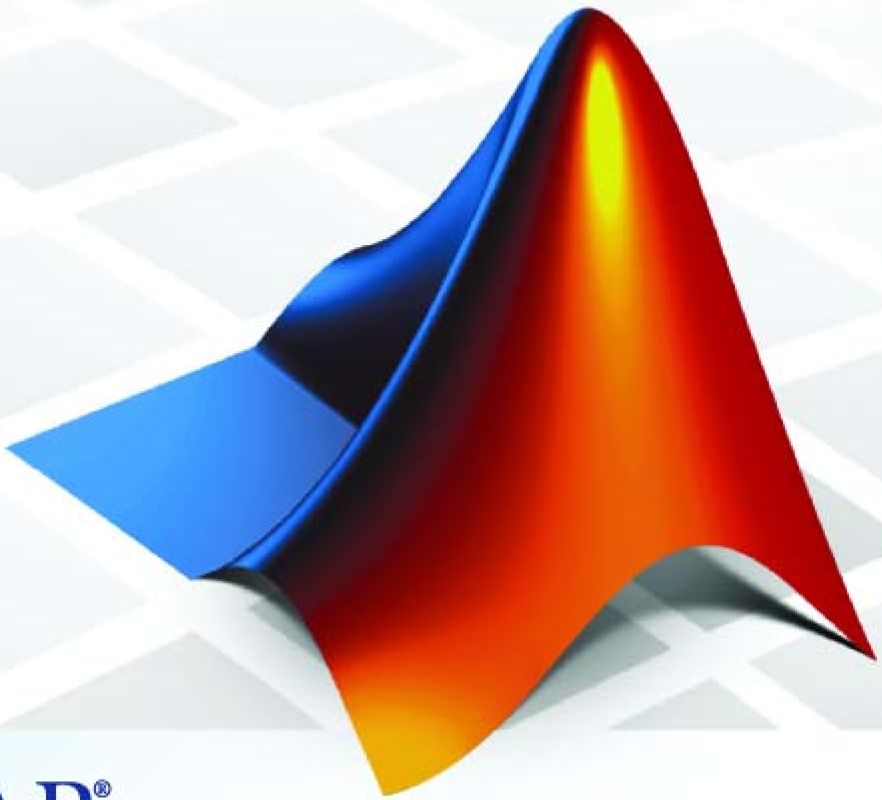
System Identification Toolbox 7

User's Guide

Lennart Ljung



MATLAB® & SIMULINK®

The MathWorks

Accelerating the pace of engineering and science

How to Contact The 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

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)

About the Developers

System Identification Toolbox 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 Introduction to System Identification Toolbox Models

2 Working with the System Identification Tool GUI

3 Representing Data for System Identification

5 Estimating Linear Nonparametric and Parametric Models

6 Estimating Nonlinear Black-Box Models

7 Estimating Grey-Box Models

8 Recursive Parameter Estimation

9 Plotting and Validating Models

11 Functions — By Category

Introduction to System Identification Toolbox Models

What Are Models and Model Objects?

System Identification Toolbox extends the MATLAB® computation environment and lets you estimate linear and nonlinear mathematical models to fit input and output data from dynamic systems.

You can estimating models using System Identification Toolbox commands in the MATLAB Command Window, or you can work in the System Identification Tool graphical user interface (GUI).

To quickly get started using System Identification Toolbox, see *Getting Started with System Identification* for an overview of the Toolbox capabilities and comprehensive hands-on tutorials.

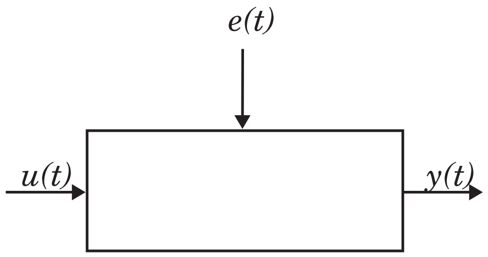
This section discusses the following topics:

- "Definition of a Model" on page 1-2
- "Summary of Supported Models" on page 1-4
- "Definition of a Model Object" on page 1-5

Definition of a Model

A *model* of a system is a computational tool you use to answer questions about the system without having to perform experiments. For example, you might use a model to simulate the output of a system for a given input and analyze the system's response. Alternatively, you might be interested in predicting the future output of a system.

Models describe the relationship between one or more measured input signals, *u(t)*, and one or more measured output signals, *y(t)*. Your data can be measured in the time-domain or frequency-domain and have single or multiple inputs and outputs. In real systems, there are additional inputs that you cannot measure or control, which affect the system's output. Such unmeasured inputs are called *disturbances* or *noise*, *e(t)*.



System Identification Toolbox lets you fit different model forms to your data. The most general description of a dynamic system is given by:

$$y(t) = g(u, t, \theta) + v(t)$$

In this case, the output of a system *y(t)* is described by *g*, which might be a function of the time *t*, system parameters θ, and the history of the inputs up to time *t*. *v(t)* is the output noise. For nonlinear models, *g* can take on a variety of forms.

Note Both *e(t)* and *v(t)* represent noise. However, *e(t)* represents the input noise, and *v(t)* represents the output noise.

For linear models, the general symbolic model description is given by:

$$y = Gu + He$$

In this equation, *G* is an operator that takes the input to the output and captures the system dynamics. You can consider *G* to be a transfer function between *u(t)* and *y(t)*. System Identification Toolbox provides a variety of mathematical forms for *G*. *H* is an operator that describes the properties of the additive output disturbance and is the noise model. Estimating a linear model always produces both the dynamic model G and the noise model H.

For an overview of supported model structures, see “Summary of Supported Models” on page 1-4.

Summary of Supported Models

The choice of model form depends on the nature of the dynamic system, on the type of behavior that is expected, and on the intended use of the model. In some cases, a specific form is preferable because the estimated parameters having physical interpretation. If you require estimates of dynamic characteristics without detailed parametric models, you can use nonparametric models.

If you understand the physics of your system and can represent the system using an ordinary differential equation (ODE), then you can use System Identification Toolbox to perform linear or nonlinear grey-box modeling. A *grey-box model* is a model where the mathematical structure of the model and possibly some of the parameters are already known. You capture the model ODE and the parameters you want to estimate in an m-file or MEX-file, and then use System Identification Toolbox objects, methods, and functions to estimate the model parameters.

In many real-world situations, it is too difficult to describe a system using known physical laws. In such cases, you can use System Identification Toolbox to perform black-box modeling. A *black-box model* is a flexible structure that is capable of describing many different systems and its parameters might not have any physical interpretation.

System Identification Toolbox supports the following types of linear and nonlinear models:

- Low-order, continuous-time transfer functions. See “Low-Order, Continuous-Time Process Models” on page 5-4.
- Linear, nonparametric models, including transient-response and frequency-response estimates. See “Correlation Analysis Models” on page 5-23 and “Spectral Analysis Models” on page 5-31.

- Linear, polynomial models, including ARX, ARMAX, Box-Jenkins, and Output-Error models. See “Black-Box Polynomial Models” on page 5-42.
- Linear state-space models with free, canonical, and structured parameterizations. See “State-Space Models” on page 5-67.
- Time-series models. See “Time-Series Models” on page 5-94.
- Nonlinear ARX and Hammerstein-Wiener models. See Chapter 6, “Estimating Nonlinear Black-Box Models”.
- Linear or nonlinear grey-box models, represented as ordinary differential equations (ODEs) or as ordinary difference equation. See Chapter 7, “Estimating Grey-Box Models”.
- Recursive parameter estimation of black-box polynomial models. See Chapter 8, “Recursive Parameter Estimation”.

Getting Started with System Identification offers several tutorials to help you quickly get started estimating linear models.

Definition of a Model Object

When you estimate a model in System Identification Toolbox, all information about this model is stored in a model object. *Model objects* are entities that 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.

If you work in the MATLAB Command Window, you only need a basic understanding of classes, constructors, and methods to work with model objects in System Identification Toolbox and to navigate the reference pages. For more information, see “Working with Model Objects” on page 1-19.

If you work in the System Identification Tool graphical user interface (GUI), you create and operate on model objects graphically using menu options and fields. New users should start by using the System Identification Tool to become familiar with the Toolbox.

Whenever you estimate a linear model, the resulting model object includes two components: the dynamic model G for measured data and the noise model *H* for unmeasured disturbances. You can extract the dynamic and noise models from the estimated model using subreferencing, as described in "Subreferencing Models" on page 1-36.

Note In addition to model objects, System Identification Toolbox uses data objects to represent time- and frequency-domain data. For detailed information about working with data objects, see Chapter 3, "Representing Data for System Identification".

This User's Guide describes both the GUI approach and the MATLAB Command Window approach for system identification. Which environment you choose is largely a matter of preference. However, some advanced functions can only be performed from the MATLAB Command Window. For more information, see the introductory chapter in the Getting Started guide.

Overview of Estimating Models

System identification is typically a trial-and-error process, where you estimate and validate different types of models until you find the simplest model that adequately captures the dynamics of your system.

System Identification Toolbox handles both time-domain and frequency-domain data. There are slight differences in how you estimate black-box models for each data domain. These differences are related to whether you want to get continuous-time or discrete-time models.

Note Nonlinear black-box models support only time-domain data.

This section provides information about supported model types and describes the following topics:

For detailed information about estimating specific types of models, see the following chapters:

- Chapter 5, "Estimating Linear Nonparametric and Parametric Models"
- Chapter 6, "Estimating Nonlinear Black-Box Models"
- Chapter 7, "Estimating Grey-Box Models"
- Chapter 8, "Recursive Parameter Estimation"

If you are primarily working in the MATLAB Command Window, see "Working with Model Objects" on page 1-19 for general information about commands for creating and manipulating models.

General Strategy for Model Estimation

Because System Identification Toolbox lets you estimate different model structures in a brief period of time, you should try many different structures to see which one gives the best results.

Before you begin estimation, import your data into MATLAB and represent the data in System Identification Toolbox format. If you prefer to use a graphical user interface (GUI), import the data into the System Identification Tool. If you prefer to work in the MATLAB Command Window, then represent your data as an `iddata` or `idfrd` object. For more information about representing your data for system identification, see Chapter 3, "Representing Data for System Identification".

To explore the basic properties of your dynamic system, you can begin by estimating linear nonparametric models. Correlation-analysis models estimate the impulse- and step-response of the system. Spectral-analysis models estimate the frequency-response of the system. Exploring the characteristics of these nonparametric models can help you select model orders and delays for your parametric models. For more information, see "Correlation Analysis Models" on page 5-23 and "Spectral Analysis Models" on page 5-31.

If you can explicitly represent your system as an ordinary differential equation with unknown parameters, you can estimate linear or nonlinear grey-box models. Grey-box modeling is useful when you know the relationships between variables, constraints on model behavior, or explicit equations of change. System Identification Toolbox supports both single-output and multioutput grey-box modeling.

- For linear models, you write an m-file to fit both time-domain and frequency-domain data and return state matrices as a function of user-defined parameters and information about the model.

- For nonlinear models, you write an m-file or MEX-file to fit time-domain data and return the derivatives of the states and output values as a function of the states, inputs, time, parameters, and auxiliary variables.

For more information, see Chapter 7, “Estimating Grey-Box Models”.

If you cannot use grey-box modeling because it is not possible or too time-consuming to construct ordinary differential equations for your system, you can estimate black-box models. For an overview of available black-box models, see the section on estimating dynamic models in *Getting Started with System Identification Toolbox*. You can also find strategies for modeling multioutput systems in the Getting Started guide.

In some cases, you can improve your initial results by refining the model using an iterative algorithm. For more information about refining models, see “Refining Models” on page 1-46.

Validate each model directly after estimation to help you 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 Chapter 9, “Plotting and Validating Models”.

After you have selected a good model to represent your system, see Chapter 10, “Postprocessing and Using Estimated Models”.

Supported Models for Time-Domain and Frequency-Domain Data

System Identification Toolbox supports model estimation using both time-domain and frequency-domain data. This section helps you decide which types of models you can estimate for a specific data domain.

Time-domain data is one or more input variables *u(t)* and one or more output variables *y(t)*, sampled as a function of time. A special case of time-domain data is time-series data, which is one or more outputs *y(t)* and no input. *Frequency-domain data* is the Fourier transform of the input and output time-domain signals. *Frequency-response data*, also called *frequency-function data*, represents complex frequency-response values for a linear system characterized by its transfer function *G*.

You can measure frequency-response data values directly using a spectrum analyzer, for example. In this section, *frequency-domain* and *frequency-response* are both referenced as frequency-domain data for the sake of brevity.

For grey-box models, you can estimate both continuous-time and discrete-time models. If the grey-box model is linear, both time-domain and frequency-domain data are supported. If the grey-box model is nonlinear, only time-domain data is supported.

Note Frequency-domain data is not relevant to nonlinear models. Thus, nonlinear models support only time-domain data.

For black-box models, the types of models you can estimate for time-domain and frequency-domain data are described in the following table.

Data	Discrete-Time Model	Continuous-Time Model
Time-Domain	You can estimate any linear and nonlinear discrete-time model supported by System Identification Toolbox.	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. You can estimate the following types of continuous-time models directly: • Low-order, continuous-time process models. • Continuous-time, state-space models. In this case, the `SSparameterization` property of the model object must be set to `canonical` or `structured` parameterizations. You must set the model sampling-interval property `Ts` to `0` before or during estimation. For more information about these properties, see the `idss` reference pages.
Frequency-Domain	You can estimate only ARX and Output-Error (OE) polynomial models using frequency-domain data. Other model structures include noise models, and noise models are not supported for frequency-domain data. You must set the data property `Ts` to the experimental data sampling interval. Setting `Ts` to `0` corresponds to taking Fourier transforms of continuous-time data.	You can estimate the following types of models: • From continuous-time data, you can directly estimate continuous-time ARX and Output-Error (OE) polynomial models. • From continuous-time data, you can estimate continuous-time state-space models. From discrete-time data, you can estimate continuous-time black-box models with canonical parameterization by setting the model sampling-interval property `Ts` to `0`.

For more information about available continuous-time and discrete-time model structures in this Toolbox, see "Supported Continuous-Time and Discrete-Time Models" on page 1-12.

Note You can estimate a linear model using time-domain data, and then validate the model using frequency domain data. If necessary, you can convert frequency-domain data to time-domain data using `ifft`.

Supported Continuous-Time and Discrete-Time Models

For linear and nonlinear grey-box models, you can specify any ordinary differential or difference equation to represent your continuous-time or discrete-time model, 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
Linear Process Models	Estimate low-order models (up to three free poles) for either time- or frequency-domain data.
Linear, Black-Box Polynomial Models • ARX • ARMAX • Output-Error • Box-Jenkins	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 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. To denote continuous-time frequency-domain data, set the data sampling-interval property `Ts` to 0.

Supported Continuous-Time Models (Continued)

Model Type	Description
State-Space Models	Use any one 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. • Set the set the model sampling-interval property `Ts` to `0`, and set the `SSparameterization` property of the model object to `canonical` or `structured` parameterizations. For more information about these properties, see the `idss` reference pages. • Use continuous-time frequency-domain data with the data property `Ts` set to `0`. In this case, no disturbance model can be estimated.
Linear Grey-Box Models	Estimate ordinary differential equations (ODE) for either time- or frequency-domain data.
Nonlinear Grey-Box Models	Estimate arbitrary differential equation (ODE) for time-domain data.

Supported Discrete-Time Models

Model Type	Description
Linear Black-Box Models • ARX • ARMAX • Output-Error • Box-Jenkins • State-Space	Estimate arbitrary-order, linear parametric models for time- or frequency-domain data.You must set the data property `Ts` to the experimental data sampling interval.
Nonlinear Black-Box Models • Nonlinear ARX • Hammerstein-Wiener	Estimate for time-domain data only.
Linear Grey-Box Models	Estimate ordinary difference equations for time- or frequency-domain data.
Nonlinear Grey-Box Models	Estimate ordinary difference equations for time-domain data.

How Noise Affects Model Choice

System Identification Toolbox lets you estimate a noise model for linear models structures. For information on how to decide whether you need to estimate a noise model, see the section on estimating dynamic models in *Getting Started with System Identification Toolbox*.

Note Nonlinear ARX and Hammerstein-Wiener models do not produce parametric noise models.

If you decide that a good noise model is important, choose the ARMAX, Box-Jenkins, or state-space model structures that include additional parameters for modeling noise.

Output-Error (OE) and ARX models are not sufficiently flexible for modeling noise. Output-Error models give a trivial noise model with H=1, and ARX models give a noise model that is coupled to the dynamics via the A polynomial.

How Feedback Affects Model Choice

All prediction-error methods for estimating models work equally well for systems with and without feedback when you estimate a model structure that includes a flexible noise model. Examples of structures with flexible noise models include ARMAX, BJ, and state-space models. However, some estimation methods might be unreliable if your system operates in a closed loop such that the past outputs affect the current inputs.

This section discusses the following topics:

- "Unreliable Models in the Presence of Feedback" on page 1-15
- "Detecting Feedback in the Data" on page 1-16

Unreliable Models in the Presence of Feedback

The following models are unreliable when feedback is present in your system:

- Nonparametric correlation-analysis models, estimated by `cra`.

  If you estimate the impulse response using `impulse`, the response before time equal to `0` is caused by the feedback mechanism and does not represent system dynamics.
- Nonparametric spectral-analysis models, estimated by `etfe`, `spa`, or `spafdr`.
- State-spate models estimated using the noniterative estimation method `n4sid`.
- Model structures that have inaccurate noise models, such as Output-Error polynomial models (OE) and state-space models with the property `DisturbanceModel` set to `None`.

Detecting Feedback in the Data

If you are unsure about the presence of feedback, you can use System Identification Toolbox to detect feedback in your data:

- Use the `advice` command on your data set. Also, you can use the `feedback` command to get detailed information about the nature of the feedback.
- Use the `impulse` command on your data set to plot the estimated impulse response. Significant values of the impulse response at negative lags might indicate feedback.
- On residual analysis plots, significant correlation between residuals and inputs at negative lags indicates feedback. For more information about residual analysis, see "Residual Analysis Plots" on page 9-15.

Supported Estimation Algorithms

System Identification Toolbox provides the following three types of estimation algorithms:

- "Nonparametric Estimation Algorithms" on page 1-16
- "Noniterative Algorithms for State-Space, ARX, and AR Models" on page 1-16
- "Prediction-Error Algorithm for Parametric Models" on page 1-17

Nonparametric Estimation Algorithms

Correlation-analysis and spectral-analysis algorithms, also called *nonparametric estimation algorithms*, provide direct estimation of transient and frequency response of the system, respectively. These algorithms only assume that the system is linear and do not impose a specific model structure.

For more information about nonparametric estimation, see "Correlation Analysis Models" on page 5-23 and "Spectral Analysis Models" on page 5-31.

Noniterative Algorithms for State-Space, ARX, and AR Models

Noniterative algorithms in System Identification Toolbox includes linear least-squares, instrumental-variable, and subspace methods.

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 `n4sid`), and then try to refine the initial estimate using the iterative prediction-error method PEM (see `pem`). N4SID is faster than PEM, but is typically less accurate and robust, and requires additional arguments that might be difficult to specify. For more information about estimating state-space models, see "State-Space Models" on page 5-67.

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, 1999.

The ARX and IV algorithms differ in the way they treat noise. ARX assumes white noise. However, the instrumental variable algorithm, IV, is not sensitive to noise color. Thus, use IV when the noise in your system is not completely white and it is incorrect to assume white noise. If the models you obtained using ARX are inaccurate, try using IV.

Note AR models apply to time-series data, which has no input. For more information, see "Time-Series Models" on page 5-94. For more information about working with AR and ARX models, see "Black-Box Polynomial Models" on page 5-42.

Prediction-Error Algorithm for Parametric Models

You can use the iterative *prediction-error minimization* (*PEM*) (maximum likelihood) algorithm for all linear and nonlinear model types.

If you are using the System Identification Tool GUI, you can specify PEM for low-order continuous-time process models, linear state-space, and polynomial models. If you are working in the MATLAB Command Window, you can use the `pem` command to both construct and estimate these linear models and to also estimate linear and nonlinear grey-box models.

Alternatively, you can use PEM to try to refine initial parameter estimates for all linear and nonlinear parametric models. For more information about refining initial model estimates, see "Refining Models" on page 1-46.

PEM uses optimization to minimize the *cost function*, defined as follows for scalar outputs:

$$V_N(G,H) = \sum_{t=1}^{N} e^2(t)$$

where *e(t)* is the difference between the measured output and the predicted output of the model. For a linear model, this error is defined by the following equation:

$$e(t) = H^{-1}(q)\left[y(t) - G(q)u(t)\right]$$

e(t) is a vector and the cost function $V_N(G,H)$ is a scalar value. The subscript *N* indicates that the cost function is a function of the number of data samples and becomes more accurate for larger values of *N*. For multioutput models, the previous equation is more complex.

For black-box models, PEM estimates an initial model and then varies the parameter values along a specific direction to decrease the cost function. As with any nonlinear optimization algorithm, there is a chance that the model might find a local minimum that is not accurate for a specific system.

Working with Model Objects

When you estimate a model in System Identification Toolbox, all the information about this model is stored in a model object. *Model objects* store model information, such as 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. A model object offers the convenience of manipulating the model and model properties as a single entity.

System Identification Toolbox provides different model objects to represent the supported model types. When you are working in the MATLAB Command Window or writing m-file scripts, you create and operate on model objects directly. For a tutorial on estimating models in the MATLAB Command Window, see *Getting Started with System Identification Toolbox*.

The section discusses the following topics:

- "Basic Object-Oriented Concepts" on page 1-19
- "Types of Model Objects" on page 1-21
- "Commands for Model Estimation" on page 1-23
- "Example – Estimating Model Objects" on page 1-24
- "Model Properties" on page 1-30
- "Subreferencing Models" on page 1-36
- "Concatenating Model Objects" on page 1-41
- "Merging Model Objects" on page 1-44

Basic Object-Oriented Concepts

There are four basic concepts you need to get started working with objects in System Identification Toolbox. These concepts include classes, methods, constructors, and properties.

Model 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.

System Identification Toolbox includes nine different classes for representing models. For example, `idpoly` represents linear black-box polynomial models, and `idss` represents linear state-space models. For a complete list of available model objects, see "Types of Model Objects" on page 1-21.

The way a model object stores information is defined by the *properties* of the corresponding class. For example, the `idpoly` model object has a property called `InputName` for storing one or more input channel names. The set of available properties differs for different model objects.

The allowed operations on an object, called *methods*, are also defined by the corresponding class. In System Identification Toolbox, some methods have the same name but apply to multiple model objects. For example, the method `bode` create a bode plot for all linear model types. 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 method for creating objects based on this class, called a *constructor*. Using a constructor creates a specific instance of the corresponding class and is called *instantiating the object*. In System Identification Toolbox, 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 constructor for instantiating the model object with specific property values.

You use model constructors to create a model object by specifying all required model properties explicitly. You use the constructed model for simulation or as an initial guess for iterative estimation. In most cases, you use the estimation commands instead of constructors. These estimation commands both construct the model object and estimate the model parameters. For an example, see "Example – Estimating Model Objects" on page 1-24.

Note Although the basic definitions in this section are formulated in the context of model object, they also apply to the data objects used to represent data in System Identification Toolbox. For detailed information about working with data objects, see Chapter 3, "Representing Data for System Identification".

Types of Model Objects

The following table summarizes the model classes available in System Identification Toolbox for representing various types of models. This table also specifies whether a specific model type supports single or multiple outputs.

For information on how to both construct and estimate models with a single command, see "Commands for Model Estimation" on page 1-23. After estimating models in System Identification Toolbox, you can recognize these model objects in the MATLAB workspace by their class names.

If you need to create a model to simulate data or to specify a model structure with initial parameters, use a model constructor to create the object. For more information, see the corresponding reference pages for each model object. The name of the object is the same as the name of the constructor for that object.

Summary of Model Classes

Model Class	Model Type	Single Output or Multiple Outputs?
`idarx`	Represents parametric multiple-output ARX models. Also represents nonparametric transient-response models.	Single- or multiple-output models.
`idfrd`	Represents nonparametric frequency-response model.	Single- or multiple-output models.
`idproc`	Represents continuous-time, low-order process models.	Single-output models only.

Summary of Model Classes (Continued)

Model Class	Model Type	Single Output or Multiple Outputs?
`idpoly`	Represents linear, black-box polynomial models: • ARX • ARMAX • Output-Error • Box-Jenkins	Single-output models only.
`idss`	Represents linear, state-space models.	Single- or multiple-output models.
`idgrey`	Represents linear state-space model (grey-box models) in terms of your own variables and parameters. You write an m-file that translates user parameters to state-space matrices.	Single- and multiple-output models.
`idnlgrey`	Represents nonlinear, grey-box models. You write an m-file or MEX-file to represent the set of first-order differential or difference equations.	Supports single- and multiple-output models.
`idnlarx`	Represents nonlinear ARX models, which define the predicted output as a nonlinear function of past inputs and outputs.	Single- or multiple-output models. Does not support time series.
`idnlhw`	Represents 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.

Commands for Model Estimation

The quickest way to both construct a model object and estimate the model in System Identification Toolbox is to use estimation commands. This approach differs from the standard object-oriented approach, where you first use a constructor to instantiate the object, and then use an estimation method to estimate the model.

Note The standard object-oriented approach does apply to grey-box models, where you must define the model structure before estimating the model parameters.

The estimator `pem` corresponds to the iterative prediction-error method, as described in "Supported Estimation Algorithms" on page 1-16. For linear models, you can use `pem` to both construct and estimate a model. You can also use `pem` to refine all initial parametric model estimates, as described in "Refining Models" on page 1-46. For nonlinear models, you can only refine the models you estimated using `nlarx` and `nlhw`.

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.

The following table summarizes System Identification Toolbox estimation methods for parametric models. In this table, `pem` is listed in those case where you can use this command to both construct and estimate a model. For detailed information about using each estimation command, see the corresponding reference pages.

Construction and Estimation Commands

Model Type	Model Estimation Command
Continuous-time low-order process models	`pem`

Construction and Estimation Commands (Continued)

Model Type	Model Estimation Command
Linear black-box polynomial models: • ARX • ARMAX • Box-Jenkins (BJ) • Output-Error (OE)	`armax` (ARMAX only) `arx` (ARX only) `bj` (BJ only) `iv4` (ARX only) `oe` (OE only) `pem` (for all models)
Linear state-space models	`n4sid` `pem`

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

Nonlinear ARX	`nlarx`
Hammerstein-Wiener	`nlhw`

Example – Estimating Model Objects

In System Identification Toolbox, 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 System Identification Toolbox, see “Commands for Model Estimation” on page 1-23.

1 Load a sample data set `z8` with three inputs and one output, measured at `1` sec 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 pages.

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 MATLAB prompt:

```
m_armax
```

MATLAB provides the following information about this model object:

```
Discrete-time IDPOLY model: A(q)y(t) = B(q)u(t) + e(t)

A(q) = 1 - 1.255q^-1 + 0.2551q^-2 + 0.2948q^-3 - 0.0619q^-4
B1(q) = -0.09168 + 1.105q^-1 + 0.7399q^-2
B2(q) = 1.022 + 0.129q^-1
B3(q) = -0.07605 + 0.08681q^-1 + 0.5619q^-2
C(q) = 1-0.06117q^-1 - 0.1461q^-2 + 0.009862q^-3 - 0.04313q^-4

Estimated using ARMAX from data set z8
Loss function 2.23844 and FPE 2.35202
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 lists the following object properties and values:

```
ans =
                      a: [1 -1.2549 0.2551 0.2948 -0.0619]
                      b: [3x3 double]
                      c: [1 -0.0612 -0.1461 0.0099 -0.0431]
                      d: 1
                      f: [3x1 double]
                     da: []
                     db: [3x0 double]
                     dc: []
                     dd: []
                     df: [3x0 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: 0.9932
             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 =
             Approach: 'Pem'
                Focus: 'Simulation'
              MaxIter: 50
            Tolerance: 1.0000e-005
           LimitError: 1.6000
              MaxSize: 'Auto'
      SearchDirection: 'Auto'
       FixedParameter: []
                Trace: '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)'
             Method: 'ARMAX'
            LossFcn: 0.9602
                FPE: 1.0263
           DataName: 'z8'
         DataLength: 500
             DataTs: 1
         DataDomain: 'Time'
    DataInterSample: {3x1 cell}
            WhyStop: 'Near (local) minimum, (norm(g)<tol).'
         UpdateNorm: 8.0572e-006
    LastImprovement: '7.4611e-006%'
         Iterations: 4
       InitialState: 'Zero'
            Warning: 'None'
```

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);
```

Model Properties

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

This section discusses the following topics:

- "Categories of Model Object Properties" on page 1-30
- "Specifying Model Properties in the Estimator" on page 1-32
- "Accessing Model Properties" on page 1-33
- "Help on Properties in MATLAB Command Window" on page 1-35

Categories of Model Object Properties

Each model object has properties for storing information that is relevant only to that specific model type. However, the `idarx`, `idgrey`, `idpoly`, `idproc`, and `idss` model objects are based on the `idmodel` superclass and share all `idmodel` properties.

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

In general, all model objects have properties that belong to the following general 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.
- Properties that store estimation results and model uncertainty.
- User comments, `Notes` and `Userdata`.

All model objects (except as notes) have the following properties:

- `Algorithm`

  This structure includes fields that specify how the nonlinear optimization search method works. `Algorithm` includes another structure, called `Advanced`, which provides additional flexibility for setting the optimization algorithm. Different fields apply for different estimation techniques.

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

  Note Does not apply to `idfrd` models.

- `EstimationInfo`

  This structure includes read-only fields that describe the estimation data set, quantitative model quality measures, the number of iterations actually used, how the initial states were handled, and whether any warnings were encountered during the estimation.

For information on how to list available object properties, see “Help on Properties in MATLAB Command Window” on page 1-35.

Specifying Model Properties in the Estimator

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 "Commands for Model Estimation" on page 1-23.

The following commands load the sample data, `z8`, and estimate an ARMAX model. 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.

When using the commands that let you both construct and estimate a model, as described in "Commands for Model Estimation" on page 1-23, 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.

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 functions, see the corresponding reference pages.

Accessing Model Properties

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

MATLAB returns the following result:

```
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
```

MATLAB returns the following result:

```
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 portion of the name you do enter uniquely identifies the property.

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

Help on Properties in MATLAB Command Window

If you need quick assistance on model properties while working in the MATLAB Command Window, 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 object, you must look up for property for both the model object and its superclass.

The following table summarizes the commands for getting help on object properties in the MATLAB Command Window.

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 pages.

`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`.

Help Commands for Model Properties (Continued)

Model Class	Help Commands

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

`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`.

Subreferencing Models

You can create models with subsets of inputs and outputs from existing models with multiple inputs and outputs using subreferencing.

Subreferencing is also useful when you want to generate model plots for only certain channel, such as when you are exploring multioutput models for input channels that have minimal effect on the output.

System Identification 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.

Note Subreferencing nonlinear models is not supported.

This section discusses the following topics:

- "Subreferencing Specific Measured Channels" on page 1-37
- "Subreferencing Measured and Noise Models" on page 1-38
- "Treating Noise Channels as Measured Inputs" on page 1-39

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 two 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 estimate a noise model, System Identification Toolbox includes one noise channel at the input e for each output in your system.

Therefore, 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 the following equation:

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

or

$$y(t) = \frac{B(q)}{A(q)}u\left(t\right) + \frac{1}{A(q)}e(t)$$

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 2-33.

Concatenating Model Objects

You can perform horizontal and vertical concatenation of model objects to grow the number of inputs or outputs in the model.

Note Concatenation is supported for linear models only.

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. For more information, see the corresponding reference pages.

This section discusses the following topics:

- "Horizontal Concatenation of Model Objects" on page 1-41
- "Vertical Concatenation of Model Objects" on page 1-42
- "Concatenating Noise Spectral Data of idfrd Objects" on page 1-43

If you have Control System Toolbox, see "Combining Model Objects" on page 10-24 about additional functionality for combining models.

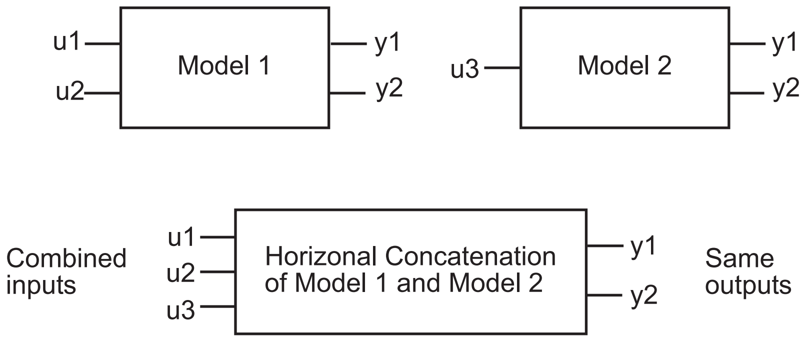
Horizontal Concatenation of Model Objects

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

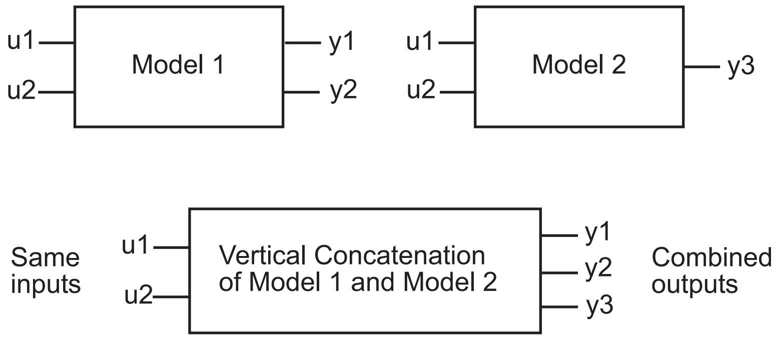
Vertical concatenation combines output channels of specified models. Vertical concatenation of model objects requires that they have the same inputs and frequency vectors. If the input channel names are different and their dimensions are the same, the concatenation operation uses the names of input channels 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 Spectral 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 spectral data does not exist (unless you also entered it manually), `SpectralData` 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 `SpectralData` 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 `SpectralData` property of individual `idfrd` objects is not empty, horizontal and vertical concatenation handle `SpectralData`, as follows.

In case of horizontal concatenation, there is no meaningful way to combine the `SpectralData` of individual `idfrd` objects, and the resulting `SpectralData` 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 `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.

Merging Model Objects

You can merge models of the same structure to obtain a 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 Sets” on page 3-37.

For more information about merging models, see the `merge` reference pages.

Refining Models

There are two situations where you can refine estimates of linear and nonlinear parametric models using the prediction-error method.

In the first situation, you have already estimated a parametric model using any of the available iterative or noniterative methods 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 1 percent.

In the second situation, you constructed a model using one of the model constructors described in “Types of Model Objects” on page 1-21. In this case, you built initial parameter guesses into the model structure and wish to refine these parameter values. This case applies only to linear models and nonlinear grey-box models. Because it is difficult to specify nonlinear model parameters in advance, you typically only estimate nonlinear models.

When you refine a model, you must provide two essential inputs: the parametric model and the 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.

If you are working in the System Identification Tool GUI, you can refine parameter estimates of any linear or nonlinear model already in the GUI. For information on importing models into the GUI, see “Importing Models into the System Identification Tool” on page 2-24.

This section discusses the following topics:

- “Refining Linear Parametric Models in the GUI” on page 1-47
- “Refining Nonlinear Black-Box Models in the GUI” on page 1-48
- “Using pem to Refine Models ” on page 1-49

For more information about the prediction-error algorithm, see “Supported Estimation Algorithms” on page 1-16.

Refining Linear Parametric Models in the GUI

The following procedure assumes that the model you wish to refine is already in the Model Board in the System Identification Tool window. You might have estimated this model in the current session or imported the model from the MATLAB workspace. For more information about estimating linear models, see Chapter 5, “Estimating Linear Nonparametric and Parametric Models”.

1. In the System Identification Tool window, 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 Working Data and Validation Data” on page 2-23.

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 window 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 window 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 9, “Plotting and Validating Models”.

Tip To continue refining the model using additional iterations, click **Continue iter**. This action continues parameter estimation using the most recent model.

Refining Nonlinear Black-Box Models in the GUI

The following procedure assumes that the model you wish to refine is already in the Model Board in the System Identification Tool window. You might have estimated this model in the current session or imported the model into the GUI from the MATLAB workspace. For more information about estimating nonlinear black-box models, see Chapter 6, “Estimating Nonlinear Black-Box Models”.

1 In the System Identification Tool window, 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 Working Data and Validation Data” on page 2-23.

2 Select **Estimate > Nonlinear models** to open the Nonlinear Models dialog box, if this dialog box is not already open.

3 In the Nonlinear Models dialog box, select the model you want to refine in the **Initial model** list.

   The list includes only those models that have the selected **Model structure** and the same number of inputs and outputs as the estimation data in **Working Data** area.

   Any settings in the Nonlinear Models dialog box related to model structure and estimation algorithms are overridden by the selected initial model.

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

5 Validate the new model, as described in Chapter 9, “Plotting and Validating Models”.

Tip To continue refining the model directly from the **Estimation** tab, select the **Use last estimated model as initial model for the next estimation** check box, and click **Estimate**. This action automatically selects the most recent model in the **Initial model** list in the **Model Type** tab.

Using pem to Refine Models

If you are working in the MATLAB Command Window, 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 in the Estimator” on page 1-32.

The following commands provide an example of how 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);
```

The next example demonstrates how to refine models for which you have initial parameter guesses. In this case, you must first create a model object using a constructor method and set the initial parameter values in the model properties. Next, you provide this initial model as input to `pem`. This example estimates an ARMAX model for the data and requires you to initialize the *A*, *B*, and *C* polynomials. For more information about estimating polynomial models, see "Black-Box Polynomial Models" on page 5-42.

```
load iddata8
% Define model parameters
A = [1 -1.2 0.7];
B(1,:) = [0 1 0.5 0.1]; % first input
B(2,:) = [0 1.5 -0.5 0]; % second input
B(3,:) = [0 -0.1 0.5 -0.1]; % third input
C = [1 0 0 0 0];
Ts = 1;
% Leading zeros in B matrix indicate input delay (nk),
% which is 1 for each input channel. The trailing zeros
% in B(2,:)) make the number of coefficients equal
% for all channels.

% 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)
```

Working with the System Identification Tool GUI

Introduction to the System Identification Tool

System Identification Tool is a graphical user interface (GUI) for working with System Identification Toolbox. If you are new to this Toolbox, you can use System Identification Tool to become familiar with the product features and workflow. For tutorials on using the System Identification Tool, see *Getting Started with System Identification Toolbox*. For more information about the system identification workflow, see “Workflow in the System Identification Tool” on page 2-13.

This section discusses the following topics:

- “Opening the System Identification Tool” on page 2-3
- “Overview of the System Identification Tool Window” on page 2-4
- “Accessing Windows in the System Identification Tool” on page 2-11
- “Getting Help” on page 2-11
- “Exiting the System Identification Tool” on page 2-12

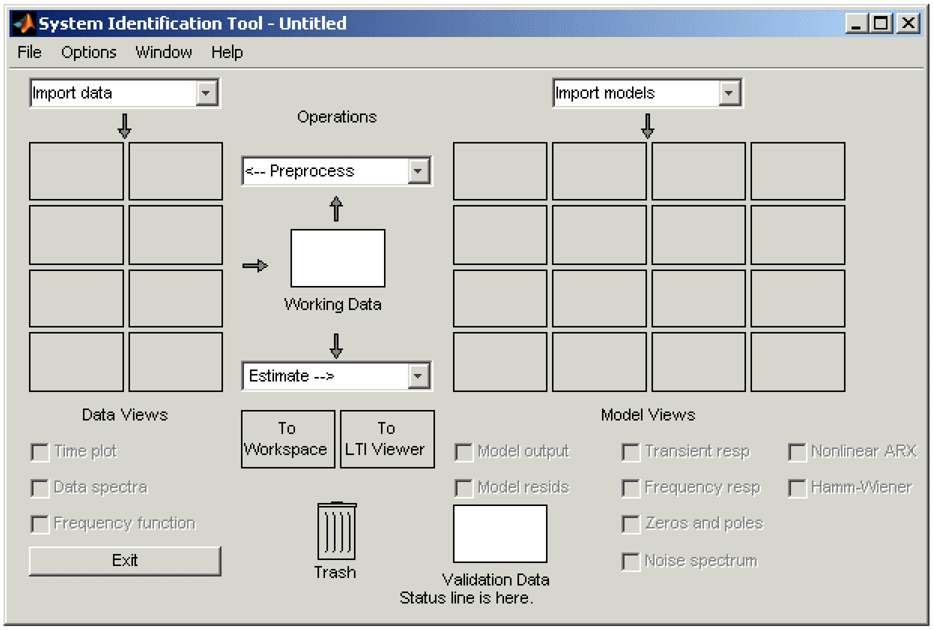
To learn more about the difference between using the Toolbox GUI and functions, see *Getting Started with System Identification Toolbox*.

Opening the System Identification Tool

To open System Identification Tool, type the following command at the MATLAB prompt:

```
ident
```

For a description of this window, see “Overview of the System Identification Tool Window” on page 2-4.



You can also open a previously saved session using the following syntax:

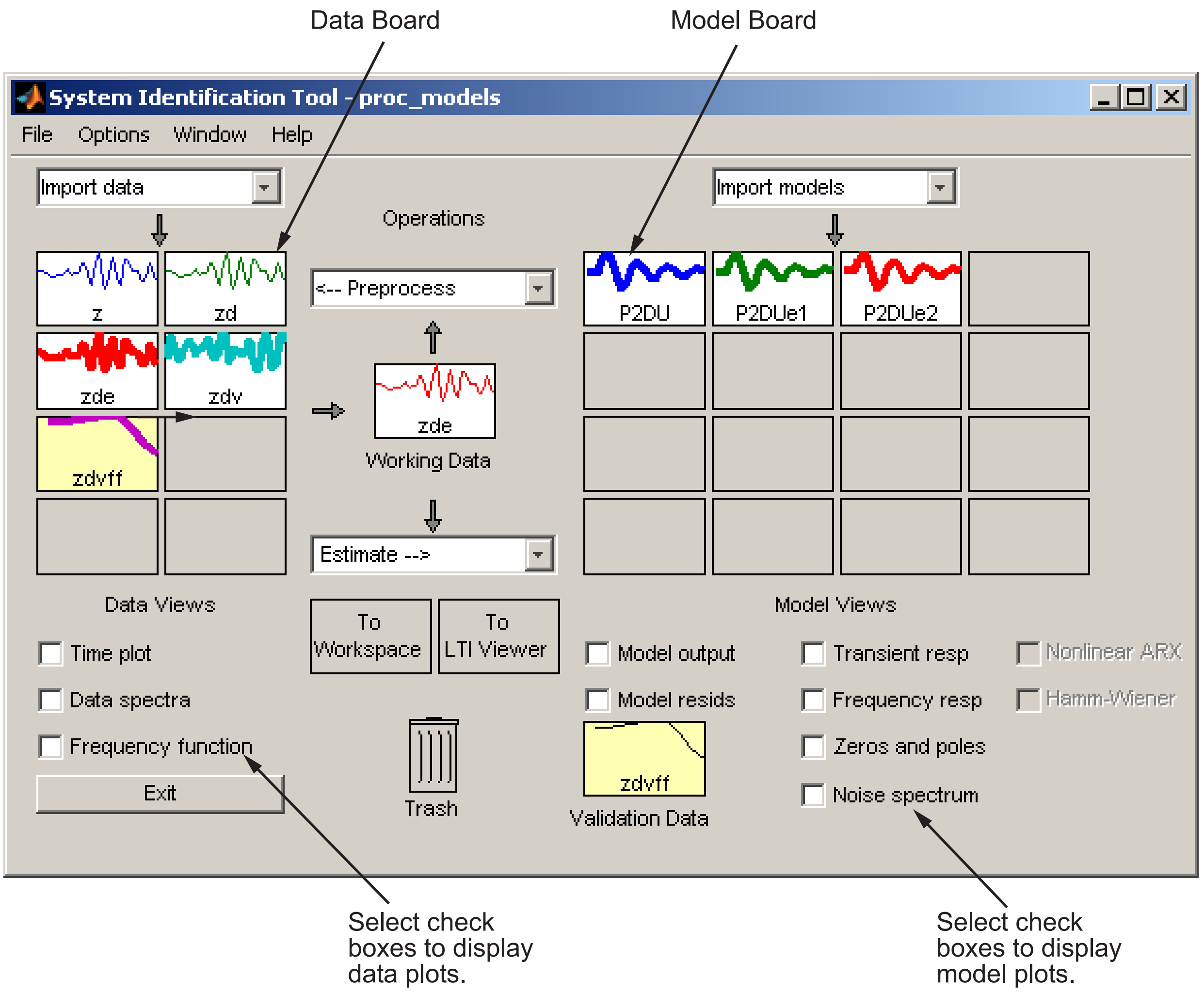
```
ident(session,path)
```

In the preceding command, `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.

You can also start System Identification Tool from the MATLAB Command Window by selecting **Start > Toolboxes > System Identification > System Identification Tool**.

Overview of the System Identification Tool Window

The following figure shows the System Identification Tool window.



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 using the **Import data** menu and end in the bottom-right corner by plotting your model characteristics. For more information about System Identification Tool workflow, see “Workflow in the System Identification Tool” on page 2-13.

This section discusses the following areas of the System Identification Tool window:

- “Data Board and Data Views” on page 2-6
- “Model Board and Model Views” on page 2-9

Data Board and Data Views

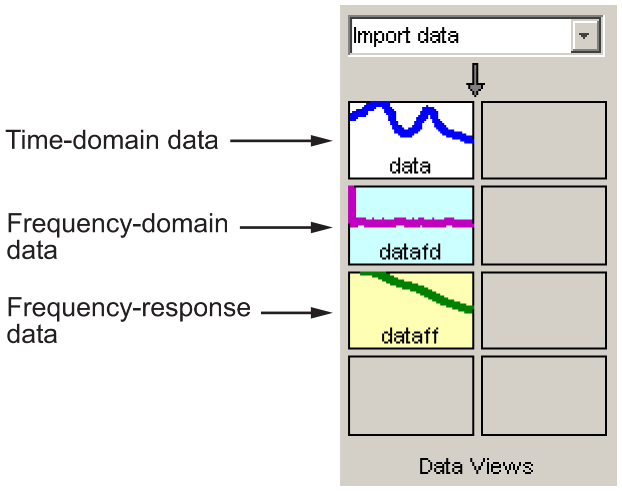
The Data Board, located below the **Import data** menu, contains rectangular icons that represent the data you imported into the System Identification Tool. You can also create new data sets by preprocessing existing data sets using the commands in the **Preprocess** menu. For more information about preprocessing data, see Chapter 4, “Plotting and Preprocessing Data”.

You can drag and drop data icons in the Data Board and into open dialog boxes. To learn how to manage data sets in the System Identification Tool, see “Managing Data Sets and Models” on page 2-21.

Tip When the Data Board is full, you can delete unnecessary data sets by dragging them to the **Trash**. You can also open an additional Data Board window by selecting **Options > Extra model/data board** in the System Identification Tool window.

Each data set has a unique color to help you distinguish it from the other data sets. In addition, the background color of the rectangle is color coded according to the data domain, as follows:

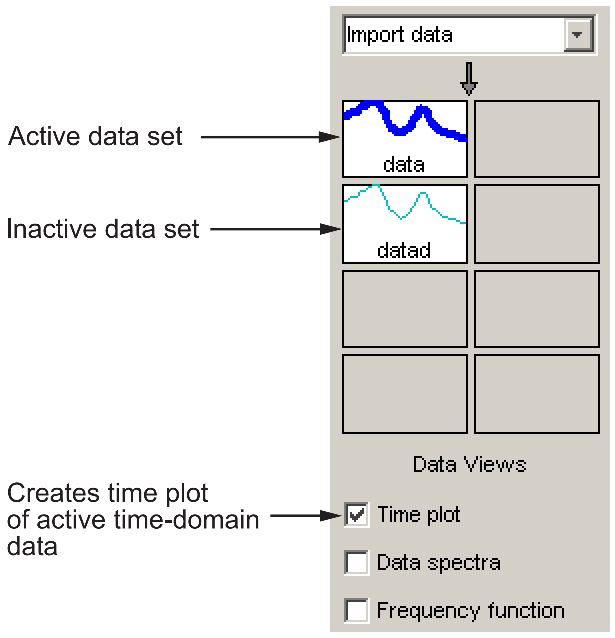
- White background represents time-domain data.
- Blue background represents frequency-domain data.
- Yellow background represents frequency-response data.



Colors Representing Data Domains

The **Data Views** area lets you create plots of data sets that are active in the Data Board. An *active* data set has a thick line in the icon, and an *inactive* data set has a thin line.

In the following figure, the **Time plot** check box is selected to create a time plot of all active time-domain data sets. In this example, only `data` is active and shows in the time plot. For more information on how to work with the plot windows, see "Working with Plots" on page 2-29.



Selected Plot Includes Only Active Data Sets

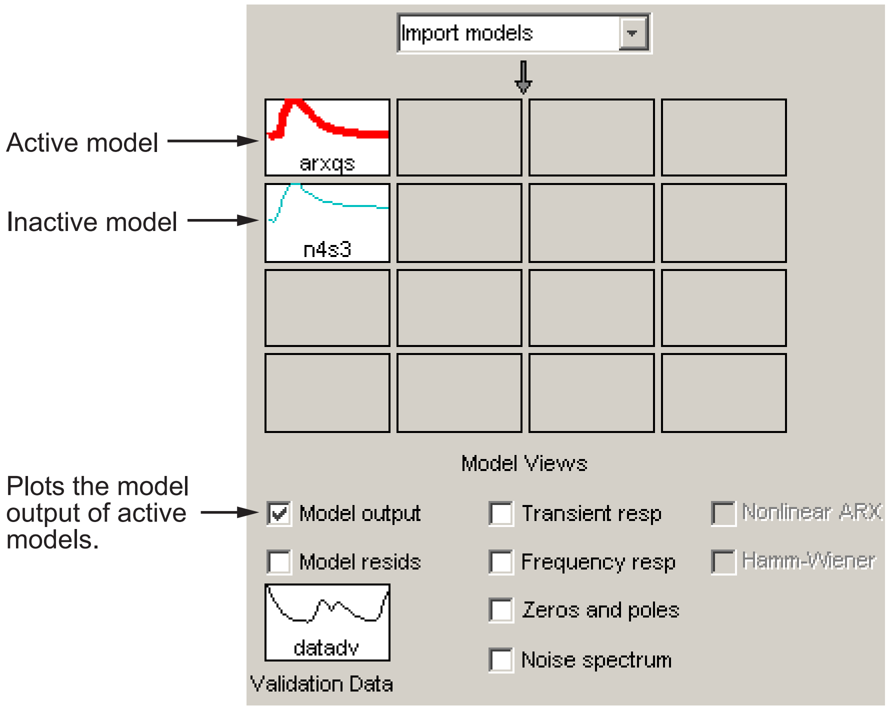
Tip To toggle between excluding and including data in a plot, click the data icon in the Data Board. Clicking the data icon also updates any plots that are currently open by adding or removing the corresponding data from the plot.

Model Board and Model Views

The Model Board, located below the **Import models** menu, contains rectangular icons that represent the models you estimated or imported into the System Identification Tool. You can drag and drop model icons in the Model Board and into open dialog boxes. To learn how to manage models in the System Identification Tool, see "Managing Data Sets and Models" on page 2-21.

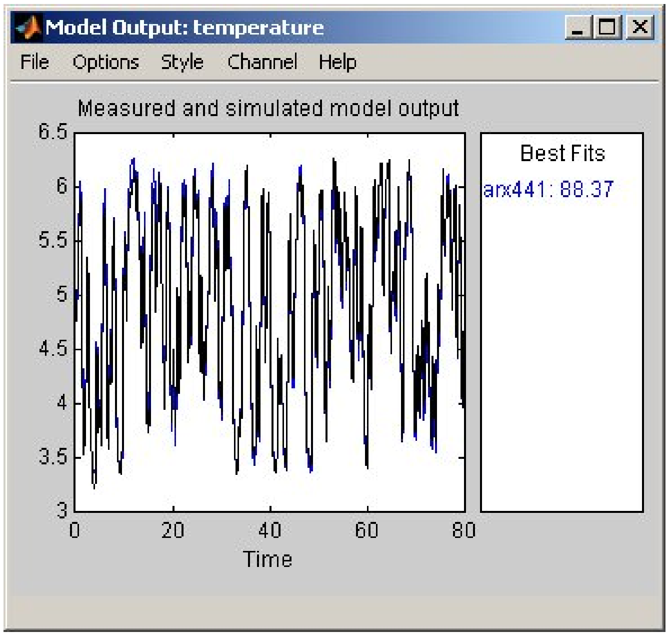
Note When the Model Board is full, you can delete unnecessary models by dragging them to the **Trash**. You can also open an additional Model Board window by selecting **Options > Extra model/data board** in the System Identification Tool window.

The **Model Views** area lets you create plots of models that are active in the Model Board. An *active* model has a thick line in the icon, while an *inactive* model has a thin line. In the following figure, **Model output** is selected to plot the output simulated by the active models. In this example, the plot contains only `arx441` and not `n4s4` because only `arx441` is active. For more information on how to work with the plot windows, see "Working with Plots" on page 2-29.



Selected Plot Includes Only Active Models

Tip To toggle between excluding and including a model in a plot, click the model icon in the Model Board. Clicking the model icon also updates any plots that are currently open by adding or removing the corresponding model from the plot.



Model Output of arx441 (Blue) and Validation Output (Black)

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

Accessing Windows in the System Identification Tool

While you work with System Identification Tool, you might have several dialog boxes open at the same time. To bring a specific window to the top, select it by name from the **Window** menu.

Getting Help

System Identification Tool provides online help topics that you can access from the **Help** menu. Furthermore, contextual help is available from each dialog box by clicking the **Help** button.

Exiting the System Identification Tool

To exit System Identification Tool, click **Exit** in the bottom-left corner of the window. Alternatively, select **File > Exit System Identification Tool**.

For more information about managing your sessions, see “Managing the System Identification Tool Sessions” on page 2-18.

Workflow in the System Identification Tool

A typical workflow in the System Identification Tool consists of the following tasks:

1. Import data from the MATLAB workspace. For more information, see “Creating Data Sets in the System Identification Tool” on page 3-13.
2. Plot the data using **Data Views**. For more information, see “Plotting Data” on page 4-7.
3. Preprocess the data using commands in the **Preprocess** menu. For example, you can remove constant offsets or linear trends (for linear models only), filter data, or select regions of interest. For more information, see Chapter 4, “Plotting and Preprocessing Data”.
4. Select estimation and validation data. For more information, see “Specifying Working Data and Validation Data” on page 2-23.
5. Estimate models using commands in the **Estimate** menu. For information on estimating models, see the following topics:
   - “Correlation Analysis Models” on page 5-23
   - “Spectral Analysis Models” on page 5-31
   - “Low-Order, Continuous-Time Process Models” on page 5-4
   - “Black-Box Polynomial Models” on page 5-42
   - “State-Space Models” on page 5-67
   - “Time-Series Models” on page 5-94
   - Chapter 6, “Estimating Nonlinear Black-Box Models”
6. Validate models using **Models Views**. For more information, see Chapter 9, “Plotting and Validating Models”.
7. Export models to the MATLAB workspace for further processing or for using with other MathWorks products. For more information about exporting models, see “Exporting to the MATLAB Workspace” on page 2-27.

This sequence of tasks is iterative. If your model is not satisfactory, you can try estimating using a different model structure.

Customizing System Identification Tool

System Identification Tool lets you customize the default window behavior and appearance in two ways.

You can configure the window during a session and then save the session state. For example, you can set the size and position of specific dialog boxes and modify the appearance of plots. Advanced users might choose to edit the m-file that controls default settings, as described in “Modifying idlayout.m” on page 2-16.

This section discusses the following topics:

- “Displaying Warnings While You Work” on page 2-15
- “Saving Session Preferences” on page 2-15
- “Modifying idlayout.m” on page 2-16

Displaying Warnings While You Work

In the System Identification Tool window, select **Options > Warnings** to display informational dialog boxes while you work. Verify that a check mark appears to the right of **Warnings** in the menu.

To stop warnings from being displayed during your session, select **Options > Warnings** again to clear the check mark.

Saving Session Preferences

Use **Options > Save preferences** to save the current state of System Identification Tool. This command saves the following settings to System Identification Toolbox preferences file, `idprefs.mat`:

- Size and position of the System Identification Tool window.
- Sizes and positions of the dialog box.
- Four most-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.

Because `idprefs.mat` is a data file, you cannot edit it directly. The `idprefs.mat` file is located in the same directory 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 directory.

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`. Plot options you can customize are:

- Color order of data and model icons. You can later edit the color for a specific data or model in the Data/model Info dialog box. You open this dialog box by right-clicking the corresponding data or model icon in the System Identification Tool window.
- Line colors on plots.
- Axis limits and tick marks.
- Plot options, set in the plot menus.
- Font sizes.

Editing `idlayout.m` lets you change more layout properties than those you can set by saving preferences to `idprefs.mat`.

To customize `idlayout.m` defaults, save a copy of `idlayout.m` to a folder in your `matlabpath` just above the `ident` directory level.

Caution Do not edit the original file to avoid overwriting the `idlayout.m` defaults shipped with the product.

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**.

Managing the System Identification Tool Sessions

System identification is an iterative process that requires you to try different model structures until you find one or more that give the best results. System Identification Tool lets you save your progress, including the data sets and the models, in a session.

You can open a saved session to continue working where you left off. You can also use sessions to organize your work by grouping the models you estimated using a specific approach.

Finally, you can save different stages of your progress as different sessions so that you can revert to any stage simply by opening the corresponding session.

This section includes the following topics:

- “What Is a Session?” on page 2-18
- “Starting a New Session” on page 2-19
- “Commands for Managing Sessions” on page 2-19

What Is a Session?

A *session* is the data and models that are currently in the System Identification Tool window. In a sense, a session is the progress of your system identification problem.

If you opened additional data and model boards, using **Options > Extra model/data board**, these additional windows are part of your current session.

Sessions are saved as files with a `.sid` extension.

Note You can only open one session at a time.

Starting a New Session

When you open System Identification Tool, you start a new session.

You can also start a new session by closing the current session using **File > Close session**. The Toolbox prompts you to save your current session if you have not yet saved it.

To delete a saved session, you must delete the session file from the folder where it was saved.

Commands for Managing Sessions

Task	Command	Description
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.
Open a saved session.	**File > Open session**	If you have data in the System Identification Tool, you must close the current session before you can open a new one.

Task	Command	Description
Save current session.	**File > Save**	Useful for saving the session repeatedly after you have already saved the session for the first time.
Save 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.

Managing Data Sets and Models

While you use System Identification Tool, you might end up with numerous data sets and models in the window. This section describes how to manage your data and models in the GUI and perform the following tasks:

For more information about the Data Board and Model Board in the System Identification Tool window, see "Overview of the System Identification Tool Window" on page 2-4.

Adding Data Sets and Models

There are three ways to add data to the System Identification Tool window:

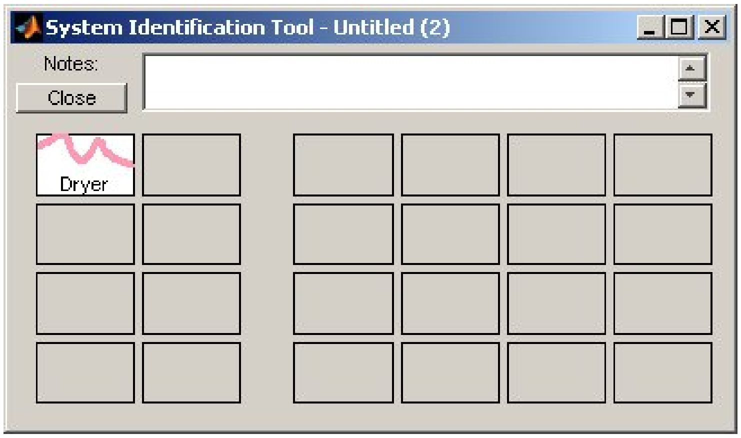
- Import data into the current session. For more information about importing data, see "Importing Data into the System Identification Tool" on page 3-13.
- Merge a saved session into the current session. For more information about merging sessions, see "Commands for Managing Sessions" on page 2-19.
- Preprocess data.

  When you detrend, transform, filter, or select a portion of a data set, this action creates a new data set in the System Identification Tool. For more information about preprocessing data, see Chapter 4, "Plotting and Preprocessing Data".

Similarly, you can add models using the following methods:

- Import model into the current session. For more information about importing models, see “Importing Models into the System Identification Tool” on page 2-24.
- Merge a saved session into the current session. For more information about merging sessions, see “Commands for Managing Sessions” on page 2-19.
- Estimate models from the data. For information on estimating linear models, see Chapter 5, “Estimating Linear Nonparametric and Parametric Models”. For information on nonlinear models, see Chapter 6, “Estimating Nonlinear Black-Box Models”.

You can only display eight data sets and sixteen models to the System Identification Tool window, as shown in the following figure. When you import or create more data sets or estimate more models, an additional window automatically opens.



You can type comments in the **Notes** field, shown in the top portion of the window, to describe the data and models.

When you save a session, as described in “Commands for Managing Sessions” on page 2-19, all additional windows and notes are also saved.

Specifying Working Data and Validation Data

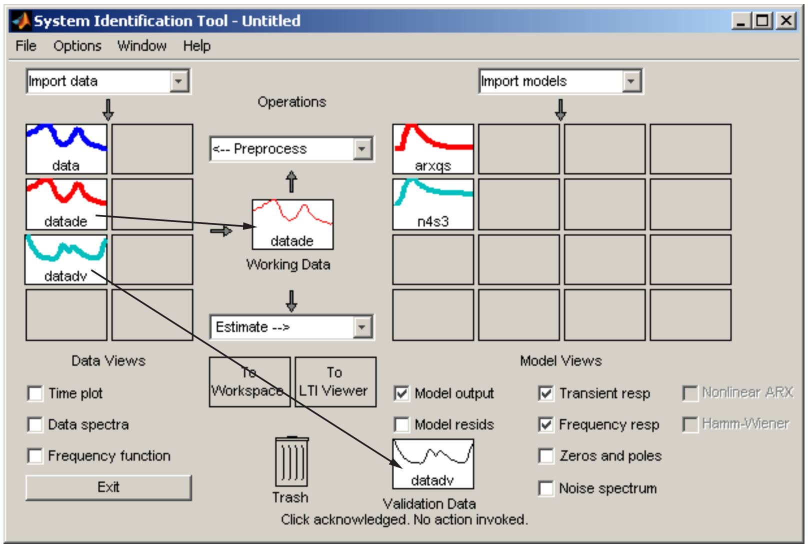
Working data is the data on which you perform preprocessing and estimation operations. When you select commands from the **Preprocess** and **Estimate** menus, these operations are applied to the working data.

Validation data is used to validate a model by comparing it to the model output and perform residual tests. When you plot the model output and residuals, the input to the model is the input signal from the validation data set. These plots compare model output to the measured output in the validation data set.

When you initially import data into the System Identification Tool, as described in "Creating Data Sets in the System Identification Tool" on page 3-13, this data set is automatically designated as both **Working Data** and **Validation Data**.

To specify working data, drag and drop a data set from the Data Board into the **Working Data** rectangle, as shown in the following figure.

Similarly, to specify validation data, drag and drop a data set from the Data Board into the **Validation Data** rectangle.



Tip The data you use to validate a model should differ from the data you use to estimate a model. For more information about validating models, see Chapter 9, "Plotting and Validating Models".

Importing Models into the System Identification Tool

You can import System Identification Toolbox model objects from the MATLAB workspace into the System Identification Tool GUI. If you have Control System Toolbox, you can also import LTI objects defined using Control System Toolbox.

The following procedure describes how to import models into the System Identification Tool. It assumes that you begin with the System Identification Tool window already open. If this window is not open, type the following command at the MATLAB prompt:

```
ident
```

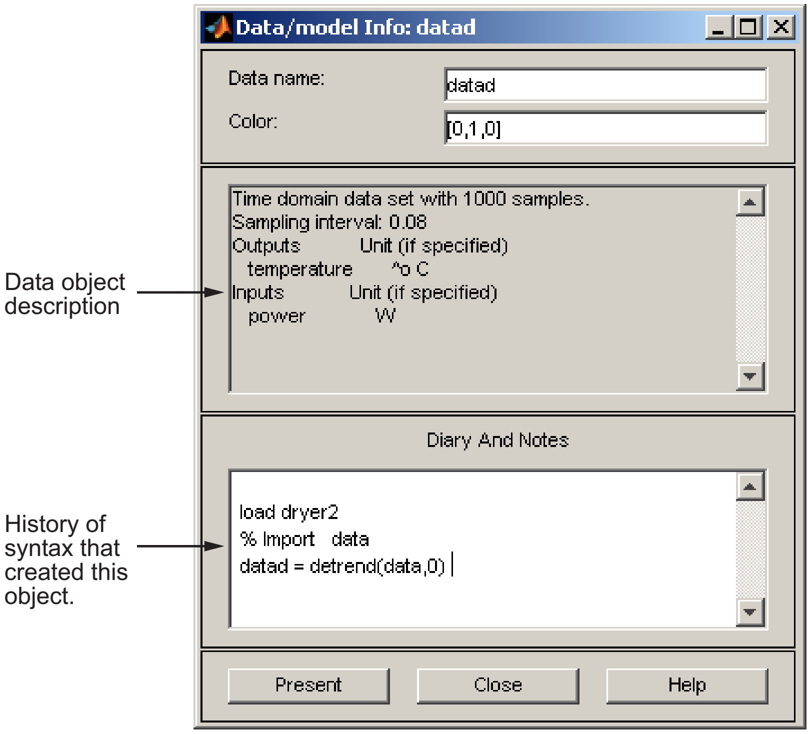
1. In the System Identification Tool window, select **Import** from the **Import models** list to open the Import Model Object dialog box.
2. In the **Enter the name** field, type the name of a model object in the MATLAB workspace. Press **Enter**.
3. (Optional) In the **Notes** field, type any notes you want to store with this model.
4. Click **Import** to add the model to the Model Board in the System Identification Tool window.
5. To close the Import Model Object window, click **Close**.

Viewing Data and Model Properties

You can get information about each data set in the System Identification Tool by right-clicking (or double-clicking) the data icon in the System Identification Tool window.

The Data/model Info dialog box opens. This dialog box describes the contents and the properties of the data or model. It also displays any notes associated with the data or model, including the command-line equivalent of the operations you used to create this data or model.

Tip To view or modify properties for several data sets or models, keep this window open and right-click each data set in the System Identification Tool window. The Data/model Info dialog box updates as you select each data set.



Data name: datad

Color: [0,1,0]

```
Time domain data set with 1000 samples.
Sampling interval: 0.08
Outputs          Unit (if specified)
  temperature      ^o C
Inputs          Unit (if specified)
  power           W
```

```
load dryer2
% Import   data
datad = detrend(data,0)
```

In the Data/model Info dialog box, you can rename the data and models. To rename data, enter a new name in the **Data name** field. To rename models, 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.

Clicking **Present** displays the basic characteristics of this data object in the MATLAB Command Window.

Organizing Data and Model Icons

You can rearrange data or model icons in the System Identification Tool window by dragging and dropping them to an empty rectangle in the Data Board or Model Board, respectively.

If you need additional space for your data and models, select **Options > Extra model/data board** to open an additional window. For more information, see "Adding Data Sets and Models" on page 2-21.

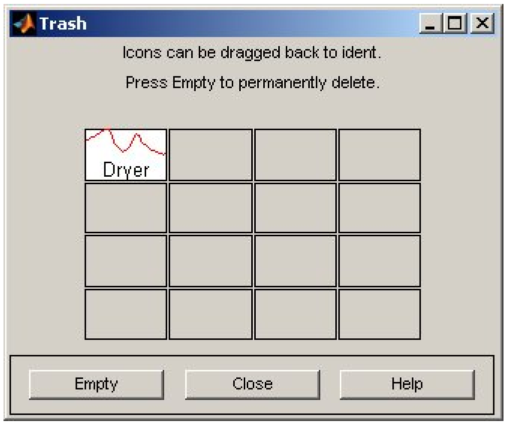
Note You cannot drag and drop a data icon into the Model Board or a model icon into the Data Board.

Deleting Data and Models

To delete data and models in the System Identification Tool window, drag and drop them 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 specify a different data set in the System Identification Tool window to be **Working Data** or **Validation Data**, as described in "Specifying Working Data and Validation Data" on page 2-23. Then, you can delete the unwanted data.

To restore a data set or a model from **Trash**, drag its icon from **Trash** to the Data or Model Board in System Identification Toolbox window. You can view the discarded contents in the Trash window by double-clicking the **Trash** icon, as shown in the following figure.



Icons can be dragged back to ident.

Press Empty to permanently delete.

Note You must restore data to the Data Board; you cannot drag data icons to the Model Board. Similarly, you must restore a model to the Model Board in System Identification Toolbox window.

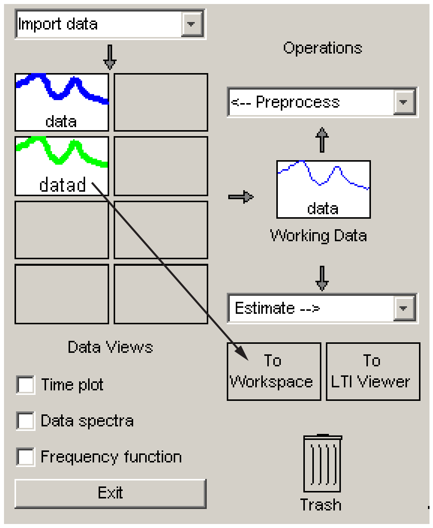
To permanently delete all data sets and models in **Trash**, select **Options > Empty trash**.

Exiting a session empties **Trash** automatically.

Exporting to the MATLAB Workspace

The data and models you create in the System Identification Tool are not available in the MATLAB workspace until you export them. Exporting to MATLAB is necessary when you need to perform an operation on the data or model that is only available using command-line syntax. You might also want to export your model to Simulink® or another Toolbox, such as Control System Toolbox.

To export a data set or model to the MATLAB workspace, drag and drop the corresponding icon to the **To Workspace** rectangle. For example, the following figure shows how to export the time-domain data object `data` to the MATLAB workspace.



Exporting Data to the MATLAB Workspace

The MATLAB workspace now contains a variable named `data`, which is an `iddata` object.

When you export data and model to the MATLAB workspace, the resulting MATLAB variables have the same name as in the System Identification Tool. Exported frequency-response data is an `idfrd` data object. Exported models are model objects. For more information about model objects and how to work with them, see “Working with Model Objects” on page 1-19.

Working with Plots

System Identification Toolbox lets you plot data sets and models while you work in the System Identification Tool. This section describes how to create plots in the System Identification Tool window and describes the plot options that are common to all plot types.

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

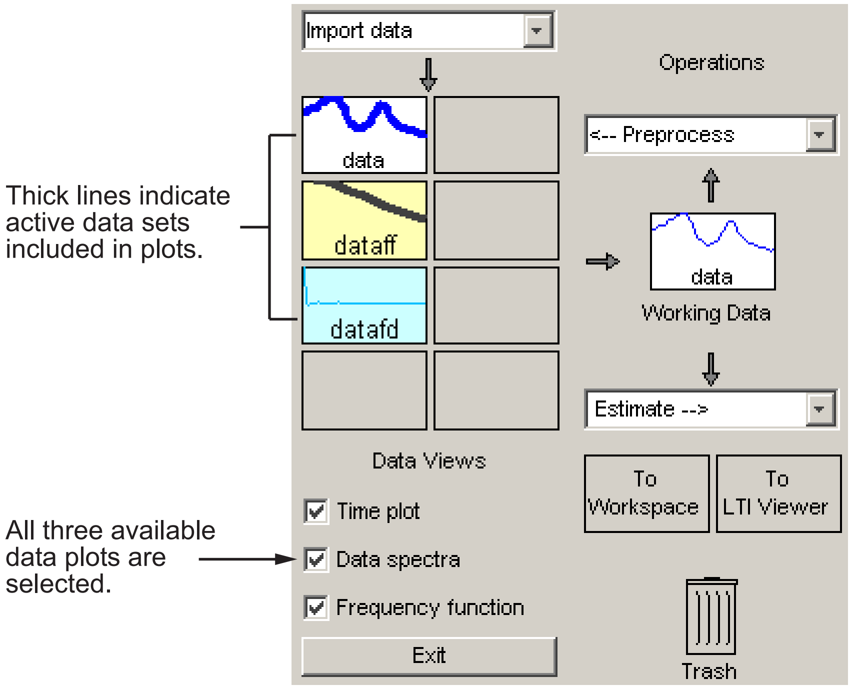
This section discusses the following topics:

- "How to Create a Plot" on page 2-29
- "Identifying Data Sets and Models on Plots" on page 2-30
- "Changing and Restoring Default Axis Limits" on page 2-31
- "Selecting Measured and Noise Channels in Plots" on page 2-33
- "Grid, Line Styles, and Redrawing Plots" on page 2-34
- "Opening a Plot in a MATLAB Figure Window" on page 2-35
- "Printing Plots" on page 2-35

How to Create a Plot

To create one or more plots, select the corresponding check box in the **Data Views** area or the **Model Views** area of the System Identification Tool window. An *active* data or model icon has a thick line in the icon, while an *inactive* data set has a thin line.

Only active data sets and models appear on the selected plots. To toggle including and excluding data or models on a plot, click the corresponding icon in the System Identification Tool window.



Identifying Data Sets and Models on Plots

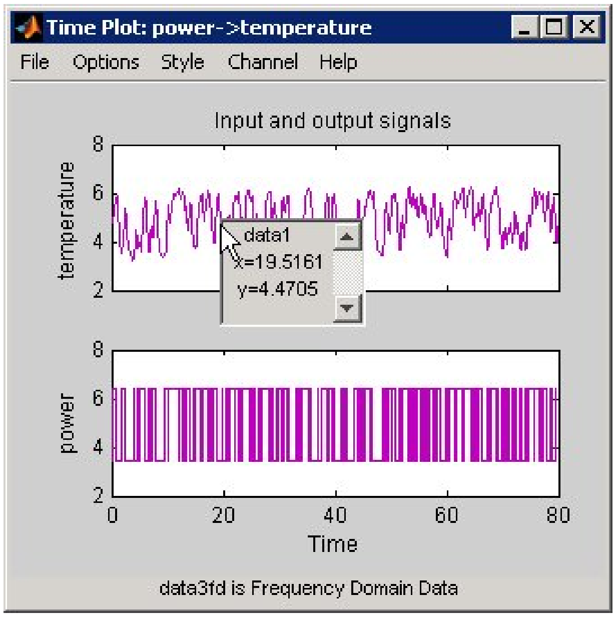
You can identify data sets and models on plot by their color: the color of the line in the data or model icon in the System Identification Tool window matches the line color on the plots. You can also display data tips for each line on the plot.

How you display the data tip depends on whether the zoom feature is enabled:

- When zoom in enabled, press and hold down **Shift**, and click the desired curve to display a data tip on a plot.
- When zoom is disabled, click on a plot curve and hold down the mouse button to display the data tip.

For more information about enabling zoom, see “Using the Zoom Feature” on page 2-31.

The following figure shows an example of a data tip, which contains the name of the data set and the coordinates of the point on the curve you click.


Time Plot: power->temperature
File Options Style Channel Help
Input and output signals
temperature
power
Time
0 20 40 60 80
2 4 6 8
data1
x=19.5161
y=4.4705
data3fd is Frequency Domain Data

Data Tip on a Plot

Changing and Restoring Default Axis Limits

There are two ways to change the portion of the plot currently in view. This section describes how to change plot axis limits and contains the following topics:

- "Using the Zoom Feature" on page 2-31
- "Setting Axis Limits" on page 2-32

Using the Zoom Feature

Enable the zoom feature by selecting **Style > Zoom** in the plot window. To disable zoom, select **Style > Zoom** again.

Tip To verify that this feature is enabled, click the **Style** menu to open it. A check mark should appear next to **Zoom**.

You can adjust zoom in the following ways:

- To zoom in default increments, left-click on the portion of the plot you want to center in the plot window.
- To zoom in 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 following format: *[LowerLimit UpperLimit]*. Click **Apply**.

   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 restore the full range of the data in view, select **Options > Autorange** in the plot window.

Selecting Measured and Noise Channels in Plots

Input and output variables are called *channels*. When you create a plot of a multivariate input or output data set or model, it only shows one input/output channel pair. The selected channel names are displayed in the title bar of the plot window.

Select a different input/output channel pair from the **Channel** menu in 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.

The **Channel** menu uses the following notation to list channels: `u1->y2` means that the transfer function from input channel `u1` to output channel `y2` is displayed.

The symbol `e` represents a noise source and might appear as `e@y3->y1`, which means that the transfer function from the noise channel (associated with `y3`) to output channel `y2` is displayed. System identification estimates as many noise sources as there are output channels. In general, `e@ynam` indicates that the noise source (innovation) corresponds to the output with name `ynam`. For more information about noise channels, see "Subreferencing Measured and Noise Models" on page 1-38.

Tip When you import data into the System Identification Tool, it is helpful to assign meaningful channel names in the Import Data dialog box. For more information about importing data, see "Creating Data Sets in the System Identification Tool" on page 3-13.

Grid, Line Styles, and Redrawing Plots

Although different plot types contain different options in the **Style** menu that are specific to each plot type, there are several **Style** options that are common to all plot types. These include the following:

- "Grid Lines" on page 2-34
- "Solid or Dashed Lines" on page 2-34
- "Plot Redrawing" on page 2-34

Grid Lines

To toggle showing or hiding grid lines, select **Style > Grid**.

Solid or Dashed Lines

Select **Style > Separate linestyles** to display currently-visible lines as a combination of solid, dashed, dotted, and dash-dotted line styles.

Select **Style > All Solid Lines** to display all solid lines. This choice is the default.

All line styles retain the color of their corresponding data or model icon in the System Identification Tool window.

Plot Redrawing

To avoid redrawing the entire plot when you add another data set or model to the plot, select **Style > Erasemode xor**. Although this selection results in faster response, it might also produce poor plot quality.

To specify that the plot be redrawn when you add another data set or model to the plot, select **Style > Erasemode normal**. This choice is the default setting.

Opening a Plot in a MATLAB Figure Window

The MATLAB Figure window contains general plot editing and printing controls that are not available in the System Identification Toolbox plot window. Sometimes it is useful to create a plot in the System Identification Tool, and then open it in a MATLAB Figure window for fine-tuning its appearance.

After you create the plot, as described in "How to Create a Plot" on page 2-29, select **File > Copy figure** in the plot window. This command opens the plot in a MATLAB Figure window, but also leaves the System Identification Toolbox plot window open.

Printing Plots

To print a System Identification Toolbox plot to a printer, select **File > Print** in the plot window. In the Print dialog box, select the printing options and click **OK**.

Representing Data for System Identification

Introduction to Representing Data

Before you can estimate models in System Identification Toolbox, you need to import your data into the MATLAB workspace. You can import the data from external data files, create data by simulation, or manually create data arrays in the MATLAB Command Window. For more information about requirements on MATLAB variables, see "Data Requirements" on page 3-5. To learn how to create data by simulation, see "Creating Data Using Simulation" on page 3-71.

After your data is in the MATLAB workspace, you must represent your data in System Identification Toolbox format. If you prefer using the graphical user interface (GUI) environment, import data from the MATLAB workspace into the System Identification Tool (see "Creating Data Sets in the System Identification Tool" on page 3-13).

If you primarily work in the MATLAB Command Window, then represent the data using custom data structures, called *data objects*, to encapsulate the data values and properties. Data objects provide the convenience of manipulating data and its properties as a single entity.

System Identification Toolbox provides the following two types of data objects:

- `iddata` — For time-domain or frequency-domain data (see "Creating iddata Objects" on page 3-31).
- `idfrd` — For frequency-response data (see "Creating idfrd Objects" on page 3-51).

Note Importing data into the System Identification Tool creates the corresponding data objects in the background.

System Identification Toolbox supports the following types of data:

- Time-domain data.
- Frequency-domain data, which is a Fourier transform of time-domain signals.

- Frequency-response data, which can be measured using a spectrum analyzer.

You can use System Identification Toolbox to process and model single-variable and multivariable data, as follows:

- Single-input and single-output (SISO) systems.
- Multiple-input and single-output (MISO) systems.
- Single-input and multiple-output (SIMO) systems.
- Multiple-input and multiple-output (MIMO) systems.
- Time-series data that has no inputs and one or more outputs.

When several time-domain or frequency-domain data sets are measured for a dynamic system, you might choose to combine these data sets into one `iddata` object and create one multiexperiment data set. Estimating models for multiexperiment data produces an average model. To learn how to do such estimates in the System Identification Tool, see “Working with Multiexperiment Data” on page 3-21. To learn how to create multiexperiment data using functions, see “Creating Multiexperiment Data Sets” on page 3-37.

Data Requirements

This section describes what you need to know before representing time-domain, frequency-domain, and frequency-response data in System Identification Toolbox format and contains the following topics:

- "Importing Data into MATLAB" on page 3-5
- "Requirements for Time-Domain Data" on page 3-6
- "Requirements for Time-Series Data" on page 3-7
- "Requirements for Frequency-Domain Data" on page 3-7
- "Requirements for Frequency-Response Data" on page 3-9

The *sampling interval* is the time between successive data samples. System Identification Toolbox interface provides limited support for nonuniformly sampled data. For more information about specifying uniform and nonuniform time vectors, see "Constructing iddata for Time-Domain Data" on page 3-32.

Note The System Identification Tool GUI only supports uniformly sampled data.

After verifying that your data meets the requirements described in this section, you can import data into the System Identification Tool. For more information, see "Creating Data Sets in the System Identification Tool" on page 3-13.

If you primarily work in the MATLAB Command Window, see "Creating iddata Objects" on page 3-31 (for time-domain and frequency-domain data) or "Creating idfrd Objects" on page 3-51 (for frequency-response data).

Importing Data into MATLAB

If you are modeling data from an external data file, you must import this data into the MATLAB workspace. The MATLAB Programming documentation provides information about supported data formats and import functions.

The simplest way to import data into MATLAB is to use the MATLAB Import Wizard, which supports the following types of data formats:

- MAT-files containing MATLAB arrays or System Identification Toolbox `iddata` and `idfrd` data structures
- Text files, such as `.txt` and `.dat`
- Spreadsheet files, such as `.xls`
- Graphics files, such as `.gif` and `.jpg`
- Audio and video files, such as `.avi` and `.wav`

For information about using the MATLAB Import Wizard, see the MATLAB Programming documentation.

Requirements for Time-Domain Data

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 output data, see "Requirements for Time-Series Data" on page 3-7.

The following variables must exist in the MATLAB workspace before you can represent time-domain data in System Identification Toolbox:

- Input data.

  For single-input and single-output (SISO) data, the input must be a columnwise vector.

  For a data set with N_u inputs and N_T samples (measurements), the input is an N_T-by-N_u matrix.
- Output data.

  For single-input and single-output (SISO) data, the output must be a columnwise vector.

  For a data set with N_y outputs and N_T samples (measurements), the output is an N_T-by-N_y matrix.

- Sampling time interval.

If you are working with uniformly-sampled data, use the actual sampling interval n 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 in the MATLAB Command Window, you can specify a vector of time instants using the `iddata` `TimeInstants` property, as described in "Constructing iddata for Time-Domain Data" on page 3-32.

Requirements for Time-Series Data

A special case of time-domain data is time-series data, which consist of one or more outputs *y(t)* with no corresponding input.

The following variables must exist in the MATLAB workspace before you can represent time-series data in System Identification Toolbox:

- Output data.
  - For single-input and single-output (SISO) data, the output must be a columnwise vector.
  - For a data set with N_y outputs and N_T samples (measurements), the output is an N_T-by-N_y matrix.
- Sampling time interval.
  - If you are working with uniformly-sampled data, use the actual sampling interval in 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 in the MATLAB Command Window, you can specify a vector of time instants using the `iddata` `TimeInstants` property, as described in "Constructing iddata for Time-Domain Data" on page 3-32.

For information on estimating time-series models, see "Time-Series Models" on page 5-94.

Requirements for Frequency-Domain Data

This section defines continuous and discrete Fourier transforms, and specifies the required MATLAB variables and their dimensions for representing frequency-domain data in System Identification Toolbox. It discusses the following topics:

- "About Frequency-Domain Data" on page 3-8
- "Representing Frequency-Domain Data" on page 3-9

About Frequency-Domain Data

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:

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

$$U(iw) = \frac{1}{2\pi}\int_{-\infty}^{\infty} u(t)e^{-iwt}dt$$

In the context of numerical computations, continuous equations must be 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.

When the frequencies are not equally spaced, it is useful to also discretize the frequencies in the Fourier transform. The resulting Discrete-Fourier Transform (DFT) of time-domain data 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 values.

Representing Frequency-Domain Data

The following variables must exist in the MATLAB workspace before you can represent frequency-domain data in System Identification Toolbox:

- Input data.
  - For single-input and single-output (SISO) data, the input must be a columnwise vector.
  - For a data set with N_u inputs and N_f frequencies, the input is an N_f-by-N_u matrix.
- Output data.
  - For single-input and single-output (SISO) data, the output must be a columnwise vector.
  - For a data set with N_y outputs and N_f frequencies, the output is an N_f-by-N_y matrix.
- Frequency values.

  Must be a columnwise vector.

Requirements for Frequency-Response Data

Before you can represent frequency-response data in the System Identification Toolbox format, this data must exist in the MATLAB workspace.

This section discusses the following topics:

- "About Frequency-Response Data" on page 3-10
- "Representing Frequency-Response Data" on page 3-11

About 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. You can measure frequency-response data values directly using a spectrum analyzer, for example, which provides a compact representation of the input and the output (compared to storing input and output independently).

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 $^{2\pi}/_{T}$.

For a sinusoidal input to the system, the output is also a sinusoid with the same frequency. The frequency-response data magnifies the amplitude of the input by $|G|$ and shifts its phase by $\varphi = \arg G$. Because the frequency function is evaluated at the sinusoid frequency, the values of the frequency function at a specific frequency describe the response of the linear system to an input at that frequency.

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 of values, to study the system frequency response. However, you cannot use this model for simulation and prediction and must create a parametric model from the frequency-response data.

Representing Frequency-Response Data

There are two ways to represent frequency-response data in System Identification Toolbox. The first approach lets you manipulate the data using both System Identification Tool and System Identification Toolbox functions in the MATLAB Command Window, and the second approach is only used for working with data in the System Identification Tool.

The following variables must exist in the MATLAB workspace before you can represent frequency-response data in System Identification Toolbox:

- In System Identification Tool or MATLAB Command Window, represent complex-valued $G(e^{iw})$

  For single-input single-output (SISO) systems, the frequency function is a columnwise vector.

  For a data set with N_u inputs, N_y outputs, and N_f frequencies, the frequency function is an N_y-by-N_u-by-N_f array.

- In System Identification Tool only, represent amplitude $|G|$ and phase shift $\varphi = \arg G$

  For single-input single-output (SISO) systems, the amplitude and the phase must each be a columnwise vector.

For a data set with N_u inputs, N_y outputs, and N_f frequencies, the amplitude and the phase must each be an N_y-by-N_u-by-N_f array.

- Frequency values must be a columnwise vector.

Creating Data Sets in the System Identification Tool

There are several ways to create new data sets in the System Identification Tool. For example, you can import data into the System Identification Tool window and combine selected subsets of input and output channels to create a new data set. You can also merge several data sets into a single multiexperiment data set, or extract selected experiments from a multiexperiment data set into a single data set.

Note Data preprocessing operations also create new data sets. See Chapter 4, “Plotting and Preprocessing Data”.

This section discusses the following topics:

- “Importing Data into the System Identification Tool” on page 3-13
- “Specifying the Sampling Interval” on page 3-18
- “Creating Data Sets from Selected Channels” on page 3-19
- “Working with Multiexperiment Data” on page 3-21
- “Renaming Data and Changing Its Display Color” on page 3-29

For information about working with System Identification Tool, see Chapter 2, “Working with the System Identification Tool GUI”.

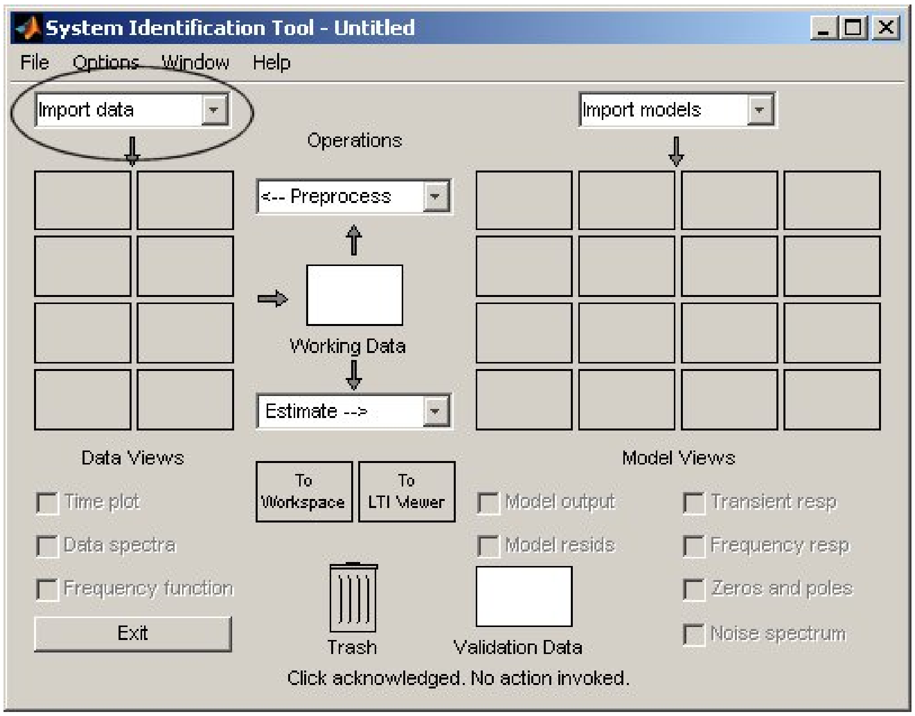
Importing Data into the System Identification Tool

To open the GUI, type the following command at the MATLAB prompt:

```
ident
```

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

Note Your data must be sampled at equal time intervals.



The following table summarizes the commands for importing different types of data into the System Identification Tool. Each command opens the Import Data dialog box. For detailed information about the fields in the Import Data dialog box, click **Help**.

This table also specifies which MATLAB variables that must be present in the MATLAB workspace. For information about the dimensions of the required MATLAB variables, see "Data Requirements" on page 3-5.

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

Commands for Importing Data

Type of Data	Import Command	Required Information
Time-domain data sampled as a function of time.	Select **Import data > Time domain data** to open the Import Data dialog box. **Time-Domain Signals** is already selected in the **Data Format for Signals** list.	Specify the following to store data as an `iddata` object: • Input signal as MATLAB vector, matrix, or expression. • Output signal as MATLAB vector, matrix, or expression. • Sampling time interval. See "Specifying the Sampling Interval" on page 3-18. **Note** For time series, only import the output signal and enter `[]` for the input.

Commands for Importing Data (Continued)

Type of Data	Import Command	Required Information
Frequency-domain data consists of Fourier transforms of time-domain data (a function of frequency).	Select **Import data > Freq. domain data** to open the Import Data dialog box. **Frequency Domain Signals** is already selected in the **Data Format for Signals** list.	Specify the following to store as an `iddata` object: • Input signal as MATLAB vector, matrix, or expression. • Output signal as MATLAB vector, matrix, or expression. • Frequency vector or MATLAB expression. • Sampling time interval. See "Specifying the Sampling Interval" on page 3-18.

Commands for Importing Data (Continued)

Type of Data	Import Command	Required Information
Frequency-response data consists of complex-valued frequency responses at specified frequencies.	Select **Import data > Freq. domain data** to open the Import Data dialog box. In the **Data Format for Signals** list, select **Freq. Function (complex)**.	Specify the following to store data as an `idfrd` object: • MATLAB variable or expression representing the complex frequency-response data $G(e^{iw})$. • Frequency MATLAB variable or expression representing a vector of frequency values.
Frequency-response data consists of frequency-response amplitude and phase values at specified frequencies.	Select **Import data > Freq. domain data** to open the Import Data dialog box. In the **Data Format for Signals** list, select **Freq. Function (Amp/Phase)**.	Specify the following to store data as an `idfrd` object: • Amplitude variable or MATLAB expression representing $\lvert G \rvert$. • Phase variable or MATLAB expression representing $\varphi = \arg G$. • Frequency variable or MATLAB expression representing a vector of frequency values.

Commands for Importing Data (Continued)

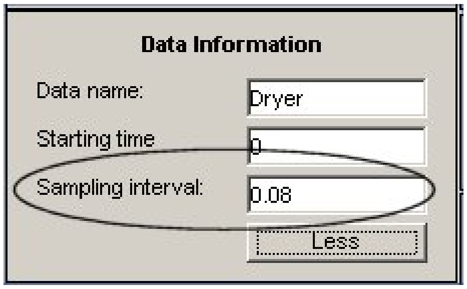
Type of Data	Import Command	Required Information
System Identification Toolbox data object (`iddata`, `idfrd`) or Control System Toolbox `frd` object.	Select **Import data > Data object** to open the Import Data dialog box. **IDDATA or IDFRD/FRD** is already selected in the **Data Format for Signals** list.	Import `iddata`, `idfrd`, or `frd` data object in the MATLAB workspace. **Note** Importing an `frd` object from Control System Toolbox converts it to an `idfrd` object.
Example data from `dryer2.mat`, consisting of 1000 single-input and single output (SISO) power and temperature values, respectively.	Select **Import data > Example** to open the Import Data dialog box with the sample data information already entered in the dialog box.	Import sample data and store as an `iddata` object.

Specifying the Sampling Interval

When you specify the input and output data in the Import Data dialog box, you must also enter the sampling interval in the units of your experiment. The *sampling interval* is the time between successive data samples in your experiment and must be the actual time interval at which your data is sampled in any units. For example, enter `0.5` if your data was sampled every `0.5 ms`, and enter `1` is 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 `fft`), the Fourier transforms are computed as Discrete Fourier Transforms (DFT) for this sampling interval. In addition, the sampling instants are used to set the horizontal axis on time plots.



Data Information

Data name: Dryer

Starting time 0

Sampling interval: 0.08

Sampling Interval in the Import Data Dialog Box

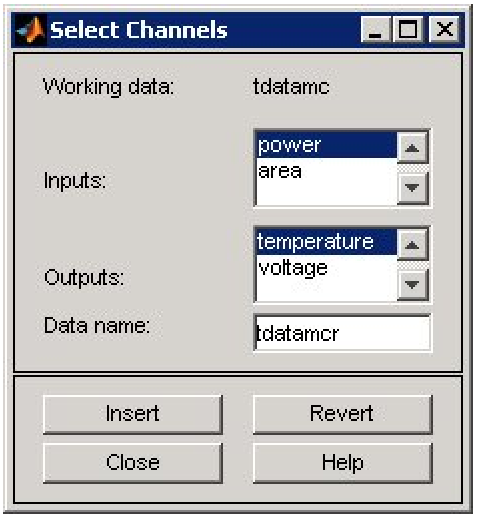
Creating Data Sets from Selected Channels

You can create a new data set in the System Identification Tool 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 window, 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 window.
7. Click **Close**.

Working with Multiexperiment Data

You can create a time-domain or frequency-domain data set in the System Identification Tool that consists of several experiments. *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 data segments. Identifying models for multiexperiment data results in an *average* model.

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 type (that is, time-domain data or frequency-domain data only).

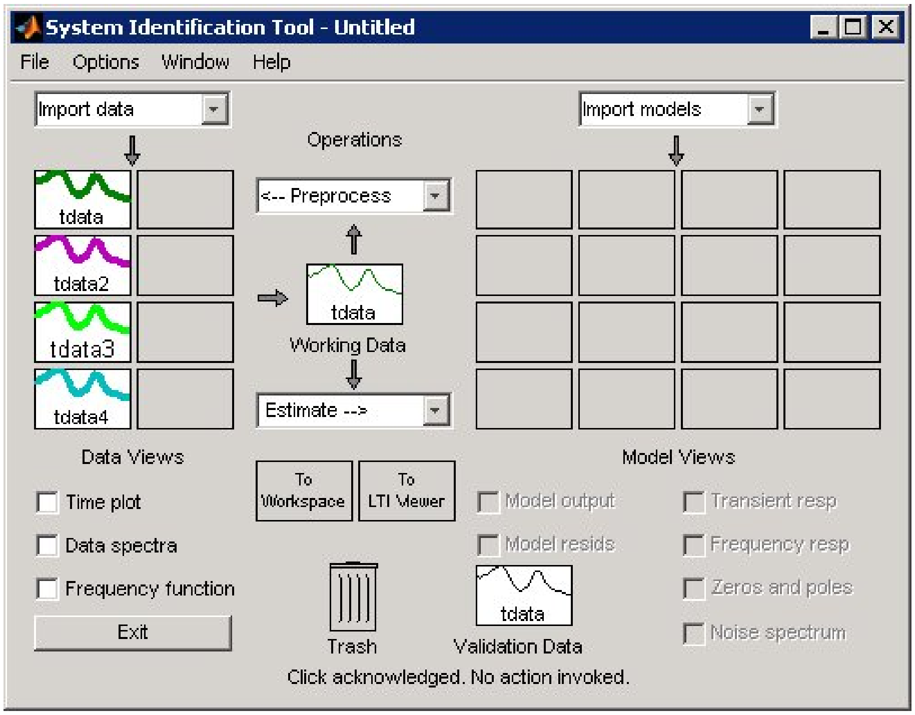
This section discusses the following topics:

Merging Data Sets

This section describes how to merge data sets in the System Identification Tool.

Note Before merging several segments of the same data set, verify that the time vector of each data starts at the time when that data segment was actually measured (relative to the other data sets).

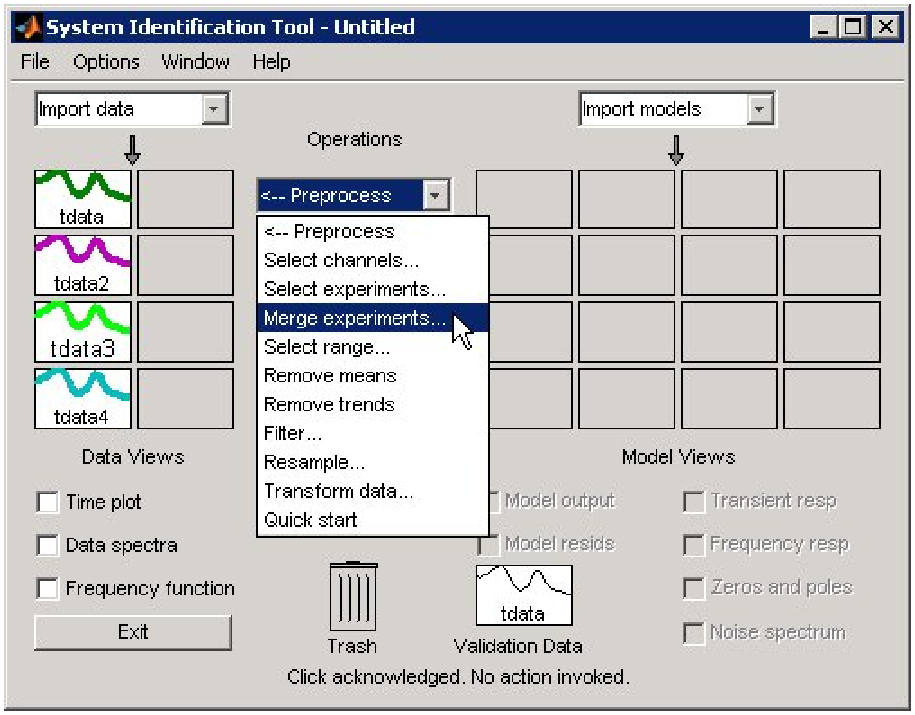
For example, suppose that you want to combine the data sets `tdata`, `tdata2`, `tdata3`, `tdata4` shown in the following figure.



GUI Contains Four Data Sets to Merge

The following procedure describes how to merge data sets.

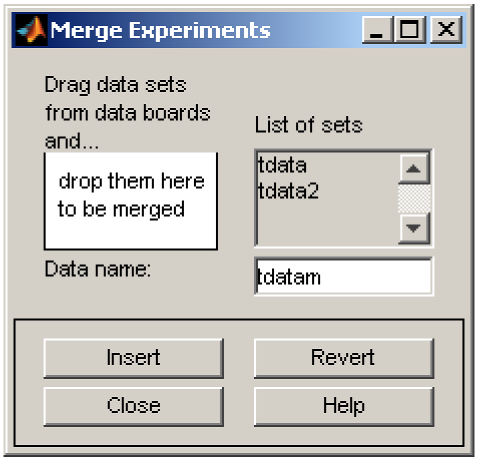
1 In the Operations area, select **Preprocess > Merge experiments** from the drop-down menu to open the Merge Experiments dialog box.



Click acknowledged. No action invoked.

2 Drag a data set from the Data Board in System Identification Toolbox window to the Merge Experiments dialog box, into the **drop them here to be merged** rectangle.

The name of the data set is added to the **List of sets**.


Merge Experiments
Drag data sets from data boards and...
drop them here to be merged
List of sets
Data name:
Insert
Revert
Close
Help

List of sets: tdata, tdata2

Data name: tdatam

tdata and tdata2 to Be Merged

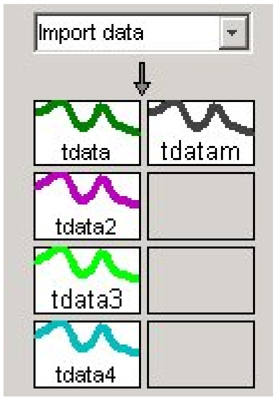
Tip To empty the list, click **Revert**.

---

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.

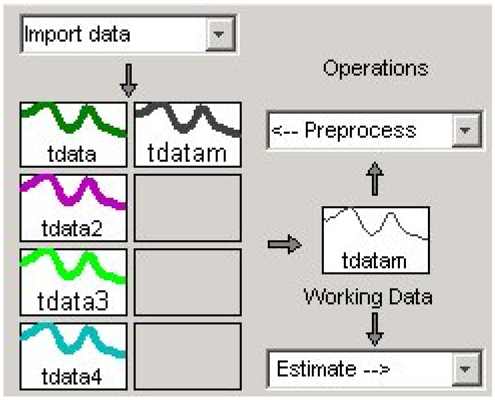
Tip To get information about the state of creation of any data set in the System Identification Tool, right-click the data icon to open the Data/model Info dialog box.

Extracting Experiments 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.

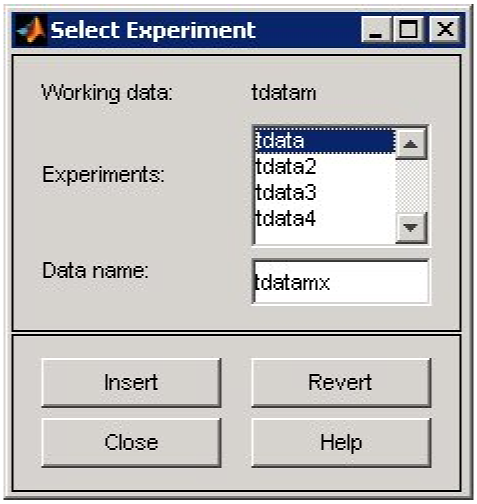
For example, suppose that `tdatam` consists of four experiments. To create a new data set that includes only the first and third experiments in this data set, perform the following procedure:

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 Experiments dialog box.



Working data: tdatam

Experiments: tdata, tdata2, tdata3, tdata4

Data name: tdatamx

3 In the **Experiments** list, select one or more data sets in one 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 window.

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

Renaming Data and Changing Its Display Color

You can rename data and change its display color by double-clicking the data icon in the System Identification Tool window.

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.

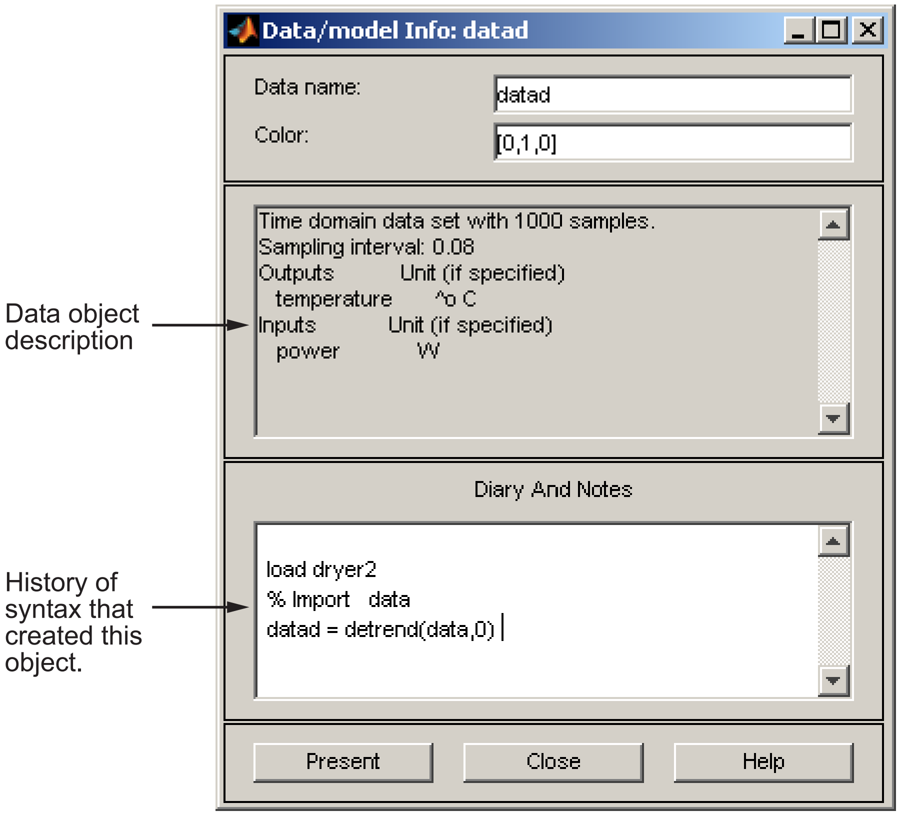
The Data/model Info dialog box also lets you rename the data by entering a new name in the **Data name** field.

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 System Identification Tool" on page 2-15.

Tip As an alternative to using three RGB values, you can enter any *one* of the following letters in single quotes:

```
'y' 'r' 'b' 'c' 'g' 'm' 'k'
```

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



Data name: datad

Color: [0,1,0]

```
Time domain data set with 1000 samples.
Sampling interval: 0.08
Outputs          Unit (if specified)
  temperature      ^o C
Inputs           Unit (if specified)
  power            W
```

```
load dryer2
% Import   data
datad = detrend(data,0)
```

Information About the Data

Finally, 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 the portions of this information in the MATLAB Command Window.

Creating iddata Objects

This section describes how to represent the values and properties of time- and frequency-domain data using `iddata` objects.

To represent data in the System Identification Tool GUI instead, see "Creating Data Sets in the System Identification Tool" on page 3-13. For more information about working with frequency-response data, see "Creating idfrd Objects" on page 3-51.

This section discusses the following topics:

- "iddata Constructor" on page 3-31
- "iddata Properties" on page 3-34
- "Creating Multiexperiment Data Sets" on page 3-37
- "Subreferencing iddata Objects" on page 3-39
- "Modifying Time and Frequency Vectors" on page 3-43
- "Naming, Adding, and Removing Input and Output Channels" on page 3-47
- "Concatenating iddata Objects" on page 3-49

iddata Constructor

The `iddata` object represents time-domain or frequency-domain data. To construct an `iddata` object, you must have the input and output data variables already in the MATLAB workspace. Before you begin, check the requirements on data dimensions in "Data Requirements" on page 3-5.

For details on how to construct `iddata` objects, see the following topics:

- "Constructing iddata for Time-Domain Data" on page 3-32
- "Constructing iddata for Frequency-Domain Data" on page 3-33

Note You can create multiexperiment `iddata` objects by combining existing `iddata` objects. For more information, see "Creating Multiexperiment Data Sets" on page 3-37.

Constructing iddata for Time-Domain Data

Use the following syntax to create a time-domain object `data`:

```
data = iddata(y,u,Ts)
```

You can specify additional properties when you create the `iddata` object using the constructor syntax:

```
data = iddata(y,u,Ts,'Property1',Value1,...,'PropertyN',ValueN)
```

For more information about accessing object properties, see “iddata Properties” on page 3-34.

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 3-43.

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 and 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 output channel. For more information about naming channels, see "Naming, Adding, and Removing Input and Output Channels" on page 3-47.

Constructing iddata for Frequency-Domain Data

Frequency-domain data is the Fourier transform of the input and output signals at specific frequency values. To store 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 value w, respectively. Ts is the time interval between successive data samples in seconds, and w is the frequency column vector that defines the frequencies at which System Identification Toolbox calculates the Fourier transform values of y and u. w, y, and u have the same number of rows.

Note You must specify the frequency vector.

For more information about `iddata` time and frequency properties, see “Modifying Time and Frequency Vectors” on page 3-43.

To specify a continuous-time system, set `Ts` to `0`.

You can specify additional properties when you create the `iddata` object using the constructor syntax:

```
data = iddata(y,u,Ts,'Property1',Value1,...,'PropertyN',ValueN)
```

For more information about accessing object properties, see “iddata Properties” on page 3-34.

iddata Properties

To view the properties of the `iddata` object, use the `get` command. For example, type the following commands at the MATLAB 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 MATLAB 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` function or dot notation. For example, to change the sampling interval to `0.05`, type the following at the MATLAB 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 portion of the name you do enter uniquely identifies 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 MATLAB 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 Sets

You can create `iddata` objects that contain several experiments in the MATLAB Command Window. In the context of System Identification Toolbox, *experiments* can 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.

Identifying models for an `iddata` object with multiple experiments results in an *average* model that takes into account all data sets stored in the object.

Note The `idfrd` object does not support the `iddata` equivalent of multiexperiment data.

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

- The same number of input and output channels.
- The same input and output channel names.
- The same type (that is, time-domain data or frequency-domain data)

This section discusses the following topics:

- "Merging Data Sets" on page 3-38
- "Adding Experiments to an Existing iddata Object" on page 3-38

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
```

This merge results in an `iddata` object with 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, as described in "Merging Data Sets" on page 3-22.

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 `ExperimentName` property of the `iddata` object stores experiment names.

For more information about subreferencing experiments in a multiexperiment data set, see "Subreferencing Experiments" on page 3-42.

Subreferencing iddata Objects

Subreferencing data and its properties lets you select data and assign new data and property values.

This sections describes the syntax for subreferencing iddata objects and contains the following sections:

- "Subreferencing Input and Output Data" on page 3-39
- "Subreferencing Data Channels" on page 3-40
- "Subreferencing Experiments" on page 3-42

Subreferencing Input and Output Data

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 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 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.

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 retrieve all data samples in `y3`, `u1`, and `u4`, type the following command at the MATLAB prompt:

```
% Uses a cell array to reference
% input channels 'u1' and 'u4'
data_sub = data(:,'y3',{'u1','u4'})
```

or equivalently

```
% Uses 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 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 ten 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 first ten values of input channel `1` of `data` to values in samples `101` through `110` in the input channel `3` of `data1`.

```
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)
```

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 iddata for Time-Domain Data" on page 3-32.

The following two tables summarize these time-vector and frequency-vector properties and gives an example of setting each value. In each of the syntax examples, `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

Name of Time-Vector 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)

Name of Time-Vector 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` uniquely identifies the `SamplingInstants` property.
`TimeUnit`	Unit of time.	To change the unit of the time vector to `msec`: `data.ti = 'msec'`

iddata Frequency-Vector Properties

Name of Frequency-Vector Property	Description	Syntax Example

`Frequency`	Frequency values at which the Fourier transforms of the signals are defined.
	
	• For a single experiment, `Frequency` is a scalar value.
	• For multiexperiement data with `Ne` experiments, `Frequency` is a `1`-by-`Ne` cell array, and each cell contains the frequencies of the corresponding experiment.
		To specify `100` frequency values in log space, ranging between `0.1` and `100`, use the following syntax:
		
		`data.freq =`
		`logspace(-1,2,100)`

`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 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*. This section describes how to perform the following operations on `iddata` channels:

- "Naming Channels" on page 3-47
- "Adding Channels" on page 3-48
- "Modifying Channel Data" on page 3-48

Naming Channels

The `iddata` properties `InputName` and `OutputName` store one or more 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, you should 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 2-33.

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, System Identification 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 channels to input and output data in 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 in the MATLAB vector Input5, use the following syntax:

```
data.u(:,5) = Input5;
```

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 "Concatenating iddata Objects" on page 3-49.

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 MATLAB prompt:

```
data = data(:,3,[1 4])
```

The resulting data object contains one output channel and two input channels.

Concatenating iddata Objects

This section describes horizontal and vertical concatenation of iddata objects. For this discussion, it is useful to know that the InputData iddata property stores columnwise input data, and the OutputData property stores columnwise output data. For more information about accessing iddata properties, see "iddata Properties" on page 3-34.

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 inputs and more outputs. 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 equal 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.

Creating idfrd Objects

This section describes how to construct `idfrd` objects to represent the values and properties of frequency-response data.

You can measure frequency-response data values directly using a spectrum analyzer, for example, which provides a compact representation of the relationship between input and output (compared to storing input and output independently).

Frequency-response values also represent a (nonparametric) model of the relationship between the input and the output data. You might use such a model, which consists of a table of values, to examine the system frequency response. However, you cannot use this model for simulation and prediction and must create a parametric model from the frequency-response data.

For more information about estimating linear models, see Chapter 5, "Estimating Linear Nonparametric and Parametric Models". To learn more about estimating nonlinear models, see Chapter 6, "Estimating Nonlinear Black-Box Models".

For more information about frequency-response data, see "Requirements for Frequency-Response Data" on page 3-9.

This section discusses the following topics:

- "idfrd Constructor" on page 3-52
- "idfrd Properties" on page 3-53
- "Subreferencing idfrd Objects" on page 3-55
- "Concatenating idfrd Objects" on page 3-56

There are also other System Identification Toolbox operations that create `idfrd` objects, including the following:

- Transforming `iddata` objects. For more information, see "Transforming Between Frequency-Domain and Frequency-Response Data" on page 3-68.
- Estimating nonparametric models using `etfe`, `spa`, and `spafdr`. For more information, see "Spectral Analysis Models" on page 5-31.

- Converting the Control Systems Toolbox `frd` object. For more information, see "Using Models with Control System Toolbox" on page 10-20.

idfrd Constructor

The `idfrd` represents complex frequency-response data.

Before you can create an `idfrd` object, see "Requirements for Frequency-Response Data" on page 3-9.

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 in the MATLAB Command Window, 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 seconds 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, 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 MATLAB prompt.

To change property values for an existing `idfrd` object, use the `set` function or dot notation. For example, to change the name of the `idfrd` object, type the following command sequence at the MATLAB 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, but only as much of the name as uniquely identify the `idfrd` property.

If you import `fr_data` into the System Identification Tool, this data has the name `DC_Converter` in the GUI, and not the MATLAB 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: []
```

Subreferencing 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)
```

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.

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 "idfrd Properties" on page 3-53).

This section discusses the following topics:

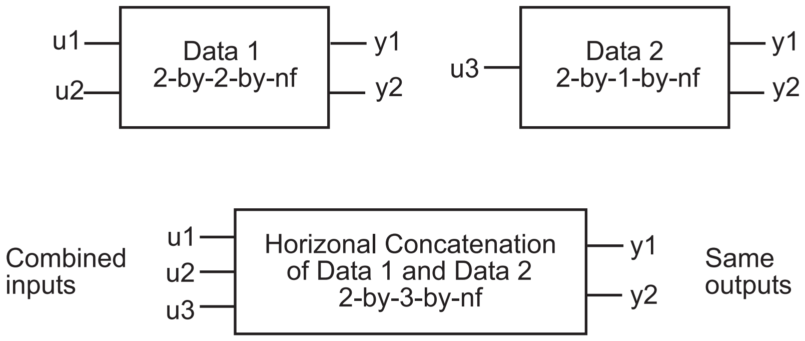
- "Horizontal Concatenation of idfrd Objects" on page 3-56
- "Vertical Concatenation of idfrd Objects" on page 3-57
- "Concatenating Noise Spectral Data of idfrd Objects" on page 3-58

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.



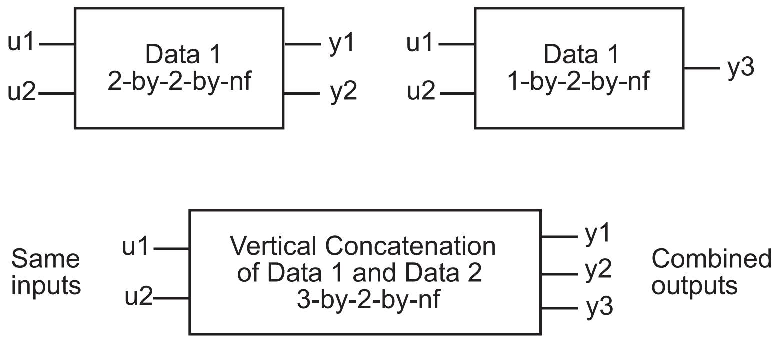
Note Horizontal concatenation of `idfrd` objects requires that they have the same outputs and frequency vectors. If the output channel names are different and their dimensions are the same, the concatenation operation uses the names of output channels in the first `idfrd` object. Input channels must 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 Spectral 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 spectral data does not exist (unless you also entered it manually), `SpectralData` 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 `SpectralData` property is not empty and contains the power spectra and cross spectra of the output noise in the system. For each output channel, System Identification Toolbox estimates one noise channel to explain the difference between the output of the model and the measured output.

When the `SpectralData` property of individual `idfrd` objects is not empty, horizontal and vertical concatenation handle `SpectralData`, as follows.

In case of horizontal concatenation, there is no meaningful way to combine the `SpectralData` of individual `idfrd` objects and the resulting `SpectralData` 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, System Identification 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.

Transforming Between Time- and Frequency-Domain Data

You can transform between time-domain, frequency domain, and frequency-response data. For a description of each of these types of data, see "Data Requirements" on page 3-5. The Toolbox commands for estimation and simulation apply indiscriminately to time-domain, frequency-domain, and frequency-response data.

This section discusses the following topics:

- "Transforming Data in the System Identification Tool" on page 3-59
- "Functions for Transforming Data" on page 3-66

Transforming Data in the System Identification Tool

This section describes how to use System Identification Tool for:

- "Transforming Time-Domain Data" on page 3-59
- "Transforming Frequency-Domain Data" on page 3-63
- "Transforming Frequency-Response Data" on page 3-64

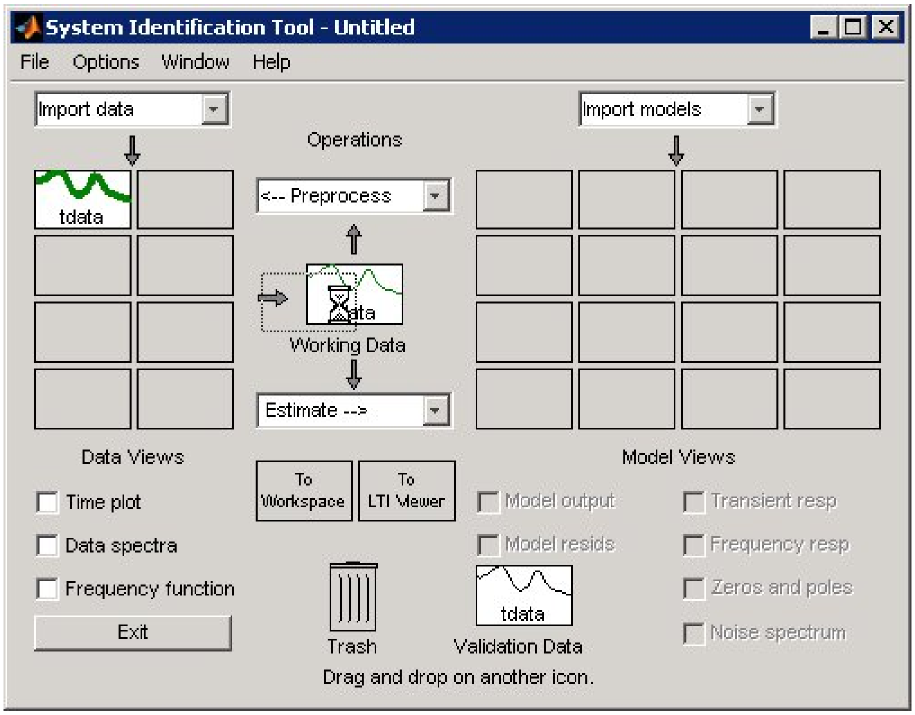
To learn how to transform data in the MATLAB Command Window instead of System Identification Tool GUI, see "Functions for Transforming Data" on page 3-66. For more information about working with System Identification Tool, see Chapter 2, "Working with the System Identification Tool GUI".

Transforming Time-Domain Data

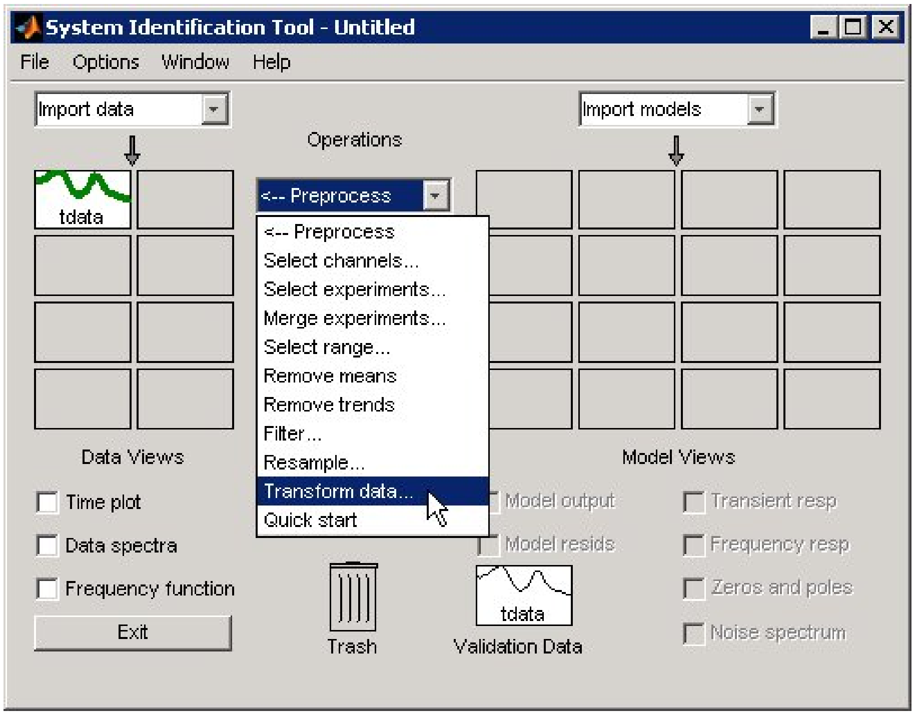
In the System Identification Tool window, time-domain data has an icon with a white background. You can transform time-domain data to frequency-domain or frequency-response data. The frequency values of the resulting frequency vector range from `0` to the Nyquist frequency $f_S = \pi / T_S$, where T_s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

Transforming from time-domain to frequency-response data is equivalent to creating a nonparametric model of the data using the `spafdr` method.

1 In the System Identification Tool window, drag the icon of the data you want to transform to the **Working Data** rectangle, as shown in the following figure.

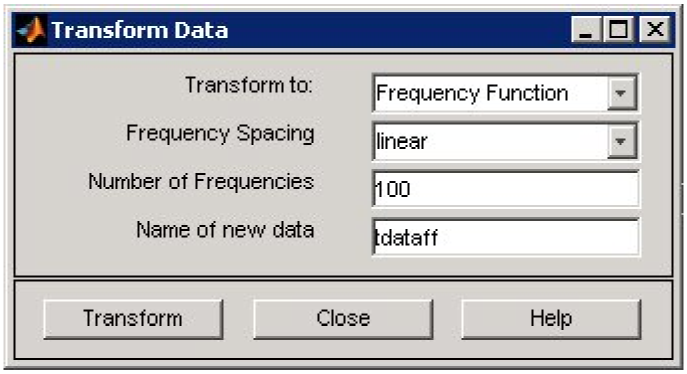


2 In the Operations area, select **Preprocess > Transform data** in the drop-down menu to open the Transform Data dialog box.



3 In the **Transform to** drop-down list, select one of the following:

- `Frequency Domain Data` — Create a new `iddata` object using the `fft` method. Go to step 6.
- `Frequency Function` — Create a new `idfrd` object using the `spafdr` method. Go to step 4.



Transform to: Frequency Function
Frequency Spacing: linear
Number of Frequencies: 100
Name of new data: tdataff

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 window.

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

Transforming Frequency-Domain Data

In the System Identification Tool window, 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 creating a nonparametric model of the data using the `spafdr` method.

1. In the System Identification Tool window, 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:
   - `Time Domain Data` — Create a new `iddata` object using the `ifft` (inverse Fast-Fourier Transform) method. Go to step 6.
   - `Frequency Function` — Create a new `idfrd` object 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 window.

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

Transforming Frequency-Response Data

In the System Identification Tool window, 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 and single-output (SISO) frequency-response data to frequency-domain data, System Identification 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, System Identification 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 make the response from each input a separate experiment in the MATLAB Command Window, see “Transforming Between Frequency-Domain and Frequency-Response Data” on page 3-68.

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’s 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 lets you specify logarithmic frequency spacing, which results in a variable frequency resolution.

Note The `spafdr` function lets you lets you specify any variable frequency resolution.

---

1. In the System Identification Tool window, 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 window.
8. Click **Close** to close the Transform Data dialog box.

Functions for Transforming Data

This section summarizes the functions for transforming data between time-domain, frequency-domain, and frequency-domain data and includes the following topics:

- "Supported Data Transformations" on page 3-66
- "Transforming Between Time and Frequency Domain" on page 3-67
- "Transforming Between Frequency-Domain and Frequency-Response Data" on page 3-68

To learn how to transform data using the System Identification Tool instead, see "Transforming Data in the System Identification Tool" on page 3-59.

Transforming from time-domain or frequency-domain data to frequency-response data is equivalent to creating a nonparametric model of the data. This section does not detail the process of estimating frequency-response models. For more information about creating nonparametric spectral models, see the `etfe`, `spa`, and `spafdr` reference pages.

Supported Data Transformations

The following table uses a matrix to show the different ways you can transform the data of one format (row) to another format (column). If the transformation is supported for a given row and column combination, 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`.

Original Data Format	To Time Domain (`iddata` object)	To Frequency Domain (`iddata` object)	To Frequency Function (`idfrd` object)
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 that has the same ratio between output and input as the original `idfrd` object.	Yes. Calculates a frequency function with different 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-domain`iddata` object `f_data` to time-domain `iddata` object `t_data`, use:
		
		`t_data =`
		`  ifft(f_data)`

Transforming Between Frequency-Domain and Frequency-Response Data

The `idfrd` object represents complex frequency-response of the system at different frequencies. For a description of this type of data, see "Requirements for Frequency-Response Data" on page 3-9.

This section describes how to transform frequency-response data to frequency-domain data (`iddata` object). When you select to transform single-input and single-output (SISO) frequency-response data to frequency-domain data, System Identification 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 pages. You might use this advanced feature to increase the number of data points near the system's 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.

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, System Identification 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`.

Creating Data Using Simulation

System Identification Toolbox lets you generate input data and simulate output data using a model structure.

You can also simulate data using Simulink and Signal Processing Toolbox. Data simulated outside System Identification Toolbox must be in the MATLAB workspace to be available to System Identification Toolbox environment. For more information about simulating models using Simulink, see “Using Models with Simulink” on page 10-26.

This section discusses the following topics:

- “Commands for Generating and Simulating Data” on page 3-71
- “Example – Creating Data with Periodic Inputs” on page 3-72
- “Example – Using sim to Simulate Model Output” on page 3-73

Commands for Generating and Simulating Data

Simulating output data requires that you have a parametric model. For more information about commands for constructing models, see “Types of Model Objects” on page 1-21.

To generate input data, use `idinput` to construct a signal with the desired characteristics, such as a random Gaussian or binary signal or a sinusoid. `idinput` returns a matrix of input values.

The following table lists the functions you can use to simulate output data. For more information about these commands, see the references pages.

Functions for Generating and Simulating Data

Function	Description	Example
`iddata`	Constructs an `iddata` object with input channels only.	To construct input data `data`, use the following command: `data = iddata([ ],[u v])` `u` is the input data, and `v` is white noise.

`idinput`	Constructs a signal with the desired characteristics, such as a random Gaussian or binary signal or a sinusoid, and returns a matrix of input values.	`u = iddata([],...`
		`    idinput(400,'rbs',[0 0.3]));`

`sim`	Simulates response data based on existing linear or nonlinear parametric model in the MATLAB workspace.	To simulate the model output `y` for a given input, use the following command: `y = sim(m,data)` `m` is the model object name, and `data` is input data matrix or `iddata` object.

Example – Creating Data with Periodic Inputs

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

```
per_u = idinput([300 2 5])
```

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

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

You can use the periodic input to simulate the output, and the use `etfe` to compute the estimated response of the model.

```
% Construct polynomial model
m0 =idpoly([1 -1.5 0.7],[0 1 0.5]);
% Construct random binary input
u = idinput([10 1 150],'rbs');
% Construct input data and noise
u = iddata([],u,'Period',10);
e = iddata([],randn(1500,1));
% Simulate model output with noise
y = sim(m0,[u e])
% Estimate frequency response
g = etfe([y u])
% Generate Bode plot
bode(g,'x',m0)
```

For periodic input, `etfe` honors the period and computes the frequency response using an appropriate frequency grid. In this case, the Bode plot shows a good fit at the five excited frequencies.

Example – Using sim to Simulate Model Output

This example demonstrates how you can create input data and a model, and then use the data and the model to simulate output data. You create the ARMAX model and simulate output data with random binary input `u`.

1. Load the three-input and one-output sample data.

   ```
   load iddata8
   ```

2 Construct an ARMAX model, using the following commands:

```
A = [1 -1.2 0.7];
B(1,:) = [0 1 0.5 0.1]; % first input
B(2,:) = [0 1.5 -0.5 0]; % second input
B(3,:) = [0 -0.1 0.5 -0.1]; % third input
C = [1 0 0 0 0];
Ts = 1;
m = idpoly(A,B,C,'Ts',1);
```

In this example, the leading zeros in the `B` matrix indicate the input delay (`nk`), which is `1` for each input channel. The trailing zero in `B(2,:)` makes the number of coefficients equal for all channels.

3 Construct pseudorandom binary data for input to the simulation.

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

4 Simulate the model output.

```
sim(m,u)
```

5 Compare model output to measured data to see how well the models captures the underlying dynamics.

```
compare(z8,m)
```

Plotting and Preprocessing Data

Preparing Data for Identification

After representing data in System Identification Toolbox, as described in Chapter 3, "Representing Data for System Identification", you can that you plot the data to examine its features. For more information about plotting data, see "Plotting Data" on page 4-7.

You can also use the `advice` command to identify constant offsets and linear trends, delays, feedback, and signal excitation levels in the data. For more information, see "Getting Advice About Your Data" on page 4-5.

Note If your data is complex valued, see "Handling Complex-Valued Data" on page 4-42 for information about supported operations in System Identification Toolbox.

Review the plots or use the `advice` command to determine whether your data requires preprocessing to prepare it for system identification. The following issues indicate 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 4-16.
- Offsets and drifts in signal levels (low-frequency disturbances).

  See "Detrending Data" on page 4-20 for information about subtracting means and linear trends, and "Filtering the Data" on page 4-31 for information about filtering.
- High-frequency disturbances above the frequency interval of interest for the system dynamics.

  See "Resampling Data" on page 4-24 for information about decimating and interpolating values, and "Filtering the Data" on page 4-31 for information about filtering.
- Nonlinearities in the data.

  On a frequency-function plot, nonlinearities might be indicated by different responses at different levels, or as different responses to a step-up versus a step-down input. If you suspect nonlinearities and have physical insight into the relationship between the variables, try nonlinear transformation of the data to make the model linear in the new (transformed) variables.

To learn more about estimating nonlinear models, see Chapter 6, "Estimating Nonlinear Black-Box Models".

By examining the quality of the signals and the frequency ranges that appropriately capture the system dynamics, you can select portions of the data to model, as described in "Selecting Data" on page 4-39. You can split a single data set into two portions and use one portion for model estimation and the other portion for model validation.

As an alternative preprocessing shortcut, you can select **Preprocess > Quick start** to simultaneously perform the following four actions:

- Subtract the mean value from each channel.
- Split data into two halves.
- Specify the first half as estimation data for models (or **Working Data**).
- Specify the second half as **Validation Data**

Getting Advice About Your Data

This section describes how to use the `advice` command to get information about your time-domain or frequency-domain data. This command does not support frequency-response data.

Note If you are using the System Identification Tool, 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 to the MATLAB workspace, see “Exporting to the MATLAB Workspace” on page 2-27.

Suppose that `data` is an `iddata` object. `advice(data)` displays the following information about the data in the MATLAB Command Window. Ask yourself:

- Does it make sense to remove constant offsets and linear trends from the data? See also `detrend`.
- What are the excitation levels of the signals and how does this affects the model orders? See also `pexcit`.
- Is there an indication of output feedback in the data? See also `feedback`. When feedback is present in the system, only prediction-error methods work well for estimating closed-loop data.

To estimate the delay from the input to the output in the system (dead time) by using the data, use the `delayest` function. You need to know the delay when specifying a model structure for estimation.

For following example shows how to get information about your data. Consider data from a single-input and single-output system sampled at 0.08 second. Use these commands to load the data and estimate the delay in the system:

```
load dryer2              % Load the sample input
                         % and output data
data=iddata(y2,u2,0.08)  % Create iddata object
delayest(data)           % Estimate delay (dead time)
```

MATLAB responds with:

```
ans =

     3
```

Type the following command to see what kind of information you receive about this data set:

```
advice(data)              % Get advice about the data
```

The results of using this command also suggest your next actions.

Plotting Data

After representing data in System Identification Toolbox, as described in Chapter 3, "Representing Data for System Identification", you can plot the data to examine its features.

System Identification Toolbox supports the following data plots:

- Time plot — Shows data values as a function of time.
- 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. For time- and frequency-domain data, shows the empirical transfer function estimate (see `etfe`) .

This section describes how to create plots when working in the System Identification Tool or the MATLAB Command Window. Plots you create using the plot commands, such as `plot`, `bode`, and `ffplot`, are displayed in the standard MATLAB figure window, which provides extensive options for formatting, saving, printing, and exporting plots to a variety of file formats. For more information, see the MATLAB Graphics documentation.

The plots you create using the System Identification Tool provide different options that are specific to System Identification Toolbox, such as selecting specific channel pairs in a multivariate signals or converting frequency units between hertz and radians per second.

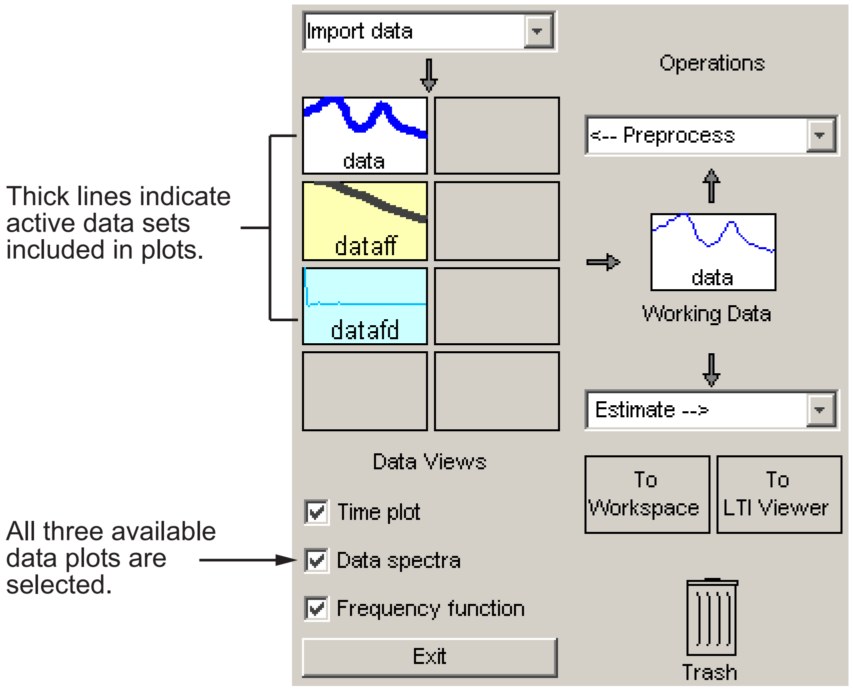
The rest of this section discusses the following topics:

- "Plotting Data in the System Identification Tool" on page 4-7
- "Functions for Plotting Data" on page 4-13

Plotting Data in the System Identification Tool

After importing data into the System Identification Tool, as described in "Creating Data Sets in the System Identification Tool" on page 3-13, 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 window.

Active data sets have a thick line in the icon. Only active data sets are displayed on the plots. To change whether to include or exclude a data set on a plot, click the data icon in the System Identification Tool window.



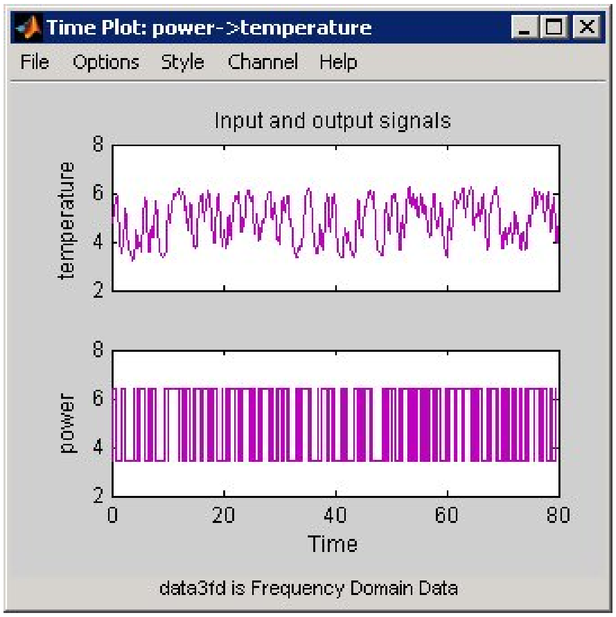
In this example, `data1` and `data3fd` are active and appear on the three selected plots.

The rest of this section discusses:

- "Working with a Time Plot" on page 4-9
- "Working with a Data Spectra Plot" on page 4-10
- "Working with a Frequency Function Plot" on page 4-12

Working with 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: power->temperature
File Options Style Channel Help
Input and output signals
temperature
8
6
4
2
power
8
6
4
2
0
20
40
60
80
Time
data3fd is Frequency Domain Data

Time Plot of data1

Note You can plot several data sets with the same input and output channel names. The plot displays data for all channels that have the same name. To view a different input-output channel pair, select it from the **Channel** menu. For more information about selecting different input and output pairs, see "Selecting Measured and Noise Channels in Plots" on page 2-33.

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 2-29.

Time Plot Options

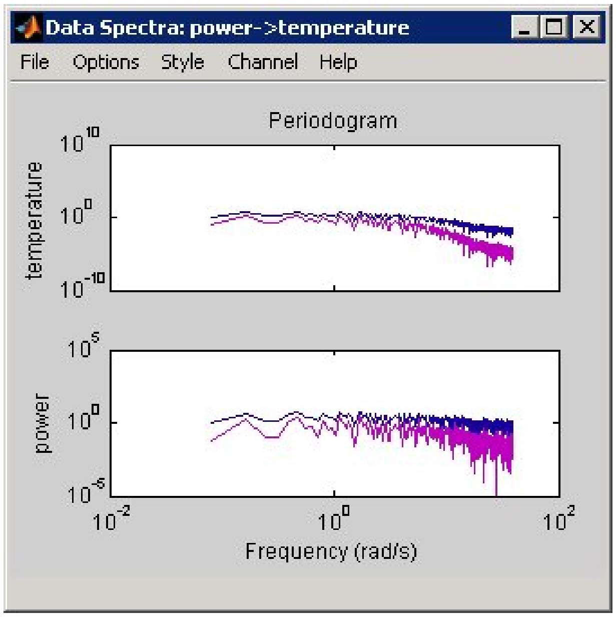
Action	Command
Toggle input display between piecewise continuous (zero-order hold) and linear interpolation (first-order hold) between samples. **Note** This option only affects the display and not the intersample behavior specified when importing the data.	Select **Style > Staircase input** for zero-order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

Working with a Data Spectra Plot

The **Data spectra** plot shows a periodogram or a spectral estimate of `data1` and `data3fd`.

The periodogram is 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. The spectral estimate for time-domain data is a smoothed spectrum calculated using `spa`. For frequency-domain data, the **Data spectra** plot shows the absolute value of the square of the actual data.

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

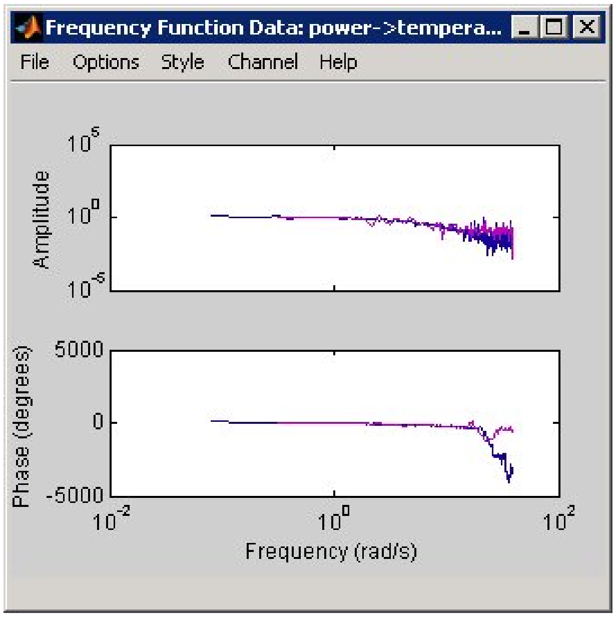
Data Spectra Plot Options

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**.

Working with 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 "Requirements for Frequency-Response Data" on page 3-9.



Frequency Functions of data1 and data3fd

Frequency Function Plot Options

Action	Command
Change frequency units.	Select **Style > Frequency (rad/s)** or **Style > Frequency (Hz)**.

Frequency Function Plot Options (Continued)

Action	Command
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**.

Functions for Plotting Data

The following table summarizes the functions available for plotting time-domain, frequency-domain, and frequency-response data:

Functions for Plotting Data

Function	Description	Example
`bode`	For frequency-response data only. Shows the magnitude and phase of the frequency response on a logarithmic frequency scale.	To plot `idfrd` data: `bode(idfrd_data)`

Functions for Plotting Data (Continued)

Function	Description	Example
`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`	Depending on the type of data, MATLAB generates the corresponding type of plot. For example, plotting time-domain data generates a time plot, and plotting frequency-response data generates a frequency-response plot. When plotting time- or frequency-domain inputs and outputs, the top axes show the output and the bottom axes show the input.	To plot `iddata` or `idfrd` data: `plot(data)` **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.

The following examples show use of the `plot` command.

To plot portions of the data, you can subreference specific samples (see "Subreferencing iddata Objects" on page 3-39 and "Subreferencing idfrd Objects" on page 3-55. 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 piecewise constant between sampling points on the plot. For more information about properties, see “iddata Properties” on page 3-34 or 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 3-40.

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.

Handling Missing Data and Outliers

Malfunctions in the data acquisition equipment can result in missing data. Malfunctions can also produce errors in measured values, called *outliers*.

When you import data that contains missing values into MATLAB 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.

This section describes how to handle these issues using functions and contains the following topics:

- "Working with Missing Data" on page 4-16
- "Working with Outliers" on page 4-17
- "Example – Extracting and Modeling Specific Data Segments" on page 4-18

Note This functionality is not supported in the System Identification Tool.

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.

Working with Missing Data

Data acquisition failures sometimes result in missing measurements both in the input and the output channels. When you plot data on a time-plot that contains missing values, gaps appear on the plot where missing data exists. For more information on creating plots, see "Plotting Data" on page 4-7.

You can use `misdata` to estimate missing values. This function 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 pages.

Note You can only use misdata on time-domain data stored in an iddata object. For more information about creating iddata objects, see "Creating iddata Objects" on page 3-31.

For example, suppose y and u are output and input signals that contain NaNs. 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 function to the new data object. For example:

```
dat1 = misdata(dat);
plot(dat,dat1)        % Check how the missing data
                      % was estimated on a time plot
```

Working with Outliers

Data acquisition failures can result in erroneous measurements. 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. This section describes ways you can identify and handle outliers in your data.

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 unreliable.
- After estimating a model, plot the residuals and identify unusually large values. Then, 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 4-18). For more information about selecting and extracting data segments, see "Selecting Data" on page 4-39.

  Note 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` function to reconstruct flagged data. This approach treats outliers as missing data and is described in "Working with Missing Data" on page 4-16. 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 the Data" on page 4-31.

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. Modeling multiexperiment data sets produces an average model for the different experiments.

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)
```

Detrending Data

Detrending data in System Identification Toolbox means removing the signal mean values and linear trends from the signals.

In System Identification Tool, you can subtract the mean values and one linear trend. If you are working in the MATLAB Command Window, the `detrend` function lets you subtract mean values and one or several linear trends connected at specified breakpoints. A *breakpoint* is a time value that defines the discontinuities between successive linear trends.

This section discusses the following topics:

- "Detrending Data for Nonlinear Versus Linear Models" on page 4-20
- "When to Subtract the Mean Values" on page 4-21
- "When to Subtract Linear Trends" on page 4-21
- "Detrending Data Using the System Identification Tool" on page 4-22
- "Using the detrend Function" on page 4-23

For a detailed discussion 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.

Detrending Data for Nonlinear Versus Linear Models

In general, you should detrend data before estimating linear models. For more information, see "When to Subtract the Mean Values" on page 4-21 and "When to Subtract Linear Trends" on page 4-21.

In general, you should not detrend data before estimating nonlinear models. In the case of nonlinear grey-box models, do not detrend the data to make sure that the models represent the actual physical levels.

For linear models, detrending is necessary because, theoretically, a zero mean in the input results in a zero mean in the output.

Nonlinear models are more general and can include the trend as part of the model. In this case, detrending is not necessary.

However, there are some cases when detrending data for nonlinear models might be helpful. For example, to improve computation accuracy, you might detrend data with a large constant trend.

Note Whatever trend you subtract from the estimation data, you must subtract the same trend from the validation data.

When to Subtract the Mean Values

You may find it useful to subtract mean values from your data when you have steady-state and not transient data. If you have steady-state data, estimating linear models for signals measured relative to an equilibrium is usually sufficient. Thus, you can find linearized models around a physical equilibrium and avoid modeling the absolute levels in physical units. You can use the `detrend` function or System Identification Tool commands to subtract mean levels from your signals.

Tip When you know the mean levels that correspond to the actual physical equilibrium, remove the equilibrium values instead of the mean value of the signals for best results.

Do not detrend your data when the physical levels are built into the underlying model or when input integrators in the system require absolute signal levels.

When you are working with transient data (such as step or impulse response), do not remove the mean from the data. With transient data, when the output at zero input is not zero, you might want to subtract the constant value corresponding to the time before the input is applied. Use core MATLAB functions to subtract a constant value from the data matrices.

When to Subtract Linear Trends

Often, the mean levels drifts during the experiment. You can eliminate drift by removing a linear trend or several piecewise linear trends from the signals.

You can use the `detrend` function or the System Identification Tool to subtract one linear trend from your signals from time-domain data. You must use the `detrend` function to remove several piecewise-linear trends in time-domain data or to remove the mean for frequency-domain data.

Signal drift is considered a low-frequency disturbance. If you know the drift rate, you can build a custom high-pass filter and apply it as described in "Filtering the Data" on page 4-31.

Detrending Data Using the System Identification Tool

If you are working in the System Identification Tool, you can detrend time-domain data by removing the mean value and linear trend from each channel. If you need to remove piecewise linear trends, see "Using the detrend Function" on page 4-23. See "Detrending Data for Nonlinear Versus Linear Models" on page 4-20 to decide whether it is appropriate to detrend your data.

For general information about working with System Identification Tool, see Chapter 2, "Working with the System Identification Tool GUI".

Note Select **Preprocess > Quick start** to remove the mean value from each channel, split data into two halves, specify the first half as estimation data for models (or **Working Data**), and specify the second half as **Validation Data**.

To detrend each input and output data channel:

1. Import time-domain data into the System Identification Tool, as described in "Creating Data Sets in the System Identification Tool" on page 3-13.

2. Drag the data set you want to detrend to the **Working Data** area.

3. Determine if you want to remove both the mean values and the linear trend from the data.

   - If yes, select **Preprocess > Remove trends**. This creates a new data set in the Data Board. You are finished.

   - If no, go to step 4.

4 To only remove constant offsets from the data, select **Preprocess > Remove means**. This selection creates a new data set in the Data Board.

Using the detrend Function

You can use the `detrend` command to remove the mean value and one or more linear trends from each channel of time-domain data. See "Detrending Data for Nonlinear Versus Linear Models" on page 4-20 to decide whether it is appropriate to detrend your data.

It is only possible to remove a zero-order trend from frequency-domain data. Detrending frequency-domain data is equivalent to setting the response to `0` at zero frequency.

Note You can only remove piecewise linear trends using `detrend` and not System Identification Tool.

To remove mean values from each channel in `data`, which is an `iddata` object that stores time-domain data or frequency-domain data, use the following syntax:

```
data = detrend(data);
```

To subtract one linear trend, use the following syntax:

```
data = detrend(data,1);
```

In this case, `1` indicates that a first-order trend is removed from each channel.

You can also remove several linear trends that have connections at specified "breakpoints". For example:

```
data = detrend(data,1,[30 60 90]);
```

The vector `[30 60 90]` specifies the sample indexes of the data where breakpoints occur.

Resampling Data

Resampling data in System Identification Toolbox 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 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 (zero-order or first-order hold).

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

This section discusses the following topics:

- "Resampling Data Using System Identification Tool" on page 4-25
- "Using the resample Function" on page 4-25
- "Resampling Your Signal Without Aliasing Effects" on page 4-27

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 System Identification Tool

Use System Identification Tool to resample time-domain data. To specify additional options, such as the prefilter order, see "Using the resample Function" on page 4-25.

System Identification Tool uses `idresamp` to interpolate or decimate the data. For more information about this function, type `help idresamp` at the MATLAB prompt.

For general information about working with System Identification Tool, see Chapter 2, "Working with the System Identification Tool GUI".

To create a new data set by resampling the input and output signals:

1. Import time-domain data into the System Identification Tool, as described in "Creating Data Sets in the System Identification Tool" on page 3-13.
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 System Identification Toolbox window.
6. Click **Close** to close the Resample dialog box.

Using the resample Function

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 Signal Processing Toolbox function, 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 pages.

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 Signal Processing Toolbox 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 more information, see "iddata Properties" on page 3-34.

Resampling Your Signal Without Aliasing Effects

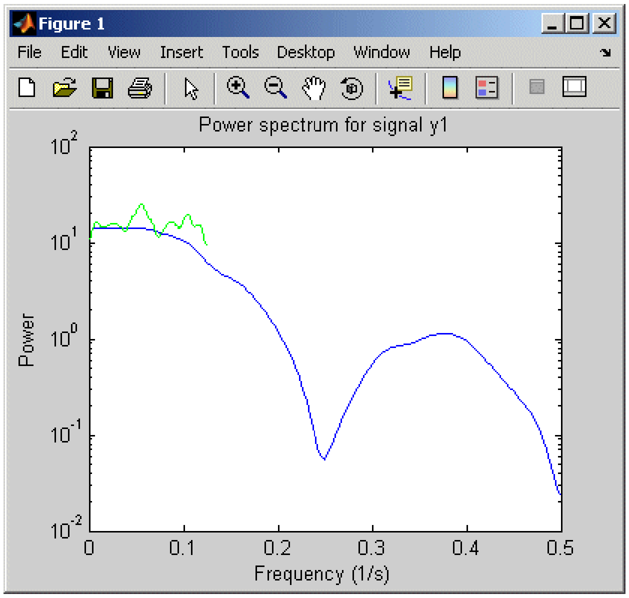
Typically, you decimate a signal to remove the high-frequency contributions that result from noise from the total energy. Ideally, you want to remove the energy contribution due to noise and preserve the energy density of the signal.

The function `resample` performs the decimation without aliasing effects. This function 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 and the Sampling Interval" on page 5-40.

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)
```

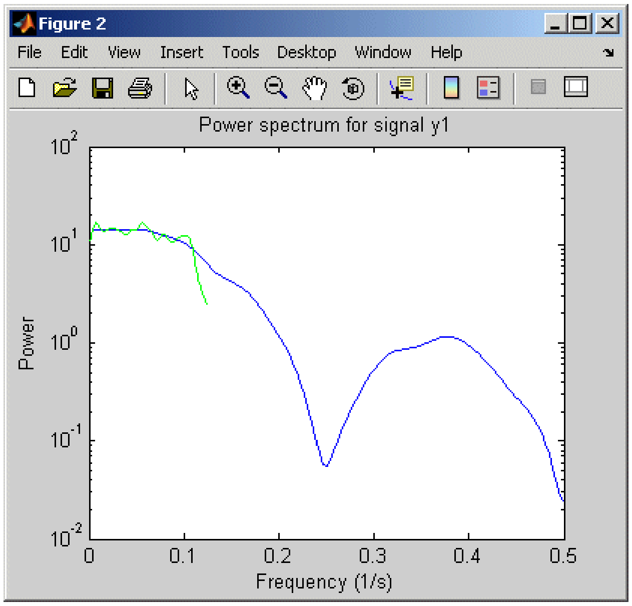


Folding Effects With Manual Decimation

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 effect when you use a function such as resample to eliminate aliasing.



No Folding Effects When Using resample

Filtering the Data

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

This section describes how to filter data before model estimation. You can also filter data during linear model estimation by setting the `Focus` property of the estimation algorithm to `Filter` and specifying the filter characteristics. For more information about model properties, see the `Algorithm Properties` reference pages.

This section discusses the following topics:

- "Deciding to Prefilter Your Data" on page 4-31
- "Filtering Data Using System Identification Tool" on page 4-32
- "Filtering Data Using the idfilt Function" on page 4-35

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.

Deciding 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 "Detrending Data" on page 4-20.

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 (see "Plotting Data" on page 4-7). 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, we recommend that you avoid prefiltering the data. Instead, set the `Focus` property of the estimation algorithm to `Simulation`. For more information, see the `Algorithm Properties` reference pages.

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.

Filtering Data Using System Identification Tool

System Identification Tool lets you filter time-domain, frequency-domain, or frequency-response data by enhancing or selecting specific passbands. The Tool filters time-domain data using a fifth-order Butterworth filter. For frequency-domain and frequency-response data, *filtering* is equivalent to selecting specific data ranges.

This section discusses the following topics:

- "Filtering Time-Domain Data" on page 4-32
- "Filtering Frequency-Domain or Frequency-Response Data" on page 4-34

Filtering Time-Domain Data

To create a filtered data set:

1. Import time-domain data into the System Identification Tool, as described in "Creating Data Sets in the System Identification Tool" on page 3-13.

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 `etfe`).

   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 window, and specify new model settings.

   ---

4. If your data contains multiple I/O 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 I/O 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.
   - By specifying the **Range** — Edit the beginning and the end frequency values.

     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` — Removes 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 5–9.

Filtering Frequency-Domain or Frequency-Response Data

To select a range of data, which is the equivalent of filtering frequency-domain and frequency-response data:

1. Import data into the System Identification Tool, as described in "Creating Data Sets in the System Identification Tool" on page 3-13.

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 I/O 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 I/O 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.

- By specifying the **Range** — Edit the beginning and the end frequency values.

  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` — Removes 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 5–8.

Filtering Data Using the idfilt Function

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)
```

This section discusses the following topics:

- "Simple Passband Filter" on page 4-36
- "Defining a Custom Filter" on page 4-36
- "Causal and Noncausal Filters" on page 4-37

Simple Passband 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, `wl` 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 Signal Processing Toolbox. For frequency-domain data, select the indicated portions of the data to perform passband filtering.

Defining a Custom Filter

You can define a general single-input and 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 Systems Toolbox documentation.

{num,den} defines the filter as a transfer function as a cell array of numerator and denominator filter coefficients.

{A,B,C,D} is a cell array of SISO state-space matrices.

Specifically for frequency-domain data, you specify the frequency response of the filter:

```
filter = Wf
```

Here, Wf is a vector of real or complex values that define the filter frequency response, where the inputs and outputs of data at frequency data.Frequency(kf) are multiplied by Wf(kf). Wf is a column vector with the length equal to the number of frequencies in data.

When data contains several experiments, Wf is a cell array with the length equal to the number of experiments in data.

Causal and Noncausal Filters

For time-domain data, the filtering is causal by default. Causal filters typically introduce a phase shift in the results. To use a noncausal zero-phase filter (corresponding to filtfilt in Signal Processing Toolbox), 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 page “Transforming Between

Frequency-Domain and Frequency-Response Data" on page 3-68). The result is an `iddata` object.

Selecting Data

When your data set contains undesirable features, such as missing data, outliers, level changes, and disturbances, you can select one or more portions of the data that are suitable for identification and exclude the rest. Later you can merge several data segments into a single multiexperiment data set and identify an average model. For more information, see "Creating Data Sets in the System Identification Tool" on page 3-13 or "Creating iddata Objects" on page 3-31.

If you only have one data set, we recommend that you split it into two portions. Then, use one portion for model estimation and use the other portion for model validation. Both portions of the data set must be contain enough samples to adequately represent the system and the inputs must provide suitable excitation to the system.

Note Selecting **Preprocess > Quick start** performs the following actions simultaneously: 1) remove the mean value from each channels, 2) split data into two halves, 3) specify the first half as estimation data for models (or **Working Data**), and 4) specify the second half as **Validation Data**.

This section discusses the following topics:

- "Selecting Data in the System Identification Tool" on page 4-39
- "Selecting Data in the MATLAB Command Window" on page 4-41

Selecting potions of frequency-domain data is equivalent to filtering the data. For more information about filtering, see "Filtering the Data" on page 4-31.

Selecting Data in the System Identification Tool

System Identification Tool lets you view time-domain or frequency-domain data on a plot and select the regions of interest. 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 Working Data and Validation Data” on page 2-23.

This sections contains the following topics:

- “Selecting a Range for Time-Domain Data” on page 4-40
- “Selecting a Range of Frequency-Domain Data” on page 4-41

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.

Note You cannot extract data in data sets containing multiple experiments. For more information about multiexperiment data, see “Working with Multiexperiment Data” on page 3-21.

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, as described in “Creating Data Sets in the System Identification Tool” on page 3-13.

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 one 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.

  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.

  Example: `342 654`

  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 4-6.

Selecting a Range of Frequency-Domain Data

Selecting a range of values in frequency domain is equivalent to filtering the data in System Identification Toolbox. For more information on data filtering, see "Filtering Frequency-Domain or Frequency-Response Data" on page 4-34.

Selecting Data in the MATLAB Command Window

Selecting ranges of data values is equivalent to subreferencing the data.

For more information about subreferencing time-domain and frequency-domain data stored in `iddata` objects, see "Subreferencing iddata Objects" on page 3-39.

For more information about subreferencing frequency-response data, see "Subreferencing idfrd Objects" on page 3-55.

Handling Complex-Valued Data

System Identification Toolbox successfully handles complex data with several exceptions.

This section discusses the following topics:

- "Supported Operations for Complex Data" on page 4-42
- "Manipulating Complex iddata Signals" on page 4-42

Supported Operations for Complex Data

System Identification Toolbox estimation routines support complex data. For example, the following estimation functions estimate complex models from complex data: `ar`, `armax`, `arx`, `bj`, `covf`, `ivar`, `iv4`, `oe`, `pem`, `spa`, and `n4sid`

Model transformation routines, such as `freqresp`, `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 functions `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.

Manipulating Complex iddata Signals

If the `iddata` object `data` contains complex values, you can use the following functions to process the complex data and create a new `iddata` object:

Function	Description
`abs(data)`	Absolute value of complex signals in `iddata` object.
`angle(data)`	Phase angle (in radians) of each complex signals in `iddata` object.

Function	**Description**
`complex(data)`	For time-domain data, this function 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
```

Estimating Linear Nonparametric and Parametric Models

Overview of Linear Nonparametric and Parametric Models

System Identification Toolbox lets you estimate linear nonparametric and parametric models to fit input-output data or time-series data.

System identification is typically a trial-and-error process, where you estimate and validate different types of models until you find the simplest model that adequately captures the dynamics of your system.

This section discusses the following topics:

- "Definitions of Nonparametric and Parametric Models" on page 5-2
- "Supported Models" on page 5-2
- "Before You Begin" on page 5-3

Definitions of Nonparametric and Parametric Models

Nonparametric models consist of data tables or curves and are not represented by a compact mathematical formula with adjustable parameters. Thus, nonparametric models do not impose a structure on your system. Typical nonparametric methods for linear models include *correlation analysis*, which estimates the impulse or step response of the system, and *spectral analysis*, which estimates the frequency response (periodogram) of the system.

Parametric models have a well-defined mathematical structure, and this structure is fit to the input-output data by adjusting the coefficient values, or *model parameters*. Parametric identification methods use numerical search to find the parameter values that correspond to the best agreement between simulated and measured output.

Supported Models

The following types of linear nonparametric and parametric models are supported by System Identification Toolbox:

- "Low-Order, Continuous-Time Process Models" on page 5-4
- "Correlation Analysis Models" on page 5-23
- "Spectral Analysis Models" on page 5-31

- “Black-Box Polynomial Models” on page 5-42
- “State-Space Models” on page 5-67
- “Time-Series Models” on page 5-94

This User’s Guide describes how to estimate these model both in the System Identification Tool GUI and in the MATLAB Command Window.

Before You Begin

Before you begin estimating models, import the data into MATLAB, and represent the data using System Identification Toolbox format. If you are using the System Identification Tool, then import the data into the GUI to make the data available to System Identification Toolbox. However, if you prefer to work in the MATLAB Command Window, then represent your data as an `iddata` or `idfrd` object. For more information about representing your data for system identification, see Chapter 3, “Representing Data for System Identification”.

After representing data in System Identification Toolbox format, plot the data on a time plot or an estimated frequency response plot to examine the data features. You can also use the `advice` command to analyze the data for the presence of constant offsets and trends, delay, feedback, and nonlinearity, and determine the order of excitation persistence.

You can preprocess your data by removing offsets and linear trends, interpolating missing values, filtering to emphasize a specific frequency range, or resampling using a different time interval.

For more information about types of available date plots and data-preprocessing operations, see Chapter 4, “Plotting and Preprocessing Data”.

Low-Order, Continuous-Time Process Models

Continuous-time process models can contain up to three poles, one zero, one integrator, and a time delay. You can use such model structures to estimate the static gain, the time delay before the system output responds to the input, and the characteristic time constants associated with poles and zeros.

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

- Time- or frequency-domain `iddata` or `idfrd` data object.
- Real data, or complex data in time domain only.
- Single-output data.

For a tutorial on estimating continuous-time process models in the GUI, see "Estimating Continuous-Time Process Models Using the System Identification Tool" in *Getting Started with System Identification Toolbox*.

This section describes the procedures required to estimate process models in the System Identification Tool GUI and the MATLAB Command Window. It includes the following topics:

After estimating the model, see Chapter 9, "Plotting and Validating Models" to validate the model.

Definition of a Process Model

In general, a linear system is characterized by a transfer function *G*, which takes the input *u* to the output *y*:

$$y = Gu$$

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

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

The structure of a continuous-time process model describe 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.

You use the System Identification Tool to specify different process-model structures by varying the number of poles, adding an integrator, or adding or removing a time delay or zero. The highest model order you can specify in this Toolbox is 3, and the poles can be real or complex (underdamped modes).

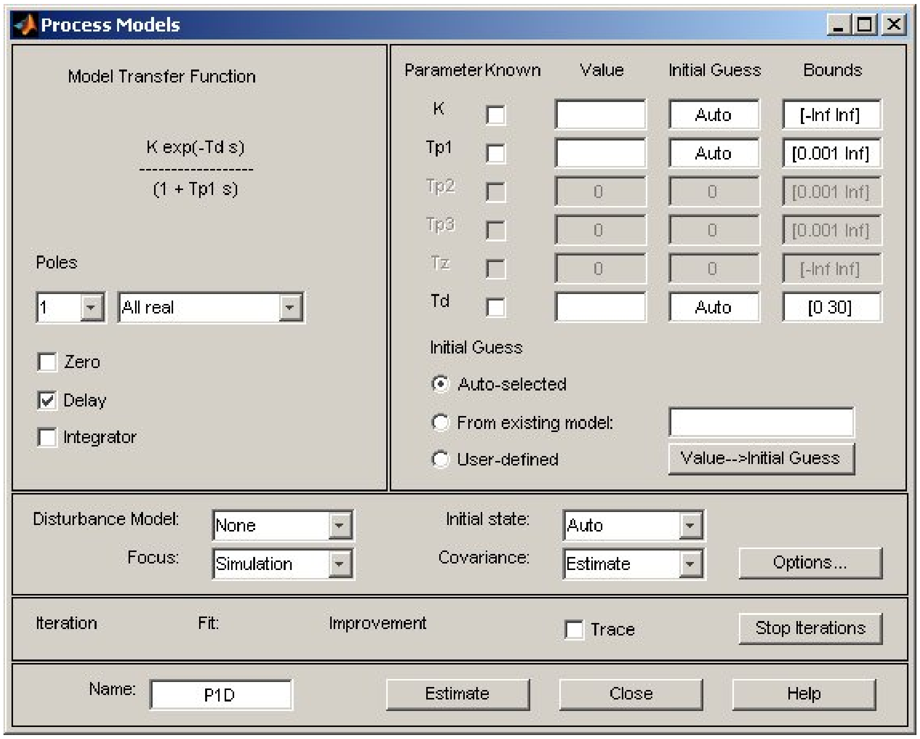
For example, the following model structure is a first-order continuous-time process model, where K is the static gain, T_{p1} is a time constant, and T_d is the input-to-output delay:

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

Estimating Process Models in the GUI

The following procedure describes how to estimate a process model in the System Identification Tool and assumes that you already have the appropriate data in the Data Board.

1 In the System Identification Tool window, select **Estimate > Process models** to open the Process Models dialog box.



Process Models

Model Transfer Function

$$\frac{K \exp(-Td\ s)}{(1 + Tp1\ s)}$$

Poles: 1, All real

Zero [✓ Delay] Integrator

Parameter	Known	Value	Initial Guess	Bounds
K	☐		Auto	[-Inf Inf]
Tp1	☐		Auto	[0.001 Inf]
Tp2	☐	0	0	[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

Disturbance Model: None Focus: Simulation Initial state: Auto Covariance: Estimate Options...

Iteration Fit: Improvement Trace Stop Iterations

Name: P1D Estimate Close Help

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 Models" on page 5-18.

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).

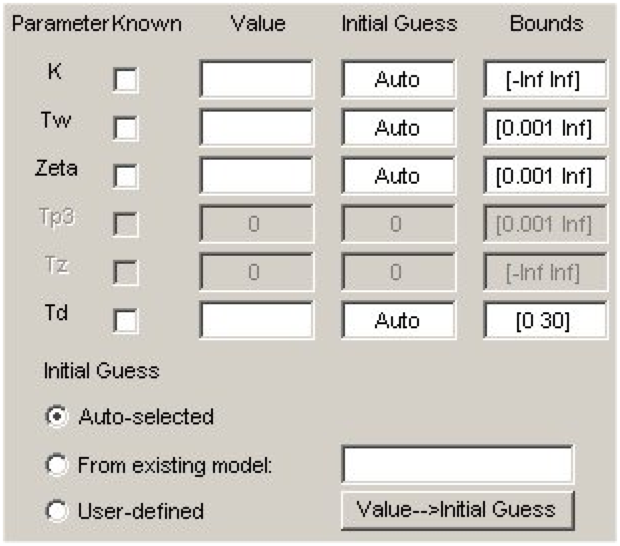
- Select the **Zero** check box to include a zero—a numerator term other than a constant, or clear the check box to exclude the zero.
- Select the **Delay** check box to include a delay, or clear the check box to exclude the delay.
- Select the **Integrator** check box to include an integrator (self-regulating 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 "Specifying the Process-Model Structure" on page 5-17.

---

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]

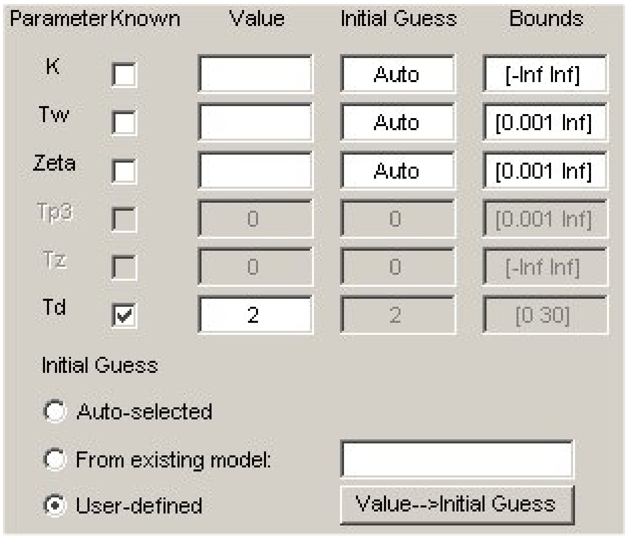
From existing model:

User-defined

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. Otherwise, you should keep the default values.

For example, the following 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]

6 In the **Disturbance Model** list, select one of the available options. For more information about each option, see "Choosing the Disturbance Model Structure" on page 5-19.

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 "Setting the Frequency-Weighing Focus" on page 5-20.

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 the Initial States" on page 5-21.

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 window.

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.

14 To refine the current estimate, click 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**.

15 To plot the model, select the appropriate check box in the Model Views area of the System Identification Tool window. For more information about working with plots and validating models, see Chapter 9, "Plotting and Validating Models".

If your model is not sufficiently accurate, try another model structure.

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 window. For more information about working with models, see Chapter 10, "Postprocessing and Using Estimated Models".

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. You can also use `pem` to refine parameter estimates of an existing process model, as described in "Refining Models" on page 1-46.

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)
```

`data` is the estimation data and `mod_struc` is a string that represents the process model structure, as described in "Specifying the Process-Model Structure" on page 5-17.

Note 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 – Using pem to Estimate Process Models With Fixed Parameters" on page 5-14.

The property-value pairs specify any model properties that configure the estimation algorithm and the initial conditions. For more information on accessing and setting model properties, see "Model Properties" on page 1-30.

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

The following examples demonstrate how to estimate process models in the MATLAB Command Window:

- "Example – Using pem to Estimate Process Models with Free Parameters" on page 5-12
- "Example – Using pem to Estimate Process Models With Fixed Parameters" on page 5-14

Example – Using pem to Estimate Process Models with Free Parameters

This example demonstrates how to estimate all of 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.

In this example, 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 responds with 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 the `get` command to get the value of any model parameter. Alternatively, you can use dot notation. For example, to get the Value field in the `K` structure, type the following command:

```
m.K.value
```

Example – Using pem to Estimate Process Models With Fixed Parameters

If you know the values of certain parameters and want to omit estimating these values, you must first construct the `idproc` model object and then specify the fixed parameters.

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 responds with 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 the Initial States” on page 5-21.
- `min` field specifies the minimum bound on the parameter.
- `max` field specifies the maximum bound on the parameter.
- `value` field 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 returns 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	Syntax
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})`

Specifying the Process-Model Structure

This section describes how to specify the model structure in the estimation procedures "Estimating Process Models in the GUI" on page 5-6 and "Using pem to Estimate Process Models" on page 5-11.

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.

In the MATLAB Command Window. Specify the model structure using an acronym that includes the following letters and numbers:

- (Required) `P` for 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 indicated enforced integration.

Typically, you specify the model-structure acronym as a string argument in the estimation function `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,'P0ID')` to estimate the following structure:

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

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

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

For more information about estimating in the MATLAB Command Window, see "Using pem to Estimate Process Models" on page 5-11.

Estimating Multiple-Input Models

If your model contains multiple inputs, you can specify whether to estimate the same transfer function for all inputs, or a different transfer for each input. The information in this section supports the estimation procedures "Estimating Process Models in the GUI" on page 5-6 and "Using pem to Estimate Process Models" on page 5-11.

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 the **Same structure for all channels** check box clear, and create a different transfer function for each input.

In the MATLAB Command Window. Specify the model structure as a cell array of acronym strings in the estimation function `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`.

Choosing the Disturbance Model Structure

This section describes how to specify a noise model in the estimation procedures "Estimating Process Models in the GUI" on page 5-6 and "Using pem to Estimate Process Models" on page 5-11.

In addition to the transfer function G, a linear system can include an additive noise term He, as follows:

$$y = Gu + He$$

where e is white noise.

You can estimate only the dynamic model G, or estimate both the dynamic model and the disturbance model H. For process models, H is a rational transfer function C/D, where the C and D polynomials for a first- or second-order ARMA model.

In the GUI. To specify whether to include or exclude a noise model in the Process Models dialog, 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 the **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.

In the MATLAB Command Window. Specify the disturbance model as an argument in the estimation function `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 pages.

Setting the Frequency-Weighing Focus

You can specify how the estimation algorithm weighs the fit at various frequencies. This information supports the estimation procedures "Estimating Process Models in the GUI" on page 5-6 and "Using pem to Estimate Process Models" on page 5-11.

In the System Identification Tool GUI. Set the **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 on model stability, see "Unstable Models" on page 9-66.
- `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 4-36 or "Defining a Custom Filter" on page 4-36. This prefiltering applies only for estimating the dynamics from input to output. The disturbance model is determined from the estimation data.

In the MATLAB Command Window. Specify the focus as an argument in the estimation function `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 the Initial States

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 "Estimating Process Models in the GUI" on page 5-6 and "Using pem to Estimate Process Models" on page 5-11.

In the System Identification Tool GUI. Set the **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.

In the MATLAB Command Window. Specify the initial states as an argument in the estimation function `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 pages.

Correlation Analysis Models

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

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

- Real or complex time-domain `iddata` object. To learn more about estimating time-series models, see “Time-Series Models” on page 5-94.
- Frequency-domain `iddata` or `idfrd` object with the sampling interval $T \neq 0$.
- Single- or multiple-output data.

This section describes the procedures required to estimate correlation-analysis models in the System Identification Tool GUI and the MATLAB Command Window. It include the following topics:

- “Definition of Correlation Analysis” on page 5-23
- “Estimating Correlation Models in the GUI” on page 5-25
- “Using impulse and step to Estimate Correlation Models” on page 5-26
- “Using impulse and step to Compute Model Data” on page 5-27
- “Identifying Delay from Impulse-Response Plots” on page 5-28

After estimating the model, see Chapter 9, “Plotting and Validating Models” to validate the model.

Definition of Correlation Analysis

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:

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

Impulse response is the response to an impulse input *u(t)*. That is, *u(t)*=1 when *t*=0 and *u(t)*=0 when *t*>0 for discrete-time data. *Step response* is the response to a step input.

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.

Estimating Correlation Models in the GUI

The following procedure describes how to estimate impulse- and step-response models in the System Identification Tool using correlation analysis. This procedure assumes that you already have the appropriate data in the Data Board.

1 In the System Identification Tool window, 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**, specify the filter order.

The prewhitening filter is determined by modeling the input as an Auto-Regressive (AR) 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 window.

6 In the Correlation Model dialog box, click **Close**.

7 To view the transient response plot, select the **Transient resp** check box in the System Identification Tool window. For more information about

working with this plot and selecting to view impulse- versus step-response, see “Impulse and Step Response Plots” on page 9-21.

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 window. For more information about continuing to work with models in the MATLAB workspace, see Chapter 10, “Postprocessing and Using Estimated Models”.

Using impulse and step to Estimate Correlation Models

You can use `impulse` and `step` commands to estimate impulse and step response directly from time- or frequency-domain data using correlation analysis. Both `impulse` and `step` produce the same FIR model.

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

The following tables summarize the commands for computing impulse- and step-response models. The resulting models are stored as `idarx` model objects and contain impulse-response coefficients in the model parameter `B`. For more information about models objects, see “Working with Model Objects” on page 1-19. For detailed information about these commands, see the corresponding reference pages.

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 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 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.

Using impulse and step to Compute Model Data

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, perform the following procedure:

1 Compute the FIR model by applying either `impulse` or `step` commands on the data, as described in "Using impulse and step to Estimate Correlation Models" on page 5-26.

2 Apply the following syntax on the resulting model:

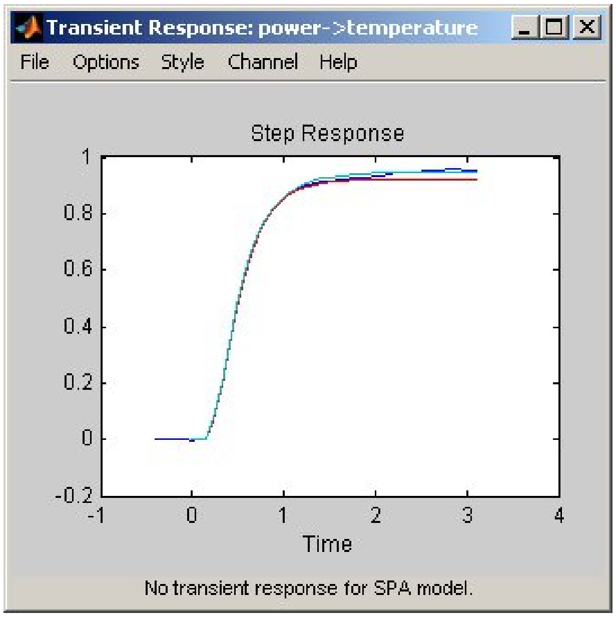
```
% To compute impulse-response data
[y,t,ysd] = impulse(model)
% To compute step-response data
[y,t,ysd] = step(model)
```

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

Identifying Delay from Impulse-Response Plots

You can use impulse-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 window. For example, the following step response plot shows a time delay of about 0.25 seconds before the system responds to the input.



No transient response for SPA model.

Step Response Plot

In the MATLAB Command Window. 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 sec.

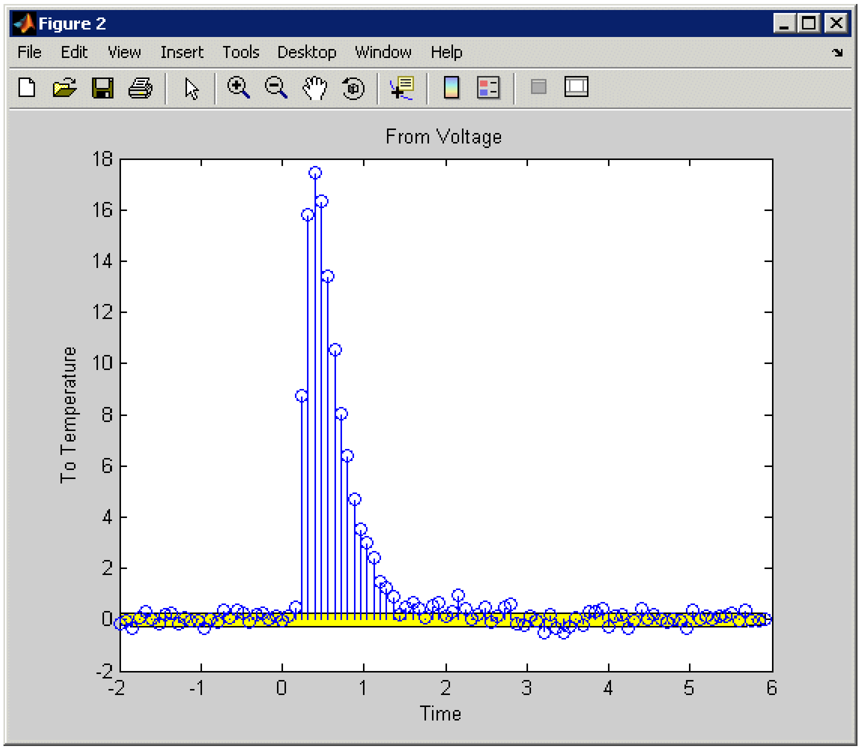


Figure 2
File Edit View Insert Tools Desktop Window Help
From Voltage
To Temperature
Time
18
16
14
12
10
8
6
4
2
0
-2
-2
-1
0
1
2
3
4
5
6

Spectral Analysis Models

Spectral analysis is a nonparametric estimate of frequency response.

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

- Complex or real data.
- Time- or frequency-domain `iddata` or `idfrd` data object. To learn more about estimating time-series models, see "Time-Series Models" on page 5-94.
- Single- or multiple-output data.

This section describes the procedures required to estimate spectral-analysis models in the System Identification Tool GUI and the MATLAB Command Window. It include the following topics:

After estimating the model, see Chapter 9, "Plotting and Validating Models" to validate the model.

Spectral Analysis Algorithm

You can estimate the frequency-response function of dynamic systems using spectral analysis.

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

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

where *u(t)* and *y(t)* are the input and output signals, respectively. *G(q)* is called the transfer function of the system—it takes the input to the output and captures the system dynamics. 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:

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

G(q) that is evaluated on the unit circle, *G(q=e^{iw})*, is the *frequency-response function*.

Together, *G(q=e^{iw})* and the output noise spectrum $\hat{\Phi}_v(\omega)$ comprise the frequency-domain description of the system.

According to the Blackman-Turkey approach, the estimated frequency-response function is given by the following equation:

$$\hat{G}_N\left(e^{i\omega}\right) = \frac{\hat{\Phi}_{yu}(\omega)}{\hat{\Phi}_u(\omega)}$$

In this case, ^ represents approximate quantities. For a derivation of this equation, see the chapter on nonparametric time- and frequency-domain methods in *System Identification: Theory for the User*, Second Edition, by Lennart Ljung, Prentice Hall, 1999.

The output noise spectrum is given by the following equation:

$$\hat{\Phi}_v(\omega) = \hat{\Phi}_y(\omega) - \frac{\left|\hat{\Phi}_{yu}(\omega)\right|^2}{\hat{\Phi}_u(\omega)}$$

This equation for the noise spectrum is derived by assuming the linear relationship $y(t) = G(q)u(t) + v(t)$, that *u(t)* is independent of *v(t)*, and the following relationships between the spectra:

$$\Phi_y(\omega) = \left|G\left(e^{i\omega}\right)\right|^2 \Phi_u(\omega) + \Phi_v(\omega)$$

$$\Phi_{yu}(\omega) = G\left(e^{i\omega}\right)\Phi_u(\omega)$$

where the noise spectrum is given by the following equation:

$$\Phi_v(\omega) \equiv \sum_{\tau=-\infty}^{\infty} R_v(\tau)e^{-iw\tau}$$

$\hat{\Phi}_{yu}(\omega)$ is the output-input cross-spectrum and $\hat{\Phi}_u(\omega)$ is the input spectrum.

When you use System Identification Toolbox to estimate the frequency function, the spectral estimation algorithms (such as `spa`) perform the following steps:

- Compute the covariances and cross-covariance from *u(t)* and *y(t)*, as follows:

$$\hat{R}_y(\tau) = \frac{1}{N}\sum_{t=1}^{N} y(t+\tau)y(t)$$
$$\hat{R}_u(\tau) = \frac{1}{N}\sum_{t=1}^{N} u(t+\tau)u(t)$$
$$\hat{R}_{yu}(\tau) = \frac{1}{N}\sum_{t=1}^{N} y(t+\tau)u(t)$$

- Compute the Fourier transforms of the covariances and the cross-covariance, as follows:

$$\hat{\Phi}_y(\omega) = \sum_{\tau=-M}^{M} \hat{R}_y(\tau) W_M(\tau) e^{-i\omega\tau}$$

$$\hat{\Phi}_u(\omega) = \sum_{\tau=-M}^{M} \hat{R}_u(\tau) W_M(\tau) e^{-i\omega\tau}$$

$$\hat{\Phi}_{yu}(\omega) = \sum_{\tau=-M}^{M} \hat{R}_{yu}(\tau) W_M(\tau) e^{-i\omega\tau}$$

  where $W_M(\tau)$ is called the *lag window* with the width M.

- Compute the frequency response function $\hat{G}_N\left(e^{i\omega}\right)$ and the output noise spectrum $\hat{\Phi}_v(\omega)$

Alternatively, the disturbance *v(t)* can be described as filtered white noise:

$$v(t) = H(q)e(t)$$

where *e(t)* is the white noise with variance λ and the noise power spectrum is given by the following equation:

$$\Phi_v(\omega) = \lambda\left|H\left(e^{i\omega}\right)\right|^2$$

Estimating Spectral Models in the GUI

The following procedure describes how to estimate spectral models in the System Identification Tool and assumes that you already have the appropriate data in the Data Board.

1 In the System Identification Tool window, 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 5-37.

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 "Specifying the Frequency Resolution" on page 5-38. 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 window.

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 window. For more information about working with this plot, see "Frequency Response Plots" on page 9-29.

9 To view the estimated disturbance spectrum, select the **Noise spectrum** check box in the System Identification Tool window. For more information about working with this plot, see “Noise Spectrum Plots” on page 9-36.

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.

Using etfe, spa, and spafdr to Estimate Spectral Models

You can use the `etfe`, `spa`, and `spafdr` functions to estimate spectral models. The following table provides a brief description of each function and usage examples.

The resulting models are stored as `idfrd` model objects. For more information about models objects, see “Working with Model Objects” on page 1-19.

For detailed information about the functions and their arguments, see the corresponding reference pages.

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.

Selecting the Method for Computing Spectral Models

This section describes how to select the method for computing spectral models in the estimation procedures "Estimating Spectral Models in the GUI" on page 5-34 and "Using etfe, spa, and spafdr to Estimate Spectral Models" on page 5-36.

System Identification Toolbox provides 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. For more information about this algorithm, see "Spectral Analysis Algorithm" on page 5-31.

- `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.

Specifying the Frequency Resolution

This section provides information about specifying frequency resolution in the estimation procedures "Estimating Spectral Models in the GUI" on page 5-34 and "Using etfe, spa, and spafdr to Estimate Spectral Models" on page 5-36.

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`, as discussed in the following topics:

- "Frequency Resolution for etfe and spa" on page 5-38
- "Frequency Resolution for spafdr" on page 5-39
- "etfe Frequency Resolution for Periodic Input" on page 5-39

Frequency Resolution for etfe and spa

For `etfe` and `spa`, the frequency resolution is approximately equal to the following value:

$$\frac{2\pi}{M}\left(\frac{\text{radians}}{\text{sampling interval}}\right)$$

where M is a scalar integer that sets the size of the lag window.

A large value of M gives good resolution, but results in mode uncertain estimates.

The default value of M for `spa` is good for systems that do not have sharp resonances.

For `etfe`, the default value of M gives the maximum resolution.

Frequency Resolution for spafdr

In case of `etfe` and `spa`, the frequency response is defined over a uniform frequency range, *0-*$F_s/2$ radians per second, where F_s is the sampling frequency—equal to twice the Nyquist frequency. In contrast, `spafdr` lets you increase the resolution in a specific frequency range, such as near a resonance frequency. Conversely, you can make the frequency grid coarser in the region where the noise dominates—at higher frequencies, for example. Such customizing of the frequency grid assists in the estimation process by achieving high fidelity in the frequency range of interest.

For `spafdr`, the frequency resolution around the frequency k is the value $R(k)$. You can enter $R(k)$ in any *one* of the following ways:

- Scalar value of the constant frequency resolution value in radians per second.

> **Note** The scalar R is inversely related to the M value used for `etfe` and `spa`.

- Vector of frequency values the same size as the frequency vector.
- Expression using MATLAB workspace variables and evaluates to a resolution vector that is the same size as the frequency vector.

The default value of the resolution for `spafdr` is twice the difference between neighboring frequencies in the frequency vector.

etfe Frequency Resolution for Periodic Input

If the input data is marked as periodic and contains an integer number of periods (`data.Period` is an integer), `etfe` computes the frequency response at frequencies $\frac{2\pi k}{T}\left(\frac{\text{k}}{\text{Period}}\right)$ where $k = 1,2,\ldots,\text{Period}$.

For periodic data, the frequency resolution is ignored.

Spectrum Normalization and the Sampling Interval

The *spectrum* of a signal is the square of the Fourier transform of the signal. The spectral estimate using the function `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 MATLAB:

```
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 responds with 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.

Black-Box Polynomial Models

A *black-box model* is a flexible structure that is capable of describing many different systems. The parameters of a black-box model might not have any physical interpretation. 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 "Time-Series Models" on page 5-94.
- Real data or complex data in any domain.
- Single-output and multiple-output.

This section describes the procedures required to estimate single- and multiple-output polynomial models in the System Identification Tool GUI and the MATLAB Command Window. It includes the following topics:

Definition of Polynomial Models

System Identification Toolbox supports linear polynomial model structures that have the following general representation:

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

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.

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.

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.

This section discusses the following aspects of black-box polynomial models:

- "Understanding the Time-Shift Operator q" on page 5-44
- "Discrete-Time Representation" on page 5-44
- "Continuous-Time Representation" on page 5-47
- "Multiple-Output ARX Models" on page 5-47

For more information on the family of transfer-function models, see the corresponding section in *System Identification: Theory for the User*, Second Edition, by Lennart Ljung, Prentice Hall, 1999.

Understanding the Time-Shift Operator q

The general polynomial equation is written in terms of the time-shift operator q. To understand this time-shift operator, consider the following discrete-time difference equation:

$$\begin{aligned} &y(t) + a_1 y(t-T) + a_2 y(t-2T) = \\ &\qquad b_1 u(t-T) + b_2 u(t-2T) \end{aligned}$$

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 $qu(t) = u(t-T)$:

$$\begin{aligned} &y(t) + a_1 q^{-1} y(t) + a_2 q^{-2} y(t) = \\ &\qquad \mathrm{b}_1 q^{-1} u(t) + b_2 q^{-2} u(t) \end{aligned}$$

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.

Discrete-Time Representation

The following table summarizes common linear polynomial model structures supported by System Identification Toolbox. These model structures are subsets of the following general polynomial equation:

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

The model structures differ by how many of these polynomials are included in the structure. Thus, different model structures provide varying levels of flexibility for modeling the dynamics and noise characteristics.

The 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 5-62.

Model Structure	Discrete-Time Form	Noise Model
ARX	$A(q)y(t) = \sum_{i=1}^{nu} B_i(q)u_i\left(t - nk_i\right) + e(t)$	The noise model is $\frac{1}{A}$ and the noise is coupled to the dynamics model. ARX does not let you model noise and dynamics independently. Use ARX to have a simple model at good
ARMAX	$A(q)y(t) = \sum_{i=1}^{nu} B_i(q)u_i\left(t - nk_i\right) + C(q)e(t)$	Extends the ARX structure by providing more flexibility for modeling noise using the C parameters (a **M**oving **A**verage 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** The white noise source *e(t)* to $H = 1$ in the general equation.

Continuous-Time Representation

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 10-3.

You can also use System Identification Toolbox to 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.

Multiple-Output ARX Models

You can use a multiple-output ARX model to model a multiple-output dynamic system. The ARX model structure is 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}:

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

$A(q)$ 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≠j. This polynomial describes how the old values of the jth output are affected by the kth output. The ith row of *A(q)* represents the contribution of the past output values for predict the current value of the ith 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.

Estimating Model Orders and Input Delays

To estimate polynomial models, you must provide input delays and model orders. If you have insight into the physics of your system, you might be able to guess the number of poles and zeros. However, in most cases, you do not know the model orders in advance.

To help you get initial model orders and delays for your system, System Identification Toolbox lets you estimate a group of ARX models with a range of orders and delays and compares 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 on residual analysis plots, see "Residual Analysis Plots" on page 9-15.

This section discusses the following topics:

- "Estimating Orders and Delays in the GUI" on page 5-50
- "Estimating Orders in the MATLAB Command Window" on page 5-53
- "Estimating Delays in the MATLAB Command Window" on page 5-55
- "Using the ARX Model Structure Selection Plot" on page 5-55

Estimating Orders and Delays in the GUI

The following procedure describes how to estimate model orders and input delays in the System Identification Tool GUI and assumes that you already have the appropriate data in the Data Board.

1 In the System Identification Tool window, 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.

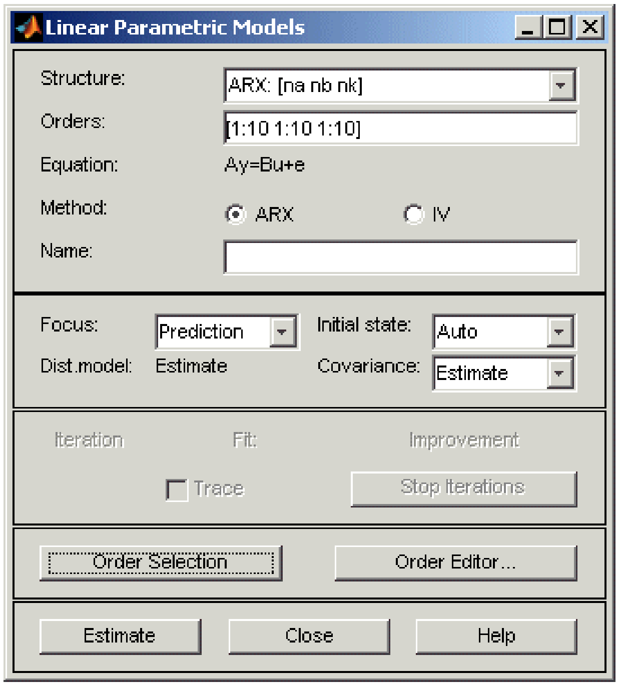
   Note For time-series models, select the AR model structure.

   ---

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:

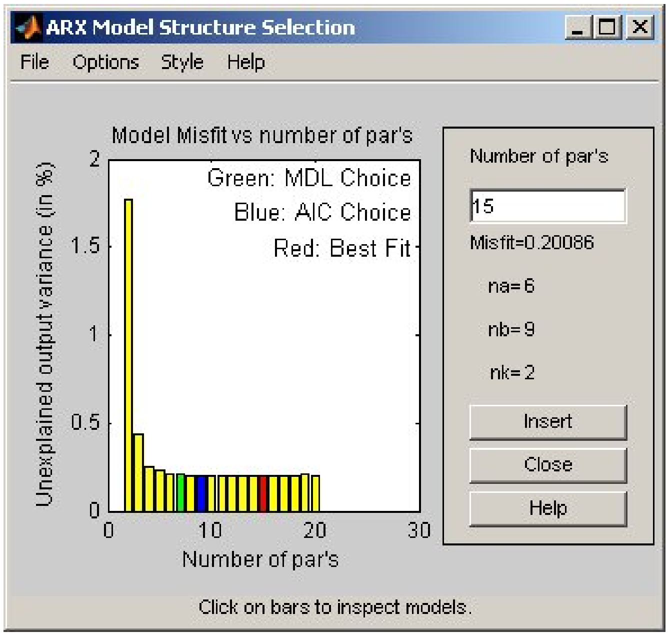
Focus: Prediction

Initial state: Auto

Dist.model: Estimate

Covariance: Estimate

3 Click **Estimate** to open the ARX Model Structure Selection plot, 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 "Using the ARX Model Structure Selection Plot" on page 5-55.

This action adds a new model to the Model Board in the System Identification Tool window. 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 2 samples.

5 Click **Close** to close the ARX Model Structure Selection plot.

After estimating model orders and delays, use these values as initial guesses for estimating other model structures, as described in "Estimating Polynomial Models in the GUI" on page 5-58.

Estimating Orders in the MATLAB Command Window

You can estimate model orders using the `struc`, `arxstruc`, and `selstruc` functions 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 you can select these data sets from a single experiment. For more information about subreferencing data, see “Subreferencing iddata Objects” on page 3-39 and “Subreferencing idfrd Objects” on page 3-55.

For an example of estimating model orders for a multiple-input system using these functions, see “Estimating Model Orders and Delays” in *Getting Started with System Identification Toolbox*.

struc. The `struc` function 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))
```

Note `struc` applies only to single-input and single-output models. If you have multiple inputs and want to use `struc`, apply this command to one input-output pair at a time.

---

arxstruc. The `arxstruc` function 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 function* 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 and 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` function takes the output from `arxstruc` and opens the ARX Model Structure Selection plot to guide your choice of the model order with the best performance.

For example, to open the ARX Model Structure Selection plot and interactively choose the optimum parameter combination, use the following command:

```
selstruc(V)
```

For more information about working with the ARX Model Structure Selection plot, see "Using the ARX Model Structure Selection Plot" on page 5-55.

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 5-62.

Estimating Delays in the MATLAB Command Window

The `delayest` function 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 MATLAB prompt:

```
delayest(data)
```

If your data has a single input, MATLAB responds with 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.

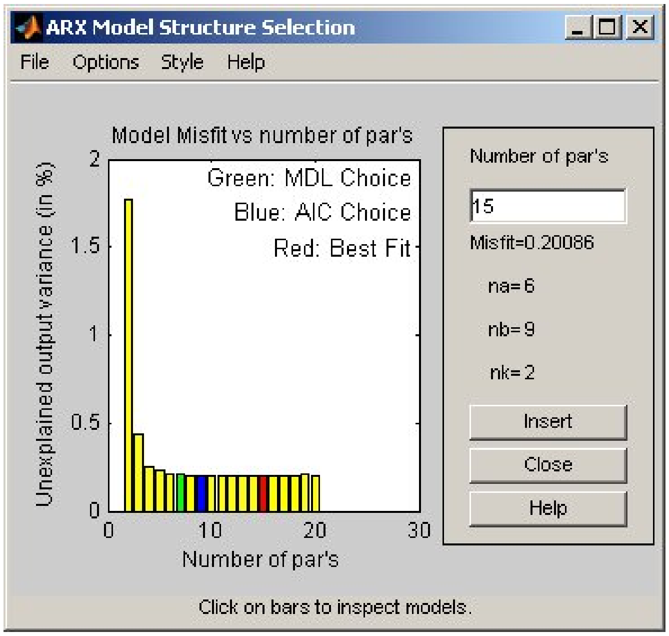
You can also use the ARX Model Structure Selection plot to estimate input delays and model order together, as described in “Estimating Orders in the MATLAB Command Window” on page 5-53.

Using the ARX Model Structure Selection Plot

You generate the ARX Model Structure Selection plot 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 5-50. To open this plot in the MATLAB Command Window, see “Estimating Orders in the MATLAB Command Window” on page 5-53.

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


ARX Model Structure Selection
File Options Style Help
Model Misfit vs number of par's
Green: MDL Choice
Blue: AIC Choice
Red: Best Fit
Unexplained output variance (in %)
Number of par's
Insert
Close
Help
Click on bars to inspect models.

Number of par's: 15

Misfit=0.20086

na= 6

nb= 9

nk= 2

The horizontal axis in the ARX Model Structure Selection plot 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 plot, 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 “Using Akaike’s Final Prediction Error and Information Criterion” on page 9-58.

Specifying Multiple-Input and Multiple-Output ARX Orders

To estimate a multiple-input and multiple-output (MIMO) ARX model, you must 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 “Multiple-Output ARX Models” on page 5-47.

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

In the MATLAB Command Window. Define variables that store the model order matrices and specify these variables in the estimation-command syntax. You can use the following syntax to estimate a model with these orders:

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

In the System Identification Tool GUI. You can enter the enter the matrices directly in the **Orders** field.

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

Estimating Polynomial Models in the GUI

The following procedure describes how to estimate a polynomial model in the System Identification Tool and assumes that you already have the appropriate data in the Data Board.

This procedure also requires that you select a model structure and specify model orders and delays. For more information on how to estimate model orders and delays, see "Estimating Orders and Delays in the GUI" on page 5-50.

If you are estimating a multiple-output ARX model, you must specify order matrices in the MATLAB workspace before estimation, as described in "Specifying Multiple-Input and Multiple-Output ARX Orders" on page 5-57.

1 In the System Identification Tool window, 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 "Definition of Polynomial Models" on page 5-43.

Note For time-series data, only AR and ARMA models are available. For more information about estimating time-series models, see "Time-Series Models" on page 5-94.

---

3 In the **Orders** field, specify the model orders, 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 "Specifying Multiple-Input and Multiple-Output ARX Orders" on page 5-57.

Tip To enter model orders and delays using the Order Editor dialog box, click **Order Editor**.

---

4 (For ARX models only) Select the estimation **Method** as **ARX** or **IV** (instrumental variable method). For more information about these methods, see "Supported Estimation Algorithms" on page 1-16.

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 "Setting the Frequency-Weighing Focus" on page 5-64.

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 the Initial States" on page 5-21.

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 (For ARMAX, OE, and BJ 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 window.

11 (For 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.

12 To plot the model, select the appropriate check box in the Model Views area of the System Identification Tool window. For more information about working with plots and validating models, see Chapter 9, “Plotting and Validating Models”.

If you get an inaccurate fit, try estimating a new model with different orders or structure.

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 window. For more information about working with models, see Chapter 10, "Postprocessing and Using Estimated Models".

Estimating Polynomial Models in the MATLAB Command Window

This section discusses the following topics:

- "Using arx and iv4 to Estimate ARX Models" on page 5-61
- "Using pem to Estimate Polynomial Models" on page 5-62

Using arx and iv4 to Estimate ARX Models

You can estimate single-output and multiple-output ARX models using the `arx` and `iv4` commands.

If you are estimating a multiple-output ARX model, you must specify order matrices in the MATLAB workspace before estimation, as described in "Specifying Multiple-Input and Multiple-Output ARX Orders" on page 5-57.

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 "Definition of Polynomial Models" on page 5-43.

The property-value pairs specify any model properties that configure the estimation algorithm and the initial conditions. For more information on accessing and setting model properties, see "Model Properties" on page 1-30.

Note You can specify all property-value pairs as a simple, 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 Toolbox does not support multiple-output continuous-time `idarx` models.

For detailed information about these commands, see the corresponding reference pages.

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.

You can also use `pem` to refine parameter estimates of an existing polynomial model, as described in "Refining Models" on page 1-46.

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 "Definition of Polynomial Models" on page 5-43.

Note 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 "Definition of Polynomial Models" on page 5-43, 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 on accessing and setting model properties, see "Model Properties" on page 1-30.

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. For example, to only model 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 detailed information about pem and idpoly, see the corresponding reference pages.

Setting the Frequency-Weighing Focus

You can specify how the estimation algorithm weighs the fit at various frequencies. This information supports the estimation procedures “Estimating Polynomial Models in the GUI” on page 5-58 and “Using pem to Estimate Polynomial Models” on page 5-62.

In the System Identification Tool GUI. Set the **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 on model stability, see "Unstable Models" on page 9-66.
- `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 4-36 or "Defining a Custom Filter" on page 4-36. This prefiltering applies only for estimating the dynamics from input to output. The disturbance model is determined from the unfiltered estimation data.

In the MATLAB Command Window. Specify the focus as an argument in the estimation function 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 the Initial States

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 "Estimating Polynomial Models in the GUI" on page 5-58 and "Using pem to Estimate Polynomial Models" on page 5-62. For more information about estimation algorithms, see "Supported Estimation Algorithms" on page 1-16.

In the System Identification Tool GUI. For ARMAX, OE, and BJ models, set the **Initial state** to one of the following options:

- Auto — Automatically chooses one of the following 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.
- 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.

In the MATLAB Command Window. Specify the initial states as an argument in the estimation function. 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 pages.

State-Space Models

State-space models are models that use state variables to describe a system by a set of first-order differential equations, rather than by one or more *n*th-order differential 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 an excellent choice for quick estimation because it requires only two parameters:

- `n` — The number of poles (size of the *A* matrix).
- `nk` — One or more input delays.

You can estimate linear state-space models from data with the following characteristics:

- Time- or frequency-domain data (`iddata` or `idfrd` data objects). To estimate state-space models for time-series data, see "Time-Series Models" on page 5-94.
- Real data or complex data in any domain.
- Single-output and multiple-output.

This section describes the procedures required to estimate single- and multiple-output state-space models in the System Identification Tool GUI and the MATLAB Command Window. It includes the following topics:

- "Estimating State-Space Models with Canonical Parameterization" on page 5-84
- "Estimating State-Space Models with Structured Parameterization" on page 5-85
- "Setting the Frequency-Weighing Focus" on page 5-91
- "Specifying the Initial States" on page 5-92

Definition of State-Space Models

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:

$$\begin{aligned}
\dot{x}(t) &= Fx(t) + Gu(t) + \tilde{K}w(t) \\
y(t) &= Hx(t) + Du(t) + w(t) \\
x(0) &= x0
\end{aligned}$$

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.

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.

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 as follows:

$$A = e^{FT}$$
$$B = \int_0^T e^{F\tau} G d\tau$$
$$C = H$$

These relationships assume that the input is piecewise-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 takes the input u to the output y and captures the system dynamics. H is an operator that describes the properties of the additive output noise model. Both G and H are called *transfer functions*.

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.

Supported State-Space Parameterizations

System Identification Toolbox supports the following parameterizations that indicate which parameters are estimated and which are set to specific values:

- Free parametrization 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 "Linear Grey-Box Models" on page 7-5.

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 in the MATLAB Command Window and import them into the System Identification Tool.

Estimating State-Space Model Orders

To estimate a state-space model, you must specify a model order and one or more input delays. 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.

To help you get an initial model order for your system, System Identification Toolbox lets you estimate a group of state-space models with a range of orders for a specific delay and compares 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.

This section discusses the following topics:

- "Estimating Orders in the GUI" on page 5-72
- "Estimating Orders in the MATLAB Command Window" on page 5-75
- "Using the Model Order Selection Plot" on page 5-75

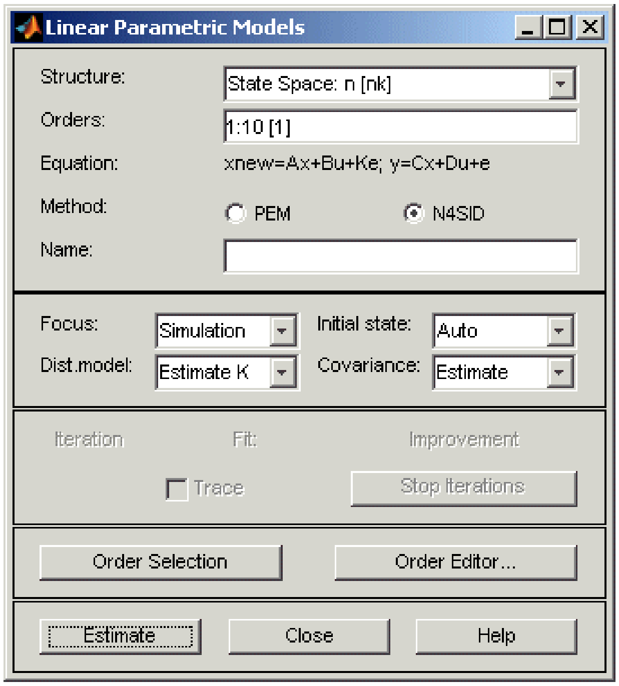
Estimating Orders in the GUI

The following procedure describes how to estimate model orders for a specific input delay in the System Identification Tool GUI and assumes that you already have the appropriate data in the Data Board.

1. In the System Identification Tool window, 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**.



Structure: State Space: n [nk]

Orders: 1:10 [1]

Equation: xnew=Ax+Bu+Ke; y=Cx+Du+e

Method: PEM [✓ N4SID]

Name:

Focus: Simulation

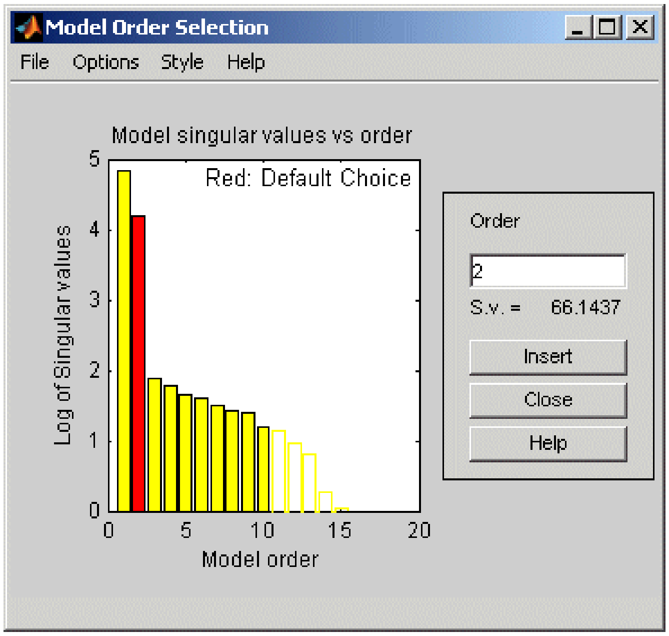
Initial state: Auto

Dist.model: Estimate K

Covariance: Estimate

4 Verify that the **Method** is set to **N4SID**.

5 Click **Estimate** to open the Model Order Selection plot, 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. The recommended choice is red. For information about using the Model Order Selection plot, see "Using the Model Order Selection Plot" on page 5-75.

This action adds a new model to the Model Board in the System Identification Tool window. The default name of the parametric model combines the string `n4s` and the selected model order.

In the previous figure, states 1 and 2 provide the most significant contribution. The contributions to the right of state 2 drop significantly.

7 Click **Close** to close the Model Order Selection plot.

After estimating model orders, use this values as an initial guess for estimating other state-space models, as described in "Estimating State-Space Models in the GUI" on page 5-77.

Estimating Orders in the MATLAB Command Window

You can estimate 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 plot. For information about using this plot, see "Using the Model Order Selection Plot" on page 5-75.

Alternatively, you can use the `pem` command to open the Model Order Selection plot, 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 plot and automatically select the best order, use the following syntax:

```
m = pem(Data,'best')
```

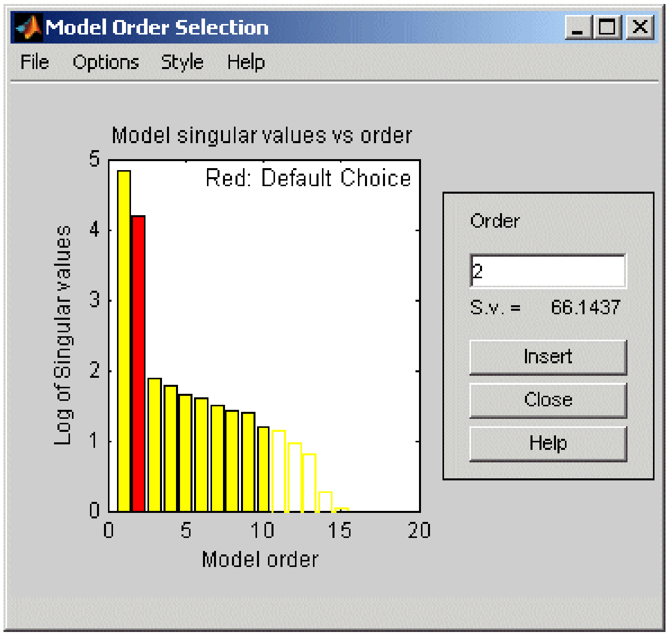
For a tutorial on estimating model orders for a multiple-input system, see "State-Space Model" in *Getting Started with System Identification Toolbox*.

Using the Model Order Selection Plot

You can generate the Model Order Selection plot 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 Orders in the GUI" on page 5-72. To open this plot in the MATLAB Command Window, see "Estimating Orders in the MATLAB Command Window" on page 5-75.

The following figure shows a sample Model Order Selection plot.



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.

Estimating State-Space Models in the GUI

The following procedure describes how to estimate a state-space model with free parameterization in the System Identification Tool GUI. It assumes that you already have the appropriate data in the Data Board.

Note Only free parameterization is directly supported in the System Identification Tool. You can estimate canonical and structured parameterizations in the MATLAB Command Window import them into the System Identification Tool for estimation.

This procedure also requires that you specify model order and any delays. For more information on how to estimate model orders, see "Estimating Orders in the GUI" on page 5-72.

1. In the System Identification Tool window, 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 "Definition of State-Space Models" on page 5-68.

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 in 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, see "Supported Estimation Algorithms" on page 1-16.

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 "Setting the Frequency-Weighing Focus" on page 5-91.

7. (If using PEM) In the **Initial state** list, specify how you want the algorithm to treat initial states. For more information about the available options, see "Specifying the Initial States" on page 5-92.

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. (If using PEM) 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 window.

11 (If using PEM) 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.

12 To plot the model, select the appropriate check box in the Model Views area of the System Identification Tool window. For more information about working with plots and validating models, see Chapter 9, "Plotting and Validating Models".

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 window. For more information about working with models, see Chapter 10, "Postprocessing and Using Estimated Models".

Using n4sid and pem to Estimate State-Space Models

You can estimate continuous-time and discrete-time polynomial model using the iterative estimation method `pem` that minimizes the prediction errors to obtain maximum-likelihood values. You can also use the noniterative subspace method `n4sid`. For more information about these methods, see "Supported Estimation Algorithms" on page 1-16.

You can only estimate discrete-time state-space models with free parameterization. Continuous state-space models are available for canonical and structured parameterizations.

The resulting models are stored as `idss` model objects.

You can also use `pem` to refine parameter estimates of an existing polynomial model, as described in “Refining Models” on page 1-46.

This section discusses the following topics:

- “General Syntax for n4sid and pem” on page 5-80
- “Common Parameters to Specify Model Estimation” on page 5-81
- “Estimating D, K, and X0 Matrices” on page 5-81

General Syntax for n4sid and pem

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 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.

For more information about these commands, see the corresponding references pages. For more information about estimating model order, see “Estimating Orders in the MATLAB Command Window” on page 5-75.

Common Parameters 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:
  - "Using n4sid and pem to Estimate Free-Parameterization State-Space Models" on page 5-83
  - "Estimating State-Space Models with Canonical Parameterization" on page 5-84
  - "Estimating State-Space Models with Structured Parameterization" on page 5-85
- `Focus` — Specifies the frequency-weighing of the noise model during estimation. See "Setting the Frequency-Weighing Focus" on page 5-91.
- `DisturbanceModel` — Specifies to estimate or omit the noise model for time-domain data. See "K Matrix" on page 5-82.
- `InitialStates` — Specifies to either set or estimate the initial states. See "Specifying the Initial States" on page 5-92

For more information about these properties, see the `idss` references pages.

Estimating 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 "Definition of State-Space Models" on page 5-68.

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'`, the *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 the Initial States” on page 5-92.

Using n4sid and pem 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)
```

Estimating State-Space Models with Canonical Parameterization

You can estimate state-space models with canonical parameterization in the MATLAB Command Window.

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*, System Identification Toolbox supports only controllable forms. Controllable canonical structures include free parameters in output rows of the A matrix, free B and K matrices, and 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, 1999 (equation 4A.16).

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 You estimate the D matrix in canonical form, you must set the `nk` property. See “Estimating D, K, and X0 Matrices” on page 5-81.

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 "Using n4sid and pem to Estimate Free-Parameterization State-Space Models" on page 5-83.

Estimating State-Space Models with Structured Parameterization

You can estimate state-space models with structured parameterization in the MATLAB Command Window. This might be simpler than estimating a grey-box model, as described in Chapter 7, "Estimating Grey-Box Models".

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 is different 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.

The following section introduce you the specifying the model structure and provide examples:

- "Specifying the State-Space Structure" on page 5-86

- "Example – Estimating Structured Discrete-Time State-Space Models" on page 5-88
- "Example – Estimating Structured Continuous-Time State-Space Models" on page 5-89

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 “Using n4sid and pem to Estimate State-Space Models” on page 5-79.

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 m-file. 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 m-file.

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, perform the following procedure:

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:

$$\dot{x} = \begin{bmatrix} 0 & 1 \\ 0 & \theta_1 \end{bmatrix} x(t) + \begin{bmatrix} 0 \\ \theta_2 \end{bmatrix} u(t)$$

$$y(t) = \begin{bmatrix} 1 & 0 \\ 0 & 1 \end{bmatrix} x(t) + e(t)$$

$$x(0) = \begin{bmatrix} \theta_3 \\ 0 \end{bmatrix}$$

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, 1999.

The continuous-time state-space model structure is defined by the following equation:

$$\begin{aligned}\dot{x}(t) &= Fx(t) + Gu(t) + \tilde{K}w(t)\\ y(t) &= Hx(t) + Du(t) + w(t)\\ x(0) &= x0\end{aligned}$$

To construct and estimate the parameters of this continuous-time state-space model, perform the following procedure:

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`, `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 the 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])
```

Setting the Frequency-Weighing Focus

You can specify how the estimation algorithm weighs the fit at various frequencies. This information supports the estimation procedures “Estimating

State-Space Models in the GUI" on page 5-77 and "Using n4sid and pem to Estimate State-Space Models" on page 5-79.

In the System Identification Tool GUI. Set the **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 on model stability, see "Unstable Models" on page 9-66.
- `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 4-36 or "Defining a Custom Filter" on page 4-36. This prefiltering applies only for estimating the dynamics from input to output. The disturbance model is determined from the estimation data.

In the MATLAB Command Window. Specify the focus as an argument in the estimation function using the same options as in the GUI. For example, use this command to emphasize the fit between the 5 and 8 rad/sec:

```
pem(data,4,'Focus',[5 8])
```

Specifying the Initial States

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 "Estimating State-Space Models in the GUI" on page 5-77 and "Using n4sid and pem to Estimate State-Space Models" on page 5-79.

In the System Identification Tool GUI. Set the **Initial state** to one of the following options:

- `Auto` — Automatically chooses one of the following 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.
- `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).

In the MATLAB Command Window. Specify the initial states as an argument in the estimation function `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 pages.

Time-Series Models

A *time series* is one or more measured output channels with no measured input.

System Identification Toolbox lets you estimate time-series spectra using both time- and frequency-domain data (`iddata` objects). 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.

Note ARMA and state-space models are supported for time-domain data only. Only single-output ARMA models are supported.

You can estimate the following types of model for time-series data:

- “Representing Time-Series Data for System Identification” on page 5-94
- “Estimating Spectral Models” on page 5-95
- “Estimating AR and ARMA Models” on page 5-97
- “Estimating State-Space Time-Series Models” on page 5-102
- “Example – Estimating Time Series” on page 5-103

Representing Time-Series Data for System Identification

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 System Identification Tool” on page 3-13.

In the MATLAB Command Window. To represent a time series vector or a matrix `s` as an `iddata` object, use the following syntax:

```
y = iddata(s,[],Ts);
```

where `Ts` is the sampling interval. For continuous-time frequency domain data, `Ts` is 0.

Estimating Spectral Models

This section describes the procedures required to estimate power spectra of time-series models in the System Identification Tool GUI and the MATLAB Command Window. It includes the following topics:

- "Estimating Spectral Models in the GUI" on page 5-95
- "Commands for Estimating Power Spectra" on page 5-96

Estimating Spectral Models in the GUI

The following procedure describes how to estimate time-series spectral models in the System Identification Tool and assumes that you already have the appropriate time-series data in the Data Board.

1. In the System Identification Tool window, 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 5-37.

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 "Specifying the Frequency Resolution" on page 5-38. 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 window.

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 window. For more information about working with this plot, see "Noise Spectrum Plots" on page 9-36.

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.

Commands for Estimating Power Spectra

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 function.

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.

For more information about models objects, see “Working with Model Objects” on page 1-19.

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 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 3 standard deviation confidence intervals:

```
g = spa(y)
p = etfe(y)
bode(g,p,'sd',3)
```

For detailed information about these functions, see the corresponding reference pages.

Estimating AR and ARMA Models

This section describes the procedures required to estimate AR and ARMA time-series models in the System Identification Tool GUI and the MATLAB Command Window. It includes the following topics:

- “Definition of AR and ARMA Models” on page 5-97
- “Estimating Polynomial Models in the GUI” on page 5-98
- “Estimating AR and ARMA Models in the MATLAB Command Window” on page 5-101

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 “Definition of Polynomial Models” on page 5-43.

Estimating Polynomial Models in the GUI

The following procedure describes how to estimate AR and ARMA models in the System Identification Tool and assumes that you already have the appropriate data in the Data Board.

This procedure also requires that you select a model structure and specify model orders and delays. For more information on how to estimate model orders and delays, see “Estimating Orders and Delays in the GUI” on page 5-50.

If you are estimating a multiple-output AR model, you must specify order matrix in the MATLAB workspace before estimation, as described in “Specifying Multiple-Input and Multiple-Output ARX Orders” on page 5-57.

1 In the System Identification Tool window, 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 5-97.

   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 "Specifying Multiple-Input and Multiple-Output ARX Orders" on page 5-57. Alternatively, enter the name of the matrix `NA` in the MATLAB workspace 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 (For AR models only) Select the estimation **Method** as **ARX** or **IV** (instrumental variable method). For more information about these methods, see "Supported Estimation Algorithms" on page 1-16.

   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 the Initial States" on page 5-21.

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 (For ARMA 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.

9 Click **Estimate** to add this model to the Model Board in the System Identification Tool window.

10 (For 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 window. For more information about

working with plots and validating models, see Chapter 9, "Plotting and Validating Models".

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 window. For more information about working with models, see Chapter 10, "Postprocessing and Using Estimated Models".

Estimating AR and ARMA Models in the MATLAB Command Window

You can estimate AR and ARMA models in the MATLAB Command Window. 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 "Working with Model Objects" on page 1-19.

The following table provides a brief description of each command 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 options:

- Burg's method—A geometric lattice method.
- Yule-Walker approach.
- Covariance method.

Estimating State-Space Time-Series Models

This section describes the procedures required to estimate state-space time-series models in the System Identification Tool GUI and the MATLAB Command Window. It includes the following topics:

- "Definition of State-Space Time-Series Model" on page 5-102
- "Estimating State-Space Models in the MATLAB Command Window" on page 5-103

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 "Definition of State-Space Models" on page 5-68.

Estimating State-Space Models in the MATLAB Command Window

You can estimate single-output and multiple-output state-space models in the MATLAB Command Window for time-domain and frequency-domain data (`iddata` object).

The following table provides a brief description of each command. For more information about each command, see the corresponding references pages.

The resulting models are `idss` model objects. For more information about models objects, see "Working with Model Objects" on page 1-19.

Commands for Estimating State-Space Time-Series Models

Method Name	Description
`n4sid`	Noniterative, subspace estimation 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 iterative estimation method that minimizes the prediction error.

Example – Estimating Time Series

Here is an example where you can simulate a time series, compare spectral estimates and covariance function estimates, and also the predictions 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 Black-Box Models

Overview of Nonlinear Black-Box Modeling

System Identification Toolbox lets you estimate discrete-time nonlinear black-box models for single-output or multiple-output time-domain data. It supports the following types of nonlinear black-box models:

- Nonlinear ARX models.

  For an example of estimating this type of model, see "Example – Estimating Nonlinear ARX Model for a Two-Tank System" on page 6-15.

- Hammerstein-Wiener models.

  For an example of estimating this type of model, see "Example – Estimating Hammerstein-Wiener Model for a Two-Tank System" on page 6-44.

You can estimate these models both in the System Identification Tool GUI and in the MATLAB Command Window.

If you are working in the MATLAB Command Window, use the `nlarx` and `nlhw` commands to construct and estimate the nonlinear ARX and Hammerstein-Wiener models, respectively. Nonlinear ARX models are `idnlarx` model objects, and Hammerstein-Wiener models are `idnlhw` model objects. For detailed information about these commands and objects, see the corresponding reference pages. For general information on working with model objects, see "Working with Model Objects" on page 1-19.

This section discusses the following topics:

- "Before You Begin" on page 6-2
- "Using Nonlinear Black-Box Models" on page 6-3

Before You Begin

You can estimate discrete-time black-box models for data with the following characteristics:

- Time-domain input-output or time-series data.

Note Time series are supported for nonlinear ARX models only.

- Single-output or multiple-output data.

Before you begin estimating models, import your data into MATLAB, and represent the data in one of the following ways:

- **In the System Identification Tool GUI.** Import the data into the GUI to make the data available to System Identification Toolbox.
- **In the MATLAB Command Window.** Represent your data as an `iddata` or `idfrd` object.

For more information about representing data for system identification, see Chapter 3, "Representing Data for System Identification".

To examine the data features, plot the data on a time plot or an estimated frequency-response plot. You can preprocess your data by interpolating missing values, filtering to emphasize a specific frequency range, or resampling using a different time interval.

Note For nonlinear modeling, do not remove offsets and linear trends from the measured signals.

For more information about types of available date plots and data-preprocessing operations, see Chapter 4, "Plotting and Preprocessing Data".

Using Nonlinear Black-Box Models

After estimating both nonlinear black-box models, you can use `present` to view parameter values, standard deviations of the parameters, loss function, and Akaike's Final Prediction Error (FPE) Criterion in the MATLAB Command Window.

You can use the `sim` command to simulate the model, or the `predict` command to predict the model output. For more information about simulation and prediction, see "Simulating and Predicting Model Output" on page 10-13.

You can linearize nonlinear ARX and Hammerstein-Wiener models using `lintan` or `linapp`. `lintan` provides a small-signal tangent linearization about a specific operating point. `linapp` computes a linear approximation for a nonlinear model of a given input. For more information about these functions, see the corresponding reference pages.

After linearization, you can perform linear analysis on your models and use the models with Control System Toolbox. For more information, see "Using Models with Control System Toolbox" on page 10-20.

Estimating Nonlinear ARX Models

You can estimate both continuous-time and discrete-time nonlinear ARX models for data with the following characteristics:

- Time-domain input-output data or time-series data.
- Single-output or multiple-output data.

For more information about representing your data for system identification, see Chapter 3, "Representing Data for System Identification".

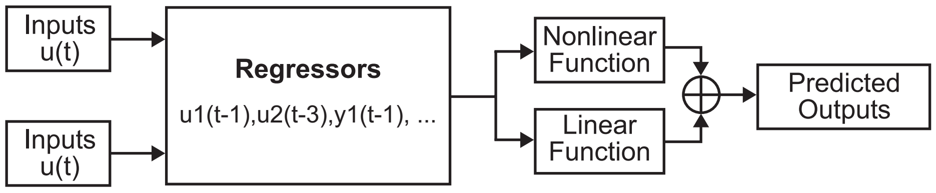
This section discusses the following topics:

- "Definition of the Nonlinear ARX Model" on page 6-5
- "Using Regressors" on page 6-6
- "Nonlinearity Estimators for Nonlinear ARX Models" on page 6-9
- "Estimating Nonlinear ARX Models in the GUI" on page 6-9
- "Using nlarx to Estimate Nonlinear ARX Models" on page 6-11

For an example of estimating a nonlinear ARX model using the System Identification Tool GUI, see "Example – Estimating Nonlinear ARX Model for a Two-Tank System" on page 6-15.

Definition of the Nonlinear ARX Model

The nonlinear ARX structure models dynamic systems using a parallel combination of nonlinear and linear blocks, as shown in the following figure.



The nonlinear and linear functions are expressed in terms of variables called *regressors*, which are functions of measured input-output data. For more information about regressors, see “Using Regressors” on page 6-6.

The predicted output $\hat{y}(t)$ of a nonlinear model at time *t* is given by the following general equation:

$$\hat{y}(t) = F(x(t))$$

where *x(t)* represents the regressors. *F* is a nonlinear regression function, which is approximated by the nonlinearity estimators. For a list of nonlinearity estimators supported by nonlinear ARX models, see “Nonlinearity Estimators for Nonlinear ARX Models” on page 6-9.

The function *F* can include both linear and nonlinear functions of *x(t)*, as shown in the previous diagram. You can specify which regressors to use a inputs to the nonlinear block.

The following equation provides a general description of *F*:

$$F(x) = \sum_{k=1}^{d} \alpha_k \kappa\left(\beta_k \left(x - \gamma_k\right)\right)$$

where κ is the unit nonlinear function, *d* is the number of nonlinearity units, and α_k , β_k , and γ_k are the parameters of the nonlinearity estimator.

Using Regressors

System Identification Toolbox supports the following types of regressors for nonlinear ARX models:

- *Standard regressor*s — Past input *u(t)* and output signals *y(t)*, computed automatically as delay transformations for specified model orders.
- *Custom regressors* — Products, powers, and other MATLAB expressions of input and output variables that you specify.

Specifying Model Order and Delays

You must specify the following model orders for computing standard regressors:

- n_a — The number of past output terms used to predict the current output.
- n_b — The number of past input terms used to predict the current output.
- n_k — The delay from input to the output in terms of the number of samples. This value defines the least delayed input regressor.

The meaning of n_a and n_b is similar to the linear-ARX model parameters in the sense that n_a represents the number of output terms and n_b represents the number of input terms. n_k represents the minimum input delay from an input to an output. For more information about the linear ARX model structure, see "Definition of Polynomial Models" on page 5-43.

Note The total number of regressors in the model must be greater than zero. If you only need to use custom regressors, set n_a=n_b=n_k=0 to omit creating standard regressors.

Example – Types of Regressors Computed from Model Orders and Delays

This example describes the regressors computed by System Identification Toolbox based on specified model orders and delays.

Suppose that you specify a nonlinear ARX model with a minimum of a two-sample input delay and the number of input terms is n_b=2. System Identification Toolbox computes the following standard regressors from the input signal:

- *u(t-2)*
- *u(t-3)*

If you specify that the number of output terms as n_a=4, System Identification Toolbox computes the following standard regressors from the output signals:

- *y(t-1)*
- *y(t-2)*
- *y(t-3)*
- *y(t-4)*

Note You cannot modify the minimum output delay—it is set to 1 sample.

If you have physical insight that your current output depends on specific delayed inputs and outputs, select the appropriate model orders to compute the required regressors.

Using Custom Regressors

In general, custom regressors are nonlinear functions of the standard regressors. You can specify custom regressors, such as *tan(u(t-1))*, *u(t-1)*^{2}, or *u(t-1)y(t-3)*.

In the System Identification Tool GUI. You can create custom regressors in the Model Regressors dialog. For more information, see “Estimating Nonlinear ARX Models in the GUI” on page 6-9.

In the MATLAB Command Window. Use the `customreg` or `polyreg` commands to construct custom regressors in terms of input-output variables. For more information, see the corresponding reference pages.

The linear block includes all standard and custom regressors. However, you can include specific standard and custom regressors in your nonlinear block to fine-tune the model structure.

To get a linear-in-the-parameters ARX model structure, you can exclude the nonlinear block from the model structure completely. When using only a linear block with custom regressors, you can create the simplest types of nonlinear models. In this case, the custom regressors capture the nonlinearities and the estimation routine computes the weights of the standard and custom regressors in the linear block to predict the output.

Nonlinearity Estimators for Nonlinear ARX Models

Nonlinear ARX models support the following nonlinearity estimators:

- Sigmoid Network
- Tree Partition
- Wavelet Network
- Custom Network
- Neural Network

Note You must have Neural Network Toolbox to specify neural network objects as nonlinearities.

You can exclude the nonlinearity function from the model structure. In this case, the model includes all standard and custom regressors and is linear in the parameters.

In the System Identification Tool GUI. You can omit the nonlinear block by selecting `None` for the **Nonlinearity**.

In the MATLAB Workspace. You can omit the nonlinear block by setting the `Nonlinearity` property value to `'Linear'`. For more information, see the `nlarx` and `idnlarx` reference pages

For a description of each nonlinearity estimator, see "Supported Nonlinearity Estimators" on page 6-60.

Estimating Nonlinear ARX Models in the GUI

The following procedure describes how to estimate a nonlinear ARX model in the System Identification Tool GUI and assumes that you already have the appropriate data in the Data Board. For more information about preparing your data, see "Overview of Nonlinear Black-Box Modeling" on page 6-2.

1 In the System Identification Tool window, select **Estimate > Nonlinear models** to open the Nonlinear Models dialog box. The **Model Type** tab is selected.

2 In the **Model Structure** list, select `Nonlinear ARX`.

   This action updates the options in the Nonlinear Models dialog box to correspond to this model structure. For information about this model structure, see "Definition of the Nonlinear ARX Model" on page 6-5.

3 In the **Model name** field, edit the name of the model, or keep the default name. The name of the model should be unique in the Model Board.

4 (Optional) If you want to try refining a previously estimated model, select the name of this model in the **Initial model** list.

   Note The model structure and algorithm properties of the initial model populate the fields in the Nonlinear Models dialog box.

   A model is available in the **Initial model** list under the following conditions:

   - The model exists in the System Identification Tool window.
   - The number of model inputs and outputs matches the dimensions of the **Working Data** (estimation data) you selected in the System Identification Tool window.

5 Keep the default settings in the Nonlinear Models dialog box that specify the model structure and the algorithm, or modify the following settings:

   - In the **Regressors** tab, change the input delay of the input signals.

     To gain insight into possible input delay values, click **Infer Input Delay**. This action opens the Infer Input Delay dialog box.

   - In the **Regressors** tab, change the number of terms to include in the nonlinear block.
   - In the **Regressors** tab, click **Edit Regressors** to select which regressors are included in the nonlinear block. This action opens the Model Regressors dialog box. You can also use this dialog box create custom regressors.
   - In the **Model Properties** tab, select and configure the nonlinearity estimator, and choose whether to include the linear block. To use all

standard and custom regressors in the linear block only, you can exclude the nonlinear block by choosing `None`.

For more information about the available options, click **Help** in the Nonlinear Models dialog box to open the GUI help.

6 Click **Estimate** to add this model to the Model Board in the System Identification Tool window.

The **Estimation** tab displays the estimation progress and results.

7 To plot this model, select the appropriate check box in the Model Views area of the System Identification Tool window. For more information about working with plots and validating models, see Chapter 9, “Plotting and Validating Models”.

If you get an inaccurate fit, try estimating a new model with different orders or nonlinearity estimator.

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 window. For more information about working with models, see Chapter 10, “Postprocessing and Using Estimated Models”.

Using nlarx to Estimate Nonlinear ARX Models

You can estimate nonlinear ARX models using `nlarx`. The resulting models are stored as `idnlarx` model objects.

You can also use `pem` to refine parameter estimates of an existing nonlinear ARX model, as described in “Refining Models” on page 1-46.

This section discusses the following topics:

- “General nlarx Syntax” on page 6-12
- “Example – Using nlarx to Estimate Nonlinear ARX Models ” on page 6-13

General nlarx Syntax

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

```
m = nlarx(data,'na',na,
               'nb',nb,
               'nk',nk,
                Nonlinearity,
               'Property1',Value1,...,
               'PropertyN',ValueN)
```

where `data` is the estimation data. `na`, `nb`, and `nk` specify the model orders and delays. For more information about model orders, see "Specifying Model Order and Delays" on page 6-7.

`Nonlinearity` specifies the nonlinearity estimator object as `'sigmoidnet'`, `'wavenet'`, `'treepartition'`, `'customnet'`, `'neuralnet'`, or `'linear'`.

The property-value pairs specify any `idnlarx` model properties that configure the estimation algorithm. You can enter all model property-value pairs and top-level algorithm properties as a comma-separated list in `nlarx`.

For multiple inputs and outputs, `na`, `nb`, and `nk` are described in "Specifying Multiple-Input and Multiple-Output ARX Orders" on page 5-57.

You can specify different nonlinearity estimators for different output channels by setting `Nonlinearity` to an object array. For example:

```
m = nlarx(data,[[2 1; 0 1] [2;1] [1;1]],...
               [wavenet;sigmoidnet('num',7)])
```

To specify the same nonlinearity for all outputs, set `Nonlinearity` to a single nonlinearity estimator. For example:

```
m = nlarx(data,[[2 1; 0 1] [2;1] [1;1]],...
               sigmoidnet('num',7))
```

For detailed information about the `nlarx` and `idnlarx` properties and values, see the corresponding reference pages.

For more information about validating your models, see Chapter 9, “Plotting and Validating Models”. To learn more about simulation and prediction output using your models, see Chapter 10, “Postprocessing and Using Estimated Models”.

Note You do not need to construct the model object using `idnlarx` before estimation.

Example – Using nlarx to Estimate Nonlinear ARX Models

This example uses `nlarx` to estimate a nonlinear ARX model for the two-tank system, as described in “Example – Estimating Nonlinear ARX Model for a Two-Tank System” on page 6-15.

Prepare the data for estimation using the following commands:

```
load twotankdata
z = iddata(y, u, 0.2);
ze = z(1:1000); zv = z(1001:3000);
```

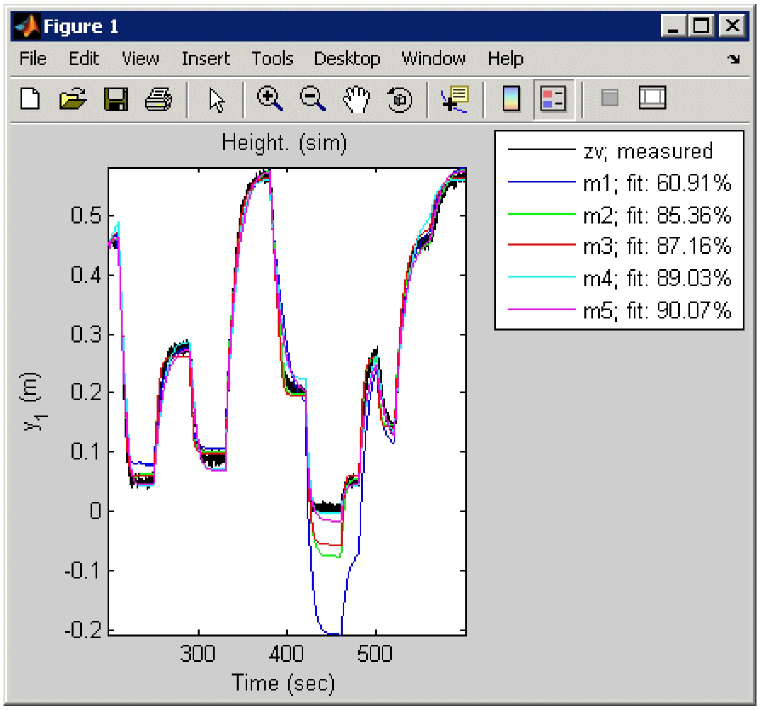
Estimate several models using different model orders, delays, and nonlinearity settings:

```
m1 = nlarx(ze,[2 2 1],'wav');
m2 = nlarx(ze,[2 2 3],wavenet);
m3 = nlarx(ze,[2 2 3],wavenet('num',8));
m4 = nlarx(ze,[2 2 3],wavenet('num',8),...
                      'nlr', [1 2]);
m5 = nlarx(ze,[2 2 3],sigmoidnet('num',14),...
                      'nlr',[1 2]);
```

Compare the resulting models by plotting the model outputs on top of the measured output:

```
compare(zv, m1,m2,m3,m4,m5)
```

MATLAB responds with the following plot:



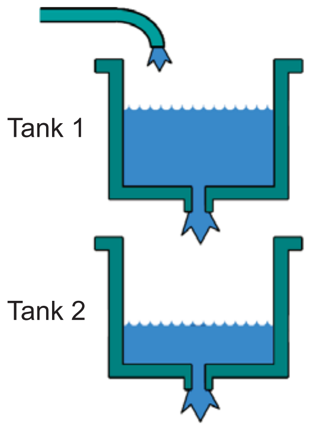
Example – Estimating Nonlinear ARX Model for a Two-Tank System

This example discusses the following topics:

About This Example

By performing the steps in this example, you get an overview of how to estimate nonlinear ARX models using the System Identification Tool GUI.

In this example, you estimate a nonlinear ARX model to fit measured single-input and single-output (SISO) data for a two-tank system, shown in the following figure.



Two-Tank System

In the two-tank system, water pours through a pipe into Tank 1, drains into Tank 2, and leaves the system through a small hole at the bottom of Tank 2. The measured input *u(t)* to the system is the voltage applied to the pump that feeds the water into Tank 1 (in volts). The measured output *y(t)* is the height of the water in the lower tank (in meters).

Based on Bernoulli's law—which states that water flowing through a small hole at the bottom of a tank depends nonlinearly on the level of the water in the tank—you expect the relationship between the input and the output data to be nonlinear.

The sample data is provided in `twotankdata.mat`, which you install with the latest version of System Identification Toolbox. This MAT-file contains SISO time-domain data of 3000 samples with a sampling interval of 0.2 sec.

Before You Begin

Before you can perform the tasks described in this tutorial, you must do the following preparation:

- "Loading Data into the MATLAB Workspace" on page 6-17

- “Creating iddata Objects” on page 6-17
- “Starting the System Identification Tool” on page 6-19
- “Loading Data into the System Identification Tool” on page 6-20

Loading Data into the MATLAB Workspace

Load the sample data in `twotankdata.mat` by typing the following command at the MATLAB prompt:

```
load twotankdata.mat
```

This command loads the following two variables into MATLAB workspace:

- `y` is the output data.
- `u` is the input data.

The input data is the voltage applied to the pump that feeds the water into Tank 1 (in volts), and the output is the water height in Tank 2 (in meters).

Creating iddata Objects

The `iddata` constructor requires three arguments for time-domain data and has the following syntax:

```
data_obj = iddata(output,input,sampling_interval);
```

Use these commands to create two data objects, `ze` and `zv`:

```
Ts = 0.2; % Sampling interval is 0.5 min
z = iddata(y,u,Ts);
% First 1000 samples used for estimation
ze = z(1:1000);
% Remaining samples used for validation
zv = z(1001:3000);
```

`ze` contains data for model estimation and `zv` contains data for model validation. `Ts` is the sampling interval.

To view the properties of an `iddata` object, use the `get` command. For example, type this command to get the properties of the estimation data:

```
get(ze)
```

MATLAB returns the following data 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: 0.2000
              Tstart: 0.2000
    SamplingInstants: [1000x0 double]
            TimeUnit: ''
      ExperimentName: 'Exp1'
               Notes: []
            UserData: []
```

To modify properties, use dot notation or the `set` command. For example, to assign channel names and units that label plot axes, type the following syntax at the MATLAB prompt:

```
% Set time units to minutes
ze.TimeUnit = 'sec';
% Set names of input channels
ze.InputName = 'Voltage';
% Set units for input variables
ze.InputUnits = 'V';
% Set name of output channel
ze.OutputName = 'Height';
% Set unit of output channel
ze.OutputUnits = 'm';

% Set validation data properties
zv.TimeUnit = 'sec';
zv.InputName = 'Voltage';
zv.InputUnits = 'V';
zv.OutputName = 'Height';
zv.OutputUnits = 'm';
```

You can verify that the `InputName` property of `ze` is changed, or "index into" this property, by typing the following syntax:

```
ze.inputname
```

Note Property names are not case sensitive.

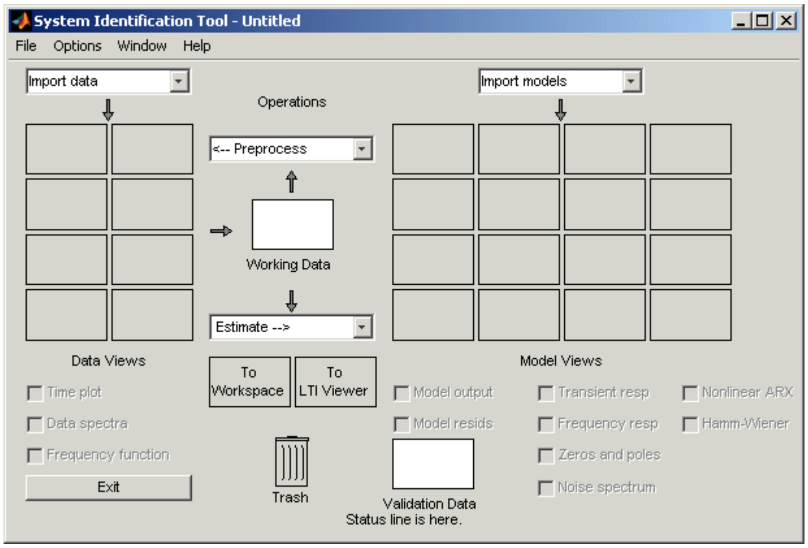
For detailed information about `iddata` objects, see "Creating iddata Objects" on page 3-31.

Starting the System Identification Tool

To open the System Identification Tool GUI, type the following command at the MATLAB prompt:

```
ident
```

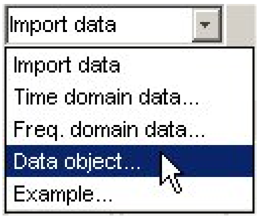
The default session name, `Untitled`, displays in the title bar.



Loading Data into the System Identification Tool

To import the data object you created in “Creating iddata Objects” on page 6-17, perform the following procedure:

1 In the System Identification Tool window, select **Import data > Data object** to open the Import Data dialog box:



2 In the Import Data dialog box, type `ze` in the **Object** field to import the estimation data. Press **Enter**. This action enters the object information into the fields.

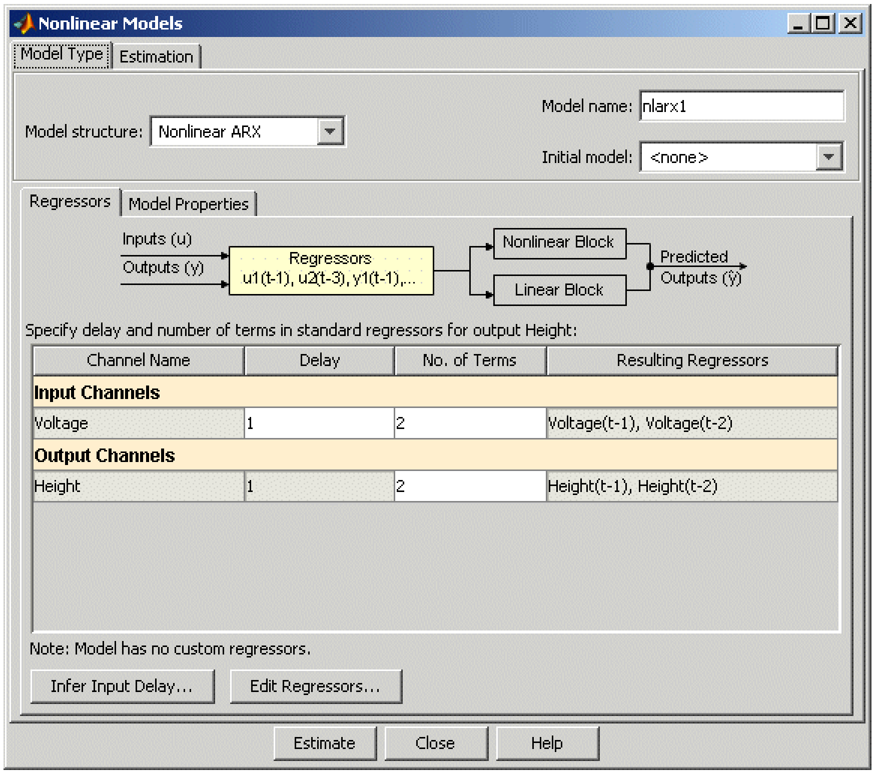
Click **More** to view the following additional information about this data, including channel names and units.

3. Click **Import** to add the icon named `ze` to the Data Board.
4. In the Import Data dialog box, type `zv` in the **Object** field to import the validation data. Press **Enter**.
5. Click **Import** to add the icon named `zv` to the Data Board.
6. In the Import Data dialog box, click **Close**.
7. In the System Identification Tool window, drag the **ze** icon to the **Working Data** rectangle, and then drag the **zv** icon to the **Validation Data** rectangle.

Estimating Nonlinear ARX Model with Default Settings

After preparing the data, as described in "Before You Begin" on page 6-16, you are ready to estimate the nonlinear ARX model.

1 In the System Identification Tool window, select **Estimate > Nonlinear models** to open the Nonlinear Models dialog box:



Model structure: Nonlinear ARX

Model name: nlarx1

Initial model: <none>

Specify delay and number of terms in standard regressors for output Height:

Channel Name	Delay	No. of Terms	Resulting Regressors
Input Channels			
Voltage	1	2	Voltage(t-1), Voltage(t-2)
Output Channels			
Height	1	2	Height(t-1), Height(t-2)

Note: Model has no custom regressors.

The **Model Type** tab is already selected and the default **Model Structure** is `Nonlinear ARX`.

In the **Regressors** tab, the model orders for both **Input Channels** and **Output Channels** are specified by the **Delay** of `1` and **No. of Terms** equal to `2`. Thus, the model output *y(t)* it related to the input *u(t)* via the following nonlinear autoregressive equation:

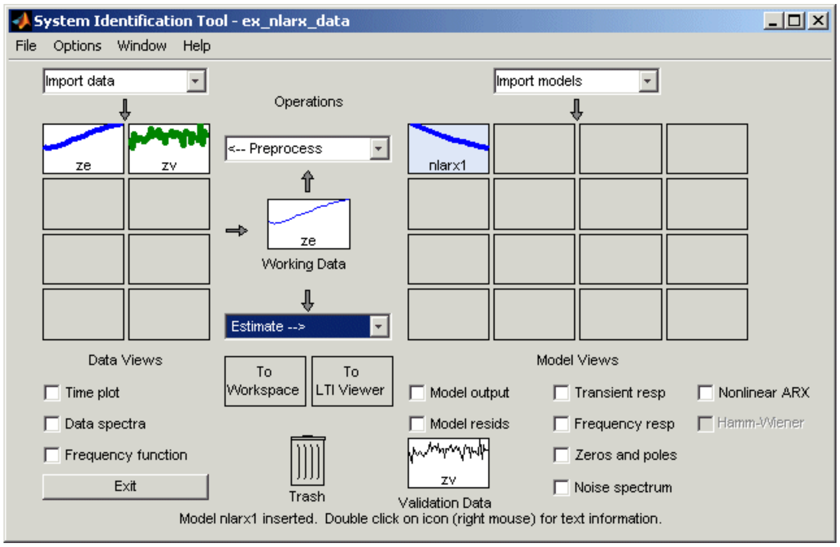
$$y(t) = f\left(y(t-1), y(t-2), u(t-1), u(t-2)\right)$$

where *f* is the nonlinearity estimator you select in the **Model Properties** tab.

2 Select the **Model Properties** tab.

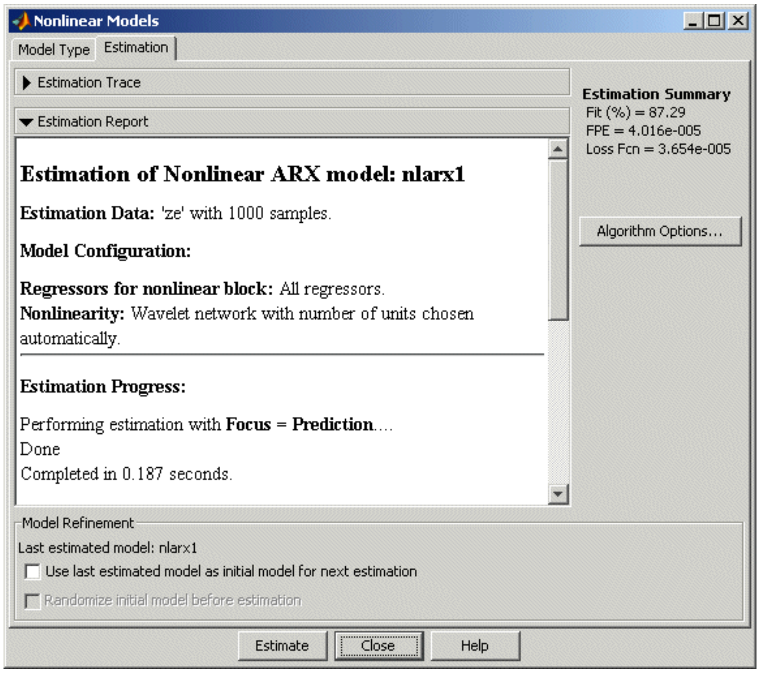
The **Nonlinearity** represents the nonlinear function *f* and is already set to `Wavelet Network`, by default. The number of units for the nonlinearity estimator is set to **Select automatically**, which lets the algorithm search for the best number of units during estimation.

3 Click **Estimate**. This selection adds the model `nlarx1` to the Model Board in the System Identification Tool window, as shown in the following figure.



Model nlarx1 inserted. Double click on icon (right mouse) for text information.

The Nonlinear Models dialog box displays the following estimation information in the **Estimation** tab:



Estimation Summary
Fit (%) = 87.29
FPE = 4.016e-005
Loss Fcn = 3.654e-005

Estimation of Nonlinear ARX model: nlarx1

Estimation Data: 'ze' with 1000 samples.

Model Configuration:

Regressors for nonlinear block: All regressors.
Nonlinearity: Wavelet network with number of units chosen automatically.

Estimation Progress:

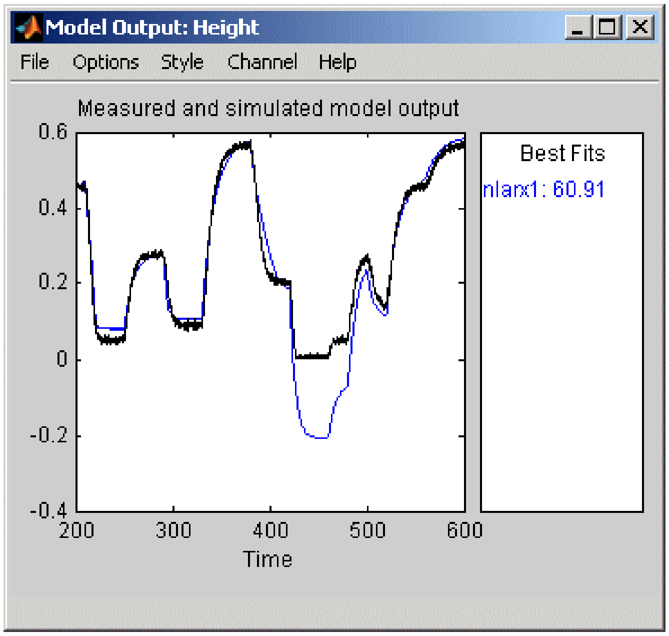
Performing estimation with **Focus** = **Prediction**....
Done
Completed in 0.187 seconds.

Model Refinement
Last estimated model: nlarx1
☐ Use last estimated model as initial model for next estimation
☐ Randomize initial model before estimation

Note The **Fit (%)** is computed using the estimation data set, and not the validation data set. However, the model output plot show the fit for the validation data set.

---

4 In the System Identification Tool window, select the **Model output** check box. Simulation of the model output uses the input validation data as input to the model. It plots the simulated output on top of the output validation data.



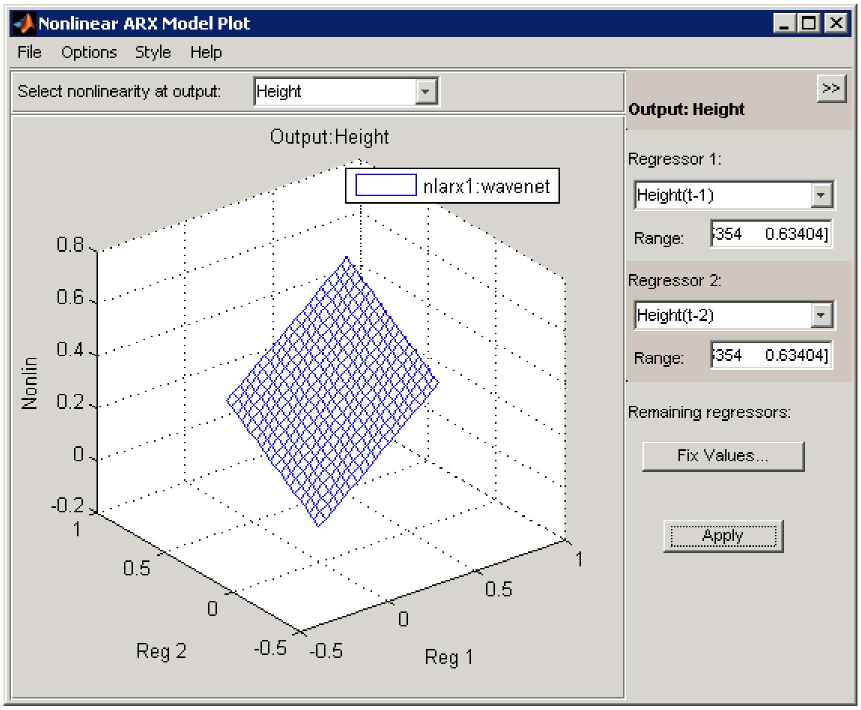
The **Best Fits** area of the Model Output plot shows that the agreement between the model output and the validation-data output is 60.91%.

Plotting Nonlinearity Cross-Sections

Perform the following procedure to view the shape of the nonlinearity as a function of regressors on a Nonlinear ARX Model plot. For more information about working with Nonlinear ARX Model plot, see "Nonlinear ARX Plots" on page 9-48.

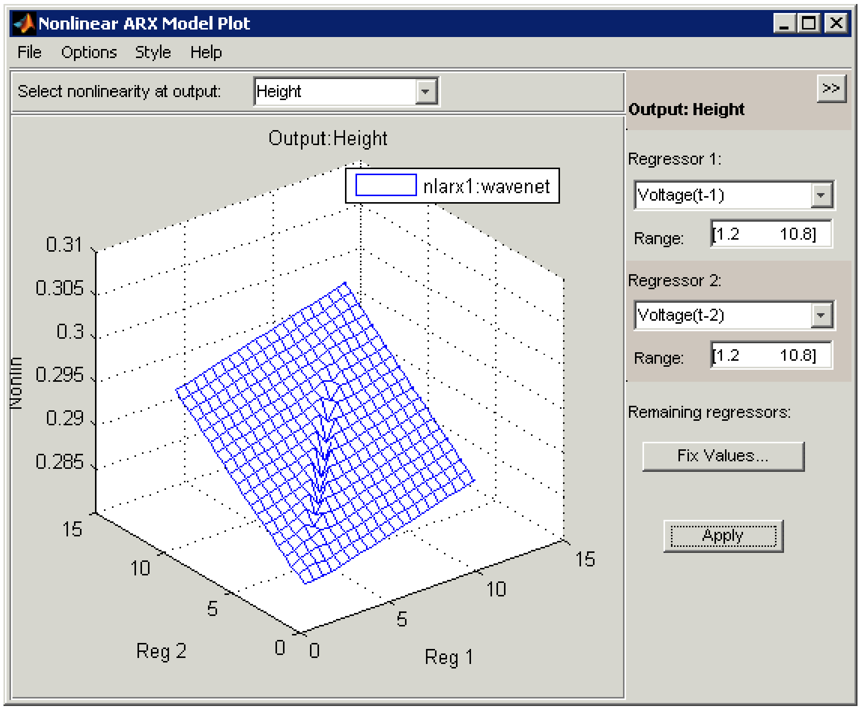
1 In the System Identification Tool window, select the **Nonlinear ARX** check box to view the nonlinearity cross-sections.

By default, the plot shows the relationship between the output regressors `Height(t-1)` and `Height(t-2)`. This plot shows a regular plane in the following figure. Thus, the relationship is approximately linear plane.



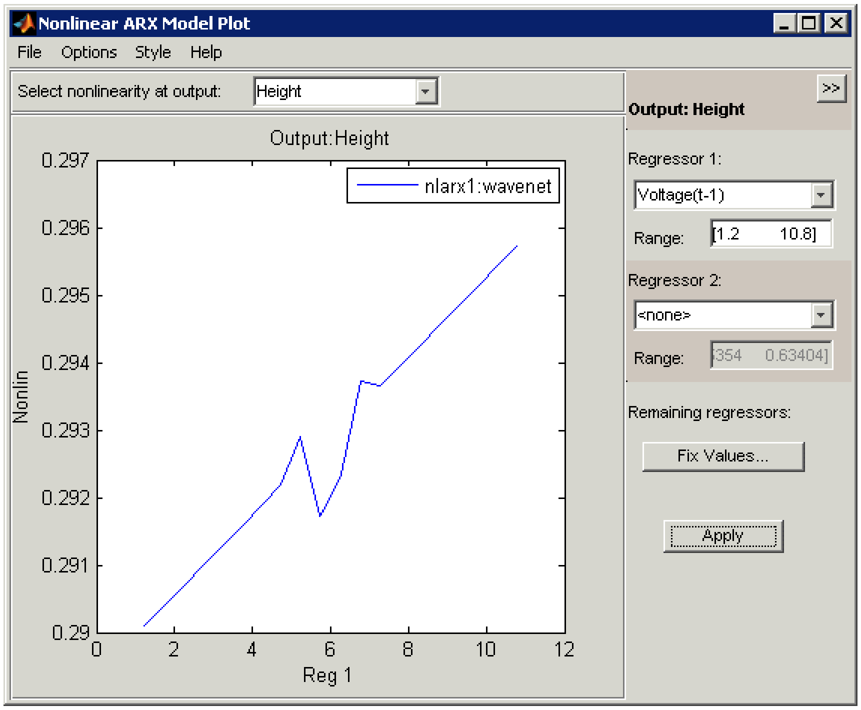
2 In the Nonlinear ARX Model Plot window, keep the default value for **Regressor 1** at `Voltage(t-1)`. Set **Regressor 2** to `Voltage(t-2)`. Click **Apply**.

The relationship between these regressors is nonlinear, as shown in the following plot.



3 To rotate the nonlinearity surface, select **Style > 3D Rotate** and drag the plot to a new orientation.

4 To display a 1–D cross-section for Regressor 1, set Regressor 2 to `none`, and click **Apply**. The following figure shows the resulting nonlinearity magnitude for Regressor 1, which is `Voltage(t-1)`.



Changing the Model Structure

After estimating the nonlinear ARX model with default settings, as described in “Estimating Nonlinear ARX Model with Default Settings” on page 6-21, you can try to improve the fit by modifying the input delay and the nonlinearity properties. Typically, you select model orders and delays by trial and error.

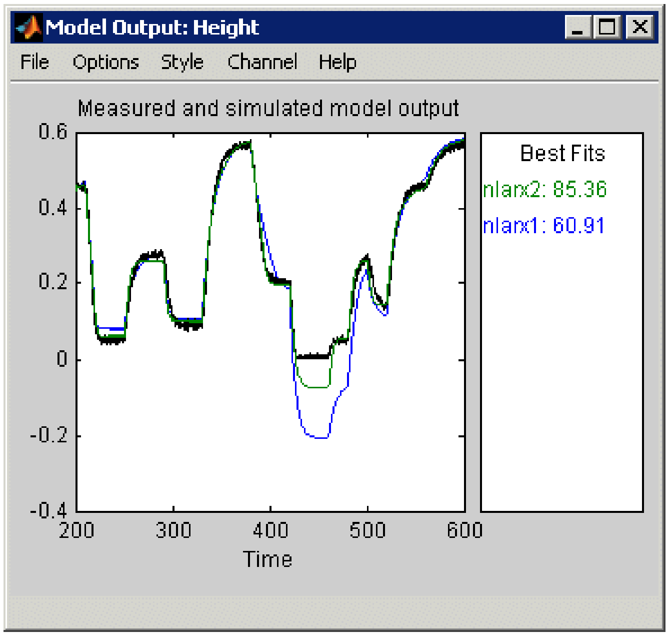
1 In the Nonlinear Models dialog box, select the **Model Type** tab, and then select the **Regressors** tab.

2 For the `Voltage` input channel, double-click the corresponding **Delay** cell, type `3`, and press **Enter**.

This action updates the **Resulting Regressors** list. The list now includes `Voltage(t-3)` and `Voltage(t-4)`—two terms with a minimum input delay of 3 samples.

3 Click **Estimate**.

This action adds the model `nlarx2` to the Model Board in the System Identification Tool window and the Model Output plot is updated to include this model. The Nonlinear Models dialog box displays the new estimation information in the **Estimation** tab.



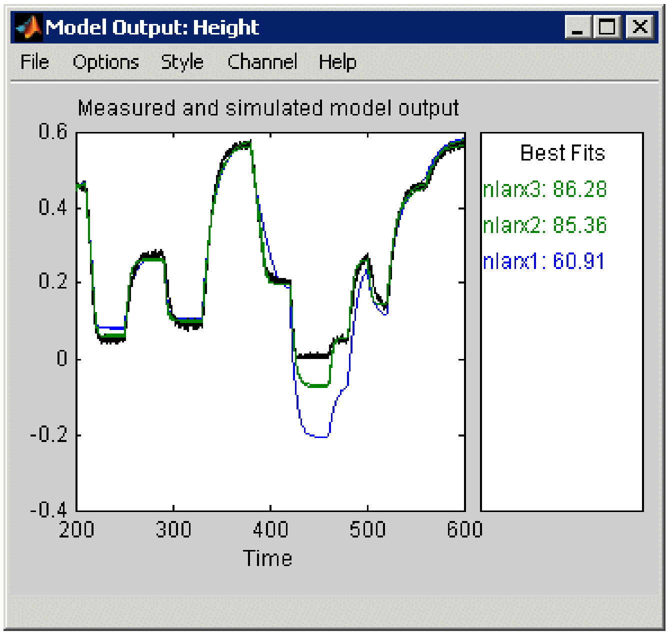
The **Best Fits** area of the Model Output plot shows that `nlarx2` fit is 85.36%.

4 In the Nonlinear Models dialog box, select the **Model Properties** tab.

5 For **Number of units in nonlinear block**, select **Enter**, and type `6`.

6 Click **Estimate**.

This action adds the model `nlarx3` to the Model Board in the System Identification Tool window. It also updates the Model Output plot, as shown in the following figure.



The **Best Fits** area of the Model Output plot shows that the `nlarx3` fit is 86.28%.

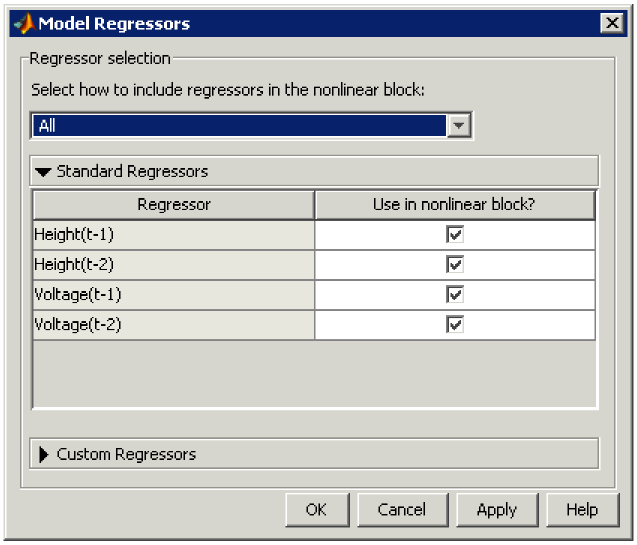
Selecting a Subset of Regressors in the Nonlinear Block

By default, all standard and custom regressors are used in the nonlinear block. In this example, you only have standard regressors. For more information about regressors, see “Using Regressors” on page 6-6.

After improving the model accuracy by changing the model structure, as described in “Changing the Model Structure” on page 6-28, you can try to improve the fit by selecting a subset of standard regressors that enter as inputs to the nonlinear block.

1 In the Nonlinear Models dialog box, select the **Model Type** tab, and then select the **Regressors** tab.

2 Click **Edit Regressors**. This action opens the Model Regressors dialog box.



Select how to include regressors in the nonlinear block:

All

Standard Regressors

Regressor	Use in nonlinear block?
Height(t-1)	✓
Height(t-2)	✓
Voltage(t-1)	✓
Voltage(t-2)	✓

3 In the Model Regressors dialog box, clear the following check boxes:

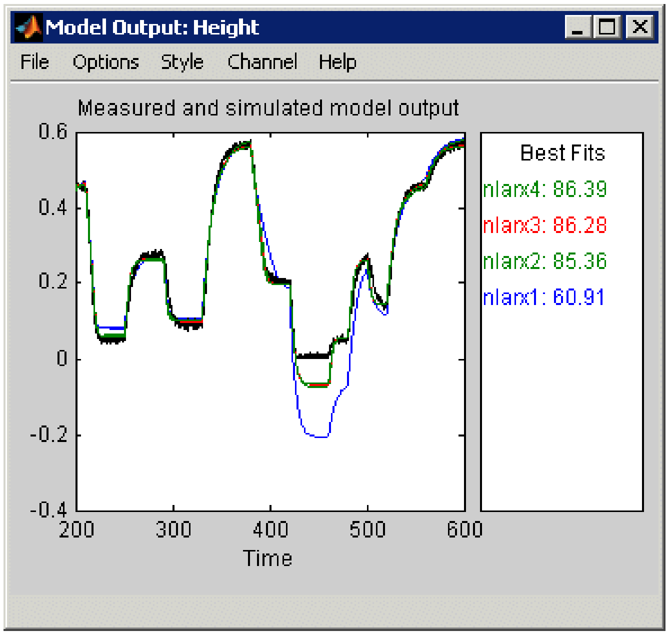
- `Height(t-2)`
- `Voltage(t-1)`

Click **OK**.

This action excludes `Height(t-2)` and `Voltage(t-1)` from the list of inputs to the nonlinear block.

4 Click **Estimate**.

This action adds the model `nlarx4` to the Model Board in the System Identification Tool window. It also updates the Model Output plot, as shown in the following figure.



The **Best Fits** area of the Model Output plot shows that the `nlarx4` fit is 86.39%, which is only a fraction of a percent improvement from the previous fit.

Changing the Nonlinearity Estimator

In this portion of the example, you improve the fit of the model you estimated with default settings, `nlarx1`, by changing the nonlinearity estimator.

1 In the Nonlinear Models dialog box, select the **Model Type** tab.

2 In the **Initial model** list, select `nlarx1`.

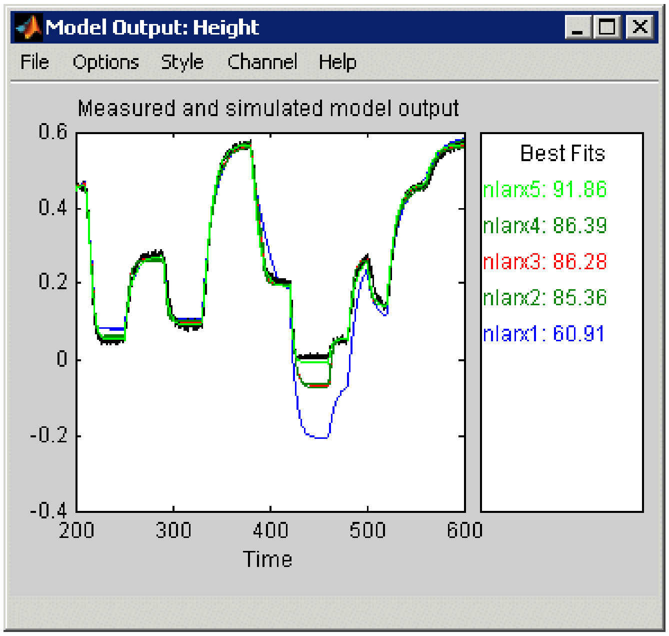
3 Select the **Model Properties** tab.

4 In the **Nonlinearity** list, select `Sigmoid Network`.

5 In the **Number of units in nonlinear block** field, enter `6`.

6 Click **Estimate**.

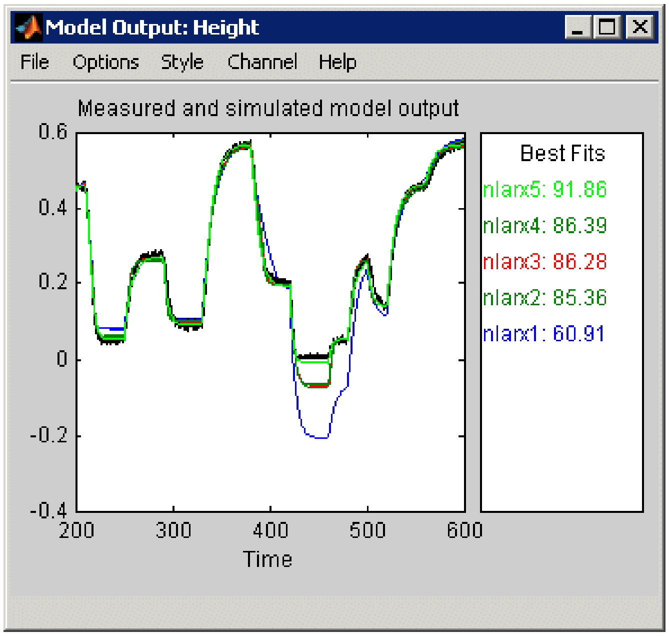
This action adds the model `nlarx5` to the Model Board in the System Identification Tool window. It also updates the Model Output plot, as shown in the following figure.



The **Best Fits** area of the Model Output plot shows that the `nlarx5` fit is 91.86%.

Selecting the Best Model

The best model is the simplest model that accurately describes the dynamics. In this example, the best model fit was produced in "Changing the Nonlinearity Estimator" on page 6-32, as shown in the following figure.


Model Output: Height
File
Options
Style
Channel
Help
Measured and simulated model output
0.6
0.4
0.2
0
-0.2
-0.4
200
300
400
500
600
Time
Best Fits
nlarx5: 91.86
nlarx4: 86.39
nlarx3: 86.28
nlarx2: 85.36
nlarx1: 60.91

Estimating Hammerstein-Wiener Models

You can estimate both continuous-time and discrete-time Hammerstein-Wiener models for data with the following characteristics:

- Time-domain input-output data.

Note Hammerstein-Wiener models do not support time-series data, where there is no input.

- Single-output or multiple-output data.

For more information about representing your data for system identification, see Chapter 3, “Representing Data for System Identification”.

This section discusses the following topics:

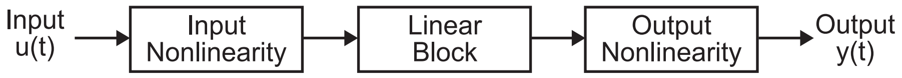
- “Definition of the Hammerstein-Wiener Model” on page 6-35
- “Nonlinearity Estimators for Hammerstein-Wiener Models” on page 6-37
- “Estimating Hammerstein-Wiener Models in the GUI” on page 6-38
- “Using nlhw to Estimate Hammerstein-Wiener Models” on page 6-39

For an example of estimating a Hammerstein-Wiener model using the System Identification Tool GUI, see “Example – Estimating Hammerstein-Wiener Model for a Two-Tank System” on page 6-44.

Definition of the Hammerstein-Wiener Model

The Hammerstein-Wiener structure models dynamic systems using up to two static nonlinear blocks in series with a linear block.

The input signal passes through the first nonlinear block, a linear block, and a second nonlinear block to produce the output signal, as shown in the following figure.



This model structure represents a nonlinear system as a linear system that is modified by static input and output nonlinearities. Thus, the linear model provides a reference point for estimating the nonlinear contributions in a system.

The following general equation describes the Hammerstein-Wiener structure:

$$w(t) = f(u(t))$$
$$x(t) = \frac{B_{j,i}(q)}{F_{j,i}(q)} w(t)$$
$$y(t) = h(x(t))$$

which contains the following variables:

- *u(t)* and *y(t)* are the inputs and outputs for the system, respectively.
- *f* and *h* are nonlinear functions that corresponding to the input and output nonlinearities, respectively.

  For multiple inputs and multiple outputs, *f* and *h* are defined component-wise.
- *w(t)* and *x(t)* are internal variables.

  w(t) has the same dimension as *u(t)*. *x(t)* has the same dimension as *y(t)*.
- *B(q)* and *F(q)* in the linear dynamic block are polynomials in the backward shift operator, as described in "Definition of Polynomial Models" on page 5-43.

  For *ny* outputs and *nu* inputs, the linear block is a trasnfer function matrix containing entries in the following form:

  $$\frac{B_{j,i}(q)}{F_{j,i}(q)}$$

  where $j = 1,2,...,ny$ and $i = 1,2,...,nu$.

If only the input nonlinearity is present, the model is called the Hammerstein model. If only the output nonlinearity is present, the model is called the Wiener model.

You must specify the following model orders for the linear block:

- n_b — The number of zeros plus 1.
- n_f — The number of poles.
- n_k — The delay from input to the output in terms of the number of samples.

For *ny* outputs and *nu* inputs, n_b, n_f, and n_k are *ny*-by-*nu* matrices. You can specify a nonlinearity for only certain inputs and outputs, and exclude the nonlinearity for other inputs and outputs.

Nonlinearity Estimators for Hammerstein-Wiener Models

Hammerstein-Wiener models support the following nonlinearity estimators for estimating the parameters of its input and output nonlinear blocks:

- Dead Zone
- Piecewise Linear
- Saturation
- Sigmoid Network
- Wavelet Network

You can exclude either the input nonlinearity or the output nonlinear from the model structure.

In the System Identification Tool GUI. Exclude a nonlinearity for a specific channel by selecting `None`.

In the MATLAB Command Window. Exclude a nonlinearity for a specific channel by specifying the `unitgain` value for the `InputNonlinearity` or `OutputNonlinearity` properties. For more information about estimation objects and their properties, see `nlhw` and `idnlhw` reference pages.

For a description of each nonlinearity estimator, see "Supported Nonlinearity Estimators" on page 6-60.

Estimating Hammerstein-Wiener Models in the GUI

The following procedure describes how to estimate a Hammerstein-Wiener model in the System Identification Tool GUI and assumes that you already have the appropriate data in the Data Board. For more information about preparing your data, see "Overview of Nonlinear Black-Box Modeling" on page 6-2.

1. In the System Identification Tool window, select **Estimate > Nonlinear models** to open the Nonlinear Models dialog box. The **Model Type** tab is shown.

2. In the **Model Structure** list, select `Hammerstein-Wiener`.

   This action updates the options in the Nonlinear Models dialog box to correspond to this model structure. For information about this model structure, see "Definition of the Hammerstein-Wiener Model" on page 6-35.

3. In the **Model name** field, edit the name of the model, or keep the default name. The name of the model should be unique in the Model Board.

4. (Optional) If you want to try refining a previously estimated model, select the name of this model in the **Initial model** list.

   Note The model structure and algorithm properties of the initial model populate the fields in the Nonlinear Models dialog box.

   A model is available in the **Initial model** list under the following conditions:

   - The model exists in the System Identification Tool window.
   - The number of model inputs and outputs matches the dimensions of the **Working Data** (estimation data) you selected in the System Identification Tool window.

5 Keep the default settings in the Nonlinear Models dialog box that specify the model structure and the algorithm, or modify the following settings:

   - In the **I/O Nonlinearity** tab, specify whether to include or exclude the input and output nonlinearities. For multiple-input and multiple-output systems, you can choose to apply nonlinearities only to specific input and output channels.
   - In the **I/O Nonlinearity** tab, change the input and output nonlinearity types and configure the nonlinearity settings.
   - In the **Linear Block** tab, specify the model orders and delays. To gain insight into possible delays, click **Infer Input Delay**.

   For more information about the available options, click **Help** in the Nonlinear Models dialog box to open the GUI help.

6 Click **Estimate** to add this model to the Model Board in the System Identification Tool window.

   The **Estimation** tab displays the estimation progress and results.

7 To plot this model, select the appropriate check box in the Model Views area of the System Identification Tool window. For more information about working with plots and validating models, see Chapter 9, “Plotting and Validating Models”.

If you get an inaccurate fit, try estimating a new model with different orders or nonlinearity estimator.

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 window. For more information about working with models, see Chapter 10, “Postprocessing and Using Estimated Models”.

Using nlhw to Estimate Hammerstein-Wiener Models

You can estimate Hammerstein-Wiener models using the `nlhw` command. The resulting models are stored as `idnlhw` model objects.

You can also use `pem` to refine parameter estimates of an existing Hammerstein-Wiener model, as described in “Refining Models” on page 1-46.

This section discusses the following topics:

- "General nlhw Syntax" on page 6-40
- "Example – Using nlhw to Estimate Hammerstein-Wiener Models " on page 6-42

General nlhw Syntax

Use the following general syntax to both configure and estimate Hammerstein-Wiener models:

```
m = nlhw(data,'nb',nb,
              'nf',nf,
              'nk',nk,
               InputNonlinearity,
               OutputNonlinearity,
              'Property1',Value1,...,
              'PropertyN',ValueN)
```

where `data` is the estimation data. `nb`, `nf`, and `nk` specify the orders and delays of the linear OE model. For more information about model orders, see "Definition of the Hammerstein-Wiener Model" on page 6-35.

`InputNonlinearity` specifies the input static nonlinearity estimator object as `'pwlinear'`, `'deadzone'`, `'saturation'`, `'sigmoidnet'`, `'wavenet'`, `'customnet'`, or `'unitgain'`. Similarly, `OutputNonlinearity` specifies the output static nonlinearity estimator object.

The property-value pairs specify any `idnlhw` model properties that configure the estimation algorithm. You can enter all model property-value pairs and top-level algorithm properties as a comma-separated list in `nlhw`. For example, you can control the iterative search for a best fit using the following properties:

```
m = nlhw(data,'nb',nb,
              'nf',nf,
              'nk',nk,
               InputNonlinearity,
               OutputNonlinearity,
              'MaxIter',N,
              'Tolerance',tol,
              'LimitError',lim,
              'Trace','on')
```

For `nu` inputs and `ny` outputs, `na`, `nb`, and `nk` are `ny`-by-`nu` matrices whose *i*-*j*th entry specifies the order and delay of the transfer function from the *j*th input to the *i*th output.

You can specify different nonlinearity estimators for different output channels by setting `InputNonlinearity` or `OutputNonlinearity` to an object array. For example:

```
m = nlarx(data,[nb,nf,nk],...
               [sigmoidnet;pwlinear],...
               [])
```

For detailed information about `nlhw` and `idnlhw`, see the corresponding reference pages.

For more information about validating your models, see Chapter 9, “Plotting and Validating Models”. To learn more about simulation and prediction output using your models, see Chapter 10, “Postprocessing and Using Estimated Models”.

Note You do not need to construct the model object using `idnlhw` before estimation.

Example – Using nlhw to Estimate Hammerstein-Wiener Models

This example uses `nlhw` to estimate a Hammerstein-Wiener model for the two-tank system, as described in "Example – Estimating Hammerstein-Wiener Model for a Two-Tank System" on page 6-44.

Prepare the data for estimation using the following commands:

```
load twotankdata
z = iddata(y, u, 0.2);
ze = z(1:1000); zv = z(1001:3000);
```

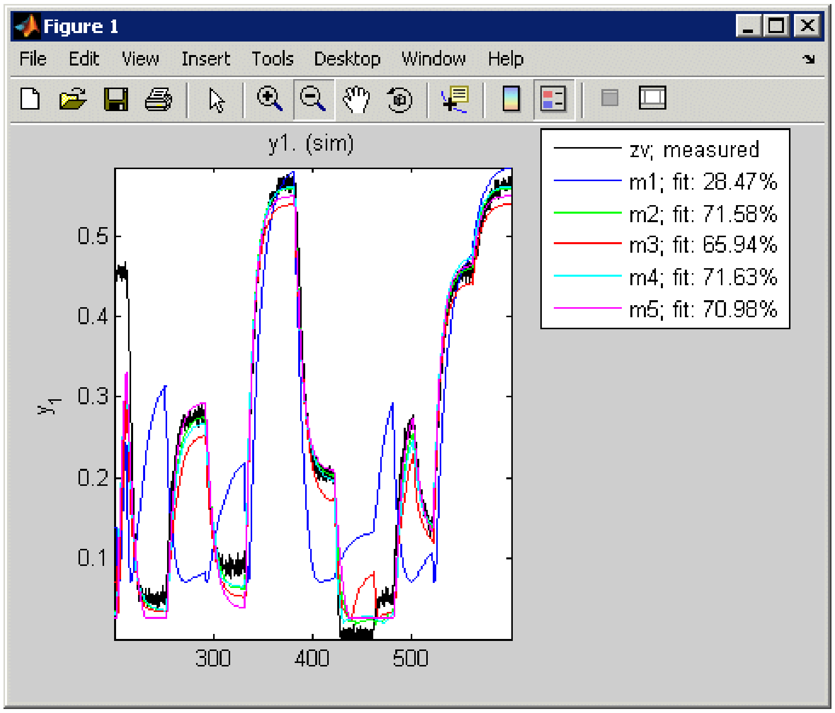
Estimate several models using different model orders, delays, and nonlinearity settings:

```
m1 = nlhw(ze,[2 3 1],'pwl','pwl');
m2 = nlhw(ze,[2 2 3],'pwl','pwl');
m3 = nlhw(ze,[2 2 3], pwlinear('num',13),...
                      pwlinear('num',10));
m4 = nlhw(ze,[2 2 3], sigmoidnet('num',2),...
                      pwlinear('num',10));
m5 = nlhw(ze,[2 2 3], 'dead','sat');
```

Compare the resulting models by plotting the model outputs on top of the measured output:

```
compare(zv,m1,m2,m3,m4,m5)
```

MATLAB responds with the following plot:



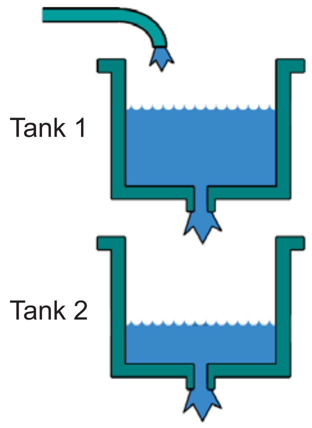
Example – Estimating Hammerstein-Wiener Model for a Two-Tank System

This example discusses the following topics:

About This Example

By performing the steps in this example, you get an overview of how to estimate nonlinear Hammerstein-Wiener models using the System Identification Tool GUI. Compare your results to the results obtained in the example on estimating nonlinear ARX models, as described in "Example – Estimating Nonlinear ARX Model for a Two-Tank System" on page 6-15.

In this tutorial, you estimate a Hammerstein-Wiener model to fit measured single-input and single-output (SISO) data for a two-tank system, shown in the following figure.



Two-Tank System

In the two-tank system, water pours through a pipe into Tank 1, drains into Tank 2, and leaves the system through a small hole at the bottom of Tank 2. The measured input *u(t)* to the system is the voltage applied to the pump that feeds the water into Tank 1 (in volts). The measured output *y(t)* is the height of the water in the lower tank (in meters).

Based on Bernoulli's law—which states that water flowing through a small hole at the bottom of a tank depends nonlinearly on the level of the water in the tank—you expect the relationship between the input and the output data to be nonlinear.

The sample data is provided in `twotankdata.mat`, which you install with the latest version of System Identification Toolbox. This MAT-file contains SISO time-domain data of 3000 samples with a sampling interval of 0.2 sec.

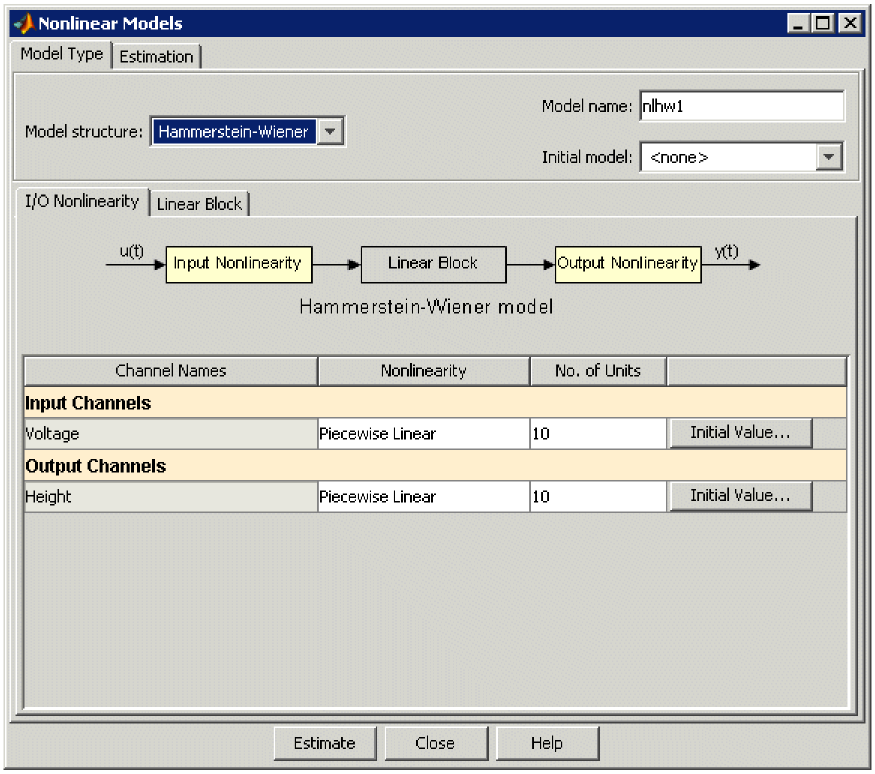
Estimating Hammerstein-Wiener Models with Default Settings

Prepare the sample data using the instruction in “Before You Begin” on page 6-16 in the example on estimating nonlinear ARX models, which uses the same data.

Perform the steps in the following procedure to estimate a Hammerstein-Wiener model for this sample data.

1. In the System Identification Tool window, select **Estimate > Nonlinear models** to open the Nonlinear Models dialog box.
2. In the **Model Type** tab, select `Hammerstein-Wiener` in the **Model Structure** list.

3 Keep the defaults in the **I/O Nonlinearity** tab



Model structure: Hammerstein-Wiener

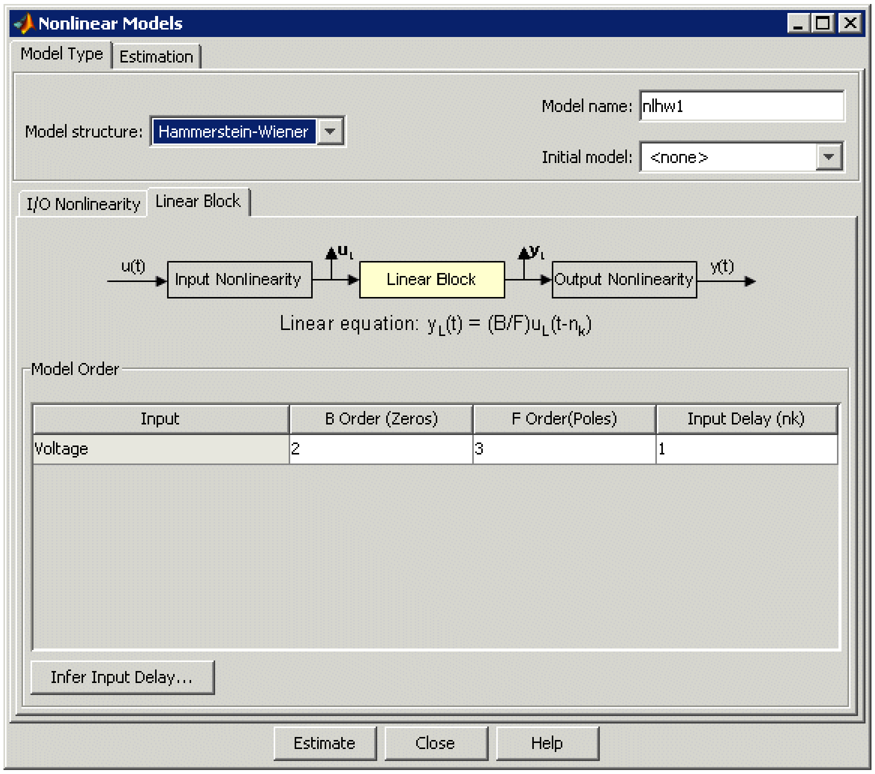
Model name: nlhw1

Initial model: <none>

Channel Names	Nonlinearity	No. of Units	
Input Channels			
Voltage	Piecewise Linear	10	Initial Value...
Output Channels			
Height	Piecewise Linear	10	Initial Value...

By default, the nonlinearity estimator is `Piecewise Linear` with `10` units for both the **Input Channels** and the **Output Channels**.

4 Keep the defaults in the **Linear Block** tab.



Model structure: Hammerstein-Wiener

Model name: nlhw1

Initial model: <none>

Linear equation: $y_L(t) = (B/F)u_L(t-n_k)$

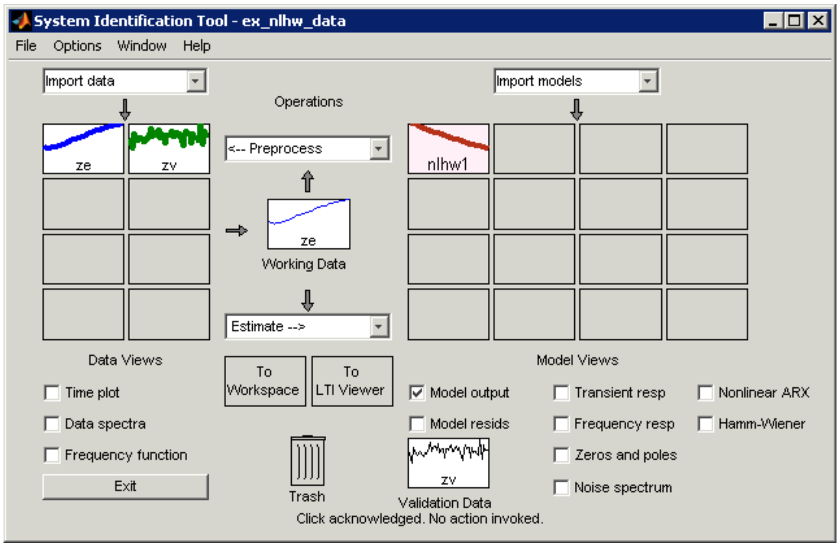
Model Order

Input	B Order (Zeros)	F Order(Poles)	Input Delay (nk)
Voltage	2	3	1

By default, the model orders and delays of the linear Output-Error (OE) model are n_b=2, n_f=3, and n_k=1.

5 Click **Estimate**.

This action adds the model `nlhw1` to the Model Board in the System Identification Tool window, as shown in the following figure.

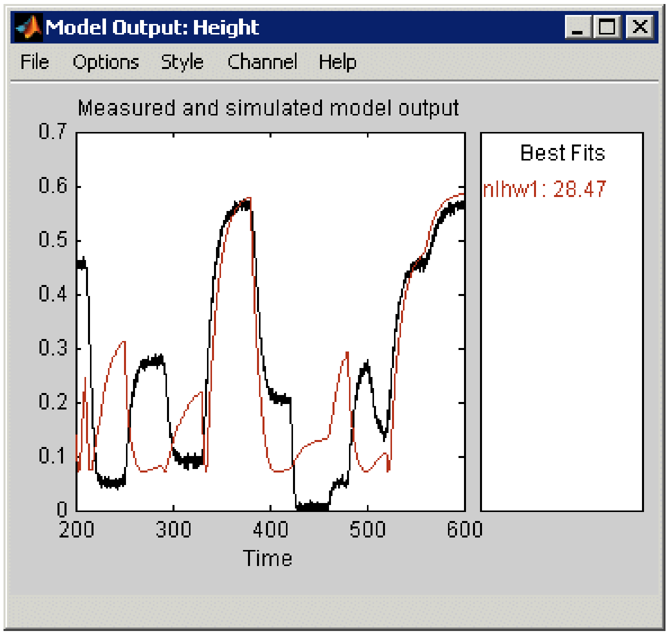


[✓ Model output]

Click acknowledged. No action invoked.

6 In the System Identification Tool window, select the **Model output** check box.

Simulation of the model output uses the input validation data as input to the model. It plots the simulated output on top of the output validation data.



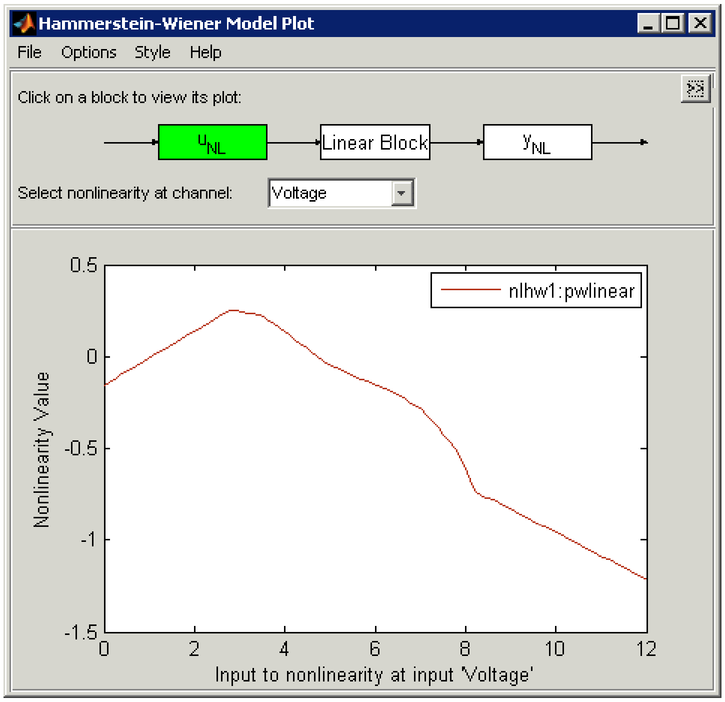
The **Best Fits** area of the Model Output plot shows that the agreement between the model output and the validation-data output is 28.47%. Thus, the default settings do not produce an accurate fit.

Plotting the Nonlinearities and Linear Transfer Function

You can view the input-output nonlinearities and the linear transfer function of the model on a Hammerstein-Wiener plot. For more information about working with Nonlinear ARX Model plot, see “Hammerstein-Wiener Plots” on page 9-53.

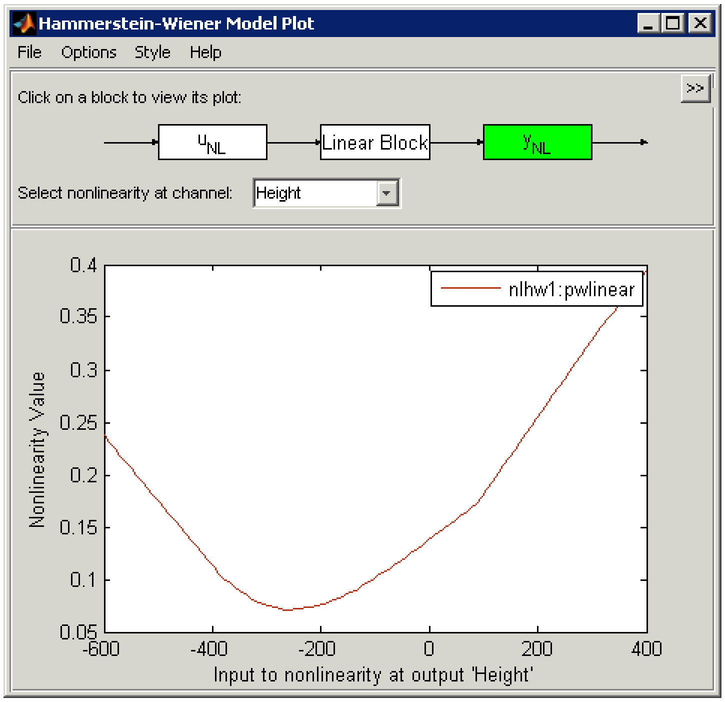
1 In the System Identification Tool window, select the **Hamm-Wiener** check box to view the Hammerstein-Wiener model plot.

The plot displays the input nonlinearity, as shown in the following figure.


Hammerstein-Wiener Model Plot
File
Options
Style
Help
Click on a block to view its plot:
u_{NL}
Linear Block
y_{NL}
Select nonlinearity at channel:
Voltage
nlhw1:pwlinear
Nonlinearity Value
Input to nonlinearity at input 'Voltage'
0.5
0
-0.5
-1
-1.5
0
2
4
6
8
10
12

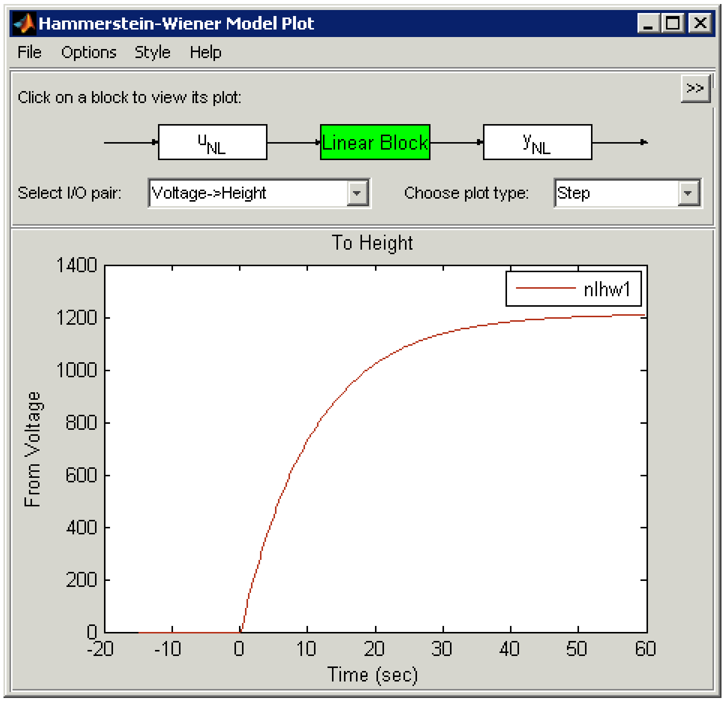
2 Click the $\mathbf{y_{NL}}$ rectangle in the top portion of the Hammerstein-Wiener Model Plot window.

The plot updates to display the output nonlinearity.

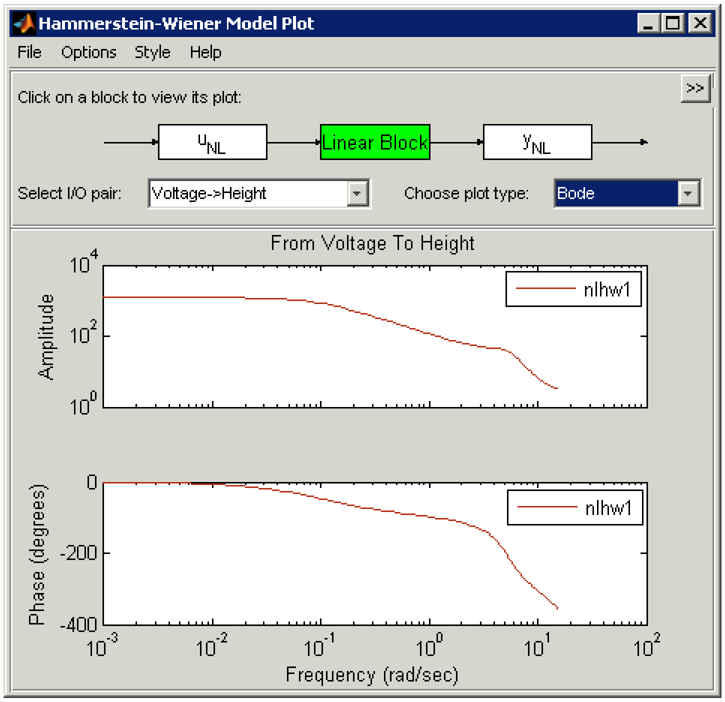


3 Click the **Linear Block** rectangle in the top portion of the Hammerstein-Wiener Model Plot window.

The plot updates to display the step response of the linear transfer function.



4 In the **Choose plot type** list, select `Bode`. This action displays a Bode plot of the linear transfer function, as shown in the following figure.



Select I/O pair: Voltage->Height

Choose plot type: Bode

Changing the Model Structure

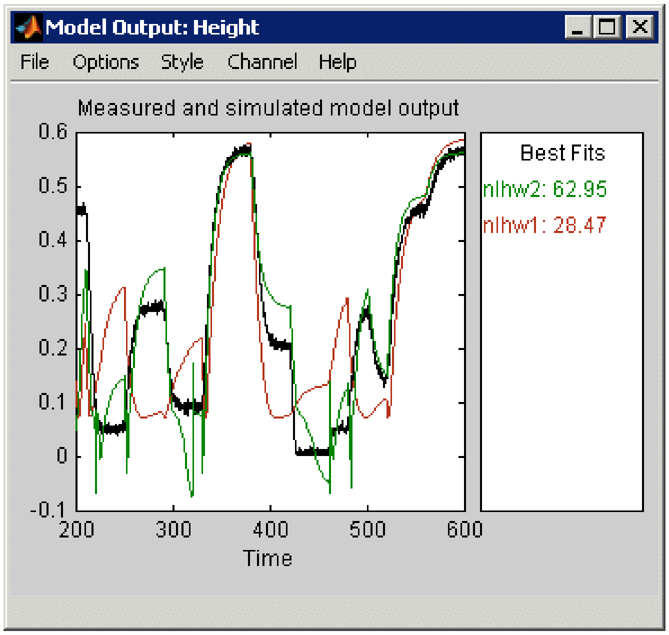
After estimating the Hammerstein-Wiener model with default settings, as described in “Estimating Hammerstein-Wiener Models with Default Settings” on page 6-46, you can try to improve the fit by modifying the model order and the nonlinearity properties. Typically, you modify model structure by trial and error until you get a model that produces an accurate fit to the data.

1 In the Nonlinear Models dialog box, select the **Model Type** tab, and then select the **Linear Block** tab.

2 For the `Voltage` input channel, double-click the corresponding **Input Delay (nk)** cell, type `3`, and press **Enter**.

3 Click **Estimate**.

This action adds the model `nlhw2` to the Model Board in the System Identification Tool window and the Model Output plot is updated to include this model, as shown in the following figure.



The **Best Fits** area of the Model Output plot shows that `nlhw2` fit is 62.95%.

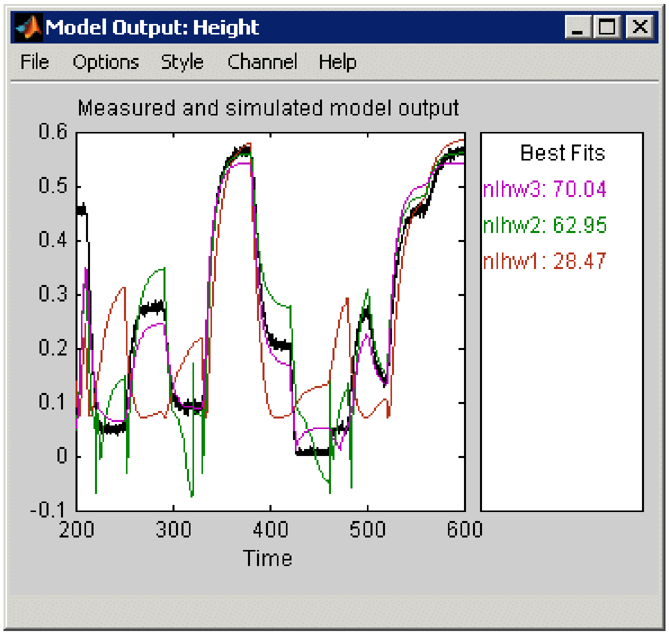
4 In the Nonlinear Models dialog box, select the **I/O Nonlinearity** tab.

5 For the `Voltage` input channel, double-click the corresponding **No. of Units** cell, type `20`, and press **Enter**.

This action changes the number of units for the `Piecewise Linear` nonlinearity estimator corresponding to the input channel.

6 Click **Estimate**.

This action adds the model `nlhw3` to the Model Board in the System Identification Tool window. It also updates the Model Output plot, as shown in the following figure.



The **Best Fits** area of the Model Output plot shows that the `nlhw3` fit is 70.04%.

Changing the Nonlinearity Estimator

In this portion of the example, you improve the fit by changing the nonlinearity estimator.

Note `Piecewise Linear` and `Sigmoid Network` are nonlinearity estimators for general nonlinearity approximation. If you know that you system includes saturation or dead-zone nonlinearities, specify these specialized nonlinearity estimators in your model.

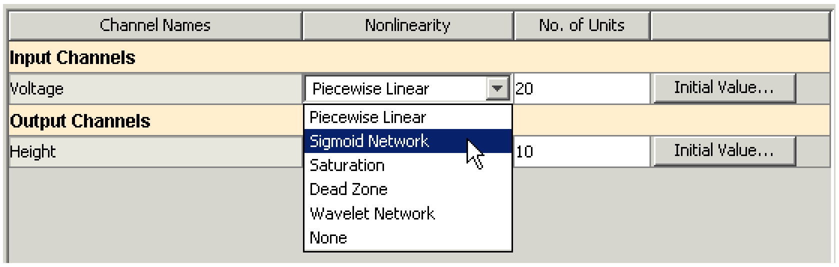
---

1 In the Nonlinear Models dialog box, select the **Model Type** tab, and then select the **Linear Block** tab.

2 For the `Voltage` input channel, double-click the corresponding **Input Delay (nk)** cell, type `1`, and press **Enter**. This action restores the input delay to the default value.

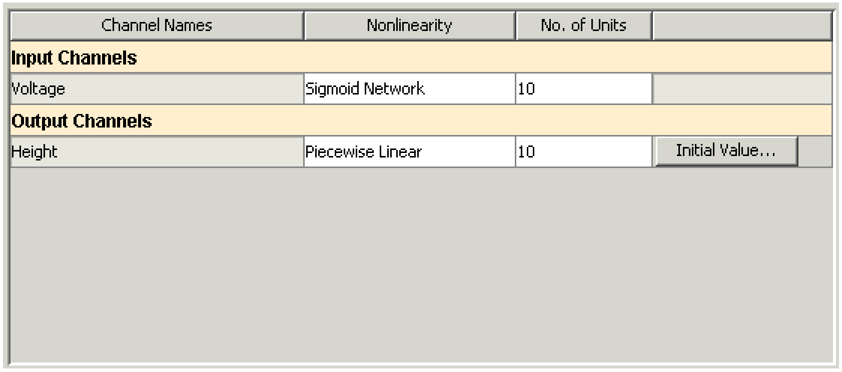
3 In the Nonlinear Models dialog box, select the **Model Type** tab, and then select the **I/O Nonlinearity** tab.

4 For the `Voltage` input, click the **Nonlinearity** cell, and select `Sigmoid Network` from the list, as shown in the following figure.



Channel Names	Nonlinearity	No. of Units	
Input Channels			
Voltage	Piecewise Linear	20	Initial Value...
Output Channels			
Height		10	Initial Value...

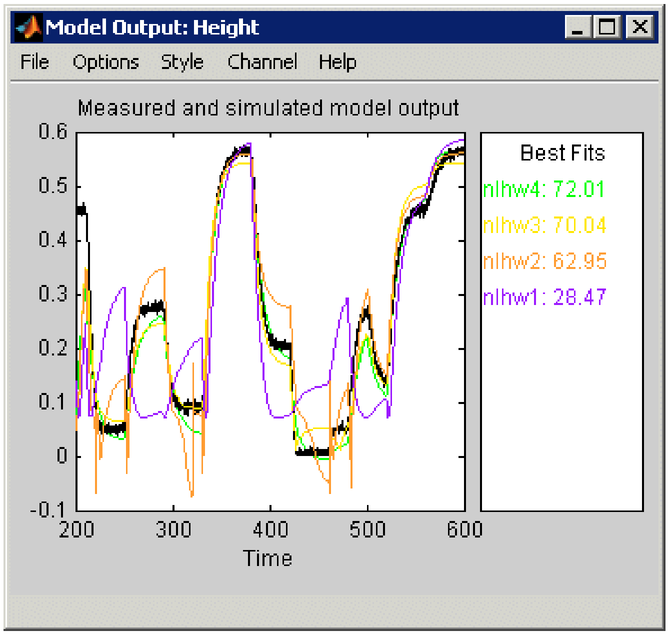
This action updates the corresponding **No. of Units** cell to 10 sigmoid units, as shown in the following figure.



Channel Names	Nonlinearity	No. of Units	
Input Channels			
Voltage	Sigmoid Network	10	
Output Channels			
Height	Piecewise Linear	10	Initial Value...

5 Click **Estimate**.

This action adds the model nlhw4 to the Model Board in the System Identification Tool window. It also updates the Model Output plot, as shown in the following figure.

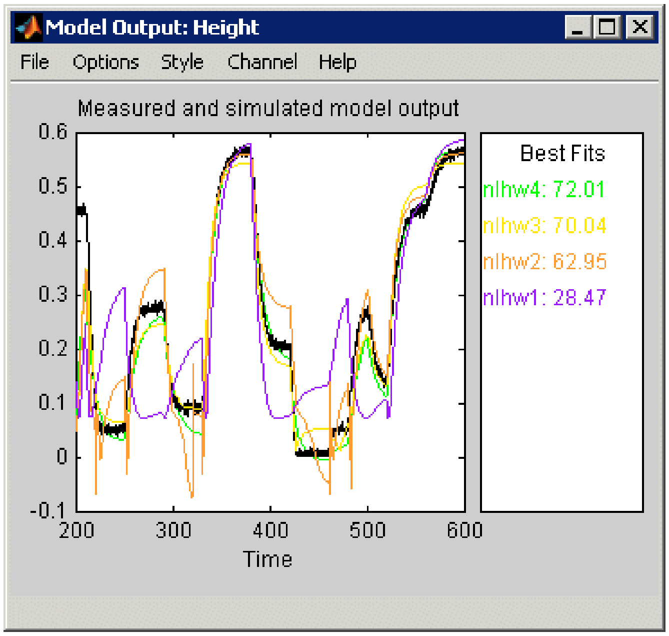


The **Best Fits** area of the Model Output plot shows that the nlhw4 fit is 72.01%.

Selecting the Best Model

The best model is the simplest model that accurately describes the dynamics.

In this example, the best model fit was produced in “Changing the Nonlinearity Estimator” on page 6-56, as shown in the following figure.


Model Output: Height
File
Options
Style
Channel
Help
Measured and simulated model output
0.6
0.5
0.4
0.3
0.2
0.1
0
-0.1
200
300
400
500
600
Time
Best Fits
nlhw4: 72.01
nlhw3: 70.04
nlhw2: 62.95
nlhw1: 28.47

Supported Nonlinearity Estimators

When configuring the nonlinear ARX and Hammerstein-Wiener models for estimation, you must specify a mathematical structure for the nonlinear portion of the model.

If you are working in the System Identification Tool GUI, specify the nonlinearity type by name when you configure the nonlinear model structure. If you are estimating or constructing a nonlinear model in the MATLAB Command Window, you specify the nonlinearity as an argument in the `nlarx` or `nlhw` estimation function.

- "Types of Nonlinearity Estimators" on page 6-60
- "Creating Custom Nonlinearities" on page 6-61

Types of Nonlinearity Estimators

System Identification Toolbox supports several nonlinearity estimators.

The following table summarizes supported nonlinearities for each type of nonlinear model. For a description of each nonlinearity, see the references pages for the corresponding nonlinearity object.

Nonlinearity	Object Name	Supported Model Type	Supports Multiple Inputs?
Custom Network (User-Defined)	`customnet`	Hammerstein-Wiener and Nonlinear ARX	Yes
Dead Zone	`deadzone`	Hammerstein-Wiener	No
Neural Network	`neuralnet`	Nonlinear ARX	Yes
Piecewise Linear	`pwlinear`	Hammerstein-Wiener	No
Saturation	`saturation`	Hammerstein-Wiener	No
Sigmoid Network	`sigmoidnet`	Hammerstein-Wiener and Nonlinear ARX	Yes
Tree Partition	`treepartition`	Nonlinear ARX	Yes
Wavelet Network	`wavenet`	Hammerstein-Wiener and Nonlinear ARX	Yes

The Neural Network nonlinearity lets you import a `network` object you created in Neural Network Toolbox.

The nonlinearity estimators `deadzone`, `pwlinear`, and `saturation` are optimized for estimating Hammerstein-Wiener models.

Creating Custom Nonlinearities

System Identification Toolbox lets you use a custom nonlinearity to estimate nonlinear ARX and Hammerstein-Wiener models.

A custom nonlinearity uses a unit functions that you define. This custom unit function uses a weighted sum of inputs to compute a scalar output.

You can use a combination of these unit functions to approximate the nonlinearity.

Note Hammerstein-Wiener models require that your custom nonlinearity have one input and one output.

```
function [f, g, a] = gaussunit(x)
%GAUSSUNIT example of customnet unit function
%
%[f, g, a] = GAUSSUNIT(x)
%
% x: unit function variable
% f: unit function value
% g: df/dx
% a: unit active range (g(x) is significantly
% nonzero in the interval [-a a])
%
% The unit function must be vectorized:
% for a vector or matrix x, the output
% arguments f and g must have the same size as x
% f and g are computed element by element.

f =  exp(-x.*x);
```

```
if nargout>1
  g = - 2*x .* f;
  a = 0.2;
end
```

Estimating Grey-Box Models

Overview of Grey-Box Modeling

Grey-box modeling is useful when you know the relationships between variables, constraints on model behavior, or explicit equations representing system dynamics. A *grey-box model* is a flexible model structure that lets you specify the mathematical structure of the model explicitly, including couplings between parameters and known parameter values.

If you understand the physics of your system and can represent the system using ordinary differential or difference equations (ODE) with unknown parameters, then you can use System Identification Toolbox to perform linear or nonlinear grey-box modeling.

System Identification Toolbox supports both continuous-time and discrete-time 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.

This section discusses the following topics:

- "Supported Data" on page 7-2
- "Before You Begin" on page 7-3
- "Specifying the Grey-Box Structure" on page 7-3
- "Using Grey-Box Models" on page 7-4

If it is too difficult to describe your system using known physical laws, you can use System Identification Toolbox to perform black-box modeling. For more information about black-box modeling, see Chapter 5, "Estimating Linear Nonparametric and Parametric Models" and Chapter 6, "Estimating Nonlinear Black-Box Models".

Supported Data

You can estimate both continuous-time or discrete-time grey-box models for data with the following characteristics:

- Time-domain or frequency-domain data.

Note Nonlinear grey-box models support only time-domain data.

- Single-output or multiple-output data.

For more information on representing your data for system identification, see Chapter 3, "Representing Data for System Identification".

Before You Begin

Before you begin estimating models, import the data into MATLAB, and represent the data using System Identification Toolbox format. If you are using the System Identification Tool, then import the data into the GUI to make the data available to System Identification Toolbox. However, if you prefer to work in the MATLAB Command Window, then represent your data as an `iddata` or `idfrd` object. For more information about representing your data for system identification, see Chapter 3, "Representing Data for System Identification".

After representing data in System Identification Toolbox format, plot the data on a time plot or an estimated frequency response plot to examine the data features. You can also use the `advice` command to analyze the data for the presence of constant offsets and trends, delay, feedback, and nonlinearity, and determine the order persistence of excitation.

You can preprocess your data by removing offsets and linear trends, interpolating missing values, filtering to emphasize a specific frequency range, or resampling using a different time interval.

For more information about types of available date plots and data-preprocessing operations, see Chapter 4, "Plotting and Preprocessing Data".

Specifying the Grey-Box Structure

The first step in grey-box modeling is to define the model structure in an m-file or MEX-file.

For linear models. Write an m-file that returns state-space matrices as a function of user-defined parameters and information about the model. For examples, see "Linear Grey-Box Models" on page 7-5.

Next, create an `idgrey` object based on this m-file. For more information, see "Specifying the Linear Grey-Box Model Structure" on page 7-5.

For nonlinear models. Write an m-file or MEX-file to return the first-order derivatives of the states and output values as a function of the states, inputs, time, parameters, and auxiliary variables. For information about the file structure, see "Nonlinear Grey-Box Models" on page 7-13

Next, create an `idnlgrey` object based on this m-file or MEX-file.

Using Grey-Box Models

After estimating both linear and nonlinear grey-box models, you can use the `sim` command to simulate the model, or the `predict` command to predict the model output. For more information about using models, see Chapter 10, "Postprocessing and Using Estimated Models".

System Identification Toolbox represents linear grey-box models as `idgrey` model objects. You can convert these models to state-space form using the `idss` command and analyze the model behavior using transient- and frequency-response plots and other linear analysis plots, as described in Chapter 9, "Plotting and Validating Models".

System Identification 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.

Linear Grey-Box Models

You can estimate linear discrete-time and continuous-time grey-box models for arbitrary ordinary differential or difference equations using single-output and multiple-output time-domain data, or time-series data that has no measured inputs.

This section describes the following topics:

- "Specifying the Linear Grey-Box Model Structure" on page 7-5
- "Example – Representing a Grey-Box Model in an M-File" on page 7-6
- "Example – Continuous-Time Grey-Box Model for Heat Diffusion" on page 7-8
- "Example – Discrete-Time Grey-Box Model for Parameterized Disturbance Models" on page 7-11

Specifying the Linear Grey-Box Model Structure

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*:

$$\begin{aligned}
x(kT+T) &= Ax(kT) + Bu(kT) + Ke(kT) \\
y(kT) &= Cx(kT) + Du(kT) + e(kT) \\
x(0) &= x0
\end{aligned}$$

Use the following format to implement the linear grey-box model in an m-file:

```
[A,B,C,D,K,x0] = myfunc(par,T,CDmfile,aux)
```

where the matrices `A`, `B`, `C`, `D`, `K`, and `x0` represent both continuous-time and discrete-time descriptions, `myfunc` is the name of the m-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 m-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 pages.

For more information about validating your models, see Chapter 9, "Plotting and Validating Models". To learn more about simulation and prediction output using your models, see Chapter 10, "Postprocessing and Using Estimated Models".

Example – Representing a Grey-Box Model in an M-File

In this example, you represent the structure of the following continuous-time model:

$$\dot{x} = \begin{bmatrix} 0 & 1 \\ 0 & \theta_1 \end{bmatrix} x(t) + \begin{bmatrix} 0 \\ \theta_2 \end{bmatrix} u(t)$$

$$y(t) = \begin{bmatrix} 1 & 0 \\ 0 & 1 \end{bmatrix} x(t) + e(t)$$

$$x(0) = \begin{bmatrix} \theta_3 \\ 0 \end{bmatrix}$$

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, 1999.

The continuous-time state-space model structure is defined by the following equation:

$$\begin{aligned}
\dot{x}(t) &= Fx(t) + Gu(t) + \tilde{K}w(t) \\
y(t) &= Hx(t) + Du(t) + w(t) \\
x(0) &= x0
\end{aligned}$$

To prepare this model for identification in System Identification Toolbox, perform the following procedure:

1 Create the following m-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(2,2);
K = zeros(2,1);
x0 =[par(3);0];
```

2 Use the following syntax to define an `idgrey` model object based on the `myfunc` m-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 5-89, where the same problem is solved using a structured state-space representation.

Example – 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:

$$k \cdot \Delta L \le \xi < \left(k+1\right)$$

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:

$$\begin{aligned}
\dot{x}(t) &= Fx(t) + Gu(t) + \tilde{K}w(t) \\
y(t) &= Hx(t) + Du(t) + w(t) \\
x(0) &= x0
\end{aligned}$$

The following m-file 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 m-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);
```

Use the following syntax to define an idgrey model object based on the myfunc m-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.

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 pages.

Example – Discrete-Time Grey-Box Model for Parameterized Disturbance Models

This example shows how to create a grey-box model structure when you know the variance of the measurement noise. The code in this example uses the Control System Toolbox function dlqr for computing the Kalman gain from the known and estimated noise variance.

Consider the following discrete-time state-space equation:

$$x(kT+T) = \begin{bmatrix} par1 & par2 \\ 1 & 0 \end{bmatrix} x(kT) + \begin{bmatrix} 1 \\ 0 \end{bmatrix} u(kT) + w(kT)$$
$$y(kT) = \begin{bmatrix} par3 & par4 \end{bmatrix} x(kT) + e(kT)$$
$$x(0) = x0$$

where w and e are independent white noises with covariance matrices *R1* and *R2*, respectively. *par1*, *par2*, *par3*, and *par4* represent the unknown parameter values to be estimated.

Suppose that you know the variance of the measurement noise *R2*, and that only the first component of*w(t)* is nonzero. The following m-file shows how to capture this information in an m-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];
K = A*dlqr(A,eye(2),C,R1,R2); % Uses Control System Toolbox
x0 = [0;0];
```

The Kalman gain is computed using the `dlqr` function in Control System Toolbox.

Nonlinear Grey-Box Models

You can estimate nonlinear discrete-time and continuous-time grey-box models for arbitrary nonlinear ordinary differential equations using single-output and multiple-output time-domain data, or time-series data that has no inputs. Your grey-box models can be static or dynamic.

Grey-box models describe the system behavior as a set of nonlinear differential or difference equations with unknown parameters.

This section describes the following topics:

Nonlinear Grey-Box Demos and Examples

System Identification Toolbox 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 MATLAB prompt:

```
iddemo
```

For examples of m-files and MEX-files that specify model structure, see the `toolbox/ident/iddemos/examples` directory. For example, the model of a DC motor—used in the demo `idnlgreydemo1`—is described in files `dcmotor_m` and `dcmotor_c`.

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 $x^{\dagger}(t) = \frac{d}{dt}x(t)$ 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 an m-file 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 object. 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 MATLAB with reasonable inputs and verify the output values.

For an example of creating grey-box model files, see the demo *Creating idnlgrey Model Files*.

Constructing the idnlgrey Object

After you create the m-file or MEX-file with you 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 m-file or MEX-file storing the model structure. This file must be on the MATLAB path.
- `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 a same way as parameters. Must be fourth input to the `idnlgrey` constructor.

For detailed information about this object and its properties, see the `idnlgrey` reference pages. For general information about working with model objects, see "Working with Model Objects" on page 1-19.

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 pages.

For more information about validating your models, see Chapter 9, "Plotting and Validating Models". To learn more about simulation and prediction output using your models, see Chapter 10, "Postprocessing and Using Estimated Models".

Specifying the Estimation Algorithm

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 7-17

- “Search Method” on page 7-17
- “Gradient Options” on page 7-18
- “Example — Specifying Algorithm Properties” on page 7-18

For detailed information about these and other model properties, see the `idnlgrey` reference pages.

Simulation Method

You can specify the simulation method using the `SimulationOptions` (`struct`) fields of the model `Algorithm` property.

MATLAB 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 MATLAB 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 generating maximum likelihood estimates of 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 .

(Requires Optimization Toolbox) 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. When the parameter bounds are different from the default +/- Inf, this search method handles the bounds better than the schemes based on line search, but the results might not be optimal in the maximum-likelihood sense.

By default, `SearchMethod` is set to `` `Auto' ``, automatically selects a method from the available minimizers. If Optimization Toolbox is available, `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, if you want to use `` SearchMethod = `gn' ``, `MaxIter = 5`, and `` Trace = `on' ``, use the following syntax in the pem command:

```
m = pem(data,init_model,`Search',`gn',...
                        `MaxIter',5,...
                        `Trace',`On')
```

Recursive Parameter Estimation

Overview of 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 an online model for batches of input-output data might be used to address 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.

You can use System Identification Toolbox commands to 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 (Auto-Regressive) and ARMA (Auto-Regressive Moving Average) single-output models.

For examples of recursive estimation and data segmentation using System Identification Toolbox, run the Recursive Estimation and Data Segmentation demonstration by typing the following command at the MATLAB prompt:

```
iddemo5
```

This chapter discusses the following topics:

- "Recursive Estimation Commands" on page 8-4
- "Algorithms for Recursive Estimation" on page 8-7
- "Data Segmentation" on page 8-14

For detailed information about recursive parameter estimation algorithms, see the corresponding chapter in *System Identification: Theory for the User* by Lennart Ljung (PTR Prentice Hall, Upper Saddle River, NJ, 1999).

Recursive Estimation Commands

Before estimating models using recursive algorithms, you must import your data into the MATLAB workspace and represent your data in one 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 3, "Representing Data for System Identification".

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 kth 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 kth 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 cancellation.

---

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 "Black-Box Polynomial Models" on page 5-42.

adm and adg specify any of the four recursive algorithm, as described in "Algorithms for Recursive Estimation" on page 8-7.

The following table summarizes the recursive estimation commands supported by System Identification Toolbox. The command description indicates whether you can estimate single-input, single-output, multiinput, and multioutput, and time-series (no input) models. For details about each command, see the corresponding reference pages.

Tip For ARX and AR models, use `rarx`. For single-input and 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 and single-output ARMAX and ARMA models.
`rarx`	Estimate parameters of single- or multiinput and single-output ARX and AR models. Does not support multioutput system.
`rbj`	Estimate parameters of single-input and single-output Box-Jenkins models.
`roe`	Estimate parameters of single-input and single-output Output-Error models.

Commands for Linear Recursive Estimation (Continued)

Command	Description
`rpem`	Estimate parameters of multiinput 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 multiinput and single-output systems when you want to use recursive pseudolinear regression method.

Algorithms for Recursive Estimation

This section describes the four recursive estimation algorithms supported by System Identification Toolbox. For detailed information about recursive parameter estimation algorithms, see the corresponding chapter in *System Identification: Theory for the User* by Lennart Ljung (PTR Prentice Hall, Upper Saddle River, NJ, 1999).

You specify the type of recursive estimation algorithms as arguments `adm` and `adg` of the recursive estimation commands in "Recursive Estimation Commands" on page 8-4.

This section discusses the following topics:

- "General Form of Recursive Estimation Algorithm" on page 8-7
- "Kalman Filter Algorithm" on page 8-8
- "Forgetting Factor Algorithm" on page 8-10
- "Unnormalized and Normalized Gradient Algorithms" on page 8-12

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 System Identification Toolbox differ based on different approaches for choosing the form of *Q(t)* and computing $\psi(t)$, where $\psi(t)$ represents the gradient of the predicted model output $\hat{y}(t \mid \theta)$ with respect to the parameters θ.

The simplest way to visualize the role of the gradient $\psi(t)$ of the parameters, is to consider models with a linear-regression form:

$$y(t) = \psi^T(t)\theta_0(t) + e(t)$$

In this equation, $\psi(t)$ is the *regression vector* that is computed based on previous values of measured inputs and outputs. $\theta_0(t)$ represents the true parameters. *e(t)* is the noise source (*innovations*), which is assumed to be white noise. The specific form of $\psi(t)$ depends on the structure of the polynomial model.

For linear regression equations, the predicted output is given by the following equation:

$$\hat{y}(t) = \psi^T(t)\hat{\theta}(t-1)$$

For models that do not have the linear regression form, it is not possible to compute exactly the predicted output and the gradient $\psi(t)$ for the current parameter estimate $\hat{\theta}(t-1)$. To learn how you can compute approximation for $\psi(t)$ and $\hat{\theta}(t-1)$ for general model structures, see the section on recursive prediction-error methods in *System Identification: Theory for the User* by Lennart Ljung (PTR Prentice Hall, Upper Saddle River, NJ, 1999).

Kalman Filter Algorithm

This section discusses the following topics:

- “Mathematics of the Kalman Filter Algorithm” on page 8-9

- "Using the Kalman Filter Algorithm" on page 8-10

Mathematics of the Kalman Filter Algorithm

The following set of equations summarize 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 8-7 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 8-9).

Forgetting Factor Algorithm

This section discusses the following topics:

- “Mathematics of the Forgetting Factor Algorithm” on page 8-10
- “Using the Forgetting Factor Algorithm” on page 8-12

Mathematics of the Forgetting Factor Algorithm

The following set of equations summarize the *forgetting factor* adaptation algorithm.

$$\hat{\theta}(t) = \hat{\theta}(t-1) + K(t)\left(y(t) - \hat{y}(t)\right)$$

$$\hat{y}(t) = \psi^T(t)\hat{\theta}(t-1)$$

$$K(t) = Q(t)\psi(t)$$

$$Q(t) = P(t) = \frac{P(t-1)}{\lambda + \psi(t)^T P(t-1)\psi(t)}$$

$$P(t) = \tfrac{1}{\lambda}\left(P(t-1) - \frac{P(t-1)\psi(t)\psi(t)^T P(t-1)}{\lambda + \psi(t)^T P(t-1)\psi(t)}\right)$$

To obtain *Q(t)*, the following functions 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 $\tau = {}^{1}\!/_{1-\lambda}$ 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 8-8.

Using the Forgetting Factor Algorithm

The general syntax for the command described in “Algorithms for Recursive Estimation” on page 8-7 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 8-10).

Tip λ typically has a positive value between `0.97` and `0.995`.

Unnormalized and Normalized Gradient Algorithms

In the linear regression case, the gradient methods are also known as the *least mean squares* (LMS) methods.

This section discusses the following topics:

- “Mathematics of the Unnormalized and Normalized Gradient Algorithm” on page 8-12
- “Using the Unnormalized and Normalized Gradient Algorithms” on page 8-13

Mathematics of the Unnormalized and Normalized Gradient Algorithm

The following set of equations summarize 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 $\psi(t)$:

$$Q(t) = \frac{\gamma}{|\psi(t)|^2} I$$

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 8-7 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 8-12).

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, such that the model parameters are piecewise constant over time. For detailed information about this function, see the corresponding reference pages.

To see an example of using data segmentation, run the Recursive Estimation and Data Segmentation demonstration by typing to the following command at the MATLAB prompt:

```
iddemo5
```

Plotting and Validating Models

Overview of Model Plots and Validation

After estimating each model, you can validate the model to determine whether the model can reproduce system behavior by simulation or prediction within acceptable bounds. You iterate between estimation and validation until you find the simplest model that adequately captures the system dynamics.

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*). When you validate a model using a fresh data set, this process is called *cross-validation*. Although you can designate the same data set to be used for estimating and validating the model, you risk overfitting your data.

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 "Detrending Data" on page 4-20.

This section discusses the following topics:

- "Supported Model Plots" on page 9-4
- "Validating Models" on page 9-5
- "Getting Advice About Models" on page 9-6

For ideas on how to adjust your modeling strategy based on validation results, see "Troubleshooting Models" on page 9-64.

Tip If you have installed Control System Toolbox, you can also view models using the LTI Viewer. For more information, see "Viewing Model Response in the LTI Viewer" on page 10-23.

Supported Model Plots

The following table summarizes the types of supported model plots in System Identification Toolbox. To learn more about each type of plot, see the corresponding section.

Plot Types	Supported Models	Learn More
Model Output	All linear and nonlinear models	"Model Output Plots" on page 9-7
Residual Analysis	All linear and nonlinear models	"Residual Analysis Plots" on page 9-15
Transient Response	• All linear parametric models • Correlation analysis (nonparametric) models • For nonlinear models, only step response.	"Impulse and Step Response Plots" on page 9-21
Frequency Response	• All linear parametric models • Spectral analysis (nonparametric) models	"Frequency Response Plots" on page 9-29
Noise Spectrum	• All linear parametric models • Spectral analysis (nonparametric) models	"Noise Spectrum Plots" on page 9-36
Poles and Zeros	All linear parametric models	"Pole-Zero Plots" on page 9-43

Plot Types	**Supported Models**	**Learn More**
Nonlinear ARX	Nonlinear ARX models only	"Nonlinear ARX Plots" on page 9-48
Hammerstein-Wiener	Hammerstein-Wiener models only	"Hammerstein-Wiener Plots" on page 9-53

You can create most plots in both the System Identification Tool GUI and the MATLAB Command Window. For general information about working with plots in System Identification Toolbox, see "Working with Plots" on page 2-29.

The plots you create using the System Identification Tool provide additional options that are specific to System Identification Toolbox, such as selecting input-output channels and displaying confidence intervals.

The plots you create in the MATLAB Command Window display in the MATLAB figure window and provide MATLAB options for formatting, saving, printing, and exporting to a variety of file formats.

Note You can only display confidence intervals for linear models.

Validating Models

The most common approach to validating models is to create Model Output and Residual Analysis plots. The Model Output plot helps you compare simulated or predicted model output to measured output. For more information, see "Model Output Plots" on page 9-7. The Residual Analysis plot displays the results of residual tests. For more information, see "Residual Analysis Plots" on page 9-15. Model Output and Residual analysis plots are available for all model types.

Displaying confidence intervals for linear and nonlinear grey-box models on these and other plots lets you assess the uncertainty of model parameters. For more information, see "Viewing Model Uncertainty" on page 9-61.

In addition to these validation approaches, System Identification Toolbox provides the following way for validating your models:

- Analyzing model response plots to gain insight into how well parametric models captures system dynamics. For more information, see "Impulse and Step Response Plots" on page 9-21 and "Frequency Response Plots" on page 9-29. For information about the response of the noise model, see "Noise Spectrum Plots" on page 9-36.
- Plot the poles and zeros of the linear parametric model. For more information, see "Pole-Zero Plots" on page 9-43.
- Plot linear and nonlinear blocks of Hammerstein-Wiener and nonlinear ARX models. For more information, see "Hammerstein-Wiener Plots" on page 9-53 and "Nonlinear ARX Plots" on page 9-48.
- 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 `aic` and `fpe` reference pages.

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?

Model Output Plots

System Identification Toolbox lets you validate all linear parametric models and nonlinear models by checking how well the simulated or predicted output of the model matches the measured output.

Note For nonparametric models, including impulse-response, step-response, and frequency-response models, model output 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.

If you plan to use the model for simulation applications, validate the model by comparing simulated output to the validation data. However, if you plan to use the model for prediction, compare the *k*-step-ahead predicted output to the validation data. For example, if you are modeling a plant for a control system, your model must perform for prediction over a horizon that corresponds to the time-constant of the system. For information about the difference between simulation and prediction, see the introductory chapter in the *Getting Started with System Identification Toolbox*.

This section discusses the following topics:

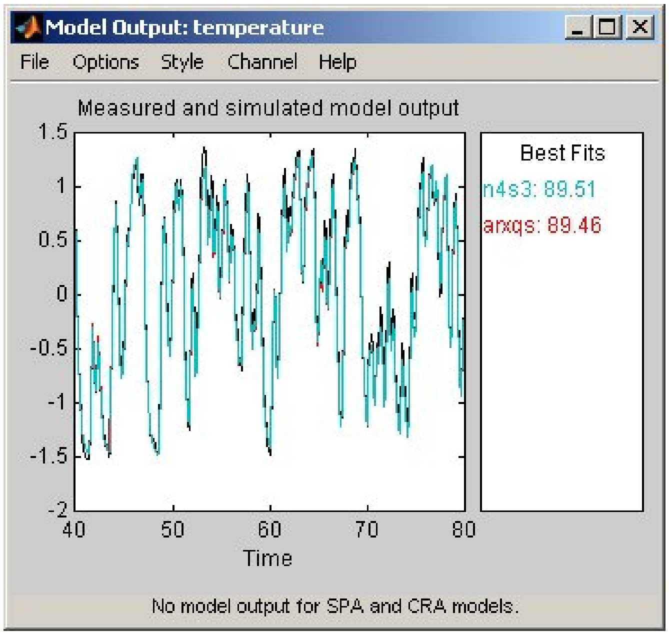
What Does a Model Output Plot Show?

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 simulated complex-valued amplitude of the model output. The complex-valued amplitude is equal to the product of the Fourier transform of the input and the model frequency function.
- For frequency-response data, the plot shows the simulated amplitude of the model frequency response.

The following figure shows a sample Model Output plot, created in the System Identification Tool GUI.


Model Output: temperature
File
Options
Style
Channel
Help
Measured and simulated model output
1.5
1
0.5
0
-0.5
-1
-1.5
-2
40
50
60
70
80
Time
Best Fits
n4s3: 89.51
arxqs: 89.46
No model output for SPA and CRA models.

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.

Choosing Simulated or Predicted Output

How you validate the model output should match how you plan to use the model. If you plan to use the model in simulation applications, then compare the simulated model output to the validation data. However, if you plan to use the model for prediction, then compare the predicted output from the model to the measured output.

The main difference between simulation and prediction is whether System Identification Toolbox uses measured or computed previous outputs for calculating the next output.

Using a model for prediction is common in controls applications where you want to predict output a certain number of steps in advance. When you use System Identification Toolbox to *predict* model output, the algorithm uses both the measured and the calculated output data values in the difference equation for computing the next output.

The predicted value *y(t)* is computed from all available inputs *u(s)*, where $s \le (t-k)$, and all available outputs *y(s)*, where $s \le (t-k)$.

Simulating models uses the input-data values from a data set to compute the output values. When you *simulate* the model output, System Identification Toolbox computes the first output value using the initial conditions and the inputs. Then, System Identification Toolbox feeds this computed output into the differential (continuous-time) or difference (discrete-time) equation for calculating the next output value. In this way, the simulation progresses using previously calculated outputs in the difference equation to produce the next output; with an *infinite prediction horizon* (k=∞), the simulation has no limit on how far out in time it computes output values. Thus, no past outputs are used in the simulation.

To check whether the model has picked up interesting dynamic properties, let the predicted time horizon kT be larger than the important time constants, where T is the sampling interval.

Note Output-error models, obtained by fixing K to zero for state-space models and setting `na=nc=nd=0` for polynomial models, do not use past outputs. Therefore, for these models, the simulated and the predicted outputs are the same for any value of k.

To learn how to display simulated or predicted output, see the description of the plot settings in “Plotting Model Output Using the GUI” on page 9-10.

Displaying the Confidence Interval

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 the description of the plot settings in “Plotting Model Output Using the GUI” on page 9-10.

The *confidence interval* corresponds to the range of output values with a specific probability of being the actual output of the system. System Identification 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.

Plotting Model Output Using the GUI

To create a model output plot for parametric linear and nonlinear models in the System Identification Tool window, select the **Model output** check box in the **Model Views** area. For general information about creating and working with plots, see “Working with Plots” on page 2-29.

To include or exclude a model on the plot, click the corresponding model icon in the System Identification Tool window. Active models display a thick line inside the Model Board icon.

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.

The following table summarizes the Model Output plot settings.

Model Output Plot Settings

Action	Command
Display confidence intervals. **Note** Confidence intervals are only available for simulated model output of linear models. Confidence internal are not available for nonlinear ARX and Hammerstein-Wiener models.	• To display the dashed lines on either side of the nominal model curve, select **Options > Show confidence intervals**. Select this option again to hide the confidence intervals. • To change the confidence value, select **Options > Set % confidence level**, and choose a value from the list. • To enter your own confidence level, select **Options > Set confidence level > Other**. Enter the value as a probability (between 0 and 1) or as the number of standard deviations of a Gaussian distribution.
Change between simulated output or predicted output. **Note** Prediction is only available for time-domain validation data.	• Select **Options > Simulated output** or **Options > k step ahead predicted output**. • To change the prediction horizon, select **Options > Set prediction horizon**, and select the number of samples. • To enter your own prediction horizon, select **Options > Set prediction horizon > Other**. Enter the value in terms of the number of samples.
Display the actual output values (**Signal plot**), or the difference between model output and measured output (**Error plot**).	Select **Options > Signal plot** or **Options > Error plot**.

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.

Using Functions to Plot Model Output

You can plot simulated and predicted model output using the `compare`, `sim`, and `predict` functions.

Simulation and prediction requires input data, a model, and the values of the initial states. If you estimated the model using one data set, but want to simulate the model using a different data set, the initial states of your simulation must be consistent with the latter data set.

By default, `sim` and `predict` use the initial states that were derived from the data you used to estimate the model. These initial states are not appropriate if you are simulating or predicting output using new data.

To use `sim` or `predict` with a data set that differs from the data you used to estimate the model, first estimate the new initial states `X0est` using `pe`:

```
[E,X0est]=pe(model,data)
```

Next, specify the estimated initial states `X0est` as an argument in `sim` or `predict`. For example:

```
y=sim(model,data,'InitialState',X0est)
```

Note The `compare` function automatically estimates the initial states from the data and ensures consistency.

Function	Description	Example
`compare`	Plots simulated or predicted model output 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`	Plots simulated model output only.	To simulate the response of the model `model` using input data `data`, use the following command: `sim(model,data)`
`predict`	Plots predicted 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)`

Residual Analysis Plots

You can validate parametric linear and nonlinear models by checking the behavior of the model residuals. *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.

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.

This section discusses the following topics:

- "What Down the Residuals Plot Show?" on page 9-15
- "Overview of Residual Analysis" on page 9-16
- "Displaying the Confidence Interval" on page 9-17
- "Plotting Residuals Using the GUI" on page 9-18
- "Using Functions to Plot Model Residuals " on page 9-20

What Down the Residuals Plot Show?

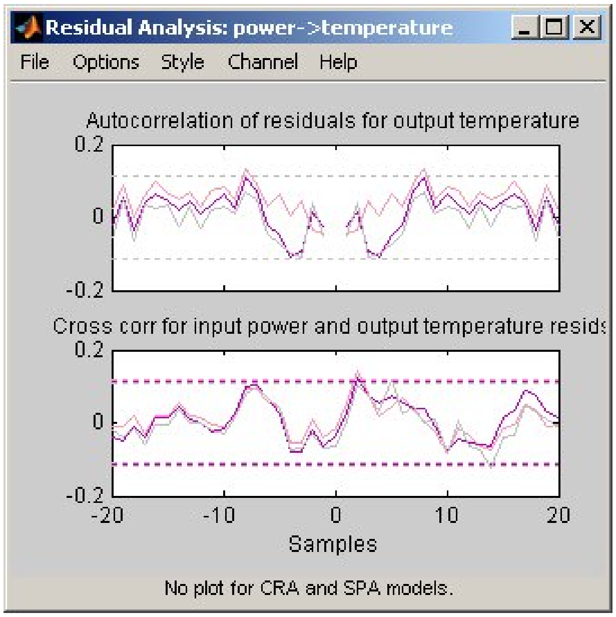
Residual analysis plots show different information depending on whether you use time-domain or frequency-domain input-output validation data.

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.

The following figure shows a sample Residual Analysis plot, created in the System Identification Tool GUI.


Residual Analysis: power->temperature
File Options Style Channel Help
Autocorrelation of residuals for output temperature
0.2
0
-0.2
Cross corr for input power and output temperature resid
0.2
0
-0.2
-20
-10
0
10
20
Samples
No plot for CRA and SPA models.

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, System Identification Toolbox supports only time-domain data.

Overview of Residual Analysis

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 model confidence interval, 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 in the independence tests, except in the following cases:

- For Output-Error 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.

Displaying the Confidence Interval

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 "Plotting Residuals Using the GUI" on page 9-18.

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%.

The *confidence interval* corresponds to the range of residual values with a specific probability of being statistically insignificant for the system. System Identification 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.

Plotting Residuals Using the GUI

To create a residual analysis plot for parametric linear and nonlinear models in the System Identification Tool window, 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 2-29.

To include or exclude a model on the plot, click the corresponding model icon in the System Identification Tool window. 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.

Using Functions to Plot Model Residuals

The following table summarizes functions that generate residual-analysis plots for linear and nonlinear models. For detailed information about this function, see the corresponding reference pages.

Note Apply `pe` and `resid` to one model at a time.

Function	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)`

Impulse and Step Response Plots

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. The step and impulse 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.

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.

This section discusses the following topics:

- "What Does a Transient Response Plot Show?" on page 9-21
- "Displaying the Confidence Interval" on page 9-23
- "Plotting Impulse and Step Response Using the GUI" on page 9-23
- "Using Functions to Plot Impulse and Step Response " on page 9-26

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.

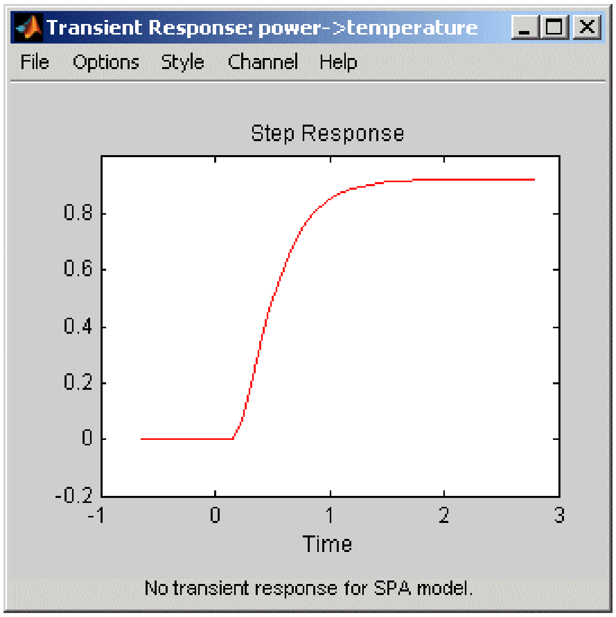
The impulse response of a dynamic model is the output signal that results when the input is an impulse. That is, *u(t)* is zero for all values of *t* except at *t*=0, where *u(0)*=1. In the following difference equation, you can compute the impulse response by setting *y(-T)*=*y(-2T)*=0, *u(0)*=1, and *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 on how to display the transient response of the noise model, see "Plotting Impulse and Step Response Using the GUI" on page 9-23

The following figure shows a sample Transient Response plot, created in the System Identification Tool GUI.


Transient Response: power->temperature
File Options Style Channel Help
Step Response
0.8
0.6
0.4
0.2
0
-0.2
-1
0
1
2
3
Time

No transient response for SPA model.

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 "Plotting Impulse and Step Response Using the GUI" on page 9-23.

The *confidence interval* corresponds to the range of response values with a specific probability of being the actual response of the system. System Identification 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.

Plotting Impulse and Step Response Using the GUI

To create a transient analysis plot in the System Identification Tool window, 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 2-29.

To include or exclude a model on the plot, click the corresponding model icon in the System Identification Tool window. 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 confidence interval. **Note** Only available for linear models.	• To display the dashed lines on either side of the nominal model curve, select **Options > Show confidence intervals**. Select this option again to hide the confidence intervals. • To change the confidence value, select **Options > Set % confidence level**, and choose a value from the list. • To enter your own confidence level, select **Options > Set confidence level > Other**. Enter the value as a probability (between 0 and 1) or as the number of standard deviations of a Gaussian distribution.

Transient Response Plot Settings (Continued)

Action	Command
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 multiinput 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.

Using Functions to Plot Impulse and Step Response

You can plot impulse- and step-response plots using the `impulse` and `step` functions, 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 functions.

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 functions that generate impulse- and step-response plots. For detailed information about each function, see the corresponding reference pages.

Function	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])`

Frequency Response Plots

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.

This section discusses the following topics:

- “What Is a Frequency Response?” on page 9-30
- “What Does a Frequency-Response Plot Show?” on page 9-30
- “Plotting Bode Plots Using the GUI” on page 9-32

- "Using Functions to Create Bode and Nyquist Plots " on page 9-34

What Is a 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 frequency function $G(e^{iwT})$ is scaled by the sampling interval T to make the frequency function periodic with the sampling frequency ${}^{2\pi}\!/_{T}$.

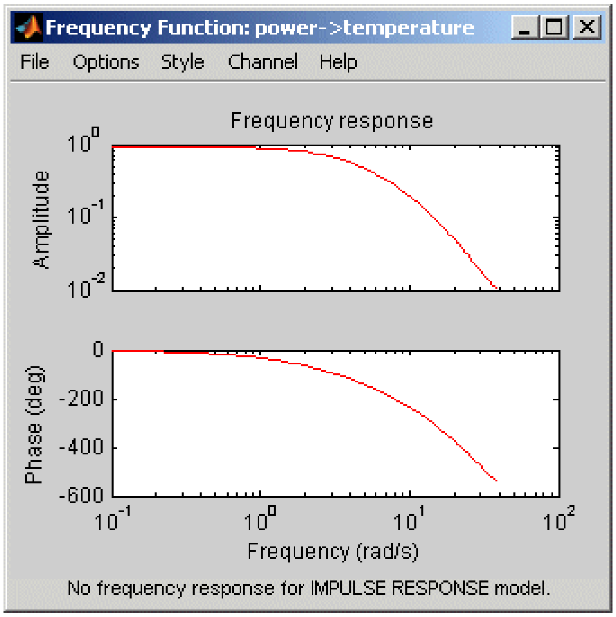
What Does a Frequency-Response Plot Show?

System Identification Tool 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 9-36.
- (Only in 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.



No frequency response for IMPULSE RESPONSE model.

Plotting Bode Plots Using the GUI

To create a frequency-response plot for parametric linear models in the System Identification Tool window, 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 2-29.

To include or exclude a model on the plot, click the corresponding model icon in the System Identification Tool window. Active models display a thick line inside the Model Board icon.

In addition to the frequency-response curve, you can display a confidence interval on the plot. The *confidence interval* corresponds to the range of response values with a specific probability of being the actual response of the system. System Identification 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.

The following table summarizes the Frequency Function plot settings.

Frequency Function Plot Settings

Action	Command
Display confidence interval.	• To display the dashed lines on either side of the nominal model curve, select **Options > Show confidence intervals**. Select this option again to hide the confidence intervals. • To change the confidence value, select **Options > Set % confidence level**, and choose a value from the list. • To enter your own confidence level, select **Options > Set confidence level > Other**. Enter the value as a probability (between 0 and 1) or as the number of standard deviations of a Gaussian distribution.
Change the frequency values for computing the noise spectrum. The default frequency vector is 128 linearly distributed values, greater than zero and less than or equal to the Nyquist frequency.	Select **Options > Frequency range** and specify a new frequency vector in units of rad/s.Enter the frequency vector using any one of following methods: • MATLAB expression, such as `[1:100]*pi/100` or `logspace(-3,-1,200)`. Cannot contain variables in the MATLAB workspace. • Row vector of values, such as `[1:.1:100]` **Note** To restore the default frequency vector, enter `[]`.
Change frequency units between hertz and radians per second.	Select **Style > Frequency (Hz)** or **Style > Frequency (rad/s)**.

Frequency Function Plot Settings (Continued)

Action	Command
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.

Using Functions to Create Bode and Nyquist Plots

You can plot Bode and Nyquist plots for linear models using the `bode`, `ffplot`, and `nyquist` functions.

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 functions.

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 functions that generate Bode and Nyquist plots for linear models. For detailed information about each function and how to specify the frequency values for computing the response, see the corresponding reference pages.

Function	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)`

Noise Spectrum Plots

When you estimate the noise model of your linear system, System Identification Toolbox lets you 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.

This section discusses the following topics:

- "What Does a Noise Spectrum Plot Show?" on page 9-36
- "Displaying the Confidence Interval" on page 9-37
- "Plotting Noise Spectrum Using the GUI" on page 9-38
- "Using Functions to Plot Noise Spectrum" on page 9-41

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. In System Identification Toolbox, the noise term is treated as filtered white noise, as follows:

$$v(t) = H(z)e(t)$$

System Identification 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 in System Identification Toolbox 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 "Plotting Noise Spectrum Using the GUI" on page 9-38.

The *confidence interval* corresponds to the range of power-spectrum values with a specific probability of being the actual noise spectrum of the system. System Identification 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.

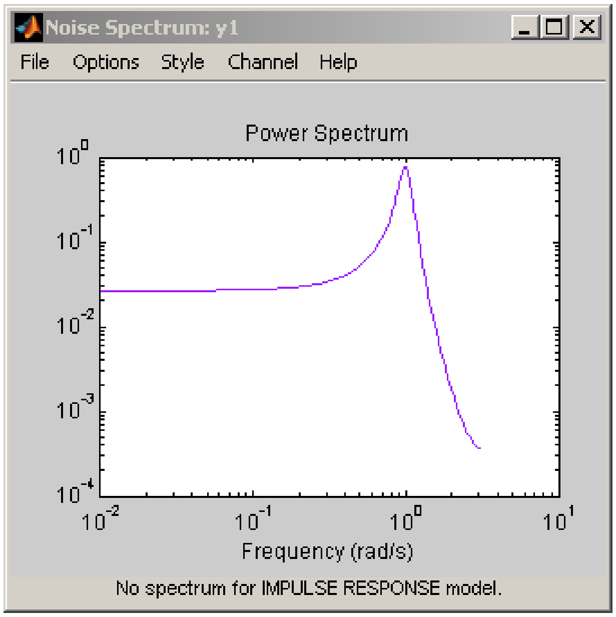
Note The calculation of the confidence interval assumes that the model sufficiently describes the system dynamics and the model residuals pass independence tests.

Plotting Noise Spectrum Using the GUI

To create a noise spectrum plot for parametric linear models in the System Identification Tool window, 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 2-29.

To include or exclude a model on the plot, click the corresponding model icon in the System Identification Tool window. Active models display a thick line inside the Model Board icon.

The following figure shows a sample Noise Spectrum plot.



No spectrum for IMPULSE RESPONSE model.

The following table summarizes the Noise Spectrum plot settings.

Noise Spectrum Plot Settings

Action	Command
Display 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]` **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**.

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.

Using Functions to Plot Noise Spectrum

To plot the frequency-response of the noise model, use a combination of System Identification Toolbox commands.

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 9-36 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);
```

System Identification 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.

Pole-Zero Plots

You can create pole-zero plots of linear polynomial, state-space, and grey-box models.

This section discusses the following topics:

- "What Does a Pole-Zero Plot Show?" on page 9-43
- "Plotting Model Poles and Zeros Using the GUI" on page 9-44
- "Using Functions to Create Pole-Zero Plots" on page 9-46
- "Reducing Model Order Using Pole-Zero Plots" on page 9-46

What Does a Pole-Zero Plot Show?

The general equation of a linear dynamic system is given by:

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

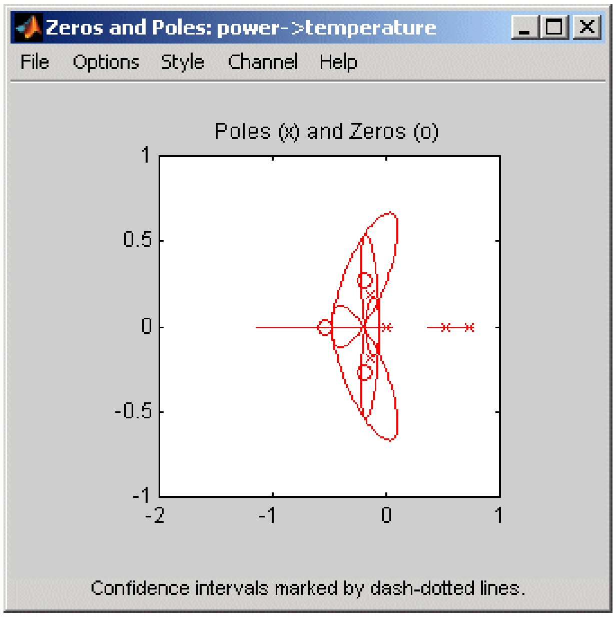
In this equation, G is an operator that takes the input to the output and captures the system dynamics, and v is the additive noise term.

The *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:

$$\begin{aligned}&y(t) - 1.5y(t-T) + 0.7y(t-2T) = \\&\qquad 0.9u(t) + 0.5u(t-T)\end{aligned}$$

The following figure shows a sample pole-zero plot of the model with confidence intervals. x indicate poles and o indicate zeros.


Zeros and Poles: power->temperature
File Options Style Channel Help
Poles (x) and Zeros (o)
1
0.5
0
-0.5
-1
-2
-1
0
1
Confidence intervals marked by dash-dotted lines.

Plotting Model Poles and Zeros Using the GUI

To create a pole-zero plot for parametric linear models in the System Identification Tool window, 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 2-29.

To include or exclude a model on the plot, click the corresponding model icon in the System Identification Tool window. Active models display a thick line inside the Model Board icon.

In addition, you can display a confidence interval for each pole and zero on the plot. 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. System Identification 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.

The following table summarizes the Zeros and Poles plot settings.

Zeros and Poles Plot Settings

Action	Command
Display confidence interval.	• To display the dashed lines on either side of the nominal pole and zero values, select **Options > Show confidence intervals**. Select this option again to hide the confidence intervals. • To change the confidence value, select **Options > Set % confidence level**, and choose a value from the list. • To enter your own confidence level, select **Options > Set confidence level > Other**. Enter the value as a probability (between 0 and 1) or as the number of standard deviations of a Gaussian distribution.
Show real and imaginary axes.	Select **Style > Re/Im-axes**. Select this option again to hide the axes.
Show 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.

Using Functions to Create Pole-Zero Plots

You can create a pole-zero plot for linear polynomial, linear state-space, and linear grey-box models using the `pzmap` function. `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.

The following table provides basic information about `pzmap`. For detailed information about this function, see the corresponding reference pages.

Function	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)`

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 cancellation.

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 "Black-Box Polynomial Models" on page 5-42.

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 “Estimating Nonlinear ARX Models” on page 6-5.

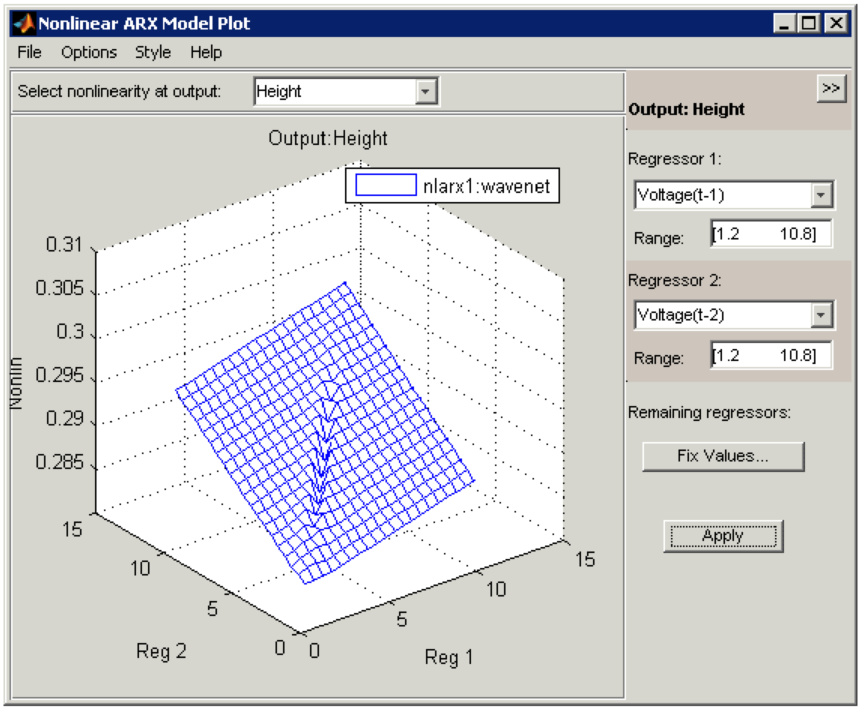
Examining a nonlinear ARX plot can help you gain insight into which regressors have the strongest effect on the model output. Understanding the relative importance of the regressors on the output can help you decide which regressors should be included in the nonlinear function.

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.

To create a nonlinear ARX plot in the System Identification Tool window, 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 2-29.

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.



This section discusses the following topics:

- "Configuring the Nonlinear ARX Plot" on page 9-49
- "Axis Limits, Legend, and 3-D Rotation" on page 9-50
- "Using plot to Create Nonlinear ARX Plots" on page 9-51

Configuring the Nonlinear ARX Plot

To include or exclude a model on the plot, click the corresponding model icon in the System Identification Tool window. Active models display a thick line inside the Model Board icon.

To configure the plot, perform the following steps:

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 plot to Create Nonlinear ARX Plots

You can plot the nonlinearity shape of nonlinear ARX models using the following syntax:

```
plot(model)
```

`model` must be an `idnlarx` model object. 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.

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 9-49 and "Axis Limits, Legend, and 3-D Rotation" on page 9-50.

For detailed information about `plot`, type the following command at the MATLAB prompt:

```
help idnlarx/plot
```

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 "Estimating Hammerstein-Wiener Models" on page 6-35.

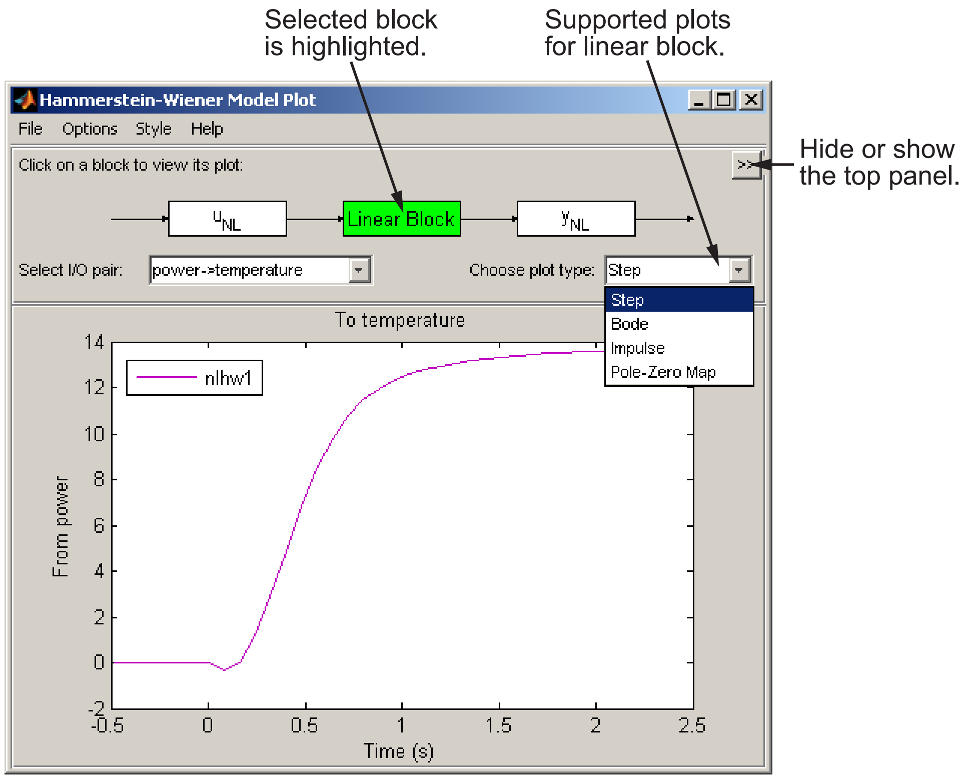
Examining a Hammerstein-Wiener plot can help you determine whether you chose an unnecessarily complicated nonlinearity for modeling your system. For example, if you chose a piecewise-linear nonlinearity (which is very general), but the plot indicates saturation behavior, then you can estimate a new model 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.

To create a Hammerstein-Wiener plot in the System Identification Tool window, 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 2-29.

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 window. Active models display a thick line inside the model icon, as shown in the following figure.


Selected block is highlighted.
Supported plots for linear block.
Hide or show the top panel.
Hammerstein-Wiener Model Plot
File Options Style Help
Click on a block to view its plot:
>>
u_NL
Linear Block
y_NL
Select I/O pair: power->temperature
Choose plot type: Step
Step
Bode
Impulse
Pole-Zero Map
To temperature
nlhw1
From power
Time (s)

This section discusses the following topics:

- “Plotting Nonlinear Block Characteristics” on page 9-54
- “Plotting Linear Block Characteristics” on page 9-55
- “Using plot to Create Hammerstein-Wiener Plots” on page 9-56

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 panel 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 function of the input data, click the u_{NL} block.
   - To plot y_{NL} as a function of its inputs, click the y_{NL} block.

   The selected block is highlighted in green.

   Note An 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 variables, 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, perform the following steps:

1 If the top panel 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 plot to Create Hammerstein-Wiener Plots

You can plot input and output nonlinearity and linear responses for Hammerstein-Wiener models using the following syntax:

```
plot(model)
```

`model` must be an `idnlhw` model object. 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 9-54 and “Plotting Linear Block Characteristics” on page 9-55.

For detailed information about `plot`, type the following command at the MATLAB prompt:

```
help idnlhw/plot
```

Using Akaike's Final Prediction Error and Information Criterion

If you use the same data set to both model estimation and validation, the fit always improves you increase the model order and the flexibility of the model structure increases.

Akaike's Final Prediction Error (FPE) criterion and his closely related Information Criterion (AIC) provide 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 these criteria. According to Akaike's theory, the most accurate model has the smallest FPE and AIC.

This section discusses the following topics:

- "Definition of FPE" on page 9-58
- "Computing FPE" on page 9-59
- "Definition of AIC" on page 9-59
- "Computing AIC" on page 9-60

Definition of FPE

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 estimation data.

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 $\hat{\theta}_N$ represents the estimated parameters.

Computing FPE

Use the `fpe` function to compute Akaike's Final Prediction Error (FPE) criterion for one or more linear or nonlinear models, 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) 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:

$$V = \det\left(\tfrac{1}{N} \sum_{1}^{N} \varepsilon\left(t, \theta_N\right)\left(\varepsilon\left(t, \theta_N\right)\right)^T \right)$$

where $\hat{\theta}_N$ represents the estimated parameters.

For $d << N$

$$AIC = \log\left(V + \left(1 + \frac{2d}{N} \right)\right)$$

Computing AIC

Use the `aic` function 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.

Viewing Model Uncertainty

In addition to estimating model parameters, the System Identification Toolbox estimation 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`.

This section discusses the following topics:

- "What Is Model Covariance?" on page 9-61
- "Viewing Model Uncertainty Information" on page 9-62

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 Plots" on page 9-15. 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.

Viewing 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 MATLAB 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 MATLAB 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 MATLAB 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 "State-Space Models" on page 5-67.

- Simulated output values for linear models with standard deviations using the `sim` command.

  Call the function `sim` 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

During validation, you might find that your model output fits the validation data poorly. You might also find some unexpected or undesirable model characteristics.

This section provides tips for handling the following issues:

- “Model Order Is Too High or Too Low” on page 9-64
- “Nonlinearity Estimator Produces a Poor Fit” on page 9-65
- “Substantial Noise in the System” on page 9-66
- “Unstable Models” on page 9-66
- “Missing Input Variables” on page 9-67
- “Complicated Nonlinearities” on page 9-68

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.

System Identification Toolbox provides assistance in finding the approximate model order, as described in “Estimating Model Orders and Input Delays ” on page 5-49. 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 piecewise-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 Plots” on page 9-15.

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

One of the most conclusive approaches to determining whether a linear model is unstable is by examining the pole-zero plot of the model, which is described in “Pole-Zero Plots” on page 9-43. 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.

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.

One way to check 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.

In the case of linear models, if you believe that your system is stable, but your model is unstable, then you can estimate the model again with a `Focus` setting that guarantees stability. For example, set `Focus` to `Stability` to find the best stable model. This setting might result in a reduced model quality. For more information about `Focus`, see the `Algorithm Properties` reference pages.

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 pages.

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 6, "Estimating Nonlinear Black-Box Models".

Postprocessing and Using Estimated Models

Working with Models After Estimation

After you identify the simplest model that adequately describes your system, you can simulate or predict model output. For more information, see "Simulating and Predicting Model Output" on page 10-13.

For linear parametric models (`idmodel` objects), you can perform the following postprocessing operations, as described in "Converting Linear Models" on page 10-3:

- Transform between continuous-time and discrete-time representation.
- Transform between linear model representations, such as between polynomial, state-space, and zero-pole representations.
- Extract numerical data from the model object, such as transfer function polynomials, model zeros and poles, and state-space matrices.

`idmodel` is a superclass of linear model objects in System Identification Toolbox and defines the shared properties and methods for `idpoly`, `idproc`, `idarx`, `idss`, and `idgrey` objects.

If you have Control System Toolbox, you can import your linear plant model for control-system design. For more information, see "Using Models with Control System Toolbox" on page 10-20.

Finally, if you have Simulink, you can exchange data between System Identification Toolbox and the Simulink simulation environment. For more information, see "Using Models with Simulink" on page 10-26.

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 2, "Working with the System Identification Tool GUI".

Converting Linear Models

System Identification Toolbox lets you convert linear models from one representation to another.

If you used the System Identification Tool GUI to estimate models, you must export the models to the MATLAB Workspace before converting models.

This section discusses the following topics:

- "Transforming Between Discrete-Time and Continuous-Time Representations" on page 10-3
- "Transforming Between Linear Model Representations" on page 10-8
- "Extracting Numerical Data from Linear Models" on page 10-9
- "Extracting Numerical Data for Dynamic Model Versus Noise Model" on page 10-11

Transforming Between Discrete-Time and Continuous-Time Representations

You can use `c2d` and `d2c` to transform any `idmodel` object between continuous-time and discrete-time representations. This capability is useful, for example, if you 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.

This section discusses the following topics:

- "Using c2d, d2c, and d2d Commands" on page 10-4
- "Specifying Intersample Behavior" on page 10-5
- "How d2c Handles Input Delays" on page 10-6
- "Effects on the Noise Model" on page 10-6

Using c2d, d2c, and d2d Commands

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 references pages.

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.

Command	Description	Usage Example
`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 piecewise 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 ith. 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 and the Sampling Interval" on page 5-40.

`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.

For detailed information about each command, see the corresponding reference pages.

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 Control System Toolbox, 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))`

Commands for Transforming Model Representations (Continued)

Command	Model Type to Convert	Usage Example
`idpoly`	Converts single-output `idmodel` object to ARMAX representation. If you have Control System Toolbox, this command converts any single-output LTI object except `frd`.	To get an ARMAX model from state-space model `m_ss`, use the following command: `m_p = idpoly(m_ss)`
`idss`	Converts any single- or multiple-output `idmodel` object to state-space representation. If you have Control System Toolbox, 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)
```

Extracting Numerical Data from Linear Models

System Identification Toolbox lets you extract the numerical parameter values and uncertainties of model objects and store these values using `double` data format. For example, you can extract state-space matrices for state-space models, and extract polynomials for polynomial models. You can operate on extracted model data as you would on any other MATLAB vectors and matrices. You can also pass these numerical values to Control System Toolbox, for example, or Simulink blocks.

If you specified to estimate model uncertainty data, this information is stored in the property `Model.CovarianceMatrix` in the estimated model. The covariance matrix is used to compute uncertainties in parameter estimates, model output plots, Bode plots, residual plots, and pole-zero plots.

The following table summarizes commands for extracting numerical data from models. 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 Numerical Model Data

Command	Description	Syntax
`arxdata`	Extracts ARX parameters from multioutput `idarx` or single-output `idpoly` objects that represent ARX models.	`[A,B,dA,dB] = arxdata(m)`
`freqresp`	Extracts frequency-response data 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)`

Commands for Extracting Numerical Model Data (Continued)

Command	Description	Syntax

`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)`

Extracting Numerical Data for Dynamic Model Versus Noise Model

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 e to the outputs, also called the *noise model*. When you estimate a noise model, System Identification Toolbox includes one noise channel at the input e for each output in your system.

The following table summarizes the results of `ssdata`, `tfdata`, and `zpkdata` commands for extracting the numerical values of the dynamic model and noise model separately. *`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 1-38.

For information about subreferencing noise channels or treating noise channels as measured input, see "Subreferencing Models" on page 1-36.

Note The syntax *`fcn`*`(m('noise'))` is equivalent to *`fcn`*`(m('n'))` .

Syntax for Extracting Transfer-Function 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.
`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*.

Simulating and Predicting Model Output

You can use System Identification Toolbox to simulate and predict model output.

This section discusses the following topics:

For information about simulating identified models in the Simulink environment, see "Using Models with Simulink" on page 10-26.

Simulating Versus Predicting Output

Simulating a model means that you compute the response of a model to a particular input. Simulation does not involve the noise model unless you explicitly specify to compute the response to the noise source input. *Predicting* future outputs of a model from previous data over a time horizon of *k* samples or *kTs* time units—where *Ts* is the sampling interval—requires both past inputs and past outputs.

The main difference between simulation and prediction is whether System Identification Toolbox uses measured or computed previous outputs for calculating the next output.

Simulating models uses only past input values to compute the output values. If the model-output expression includes past outputs, System Identification Toolbox computes the first output value using the initial conditions and the inputs. Then, System Identification Toolbox feeds this computed output into the difference equation or differential equation for calculating the next output value. Thus, no past outputs are used in the computation of output at the current time.

Using a model for prediction is common in controls applications where you want to predict output for a specific number of steps in advance. When you use System Identification Toolbox to *predict* model output, the algorithm uses both the measured and the calculated output data values in the difference equation for computing the next output. The k-step-ahead-predicted values of *y(t)* is computed from all available inputs *u(s)* and relevant previous outputs *y(s)*—where $s \le (t-k)$. The argument s represents the data sample number.

To make sure that the model picks up important dynamic properties, let the predicted time horizon kT be larger than the system time constants, where T is the sampling interval. Prediction with $k=\infty$ means that no previous inputs are used in the computation and prediction matches simulation.

Note Output-error models, state-space models with K set to zero, polynomial models with `na=nc=nd=0`, and nonlinear grey-box models do not use past outputs. In these cases, the simulated and predicted outputs are the same for any prediction horizon k.

Simulation and Prediction Using the System Identification Tool GUI

To learn how to display simulated or predicted output using the System Identification Tool GUI, see the description of the plot settings in “Plotting Model Output Using the GUI” on page 9-10.

Example – Using sim to Simulate Model Output with Noise

This example demonstrates 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:

$$\begin{aligned} y(t) - 1.5y(t-1) + 0.7y(t-2) = \\ u(t-1) + 0.5u(t-2) + e(t) - e(t-1) + 0.2e(t-1) \end{aligned}$$

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.

Example – Using sim to Simulate a Continuous-Time State-Space Model

This example demonstrates how to simulate a continuous-time state-space model with random binary input u and a sampling interval of `0.1` second.

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.

Predicting Model Output

You can use System Identification Toolbox to predict model output.

Use the following syntax to compute `k`-step-ahead prediction of the output signal using model `m`:

```
yhat = predict(m,[y u],k)
```

The predicted value $\hat{y}(t \mid t-k)$ is computed using information in *u(s)* up to time *s*=*t*, and then information in *y(s)* up to time *s*=*t*-*kT*, where *T* is the sampling interval.

The way information in past outputs is used depends on the disturbance model of `m`. For example, because H = `1` in the output-error model, there is no information in past outputs. In this case, predictions and simulations coincide.

The following example demonstrates commands you can use to evaluate how well a time-series model predicts future values. In this case, y is the original series of monthly sales figures. The first half of the measured data is used to estimate the time-series model, and then the second half of the data is used to predict half a year ahead.

```
% Split time-series data into
% two halves
y1 = y(1:48),
y2 = y(49:96)
% Estimate a fourth-order autoregressive model
% using the first half of the data.
m = ar(y1,4)
% Predict time-series output
yhat = predict(m4,y2,6)
% Plot predicted output
plot(y2,yhat)
```

Specifying Initial States

The sim and predict functions require initial states to start the computations.

By default, simulating or predicting output for state-space models uses the initial states stored in the X0 model property after estimation. For multiexperiment state-space models, the stored initial states correspond to the data in the last experiment. These stored initial states might not be appropriate when you simulate or predict model output using new data.

If you prefer to use different initial states for state-space models, or if you are working with other model types, you must specify the initial states for simulation or prediction.

Use the following general syntax for specifying initial states for simulation or prediction:

```
y=sim(model,data,'InitialState',S)
y=predict(model,data,'InitialState',S)
```

where S represents a vector of states.

The following topics discuss how to handle initial states:

- "Set Initial States to Zero" on page 10-18
- "Set Initial States to Equilibrium Values" on page 10-18
- "Estimate Initial States from the Data" on page 10-18

Note The `compare` function automatically estimates the initial states from the data and ensures consistency.

Set Initial States to Zero

If the system starts at rest, or if transient effects are not important, then you can set the initial states to zero.

You can use the following shortcut syntax for setting initial states to zero:

```
y=sim(model,data,'InitialState','z')
y=predict(model,data,'InitialState','z')
```

Set Initial States to Equilibrium Values

If you have physical insight about the starting point of the system, create a vector of specific initial states in the MATLAB Command Window.

Use the following syntax to specify initial states for simulation or prediction:

```
y=sim(model,data,'InitialState',S)
y=predict(model,data,'InitialState',S)
```

where `S` represents a vector of initial states.

If you are working with multiexperiment data, specify `S` as a matrix containing as many columns as there are experiments.

Estimate Initial States from the Data

Simulation or prediction using data from a different experiment than the data used to estimate the model requires the corresponding initial states.

Estimate States for sim. To use `sim` with a data set that differs from the data you used to estimate the model, first estimate the new initial states `Sest` using `pe`:

```
[E,XOest]=pe(model,data)
```

Next, specify the estimated initial states `Sest` as an argument in `sim`. For example:

```
y=sim(model,data,'InitialState',Sest)
```

When you simulate a multiexperiment model, use the `pe` function to estimate initial states for the data from that specific experiment. For example, suppose you estimate a three-state model `M` using a merged data set `Z`, which contains data from five experiments—`z1`, `z2`, `z3`, `z4`, and `z5`:

```
Z = merge(z1,z2,z3,z4,z5);
M = n4sid(Z,3);
```

If you want to simulate using data from `z2`, you must estimate the initial states for the second experiment `Z(z2.u)`, as follows:

```
[E,XOest] = pe(M,getexp(Z,2))
```

where `getexp(Z,2)` gets the data in `z2`. The estimated states matrix `Sest` contains one column of initial-state values for each experiment.

To simulate with these initial states, specify the estimated initial states `Sest` as an argument in `sim`. For example:

```
y=sim(M,getexp(Z,2),'InitialState',Sest)
```

Estimate States for predict. Unlike for `sim`, you can specify to estimate the initial states directly in the `predict` command.

To estimate the initial states that correspond to the data set you use for prediction, use the following syntax:

```
y=predict(M,data,'InitialState','Estimate')
```

Using Models with Control System Toolbox

System Identification Toolbox integrates with Control System Toolbox by providing linear plant identification for control-system design. If you have Control System Toolbox, you can use System Identification Toolbox to identify a linear, time-invariant plant system, and then use Control System Toolbox to design a controller for this plant.

Control System Toolbox also provides the LTI Viewer to extend System Identification Toolbox functionality for linear model analysis.

This section discusses the following topics:

Only linear models are supported in Control System Toolbox. If you identified a nonlinear plant model using System Identification Toolbox, you must linearize it using `linapp` or `lintan` before you can work with this model in Control System Toolbox.

Note You can only use System Identification Toolbox to linearize nonlinear ARX (`idnlarx`) and Hammerstein-Wiener (`idnlhw`) models. Linearization of nonlinear grey-box (`idnlgrey`) models is not supported.

For information about using Control System Toolbox, see the Control System Toolbox documentation.

Using balred to Reduce Model Order

In some cases, your identified model order might be higher than necessary to capture the dynamics. If you have Control System Toolbox, 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 pages. To learn how you can reduce model order using pole-zero plots, see "Reducing Model Order Using Pole-Zero Plots" on page 9-46.

Compensator Design Using Control System Toolbox

After you estimate a plant model in System Identification Toolbox, you can use Control System Toolbox to design a controller for this plant.

System Identification Toolbox models in the MATLAB workspace are immediately available to Control System Toolbox commands in the MATLAB Command Window. 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 provides both the SISO Design Tool GUI and commands for working in the MATLAB Command Window. 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 1-38. To design a controller using Control System Toolbox functions and methods in the MATLAB Command Window, you must convert the plant model to an LTI object. For more information, see "Converting Models to LTI Objects" on page 10-22.

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 functions and methods operate on Control System Toolbox LTI objects. To design a controller for a plant model you estimated in System Identification Toolbox, you must first convert the plant model 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 model objects.

The following table summarizes the commands for transforming linear state-space and polynomial models to LTI model object.

Commands for Converting Models to LTI Objects

Command	Description	Usage 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 command 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 System Identification Toolbox.

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 1-38.

Viewing Model Response in the LTI Viewer

If you have Control Systems Toolbox, you can plot models in the LTI Viewer from either the System Identification Tool window or the MATLAB Command Window.

This section discusses the following topics:

- "What Is the LTI Viewer?" on page 10-23
- "Displaying Identified Models in the LTI Viewer" on page 10-24

For more information about working with plots in the LTI Viewer, see the Control System Toolbox documentation.

What Is the LTI Viewer?

The LTI Viewer is a graphical user interface for viewing and manipulating the response plots of linear models and displays the following plot types:

- Step and impulse responses
- Bode and Nyquist plots

- Nichols plots
- Singular values of the frequency response
- Pole/zero plots
- Response to a general input signal
- Unforced response starting from given initial states (only for state-space models)

Note The LTI Viewer does not display model uncertainty.

Displaying Identified Models in the LTI Viewer

When Control System Toolbox is available, 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 window.

Alternatively, use the following syntax when working in the MATLAB Command Window to view a model in the LTI Viewer:

```
view(model)
```

Combining Model Objects

If you have Control System Toolbox, 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 estimated models:

- `G1+G2`
- `G1*G2`
- `append(G1,G2)`
- `feedback(G1,G2)`

Note These operations lose covariance information.

Example – Using System Identification Toolbox and Control System Toolbox

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)
```

Using Models with Simulink

System Identification Toolbox provides the System Identification block for exchanging information between System Identification Toolbox and the Simulink environment.

You can use the Simulink library to perform the following tasks:

- Import estimated models into a Simulink model and simulate the model with or without noise.

  The model you import might be a subsystem or the main model. For example, if you estimated a plant model using System Identification Toolbox, you can import this plant into Simulink for controller design.
- Stream time-domain data sources (`iddata` objects) into a Simulink model.
- Save data from a simulation in Simulink as a System Identification Toolbox data sink object (`iddata` objects).
- Estimate linear polynomial and state-space models during simulation for single-output data.

Note Simulink supports linear polynomial, linear state-space, linear grey-box, and nonlinear grey-box System Identification Toolbox model objects (`idarx`, `idpoly`, `idproc`, `idss`, `idgrey`, and `idnlgrey` model objects). If you estimated a nonlinear ARX or Hammerstein-Wiener model in System Identification Toolbox, you must linearize this model using `lintan` or `linapp` before importing it into Simulink.

This section discusses the following topics:

- "Opening the System Identification Library" on page 10-27
- "Working with System Identification Blocks" on page 10-27
- "Blocks for Importing Data and Models into Simulink" on page 10-27
- "Blocks for Model Estimation" on page 10-28
- "Example – Using Simulink to Simulate a Multiexperiment Model" on page 10-29

Opening the System Identification Library

To open the System Identification Library window, type the following command at the MATLAB prompt:

```
slident
```

The System Identification Library contains blocks for exchanging information between Simulink and System Identification Toolbox.

Working with System Identification Blocks

You can exchange data between Simulink and System Identification Toolbox by adding System Identification blocks to your Simulink model and specifying block parameters. For more information about creating and running Simulink models, see the discussion on building a model in the *Simulink User's Guide*.

To add a block from the System Identification Library, drag the block into the model window. To specify block parameters, double-click the block icon to open the Block Parameters dialog box.

You can get help on a specific block by right-clicking the block in the Library window and selecting **Help**.

Blocks for Importing Data and Models into Simulink

System Identification blocks let you import and simulate estimated models in Simulink, stream System Identification Toolbox data to Simulink, and save data from a simulation as a System Identification Toolbox object.

After you add a block to the 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 *Getting Started with System Identification Toolbox*. To get help on a specific block, right-click the block in the Library window, and select **Help**.

Block	Description
Iddata Sink	Saves model input and output signals to the MATLAB Workspace as an `iddata` object.
Iddata Source	Imports `iddata` object from the MATLAB Workspace. Input and output ports of the block correspond to input and output channels of the data object. These inputs and outputs provide signals to the next block in a Simulink model.
Idmodel	Imports linear polynomial, state-space, and grey-box models (`idarx`, `idpoly`, `idproc`, `idss`, and `idgrey` model objects) into a Simulink model.
Idnlgrey Model	Imports a nonlinear grey-box model (`idnlgrey` model object) into a Simulink model.

Blocks for Model Estimation

System Identification blocks let you estimate linear polynomial and state-space models during simulation. For information about AR, ARX, ARMAX, Box-Jenkins, and Output-Error models, see "Black-Box Polynomial Models" on page 5-42.

After you add a block to the model, double-click the block to specify block parameters. To get help on a specific block, right-click the block in the Library window, and select **Help**.

Block	Description
AR	Estimates AutoRegressive model as an `idpoly` model object for time-series data, which has one output and no input. Use system output as the input to this block.

Block	Description
ARX	Estimates an AutoRegressive model with an eXternal input as an `idpoly` object for input-output data. Use system input and output as inputs to this block.
ARMAX	Estimates an AutoRegressive Moving Average model with an eXternal input as an `idpoly` object for input-output data. Use system input and output as inputs to this block.
BJ	Estimates a Box-Jenkins model as an `idpoly` object for input-output data. Use system input and output as inputs to this block.
OE	Estimates an Output-Error model as an `idpoly` object for input-output data. Use system input and output as inputs to this block.
PEM	Uses a general prediction-error method to estimate any single-input and single output `idpoly` object, such as ARX, ARMAX, Box-Jenkins, and Output-Error models. Use system input and output as inputs to this block.

Example – Using Simulink to Simulate a Multiexperiment Model

This example demonstrates how to set initial states before simulating a multiexperiment model. For multiexperiment data, Simulink uses the initial states of the last experiment unless you specify otherwise.

Suppose you estimate a three-state model `M` using a merged data set `Z`, which contains data from 5 experiments—`z1`, `z2`, `z3`, `z4`, and `z5`:

```
Z = merge(z1,z2,z3,z4,z5);
```

```
M = n4sid(Z,3);
```

When a model uses several data sets, the initial-states property stores only the estimated states corresponding to the last data set. In this example, `M.X0` is a vector of length `3` (corresponding to the three states of the model). The values of `M.X0` are the estimated state values corresponding to `z5`.

The following procedure describes how to access the initial states that correspond to `z2` for the simulation, where `z2` is a portion of the estimation data `Z`.

To specify the settings of the `idmodel` block in Simulink for comparing the measured output from experiment `z2` with the simulated output:

1 Estimate the initial states using the second experiment as input, that is `Z(z2.u)`, as follows:

   ```
   [E,X0est] = pe(M,getexp(Z,2))
   ```

   In this case, the function `getexp(Z,2)` gets the data in `z2`.

2 In Simulink, open the Function Block Parameters dialog box for the `idmodel` block.

3 In the **idmodel variable** field, type `M` to specify the estimated model.

4 In the **Initial states** field, type `X0est` to specify the estimated initial states.

5 Click **OK**.

Run the simulation to compare the measured output `z2.y` to the simulated output.

Functions — By Category

Data Manipulation

`advice`	Advice about data or estimated linear polynomial and state-space models
`delayest`	Estimate time delay (dead time) from data
`detrend`	Remove trends from output-input data
`diff`	Difference signals in `iddata` objects
`fcat`	Concatenate frequency-domain signals in `idfrd` and `iddata` objects
`feedback`	Identify possible feedback in `iddata` data
`fft`	Transform `iddata` object to frequency domain
`fselect`	Select frequencies from `idfrd` object
`get`	Query properties of data and model objects
`getexp`	Retrieve experiment(s) from multiple-experiment `iddata` objects
`iddata`	Class for storing time-domain and frequency-domain data
`idfilt`	Filter data using user-defined passbands, general filters, or Butterworth filters
`idfrd`	Class for storing frequency-response or spectral-analysis data
`idresamp`	Resample `iddata` object by decimation and interpolation
`ifft`	Transform `iddata` objects from frequency to time domain

`isreal`	Determine whether model or data set contains real parameters or data
`merge (iddata)`	Merge data sets into one `iddata` object
`misdata`	Reconstruct missing input and output data
`nkshift`	Shift data sequences
`nuderst`	Set step size for numerical differentiation
`pexcit`	Level of excitation of input signals
`plot`	Plot `iddata` or model objects
`realdata`	Determine whether `iddata` is based on real-valued signals
`resample`	Resample data by interpolation and decimation
`set`	Set properties of data and model objects

Estimating Linear Parametric Models

`ar`	Estimate parameters of AR model for scalar time series returning `idpoly` object
`armax`	Estimate parameters of ARMAX or ARMA model returning `idpoly` object
`arx`	Estimate parameters of ARX or AR model using least squares returning `idpoly` or `idarx` object
`bj`	Estimate parameters of Box-Jenkins model returning `idpoly` object

`iv4`	Estimate ARX model using four-stage instrumental variable method returning `idpoly` or `idarx` object
`ivar`	Estimate AR model using instrumental variable method returning `idpoly` object
`ivx`	Estimate parameters of ARX model using instrumental variable method with arbitrary instruments returning `idpoly` or `idarx` object
`n4sid`	Estimate state-space models using subspace method returning `idss` object
`oe`	Estimate parameters of output-error model returning `idpoly` object
`pem`	Estimate model parameters using iterative prediction-error minimization method

Estimating Models Recursively

`rarmax`	Estimate recursively parameters of ARMAX or ARMA models
`rarx`	Estimate recursively parameters of ARX or AR models
`rbj`	Estimate recursively parameters of Box-Jenkins models
`roe`	Estimate recursively output-error models (IIR-filters)

`rpem`	Estimate general input-output models using recursive prediction-error minimization method
`rplr`	Estimate general input-output models using recursive pseudolinear regression method
`segment`	Segment data and estimate models for each segment

Estimating Nonlinear Models

`addreg`	Add custom regressors to `idnalrx` model
`customnet`	Store nonlinearity estimator with user-defined unit function for nonlinear ARX and Hammerstein-Wiener models
`customreg`	Store custom regressor for nonlinear ARX models
`deadzone`	Store dead-zone nonlinearity estimator for Hammerstein-Wiener models
`evaluate`	Value of nonlinearity estimator at given input
`getinit`	Values of `idnlgrey` model initial states
`getpar`	Parameter values and properties of `idnlgrey` model parameters
`getreg`	Returns names of standard or custom regressors in nonlinear ARX model

`idnlgrey`	Class for storing nonlinear grey-box models
`linear`	Specify to estimate nonlinear ARX model that is linear in (nonlinear) custom regressors
`neuralnet`	Store neural network object created in Neural Network Toolbox for estimating nonlinear ARX and Hammerstein-Wiener models
`nlarx`	Estimate nonlinear ARX models
`nlhw`	Estimate Hammerstein-Wiener models
`pem`	Estimate model parameters using iterative prediction-error minimization method
`polyreg`	Generate custom regressors by computing powers and products of standard regressors
`pwlinear`	Store piecewise-linear nonlinear estimator for Hammerstein-Wiener models
`saturation`	Store saturation nonlinearity estimator for Hammerstein-Wiener models
`setinit`	Set initial states of `idnlgrey` model object
`setpar`	Set initial parameter values of `idnlgrey` model object
`sigmoidnet`	Store sigmoid network nonlinearity estimator for nonlinear ARX and Hammerstein-Wiener models
`treepartition`	Store binary-tree nonlinearity estimator for nonlinear ARX models

`unitgain`	Specify absence of nonlinearities for specific input or output channels in Hammerstein-Wiener models
`wavenet`	Store wavelet network nonlinearity estimator for nonlinear ARX and Hammerstein-Wiener models

Estimating Nonparametric Models

`covf`	Estimate time-series covariance functions
`cra`	Estimate impulse response using prewhitened-based correlation analysis
`delayest`	Estimate time delay (dead time) from data
`etfe`	Estimate empirical transfer functions and periodograms returning `idfrd` object
`feedback`	Identify possible feedback in `iddata` data
`impulse`	Plot impulse response with confidence interval
`pexcit`	Level of excitation of input signals
`spa`	Estimate frequency response and spectrum using spectral analysis returning `idfrd` object

`spafdr`	Estimate frequency response and spectrum using spectral analysis with frequency-dependent resolution returning `idfrd` object
`step`	Plot step response with confidence interval

General

`advice`	Advice about data or estimated linear polynomial and state-space models
`get`	Query properties of data and model objects
`set`	Set properties of data and model objects
`setpname`	Set mnemonic parameter names for black-box model structures
`size`	Dimensions of `iddata`, `idmodel`, and `idfrd` objects
`timestamp`	Return date and time when object was created or last modified

Graphical User Interface

`ident`	Open System Identification Tool GUI
`midprefs`	Set directory for storing `idprefs.mat` containing GUI startup information

Model Analysis

`advice`	Advice about data or estimated linear polynomial and state-space models
`bode`	Plot Bode diagram of frequency response with confidence interval
`compare`	Compare model output and measured output
`ffplot`	Plot frequency response and spectra
`impulse`	Plot impulse response with confidence interval
`isreal`	Determine whether model or data set contains real parameters or data
`nyquist`	Plot Nyquist curve of frequency response with confidence interval
`plot`	Plot `iddata` or model objects
`present`	Display model information, including estimated uncertainty
`pzmap`	Plot zeros and poles with confidence interval
`step`	Plot step response with confidence interval
`view`	Plot model characteristics using LTI viewer in Control System Toolbox

Model Constructors

`idarx`	Class for storing multioutput ARX polynomials and estimated impulse- and step-response
`idfrd`	Class for storing frequency-response or spectral-analysis data
`idgrey`	Class for storing linear grey-box models
`idmodel`	Superclass for linear models
`idnlarx`	Class for storing nonlinear ARX models
`idnlgrey`	Class for storing nonlinear grey-box models
`idnlhw`	Class for storing Hammerstein-Wiener input-output models
`idnlmodel`	Superclass for nonlinear models
`idpoly`	Class for storing linear polynomial input-output models
`idproc`	Class for storing low-order, continuous-time process models
`idss`	Class for storing linear state-space models with known and unknown parameters

Model Conversion

`arxdata`	ARX parameters with variance information from `idarx` models
`balred`	Reduce model order (requires Control System Toolbox)
`c2d`	Convert model from continuous to discrete time
`d2c`	Convert model from discrete to continuous time
`frd`	Convert `idfrd` objects to frequency-response LTI models of Control System Toolbox
`freqresp`	Compute frequency function for model
`fselect`	Select frequencies from `idfrd` object
`idfrd`	Class for storing frequency-response or spectral-analysis data
`linapp`	Linear approximation of nonlinear ARX and Hammerstein-Wiener models for given input
`lintan`	Tangent linearization of Hammerstein-Wiener models about operating point
`noisecnv`	Convert `idmodel` with noise channels to model with only measured channels
`polydata`	Convert model to input-output polynomials
`ss`	Convert `idmodel` objects of System Identification Toolbox to LTI models of Control System Toolbox
`ssdata`	Convert model to state-space form

`tf`	Convert `idmodel` objects of System Identification Toolbox to transfer-function LTI models of Control System Toolbox
`tfdata`	Convert model to transfer-function form
`zpk`	Convert `idmodel` objects of System Identification Toolbox to state-space LTI models of Control System Toolbox
`zpkdata`	Compute zeros, poles, and gains of transfer-function models

Model Manipulation

`get`	Query properties of data and model objects
`init`	Set or randomize initial parameter values
`merge`	Merge estimated `idmodel` models
`selstruc`	Select model order (structure)
`set`	Set properties of data and model objects
`setstruc`	Set matrix structure for `idss` objects

Model Structure Selection

`arxstruc`	Compute and compare loss functions for single-output ARX models
`ivstruc`	Compute loss functions for sets of output-error model structures
`n4sid`	Estimate state-space models using subspace method returning `idss` object
`selstruc`	Select model order (structure)
`struc`	Generate model structure matrices

Model Uncertainty

`arxdata`	ARX parameters with variance information from `idarx` models
`bode`	Plot Bode diagram of frequency response with confidence interval
`impulse`	Plot impulse response with confidence interval
`nyquist`	Plot Nyquist curve of frequency response with confidence interval
`polydata`	Convert model to input-output polynomials
`pzmap`	Plot zeros and poles with confidence interval
`sim`	Simulate linear models with confidence interval
`simsd`	Simulate models with uncertainty using Monte Carlo method
`ssdata`	Convert model to state-space form

`step`	Plot step response with confidence interval
`tfdata`	Convert model to transfer-function form
`zpkdata`	Compute zeros, poles, and gains of transfer-function models

Model Validation

`aic`	Akaike Information Criterion for estimated model
`arxstruc`	Compute and compare loss functions for single-output ARX models
`compare`	Compare model output and measured output
`fpe`	Akaike Final Prediction Error for estimated model
`pe`	Compute prediction errors associated with model and data set
`predict`	Predict output k steps ahead
`resid`	Compute and test model residuals (prediction errors)
`selstruc`	Select model order (structure)
`sim`	Simulate linear models with confidence interval

Simulation and Prediction

`idinput`	Generate input signals
`idmdlsim`	Simulate `idmodel` objects in Simulink
`pe`	Compute prediction errors associated with model and data set
`predict`	Predict output k steps ahead
`sim`	Simulate linear models with confidence interval

12

Functions–Alphabetical List

addreg

Purpose

Add custom regressors to `idnalrx` model

Syntax

```
addreg(model,R)
addreg(model,R,I)
addreg(model,R1,I1,R2,I2,Rn,In,)
```

Arguments

`model`

Name of the `idnlarx` model object.

`R`

For single-output models, `R` can be an array of `customreg` objects. `R` can also be a cell array of strings, where is string is an expression in terms of input and output variables.

For multiple-output models with `ny` outputs, `R` can be a 1–by-`ny` cell array containing names of `customreg` objects or strings of expressions.

`I`

Scalar integer or vector of integers. Each integer specifies the index of a model output channel.

Description

`addreg(model,R)` adds one or more custom regressors `R` to nonlinear ARX model `model`. For multiple-output systems, each element of the `R` cell array is added to the corresponding output channel of the model.

`addreg(model,R,I)` is used for multiple-output models and adds one or more custom regressors `R` to specific output channels.

`addreg(model,R1,I1,R2,I2,Rn,In,)` is used for multiple-output models and specifies the regressor-channel pairs.

Examples

The following example shows how to add regressors to a nonlinear ARX model using a cell array of strings. `u1` and `y2` are input and output variables, respectively:

```
load iddata1
m1=nlarx(z1,[4 2 1],'wave','nlr',[1:3]);
```

```
% Add regressors using cell array of strings
m2=addreg(m1,{'y1(t-2^2)';'u1(t)*y1(t-7)'})
```

Alternatively, you can use the `customreg` constructor to create regressors, and then add them to the model:

```
r1=customreg(@(x,y)x*y,{'y1','u1'},[2 3])
r2=customreg(@(x,y)x+y,{'y1','u1'},[2 3])
m2=addreg(m1,[r1 r2]);
```

See Also

`customreg`

`getreg`

`nlarx`

`polyreg`

advice

Purpose

Advice about data or estimated linear polynomial and state-space models

Syntax

```
advice(model)
advice(data)
```

Arguments

`model`
: Name of the `idarx`, `idgrey`, `idpoly`, `idproc`, or `idss` model object. These model objects belong to the `idmodel` abstract class, representing linear polynomial and state-space models.

`data`
: Name of the `iddata` object.

Description

`advice(model)` displays the following information about the estimated model in the MATLAB Command Window:

- Does the model capture essential dynamics of the system and the disturbance characteristics?
- Is the model order higher than necessary?
- Is there potential output feedback in the validation data?
- Would a nonlinear ARX model perform better than a linear ARX model?

`advice(data)` displays the following information about the data in the MATLAB Command Window:

- What are the excitation levels of the signals and how does this affects the model orders? See also `pexcit`.
- Does it make sense to remove constant offsets and linear trends from the data? See also `detrend`.
- Is there an indication of output feedback in the data? See also `feedback`.

See Also

`detrend`

`feedback`

`iddata`

`pexcit`

Purpose

Akaike Information Criterion for estimated model

Syntax

```
am = aic(model)
am = aic(model1,model2,...)
```

Arguments

`model`

Name of an `idarx`, `idgrey`, `idpoly`, `idproc`, or `idss` model object. These model objects belong to the `idmodel` abstract class.

Description

`am = aic(model)` returns a scalar value of the Akaike's Information Criterion (AIC) for the estimated `model`.

`am = aic(model1,model2,...)` returns a row vector containing AIC values for the estimated models `model1,model2,...`.

Remarks

Use Akaike Information Criterion (AIC) to perform a relative comparison of models with different structures. Smaller value of AIC indicates a better model.

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.

For $d << N$

$$AIC = \log\left(V + \left(1 + \frac{2d}{N}\right)\right)$$

The loss function V is

$$V = \det\left(\tfrac{1}{N}\sum_{1}^{N}\varepsilon\left(t,\theta_N\right)\left(\varepsilon\left(t,\theta_N\right)\right)^T\right)$$

where $\hat{\theta}_N$ represents the estimated parameters.
AIC is formally defined as the negative log-likelihood function Λ, evaluated at the estimated parameters, plus the number of estimated parameters. You can derive AIC from this definition, as follows:

If the disturbance source is Gaussian with the covariance matrix Λ, the logarithm of the likelihood function is

$$L(\theta,\Lambda) = -\tfrac{1}{2}\sum_{1}^{N}\varepsilon(t,\theta)^T\Lambda^{-1}\varepsilon(t,\theta) - \tfrac{N}{2}\log\left(\det\Lambda\right) + const$$

Maximizing this analytically with respect to Λ, and then maximizing the result with respect to θ, gives

$$L(\theta, \Lambda) = const + \frac{Np}{2} + \frac{N}{2}\log(V)$$

where p is the number of outputs.

To obtain the AIC expression from the last result, remove the constants and normalize.

References

Ljung, L. *System Identification: Theory for the User*, Upper Saddle River, NJ, Prentice-Hal PTR, 1999. See sections about the statistical framework for parameter estimation and maximum likelihood method and comparing model structures.

See Also

`EstimationInfo`

`fpe`

Algorithm Properties

Purpose

Algorithm properties affecting estimation process for linear models

Syntax

```
idprops algorithm
m.algorithm.PropertyName='PropertyValue'
```

Description

`Algorithm` is a property of the `idmodel` abstract class that specifies the estimation algorithm. The `idmodel` subclasses are linear models that you actually work with in System Identification Toolbox, such as `idarx`, `idss`, `idpoly`, `idproc`, and `idgrey`, inherit this property.

Property names are not case sensitive. When you type a property name, you only need to enter enough characters to uniquely identify the property.

For a model `m`, you can retrieve the fields of this property using the `get` method. For example, `get(m,'MaxIter')`.

You can also set the `SearchDirection` property of a model using dot notation. For example, `m.SearchDirection = 'gn'`.

When you create a new model by refining an existing model `m`, the algorithm properties of `m` are inherited by the new model.

Note You can estimate a model with specific algorithm settings and define a structure variable to store the algorithm values. For example:

```
model = n4sid(data,order)
myalg = model.Algorithm;
myalg.Focus='Simulation';
m = pem(data,model,'alg',myalg)
```

The fields of the `Algorithm` structure are as follows:

Properties for All Estimation Methods

- `Focus`: This property defines how the errors e between the measured and the modeled outputs are weighed at specific frequencies during the minimization of the loss function $V = \sum_i \lambda_i e_i^2$. Higher weighting at specific frequencies emphasizes the requirement for a good fit at these frequencies. `Focus` can have the following values:
  - `'Prediction'`: (Default) Automatically calculates the weighting function as a product of the input spectrum and the inverse of the noise model. This minimizes the one-step-ahead prediction, which typically favors fitting small time intervals (high frequency range). From a statistical-variance point of view, this is the optimal weighting function. However, this method neglects the approximation aspects (bias) of the fit. Might not result in a stable model. Use `'Stability'` when you want to ensure a stable model.
  - `'Simulation'`: Estimates the model using the frequency weighting of the transfer function that is given by the input spectrum. Typically, this method favors the frequency range where the input spectrum has the most power. In other words, the resulting model will produce good simulations for inputs that have the same spectra as used for estimation. For models that have no disturbance model, there is no difference between `'Simulation'` and `'Prediction'`. In this case, $y = Gu + He$ with H=`1`, which is equivalent to `A=C=D=1` for `idpoly` models and `K = 0` for `idss` models.

    For models that have a disturbance model, G is first estimated with H=`1`, and then H is estimated using a prediction-error method with a fixed estimated transfer function $\hat{G}$. This guarantees a stable transfer function G.
  - `'Stability'`: This weighting is the same as for `'Prediction'`, but the model is forced to be stable. Use only when you know the

system is stable. In some cases, forcing the model to be stable can result in a bad model.

- Enter a row vector or a matrix containing frequency values that define desired passbands. For example:

  ```
  [wl,wh]
  [w1l,w1h;w2l,w2h;w3l,w3h;...]
  ```

  where `wl` and `wh` represent upper and lower limits of a passband. For a matrix with several rows defining frequency passbands, the algorithm uses union of frequency ranges to define the estimation passband.

- Enter any SISO linear filter in any of the following ways:

  A single-input-single-output (SISO) `idmodel` object.

  An `ss`, `tf`, or `zpk` model from Control System Toolbox.

  Using the format `{A,B,C,D}`, which specifies the state-space matrices of the filter.

  Using the format `{numerator, denominator}` , which specifies the numerator and denominator of the filter transfer function

  This calculates the weighting function as a product of the filter and the input spectrum to estimate the transfer function from input to output, *G*. To obtain a good model fit for a specific frequency range, you must choose the filter with a passband in this range. After estimating *G*, the algorithm computes the disturbance model using a prediction-error method and keeping the estimated transfer function fixed (similar to the `'Simulation'` case). For a model that has no disturbance model, the estimation result is the same if you first prefilter the data using `idfilt`.

- For frequency-domain data only, enter a column vector of weights for `'Focus'`. This vector must have the same size as length of the frequency vector of the data set, `Data.Frequency`. Each input and output response in the data is multiplied by the corresponding weight at that frequency.

- `MaxSize`: A positive integer, specified such that the input-output data is split into segments where each contains fewer than `MaxSize` elements. Setting `MaxSize` can improve computational performance. The default value of `MaxSize` is `'Auto'`, which uses the M-file `idmsize` to set the value. You can edit this file to optimize computational speed on a particular computer. `MaxSize` does not affect the numerical properties of the estimate except when you use `InitialState = 'backcast'` for frequency-domain data. In this case, the frequency ranges where backcasting takes place might depend on `MaxSize` and affects estimates.
- `FixedParameter`: Vector of integers containing the indices of parameters that are not estimated and remain fixed at nominal or initial values. Parameter indices refer to their position in the list, stored in the property `'ParameterVector'`. You can also specify parameter names as values from the property `'PName'`. To specify fixed parameters using parameter names, enter `Fixedparameter` as a cell array of strings. For example, to fix parameters with names `'a'` and `'b'`, type `m.FixedParameter = {'a','b','c'}`. Strings can contain wildcards, such as `'*'` to specify the automatic completion of a string, or `'?'` to represent an arbitrary character. For example, to fix three parameters in a disturbance model that start with `'k'`, such as `'k1'`, `'k2'`,`'k3'`, use `FixedParameter = {'k*'}`. For structured state-space models, you can fix and free parameters by specifying structure matrices, such as `As` and `Bs` (see `idss`).

Note By default, the property `'PName'` is empty. Use `setpname` to assign default parameter names. For example, `m = setpname(m)`.

Properties for n4sid, Estimating State-Space Models

Note These properties also apply to `pem`, used for estimating black-box state-space models that are initialized by the `n4sid` estimate.

- `N4Weight`: Calculates the weighting matrices used in the singular-value decomposition step of the algorithm and has three possible values:
  - `'Auto'`: (Default) Automatically chooses between `'MOESP'` and `'CVA'`
  - `'MOESP'`: Uses the MOESP algorithm by Verhaegen.
  - `'CVA'`: Uses the canonical variable algorithm by Larimore.

  For more information about setting this property, see the `n4sid` reference page.
- `N4Horizon`: Determines the forward and backward prediction horizons used by the algorithm. Enter a row vector with three elements: `N4Horizon=[r sy su]`, where `r` is the maximum forward prediction horizon; that is, the algorithm uses up to `r` step-ahead predictors. `sy` is the number of past outputs, and `su` is the number of past inputs used for predictions. For an exact definition of these integers, see the section about subspace methods in [2], where they are called *r*, *s1*, and *s2*. These numbers can have a substantial influence on the quality of the resulting model and there are no simple rules for choosing them. Making `'N4Horizon'` a `k`-by-3 matrix means that the algorithm tries each row of `'N4Horizon'` and selects the value that gives the best (prediction) fit to the data. Choosing the best row is not available when you also specify to select the best model order. When you specify one column in `'N4Horizon'`, the interpretation is `r=sy=su`. The default choice is `'N4Horizon' = 'Auto'`, which uses an Akaike Information Criterion (AIC) for the selection of `sy` and `su`.

Properties for Iterative Estimation Methods armax, bj, oe, and pem

- `Trace`: This property specifies what information displays in the MATLAB Command Window about the iterative search during estimation.
  - `'Trace'='Off'`: Displays no information.

  - `'Trace'='On'`: Displays the loss-function values for each iteration.
  - `'Trace'='Full'`: Displays the same information as 'On' and also include the current parameter values and the search direction (except when the `Advanced SSParameterization` model property is set to `'Free'` for `idss` models and the list of parameters can change between iterations).

- `LimitError`: Specifies when to adjust the weight of large errors from quadratic to linear. Errors larger than `LimitError` times the estimated standard deviation have a linear weight in the criteria. The default value of `LimitError` is `1.6`. `LimitError = 0` disables the robustification and leads to a purely quadratic criterion. The standard deviation is estimated robustly as the median of the absolute deviations from the median and divided by `0.7`. (See the section about choosing a robust norm in [2].) When estimating with frequency-domain data, `LimitError` is set to zero.
- `MaxIter`: Specifies the maximum number of iterations during loss-function minimization. The iterations stop when `MaxIter` is reached or another stopping criterion is satisfied, such as the `Tolerance`. The default value of `MaxIter` is `20`. Setting `MaxIter = 0` returns the result of the startup procedure. Use `EstimationInfo.Iterations` to get the actual number of iterations during an estimation.
- `Tolerance`: Specifies the minimum percentage difference (divided by 100) between the current value of the loss function and its expected improvement after the next iteration: When the percentage of expected improvement is less than `Tolerance`, the iterations are stopped. Default value is `0.01`. The estimate of the expected loss-function improvement at the next iteration is made based on the Gauss-Newton vector computed for the current parameter value.
- `SearchDirection`: The direction of a line search for finding a lower value of the criterion function. It has the following values:
  - `'gn'`: The Gauss-Newton direction (inverse of the Hessian times the gradient direction). If there is no improvement along this direction, the gradient direction is also tried.

- `'gns'`: A regularized version of the Gauss-Newton direction. Eigenvalues less than `GnsPinvTol` of the Hessian are neglected and the Gauss-Newton direction is computed in the remaining subspace.
- `'gna'`: An adaptive version of `gns`, suggested by Wills and Ninness (IFAC World congress, Prague 2005). Eigenvalues less than `gamma*max(sv)` of the Hessian are neglected , where `sv` are the singular values of the Hessian. The Gauss-Newton direction is computed in the remaining subspace. `gamma` has the initial value `InitGnaTol` (see below) and is increased by the factor `LmStep` each time the search fails to find a lower value of the criterion in less than 5 bisections. It is decreased by the factor `2LmStep` each time a search is successful without any bisections.
- `'lm'`: Uses the Levenberg-Marquardt method. This means that the next parameter value is `-pinv(H+d*I)*grad` from the previous one, where `H` is the Hessian, `I` is the identity matrix, and `grad` is the gradient. `d` is a number that is increased until a lower value of the criterion is found.
- `'Auto'`: A choice among the above is made in the algorithm. This is the default choice.

• `Advanced`: This is a structure that specifies advanced algorithm options and has the following fields:

- `Search`: Uses the following fields to specify options for the iterative search:

a `GnsPinvTol`: Tolerance for the pseudoinverse, used to compute the `gns` direction. See `SearchDirection` for description of `gns`. Default is `10^-9`.

b `InitGnaTol`: The initial value of `gamma` in the `gna` search algorithm. See `SearchDirection` for description of `gna`. Default is `10^-4`.

c `LmStep`: The size of the Levenberg-Marquardt step. The next value of the search-direction length `d` in the Levenberg-Marquardt method is `LmStep` times the previous one. Default is `2`.

d `StepReduction`: For search directions other than the Levenberg-Marquardt direction, the step is reduced by the factor `StepReduction` after each iteration. Default is `2`.

e `MaxBisection`: The maximum number of bisections used by the line search along the search direction. Default is `25`.

f `LmStartValue`: The starting value of search-direction length `d` in the Levenberg-Marquardt method. Default is `0.001`.

g `RelImprovement`: The iterations are stopped if the relative improvement of the criterion is less than `RelImprovement`. Default is `RelImprovement = 0`. This property is different from `Tolerance` in that it uses the actual improvement of the loss function, as opposed to the expected improvement.

- `Threshold`: Contains fields with thresholds for several tests:

a `Sstability`: Specifies the location of the rightmost pole to test the stability of continuous-time models. A model is considered stable when its rightmost pole is to the left of `Sstability`. Default is `0`.

b `Zstability`: Specifies the maximum distance of all poles from the origin to test stability of discrete-time models. A model is considered stable if all poles are within the distance `Zstability` from the origin. Default is `1.01`.

- `AutoInitialState`: Specifies when to automatically estimate the initial state. When `InitialState = 'Auto'`, the initial state is estimated when the ratio of the prediction-error norm with a zero initial state to the norm with an estimated initial state exceeds `AutoInitialState`. Default is `1.2`.

References

[1] Dennis, J.E., Jr., and R.B. Schnabel, *Numerical Methods for Unconstrained Optimization and Nonlinear Equations*, Prentice Hall, Englewood Cliffs, N.J., 1983. See about iterative minimization.

[2] Ljung, L. *System Identification: Theory for the User*, Upper Saddle River, NJ, Prentice-Hal PTR, 1999. See chapter about computing the estimate.

See Also

`armax`

`bj`

`EstimationInfo`

`idpoly`

`idss`

`n4sid`

`oe`

`pem`

Purpose

Estimate parameters of AR model for scalar time series returning `idpoly` object

Syntax

```
m = ar(y,n)
[m,refl] = ar(y,n,approach,window)
[m,refl] = ar(y,n,approach,window,'P1',V1,...,'PN',VN)
```

Arguments

`y`

`iddata` object that contains the time-series data (one output channel).

`n`

Scalar that specifies the order of the model you want to estimate (the number of *A* parameters in the AR model).

`approach`

Lets you choose the algorithm for computing the least squares AR model from the following options:

- `'burg'`: Burg's lattice-based method. Solves the lattice filter equations using the harmonic mean of forward and backward squared prediction errors.
- `'fb'`: (Default) Forward-backward approach. Minimizes the sum of a least- squares criterion for a forward model, and the analogous criterion for a time-reversed model.
- `'gl'`: Geometric lattice approach. Similar to Burg's method, but uses the geometric mean instead of the harmonic mean during minimization.
- `'ls'`: Least-squares approach. Minimizes the standard sum of squared forward-prediction errors.
- `'yw'`: Yule-Walker approach. Solves the Yule-Walker equations, formed from sample covariances.

`window`

Lets you specify how to use information about the data outside the measured time interval (past and future values). The following windowing options are available:

- `'now'`: (Default) No windowing. This value is the default except when the `approach` argument is `'yw'`. Only measured data is used to form regression vectors. The summation in the criteria starts at the sample index equal to `n+1`.
- `'pow'`: Postwindowing. Missing end values are replaced with zeros and the summation is extended to time `N+n` (`N` is the number of observations).
- `'ppw'`: Pre- and postwindowing. Used in the Yule-Walker approach.
- `'prw'`: Prewindowing. Missing past values are replaced with zeros so that the summation in the criteria can start at time equal to zero.

`'P1',V1,...,'PN',VN`

Pairs of property names and property values can include any of the following:

Property Name	Property Value	Description
`'Covariance'`	• `'None'` Suppresses the calculation of the covariance matrix. • [] Empty. • Square matrix containing covariance values of size equal to the length of the parameter vector	Specifies calculation of uncertainties in parameter estimates.

Property Name	Property Value	Description
`'MaxSize'`	Integer	See `Algorithm Properties` for the description.
`'Ts'`	Real positive number	Sets the sampling time and overrides the sampling time of `y`.

Description

Note Use for scalar time series only. For multivariate data, use `arx`.

`m = ar(y,n)` returns an `idpoly` model `m`.

`[m,refl] = ar(y,n,approach,window)` returns an `idpoly` model `m` and the variable `refl`. For the two lattice-based approaches, `'burg'` and `'gl'`, `refl` stores the reflection coefficients in the first row, and the corresponding loss function values in the second row. The first column of `refl` is the zeroth-order model, and the `(2,1)` element of `refl` is the norm of the time series itself.

`[m,refl] = ar(y,n,approach,window,'P1',V1,...,'PN',VN)` returns an `idpoly` model `m` and the variable `refl` using additional windowing criteria.

Remarks

The AR model structure is given by the following equation:

$$A(q)y(t) = e(t)$$

AR model parameters are estimated using variants of the least-squares method. The following table summarizes the common names for methods with a specific combination of `approach` and `window` argument values.

Method	Combination of `approach` and `window` values
Modified Covariance Method	(Default) Forward-backward approach and no windowing.
Correlation Method	Yule-Walker approach, which corresponding to least squares plus pre- and postwindowing.
Covariance Method	Least squares approach with no windowing. `arx` uses this routine.

Examples

Given a sinusoidal signal with noise, compare the spectral estimates of Burg's method with those found from the forward-backward approach and no-windowing method on a Bode plot.

```
y = sin([1:300]') + 0.5*randn(300,1);
y = iddata(y);
mb = ar(y,4,'burg');
mfb = ar(y,4);
bode(mb,mfb)
```

References

Marple, Jr., S.L., *Digital Spectral Analysis with Applications*, Prentice Hall, Englewood Cliffs, 1987, Chapter 8.

See Also

`Algorithm Properties`

`arx`

`EstimationInfo`

`etfe`

`idpoly`

`ivar`

`pem`

`spa`

`step`

Purpose

Estimate parameters of ARMAX or ARMA model returning `idpoly` object

Syntax

```
m = armax(data,orders)
m = armax(data,orders,'P1',V1,...,'PN',VN)
m = armax(data,'na',na,'nb',nb,'nc',nc,'nk',nk)
```

Arguments

`data`

`iddata` object that contains the input-output data.

`orders`

Vector of integers, specified using the format

```
orders = [na nb nc nk]
```

For multiinput systems, `nb` and `nk` are row vectors where the `i`th element corresponds to the order and delay associated with the `i`th input.

When `data` is a time series, which has no input and one output, then

```
orders = [na nc]
```

Note When refining an estimated model `mi`, set the model orders as follows:

```
orders = mi
```

`'na',na,'nb',nb,'nc',nc,'nk',nk`

`'na'`, `'nb'`, and `'nc'` are orders of the ARMAX model. `nk` is the delay. `na`, `nb`, `nc`, and `nk` are the corresponding integer values.

`'P1',V1,...,'PN',VN`

Pairs of property names and property values can include any of the following `idmodel` properties:

`'Focus'`, `'InitialState'`, `'Trace'`, `'MaxIter'`, `'Tolerance'`, `'LimitError'`, and `'FixedParameter'`.

See `Algorithm Properties`, `idpoly`, and `idmodel` for more information.

Description

Note `armax` only supports time-domain data with single or multiple inputs and single output. For frequency-domain data, use `oe`. For the multioutput case, use ARX or a state-space model (see `n4sid` and `pem`).

`m = armax(data,orders)` returns an `idpoly` model `m` with estimated parameters and covariances (parameter uncertainties). Estimates the parameters using the prediction-error method and specified `orders`.

`m = armax(data,orders,'P1',V1,...,'PN',VN)` returns an `idpoly` model `m`. Use additional property-value pairs to specify the estimation algorithm properties.

`m = armax(data,'na',na,'nb',nb,'nc',nc,'nk',nk)` returns an `idpoly` model `m` with orders and delays specified as parameter-value pairs.

Remarks

The ARMAX model structure is

$$\begin{aligned}
&y(t) + a_1 y(t-1) + \ldots + a_{n_a} y(t-n_a) = \\
&\quad b_1 u(t-1) + \ldots + b_{n_b} u(t-n_k-n_b+1) + \\
&\qquad c_1 u(t-1) + \ldots + c_{n_c} u(t-n_c) + e(t)
\end{aligned}$$

A more compact way to write the difference equation is

$$A(q)y(t) = B(q)u(t-n_k) + C(q)e(t)$$

where

- $y(t)$ — Output at time t.
- n_a — Number of poles.
- n_b — Number of zeroes plus 1.
- n_c — Number of C coefficients.
- n_k — Number of input samples that occur before the input affects the output, also called the *dead time* in the system. For discrete systems with no dead time, there is a minimum 1–sample delay because the output depends on the previous input and $n_k = 1$.
- $y(t-1)\ldots y(t-n_a)$ — Previous outputs on which the current output depends.
- $u(t-1)\ldots u(t-n_k-n_b+1)$ — Previous and delayed inputs on which the current output depends.
- $e(t)$ — White-noise disturbance value.

The parameters `na`, `nb`, and `nc` are the orders of the ARMAX model, and `nk` is the delay. q is the delay operator. Specifically,

$$A(q) = 1 + a_1 q^{-1} + \ldots + a_{n_a} q^{-n_a}$$

$$B(q) = b_1 + b_2 q^{-1} + \ldots + b_{n_b} q^{-n_b+1}$$

$$C(q) = 1 + c_1 q^{-1} + \ldots + c_{n_c} q^{-n_c}$$

If `data` is a time series, which has no input channels and one output channel, then `armax` calculates an ARMA model for the time series

$$A(q)y(t) = e(t)$$

In this case

```
orders = [na nc]
```

Algorithm

An iterative search algorithm with the properties `'SearchDirection'`, `'MaxIter'`, `'Tolerance'`, and `'Advanced'` minimizes a robustified quadratic prediction error criterion is minimized. The iterations are terminated either when `MaxIter` is reached, or when the expected improvement is less than `Tolerance`, or when a lower value of the criterion cannot be found. You can get information about the search criteria using `m.EstimationInfo`.

When you do not specify initial parameter values for the iterative search in `orders`, they are constructed in a special four-stage LS-IV algorithm.

The cutoff value for the robustification is based on the property `LimitError` and on the estimated standard deviation of the residuals from the initial parameter estimate. It is not recalculated during the minimization.

To ensure that only models corresponding to stable predictors are tested, the algorithm performs a stability test of the predictor. Generally, both $C(q)$ and $F(q)$ (if applicable) must have all zeros inside the unit circle.

Minimization information is displayed on the screen when the property `'Trace'` is `'On'` or `'Full'`. With `'Trace' ='Full'`, both the current and the previous parameter estimates are displayed in column-vector form, listing parameters in alphabetical order. Also, the values of the criterion function are given and the Gauss-Newton vector and its norm are also displayed. With `'Trace' = 'On'` only the criterion values are displayed.

References

Ljung, L. *System Identification: Theory for the User*, Upper Saddle River, NJ, Prentice-Hal PTR, 1999. See chapter about computing the estimate.

See Also

`Algorithm Properties`

`EstimationInfo`

`idpoly`

`pem`

Purpose

Estimate parameters of ARX or AR model using least squares returning `idpoly` or `idarx` object

Syntax

```
m = arx(data,orders)
m = arx(data,orders,'P1',V1,...,'PN',VN)
m = arx(data,'na',na,'nb',nb,'nc',nc,'nk',nk)
```

Arguments

`data`

An `iddata` object, an `frd` object, or an `idfrd` frequency-response-data object.

`orders`

Vector of integers, specified using the format

```
orders = [na nb nk]
```

For multiinput systems, `nb` and `nk` are row vectors where the `i`th element corresponds to the order and delay associated with the `i`th input.

When `data` is a time series, which has no input and one output, then

```
orders = [na]
```

Note When refining an estimated model `mi`, set the model orders as follows:

```
orders = mi
```

`'na',na,'nb',nb,'nc',nc,'nk',nk`

`'na'`, `'nb'`, and `'nc'` are orders of the ARMAX model. `nk` is the delay. `na`, `nb`, `nc`, and `nk` are the corresponding integer values.

`'P1',V1,...,'PN',VN`
: Pairs of property names and property values can include any of the following `idmodel` properties:

  `'Focus'`, `'InitialState'`, `'Trace'`, `'MaxIter'`, `'Tolerance'`, `'LimitError'`, and `'FixedParameter'`.

  See `Algorithm Properties`, `idpoly`, and `idmodel` for more information.

Description

Note `arx` does not support multioutput continuous-time models. Use state-space model structure instead. When the true noise term $e(t)$ in the ARX model structure is not white noise and `na` is nonzero, the model estimate is incorrect. In this case, use `armax`, `bj`, `iv4`, or `oe`.

`m = arx(data,orders)` returns a model `m` with estimated parameters and covariances (parameter uncertainties). For single-output data, the model is an `idpoly` object. For multioutput data, the model is an `idarx` object. Uses the least-squares method and specified `orders`.

`m = arx(data,orders,'P1',V1,...,'PN',VN)` returns a model `m`. Use additional property-value pairs to specify the estimation algorithm properties.

`m = arx(data,'na',na,'nb',nb,'nc',nc,'nk',nk)` returns a model `m` with orders and delays specified as parameter-value pairs.

Remarks

`arx` uses the least-squares method to estimate the parameters of the ARX model structure:

$$\begin{aligned} & y(t) + a_1 y(t-1) + \ldots + a_{n_a} y(t-n_a) = \\ & \quad b_1 u(t-1) + \ldots + b_{n_b} u(t-n_k-n_b+1) + e(t) \end{aligned}$$

The parameters `na` and `nb` are the orders of the ARX model, and `nk` is the delay.

- $y(t)$ — Output at time t.
- n_a — Number of poles.
- n_b — Number of zeroes plus 1.
- n_k — Number of input samples that occur before the input affects the output, also called the *dead time* in the system. For discrete systems with no dead time, there is a minimum 1–sample delay because the output depends on the previous input and $n_k = 1$.
- $y(t-1)\ldots y(t-n_a)$ — Previous outputs on which the current output depends.
- $u(t-1)\ldots u(t-n_k-n_b+1)$ — Previous and delayed inputs on which the current output depends.
- $e(t)$ — White-noise disturbance value.

A more compact way to write the difference equation is

$$A(q)y(t) = B(q)u(t-n_k) + e(t)$$

q is the delay operator. Specifically,

$$A(q) = 1 + a_1 q^{-1} + \ldots + a_{n_a} q^{-n_a}$$

$$B(q) = b_1 + b_2 q^{-1} + \ldots + b_{n_b} q^{-n_b+1}$$

Time Series Models

For time-series data that contains no inputs, one output and `orders = na`, the model has AR structure of order `na`.

The AR model structure is

$$A(q)y(t) = e(t)$$

Multiple Inputs and Single-Output Models

For multiinput systems, `nb` and `nk` are row vectors where the `i`th element corresponds to the order and delay associated with the `i`th input.

$$A(q)y(t) = B_1(q)u_1(t - n_{k1}) + \ldots + B_{nu}(q)u_{nu}(t - n_{k_{nu}}) + e(t)$$

Multioutput Models

For models with multiple inputs and multiple outputs, `na`, `nb`, and `nk` contain one row for each output signal.

In the multioutput case, `arx` minimizes the trace of the prediction error covariance matrix, or the norm

$$\sum_{t=1}^{N} e^T(t)e(t)$$

To transform this to an arbitrary quadratic norm using a weighting matrix `Lambda`

$$\sum_{t=1}^{N} e^T(t)\Lambda^{-1}e(t)$$

use the syntax

```
m = arx(data,orders,'NoiseVariance', Lambda)
```

You can use `arx` to refine an existing model `m_initial` as an argument.

```
m = arx(data,m_initial)
```

The new model m uses the orders and the weighting matrix for the prediction errors from `m_initial`. You can further modify `m_initial` by adding a list of property name and value pairs as arguments.

This is especially useful when some parameters must be fixed using `'FixedParameter'` property (see xref).

Continuous-Time Models

For models with one output, continuous-time models can be estimated from continuous-time frequency-domain data. In this case, `na` is the number of estimated denominator coefficients and `nb` is number of estimated numerator coefficients.

Note For continuous-time models, omit the delay parameter `nk` because it has no meaning in this case. Because estimating continuous-time ARX models often produces bias, you might get better results by using the `oe` method.

For example, when `na = 4`, `nb = 2`, the model structure is

$$G(s) = \frac{b_1 s + b_2}{s^4 + a_1 s^3 + a_2 s^2 + a_3 s + a_4}$$

Tip When using continuous-time data, limit the fit to a smaller frequency range using the `'Focus'` `idmodel` property:

```
m = arx(datac,[na nb],'focus',[0 wh])
```

Estimating Initial Conditions

For time-domain data, the signals are shifted such that unmeasured signals are never required in the predictors. Therefore, there is no need to estimate initial conditions.

For frequency-domain data, it might be necessary to adjust the data by initial conditions that support circular convolution. See "Specifying the Initial States" on page 5-65.

You can set the property `'InitialState'` to one of the following values:

- `'zero'` — No adjustment.
- `'estimate'` — Perform adjustment to the data by initial conditions that support circular convolution.
- `'auto'` — Automatically choose between `'zero'` and `'estimate'` based on the data.

See `Algorithm Properties` for more information on model properties.

Algorithm

QR factorization solves the overdetermined set of linear equations that constitutes the least-squares estimation problem.

The regression matrix is formed so that only measured quantities are used (no fill-out with zeros). When the regression matrix is larger than `MaxSize`, data is segmented and QR factorization is performed iteratively on these data segments.

Examples

This example generates input data based on a specified ARX model, and then uses this data to estimate an ARX model.

```
A = [1  -1.5  0.7]; B = [0 1 0.5];
m0 = idpoly(A,B);
u = iddata([],idinput(300,'rbs'));
e = iddata([],randn(300,1));
y = sim(m0, [u e]);
z = [y,u];
m = arx(z,[2 2 1]);
```

See Also

`Algorithm Properties`

`EstimationInfo`

`ar`

`idarx`

idpoly

iv4

ivar

ivx

pem

Purpose

ARX parameters with variance information from `idarx` models

Syntax

```
[A,B] = arxdata(m)
[A,B,dA,dB] = arxdata(m)
```

Arguments

`model`

Name of the `idarx` model object, which belongs to the `idmodel` abstract class.

Note Also accepts `idpoly` models with an underlying ARX structure with orders `nc=nd=nf=0`.

Description

`[A,B] = arxdata(m)` returns `A` and `B` as 3–D arrays.

Suppose *ny* is the number of outputs (the dimension of the vector *y(t)*) and *nu* is the number of inputs.

`A` is an `ny-by-ny-by-(na+1)` array such that

```
A(:,:,k+1) = Ak
A(:,:,1) = eye(ny)
```

where `k=0,1,...,na`.

`B` is an `ny-by-nu-by-(nb+1)` array with

```
B(:,:,k+1) = Bk
```

`A(0)` is always the identity matrix. The leading entries in `B` equal to zero, which means there are no delays in the model.

Note For a time series, `B = []`.

`[A,B,dA,dB] = arxdata(m)` returns `A` and `B` matrices, and `dA` and `dB` as the estimated standard deviations of `A` and `B`, respectively.

Remarks

A and B are 2–D or 3–D arrays and are returned in the standard multivariable ARX format (see `idarx`), describing the model.

$$A(q)y(t) = B_1(q)u_1(t-n_{k1}) + \ldots + B_{nu}(q)u_{nu}(t-n_{k_{nu}}) + e(t)$$

$$y(t) + A_1 y(t-1) + A_2 y(t-2) + \ldots + A_{na} y(t-na) =$$

$$B_0 u(t) + B_1 u(t-1) + \ldots + B_{nb} u(t-nb) + e(t)$$

Here A_k and B_k

`A` and `B` matrices, which have dimensions *ny*-by-*ny* and *ny*-by-*nu*, respectively. Here, *ny* is the number of outputs (the dimension of the vector *y(t)*) and *nu* is the number of inputs.

See Also

`idarx`

`idpoly`

Purpose

Compute and compare loss functions for single-output ARX models

Syntax

```
V = arxstruc(ze,zv,NN)
V = arxstruc(ze,zv,NN,maxsize)
```

Arguments

`ze`
: Estimation data set can be `iddata` or `idfrd` object.

`zv`
: Validation data set can be `iddata` or `idfrd` object.

`NN`
: Matrix defines the number of different ARX-model structures . Each row of `NN` is of the form

```
nn = [na nb nk]
```

`maxsize`

Description

Note Use `arxstruc` single-output systems only.

`V = arxstruc(ze,zv,NN)` returns `V`, which contains the loss functions in its first row. The remaining rows of `V` contain the transpose of `NN`, so that the orders and delays are given just below the corresponding loss functions. The last column of `V` contains the number of data points in `ze`.

`V = arxstruc(ze,zv,NN,maxsize)` uses the additional specification of the maximum data size.

See `Algorithm Properties` for an explanation of `maxsize`.

with the same interpretation as described for `arx`. See `struc` for easy generation of typical `NN` matrices for single-input systems.

The output argument `V` is best analyzed using `selstruc`. The selection of a suitable model structure based on the information in `v` is normally done using `selstruc`.

Remarks

Each of ze and zv is an iddata object containing output-input data. Frequency-domain data and idfrd objects are also supported. Models for each of the model structures defined by NN are estimated using the data set ze. The loss functions (normalized sum of squared prediction errors) are then computed for these models when applied to the validation data set zv. The data sets ze and zv need not be of equal size. They could, however, be the same sets, in which case the computation is faster.

Examples

Compare first- and fifth-order models with one delay using cross-validation on the second half of the data set. Then, select the order that gives the best fit to the validation data set.

```
NN = struc(1:5,1:5,1);
V = arxstruc(z(1:200),z(201:400),NN);
nn = selstruc(V,0);
m = arx(z,nn);
```

See Also

Algorithm Properties

arx

idpoly

ivstruc

selstruc

struc

Purpose

Reduce model order (requires Control System Toolbox)

Syntax

```
MRED = balred(M)
MRED = balred(M,ORDER,'DisturbanceModel','None')
```

Description

This function reduces the order of any model `M` given as an `idmodel` object. The resulting reduced-order model, `MRED`, is an `idss` model.

The function requires several routines in Control System Toolbox.

`ORDER`: The desired order (dimension of the state-space representation). If `ORDER = []`, which is the default, a plot shows how the diagonal elements of the observability and controllability Gramians of a balanced realization decay with the order of the representation. You are then prompted to select an order based on this plot. The idea is that such a small element has a negligible influence on the input-output behavior of the model. We recommend that you choose an order such that only large elements in these matrices are retained.

`'DisturbanceModel'`: If the property `DisturbanceModel` is set to `'None'`, then an output-error model `MRED` is produced: that is, one with the Kalman gain K equal to zero. Otherwise (default), the disturbance model is also reduced.

The function recognizes whether `M` is a continuous- or discrete-time model and performs the reduction accordingly. The resulting model, `MRED`, is similar to `M` in this respect.

There are several options for how the reduction is performed: `AbsTol, RelTol, Offset, Elimination`.

Algorithm

The function `balred` from Control System Toolbox is used. The plot, in case `ORDER = []`, shows the vector `g` returned by `balreal`.

Examples

Build a high-order multivariable ARX model, reduce its order to 3, and compare the frequency responses of the original and reduced models:

```
M = arx(data,[4*ones(3,3),4*ones(3,2),ones(3,2)]);
MRED = balred(M,3);
```

```
bode(M,MRED)
```

Use the reduced-order model as an initial condition for a third-order state-space model.

M2 = pem(data,MRED);

See Also `balreal`

Purpose

Estimate parameters of Box-Jenkins model returning `idpoly` object

Syntax

```
m = bj(data,orders)
m = bj(data,'nb',nb,'nc',nc,'nd',nd,'nf',nf,'nk',nk)
m = bj(data,orders,'Property1',Value1,'Property2',Value2,...)
```

Description

`bj` returns `m` as an `idpoly` object with the resulting parameter estimates, together with estimated covariances. The `bj` function estimates parameters of the Box-Jenkins model structure

$$y(t) = \frac{B(q)}{F(q)}u(t-nk) + \frac{C(q)}{D(q)}e(t)$$

using a prediction error method.

`data` is an `iddata` object containing the output-input data. Frequency-domain signals are not supported by `bj`. Use `oe` instead.

The model orders can be specified by setting the argument `orders` to

```
orders = [ nb nc nd nf nk]
```

The parameters `nb`, `nc`, `nd`, and `nf` are the orders of the Box-Jenkins model and `nk` is the delay. Specifically,

$$nf: \qquad F(q) = 1 + f_1 q^{-1} + \ldots + f_{nf} q^{-nf}$$

$$nb: \qquad B(q) = b_1 + b_2 q^{-1} + \ldots + b_{nb} q^{-nb+1}$$

$$nc: \qquad C(q) = 1 + c_1 q^{-1} + \ldots + c_{nc} q^{-nc}$$

$$nd: \qquad D(q) = 1 + d_1 q^{-1} + \ldots + d_{nd} q^{-nd}$$

The orders can also be defined as property name/property value pairs `(...,'nb',nb,...)`. Alternatively, you can specify the vector as

```
orders = mi
```

where mi is an initial guess at the Box-Jenkins model given in idpoly format.

For multiinput systems, nb, nf, and nk are row vectors with as many entries as there are input channels. Entry number i then describes the orders and delays associated with the ith input.

The structure and the estimation algorithm are affected by any property name/property value pairs that are set in the input argument list. Useful properties are 'Focus', 'InitialState', 'Trace', 'MaxIter', 'Tolerance', 'LimitError', and 'FixedParameter'.

See Algorithm Properties and the reference pages for idmodel and idpoly for details of these properties and their possible values.

bj does not support multioutput models. Use a state-space model for this case (see n4sid and pem).

Examples

Here is an example that generates data and stores the results of the startup procedure separately.

```
B = [0 1 0.5];
C = [1 -1 0.2];
D = [1 1.5 0.7];
F = [1 -1.5 0.7];
m0 = idpoly(1,B,C,D,F,0.1);
e = iddata([],randn(200,1));
u = iddata([],idinput(200));
y = sim(m0,[u e]);
z = [y u];
mi = bj(z,[2 2 2 2 1],'MaxIter',0)
m = bj(z,mi,'Maxi',10)
m.EstimationInfo
m = bj(z,m); % Continue if m.es.WhyStop shows that
             % maxiter is reached.
compare(z,m,mi)
```

Algorithm

`bj` uses essentially the same algorithm as `armax` with modifications to the computation of prediction errors and gradients.

See Also

`Algorithm Properties`

`EstimationInfo`

`idpoly`

`pem`

Purpose

Plot Bode diagram of frequency response with confidence interval

Syntax

```
bode(m)
[mag,phase,w] = bode(m)
[mag,phase,w,sdmag,sdphase] = bode(m)
bode(m1,m2,m3,...,w)
bode(m1,'PlotStyle1',m2,'PlotStyle2',...)
bode(m1,m2,m3,..'sd',sd,'mode',mode,'ap',ap)

bode(m1,m2,m3,'sd',sd,'mode',mode,'ap',ap,'fill')
```

Description

`bode` computes the magnitude and phase of the frequency response of `idmodel` and `idfrd` models. When invoked without left-hand arguments, `bode` produces a Bode plot on the screen.

`bode(m)` plots the Bode response of an arbitrary `idmodel` or `idfrd` model `m`. This model can be continuous or discrete, and SISO or MIMO. The `InputNames` and `OuputNames` properties of the models are used to plot the responses for different I/O channels in separate plots. Pressing the **Enter** key advances the plot from one input-output pair to the next one. Typing **Ctrl+C** aborts the plotting in an orderly fashion

If `m` contains information about both I/O channels and output noise spectra, only the I/O channels are shown. To show the output noise spectra, enter `m('n')` (`'n'` for `'noise'`) in the model list. Analogously, you can select specific I/O channels with normal subreferencing `m(ky,ku)`.

Argument w

`bode(m,w)` explicitly specifies the frequency range or frequency points to be used for the plot or for computing the response.

To focus on a particular frequency interval `[wmin,wmax]`, set `w = {wmin,wmax}` (notice the curly brackets). This plots the response for 100 frequency points logarithmically spaced from `wmin` to `wmax`. You can change this to `NP` points by using `w = {wmin,wmax,NP}`.

To use particular frequency points, set `w` to the vector of desired frequencies. Use `logspace` to generate logarithmically spaced frequency vectors. All frequencies should be specified in rad/s.

Note that the frequencies cannot be specified for `idfrd` objects. For those the plot and response are calculated for the internally stored frequencies. However, the plot is restricted to the range `{wmin,wmax}` if this is specified.

If no frequency range is specified, a default choice is made based on the dynamics of the model.

Property Name/Property Value Pairs 'sd'/sd, 'ap'/ap, and 'mode'/mode

The pairs can appear in any order or be omitted.

- `sd`: If `sd` is specified as a number larger than zero, confidence intervals for the functions are added to the graph as dash-dotted curves (of the same color as the estimate curve). They indicate the confidence regions corresponding to `sd` standard deviations. If an argument `'fill'` is included in the argument list, the confidence region is marked as a filled band instead.
- `ap`: By default, amplitude and phase plots are shown simultaneously for each I/O channel present in `m`. For spectra, phase plots are omitted. To show amplitude plots only, use `ap = 'A'`. For phase plots only, use `ap = 'P'`. The default is `ap = 'B'` for both plots.
- `mode`: To obtain all input/output plots in the same diagram use `mode = 'same'`.

Several Models

`bode(m1,m2,...,mN)` or `bode(m1,m2,...mN,w)` plots the Bode response of several `idmodel` or `idfrd` models on a single figure. The models can be mixes of different sizes and continuous/discrete. The sorting of the plots is based on the `InputNames` and `OutputNames`. If the frequencies `w` are specified, these will apply to all non-`idfrd` models in the list. If you want different frequencies for different models, you should thus first convert them to `idfrd` objects using the `idfrd` command.

`bode(m1,'PlotStyle1',...,mN,'PlotStyleN')` further specifies which color, line style, and/or marker should be used to plot each system, as in

```
bode(m1,'r--',m2,'gx')
```

Arguments

The output argument `w` contains the frequencies for which the response is given, whether specified among the input arguments or not. The output arguments `mag` and `phase` are 3-D arrays with dimensions

```
(number of outputs)x(number of inputs)x(length of w)
```

For SISO systems, `mag(1,1,k)` and `phase(1,1,k)` give the magnitude and phase (in degrees) at the frequency ω_k`= w(k)`. To obtain the result as a normal vector of responses, use `mag = mag(:)` and `phase = phase(:)`.

For MIMO systems, `mag(i,j,k)` is the magnitude of the frequency response at frequency `w(k)` from input `j` to output `i`, and similarly for `phase(i,j,k)`.

If `sdmag` and `sdphase` are specified, the standard deviations of the magnitude and phase are also computed. Then `sdmag` is an array of the same size as `mag`, containing the estimated standard deviations of the response, and analogously for `sdphase`.

See Also

`etfe`

`ffplot`

`freqresp`

`idfrd`

`nyquist`

`spa`

`spafdr`

Purpose

Compare model output and measured output

Syntax

```
compare(data,m);
compare(data,m,k)
compare(data,m,k,'Samples',sampnr,'InitialState',init,'OutputPlots
',Yplots)
compare(data,m1,m2,...,mN)
compare(data,m1,'PlotStyle1',...,mN,'PlotStyleN')
[yh,fit,x0] = compare(data,m1,'PlotStyle1',...,mN,'PlotStyleN',k)
```

Description

`data` is the output-input data in the usual `iddata` object format. `data` can also be an `idfrd` object with frequency-response data.

`compare` computes the output `yh` that results when the model `m` is simulated with the input `u`. `m` can be any `idmodel` or `idnlmodel` model object. The result is plotted together with the corresponding measured output `y`. The percentage of the output variation that is explained by the model

```
  fit = 100*(1 - norm(yh - y)/norm(y-mean(y)))
```

is also computed and displayed. For multioutput systems, this is done separately for each output. For frequency-domain data (or in general for complex valued data) the `fit` is still calculated as above, but only the absolute values of `y` and `yh` are plotted.

When the argument `k` is specified, the *k* step-ahead prediction of `y` according to the model `m` are computed instead of the simulated output. In the calculation of $yh(t)$, the model can use outputs up to time $t-k$: $y(s), s = t-k$, $t-k-1$, ... (and inputs up to the current time *t*). The default value of `k` is `inf`, which gives a pure simulation from the input only. Note that for frequency-domain data, only simulation (`k = inf`) is allowed, and for time-series data (no input) only prediction (`k` not `inf`) is possible.

Property Name/Property Value Pairs

The optional property name/property value pairs `'Samples'/sampnr`, `'InitialState'/init`, and `'OutputPlots'/Yplots` can be given in any order.

The argument `Yplots` can be a cell array of strings. Only the outputs with `OutputName` in this array are plotted, while all are used for the necessary computations. If `Yplots` is not specified, all outputs are plotted.

The argument `sampnr` indicates that only the sample numbers in this row vector are plotted and used for the calculation of the fit. The whole data record is used for the simulation/prediction.

The argument `init` determines how to handle initial conditions in the models:

- `init = 'e'` (for `'estimate'`) estimates the initial conditions for best fit.
- `init = 'm'` (for `'model'`) used the model's internally stored initial state.
- `init = 'z'` (for `'zero'`) uses zero initial conditions.
- `init = x0`, where `x0` is a column vector of the same size as the state vector of the models, uses `x0` as the initial state.
- `init = 'e'` is the default.

Several Models

When several models are specified, as in `compare(data,m1,m2,...,mN)`, the plots show responses and fits for all models. In that case `data` should contain all inputs and outputs that are required for the different models. However, some models might correspond to subselections of channels and might not need all channels in `data`. In that case the proper handling of signals is based on the `InputNames` and `OutputNames` of `data` and the models.

With `compare(data,m1,'PlotStyle1',...mN,'PlotStyle2')`, the color, line style, and/or marker can be specified for the curves associated with the different models. The markers are the same as for the regular `plot` command. For example,

```
compare(data,m1,'g_*',m2,'r:')
```

If `data` contains several experiments, separate plots are given for the different experiments. In this case `sampnr`, if specified, must be a cell array with as many entries as there are experiments.

Arguments

When output arguments `[yh,fit,x0] = compare(data,m1,..,mN)` are specified, no plots are produced.

`yh` is a cell array of length equal to the number of models. Each cell contains the corresponding model output as an `iddata` object.

`fit` is, in the general case, a 3-D array with `fit(kexp,kmod,ky)` containing the fit (computed as above) for output `ky`, model `kmod`, and experiment `kexp`.

`x0` is a cell array, such that `x0{kmod}` is the estimated initial state for model number `kmod`. If data is multiexperiment, `X0{kmod}` is a matrix whose column number `kexp` is the initial state vector for experiment number `kexp`.

Examples

Split the data record into two parts. Use the first one for estimating a model and the second one to check the model's ability to predict six steps ahead.

```
ze = z(1:250);
zv = z(251:500);
m= armax(ze,[2 3 1 0]);
compare(zv,m,6);
compare(zv,m,6,'Init','z') % No estimation of
                           % the initial state.
```

See Also

`pe`

`predict`

`sim`

Purpose

Estimate time-series covariance functions

Syntax

```
R = covf(data,M)
R = covf(data,M,maxsize)
```

Description

`data` is an `iddata` object and `M` is the maximum delay -1 for which the covariance function is estimated. The routine is intended for time-domain data only.

Let `z` contain the output and input channels

$$z(t) = \begin{bmatrix} y(t) \\ u(t) \end{bmatrix}$$

where y and u are the rows of `data.OutputData` and `data.InputData`, respectively, with a total of `nz` channels.

`R` is returned as an nz^2 -by- M matrix with entries

$$R(i+(j-1)nz,k+1) = \frac{1}{N}\sum_{t=1}^{N} z_i(t)z_j(t+k) = \hat{R}_{ij}(k)$$

where z_j is the `j`th row of z, and missing values in the sum are replaced by zero.

The optional argument `maxsize` controls the memory size as explained under `Algorithm Properties`.

The easiest way to describe and unpack the result is to use

```
reshape(R(:,k+1),nz,nz) = E z(t)*z'(t+k)
```

Here `'` is complex conjugate transpose, which also explains how complex data is handled. The expectation symbol `E` corresponds to the sample means.

Algorithm

When `nz` is at most two, and when permitted by `maxsize`, a fast Fourier transform technique is applied. Otherwise, straightforward summing is used.

See Also

`iddata`

`spa`

Purpose

Estimate impulse response using prewhitened-based correlation analysis

Syntax

```
cra(data);
[ir,R,cl] = cra(data,M,na,plot);
cra(R);
```

Description

`data` is the output-input data given as an `iddata` object. The routine is intended for time-domain data only.

The routine only handles single-input-single-output data pairs. (For the multivariate case, apply `cra` to two signals at a time, or use `impulse`.) `cra` prewhitens the input sequence; that is, `cra` filters `u` through a filter chosen so that the result is as uncorrelated (white) as possible. The output `y` is subjected to the same filter, and then the covariance functions of the filtered `y` and `u` are computed and graphed. The cross correlation function between (prewhitened) input and output is also computed and graphed. Positive values of the lag variable then correspond to an influence from `u` to later values of `y`. In other words, significant correlation for negative lags is an indication of feedback from `y` to `u` in the data.

A properly scaled version of this correlation function is also an estimate of the system's impulse response `ir`. This is also graphed along with 99% confidence levels. The output argument `ir` is this impulse response estimate, so that its first entry corresponds to lag zero. (Negative lags are excluded in `ir`.) In the plot, the impulse response is scaled so that it corresponds to an impulse of height $1/T$ and duration T, where T is the sampling interval of the data.

The output argument `R` contains the covariance/correlation information as follows:

- The first column of `R` contains the lag indices.
- The second column contains the covariance function of the (possibly filtered) output.

- The third column contains the covariance function of the (possibly prewhitened) input.
- The fourth column contains the correlation function. The plots can be redisplayed by `cra(R)`.

The output argument `cl` is the 99% confidence level for the impulse response estimate.

The optional argument `M` defines the number of lags for which the covariance/correlation functions are computed. These are from `-M` to `M`, so that the length of `R` is `2M+1`. The impulse response is computed from `0` to `M`. The default value of `M` is `20`.

For the prewhitening, the input is fitted to an AR model of order `na`. The third argument of `cra` can change this order from its default value `na = 10`. With `na = 0` the covariance and correlation functions of the original data sequences are obtained.

`plot:` `plot = 0` gives no plots. `plot = 1` (the default) gives a plot of the estimated impulse response together with a 99% confidence region. `plot = 2` gives a plot of all the covariance functions.

An often better alternative to `cra` is the functions `impulse` and `step`, which use a high-order FIR model to estimate the impulse response.

Examples

Compare a second-order ARX model's impulse response with the one obtained by correlation analysis.

```
ir = cra(z);
m = arx(z,[2 2 1]);
imp = [1;zeros(19,1)];
irth = sim(m,imp);
subplot(211)
plot([ir irth])
title('impulse responses')
subplot(212)
plot([cumsum(ir),cumsum(irth)])
title('step responses')
```

See Also

`impulse`

`step`

Purpose

Store nonlinearity estimator with user-defined unit function for nonlinear ARX and Hammerstein-Wiener models

Syntax

```
C=customnet(H)
C=customnet(H,Property1,Value1,...PropertyN,ValueN)
```

Arguments

`H`

User-defined function handle of the unit function of the custom net.

`H` must point to a function of the form `[f,g,a] = function_name(x)`, where `f` is the value of the function, `g=df/dx` and indicates the unit function active range. `g` is significantly nonzero in the interval `[-a a]`.

Description

`customnet` is an object that stores a custom nonlinear estimator with a user-defined unit function.

You can use the constructor to create the nonlinearity object, as follows:

`C=customnet(H)` creates a nonlinearity estimator object with a user-defined unit function using the function handle `H`.

`C=customnet(H,Property1,Value1,...PropertyN,ValueN)` creates a nonlinearity estimator using property-value pairs defined in "customnet Properties" on page 12-57.

Remarks

Use `customnet` to define a nonlinear function $y = F(x)$, where y is scalar and x is an `m`-dimensional row vector. The unit function is based on the following function expansion with a possible linear term L:

$$F(x) = (x-r)PL + a_1 f\left((x-r)Qb_1 + c_1\right) + \dots + a_n f\left((x-r)Qb_n + c_n\right) + d$$

where f is a unit function that you define using the function handle H.

P and Q are `m`-*by*-`p` and `m`-*by*-`q` projection matrices, respectively. The projection matrices P and Q are determined by principal component analysis of estimation data. Usually, `p=m`. If the components of x in the estimation data are linearly dependent, then `p<m`. The number of columns of Q, `q`, corresponds to the number of components of `x` used in the unit function.

When used to estimate nonlinear ARX models, `q` is equal to the size of the `NonlinearRegressors` property of the `idnlarx` object. When used to estimate Hammerstein-Wiener models, `m=q=1` and Q is a scalar.

r is a 1-*by*-`m` vector and represents the mean value of the regressor vector computed from estimation data.

d, a, and c are scalars.

L is a `p`-*by*-1 vector.

b are `q`-*by*-1 vectors.

The function handle of the unit function of the custom net must have the form `[f,g,a] = function_name(x)`. This function must be vectorized, which means that for a vector or matrix `x`, the output arguments `f` and `g` must have the same size as `x` and be computed element-by-element.

customnet Properties

You can include property-value pairs in the constructor to specify the object.

After creating the object, you can use `get` or dot notation to access the object property values. For example:

```
% List all property values
get(C)
% Get value of NumberOfUnits property
C.NumberOfUnits
```

You can use dot notation to assign property values to the object. `set` is not supported for MCOS objects.

For example, the following two commands are equivalent:

```
C.NumberOfUnits=5
```

The Parameters property is a structure. Typically, the values of this structure are set by estimating a model with a customnet nonlinearity. If you need to set the values of this structure to values given by variables r, P, Q, L, a_k, b_k, c_k and d, you can use the following syntax:

```
X=struct('RegressorMean',r,
         'NonLinearSubspace',P,
         'LinearSubspace',Q,
         'LinearCoef',L,
         'Dilation',b_k,
         'Translation',c_k,
         'OutputCoef',a_k,
         'OutputOffset',d);
C.Parameters=X;
```

Property Name	Description
NumberOfUnits	Integer specifies the number of nonlinearity units in the expansion. Default=10. For example: `customnet(H,'NumberOfUnits',5)`
LinearTerm	Can have the following values: • 'on'—Estimates the vector L in the expansion. • 'off'—Fixes the vector L to zero. For example: `customnet(H,'LinearTerm','on')`

Property Name	Description
`Parameters`	A structure containing the parameters in the nonlinear expansion, as follows: • `RegressorMean`: 1-*by*-`m` vector containing the means of `x` in estimation data, `r`. • `NonLinearSubspace`: `m`-*by*-`q` matrix containing *Q*. • `LinearSubspace`: `m`-*by*-`p` matrix containing *P*. • `LinearCoef`: `p`-*by*-1 vector *L*. • `Dilation`: `q`-*by*-1 matrix containing the values `b_k`. • `Translation`: 1-*by*-`n` vector containing the values `c_k`. • `OutputCoef`: `n`-*by*-1 vector containing the values `a_k`. • `OutputOffset`: scalar `d`.
`UnitFcn`	Stores the function handle that points to the unit function.

Examples

The following code, contained in `gaussunit.m`, defines a sample unit function.

```
[f, g, a] = GAUSSUNIT(x)
% x: unit function variable
% f: unit function value
% g: df/dx
% a: unit active range (g(x) is significantly
% nonzero in the interval [-a a])

% The unit function must be "vectorized": for
% a vector or matrix x, the output arguments f and g
% must have the same size as x,
% computed element-by-element.

% GAUSSUNIT customnet unit function example
[f, g, a] = gaussunit(x)
```

```
  f =  exp(-x.*x);
  if nargout>1
    g = - 2*x.*f;
    a = 0.2;
  end
```

You typically use custom networks in `nlarx` and `nlhw` model estimation commands. For example:

```
  % Define handle to example unit function.
  H = @gaussunit;
  % Estimate nonlinear ARX model using
  % Gauss unit function with 5 units.
  m = nlarx(Data,Orders,customnet(H,'NumberOfUnits',5));
```

See Also

`evaluate`

`nlarx`

`nlhw`

Purpose

Store custom regressor for nonlinear ARX models

Syntax

```
C=customreg(Function,Arguments)
C=customreg(Function,Arguments,Delays,Vectorized)
```

Arguments

`Function`
: Function handle or string representing a function of input and output variables.

`Arguments`
: Cell array of strings that represent the names of model inputs and outputs used in the function `Function`. Each input and output name must coincide with the strings in the `InputName` and `OutputName` properties of the corresponding `idnlarx` object. The size of `Arguments` must match the number of `Function` inputs.

`Delays`
: Vector of positive integers representing the delays of `Arguments` variables. The size of `Delays` must match the size of `Arguments`. Default: `1` for each vector element.

`Vectorized`
: Flag that indicates whether `Function` is in a format that supports vectorized computations when applied to vector arguments. Can have values `1` (true) and `0` (false). Default: `0`.

Description

`C=customreg(Function,Arguments)` creates an object that stores custom regressors for nonlinear ARX models. Custom regressor is defined using the function of input and output variables, `Function`.

`C=customreg(Function,Arguments,Delays,Vectorized)` create a custom regressor that includes the delays corresponding to inputs or outputs in `Arguments`.

For multioutput models with `p` outputs, the custom regressor is `p`-by-`1` cell array or an array of `customreg` object, where the `ky`th entry defines the custom regressor for output `ky`.

Remarks

Use the `customreg` object to define a custom regressor entering the nonlinearity estimator of the nonlinear ARX models (`idnlarx` object). The custom regressor function is usually a nonlinear function of inputs and outputs.

To list custom regressors of a multioutput model, type the following command:

```
model.custom
```

To retrieve `r`th custom regressor for output `ky`, type the following command:

```
model.custom{ky}(r)
```

For more information about regressors, type `help regressors`.

For more information on creating custom regressors that are polynomial combinations of delayed inputs and outputs, type `help polyreg`.

Use the `Vectorized` property to specify whether to compute custom regressors using vectorized form. If you know that your regressor formulas can be vectorized, set `Vectorized` to `1` after creating the `customreg` object to achieve better performance. To better understand vectorization, consider that custom regressors are defined by a formula, such as `z=@(x,y)x^2*y`. `x` and `y` are vectors and each variable is evaluated over a time grid. Therefore, `z` must be evaluated for each `(xi,yi)` pair, and the results are concatenated to produce a `z` vector, as follows:

```
for k = 1:length(x)
   z(k) = x(k)^2*y(k)
end
```

The above expression is a nonvectorized computation and tends to be slow. Specifying a `Vectorized` computation uses MATLAB vectorization rules to evaluate the regressor expression using matrices instead of the `FOR`-loop and results in faster computation, as follows:

```
% ".*" indicates element-wise operation
```

```
z=(x.^2).*y
```

customreg Properties

You can include property-value pairs in the constructor to specify the object.

After creating the object, you can use `get` or dot notation to access the object property values. For example:

```
% List all property values
get(C)
% Get value of Arguments property
C.Arguments
```

You can use dot notation to assign property values to the object. `set` is not supported for MCOS objects.

For example, the following two commands are equivalent:

```
C.Delays=[2 3]
```

Property Name	Description
`Function`	Function handle or string representing a function of standards regressors. For example: `cr = @(x,y) x*y`
`Arguments`	Cell array of strings that represent the names of model inputs and outputs used in the function `Function`. Each input and output name must coincide with the strings in the `InputName` and `OutputName` properties of the `idnlarx` object with the custom regressor. The size of `Arguments` must match the number of `Function` inputs. For example, `Arguments` correspond to `{'y1','u1'}` in the following syntax: `C = customreg(cr,{'y1','u1'},[2 3])`

Property Name	Description
`Delays`	Vector of positive integers representing the delays of `Arguments` variables. The size of `Delays` must match the size of `Arguments`. Default: `1` for each vector element. For example, `Delays` correspond to `[2 3]` in the following syntax: `C = customreg(cr,{'y1','u1'},[2 3])`
`Vectorized`	Can have the following values: • `1`—`Function` is computed in vectorized form when called with vector arguments. • `0`—`Function` is not computed in vectorized form when called with vector arguments.

Examples

Consider a system that has input `u` and output `y`. Suppose that the following transformations are useful for predicting future outputs, based on physical insight into the relationship between measured variables:

- `u(t-1)sin(y(t-3))`
- `u(t-2)^3`

You can define custom regressors in any of the two ways:

- Cell array of strings. For example:

```
C={'u(t-1)*sin(y(t-3)','u(t-2)^3'}
```

- Object array of `customreg` objects. For example:

```
cr1=@(x,y) x*sin(y)
cr2=@(x) x^3
C=[customreg(cr1,{'u' 'y'},[1 3]),...
   customreg(cr2,{'u'},2)]
```

After you define the custom regressors, estimate the nonlinear ARX model using the following syntax:

```
m = nlarx(Data,Orders,linear,'CustomRegressors',C)
```

In this case, `u` and `y` are `Data` channel names, and the nonlinearity estimator `linear` specifies that the prediction is a linear function of custom regressors, where the custom regressor might be nonlinear. You can introduce additional nonlinearities using nonlinearity estimators. For a complete list of available nonlinearities, type `idprops idnlestimators`.

To create two custom regressors as an object array, use the following commands:

```
cr1=@(x,y) x*sin(y);
cr2=@(x) x^3;
C=[customreg(cr1,{'u1' 'y1'},[1 3]),...
   customreg(cr2,{'u1'},2)]
% Use this customreg object array for estimating
% nonlinear ARX model
m=nlarx(Data,Orders,'wavenet','CustomRegressor',C);
```

The following command is equivalent to the previous code snippet and incorporates custom regressor definitions directly in the estimator command:

```
m = nlarx(Data,Orders,'wavenet',...
           'CustomRegressors',...
          {'u1(t-1)*sin(y1(t-3))','u1(t-2)^3'});
```

See Also

`customreg`

`nlarx`

Purpose

Convert model from continuous to discrete time

Syntax

```
md = c2d(mc,T)
md = c2d(mc,T,method)
[md,G] = c2d(mc,T,method)
```

Description

`mc` is a continuous-time model such as any `idmodel` object (`idgrey`, `idproc`, `idpoly`, or `idss`). `md` is the model that is obtained when it is sampled with sampling interval `T`.

`method = 'zoh'` (default) makes the translation to discrete time under the assumption that the input is piecewise constant (zero-order hold).

`method = 'foh'` assumes the input to be piecewise linear between the sampling instants (first-order hold).

With Control System Toolbox, methods `'tustin'`, `'prewarp'`, and `'matched'` are also supported. In these cases the covariance matrix is not transformed.

Note that the innovations variance λ of the continuous-time model is interpreted as the intensity of the spectral density of the noise spectrum. The noise variance in `md` is thus given as λ`/T`.

`idpoly` and `idss` models are returned in the same format. `idgrey` structures are preserved if their `CDMfile` property is equal to `'cd'`. Otherwise they are transformed to `idss` objects. `idproc` models are returned as `idgrey` objects.

For `idpoly` models, the covariance matrix is translated by the use of numerical derivatives. The step sizes used for the differentiation are given by the function `nuderst`. For `idss, idproc,` and `idgrey` models, the covariance matrix is not translated, but covariance information about the input-output properties is included in `md`. To inhibit the translation of covariance information (which may take some time), use `c2d(mc,T,'covariance','none')`.

The output argument `G` is a matrix that transforms the initial state `x0` of `mc` to the initial state of `md` as

```
XOd=G * [X0; u(0)],
```

where `u(0)` is the input at time 0. For `idproc` models, the state variables correspond to those of `idgrey(mc)`. For `idpoly` models, `G` is returned as the empty matrix.

Examples

Define a continuous-time system and study the poles and zeros of the sampled counterpart.

```
mc = idpoly(1,1,1,1,[1 1 0],'Ts',0);
md = c2d(mc,0.5);
pzmap(md)
```

See Also

`d2c`

Purpose

Store dead-zone nonlinearity estimator for Hammerstein-Wiener models

Syntax

```
s=deadzone(ZeroInterval,I)
```

Description

`deadzone` is an object that stores the dead-zone nonlinearity estimator for estimating Hammerstein-Wiener models.

You can use the constructor to create the nonlinearity object, as follows:

`s=deadzone(ZeroInterval,I)` creates a dead-zone nonlinearity estimator object, initialized with the zero interval `I`.

Use `evaluate(d,x)` to compute the value of the function defined by the `deadzone` object `d` at `x`.

Remarks

Use `deadzone` to define a nonlinear function $y = F(x)$, where F is a function of x and has the following characteristics:

$a \le x < b$	$F(x) = 0$
$x < a$	$F(x) = x - a$
$x \ge b$	$F(x) = x - b$

y and x are scalars.

saturation Properties

You can specify the property value as an argument in the constructor to specify the object.

After creating the object, you can use `get` or dot notation to access the object property values. For example:

```
% List ZeroInterval property value
get(d)
d.ZeroInterval
```

You can use dot notation to assign property values to the object. `set` is not supported for MCOS objects.

For example, the following two commands are equivalent:

```
d.ZeroInterval=[-1 1]
```

Property Name	Description
`ZeroInterval`	1-*by*-2 row vector that specifies the initial zero interval of the nonlinearity. Default=`[NaN NaN]`. For example: `deadzone('ZeroInterval',[-1.5 1.5])`

Examples

Use `deadzone` to specify the dead-zone nonlinearity estimator in Hammerstein-Wiener models. For example:

```
m=nlhw(Data,Orders,deadzone([-1 1]),[]);
```

The dead-zone nonlinearity is initialized at the interval `[-1 1]`. The interval values are adjusted to the estimation data by `nlhw`.

See Also

`nlhw`

delayest

Purpose

Estimate time delay (dead time) from data

Syntax

```
nk = delayest(Data)
nk = delayest(Data,na,nb,nkmin,nkmax,maxtest)
```

Description

`Data` is an `iddata` object containing the input-output data. It can also be an `idfrd` object defining frequency-response data. Only single-output data can be handled.

`nk` is returned as an integer or a row vector of integers, containing the estimated time delay in samples from the input(s) to the output in `Data`.

The estimate is based on a comparison of ARX models with different delays:

$$y(t) + a_1 y(t-1) + \ldots + a_{na} y(t-na) = \ldots$$
$$b_1 u(t-nk) + \ldots + b_{nb} u(t-nb-nk+1)$$

The integer `na` is the order of the A polynomial (default 2). `nb` is a row vector of length equal to the number of inputs, containing the order(s) of the B polynomial(s) (default all 2).

`nkmin` and `nkmax` are row vectors of the same length as the number of inputs, containing the smallest and largest delays to be tested. Defaults are `nkmin = 0` and `nkmax = nkmin+20`.

If `nb`, `nkmax`, and/or `nkmin` are entered as scalars in the multiinput case, all inputs will be assigned the same values.

`maxtest` is the largest number of tests allowed (default 10,000).

Purpose

Remove trends from output-input data

Syntax

```
zd = detrend(z)
zd = detrend(z,o,brkp)
```

Description

`z` is an `iddata` object containing the input-output data. `detrend` removes the trend from each signal and returns the result as an `iddata` object `zd`.

The default (`o = 0`) removes the zeroth order trends; that is, the sample means are subtracted. If `z` is a frequency-domain data object, the response at frequency `0` is then set to zero,

With `o = 1`, linear trends are removed after a least squares fit. With `brkp` not specified, one single line is subtracted from the entire data record. A continuous piecewise linear trend is subtracted if `brkp` contains breakpoints at sample numbers given in a row vector.

Note that `detrend` for `iddata` objects differs somewhat from `detrend` in the Signal Processing Toolbox.

Examples

Remove a V-shaped trend from the output with its peak at sample number 119, and remove the sample mean from the input.

```
zd1(:,1,[]) = detrend(z(:,1,[]),1,119);
zd2(:,[],1) = detrend(z(:,[],1));
zd = [zd1,zd2];
```

Purpose

Difference signals in `iddata` objects

Syntax

```
zdi = diff(z)
zdi = diff(z,n)
```

Description

`z` is a time-domain `iddata` object. `diff(z)` and `diff(z,n)` apply this command to each of the input/output signals in `z`.

Purpose

Convert model from discrete to continuous time

Syntax

```
mc = d2c(md)
mc = d2c(md,method)
mc = d2c(md,'CovarianceMatrix',cov,'InputDelay',inpd)
```

Description

The discrete-time model `md`, given as any `idmodel` object, is converted to a continuous-time counterpart `mc`. The covariance matrix of the parameters in the model is also translated using the Gauss approximation formula and numerical derivatives of the transformation. The step sizes in the numerical derivatives are determined by the function `nuderst`. To inhibit the translation of the covariance matrix and save time, enter among the input arguments `(...,'CovarianceMatrix,'None,....)`) (any abbreviations will do).

`method` is one of the input intersample behaviors `'zoh'` (zero-order hold) or `'foh'` (first-order hold). If `method` is not specified, the `InterSample` behavior of the data from which `md` was estimated is used.

With Control System Toolbox, methods `'tustin'`, `'prewarp'`, and `'matched'` are also supported. In these cases no translation of the covariance matrix takes place.

If the discrete-time model contains pure time delays, that is, $nk > 1$, then these are first removed before the transformation is made. These delays are appended as pure time delay (dead time) to the continuous-time model as the property `InputDelay`. To have the time delay approximated by a finite-dimensional continuous system, enter among the input arguments `(...,'InputDelay',0,...)`.

If the noise variance is λ in `md`, and its sampling interval is T, then the continuous-time model has an indicated level of noise spectral density equal to $T\,\lambda$.

While `idpoly` and `idss` models are returned in the same format, `idarx` models are returned as `idss` models `mc`. The reason is that the transformation does not preserve the special structure of `idarx`. The `idgrey` structures are preserved if their `CDMfile` property is equal to `cd`. Otherwise they are transformed to `idss` objects.

Note The transformation from discrete to continuous time is not unique. `d2c` selects the continuous-time counterpart with the slowest time constants consistent with the discrete-time model. The lack of uniqueness also means that the transformation can be ill-conditioned or even singular. In particular, poles on the negative real axis, in the origin, or in the point 1, are likely to cause problems. Interpret the results with care.

Examples

Transform an identified model to continuous time and compare the frequency responses of the two models.

```
m = n4sid(data,3)
mc = d2c(m);
bode(m.mc,'sd',3)
```

Note that you can include the transformation to continuous time in the `n4sid` command by specifying the model to be continuous time.

```
mc = n4sid(data,3,'Ts',0)
```

See Also

`c2d`

`nuderst`

Purpose

Information about estimation process results

Syntax

```
m.EstimationInfo
m.es
m.es.DataLength, etc
```

Description

Any estimated model has the property `EstimationInfo`, which is a structure whose fields give information about the results of the estimation. Depending on whether it is an estimated parametric `idmodel` or an estimated frequency response `idfrd`, `EstimationInfo` will contain different fields.

idmodel Case

The model structure will contain the properties `ParameterVector`, `CovarianceMatrix`, and `NoiseVariance`, which are all calculated in the estimation process (see the reference page for `idmodel`). In addition, `EstimationInfo` contains the following fields:

- `Status`: Information whether the model has been estimated, or modified after being estimated.
- `Method`: Name of the estimation command that produced the model.
- `LossFcn`: Value of the identification criterion at the estimate. Normally equal to the determinant of the covariance matrix of the prediction errors, that is, the determinant of `NoiseVariance`. Note that the loss function for the minimization might be different due to `LimitError`. The value of the nonrobustified loss function is always stored in `LossFcn`.
- `FPE`: Akaike's Final Prediction Error, defined as `LossFcn *(1+d/N}/(1-d/N)`, where `d` is the number of estimated parameters and `N` is the length of the data record.
- `DataName`: Name of the data set from which the model was estimated. This is equal to the property `name` of the `iddata` object. If this was not defined, the name of the MATLAB `iddata` variable is used.
- `DataLength`: Length of the data record.

- DataTs: Sampling interval of the data.
- DataDomain: 'Time' or 'Frequency', depending on the data domain.
- DataInterSample: Intersample behavior of the data from which the model was estimated. This equals the property InterSample of the iddata object. (See iddata.)
- WhyStop: For models that have been estimated by iterative search. The stopping rule that caused the iterations to terminate. Assumes values such as'MaxIter reached', 'No improvement possible along the search vector', or 'Near (local) minimum'. The latter means that the expected improvement is less than Tolerance (see Algorithm Properties).
- UpdateNorm: Norm of the Gauss-Newton vector at the last iteration.
- LastImprovement: Relative improvement of the criterion value at the last iteration.
- Iterations: Number of iterations used in the search.
- InitialState: Option actually used when Model.InitialState = 'auto'.
- N4Weight: For n4sid estimates, or estimates that have been initialized by n4sid: the actual value of N4Weight used.
- N4Horizon: For n4sid estimates, or estimates that have been initialized by n4sid: the actual value of N4Horizon used. See n4sid and Algorithm Properties.

idfrd Case

If the idfrd model is obtained from an estimated parametric model,

```
g = idfrd(Model)
```

g.EstimationInfo is the same as Model.EstimationInfo as described above.

For an idfrd model that has been estimated from etfe, spa, or spafdr, EstimationInfo contains the following fields:

- `Status`: Whether the model is estimated or directly constructed.
- `Method`: `etfe`, `spa`, or `spafdr`
- `WindowSize`: Resolution parameter (or vector) used for the estimation
- `DataName`, `DataLength`, `DataTs`, `DataDomain`, `DataInterSample`: Properties of the estimation data as above.

See Also

`Algorithm Properties`

`idpoly`

`idss`

Purpose Estimate empirical transfer functions and periodograms returning `idfrd` object

Syntax

```
g = etfe(data)
g = etfe(data,M,N)
```

Description `etfe` estimates the transfer function `g` of the general linear model

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

`data` contains the output-input data and is an `iddata` object (time or frequency domain).

`g` is given as an `idfrd` object with the estimate of $G(e^{i\omega})$ at the frequencies

```
w = [1:N]/N*pi/T
```

The default value of `N` is `128`.

In case `data` contains a time series (no input channels), `g` is returned as the periodogram of `y`.

When `M` is specified other than the default value `M = []`, a smoothing operation is performed on the raw spectral estimates. The effect of `M` is then similar to the effect of `M` in `spa`. This can be a useful alternative to `spa` for narrowband spectra and systems, which require large values of `M`.

When `etfe` is applied to time series, the corresponding spectral estimate is normalized in the way that is defined in “Spectrum Normalization and the Sampling Interval” on page 5-40. Note that this normalization might differ from the one used by `spectrum` in the Signal Processing Toolbox.

If the (input) data is marked as periodic (`data.Period = integer`) and contains an even number of periods, the response is computed at the frequencies `k*2*pi/period` for `k = 0` up to the Nyquist frequency.

Examples

Compare an empirical transfer function estimate to a smoothed spectral estimate.

```
ge = etfe(z);
gs = spa(z);
bode(ge,gs)
```

Generate a periodic input, simulate a system with it, and compare the frequency response of the estimated model with the true system at the excited frequency points.

```
m = idpoly([1 -1.5 0.7],[0 1 0.5]);
u = iddata([],idinput([50,1,10],'sine'));
u.Period = 50;
y = sim(m,u);
me = etfe([y u])
bode(me,'b*',m)
```

Algorithm

The empirical transfer function estimate is computed as the ratio of the output Fourier transform to the input Fourier transform, using `fft`. The periodogram is computed as the normalized absolute square of the Fourier transform of the time series.

You obtain the smoothed versions (`M` less than the length of `z`) by applying a Hamming window to the output fast Fourier transform (FFT) times the conjugate of the input FFT, and to the absolute square of the input FFT, respectively, and subsequently forming the ratio of the results. The length of this Hamming window is equal to the number of data points in `z` divided by `M`, plus one.

See Also

`bode`

`ffplot`

`freqresp`

`idfrd`

`nyquist`

`spa`

`spafdr`

Purpose

Value of nonlinearity estimator at given input

Syntax

```
value = evaluate(nl,x)
```

Arguments

`nl`

Nonlinearity estimator object.

`x`

Value at which to evaluate the nonlinearity.

If `nl` is a single nonlinearity estimator, then `x` is a `1`-by-`nx` row vector or an `nv`-by-`nx` matrix, where `nx` is the dimension of the regression vector input to `nl` (`size(nl)`) and `nv` is the number of points where `nl` is evaluated.

If `nl` is an array of `ny` nonlinearity estimators, then `x` is a `1`-by-`ny` cell array of `nv`-by-`nx` matrices.

Description

`value = evaluate(nl,x)` computes the value of a nonlinear estimator object of type `customnet`, `deadzone`, `linear`, `neuralnet`, `pwlinear`, `saturation`, `sigmoidnet`, `treepartition`, or `wavenet`.

Example

The following syntax evaluates the nonlinearity of an estimated nonlinear ARX model `m`:

```
value = evaluate(m.Nonlinearity,x)
```

where `m.Nonlinearity` accesses the nonlinearity estimator of the nonlinear ARX model.

See Also

`idnlarx`

`idnlhw`

Purpose Concatenate frequency-domain signals in `idfrd` and `iddata` objects

Syntax

```
Mc = fcat(M1,M2,...Mn)
```

Description `M1`, `M2`, etc., are all `idfrd` objects or `iddata` frequency-domain objects.

`Mc` is the corresponding object obtained by concatenation of the responses at all the frequencies in `Mk`.

Note that for `iddata` objects, this is the same as vertical concatenation (`vertcat`).

```
Mc = [M1;M2;..;Mn].
```

See Also

`fselect`

`iddata`

`idfrd`

Purpose

Identify possible feedback in `iddata` data

Syntax

```
[fbck,fbck0,nudir] = feedback(Data)
```

Description

`Data` is an `iddata` set with *Ny* outputs and *Nu* inputs.

`fbck` is an *Ny*-by-*Nu* matrix indicating the feedback. The *ky,ku* entry is a measure of feedback from output *ky* to input *ku*. The value is a probability *P* in percent. Its interpretation is that if the hypothesis that there is no feedback from output *ky* to input *ku* were tested at the level *P*, it would have been rejected. An intuitive but technically incorrect way of thinking about this is to see *P* as "the probability of feedback." Often only values above 90% are taken as indications of feedback. When `fbck` is calculated, direct dependence at lag zero between $u(t)$ and $y(t)$ is not regarded as a feedback effect.

`fbck0`: Same as `fbck`, but direct dependence at lag 0 between $u(t)$ and $y(t)$ is viewed as feedback effect.

`nudir`: A vector containing those input numbers that appear to have a direct effect on some outputs, that is, no delay from input to output.

See Also

`advice`

`iddata`

`idmodel`

ffplot

Purpose

Plot frequency response and spectra

Syntax

```
ffplot(m)
[mag,phase,w] = ffplot(m)
[mag,phase,w,sdmag,sdphase] = ffplot(m)
ffplot(m1,m2,m3,...,w)
ffplot(m1,'PlotStyle1',m2,'PlotStyle2',...)
ffplot(m1,m2,m3,..'sd',sd,'mode',mode,'ap',ap)
```

Description

This function has exactly the same syntax as `bode`. The only difference is that it gives graphs with linear frequency scales and Hz as the frequency unit.

See Also

`bode`

`etfe`

`freqresp`

`idfrd`

`nyquist`

`spa`

`spafdr`

Purpose

Transform `iddata` object to frequency domain

Syntax

```
Datf = fft(Data)
Datf = fft(Data,N)
Datf = fft(Data,N,'complex')
```

Description

If `Data` is a time-domain `iddata` object with real-valued signals and with constant sampling interval `Ts`, `Datf` is returned as a frequency-domain `iddata` object with the frequency values equally distributed from frequency 0 to the Nyquist frequency. Whether the Nyquist frequency actually is included or not depends on the signal length (even or odd). Note that the FFTs are normalized by dividing each transform by the square root of the signal length. That is in order to preserve the signal power and noise level.

In the default case, the length of the transformation is determined by the signal length. A second argument `N` will force FFT transformations of length `N`, padding with zeros if the signals in `Data` are shorter and truncating otherwise. Thus the number of frequencies in the real signal case will be `N/2` or `(N+1)/2`. If `Data` contains several experiments, `N` can be a row vector of corresponding length.

For real signals, the default is that `Datf` only contains nonnegative frequencies. For complex-valued signals, negative frequencies are also included. To enforce negative frequencies in the real case, add a last argument, `'Complex'`.

See Also

`iddata`

`ifft`

frd

Purpose

Convert `idfrd` objects to frequency-response LTI models of Control System Toolbox

Syntax

```
sys = frd(mod)
```

Description

`mod` is an `idfrd` object. `sys` is returned as an `frd` object.

The fields `Frequency, ResponseData, Units, Ts, InputDelay, InputName, OutputName` and `Notes` in `mod` are transferred to `sys`. The remaining fields (`SpectrumData, CovarianceData` and `NoiseCovariance`) are ignored. The command, therefore, cannot be applied to a time-series `idfrd` model object.

See Also

`ss`

`tf`

`zpk`

Purpose

Compute frequency function for model

Syntax

```
H = freqresp(m)
[H,w,covH] = freqresp(m,w)
```

Description

`m` is any `idmodel` or `idfrd` object.

`H = freqresp(m,w)` computes the frequency response `H` of the `idmodel` model `m` at the frequencies specified by the vector `w`. These frequencies should be real and in rad/s.

If `m` has `ny` outputs and `nu` inputs, and `w` contains `Nw` frequencies, the output `H` is an `ny`-by-`nu`-by-`Nw` array such that `H(:,:,k)` gives the complex-valued response at the frequency `w(k)`.

For a SISO model, `H(:)` to obtain a vector of the frequency response.

If `w` is not specified, a default choice is made based on the dynamics of the model.

Output Arguments

```
[H,w,covH] = freqresp(M,w)
```

also returns the frequencies `w` and the covariance `covH` of the response. `covH` is a 5-D array where `covH(ky,ku,k,:,:)` is the 2-by-2 covariance matrix of the response from input `ku` to output `ky` at frequency `w(k)`. The 1,1 element is the variance of the real part, the 2,2 element is the variance of the imaginary part, and the 1,2 and 2,1 elements are the covariance between the real and imaginary parts. `squeeze(covH(ky,ku,k,:,:))` gives the covariance matrix of the corresponding response.

If `m` is a time series (no input channels), `H` is returned as the (power) spectrum of the outputs, an `ny`-by-`ny`-by-`Nw` array. Hence `H(:,:,k)` is the spectrum matrix at frequency `w(k)`. The element `H(k1,k2,k)` is the cross spectrum between outputs `k1` and `k2` at frequency `w(k)`. When `k1 = k2`, this is the real-valued power spectrum of output `k1`.

`covH` is then the covariance of the estimated spectrum `H`, so that `covH(k1,k1,k)` is the variance of the power spectrum estimate of output `k1` at frequency `W(k)`. No information about the variance of the cross spectra is normally given; that is, `covH(k1,k2,k) = 0` for `k1` is not equal to `k2`.)

If the model `m` is not a time series, use `freqresp(m('n'))` to obtain the spectrum information of the noise (output disturbance) signals.

Note that `idfrd` computes the same information as `freqresp`, and stores it in the `idfrd` object.

See Also

`bode`

`etfe`

`ffplot`

`idfrd`

`nyquist`

`spa`

`spafdr`

Purpose

Akaike Final Prediction Error for estimated model

Syntax

```
fp = fpe(Model1,Model2,Model3,...)
```

Description

`Model` is any estimated `idmodel` (`idarx`, `idgrey`, `idpoly`, `idproc`, `idss`).

`fp` is returned as a row vector containing the values of the Akaike Final Prediction Error (FPE) for the different models. This is defined as

$$FPE = V\left(\frac{1+{}^{d}\!/\!{}_{N}}{1-{}^{d}\!/\!{}_{N}}\right)$$

where V is the loss function, d is the number of estimated parameters, and N is the number of estimation data.

The loss function V is

$$V = \det\left(\frac{1}{N}\sum_{1}^{N}\varepsilon\left(t,\theta_N\right)\left(\varepsilon\left(t,\theta_N\right)\right)^T\right)$$

where $\hat{\theta}_N$ represents the estimated parameters.

FPE can be negative when the number of estimated parameters exceeds the number of data samples, which can occur for models with multiple outputs. For models with multiple output, the assumption that d/N is small is not valid. In when this assumption is not valid, use AIC instead.

References

Sections 7.4 and 16.4 in Ljung (1999).

See Also

`EstimationInfo`

`aic`

fselect

Purpose

Select frequencies from `idfrd` object

Syntax

```
idfm = fselect(idf,index)
idfm = fselect(idf,Fmin,Fmax)
```

Description

`idf` is any `idfrd` object. `index` is a row vector of frequency indices, so that `idfm` is the `idfrd` object that contains the response at frequencies `idf.Frequency(Index)`.

If `Fmin` and `Fmax` are specified, `idfm` contains responses at frequencies between `Fmin` and `Fmax`.

Note that the operation is the same as `dat(index)` for an `iddata` object.

Examples

Select every fifth frequency:

```
idfm = fselect(idf,5:5:100)
```

Select the response in the third quadrant:

```
ph = angle(squeeze(idf.response));
idfm = fselect(idf,find(ph>-pi & ph <-pi/2))
```

See Also

`fcat`

`iddata`

`idfrd`

Purpose

Query properties of data and model objects

Syntax

```
Value = get(m,'PropertyName')
get(m)
Struct = get(m)
```

Description

`value = get(m,'PropertyName')` returns the current value of the property `PropertyName` of the `iddata` object or `idfrd` object, or `idmodel` object (`idgrey`, `idarx`, `idpoly`, `idss`), or `idnlgrey`, `idnlarx`, or `idnlhw` model object.

The string `'PropertyName'` can be the full property name (for example, `'SSParameterization'`) or any unambiguous case-insensitive abbreviation (for example, `'ss'`).

`Struct = get(m)` converts the object `m` into a standard MATLAB structure with the property names as field names and the property values as field values.

Without a left-hand argument

```
get(m)
```

displays all properties of `m` and their values.

Remarks

An alternative to the syntax

```
Value = get(m,'PropertyName')
```

is the structure-like referencing

```
Value = m.PropertyName
```

See Also

`Algorithm Properties`

`idarx`

`idfrd`

idgrey

idnlarx

idnlgrey

idnlhw

idpoly

idproc

idss

Purpose

Retrieve experiment(s) from multiple-experiment `iddata` objects

Syntax

```
d1 = getexp(data,ExperimentNumber)
d1 = getexp(data,ExperimentName)
```

Description

`data` is an `iddata` object that contains several experiments. `d1` is another `iddata` object containing the indicated experiment(s). The reference can either be by `ExperimentNumber`, as in `d1 = getexp(data,3)` or `d1 = getexp(data,[4 2])`; or by `ExperimentName`, as in `d1 = getexp(data,'Period1')` or `d1 = getexp(data,{'Day1','Day3'})`.

See `merge (iddata)` and `iddata` for how to create multiple-experiment data objects.

You can also retrieve the experiments using a fourth subscript, as in `d1 = data(:,:,:,ExperimentNumber)`. Type `help iddata/subsref` for details on this.

getinit

Purpose

Values of `idnlgrey` model initial states

Syntax

```
getinit(model)
getinit(model,prop)
```

Arguments

`model`
: Name of the `idnlgrey` model object.

`Property`
: Name of the `InitialStates` model property field, such as `'Name'`, `'Unit'`, `'Value'`, `'Minimum'`, `'Maximum'`, and `'Fixed'`.

  Default: `'Value'`.

Description

`getinit(model)` gets the initial-state values in the `'Value'` field of the `InitialStates` model property.

`getinit(model,prop)` gets the initial-state values of the `prop` field of the `InitialStates` model property. `prop` can be `'Name'`, `'Unit'`, `'Value'`, `'Minimum'`, `'Maximum'`, and `'Fixed'`.

The returned values are an `Nx`-by-1 cell array of values, where `Nx` is the number of states.

See Also

`getpar`

`idnlgrey`

`setinit`

`setpar`

Purpose

Parameter values and properties of `idnlgrey` model parameters

Syntax

```
getpar(model)
getpar(model,prop)
```

Arguments

`model`
: Name of the `idnlgrey` model object.

`Property`
: Name of the `Parameters` model property field, such as `'Name'`, `'Unit'`, `'Value'`, `'Minimum'`, or `'Maximum'`.

  Default: `'Value'`.

Description

`getpar(model)` gets the model parameter values in the `'Value'` field of the `Parameters` model property.

`getpar(model,prop)` gets the model parameter values in the `prop` field of the `Parameters` model property. `prop` can be `'Name'`, `'Unit'`, `'Value'`, `'Minimum'`, and `'Maximum'`.

The returned values are an `Np`-by-1 cell array of values, where `Np` is the number of parameters.

See Also

`getinit`

`idnlgrey`

`setinit`

`setpar`

Purpose

Returns names of standard or custom regressors in nonlinear ARX model

Syntax

```
getreg(model)
getreg(model,subset)
R = getreg(model,subset)
```

Arguments

`model`
: Name of the `idnlarx` model object.

`subset`
: Has one of the following values: `'all'`, `'input'`, `'output'`, `'standard'`, `'custom'`, `'linear'`, and `'nonlinear'`.

Description

`getreg(model)` returns the regressor expressions for all of the regressors in the nonlinear ARX model.

`getreg(model,subset)` returns the regressor expressions for a specified subset of the regressors, as follows:

- `'all'` — All regressors.
- `'input'` — Input regressors only.
- `'output'` — Output regressors only.
- `'standard'` — Standard regressors only.
- `'custom'` — Custom regressors only.
- `'linear'` — Regressors that are not used in the in the nonlinear block.
- `'nonlinear'` — Regressors used in the nonlinear block.

Note You can use `'nl'` as an abbreviation for `'nonlinear'`.

`R = getreg(model,subset)` returns a cell array of strings of the regressors for a specified subset of the regressors. For multiple-output, returns a cell array of cell arrays.

See Also

`addreg`

`customreg`

`nlarx`

`polyreg`

idarx

Purpose

Class for storing multioutput ARX polynomials and estimated impulse- and step-response

Syntax

```
m = idarx(A,B,Ts)
m = idarx(A,B,Ts,'Property1',Value1,...,,'PropertyN',ValueN)
```

Description

`idarx` creates an object containing parameters that describe the general multiinput, multioutput model structure of ARX type.

$$y(t) + A_1 y(t-1) + A_2 y(t-2) + \ldots + A_{na} y(t-na) =$$

$$B_0 u(t) + B_1 u(t-1) + \ldots + B_{nb} u(t-nb) + e(t)$$

Here A_k and B_k are matrices of dimensions *ny*-by-*ny* and *ny*-by-*nu*, respectively. (*ny* is the number of outputs, that is, the dimension of the vector $y(t)$, and *nu* is the number of inputs.)

The arguments `A` and `B` are 3-D arrays that contain the `A` matrices and the `B` matrices of the model in the following way.

`A` is an `ny`-by-`ny`-by-`(na+1)` array such that

```
A(:,:,k+1) = Ak
A(:,:,1) = eye(ny)
```

Similarly `B` is an `ny`-by-`nu`-by-`(nb+1)` array with

```
B(:,:,k+1) = Bk
```

Note that `A` always starts with the identity matrix, and that delays in the model are defined by setting the corresponding leading entries in `B` to zero. For a multivariate time series take `B = []`.

The optional property `NoiseVariance` sets the covariance matrix of the driving noise source $e(t)$ in the model above. The default value is the identity matrix.

The argument `Ts` is the sampling interval. Note that continuous-time models (`Ts` = 0) are not supported.

The use of idarx is twofold. You can use it to create models that are simulated (using sim) or analyzed (using bode, pzmap, etc.). You can also use it to define initial value models that are further adjusted to data (using arx). The free parameters in the structure are consistent with the structure of A and B; that is, leading zeros in the rows of B are regarded as fixed delays, and trailing zeros in A and B are regarded as a definition of lower-order polynomials. These zeros are fixed, while all other parameters are free.

For a model with one output, ARX models can be described both as idarx and idpoly models. The internal representation is different, however.

idarx Properties

- A, B: The A and B polynomials as 3-D arrays, described above
- dA, dB: The standard deviations of A and B. Same format as A and B. Cannot be set.
- na, nb, nk: The orders and delays of the model. na is an ny-by-ny matrix whose *i-j* entry is the order of the polynomial corresponding to the *i-j* entry of A. Similarly nb is an ny-by-nu matrix with the orders of B. nk is also an ny-by-nu matrix, whose *i-j* entry is the delay from input *j* to output *i*, that is, the number of leading zeros in the *i-j* entry of B.
- InitialState: This describes how the initial state (initial values in filtering, etc.) should be handled. For time-domain applications, this is typically handled by starting the filtering when all data are available. For frequency-domain data, you must estimate initial states. The possible values of InitialState are 'zero', 'estimate', and 'auto' (which makes a data-dependent choice between zero and estimate).

In addition to these properties, idarx objects also have all the properties of the idmodel object. See idmodel, Algorithm Properties, and EstimationInfo.

Note that you can set and retrieve all properties either with the `set` and `get` commands or by subscripts. Autofill applies to all properties and values, and they are case insensitive.

For a complete list of property values, use `get(m)`. To see possible value assignments, use `set(m)`. See also `idprops idarx`.

Examples

Simulate a second-order ARX model with one input and two outputs, and then estimate a model using the simulated data.

```
A = zeros(2,2,3);
B = zeros(2,1,3)
A(:,:,1) =eye(2);
A(:,:,2) = [-1.5 0.1;-0.2 1.5];
A(:,:,3) = [0.7 -0.3;0.1 0.7];
B(:,:,2) = [1;-1];
B(:,:,3) = [0.5;1.2];
m0 = idarx(A,B,1);
u = iddata([],idinput(300));
e = iddata([],randn(300,2));
y = sim(m0,[u e]);
m = arx([y u],[[2 2;2 2],[2;2],[1;1]]);
```

See Also

`arx`

`arxdata`

`idmodel`

`idpoly`

Purpose

Class for storing time-domain and frequency-domain data

Syntax

```
data = iddata(y,[],Ts)
data = iddata(y,u,Ts)
data = iddata(y,u,Ts,'Frequency',W)
data = iddata(y,u,Ts,'P1',V1,...,'PN',VN)
data = iddata(idfrd_object)
```

Arguments

`y`

Name of MATLAB variable that represents the output signal from the dynamic system. Sets the `OutputData` `iddata` property. For a single-output system, this is a column vector. For a multioutput system with N_y output channels and N_T time samples, this is an N_T-by-N_y matrix.

Note Output data must be in the same domain as input data.

`u`

Name of MATLAB variable that represents the input signal for the dynamic system. Sets the `InputData` `iddata` property. For a single-input system, this is a column vector. For a multioutput system with N_u output channels and N_T time samples, this is an N_T-by-N_u matrix.

Note Input data must be in the same domain as output data.

`Ts`

Time interval between successive data samples in seconds. Default value is `1`. For continuous-time data in the frequency domain, set `Ts` to `0`.

`'P1',V1,...,'PN',VN`

Pairs of `iddata` property names and property values.

`idfrd_object`
: Name of `idfrd` data object.

Description

`data = iddata(y,[],Ts)` creates an `iddata` object for time-series data, containing a time-domain output signal `y` and an empty input signal `[]`, respectively. `Ts` specifies the sampling interval of the experimental data.

`data = iddata(y,u,Ts)` creates an `iddata` object containing a time-domain output signal `y` and input signal `u`, respectively. `Ts` specifies the sampling interval of the experimental data.

`data = iddata(y,u,Ts,'Frequency',W)` creates an `iddata` object containing a frequency-domain output signaly and input signal `u`, respectively.`Ts` specifies the sampling interval of the experimental data. `W` specifies the `iddata` property `'frequency'` as a vector of frequencies.

`data = iddata(y,u,Ts,'P1',V1,...,'PN',VN)` creates an `iddata` object containing a time-domain or frequency-domain output signal `y` and input signal `u`, respectively. `Ts` specifies the sampling interval of the experimental data. `'P1',V1,...,'PN',VN` are property-value pairs, as described in "iddata Properties" on page 12-102.

`data = iddata(idfrd_object)` transforms an `idfrd` object to a frequency-domain `iddata` object.

iddata Properties

The following table describes `iddata` object properties and their values. These properties are specified as property-value arguments `'P1',V1,...,'PN',VN`' in the `iddata` constructor, or you can set them using the `set` command or dot notation. In the list below, `N` denotes the number of data samples in the input and output signals, `ny` is the number of output channels, `nu` is the number of input channels, and `Ne` is the number of experiments.

Tip Property names are not case sensitive. You do not need to type the entire property name. However, the portion you enter must by enough to uniquely identify the property.

Property Name	Description	Value
`Domain`	Specifies whether the data is in the time domain or frequency domain.	• `'Frequency'` — Frequency-domain data. • `'Time'` (Default) — Time-domain data.
`ExperimentName`	Name of each data set contained in the `iddata` object.	For `Ne` experiments, a `1`-by-`Ne` cell array of strings. Each cell contains the name of the corresponding experiment. Default names are `{'Exp1', 'Exp2',...}`.
`Frequency`	(Frequency-domain data only) Frequency values for defining the Fourier Transforms of the signals.	For a single experiment, this is an `N`-by-`1` vector. For `Ne` experiments, a `1`-by-`Ne` cell array and each cell contains the frequencies of the corresponding experiment.
`InputData`	Name of MATLAB variable that stores the input signal for the dynamic system.	For `nu` input channels and `N` data samples, this is an `N`-by-`nu` matrix.
`InputName`	Specifies the names of individual input channels.	Cell array of length `nu`-by-`1` contains the name string of each input channel. Default names are `{'u1';'u2';...}`.
`InputUnit`	Specifies the units of each input channel.	Cell array of length `nu`-by-`1`. Each cell contains a string that specifies the units of each input channel.

Property Name	Description	Value
`InterSample`	Specifies the behavior of the input signals between samples for transformations between discrete-time and continuous-time.	For a single experiment: • `zoh`— (Default) Zero-order hold maintains a piecewise-contant input signal between samples. • `foh`— First-order hold maintains a piecewise-linear input signal between samples. • `bl`— Band-limited behavior specifies that the continuous-time input signal has zero power above the Nyquist frequency. For `Ne` experiments, `InterSample` is an `nu`-by-`Ne` cell array. Each cell contains one of these values corresponding to each experiment.
`Name`	Name of the data set.	Text string.
`Notes`	Comments about the data set.	Text string.
`OutputData`	Name of MATLAB variable that stores the output signal from the dynamic system.	For `ny` output channels and `N` samples, this is an `N`-by-`ny` matrix.

Property Name	Description	Value
`OutputName`	For a multioutput system, specifies the names of individual output channels.	Cell array of length `ny`-by-1 contains the name string of each output channel. Default names are `{'y1';'y2';...}`.
`OutputUnit`	Specifies the units of each output channel.	For `ny` output channels, a cell array of length `ny`-by-1. Each cell contains a string that specifies the units of the corresponding output channel.
`Period`	Period of the input signal.	(Default) For a nonperiodic signal, set to `inf`. For a multiinput signal, this is an `nu`-by-1 vector and the `k`th entry contains the period of the `k`th input. For `Ne` experiments, this is a 1-by-`Ne` cell array and each cell contains a scalar or vector of periods for the corresponding experiment.
`SamplingInstants`	(Time-domain data only) The time values in the time vector calculated from the properties `Tstart` and `Ts`.	For a single experiment, this is an `N`-by-1 vector. For `Ne` experiments, this is a 1-by-`Ne` cell array and each cell contains the sampling instants of the corresponding experiment.
`TimeUnit`	(Time-domain data only) Time unit.	A string that specifies the time unit for the time vector.

Property Name	Description	Value
`Ts`	Time interval between successive data samples in seconds. Must be specified for both time- and frequency-domain data. For frequency-domain, it is used to compute Fourier transforms of the signals as discrete-time Fourier transforms (DTFT) with the indicated sampling interval. **Note** System Identification Toolbox requires that your data be uniformly sampled.	Default value is `1`. For continuous-time data in the frequency domain, set to `0`; the inputs and outputs are interpreted as continuous-time Fourier transforms of the signals. Note that `Ts` is essential also for frequency-domain data, for proper interpretation of how the Fourier transforms were computed: They are interpreted as discrete-time Fourier transforms (DTFT) with the indicated sampling interval.. For multiple-experiment data, `Ts` is a `1`-by-`Ne` cell array and each cell contains the sampling interval of the corresponding experiment.
`Tstart`	(Time-domain data only) Specifies the start time of the time vector.	For a single experiment, this is a scalar. For `Ne` experiments, `Ts` is a `1`-by-`Ne` cell array and each cell contains the sampling interval of the corresponding experiment.

Property Name	Description	Value
`Units`	(Frequency-domain data only) Frequency unit.	Specified as `rad/s` or `Hz`.For multiexperiement data with `Ne` experiments, `Units` is a `1`-by-`Ne` cell array and each cell contains the frequency unit for each experiment.
`UserData`	Additional comments.	Text string.

See Also

`advice`

`detrend`

`fcat`

`getexp`

`idfilt`

`idfrd`

`plot`

`resample`

`size`

Purpose

Open System Identification Tool GUI

Syntax

```
ident
ident(session,path)
```

Description

`ident` by itself opens the main interface window, or brings it forward if it is already open.

`session` is the name of a previous session with the graphical user interface, and typically has extension `.sid`. The `path` argument is the complete path for the location of this file. If the session file is on the `MATLABPATH`, `path` can be omitted.

When the session is specified, the interface will open with this session active. Typing `ident(session,path)` on the MATLAB command line, when the interface is active, will load and open the session in question.

For more information about the graphical user interface, see Chapter 2, "Working with the System Identification Tool GUI".

Examples

```
ident('iddata1.sid')
ident('mydata.sid','\matlab\data\cdplayer\')
```

See Also

`midprefs`

Purpose

Filter data using user-defined passbands, general filters, or Butterworth filters

Syntax

```
Zf = idfilt(Z,filter)
Zf = idfilt(Z,filter,causality)
Zf = idfilt(Z,filter,'FilterOrder',NF)
```

Description

`Z` is the data, defined as an `iddata` object. `Zf` contains the filtered data as an `iddata` object. The filter can be defined in three ways:

- As an explicit system that defines the filter,

  ```
  filter = idm or filter = {num,den} or filter = {A,B,C,D}
  ```

  `idm` can be any SISO `idmodel` or LTI model object. Alternatively the filter can be defined as a cell array `{A,B,C,D}` of SISO state-space matrices or as a cell array `{num,den}` of numerator/denominator filter coefficients.

- As a vector or matrix that defines one or several passbands,

  ```
  filter=[[wp1l,wp1h];[ wp2l,wp2h]; ....;[wpnl,wpnh]]
  ```

  The matrix is `n`-by-2, where each row defines a passband in rad/s. A filter is constructed that gives the union of these passbands. For time-domain data, it is computed as cascaded Butterworth filters or order NF. The default value of NF is `5`.

  For example, to define a stopband between `ws1` and `ws2`, use

  ```
  filter = [0 ws1; ws2,Nyqf]
  ```

  where `Nyqf` is the Nyquist frequency.

- For frequency-domain data, only the frequency response of the filter can be specified:

  ```
  filter = Wf
  ```

Here `Wf` is a vector of possibly complex values that define the filter's frequency response, so that the inputs and outputs at frequency `Z.Frequency(kf)` are multiplied by `Wf(kf)`. `Wf` is a column vector of length = number of frequencies in `Z`. If the data object has several experiments, `Wf` is a cell array of length = # of experiments in `Z`.

For time-domain data, the filtering is carried out in the time domain as causal filtering as default. This corresponds to a last argument `causality = 'causal'`. With `causality = 'noncausal'`, a noncausal, zero-phase filter is used for the filtering (corresponding to `filtfilt` in the Signal Processing Toolbox).

For frequency-domain data, the signals are multiplied by the frequency response of the filter. With the filters defined as passband, this gives ideal, zero-phase filtering ("brickwall filters"). Frequencies that have been assigned zero weight by the filter (outside the passband, or via the frequency response) are removed from the `iddata` object `Zf`.

It is common practice in identification to select a frequency band where the fit between model and data is concentrated. Often this corresponds to bandpass filtering with a passband over the interesting breakpoints in a Bode diagram. For identification where a disturbance model is also estimated, it is better to achieve the desired estimation result by using the property `'Focus'` (see `Algorithm Properties`) than just to prefilter the data. The proper values for `'Focus'` are the same as the argument `filter` in `idfilt`.

Algorithm

The Butterworth filter is the same as `butter` in the Signal Processing Toolbox. Also, the zero-phase filter is equivalent to `filtfilt` in that toolbox.

References

Ljung (1999), Chapter 14.

See Also

`Algorithm Properties`

`iddata`

Purpose

Class for storing frequency-response or spectral-analysis data

Syntax

```
h = idfrd(Response,Freq,Ts)
h = idfrd(Response,Freq,Ts,
   'CovarianceData',Covariance, ...
'SpectrumData',Spec,
   'NoiseCovariance',Speccov,'P1', ...
 V1,'PN',
   VN)
h = idfrd(mod)
h = idfrd(mod,Freqs)
```

Description

```
h = idfrd(Response,Freq,Ts)
h = idfrd(Response,Freq,Ts,'CovarianceData',Covariance, ...
'SpectrumData',Spec,'NoiseCovariance',Speccov,'P1', ...
 V1,'PN',VN)
h = idfrd(mod)
h = idfrd(mod,Freqs)
```

`idfrd` creates the `idfrd` model object.

For a model

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

stores the transfer function estimate G

$$G(e^{i\omega})$$

as well as the spectrum of the additive noise (Φ_v) at the output

$$\Phi_v(\omega) = \lambda T\left|H(e^{i\omega T})\right|^2$$

where λ is the estimated variance of $e(t)$, and T is the sampling interval.

Creating idfrd from Given Responses

`Response` is a 3-D array of dimension `ny`-by-`nu`-by-`Nf`, with `ny` being the number of outputs, `nu` the number of inputs, and `Nf` the number of frequencies (that is, the length of `Freqs`). `Response(ky,ku,kf)` is thus the complex-valued frequency response from input `ku` to output `ky` at frequency ω=`Freqs(kf)`. When defining the response of a SISO system, `Response` can be given as a vector.

`Freqs` is a column vector of length `Nf` containing the frequencies of the response.

`Ts` is the sampling interval. `T = 0` means a continuous-time model.

`Covariance` is a 5-D array containing the covariance of the frequency response. It has dimension `ny`-by-`nu`-by-`Nf`-by-2-by-2. The structure is such that `Covariance(ky,ku,kf,:,:)` is the 2-by-2 covariance matrix of the response `Response(ky,ku,kf)`. The 1-1 element is the variance of the real part, the 2-2 element is the variance of the imaginary part, and the 1-2 and 2-1 elements are the covariance between the real and imaginary parts. `squeeze(Covariance(ky,ku,kf,:,:))` thus gives the covariance matrix of the corresponding response.

The information about spectrum is optional. The format is as follows:

`spec` is a 3-D array of dimension `ny`-by-`ny`-by-`Nf`, such that `spec(ky1,ky2,kf)` is the cross spectrum between the noise at output `ky1` and the noise at output `ky2`, at frequency `Freqs(kf)`. When `ky1 = ky2` the (power) spectrum of the noise at output `ky1` is thus obtained. For a single-output model, `spec` can be given as a vector.

`speccov` is a 3-D array of dimension `ny`-by-`ny`-by-`Nf`, such that `speccov(ky1,ky1,kf)` is the variance of the corresponding power spectrum. Normally, no information is included about the covariance of the nondiagonal spectrum elements.

If only `SpectrumData` is to be packaged in the `idfrd` object, set `Response = []`.

Creating idfrd from a Given Model

`idfrd` can also be computed from a given model `mod` (defined as any `idmodel` object).

If the frequencies `Freqs` are not specified, a default choice is made based on the dynamics of the model `mod`.

If mod has InputDelay different from zero, these are appended as phase lags, and h will then have an InputDelay of 0.

The estimated covariances are computed using the Gauss approximation formula from the uncertainty information in mod. For models with complicated parameter dependencies, numerical differentiation is applied. The step sizes for the numerical derivatives are determined by nuderst.

Frequency responses for submodels can be obtained by the standard subreferencing, `h = idfrd(m(2,3))`. See idmodel. In particular, `h = idfrf(m('measured'))` gives an h that just contains the `ResponseData (G)` and no spectra. Also `h = idfrd(m('noise'))` gives an h that just contains SpectrumData.

The idfrd models can be graphed with bode, ffplot, and nyquist, which all accept mixtures of idmodel and idfrd models as arguments. Note that spa, spafdr, and etfe return their estimation results as idfrd objects.

idfrd Properties

- ResponseData: 3-D array of the complex-valued frequency response as described above. For SISO systems use `Response(1,1,:)` to obtain a vector of the response data.
- Frequency: Column vector containing the frequencies at which the responses are defined.
- CovarianceData: 5-D array of the covariance matrices of the response data as described abfove.
- SpectrumData: 3-D array containing power spectra and cross spectra of the output disturbances (noise) of the system.
- NoiseCovariance: 3-D array containing the variances of the power spectra, as explained above.
- Units: Unit of the frequency vector. Can assume the values 'rad/s' and 'Hz'.

- Ts: Scalar denoting the sampling interval of the model whose frequency response is stored. 'Ts' = 0 means a continuous-time model.
- Name: An optional name for the object.
- InputName: String or cell array containing the names of the input channels. It has as many entries as there are input channels.
- OutputName: Correspondingly for the output channels.
- InputUnit: Units in which the input channels are measured. It has the same format as 'InputName'.
- OutputUnit: Correspondingly for the output channels.
- InputDelay: Row vector of length equal to the number of input channels. Contains the delays from the input channels. These should thus be appended as phase lags when the response is calculated. This is done automatically by freqresp, bode, ffplot, and nyquist. Note that if the idfrd is calculated from an idmodel, possible input delays in that model are converted to phase lags, and the InputDelay of the idfrd model is set to zero.
- Notes: An arbitrary field to store extra information and notes about the object.
- UserData: An arbitrary field for any possible use.
- EstimationInfo: Structure that contains information about the estimation process that is behind the frequency data. It contains the following fields (see also the reference page for EstimationInfo).
  - Status: Gives the status of the model, for example, 'Not estimated'.
  - Method: The identification routine that created the model.
  - WindowSize: If the model was estimated by spa, spafdr, or etfe, the size of window (input argument M, the resolution parameter) that was used. This is scalar or a vector.

- `DataName`: Name of the data set from which the model was estimated.
- `DataLength`: Length of this data set.

Note that you can set or retrieve all properties either with the `set` and `get` commands or by subscripts. Autofill applies to all properties and values, and these are case insensitive:

```
h.ts = 0
loglog(h.fre,squeeze(h.spe(2,2,:)))
```

For a complete list of property values, use `get(m)`. To see possible value assignments, use `set(m)`. See also `idprops idfrd`.

Subreferencing

The different channels of the `idfrd` are retrieved by subreferencing.

```
h(outputs,inputs)
```

`h(2,3)` thus contains the response data from input channel 3 to output channel 2, and, if applicable, the output spectrum data for output channel 2. The channels can also be referred to by their names, as in `h('power',{'voltage', 'speed'})`.

```
h('m')
```

contains the information for measured inputs only, that is, just `ResponseData`, while

```
h('n')
```

(`'n'` for `'noise'`) just contains `SpectrumData`.

Horizontal Concatenation

Adding input channels,

```
h = [h1,h2,...,hN]
```

creates an `idfrd` model `h`, with `ResponseData` containing all the input channels in `h1,...,hN`. The output channels of `hk` must be the same, as well as the frequency vectors. `SpectrumData` is ignored.

Vertical Concatenation

Adding output channels,

```
h = [h1;h2;... ;hN]
```

creates an `idfrd` model `h` with `ResponseData` containing all the output channels in `h1, h2,...,hN`. The input channels of `hk` must all be the same, as well as the frequency vectors. `SpectrumData` is also appended for the new outputs. The cross spectrum between output channels is then set to zero.

Converting to iddata

You can convert an `idfrd` object to a frequency-domain `iddata` object by

```
Data = iddata(Idfrdmodel)
```

See `iddata`.

Examples

Compare the results from spectral analysis and an ARMAX model.

```
m = armax(z,[2 2 2 1]);
g = spa(z)
g = spafdr(z,[],{0,10})
bode(g,m)
```

Compute separate `idfrd` models, one containing the frequency function and the other the noise spectrum.

```
g = idfrd(m('m'))
phi = idfrd(m('n'))
```

See Also

`bode`

`etfe`

`ffplot`

`freqresp`

`nyquist`

`spa`

`spafdr`

Purpose

Class for storing linear grey-box models

Syntax

```
m = idgrey(MfileName,ParameterVector,CDmfile)
m = idgrey(MfileName,ParameterVector,CDmfile,FileArgument,Ts,...
'Property1',Value1,...,'PropertyN',ValueN)
```

Description

The function `idgrey` is used to create arbitrarily parameterized state-space models as `idgrey` objects.

`MfileName` is the name of an M-file that defines how the state-space matrices depend on the parameters to be estimated. The format of this M-file is given by

```
[A,B,C,D,K,X0] = mymfile(pars,Tsm,Auxarg)
```

and is further discussed below.

`ParameterVector` is a column vector of the nominal/initial parameters. Its length must be equal to the number of free parameters in the model (that is, the argument `pars` in the example below).

The argument `CDmfile` describes how the user-written M-file handles continuous and discrete-time models. It takes the following values:

- `CDmfile = 'cd'`: The M-file returns the continuous-time state-space matrices when called with the argument `Tsm = 0`. When called with a value `Tsm > 0`, the M-file returns the discrete-time state-space matrices, obtained by sampling the continuous-time system with sampling interval `Tsm`. The M-file must consequently in this case include the sampling procedure.
- `CDmfile = 'c'`. The M-file always returns the continuous-time state-space matrices, no matter the value of `Tsm`. In this case the toolbox's estimation routines will provide the sampling when you are fitting the model to discrete-time data.
- `CDmfile = 'd'`. The M-file always returns discrete-time state-space matrices that may or may not depend on `Tsm`.

The argument `FileArgument` corresponds to the auxiliary argument `Auxarg` in the user-written M-file. It can be used to handle several variants of the model structure, without your having to edit the M-file. If it is not used, enter `FileArgument = []`. (Default.)

`Ts` denotes the sampling interval of the model. Its default value is `Ts = 0`, that is, a continuous-time model.

The `idgrey` object is a child of `idmodel`. Therefore any `idmodel` properties can be set as property name/property value pairs in the `idgrey` command. They can also be set by the command `set`, or by subassignment, as in

```
m.InputName = {'speed','voltage'}
m.FileArgument = 0.23
```

There are also two properties, `DisturbanceModel` and `InitialState`, that can be used to affect the parameterizations of *K* and *X0*, thus overriding the outputs from the M-file.

idgrey Properties

- `MfileName`: Name of the user-written M-file.
- `CDmfile`: How this file handles continuous and discrete-time models depending on its second argument, `T`.
  - `CDmfile = 'cd'` means that the M-file returns the continuous-time state-space model matrices when the argument `T = 0`, and the discrete-time model, obtained by sampling with sampling interval `T`, when `T > 0`.
  - `CDmfile = 'c'` means that the M-file always returns continuous-time model matrices, no matter the value of T.
  - `CDmfile = 'd'` means that the M-file always returns discrete-time model matrices that may or may not depend on the value of `T`.
- `FileArgument`: Possible extra input arguments to the user-written M-file.
- `DisturbanceModel`: Affects the parameterization of the K matrix. It can assume the following values:

- 'Model': This is the default. It means that the K matrix obtained from the user-written M-file is used.
- 'Estimate': The K matrix is treated as unknown and all its elements are estimated as free parameters.
- 'Fixed': The K matrix is fixed to a given value.
- 'None': The K matrix is fixed to zero, thus producing an output-error model.

Note that in the three last cases the output K from the user-written M-file is ignored. The estimated/fixed value is stored internally and does not change when the model is sampled, resampled, or converted to continuous time. Note also that this estimated value is tailored only to the sampling interval of the data.

- InitialState: Affects the parameterization of the X0 vector. It can assume the following values:
  - 'Model': This is the default. It means that the X0 vector is obtained from the user-written M-file.
  - 'Estimate': The X0 matrix is treated as unknown and all its elements are estimated as free parameters.
  - 'Fixed': The X0 vector is fixed to a given value.
  - 'Backcast': The X0 vector is estimated using a backcast operation analogous to the idss case.
  - 'Auto': Makes a data-dependent choice among 'Estimate', 'Backcast', and 'Model'.
- A, B, C, D, K, and X0: The state-space matrices. For idgrey models, only 'K' and 'X0' can be set; the others can only be retrieved. The set 'K' and 'X0' are relevant only when DisturbanceModel/InitialState are Estimate or Fixed.
- dA, dB, dC, dD, dK, and dX0: The estimated standard deviations of the state-space matrices. These cannot be set, only retrieved.

In addition, any `idgrey` object also has all the properties of `idmodel`. See `Algorithm Properties` and the reference page for `idmodel`.

Note that you can set or retrieve all properties using either the `set` and `get` commands or subscripts. Autofill applies to all properties and values, and they are case insensitive.

```
m.fi = 10;
set(m,'search','gn')
p = roots(m.a)
```

For a complete list of property values, use `get(m)`. To see possible value assignments, use `set(m)`. See also `idprops` and `idgrey`.

M-File Details

The model structure corresponds to the general linear state-space structure

$$\tilde{x}(t) = A(\theta)x(t) + B(\theta)u(t) + K(\theta)e(t)$$

$$x(0) = x_0(\theta)$$

$$y(t) = C(\theta)x(t) + D(\theta)u(t) + e(t)$$

Here $\tilde{x}(t)$ is the time derivative $\dot{x}(t)$ for a continuous-time model and $x(t + Ts)$ for a discrete-time model.

The matrices in this time-discrete model can be parameterized in an arbitrary way by the vector θ. Write the format for the M-file as follows:

```
[A,B,C,D,K,x0] = mymfile(pars,T,Auxarg)
```

Here the vector `pars` contains the parameters θ, and the output arguments `A`, `B`, `C`, `D`, `K`, and `x0` are the matrices in the model description that correspond to this value of the parameters and this value of the sampling interval `T`.

`T` is the sampling interval, and `Auxarg` is any variable of auxiliary quantities with which you want to work. (In that way you can change certain constants and other aspects in the model structure without having to edit the M-file.) Note that the two arguments `T` and `Auxarg` must be included in the function head of the M-file, even if they are not used within the M-file.

Chapter 7, "Estimating Grey-Box Models" contains several examples of typical M-files that define model structures.

A comment about `CDmfile`: If a continuous-time model is sought, it is easiest to let the M-file deliver just the continuous-time model, that is, have `CDmfile = 'c'` and rely upon the toolbox's routines for the proper sampling. Similarly, if the underlying parameterization is indeed discrete time, it is natural to deliver the discrete-time model matrices and let `CDmfile = 'd'.` If the underlying parameterization is continuous, but you prefer for some reason to do your own sampling inside the M-file in accordance with the value of `T`, then let your M-file deliver the continuous-time model when called with `T = 0`, that is, the alternative `CMmfile = 'cd'`. This avoids sampling and then transforming back (using `d2c`) to find the continuous-time model.

Examples

Use the M-file `mynoise` given in "Linear Grey-Box Models" on page 7-5 to obtain a physical parameterization of the Kalman gain.

```
mn = idgrey('mynoise',[0.1,-2,1,3,0.2]','d')
m = pem(z,mn)
```

Purpose

Generate input signals

Syntax

```
u = idinput(N)
u = idinput(N,type,band,levels)
[u,freqs] = idinput(N,'sine',band,levels,sinedata)
```

Description

`idinput` generates input signals of different kinds, which are typically used for identification purposes. `u` is returned as a matrix or column vector.

For further use in the toolbox, we recommend that you create an `iddata` object from `u`, indicating sampling time, input names, periodicity, and so on:

```
  u = iddata([],u);
```

`N` determines the number of generated input data. If `N` is a scalar, `u` is a column vector with this number of rows.

`N = [N nu]` gives an input with `nu` input channels each of length `N`.

`N = [P nu M]` gives a periodic input with `nu` channels, each of length `M*P` and periodic with period `P`.

Default is `nu = 1` and `M = 1`.

`type` defines the type of input signal to be generated. This argument takes one of the following values:

- `type = 'rgs'`: Gives a random, Gaussian signal.
- `type = 'rbs'`: Gives a random, binary signal. This is the default.
- `type = 'prbs'`: Gives a pseudorandom, binary signal.
- `type = 'sine'`: Gives a signal that is a sum of sinusoids.

The frequency contents of the signal is determined by the argument `band`. For the choices `type = 'rs'`, `'rbs'`, and `'sine'`, this argument is a row vector with two entries

```
band = [wlow, whigh]
```

that determine the lower and upper bound of the passband. The frequencies `wlow` and `whigh` are expressed in fractions of the Nyquist frequency. A white noise character input is thus obtained for `band = [0 1]`, which is also the default value.

For the choice `type = 'prbs'`,

```
band = [0, B]
```

where `B` is such that the signal is constant over intervals of length `1/B` (the clock period). In this case the default is `band = [0 1]`.

The argument `levels` defines the input level. It is a row vector

```
levels = [minu, maxu]
```

such that the signal `u` will always be between the values `minu` and `maxu` for the choices `type = 'rbs'`, `'prbs'`, and `'sine'`. For `type = 'rgs'`, the signal level is such that `minu` is the mean value of the signal, minus one standard deviation, while `maxu` is the mean value plus one standard deviation. Gaussian white noise with zero mean and variance one is thus obtained for `levels = [-1, 1]`, which is also the default value.

Some PRBS Aspects

If more than one period is demanded (that is, `M > 1`), the length of the data sequence and the period of the PRBS signal are adjusted so that an integer number of maximum length PRBS periods is always obtained. If `M = 1`, the period of the PRBS signal is chosen to that it is longer than `P = N`. In the multiinput case, the signals are maximally shifted. This means `P/nu` is an upper bound for the model orders that can be estimated with such a signal.

Some Sine Aspects

In the `'sine'` case, the sinusoids are chosen from the frequency grid

```
freq = 2*pi*[1:Grid_Skip:fix(P/2)]/P
```

intersected with `pi*[band(1) band(2)]`. For `Grid_Skip`, see below. For multiinput signals, the different inputs use different frequencies from this grid. An integer number of full periods is always delivered. The selected frequencies are obtained as the second output argument, `freqs`, where row `ku` of `freqs` contains the frequencies of input number `ku`. The resulting signal is affected by a fifth input argument, `sinedata`

```
sinedata = [No_of_Sinusoids, No_of_Trials, Grid_Skip]
```

meaning that `No_of_Sinusoids` is equally spread over the indicated `band`. `No_of_Trials` (different, random, relative phases) are tried until the lowest amplitude signal is found.

```
Default: sinedata = [10,10,1];
```

`Grid_Skip` can be useful for controlling odd and even frequency multiples, for example, to detect nonlinearities of various kinds.

Algorithm

Very simple algorithms are used. The frequency contents are achieved for `'rgs'` by an eighth-order Butterworth, noncausal filter, using `idfilt`. This is quite reliable. The same filter is used for the `'rbs'` case, before making the signal binary. This means that the frequency contents are not guaranteed to be precise in this case.

For the `'sine'` case, the frequencies are selected to be equally spread over the chosen grid, and each sinusoid is given a random phase. A number of trials are made, and the phases that give the smallest signal amplitude are selected. The amplitude is then scaled so as to satisfy the specifications of `levels`.

References

See Söderström and Stoica (1989), Chapter C5.3. For a general discussion of input signals, see Ljung (1999), Section 13.3.

Examples

Create an input consisting of five sinusoids spread over the whole frequency interval. Compare the spectrum of this signal with that of its square. The frequency splitting (the square having spectral support at other frequencies) reveals the nonlinearity involved:

```
u = idinput([100 1 20],'sine',[],[],[5 10 1]);
u = iddata([],u,1,'per',100);
u2 = u.u.^2;
u2 = iddata([],u2,1,'per',100);
ffplot(etfe(u),'r*',etfe(u2),'+')
```

Purpose

Simulate `idmodel` objects in Simulink

Syntax

```
idmdlsim
```

Description

Typing `idmdlsim` launches the Idmodel block in Simulink. By clicking the block you can specify the `idmodel` to simulate, whether to include initial state values, and whether to add noise to the simulation in accordance with the model's own noise description.

See Also

`compare`

`pe`

`predict`

`sim`

`simsd`

Purpose

Superclass for linear models

Description

`idmodel` is an object that the user does not deal with directly. It contains all the common properties of the model objects `idarx`, `idgrey`, `idpoly`, `idproc`, and `idss`, which are returned by the different estimation routines.

Basic Use

If you just estimate models from data, the model objects should be transparent. All parametric estimation routines return `idmodel` results.

```
m = arx(Data,[2 2 1])
```

The model `m` contains all relevant information. Just typing `m` will give a brief account of the model. `present(m)` also gives information about the uncertainties of the estimated parameters. `get(m)` gives a complete list of model properties.

Most of the interesting properties can be directly accessed by subreferencing.

```
m.a
m.da
```

See the property list obtained by `get(m)`, as well as the property lists of `idgrey`, `idarx`, `idpoly`, and `idss` in Chapter 12, "Functions–Alphabetical List" for more details on this. See also `idprops`.

The characteristics of the model `m` can be directly examined and displayed by commands like `impulse`, `step`, `bode`, `nyquist`, and `pzmap`. The quality of the model is assessed by commands like `compare` and `resid`. If you have Control System Toolbox, typing `view(m)` gives access to various display functions.

To extract state-space matrices, transfer function polynomials, etc., use the commands `arxdata`, `polydata`, `tfdata`, `ssdata`, and `zpkdata`.

To compute the frequency response of the model, use the commands `idfrd` and `freqresp`.

Creating and Modifying Model Objects

If you want to define a model to use, for example, for simulating data, you need to use the model creator functions:

- `idarx`, for multivariable ARX models
- `idgrey`, for user-defined gray-box state-space models
- `idpoly`, for single-output polynomial models
- `idproc`, for simple, continuous-time process models
- `idss`, `for` state-space models

If you want to estimate a state-space model with a specific internal parameterization, you need to create an `idss` model or an `idgrey` model. See the reference pages for these functions.

Dealing with Input and Output Channels

For multivariable models, you construct submodels containing a subset of inputs and outputs by simple subreferencing. The outputs and input channels can be referenced according to

```
m(outputs,inputs)
```

Use a colon (:) to denote all channels and an empty matrix (`[]`) to denote no channels. The channels can be referenced by number or by name. For several names, you must use a cell array, such as

```
m3 = m('position',{'power','speed'})
```

or

```
m3 = m(3,[1 4])
```

Thus m3 is the model obtained from m by looking at the transfer functions from input numbers 1 and 4 (with input names 'power' and 'speed') to output number 3 (with name position).

For a single-output model m,

```
m4 = m(inputs)
```

selects the corresponding input channels, and for a single-input model,

```
m5 = m(outputs)
```

selects the indicated output channels.

Subreferencing is quite useful, for example, when a plot of just some channels is desired.

The Noise Channels

The estimated models have two kinds of input channels: the measured inputs *u* and the noise inputs *e*. For a general linear model m, we have

$$y(t) = G(q)u(t) + H(q)e(t) \tag{12-1}$$

where u is the *nu*-dimensional vector of measured input channels and e is the *ny*-dimensional vector of noise channels. The covariance matrix of e is given by the property 'NoiseVariance'. Occasionally this matrix Λ is written in factored form,

$$\Lambda = LL^T$$

This means that e can be written as

$$e = Lv$$

where v is white noise with identity covariance matrix (independent noise sources with unit variances).

If m is a time series ($nu = 0$), G is empty and the model is given by

$$y(t) = H(q)e(t)$$

For the model m, the restriction to the transfer function matrix *G* is obtained by

```
m1 = m('measured') or just m1 = m('m')
```

Then e is set to 0 and H is removed.

Analogously,

```
m2 = m('noise') or just m2 = m('n')
```

creates a time-series model m2 from m by ignoring the measured input. That is, m2 describes the signal *He*.

For a system with measured inputs, bode, step, and other transformation and display functions deal with the transfer function matrix *G*. To obtain or graph the properties of the disturbance model *H*, it is therefore important to make the transformations m('n'). For example,

```
bode(m('n'))
```

plots the additive noise spectra according to the model m, while

```
bode(m)
```

just plots the frequency responses of *G*.

To study the noise contributions in more detail, it is useful to convert the noise channels to measured channels, using the command noisecnv.

```
m3 = noisecnv(m)
```

This creates a model m3 with all input channels, both measured u and noise sources e, treated as measured signals,. That is, m3 is a model from u and e to y, describing the transfer functions *G* and *H*. The information about the variance of the innovations e is lost. For example, studying the step response from the noise channels does not take into consideration how large the noise contributions actually are.

To include that information, e should first be normalized, $e = Lv$, so that v becomes white noise with an identity covariance matrix.

```
m4 = noisecnv(m,'Norm')
```

This creates a model m4 with u and v treated as measured signals.

$$y(t) = G(q)u(t) + H(q)Lv(t) = \begin{bmatrix} G & HL \end{bmatrix} \begin{bmatrix} u \\ v \end{bmatrix}$$

For example, the step responses from *v* to *y* will now reflect the typical size of the disturbance influence because of the scaling by *L*. In both cases, the previous noise sources that have become regular inputs will automatically get input names that are related to the corresponding output. The unnormalized noise sources e have names like 'e@ynam1' (noise e at output channel ynam1), while the normalized sources v are called 'v@ynam1'.

Retrieving Transfer Functions

The functions that retrieve transfer function properties, ssdata, tfdata, and zpkdata, will thus work as follows for a model (Equation 12-1) with measured inputs. (fcn is ssdata, tfdata, or zpkdata.)

fcn(m) returns the properties of *G* (*ny* outputs and *nu* inputs).

fcn(m('n')) returns the properties of the transfer function *H* (*ny* outputs and *ny* inputs).

fcn(noisec nv(m,'Norm')) returns the properties of the transfer function [*G HL*} (*ny* outputs and *ny*+*nu* inputs). Analogously,

```
m1 = m('n'). fcn(noisecnv(m1,'Norm'))
```

returns the properties of the transfer function *HL* (*ny* outputs and *ny* inputs).

If m is a time-series model, fcn(m) returns the properties of *H*, while

```
fcn(noisecnv(m,'Norm'))
```

returns the properties of *HL*.

Note that the estimated covariance matrix NoiseVariance itself is uncertain. This means that the uncertainty information about *H* is different from that of *HL*.

idmodel Properties

In the list below, ny is the number of output channels, and nu is the number of input channels:

- Name: An optional name for the data set. An arbitrary string.
- OutputName, InputName: Cell arrays of length ny-by-1 and nu-by-1 containing the names of the output and input channels. For estimated models, these are inherited from the data. If not specified, they are given default names {'y1','y2',...} and {'u1','u2',...}.
- OutputUnit, InputUnit: Cell arrays of length ny-by-1 and nu-by-1 containing the units of the output and input channels. Inherited from data for estimated models.
- TimeUnit: Unit for the sampling interval.
- Ts: Sampling interval. A nonnegative scalar. Ts = 0 denotes a continuous-time model. Note that changing just Ts will not recompute the model parameters. Use c2d and d2c for recomputing the model to other sampling intervals.
- ParameterVector: Vector of adjustable parameters in the model structure. Initial/nominal values or estimated values, depending on the status of the model. A column vector.
- PName: The names of the parameters. A cell array of the length of the parameter vector. If not specified, it will contain empty strings. See also setpname.
- CovarianceMatrix: Estimated covariance matrix of the parameter vector. For a nonestimated model this is the empty matrix. For state-space models in the 'Free' parameterization the covariance matrix is also the empty matrix, since the individual matrix elements are not identifiable then. Instead, in this case, the

covariance information is hidden (in the hidden property `'Utility'`) and retrieved by the relevant functions when necessary. Setting `CovarianceMatrix` to `'None'` inhibits calculation of covariance and uncertainty information. This can save substantial time for certain models.

- `NoiseVariance`: Covariance matrix of the noise source `e`. An `ny`-by-`ny` matrix.
- `InputDelay`: Vector of size `nu`-by-1, containing the input delay from each input channel. For a continuous-time model (`Ts = 0`) the delay is measured in `TimeUnit`, while for discrete-time models (`Ts > 0`) the delay is measured as the number of samples. Note the difference between `InputDelay` and `nk` (which is a property of `idarx`, `idss`, and `idpoly`). `'Nk'` is a model structure property that tells the model structure to include such an input delay. In that case, the corresponding state-space matrices and polynomials will explicitly contain `Nk` input delays. The property `InputDelay`, on the other hand, is an indication that in addition to the model as defined, the inputs should be shifted by the given amount. `InputDelay` is used by `sim` and the estimation routines to shift the input data. When computing frequency responses, the `InputDelay` is also respected. Note that `InputDelay` can be both positive and negative.
- `Algorithm`: See the reference page for `Algorithm Properties`.
- `EstimationInfo`: See the reference page for `EstimationInfo`.
- `Notes`: An arbitrary field to store extra information and notes about the object.
- `UserData`: An arbitrary field for any possible use.

Note All properties can be set or retrieved either by these commands or by subscripts. Autofill applies to all properties and values, and is case insensitive.

For a complete list of property values, use `get(m)`. To see possible value assignments, use `set(m)`.

Subreferencing

The outputs and input channels can be referenced according to

```
m(outputs,inputs)
```

Use a colon (:) to denote all channels and an empty matrix ([ ]) to denote no channels. The channels can be referenced by number or by name. For several names, you must use a cell array.

```
m2 = m('y3',{'u1','u4'})
m3 = m(3,[1 4])
```

For a single output model `m`,

```
m4 = m(inputs)
```

selects the corresponding input channels, and for a single input model,

```
m5 = m(outputs)
```

selects the indicated output channels.

The string `'measured'` (or any abbreviation like `'m'`) means the measured input channels.

```
m4 = m(3,'m')
m('m') is the same as m(:,'m')
```

Similarly, the string `'noise'` (or any abbreviation) refers to the noise input channels. See “The Noise Channels” on page 12-130 for more details.

Horizontal Concatenation

Adding input channels,

```
m = [m1,m2,...,mN]
```

creates an `idmodel` object `m`, consisting of all the input channels in `m1,...  mN`. The output channels of `mk` must be the same.

Vertical Concatenation

Adding output channels,

```
m = [m1;m2;... ;mN]
```

creates an `idmodel` object `m` consisting of all the output channels in `m1, m2, ..mN`. The input channels of `mk` must all be the same.

Online Help Functions

Type `idhelp idmodel`, `idprops idmodel`, `Methods(idmodel)`, `idprops idmodel algorithm`.

See Also

`Algorithm Properties`

`EstimationInfo`

`compare`

`idarx`

`idgrey`

`idpoly`

`idproc`

`idss`

`noisecnv`

Purpose

Class for storing nonlinear ARX models

Syntax

```
m=idnlarx([na nb nk])
m=idnlarx([na nb nk],Nonlinearity)
m=idnlarx([na nb nk],Nonlinearity,P1,V1,...,PN,VN)
```

Description

`idnlarx` is an object that stores nonlinear ARX model properties, including model parameters.

Typically, you use the `nlarx` command to both specify the nonlinear ARX model properties and estimate the model. You can specify the model properties directly in the `nlarx` syntax.

For information about the nonlinear ARX model structure, see "Definition of the Nonlinear ARX Model" on page 6-5.

The information in these reference pages summarizes the `idnlarx` model constructor and properties. It discusses the following topics:

- "idnlarx Constructor" on page 12-137
- "idnlarx Properties" on page 12-139
- "idnlarx Algorithm Properties" on page 12-141
- "idnlarx Advanced Algorithm Properties" on page 12-143
- "idnlarx EstimationInfo Properties" on page 12-145

idnlarx Constructor

Typically, you use the `nlarx` estimator command to specify the model properties and estimate the nonlinear ARX model. However, you can also use the `idnlarx` constructor to create the nonlinear ARX model structure in advance, and then estimate the parameters of this structure using `pem`.

`m=idnlarx([na nb nk])` creates an `idnlarx` object with the specified number of output terms `na`, input terms `nb`, and input delays `nk`.

`m=idnlarx([na nb nk],Nonlinearity)` creates an `idnlarx` object with the specified nonlinearity structure.

m=idnlarx([na nb nk],Nonlinearity,P1,V1,...,PN,VN) creates an idnlarx object and specifies idnlarx property-value pairs. For more information about idnlarx properties, see "idnlarx Properties" on page 12-139.

The constructor arguments have the following specifications:

[na nb nk]

na is the number of output terms, nb is the number of input terms, and nk is the input delays from each input to output.

For ny outputs and nu inputs, [na nb nk] has as many rows as there are outputs. In this case, na is an ny-by-ny matrix whose *i*-*j*th entry gives the number of delayed *j*th outputs used to compute the *i*th output. nb and nk are ny-by-nu matrices.

These orders specify the regressors and the predicted output is the following function of these regressors:

$$F\left(y(t-1),\ldots,y(t-na),u(t-nk),\ldots,u(t-nk-nb+1)\right)$$

Nonlinearity

Specifies the nonlinearity estimator object as one of the following: sigmoidnet (default), wavenet, treepartition, customnet, neuralnet, and linear. The nonlinearity estimator objects have properties that you can set in the constructor, as follows:

```
m=idnlarx([2 3 1],sigmoidnet('Num',15))
```

For ny outputs, Nonlinearity is an ny-by-1 array, such as [sigmoidnet;wavenet]. However, if you specify a scalar object, this nonlinearity object applies to all outputs.

To use default nonlinearity properties, specify the nonlinearity object name as a string. For example:

```
m=idnlarx([3 2 1],'sigmoidnet')
m=idnlarx([3 2 1],'sig') % Abbreviated
```

For more information about nonlinearity properties, see the corresponding reference pages.

idnlarx Properties

You can include property-value pairs in the model estimator or constructor to specify the model structure and estimation algorithm properties.

After creating the object, you can use `get` or dot notation to access the object property values. For example:

```
% Get the model time unit
get(m,'TimeUnit')
% Get value of Nonlinearity property
m.Nonlinearity
```

You can also use `set` or dot notation to assign property values to the object.

For example, the following two commands are equivalent:

```
set(m,'Nonlinearity','sigmoidnet')
m.Nonlinearity='sigmoidnet'
```

The following table summarizes `idnlarx` model properties. The general `idnlmodel` properties also apply to this nonlinear model object (see the corresponding reference pages).

Property Name	Description
`Algorithm`	A structure that specifies the estimation algorithm options, as described in "idnlarx Algorithm Properties" on page 12-141.
`CustomRegressors`	Custom expression in terms of standard regressors. Assignable values: • Cell array of strings. For example: `{'y1(t-3)^3','y2(t-1)*u1(t-3)','sin(u3(t-2))'}`. • Object array of `customreg` objects. For more information about this object, see the corresponding reference pages.

Property Name	Description
`EstimationInfo`	A read-only structure that stores estimation settings and results, as described in "idnlarx EstimationInfo Properties" on page 12-145.
`Focus`	Specifies `'Prediction'` or `'Simulation'`. Assignable values: • `'Prediction'` — The estimation algorithm minimizes $\|y-\hat{y}\|$, where $\hat{y}$ is the predicted output. • `'Simulation'` — The estimation algorithm minimizes the output error fit. That is, when computing $\hat{y}$, y in the regressors in F are replaced by values simulated from the input only. **Note** You cannot use `'Simulation'` when the model contains custom regressors with past outputs.
`NonlinearRegressors`	Specifies which standard or custom regressors enter the nonlinear block. Assignable values: • `'all'` — All regressors enter the nonlinear block. • `'input'` — Input regressors only. • `'output'` — Output regressors only. • `'standard'` — Standard regressors only. • `'custom'` — Custom regressors only. • `'search'` — Specifies that the estimation algorithm perform an exhaustive search of the best regressor combination. • `'[]'` — No regressors enter the nonlinear block.

Property Name	Description
`Nonlinearity`	Nonlinearity estimator object. Assignable values include `sigmoidnet` (default), `wavenet`, `treepartition`, `customnet`, `neuralnet`, and `linear`. For `ny` outputs, `Nonlinearity` is an `ny`-by-1 array, such as `[sigmoidnet;wavenet]`. However, if you specify a scalar object, this nonlinearity object applies to all outputs.

`na`
`nb`
`nk`
	Model orders and input delays, where `na` is the number of output terms, `nb` is the number of input terms, and `nk` is the delay from input to output in terms of the number of samples.
	
	For `ny` outputs and `nu` inputs, `na` is an `ny`-by-`ny` matrix whose *i*-*j*th entry gives the number of delayed *j*th outputs used to compute the *i*th output. `nb` and `nk` are `ny`-by-`nu` matrices.

idnlarx Algorithm Properties

The following table summarizes the fields of the `Algorithm` `idnlarx` model properties. `Algorithm` is a structure that specifies the estimation-algorithm options.

Property Name	Description
`Advanced`	A structure that specifies additional estimation algorithm options, as described in “idnlarx Advanced Algorithm Properties” on page 12-143.
`IterWavenet`	(For `wavenet` nonlinear estimator only) Toggles performing iterative or noniterative estimation. Default: `'auto'`. Assignable values: • `'auto'` — First estimation is noniterative and subsequent estimation are iterative. • `'On'` — Perform iterative estimation only. • `'Off'` — Perform noniterative estimation only.

Property Name	Description
`LimitError`	Robustification criterion that limits the influence of large residuals, specified as a positive real value. Residual values that are larger than `'LimitError'` times the estimated residual standard deviation have a linear cost instead of the usual quadratic cost. Default: `0` (no robustification).
`MaxIter`	Maximum number of iterations for the estimation algorithm, specified as a positive integer. Default: `20`.
`MaxSize`	The number of elements (size) of the largest matrix to be formed by the algorithm. Computational loops are used for larger matrices. Use this value for memory/speed trade-off. Default: `25000`. Assignable values: • `'Auto'` — Matrix size is determined from the m-file `idmsize`. • Any positive integer. **Note** The original data matrix of u and y must be smaller than `MaxSize`.

Property Name	Description
`SearchMethod`	Method used by the iterative search algorithm. Assignable values: • `'Auto'` — Automatically chooses from the following methods. • `'gn'` — Gauss-Newton method. • `'gna'` — Adaptive Gauss-Newton method. • `'grad'` — A gradient method. • `'lm'` — Levenberg-Marquardt method. • `'lsqnonlin'` — Nonlinear least-squares method (requires Optimization Toolbox).
`Tolerance`	Specifies to terminate the iterative search when the expected improvement of the parameter values is less than `Tolerance`, specified as a positive real value in %. Default: `0.01`.
`Trace`	Toggles displaying or hiding estimation progress information in the MATLAB Command Window. Default: `'Off'`. Assignable values: • `'Off'` — Hide estimation information. • `'On'` — Display estimation information.

idnlarx Advanced Algorithm Properties

The following table summarizes the fields of the `Algorithm.Advanced` model properties. The fields in the `Algorithm.Advanced` structure specify additional estimation-algorithm options.

Property Name	Description
`GnPinvConst`	When the search direction is computed, the algorithm discards the singular values of the Jacobian that are smaller than `GnPinvConst*max(size(J))*norm(J)*eps`. Singular values that are closer to 0 are included when `GnPinvConst` is decreased. Default: `1e4`. Assign a positive, real value.
`LMStartValue`	(For Levenberg-Marquardt search algorithm) The starting level of *regularization* when using the Levenberg-Marquardt search method (`Algorithm.SearchMethod='lm'`). Default: `0.001`. Assign a positive real value.
`LMStep`	(For Levenberg-Marquardt search algorithm) Try this next level of *regularization* to get a lower value of the criterion function. The level of regularization is `LMStep` times the previous level. At the start of a new iteration, the level of regularization is computed as`1/LMStep` times the previous level. Default: `10`. Assign a real value `>1`.
`MaxBisections`	Maximum number of bisections performed by the line search algorithm along the search direction. Used by `'gn'`, `'gna'` and `'grad'` search methods (`Algorithm.SearchMethod` property) Default: `10`. Assign a positive integer value.
`MaxFunEvals`	The iterations are stopped if the number of calls to the model file exceeds this value. Default: `Inf`. Assign a positive integer value.
`MinParChange`	The smallest parameter update allowed per iteration. Default: `1e-16`. Assign a positive, real value.

Property Name	Description
RelImprovement	The iterations are stopped if the relative improvement of the criterion function is less than RelImprovement. Default: 0. Assign a positive real value. **Note** Does not apply to Algorithm.SearchMethod='lsqnonlin'
StepReduction	(For line search algorithm) The suggested parameter update is reduced by the factor 'StepReduction' after each try until either 'MaxBisections' tries are completed or a lower value of the criterion function is obtained. Default: 2. Assign a positive, real value >1.

idnlarx EstimationInfo Properties

The following table summarizes the fields of the EstimationInfo model properties. The read-only fields of the EstimationInfo structure store estimation settings and results.

Property Name	Description
Status	Shows whether the model parameters were estimated.
Method	Shows the estimation method.
LossFcn	Value of the loss function, equal to det(E'*E/N), where E is the residual error matrix (one column for each output) and N is the total number of samples.
FPE	Value of Akaike's Final Prediction Error (see fpe).
DataName	Name of the data from which the model is estimated.
DataLength	Length of the estimation data.
DataTs	Sampling interval of the estimation data.

Property Name	Description
`DataDomain`	`'Time'` means time domain data. `'Frequency'` is not supported.
`DataInterSample`	Intersample behavior of the input estimation data used for interpolation: • `'zoh'` means zero-order-hold, or piecewise constant. • `'foh'` means first-order-hold, or piecewise linear.
`WhyStop`	Reason for terminating parameter estimation iterations.
`UpdateNorm`	Norm of the Gauss-Newton in the last iteration. Empty when `'lsqnonlin'` is the search method.
`LastImprovement`	Criterion improvement in the last iteration, shown in %. Empty when `'lsqnonlin'` is the search method.
`Iterations`	Number of iterations performed by the estimation algorithm.
`Warning`	Any warnings encountered during parameter estimation.
`InitRandState`	The value of `randn('state')` at the last randomization of the initial parameter vector.
`EstimationTime`	Duration of the estimation.

See Also

`nlarx`

`pem`

Purpose

Class for storing Hammerstein-Wiener input-output models

Syntax

```
m=idnlhw([nb nf nk])
m=idnlhw([nb nf nk],InputNL,OutputNL)
m=idnlhw([nb nf nk],InputNL,OutputNL,P1,V1,...,PN,VN)
```

Description

`idnlhw` is an object that stores Hammerstein-Wiener model properties, including model parameters.

Typically, you use the `nlhw` command to both specify the Hammerstein-Wiener model properties and estimate the model. You can specify the model properties directly in the `nlhw` syntax.

For information about the Hammerstein-Wiener model structure, see "Definition of the Hammerstein-Wiener Model" on page 6-35.

The information in these reference pages summarizes the `idnlhw` model constructor and properties. It discusses the following topics:

- "idnlhw Constructor" on page 12-147
- "idnlhw Properties" on page 12-148
- "idnlhw Algorithm Properties" on page 12-150
- "idnlhw Advanced Algorithm Properties" on page 12-152
- "idnlhw EstimationInfo Properties" on page 12-154

idnlhw Constructor

Typically, you use the `nlhw` estimator command to specify the model properties and estimate the Hammerstein-Wiener model. However, you can also use the `idnlhw` constructor to create the Hammerstein-Wiener model structure in advance, and then estimate the parameters of this structure using `pem`.

`m=idnlhw([nb nf nk])` creates an `idnlhw` object with the specified orders `nb`, `nf`, and input delays `nk`.

`m=idnlhw([nb nf nk],InputNL,OutputNL)` creates an `idnlhw` object with the specified input and output nonlinearity estimator.

`m=idnlhw([nb nf nk],InputNL,OutputNL,P1,V1,...,PN,VN)` creates an `idnlhw` object and specifies `idnlhw` property-value pairs. For more information about `idnlhw` properties, see “idnlarx Properties” on page 12-139.

The constructor arguments have the following specifications:

`[nb nf nk]`
: Model orders and input delays, where `nb` is the number of zeros plus 1, `nf` is the number of poles, and `nk` is the delay from input to output in terms of the number of samples.

  For `nu` inputs and `ny` outputs, `nb`, `nf` and, `nk` are `ny`-by-`nu` matrices whose *i*-*j*th entry specifies the orders and delay of the transfer function from the *j*th input to the *i*th output.

`InputNL` and `OutputNL`
: Specify the input and output nonlinearity estimator objects as one of the following: `pwlinear` (default), `deadzone`, `wavenet`, `saturation`, `customnet`, `sigmoidnet`, and `unitgain`. The nonlinearity estimator objects have properties that you can set in the constructor, as follows:

```
m=idnlhw([2 2 1],sigmoidnet('num',5),deadzone([-1,2]))
```

  To use default nonlinearity properties, specify the nonlinearity object name as a string. For example:

```
m=idnlhw([2 2 1],'sigmoidnet','deadzone')
m=idnlhw([2 2 1],'sig','dead') % Abbreviated
```

  The estimator `unitgain` (can also be entered as `[]`) means nonlinearity. Thus, `m=idnlhw([2 2 1],'saturation',[])` gives a Hammerstein model. For more information about nonlinearity properties, see the corresponding reference pages.

idnlhw Properties

You can include property-value pairs in the model estimator or constructor to specify the model structure and estimation algorithm properties.

After creating the object, you can use `get` or dot notation to access the object property values. For example:

```
% Get the model B parameters
get(m,'b')
% Get value of Nonlinearity property
m.b
```

You can also use `set` or dot notation to assign property values to the object.

For example, the following two commands are equivalent:

```
set(m,'InputNonlinearity','sigmoidnet')
m.InputNonlinearity='sigmoidnet'
```

The following table summarizes `idnlhw` model properties. The general `idnlmodel` properties also apply to this nonlinear model object (see the corresponding reference pages).

Property Name	Description
`Algorithm`	A structure that specifies the estimation algorithm options, as described in "idnlhw Algorithm Properties" on page 12-150.
`b`	*B* polynomial as a cell array. `b{1}` is a vector with as many leading zeros as there are input delays.
`f`	*F* polynomial as a cell array. `f{1}` is a vector.
`LinearModel`	(Read only) The linear model is an Output-Error (OE). For single output, represented as an `idpoly` object. For muliple output, represented as an `idss` object.
`EstimationInfo`	A read-only structure that stores estimation settings and results, as described in "idnlhw EstimationInfo Properties" on page 12-154.

Property Name	Description
`InputNonlinearity`	Nonlinearity estimator object. Assignable values include `pwlinear` (default), `deadzone`, `wavenet`, `saturation`, `customnet`, `sigmoidnet`, and `unitgain`. For more information, see the corresponding reference pages. For `ny` outputs, `Nonlinearity` is an `ny`-by-1 array, such as `[sigmoidnet;wavenet]`. However, if you specify a scalar object, this nonlinearity object applies to all outputs.
`OutputNonlinearity`	Same as `InputNonlinearity`.

`nb`
`nf`
`nk`
	Model orders and input delays, where `nb` is the number of zeros plus 1, `nf` is the number of poles, and `nk` is the delay from input to output in terms of the number of samples.
	
	For `nu` inputs and `ny` outputs, `nb`, `nf` and, `nk` are `ny`-by-`nu` matrices whose *i*-*j*th entry specifies the orders and delay of the transfer function from the *j*th input to the *i*th output.

idnlhw Algorithm Properties

The following table summarizes the fields of the `Algorithm` `idnlhw` model properties. `Algorithm` is a structure that specifies the estimation-algorithm options.

Property Name	Description
`Advanced`	A structure that specifies additional estimation algorithm options, as described in "idnlhw Advanced Algorithm Properties" on page 12-152.

Property Name	Description
`IterWavenet`	(For `wavenet` nonlinear estimator only) Toggles performing iterative or noniterative estimation. Default: `'auto'`. Assignable values: • `'auto'` — First estimation is noniterative and subsequent estimation are iterative. • `'On'` — Perform iterative estimation only. • `'Off'` — Perform noniterative estimation only.
`LimitError`	Robustification criterion that limits the influence of large residuals, specified as a positive real value. Residual values that are larger than `'LimitError'` times the estimated residual standard deviation have a linear cost instead of the usual quadratic cost. Default: `0` (no robustification).
`MaxIter`	Maximum number of iterations for the estimation algorithm, specified as a positive integer. Default: `20`.
`MaxSize`	The number of elements (size) of the largest matrix to be formed by the algorithm. Computational loops are used for larger matrices. Use this value for memory/speed trade-off. Default: `25000`. Assignable values: • `'Auto'` — Matrix size is determined from the m-file `idmsize`. • Any positive integer. **Note** The original data matrix of *u* and *y* must be smaller than `MaxSize`.

Property Name	Description
`SearchMethod`	Method used by the iterative search algorithm. Assignable values: • `'Auto'` — Automatically chooses from the following methods. • `'gn'` — Gauss-Newton method. • `'gna'` — Adaptive Gauss-Newton method. • `'grad'` — A gradient method. • `'lm'` — Levenberg-Marquardt method. • `'lsqnonlin'` — Nonlinear least-squares method (requires Optimization Toolbox).
`Tolerance`	Specifies to terminate the iterative search when the expected improvement of the parameter values is less than `Tolerance`, specified as a positive real value in %. Default: `0.01`.
`Trace`	Toggles displaying or hiding estimation progress information in the MATLAB Command Window. Default: `'Off'`. Assignable values: • `'Off'` — Hide estimation information. • `'On'` — Display estimation information.

idnlhw Advanced Algorithm Properties

The following table summarizes the fields of the `Algorithm.Advanced` model properties. The fields in the `Algorithm.Advanced` structure specify additional estimation-algorithm options.

Property Name	Description
GnPinvConst	When the search direction is computed, the algorithm discards the singular values of the Jacobian that are smaller than GnPinvConst*max(size(J))*norm(J)*eps. Singular values that are closer to 0 are included when GnPinvConst is decreased. Default: 1e4. Assign a positive, real value.
LMStartValue	(For Levenberg-Marquardt search algorithm) The starting level of *regularization* when using the Levenberg-Marquardt search method (Algorithm.SearchMethod='lm'). Default: 0.001. Assign a positive real value.
LMStep	(For Levenberg-Marquardt search algorithm) Try this next level of *regularization* to get a lower value of the criterion function. The level of regularization is LMStep times the previous level. At the start of a new iteration, the level of regularization is computed as1/LMStep times the previous level. Default: 10. Assign a real value >1.
MaxBisections	Maximum number of bisections performed by the line search algorithm along the search direction. Used by 'gn', 'gna' and 'grad' search methods (Algorithm.SearchMethod property) Default: 10. Assign a positive integer value.
MaxFunEvals	The iterations are stopped if the number of calls to the model file exceeds this value. Default: Inf. Assign a positive integer value.
MinParChange	The smallest parameter update allowed per iteration. Default: 1e-16. Assign a positive, real value.

Property Name	Description
`RelImprovement`	The iterations are stopped if the relative improvement of the criterion function is less than `RelImprovement`. Default: `0`. Assign a positive real value. **Note** Does not apply to `Algorithm.SearchMethod='lsqnonlin'`
`StepReduction`	(For line search algorithm) The suggested parameter update is reduced by the factor `'StepReduction'` after each try until either `'MaxBisections'` tries are completed or a lower value of the criterion function is obtained. Default: `2`. Assign a positive, real value `>1`.

idnlhw EstimationInfo Properties

The following table summarizes the fields of the `EstimationInfo` model properties. The read-only fields of the `EstimationInfo` structure store estimation settings and results.

Property Name	Description
`Status`	Shows whether the model parameters were estimated.
`Method`	Shows the estimation method.
`LossFcn`	Value of the loss function, equal to `det(E'*E/N)`, where `E` is the residual error matrix (one column for each output) and `N` is the total number of samples.
`FPE`	Value of Akaike's Final Prediction Error (see `fpe`).
`DataName`	Name of the data from which the model is estimated.

Property Name	Description
`DataLength`	Length of the estimation data.
`DataTs`	Sampling interval of the estimation data.
`DataDomain`	`'Time'` means time domain data. `'Frequency'` is not supported.
`DataInterSample`	Intersample behavior of the input estimation data used for interpolation: • `'zoh'` means zero-order-hold, or piecewise constant. • `'foh'` means first-order-hold, or piecewise linear.
`WhyStop`	Reason for terminating parameter estimation iterations.
`UpdateNorm`	Norm of the search vector (gn-vector) in the last iteration. Empty when `'lsqnonlin'` is the search method.
`LastImprovement`	Criterion improvement in the last iteration, shown in %. Empty when `'lsqnonlin'` is the search method.
`Iterations`	Number of iterations performed by the estimation algorithm.
`Warning`	Any warnings encountered during parameter estimation.
`InitRandState`	The value of `randn('state')` at the last randomization of the initial parameter vector.
`EstimationTime`	Duration of the estimation.

See Also

`nlhw`

`pem`

idnlgrey

Purpose

Class for storing nonlinear grey-box models

Syntax

```
m = idnlgrey('filename',Order,Parameters)
m = idnlgrey('filename',Order,Parameters,InitialStates)
m = idnlgrey('filename',Order,Parameters,InitialStates,Ts)
```

Description

`idnlgrey` is an object that stores the nonlinear grey-box model structure.

For information about the nonlinear grey-box model structure, see "Specifying the Nonlinear Grey-Box Model Structure" on page 7-13.

The information in these reference pages summarizes the `idnlgrey` model constructor and properties. It discusses the following topics:

- "idnlgrey Constructor" on page 12-156
- "idnlgrey Properties" on page 12-157
- "idnlgrey Advanced Algorithm Properties" on page 12-161
- "idnlgrey Simulation Options" on page 12-162
- "idnlgrey Gradient Options" on page 12-166
- "idnlgrey EstimationInfo Properties" on page 12-167

idnlgrey Constructor

Use the following syntax to define the `idnlgrey` model object:

```
m = idnlgrey('filename',Order,Parameters)
```

```
m = idnlgrey('filename',Order,Parameters,InitialStates)
```

```
m = idnlgrey('filename',Order,Parameters,InitialStates,Ts)
```

The `idnlgrey` arguments are defined as follows:

- `'`*`filename`*`'` — Name of the m-file or MEX-file storing the model structure. This file must be on the MATLAB path.

- 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 a same way as parameters. Must be fourth input to the idnlgrey constructor.

Estimate the parameters of this structure using pem.

idnlgrey Properties

You can include property-value pairs in the model estimator or constructor to specify the model structure and estimation algorithm properties.

After creating the object, you can use get or dot notation to access the object property values. For example:

```
% Get the model time unit
get(m,'TimeUnit')
m.TimeUnit
```

You can also use set or dot notation to assign property values to the object.

For example, the following two commands are equivalent:

```
set(m,'TimeUnit','sec')
m.TimeUnit='sec'
```

The following table summarizes idnlgrey model properties. The general idnlmodel properties also apply to this nonlinear model object (see the corresponding reference pages).

Property Name	Description
`Algorithm`	A structure that specifies the estimation algorithm options, as described in "idnlgrey Algorithm Properties" on page 12-159.
`CovarianceMatrix`	Covariance matrix of the estimated `Parameters`. Symmetric and positive `Np`-by-`Np` matrix (or `[]`), where `Np` is the number of free model parameters. Assignable values: • `'None'` to omit computing uncertainties and save time during parameter estimation. • `'Estimate'` to estimation covariance.
`EstimationInfo`	A read-only structure that stores estimation settings and results, as described in "idnlgrey EstimationInfo Properties" on page 12-167.
`FileArgument`	Contains auxiliary variables passed to the m-file or MEX-file. There variables might be required for updating the constants in the state equations. `FileArgument` data is a cell array. Default: `{}`.
`FileName`	File name string (without extension) or a function handle for computing the states and the outputs. If 'FileName' is a string, then it must point to an m-file or MEX-file. For more information about the file variables, see "Specifying the Nonlinear Grey-Box Model Structure" on page 7-13.
`InitialStates`	Specified in a same way as parameters. Must be fourth input to the `idnlgrey` constructor.

Property Name	Description
Order	Structure with following fields: • ny — Number of outputs of the model structure. • nu — Number of inputs of the model structure. • nx — Number of states of the model structure. For time-series, nu is 0.
Parameters	Np-by-1 structure array with information about the model parameters. Parameters can be real scalars, column vectors, or two-dimensional matrices. Np is the number of parameter object. For scalar parameters, Np is the total number of parameter elements.

idnlgrey Algorithm Properties

The following table summarizes the fields of the Algorithm idnlhw model properties. Algorithm is a structure that specifies the estimation-algorithm options.

Property Name	Description
Advanced	A structure that specifies additional estimation algorithm options, as described in "idnlgrey Advanced Algorithm Properties" on page 12-161.
LimitError	Robustification criterion that limits the influence of large residuals, specified as a positive real value. Residual values that are larger than 'LimitError' times the estimated residual standard deviation have a linear cost instead of the usual quadratic cost. Default: 1.6.
MaxIter	Maximum number of iterations for the estimation algorithm, specified as a positive integer. Default: 20.

Property Name	Description
`SearchMethod`	Method used by the iterative search algorithm. Assignable values: • `'Auto'` — Automatically chooses from the following methods. • `'gn'` — Gauss-Newton method. • `'gna'` — Adaptive Gauss-Newton method. • `'grad'` — A gradient method. • `'lm'` — Levenberg-Marquardt method. • `'lsqnonlin'` — Nonlinear least-squares method (requires Optimization Toolbox).
`Tolerance`	Specifies to terminate the iterative search when the expected improvement of the parameter values is less than `Tolerance`, specified as a positive real value in %. Default: `0.01`.
`GradientOptions`	A structure that specifies the options related to calculation of gradient of the cost, "idnlgrey Gradient Options" on page 12-166.
`SimulationOptions`	A struct that specifies the simulation method and related options, as described in "idnlgrey Simulation Options" on page 12-162.
`Trace`	Toggles displaying or hiding estimation progress information in the MATLAB Command Window. Default: `'Off'`. Assignable values: • `'Off'` — Hide estimation information. • `'On'` — Display estimation information.

idnlgrey Advanced Algorithm Properties

The following table summarizes the fields of the `Algorithm.Advanced` model properties. The fields in the `Algorithm.Advanced` structure specify additional estimation-algorithm options.

Property Name	Description
`GnPinvConst`	When the search direction is computed, the algorithm discards the singular values of the Jacobian that are smaller than `GnPinvConst*max(size(J))*norm(J)*eps`. Singular values that are closer to 0 are included when `GnPinvConst` is decreased. Default: `1e4`. Assign a positive, real value.
`LMStartValue`	(For Levenberg-Marquardt search algorithm) The starting level of *regularization* when using the Levenberg-Marquardt search method (`Algorithm.SearchMethod='lm'`). Default: `0.001`. Assign a positive real value.
`LMStep`	(For Levenberg-Marquardt search algorithm) Try this next level of *regularization* to get a lower value of the criterion function. The level of regularization is `LMStep` times the previous level. At the start of a new iteration, the level of regularization is computed as`1/LMStep` times the previous level. Default: `10`. Assign a real value `>1`.
`MaxBisections`	Maximum number of bisections performed by the line search algorithm along the search direction. Used by `'gn'`, `'gna'` and `'grad'` search methods (`Algorithm.SearchMethod` property) Default: `10`. Assign a positive integer value.

Property Name	Description
`MaxFunEvals`	The iterations are stopped if the number of calls to the model file exceeds this value. Default: `Inf`. Assign a positive integer value.
`MinParChange`	The smallest parameter update allowed per iteration. Default: `1e-16`. Assign a positive, real value.
`RelImprovement`	The iterations are stopped if the relative improvement of the criterion function is less than `RelImprovement`. Default: `0`. Assign a positive real value. **Note** Does not apply to `Algorithm.SearchMethod='lsqnonlin'`
`StepReduction`	(For line search algorithm) The suggested parameter update is reduced by the factor `'StepReduction'` after each try until either `'MaxBisections'` tries are completed or a lower value of the criterion function is obtained. Default: `2`. Assign a positive, real value `>1`.

idnlgrey Simulation Options

The following table summarizes the fields of `Algorithm.SimulationOptions` model properties.

Property Name	Description
AbsTol	(For variable-step time-continuous solvers) Specifies the smallest time step the ODE solver. Default: 1e-6. Assignable value: A positive real value.
FixedStep	(For fixed-step time-continuous solvers) Step size used by the solver. Default: 'Auto'. Assignable values: • 'Auto' — Automatically chooses the initial step. • A real value such that 0<FixedStep<=1.
InitialStep	(For variable-step time-continuous solvers) Specifies the initial step at which the ODE solver starts. Default: 'Auto'. Assignable values: • 'Auto' — Automatically chooses the initial step. • A positive real value such that MinStep<=InitialStep<=MaxStep.
MaxOrder	(For ode15s) Specifies the order of the Numerical Differentiation Formulas (NDF). Default: 5. Assignable values: 1, 2, 3, 4 or 5.
MaxStep	(For variable-step time-continuous solvers) Specifies the largest time step of the ODE solver. Default: 'Auto' — 1/15 of the simulation interval. Assignable values: • 'Auto' — Automatically chooses the time step. • A positive real value > MinStep.

Property Name	Description
`MinStep`	(For variable-step time-continuous solvers) Specifies the smallest time step of the ODE solver. Default: `'Auto'`. Assignable values: • `'Auto'` — Automatically chooses the time step. • A positive real value < `MaxStep`.

Property Name	Description
`RelTol`	(For variable-step time-continuous solvers) Relative error tolerance that applies to all components of the state vector. The estimated error in each integration step satisfies `|e(i)| <= max(RelTol*abs(x(i)), AbsTol(i))`. Default: `1e-3` (0.1% accuracy). Assignable value: A positive real value.
`Solver`	ODE (Ordinary Differential/Difference Equation) solver for solving state space equations. A. Variable-step solvers for time-continuous `idnlgrey` models: • `'ode45'` — Runge-Kutta (4,5) solver for nonstiff problems. • `'ode23'` — Runge-Kutta (2,3) solver for nonstiff problems. • `'ode113'` — Adams-Bashforth-Moulton solver for nonstiff problems. • `'ode15s'` — Numerical Differential Formula solver for stiff problems. • `'ode23s'` — Modified Rosenbrock solver for stiff problems. • `'ode23t'` — Trapezoidal solver for moderately stiff problems. • `'ode23tb'` — Implicit Runge-Kutta solver for stiff problems. B. Fixed-step solvers for time-continuous `idnlgrey` models: • `'ode5'` — Dormand-Prince solver. • `'ode4'` — Fourth-order Runge-Kutta solver. • `'ode3'` — Bogacki-Shampine solver. • `'ode2'` — Heun or improved Euler solver. • `'ode1'` — Euler solver. C. Fixed-step solvers for time-discrete `idnlgrey` models: `'FixedStepDiscrete'` D. General: `'Auto'` — Automatically chooses one of the previous solvers (default).

idnlgrey Gradient Options

The following table summarizes the fields of the `Algorithm` `idnlhw` model properties. `Algorithm` is a structure that specifies the estimation-algorithm options.

Property Name	Description
`DiffMaxChange`	Largest allowed parameter perturbation when computing numerical derivatives. Default: `Inf`. Assignable value: A positive real value >`'DiffMinChange'`.
`DiffMinChange`	Smallest allowed parameter perturbation when computing numerical derivatives. Default: `0.01*sqrt(eps)`. Assignable value: A positive real value <`'DiffMaxChange'`.

Property Name	Description
`DiffScheme`	Method for computing numerical derivatives with respect to the components of the parameters and/or the initial state(s) to form the Jacobian. Default: `'Auto'` Assignable values: • `'Auto'` - Automatically chooses from the following methods. • `'Central approximation'` • `'Forward approximation'` • `'Backward approximation'`
`GradientType`	Method used when computing derivatives (Jacobian) of the parameters or the initial states to be estimated. Default: `'Auto'`. Assignable values: • `'Auto'` — Automatically chooses from the following methods. • `'Basic'` — Individually computes all numerical derivatives required to form each column of the Jacobian. • `'Refined'` — Simultaneously computes all numerical derivatives required to form each column of the Jacobian.

idnlgrey EstimationInfo Properties

The following table summarizes the fields of the `EstimationInfo` model properties. The read-only fields of the `EstimationInfo` structure store estimation settings and results.

Property Name	Description
`Status`	Shows whether the model parameters were estimated.
`Method`	Names of the solver and the optimizer used during estimation.
`LossFcn`	Value of the loss function, equal to `det(E'*E/N)`, where `E` is the residual error matrix (one column for each output) and `N` is the total number of samples. Provides a quantitative description of the model quality.
`FPE`	Value of Akaike's Final Prediction Error (see `fpe`).
`DataName`	Name of the data from which the model is estimated.
`DataLength`	Length of the estimation data.
`DataTs`	Sampling interval of the estimation data.
`DataDomain`	`'Time'` means time domain data. `'Frequency'` is not supported.
`DataInterSample`	Intersample behavior of the input estimation data used for interpolation: • `'zoh'` means zero-order-hold, or piecewise constant. • `'foh'` means first-order-hold, or piecewise linear.
`WhyStop`	Reason for terminating parameter estimation iterations.
`UpdateNorm`	Norm of the search vector (Gauss-Newton vector) at the last iteration. Empty when `'lsqnonlin'` is the search method.
`LastImprovement`	Criterion improvement in the last iteration, shown in %. Empty when `SearchMethod='lsqnonlin'` is the search method.
`Iterations`	Number of iterations performed by the estimation algorithm.

Property Name	Description
`InitialGuess`	Structure with the fields `InitialStates` and `Parameters`, specifying the values of these quantities before the last estimation.
`Warning`	Any warnings encountered during parameter estimation.

See Also

`pem`

Purpose

Superclass for nonlinear models

Description

You do not use the `idnlmodel` class directly. Instead, `idnlmodel` defines the common properties and methods inherited by its subclasses, `idnlarx`, `idnlgrey`, and `idnlhw`.

idnlmodel Properties

The following table lists the properties shared by the `idnlarx`, `idnlgrey`, and `idnlhw`, defined in terms of `Ny` outputs and `Nu` inputs.

Property Name	Description
`InputName`	Specifies the names of individual input channels. Default: `{'u1';'u2';...;'uNu'}`. Assignable values: • For single-output models, a string. For example, `'torque'`. • For multiple-output models, an `nu-by-1` cell array. For example: `{'thrust'; 'aileron deflection'}`
`InputUnit`	Specifies the units of each input channel. Default: `''`. Assignable values: • For single-output models, a string. For example, `'m/s'`. • For multiple-output models, an `nu-by-1` cell array.
`Name`	Name of the model, specified as a string.
`NoiseVariance`	Noise variance (covariance matrix) of the model innovations *e*. Assignable value is an `Ny-by-Ny` matrix. Typically set automatically by the estimation algorithm.

Property Name	Description
`OutputName`	Specifies the names of individual output channels. Default: `{'y1';'y2';...;'yNy'}`. Assignable values: • For single-output models, a string. For example, `'torque'`. • For multiple-output models, an `nu-by-1` cell array. For example: `{'thrust'; 'aileron deflection'}`
`OutputUnit`	Specifies the units of each output channel. Default: `''`. Assignable values: • For single-output models, a string. For example, `'m/s'`. • For multiple-output models, an `nu-by-1` cell array.
`TimeUnit`	Unit of the sampling interval and time vector, specified as a string. Default: `''`.
`TimeVariable`	Independent variable for the inputs, outputs, and—when available—internal states, specified as a string. Default: `'t'` (time).
`Ts`	Sampling interval with the unit specified by `TimeUnit`. Default: `1`. Assignable values: • For discrete-time models, positive scalar value of the sampling interval. • For continuous-time models, `0`.

See Also

`idnlarx`

`idnlgrey`

`idnlhw`

Purpose

Class for storing linear polynomial input-output models

Syntax

```
m = idpoly(A,B)
m = idpoly(A,B,C,D,F,NoiseVariance,Ts)
m = idpoly(A,B,C,D,F,NoiseVariance,Ts,'Property1',Value1,...
      'PropertyN',ValueN)

m = idpoly(mi)
```

Description

`idpoly` creates a model object containing parameters that describe the general multiinput single-output model structure.

$$A(q)y(t) = \frac{B_1(q)}{F_1(q)}u_1(t-nk_1) + \ldots + \frac{B_{nu}(q)}{F_{nu}(q)}u_{nu}(t-nk_{nu}) + \frac{C(q)}{D(q)}e(t)$$

`A`, `B`, `C`, `D`, and `F` specify the polynomial coefficients.

For single-input systems, these are all row vectors in the standard MATLAB format.

```
A = [1 a1 a2 ... ana]
```

consequently describes

$$A(q) = 1 + a_1 q^{-1} + \ldots + a_{na} q^{-na}$$

`A`, `C`, `D`, and `F` all start with 1, while `B` contains leading zeros to indicate the delays. See "Definition of Polynomial Models" on page 5-43.

For multiinput systems, `B` and `F` are matrices with one row for each input.

For time series, `B` and `F` are entered as empty matrices.

```
B = []; F = [];
```

`NoiseVariance` is the variance of the white noise sequence $e(t)$, while `Ts` is the sampling interval.

Trailing arguments `C`, `D`, `F`, `NoiseVariance`, and `Ts` can be omitted, in which case they are taken as 1. (If `B = []`, then `F` is taken as `[]`.) The property name/property value pairs can start directly after `B`.

`Ts = 0` means that the model is a continuous-time one. Then the interpretation of the arguments is that

```
A = [1 2 3 4]
```

corresponds to the polynomial $s^3 + 2s^2 + 3s + 4$ in the Laplace variable *s*, and so on. For continuous-time systems, `NoiseVariance` indicates the level of the spectral density of the innovations. A sampled version of the model has the innovations variance `NoiseVariance/Ts`, where `Ts` is the sampling interval. The continuous-time model must have a white noise component in its disturbance description. See "Spectrum Normalization and the Sampling Interval" on page 5-40.

For discrete-time models (`Ts > 0`), note the following: `idpoly` strips any trailing zeros from the polynomials when determining the orders. It also strips leading zeros from the `B` polynomial to determine the delays. Keep this in mind when you use `idpoly` and `polydata` to modify earlier estimates to serve as initial conditions for estimating new structures.

`idpoly` can also take any single-output `idmodel` or LTI object `mi` as an input argument. If an LTI system has an input group with name `'Noise'`, these inputs are interpreted as white noise with unit variance, and the noise model of the `idpoly` model is computed accordingly.

Properties

- `na`, `nb`, `nc`, `nd`, `nf`, `nk`: The orders and delays of the polynomials. Integers or row vectors of integers.
- `a`, `b`, `c`, `d`, `f`: The polynomials, described by row vectors and matrices as detailed above.
- `da`, `db`, `dc`, `dd`, `df`: The estimated standard deviation of the polynomials. Cannot be set.
- `'InitialState'`: How to deal with the initial conditions that are required to compute the prediction of the output. Possible values are

- 'Estimate': The necessary initial states are estimated from data as extra parameters.
- 'Backcast': The necessary initial states are estimated by a backcasting (backward filtering) process, described in Knudsen (1994).
- 'Zero': All initial states are taken as zero.
- 'Auto': An automatic choice among the above is made, guided by the data.

In addition, any idpoly object also has all the properties of idmodel. See idmodel properties and Algorithm Properties.

Note that you can set or retrieve all properties either with the set and get commands or by subscripts. Autofill applies to all properties and values, and these are case insensitive.

```
m.a=[1 -1.5 0.7];
set(m,'ini','b')
p = roots(m.a)
```

For a complete list of property values, use get(m). To see possible value assignments, use set(m). See also idprops idpoly.

Examples

To create a system of ARMAX, type

```
A = [1 -1.5 0.7];
B = [0 1 0.5];
C = [1 -1 0.2];
m0 = idpoly(A,B,C);
```

This gives a system with one delay (nk = 1).

Create the continuous-time model

$$y(t) = \frac{1}{s(s+1)}u_1(t) + \frac{s+3}{s^2+2s+4}u_2(t) + e(t)$$

Sample it with `T = 0.1` and then simulate it without noise.

```
B=[0 1;1 3];
F=[1 1 0;1 2 4]
m = idpoly(1,B,1,1,F,1,0)
md = c2d(m,0.1)
y = sim(md,[u1 u2])
```

Note that the continuous-time model is automatically sampled to the sampling interval of the data, when simulated, so the above is also achieved by

```
u = iddata([],[u1 u2],0.1)
y = sim(m,u)
```

References

Ljung (1999) Section 4.2 for the model structure family.

Knudsen, T., (1994), " new method for estimating ARMAX models,"In *Proc. 10th IFAC Symposium on System Identification,* pp. 611-617, Copenhagen, Denmark, for the backcast method.

See Also

`idss`

`sim`

Purpose

Class for storing low-order, continuous-time process models

Syntax

```
m = idproc(Type)
m = idproc(Type,'Property1',Value1,...,'PropertyN',ValueN)
m = pem(Data,Type) % to directly estimate an idproc model
```

Description

The function `idproc` is used to create typical simple, continuous-time process models as `idproc` objects. The model has one output, but can have several inputs.

The character of the model is defined by the argument `Type`. This is an acronym made up of the following symbols:

- P: All `'Type'` acronyms start with this letter.
- 0, 1, 2, or 3: This integer denotes the number of time constants (poles) to be modeled. Possible integrations (poles in the origin) are not included in this number.
- I: The letter I is included to mark that an integration is enforced (self-regulation process).
- D: The letter D is used to mark that the model contains a time delay (dead time).
- Z: The letter Z is used to mark an extra numerator term: a zero.
- U: The letter U is included to mark that underdamped modes (complex-valued poles) are permitted. If U is not included, all poles are restricted to be real.

This means, for example, that `Type = 'P1D'` corresponds to the model with transfer function

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

while `Type = 'P0I'` is

$$G(s) = \frac{K_p}{s}$$

and `Type = 'P3UZ'` is

$$G(s) = K_p \frac{1 + T_z s}{(1 + 2\varsigma T_w s + (T_w s)^2)(1 + T_{p3} s)}$$

For multiinput systems, `Type` is a cell array where each cell describes the character of the model from the corresponding input, like `Type = {'P1D'.'POI'}` for the two-input model

$$Y(s) = \frac{K_p(1)}{1 + sT_{p1}(1)} e^{-T_d s} U_1(s) + \frac{K_p(2)}{s} U_2(s) \qquad \textbf{(12-2)}$$

The parameters of the model are

- `Kp`: The static gain
- `Tp1`, `Tp2`, `Tp3`: The real-time constants (corresponding to poles in `1/Tp1`, etc.)
- `Tw` and `Zeta`: The "resonance time constant" and the damping factor corresponding to a denominator factor `(1+2 Zeta Tw s + (Tw s)^2)`. If underdamped modes are allowed, `Tw` and `Zeta` replace `Tp1` and `Tp2`. A third real pole, `Tp3`, could still be included.
- `Td`: The time delay
- `Tz`: The numerator zero

These properties contain fields that give the values of the parameters, upper and lower bounds, and information whether they are locked to zero, have a fixed value, or are to be estimated. For multiinput models, the number of entries in these fields equals the number of inputs. This is described in more detail below.

The `idproc` object is a child of `idmodel`. Therefore any `idmodel` properties can be set as property name/property value pairs in the `idproc` command. They can also be set by the command `set`, or by subassignment, as in

```
m.InputName = {'speed','voltage'}
m.kp = 12
```

In the multiinput case, models for specific inputs can be obtained by regular subreferencing.

```
m(ku)
```

There are also two properties, `DisturbanceModel` and `InitialState`, that can be used to expand the model. See below.

idproc Properties

- `Type`: A string or a cell array of strings with as many elements as there are inputs. The string is an acronym made up of the characters `P`, `Z`, `I`, `U`, `D` and an integer between 0 and 3. The string must start with `P`, followed by the integer, while possible other characters can follow in any order. The integer is the number of poles (not counting a possible integration), `Z` means the inclusion of a numerator zero, `D` means inclusion of a time delay, while `U` marks that the modes can be underdamped (a pair of complex conjugated poles). `I` means that an integration in the model is enforced.
- `Kp`, `Tp1`, `Tp2`, `Tp3`, `Tw`, `Zeta`, `Tz`, `Td`: These are the parameters as explained above. Each of these is a structure with the following fields:
  - `value`: Numerical value of the parameter.
  - `max`: Maximum allowed value of the parameter when it is estimated.
  - `min`: Minimum allowed value of the parameter when it is estimated. For multiinput models, these are row vectors.
  - `status`: Assumes one of `'Estimate'`, `'Fixed'`, or `'Zero'`.

    `'Zero'` means that the parameter is locked to zero and not included in the model. Assigning, for example, `Type = 'P1'` means that the status of `Tp2`, `Tp3`, `Tw`, and `Zeta` will be `'Zero'`.

The value `'Fixed'` means that the parameter is fixed to its value, and will not be estimated.

The value `'Estimate'` means that the parameter value should be estimated.

For multiinput modes, `status` is a cell array with one element for each input, while `value`, `max`, and `min` are row vectors.

- `DisturbanceModel`: Allows an additive disturbance model as in

$$y(t) = G(s)u(t) + \frac{C(s)}{D(s)}e(t) \tag{12-3}$$

  where $G(s)$ is a process model and $e(t)$ is white noise, and C/D is a first- or second-order transfer function.

  `DisturbanceModel` can assume the following values:

  - `'None'`: This is the default. No disturbance model is included (that is, *C*=*D*=*1*).
  - `'arma1'`: The disturbance model is a first-order ARMA model (that is, *C* and *D* are first-order polynomials).
  - `'arma2'` or `'Estimate'`: The disturbance model is a second-order ARMA model (that is, *C* and *D* are second-order polynomials).

  When a disturbance model has been estimated, the property `DisturbanceModel` is returned as a cell array, with the first entry being the status as just defined, and the second entry being the actual model, delivered as a continuous-time `idpoly` object.

- `InitialState`: Affects the parameterization of the initial values of the states of the model. It assumes the same values as for other models:

  - `'Zero'`: The initial states are fixed to zero.
  - `'Estimate'`: The initial states are treated as parameters to be estimated.
  - `'Backcast'`: The initial state vector is adjusted, during the parameter estimation step, to a suitable value, but it is not stored.

    - `'Auto'`: Makes a data-dependent choice among the values above.

- `InputLevel`: The offset level of the input signal(s). This is of particular importance for those input channels that contain an integration. `InputLevel` will then define the level from which the integration takes place, and that cannot be handled by estimating initial states. `InputLevel` has the same structure as the model parameters `Kp`, etc., and thus contains the following fields:
    - `value`: Numerical value of the parameter. For multiinput models, this is a row vector.
    - `max`: Maximum allowed value of the parameter when it is estimated.
    - `min`: Minimum allowed value of the parameter when it is estimated. For multiinput models, these are row vectors.
    - `status`: Assumes one of `'Estimate'`, `'Fixed'`, or `'Zero'` with the same interpretations.

In addition, any `idproc` object also has all the properties of `idmodel`. See `Algorithm Properties`, `EstimationInfo`, and `idmodel`.

Note that all properties can be set or retrieved using either the `set` and `get` commands or subscripts. Autofill applies to all properties and values, and these are case insensitive. Also `'u'` and `'y'` are short for `'Input'` and `'Output'`, respectively. You can also set all properties at estimation time as property name/property value pairs in the call to `pem`. An extended syntax allows direct setting of the fields of the parameter values, so that assigning a numerical value is automatically attributed to the value field, while a string is attributed to the status field.

```
m.kp = 10
m.tp1 = 'estimate'
% Initializing the parameter Kp at the value 10
m = pem(Data,'P1D','kp',10)
% Fixing the parameter Kp to the value 10
m = pem(Data,'P1D','kp',10,'kp','fix')
% constraining Kp to lie between 3 and 4
```

```
m = pem(Data,'P2U','kp',{'max',4},'kp',{'min',3})
% For two inputs, estimate the offset level
% of the first input
m = pem(Data,{'P2I','P1D',},'ulevel',{'est','zer'})
% estimate a noise model
m = pem(Data,'P2U','dist','est')
% Use a fixed noisemodel,
% given by the continuous-time idpoly model noimod
m = pem(Data,'P2U','dist',{'fix',noimod})
% (minimum Kp for the second input)
m.kp.min(2) = 12
% fixing the gain for the second input.
m.kp.status{2} = 'fix'
```

For a complete list of property values, use `get(m)`. To see possible value assignments, use `set(m)`. See also `idprops` and `idproc`.

Examples

```
m = pem(Data,'P2D','dist','arma1')
```

Purpose

Resample `iddata` object by decimation and interpolation

Syntax

```
datar = idresamp(data,R)
[datar,res_fact] = idresamp(data,R,order,tol)
```

Arguments

Z : The output-input data as a matrix or as an IDDATA object. ZD : The resampled data. If Z is IDDATA, so is ZD. Otherwise the columns of ZD correspond to those of Z. R : The resampling factor. The new data record ZD will correspond to a sampling interval of R times that of the original data. R > 1 thus means decimation and R < 1 means interpolation. Any positive number for R is allowed, but it will be replaced by a rational approximation.

[ZD, ACT_R] = IDRESAMP(Z,R,ORDER,TOL) gives access to the following options: ORDER determines the filter orders used at decimation and interpolation (Default 8). TOL gives the tolerance of the rational approximation (Default 0.1). ACT_R is the actually used resampling factor.

Description

```
datar = idresamp(data,R)
```

```
[datar,res_fact] = idresamp(data,R,order,tol)
```

See Also

`idfilt`

Purpose

Class for storing linear state-space models with known and unknown parameters

Syntax

```
m = idss(A,B,C,D)
m =
idss(A,B,C,D,K,x0,Ts,'Property1',Value1,...,'PropertyN',ValueN)
mss = idss(m1)
```

Description

The function `idss` is used to construct state-space model structures with various parameterizations. It is a complement to `idgrey` and deals with parameterizations that do not require the user to write a special M-file. Instead it covers parameterizations that are either `'Free'`, that is, all parameters in the A, B, and C matrices can be adjusted freely, or `'Canonical'`, meaning that the matrices are parameterized as canonical forms. The parameterization can also be `'Structured'`, which means that certain elements in the state-space matrices are free to be adjusted, while others are fixed. This is explained below.

`Ts` is the sampling interval. `Ts = 0` means a continuous-time model. The default is `Ts = 1`.

The `idss` object `m` describes state-space models in innovations form of the following kind:

$$\tilde{x}(t) = A(\theta)x(t) + B(\theta)u(t) + K(\theta)e(t)$$

$$x(0) = x_0(\theta)$$

$$y(t) = C(\theta)x(t) + D(\theta)u(t) + e(t)$$

Here $\tilde{x}(t)$ is the time derivative $\dot{x}(t)$ for a continuous-time model and $x(t+Ts)$ for a discrete-time model.

The model `m` will contain information both about the nominal/initial values of the A, B, C, D, K, and X0 matrices and about how these matrices are parameterized by the parameter vector θ (to be estimated).

The nominal model is defined by `idss(A,B,C,D,K,X0)`. If `K` and `X0` are omitted, they are taken as zero matrices of appropriate dimensions.

Defining an `idss` object from a given model,

```
mss = idss(m1)
```

constructs an `idss` model from any `idmodel` or LTI system `m1`.

If `m1` is an LTI system (`ss`, `tf`, or `zpk`) that has no `InputGroup` called `'Noise'`, the corresponding state-space matrices *A*, *B*, *C*, *D* are used to define the `idss` object. The Kalman gain *K* is then set to zero.

If the LTI system has an `InputGroup` called `'Noise'`, these inputs are interpreted as white noise with a covariance matrix equal to the identity matrix. The corresponding Kalman gain and noise variance are then computed and entered into the `idss` model together with *A*, *B*, *C*, and *D*.

Parameterizations

There are several different ways to define the parameterization of the state-space matrices. The parameterization determines which parameters can be adjusted to data by the parameter estimation routine `pem`.

- Free black-box parameterizations: This is the default situation and corresponds to letting all parameters in *A*, *B*, and *C* be freely adjustable. You do this by setting the property `'SSParameterization' = 'Free'`. The parameterizations of *D*, *K*, and *X0* are then determined by the following properties:
  - `'nk'`: A row vector of the same length as the number of inputs. The `ku`th element is the delay from input channel number `ku`. Thus `nk = [0,...,0]` means that there is no delay from any of the inputs, and that consequently all elements of the *D* matrix should be estimated. `nk =[1,...,1]` means that there is a delay of 1 from each input, so that the *D* matrix is fixed to be zero.
  - `'DisturbanceModel'`: This property affects the parameterization of K and can assume the following values:

'Estimate': All elements of the K matrix are to be estimated.

'None': All elements of K are fixed to zero.

'Fixed': All elements of K are fixed to their nominal/initial values.

- 'InitialState': Affects the parameterization of $X0$ and can assume the following values:

  'Auto': An automatic choice of the following is made, depending on data (default).

  'Estimate': All elements of $X0$ are to be estimated.

  'Zero': All elements of $X0$ are fixed to zero.

  'Fixed': All elements of $X0$ are fixed to their nominal/initial values.

  'Backcast': The vector $X0$ is adjusted, during the parameter estimation step, to a suitable value, but it is not stored as an estimation result.

- Canonical black-box parameterizations: You do this by setting the property 'SSParameterization' = 'Canonical'. The matrices A, B, and C are then parameterized as an observer canonical form, which means that ny (number of output channels) rows of A are fully parameterized while the others contain 0's and 1's in a certain pattern. The C matrix is built up of 0's and 1's while the B matrix is fully parameterized. See Equation (A.16) in Ljung (1999) for details. The exact form of the parameterization is affected by the property 'CanonicalIndices'. The default value 'Auto' is a good choice. The parameterization of the D, K, and $X0$ matrices in this case is determined by the properties 'nk', 'DisturbanceModel', and 'InitialState'.

- Arbitrarily structured parameterizations: The general case, where arbitrary elements of the state-space matrices are fixed and others can be freely adjusted, corresponds to the case 'SSParameterization' = 'Structured'. The parameterization is determined by the idss properties As, Bs, Cs, Ds, Ks, and X0s. These are the *structure matrices* that are “shadows” of the state-space matrices, so that an element in these matrices that is equal to `NaN` indicates a freely adjustable parameter, while a numerical value in these matrices indicates that the corresponding system matrix element is fixed (nonadjustable) to this value.

idss Properties

- `SSParameterization` has the following possible values:
  - `'Free'`: Means that all parameters in *A*, *B*, and *C* are freely adjustable, and the parameterizations of *D*, *K*, and *X0* depend on the properties `'nk'`, `'DisturbanceModel'`, and `'InitialState'`.
  - `'Canonical'`: Means that *A* and *C* are parameterized as an observer canonical form. The details of this parameterization depend on the property `'CanonicalIndices'`. The *B* matrix is always fully parameterized, and the parameterizations of *D*, *K*, and *X0* depend on the properties `'nk'`, `'DisturbanceModel'`, and `'InitialState'`.
  - `'Structured'`: Means that the parameterization is determined by the properties (the structure matrices) `'As'`, `'Bs'`, `'Cs'`, `'Ds'`, `'Ks'`, and `'X0s'`. A `NaN` in any position in these matrices denotes a freely adjustable parameter, and a numeric value denotes a fixed and nonadjustable parameter.
- `nk`: A row vector with as many entries as the number of input channels. The entry number `k` denotes the time delay from input number `k` to `y(t)`. This property is relevant only for `'Free'` and `'Canonical'` parameterizations. If any delay is larger than 1, the structure of the *A*, *B*, and *C* matrices will accommodate this delay, at the price of a higher-order model.
- `DisturbanceModel` has the following possible values:
  - `'Estimate'`: Means that the *K* matrix is fully parameterized.
  - `'None'`: Means that the *K* matrix is fixed to zero. This gives a so-called output-error model, since the model output depends on past inputs only.
  - `'Fixed'`: Means that the *K* matrix is fixed to the current nominal values.

- `InitialState` has the following possible values:
  - `'Estimate'`: Means that *X0* is fully parameterized.
  - `'Zero'`: Means that *X0* is fixed to zero.
  - `'Fixed'`: Means that *X0* is fixed to the current nominal value.
  - `'Backcast'`: The value of *X0* is estimated by the identification routines as the best fit to data, but it is not stored.
  - `'Auto'`: Gives an automatic and data-dependent choice among `'Estimate'`, `'Zero'`, and `'Backcast'`.
- `A`, `B`, `C`, `D`, `K`, and `X0`: The state-space matrices that can be set and retrieved at any time. These contain both fixed values and estimated/nominal values.
- `dA`, `dB`, `dC`, `dD`, `dK`, and `dX0`: The estimated standard deviations of the state-space matrices. These cannot be set, only retrieved. Note that these are not defined for an `idss` model with `'Free'` `SSParameterization`. You can then convert the parameterization to `'Canonical'` and study the uncertainties of the matrix elements in that form.
- `As`, `Bs`, `Cs`, `Ds`, `Ks`, and `X0s`: These are the structure matrices that have the same sizes as `A, B, C,` etc., and show the freely adjustable parameters as `NaN`s in the corresponding position. These properties are used to define the model structure for `'SSParameterization' = 'Structured'`. They are always defined, however, and can be studied also for the other parameterizations.
- `CanonicalIndices`: Determines the details of the canonical parameterization. It is a row vector of integers with as many entries as there are outputs. They sum up to the system order. This is the so-called pseudocanonical multiindex with an exact definition, for example, on page 132 in Ljung (1999). A good default choice is `'Auto'`. This property is relevant only for the canonical parameterization case. Note however, that for `'Free'` parameterizations, the estimation algorithms also store a canonically parameterized model to handle the model uncertainty.

In addition to these properties, `idss` objects also have all the properties of the `idmodel` object. See `idmodel` properties, `Algorithm Properties`, and `EstimationInfo`.

Note that all properties can be set and retrieved either by the `set` and `get` commands or by subscripts. Autofill applies to all properties and values, and these are case insensitive.

```
m.ss='can'
set(m,'ini','z')
p = eig(m.a)
```

For a complete list of property values, use `get(m)`. To see possible value assignments, use `set(m)`. See also `idprops idss`.

Examples

Define a continuous-time model structure corresponding to

$$\dot{x} = \begin{bmatrix} \theta_1 & 0 \\ 0 & \theta_2 \end{bmatrix} x + \begin{bmatrix} \theta_3 \\ \theta_4 \end{bmatrix} u$$

$$y = \begin{bmatrix} 1 & 1 \end{bmatrix} x + e$$

with initial values

$$\theta = \begin{bmatrix} -0.2 \\ -0.3 \\ 2 \\ 4 \end{bmatrix}$$

and estimate the free parameters.

```
A = [-0.2, 0; 0, -0.3]; B = [2;4]; C=[1, 1]; D = 0
m0 = idss(A,B,C,D);
m0.As = [NaN,0;0,NaN];
m0.Bs = [NaN;NaN];
m0.Cs = [1,1];
```

```
m0.Ts = 0;
m = pem(z,m0);
```

Estimate a model in free parameterization. Convert it to continuous time, then convert it to canonical form and continue to fit this model to data.

```
m1 = n4sid(data,3);
m1 = d2c(m1);
m1.ss ='can';
m = pem(data,m1);
```

All of this can be done at once by

```
m = pem(data,3,'ss','can','ts',0)
```

See Also

`n4sid`

`pem`

`setstruc`

Purpose

Transform `iddata` objects from frequency to time domain

Syntax

```
dat = ifft(Datf)
```

Description

`ifft` transforms a frequency-domain `iddata` object to the time domain. It requires the frequencies on `Datf` to be equally spaced from frequency 0 to the Nyquist frequency. This means that if there are `N` frequencies in `Datf` and the time sampling interval is `Ts`, then

`Datf.Frequency = [0:df:F]`, where `F` is `pi/Ts` if `N` is odd and `F = pi/Ts*(1-1/N)` if `N` is even.

See Also

`iddata`

`fft`

Purpose

Plot impulse response with confidence interval

Syntax

```
impulse(m)
impulse(data)
impulse(m,'sd',sd,Time)
impulse(m,'sd',sd,Time,'fill')
impulse(data,'sd',sd,'pw',na,Time)
impulse(m1,m2,...,dat1, ...,mN,Time,'sd',sd)
impulse(m1,'PlotStyle1',m2,'PlotStyle2',...,dat1,'PlotStylek',...,
mN,'PlotStyleN',Time,'sd',sd)
[y,t,ysd] = impulse(m)
mod = impulse(data)
```

Description

`impulse` can be applied both to `idmodels` and to `iddata` sets, as well as to any mixture.

For a discrete-time `idmodel` `m`, the impulse response `y` and, when required, its estimated standard deviation `ysd`, are computed using `sim`. When called with output arguments, `y`, `ysd`, and the time vector `t` are returned. When `impulse` is called without output arguments, a plot of the impulse response is shown. If `sd` is given a value larger than zero, a confidence region around zero is drawn. It corresponds to the confidence of `sd` standard deviations. In the plots, the impulse is inversely scaled with the sampling interval so that it has the same energy regardless of the sampling interval.

Adding an argument `'fill'` among the input arguments gives an uncertainty region marked by a filled area rather than by dash-dotted lines.

Setting the Time Interval

You can specify the start time `T1` and the end time `T2` using `Time= [T1 T2]`. If `T1` is not given, it is set to `-T2/4`. The negative time lags (the impulse is always assumed to occur at time 0) show possible feedback effects in the data when the impulse is estimated directly from data. If `Time` is not specified, a default value is used.

Estimating the Impulse Response from data

For an `iddata` set data, `impulse(data)` estimates a high-order, noncausal FIR model after first having prefiltered the data so that the input is “as white as possible.” The impulse response of this FIR model and, when asked for, its confidence region, are then plotted. Note that it is not always possible to deliver the demanded time interval when the response is estimated. A warning is then issued. When called with an output argument, `impulse`, in the `iddata` case, returns this FIR model, stored as an `idarx` model. The order of the prewhitening filter can be specified by the property name/property value pair `'pw'/na`. The default value is `na = 10`.

Several Models/Data Sets

Any number and any mixture of models and data sets can be used as input arguments. The responses are plotted with each input/output channel (as defined by the model and data set `InputName` and `OutputName` properties) as a separate plot. Colors, line styles, and marks can be defined by `PlotStyle` values. These are the same as for the regular `plot` command, as in

```
impulse(m1,'b-*',m2,'y--',m3,'g')
```

Noise Channels

The noise input channels in `m` are treated as follows: Consider a model `m` with both measured input channels *u* (*nu* channels) and noise channels *e* (*ny* channels) with covariance matrix Λ

$$y = Gu + He$$
$$\mathtt{cov}(e) = \Lambda = LL'$$

where L is a lower triangular matrix. Note that `m.NoiseVariance =` Λ. The model can also be described with unit variance, normalized noise source v:

$$y = Gu + HLv$$
$$\mathtt{cov}(v) = I$$

- `impulse(m)` plots the impulse response of the transfer function *G*.
- `impulse(m('n'))` plots the impulse response of the transfer function *H* (*ny* inputs and *ny* outputs). The input channels have names `e@yname`, where `yname` is the name of the corresponding output.
- If `m` is a time series, that is *nu* `= 0,impulse(m)`, plots the impulse response of the transfer function *H*.
- `impulse(noisecnv(m))` plots the impulse response of the transfer function [*G H*] (*nu*+*ny* inputs and *ny* outputs). The noise input channels have names `e@yname`, where `yname` is the name of the corresponding output.
- `impulse(noisecnv(m,'norm'))p` lots the impulse response of the transfer function [*G HL*] (*nu*+*ny* inputs and *ny* outputs). The noise input channels have names `v@yname`, where `yname` is the name of the corresponding output.

Arguments

If `impulse` is called with a single `idmodel m`, the output argument `y` is a 3-D array of dimension `Nt`-by-`ny`-by-`nu`. Here `Nt` is the length of the time vector `t`, `ny` is the number of output channels, and `nu` is the number of input channels. Thus `y(:,ky,ku)` is the response in output `ky` to an impulse in the `ku`th input channel.

`ysd` has the same dimensions as `y` and contains the standard deviations of `y`.

If `impulse` is called with an output argument and a single data set in the input arguments, the output is returned as an `idarx` model `mod` containing the high-order FIR model and its uncertainty. By calling `impulse` with `mod`, the responses can be displayed and returned without your having to redo the estimation.

Examples

`impulse(data,'sd',3)` estimates and plots the impulse response. To take a closer look at subsystems, do the following:

```
mod = impulse(data)
impulse(mod(2,3),'sd',3)
```

See Also

`cra`

`step`

Purpose

Set or randomize initial parameter values

Syntax

```
m = init(m0)
m = init(m0,R,pars,sp)
```

Description

This function randomizes initial parameter estimates for model structures `m0` for any `idmodel`, `idnlarx`, and `idnlhw` model object. `m` is the same model structure as `m0`, but with a different nominal parameter vector. This vector is used as the initial estimate by `pem`.

The parameters are randomized around `pars` with variances given by the row vector `R`. Parameter number *k* is randomized as `pars(k) + e*sqrt(R(k))`, where `e` is a normal random variable with zero mean and a variance of 1. The default value of `R` is all ones, and the default value of `pars` is the nominal parameter vector in `m0`.

Only models that give stable predictors are accepted. If `sp = 'b'`, only models that are both stable and have stable predictors are accepted.

`sp = 's'` requires stability only of the model, and `sp = 'p'` requires stability only of the predictor. `sp = 'p'` is the default.

Sufficiently free parameterizations can be stabilized by direct means without any random search. To just stabilize such an initial model, set `R = 0`. With `R > 0`, randomization is also done.

For model structures where a random search is necessary to find a stable model/predictor, a maximum of 100 trials is made by `init`. It can be difficult to find a stable predictor for high-order systems by trial and error.

See Also

`idnlarx`

`idnlhw`

`idmodel`

`pem`

Purpose

Determine whether model or data set contains real parameters or data

Syntax

```
isreal(Data)
isreal(Model)
```

Description

`Data` is an `iddata` set and `Model` is any `idmodel`. The `isreal` function returns `1` if all parameters of the model are real and if all signals of the data set are real.

See Also

`realdata`

Purpose

Estimate AR model using instrumental variable method returning `idpoly` object

Syntax

```
m = ivar(y,na)
m = ivar(y,na,nc,maxsize)
```

Description

The parameters of an AR model structure

$$A(q)y(t) = v(t)$$

are estimated using the instrumental variable method. `y` is the signal to be modeled, entered as an `iddata` object (outputs only). `na` is the order of the A polynomial (the number of A parameters). The resulting estimate is returned as an `idpoly` model `m`. The routine is for scalar time-domain signals only.

In the above model, $v(t)$ is an arbitrary process, assumed to be a moving average process of order `nc`, possibly time varying. (Default is `nc = na`.) Instruments are chosen as appropriately filtered outputs, delayed `nc` steps.

The optional argument `maxsize` is explained under `Algorithm Properties`.

Examples

Compare spectra for sinusoids in noise, estimated by the IV method and by the forward-backward least squares method.

```
y = iddata(sin([1:500]'*1.2) + sin([1:500]'*1.5) +
0.2*randn(500,1),[]);
miv = ivar(y,4);
mls = ar(y,4);
bode(miv,mls)
```

References

Stoica, P., et al., *Optimal Instrumental variable estimates of the AR-parameters of an ARMA process*, IEEE Trans. Autom. Control, Vol. AC-30, 1985, pp. 1066-1074.

See Also

`Algorithm Properties`

`EstimationInfo`

`ar`

`arx`

`etfe`

`idpoly`

`pem`

`spa`

`step`

Purpose

Compute loss functions for sets of output-error model structures

Syntax

```
v = ivstruc(ze,zv,NN)
v = ivstruc(ze,zv,NN,p,maxsize)
```

Description

`NN` is a matrix that defines a number of different structures of the ARX type. Each row of `NN` is of the form

```
nn = [na nb nk]
```

with the same interpretation as described for `arx`. See `struc` for easy generation of typical `NN` matrices for single-input systems.

`ze` and `zv` are `iddata` objects containing output-input data. Only time-domain data is supported. Models for each model structure defined in `NN` are estimated using the instrumental variable (IV) method on data set `ze`. The estimated models are simulated using the inputs from data set `zv`. The normalized quadratic fit between the simulated output and the measured output in `zv` is formed and returned in `v`. The rows below the first row in `v` are the transpose of `NN`, and the last row contains the logarithms of the condition numbers of the IV matrix

$$\sum \varsigma(t)\varphi^T(t)$$

A large condition number indicates that the structure is of unnecessarily high order (see page 498 in Ljung (1999)).

The information in `v` is best analyzed using `selstruc`.

If `p` is equal to zero, the computation of condition numbers is suppressed. For the use of `maxsize`, see `Algorithm Properties`.

The routine is for single-output systems only.

Note The IV method used does not guarantee that the models obtained are stable. The output-error fit calculated in `v` can then be misleading.

Examples

Compare the effect of different orders and delays, using the same data set for both the estimation and validation.

```
v = ivstruc(z,z,struc(1:3,1:2,2:4));
nn = selstruc(v)
m = iv4(z,nn);
```

Algorithm

A maximum-order ARX model is computed using the least squares method. Instruments are generated by filtering the input(s) through this model. The models are subsequently obtained by operating on submatrices in the corresponding large IV matrix.

See Also

`arxstruc`

`iv4`

`selstruc`

`struc`

Purpose

Estimate parameters of ARX model using instrumental variable method with arbitrary instruments returning `idpoly` or `idarx` object

Syntax

```
m = ivx(data,orders,x)
m = ivx(data,orders,x,maxsize)
```

Description

`ivx` is a routine analogous to the `iv4` routine, except that you can use arbitrary instruments. These are contained in the matrix `x`. Make this the same size as the output, `data.y`. In particular, if data contains several experiments, `x` must be a cell array with one matrix/vector for each experiment. The instruments used are then analogous to the regression vector itself, except that `y` is replaced by `x`.

Note that `ivx` does not return any estimated covariance matrix for `m`, since that requires additional information. `m` is returned as an `idpoly` object for single-output systems and as an `idarx` object for multioutput systems.

Use `iv4` as the basic IV routine for ARX model structures. The main interest in `ivx` lies in its use for nonstandard situations, for example, when there is feedback present in the data, or when other instruments need to be tried out. Note that there is also an IV version that automatically generates instruments from certain filters you define (type `help iv`).

References

Ljung (1999), page 222.

See Also

`Algorithm Properties`

`EstimationInfo`

`arx`

`idarx`

`idpoly`

`iv4`

`pem`

Purpose

Estimate ARX model using four-stage instrumental variable method returning `idpoly` or `idarx` object

Syntax

```
m = iv4(data,orders)
m = iv4(data,'na',na,'nb',nb,'nk',nk)
m= iv4(data,orders,'Property1',Value1,...,'PropertyN',ValueN)
```

Description

This function is an alternative to `arx` and the use of the arguments is entirely analogous to the `arx` function. The main difference is that the procedure is not sensitive to the color of the noise term $e(t)$ in the model equation.

Examples

Here is an example of a two-input, one-output system with different delays on the inputs u_1 and u_2.

```
z = iddata(y, [u1 u2]);
nb = [2 2];
nk = [0 2];
m= iv4(z,[2 nb nk]);
```

Algorithm

The first stage uses the `arx` function. The resulting model generates the instruments for a second-stage IV estimate. The residuals obtained from this model are modeled as a high-order AR model. At the fourth stage, the input-output data is filtered through this AR model and then subjected to the IV function with the same instrument filters as in the second stage.

For the multioutput case, optimal instruments are obtained only if the noise sources at the different outputs have the same color. The estimates obtained with the routine are reasonably accurate, however, even in other cases.

References

Ljung (1999), equations (15.21) through (15.26).

See Also

`Algorithm Properties`

`EstimationInfo`

`arx`

`idarx`

`idpoly`

`ivx`

`pem`

Purpose

Linear approximation of nonlinear ARX and Hammerstein-Wiener models for given input

Syntax

```
lm = linapp(nlmodel,u)
lm = linapp(nlmodel,umin,umax,nsample)
```

Arguments

`nlmodel`
: Name of the `idnlarx` or `idnlhw` model object you want to linearize.

`u`
: Input signal as an iddata object or a real matrix.

  Dimensions of `u` must match the number of inputs in `nlmodel`.

`[umin,umax]`
: Minimum and maximum input values for generating white-noise input with a magnitude in this rectangular range. The sample length of this signal is `nsample`.

`nsample`
: Optional argument when you specify `[umin,umax]`. Specified the length of the white-noise input. Default: `1024`.

Description

`lm = linapp(nlmodel,u)` computes a linear approximation for a nonlinear black-box model for a given input.

`lm = linapp(nlmodel,umin,umax,nsample)` computes a linear approximation of a nonlinear black-box model for a generated input. The input signal is specified by the magnitude range and (optionally) the number of samples.

The following table summarizes the linear model objects that store the linear approximation for each type of nonlinear model and the number of outputs.

Nonlinear Model Type	Number of Outputs	Linear Model Object
`idnlarx`	Single output	`idpoly`
`idnlarx`	Multiple outputs	`idarx`
`idnlhw`	Single output	`idpoly`
`idnlhw`	Multiple outputs	`idss`

Remarks

`linapp` computes the best linear approximation—in an mean-square-error sense—of a nonlinear ARX or Hammerstein-Wiener model for a given input or a randomly generated input in a specified range.

For Hammerstein-Wiener models, `linapp` differs from `lintan`, which linearizes in a small neighborhood of a constant input using series expansion.

See Also

`idarx`

`idnlarx`

`idnlhw`

`idpoly`

`idss`

`lintan`

Purpose

Specify to estimate nonlinear ARX model that is linear in (nonlinear) custom regressors

Syntax

```
lin=linear
lin=linear('Parameters',Par)
```

Description

`linear` is an object that specifies that the nonlinear ARX model is linear in custom (nonlinear) regressors. You define custom regressors using `customreg`.

`lin=linear` instantantiates the `linear` object.

`lin=linear('Parameters',Par)` instantantiates the `linear` object and specifies optional values in the `Par` structure. For more information about this structure, see "linear Properties" on page 12-208.

Remarks

`linear` is a linear (affine) function $y = F(x)$, defined as follows:

$$F(x) = xL + d$$

y is scalar, and x is a 1-*by*-`m` vector.

Use `evaluate(lin,x)` to compute the value of the function defined by the `linear` object `lin` at `x`.

linear Properties

You can include property-value pairs in the constructor to specify the object.

After creating the object, you can use `get` or dot notation to access the object property values. For example:

```
% List Parameters values
get(lin)
% Get value of Parameters property
lin.Parameters
```

You can also use `set` or dot notation to assign property values to the object.

The Parameters property is a structure. To set the values of this structure to MATLAB variable values L and d, you can use the following syntax, for example:

```
Par=struct('LinearCoef',L,
           'OutputOffset',d);
C.Parameters=Par;
```

Property Name	Description
Parameters	Structure containing the following fields: • LinearCoef: m-*by*-1 vector *L*. • OutputOffset: Scalar d.

Examples

To specify that the nonlinear ARX model is linear in custom regressors, first specify one or more customreg objects. Then, include the linear object in the nlarx estimator command.

For example, to estimate a nonlinear ARX model linear in the custom regressors that might be nonlinear in input u and output y data, use the following syntax:

```
m=nlarx(Data,Orders,linear,'custom',{'y(t-1)*u(t-2)'});
```

See Also

customreg

nlarx

lintan

Purpose

Tangent linearization of Hammerstein-Wiener models about operating point

Syntax

```
lm = lintan(nlmodel)
lm = lintan(nlmodel,u)
```

Arguments

`nlmodel`

Name of the `idnlhw` model object you want to linearize.

`u`

Constant input signal, given as a scalar value or a vector. The length of this vector must match the number of inputs in `nlmodel`.

Default: `0`.

Description

`lm = lintan(nlmodel)` computes a linearization of a nonlinear `idnlhw` model around an equilibrium point such that the `nlmodel` output is constant when the input is constant (after transient effects subside).

`lm = lintan(nlmodel,u)` computes a linearization for a nonzero value of the constant input.

The following table summarizes the linear model objects that store the linear approximation for the number of outputs.

Number of Outputs	Linear Model Object
Single output	`idpoly`
Multiple outputs	`idss`

Remarks

`lintan` computes a linear model for inputs that vary in a small (infinitesimal) neighborhood of a constant input. This linearization method is based on performing a series expansion of a function.

For linear approximations that must perform accurately over larger input ranges, use `linapp`.

See Also

`idnlhw`

`idpoly`

`idss`

`linapp`

LTI Commands

Purpose Allow direct calls to LTI commands from `idmodel` objects (requires Control System Toolbox)

Syntax `append, augstate, balreal, canon, d2d, feedback, inv, minreal, modred, norm, parallel, series, ss2ss`

Description If you have Control System Toolbox, most of the relevant LTI commands, listed above, can be directly applied to any `idmodel` (`idarx`, `idgrey`, `idpoly`, `idss`). You can also use the overloaded operations +, -, and *. The same operations are performed and the result is delivered as an `idmodel`. The original covariance information is lost most of the time, however.

Examples You have two more or less identical processes connected in series. Estimate a model for one of them, and use that to form an initial estimate for a model of the connected process.

```
% data concerns one of the processes
m = pem(data)
% data2 is from the entire connected process
m2 = pem(data2,m*m)
```

Purpose

Merge data sets into one `iddata` object

Syntax

```
dat = merge(dat1,dat2,....,datN)
```

Description

`dat` collects the data sets in `dat1,.. datN` into one `iddata` object, with several *experiments*. The number of experiments in `dat` will be the sum of the number of experiments in `datk`. For the merging to be allowed, a number of conditions must be satisfied:

- All of `datk` must have the same number of input channels, and the `InputNames` must be the same.
- All of `datk` must have the same number of output channels, and the `OutputNames` must be the same. If some input or output channel is lacking in one experiment, it can be replaced by a vector of `NaN`s to conform with these rules.
- If the `ExperimentNames` of `datk` have been specified as something other than the default `'Exp1'`, `'Exp2'`, etc., they must all be unique. If default names overlap, they are modified so that `dat` will have a list of unique `ExperimentNames`.

The sampling intervals, the number of observations, and the input properties (`Period`, `InterSample`) might be different in the different experiments.

You can retrieve the individual experiments by using the command `getexp`. You can also retrieve them by subreferencing with a fourth index.

```
dat1 = dat(:,:,:,ExperimentNumber) or
dat1 = dat(:,:,:,ExperimentName)
```

Storing multiple experiments as one `iddata` object can be very useful for handling experimental data that has been collected on different occasions, or when a data set has been split up to remove “bad” portions of the data. All the toolbox’s routines accept multiple-experiment data.

Examples

Bad portions of data have been detected around sample 500 and between samples 720 to 730. Cut out these bad portions and form a multiple-experiment data set that can be used to estimate models without the bad data destroying the estimate.

```
dat = merge(dat(1:498),dat(502:719),dat(719:1000))
m = pem(dat)
```

Use the first two parts to estimate the model and the third one for validation.

```
m = pem(getexp(dat,[1,2]));
compare(getexp(dat,3),m)
```

See also `iddemo` #9.

See Also

`iddata`

`getexp`

Purpose

Merge estimated idmodel models

Syntax

```
m = merge(m1,m2,....,mN)
[m,tv] = merge(m1,m2)
```

Description

The models m1,m2,...,mN must all be of the same structure, just differing in parameter values and covariance matrices. Then m is the merged model, where the parameter vector is a statistically weighted mean (using the covariance matrices to determine the weights) of the parameters of mk.

When two models are merged,

```
   [m, tv] = merge(m1,m2)
```

returns a test variable tv. It is χ^2 distributed with n degrees of freedom, if the parameters of m1 and m2 have the same means. Here n is the length of the parameter vector. A large value of tv thus indicates that it might be questionable to merge the models.

Merging models is an alternative to merging data sets and estimating a model for the merged data. Consequently,

```
   m1 = arx(z1,[2 3 4]);
   m2 = arx(z2,[2 3 4]);
   ma = merge(m1,m2);
```

and

```
   mb = arx(merge(z1,z2),[2 3 4]);
```

lead to models ma and mb that are related and should be close. The difference is that merging the data sets assumes that the signal-to-noise ratios are about the same in the two experiments. Merging the models allows one model to be much more uncertain, for example, due to more disturbances in that experiment. If the conditions are about the same, we recommend that you merge data rather than models, since this is more efficient and typically involves better conditioned calculations.

midprefs

Purpose

Set directory for storing `idprefs.mat` containing GUI startup information

Syntax

```
midprefs
midprefs(path)
```

Description

The graphical user interface `ident` allows a large number of variables for customized choices. These include the window layout, the default choices of plot options, and names and directories of the four most recent sessions with `ident`. This information is stored in the file `idprefs.mat`, which should be placed on the user's `MATLABPATH`. The default, automatic location for this file is in the same directory as the user's `startup.m` file.

`midprefs` is used to select or change the directory where you store `idprefs.mat`. Either type `midprefs` and follow the instructions, or give the directory name as the argument. Include all directory delimiters, as in the PC case

```
midprefs('c:\matlab\toolbox\local\')
```

or in the UNIX case

```
midprefs('/home/ljung/matlab/')
```

See Also

`ident`

Purpose

Reconstruct missing input and output data

Syntax

```
Datae = misdata(Data)
Datae = misdata(Data,Model)
Datae = misdata(Data,Maxiter,Tol)
```

Description

`Data` is time-domain input-output data in the `iddata` object format. Missing data samples (both in inputs and in outputs) are entered as `NaN`s.

`Datae` is an `iddata` object where the missing data has been replaced by reasonable estimates.

`Model` is any `idmodel` (`idarx`, `idgrey`, `idpoly`, `idss`) used for the reconstruction of missing data.

If no suitable model is known, it is estimated in an iterative fashion using default order state-space models.

`Maxiter` is the maximum number of iterations carried out (the default is 10). The iterations are terminated when the difference between two consecutive data estimates differs by less than `tol`%. The default value of `tol` is `1`.

Algorithm

For a given model, the missing data is estimated as parameters so as to minimize the output prediction errors obtained from the reconstructed data. See Section 14.2 in Ljung (1999). Treating missing outputs as parameters is not the best approach from a statistical point of view, but is a good approximation in many cases.

When no model is given, the algorithm alternates between estimating missing data and estimating models, based on the current reconstruction.

neuralnet

Purpose

Store neural network object created in Neural Network Toolbox for estimating nonlinear ARX and Hammerstein-Wiener models

Syntax

```
n=neuralnet(Network)
```

Description

`neuralnet` is an object that stores the neural network nonlinearity estimator for estimating nonlinear ARX and Hammerstein-Wiener models. Requires Neural Network Toolbox.

You can use the constructor to create the nonlinearity object, as follows:

`n=neuralnet(Network)` creates a neural network nonlinearity estimator based on the network object you created in Neural Network Toolbox.

Use `evaluate(n,x)` to compute the value of the function defined by the `neuralnet` object `n` at `x`.

Remarks

Use `neuralnet` to define a nonlinear function $y = F(x)$, where F is a multilayer feedforward neural network, as defined in Neural Network Toolbox.

y is a scalar and x is an `m`-dimensional row vector.

If you purchased Neural Network Toolbox, you can create a multilayer feedforward neural network using `newff`:

```
ff = newff(MV,[nL_1,nL_2,..,nL_r],{tf_1,tf_2,...,tf_r})
```

where `MV` is an `m`-*by*-2 matrix containing minimum and maximum values of `x`.

There are `r` layers and `nL_k` neurons in the `k`th layer. In this application, the last layer must have one neuron, such that `nL_r=1`. The transfer function (or unit function) in the `k`th layer is `tf_k`.

If `m` is unknown at the time of creation of the network, use `MV = zeros(0,2)`. After this initialization, `m` is adjusted to the estimation data by `nlarx` or `nlhw`.

neuralnet Properties

You include the property as an argument in the constructor to specify the object.

After creating the object, you can use get or dot notation to access the object property values. For example:

```
% List Network property value
get(n)
n.Network
```

You can use dot notation to assign property values to the object. set is not supported for MCOS objects.

For example, the following two commands are equivalent:

```
n.Network=net_obj
```

Property Name	Description
Network	Multilayer feedforward neural network object, defined in Neural Network Toolbox using newff.

Examples

Use neuralnet to specify the neural network nonlinearity estimator in nonlinear ARX and Hammerstein-Wiener models. For example:

```
% Create network object using Neural Network Toolbox
net_obj=newff(zeros(0,2),[6 8 1],...
              {'logsig','logsig','purelin'})
% Estimate nonlinear ARX model using
% net_obj as the neural network
m=nlarx(Data,Orders,neuralnet(net_obj));
```

See Also

nlarx

nlhw

nkshift

Purpose

Shift data sequences

Syntax

```
Datas = nkshift(Data,nk)
```

Description

`Data` contains input-output data in the `iddata` format.

`nk` is a row vector with the same length as the number of input channels in `Data`.

`Datas` is an `iddata` object where the input channels in `Data` have been shifted according to `nk`. A positive value of `nk(ku)` means that input channel number `ku` is delayed `nk(ku)` samples.

`nkshift` supports both frequency- and time-domain data. For frequency-domain data it multiplies with $e^{in k\omega T}$ to obtain the same effect as shifting in the time domain. For continuous-time frequency-domain data (`Ts = 0`), `nk` should be interpreted as the shift in seconds.

`nkshift` lives in symbiosis with the `InputDelay` property of `idmodel`:

```
   m1 = pem(dat,4,'InputDelay',nk)
```

is related to

```
   m2 = pem(nkshift(dat,nk),4);
```

such that `m1` and `m2` are the same models, but `m1` stores the delay information for use when frequency responses, etc., are computed.

Note the difference from the `idss` and `idpoly` property `nk`.

```
   m3 = pem(dat,4,'nk',nk)
```

gives a model that itself explicitly contains a delay of `nk` samples.

See Also

`Algorithm Properties`

`idss`

Purpose

Estimate nonlinear ARX models

Syntax

```
m=nlarx(data,[na nb nk],Nonlinearity)
m=nlarx(data,[na nb nk],P1,V1,...,PN,VN)
```

Arguments

`data`

Time-domain `iddata` model object.

`[na nb nk]`

`na` is the number of output terms, `nb` is the number of input terms, and `nk` is the input delays from each input to output.

For `ny` outputs and `nu` inputs, `[na nb nk]` has as many rows as there are outputs. In this case, `na` is an `ny`-by-`ny` matrix whose *i*-*j*th entry gives the number of delayed *j*th outputs used to compute the *i*th output. `nb` and `nk` are `ny`-by-`nu` matrices.

These orders specify the regressors and the predicted output is the following function of these regressors:

$$F\left(y(t-1),\ldots,y(t-na),u(t-nk),\ldots,u(t-nk-nb+1)\right)$$

`Nonlinearity`

Specifies the nonlinearity estimator object as one of the following: `sigmoidnet` (default), `wavenet`, `treepartition`, `customnet`, `neuralnet`, and `linear`.

For `ny` outputs, `Nonlinearity` is an `ny`-by-1 array, such as `[sigmoidnet;wavenet]`. However, if you specify a scalar object, this nonlinearity object applies to all outputs.

For more information about nonlinearity properties, see the corresponding reference pages.

Description

`m=nlarx(data,[na nb nk],Nonlinearity)` constructs and estimates a nonlinear ARX model with orders `[na nb nk]` and `Nonlinearity`. data is the estimation data set.

`m=nlarx(data,[na nb nk],P1,V1,...,PN,VN)` constructs and estimates the model with additional property-value pairs. For more information about model `idnlarx` model properties, see the corresponding reference pages.

Examples

The following commands construct and estimate a nonlinear ARX model:

```
load iddata1
m1=nlarx(z1,[4 2 1],'wave','nlr',[1:3])
```

To perturb the parameters slightly and avoid being trapped in local minima, use the `init` command:

```
m2=init(m1)
```

Estimate the model with perturbed initial parameter values, use the following command:

```
m2=nlarx(z1,m2)
```

See Also

`addreg`

`customreg`

`getreg`

`idnlarx`

`init`

`polyreg`

Purpose

Estimate Hammerstein-Wiener models

Syntax

```
m=nlarx(data,[na nb nk],Nonlinearity)
m=nlarx(data,[na nb nk],P1,V1,...,PN,VN)
```

Arguments

`data`

Time-domain `iddata` model object.

`[nb nf nk]`

Model orders and input delays, where `nb` is the number of zeros plus 1, `nf` is the number of poles, and `nk` is the delay from input to output in terms of the number of samples.

For `nu` inputs and `ny` outputs, `nb`, `nf` and, `nk` are `ny`-by-`nu` matrices whose *i*-*j*th entry specifies the orders and delay of the transfer function from the *j*th input to the *i*th output.

`InputNL` and `OutputNL`

Specify the input and output nonlinearity estimator objects as one of the following: `pwlinear` (default), `deadzone`, `wavenet`, `saturation`, `customnet`, `sigmoidnet`, and `unitgain`. The nonlinearity estimator objects have properties that you can set in the constructor, as follows:

```
m=nlhw(data,[2 2 1],sigmoidnet('num',5),deadzone([-1,2]))
```

To use default nonlinearity properties, specify the nonlinearity object name as a string. For example:

```
m=nlhw(data,[2 2 1],'sigmoidnet','deadzone')
m=nlhw(data,[2 2 1],'sig','dead') % Abbreviated
```

The estimator `unitgain` (can also be entered as `[]`) means no nonlinearity. Thus, `m=nlhw(data,[2 2 1],'saturation','[]')`. For more information about nonlinearity properties, see the corresponding reference pages.

Description

`m=nlarx(data,[na nb nk],Nonlinearity)` constructs and estimates a nonlinear ARX model with orders `[na nb nk]` and `Nonlinearity`. `data` is the estimation data set.

`m=nlarx(data,[na nb nk],P1,V1,...,PN,VN)` constructs and estimates the model with additional property-value pairs. For more information about model `idnlarx` model properties, see the corresponding reference pages.

Examples

The following commands construct and estimate a nonlinear ARX model:

```
load iddata1
m1=nlarx(z1,[4 2 1],'wave','nlr',[1:3])
```

To perturb the parameters slightly and avoid being trapped in local minima, use the `init` command:

```
m2=init(m1)
```

Estimate the model with perturbed initial parameter values, use the following command:

```
m2=nlarx(z1,m2)
```

See Also

`idnlhw`

`init`

`pem`

Purpose

Convert `idmodel` with noise channels to model with only measured channels

Syntax

```
mod1 = noisecnv(mod)
mod2 = noisecnv(mod,'norm')
```

Description

`mod` is any `idmodel`, `idarx`, `idgrey`, `idpoly`, or `idss`.

The noise input channels in `mod` are converted as follows: Consider a model with both measured input channels u (*nu* channels) and noise channels e (*ny* channels) with covariance matrix Λ

$$y = Gu + He$$
$$\mathrm{cov}(e) = \Lambda = LL'$$

where L is a lower triangular matrix. Note that `mod.NoiseVariance` = Λ. The model can also be described with unit variance, normalized noise source v:

$$y = Gu + HLv$$
$$\mathrm{cov}(v) = I$$

- `mod1 = noisecnv(mod)` converts the model to a representation of the system [G H] with *nu*+*ny* inputs and *ny* outputs. All inputs are treated as measured, and `mod1` does not have any noise model. The former noise input channels have names `e@yname`, where `yname` is the name of the corresponding output.
- `mod2 = noisecnv(mod,'norm')` converts the model to a representation of the system [G HL] with *nu*+*ny* inputs and *ny* outputs. All inputs are treated as measured, and `mod2` does not have any noise model. The former noise input channels have names `v@yname`, where `yname` is the name of the corresponding output. Note that the noise variance matrix factor L typically is uncertain (has a nonzero covariance). This is taken into account in the uncertainty description of `mod2`.

- If `mod` is a time series, that is, nu `= 0`, `mod1` is a model that describes the transfer function H with measured input channels. Analogously, `mod2` describes the transfer function HL.

Note the difference with subreferencing:

- `mod('m')` gives a description of G only.
- `mod('n')` gives a description of the noise model characteristics as a time-series model, that is, it describes H and also the covariance of e. In contrast, `noisecnv(m('n'))` describes just the transfer function H. To obtain a description of the normalized transfer function HL, use `noisecnv(m('n'),'norm')`.

Converting the noise channels to measured inputs is useful to study the properties of the individual transfer functions from noise to output. It is also useful for transforming `idmodel` objects to representations that do not handle disturbance descriptions explicitly.

Purpose

Set step size for numerical differentiation

Syntax

```
nds = nuderst(pars)
```

Description

The function `pem` uses numerical differentiation with respect to the model parameters when applied to state-space structures. The same is true for many functions that transform model uncertainties to other representations.

The step size used in these numerical derivatives is determined by the M-file `nuderst`. The output argument `nds` is a row vector whose `k`th entry gives the increment to be used when differentiating with respect to the `k`th element of the parameter vector `pars`.

The default version of `nuderst` uses a very simple method. The step size is the maximum of 10^{-4} times the absolute value of the current parameter and 10^{-7}. You can adjust this to the actual value of the corresponding parameter by editing `nuderst`. Note that the nominal value, for example 0, of a parameter might not reflect its normal size.

Purpose

Plot Nyquist curve of frequency response with confidence interval

Syntax

```
nyquist(m)
[fr,w] = nyquist(m)
[fr,w,covfr] = nyquist(m)
nyquist(m1,m2,m3,...,w)
nyquist(m1,'PlotStyle1',m2,'PlotStyle2',...)
nyquist(m1,m2,m3,..'sd*5',sd,'mode',mode)
```

Description

`nyquist` computes the complex-valued frequency response of `idmodel` and `idfrd` models. When invoked without left-hand arguments, `nyquist` produces a Nyquist plot on the screen, that is, a graph of the frequency response's imaginary part against its real part.

The argument `m` is an arbitrary `idmodel` or `idfrd` model. This model can be continuous or discrete, and SISO or MIMO. The `InputNames` and `OuputNames` of the models are used to plot the responses for different I/O channels in separate plots. Pressing the **Enter** key advances the plot from one input-output pair to the next one. You can select specific I/O channels with normal subreferencing: `m(ky,ku)`. With `mode = 'same'`, all plots are given in the same diagram.

`nyquist(m,w)` explicitly specifies the frequency range or frequency points to be used for the plot. To focus on a particular frequency interval `[wmin,wmax]`, set `w = {wmin,wmax}`. To use particular frequency points, set `w` to the vector of desired frequencies. Use `logspace` to generate logarithmically spaced frequency vectors. All frequencies should be specified in rad/s.

`nyquist(m1,m2,...,mN)` or `nyquist(m1,m2,...mN,w)` plots the Bode responses of several `idmodels` or `idfrd` models on a single figure. The models can be mixes of different sizes, and continuous or discrete. The sorting of the plots is based on the `InputNames` and `OutputNames`.

`nyquist(m1,'PlotStyle1',...,mN,'PlotStyleN')` further specifies which color, line style, and/or marker should be used to plot each system, as in

```
nyquist(m1,'r--',m2,'gx')
```

When sd is specified as a number larger than zero, confidence regions are also plotted. These are ellipses in the complex plane and correspond to the region where the true response at the frequency in question is to be found with a confidence corresponding to sd standard deviations (of the Gaussian distribution).

If the argument indicating standard deviations is given as in 'sd+5', a confidence region is plotted for every 5:th frequency, marking the center point by '+'. The default is 'sd+10'.

Note that the frequencies cannot be specified for idfrd objects. For those, the plot and response are calculated for the internally stored frequencies. If the frequencies w are specified when several models are treated, they will apply to all non-idfrd models in the list. If you want different frequencies for different models, you should first convert them to idfrd objects using the idfrd command.

For time-series models (no input channels), the Nyquist plot is not defined.

Arguments

When nyquist is called with a single system and output arguments,

```
fr = nyquist(m,w) or [fr,w,covfr] = nyquist(m)
```

no plot is drawn on the screen. If m has ny outputs and nu inputs, and w contains Nw frequencies, then fr is an ny-by-nu-by-Nw array such that fr(ky,ku,k) gives the complex-valued frequency response from input ku to output ky at the frequency w(k). For a SISO model, use fr(:) to obtain a vector of the frequency response. The uncertainty information covfr is a 5-D array where covfr(ky,ku,k,:,:)) is the 2-by-2 covariance matrix of the response from input ku to output ky at frequency w(k). The 1,1 element is the variance of the real part, the 2,2 element is the variance of the imaginary part, and the 1,2 and 2,1 elements are the covariance between the real and imaginary parts.

squeeze(covfr(ky,ku,k,:,:)) gives the covariance matrix of the corresponding response.

If `m` is a time series (no input), `fr` is returned as the (power) spectrum of the outputs, an `ny`-by-`ny`-by-`Nw` array. Hence `fr(:,:,k)` is the spectrum matrix at frequency `w(k)`. The element `fr(k1,k2,k)` is the cross spectrum between outputs `k1` and `k2` at frequency `w(k)`. When `k1 = k2`, this is the real-valued power spectrum of output `k1`. The `covfr` is then the covariance of the spectrum `fr`, so that `covfr(k1,k1,k)` is the variance of the power spectrum of output `k1` at frequency `w(k)`. No information about the variance of the cross spectra is normally given. (That is, `covfr(k1,k2,k) = 0` for `k1` not equal to `k2`.)

If the model `m` is not a time series, use `fr = nyquist(m('n'))` to obtain the spectrum information of the noise (output disturbance) signals.

Examples

```
g = spa(data)
m = n4sid(data,3)
nyquist(g,m,3)
```

See Also

`bode`

`etfe`

`ffplot`

`freqresp`

`idfrd`

`spa`

`spafdr`

Purpose

Estimate state-space models using subspace method returning `idss` object

Syntax

```
m = n4sid(data)
m = n4sid(data,order,'Property1',Value1,...,'PropertyN',ValueN)
```

Description

The function `n4sid` estimates models in state-space form and returns them as an `idss` object `m`. It handles an arbitrary number of input and outputs, including the time-series case (no input). The state-space model is in the innovations form

$$x(t+Ts) = Ax(t)+Bu(t)+Ke(t)$$
$$y(t) = Cx(t)+Du(t)+e(t)$$

`m`: The resulting model as an `idss` object.

If `data` is continuous-time (frequency-domain) data, a corresponding continuous-time state-space model is estimated.

`data`: An `iddata` object containing the output-input data. Both time-domain and frequency-domain signals are supported. `data` can also be a `frd` or `idfrd` frequency-response data object.

`order`: The desired order of the state-space model. If `order` is entered as a row vector (as in `order = [1:10]`), preliminary calculations for all the indicated orders are carried out. A plot is then given that shows the relative importance of the dimension of the state vector. More precisely, the singular values of the Hankel matrices of the impulse response for different orders are graphed. You are prompted to select the order, based on this plot. The idea is to choose an order such that the singular values for higher orders are comparatively small. If `order = 'best'`, a model of “best” (default choice) order is computed among the orders 1:10. This is the default choice of `order`.

Estimating the D Matrix

Whether the *D* matrix is estimated or not is governed by the property `nk`, which is further described below. The default is that *D* is not estimated. By setting the `k`th entry of `nk` to 0, the `k`th column of *D* (corresponding to the kth input) is estimated. To estimate a full *D* matrix thus, let nk = zeros(1,nu) as in

```
m = n4sid(data,order,'nk',[0 .. 0])
```

This holds for both discrete- and continuous-time models.

Property Name/Property Value Pairs

The list of property name/property value pairs can contain any idss and algorithm properties. See idss and Algorithm Properties.

idss properties that are of particular interest for n4sid are

- nk: For time-domain data, this gives delays from the inputs to the outputs, a row vector with the same number of entries as the number of input channels. Default is nk = [1 1... 1]. Note that delays of 0 or 1 show up as zeros or estimated parameters in the D matrix. Delays larger than 1 mean that a special structure of the A, B, and C matrices is used to accommodate the delays. This also means that the actual order of the state-space model will be larger than order. For continuous-time models estimated from continuous-time (frequency-domain) data, the elements of nk are restricted to the values 1 and 0.
- CovarianceMatrix (can be abbreviated to 'co'): Setting CovarianceMatrix to 'None' blocks all calculations of uncertainty measures. These can take the major part of the computation time. Note that, for a 'Free' parameterization, the individual matrix elements cannot be associated with any variance. (These parameters are not identifiable.) Instead, the resulting model m stores a hidden state-space model in canonical form that contains covariance information. This is used when the uncertainty of various input-output properties is calculated. It can also be retrieved by m.ss = 'can'. The actual covariance properties of n4sid estimates are not known today. Instead the Cramer-Rao bound is computed and stored as an indication of the uncertainty.
- DisturbanceModel: Setting DisturbanceModel to 'None' will deliver a model with K = 0. This has no direct effect on the dynamics

model other than that the default choice of `N4Horizon` will not involve past outputs.

- `InitialState`: The initial state is always estimated for better accuracy. However, it is returned with `m` only if `InitialState = 'Estimate'`.

Algorithm properties that are of special interest are

- `Focus`: Assumes the values `'Prediction'` (default), `'Simulation'`, `'Stability'`, passbands, or any SISO linear filter (given as an LTI or `idmodel` object, or as filter coefficients. See `Algorithm Properties`.) Setting `'Focus'` to `'Simulation'` chooses weights that should give a better simulation performance for the model. In particular, a stable model is guaranteed. Selecting a linear filter focuses the fit to the frequency ranges that are emphasized by this filter.
- `N4Weight`: This property determines some weighting matrices used in the singular-value decomposition that is a central step in the algorithm. Two choices are offered: `'MOESP'`, corresponding to the MOESP algorithm by Verhaegen, and `'CVA'`, which is the canonical variable algorithm by Larimore. The default value is `'N4Weight' = 'Auto'`, which gives an automatic choice between the two options. `m.EstimationInfo.N4Weight` tells you what the actual choice turned out to be.
- `N4Horizon`: Determines the prediction horizons forward and backward used by the algorithm. This is a row vector with three elements: `N4Horizon = [r sy su]`, where `r` is the maximum forward prediction horizon. That is, the algorithm uses up to `r` step-ahead predictors. `sy` is the number of past outputs, and `su` is the number of past inputs that are used for the predictions. See pages 209 and 210 in Ljung (1999) for the exact meaning of this. These numbers can have a substantial influence on the quality of the resulting model, and there are no simple rules for choosing them. Making `'N4Horizon'` a `k`-by-3 matrix means that each row of `'N4Horizon'` is tried, and the value that gives the best (prediction) fit to data is selected. (This option cannot be combined with selection

of model order.) If the property 'Trace' is 'On', information about the results is given in the MATLAB Command Window.

If you specify only one column in 'N4Horizon', the interpretation is r=sy=su. The default choice is 'N4Horizon' = 'Auto', which uses an Akaike Information Criterion (AIC) for the selection of sy and su. If 'DisturbanceModel' = 'None', sy is set to 0. Typing m.EstimationInfor.N4Horizon will tell you what the final choices of horizons were.

Algorithm

The variants of the implemented algorithm are described in Section 10.6 in Ljung (1999).

Examples

Build a fifth-order model from data with three inputs and two outputs. Try several choices of auxiliary orders. Look at the frequency response of the model.

```
z = iddata([y1 y2],[ u1 u2 u3]);
m = n4sid(z,5,'n4h',[7:15]','trace','on');
bode(m,'sd',3)
```

Estimate a continuous-time model, in a canonical form parameterization, focusing on the simulation behavior. Determine the order yourself after seeing the plot of singular values.

```
m = n4sid(m,[1:10],'foc','sim','ssp','can','ts',0)
bode(m)
```

References

vanOverschee, P., and B. DeMoor, *Subspace Identification of Linear Systems: Theory, Implementation, Applications*, Kluwer Academic Publishers, 1996.

Verhaegen, M., “Identification of the deterministic part of MIMO state space models,” *Automatica,* Vol. 30, pp. 61-74, 1994.

Larimore, W.E., “Canonical variate analysis in identification, filtering and adaptive control,” In *Proc. 29th IEEE Conference on Decision and Control*, pp. 596-604, Honolulu, 1990.

See Also

`Algorithm Properties`

`idss`

`pem`

Purpose

Estimate parameters of output-error model returning `idpoly` object

Syntax

```
m = oe(data,orders)
m = oe(data,'nb',nb,'nf',nf,'nk',nk)
m = oe(data,orders,'Property1',Value1,'Property2',Value2,...)
```

Description

`oe` returns `m` as an `idpoly` object with the resulting parameter estimates, together with estimated covariances. The parameters of the output-error model structure

$$y(t) = \frac{B(q)}{F(q)}u(t-nk) + e(t)$$

are estimated using a prediction error method.

`data` is an `iddata` object containing the output-input data. Both time- and frequency-domain data are supported. Moreover, `data` can be an `frd` or `idfrd` frequency-response data object.

The structure information can either be given explicitly as

```
(...,'nb',nb,'nf',nf,'nk',nk,...)
```

or in the argument `orders`, given as

```
orders = [nb nf nk]
```

The parameters `nb` and `nf` are the orders of the output-error model and `nk` is the delay. Specifically,

$$nb: \qquad B(q) = b_1 + b_2q^{-1} + \ldots + b_{nb}q^{-nb+1}$$

$$nf: \qquad F(q) = 1 + f_1q^{-1} + \ldots + f_{nf}q^{-nf}$$

Alternatively, you can specify the vector as

```
orders = mi
```

where mi is an initial guess at the output-error model given in idpoly format. See “Definition of Polynomial Models” on page 5-43.

For multiinput systems, nb, nf, and nk are row vectors with as many entries as there are input channels. Entry number i then describes the orders and delays associated with the ith input.

Continuous-Time Models

If data is continuous-time (frequency-domain) data, oe estimates a continuous-time model with transfer function

$$G(s) = \frac{B(s)}{F(s)} = \frac{b_{nb}s^{(nb-1)} + b_{nb-1}s^{(nb-2)} + \ldots + b_1}{s^{nf} + f_{nf}s^{(nf-1)} + \ldots + f_1}$$

The orders of the numerator and denominator are thus determined by nb and nf just as in the discrete-time case. However, the delay nk has no meaning and should be omitted. For multiinput systems, nb and nf are row vectors with obvious interpretation.

Properties

The structure and the estimation algorithm are affected by any property name/property value pairs that are set in the input argument list. Useful properties are 'Focus', 'InitialState', 'InputDelay', 'SearchDirection', 'MaxIter', 'Tolerance', 'LimitError', 'FixedParameter', and 'Trace'.

See Algorithm Properties, idpoly, and idmodel for details of these properties and their possible values.

oe does not support multioutput models. Use a state-space model for this case (see n4sid and pem).

Algorithm

oe uses essentially the same algorithm as armax, with modifications to the computation of prediction errors and gradients.

Examples

Suppose fast sampled data (`Ts = 0.001`) is available from a plant with a bandwidth of about 500 rad/s. The data is treated as continuous-time frequency-domain data, and a model of the type

$$G(s) = \frac{b}{s^3 + f_1 s^2 + f_2 s + f_3}$$

is estimated.

```
z = iddata(y,u,0.001);
zf = fft(z);
zf.ts = 0;
m = oe(zf,[1 3],'foc',[0 500])
```

See Also

`Algorithm Properties`

`EstimationInfo`

`idpoly`

`pem`

Purpose

Compute prediction errors associated with model and data set

Syntax

```
e = pe(m,data)
[e,x0] = pe(m,data,init)
```

Description

`data` is the output-input data set, given as an `iddata` object, and `m` is any `idmodel` or `idnlmodel` object. Both time-domain and frequency-domain data are supported, and `data` can also be an `idfrd` object.

`e` is returned as an `iddata` object, so that `e.OutputData` contains the prediction errors that result when model `m` is applied to the data.

$$e(t) = H^{-1}(q)[y(t) - G(q)u(t)]$$

The argument `init` determines how to deal with the initial conditions:

- `init = 'e(stimate)'` means that the initial state is chosen so that the norm of prediction error is minimized. This initial state is returned as `x0`.
- `` init = `d(elayexpand)' ``: Same as `'estimate'`, but for a model with nonzero `InputDelay`, the delays are first converted to explicit model delays (using `inpd2nk`) so that they are contained in x0.
- `init = 'z(ero)'` sets the initial state to zero.
- `init = 'm(odel)'` uses the model's internally stored initial state.
- `init = x0i`, where `x0i` is a column vector of appropriate dimension, uses that value as initial state. For multiexperiment data, `x0i` may be a matrix whose columns give different initial states for each experiment. For a continuous-time model `m`, `x0` is the initial state for this model. Any modifications of the initial state that sampling might require are automatically handled. If `m` has a non-zero `InputDelay`, and you need to access the values of the inputs during this delay, you must first apply `inpd2nk(m)`. If `m` is continuous in time, it must first be sampled before `inpd2nk` can be applied.

If `init` is not specified, the model property `m.InitialState` is used, so that `'Estimate'`, `'Backcast'`, and `'Auto'` set `init = 'Estimate'`, while `m.InitialState = 'Zero'` sets `init = 'zero'`, and `'Fixed'` and `'Model'` set `init = 'model'`.

The output argument `x0` is the value of the initial state used. If `data` contains several experiments, `x0` is a matrix containing the initial states from each experiment.

See Also

`compare`

`predict`

`resid`

`sim`

`simsd`

Purpose

Estimate model parameters using iterative prediction-error minimization method

Syntax

```
m = pem(data)
m = pem(data,mi)
m = pem(data,mi,'Property1',Value1,...,'PropertyN',ValueN)
m = pem(data,orders)
m = pem(data,'P1D')
m = pem(data,'nx',ssorder)
m = pem(data,'na',na,'nb',nb,'nc',nc,'nd',nd,'nf',nf,'nk',nk)
m = pem(data,orders,'Property1',Value1,...,'PropertyN',ValueN)
```

Description

`pem` is the basic estimation command in the toolbox and covers a variety of situations.

`data` is always an `iddata` object that contains the input/output data. Both time-domain and frequency-domain signals are supported. `data` can also be an `frd` or `idfrd` frequency-response data object. Estimation of noise models (`K` in state-space models and `A`, `C`, and `D` in polynomial models) is not supported for frequency-domain data.

With Initial Model

`mi` is any `idmodel` or `idnlmodel` object. It could be a result of another estimation routine, or constructed and modified by the constructors (`idarx`, `idpoly`, `idss`, `idgrey`, `idproc`) and `set`. The properties of `mi` can also be changed by any property name/property value pairs in `pem` as indicated in the syntax.

`m` is then returned as the best fitting model in the model structure defined by `mi`. The iterative search is initialized at the parameters of the initial/nominal model `mi`. `m` will be of the same class as `mi`.

Black-Box State-Space Models

With `m = pem(data,n)`, where `n` is a positive integer, or `m = pem(data,'nx',n)`, a state-space model of order `n` is estimated.

$$x(t+Ts) = Ax(t) + Bu(t) + Ke(t)$$
$$y(t) = Cx(t) + Du(t) + e(t)$$

If `data` is continuous-time (frequency-domain) data, a corresponding continuous-time state space model is estimated.

The default is that it is estimated in a `'Free'` parameterization that can be further modified by the properties `'nk'`, `'DisturbanceModel'`, and `'InitialState'` (see the corresponding reference pages for `idss` and `n4sid`). The model is initialized by `n4sid` and then further adjusted by optimizing the prediction error fit.

You can choose among several different orders by

```
m = pem(data,'nx',[n1,n2,...nN])
```

and you are then prompted for the “best” order. By

```
m = pem(data,'best')
```

an automatic choice of order among 1:10 is made.

```
m = pem(data)
```

is short for `m = pem(data,'best')`. To work with other delays, use, for example, `m = pem(data,'best','nk',[0,...0])`.

In this case `m` is returned as an `idss` model.

Estimating the D, K, and X0 Matrices

Whether the *D* matrix is estimated or not is governed by the property `nk`, which is further described below. The default is that *D* is not estimated. By setting the `k`th entry of `nk` to `0`, the `k`th column of *D* (corresponding to the `k`th input) is estimated. To estimate a full *D* matrix, let `nk = zeros(1,nu)`, as in

```
m = pem(data,order,'nk',[0 .. 0])
```

This holds for both discrete- and continuous-time models.

For frequency-domain data, K is always fixed to `0`. For time-domain data, K is estimated by default. To fix K to `0` in this case, use

```
m = pem(data,order,'DisturbanceModel','none')
```

Similarily, `X0` is estimated if `'InitialState'` is set to `'Estimate'`, and fixed to `0` if `'InitialState'` is set to `'Zero'`.

Black-Box Multiple-Input-Single-Output Models

The function `pem` also handles the general multiple-input-single-output structure

$$A(q)y(t) = \frac{B_1(q)}{F_1(q)}u_1(t-nk_1) + \ldots + \frac{B_{nu}(q)}{F_{nu}(q)}u_{nu}(t-nk_{nu}) + \frac{C(q)}{D(q)}e(t)$$

The orders of this general model are given either as

```
orders = [na nb nc nd nf nk]
```

or with `(...'na',na,'nb',nb,...)` as shown in the syntax. Here `na`, `nb`, `nc`, `nd`, and `nf` are the orders of the model, and `nk` is the delay(s). For multiinput systems, `nb`, `nf`, and `nk` are row vectors giving the orders and delays of each input. (See “Definition of Polynomial Models” on page 5-43 for exact definitions of the orders.) When the orders are specified with separate entries, those not given are taken as zero.

For frequency-domain data, only estimation of `B` and `F` is supported. It is simpler to use `oe` in that case.

In this case `m` is returned as an `idpoly` object.

Continuous-Time Process Models

Entering for the initial model an acronym for a process model, as in

```
m = pem(data,'P2UI')
```

will estimate a continuous-time process model of the indicated type. See the reference page for `idproc` for details of possible model types and associated property name/property value pairs.

In this case `m` is returned as an `idproc` model.

Properties

In all cases the algorithm is affected by the properties (see `Algorithm Properties` for details):

- `Focus`, with possible values `'Prediction'` (default), `'Simulation'`, or a passband range.
- `MaxIter` and `Tolerance` govern the stopping criteria for the iterative search.
- `LimitError` deals with how the criterion can be made less sensitive to outliers and bad data.
- `MaxSize` determines the largest matrix ever formed by the algorithm. The algorithm goes into `for` loops to avoid larger matrices, which can be more efficient than using virtual memory.
- `Trace`, with possible values `'Off'`, `'On'`, and `'Full'`, governs the information sent to the MATLAB Command Window.

For black-box state-space models, `'N4Weight'` and `'N4Horizon'` will also affect the result, since these models are initialized with an `n4sid` estimate. See the reference page for `n4sid`.

Typical `idmodel` properties are (see `idmodel` properties for more details)

- `Ts` is the sampling interval. Set `'Ts'= 0` to obtain a continuous-time state-space model. For discrete-time models, `'Ts'` is automatically set to the sampling interval of the data. Note that, in the black-box case, it is usually better to first estimate a discrete-time model, and then convert that to continuous time using `d2c`.
- `nk` is the time delays from the inputs (not applicable to structured state-space models). Time delays specified by `'nk'` will be included in the model.
- `DisturbanceModel` determines the parameterization of `K` for free and canonical state-space parameterizations, as well as for `idgrey`

models. It also determines whether a noise model should be included for `idproc` models.

- `InitialState`: The initial state can have a substantial influence on the estimation result for systems with slow responses. It is most pronounced for output-error models (`K = 0` for state-space and `na = nc = nd =0` for input/output models). The default value `'Auto'`” estimates the influence of the initial state and sets the value to `'Estimate'`, `'Backcast'`, or `'Zero'` based on this effect. Possible values of `'InitialState'` are `'Auto'`, `'Estimate'`, `'Backcast'`, `'Zero'`, and `'Fixed'`.

Examples

Here is an example of a system with three inputs and two outputs. A canonical form state-space model of order 5 is sought.

```
z = iddata([y1 y2],[ u1 u2 u3]);
m = pem(z,5,'ss','can')
```

Building an ARMAX model for the response to output 2,

```
ma = pem(z(:,2,:),'na',2,'nb',[2 3 1],'nc',2,'nk',[1 2 0])
```

Comparing the models (`compare` automatically matches the channels using the channel names),

```
compare(z,m,ma)
```

Algorithm

`pem` uses essentially the same algorithm as `armax`, with modifications to the computation of prediction errors and gradients.

See Also

`Algorithm Properties`

`EstimationInfo`

`armax`

`bj`

oe

pem

Purpose

Level of excitation of input signals

Syntax

```
Ped = pexcit(Data)
[Ped.Maxnr] = pexcit(Data,Maxnr,Threshold)
```

Description

`Data` is an `iddata` object with time- or frequency-domain signals.

`Ped` is the degree or order of excitation of the inputs in `Data`. A row vector of integers with as many components as there are inputs in `Data`. The intuitive interpretation of the degree of excitation in an input is the order of a model that the input is capable of estimating in an unambiguous way.

`Maxnr` is the maximum order tested. Default is `min(N/3,50)`, where `N` is the number of input data.

`Threshold` is the threshold level used to measure which singular values are significant. Default is `1e-9`.

References

Section 13.2 in Ljung (1999).

See Also

`advice`

`iddata`

Purpose

Plot `iddata` or model objects

Syntax

```
plot(data)
plot(d1,...,dN)
plot(d1,PlotStyle1,...,dN,PlotStyleN)
plot(model)
```

Description

`data` is the output-input data to be graphed, given as an `iddata` object. A split plot is obtained with the outputs on top and the inputs at the bottom.

One plot for each I/O channel combination is produced. Pressing the **Enter** key advances the plot. Typing **Ctrl+C** aborts the plotting in an orderly fashion.

To plot a specific interval, use `plot(data(200:300))`. To plot specific input/output channels, use `plot(data(:,ky,ku))`, consistent with the subreferencing of `iddata` objects.

If `data.intersample = 'zoh'`, the input is piecewise constant between sampling points, and it is then graphed accordingly.

To plot several`iddata` sets `d1,...,dN`, use `plot(d1,...,dN)`. I/O channels with the same experiment name, input name, and output name are always plotted in the same plot.

With `PlotStyle`, the color, line style, and marker of each data set can be specified

```
plot(d1,'y:*',d2,'b')
```

just as in the regular `plot` command.

`model` is an `idmodel`, `idnlarx`, or `idnlhw` object.

See Also

`iddata`

Purpose

Convert model to input-output polynomials

Syntax

```
[A,B,C,D,F] = polydata(m)
[A,B,C,D,F,dA,dB,dC,dD,dF] = polydata(m)
```

Description

This is essentially the inverse of the `idpoly` constructor. It returns the polynomials of the general model

$$A(q)y(t) = \frac{B_1(q)}{F_1(q)}u_1(t-nk_1) + \ldots + \frac{B_{nu}(q)}{F_{nu}(q)}u_{nu}(t-nk_{nu}) + \frac{C(q)}{D(q)}e(t)$$

as contained in the model `m`.

`dA`, `dB`, etc. are the standard deviations of `A`, `B`, etc.

`m` can be any single-output `idmodel`, that is, not just `idpoly`. For multioutput models you can use `[A,B,C,D,F] = polydata(m(ky,:))` to obtain the polynomials for the `ky`th output.

See Also

`idmodel`

`idpoly`

`tfdata`

polyreg

Purpose

Generate custom regressors by computing powers and products of standard regressors

Syntax

```
polyreg(model,'MaxPower',n)
polyreg(model,'MaxPower',n,'CrossTerm','on')
polyreg(model,'MaxPower',n,'CrossTerm','off')
```

Arguments

`model`
: Name of the `idnlarx` model object.

`MaxPower—n`
: Property-value pair specifies the maximum power of the multivariable polynomial in terms of standard regressors of the model.

  Default: `2`.

`CrossTerms—'on'` or `'off'`
: Property-value pair specifies whether to include or exclude cross-terms of the polynomial, or products of standard regressors.

  Default: `'off'`.

Description

`polyreg` is a method of the `idnlarx` model object.

`polyreg(model,'MaxPower',n)` adds one or more custom regressors to nonlinear ARX model `model`. Custom regressors are powers of custom regressors up to the maximum power `n`, but excluding terms of power `1`.

`polyreg(model,'MaxPower',n,'CrossTerm','on')` one or more custom regressors to nonlinear ARX model `model` and includes cross-terms of the polynomial (products of standards regressors).

`polyreg(model,'MaxPower',n,'CrossTerm','off')` one or more custom regressors to nonlinear ARX model `model` and excludes cross-terms of the polynomial (products of standards regressors).

Examples

```
  % Construct a nonlinear ARX model that is
  % linear in the regressors.
```

```
M=idnlarx([2 2 1],'linear')
% Define custom regressors with default settings,
% which include second-order polynomial of standard
% regressors and no cross-terms.
P=polyreg(M)
% Estiomate model using custom regressors
% in the nonlinear block.
M=pem(Data,M,'customreg',P)
```

See Also

`customreg`

`getreg`

`idnlarx`

`polyreg`

predict

Purpose

Predict output k steps ahead

Syntax

```
yp = predict(m,data)
[yp,x0p,mpred] = predict(m,data,k,'InitialState',init)
```

Description

`data` is the output-input data as an `iddata` object, and `m` is any `idmodel` or `idnlmodel` object. `predict` is meaningful only for time-domain data.

The argument `k` indicates that the *k* step-ahead prediction of `y` according to the model `m` is computed. In the calculation of *yp*(*t*), the model can use outputs up to time

$$t - k : y(s), s = t - k, t - k - 1, \ldots$$

and inputs up to the current time *t*. The default value of `k` is `1`.

The output `yp` is an `iddata` object containing the predicted values as `OutputData`.

`x0p` is the used (estimated) initial state vector. For multiexperiment data, `x0p` is a matrix, whose columns contain the initial states for each experiment.

The output argument `mpred` contains the *k* step-ahead predictor. This is given as a cell array, whose *k*th entry is an `idpoly` model for the predictor of output number *k*. Note that these predictor models have as input both input and output signals in the data set. The channel names indicate how the predictor model and the data fit together.

`init` determines how to deal with the initial state:

- `init ='e(stimate)`': The initial state is set to a value that minimizes the norm of the prediction error associated with the model and the data.
- `init = 'd(elayexpand)'`: Same as `'estimate'`, but for a model with nonzero `InputDelay`, the delays are first converted to explicit model delays (using `inpd2nk`) so that they are contained in `x0p`.
- `init = 'z(ero)'` sets the initial state to zero.

- `init = 'm(odel)'` uses the model's internally stored initial state.
- `init = x0`, where `x0` is a column vector of appropriate dimension, uses that value as initial state. For multiexperiment data, `x0` can be a matrix whose columns give different initial states for each experiment. For a continuous-time model `m`, `x0` is the initial state for this model. Any modifications of the initial state that sampling might require are automatically handled. If `m` has a non-zero `InputDelay`, and you need to access the values of the inputs during this delay, you must first apply `inpd2nk(m)`. When `m` is a continuous-time model, it must first be sampled before `inpd2nk` can be applied.

If `init` is not specified, the model property `m.InitialState` is used, so that `'Estimate'`, `'Backcast'`, and `'Auto'` set `init = 'Estimate'`, while `m.InitialState = 'Zero'` sets `init = 'zero'`, and `'Model'` and `'Fixed'` set `init = 'model'`.

An important use of `predict` is to evaluate a model's properties in the mid-frequency range. Simulation with `sim` (which conceptually corresponds to `k = inf`) can lead to levels that drift apart, since the low-frequency behavior is emphasized. One step-ahead prediction is not a powerful test of the model's properties, since the high-frequency behavior is stressed. The trivial predictor $\hat{y}(t) = y(t-1)$ can give good predictions in case the sampling of the data is fast.

Another important use of `predict` is to evaluate time-series models. The natural way of studying a time-series model's ability to reproduce observations is to compare its *k* step-ahead predictions with actual data.

Note that for output-error models, there is no difference between the *k* step-ahead predictions and the simulated output, since, by definition, output-error models only use past inputs to predict future outputs.

Algorithm

The model is evaluated in state-space form, and the state equations are simulated *k* steps ahead with initial value $x(t-k) = \hat{x}(t-k)$, where $\hat{x}(t-k)$ is the Kalman filter state estimate.

Examples

Simulate a time series, estimate a model based on the first half of the data, and evaluate the four step-ahead predictions on the second half.

```
m0 = idpoly([1 -0.99],[],[1 -1 0.2]);
e = iddata([],randn(400,1));
y = sim(m0,e);
m = armax(y(1:200),[1 2]);
yp = predict(m,y,4);
plot(y(201:400),yp(201:400))
```

Note that the last two commands are also achieved by

```
compare(y,m,4,201:400);
```

See Also

`compare`

`pe`

`sim`

`simsd`

Purpose

Display model information, including estimated uncertainty

Syntax

```
present(m)
```

Description

The `present` function displays the model `m`, together with the estimated standard deviations of the parameters, loss function, and Akaike's Final Prediction Error (FPE) Criterion (which essentially equals the AIC). It also displays information about how `m` was created.

`m` is any `idmodel` or `idnlmodel` object.

`present` thus gives more detailed information about the model than the standard `display` function.

Purpose

Store piecewise-linear nonlinear estimator for Hammerstein-Wiener models

Syntax

```
t=pwlinear('NumberOfUnits',N)
t=pwlinear('BreakPoints',BP)
t=pwlinear(Property1,Value1,...PropertyN,ValueN)
```

Description

`pwlinear` is an object that stores the piecewise-linear nonlinear estimator for estimating Hammerstein-Wiener models.

You can use the constructor to create the nonlinearity object, as follows:

`t=pwlinear('NumberOfUnits',N)` creates a piecewise-linear nonlinearity estimator object with `N` breakpoints.

`t=pwlinear('BreakPoints',BP)` creates a piecewise-linear nonlinearity estimator object with breakpoints at values `BP`.

`t=pwlinear(Property1,Value1,...PropertyN,ValueN)` creates a piecewise-linear nonlinearity estimator object specified by properties in "pwlinear Properties" on page 12-257.

Use `evaluate(p,x)` to compute the value of the function defined by the `pwlinear` object `p` at `x`.

Remarks

Use `pwlinear` to define a nonlinear function $y = F(x)$, where F is a piecewise-linear (affine) function of x and there are `n` breakpoints `(x_k,y_k), k=1,...,n. y_k = F(x_k)`. F is linearly interpolated between the breakpoints.

y and x are scalars.

F is also linear to the left and right of the extreme breakpoints. The slope of these extension is a function of `x_i` and `y_i` breakpoints. The breakpoints are ordered by ascending `x`-values, which is important when you set a specific breakpoint to a different value.

There are minor deviations from the breakpoint values you set and the values actually stored in the object because the Toolbox represent breakpoints differently internally.

pwlinear Properties

You can include property-value pairs in the constructor to specify the object.

After creating the object, you can use get or dot notation to access the object property values. For example:

```
% List all property values
get(p)
% Get value of NumberOfUnits property
p.NumberOfUnits
```

You can use dot notation to assign property values to the object. set is not supported for MCOS objects.

For example, the following two commands are equivalent:

```
p.NumberOfUnits=5
```

Property Name	Description
NumberOfUnits	Integer specifies the number of breakpoints Default=10. For example: `pwlinear('NumberOfUnits',5)`
BreakPoints	2-*by*-n matrix containing the breakpoint x and y value, specified using the following format: `[x_1,x_2,...,x_n;y_1,y_2,...,y_n]` If set to a 1-*by*-n vector, the values are interpreted as x-values and the corresponding y-values are set to zero.

Examples

Use pwlinear to specify the piecewise nonlinearity estimator in Hammerstein-Wiener models. For example:

```
m=nlhw(Data,Orders,pwlinear('Br',[-1:0.1:1]),[]);
```

The piecewise nonlinearity is initialized at the specified breakpoints. The breakpoint values are adjusted to the estimation data by `nlhw`.

See Also

`nlhw`

Purpose

Plot zeros and poles with confidence interval

Syntax

```
pzmap(m)
pzmap(m,'sd',sd)
pzmap(m1,m2,m3,...)
pzmap(m1,'PlotStyle1',m2,'PlotStyle2',...,'sd',sd)
pzmap(m1,m2,m3,..,'sd',sd,'mode',mode,'axis',axis)
```

Description

`m` is any `idmodel` object: `idarx`, `idgrey`, `idss`, `idproc`, or `idpoly`.

The zeros and poles of `m` are graphed, with `o` denoting zeros and `x` denoting poles. Poles and zeros at infinity are ignored. For discrete-time models, zeros and poles at the origin are also ignored.

The Property/Value pairs `'sd'/sd, 'mode'/mode` and `` `axis'/axis `` can appear in any order. They are explained below.

If `sd` has a value larger than zero, confidence regions around the poles and zeros are also graphed. The regions corresponding to `sd` standard deviations are marked. The default value is `sd = 0`. Note that the confidence regions might sometimes stretch outside the plot, but they are always symmetric around the indicated zero or pole.

If the poles and zeros are associated with a discrete-time model, a unit circle is also drawn. For continuous-time models, the real and imaginary axes are drawn.

When `mi` contains information about several different input/output channels, you have the following options:

`mode = 'sub'` splits the screen into several plots, one for each input/output channel. These are based on the `InputName` and `OutputName` properties associated with the different models.

`mode = 'same'` gives all plots in the same diagram. Pressing the **Enter** key advances the plots.

`mode = 'sep'` erases the previous plot before the next channel pair is treated.

The default value is `mode = 'sub'`.

`axis = [x1 x2 y1 y2]` fixes the axis scaling accordingly. `axis = s` is the same as

```
axis = [-s s -s s]
```

You can select the colors associated with the different models by using the argument `PlotStyle`. Use `PlotStyle = 'b'`, `'g'`, etc. Markers and line styles are not used.

The noise input channels in `m` are treated as follows: Consider a model `m` with both measured input channels *u* (*nu* channels) and noise channels *e* (*ny* channels) with covariance matrix Λ

$$y = Gu + He$$
$$\mathrm{cov}(e) = \Lambda = LL'$$

where L is a lower triangular matrix. Note that `m.NoiseVariance =` Λ. The model can also be described with a unit variance, normalized noise source v.

$$y = Gu + HLv$$
$$\mathrm{cov}(v) = I$$

Then,

- `pzmap(m)` plots the zeros and poles of the transfer function G.
- `pzmap(m('n'))` plots the zeros and poles of the transfer function H (*ny* inputs and *ny* outputs). The input channels have names `e@yname`, where `yname` is the name of the corresponding output.
- If `m` is a time series, that is *nu* `= 0`, `pzmap(m)` plots the zeros and poles of the transfer function H.
- `pzmap(noisecnv(m))` plots the zeros and poles of the transfer function [*G H*] (*nu*+*ny* inputs and *ny* outputs). The noise input channels have names `e@yname`, where `yname` is the name of the corresponding output.

- `pzmap(noisecnv(m,'norm')` plots the zeros and poles of the transfer function [*G HL*] (*nu*+*ny* inputs and *ny* outputs). The noise input channels have names `v@yname`, where `yname` is the name of the corresponding output.

Examples

```
mbj = bj(data,[2 2 1 1 1]);
mar = armax(data,[2 2 2 1]);
pzmap(mbj,mar,'sd',3)
```

shows all zeros and poles of two models along with the confidence regions corresponding to three standard deviations.

See Also

`idmodel`

`zpkdata`

Purpose Estimate recursively parameters of ARMAX or ARMA models

Syntax

```
thm = rarmax(z,nn,adm,adg)
[thm,yhat,P,phi,psi] = rarmax(z,nn,adm,adg,th0,P0,phi0,psi0)
```

Description The parameters of the ARMAX model structure

$$A(q)y(t) = B(q)u(t-nk) + C(q)e(t)$$

are estimated using a recursive prediction error method.

The input-output data is contained in `z`, which is either an `iddata` object or a matrix `z = [y u]` where `y` and `u` are column vectors. `nn` is given as

```
nn = [na nb nc nk]
```

where `na`, `nb`, and `nc` are the orders of the ARMAX model, and `nk` is the delay. Specifically,

$$na: \quad A(q) = 1 + a_1 q^{-1} + \ldots + a_{na} q^{-na}$$

$$nb: \quad B(q) = b_1 + b_2 q^{-1} + \ldots + b_{nb} q^{-nb+1}$$

$$nc: \quad C(q) = 1 + c_1 q^{-1} + \ldots + c_{nc} q^{-nc}$$

See"Definition of Polynomial Models" on page 5-43 for more information.

If `z` represents a time series `y` and `nn = [na nc]`, `rarmax` estimates the parameters of an ARMA model for `y`.

$$A(q)y(t) = C(q)e(t)$$

Only single-input, single-output models are handled by `rarmax`. Use `rpem` for the multiinput case.

The estimated parameters are returned in the matrix `thm`. The `k`th row of `thm` contains the parameters associated with time `k`; that is, they are based on the data in the rows up to and including row `k` in `z`. Each row of `thm` contains the estimated parameters in the following order:

```
thm(k,:) = [a1,a2,...,ana,b1,...,bnb,c1,...,cnc]
```

`yhat` is the predicted value of the output, according to the current model; that is, row *k* of `yhat` contains the predicted value of `y(k)` based on all past data.

The actual algorithm is selected with the two arguments `adm` and `adg`. These are described under `rarx`.

The input argument `th0` contains the initial value of the parameters, a row vector consistent with the rows of `thm`. The default value of `th0` is all zeros.

The arguments `P0` and `P` are the initial and final values, respectively, of the scaled covariance matrix of the parameters. See `rarx`. The default value of `P0` is 10^4 times the unit matrix. The arguments `phi0`, `psi0`, `phi`, and `psi` contain initial and final values of the data vector and the gradient vector, respectively. The sizes of these depend on the chosen model orders. The normal choice of `phi0` and `psi0` is to use the outputs from a previous call to `rarmax` with the same model orders. (This call could be a dummy call with default input arguments.) The default values of `phi0` and `psi0` are all zeros.

Note that the function requires that the delay `nk` be larger than `0`. If you want `nk = 0`, shift the input sequence appropriately and use `nk = 1`.

Algorithm

The general recursive prediction error algorithm (11.44), (11.47) through (11.49) of Ljung (1999) is implemented. See Chapter 8, “Recursive Parameter Estimation” for more information.

Examples

Compute and plot, as functions of time, the four parameters in a second-order ARMA model of a time series given in the vector `y`. The forgetting factor algorithm with a forgetting factor of 0.98 is applied.

```
thm = rarmax(y,[2 2],'ff',0.98);
plot(thm)
```

Purpose

Estimate recursively parameters of ARX or AR models

Syntax

```
thm = rarx(z,nn,adm,adg)
[thm,yhat,P,phi] = rarx(z,nn,adm,adg,th0,P0,phi0)
```

Description

The parameters of the ARX model structure

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

are estimated using different variants of the recursive least squares method.

The input-output data is contained in `z`, which is either an `iddata` object or a matrix `z = [y u]` where `y` and `u` are column vectors. `nn` is given as

```
nn = [na nb nk]
```

where `na` and `nb` are the orders of the ARX model, and `nk` is the delay. Specifically,

$$na: \qquad A(q) = 1+a_1q^{-1}+\ldots+a_{na}q^{-na}$$

$$nb: \qquad B(q) = b_1+b_2q^{-1}+\ldots+b_{nb}q^{-nb+1}$$

If `z` is a time series `y` and `nn = na`, `rarx` estimates the parameters of an AR model for `y`.

Models with several inputs

$$A(q)y(t) = B_1(q)u_1(t-nk_1)+\ldots B_{nu}u_{nu}(t-nk_{nu})+e(t)$$

are handled by allowing `u` to contain each input as a column vector,

```
u = [u1 ... unu]
```

and by allowing `nb` and `nk` to be row vectors defining the orders and delays associated with each input.

Only single-output models are handled by `rarx`.

The estimated parameters are returned in the matrix thm. The kth row of thm contains the parameters associated with time k; that is, they are based on the data in the rows up to and including row k in z. Each row of thm contains the estimated parameters in the following order.

```
thm(k,:) = [a1,a2,...,ana,b1,...,bnb]
```

In the case of a multiinput model, all the *b* parameters associated with input number 1 are given first, and then all the *b* parameters associated with input number 2, and so on.

yhat is the predicted value of the output, according to the current model; that is, row k of yhat contains the predicted value of y(k) based on all past data.

The actual algorithm is selected with the two arguments adg and adm. These are described in Chapter 8, "Recursive Parameter Estimation". The options are as follows:

- With adm = 'ff' and adg = lam the *forgetting factor* algorithm is obtained with forgetting factor λ= lam. This is what is often referred to as recursive least squares (RLS). In this case the matrix P has the following interpretation: R_2/2 * P is approximately equal to the covariance matrix of the estimated parameters. Here R_2 is the variance of the innovations (the true prediction errors *e*(*t*).
- With adm ='ug' and adg = gam, the *unnormalized gradient* algorithm is obtained with gain *gamma* = gam. This algorithm is commonly known as normalized least mean squares (LMS).
- Similarly, adm ='ng' and adg = gam give the *normalized gradient* or normalized least mean squares (NLMS) algorithm. In these cases, P is not defined or applicable.
- With adm ='kf' and adg = R1, the *Kalman filter based* algorithm is obtained with R_2=1 and R_1 = R1. If the variance of the innovations *e*(*t*) is not unity but R_2; then R_2 * P is the covariance matrix of the parameter estimates, while R_1 = R1 /R_2 is the covariance matrix of the parameter changes.

- The input argument `th0` contains the initial value of the parameters, a row vector consistent with the rows of `thm`. The default value of `th0` is all zeros.
- The arguments `P0` and `P` are the initial and final values, respectively, of the scaled covariance matrix of the parameters. The default value of `P0` is 10^4 times the identity matrix.
- The arguments `phi0` and `phi` contain initial and final values, respectively, of the data vector.

$$\varphi(t) = [y(t-1), \ldots, y(t-na), u(t-1), \ldots u(t-nb-nk+1)]$$

Then, if

```
z = [y(1),u(1); ... ;y(N),u(N)]
```

you have `phi0=` $\varphi(1)$ and `phi=` $\varphi(N)$. The default value of `phi0` is all zeros. For online use of `rarx`, use `phi0`, `th0`, and `P0` as the previous outputs `phi`, `thm` (last row), and `P`.

Note that the function requires that the delay `nk` be larger than `0`. If you want `nk = 0`, shift the input sequence appropriately and use `nk = 1`. See `nkshift`.

Examples

Adaptive noise canceling: The signal *y* contains a component that has its origin in a known signal *r*. Remove this component by estimating, recursively, the system that relates *r* to *y* using a sixth-order FIR model together with the NLMS algorithm.

```
z = [y r];
[thm,noise] = rarx(z,[0 6 1],'ng',0.1);
% noise is the adaptive estimate of the noise
% component of y
plot(y-noise)
```

If the above application is a true online one, so that you want to plot the best estimate of the signal `y - noise` at the same time as the data *y* and *u* become available, proceed as follows.

```
phi = zeros(6,1); P=1000*eye(6);
th = zeros(1,6); axis([0 100 -2 2]);
plot(0,0,'*'), hold on
% The loop:
while ~abort
[y,r,abort] = readAD(time);
[th,ns,P,phi] = rarx([y r],'ff',0.98,th,P,phi);
plot(time,y-ns,'*')
time = time +Dt
end
```

This example uses a forgetting factor algorithm with a forgetting factor of 0.98. readAD represents an M-file that reads the value of an A/D converter at the indicated time instant.

See Also

nkshift

rarmax

rbj

roe

rpem

rplr

Purpose

Estimate recursively parameters of Box-Jenkins models

Syntax

```
thm = rbj(z,nn,adm,adg)
[thm,yhat,P,phi,psi] = ... rbj(z,nn,adm,adg,th0,P0,phi0,psi0)
```

Description

The parameters of the Box-Jenkins model structure

$$y(t) = \frac{B(q)}{F(q)}u(t-nk) + \frac{C(q)}{D(q)}e(t)$$

are estimated using a recursive prediction error method.

The input-output data is contained in `z`, which is either an `iddata` object or a matrix `z = [y u]` where `y` and `u` are column vectors. `nn` is given as

```
nn = [nb nc nd nf nk]
```

where `nb`, `nc`, `nd`, and `nf` are the orders of the Box-Jenkins model, and `nk` is the delay. Specifically,

$$nb: \quad B(q) = b_1 + b_2 q^{-1} + \ldots + b_{nb} q^{-nb+1}$$

$$nc: \quad C(q) = 1 + c_1 q^{-1} + \ldots + c_{nc} q^{-nc}$$

$$nd: \quad D(q) = 1 + d_1 q^{-1} + \ldots + d_{nd} q^{-nd}$$

$$nf: \quad F(q) = 1 + f_1 q^{-1} + \ldots + f_{nf} q^{-nf}$$

See"Definition of Polynomial Models" on page 5-43 for more information.

Only single-input, single-output models are handled by `rbj`. Use `rpem` for the multiinput case.

The estimated parameters are returned in the matrix `thm`. The `k`th row of `thm` contains the parameters associated with time `k`; that is, they are based on the data in the rows up to and including row `k` in `z`. Each row of `thm` contains the estimated parameters in the following order.

```
thm(k,:) = [b1,...,bnb,c1,...,cnc,d1,...,dnd,f1,...,fnf]
```

`yhat` is the predicted value of the output, according to the current model; that is, row `k` of `yhat` contains the predicted value of `y(k)` based on all past data.

The actual algorithm is selected with the two arguments `adm` and `adg`. These are described under `rarx`.

The input argument `th0` contains the initial value of the parameters, a row vector consistent with the rows of `thm`. The default value of `th0` is all zeros.

The arguments `P0` and `P` are the initial and final values, respectively, of the scaled covariance matrix of the parameters. See `rarx`. The default value of `P0` is 10^4 times the unit matrix. The arguments `phi0`, `psi0`, `phi`, and `psi` contain initial and final values of the data vector and the gradient vector, respectively. The sizes of these depend on the chosen model orders. The normal choice of `phi0` and `psi0` is to use the outputs from a previous call to `rbj` with the same model orders. (This call could be a dummy call with default input arguments.) The default values of `phi0` and `psi0` are all zeros.

Note that the function requires that the delay `nk` be larger than `0`. If you want `nk = 0`, shift the input sequence appropriately and use `nk = 1`.

Algorithm

The general recursive prediction error algorithm (11.44) of Ljung (1900) is implemented. See also Chapter 8, “Recursive Parameter Estimation”.

See Also

`nkshift`

`rarmax`

`rarx`

`roe`

`rpem`

`rplr`

Purpose

Determine whether iddata is based on real-valued signals

Syntax

```
realdata(data)
```

Description

realdata returns 1 if

- data contains only real-valued signals.
- data contains frequency-domain signals, obtained by Fourier transformation of real-valued signals.

Otherwise realdata returns 0.

Notice the difference with isreal:

```
load iddata1
isreal(z1); % returns 1
zf = fft(z1);
isreal(zf) % returns 0
realdata(zf) % returns 1
zf = complex(zf) % adds negative frequencies to zf
realdata(zf) % still returns 1
```

Purpose

Resample data by interpolation and decimation

Syntax

```
datar = resample(data,P,Q)
datar = resample(data,P,Q,,filter_order)
```

Description

`data`: The data to be resampled, given as an `iddata` object

`datar`: The resampled data returned as an `iddata` object

P, Q: Integers that determine the resampling factor. The new sampling interval will be `Q/P` times the original one, so `resample(z,1,Q)` means decimation with a factor `Q`.

`filter_order`: Determines the order of the presampling filters used before interpolation and decimation. Default is `10`.

Algorithm

If the Signal Processing Toolbox is available, the resampling is achieved by calls to the `resample` function in that toolbox. The intersample character of the input, as described by `data.InterSample`, is taken into account.

Otherwise, use the function `datar = idresamp(data,R)`, where `R=Q/P`. Then the data is interpolated by a factor `P` and then decimated by a factor `Q`. The interpolation and decimation are preceded by prefiltering, and follow the same algorithms as in the routines `interp` and `decimate` in the Signal Processing Toolbox.

Examples

Resample by increasing the sampling rate by a factor of 1.5 and compare the signals.

```
plot(u)
ur = resample(u,3,2);
plot(u,ur)
```

Purpose

Compute and test model residuals (prediction errors)

Syntax

```
resid(m,data)
resid(m,data,Type)
resid(m,data,Type,M)
e = resid(m,data);
```

Description

`data` contains the output-input data as an `iddata` object. Both time-domain and frequency-domain data are supported. `data` can also be an `idfrd` object.

`m` is any `idmodel` or `idnlmodel` object.

In all cases the residuals *e* associated with the data and the model are computed. This is done as in the command `pe` with a default choice of `init`.

When called without output arguments, `resid` produces a plot. The plot can be of three kinds depending on the argument `Type`:

- `Type = 'Corr'` (only available for time-domain data): The autocorrelation function of `e` and the cross correlation between `e` and the input(s) `u` are computed and displayed. The 99% confidence intervals for these values are also computed and shown as a yellow region. The computation of the confidence region is done assuming `e` to be white and independent of `u`. The functions are displayed up to lag `M`, which is 25 by default.
- `Type = 'ir'`: The impulse response (up to lag `M`, which is 25 by default) from the input to the residuals is plotted with a 99% confidence region around zero marked as a yellow area. Negative lags up to `M/4` are also included to investigate feedback effects. (The result is the same as `impulse(e,'sd',2.58,',M))`.
- `Type = 'fr'`: The frequency response from the input to the residuals (based on a high-order FIR model) is shown as a Bode plot. A 99% confidence region around zero is also marked as a yellow area.

The default for time-domain data is Type = 'Corr'. For frequency-domain data, the default is Type = 'fr'.

With an output argument, no plot is produced, and e is returned with the residuals (prediction errors) associated with the model and the data. It is an iddata object with the residuals as outputs and the input in data as inputs. That means that e can be directly used to build model error models, that is, models that describe the dynamics from the input to the residuals (which should be negligible if m is a good description of the system).

Examples

Here are some typical model validation commands.

```
e = resid(m,data);
plot(e)
compare(data,m);
```

To compute a model error model, that is, a model to input to the residuals to see if any essential unmodeled dynamics are left, do the following:

```
e = resid(m,data);
me = arx(e,[10 10 0]);
bode(me,'sd',3,fill')
```

References

Ljung (1999), Section 16.6.

See Also

compare

predict

sim

simsd

Purpose

Estimate recursively output-error models (IIR-filters)

Syntax

```
thm = roe(z,nn,adm,adg)
[thm,yhat,P,phi,psi] = roe(z,nn,adm,adg,th0,P0,phi0,psi0)
```

Description

The parameters of the output-error model structure

$$y(t) = \frac{B(q)}{F(q)}u(t-nk)+e(t)$$

are estimated using a recursive prediction error method.

The input-output data are contained in `z`, which is either an `iddata` object or a matrix `z = [y u]` where `y` and `u` are column vectors. `nn` is given as

```
nn = [nb nf nk]
```

where `nb` and `nf` are the orders of the output-error model, and `nk` is the delay. Specifically,

$$nb: \qquad B(q) = b_1+b_2q^{-1}+\ldots+b_{nb}q^{-nb+1}$$

$$nf: \qquad F(q) = 1+f_1q^{-1}+\ldots+f_{nf}q^{-nf}$$

See“Definition of Polynomial Models” on page 5-43 for more information.

Only single-input, single-output models are handled by `roe`. Use `rpem` for the multiinput case.

The estimated parameters are returned in the matrix `thm`. The `k`th row of `thm` contains the parameters associated with time `k`; that is, they are based on the data in the rows up to and including row `k` in `z`.

Each row of `thm` contains the estimated parameters in the following order.

```
thm(k,:) = [b1,...,bnb,f1,...,fnf]
```

`yhat` is the predicted value of the output, according to the current model; that is, row `k` of `yhat` contains the predicted value of `y(k)` based on all past data.

The actual algorithm is selected with the two arguments `adg` and `adm`. These are described under `rarx`.

The input argument `th0` contains the initial value of the parameters, a row vector consistent with the rows of `thm`. The default value of `th0` is all zeros.

The arguments `P0` and `P` are the initial and final values, respectively, of the scaled covariance matrix of the parameters. See `rarx`. The default value of `P0` is 10^4 times the unit matrix. The arguments `phi0`, `psi0`, `phi`, and `psi` contain initial and final values of the data vector and the gradient vector, respectively. The sizes of these depend on the chosen model orders. The normal choice of `phi0` and `psi0` is to use the outputs from a previous call to `roe` with the same model orders. (This call could be a dummy call with default input arguments.) The default values of `phi0` and `psi0` are all zeros.

Note that the function requires that the delay `nk` be larger than `0`. If you want `nk = 0`, shift the input sequence appropriately and use `nk = 1`.

Algorithm

The general recursive prediction error algorithm (11.44) of Ljung (1999) is implemented. See also Chapter 8, “Recursive Parameter Estimation”.

See Also

`nkshift`

`rarmax`

`rarx`

`rbj`

`rpem`

`rplr`

Purpose

Estimate general input-output models using recursive prediction-error minimization method

Syntax

```
thm = rpem(z,nn,adm,adg)
[thm,yhat,P,phi,psi] = rpem(z,nn,adm,adg,th0,P0,phi0,psi0)
```

Description

The parameters of the general linear model structure

$$A(q)y(t) = \frac{B_1(q)}{F_1(q)}u_1(t-nk_1) + \ldots + \frac{B_{nu}(q)}{F_{nu}(q)}u_{nu}(t-nk_{nu}) + \frac{C(q)}{D(q)}e(t)$$

are estimated using a recursive prediction error method.

The input-output data is contained in `z`, which is either an `iddata` object or a matrix `z = [y u]` where `y` and `u` are column vectors. (In the multiinput case, `u` contains one column for each input.) `nn` is given as

```
nn = [na nb nc nd nf nk]
```

where `na`, `nb`, `nc`, `nd`, and `nf` are the orders of the model, and `nk` is the delay. For multiinput systems, `nb`, `nf`, and `nk` are row vectors giving the orders and delays of each input. See"Definition of Polynomial Models" on page 5-43 for an exact definition of the orders.

The estimated parameters are returned in the matrix `thm`. The `k`th row of `thm` contains the parameters associated with time `k`; that is, they are based on the data in the rows up to and including row `k` in `z`. Each row of `thm` contains the estimated parameters in the following order.

```
thm(k,:) = [a1,a2,...,ana,b1,...,bnb,...
  c1,...,cnc,d1,...,dnd,f1,...,fnf]
```

For multiinput systems, the *B* part in the above expression is repeated for each input before the *C* part begins, and the *F* part is also repeated for each input. This is the same ordering as in `m.par`.

`yhat` is the predicted value of the output, according to the current model; that is, row `k` of `yhat` contains the predicted value of `y(k)` based on all past data.

The actual algorithm is selected with the two arguments `adg` and `adm`. These are described under `rarx`.

The input argument `th0` contains the initial value of the parameters, a row vector consistent with the rows of `thm`. The default value of `th0` is all zeros.

The arguments `P0` and `P` are the initial and final values, respectively, of the scaled covariance matrix of the parameters. See `rarx`. The default value of `P0` is 10^4 times the unit matrix. The arguments `phi0`, `psi0`, `phi`, and `psi` contain initial and final values of the data vector and the gradient vector, respectively. The sizes of these depend on the chosen model orders. The normal choice of `phi0` and `psi0` is to use the outputs from a previous call to `rpem` with the same model orders. (This call could be a dummy call with default input arguments.) The default values of `phi0` and `psi0` are all zeros.

Note that the function requires that the delay `nk` be larger than `0`. If you want `nk = 0`, shift the input sequence appropriately and use `nk = 1`.

Algorithm

The general recursive prediction error algorithm (11.44) of Ljung (1999) is implemented. See also Chapter 8, “Recursive Parameter Estimation”.

For the special cases of ARX/AR models, and of single-input ARMAX/ARMA, Box-Jenkins, and output-error models, it is more efficient to use `rarx`, `rarmax`, `rbj`, and `roe`.

See Also

`nkshift`

`rarmax`

`rarx`

`rbj`

`roe`

`rplr`

Purpose

Estimate general input-output models using recursive pseudolinear regression method

Syntax

```
thm = rplr(z,nn,adm,adg)
[thm,yhat,P,phi] = rplr(z,nn,adm,adg,th0,P0,phi0)
```

Description

This is a direct alternative to `rpem` and has essentially the same syntax. See `rpem` for an explanation of the input and output arguments.

`rplr` differs from `rpem` in that it uses another gradient approximation. See Section 11.5 in Ljung (1999). Several of the special cases are well-known algorithms.

When applied to ARMAX models, (`nn = [na nb nc 0 0 nk]`), `rplr` gives the extended least squares (ELS) method. When applied to the output-error structure (`nn = [0 nb 0 0 nf nk]`), the method is known as HARF in the `adm = 'ff'` case and SHARF in the `adm = 'ng'` case.

`rplr` can also be applied to the time-series case in which an ARMA model is estimated with

```
z = y
nn = [na nc]
```

You can thus use `rplr` as an alternative to `rarmax` for this case.

See Also

`nkshift`

`rarmax`

`rarx`

`rbj`

`roe`

`rpem`

Purpose

Store saturation nonlinearity estimator for Hammerstein-Wiener models

Syntax

```
s=saturation(LinearInterval,L)
```

Description

`saturation` is an object that stores the saturation nonlinearity estimator for estimating Hammerstein-Wiener models.

You can use the constructor to create the nonlinearity object, as follows:

`s=saturation(LinearInterval,L)` creates a saturation nonlinearity estimator object, initialized with the linear interval `L`.

Use `evaluate(s,x)` to compute the value of the function defined by the `saturation` object `s` at `x`.

Remarks

Use `saturation` to define a nonlinear function $y = F(x)$, where F is a function of x and has the following characteristics:

$$\begin{array}{ll} a \le x < b & F(x) = x \\ a > x & F(x) = a \\ b \le x & F(x) = b \end{array}$$

y and x are scalars.

saturation Properties

You can specify the property value as an argument in the constructor to specify the object.

After creating the object, you can use `get` or dot notation to access the object property values. For example:

```
% List LinearInterval property value
get(s)
s.LinearInterval
```

You can use dot notation to assign property values to the object. `set` is not supported for MCOS objects.

For example, the following two commands are equivalent:

```
s.LinearInterval=[-1 1]
```

Property Name	Description
`LinearInterval`	1-*by*-2 row vector that specifies the initial interval of the saturation Default=`[NaN NaN]`. For example: `saturation('LinearInterval',[-1.5 1.5])`

Examples

Use `saturation` to specify the saturation nonlinearity estimator in Hammerstein-Wiener models. For example:

```
m=nlhw(Data,Orders,saturation([-1 1]),[]);
```

The saturation nonlinearity is initialized at the interval `[-1 1]`. The interval values are adjusted to the estimation data by `nlhw`.

See Also

`nlhw`

Purpose

Segment data and estimate models for each segment

Syntax

```
segm = segment(z,nn)
[segm,V,thm,R2e] = segment(z,nn,R2,q,R1,M,th0,P0,ll,mu)
```

Description

`segment` builds models of AR, ARX, or ARMAX/ARMA type,

$$A(q)y(t) = B(q)u(t-nk)+C(q)e(t)$$

assuming that the model parameters are piecewise constant over time. It results in a model that has split the data record into segments over which the model remains constant. The function models signals and systems that might undergo abrupt changes.

The input-output data is contained in `z`, which is either an `iddata` object or a matrix `z = [y u]` where `y` and `u` are column vectors. If the system has several inputs, `u` has the corresponding number of columns.

The argument `nn` defines the model order. For the ARMAX model

```
nn = [na nb nc nk]
```

where `na`, `nb`, and `nc` are the orders of the corresponding polynomials. See "Definition of Polynomial Models" on page 5-43. Moreover, `nk` is the delay. If the model has several inputs, `nb` and `nk` are row vectors, giving the orders and delays for each input.

For an ARX model (`nc = 0`) enter

```
nn = [na nb nk]
```

For an ARMA model of a time series

```
z = y
nn = [na nc]
```

and for an AR model

```
nn = na
```

The output argument segm is a matrix, whose k row contains the parameters corresponding to time k. This is analogous to the output argument thm in rarx and rarmax. The output argument thm of segment contains the corresponding model parameters that have not yet been segmented. That is, they are not piecewise constant, and therefore correspond to the outputs of the functions rarmax and rarx. In fact, segment is an alternative to these two algorithms, and has a better capability to deal with time variations that might be abrupt.

The output argument V contains the sum of the squared prediction errors of the segmented model. It is a measure of how successful the segmentation has been.

The input argument R2 is the assumed variance of the innovations $e(t)$ in the model. The default value of R2, R2 = [], is that it is estimated. Then the output argument R2e is a vector whose kth element contains the estimate of R2 at time k.

The argument q is the probability that the model undergoes at an abrupt change at any given time. The default value is 0.01.

R1 is the assumed covariance matrix of the parameter jumps when they occur. The default value is the identity matrix with dimension equal to the number of estimated parameters.

M is the number of parallel models used in the algorithm (see below). Its default value is 5.

th0 is the initial value of the parameters. Its default is zero. P0 is the initial covariance matrix of the parameters. The default is 10 times the identity matrix.

ll is the guaranteed life of each of the models. That is, any created candidate model is not abolished until after at least ll time steps. The default is ll = 1. Mu is a forgetting parameter that is used in the scheme that estimates R2. The default is 0.97.

The most critical parameter for you to choose is R2. It is usually more robust to have a reasonable guess of R2 than to estimate it. Typically, you need to try different values of R2 and evaluate the results. (See the example below.) sqrt(R2) corresponds to a change in the value $y(t)$

that is normal, giving no indication that the system or the input might have changed.

Algorithm

The algorithm is based on `M` parallel models, each recursively estimated by an algorithm of Kalman filter type. Each is updated independently, and its posterior probability is computed. The time-varying estimate `thm` is formed by weighting together the `M` different models with weights equal to their posterior probability. At each time step the model (among those that have lived at least `ll` samples) that has the lowest posterior probability is abolished. A new model is started, assuming that the system parameters have jumped, with probability `q`, a random jump from the most likely among the models. The covariance matrix of the parameter change is set to `R1`.

After all the data are examined, the surviving model with the highest posterior probability is tracked back and the time instances where it jumped are marked. This defines the different segments of the data. (If no models had been abolished in the algorithm, this would have been the maximum likelihood estimates of the jump instances.) The segmented model `segm` is then formed by smoothing the parameter estimate, assuming that the jump instances are correct. In other words, the last estimate over a segment is chosen to represent the whole segment.

Examples

Check how the algorithm segments a sinusoid into segments of constant levels. Then use a very simple model $y(t) = b_1 * 1$, where `1` is a fake input and b`1` describes the piecewise constant level of the signal $y(t)$ (which is simulated as a sinusoid).

```
y = sin([1:50]/3)';
thm = segment([y,ones(size(y))],[0 1 1],0.1);
plot([thm,y])
```

By trying various values of `R2` (`0.1` in the above example), more levels are created as `R2` decreases.

selstruc

Purpose

Select model order (structure)

Syntax

```
nn = selstruc(v)
[nn,vmod] = selstruc(v,c)
```

Description

`selstruc` is a function to help choose a model structure (order) from the information contained in the matrix `v` obtained as the output from `arxstruc` or `ivstruc`.

The default value of `c` is `'plot'`. The plot shows the percentage of the output variance that is not explained by the model as a function of the number of parameters used. Each value shows the best fit for that number of parameters. By clicking in the plot you can examine which orders are of interest. When you click `'Select'`, the variable `nn` is returned in the workspace as the optimal model structure for your choice of number of parameters. Several choices can be made.

`c = 'aic'` gives no plots, but returns in `nn` the structure that minimizes Akaike's Information Criterion (AIC),

$$V_{mod} = V\left(1 + \frac{2d}{N}\right)$$

where V is the loss function, d is the total number of parameters in the structure in question, and N is the number of data points used for the estimation. See `aic` for more details.

`c = 'mdl'` returns in `nn` the structure that minimizes Rissanen's Minimum Description Length (MDL) criterion.

$$V_{mod} = V\left(1 + \frac{d\log(N)}{N}\right)$$

When `c` equals a numerical value, the structure that minimizes

$$V_{mod} = V\left(1 + \frac{cd}{N}\right)$$

is selected.

The output argument vmod has the same format as v, but it contains the logarithms of the accordingly modified criteria.

Examples

```
V = arxstruc(data(1:200),data(201:400),...
             struc(1:10,1:10,1:10))
nn = selstruc(V,0); %best fit to validation data
m = arx(data,nn)
```

Purpose

Set properties of data and model objects

Syntax

```
set(m,'Property',Value)
set(m,'Property1',Value1,...'PropertyN',ValueN)
set(m,'Property')
set(m)
```

Description

`set` is used to set or modify the properties of any of the objects in the toolbox (`iddata`, `idmodel`, `idgrey`, `idarx`, `idpoly`, `idss`, `idnlgrey`, `idnlarx`, `idnlhw`). See the corresponding reference pages for a complete list of properties.

`set(m,'Property',Value)` assigns the value `Value` to the property of the object `m` specified by the string `'Property'`. This string can be the full property name (for example, `'SSParameterization'`) or any unambiguous case-insensitive abbreviation (for example, `'ss'`).

`set(m,'Property1',Value1,...'PropertyN',ValueN)` sets multiple properties with a single statement. In certain cases this might be necessary, since the model `m` must, for example, have state-space matrices of consistent dimensions after each `set` statement.

`set(m,'Property')` displays admissible values for the property specified by `'Property'`.

`set(m)` displays all assignable values of `m` and their admissible values.

The same result is also obtained by subassignment.

```
   m.Property = Value
```

Purpose

Set initial states of `idnlgrey` model object

Syntax

```
setinit(model)
setinit(model,prop,values)
```

Arguments

`model`
: Name of the `idnlgrey` model object.

`Property`
: Name of the `InitialStates` model property field, such as `'Name'`, `'Unit'`, `'Value'`, `'Minimum'`, `'Maximum'`, and `'Fixed'`.

`Values`
: Values of the specified property `Property`. `Values` are an `Nx`-by-1 cell array of values, where `Nx` is the number of states.

Description

`setinit(model)` sets the initial-state values in the `'Value'` field of the `InitialStates` model property.

`setinit(model,prop,values)` sets the `values` of the `prop` field of the `InitialStates` model property. `prop` can be `'Name'`, `'Unit'`, `'Value'`, `'Minimum'`, `'Maximum'`, and `'Fixed'`.

See Also

`getinit`

`getpar`

`idnlgrey`

`setpar`

Purpose

Set initial parameter values of `idnlgrey` model object

Syntax

```
setpar(model)
setpar(model,prop)
```

Arguments

`model`
: Name of the `idnlgrey` model object.

`Property`
: Name of the `Parameters` model property field, such as `'Name'`, `'Unit'`, `'Value'`, `'Minimum'`, or `'Maximum'`.

  Default: `'Value'`.

Description

`setpar(model)` sets the initial model parameter values in the `'Value'` field of the `Parameters` model property.

`setpar(model,prop)` sets the model parameter values in the `prop` field of the `Parameters` model property. `prop` can be `'Name'`, `'Unit'`, `'Value'`, `'Minimum'`, and `'Maximum'`.

See Also

`getinit`

`getpar`

`idnlgrey`

`setinit`

Purpose

Set mnemonic parameter names for black-box model structures

Syntax

```
model = setpname(model)
```

Description

`model` is an `idmodel` object of `idarx`, `idpoly`, `idproc`, or `idss` type. The returned model has the `'PName'` property set to a cell array of strings that correspond to the symbols used in this manual to describe the parameters.

For single-input `idpoly` models, the parameters are called `'a1', 'a2', ...,'fn'`.

For multiinput `idpoly` models, the *b* and *f* parameters have the output/input channel number in parentheses, as in `'b1(1,2)'`, `'f3(1,2)'`, etc.

For `idarx` models, the parameter names are as in `'-A(ky,ku)'` for the negative value of the *ky*-*ku* entry of the matrix in *A(q)* polynomial of the multioutput ARX equation, and similarly for the *B* parameters.

For `idss` models, the parameters are named for the matrix entries they represent, such as `'A(4,5)'`, `'K(2,3)'`, etc.

For `idproc` models, the parameter names are as described under `idproc`.

This function is particularly useful when certain parameters are to be fixed. See the property `FixedParameter` under `Algorithm Properties`.

Purpose

Set matrix structure for `idss` objects

Syntax

`setstruc(M,As,Bs,Cs,Ds.Ks,X0s)setstruc(M,Mods)`

Description

```
setstruc(M,As,Bs,Cs,Ds.Ks,X0s)
```

is the same as

```
set(M,'As',As,'Bs',Bs,'Cs',Cs,'Ds',Ds,'Ks',Ks,'X0s',X0s)
```

Use empty matrices for structure matrices that should not be changed. You can omit trailing arguments.

In the alternative syntax, `Mods` is a structure with fieldnames `As`, `Bs`, etc., with the corresponding values of the fields.

See Also

`idss`

Purpose

Store sigmoid network nonlinearity estimator for nonlinear ARX and Hammerstein-Wiener models

Syntax

```
s=sigmoidnet('NumberOfUnits',N)
s=sigmoidnet(Property1,Value1,...PropertyN,ValueN)
```

Description

`sigmoidnet` is an object that stores the sigmoid network nonlinear estimator for estimating nonlinear ARX and Hammerstein-Wiener models.

You can use the constructor to create the nonlinearity object, as follows:

`s=sigmoidnet('NumberOfUnits',N)` creates a sigmoid nonlinearity estimator object with `N` terms in the sigmoid expansion.

`s=sigmoidnet(Property1,Value1,...PropertyN,ValueN)` creates a sigmoid nonlinearity estimator object specified by properties in "sigmoidnet Properties" on page 12-292.

Use `evaluate(s,x)` to compute the value of the function defined by the `sigmoidnet` object `s` at `x`.

Remarks

Use `sigmoidnet` to define a nonlinear function $y = F(x)$, where y is scalar and x is an `m`-dimensional row vector. The sigmoid network function is based on the following expansion:

$$F(x) = (x-r)PL + a_1 f\left((x-r)Qb_1 - c_1\right) + \ldots$$
$$+ a_n f\left((x-r)Qb_n - c_n\right) + d$$

where f is the sigmoid function, given by the following equation:

$$f(z) = \frac{1}{e^{-z}+1}$$

P and Q are `m`-*by*-`p` and `m`-*by*-`q` projection matrices. The projection matrices P and Q are determined by principal component analysis of estimation data. Usually, `p=m`. If the components of x in the estimation data are linearly dependent, then `p<m`. The number of columns of Q,

`q`, corresponds to the number of components of `x` used in the sigmoid function.

When used in a nonlinear ARX model, `q` is equal to the size of the `NonlinearRegressors` property of the `idnlarx` object. When used in a Hammerstein-Wiener model, `m=q=1` and Q is a scalar.

r is a 1-*by*-`m` vector and represents the mean value of the regressor vector computed from estimation data.

d, a_k, and c_k are scalars.

L is `p`-*by*-1 vector.

b_k are `q`-*by*-1 vectors.

sigmoidnet Properties

You can include property-value pairs in the constructor to specify the object.

After creating the object, you can use `get` or dot notation to access the object property values. For example:

```
% List all property values
get(s)
% Get value of NumberOfUnits property
s.NumberOfUnits
```

You can use dot notation to assign property values to the object. `set` is not supported for MCOS objects.

For example, the following two commands are equivalent:

```
s.NumberOfUnits=5
```

The `Parameters` property is a structure. Typically, the values of this structure are set by estimating a model with a `sigmoidnet` nonlinearity. If you need to set the values of this structure, you can use the following syntax:

```
X=struct('RegressorMean',r,
         'NonLinearSubspace',P,
          'LinearSubspace',Q,
          'LinearCoef',L,
          'Dilation',b_k,
          'Translation',c_k,
          'OutputCoef',a_k,
          'OutputOffset',d);
s.Parameters=X;
```

Property Name	Description
`NumberOfUnits`	Integer specifies the number of nonlinearity units in the expansion. Default=10. For example: `sigmoidnet(H,'NumberOfUnits',5)`

Property Name	Description
`LinearTerm`	Can have the following values: • `'on'`—Estimates the vector L in the expansion. • `'off'`—Fixes the vector L to zero. For example: `sigmoidnet(H,'LinearTerm','on')`
`Parameters`	A structure containing the parameters in the nonlinear expansion, as follows: • `RegressorMean`: 1-*by*-`m` vector containing the means of `x` in estimation data, `r`. • `NonLinearSubspace`: `m`-*by*-`q` matrix containing Q. • `LinearSubspace`: `m`-*by*-`p` matrix containing P. • `LinearCoef`: `p`-*by*-1 vector L. • `Dilation`: `q`-*by*-`n` matrix containing the values `b_k`. • `Translation`: 1-*by*-`n` vector containing the values `c_k`. • `OutputCoef`: `n`-*by*-1 vector containing the values `a_k`. • `OutputOffset`: scalar `d`.

Examples

Use `sigmoidnet` to specify the nonlinear estimator in nonlinear ARX and Hammerstein-Wiener models. For example:

```
m=nlarx(Data,Orders,sigmoidnet('num',5));
```

See Also

`nlarx`

`nlhw`

Purpose

Simulate linear models with confidence interval

Syntax

```
y = sim(m,u)
y = sim(m,u,'noise')
[y, ysd] = sim(m,u,'InitialState',init)
```

Description

`m` is any `idmodel` or `idnlmodel` object.

`u` is an `iddata` object, containing inputs only. (Any outputs are ignored). Both time-domain and frequency-domain signals are supported. The number of input channels in `u` must either be equal to the number of inputs of the model `m` or equal to the sum of the number of inputs and noise sources (number of outputs). In the latter case the last inputs in `u` are regarded as noise sources and a noise-corrupted simulation is obtained. The noise is scaled according to the property `m.NoiseVariance` in `m`. To obtain the right noise level according to the model, the noise inputs should be white noise with zero mean and unit covariance matrix. A simpler way of obtaining a noise-corrupted simulation with Gaussian noise is to add the argument `'noise'`. If no noise sources are contained in `u`, a noise-free simulation is obtained. `sim` applies both to time-domain and frequency-domain `iddata` objects, but no standard deviations are obtained for frequency-domain signals.

`sim` returns `y`, containing the simulated output, as an `iddata` object.

`init` gives access to the initial states:

- `init = 'm'` (default) uses the internally stored initial state of model `m`.
- `init = 'z'` uses zero initial state.
- `init = x0`, where `x0` is a column vector of appropriate length, uses this value as the initial state. For multi-experiment inputs, `x0` has as many columns as there are experiments to allow for different initial conditions. For a continuous-time model `m`, `x0` is the initial state for this model. Any modifications of the initial state that sampling might require are automatically handled. If `m` has a non-zero `InputDelay`,

and you need to access the values of the inputs during this delay, you must first apply `inpd2nk(m)`. If `m` is a continuous-time model, it must first be sampled before `inpd2nk` can be applied.

The second output argument `ysd` is the standard deviation of the simulated output. This is not available for frequency-domain data.

`u` can also be given as a matrix with the number of columns being either the number of inputs in `m` or the sum of the number of inputs and outputs. Then `y` and `ysd` are returned as matrices. Continuous-time models, however, require `u` to be given as `iddata`.

If `m` is a continuous-time model, it is first converted to discrete time with the sampling interval given by `ue`, taking into account the intersample behavior of the input (`ue.InterSample`).

Examples

Simulate a given system `m0` (for example, created by `idpoly`).

```
e = iddata([],randn(500,1));
u = iddata([],idinput(500,'prbs'));
y = sim(m0,[u e]);
% iddata object with output y and input u.
z = [y u];
```

The same result is obtained by

```
u = iddata([],idinput(500,'prbs'));
y = sim(m0,u,'noise');
z = [ y u];
```

or

```
u = idinput(500,'prbs');
y = sim(m0,u,'noise');
z = iddata(y,u);
```

Validate a model by comparing a measured output `y` with one simulated using an estimated model `m`.

```
yh = sim(m,u);
plot(y,yh)
```

See Also

`compare`

`idmdlsim`

`pe`

`predict`

`simsd`

simsd

Purpose

Simulate models with uncertainty using Monte Carlo method

Syntax

```
simsd(m,u)
simsd(m,u,N,'noise',Ky)
[y,ysd] = simsd(m,u)
```

Description

`u` is an `iddata` object containing the inputs. `m` is a model given as any `idmodel` object. `N` random models are created according to the covariance information given in `m`. The responses of each of these models to the input `u` are computed and graphed in the same diagram. If the argument `'noise'` is included, noise is added to the simulation in accordance with the noise model of `m` and its own uncertainty. `Ky` denotes the output numbers to be plotted. (The default is `all`).

The default value is `N=10`.

With output arguments

```
[y,ysd] = simsd(m,u)
```

No plots are produced, but `y` is returned as a cell array with the simulated outputs, and `ysd` is the estimated standard deviation of `y`, based on the `N` different simulations. If `u` is an `iddata` object, so are the contents of the cells of `y` and `ysd`; otherwise, they are returned as vectors/matrices. In the `iddata` case,

```
plot(y{:})
```

thus plots all the responses.

`sim` and `simsd` have similar syntaxes. Note that `simsd` computes the standard deviation by Monte Carlo simulation, while `sim` uses differential approximations (the Gauss approximation formula). They might give different results.

Examples

Plot the step response of the model `m` and evaluate how it varies in view of the model's uncertainty.

```
step1 = [zeros(5,1); ones(20,1)];
```

```
  simsd(m,step1)
```

See Also

compare

idmdlsim

pe

predict

sim

Purpose Dimensions of `iddata`, `idmodel`, and `idfrd` objects

Syntax

```
d = size(m)
[ny,nu,Npar,Nx] = size(model)
[N, ny, nu, Nexp] = size(data)
ny = size(data,2)
ny = size(data,'ny')
size(model)
size(idfrd_object)
```

Description `size` describes the dimensions of `iddata`, `idmodel`, and `idfrd` objects.

iddata

For `iddata` objects, the sizes returned are, in this order,

- `N` = the length of the data record. For multiple-experiment data, `N` is a row vector with as many entries as there are experiments.
- `ny` = the number of output channels.
- `ny` = the number of input channels.
- `Ne` = the number of experiments.

To access just one of these sizes, use `size(data,k)` for the `k` `size(data,'N'),size(data,'ny')`, etc.

When called with one output argument, `d = size(data)` returns

- `d = [N ny nu]` if the number of experiments is `1`.
- `d = [sum(N) ny nu Ne]` if the number of experiments is `Ne > 1`.

idmodel

For `idmodel` objects the sizes returned are, in this order,

- `ny` = the number of output channels.
- `nu` = the number of input channels.

- `Npar` = the length of the `ParameterVector` (number of estimated parameters).
- `Nx` = the number of states for `idss` and `idgrey` models.

In this case the individual dimensions are obtained by `size(mod,2)`, `size(mod,'Npar')`, etc.

When `size` is called with one output argument, `d = size(mod)`, `d` is given by

```
    [ny nu Npar]
```

idfrd

For `idfrd` models, the sizes returned are, in this order,

- `ny` = the number of output channels.
- `nu` = the number of input channels.
- `Nf` = the number of frequencies.
- `Ns` = the number of spectrum channels.

In this case the individual dimensions are obtained by `size(mod,2)`, `size(mod,'Nf')`, etc.

When `size` is called with one output argument, `d = size(fre)`, `d` is given by

```
   [ny nu Nf Ns]
```

When `size` is called with no output arguments, in any of these cases, the information is displayed in the MATLAB Command Window.

Purpose

Estimate frequency response and spectrum using spectral analysis returning `idfrd` object

Syntax

```
g = spa(data)
g = spa(data,M,w,maxsize)
[g,phi,spe] = spa(data)
```

Description

`spa` estimates the transfer function `g` and the noise spectrum Φ_v of the general linear model

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

where $\Phi_v(\omega)$ is the spectrum of $v(t)$.

`data` contains the output-input data as an `iddata` object. The data can be complex valued. `data` can be both time domain and frequency domain. `data` can also be an `idfrd` object.

`g` is returned as an `idfrd` object (see `idfrd`) with the estimate of $G(e^{i\omega})$at the frequencies ω specified by row vector `w`. The default value of `w` is

```
w = [1:128]/128*pi/Ts
```

Here `Ts` is the sampling interval of `data`.

`g` also includes information about the spectrum estimate of $\Phi_v(\omega)$ at the same frequencies. Both outputs are returned with estimated covariances, included in `g`. See `idfrd`.

`M` is the length of the lag window used in the calculations. The default value is

```
M = min(30,length(data)/10)
```

Changing the value of `M` controls the frequency resolution of the estimate. The resolution corresponding to `M` is approximately π`/M` rad/sampling interval. The value of `M` exchanges bias for variance in the spectral estimate. As `M` is increased, the estimated functions show more detail, but are more corrupted by noise. The sharper peaks a true frequency function has, the higher `M` it needs. See `etfe` as an alternative for narrowband signals and systems. The function `spafdr` allows the frequency resolution to depend on the frequency. See also “Spectral Analysis Models” on page 5-31.

`maxsize` controls the memory-speed tradeoff (see `Algorithm Properties`).

For time series, where `data` contains no input channels, `g` is returned with the estimated output spectrum and its estimated standard deviation.

When `spa` is called with two or three output arguments,

- `g` is returned as an `idfrd` model with just the estimated frequency response from input to output and its uncertainty.
- `phi` is returned as an `idfrd` model, containing just the spectrum data for the output spectrum $\Phi_v(\omega)$ and its uncertainty.
- `spe` is returned as an `idfrd` model containing spectrum data for all output-input channels in `data`. That is, if `z = [data.OutputData, data.InputData]`, `spe` contains as spectrum data the matrix-valued power spectrum of `z`.

$$S = \sum_{m=-M}^{M} Ez(t+m)z(t)' \exp(-iWmT) win(m)$$

Here *win*(*m*) is weight at lag `m` of an M-size Hamming window and *W* is the frequency value i rad/s. Note that `'` denotes complex-conjugate transpose.

The normalization of the spectrum differs from the one used by `spectrum` in the Signal Processing Toolbox. See“Spectrum Normalization and the Sampling Interval” on page 5-40 for a more precise definition.

Examples

With default frequencies,

```
g = spa(z);
bode(g)
```

With logarithmically spaced frequencies,

```
w = logspace(-2,pi,128);
g= spa(z,[],w); % (empty matrix gives default)
bode(g,'sd',3)
bode(g('noise'),'sd',3) % noise spectrum with confidence
interval of 3 standard deviations.
```

Algorithm

The covariance function estimates are computed using `covf`. These are multiplied by a Hamming window of lag size `M` and then transformed using a Fourier transform. The relevant ratios and differences are then formed. For the default frequencies, this is done using a fast Fourier transform, which is more efficient than for user-defined frequencies. For multivariable systems, a straightforward `for` loop is used.

Note that `M` = γ is in Table 6.1 of Ljung (1999). The standard deviations are computed as on pages 184 and 188 in Ljung (1999).

See Also

`bode`

`etfe`

`ffplot`

`freqresp`

`idfrd`

`nyquist`

`spafdr`

Purpose

Estimate frequency response and spectrum using spectral analysis with frequency-dependent resolution returning `idfrd` object

Syntax

```
g = spafdr(data)
g = spafdr(data,Resol,w)
```

Description

`spafdr` estimates the transfer function `g` and the noise spectrum Φ_v of the general linear model

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

where $\Phi_v(\omega)$ is the spectrum of $v(t)$.

`data` contains the output-input data as an `iddata` object. The data can be complex valued, and either time or frequency domain. It can also be an `idfrd` object containing frequency-response data.

`g` is returned as an `idfrd` object (see `idfrd`) with the estimate of $G(e^{i\omega})$ at the frequencies ω specified by row vector `w`. `g` also includes information about the spectrum estimate of $\Phi_v(\omega)$ at the same frequencies. Both results are returned with estimated covariances, included in `g`. See `idfrd`. The normalization of the spectrum is the same as described under `spa`.

Frequencies

The frequency variable `w` is either specified as a row vector of frequencies, or as a cell array `{wmin,wmax}`. In the latter case the covered frequencies will be 50 logarithmically spaced points from `wmin` to `wmax`. You can change the number of points to `NP` by entering `{wmin,wmax,NP}`.

Omitting `w` or entering it as an empty matrix gives the default value, which is 100 logarithmically spaced frequencies between the smallest and largest frequency in data. For time-domain data, this means from `1/N*Ts` to `pi*Ts`, where `Ts` is the sampling interval of data and `N` is the number of data.

Resolution

The argument `Resol` defines the frequency resolution of the estimates. The resolution (measured in rad/s) is the size of the smallest detail in the frequency function and the spectrum that is resolved by the estimate. The resolution is a tradeoff between obtaining estimates with fine, reliable details, and suffering from spurious, random effects: The finer the resolution, the higher the variance in the estimate. `Resol` can be entered as a scalar (measured in rad/s), which defines the resolution over the whole frequency interval. It can also be entered as a row vector of the same length as `w`. Then `Resol(k)` is the local, frequency-dependent resolution around frequency `w(k)`.

The default value of `Resol`, obtained by omitting it or entering it as the empty matrix, is `Resol(k) = 2(w(k+1)-w(k))`, adjusted upwards, so that a reasonable estimate is guaranteed. In all cases, the resolution is returned in the variable `g.EstimationInfo.WindowSize`.

Algorithm

If the data is given in the time domain, it is first converted to the frequency domain. Then averages of `Y(w)Conj(U(w))` and `U(w)Conj(U(w))` are formed over the frequency ranges `w`, corresponding to the desired resolution around the frequency in question. The ratio of these averages is then formed for the frequency-function estimate, and corresponding expressions define the noise spectrum estimate.

See Also

`bode`

`etfe`

`ffplot`

`freqresp`

`idfrd`

`nyquist`

`spa`

Purpose

Convert `idmodel` objects of System Identification Toolbox to LTI models of Control System Toolbox

Syntax

```
sys = ss(mod)
sys = ss(mod,'m')
```

Description

`mod` is any `idmodel` object: `idgrey`, `idarx`, `idpoly`, `idproc`, `idss`, or `idmodel`.

`sys` is returned as an `ss` LTI model object. The noise input channels in `mod` are treated as follows: consider a model `mod` with both measured input channels u (nu channels) and noise channels e (ny channels) with covariance matrix Λ

$$y = Gu + HLv$$
$$\mathrm{cov}(v) = I$$

Both measured input channels u and normalized noise input channels v in `mod` are input channels in `sys`. The noise input channels belong to the `InputGroup 'Noise'`, while the others belong to the `InputGroup 'Measured'`. The names of the noise input channels are `v@yname`, where `yname` is the name of the corresponding output channel. This means that the LTI object realizes the transfer function [G HL].

To transform only the measured input channels in `sys`, use

```
sys = ss(mod('m')) or sys = ss(mod,'m')
```

This gives a representation of G only.

For a time series, (no measured input channels, nu = `0`), the LTI representations in `ss` contains the transfer functions from the normalized noise sources v to the outputs, that is, HL. If the model `mod` has both measured and noise inputs, `sys = ss(mod('n'))` gives a representation of the additive noise.

In addition, the normal subreferencing can be used.

```
sys = ss(mod(1,[3 4]))
```

If you want to describe [G H] or H (unnormalized noise), from e to y, first use

```
mod = noisecnv(mod)
```

to convert the noise channels e to regular input channels. These channels are assigned the names `e@yname`.

See Also

`frd`

`tf`

`zpk`

Purpose

Convert model to state-space form

Syntax

```
[A,B,C,D,K,X0] = ssdata(m)
[A,B,C,D,K,X0,dA,dB,dC,dD,dK,dX0] = ssdata(m)
```

Description

m is the model given as any idmodel object. A, B, C, D, K, and X0 are the matrices in the state-space description

$$\tilde{x}(t) = Ax(t) + Bu(t) + Ke(t)$$

$$x(0) = x0$$

$$y(t) = Cx(t) + Dx(t) + e(t)$$

where $\tilde{x}(t)$ is $\dot{x}(t)$ or $x(t+Ts)$ depending on whether m is a continuous-time or discrete-time model.

dA, dB, dC, dD, dK, and dX0 are the standard deviations of the state-space matrices.

If the underlying model itself is a state-space model, the matrices correspond to the same basis. If the underlying model is an input-output model, an observer canonical form representation is obtained.

For a time-series model (no measured input channels, u = [ ]), B and D are returned as the empty matrices.

Subreferencing models in the usual way (see idmodel properties) will give the state-space representation of the chosen channels. Notice in particular that

```
[A,B,C,D] = ssdata(m('m'))
```

gives the response from the measured inputs. This is a model without a disturbance description. Moreover,

```
[A,B,C,D,K] = ssdata(m('n'))
```

('n' as in “noise”) gives the disturbance description, that is, a time-series description of the additive noise with no measured inputs, so that `B = []` and `D = []`.

To obtain state-space descriptions that treat all input channels, both u and e, as measured inputs, first apply the conversion

```
m = noisecnv(m)
```

or

```
m = noisecnv(m,'norm')
```

where the latter case first normalizes e to v, where v has a unit covariance matrix. See the reference page for `noisecnv`.

Algorithm

The computation of the standard deviations in the input-output case assumes that an A polynomial is not used together with an F or D polynomial in the general polynomial equation (see “Definition of Polynomial Models” on page 5-43. For the computation of standard deviations in the case that the state-space parameters are complicated functions of the parameters, the Gauss approximation formula is used together with numerical derivatives. The step sizes for this differentiation are determined by `nuderst`.

See Also

`idmodel`

`idss`

`nuderst`

Purpose

Plot step response with confidence interval

Syntax

```
step(m)
step(data)
step(m,'sd',sd,Time)
step(data,'sd',sd,'PW',na,Time)
step(m1,m2,...,dat1, ...,mN,Time,'sd',sd)
step(m1,'PlotStyle1',m2,'PlotStyle2',...,dat1,'PlotStylek',...,mN,
'PlotStyleN',Time,'sd',sd)
[y,t,ysd] = step(m)
mod = step(data)
```

Description

`step` can be applied both to any `idmodel` or `idnlmodel` object and to `iddata` sets.

For a discrete-time `idmodel` `m`, the step response `y` and, when required, its estimated standard deviation `ysd`, are computed using `sim`. When called with output arguments, `y`, `ysd`, and the time vector `t` are returned. When `step` is called without output arguments, a plot of the step response is shown. If `sd` is given a value larger than zero, a confidence region around the response is drawn. It corresponds to the confidence of `sd` standard deviations. If the input argument list contains `'fill'`, this region is plotted as a filled area.

Setting the Time Interval

The start time `T1` and the end time `T2` can be specified by `Time = [T1 T2]`. If `T1` is not given, it is set to `-T2/4`. The negative time lags (the step is always assumed to occur at time `0`) show possible feedback effects in the data when the step is estimated directly from data. If `Time` is not specified, a default value is used.

Estimating the Step Response from data

For an `iddata` set data, `step(data)` estimates a high-order, noncausal FIR model after first having prefiltered the data so that the input is "as white as possible." The step response of this FIR model and, when asked for, its confidence region, are then plotted. Note that it might not be possible always to deliver the demanded time interval in this case, because of lack of excitation in the data. A warning is then issued. When called with an output argument, `step`, in the `iddata` case, returns this FIR model, stored as an `idarx` model. The order of the prewhitening filter can be specified as `na`. The default value is `na = 10`.

Several Models/Data Sets

Any number and any mixture of models and data sets can be used as input arguments. The responses are plotted with each input/output channel (as defined by the models and data sets `InputName` and `OutputName`) as a separate plot. Colors, line styles, and marks can be defined by `PlotStyle` values, as in

```
step(m1,'b-*',m2,'y--',m3,'g')
```

Noise Channels

The noise input channels in `m` are treated as follows: Consider a model `m` with both measured input channels u (nu channels) and noise channels e (ny channels) with covariance matrix Λ

$$y = Gu + He$$
$$\text{cov}(e) = \Lambda = LL'$$

where L is a lower triangular matrix. Note that `m.NoiseVariance =` Λ. The model can also be described with a unit variance, normalized noise source v:

$$y = Gu + HLv$$
$$\text{cov}(v) = I$$

- `step(m)` plots the step response of the transfer function G.
- `step(m('n'))` plots the step response of the transfer function H (ny inputs and ny outputs).The input channels have names `e@yname`, where `yname` is the name of the corresponding output.
- If `m` is a time series, that is, nu `= 0`, `step(m)` plots the step response of the transfer function H.

- `step(noisecnv(m))` plots the step response of the transfer function [*G H*] (*nu*+*ny* inputs and *ny* outputs). The noise input channels have names `e@yname`, where `yname` is the name of the corresponding output.
- `step(noisecnv(m,'norm'))` plots the step response of the transfer function [*G HL*] (*nu*+*ny* inputs and *ny* outputs). The noise input channels have names `v@yname`, where `yname` is the name of the corresponding output.

Arguments

If `step` is called with a single `idmodel` `m`, the output argument `y` is a 3-D array of dimension `Nt`-by-`ny`-by-`nu`. Here `Nt` is the length of the time vector `t`, `ny` is the number of output channels, and `nu` is the number of input channels. Thus `y(:,ky,ku)` is the response in the `ky`th output channel to a step in the `ku`th input channel. No plot is produced when output arguments are used.

`ysd` has the same dimensions as `y` and contains the standard deviations of `y`. This is normally computed using `sim`. However, when the model `m` contains an estimated delay (dead time) as in certain process models, the standard deviation is estimated with Monte Carlo techniques, using `simsd`.

If `step` is called with an output argument and a single data set in the input arguments, the output is returned as an `idarx` modelmod containing the high-order FIR model, and its uncertainty. By calling `step` with `mod`, the responses can be displayed and returned without your having to redo the estimation.

Examples

`step(data,'sd',3)` estimates and plots the step response

```
mod = step(data)
step(mod,'sd',3)
```

See Also

`cra`

`impulse`

Purpose

Generate model structure matrices

Syntax

```
NN = struc(NA,NB,NK)
```

Description

`struc` returns in `NN` the set of model structures composed of all combinations of the orders and delays given in row vectors `NA`, `NB`, and `NK`. The format of `NN` is consistent with the input format used by `arxstruc` and `ivstruc`. The command is intended for single-input systems only.

Examples

The statement

```
NN = struc(1:2,1:2,4:5);
```

produces

```
NN =
  1  1  4
  1  1  5
  1  2  4
  1  2  5
  2  1  4
  2  1  5
  2  2  5
```

See Also

`arxstruc`

`ivstruc`

`selstruc`

Purpose

Convert `idmodel` objects of System Identification Toolbox to transfer-function LTI models of Control System Toolbox

Syntax

```
sys = tf(mod)
sys = tf(mod,'m')
```

Description

`mod` is any `idmodel` object: `idgrey`, `idarx`, `idpoly`, `idproc`, `idss`, or `idmodel`.

`sys` is returned as a transfer function `tf` LTI model object. The noise input channels in `mod` are treated as follows:

Consider a model `mod` with both measured input channels *u* (*nu* channels) and noise channels *e* (*ny* channels) with covariance matrix Λ

$$y = Gu + He$$
$$\mathrm{cov}(e) = \Lambda = LL'$$

where *L* is a lower triangular matrix. `mod.NoiseVariance` = Λ. The model can also be described with a unit variance, normalized noise source *v*.

$$y = Gu + HLv$$
$$\mathrm{cov}(v) = I$$

Both measured input channels *u* and normalized noise input channels *v* in `mod` are input channels in `sys`. The noise input channels belongs to the `InputGroup 'Noise'`, while the others belong to the `InputGroup 'Measured'`. The names of the noise input channels will be `v@yname`, where `yname` is the name of the corresponding output channel. This means that the LTI object realizes the transfer function [*G HL*].

To transform only the measured input channels in `mod`, use

```
sys = tf(mod('m')) or sys = tf(mod,'m')
```

This gives a representation of *G* only.

For a time series, (no measured input channels, *nu* `= 0`), the LTI representation contains the transfer functions from the normalized noise sources *v* to the outputs, that is, *HL*. If the model `mod` has both measured and noise inputs, `sys = tf(mod('n'))` gives a representation of the additive noise.

In addition, you can use normal subreferencing.

```
sys = tf(mod(1,[3 4]))
```

If you want to describe [*G H*] or *H* (unnormalized noise), from *e* to *y*, first use

```
mod = noisecnv(mod)
```

to convert the noise channels *e* to regular input channels. These channels are assigned the names `e@yname`.

See Also

`frd`

`ss`

`zpk`

Purpose

Convert model to transfer-function form

Syntax

```
[num,den] = tfdata(m)
[num,den,sdnum,sdden] = tfdata(m)
[num,den,sdnum,sdden] = tfdata(m,'v')
```

Description

`m` is a model given as any `idmodel` object with `ny` output channels and `nu` input channels.

`num` is a cell array of dimension `ny`-by-`nu`. `num{ky,ku}` (note the curly brackets) contains the numerator of the transfer function from input `ku` to output `ky`. This numerator is a row vector whose interpretation is described below.

Similarly, `den` is an `ny`-by-`nu` cell array of the denominators.

`sdnum` and `sdden` have the same formats as `num` and `den`. They contain the standard deviations of the numerator and denominator coefficients.

If `m` is a SISO model, adding an extra input argument `'v'` (for vector) will return `num` and `den` as vectors rather than cell arrays.

The formats of `num` and `den` are the same as those used by the Signal Processing Toolbox and Control System Toolbox, both for continuous-time and discrete-time models.

The noise input channels in `m` are treated as follows: Consider a model `m` with both measured input channels *u* (*nu* channels) and noise channels *e* (*ny* channels) with covariance matrix Λ

$$y = Gu + He$$
$$\mathrm{cov}(e) = \Lambda = LL'$$

where L is a lower triangular matrix. Note that `m.NoiseVariance` = Λ. The model can also be described with a unit variance, normalized noise source v:

$$y = Gu + HLv$$
$$\mathrm{cov}(v) = I$$

- `tfdata(m)` returns the transfer function *G*.
- `tfdata(m('n'))` returns the transfer function *H* (*ny* inputs and *ny* outputs).
- If `m` is a time series, that is, *nu* `= 0`, `tfdata(m)` returns the transfer function *H*.
- `tfdata(noisecnv(m))` returns the transfer function [*G H*] (*nu*+*ny* inputs and *ny* outputs).
- `tfdata(noisecnv(m,'norm'))` returns the transfer function [*G HL*] (*nu*+*ny* inputs and *ny* outputs).

Examples

For a continuous-time model,

```
num = [1 2]
den = [1 3 0]
```

corresponds to the transfer function

$$G(s) = \frac{s+2}{s^2+3s}$$

For a discrete-time model,

```
num = [2 4 0]
den = [1 2 3 5]
```

corresponds to the transfer function

$$H(z) = \frac{2z^2+4z}{z^3+2z^2+3z+5}$$

which is the same as

$$H(q) = \frac{2q^{-1}+4q^{-2}}{1+2q^{-1}+3q^{-2}+5q^{-3}}$$

Note that for discrete-time models, `idpoly` and `polydata` have a different interpretation of the numerator vector, in case it does not have the same length as the denominator vector. To avoid confusion, fill out with zeros to make numerator and denominator vectors the same length. Do this with `tfdata`.

See Also

`idpoly`

`noisecnv`

Purpose

Return date and time when object was created or last modified

Syntax

```
timestamp(obj)
ts = timestamp(obj)
```

Description

`obj` is any `idmodel`, `iddata`, or `idfrd` object. `timestamp` returns or displays a string with information about when the object was created and last modified.

Purpose

Store binary-tree nonlinearity estimator for nonlinear ARX models

Syntax

```
t=treepartition('NumberOfUnits',N)
t=treepartition(Property1,Value1,...PropertyN,ValueN)
```

Description

`treepartition` is an object that stores the binary-tree nonlinear estimator for estimating nonlinear ARX models.

You can use the constructor to create the nonlinearity object, as follows:

`t=treepartition('NumberOfUnits',N)` creates a binary tree nonlinearity estimator object with `N` terms in the binary tree expansion. The tree has the number of leaves equal to the largest integer less than `N` of the form `2^n-1`.

`t=treepartition(Property1,Value1,...PropertyN,ValueN)` creates a binary tree nonlinearity estimator object specified by properties in "treepartition Properties" on page 12-322.

Use `evaluate(t,x)` to compute the value of the function defined by the `treepartition` object `t` at `x`. At this stage, an adaptive *pruning algorithm* is used to select an active partition `D_a(= D_a(x))` on the branch of tree partitions that contain `x`.

Remarks

Use `treepartition` to define a nonlinear function $y = F(x)$, where F is a piecewise-linear (affine) function of x, y is scalar, and x is a 1-*by*-`m` vector. F is a local linear mapping, where x-space partitioning is determined by a binary tree.

The binary-tree network function is based on the following function expansion:

$$F(x) = xL + [1, x]C_a + d$$

x belongs to the active partition D_a. D_k is a partition of x-space. L is 1-*by*-`m` vector.

C_k is a 1-*by*-`(m+1)` vector.

d is a scalar.

The active partition D_a is computed as an intersection of half-spaces by a binary tree, as follows:

1. Tree with `N` nodes and `J` levels is initialized.

2. Node at level `J` is a terminating leaf and a node at level `j<J` has two descendants at level `j+1`. The number of leaves in the tree is `N = 2^(J+1)-1`, which is determined by the `NumberOfUnits` property of the `treepartition` object.

3. Partition at node `r` is based on `[1,x]*B_r > 0` or <= 0 (move to left or right descendant), where `B_r` is chosen to improve the stability of least-square computation on the partitions at the descendant nodes.

4. Compute at each node `r` the coefficients `C_r` of best linear approximation of unknown regression function on `D_r` using penalized least-squares algorithm.

treepartition Properties

You can include property-value pairs in the constructor to specify the object.

After creating the object, you can use `get` or dot notation to access the object property values. For example:

```
% List all property values
get(t)
% Get value of NumberOfUnits property
t.NumberOfUnits
```

You can use dot notation to assign property values to the object. `set` is not supported for MCOS objects.

For example, the following two commands are equivalent:

```
t.NumberOfUnits=5
```

The `Options` property is a structure. To set the values of this structure, you can use the following syntax, for example:

```
X=struct('FinestCell',2,
         'Threshold',0.5,
         'Stabilizer',1e-5);
C.Options=X;
```

Property Name	Description
`NumberOfUnits`	Integer specifies the number of nodes in the tree. Default=`'auto'` select the number of nodes from the data using the pruning algorithm. For example: `treepartition('NumberOfUnits',5)`

Property Name	Description
`Parameters`	Structure containing the following fields: • `RegressorMean`: 1-*by*-`m` vector containing the means of `x` in estimation data, `r`. • `RegressorMinMax`: `m`-*by*-2 matrix containing the maximum and minimum estimation-data regressor values. • `OutputOffset`: scalar `d`. • `LinearCoef`: `m`-*by*-1 vector *L*. • `SampleLength`: Length of estimation data. • `NoiseVariance`: Estimated variance of the noise in estimation data. • `NonlinearParameters`: A structure containing the following tree parameters: - `TreeLevelPntr`: `N`-*by*-1 vector containing the levels `j` of each node. - `AncestorDescendantPntr`: `N`-*by*-3 matrix, such that the entry `(k,1)` is the ancestor of node `k`, and entries `(k,2)` and `(k,3)` are the left and right descendants, respectively. - `LocalizingVectors`: `N`-*by*-`(m+1)` matrix, such that the `r`th row is `B_r`. - `LocalParVector`: `N`-*by*-`(m+1)` matrix, such that the `k`th row is `C_k`. - `LocalCovMatrix`: `N`-*by*-`((m+1)m/2)` matrix such that the `k`th row is the covariance matrix of `C_k`. `C_k` is reshaped as a row vector.

Property Name	Description
`Options`	Structure containing the following fields that affect the initial model: • `FinestCell`: Integer or string specifying the minimum number of data points in the smallest partition. Default: `'auto'`, which computes the value from the data. • `Threshold`: Threshold parameter used by the adaptive pruning algorithm. Smaller threshold values correspond to shorter is the branch that is terminated by the active partition `D_a`. Higher threshold value results in a longer branch. Default: `1.0`. • `Stabilizer`: Penalty parameter of the penalized least-squares algorithm used to compute local parameter vectors `C_k`. Higher stabilizer value improves stability, but may deteriorate the accuracy of the least-square estimate. Default: `1e-6`.

Examples

Use `treepartition` to specify the nonlinear estimator in nonlinear ARX models. For example:

```
m=nlarx(Data,Orders,treepartition('num',5));
```

The following commands provide an example examples of using advanced `treepartition` options:

```
% Define the treepartition object
t=treepartition('num',100);
% Set the Threshold, which is a field
% in the Options structure
opt=t.options;
opt.Threshold=2;
t.options=opt;
% Estimate the nonlinear ARX model
```

```
m=nlarx(Data,Orders,t);
```

See Also `nlarx`

Purpose

Specify absence of nonlinearities for specific input or output channels in Hammerstein-Wiener models

Syntax

```
unit=unitgain
```

Description

`unit=unitgain` instantiates an object that specifies an identity mapping *F(x)=x* to exclude specific input and output channels from being affected by a nonlinearity in Hammerstein-Wiener models.

Use the `unitgain` object as an argument in the `nlhw` estimator to set the corresponding channel nonlinearity to unit gain.

For example, for a two-input and one-output model, to exclude the second input from being affected by a nonlinearity, you the following syntax:

```
m = nlhw(data,orders,['saturation''unitgain'],'deadzone')
```

In this case, the first input saturates and the output has an associated `deadzone` nonlinearity.

Remarks

Use the `unitgain` object to exclude specific input and output channels from being affected by a nonlinearity in Hammerstein-Wiener models.

`unitgain` is a linear function $y = F(x)$, where *F(x)=x*.

unitgain Properties

`unitgain` does not have properties.

Examples

For example, for a one-input and one-output model, to exclude the output from being affected by a nonlinearity, you the following syntax:

```
m = nlhw(Data,Orders,'saturation','unitgain')
```

In this case, the input saturates.

If nonlinearities are absent in input or output channels, you can replace `unitgain` with an empty matrix. For example, to specify a Wiener model with a sigmoid nonlinearity at the output and a unit gain at the input, use the following command:

```
m = nlhw(Data,Orders,[],'sigmoid');
```

See Also

`deadzone`

`nlhw`

`saturation`

`sigmoidnet`

Purpose

Plot model characteristics using LTI viewer in Control System Toolbox

Syntax

```
view(m)
view(m('n'))
view(m1,...,mN,Plottype)
view(m1,PlotStyle1,...,mN,PlotStyleN)
```

Description

`m` is the output-input data to be graphed, given as any `idfrd` or `idmodel` object. After appropriate model transformations, the LTI viewer of Control System Toolbox is invoked. This allows `bode`, `nyquist`, `impulse`, `step`, and `zero/poles` plots.

To compare several models `m1,...,mN`, use `view(m1,...,mN)`. With `PlotStyle`, the color, line style, and marker of each model can be specified.

```
view(m1,'y:*',m2,'b')
```

Adding `Plottype` as a last argument specifies the type of plot in which `view` is initialized. `Plottype` is any of `'impulse'`,`'step'`,`'bode'`,`'nyquist'`,`'nichols'`,`'sigma'`, or `'pzmap'`. It can also be given as a cell array containing any collection of these strings (up to 6) in which case a multiplot is shown.

`view` does not display confidence regions. For that, use `bode`, `nyquist`, `impulse`, `step`, and `pzmap`.

The noise input channels in `m` are treated as follows: Consider a model `m` with both measured input channels *u* (*nu* channels) and noise channels *e* (*ny* channels) with covariance matrix Λ

$$y = Gu + He$$
$$cov(e) = \Lambda = LL^T$$

where L is a lower triangular matrix. Note that `m.NoiseVariance =` Λ. The model can also be described with a unit variance, normalized noise source v:

$$y = Gu + HLv$$
$$cov(v) = I$$

- `view(m)` plots the characteristics of the transfer function *G*.
- `view(m('n'))` plots the characteristics of the transfer function *HL* (*ny* inputs and *ny* outputs). The input channels have names `v@yname`, where `yname` is the name of the corresponding output.
- If `m` is a time series, that is, *nu* `= 0`, `view(m)` plots the characteristics of the transfer function *HL*.
- `view(noisecnv(m))` plots the characteristics of the transfer function [*G H*] (*nu*+*ny* inputs and *ny* outputs). The noise input channels have names `e@yname`, where `yname` is the name of the corresponding output.
- `view(noisecnv(m,'norm'))` plots the characteristics of the transfer function [*G HL*] (*nu*+*ny* inputs and *ny* outputs). The noise input channels have names `v@yname`, where `yname` is the name of the corresponding output.

`view` does not give access to all of the features of `ltiview`. Use

```
ml = ss(m), ltiview(Plottype,ml,...)
```

to reach these options.

See Also

`bode`

`impulse`

`nyquist`

`pzmap`

`step`

Purpose

Store wavelet network nonlinearity estimator for nonlinear ARX and Hammerstein-Wiener models

Syntax

```
s=wavenet('NumberOfUnits',N)
s=wavenet(Property1,Value1,...PropertyN,ValueN)
```

Description

`wavenet` is an object that stores the wavelet network nonlinear estimator for estimating nonlinear ARX and Hammerstein-Wiener models.

You can use the constructor to create the nonlinearity object, as follows:

`s=wavenet('NumberOfUnits',N)` creates a wavelet nonlinearity estimator object with `N` terms in the wavelet expansion.

`s=wavenet(Property1,Value1,...PropertyN,ValueN)` creates a wavelet nonlinearity estimator object specified by properties in "wavenet Properties" on page 12-332.

Use `evaluate(s,x)` to compute the value of the function defined by the `wavenet` object `s` at `x`.

Remarks

Use `wavenet` to define a nonlinear function $y = F(x)$, where y is scalar and x is an `m`-dimensional row vector. The wavelet network function is based on the following function expansion:

$$\begin{aligned} F(x) = (x-r)PL &+ a_{s1} f\left(\left(b_{s1}(x-r)\right)Q - c_{s1}\right) + \ldots \\ &+ a_{sn} f\left(\left(b_{sn} s(x-r)\right)Q - c_{sn}\right) \\ &+ a_{w1} g\left(\left(b_{w1}(x-r)\right)Q - c_{w1}\right) + \ldots \\ &+ a_{wn} w g\left(\left(b_{wn} w(x-r)\right)Q - c_{wn}\right) + d \end{aligned}$$

where f is a scaling function and g is the wavelet function. P and Q are `m`-*by*-`p` and `m`-*by*-`q` projection matrices, respectively. The projection matrices P and Q are determined by principal component analysis of estimation data. Usually, `p=m`. If the components of x in the estimation data are linearly dependent, then `p<m`. The number of columns of Q, `q`,

corresponds to the number of components of `x` used in the scaling and wavelet function.

When used in a nonlinear ARX model, `q` is equal to the size of the `NonlinearRegressors` property of the `idnlarx` object. When used in a Hammerstein-Wiener model, `m=q=1` and Q is a scalar.

r is a 1-*by*-`m` vector and represents the mean value of the regressor vector computed from estimation data.

d, a_s, b_s, a_w, and b_w are scalars. Parameters with the s subscript are scaling parameters, and parameters with the w subscript are wavelet parameters.

L is `p`-*by*-1 vector.

c_s and c_w are 1-*by*-`q` vectors.

The scaling function f and the wavelet function `g` are both radial functions, as follows:

$$f(x) = e^{-0.5x'x}$$

$$g(x) = (\dim(x) - x'x)e^{-0.5x'x}$$

wavenet Properties

You can include property-value pairs in the constructor to specify the object.

After creating the object, you can use `get` or dot notation to access the object property values. For example:

```
% List all property values
get(w)
% Get value of NumberOfUnits property
w.NumberOfUnits
```

You can use dot notation to assign property values to the object. `set` is not supported for MCOS objects.

For example, the following two commands are equivalent:

```
w.NumberOfUnits=5
```

The `Options` property is a structure. Typically, the values of this structure are set by estimating a model with a `wavenet` nonlinearity. If you need to set the values of this structure, you can use the following syntax:

```
O=struct('FinestCell',2,
         'MinCells',10,
         'MaxCells',50,
         'MaxLevels',5,
         'DilationStep',1,
         'TranslationStep',2);
w.Options=O;
```

Property Name	**Description**
`NumberOfUnits`	Integer specifies the number of nonlinearity units in the expansion. Default=`'auto'`. For example: `wavenet('NumberOfUnits',5)`
`LinearTerm`	Can have the following values: • `'on'`—(Default) Estimates the vector L in the expansion. • `'off'`—Fixes the vector L to zero and omits the term $(x-r)PL$. For example: `wavenet(H,'LinearTerm','on')`

Property Name	Description
`Parameters`	Structure containing the parameters in the nonlinear expansion, as follows: • `RegressorMean`: 1-*by*-`m` vector containing the means of `x` in estimation data, `r`. • `NonLinearSubspace`: `m`-*by*-`q` matrix containing Q. • `LinearSubspace`: `m`-*by*-`p` matrix containing P. • `LinearCoef`: `p`-*by*-1 vector L. • `ScalingDilation`: `ns`-*by*-1 matrix containing the values `bs_k`. • `WaveletDilation`: `nw`-*by*-1 matrix containing the values `bw_k`. • `ScalingTranslation`: `ns`-*by*-`q` matrix containing the values `cs_k`. • `WaveletTranslation`: `nw`-*by*-`q` matrix containing the values `cw_k`. • `ScalingCoef`: `ns`-*by*-1 vector containing the values `as_k`. • `WaveletCoef`: `nw`-*by*-1 vector containing the values `aw_k`. • `OutputOffset`: scalar `d`.

Property Name	Description
`Options`	Structure containing the following fields that affect the initial model: • `FinestCell`: Integer or string specifying the minimum number of data points in the smallest cell. A *cell* is the area covered by the significantly nonzero portion of a wavelet. Default: `'auto'`, which computes the value from the data. • `MinCells`: Integer specifying the minimum number of cells in the partition. Default: `16`. • `MaxCells`: Integer specifying the maximum number of cells in the partition. Default: `128`. • `MaxLevels`: Integer specifying the maximum number of wavelet levels. Default: `10`. • `DilationStep`: Real scalar specifying the dilation step size. Default: `2`. • `TranslationStep`: Real scalar specifying the translation step size. Default: `1`.

Examples

Use `wavenet` to specify the nonlinear estimator in nonlinear ARX and Hammerstein-Wiener models. For example:

```
m=nlarx(Data,Orders,wavenet);
```

See Also

`nlarx`

`nlhw`

Purpose

Convert `idmodel` objects of System Identification Toolbox to state-space LTI models of Control System Toolbox

Syntax

```
sys = zpk(mod)
sys = zpk(mod,'m')
```

Description

`mod` is any `idmodel` object: `idgrey`, `idarx`, `idpoly`, `idproc`, `idss`, or `idmodel`.

`sys` is returned as a `zpk` LTI model object. The noise input channels in `mod` are treated as follows: consider a model `mod` with both measured input channels u (nu channels) and noise channels e (ny channels) with covariance matrix Λ

$$y = Gu + He$$
$$\mathrm{cov}(e) = \Lambda = LL'$$

where L is a lower triangular matrix. Note that `mod.NoiseVariance` = Λ. The model can also be described with a unit variance, normalized noise source v.

$$y = Gu + HLv$$
$$\mathrm{cov}(v) = I$$

Both measured input channels u and normalized noise input channels v in `mod` are input channels in `sys`. The noise input channels belongs to the `InputGroup 'Noise'`, while the others belong to the `InputGroup 'Measured'`. The names of the noise input channels are given by `v@yname`, where `yname` is the name of the corresponding output channel. This means that the LTI object realizes the transfer function [G HL].

To transform only the measured input channels in `sys`, use

```
sys = zpk(mod('m')) or sys = zpk(mod,'m')
```

This gives a representation of G only.

For a time series, (no measured input channels, nu = `0`), the LTI representation contains the transfer functions from the normalized noise sources *v* to the outputs, that is, *HL*. If the model `mod` has both measured and noise inputs, `sys = zpk(mod('n'))` gives a representation of the additive noise.

In addition, the normal subreferencing can be used.

```
sys = zpk(mod(1,[3 4]))
```

If you want to describe [*GH*] or *H* (unnormalized noise), from *e* to *y*, first use

```
mod = noisecnv(mod)
```

to convert the noise channels *e* to regular input channels. These channels are assigned have the names `e@yname`.

See Also

`frd`

`ss`

`tf`

Purpose

Compute zeros, poles, and gains of transfer-function models

Syntax

```
[z,p,k] = zpkdata(m)
[z,p,k,dz,dp,dk] = zpkdata(m)
[z,p,k,dz,dp,dk] = zpkdata(m,'v')
```

Description

`m` is a model given as any `idmodel` object with `ny` output channels and `nu` input channels.

`z` is a a cell array of dimension `ny`-by-`nu`. `z{ky,ku}` (note the curly brackets) contains the zeros of the transfer function from input `ku` to output `ky`. This is a column vector of possibly complex numbers.

Similarly, `p` is an `ny`-by-`nu` cell array containing the poles.

`k` is a `ny`-by-`nu` matrix whose `ky`-`ku` entry is the transfer function gain of the transfer function from input `ku` to output `ky`. Note that the transfer function gain is the value of the leading coefficient of the numerator when the leading coefficient of the denominator is normalized to 1. It thus differs from the static gain. The static gain can be retrieved as `Ks = freqresp(m,0)`.

`dz` contains the covariance matrices of the zeros in the following way: `dz` is a `ny`-by-`nu` cell array. `dz{ky,ku}` contains the covariance information about the zeros of the transfer function from `ku` to `ky`. It is a 3-D array of dimension 2-by-2-by-`Nz`, where `Nz` is the number of zeros. `dz{ky,ku}(:,:,kz)` is the covariance matrix of the zero `z{ky,ku}(kz)`, so that the 1-1 element is the variance of the real part, the 2-2 element is the variance of the imaginary part, and the 1-2 and 2-1 elements contain the covariance between the real and imaginary parts.

`dp` contains the covariance matrices of the poles in the same way.

`dk` is a matrix containing the variances of the elements of `k`.

If `m` is a SISO model, adding an extra input argument `'v'` (for vector) returns `z` and `p` as vectors rather than cell arrays.

Note that the zeros and the poles are associated with the different channel combinations. To obtain the so-called transmission zeros, use `tzero`.

The noise input channels in `m` are treated as follows: Consider a model `m` with both measured input channels *u* (*nu* channels) and noise channels *e* (*ny* channels) with covariance matrix Λ

$$y = Gu + He$$
$$\mathrm{cov}(e) = \Lambda = LL'$$

where *L* is a lower triangular matrix. Note that `m.NoiseVariance =` Λ. The model can also be described with a unit variance, normalized noise source *v*.

$$y = Gu + HLv$$
$$\mathrm{cov}(v) = I$$

Then,

- `zpkdata(m)` returns the zeros and poles of *G*.
- `zpkdata(m('n'))` returns the zeros and poles of *H* (*ny* inputs and *ny* outputs).
- If `m` is a time series, that is, *nu* `= 0`, `zpkdata(m)` returns the zeros and poles of *H*.
- `zpkdata(noisecnv(m))` returns the zeros and poles of the transfer function [*G H*] (*nu*+*ny* inputs and *ny* outputs).
- `zpkdata(noisecnv(m,'norm'))` returns the zeros and poles of the transfer function [*GHL*] (*nu*+*ny* inputs and *ny* outputs).

The procedure handles both models in continuous and discrete time.

Note that you cannot rely on information about zeros and poles at the origin and at infinity for discrete-time models. (This is a somewhat confusing issue anyway.)

Algorithm

The poles and zeros are computed using `ss2zp`. The covariance information is computed using the Gauss approximation formula, using the parameter covariance matrix contained in `m`. When the transfer function depends on the parameters, numerical differentiation is applied. The step sizes for the differentiation are determined in the M-file `nuderst`.

Index