System Identification Toolbox™

User's Guide

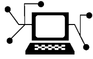
R2011b

Lennart Ljung

MATLAB® & SIMULINK®

MathWorks®

How to Contact MathWorks

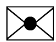


`www.mathworks.com`	Web
`comp.soft-sys.matlab`	Newsgroup
`www.mathworks.com/contact_TS.html`	Technical Support

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

508-647-7000 (Phone)

508-647-7001 (Fax)



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

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

System Identification Toolbox™ User's Guide

© COPYRIGHT 1988–2011 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)

About the Developers

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.

Contents

1 Choosing Your System Identification Approach

Linear Model Structures **1-2**

Nonlinear Model Structures **1-4**

Recommended Model Estimation Sequence **1-5**

Supported Models for Time- and Frequency-Domain Data **1-7**
Supported Models for Time-Domain Data **1-7**
Supported Models for Frequency-Domain Data **1-8**
See Also **1-9**

Supported Continuous- and Discrete-Time Models **1-10**

Model Estimation Commands **1-12**

Creating Model Structures at the Command Line **1-14**
About System Identification Toolbox Model Objects **1-14**
When to Construct a Model Structure Independently of Estimation **1-15**
Commands for Constructing Model Structures **1-16**
Model Properties **1-17**
See Also **1-23**

Modeling Multiple-Output Systems **1-24**
About Modeling Multiple-Output Systems **1-24**
Modeling Multiple Outputs Directly **1-25**
Modeling Multiple Outputs as a Combination of Single-Output Models **1-25**
Improving Multiple-Output Estimation Results by Weighing Outputs During Estimation **1-26**

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 **2-11**

Importing Data into the GUI **2-17**
Types of Data You Can Import into the GUI **2-17**
Importing Time-Domain Data into the GUI **2-18**
Importing Frequency-Domain Data into the GUI **2-22**
Importing Data Objects into the GUI **2-30**
Specifying the Data Sampling Interval **2-34**
Specifying Estimation and Validation Data **2-35**
Preprocessing Data Using Quick Start **2-36**
Creating Data Sets from a Subset of Signal Channels **2-37**
Creating Multiexperiment Data Sets in the GUI **2-39**
Managing Data in the GUI **2-45**

Representing Time- and Frequency-Domain Data Using iddata Objects **2-53**
iddata Constructor **2-53**
iddata Properties **2-56**
Creating Multiexperiment Data at the Command Line ... **2-59**
Select Data Channels, I/O Data and Experiments in iddata Objects **2-61**
Increasing Number of Channels or Data Points of iddata Objects **2-65**
Managing iddata Objects **2-67**

Representing Frequency-Response Data Using idfrd Objects **2-73**
idfrd Constructor **2-73**
idfrd Properties **2-74**
Select I/O Channels and Data in idfrd Objects **2-76**
Adding Input or Output Channels in idfrd Objects **2-77**
Managing idfrd Objects **2-80**
Operations That Create idfrd Objects **2-81**

Analyzing Data Quality **2-82**
Is Your Data Ready for Modeling? **2-82**
Plotting Data in the GUI Versus at the Command Line **2-83**
How to Plot Data in the GUI **2-83**
How to Plot Data at the Command Line **2-89**
How to Analyze Data Using the advice Command **2-91**

Selecting Subsets of Data **2-93**
Why Select Subsets of Data? **2-93**
Extract Subsets of Data Using the GUI **2-94**
Extract Subsets of Data at the Command Line **2-96**

Handling Missing Data and Outliers **2-97**
Handling Missing Data **2-97**
Handling Outliers **2-98**
Example – Extracting and Modeling Specific Data Segments **2-99**
See Also **2-100**

Handling Offsets and Trends in Data **2-101**
When to Detrend Data **2-101**
Alternatives for Detrending Data in GUI or at the Command-Line **2-102**
Next Steps After Detrending **2-103**

How to Detrend Data Using the GUI **2-104**

How to Detrend Data at the Command Line **2-105**
Detrending Steady-State Data **2-105**
Detrending Transient Data **2-105**
See Also **2-106**

Resampling Data **2-107**
What Is Resampling? **2-107**
Resampling Data Without Aliasing Effects **2-108**
See Also **2-112**

Resampling Data Using the GUI **2-113**

Resampling Data at the Command Line **2-114**

Filtering Data **2-116**
Supported Filters **2-116**
Choosing to Prefilter Your Data **2-116**
See Also **2-117**

How to Filter Data Using the GUI **2-118**
Filtering Time-Domain Data in the GUI **2-118**
Filtering Frequency-Domain or Frequency-Response Data in the GUI **2-119**

How to Filter Data at the Command Line **2-122**
Simple Passband Filter **2-122**
Defining a Custom Filter **2-123**
Causal and Noncausal Filters **2-124**

Generating Data Using Simulation **2-126**
Commands for Generating Data Using Simulation **2-126**
Example – Creating Periodic Input Data **2-127**
Example – Generating Output Data Using Simulation ... **2-128**
Simulating Data Using Other MathWorks Products **2-129**

Transforming Between Time- and Frequency-Domain Data **2-130**
Transforming Data Domain in the GUI **2-130**
Transforming Data Domain at the Command Line **2-137**

Manipulating Complex-Valued Data **2-142**
Supported Operations for Complex Data **2-142**
Processing Complex iddata Signals at the Command Line **2-142**

3 Linear Model Identification

Identifying Frequency-Response Models **3-2**
What Is a Frequency-Response Model? **3-2**
Data Supported by Frequency-Response Models **3-3**
How to Estimate Frequency-Response Models in the GUI **3-3**
How to Estimate Frequency-Response Models at the Command Line **3-5**
Selecting the Method for Computing Spectral Models **3-5**
Controlling Frequency Resolution of Spectral Models **3-6**
Spectrum Normalization **3-8**

Identifying Impulse-Response Models **3-11**
What Is Time-Domain Correlation Analysis? **3-11**
Data Supported by Correlation Analysis **3-12**
How to Estimate Impulse and Step Response Models Using the GUI **3-12**
How to Estimate Impulse and Step Response Models at the Command Line **3-14**
How to Compute Response Values **3-15**
How to Identify Delay Using Transient-Response Plots **3-16**
Correlation Analysis Algorithm **3-18**

Identifying Low-Order Transfer Functions (Process Models) **3-20**
What Is a Process Model? **3-20**
Data Supported by Process Models **3-21**
How to Estimate Process Models Using the GUI **3-21**
How to Estimate Process Models at the Command Line **3-27**
Process Model Structure Specification **3-33**
Estimating Multiple-Input Process Models **3-34**
Disturbance Model Structure for Process Models **3-35**
Assigning Estimation Weightings **3-36**
Specifying Initial States for Iterative Estimation Algorithms **3-37**

Identifying Input-Output Polynomial Models **3-39**
What Are Black-Box Polynomial Models? **3-39**
Data Supported by Polynomial Models **3-46**

Preliminary Step – Estimating Model Orders and Input Delays 3-48
How to Estimate Polynomial Models in the GUI 3-56
How to Estimate Polynomial Models at the Command Line 3-59
Estimating Multiple-Input and Multiple-Output ARX Orders 3-64
Assigning Estimation Weightings 3-65
Specifying Initial States for Iterative Estimation Algorithms 3-66
Polynomial Model Estimation Algorithms 3-66
Example – Estimating Models Using armax 3-67

Identifying State-Space Models **3-73**
What Are State-Space Models? 3-73
Data Supported by State-Space Models 3-77
Supported State-Space Parameterizations 3-78
Preliminary Step – Estimating State-Space Model Orders 3-79
How to Estimate State-Space Models in the GUI 3-84
How to Estimate State-Space Models at the Command Line 3-87
How to Estimate Free-Parameterization State-Space Models 3-91
How to Estimate State-Space Models with Canonical Parameterization 3-92
How to Estimate State-Space Models with Structured Parameterization 3-94
How to Estimate the State-Space Equivalent of ARMAX and OE Models 3-100
Assigning Estimation Weightings 3-101
Specifying Initial States for Iterative Estimation Algorithms 3-102
State-Space Model Estimation Algorithms 3-103

Refining Linear Parametric Models **3-104**
When to Refine Models 3-104
What You Specify to Refine a Model 3-104
How to Refine Linear Parametric Models in the GUI 3-105
How to Refine Linear Parametric Models at the Command Line 3-106

Extracting Numerical Model Data **3-109**

Transforming Between Discrete-Time and Continuous-Time Representations **3-112**
Why Transform Between Continuous and Discrete Time? **3-112**
Using the c2d, d2c, and d2d Commands **3-112**
Specifying Intersample Behavior **3-114**
How d2c Handles Input Delays **3-114**
Effects on the Noise Model **3-115**

Transforming Between Linear Model Representations **3-117**

Subreferencing Models **3-119**
What Is Subreferencing? **3-119**
Limitation on Supported Models **3-119**
Subreferencing Specific Measured Channels **3-119**
Subreferencing Measured and Noise Models **3-120**
Treating Noise Channels as Measured Inputs **3-122**

Concatenating Models **3-124**
About Concatenating Models **3-124**
Limitation on Supported Models **3-124**
Horizontal Concatenation of Model Objects **3-125**
Vertical Concatenation of Model Objects **3-125**
Concatenating Noise Spectrum Data of idfrd Objects **3-126**
See Also **3-127**

Merging Models **3-128**

4 Nonlinear Black-Box Model Identification

About Nonlinear Model Identification **4-2**
What Are Nonlinear Models? **4-2**
When to Fit Nonlinear Models **4-2**
Available Nonlinear Models **4-4**

Preparing Data for Nonlinear Identification **4-7**

Identifying Nonlinear ARX Models **4-8**
Nonlinear ARX Model Extends the Linear ARX Structure **4-8**
Structure of Nonlinear ARX Models **4-9**
Nonlinearity Estimators for Nonlinear ARX Models **4-10**
Ways to Configure Nonlinear ARX Estimation **4-12**
How to Estimate Nonlinear ARX Models in the GUI **4-16**
How to Estimate Nonlinear ARX Models at the Command Line **4-19**
Using Linear Model for Nonlinear ARX Estimation **4-28**
Validating Nonlinear ARX Models **4-35**
Using Nonlinear ARX Models **4-40**
How the Software Computes Nonlinear ARX Model Output **4-41**

Identifying Hammerstein-Wiener Models **4-49**
Applications of Hammerstein-Wiener Models **4-49**
Structure of Hammerstein-Wiener Models **4-50**
Nonlinearity Estimators for Hammerstein-Wiener Models **4-52**
Ways to Configure Hammerstein-Wiener Estimation **4-53**
Estimation Algorithm for Hammerstein-Wiener Models .. **4-55**
How to Estimate Hammerstein-Wiener Models in the GUI **4-55**
How to Estimate Hammerstein-Wiener Models at the Command Line **4-58**
Using Linear Model for Hammerstein-Wiener Estimation **4-64**
Validating Hammerstein-Wiener Models **4-70**
Using Hammerstein-Wiener Models **4-76**
How the Software Computes Hammerstein-Wiener Model Output **4-78**

Linear Approximation of Nonlinear Black-Box Models **4-81**
Why Compute a Linear Approximation of a Nonlinear Model? **4-81**
Choosing Your Linear Approximation Approach **4-81**
Linear Approximation of Nonlinear Black-Box Models for a Given Input **4-82**
Tangent Linearization of Nonlinear Black-Box Models ... **4-82**
Computing Operating Points for Nonlinear Black-Box Models **4-83**

5 ODE Parameter Estimation (Grey-Box Modeling)

Supported Grey-Box Models **5-2**

Data Supported by Grey-Box Models **5-3**

Choosing idgrey or idnlgrey Model Object **5-4**

Estimating Linear Grey-Box Models **5-6**
Specifying the Linear Grey-Box Model Structure **5-6**
Example – Creating a Function for Representing a Grey-Box Model **5-7**
Example – Estimating a Continuous-Time Grey-Box Model for Heat Diffusion **5-9**
Example – Estimating a Discrete-Time Grey-Box Model with Parameterized Disturbance **5-12**

Estimating Nonlinear Grey-Box Models **5-15**
Specifying the Nonlinear Grey-Box Model Structure **5-15**
Constructing the idnlgrey Object **5-17**
Using pem to Estimate Nonlinear Grey-Box Models **5-17**
Nonlinear Grey-Box Model Estimation Algorithm Options **5-18**
Represent Nonlinear Dynamics Using MATLAB File for Grey-Box Estimation **5-20**
Nonlinear Grey-Box Demos and Examples **5-38**

After Estimating Grey-Box Models **5-39**

6 Time Series Identification

What Are Time-Series Models? **6-2**

Preparing Time-Series Data **6-3**

Estimating Time-Series Power Spectra **6-4**
How to Estimate Time-Series Power Spectra Using the GUI **6-4**
How to Estimate Time-Series Power Spectra at the Command Line **6-5**

Estimating AR and ARMA Models **6-7**
Definition of AR and ARMA Models **6-7**
Estimating Polynomial Time-Series Models in the GUI ... **6-7**
Estimating AR and ARMA Models at the Command Line **6-10**

Estimating State-Space Time-Series Models **6-12**
Definition of State-Space Time-Series Model **6-12**
Estimating State-Space Models at the Command Line ... **6-12**

Example – Identifying Time-Series Models at the Command Line **6-14**

Estimating Nonlinear Models for Time-Series Data ... **6-15**

7 Recursive Model Identification

What Is Recursive Estimation? **7-2**

Commands for Recursive Estimation **7-3**

Algorithms for Recursive Estimation **7-6**
Types of Recursive Estimation Algorithms **7-6**
General Form of Recursive Estimation Algorithm **7-6**
Kalman Filter Algorithm **7-8**
Forgetting Factor Algorithm **7-10**
Unnormalized and Normalized Gradient Algorithms **7-11**

Data Segmentation **7-14**

8 Model Analysis

Validating Models After Estimation **8-3**
- When to Validate Models **8-3**
- Ways to Validate Models **8-3**
- Data for Model Validation **8-4**
- Supported Model Plots **8-5**
- Definition: Confidence Interval **8-6**

Plotting Models in the GUI **8-8**

Getting Advice About Models **8-10**

Simulating and Predicting Model Output **8-11**
- Why Simulate or Predict Model Output **8-11**
- Definition: Simulation and Prediction **8-12**
- Simulation and Prediction in the GUI **8-14**
- Simulation and Prediction at the Command Line **8-20**
- Compare Simulated Output with Measured Data **8-22**
- Simulate Model Output with Noise **8-23**
- Simulate a Continuous-Time State-Space Model **8-23**
- Predict Using Time-Series Model **8-25**

Residual Analysis **8-26**
- What Is Residual Analysis? **8-26**
- Supported Model Types **8-27**
- What Residual Plots Show for Different Data Domains . . . **8-27**
- Displaying the Confidence Interval **8-28**
- How to Plot Residuals Using the GUI **8-29**
- How to Plot Residuals at the Command Line **8-31**
- Example – Examining Model Residuals **8-31**

Impulse and Step Response Plots **8-35**
- Supported Models **8-35**
- How Transient Response Helps to Validate Models **8-35**
- What Does a Transient Response Plot Show? **8-36**
- Displaying the Confidence Interval **8-37**

How to Plot Impulse and Step Response Using the GUI 8-39

How to Plot Impulse and Step Response at the Command Line 8-42

Frequency Response Plots 8-44
What Is Frequency Response? 8-44
How Frequency Response Helps to Validate Models 8-45
What Does a Frequency-Response Plot Show? 8-46
Displaying the Confidence Interval 8-47

How to Plot Bode Plots Using the GUI 8-48

How to Plot Bode and Nyquist Plots at the Command Line 8-51

Noise Spectrum Plots 8-53
Supported Models 8-53
What Does a Noise Spectrum Plot Show? 8-53
Displaying the Confidence Interval 8-54

How to Plot the Noise Spectrum Using the GUI 8-56

How to Plot the Noise Spectrum at the Command Line 8-59

Pole and Zero Plots 8-61
Supported Models 8-61
What Does a Pole-Zero Plot Show? 8-61
Reducing Model Order Using Pole-Zero Plots 8-63
Displaying the Confidence Interval 8-63

How to Plot Model Poles and Zeros Using the GUI 8-65

How to Plot Poles and Zeros at the Command Line . . . 8-67

Akaike's Criteria for Model Validation 8-68
Definition of FPE 8-68

Computing FPE 8-69
Definition of AIC 8-69
Computing AIC 8-70

Computing Model Uncertainty **8-71**
Why Analyze Model Uncertainty? 8-71
What Is Model Covariance? 8-71
Types of Model Uncertainty Information 8-72

Troubleshooting Models **8-74**
About Troubleshooting Models 8-74
Model Order Is Too High or Too Low 8-74
Nonlinearity Estimator Produces a Poor Fit 8-75
Substantial Noise in the System 8-76
Unstable Models 8-76
Missing Input Variables 8-78
Complicated Nonlinearities 8-78

Next Steps After Getting an Accurate Model **8-79**

9 Control Design Applications

Using Identified Models for Control Design Applications **9-2**
How Control System Toolbox Software Works with Identified Models 9-2
Using balred to Reduce Model Order 9-3
Compensator Design Using Control System Toolbox Software 9-3
Converting Models to LTI Objects 9-4
Viewing Model Response Using the LTI Viewer 9-5
Combining Model Objects 9-6

Example – Using System Identification Toolbox Software with Control System Toolbox Software . . . **9-7**

10 System Identification Toolbox Blocks

Using System Identification Toolbox Blocks in Simulink Models **10-2**

Preparing Data **10-3**

Identifying Linear Models **10-4**

Simulating Identified Model Output in Simulink **10-5**
When to Use Simulation Blocks **10-5**
Summary of Simulation Blocks **10-5**
Specifying Initial Conditions for Simulation **10-6**

Example – Simulating an Identified Model Using Simulink Software **10-8**

11 System Identification Tool GUI

Steps for Using the System Identification Tool GUI . . . **11-2**

Working with the System Identification Tool GUI **11-3**
Starting and Managing GUI Sessions **11-3**
Managing Models **11-7**
Working with Plots **11-13**
Customizing the System Identification Tool GUI **11-17**
Related Examples **11-20**

Index

Choosing Your System Identification Approach

- “Linear Model Structures” on page 1-2
- “Nonlinear Model Structures” on page 1-4
- “Recommended Model Estimation Sequence” on page 1-5
- “Supported Models for Time- and Frequency-Domain Data” on page 1-7
- “Supported Continuous- and Discrete-Time Models” on page 1-10
- “Model Estimation Commands” on page 1-12
- “Creating Model Structures at the Command Line” on page 1-14
- “Modeling Multiple-Output Systems” on page 1-24

Linear 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. Available linear structures include transfer functions and state-space models, summarized in the following table.

Model Type	Usage	Learn More
Process model (`idproc` object)	Use this structure to represent low-order transfer function that include integrator, delay, zero, and up to 3 poles. You can also specify parameter bounds.	"Identifying Low-Order Transfer Functions (Process Models)" on page 3-20
State-space model (`idss` object)	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. If you need to specify parameter dependencies or constraints, use the grey-box model structure.	"Identifying State-Space Models" on page 3-73

Model Type	Usage	Learn More
Generalized transfer function (`idpoly` object)	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.	"Identifying Input-Output Polynomial Models" on page 3-39
Grey-box model (`idgrey` object)	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.	"Estimating Linear Grey-Box Models" on page 5-6

Nonlinear Model Structures

System Identification Toolbox provides several nonlinear black-box model structures, which have traditionally been useful for representing dynamic systems.

Model Type	Usage	Learn More
Nonlinear ARX model (`idnlarx` object)	Use 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.	"Identifying Nonlinear ARX Models" on page 4-8
Linear models with input/output nonlinearities, or Hammerstein-Wiener model (`idnlhw` object)	Use to represent linear models with static nonlinearities.	"Identifying Hammerstein-Wiener Models" on page 4-49
Nonlinear grey-box model (`idnlgrey` object)	Use to represent nonlinear ODEs with unknown parameters.	"Estimating Nonlinear Grey-Box Models" on page 5-15

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. Then, estimate parametric models in the following order:

1. ARX polynomial and state-space models provide the simplest structures. These models let you estimate the model order and noise dynamics.

   In the System Identification Tool GUI. Select to estimate the ARX linear parametric model and the state-space model using the N4SID method.

   At the command line. Use the `arx` and the `n4sid` commands.

   For more information, see "Identifying Input-Output Polynomial Models" on page 3-39 and "Identifying State-Space Models" on page 3-73.

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 Tool GUI. Select to estimate the BJ and ARMAX linear parametric models.

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

For more information, see “Identifying Input-Output Polynomial Models” on page 3-39.

3. Nonlinear ARX or Hammerstein-Wiener models provide nonlinear structures. For more information, see Chapter 4, “Nonlinear Black-Box Model Identification”.

For general information about choosing you model strategy, see “About System Identification”. For information about validating models, see “Validating Models After Estimation” on page 8-3.

Supported Models for Time- and Frequency-Domain Data

In this section...

"Supported Models for Time-Domain Data" on page 1-7

"Supported Models for Frequency-Domain Data" on page 1-8

"See Also" on page 1-9

Supported Models for Time-Domain Data

Continuous-Time Models

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

- Low-order transfer functions. See "Identifying Low-Order Transfer Functions (Process Models)" on page 3-20.
- State-space models. See "Identifying State-Space Models" on page 3-73.

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.

Discrete-Time Models

You can estimate all linear and nonlinear models supported by the System Identification Toolbox product as discrete-time models, except the continuous-time transfer functions (process models).

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. See Chapter 5, "ODE Parameter Estimation (Grey-Box Modeling)".

Nonlinear Models

You can estimate discrete-time Hammerstein-Wiener and nonlinear ARX models from time-domain data. See Chapter 4, "Nonlinear Black-Box Model Identification".

You can also estimate nonlinear grey-box models from time-domain data. See "Estimating Nonlinear Grey-Box Models" on page 5-15.

Supported Models for Frequency-Domain Data

There are two types of frequency-domain data:

- Continuous-time data
- Discrete-time data

You specify frequency-domain data as continuous- or discrete-time when you either import data into the System Identification Tool GUI or create a System Identification Toolbox data object. For more information about representing your data as System Identification Toolbox data objects, see Chapter 2, "Data Import and Processing".

To designate discrete-time data, you set the sampling interval of the data to the experimental data sampling interval. To designate continuous-time data, you must set the sampling interval of the data to zero. Setting the sampling interval to zero corresponds to taking a Fourier transform of continuous-time data.

Continuous-Time Models

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

- Low-order transfer functions. See "Identifying Low-Order Transfer Functions (Process Models)" on page 3-20.
- Input-output polynomial models. See "Identifying Input-Output Polynomial Models" on page 3-39.
- State-space models.

  From continuous-time frequency-domain data, you can estimate continuous-time state-space models. From discrete-time frequency-domain data, you can estimate continuous-time black-box models with canonical parameterization. See "Identifying State-Space Models" on page 3-73.

To get a linear, continuous-time model of arbitrary structure for frequency-domain data, you can estimate a discrete-time model and use `d2c` to transform it to a continuous-time model.

Discrete-Time Models

You can estimate only output-error (OE) polynomial models using frequency-domain data. See "Identifying Input-Output Polynomial Models" on page 3-39.

Other linear model structures include noise models, which are not supported for frequency-domain data.

ODEs (Grey-Box Models)

For linear grey-box models, you can estimate both continuous-time and discrete-time models from frequency-domain data.

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

See Chapter 5, "ODE Parameter Estimation (Grey-Box Modeling)".

Nonlinear Black-Box Models

Frequency-domain data is not relevant to nonlinear black-box models, which support only time-domain data.

See Also

"Supported Continuous- and Discrete-Time Models" on page 1-10

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
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. For frequency-domain data, you can directly estimate only the ARX and output-error (OE) continuous-time polynomial models by setting the sampling interval of the data to `0`. 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. For continuous-time frequency-domain data, you can estimate continuous-time state-space models directly.

Supported Continuous-Time Models (Continued)

Model Type	Description
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 sampling interval must be set to the (nonzero) value you used to sample in
Nonlinear black-box models	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 (ODE) 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

The quickest way to both construct a model object and estimate the model parameters is to use estimation commands.

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

For ARMAX, Box-Jenkins, and Output-Error Models—which you can only estimate using the iterative prediction-error method—use the `armax`, `bj`, and `oe` estimation commands, respectively. For more information about choosing the models to estimate first, see "Recommended Model Estimation Sequence" on page 1-5.

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
Continuous-time low-order transfer functions (process models)	`pem`
Linear input-output polynomial models	`armax` (ARMAX only) `arx` (ARX only) `bj` (BJ only) `iv4` (ARX only) `oe` (OE only) `pem` (for all models)
State-space models	`n4sid` `pem`

Commands for Constructing and Estimating Models (Continued)

Model Type	Estimation Commands

Linear time-series models	`ar`
	`arx` (for multiple outputs)
	`ivar`

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

Creating Model Structures at the Command Line

In this section...

"About System Identification Toolbox Model Objects" on page 1-14

"When to Construct a Model Structure Independently of Estimation" on page 1-15

"Commands for Constructing Model Structures" on page 1-16

"Model Properties" on page 1-17

"See Also" on page 1-23

About System Identification Toolbox Model Objects

Objects are based on 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, `idpoly` represents linear input-output polynomial models, and `idss` represents linear state-space models. For a complete list of available model objects, see "Commands for Constructing Model Structures" on page 1-16.

Model *properties* define how a model object stores information. Model objects store information about a model, including the mathematical form of a model, names of input and output channels, units, names and values of estimated parameters, parameter uncertainties, algorithm specifications, and estimation information. For example, the `idpoly` model class has a property called `InputName` for storing one or more input channel names. Different model objects have different properties.

The allowed operations on an object are called *methods*. In the System Identification Toolbox product, some methods have the same name but apply to multiple model objects. For example, the method `bode` creates a bode plot for all linear model objects. However, other methods are unique to a specific

model object. For example, the estimation method `n4sid` is unique to the state-space model object `idss`.

Every class has a special method for creating objects of that class, called the *constructor*. 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, `idpoly` is both the name of the class representing linear black-box polynomial models and the name of the constructor for instantiating the model object.

For a tutorial about estimating models at the command line, see "Tutorial – Identifying Linear Models Using the Command Line" in *System Identification Toolbox Getting Started Guide*.

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 a model
- Analyze a model
- Specify an initial guess for specific model parameter values before estimation

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 `pem` lets you specify both the model structure with unknown parameters and the estimation algorithm. For information about how to both construct and estimate models with a single command, see "Model Estimation Commands" on page 1-12.

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. For more information, see Chapter 5, "ODE Parameter Estimation (Grey-Box Modeling)".

Commands for Constructing Model Structures

The following table summarizes the model constructors available in the System Identification Toolbox product for representing various types of 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 1-12.

Summary of Model Constructors

Model Constructor	Resulting Model Class	Single or Multiple Outputs?
`idarx`	Parametric multiple-output ARX models. Also represents nonparametric transient-response models.	Single- or multiple-output models.
`idfrd`	Nonparametric frequency-response model.	Single- or multiple-output models.
`idproc`	Continuous-time, low-order transfer functions (process models).	Single-output models only.
`idpoly`	Linear input-output polynomial models: • ARX • ARMAX • Output-Error • Box-Jenkins	Single-output models only.
`idss`	Linear state-space models.	Single- or multiple-output models.

Summary of Model Constructors (Continued)

Model Constructor	Resulting Model Class	Single or Multiple Outputs?
`idgrey`	Linear ordinary differential or difference equations (grey-box models). You write a function that translates user parameters to state-space matrices.	Single- and multiple-output models.
`idnlgrey`	Nonlinear ordinary differential or difference equation (grey-box models). You write a function or MEX-file to represent the set of first-order differential or difference equations.	Supports single- and multiple-output models.
`idnlarx`	Nonlinear ARX models, which define the predicted output as a nonlinear function of past inputs and outputs.	Single- or multiple-output models.
`idnlhw`	Nonlinear Hammerstein-Wiener models, which include a linear dynamic system with nonlinear static transformations of inputs and outputs.	Single- or multiple-output models. Does not support time series.

For more information about when to use these commands, see "When to Construct a Model Structure Independently of Estimation" on page 1-15.

Model Properties

- "Categories of Model Properties" on page 1-18
- "Specifying Model Properties for Estimation" on page 1-19
- "Viewing Model Properties and Estimated Parameters" on page 1-20
- "Getting Help on Model Properties at the Command Line" on page 1-22

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. However, the `idarx`, `idgrey`, `idpoly`, `idproc`, and `idss` model objects are based on the `idmodel` superclass and inherit all `idmodel` properties.

Similarly, the nonlinear models `idnlarx`, `idnlhw`, and `idnlgrey` are based on the `idnlmodel` superclass and inherit all `idnlmodel` properties.

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

- Names of input and output channels, such as `InputName` and `OutputName`
- Sampling interval 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 and model uncertainty
- User comments, such as `Notes` and `Userdata`
- Estimation algorithm information
  - `Algorithm`

    Structure includes fields that specify the estimation method. `Algorithm` includes another structure, called `Advanced`, which provides additional flexibility for setting the search algorithm. Different fields apply to different estimation techniques.

    For linear parametric models, `Algorithm` specifies the frequency weighing of the estimation using the `Focus` property.

Note `Algorithm` does not apply to `idfrd` models.

- EstimationInfo

  Structure includes read-only fields that describe the estimation data set, quantitative model quality measures, search termination conditions, how the initial states are handled, and any warnings encountered during the estimation.

For information about getting help on object properties, see "Getting Help on Model Properties at the Command Line" on page 1-22.

Specifying Model Properties for Estimation

If you are estimating a new model, you can specify model properties directly in the estimator syntax. For a complete list of model estimation commands, see "Model Estimation Commands" on page 1-12.

When using the commands that let you both construct and estimate a model, you can specify all top-level model properties in the estimator syntax. Top-level properties are those listed when you type `get(object_name)`. You can also specify the top-level fields of the `Algorithm` structure directly in the estimator using property-value pairs—such as `focus` in the previous example—without having to define the structure fields first.

The following commands load the sample data, `z8`, construct an ARMAX model, and estimate the model parameters. The arguments of the `armax` estimator specify model properties as property-value pairs.

```
load iddata8
m_armax=armax(z8,'na',4,...
                 'nb',[3 2 3],...
                 'nc',4,...
                 'nk',[0 0 0],...
                 'focus', 'simulation',...
                 'covariance', 'none',...
                 'tolerance',1e-5,...
                 'maxiter',50);
```

`focus`, `covariance`, `tolerance`, and `maxiter` are fields in the `Algorithm` model property and specify aspects of the estimation algorithm.

For linear models, you can use a shortcut to specify the second-level `Algorithm` properties, such as `Advanced`. With this syntax, you can reference the structure fields by name without specifying the structure to which these fields belong.

However, when estimating nonlinear black-box models, you must set the specific fields of the `Advanced` `Algorithm` structure and the nonlinearity estimators before estimation. For example, suppose you want to set the value of the `wavenet` object property `Options`, which is a structure. The following commands set the `Options` values before estimation and include the modified `wavenet` object in the estimator:

```
% Define wavenet object with defaul properties
W = wavenet;
% Specify variable to represent Options field
O = W.Options;
% Modify values of specific Options fields
O.MaxLevels = 5 ;
O.DilationStep = 2;
% Estimate model using new Options settings
M = nlarx(data,[2 2 1],wavenet('options',O))
```

where `O` specifies the values of the `Options` structure fields and `M` is the estimated model. For more information about these and other commands, see the corresponding reference page.

Viewing Model Properties and Estimated Parameters

To view all the properties and values of any model object, use the `get` command. For example, type the following at the prompt to 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)
```

To access a specific property, use dot notation. For example, to view the *A* matrix containing the estimated parameters in the previous model, type the following command:

```
m_arx.a
```

```
ans =
    1.0000   -0.8441   -0.1539    0.2278    0.1239
```

Similarly, to access the uncertainties in these parameter estimates, type the following command:

```
m_arx.da
ans =
    0    0.0357    0.0502    0.0438    0.0294
```

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.

To change property values for an existing model object, use the `set` command or dot notation. For example, to change the input delays for all three input channels to `[1 1 1]`, type the following at the prompt:

```
set(m_arx,'nk',[1 1 1])
```

or equivalently

```
m_arx.nk = [1 1 1]
```

Some model properties, such as `Algorithm`, are structures. To access the fields in this structure, use the following syntax:

```
model.algorithm.PropertyName
```

where *`PropertyName`* represents any of the `Algorithm` fields. For example, to change the maximum number of iterations using the `MaxIter` property, type the following command:

```
m_arx.algorithm.MaxIter=50
```

To verify the new property value, type the following:

```
m_arx.algorithm.MaxIter
```

Note *`PropertyName`* refers to fields in a structure and is case sensitive. You must type the entire property name. Use the **Tab** key when typing property names to get completion suggestions.

Getting Help on Model Properties at the Command Line

If you need to learn more about model properties while working at the command line, you can use the `idprops` command to list the properties and values for each object.

Some model objects are based on the superclasses `idmodel` and `idnlmodel` and inherit the properties of these superclasses. For such model objects, you must independently look up the properties for both the model object and for its superclass.

The following table summarizes the commands for getting help on object properties.

Help Commands for Model Properties

Model Class	Help Commands
`idarx`	`idprops idarx` Also inherits properties from `idmodel`.
`idfrd`	`idprops idfrd`
`idnlmodel`	`idprops idnlmodel`

`idmodel`	`idprops idmodel`
	`idprops idmodel Algorithm`
	`idprops idmodel EstimationInfo`
	Also see the `Algorithm` and `EstimationInfo` reference page.

`idproc`	`idprops idproc` Also inherits properties from `idmodel`.
`idpoly`	`idprops idpoly` Also inherits properties from `idmodel`.
`idss`	`idprops idss` Also inherits properties from `idmodel`.
`idgrey`	`idprops idgrey` Also inherits properties from `idmodel`.

`idnlgrey`	`idprops idnlgrey`
	`idprops idnlgrey Algorithm`
	`idprops idnlgrey EstimationInfo`
	Also inherits properties from `idnlmodel`.

Help Commands for Model Properties (Continued)

Model Class	Help Commands
idnlarx	idprops idnlarx idprops idnlarx Algorithm idprops idnlarx EstimationInfo Also inherits properties from idnlmodel.
idnlhw	idprops idnlhw idprops idnlhw Algorithm idprops idnlhw EstimationInfo Also inherits properties from idnlmodel.

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 8-3.

Modeling Multiple-Output Systems

In this section...

"About Modeling Multiple-Output Systems" on page 1-24

"Modeling Multiple Outputs Directly" on page 1-25

"Modeling Multiple Outputs as a Combination of Single-Output Models" on page 1-25

"Improving Multiple-Output Estimation Results by Weighing Outputs During Estimation" on page 1-26

About Modeling Multiple-Output Systems

You can estimate multiple-output model directly using all the 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 1-25
- "Modeling Multiple Outputs as a Combination of Single-Output Models" on page 1-25

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 1-26.

Modeling Multiple Outputs Directly

You can estimate the following types of models for multiple-output data:

- Impulse- and step-response models
- Frequency-response models
- Linear ARX models
- State-space models
- Nonlinear ARX and Hammerstein-Wiener models
- Linear and nonlinear ODEs

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 2-91.

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 sections in the *System Identification Toolbox User's Guide*:

- For working in the System Identification Tool GUI, see "Creating Data Sets from a Subset of Signal Channels" on page 2-37.

- For working at the command line, see the "Select Data Channels, I/O Data and Experiments in iddata Objects" on page 2-61.

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 2-65 or "Adding Input or Output Channels in idfrd Objects" on page 2-77.

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.

You can specify output weights directly in the estimation command using the `Criterion` and `Weighting` fields of the `Algorithm` property. You must set the `Criterion` field to `Trace`, and set the `Weighting` field to the matrix that contains the output weights. The `Trace` criterion minimizes the weighted sum of the prediction errors using the weights specified by `Weighting`.

The following code snippet shows how to specify the `Criterion` and `Weighting` `Algorithm` fields as part of the `pem` command:

```
model=pem(z,2,'criterion','trace','weighting',diag(Q,1))
```

where `Q` is a vector of positive values and the higher values for outputs to be emphasized more during estimation.

You set `Weighting` to a positive semi-definite symmetric matrix of size equal to number of outputs. By default, `Weighting` is an identity matrix, which means that all outputs are weighed equally during estimation.

For more information about these Algorithm fields for linear estimation, see the `Algorithm Properties` reference page. For more information about the `Algorithm` fields for nonlinear estimation, see the `idnlarx` and `idnlhw` reference pages.

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

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
- "Importing Data into the GUI" on page 2-17
- "Representing Time- and Frequency-Domain Data Using iddata Objects" on page 2-53
- "Representing Frequency-Response Data Using idfrd Objects" on page 2-73
- "Analyzing Data Quality" on page 2-82
- "Selecting Subsets of Data" on page 2-93
- "Handling Missing Data and Outliers" on page 2-97
- "Handling Offsets and Trends in Data" on page 2-101
- "How to Detrend Data Using the GUI" on page 2-104
- "How to Detrend Data at the Command Line" on page 2-105
- "Resampling Data" on page 2-107
- "Resampling Data Using the GUI" on page 2-113
- "Resampling Data at the Command Line" on page 2-114
- "Filtering Data" on page 2-116
- "How to Filter Data Using the GUI" on page 2-118
- "How to Filter Data at the Command Line" on page 2-122

- "Generating Data Using Simulation" on page 2-126
- "Transforming Between Time- and Frequency-Domain Data" on page 2-130
- "Manipulating Complex-Valued Data" on page 2-142

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 1-7.

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

Your 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.

Supported Data	Description
Frequency-domain data	Fourier transform of the input and output time-domain signals.
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*.

Note If your data is complex valued, see "Manipulating Complex-Valued Data" on page 2-142 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 important 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 Tool GUI 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 "Generating Data Using Simulation" on page 2-126.

  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 "Exporting Signal Data Using Signal Logging" in the Simulink documentation.

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 GUI, import data into the System Identification Tool GUI.

  See "Importing Data into the GUI" on page 2-17.

- 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 2-53.

  For frequency-response data, see "Representing Frequency-Response Data Using idfrd Objects" on page 2-73.

- To simulate data with and without noise, see "Generating Data Using Simulation" on page 2-126.

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 "Analyzing Data Quality" on page 2-82.

- 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 2-91.

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 2-97.
- Offsets and drifts in signal levels (low-frequency disturbances).

  See “Handling Offsets and Trends in Data” on page 2-101 for information about subtracting means and linear trends, and “Filtering Data” on page 2-116 for information about filtering.
- High-frequency disturbances above the frequency interval of interest for the system dynamics.

  See “Resampling Data” on page 2-107 for information about decimating and interpolating values, and “Filtering Data” on page 2-116 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 “Selecting Subsets of Data” on page 2-93.

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 “Creating Multiexperiment Data Sets in the GUI” on page 2-39 or “Creating Multiexperiment Data at the Command Line” on page 2-59.

Requirements on Data Sampling

A *sampling interval* is the time between successive data samples.

The System Identification Tool GUI 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 2-54.

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 2-11

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

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 sampling interval. If you are working with uniformly sampled data, use the actual sampling interval from your experiment. Each data value is assigned a sample time, which is calculated from the start time and sampling interval. 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 2-54.

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 *MATLAB Data Import and Export*.

After you have the variables in the MATLAB workspace, import them into the System Identification Tool GUI or create a data object for working at the command line. For more information, see "Importing Time-Domain Data into the GUI" on page 2-18 and "Representing Time- and Frequency-Domain Data Using iddata Objects" on page 2-53.

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 sampling interval. If you are working with uniformly sampled data, use the actual sampling interval from your experiment. Each data value is assigned a sample time, which is calculated from the start time and the sampling interval. 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 2-54.

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

After you have the variables in the MATLAB workspace, import them into the System Identification Tool GUI or create a data object for working at the command line. For more information, see "Importing Time-Domain Data into

the GUI” on page 2-18 and “Representing Time- and Frequency-Domain Data Using iddata Objects” on page 2-53.

For information about estimating time-series model parameters, see Chapter 6, “Time Series Identification”.

Frequency-Domain Data Representation

Frequency-domain data consists of either transformed input and output time-domain signals or system frequency response 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 sampling interval 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:

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

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

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

    $y\left(e^{i\omega kT}\right)$

  $k=1, 2, ..., N_f$, 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 *MATLAB Data Import and Export*.

After you have the variables in the MATLAB workspace, import them into the System Identification Tool GUI or create a data object for working at the command line. For more information, see "Importing Frequency-Domain Input/Output Signals into the GUI" on page 2-23 and "Representing Time- and Frequency-Domain Data Using iddata Objects" on page 2-53.

Frequency-Response Data Representation

- "What Is Frequency-Response Data?" on page 2-13
- "How to Represent Frequency-Response Data in MATLAB" on page 2-15

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 sampling interval 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 Tool GUI. At the command line, you must represent complex data using `idfrd` object. If the data is in amplitude and phase format, convert it to complex frequency-response vector using $h(\omega) = A(\omega)e^{j\varphi(\omega)}$.

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

Rrequency-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.
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 *MATLAB Data Import and Export*.

After you have the variables in the MATLAB workspace, import them into the System Identification Tool GUI or create a data object for working at the command line. For more information about importing data into the GUI, see "Importing Frequency-Response Data into the GUI" on page 2-26. To learn more about creating a data object, see "Representing Frequency-Response Data Using idfrd Objects" on page 2-73.

Importing Data into the GUI

In this section...

"Types of Data You Can Import into the GUI" on page 2-17

"Importing Time-Domain Data into the GUI" on page 2-18

"Importing Frequency-Domain Data into the GUI" on page 2-22

"Importing Data Objects into the GUI" on page 2-30

"Specifying the Data Sampling Interval" on page 2-34

"Specifying Estimation and Validation Data" on page 2-35

"Preprocessing Data Using Quick Start" on page 2-36

"Creating Data Sets from a Subset of Signal Channels" on page 2-37

"Creating Multiexperiment Data Sets in the GUI" on page 2-39

"Managing Data in the GUI" on page 2-45

Types of Data You Can Import into the GUI

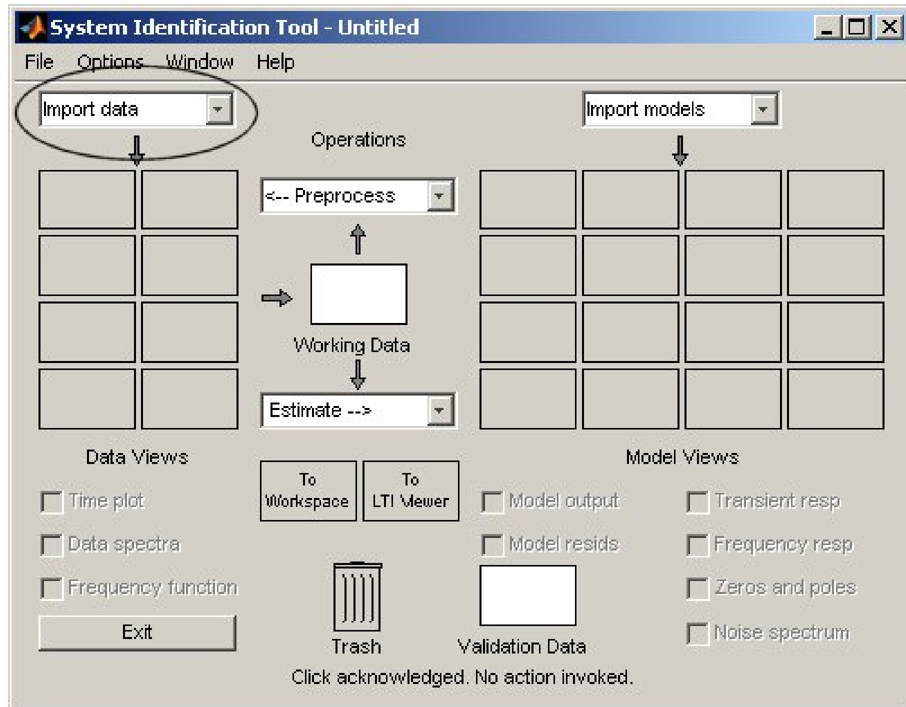
You can import the following types of data from the MATLAB workspace into the System Identification Tool GUI:

- "Importing Time-Domain Data into the GUI" on page 2-18
- "Importing Frequency-Domain Input/Output Signals into the GUI" on page 2-23
- "Importing Frequency-Response Data into the GUI" on page 2-26
- "Importing Data Objects into the GUI" on page 2-30

To open the GUI, type the following command in the MATLAB Command Window:

```
ident
```

In the **Import data** list, select the type of data to import from the MATLAB workspace, as shown in the following figure.



For an example of importing data into the System Identification Tool GUI, see the Getting Started documentation.

Importing Time-Domain Data into the GUI

Before you can import time-domain data into the System Identification Tool GUI, 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 GUI:

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

```
ident
```

2 In the System Identification Tool 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 Tool window after the import operation is completed.
- **Starting time** — Enter the starting value of the time axis for time plots.
- **Sampling interval** — Enter the actual sampling interval in the experiment. For more information about this setting, see "Specifying the Data Sampling Interval" on page 2-34.

Tip The System Identification Toolbox product uses the sampling interval 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 sampling interval.

---

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 sampling interval).

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 a string to specify 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 a string to specify the name of one or more output channels.

Physical Units of Variables

- **Input** — Enter a string to specify 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 a string to specify 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 Tool window.

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

Importing Frequency-Domain Data into the GUI

- "Importing Frequency-Domain Input/Output Signals into the GUI" on page 2-23
- "Importing Frequency-Response Data into the GUI" on page 2-26

Importing Frequency-Domain Input/Output Signals into the GUI

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 Tool GUI, 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 GUI:

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

   ```
   ident
   ```

2. In the System Identification Tool 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 Tool window after the import operation is completed.
   - **Frequency unit** — Enter `Hz` for Hertz or keep the `rad/s` default value.
   - **Sampling interval** — Enter the actual sampling interval in the experiment. For continuous-time data, enter `0`. For more information about this setting, see "Specifying the Data Sampling Interval" on page 2-34.
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 sampling interval).

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 a string to specify 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 a string to specify the name of one or more output channels.

Physical Units of Variables

- **Input** — Enter a string to specify 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 a string to specify 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 Tool window.

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

Importing Frequency-Response Data into the GUI

- "Prerequisite" on page 2-26
- "Importing Complex-Valued Frequency-Response Data" on page 2-26
- "Importing Amplitude and Phase Frequency-Response Data" on page 2-28

Prerequisite. Before you can import frequency-response data into the System Identification Tool GUI, 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 GUI:

```
ident
```

2 In the System Identification Tool 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:

- **Freq. Func.** — 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 Tool window after the import operation is completed.
- **Frequency unit** — Enter `Hz` for Hertz or keep the `rad/s` default value.
- **Sampling interval** — Enter the actual sampling interval in the experiment. For continuous-time data, enter `0`. For more information about this setting, see "Specifying the Data Sampling Interval" on page 2-34.

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

Channel Names

- **Input** — Enter a string to specify 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 a string to specify the name of one or more output channels.

Physical Units of Variables

- **Input** — Enter a string to specify 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 a string to specify 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 Tool 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 theMATLAB Command Window to open the GUI:

   ```
   ident
   ```

2. In the System Identification Tool 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 Tool window after the import operation is completed.
- **Frequency unit** — Enter `Hz` for Hertz or keep the `rad/s` default value.
- **Sampling interval** — Enter the actual sampling interval in the experiment. For continuous-time data, enter `0`. For more information about this setting, see "Specifying the Data Sampling Interval" on page 2-34.

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

Channel Names

- **Input** — Enter a string to specify 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 a string to specify the name of one or more output channels.

Physical Units of Variables

- **Input** — Enter a string to specify 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 a string to specify 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 Tool window.

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

Importing Data Objects into the GUI

You can import the System Identification Toolbox `iddata` and `idfrd` data objects into the System Identification Tool GUI.

Before you can import a data object into the System Identification Tool GUI, you must create the data object in the MATLAB workspace, as described in "Representing Time- and Frequency-Domain Data Using iddata Objects" on page 2-53 or "Representing Frequency-Response Data Using idfrd Objects" on page 2-73.

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.

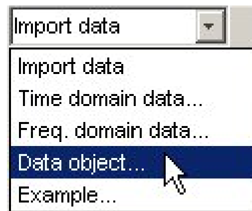
Import `iddata`, `idfrd`, or `frd` data object in the MATLAB workspace.

To import a data object into the GUI:

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

```
ident
```

2 In the System Identification Tool 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 Tool 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.
- **Sampling interval** — Enter the actual sampling interval in the experiment. For more information about this setting, see "Specifying the Data Sampling Interval" on page 2-34.

Tip The System Identification Toolbox product uses the sampling interval 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 sampling interval.

---

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 sampling interval).

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 a string to specify 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 a string to specify the name of one or more output channels.

Physical Units of Variables

- **Input** — Enter a string to specify 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 a string to specify 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 Tool window.

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

Specifying the Data Sampling Interval

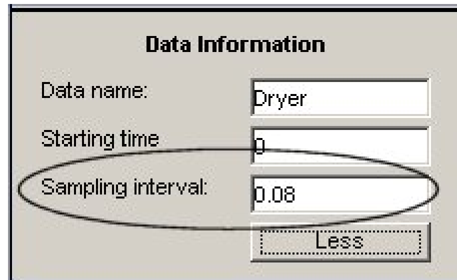
When you import data into the GUI, you must specify the data sampling interval.

The *sampling interval* 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 sampling interval as a flag to specify continuous-time data. When importing continuous-time frequency domain or frequency-response data, set the **Sampling interval** to `0`.

The sampling interval is used during model estimation. For time-domain data, the sampling interval 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 sampling

interval. In addition, the sampling instants are used to set the horizontal axis on time plots.



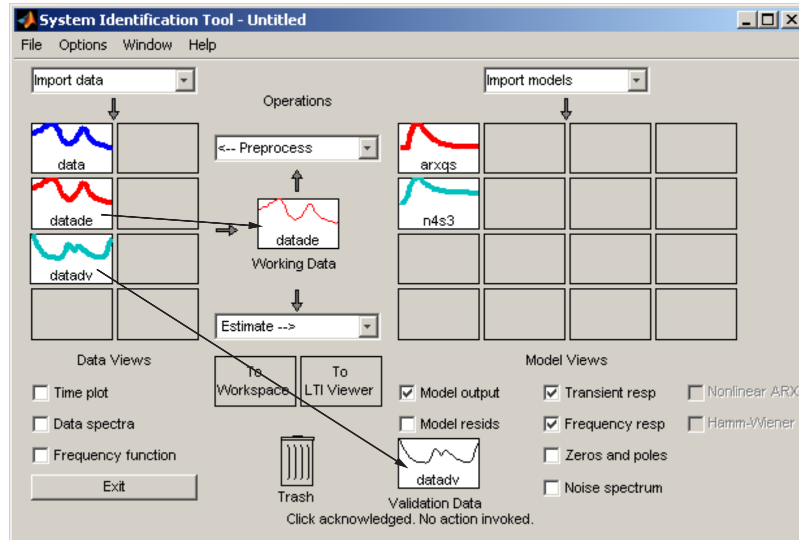
Sampling Interval in the Import Data dialog box

Specifying Estimation and Validation Data

You should use different data sets to estimate and validate your model for best validation results.

In the System Identification Tool GUI, **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.

Preprocessing 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 2-101.

- Split data into two parts.
- Specify the first part as estimation data for models (or **Working Data**).
- Specify the second part as **Validation Data**.

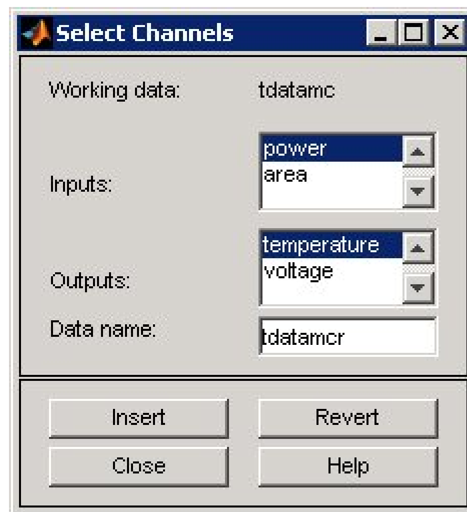
Creating Data Sets from a Subset of Signal Channels

You can create a new data set in the System Identification Tool GUI 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 Tool GUI, 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 Tool GUI.

7 Click **Close**.

Creating Multiexperiment Data Sets in the GUI

- "Why Create Multiexperiment Data?" on page 2-39
- "Limitations on Data Sets" on page 2-39
- "Merging Data Sets" on page 2-39
- "Extracting Specific Experiments from a Multiexperiment Data Set into a New Data Set" on page 2-43

Why Create Multiexperiment Data?

You can create a time-domain or frequency-domain data set in the System Identification Tool GUI 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

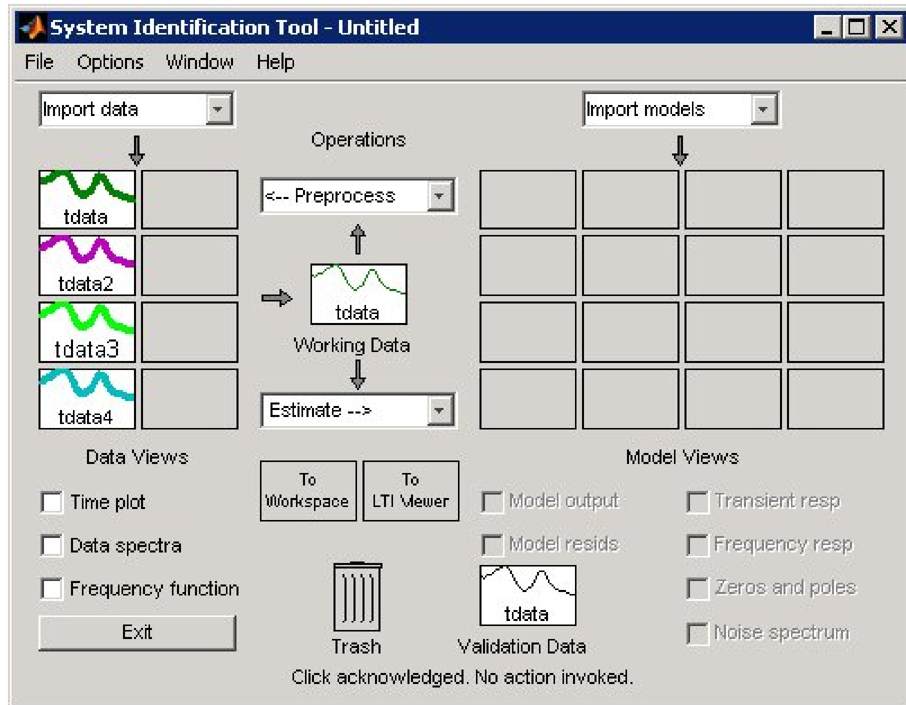
You can only merge data sets that have *all* of the following characteristics:

- 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 Tool GUI.

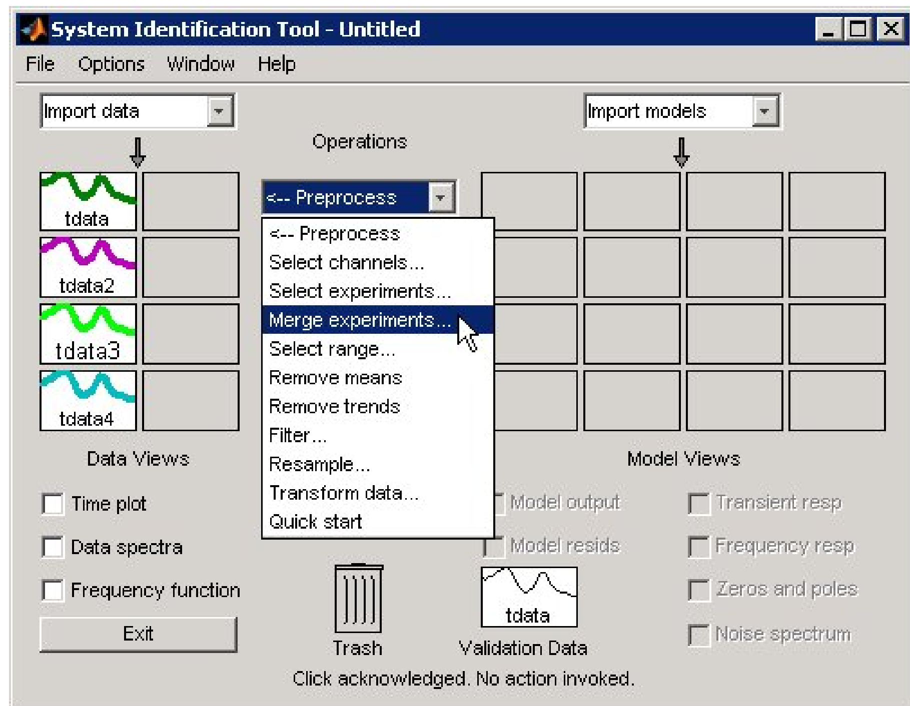
For example, suppose that you want to combine the data sets `tdata`, `tdata2`, `tdata3`, `tdata4` shown in the following figure.



GUI Contains Four Data Sets to Merge

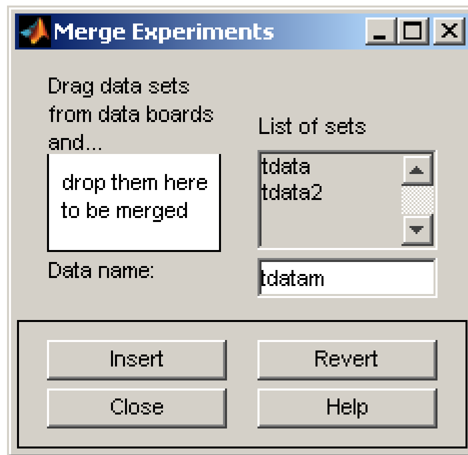
To merge data sets in the GUI:

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 Tool 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**.



tdata and tdata2 to Be Merged

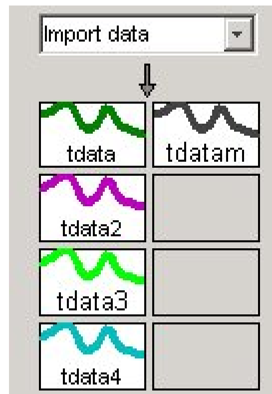
Tip To empty the list, click **Revert**.

---

3 Repeat step 2 for each data set you want to merge. Go to the next step after adding data sets.

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 Tool window.



Data Board Now Contains tdatam with Merged Experiments

6 Click **Close** to close the Merge Experiments dialog box.

Tip To get information about a data set in the System Identification Tool GUI, right-click the data icon to open the Data/model Info dialog box.

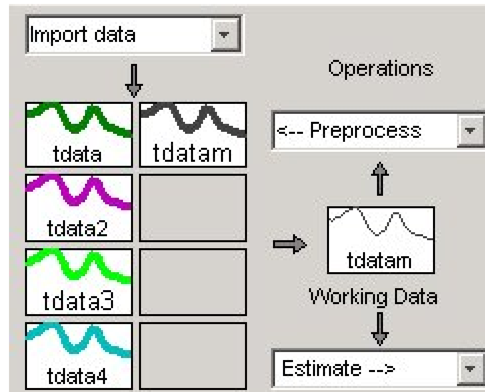
Extracting Specific Experiments from a Multiexperiment Data Set into a New Data Set

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 Tool GUI.

For example, suppose that `tdatam` consists of four experiments.

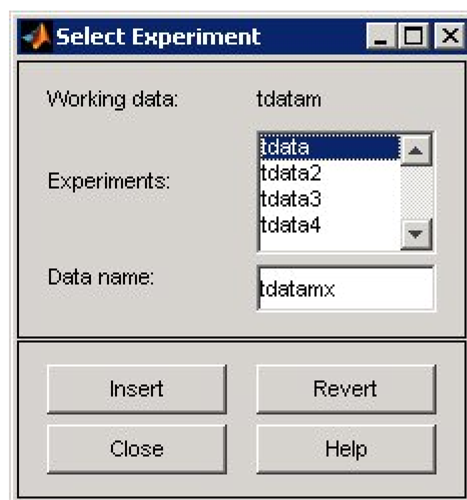
To create a new data set that includes only the first and third experiments in this data set:

1 In the System Identification Tool 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 Tool GUI.

6 Click **Close** to close the Select Experiment dialog box.

Managing Data in the GUI

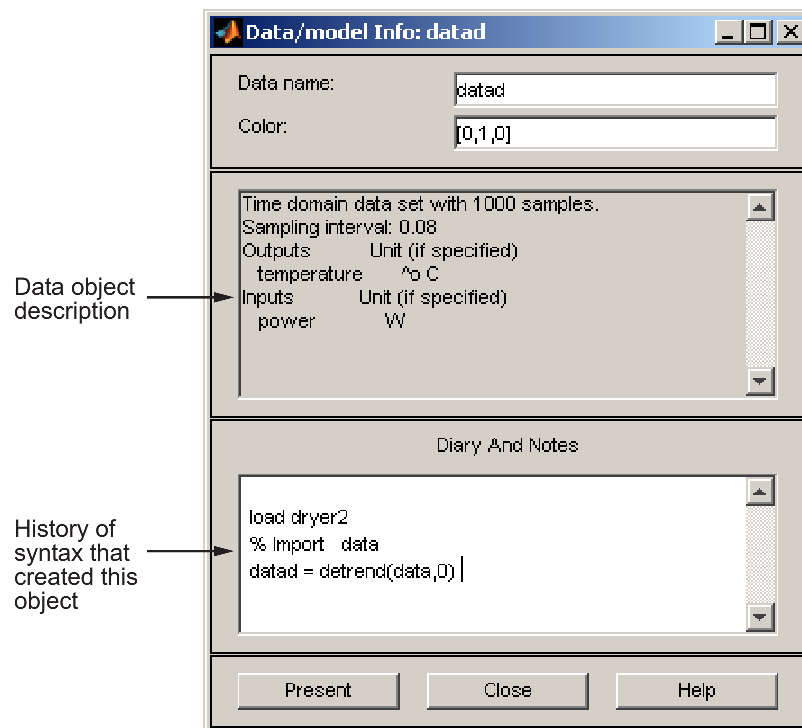
- "Viewing Data Properties" on page 2-46
- "Renaming Data and Changing Display Color" on page 2-47
- "Distinguishing Data Types" on page 2-49
- "Organizing Data Icons" on page 2-49
- "Deleting Data Sets" on page 2-50
- "Exporting Data to the MATLAB Workspace" on page 2-51

Viewing Data Properties

You can get information about each data set in the System Identification Tool GUI 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 Tool GUI. 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 Tool GUI.

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 GUI.

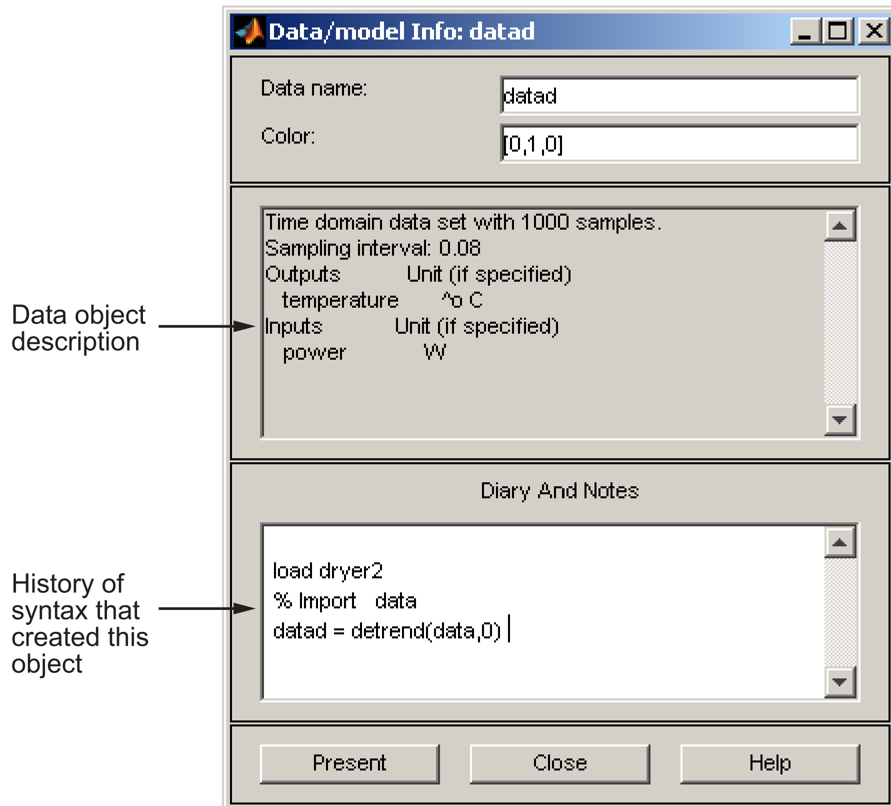
The Data/model Info dialog box also lets you rename the data by entering a new name in the **Data name** field.

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 Tool GUI" on page 11-17.

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 strings represent yellow, red, blue, cyan, green, magenta, and black, respectively.



Information About the Data

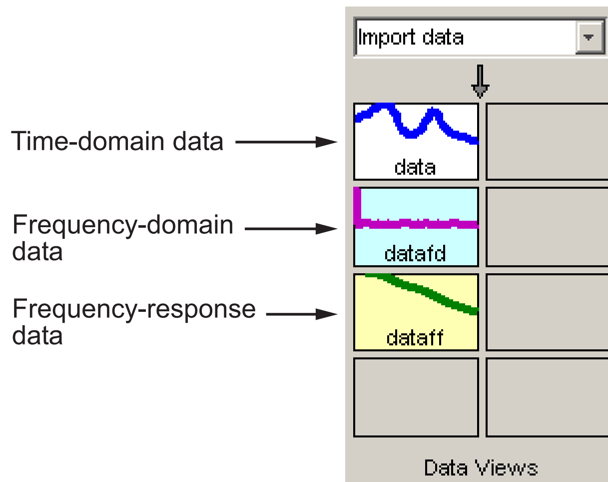
You can enter comments about the origin and state of the data in the **Diary And Notes** area. For example, you might want to include the experiment name, date, and the description of experimental conditions. When you estimate models from this data, these notes are associated with the models.

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.



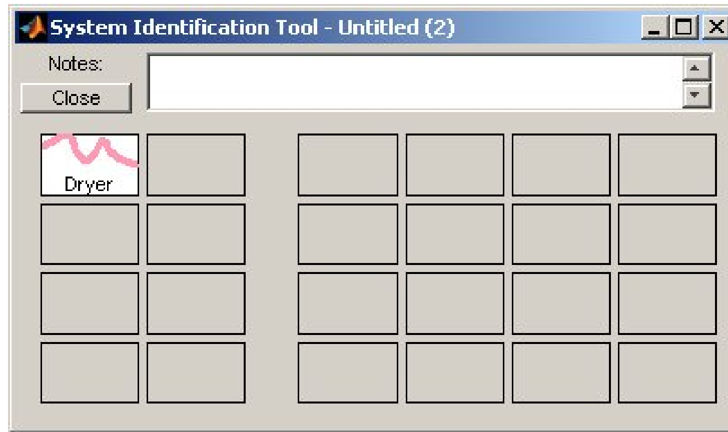
Colors Representing Type of Data

Organizing Data Icons

You can rearrange data icons in the System Identification Tool GUI by dragging and dropping the icons to empty Data Board rectangles in the GUI.

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 Tool GUI. 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 Tool GUI 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 11-6, all additional windows and notes are also saved.

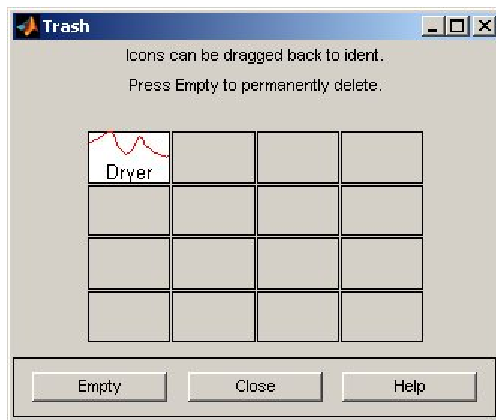
Deleting Data Sets

To delete data sets in the System Identification Tool GUI, drag and drop the corresponding icon into **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 Tool GUI to be **Working Data** or **Validation Data**, as described in "Specifying Estimation and Validation Data" on page 2-35.

To restore a data set from **Trash**, drag its icon from **Trash** to the Data or Model Board in the System Identification Tool 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**.

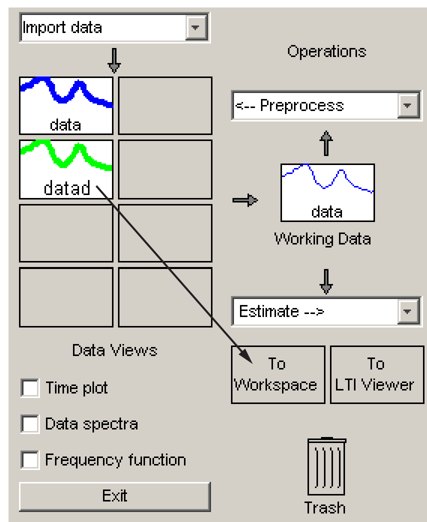
Exiting a session empties the **Trash** automatically.

Exporting Data to the MATLAB Workspace

The data you create in the System Identification Tool GUI 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, drag and drop the corresponding icon to the **To Workspace** rectangle.

When you export data to the MATLAB workspace, the resulting variables have the same name as in the System Identification Tool GUI. 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 2-53

"iddata Properties" on page 2-56

"Creating Multiexperiment Data at the Command Line" on page 2-59

"Select Data Channels, I/O Data and Experiments in iddata Objects" on page 2-61

"Increasing Number of Channels or Data Points of iddata Objects" on page 2-65

"Managing iddata Objects" on page 2-67

iddata Constructor

- "Requirements for Constructing an iddata Object" on page 2-53
- "Constructing an iddata Object for Time-Domain Data" on page 2-54
- "Constructing an iddata Object for Frequency-Domain Data" on page 2-55

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 sampling time, or the time interval, between successive data samples. For uniformly sampled data, Ts is a scalar value equal to the sampling interval of your experiment. The default time unit is seconds, but you can specify any unit string using the TimeUnit property. For more information about iddata time properties, see "Modifying Time and Frequency Vectors" on page 2-67.

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,Ts,[],'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 sampling interval.

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 sampling interval of 0.08 second.

```
load dryer2                   % Load input u2 and output y2.
data = iddata(y2,u2,0.08)     % Create iddata object.
```

MATLAB returns the following output:

```
Time domain data set with 1000 samples.
Sampling interval: 0.08

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 2-70.

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 2-67.

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 dryer2                    % Load input u2 and output y2
data = iddata(y2,u2,0.08);    % Create iddata object
get(data)                      % Get property values of data
```

MATLAB returns the following object properties and values:

```
            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: ''
    ExperimentName: 'Exp1'
             Notes: []
          UserData: []
```

For a complete description of all properties, see the `iddata` reference page or type `idprops iddata` at the prompt.

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 sampling interval 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 `SamplingIntervals`).

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)
```

MATLAB returns the following object properties and values:

```
           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
            Units: 'rad/s'
        Frequency: [501x1 double]
         TimeUnit: ''
   ExperimentName: 'Exp1'
            Notes: []
         UserData: []
```

Creating Multiexperiment Data at the Command Line

- "Why Create Multiexperiment Data Sets?" on page 2-59
- "Limitations on Data Sets" on page 2-60
- "Entering Multiexperiment Data Directly" on page 2-60
- "Merging Data Sets" on page 2-60
- "Adding Experiments to an Existing iddata Object" on page 2-61

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

Create a multiexperiment `iddata` object by merging `iddata` objects, where each contains data from a single experiment or is a multiexperiment data set. For example, you can use the following syntax to merge data:

```
load iddata1     % Loads iddata object z1
load iddata3     % Loads iddata object z3
z = merge(z1,z3) % Merges experiments z1 and z3 into
                 % the iddata object z
```

These commands create an `iddata` object that conatains 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 2-64.

Select Data Channels, I/O Data and Experiments in iddata Objects

- "Subreferencing Input and Output Data" on page 2-61
- "Subreferencing Data Channels" on page 2-63
- "Subreferencing Experiments" on page 2-64

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 name strings.

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 name strings. 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" on page 2-65
- "Horizontal Concatenation" on page 2-65
- "Vertical Concatenation" on page 2-66

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 2-56.

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.

Managing iddata Objects

- "Modifying Time and Frequency Vectors" on page 2-67
- "Naming, Adding, and Removing Data Channels" on page 2-70
- "Subreferencing iddata Objects" on page 2-72
- "Concatenating iddata Objects" on page 2-72

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 2-54.

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`	Sampling time interval.
	• For a single experiment, `Ts` is a scalar value.
	• For multiexperiement data with `Ne` experiments, `Ts` is a `1`-by-`Ne` cell array, and each cell contains the sampling interval of the corresponding experiment.
		To set the sampling interval 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 multiexperiement data with `Ne` experiments, `Ts` is a `1`-by-`Ne` cell array, and each cell contains the sampling interval 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`.

iddata Time-Vector Properties (Continued)

Property	Description	Syntax Example

`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 multiexperiement 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.

`TimeUnit`	Unit of time.	To change the unit of the time vector to `msec`: `data.ti = 'msec'`

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.	To specify `100` frequency values in log space, ranging between `0.1` and `100`, use the following syntax: `data.freq =`

iddata Frequency-Vector Properties (Continued)

Property	Description	Syntax Example
	• For multiexperiement data with `Ne` experiments, `Frequency` is a 1-by-`Ne` cell array, and each cell contains the frequencies of the corresponding experiment.	`logspace(-1,2,100)`

`Units`	Frequency unit must have the following values:
	• If the `TimeUnit` is empty or `s` (seconds), enter `rad/s` or `Hz`
	• If the `TimeUnit` is any string *unit* (other than `s`), enter `rad/`*unit*.
	For multiexperiement data with `Ne` experiments, `Units` is a 1-by-`Ne` cell array, and each cell contains the frequency unit for each experiment.
		If you specified the `TimeUnit` as `msec`, your frequency units must be:
		`data.unit=`
		`'rad/msec'`

Naming, Adding, and Removing Data Channels

- "What Are Input and Output Channels?" on page 2-71
- "Naming Channels" on page 2-71
- "Adding Channels" on page 2-71

• "Modifying Channel Data" on page 2-72

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 11-16.

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 2-65.

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 2-61.

Concatenating iddata Objects

See "Increasing Number of Channels or Data Points of iddata Objects" on page 2-65.

Representing Frequency-Response Data Using idfrd Objects

In this section...

"idfrd Constructor" on page 2-73
"idfrd Properties" on page 2-74
"Select I/O Channels and Data in idfrd Objects" on page 2-76
"Adding Input or Output Channels in idfrd Objects" on page 2-77
"Managing idfrd Objects" on page 2-80
"Operations That Create idfrd Objects" on page 2-81

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 sampling 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 sampling time interval of 0.08 s and get its properties:

```
% Create the idfrd data object
  fr_data = idfrd(response,f,0.08)
% Get property values of data
  get(fr_data)
```

`response` and `f` are variables in the MATLAB Workspace browser, representing the frequency-response data and frequency values, respectively.

MATLAB returns the following object properties and values:

```
ans =

              Name: ''
         Frequency: [100x1 double]
      ResponseData: [1x1x100 double]
      SpectrumData: []
    CovarianceData: []
   NoiseCovariance: []
             Units: 'rad/s'
                Ts: 0.0800
        InputDelay: 0
    EstimationInfo: [1x1 struct]
         InputName: {'u1'}
        OutputName: {'y1'}
         InputUnit: {''}
        OutputUnit: {''}
             Notes: []
          UserData: []
```

For a complete description of all `idfrd` object properties, see the `idfrd` reference page or type `idprops idfrd` at the prompt.

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:

```
% Set the name of the f_data object
  set(fr_data,'name','DC_Converter')
% Get fr_data properties and values
  get(fr_data)
```

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.

If you import `fr_data` into the System Identification Tool GUI, this data has the name `DC_Converter` in the GUI, and not the variable name `fr_data`.

MATLAB returns the following object properties and values:

```
ans =

             Name: 'DC_Converter'
        Frequency: [100x1 double]
     ResponseData: [1x1x100 double]
     SpectrumData: []
   CovarianceData: []
  NoiseCovariance: []
            Units: 'rad/s'
               Ts: 0.0800
       InputDelay: 0
   EstimationInfo: [1x1 struct]
        InputName: {'u1'}
       OutputName: {'y1'}
        InputUnit: {''}
       OutputUnit: {''}
            Notes: []
         UserData: []
```

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" on page 2-78
- "Horizontal Concatenation of idfrd Objects" on page 2-78
- "Vertical Concatenation of idfrd Objects" on page 2-79
- "Concatenating Noise Spectrum Data of idfrd Objects" on page 2-79

About Concatenating idfrd Objects

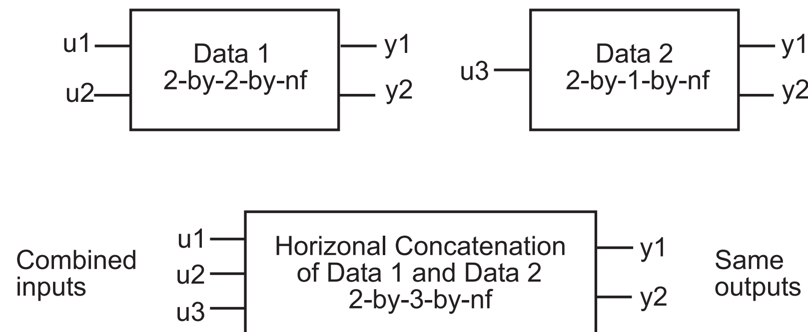
The horizontal and vertical concatenation of `idfrd` objects combine information in the `ResponseData` properties of these objects. `ResponseData` is an `ny`-by-`nu`-by-`nf` array that stores the response of the system, where `ny` is the number of output channels, `nu` is the number of input channels, 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.



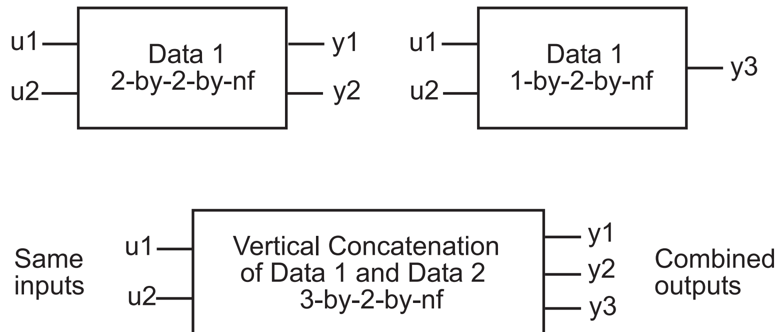
Note Horizontal concatenation of `idfrd` objects requires that they have the same outputs and frequency vectors. If the output channel names are different and their dimensions are the same, the concatenation operation uses the names of output channels in the first `idfrd` object. Input channels must have unique names.

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 uses the names of input channels in the first `idfrd` object you listed. Output channels must have unique names.

Concatenating Noise Spectrum Data of idfrd Objects

When the `idfrd` objects contain the frequency-response data you measured or constructed manually, the concatenation operation combines only the `ResponseData` properties. Because the noise spectrum data does not exist (unless you also entered it manually), `SpectrumData` is empty in both the individual `idfrd` objects and the concatenated `idfrd` object.

However, when the `idfrd` objects are spectral models that you estimated, 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, the 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, 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” on page 2-80
- “Concatenating idfrd Objects” on page 2-81

Subreferencing idfrd Objects

See “Select I/O Channels and Data in idfrd Objects” on page 2-76.

Concatenating idfrd Objects

See "Adding Input or Output Channels in idfrd Objects" on page 2-77.

Operations That Create idfrd Objects

The following operations create `idfrd` objects:

- Constructing `idfrd` objects.
- Transforming `iddata` objects. For more information, see "Transforming Between Frequency-Domain and Frequency-Response Data" on page 2-139.
- Estimating nonparametric models using `etfe`, `spa`, and `spafdr`. For more information, see "Identifying Frequency-Response Models" on page 3-2.
- Converting the Control System Toolbox `frd` object. For more information, see "Using Identified Models for Control Design Applications" on page 9-2.

Analyzing Data Quality

In this section...

"Is Your Data Ready for Modeling?" on page 2-82

"Plotting Data in the GUI Versus at the Command Line" on page 2-83

"How to Plot Data in the GUI" on page 2-83

"How to Plot Data at the Command Line" on page 2-89

"How to Analyze Data Using the advice Command" on page 2-91

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 sampling interval.
- 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`) .

The plots you create using the System Identification Tool GUI 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. The plots you create using the plot commands, such as `plot`, `bode`, and `ffplot`, are displayed in the standard MATLAB Figure window, which provides options for formatting, saving, printing, and exporting plots to a variety of file formats.

See Also

"How to Analyze Data Using the advice Command" on page 2-91

"Ways to Prepare Data for System Identification" on page 2-6

Plotting Data in the GUI Versus at the Command Line

The plots you create using the System Identification Tool GUI 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. For more information, see "How to Plot Data in the GUI" on page 2-83.

The plots you create using the plot commands, such as `plot`, `bode`, and `ffplot`, are displayed in the standard MATLAB Figure window, which provides options for formatting, saving, printing, and exporting plots to a variety of file formats. To learn about plotting at the command line, see "How to Plot Data at the Command Line" on page 2-89. For more information about working with Figure window, see the MATLAB Graphics documentation.

How to Plot Data in the GUI

- "How to Plot Data in the GUI" on page 2-83
- "Manipulating a Time Plot" on page 2-85
- "Manipulating Data Spectra Plot" on page 2-86
- "Manipulating a Frequency Function Plot" on page 2-88

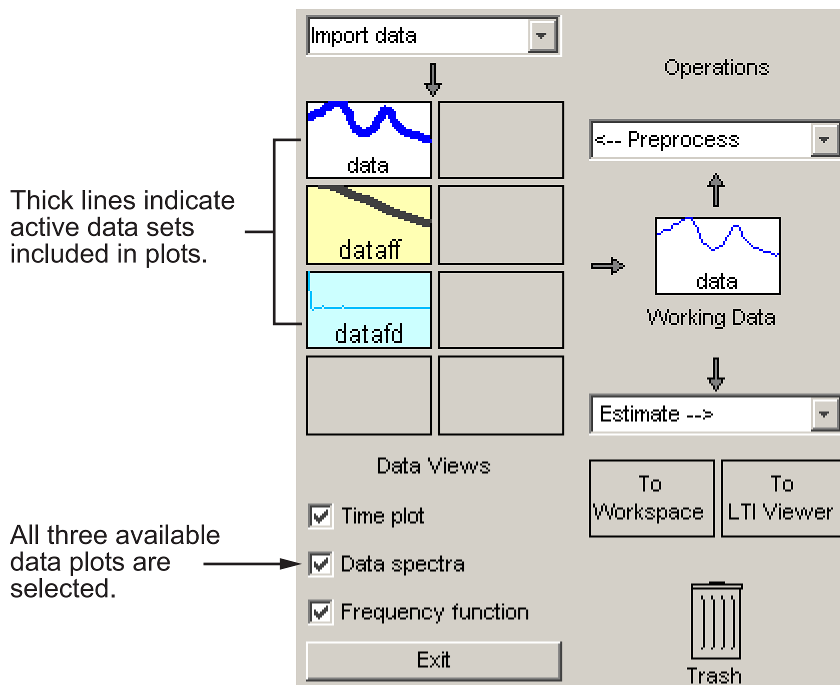
How to Plot Data in the GUI

After importing data into the System Identification Tool GUI, as described in "Importing Data into the GUI" on page 2-17, 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 Tool GUI.

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 Tool GUI. 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 11-16.



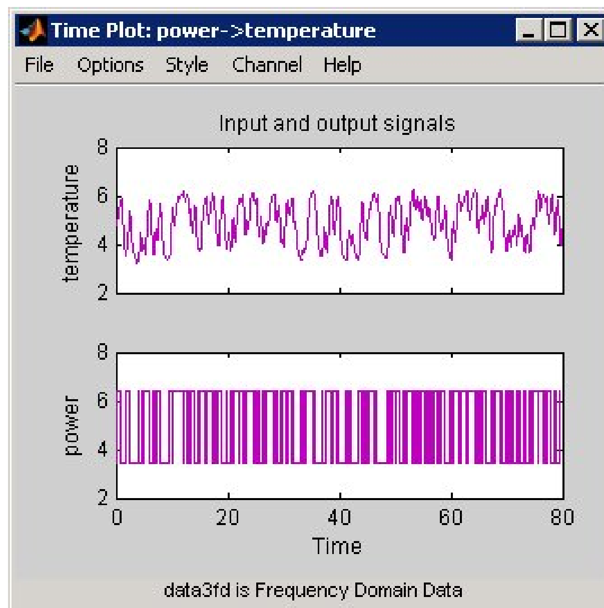
In this example, `data` and `dataff` are active and appear on the three selected plots.

To close a plot, clear the corresponding check box in the System Identification Tool GUI.

Tip To get information about working with a specific plot, select a help topic from the **Help** menu in the plot window.

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 11-13.

Time Plot Options

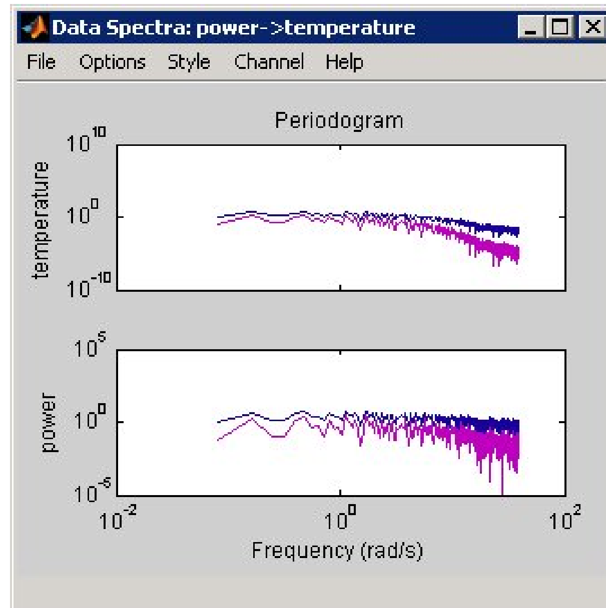
Action	Command
Toggle input display between piece-wise continuous (zero-order hold) and linear interpolation (first-order hold) between samples. **Note** This option only affects the display and not the intersample behavior specified when importing the data.	Select **Style > Staircase input** for zero-order 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 sampling interval. 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 sampling interval.

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

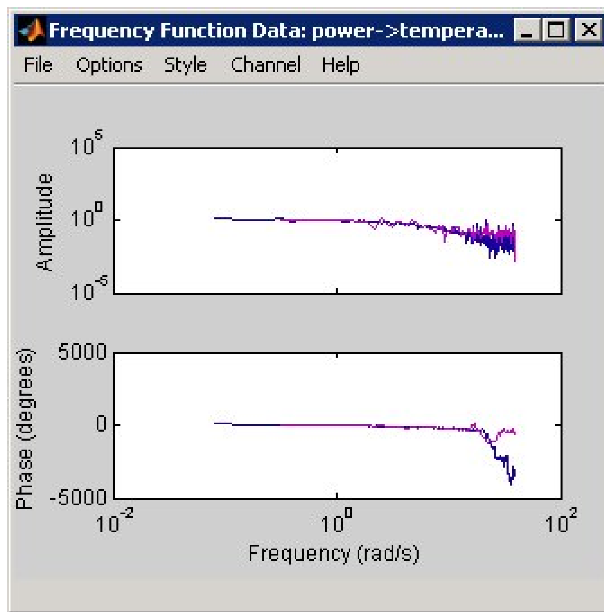
Data Spectra Plot Options

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**.

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`	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)`
`ffplot`	For frequency-response data only. Shows the magnitude and phase of the frequency response on a linear frequency scale (hertz).	To plot `idfrd` data: `ffplot(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	To plot `iddata` or `idfrd` data: `plot(data)`

Commands for Plotting Data (Continued)

Command	Description	Example
	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.	**Note** For `idfrd` data, this command is equivalent to `ffplot(data)`.

All plot commands display the data in the standard MATLAB Figure window. For more information about working with the Figure window, see the MATLAB Graphics documentation.

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 2-61 and "Select I/O Channels and Data in idfrd Objects" on page 2-76. 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.sa,data.u)
```

To plot frequency-domain data, you can use the following syntax:

```
semilogx(data.fr,abs(data.u))
```

In this case, `sa` is an abbreviation of the `iddata` property `SamplingInstants`. Similarly, `fr` is an abbreviation of `Frequency`. `u` is the input signal.

Note The frequencies are linearly spaced on the plot.

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 2-63.

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.

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 2-53.

If you are using the System Identification Tool GUI, 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 GUI to the MATLAB Workspace" on page 11-12.

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.
- What is the estimated input-output delay in the system (dead time)?

See Also

`advice`

`delayest`

`detrend`

`feedback`

`pexcit`

Selecting Subsets of Data

In this section...

"Why Select Subsets of Data?" on page 2-93

"Extract Subsets of Data Using the GUI" on page 2-94

"Extract Subsets of Data at the Command Line" on page 2-96

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 "Importing Data into the GUI" on page 2-17 or "Representing Time- and Frequency-Domain Data Using iddata Objects" on page 2-53.

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 2-116.

Extract Subsets of Data Using the GUI

- "Ways to Select Data in the GUI" on page 2-94
- "Selecting a Range for Time-Domain Data" on page 2-94
- "Selecting a Range of Frequency-Domain Data" on page 2-96

Ways to Select Data in the GUI

You can use System Identification Tool GUI 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 "Specifying Estimation and Validation Data" on page 2-35.

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 Tool GUI.

Note Selecting data does not extract experiments from a data set containing multiple experiments. For more information about multiexperiment data, see "Creating Multiexperiment Data Sets in the GUI" on page 2-39.

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 Tool GUI, as described in "Importing Data into the GUI" on page 2-17.

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 GUI" on page 2-119.

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 2-61.

For more information about subreferencing frequency-response data, see "Select I/O Channels and Data in idfrd Objects" on page 2-76.

Handling Missing Data and Outliers

In this section...

"Handling Missing Data" on page 2-97

"Handling Outliers" on page 2-98

"Example – Extracting and Modeling Specific Data Segments" on page 2-99

"See Also" on page 2-100

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` ("Not-a-Number"). `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 2-53.

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 "Residual Analysis" on page 8-26. 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(Model,Data)
  % 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 "Example – Extracting and Modeling Specific Data Segments" on page 2-99). For more information about selecting and extracting data segments, see "Selecting Subsets of Data" on page 2-93.

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 2-97. 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 2-116.

Note The estimation algorithm handles outliers automatically 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. The value `LimitError` (see `Algorithm Properties`) quantitatively distinguishes between moderate and large outliers.

Example – Extracting and Modeling Specific Data Segments

The following example shows how to create a multiexperiment, time-domain data set by merging only the accurate-data segments and ignoring the rest.

Assume that the data has poor or no measurements for some sample ranges (for example 341–499). 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, as follows:

```
% Plot the data in a MATLAB Figure window
plot(data)

% Create multiexperiment data set
% by merging data segments
  datam = merge(data(1:340),...
                data(500:897),...
                data(1001:1200),...
                data(1550:2000));

% Model the multiexperiment data set
% using "experiments" 1, 2, and 4
m =pem(getexp(datam,[1,2,4]))

% Validate the model by comparing its output to
% the output data of experiment 3
compare(getexp(datam,3),m)
```

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.

Handling Offsets and Trends in Data

In this section...

"When to Detrend Data" on page 2-101

"Alternatives for Detrending Data in GUI or at the Command-Line" on page 2-102

"Next Steps After Detrending" on page 2-103

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 "Identifying Low-Order Transfer Functions (Process Models)" on page 3-20.

- Nonlinear black-box models, such as nonlinear ARX or Hammerstein-Wiener models. For more information, see Chapter 4, “Nonlinear Black-Box 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 “Estimating Nonlinear Grey-Box Models” on page 5-15.

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.

Examples

“How to Detrend Data Using the GUI” on page 2-104

“How to Detrend Data at the Command Line” on page 2-105

Alternatives for Detrending Data in GUI or at the Command-Line

You can detrend data using the System Identification Tool GUI and at the command line using the `detrend` command.

Both the GUI 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 GUI):

- 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.

To learn how to detrend data, see:

- "How to Detrend Data Using the GUI" on page 2-104
- "How to Detrend Data at the Command Line" on page 2-105

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 Chapter 3, "Linear Model Identification".

How to Detrend Data Using the GUI

Before you can perform this task, you must have regularly-sampled, steady-state time-domain data imported into the System Identification Tool GUI. See "Importing Time-Domain Data into the GUI" on page 2-18). For transient data, see "How to Detrend Data at the Command Line" on page 2-105.

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 Tool, 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**.

More About

"Handling Offsets and Trends in Data" on page 2-101

How to Detrend Data at the Command Line

In this section...

"Detrending Steady-State Data" on page 2-105

"Detrending Transient Data" on page 2-105

"See Also" on page 2-106

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 2-53.

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.

More About

"Handling Offsets and Trends in Data" on page 2-101

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 2-53.

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 2-89.

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.

More About

"Handling Offsets and Trends in Data" on page 2-101

See Also

`detrend`

`TrendInfo`

Resampling Data

In this section...

"What Is Resampling?" on page 2-107

"Resampling Data Without Aliasing Effects" on page 2-108

"See Also" on page 2-112

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 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 Tool GUI (zero-order or first-order hold). For more information about the data properties you specify before importing the data, see "Importing Data into the GUI" on page 2-17.

You can resample data using the System Identification Tool GUI or the `resample` command. You can only resample time-domain data at uniform time intervals.

Examples

"Resampling Data Using the GUI" on page 2-113

"Resampling Data at the Command Line" on page 2-114

Resampling Data Without Aliasing Effects

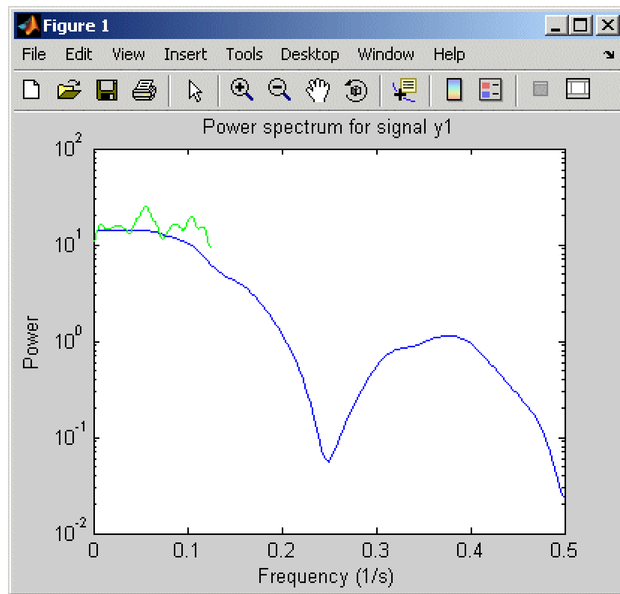
Typically, you decimate a signal to remove the high-frequency contributions that result from noise from the total energy. Ideally, you want to remove the energy contribution due to noise and preserve the energy density of the signal.

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 3-8.

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. 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.

The following example illustrates how `resample` avoids folding effects:

```
% Construct fourth-order MA-process
m0 = idpoly(1,[ ],[1 1 1 1]);
% Generate error signal
e = idinput(2000,'rgs');
e = iddata([],e,'Ts',1);
% Simulate the output using error signal
y = sim(m0,e);
% Estimate signal spectrum
g1 = spa(y);
% Estimate spectrum of modified signal including
% every fourth sample of the original signal.
% This command automatically sets Ts to 4.
g2 = spa(y(1:4:2000));
% Plot frequency response to view folding effects
ffplot(g1,g2)
% Estimate spectrum after prefiltering that does not
% introduce folding effects
g3 = spa(resample(y,1,4));
figure
ffplot(g1,g3)
```

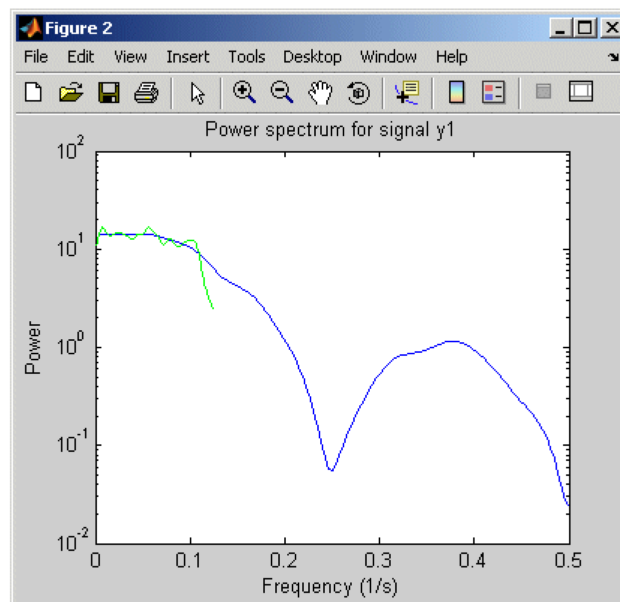


Folding Effects with Manual Decimation

Use `resample` to decimate the signal before estimating the spectrum and plot the frequency response, as follows:

```
g3 = spa(resample(y,1,4));
figure
ffplot(g1,g3)
```

The following figure 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.



No Folding Effects When Using resample

Examples

“Resampling Data Using the GUI” on page 2-113

“Resampling Data at the Command Line” on page 2-114

See Also

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 Using the GUI

Use the System Identification Tool GUI to resample time-domain data. To specify additional options, such as the prefilter order, see "Resampling Data at the Command Line" on page 2-114.

The System Identification Tool GUI 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 Tool GUI, as described in "Importing Data into the GUI" on page 2-17.

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 sampling interval:

   - For decimation (fewer samples), enter a factor greater than `1` to increase the sampling interval by this factor.

   - For interpolation (more samples), enter a factor less than `1` to decrease the sampling interval 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.

More About

"Resampling Data" on page 2-107

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 sampling interval: the new sampling interval 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 sampling interval 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 2-56.

More About

"Resampling Data" on page 2-107

Filtering Data

In this section...

"Supported Filters" on page 2-116
"Choosing to Prefilter Your Data" on page 2-116
"See Also" on page 2-117

Supported Filters

You can filter the input and output signals through a linear filter before estimating a model in the System Identification Tool GUI 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 Tool GUI is a fifth-order (passband) Butterworth filter. If you need to specify a custom filter, use the `idfilt` command.

Examples

"How to Filter Data Using the GUI" on page 2-118

"How to Filter Data at the Command Line" on page 2-122

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 2-101.

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 model of the system.

To get a reliable noise model, avoid prefiltering the data. Instead, set the `Focus` property of the estimation algorithm to `Simulation`. For more information about the `Focus` property, see the `Algorithm Properties` reference page.

Note When you prefilter during model estimation, the filtered data is used to only model the input-to-output dynamics. However, the disturbance model is calculated from the unfiltered data.

Examples

"How to Filter Data Using the GUI" on page 2-118

"How to Filter Data at the Command Line" on page 2-122

See Also

To learn how to filter data during linear model estimation instead, you can set the `Focus` property of the estimation algorithm to `Filter` and specify the filter characteristics. For more information about model properties, see the `Algorithm Properties` reference page.

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.

How to Filter Data Using the GUI

In this section...

"Filtering Time-Domain Data in the GUI" on page 2-118

"Filtering Frequency-Domain or Frequency-Response Data in the GUI" on page 2-119

Filtering Time-Domain Data in the GUI

The System Identification Tool GUI 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 Tool GUI, as described in "Importing Data into the GUI" on page 2-17.

2. Drag the data set you want 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 Tool GUI, 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.

More About

"Filtering Data" on page 2-116

Filtering Frequency-Domain or Frequency-Response Data in the GUI

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 Tool GUI, as described in "Importing Data into the GUI" on page 2-17.

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.

More About

"Filtering Data" on page 2-116

How to Filter Data at the Command Line

In this section...

"Simple Passband Filter" on page 2-122
"Defining a Custom Filter" on page 2-123
"Causal and Noncausal Filters" on page 2-124

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.

More About

"Filtering Data" on page 2-116

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 `idmodel` 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`.

More About

"Filtering Data" on page 2-116

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 2-139). The result is an `iddata` object.

More About

"Filtering Data" on page 2-116

Generating Data Using Simulation

In this section...

"Commands for Generating Data Using Simulation" on page 2-126
"Example – Creating Periodic Input Data" on page 2-127
"Example – Generating Output Data Using Simulation" on page 2-128
"Simulating Data Using Other MathWorks Products" on page 2-129

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 Model Structures" on page 1-16.

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	`u = iddata([],...` `    idinput(400,'rbs',[0 0.3]));`

Commands for Generating Data (Continued)

Command	Description	Example
	sinusoid, and returns a matrix of input values.	
`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.

Example – Creating Periodic Input Data

This example shows how to create a periodic random Gaussian input signal using `idinput`.

1 Create a periodic input for one input and consisting of five periods, where each period is 300 samples.

```
per_u = idinput([300 1 5]);
```

2 Create an `iddata` object using the periodic input and leaving the output empty.

```
u = iddata([],per_u,'Period',.300);
```

3 View the data characteristics in time- and frequency-domain.

```
Plot data in time-domain.
plot(u);
Plot the spectrum.
bode(spa(u));
```

4 (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(m0,u,'noise')
```

Example – Generating 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.

1 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;
m = 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.

2 Construct a pseudorandom binary input data.

```
u = idinput([200,3],'prbs');
```

3 Simulate model output with noise using the input data.

```
y = sim(m,u,'noise');
```

4 Represent the simulation data as an `iddata` object.

```
iodata = iddata(y,u,m.Ts);
```

5 (Optional) Estimate a model of the same order as `m` using `iodata`.

```
na = m.na;
nb = m.nb;
nc= m.nc;
nk= m.nk
me = armax(iodata,[na,nb,nc,nk]);
```

Use `bode(m,me)` and `compare(iodata,me)` to check how closely `me` and `m` match.

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 10-5.

Transforming Between Time- and Frequency-Domain Data

In this section...

"Transforming Data Domain in the GUI" on page 2-130
"Transforming Data Domain at the Command Line" on page 2-137

Transforming Data Domain in the GUI

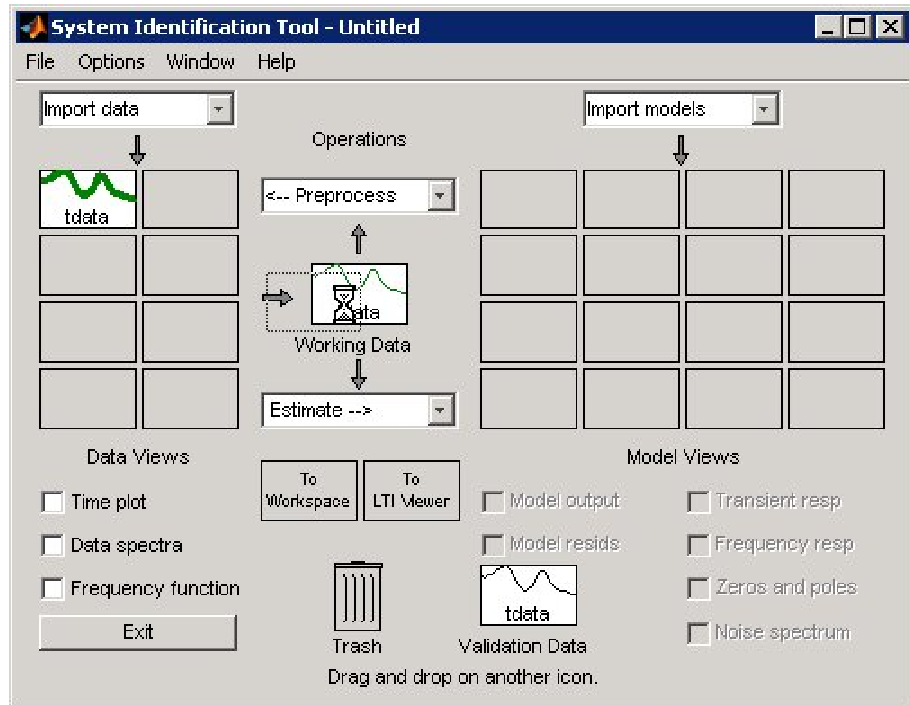
- "Transforming Time-Domain Data" on page 2-130
- "Transforming Frequency-Domain Data" on page 2-134
- "Transforming Frequency-Response Data" on page 2-135
- "See Also" on page 2-137

Transforming Time-Domain Data

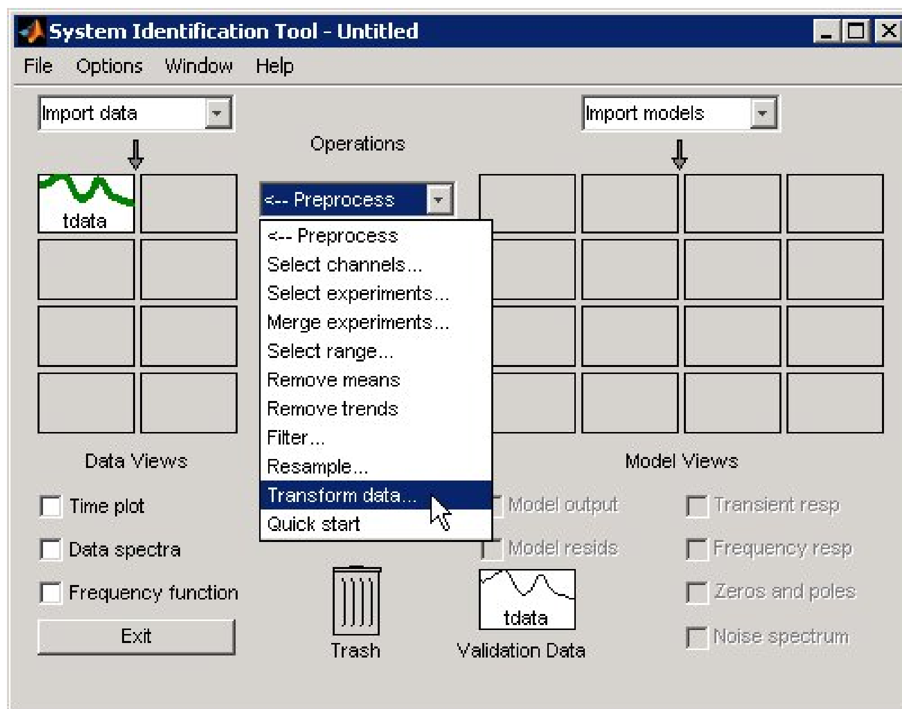
In the System Identification Tool GUI, time-domain data has an icon with a white background. You can transform time-domain data to frequency-domain or frequency-response data. The frequency values of the resulting frequency vector range from `0` to the Nyquist frequency $f_S = \pi / T_S$, where T_s is the sampling interval.

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 Tool GUI, drag the icon of the data you want to transform to the **Working Data** rectangle, as shown in the following figure.

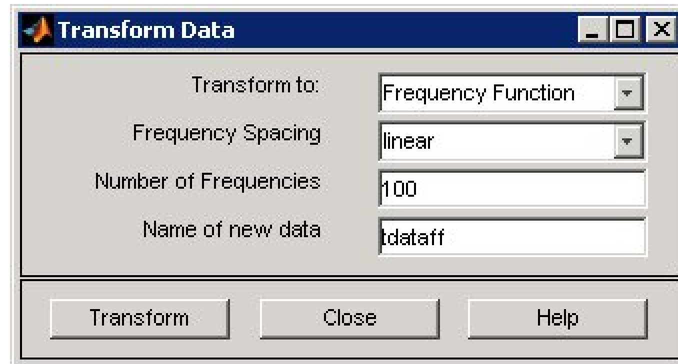


2 In the **Operations** area, select **<–Preprocess > Transform data** in the drop-down menu to open the Transform Data dialog box.



3 In the **Transform to** drop-down 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 Tool GUI.

8 Click **Close** to close the Transform Data dialog box.

Transforming Frequency-Domain Data

In the System Identification Tool GUI, 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 Tool GUI, 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 Tool GUI.

8. Click **Close** to close the Transform Data dialog box.

Transforming Frequency-Response Data

In the System Identification Tool GUI, 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 2-139.

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 Tool GUI 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 Tool GUI, 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 Tool GUI.
8. Click **Close** to close the Transform Data dialog box.

See Also

For a description of time-domain, frequency-domain, and frequency-response data, see “Representing Data in MATLAB Workspace” on page 2-9.

To learn how to transform data at the command line instead of the GUI, see “Transforming Data Domain at the Command Line” on page 2-137.

Transforming Data Domain at the Command Line

- “Supported Data Transformations” on page 2-137
- “Transforming Between Time and Frequency Domain” on page 2-138
- “Transforming Between Frequency-Domain and Frequency-Response Data” on page 2-139
- “See Also” on page 2-140

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 method used by the software is listed in the cell at their intersection.

Original Data Format	To Time Domain (iddata object)	To Frequency Domain (iddata object)	To Frequency Function (idfrd object)
Time Domain (`iddata` object)	No.	Yes, using `fft`.	Yes, using `etfe`, `spa`, or `spafdr`.
Frequency Domain (`iddata` object)	Yes, using `ifft`.	No.	Yes, using `etfe`, `spa`, or `spafdr`.
Frequency Function (`idfrd` object)	No.	Yes. Calculation creates frequency-domain `iddata` object	Yes. Calculates a frequency function with different

Original Data Format	To Time Domain (iddata object)	To Frequency Domain (iddata object)	To Frequency Function (idfrd object)
		that has the same ratio between output and input as the original idfrd object.	resolution (number and spacing of frequencies) using spafdr.

Transforming Between Time and Frequency Domain

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-domainiddata object f_data to time-domain iddata object t_data, use: t_data = ifft(f_data)

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 advanced 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

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 "Identifying Frequency-Response Models" on page 3-2.

Manipulating Complex-Valued Data

In this section...

"Supported Operations for Complex Data" on page 2-142
"Processing Complex iddata Signals at the Command Line" on page 2-142

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`, `covf`, `ivar`, `iv4`, `oe`, `pem`, `spa`, 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, but only show the absolute values of the signals. 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.

Command	Description
`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.
`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
```

3

Linear Model Identification

- "Identifying Frequency-Response Models" on page 3-2
- "Identifying Impulse-Response Models" on page 3-11
- "Identifying Low-Order Transfer Functions (Process Models)" on page 3-20
- "Identifying Input-Output Polynomial Models" on page 3-39
- "Identifying State-Space Models" on page 3-73
- "Refining Linear Parametric Models" on page 3-104
- "Extracting Numerical Model Data" on page 3-109
- "Transforming Between Discrete-Time and Continuous-Time Representations" on page 3-112
- "Transforming Between Linear Model Representations" on page 3-117
- "Subreferencing Models" on page 3-119
- "Concatenating Models" on page 3-124
- "Merging Models" on page 3-128

Identifying Frequency-Response Models

In this section...

"What Is a Frequency-Response Model?" on page 3-2

"Data Supported by Frequency-Response Models" on page 3-3

"How to Estimate Frequency-Response Models in the GUI" on page 3-3

"How to Estimate Frequency-Response Models at the Command Line" on page 3-5

"Selecting the Method for Computing Spectral Models" on page 3-5

"Controlling Frequency Resolution of Spectral Models " on page 3-6

"Spectrum Normalization" on page 3-8

What Is a Frequency-Response Model?

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.

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.

For a discrete-time system sampled with a time interval T, the frequency-response model $G(z)$ relates the Z-transforms of the input $U(z)$ and output $Y(z)$:

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

In other words, the frequency-response function, $G(e^{iwT})$, is the Laplace transform of the impulse response that is evaluated on the imaginary axis. The frequency-response function is the transfer function $G(z)$ evaluated on the unit circle.

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 Chapter 6, "Time Series Identification".
- Single- or multiple-output data.

How to Estimate Frequency-Response Models in the GUI

You must have already imported your data into the GUI and performed any necessary preprocessing operations. For more information, see Chapter 2, "Data Import and Processing".

To estimate frequency-response models in the System Identification Tool GUI:

1. In the System Identification Tool GUI, 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 3-5.

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 3-6. 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 Tool GUI.

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 Tool GUI. For more information about working with this plot, see "Frequency Response Plots" on page 8-44.

9. To view the estimated disturbance spectrum, select the **Noise spectrum** check box in the System Identification Tool GUI. For more information about working with this plot, see "Noise Spectrum Plots" on page 8-53.

10. After estimating the model, see Chapter 8, "Model Analysis", to validate the model.

To export the model to the MATLAB workspace, drag it to the **To Workspace** rectangle in the System Identification Tool GUI. You can retrieve the responses from the resulting `idfrd` model object using the `bode` or `nyquist` command.

How to 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.

After estimating the model, see Chapter 8, “Model Analysis” to validate the model.

Selecting the Method for Computing Spectral Models

This section describes how to select the method for computing spectral models in the estimation procedures “How to Estimate Frequency-Response Models in the GUI” on page 3-3 and “How to Estimate Frequency-Response Models at the Command Line” on page 3-5.

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.

Controlling Frequency Resolution of Spectral Models

- “What Is Frequency Resolution?” on page 3-7
- “Frequency Resolution for etfe and spa” on page 3-7
- “Frequency Resolution for spafdr” on page 3-7
- “etfe Frequency Resolution for Periodic Input” on page 3-8

This section supports the estimation procedures “How to Estimate Frequency-Response Models in the GUI” on page 3-3 and “How to Estimate Frequency-Response Models at the Command Line” on page 3-5.

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, $0\text{-}F_s/2$ 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 $\frac{2\pi k}{T}\left(\frac{\text{k}}{\text{Period}}\right)$ where $k = 1, 2, \ldots, \text{Period}$.

For periodic data, the frequency resolution is ignored.

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 sampling interval T:

$$\Phi_y(\omega) = T \sum_{k=-M}^{M} R_y(kT) e^{-iwT} W_M(k)$$

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 the MATLAB Command Window:

```
load iddata1
% Create time-series iddata object
y = z1(:,1,[]);
% Define sample interval from the data
T = y.Ts;
% Estimate frequency response
sp = spa(y);
% Remove spurious dimensions
phiy = squeeze(sp.spec);
% Compute average energy from the estimated
```

```
% energy spectrum, where S1 is scaled by T
S1 = sum(phiy)/length(phiy)/T
% Compute average energy of the signal
S2 = sum(y.y.^2)/size(y,1)
```

In this code, phiy contains $\Phi_y(\omega)$ between $\omega = 0$ and $\omega = \pi/T$ with the frequency step given as follows:

$$\left(\frac{\pi}{T \cdot \text{length(phiy)}}\right)$$

MATLAB computes the following values for S1 and S2:

```
S1 =

   19.2076
S2 =

   19.4646
```

Thus, the average energy of the signal approximately equals the average energy in the estimated spectrum.

Identifying Impulse-Response Models

In this section...

"What Is Time-Domain Correlation Analysis?" on page 3-11

"Data Supported by Correlation Analysis" on page 3-12

"How to Estimate Impulse and Step Response Models Using the GUI" on page 3-12

"How to Estimate Impulse and Step Response Models at the Command Line" on page 3-14

"How to Compute Response Values" on page 3-15

"How to Identify Delay Using Transient-Response Plots" on page 3-16

"Correlation Analysis Algorithm" on page 3-18

What Is Time-Domain Correlation Analysis?

Time-domain correlation analysis is a nonparametric estimate of transient response of dynamic systems, which computes a finite impulse response (FIR) model from the data. Correlation analysis assumes a linear system and does not require a specific model structure.

There are two types of transient response for a dynamic model:

- Impulse response

  Impulse response is the output signal that results when the input is an impulse and has the following definition for a discrete model:

$$\begin{aligned} u(t) &= 0 \quad & t > 0 \\ u(t) &= 1 \quad & t = 0 \end{aligned}$$

- Step response

Step response is the output signal that results from a step input, defined as follows:

$$u(t) = 0 \qquad t < 0$$
$$u(t) = 1 \qquad t \geq 0$$

The response to an input *u(t)* is equal to the convolution of the impulse response, as follows:

$$y(t) = \int_0^t h(t-z) \cdot u(z)dz$$

Data Supported by Correlation Analysis

You can estimate correlation analysis models from data with the following characteristics:

- Real or complex time-domain `iddata` object. To learn about estimating time-series models, see Chapter 6, "Time Series Identification".
- Frequency-domain `iddata` or `idfrd` object with the sampling interval $T \neq 0$.
- Single- or multiple-output data.

How to Estimate Impulse and Step Response Models Using the GUI

Before you can perform this task, you must have

- Regularly sampled data imported into the System Identification Tool GUI. See "Importing Time-Domain Data into the GUI" on page 2-18. For supported data formats, see "Data Supported by Correlation Analysis" on page 3-12.
- 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 Tool GUI using time-domain correlation analysis:

1. In the System Identification Tool GUI, 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 Tool GUI.

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 Tool GUI.
- View the transient response plot by selecting the **Transient resp** check box in the System Identification Tool GUI. For more information about

working with this plot and selecting to view impulse- versus step-response, see "Impulse and Step Response Plots" on page 8-35.

How to Estimate Impulse and Step Response Models at the Command Line

Before you can perform this task, you must have

- Regularly sampled data. See "Representing Time- and Frequency-Domain Data Using iddata Objects" on page 2-53. For supported data formats, see "Data Supported by Correlation Analysis" on page 3-12.
- Performed any required data preprocessing operations. Ro improve the accuracy of your model, you should detrend your data. See "Ways to Prepare Data for System Identification" on page 2-6.

The following tables summarize the commands for computing impulse- and step-response models. Both `impulse` and `step` produce the same FIR model, but generate different plots. The resulting models are stored as `idarx` model objects and contain impulse-response coefficients in the model parameter `B`. For detailed information about these commands, see the corresponding reference page.

Note `cra` is an alternative method for computing impulse response from time-domain data only.

Commands for Impulse and Step Response

Command	Description	Example
`impulse`	Estimates a high-order, noncausal FIR model using correlation analysis.	To estimate the model `m` and plot the impulse response, use the following syntax: `m=impulse(data,Time,'pw',N)` where `data` is a single- or multiple-output time-domain `iddata` object, and `Time` is a scalar value representing the time interval

Commands for Impulse and Step Response (Continued)

Command	Description	Example
		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*. `'pw'` and `N` is an option property-value pair that specifies the order `N` of the prewhitening filter `'pw'`.
`step`	Estimates a high-order, noncausal FIR model using correlation analysis.	To estimate the model `m` and plot the step response, use the following syntax: `step(data,Time)` where `data` is a single- or multiple-output time-domain `iddata` object, and `Time` is the time span.

Next Steps

- Perform model analysis. See "Validating Models After Estimation" on page 8-3.

How to 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 applying either `impulse` or `step` commands on the data, as described in "How to Estimate Impulse and Step Response Models at the Command Line" on page 3-14.

2. Apply the following syntax on the resulting model:

```
% To compute impulse-response data
[y,t,ysd] = impulse(model)
```

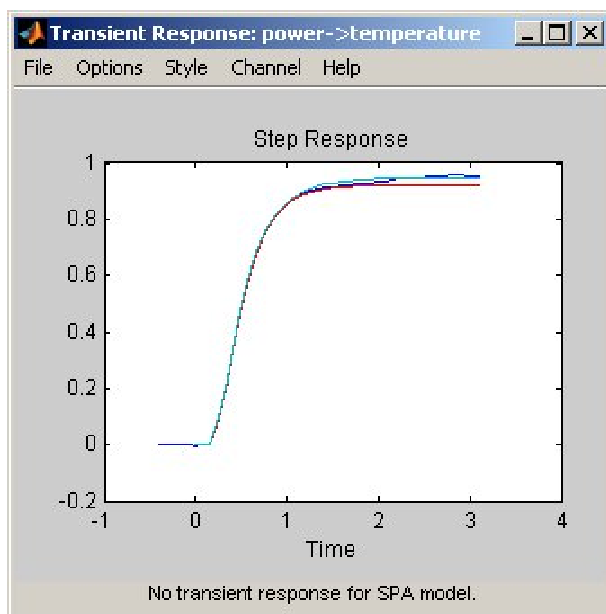
```
% To compute step-response data
[y,t,ysd] = step(model)
```

where y is the response data, t is the time vector, and ysd is the standard deviations of the response.

How to 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 Tool GUI. To view the transient response plot, select the **Transient resp** check box in the System Identification Tool GUI. 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

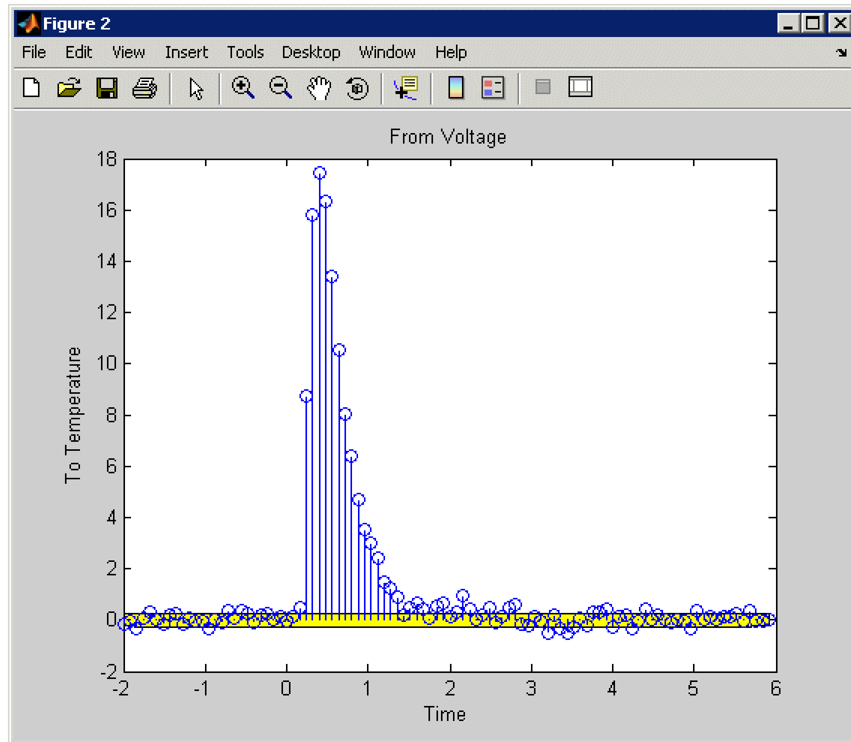
At the command line. You can use the impulse command to plot the impulse response. The time delay is equal to the first positive peak in the

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 sample data
load dry2
% Split data into estimation and
% validation data sets
ze = dry2(1:500);
zr = dry2(501:1000);
impulse(ze,'sd',1,'fill')
```

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.



Correlation Analysis Algorithm

To better understand the algorithm underlying correlation analysis, consider the following description of a dynamic system:

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

where $u(t)$ and $y(t)$ are the input and output signals, respectively. $v(t)$ is the additive noise term. $G(q)$ is the transfer function of the system. The $G(q)u(t)$ notation represents the following operation:

$$G(q)u(t) = \sum_{k=1}^{\infty} g(k)u(t-k)$$

q is the *shift operator*, defined by the following equation:

$$G(q) = \sum_{k=1}^{\infty} g(k) q^{-k} \qquad q^{-1} u(t) = u(t-1)$$

For impulse response, the algorithm estimates impulse response coefficients g for both the single- and multiple-output data. The impulse response is estimated as a high-order, noncausal FIR model:

$$\begin{aligned} y(t) = {} & g(-m)u(t+m) + \ldots + g(-1)u(t+1) + g(0)u(t) \\ & + g(1)u(t-1) + \ldots + g(n)u(t-n) \end{aligned}$$

The estimation algorithm prefilters the data such that the input is as white as possible. It then computes the correlations from the prefiltered data to obtain the FIR coefficients.

g is also estimated for negative lags, which takes into account any noncausal effects from input to output. Noncausal effects can result from feedback. The coefficients are computed using the least-squares method.

For a multiple-input or multiple-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*th element of the impulse response matrix describes the behavior of the *i*th output after an impulse in the *j*th input.

Identifying Low-Order Transfer Functions (Process Models)

In this section...

"What Is a Process Model?" on page 3-20
"Data Supported by Process Models" on page 3-21
"How to Estimate Process Models Using the GUI" on page 3-21
"How to Estimate Process Models at the Command Line" on page 3-27
"Process Model Structure Specification" on page 3-33
"Estimating Multiple-Input Process Models" on page 3-34
"Disturbance Model Structure for Process Models" on page 3-35
"Assigning Estimation Weightings" on page 3-36
"Specifying Initial States for Iterative Estimation Algorithms" on page 3-37

What Is a Process Model?

The structure of a *continuous-time process model* is a simple 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 primary advantages of these models are that they provide delay estimation, and the model coefficients have a physical interpretation.

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).

Note Continuous-time process models let you estimate the input delay.

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:

$$G(s) = \frac{K}{1 + sT_{p1}} e^{-sT_d}$$

To learn more about estimating continuous-time process models in the GUI, see "Tutorial – Identifying Low-Order Transfer Functions (Process Models) Using the GUI" in *System Identification Toolbox Getting Started Guide*.

Data Supported by Process Models

You can estimate low-order (up to third order), continuous-time transfer functions from data with the following characteristics:

- Regularly sampled time- or frequency-domain `iddata` or `idfrd` data object
- Real data, or complex data in the time domain only
- Single-output data

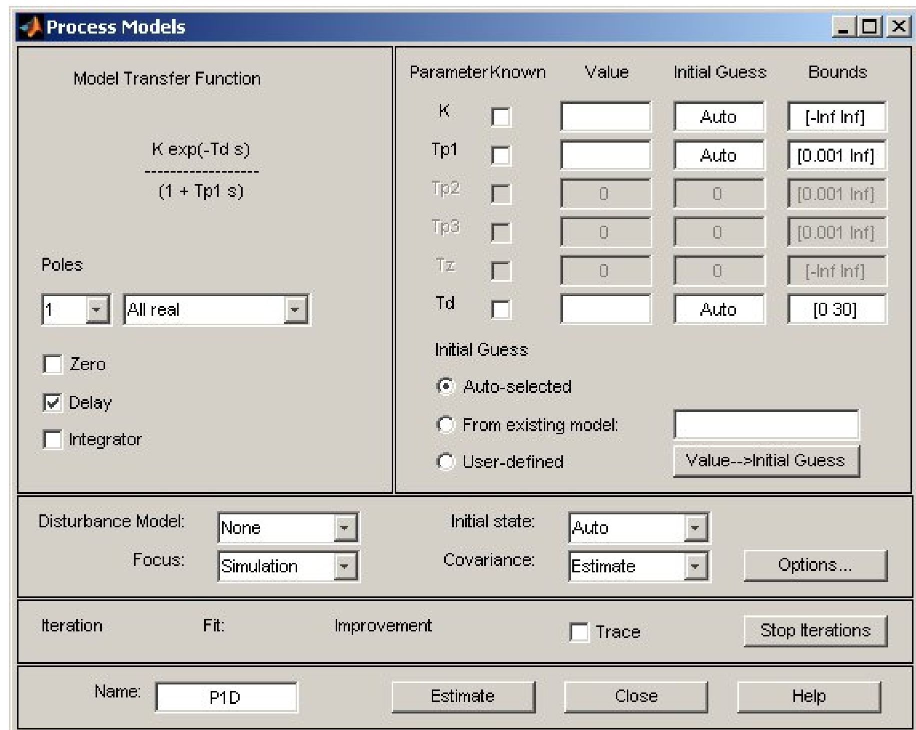
You must import your data into the MATLAB workspace, as described in Chapter 2, "Data Import and Processing".

How to Estimate Process Models Using the GUI

Before you can perform this task, you must have

- Imported data into the System Identification Tool GUI. See “Importing Time-Domain Data into the GUI” on page 2-18. For supported data formats, see “Data Supported by Process Models” on page 3-21.
- 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 Tool GUI, select **Estimate > Process models** to open the Process Models dialog box.



2. If your model contains multiple inputs, select the input channel in the **Input** list. This list only appears when you have multiple inputs. For more information, see “Estimating Multiple-Input Process Models” on page 3-34.

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 3-33.

---

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.

Parameter	Known	Value	Initial Guess	Bounds
K	☐		Auto	[-Inf Inf]
Tw	☐		Auto	[0.001 Inf]
Zeta	☐		Auto	[0.001 Inf]
Tp3	☐	0	0	[0.001 Inf]
Tz	☐	0	0	[-Inf Inf]
Td	☐		Auto	[0 30]

Initial Guess

- (•) Auto-selected
- ( ) From existing model:
- ( ) User-defined — Value-->Initial Guess

5 (Optional) If you approximately know a parameter value, enter this value in the **Initial Guess** column of the Parameter table. 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 also 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.

For example, the following figure shows that the delay value `Td` is fixed at `2` s and is not estimated.

Parameter	Known	Value	Initial Guess	Bounds
K	☐		Auto	[-Inf Inf]
Tw	☐		Auto	[0.001 Inf]
Zeta	☐		Auto	[0.001 Inf]
Tp3	☐	0	0	[0.001 Inf]
Tz	☐	0	0	[-Inf Inf]
Td	☑	2	2	[0 30]

Initial Guess

- ○ Auto-selected
- ○ From existing model:
- ◉ User-defined — Value-->Initial Guess

6 In the **Disturbance Model** list, select one of the available options. For more information about each option, see "Disturbance Model Structure for Process Models" on page 3-35.

7 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 3-36.

8 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 3-37.

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** 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 in the MATLAB Command Window, select the **Trace** check box. During estimation, the following information is displayed for each iteration:

   - Loss function — Equals the determinant of the estimated covariance matrix of the input noise.
   - Parameter values — Values of the model structure coefficients you specified.
   - Search direction — Change in parameter values from the previous iteration.
   - Fit improvements — Shows the actual versus expected improvements in the fit.

12 Click **Estimate** to add this model to the Model Board in the System Identification Tool GUI.

13 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.

Next Steps

- Validate the model by selecting the appropriate check box in the **Model Views** area of the System Identification Tool GUI. For more information about validating models, see "Validating Models After Estimation" on page 8-3.
- Refine the model by clicking the **Value —> Initial Guess** button to assign current parameter values as initial guesses for the next search, edit the **Model 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 Tool GUI.

How to Estimate Process Models at the Command Line

- "Prerequisites" on page 3-27
- "Using pem to Estimate Process Models" on page 3-27
- "Example – Estimating Process Models with Free Parameters at the Command Line" on page 3-29
- "Example – Estimating Process Models with Fixed Parameters at the Command Line" on page 3-30

Prerequisites

Before you can perform this task, you must have

- Regularly sampled data as an `iddata` object. See "Representing Time- and Frequency-Domain Data Using iddata Objects" on page 2-53. For supported data formats, see "Data Supported by Process Models" on page 3-21.
- 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.

Using pem to Estimate Process Models

You can estimate process models using the iterative estimation method `pem` 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 = pem(data,mod_struc,'Property1',Value1,...,
                       'PropertyN',ValueN)
```

To capture offsets that are essential to describe the dynamics of interest, such as when the model contains integration behavior, set the `InputLevel` property set to “estimate”.

`data` is the estimation data and `mod_struc` is a string that represents the process model structure, as described in “Process Model Structure Specification” on page 3-33.

Tip You do not need to construct the model object using `idproc` before estimation unless you want to specify initial parameter guesses or fixed parameter values, as described in “Example – Estimating Process Models with Fixed Parameters at the Command Line” on page 3-30.

The property-value pairs specify any model properties that configure the estimation algorithm and the initial conditions. For more information about accessing and setting model properties, see “Model Properties” on page 1-17.

Note You can specify all property-value pairs in `pem` as a simple, comma-separated list without worrying about the hierarchy of these properties in the `idproc` model object.

For more information about validating a process model, see “Validating Models After Estimation” on page 8-3.

You can use `pem` to refine parameter estimates of an existing process model, as described in “Refining Linear Parametric Models” on page 3-104.

For detailed information about `pem` and `idproc`, see the corresponding reference page.

Example – Estimating Process Models with Free Parameters at the Command Line

This example demonstrates how to estimate the parameters of a first-order process model:

$$G(s) = \frac{K}{1 + sT_{p1}} e^{-sT_d}$$

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.

Use the following commands to estimate a model `m` from sample data:

```
% Load sample data
load co2data
% Sampling interval is 0.5 min (known)
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 and a delay
m = pem(ze,'P1D')
```

MATLAB computes the following output:

```
Process model with 2 inputs:
y = G_1(s)u_1 + G_2(s)u_2
where
           K
G_1(s) = ---------- * exp(-Td*s)
          1+Tp1*s

with   K = -3.2168
     Tp1 = 23.033
      Td = 10.101

           K
G_2(s) = ---------- * exp(-Td*s)
          1+Tp1*s

with   K = 9.9877
     Tp1 = 2.0314
      Td = 4.8368
```

Use dot notation to get the value of any model parameter. For example, to get the `Value` field in the `K` structure, type the following command:

```
m.K.value
```

Example – Estimating Process Models with Fixed Parameters at the Command Line

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 sample data
load co2data
% Sampling interval is 0.5 min (known)
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')
```

MATLAB computes the following output:

```
Process model with transfer function
           K
G(s) = ---------- * exp(-Td*s)
        1+Tp1*s

with   K = NaN
     Tp1 = NaN
      Td = NaN

This model was not estimated from data.
```

The model parameters `K`, `Tp1`, and `Td` are assigned `NaN` values, which means that the parameters have not yet been estimated from the data.

All process-model parameters are structures with the following fields:

- `status` field specifies whether to estimate the parameter, or keep the initial value fixed (do not estimate), or set the value to zero. This field can have the values `'estimate'`, `'fixed'`, or `'zero'`. For more information, see "Specifying Initial States for Iterative Estimation Algorithms" on page 3-37.
- `min` specifies the minimum bound on the parameter.
- `max` specifies the maximum bound on the parameter.

- `value` specifies the numerical value of the parameter, if known.

To set the value of `K` to `12` and keep it fixed, use the following commands:

```
mod.K.value=12;
mod.K.status='fixed';
```

Note `mod` is defined for one input. This model is automatically adjusted to have a duplicate for each input.

To estimate `Tp1` and `Td` only, use the following command:

```
mod_proc=pem(ze,mod)
```

MATLAB computes the following result:

```
Process model with 2 inputs:
y = G_1(s)u_1 + G_2(s)u_2
where
           K
G_1(s) = ---------- * exp(-Td*s)
          1+Tp1*s

with   K = 12
     Tp1 = 7.0998e+007
      Td = 15


           K
G_2(s) = ---------- * exp(-Td*s)
          1+Tp1*s

with   K = 12
     Tp1 = 3.6962
      Td = 3.817
```

In this case, the value of `K` is fixed at `12`, but `Tp1` and `Td` are estimated.

If you prefer to specify parameter constraints directly in the estimator syntax, the following table provides examples of `pem` commands.

Action	Example
Fix the value of `K` to `12`.	`m=pem(ze,'p1d','k','fix','k',12)`
Initialize `K` for the iterative search without fixing this value.	`m=pem(ze,'p1d','k',12)`
Constrain the value of `K` between `3` and `4`.	`m=pem(ze,'p1d','k',...` `  {'min',3},'k',{'max',4})`

Process Model Structure Specification

This topic describes how to specify the model structure in the estimation procedures "How to Estimate Process Models Using the GUI" on page 3-21 and "How to Estimate Process Models at the Command Line" on page 3-27.

In the System Identification Tool GUI. 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 a string argument in the estimation command `pem`:

- `pem(data,'P1D')` to estimate the following structure:

$$G(s) = \frac{K}{1 + sT_{p1}} e^{-sT_d}$$

- `pem(data,'P2ZU')` to estimate the following structure:

$$G(s) = \frac{K_p (1 + sT_z)}{1 + 2s\zeta T_w + s^2 T_w^2}$$

- `pem(data,'POID')` to estimate the following structure:

$$G(s) = \frac{K_p}{s} e^{-sT_d}$$

- `pem(data,'P3Z')` to estimate the following structure:

$$G(s) = \frac{K_p (1 + sT_z)}{(1 + sT_{p1})(1 + sT_{p2})(1 + sT_{p3})}$$

For more information about estimating models, see "How to Estimate Process Models at the Command Line" on page 3-27.

Estimating Multiple-Input Process Models

If your model contains multiple inputs, you can specify whether to estimate the same transfer function for all inputs, or a different transfer function for each input. The information in this section supports the estimation procedures "How to Estimate Process Models Using the GUI" on page 3-21 and "How to Estimate Process Models at the Command Line" on page 3-27.

In the System Identification Tool GUI. To fit a data set with multiple inputs in the Process Models dialog box, configure the process model settings for one input at a time. When you finish configuring the model and the

estimation settings for one input, select a different input in the **Input Number** list.

If you want the same transfer function to apply to all inputs, select the **Same structure for all channels** 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 acronym strings in the estimation command `pem`. 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 = pem(data,m)
```

To apply the same structure to all inputs, define a single structure in `idproc`.

Disturbance Model Structure for Process Models

This section describes how to specify a noise model in the estimation procedures "How to Estimate Process Models Using the GUI" on page 3-21 and "How to Estimate Process Models at the Command Line" on page 3-27.

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 GUI. 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 as an argument in the estimation command `pem`. For example, use this command to estimate a first-order transfer function and a first-order noise model:

```
pem(data,'P1D','DisturbanceModel','ARMA1')
```

Tip You can type `'dis'` instead of `'DisturbanceModel'`.

For a complete list of values for the `DisturbanceModel` model property, see the `idproc` reference page.

Assigning Estimation Weightings

You can specify how the estimation algorithm weighs the fit at various frequencies. This information supports the estimation procedures "How to Estimate Process Models Using the GUI" on page 3-21 and "How to Estimate Process Models at the Command Line" on page 3-27.

In the System Identification Tool GUI. 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 8-76.
- `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 2-122 or "Defining a Custom Filter" on page 2-123. This prefiltering applies only for estimating the dynamics from input to output. The disturbance model is determined from the estimation data.

At the command line. Specify the focus as an argument in the estimation command `pem` using the same options as in the GUI. For example, use this command to optimize the fit for simulation and estimate a disturbance model:

```
pem(data,'P1D','dist','arma2','Focus','Simulation')
```

Specifying Initial States for Iterative Estimation Algorithms

Because the process models are dynamic, you need initial states that capture past input properties. Thus, you must specify how the iterative algorithm treats initial states. This information supports the estimation procedures "How to Estimate Process Models Using the GUI" on page 3-21 and "How to Estimate Process Models at the Command Line" on page 3-27.

In the System Identification Tool GUI. Set **Initial state** 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 states and the `InputLevel` model property that represents the input offset level. 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 initial states have negligible effect on the prediction errors, the initial states are set to zero to optimize algorithm performance.

At the command line. Specify the initial states as an argument in the estimation command `pem` using the same options as in the GUI. For example, use this command to estimate a first-order transfer function and set the initial states to zero:

```
m=pem(data,'P1D','InitialState','zero')
```

For a complete list of values for the `InitialState` model property, see the `idproc` reference page.

Identifying Input-Output Polynomial Models

In this section...

"What Are Black-Box Polynomial Models?" on page 3-39

"Data Supported by Polynomial Models" on page 3-46

"Preliminary Step – Estimating Model Orders and Input Delays" on page 3-48

"How to Estimate Polynomial Models in the GUI" on page 3-56

"How to Estimate Polynomial Models at the Command Line" on page 3-59

"Estimating Multiple-Input and Multiple-Output ARX Orders" on page 3-64

"Assigning Estimation Weightings" on page 3-65

"Specifying Initial States for Iterative Estimation Algorithms" on page 3-66

"Polynomial Model Estimation Algorithms" on page 3-66

"Example – Estimating Models Using armax" on page 3-67

What Are Black-Box Polynomial Models?

- "Polynomial Model Structure" on page 3-40
- "Understanding the Time-Shift Operator q" on page 3-41
- "Definition of a Discrete-Time Polynomial Model" on page 3-41
- "Definition of a Continuous-Time Polynomial Model" on page 3-44
- "Definition of Multiple-Output ARX Models" on page 3-44

Polynomial Model Structure

You can estimate the following types of linear polynomial model structures:

$$A(q)y(t) = \sum_{i=1}^{nu} \frac{B_i(q)}{F_i(q)} u_i\left(t - nk_i\right) + \frac{C(q)}{D(q)} e(t)$$

The polynomials A, B_i, C, D, and F_i contain the time-shift operator q. u_i is the ith input, nu is the total number of inputs, and nk_i is the ith input delay that characterizes the delay response time. 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 3-41.

Note This form is completely equivalent to the Z-transform form: q corresponds to z.

To estimate polynomial models, 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 sampling interval. 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*.

Definition of a Discrete-Time Polynomial Model

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. For more

information about the time-shift operator, see "Understanding the Time-Shift Operator q" on page 3-41.

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	Discrete-Time Form	Noise Model
ARX	$A(q)y(t) = \sum_{i=1}^{nu} B_i(q)u_i(t - nk_i) + e(t)$	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
ARMAX	$A(q)y(t) = \sum_{i=1}^{nu} B_i(q)u_i(t - nk_i) + C(q)e(t)$	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*.

Model Structure	Discrete-Time Form	Noise Model
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 input-output polynomial models for single output systems are represented by the `idpoly` object. Multi-output polynomial ARX models are represented by the `idarx` object.

The System Identification Tool GUI supports only the polynomial models listed in the table. However, you can use `pem` to estimate all five polynomial or any subset of polynomials in the general equation. For more information about working with pem, see "Using pem to Estimate Polynomial Models" on page 3-61.

Definition of a Continuous-Time Polynomial Model

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:

$$A(s) = s^4 + a_1 s^3 + a_2 s^2 + a_3$$
$$B(s) = b_1 s + b_2$$

The simplest way to estimate continuous-time polynomial models of arbitrary structure is to first estimate a discrete-time model of arbitrary order and then use `d2c` to convert this model to continuous time. For more information, see “Transforming Between Discrete-Time and Continuous-Time Representations” on page 3-112.

You can also 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.

Definition of Multiple-Output ARX Models

You can use a multiple-output ARX model to model a multiple-output dynamic system. The ARX model structure is represented by the `idarx` object, and given by the following equation:

$$A(q)y(t) = B(q)u\left(t - nk\right) + e(t)$$

For a system with *nu* inputs and *ny* outputs, $A(q)$ is an *ny*-by-*ny* matrix. $A(q)$ can be represented as a polynomial in the shift operator q^{-1}:

$$A(q) = I_{ny} + A_1 q^{-1} + \ldots + A_{na} q^{-na}$$

For more information about the time-shift operator, see “Understanding the Time-Shift Operator q” on page 3-41.

$A(q)$ can also be represented as a matrix:

$$A(q) = \begin{pmatrix} a_{11}(q) & a_{12}(q) & \ldots & a_{1ny}(q) \\ a_{21}(q) & a_{22}(q) & \ldots & a_{2ny}(q) \\ \ldots & \ldots & \ldots & \ldots \\ a_{ny1}(q) & a_{ny2}(q) & \ldots & a_{nyny(q)} \end{pmatrix}$$

where the matrix element a_{kj} is a polynomial in the shift operator q^{-1}:

$$a_{kj}(q) = \delta_{kj} + a_{kj}^{1} q^{-1} + \ldots + a_{kj}^{na_{kj}} q^{-na_{kj}}$$

δ_{kj} represents the Kronecker delta, which equals 1 for $k=j$ and equals 0 for $k \neq j$. This polynomial describes how the old values of the *j*th output are affected by the *k*th output. The *i*th row of *A(q)* represents the contribution of the past output values for predict the current value of the *i*th output.

$B(q)$ is an *ny*-by-*ny* matrix. $B(q)$ can be represented as a polynomial in the shift operator q^{-1}:

$$B(q) = B_0 + B_1 q^{-1} + \ldots + B_{nb} q^{-nb}$$

$B(q)$ can also be represented as a matrix:

$$B(q) = \begin{pmatrix} b_{11}(q) & b_{12}(q) & \dots & b_{1nu}(q) \\ b_{21}(q) & b_{22}(q) & \dots & b_{2nu}(q) \\ \dots & \dots & \dots & \dots \\ b_{ny1}(q) & b_{ny2}(q) & \dots & b_{nynu(q)} \end{pmatrix}$$

where the matrix element b_{kj} is a polynomial in the shift operator q^{-1}:

$$b_{kj}(q) = a_{kj}^{1} q^{-nb_{kj}} + \dots + a_{kj}^{nk_{kj}} q^{-nb_{kj} - nb_{kj} + 1}$$

nk_{kj} is the delay from the jth input to the kth output. *B(q)* represents the contributions of inputs to predicting all output values.

Data Supported by Polynomial Models

- "Types of Supported Data" on page 3-46
- "Designating Data for Estimating Continuous-Time Models" on page 3-47
- "Designating Data for Estimating Discrete-Time Models" on page 3-47

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 black-box polynomial models for time-series data, see Chapter 6, "Time Series Identification".

- Real data or complex data in any domain.

- Single-output and multiple-output (ARX structure only).

You must import your data into the MATLAB workspace, as described in Chapter 2, "Data Import and Processing".

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 the ARX and 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 sampling interval to 0. You can set the sampling interval when you import data into the GUI 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 sampling interval when you import data into the GUI or set the `Ts` property of the data object at the command line.

Preliminary Step – Estimating Model Orders and Input Delays

- "Why Estimate Model Orders and Delays?" on page 3-48
- "Estimating Orders and Delays in the GUI" on page 3-48
- "Estimating Model Orders at the Command Line" on page 3-52
- "Estimating Delays at the Command Line" on page 3-54
- "Selecting Model Orders from the Best ARX Structure" on page 3-54

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 "Residual Analysis" on page 8-26.

Estimating Orders and Delays in the GUI

The following procedure assumes that you have already imported your data into the GUI and performed any necessary preprocessing operations. For more information, see Chapter 2, "Data Import and Processing".

To estimate model orders and input delays in the System Identification Tool GUI:

1. In the System Identification Tool GUI, select **Estimate > Linear parametric models** to open the Linear Parametric Models dialog box.

   The ARX model is already selected by default in the **Structure** list.

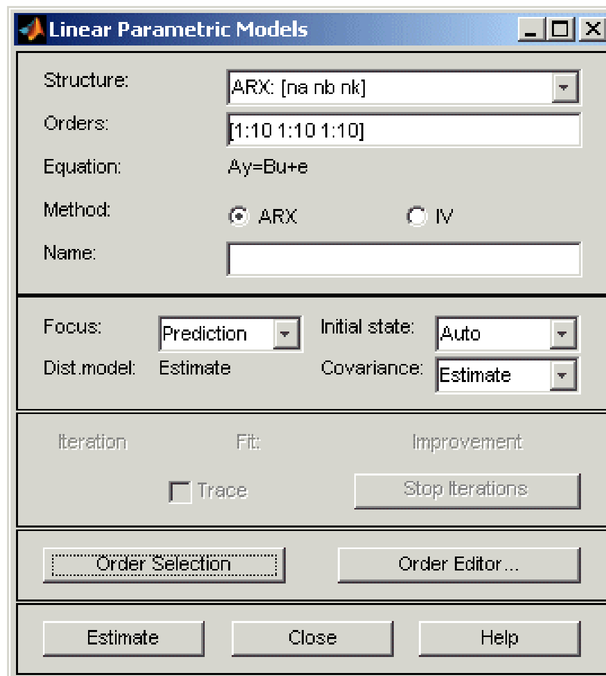
   Note For time-series models, select the AR model structure.

   ---

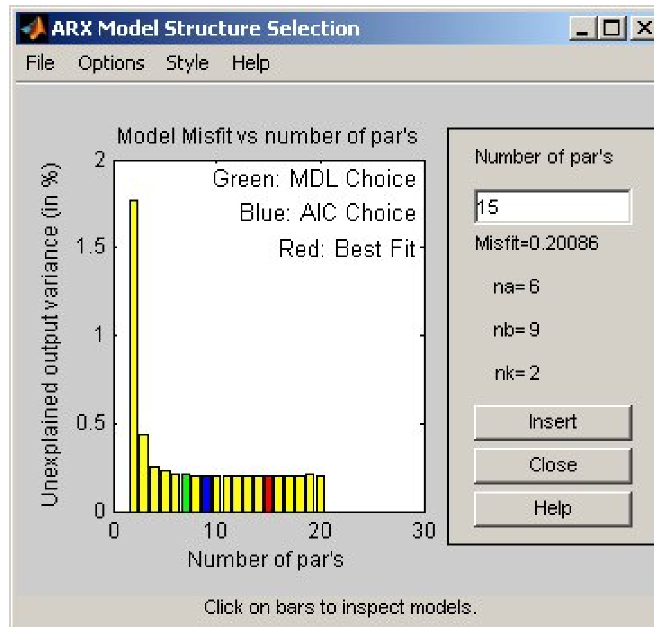
2. Edit the **Orders** field to specify a range of poles, zeros, and delays. For example, enter the following values for *na*, *nb*, and *nk*:

   ```
   [1:10 1:10 1:10]
   ```

   Tip As a shortcut for entering `1:10` for each required model order, click **Order Selection**.


Linear Parametric Models
Structure:
ARX: [na nb nk]
Orders:
[1:10 1:10 1:10]
Equation:
Ay=Bu+e
Method:
ARX
IV
Name:
Focus:
Prediction
Initial state:
Auto
Dist.model:
Estimate
Covariance:
Estimate
Iteration
Fit:
Improvement
Trace
Stop Iterations
Order Selection
Order Editor ...
Estimate
Close
Help

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 3-54.

This action adds a new model to the Model Board in the System Identification Tool GUI. 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.

After estimating model orders and delays, use these values as initial guesses for estimating other model structures, as described in "How to Estimate Polynomial Models in the GUI" on page 3-56.

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 `struc`, `arxstruc`, and `selstruc` commands for each output. In this case, 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 2-61 and "Select I/O Channels and Data in idfrd Objects" on page 2-76.

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 3-54.

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 pem to Estimate Polynomial Models" on page 3-61.

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 sampling interval of the data.

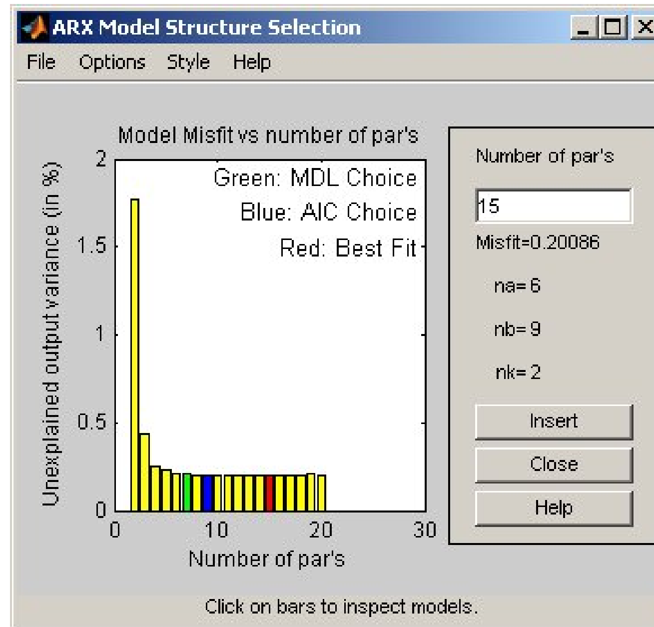
You can also use the ARX Model Structure Selection window to estimate input delays and model order together, as described in "Estimating Model Orders at the Command Line" on page 3-52.

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 Tool GUI, see "Estimating Orders and Delays in the GUI" on page 3-48. To open this plot at the command line, see "Estimating Model Orders at the Command Line" on page 3-52.

The following figure shows a sample plot in the ARX Model Structure Selection window.



The horizontal axis in the ARX Model Structure Selection window is the total number of ARX parameters:

$$\text{Number of parameters} = n_a + n_b$$

The vertical axis, called **Unexplained output variance (in %)**, is the ARX model prediction error for a specific number of parameters. The *prediction error* is the sum of the squares of the differences between the validation data output and the model output. In other words, **Unexplained output variance (in %)** is the portion of the output not explained by the model.

Three rectangles are highlighted on the plot—green, blue, and red. Each color indicates a type of best-fit criterion, as follows:

- Red minimizes the sum of the squares of the difference between the validation data output and the model output. This option is considered the overall best fit.
- Green minimizes Rissanen MDL criterion.

- Blue 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 “Akaike’s Criteria for Model Validation” on page 8-68.

How to Estimate Polynomial Models in the GUI

Prerequisites

Before you can perform this task, you must have:

- Imported data into the System Identification Tool GUI. See “Importing Time-Domain Data into the GUI” on page 2-18. For supported data formats, see “Data Supported by Polynomial Models” on page 3-46.
- 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 Black-Box Polynomial Models?” on page 3-39 For more information about how to estimate model orders and delays, see “Estimating Orders and Delays in the GUI” on page 3-48. For multiple-output ARX model, you must specify order matrices in the MATLAB workspace, as described in “Estimating Multiple-Input and Multiple-Output ARX Orders” on page 3-64.

1. In the System Identification Tool GUI, select **Estimate > Linear parametric models** to open the Linear Parametric Models dialog box.

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 Linear Parametric Models dialog box to correspond with this model structure. For information about each model structure, see "What Are Black-Box Polynomial Models?" on page 3-39.

   Note For time-series data, only AR and ARMA models are available. For more information about estimating time-series models, see Chapter 6, "Time Series Identification".

   ---

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 ARX models.** Enter the model orders, as described in "Estimating Multiple-Input and Multiple-Output ARX Orders" on page 3-64.

   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 3-66.

5 In the **Name** field, edit the name of the model or keep the default.

6 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 3-65.

7 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 3-37.

Tip If you get an inaccurate fit, try setting a specific method for handling initial states rather than choosing it automatically.

---

8 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 ARX models might reduce computation time.

9 (ARMAX, OE, and BJ models only) To view the estimation progress in the MATLAB Command Window, select the **Trace** check box. During estimation, the following information is displayed for each iteration:

   - Loss function — Equals the determinant of the estimated covariance matrix of the input noise.
   - Parameter values — Values of the model structure coefficients you specified.
   - Search direction — Change in parameter values from the previous iteration.
   - Fit improvements — Shows the actual versus expected improvements in the fit.

10 Click **Estimate** to add this model to the Model Board in the System Identification Tool GUI.

11 (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.

Next Steps

- Validate the model by selecting the appropriate check box in the **Model Views** area of the System Identification Tool GUI. For more information about validating models, see "Validating Models After Estimation" on page 8-3.
- Export the model to the MATLAB workspace for further analysis by dragging it to the **To Workspace** rectangle in the System Identification Tool GUI.

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 "Using Linear Model for Nonlinear ARX Estimation" on page 4-28, and "Using Linear Model for Hammerstein-Wiener Estimation" on page 4-64.

How to Estimate Polynomial Models at the Command Line

- "Using arx and iv4 to Estimate ARX Models" on page 3-60
- "Using pem to Estimate Polynomial Models" on page 3-61

Prerequisites

Before you can perform this task, you must have

- Regularly sampled data as an `iddata` object. See "Representing Time- and Frequency-Domain Data Using iddata Objects" on page 2-53. For supported data formats, see "Data Supported by Polynomial Models" on page 3-46.

- 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 Black-Box Polynomial Models?" on page 3-39 For more information about how to estimate model orders and delays, see "Estimating Model Orders at the Command Line" on page 3-52 and "Estimating Delays at the Command Line" on page 3-54. For multiple-output ARX model, you must specify order matrices in the MATLAB workspace, as described in "Estimating Multiple-Input and Multiple-Output ARX Orders" on page 3-64.

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 3-66.

If you are estimating a multiple-output ARX model, you must specify order matrices in the MATLAB workspace before estimation, as described in "Estimating Multiple-Input and Multiple-Output ARX Orders" on page 3-64.

For single-output data, the `arx` and `iv4` commands produce an `idpoly` model object, and for multiple-output data these commands produce an `idarx` model object.

You can use the following general syntax to both configure and estimate ARX models:

```
% Using ARX method
m = arx(data,[na nb nk],'Property1',Value1,...,
                        'PropertyN',ValueN)
% Using IV method
m = iv4(data,[na nb nk],'Property1',Value1,...,
                        'PropertyN',ValueN)
```

data is the estimation data and [na nb nk] specifies the model orders, as discussed in “What Are Black-Box Polynomial Models?” on page 3-39.

The property-value pairs specify any model properties that configure the estimation algorithm and the initial conditions. For more information about accessing and setting model properties, see “Model Properties” on page 1-17.

Note You can specify all property-value pairs as a comma-separated list.

To get discrete-time models, use the time-domain data (iddata object). To get a single-output continuous-time model, apply d2c to a discrete-time model or use continuous-time frequency-domain data—either idfrd object, or frequency-domain iddata with Ts=0.

Note The System Identification Toolbox product does not support multiple-output continuous-time idarx models.

For more information about validating you model, see “Validating Models After Estimation” on page 8-3.

You can use pem to refine parameter estimates of an existing polynomial model, as described in “Refining Linear Parametric Models” on page 3-104.

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 “Using Linear Model for Nonlinear ARX Estimation” on page 4-28.

Using pem to Estimate Polynomial Models

You can estimate any single-output polynomial model using the iterative prediction-error estimation method pem. For Gaussian disturbances, this

method gives the maximum likelihood estimate. that minimizes the prediction errors to obtain maximum-likelihood values. The resulting models are stored as `idpoly` model objects.

Use the following general syntax to both configure and estimate polynomial models:

```
m = pem(data,'na',na,
             'nb',nb,
             'nc',nc,
             'nd',nb,
             'nf',nc,
             'nk',nk,
             'Property1',Value1,...,
             'PropertyN',ValueN)
```

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. If you skip any property-value pair, the corresponding parameter value is set to zero—except `nk`, which has the default value 1. For more information about model orders, see "What Are Black-Box Polynomial Models?" on page 3-39.

Tip You do not need to construct the model object using `idoly` 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 0.

Note To get faster estimation of ARX models, use `arx` or `iv4` instead of `pem`.

In addition to the polynomial models listed in "What Are Black-Box Polynomial Models?" on page 3-39, you can use `pem` 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 property-value pairs specify any model properties that configure the estimation algorithm and the initial conditions. You can enter all property-value pairs in `pem` as a comma-separated list without worrying about the hierarchy of these properties in the `idpoly` model object. For more information about accessing and setting model properties, see "Model Properties" on page 1-17.

For multiple inputs, `nb`, `nf`, and `nk` are row vectors of the same lengths as the number of input channels:

```
nb = [nb1 ...  nbnu];
nf = [nf1 ...  nfnu];
nk = [nk1 ...  nknu];
```

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 `pem` 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])
```

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 `Focus` property during estimation. For example, to model only data in the frequency range 0-10 rad/s, use the `Focus` property, as follows:

```
m = oe(Data,[nb nf nk],'Focus',[0 10])
```

For more information about validating your model, see "Validating Models After Estimation" on page 8-3.

You can use `pem` to refine parameter estimates of an existing polynomial model, as described in "Refining Linear Parametric Models" on page 3-104.

Tip For ARX and OE models, you can use the model for initializing a nonlinear estimation at the command line, which may improve the fit of the model. See "Using Linear Model for Nonlinear ARX Estimation" on page 4-28, and "Using Linear Model for Hammerstein-Wiener Estimation" on page 4-64.

For detailed information about `pem` and `idpoly`, see the corresponding reference page.

Estimating Multiple-Input and Multiple-Output ARX Orders

To estimate a multiple-input and multiple-output (MIMO) ARX model, you must first specify the model order matrices, as follows:

- `NA` — An *ny*-by-*ny* matrix whose *i*-*j*th entry is the order of the polynomial that relates the *j*th output to the *i*th output.
- `NB` — An *ny*-by-*nu* matrix whose *i*-*j*th entry is the order of the polynomial that relates the *j*th input to the *i*th output.
- `NK` — An *ny*-by-*nu* matrix whose *i*-*j*th entry is the delay from the *j*th input to the *i*th output.
- For `ny` outputs and `nu` inputs, the *A* coefficients are *ny*-by-*ny* matrices and the *B* coefficients are *ny*-by-*nu* matrices. For more information about MIMO ARX structure, see "Definition of Multiple-Output ARX Models" on page 3-44.

Note For multiple-output time-series models, only AR models are supported. AR models require only the `NA` matrix.

In the System Identification Tool GUI. 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. You can use the following syntax to estimate a model with these orders:

```
arx(data,'na',NA,'nb',NB,'nk',NK)
```

Tip To simplify entering large matrices orders in the System Identification Tool GUI, define the variable `NN=[NA NB NK]` at the command line. You can specify this variable in the **Orders** field.

Assigning Estimation Weightings

You can specify how the estimation algorithm weighs the fit at various frequencies. This information supports the estimation procedures "How to Estimate Polynomial Models in the GUI" on page 3-56 and "Using pem to Estimate Polynomial Models" on page 3-61.

In the System Identification Tool GUI. 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 8-76.
- `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 2-122 or "Defining a Custom Filter" on page 2-123. This prefiltering applies only for estimating the dynamics from input to output. The disturbance model is determined from the unfiltered estimation data.

At the command line. Specify the focus as an argument in the model-estimation command using the same options as in the GUI. For example, use this command to estimate an ARX model and emphasize the frequency content related to the input spectrum only:

```
m=arx(data,[2 2 3],'Focus','Simulation')
```

This `Focus` setting might produce more accurate simulation results.

Specifying Initial States for Iterative Estimation Algorithms

When you use the iterative estimation algorithm PEM to estimate ARMAX, Box-Jenkins (BJ), Output-Error (OE), you must specify how the algorithm treats initial states.

This information supports the estimation procedures "How to Estimate Polynomial Models in the GUI" on page 3-56 and "Using pem to Estimate Polynomial Models" on page 3-61.

In the System Identification Tool GUI. 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 states as an argument in the model-estimation command. For example, use this command to estimate an ARMAX model and set the initial states to zero:

```
m=armax(data,[2 2 2 3],'InitialState','zero')
```

For a complete list of values for the `InitialState` model property, see the `idpoly` reference page.

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 *instrumental 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 Chapter 6, "Time Series Identification". For more information about working with AR and ARX models, see "Identifying Input-Output Polynomial Models" on page 3-39.

Example – Estimating Models Using armax

You can use estimation commands to both construct a model object and estimate the model parameters. In this example, you 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 1-12.

1 Load a sample data set `z8` with three inputs and one output, measured at `1`-second intervals and containing 500 data samples:

```
load iddata8
```

2 Use `armax` to both construct the `idpoly` model object, and estimate the parameters:

$$A(q)y(t) = \sum_{i=1}^{nu} B_i(q)u_i\left(t - nk_i\right) + C(q)e(t)$$

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.

```
m_armax=armax(z8,'na',4,...
                 'nb',[3 2 3],...
                 'nc',4,...
                 'nk',[0 0 0],...
                 'focus', 'simulation',...
                 'tolerance',1e-5,...
                 'maxiter',50);
```

covariance, focus, tolerance, and maxiter are optional arguments specify additional information about the computation. focus specifies whether the model is optimized for simulation or prediction applications, tolerance and maxiter specify when to stop estimation. For more information about these properties, see the algorithm properties reference page.

armax is a version of pem with simplified syntax for the ARMAX model structure. The armax method both constructs the idpoly model object and estimates its parameters.

Tip Instead of specifying model orders and delays as individual property-value pairs, you can use the equivalent shorthand notation that includes all of the order integers in a single vector, as follows:

```
m_armax=armax(z8,[4 3 2 3 4 0 0 0],...
                 'focus', 'simulation',...
                 'tolerance',1e-5,...
                 'maxiter',50);
```

---

3 To view information about the resulting model object, type the following at the prompt:

```
m_armax
```

MATLAB returns the following information about this model object:

```
Discrete-time IDPOLY model: A(q)y(t) = B(q)u(t) + C(q)e(t)
A(q) = 1 - 1.284 q^-1 + 0.3048 q^-2 + 0.2648 q^-3 - 0.05708 q^-4

B1(q) = -0.07547 + 1.087 q^-1 + 0.7166 q^-2

B2(q) = 1.019 + 0.1142 q^-1

B3(q) = -0.06739 + 0.06829 q^-1 + 0.5509 q^-2

C(q) = 1 - 0.06097 q^-1 - 0.1296 q^-2 + 0.02488 q^-3 - 0.04698 q^-4

Estimated using ARMAX from data set z8
Loss function 0.957009 and FPE 1.02068
Sampling interval: 1
```

`m_armax` is an `idpoly` model object. The coefficients represent estimated parameters of this polynomial model.

Tip You can use `present(m_armax)` to show additional information about the model, including parameter uncertainties.

---

4 To view all property values for this model, type the following command:

```
get(m_armax)
```

MATLAB returns the following information:

```
ans =

                   a: [1 -1.2836 0.3048 0.2648 -0.0571]
                   b: [3x3 double]
                   c: [1 -0.0610 -0.1296 0.0249 -0.0470]
```

```
                d: 1
                f: [3x1 double]
               da: [0 0.1012 0.1804 0.1210 0.0303]
               db: [3x3 double]
               dc: [0 0.1111 0.0767 0.0484 0.0460]
               dd: 0
               df: [3x1 double]
               na: 4
               nb: [3 2 3]
               nc: 4
               nd: 0
               nf: [0 0 0]
               nk: [0 0 0]
     InitialState: 'Auto'
             Name: ''
               Ts: 1
        InputName: {3x1 cell}
        InputUnit: {3x1 cell}
       OutputName: {'y1'}
       OutputUnit: {''}
         TimeUnit: ''
  ParameterVector: [16x1 double]
            PName: {}
 CovarianceMatrix: [16x16 double]
    NoiseVariance: 0.9899
       InputDelay: [3x1 double]
        Algorithm: [1x1 struct]
   EstimationInfo: [1x1 struct]
            Notes: {}
         UserData: []
```

5 The `Algorithm` and `EstimationInfo` model properties are structures. To view the properties and values inside these structure, use dot notation. For example:

```
m_armax.Algorithm
```

This action displays the complete list of `Algorithm` properties and values that specify the iterative computational algorithm:

```
ans =

           Focus: 'Simulation'
         MaxIter: 50
       Tolerance: 1.0000e-005
      LimitError: 0
         MaxSize: 'Auto'
    SearchMethod: 'Auto'
       Criterion: 'det'
       Weighting: 1
  FixedParameter: []
         Display: 'Off'
        N4Weight: 'Auto'
       N4Horizon: 'Auto'
        Advanced: [1x1 struct]
```

Similarly, to view the properties and values of the `EstimationInfo` structure, type the following command:

```
m_armax.EstimationInfo
```

This action displays the complete list of read-only `EstimationInfo` properties and values that describe the estimation data set, quantitative measures of model quality (loss function and FPE), the number of iterations actually used, and the behavior of the iterative model estimation.

```
ans =

            Status: 'Estimated model (PEM with focus)'
            Method: 'ARMAX'
           LossFcn: 0.9570
               FPE: 1.0207
          DataName: 'z'
        DataLength: 500
            DataTs: 1
        DataDomain: 'time'
   DataInterSample: {3x1 cell}
           WhyStop: 'Near (local) minimum, (norm(g) < tol).'
        UpdateNorm: 4.7296e-006
   LastImprovement: 4.7296e-006
        Iterations: 4
      InitialState: 'Zero'
           Warning: ''
```

6 If you want to repeat the model estimation using different model orders, but keep the algorithm properties the same, you can store the model properties used for `m_armax` in a variable, as follows:

```
myAlg=m_armax.Algorithm;
```

This action stores the specified `focus`, `tolerance`, and `maxiter`, and the default algorithm.

7 To reuse the algorithm properties in estimating the ARMAX model with different orders, use the following command:

```
m_armax2=armax(z8,[4 3 2 3 3 1 1 1],...
                 'algorithm',myAlg);
```

Identifying State-Space Models

In this section...

"What Are State-Space Models?" on page 3-73

"Data Supported by State-Space Models" on page 3-77

"Supported State-Space Parameterizations" on page 3-78

"Preliminary Step – Estimating State-Space Model Orders" on page 3-79

"How to Estimate State-Space Models in the GUI" on page 3-84

"How to Estimate State-Space Models at the Command Line" on page 3-87

"How to Estimate Free-Parameterization State-Space Models" on page 3-91

"How to Estimate State-Space Models with Canonical Parameterization" on page 3-92

"How to Estimate State-Space Models with Structured Parameterization" on page 3-94

"How to Estimate the State-Space Equivalent of ARMAX and OE Models" on page 3-100

"Assigning Estimation Weightings" on page 3-101

"Specifying Initial States for Iterative Estimation Algorithms" on page 3-102

"State-Space Model Estimation Algorithms" on page 3-103

What Are State-Space Models?

- "Definition of State-Space Models" on page 3-74
- "Continuous-Time Representation" on page 3-74
- "Discrete-Time Representation" on page 3-75
- "Relationship Between Continuous-Time and Discrete-Time State Matrices" on page 3-75
- "State-Space Representation of Transfer Functions" on page 3-76

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 only two parameters:

- `n` — Model order or the number of poles (size of the *A* matrix).
- `nk` — One or more input delays.

The *model order* for state-space models is an integer equal to the dimension of *x(t)* and relates to the number of delayed inputs and outputs used in the corresponding linear difference equation.

Continuous-Time Representation

In continuous-time, the state-space description has the following form:

$$\dot{x}(t) = Fx(t) + Gu(t) + \tilde{K}w(t)$$
$$y(t) = Hx(t) + Du(t) + w(t)$$
$$x(0) = x0$$

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 this case, the matrices *F*, *G*, *H*, and *D* contain elements with physical significance—for example, material constants. *x0* specifies the initial states.

Note *K*=0 gives the state-space representation of an Output-Error model. For more information about Output-Error models, see “What Are Black-Box Polynomial Models?” on page 3-39.

Discrete-Time Representation

Discrete-time state-space models provide the same type of linear difference relationship between the inputs and the outputs as the linear ARX model, but are rearranged such that there is only one delay in the expressions. 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 sampling interval, $u(kT)$ 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 Black-Box Polynomial Models?" on page 3-39.

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 sampling intervals 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 symbolic 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 discrete-time state-space representation is given by the following equation:

$$\begin{aligned} x(kT+T) &= Ax(kT) + Bu(kT) + Ke(kT) \\ y(kT) &= Cx(kT) + Du(kT) + e(kT) \\ x(0) &= x0 \end{aligned}$$

where T is the sampling interval, $u(kT)$ is the input at time instant kT, and $y(kT)$ is the output at time instant kT.

The relationships between the transfer functions and the discrete-time state-space matrices are given by the following equations:

$$\begin{aligned} G(q) &= C(qI_{nx} - A)^{-1}B + D \\ H(q) &= C(qI_{nx} - A)^{-1}K + I_{ny} \end{aligned}$$

where I_{nx} is the nx-by-nx identity matrix, I_{ny} is the nx-by-nx identity matrix, and ny is the dimension of y and e.

Data Supported by State-Space Models

- "Types of Supported Data" on page 3-77
- "Estimating Continuous-Time Models" on page 3-77
- "Designating Data for Estimating Discrete-Time Models" on page 3-78

Types of Supported Data

You can estimate linear state-space models from data with the following characteristics:

- Real data or complex data in any domain
- Single-output and multiple-output
- Time- or frequency-domain data

  To estimate state-space models for time-series data, see Chapter 6, "Time Series Identification".

You must first import your data into the MATLAB workspace, as described in Chapter 2, "Data Import and Processing".

Estimating Continuous-Time Models

Use either of the following ways to estimate continuous-time, state-space 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.
- Use continuous-time frequency-domain data.

  To denote continuous-time frequency-domain data, set the data sampling interval to 0. You can set the sampling interval when you import data into the GUI or set the `Ts` property of the data object at the command line.

Tip Continuous state-space models are available for canonical and structured parameterizations and grey-box models. In this case, no disturbance model can be estimated.

Designating Data for Estimating Discrete-Time Models

You can estimate arbitrary-order, linear state-space models for both time- or frequency-domain data.

You must specify your data to have the sampling interval equal to the experimental data sampling interval.

You can set the sampling interval when you import data into the GUI or set the `Ts` property of the data object at the command line.

Supported State-Space Parameterizations

The System Identification Toolbox product 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 system matrix elements A, B, C, D, and K.
- Canonical forms of A, B, C, D, and K matrices.

  Canonical parameterization represents a state-space system in its minimal form, using the minimum number of free parameters to capture the dynamics. Thus, free parameters appear in only a few of the rows and columns in system matrices A, B, C, and D, and the remaining matrix elements are fixed to zeros and ones.
- Structured parameterization lets you specify the values of specific parameters and exclude these parameters from estimation.
- Completely arbitrary mapping of parameters to state-space matrices. For more information, see “Estimating Linear Grey-Box Models” on page 5-6.

You can only estimate free state-space models in discrete time. Continuous state-space models are available for canonical and structured parameterizations and grey-box models.

Note To estimate canonical and structured state-space models in the System Identification Tool GUI, define the corresponding model structures at the command line and import them into the System Identification Tool GUI.

Preliminary Step – Estimating State-Space Model Orders

- "Why Estimate Model Orders?" on page 3-79
- "Estimating Model Order in the GUI" on page 3-79
- "Estimating the Model Order at the Command Line" on page 3-82
- "Using the Model Order Selection Window" on page 3-83

Why Estimate Model Orders?

To estimate a state-space model, you must provide a model order and one or more input delays.

To get an initial model order for your system, you can estimate a group of state-space models with a range of orders for a specific delay and compare the performance of these models. You choose the model order that include states with the highest contribution to the input-output behavior of the model and use this order as an initial guess for further modeling.

The model order is always a single integer—regardless of the number of inputs and outputs. However, the number of input delays must correspond to the number of input channels.

Estimating Model Order in the GUI

You must have already imported your data into the GUI, as described in "Importing Data into the GUI" on page 2-17.

To estimate model orders for a specific input delay:

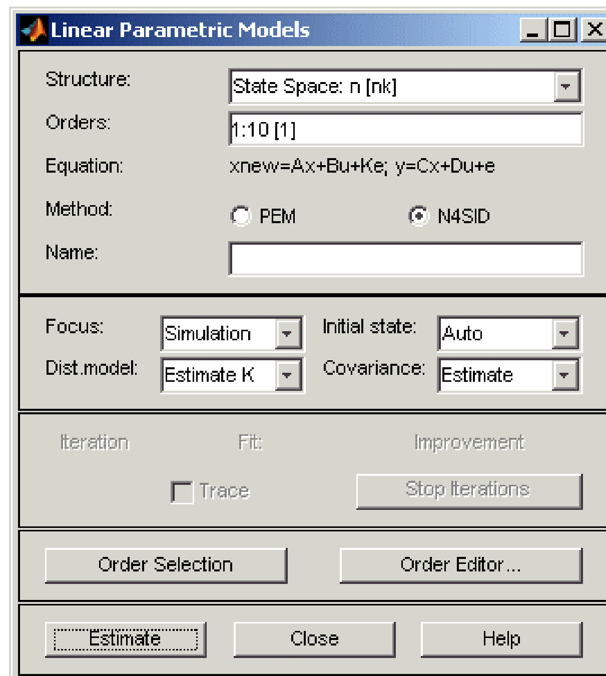
1 In the System Identification Tool GUI, select **Estimate > Linear parametric models** to open the Linear Parametric Models dialog box.

2 In the **Structure** list, select `State Space:  n [nk]`.

3 Edit the **Orders** field to specify a range of orders for a specific delay. For example, enter the following values for *n* and *nk*:

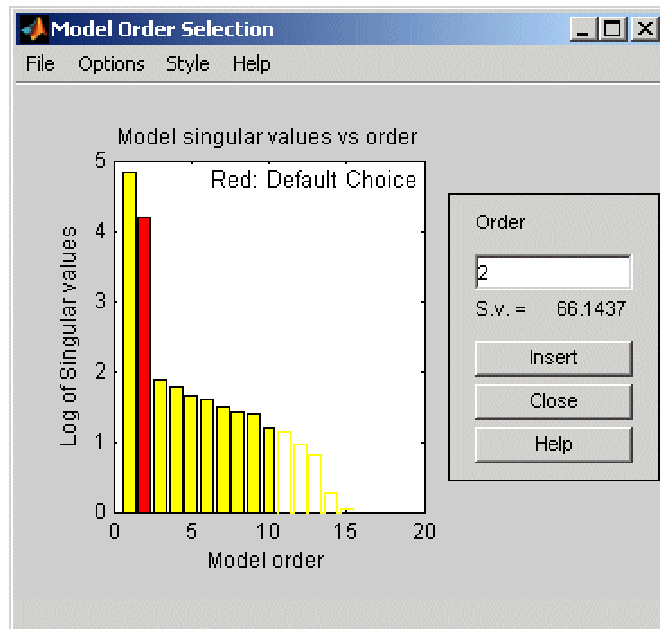
```
1:10 [1]
```

Tip As a shortcut for entering `1:10 [1]`, click **Order Selection**.



4 Verify that the **Method** is set to **N4SID**.

5 Click **Estimate** to open 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, and click **Insert** to estimate a model with this order. Red indicates the recommended choice. States 1 and 2 provide the most significant contribution. The contributions to the right of state 2 drop significantly. For information about using the Model Order Selection window, see "Using the Model Order Selection Window" on page 3-83.

This action adds a new model to the Model Board in the System Identification Tool GUI. The default name of the parametric model combines the string `n4s` and the selected model order.

7 Click **Close** to close the Model Order Selection window.

After estimating model orders, use this value as an initial guess for estimating other state-space models, as described in "How to Estimate State-Space Models in the GUI" on page 3-84.

Estimating the Model Order at the Command Line

You can estimate the state-space model order using the `n4sid` command.

Use following syntax to specify the range of model orders to try for a specific input delay.

```
m = n4sid(data,n1:n2,'nk',nk);
```

where `data` is the estimation data set, `n1` and `n2` specify the range of orders, and `nk` specifies the input delay. For multiple-input systems, `nk` is a vector of input delays.

This command opens the Model Order Selection window. For information about using this plot, see "Using the Model Order Selection Window" on page 3-83.

Alternatively, you can use the `pem` command to open the Model Order Selection window, as follows:

```
m = pem(Data,'nx',nn)
```

where `nn = [n1,n2,...,nN]` specifies the vector or range of orders you want to try.

To omit opening the Model Order Selection window and automatically select the best order, use the following syntax:

```
m = pem(Data,'best')
```

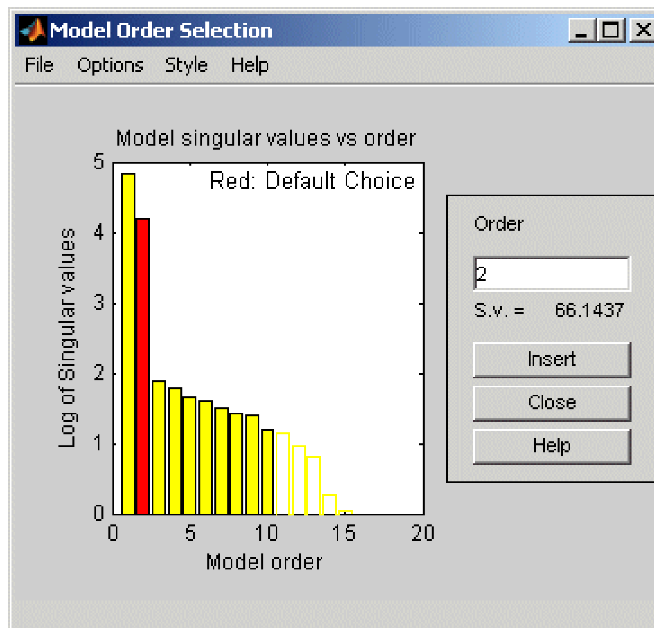
For a tutorial on estimating model orders for a multiple-input system, see "Estimating a State-Space Model" in *System Identification Toolbox Getting Started Guide*.

Using the Model Order Selection Window

You can generate the Model Order Selection window for your data to select the number of states that provide the highest relative contribution to the input-output behavior of the model (*log of singular values of the covariance matrix*).

For a procedure on generating this plot in the System Identification Tool GUI, see "Estimating Model Order in the GUI" on page 3-79. To open this plot at the command line, see "Estimating the Model Order at the Command Line" on page 3-82.

The following figure shows a sample Model Order Selection window.



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.

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 red.

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.

How to Estimate State-Space Models in the GUI

- "Supported State-Space Models in the GUI" on page 3-84
- "Prerequisites" on page 3-84
- "Estimating State-Space Models in the GUI" on page 3-85
- "Next Steps" on page 3-87

Supported State-Space Models in the GUI

Only free parameterization is directly supported in the System Identification Tool GUI. You can also estimate canonical and structured parameterizations at the command line and import them into the System Identification Tool GUI for parameter estimation. For more information about state-space parameterization, see "Supported State-Space Parameterizations" on page 3-78.

Prerequisites

Before you can perform this task, you must have

- Imported data into the System Identification Tool GUI. See "Importing Time-Domain Data into the GUI" on page 2-18. For supported data formats, see "Data Supported by State-Space Models" on page 3-77.
- 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.

- Select a model order. For more information about how to estimate model orders, see "Preliminary Step – Estimating State-Space Model Orders" on page 3-79.

Estimating State-Space Models in the GUI

To estimate a state-space model with free parameterization in the System Identification Tool GUI:

1. In the System Identification Tool GUI, select **Estimate > Linear parametric models** to open the Linear Parametric Models dialog box.

2. In the **Structure** list, select `State Space:  n [nk]`.

   This action updates the options in the Linear Parametric Models dialog box to correspond with this model structure. For information about each model structure, see "What Are State-Space Models?" on page 3-73.

3. In the **Orders** field, specify the model order and delay, as follows:

   - **For single-input models.** Enter the model order integer and the input delay in terms of the number of samples. Omitting `nk` uses the default value `nk=1`.

     For example, enter `4 [2]` for a fourth-order model and `nk=2`.

   - **For multiple-input models.** Enter the model order integer and the input delay vector—which is a 1-by-*nu* vector whose *i*th entry is the delay for the *i*th input.

     For example, for a two-input system, enter `4 [1 1]` for a fourth-order model and a delay of 1 for each input.

   - **For multiple-output models.** Enter the model order integer the same way as for single-input models.

Tip To enter model order and any delays using the Order Editor dialog box, click **Order Editor**.

---

4. Select the estimation **Method** as **N4SID** or **PEM**. For more information about these methods, "State-Space Model Estimation Algorithms" on page 3-103.

5. In the **Name** field, edit the name of the model or keep the default. The name of the model should be unique in the Model Board.

6. 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 3-101.

7. (PEM only) 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 3-102.

Tip If you get an inaccurate fit, try setting a specific method for handling initial states rather than choosing it automatically.

---

8. 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 reduces computation time for complex models and large data sets.

9. (PEM only) To view the estimation progress in the MATLAB Command Window, select the **Trace** check box. During estimation, the following information is displayed for each iteration:

   - Loss function — Equals the determinant of the estimated covariance matrix of the input noise.
   - Parameter values — Values of the model structure coefficients you specified.
   - Search direction — Change in parameter values from the previous iteration.

- Fit improvements — Shows the actual versus expected improvements in the fit.

10 Click **Estimate** to add this model to the System Identification Tool GUI.

11 (PEM 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.

Next Steps

- Validate the model by selecting the appropriate check box in the **Model Views** area of the System Identification Tool GUI. For more information about validating models, see "Validating Models After Estimation" on page 8-3.
- Export the model to the MATLAB workspace for further analysis by dragging it to the **To Workspace** rectangle in the System Identification Tool GUI.

How to Estimate State-Space Models at the Command Line

- "Supported State-Space Models" on page 3-87
- "Prerequisites" on page 3-88
- "Estimating State-Space Models Using pem and n4sid" on page 3-88
- "Common Properties to Specify Model Estimation" on page 3-89
- "Choosing to Estimate D, K, and X0 Matrices" on page 3-90

Supported State-Space Models

You can only estimate discrete-time state-space models with free parameterization. Continuous state-space models are available for canonical and structured parameterizations.

Prerequisites

Before you can perform this task, you must have

- Regularly sampled data as an iddata object. See "Representing Time- and Frequency-Domain Data Using iddata Objects" on page 2-53. For supported data formats, see "Data Supported by State-Space Models" on page 3-77.
- 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.
- Select a model order. For more information about how to estimate model orders, see "Preliminary Step – Estimating State-Space Model Orders" on page 3-79.

Estimating State-Space Models Using pem and n4sid

You can estimate continuous-time and discrete-time polynomial model using the iterative estimation command pem that minimizes the prediction errors to obtain maximum-likelihood values. You can also use the noniterative subspace command n4sid.

Use the following general syntax to both configure and estimate state-space models:

```
m = pem(data,n,
              'nk',nk,
              'Property1',Value1,...,
              'PropertyN',ValueN)
```

where data is the estimation data, n is the model order, and nk specifies the input delays for each input.

As an alternative to pem, you can use n4sid:

```
m = n4sid(data,n,
               'nk',nk,
               'Property1',Value1,...,
               'PropertyN',ValueN)
```

Note `pem` uses `n4sid` to initialize the state-space matrices, and takes longer than `n4sid` to estimate a model but typically provides better fit to data.

For more information about the most common property-value pairs you can specify, see "Common Properties to Specify Model Estimation" on page 3-89.

For detailed information about the syntax, see the corresponding reference page.

For more information about estimating model order, see "Estimating the Model Order at the Command Line" on page 3-82.

For information about validating your model, see "Validating Models After Estimation" on page 8-3

Common Properties to Specify Model Estimation

The following properties are common to specify in the estimation syntax:

- `SSparameterization` — Specifies the state-space parameterization form. For more information about estimating a specific state-space parameterization, see the following topics:
  - "How to Estimate Free-Parameterization State-Space Models" on page 3-91
  - "How to Estimate State-Space Models with Canonical Parameterization" on page 3-92
  - "How to Estimate State-Space Models with Structured Parameterization" on page 3-94
- `Focus` — Specifies the frequency-weighing of the noise model during estimation. See "Assigning Estimation Weightings" on page 3-101.
- `DisturbanceModel` — Specifies to estimate or omit the noise model for time-domain data. See "K Matrix" on page 3-90.
- `InitialStates` — Specifies to set or estimate the initial states. See "Specifying Initial States for Iterative Estimation Algorithms" on page 3-102.

For more information about these properties, see the `idss` reference page.

Choosing to Estimate D, K, and X0 Matrices

For state-space models with any parameterization, you can specify whether to estimate the *K* and *X0* matrices, which represent the 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 of `nk`.

For more information about state-space structure, see "What Are State-Space Models?" on page 3-73.

D Matrix. By default, the *D* matrix is not estimated. Set the model property `nk` to estimate the D matrix, as follows:

- To estimate the *k*th column of *D* (corresponding to the *k*th input), set `nk` to 0. For `nu` inputs, `nk` is a 1-*by*-`nu` vector.
- To estimate the full *D* matrix, set all `nk` values to 0. For example, for two inputs:

```
m = pem(Data,n,'nk',[0 0])
```

To omit estimating the D matrix, set the `nk` value or values to 1, which is the default.

K Matrix. *K* represents the noise model.

For frequency-domain data, no noise model is estimated and *K* is set to 0. For time-domain data, *K* is estimated by default.

To modify whether *K* is estimated for time-domain data, you can specify the `DisturbanceModel` property in the estimator syntax.

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 `pem` to refine the model and set the *K* parameters to be estimated. For example:

```
m = pem(Data,md,'DisturbanceModel','Estimate')
```

where `md` is the dynamic model without noise.

To set *K* to zero, set the value of the `DisturbanceModel` property to `'None'`. For example:

```
m = pem(Data,n,'DisturbanceModel','None')
```

XO Matrices. *X0* stores the estimated or specified initial states of the model.

To specify how to handle the initial states, set the value of the `InitialStates` model property. For example, to set the initial states to zero, set the `InitialStates` property to `'zero'`, as follows:

```
m = pem(Data,n,'InitialStates','zero')
```

When you estimate models using multiexperiment data and `InitialStates` is set to `'Estimate'`, *X0* stores the estimated initial states corresponding to the last experiment in the data set.

For a complete list of values for the `InitialStates` property, see “Specifying Initial States for Iterative Estimation Algorithms” on page 3-102.

How to Estimate Free-Parameterization State-Space Models

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.

You can only estimate discrete-time state-space models with any parameterization. Continuous state-space models are available for canonical and structured parameterizations only.

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 = pem(data)
```

where `data` is your estimation data. This command estimates a state-space model for an automatically selected order between 1 and 10.

To find a black-box model of a specific order `n`, use the following syntax:

```
m = pem(Data,n)
```

The iterative algorithm pem is initialized by the subspace method `n4sid`. You can use `n4sid` directly, as an alternative to `pem`:

```
m = n4sid(Data,n)
```

How to Estimate State-Space Models with Canonical Parameterization

- "What Is Canonical Parameterization?" on page 3-92
- "Estimating Canonical State-Space Models" on page 3-93

What Is Canonical Parameterization?

Canonical parameterization represents a state-space system in its minimal form, using the minimum number of free parameters to capture the dynamics. Thus, free parameters appear in only a few of the rows and columns in system matrices A, B, C, and D, and the remaining matrix elements are fixed to zeros and ones.

Of the two popular canonical forms, which include *controllable canonical form* and *observable canonical form*, the toolbox supports only controllable

forms. Controllable canonical structures include free parameters in output rows of the *A* matrix, free *B* and *K* matrices, and the fixed *C* matrix. The representation within controllable canonical forms is not unique and the exact form depends on the actual choices of canonical indices. For more information about the distribution of free parameters in canonical forms, see the appendix 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 canonical parameterization at the command line.

To specify a canonical form for `A`, `B`, `C`, and `D`, set the `SSparameterization` model property directly in the estimator syntax, as follows:

```
m = pem(data,n,'SSparameterization','canonical')
```

If you have time-domain data, the preceding command estimates a discrete-time model.

Note When you estimate the *D* matrix in canonical form, you must set the `nk` property. See "Choosing to Estimate D, K, and X0 Matrices" on page 3-90.

If you have continuous-time frequency-domain data, the preceding syntax estimates an `n`th order continuous-time state-space model with no direct contribution from the input to the output (*D*=0). To include a *D* matrix, set the `nk` property to 0 in the estimation, as follows:

```
m = pem(data,n,'SSparameterization','canonical',
               'nk',0)
```

You can specify additional property-value pairs similar to the free-parameterization case, as described in "How to Estimate Free-Parameterization State-Space Models" on page 3-91.

For information about validating your model, see "Validating Models After Estimation" on page 8-3.

How to Estimate State-Space Models with Structured Parameterization

- "What Is Structured Parameterization?" on page 3-94
- "Specifying the State-Space Structure" on page 3-95
- "Are Grey-Box Models Similar to State-Space Models with Structured Parameterization?" on page 3-96
- "Example – Estimating Structured Discrete-Time State-Space Models" on page 3-97
- "Example – Estimating Structured Continuous-Time State-Space Models" on page 3-98

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.

In the case of structured parameterization, there are two stages to the estimation procedure:

1. Using the `idss` command to specify the structure of the state-space matrices and the initial values of the free parameters

2. Using the `pem` estimation command to estimate the free model parameters

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.

For information about validating your model, see "Validating Models After Estimation" on page 8-3.

Specifying the State-Space Structure

To specify the state-space model structure, first define the A, B, C, D, K and X0 matrices in the MATLAB workspace.

To define a discrete-time state-space structure, use the following syntax:

```
m = idss(A,B,C,D,K,X0,...
         'Ts',T,...
         'SSparameterization','structured')
```

where A, B, C, D, and K specify both the fixed parameter values and the initial values for the free parameters. T is the sampling interval. Setting SSparameterization to 'structured' flags that you want to estimate a partial structure for this state-space model.

Similarly, to define a continuous-time state-space structure, use the following syntax:

```
m = idss(A,B,C,D,K,X0,...
         'Ts',0,...
         'SSparameterization','structured')
```

In the continuous-time case, you must set the sampling interval property Ts to zero.

After you create the nominal model structure, you must specify which parameters to estimate and which to set to specific values. To accomplish this, you must edit the structures of the following model properties: As, Bs, Cs, Ds, Ks, and x0s. These *structure matrices* are properties of the nominal model you constructed and have the same sizes as A, B, C, D, K, and x0, respectively. Initially, the structure matrices contain NaN values.

Specify the structure matrix values, as follows:

- Set a NaN value to flag free parameters at the corresponding locations in A, B, C, D, K, and x0.
- Specify the values of fixed parameters at the corresponding locations in A, B, C, D, K, and x0.

For example, suppose that you constructed a nominal state-space model m with the following A matrix:

```
A = [2 0; 0 3]
```

Suppose you want to fix A(1,2)=A(2,1)=0. To specify the parameters you want to fix, enter their values at the corresponding locations in the structure matrix As:

```
m.As = [NaN 0; 0 NaN]
```

The estimation algorithm only estimates the parameters in A that have a NaN value in As.

Finally, use pem to estimate the model, as described in "How to Estimate State-Space Models at the Command Line" on page 3-87.

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 the init command. 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.

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 code file.

Are Grey-Box Models Similar to State-Space Models with Structured Parameterization?

Structured parameterization state-space models are similar to grey-box modeling. However, the state-space models are simpler to estimate than grey-box models. To learn more about grey-box models, see Chapter 5, "ODE Parameter Estimation (Grey-Box Modeling)".

Example – Estimating Structured Discrete-Time State-Space Models

In this example, you estimate the unknown parameters ($\theta_1, \theta_2, \theta_3, \theta_4, \theta_5$) 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:

$$x(kT+T) = Ax(kT) + Bu(kT) + Ke(kT)$$

$$y(kT) = Cx(kT) + Du(kT) + e(kT)$$

$$x(0) = x0$$

To construct and estimate the parameters of this discrete-time state-space model:

1 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];
```

2 Construct the state-space model object:

```
m = idss(A,B,C,D,K);
```

3 Specify the parameter values in the structure matrices that you do not want to estimate:

```
m.As = [1, NaN; 0 ,1];
m.Bs = [NaN;NaN];
m.Cs = [1, 0];
m.Ds = 0;
m.Ks = [NaN;NaN];
m.x0s = [0;0];
```

4 Estimate the model structure:

```
m = pem(data,m)
```

where `data` is name of the `iddata` object containing time-domain or frequency-domain data. The iterative search starts with the nominal values in the `A`, `B`, `C`, `D`, `K`, and `x0` matrices.

Example – Estimating Structured Continuous-Time State-Space Models

In this example, you estimate the unknown parameters $(\theta_1, \theta_2, \theta_3)$ 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 $-\theta_2/\theta_1$ 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 $\theta_1 = -1$ 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:

$$\dot{x}(t) = Fx(t) + Gu(t) + \tilde{K}w(t)$$
$$y(t) = Hx(t) + Du(t) + w(t)$$
$$x(0) = x0$$

To construct and estimate the parameters of this continuous-time state-space model:

1 Construct the parameter matrices and initialize the parameter values using the nominal parameter values:

Note The following 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`.

```
A = [0 1;0 -1];
B = [0;0.25];
C = eye(2);
D = [0;0];
K = zeros(2,2);
x0 = [0;0];
```

2 Construct the continuous-time state-space model object:

```
m = idss(A,B,C,D,K,x0,'Ts',0);
```

3. Specify the parameter values in the structure matrices that you do not want to estimate:

```
m.As = [0 1;0 NaN];
m.Bs = [0;NaN];
m.Cs = m.c;
m.Ds = m.d;
m.Ks = m.k;
m.x0s = [NaN;0]
m.NoiseVariance = [0.01 0; 0 0.1];
```

4. Estimate the model structure:

```
m = pem(data,m)
```

where `data` is name of the `iddata` object containing time-domain or frequency-domain data. 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 sampling interval as the data.

5. To simulate this system using the sampling interval `T = 0.1` for input `u` and the noise realization `e`, use the following commands:

```
e = randn(300,2);
u = idinput(300);
simdat = iddata([],[u e],'Ts',0.1);
y = sim(m,simdat)
```

The continuous system is automatically sampled using `Ts=0.1`. The noise sequence is scaled according to the matrix `m.noisevar`.

If you discover that the motor was not initially at rest, you can estimate $x_2(0)$ by setting the second element of the `x0s` structure matrix to `NaN`, as follows:

```
m_new = pem(data,m,'x0s',[NaN;NaN])
```

How to Estimate the State-Space Equivalent of ARMAX and OE Models

You can estimate the equivalent of ARMAX and output-error (OE) multiple-output models using state-space model structures. For the ARMAX

case, specify to estimate the *K* matrix for the state-space model. For the OE case, set K = 0.

Tip 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 "Using Linear Model for Hammerstein-Wiener Estimation" on page 4-64.

For more information about ARMAX and OE models, see "Identifying Input-Output Polynomial Models" on page 3-39.

Assigning Estimation Weightings

You can specify how the estimation algorithm weighs the fit at various frequencies. This information supports the estimation procedures "How to Estimate State-Space Models in the GUI" on page 3-84 and "How to Estimate State-Space Models at the Command Line" on page 3-87.

In the System Identification Tool GUI. 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 8-76.
- `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 2-122 or "Defining a Custom Filter" on page 2-123. This prefiltering

applies only for estimating the dynamics from input to output. The disturbance model is determined from the estimation data.

At the command line. Specify the focus as an argument in the model-estimation command using the same options as in the GUI. For example, use this command to emphasize the fit between the 5 and 8 rad/s:

```
pem(data,4,'Focus',[5 8])
```

Specifying Initial States for Iterative Estimation Algorithms

If you estimate state-space models using the iterative estimation algorithm `pem`, you must specify how the algorithm treats initial states. This information supports the estimation procedures "How to Estimate State-Space Models in the GUI" on page 3-84 and "How to Estimate State-Space Models at the Command Line" on page 3-87.

In the System Identification Tool GUI. 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 initial states as an argument in the estimation command `pem`. For example, use this command to estimate a fourth-order state-space model and set the initial states to be estimated from the data:

```
m=pem(data,4,'InitialState','estimate')
```

For a complete list of values for the `InitialState` model property, see the `idss` reference page.

State-Space Model Estimation Algorithms

For linear state-space models, you can use the subspace method, called *N4SID*. You can use the subspace method N4SID to get an initial model (see the `n4sid` reference page), and then try to refine the initial estimate using the iterative prediction-error method PEM (see the `pem` reference page).

N4SID is faster than PEM, but is typically less accurate and robust, and requires additional arguments that might be difficult to specify.

You can use the iterative *prediction-error minimization* (*PEM*) (maximum likelihood) algorithm for all linear and nonlinear model types.

Refining Linear Parametric Models

In this section...

"When to Refine Models" on page 3-104

"What You Specify to Refine a Model" on page 3-104

"How to Refine Linear Parametric Models in the GUI" on page 3-105

"How to Refine Linear Parametric Models at the Command Line" on page 3-106

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 refine the model. 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 Model Structures" on page 1-16. In this case, you built initial parameter guesses into the model structure and wish to refine these parameter values.

Note Because it is difficult to specify nonlinear model parameters in advance, you typically only estimate nonlinear models.

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.

How to Refine Linear Parametric Models in the GUI

The following procedure assumes that the model you want to refine is already in the System Identification Tool GUI. 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 GUI, see “Importing Models into the GUI” on page 11-8.

To refine your model:

1. In the System Identification Tool GUI, 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 “Specifying Estimation and Validation Data” on page 2-35.

2. Select **Estimate > Linear parametric models** to open the Linear Parametric Models dialog box, if this dialog box is not already open.

3. In the Linear Parametric Models dialog box, select `By Initial Model` from the **Structure** list.

4. Enter the model name into the **Initial model** field, and press **Enter**.

   The model name must be in the Model Board of the System Identification Tool GUI or a variable in the MATLAB workspace.

   Tip As a shortcut for specifying a model in the Model Board, you can drag the model icon from the System Identification Tool GUI into the **Initial model** field.

   When you enter the model name, algorithm settings in the Linear Parametric Models dialog box override the initial model settings.

5. Modify the algorithm settings, displayed in the Linear Parametric Models dialog box, if necessary.

6. Click **Estimate** to refine the model.

7 Validate the new model, as described in Chapter 8, "Model Analysis".

Tip To continue refining the model using additional iterations, click **Continue iter**. This action continues parameter estimation using the most recent model.

How to 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.

The general syntax for refining initial models is as follows:

```
m = pem(data,init_model)
```

`pem` uses the properties of the initial model unless you specify different properties. For more information about specifying model properties directly in the estimator, see "Specifying Model Properties for Estimation" on page 1-19.

Example – Refining an Initial ARMAX Model at the Command Line

The following example shows to estimate an initial model and try to refine this model using `pem`:

```
load iddata8

% Split the data z8 into two parts.
% Create new data object with first hundred samples
z8a = z8(1:100);

% Create new data object with remaining samples
z8b = z8(101:end);

% Estimate ARMAX model with default Algorithm
% properties, na=4, nb=[3 2 3], nc=2, and nk=[0 0 0]
m1 = armax(z8a,[4 3 2 3 2 0 0 0]);

% Refine the initial model m1 using the data set z8b,
% and stricter algorithm settings with increased number
% of maximum iterations (MaxIter) and smaller tolerance
m2 = pem(z8b,m1,'tol',1e-5,'maxiter',50);
```

For more information about estimating polynomial models, see "Identifying Input-Output Polynomial Models" on page 3-39.

Example – Refining an ARMAX Model with Initial Parameter Guesses at the Command Line

The following example shows how to refine models for which you have initial parameter guesses. This example estimates an ARMAX model for the data and requires you to initialize the *A*, *B*, and *C* polynomials.

In this case, 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 `pem`, which refines the initial parameter guesses using the 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 pem to refine initial model.
model = pem(z8,init_model)

% Compare the two models.
compare(z8,init_model,model)
```

For more information about estimating polynomial models, see “Identifying Input-Output Polynomial Models” on page 3-39.

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, and polynomials (a, b, c, d and f) for polynomial models.

  If you estimated model uncertainty data, this information is stored in the Model.CovarianceMatrix property of the estimated model. 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, stored in the da property of an idpoly 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 at the input for each output in your system.

The following table summarizes the commands for extracting model coefficients and uncertainty. All of these commands have the following syntax form:

```
[G,dG] = command(model)
```

where G stores model parameters and dG stores standard deviation of parameters or covariance.

Commands for Extracting Model Coefficients and Uncertainty Data

Command	Description	Syntax
`arxdata`	Extracts ARX parameters from multiple-output `idarx` or single-output `idpoly` objects that represent ARX models.	`[A,B,dA,dB] = arxdata(m)`
`freqresp`	Extracts frequency-response data and corresponding covariance from any `idmodel` or `idfrd` object.	`[H,w,CovH] = freqresp(m)`
`polydata`	Extracts polynomials from any single-output `idmodel` object.	`[A,B,C,D,F,dA,dB,dC,dD,dF] = ... polydata(m)`
`ssdata`	Extracts state-space matrices from any `idmodel` object.	`[A,B,C,D,K,X0,... dA,dB,dC,dD,dK,dX0] = ... ssdata(Model)`
`tfdata`	Extracts numerator and denominator polynomials from any `idmodel` object.	`[Num,Den,dNum,dDen] = ... tfdata(Model)`
`zpkdata`	Extracts zeros, poles, and transfer function gains from any `idmodel` object.	`[Z,P,K,covZ,covP,covK] = ... zpkdata(m)`

You can also use `ssdata`, `tfdata`, and `zpkdata` to extract the numerical values of the dynamic and noise models separately, as shown in the following table. Here, *`fcn`* represents `ssdata`, `tfdata`, and `zpkdata`, and `m` is a model object. L represents the covariance matrix e, as defined in "Subreferencing Measured and Noise Models" on page 3-120.

Note The syntax `fcn(m('noise'))` is equivalent to `fcn(m('n'))`.

Syntax for Extracting Dynamic and Noise Model Data

Command	Syntax
`fcn(m)`	Returns the properties of *G* for *ny* outputs and *nu* inputs.
`fcn(m('noise'))`	Returns the properties of *H* for *ny* outputs and *ny* inputs.
`fcn(noisecnv(m))`	Returns the properties of [*G H*] *ny* outputs and *ny*+*nu* inputs.
`fcn(noisecnv(m,'Norm'))`	Returns the properties of [*G HL*] *ny* outputs and *ny*+*nu* inputs, , where `L'L` is the noise variance.
`fcn(noisecnv(m('noise'),'Norm'))`	Returns the properties of *HL* *ny* outputs and *ny* inputs.
`fcn(m)`	If `m` is a time-series model, returns the properties of *H*.
`fcn(noisecnv(m,'Norm'))`	If `m` is a time-series model, returns the properties of *HL*.

Note The estimated covariance matrix `NoiseVariance` is uncertain. Thus, the uncertainty of *H* differs from the uncertainty of *HL*.

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)`.

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 3-112
"Using the c2d, d2c, and d2d Commands" on page 3-112
"Specifying Intersample Behavior" on page 3-114
"How d2c Handles Input Delays" on page 3-114
"Effects on the Noise Model" on page 3-115

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.

`d2d` is useful is you want to change the sampling interval of a discrete model. All of these operations change the sampling interval, which is called *resampling* the model.

Using the c2d, d2c, and d2d Commands

You can use `c2d` and `d2c` to transform any `idmodel` object between continuous-time and discrete-time representations.

The following table summarizes the commands for transforming between continuous-time and discrete-time model representations. These commands also transform the estimated model uncertainty, which corresponds to the estimated covariance matrix of the parameters. For detailed information about these commands, see the corresponding reference page.

Note `c2d` and `d2d` correctly approximate the transformation of the noise model when the sampling interval `T` is small compared to the bandwidth of the noise.

Command	Description	Usage Example
`c2d`	Converts continuous-time models to discrete-time models.	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 sampling interval of the discrete-time model.
`d2c`	Converts parametric discrete-time models to continuous-time models.	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 sampling interval. 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 sampling interval `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')
```

How d2c Handles Input Delays

The discrete-to-continuous-time conversion d2c properly handles any input delays in the discrete-time model, and stores this information in the continuous-time model. An *input delay* is the delay in the response of the output to the input signal.

The relationship between discrete-time and continuous-time delays depends on the input intersample behavior. For example, a continuous-time system without a delay shows a delay when sampled with a zero-order-hold input.

A delay in the discrete-time model that corresponds to an actual delay in the continuous-time model is stored in the in the `InputDelay` property of the resulting continuous-time model. Typically, this `InputDelay` is `(nk-1)/Ts`, where `nk` is the delay of the discrete-time system and `Ts` is the sampling interval.

Note Unlike for discrete-time models, the `nk` property of continuous-time model is only used to flag when immediate response to step changes is present; `nk` is not used to store input delays greater than or equal to 1. When `nk(i)`=0, then there is an immediate response to a step change in the input `i`th. When `nk(i)`=1, then there is no immediate response to the input.

Effects on the Noise Model

`c2d`, `d2c`, and `d2d` change the sampling interval 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 3-8.

`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 sampling interval and T_{old} is the original sampling interval. before applying `sim`.

`c2d` and `d2c` transformations produce warnings when the continuous-time disturbance model does not have the required white-noise component. These warnings occur because the underlying state-space model, which is formed and used by these transformations, is ill-defined. In this case, modify the *C*-polynomial such that the degree of the monic *C*-polynomial in continuous-time equals the sum of the degrees of the monic *A*- and *D*-polynomials in continuous-time. For example:

```
length(C)-1 = (length(A)-1)+(length(D)-1)
```

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 Tool GUI 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 single- or multiple-output `idmodel` object to `idfrd` model.
	If you have the Control System Toolbox product, this command converts any LTI object.
		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 single-output `idmodel` object to ARMAX representation. If you have the Control System Toolbox product, this command converts any	To get an ARMAX model from state-space model `m_ss`, use the following command: `m_p = idpoly(m_ss)`

Commands for Transforming Model Representations (Continued)

Command	Model Type to Convert	Usage Example
	single-output LTI object except `frd`.	
`idss`	Converts any single- or multiple-output `idmodel` object to state-space representation. If you have the Control System Toolbox product, this command converts any LTI object except `frd`.	To get a state-space model from an ARX model `m_arx`, use the following command: `m_ss = idss(m_arx)`

Note The `idss` conversion produces warnings when the continuous-time disturbance model does not have the required white-noise component. These warnings occur because the underlying state-space model, which is formed and used by these transformations, is ill defined. In this case, modify the *C*-polynomial such that the degree of the monic *C*-polynomial in continuous-time equals the sum of the degrees of the monic *A*- and *D*-polynomials in continuous-time. For example:

```
length(C)-1 = (length(A)-1)+(length(D)-1)
```

Subreferencing Models

In this section...

"What Is Subreferencing?" on page 3-119

"Limitation on Supported Models" on page 3-119

"Subreferencing Specific Measured Channels" on page 3-119

"Subreferencing Measured and Noise Models" on page 3-120

"Treating Noise Channels as Measured Inputs" on page 3-122

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 `idarx`, `idgrey`, `idpoly`, `idproc`, `idss`, and `idfrd` model objects.

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 strings.

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)
```

Subreferencing Measured 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.

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:

$$A(q)y(t) = B(q)u(t - nk) + e(t)$$

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 by subreferencing G due to measured inputs, use the following syntax:

```
m_G = m('measured')
```

Tip Alternatively, you can use the following shorthand syntax: `m_G = m('m')`

To create a new model by subreferencing H due to unmeasured inputs, use the following syntax:

```
m_H = m('noise')
```

Tip Alternatively, you can use the following shorthand syntax: `m_H = m('n')`

This operation creates a time-series model from `m` by ignoring the measured input.

The covariance matrix of e is given by the `idmodel` 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.

Treating Noise Channels as Measured Inputs

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'`.

Note When you plot models in the GUI 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 11-16.

Concatenating Models

In this section...

"About Concatenating Models" on page 3-124

"Limitation on Supported Models" on page 3-124

"Horizontal Concatenation of Model Objects" on page 3-125

"Vertical Concatenation of Model Objects" on page 3-125

"Concatenating Noise Spectrum Data of idfrd Objects" on page 3-126

"See Also" on page 3-127

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 parametric models, such as `idarx`, `idgrey`, `idpoly`, `idproc`, and `idss` model objects, the resulting model combines the parameters of the individual models.

You can also concatenate nonparametric models, which contain the estimated impulse-response (`idarx` 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.

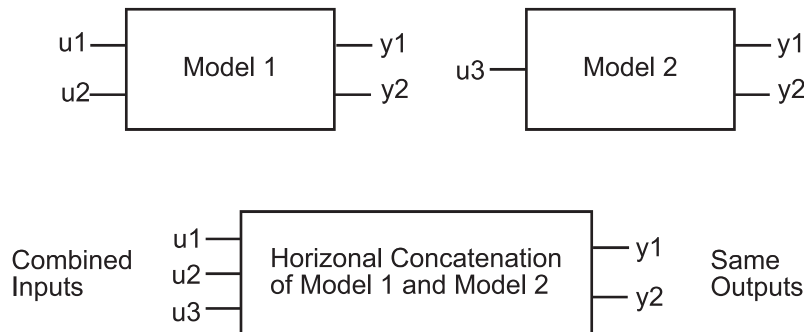
Horizontal Concatenation of Model Objects

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 uses the names of output channels in the first model object you listed. Input channels must have unique names.

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.



Note Horizontal concatenation of `idarx` objects creates an `idss` object.

Vertical Concatenation of Model Objects

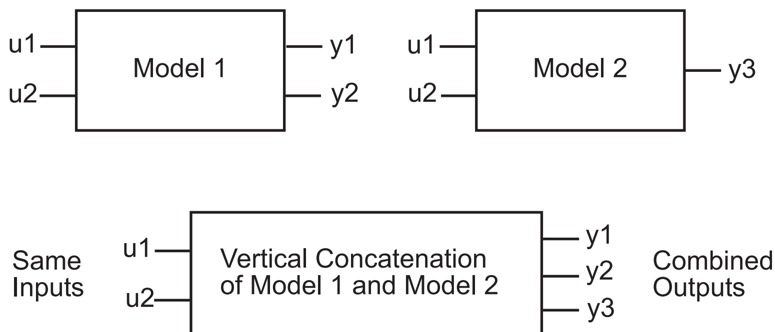
Vertical concatenation combines output channels of specified models. Vertical concatenation of model objects requires that they have the same inputs and frequency vectors. If the input channel names are different and their dimensions are the same, the concatenation operation uses the names of input channels in the first model object you listed. Output channels must have unique names.

Note You cannot concatenate the single-output `idproc` and `idpoly` model objects.

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 the `idfrd` objects contain the frequency-response data you measured or constructed manually, the concatenation operation combines only the `ResponseData` properties. Because noise spectrum data does not exist (unless you also entered it manually), `SpectrumData` is empty in both the individual `idfrd` objects and the concatenated `idfrd` object.

However, when the `idfrd` objects are spectral models that you estimated, 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 9-6 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 `idarx`, `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 "Creating Multiexperiment Data at the Command Line" on page 2-59.

For more information about merging models, see the `merge` reference page.

Nonlinear Black-Box Model Identification

- "About Nonlinear Model Identification" on page 4-2
- "Preparing Data for Nonlinear Identification" on page 4-7
- "Identifying Nonlinear ARX Models" on page 4-8
- "Identifying Hammerstein-Wiener Models" on page 4-49
- "Linear Approximation of Nonlinear Black-Box Models" on page 4-81

About Nonlinear Model Identification

In this section...

"What Are Nonlinear Models?" on page 4-2
"When to Fit Nonlinear Models" on page 4-2
"Available Nonlinear Models" on page 4-4

What Are Nonlinear Models?

Dynamic models in System Identification Toolbox software are mathematical relationships between the system's inputs *u(t)* and outputs *y(t).* You can use these relationships to compute the current output from previous inputs and outputs. The general form of a model in discrete time is:

$$y(t) = f(u(t - 1), y(t - 1), u(t - 2), y(t - 2), . . .)$$

Such a model is nonlinear if the function f is a nonlinear function, which might include nonlinear components representing arbitrary nonlinearities, such as switches and saturations.

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 4-3
- "Physical System Is Weakly Nonlinear" on page 4-3
- "Physical System Is Inherently Nonlinear" on page 4-3
- "Linear and Nonlinear Dynamics Are Captured Separately" on page 4-4

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 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 “Using Linear Model for Nonlinear ARX Estimation” on page 4-28 and “Using Linear Model for Hammerstein-Wiener Estimation” on page 4-64.

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 electro-mechanical 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 4-49.

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 4-8.

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 Chapter 5, “ODE Parameter Estimation (Grey-Box Modeling)”.

Before fitting a nonlinear model, try transforming your input and output variables such that the relationship between the transformed variables is 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 and then fitting 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 "Using Linear Model for Nonlinear ARX Estimation" on page 4-28 and "Using Linear Model for Hammerstein-Wiener Estimation" on page 4-64.

Available Nonlinear Models

System Identification Toolbox supports these nonlinear models:

- "Nonlinear ARX Models" on page 4-4
- "Hammerstein-Wiener Models" on page 4-5
- "Nonlinear State-Space Models" on page 4-5

Nonlinear ARX Models

Nonlinear ARX models extend the linear ARX models 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 used to predict the current output. 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.

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:

- "Tutorial – Identifying Nonlinear Black-Box Models Using the GUI"
- "Identifying Nonlinear ARX Models" on page 4-8

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:

- "Tutorial – Identifying Nonlinear Black-Box Models Using the GUI"
- "Identifying Hammerstein-Wiener Models" on page 4-49

Nonlinear State-Space 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. You use the `idnlgrey` object to specify the structures of such models as nonlinear ODEs, based on physical insight about your system. The parameters of such models typically have physical interpretations.

For more information about estimating nonlinear state-space models, see Chapter 5, “ODE Parameter Estimation (Grey-Box Modeling)”.

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.

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 Tool GUI.** Import data into the GUI, as described in "Importing Data into the GUI" on page 2-17.
- **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 time interval (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.

Identifying Nonlinear ARX Models

In this section...

"Nonlinear ARX Model Extends the Linear ARX Structure" on page 4-8
"Structure of Nonlinear ARX Models" on page 4-9
"Nonlinearity Estimators for Nonlinear ARX Models" on page 4-10
"Ways to Configure Nonlinear ARX Estimation" on page 4-12
"How to Estimate Nonlinear ARX Models in the GUI" on page 4-16
"How to Estimate Nonlinear ARX Models at the Command Line" on page 4-19
"Using Linear Model for Nonlinear ARX Estimation" on page 4-28
"Validating Nonlinear ARX Models" on page 4-35
"Using Nonlinear ARX Models" on page 4-40
"How the Software Computes Nonlinear ARX Model Output" on page 4-41

Nonlinear ARX Model Extends the Linear ARX Structure

A linear SISO ARX model has this structure:

$$
\begin{aligned}
&y(t) + a_1 y(t-1) + a_2 y(t-2) + \ldots + a_{na} y(t-na) = \\
&\qquad b_1 u(t) + b_2 u(t-1) + \ldots + b_{nb} u(t-nb+1) + e(t)
\end{aligned}
$$

where the input delay *nk* is zero to simplify the notation.

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. Rewriting the equation as a product:

$$
\begin{aligned}
&y_p(t) = [-a_1, -a_2, \ldots, -a_{na}, b_1, b_2, .., b_{nb}] * \\
&\qquad [y(t-1), y(t-2), \ldots, y(t-na), u(t), u(t-1), \ldots, u(t-nb-1)]^T
\end{aligned}
$$

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 linear ARX model predicts the current output y_p as a weighted sum of its regressors.

The nonlinear ARX structure is an extension of the linear ARX structure:

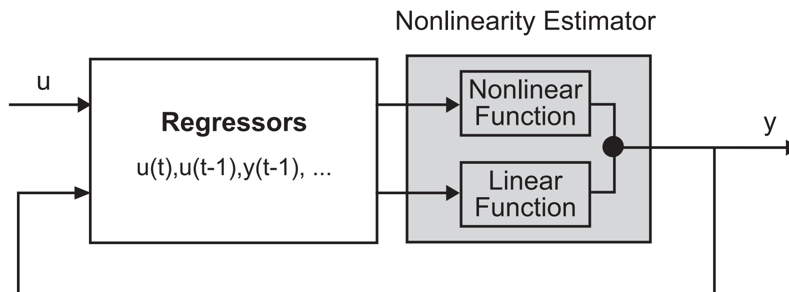
- Instead of the weighted sum that represents a linear mapping, the nonlinear ARX model has a more flexible nonlinear mapping function:

  $$y_p(t) = f(y(t-1), y(t-2), y(t-3), \ldots, u(t), u(t-1), u(t-2), ..)$$

  where *f* is a nonlinear function. Inputs to *f* are model regressors. When you specify the nonlinear ARX model structure, you can choose one of several available nonlinear mapping functions in this toolbox (see "Nonlinearity Estimators for Nonlinear ARX Models" on page 4-10).
- 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 $y(t\text{-}1)^2$, $u(t\text{-}1)*y(t\text{-}2)$, $\tan(u(t\text{-}1))$, and $u(t\text{-}1)*y(t\text{-}3)$.

Structure of Nonlinear ARX Models

This block diagram represents the structure of a nonlinear ARX model:



The nonlinear ARX model computes the output y in two stages:

1. Computes regressors 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)*—called *standard* regressors. You can also specify *custom* regressors, which are nonlinear functions of delayed inputs and outputs. For example, tan($u(t$-1)) or $u(t$-1)*$y(t$-3).

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. The nonlinearity estimator block maps the regressors to the model output using a combination of nonlinear and linear functions. You can select from available nonlinearity estimators, such as tree-partition networks, wavelet networks, and multi-layer neural networks. You can also exclude either the linear or the nonlinear function block from the nonlinearity estimator.

The nonlinearity estimator block can include linear and nonlinear blocks in parallel. For example:

$$F(x) = L^T(x-r) + d + g\left(Q(x-r)\right)$$

x is a vector of the regressors. $L^T(x) + d$ 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. r is the mean of the regressors x. 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.

Estimating a nonlinear ARX model computes the model parameter values, such as L, r, d, Q, and other parameters specifying g. Resulting models are `idnlarx` objects that store all model data, including model regressors and parameters of the nonlinearity estimator. See the `idnlarx` reference page for more information.

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 4-9.

Each nonlinearity estimator corresponds to an object class in this toolbox. When you estimate nonlinear ARX models in the GUI, 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`	$g(x)=\sum_{k=1}^{n}\alpha_k\kappa\left(\beta_k\left(x-\gamma_k\right)\right)$ where $\kappa(s)$ is the wavelet function.	By default, the estimation algorithm determines the number of units *n* automatically.
One layer sigmoid network	`sigmoidnet`	$g(x)=\sum_{k=1}^{n}\alpha_k\kappa\left(\beta_k\left(x-\gamma_k\right)\right)$ 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.

Ways to Configure Nonlinear ARX Estimation

- "Configurable Elements of Nonlinear ARX Structure" on page 4-13
- "Default Nonlinear ARX Structure" on page 4-14
- "Nonlinear ARX Order and Delay" on page 4-14
- "Estimation Algorithm for Nonlinear ARX Models" on page 4-15

Configurable Elements of Nonlinear ARX Structure

You can adjust various elements of the nonlinear ARX model structure and fit different models to your data.

Configure model regressors by:

- Specifying model order and delay, which creates the set of standard regressors.

  For a definition, see "Nonlinear ARX Order and Delay" on page 4-14.

- Creating 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. You can specify custom regressors in addition to or instead of standard regressors for greater flexibility in modeling your data.

- Including a subset of regressors in the nonlinear function of the nonlinear estimator block.

  Selecting which regressors are inputs to the nonlinear function reduces model complexity and keeps the estimation well-conditioned.

- Initializing using a linear ARX model.

  You can perform this operation only at the command line. The initialization configures the nonlinear ARX model to use standard regressors, which the toolbox computes using the orders and delays of the linear model. See "Using Linear Model for Nonlinear ARX Estimation" on page 4-28.

Configure the nonlinearity estimator block by:

- Specifying and configuring the nonlinear function, including the number of units.
- Excluding the nonlinear function from the nonlinear estimator such that

  $F(x) = L^T(x) + d$.

- Excluding the linear function from the nonlinear estimator such that

  $F(x) = g\left(Q(x - r)\right)$.

Note You cannot exclude the linear function from tree partitions and neural networks.

See these topics for detailed steps to change the model structure:

- "Tutorial – Identifying Nonlinear Black-Box Models Using the GUI"
- "How to Estimate Nonlinear ARX Models in the GUI" on page 4-16
- "How to Estimate Nonlinear ARX Models at the Command Line" on page 4-19

Default Nonlinear ARX Structure

Estimate a nonlinear ARX model with default configuration by:

- Specifying only model order and input delay. Specifying the order automatically creates standard regressors.
- Specifying a linear ARX model. The linear model sets the model orders and linear function of the nonlinear model. You can perform this operation only at the command line.

By default:

- The nonlinearity estimator is a wavelet network (see the `wavenet` reference page).

  This nonlinearity often provides satisfactory results and uses a fast estimation method.
- All of the standard regressors are inputs to the linear and nonlinear functions of the wavelet network.

Nonlinear ARX Order and Delay

The order and delay of nonlinear ARX models are positive integers:

- *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 linear ARX model parameters. Orders are scalars for SISO data, and matrices for MIMO data. If you are not sure how to specify the order and delay, you can estimate them as described in "Preliminary Step – Estimating Model Orders and Input Delays" on page 3-48. 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 model orders.

For example, if you specify this order and delay for a SISO model with input *u* and output *y*:

na=2, *nb*=3, and *nk*=5

the toolbox computes standard regressors *y(t-2)*, *y(t-1)*, *u(t-5)*, *u(t-6)*, and *u(t-7)*.

You can specify custom regressors in addition to standard regressors, as described in "How to Estimate Nonlinear ARX Models in the GUI" on page 4-16 and "How to Estimate Nonlinear ARX Models at the Command Line" on page 4-19.

Estimation Algorithm for Nonlinear ARX Models

The estimation algorithm depends on your choice of nonlinearity estimator and other properties of the `idnlarx` class. You can set algorithm properties both in the GUI and at the command line.

Focus property of idnlarx class. By default, estimating nonlinear ARX models minimizes one-step prediction errors, which corresponds to `Focus` value of `Prediction`.

If you want a model that is optimized for reproducing simulation behavior, try setting the `Focus` value to `Simulation`. In this case, you cannot use `treepartition` and `neuralnet` because these nonlinearity estimators are not differentiable. Minimization of simulation error requires differentiable

nonlinear functions. Simulation error minimization takes more time than one-step-ahead prediction error minimization.

Common algorithm properties of idnlarx class.

- `MaxIter` — Maximum number of iterations.
- `SearchMethod` — Search method for minimization of prediction or simulation errors, such as Gauss-Newton and Levenburg-Marquardt line search, and Trust-region reflective Newton approach. By default, the algorithm uses a combination of these methods.
- `Tolerance` — Condition for terminating iterative search when the expected improvement of the parameter values is less than a specified value.
- `Display` — Shows progress of iterative minimization in the MATLAB Command Window.

How to Estimate Nonlinear ARX Models in the GUI

Prerequisites

- Learn about the nonlinear ARX model structure (see “Structure of Nonlinear ARX Models” on page 4-9).
- Import data into the System Identification Tool GUI (see “Preparing Data for Nonlinear Identification” on page 4-7).
- (Optional) Choose a nonlinearity estimator in “Nonlinearity Estimators for Nonlinear ARX Models” on page 4-10.

1. In the System Identification Tool GUI, select **Estimate > Nonlinear models** to open the Nonlinear Models dialog box.

2. In the **Configure** tab, select `Nonlinear ARX` from the **Model type** list (if it is not already selected).

3. (Optional) Edit the **Model name** by clicking the pencil icon. The name of the model should be unique to all nonlinear ARX models in the System Identification Tool GUI.

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. The algorithm uses default estimation settings unless you specify to use the initial model settings, or change these settings.

The **Initial Model** list includes models that:

- Exist in the System Identification Tool GUI.
- Have the same number of inputs and outputs as the dimensions of the estimation data (selected as **Working Data** in the System Identification Tool GUI).

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 GUI 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 **No. of Terms**, and *nb* is the input **No. 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

What to Configure	Options in Nonlinear Models GUI	Comment
		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 Click **Estimate** to add this model to the System Identification Tool GUI.

The **Estimate** tab displays the estimation progress and results.

7 Validate the model response by selecting the desired plot in the **Model Views** area of the System Identification Tool GUI. For more information about validating models, see "How to Plot Nonlinear ARX Plots Using the GUI" on page 4-35.

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 Tool GUI.

How to Estimate Nonlinear ARX Models at the Command Line

Prerequisites

- Learn about the nonlinear ARX model structure in "Structure of Nonlinear ARX Models" on page 4-9.
- Prepare your data, as described in "Preparing Data for Nonlinear Identification" on page 4-7.
- (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 3-48.
- (Optional) Choose a nonlinearity estimator in "Nonlinearity Estimators for Nonlinear ARX Models" on page 4-10.
- (Optional) Estimate or construct an linear ARX model for initialization of nonlinear ARX model. See "Using Linear Model for Nonlinear ARX Estimation" on page 4-28.

Estimate model using nlarx.

Use `nlarx` to both construct and estimate a nonlinear ARX model. After each estimation, validate the model 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:

```
m = nlarx(data,[2 2 1]) % na=nb=2 and nk=1
```

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(data,[2 2 1],'sigmoid')
```

Set the `Focus` property of the `idnlarx` object estimation to simulation error minimization:

```
M = nlarx(data,[2 2 1],'sigmoid','Focus','simulation')
```

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 Order and Delay" on page 4-14).

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
% Generate polynomial regressors up to order 2:
  reg = polyreg(m)
% Append polynomial regressors to CustomRegressors:
  m = addreg(m,reg);
  getreg(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
  % Append polynomial regressors to CustomRegressors:
    m = addreg(m,reg);
    getreg(m) % polynomial regressors
```

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;
```

To view the regressor expression as a string, use:

```
m.CustomReg(1).Display
```

You can exclude all standard regressors and use only custom regressors in the model structure by setting `na=nb=nk=0`:

```
m = nlarx(data,[0 0 0],'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.

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(data,[2 2 1]) % all standard regressors are
                        % inputs to the nonlinear function
```

```
getreg(m) % lists all standard regressors
m = nlarx(data,[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(data,[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(data,[4 4 1],sigmoidnet,'nlreg','search')
```

After estimation, use `m.NonlinearRegressors` to view which regressors were selected by the automatic regressor search. This search typically takes a long time, and you can display the search progress using:

```
m = nlarx(data,[4 4 1],sigmoidnet,'nlreg','search',...
                                  'Display', 'on')
```

Configure the nonlinearity estimator.

Specify the nonlinearity estimator directly in the estimation command as:

- A string of the nonlinearity name, which uses the default nonlinearity configuration.

  ```
  m = nlarx(data, [2 2 1],'sigmoidnet')
  ```

  or

  ```
  m = nlarx(data,[2 2 1],'sig') % abbreviated string
  ```

- Nonlinearity object.

```
m = nlarx(data,[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(data,[2 2 1],w)
```

or

```
w = wavenet;
w.NumberOfUnits = 5;
m = nlarx(data,[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:

```
M = nlarx(data,[na nb nk],[sigmoidnet; wavenet])
```

If you want the same nonlinearity for all output channels, specify one nonlinearity.

This 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)
```

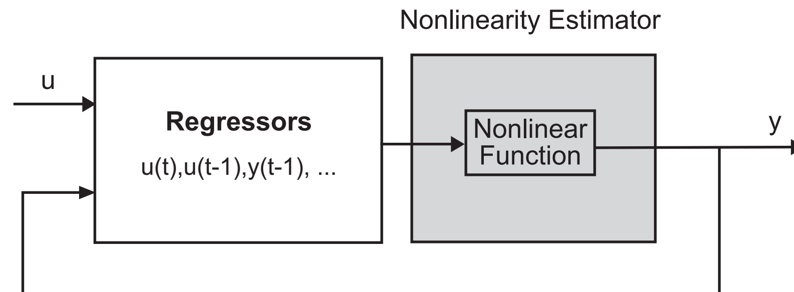
where `NNet` is the neural network object you create using the Neural Network Toolbox software. See the `neuralnet` reference page.

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 = cutomnet(@gaussunit);
m = nlarx(data,[na nb nk],CNetw)
```

Include only nonlinear 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 $F(x) = g(Q(x-r))$.



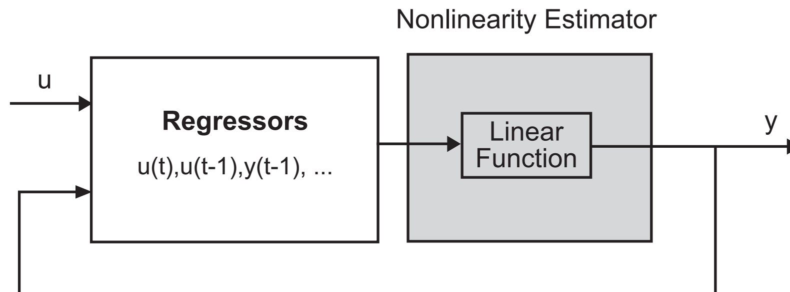
For example:

```
SNL = sigmoidnet('LinearTerm','off'')
m = nlarx(data,[2 2 1],SNL);
```

Note You cannot exclude the linear function from tree partition and neural network nonlinearity estimators.

Include only linear 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(data,[na nb nk],'linear');
```

is similar to the linear ARX model:

```
lin_m = arx(data,[na nb nk]);
```

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 as system, use:

```
M = nlarx(data,[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 the model.

If your model structure includes nonlinearities that support iterative search (see "Estimation Algorithm for Nonlinear ARX Models" on page 4-15), you can use pem to refine model parameters:

```
m = nlarx(data,[2 2 1],'sigmoidnet')
m2 = pem(data,m)
```

You can also use nlarx to refine the original model:

```
m1 = nlarx(data, [2 2 1],'sigmoidnet','wavenet');
m2 = nlarx(data,m1) % can repeatedly run this command
```

Check the search termination criterion m.EstimationInfo.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 idnlarx property:

```
m2 = pem(data,m1,'MaxIter',30) % runs 30 more iterations
                               % starting from m1
```

When the m.EstimationInfo.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 Algorithm.Tolerance property value of the idnlarx class or the Algorithm.SearchMethod property, and repeat the estimation. You can also try perturbing the parameters of the last model using init (called *randomization*) and refining the model using pem:

```
M1 = nlarx(data, [2 2 1], `sigm'); % original model
M1p = init(M1); % randomly perturbs parameters about nominal values
M2 = pem(data, M1p);  % estimates parameters of perturbed model
```

You can display the progress of the iterative search in the MATLAB Command Window using the `Display` property of the `idnlarx` class:

```
M2= pem(data,M1p,'Display','On')
```

What if you cannot get a satisfactory model?

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.

Example – Using nlarx to Estimate Nonlinear ARX Models

Use `nlarx` to estimate a nonlinear ARX model for the data in the "Tutorial – Identifying Nonlinear Black-Box Models Using the GUI".

1. Prepare the data for estimation:

   ```
   load twotankdata
   z = iddata(y, u, 0.2);
   ze = z(1:1000); zv = z(1001:3000);
   ```

2. Estimate several models using different model orders, delays, and nonlinearity settings:

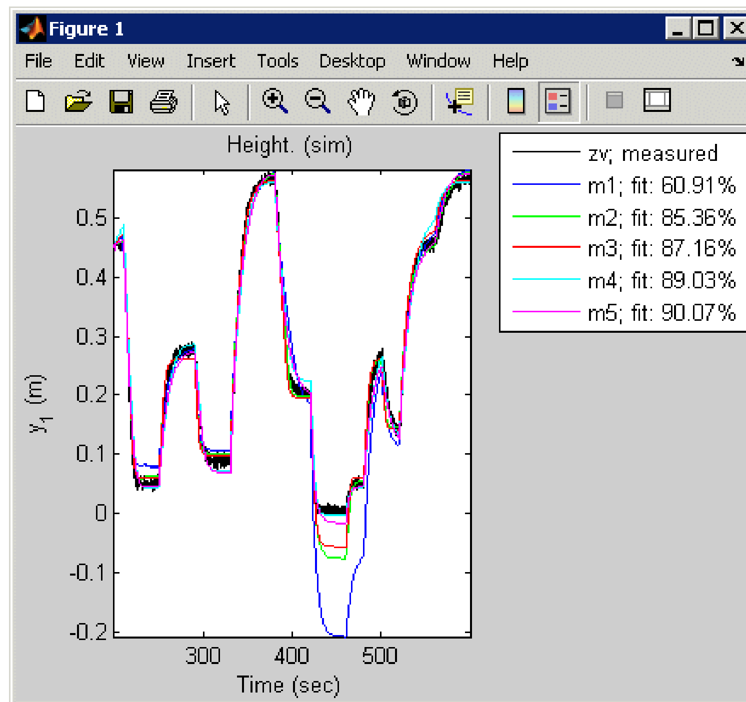
   ```
   m1 = nlarx(ze,[2 2 1]);
   m2 = nlarx(ze,[2 2 3]);
   m3 = nlarx(ze,[2 2 3],wavenet('num',8));
   m4 = nlarx(ze,[2 2 3],wavenet('num',8),...
                         'nlr', [1 2]);
   ```

   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 = pem(ze,m5);
   ```

3 Compare the resulting models by plotting the model outputs with the measured output:

```
compare(zv, m1,m2,m3,m4,m5)
```



The following examples

Using Linear Model for Nonlinear ARX Estimation

- “About Using Linear Models” on page 4-29
- “How to Initialize Nonlinear ARX Estimation Using Linear ARX Models” on page 4-30
- “Example – Using Linear ARX Models to Estimate Nonlinear ARX Models” on page 4-30

About Using Linear Models

You can use the following discrete-time linear models for nonlinear ARX estimation. The linear model must sufficiently represent the linear dynamics of your system.

Discrete-Time Linear Model	Use for Initializing...
Single-output polynomial model of ARX structure (`idpoly`)	Single-output nonlinear ARX model estimation
Multi-output polynomial model of ARX structure (`idarx`)	Multi-output nonlinear ARX model estimation

---

Tip To learn more about when to use linear models, see "When to Fit Nonlinear Models" on page 4-2.

Typically, you create a linear ARX model using the `arx` command. You can provide the linear model only at the command line when constructing (see `idnlarx`) or estimating (see `nlarx`) a nonlinear ARX model.

The software uses the linear model for initializing the nonlinear ARX estimation:

- Assigns the linear model orders as initial values of nonlinear model orders (`na` and `nb` properties of the `idnlarx` object) and delays (`nk` property) to compute standard regressors in the nonlinear ARX model structure.
- Uses 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 for neural network nonlinearity estimator.

During estimation, the estimation algorithm uses these values to further adjust the nonlinear model to the data. The initialization always provides a better fit to the estimation data than the linear ARX model.

How to Initialize Nonlinear ARX Estimation Using Linear ARX Models

Estimate a nonlinear ARX model initialized using a linear model by typing

```
m = nlarx(data,LinARXModel)
```

LinARXModel is an idpoly object of ARX structure for single-output models, and idarx object for multi-output models. *m* is an idnlarx object. *data* is a time-domain iddata object.

By default, the nonlinearity estimator is the wavelet network (wavenet object). This network takes all standard regressors computed using orders and delay of *LinARXModel* as inputs to its linear and nonlinear functions. The software computes the LinearCoef parameter of the wavenet object using the A and B polynomials of the linear ARX model.

Tip When you use the same data set, a nonlinear ARX model initialized using a linear ARX model produces a better fit than the linear ARX model.

Specify a different nonlinearity estimator, for example a sigmoid network:

```
m = nlarx(data,LinARXModel,'sigmoid')
```

Set the Focus property of the idnlarx object estimation to simulation error minimization:

```
m = nlarx(data,LinARXModel,'sigmoid','Focus','simulation')
```

After each estimation, validate the model by comparing the simulated response to the data. To improve the fit of the nonlinear ARX model, adjust various elements of the nonlinear ARX structure. For more information, see "Ways to Configure Nonlinear ARX Estimation" on page 4-12.

Example – Using Linear ARX Models to Estimate Nonlinear ARX Models

1 Load the estimation data.

```
load throttledata.mat
```

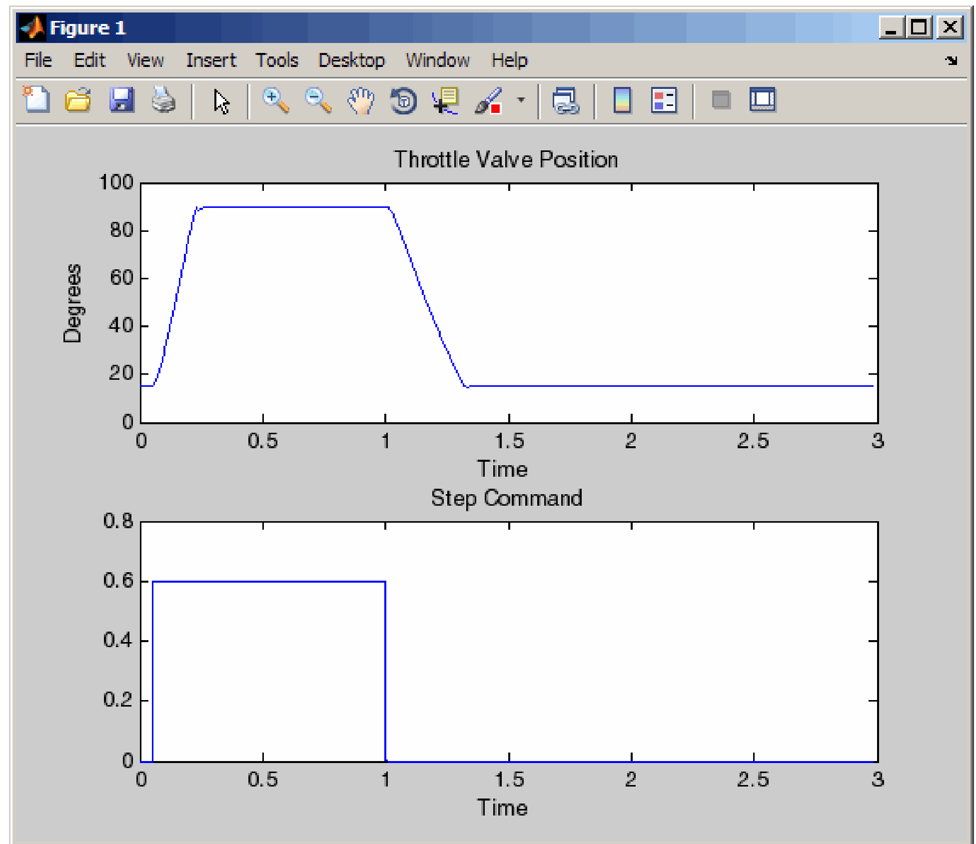
This command loads the data object `ThrottleData` into the MATLAB workspace. The object contains input and output samples collected from an engine throttle system, sampled at a rate of 100 Hz.

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.

2 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, as shown in the following figure. 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.

3 Estimate an ARX model to model the linear behavior of this single-input single-output system in the normal operating range.

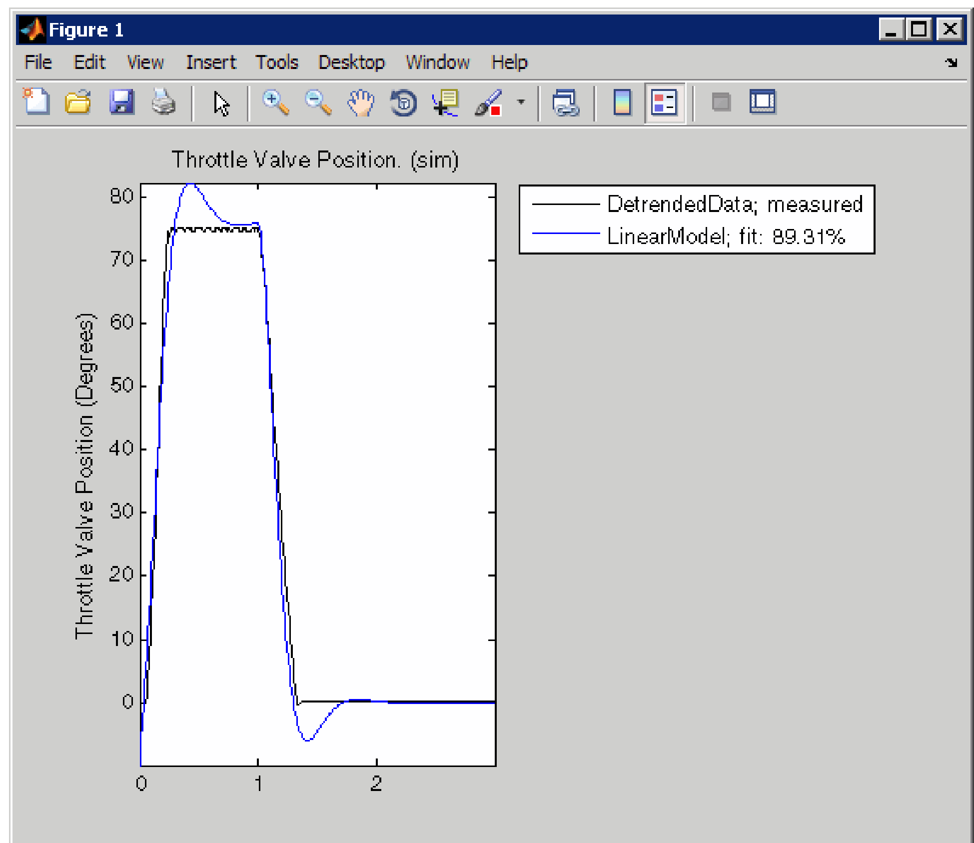
```
% Detrend the data because linear models cannot capture offsets.
Tr = getTrend(ThrottleData);
Tr.OutputOffset = 15;
DetrendedData = detrend(ThrottleData, Tr);
```

```
% Estimate a linear ARX model with na=2, nb=1, nk=1.
LinearModel = arx(DetrendedData, [2 1 1], 'Focus', 'Simulation');
```

4 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, as shown in the next figure.



5 Estimate a nonlinear ARX model to model the output saturation.

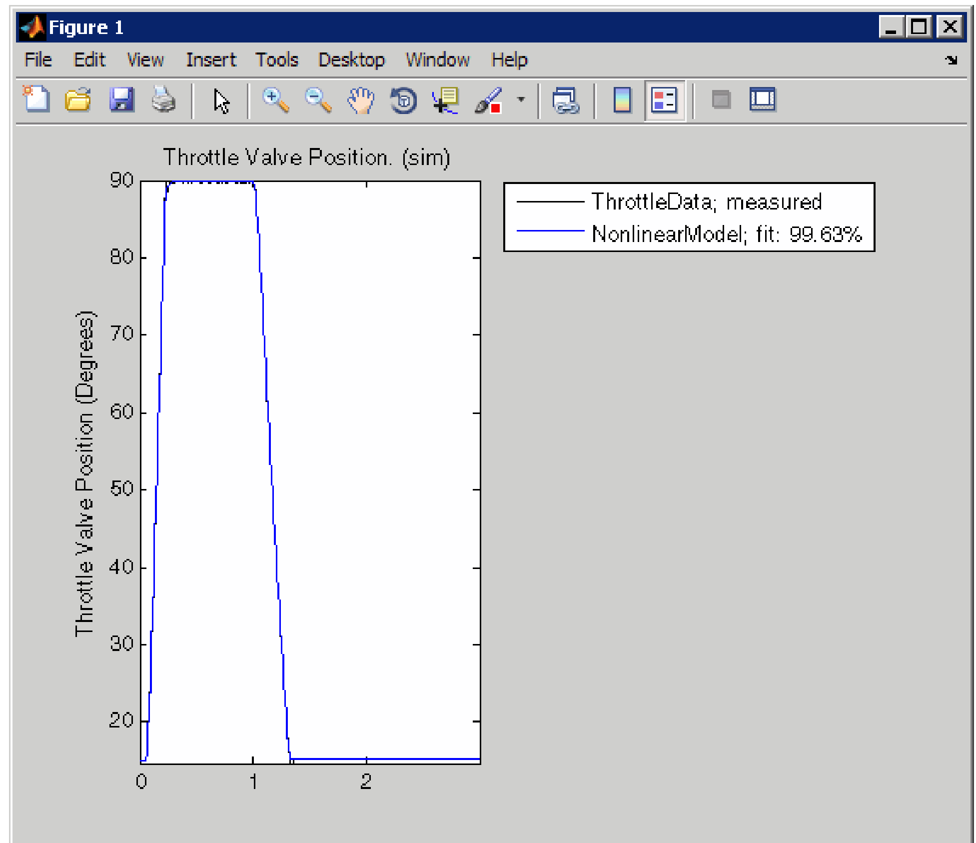
```
NonlinearModel = nlarx(ThrottleData, LinearModel, 'sigmoidnet',...
 'Focus', 'Simulation');
```

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.

6 Compare the nonlinear model with data.

```
compare(ThrottleData, NonlinearModel)
```

The model captures the nonlinear effects (output saturation) and improves the overall fit to data, as shown in the next figure.



Validating Nonlinear ARX Models

- "About Nonlinear ARX Plots" on page 4-35
- "How to Plot Nonlinear ARX Plots Using the GUI" on page 4-35
- "How to Validate Nonlinear ARX Models at the Command Line" on page 4-36
- "Configuring the Nonlinear ARX Plot" on page 4-38
- "Axis Limits, Legend, and 3-D Rotation" on page 4-39

About Nonlinear ARX Plots

The Nonlinear ARX plot displays the characteristics of model nonlinearities as a function of one or two regressors. For more information about estimating nonlinear ARX models, see "Identifying Nonlinear ARX Models" on page 4-8.

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 should be included in the nonlinear function.

Furthermore, you can create several nonlinear models for the same data set using different nonlinearity estimators, such a wavelet network and tree partition, and then compare the nonlinear surfaces of these models. Agreement between nonlinear surfaces increases the confidence that these nonlinear models capture the true dynamics of the system.

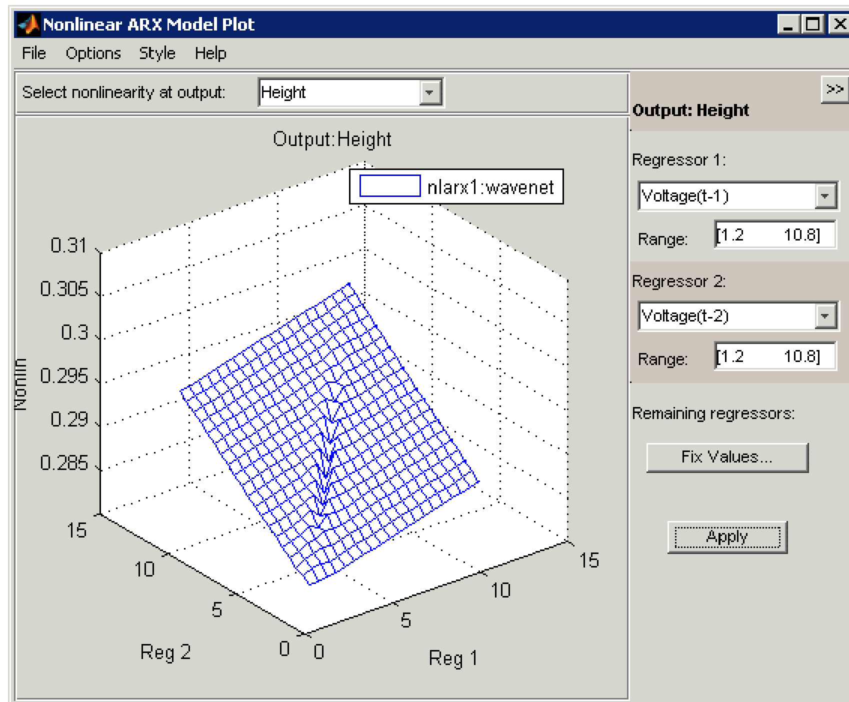
How to Plot Nonlinear ARX Plots Using the GUI

You can plot linear and nonlinear blocks of nonlinear ARX models.

To create a nonlinear ARX plot in the System Identification Tool GUI, select the **Nonlinear ARX** check box in the **Model Views** area. For general information about creating and working with plots, see "Working with Plots" on page 11-13.

Note The **Nonlinear ARX** check box is unavailable if you do not have a nonlinear ARX model in the Model Board.

The following figure shows a sample nonlinear ARX plot.



How to Validate Nonlinear ARX Models at the Command Line

You can use the following approaches to validate nonlinear ARX models at the command line:

Compare Model Output to Measured Output

Compare estimated models using `compare`. Use an independent validation data set whenever possible. For more information about validating models, see "Validating Models After Estimation" on page 8-3.

For example, compare linear and nonlinear ARX models of same order:

```
load iddata1
LM = arx(z1,[2 2 1]) % estimates linear ARX model
M = nlarx(z1,[2 2 1],'sigmoidnet') % estimates nonlinear ARX model
compare(z1,LM,M) % compares responses of LM and M
                 % against measured data
```

Compare the performance of several models using the properties `M.EstimationInfo.FPE` (final prediction error) and `M.EstimationInfo.LossFcn` (value of loss function at estimation termination). Smaller values typically indicate better performance. However, `m.EstimationInfo.FPE` values might be unreliable when the model contains a large number of parameters relative to the estimation data size.

Simulate and Predict Model Response

Use `sim(idnlarx)` and `predict(idnlarx)` to simulate and predict model response, respectively. To compute the step response of the model, use `step`. See the corresponding reference page for more information.

Analyze Residuals

Residuals are differences between the one-step-ahead predicted output from the model and the measured output from the validation data set. Thus, residuals represent the portion of the validation data output not explained by the model. Use `resid` to compute and plot the residuals.

Plot Nonlinearity

Use `plot` to view the shape of the nonlinearity. For example:

```
plot(M)
```

where `M` is the nonlinear ARX (`idnlarx`) model. The plot command opens the Nonlinear ARX Model Plot window. For more information about working with this plot window, see “Configuring the Nonlinear ARX Plot” on page 4-38 and “Axis Limits, Legend, and 3-D Rotation” on page 4-39.

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 by specifying the `M.NonlinearRegressors` property and repeat the estimation.

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.

For detailed information about `plot`, type the following command at the prompt:

```
help idnlarx/plot
```

Check Iterative Search Termination Conditions

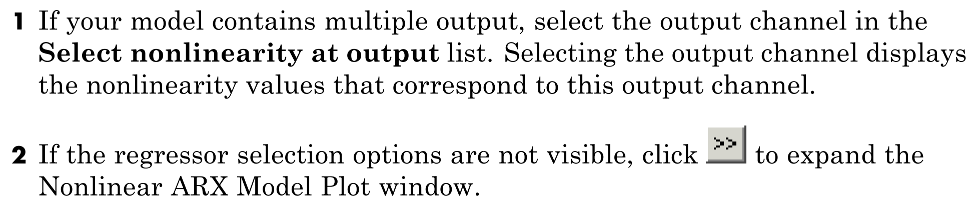
If your `idnlarx` model structure uses iterative search to minimize prediction or simulation errors, use `M.EstimationInfo` to display the estimation termination conditions. `M` is the estimated `idnlarx` model. For example, check the `WhyStop` field of the `EstimationInfo` property, which describes why the estimation stopped—the algorithm might have reached the maximum number of iterations or the required tolerance value. For more information about iterative search, see “Estimation Algorithm for Nonlinear ARX Models” on page 4-15.

Configuring the Nonlinear ARX Plot

To include or exclude a model on the plot, click the corresponding model icon in the System Identification Tool GUI. Active models display a thick line inside the Model Board icon.

To configure the plot:

1. If your model contains multiple output, select the output channel in the **Select nonlinearity at output** list. Selecting the output channel displays the nonlinearity values that correspond to this output channel.

2. If the regressor selection options are not visible, click  to expand the Nonlinear ARX Model Plot window.

3. 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.

4. Specify a second regressor for a 3-D plot by selecting one of the following types of options:

   - Select **Regressor 2** to display three axes. 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.

   - Select `<none>` in the **Regressor 2** list to display only two axes.

5. To fix the values of the regressor that are not displayed, click **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**.

6. In the Nonlinear ARX Model Plot window, click **Apply** to update the plot.

7. To change the grid of the regressor space along each axis, **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.

Axis Limits, Legend, and 3-D Rotation

The following table summarizes the commands to modify the appearance of the Nonlinear ARX plot.

Changing Appearance of the Nonlinear ARX Plot

Action	Command
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 legend.	Select **Style > Legend**. Select this option again to show the legend.
(Three axes only) Rotate in three dimensions. **Note** Available only when you have selected two regressors as independent variables.	Select **Style > 3D Rotate** and drag the axes on the plot to a new orientation. To disable three-dimensional rotation, select **Style > 3D Rotate** again.

Using Nonlinear ARX Models

Simulation and Prediction

Use `sim(idnlarx)` to simulate the model output, and `predict(idnlarx)` to predict the model output. To compare models to measured output and to each other, use `compare`.

Simulation and prediction commands provide default handling of the model's initial conditions, or initial state values. See the `idnlarx` reference page for a definition of the nonlinear ARX model states.

This toolbox provides several options to facilitate how you specify initial states. For example, you can use `findstates(idnlarx)` and `data2state(idnlarx)` to compute state values based on operating conditions or the requirement to maximize fit to measured output.

To learn more about how `sim` and `predict` compute the model output, see "How the Software Computes Nonlinear ARX Model Output" on page 4-41.

Linearization

Compute linear approximation of nonlinear ARX models using `linearize(idnlarx)` or `linapp`.

`linearize` provides a first-order Taylor series approximation of the system about an operation point (also called *tangent linearization*). `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 4-81.

You can compute the operating point for linearization using `findop(idnlarx)`.

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 Chapter 9, "Control Design Applications".

Simulation and Code Generation Using Simulink

You can import estimated Nonlinear ARX models into Simulink software using the Nonlinear ARX block 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 Nonlinear ARX 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 about working in the Simulink environment, see Chapter 10, "System Identification Toolbox Blocks".

How the Software Computes Nonlinear ARX Model Output

In most applications, `sim(idnlarx)` and `predict(idnlarx)` are sufficient for computing the simulated and predicted model response, respectively. This advanced topic describes how the software evaluates the output of nonlinearity estimators and uses this output to compute the model response.

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 = get(m,'Nonlinearity') % equivalent to F = m.nl
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])
```

Example – Low-level Simulation and Prediction of Sigmoid Network

This example shows how the software computes the simulated and predicted output of the model as a result of evaluating the output of its nonlinearity estimator for given regressor values.

Estimating and Exploring a Nonlinear ARX Model

1. 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');
   ```

2. Inspect the model properties and estimation result:

   ```
   present(M)
   ```

   which provides information about input and output variables, regressors, and nonlinearity estimator:

   ```
   Input name: u1
   ```

```
Output name: y1
Standard regressors corresponding to the orders
  na = 1, nb = 1, nk = 0
No custom regressor
Nonlinear regressors:
  y1(t-1)
  u1(t)
Nonlinearity estimator: sigmoidnet with 10 units
```

3 Inspect the nonlinearity estimator:

```
NL = M.Nonlinearity % equivalent to M.nl
class(NL)   % nonlinearity class
display(NL) % equivalent to NL
```

Inspect the sigmoid network parameter values:

```
NL.Parameters
```

Prediction of Output

The model output is:

$$y1(t)=f(y1(t\text{-}1),u1(t))$$

where f is the sigmoid network function. The model regressors $y1(t\text{-}1)$ and $u1(t)$ are inputs to the nonlinearity estimator. Time t is a discrete variable representing kT, where $k = 0, 1,\ldots,$ and T is the sampling interval. In this example, T=`0.2` second.

The output prediction equation is:

$$yp(t)=f(y1_meas(t\text{-}1),u1_meas(t))$$

where $yp(t)$ is the predicted value of the response at time t. $y1_meas(t\text{-}1)$ 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:

1. 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;
   ```

2. 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))`.

3. 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

   - $y1_meas(t{=}{-}1)$ 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)`.

4. Compute the predicted output at time t=1.

   ```
   RegValue = [estData.y(1),estData.u(2)];
   yp_1 = evaluate(NL,RegValue)
   ```

   The predicted output is $yp(t{=}1){=}f(y1_meas(t{=}0),u1_meas(t{=}1))$. In terms of MATLAB variables, this output is *f*(`estData.y(1),estData.u(2)`), 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)`.

5 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(idnlarx)`:

```
yp = predict(M,estData,1,'InitialState',x0)
```

Simulation of Output

The model output is:

$$y1(t)=f(y1(t\text{-}1),u1(t))$$

where f is the sigmoid network function. The model regressors $y1(t\text{-}1)$ and $u1(t)$ are inputs to the nonlinearity estimator. Time t is a discrete variable representing kT, where $k = 0, 1,\ldots,$ and T is the sampling interval. In this example, T=`0.2` second.

The simulated output is:

$$ys(t)=f(ys(t\text{-}1),u1_meas(t))$$

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 $ys(t\text{-}1)$ 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:

1. 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
   ```

2. 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))`.

3. 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)`.

4. 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)`.

5 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)
```

Low-Level Nonlinearity Evaluation

This examples performs a low-level computation of the nonlinearity response for the `sigmoidnet` network function:

$$F(x) = (x-r)PL + a_1 f\left((x-r)Qb_1 + c_1\right) + \ldots + a_n f\left((x-r)Qb_n + c_n\right) + d$$

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, b_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
% Total response y = F(x) = yLinear + yNonlinear
  y = yLinear + yNonlinear; % y is equal to evaluate(NL,x)
```

Identifying Hammerstein-Wiener Models

In this section...

"Applications of Hammerstein-Wiener Models" on page 4-49
"Structure of Hammerstein-Wiener Models" on page 4-50
"Nonlinearity Estimators for Hammerstein-Wiener Models" on page 4-52
"Ways to Configure Hammerstein-Wiener Estimation" on page 4-53
"Estimation Algorithm for Hammerstein-Wiener Models" on page 4-55
"How to Estimate Hammerstein-Wiener Models in the GUI" on page 4-55
"How to Estimate Hammerstein-Wiener Models at the Command Line" on page 4-58
"Using Linear Model for Hammerstein-Wiener Estimation" on page 4-64
"Validating Hammerstein-Wiener Models" on page 4-70
"Using Hammerstein-Wiener Models" on page 4-76
"How the Software Computes Hammerstein-Wiener Model Output" on page 4-78

Applications of Hammerstein-Wiener Models

If the output of a system depends nonlinearly on its inputs, it might be 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.

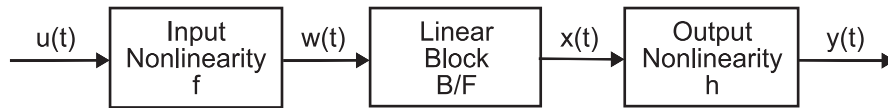
Hammerstein-Wiener model applications span several areas, such as modeling electro-mechanical system and radio frequency components, audio and speech processing and predictive control of chemical processes. These models are popular because they have a convenient block representation, transparent relationship to linear systems, and are easier to implement than heavy-duty nonlinear models (such as neural networks and Volterra models).

You can use the Hammerstein-Wiener model as a black-box model structure because it provides a flexible parameterization for nonlinear models. For example, you might estimate a linear model and try to improve its quality 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 might represent typical physical transformations in actuators and the output nonlinearity might describe common sensor characteristics.

Structure of Hammerstein-Wiener Models

This block diagram represents the structure of a Hammerstein-Wiener model:



where:

- $w(t) = f(u(t))$ is a nonlinear function transforming input data $u(t)$. $w(t)$ has the same dimension as $u(t)$.
- $x(t) = (B/F)w(t)$ is a linear transfer function. $x(t)$ has the same dimension as $y(t)$.

  where B and F are similar to polynomials in the linear Output-Error model, as described in "What Are Black-Box Polynomial Models?" on page 3-39.

  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 $j = 1,2,...,ny$ and $i = 1,2,...,nu$.

- $y(t) = h(x(t))$ is a nonlinear function that maps the output of the linear block to the system output.

w(t) and *x(t)* are internal variables that define the input and output of the linear block, respectively.

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 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 nonlinearities f and h are scalar functions, one nonlinear function for each input and output channel.

The Hammerstein-Wiener model computes the output y in three stages:

1. Computes $w(t) = f(u(t))$ 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. Computes 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 numerator B and denominator F orders.

3. Compute the model output by transforming the output of the linear block $x(t)$ using the nonlinear function h: $y(t) = h(x(t))$.

Similar to the input nonlinearity, the output nonlinearity is a static function. 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 estimator. See the `idnlhw` reference page for more information.

Nonlinearity Estimators for Hammerstein-Wiener Models

System Identification Toolbox software provides several scalar nonlinearity estimators $F(x)$ for Hammerstein-Wiener models. The nonlinearity estimators are available for both the input and output nonlinearities f and h, respectively. For more information about $F(x)$, see "Structure of Hammerstein-Wiener Models" on page 4-50.

Each nonlinearity estimator corresponds to an object class in this toolbox. When you estimate Hammerstein-Wiener models in the GUI, System Identification 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.

Nonlinearity	Class	Structure	Comments
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.
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.

Ways to Configure Hammerstein-Wiener Estimation

Estimate a Hammerstein-Wiener model with default configuration by:

- Specifying model order and input delay:
  - *nb*—The number of zeros plus one.
  - *nf*—The number of poles.
  - *nk*—The delay from input to the output in terms of the number of samples.

  nb is the order of the transfer function numerator (*B* polynomial), and *nf* is the order of the transfer function denominator (*F* polynomial). As you fit different Hammerstein-Wiener models to your data, you can configure the linear block structure by specifying a different order and delay. For MIMO systems with *ny* outputs and *nu* inputs, *nb*, *nf*, and *nk* are *ny*-by-*nu* matrices.

- Initializing using one of the following discrete-time linear models:
  - An input-output polynomial model of Output-Error (OE) structure (`idpoly`)
  - A linear state-space model with no disturbance component (`idss` object with K=0)

  You can perform this operation only at the command line. The initialization configures the Hammerstein-Wiener model to use orders and delay of the linear model, and the *B* and *F* polynomials as the transfer function numerator and denominator. See "Using Linear Model for Hammerstein-Wiener Estimation" on page 4-64.

By default, the input and output nonlinearity estimators are both piecewise linear functions, parameterized by breakpoint locations (see the `pwlinear` reference page). You can configure the input and output nonlinearity estimators by:

- Configuring the input and output nonlinearity properties.
- Excluding the input or output nonlinear block.

See these topics for detailed steps to change the model structure:

- "Tutorial – Identifying Nonlinear Black-Box Models Using the GUI"
- "How to Estimate Hammerstein-Wiener Models in the GUI" on page 4-55

- "How to Estimate Hammerstein-Wiener Models at the Command Line" on page 4-58

Estimation Algorithm for Hammerstein-Wiener Models

Estimation of Hammerstein-Wiener models uses iterative search to minimize the simulation error between the model output and the measured output.

You can configure the estimation method using the `Algorithm` properties of the `idnlhw` class. The most common of these properties are:

- `MaxIter` — Maximum number of iterations.
- `SearchMethod` — Search method for minimization of prediction or simulation errors, such as Gauss-Newton and Levenburg-Marquardt line search, and Trust-region reflective Newton approach.
- `Tolerance` — Condition for terminating iterative search when the expected improvement of the parameter values is less than a specified value.
- `Display` — Shows progress of iterative minimization in the MATLAB Command Window.

By default, the initial states of the model are zero and not estimated. However, you can choose to estimate initial states during model estimation, which sometimes helps to achieve better results.

How to Estimate Hammerstein-Wiener Models in the GUI

Prerequisites

- Learn about the Hammerstein-Wiener model structure (see "Structure of Hammerstein-Wiener Models" on page 4-50).
- Import data into the System Identification Tool GUI (see "Preparing Data for Nonlinear Identification" on page 4-7).
- (Optional) Choose a nonlinearity estimator in "Nonlinearity Estimators for Hammerstein-Wiener Models" on page 4-52.

- (Optional) Estimate or construct an OE or state-space model to use for initialization. See "Using Linear Model for Hammerstein-Wiener Estimation" on page 4-64.

1. In the System Identification Tool GUI, 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 Tool GUI.

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 Tool GUI.
   - Have the same number of inputs and outputs as the dimensions of the estimation data (selected as **Working Data** in the System Identification Tool GUI).

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 GUI 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 **Algorithm Options**.	You can specify to estimate initial states.

6 Click **Estimate** to add this model to the System Identification Tool GUI.

The **Estimate** tab displays the estimation progress and results.

7 Validate the model response by selecting the desired plot in the **Model Views** area of the System Identification Tool GUI. For more information about validating models, see Chapter 8, "Model Analysis".

   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 Tool GUI.

How to Estimate Hammerstein-Wiener Models at the Command Line

Prerequisites

- Learn about the Hammerstein-Wiener model structure described in "Structure of Hammerstein-Wiener Models" on page 4-50.
- Prepare your data, as described in "Preparing Data for Nonlinear Identification" on page 4-7.
- (Optional) Choose a nonlinearity estimator in "Nonlinearity Estimators for Hammerstein-Wiener Models" on page 4-52.
- (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 "Using Linear Model for Hammerstein-Wiener Estimation" on page 4-64.

Estimate model using nlhw.

Use `nlhw` to both construct and estimate a Hammerstein-Wiener model. After each estimation, validate the model 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:

```
m = nlhw(data,[2 2 1]) % nb=nf=2 and nk=1
```

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 the nonlinearity estimator.

Specify a different nonlinearity estimator using *m* = nlhw(*data*,[*nb nf nk*],*InputNL*,*OutputNL*). *InputNL* and *OutputNL* are nonlinearity estimator objects.

Note If your input signal is binary, set *InputNL* to unitgain.

To use nonlinearity estimators with default settings, specify *InputNL* and *OutputNL* using strings (such as 'wave' for wavelet network or 'sig' for sigmoid network).

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(data,[2 2 1],'saturation',poly1d('Degree',3))
```

The third input 'saturation' is a string representation of 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'` or `'pwlin'`	`pwlinear`
One layer sigmoid network	`'sigmoidnet'` or `'sigm'`	`sigmoidnet`
Wavelet network	`'wavenet'` or `'wave'`	`wavenet`
Saturation	`'saturation'` or `'sat'`	`saturation`
Dead zone	`'deadzone'` or `'dead'`	`deadzone`
One-dimensional polynomial	`'poly1d'` or `'poly'`	`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:

```
CNetw = cutomnet(@gaussunit);
m = nlhw(data,[na nb nk],CNetw)
```

Exclude the 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 "Nonlinearity Estimators for Hammerstein-Wiener Models" on page 4-52.

Iteratively refine the model.

Use `pem` to refine the original model. For example:

```
m1 = nlhw(data, [2 2 1],'sigmoidnet','wavenet');
m2 = pem(data,m1) % can repeatedly run this command
```

You can also use `nlhw` to refine the original model:

```
m1 = nlhw(data, [2 2 1],'sigmoidnet','wavenet');
m2 = nlhw(data,m1) % can repeatedly run this command
```

Check the search termination criterion in `m.EstimationInfo.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` `idnlhw` property:

```
m2 = pem(data,m1,'MaxIter',30) % runs 30 more iterations starting at m1
```

When the `m.EstimationInfo.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 `Algorithm.Tolerance` property value of the `idnlhw` class or the `Algorithm.SearchMethod` property, and repeat the estimation. You can also try perturbing the parameters of the last model using `init` (called *randomization*) and refining the model using `pem`:

```
M1 = nlhw(data, [2 2 1], `sigm','wave'); % original model
M1p = init(M1); % randomly perturbs parameters about nominal values
M2 = pem(data, M1p);  % estimates parameters of perturbed model
```

You can display the progress of the iterative search in the MATLAB Command Window using the `Display` property of the `idnlhw` class:

```
M2= pem(data,M1p,'Display','On')
```

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, use:

```
m = nlhw(data,[nb nf nk],[sigmoidnet;pwlinear],[],...
                         'InitialState','e');
```

What if you cannot get a satisfactory model?

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.

Example – Using nlhw to Estimate Hammerstein-Wiener Models

Use `nlhw` to estimate a Hammerstein-Wiener model for the data in "Tutorial – Identifying Nonlinear Black-Box Models Using the GUI".

1 Prepare the data for estimation:

```
load twotankdata
z = iddata(y, u, 0.2);
ze = z(1:1000); zv = z(1001:3000);
```

2 Estimate several models using different model orders, delays, and nonlinearity settings:

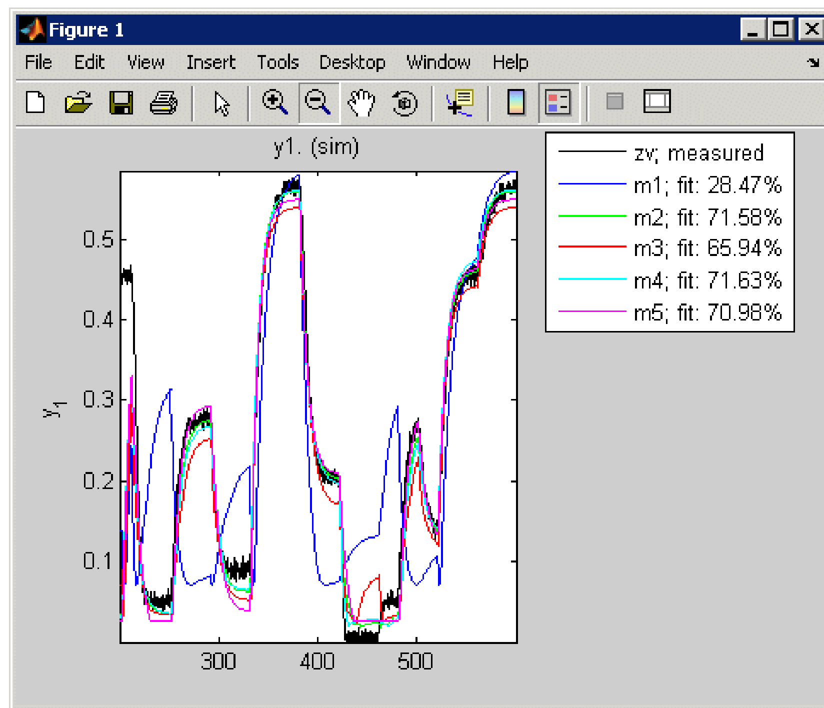
```
m1 = nlhw(ze,[2 3 1]);
m2 = nlhw(ze,[2 2 3]);
m3 = nlhw(ze,[2 2 3], pwlinear('num',13),...
                      pwlinear('num',10));
m4 = nlhw(ze,[2 2 3], sigmoidnet('num',2),...
                      pwlinear('num',10));
```

An alternative way to perform the estimation is to configure the model structure first, and then to estimate this model:

```
m5 = idnlhw([2 2 3], 'dead','sat')
m5 = pem(ze,m5);
```

3 Compare the resulting models by plotting the model outputs on top of the measured output:

```
compare(zv,m1,m2,m3,m4,m5)
```



Example – Improving a Linear Model Using Hammerstein-Wiener Structure

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.

1 Estimate a linear model:

```
load iddata1
LM = arx(z1,[2 2 1]);
```

2 Extract the transfer function coefficients from the linear model:

```
[Num, Den] = tfdata(LM);
```

3 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
               % nu is number of inputs
nf = nb;
nk = 0;        % In general, nk = zeros(ny,nu)
               % ny is number of outputs
               % nu is number of inputs
M = idnlhw([nb nf nk],'poly1d','pwlinear');
M.b = Num;
M.f = Den;
```

4 Estimate the model coefficients, which refines the linear model coefficients in `Num` and `Den`:

```
M = pem(z1, M);
```

5 Compare responses of linear and nonlinear model against measured data:

```
compare(z1,LM,M)
```

Using Linear Model for Hammerstein-Wiener Estimation

- "About Using Linear Models" on page 4-65
- "How to Initialize Hammerstein-Wiener Estimation Using Linear Polynomial Output-Error or State-Space Models" on page 4-66
- "Example – Using Linear OE Models to Estimate Hammerstein-Wiener Models" on page 4-66

About Using Linear Models

You can use the following discrete-time linear models for Hammerstein-Wiener estimation. The linear model must sufficiently represent the linear dynamics of your system.

Discrete-Time Linear Model	Use for Initializing...
Single-output polynomial model of Output-Error (OE) structure (`idpoly`) or state-space model with no disturbance component (`idss` model with K = 0)	Single-output Hammerstein-Wiener model estimation
State-space with no disturbance component (`idss` model with K = 0)	Multi-output Hammerstein-Wiener model estimation

Tip To learn more about when to use linear models, see "When to Fit Nonlinear Models" on page 4-2.

Typically, you use the `oe` or `n4sid` command to obtain the linear model. You can provide the linear model only at the command line when constructing (see `idnlhw`) or estimating (see `nlhw`) a Hammerstein-Wiener model.

The software uses the linear model for initializing the Hammerstein-Wiener estimation:

- Assigns the linear model orders as initial values of nonlinear model orders (`nb` and `nf` properties of the Hammerstein-Wiener (`idnlhw`) and delays (`nk` property).
- Sets the B and F polynomials of the linear transfer function in the Hammerstein-Wiener model structure.

During estimation, the estimation algorithm uses these values to further adjust the nonlinear model to the data.

How to Initialize Hammerstein-Wiener Estimation Using Linear Polynomial Output-Error or State-Space Models

Estimate a Hammerstein-Wiener model using either a linear input-output polynomial model of OE structure or state-space model by typing

```
m = nlhw(data,LinModel)
```

LinModel must be an `idpoly` object of OE structure or state-space (`idss`) model for single-output models, and state-space model for multi-output models. *m* is an `idnlhw` object. *data* is a time-domain `iddata` object.

By default, both the input and output nonlinearity estimators are piecewise linear functions (see `pwlinear`).

Specify different input and output nonlinearity, for example `sigmoid` and `deadzone`:

```
m = nlarx(data,LinModel, 'sigmoid','deadzone')
```

After each estimation, validate the model by comparing the simulated response to the data. To improve the fit of the Hammerstein-Wiener model, adjust various elements of the Hammerstein-Wiener structure. For more information, see "Ways to Configure Hammerstein-Wiener Estimation" on page 4-53.

Example – Using Linear OE Models to Estimate Hammerstein-Wiener Models

1 Load the estimation data.

   ```
   load throttledata.mat
   ```

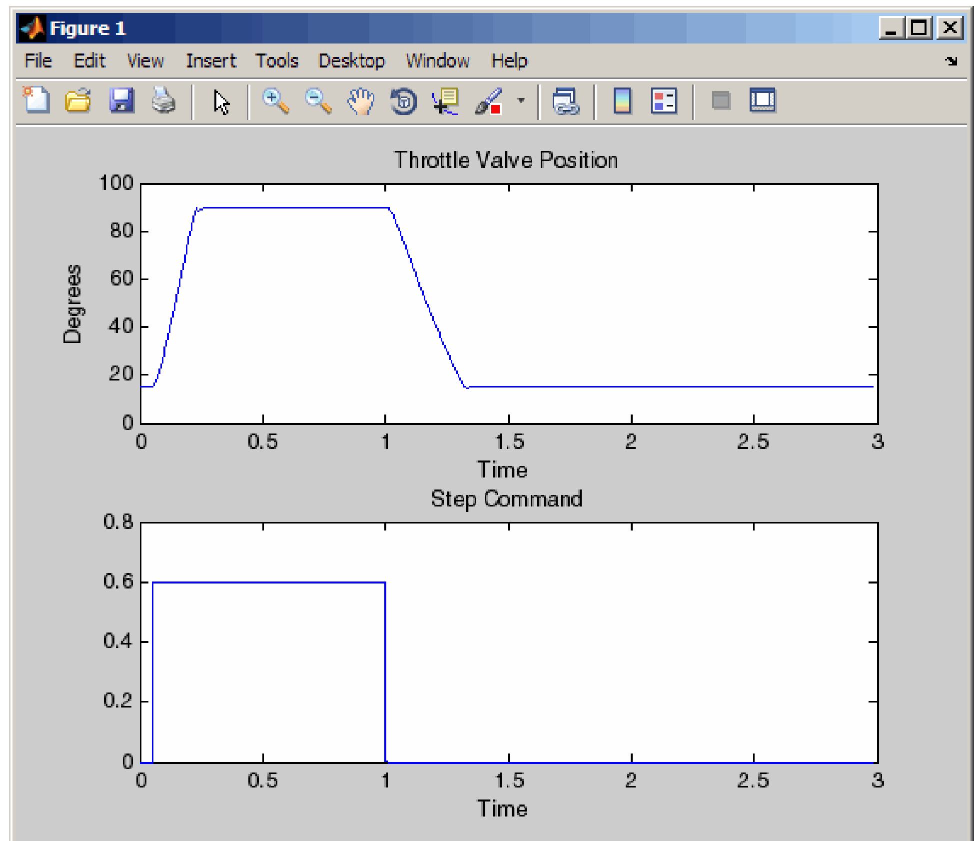
   This command loads the data object `ThrottleData` into the MATLAB workspace. The object contains input and output samples collected from an engine throttle system, sampled at a rate of 100 Hz.

   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.

2 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, as shown in the following figure. 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.

3 Estimate a Hammerstein-Wiener model to model the linear behavior of this single-input single-output system in the normal operating range.

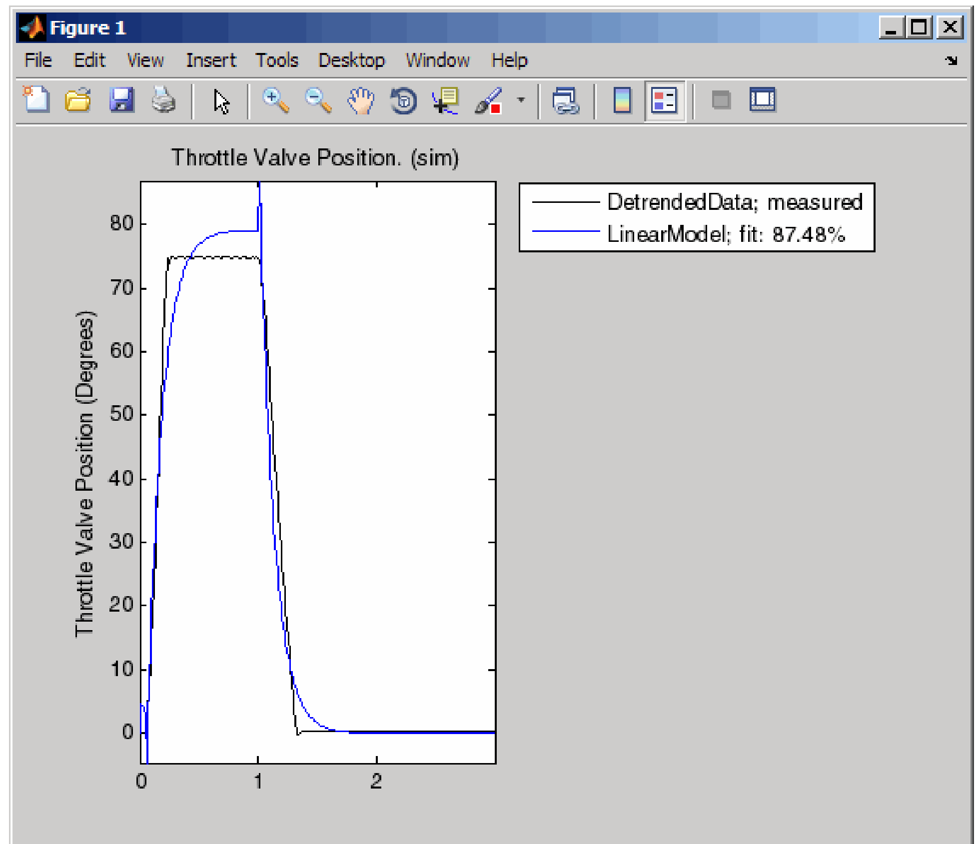
```
% Detrend the data because linear models cannot capture offsets.
Tr = getTrend(ThrottleData);
Tr.OutputOffset = 15;
DetrendedData = detrend(ThrottleData, Tr);

% Estimate a linear OE model with na=2, nb=1, nk=1.
LinearModel = oe(DetrendedData, [2 1 1], 'Focus', 'Simulation');
```

4 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, as shown in the next figure.



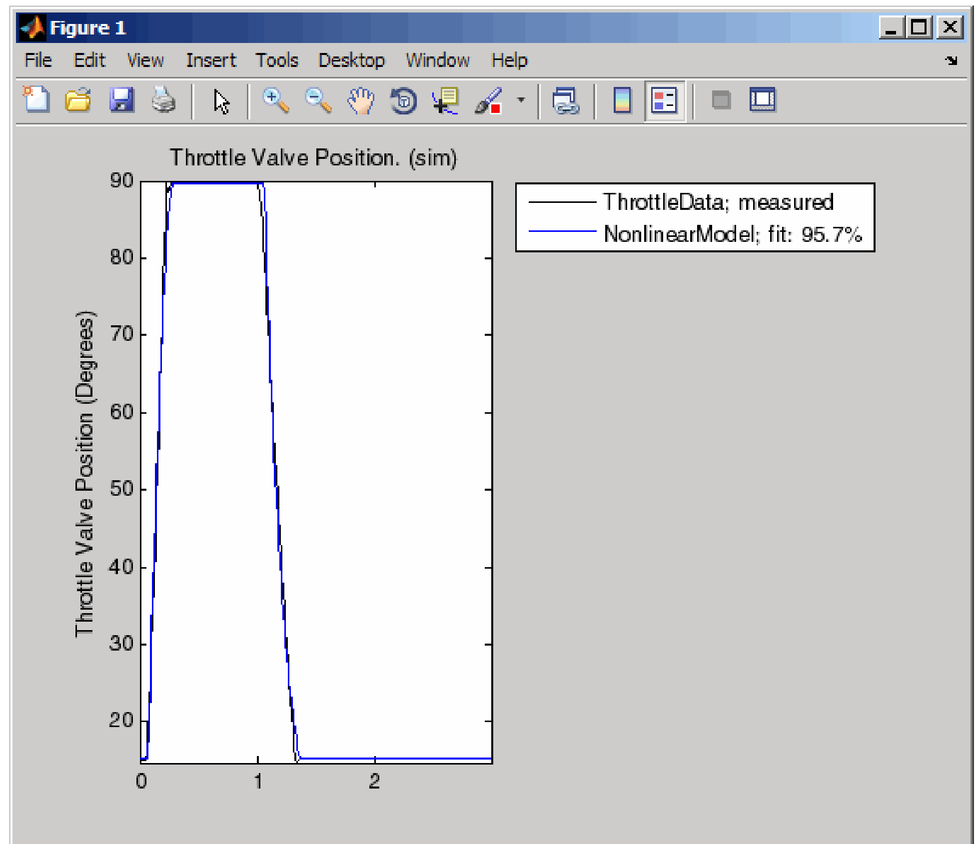
5 Estimate a Hammerstein-Wiener model to model the output saturation.

```
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.

6 Compare the nonlinear model with data.

```
compare(ThrottleData, NonlinearModel)
```



Validating Hammerstein-Wiener Models

- “About Hammerstein-Wiener Plots” on page 4-71
- “How to Create Hammerstein-Wiener Plots in the GUI” on page 4-71
- “How to Validate Hammerstein-Wiener Models at the Command Line” on page 4-72
- “Plotting Nonlinear Block Characteristics” on page 4-74
- “Plotting Linear Block Characteristics” on page 4-75

About Hammerstein-Wiener Plots

Hammerstein-Wiener model plot lets you explore the characteristics of the linear block and the static nonlinearities of the Hammerstein-Wiener model. For more information about estimating nonlinear Hammerstein-Wiener models, see "Identifying Hammerstein-Wiener Models" on page 4-49.

Examining a Hammerstein-Wiener plot can help you determine whether you chose an unnecessarily complicated nonlinearity for modeling your system. For example, if you chose a piece-wise-linear nonlinearity (which is very general), but the plot indicates saturation behavior, then you can estimate a new model using the simpler saturation nonlinearity instead.

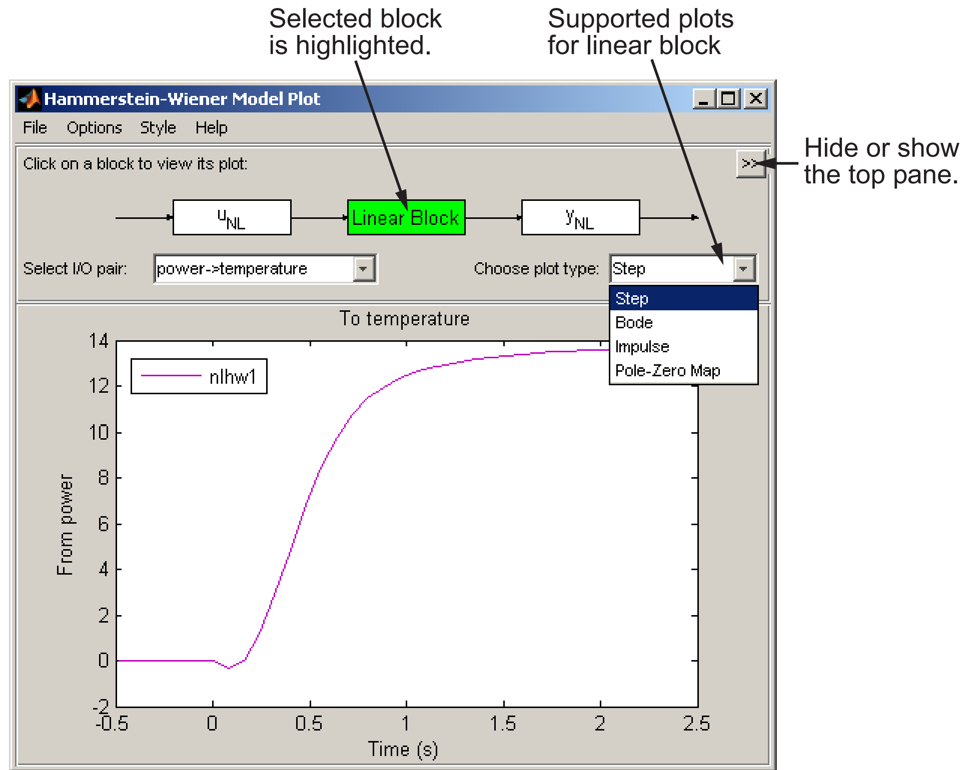
For multivariable systems, you can use the Hammerstein-Wiener plot to determine whether to exclude nonlinearities for specific channels. If the nonlinearity for a specific input or output channel does not exhibit strong nonlinear behavior, you can estimate a new model after setting the nonlinearity at that channel to unit gain.

How to Create Hammerstein-Wiener Plots in the GUI

To create a Hammerstein-Wiener plot in the System Identification Tool GUI, select the **Hamm-Wiener** check box in the **Model Views** area. For general information about creating and working with plots, see "Working with Plots" on page 11-13.

Note The **Hamm-Wiener** check box is unavailable if you do not have a Hammerstein-Wiener model in the Model Board.

To include or exclude a model on the plot, click the corresponding model icon in the System Identification Tool GUI. By default, the input nonlinearity block U_{NL} is selected. You can select the output nonlinearity block Y_{NL} or `Linear Block`, as shown in the next figure.



After you generate a plot, you can learn more about your model by:

- "Plotting Nonlinear Block Characteristics" on page 4-74
- "Plotting Linear Block Characteristics" on page 4-75

How to Validate Hammerstein-Wiener Models at the Command Line

You can use the following approaches to validate Hammerstein-Wiener models at the command line:

Compare Model Output to Measured Output

Compare estimated models using `compare`. Use an independent validation data set whenever possible. For more information about validating models, see "Validating Models After Estimation" on page 8-3.

For example, compare linear and nonlinear ARX models of same order:

```
load iddata1
LM = arx(z1,[2 2 1]) % estimates linear ARX model
M = nlhw(z1,[2 2 1]) % estimates Hammerstein-Wiener model
compare(z1,LM,M) % compares responses of LM and M
                 % against measured data
```

Compare the performance of several models using the properties `M.EstimationInfo.FPE` (final prediction error) and `M.EstimationInfo.LossFcn` (value of loss function at estimation termination). Smaller values typically indicate better performance. However, `m.EstimationInfo.FPE` values might be unreliable when the model contains a large number of parameters relative to the estimation data size. Use these indicators in combination with other validation techniques to draw reliable conclusions.

Simulate and Predict Model Response

Use `sim(idnlhw)` and `predict(idnlhw)` to simulate and predict model response, respectively. To compute the step response of the model, use `step`. See the corresponding reference page for more information.

Analyze Residuals

Residuals are differences between the model output and the measured output. Thus, residuals represent the portion of the output not explained by the model. Use `resid` to compute and plot the residuals.

Plot Nonlinearity

Access the object representing the nonlinearity estimator and its parameters using `M.InputNonlinearity` (or `M.unl`) and `M.OutputNonlinearity` (or `M.ynl`), where `M` is the estimated model.

Use `plot` to view the shape of the nonlinearity and properties of the linear block. For example:

```
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.

The plot command opens the Hammerstein-Wiener Model Plot window. For more information about working with this plot window, see "Plotting Nonlinear Block Characteristics" on page 4-74 and "Plotting Linear Block Characteristics" on page 4-75.

For detailed information about `plot`, type the following command at the prompt:

```
help idnlhw/plot
```

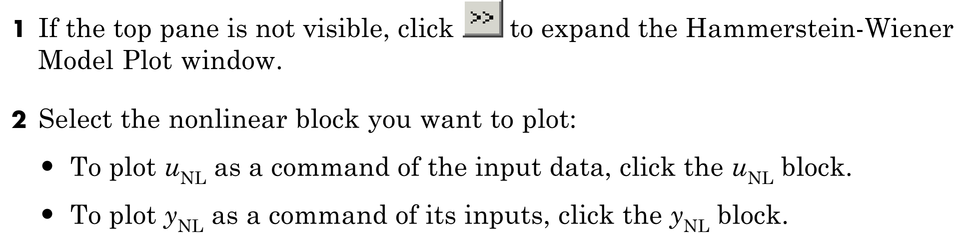
Check Iterative Search Termination Conditions

Use `M.EstimationInfo` to display the estimation termination conditions, where `M` is the estimated `idnlhw` model. For example, check the `WhyStop` field of the `EstimationInfo` property, which describes why the estimation was stopped. For example, the algorithm might have reached the maximum number of iterations or the required tolerance value.

Plotting Nonlinear Block Characteristics

The Hammerstein-Wiener model can contain up to two nonlinear blocks. The nonlinearity at the input to the Linear Block is labeled u_{NL} and is called the *input nonlinearity*. The nonlinearity at the output of the Linear Block is labeled y_{NL} and is called the *output nonlinearity*.

To configure the plot, perform the following steps:



1. If the top pane is not visible, click >> to expand the Hammerstein-Wiener Model Plot window.

2. Select the nonlinear block you want to plot:

   - To plot u_{NL} as a command of the input data, click the u_{NL} block.
   - To plot y_{NL} as a command of its inputs, click the y_{NL} block.

   The selected block is highlighted in green.

   Note The input to the output nonlinearity block y_{NL} is the output from the Linear Block and not the measured input data.

   ---

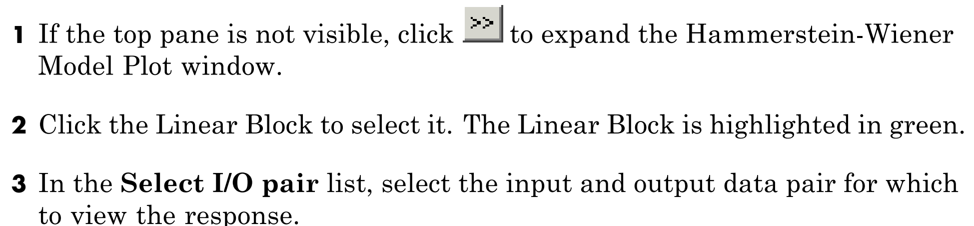
3. If your model contains multiple inputs or outputs, select the channel in the **Select nonlinearity at channel** list. Selecting the channel updates the plot and displays the nonlinearity values versus the corresponding input to this nonlinear block.

4. To change the range of the horizontal axis, select **Options > Set input range** to open the Range for Input to Nonlinearity dialog box. Enter the range using the format `[MinValue MaxValue]`. Click **Apply** and then **Close** to update the plot.

Plotting Linear Block Characteristics

The Hammerstein-Wiener model contains one Linear Block that represents the embedded linear model.

To configure the plot:



1. If the top pane is not visible, click >> to expand the Hammerstein-Wiener Model Plot window.

2. Click the Linear Block to select it. The Linear Block is highlighted in green.

3. In the **Select I/O pair** list, select the input and output data pair for which to view the response.

4 In the **Choose plot type** list, select the linear plot from the following options:

- `Step`
- `Impulse`
- `Bode`
- `Pole-Zero Map`

5 If you selected to plot step or impulse response, you can set the time span. Select **Options > Time span** and enter a new time span in units of time you specified for the model.

For a time span *T*, the resulting response is plotted from *-T/4* to *T*. The default time span is 10.

Click **Apply** and then **Close**.

6 If you selected to plot a Bode plot, you can set the frequency range.

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**, and specify a new frequency vector in units of rad per model time units.

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]`.

Click **Apply** and then **Close**.

Using Hammerstein-Wiener Models

Simulation and Prediction

Use `sim(idnlhw)` to simulate the model output, and `predict(idnlhw)` to predict the model output. To compare models to measured output and to each other, use `compare`.

This toolbox provides a number of options to facilitate how you specify initial states. For example, you can use `findstates(idnlhw)` to automatically search for state values in simulation and prediction applications. You can also specify the states manually.

If you need to specify the states manually, see the `idnlhw` reference page for a definition of the Hammerstein-Wiener model states.

To learn more about how `sim` and `predict` compute the model output, see "How the Software Computes Hammerstein-Wiener Model Output" on page 4-78.

Linearization

Compute linear approximation of Hammerstein-Wiener models using `linearize(idnlhw)` or `linapp`.

`linearize` provides a first-order Taylor series approximation of the system about an operation point (also called *tangent linearization*). `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 4-81.

You can compute the operating point for linearization using `findop(idnlhw)`.

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 Chapter 9, "Control Design Applications".

Simulation and Code Generation Using Simulink

You can import estimated Hammerstein-Wiener Model block into Simulink software using the Hammerstein-Wiener block 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 Hammerstein-Wiener 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 about working in the Simulink environment, see Chapter 10, "System Identification Toolbox Blocks".

How the Software Computes Hammerstein-Wiener Model Output

In most applications, `sim(idnlhw)` and `predict(idnlhw)` are sufficient for computing the simulated and predicted model response, respectively. This advanced topic describes how the software evaluates the output of nonlinearity estimators and uses this output to compute the model response.

Evaluating Nonlinearities (SISO)

Evaluating the predicted output of a nonlinearity for a input u requires that you first extract the input or output nonlinearity F from the model:

```
F = M.InputNonlinearity  % equivalent to F = M.unl
H = M.OutputNonlinearity % equivalent to F = M.ynl
```

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 predicted output values 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 %equivalent to H = M.ynl
```

Example – Low-level 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. For same initial conditions, the prediction results match the simulation results.

1. Estimate Hammerstein-Wiener model:

   ```
   load twotankdata
   estData = iddata(y,u,0.2)
   M = nlhw(estData,[1 5 3],'pwlinear','poly1d');
   ```

2. Extract the input nonlinearity, linear model, and output nonlinearity as separate variables:

```
uNL = M.InputNonlinearity;
linModel = M.LinearModel;
yNL = M.OutputNonlinearity;
```

3. Simulate the output of the input nonlinearity estimator:

```
  u = estData.u; %input data for simulation
% Compute output of input nonlinearity:
  w = evaluate(uNL, u);
% 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);
% Previous commands are equivalent to:
  ysim = sim(M, u);
% Compare low-level and direct simulation results:
  time = estData.SamplingInstants;
  plot(time, y, time, ysim, '.')
```

Linear Approximation of Nonlinear Black-Box Models

In this section...

"Why Compute a Linear Approximation of a Nonlinear Model?" on page 4-81

"Choosing Your Linear Approximation Approach" on page 4-81

"Linear Approximation of Nonlinear Black-Box Models for a Given Input" on page 4-82

"Tangent Linearization of Nonlinear Black-Box Models" on page 4-82

"Computing Operating Points for Nonlinear Black-Box Models" on page 4-83

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. For more information, see Chapter 9, "Control Design Applications".

Choosing Your Linear Approximation Approach

System Identification Toolbox software provides two approaches for computing a linear approximation of nonlinear ARX and Hammerstein-Wiener 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 4-82.

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 4-82.

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 4-83.

For more information, see the `linearize(idnlarx)` or `linearize(idnlhw)` 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 4-83
- "Computing Operating Points at a Simulation Snapshot" on page 4-84

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 `operspec(idnlarx)` or `operspec(idnlhw)`.

For more information, see the `findop(idnlarx)` or `findop(idnlhw)` 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(idnlarx)` 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.

ODE Parameter Estimation (Grey-Box Modeling)

- "Supported Grey-Box Models" on page 5-2
- "Data Supported by Grey-Box Models" on page 5-3
- "Choosing idgrey or idnlgrey Model Object" on page 5-4
- "Estimating Linear Grey-Box Models" on page 5-6
- "Estimating Nonlinear Grey-Box Models" on page 5-15
- "After Estimating Grey-Box Models" on page 5-39

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 and known parameter values. 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 perform black-box modeling.

You can also use the `idss` model object to perform structured model estimation by using structure matrices `As`, `Bs`, `Cs`, `Ds`, `X0s`, `Ks` 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 "How to Estimate State-Space Models with Structured Parameterization" on page 3-94.

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. If you are using the System Identification Tool GUI, then import the data into the GUI to make the data available to the toolbox. However, if you prefer to work at the command line, then represent your data as an `iddata` or `idfrd` object. For more information about preparing data for identification, see Chapter 2, "Data Import and Processing".

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 5-6.

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 5-15.

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.	No	Yes
Handle initial states individually.	No	Yes
Perform linear analysis.	Yes For example, use the `bode` command.	No

Comparison of idgrey and idnlgrey Objects (Continued)

Settings and Operations	Supported by idgrey?	Supported by idnlgrey?
Honor stability constraints.	Yes Specify constraints using `Algorithm.Advanced.Threshold.Zstability` and `Algorithm.Advanced.Threshold.Sstability` model properties.	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 `Algorithm.Focus` property to `'Simulation'` or `'Prediction'`.	No Because `idnlgrey` models are Output-Error models, there is no difference between simulation and prediction results.

Estimating Linear Grey-Box Models

In this section...

"Specifying the Linear Grey-Box Model Structure" on page 5-6

"Example – Creating a Function for Representing a Grey-Box Model" on page 5-7

"Example – Estimating a Continuous-Time Grey-Box Model for Heat Diffusion" on page 5-9

"Example – Estimating a Discrete-Time Grey-Box Model with Parameterized Disturbance" on page 5-12

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 output-only time-series data.

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.

In continuous-time, the state-space description has the following form:

$$\begin{aligned} \dot{x}(t) &= Fx(t) + Gu(t) + \tilde{K}w(t) \\ y(t) &= Hx(t) + Du(t) + w(t) \\ x(0) &= x0 \end{aligned}$$

The discrete-time state-space model structure is often written in the *innovations form*:

$$\begin{aligned} x(kT+T) &= Ax(kT) + Bu(kT) + Ke(kT) \\ y(kT) &= Cx(kT) + Du(kT) + e(kT) \\ x(0) &= x0 \end{aligned}$$

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,K,x0] = myfunc(par,T,aux)
```

where the matrices A, B, C, D, K, and x0 represent both the continuous-time and discrete-time state-space description of the system, myfunc is the name of the file, par contains the parameters as a column vector, and T is the sampling interval. 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.

CDmfile is an optional argument that describes whether the resulting state-space matrices are in discrete time or continuous time. By default, CDmfile='cd', which means that the sampling interval property of the model Ts determines whether the model is continuous or discrete in time. For more information about these arguments, see the idgrey reference page.

Use pem to estimate your grey-box model.

Example – Creating a Function for Representing a Grey-Box Model

In this example, you 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 $\theta_1 = -1$ and $\theta_2 = 0.25$. 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:

$$\dot{x}(t) = Fx(t) + Gu(t) + \tilde{K}w(t)$$
$$y(t) = Hx(t) + Du(t) + w(t)$$
$$x(0) = x0$$

To prepare this model for identification:

1 Create the following file to represent the model structure in this example:

```
function [A,B,C,D,K,x0] = myfunc(par,T,aux)
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];
```

2. Use the following syntax to define an `idgrey` model object based on the `myfunc` file:

   ```
   m = idgrey('myfunc',par,'c',T,aux)
   ```

   where `par` represents user-defined parameters and contains their nominal (initial) values. `'c'` specifies that the underlying parameterization is in continuous time. `aux` contains the values of the auxiliary parameters.

   Note You must specify `T` and `aux` even if they are not used by the `myfunc` code.

Use `pem` to estimate the grey-box parameter values:

```
m = pem(data,m)
```

where `data` is the estimation data and `m` is the `idgrey` object with unknown parameters.

Note Compare this example to "Example – Estimating Structured Continuous-Time State-Space Models" on page 3-98, where the same problem is solved using a structured state-space representation.

Example – Estimating a Continuous-Time Grey-Box Model for Heat Diffusion

In this example, you 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:

$$\frac{\partial^2 x(t,\xi)}{\partial \xi^2} = \frac{x\left(t,\xi+\Delta L\right) - 2x(t,\xi) + x\left(t,\xi-\Delta L\right)}{\left(\Delta L\right)^2}$$

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:

$$k \cdot \Delta L \le \xi < \left(k+1\right)\Delta L$$

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:

$$\begin{aligned}
\dot{x}(t) &= Fx(t) + Gu(t) + \tilde{K}w(t) \\
y(t) &= Hx(t) + Du(t) + w(t) \\
x(0) &= x0
\end{aligned}$$

1 Create a MATLAB file.

The following code describes the state-space equation for this model. In this case, the auxiliary variables specify grid-size variables, so that you can modify the grid size without the code file.

```
function [A,B,C,D,K,x0] = heatd(pars,T,aux)
% Number of points in the space-discretization
```

```
Ngrid = aux(1);
% Length of the rod
L = aux(2);
% Initial rod temperature (uniform)
temp = aux(3);
% Space interval
deltaL = L/Ngrid;
% Heat-diffusion coefficient
kappa = pars(1);
% Heat transfer coefficient at far end of rod
htf = pars(2);
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 = zeros(Ngrid,1);
B(Ngrid,1) = htf/deltaL;
C = zeros(1,Ngrid);
C(1,1) = 1;
D = 0;
K = zeros(Ngrid,1);
x0 = temp*ones(Ngrid,1);
```

2 Use the following syntax to define an `idgrey` model object based on the `heatd` code file:

```
m = idgrey('heatd',[0.27 1],'c',[10,1,22])
```

This command specifies the auxiliary parameters as inputs to the function, include the model order `10`, the rod length of 1 meter, and an initial temperature of `22` degrees Celsius. The command also specifies the initial values for heat conductivity as `0.27`, and for the heat transfer coefficient as `1`.

3 For given `data`, you can use `pem` to estimate the grey-box parameter values:

```
me = pem(data,m)
```

The following command shows how you can specify to estimate a new model with different auxiliary variables directly in the estimator command:

```
me = pem(data,m,'FileArgument',[20,1,22])
```

This syntax uses the `FileArgument` model property to specify a finer grid using a larger value for `Ngrid`. For more information about linear grey-box model properties, see the `idgrey` reference page.

Example – Estimating a Discrete-Time Grey-Box Model with Parameterized Disturbance

This example shows how to create a single-input and single-output grey-box model structure when you know the variance of the measurement noise. The code in this example uses the Control System Toolbox command `kalman` for computing the Kalman gain from the known and estimated noise variance.

Description of the SISO System

This example is based on a discrete, single-input and single-output (SISO) system represented by the following state-space equations:

$$x(kT+T) = \begin{bmatrix} par1 & par2 \\ 1 & 0 \end{bmatrix} x(kT) + \begin{bmatrix} 1 \\ 0 \end{bmatrix} u(kT) + w(kT)$$

$$y(kT) = \begin{bmatrix} par3 & par4 \end{bmatrix} x(kT) + e(kT)$$

$$x(0) = x0$$

where w and e are independent white-noise terms with covariance matrices *R1* and *R2*, respectively. *R1*=E{ww'} is a 2–by-2 matrix and *R2*=E{ee'} is a scalar. *par1*, *par2*, *par3*, and *par4* represent the unknown parameter values to be estimated.

Assume that you know the variance of the measurement noise *R2* to be 1. *R1*(1,1) is unknown and is treated as an additional parameter *par5*. The remaining elements of *R1* are known to be zero.

Estimating the Parameters of an idgrey Model

You can represent the system described in "Description of the SISO System" on page 5-12 as an `idgrey` (grey-box) model using a function. Then, you can use this file and the `pem` command to estimate the model parameters based on initial parameter guesses.

To run this example, you must load an input-output data set and represent it as an `iddata` or `idfrd` object called `data`. For more information about this operation, see "Representing Time- and Frequency-Domain Data Using iddata Objects" on page 2-53 or "Representing Frequency-Response Data Using idfrd Objects" on page 2-73.

To estimate the parameters of a grey-box model:

1. Create the file `mynoise` that computes the state-space matrices as a function of the five unknown parameters and the auxiliary variable that represents the known variance `R2`.

Note `R2` is treated as an auxiliary variable rather than assigned a value in the file to let you change this value directly at the command line and avoid editing the file.

```
function [A,B,C,D,K,x0] = mynoise(par,T,aux)
R2 = aux(1); % Known measurement noise variance
A = [par(1) par(2);1 0];
B = [1;0];
C = [par(3) par(4)];
D = 0;
R1 = [par(5) 0;0 0];
[est,K] = kalman(ss(A,eye(2),C,0,T),R1,R2);
      % Uses Control System Toolbox product
      % u=[]
x0 = [0;0];
```

2 Specify initial guesses for the unknown parameter values and the auxiliary parameter value `R2`:

```
par1 = 0.1; % Initial guess for A(1,1)
par2 = -2;  % Initial guess for A(1,2)
par3 = 1;   % Initial guess for C(1,1)
par4 = 3;   % Initial guess for C(1,2)
par5 = 0.2; % Initial guess for R1(1,1)
Pvec = [par1; par2; par3; par4; par5]
auxVal = 1; % R2=1
```

3 Construct an `idgrey` model using the `mynoise` file:

```
Minit = idgrey('mynoise',Pvec,'d',auxVal);
```

The third input argument `'d'` specifies a discrete-time system.

4 Estimate the model parameter values from data:

```
Model = pem(data,Minit)
```

Estimating Nonlinear Grey-Box Models

In this section...

"Specifying the Nonlinear Grey-Box Model Structure" on page 5-15

"Constructing the idnlgrey Object" on page 5-17

"Using pem to Estimate Nonlinear Grey-Box Models" on page 5-17

"Nonlinear Grey-Box Model Estimation Algorithm Options" on page 5-18

"Represent Nonlinear Dynamics Using MATLAB File for Grey-Box Estimation" on page 5-20

"Nonlinear Grey-Box Demos and Examples" on page 5-38

Specifying the Nonlinear Grey-Box Model Structure

You must represent your system as a set of first-order nonlinear difference or differential equations:

$$x^{\dagger}(t) = F(t, x(t), u(t), par1, par2, ..., parN)$$
$$y(t) = H(t, x(t), u(t), par1, par2, ..., parN) + e(t)$$
$$x(0) = x0$$

where $x^{\dagger}(t) = dx(t)/dt$ for continuous-time representation and $x^{\dagger}(t) = x(t + T_s)$ for discrete-time representation with Ts as the sampling interval. F and H are arbitrary linear or nonlinear functions with Nx and Ny components, respectively. Nx is the number of states and Ny is the number of outputs.

After you establish the equations for your system, create a function or MEX-file. MEX-files, which can be created in C or Fortran, are dynamically linked subroutines that can be loaded and executed by the MATLAB interpreter. For more information about MEX-files, see the MATLAB documentation.

The purpose of the model file is to return the state derivatives and model outputs as a function of time, states, inputs, and model parameters, as follows:

```
[dx,y] = MODFILENAME(t,x,u,p1,p2, ...,pN,FileArgument)
```

Tip The template file for writing the C MEX-file, `IDNLGREY_MODEL_TEMPLATE.c`, is located in `matlab/toolbox/ident/nlident`.

The output variables are:

- `dx` — Represents the right side(s) of the state-space equation(s). A column vector with *Nx* entries. For static models, `dx=[]`.

  For discrete-time models. `dx` is the value of the states at the next time step `x(t+Ts)`.

  For continuous-time models. `dx` is the state derivatives at time *t*, or $\frac{dx}{dt}$.
- `y` — Represents the right side(s) of the output equation(s). A column vector with *Ny* entries.

The file inputs are:

- `t` — Current time.
- `x` — State vector at time `t`. For static models, equals `[]`.
- `u` — Input vector at time `t`. For time-series models, equals `[]`.
- `p1,p2, ...,pN` — Parameters, which can be real scalars, column vectors or two-dimensional matrices. `N` is the number of parameter objects. For scalar parameters, `N` is the total number of parameter elements.
- `FileArgument` — Contains auxiliary variables that might be required for updating the constants in the state equations.

Tip After creating a model file, call it directly from the MATLAB software with reasonable inputs and verify the output values.

For an example of creating grey-box model files and `idnlgrey` model object, see the demo Creating idnlgrey Model Files.

Constructing the idnlgrey Object

After you create the function or MEX-file with your model structure, you must define an `idnlgrey` object. This object shares many of the properties of the linear `idgrey` model object.

Use the following syntax to define the `idnlgrey` model object:

```
m = idnlgrey('filename',Order,Parameters,InitialStates)
```

The `idnlgrey` arguments are defined as follows:

- `'filename'` — Name of the function or MEX-file storing the model structure. This file must be on the MATLAB path when you use this model object for model estimation, prediction, or simulation.
- `Order` — Vector with three entries `[Ny Nu Nx]`, specifying the number of model outputs `Ny`, the number of inputs `Nu`, and the number of states `Nx`.
- `Parameters` — Parameters, specified as `struct` arrays, cell arrays, or double arrays.
- `InitialStates` — Specified in the same way as parameters. Must be the fourth input to the `idnlgrey` constructor.

For detailed information about this object and its properties, see the `idnlgrey` reference page.

Use `pem` to estimate your grey-box model.

Using pem to Estimate Nonlinear Grey-Box Models

You can use the `pem` command to estimate the unknown `idnlgrey` model parameters and initial states using measured data.

The input-output dimensions of the data must be compatible with the input and output orders you specified for the `idnlgrey` model.

Use the following general estimation syntax:

```
m = pem(data,m)
```

where `data` is the estimation data and `m` is the `idnlgrey` model object you constructed.

You can pass additional property-value pairs to `pem` to specify the properties of the model or the estimation algorithm. Assignable properties include the ones returned by the `get(idnlgrey)` command and the algorithm properties returned by the `get(idnlgrey, 'Algorithm')`, such as `MaxIter` and `Tolerance`. For detailed information about these model properties, see the `idnlgrey` reference page.

For more information about validating your models, see Chapter 8, "Model Analysis".

Nonlinear Grey-Box Model Estimation Algorithm Options

The `Algorithm` property of the model specifies the estimation algorithm, which simulates the model several times by trying various parameter values to reduce the prediction error.

The following algorithm properties can affect the quality of the results:

- "Simulation Method" on page 5-18
- "Search Method" on page 5-19
- "Gradient Options" on page 5-19
- "Example – Specifying Algorithm Properties" on page 5-20

For detailed information about these and other model properties, see the `idnlgrey` reference page.

Simulation Method

You can specify the simulation method using the `SimulationOptions` (`struct`) fields of the model `Algorithm` property.

System Identification Toolbox software provides several variable-step and fixed-step solvers for simulating `idnlgrey` models. To view a list of available solvers and their properties, type the following command at the prompt:

```
idprops idnlgrey algorithm.simulationoptions
```

For discrete-time systems, the default solver is `'FixedStepDiscrete'`. For continuous-time systems, the default solver is `'ode45'`.

By default, `SimulationOptions.Solver` is set to `'Auto'`, which automatically selects either `'ode45'` or `'FixedStepDiscrete'` during estimation and simulation—depending on whether the system is continuous or discrete in time.

Search Method

You can specify the search method for estimating model parameters using the `SearchMethod` field of the `Algorithm` property. Two categories of methods are available for nonlinear grey-box modeling.

One category of methods consists of the minimization schemes that are based on line-search methods, including Gauss-Newton type methods, steepest-descent methods, and Levenberg-Marquardt methods.

The Trust-Region Reflective Newton method of nonlinear least-squares (`lsqnonlin`), where the cost is the sum of squares of errors between the measured and simulated outputs, requires Optimization Toolbox™ software. When the parameter bounds differ from the default +/- Inf, this search method handles the bounds better than the schemes based on a line search. However, unlike the line-search-based methods, `lsqnonlin` only works with `Criterion='Trace'`.

By default, `SearchMethod` is set to `Auto`, which automatically selects a method from the available minimizers. If the Optimization Toolbox product is installed, `SearchMethod` is set to `'lsqnonlin'`. Otherwise, `SearchMethod` is a combination of line-search based schemes.

Gradient Options

You can specify the method for calculating gradients using the `GradientOptions` field of the `Algorithm` property. *Gradients* are the derivatives of errors with respect to unknown parameters and initial states.

Gradients are calculated by numerically perturbing unknown quantities and measuring their effects on the simulation error.

Option for gradient computation include the choice of the differencing scheme (forward, backward or central), the size of minimum perturbation of the unknown quantities, and whether the gradients are calculated simultaneously or individually.

Example – Specifying Algorithm Properties

You can specify the `Algorithm` fields directly in the estimation syntax, as property-value pairs.

For example, you can specify the following properties as part of the `pem` syntax:

```
m = pem(data,init_model,'Search','gn',...
                        'MaxIter',5,...
                        'Display','On')
```

Represent Nonlinear Dynamics Using MATLAB File for Grey-Box Estimation

This example shows how to construct, estimate and analyze nonlinear grey-box models.

Nonlinear grey-box (`idnlgrey`) models are suitable for estimating parameters of systems that are described by nonlinear state-space structures in continuous or discrete time. You can use both `idgrey` (linear grey-box model) and `idnlgrey` objects to model linear systems. However, you can only use `idnlgrey` to represent nonlinear dynamics. To learn about linear grey-box modeling using `idgrey`, see "Building Structured and User-Defined Models Using System Identification Toolbox™".

About the Model

In this example, you model the dynamics of a linear DC motor using the `idnlgrey` object.

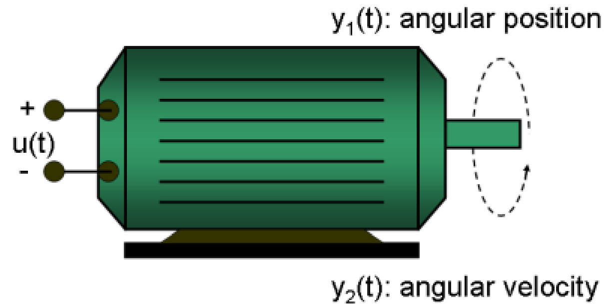


Figure 1: Schematic diagram of a DC-motor.

If you ignore the disturbances and choose `y(1)` as the angular position [rad] and `y(2)` as the angular velocity [rad/s] of the motor, you can set up a linear state-space structure of the following form (see Ljung, L. System Identification: Theory for the User, Upper Saddle River, NJ, Prentice-Hall PTR, 1999, 2nd ed., p. 95-97 for the derivation):

```
d         | 0      1    |          |   0    |
-- x(t) = |             | x(t) +   |        | u(t)
dt        | 0   -1/tau  |          | k/tau  |

          | 1   0 |
   y(t) = |       | x(t)
          | 0   1 |
```

`tau` is the time-constant of the motor in [s] and `k` is the static gain from the input to the angular velocity in [rad/(V*s)] . See Ljung (1999) for how `tau` and `k` relate to the physical parameters of the motor.

About the Input-Output Data

1. Load the DC motor data.

   ```
   load(fullfile(matlabroot, 'toolbox', 'ident', 'iddemos', 'data', 'dcmotor
   ```

2. Represent the estimation data as an `iddata` object.

   ```
   z = iddata(y, u, 0.1, 'Name', 'DC-motor');
   ```

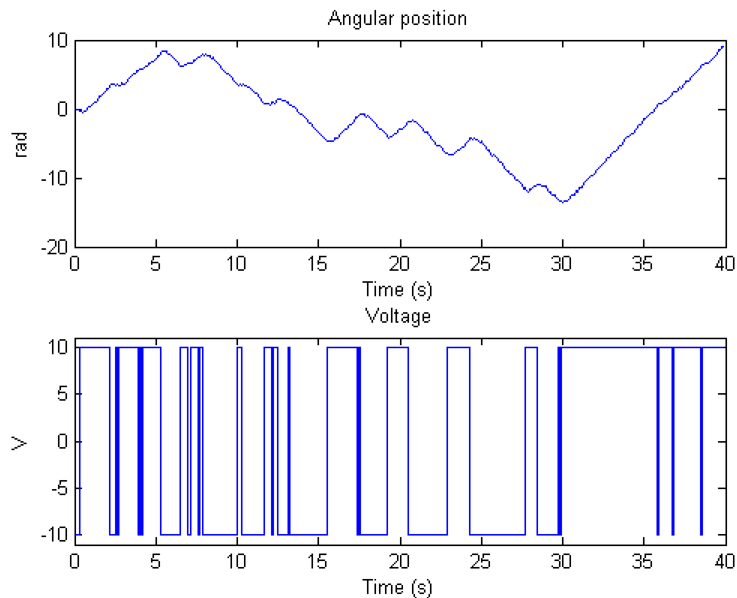
3. Specify input and output signal names, start time and time units.

```
set(z, 'InputName', 'Voltage', 'InputUnit', 'V');
set(z, 'OutputName', {'Angular position', 'Angular velocity'});
set(z, 'OutputUnit', {'rad', 'rad/s'});
set(z, 'Tstart', 0, 'TimeUnit', 's');
```

4. Plot the data.

The data is shown in two plot windows.

```
figure('Name', [z.Name ': Voltage input -> Angular position output']);
plot(z(:, 1, 1));   % Plot first input-output pair (Voltage -> Angular po
figure('Name', [z.Name ': Voltage input -> Angular velocity output']);
plot(z(:, 2, 1));   % Plot second input-output pair (Voltage -> Angular v
```



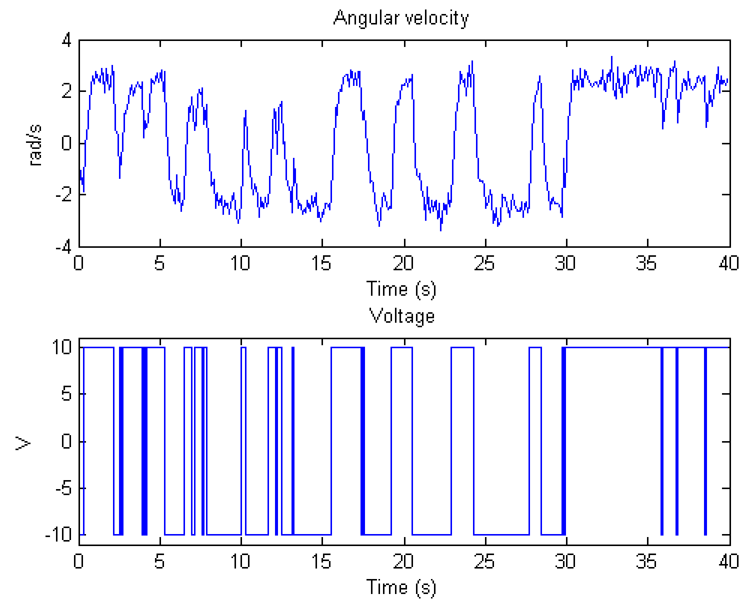


Figure 2: Input-output data from a DC-motor.

Linear Modeling of the DC-Motor

1. Represent the DC motor structure in a function.

In this example, you use a MATLAB file, but you can also use C MEX-files (to gain computational speed), P-files or function handles. For more information, see Creating IDNLGREY Model Files".

The DC-motor function is called `dcmotor_m.m` and is shown below.

```
function [dx, y] = dcmotor_m(t, x, u, tau, k, varargin)

% Output equations.
y = [x(1);                              ... % Angular position.
     x(2)                               ... % Angular velocity.
    ];

% State equations.
```

```
dx = [x(2);                              ... % Angular position.
      -(1/tau)*x(2)+(k/tau)*u(1)   ... % Angular velocity.
     ];
```

The file must always be structured to return the following:

Output arguments:

- `dx` is the right-hand side(s) of the state-space equation(s)
- `y` is the output equation(s)

Input arguments:

- The first three input arguments must be: `t` (time), `x` (state vector, [] for static systems), `u` (input vector, [] for time-series).
- Ordered list of parameters follow. The parameters can be scalars, column vectors, or 2-dimensional matrices.
- `varargin` for the auxiliary input arguments

2. Represent the DC motor dynamics using an `idnlgrey` object.

The model describes how the inputs generate the outputs using the state equation(s).

```
FileName      = 'dcmotor_m';       % File describing the model structure
Order         = [2 1 2];           % Model orders [ny nu nx].
Parameters    = [1; 0.28];         % Initial parameters. Np = 2.
InitialStates = [0; 0];            % Initial initial states.
Ts            = 0;                 % Time-continuous system.
nlgr = idnlgrey(FileName, Order, Parameters, InitialStates, Ts, ...
                'Name', 'DC-motor');
```

In practice, there are disturbances that affect the outputs. An `idnlgrey` model does not explicitly model the disturbances, but assumes that these are just added to the output(s). Thus, `idnlgrey` models are equivalent to Output-Error (OE) models. Without a noise model, past outputs do

not influence future outputs, which means that predicted output for any prediction horizon k coincide with simulated outputs.

3. Specify input and output names, and units.

```
set(nlgr, 'InputName', 'Voltage', 'InputUnit', 'V',               ...
          'OutputName', {'Angular position', 'Angular velocity'}, ...
          'OutputUnit', {'rad', 'rad/s'},                         ...
          'TimeUnit', 's');
```

4. Specify names and units of the initial states and parameters.

```
setinit(nlgr, 'Name', {'Angular position' 'Angular velocity'});
setinit(nlgr, 'Unit', {'rad' 'rad/s'});
setpar(nlgr, 'Name', {'Time-constant' 'Static gain'});
setpar(nlgr, 'Unit', {'s' 'rad/(V*s)'});
```

You can also use `setinit` and `setpar` to assign values, minima, maxima, and estimation status for all initial states or parameters simultaneously.

5. View the initial model.

a. Get basic information about the model.

The DC-motor has 2 (initial) states and 2 model parameters.

```
size(nlgr)

Nonlinear state space model with 2 outputs, 1 input, 2 states, and 2 para
```

b. View the initial states and parameters.

Both the initial states and parameters are structure arrays. The fields specify the properties of an individual initial state or parameter. Type `idprops idnlgrey InitialStates` and `idprops idnlgrey Parameters` for more information.

```
nlgr.InitialStates(1)
nlgr.Parameters(2)
```

```
ans =

       Name: 'Angular position'
       Unit: 'rad'
      Value: 0
    Minimum: -Inf
    Maximum: Inf
      Fixed: 1


ans =

       Name: 'Static gain'
       Unit: 'rad/(V*s)'
      Value: 0.2800
    Minimum: -Inf
    Maximum: Inf
      Fixed: 0
```

c. Retrieve information for all initial states or model parameters in one call.

For example, obtain information on initial states that are fixed (not estimated) and the minima of all model parameters.

```
getinit(nlgr, 'Fixed')
getpar(nlgr, 'Min')


ans =

    [1]
    [1]


ans =

    [-Inf]
    [-Inf]
```

d. Obtain basic information about the object:

```
nlgr

Time-continuous nonlinear state-space model defined by 'dcmotor_m' (MATLA

   dx/dt = F(t, u(t), x(t), p1, p2)
    y(t) = H(t, u(t), x(t), p1, p2) + e(t)

with 1 input, 2 states, 2 outputs, and 2 free parameters (out of 2).
```

Use `get` to obtain more information about the model properties. The `idnlgrey` object shares many properties of parametric linear model objects.

```
get(nlgr)

              Name: 'DC-motor'
                Ts: 0
          TimeUnit: 's'
      TimeVariable: 't'
         InputName: {'Voltage'}
         InputUnit: {'V'}
        OutputName: {2x1 cell}
        OutputUnit: {2x1 cell}
          FileName: 'dcmotor_m'
             Order: [1x1 struct]
        Parameters: [2x1 struct]
     InitialStates: [2x1 struct]
      FileArgument: {}
  CovarianceMatrix: 'Estimate'
     NoiseVariance: [2x2 double]
         Algorithm: [1x1 struct]
    EstimationInfo: [1x1 struct]
             Notes: {}
          UserData: []
```

Performance Evaluation of the Initial DC-Motor Model

Before estimating the parameters `tau` and `k`, simulate the output of the system with the parameter guesses using the default differential equation solver (a Runge-Kutta 45 solver with adaptive step length adjustment).

1. Set the absolute and relative error tolerances to small values (`1e-6` and `1e-5`, respectively).

```
nlgr.Algorithm.SimulationOptions.AbsTol = 1e-6;
nlgr.Algorithm.SimulationOptions.RelTol = 1e-5;
```

2. Compare the simulated output with the measured data.

`compare` displays both measured and simulated outputs of one or more models, whereas `predict`, called with the same input arguments, displays the simulated outputs.

The simulated and measured outputs are shown in a plot window.

```
figure;
compare(z, nlgr);
```

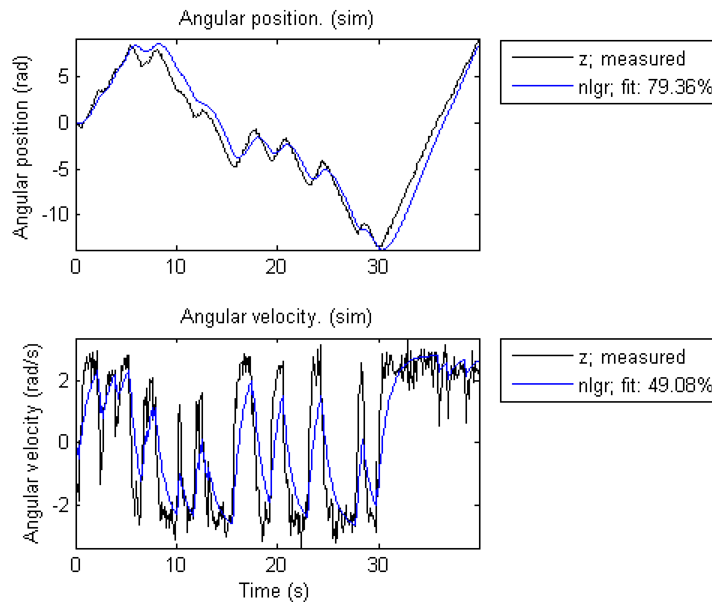


Figure 3: Comparison between measured outputs and the simulated outputs of the initial DC-motor model.

Parameter Estimation

Estimate the parameters and initial states using `pem` (Prediction-Error identification Method).

```
setinit(nlgr, 'Fixed', {false false});   % Estimate the initial state.
nlgr = pem(z, nlgr, 'Display', 'Full');


Criterion: Trace minimization
   Scheme: Nonlinear least squares with automatically chosen line search
--------------------------------------------------------------------------
                           Norm of      First-order      Improvement (%)
 Iteration       Cost       step         optimality     Expected   Achiev
--------------------------------------------------------------------------
     0        2.78714          -      5.06e+003         75.6           -
     1       0.223389      0.827      1.89e+003         75.6          92
     2       0.119343      0.106           44.4         45.7        46.6
     3       0.118707     0.0124           0.67        0.527       0.533
     4       0.118707   0.000834        0.00891     0.000195    0.000198
--------------------------------------------------------------------------
```

Performance Evaluation of the Estimated DC-Motor Model

1. Review the information about the estimation process.

This information is stored in the `EstimationInfo` property of the `idnlgrey` object. The property also contains information about how the model was estimated, such as solver and search method, data set, and why the estimation was terminated.

```
nlgr.EstimationInfo


ans =

            Status: 'Estimated model (PEM)'
```

```
              Method: 'Solver: ode45; Search: gn, lm, gna, grad'
             LossFcn: 0.0011
                 FPE: 0.0011
            DataName: 'DC-motor'
          DataLength: 400
              DataTs: {[0.1000]}
          DataDomain: 'Time'
     DataInterSample: {'zoh'}
             WhyStop: 'Near (local) minimum, (norm(g) < tol).'
          UpdateNorm: 1.9799e-004
     LastImprovement: 1.9799e-004
          Iterations: 4
        InitialGuess: [1x1 struct]
             Warning: ''
      EstimationTime: 9.2665
```

2. Evaluate the model quality by comparing simulated and measured outputs.

The fits are 98% and 84%, which indicate that the estimated model captures the dynamics of the DC motor well.

```
figure;
compare(z, nlgr);
```

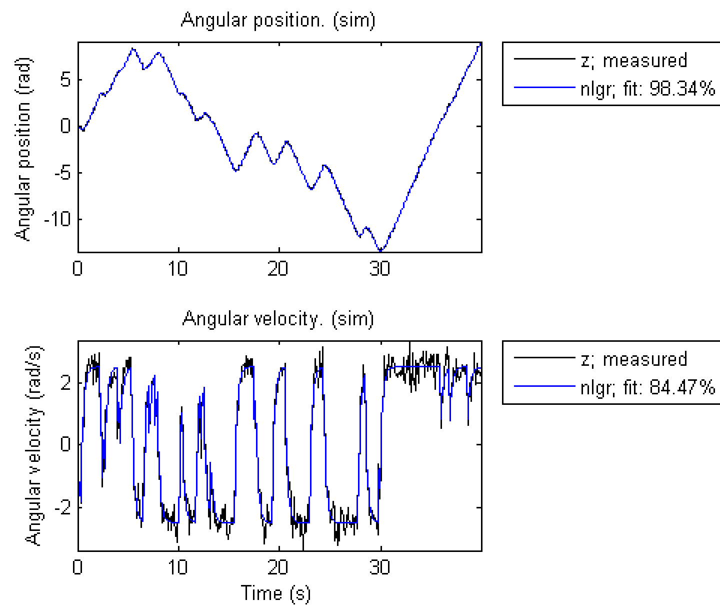


Figure 4: Comparison between measured outputs and the simulated outputs of the estimated IDNLGREY DC-motor model.

3. Compare the performance of the `idnlgrey` model with a second-order ARX model.

```
dcarx = arx(z, 'na', [2 2; 2 2], 'nb', [2; 2], 'nk', [1; 1]);
figure;
compare(z, nlgr, dcarx);
```

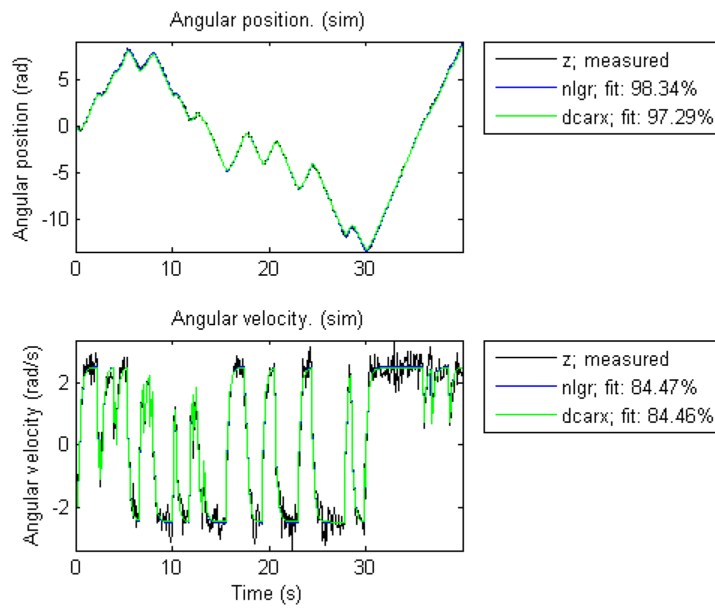


Figure 5: Comparison between measured outputs and the simulated outputs of the estimated IDNLGREY and ARX DC-motor models.

4. Check the prediction errors.

The prediction errors obtained are small and are centered around zero (non-biased).

```
figure;
pe(z, nlgr);
```

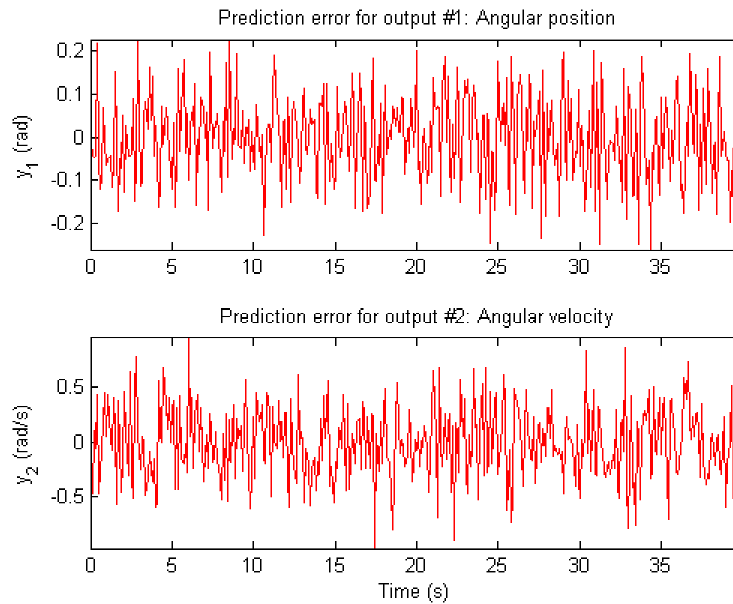
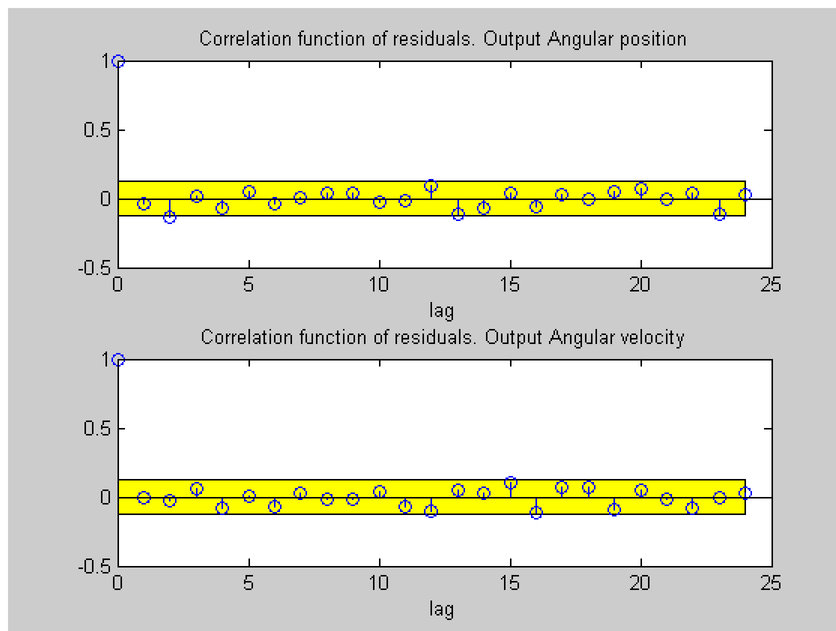


Figure 6: Prediction errors obtained with the estimated IDNLGREY DC-motor model.

5. Check the residuals ("leftovers").

Residuals indicate what is left unexplained by the model and are small for good model quality. Execute the following two lines of code to generate the residual plot. Press any key to advance from one plot to another.

```
figure('Name', [nlgr.Name ': residuals of estimated model']);
resid(z, nlgr);
```


Correlation function of residuals. Output Angular position
1
0.5
0
-0.5
0
5
10
15
20
25
lag
Correlation function of residuals. Output Angular velocity
1
0.5
0
-0.5
0
5
10
15
20
25
lag

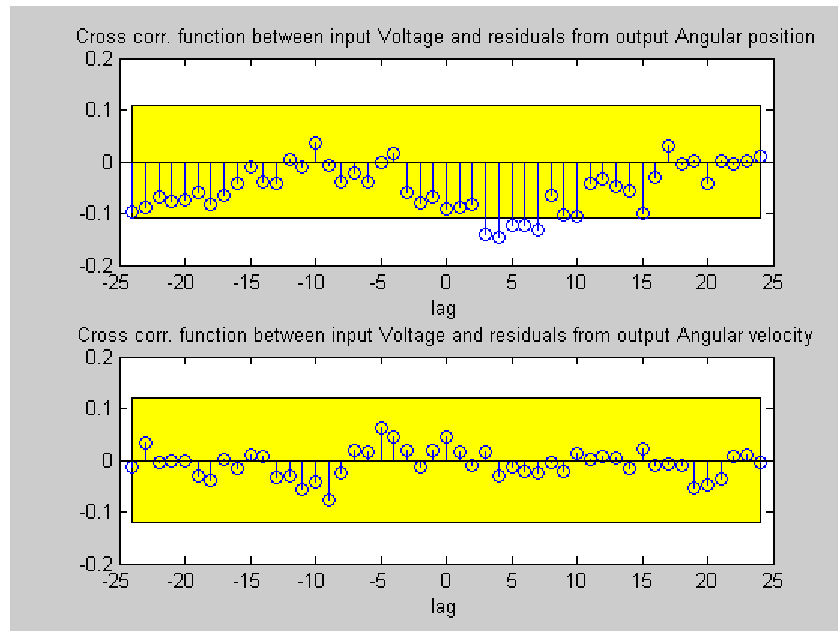


Figure 7: Residuals obtained with the estimated IDNLGREY DC-motor model.

6. Plot the step response.

A unit input step results in an angular position showing a ramp-type behavior and to an angular velocity that stabilizes at a constant level.

```
figure('Name', [nlgr.Name ': step response of estimated model']);
step(nlgr);
```

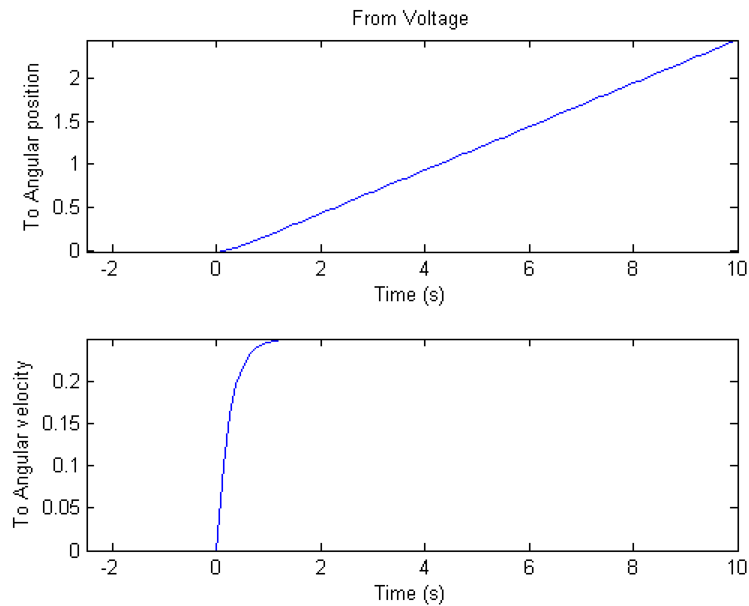


Figure 8: Step response with the estimated IDNLGREY DC-motor model.

7. Examine the model covariance.

You can assess the quality of the estimated model to some extent by looking at the estimated covariance matrix and the estimated noise variance. A "small" value of the `(i, i)` diagonal element of the covariance matrix indicates that the `i`:th model parameter is important for explaining the system dynamics when using the chosen model structure. Small noise variance (covariance for multi-output systems) elements are also a good indication that the model captures the estimation data in a good way.

```
nlgr.CovarianceMatrix
nlgr.NoiseVariance


ans =

  1.0e-004 *
```

```
    0.1521    0.0015
    0.0015    0.0007


ans =

    0.0099   -0.0004
   -0.0004    0.1094
```

For more information about the estimated model, use `present` to display the initial states and estimated parameter values, and estimated uncertainty (standard deviation) for the parameters.

```
present(nlgr);

Time-continuous nonlinear state-space model defined by 'dcmotor_m' (MATLA

   dx/dt = F(t, u(t), x(t), p1, p2)
    y(t) = H(t, u(t), x(t), p1, p2) + e(t)

with 1 input, 2 states, 2 outputs, and 2 free parameters (out of 2).

Input:
   u(1)  Voltage(t) [V]
States:                                initial value
   x(1)  Angular position(t) [rad]     xinit@exp1   0.0302986   (est) in
   x(2)  Angular velocity(t) [rad/s]   xinit@exp1   -0.133728   (est) in
Outputs:
   y(1)  Angular position(t) [rad]
   y(2)  Angular velocity(t) [rad/s]
Parameters:                           value      standard dev
   p1   Time-constant [s]           0.243646   0.00390033   (est) in [-Inf,
   p2   Static gain [rad/(V*s)]     0.249645   0.00027217   (est) in [-Inf,

The model was estimated from the data set 'DC-motor', which
contains 400 data samples.
Loss function 0.00107462 and Akaike's FPE 0.00108536
```

```
Created:       09-Jul-2011 07:31:06
Last modified: 09-Jul-2011 07:31:16
```

Conclusions

This example illustrates the basic tools for performing nonlinear grey-box modeling. See the other nonlinear grey-box examples to learn about:

- Using nonlinear grey-box models in more advanced modeling situations, such as building nonlinear continuous- and discrete-time, time-series and static models.
- Writing and using C MEX model-files.
- Handling nonscalar parameters.
- Impact of certain algorithm choices.

For more information on identification of dynamic systems with System Identification Toolbox, visit the System Identification Toolbox product information page.

Nonlinear Grey-Box Demos and Examples

The System Identification Toolbox product provides several demos and case studies on creating, manipulating, and estimating nonlinear grey-box models. You can access these demos by typing the following command at the prompt:

```
iddemo
```

For examples of code files and MEX-files that specify model structure, see the `toolbox/ident/iddemos/examples` folder. For example, the model of a DC motor—used in the demo `idnlgreydemo1`—is described in files `dcmotor_m` and `dcmotor_c`.

After Estimating Grey-Box Models

After estimating linear and nonlinear grey-box models, you can simulate the model output using the `sim` command. For more information, see "Validating Models After Estimation" on page 8-3.

The toolbox represents linear grey-box models using the `idgrey` model object. To convert grey-box models to state-space form, use the `idss` command, as described in "Transforming Between Linear Model Representations" on page 3-117. You can then analyze the model behavior using transient- and frequency-response plots and other linear analysis plots.

The toolbox represents nonlinear grey-box models as `idnlgrey` model objects. These model objects store the parameter values resulting from the estimation. You can access these parameters from the model objects to use these variables in computation in the MATLAB workspace.

Note Linearization of nonlinear grey-box models is not supported.

You can import grey box models into a Simulink model using the System Identification Toolbox Block Library. For more information, see "Simulating Identified Model Output in Simulink" on page 10-5.

6

Time Series Identification

- "What Are Time-Series Models?" on page 6-2
- "Preparing Time-Series Data" on page 6-3
- "Estimating Time-Series Power Spectra" on page 6-4
- "Estimating AR and ARMA Models" on page 6-7
- "Estimating State-Space Time-Series Models" on page 6-12
- "Example – Identifying Time-Series Models at the Command Line" on page 6-14
- "Estimating Nonlinear Models for Time-Series Data" on page 6-15

What Are Time-Series Models?

A *time series* is one or more measured output channels with no measured input.

You can estimate time-series spectra using both time- and frequency-domain data. Time-series spectra describe time-series variations using cyclic components at different frequencies.

You can also estimate parametric autoregressive (AR), autoregressive and moving average (ARMA), and state-space time-series models. For a definition of these models, see "Definition of AR and ARMA Models" on page 6-7, and "Definition of State-Space Time-Series Model" on page 6-12.

Note ARMA and state-space models are supported for time-domain data only. Only single-output ARMA models are supported.

Preparing Time-Series Data

Before you can estimate models for time-series data, you must import your data into the MATLAB software. You can estimate models from either time-domain and frequency-domain data. For information about which variables you need to represent time-series data, see "Time-Series Data Representation" on page 2-10.

For more information about preparing data for modeling, see "Ways to Prepare Data for System Identification" on page 2-6.

If your data is already in the MATLAB workspace, you can import it directly into the System Identification Tool GUI. If you prefer to work at the command line, you must represent the data as a System Identification Toolbox data object instead.

In the System Identification Tool GUI. When you import scalar or multiple-output time series data into the GUI, leave the **Input** field empty. For more information about importing data, see "Importing Data into the GUI" on page 2-17.

At the command line. To represent a time series vector or a matrix `s` as an `iddata` object, use the following syntax:

```
y = iddata(s,[],Ts);
```

`s` contains as many columns as there are measured outputs. For time-domain data, set `Ts` to the sampling interval. For continuous-time frequency domain data, set `Ts` to 0.

Estimating Time-Series Power Spectra

In this section...

"How to Estimate Time-Series Power Spectra Using the GUI" on page 6-4

"How to Estimate Time-Series Power Spectra at the Command Line" on page 6-5

How to Estimate Time-Series Power Spectra Using the GUI

You must have already imported your data into the GUI, as described in "Preparing Time-Series Data" on page 6-3.

To estimate time-series spectral models in the System Identification Tool GUI:

1. In the System Identification Tool GUI, 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 3-5.

3. Specify the frequencies at which to compute the spectral model in either 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 3-6. To use the default value, enter `default` or leave the field empty.

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 Tool GUI.

7 In the Spectral Model dialog box, click **Close**.

8 To view the estimated disturbance spectrum, select the **Noise spectrum** check box in the System Identification Tool GUI. For more information about working with this plot, see “Noise Spectrum Plots” on page 8-53.

To export the model to the MATLAB workspace, drag it to the **To Workspace** rectangle in the System Identification Tool GUI. You can view the power spectrum and the confidence intervals of the resulting `idfrd` model object using the `bode` command.

How to Estimate Time-Series Power Spectra at the Command Line

You can use the `etfe`, `spa`, and `spafdr` commands to estimate power spectra of time series for both time-domain and frequency-domain data. The following table provides a brief description of each command.

You must have already prepared your data, as described in “Preparing Time-Series Data” on page 6-3.

The resulting models are stored as an `idfrd` model object, which contains `SpectrumData` and its variance. For multiple-output data, `SpectrumData` contains power spectra of each output and the cross-spectra between each output pair.

Estimating Frequency Response of Time Series

Command	Description
etfe	Estimates a periodogram using Fourier analysis.
spa	Estimates the power spectrum with its standard deviation using spectral analysis.
spafdr	Estimates the power spectrum with its standard deviation using a variable frequency resolution.

For example, suppose y is time-series data. The following commands estimate the power spectrum g and the periodogram p, and plot both models with three standard deviation confidence intervals:

```
g = spa(y)
p = etfe(y)
bode(g,p,'sd',3)
```

For detailed information about these commands, see the corresponding reference pages.

Estimating AR and ARMA Models

In this section...
"Definition of AR and ARMA Models" on page 6-7
"Estimating Polynomial Time-Series Models in the GUI" on page 6-7
"Estimating AR and ARMA Models at the Command Line" on page 6-10

Definition of AR and ARMA Models

For a single-output signal *y(t)*, the AR model is given by the following equation:

$$A(q)y(t) = e(t)$$

The AR model is a special case of the ARX model with no input.

The ARMA model for a single-output time-series is given by the following equation:

$$A(q)y(t) = C(q)e(t)$$

The ARMA structure reduces to the AR structure for *C(q)*=1. The ARMA model is a special case of the ARMAX model with no input.

For more information about polynomial models, see "What Are Black-Box Polynomial Models?" on page 3-39.

Estimating Polynomial Time-Series Models in the GUI

Before you begin, you must have accomplished the following:

- Prepared the data, as described in "Preparing Time-Series Data" on page 6-3
- Estimated model order, as described in "Preliminary Step – Estimating Model Orders and Input Delays" on page 3-48

- (Multiple-output AR models only) Specified the model-order matrix in the MATLAB workspace before estimation, as described in "Estimating Multiple-Input and Multiple-Output ARX Orders" on page 3-64

To estimate AR and ARMA models using the System Identification Tool GUI:

1. In the System Identification Tool GUI, select **Estimate > Linear parametric models** to open the Linear Parametric Models dialog box.

2. In the **Structure** list, select the polynomial model structure you want to estimate from the following options:

   - `AR:[na]`
   - `ARMA:[na nc]`

   This action updates the options in the Linear Parametric Models dialog box to correspond with this model structure. For information about each model structure, see "Definition of AR and ARMA Models" on page 6-7.

   Note OE and BJ structures are not available for time-series models.

   ---

3. In the **Orders** field, specify the model orders, as follows:

   - **For single-output models.** Enter the model orders according to the sequence displayed in the **Structure** field.
   - **For multiple-output ARX models.** (AR models only) Enter the model orders directly, as described in "Estimating Multiple-Input and Multiple-Output ARX Orders" on page 3-64. Alternatively, enter the name of the matrix `NA` in the MATLAB Workspace browser that stores model orders, which is `Ny`-by-`Ny`.

   Tip To enter model orders and delays using the Order Editor dialog box, click **Order Editor**.

   ---

4 (AR models only) Select the estimation **Method** as **ARX** or **IV** (instrumental variable method). For more information about these methods, see "Polynomial Model Estimation Algorithms" on page 3-66.

Note **IV** is not available for multiple-output data.

---

5 In the **Name** field, edit the name of the model or keep the default. The name of the model should be unique in the Model Board.

6 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 3-66.

Tip If you get an inaccurate fit, try setting a specific method for handling initial states rather than choosing it automatically.

---

7 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 might reduce computation time for complex models and large data sets.

8 (ARMA only) To view the estimation progress at the command line, select the **Trace** check box. During estimation, the following information is displayed for each iteration:

- Loss function — Equals the determinant of the estimated covariance matrix of the input noise.
- Parameter values — Values of the model structure coefficients you specified.
- Search direction — Changes in parameter values from the previous iteration.

- Fit improvements — Shows the actual versus expected improvements in the fit.

9. Click **Estimate** to add this model to the Model Board in the System Identification Tool GUI.

10. (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.

11. To plot the model, select the appropriate check box in the **Model Views** area of the System Identification Tool GUI. For more information about validating models, see Chapter 8, "Model Analysis".

You can export the model to the MATLAB workspace for further analysis by dragging it to the **To Workspace** rectangle in the System Identification Tool GUI.

Estimating AR and ARMA Models at the Command Line

You can estimate AR and ARMA models at the command line. For single-output time-series, the resulting models are `idpoly` model objects. For multiple-output time-series, the resulting models are `idarx` model objects. For more information about models objects, see "Creating Model Structures at the Command Line" on page 1-14.

The following table summarizes the commands and specifies whether single-output or multiple-output models are supported.

Commands for Estimating Polynomial Time-Series Models

Method Name	Description	Supported Data
ar	Noniterative, least-squares method to estimate linear, discrete-time single-output AR models.	Time-domain, time-series iddata data object.
armax	Iterative prediction-error method to estimate linear, single-output ARMAX models.	Time-domain, time-series iddata data object.
arx	Noniterative, least-squares method for estimating single-output and multiple-output linear AR models.	Supports time- and frequency-domain time-series iddata data.
ivar	Noniterative, instrumental variable method for estimating single-output AR models.	Supports time-domain, time-series iddata data.

The following code shows usage examples for estimating AR models:

```
% For scalar signals
m = ar(y,na)
% For multiple-output vector signals
m = arx(y,na)
% Instrumental variable method
m = ivar(y,na)
% For ARMA, do not need to specify nb and nk
th = armax(y,[na nc])
```

The `ar` command provides additional options to let you choose the algorithm for computing the least-squares from a group of several popular techniques from the following methods:

- Burg (geometric lattice)
- Yule-Walker
- Covariance

For more information about validating models, see “Validating Models After Estimation” on page 8-3.

Estimating State-Space Time-Series Models

In this section...

"Definition of State-Space Time-Series Model" on page 6-12
"Estimating State-Space Models at the Command Line" on page 6-12

Definition of State-Space Time-Series Model

The discrete-time state-space model for a time series is given by the following equations:

$$x(kT+T) = Ax(kT) + Ke(kT)$$
$$y(kT) = Cx(kT) + e(kT)$$

where T is the sampling interval and $y(kT)$ is the output at time instant kT.

The time-series structure corresponds to the general structure with empty B and D matrices.

For information about general discrete-time and continuous-time structures for state-space models, see "What Are State-Space Models?" on page 3-73.

Estimating State-Space Models at the Command Line

You can estimate single-output and multiple-output state-space models at the command line for time-domain and frequency-domain data (`iddata` object).

The following table provides a brief description of each command. The resulting models are `idss` model objects.

Commands for Estimating State-Space Time-Series Models

Command	Description
`n4sid`	Noniterative subspace method for estimating discrete-time linear state-space models. **Note** When you use `pem` to estimate a state-space model, `n4sid` creates the initial model.
`pem`	Estimates linear, discrete-time time-series models using an iterative estimation method that minimizes the prediction error.

Example – Identifying Time-Series Models at the Command Line

The following example simulates a time-series model, compares spectral estimates, covariance estimates, and predicts output of the model:

```
ts0 = idpoly([1 -1.5 0.7],[]);
ir = sim(ts0,[1;zeros(24,1)]);
% Define the true covariance function
Ry0 = conv(ir,ir(25:-1:1));
e = idinput(200,'rgs');
% Define y vector
y = sim(ts0,e);
% iddata object with sampling time 1
y = iddata(y)
plot(y)
per = etfe(y);
speh = spa(y);
ffplot(per,speh,ts0)
% Estimate a second-order AR model
ts2 = ar(y,2);
ffplot(speh,ts2,ts0,'sd',3)
% Get covariance function estimates
Ryh = covf(y,25);
Ryh = [Ryh(end:-1:2),Ryh]';
ir2 = sim(ts2,[1;zeros(24,1)]);
Ry2 = conv(ir2,ir2(25:-1:1));
plot([-24:24]'*ones(1,3),[Ryh,Ry2,Ry0])
% The prediction ability of the model
compare(y,ts2,5)
```

Estimating Nonlinear Models for Time-Series Data

When a linear model provides an insufficient description of the dynamics, you can try estimating a nonlinear models. To learn more about when to estimate nonlinear models, see "Building Models from Data" in the Getting Started Guide.

Before you can estimate models for time-series data, you must have already prepared the data as described in "Preparing Time-Series Data" on page 6-3.

For black-box modeling of time-series data, the toolbox supports nonlinear ARX models. To learn how to estimate this type of model, see "Identifying Nonlinear ARX Models" on page 4-8.

If you understand the underlying physics of the system, you can specify an ordinary differential or difference equation and estimate the coefficients. To learn how to estimate this type of model, see "Estimating Nonlinear Grey-Box Models" on page 5-15.

For more information about validating models, see "Validating Models After Estimation" on page 8-3.

Recursive Model Identification

- “What Is Recursive Estimation?” on page 7-2
- “Commands for Recursive Estimation” on page 7-3
- “Algorithms for Recursive Estimation” on page 7-6
- “Data Segmentation” on page 7-14

What Is Recursive Estimation?

Many real-world applications, such as adaptive control, adaptive filtering, and adaptive prediction, require a model of the system to be available online while the system is in operation. Estimating models for batches of input-output data is useful for addressing the following types of questions regarding system operation:

- Which input should be applied at the next sampling instant?
- How should the parameters of a matched filter be tuned?
- What are the predictions of the next few outputs?
- Has a failure occurred? If so, what type of failure?

You might also use online models to investigate time variations in system and signal properties.

The methods for computing online models are called *recursive identification methods*. Recursive algorithms are also called *recursive parameter estimation*, *adaptive parameter estimation*, *sequential estimation*, and *online* algorithms.

For examples of recursive estimation and data segmentation, run the Recursive Estimation and Data Segmentation demo by typing the following command at the prompt:

```
iddemo5
```

For detailed information about recursive parameter estimation algorithms, see the corresponding chapter in *System Identification: Theory for the User* by Lennart Ljung (Prentice Hall PTR, Upper Saddle River, NJ, 1999).

Commands for Recursive Estimation

You can recursively estimate linear polynomial models, such as ARX, ARMAX, Box-Jenkins, and Output-Error models. If you are working with time-series data that contains no inputs and a single output, you can estimate AR (autoregressive) and ARMA (autoregressive and moving average) single-output models.

Before estimating models using recursive algorithms, you must import your data into the MATLAB workspace and represent your data in either of the following formats:

- Matrix of the form `[y u]`. `y` represents the output data using one or more column vectors. Similarly, `u` represents the input data using one or more column vectors.
- `iddata` or `idfrd` object. For more information about creating these objects, see Chapter 2, “Data Import and Processing”.

The general syntax for recursive estimation commands is as follows:

```
[params,y_hat]=command(data,nn,adm,adg)
```

`params` matrix contains the values of the estimated parameters, where the `k`th row contains the parameters associated with time `k`, which are computed using the `data` values in the rows up to and including the row `k`.

`y_hat` contains the predicted output values such that the `k`th row of `y_hat` is computed based on the `data` values in the rows up to and including the row `k`.

Tip `y_hat` contains the adaptive predictions of the output and is useful for adaptive filtering applications, such as noise cancelation.

`nn` specified the model orders and delay according to the specific polynomial structure of the model. For example, `nn=[na nb nk]` for ARX models. For more information about specifying polynomial model orders and delays, see “Identifying Input-Output Polynomial Models” on page 3-39.

adm and adg specify any of the four recursive algorithm, as described in "Algorithms for Recursive Estimation" on page 7-6.

The following table summarizes the recursive estimation commands supported by the System Identification Toolbox product. The command description indicates whether you can estimate single-input, single-output, multiple-input, and multiple-output, and time-series (no input) models. For details about each command, see the corresponding reference page.

Tip For ARX and AR models, use rarx. For single-input/single-output ARMAX or ARMA, Box-Jenkins, and Output-Error models, use rarmax, rbj, and roe, respectively.

Commands for Linear Recursive Estimation

Command	Description
rarmax	Estimate parameters of single-input/single-output ARMAX and ARMA models.
rarx	Estimate parameters of single- or multiple-input and single-output ARX and AR models. Does not support multiple-output system.
rbj	Estimate parameters of single-input/single-output Box-Jenkins models.
roe	Estimate parameters of single-input/single-output Output-Error models.

Commands for Linear Recursive Estimation (Continued)

Command	Description
`rpem`	Estimate parameters of multiple-input and single-output ARMAX/ARMA, Box-Jenkins, or Output-Error models using the general recursive prediction-error algorithm for estimating the parameter gradient. **Note** Unlike `pem`, `rpem` does not support state-space models.
`rplr`	Use as an alternative to `rpem` to estimate parameters of multiple-input and single-output systems when you want to use recursive pseudolinear regression method.

Algorithms for Recursive Estimation

In this section...

"Types of Recursive Estimation Algorithms" on page 7-6
"General Form of Recursive Estimation Algorithm" on page 7-6
"Kalman Filter Algorithm" on page 7-8
"Forgetting Factor Algorithm" on page 7-10
"Unnormalized and Normalized Gradient Algorithms" on page 7-11

Types of Recursive Estimation Algorithms

You can choose from the following four recursive estimation algorithms:

- "General Form of Recursive Estimation Algorithm" on page 7-6
- "Kalman Filter Algorithm" on page 7-8
- "Forgetting Factor Algorithm" on page 7-10
- "Unnormalized and Normalized Gradient Algorithms" on page 7-11

You specify the type of recursive estimation algorithms as arguments `adm` and `adg` of the recursive estimation commands in "Commands for Recursive Estimation" on page 7-3.

For detailed information about these algorithms, see the corresponding chapter in *System Identification: Theory for the User* by Lennart Ljung (Prentice Hall PTR, Upper Saddle River, NJ, 1999).

General Form of Recursive Estimation Algorithm

The general recursive identification algorithm is given by the following equation:

$$\hat{\theta}(t) = \hat{\theta}(t-1) + K(t)\big(y(t) - \hat{y}(t)\big)$$

$\hat{\theta}(t)$ 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 general 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 $\hat{y}(t\,|\,\theta)$ 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 *System Identification: Theory for the User* by Lennart Ljung (Prentice Hall PTR, Upper Saddle River, NJ, 1999).

Kalman Filter Algorithm

- “Mathematics of the Kalman Filter Algorithm” on page 7-8
- “Using the Kalman Filter Algorithm” on page 7-9

Mathematics of the Kalman Filter Algorithm

The following set of equations summarizes the *Kalman filter* adaptation algorithm:

$$\hat{\theta}(t) = \hat{\theta}(t-1) + K(t)\left(y(t) - \hat{y}(t)\right)$$

$$\hat{y}(t) = \psi^T(t)\hat{\theta}(t-1)$$

$$K(t) = Q(t)\psi(t)$$

$$Q(t) = \frac{P(t-1)}{R_2 + \psi(t)^T P(t-1)\psi(t)}$$

$$P(t) = P(t-1) + R_1 - \frac{P(t-1)\psi(t)\psi(t)^T P(t-1)}{R_2 + \psi(t)^T P(t-1)\psi(t)}$$

This formulation assumes the linear-regression form of the model:

$$y(t) = \psi^T(t)\theta_0(t) + e(t)$$

The Kalman filter is used to obtain *Q(t)*.

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 To simplify the inputs, you can scale R_1, R_2, and $P(t=0)$ of the original problem by the same value such that R_2 is equal to 1. This scaling does not affect the parameters estimates.

Using the Kalman Filter Algorithm

The general syntax for the command described in "Algorithms for Recursive Estimation" on page 7-6 is the following:

```
[params,y_hat]=command(data,nn,adm,adg)
```

To specify the Kalman filter algorithm, set `adm` to `'kf'` and `adg` to the value of the drift matrix R_1 (described in "Mathematics of the Kalman Filter Algorithm" on page 7-8).

Forgetting Factor Algorithm

- "Mathematics of the Forgetting Factor Algorithm" on page 7-10
- "Using the Forgetting Factor Algorithm" on page 7-11

Mathematics of the Forgetting Factor Algorithm

The following set of equations summarizes the *forgetting factor* adaptation algorithm:

$$\hat{\theta}(t) = \hat{\theta}(t-1) + K(t)\left(y(t) - \hat{y}(t)\right)$$

$$\hat{y}(t) = \psi^T(t)\hat{\theta}(t-1)$$

$$K(t) = Q(t)\psi(t)$$

$$Q(t) = P(t) = \frac{P(t-1)}{\lambda + \psi(t)^T P(t-1)\psi(t)}$$

$$P(t) = \frac{1}{\lambda}\left(P(t-1) - \frac{P(t-1)\psi(t)\psi(t)^T P(t-1)}{\lambda + \psi(t)^T P(t-1)\psi(t)}\right)$$

To obtain *Q(t)*, the following function is minimized at time *t*:

$$\sum_{k=1}^{t} \lambda^{t-k} e^2(k)$$

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 $\tau = \frac{1}{1-\lambda}$ typically carry a weight that is less than about 0.3.

λ is called the forgetting factor and typically has a positive value between `0.97` and `0.995`.

Note In the linear regression case, the forgetting factor algorithm is known as the *recursive least-squares* (RLS) algorithm. 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 Algorithm” on page 7-8.

Using the Forgetting Factor Algorithm

The general syntax for the command described in “Algorithms for Recursive Estimation” on page 7-6 is the following:

```
[params,y_hat]=command(data,nn,adm,adg)
```

To specify the forgetting factor algorithm, set `adm` to `'ff'` and `adg` to the value of the forgetting factor λ (described in “Mathematics of the Forgetting Factor Algorithm” on page 7-10).

Tip λ typically has a positive value from `0.97` to `0.995`.

Unnormalized and Normalized Gradient Algorithms

- “Mathematics of the Unnormalized and Normalized Gradient Algorithm” on page 7-12
- “Using the Unnormalized and Normalized Gradient Algorithms” on page 7-12

Mathematics of the Unnormalized and Normalized Gradient Algorithm

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:

$$\hat{\theta}(t) = \hat{\theta}(t-1) + K(t)\big(y(t) - \hat{y}(t)\big)$$

$$\hat{y}(t) = \psi^T(t)\hat{\theta}(t-1)$$

$$K(t) = Q(t)\psi(t)$$

In the unnormalized gradient approach, *Q(t)* is the product of the gain γ and the identity matrix:

$$Q(t) = \gamma I$$

In the normalized gradient approach, *Q(t)* is the product of the gain γ, and the identity matrix is normalized by the magnitude of the gradient $\psi(t)$:

$$Q(t) = \frac{\gamma}{\left|\psi(t)\right|^2} I$$

These choices of *Q(t)* update the parameters in the negative gradient direction, where the gradient is computed with respect to the parameters.

Using the Unnormalized and Normalized Gradient Algorithms

The general syntax for the command described in "Algorithms for Recursive Estimation" on page 7-6 is the following:

```
[params,y_hat]=command(data,nn,adm,adg)
```

To specify the unnormalized gain algorithm, set `adm` to `'ug'` and `adg` to the value of the gain γ (described in "Mathematics of the Unnormalized and Normalized Gradient Algorithm" on page 7-12).

To specify the normalized gain algorithm, set `adm` to `'ng'` and `adg` to the value of the gain γ.

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.

To see an example of using data segmentation, run the Recursive Estimation and Data Segmentation demonstration by typing to the following command at the prompt:

```
iddemo5
```

8

Model Analysis

- "Validating Models After Estimation" on page 8-3
- "Plotting Models in the GUI" on page 8-8
- "Getting Advice About Models" on page 8-10
- "Simulating and Predicting Model Output" on page 8-11
- "Residual Analysis" on page 8-26
- "Impulse and Step Response Plots" on page 8-35
- "How to Plot Impulse and Step Response Using the GUI" on page 8-39
- "How to Plot Impulse and Step Response at the Command Line" on page 8-42
- "Frequency Response Plots" on page 8-44
- "How to Plot Bode Plots Using the GUI" on page 8-48
- "How to Plot Bode and Nyquist Plots at the Command Line" on page 8-51
- "Noise Spectrum Plots" on page 8-53
- "How to Plot the Noise Spectrum Using the GUI" on page 8-56
- "How to Plot the Noise Spectrum at the Command Line" on page 8-59
- "Pole and Zero Plots" on page 8-61
- "How to Plot Model Poles and Zeros Using the GUI" on page 8-65
- "How to Plot Poles and Zeros at the Command Line" on page 8-67
- "Akaike's Criteria for Model Validation" on page 8-68
- "Computing Model Uncertainty" on page 8-71
- "Troubleshooting Models" on page 8-74

- “Next Steps After Getting an Accurate Model” on page 8-79

Validating Models After Estimation

In this section...

"When to Validate Models" on page 8-3

"Ways to Validate Models" on page 8-3

"Data for Model Validation" on page 8-4

"Supported Model Plots" on page 8-5

"Definition: Confidence Interval" on page 8-6

When to Validate Models

After estimating each model, you can validate whether the model reproduces system behavior within acceptable bounds. You iterate between estimation and validation until you find the simplest model that best captures the system dynamics.

For ideas on how to adjust your modeling strategy based on validation results, see "Troubleshooting Models" on page 8-74.

Tip If you have installed the Control System Toolbox product, you can also view models using the LTI Viewer. For more information, see "Viewing Model Response Using the LTI Viewer" on page 9-5.

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 8-11.

  To simulate identified models in the Simulink environment, see "Simulating Identified Model Output in Simulink" on page 10-5.

- Analyzing autocorrelation and cross-correlation of the residuals with input.

  See "Residual Analysis" on page 8-26.

- Analyzing model response. For more information, see the following:
  - "Impulse and Step Response Plots" on page 8-35
  - "Frequency Response Plots" on page 8-44

  For information about the response of the noise model, see "Noise Spectrum Plots" on page 8-53.

- Plotting the poles and zeros of the linear parametric model.

  For more information, see "Pole and Zero Plots" on page 8-61.

- 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.

  For more information, see Chapter 4, "Nonlinear Black-Box Model Identification".

Displaying confidence intervals on supported plots helps you assess the uncertainty of model parameters. For more information, see "Computing Model Uncertainty" on page 8-71.

Data for Model Validation

For plots that compare model response to measured response, such as model output and residual analysis plots, 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 overfitting 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 2-101.

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 8-11
Residual Analysis	All linear and nonlinear models	"Residual Analysis" on page 8-26
Transient Response	• All linear parametric models • Correlation analysis (nonparametric) models • For nonlinear models, only step response.	"Impulse and Step Response Plots" on page 8-35
Frequency Response	• All linear parametric models • Spectral analysis (nonparametric) models	"Frequency Response Plots" on page 8-44

Plot Type	Supported Models	Learn More
Noise Spectrum	• All linear parametric models • Spectral analysis (nonparametric) models	"Noise Spectrum Plots" on page 8-53
Poles and Zeros	All linear parametric models	"Pole and Zero Plots" on page 8-61
Nonlinear ARX	Nonlinear ARX models only	Nonlinear ARX Plots
Hammerstein-Wiener	Hammerstein-Wiener models only	Hammerstein-Wiener Plots

Definition: Confidence Interval

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
Residuals	Residual values with a specific probability of being statistically insignificant for the system.	Residuals Plots
Impulse and Step	Response values with a specific probability of being the actual response of the system.	Impulse and Step Plots

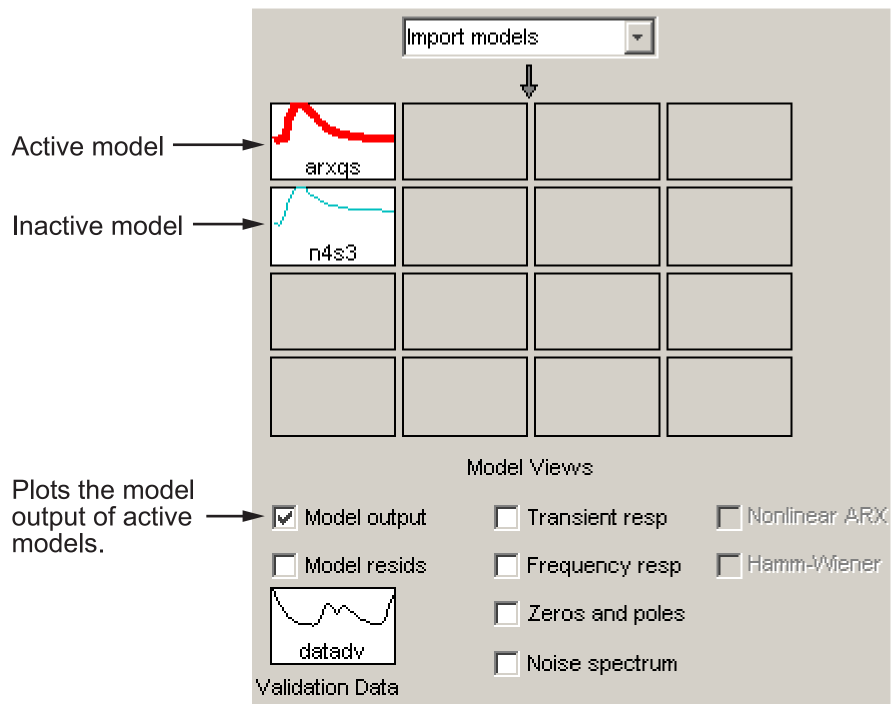
Plot Type	Confidence Interval Corresponds to the Range of ...	More Information on Displaying Confidence Interval
Frequency Response	Response values with a specific probability of being the actual response of the system.	Frequency Response Plots
Noise Spectrum	Power-spectrum values with a specific probability of being the actual noise spectrum of the system.	Noise Spectrum Plots
Poles and Zeros	Pole or zero values with a specific probability of being the actual pole or zero of the system.	Pole-Zero Plots

Plotting Models in the GUI

To create one or more plots of your models, select the corresponding check box in the **Model Views** area of the System Identification Tool GUI. 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 Tool GUI. 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 `n4s4` is not included on the plot because only `arx441` is active.



Plots Include Only Active Models

To close a plot, clear the corresponding check box in the System Identification Tool GUI.

Tip To get information about a specific plot, select a help topic from the **Help** menu in the plot window.

For general information about working with plots in the System Identification Toolbox product , see "Working with Plots" on page 11-13.

Getting Advice About Models

Use the `advice` command on an estimated model to answer the following questions about the model:

- Should I increase or decrease the model order?
- Should I estimate a noise model?
- Is feedback present?

Simulating and Predicting Model Output

In this section...

"Why Simulate or Predict Model Output" on page 8-11
"Definition: Simulation and Prediction" on page 8-12
"Simulation and Prediction in the GUI" on page 8-14
"Simulation and Prediction at the Command Line" on page 8-20
"Compare Simulated Output with Measured Data" on page 8-22
"Simulate Model Output with Noise" on page 8-23
"Simulate a Continuous-Time State-Space Model" on page 8-23
"Predict Using Time-Series Model" on page 8-25

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. For more information, see "Definition: Simulation and Prediction" on page 8-12.

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. For more information, see "Simulation and Prediction in the GUI" on page 8-14 and "Simulation and Prediction at the Command Line" on page 8-20.

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.

Related Examples

"Compare Simulated Output with Measured Data" on page 8-22

"Simulate Model Output with Noise" on page 8-23

"Simulate a Continuous-Time State-Space Model" on page 8-23

"Predict Using Time-Series Model" on page 8-25

See Also

Chapter 10, "System Identification Toolbox Blocks"—Simulate models in Simulink software.

Definition: 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 sampling interval 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 sampling interval.

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:

$$y_p(t+k) = f(u(t+k),u(t+k-1),...,u(t),u(t-1),...,u(0) \\ y(t),y(t-1),y(t-2),...,y(0))$$

$u(0)$ 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=∞ 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.

Simulation and Prediction in the GUI

- "How to Plot Simulated and Predicted Model Output" on page 8-14
- "Interpreting the Model Output Plot" on page 8-15
- "Changing Model Output Plot Settings" on page 8-17
- "Definition: Confidence Interval" on page 8-19

How to Plot Simulated and Predicted Model Output

To create a model output plot for parametric linear and nonlinear models in the System Identification Tool GUI, 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 Tool GUI. Active models display a thick line inside the Model Board icon.

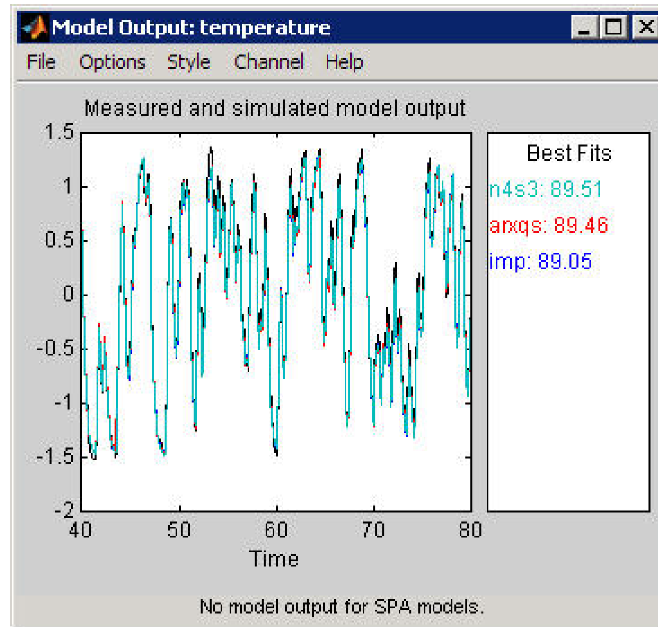
To learn how to interpret the model output plot, see “Interpreting the Model Output Plot” on page 8-15.

To change plot settings, see “Changing Model Output Plot Settings” on page 8-17.

For general information about creating and working with plots, see “Working with Plots” on page 11-13.

Interpreting the Model Output Plot

The following figure shows a sample Model Output plot, created in the System Identification Tool GUI.



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.

Changing Model Output Plot Settings

The following table summarizes the Model Output plot settings.

Model Output Plot Settings

Action	Command
Display confidence intervals.	• To display the dashed lines on either side of the nominal model

Model Output Plot Settings (Continued)

Action	Command
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 8-19.	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**.

Model Output Plot Settings (Continued)

Action	Command
(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 GUI, 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 "Changing Model Output Plot Settings" on page 8-17.

Simulation and Prediction at the Command Line

- "Summary of Simulation and Prediction Commands" on page 8-20
- "Initial States in Simulation and Prediction" on page 8-21

Summary of 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 the `retrend` command to the simulated or predicted output.

Command	Description	Example
`compare`	Use this command for model validation to 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 simulation.
`sim`	Simulate and plot the model output only.	To simulate the response of the model `model` using input data `data`, use the following command:

Command	Description	Example
		`sim(model,data)`

`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)`

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. Both `sim` and `predict` commands provide defaults for handling initial conditions.

Simulation: Default initial conditions are zero for polynomial (`idpoly` and `idarx`) models. For state-space (`idss`), low-order transfer functions (`idproc`), and linear grey-box (`idgrey`) models, the default initial conditions are the internal model initial states (model property `x0`). You can specify other initial conditions using the `InitialState` argument in `sim`, as described in the next example.

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 the `findstates` command and specify these initial states as input arguments to the `sim` command. 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
sim(m,z.InputData,'InitialState',X0est)
```

See Also: `sim` (for linear models), `sim(idnlarx)`, `sim(idnlgrey)`, `sim(idnlhw)`

Prediction: Default initial conditions depend on the type of model. You can specify other initial conditions using the `InitialState` argument in `predict`. For example, to compute the initial states that optimize the 1-step-ahead predicted response of the model `m` to the output data `z`:

```
[Yp,X0est] = predict(m,z,1,'InitialState','Estimate')
```

This command returns the estimated initial states as the output argument `X0est`. For information about other ways to specify initials states, see the `predict` reference page for the corresponding model type.

See Also: `predict` (for linear models), `predict(idnlarx)`, `predict(idnlgrey)`, `predict(idnlhw)`

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.
 data1
ze = z1(1:150);
zv = z1(151:300);
% Estimate model.
m= armax(ze,[2 3 1 0]);
```

```
% Validate model.
compare(zv,m);
```

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 case, you use the following ARMAX model with Gaussian noise e:

$$y(t) - 1.5y(t-1) + 0.7y(t-2) =$$
$$u(t-1) + 0.5u(t-2) + e(t) - e(t-1) + 0.2e(t-1)$$

Create the ARMAX model and simulate output data with random binary input `u` using the following commands:

```
% 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
  y = sim(m_armax,u,'noise');
```

Note The argument `'noise'` specifies to include in the simulation the Gaussian noise e present in the model. Omit this argument to simulate the noise-free response to the input `u`, which is equivalent to setting e to zero.

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 sampling interval 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`.

Use the following commands to simulate the model:

```
% Set up the model matrices
  A = [-1 1;-0.5 0]; B = [1; 0.5];
  C = [1 0]; D = 0; K = [0.5;0.5];
% Create a continuous-time state-space model
% 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
  data = iddata([],u);
  data.ts = 0.1
% Simulate the output using the model
  y=sim(model_ss,data,'noise');
```

Note The argument `'noise'` specifies to simulate with the Gaussian noise e present in the model. Omit this argument to simulate the noise-free response to the input `u`, which is equivalent to setting e to zero.

Predict Using Time-Series Model

This example shows how to evaluate how well a time-series model predicts future values.

In this example, y is the original series of monthly sales figures. You use the first half of the measured data to estimate the time-series model and test the model's ability to forecast sales six months ahead using the entire data set.

```
% Select the first half of the data for estimation
% y1 = y(1:48)
% Estimate a fourth-order autoregressive model
% using the first half of the data.
m = ar(y1,4)
% Compute 6-step ahead prediction
yhat = predict(m,y,6)
% Plot the predicted and measured outputs
plot(y,yhat)
```

Residual Analysis

In this section...

"What Is Residual Analysis?" on page 8-26
"Supported Model Types" on page 8-27
"What Residual Plots Show for Different Data Domains" on page 8-27
"Displaying the Confidence Interval" on page 8-28
"How to Plot Residuals Using the GUI" on page 8-29
"How to Plot Residuals at the Command Line" on page 8-31
"Example – Examining Model Residuals" on page 8-31

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 8-27.

Note For nonparametric models, including impulse-response, step-response, and frequency-response models, residual analysis plots are not available. 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

---

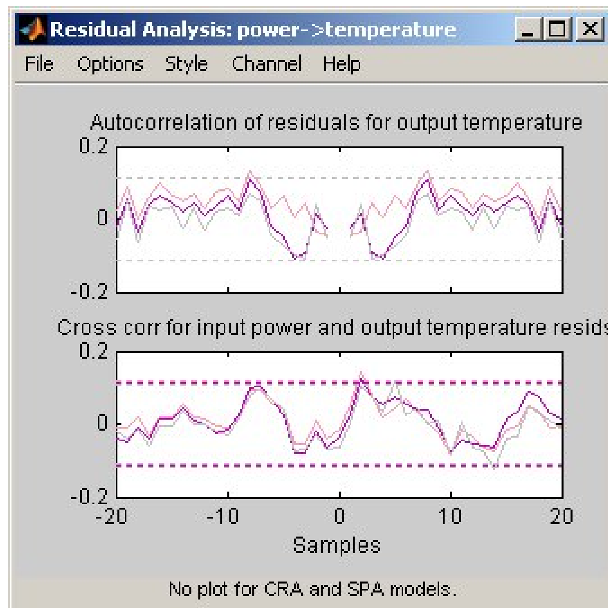
Note For time-series models, the residual analysis plot does not provide any input-residual correlation plots.

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 Tool GUI.



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 GUI 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 Using the GUI" on page 8-29.

Note If you are working in the System Identification Tool GUI, you can specify a custom confidence interval. If you are using the `resid` command, the confidence interface is fixed at 99%.

How to Plot Residuals Using the GUI

To create a residual analysis plot for parametric linear and nonlinear models in the System Identification Tool GUI, 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 11-13.

To include or exclude a model on the plot, click the corresponding model icon in the System Identification Tool GUI. 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.

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(model,data)`

Example – Examining 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 Tool session that contains estimated models, type the following command in the MATLAB Command Window:

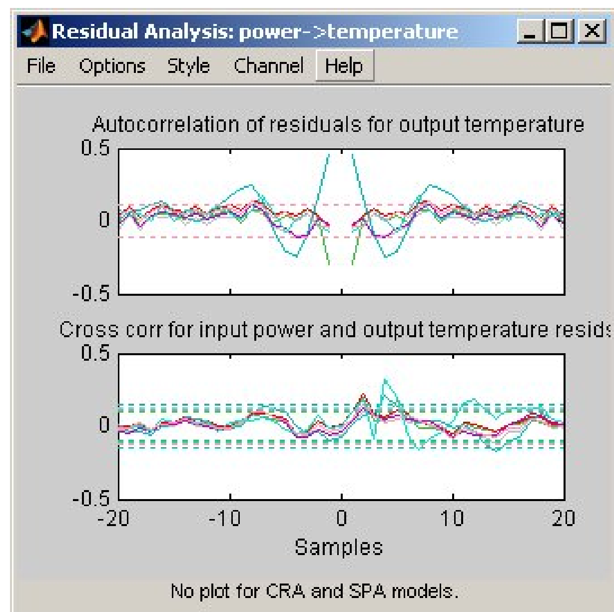
   ```
   ident('dryer2_linear_models')
   ```

2. To generate a residual analysis plot, select the **Model resids** check box in the System Identification Tool GUI.

This opens an empty plot.

3 In the System Identification Tool 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

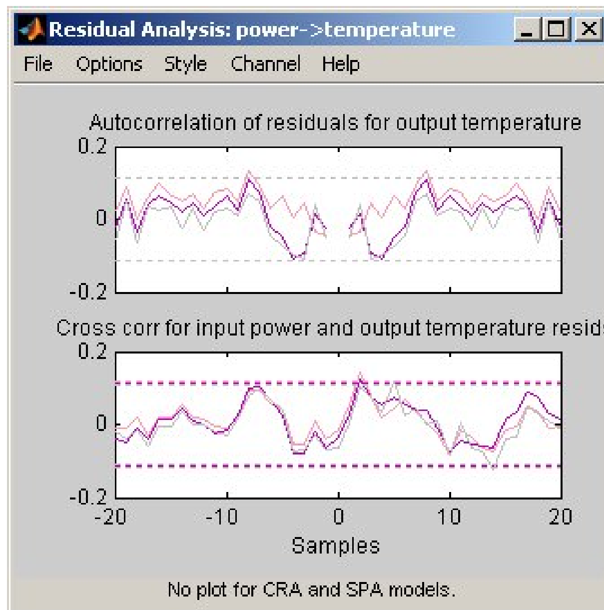
model should have a residual autocorrelation function within the confidence interval, indicating that the residuals are uncorrelated.

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 Tool GUI.

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.

Impulse and Step Response Plots

In this section...

"Supported Models" on page 8-35
"How Transient Response Helps to Validate Models" on page 8-35
"What Does a Transient Response Plot Show?" on page 8-36
"Displaying the Confidence Interval" on page 8-37

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

"How to Plot Impulse and Step Response Using the GUI" on page 8-39

"How to Plot Impulse and Step Response at the Command Line" on page 8-42

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.

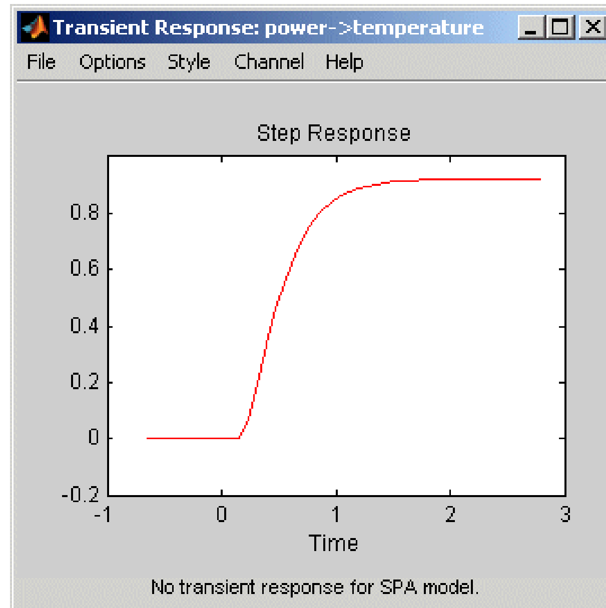
The impulse response of a dynamic model is the output signal that results when the input is an impulse. That is, $u(t)$ is zero for all values of t except at $t=0$, where $u(0)=1$. In the following difference equation, you can compute the impulse response by setting $y(-T)=y(-2T)=0$, $u(0)=1$, and $u(t>0)=0$.

$$y(t) - 1.5y(t-T) + 0.7y(t-2T) = 0.9u(t) + 0.5u(t-T)$$

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 "How to Plot Impulse and Step Response Using the GUI" on page 8-39.

The following figure shows a sample Transient Response plot, created in the System Identification Tool GUI.



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 "How to Plot Impulse and Step Response Using the GUI" on page 8-39.

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 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.

How to Plot Impulse and Step Response Using the GUI

To create a transient analysis plot in the System Identification Tool GUI, 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 11-13.

To include or exclude a model on the plot, click the corresponding model icon in the System Identification Tool GUI. 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

Transient Response Plot Settings (Continued)

Action	Command
	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.
Toggle between line plot or stem plot. **Tip** Use a stem plot for displaying impulse response.	Select **Style > Line plot** or **Style > Stem plot**.

Transient Response Plot Settings (Continued)

Action	Command
(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 8-35

How to Plot Impulse and Step Response at the Command Line

You can plot impulse- and step-response plots using the `impulse` and `step` commands, respectively.

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:

```
command(model,'sd',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.

To display a filled confidence region, use the following syntax:

```
command(model,'sd',sd,'fill')
```

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`	Plots impulse response for `idpoly`, `idproc`, `idarx`, `idss`, and `idgrey` model objects. Estimates and plots impulse response models for `iddata` objects. **Note** Does not support nonlinear models.	To plot the impulse response of the model `mod`, type the following command: `impulse(mod)`

`step`	Plots the step response of all linear and nonlinear models.
	
	Estimates and plots step response models for `iddata` objects.
		To plot the step response of the model `mod`, type the following command:
		
		`step(mod)`
		
		To specify step levels for a nonlinear model, type the following command:
		
		`step(mod, 'InputLevel',[u1;u2])`

More About

"Impulse and Step Response Plots" on page 8-35

Frequency Response Plots

In this section...

"What Is Frequency Response?" on page 8-44
"How Frequency Response Helps to Validate Models" on page 8-45
"What Does a Frequency-Response Plot Show?" on page 8-46
"Displaying the Confidence Interval" on page 8-47

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 sampling interval T to make the frequency function periodic with the sampling frequency $2\pi/T$.

Examples

"How to Plot Bode Plots Using the GUI" on page 8-48

"How to Plot Bode and Nyquist Plots at the Command Line" on page 8-51

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 parametric models and spectral analysis (nonparametric) 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.

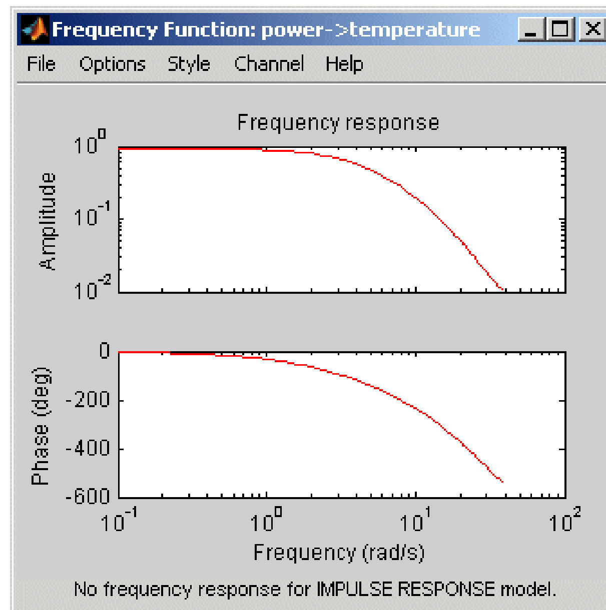
One example of how frequency-response plots help validate other models is that you can estimate a frequency response from the data using spectral analysis (nonparametric model), and then plot the spectral analysis result on top of the frequency response of the parametric models. Because nonparametric 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 Tool GUI 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.
- Bode 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 frequency response of the noise model instead. For more information, see "Noise Spectrum Plots" on page 8-53.
- (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 Tool GUI.



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 "How to Plot Bode Plots Using the GUI" on page 8-48

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 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.

How to Plot Bode Plots Using the GUI

To create a frequency-response plot for parametric linear models in the System Identification Tool GUI, 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 11-13.

To include or exclude a model on the plot, click the corresponding model icon in the System Identification Tool GUI. 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	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)`.

Frequency Function Plot Settings (Continued)

Action	Command
or equal to the Nyquist frequency.	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**.
(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 8-44

How to Plot Bode and Nyquist Plots at the Command Line

You can plot Bode and Nyquist plots for linear models using the `bode`, `ffplot`, and `nyquist` commands.

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:

```
command(model,'sd',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.

To display a filled confidence region, use the following syntax:

```
command(model,'sd',sd,'fill')
```

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`	Plots the magnitude and phase of the frequency response on a logarithmic frequency scale.	To create the bode plot of the model `mod`, use the following command: `bode(mod)`
`ffplot`	Plots the magnitude and phase of the frequency response on a linear frequency scale (hertz).	To create the bode plot of the model `mod`, use the following command: `ffplot(mod)`
`nyquist`	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 `mod`, use the following command: `nyquist(mod)`

More About

"Frequency Response Plots" on page 8-44

Noise Spectrum Plots

In this section...

"Supported Models" on page 8-53

"What Does a Noise Spectrum Plot Show?" on page 8-53

"Displaying the Confidence Interval" on page 8-54

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

"How to Plot the Noise Spectrum Using the GUI" on page 8-56

"How to Plot the Noise Spectrum at the Command Line" on page 8-59

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:

$$v(t) = H(z)e(t)$$

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. $e(t)$ is a white-noise source with variance λ.

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 "How to Plot the Noise Spectrum Using the GUI" on page 8-56.

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 of values that have a 95% probability of being the true system noise spectrum. 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.

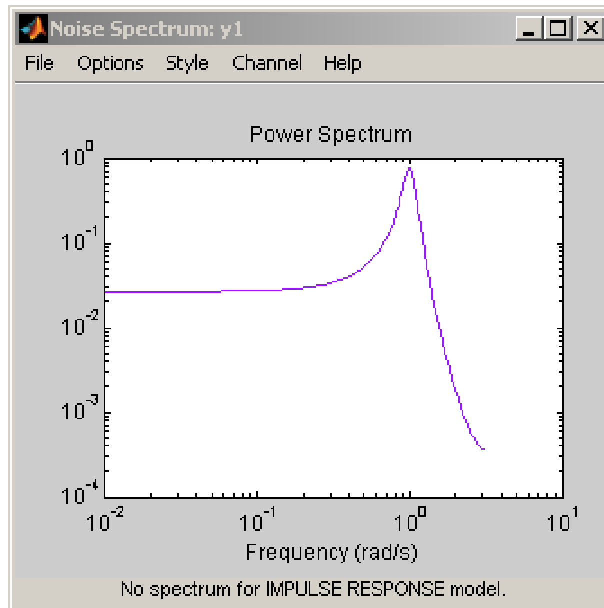
Note The calculation of the confidence interval assumes that the model sufficiently describes the system dynamics and the model residuals pass independence tests.

How to Plot the Noise Spectrum Using the GUI

To create a noise spectrum plot for parametric linear models in the GUI, 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 11-13.

To include or exclude a model on the plot, click the corresponding model icon in the System Identification Tool GUI. 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**.

Noise Spectrum Plot Settings (Continued)

Action	Command
Change amplitude scale between linear and logarithmic.	Select **Style > Linear amplitude scale** or **Style > Log amplitude scale**.
(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 8-53

How to Plot the Noise Spectrum at the Command Line

You can plot the frequency-response of the noise model.

First, select the portion of the model object that corresponds to the noise model *H*. For example, to select the noise model in the model object `m`, type the following command:

```
m_noise=m('noise')
```

Tip You can abbreviate the command to `m_noise=m('n')`.

To plot the frequency-response of the noise model, use the `bode` command:

```
bode(m_noise)
```

To determine if your estimated noise model is good enough, you can compare the frequency-response of the estimated noise-model *H* to the estimated frequency response 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 8-53 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 `bode`:

```
bode(V,m('noise'))
```

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 8-53

Pole and Zero Plots

In this section...

"Supported Models" on page 8-61
"What Does a Pole-Zero Plot Show?" on page 8-61
"Reducing Model Order Using Pole-Zero Plots" on page 8-63
"Displaying the Confidence Interval" on page 8-63

Supported Models

You can create pole-zero plots of linear input-output polynomial, state-space, and grey-box models.

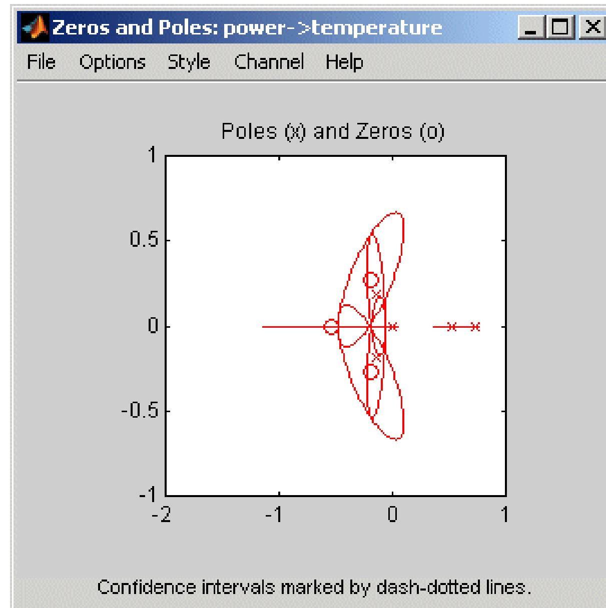
Examples

"How to Plot Model Poles and Zeros Using the GUI" on page 8-65

"How to Plot Poles and Zeros at the Command Line" on page 8-67

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:

$$y(t) - 1.5y(t-T) + 0.7y(t-2T) = \\ 0.9u(t) + 0.5u(t-T)$$

Reducing Model Order Using Pole-Zero Plots

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 cancelation.

For example, you can use the following syntax to plot a 1-standard-deviation confidence interval around model poles and zeros.

```
pzmap(model,'sd',1)
```

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 "Identifying Input-Output Polynomial Models" on page 3-39.

Displaying the Confidence Interval

In addition, you can display a confidence interval for each pole and zero on the plot. To learn how to show or hide confidence interval, see "How to Plot Model Poles and Zeros Using the GUI" on page 8-65.

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.

How to Plot Model Poles and Zeros Using the GUI

To create a pole-zero plot for parametric linear models in the System Identification Tool GUI, 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 11-13.

To include or exclude a model on the plot, click the corresponding model icon in the System Identification Tool GUI. 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.

Zeros and Poles Plot Settings (Continued)

Action	Command
Show the unit circle.	Select **Style > Unit circle**. Select this option again to hide the unit circle.
(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.

More About

"Pole and Zero Plots" on page 8-61

How to Plot Poles and Zeros at the Command Line

You can create a pole-zero plot for linear polynomial, linear state-space, and linear grey-box models using the `pzmap` command. `pzmap` lets you include several models on a plot.

To display confidence intervals for a specified number of standard deviations, use the following syntax:

```
pzmap(model,'sd',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
`pzmap`	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 `mod`, use the following command: `pzmap(mod)`

For detailed information about `pzmap`, see the corresponding reference page.

More About

"Pole and Zero Plots" on page 8-61

Akaike's Criteria for Model Validation

In this section...

"Definition of FPE" on page 8-68
"Computing FPE" on page 8-69
"Definition of AIC" on page 8-69
"Computing AIC" on page 8-70

Definition of FPE

Akaike's Final Prediction Error (FPE) criterion provides a measure of model quality by simulating the situation where the model is tested on a different data set. After computing several different models, you can compare them using this criterion. According to Akaike's theory, the most accurate model has the smallest FPE.

Note If you use the same data set for both model estimation and validation, the fit always improves as you increase the model order and, therefore, the flexibility of the model structure.

Akaike's Final Prediction Error (FPE) is defined by the following equation:

$$FPE = V\left(\frac{1 + d/N}{1 - d/N}\right)$$

where V is the loss function, d is the number of estimated parameters, and N is the number of values in the estimation data set.

The toolbox assumes that the final prediction error is asymptotic for $d \ll N$ and uses the following approximation to compute FPE:

$$FPE = V\left(1 + 2d/N\right)$$

The loss function V is defined by the following equation:

$$V = \det\left(\frac{1}{N}\sum_{1}^{N}\varepsilon\left(t,\theta_N\right)\left(\varepsilon\left(t,\theta_N\right)\right)^T\right)$$

where θ_N represents the estimated parameters.

Computing FPE

You can compute Akaike's Final Prediction Error (FPE) criterion for linear and nonlinear models.

Note FPE for nonlinear ARX models that include a tree partition nonlinearity is not supported.

To compute FPE, use the `fpe` command, as follows:

```
FPE = fpe(m1,m2,m3,...,mN)
```

According to Akaike's theory, the most accurate model has the smallest FPE.

You can also access the FPE value of an estimated model by accessing the `FPE` field of the `EstimationInfo` property of this model. For example, if you estimated the model `m`, you can access its FPE using the following command:

```
m.EstimationInfo.FPE
```

Definition of AIC

Akaike's Information Criterion (AIC) provides a measure of model quality by simulating the situation where the model is tested on a different data set. After computing several different models, you can compare them using this criterion. According to Akaike's theory, the most accurate model has the smallest AIC.

Note If you use the same data set for both model estimation and validation, the fit always improves as you increase the model order and, therefore, the flexibility of the model structure.

Akaike's Information Criterion (AIC) is defined by the following equation:

$$AIC = \log V + \frac{2d}{N}$$

where V is the loss function, d is the number of estimated parameters, and N is the number of values in the estimation data set.

The loss function V is defined by the following equation:

$$V = \det\left(\frac{1}{N} \sum_{1}^{N} \varepsilon\left(t, \theta_N\right) \left(\varepsilon\left(t, \theta_N\right)\right)^T \right)$$

where θ_N represents the estimated parameters.

For $d<<N$:

$$AIC = \log\left(V\left(1 + \frac{2d}{N} \right) \right)$$

Note AIC is approximately equal to log(FPE).

Computing AIC

Use the `aic` command to compute Akaike's Information Criterion (AIC) for one or more linear or nonlinear models, as follows:

```
AIC = aic(m1,m2,m3,...,mN)
```

According to Akaike's theory, the most accurate model has the smallest AIC.

Computing Model Uncertainty

In this section...

"Why Analyze Model Uncertainty?" on page 8-71

"What Is Model Covariance?" on page 8-71

"Types of Model Uncertainty Information" on page 8-72

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`, `idarx`, `idgrey`, `idfrd`, and `idnlgrey`.

What Is Model Covariance?

Uncertainty in the model is called *model covariance*.

If you estimate model uncertainty data, this information is stored in the `Model.CovarianceMatrix` model property. 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, as described in "Residual Analysis" on page 8-26. 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.

In the case of output-error models, where the noise model *H* is fixed to `1`, computing the covariance matrix does not assume that the residuals are white. 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, and pole-zero plots.

  Confidence intervals are computed based on the variability in the model parameters. For information about displaying confidence intervals, see the corresponding plot section.

- Covariance matrix of the estimated parameters in linear and nonlinear grey-box models.

  Type `model.CovarianceMatrix` at the prompt, where `model` represents the name of the model object.

- Estimated standard deviations of polynomial coefficients or state-space matrices

Type `model.dA` at the prompt to access the estimated standard deviations of the `model.A` estimated property, where `model` represents the name of the model object, and `A` represents any estimated model property. In general, you prefix the name of the estimated property with a `d` to get the standard deviation estimate for that property. For example, to get the standard deviation value of the *A* polynomial in an estimated ARX model, type `model.da`.

Note State-space models, estimated with free parameterization, do not have well-defined standard deviations of the matrix elements. To display matrix parameter uncertainties in this case, first transform the model to a canonical parameterization by setting the `ss` model property to `model.ss = 'canon'`. For more information about free and canonical parameterizations, see "Identifying State-Space Models" on page 3-73.

- Simulated output values for linear models with standard deviations using the `sim` command.

  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.

Troubleshooting Models

In this section...

"About Troubleshooting Models" on page 8-74

"Model Order Is Too High or Too Low" on page 8-74

"Nonlinearity Estimator Produces a Poor Fit" on page 8-75

"Substantial Noise in the System" on page 8-76

"Unstable Models" on page 8-76

"Missing Input Variables" on page 8-78

"Complicated Nonlinearities" on page 8-78

About Troubleshooting Models

During validation, you might find that your model output fits the validation data poorly. You might also find some unexpected or undesirable model characteristics.

If the tips suggested in these sections do not help improve your models, then a good model might not be possible for this data. For example, your data might have poor signal-to-noise ratio, large and nonstationary disturbances, or varying system properties.

Model Order Is Too High or Too Low

When the Model Output plot does not show a good fit, there is a good chance that you need to try a different 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.

You can estimate the model order as described in "Preliminary Step – Estimating Model Orders and Input Delays" on page 3-48. Typically, you use the suggested order as a starting point to estimate the lowest possible order with different model structures. After each estimation, you monitor the Model Output and the Residual Analysis plots, and then adjust your settings for the next estimation.

When a low-order model fits the validation data poorly, try estimating a higher-order model to see if the fit improves. For example, if a Model Output plot shows that a fourth-order model gives poor results, try estimating an eighth-order model. When a higher-order model improves the fit, you can conclude that higher-order models might be required and linear models might be sufficient.

You should use an independent data set to validate your models. If you use the same data set to both estimate and validate a model, the fit always improves as you increase model order, and you risk overfitting. However, if you use an independent data set to validate your models, the fit eventually deteriorates if your model orders are too high.

High-order models are more expensive to compute and result in greater parameter uncertainty.

Nonlinearity Estimator Produces a Poor Fit

In the case of nonlinear ARX and Hammerstein-Wiener models, the Model Output plot does not show a good fit when the nonlinearity estimator has incorrect complexity.

You specify the complexity of piece-wise-linear, wavelet, sigmoid, and custom networks using the number of units (`NumberOfUnits` nonlinear estimator property). A high number of units indicates a complex nonlinearity estimator. In the case of neural networks, you specify the complexity using the parameters of the network object. For more information, see the Neural Network Toolbox documentation.

To select the appropriate complexity of the nonlinearity estimator, start with a low complexity and validate the model output. Next, increate the complexity and validate the model output again. The model fit degrades when the nonlinearity estimator becomes too complex.

Note To see the model fit degrade when the nonlinearity estimator becomes too complex, you must use an independent data set to validate the data that is different from the estimation data set.

Substantial Noise in the System

There are a couple of indications that you might have substantial noise in your system and might need to use linear model structures that are better equipped to model noise.

One indication of noise is when a state-space model is better than an ARX model at reproducing the measured output; whereas the state-space structure has sufficient flexibility to model noise, the ARX model structure is less able to model noise because the A polynomial must account for both the system dynamics and the noise. The following equation represents the ARX model and shows that A couples the dynamics and the noise by appearing in the denominator of both the dynamics term and the noise terms:

$$y = \frac{B}{A}u + \frac{1}{A}e$$

Another indication that a noise model is needed appears in residual analysis plots when you see significant autocorrelation of residuals at nonzero lags. For more information about residual analysis, see "Residual Analysis" on page 8-26.

To model noise more carefully, use the ARMAX or the Box-Jenkins model structure, where the dynamics term and the noise term are modeled by 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 8-61. 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 might cause linear models to be unstable. However, detrending the model does not guarantee stability.

When an unstable model is OK: In some cases, an unstable model is still a useful model. For example, your system might be unstable without a controller, and you plan to use your model for control design. In this case, you can import your unstable model into Simulink or Control System Toolbox products.

Forcing stability during estimation: If you believe that your system is stable, but your model is unstable, then you can estimate the model with a `Focus` set to `Stability`. This setting might result in a reduced model quality. For more information about `Focus`, see the `Algorithm Properties` reference page.

Allowing for some instability: A more advanced approach to achieving a stable model is by setting the stability threshold property to allow a margin of error. The threshold model property is accessed as a field in the `algorithm` structure:

- For continuous-time models, set the value of `model.algorithm.advanced.sstability`. The model is considered stable if the pole on the far right is to the left of `sstability` threshold.
- For discrete-time models, set the value of `model.algorithm.advanced.zstability`. The model is considered stable if all poles inside the circle centered at the origin and with a radius `zstability`.

For more information about `Threshold` fields for linear models, see the `Algorithm Properties` reference page.

Unstable Nonlinear Models

To test if a *nonlinear model* is unstable is to 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.

Missing Input Variables

If the Model Output plot and Residual Analysis plot shows a poor fit and you have already tried different structures and orders and modeled noise, it might be that there are one or more missing inputs that have a significant effect on the output.

Try including other measured signals in your input data, and then estimating the models again.

Inputs need not be control signals. Any measurable signal can be considered an input, including measurable disturbances.

Complicated Nonlinearities

If the Model Output plot and Residual Analysis plot shows a poor fit, consider if nonlinear effects are present in the system.

You can model the nonlinearities by performing a simple transformation on the signals to make the problem linear in the new variables. For example, if electrical power is the driving stimulus in a heating process and temperature is the output, you can form a simple product of voltage and current measurements.

If your problem is sufficiently complex and you do not have physical insight into the problem, you might try fitting nonlinear black-box models. For more information, see Chapter 4, "Nonlinear Black-Box Model Identification".

Next Steps After Getting an Accurate Model

For linear parametric models (`idmodel` objects), 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 3-112.

- Transform between linear model representations, such as between polynomial, state-space, and zero-pole representations.

  See "Transforming Between Linear Model Representations" on page 3-117.

- Extract numerical data from transfer functions, pole-zero models, and state-space matrices.

  See "Extracting Numerical Model Data" on page 3-109.

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 4-81.

System Identification Toolbox models in the MATLAB workspace are immediately available to other MathWorks® products. However, if you used the System Identification Tool GUI to estimate models, you must first export the models to the MATLAB workspace.

Tip To export a model from the GUI, drag the model icon to the **To Workspace** rectangle. For more information about working with the GUI, see Chapter 11, "System Identification Tool GUI".

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 9-2.

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 Chapter 10, “System Identification Toolbox Blocks”.

Control Design Applications

- "Using Identified Models for Control Design Applications" on page 9-2
- "Example – Using System Identification Toolbox Software with Control System Toolbox Software" on page 9-7

Using Identified Models for Control Design Applications

In this section...

"How Control System Toolbox Software Works with Identified Models" on page 9-2

"Using balred to Reduce Model Order" on page 9-3

"Compensator Design Using Control System Toolbox Software" on page 9-3

"Converting Models to LTI Objects" on page 9-4

"Viewing Model Response Using the LTI Viewer" on page 9-5

"Combining Model Objects" on page 9-6

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 LTI Viewer GUI 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`, `linearize(idnlarx)`, or `linearize(idnlhw)` 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.

For information about using the Control System Toolbox software, see the Control System Toolbox documentation.

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 for any `idmodel` object, including `idarx`, `idpoly`, `idss`, and `idgrey`.

For more information about using `balred`, see the corresponding reference page. To learn how you can reduce model order using pole-zero plots, see "Reducing Model Order Using Pole-Zero Plots" on page 8-63.

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 Tool GUI to estimate models, you must first export the models to the MATLAB workspace. To export a model from the GUI, drag the model icon to the **To Workspace** rectangle.

Control System Toolbox software provides both the SISO Design Tool GUI and commands for working at the command line. You can import polynomial and state-space models directly into SISO Design Tool using the following command:

```
sisotool(model('measured'))
```

where you use only the dynamic model and not the noise model. For more information about subreferencing the dynamic or the noise model, see "Subreferencing Measured and Noise Models" on page 3-120. To design a controller using Control System Toolbox commands and methods at the command line, you must convert the plant model to an LTI object. For more information, see "Converting Models to LTI Objects" on page 9-4.

Note The syntax `sisotool(model('m'))` is equivalent to `sisotool(model('measured'))`.

For more information about controller design using SISO Design Tool and Control System Toolbox commands, see the Control System Toolbox documentation.

Converting Models to LTI Objects

Control System Toolbox commands operate on LTI objects. To design a controller for a plant model, you must first convert the System Identification Toolbox model object to an LTI object.

You can convert linear polynomial, state-space, and grey-box model objects, including `idarx`, `idpoly`, `idproc`, `idss`, or `idgrey`, to LTI objects.

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 transforms an `idmodel` object to an LTI state-space object:

```
% Extract the measured model
% and ignore the noise model
```

```
model = model('measured')
% Convert to LTI object
LTI_sys = idss(model)
```

The LTI object includes only the dynamic model and not the noise model, which is estimated for every linear model in the System Identification Toolbox software.

Note To include noise channels in the LTI models, first use `noisecnv` to convert the noise in the `idmodel` object to measured channels, and then convert to an LTI object.

For more information about subreferencing the dynamic or the noise model, see "Subreferencing Measured and Noise Models" on page 3-120.

Viewing Model Response Using the LTI Viewer

- "What Is the LTI Viewer?" on page 9-5
- "Displaying Identified Models in the LTI Viewer" on page 9-6

What Is the LTI Viewer?

If you have the Control System Toolbox software, you can plot models in the LTI Viewer from either the System Identification Tool GUI or the MATLAB Command Window.

The LTI Viewer is a graphical user interface for viewing and manipulating the response plots of linear models.

Note The LTI Viewer does not display model uncertainty.

For more information about working with plots in the LTI Viewer, see the Control System Toolbox documentation.

Displaying Identified Models in the LTI Viewer

When the MATLAB software is installed, the System Identification Tool GUI contains the **To LTI Viewer** rectangle. To plot models in the LTI Viewer, drag and drop the corresponding icon to the **To LTI Viewer** rectangle in the System Identification Tool GUI.

Alternatively, use the following syntax when working at the command line to view a model in the LTI Viewer:

```
view(model)
```

Combining Model Objects

If you have the Control System Toolbox software, you can combine linear model objects, such as `idarx`, `idgrey`, `idpoly`, `idproc`, and `idss` model objects, similar to the way you combine LTI objects.

For example, you can perform the following operations on identified models:

- `G1+G2`
- `G1*G2`
- `append(G1,G2)`
- `feedback(G1,G2)`

Note These operations lose covariance information.

Example – Using System Identification Toolbox Software with Control System Toolbox Software

This example demonstrates how to use both System Identification Toolbox commands and Control System Toolbox commands to create and plot models:

```
% Construct model using Control System Toolbox
m0 = drss(4,3,2)
% Convert model to an idss object
m0 = idss(m0,'NoiseVar',0.1*eye(3))
% Generate input data for simulating output
u = iddata([], idinput([800 2],'rbs'));
% Simulate model output using System Identification Toolbox
% with added noise
y = sim(m0,u,'noise')
% Form an input-output iddata object
Data = [y u];
% Estimate state-space model from the generated data
% using System Identification Toolbox command pem
m = pem(Data(1:400))
% Convert the model to a Control System Toolbox transfer function
tf(m)
% Plot model output for model m using System Identification Toolbox
compare(Data(401:800),m)
% Display identified model m in LTI Viewer
view(m)
```

10

System Identification Toolbox Blocks

- "Using System Identification Toolbox Blocks in Simulink Models" on page 10-2
- "Preparing Data" on page 10-3
- "Identifying Linear Models" on page 10-4
- "Simulating Identified Model Output in Simulink" on page 10-5
- "Example – Simulating an Identified Model Using Simulink Software" on page 10-8

Using System Identification Toolbox Blocks in Simulink Models

System Identification Toolbox 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, select **Start > Simulink > Library Browser**. 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
```

For more information about blocks, see “Block Reference” in the *System Identification Toolbox Reference*. To get help on a specific block, right-click the block in the Library Browser, and select **Help**.

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 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
AutoRegressive model estimator	Estimate AR model parameters from time-series data, which has one output and no input.
AutoRegressive Moving Average with eXternal input model estimator	Estimate ARMAX model parameters from input/output data.
AutoRegressive with eXternal input model estimator	Estimate ARX model parameters from input/output data.
Box-Jenkins model estimator	Estimate BJ model parameters from input/output data.
Output-error model estimator	Estimate OE model parameters from input/output data.
General model estimator using Predictive Error Method	Estimate ARX, ARMAX, Box-Jenkins, and Output-Error models (`idpoly` objects) from single-input and single output data using general prediction-error method.

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 10-5
"Summary of Simulation Blocks" on page 10-5
"Specifying Initial Conditions for Simulation" on page 10-6

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 10-5.

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 `idmodel` model in Simulink, including low-order transfer function (`idproc`), linear polynomial (`idpoly`), state-space (`idss`), and grey-box (`idgrey`) models. Also simulates `idarx` model objects.
Nonlinear ARX Model	Simulate `idnlarx` model in Simulink.

Block	Description
Hammerstein-Wiener Model	Simulate `idnlhw` model in Simulink.
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 10-6.) 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.

For more information on specifying initial conditions, see the following topics:

- "Specifying Initial States of Linear Models" on page 10-6
- "Specifying Initial States of Nonlinear ARX Models" on page 10-7
- "Specifying Initial States of Hammerstein-Wiener Models" on page 10-7

Specifying Initial States of Linear Models

For `idss` and `idgrey` models, specify the initial states for simulation in the **Initial state** field of the Function Block Parameters: Idmodel dialog box:

- To specify the initial states values as zero, use `'z'`.
- To use the initial states values stored in the `X0` property of the model, use `'m'`.
- To match the simulated response of the model to a certain input/output data set, first use findstates to estimate initial states values that maximize the fit. Specify the estimated initial states values in the **Initial state** field.

For `idpoly` and `idarx` models, the default initial states values are zero. If you want to specify different values, such as maximize fit to a given output

data, convert the model to `idss` object, and then specify its initial states as described previously. For example, for the following `idpoly` model:

```
m1=idpoly([1 2 1],[2 2]);
```

the initial states correspond to those of the equivalent state-space model:

```
m2=idss(m1);
```

For more information about specifying initial conditions for simulation, see the IDMODEL Model reference page.

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.

Example – Simulating an Identified Model Using Simulink Software

In this example, you set the initial states for simulating a model such that the simulation provides a best fit to measured input-output data.

Prerequisites

Estimate a three-state 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.
M = n4sid(Z,5,'Focus','Simulation');
```

When you estimate a model using multiple data sets, the initial-states property, X0, of the model, M, stores only the estimated states corresponding to the last data set. In this example:

- Values of M.X0 are the estimated state values, corresponding to the last experiment in Z.
- M.X0 is a vector of length 5, corresponding to the five states of the model.

To compute initial states that maximizes the fit to the corresponding output y2, and simulate the model using the second experiment:

1. Estimate the initial states using the second experiment:

   ```
   X0est = findstates(M,z2);
   ```

2 Open the System Identification Toolbox library by typing the following command at the MATLAB prompt:

```
slident
```

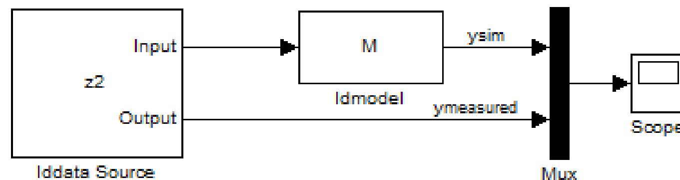
3 Open a new Simulink model window. Then, drag and drop an `Idmodel` block from the library into the model window.

4 Open the Function Block Parameters dialog box by double-clicking the `idmodel` block. Specify the following block parameters:

   a In the **idmodel variable** field, type `M` to specify the estimated model.

   b In the **Initial states** 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.

System Identification Tool GUI

- "Steps for Using the System Identification Tool GUI" on page 11-2
- "Working with the System Identification Tool GUI" on page 11-3

Steps for Using the System Identification Tool GUI

A typical workflow in the System Identification Tool GUI 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 Tool GUI, or open a saved session. For more information, see “Starting a New Session in the GUI” on page 11-4.
3. Import data into the GUI from the MATLAB workspace. For more information, see “Importing Data into the GUI” on page 2-17.
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 Chapter 2, “Data Import and Processing”.
5. Specify the data for estimation and validation. For more information, see “Specifying Estimation and Validation Data” on page 2-35.
6. Select the model to estimating using the **Estimate** menu. For more information, see Chapter 1, “Choosing Your System Identification Approach”.
7. Validate models. For more information, see Chapter 8, “Model Analysis”.
8. Export models to the MATLAB workspace for further analysis. For more information, see “Exporting Models from the GUI to the MATLAB Workspace” on page 11-12.

Related Examples

- “Tutorial – Identifying Linear Models Using the GUI”
- “Tutorial – Identifying Low-Order Transfer Functions (Process Models) Using the GUI”
- “Tutorial – Identifying Nonlinear Black-Box Models Using the GUI”

Working with the System Identification Tool GUI

In this section...

"Starting and Managing GUI Sessions" on page 11-3

"Managing Models" on page 11-7

"Working with Plots" on page 11-13

"Customizing the System Identification Tool GUI" on page 11-17

"Related Examples" on page 11-20

Starting and Managing GUI Sessions

- "What Is a System Identification Tool Session?" on page 11-3
- "Starting a New Session in the GUI" on page 11-4
- "Description of the System Identification Tool Window" on page 11-5
- "Opening a Saved Session" on page 11-6
- "Saving, Merging, and Closing Sessions" on page 11-6
- "Deleting a Session" on page 11-7

What Is a System Identification Tool Session?

A *session* represents the total progress of your identification process, including any data sets and models in the System Identification Tool GUI.

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 GUI" on page 11-4.

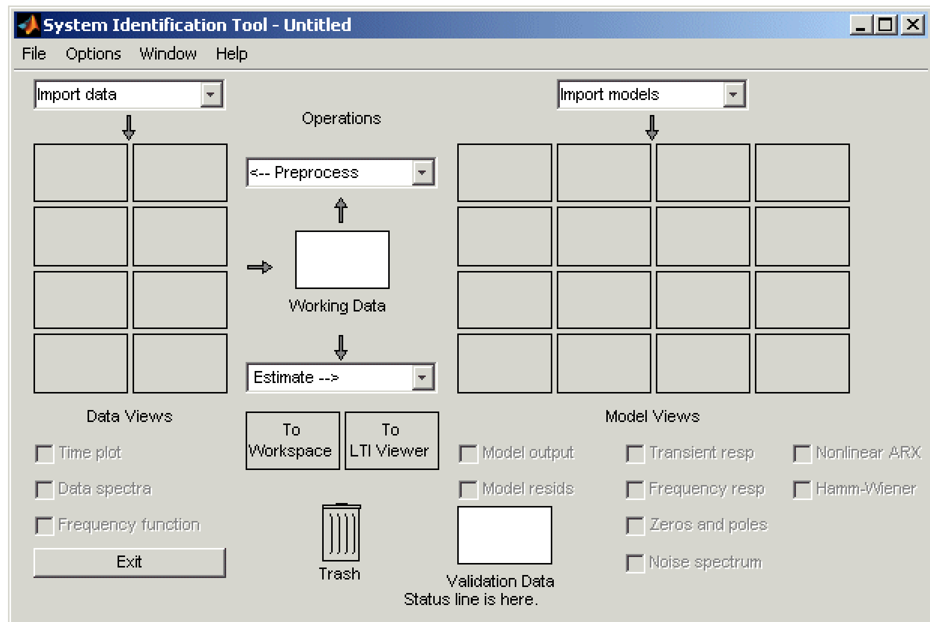
For more information about the steps for using the System Identification Tool GUI, see "Steps for Using the System Identification Tool GUI" on page 11-2.

Starting a New Session in the GUI

To start a new session in the System Identification Tool GUI, type the following command in the MATLAB Command Window:

```
ident
```

Alternatively, you can start a new session by selecting **Start > Toolboxes > System Identification > System Identification Tool GUI** in the MATLAB desktop. This action opens the System Identification Tool GUI.

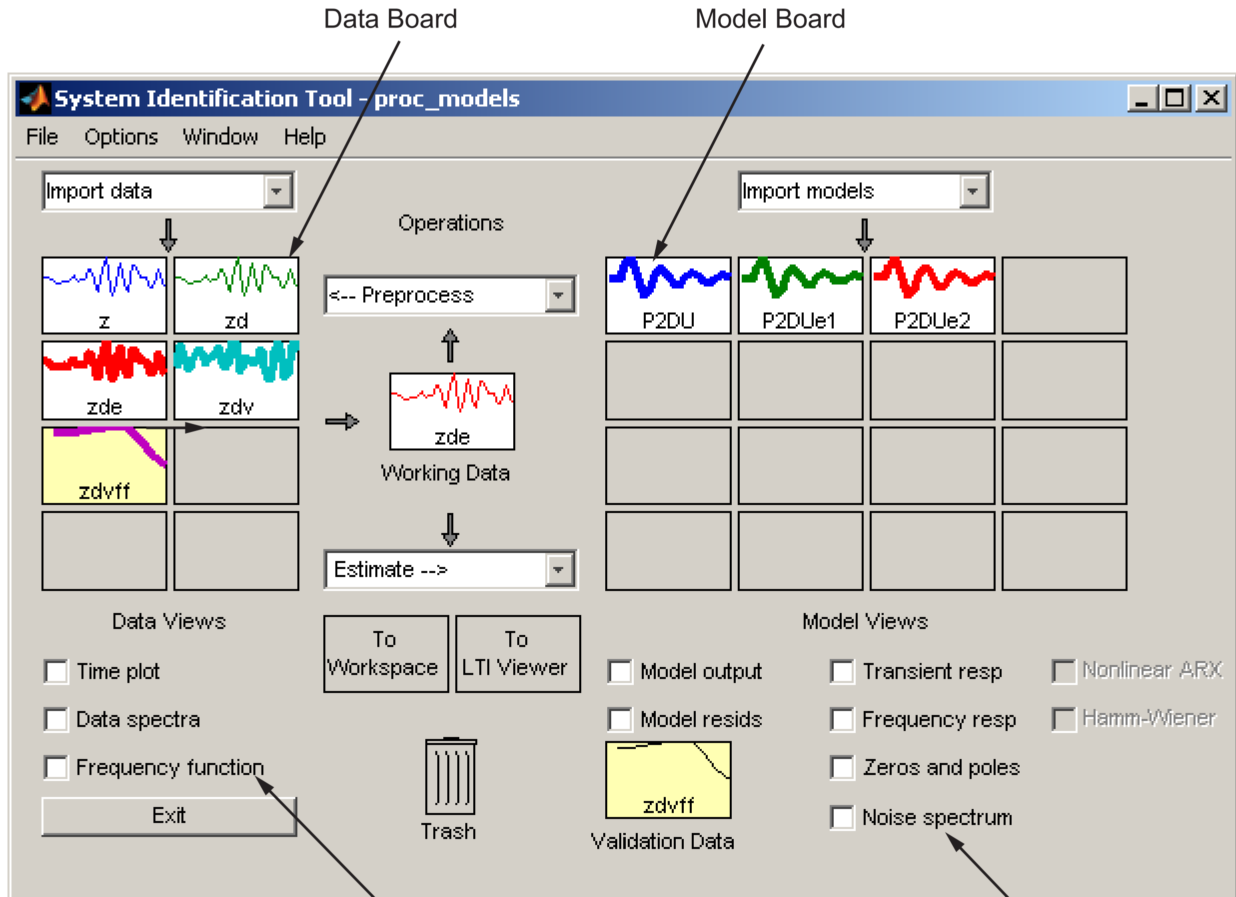


Note Only one session can be open at a time.

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 Tool Window

The following figure describes the different areas in the System Identification Tool GUI.



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 Tool GUI 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 Tool GUI, see "Steps for Using the System Identification Tool GUI" on page 11-2.

The **Data Board** area, located below the **Import data** menu in the System Identification Tool GUI, contains rectangular icons that represent the data you imported into the GUI.

The Model Board, located to the right of the **<--Preprocess** menu in the System Identification Tool GUI, contains rectangular icons that represent the models you estimated or imported into the GUI. 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:

```
ident(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 Tool GUI is already open, you can open a session by selecting **File > Open session**.

Note If there is data in the System Identification Tool GUI, 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 Tool GUI.

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.
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 GUI" on page 11-8
- "Viewing Model Properties" on page 11-9
- "Renaming Models and Changing Display Color" on page 11-10
- "Organizing Model Icons" on page 11-10
- "Deleting Models in the GUI" on page 11-11

- "Exporting Models from the GUI to the MATLAB Workspace" on page 11-12

Importing Models into the GUI

You can import System Identification Toolbox models from the MATLAB workspace into the System Identification Tool GUI. 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 Tool GUI already open. If this window is not open, type the following command at the prompt:

```
ident
```

To import models into the System Identification Tool GUI:

1. In the System Identification Tool GUI, 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 Tool GUI 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 Tool GUI. 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 Tool GUI.

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 GUI.

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 Tool GUI" on page 11-17.

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 strings 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.

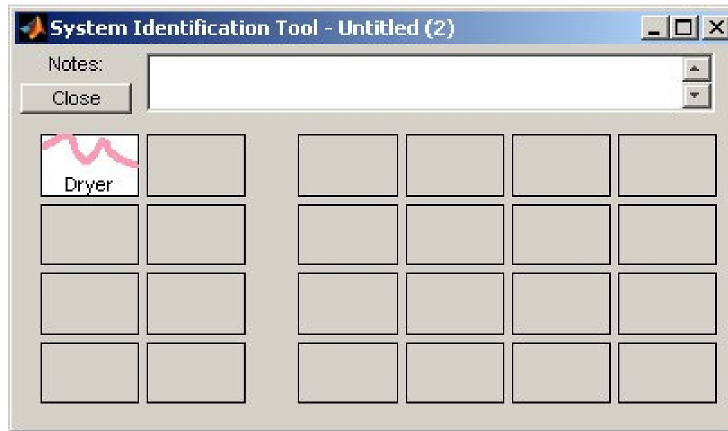
To view model properties in the MATLAB Command Window, click **Present**.

Organizing Model Icons

You can rearrange model icons in the System Identification Tool GUI 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 Tool GUI. 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 Tool GUI and any extra session windows.

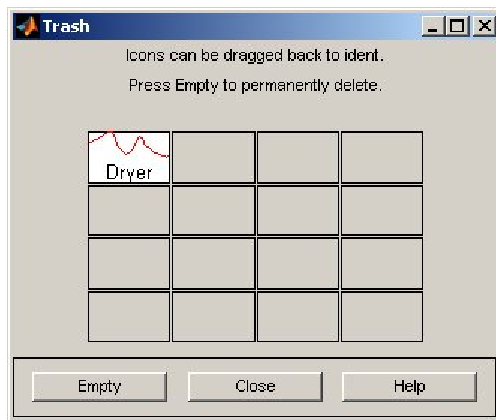
Type comments in the **Notes** field to describe the models. When you save a session, as described in "Saving, Merging, and Closing Sessions" on page 11-6, all additional windows and notes are also saved.

Deleting Models in the GUI

To delete models in the System Identification Tool GUI, drag and drop the corresponding icon into **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 Tool GUI. 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 GUI to the MATLAB Workspace

The models you create in the System Identification Tool GUI 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, drag and drop the corresponding icon to the **To Workspace** rectangle.

When you export models to the MATLAB workspace, the resulting variables have the same name as in the System Identification Tool GUI.

Working with Plots

- "Identifying Data Sets and Models on Plots" on page 11-13
- "Changing and Restoring Default Axis Limits" on page 11-14
- "Selecting Measured and Noise Channels in Plots" on page 11-16
- "Grid and Line Styles in Plots" on page 11-17
- "Opening a Plot in a MATLAB Figure Window" on page 11-17
- "Printing Plots" on page 11-17

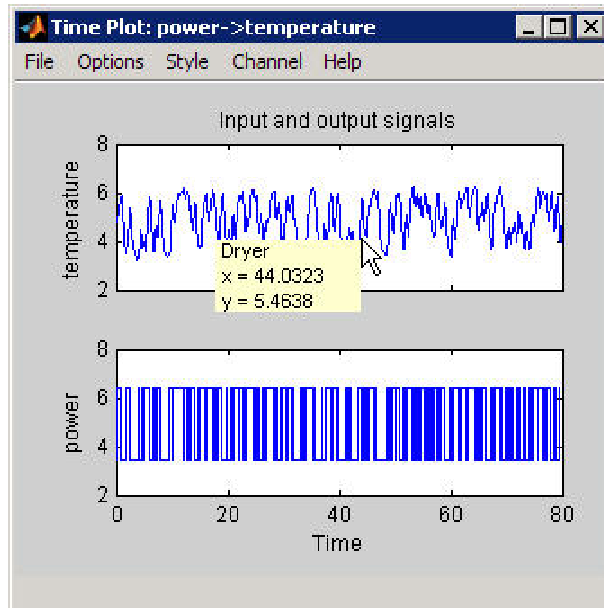
Identifying Data Sets and Models on Plots

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 Tool GUI 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 11-14.

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 11-14
- "Setting Axis Limits" on page 11-15

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 "Subreferencing Measured and Noise Models" on page 3-120.

Tip When you import data into the System Identification Tool GUI, it is helpful to assign meaningful channel names in the Import Data dialog box. For more information about importing data, see "Importing Data into the GUI" on page 2-17.

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 11-17
- "Solid or Dashed Lines" on page 11-17

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 Tool GUI.

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 Tool GUI, and then open it in a MATLAB Figure window to fine-tune the display.

After you create the plot, as described in "Plotting Models in the GUI" on page 8-8, 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 Tool GUI

- "Types of GUI Customization" on page 11-18

- "Displaying Warnings While You Work" on page 11-18
- "Saving Session Preferences" on page 11-18
- "Modifying idlayout.m" on page 11-19

Types of GUI Customization

The System Identification Tool GUI 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 GUI state.

Advanced users might choose to edit the file that controls default settings, as described in "Modifying idlayout.m" on page 11-19.

Displaying Warnings While You Work

In the System Identification Tool GUI, select **Options > Warnings** to display informational dialog boxes while you work. Verify that a check mark appears to the right of **Warnings**.

To stop warnings from being displayed during your session, select **Options > Warnings** and clear the check mark.

Saving Session Preferences

Use **Options > Save preferences** to save the current state of the System Identification Tool GUI. This command saves the following settings to a preferences file, `idprefs.mat`:

- Size and position of the System Identification Tool GUI
- 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 GUI.

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

Advanced users might want to customize the default plot options by editing `idlayout.m`.

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**.

Related Examples

- "Tutorial – Identifying Linear Models Using the GUI"
- "Tutorial – Identifying Low-Order Transfer Functions (Process Models) Using the GUI"
- "Tutorial – Identifying Nonlinear Black-Box Models Using the GUI"

Index

A

active
  model in GUI 8-8
`advice`
  for data 2-91
  for models 8-10
AIC 8-68
  definition 8-69
Akaike's Final Prediction Error (FPE) 8-68
Akaike's Information Criterion (AIC) 8-68
`Algorithm` property 1-18
algorithms for estimation
  recursive 7-6
  spectral models 3-5
aliasing effects 2-108
AR 6-7
ARMA 6-7
ARMAX 3-42
ARX 3-42
ARX Model Structure Selection window 3-54

B

best fit
  definition 8-16
  negative value 8-17
BJ model. *See* Box-Jenkins model
Bode plot 8-46
Box-Jenkins model 3-42
Burg's method 6-11

C

`c2d` 3-112
canonical parameterization 3-92
complex data 2-142
concatenating
  `iddata` objects 2-65
  `idfrd` objects 2-77
  models 3-124
confidence interval
  impulse response plot 8-37
  model output plot 8-19
  noise spectrum plot 8-54
  residual plot 8-28
  step response plot 8-37
confidence interval on plots 8-72
constructor 1-15
continuous-time models
  supported 1-10
continuous-time process models 3-20
Control System Toolbox
  combining model objects 9-6
  converting models to LTI objects 9-4
  for compensator design 9-3
  LTI Viewer 9-5
  reducing model order 9-3
correlation analysis 3-11
covariance 8-71
`CovarianceMatrix` 8-71
`cra` 3-14
cross-validation 8-4

D

D matrix 3-90
`d2c` 3-112
`d2d` 3-112
data
  creating `iddata` object 2-53
  creating `idfrd` object 2-73
  creating subsets 2-37
  detrending 2-101
  exporting to MATLAB workspace 2-51
  filter 2-116
  frequency-domain 2-11
  frequency-response 2-13
  importing into System Identification Tool
    GUI 2-17
  managing in GUI 2-17

merging 2-39
missing data 2-97
multiexperiment data 2-39
outliers 2-98
plotting 2-82
renaming in GUI 2-47
resampling 2-107
sampling interval 2-34
segmentation 7-14
selecting 2-93
simulating 2-126
supported types 2-3
time-domain 2-9
time-series 2-10
transforming domain 2-130
viewing properties in GUI 2-46
Data Board 11-6
arranging icons 2-49
deleting icons 2-50
data tip 11-13
dead time 3-39
delay
estimating for polynomial models 3-48
detrending data 2-101
discrete-time models
supported 1-11

E

estimating models
black-box polynomial 3-39
commands 1-12
frequency response 3-2
Hammerstein-Wiener 4-49
linear grey-box 5-6
nonlinear ARX 4-8
nonlinear grey-box 5-15
process models 3-20
recursive estimation 7-2
state-space 3-73
time-series 6-1
transient response 3-11
uncertainty 8-71
`EstimationInfo` property 1-19
`etfe`
algorithm 3-5
export
data to MATLAB workspace 2-51
model to MATLAB workspace 11-12

F

filtering data 2-116
forgetting factor algorithm 7-10
FPE 8-68
free parameterization 3-84
frequency resolution 3-6
frequency response
estimating in the GUI 3-3
`etfe` 3-5
`spa` 3-5
`spafdr` 3-5
frequency-domain data 2-11
frequency-response data 2-13
frequency-response plot 8-44
Bode plot 8-48
Nyquist plot 8-51

H

Hammerstein-Wiener models 4-49

I

`idarx` 1-16
`iddata`
concatenating 2-65
creating 2-53
subreferencing 2-61
`ident` 11-4
`idfrd`

concatenating 2-77
creating 2-73
model 1-16
subreferencing 2-76
`idgrey` 1-16
`idlayout.m` 11-19
`idnlarx` 1-16
`idnlgrey` 1-16
`idnlhw` 1-16
`idpoly` 1-16
`idproc` 1-16
`idss` 1-16
importing
data into System Identification Tool GUI 2-17
impulse response
computing values 3-15
confidence interval 8-37
definition 3-11
estimating in the GUI 3-12
`impulse` 3-14
impulse-response plot 8-35
independence test 8-26

K

K matrix 3-90
Kalman filter algorithm 7-8

L

linear grey-box models 5-6
linear models
extracting numerical data 3-109
transforming between continuous and discrete time 3-112
transforming between structures 3-117
LTI Viewer 9-5

M

MDL 3-53
merging
data 2-39
models 3-128
methods 1-14
missing data 2-97
model
black-box polynomial 3-39
estimating frequency response 3-2
estimating process model 3-20
estimating transient response 3-11
exporting to MATLAB workspace 11-12
grey-box estimation 5-1
Hammerstein-Wiener estimation 4-49
importing into GUI 11-8
linear grey-box estimation 5-6
managing in GUI 11-7
nonlinear ARX estimation 4-8
nonlinear black-box estimation 4-1
nonlinear grey-box estimation 5-15
ordinary difference equation 5-1
ordinary differential equation 5-1
plotting 8-5
properties 1-17
recursive estimation 7-2
reducing order using `balred` 9-3
reducing order using pole-zero plot 8-63
refining linear parametric 3-104
renaming in GUI 11-10
state-space 3-73
time-series 6-1
uncertainty 8-71
validating 8-3
viewing properties in GUI 11-9
Model Board 11-6
arranging icons 11-10
deleting icons 11-11
model object
concatenating 3-124

definition 1-14
instantiating 1-15
merging 3-128
methods 1-14
properties 1-17
types of 1-16
model order
definition 3-39
estimating for polynomial models 3-48
estimating for state-space 3-79
too high or too low 8-74
Model Order Selection window 3-83
model output
confidence interval 8-19
model output plot 8-11
model properties
accessing 1-20
help on 1-22
specifying 1-19
multiexperiment data 2-39

N

noise
converting to measured channels 3-122
evidence in estimated model 8-76
subreferencing 3-120
noise spectrum
confidence interval 8-54
noise spectrum plot 8-53
nonlinear ARX models 4-8
nonlinear grey-box models 5-15
nonlinear models 4-1
nonlinearity estimators
troubleshooting 8-75
normalized gradient algorithm 7-11

O

OE model. *See* Output-Error model
offset levels 2-101
order. *See* model order
outliers 2-98
Output-Error model 3-42

P

`pem`
for polynomial models 3-61
for process models 3-27
for state-space models 3-87
periodogram
`etfe` for time series 6-5
physical equilibrium 2-101
plot
copy to MATLAB Figure window 11-17
data 2-82
data tip 11-13
in LTI Viewer 9-5
models 8-5
models in the GUI 8-8
print 11-17
selecting noise channels 11-16
pole-zero cancelation 8-63
pole-zero plot 8-61
polynomial models 3-39
estimating order 3-48
for time-series 6-7
print plot 11-17
process model 3-20
definition 3-20
properties
for models 1-17

R

recursive estimation 7-2
reducing model order
using `balred` 9-3
using pole-zero plot 8-63

refining models
linear parametric 3-104
renaming data 2-47
resampling data 2-107
avoiding aliasing 2-108
residual analysis
confidence interval 8-28
plot 8-26
residuals
plotting using the System Identification Tool 8-31
Rissanen's Minimum Description Length (MDL) 3-53
robust criterion
for outliers 2-98

S

sampling interval 2-34
saving session preferences 11-18
segmentation of data 7-14
selecting data 2-93
session
definition 11-3
managing in GUI 11-3
preferences 11-18
starting 11-4
simulating data 2-126
Simulink 10-2
`slident` 10-2
`spa`
algorithm 3-5
`spafdr`
algorithm 3-5
spectral analysis 3-2
algorithm 3-5
frequency resolution 3-6
spectrum normalization 3-8
spectrum normalization 3-8
state-space models 3-73
canonical parameterization 3-92
estimating order 3-79
for time series 6-12
free parameterization 3-84
structured parameterization 3-94
supported parameterization 3-78
step response
computing values 3-15
confidence interval 8-37
definition 3-11
estimating in the GUI 3-12
`step` 3-14
step-response plot 8-35
structured parameterization 3-94
subreferencing
`iddata` objects 2-61
`idfrd` objects 2-76
model channels 3-119
model noise channels 3-120
models 3-119
System Identification Tool GUI
customizing 11-17
open 11-4
plots 11-13
window 11-5
workflow 11-2
System Identification Toolbox blocks 10-2
for data 10-3
for model identification 10-4
for simulating models 10-5
open 10-2

T

time-domain data 2-9
time-series data 2-10
time-series models 6-1
transforming data domain 2-130
troubleshooting models 8-74
complicated nonlinearities 8-78

high noise content 8-76
missing inputs 8-78
model order 8-74
nonlinearity estimators 8-75
unstable models 8-76

U

uncertainty of models 8-71
confidence interval on plots 8-72
covariance 8-71
unnormalized gradient algorithm 7-11
unstable models 8-76

V

validating models 8-3
comparing model output 8-11
residual analysis 8-26
troubleshooting 8-74
Validation Data 2-35

W

warnings 11-18
whiteness test 8-26
Working Data 2-35

X

X0 matrix 3-90

Y

Yule-Walker approach 6-11