1,2,1);
plot(t0,z.y(:,1),...
   te,yhat3.y(:,1),...
   t,yf3.y(:,1),'m',...
   t,yf3.y(:,1)+ysd3(:,1),'k--', ...
   t,yf3.y(:,1)-ysd3(:,1),'k--')
xlabel('Time (year)');
ylabel('Predator population (thousands)');
subplot(1,2,2);
plot(t0,z.y(:,2),...
   te,yhat3.y(:,2),...
   t,yf3.y(:,2),'m',...
   t,yf3.y(:,2)+ysd3(:,2),'k--', ...
   t,yf3.y(:,2)-ysd3(:,2),'k--')

legend('Measured','Predicted','Forecasted','Forecast uncertainty (1 sd)')
xlabel('Time (years)');
ylabel('Prey population (thousands)');
```

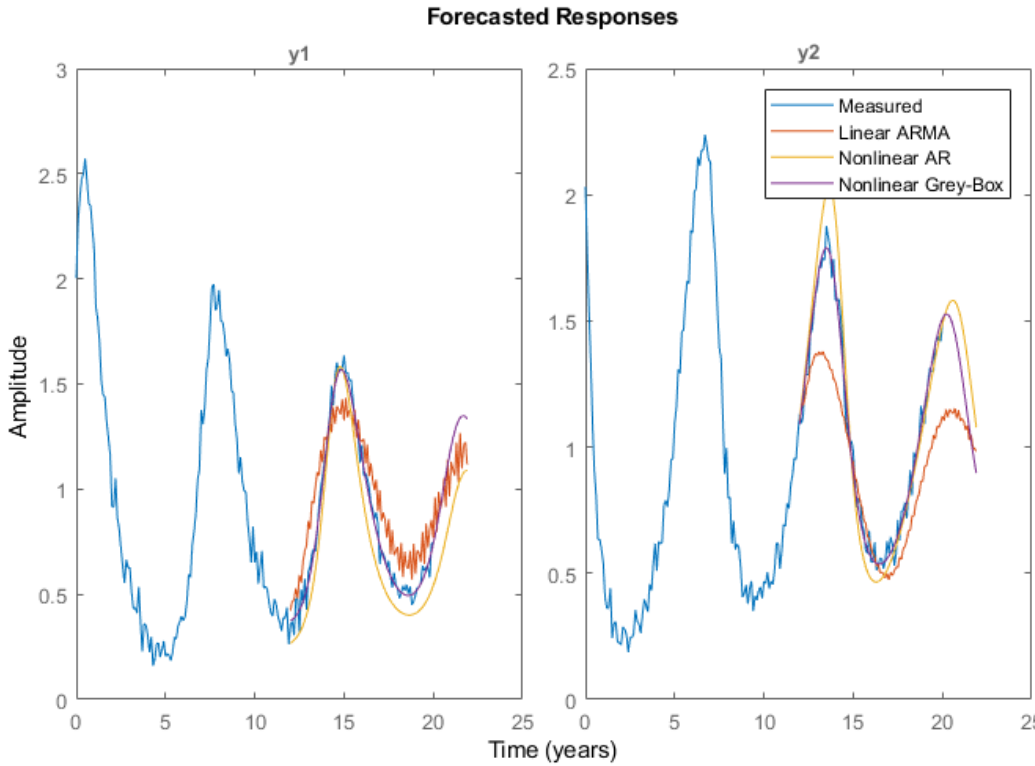


The plots show that forecasting using a nonlinear grey-box model gave good forecasting results and low forecasting output uncertainty.

Compare Forecasting Performance

Compare the forecasted response obtained from the identified models, `sysARMA`, `sysNLARX`, and `sysGrey`. The first two are discrete-time models and `sysGrey` is a continuous-time model.

```
clf
plot(z,yf1,yf2,yf3)
legend({'Measured','Linear ARMA','Nonlinear AR','Nonlinear Grey-Box'})
title('Forecasted Responses')
```



Forecasting with a nonlinear ARX model gave better results than forecasting with a linear model. Inclusion of the knowledge of the dynamics in the nonlinear grey-box model further improved the reliability of the model and therefore the forecasting accuracy.

Note that the equations used in grey-box modeling are closely related to the polynomial regressors used by the Nonlinear ARX model. If you approximate the derivatives in the governing equations by first-order differences, you will get equations similar to:

$$y_1(t) = s_1 * y_1(t-1) + s_2 * y_1(t-1) * y_2(t-1)$$

$$y_2(t) = s_3 * y_2(t-1) - s_4 * y_1(t-1) * y_2(t-1) - s_5 * y_2(t-1)^2$$

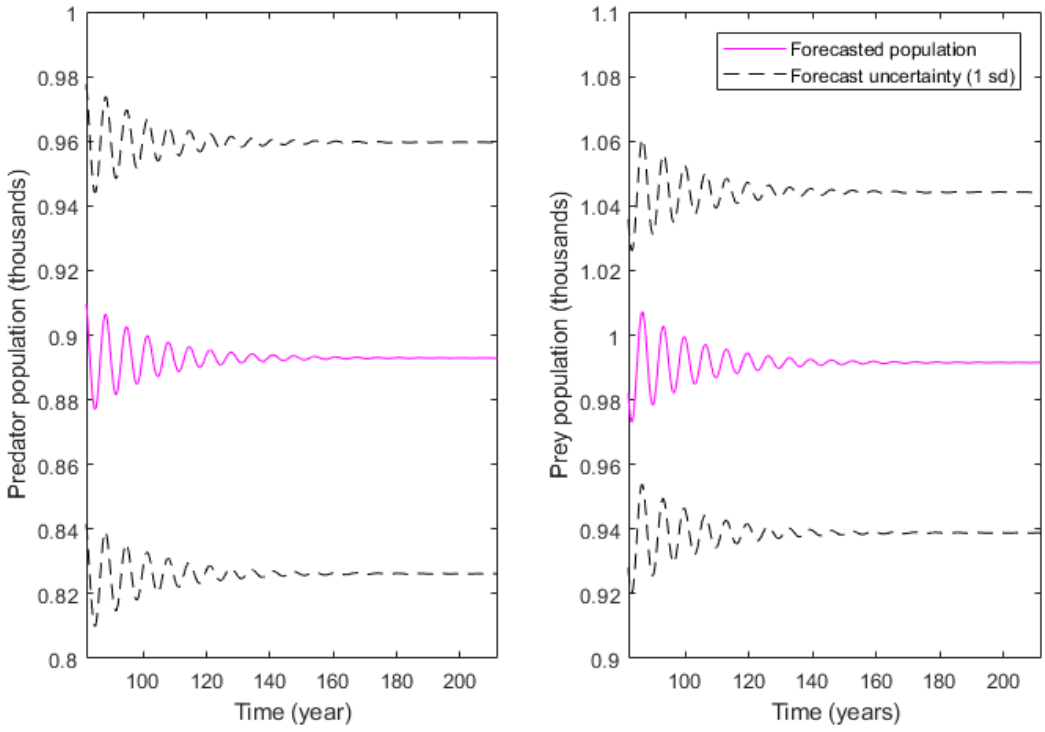
Where, $s_1, \ldots, s_5$ are some parameters related to the original parameters $p_1, \ldots, p_5$ and to the sample time used for differencing. These equations suggest that only 2 regressors are needed for the first output (y1(t-1) and *y1(t-1)*y2(t-1)) and 3 for the second output when constructing the Nonlinear ARX model. Even in absence of such prior knowledge, linear-in-regressor model structures employing polynomial regressors remain a popular choice in practice.

Forecast the values using the nonlinear grey-box model over 200 years.

```
[yf4,~,~,ysd4] = forecast(sysGrey, ze, 2000);
```

Plot the latter part of the data (showing 1 sd uncertainty)

```
t = yf4.SamplingInstants;
N = 700:2000;
subplot(1,2,1);
plot(t(N), yf4.y(N,1), 'm',...
   t(N), yf4.y(N,1)+ysd4(N,1), 'k--', ...
   t(N), yf4.y(N,1)-ysd4(N,1), 'k--')
xlabel('Time (year)');
ylabel('Predator population (thousands)');
ax = gca;
ax.YLim = [0.8 1];
ax.XLim = [82 212];
subplot(1,2,2);
plot(t(N),yf4.y(N,2),'m',...
   t(N),yf4.y(N,2)+ysd4(N,2),'k--', ...
   t(N),yf4.y(N,2)-ysd4(N,2),'k--')
legend('Forecasted population','Forecast uncertainty (1 sd)')
xlabel('Time (years)');
ylabel('Prey population (thousands)');
ax = gca;
ax.YLim = [0.9 1.1];
ax.XLim = [82 212];
```



The plot show that the predatory population is forecasted to reach a steady-state of approximately 890 and the prey population is forecasted to reach 990.

See Also

Related Examples

- “Introduction to Forecasting of Dynamic System Response” on page 14-37
- “Simulation and Prediction at the Command Line” on page 17-17
- “Identify Time-Series Models at the Command Line” on page 14-14

What Is Residual Analysis?

Residuals are differences between the one-step-predicted output from the model and the measured output from the validation data set. Thus, residuals represent the portion of the validation data not explained by the model.

Residual analysis consists of two tests: the whiteness test and the independence test.

According to the *whiteness test* criteria, a good model has the residual autocorrelation function inside the confidence interval of the corresponding estimates, indicating that the residuals are uncorrelated.

According to the *independence test* criteria, a good model has residuals uncorrelated with past inputs. Evidence of correlation indicates that the model does not describe how part of the output relates to the corresponding input. For example, a peak outside the confidence interval for lag k means that the output $y(t)$ that originates from the input $u(t-k)$ is not properly described by the model.

Your model should pass both the whiteness and the independence tests, except in the following cases:

- For output-error (OE) models and when using instrumental-variable (IV) methods, make sure that your model shows independence of `e` and `u`, and pay less attention to the results of the whiteness of `e`.

  In this case, the modeling focus is on the dynamics G and not the disturbance properties H.
- Correlation between residuals and input for negative lags, is not necessarily an indication of an inaccurate model.

  When current residuals at time t affect future input values, there might be feedback in your system. In the case of feedback, concentrate on the positive lags in the cross-correlation plot during model validation.

Supported Model Types

You can validate parametric linear and nonlinear models by checking the behavior of the model residuals. For a description of residual analysis, see “What Residual Plots Show for Different Data Domains” on page 17-44.

Note Residual analysis plots are not available for frequency response (FRD) models. For time-series models, you can only generate model-output plots for parametric models using time-domain time-series (no input) measured data.

What Residual Plots Show for Different Data Domains

Residual analysis plots show different information depending on whether you use time-domain or frequency-domain input-output validation data.

For time-domain validation data, the plot shows the following two axes:

- Autocorrelation function of the residuals for each output
- Cross-correlation between the input and the residuals for each input-output pair

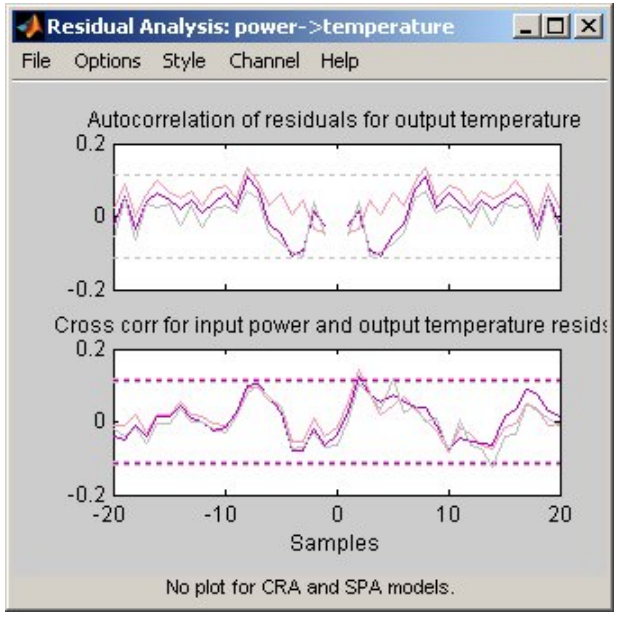
Note For time-series models, the residual analysis plot does not provide any input-residual correlation plots.

For frequency-domain validation data, the plot shows the following two axes:

- Estimated power spectrum of the residuals for each output
- Transfer-function amplitude from the input to the residuals for each input-output pair

For linear models, you can estimate a model using time-domain data, and then validate the model using frequency domain data. For nonlinear models, the System Identification Toolbox product supports only time-domain data.

The following figure shows a sample Residual Analysis plot, created in the System Identification app.



Displaying the Confidence Interval

The *confidence interval* corresponds to the range of residual values with a specific probability of being statistically insignificant for the system. The toolbox uses the estimated uncertainty in the model parameters to calculate confidence intervals and assumes the estimates have a Gaussian distribution.

For example, for a 95% confidence interval, the region around zero represents the range of residual values that have a 95% probability of being statistically insignificant. You can specify the confidence interval as a probability (between 0 and 1) or as the number of standard deviations of a Gaussian distribution. For example, a probability of 0.99 (99%) corresponds to 2.58 standard deviations.

You can display a confidence interval on the plot in the app to gain insight into the quality of the model. To learn how to show or hide confidence interval, see the description of the plot settings in “How to Plot Residuals in the App” on page 17-47.

Note If you are working in the System Identification app, you can specify a custom confidence interval. If you are using the `resid` command, the confidence interval is fixed at 99%.

See Also

Related Examples

- “How to Plot Residuals in the App” on page 17-47
- “Examine Model Residuals” on page 17-51

How to Plot Residuals in the App

To create a residual analysis plot for parametric linear and nonlinear models in the System Identification app, select the **Model resids** check box in the **Model Views** area. For general information about creating and working with plots, see “Working with Plots” on page 21-11.

To include or exclude a model on the plot, click the corresponding model icon in the System Identification app. Active models display a thick line inside the Model Board icon.

The following table summarizes the Residual Analysis plot settings.

Residual Analysis Plot Settings

Action	Command
Display confidence intervals around zero. **Note** Confidence internal are not available for nonlinear ARX and Hammerstein-Wiener models.	• To display the dashed lines on either side of the nominal model curve, select **Options > Show confidence intervals**. Select this option again to hide the confidence intervals. • To change the confidence value, select **Options > Set % confidence level** and choose a value from the list. • To enter your own confidence level, select **Options > Set confidence level > Other**. Enter the value as a probability (between 0 and 1) or as the number of standard deviations of a Gaussian distribution.
Change the number of lags (data samples) for which to compute autocorellation and cross-correlation functions. **Note** For frequency-domain validation data, increasing the number of lags increases the frequency resolution of the residual spectrum and the transfer function.	• Select **Options > Number of lags** and choose the value from the list. • To enter your own lag value, select **Options > Set confidence level > Other**. Enter the value as the number of data samples.
(Multiple-output system only) Select a different input-output pair.	Select the input-output by name in the **Channel** menu.

See Also

Related Examples

- “How to Plot Residuals at the Command Line” on page 17-50
- “Examine Model Residuals” on page 17-51

More About

- "What Is Residual Analysis?" on page 17-43

How to Plot Residuals at the Command Line

The following table summarizes commands that generate residual-analysis plots for linear and nonlinear models. For detailed information about this command, see the corresponding reference page.

Note Apply `pe` and `resid` to one model at a time.

Command	Description	Example
`pe`	Computes and plots model prediction errors.	To plot the prediction errors for the model `model` using data `data`, type the following command: `pe(model,data)`
`resid`	Performs whiteness and independence tests on model residuals, or prediction errors. Uses validation data input as model input.	To plot residual correlations for the model `model` using data `data`, type the following command: `resid(data,model)`

See Also

Related Examples

- “How to Plot Residuals in the App” on page 17-47
- “Examine Model Residuals” on page 17-51

More About

- “What Is Residual Analysis?” on page 17-43

Examine Model Residuals

This example shows how you can use residual analysis to evaluate model quality.

Creating Residual Plots

1. To load the sample System Identification app session that contains estimated models, type the following command in the MATLAB Command Window:

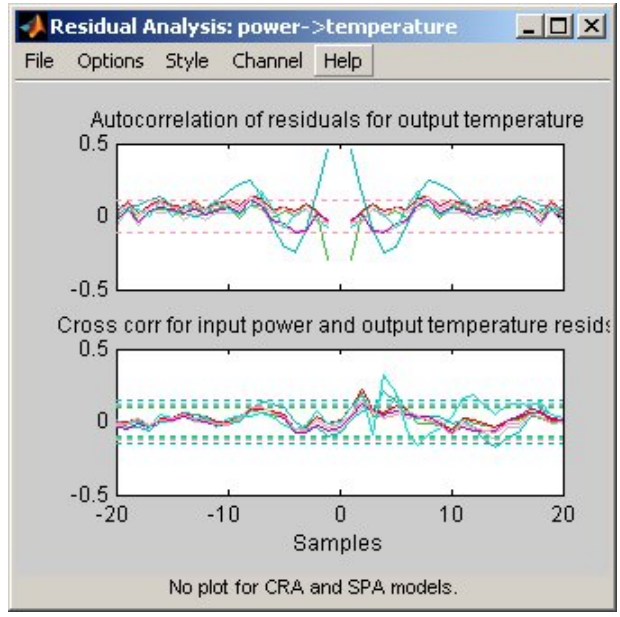
   ```
   systemIdentification('dryer2_linear_models')
   ```

2. To generate a residual analysis plot, select the **Model resids** check box in the System Identification app.

   This opens an empty plot.

3. In the System Identification app window, click each model icon to display it on the Residual Analysis plot.

Note For the nonparametric models, `imp` and `spad`, residual analysis plots are not available.



Description of the Residual Plot Axes

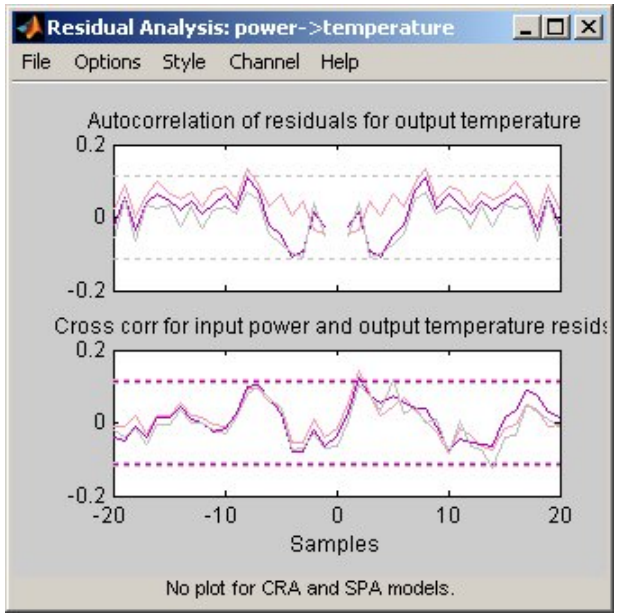
The top axes show the autocorrelation of residuals for the output (whiteness test). The horizontal scale is the number of lags, which is the time difference (in samples) between the signals at which the correlation is estimated. The horizontal dashed lines on the plot represent the confidence interval of the corresponding estimates. Any fluctuations within the confidence interval are considered to be insignificant. Four of the models, `arxqs`, `n4s3`, `arx223` and `amx2222`, produce residuals that enter outside the confidence interval. A good model should have a residual autocorrelation function within the confidence interval, indicating that the residuals are uncorrelated.

The bottom axes show the cross-correlation of the residuals with the input. A good model should have residuals uncorrelated with past inputs (independence test). Evidence of correlation indicates that the model does not describe how the output is formed from the corresponding input. For example, when there is a peak outside the confidence interval for lag k, this means that the contribution to the output $y(t)$ that originates from the input $u(t-k)$ is not properly described by the model. The models `arxqs` and `amx2222` extend beyond the confidence interval and do not perform as well as the other models.

Validating Models Using Analyzing Residuals

To remove models with poor performance from the Residual Analysis plot, click the model icons `arxqs`, `n4s3`, `arx223`, and `amx2222` in the System Identification app.

The Residual Analysis plot now includes only the three models that pass the residual tests: `arx692`, `n4s6`, and `amx3322`.



The plots for these models fall within the confidence intervals. Thus, when choosing the best model among several estimated models, it is reasonable to pick `amx3322` because it is a simpler, low-order model.

See Also

Related Examples

- “How to Plot Residuals in the App” on page 17-47
- “How to Plot Residuals at the Command Line” on page 17-50

More About

- “What Is Residual Analysis?” on page 17-43

Impulse and Step Response Plots

In this section...
"Supported Models" on page 17-55
"How Transient Response Helps to Validate Models" on page 17-55
"What Does a Transient Response Plot Show?" on page 17-56
"Displaying the Confidence Interval" on page 17-57

Supported Models

You can plot the simulated response of a model using impulse and step signals as the input for all linear parametric models and correlation analysis (nonparametric) models.

You can also create step-response plots for nonlinear models. These step and impulse response plots, also called *transient response* plots, provide insight into the characteristics of model dynamics, including peak response and settling time.

Note For frequency-response models, impulse- and step-response plots are not available. For nonlinear models, only step-response plots are available.

Examples

"Plot Impulse and Step Response Using the System Identification App" on page 17-59

"Plot Impulse and Step Response at the Command Line" on page 17-62

How Transient Response Helps to Validate Models

Transient response plots provide insight into the basic dynamic properties of the model, such as response times, static gains, and delays.

Transient response plots also help you validate how well a linear parametric model, such as a linear ARX model or a state-space model, captures the dynamics. For example, you can estimate an impulse or step response from the data using correlation analysis (nonparametric model), and then plot the correlation analysis result on top of the transient responses of the parametric models.

Because nonparametric and parametric models are derived using different algorithms, agreement between these models increases confidence in the parametric model results.

What Does a Transient Response Plot Show?

Transient response plots show the value of the impulse or step response on the vertical axis. The horizontal axis is in units of time you specified for the data used to estimate the model.

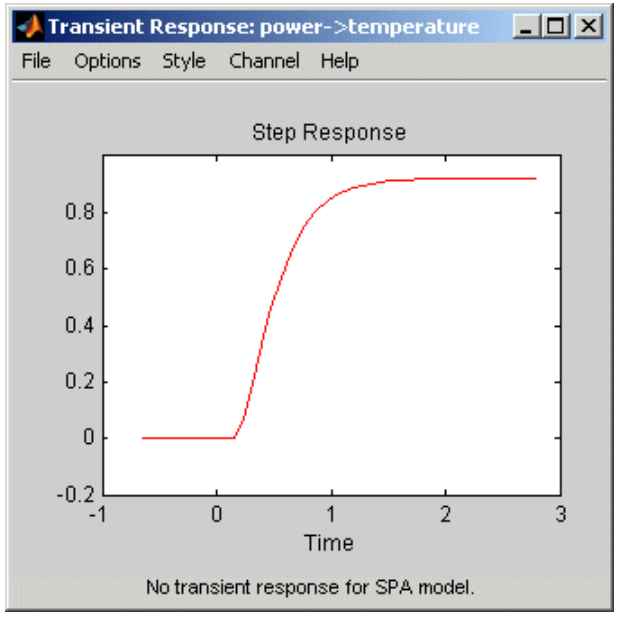
The impulse response of a dynamic model is the output signal that results when the input is an impulse. That is, *u(t)* is zero for all values of *t* except at *t*=0, where *u(0)*=1. In the following difference equation, you can compute the impulse response by setting *y(-T)*=*y(-2T)*=0, *u(0)*=1, and *u(t>0)*=0.

$$y(t) - 1.5y(t-T) + 0.7y(t-2T) = 0.9u(t) + 0.5u(t-T)$$

The step response is the output signal that results from a step input, where *u(t<0)*=0 and *u(t>0)*=1.

If your model includes a noise model, you can display the transient response of the noise model associated with each output channel. For more information about how to display the transient response of the noise model, see “Plot Impulse and Step Response Using the System Identification App” on page 17-59.

The following figure shows a sample Transient Response plot, created in the System Identification app.



Displaying the Confidence Interval

In addition to the transient-response curve, you can display a confidence interval on the plot. To learn how to show or hide confidence interval, see the description of the plot settings in “Plot Impulse and Step Response Using the System Identification App” on page 17-59.

The *confidence interval* corresponds to the range of response values with a specific probability of being the actual response of the system. The toolbox uses the estimated uncertainty in the model parameters to calculate confidence intervals and assumes the estimates have a Gaussian distribution.

For example, for a 95% confidence interval, the region around the nominal curve represents the range where there is a 95% chance that it contains the true system response. You can specify the confidence interval as a probability (between 0 and 1) or as the number of standard deviations of a Gaussian distribution. For example, a probability of 0.99 (99%) corresponds to 2.58 standard deviations.

Note The calculation of the confidence interval assumes that the model sufficiently describes the system dynamics and the model residuals pass independence tests.

Plot Impulse and Step Response Using the System Identification App

To create a transient analysis plot in the System Identification app, select the **Transient resp** check box in the **Model Views** area. For general information about creating and working with plots, see “Working with Plots” on page 21-11.

To include or exclude a model on the plot, click the corresponding model icon in the System Identification app. Active models display a thick line inside the Model Board icon.

The following table summarizes the Transient Response plot settings.

Transient Response Plot Settings

Action	Command
Display step response for linear or nonlinear model.	Select **Options > Step response**.
Display impulse response for linear model.	Select **Options > Impulse response**. **Note** Not available for nonlinear models.
Display the confidence interval. **Note** Only available for linear models.	• To display the dashed lines on either side of the nominal model curve, select **Options > Show confidence intervals**. Select this option again to hide the confidence intervals. • To change the confidence value, select **Options > Set % confidence level**, and choose a value from the list. • To enter your own confidence level, select **Options > Set confidence level > Other**. Enter the value as a probability (between 0 and 1) or as the number of standard deviations of a Gaussian distribution.
Change time span over which the impulse or step response is calculated. For a scalar time span *T*, the resulting response is plotted from *-T/4* to *T*. **Note** To change the time span of models you estimated using correlation analysis models, select **Estimate > Correlation models** and reestimate the model using a new time span.	• Select **Options > Time span (time units)**, and choose a new time span in units of time you specified for the model. • To enter your own time span, select **Options > Time span (time units) > Other**, and enter the total response duration. • To use the time span based on model dynamics, type `[]` or `default`. The `default` time span is computed based on the model dynamics and might be different for different models. For nonlinear models, the default time span is 10.

Action	Command
Toggle between line plot or stem plot. **Tip** Use a stem plot for displaying impulse response.	Select **Style** > **Line plot** or **Style** > **Stem plot**.
(Multiple-output system only) Select an input-output pair to view the noise spectrum corresponding to those channels.	Select the output by name in the **Channel** menu. If the plotted models include a noise model, you can display the transient response properties associated with each output channel. The name of the channel has the format `e@OutputName`, where `OutputName` is the name of the output channel corresponding to the noise model.
(Step response for nonlinear models only) Set level of the input step. **Note** For multiple-input models, the input-step level applies only to the input channel you selected to display in the plot.	Select **Options** > **Step Size**, and then chose from two options: • **0–>1** sets the lower level to `0` and the upper level to `1`. • **Other** opens the Step Level dialog box, where you enter the values for the lower and upper level values.

More About

“Impulse and Step Response Plots” on page 17-55

Plot Impulse and Step Response at the Command Line

You can plot impulse- and step-response plots using the `impulseplot` and `stepplot` commands, respectively. If you want to fetch the response data, use `impulse` and `step` instead.

All plot commands have the same basic syntax, as follows:

- To plot one model, use the syntax *`command(model)`*.
- To plot several models, use the syntax *`command(model1,model2,...,modelN)`*.

In this case, `command` represents any of the plotting commands.

To display confidence intervals for a specified number of standard deviations, use the following syntax:

```
h = impulseplot(model);
showConfidence(h,sd);
```

where `h` is the plot handle returned by `impulseplot`. You could also use the plot handle returned by `stepplot`. `sd` is the number of standard deviations of a Gaussian distribution. For example, a confidence value of 99% for the nominal model curve corresponds to 2.58 standard deviations.

Alternatively, you can turn on the confidence region view interactively by right-clicking on the plot and selecting **Characteristics** > **Confidence Region**. Use the plot property editor to specify the number of standard deviations.

The following table summarizes commands that generate impulse- and step-response plots. For detailed information about each command, see the corresponding reference page.

Command	Description	Example
`impulse`,`impulseplot`	Plot impulse response for `idpoly`, `idproc`, `idtf`, `idss`, and `idgrey` model objects. **Note** Does not support nonlinear models.	To plot the impulse response of the model `sys`, type the following command: `impulse(sys)`

Command	Description	Example

`step`,`stepplot`	Plots the step response of all linear and nonlinear models.	To plot the step response of the model `sys`, type the following command:
		`step(sys)`
		To specify the step level offset (`u0`) and amplitude (`A`) for a model:
		`opt = stepDataOptions;`
		`opt.InputOffset = u0;`
		`opt.StepAmplitude = A;`
		`step(sys,opt)`

More About

"Impulse and Step Response Plots" on page 17-55

Frequency Response Plots

In this section...
"What Is Frequency Response?" on page 17-64
"How Frequency Response Helps to Validate Models" on page 17-65
"What Does a Frequency-Response Plot Show?" on page 17-65
"Displaying the Confidence Interval" on page 17-66

What Is Frequency Response?

Frequency response plots show the complex values of a transfer function as a function of frequency.

In the case of linear dynamic systems, the transfer function G is essentially an operator that takes the input u of a linear system to the output y:

$$y = Gu$$

For a continuous-time system, the transfer function relates the Laplace transforms of the input $U(s)$ and output $Y(s)$:

$$Y(s) = G(s)U(s)$$

In this case, the frequency function $G(iw)$ is the transfer function evaluated on the imaginary axis $s=iw$.

For a discrete-time system sampled with a time interval T, the transfer function relates the Z-transforms of the input $U(z)$ and output $Y(z)$:

$$Y(z) = G(z)U(z)$$

In this case, the frequency function $G(e^{iwT})$ is the transfer function $G(z)$ evaluated on the unit circle. The argument of the frequency function $G(e^{iwT})$ is scaled by the sample time T to make the frequency function periodic with the sampling frequency $2\pi/T$.

Examples

"Plot Bode Plots Using the System Identification App" on page 17-68

"Plot Bode and Nyquist Plots at the Command Line" on page 17-71

How Frequency Response Helps to Validate Models

You can plot the frequency response of a model to gain insight into the characteristics of linear model dynamics, including the frequency of the peak response and stability margins. Frequency-response plots are available for all linear models.

Note Frequency-response plots are not available for nonlinear models. In addition, Nyquist plots do not support time-series models that have no input.

The frequency response of a linear dynamic model describes how the model reacts to sinusoidal inputs. If the input *u(t)* is a sinusoid of a certain frequency, then the output *y(t)* is also a sinusoid of the same frequency. However, the magnitude of the response is different from the magnitude of the input signal, and the phase of the response is shifted relative to the input signal.

Frequency response plots provide insight into linear systems dynamics, such as frequency-dependent gains, resonances, and phase shifts. Frequency response plots also contain information about controller requirements and achievable bandwidths. Finally, frequency response plots can also help you validate how well a linear parametric model, such as a linear ARX model or a state-space model, captures the dynamics.

One example of how frequency-response plots help validate other models is that you can estimate a frequency response from the data using spectral analysis (nonparametric model), and then plot the spectral analysis result on top of the frequency response of the parametric models. Because nonparametric and parametric models are derived using different algorithms, agreement between these models increases confidence in the parametric model results.

What Does a Frequency-Response Plot Show?

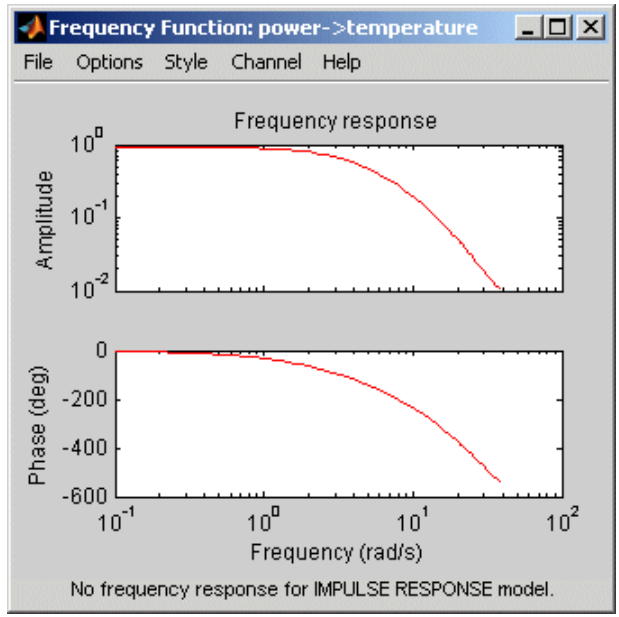
System Identification app supports the following types of frequency-response plots for linear parametric models, linear state-space models, and nonparametric frequency-response models:

- Bode plot of the model response. A Bode plot consists of two plots. The top plot shows the magnitude $|G|$ by which the transfer function G magnifies the amplitude of the sinusoidal input. The bottom plot shows the phase $\varphi = \arg G$ by which the transfer

function shifts the input. The input to the system is a sinusoid, and the output is also a sinusoid with the same frequency.

- Plot of the disturbance model, called *noise spectrum*. This plot is the same as a Bode plot of the model response, but it shows the output power spectrum of the noise model instead. For more information, see "Noise Spectrum Plots" on page 17-73.
- (Only in the MATLAB Command Window)
  Nyquist plot. Plots the imaginary versus the real part of the transfer function.

The following figure shows a sample Bode plot of the model dynamics, created in the System Identification app.



Displaying the Confidence Interval

In addition to the frequency-response curve, you can display a confidence interval on the plot. To learn how to show or hide confidence interval, see the description of the plot settings in "Plot Bode Plots Using the System Identification App" on page 17-68

The *confidence interval* corresponds to the range of response values with a specific probability of being the actual response of the system. The toolbox uses the estimated

uncertainty in the model parameters to calculate confidence intervals and assumes the estimates have a Gaussian distribution.

For example, for a 95% confidence interval, the region around the nominal curve represents the range where there is a 95% chance that it contains the true system response. You can specify the confidence interval as a probability (between 0 and 1) or as the number of standard deviations of a Gaussian distribution. For example, a probability of 0.99 (99%) corresponds to 2.58 standard deviations.

Plot Bode Plots Using the System Identification App

To create a frequency-response plot for linear models in the System Identification app, select the **Frequency resp** check box in the **Model Views** area. For general information about creating and working with plots, see “Working with Plots” on page 21-11.

To include or exclude a model on the plot, click the corresponding model icon in the System Identification app. Active models display a thick line inside the Model Board icon.

The following table summarizes the Frequency Function plot settings.

Frequency Function Plot Settings

Action	Command
Display the confidence interval.	• To display the dashed lines on either side of the nominal model curve, select **Options** > **Show confidence intervals**. Select this option again to hide the confidence intervals. • To change the confidence value, select **Options** > **Set % confidence level**, and choose a value from the list. • To enter your own confidence level, select **Options** > **Set confidence level** > **Other**. Enter the value as a probability (between 0 and 1) or as the number of standard deviations of a Gaussian distribution.
Change the frequency values for computing the noise spectrum. The default frequency vector is 128 linearly distributed values, greater than zero and less than or equal to the Nyquist frequency.	Select **Options** > **Frequency range** and specify a new frequency vector in units of rad/s. Enter the frequency vector using any one of following methods: • MATLAB expression, such as `[1:100]*pi/100` or `logspace(-3,-1,200)`. Cannot contain variables in the MATLAB workspace. • Row vector of values, such as `[1:.1:100]` **Note** To restore the default frequency vector, enter `[]`.
Change frequency units between hertz and radians per second.	Select **Style** > **Frequency (Hz)** or **Style** > **Frequency (rad/s)**.
Change frequency scale between linear and logarithmic.	Select **Style** > **Linear frequency scale** or **Style** > **Log frequency scale**.
Change amplitude scale between linear and logarithmic.	Select **Style** > **Linear amplitude scale** or **Style** > **Log amplitude scale**.

Action	Command
(Multiple-output system only) Select an input-output pair to view the noise spectrum corresponding to those channels. **Note** You cannot view cross spectra between different outputs.	Select the output by name in the **Channel** menu.

More About

"Frequency Response Plots" on page 17-64

Plot Bode and Nyquist Plots at the Command Line

You can plot Bode and Nyquist plots for linear models using the `bode` and `nyquist` commands. If you want to customize the appearance of the plot, or turn on the confidence region programmatically, use `bodeplot`, and `nyquistplot` instead.

All plot commands have the same basic syntax, as follows:

- To plot one model, use the syntax *`command(model)`*.
- To plot several models, use the syntax *`command(model1,model2,...,modelN)`*.

In this case, *`command`* represents any of the plotting commands.

To display confidence intervals for a specified number of standard deviations, use the following syntax:

```
h=command(model);
showConfidence(h,sd)
```

where `sd` is the number of standard deviations of a Gaussian distribution and command is `bodeplot`or `nyquistplot`. For example, a confidence value of 99% for the nominal model curve corresponds to 2.58 standard deviations.

The following table summarizes commands that generate Bode and Nyquist plots for linear models. For detailed information about each command and how to specify the frequency values for computing the response, see the corresponding reference page.

Command	Description	Example
`bode` and `bodeplot`	Plots the magnitude and phase of the frequency response on a logarithmic frequency scale. **Note** Does not support time-series models.	To create the bode plot of the model, `sys`, use the following command: `bode(sys)`

Command	Description	Example
`nyquist` and `nyquistplot`	Plots the imaginary versus real part of the transfer function. **Note** Does not support time-series models.	To plot the frequency response of the model, `sys`, use the following command: `nyquist(sys)`
`spectrum` and `spectrumplot`	Plots the disturbance spectra of input-output models and output spectra of time series models.	To plot the output spectrum of a time series model, `sys`, with 1 standard deviation confidence region, use the following command: `showConfidence(spectrumplot(sys));`

More About

“Frequency Response Plots” on page 17-64

Noise Spectrum Plots

In this section...
“Supported Models” on page 17-73 “What Does a Noise Spectrum Plot Show?” on page 17-73 “Displaying the Confidence Interval” on page 17-74

Supported Models

When you estimate the noise model of your linear system, you can plot the spectrum of the estimated noise model. Noise-spectrum plots are available for all linear parametric models and spectral analysis (nonparametric) models.

Note For nonlinear models and correlation analysis models, noise-spectrum plots are not available. For time-series models, you can only generate noise-spectrum plots for parametric and spectral-analysis models.

Examples

“Plot the Noise Spectrum Using the System Identification App” on page 17-76

“Plot the Noise Spectrum at the Command Line” on page 17-79

What Does a Noise Spectrum Plot Show?

The general equation of a linear dynamic system is given by:

$$y(t) = G(z)u(t) + v(t)$$

In this equation, G is an operator that takes the input to the output and captures the system dynamics, and v is the additive noise term. The toolbox treats the noise term as filtered white noise, as follows:

$$v(t) = H(z)e(t)$$

where $e(t)$ is a white-noise source with variance λ.

The toolbox computes both H and λ during the estimation of the noise model and stores these quantities as model properties. The *H(z)* operator represents the noise model.

Whereas the frequency-response plot shows the response of G, the noise-spectrum plot shows the frequency-response of the noise model H.

For input-output models, the noise spectrum is given by the following equation:

$$\Phi_v(\omega) = \lambda \left| H\left(e^{i\omega}\right) \right|^2$$

For time-series models (no input), the vertical axis of the noise-spectrum plot is the same as the dynamic model spectrum. These axes are the same because there is no input for time series and $y = He$.

Note You can avoid estimating the noise model by selecting the Output-Error model structure or by setting the `DisturbanceModel` property value to `'None'` for a state space model. If you choose to not estimate a noise model for your system, then H and the noise spectrum amplitude are equal to 1 at all frequencies.

Displaying the Confidence Interval

In addition to the noise-spectrum curve, you can display a confidence interval on the plot. To learn how to show or hide confidence interval, see the description of the plot settings in “Plot the Noise Spectrum Using the System Identification App” on page 17-76.

The *confidence interval* corresponds to the range of power-spectrum values with a specific probability of being the actual noise spectrum of the system. The toolbox uses the estimated uncertainty in the model parameters to calculate confidence intervals and assumes the estimates have a Gaussian distribution.

For example, for a 95% confidence interval, the region around the nominal curve represents the range where there is a 95% chance that the true response belongs.. You can specify the confidence interval as a probability (between 0 and 1) or as the number of standard deviations of a Gaussian distribution. For example, a probability of 0.99 (99%) corresponds to 2.58 standard deviations.

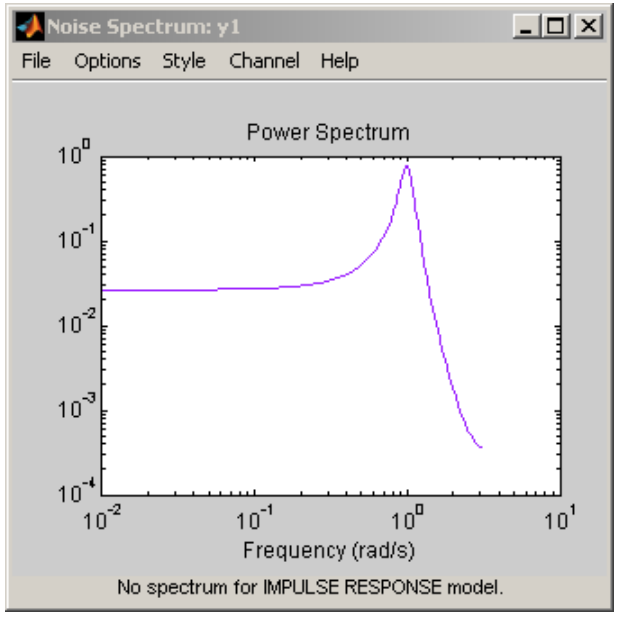
Note The calculation of the confidence interval assumes that the model sufficiently describes the system dynamics and the model residuals pass independence tests.

Plot the Noise Spectrum Using the System Identification App

To create a noise spectrum plot for parametric linear models in the app, select the **Noise spectrum** check box in the **Model Views** area. For general information about creating and working with plots, see “Working with Plots” on page 21-11.

To include or exclude a model on the plot, click the corresponding model icon in the System Identification app. Active models display a thick line inside the Model Board icon.

The following figure shows a sample Noise Spectrum plot.



The following table summarizes the Noise Spectrum plot settings.

Noise Spectrum Plot Settings

Action	Command
Display the confidence interval.	• To display the dashed lines on either side of the nominal model curve, select **Options > Show confidence intervals**. Select this option again to hide the confidence intervals. • To change the confidence value, select **Options > Set % confidence level**, and choose a value from the list. • To enter your own confidence level, select **Options > Set confidence level > Other**. Enter the value as a probability (between 0 and 1) or as the number of standard deviations of a Gaussian distribution.
Change the frequency values for computing the noise spectrum. The default frequency vector is 128 linearly distributed values, greater than zero and less than or equal to the Nyquist frequency.	Select **Options > Frequency range** and specify a new frequency vector in units of radians per second. Enter the frequency vector using any one of following methods: • MATLAB expression, such as `[1:100]*pi/100` or `logspace(-3,-1,200)`. Cannot contain variables in the MATLAB workspace. • Row vector of values, such as `[1:.1:100]` **Tip** To restore the default frequency vector, enter `[]`.
Change frequency units between hertz and radians per second.	Select **Style > Frequency (Hz)** or **Style > Frequency (rad/s)**.
Change frequency scale between linear and logarithmic.	Select **Style > Linear frequency scale** or **Style > Log frequency scale**.
Change amplitude scale between linear and logarithmic.	Select **Style > Linear amplitude scale** or **Style > Log amplitude scale**.

Action	Command
(Multiple-output system only) Select an input-output pair to view the noise spectrum corresponding to those channels. **Note** You cannot view cross spectra between different outputs.	Select the output by name in the **Channel** menu.

More About

"Noise Spectrum Plots" on page 17-73

Plot the Noise Spectrum at the Command Line

To plot the disturbance spectrum of an input-output model or the output spectrum of a time series model, use `spectrum`. To customize such plots, or to turn on the confidence region view programmatically for such plots, use `spectrumplot` instead.

To determine if your estimated noise model is good enough, you can compare the output spectrum of the estimated noise-model *H* to the estimated output spectrum of *v(t)*. To compute *v(t)*, which represents the actual noise term in the system, use the following commands:

```
ysimulated = sim(m,data);
v = ymeasured-ysimulated;
```

`ymeasured` is `data.y`. `v` is the noise term *v(t)*, as described in "What Does a Noise Spectrum Plot Show?" on page 17-73 and corresponds to the difference between the simulated response `ysimulated` and the actual response `ymeasured`.

To compute the frequency-response model of the actual noise, use `spa`:

```
V = spa(v);
```

The toolbox uses the following equation to compute the noise spectrum of the actual noise:

$$\Phi_v(\omega) = \sum_{\tau=-\infty}^{\infty} R_v(\tau) e^{-i\omega\tau}$$

The covariance function R_v is given in terms of *E*, which denotes the mathematical expectation, as follows:

$$R_v(\tau) = Ev(t)v(t-\tau)$$

To compare the parametric noise-model *H* to the (nonparametric) frequency-response estimate of the actual noise *v(t)*, use `spectrum`:

```
spectrum(V,m)
```

If the parametric and the nonparametric estimates of the noise spectra are different, then you might need a higher-order noise model.

More About

“Noise Spectrum Plots” on page 17-73

Pole and Zero Plots

In this section...
“Supported Models” on page 17-81 “What Does a Pole-Zero Plot Show?” on page 17-81 “Displaying the Confidence Interval” on page 17-83

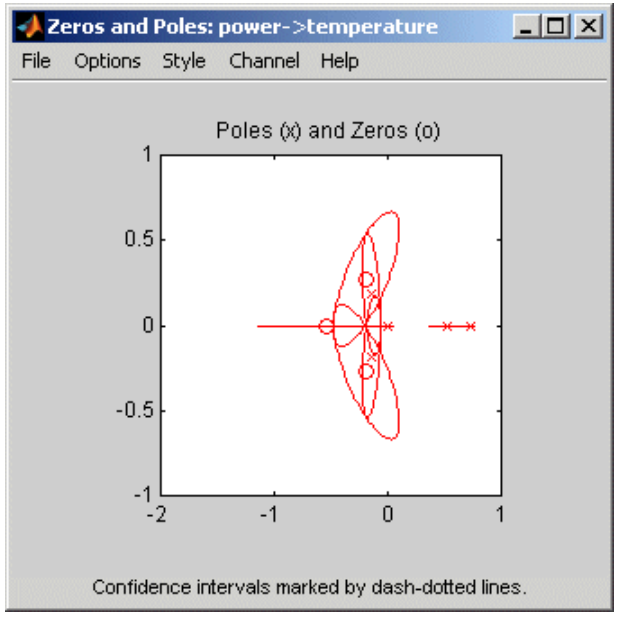
Supported Models

You can create pole-zero plots of linear identified models. To study the poles and zeros of the noise component of an input-output model or a time series model, use `noise2meas` to first extract the noise model as an independent input-output model, whose inputs are the noise channels of the original model.

For examples of creating pole-zero plots, see “Model Poles and Zeros Using the System Identification App” on page 17-85 and “Plot Poles and Zeros at the Command Line” on page 17-87.

What Does a Pole-Zero Plot Show?

The following figure shows a sample pole-zero plot of the model with confidence intervals. `x` indicate poles and `o` indicate zeros.



The general equation of a linear dynamic system is given by:

$$y(t) = G(z)u(t) + v(t)$$

In this equation, G is an operator that takes the input to the output and captures the system dynamics, and v is the additive noise term.

The *poles* of a linear system are the roots of the denominator of the transfer function G. The poles have a direct influence on the dynamic properties of the system. The *zeros* are the roots of the numerator of G. If you estimated a noise model H in addition to the dynamic model G, you can also view the poles and zeros of the noise model.

Zeros and the poles are equivalent ways of describing the coefficients of a linear difference equation, such as the ARX model. Poles are associated with the output side of the difference equation, and zeros are associated with the input side of the equation. The number of poles is equal to the number of sampling intervals between the most-delayed and least-delayed output. The number of zeros is equal to the number of sampling intervals between the most-delayed and least-delayed input. For example, there two poles and one zero in the following ARX model:

$$y(t) - 1.5y(t-T) + 0.7y(t-2T) = 0.9u(t) + 0.5u(t-T)$$

Displaying the Confidence Interval

You can display a confidence interval for each pole and zero on the plot. To learn how to show or hide confidence interval, see "Model Poles and Zeros Using the System Identification App" on page 17-85 and "Plot Poles and Zeros at the Command Line" on page 17-87.

The *confidence interval* corresponds to the range of pole or zero values with a specific probability of being the actual pole or zero of the system. The toolbox uses the estimated uncertainty in the model parameters to calculate confidence intervals and assumes the estimates have a Gaussian distribution.

For example, for a 95% confidence interval, the region around the nominal pole or zero value represents the range of values that have a 95% probability of being the true system pole or zero value. You can specify the confidence interval as a probability (between 0 and 1) or as the number of standard deviations of a Gaussian distribution. For example, a probability of 0.99 (99%) corresponds to 2.58 standard deviations.

You can use pole-zero plots to evaluate whether it might be useful to reduce model order. When confidence intervals for a pole-zero pair overlap, this overlap indicates a possible pole-zero cancellation. For more information, see "Reducing Model Order Using Pole-Zero Plots" on page 17-84.

See Also

More About

- "Model Poles and Zeros Using the System Identification App" on page 17-85
- "Plot Poles and Zeros at the Command Line" on page 17-87

Reducing Model Order Using Pole-Zero Plots

You can use pole-zero plots of linear identified models to evaluate whether it might be useful to reduce model order. When confidence intervals for a pole-zero pair overlap, this overlap indicates a possible pole-zero cancellation.

For example, you can use the following syntax to plot a 1-standard deviation confidence interval around model poles and zeros.

```
showConfidence(iopzplot(model))
```

If poles and zeros overlap, try estimating a lower order model.

Always validate model output and residuals to see if the quality of the fit changes after reducing model order. If the plot indicates pole-zero cancellations, but reducing model order degrades the fit, then the extra poles probably describe noise. In this case, you can choose a different model structure that decouples system dynamics and noise. For example, try ARMAX, Output-Error, or Box-Jenkins polynomial model structures with an A or F polynomial of an order equal to that of the number of uncanceled poles. For more information about estimating linear polynomial models, see “Input-Output Polynomial Models”.

See Also

`iopzplot`

More About

- “Pole and Zero Plots” on page 17-81
- “Plot Poles and Zeros at the Command Line” on page 17-87
- “Model Poles and Zeros Using the System Identification App” on page 17-85

Model Poles and Zeros Using the System Identification App

To create a pole-zero plot for parametric linear models in the System Identification app, select the **Zeros and poles** check box in the **Model Views** area. For general information about creating and working with plots, see “Working with Plots” on page 21-11.

To include or exclude a model on the plot, click the corresponding model icon in the System Identification app. Active models display a thick line inside the Model Board icon.

The following table summarizes the Zeros and Poles plot settings.

Zeros and Poles Plot Settings

Action	Command
Display the confidence interval.	• To display the dashed lines on either side of the nominal pole and zero values, select **Options > Show confidence intervals**. Select this option again to hide the confidence intervals. • To change the confidence value, select **Options > Set % confidence level**, and choose a value from the list. • To enter your own confidence level, select **Options > Set confidence level > Other**. Enter the value as a probability (between 0 and 1) or as the number of standard deviations of a Gaussian distribution.
Show real and imaginary axes.	Select **Style > Re/Im-axes**. Select this option again to hide the axes.
Show the unit circle.	Select **Style > Unit circle**. Select this option again to hide the unit circle. The unit circle is useful as a reference curve for discrete-time models.
(Multiple-output system only) Select an input-output pair to view the poles and zeros corresponding to those channels.	Select the output by name in the **Channel** menu.

See Also

More About

- “Pole and Zero Plots” on page 17-81
- “Plot Poles and Zeros at the Command Line” on page 17-87

Plot Poles and Zeros at the Command Line

You can create a pole-zero plot for linear identified models using the `iopzmap` and `iopzplot` commands.

To display confidence intervals for a specified number of standard deviations, use the following syntax:

```
h = iopzplot(model);
showConfidence(h,sd)
```

where `sd` is the number of standard deviations of a Gaussian distribution. For example, a confidence value of 99% for the nominal model curve corresponds to 2.58 standard deviations.

Command	Description	Example
`iopzmap`,`iopzplot`	Plots zeros and poles of the model on the S-plane or Z-plane for continuous-time or discrete-time model, respectively.	To plot the poles and zeros of the model `sys`, use the following command: `iopzmap(sys)`

See Also

More About

- “Pole and Zero Plots” on page 17-81
- “Model Poles and Zeros Using the System Identification App” on page 17-85

Analyzing MIMO Models

In this section...
“Overview of Analyzing MIMO Models” on page 17-88
“Array Selector” on page 17-89
“I/O Grouping for MIMO Models” on page 17-91
“Selecting I/O Pairs” on page 17-92

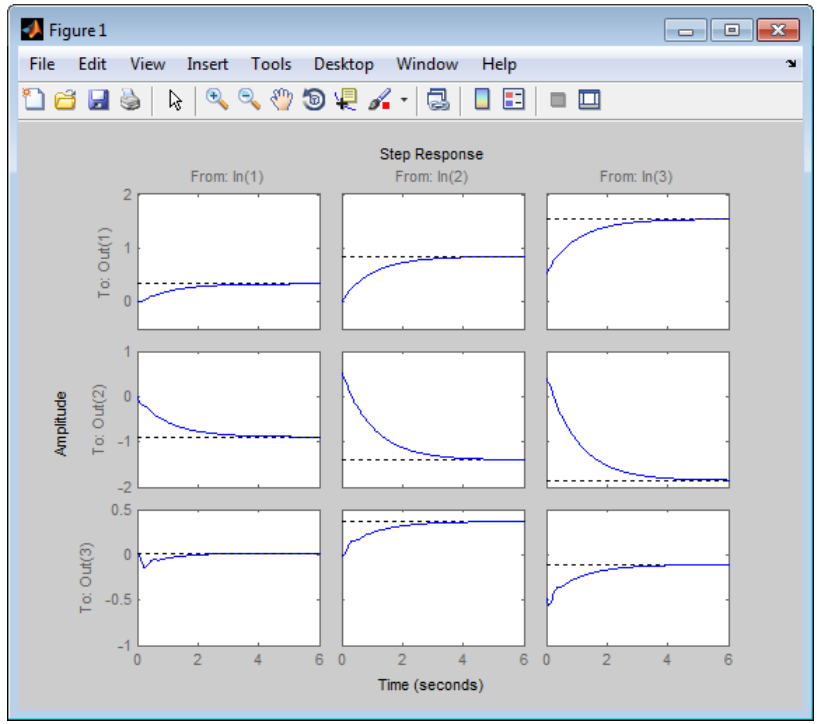
Overview of Analyzing MIMO Models

If you plot a MIMO system, or an LTI array containing multiple identified 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 a random 3-input, 3-output MIMO system and then randomly sample it 10 times. Plot the step response for all the models.

```
sys_mimo=rsample(idss(rss(3,3,3)),10);
step(sys_mimo);
```

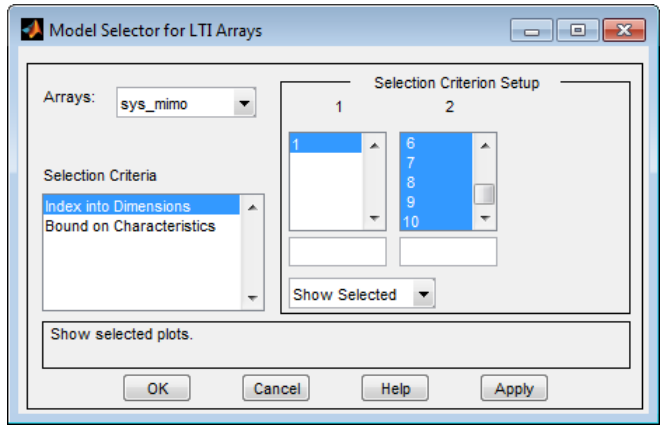
`sys_mimo` is an array of ten 3-input, 3-output systems.

A set of 9 plots appears, one from each input to each output, for the ten model samples.



Array Selector

If you plot an identified linear model array, **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.

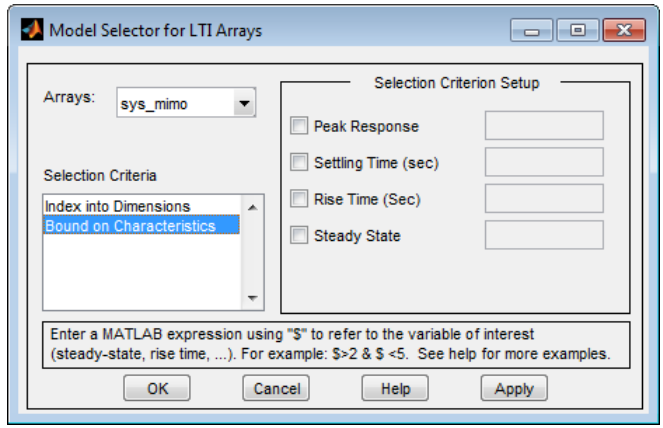
Arrays

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

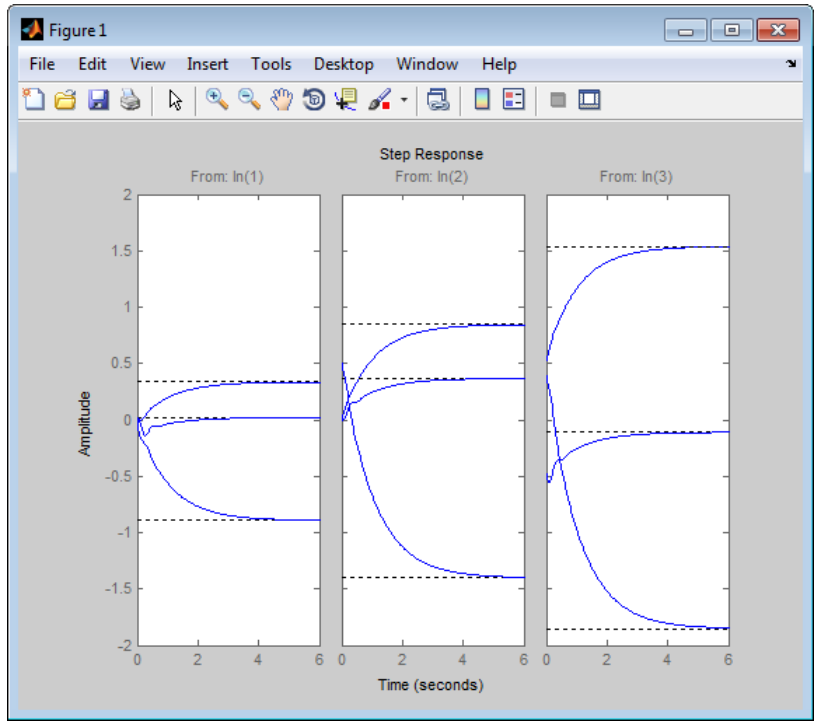


Use this option to select systems for inclusion or exclusion in your response plot based on their time response characteristics. The panel directly above the buttons describes how to set the inclusion or exclusion criteria based on which selection criteria you select from the reconfigured **Selection 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**.

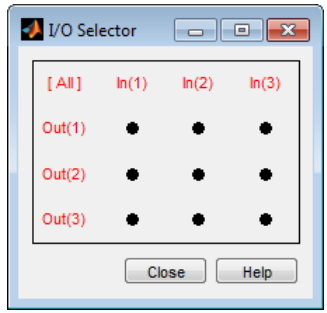
For example, if you select **Outputs**, the step plot reconfigures into 3 plots, grouping all the outputs together on each plot. Each plot now displays the responses from one of the inputs to all of the MIMO system's outputs, for all of the models in the array.



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” (Control System Toolbox)

Customize Response Plots Using the Response Plots Property Editor

In this section...
“Opening the Property Editor” on page 17-94 “Overview of Response Plots Property Editor” on page 17-95 “Labels Pane” on page 17-97 “Limits Pane” on page 17-98 “Units Pane” on page 17-98 “Style Pane” on page 17-108 “Options Pane” on page 17-110 “Editing Subplots Using the Property Editor” on page 17-114

Opening the Property Editor

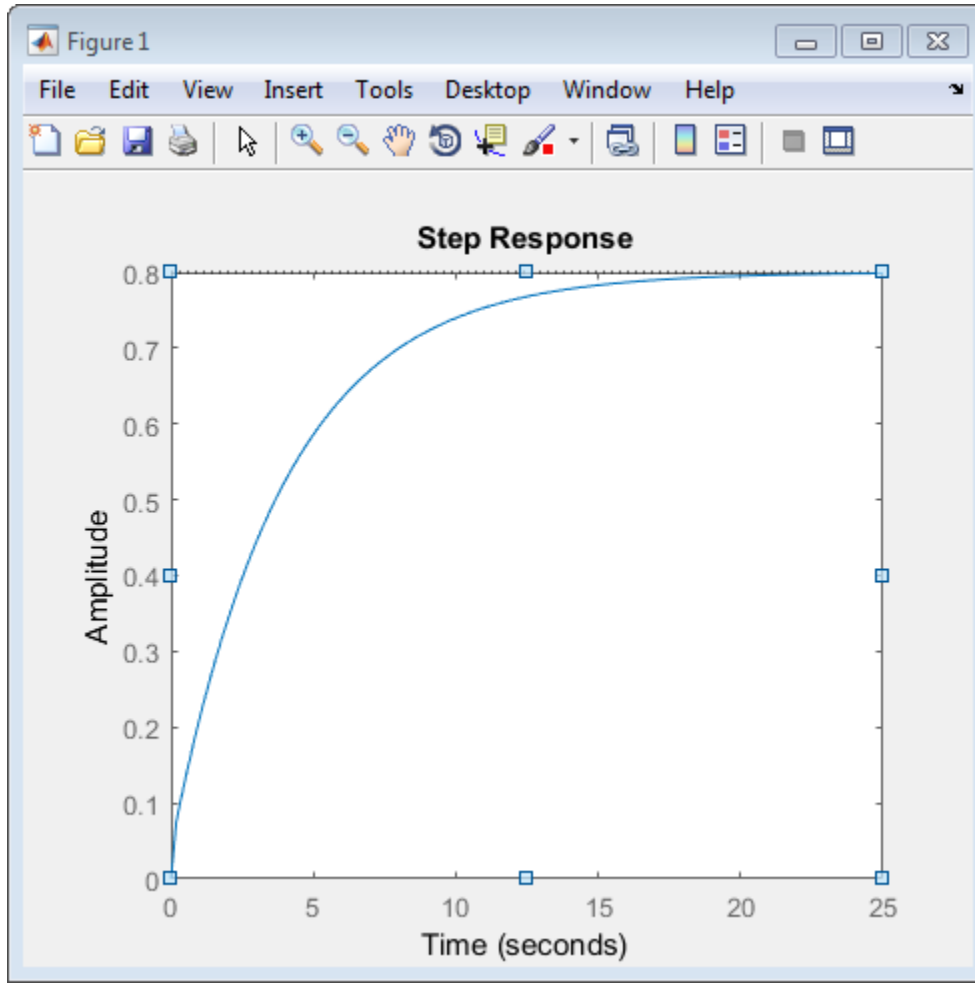
After you create a response plot, there are two ways to open the Property Editor:

- 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.

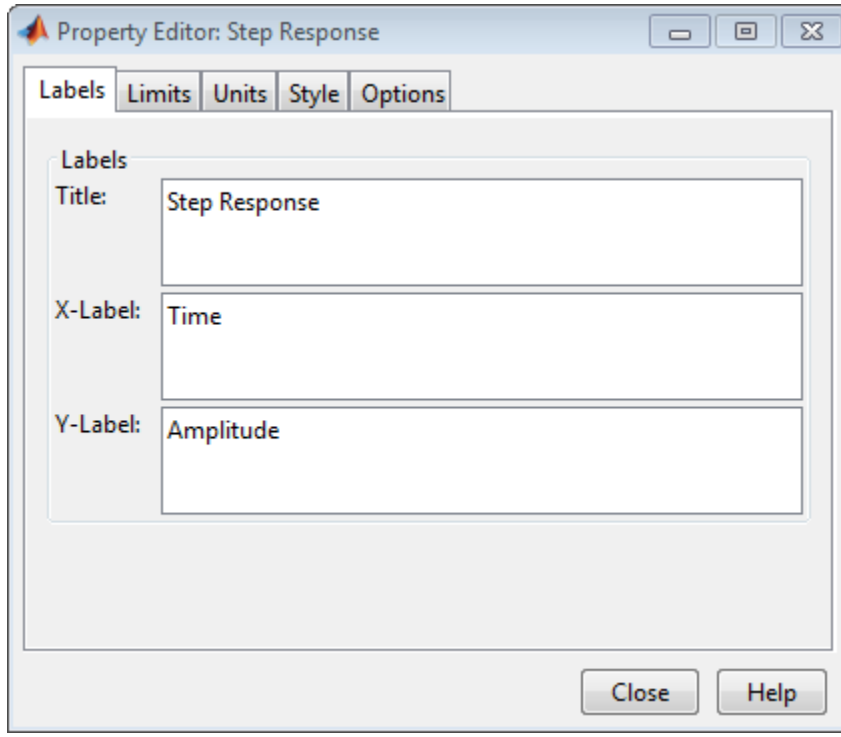
```
sys_dc = idtf([1 4],[1 20 5]);
step(sys_dc)
```

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

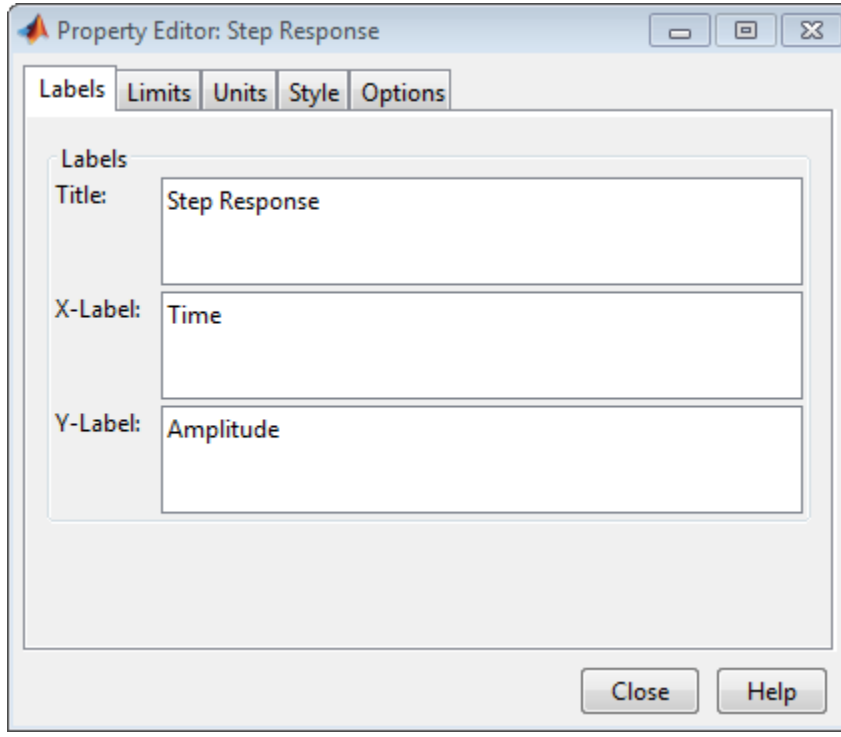
- Change options where applicable in the **Options** pane.

  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

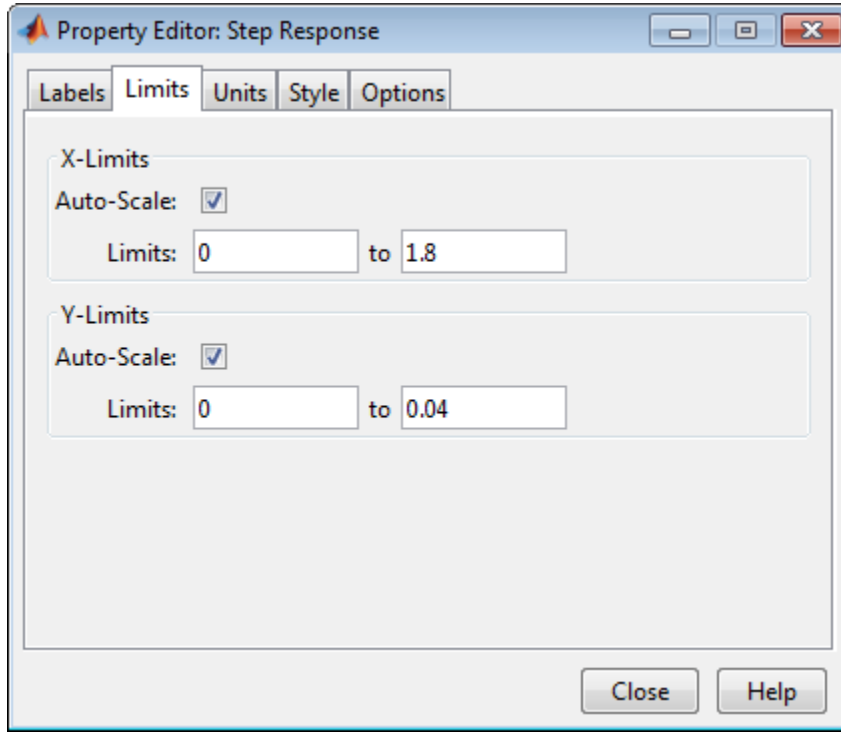
To specify new text for plot titles and axis labels, type the new names in the field next to the label you want to change. The label changes immediately as you type, so you can see how the new text looks as you are typing.



Limits Pane

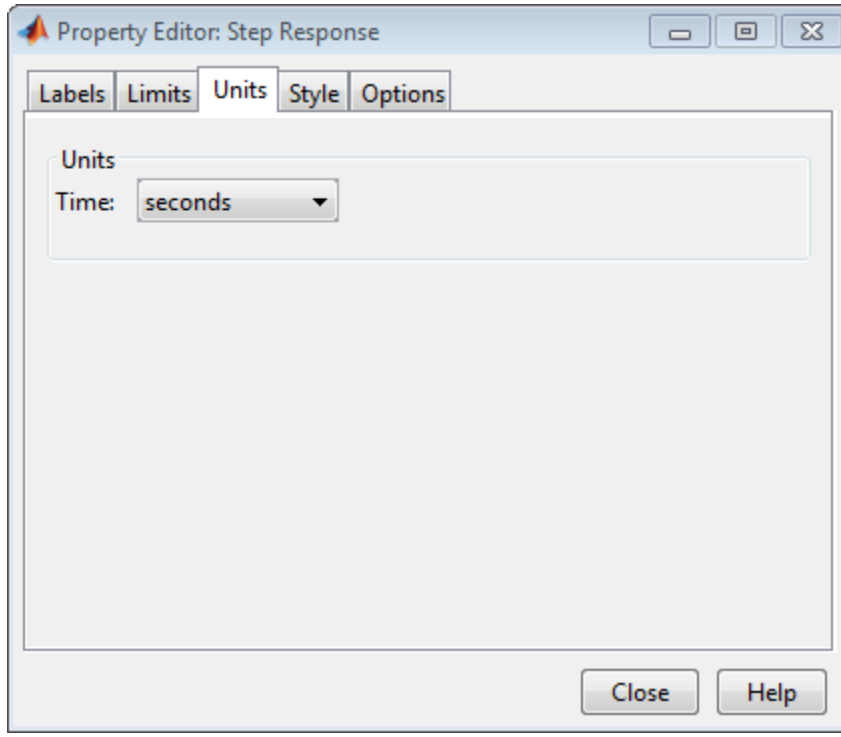
Default values for the axes limits make sure that the maximum and minimum x and y values are displayed. If you want to override the default settings, change the values in the Limits fields. The **Auto-Scale** box automatically clears if you click a different field. The new limits appear immediately in 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
Close
Help

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'`

Response Plot	Unit Conversions
	• `'cycles/year'` • Frequency scale is logarithmic or linear. • Magnitude in decibels (dB) or the absolute value • Phase in degrees or radians
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'`

Response Plot	Unit Conversions
	• 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'`

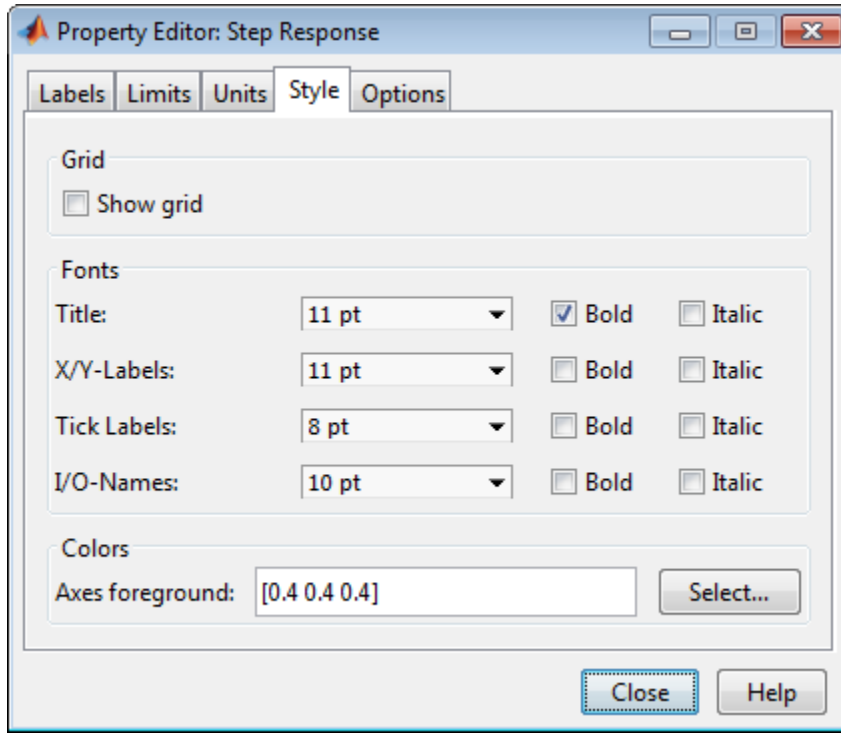
Response Plot	Unit Conversions
	• 'rad/millisecond' • 'rad/minute' • 'rad/hour' • 'rad/day' • 'rad/week' • 'rad/month' • 'rad/year' • 'cycles/nanosecond' • 'cycles/microsecond' • 'cycles/millisecond' • 'cycles/hour' • 'cycles/day' • 'cycles/week' • 'cycles/month' • 'cycles/year'

Response Plot	Unit Conversions
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'`

Response Plot	Unit Conversions
	• Frequency scale is logarithmic or linear. • Magnitude in decibels or the absolute value using logarithmic or linear scale
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.



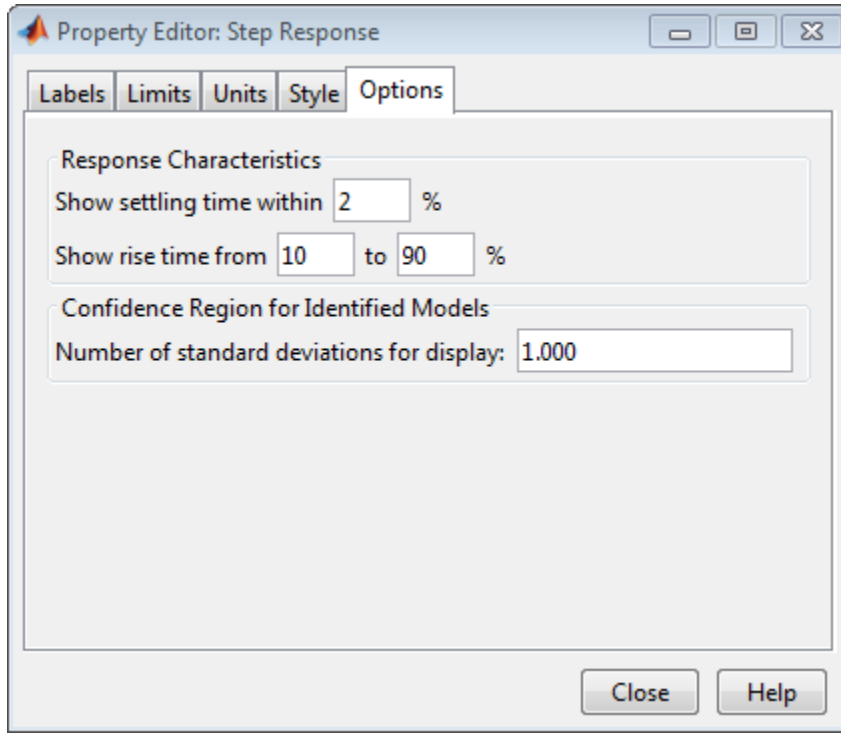
You have the following choices:

- **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**.

Plot	Customizable Feature
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**.
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.

Plot	Customizable Feature
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

- “Toolbox Preferences Editor” on page 18-2

Computing Model Uncertainty

In this section...
“Why Analyze Model Uncertainty?” on page 17-116
“What Is Model Covariance?” on page 17-116
“Types of Model Uncertainty Information” on page 17-117
“Definition of Confidence Interval for Specific Model Plots” on page 17-118

Why Analyze Model Uncertainty?

In addition to estimating model parameters, the toolbox algorithms also estimate variability of the model parameters that result from random disturbances in the output.

Understanding model variability helps you to understand how different your model parameters would be if you repeated the estimation using a different data set (with the same input sequence as the original data set) and the same model structure.

When validating your parametric models, check the uncertainty values. Large uncertainties in the parameters might be caused by high model orders, inadequate excitation, and poor signal-to-noise ratio in the data.

Note You can get model uncertainty data for linear parametric black-box models, and both linear and nonlinear grey-box models. Supported model objects include `idproc`, `idpoly`, `idss`, `idtf`, `idgrey`, `idfrd`, and `idnlgrey`.

What Is Model Covariance?

Uncertainty in the model is called *model covariance*.

When you estimate a model, the covariance matrix of the estimated parameters is stored with the model. Use `getcov` to fetch the covariance matrix. Use `getpvec` to fetch the list of parameters and their individual uncertainties that have been computed using the covariance matrix. The covariance matrix is used to compute all uncertainties in model output, Bode plots, residual plots, and pole-zero plots.

Computing the covariance matrix is based on the assumption that the model structure gives the correct description of the system dynamics. For models that include a

disturbance model *H*, a correct uncertainty estimate assumes that the model produces white residuals. To determine whether you can trust the estimated model uncertainty values, perform residual analysis tests on your model. For more details about residual analysis, see the topics on the "Residual Analysis" page. If your model passes residual analysis tests, there is a good chance that the true system lies within the confidence interval and any parameter uncertainties results from random disturbances in the output.

For output-error models, such as transfer function models, state-space with K=0 and polynomial models of output-error form, with the noise model *H* fixed to `1`, the covariance matrix computation does not assume white residuals. Instead, the covariance is estimated based on the estimated color of the residual correlations. This estimation of the noise color is also performed for state-space models with *K*=`0`, which is equivalent to an output-error model.

Types of Model Uncertainty Information

You can view the following uncertainty information from linear and nonlinear grey-box models:

- Uncertainties of estimated parameters.

  Type `present(model)` at the prompt, where `model` represents the name of a linear or nonlinear model.
- Confidence intervals on the linear model plots, including step-response, impulse-response, Bode, Nyquist, noise spectrum and pole-zero plots.

  Confidence intervals are computed based on the variability in the model parameters. For information about displaying confidence intervals, see "Definition of Confidence Interval for Specific Model Plots" on page 17-118.
- Covariance matrix of the estimated parameters in linear models and nonlinear grey-box models using `getcov`.
- Estimated standard deviations of polynomial coefficients, poles/zeros, or state-space matrices using `idssdata`, `tfdata`, `zpkdata`, and `polydata`.
- Simulated output values for linear models with standard deviations using `sim`.

  Call the `sim` command with output arguments, where the second output argument is the estimated standard deviation of each output value. For example, type `[ysim,ysimsd] = sim(model,data)`, where `ysim` is the simulated output,

`ysimsd` contains the standard deviations on the simulated output, and `data` is the simulation data.

- Perform Monte-Carlo analysis using `rsample` to generate a random sampling of an identified model in a given confidence region. An array of identified systems of the same structure as the input system is returned. The parameters of the returned models are perturbed about their nominal values in a way that is consistent with the parameter covariance.
- Simulate the effect of parameter uncertainties on a model's response using `simsd`.

Definition of Confidence Interval for Specific Model Plots

You can display the confidence interval on the following plot types:

Plot Type	Confidence Interval Corresponds to the Range of ...	More Information on Displaying Confidence Interval
Simulated and Predicted Output	Output values with a specific probability of being the actual output of the system.	Model Output Plots on page 17-16
Residuals	Residual values with a specific probability of being statistically insignificant for the system.	Residuals Plots on page 17-45
Impulse and Step	Response values with a specific probability of being the actual response of the system.	Impulse and Step Plots on page 17-57
Frequency Response	Response values with a specific probability of being the actual response of the system.	Frequency Response Plots on page 17-66
Noise Spectrum	Power-spectrum values with a specific probability of being the actual noise spectrum of the system.	Noise Spectrum Plots on page 17-74
Poles and Zeros	Pole or zero values with a specific probability of being the actual pole or zero of the system.	Pole-Zero Plots on page 17-74

Troubleshooting Model Estimation

In this section...

"About Troubleshooting Models" on page 17-119
"Model Order Is Too High or Too Low" on page 17-119
"Substantial Noise in the System" on page 17-120
"Unstable Models" on page 17-120
"Missing Input Variables" on page 17-121
"System Nonlinearities" on page 17-122
"Nonlinearity Estimator Produces a Poor Fit" on page 17-122

About Troubleshooting Models

During validation, models can exhibit undesirable characteristics or a poor fit to the validation data.

Use the tips in these sections to help improve your model performance. Some features, such as low signal-to-noise ratio, varying system properties, or nonstationary disturbances, can produce data for which a good model fit is not possible.

Model Order Is Too High or Too Low

A poor fit in the Model Output plot can be the result of an incorrect model order. System identification is largely a trial-and-error process when selecting model structure and model order. Ideally, you want the lowest-order model that adequately captures the system dynamics. High-order models are more expensive to compute and result in greater parameter uncertainty.

Start by estimating the model order as described in "Preliminary Step – Estimating Model Orders and Input Delays" on page 6-10. Use the suggested order as a starting point to estimate the lowest possible order with different model structures. After each estimation, monitor the Model Output and Residual Analysis plots, and then adjust your settings for the next estimation.

When a low-order model fits the validation data poorly, estimate a higher-order model to see if the fit improves. For example, if the Model Output plot shows that a fourth-order model gives poor results, estimate an eighth-order model. When a higher-order model

improves the fit, you can conclude that higher-order linear models are potentially sufficient for your application.

Use an independent data set to validate your models. If you use the same data set for both estimation and validation, the fit always improves as you increase the model order and you risk overfitting. However, if you use an independent data set to validate your model, the fit eventually deteriorates if the model orders are too high.

Substantial Noise in the System

Substantial noise in your system can result in a poor model fit. The presence of such noise is indicated when:

- A state-space model produces a better fit than an ARX model. While a state-space structure has sufficient flexibility to model noise, an ARX structure is unable to independently model noise and system dynamics. The following ARX model equation shows that *A* couples the dynamics and the noise terms by appearing in the denominator of both:

  $$y = \frac{B}{A}u + \frac{1}{A}e$$

- A residual analysis plot shows significant autocorrelation of residuals at nonzero lags. For more information about residual analysis, see the topics on the "Residual Analysis" page.

To model noise more carefully, use either an ARMAX or the Box-Jenkins model structure, both of which model the noise and dynamics terms using different polynomials.

Unstable Models

Unstable Linear Model

You can test whether a *linear model* is unstable is by examining the pole-zero plot of the model, which is described in "Pole and Zero Plots" on page 17-81. The stability threshold for pole values differs for discrete-time and continuous-time models, as follows:

- For stable continuous-time models, the real part of the pole is less than 0.
- For stable discrete-time models, the magnitude of the pole is less than 1.

Note Linear trends in estimation data can cause the identified linear models to be unstable. However, detrending the model does not guarantee stability.

If your model is unstable, but you believe that your system is stable, you can.

- Force stability during estimation — Set the `Focus` estimation option to a value that guarantees a stable model. This setting can result in reduced model quality.
- Allow for some instability — Set the stability threshold advanced estimation option to allow for a margin of error:
  - For continuous-time models, set the value of `Advanced.StabilityThreshold.s`. The model is considered stable if the pole on the far right is to the left of *s*.
  - For discrete-time models, set the value of `Advanced.StabilityThreshold.z`. The model is considered stable if all of the poles are inside a circle with a radius of *z* that is centered at the origin.

For more information about `Focus` and `Advanced.StabilityThreshold`, see the various commands for creating estimation option sets, such as `tfestOptions`, `ssestOptions`, and`procestOptions`.

Unstable Nonlinear Models

To test if a *nonlinear model* is unstable, plot the simulated model output on top of the validation data. If the simulated output diverges from measured output, the model is unstable. However, agreement between model output and measured output does not guarantee stability.

When an Unstable Model Is OK

In some cases, an unstable model is still useful. For example, if your system is unstable without a controller, you can use your model for control design. In this case, you can import the unstable model into Simulink or Control System Toolbox products.

Missing Input Variables

If modeling noise and trying different model structures and orders still results in a poor fit, try adding more inputs that can affect the output. Inputs do not need to be control signals. Any measurable signal can be considered an input, including measurable disturbances.

Include additional measured signals in your input data, and estimate the model again.

System Nonlinearities

If a linear model shows a poor fit to the validation data, consider whether nonlinear effects are present in the system.

You can model the nonlinearities by performing a simple transformation on the input signals to make the problem linear in the new variables. For example, in a heating process with electrical power as the driving stimulus, you can multiply voltage and current measurements to create a power input signal.

If your problem is sufficiently complex and you do not have physical insight into the system, try fitting nonlinear black-box models to your data, see "About Identified Nonlinear Models" on page 11-2.

Nonlinearity Estimator Produces a Poor Fit

For nonlinear ARX and Hammerstein-Wiener models, the Model Output plot does not show a good fit when the nonlinearity estimator has incorrect complexity.

Specify the complexity of piece-wise-linear, wavelet, sigmoid, and custom networks using the `NumberOfUnits` nonlinear estimator property. A higher number of units indicates a more complex nonlinearity estimator. When using neural networks, specify the complexity using the parameters of the network object. For more information, see the Neural Network Toolbox documentation.

To select the appropriate nonlinearity estimator complexity, first validate the output of a low-complexity model. Next, increase the model complexity and validate the output again. The model fit degrades when the nonlinearity estimator becomes too complex. This degradation in performance is only visible if you use independent estimation and validation data sets

See Also

More About

- "Ways to Validate Models" on page 17-3

- “Preliminary Step – Estimating Model Orders and Input Delays” on page 6-10
- “Pole and Zero Plots” on page 17-81
- “What Is Residual Analysis?” on page 17-43
- “Next Steps After Getting an Accurate Model” on page 17-124

Next Steps After Getting an Accurate Model

For linear parametric models, you can perform the following operations:

- Transform between continuous-time and discrete-time representation.

  See "Transforming Between Discrete-Time and Continuous-Time Representations" on page 4-17.
- Transform between linear model representations, such as between polynomial, state-space, and zero-pole representations.

  See "Transforming Between Linear Model Representations" on page 4-35.
- Extract numerical data from transfer functions, pole-zero models, and state-space matrices.

  See "Extracting Numerical Model Data" on page 4-14.

For nonlinear black-box models (`idnlarx` and `idnlhw` objects), you can compute a linear approximation of the nonlinear model. See "Linear Approximation of Nonlinear Black-Box Models" on page 11-64.

System Identification Toolbox models in the MATLAB workspace are immediately available to other MathWorks® products. However, if you used the System Identification app to estimate models, you must first export the models to the MATLAB workspace.

Tip To export a model from the app, drag the model icon to the **To Workspace** rectangle. Alternatively, right-click the model to open the Data/model Info dialog box. Click **Export**.

If you have the Control System Toolbox software installed, you can import your linear plant model for control-system design. For more information, see "Using Identified Models for Control Design Applications" on page 19-2.

Finally, if you have Simulink software installed, you can exchange data between the System Identification Toolbox software and the Simulink environment. For more information, see "Simulating Identified Model Output in Simulink" on page 20-5.

Setting Toolbox Preferences

Toolbox Preferences Editor

In this section...
“Overview of the Toolbox Preferences Editor” on page 18-2
“Opening the Toolbox Preferences Editor” on page 18-2
“Units Pane” on page 18-3
“Style Pane” on page 18-5
“Options Pane” on page 18-7
“Control System Designer Pane” on page 18-7

Overview of the Toolbox Preferences Editor

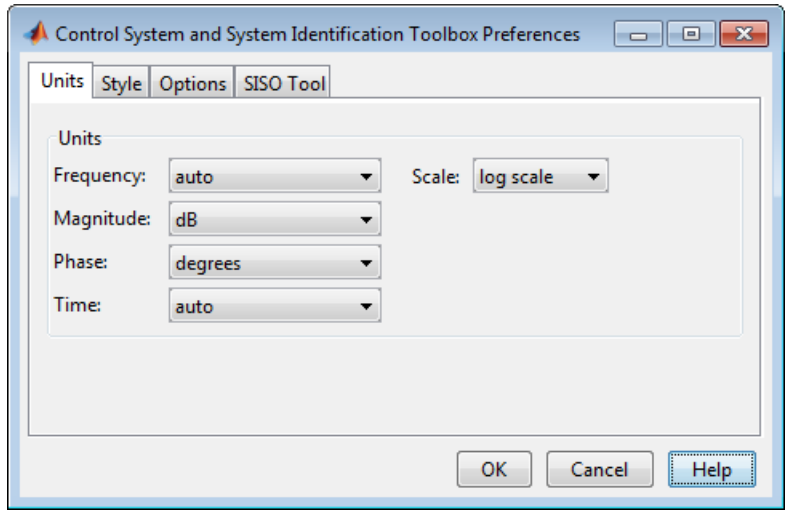
The Toolbox Preferences editor allows you to set plot preferences that will persist from session to session.

Opening the Toolbox Preferences Editor

To open the Toolbox Preferences editor, select **Toolbox Preferences** from the **File** menu of the **Linear System Analyzer** or the **Control System Designer**. Alternatively, you can type

```
identpref
```

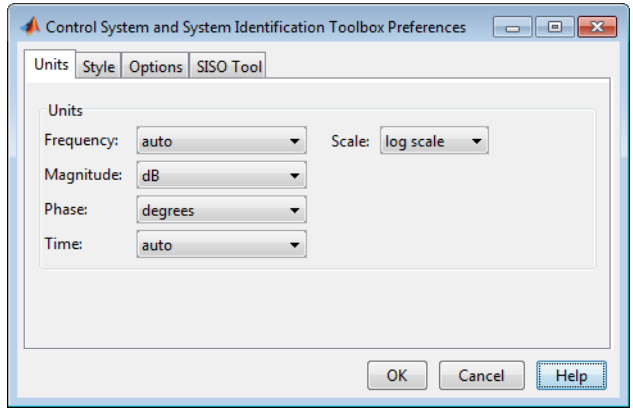
at the MATLAB prompt.



Control System Toolbox Preferences Editor

Note **Control System Designer** requires the Control System Toolbox software.

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**

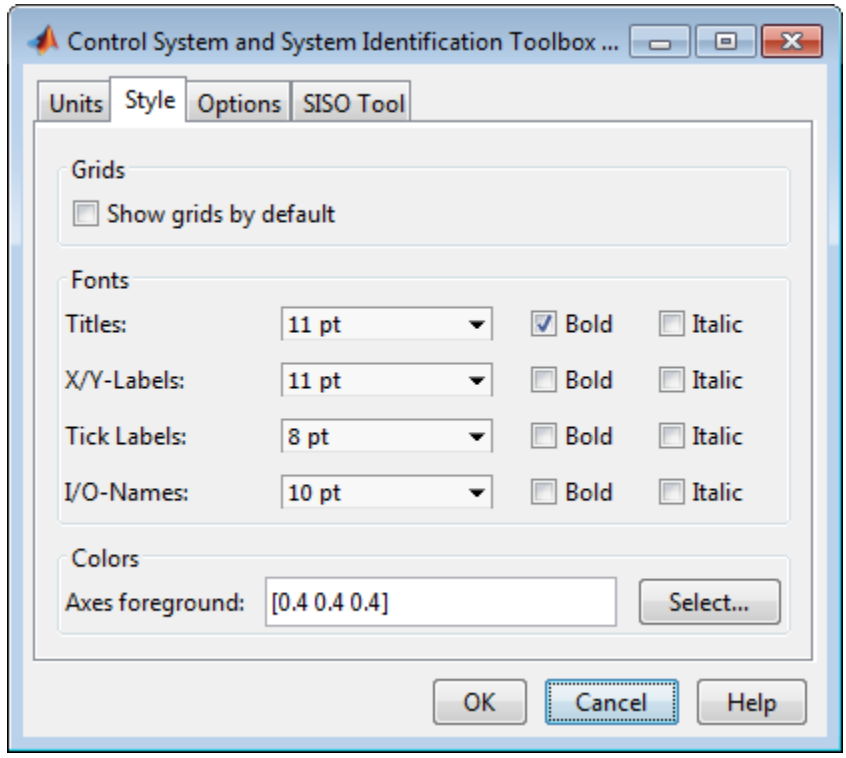
  The default `auto` option uses the time units specified in the `TimeUnit` property of the system on the time- and frequency-domain plots. For multiple systems with different time units, the units of the first system is used.

  #### 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.



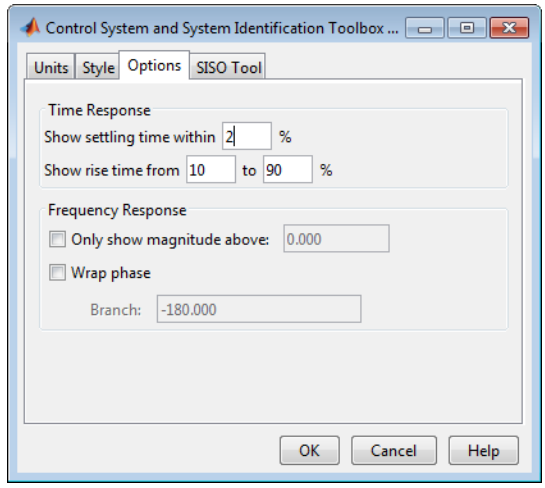
You have the following choices:

- **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:

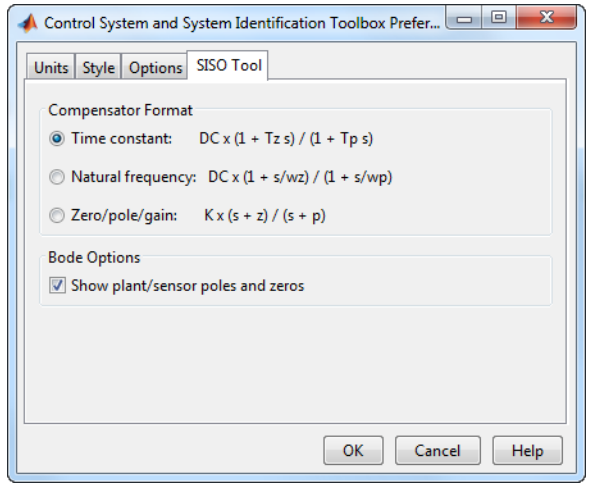
- **Show settling time within** *xx*% — Set the threshold of the settling time calculation to any percentage from 0 to 100%. The default is 2%.
- **Specify rise time from** *xx*% to *yy*%— The standard definition of rise time is the time it takes the signal to go from 10% to 90% of the final value. Specify 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).

Control System Designer Pane

The Control System Designer pane has settings for **Control System Designer**. This figure shows the Control System Designer 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

$$K \times \frac{\left(1+T_{z_1} s\right)\left(1+T_{z_2} s\right)}{\left(1+T_{p_1} s\right)\left(1+T_{p_2} s\right)} \cdots$$

  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

$$K \times \frac{\left(1+s/\omega_{z_1}\right)\left(1+s/\omega_{z_2}\right)}{\left(1+s/\omega_{p_1}\right)\left(1+s/\omega_{p_2}\right)} \cdots$$

  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

$$K \times \frac{\left(s+z_1\right)\left(s+z_2\right)}{\left(s+p_1\right)\left(s+p_2\right)}$$

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.

Control Design Applications

- "Using Identified Models for Control Design Applications" on page 19-2
- "Create and Plot Identified Models Using Control System Toolbox Software" on page 19-7

Using Identified Models for Control Design Applications

In this section...

"How Control System Toolbox Software Works with Identified Models" on page 19-2
"Using balred to Reduce Model Order" on page 19-2
"Compensator Design Using Control System Toolbox Software" on page 19-3
"Converting Models to LTI Objects" on page 19-3
"Viewing Model Response Using the Linear System Analyzer" on page 19-4
"Combining Model Objects" on page 19-5

How Control System Toolbox Software Works with Identified Models

System Identification Toolbox software integrates with Control System Toolbox software by providing a plant for control design.

Control System Toolbox software also provides the Linear System Analyzer to extend System Identification Toolbox functionality for linear model analysis.

Control System Toolbox software supports only linear models. If you identified a nonlinear plant model using System Identification Toolbox software, you must linearize it before you can work with this model in the Control System Toolbox software. For more information, see the `linapp`, `idnlarx/linearize`, or `idnlhw/linearize` reference page.

Note You can only use the System Identification Toolbox software to linearize nonlinear ARX (`idnlarx`) and Hammerstein-Wiener (`idnlhw`) models. Linearization of nonlinear grey-box (`idnlgrey`) models is not supported.

Using balred to Reduce Model Order

In some cases, the order of your identified model might be higher than necessary to capture the dynamics. If you have the Control System Toolbox software, you can use `balred` to compute a state-spate model approximation with a reduced model order.

To learn how you can reduce model order using pole-zero plots, see "Reducing Model Order Using Pole-Zero Plots" on page 17-84.

Compensator Design Using Control System Toolbox Software

After you estimate a plant model using System Identification Toolbox software, you can use Control System Toolbox software to design a controller for this plant.

System Identification Toolbox models in the MATLAB workspace are immediately available to Control System Toolbox commands. However, if you used the System Identification app to estimate models, you must first export the models to the MATLAB workspace. To export a model from the app, drag the model icon to the **To Workspace** rectangle. Alternatively, right-click the icon to open the Data/model Info dialog box. Click **Export** to export the model.

Control System Toolbox software provides both the **Control System Designer** and commands for working at the command line. You can import linear models directly into **Control System Designer** using the following command:

```
controlSystemDesigner(model)
```

You can also identify a linear model from measured SISO data and tune a PID controller for the resulting model in the PID Tuner. You can interactively adjust the identified parameters to obtain an LTI model whose response fits your response data. The PID Tuner automatically tunes a PID controller for the identified model. You can then interactively adjust the performance of the tuned control system, and save the identified plant and tuned controller. To access the PID Tuner, enter `pidTuner` at the MATLAB command line. For more information, see “PID Controller Tuning” (Control System Toolbox).

Converting Models to LTI Objects

You can convert linear identified models into numeric LTI models (`ss`, `tf`, `zpk`) of Control System Toolbox software.

The following table summarizes the commands for transforming linear state-space and polynomial models to an LTI object.

Commands for Converting Models to LTI Objects

Command	Description	Example
`frd`	Convert to frequency-response representation.	`ss_sys = frd(model)`
`ss`	Convert to state-space representation.	`ss_sys = ss(model)`
`tf`	Convert to transfer-function form.	`tf_sys = tf(model)`
`zpk`	Convert to zero-pole form.	`zpk_sys = zpk(model)`

The following code converts the noise component of a linear identified model, `sys`, to a numeric state-space model:

```
noise_model_ss = idss(sys,'noise');
```

To convert both the measured and noise components of a linear identified model, `sys`, to a numeric state-space model:

```
model_ss = idss(sys,'augmented');
```

For more information about subreferencing the dynamic or the noise model, see “Separation of Measured and Noise Components of Models” on page 4-39.

Viewing Model Response Using the Linear System Analyzer

- “What Is the Linear System Analyzer?” on page 19-4
- “Displaying Identified Models in the Linear System Analyzer” on page 19-5

What Is the Linear System Analyzer?

If you have the Control System Toolbox software, you can plot models in the Linear System Analyzer from either the System Identification app or the MATLAB Command Window.

The Linear System Analyzer is a graphical user interface for viewing and manipulating the response plots of linear models.

Note The Linear System Analyzer does not display model uncertainty.

For more information about working with plots in the Linear System Analyzer, see the "Linear System Analyzer Overview" (Control System Toolbox).

Displaying Identified Models in the Linear System Analyzer

When the MATLAB software is installed, the System Identification app contains the **To LTI Viewer** rectangle. To plot models in the Linear System Analyzer, do one of the following:

- Drag and drop the corresponding icon to the **To LTI Viewer** rectangle in the System Identification app.
- Right-click the icon to open the Data/model Info dialog box. Click **Show in LTI Viewer** to plot the model in the Linear System Analyzer.

Alternatively, use the following syntax when working at the command line to view a model in the Linear System Analyzer:

```
linearSystemAnalyzer(model)
```

Combining Model Objects

If you have the Control System Toolbox software, you can combine linear model objects, such as `idtf`, `idgrey`, `idpoly`, `idproc`, and `idss` model objects, similar to the way you combine LTI objects. The result of these operations is a numeric LTI model that belongs to the Control System Toolbox software. The only exceptions are the model stacking and model concatenation operations, which deliver results as identified models.

For example, you can perform the following operations on identified models:

- `G1+G2`
- `G1*G2`
- `append(G1,G2)`
- `feedback(G1,G2)`

See Also

Related Examples

- “Create and Plot Identified Models Using Control System Toolbox Software” on page 19-7

Create and Plot Identified Models Using Control System Toolbox Software

This example shows how to create and plot models using the System Identification Toolbox software and Control System Toolbox software. The example requires a Control System Toolbox license.

Construct a random numeric model using the Control System Toolbox software.

```
rng('default');
sys0 = drss(3,3,2);
```

`rng('default')` specifies the setting of the random number generator as its default setting.

`sys0` is a third-order numeric state-space model with three outputs and two inputs.

Convert `sys0` to an identified state-space model and set its output noise variance.

```
sys = idss(sys0);
sys.NoiseVariance = 0.1*eye(3);
```

Generate input data for simulating the output.

```
u = iddata([],idinput([800 2],'rbs'));
```

Simulate the model output with added noise.

```
opt = simOptions('AddNoise',true);
y = sim(sys,u,opt);
```

`opt` is an option set specifying simulation options. `y` is the simulated output for `sys0`.

Create an input-output (`iddata`) object.

```
data = [y u];
```

Estimate the state-space model from the generated data using `ssest` .

```
estimated_ss = ssest(data(1:400));
```

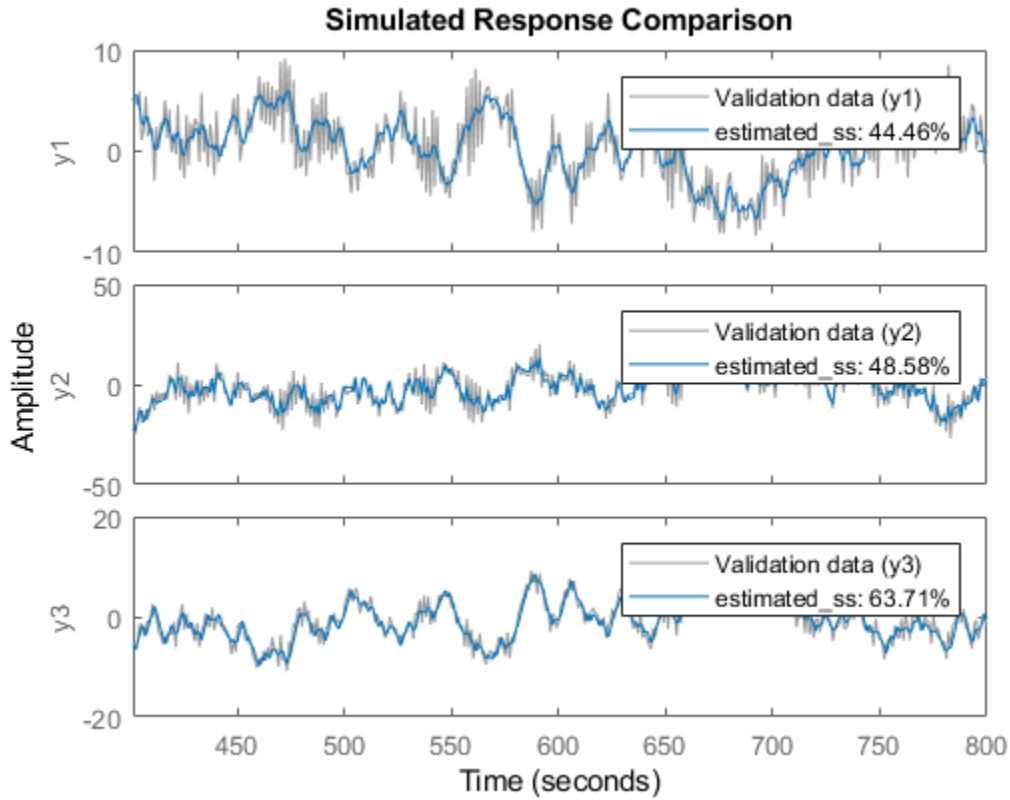
`estimated_ss` is an identified state-space model.

Convert the identified state-space model to a numeric transfer function.

```
sys_tf = tf(estimated_ss);
```

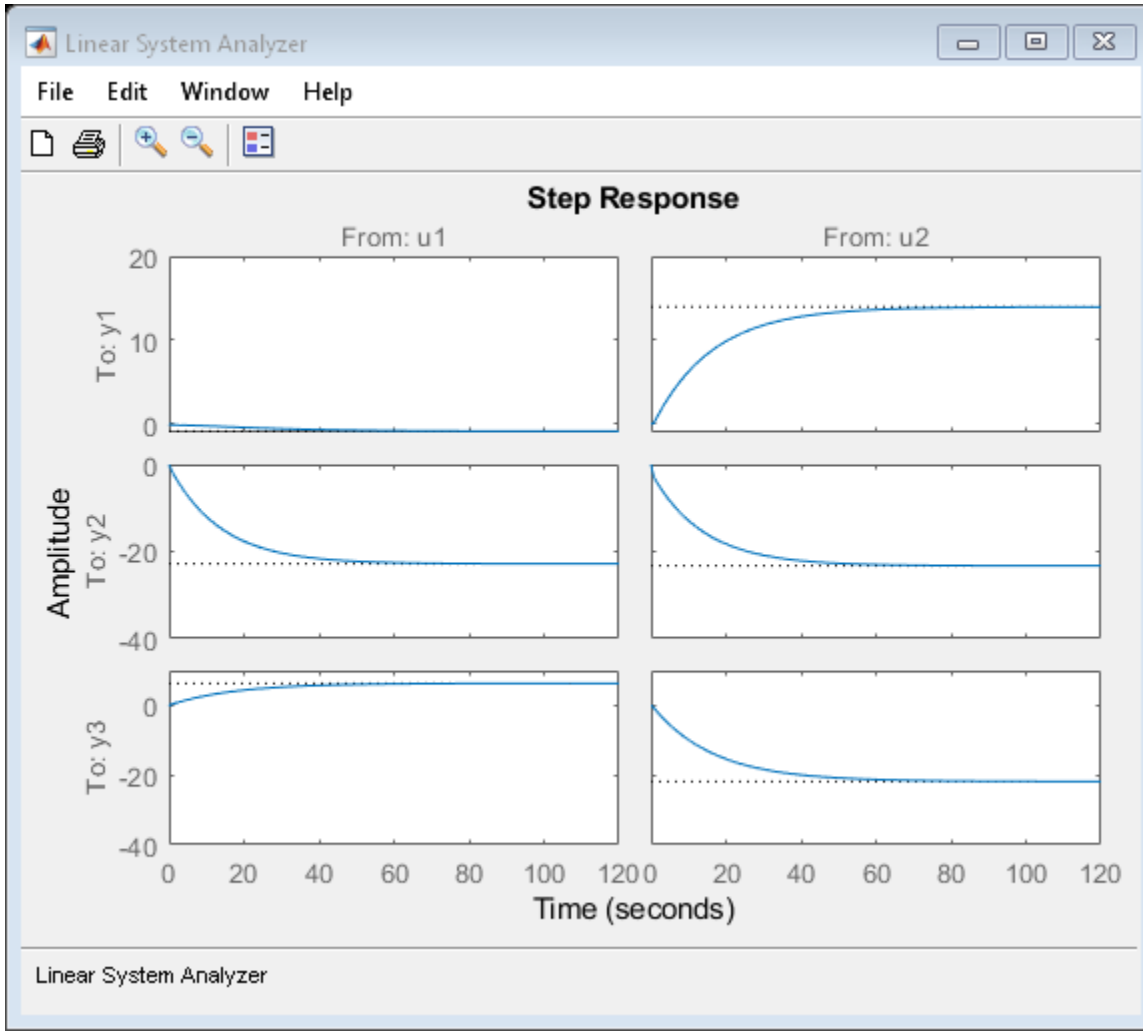
Plot the model output for identified state-space model.

```
compare(data(401:800),estimated_ss)
```



Plot the response of identified model using the Linear System Analyzer.

```
linearSystemAnalyzer(estimated_ss);
```



See Also

More About

- “Using Identified Models for Control Design Applications” on page 19-2

System Identification Toolbox Blocks

- "Using System Identification Toolbox Blocks in Simulink Models" on page 20-2
- "Preparing Data" on page 20-3
- "Identifying Linear Models" on page 20-4
- "Simulating Identified Model Output in Simulink" on page 20-5

Using System Identification Toolbox Blocks in Simulink Models

System Identification Toolbox software provides blocks for sharing information between the MATLAB and Simulink environments.

You can use the System Identification Toolbox block library to perform the following tasks:

- Stream time-domain data source (`iddata` object) into a Simulink model.
- Export data from a simulation in Simulink software as a System Identification Toolbox data object (`iddata` object).
- Import estimated models into a Simulink model, and simulate the models with or without noise.

  The model you import might be a component of a larger system modeled in Simulink. For example, if you identified a plant model using the System Identification Toolbox software, you can import this plant into a Simulink model for control design.
- Estimate parameters of linear polynomial models during simulation from single-output data.

To open the System Identification Toolbox block library, enter `slLibraryBrowser` at the MATLAB prompt. In the Library Browser, select **System Identification Toolbox**.

You can also open the System Identification Toolbox block library directly by typing the following command at the MATLAB prompt:

```
slident
```

To get help on a block, right-click the block in the Library Browser, and select **Help**.

See Also

More About

- “Simulating Identified Model Output in Simulink” on page 20-5

Preparing Data

The following table summarizes the blocks you use to transfer data between the MATLAB and Simulink environments.

After you add a block to the Simulink model, double-click the block to specify block parameters. For an example of bringing data into a Simulink model, see the tutorial on estimating process models in the *System Identification Toolbox Getting Started Guide*.

Block	Description
Iddata Sink	Export input and output signals to the MATLAB workspace as an `iddata` object.
Iddata Source	Import `iddata` object from the MATLAB workspace. Input and output ports of the block correspond to input and output signals of the data. These inputs and outputs provide signals to blocks that are connected to this data block.

For information about configuring each block, see the corresponding reference pages.

Identifying Linear Models

The following table summarizes the blocks you use to recursively estimate model parameters in a Simulink model during simulation and export the results to the MATLAB environment.

After you add a block to the model, double-click the block to specify block parameters.

Block	Description
Recursive Least Squares Estimator	Estimate model coefficients using recursive least squares (RLS) algorithm
Recursive Polynomial Model Estimator	Estimate input-output and time-series model coefficients
Kalman Filter	Estimate states of discrete-time or continuous-time linear system
Model Type Converter	Convert polynomial model coefficients to state-space model matrices

For information about configuring each block, see the corresponding reference pages.

Simulating Identified Model Output in Simulink

In this section...
“When to Use Simulation Blocks” on page 20-5 “Summary of Simulation Blocks” on page 20-5 “Specifying Initial Conditions for Simulation” on page 20-6 “Simulate Identified Model Using Simulink Software” on page 20-7

When to Use Simulation Blocks

Add model simulation blocks to your Simulink model from the System Identification Toolbox block library when you want to:

- Represent the dynamics of a physical component in a Simulink model using a data-based nonlinear model.
- Replace a complex Simulink subsystem with a simpler data-based nonlinear model.

You use the model simulation blocks to import the models you identified using System Identification Toolbox software from the MATLAB workspace into the Simulink environment. For a list of System Identification Toolbox simulation blocks, see “Summary of Simulation Blocks” on page 20-5.

Summary of Simulation Blocks

The following table summarizes the blocks you use to import models from the MATLAB environment into a Simulink model for simulation. Importing a model corresponds to entering the model variable name in the block parameter dialog box.

Block	Description
Idmodel	Simulate a linear identified model in Simulink software. The model can be a process (`idproc`), linear polynomial (`idpoly`), state-space (`idss`), grey-box (`idgrey`) and transfer-function (`idtf`) model.
Nonlinear ARX Model	Simulate `idnlarx` model in Simulink.
Hammerstein-Wiener Model	Simulate `idnlhw` model in Simulink.

Block	Description
Nonlinear Grey-Box Model	Simulate nonlinear ODE (`idnlgrey` model object) in Simulink.

After you import the model into Simulink software, use the block parameter dialog box to specify the initial conditions for simulating that block. (See “Specifying Initial Conditions for Simulation” on page 20-6.) For information about configuring each block, see the corresponding reference pages.

Specifying Initial Conditions for Simulation

For accurate simulation of a linear or a nonlinear model, you can use default initial conditions or specify the initial conditions for simulation using the block parameters dialog box.

- “Specifying Initial States of Linear Models” on page 20-6
- “Specifying Initial States of Nonlinear ARX Models” on page 20-7
- “Specifying Initial States of Hammerstein-Wiener Models” on page 20-7

Specifying Initial States of Linear Models

Specify the initial states for simulation in the **Initial states (state space only: idss, idgrey)** field of the Function Block Parameters: Idmodel dialog box:

- For `idss` and `idgrey` models, initial states must be a vector of length equal to the order of the model.
- For models other than `idss` and `idgrey`, initial conditions are zero.
- In some situations, you may want to match the simulated response of the model to a certain input/output data set:

  1. Convert the identified model into state-space form (`idss` model), and use the state-space model in the block.
  2. Compute the initial state values that produce the best fit between the model output and the measured output signal using `findstates`.
  3. Specify the same input signal for simulation that you used as the validation data in the app or in the `compare` plot.

     For example:

```
% Convert to state-space model
mss = idss(m);
% Estimate initial states from data
X0 = findstates(mss,z);
```

`z` is the data set you used for validating the model `m`. Use the model `mss` and initial states `X0` in the Idmodel block to perform the simulation.

Specifying Initial States of Nonlinear ARX Models

The states of a nonlinear ARX model correspond to the dynamic elements of the nonlinear ARX model structure, which are the model regressors. *Regressors* can be the delayed input/output variables (standard regressors) or user-defined transformations of delayed input/output variables (custom regressors). For more information about the states of a nonlinear ARX model, see the `idnlarx` reference page.

For simulating nonlinear ARX models, you can specify the initial conditions as input/output values, or as a vector. For more information about specifying initial conditions for simulation, see the IDNLARX Model reference page.

Specifying Initial States of Hammerstein-Wiener Models

The states of a Hammerstein-Wiener model correspond to the states of the embedded linear (`idpoly` or `idss`) model. For more information about the states of a Hammerstein-Wiener model, see the `idnlhw` reference page.

The default initial state for simulating a Hammerstein-Wiener model is 0. For more information about specifying initial conditions for simulation, see the IDNLHW Model reference page.

Simulate Identified Model Using Simulink Software

This example shows how to set the initial states for simulating a model such that the simulation provides a best fit to measured input-output data.

Prerequisites

Estimate a model, `M`, using a multiple-experiment data set, `Z`, which contains data from three experiments — `z1`, `z2`, and `z3`:

```
% Load multi-experiment data.
load(fullfile(matlabroot,'toolbox','ident','iddemos',...
```

```
'data', 'twobodiesdata'));

% Create an iddata object to store the multi-experiment data.
z1=iddata(y1,u1,0.005,'Tstart',0);
z2=iddata(y2,u2,0.005,'Tstart',0);
z3=iddata(y3,u3,0.005,'Tstart',0);
Z = merge(z1,z2,z3);

% Estimate a 5th order state-space model.
opt = n4sidOptions('Focus','simulation');
[M,x0] = n4sid(Z,5,opt);
```

To simulate the model using input `u2`, use `x0(:,2)` as the initial states. `x0(:,2)` is computed to maximize the fit between the measured output, `y2`, and the response of `M`.

To compute initial states that maximizes the fit to the corresponding output `y2`, and simulate the model using the second experiment:

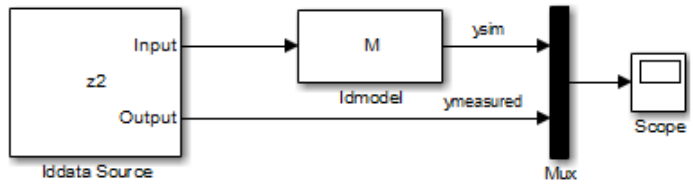
1. Extract the initial states that correspond to the second experiment for simulation.

   ```
   X0est = x0(:,2);
   ```

2. Open the System Identification Toolbox library by typing the following command at the MATLAB prompt.

   ```
   slident
   ```

3. Open a new Simulink model window. Then, drag and drop an `Idmodel` block from the library into the model window.
4. Open the Function Block Parameters dialog box by double-clicking the `Idmodel` block. Specify the following block parameters:
   a. In the **Model variable** field, type `M` to specify the estimated model.
   b. In the **Initial state** field, type `X0est` to specify the estimated initial states. Click **OK**.
5. Drag and drop an `Iddata Source` block into the model window. Then, configure the model, as shown in the following figure.



6 Simulate the model for 2 seconds, and compare the simulated output `ysim` with the measured output `ymeasured` using the Scope block.

See Also

IDNLARX Model | IDNLHW Model

More About

- "Simulating and Predicting Model Output" on page 17-9

System Identification App

- “Steps for Using the System Identification App” on page 21-2
- “Working with System Identification App” on page 21-3

Steps for Using the System Identification App

A typical workflow in the **System Identification** app includes the following steps:

1. Import your data into the MATLAB workspace, as described in "Representing Data in MATLAB Workspace" on page 2-9.
2. Start a new session in the System Identification app, or open a saved session. For more information, see "Starting a New Session in the App" on page 21-3.
3. Import data into the app from the MATLAB workspace. For more information, see "Represent Data".
4. Plot and preprocess data to prepare it for system identification. For example, you can remove constant offsets or linear trends (for linear models only), filter data, or select data regions of interest. For more information, see "Preprocess Data".
5. Specify the data for estimation and validation. For more information, see "Specify Estimation and Validation Data in the App" on page 2-30.
6. Select the model type to estimate using the **Estimate** menu.
7. Validate models. For more information, see "Model Validation".
8. Export models to the MATLAB workspace for further analysis. For more information, see "Exporting Models from the App to the MATLAB Workspace" on page 21-10.

Working with System Identification App

In this section...
“Starting and Managing Sessions” on page 21-3
“Managing Models” on page 21-7
“Working with Plots” on page 21-11
“Customizing the System Identification App” on page 21-15

Starting and Managing Sessions

- “What Is a System Identification Session?” on page 21-3
- “Starting a New Session in the App” on page 21-3
- “Description of the System Identification App Window” on page 21-4
- “Opening a Saved Session” on page 21-6
- “Saving, Merging, and Closing Sessions” on page 21-6
- “Deleting a Session” on page 21-7

What Is a System Identification Session?

A *session* represents the total progress of your identification process, including any data sets and models in the **System Identification** app.

You can save a session to a file with a `.sid` extension. For example, you can save different stages of your progress as different sessions so that you can revert to any stage by simply opening the corresponding session.

To start a new session, see “Starting a New Session in the App” on page 21-3.

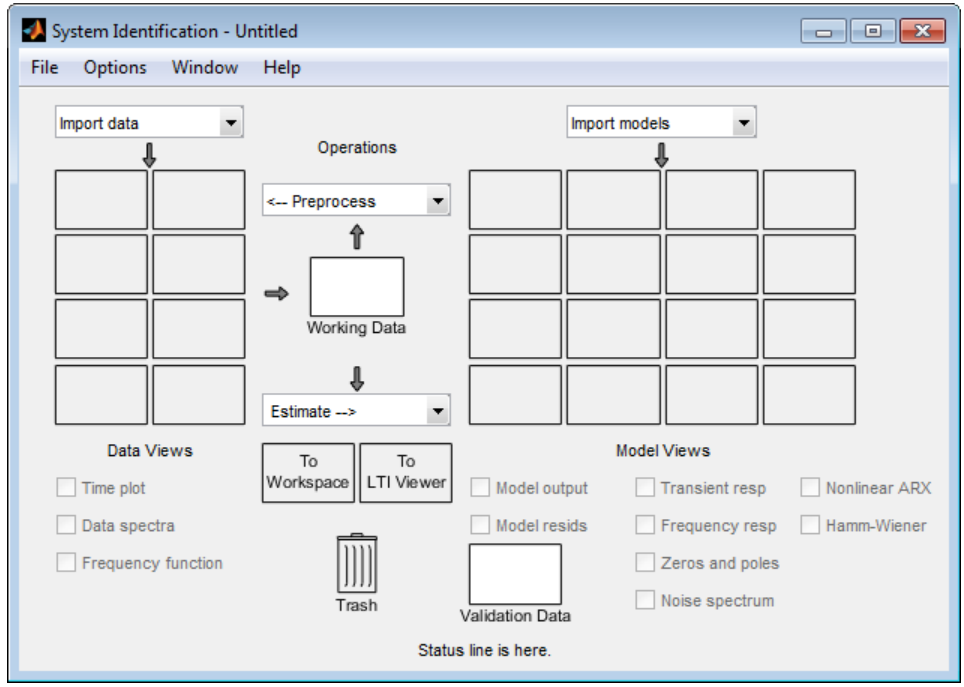
For more information about the steps for using the System Identification app, see “Steps for Using the System Identification App” on page 21-2.

Starting a New Session in the App

To start a new session in the System Identification app, type `systemIdentification` in the MATLAB Command Window:

```
systemIdentification
```

Alternatively, you can start a new session by selecting the **Apps** tab of MATLAB desktop. In the **Apps** section, click **System Identification**. This action opens the System Identification app.

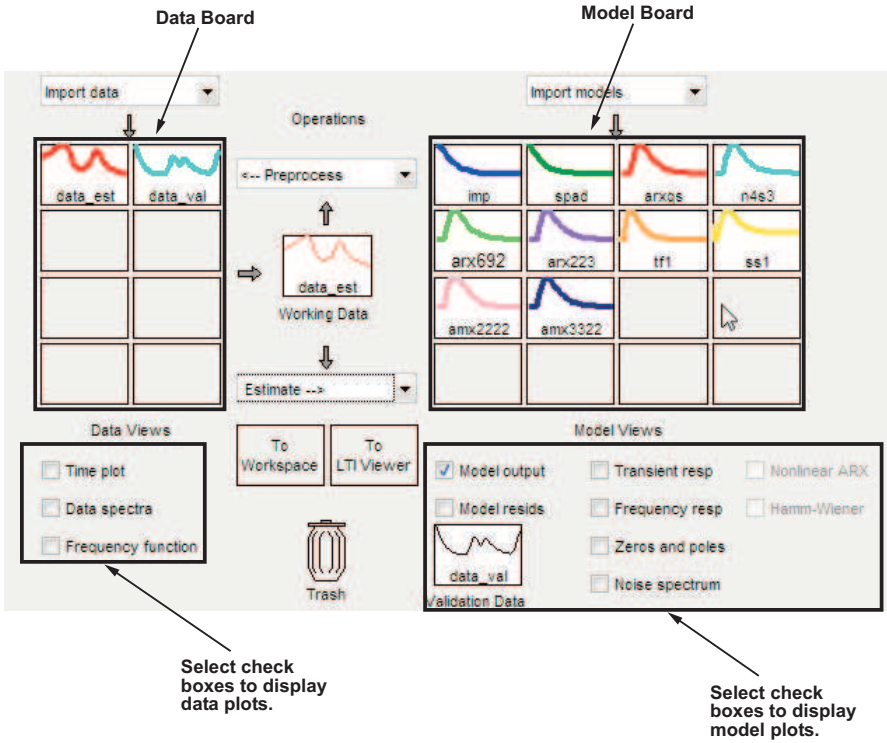


Note Only one session can be open at a time.

You can also start a new session by closing the current session using **File > Close session**. This toolbox prompts you to save your current session if it is not already saved.

Description of the System Identification App Window

The following figure describes the different areas in the System Identification app.



The layout of the window organizes tasks and information from left to right. This organization follows a typical workflow, where you start in the top-left corner by importing data into the System Identification app using the **Import data** menu and end in the bottom-right corner by plotting the characteristics of your estimated model on model plots. For more information about using the System Identification app, see “Steps for Using the System Identification App” on page 21-2.

The **Data Board** area, located below the **Import data** menu in the System Identification app, contains rectangular icons that represent the data you imported into the app.

The Model Board, located to the right of the **<–Preprocess** menu in the System Identification app, contains rectangular icons that represent the models you estimated or imported into the app. You can drag and drop model icons in the Model Board into open dialog boxes.

Opening a Saved Session

You can open a previously saved session using the following syntax:

```
systemIdentification(session,path)
```

`session` is the file name of the session you want to open and `path` is the location of the session file. Session files have the extension `.sid`. When the session file in on the `matlabpath`, you can omit the `path` argument.

If the System Identification app is already open, you can open a session by selecting **File > Open session**.

Note If there is data in the System Identification app, you must close the current session before you can open a new session by selecting **File > Close session**.

Saving, Merging, and Closing Sessions

The following table summarizes the menu commands for saving, merging, and closing sessions in the System Identification app.

Task	Command	Comment
Close the current session and start a new session.	**File > Close session**	You are prompted to save the current session before closing it.
Merge the current session with a previously saved session.	**File > Merge session**	You must start a new session and import data or models before you can select to merge it with a previously saved session. You are prompted to select the session file to merge with the current. This operation combines the data and the models of both sessions in the current session.
Save the current session.	**File > Save**	Useful for saving the session repeatedly after you have already saved the session once.

Task	Command	Comment
Save the current session under a new name.	**File > Save As**	Useful when you want to save your work incrementally. This command lets you revert to a previous stage, if necessary.

Deleting a Session

To delete a saved session, you must delete the corresponding session file.

Managing Models

- "Importing Models into the App" on page 21-7
- "Viewing Model Properties" on page 21-8
- "Renaming Models and Changing Display Color" on page 21-8
- "Organizing Model Icons" on page 21-9
- "Deleting Models in the App" on page 21-10
- "Exporting Models from the App to the MATLAB Workspace" on page 21-10

Importing Models into the App

You can import System Identification Toolbox models from the MATLAB workspace into the System Identification app. If you have Control System Toolbox software, you can also import any models (LTI objects) you created using this toolbox.

The following procedure assumes that you begin with the System Identification app already open. If this window is not open, type the following command at the prompt:

```
systemIdentification
```

To import models into the System Identification app:

1. Select **Import** from the **Import models** list to open the Import Model Object dialog box.
2. In the **Enter the name** field, type the name of a model object. Press **Enter**.
3. (Optional) In the **Notes** field, type any notes you want to store with this model.
4. Click **Import**.

5 Click **Close** to close the Import Model Object dialog box.

Viewing Model Properties

You can get information about each model in the System Identification app by right-clicking the corresponding model icon.

The Data/model Info dialog box opens. This dialog box describes the contents and the properties of the corresponding model. It also displays any associated notes and the command-line equivalent of the operations you used to create this model.

Tip To view or modify properties for several models, keep this window open and right-click each model in the System Identification app. The Data/model Info dialog box updates when you select each model.

Renaming Models and Changing Display Color

You can rename a model and change its display color by double-clicking the model icon in the System Identification app.

The Data/model Info dialog box opens. This dialog box describes both the contents and the properties of the model. The object description area displays the syntax of the operations you used to create the model in the app.

To rename the model, enter a new name in the **Model name** field.

You can also specify a new display color using three RGB values in the **Color** field. Each value is between `0` to `1` and indicates the relative presence of red, green, and blue, respectively. For more information about specifying default data color, see “Customizing the System Identification App” on page 21-15.

Tip As an alternative to using three RGB values, you can enter any *one* of the following letters in single quotes:

```
'y' 'r' 'b' 'c' 'g' 'm' 'k'
```

These represent yellow, red, blue, cyan, green, magenta, and black, respectively.

Finally, you can enter comments about the origin and state of the model in the **Diary And Notes** area.

To view model properties in the MATLAB Command Window, click **Present**.

Organizing Model Icons

You can rearrange model icons in the System Identification app by dragging and dropping the icons to empty Model Board rectangles.

Note You cannot drag and drop a model icon into the data area on the left.

When you need additional space for organizing model icons, select **Options > Extra model/data board** in the System Identification app. This action opens an extra session window with blank rectangles. The new window is an extension of the current session and does not represent a new session.

Tip When you import or estimate models and there is insufficient space for the icons, an additional session window opens automatically.

You can drag and drop model icons between the main System Identification app and any extra session windows.

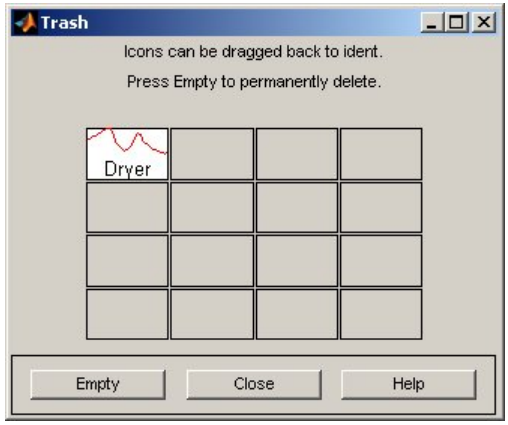
Type comments in the **Notes** field to describe the models. When you save a session, as described in “Saving, Merging, and Closing Sessions” on page 21-6, all additional windows and notes are also saved.

Deleting Models in the App

To delete models in the System Identification app, drag and drop the corresponding icon into **Trash**. You can also use the **Delete** key on your keyboard to move items to the **Trash**. Moving items to **Trash** does not permanently delete these items.

To restore a model from **Trash**, drag its icon from **Trash** to the Model Board in the System Identification app. You can view the **Trash** contents by double-clicking the **Trash** icon.

Note You must restore a model to the Model Board; you cannot drag model icons to the Data Board.



To permanently delete all items in **Trash**, select **Options** > **Empty trash**.

Exiting a session empties **Trash** automatically.

Exporting Models from the App to the MATLAB Workspace

The models you create in the System Identification app are not available in the MATLAB workspace until you export them. Exporting is necessary when you need to perform an operation on the model that is only available at the command line. Exporting models to the MATLAB workspace also makes them available to the Simulink software or another toolbox, such as the Control System Toolbox product.

To export a model to the MATLAB workspace, do one of the following:

- Drag and drop the corresponding icon to the **To Workspace** rectangle.
- Right-click the icon to open the Data/model Info dialog box. Click **Export** to export the model.

When you export models to the MATLAB workspace, the resulting variables have the same name as in the System Identification app.

Working with Plots

- "Identifying Data Sets and Models on Plots" on page 21-11
- "Changing and Restoring Default Axis Limits" on page 21-12
- "Selecting Measured and Noise Channels in Plots" on page 21-14
- "Grid and Line Styles in Plots" on page 21-14
- "Opening a Plot in a MATLAB Figure Window" on page 21-15
- "Printing Plots" on page 21-15

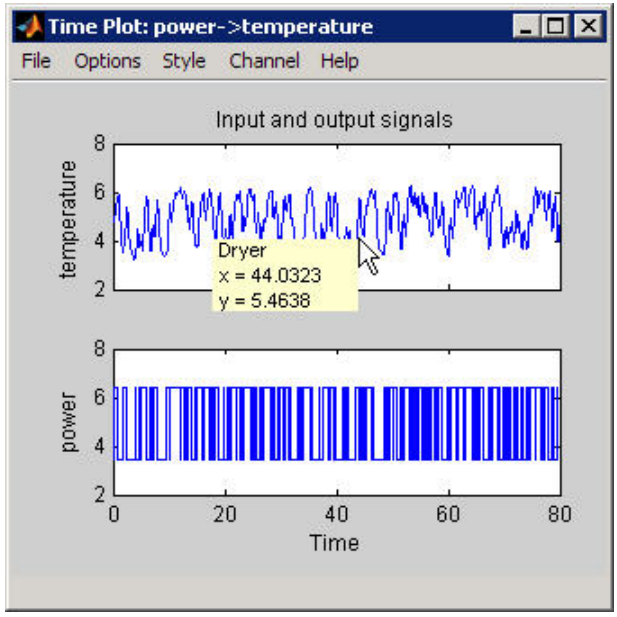
Identifying Data Sets and Models on Plots

You can identify data sets and models on a plot by color: the color of the line in the data or model icon in the System Identification app matches the line color on the plots.

You can also display data tips for each line on the plot by clicking a plot curve and holding down the mouse button.

Note You must disable zoom by selecting **Style > Zoom** before you can display data tips. For more information about enabling zoom, see "Magnifying Plots" on page 21-12.

The following figure shows an example of a data tip, which contains the name of the data set and the coordinates of the data point.



Data Tip on a Plot

Changing and Restoring Default Axis Limits

There are two ways to change which portion of the plot is currently in view:

- “Magnifying Plots” on page 21-12
- “Setting Axis Limits” on page 21-13

Magnifying Plots

Enable zoom by selecting **Style** > **Zoom** in the plot window. To disable zoom, select **Style** > **Zoom** again.

Tip To verify that zoom is active, click the **Style** menu. A check mark should appear next to **Zoom**.

You can adjust magnification in the following ways:

- To zoom in default increments, left-click the portion of the plot you want to center in the plot window.
- To zoom in on a specific region, click and drag a rectangle that identifies the region for magnification. When you release the mouse button, the selected region is displayed.
- To zoom out, right-click on the plot.

Note To restore the full range of the data in view, select **Options** > **Autorange** in the plot window.

Setting Axis Limits

You can change axis limits for the vertical and the horizontal axes of the input and output channels that are currently displayed on the plot.

1. Select **Options** > **Set axes limits** to open the Limits dialog box.
2. Specify a new range for each axis by editing its lower and upper limits. The limits must be entered using the format *[LowerLimit UpperLimit]*. Click **Apply**. For example:

   ```
   [0.1 100]
   ```

   Note To restore full axis limits, select the **Auto** check box to the right of the axis name, and click **Apply**.
3. To plot data on a linear scale, clear the **Log** check box to the right of the axis name, and click **Apply**.

   Note To revert to base-10 logarithmic scale, select the **Log** check box to the right of the axis name, and click **Apply**.
4. Click **Close**.

Note To view the entire data range, select **Options** > **Autorange** in the plot window.

Selecting Measured and Noise Channels in Plots

Model inputs and outputs are called *channels*. When you create a plot of a multivariable input-output data set or model, the plot only shows one input-output channel pair at a time. The selected channel names are displayed in the title bar of the plot window.

Note When you select to plot multiple data sets, and each data set contains several input and output channels, the **Channel** menu lists channel pairs from all data sets.

You can select a different input-output channel pair from the **Channel** menu in any System Identification Toolbox plot window.

The **Channel** menu uses the following notation for channels: `u1->y2` means that the plot displays a transfer function from input channel `u1` to output channel `y2`. System Identification Toolbox estimates as many noise sources as there are output channels. In general, `e@ynam` indicates that the noise source corresponds to the output with name `ynam`.

For example, `e@y3->y1` means that the transfer function from the noise channel (associated with `y3`) to output channel `y2` is displayed. For more information about noise channels, see “Separation of Measured and Noise Components of Models” on page 4-39.

Tip When you import data into the System Identification app, it is helpful to assign meaningful channel names in the Import Data dialog box. For more information about importing data, see “Represent Data”.

Grid and Line Styles in Plots

There are several **Style** options that are common to all plot types. These include the following:

- “Grid Lines” on page 21-14
- “Solid or Dashed Lines” on page 21-15

Grid Lines

To toggle showing or hiding grid lines, select **Style > Grid**.

Solid or Dashed Lines

To display currently visible lines as a combination of solid, dashed, dotted, and dash-dotted line style, select **Style > Separate linestyles**.

To display all solid lines, select **Style > All solid lines**. This choice is the default.

All line styles match the color of the corresponding data or model icon in the System Identification app.

Opening a Plot in a MATLAB Figure Window

The MATLAB Figure window provides editing and printing commands for plots that are not available in the System Identification Toolbox plot window. To take advantage of this functionality, you can first create a plot in the System Identification app, and then open it in a MATLAB Figure window to fine-tune the display.

After you create the plot, as described in "Plot Models in the System Identification App" on page 17-7, select **File > Copy figure** in the plot window. This command opens the plot in a MATLAB Figure window.

Printing Plots

To print a System Identification Toolbox plot, select **File > Print** in the plot window. In the Print dialog box, select the printing options and click **OK**.

Customizing the System Identification App

- "Types of App Customization" on page 21-15
- "Saving Session Preferences" on page 21-16
- "Modifying idlayout.m" on page 21-16

Types of App Customization

The System Identification app lets you customize the window behavior and appearance. For example, you can set the size and position of specific dialog boxes and modify the appearance of plots.

You can save the session to save the customized app state.

You might choose to edit the file that controls default settings, as described in "Modifying idlayout.m" on page 21-16 (advanced usage).

Saving Session Preferences

Use **Options > Save preferences** to save the current state of the System Identification app. This command saves the following settings to a preferences file, `idprefs.mat`:

- Size and position of the System Identification app
- Sizes and positions of dialog boxes
- Four recently used sessions
- Plot options, such as line styles, zoom, grid, and whether the input is plotted using zero-order hold or first-order hold between samples

You can only edit `idprefs.mat` by changing preferences in the app.

The `idprefs.mat` file is located in the same folder as `startup.m`, by default. To change the location where your preferences are saved, use the `midprefs` command with the new path as the argument. For example:

```
midprefs('c:\matlab\toolbox\local\')
```

You can also type `midprefs` and browse to the desired folder.

To restore the default preferences, select **Options > Default preferences**.

Modifying idlayout.m

You might want to customize the default plot options by editing `idlayout.m` (advanced usage).

To customize `idlayout.m` defaults, save a copy of `idlayout.m` to a folder in your `matlabpath` just above the `ident` folder level.

Caution Do not edit the original file to avoid overwriting the `idlayout.m` defaults shipped with the product.

You can customize the following plot options in `idlayout.m`:

- Order in which colors are assigned to data and model icons
- Line colors on plots

- Axis limits and tick marks
- Plot options, set in the plot menus
- Font size

Note When you save preferences using **Options > Save preferences** to `idprefs.mat`, these preferences override the defaults in `idlayout.m`. To give `idlayout.m` precedence every time you start a new session, select **Options > Default preferences**.

Diagnostics and Prognostics

- "Time Series Prediction and Forecasting for Prognosis" on page 22-2
- "Fault Detection Using Data Based Models" on page 22-16

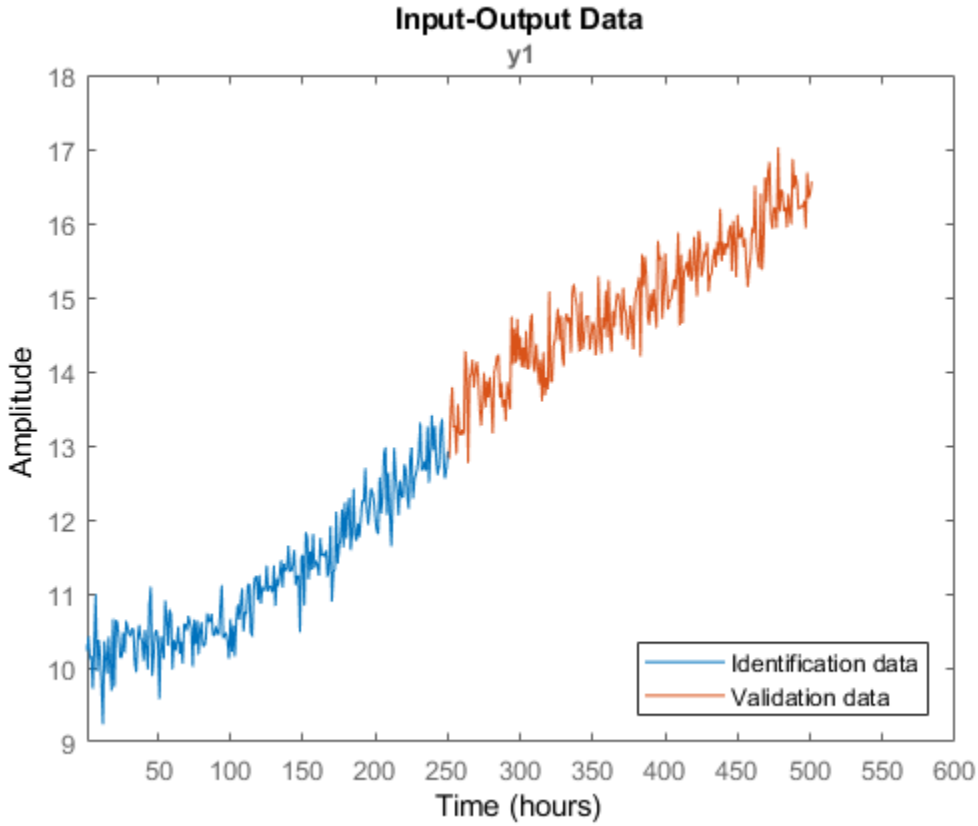
Time Series Prediction and Forecasting for Prognosis

This example shows how to create a time series model and use the model for prediction, forecasting, and state estimation. The measured data is from an induction furnace whose slot size erodes over time. The slot size cannot be measured directly but the furnace current and consumed power are measured. It is known that as the slot size increases, the slot resistance decreases. The ratio of measured current squared to measured power is thus proportional to the slot size. You use the measured current-power ratio (both current and power measurements are noisy) to create a time series model and use the model to estimate the current slot size and forecast the future slot size. Through physical inspection the induction furnace slot size is known at some points in time.

Load and Plot the Measured Data

The measured current-power ratio data is stored in the `iddata_TimeSeriesPrediction` MATLAB file. The data is measured at hourly intervals and shows that over time the ratio increases indicating erosion of the furnace slot. You develop a time series model using this data. Start by separating the data into an identification and a validation segment.

```
load iddata_TimeSeriesPrediction
n = numel(y);
ns = floor(n/2);
y_id = y(1:ns,:);
y_v = y((ns+1:end),:);
data_id = iddata(y_id, [], Ts, 'TimeUnit', 'hours');
data_v  = iddata(y_v, [], Ts, 'TimeUnit', 'hours', 'Tstart', ns+1);

plot(data_id,data_v)
legend('Identification data','Validation data','location','SouthEast');
```



Model Identification

The slot erosion can be modelled as a state-space system with noise input and measured current-power ratio as output. The measured current-power ratio is proportional to the system state, or

$$x_{n+1} = Ax_n + Ke_n$$

$$y_n = Cx_n + e_n$$

Where x_n the state vector, contains the slot size; y_n is the measured current-power ratio; e_n noise and A, C, K are to be identified.

Use the `ssest()` command to identify a discrete state-space model from the measured data.

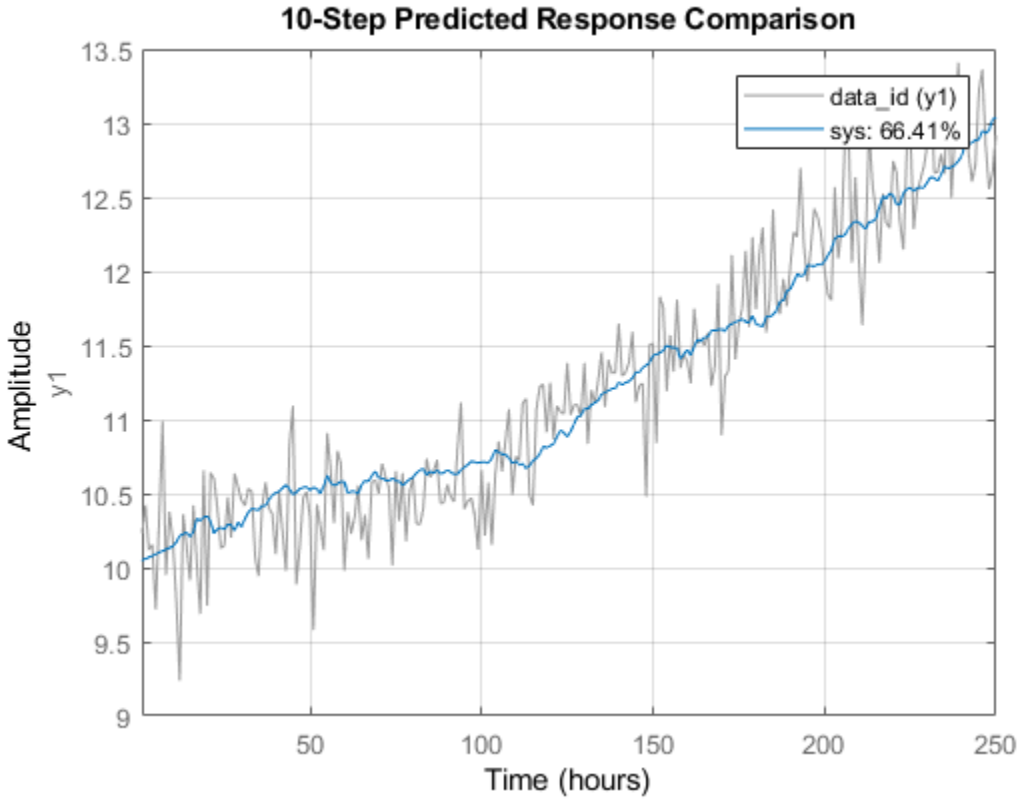
```
sys = ssest(data_id,1,'Ts',Ts,'form','canonical')
```

```
sys =
  Discrete-time identified state-space model:
    x(t+Ts) = A x(t) + K e(t)
       y(t) = C x(t) + e(t)

  A =
         x1
   x1  1.001

  C =
       x1
   y1   1

  K =
           y1
   x1  0.09465

Sample time: 1 hours

Parameterization:
   CANONICAL form with indices: 1.
   Disturbance component: estimate
   Number of free coefficients: 2
   Use "idssdata", "getpvec", "getcov" for parameters and their uncertainties.

Status:
Estimated using SSEST on time domain data "data_id".
Fit to estimation data: 67.38% (prediction focus)
FPE: 0.09575, MSE: 0.09348
```

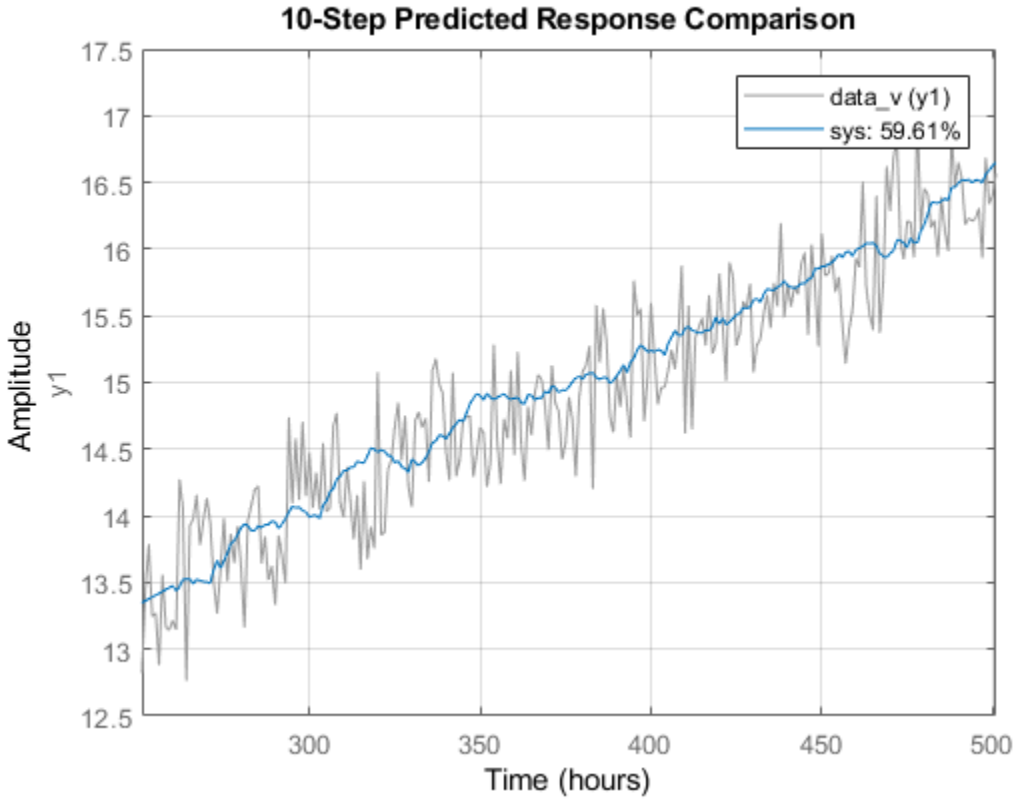
The identified model minimizes the 1-step ahead prediction. Validate the model using a 10 step ahead predictor, i.e., given $y_0, \ldots, y_n$ use the model to predict y_{n+10}. Note that the error between the measured and predicted values, $y_0 - \hat{y}_0, \ldots, y_n - \hat{y}_n$, are used to make the y_{n+10} prediction.

Use the 10 step ahead predictor for the identification data and the independent validation data.

```
nstep = 10;
compare(sys,data_id,nstep)  % comparison of 10-step prediction to estimation data
grid('on');
```




```
figure; compare(sys,data_v,nstep)  % comparison to validation data
grid('on');
```

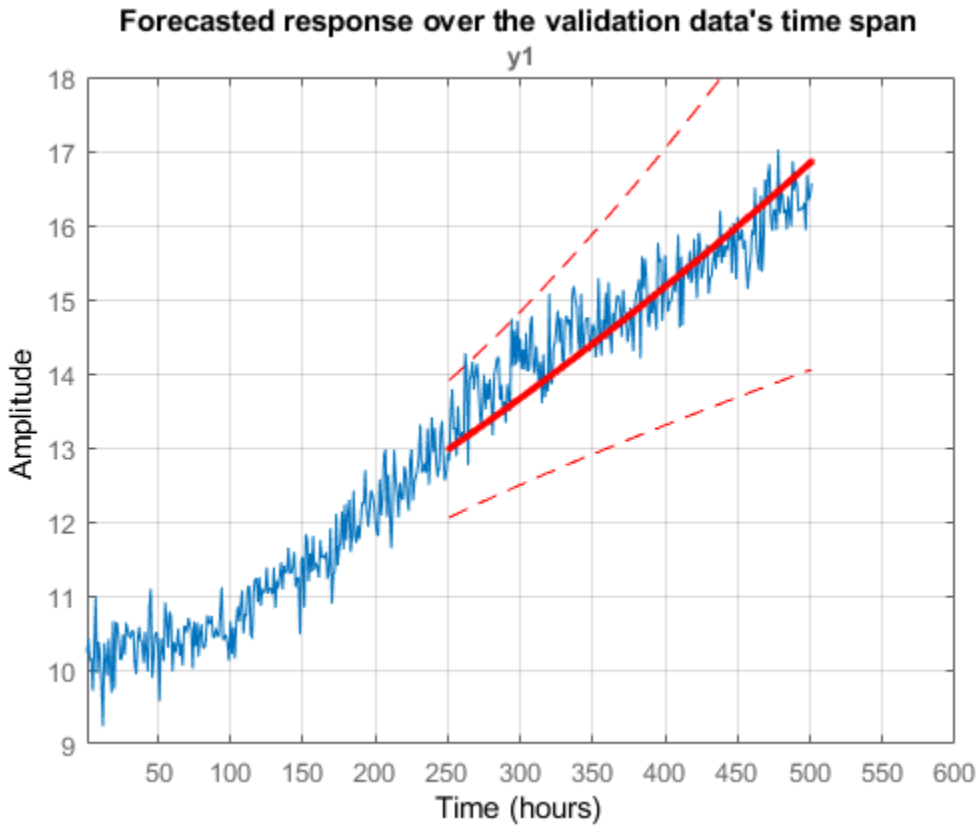


The above exercise Both data sets show that the predictor matches the measured data.

Forecasting is used to further verify the model. Forecasting uses the measured data record $y_0, y_1, \ldots, y_n - \hat{y}_n$ to compute the model state at time step n. This value is used as initial condition for forecasting the model response for a future time span. We forecast the model response over the time span of the validation data and then compare the two. We can also compute the uncertainty in forecasts and plot +/- 3 sd of their values.

```
MeasuredData = iddata(y, [], Ts, 'TimeUnit', 'hours'); % = [data_id;data_v]
t0 = MeasuredData.SamplingInstants;

Horizon = size(data_v,1); % forecasting horizon
[yF, ~, ~, yFSD]  = forecast(sys, data_id, Horizon);
% Note: yF is IDDATA object while yFSD is a double vector
```

```
t = yF.SamplingInstants; % extract time samples
yFData = yF.OutputData;      % extract response as double vector
plot(MeasuredData)
hold on
plot(t, yFData, 'r.-', t, yFData+3*yFSD, 'r--', t, yFData-3*yFSD, 'r--')
hold off
title('Forecasted response over the validation data''s time span')
grid on
```

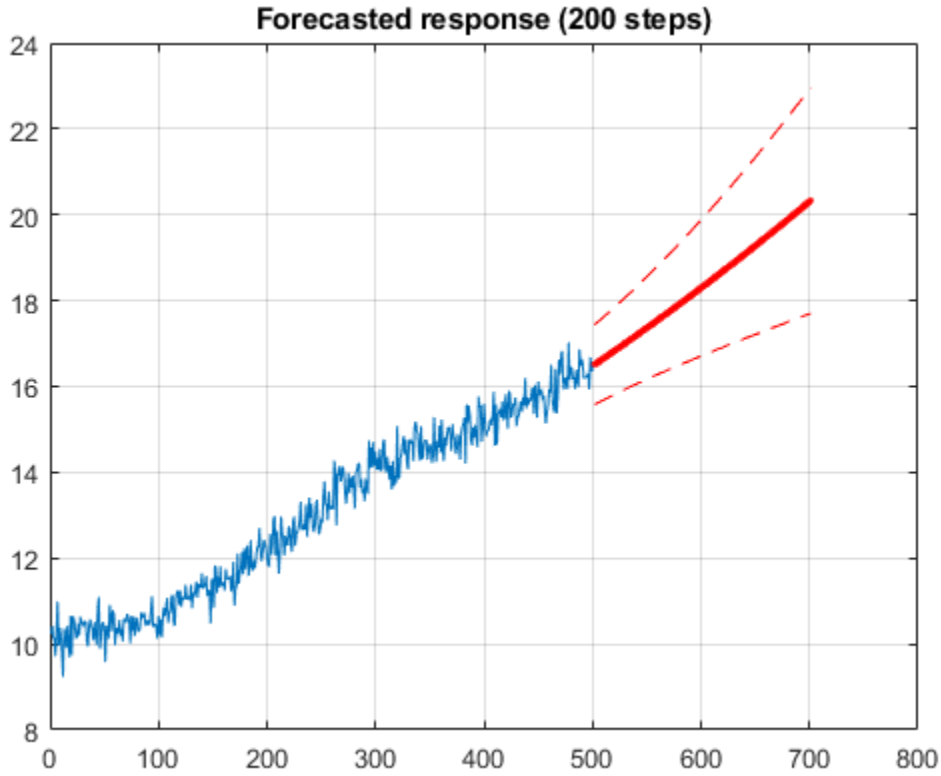


The plot shows that the model response with confidence intervals (indicated by the red colored dashed curves) overlap the measured value for the validation data. The combined prediction and forecasting results indicate that the model represents the measured current-power ratio.

The forecasting results also show that over large horizons the model variance is large and for practical purposes future forecasts should be limited to short horizons. For the induction furnace model a horizon of 200 hours is appropriate.

Finally we use the model to forecast the response 200 steps into future for the time span of 502-701 hours.

```
Horizon = 200; % forecasting horizon
[yFuture, ~, ~, yFutureSD] = forecast(sys, MeasuredData, Horizon);
t = yFuture.SamplingInstants; % extract time samples
yFutureData = yFuture.OutputData;      % extract response as double vector
plot(t0, y,...
   t, yFutureData, 'r.-', ...
   t, yFutureData+3*yFutureSD, 'r--', ...
   t, yFutureData-3*yFutureSD, 'r--')
title('Forecasted response (200 steps)')
grid on
```



The blue curve shows the measured data that spans over 1-501 hours. The red curve is the forecasted response for 200 hours beyond the measured data's time range. The red dashed curves shows the 3 sd uncertainty in the forecasted response based on random sampling of the identified model.

State Estimation

The identified model matches the measured current-power ratio but we are interested in the furnace slot size which is a state in the model. The identified model has an arbitrary state that can be transformed so that the state has meaning, in this case the slot size.

Create a predictor for the arbitrary state. The identified model covariances need to be translated to the predictor model using the `translatecov()` command. The

`createPredictor()` function simply extracts the third output argument of the `predict()` function to be used with `translatecov()`.

```
type createPredictor
est = translatecov(@(s) createPredictor(s,data_id),sys)
```

```
function pred = createPredictor(mdl,data)
%CREATEPREDICTOR Return 1-step ahead predictor.
%
%   sys = createPredictor(mdl,data)
%
%   Create a 1-step ahead predictor model sys for the specified model mdl
%   and measured data. The function is used by
%   |TimeSeriedPredictionExample| and the |translatecov()| command to
%   translate the identified model covariance to the predictor.

% Copyright 2015 The MathWorks, Inc.
[~,~,pred] = predict(mdl,data,1);
```

```
est =
  Discrete-time identified state-space model:
    x(t+Ts) = A x(t) + B u(t)
       y(t) = C x(t) + D u(t)

  A =
          x1
   x1  0.9064

  B =
           y1
   x1  0.09465

  C =
      x1
   y1   1

  D =
      y1
   y1   0

Sample time: 1 hours

Parameterization:
```

```
   CANONICAL form with indices: 1.
   Feedthrough: none
   Disturbance component: none
   Number of free coefficients: 2
   Use "idssdata", "getpvec", "getcov" for parameters and their uncertainties.

Status:
Created by direct construction or transformation. Not estimated.
```

The model `est` is a 1-step ahead predictor expressed in the same state coordinates as the original model `sys`. How do we transform the state coordinates so that the model's state corresponds to the (time dependent) slot size? The solution is to rely on actual, direct measurements of the slot size taken intermittently. This is not uncommon in practice where the cost of taking direct measurements is high and only be done periodically (such as when the component is being replaced).

Specifically, transform the predictor state, x_n, to z_n, so that $y_n = Cz_n$ where y_n the measured current-power ratio, and z_n is the furnace slot size. In this example, four direct measurements of the furnace slot size, `sizeMeasured`, and furnace current-power ratio, `ySizeMeasured`, are used to estimate C. In transforming the predictor the predictor covariances also need to be transformed. Hence we use the `translatecov()` command to carry out the state coordinate transformation.

```
Cnew = sizeMeasured\ySizeMeasured;
est  = translatecov(@(s) ss2ss(s,s.C/Cnew),est)
```

```
est =
  Discrete-time identified state-space model:
    x(t+Ts) = A x(t) + B u(t)
       y(t) = C x(t) + D u(t)

  A =
          x1
   x1  0.9064

  B =
          y1
   x1  0.9452

  C =
          x1
   y1  0.1001
```

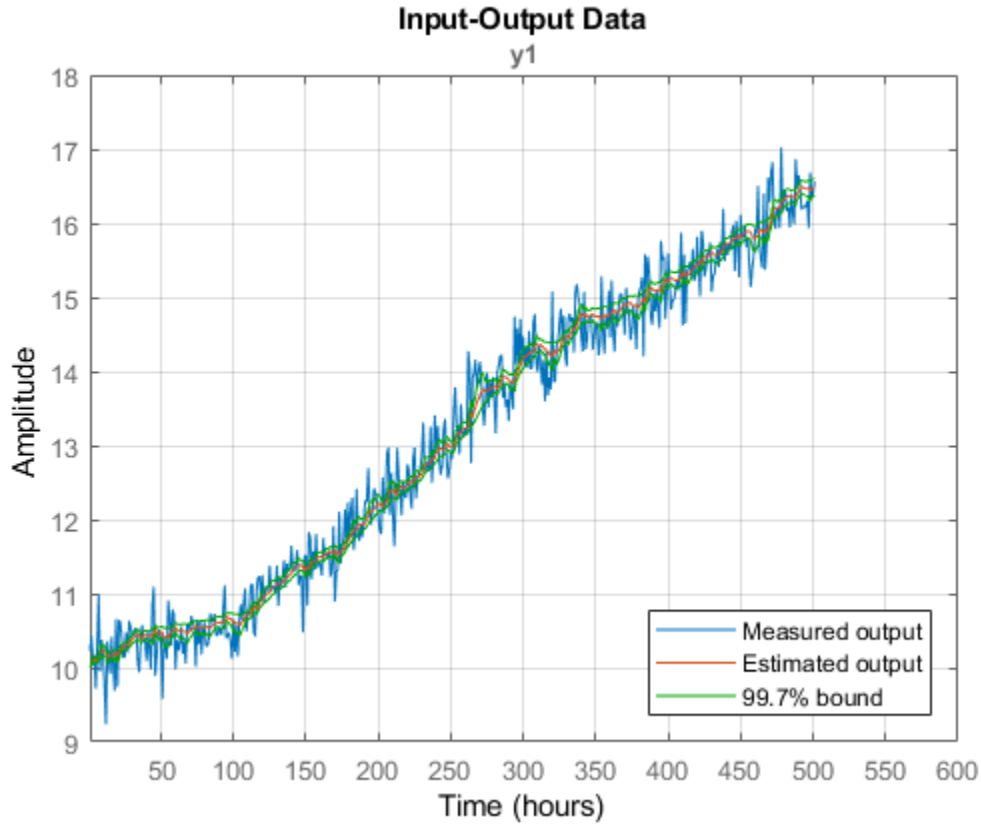
```
  D =
       y1
   y1   0

Sample time: 1 hours

Parameterization:
   CANONICAL form with indices: 1.
   Feedthrough: none
   Disturbance component: none
   Number of free coefficients: 2
   Use "idssdata", "getpvec", "getcov" for parameters and their uncertainties.

Status:
Created by direct construction or transformation. Not estimated.
```

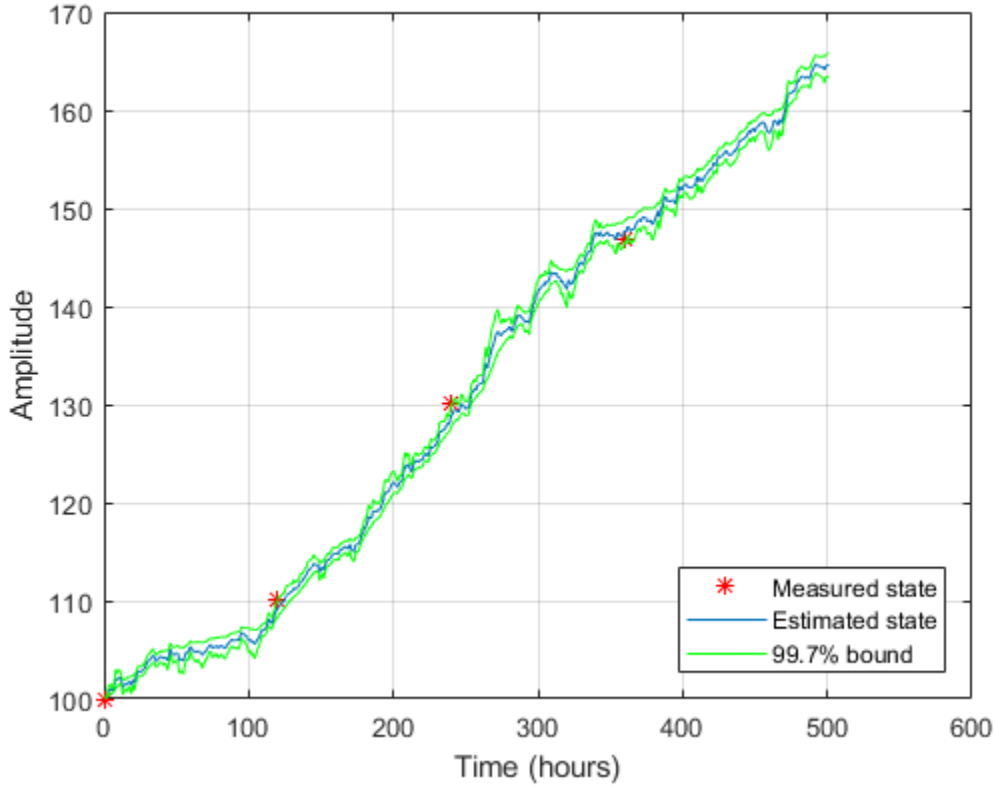
The predictor is now expressed in the desired state coordinates. It has one input that is the measured system output (the furnace current-power ratio) and one output that is the predicted system output (the furnace slot size). The predictor is simulated to estimate the system output and system state.

```
opts = simOptions;
opts.InitialCondition = sizeMeasured(1);
U = iddata([],[data_id.Y; data_v.Y],Ts,'TimeUnit','hours');
[ye,ye_sd,xe] = sim(est,U,opts);
```

Compare the estimated output and slot size with measured and known values.

```
yesdp   = ye;
yesdp.Y = ye.Y+3*ye_sd;
yesdn   = ye;
yesdn.Y = ye.Y-3*ye_sd;
n = numel(xe);
figure, plot([data_id;data_v],ye,yesdp,'g',yesdn,'g')
legend('Measured output','Estimated output','99.7% bound','location','SouthEast')
grid('on')
figure, plot(tSizeMeasured,sizeMeasured,'r*',1:n,xe,1:n,yesdp.Y/est.C,'g',1:n,yesdn.Y/e
legend('Measured state','Estimated state','99.7% bound','location','SouthEast')
xlabel('Time (hours)')
ylabel('Amplitude');
grid('on')
```


Input-Output Data
y1
Amplitude
Time (hours)
Measured output
Estimated output
99.7% bound
18
17
16
15
14
13
12
11
10
9
50
100
150
200
250
300
350
400
450
500
550
600



Using Prediction and Forecasting for Prognosis

The combination of predictor model and forecasting allow us to perform prognosis on the induction furnace.

The predictor model allows us to estimate the current furnace slot size based on measured data. If the estimated value is at or near critical values an inspection or maintenance can be scheduled. Forecasting allows us to, from the estimated current state, predict the future system behaviour allowing us to predict when an inspection or maintenance may be needed.

Further the predictor and forecast model can be re-identified as more data becomes available. In this example one data set was used to identify the predictor and forecast models but as more data is accumulated the models can be re-identified.

See Also

`compare` | `forecast` | `predict`

More About

- “Perform Multivariate Time Series Forecasting” on page 17-25
- “Fault Detection Using Data Based Models” on page 22-16
- “Introduction to Forecasting of Dynamic System Response” on page 14-37

Fault Detection Using Data Based Models

This example shows how to use a data-based modeling approach for fault detection. This example requires Statistics and Machine Learning Toolbox™.

Introduction

Early detection and isolation of anomalies in a machines operation can help to reduce accidents, reduce downtime and thus save operational costs. The approach involves processing live measurements from a systems operation to flag any unexpected behavior that would point towards a newly developed fault.

This example explores the following fault diagnosis aspects:

1. Detection of abnormal system behavior by residual analysis
2. Detection of deterioration by building models of damaged system
3. Tracking system changes using online adaptation of model parameters

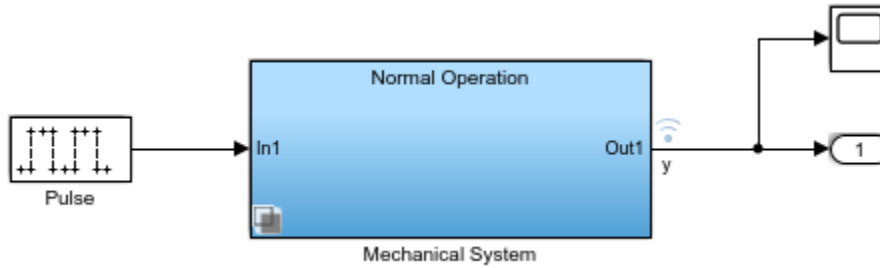
Identifying a Dynamic Model of System Behavior

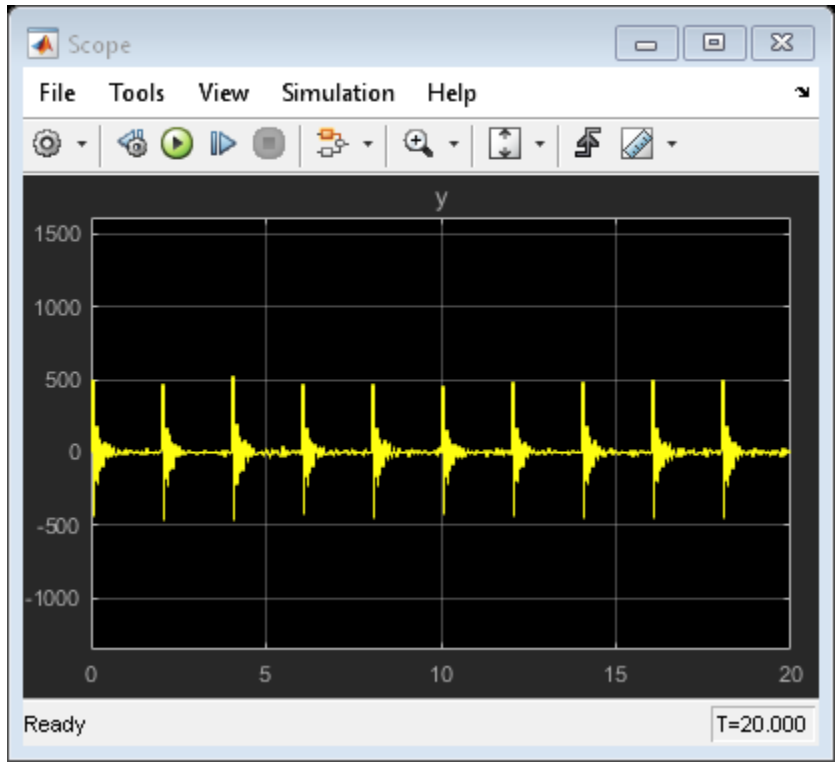
In a model based approach to detection, a dynamic model of the concerned system is first built using measured input and output data. A good model is able to accurately predict the response of the system for a certain future time horizon. When the prediction is not good, the residuals may be large and could contain correlations. These aspects are exploited to detect the incidence of failure.

Consider a building subject to impacts and vibrations. The source of vibrations can be different types of stimuli depending upon the system such as wind gusts, contact with running engines and turbines, or ground vibrations. The impacts are a result of impulsive bump tests on the system that are added to excite the system sufficiently. Simulink model `idMechanicalSystem.slx` is a simple example of such as structure. The excitation comes from periodic bumps as well as ground vibrations modeled by filtered white noise. The output of the system is collected by a sensor that is subject to measurement noise. The model is able to simulate various scenarios involving the structure in a healthy or a damaged state.

```
sysA = 'idMechanicalSystem';
open_system(sysA)
% Set the model in the healthy mode of operation
set_param([sysA,'/Mechanical System'],'OverrideUsingVariant','Normal')
% Simulate the system and log the response data
```

```
sim(sysA)
ynormal = logsout.getElement('y').Values;
```

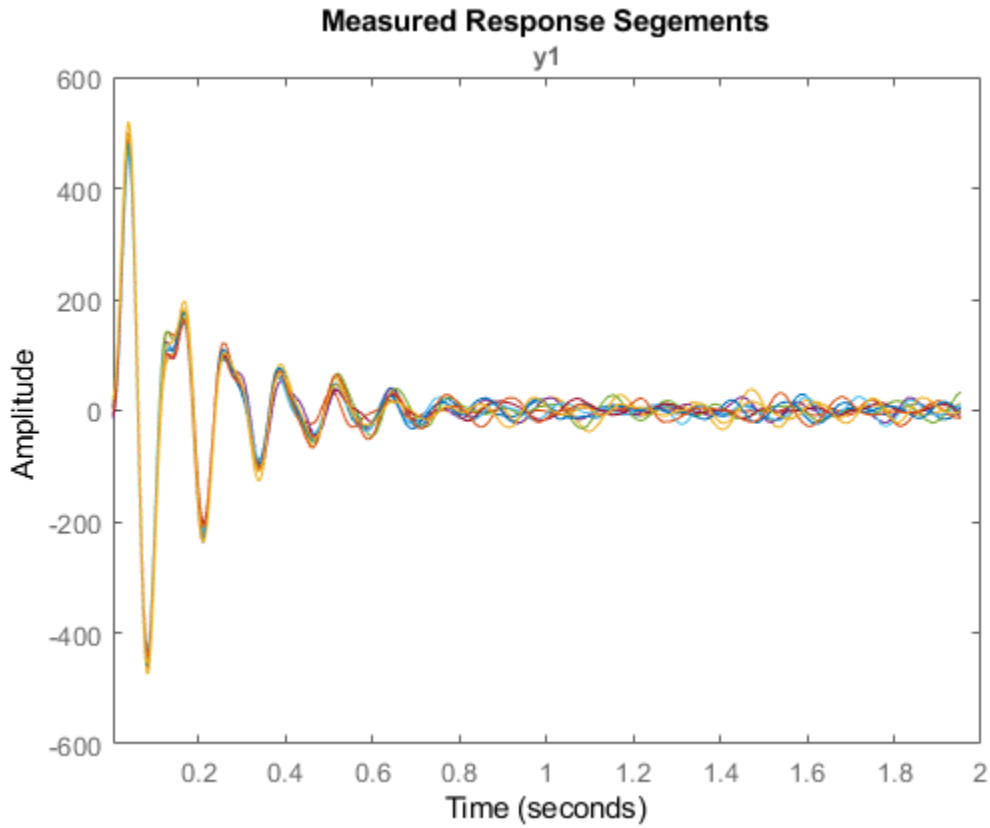




The input signal was not measured; all we have recorded is the response `ynormal`. Hence we build a dynamic model of the system using "blind identification" techniques. In particular, we build an ARMA model of the recorded signal as a representation of the

system. This approach works when the input signal is assumed to be (filtered) white noise. Since the data is subject to periodic bumps, we split the data into several pieces each starting at the incidence of a bump. This way, each data segment contains the response to one bump plus random excitations - a situation that can be captured using a time series model, where the effect of the bump is attributed to suitable initial conditions.

```
Ts = 1/256;  % data sample time
nr = 10;     % number of bumps in the signal
N = 512;     % length of data between bumps
znormal = cell(nr,1);
for ct = 1:nr
   ysegment = ynormal.Data((ct-1)*N+(1:500));
   z = iddata(ysegment,[],Ts);
   znormal{ct} = z;  % each segment has only one bump
end
plot(znormal{:}) % plot a sampling of the recorded segments
title('Measured Response Segements')
```



Split the data into estimation and validation pieces.

```
ze = merge(znormal{1:5});
zv = merge(znormal{6:10});
```

Estimate a 7:th order time-series model in state-space form using the `ssest()` command. The model order was chosen by cross validation (checking the fit to validation data) and residual analysis (checking that residuals are uncorrelated).

```
nx = 7;
model = ssest(ze, nx, 'form', 'canonical', 'Ts', Ts);
present(model)  % view model equations with parameter uncertainty
```

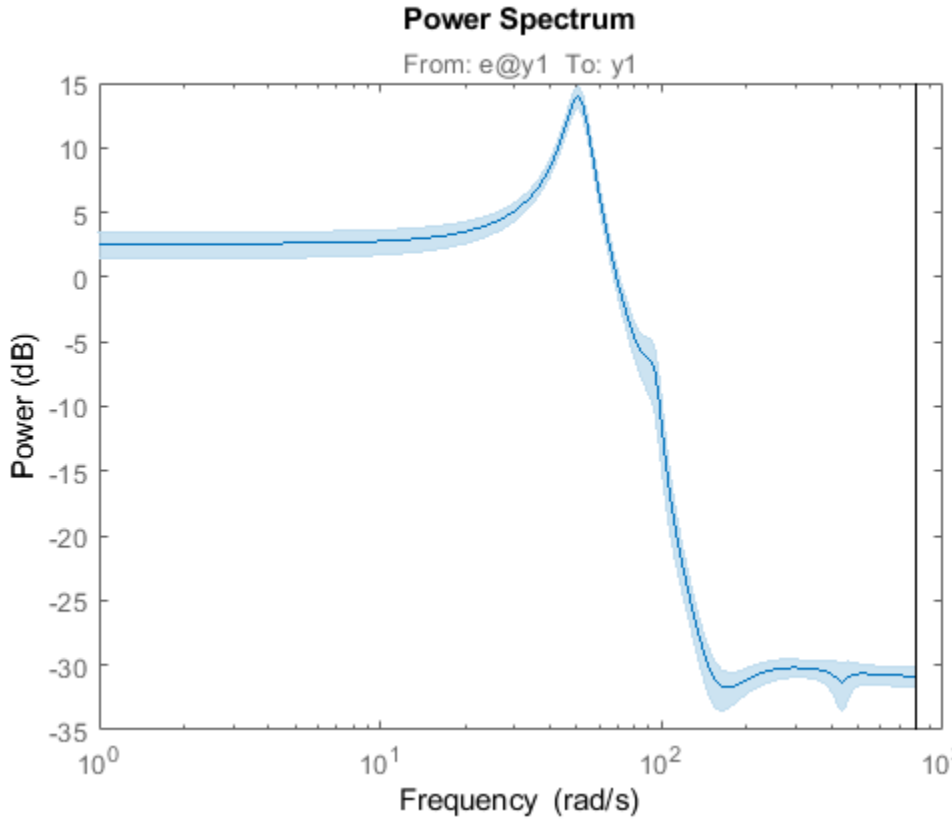
```
model =
```

```
Discrete-time identified state-space model:
  x(t+Ts) = A x(t) + K e(t)
     y(t) = C x(t) + e(t)

A =
                      x1                    x2                    x3
 x1                    0                     1                     0
 x2                    0                     0                     1
 x3                    0                     0                     0
 x4                    0                     0                     0
 x5                    0                     0                     0
 x6                    0                     0                     0
 x7   0.5668 +/- 0.04582   -2.778 +/- 0.2189    6.029 +/- 0.4491

                      x4                    x5                    x6
 x1                    0                     0                     0
 x2                    0                     0                     0
 x3                    1                     0                     0
 x4                    0                     1                     0
 x5                    0                     0                     1
 x6                    0                     0                     0
 x7    -8.441 +/- 0.5142     9.348 +/- 0.3551   -7.996 +/- 0.1434

                      x7
 x1                    0
 x2                    0
 x3                    0
 x4                    0
 x5                    0
 x6                    1
 x7    4.268 +/- 0.02657

C =
     x1  x2  x3  x4  x5  x6  x7
 y1   1   0   0   0   0   0   0

K =
                     y1
 x1  1.025 +/- 0.01396
 x2  1.444 +/- 0.01295
 x3  1.906 +/- 0.01252
 x4  2.386 +/- 0.01187
 x5  2.857 +/- 0.01448
 x6   3.26 +/-  0.0222
```

```
   x7  3.552 +/- 0.03367

Sample time: 0.0039063 seconds

Parameterization:
   CANONICAL form with indices: 7.
   Disturbance component: estimate
   Number of free coefficients: 14
   Use "idssdata", "getpvec", "getcov" for parameters and their uncertainties.

Status:
Termination condition: Near (local) minimum, (norm(g) < tol).
Number of iterations: 7, Number of function evaluations: 15

Estimated using SSEST on time domain data "ze".
Fit to estimation data: [99.07 99.04 99.15 99.05 99.04]% (prediction focus)
FPE: 0.6242, MSE: [0.5971 0.6535 0.5989 0.5871 0.6497]
More information in model's "Report" property.
```

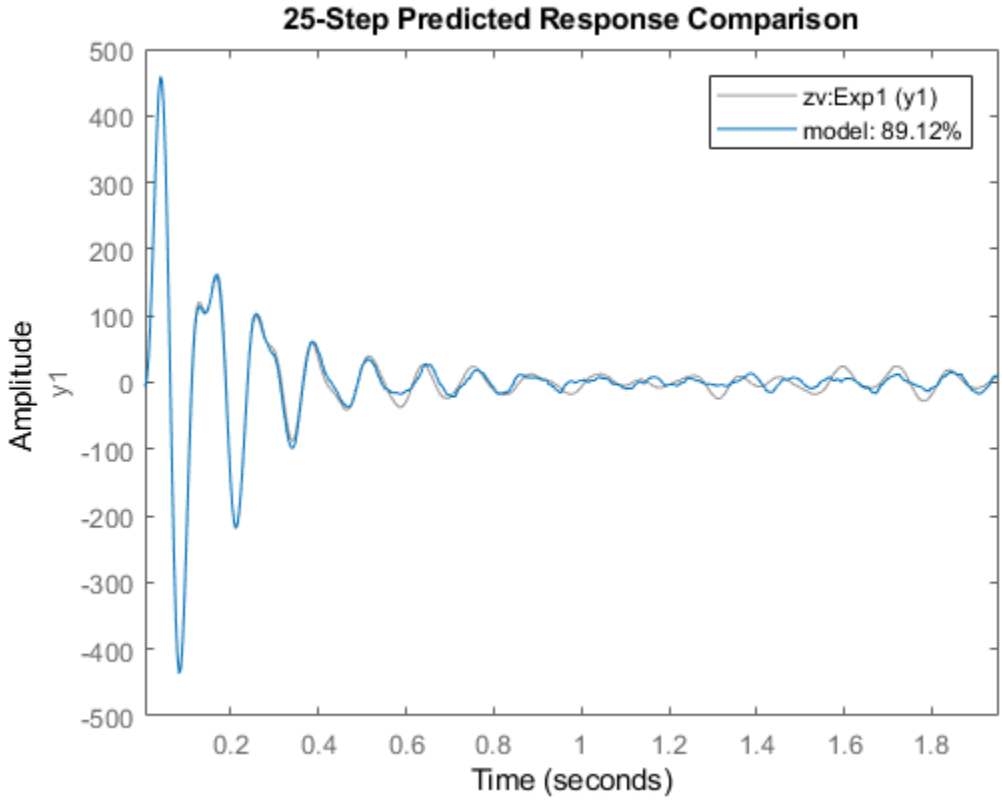
The model display shows relatively small uncertainty in parameter estimates. We can confirm the reliability by computing the 1-sd (99.73%) confidence bound on the estimated spectrum of the measured signal.

```
h = spectrumplot(model);
showConfidence(h, 3)
```



The confidence region is small, although there is about 30% uncertainty in the response at lower frequencies. The next step in validation is to see how well the model predicts the responses in the validation dataset `zv`. We use a 25-step ahead prediction horizon.

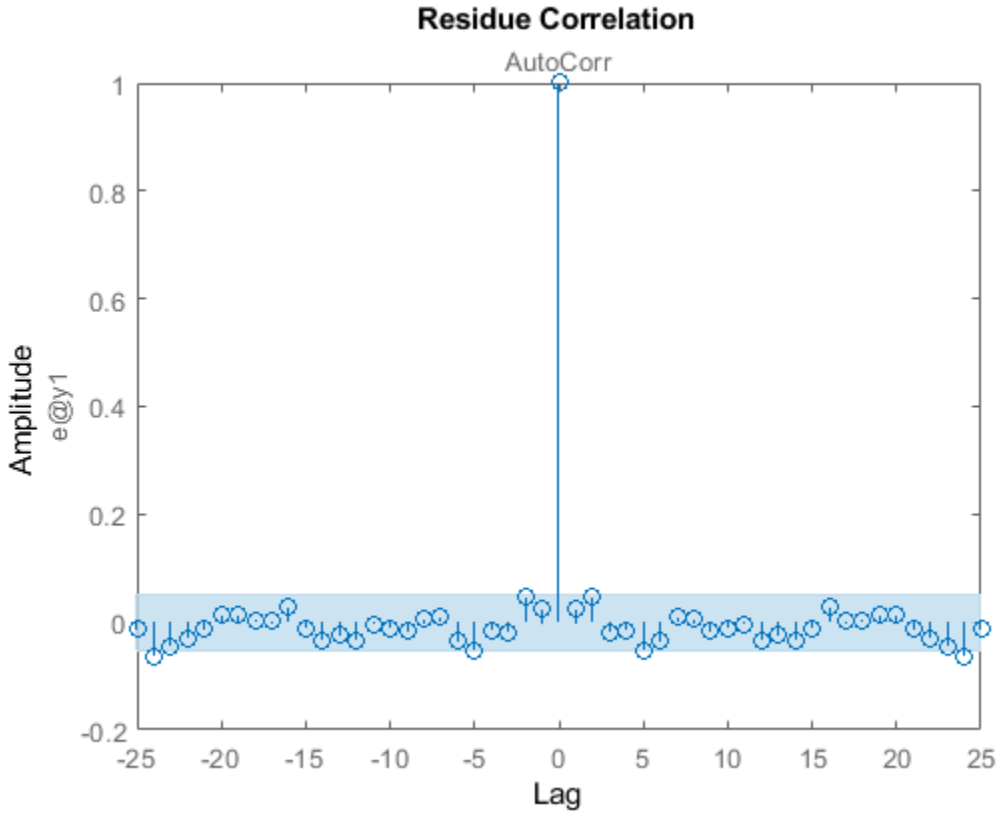
```
compare(zv, model, 25) % Validation against one dataset
```



The plot shows that the model is able to predict the response in the first experiment of the validation dataset 25 time steps (= 0.1 sec) in future with > 85% accuracy. To view thw fit to other experiments in the dataset, use the right-click context menu of the plot axes.

The final step in validating the model is to analyze the residuals generated by it. For a good model, these residuals should be white, i.e., show statistically insignificant correlations for non-zero lags:

```
resid(model, zv)
```



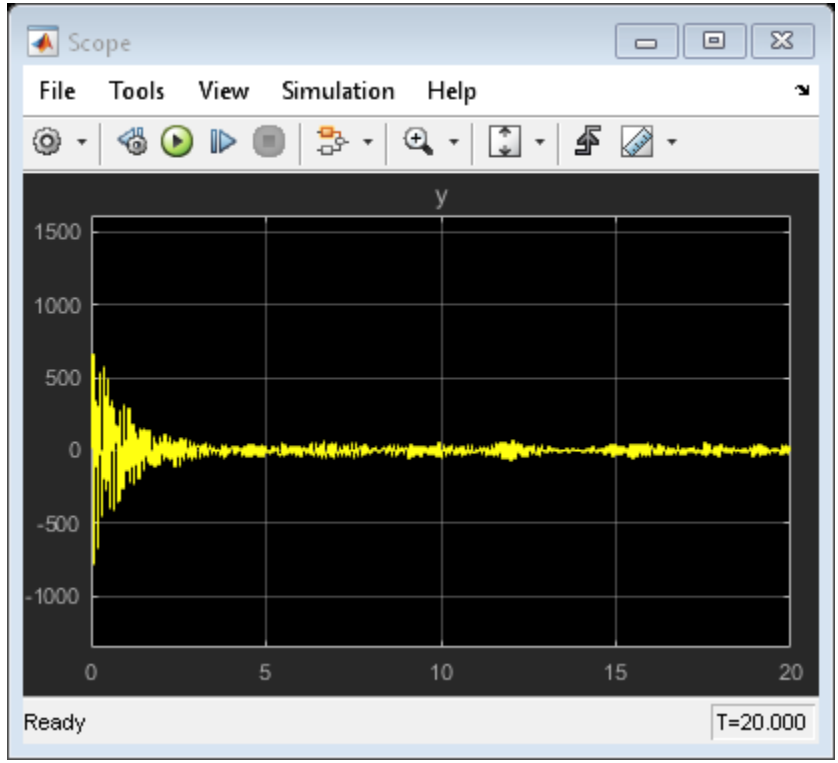
The residuals are mostly uncorrelated at nonzero lags. Having derived a model of the normal behavior we move on to investigate how the model can be used to detect faults.

Fault Detection by Residual Analysis Using Model of Healthy State

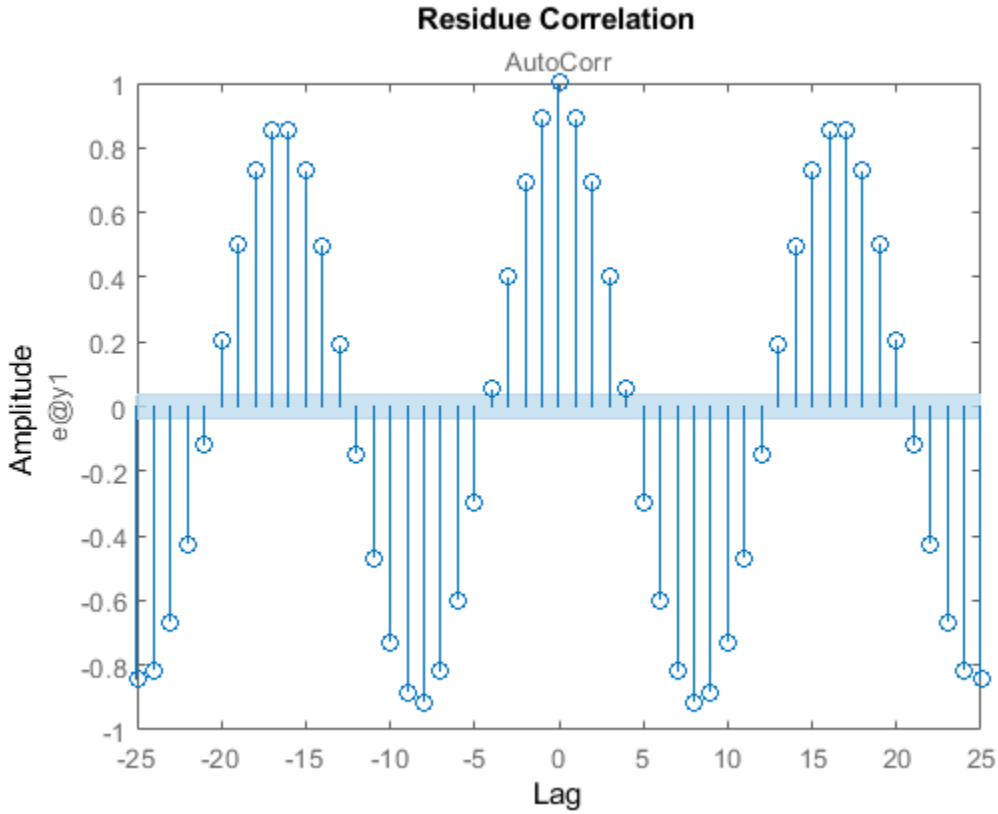
Fault detection is tagging of unwanted or unexpected changes in observations of the system. A fault causes changes in the system dynamics owing either to gradual wear and tear or sudden changes caused by sensor failure or broken parts. When a fault appears, the model obtained under normal working conditions is unable to predict the observed responses. This causes the difference between the measured and predicted response (the residuals) to increase. Such deviations are usually flagged by a large squared-sum-of-residuals or by presence of correlations.

Put the Simulink model in the damaged-system variant and simulate. We use a single bump as input since the residual test needs white input with possibly a transient owing to initial conditions.

```
set_param([sysA,'/Mechanical System'],'OverrideUsingVariant','DamagedSystem');
set_param([sysA,'/Pulse'],'Period','5120') % to force only one bump
sim(sysA)
y = logsout.getElement('y').Values;
```




```
resid(model, y.Data)
set_param([sysA,'/Pulse'],'Period','512') % restore original
```



The residuals are now larger and show correlations at non-zero lags. This is the basic idea behind detection of faults - creating a residual metric and observing how it changes with each new set of measurements. What is used here is a simple residual based on 1-step prediction error. In practice, more advanced residuals are generated that are tailor-made to the application needs.

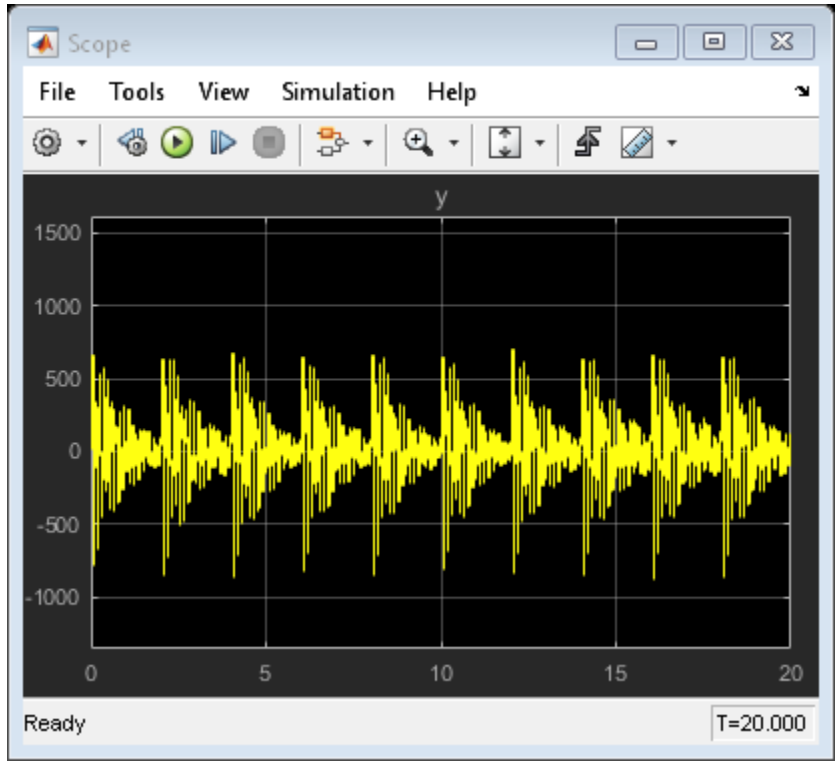
Fault Detection Using Models of Normal and Deteriorated State

A more detailed approach to fault detection is to also identify a model of the faulty (damaged) state of the system. We can then analyze which model is more likely to explain the live measurements from the system. This arrangement can be generalized to models for various types of faults and thus used for not just detecting the fault but also identifying which one ("isolation"). In this example, we take the following approach:

1. We collect data with system operating in the normal (healthy) and a known wear-and-tear induced end-of-life state.
2. We identify a dynamic model representing the behavior in each state.
3. We use a data clustering approach to draw a clear distinction between these states.
4. For fault detection, we collect data from the running machine and identify a model of its behavior. We then predict which state (normal or damaged) is most likely to explain the observed behavior.

We have already simulated the system in its normal operation mode. We now simulate the model `idMechanicalSystem` in the "end of life" mode. This is the scenario where the system has already deteriorated to its final state of permissible operation.

```
set_param([sysA,'/Mechanical System'],'OverrideUsingVariant','DamagedSystem');
sim(sysA)
y = logsout.getElement('y').Values;
zfault = cell(nr,1);
for ct = 1:nr
   z = iddata(y.Data((ct-1)*N+(1:500)),[],Ts);
   zfault{ct} = z;
end
```

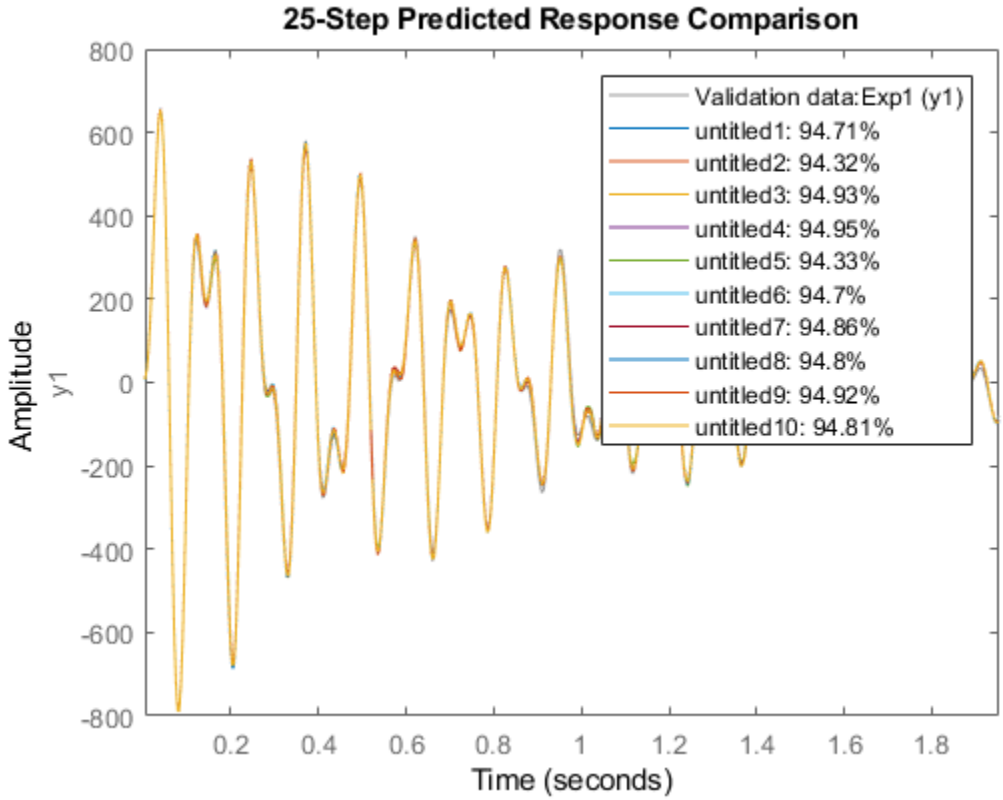


We now create a set of models, one for each data segment. As before we build 7:th order time series models in state-space form.

```
mNormal =  cell(nr,1);
mFault = cell(nr, 1);
nx = order(model);
for ct = 1:nr
   mNormal{ct} = ssest(znormal{ct}, nx, 'form', 'canonical', 'Ts', Ts);
   mFault{ct} = ssest(zfault{ct}, nx, 'form', 'canonical', 'Ts', Ts);
end
```

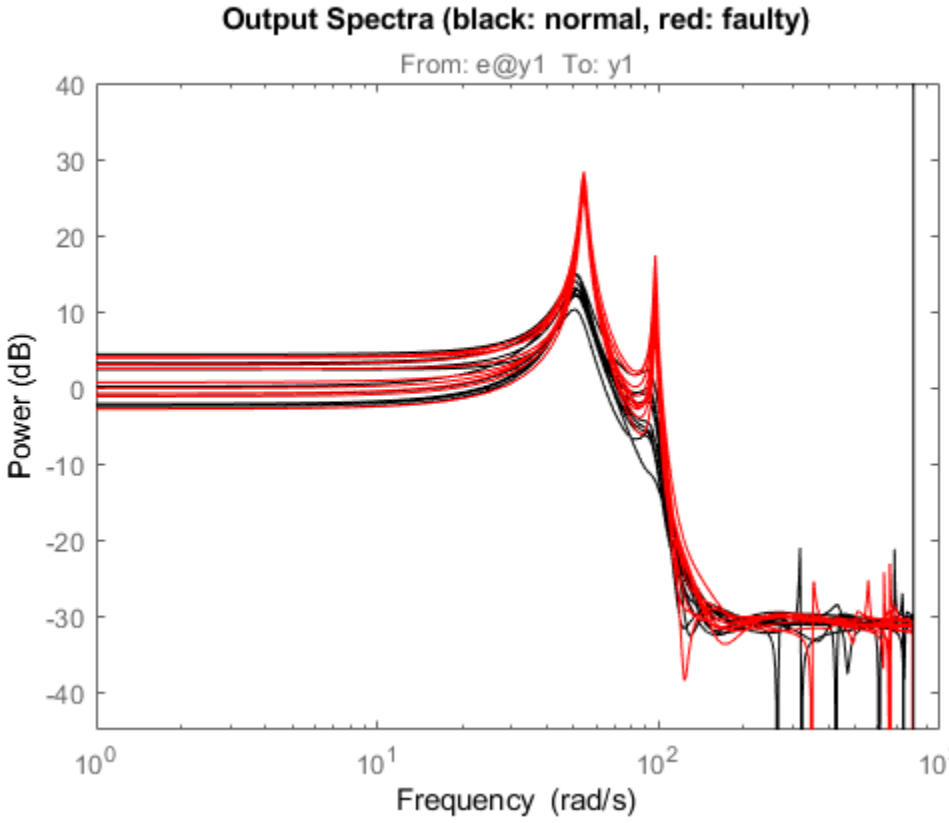
Verify that the models `mFault` are a good representation of the faulty mode of operation:

```
compare(merge(zfault{:}), mFault{:}, 25)
```



Normal and faulty estimated spectra are plotted below.

```
Color1 = 'k'; Color2 = 'r';
ModelSet1 = cat(2,mNormal,repmat({Color1},[nr, 1]))';
ModelSet2 = cat(2,mFault,repmat({Color2},[nr, 1]))';

spectrum(ModelSet1{:},ModelSet2{:})
axis([1  1000  -45  40])
title('Output Spectra (black: normal, red: faulty)')
```



The spectrum plot shows the difference: the damaged mode has its primary resonances amplified but the spectra are otherwise overlapping. Next, we create a way to quantitatively distinguish between the normal and the faulty state. We can use data clustering approaches such as:

- Fuzzy C-Means Clustering. See `fcm()` in Fuzzy Logic Toolbox.
- Support Vector Machine Classifier. See `fitcsvm ()` in Statistics and Machine Learning Toolbox.
- Self-organizing Maps. See `selforgmap()` in Neural Network Toolbox.

In this example, we use the Support Vector Machine classification technique. The clustering of information from the two types of models (`mNormal` and `mFault`) can be

based on different kinds of information that these models can provide such as the locations of their poles and zeroes, their locations of peak resonances or their list of parameters. Here, we classify the modes by the pole locations corresponding to the two resonances. For clustering, we tag the poles of the healthy state models with 'good' and the poles of the faulty state models with 'faulty';

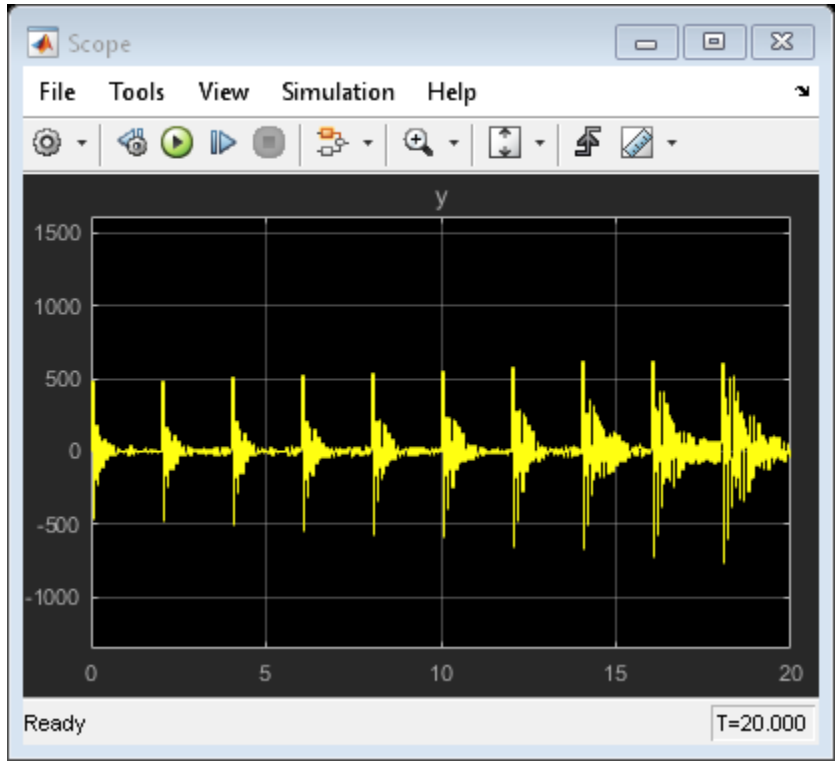
```
ModelTags = cell(nr*2,1);  % nr is number of data segments
ModelTags(1:nr) = {'good'};
ModelTags(nr+1:end) = {'faulty'};
ParData = zeros(nr*2,4);
plist = @(p)[real(p(1)),imag(p(1)),real(p(3)),imag(p(3))]; % poles of dominant resonanc
for ct = 1:nr
   ParData(ct,:) =  plist(esort(pole(mNormal{ct})));
   ParData(nr+ct,:) = plist(esort(pole(mFault{ct})));
end
cl = fitcsvm(ParData,ModelTags,'KernelFunction','rbf', ...
   'BoxConstraint',Inf,'ClassNames',{'good', 'faulty'});
cl.ConvergenceInfo.Converged


ans =

  logical

   1
```

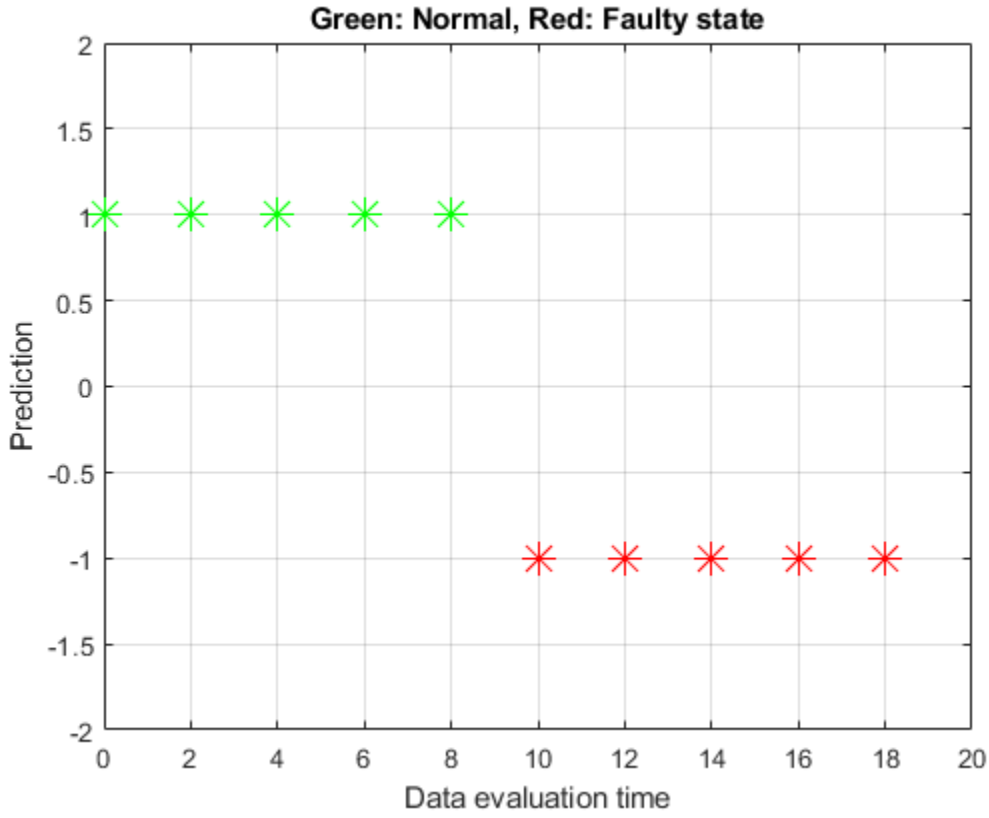
`cl` is an SVM classifier that separates the training data `ParData` into good and faulty regions. Using the `predict` method of this classifier one can assign an input nx-by-1 vector to one of the two regions.

Now we can test the classifier for its prediction (normal vs damaged) collect data batches from a system whose parameters are changing in a manner that it goes from being healthy (mode = 'Normal') to being fully damaged (mode = 'DamagedSystem') in a continuous manner. To simulate this scenario, we put the model in 'DeterioratingSystem' mode.

```
set_param([sysA,'/Mechanical System'],'OverrideUsingVariant','DeterioratingSystem');
sim(sysA)
ytv = logsout.getElement('y').Values; ytv = squeeze(ytv.Data);
PredictedMode = cell(nr,1);
for ct = 1:nr
   zSegment = iddata(ytv((ct-1)*512+(1:500)),[],Ts);
   mSegment = ssest(zSegment, nx, 'form', 'canonical', 'Ts', Ts);
```

```
    PredictedMode(ct) = predict(cl, plist(esort(pole(mSegment))));
end

I = strcmp(PredictedMode,'good');
Tags = ones(nr,1);
Tags(~I) = -1;
t = (0:5120)'*Ts;  % simulation time
Time = t(1:512:end-1);
plot(Time(I),Tags(I),'g*',Time(~I),Tags(~I),'r*','MarkerSize',12)
grid on
axis([0 20 -2 2])
title('Green: Normal, Red: Faulty state')
xlabel('Data evaluation time')
ylabel('Prediction')
```





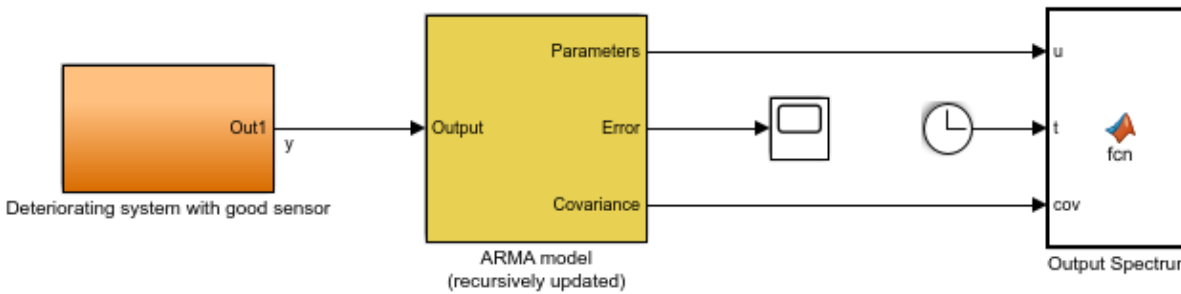
The plot shows that the classifier predicts the behavior up to about the mid-point to be normal and in a state of fault thereafter.

Fault Detection by Online Adaptation of Model Parameters

The preceding analysis used batches of data collected at different times during the operation of the system. An alternative, often more convenient, way of monitoring the health of the system is to create an adaptive model of its behavior. The new measurements are processed continuously and are used to update the parameters of a model in a recursive fashion. The effect of wear and tear or a fault is indicated by a change in the model parameter values.

Consider the wear-and-tear scenario again. As the system ages, there is a greater "rattling" which manifests itself as excitation of several resonant modes as well as a rise in the system's peak response. This scenario is described in model `idDeterioratingSystemEstimation` which is same as the 'DeterioratingSystem' mode of `idMechanicalSystem` except that the impulsive bumps that were added for offline identification are not present. The response of the system is passed to a "Recursive Polynomial Model Estimator" block which has been configured to estimate the parameters of an ARMA model structure. The actual system starts in a healthy state but deteriorates to end-of-life conditions over a time span of 200 seconds.

```
initial_model = translatecov(@(x)idpoly(x),model);
sysB = 'idDeterioratingSystemEstimation';
open_system(sysB);
```

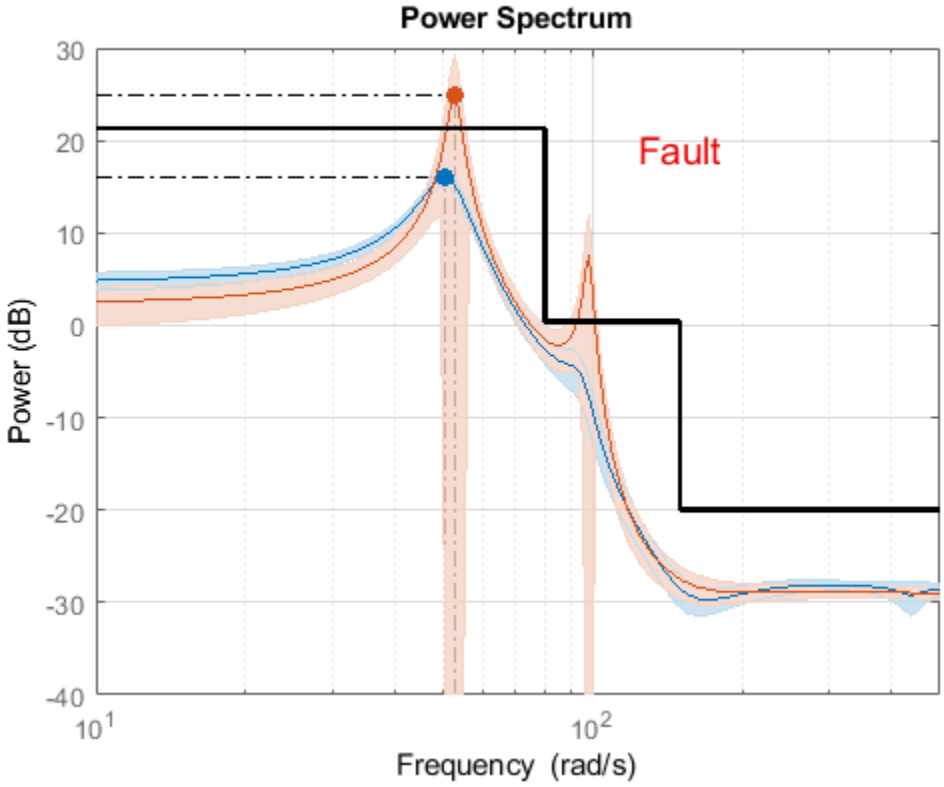


The "ARMA model" block has been initialized using the parameters and covariance data from the estimated model of normal behavior derived in the previous section after conversion to polynomial (ARMA) format. The `translatecov()` function is used so that the parameter covariance data is also converted. THe block uses a "Forgetting factor" algorithm with the forgetting factor set to slightly less than 1 to update the parameters at each sampling instant. The choice of forgetting factor influences how rapidly the system updates. A small value means that the updates will have high variance while a large value will lead make it harder for the estimator to adapt to fast changes.

The model parameters estimate is used to update the output spectrum and its 3-sd confidence region. The system will have clearly changed when the spectrum's confidence region does not overlap that of the healthy system at frequencies of interest. A fault detection threshold is shown using a black line in the plot marking the maximum allowed gains at certain frequencies. As changes in the system accumulate, the spectrum drifts across this line. This serves as a visual indicator of a fault which can be used to call for repairs in real-life systems.

Run the simulation and watch the spectrum plot as it updates.

```
sim(sysB)
```



The running estimates of model parameters are also used to compute the system pole locations which are then fed into the SVM classifier to predict if the system is in the "good" or "fault" state. This decision is also displayed on the plot. When the normalized score of prediction is less than .3, the decision is considered tentative (close to the boundary of distinction). See the script `idARMASpectrumPlot.m` for details on how the running estimate of spectrum and classifier prediction is computed.

It is possible to implement the adaptive estimation and plotting procedure outside Simulink using the `recursiveARMA()` function. Both the "Recursive Polynomial Model

Estimator" block as well as the `recursiveARMA()` function support code generation for deployment purposes.

The classification scheme can be generalized to the case where there are several known modes of failure. For this we will need multi-group classifiers where a mode refers to a certain type of failure. These aspects are not explored in this example.

Conclusions

This example showed how system identification schemes combined with data clustering approaches can assist in detection and isolation of faults. Both sequential batch analysis as well as online adaptation schemes were discussed. A model of ARMA structure of the measured output signal was identified. A similar approach can be adopted in situations where one has access to both input and output signals, and would like to employ other types of model structures such as the State-space or Box-Jenkins polynomial models.

In this example, we found that:

1. Correlations in residuals based on a model of normal operation can indicate onset of failure.
2. Gradually worsening faults can be detected by employing a continuously adapting model of the system behavior. Preset thresholds on model's characteristics such as bounds on its output spectrum can help visualize the onset and progression of failures.
3. When the source of a fault needs to be isolated, a viable approach is to create separate models of concerned failure modes beforehand. Then a clustering approach can be used to assign the predicted state of the system to one of these modes.

See Also

Related Examples

- “Perform Multivariate Time Series Forecasting” on page 17-25
- “Time Series Prediction and Forecasting for Prognosis” on page 22-2